Circuit Breaker Concept

<https://msdn.microsoft.com/en-us/library/dn589784.aspx>

Handle faults that may take a variable amount of time to rectify when connecting to a remote service or resource. This pattern can improve the stability and resiliency of an application.

# Context and Problem

In a distributed environment such as the cloud, where an application performs operations that access remote resources and services, it is possible for these operations to fail due to transient faults such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as that described by the [Retry pattern](https://msdn.microsoft.com/en-us/library/dn589788.aspx).

However, there may also be situations where faults are due to unexpected events that are less easily anticipated, and that may take much longer to rectify. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it may be pointless for an application to continually retry performing an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system may lead to cascading failures. For example, an operation that invokes a service could be configured to implement a timeout, and reply with a failure message if the service fails to respond within this period. However, this strategy could cause many concurrent requests to the same operation to be blocked until the timeout period expires. These blocked requests might hold critical system resources such as memory, threads, database connections, and so on. Consequently, these resources could become exhausted, causing failure of other possibly unrelated parts of the system that need to use the same resources. In these situations, it would be preferable for the operation to fail immediately, and only attempt to invoke the service if it is likely to succeed. Note that setting a shorter timeout may help to resolve this problem, but the timeout should not be so short that the operation fails most of the time, even if the request to the service would eventually succeed.

# Solution

The Circuit Breaker pattern can prevent an application repeatedly trying to execute an operation that is likely to fail, allowing it to continue without waiting for the fault to be rectified or wasting CPU cycles while it determines that the fault is long lasting. The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been rectified, the application can attempt to invoke the operation.

|  |
| --- |
| **Dn589784.note(en-us,PandP.10).gifNote:** |
| The purpose of the Circuit Breaker pattern is different from that of the Retry Pattern. The Retry Pattern enables an application to retry an operation in the expectation that it will succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application may combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient. |

A circuit breaker acts as a proxy for operations that may fail. The proxy should monitor the number of recent failures that have occurred, and then use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

The proxy can be implemented as a state machine with the following states that mimic the functionality of an electrical circuit breaker:

* **Closed**: The request from the application is routed through to the operation. The proxy maintains a count of the number of recent failures, and if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the **Open** state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the **Half-Open** state.

|  |
| --- |
| **Dn589784.note(en-us,PandP.10).gifNote:** |
| The purpose of the timeout timer is to give the system time to rectify the problem that caused the failure before allowing the application to attempt to perform the operation again. |

* **Open**: The request from the application fails immediately and an exception is returned to the application.
* **Half-Open**: A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it is assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state (the failure counter is reset). If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure.

|  |
| --- |
| **Dn589784.note(en-us,PandP.10).gifNote:** |
| The **Half-Open** state is useful to prevent a recovering service from suddenly being inundated with requests. As a service recovers, it may be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work may cause the service to time out or fail again. |

Figure 1 illustrates the states for one possible implementation of a circuit breaker.

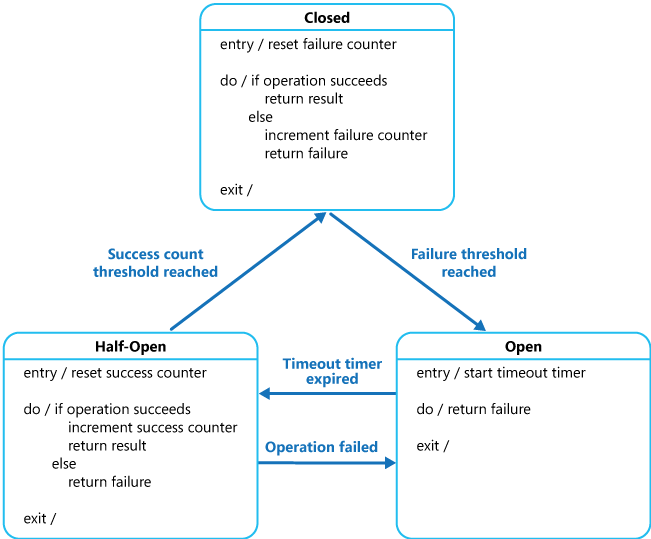


Figure 1 - Circuit Breaker states

Note that, in Figure 1, the failure counter used by the **Closed** state is time-based. It is automatically reset at periodic intervals. This helps to prevent the circuit breaker from entering the **Open** state if it experiences occasional failures; the failure threshold that trips the circuit breaker into the **Open** state is only reached when a specified number of failures have occurred during a specified interval. The success counter used by the **Half-Open** state records the number of successful attempts to invoke the operation. The circuit breaker reverts to the **Closed** state after a specified number of consecutive operation invocations have been successful. If any invocation fails, the circuit breaker enters the **Open**state immediately and the success counter will be reset the next time it enters the **Half-Open** state.

|  |
| --- |
| **Dn589784.note(en-us,PandP.10).gifNote:** |
| How the system recovers is handled externally, possibly by restoring or restarting a failed component or repairing a network connection. |

Implementing the circuit breaker pattern adds stability and resiliency to a system, offering stability while the system recovers from a failure and minimizing the impact of this failure on performance. It can help to maintain the response time of the system by quickly rejecting a request for an operation that is likely to fail, rather than waiting for the operation to time out (or never return). If the circuit breaker raises an event each time it changes state, this information can be used to monitor the health of the part of the system protected by the circuit breaker, or to alert an administrator when a circuit breaker trips to the **Open** state.

The pattern is customizable and can be adapted according to the nature of the possible failure. For example, you can apply an increasing timeout timer to a circuit breaker. You could place the circuit breaker in the **Open** state for a few seconds initially, and then if the failure has not been resolved increase the timeout to a few minutes, and so on. In some cases, rather than the **Open** state returning failure and raising an exception, it could be useful to return a default value that is meaningful to the application.

# Issues and Considerations

You should consider the following points when deciding how to implement this pattern:

* **Exception Handling**. An application invoking an operation through a circuit breaker must be prepared to handle the exceptions that could be raised if the operation is unavailable. The way in which such exceptions are handled will be application specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.
* **Types of Exceptions**. A request may fail for a variety of reasons, some of which may indicate a more severe type of failure than others. For example, a request may fail because a remote service has crashed and may take several minutes to recover, or failure could be caused by a timeout due to the service being temporarily overloaded. A circuit breaker may be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it may require a larger number of timeout exceptions to trip the circuit breaker to the **Open** state compared to the number of failures due to the service being completely unavailable.
* **Logging**. A circuit breaker should log all failed requests (and possibly successful requests) to enable an administrator to monitor the health of the operation that it encapsulates.
* **Recoverability**. You should configure the circuit breaker to match the likely recovery pattern of the operation it is protecting. For example, if the circuit breaker remains in the **Open** state for a long period, it could raise exceptions even if the reason for the failure has long since been resolved. Similarly, a circuit breaker could oscillate and reduce the response times of applications if it switches from the **Open** state to the **Half-Open** state too quickly.
* **Testing Failed Operations**. In the **Open** state, rather than using a timer to determine when to switch to the **Half-Open** state, a circuit breaker may instead periodically ping the remote service or resource to determine whether it has become available again. This ping could take the form of an attempt to invoke an operation that had previously failed, or it could use a special operation provided by the remote service specifically for testing the health of the service, as described by the [Health Endpoint Monitoring pattern](https://msdn.microsoft.com/en-us/library/dn589789.aspx).
* **Manual Override**. In a system where the recovery time for a failing operation is extremely variable, it may be beneficial to provide a manual reset option that enables an administrator to forcibly close a circuit breaker (and reset the failure counter). Similarly, an administrator could force a circuit breaker into the **Open** state (and restart the timeout timer) if the operation protected by the circuit breaker is temporarily unavailable.
* **Concurrency**. The same circuit breaker could be accessed by a large number of concurrent instances of an application. The implementation should not block concurrent requests or add excessive overhead to each call to an operation.
* **Resource Differentiation.**Be careful when using a single circuit breaker for one type of resource if there might be multiple underlying independent providers. For example, in a data store that comprises multiple shards, one shard may be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are conflated, an application may attempt to access some shards even when failure is highly likely, while access to other shards may be blocked even though it is likely to succeed.
* **Accelerated Circuit Breaking.**Sometimes a failure response can contain enough information for the circuit breaker implementation to know it should trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that is overloaded could indicate that an immediate retry is not recommended and that the application should instead try again in a few minutes time.

|  |
| --- |
| **Dn589784.note(en-us,PandP.10).gifNote:** |
| The HTTP protocol defines the “HTTP 503 Service Unavailable” response that can be returned if a requested service is not currently available on a particular web server. This response can include additional information, such as the anticipated duration of the delay. |

* **Replaying Failed Requests**. In the **Open** state, rather than simply failing quickly, a circuit breaker could also record the details of each request to a journal and arrange for these requests to be replayed when the remote resource or service becomes available.
* **Inappropriate Timeouts on External Services**. A circuit breaker may not be able to fully protect applications from operations that fail in external services that are configured with a lengthy timeout period. If the timeout is too long, a thread running a circuit breaker may be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other application instances may also attempt to invoke the service through the circuit breaker and tie up a significant number of threads before they all fail.

# When to Use this Pattern

Use this pattern:

* To prevent an application from attempting to invoke a remote service or access a shared resource if this operation is highly likely to fail.

This pattern might not be suitable:

* For handling access to local private resources in an application, such as in-memory data structure. In this environment, using a circuit breaker would simply add overhead to your system.
* As a substitute for handling exceptions in the business logic of your applications.

<https://en.wikipedia.org/wiki/Circuit_breaker_design_pattern>

**Circuit breaker** is a [design pattern](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) in modern [software development](https://en.wikipedia.org/wiki/Software_development).

Circuit breaker is used to detect failures and encapsulates logic of preventing a failure to reoccur constantly (during maintenance, temporary external system failure or unexpected system difficulties).

## **Common Uses[**[**edit**](https://en.wikipedia.org/w/index.php?title=Circuit_breaker_design_pattern&action=edit&section=1)**]**

Assume that an application connects to a [database](https://en.wikipedia.org/wiki/Database) 100 times per second and the database fails. The application designer does not want to have the same error reoccur constantly. They also want to handle the error quickly and gracefully without waiting for [TCP connection](https://en.wikipedia.org/wiki/TCP_connection) timeout.

Generally Circuit Breaker can be used to check the availability of an external service. An external service can be a database server or a web service used by the application.

[Circuit breaker](https://en.wikipedia.org/wiki/Circuit_breaker) detects failures and prevents the application from trying to perform the action that is doomed to fail (until it's safe to retry).

## **Implementation[**[**edit**](https://en.wikipedia.org/w/index.php?title=Circuit_breaker_design_pattern&action=edit&section=2)**]**

The Circuit Breaker Design Pattern should be implemented asynchronously. The reason is to offload the logic to detect failures from the actual request.

This requires Circuit Breaker to use a persistent storage layer, e.g. a network cache such as [Memcached](https://en.wikipedia.org/wiki/Memcached) or [Redis](https://en.wikipedia.org/wiki/Redis), or local cache (disk or memory based) to record the availability of what is, to the application, an external service.

Circuit Breaker records the state of the external service on a given interval.

Before the external service is used from the application, the storage layer is queried to retrieve the current state.

## **Performance Implication[**[**edit**](https://en.wikipedia.org/w/index.php?title=Circuit_breaker_design_pattern&action=edit&section=3)**]**

While it's safe to say that the benefits outweigh the consequences, implementing Circuit Breaker will negatively affect the performance.

By how much depends on the storage layer used and generally available resources. The largest factors in this regard are the type of cache, for example, disk-based vs. memory-based and local vs. network.

References

<https://github.com/grro/stability>

<http://www.javaworld.com/article/2824163/application-performance/stability-patterns-applied-in-a-restful-architecture.html?page=3>

<https://github.com/Netflix/Hystrix/wiki/How-it-Works>

<https://spring.io/guides/gs/circuit-breaker/>

Hystrix API – Circuit Breaker as an alternative to Exception Handling

A small java application is given below.

# CircuitBreakerCommand.java

package com.ddlab.rnd.hystrix;  
import com.netflix.hystrix.HystrixCommand;  
import com.netflix.hystrix.HystrixCommandGroupKey;  
import com.netflix.hystrix.HystrixCommandProperties;  
  
public class CircuitBreakerCommand extends HystrixCommand<String> {  
 private final String message;  
  
 public CircuitBreakerCommand(String message) {  
 **super(HystrixCommand.Setter.*withGroupKey*(HystrixCommandGroupKey.Factory.*asKey*("MyGroup"))  
 .andCommandPropertiesDefaults(  
 HystrixCommandProperties.*Setter*()  
 .withCircuitBreakerEnabled(true)  
// .withCircuitBreakerRequestVolumeThreshold(0)  
// .withCircuitBreakerErrorThresholdPercentage(1)  
 ));**  
 this.message = message;  
 }  
  
 @Override  
 protected String run() {  
 int val = Integer.*parseInt*(message);  
 int k = val%2;  
 if( k == 0 )  
 throw new RuntimeException("Failed!");  
 else  
 return "Hi";  
 }  
  
 @Override  
 protected String getFallback() {  
 return "Hello Fallback";  
 }  
  
}

# Main.java

package com.ddlab.rnd.hystrix;  
import com.netflix.hystrix.Hystrix;  
import java.util.concurrent.ExecutionException;  
  
public class Main {  
  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
  
 for (int i = 0; i < 10; i++) {  
 String s1 = new CircuitBreakerCommand("" + i).execute();  
 System.*out*.println(i + "----" + s1);  
 }  
 Hystrix.*reset*();  
 }  
}