Understanding Retry Pattern With Exponential Back-Off and Circuit Breaker Pattern

(/users/1255375/rahulrajatsingh.html) by Rahul Rajat Singh **(/users/1255375/rahulrajatsingh.html)** MVB · **Oct. 10, 16 · Integration Zone (/enterprise-integration-training-tools-news) · Opinion**

In this article, we will discuss the importance of the retry pattern and how to implement it effectively in our applications. We will also discuss how exponential backoff and circuit breaker pattern can be used along with retry pattern. This is more of a theoretical article since the actual implementation of retry will depend a lot on the application needs.

Background

Most of us have implemented a retry mechanism one time or other in our applications. Why we need to call this a pattern and talk at length about this is the first question that comes to mind. To understand this, we first need to understand the concept of transient failures and why transient failures are something to be considered more in this modern cloud-based world.

Transient Failures

Transient failures are the failures that occur while communicating to the external component or service and that external service is not available. This unavailability or failure to connect is not due to any problem in the service, but due to some reasons like network failure or server overload. Such issues are ephemeral. If we call the service again, chances are that our call will succeed. Such failures are called transient failures.

Traditionally, we have been getting such errors in database connections and service calls. But in the new cloud world, the chances of getting such errors has increased since our application itself might also have some elements and components running in the cloud. It might be possible that different parts of our applications are hosted separately on the cloud. Failures like momentary loss of network, service unavailability, and timeouts have become more prominent (comparatively).

These kinds of failures can easily be circumvented by simply calling the service after a delay.

How to Handle Transient Faults Using the Retry Pattern

Before we even start talking about how to handle the transient faults, the first task should be to identify the transient faults. The way to do that is to check if the fault is something that the target service is sending and actually has some context from the application perspective. If this is the case then we know that this is not a transient fault since the service is sending us a fault. But if we are getting a fault that is not coming from the service and perhaps coming from some other reasons like infrastructure issues and the fault appears to be something that can be resolved by simply the calling service again, then we can classify it as a transient fault.

Once we have identified the fault as a transient fault, we need to put some retry logic so that the issue will get resolved simply by calling the service again. The typical way to implement the retry is as follows:

1. Identify if the fault is a transient fault.
2. Define the maximum retry count.
3. Retry the service call and increment the retry count.
4. If the calls succeeds, return the result to the caller.
5. If we are still getting the same fault, keep retrying until the maximum retry count is hit.
6. If the call is failing even after maximum retries, let the caller module know that the target service is unavailable.

The Problem With Simple Retry

The retry mechanism we discussed in the previous section is fairly straight forward; one would wonder why we're even discussing such a simple thing. Most of us use this retry mechanism in our applications. The reason we are discussing is this because there is a small problem in the above-mentioned retry mechanism.

To understand the problem, let's imagine a scenario where the transient fault is happening because the service is overloaded or some throttling is implemented at the service end. This service is rejecting new calls. This is a transient fault as if we call the service after some time, our call could succeed. There could also be a possibility that our retry requests are further adding to the overload of the busy service, which would mean that if there are multiple instances of our applications retrying for the same service, the service will be in the overloaded state longer and will take longer to recover from this state.

So in a way, our requests are contributing further to the reason of the fault. Ultimately, it's our application only that is suffering due to the longer recovery time of the service. How can we solve this problem?

Exponential Backoff to the Rescue

The idea behind using exponential backoff with retry is that instead of retrying after waiting for a fixed amount of time, we increase the waiting time between reties after each retry failure.

For example, when the request fails the first time, we retry after one second. If it fails for the second time, we wait for 2 seconds before next retry. If the second retry fails, we wait for four seconds before next retry. So we are incrementally increasing the wait time between the consecutive retry requests after each failure. This gives the service some breathing time so that if the fault is due to service overload, it could get resolved faster.

Note: These wait durations are indicative only. Actual wait time should depend on application and service. In practical scenarios, no user will wait for more than seven seconds for their request to process (like in the above example).

So with exponential backoff, our retry algorithm will look like following:

1. Identify if the fault is a transient fault.
2. Define the maximum retry count.
3. Retry the service call and increment the retry count

3. Retry the service call and increment the retry count.
4. If the calls succeeds, return the result to the caller.
5. If we are still getting the same fault, Increase the delay period for next retry.
6. Keep retrying and keep increasing the delay period until the maximum retry count is hit.
7. If the call is failing even after maximum retries, let the caller module know that the target service is unavailable.

Example Usage of Retry With Exponential Backoff

The best example for the implementation of this pattern is Entity Framework. Entity framework provides connection resiliency. Connection resiliency means that the entity framework is able to automatically reestablish the broken connections using the retry pattern.

Entity framework has an IDbExecutionStrategy interface which takes care of the connection retry mechanism. There are for implementations for this interface that comes with Entity Framework: DefaultExecutionStrategy, DefaultSqlExecutionStrategy, DbExecutionStrategy and SqlAzureExecutionStrategy. The first two strategies (DefaultExecutionStrategy and DefaultSqlExecutionStrategy) don't retry at all in case of failure. The other two i.e. DbExecutionStrategy and SqlAzureExecutionStrategy implements the retry with exponential backoff. In these strategies the first retry happens immediately and if that fails the next retry happens after a delay. This delay will increase exponentially until the maximum retry count is hit.

Long-Lasting Transient Faults

There is one more scenario which we need to handle where the transient faults are long lasting. The first question that comes to mind that is how a transient fault can be long-lasting or how can we call a long lasting fault a transient fault. One possible reason for the failure could be a network issue which is identified as a transient fault by our application but this connectivity issue is taking more that usual time to recover from. In this case, our application thinks of the fault as a transient fault but the fault exists for a long time. The problem with long-lasting transient faults is that our application will end up retrying, waiting and retrying till the maximum retry count is hit and thus wasting resources. So we should have some mechanism to identify long-lasting transient faults and let this retry mechanism not come into play when we have identified the fault as long-lasting transient fault.

Hello Circuit Breaker

The intent of the Circuit Breaker pattern is to handle the long-lasting transient faults. The idea behind this pattern is that we will wrap the service calls in a circuit breaker. If the number of retry failures reaches above a certain threshold value, we will make this circuit as OPEN, which means the service is not available at the time. Once the circuit reaches the OPEN state, further calls to the service will immediately return failure to the caller instead of executing our retry logic.

This circuit breaker will have some timeout period after which it will move to HALF-OPEN state. In this state, it will allow a service call which will determine if the service has become available or not. If the service is not available it goes back to OPEN state. If the service becomes available after this timeout, the circuit breaker moves to CLOSED state. The callers will be able to call the service and the usual retry mechanism will come in play again.

This will help is in circumventing all the useless retry execution in case the fault is long lasting and this saving resources and providing more immediate feedback to the callers.

Point of Interest

Transient faults are inevitable. And in the new world of cloud, we should be more considerate in anticipating transient faults. In this small article, we talked about how to handle transient faults using retry pattern with exponential backoff. We looked at how Entity framework uses this pattern. We also looked at how we can use circuit breaker pattern to take care of long-lasting transient faults.

One important thing to understand is that we should not always implement exponential backoff and circuit breaker blindly in our application. Whether to use them or not should be determined by the application requirements and the service behaviors. This article has been written from a beginner's perspective. I hope this has been somewhat informative.

Topics: INTEGRATION, RETRY PATTERN, TRANSIENT FAULTS, EXPONENTIAL BACKOFF