

Circuit Breaker pattern

06/23/2017 • 17 minutes to read • Contributors      [all](#)

In this article

[Context and problem](#)

[Solution](#)

[Issues and considerations](#)

[When to use this pattern](#)

[Example](#)

[Related patterns and guidance](#)

Handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.

Context and problem

In a distributed environment, calls to remote resources and services can 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 the [Retry pattern](#).

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations it might be pointless for an application to continually retry 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 might 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's likely to succeed. Note that setting a shorter timeout might help to resolve this problem, but the timeout shouldn't 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, popularized by Michael Nygard in his book, [Release It!](#), can prevent an application from repeatedly trying to execute an operation that's likely to fail. Allowing it to continue without waiting for the fault to be fixed 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 fixed, the application can try to invoke the operation.

The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can 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 might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an exception immediately.

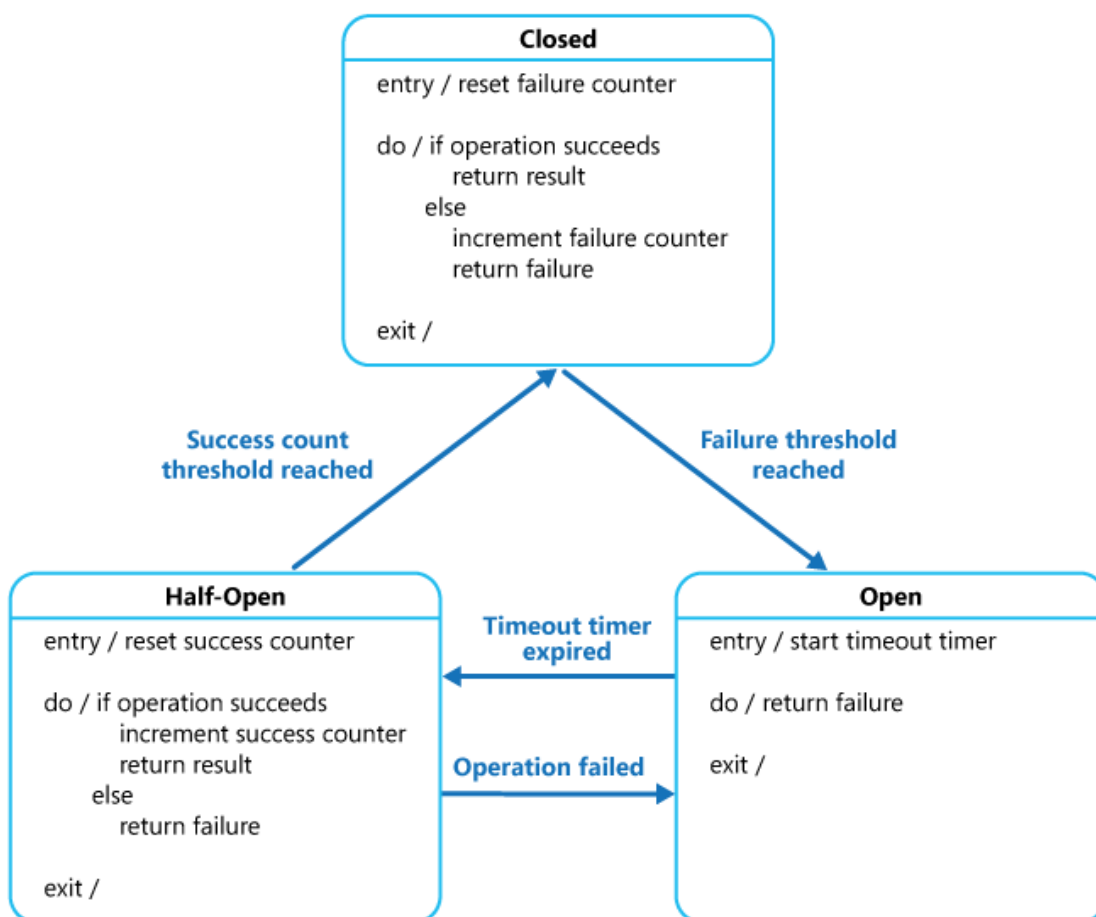
The proxy can be implemented as a state machine with the following states that mimic the functionality of an electrical circuit breaker:

- **Closed**: The request from the application is routed 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.

The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try 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's 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.

The **Half-Open** state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress a flood of work can cause the service to time out or fail again.



In the figure, the failure counter used by the **Closed** state is time based. It's 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 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.

How the system recovers is handled externally, possibly by restoring or restarting a failed component or repairing a network connection.

The Circuit Breaker pattern provides stability while the system recovers from a failure and minimizes the impact on performance. It can help to maintain the response time of the system by quickly rejecting a request for an operation that's 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 type 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 hasn't 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 raised if the operation is unavailable. The way 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 might fail for many reasons, some of which might indicate a more severe type of failure than others. For example, a request might fail because a remote service has crashed and will take several minutes to recover, or because of a timeout due to the service being temporarily overloaded. A circuit breaker might be able to examine the types of exceptions that occur and adjust its strategy depending on the nature of these exceptions. For example, it might 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.

Recoverability. You should configure the circuit breaker to match the likely recovery pattern of the operation it's 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 been resolved. Similarly, a circuit breaker could fluctuate 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 can instead periodically ping the remote service or resource to determine whether it's 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](#).

Manual Override. In a system where the recovery time for a failing operation is extremely variable, it's beneficial to provide a manual reset option that enables an administrator to 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 shouldn't 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 contains multiple shards, one shard might be fully accessible while another is experiencing a temporary issue. If the error responses in these scenarios are merged, an application might try to access some shards even when failure is highly likely, while access to other shards might be blocked even though it's likely to succeed.

Accelerated Circuit Breaking. Sometimes a failure response can contain enough information for the circuit breaker to trip immediately and stay tripped for a minimum amount of time. For example, the error response from a shared resource that's overloaded could indicate that an immediate retry isn't recommended and that the application should instead try again in a few minutes.

⚠ Note

A service can return HTTP 429 (Too Many Requests) if it is throttling the client, or HTTP 503 (Service Unavailable) if the service is not currently available. The 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 might 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 might be blocked for an extended period before the circuit breaker indicates that the operation has failed. In this time, many other application instances might also try 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 trying to invoke a remote service or access a shared resource if this operation is highly likely to fail.

This pattern isn't recommended:

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

Example


In a web application, several of the pages are populated with data retrieved from an external service. If the system implements minimal caching, most hits to these pages will cause a round trip to the service. Connections from the web application to the service could be configured with a timeout period (typically 60 seconds), and if the service doesn't respond in this time the logic in each web page will assume that the service is unavailable and throw an exception.

However, if the service fails and the system is very busy, users could be forced to wait for up to 60 seconds before an exception occurs. Eventually resources such as memory, connections, and threads could be exhausted, preventing other users from connecting to the system, even if they aren't accessing pages that retrieve data from the service.

Scaling the system by adding further web servers and implementing load balancing might delay when resources become exhausted, but it won't resolve the issue because user requests will still be unresponsive and all web servers could still eventually run out of resources.

Wrapping the logic that connects to the service and retrieves the data in a circuit breaker could help to solve this problem and handle the service failure more elegantly. User requests will still fail, but they'll fail more quickly and the resources won't be blocked.


The `CircuitBreaker` class maintains state information about a circuit breaker in an object that implements the `ICircuitBreakerStateStore` interface shown in the following code.

C#	
<pre>interface ICircuitBreakerStateStore { CircuitBreakerStateEnum State { get; } Exception LastException { get; } DateTime LastStateChangedDateUtc { get; } void Trip(Exception ex); void Reset(); void HalfOpen(); bool IsClosed { get; } }</pre>	

The `State` property indicates the current state of the circuit breaker, and will be either **Open**, **HalfOpen**, or **Closed** as defined by the `CircuitBreakerStateEnum` enumeration. The `IsClosed` property should be true if the circuit breaker is closed, but false if it's open or half open. The `Trip` method switches the state of the circuit breaker to the open state and records the exception that caused the change in state, together with the date and time that the exception occurred. The `LastException` and the `LastStateChangedDateUtc` properties return this information. The `Reset` method closes the circuit breaker, and the `HalfOpen` method sets the circuit breaker to half open.

The `InMemoryCircuitBreakerStateStore` class in the example contains an implementation of the `ICircuitBreakerStateStore` interface. The `CircuitBreaker` class creates an instance of this class to hold the state of the circuit breaker.

The `ExecuteAction` method in the `CircuitBreaker` class wraps an operation, specified as an `Action` delegate. If the circuit breaker is closed, `ExecuteAction` invokes the `Action` delegate. If the operation fails, an exception handler calls `TrackException`, which sets the circuit breaker state to open. The following code example highlights this flow.

C#	
<pre>public class CircuitBreaker { private readonly ICircuitBreakerStateStore stateStore = CircuitBreakerStateStoreFactory.GetCircuitBreakerStateStore(); private readonly object halfOpenSyncObject = new object (); ... public bool IsClosed { get { return stateStore.IsClosed; } }</pre>	

```

public bool IsOpen { get { return !IsClosed; }}

public void ExecuteAction(Action action)
{
    ...
    if (IsOpen)
    {
        // The circuit breaker is Open.
        ... (see code sample below for details)
    }

    // The circuit breaker is Closed, execute the action.
    try
    {
        action();
    }
    catch (Exception ex)
    {
        // If an exception still occurs here, simply
        // retrip the breaker immediately.
        this.TrackException(ex);

        // Throw the exception so that the caller can tell
        // the type of exception that was thrown.
        throw;
    }
}

private void TrackException(Exception ex)
{
    // For simplicity in this example, open the circuit breaker on the first exception.
    // In reality this would be more complex. A certain type of exception, such as one
    // that indicates a service is offline, might trip the circuit breaker immediately.
    // Alternatively it might count exceptions locally or across multiple instances and
    // use this value over time, or the exception/success ratio based on the exception
    // types, to open the circuit breaker.
    this.stateStore.Trip(ex);
}
}

```

The following example shows the code (omitted from the previous example) that is executed if the circuit breaker isn't closed. It first checks if the circuit breaker has been open for a period longer than the time specified by the local `OpenToHalfOpenWaitTime` field in the `CircuitBreaker` class. If this is the case, the `ExecuteAction` method sets the circuit breaker to half open, then tries to perform the operation specified by the `Action` delegate.

If the operation is successful, the circuit breaker is reset to the closed state. If the operation fails, it is tripped back to the open state and the time the exception occurred is updated so that the circuit breaker will wait for a further period before trying to perform the operation again.

If the circuit breaker has only been open for a short time, less than the `OpenToHalfOpenWaitTime` value, the `ExecuteAction` method simply throws a `CircuitBreakerOpenException` exception and returns the error that caused the circuit breaker to transition to the open state.

Additionally, it uses a lock to prevent the circuit breaker from trying to perform concurrent calls to the operation while it's half open. A concurrent attempt to invoke the operation will be handled as if the circuit breaker was open, and it'll fail with an exception as described later.

C#

 Copy

```

...
if (IsOpen)
{
    // The circuit breaker is Open. Check if the Open timeout has expired.
    // If it has, set the state to HalfOpen. Another approach might be to

```

```

// check for the HalfOpen state that had be set by some other operation.
if (stateStore.LastStateChangedDateUtc + OpenToHalfOpenWaitTime < DateTime.UtcNow)
{
    // The Open timeout has expired. Allow one operation to execute. Note that, in
    // this example, the circuit breaker is set to HalfOpen after being
    // in the Open state for some period of time. An alternative would be to set
    // this using some other approach such as a timer, test method, manually, and
    // so on, and check the state here to determine how to handle execution
    // of the action.
    // Limit the number of threads to be executed when the breaker is HalfOpen.
    // An alternative would be to use a more complex approach to determine which
    // threads or how many are allowed to execute, or to execute a simple test
    // method instead.
    bool lockTaken = false;
    try
    {
        Monitor.TryEnter(halfOpenSyncObject, ref lockTaken);
        if (lockTaken)
        {
            // Set the circuit breaker state to HalfOpen.
            stateStore.HalfOpen();

            // Attempt the operation.
            action();

            // If this action succeeds, reset the state and allow other operations.
            // In reality, instead of immediately returning to the Closed state, a counter
            // here would record the number of successful operations and return the
            // circuit breaker to the Closed state only after a specified number succeed.
            this.stateStore.Reset();
            return;
        }
    }
    catch (Exception ex)
    {
        // If there's still an exception, trip the breaker again immediately.
        this.stateStore.Trip(ex);

        // Throw the exception so that the caller knows which exception occurred.
        throw;
    }
    finally
    {
        if (lockTaken)
        {
            Monitor.Exit(halfOpenSyncObject);
        }
    }
}
// The Open timeout hasn't yet expired. Throw a CircuitBreakerOpen exception to
// inform the caller that the call was not actually attempted,
// and return the most recent exception received.
throw new CircuitBreakerOpenException(stateStore.LastException);
}
...

```

To use a `CircuitBreaker` object to protect an operation, an application creates an instance of the `CircuitBreaker` class and invokes the `ExecuteAction` method, specifying the operation to be performed as the parameter. The application should be prepared to catch the `CircuitBreakerOpenException` exception if the operation fails because the circuit breaker is open. The following code shows an example:

C#

 Copy

```

var breaker = new CircuitBreaker();

try
{

```

```
breaker.ExecuteAction(() =>
{
    // Operation protected by the circuit breaker.
    ...
});
}
catch (CircuitBreakerOpenException ex)
{
    // Perform some different action when the breaker is open.
    // Last exception details are in the inner exception.
    ...
}
catch (Exception ex)
{
    ...
}
```

Related patterns and guidance

The following patterns might also be useful when implementing this pattern:

- [Retry pattern](#). Describes how an application can handle anticipated temporary failures when it tries to connect to a service or network resource by transparently retrying an operation that has previously failed.
- [Health Endpoint Monitoring pattern](#). A circuit breaker might be able to test the health of a service by sending a request to an endpoint exposed by the service. The service should return information indicating its status.