JetStream Subscribe Workflow

Metadata	Value
Date	2021-08-11
Author	@kozlovic
Status	Deprecated
Tags	deprecated

Revision History

Revision	Date	Description
final	2024/07/17	Deprecate in favor of ADR-37

Context

This document attempts to describe the workflow of a JetStream subscription in the library, from the creation, the behavior at runtime, to the deletion of a subscription.

Design

Creation

The library should have API(s) that allows creation of a JetStream subscription. Depending on the language, there may be various type of subscriptions (like Channel based in Go), or with various options such as Queue subscriptions, etc... There can be different APIs or one with a set of options.

Configuration

When creating a subscription, the user can provide several things:

- A subject that is in some cases required (see "Stream name" section below)
- An optional stream name
- An optional consumer name
- An optional JS consumer configuration (see below)
- A Queue name if this subscription is meant to be a queue subscription
- Indication if this is a "Push" or "Pull" subscription
- Handlers for receiving messages asynchronously, whatever that means in the library language

Some misconfiguration should be checked by the subscribe API and return an error outright.

For instance:

- If a queue name is provided, configuring heartbeat or flow control is a mistake, since the server would send those message to random members.

- If no subject is provided, along with no stream name, the library won't be able to locate/create the consumer.
- For pull subscriptions ack policy of "none" or "all" is an error.

At the time of this writing, the consumer configuration object looks like this (ordered alphabetically, not as seen in js.go):

```go
type ConsumerConfig struct {
    AckPolicy       AckPolicy     `json:"ack_policy"`
    AckWait         time.Duration `json:"ack_wait,omitempty"`
    DeliverGroup    string        `json:"deliver_group,omitempty"`
    DeliverPolicy   DeliverPolicy `json:"deliver_policy"`
    DeliverSubject  string        `json:"deliver_subject,omitempty"`
    Description     string        `json:"description,omitempty"`
    Durable         string        `json:"durable_name,omitempty"`
    FilterSubject   string        `json:"filter_subject,omitempty"`
    FlowControl     bool          `json:"flow_control,omitempty"`
    Heartbeat       time.Duration `json:"idle_heartbeat,omitempty"`
    MaxAckPending   int           `json:"max_ack_pending,omitempty"`
    MaxDeliver      int           `json:"max_deliver,omitempty"`
    MaxWaiting      int           `json:"max_waiting,omitempty"`    // server
returns error if provided if a DeliverSubject is provided
    OptStartSeq     uint64        `json:"opt_start_seq,omitempty"`  // server
returns error if provided when Deliver Policy is not by_start_sequence
    OptStartTime    *time.Time    `json:"opt_start_time,omitempty"` // server
returns error if provided when Deliver Policy is not by_start_time
    RateLimit       uint64        `json:"rate_limit_bps,omitempty"` // Bits per
sec
    ReplayPolicy    ReplayPolicy  `json:"replay_policy"`
    SampleFrequency string        `json:"sample_freq,omitempty"`
}
```

Stream name

When no stream is specified, the `subject` to the subscribe API calls is required. An error will be returned if it is not the case. The library will use the subject provided as a way to find out which stream this subscription is for. A request is sent to the server on the `<JS prefix>.STREAM.NAMES` subject with a JSON content `{"subject":"<subject here>"}`. If the response (a list of stream names) is positive and contains a single entry, then the library will use this stream name, otherwise an error indicating that there is no matching stream name should be returned.

However, even with a provided stream name, the `subject` may be needed, for instance when the JS consumer will be created by the library, the `FilterSubject` is set to that, and when a consumer lookup is performed, if the incoming `FilterSubject` is not empty, we ensure that it matches the `subject`.

Consumer name

When a consumer name is specified, it indicates the the library that the intent is to use an existing JS Consumer. The library will lookup the consumer from the server and get a `ConsumerInfo`. If the lookup fails, it is considered an error, unless the subscription is for a "pull subscriber", in which case the library still proceeds with the JS subscription.

Queue name

If the user attempts to create a queue subscription, a consumer name or durable needs to be specified. The common pattern error that we noticed user doing was this:

```
member1, _ := js.QueueSubscribe("foo", "bar")
member2, _ := js.QueueSubscribe("foo", "bar")
```

And then report that both members were receiving the same message.

This is because each subscription call would create their own ephemeral JetStream consumer. We could break the API and force the user to specify a consumer name. Instead, for the Go client we have taken the approach to describe that if no consumer name is provided (in Go with `nats.Bind(stream, consumer)`) nor a durable (in Go with `nats.Durable(durableName)`) then the library would use the queue name as the durable name.

Push Consumer active information and queue group binding

If the lookup succeeds, with newer server (v2.3.5+), the `ConsumerInfo` has now a field called `PushBound` which is a boolean:

```
    PushBound    bool     `json:"push_bound,omitempty"`
```

This boolean indicates that the server has already registered an interest on the push consumer's deliver subject. The `ConsumerInfo.Config` which is a `ConsumerConfig` object, can be inspected to detect if a `DeliverGroup` is set. With both in hand, the library can now return a proper error to the user if they attempt to create an invalid subscription.

- If `PushBound` is true and there is no `DeliverGroup` and the user tries to create a non queue subscription, this should return an error such as "duplicate subscription".
- Regardless of `PushBound` value:
  - If the user tries to create a non queue subscription and `DeliverGroup` is non empty, this should return an error such as "trying to create non queue subscription for consumer created for a queue group".
  - If the user tries to create a queue subscription and `DeliverGroup` is non empty but does not match the user's queue name, this should return an error such as "trying to create a queue subscription for a consumer created for a different queue group".
  - If the user tries to create a queue subscription and `DeliverGroup` is empty, this should return an error such as "trying to create a queue subscription for a consumer create without a deliver group".

Other checks:

- The `FilterSubject` (if not empty) must match the subject passed to the "Subscribe" API, if not return an error indicating the subject mismatch
- If the user was trying to create a Pull subscription, but `DeliverSubject` is not empty, then return an error indicating that user can't create a pull subscription for a push based JS consumer
- The opposite: if the user was not creating a Pull subscription but the `DeliverSubject` is empty, then return an error indicating that a pull susbcription is required
- More generally, if the user provided configuration does not match the configuration that we get from the `ConsumerInfo.Config`, and error should be returned to indicate that the changes are not applied (only a deliver subject can be changed for an existing JS consumer).
- If Flow Control is set in the consumer when a queue / deliver group is requested, return an error indicating the Flow Control is not supported over queues.
- If Heartbeats are configured in the consumer when a queue / deliver group is requested, return an error indicating the Heartbeats are not supported over queues.

NATS subscription

At this point the NATS subscription will be created on the deliver subject found from the consumer info or on an inbox.

Note that the subscription should be created prior to attempt to create the JS consumer (when applicable) because for durable subscriptions, this is the only way for the server to detect if the durable is already in use or not. If there is no subscription (no interest), then the server will simply update the delivery subject of the JS durable consumer.

Creating the JS Consumer

If the library determined that it should attempt to create a JS consumer (no consumer name provided, or durable name that does not exists), the library will fill a consumer config with what was provided by the user.

For push consumers, if the `DeliverSubject` is not specified, the library will pick an inbox name. For pull consumers, `DeliverSubject` needs to be left blank.

The library will set the `FilterSubject` to the user provided subject, ensure that `AckPolicy` is set and possibly some other fields such as `MaxAckPending` and set the `DeliverGroup` to the queue name if the subscribe call is for a queue subscription.

The request is then sent to `<JS prefix>.CONSUMER.DURABLE.CREATE.<stream name>.<durable name>` for a durable subscription or `<JS prefix>.CONSUMER.CREATE.<stream name>` for an ephemeral.

If the operation is successful, the library will get as an API response with a `ConsumerInfo`. Since the library successfully created the JetStream consumer, it will keep track of this fact and will delete the consumer on Unsubscribe() or Drain(). For ephemerals, the consumer name will be saved from the `ConsumerInfo`'s response, since it was not available beforehand.

If the result indicates that the consumer already exists, then it means that there was a race where a process got a lookup "not found" error, but when attempting to create the JetStream consumer, got an "already exists" error. In this case, the library will perform a consumer lookup and will perform the same checks that are described in the "Push Consumer active information and queue group binding" section. Note that for pull subscriptions, basic checks of consumer type validity should be done, but not the checks specific to "push" consumers. For them, unless this is a queue subscription, the API call will return an error.

When the consumer already exists, for push consumers, the NATS queue subscription that was created prior to the "AddConsumer" call needs to be destroyed and replaced with the new NATS queue subscription on the consumer info's `DeliverSubject`.

JSAck

When the server sends a message to a subscriber, the message will have a reply-to value called the "ACK Reply Subject" aka JSAck.

A message must contain a reply-to in the JSAck format to be considered a JetStream message. JSAck will always start with `$JS.ACK`

The JSAck should be used without modification as the publish subject for the message that ACKs the JetStream message.

This JSAck encodes some delivery information, such as stream and consumer name, stream sequence. It contains information allowing proper routing in context of multiple accounts and preventing an ACK for an account to ACK a message for the same stream and consumer names in another account.

There are multiple formats of the JSAck. Both are a period (`.`) delimited strings.

Version 1 (v1) contains 9 fields. It must be supported to ensure clients are backward compatible with previous versions of the server.

For Version 2 (v2), Domain and Account Hash are inserted in positions directly after `$JS.ACK.`, bringing the number of tokens to 11. There may be additional tokens added in the future.

You must ensure that your client can support the 9 token version (v1) and versions with 11 or more tokens (v2), as new tokens may be added to the end of the 11 token format. If the reply-to does not start with `$JS.ACK`, is not 9 tokens, is not 11 or more tokens, or the tokens are not of the correct data type, the client should raise an error.

Token Version 1

Version 1 (v1) is a 9 token format:

```
$JS.ACK.<stream name>.<consumer name>.<num delivered>.<stream sequence>.<consumer sequence>.<timestamp>.<num pending>
```

Token Version 2

Version 2 (v2) is an 11 or more token format:

```
$JS.ACK.<domain>.<account hash>.<stream name>.<consumer name>.<num delivered>.<stream sequence>.<consumer sequence>.<timestamp>.<num pending>
```

When there is no domain, the server will still set the token to a special value of underscore `_` (the server will make sure that users can't pick this as a domain name.)

To the user, the client should expose domain values of `_` as either an empty/null string or as the `_`, documenting its meaning of "no domain".

V2 Notes

Having the domain always present simplifies the library code which does not have to bother with a variable number of tokens somewhere close to the beginning of the subject, and possibly doing some shifting to find out the location of the fields it cares about.

Why not append those new tokens at the end? This is to simplify the export/import subject, ie `$JS.ACK.<domain>.<account>.>`. Otherwise, you would need to possibly have something like `$JS.ACK.*.*.*.*.*.*.*.<domain>.<account>.>`

The account hash is not used by the client at this time, only used for routing, same for the last (12th) token.

Ack Handling

An ack is accomplished by publishing a message using the JSAck as a subject with the payload consisting of the bytes of the Ack Type

Clients must support 3 terminal Ack Types, `+ACK`, `-NAK` and `+TERM` and the progress Ack Type `+WPI`.

Once a terminal Ack Type has been published for a message, the client should ignore all other user requests to send any type of ack.

> A suggested implementation is to mark the message as having sent a terminal Ack Type in some way *after* successful publish of the ack, and to rely on that state for subsequent acks.

If the consumer is configured with Ack Policy of `none`, the client should ignore all requests to send any type of ack.

The client should not return, throw or escalate an error unless the publishing of an ack has an error from the server or for instance due to connectivity.

Automatic Status Management

The client will provide Automatic Status Management (ASM) as the default client behavior. ASM means handling all status messages such as Heartbeats, Flow Control, Pull 404s and 408s, Gap Awareness, etc. A client can optionally provide a mode to allow the user to handle statuses, but that is not required. It is offered to support backward compatibility for pre-existing users.

Heartbeats

If the JetStream consumer is configured with an idle heartbeat interval, the server will send heartbeats when the server has no pending messages. The library should set up a timer to monitor that either messages or status is received, ensuring that the server is connected.

If no message or status is received within a certain time period, an alarm should be surfaced to the user in a way that makes sense in the client language, for example by notifying the user through the asynchronous error callback.

The time allowed before an alarm is raised should be 3 times the idle heartbeat interval, but clients may optionally provide a way for the user to configure this time.

Messages Gaps

Checking for message gaps only applies to consumers that are not participating in queues.

The library should interrogate JetStream messages and Heartbeat messages as each contains the last consumer sequence and the last stream sequence. For JetStream messages, the information is found in the JS Ack reply-to subject (see JS Ack above). For Heartbeat messages, the information can be found in headers `Nats-Last-Consumer` and `Nats-Last-Stream`

If the library detects a gap in the consumer sequence, it can surface a notification to the user in a way that makes sense in the client language, for example by notifying the user through the asynchronous error callback.

Note: Gap checking should be disabled on queue / deliver group subscriptions

Ordered Consumers

Ordered consumers (see ADR-17 when it's ready) will rely on message gap handling. The implementation of ordered consumers will possibly want to handle detection of gaps itself, so clients may want to provide for optional detection of message gaps.

Flow Control

When a JetStream consumer has flow control enabled (not applicable to pull consumers), the library may receive a flow control status message instead of a regular JetStream message.

At the time when the message would have been normally passed to the user for processing, either synchronously via "Next Message" or via an asynchronous callback, the client should respond to the provided reply-to by publishing the value as the subject for a message with an empty payload.

When a flow control message is reached, the server likely had sent more messages. If the user is requesting messages synchronously via Next Message, the client should try to respond with the next buffered message after it responds to the flow control, if it is within any wait time supplied by the user as part of the next message call.

Note: The format of the Flow Control subject should not to be inspected, since it may change at the server discretion, but it will always be a publishable subject.

It is possible, that either the flow control or its response is missed and that the consumer is considered stalled from a server perspective. So the server requires that when flow control is turned on, an idle heartbeat is set. If a flow control message is missed, heartbeat messages will contain a header called `Nats-Consumer-Stalled`, the value being identical to the flow control subject. When the client detects a heartbeat with the stalled header, it publishes to the server as it would respond to flow control.

It's possible that because of the idle heartbeat duration, that both flow control and multiple heartbeat messages contain the flow control subject. It is only required to respond to the specific subject once, so it is suggested that clients track the last flow control subject responded to avoid replying multiple times for the same flow control. The server will ignore duplicates, it is just unnecessary traffic.

Pull Mode Statuses

In pull mode, 404 and 408 statuses should not be surfaced to the user or considered in fetch or iterate counts.

Miscellaneous Error Statuses

These statuses are know to be error conditions and should not be passed to the user but raised as an error.

- 409 - doing conflicting things like trying to pull from a push consumer, exceeding max ack pending etc
- 503 - no responder, like if you try to reply to fc and the subscription went away

Unknown Error Statuses

Status that are not recognized by the client should not be passed to the user but raised as an error.

Unsubscribe and Drain

When a subscription is unsubscribed or drained, if the library created the JS consumer (when it called "AddConsumer" and got an OK), the JetStream consumer is deleted at the end of those calls.

Note I think it is problematic for queue subscriptions since the first member to start will create the JS consumer (in cases where the JS consumer does not already exists), but will be marked as needing to delete the JS consumer on Unsubscribe()/Drain(), while other members may have attached to the same JS consumer.

For instance:

```

// No JS consumer exists.

// Create a queue group on a durable named "shared". Since the JS consumer
// does not exist, this call will create one.
member1, _ := js.QueueSubscribe("foo", "bar", nats.Durable("shared"))

// From another application (or not), we add a member:
// Since the JS consumer existed prior to this call, the following call
// will not create the consumer and the subscription will NOT be marked
// as having to delete the consumer.
member2, _ := js.QueueSubscribe("foo", "bar", nats.Durable("shared"))

// However, at this point, member1 goes away:
member1.Drain()

// When the above completes, the library will delete the JS consumer "shared"
// because the library created it. This will cause member2 to stop receiving
// messages.
```

Unsubscribe

Once the normal unsubscribe processing is done, and for a JS subscription that has been marked as needing to delete the JS consumer, the library should send a "DeleteConsumer" request and return the error if any.

Drain

The deletion of the JS consumer need to be delayed past the point when the subscription is fully drained and removed from the connection. Since this is an asynchronous process, if the deletion of the JS consumer fails, an error will be pushed to the asynchronous error callback.

Consequences

It is possible that some existing libraries will need changes that would break SimVer. It will be left to the library maintainer to evaluate this.