NATS Streaming Server Clustering Design Proposal

This document has been deprecated, but will be available as a reference for historical purposes and to refer to some past design decisions.

Author: Colin Sullivan

Overview

This design document describes a proposal for a clustering solution specific to NATS streaming, supporting data replication to provide hot failover to another server and distribute the overall workload.

In this document servers are considered **leader** or **followers**. The leader replicates data to followers.

The general idea: Leadership election will use a RAFT based algorithm, but replication will be a proprietary protocol that will take advantage of data already being sent by NATS clients. Both the leader and followers receive protocol messages from clients. The follower holds data until the leader sends information required to achieve consensus for persisted data (a form of history, usually a sequence number). In cases where the follower is catching up to achieve consistency, the follower will queue up incoming messages from clients while replicating old data and updates from the leader to then seamlessly transition into a steady replication state.

NATS streaming server clustering design takes the following ideas/assumptions into account:

- At least once delivery; duplicate messages are acceptable.
- Clustered servers are hot, replicating data in real-time.
- Any communication means of replication may be used. This could mean a direct connection between leader and follower cluster members, the NATS server cluster, or even a dedicated NATS server cluster. To begin with, the current NATS server cluster will be used.
- It is simpler to begin with clustering with granularity per channel. For example, one server may be the leader for channel A, but act as follower for channels B and C. This does not preclude future plans to partition channels themselves for further horizontal scalability.
- Cluster member configuration are identical.

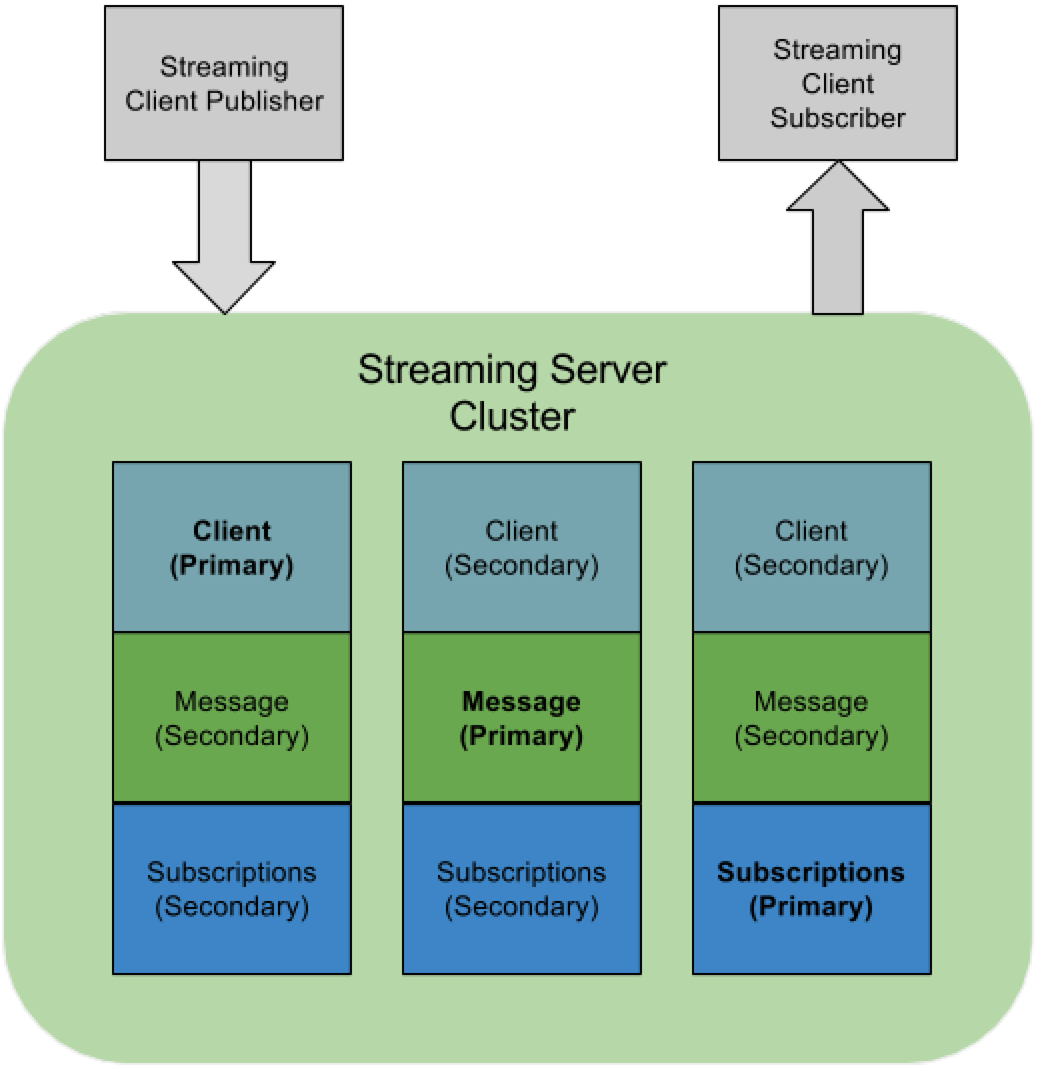
Design Priorities

1. Consistency - ordering and content is guaranteed to be consistent across cluster members, but lack of consistency is tolerable with subscriber acknowledgements to favor performance.
2. Partition Tolerance
3. Subscriber Performance
4. Performance to a lesser extent can be a user decision
    - Memory based (Performance) vs File based (Persistent)
    - **IDEA**: Hybrid Feature - memory based, but allow a user signal to persist for shutdown, or checkpoint? Speed + persistence for maintenance.

NATS Streaming Server Subsystems

NATS streaming can be broken down into three distinct subsystems for discussion as it relates to data replication. These each have distinct consensus protocols and optimization strategies. Message and Subscriber State sub systems operate by **channel**, in that each channel has it own leadership election and raft group. The Client State is global and that one server will be leader for all client state data in the cluster.

- **Message** - *messages clients have published that are stored in the cluster.*
- **Client State** - *the state of clients in the cluster (currently only connection state).*
- **Subscriber State** - *Subscription creation, close, and unsubscribes are handled here. Also included are acknowledged messages by subscribers, shared across servers in the cluster.*



Consensus Algorithms

NATS streaming servers will use RAFT (graft) for purposes of leadership election in each cluster (sub) group, but the use of RAFT stops there. This document will not cover the specifics of RAFT, except for the concept of history, as it will be used for leadership election. The levels of consensus we need for each subsystem will achieved through custom protocols detailed below for each subsystem. These take advantage of the feature that data is propagated by NATS without requiring the streaming server to broker protocol messages. With custom consensus algorithms, the high frequency of heartbeats typically found in RAFT can be reduced to minimize chatter, allowing the consensus algorithms to detect failures and initiate synchronization (catch-up). Graft will be extended to allow custom history comparison functions that will be used during voting for a leader.

Each subsystem has two states when acting as a follower:

Steady State

The system is up to date, consistent, and accepting real-time updates. A server in this state is a solid candidate for leader if leadership election gets triggered. The steady state protocol incoming records are first sent to a queue and then processed, to facilitate the transition from “catching up” to steady state. **Steady state protocol messages from the leader, for all subsystems will contain the RAFT term. If a leader detects a new term, it’ll step down to become a follower (this allows for recovery from split-brain scenario quicker than relying solely on heartbeats).**

Catching Up

The server is inconsistent and is rebuilding its state. It queues incoming steady state updates to seamlessly assume steady state after caught up. In this state, if an event triggers leadership election, this server will not assume leadership and will return to the catching-up state after a new leader has been elected.

Heartbeats

Heartbeats between cluster members will let the system detect a loss of quorum faster, mitigating and minimizing disruption to clients. We’ll use RAFT-like heartbeats configured to use custom history, and associated rules (e.g. a newer term, etc). To parallel RAFT, these could be empty steady state protocol messages.

- Idea: Allow slower configurable heartbeats to minimize chatter, but we’d want to provide a way to trigger leadership election by extending an API on RAFT to trigger leadership election.

Leadership Changes

TODO: Tie raft data to our custom synchronization protocol. Include each state, various scenarios (e.g. split-brain). This will be centered around detecting term inconsistencies.

History

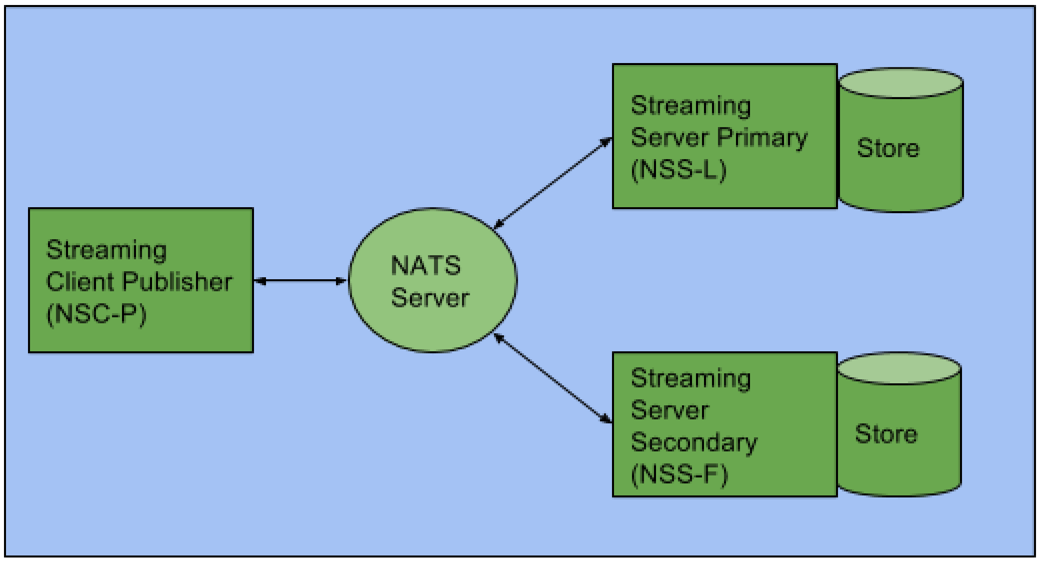
In order to maintain consistency, each subsystem needs the concept of history to determine leadership for a subsystem. It is simple in most cases. Most events that need to be synchronized are stored in a sub system will be published with a **GUID** (a NUID). ***This is a client protocol change.*** The leader will assign a sequence number (64 bit unsigned integer) that is incremented for each event (message or action), and will send this to the secondaries with the GUID to all followers. Following servers use the GUID to correlate received messages/events with the id provided by the leader, to persist the event with the ID with consistency. The GUID is no longer needed after this point and is not persisted.

Sub System	**Granularity**	**History**
Client State	Global	Sequence #
Messages	Channel	Sequence #
Subscriber	Channel	Sequence # then Ack Count A subscriber sequence number number representing the general state of subscribers (*active, closed, unsubscribed*), determines leadership. In the case of a tie where subscriber states are consistent, the highest count of persisted acks is used. This reduces resends.

Diagram Legends

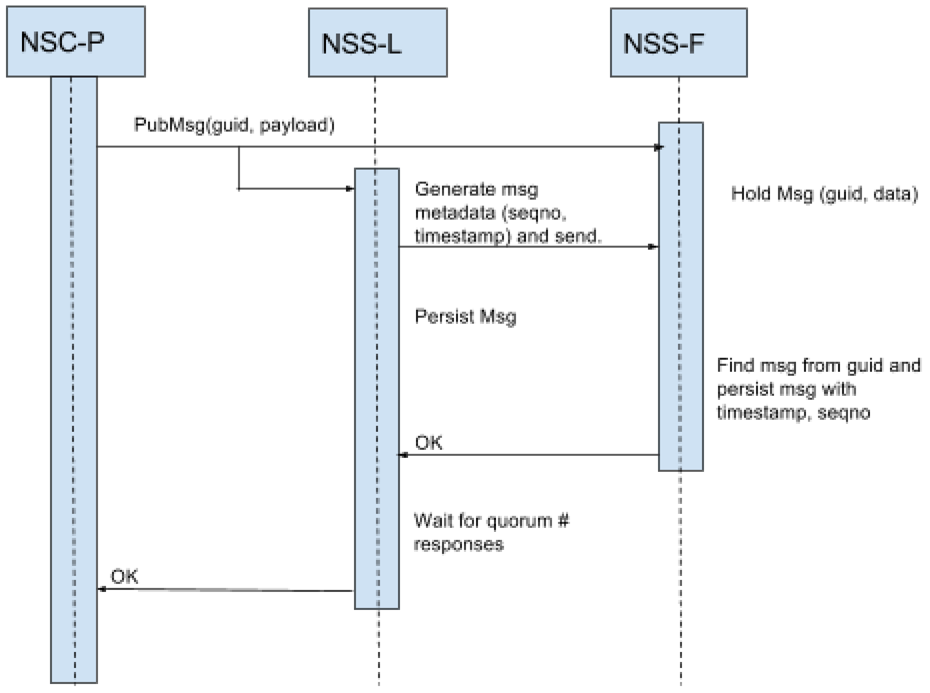
In the diagrams throughout this document, some abbreviations are used, and represent components as follows:
NSC-P: NATS Streaming Client Publisher
NSC-S: NATS Streaming Client Subscriber
NSS-L: NATS Streaming Server Leader (for the subsystem being described)
NSS-F: NATS Streaming Server Follower (for the subsystem being described)

Steady State Message Replication Protocol


Streaming Client Publisher (NSC-P)
NATS Server
Streaming Server Primary (NSS-L)
Store
Streaming Server Secondary (NSS-F)
Store

This message replication protocol is pessimistic in that the assumption is made where multiple publishers on the same channel create a relatively high probability of receiving messages in different order amongst members of a cluster. The leader and followers will create a channel as necessary when persisting a message.

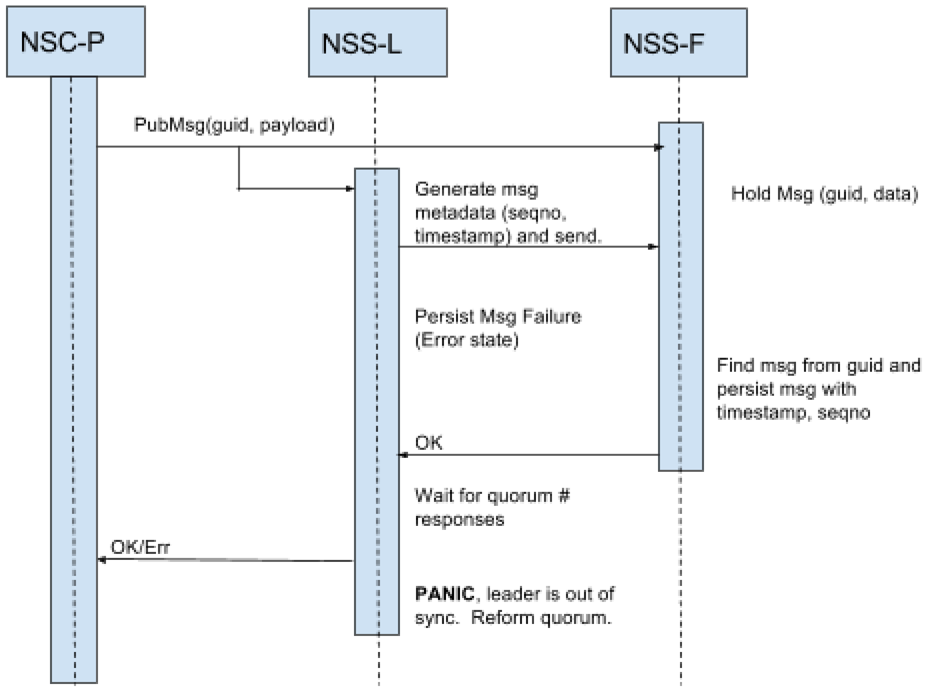
Message Replication Protocol (Success Case for Leader and Follower)


NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Generate msg metadata (seqno, timestamp) and send.
Hold Msg (guid, data)
Persist Msg
Find msg from guid and persist msg with timestamp, seqno
OK
Wait for quorum # responses
OK

Description

1. The streaming publisher publishes a message
2. The message is received by **all** streaming servers
    a. The leader generates message metadata, and sends the metadata and the GUID to all followers, the persist the message locally. *This can be through the NATS server, or another type of communication channel, TBD.* The **StoreRequest** protocol message includes the guid, timestamp and channel sequence number.
    b. Secondary servers hold the message and waits for the leader's **StoreRequest** protocol messages under a timeout. Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata from the **StoreRequest** protocol, correlate the payload with the guid, and persist the message with the timestamp, sequence number, and data.
4. The follower sends a success response to the leader.
5. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error)
6. The leader returns a success to the streaming publisher.

Message Replication Protocol (Leader fails to persist)

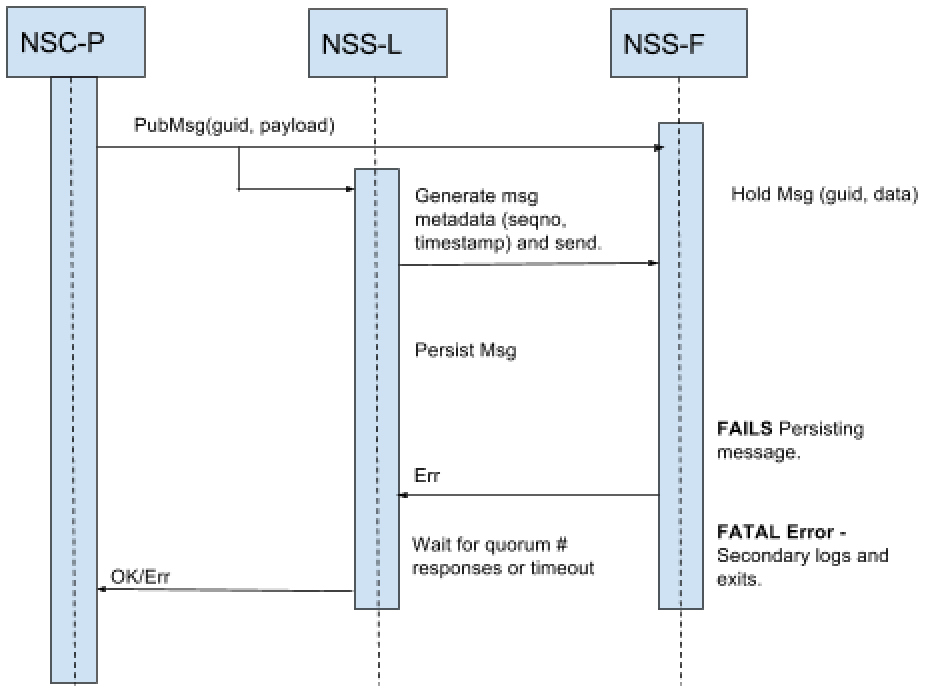

NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Generate msg metadata (seqno, timestamp) and send.
Hold Msg (guid, data)
Persist Msg Failure (Error state)
Find msg from guid and persist msg with timestamp, seqno
OK
Wait for quorum # responses
OK/Err
PANIC, leader is out of sync. Reform quorum.

Description

1. The streaming publisher publishes a message
2. The message is received by **all** streaming servers
    a. The leader generates message metadata, and sends the metadata and the GUID to all followers
    b. The leader **fails to persist the message.**
    c. Secondary servers hold the message and waits for the leader's **StoreRequest** protocol messages under a timeout. Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata(guid, timestamp, and sequo) from the **StoreRequest** protocol, correlate the payload with the guid, and persist the message with the timestamp, sequence number, and data.
4. The follower sends a success response to the leader.
5. The leader is in a fatal error condition and **PANICS** (persistence errors are considered non-recoverable).
6. Quorum reforms.

Message Replication Protocol (Error: Secondary fails persisting)

This describes the error condition where the follower cannot persist data.

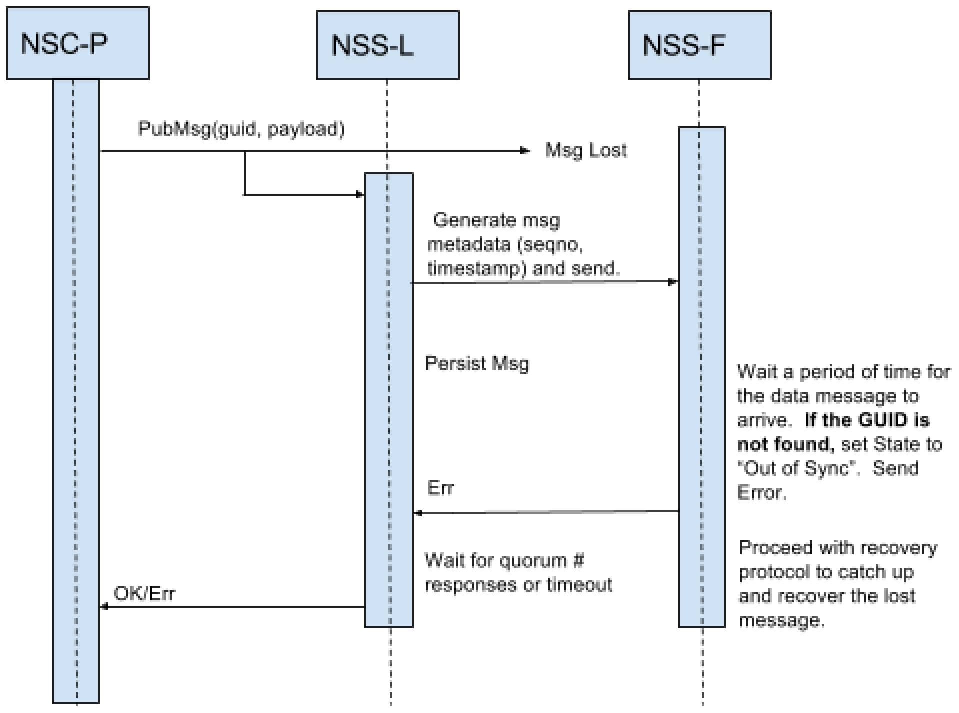

NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Generate msg metadata (seqno, timestamp) and send.
Hold Msg (guid, data)
Persist Msg
FAILS Persisting message.
Err
FATAL Error - Secondary logs and exits.
Wait for quorum # responses or timeout
OK/Err

Description

1. The streaming publisher publishes a message
2. The message is received by **all** streaming servers
    a. The leader generates message metadata, and send the metadata and the GUID to all followers, the persist the message locally, as in the success case.
    b. Secondary servers hold the message and waits for the leader's **StoreRequest** protocol messages under a timeout.
3. A follower receives the metadata from the **StoreRequest** protocol, correlate the payload with the guid, and but fails persist the message with the timestamp, sequence number, and data.
4. The follower fails persisting the data.
5. The follower sends an error to the leader, logs an error, then exits.

Message Replication Protocol (Error: Secondary missing publisher message)

This describes the error condition where the follower receives a request to persist a message it does not have.

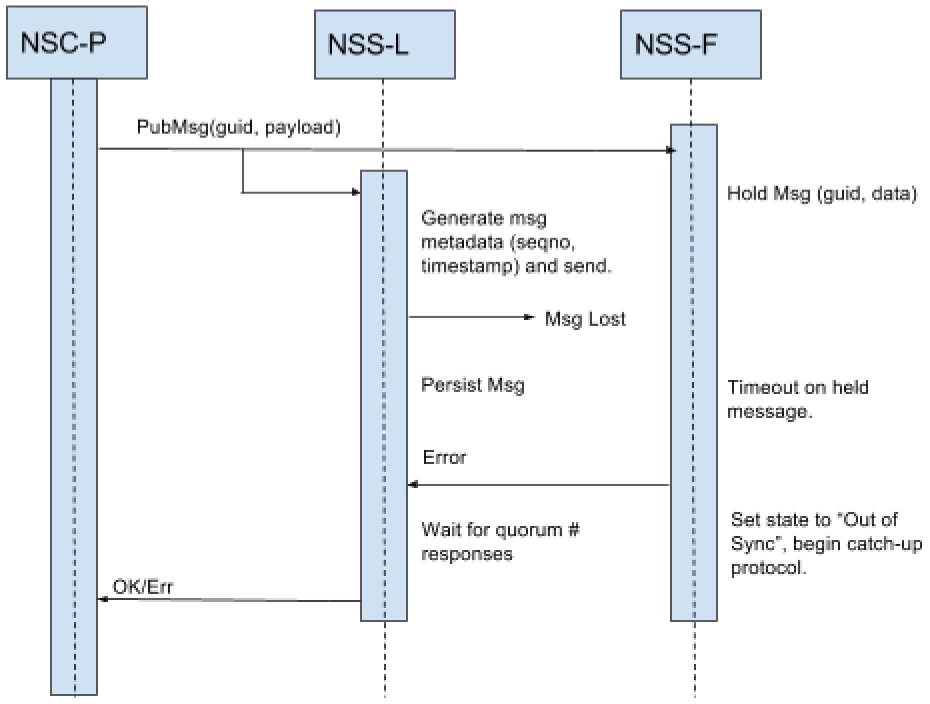

NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Msg Lost
Generate msg metadata (seqno, timestamp) and send.
Persist Msg
Wait a period of time for the data message to arrive. **If the GUID is not found,** set State to "Out of Sync". Send Error.
Err
Wait for quorum # responses or timeout
OK/Err
Proceed with recovery protocol to catch up and recover the lost message.

Description

1. The streaming publisher publishes a message
2. The message is received by the leader but **not** the follower.
    a. The leader generates message metadata, and sends the metadata and the GUID to all followers, the persist the message locally as in the success case.
3. The follower receives a **StoreRequest** with a GUID for a message it does not have.
4. The follower waits a period of time for the publishers PubMsg with the corresponding GUID to arrive. It does not.
5. The follower sets its state to "out of sync", and sends an error to the leader.
6. Secondary server begins to catch up to return to steady state replication.

Message Replication Protocol (Error: Secondary misses StoreRequest)

This describes the error condition where the follower does not receive the StoreRequest message message from the leader.

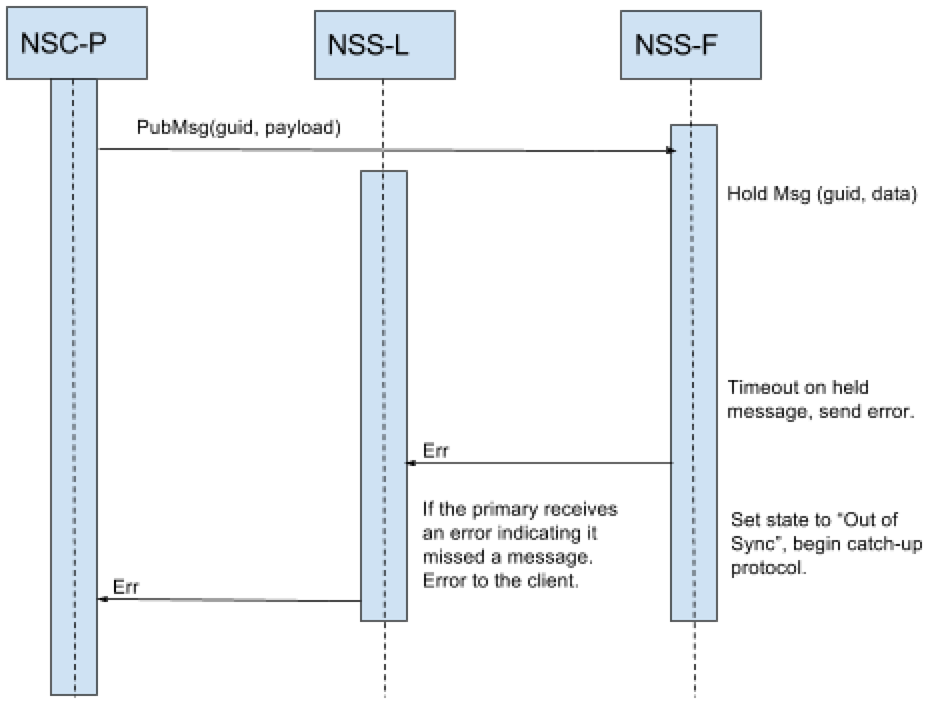

NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Hold Msg (guid, data)
Generate msg metadata (seqno, timestamp) and send.
Msg Lost
Persist Msg
Timeout on held message.
Error
Wait for quorum # responses
Set state to "Out of Sync", begin catch-up protocol.
OK/Err

Description

1. The streaming publisher publishes a message
2. The message is received by **all** streaming servers
    a. The leader generates message metadata, and sends the metadata and the GUID to all followers, the persist the message locally. *This can be through the NATS server, or another type of communication channel, TBD.* The **StoreRequest** protocol message includes the guid, timestamp and channel sequence number.
    b. Secondary servers hold the message and waits for the leader **StoreRequest** protocol messages under a timeout.
3. The follower does not receive the the **StoreRequest** protocol message, and the timeout is invoked.
4. Secondary sends an error message to the leader.
5. The follower sets its state to "out of sync", begins to catch up to return to steady state replication.

Message Replication Protocol (Error: Leader misses PubMsg)

This describes the error condition where the leader misses the PubMsg, but the follower receives one.

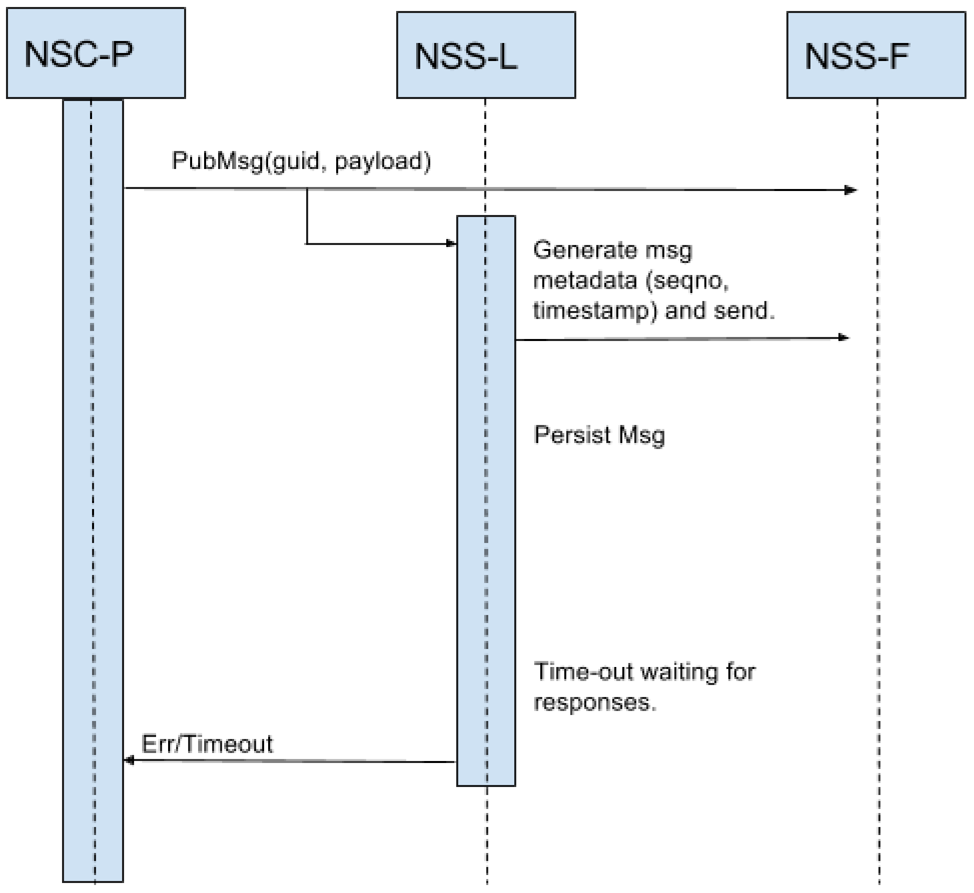

NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Hold Msg (guid, data)
Timeout on held message, send error.
Err
If the primary receives an error indicating it missed a message. Error to the client.
Set state to "Out of Sync", begin catch-up protocol.
Err

Description

1. The streaming publisher publishes a message.
2. The message is received by **follower** streaming servers, the leader misses it.
    a. Secondary servers hold the message and waits for the leader **StoreRequest** protocol messages under a timeout.
3. The follower's timeout is invoked because it has not received a **StoreRequest** for a guid.
4. The follower sends an error to the leader. (Investigate: Is this necessary? Catch up protocol could remove the last message if a leader isn't aware of it, and the client would simply resend the message upon erroring)
5. The leader detects it is receiving an error for a guid it does not have. It does this by keeping outstanding **StoreRequests** it has sent to correlate with responses. The leader is out of sync, and invalidates itself as leader.

Message Replication Protocol (Secondary servers fail to respond)

This case describes the situation where a follower fails to respond.


NSC-P
NSS-L
NSS-F
PubMsg(guid, payload)
Generate msg metadata (seqno, timestamp) and send.
Persist Msg
Time-out waiting for responses.
Err/Timeout

Description

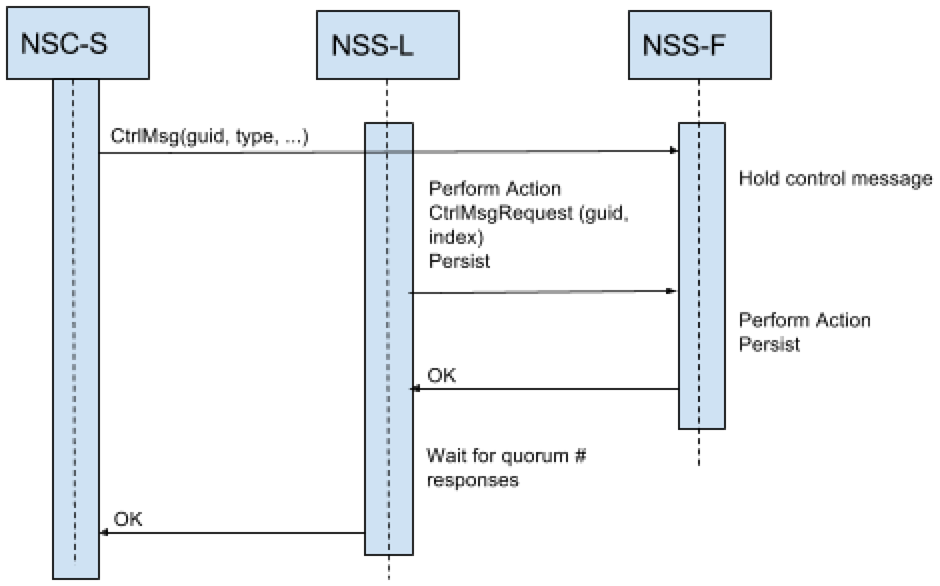
1. The streaming publisher publishes a message
2. The message is received by the leader, and an insufficient # of followers.
    a. The leader generates message metadata, and sends the metadata and the GUID to all followers, the persist the message locally. *This can be through the NATS server, or another type of communication channel, TBD.* The **StoreRequest** protocol message includes the guid, timestamp and channel sequence number.
3. The leader times out waiting for the required number of responses from followers.
4. The leader sends a failure message to the publisher.

* Note: The leader will be out of sync with secondaries, and have a persisted message. We can clear the message, or as it the case today, keep the message - the client may send a duplicate, but that is OK.

Steady State Client State Replication

Client state replication is directed by the leader. Client state includes events such as a client connecting, and closing a connection.

Client State Replication Protocol (Success Case for Leader and Secondary)

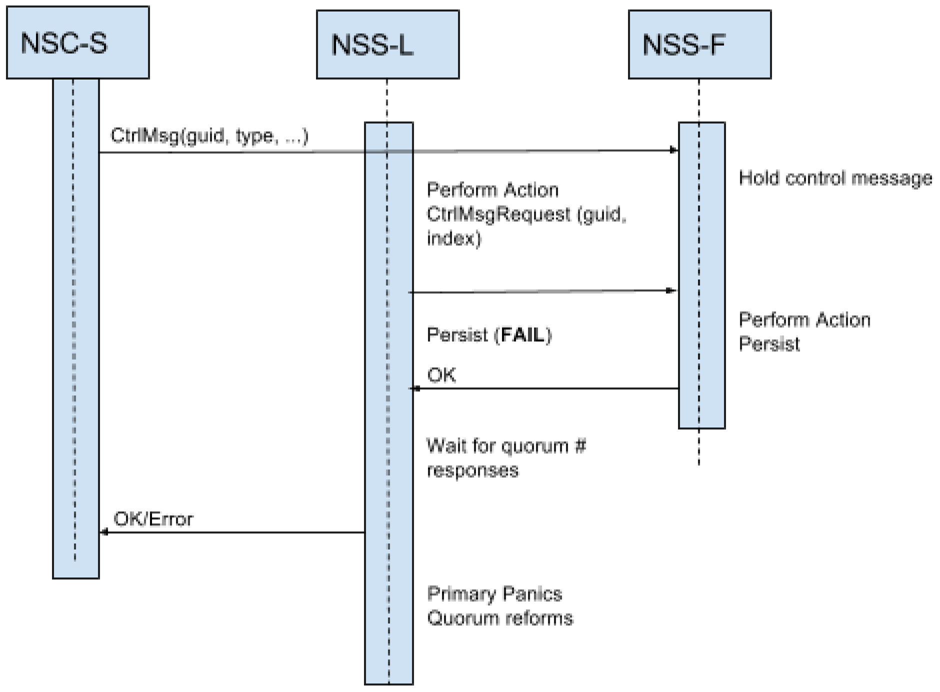


This Control and Client Message protocol covers client create (connection open) and client delete (connection close). Description

1. A client issues a control request (create, close connection). The leader and followers receive the message.
    a. The action is performed and succeeds in the leader. An ID (index) is generated to represent history.
    b. The message is persisted by the leader.
    c. The follower holds the message until the leader sends a control request.
2. The follower receives a control request.
    a. The follower performs the action, successfully persists the message, and sends an OK to the leader
3. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error) (discussed later)
4. The action succeeds and the client is notified of success.

Client State Replication Protocol (Persistence failure for Leader)

This Control and Client Message protocol covers client create and delete. This is the failure case where the leader cannot persist the message.

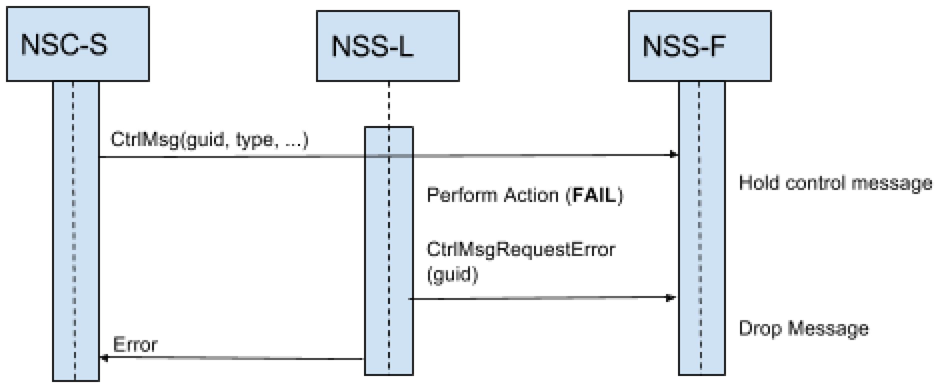


Description

1. A client issues a control request and all servers receive the message.
2. The leader performs the action.
3. The leader send control message requests to the follower (which succeeds).
4. During step the leader fails to persist the action.
5. The leader tallies quorum responses, and returns the appropriate OK/error to the client.
6. The leader has had a persistence failure, and **PANICS**, forcing the quorum to reform and and another server to become leader.

Client State Replication Protocol (Action failure for Leader)

This Control and Client Message protocol covers client create and delete. This is the failure case where the leader cannot complete the action.

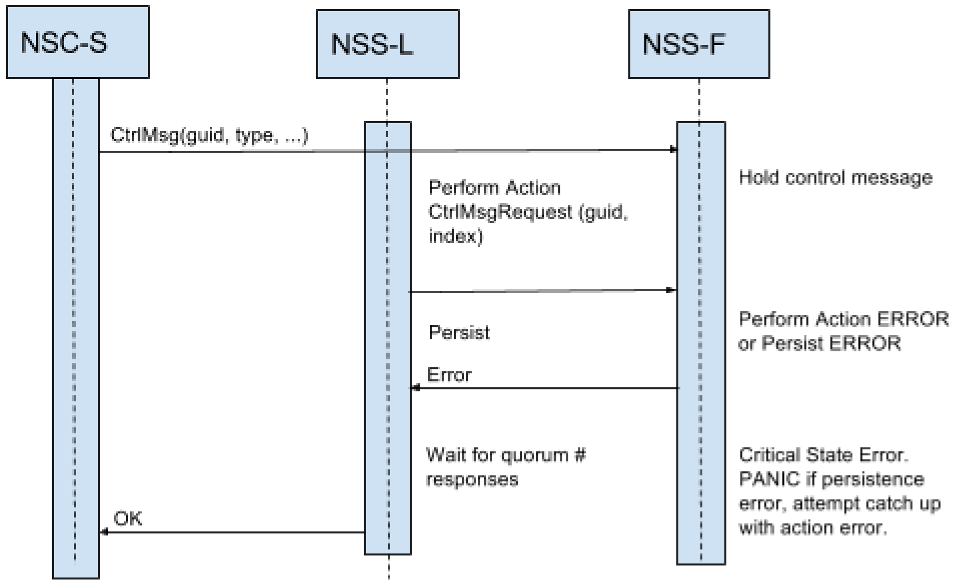


Description

1. A client issues a control request and all servers receive the message.
2. The leader cannot perform the action due to a state error with the client (duplicate ID, invalid client id, etc).
3. The leader send control message requests to the follower to drop the message.

Client State Replication Protocol (Failure action/persistence fails for follower)

This Control and Client Message protocol covers client create and delete. This is the failure case where the follower cannot persist or process the message. These could be two different cases, but one seems to work.

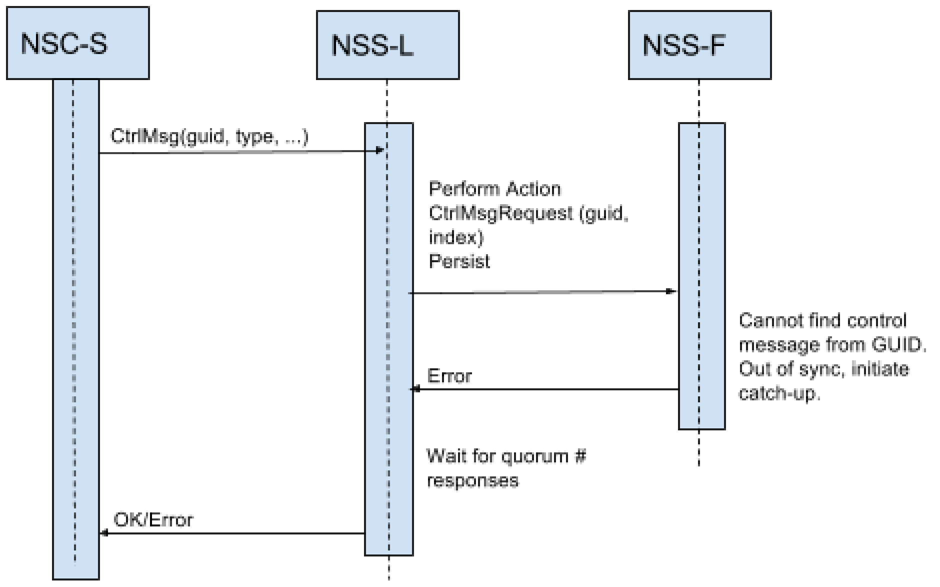

NSC-S
NSS-L
NSS-F
CtrlMsg(guid, type, ...)
Hold control message
Perform Action CtrlMsgRequest (guid, index)
Persist
Perform Action ERROR or Persist ERROR
Error
Wait for quorum # responses
Critical State Error. PANIC if persistence error, attempt catch up with action error.
OK

Description

1. A client issues a control request (unsubscribe, close subscription, close connection), and the leader receives the message.
2. The action is performed and succeeds
3. The leader sends a control request to the follower
4. The follower fails with the action or persisting the client control message.
    a. The follower sends an error message back to the leader to fail fast.
    b. Process failure
        i. If it is a store failure (persistence), **PANIC**.
        ii. If it is an action error, there is a critical state error. Rebuild the state through catching up and rebuild the entire client store. This likely means a critical message has been missed someplace.
5. The leader persists the client state
6. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error) (discussed later)
7. If not enough success messages are aggregated, an error is returned to the client.

Client State Replication Protocol (Failure: Follower did not receive the client state message)

This Control and Client Message protocol covers client create and delete. This is the failure case where the follower does not receive the message, or crashes processing it.

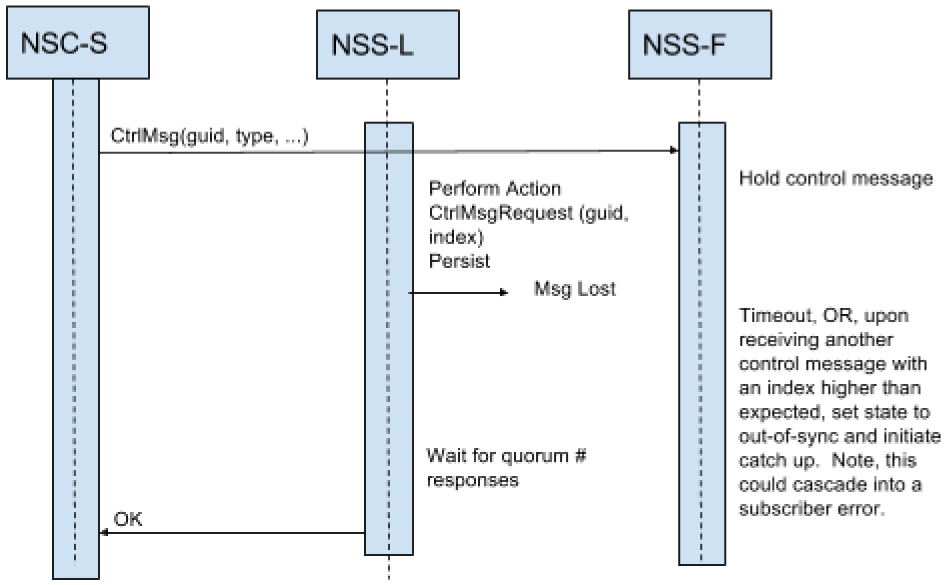

NSC-S
NSS-L
NSS-F
CtrlMsg(guid, type, ...)
Perform Action
CtrlMsgRequest (guid, index)
Persist
Cannot find control message from GUID. Out of sync, initiate catch-up.
Error
Wait for quorum # responses
OK/Error

Description

1. A client issues a control request (unsubscribe, close subscription, close connection), and the leader receives the message.
2. The action is performed and succeeds
3. The leader sends a control request to the follower
4. The follower discovers that it hasn't receive the control request
5. Follower sends an error, sets its state to out-of-sync, and begins the catch-up protocol.
6. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error) (discussed later)
7. If not enough success messages are aggregated, an error is returned to the client.

Client State Replication Protocol (Follower does not receive Control Message Request)

This Control and Client Message protocol covers client create and delete. This is the failure case where the follower cannot persist or process the message. These could be two different cases, but one seems to work.

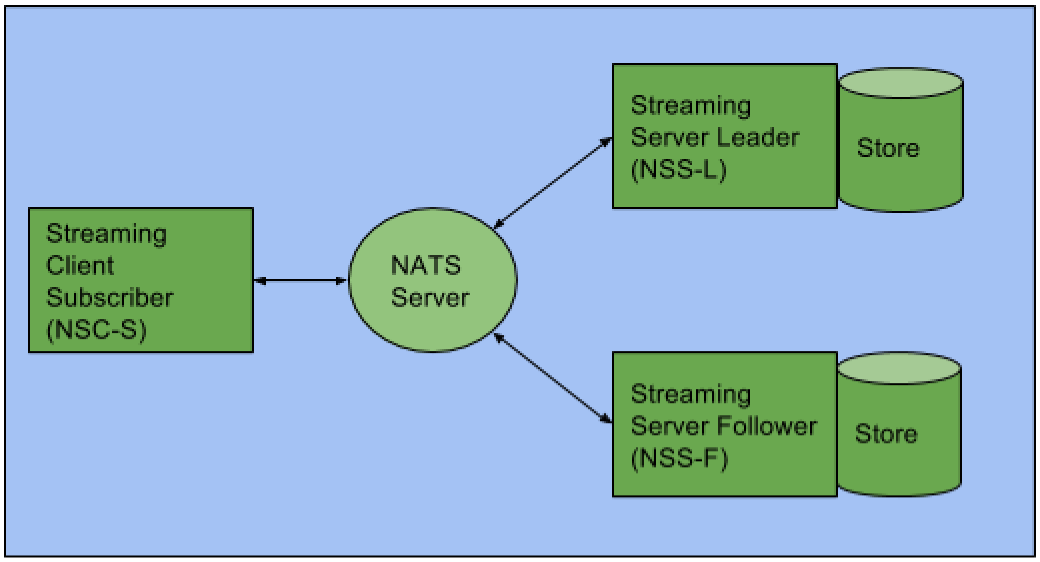

NSC-S
NSS-L
NSS-F
CtrlMsg(guid, type, ...)
Hold control message
Perform Action
CtrlMsgRequest (guid, index)
Persist
Msg Lost
Timeout, OR, upon receiving another control message with an index higher than expected, set state to out-of-sync and initiate catch up. Note, this could cascade into a subscriber error.
Wait for quorum # responses
OK

Description

1. A client issues a control request and both the leader and follower receives the message.
2. The action is performed and succeeds on the leader.
3. The leader sends a control request to the follower, which is lost.
4. The follower either:
    a. Times out on outstanding control message
    b. Receives another control message with an index indicating one was missed.
5. The follower sets its state to out-of-sync and initiates the catch up protocol.

Note: The subscriber will need to be resilient to subscriptions that do not have a connection associated with them. **The two subsystems are dependent on each other, and a client catchup will need to trigger a subscription catch up.**

Steady State Subscriber Replication


Streaming Server Leader (NSS-L)
Store
Streaming Client Subscriber (NSC-S)
NATS Server
Streaming Server Follower (NSS-F)
Store

Subscriber replication is the most complex, but most tolerant of errors. Given that this area will likely have the most impact on performance, the protocol design intends to keep the state close amongst cluster members, but not entirely consistent to favor performance.

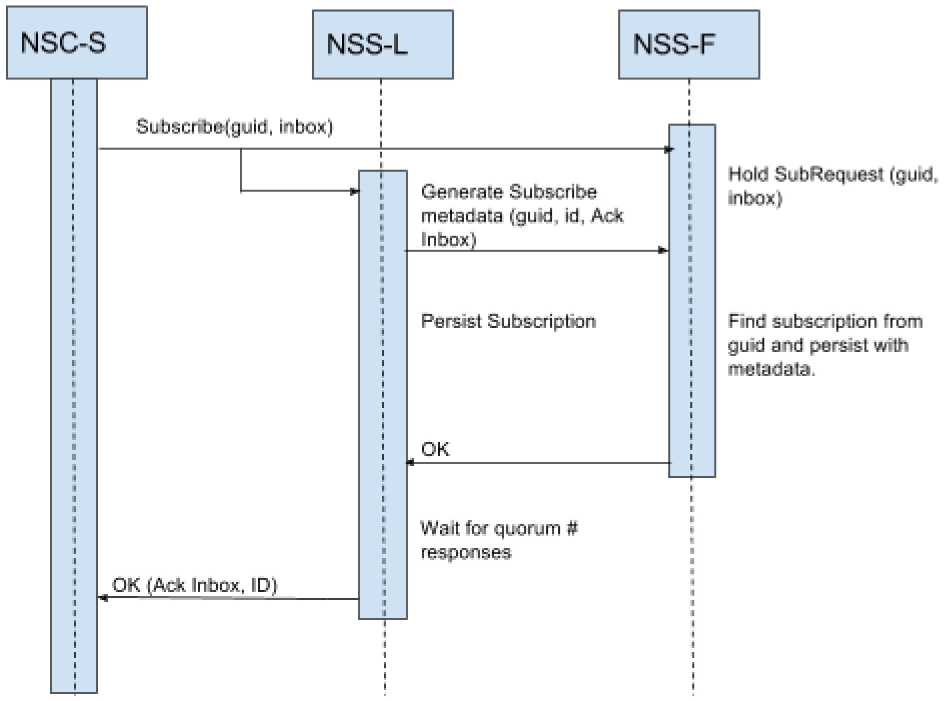
The following is assumed:

- Duplicate messages are acceptable
- The redelivered flag is not guaranteed to be correct
- Subscriber acknowledgements can be lost

We can provide an optimistic approach to handling subscriber acknowledgements that favors performance over correctness, given that inconsistencies will simply result in a message being re-delivered. If a channel does not exist, it will be created when a subscription is made on the associated topic.

Only the leader will send messages to subscribers. However, each server in the cluster receives client acknowledgements, and updates their stores accordingly. The leader will publish periodic updates of subscriber states to maintain a best-effort consistency with followers. It is understood that followers can miss subscriber acknowledgements, and when a follower is elected leader in the future, it may redeliver messages that have already been sent.

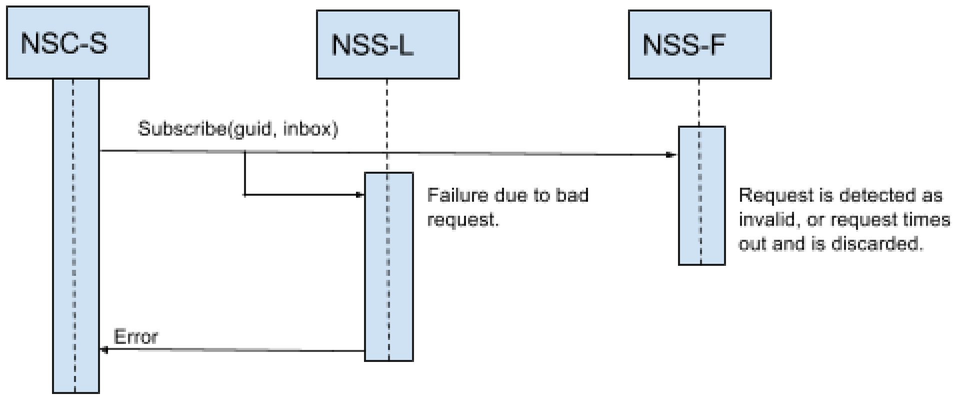
Subscriber SubRequest Protocol (Success Case for Leader and Follower)


NSC-S
NSS-L
NSS-F
Subscribe(guid, inbox)
Hold SubRequest (guid, inbox)
Generate Subscribe metadata (guid, id, Ack Inbox)
Persist Subscription
Find subscription from guid and persist with metadata.
OK
Wait for quorum # responses
OK (Ack Inbox, ID)

Description

1. The streaming client publishes a **SubscribeRequest** message. A **NUID** is provided (this is new to the protocol, but will address some future problems).
2. The **SubscribeRequest** message is received by **all** streaming servers
    a. The leader generates metadata, and sends the metadata and the GUID to all followers, then persists the message locally. *This can be through the NATS server, or another type of communication channel, TBD.* The **SubStoreRequest** protocol message includes the guid, id, and ack inbox.
    b. Followers hold the message and waits for the leader **SubStoreRequest** protocol messages under a timeout. Subscription Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata from the **SubStoreRequest** protocol, correlate the subscription with the guid, and persist the data with the ID and AckInbox. The streaming server will subscribe to the ack inbox to start processing acknowledgements.
4. The follower sends a success response to the leader.
5. The leader aggregates responses a quorum has responded returning a success to the subscribing client.

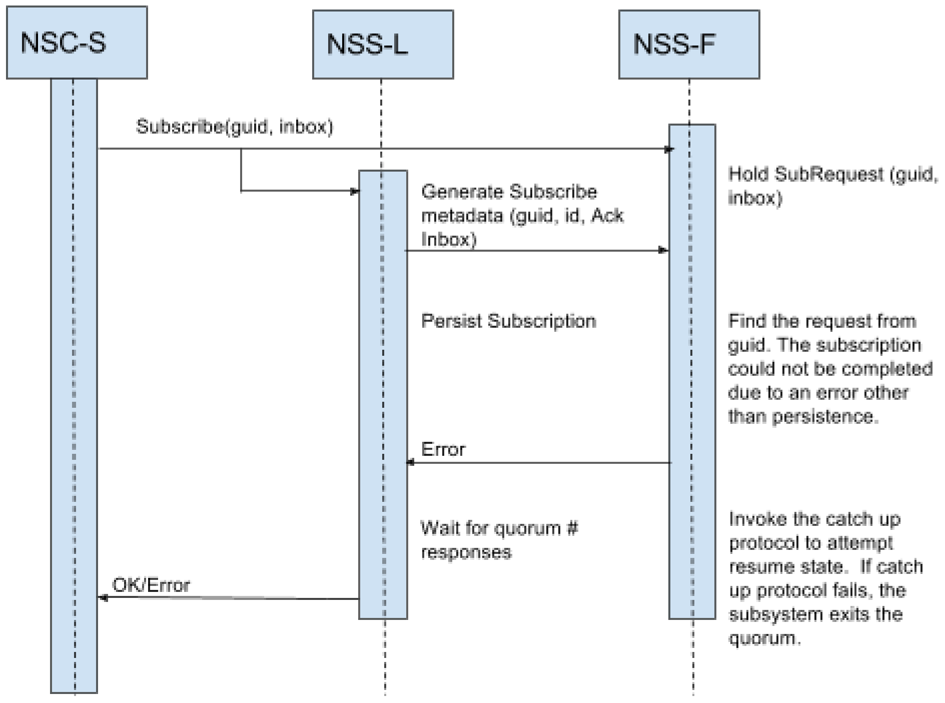
Subscriber SubRequest Protocol (Bad Request Failure for Leader)


NSC-S
NSS-L
NSS-F
Subscribe(guid, inbox)
Failure due to bad request.
Request is detected as invalid, or request times out and is discarded.
Error

Description

1. The streaming client publishes a **SubscribeRequest** message.
2. The **SubscribeRequest** message is received by **all** streaming servers
    a. The leader cannot honor the subscription request due to a bad request. An error is returned to the client immediately.
    c. Followers either detect the invalid request, or hold the message and eventually time out because they did not receive a **SubStoreRequest** protocol messages. *A request could be invalid in the leader and valid in a follower if different configurations were used, namely the maximum number of subscriptions has been exceeded.*

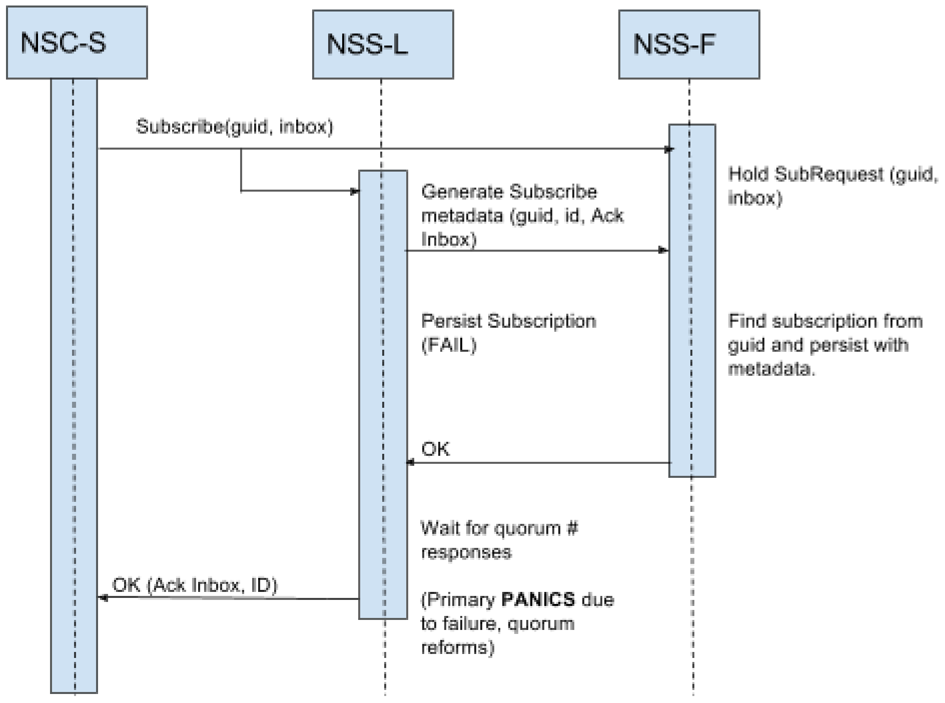
Subscriber SubRequest Protocol (Bad Request Case for Follower)


NSC-S
NSS-L
NSS-F
Subscribe(guid, inbox)
Hold SubRequest (guid, inbox)
Generate Subscribe metadata (guid, id, Ack Inbox)
Persist Subscription
Find the request from guid. The subscription could not be completed due to an error other than persistence.
Error
Wait for quorum # responses
Invoke the catch up protocol to attempt resume state. If catch up protocol fails, the subsystem exits the quorum.
OK/Error

Description

1. The streaming client publishes a **SubscribeRequest** message. A **GUID** is provided (this is new to the protocol, but will address some future problems).
2. The **SubscribeRequest** message is received by **all** streaming servers
    a. The leader generates metadata, and sends the metadata and the GUID to all followers, the persist the message locally. The **SubStoreRequest** protocol message includes the guid, id, and ack inbox.
    b. Secondarys hold the message and waits for the leader **SubStoreRequest** protocol messages under a timeout. Subscription Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata from the **SubStoreRequest** protocol, correlate the subscription with the guid, and attempt to create the Subscriber.
4. Subscriber creation fails
    a. Likely, this is a configuration error (maximum subscriptions), but it could be a state error where the subscription subsystem if out of sync. Catch up protocol is invoked. If there are errors in the catch up protocol, the subsystem removes itself from the quorum.
5. The follower sends a success response to the leader.
6. The leader aggregates responses and returns a success or failure to the streaming publisher.

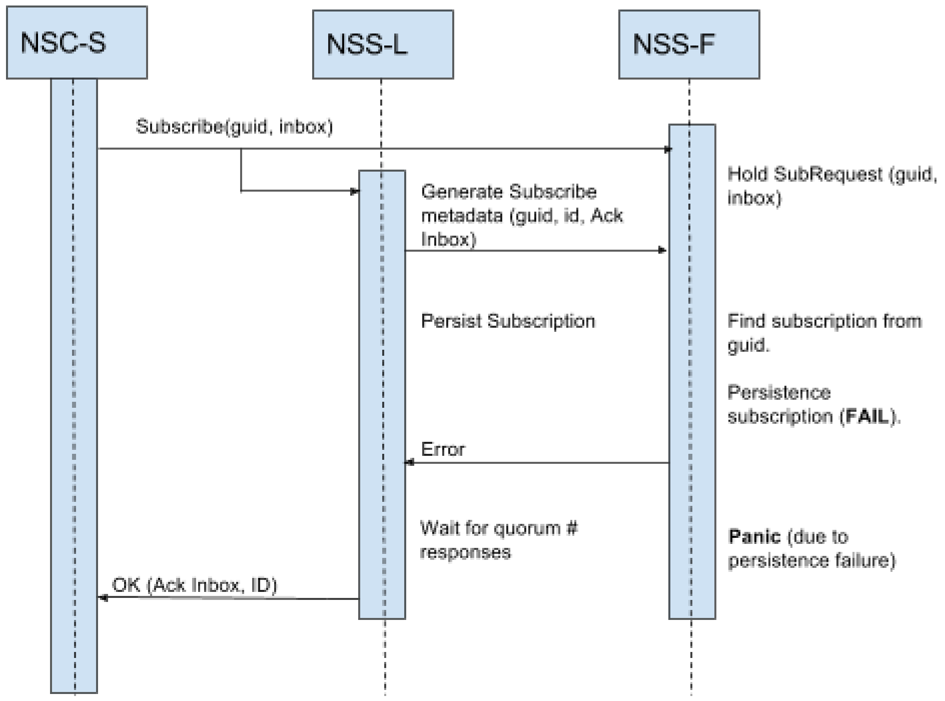
Subscriber SubRequest Protocol (Persistence failure for Leader)


NSC-S
NSS-L
NSS-F
Subscribe(guid, inbox)
Hold SubRequest (guid, inbox)
Generate Subscribe metadata (guid, id, Ack Inbox)
Persist Subscription (FAIL)
Find subscription from guid and persist with metadata.
OK
Wait for quorum # responses
OK (Ack Inbox, ID)
(Primary **PANICS** due to failure, quorum reforms)

Description

1. The streaming client publishes a **SubscribeRequest** message. A **GUID** is provided (this is new to the protocol, but will address some future problems).
2. The **SubscribeRequest** message is received by **all** streaming servers
    a. The leader generates metadata, and sends the metadata and the GUID to all followers. The **SubStoreRequest** protocol message includes the guid, id, and ack inbox. Persistence **FAILS**.
    b. Followers hold the message and waits for the leader **SubStoreRequest** protocol messages under a timeout. Subscription Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata from the **SubStoreRequest** protocol, correlate the subscription with the guid, and persist the data with the ID and AckInbox. The streaming server will subscribe to the ack inbox to start processing acknowledgements.
4. The follower sends a success response to the leader.
5. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error)
6. The leader returns the result to the streaming publisher.
7. Because of a fatal persistence error, the leader panics causing the quorum to reform.

Subscriber SubRequest Protocol (Persistence Failure for Follower)

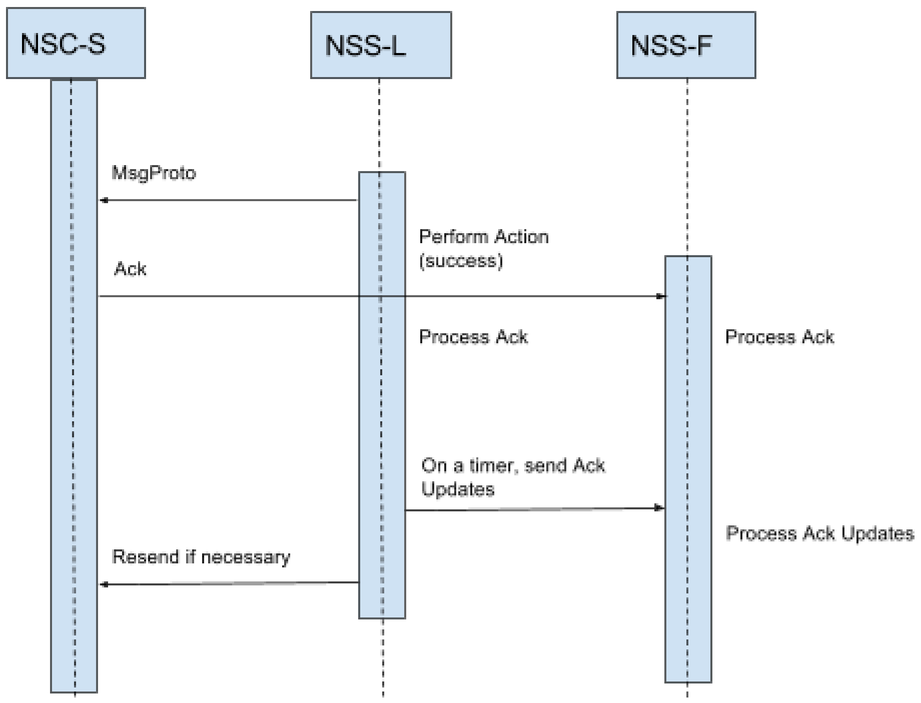

NSC-S
NSS-L
NSS-F
Subscribe(guid, inbox)
Hold SubRequest (guid, inbox)
Generate Subscribe metadata (guid, id, Ack Inbox)
Persist Subscription
Find subscription from guid.
Persistence subscription (FAIL).
Error
Wait for quorum # responses
Panic (due to persistence failure)
OK (Ack Inbox, ID)

Description

1. The streaming client publishes a **SubscribeRequest** message. A **GUID** is provided (this is new to the protocol, but will address some future problems).
2. The **SubscribeRequest** message is received by **all** streaming servers
    a. The leader generates metadata, and sends the metadata and the GUID to all followers, the persist the message locally. The **SubStoreRequest** protocol message includes the guid, id, and ack inbox.
    b. Follower servers hold the message and waits for the leader **SubStoreRequest** protocol messages under a timeout. Subscription Store requests are put into a queue and dequeued to be processed (this is for the catch-up protocol, described later).
3. All followers receive the metadata from the **SubStoreRequest** protocol, correlate the subscription with the guid, but persisting the data into the store request **FAILS**.
4. The follower sends a failure response to the leader.
5. The leader aggregates responses until:
    a. A quorum has responded
    b. A timeout occurs (error)
6. The follower panics due to the persistence failure, taking it out of the quorum.

Subscriber Acknowledgement Protocol (Success Case)

This is a diagram of the success case for a subscriber acknowledgment.



Description

1. Leader sends a message to the client.
2. Client acknowledges
3. Leader and Follower receive the acknowledgement.
    a. They update their respective stores
4. Only the leader resends unacknowledged messages.

Acknowledgment Updates from the Leader

Acknowledgement updates are an optimization to minimize resends. This is not required, but can be added if through investigation we find that clients are receiving an unacceptable number of resent messages.

The leader will periodically send an **AckUpdate** message to the followers. This is a best effort attempt to maintain consistency. It has the following fields:

- Channel
    - Name
    - Array of sequence numbers that have been acknowledged since the last **AckUpdate**.
    - Last published seqno.
- AckUpdate Sequence - Integer representing this particular ack update. Used to detect missing **AckUpdate** messages.

When a follower receives an **AckUpdate**, any pending acknowledgements that it has not received an acknowledgement for it tallies and stores as being acknowledged, eliminating a potential resend. This will require a follower to hold in memory a list of pending acknowledgement sequence numbers. Note, this will create a subscriber log that is likely inconsistent with the leader. Those inconsistencies will be resolved when the server compacts the stores. See Compaction for more information.

Subscriber Acknowledgement Protocol Failure Cases

There are a number of failure cases, but they are straightforward to handle.

Failure Case	**Action**
The leader does not receive the acknowledgement and the follower does.	**None**. Leader is out of sync and may resend the message which is OK. The follower is valid. If the leader fails in this state, the follower would assume leadership and have the correct data.
The follower receives a duplicate acknowledgement	**None**. The leader was out of sync and resent the message to the client, who acked. Ignore.
The follower does not receive the acknowledgement	**None**. The follower is out of sync, and one of the following will occur. ● Is synchronized by the **AckUpdate**, to become consistent. ● If a failure causes the follower to assume leadership, it may resend the message to a client, which is OK.
The follower fails to find the channel to acknowledge	**Catch-up**. The follower missed a subscription message update, and initiates the catch up protocol.
The follower detects missing **AckUpdates**.	**Catch-up?** There are likely pretty big problems going on. Another subsystem interruption likely would have triggered a quorum reformation.

Pending Acknowledgements

Pending acknowledgements are records persisted that account for messages sent to subscribers that have not yet been acknowledged. Their basic purpose is to trigger a resend messages that have not been acknowledged within a time period. In a cluster, all servers will receive pending acknowledgements from clients, but there is no steady state replication of these records per se. Only the leader will send or resend messages to subscribers.

The process of sending messages to a subscriber can be described to be triggered by a *sendToSubscriber* event internal to each server. This event could be triggered from subscriber creation or during the processing of incoming messages. For both leader and followers, subscriber creation triggers this event using the same mechanism. However, during the processing of incoming messages, this event generation differs based on the role of the message subsystem.

If the message subsystem is leader, a *sendToSubscriber* event is generated after the message is persisted and the **StoreRequest** message has been sent. If the message system is in a follower state, this occurs after the streaming message receives the **StoreRequest** and has persisted the incoming message. *Note, this may create a conditions where a message is successfully persisted, but the client receives an error. This condition exists today in single node, and can result in the client resending a message. Is this false negative to the client OK?*

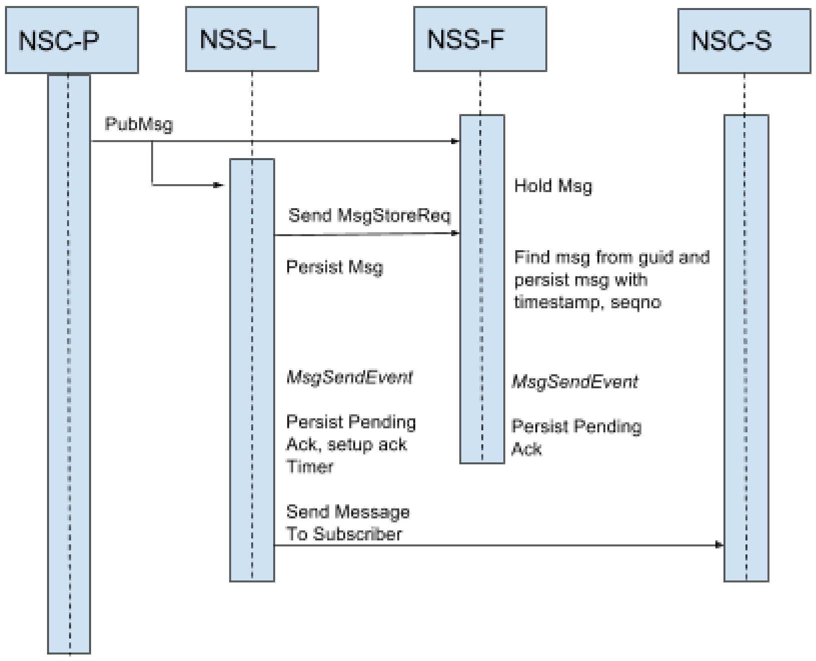
Next, if the subscriber subsystem is leader, for each subscriber, it sends the message to the client, creates an ack timer for resend, and writes a pending acknowledgement record. If subscriber subsystem is a follower, there is no need to send or create a timer - it simply writes the pending acknowledgement record.

We can run into an follower ordering issue where an acknowledgement can be received before a pending acknowledgement is created. If an ack is received for a pending acknowledgement we do not have, the store must assume that the message subsystem is out of sync and is, or will be, catching up. Since we should recognize the ack is valid, the pending acknowledgement record **and** and an acknowledgement record is written to maintain consistency. When compaction occurs, the stores should become truly consistent.

Note: This makes a few assumptions. Will this work?

Pending Acknowledgement Generation

This diagram illustrates how a published message generates a pending acknowledgement in the leader and follower.



NSC-P
NSS-L
NSS-F
NSC-S
PubMsg
Hold Msg
Send MsgStoreReq
Persist Msg
Find msg from guid and persist msg with timestamp, seqno
MsgSendEvent
MsgSendEvent
Persist Pending Ack, setup ack Timer
Persist Pending Ack
Send Message To Subscriber

Here the message subsystem and subscriber subsystem are both acting as the leader; this does not need to be the case. Just as easily, a follower for messages could be a the leader for subscriber responsibilities.

“Catch-Up” Protocols

At various points, a follower can fall behind the leader and needs to catch up. This may be a new follower starting, a server that has been offline for a period of time, or a server that has detected it is out of sync with the active leader and needs to return to steady state. The general flow is that a follower begins to receives and queue to live data, then requests historical data from the leader, and the leader give the follower what is necessary to become consistent. Once the follower has caught up, it reaches steady state and is an active quorum member. Error conditions, and merging data from two sources (as in the message and subscriber acknowledgement subsystems) will be detailed below.

There are a number of states a follower can be in:

State	**Permitted Live Data Operations**
Messages catching up	No other operations can occur.
Client State catching up	No other operations can occur.
Subscriber catching up	Messages and Client state steady state updates are permitted. *TODO: What about closing a connection?*
Steady State (all)	All subsystems can update their stores.

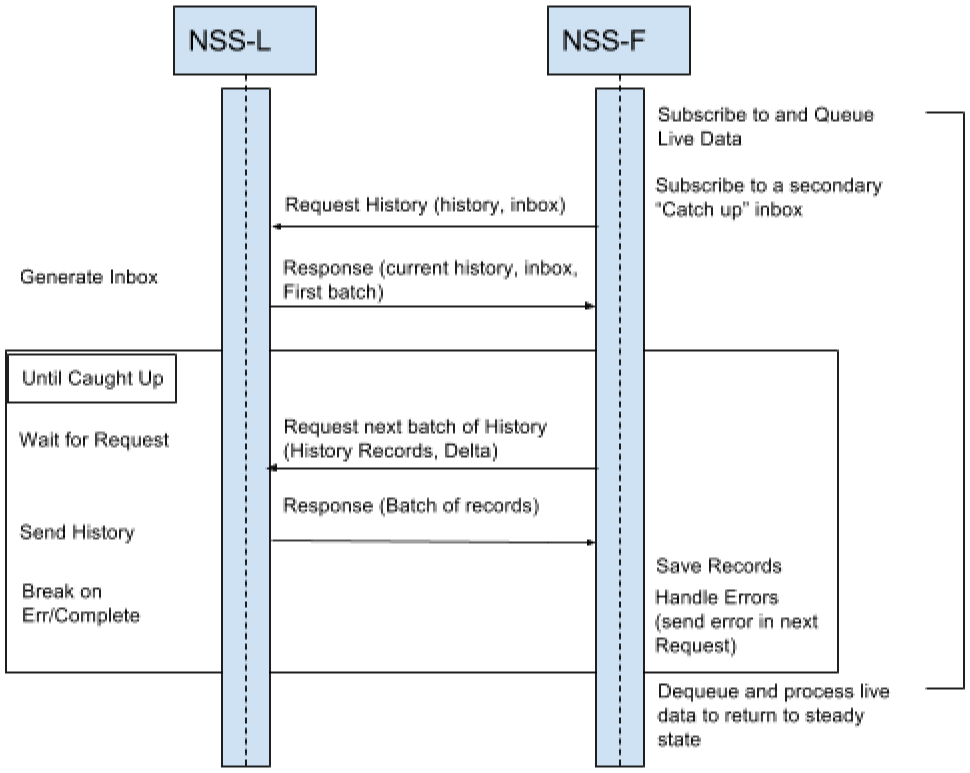
Subsystem Dependencies

Subsystems have dependencies on each other. The message subsystem is stand-alone, but the subscriber subsystem depends on both the message and client subsystems. So, anytime a server joins a cluster as a follower, it will invoke the following catchup protocols in the following order:

1) Client Catch-up protocol
2) Messages Catch-up protocol
3) Subscriber Catch up protocol

General “Catch-up” Protocol Detail

While all of the subsystem types will have their specifics, they share much in common in the way the achieve a consistent state with the leader.


NSS-L
NSS-F
Subscribe to and Queue Live Data
Subscribe to a secondary "Catch up" inbox
Request History (history, inbox)
Generate Inbox
Response (current history, inbox, First batch)
Until Caught Up
Wait for Request
Request next batch of History (History Records, Delta)
Response (Batch of records)
Send History
Save Records
Break on Err/Complete
Handle Errors (send error in next Request)
Dequeue and process live data to return to steady state

1. The follower is in a state where it has become inconsistent and needs to catch up. This is discovered through detection of lost message in steady state synchronization, or upon joining a quorum and by comparing history exchanged during leadership election and detecting it is behind the leader.
2. The follower begins to receive live data, if not already. This is data from all sources - the leader and clients. The live data is queued in a size bound buffer, where older entries are dropped. This prevents unbounded memory growth if there is an extremely large amount of data to be replicated.
3. The follower makes a request to the leader to send history for the given subsystem, starting from the latest history known by the follower. The leader responds with any information the follower needs to start receiving data (Inbox, current history)
    - If the current history is the same as the follower’s history, it immediately processes live queued data and enters steady state.
4. The leader begins sending history, and the follower updates its store.

5. The follower continues requesting and receiving data until the follower finds it is in steady state and sends a message to the leader. Note, this can create an overlap with the live data the follower has received. The follower's live data queue must be large enough to allow it to catch up - the size of a batch of records from the leader is a safe value.
6. When the follower recognizes it is receiving data from the leader that duplicates the live data it has collected, it discards data from the leader and begins dequeuing the live data. This is the transition from catching up to steady state. At this point, it can will send a message to the leader that it is caught up.
7. The leader stops sending data to the follower.

A variation…

The follower begins queuing live data later. History requested is sent. After the initial set of data requested by the leader has been sent, then the follower starts queuing live data. The leader continues sending "history", until live data and history overlap. The drawback here is that the follower may remain in a "catching up" state longer than it needs to be, especially if there is a lull in new data.

Catch-Up Protocol Example

1) The follower recognizes that it is out of sync with the leader because history does not match. Leader has record #4, follower record #2.

Leader Store	Follower Store	Follower Live Data Queue
1	1	
2	2	
3		
4		

2) Next, the follower begins collecting live data, and the leader starts sending its history to the follower upon request (it will also start sending live updates). At this point, record #3 has been replicated, and a new record #5 has been persisted at the leader since replication began. It will be sent to the follower as steady state update.

Leader Store	Follower Store	Follower Live Data Queue
1	1	
2	2	
3	3	
4		
5		

3) The follower continues to collect live data as it catches up. At this point, record #4 has been saved to the follower store through the catch-up protocol, and record #5 has been sent by the steady state mechanism and queued in the live data queue. The follower store has caught up to the point toward where replication began.

Leader Store	Follower Store	Follower Live Data Queue
1	1	5
2	2	
3	3	
4	4	
5		
6		

4) The leader continues sending records from its store, which now contain #5 and #6. The follower detects that record #5 is found in its live data queue. At this point, records #5 and #6 have been sent by the steady state mechanism, while the follower store has caught up. The follower can notify the leader through a response that it is caught up.

Leader Store	Follower Store	Follower Live Data Queue
1	1	5 (being persisted to the store)
2	2	6
3	3	
4	4	
5	5 (since this is the beginning of the live data queue, move to steady state)	
6		
7		

5) Now the follower moves to live data, while data is appended to the live data queue. The leader may have stopped sending catch-up protocol messages. Record #6 has been persisted, and are ready to persist #7, entering steady state.

Leader Store	Follower Store	Follower Live Data Queue
1	1	7
2	2	
3	3	
4	4	
5	5	
6	6	
7		

6) At this point, the follower is caught up, and in steady state. When records arrive from the leader through the steady state protocol and are places in the follower live data queue, they are immediately processed.

Leader Store	Follower Store	Follower Live Data Queue
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	

Detecting and Handling Slow Following Servers - TBD

Some followers will simply not be able to perform to required levels and need to be removed from the cluster. Reasons may be insufficient disk performance, network issues, swapping, or a plethora of other resource issues. Pruning these poorly performing members will prevent the issue of frequent cycling of catch-up to steady state, which will improve the overall performance and health of the cluster.

To this end, in all catch up protocols, the follower requests history from the leader and waits for the response (which may be a message to signal it's complete). Along with history, the follower can calculate its delta from the latest live sequence number and where catchup began. This serves two purposes - it is a form of flow control to meter the outflux of data to the follower as to not overwhelm it and allows the follower to determine its own fitness with respect to cluster participation through measuring the amount of outstanding data.

Upon receiving the request, if the follower finds the delta is continuing to increase over time, it determines it cannot catch up, sends an "cannot complete" message to the leader logs a message, and self terminates having determined it does not have adequate performance characteristics to successfully participate in the cluster. The leader stops sending data, and cleans up any resources associated with the follower.

Discussion: What about the case where followers are catching up during a burst of data? That may perhaps invalidate this idea during catch-up. In steady state, a follower can exit if it detects it is falling too far behind (by measuring the incoming queue of requests to store data by the leader). Is this the best way to handle slow followers?

This may be a feature to add later.

Alternative

An alternative is to simply let followers fail to catch up… This becomes a maintenance issue for users. Also, this could result in cycling of catch up protocols as a follower falls behind, fails to respond to heartbeats, etc, and begins to catch up again.

Message Catch-up Protocol

TODO: Eval snapshot for new servers. Could chunk and copy entire file, reducing metadata here.

There are three types of messages used in the message catch up protocol:

- **MsgSyncRequest**
    - Sequence number - the earliest point in history the follower requires
    - Return Inbox
- **StoreRequest -** this the store request described in steady state. These are queued up during catch-up state to facilitate a seamless transition into steady state.
- **MsgSync -** a message sync message containing everything required to persist a message from the leader's history.
    - Channel
    - Array of messages
        - Sequence
        - Timestamp
        - Payload
        - Optional End of Data Flag

1. The Follower begins subscribing and holding messages in a bound queue received by client publishes. It begins receiving and queueing **StoreRequest** messages from the leader. This is to later make a seamless switch to steady state.
2. The follower requests history from leader in a **MsgSyncRequest**
    a. Request to provide known history from a sequence number through an update request to the leader
    b. If the follower is up to date, the leader sends an **MsgSync** containing no messages, and follower confirms the latest sequence to become steady state.
3. The leader sends historical data to the follower as a **MsgSync** message upon follower requests. The follower persists the data returned from the **MsgSync** protocol message.
    a. If the follower detects it has missed a message sequence number, it restarts the catch-up protocol at the missing index.
    b. Multiple messages can be encapsulated in a **MsgSync** message for performance.
    c. The last **MsgSync** will have a completed status.
    d. The leader periodically checks the follower for inadequate performance (see Slow Following Servers)
    e. When the leader has sent its most recent record, it replies with a completed status.
4. The follower continues requesting data and upon identifying it is receiving overlapping data, the follower assumes steady state, dequeues any queued **StoreRequests** and processes them as live data, finally catching up.

   a. **StoreRequests** with a sequence # lower than the last historical data are discarded. This is overlapping data - messages published since the follower began the catch up protocol.

Client State Catch-up Protocol

The transition of state from catching up to steady state is similar to the other catch up protocols.

1. The follower requests an update from leader for history it does not have
   - Provide last known history in an update request to the leader
   - If the store has been compacted, a complete snapshot needs to be provided and the follower replaces its entire store.
2. The leader sends requested historical data to the follower as requested.
   - The follower check for inadequate performance (see Detecting Slow Following Servers)
   - Each data record has an sequence number. If the follower detects it has missed a record, it restarts the catch-up protocol at the missing index.
   - When the follower identifies it has caught up, it sets a completion status field in the response to the leader.
   - The leader cannot compact during this time.
3. The follower achieves steady state.

Subscriber Catch-up Protocol

- **SubscriberSyncRequest** This is request from the follower to initiate catch up from the leader.
    - Sequence number - the earliest point in history the follower requires
    - Channel Name - *may not be required, depending on the subject namespace.*
- **SubStoreRequest** - this the store request described in steady state. These are queued up during catch-up state to facilitate a seamless transition into steady state, and will have all data necessary to create the record as required. These records will correspond to the create,close, unsubscribe, send message, ack events. *If you are correlating this to existing code: subRecNew, subRecUpdate, subRecDel, subRecAck, subRecMsg.*
    - Sequence number
    - Action
    - Type (*SubState*, *SubStateUpdate*, *SubStateDelete*, *subRecAck*, *SubRecMsg*)
    - Marshalled buffer containing protobuf of type

1. Ensure client state is updated.
2. Existing subscriptions are created if not already, and the subscriber begins listening for live data.
3. Send a **SubscriberSyncRequest** to catch up on subscriber history.
    a. If the subscriber is up to date, nothing is to be done.
4. If synchronization is required, the leader compacts, then begins sending historical data. Compaction will optimize the follower allowing it to avoid creating and deleting subscriptions. The follower rebuilds its store and begins to recreate subscribers. It begins listening for acknowledgements to later transition to steady state.

Client State Catch-up Protocol

- **ClientSyncRequest** This is request from the follower to initiate catch up from the leader.
    - Sequence number - the earliest point in history the follower requires
- **ClientStoreRequest** - this the client store request described in steady state. These are queued up during catch-up state to facilitate a seamless transition into steady state, and will have all data necessary to create the record as required. These records will correspond to the create and close of a client. *If you are correlating this to existing code: see addClient and delClient.* When retrieving history from the leader, there should be no *delClient* records included (see Compaction), but receiving one should not be considered an error condition as to keep logic between catch-up and steady state similar.
    - Sequence number
    - Action
    - Type (addClientRec, delClientRec)
    - Marshalled buffer containing protobuf of type

1. Existing subscriptions are created, and the subscriber begins listening for live data.
2. Send a **ClientSyncRequest** to catch up on subscriber history.
    a. If the client state is up to date, nothing is to be done.
3. If synchronization is required then the leader begins sending historical data. Compaction at the leader will optimize the follower. The follower rebuilds its store and begins to recreate clients. It begins listening for updates as to later transition to steady state, and when complete, transitions.

Compaction

Compaction is the consolidation and rewrite of a store to eliminate unnecessary records, thus freeing disk space. Today, two stores are implemented to support compaction - the clients store (client state), and the subscription store. These stores have record types that indicate the creation of an object and deletion of an object, and are logged sequentially as these events occur. Periodically compaction occurs removing records that no longer relevant. Naturally, this will affect the history of a store. We need to account for compaction - it is essentially a complete reorganization and pruning of records in the client and subscription subsystems.

Synchronizing history can be achieved by ensuring existing store history that known to be consistent is identical. During steady state, compaction occurs as it normally would, but when a follower has determined it is out of sync, it performs compaction before initiating the catch up protocol. Upon receiving a catch-up protocol request for a store that supports compacting, the leader compacts before synchronizing records with the follower. This should maintain consistency.

Discarded ideas

- Compaction accounts for history. Historical data concerning indexes remains the same, but gaps are tolerated or accounted for. This could be an additional field in each record that would contain the next expected historical sequence. ***This adds too much complexity in compaction.***
- Disable compaction entirely. This is the simple approach but will result in ***out of control growth of store files.***
- Coordinate compaction between the leader and followers through a control message. The resulting file is checksummed and compared with the leader to determine if compaction was successful. ***This solution adds too much chattiness to the protocol, and may disrupt steady state processing more than necessary.***

Discussion and Meeting Notes

This section of the document is to maintain a history of discussion notes. As they are implemented or rejected, they'll be marked as so, with a reason if applicable. Maintaining a history of rejected ideas help prevent rehashing.

3/27/2017 - Ivan/Colin

- **DONE: (updated document):** PubMsg - leader sends before trying to write. Any write successes with an overall quorum failure remain, and become part of history. Leader fails a write, waits for responses, then panics, a follower takes over.
- **DONE: (updated document)**: Client-State - Followers process requests as in msgpubs, responds back to client with guid and subid follower fails on inconsistency and initiates catch-up. Subs have a guid added by the client, which allow the leader to aggregate responses to the follower. Primay sends [guid, id, ackinbox], Pre-Work for retry API.
- **DONE: (updated document):** Move Sub client states to suback raft group.
- **Posed for discussion:** Substates recover store (for ackPending) on assumption of leader role
- TODO: Describe subject namespace

5/2/2017 - Ivan/Colin

Agenda/Discussion points
RAFT modifications

- Carefully walked through scenarios (leadership election, split brain) **(Looked good so far**)
- Discussed adding history to RAFT election, can be as simple as a callback that is set to be invoked to provide a comparison after term checks during leadership election.
- Should the RequestVote response have the candidates inboxes for catch up once they are followers - Leader can initiate? **(Did not discuss).**
    - Pros: Less chatty at startup
    - Cons: slightly different protocol between filling in a gap and catching up

Catch up

- Spot holes with custom catch up and RAFT. What is missing?
    - Walkthrough leadership change during catch up. Can/should we optimize? **(Looked good so far**)
    - Walk through transition from catchup to steady state, in the context of RAFT - are there holes? (**Looked good so far)**

Discussion:

- During catch-up, the follower will self terminate if it cannot catch-up; less work for the leader.
- The leader has a common catchup subject. Then, if there is a leadership change during catchup, any follower catching up will time out on a request for data, and will retry with the new leader, continuing where it left off. Follower will request a start index, and a batch size.
- Follower will restart the entire channel if messages have been rolled off (If leaders earliest > followers max). Otherwise, gaps could exist.

Meeting Notes (5/23/2017)

- Publisher
    - Publisher leader should error, but not step down on a missing client message - just send an error to the client. Don't give up election on small gaps.
        - Flesh this out...
    - Client perspective - Timeout from server = maybe delivered, error - absolutely will not be delivered.
    - Research/Prototype - Publisher leader exchanges metadata, followers use metadata to participate, while catching up. Heartbeats can have a map of messages synced/not-synced. Attempt RAFT for metadata, repair mechanism for payloads.
    - Add a **repair section**, different raft group? Repair gaps in publisher data

        Subscribers
        - Split up work - RAFT Group for each sub…
    - Repair (Publish side)
        - Pubs can participate as pub leaders so long as they have metadata
        - Repair mechanism (group) fills in any gaps based on heartbeats from pubs with map of data.
    - RAFT - Create our own, sync, groundwork for lazy writes, etc. Make our own w/ leadership election, etc. repair… Map with Sequence low, high, gaps.
    - Monitoring in RAFT.

Research Areas

Here are some areas to do some in-depth research and flush out issues.

GRAFT

Identify required changes:

- lastLogIndex functionality needs to be implemented. This should be a binary field containing a marshalled protobuf. Do we need last log term? (I don't think so, but need to dig deeper)
- Rather than using a log file, provide an interface to a log implementation that allows for exchanging log information in the vote request, as a binary field. We'll use a protobuf, as we need more than just an index for the subscriber subsystem. The log interface will also need to have a history comparison function to be used in leadership election that will use this log comparison method.
- Override parameters, namely heartbeat interval. I think we can figure out a way to live with a very slow heartbeat (e.g. 5s) for streaming after election, so long as we have a way to trigger re-election. This can be a message from the leader to the followers indicating it is stepping down and followers convert to candidates to then elect a new leader.

Catch-up Protocol

Create a package for the catch-up protocol, allowing for encapsulating existing protobufs. The idea is to create a package that:

- Initiates (requests) Catchup
- Responds to a Catchup Request
- Transfer Data
- Error notification

Storage Optimization

Alberto had a great idea: Split each channel message store into two, one with metadata(seq, timestamp, and guid), and another with guid and payload. This allows the follower nodes to immediately write ½ the record (payload), then later write the first half (metadata), rather than hold messages in memory. However this could leave gaps where the “payload” store has records that have no associated metadata after errors, etc. When reading the store, first read metadata and quickly compare the guids. If there isn’t a match keep reading the payload store until there is a match…   This reduces memory, could allow parallel writes, but would result in another file descriptor for each channel.

Repair

- This is a raft group? that repairs data from publisher. Assuming publisher can have gaps. All publish gap records, indicating they have a gap. Upon receiving a record (could be in an array of gaps), the leader of this group responds with the empty records.
    - Separate GAPS file, can be compressed.
    - GAP records
        - Index
        - Timestamp
        - Other meta data
    - GAP response record
        - Index
        - Payload