Streams and Monk – How Yelp is Approaching Kafka in 2020

Yelp established its first Kafka cluster five years ago. They didn't have somebody available to handle it, it wasn't monitored, it didn't reveal any analytics, and it wasn't monitored. Kafka had already established itself as one of Yelp's most critical distributed systems one year later, and it is now one of the basic components of our infrastructure. With a few exceptions, our Kafka clusters in production send billions of bytes of data every day. The squad is under little or no operational stress. There were a few major issues with our deployment as well. We had one primary cluster per geographical region where the majority of our topics were housed, and while this strategy worked well at first by making finding simpler, it proved to be ineffective over time. It soon became clear that it had limitations: clusters grew larger and more difficult to maintain. Rolling restarts and finding the optimal partition allocation for cluster balancing became increasingly time demanding. The same resources were used for both important and non-critical issues. Because information on the criticality of such concerns was not always accessible, it was impossible to assess the relevance of having some of them damaged during occurrences. Because Yelp has hundreds of services, the bulk of which rely on streaming or logging in some fashion, updates to Kafka (some with breaking changes) may incur considerable migration costs. While stream abstraction has lowered the cognitive load on developers who use Kafka for streaming use cases, it may also have a negative impact. Yelp has hundreds of services, the most of which rely on streaming or logging in some fashion, making updates to Kafka (some with breaking changes) costly. For an engineer who wants to publish events and have them eventually loaded into Kafka, it's as simple as sending Thrift-formatted bytes to the appropriate port on localhost. At Yelp, our team develops client libraries for frequently used languages (Python, Java/Scala). Client libraries also manage retries (keeping track of timeouts) and in-memory buffering if the daemon is temporarily unavailable (e.g., if it is restarting). As can be seen, sending messages to Monk just requires three or four high-level parameters, as opposed to the dozens available with a traditional Kafka Producer. Monk and the SDAA service are effective. Even when publishing events to disk, the original version of the Monk server was synchronous from start to finish. When events were published to a FIRE FORGET stream, the caller was stopped until just before events were added to our disk buffer in the leaf. This meant that even if we weren't blocking callers, Monk server threads were stopped until events were (successfully or unsuccessfully) written to disk. Because Monk leaves operate on a wide range of hardware, we were at the mercy of page cache flushes or just increased I/O from whatever else was running on the server. During this period of stalling, Monk server threads were unable to take incoming events from other callers, forcing them to time out.