# Circuit Breaker

You have building or refactoring an application to be cloud-Native. You are now running it as a collection of microservices as various types in a Microservices Architecture. As you have deployed continuous deployment pipelines, you are constantly upgrading your application which always introduces the risk of introducing little bugs. As argued in CI-CD Pipeline, this is good - issues with small, incremental releases are quickly uncovered and fixed. It does however present you with the problem of how to make clients of services that temporarily show issues handle these well. Apart from bugs, there are more reasons for hiccups in a microservices collective: small network outages, server restarts, etc.

**If a service breaks, how do you limit the failure scope as much as possible and not trigger a cascade?**

If service A depends on service B which in turn depends on service C, then a bug in service C could trigger reduced availability in service A. Without having this knowledge at the fingertips, such cascading outages often are hard to trace in a production system. Furthermore, often the dependency graph doesn't tell the whole story: it might very well be that B can deliver partial results to A when C is not available. It is therefore important to contain outages as much as possible, meaning that B needs to be able to detect changes in C's availability and still do its utmost best to deliver meaningful results (or at least meaningful error messages) back to A. A further issue of concern is that if C comes back after a restart, it comes back "cold" - cached data needs to be warmed up, and when a whole backlog of requests from its dependents storms in after it comes back, it may go down again. It is therefore important to protect against this as well.

Therefore,

**Build your services such that they can detect outages in their dependencies and then stop using services that have failed; coming back up slowly and gracefully is equally important**

Circuit breakers is a technique that resolves these issues. The term is, of course, borrowed from electrical cicruit breaks that disable an electrical connection rapidly when something goes wrong and protects other systems in the circuit from overloading.

In software terms, circuit breakers detect anomolous behavior when calling services. This may be services that don't respond (or respond too slow - the difference is normally not detectable) or respond with errors. Calls to services are made through a circuit breaker library which monitors performance of calls, registering bad responses. When the service that the client library monitors is deemed bad, further calls are not attempted for a certain period but immediately return an error. This disables any network traffic to the service giving it time to restart. Regularly, one call is let through to check whether the failed service is available again; when this happens, ideally the amount of calls let through is slowly ramped up in order to let the service warm up again before getting back to regular traffic levels.

There are a couple of important design decisions to be made around circuit breaker libraries: deciding when to enable the circuit breaker is an important parameter. Too fast, and applications will run in a degraded mode too often; too slow, and there may be a domino effect through the call chain. For example, a slow responding service may tie up a lot of resources in its dependents, and this can cascade further down the call chain, effectively grinding major parts of a microservices application to a halt. For every system, a sweet spot needs to be decided on based on actual observed behavior.

Then, there is the question what to do in a client when a circuit breaker signals it is triggered. In essence, this is identical to the case where a circuit breaker does not exist at all: what to do when a remote call to a service fails? Microservices applications need to be designed with failure in mind, and it is important to attempt to continue as much as possible.

Assuming that messaging systems (which rarely change) are more available than individual services, a good solution to completely bypass this problem is by decoupling services through asynchronous messaging and build up an Event-Driven Architecture.

Many Service Meshes implement the Circuit Breaker pattern. Netflix Hysterix was an early implementation of the Circuit Breaker pattern. Istio also implements the Circuit Breaker pattern.

This pattern was first called out by Michael Nygard in Release It!