<https://resilience4j.readme.io/docs/circuitbreaker>

Circuit Breaker I n Microservices :

Software systems make remote calls to software running in different processes, usually on different machines across a network. One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached. What's worse, if you have many callers on an unresponsive supplier, you can run out of critical resources leading to cascading failures across multiple systems.

Solution

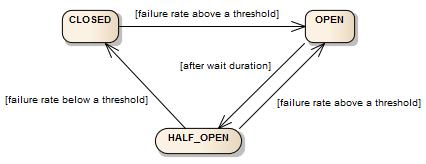
The circuit breaker pattern is the solution to this problem. The basic idea behind the circuit breaker is very simple. You wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error or with some alternative service or default message, without the protected call being made at all. This will make sure system is responsive and threads are not waiting for an unresponsive call.

**Different States of the Circuit Breaker**

The circuit breaker has three distinct states: Closed, Open, and Half-Open:

* Closed – When everything is normal, the circuit breaker remains in the closed state and all calls pass through to the services. When the number of failures exceeds a predetermined threshold the breaker trips, and it goes into the Open state.
* Open – The circuit breaker returns an error for calls without executing the function.
* Half-Open – After a timeout period, the circuit switches to a half-open state to test if the underlying problem still exists. If a single call fails in this half-open state, the breaker is once again tripped. If it succeeds, the circuit breaker resets back to the normal, closed state. Different States of the Circuit Breaker

The CircuitBreaker is implemented via a finite state machine with three normal states: CLOSED, OPEN and HALF\_OPEN and two special states DISABLED and FORCED\_OPEN.



**Create and configure a CircuitBreaker :**

**Add below things in application.yml :**

**resilience4j.circuitbreaker:**

instances:

service1:

registerHealthIndicator: true

ringBufferSizeInClosedState: 5

ringBufferSizeInHalfOpenState: 3

waitDurationInOpenState: 10s

failureRateThreshold: 50

recordExceptions:

- org.springframework.web.client.HttpServerErrorException

- java.io.IOException

- java.util.concurrent.TimeoutException

- org.springframework.web.client.ResourceAccessException

ignoreExceptions:

- com.gl.orderManagementApp.service.MyException

service2:

registerHealthIndicator: true

ringBufferSizeInClosedState: 6

ringBufferSizeInHalfOpenState: 4

waitDurationInOpenState: 20s

failureRateThreshold: 60

**failureRateThreshold** : Configures the failure rate threshold in percentage.

When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.

**permittedNumberOfCalls InHalfOpenState :** Configures the number of permitted calls when the CircuitBreaker is half open.

**maxWaitDurationInHalfOpenState :** Configures a maximum wait duration which controls the longest amount of time a CircuitBreaker could stay in Half Open state, before it switches to open.

Value 0 means Circuit Breaker would wait infinitely in HalfOpen State until all permitted calls have been completed.

**waitDurationInOpenStatewaitDurationInOpenState** : The time that the CircuitBreaker should wait before transitioning from open to half-open.

**recordExceptions** : A list of exceptions that are recorded as a failure and thus increase the failure rate.

Any exception matching or inheriting from one of the list counts as a failure, unless explicitly ignored via ignoreExceptions.

If you specify a list of exceptions, all other exceptions count as a success, unless they are explicitly ignored by ignoreExceptions.

**ignoreExceptions** : A list of exceptions that are ignored and neither count as a failure nor success.

Any exception matching or inheriting from one of the list will not count as a failure nor success, even if the exceptions is part of recordExceptions.

You can check orderManagementApp under resileince4j project of GreenLearner (old and new eclipse workspace)

@CircuitBreaker(name = "service1", fallbackMethod = "fallbackForRegisterSeller")

RateLimiter :

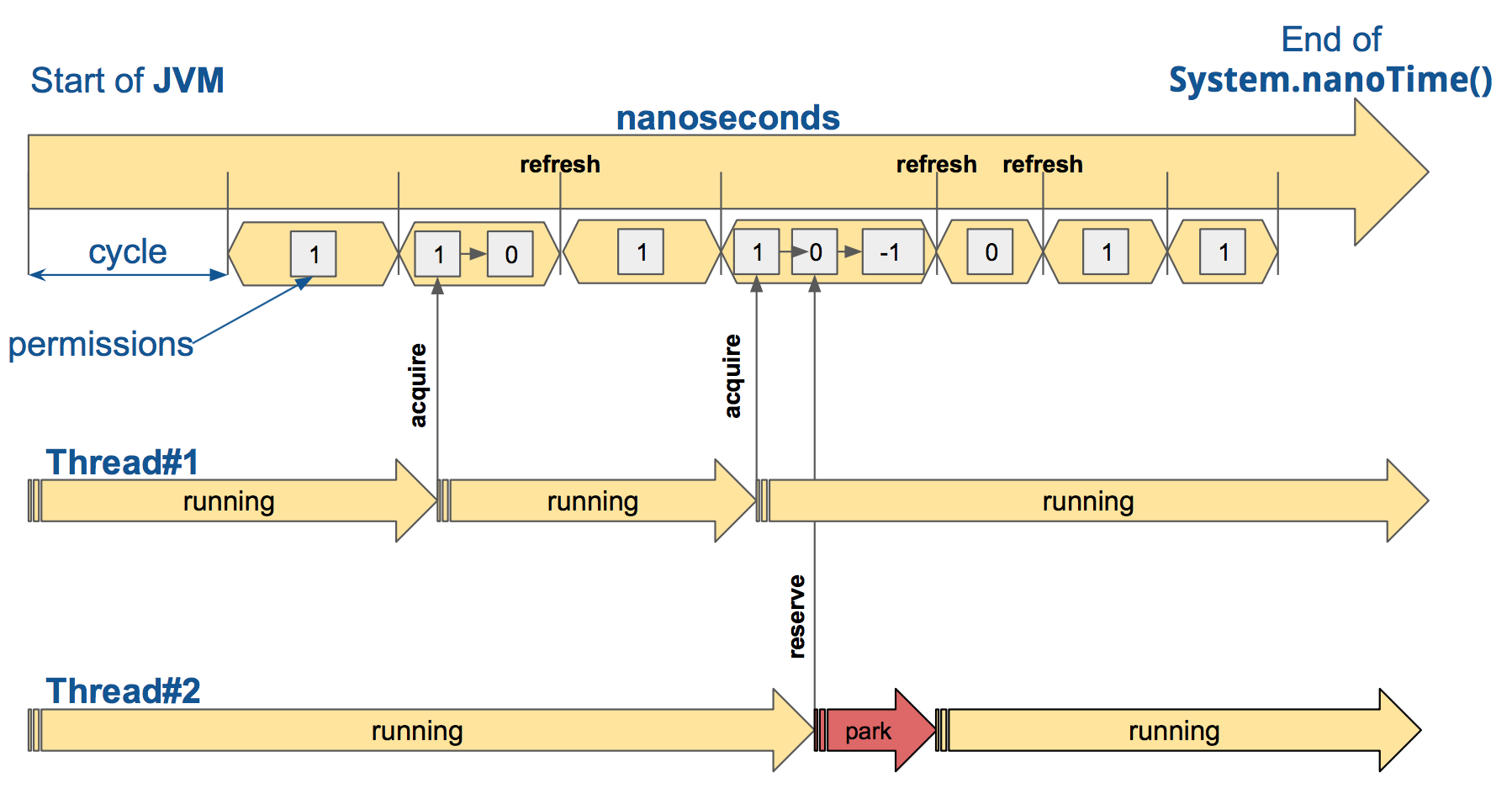
<https://www.vinsguru.com/rate-limiter-pattern/#:~:text=Rate%20Limiter%20helps%20to%20protect,target%20server%20is%20failing%20%2F%20unresponsive>.

Rate Limiter Pattern helps us to make our services highly available just by limiting the number of calls we could make/process in a specific window. In other words, It helps us to control the throughput. When we receive too many requests, the Service might simply reject the call. The client has to retry at a later time or can go with some default/cached values.

**Rate Limiter vs Circuit Breaker:**

Rate Limiter helps to protect the server from over loading by controlling throughput.

Circuit Breaker helps to keep the client safe and functional when the target server is failing / unresponsive.



timeoutDuration : The default wait time a thread waits for a permission

limitRefreshPeriod : The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value

limitForPeriod : The number of permissions available during one limit refresh period

**@RateLimiter(name = "service1", fallbackMethod = "rateLimiterfallback")**

In application.properties

resilience4j.ratelimiter:

instances:

service1:

limitForPeriod: 10

limitRefreshPeriod: 100000

timeoutDuration: 1000ms

for testing

limitForPeriod: 5

limitRefreshPeriod: 60s

timeoutDuration: 0

First 5 calls every 60 seconds will get success response

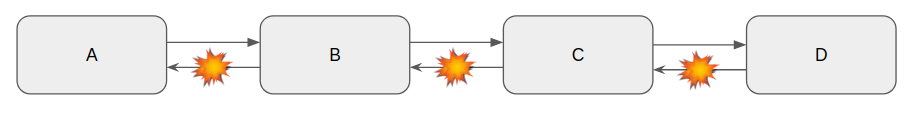
After 5 calls to ordermanagement app next request failed.

Next few requests will get failure / fallback response.

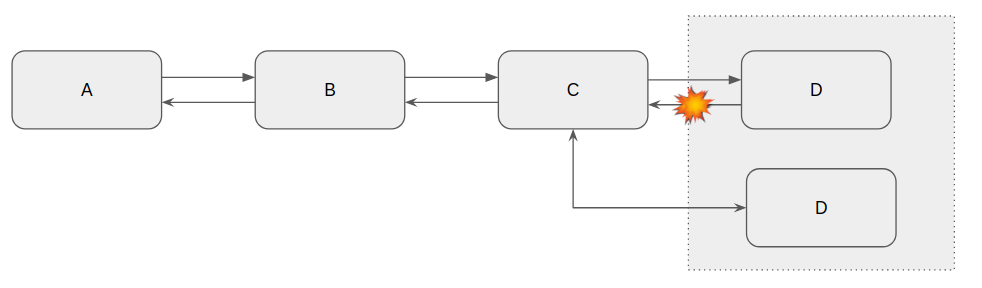
Retry Pattern :

<https://www.vinsguru.com/retry-pattern/>

In Microservice architecture, when there are multiple services (A, B, C & D), one service (A) might depend on the other service (B) which in turn might depend on C and so on. Sometimes due to some issue, Service D might not respond as expected. Service D might have thrown some exception like OutOfMemory Error or Internal Server Error. Such exceptions are cascaded to the downstream services which might result in poor user experience as shown below



Sometimes when google.com does not work for us, we just do not give up. We simply refresh the page once assuming things will work next time and it does most of the times. Intermittent network issues are very common. In the Microservices world, we might be running multiple instances of same Service D for high availability and load balancing. If one of the instances could be having the issue and it does not respond properly to our request, If we retry the request, the load balancer could send the request to a healthy node and get the response properly. So with Retry option, we have more chance for getting the proper response



resilience4j.retry:

instances:

retryService1:

maxRetryAttempts: 3

waitDuration: 10000

maxAttempts : The maximum number of attempts (including the initial call as the first attempt)

waitDuration : A fixed wait duration between retry attempts

failAfterMaxAttempts :

A boolean to enable or disable throwing of MaxRetriesExceededException when the Retry has reached the configured maxAttempts, and the result is still not passing the retryOnResultPredicate

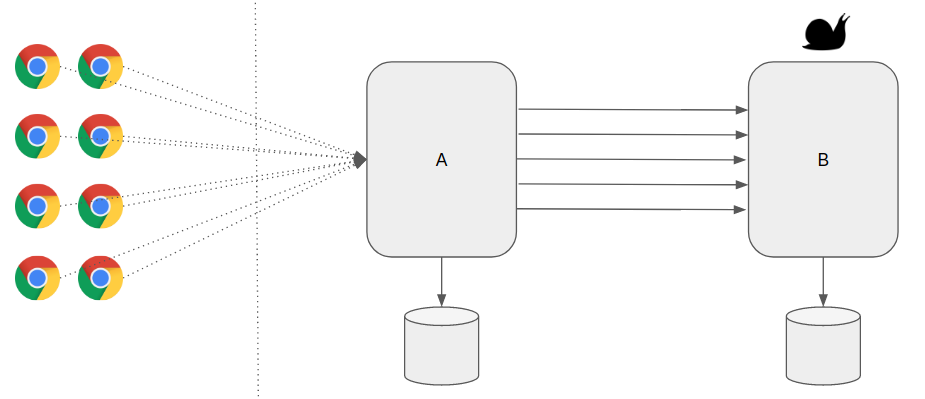
BulkHead Pattern :

<https://www.vinsguru.com/bulkhead-pattern/>

Lets assume that there are 2 services A and B. Service A has very limited resources (say 5 threads). It can process only 5 concurrent requests with its available threads. Service A has 2 sets of APIs as shown below

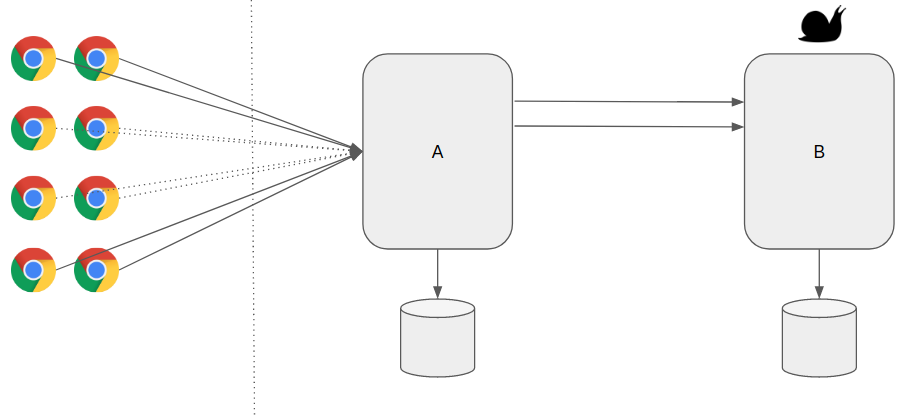
/a/b – this depends on Service B which is slow sometimes.

/a – this does not depend on Service B



When there are multiple concurrent requests to Service A, say 10, 5 of them are for endpoint /a/b and 5 of them are for endpoint /a, there is a chance that Service A might use all its threads to work on the requests for /a/b and block all the 5 threads. Even though the remaining requests are for /a which does not have any other service dependency, Service A does not have free threads to work on the requests (resource exhaustion)! This behavior will affect overall performance of the application and might cause poor user experience. Service B slowness indirectly affects Service A performance as well.

Bulkhead Pattern helps us to allocate limit the resources which can be used for specific services. So that resource exhaustion can be reduced. For example, in the above scenario, we limit the max number of threads which can be used for /a/b is 2. So, we have always some resources to process both endpoints



maxConcurrentCalls: max number of concurrent calls allowed to a particular service.

maxWaitDuration: any additional requests will wait for the given duration. Otherwise it will go with default/fallback method.