Synadia

Beyond streaming log processing

Comparing and contrasting NATS JetStream and Kafka

By Jean-Noël Moyne

V1.1

Contents

Introduction

Apache Kafka doesn't need any introduction and its undeniable popularity means it is often the default choice for anyone looking for a streaming platform. However, if you are reading this you know that the streaming platform landscape doesn't start and end with Kafka and that there are alternatives such as NATS JetStream that is easier to use, can be more efficient, and provides features that could be a better fit for your use cases.

NATS was a pure messaging system without persistence functionalities until the introduction of JetStream (leaving its deprecated predecessor 'STAN' aka "NATS Streaming" aside), which adds Kafka like functionality in addition to Key/Value and Object store capabilities. Because of the overlap in functionality between JetStream and Kafka, it is inevitable that the question of comparing the two gets frequently asked. This white paper explains some of the fundamental differences between the two and is focused on the differences, not the things they both do well (regardless of the cost required to meet throughput and latency requirements, see our **NATS.io TCO Report** for that). While it gets into some detail of some more subtle differences, it is not a definitive list. It is meant as a guide to help you identify where NATS with JetStream has features that Kafka doesn't have or requires you to use Apache Kafka Streams or to subscribe to the Enterprise version of Confluent Platform.

Summary

Apache Kafka is a distributed persistent log processing system designed for write throughput and at its core, that is its sole function. However, that core functionality is limited which is why the Confluent Platform includes other components to offer a more complete set of functionalities. If you want to do things such as: filtering of messages in topics, merging topics into other topics, use streams as Key/Value tables or as queues, then you will have to also use Apache Kafka Streams. This means that you need to run many more JVMs than just the Kafka brokers and Zookeepers, and that some of the data will be constantly shuffled back and forth between the Kafka broker and Kafka Streams JVMs and may end up being stored multiple times over multiple topics, affecting latency and efficiency and resource usage and, when using cloud services, can add noticeably to the costs (networking, storage).

In contrast, those above functionalities are already built into NATS JetStream (which leads to lower latencies and increased efficiency) along with real-time messaging, microservices, and an intrinsically immediately consistent distributed data store. All of this rolled into a single 17MB binary with zero dependencies (which leads to reduced infrastructure sprawl).

NATS is also much more flexible than Kafka in terms of deployments with clusters, super-clusters and leaf nodes as well as stream mirroring and sourcing being built-in. Is it also easier to use, administer and configure and requires no tuning. And finally, to turn JetStream into a comprehensive and powerful stream processing platform that rivals Kafka plus the Confluent Platform components, you can add an open-source project called **Benthos**, which has NATS JetStream integration, to your solution.

And finally: NATS JetStream and Kafka are not mutually exclusive! They can happily live together and it is easy to connect the two together by using the **NATS-Kafka adapter** (or through Benthos). They can even be combined together to get a better overall architecture.

Components

First obvious difference is that Kafka is written in Java and Scala, while NATS is written in Go, which means that with NATS you will never need to install or tune a JVM (e.g., adjust the heap size), and arguably Go is more efficient when it comes to memory usage compared to Java.

Artifacts

The Apache Kafka installation comes as a single compressed archive file that expands to a directory tree containing around 200 files (half of them being JAR files) and weighing in at a bit over 100 MB. In comparison NATS with JetStream comes as a compressed archive file containing the nats-server binary (less than 17 MB in size) plus the README and a license file, and that's it! Admins and developers may also want to install the companion **'nats' CLI tool**, a single powerful tool in a single binary that lets you administer, manage, test, monitor and exercise all of the features of NATS and JetStream.

Processes

Another difference is that with Kafka you need to run more than one kind of JVM process in your server infrastructure: at a minimum, you need to run Zookeeper

nodes (unless you use KRaft), and broker nodes. However, in many cases to get some of the functionalities provided out of the box by NATS JetStream you would also need to run Kafka Streams JVMs. If you are doing a multi-site deployment, you may also need to run MirrorMaker JVMs as well (see Deployment and operations below).

In comparison, with NATS everything is included inside the `nats-server` Go binary. You enable the JetStream functionality in the server configuration and clustering, super-clustering, leaf-node functionality as well as stream mirroring (and sourcing) between clusters are all built-in.

While NATS JetStream client libraries are available in most currently **relevant languages**, if your application is written in Go, it is also extremely simple to run a NATS server embedded within your application code allowing you to, for example, create self-contained distributed applications by embedding the NATS server in each instance and have all of the instances form a self-contained NATS cluster.

Functionality

“Messaging” vs “Streaming”

Kafka is designed for distributed log processing, and at its core (without needing to use Kafka Streams) that is its sole function. And sure, because it is ‘real-time’, **albeit not ‘real real-time’**, you can indeed do basic publish/subscribe to a topic using Kafka, but when it comes to proper messaging functionality it is not comparable to NATS. At its core, NATS is a dedicated high speed distributed messaging system designed with an aim for high speed and low latency and is used extensively in real-time environments where latencies have to be measured in micro-seconds rather than milliseconds, with the other functionalities built on top of that messaging base.

Messages

NATS and Kafka both have a default max message size of 1 MB. With NATS there is a single server setting to change the max message size up to 64 MB, while with Kafka you need to change multiple settings in the server, the producer side and the consumer side. With NATS JetStream you can also use the built-in 'Object Store' functionality to store data blobs of any size. Both NATS and Kafka also support message headers. Kafka has built-in support for schemas and a schema registry.

"Subjects" vs "Topics"

In terms of messaging functionality NATS has proper subject-based addressing with a hierarchy of tokens and powerful partial or full wildcard support. You do not need to define subjects before using them (they have no attributes) and applications can publish to a virtually unlimited number of subjects. Streams can contain messages on any number of subject name hierarchies, and you can add or remove subject name hierarchies from streams on the fly. There is no hard limit to the number of subjects that you can use.

In Kafka there is a one-to-one relationship between a topic and a stream (in the JetStream definition of the term, not the same as a 'KStream' which is implemented by the client in the Kafka Streams library, meaning that Kafka brokers have no concept of KStreams or KTables). Topics have to be defined ahead of time (unless you are ok with default values) and topic creation causes re-partitioning. While you can filter topics with a regular expression, that is much harder to use and much less powerful than hierarchical subjects (with available mapping) and it means that the clients have to constantly refresh the list of topics from the server to match against the expression.

NATS also has the ability for the servers to automatically transform message subjects, by defining subject mappings at various points of the server processing path, allowing you to replace, remove, split or slice subject name tokens or insert a partition number (deterministically calculated from the value of any part of the subject) into the subject (should the application need partitioning). Subject mappings can also be used with request-reply for things like canary or A/B testing of services.

“Queues” vs “Topics”

Both Core NATS and JetStream have distributed message queuing functionality where the messages are actually deleted from the stream as they are consumed by the client applications and where the consumption of the messages is distributed in a demand-driven fashion without the need for partitions, messages can be acknowledged individually.

Message redelivery is automated and powerful, besides acknowledging a message you can also:

- Punt on it and ask for redelivery at a later time (you can even specify you own backoff timings in the nack), e.g., an external resource is unavailable at the time.
- Ask for more time to process the message, e.g., an external resource is available but slower than normal.
- Permanently reject (terminate) a bad message, e.g., the payload is bad.
- Adjust timeouts, the number of acks pending

In comparison, while you can use a Kafka topic as a pseudo queue by committing consumer offsets, there is no concept of individual acknowledgement. Committing an offset is equivalent to acknowledging not just a specific message but also all the messages prior to it.

This is a small but important distinction as it means that if you want to be protected against potential duplicate processing or message loss in some failure scenarios the client needs to process and then synchronously commits its offset exactly one message at a time and there cannot be more than one client consuming from the same topic at the same time. Obviously, Kafka deals with this limitation of only one consumer at a time per topic through partitioning, but even leaving the administrative burden associated with partitioning aside, partitioning only solves the problem down to the process level, if you want to safely parallelize between threads then you need to deal with not being able to acknowledge individual messages. Kafka has no concept of automated scheduled redeliveries.

Microservices vs CQRS

Core NATS has synchronous request-reply functionality that distributes (as always, without the need for partitions) the requests between the servers with multi-cluster geographical affinity (i.e., the requests are transparently directed in preference to the servers in the same cluster if there are any there). It also has a services SDK that implements many of the features of a **service-mesh**.

Kafka on the other hand can only do asynchronous 'CQRS' like request-reply where the request is put on one topic and the reply expected to appear in another one (where the requestor must go through all the replies in the topic to find its own which is inefficient). Asynchronous request-reply can also be accomplished with JetStream; however, the difference is that the requesting application only receives its own replies from a shared 'reply stream' using subject filtering while with Kafka it needs to either scan the whole shared reply topic for its answer(s) or use a separate topic per requesting application instance.

Durability

At the highest level, the primary functionality that Kafka and other Streaming systems provide over traditional publish/subscribe messaging is the temporal decoupling between producers and consumers of messages. Removing the requirement that the subscribers must be connected and actively subscribing at the time of the publication of the message to receive it.

Requiring subscribers to always be up and running is a serious limitation for a middleware and soon messaging systems introduced durable subscribers and queuing as an initial form of temporal de-coupling where the messaging servers store and persist ‘in-flight’ messages.

However, those queuing messaging systems where not distributed, used one persistent storage system shared by all (active and passive) instances and therefore could only be scaled vertically. Kafka’s success stems from the fact that it was the first popular system providing temporal decoupling “at least once” quality of service with replay capabilities (i.e., “Streaming”) in a distributed and therefore horizontally scalable way.

Log vs Store

Both Kafka and JetStream are a distributed persistence systems that are used to provide that durable “Streaming” functionality. They both are ‘shared-nothing’ designs and their servers can store the data on local files with replication between the servers to ensure fault-tolerance. But beyond that, they expose fundamentally different kinds of storage functionalities.

Kafka is an append only LOG

- There are no deletes inside the log, just compaction of the tail end. The only compaction options are topic size and time.
- While Kafka's log compaction can be set to keep at least one message per key in the stream, it cannot provide a guarantee that there is exactly just zero or one message per key in the topic.
- You can always publish a message into topic as long as there is room, compaction happens periodically, not each time a message is published. There is no concept of constraints or concurrency write control.
- You can only iterate over the data in a topic, there is no addressing of messages in the topic other than by offset (you can get the offsets for a particular point in time through a separate call). The key of messages is used for distribution (partitioning) purposes only, not addressing. Seeking to different offsets all the time noticeably affects performance.

JetStream is a consistent message STORE

- JetStream Streams are 'read/write', you (or JetStream itself) can delete individual messages anywhere in a stream.
- You can get messages and query stream using subject based addressing.
- You can specify and combine exact limits (constraints) on streams such as:
  - a maximum number of messages
  - a maximum number of bytes
  - a maximum number of messages per subject

- You can decide on whether you want a new publication that would breach a limit to either discard an old message in the stream to make room for the new message or be rejected with an error:
    - For example, use discard old message combined with a limit of exactly one message per subject to use a stream as a 'last value published cache'.
    - Or by combining rejection with a **maximum number of messages per subject limit** you can use a stream for many kinds of logic gating use cases such as distributed locking or semaphores or infinite message deduplication.

JetStream has 'CRUD' operations: you can control concurrent writes using the atomic 'compare and set' functionality to do "insert", "update", or "insert or update" of a message according to its subject.
You can do per subject "Rollups" replacing all of the messages in the stream for a specific subject with the new message.

'Physical' Storage

Both Kafka and JetStream use file-based storage, but JetStream also has the option of using memory, should you need the lowest possible latency.

If you are paying for the number of disk I/O per second that you make (i.e., if you are using Block Storage), you should know that **JetStream has been shown to consume much less IOPS than Kafka** does for an equivalent workload.

Streaming functionality

Terminology

As a quick preamble, there is a difference in terminology between NATS JetStream and Kafka that must be mentioned first as it is often a source of confusion:

A JetStream **Stream** is equivalent to a Kafka **Topic** in the sense that it persists a series of messages (but with a lot more functionalities), while the term **Stream** in a Kafka context is often used to refer specifically to a Kafka Stream **KStream**.

A JetStream **Consumer** is equivalent to a Kafka **Consumer Group**

And subsequently...

A JetStream **Subscriber** (to a Consumer) is equivalent to a Kafka **Consumer**. Equivalent here means that at a high level they provide similar functionality but there are still differences (see below).

Addressing messages in a stream

JetStream can address the data (messages) that it stores (in streams) using subjects, as well as by sequence number.

This means that you can directly (and quickly, as the servers maintain indexes) address messages in a stream by their subject:

- Get just the current first, or last message for a specific subject name.
- Get (query) all the messages for a subject name or hierarchy.
- Get the number of messages stored in the stream for a specific subject name.

In comparison, the only way to address messages in Kafka is by sequence number (offset) or indirectly using a timestamp, there is no way for example to get a message in a topic by key (you must scan the whole topic or use Kafka Streams to create a materialized view in the application to cache all keys and values), or to query the topic for all keys matching a hierarchy.

Producing messages

With subject based addressing in JetStream, the key (and/or unique id) of the message is naturally included inside the subject (vs explicitly passed as a separate argument).

- You can use as many tokens as you need (like a composite key).
- You can include attributes you know you are going to want to filter on when consuming as subject tokens (vs having to use Kafka Streams).

With Kafka the key is a non-hierarchical optional value that must be presented by the publishing application at publication time, if the application does not provide a value for the key (or a bad one in terms of distribution i.e., one with low cardinality) you will not be able to distribute the messages between the partitions.

NATS and JetStream do not have transactional support when producing messages: instead you can publish the individual messages to stream(s) and either remove them (using their sequence number) to do a 'rollback' or publish a message on a specific subject to indicate a 'commit'.

Consuming messages

Kafka message consumption:

- Replay from start, end or specific offset only. Replaying from a point in time requires a separate operation to get the offsets first and then seek to the those offsets.

- Anything beyond that requires you to do filtering in the client application by discarding unwanted messages after they have been received from the brokers (e.g., using Kafka Streams).
- The Kafka client is single-threaded, meaning that the throughput can be very limited unless many partitions (and large consumer groups) are used and many instances of the client are deployed. There is a 'parallel consumer', but it's only available in Java.

In comparison JetStream Consumers:

- Support: all (start), new (end), message sequence (offset), plus last message, last message for each subject, and from a period of time in the past.
- Can replay the messages at the same rate they were initially published.
- Can leverage subject based addressing to get specific messages or query for messages matching subject filters, like a 'view' on a Stream (vs having to use Kafka Streams).
- Allow you to use a Stream as a working queue that deletes the messages as they are acknowledged by the instances of the consuming application(s).
- Are like 'partition-less Consumer Groups' (durable consumers) or can just be created by the client application for its own use only (ephemeral consumers)
- Are stateful, and their state is persisted and replicated by the servers (just like a Stream).
- Provide pull and push delivery mechanisms (Kafka is pull only).
- Have one-to-many flow control (push and pull) as well as one-to-one flow control (pull).
- Messages are delivered in order by the JetStream consumers.
- JetStream consumer functionality (like all NATS functionalities) in the client libraries is thread-safe (i.e. the message consumption can be multi-threaded in the client code if so desired).

Partitioning

As mentioned above Kafka has a ‘hard’ relationship between partitions and consumers to a consumer group:

- You need partitions to distribute the messages between consumers in a group.
- You can only ever have one consumer per partition at a time in a consumer group.
- You always have to be able to answer the questions “what partition strategy?” and “how many partitions?”.
- You can only ever add partitions (and it doesn’t re-distribute the data) to an existing topic.
- You always have to be mindful of your partition counts, especially as you create new topics. A Kafka cluster can only have about 200,000 topic-partitions total (although KRaft increases that number to about 2 millions, there is still a hard limit). This means that if you need high parallelization and for example 100 partitions per topic, then you can only have 2000 topics total.

JetStream Stream on the other hand are partition-less by nature:

- You can scale up or down your client application subscribers (consumers) to a consumer (consumer group) automatically and at any time, no re-partitioning or re-balancing of partitions.
- No need to worry about consumer polling, “livelock” situations and commit failures.
- No need to worry about automatic vs manual partition assignment or about a partition strategy.

While arguably partition-less and demand-driven distribution of the messages in a Stream works for most use cases, there are still use cases where you may need to

partition your messages according to the value of a ‘key’ over a number of partitions. You can still do partitioning of message streams with NATS JetStream by using the subject mapping features:

- Administrative operation only: the publishers are oblivious and don’t need to know anything about it being used or not.
- Inserts a partition number token into the subject
- You can select any number of subject tokens (i.e., the ‘key’ value is the union of those subject tokens) to partition on.
- Can be adjusted (up or down, no redistribution) at any time.
- Partitioning is deterministic: all the messages with a particular subject name are assigned to the same partition.
- You can use the partition number token to create one stream per partition.
- Or you can use the partition number token to create one JetStream consumer per partition.

Qualities of Service

Qualities of services, in the context of messaging and streaming are defined as (in increasing order of quality): at most once, at least once and exactly once.

At most once is the quality of service offered by messaging systems without persistence: it is ‘best effort’ and if the listening process is not running and connected at the time the message is published then it never gets the message.

Both Kafka and JetStream claim to offer at least once (through persistence) and exactly once quality of service (which is difficult to do properly), however when it comes to the details there are some notable differences:

- **‘at least once’**
  - JetStream consumers can automatically re-deliver messages and manage acknowledgements at the individual message sequence number level while Kafka consumer groups only support ‘Ack All’ (see ‘Queues vs Topics’ above).
- **‘exactly once’: message de-duplication**
  - JetStream has built-in message de-duplication over a configurable time window which works across client application sessions.
    - It is now also possible to have **message de-duplication that is not time limited and covers the entire contents of the stream**, leveraging the fact that JetStream has proper data store (rather than just WAL) functionalities.
  - Kafka’s built-in message de-duplication feature is limited: “the producer can only guarantee idempotence for messages sent within a single session”.
    - Application crashing before receiving the publication’s ack and then restart and re-publishing is IMHO actually quite a likely scenario!
    - If you want de-duplication at the same level of quality as with JetStream you need to use Kafka Streams and a data store.
- ‘exactly once’: reliable message consumption
  - JetStream has a synchronous double acknowledgement of the consumption of a message feature (AckAck).
- Kafka does have synchronous commit of the offset, but that is equivalent to an ‘Ack All’ rather than explicitly Acking each individual message as you do with JetStream, which practically means that you can not safely parallelize the consumption of messages beyond the number of partitions if you need to process each message individually exactly once.

Data Store: JetStream persistence enables more than just Streams

Because of the data store functionalities described above, NATS JetStream also offers out of the box data store functionalities that go beyond ‘just streaming’.

NATS JetStream also exposes an immediately consistent Key/Value store functionality with concurrency access control, history per key and real-time updates. With Kafka, you would have to use Kafka Streams (to create materialized views in KTables) or something like ksql or even Redis.

NATS JetStream also exposes an Object Store feature which is a lot like the Key/Value store but allows you to store BLOBs of any size and does all the chunking underneath the covers for you.

Security

Authentication

Apache Kafka is only plain (user name and password) or certificate-based authentication, with only one level: users, while JetStream supports plain, certificate or JWT-based delegated authentication/authorization with accounts and user levels of authorization.

Apache Kafka only has ACLs (RBAC requiring the Enterprise version of Confluent Platform), while NATS has both ACLs and RBAC (scoped signing keys).

Multi-tenancy

NATS has a hierarchical security model that provides proper multi-tenancy at the account level, with users belonging to those accounts

- It is true multi-tenancy: each account (i.e., tenant) has its own completely independent subject namespace.
- Within an account, each user (a ‘user’ being typically a client application, not an actual human) in an account can be allowed/denied publishing or subscribing on a per-subject basis.
    - This extends to being able to limit what JetStream functionality is available to the user: for example, you could allow or disallow users from creating or deleting streams or from creating or deleting consumers, on a per stream basis.
- You have the ability to import/export message streams (with subject mapping if needed) between accounts.
    - You can set account-level limits on the resources that an account can have access to (e.g., stream size, number of streams, etc.)
    - You can also import/export services (i.e., request-reply messages for a service) between accounts.
- Administration is delegated: account administrators can use account signing keys (with scoped authorization) to create user credentials independently without those credentials needing to be uploaded to the servers or added to server configuration files.
- Changes to account JWT such as authorization attributes, signing keys, revocation lists, imports and exports take effect immediately (including closing any current client connections for revoked users or signing keys) as soon as the new JWT is uploaded to the cluster.

- Client applications do not need to know anything about multi-tenancy, they simply connect with user credentials and don't need to be aware of what tenant they belong to.
- The security callout functionality introduced with version 2.10 allows you to integrate NATS authentication and authorization with any existing security infrastructure you may already have.

Kafka in comparison only has the equivalent to NATS 'users' and no concept of 'accounts' with completely isolated topic namespaces.

- To emulate multi-tenancy functionality, you must create a hierarchical topic naming structure and manually carve parts of the topic namespace in "user spaces" for each tenant.
- Tenants may have to use more than one "user space" depending on how you had to organize your topic hierarchy.
- Careful consideration is required regarding grouping topic names by team, or by organizational unit.
- No concept of import/export of topic (or of request-reply services) between tenants.
- Client applications need to be aware of the multi-tenancy topic naming structure and know which 'tenant' id they should use when publishing or receiving messages.

Encryption

Both Kafka and NATS support encryption of the data being transmitted over the network between the clients and the servers and between the servers in a cluster. JetStream can also do encryption of the data at rest and it encrypts what is being persisted, and safe deletion by overwriting the deleted data. Kafka does not support encryption at rest or deleting messages (other than by log compaction).

Deployment and operations

Resilience

JetStream utilizes the RAFT distributed consensus algorithm to protect against data inconsistency in light of node or network failures. If the replica count is above one, then a vote takes place for every message being stored or deleted.

- Nodes vote only after they have done the write to the file system, there is no internal page cache.
- This means that streams are immediately consistent, JetStream favors Consistency over Availability (see the CAP theorem) meaning that you need at least 2 out of 3 nodes to be up to accept new messages but it can detect any number of network Partitioning ‘split-brain’ failures.
- It detects Byzantine failures.
- You have a guarantee that there will be no loss of data once you receive the acknowledgement for your publication back from the server, always.
- You can change the replication factor up and down on the fly and it is a simple single CLI operation without interruption of service.

Kafka doesn’t do a RAFT vote for every message and replicates messages published to a topic asynchronously in batches with the option to wait for all of the ‘in sync replicas’ to have received the message.

- The flush interval on the Kafka brokers determines how frequently the data in the page cache is flushed to disk, not the producer flush method, which simply controls sending the message and synchronously getting an ack for that message on the client side.

- It relies on heart-beats timeouts with Zookeeper nodes to assert whether nodes are operational or not, and (this is an important distinction) by default the number of ‘in sync’ servers can go down to 1.
    - This means that Kafka favors Availability over Consistency in that it can commit writes to a replicated topic with just one ‘in sync’ node being up: if you then loose that node (i.e. its files), you can lose committed data.
        - Unless you explicitly disable ‘unclean leader election’ you may even lose data without realizing it.
    - To get more Consistency you need to set a minimum number of in sync replicas to something higher than 1.
        - To handle one single network partition you need to set `min.insync.replicas = 2` and `replication.factor = 3`
        - To handle any number of network partitions you need to set `min.insync.replicas = replication.factor`.
- Kafka replication does not detect Byzantine failures.
- Changing the replication factor of a Kafka topic means crafting a reassignment file with an entry for every partition (!) the topic has to first execute and then check using a dedicated tool.

Clustering

JetStream automatically distributes the streams (i.e. the data and its replicated copies) over the servers in the cluster:

- Adding or removing servers from a cluster doesn’t cause stream re-distribution (unless administratively triggered).
- Adding or removing client applications from a consumer doesn’t cause any redistribution.

Kafka distributes the partitions over the servers and the client applications over the partitions:

- Adding or removing servers from a cluster means redistributing the partitions.
- Adding topics means distributing the partitions over the servers in the cluster.
- Removing topics may mean re-distributing remaining streams' partitions over the servers in the cluster.
- Adding or removing client applications from a consumer group means re-distributing the partitions over the client applications.
- Data can get unbalanced in your cluster and you need to use either `kafka-reassign-partitions` or CruiseControl (which adds to the infrastructure sprawl).

Super-clusters and Leaf Nodes

NATS JetStream has a lot of deployment flexibility: besides clusters, you can create Super-clusters spanning the globe and extend NATS JetStream service to the edge with Leaf Nodes.

- You can deploy clusters in each region/site/cloud provider and connect them together to form a Super-cluster that can span multiple public clouds and the entire globe like Synadia's NGS.
- Connecting your clusters together in a super-cluster is completely transparent for the client applications as they can access data (streams) regardless of which cluster the servers storing and replicating the data are located and regardless of which cluster the client application is connected to.
- Super-clusters provide automatic geo-affinity for the Core NATS request-reply functionality: if there are service instance(s) deployed to the cluster the client application is connected to the request remains in the local cluster, otherwise it is routed another (nearest) cluster where service instances are running.

- Leaf Nodes are ‘local’ NATS servers configured to act as extension points to the clusters or super cluster. They can be their own JetStream domain to provide local always-on service even when disconnected from the cluster. A common use case is to can have many Leaf Node servers running on small embedded devices located in things with transient connectivity (e.g. a vehicle or any mobile device using a cell phone network) connecting to a hub cluster in the cloud where the applications in the vehicle can always safely publish to a stream (the message is persisted by the leaf node and then reliably mirrored or sourced to streams in the cloud) and receive all messages published for them in the hub despite the connectivity gaps.

Apache Kafka in comparison only has the concept of a single cluster, there is no built-in functionality to connect clusters together (doing a “multi-region cluster” actually means having a single cluster and using the ‘rack id’ feature where each ‘rack’ is a region).

Data placement

NATS JetStream has a two levels of data placement functionality (meaning which server nodes are selected to service and store data for a specific stream) using placement tags as opposed to Kafka’s single ‘rack id’ placement functionality:

- Within a cluster you can ensure that the servers automatically selected to replicate the stream are located in different availability zones (or on different racks) using the placement tags.
- You may think that placement tags are equivalent to the ‘rack id’ feature of Kafka but they are actually more powerful as they also give you the ability to control data placement between clusters within a Super-Cluster, allowing you to:
    - Ensure that the streams containing data for EU customers are stored only in the EU cluster

    - Ensure that the data is replicated over different AZs or racks within the cluster hosting the stream.
    - Very easily move streams and their data from one cluster to another using a single simple administrative operation and without any interruption of service.
- NATS JetStream also has built-in support for stream mirroring and sourcing.
    - Mirroring and sourcing use reliable store and forward mechanisms and are resilient to server or network failures.

In contrast, with Kafka:

- There is no concept of a Super-cluster connecting independent clusters together to make a single Kafka cluster “multi-region” you need to rely on replication counts and constraints and using one ‘rack id’ per region.
    - You can only control placement of replicas at a single level: the “rack id”, meaning within a cluster only, not between clusters.
    - There is no built-in ability to move topic data from one cluster to another.
- No super-cluster concept means clusters are independent from each other and you need to use MirrorMaker 2 or the Replicator, which require running Connect if you need to mirror topics from one cluster to another.
    - Or with the Confluent Platform you need to use the Cluster Linking for Confluent Platform (which is part of the Confluent Platform Enterprise subscription).
    - It is more meant for Active-Passive replication, as Active-Active adds extra complexity, especially with Cluster Linking.
- In order to do stream sourcing (the ability to reliably copy messages with filtering and subject transformation between streams) you need to code it with Kafka Stream (and the data will be shuffled back and forth between the broker and Stream JVMs rather than happening directly at the NATS server level).

Operations

Container orchestration

NATS has a free full featured Helm chart for deploying **NATS servers as stateful sets in Kubernetes**. There is also a NATS JetStream controller (**NACK**).

The Kubernetes operator is part of the Confluent Enterprise Platform subscription.

Administration

Kafka administration tools include many different admin scripts and CLI tools while the single `nats` CLI tool is extremely powerful and does absolutely everything from testing, JetStream admin, monitoring, to traffic generation and benchmarking and a lot more in between.

Note that you can now also use the "**Synadia Control Plane**", which is a centralized administrative portal for managing and monitoring one or more NATS systems.

JetStream is also easy to use (e.g. partition-less consumer groups), configure and deploy, requires no tuning and very flexible when it comes to deployment options with support for server clustering, super-clustering for global or multi-cloud deployments, and support for store and forward reliable service to the edge (even the partially connected edge).

Kafka requires a _lot_ of configuration and tuning, including many 'timing' settings. It is quite easy to end up with a problematic installation that sometimes fails due to misconfiguration of the tuning settings.

Administrators also have to deal with other things beyond the brokers' configuration such as topic partitioning and Zookeeper. Security administration requires coordination between the administrators and the developers.

NATS and JetStream configuration is comparatively a lot simpler. No tuning or partitioning is required, you don't have to figure out timings in order to get the best performance. Multi-tenant security means the developers do not need to do anything besides connect with credentials, and you can delegate the administration of accounts.

Cost

With **JetStream you can require less nodes, less types of nodes, and less disk IO per second and network bandwidth** (which are not free in the cloud) than you would when using Kafka and the other components of the Kafka Platform, which leads to infrastructure sprawl.

If you want to do mirroring between sites, have multi-region clusters, or even facilitate deployment over Kubernetes then you may have to subscribe to the Enterprise features of the Confluent Platform, while JetStream has stream mirroring built-in and comes with a Helm chart.

"Direct connectivity" vs "Kafka Connect adapters"

While both NATS and Kafka have their own specific communication protocols over TCP, NATS also support direct communications between the clients and the servers (and between the servers) over WebSockets: which can become necessary when you have no control over what kinds of proxies and firewalls may be deployed between the client applications and the servers. If you take any Go, Java, or JavaScript/TypeScript NATS client application, to switch it to use WebSockets is as simple as changing the connection URL.

NATS also has built-in direct support for the MQTT messaging protocol, meaning that MQTT client applications can connect directly to the NATS servers without the need to deploy a connector.

NATS JetStream vs Kafka Streams

You probably noticed in the previous sections that a lot of the Streaming functionalities that JetStream has and that are not part of the core Kafka log processing service can be implemented using Kafka Streams.

So, let's extend the comparison between NATS JetStream and Kafka to include Kafka Streams as well.

Kafka Stream is a Java/Scala "client library where the input and output data are stored in Kafka clusters", that you can use to create JARs that are then deployed on and run on Kafka Streams JVMs who receive data from the broker JVMs, do the stream processing and then send data back to the broker JVMs where it is stored again. Kafka Streams augments the Kafka client library with higher level functionalities such as KStreams and KTables and a Streams DSL to facilitate the writing of those stream processing applications.

In comparison, there is only one version of NATS the client libraries that already expose both a Stream and Key/Value functionalities that even have some functionalities that KStreams/KTable do not have:

- You can store a history of a specific maximum number of values per key: KTable only stores one record per key, KStreams cannot enforce an exact maximum number of records per key.
- You can have constraints and concurrent writes access control in both Streams and Key/Value allowing you to select between doing an "insert", an "update" or an "upsert" (that last option being the only behavior of KStream/Ktable).

Some of the functionalities of the Streams DSL are also offered and executed directly (and therefore much more efficiently) by the NATS JetStream servers by leveraging subject-based addressing: namely filtering of messages by subject (like being able

to filter messages from a KStream by key), and stream sourcing (being able to copy messages from stream to stream with subject filtering and simple transformations).

However, obviously, there are a lot more functionalities of Kafka Streams DSL, that you can't do just at the server level with NATS JetStream, such as any kind of filtering, transformation, joins or aggregation of the message payload data, meaning that you will also need to deploy "Streaming processing" client worker processes, just like you do with Kafka Streams.

You can use the NATS client library for the language of your choice (you are not limited to Java and Scala) to build your stream processing applications like you would use Kafka Streams and have similar functionality (e.g., Key/Value buckets instead of KTables), but that is not the only option.

Another option is to use an open source project called Benthos [https://benthos.dev] which is a declarative data streaming service that solves a wide range of data engineering problems with simple, chained, stateless **processing steps.**

Benthos is integrated with NATS JetStream messaging, streaming and Key/Value (including a Key/Value processor, so you can do joins against Key/Value buckets for example).

Benthos can do a lot more than Kafka Stream, it has functionality that overlaps with integration frameworks, log aggregators and ETL workflow engines, it has its own powerful and easy to use data mapping language.

Benthos integrates out of the box with a very extensive list of sources and sinks (including Kafka, naturally) so it is also a replacement for Kafka Connect.

Finally

For a conclusion, see the summary section at the start of this white paper.

Synadia