JetStream Simplification

Metadata	Value
Date	2022-11-23
Author	@aricart, @derekcollison, @tbeets, @scottf, @Jarema, @piotrpio
Status	Approved
Tags	jetstream, client, spec

Release History

Revision	Date	Description
1	2023-05-30	Initial stable release
2	2024-06-07	Change server reconnect behavior during `consume()`

Context and Problem Statement

Consuming messages from a JetStream require a large number of options and design decisions from client API users:

- Current JetStream clients create and update a consumer definition on the fly as `subscribe()` or some other functionality for consuming messages is invoked. This leads to some unexpected behaviors as different clients, possibly written using different versions using different options of the library attempt to consume from the same consumer.
- Clients implementing JetStream code are also confronted with a choice on whether they should be implementing a `Pull` or `Push` subscriber.

The goal of this ADR is to provide a simpler API to JetStream users that reduces the number of options they are confronted with and provides the expected performance.

Design

The JetStream API consists of three main components: JetStreamContext, Streams, and Consumers.

JetStream Context

The JetStream Context is mainly responsible for managing streams. It serves as the entry point for creating, configuring, and controlling the streams. JetStreamContext should also expose methods co manage consumers directly, bypassing the need to get/create a stream.

Example set of methods on JetStreamContext:

- Stream operations:
  - `addStream(streamConfig)`

    - `updateStream(streamConfig)`
    - `getStream(streamName)`
    - `deleteStream(streamName)`
    - `listStreams()`
- Consumer operations:
    - `getConsumer(streamName, consumerName)`
    - `createConsumer(streamName, consumerConfig)`
    - `updateConsumer(streamName, consumerConfig)`
    - `createOrUpdateConsumer(streamName, consumerConfig)`
    - `deleteConsumer(streamName, consumerName)`
- `accountInfo()`

Streams

Streams are created from the JetStreamContext. They provide a set of operations for managing the stream and its contents. With streams, you can perform operations such as purging the entire stream and fetching/deleting individual messages. Streams also allow for and managing consumers.

Example set of methods on Stream:

- operations on consumers:
    - `getConsumer(consumerName)`
    - `createConsumer(consumerConfig)`
    - `updateConsumer(consumerConfig)`
    - `createOrUpdateConsumer(consumerConfig)`
    - `deleteConsumer(consumerName)`
- operations a stream:
    - `purge(purgeOpts)`
    - `info()`
- getting/deleting individual messages
    - `getMsg(getMsgOpts)`
    - `deleteMsg(deleteMsgOpts)`

Consumers

Consumer are JetStream API entities from which messages can be read. Consumers should expose methods for getting consumer info, as well as methods for consuming messages (`consume()`, `fetch()` and `next()`).

Example set of methods on Consumer:

- operations on consumer instance:
    - `info()`
- operations used to consume messages:
    - `consume()`
    - `fetch()`
    - `next()`

Design and naming in individual client libraries

Client libraries implementing the JetStream API should adhere to language-specific and library best practices while following the outlined designs. Method names may vary, such as add versus create, and certain methods may be placed differently based on idiomatic conventions (e.g., `consumer.delete()` instead of `stream.deleteConsumer(consumerName)`).

Some libraries may support chaining functionality if it aligns with their JetStream implementation semantics, such as `getStream(name).getConsumer(name)`.

Operations

Consumers will have the following operations:

- `Fetch`
- `Next`
- `Consume`
- `Info` - An optional operation that returns the consumer info of the consumer
- `Delete` - An optional operation to delete the referenced consumer

Lifecycle of Consume may need to be controlled - for example to stop delivering messages to the callback or drain messages already accumulated before stopping the consumer, these can be additional methods on the consumer implementation if appropriate or an object that is the return value of callback driven consumers.

Note: pull requests issued by clients should have client-side timeouts in addition to server-side timeout (`expiry`). Client-side timeout value should always be larger than expiry.

Fetch

Get one or more messages. This operation will end once the RPC expires or the number of messages/data batch requested is provided. The user is in control of when they retrieve the messages from the server.

Depending on the language, the messages will be delivered via a callback with some signal to indicate that the fetch has finished (could be message is null) or via some iterator functionality where getting the next message will block until a message is yielded or the operation or the operation finishes, which terminates the iterator.

Client should also expose the options to fetch certain amount of data (`max_bytes`) instead of messages. Depending on the language, this can either be an option on `Fetch` or a different consumer method (e.g. `FetchBytes`).

Fetch Configuration options

- `max_messages?: number` - max number of messages to return
- `expires: number` - amount of time to wait for the request to expire (in nanoseconds)
- `max_bytes?: number` - max number of bytes to return
- `idle_heartbeat?: number` - amount idle time the server should wait before sending a heartbeat. For requests with `expires > 30s`, heartbeats should be enabled by default

Note that while `max_messages` and `max_bytes` are described as optional at least one of them is required.

Next

Get a single message from the server. The pull request to fetch the message should only be sent once `Next` is invoked.

Depending on the language, the implementation may do `Fetch(max_messages: 1)` or or return an iterator.

Next Configuration options

- `expires: number` - amount of time to wait for the request to expire (in nanoseconds)
- `idle_heartbeat?: number` - amount idle time the server should wait before sending a heartbeat. For requests with `expires > 30s`, heartbeats should be enabled by default

Consume

Retrieve messages from the server while maintaining a buffer that will refill at some point during the message processing.

Client may want some way to `drain()` the buffer or iterator without pulling messages, so that the client can cleanly stop without leaving many messages un-acked.

Consume Configuration Options

- `max_messages?: number` - max number of messages stored in the buffer
- `expires: number` - amount of time to wait for a single pull request to expire
- `max_bytes?: number` - max number of bytes stored in the buffer
- `idle_heartbeat?: number` - amount idle time the server should wait before sending a heartbeat
- `threshold_messages?: number` - number of messages left in the buffer that should trigger a low watermark on the client, and influence it to request more messages
- `threshold_bytes?: number` - hint for the number of bytes left in buffer that should trigger a low watermark on the client, and influence it to request more data.

Note that `max_messages` and `max_bytes` are exclusive. Clients should not allow depending on both constraints. If no options is provided, clients should use a default value for `max_messages` and not set `max_bytes`. For each constraint, a corresponding threshold can be set.

Note that if `max_bytes` is set, client should set `batch_size` in requests to a large number (e.g. 1 000 000) instead of leaving it empty to bypass server sending only one message if `batch_size == 0`.

Defaults and constraints

Default configuration values for `Consume` may vary between client implementations, depending on what values are most efficient using a specific programming language.

- `max_messages` - depends on a client, probably between 100 and 1000 messages
- `expires` - default 30s, minimum 1s
- `max_bytes` - not set, use `max_messages` if not provided
- `idle_heartbeat` - default 1/2 of expires capping at 30s, minimum 500ms, maximum 30s
- `threshold_messages` - 50% of `max_messages`
- `threshold_bytes` - 50% of `max_bytes`

Clients should make sure that `Consume` works properly when `max_messages` is set to 1 (it's not getting stuck when using default `threshold_messages`).

Consume specification

An algorithm for continuously fetching messages should be implemented in clients, taking into account language constructs.

NATS subscription

Consume should create a single subscription to handle responses for all pull requests. The subject on which the subscription is created is used as reply for each `CONSUMER.MSG.NEXT` request.

Max messages and max bytes options

Users should be able to set either max_messages or max_bytes values, but not both:

- If no option is provided, the default value for `max_messages` should be used, and `max_bytes` should not be set.
- If `max_messages` is set by the user, the value should be set for `max_messages` and `max_bytes` should not be set.
- If `max_bytes` is set by the user, the value should be set for `max_bytes` and `max_messages` should be set to a large value internally (e.g. 1 000 000)
- User cannot set both constraints for a single `Consume()` execution.
- For each constraint, a custom threshold can be set, containing the number of messages/bytes that should be processed to trigger the next pull request. The value of threshold cannot be higher than the corresponding constraint's value.
- For each pull request, `batch` or `max_bytes` value should be calculated so that the pull request will fill the buffer.

Buffering messages

`Consume()` should pre-buffer messages up to a limit set by `max_messages` or `max_bytes` options (whichever is provided). Clients should track the total amount of messages pending in a buffer. Whenever a threshold is reached, a new request to `CONSUMER.MSG.NEXT` should be published.

There is no need to track specific pull request's status - as long as the aggregate message and byte count is maintained, `Consume()` should be able to fill the buffer appropriately.

Pending messages and bytes count should be updated when:

- A new pull request is published - add a value of `request.batch_size` to the pending messages count and the value of `request.max_bytes` to the pending byte count.
- A new user message is processed - subtract 1 from pending messages count and subtract message size from pending byte count.
- A pull request termination status is received containing `Nats-Pending-Messages` and `Nats-Pending-Bytes` headers, subtract the value of `Nats-Pending-Messages` header from pending messages count

and subtract the value of `Nats-Pending-Bytes` from pending bytes count. Clients could just check all statuses for the headers to future proof.
    - 408 Request Timeout
    - 409 Message Size Exceeds MaxBytes
    - 409 Batch-Completed

Message Size Calculation

The message size (in bytes) should be calculated as the server does it. Size consists of:

- Data (payload + headers)
- Subject
- Reply subject

From `consumer.go`:

```
// Calculate payload size. This can be calculated on client side.
// We do not include transport subject here since not generally known on client.
sz = len(pmsg.subj) + len(ackReply) + len(pmsg.hdr) + len(pmsg.msg)
```

Status handling

In addition to providing termination `Nats-Pending-Messages` and `Nats-Pending-Bytes` headers, status messages indicate the termination of the pull. Statuses that are errors should be telegraphed to the user in language specific way. Telegraphing warnings is optional.

Errors:

- 400 Bad Request
- 409 Consumer Deleted
- 409 Consumer is push based

Warnings:

- 409 Exceeded MaxRequestBatch of %d
- 409 Exceeded MaxRequestExpires of %v
- 409 Exceeded MaxRequestMaxBytes of %v
- 409 Exceeded MaxWaiting

Not Telegraphed:

- 404 No Messages
- 408 Request Timeout
- 409 Message Size Exceeds MaxBytes

Calls to `next()` and `fetch()` should be concluded when the pull is terminated. On the other hand `consume()` should recover while maintaing its state (e.g. pending counts) by issuing a new pull request unless the status

is `409 Consumer Deleted` or `409 Consumer is push based` in which case `consume()` call should conclude in an implementation specific way idiomatic to the language being used.

Idle heartbeats

`Consume()` should always utilize idle heartbeats. Heartbeat values are calculated as follows:

A warning is triggered if the timer reaches 2 * request's idle_heartbeat value. The timer is reset on each received message (this can be either user message, status message or heartbeat message).

Heartbeat timer should be reset and paused in the event of client disconnect and resumed on reconnect.

Heartbeat errors are not terminal - they should rather be telegraphed to the user in language idiomatic way.

Server reconnects

Clients should detect server disconnect and reconnect.

When a disconnect event is received, client should:

- Pause the heartbeat timer.
- Stop publishing new pull requests.

When a reconnect event is received, client should:

- Reset the heartbeat timer.
- Publish a new pull request.

Clients should never terminate the `Consume()` call on disconnect and reconnect events and should not check if consumer is still available after reconnect.

Message processing algorithm

Below is the algorithm for receiving and processing messages. It does not take into account server reconnects and heartbeat checks - Process of handling those was described in previous sections.

1. Verify whether a new pull request needs to be sent:
    - pending messages count reaches threshold
    - pending byte count reaches threshold
2. If yes, publish a new pull request and add request's `batch` and `max_bytes` to pending messages and bytes counters.
3. Check if new message is available.
    - if yes, go to #4
    - if not, go to #1
4. Reset the heartbeat timer.
5. Verify the type of message:
    - if message is a heartbeat message, go to #1
    - if message is a user message, handle it (return or execute callback) and subtract 1 message from pending message count and message size from pending bytes count and go to #1
    - if message is an error, go to #6

6. Verify error type:
    - if message contains `Nats-Pending-Messages` and `Nats-Pending-Bytes` headers, go to #7
    - verify if error should be terminal based on Status handling, then issue a warning/error (if required) and conclude the call if necessary.
7. Read the values of `Nats-Pending-Messages` and `Nats-Pending-Bytes` headers.
8. Subtract the values from pending messages count and pending bytes count respectively.
9. Go to #1.

Info

An optional operation that returns the consumer info. Note that depending on the context (a consumer that is exported across account) the JS API to retrieve the info on the consumer may not be available.

Clients may optionally expose a way to retrieved cached info (from the Consumer instance itself), bypassing `CONSUMER.INFO` request to the server.

Delete

An optional operation that allows deleting the consumer. Note that depending on the context (a consumer that is exported across account) the JS API to delete the consumer may not be available.

Consequences

The new JetStream simplified consumer API is separate from the *legacy* functionality. The legacy functionality will be deprecated.