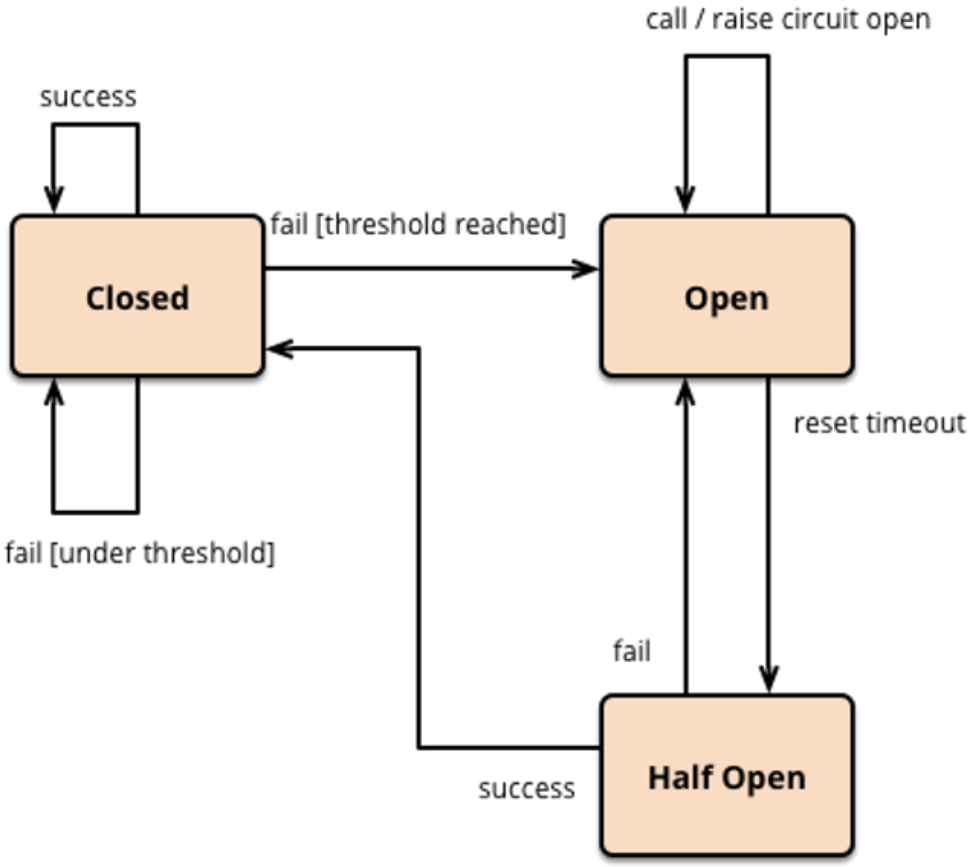
CIRCUIT BREAKER



Alberto Freije Carballo – UO257351

Guillermo Astorga Manzanal – UO269450

¿Qué es un Circuit Breaker?

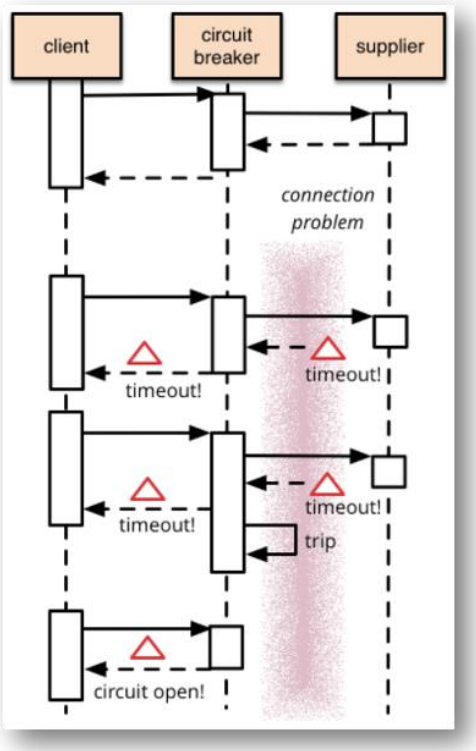
Alguna vez hemos experimentado un apagón en nuestra casa ya sea debido a un cortocircuito en la nevera o en el televisor, los automáticos de la casa saltan protegiendo el resto de la instalación eléctrica cuando la corriente es demasiado alta y evitando que esta se sobrecargue.

Esto es en lo que se basa el Circuit Breaker pero en este caso en nuestro servicio software.

El Circuit Breaker es un patrón Software diseñado para las llamadas remotas. Es un patrón fácil de implementar pues basta con implementar simplemente una librería aunque es importante configurarlo bien puesto que puede llegar a causar mas problemas que virtudes en tal caso.

¿En que consiste?

El Circuit Breaker encapsula las llamadas remotas en un objeto que se encarga de gestionar los fallos . Esto evita sobrecargar el sistema con las llamadas.



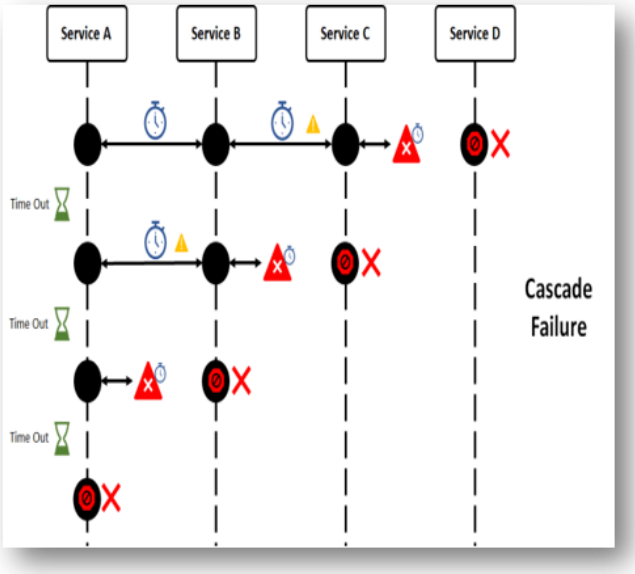
¿ Pero que son las llamadas remotas ?

Las llamadas remotas por ejemplo serian las peticiones que le hacemos a una base de datos. Actualmente las llamadas remotas forman parte de casi cualquier aplicación software.

A diferencia de las llamadas en memoria , las llamadas remotas pueden provocar fallos de conexión , ya sea porque la conexión al recurso simplemente no se llega a realizar (P.E: por

falta de conexión) o quedarse colgadas ya que el recurso no es capaz de responder a la petición.

Si las llamadas son múltiples pueden provocar un fallo en cascada (muchas veces por TimeOut) como en la siguiente fotografía.



¿Cómo funciona?

La idea básica del circuit braker es **encapsular** a una llamada a una función en un objeto **CircuitBraker**, quien maneja los fallos (mencionados anteriormente). Una vez que los fallos lleguen a una cantidad, el sistema cambia de dinámica, y a partir de ese momento devuelve **errores** a todas las llamadas hechas el CircuitBraker, **sin siquiera ejecutar** la función protegida.

Un circuit breaker tiene principalmente tres estados:

Cerrado : Permite la realización de llamadas.

Semi-abierto: Se evalúa el tiempo transcurrido desde el último error de conexión, y se realiza o no una nueva llamada.

Abierto: Las llamadas no se realizan y se devuelve , cada vez que se intente realizar una, un error.

Mediante el registro de fallos podremos decidir cuando el “circuito” se abrirá impidiendo la realización de más llamadas.

Cambios de estado:

A cerrado: cuando la conexión es exitosa, se establece el numero de fallos a cero y se pone el estado del CircuitBraker a “cerrado”.

A abierto: cuando numerosas conexiones han resultado en fallo (número acumulativo de fallos > número máximo de fallos) el circuito se abre, y se impiden las conexiones

A semi-Abierto: Si el estado del circuitBraker es “abierto” y la diferencia de tiempo (tiempo desde la última llamada con error > un valor arbitrario) se pone el estado a “semi-abierto”. La próxima vez que se intente realizar una llamada será como si el circuito estuviese “cerrado”, sin alterar el numero de fallos.

¿En que situación beneficiaria usarlo?

Una situación de ejemplo en el que un servicio de pago de una tienda de ropa online está sobrecargado actualmente. El servicio no puede manejar las solicitudes entrantes y se agregan a una cola de solicitudes hasta que finalmente se eliminan, lo que genera tiempos de espera para consumir servicios.

Aparte del hecho obvio de que los tiempos de respuesta prolongados y los tiempos de espera son algo que queremos evitar, continuar enviando solicitudes (que inevitablemente se agregan a la cola de solicitudes) va a reserva recursos tanto para el sistema sobrecargado como para nuestro propio servicio, Este es un problema que se puede propagar fácilmente a otros servicios también en una arquitectura de sistemas distribuidos.

Una consecuencia de esto podría ser:

El sistema sobrecargado no tiene la oportunidad de recuperarse y podría terminar en un estado irrecuperable, dejando de responder o no disponible.

Implementación básica

Un CircuitBraker sencillo requiere mínimamente :

- Un atributo de estado: “cerrado” o “abierto”.
- Un contador con el número de veces que ha fallado la conexión.
- Un valor máximo de fallo de conexiones, que cambiara el estado del objeto a abierto.
- Una función de reseteo del sistema.
- Una función para registrar el fallo.
- La función que realiza la lógica de la conexión y, si así fuera, la llamada.

Implementación 1

```python
class CircuitBraker():
    def __init__(self):
        self.tiempo_timeout=0.1 #segundos por ejemplo
        self.fallos_maximos=5
        self.fallos=0
        self.cerrado=True

    def call(self,funcion):
        if(self.cerrado):
            try:
                funcion()
                self.reset()
            except TimeoutError:
                self.registrar_fallo()
        else:
            raise Exception("UnrechableCode")

    def reset(self):
        self.fallos=0
        self.cerrado=True

    def registrar_fallo(self):
        self.fallos+=1
        if(self.fallos==self.fallos_maximos):
            self.cerrado=False
            self.time_failed=time.time()
def doSomething():
    #Funcion que hace algo
    print("Realizar operación de conexión")

circuit_braker= CircuitBraker()
circuit_braker.call(doSomething)
```

En esta primera implementación, podemos ver diferentes métodos del objeto:

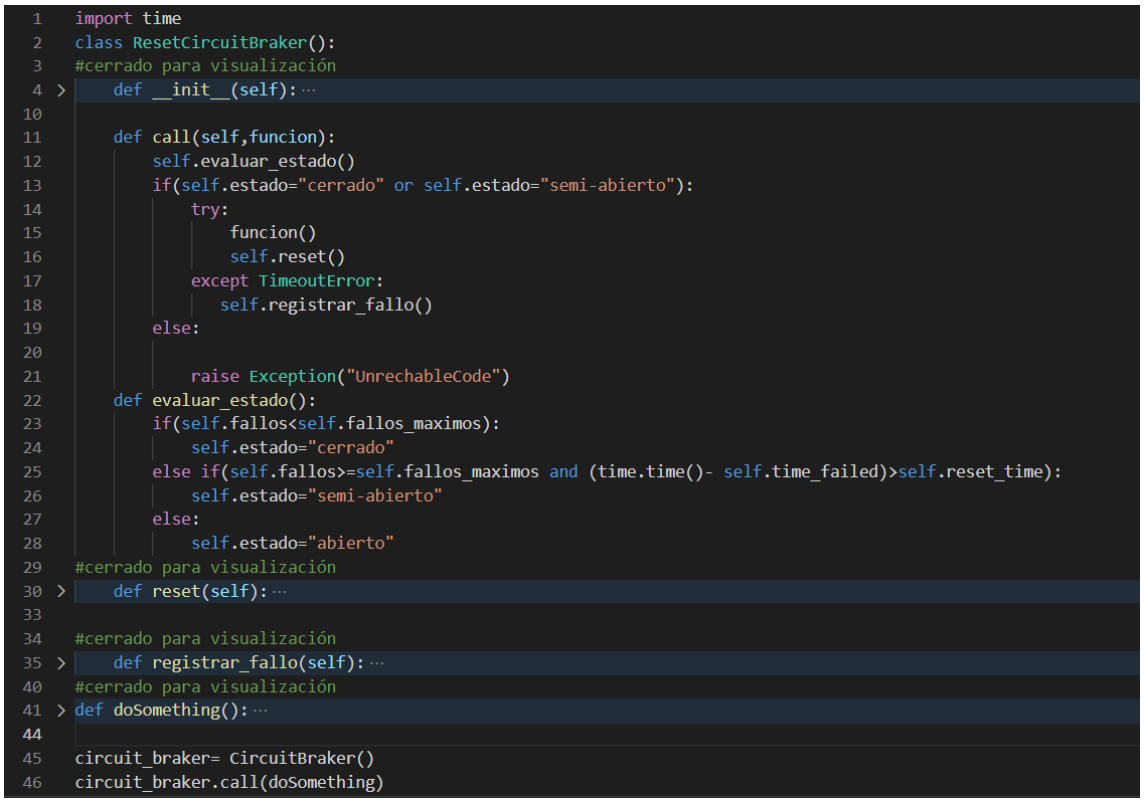
- __init__ : donde inicializamos las variables de fallos, fallos_máximos y el estado del Circuito
- Call : la función que se encarga de la lógica de las llamadas
- Reset: que restablece el numero de fallos a 0 y cierra el circuito
- Registrar_fallo: que contabiliza el fallo

Problemas de esta implementación.

- Esta es una implementación muy simplificada, y no muy recomendable…
- No tiene el estado de semi-abierto, que es parte de la clave del patrón.
- El principal problema que supone este diseño del patrón es que una vez que el circuito se abre, no hay otra manera de cerrarlo (y volver a permitir llamadas) que con una intervención manual del exterior, haciendo uso de la función reset().

Implementación 2

```python
def __init__(self):
    self.fallos_maximos=5
    self.fallos=0
    self.time_failed=None
    self.estado="cerrado"
    self.reset_time=0.01
```



```python
import time
class ResetCircuitBraker():
#cerrado para visualización
    def __init__(self):···

    def call(self,funcion):
        self.evaluar_estado()
        if(self.estado="cerrado" or self.estado="semi-abierto"):
            try:
                funcion()
                self.reset()
            except TimeoutError:
                self.registrar_fallo()
        else:

            raise Exception("UnrechableCode")
    def evaluar_estado():
        if(self.fallos<self.fallos_maximos):
            self.estado="cerrado"
        else if(self.fallos>=self.fallos_maximos and (time.time()- self.time_failed)>self.reset_time):
            self.estado="semi-abierto"
        else:
            self.estado="abierto"
#cerrado para visualización
    def reset(self):···

#cerrado para visualización
    def registrar_fallo(self):···
#cerrado para visualización
def doSomething():···

circuit_braker= CircuitBraker()
circuit_braker.call(doSomething)
```

Ahora sí, el patrón ya tiene un sistema de volver a activar las llamadas, controlando también los fallos.

Se añade la función evaluar_estado, que será quien nos cambie el estado del circuito y que se evalia cada vez que se realiza una llamada a call.

Aún así podríamos incluir alguna mejoras sencillas:

- Parametrizar parámetros tales como: fallos_máximos, reset_time; que gestionan la base de este CircuitBraker.
- Controlar los fallos con un sistema de frecuencia de los mismos: (fallos totales/ numero conexiones totales), y actuar en consecuencia de este (cambiando también fallos_máximos por frecuencia_fallos_máximos).
- Añadir la funcionalidad de poder pasar argumentos a la función pasada por parámetro.

Pero… ¿Esto es todo del CircuitBraker?

Por supuesto que no. En la realidad este patrón es:

- Mucho más parametrizable que el mostrado
- Gestiona muchas más excepciones que un TimeoutError.
- Incluyen numerosas funcionalidades.

Además, dado que las llamadas de otros servicios pueden estar esperando un timeout, es aconsejable poner cada llamada (línea 15 Ej 2) en un hilo.

Este ejemplo es síncrono, pero el CircuitBraker puede ser también asíncrono. En ese caso, las llamadas pasarían a ocupar una cola de “espera”, permitiendo que el servicio remoto las vaya procesando “a su ritmo”. El circuito se abre cuando la cola se llena.

¿El CircuitBraker puede causar problemas?

Si no se ajusta correctamente puedes provocar algunos problemas:

- Menor rendimiento.
- Mayor tiempo de inactividad: Podemos estarle dando al servidor demasiado tiempo de recuperación, y estar esperando en vano.
- Genera errores que sin este patrón no aparecerían: P.E. evaluación de Errores 404 como errores de conexión.

Implementaciones reales

Dejamos a vuestra disposición algunos repositorios en los que se implementa el patrón de una manera real:

- https://github.com/fabfuel/circuitbreaker
- https://github.com/danielfm/pybreaker
- https://github.com/eelabs/circuit-breaker-python