**Transient Faults:**

Transient faults are errors that occur due to some temporary conditions like network connectivity issues or service unavailability for a short time period. Typically, if you retry the same operation after few seconds, you find that the error has disappeared. Service downtime for maintenance should not be confused with the transient errors.

Unfortunately, there is no easy way to distinguish transient from non-transient faults; both would most likely result in exceptions being raised in your application. If you retry the operation that causes a non-transient fault (for example a "file not found" error), you most likely get the same exception raised again. There is no intrinsic way to distinguish between transient and non-transient faults unless the developer of the service explicitly isolated transient faults into a specified subset of exception types or error codes.

**Transient Fault Handling Application Block**

The Microsoft Enterprise Library Transient Fault Handling Application Block lets developers make their applications more resilient by adding robust transient fault handling logic. It does this in two ways.

First, the block includes logic to identify transient faults for a number of common cloud-based services in the form of detection strategies. These detection strategies contain built-in knowledge that is capable of identifying whether a particular exception is likely to be caused by a transient fault condition.

The block includes detection strategies for the following services:

* SQL Azure
* Azure Service Bus
* Azure Storage Service
* Azure Caching Service

Second, the application block enables us to define your retry strategies so that we can follow a consistent approach to handling transient faults in our applications. The specific retry strategy we use will depend on several factors; for example, how aggressively we want your application to perform retries, and how the service typically behaves when we perform retries. Some services can further throttle or even block client applications that retry too aggressively. A retry strategy defines how many retries we want to make before you decide that the fault is not transient, and what the intervals should be between the retries.

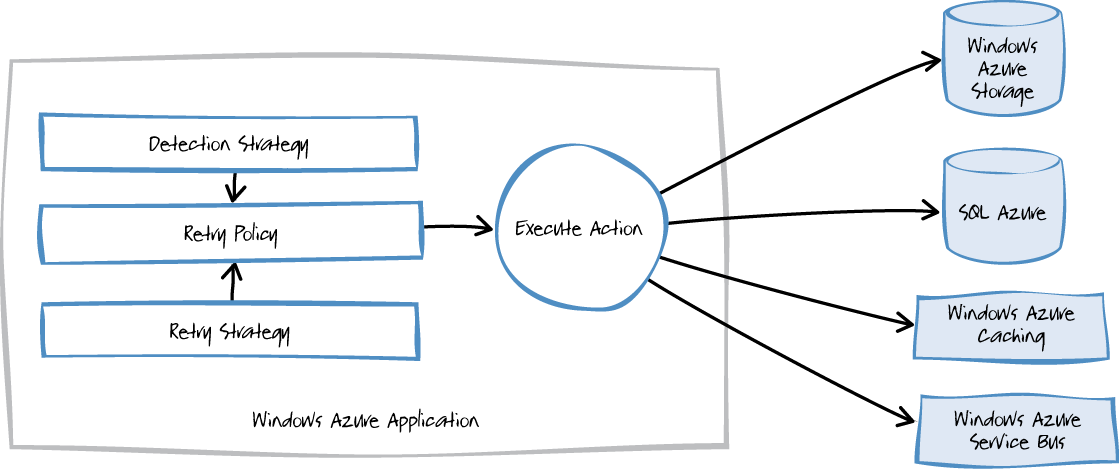
The built-in retry strategies allow we to specify that retries should happen at fixed intervals, at intervals that increase by the same amount each time, and at intervals that increase exponentially but with some random variation. The following table shows examples of all three strategies.

|  |  |
| --- | --- |
| **Retry Strategy** | **Example (intervals between retries in seconds)** |
| Fixed interval | 4,4,4,4,4 |
| Incremental intervals | 2,4,6,8,10,12 |
| Random exponential back-off intervals | 2, 3.755, 9.176, 14.306, 31.895 |

In many cases, we should use the random exponential back-off strategy to gracefully back off the load on the service. This is especially true if the service is throttling client requests.

We can define our own custom detection strategies if the built-in detection strategies included with the application block do not meet our requirements. The application block also allows us to define our own custom retry strategies that define additional patterns for retry intervals.

Fig: The Transient Fault Handling Application Block



**Technical Implementation**

* Install Transient Fault Handling Application Block using NUGET.
* Install the Application config tool to provide the configuration for retry strategy.
* Define the retry strategy with a specified number of retry attempts and an incremental time interval between retries.
* Define the retry policy - mechanism for unreliable actions and transient conditions.
* User ***ITransientErrorDetectionStrategy*** interface to define the transient exceptions

Sample Code:

class TransientErrorDetectionStrategy : ITransientErrorDetectionStrategy

{

public bool IsTransient(Exception ex)

{ // Logic to define the transient faults

}

}

* Use ***ReliableSqlConnection*** class with the ADO.NET along with the retry policy.

Sample Code:

var retryStrategy = new Incremental(RETRY\_COUNT, TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(2));

var policy = new RetryPolicy< TransientErrorDetectionStrategy >(retryStrategy);

using (var conn = new ReliableSqlConnection(conStr, policy))

{

// Logic to according to the requirement

}

* If we need to connect any other service, we can use the ***ExecuteAction*** method of the ***RetryPolicy*** class. It will take the method delegate as the input parameter and we can use that method to call the other service.

Sample Code:

try

{

var retryStrategy = new Incremental(RETRY\_COUNT, TimeSpan.FromSeconds(2), TimeSpan.FromSeconds(2));

var retryPolicy = new RetryPolicy< TransientErrorDetectionStrategy >(retryStrategy);

retryPolicy.ExecuteAction(GetData);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

Private Static void GetData()

{

WebClient wc = new WebClient();

wc.DownloadString("c:\\sample.txt");

}

**Problem**

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.

**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.

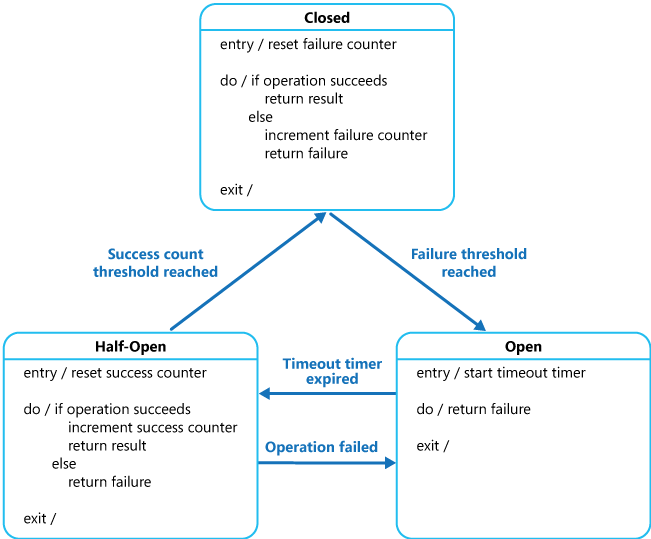
The purpose of the Circuit Breaker pattern is different from that of the Retry Pattern. The Retry Pattern enables an application to retry an operation in the expectation that it will succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application 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.

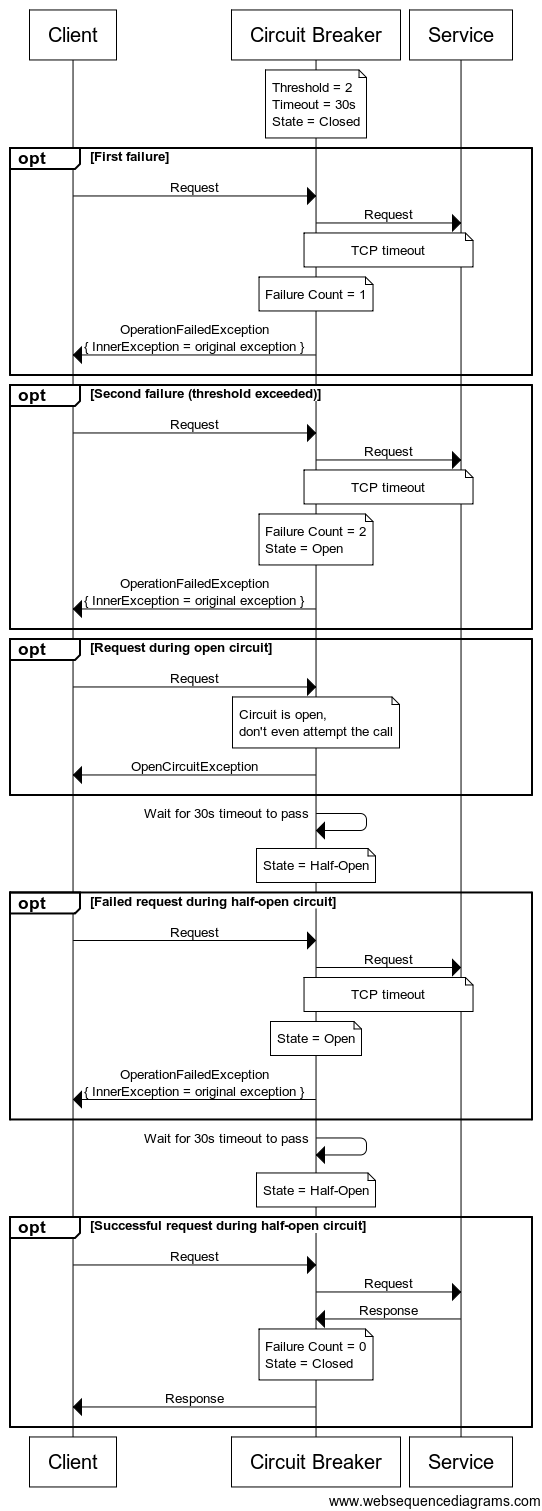
The proxy can be implemented as a state machine with the following states that mimic the functionality of an electrical circuit breaker:

* **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.
* **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.

**Fig** : Circuit Breaker States



Below diagram explains the pattern flow.



**When to Use:**

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

**When not to Use:**

* 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.

**Refrences:**

http://social.technet.microsoft.com/wiki/contents/articles/18665.cloud-service-fundamentals-data-access-layer-transient-fault-handling.aspx

<https://msdn.microsoft.com/en-us/library/hh680901%28v=pandp.50%29.aspx?f=255&MSPPError=-2147217396>

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