Building Reliable Reprocessing and Dead Letter Queues with Apache Kafka

In distributed systems, retries are inevitable. From network errors to replication issues and even outages in downstream dependencies, services operating at a massive scale must be prepared to encounter, identify, and handle failure as gracefully as possible.

Given the scope and pace at which Uber operates, our systems must be fault-tolerant and uncompromising when it comes to failing intelligently. To accomplish this, we leverage Apache Kafka, an open source distributed messaging platform, which has been industry-tested for delivering high performance at scale.

Utilizing these properties, the Uber Insurance Engineering team extended Kafka's role in our existing event-driven architecture by using non-blocking request reprocessing and dead letter queues (DLQ) to achieve decoupled, observable error-handling without disrupting real-time traffic. This strategy helps our opt-in Driver Injury Protection program run reliably in more than 200 cities, deducting per-mile premiums per trip for enrolled drivers.

In this article, we highlight our approach for reprocessing requests in large systems with real-time SLAs and share lessons learned.

Working in an event-driven architecture

The backend of Driver Injury Protection sits in a Kafka messaging architecture that runs through a Java service hooked into multiple dependencies within Uber's larger microservices ecosystem. For the purpose of this article, however, we focus more specifically on our strategy for retrying and dead-lettering, following it through a theoretical application that manages the pre-order of different products for a booming online business.

In this model, we want to both a) make a payment and b) create a separate record capturing data for each product pre-order per user to generate real-time product analytics. This is analogous to how a single Driver Injury Protection trip premium processed by our program's back-end architecture has both an actual charge component and a separate record created for reporting purposes.

In our example, each function is made available via the API of its respective service. Figure 1, below, models them within two corresponding consumer groups, both subscribed to the same channel of pre-order events (in this case, the Kafka topic `PreOrder`):

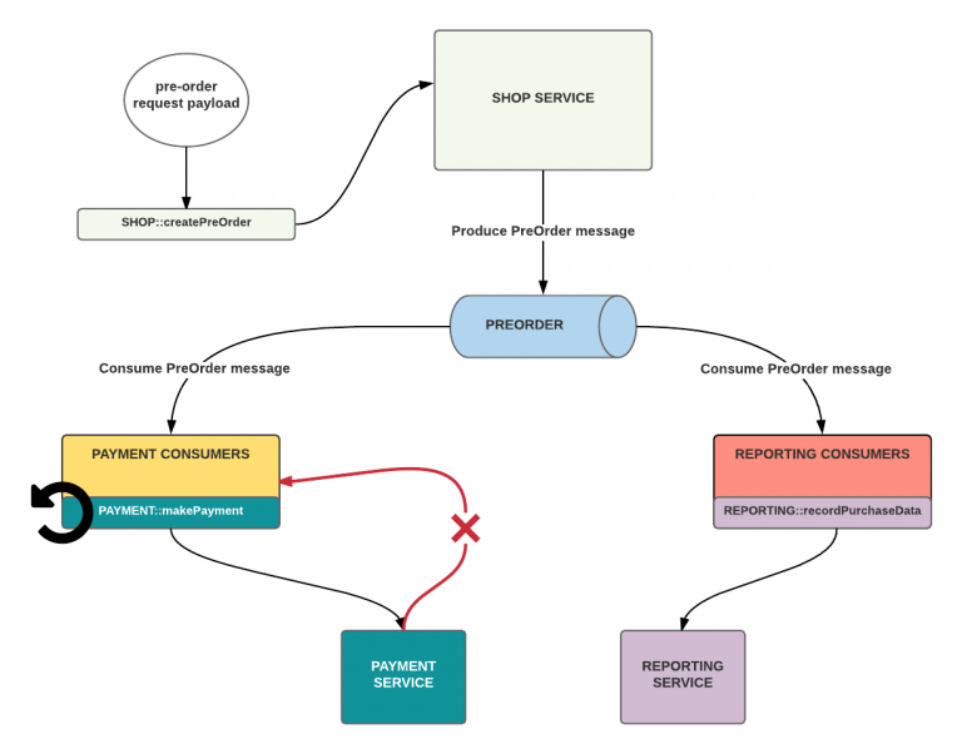

Figure 1: When a pre-order request is received, `Shop Service` publishes a `PreOrder` message containing relevant data about the request. From there, each of the two sets of listeners reads the produced event to execute its own business logic and call its corresponding service.

A quick and simple solution for implementing retries is to use a feedback cycle at the point of the client call. For example, if the Payment Service in Figure 1 is experiencing prolonged latency and starts throwing timeout exceptions, the shop service would continue to call `makePayment` under some prescribed retry limit—perhaps with some backoff strategy—until it succeeds or another stop condition is reached.

The problem with simple retries

While retrying at the client level with a feedback cycle can be useful, retries in large-scale systems may still be subject to:

- **Clogged batch processing**. When we are required to process a large number of messages in real time, repeatedly failed messages can clog batch processing. The worst offenders consistently exceed the retry limit, which also means that they take the longest and use the most resources. Without a success response, the Kafka consumer will not commit a new offset and the batches with these bad messages would be blocked, as they are re-consumed again and again, as illustrated in Figure 2, below.
- **Difficulty retrieving metadata.** It can be cumbersome to obtain metadata on the retries, such as timestamps and *nth* retry.

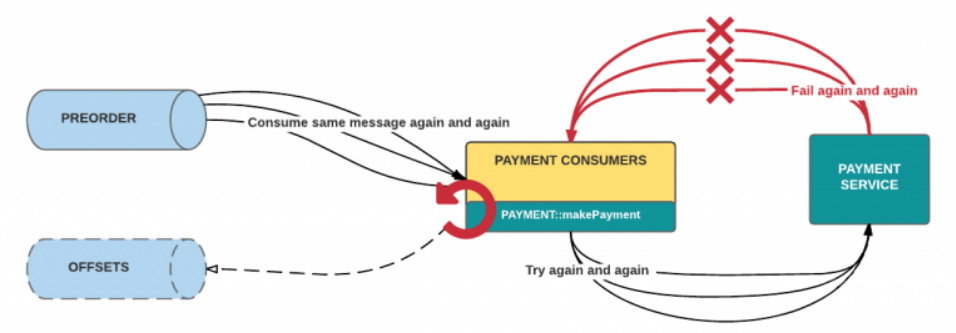

Figure 2: If there is a breaking change in the downstream Payment Service, for instance, unexpected charge denial for previously valid pre-orders, then these messages would fail all retries. The consumer that

received that specific message does not commit the message's offset, meaning that this message would be consumed again and again at the expense of new messages that are arriving in the channel and now must wait to be read.

If requests continue to fail retry after retry, we want to collect these failures in a DLQ for visibility and diagnosis. A DLQ should allow listing for viewing the contents of the queue, purging for clearing those contents, and merging for reprocessing the dead-lettered messages, allowing comprehensive resolution for all failures affected by a shared issue. At Uber, we needed a retry strategy that would reliably and scalably afford us these capabilities .

Processing in separate queues

To address the problem of blocked batches, we set up a distinct retry queue using a separately defined Kafka topic. Under this paradigm, when a consumer handler returns a failed response for a given message after a certain number of retries, the consumer publishes that message to its corresponding retry topic. The handler then returns true to the original consumer, which commits its offset.

Consumer success is redefined from a successful handler response, meaning zero failure, to the establishment of a conclusive result for the consumed message, which is either the expected response or its placement elsewhere to be separately handled.

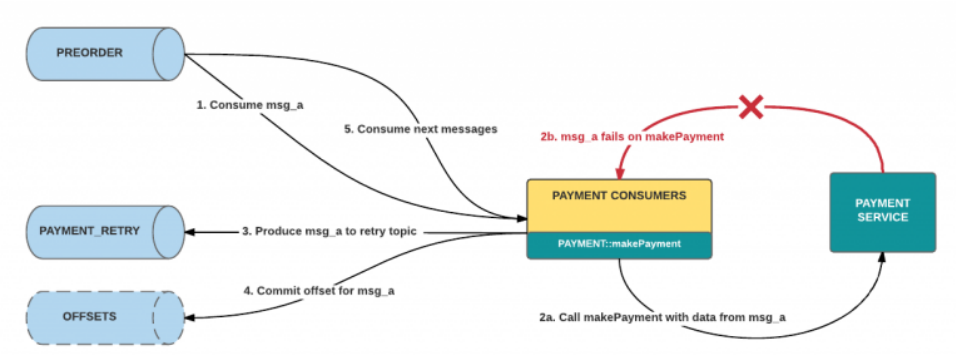

Figure 3: msg_a is an example message whose consumer receives an error response when handling it. This consumer publishes `msg_a` to the `payment_retry` queue and then commits `msg_a`'s offset relative to pre-orders, the original processing topic. With the consumption of `msg_a` complete, the consumer moves onto the next message.

Retrying requests in this type of system is very straightforward. As with the main processing flow, a separate group of retry consumers will read off their corresponding retry queue. These consumers behave like those in the original architecture, except that they consume from a different Kafka topic. Meanwhile, executing multiple retries is accomplished by creating multiple topics, with a different set of listeners subscribed to each retry topic. When the handler of a particular topic returns an error response for a given message, it will publish that message to the next retry topic below it, as depicted in Figures 3 and 4.

Finally, the DLQ is defined as the end-of-the-line Kafka topic in this design. If a consumer of the last retry topic still does not return success, then it will publish that message to the dead letter topic. From there, a number of techniques can be employed for listing, purging, and merging from the topic, such as creating a command-line tool backed by its own consumer that uses offset tracking. Dead letter messages are merged to re-enter processing by being published back into the first retry topic. This way, they remain separate from, and are unable to impede, live traffic.

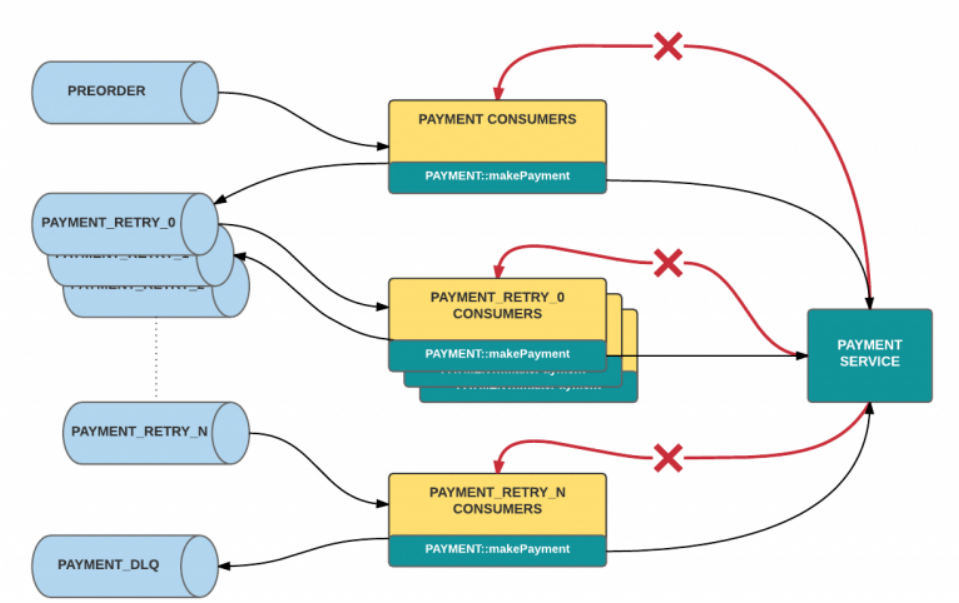

Figure 4: Errors trickle down levels of retry topics until landing in the DLQ.

It is important not to simply re-attempt failed requests immediately one after the other; doing so will amplify the number of calls, essentially spamming bad requests. Rather, each subsequent level of retry consumers can enforce a processing delay, in other words, a timeout that increases as a message steps down through each retry topic. This mechanism follows a leaky bucket pattern where flow rate is expressed by the blocking nature of the delayed message consumption within the retry queues. Consequently, our queues are not so much retry queues as they are delayed processing queues, where the re-execution of error cases is our best-effort delivery: handler invocation will occur at least after the configured timeout but possibly later.

What we gain with queue-based reprocessing

Now, we discuss the benefits of our described approach as it relates to ensuring reliable and scalable reprocessing:

Unblocked batch processing

Failed messages enter their own designated channels, enabling successes in the same batch to proceed instead of requiring them to be reprocessed along with the failures. Thus, consumption of incoming requests moves forward unblocked, achieving higher real-time throughput.

Decoupling

Independent work streams that operate on the same event each have their own consumer flows, with separate reprocessing and dead letter queues. Failure in one dependency does not require retrying that particular message for others that had succeeded. For example, in Figure 1, if reporting had errored out, but payment had succeeded, only the former would need to be re-attempted and potentially dead-lettered.

Configurability

Creating new topics incurs practically no overhead, and the messages produced to these topics can abide by the same schema. The original processing, along with each of the retry channels, can be managed under an easily-written, higher-level consumer class, governed by config when it comes to which topic name to read and to publish to (in the event of failure), as well as the length of the enforced delay before executing an instance's handler.

We can also differentiate treatment of different types of errors, allowing cases such as network flakiness to be re-attempted, while null pointer exceptions and other code bugs should go straight into the DLQ because retries would not fix them.

Observability

Segmentation of message processing into different topics facilitates the easy tracing of an errored message's path, when and how many times the message has been retried, and the exact properties of its payload. Monitoring the rate of production into the original processing topic compared to those of the reprocessing topic and DLQ can inform thresholds for automated alerts and tracking real-service uptime.

Flexibility

Though Kafka itself is written in Scala and Java, Kafka supports client libraries in several languages. For example, many services at Uber use Go for their Kafka client.

Kafka message formatting with a serialization framework like Avro supports evolvable schemas. In the event that our data model needs to be updated, minimal rejiggering is required to reflect this change.

Performance and dependability

Kafka offers at-least-once semantics by default. This durability guarantee is highly valuable in the context of fault-tolerance and message failure; when it comes to delivering business-critical data (as in Uber's case), message losslessness is paramount. Moreover, Kafka's parallelism model and pull-based system enable high throughput and low latency.

Other considerations

Since Kafka only guarantees in-order processing within partitions and not across them, it must be acceptable for an application to handle events outside of the exact order in which they occur. Furthermore, at-least-once message delivery necessitates consumer dependency idempotency, a common feature of any distributed system.

The advantages outlined in the previous section offer significant benefits, but mileage and implementation may vary by use case. For instance, depending on how many data types a given application handles, a set of topics for each work stream of each event type could result in a large number of topics to manage. In such a case, an alternative to our count-based queues might be to wrap the event type with additional fields, thereby tracking retry count and timestamp in a more manageable way. This tradeoff would require some reconsideration to how scheduling is performed, as this was managed through the queue ladder.

Moving forward

Using count-based Kafka topics as separate reprocessing and dead lettering queues enabled us to retry requests in an event-based system without blocking batch consumption of real-time traffic. Within this framework, engineers can configure, grow, update, and monitor as needed without penalty to developer time or application uptime.

The design described in this article sits behind our Driver Injury Protection program, launched as part of Uber's 180 Days of Change campaign. If you are interested in building the reliable and scalable systems behind this service and others for our drivers, consider applying for a role on our team!

Comments