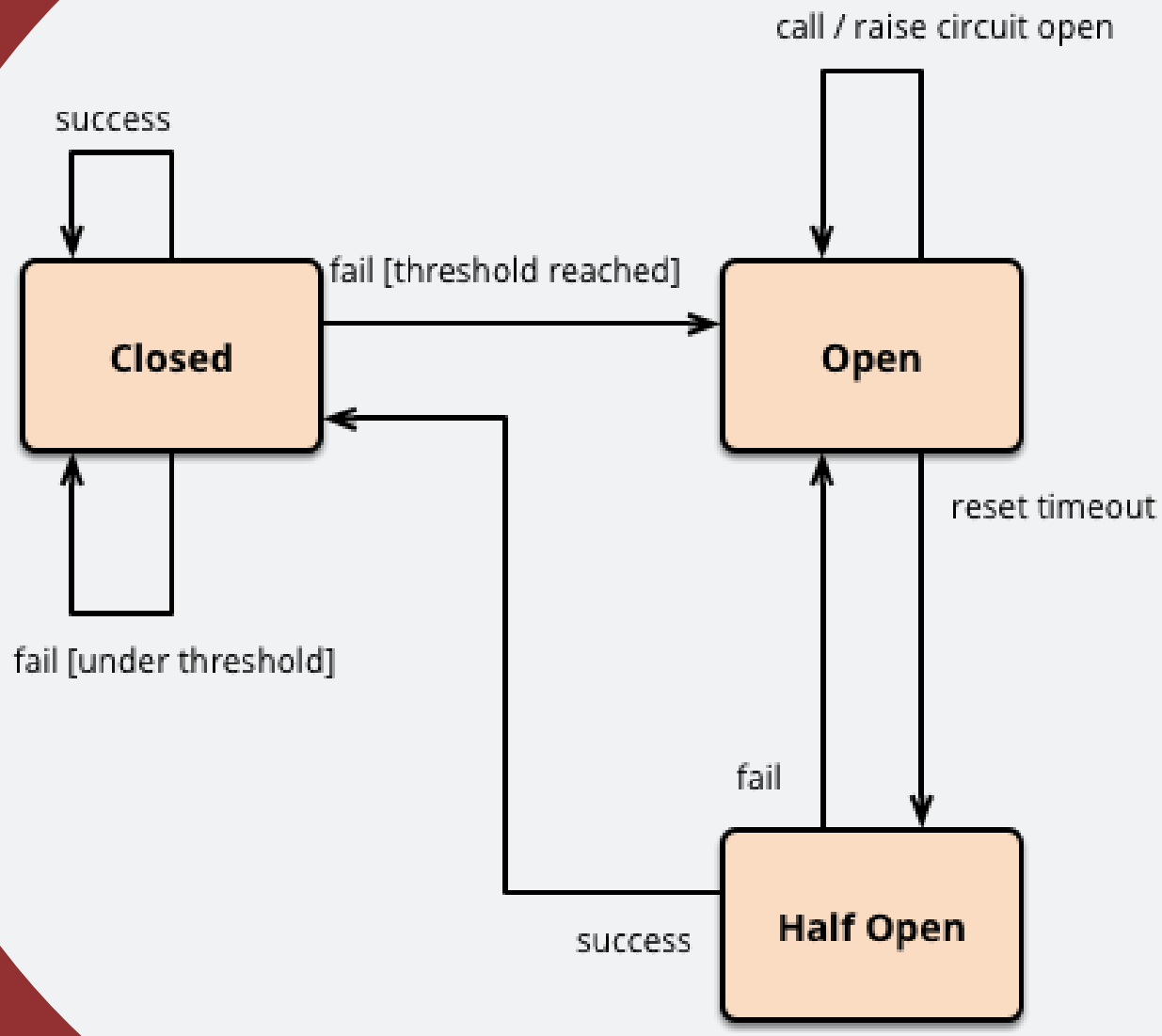
CircuitBraker

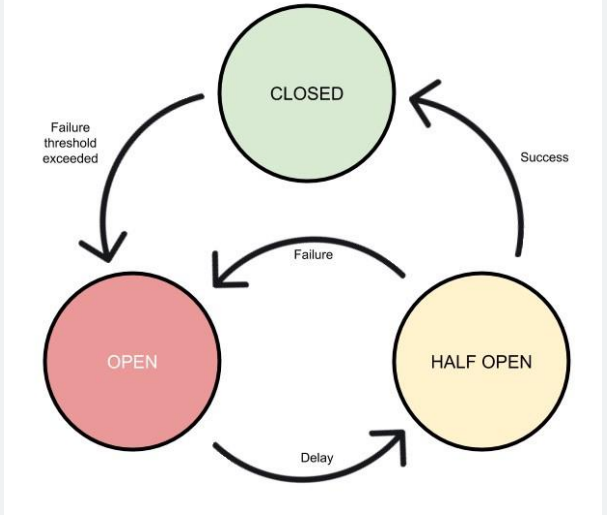
UO257351 - ALBERTO FREIJE CARBALLO

UO269450 - GUILLERMO ASTORGA MANZANAL



CircuitBraker: Origen

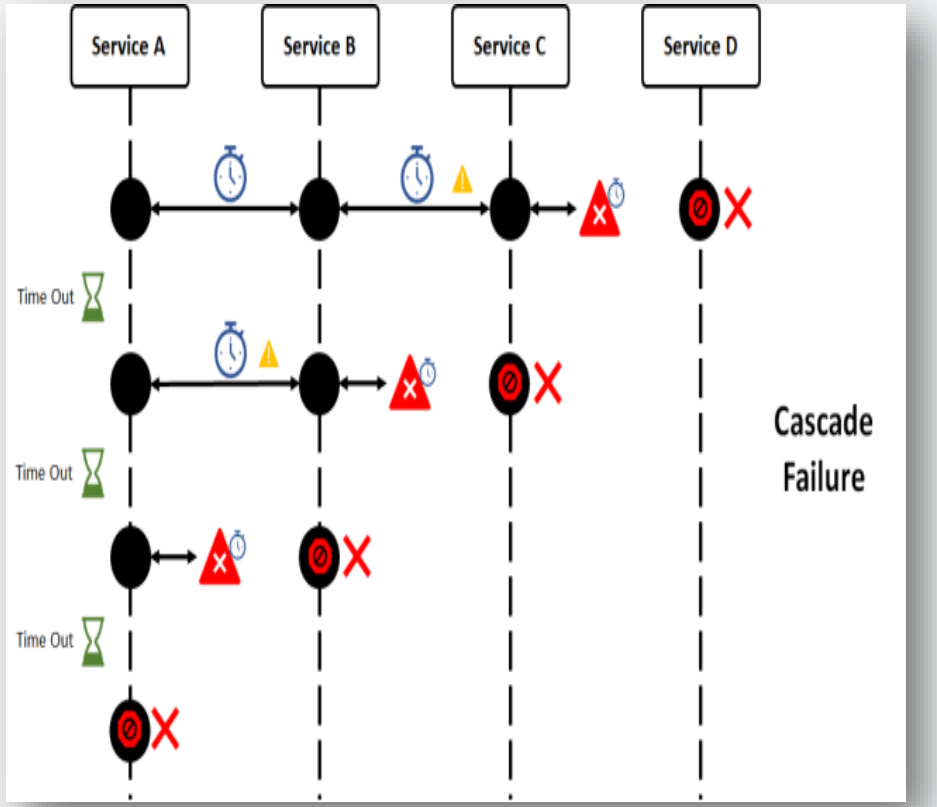
- Inspirado en la funcionalidad de los automáticos de una casa (proteger al resto de la instalación eléctrica cuando la corriente es demasiado alta) el Circuitbraker está diseñado para exactamente lo mismo, proteger un sistema software.



¿Qué es?

- Circuit Braker es un patrón software diseñado para las llamadas remotas.
- Encapsula las llamadas dentro de un objeto que gestiona los fallos.
- Protege tanto al sistema que lo utiliza, como a otros procesos dependientes del recurso: fallo en cascada.

Llamadas Remotas

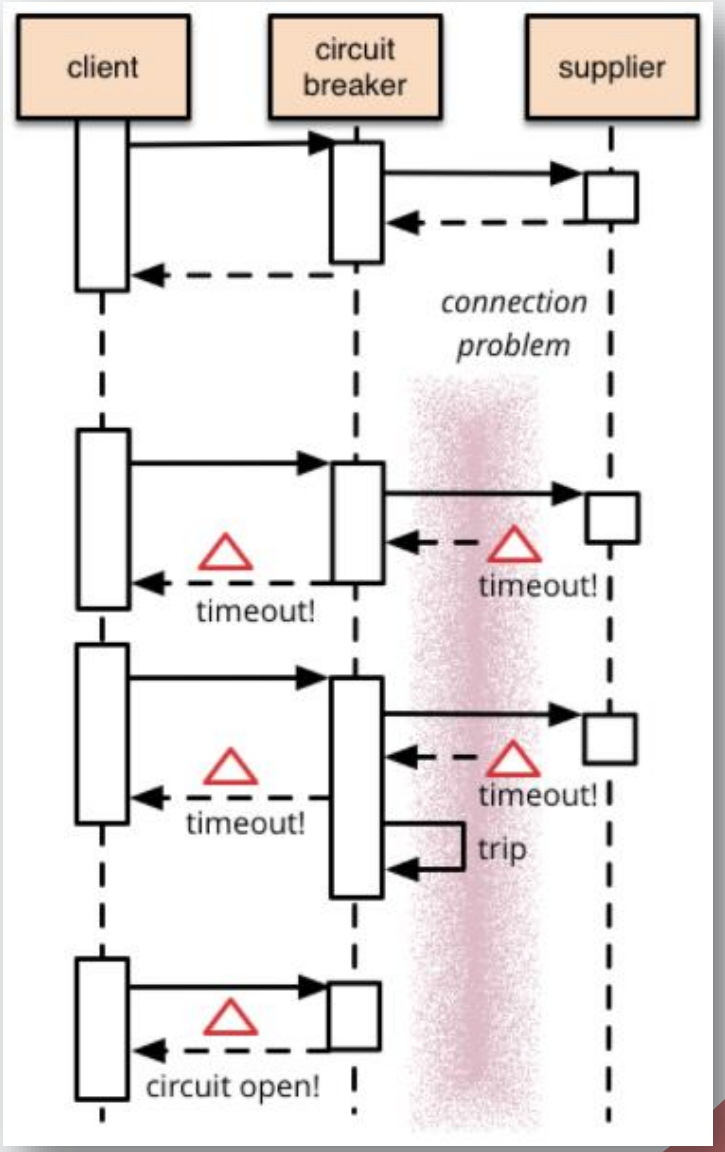


- Actualmente, las llamadas remotas forman parte de casi cualquier aplicación software.
- Pueden provocar fallos de conexión:
  - Fallar
  - Quedarse Colgadas
  - Si las llamadas son múltiples, pueden provocar un fallo en cascada (muchas veces por timeout), incluso entre varios sistemas.

CircuitBraker

Funciones:

- Encapsular la llamada en un objeto.
- Este objeto será quien gestione los fallos de conexión.
- Evitar sobrecargar el sistema con las llamadas.



CircuitBraker: Funcionamiento

- Un CircuitBraker gestiona los fallos de las llamadas remotas a través de diferentes estados:
  - Cerrado
  - Abierto
  - Semi-Abierto
- Registro de fallos

CircuitBraker: Cambios de estado

- Cambios de estado:
  - A cerrado: cuando la conexión es exitosa.
  - A abierto: (numero acumulativo de fallos > número máximo de fallos)
  - A semi-abierto: si el estado del circuitbraker es “abierto” y (tiempo desde la última llamada con error > un valor arbitrario).

¿En qué situación beneficiaria usarlo?

- Pongamos de ejemplo un servicio de pago sobrecargado.
- Grandes tiempos de espera
- Podria causar un estado irrecuperable

CircuitBraker: Implementación

Un CircuitBraker sencillo requiere mínimamente de:

- 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.

```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)
```

CircuitBraker: Implementación 1

CircuitBraker: Implementación 1

- 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().

```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)
```

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

CircuitBraker: Implementación 2

CircuitBraker: Implementación 2

- Ahora sí, el patrón ya tiene un sistema de volver a activar las llamadas, controlando también los fallos.
- Posibles cambios/mejoras que se le podrían hacer:
  - 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 suele ser:
  - 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
  - Genera errores que sin este patrón no aparecerían.

- Requiere un mantenimiento constante a lo largo de la vida de la aplicación software.

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

¿Preguntas?