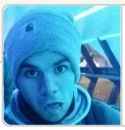
yelp Engineering

Streams and Monk – How Yelp is Approaching Kafka in 2020



Flavien Raynaud, Software Engineer
Jan 22, 2020

We launched our very first Kafka cluster at Yelp more than five years ago. It was not monitored, did not expose any metrics, and we definitely did not have anyone on call for it. One year later, Kafka had already become one of the most important distributed systems running at Yelp, and today has become one of the core components of our infrastructure.

Kafka has come a long way since the 0.8 version we were running back then, and our tooling (some of it open-source) has also significantly improved, increasing reliability and reducing the amount of operational work required to run our clusters. We can now scale clusters with a single configuration push, load balance and decommission brokers, automatically trigger rolling-restarts to pick up new cluster configuration, and more. Our production Kafka clusters transport billions of messages every day, with little to no operational load on the team.

While most of the improvements to our streaming infrastructure have been gradual and incremental, we recently went through a major redesign that significantly changed the way we manage our clusters and how we interact with them.

Why the Change?

One of the core values of Yelp infrastructure engineers is developer velocity: developers should be able to use the technology we provide with as little effort as possible. Providing Kafka as a service by hosting and maintaining Kafka clusters that other teams could directly access was our first approach, and allowed us to quickly power many critical use cases.

However, one of the first issues we encountered with this approach was that not everyone is (or should be) a Kafka expert. Configuring producers, consumers, and topics to get exactly the reliability and delivery semantics that developers want imposes a significant load on any team using Kafka. Additionally, the right configuration is a moving target, as new parameters are constantly being added and new best practices discovered.

Upgrading Kafka has also proved to be a challenging endeavour, especially with hundreds of

services–spread across different client library versions and different languages–depending on it.

Our deployment also had a few significant obstacles. We had one main cluster per geographical region, where the majority of our topics were stored, and while this approach worked well initially by simplifying discovery, we soon ran into its limitations:

- Clusters were getting big and difficult to maintain. Operations such as rolling restarts and computation of the best partition allocation for cluster balancing became increasingly slow.
- A very wide variety of critical and non-critical topics were sharing the same resources. Information on the criticality of topics was not always available, making it hard to understand the impact of having some of those topics degraded during incidents.

It eventually became clear that giving direct access to Kafka clusters and configuration was not going to scale well in terms of data volume, developers, and clusters. Some fundamental changes had to be made.

Know Your Streams

To increase isolation between clients and clusters, we decided to create a level of abstraction in the middle that we call a **logical stream**. Here is a high level comparison between topics and logical streams:

Topic	Logical Stream
Lives in a specific cluster	Cluster-agnostic
Low-level topic configuration	High-level stream properties
Low-level consumer/producer configuration	Producers and consumers configured automatically

The most important aspect of the new abstraction is that we now know what high-level properties a stream needs. Given these properties, everything else falls into place. So, what are they? We have identified a few that can be used to classify streams and will discuss the two most important ones, *policy* and *priority*, in the next section.

Policy

All streams are not created equal: some favor consistency over availability and are willing to trade performance for certainty of delivery. As a hypothetical example, consider a food ordering scenario where Kafka is used to store transactions: the backend service that takes care of handling the order needs to be sure that the order transaction was securely stored for later processing. We definitely do not want our customers getting “hangry” while waiting for an order that will never arrive! If Kafka is not able to reliably store the message, we would rather show an error message

to the user than fail silently. We call this category of streams “acked.”

The other big category contains streams that favour availability over consistency, where performance is more important than being able to react to failure. As a simple example, think of a user checking in to a local business: if Kafka is not available, we want to be able to store that event somewhere and flush it to Kafka when it recovers. If Kafka cannot receive data, there is not much the producer code can do other than retrying, and we definitely do not want to block the user until the click has been registered. We call these streams “fire and forget.”

Policy	Available vs. Consistent	Result Callback	Enqueue Latency
Acked	Consistent	Yes	~50 ms
Fire and Forget	Available	No	~100 µs

Priority

Another fundamental property of a stream is its business criticality, in terms of both impact on revenue and user experience. For example, consider two streams: user check-in and debug logs from an experimental service. They both belong to the “fire and forget” policy, but data loss or delay on the former has a much bigger impact than on the latter. We define the check-in stream as *high priority*, while the debug log would be *low priority*.

Higher priority streams are automatically allocated to clusters with better SLOs (Service-level Objectives). Since providing better SLOs is more expensive (higher provisioning, isolated ZooKeeper clusters, etc.), knowing which streams are actually high priority allows us to allocate resources more efficiently. Having priorities also means that if one cluster is unavailable, we know what kind of stream will be impacted and can react accordingly.

In the new architecture, we group clusters according to the properties of the streams they handle. Here is a (very simplified) graphic representation of our clusters:

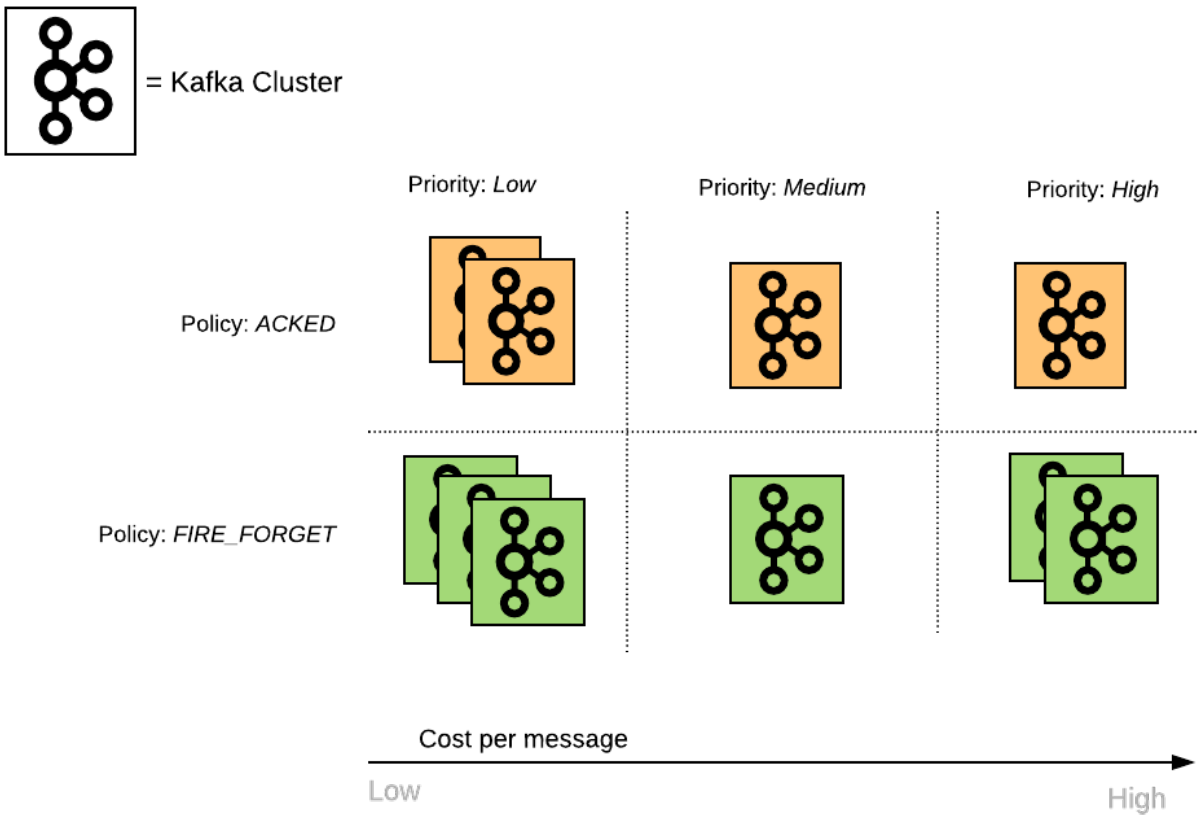


Figure: Representation of clusters, based on properties of the topics they host

Having multiple clusters within the same group enables us to scale horizontally without risking ending up with huge and difficult-to-maintain clusters. Additionally, since clusters are similar to one another, rolling out new changes is safer.

Accessing Streams

So now that we are more familiar with our streams and how they are mapped to our clusters: how does a client actually produce to or consume from them? First of all, a stream must be declared; this can happen programmatically or via a configuration file. Our streaming infrastructure is schema-centric, and has been extended to allow stream configuration to be included as part of a schema declaration. An example configuration could be:

```
schemas:
  - name: user_checkins
    schema: <schema definition>
    policy: fire_forget
    priority: high
    owner: userx_team
```

When a client wants to access a logical stream for production, it sends a request to the Stream Discovery and Allocation service (SDAA - this is actually tightly coupled to our schematizer service), which will return the coordinates of a specific topic that transports that stream, together with the configuration that the producer should use.

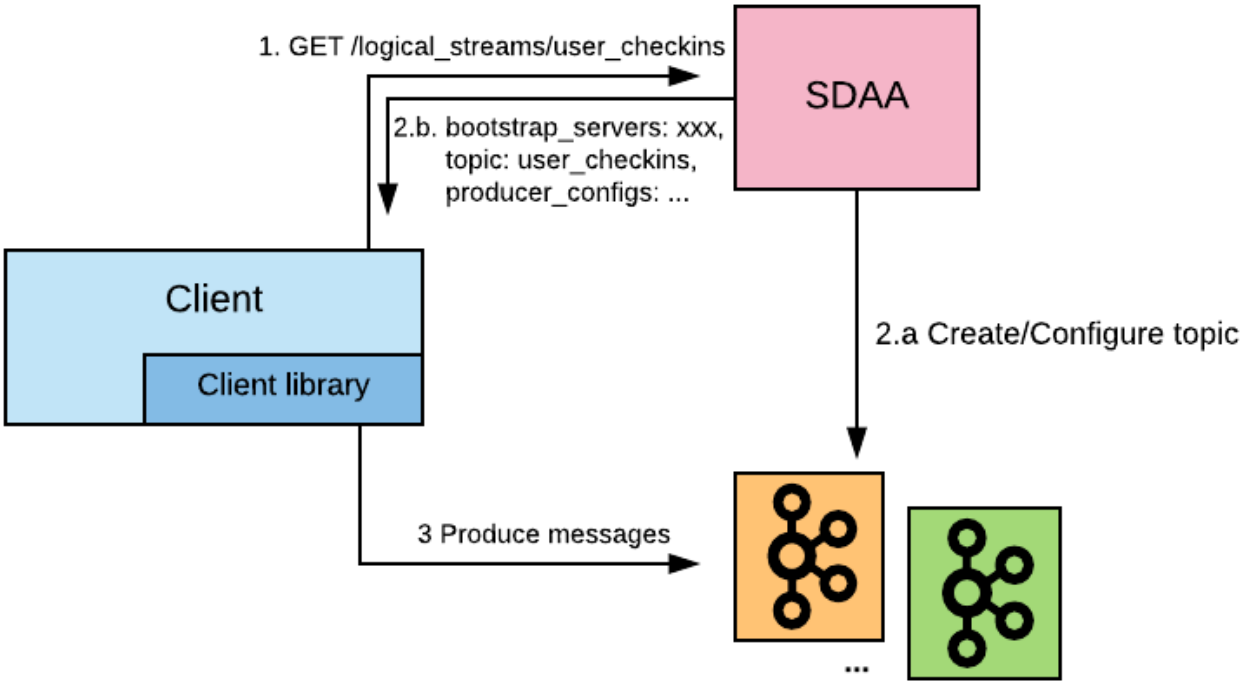


Figure: Interactions between clients and the SDAA service

Once the developer declares which policy and priority they want for their stream, the system has all the information required to allocate it in the appropriate cluster and configure it. Producers and consumers will automatically pick up this information and configure themselves accordingly.

Abstract all the things!

While this stream abstraction has allowed us to reduce the cognitive load on developers using Kafka for their streaming use cases, it could also add complexity to the client libraries provided to said developers. Yelp operates hundreds of services, most of them relying on streaming or logging in some way, so operations such as upgrading Kafka (some with breaking changes) can generate enormous migration costs.

For many years, Yelp has been using Scribe as its log ingestion pipeline, with an in-house aggregator called Sekretar (more details are available in the Scaling the Yelp's logging pipeline with Apache Kafka presentation). The Scribe model has made management of our logging

infrastructure easier; developers only need to write to a Scribe leaf, and need not worry about Kafka internals. Upgrading Kafka is also seamless for developers: Sekretar is the only entrypoint to these Kafka clusters and is thereby the only part of the system that has to incorporate Kafka configuration changes and upgrades.

We decided to take this approach one step further, using this model for all Kafka interactions at Yelp, and abstracting production away from developers and into our new unified stream ingestion pipeline: **Monk**.

From a high-level perspective, Monk is made of two main components: the Monk leaf, and the Monk relay.

A Monk leaf is a daemon that runs on almost every host at Yelp (mostly PaaSTA hosts). This daemon is responsible for handling all streaming traffic generated by the host, PaaSTA services, and other system daemons, trying to ensure the traffic eventually makes it to Kafka within a reasonable time.

Monk leaves speak the Thrift protocol, and listen on the same port on every host, removing the burden of discovery for developers. For most use cases, the Thrift sockets are blocking the caller until Monk returns an acknowledgement (or times out).

As mentioned above, streams can choose between two policies: *ACKED* and *FIRE_FORGET*, which have different latency and delivery guarantees. The information on stream policies is available within our Stream Discovery and Allocation (SDAA) service, which Monk integrates with to determine which “route” events from a stream should take.

Let’s dive into how Monk handles each of these use cases.

Acked Policy

When a Monk leaf requests metadata about a stream to the SDAA service, it gains information about the policy and Kafka cluster it should produce to. Events are enqueued to an in-memory queue and a Kafka producer is responsible for picking events up from the queue and sending them to Kafka. Once Kafka acknowledges these events, the Monk leaf returns its acknowledgement to the caller.

Fire and Forget policy

The above scenario involves blocking callers until Kafka acknowledges events that need to be produced. *FIRE_FORGET* traffic does not need to wait for Kafka to acknowledge every single event, trading this acknowledgement for an enormous reduction of the time spent being blocked by Monk. For some use cases this is essential: imagine a 50ms latency hit per logging event generated while serving a web request!

When a Monk leaf receives events for *FIRE_FORGET* streams, they are immediately enqueued to a buffer and an acknowledgement is sent to the caller. The leaf then uses a hybrid buffer, where

each stream gets up to 1MB of in-memory space to avoid overloading disks, then falls back to buffering on disk to keep memory usage low. In order to increase isolation between streams, each stream gets its own buffer, decreasing the impact of noisy neighbours.

A pool of egress worker threads then picks up events from the buffer (in round-robin fashion, to increase fairness) and sends them onto the next stage.

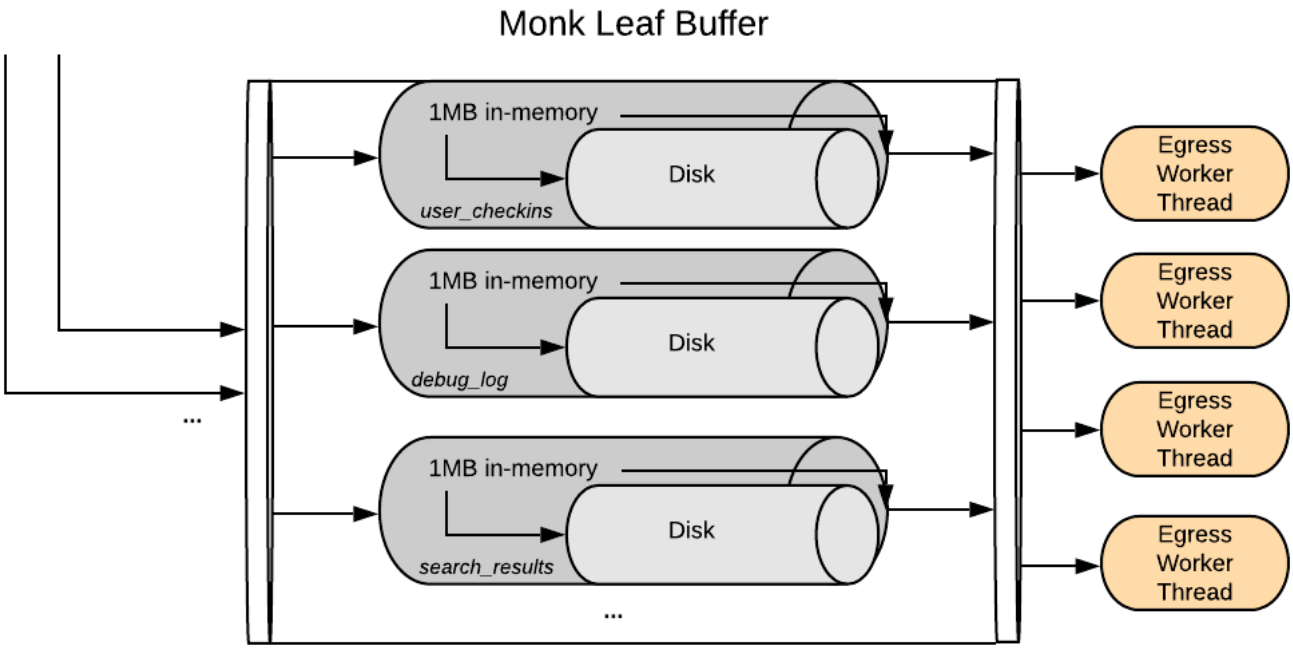


Figure: Monk leaf hybrid buffer

This is where Monk relays come into the picture (as replacements for the Sekretar service).

The Monk leaf daemon can run on any host at Yelp across heterogeneous hardware: large instances, small instances, dedicated EC2 instances, spot fleet EC2 instances, SSD-backed instances, HDD-backed instances, etc. The team responsible for Monk has limited control over these hosts.

For this reason (among others, see below), we intend to move all streaming traffic off of leaf hosts and onto Monk relays as fast as possible.

Monk relays act as physical buffers for *FIRE_FORGET* events that are waiting to be flushed into their Kafka cluster. Even though Kafka has proven reliable, it is still a system that can fail and have periods of unavailability. In case of a Kafka outage, data is buffered inside the Monk relays until availability in Kafka is recovered. This has proven to be useful for traffic from leaves running on EC2 Spot instances, which only have two minutes of notice before being terminated, losing any data not stored elsewhere. Monk relay clusters can easily be scaled up or down to withstand

varying loads.

Buffers in a Monk relay use a similar design as buffers in a Monk leaf, only without the memory component. Every event is appended to a buffer on disk. Not relying on memory for the buffer increases the resilience of Monk relays so that in case the process is restarted/killed, little to no data is lost. Each stream gets its share of the buffer to increase isolation.

Monk relays are also integrated with the SDAA service, which is used to determine the destination Kafka cluster and producer configurations.

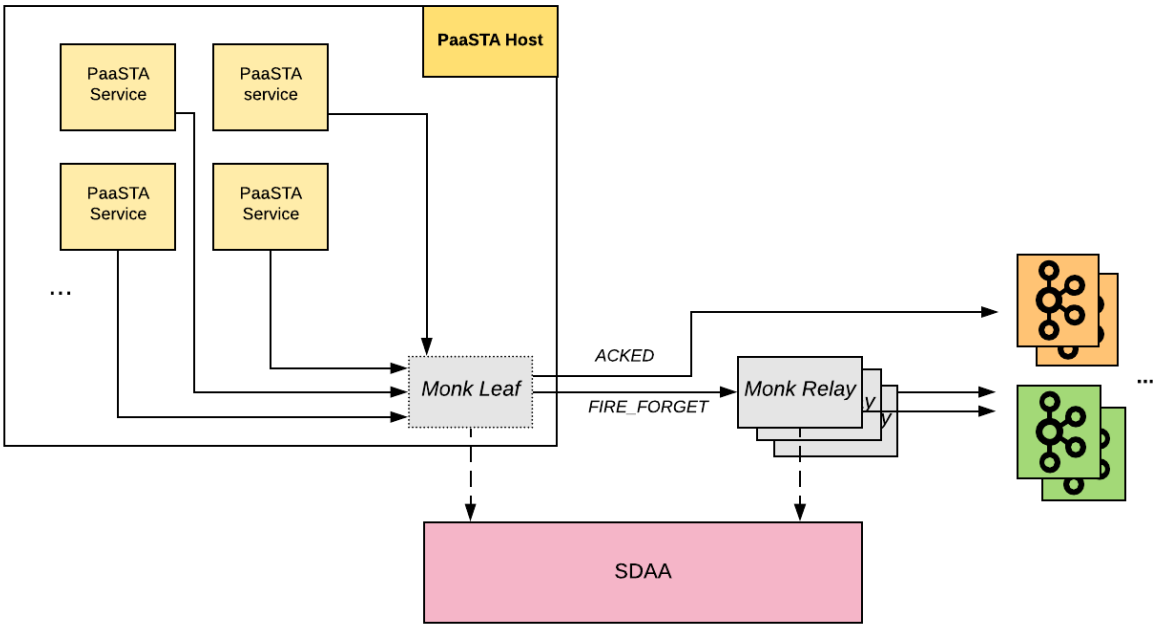


Figure: High-level Monk architecture

Interactions

For an engineer wishing to publish events and have them eventually loaded into Kafka, the process is as simple as sending Thrift-formatted bytes to the right port on `localhost`. Our team provides client libraries for languages with a wide adoption within Yelp (Python, Java/Scala).

```
producer = MonkProducer(client_id="monk-example")

producer.send_messages(stream="user_checkins", messages=[user_checkin])

results = producer.send_messages(
    stream="online_orders",
    messages=[online_order],
    timeout_ms=300,
)
```

Client libraries also handle retries (keeping track of timeouts) and in-memory buffering if the daemon is temporarily unavailable (e.g., if it is restarting). As you may have noticed, sending messages to Monk requires just three or four high-level parameters, a good improvement over the dozens available for a regular KafkaProducer. Cluster discovery is handled automatically by Monk and the SDAA service.

The vast majority of traffic handled by Monk uses the *FIRE_FORGET* policy. Even if an acknowledgement from Monk is not a guarantee that events will always make it to Kafka, the lower latency guarantee makes it very appealing.

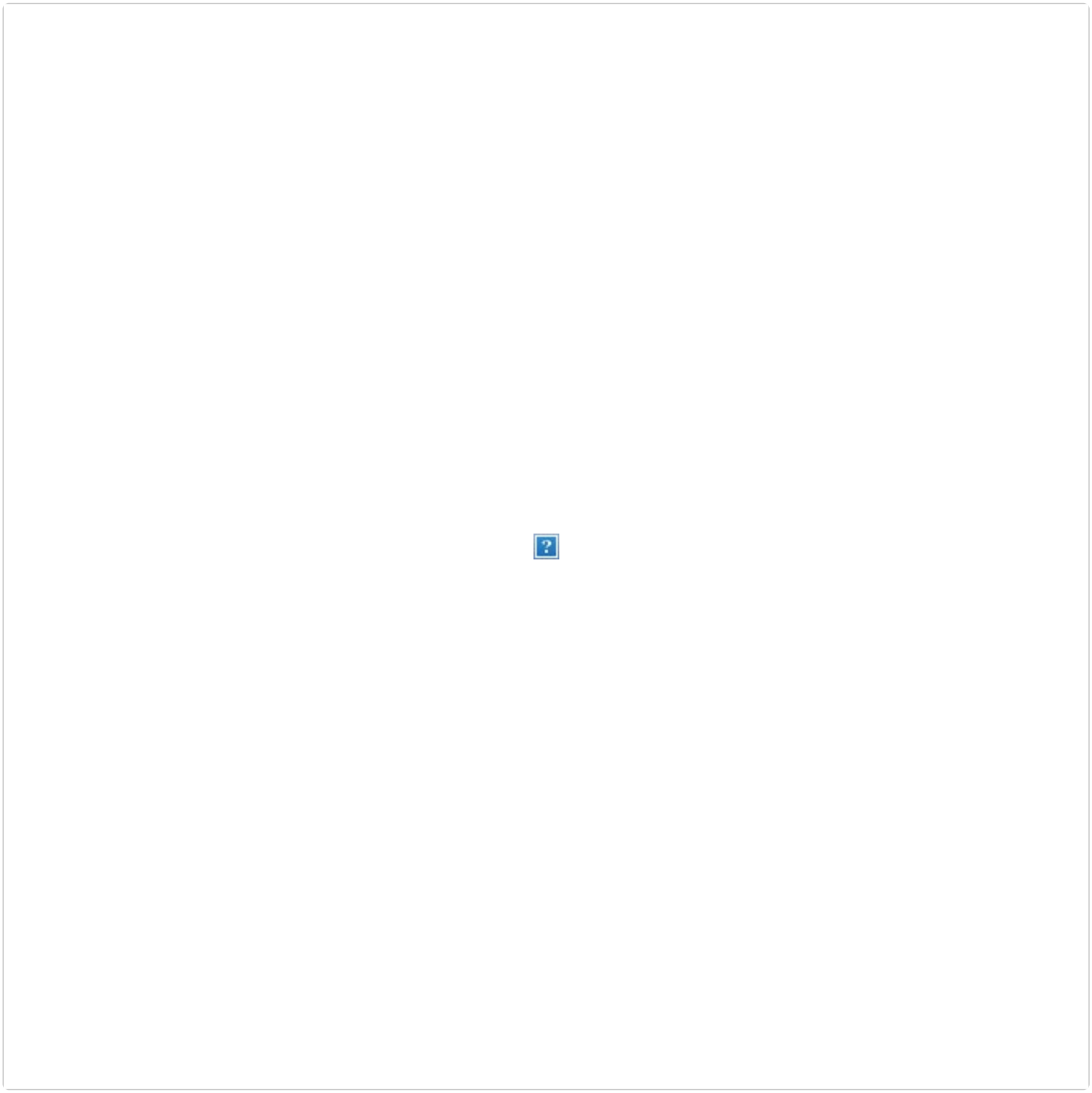


Figure: m5.4xlarge EC2 instance - 1 KiB events for 120 seconds (Open in New Tab for more details)

We are also working on a service to measure the amount of data lost per stream, and exposing it to developers. Stay tuned for a future blog entry!

No Pain, No Gain

Developing, deploying, and adopting Monk was not easy. Large infrastructure changes require ~~blood, sweat, and tears~~ coordination across teams, extensive testing, and countless hours of debugging.

During the initial rollout of some of our high-value streams, events were sent to both Scribe and Monk. We compared the events in each pipeline before moving forward. The first results showed that Monk was producing fewer events than Scribe, indicating data loss. We quickly observed a very high number of connection and production timeouts from the Monk thrift sockets.

The initial version of the Monk server was synchronous all the way to writing events to disk. When events were published to a *FIRE_FORGET* stream, the caller was blocked until just before events were appended to our disk buffer in the leaf. This meant that, even if we were not actually blocking callers, Monk server threads were blocked until events were (successfully or not) written to disk. Recall that Monk leaves run on varied hardware: we were at the mercy of page cache flush, or just increased I/O from whatever else was running on the hardware. During this blocking, Monk server threads were unable to accept incoming events from other callers, which were then timing out. Monk leaves now do all of their I/O in separate threads, and clients no longer see timeouts.

Once the initial rollout was complete, we noticed Monk leaves using a lot of disk resources on each host, sometimes preventing other processes from using the disk (e.g., due to the limit on the number of IOPS in AWS EBS). This was the main motivation for introducing a 1MB in-memory buffer for each stream in Monk leaves. This has proven to be sufficient in reducing the I/O usage of Monk leaves, as most streams flowing through Monk leaves rarely have to rely on the disk buffer.

KafkaProducer instances buffer events in memory in order to batch them before sending or retrying, thereby increasing throughput. One of the aims of Monk is to provide (as much as possible) isolation between streams. Using more Kafka clusters is one way of achieving this, but does not solve the issue of noisy neighbours within the same `(policy, priority)` guarantees. Consider a spike of events coming from a single stream (e.g., a service entering a crash loop and logging exceptions): if this happens, Kafka may not be able to immediately handle the traffic increase. Timeouts start occurring and events start accumulating in the buffer of the specific KafkaProducer, preventing events from other streams with the same guarantees from being appended to the buffer. So much for isolation!

To circumvent this scenario, we use a simple logic before sending events to Kafka: limit the number of bytes “in-flight” per stream. Whenever an event is picked up from the buffer, the “in-

flight" bytes counter for the corresponding stream is increased. When the event is acknowledged by Kafka, the counter is decreased. If a stream has too many "in-flight" bytes (this can be configured on a per-stream basis, and reloaded at runtime), events will not be picked up from the buffer until Kafka acknowledges the in-flight events. This sliding window logic is simple, easy to comprehend, and has been sufficient in avoiding Kafka buffer-related issues for now.

Producing Directly to Kafka

ACKED production in Monk leaves is configured for low-latency (`linger.ms` is usually set to 1) but relatively low throughput, while *FIRE_FORGET* production is configured for high throughput but may result in data loss without a guaranteed ordering. While MonkProducer is the solution to most streaming use cases we have, specific applications still require something extra. The output of ETL jobs is a good example of such an application; it requires high throughput and rarely tolerates out-of-order events. Instead of banning direct production to Kafka, we integrated the SDAA service with a few supported client libraries, such as Flink and Spark KafkaProducers, so that they follow the guidelines of our Data Pipeline.

Consumers

Running smaller Kafka clusters reduced the burden on infrastructure engineers, but this change would not be complete without a way for consumers to read from multiple clusters. Most "regular" Kafka consumers are able to consume from a single cluster, but what about the other multi-cluster use cases (e.g., joining topics of different policies/priorities or aggregation across regions)?

Yelp has widely adopted Apache Flink for stream processing use cases, as it can easily consume events from multiple Kafka clusters. Instead of writing an equivalent of Monk for Kafka consumers, we chose to treat Flink as our abstraction for consumers. Since our Flink consumers are all integrated with the SDAA service this reduces the burden of configuration and cluster discovery on developers.

A more detailed explanation of the Flink infrastructure can be found in our presentation Powering Yelp's Data Pipeline Infrastructure with Apache Flink.

Looking Further

Because a proper blog post cannot end without a utopic future vision, let's have at it!

In the last few years, Kafka at Yelp has come a long way. From a single, unmonitored, three-broker cluster in production, to numerous clusters and production brokers, it has become the backbone of our data infrastructure. This wide adoption would not have been possible without scaling the infrastructure sitting on top of it. The introduction of Monk and the SDAA service has served our two principal goals: increasing developer velocity and making infrastructure more reliable. Our Data Pipeline is now integrated with the SDAA service with a good part of it powered by Monk. Developers do not need to worry about Kafka specifics, and infrastructure engineers feel

more empowered to operate on production clusters knowing that maintenance operations will be shorter and safer.

Shorter and safer operations are a big benefit, but automation is arguably a better one. Now that cluster operations are invisible to developers, why not have non-human operators perform them? The future includes automating these operations by making Kafka just another service run by PaaSTA (our platform as a service) on Kubernetes, reducing the load on infrastructure engineers.

Acknowledgements

- Brian Sang, Federico Giraud, Jonathan Zernik, Manpreet Singh, Piotr Chabierski & the Data Streams group as contributors;
- The fancy Latency Heat Maps from Brendan Gregg.

Back to blog