Avoiding fallback in distributed systems

Jacob Gabrielson

Critical failures prevent a service from producing useful results. For example, in an ecommerce website, if a database query for product information fails, the website cannot display the product page successfully. Amazon services must handle the majority of critical failures in order to be reliable. There are four broad categories of strategies for handling critical failures:

- **Retry**: Perform the failed activity again, either immediately or after some delay.
- **Proactive retry**: Perform the activity multiple times in parallel, and make use of the first one to finish.
- **Failover**: Perform the activity again against a different copy of the endpoint, or, preferably, perform multiple, parallel copies of the activity to raise the odds of at least one of them succeeding.
- **Fallback**: Use a different mechanism to achieve the same result.

This article covers fallback strategies and why we almost never use them at Amazon. You might find this surprising. After all, engineers often use the real world as a starting point for their designs. And in the real world, fallback strategies must be planned in advance and used when necessary. Let's say an airport's display boards go out. A contingency plan (such as humans writing flight information on whiteboards) must be in place to handle this situation, because passengers still need to find their gates. But consider how awful the contingency plan is: the difficulty of reading the whiteboards, the difficulty of keeping them up-to-date, and the risk that humans will add incorrect information. The whiteboard fallback strategy is necessary but it’s riddled with problems.

In the world of distributed systems, fallback strategies are among the most difficult challenges to handle, especially for time-sensitive services. Compounding this difficulty is that bad fallback strategies can take a long time (even years) to leave repercussions, and the difference between a good strategy and a bad strategy is subtle. In this article, the focus will be on how fallback strategies can cause more problems than they fix. We’ll include examples of where fallback strategies have caused problems at Amazon. Finally, we’ll discuss alternatives to fallback that we use at Amazon.

Analyzing fallback strategies for services isn’t intuitive and their ripple effects are difficult to foresee in distributed systems, so let’s start by first looking at fallback strategies for a single-machine application.

Single-machine fallback

Consider the following C code snippet that illustrates a common pattern for handling memory allocation failures in many applications. This code allocates memory by using the malloc() function, and then copies an image buffer into that memory while performing some kind of transformation:

```
pixel_ranges = malloc(image_size); // allocates memory
if (pixel_ranges == NULL) {

  // On error, malloc returns NULL

  exit(1); }

for (i = 0; i < image_size; i++) {
  pixel_ranges[i] = xform(original_image[i]); }
```

The code doesn't gracefully recover from the case where malloc fails. In practice, calls to malloc fail rarely, so developers often ignore its failures in code. Why is this strategy so commonplace? The reasoning is that, on a single machine, if malloc fails, the machine is probably out of memory. So there are bigger problems than one malloc call failing—the machine might crash soon. And most of the time, on a single machine, that is sound reasoning. Many applications aren't critical enough to be worth the effort for solving such a thorny problem. But what if you did want to handle the error? Trying to do something useful in that situation is tricky. Let's say we implement a second method called malloc2 that allocates memory differently, and we call malloc2 if the default malloc implementation fails:

```
pixel_ranges = malloc(image_size);
if (pixel_ranges == NULL) {
  pixel_ranges = malloc2(image_size);
}
```

At first glance, this code looks like it could work, but there are problems with it, some less obvious than others. To begin with, fallback logic is **hard to test**. We could intercept the call to malloc and inject a failure, but that might not accurately simulate what would happen in the production environment. In production, if malloc fails, the machine is most likely out of memory or low on memory. How do you simulate those broader memory problems? Even if you could generate a low-memory environment to run the test in (say, in a Docker container), how would you time the low-memory condition to coincide with the execution of the malloc2 fallback code?

Another problem is that the **fallback itself could fail**. The previous fallback code doesn't handle malloc2 failures, so the program doesn't provide as much benefit as you might think. The fallback strategy possibly makes complete failure less likely but not impossible. At Amazon we have found that spending engineering resources on making the primary (non-fallback) code more reliable usually raises our odds of success more than investing in an infrequently used fallback strategy.

Furthermore, if availability is our highest priority, the **fallback strategy might not be worth the risk**. Why bother with malloc at all if malloc2 has a higher chance of succeeding? Logically, malloc2 must be making a trade-off in exchange for its higher availability. Maybe it allocates memory in higher-latency, but larger, SSD-based storage. But that raises the question, why is it okay for malloc2 to make this trade-off? Let's consider a potential sequence of events that might happen with this fallback strategy. First, the customer is using the application. Suddenly (because malloc failed), malloc2 kicks in and the application slows down. That's bad: Is it actually okay to be slower? And the problems don't stop there. Consider that the machine is most likely out of (or very low on) memory. The customer is now experiencing two problems (slower application and slower machine) instead of one. The side-effects of switching to malloc2 might even make the overall problem worse. For example, other subsystems might also be contending for the same SSD-based storage.

Fallback logic can also place **unpredictable load** on the system. Even simple common logic like writing an error message to a log with a stack trace is harmless on the surface, but if something changes suddenly to cause that error to occur at a high rate, a CPU-bound application might suddenly morph into an I/O-bound application. And if the disk wasn't provisioned to handle writing at that rate or to store that amount of data, it can slow down or crash the application.

Not only might the fallback strategy make the problem worse, this will likely occur as a **latent bug**.

It is easy to develop fallback strategies that rarely trigger in production. It might take years before even one customer's machine actually runs out of memory at just the right moment to trigger the specific line of code with the fallback to malloc2 shown earlier. If there is a bug in the fallback logic or some kind of side-effect that makes the overall problem worse, the engineers who wrote the code will likely have forgotten how it worked in the first place, and the code will be harder to fix. For a single-machine application, this may be an acceptable business trade-off, but in distributed systems the consequences are much more significant, as we'll discuss later.

All of these problems are thorny, but in our experience, they can often be safely ignored in single-machine applications. The most common solution is the one mentioned earlier: Just let memory allocation errors crash the application. The code that allocates memory shares fate with the rest of the machine, and it's quite likely that the rest of the machine is about to fail in this case. Even if it didn't share fate, the application would now be in a state that wasn't anticipated, and failing fast is a good strategy. The business trade-off is reasonable.

For critical single-machine applications that must work in case of memory allocation failures, one solution is to pre-allocate all heap memory on startup and never rely on malloc again, even under error conditions. Amazon has implemented this strategy multiple times; for example, in monitoring daemons that run on production servers and Amazon Elastic Compute Cloud (Amazon EC2) daemons that monitor customers' CPU bursts.

Distributed fallback

At Amazon, we don't let distributed systems, especially systems that were meant to respond in real time, make the same trade-offs as single-machine applications. One of our reasons is the lack of shared fate with the customer. We can assume that applications are running on the machine sitting in front of the customer. If the application runs out of memory, the customer probably doesn't expect it to keep running. Services don't run on the machine the customer is using directly, so the expectation is different. Beyond that, customers typically use services precisely because they're more available than running an application on a single server, so we need to make them so. In theory, this would lead us to implement fallback as a way to make the service more reliable. Unfortunately, distributed fallback has all the same problems, and more, when it comes to critical system failures.

Distributed fallback strategies are harder to test. Service fallback is more complicated than the single-machine application case, because multiple machines and downstream services play a part in the failures. The failure modes themselves, such as overload scenarios, are hard to replicate in a test, even if test orchestration across multiple machines is readily available. The combinatorics also increase the sheer number of cases to test, so you need more tests and they're much harder to set up.

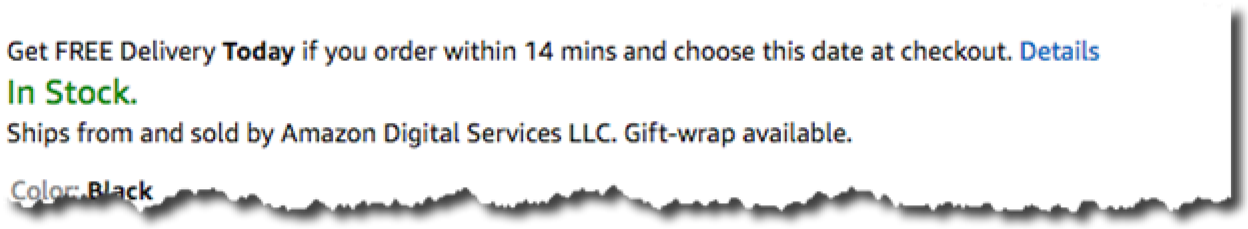
Distributed fallback strategies themselves can fail. While it might seem that fallback strategies guarantee success, in our experience, they usually improve only the odds of success.

Distributed fallback strategies often make the outage worse. In our experience, fallback strategies increase the scope of impact of failures as well as increasing recovery times.

Distributed fallback strategies are often not worth the risk. As with malloc2, the fallback strategy often makes some kind of trade-off; otherwise, we'd use it all the time. Why use a fallback that's worse, when something is already going wrong?

Distributed fallback strategies often have latent bugs that show up only when an unlikely set of coincidences occur, potentially months or years after their introduction.

A real-world major outage triggered by a fallback mechanism in the Amazon retail website illustrates all these problems. The outage occurred around 2001 and was caused by a new feature that provided up-to-date shipping speeds for all products shown on the website. The new feature looked something like this:



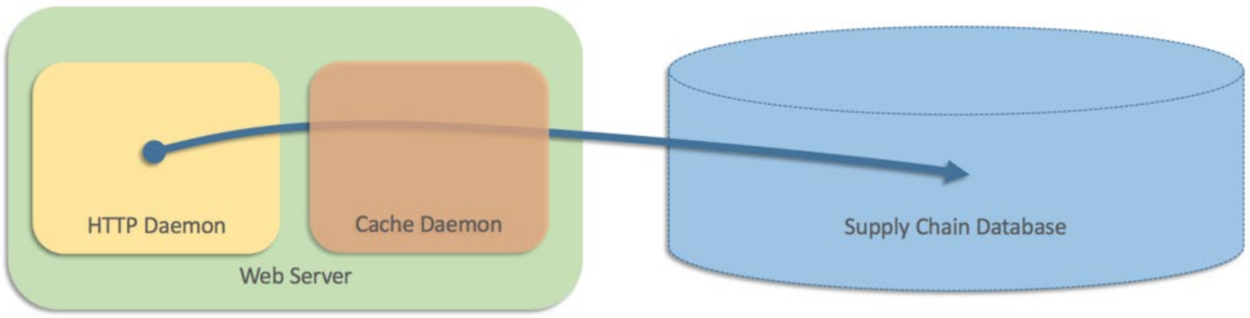
Get FREE Delivery **Today** if you order within 14 mins and choose this date at checkout. Details

In Stock.

Ships from and sold by Amazon Digital Services LLC. Gift-wrap available.

Color: **Black**

At the time, the website architecture had only two tiers, and since this data was stored in a supply chain database, the web servers needed to query the database directly. But the database couldn't keep up with the request volume from the website. The website had a high volume of traffic, and some pages would show 25 or more products, with shipping speeds for each product displayed inline. So we added a caching layer running as a separate process on each web server (somewhat like Memcached):



This worked well, but the team also attempted to handle the case where the cache (a separate process) failed for some reason. In this scenario, the web servers reverted to querying the database directly. In pseudo-code, we wrote something like this:

```
if (cache_healthy) {
  shipping_speed = get_speed_via_cache(sku);
} else {
  shipping_speed = get_speed_from_database(sku);
}
```

Falling back to direct database queries was an intuitive solution that did work for a number of months. But eventually the caches all failed around the same time, which meant that every web server hit the database directly. This created enough load to completely lock up the database. The entire website went down because all web server processes were blocked on the database. This supply chain database was also critical for fulfillment centers, so the outage spread even further, and all fulfillment centers worldwide ground to a halt until the problem was fixed.

All the problems we saw in the single-machine case were present in the distributed case with more dire consequences. It was **hard to test** the distributed fallback case; even if we had simulated the cache failure, we wouldn't have found the problem, which required failures across multiple machines in order to trigger. And in this case, the **fallback strategy itself amplified the problem** and was worse than no fallback strategy at all. The fallback turned a partial website outage (not being able to display shipping speeds) into a full-site outage (no pages loaded at all) and took down the entire Amazon fulfillment network in the back end.

The thinking behind our fallback strategy in this case was **illogical**. If hitting the database directly was more reliable than going through the cache, why bother with the cache in the first place? We were afraid that not using the cache would result in overloading the database, but why bother having the fallback code if it was potentially so harmful? We might have noticed our error early on, but the bug was a **latent** one, and the situation that caused the outage showed up months after launch.

How Amazon avoids fallback

Given these pitfalls we encountered in distributed fallback, we now almost always prefer alternatives to fallback. These are outlined here.

Improve the reliability of non-fallback cases

As mentioned previously, fallback strategies merely reduce the likelihood of complete failures. A service can be much more available if the main (non-fallback) code is made more robust. For example, instead of implementing fallback logic between two different data stores, a team could invest in using a database with higher inherent availability, such as Amazon DynamoDB. This strategy is frequently used successfully across Amazon. For example, this talk describes using DynamoDB to power amazon.com on Prime Day 2017.

Let the caller handle errors

One solution to critical system failures is not to fall back, but to let the calling system handle the failure (by retrying, for example). This is a preferred strategy for AWS services, where our CLIs and SDKs already have built-in retry logic. Where possible we prefer this strategy, especially in situations where enough effort has been put into sharing fate and reducing the likelihood of the main case failing (and the fallback logic would be highly unlikely to improve availability at all).

Push data proactively

Another tack we use to avoid needing to fall back is to reduce the number of moving parts when responding to requests. If, for example, a service needs data to fulfill a request, and that data is already present locally (it doesn’t need to be fetched), there is no need for a failover strategy. A successful example of this is in the implementation of AWS Identity and Access Management (IAM) roles for Amazon EC2. The IAM service needs to provide signed, rotated credentials to code running on EC2 instances. To avoid ever needing to fall back, the credentials are proactively pushed to every instance and remain valid for many hours. This means that IAM role-related requests keep working in the unlikely event of a disruption in the push mechanism.

Convert fallback into failover

One of the worst things about fallback is that it isn't exercised regularly and is likely to fail or increase the scope of impact when it triggers during an outage. The circumstances that trigger fallback might not naturally occur for months or even years! To address the problem of latent failures of the fallback strategy, it's important to exercise it regularly in production. A service must run both the fallback and the non-fallback logic continuously. It must not merely run the fallback case but also treat it as an equally valid source of data. For example, a service might randomly choose between the fallback and non-fallback responses (when it gets both back) to make sure they're both working. But at this point, the strategy can no longer be considered fallback and falls firmly into the category of failover.

Ensure that retries and timeouts don't become fallback

Retries and timeouts are discussed in the article Timeouts, Retries, and Backoff with Jitter. The article says that retries are a powerful mechanism for providing high availability in the face of **transient and random errors**. In other words, retries and timeouts provide insurance against occasional failures due to minor issues like spurious packet loss, uncorrelated single-machine failure, and the like. It’s easy to get retries and timeouts wrong, however. Services often go for months or longer without needing many retries, and these might finally kick in during scenarios that your team has never tested. For this reason, we maintain metrics that monitor overall retry rates and alarms that alert our teams if retries are happening frequently.

One other way to avoid retries turning into fallback is to execute them all the time with **proactive retry** (also known as hedging or parallel requests). This technique is inherently built into systems that perform quorum reads or writes, where a system might require an answer from two out of three servers in order to respond. Proactive retry follows the design pattern of **constant work**. Because the redundant requests are always being made, no extra load from retries is added to the system as the need for the redundant requests increases.

In conclusion

At Amazon, we avoid fallback in our systems because it’s difficult to prove and its effectiveness is hard to test. Fallback strategies introduce an operational mode that a system enters only in the most chaotic moments where things begin to break, and switching to this mode only increases the chaos. There is often a long delay between the time a fallback strategy is implemented and the time it’s encountered in a production environment.

Instead, we favor code paths that are exercised in production continuously rather than rarely. We focus on improving the availability of our primary systems, by using patterns like pushing data to systems that need it instead of pulling and risking failure of a remote call at a critical time. Finally, we watch out for subtle behavior in our code that could flip it into a fallback-like mode of operation, such as by performing too many retries.

If fallback is essential in a system, we exercise it as often as possible in production, so that fallback behaves just as reliably and predictably as the primary mode of operating.