Design for failure

Table of Contents

Traditionally we've tried to "avoid" failure by doing whatever we could to make services/components "not fail". We would deploy monolithic "appliances" (hardware or software) and if that appliance failed it was a significant event. As we start to move toward distributed systems, especially Microservices, we find that there are many more points of failure. Microservices are implemented as a collection of services that evolve independently and interact and react to their surrounding environments including the changing of user/customer behaviors, business strategy/direction, system behavior, and even other services. One premise of these types of complex-adaptive systems is the ability to withstand failures and unexpected faults.

To do this, you must design your services with failure in mind. There are a handful of practices that come to mind:

- maximize service availability
- self healing
- isolate the "blast radius" of any single failure
- continually prove your system has been designed for failure

Maximize service availability

We know services individually can fail or be removed from a cluster for any number of reasons. We'd like for clients to have many options for connecting to a service so we run many `replicas` of them as described in Elasticity and Resilience. When services fail, or are removed from the cluster, we also want to maximize the ability to failover from a client perspective. For this we rely on Service Discovery. With both of those concepts in place we can simply rely on the infrastructure to maximize the availability of our services.

Self healing

Another critical pattern of a Microservices environment is to enable self healing. If services fail, do we need manual intervention? Ideally the system would be able to identify faults and correct them. The simplest way to correct a faulty service is to just kill it and restart. Kubernetes provides this type of ability for us with Liveness and Health checks. When liveness or health checks run, and they find that a system is not in a healthy state, the service will be killed. Combined with ReplicaSets, Kubernetes will restore the service to maintain the desired number of `replicas` Fabric8 has tooling for enabling liveness and health checks by default when we deploy our services.

Isolate blast radius of failures

When services on which we depend (other Microservices, databases, message queues, caches, etc) start experiencing faults we need to be able to limit the extent of that damage so it doesn't cause cascading failures. The first step for doing this starts at the application level. Tools like Netflix Hystrix provide bulkheading and within Fabric8 we can use kubeflix project. Hystrix within the Kubeflix project allows us to:

- limit the number of callers affected by this failure
- shed load with circuit breakers
- limit the number of calls to a predefined set of threads that can withstand failures
- put a cap on how long a caller can assume the service is still working (timeouts on service calls). without these limits, latency can make calls think the service is still functioning fine and continue sending traffic potentially further overwhelming the service.
- visualize this in a dynamic environment where services will be coming and down, potentially alleviating or amplifying faults

From the domain perspective, we have to be able to degrade gracefully when these downstream components are faulting. We can limit the blast radius of a faulting component, but if we provide a service, how do we maintain our promise to deliver that service? Hystrix also allows us to use fallback methods and workflows to provide some level of (possibly degraded) service even in the event of failures.

Prove your system has been designed for failure

When you've designed your system with failure in mind and able to withstand faults, a useful technique is to continuously prove whether or not this is true. With fabric8, we have an out of the box chaos monkey that can go through your Kubernetes namespaces and randomly kill pods in any of your environments including production. If you've not designed your services to be able to withstand these types of faults, then you want to know with fast feedback. Chaos monkey can provide that feedback.