Contrasting NATS with Apache Kafka

[Emil Koutanov](https://ekoutanov.medium.com/?source=post_page-----1d3bdb9aa767--------------------------------)

[Sep 2, 2020](https://itnext.io/contrasting-nats-with-apache-kafka-1d3bdb9aa767?source=post_page-----1d3bdb9aa767--------------------------------) · 6 min read

**TL;DR** Kafka is an *Event Streaming Platform*, while NATS is a closer to a conventional *Message Queue*. Kafka is optimised around the unique needs of emerging *Event-Driven Architectures*, which enrich the traditional pub-sub model with strong ordering and persistence semantics. Conversely, NATS is highly optimised around pub-sub topologies, and is an excellent platform for decoupling systems where message order and reliable delivery is a non-issue.

I’ll preface this post by pointing out that there is another product — **NATS Streaming** — which is a different beast and is closer to Kafka. You may want to take a detour to NATS Streaming if you after an alternative event streaming platform; otherwise, read on.

**Subscriptions**

At its core, **NATS is about publishing and listening for messages**. These depend heavily on **subjects** which scope messages into streams or topics. Consumers subscribe to topics either verbatim (matching the topic name precisely), or using wildcards. Below is an illustration of publisher-subject-consumer relationship in NATS.

Diagram

Description automatically generated

Credit: <https://nats.io/>

On the face of it, this isn’t diametrically opposed from Kafka, which also decouples producers and consumes by way of **topics**. However, their semantics differ. Kafka organises its topics into **partitions** — unbounded, totally ordered streams of **records** (Kafka’s substitute terminology for messages). A topic comprises one or more partitions, and exhibits **partial order**. (In other words, while records are totally ordered within their respective partition, their order across partitions is arbitrary.) This flexible arrangement makes **Kafka highly suited to applications where order matters**; for example, state machine replication, event sourcing, log shipping, log aggregation, SEDA (staged event- driven architecture) and CEP (complex event processing).

Speaking of topics, the equivalent NATS **subject** is a lightweight construct that is created automatically based on demand (subscriptions) and is pruned automatically when the demand ceases. NATS subjects are cheap to create, which makes them great for hierarchically organised data, allowing for a fine-grained subscription model. Anyone who’s used MQTT-style brokers (such as HiveMQ) should feel right at home with NATS. By comparison, Kafka’s topics are heavyweight entities that take time to spin up and lack the finesse that you get with NATS. Consequently, Kafka consumers must perform a lot of the requisite data filtering locally — consuming records from all assigned partitions and silently discarding those that are deemed irrelevant.

A Kafka partition is like an artery in a biological sense — drawing away from the source to feed downstream organs — *consumers*. A record has an **offset**, and it may have a **key** and a **value**; both are byte arrays and both are optional. A record’s key influences its ordering — records sharing a common key are guaranteed to occupy the same partition, and thus preserve their intrinsic order. The concept of partitions, records and offsets is illustrated below.

Diagram

Description automatically generated

Credit: <https://kafka.apache.org/documentation>

**Load balancing**

To further crystallise the differences between the two platforms, let’s consider how NATS and Kafka address *load balancing* — an essential characteristic of any message-oriented middleware.

NATS optionally balances message delivery across a group of subscribers which can be used to provide application fault tolerance and scale workload processing. To create a queue subscription, subscribers register a **queue name**. All subscribers with the same queue name form the corresponding **queue group**. As messages on the registered subject are published, *one member of the group is chosen randomly to receive the message*. Although queue groups may have multiple subscribers, each message is consumed by only one. The diagram below illustrates this.

Diagram

Description automatically generated

Credit: <https://nats.io/>

While NATS provides for fine-grained consumer scalability down to the message level, it does so at the expense of message ordering. Messages may be concurrently processed out-of-order at two or more disparate subscribers, making it unsuitable for order-sensitive applications. (Note: NATS Streaming addresses this, but as stated earlier, it is a different product in its own right.)

Kafka consumers subscribe to a topic as part of an encompassing **consumer group**. When the first consumer in a group joins the topic, it will receive all partitions in that topic. When a second consumer subsequently joins, it will get approximately half of the partitions, relieving the first consumer of half of its prior load. The process runs in reverse when consumers leave (by disconnecting or timing out) — the remaining consumers will absorb a greater number of partitions. So a consumer group balances the partition load; the more consumers you add, the fewer partitions each consumer receives. Adding more consumers than partitions will leave some consumers in an idle state; **Kafka will never assign a partition to multiple consumers in the same group**. So, although Kafka’s load balancing scheme is more coarse-grained than NATS’; it manages to preserve the order of records at the consumer nodes. The diagram below illustrates the relationship between producers, topics, partitions, consumers and consumer groups. Observe that consumer groups are logically isolated, from both a record flow and a load balancing perspective.

Diagram, schematic

Description automatically generated

**Delivery guarantees**

Yet another significant differentiator is *persistence*. **Kafka is a persistent datastore, offering at-least-once delivery semantics**. The act of reading a record by a consumer does not delete the record — it merely advances an internal pointer to the next record in the partition. This is called **committing** an offset. Should a consumer crash before it successfully processes a record, Kafka will re-deliver the last set of records (for which the offsets have yet to be committed). Because records are persisted for some time (subject to the configurable retention policy), consumers have the luxury of processing records that were published long before their tenure.

By contrast, NATS implements what is commonly referred to as **at-most-once delivery**. NATS strives to remain on and provide a ‘dial-tone’. However, if a subscriber drops out, it will not receive messages, as the basic NATS platform is a simple pub-sub transport system that offers only TCP-grade reliability. (This makes NATS a little different from more conventional MQ brokers, which tend to persist messages for the benefit of those subscribers that registered their interest prior to the point of publication, deleting messages after they have been delivered to all endpoints.) Simply stated, NATS isn’t designed to be used as a long-term event store; it is best used as a subscription-oriented message-centric transport layer, as opposed to a datastore.

**Operational concerns**

From an operational perspective, the differences are also pronounced. Kafka is a behemoth. Its deployment topology consists of a mixture of **Broker** and **ZooKeeper** nodes, with hundreds, if not thousands, of tuneable “knobs”, controlling all aspects of its behaviour. (And some are quite dangerous, in uneducated hands.)

A NATS cluster is much simpler in this regard, with a lot fewer parameters — unsurprising, given its lack of persistence.

**Summary**

So there you have it. The differences between the two should now be apparent. The points above do not aim to imply that one is better than the other; this is not an A *vs* B discussion. While it can be objectively stated that Kafka provides more overall flexibility by catering to a broader spectrum of messaging and eventing scenarios, it is also proportionally more complex to configure and maintain, and can be an overkill in some scenarios. NATS is a simpler solution — it’s a lot easier to get started with and operationalise. And let’s not forget, the latter is hugely important. Use the simplest platform that meets your present and anticipated needs, and is well-aligned to the current skill-sets prevalent in your organisation.