Versioning for JetStream Assets

Metadata	Value
Date	2024-07-22
Author	@ripienaar
Status	Partially Implemented
Tags	jetstream, server, 2.11

Context and Problem Statement

As development of the JetStream feature progress there is a complex relationship between connected-server, JetStream meta-leader and JetStream asset host versions that requires careful consideration to maintain compatibility.

- The server a client connects to might be a Leafnode on a significantly older release and might not even have JetStream enabled
- The meta-leader could be on a version that does not support a feature and so would not forward the API fields it is unaware of to the assigned servers
- The server hosting an asset might not match the meta-leader and so can't honor the request for a certain configuration and would silently drop fields

In general our stance is to insist on homogenous cluster versions in the general case but it's not reasonable to expect this for leafnode servers nor is it possible during upgrading and other maintenance windows.

Our current approach is to check the connected-server version and project that it's representitive of the cluster as a whole but this is a known incorrect approach especially for Leafnodes as mentioned above.

We have considered approaches like fully versioning the API but this is unlikely to work for our case or be accepted by the team. Versioning the API would anyway not alleviate many of the problems encountered when upgrading and downgrading servers hosting long running assets. As a result we are evaluating some more unconventional approaches that should still improve our overall stance.

This ADR specifies a way for the servers to expose some versioning information to help clients improve the compatability story.

Solution Overview

In lieu of enabling fine grained API versioning we want to start thinking about asset versioning instead, the server should report it's properties when creating an asset and it should report similar properties when hosting an asset.

Fine grained API versioning would not really fix the entire problem as our long-lived assets would have to be upgraded over time between API versions as they get updated with new features.

So we attempt to address both classes of problem here by utilizing Metadata and a few new server features and concepts.

API Support Level

The first concept we wish to introduce is the concept of a number that indicates the API level of the servers JetStream support.

Level	Versions
0	< 2.11.0
1	2.11.x
2	2.12.x

While here it's shown incrementing at the major boundaries it's not strictly required, if we were to introduce a critical new feature mid 2.11 that could cause a bump in support level mid release without it being an issue - we do not require strict SemVer adherence.

The server will calculate this for a Stream and Consumer configuration. Here is example for the 2.11.x. It's not anticipated we would have to keep supporting versions for every asset for ever, it should be sufficient to support the most recent ones corresponding to the actively supported server versions.

```
func (s *Server) setConsumerAssetVersionMetadata(cfg *ConsumerConfig, create bool)
{
    if cfg.Metadata == nil {
        cfg.Metadata = make(map[string]string)
    }

    if create {
        cfg.Metadata[JSCreatedVersionMetadataKey] = VERSION
        cfg.Metadata[JSCreatedLevelMetadataKey] = JSFeatureLevel
    }

    featureLevel := "0"

    // Added in 2.11, absent | zero is the feature is not used.
    // one could be stricter and say even if its set but the time
    // has already passed it is also not needed to restore the consumer
    if cfg.PauseUntil != nil && !cfg.PauseUntil.IsZero() {
        featureLevel = "1"
    }

    cfg.Metadata[JSRequiredFeatureMetadataKey] = featureLevel
}
```

In this way we know per asset what server feature set it requires and only the server need this logic vs all the clients if the client had to assert `needs a server of at least level 1`.

We have to handle updates to asset configuration since an update might use a feature only found in newer servers.

Servers would advertise their supported API level in `jsz`, `varz` and `$JS.API.INFO`. It should also be logged at JetStream startup.

When to increment API Level

Generally when adding any new feature/field to the `StreamConfig` or `ConsumerConfig`. Especially when the field was set by a user and the asset should be loaded in offline mode when the feature is not supported by the server.

If a new feature is added, the API level only needs to be incremented if another feature planned for the same release didn't already increment the API level. Meaning if multiple new features are added within the same release cycle, the API level only needs to be incremented once and not for every new feature.

Server-set metadata

We'll store current server and asset related information in the existing `metadata` field allowing us to expand this in time, today we propose the following:

Name	Description
`_nats.ver`	The current server version hosting an asset
`_nats.level`	The current server API level hosting an asset
`_nats.req.level`	The required API level to start an asset

We intend to store some client hints in here to help us track what client language and version created assets.

As some of these these are dynamic fields tools like Terraform and NACK will need to understand to ignore these fields when doing their remediation loops.

Offline Assets

Today when an asset cannot be loaded it's simply not loaded. But to improve compatability, user reporting and discovery we want to support a mode where a stream is visible in Stream reports but marked as offline with a reason.

An offline stream should still be reporting, responding to info and more but no messages should be accepted into it and no messages should be delivered to any consumers, messages can't be deleted, configuration cannot be updated - it is offline in every way that would result in a change. Likewise a compatible offline mode should exist for Consumers.

The Stream and Consumer state should get new fields `Offline bool` and `OfflineReason string` that should be set for such assets.

When the server starts and determines it cannot start an asset for any reason, due to error or required API Level, it should set this mode and fields.

For starting incompatible streams in offline mode we would need to load the config in the current manner to figure out which subjects Streams would listen on since even while Streams are offline we do need the protections of overlapping subjects to be active to avoid issues later when the Stream can come online again.

Safe unmarshalling of JSON data

The JetStream API and Meta-layer should start using the `DisallowUnknownFields` feature in the go json package and detect when asked to load incompatible assets or serve incompatible API calls and should error in the case of the API and start assets in Offline mode in the case of assets.

This will prevent assets inadvertently reverting some settings and changing behaviour during downgrades.

A POC branch against 2.11 main identified only 1 test failure after changing all JSON Unmarshall calls and this was a legit bug in a test.

One possible approach that can be introduced in 2.11 is to already perform strict unmarshalling in all cases but when a issue is detected we would log the error and then do a normal unmarshal to remain compatible. A configuration option should exist to turn this into a fatal error rather than a logged warning only.

It could also be desirable to allow a header in the API requests to signal strict unmarshalling should be fatal for a specific API call in order to facilitate testing.

Minimal supported API level for assets

When the server loads assets it should detect incompatible features using a combination of `DisallowUnknownFields` and comparing the server API levels to those required by the asset.

Incompatible assets should be loaded in Offline mode and an advisory should be published.

Concerns

The problem with this approach is that we only know if something worked, or was compatible with the server, after the asset was created. In cases where a feature adds new configuration fields this would be easily detected by the Marshaling but on more subtle features like previous changes to `DiscardNew` the client would need to verify the response to ensure the version requirements were met and then remove the bad asset, this is a pretty bad error handling scenario.

This can be slightly mitigated by including the current server version in the `$JS.API.INFO` response so at least the meta leader version is known - but in reality one cannot really tell a lot from this version since it is not guaranteed to be representative of the cluster and is particularly problematic in long running clients as the server may have been upgraded since last `INFO` call.

In the future we could have clients assert that a certain API call requires a certain API Level but it was felt that today that would not be feasible to do given the amount of clients and their current state.

Implementation Targets

For 2.11 we should:

- calculate and report the api support level
- start reporting the metadata
- soft unmarshalling failures with an option to enable fatal failures

For future versions we can round the feature set out with the offline features and more.