**Circuit Breaker Design Pattern**

Circuit Breaker is a self healing design pattern, used to build fault tolerant systems, where in a call to a backend service is proxied through an object called Circuit Breaker.

**The Problem**

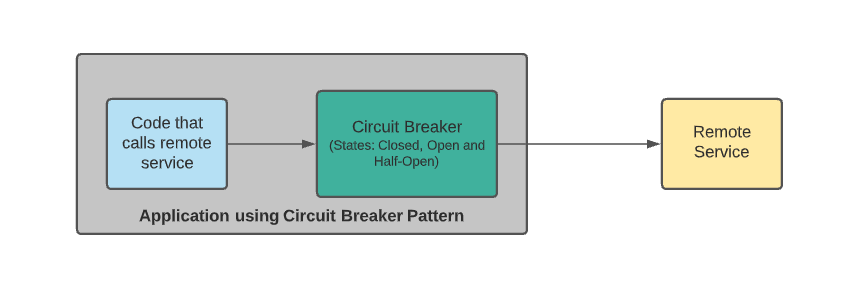
In a distributed environment calls to a remote service can fail due to various reasons like

* Network issues
* Remote service unavailable
* Slow network connections
* Time outs etc.

Sometimes these failures are very transient in nature or very call specific like a corrupt network packet and we can get around such failures by simply retrying the call to the remote resource. However, there are also situations when the failure is due to a much bigger problem and may linger around a bit in which case it is futile to do a retry operation. These failures can exhaust critical resources like memory, threads etc. and severely slow down your application which in turn can have a cascading impact on the availability of other systems in the enterprise. We have seen this happen time and again in ABC where the root cause of a RRT and degradation of many systems is because of a failure in a single application. We need automated ways where a distributed application is able to self-heal itself when such problems occur with minimal manual intervention.

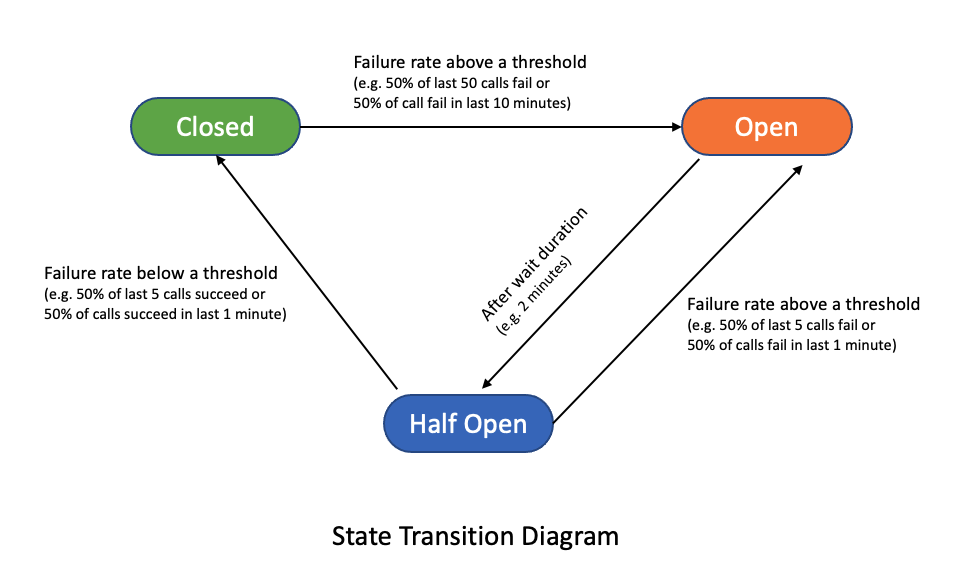
**The solution (Circuit Breaker Pattern)**

Circuit breaker is a self-healing design pattern popularized by Michael Nygard in his book "Release It!". As per this pattern a program or a library named circuit breaker (as shown in the picture below) is used to proxy all calls to a remote service that might fail.



The circuit breaker is implemented as a finite state machine and all calls to the remote service are controlled by the definition of these states. The circuit breaker has three states and it transitions from one state to another depending upon the health of the circuit (network connection and the remote service itself). In that sense the pattern has a close resemblance to a typical electrical circuit breaker. The three states of the circuit breaker are

1. Closed: Remote service is healthy and all calls to it are happening as normal.
2. Open: Remote service is having issues and instead of allowing any calls, the circuit breaker instantly responds with a failure/exception.
3. Half Open: Certain probing calls to the remote service are allowed to monitor the health of the circuit.



**Circuit Breaker Configuration**

A failed call can be defined as

* A call that did not go through and encountered an exception. You can choose which exception is counted as a failure or ignore an exception depending upon the framework used.
* A call that was deemed slow as per some predefined criteria.

Failure threshold can be defined using

* A time window like 50% failure rate in last 10 minute
* A count window like 50% failure rate in last 50 calls.

**Advantages of circuit breaker pattern**

There are significant advantages in preventing an application from repeatedly making a remote service call which is most likely to fail.

* Prevents critical system resources like memory, threads
* Prevents cascading failures due to errors and slowness
* Helps build fault tolerant and resilient systems
* It allows us to handle failures gracefully.

**Resilience4j (Library for implementing circuit breaker pattern)**

Resilience4j is a lightweight fault tolerance library inspired by Netflix Hystrix, but designed for functional programming. (Netflix Hystrix is officially in maintenance mode and should not be considered for any new implementations and existing hystrix implementations should consider migrating to Resilience4j). Resilience4j has many other modules apart from circuit breaker namely TimeLimiter, Threadpool Bulkhead, Semaphore Bulkhead, Retry, and RateLimiter but this article is focused only on circuit breaker. As compared to Hystrix, Resilience4j

* is light weight in nature
* has flexible configuration options and no boiler plate code required
* has support for functional programming and reactive programming
* has spring annotations support
* can combine all resiliency 4j modules in one call using annotations or functional programming
* built in micrometer support and can easily integrate with other tools
* has special states where resilency4j will always allow calls or always deny calls

**How to use Resilience4j in applications:**

Let's say our application is calling "rewards-service" for getting the rewards information of the user. We will configure Resilience4j for proxying all calls to "rewards-service". Resilience4j will short circuit any calls to "rewards-service" in case it is not working properly and instead call a fallback method.

**Add the following compile time dependencies**

compile "io.github.resilience4j:resilience4j-spring-boot2:${resilience4jVersion}"

This module expects following dependencies as well

compile 'org.springframework.boot:spring-boot-starter-actuator'

compile 'org.springframework.boot:spring-boot-starter-aop'

**Configuring Resilience4j using application.yaml**

resilience4j:

circuitbreaker:

instances:

rewards-service:

#configures the type of the sliding window. Possible values are COUNT\_BASED OR TIME\_BASED

slidingWindowType: COUNT\_BASED

#Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.

slidingWindowSize: 20

#configures the minimum no of calls with in a sliding window before failure rate can be calculated

minimumNumberOfCalls: 20

#circuit will be short-circuited in case the failure rate reaches 50%

failureRateThreshold: 50

#configures the time in ms after which circuit breaker will transition to HALF\_OPEN state

waitDurationInOpenState: 120000

#probing calls allowed while in open state to determine the future state of circuit breaker.

permittedNumberOfCallsInHalfOpenState: 3

#circuit will also be short circuited in case slow calls reach 50%

slowCallRateThreshold: 50

#configures the duration after which a call is deemed slow.

slowCallDurationThreshold: 3000

Resilience4j also provides default configuration if these values are left unconfigured. It is also possible to configure purely in java code (<https://resilience4j.readme.io/docs/examples>)

**Use circuit breaker object in rewards-service call**

Simply use the @CircuitBreaker annotation to configure any method to be used with the circuit breaker pattern.

//name of the circuit breaker is same as the one used in application.yaml

//This method will not be called when the circuit breaker is in open state

@CircuitBreaker(name = "rewards-service", fallbackMethod = "rewardsServiceFallBack")

public Rewards getRewardsInfo (){

//call to rewards service happens here.

//By default circuit breaker considers any exception as a failure

//Use ignoreExceptions configuration to ignore certain exceptions

}

//This method will be called whenever the circuit is in open state.

private Rewards rewardsServiceFallBack(Throwable t)

{

//any fall back options like calling RewardsLiteService goes here.

}

Fallback method should be placed in same class and it should have the same method signature as main method with one extra parameter (target exception parameter). If we have multiple fallback methods it will execute nearest fallback method that matches the exception. When circuit breaker is in OPEN state it throws CallNotPermittedException.

**Special States**

Resilience4j has additional two special states

* DISABLED - Always allow calls
* FORCED\_OPEN - always deny calls

**Resilience4j Actuator Integration**

Resilience4j provides some actuator endpoints that can be used for variety of purposes.

1. We can use actuator endpoints to force change the states curl -X POST [http://localhost:port/actuator/circuitbreakers/{circuitbreakername}](http://localhost:port/actuator/circuitbreakers/%7Bcircuitbreakername%7D) -H 'Content-Type: application/json' -d '{"updateState": "CLOSE"}'
2. Metrics endpoint CircuitBreaker Metrics are published to Metrics endpoint at /actuator/metrics <https://resilience4j.readme.io/docs/getting-started-3#metrics-endpoint>
3. Health End point Circuit breaker health can be monitored using circuit breaker health indicator which is disabled by default. <https://resilience4j.readme.io/docs/getting-started-3#health-endpoint>
4. Events endpoint Circuit breaker will emit events during its life cycle, you can monitor them at /actuator/circuitbreakerevents end point <https://resilience4j.readme.io/docs/getting-started-3#events-endpoint>

**References:**

* <https://resilience4j.readme.io/docs/getting-started>
* <https://resilience4j.readme.io/docs/micrometer>
* <https://resilience4j.readme.io/docs/examples-1>

application.yml

|  |
| --- |
| dataSource: |
|  | properties: |
|  | name: acde-db-dev |
|  | minPoolSize: 1 |
|  | maxPoolSize: 10 |
|  | maxWaitTime: 20000 |
|  | jdbcDriver: oracle.jdbc.driver.OracleDriver |
|  | pwdParam: password |
|  | logging: |
|  | level: |
|  | com.dfs.edt.acde: INFO |
|  | jwt2: |
|  | provider: enable |
|  | client: enable |
|  | swagger: |
|  | package: com.dfs.edt.acde.controller |
|  | title: ACDE Decision Service API |
|  | contact: DC-EDTSupport |
|  | email: dcedtsupport@ABC.com |
|  | version: 1.0.0 |
|  | description: "This API authorizes high risk account center transactions. It is invoked during high risk transactions post |
|  | login and during login.This API invokes blaze rules to make decision on the transaction - whether to allow or deny." |
|  |  |
|  |  |
|  | service: |
|  | context-root: /enterprise/acDecisionEngine/v1/\* |
|  |  |
|  | ws: |
|  | dsName: acde |
|  |  |
|  | neustar: |
|  | baseUrl: https://enterprisedpneustarvendordataapi-pa.cf-ssb-z3-dev.ABCfinancial.com |
|  | uri: /enterprise/decisionplatform/vendordata/neustar/v1/data |
|  | readTimeout: 3000 |
|  | connectTimeout: 3000 |
|  | maxRetryAttempts: 2 |
|  | fixedDelay: 200 |
|  |  |
|  | resilience4j: |
|  | circuitbreaker: |
|  | configs: |
|  | default: |
|  | slidingWindowSize: ${CB\_slidingWindowSize} |
|  | slidingWindowType: COUNT\_BASED |
|  | failureRateThreshold: ${CB\_failureRateThreshold} |
|  | waitDurationInOpenState: ${CB\_waitDurationInOpenState} |
|  | minimumNumberOfCalls: ${CB\_minimumNumberOfCalls} |
|  | permittedNumberOfCallsInHalfOpenState: ${CB\_permittedNumberOfCallsInHalfOpenState} |
|  | slowCallRateThreshold: ${CB\_slowCallRateThreshold} |
|  | slowCallDurationThreshold: ${CB\_slowCallDurationThreshold} |
|  | instances: |
|  | neustar: |
|  | baseConfig: default |
|  | retry: |
|  | instances: |
|  | neustar-retry: |
|  | maxRetryAttempts: 2 |
|  | waitDuration: 100 |
|  | retryExceptions: |
|  | - com.dfs.edt.acde.exception.ACDERetryableException |