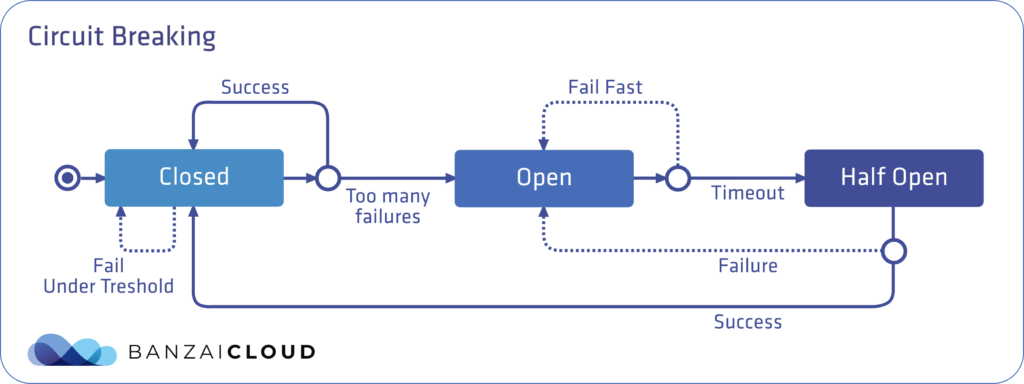
Istio Circuit Breaker

Downstream clients need to be protected from excessive slowness of upstream services.

Upstream services, in turn, must be protected from being overloaded by a backlog of requests.

The solution to this problem is the time-tested circuit breaker pattern.

A circuit breaker can have three states: **closed**, **open** and **half open**, and by default exists in a **closed** state. In the **closed** state, requests succeed or fail until the number of failures reach a predetermined threshold, with no interference from the breaker. When the threshold is reached, the circuit breaker *opens*. When calling a service in an **open** state, the circuit breaker *trips* the requests, which means that it returns an error without attempting to execute the call. In this way, by tripping the request downstream at the client, cascading failures can be prevented in a production system. After a configurable timeout, the circuit breaker enters a **half open** state, in which the failing service is given time to recover from its broken behavior. If requests continue to fail in this state, then the circuit breaker is opened again and keeps tripping requests. Otherwise, if the requests succeed in the **half open** state, then the circuit breaker will *close* and the service will be allowed to handle requests again.



To configure circuit breaking for a service in the Istio mesh, create a destination rule for that service.  
There are two fields under **TrafficPolicy** which are relevant to circuit breaking: ConnectionPoolSettings and OutlierDetection.

In **ConnectionPoolSettings**, the volume of connections can be configured for a service. **OutlierDetection** is for controlling the eviction of unhealthy services from the load balancing pool.

I.e. **ConnectionPoolSettings** controls the maximum number of requests, pending requests, retries or timeouts, while **OutlierDetection** controls the number of errors before a service is ejected from the connection pool, and is where minimum ejection duration and maximum ejection percentage can be set.

Let’s take a look at a **Destination Rule** with circuit breaking configured:

**apiVersion: networking.istio.io/v1alpha3**

**kind: DestinationRule**

**metadata:**

**name: notifications**

**spec:**

**host: notifications**

**trafficPolicy:**

**connectionPool:**

**tcp:**

**maxConnections: 1**

**http:**

**http1MaxPendingRequests: 1**

**maxRequestsPerConnection: 1**

**outlierDetection:**

**consecutiveErrors: 1**

**interval: 1s**

**baseEjectionTime: 3m**

**maxEjectionPercent: 100**

With these settings in the **ConnectionPoolSettings** field, only one connection can be made to the **notifications** service within a given time frame: one pending request with a maximum of one request per connection. If a threshold is reached, the circuit breaker will start tripping requests.

The **OutlierDetection** section is set so that it checks whether there is an error calling the service every second. If there is, the service is ejected from the load balancing pool for at least three minutes (the 100% maximum ejection percent indicates that all services can be ejected from the pool at the same time, if necessary).

There’s one thing which you need to pay special attention to when manually creating the **Destination Rule** resource, which is whether or not you have mutual TLS enabled for this service. If you do, you’ll also need to set the field below inside your **Destination Rule**, otherwise your caller services will probably receive 503 responses

**trafficPolicy:**

**tls:**

**mode: ISTIO\_MUTUAL**

Mutual TLS can be enabled [globally](https://istio.io/docs/tasks/security/authn-policy/#globally-enabling-istio-mutual-tls) for a specific [namespace](https://istio.io/docs/tasks/security/authn-policy/#namespace-wide-policy) or for a specific [service](https://istio.io/docs/tasks/security/authn-policy/#service-specific-policy), as well. You should be aware of these settings in order to determine whether you should set **trafficPolicy.tls.mode** to **ISTIO\_MUTUAL** or not. More importantly, it is very easy to forget to set this field when you are trying to configure a completely different feature (e.g. circuit breaking).

**Monitoring circuit breakers**

It is an absolute must that you monitor your services in a production environment, and that you are notified and be able to investigate when errors occur in the system. It stands to reason, then, that if you’ve configured a circuit breaker for your service, you’ll want to know when that breaker is tripped; what percentage of your requests were tripped by the circuit breaker; how many requests were tripped and when, and from which downstream client? If you can answer these questions, you can determine how well your circuit breaker is working, fine tune the circuit breaker configurations as needed, or optimize your service to handle additional concurrent requests.

Let’s see how to determine the trips caused by the circuit breaker in Istio:

The response code in the event of a circuit breaker trip is 503, so you won’t be able to differentiate it from other 503 errors based merely on that response. In Envoy, there is a counter called **upstream\_rq\_pending\_overflow**, which is the *total number of requests that overflowed the connection pool circuit breaker and were failed*. If you dig into Envoy’s statistics for your service, you can acquire this information, but it’s not particularly easy to reach.

Envoy also returns [response flags](https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/access_log#config-access-log-format-response-flags) in addition to response codes, and there exists a dedicated response flag to indicate circuit breaker trips: UO. This wouldn’t be particularly helpful if this flag could only be obtained through Envoy logs, but, fortunately, it was [implemented in Istio](https://github.com/istio/istio/pull/9945), so that response flags that are available in Istio metrics and can be fetched by Prometheus.

Circuit breaker trips can be queried like this:

sum(istio\_requests\_total{response\_code=&quot;503&quot;, response\_flags=&quot;UO&quot;}) by (source\_workload, destination\_workload, response\_code)