Short Retry vs Long Retry in Apache Camel

javacodegeeks.com/2017/06/short-retry-vs-long-retry-apache-camel-2.html

Camel Design Patterns book describes 20 patterns and numerous tips and best practices for designing Apache Camel based integration solutions. Each pattern is based on a real world use case and provides Camel specific implementation details and best practises. To get a feel of the book, below is an extract from the Retry Pattern from the book describing how to do Short and Long retires in Apache Camel.

Context and Problem

By their very nature integration applications have to interact with other systems over the network. With dynamic cloud-based environments becoming the norm, and the microservices architectural style partitioning applications into more granular services, the successful service communication has become a fundamental prerequisite for many distributed applications. Services that communicate with other services must be able to handle transient failures that can occur in downstream systems transparently, and continue operating without any disruption. As a transient failure can be considered an infrastructure-level fault, a loss of network connectivity, timeouts and throttling applied by busy services, etc. These conditions occur infrequently and they are typically self- correcting, and usually retrying an operation succeeds.

Forces and Solution

Reproducing and explaining transient failures can be a difficult task as these might be caused by a combination of factors happening irregularly and related to external systems. Tools such as Chaos Monkey can be used to simulate unpredictable system outages and let you test the application resiliency if needed. A good strategy for dealing with transient failures is to retry the operation and hope that it will succeed (if the error is truly transient, it will succeed; just keep calm and keep retrying).

To implement a “retry” logic there are a few areas to consider:

Which failures to retry?

Certain service operations, such as HTTP calls and relational database interactions, are potential candidates for a retry logic, but further analysis is needed before implementing it. A relational database may reject a connection attempt because it is throttling against excessive resource usage, or reject an SQL insert operation because of concurrent modification. Retrying in these situations could be successful. But if an relational database rejects a connection because of wrong credentials, or an SQL insert operation has failed because of foreign key constraints, retrying the operation will not help. Similarly with HTTP calls, retrying a connection timeout or response timeout may help, but retrying a SOAP Fault caused by a business error does not make any sense. So choose your retries carefully.

How often to retry?

Once a retry necessity has been identified, the specific retry policy should be tuned to satisfy the nature of both applications: the service consumer with the retry logic and the service provider with the transient failure. For example, if a real time integration service fails to process a request, it might be allowed to do only few retry attempts with short delays before returning a response, whereas a batch-based asynchronous service may be able to afford to do more retries with longer delays and exponential back off. The retry strategy should also consider other factors such as the service consumption contracts and the SLAs of the service provider. For example, a very aggressive retry strategy may cause further throttling and even a blacklisting of a service consumer, or it can fully overload and degrade a busy service and prevent it from recovering at all. Some APIs may give you an indication of the remaining request count for a time period and blacklisting information in the response, but some may not. So a retry strategy defines how often to

retry and for how long before you should accept the fact that it is a non-transient failure and give up.

Idempotency

When retrying an operation, consider the possible side effects on that operation. A service operation that will be consumed with retry logic should be designed and implemented as idempotent. Retrying the same operation with the same data input should not have any side effects. Imagine a request that has processed successfully, but the response has not reached back. The service consumer may assume that the request has failed and retry the same operation which may have some unexpected side effects.

Monitoring

Tracking and reporting retries is important too. If certain operations are constantly retried before succeeding or they are retried too many times before failing, these have to be identified and fixed. Since retries in a service are supposed to be transparent to the service consumer, without proper monitoring in place, they may remain undetected and affect the stability and the performance of the whole system in a negative way.

Timeouts and SLAs

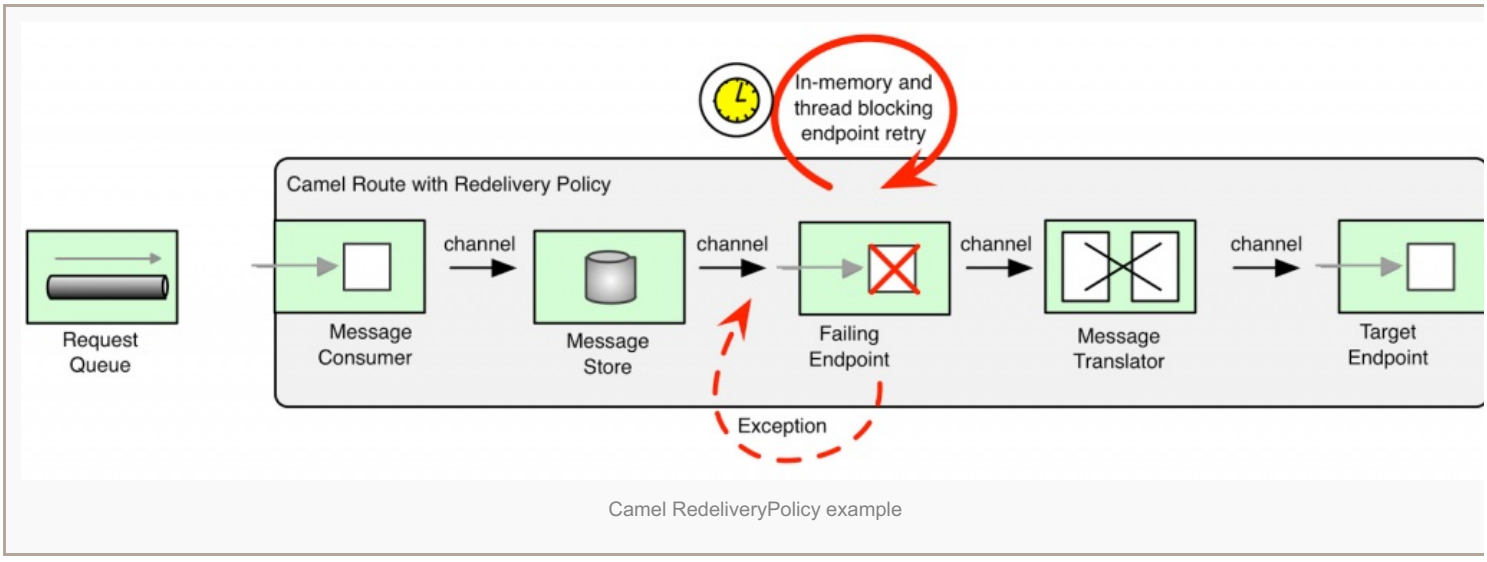
When transient failures happen in the downstream systems and the retry logic kicks in, the overall processing time of the retrying service will increase significantly. Rather than thinking about the retry parameters from the perspective of the number of retries and delays, it is important to drive these values from the perspective of service SLAs and service consumer timeouts. So take the maximum amount of time allowed to handle the request, and determine the maximum number of retries and delays (including the processing time) that can be squeezed into that time frame.

Mechanics

There are a few different ways of performing retries with Camel and ActiveMQ.

Camel RedeliveryPolicy (Short Retry)

This is the most popular and generic way of doing retries in a Camel. A redelivery policy defines the retry rules (such as the number of retries and delays, whether to use collision avoidance and an exponential backoff multiplier, and logging) which can then be applied to multiple errorHandler and onException blocks of the processing flow. Whenever an exception is thrown up, the rules in the redelivery policy will be applied.



Camel RedeliveryPolicy example

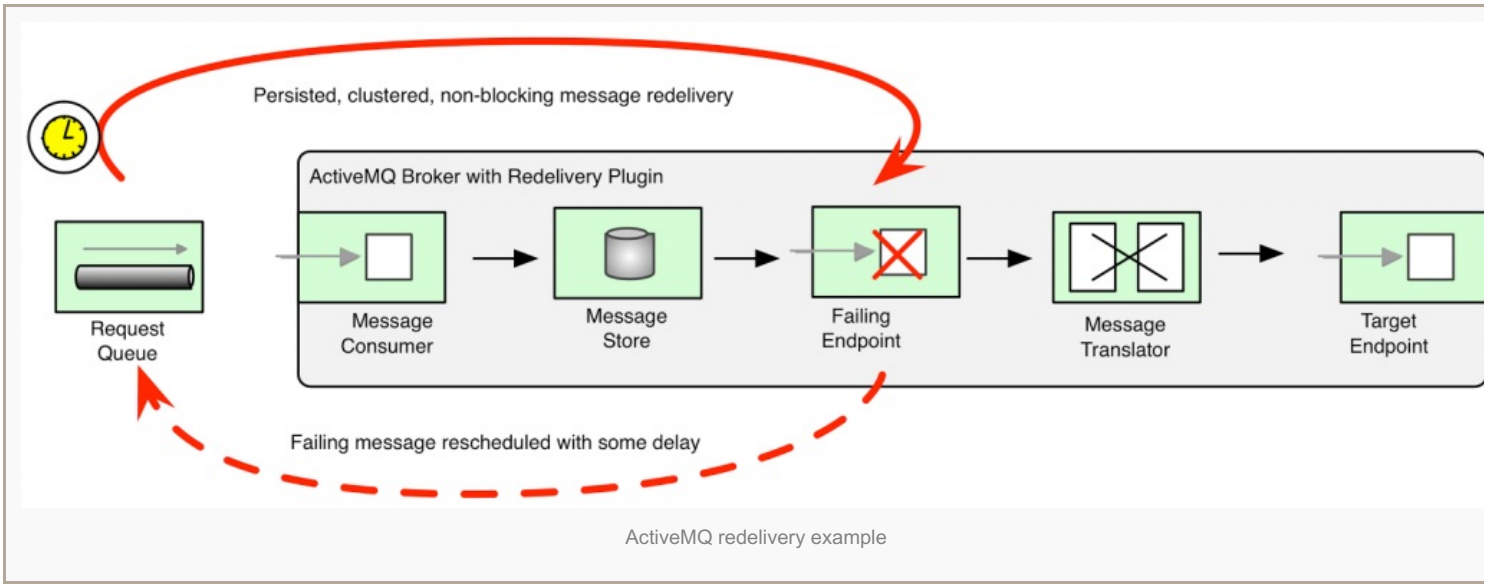
The key differentiator of the retry mechanism is that Camel error handling logic will not retry the whole route, but it will

retry only the failed endpoint in the processing flow. This is achieved thanks to the channels that connect the endpoints in the Camel route. Whenever an exception is thrown up by the processing node, it is propagated back and caught by the channel, which can then apply various error handling policies. Another important difference here is that Camel-based error handling and redelivery logic is in-memory, and it blocks a thread during retries, which has consequences. You may run out of threads if all threads are blocked and waiting to do retries. The owner of the threads may be the consumer, or some parallel processing construct with a thread pool from the route (such as a parallel splitter, recipient list, or Threads DSL). If, for example, we have an HTTP consumer with ten request processing threads, a database that is busy and rejects connections, and a RedeliveryPolicy with exponential backoff, after ten requests all the threads will end up waiting to do retries and no thread will be available to handle new requests. A solution for this blocking of threads problem is opting for
asyncDelayedRedelivery where Camel will use a thread pool and schedule the redelivery asynchronously. But the thread pool stores the redelivery requests in an internal queue, so this option can consume all of the heap very quickly. Also keep in mind that there is one thread pool for all error handlers and redeliveries for a CamelContext, so unless you configure a specific thread pool for long-lasting redelivery, the pool can be exhausted in one route and block threads in another. Another implication is that because of the in-memory nature of the retry logic, restarting the application will lose the retry state, and there will be no way of distributing or persisting this state.

Overall, this Camel retry mechanism is good for short-lived local retries, and to overcome network glitches or short locks on resources. For longer-lasting delays, it is a better option to redesign the application with persistent redeliveries that are clustered and non-thread-blocking (such a solution is described below).

ActiveMQ Broker Redelivery (Long Retry)

This retry mechanism has different characteristics to the previous two since it is managed by the broker itself (rather than the message consumer or the Camel routing engine). ActiveMQ has the ability to deliver messages with delays thanks to its scheduler. This functionality is the base for the broker redelivery plug-in. The redelivery plug-in can intercept dead letter processing and reschedule the failing messages for redelivery. Rather than being delivered to a DLQ, a failing message is scheduled to go to the tail of the original queue and redelivered to a message consumer. This is useful when the total message order is not important and when throughput and load distribution among consumers is.



ActiveMQ redelivery example

*Side note – I know, shameless plug, but I'm pretty excited about my book on this topic. You can check it out here at a 40% discount until end of June! And hope you like it.*The difference to the previous approaches is that the message is persistent in the broker message store and it would survive broker or Camel route restart without affecting the redelivery timings. Another advantage is that there is no thread blocked for each retried message. Since the message

is returned back to the broker, the Competing Consumers Pattern can be used to deliver the message to a different consumer. But the side effect is that the message order is lost as the message will be put at the tail of the message queue. Also, running the broker with a scheduler has some performance impact. This retry mechanism is useful for long-delayed retries where you cannot afford to have a blocked thread for every failing message. It is also useful when you want the message to be persisted and clustered for the redelivery.

Notice that it is easy to implement the broker redelivery logic manually rather than by using the broker redelivery plug-in. All you have to do is catch the exception and send the message with an AMQ_SCHEDULED_DELAY header to an intermediary queue. Once the delay has passed, the message will be consumed and the same operation will be retried. You can reschedule and process the same message multiple times until giving up and putting the message in a backoff or dead letter queue.

Reference: Short Retry vs Long Retry in Apache Camel from our JCG partner Bilgin Ibryam at the OFBIZian blog.