NATS

Seminar Objectives

- Introduce NATS as a technology to connect services
- Explain main NATS concepts
- Use examples of usage
- Gain practical skills with NATS

https://docs.nats.io

Contents

- Core NATS
  - Pub Sub
  - Services
  - Load Balancing
- Jetstream
  - KV Store
  - Object Store
- Advanced configurations

NATS: Setup for seminar

- Use docker to run the server locally
  - docker run –-name nats -it –p 4222:4222 nats --js
- Add a “-d” flag if run in the background
- See: https://github.com/nats-io/natscli/releases

01

Core NATS

Ephemeral Messaging

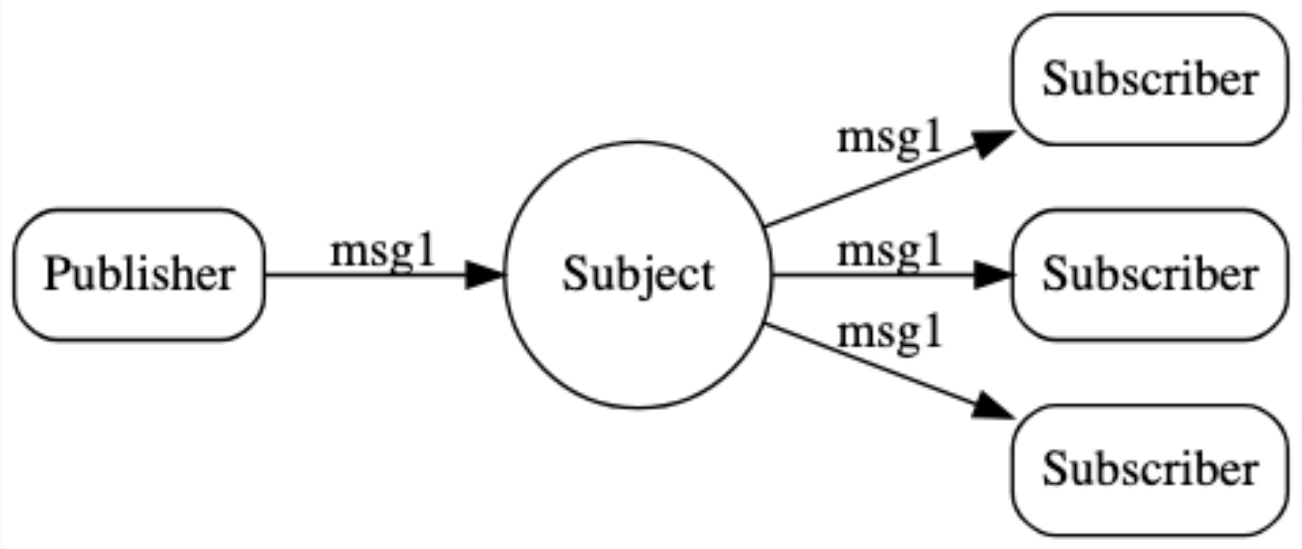
Core NATS

- PUB/SUB model: At most once semantics
- Producers publish messages to subjects
- Consumers can subscribe to subjects
- A consumer receives a message if it is subscribed when the message is published
  - Volatile, but ordered
- Addressing based on subject (topic)
- Clustering/Fault tolerance/Scalability techniques

Several patterns

- PUB/SUB
- Request-Reply
- Load Balancing: Queue Groups

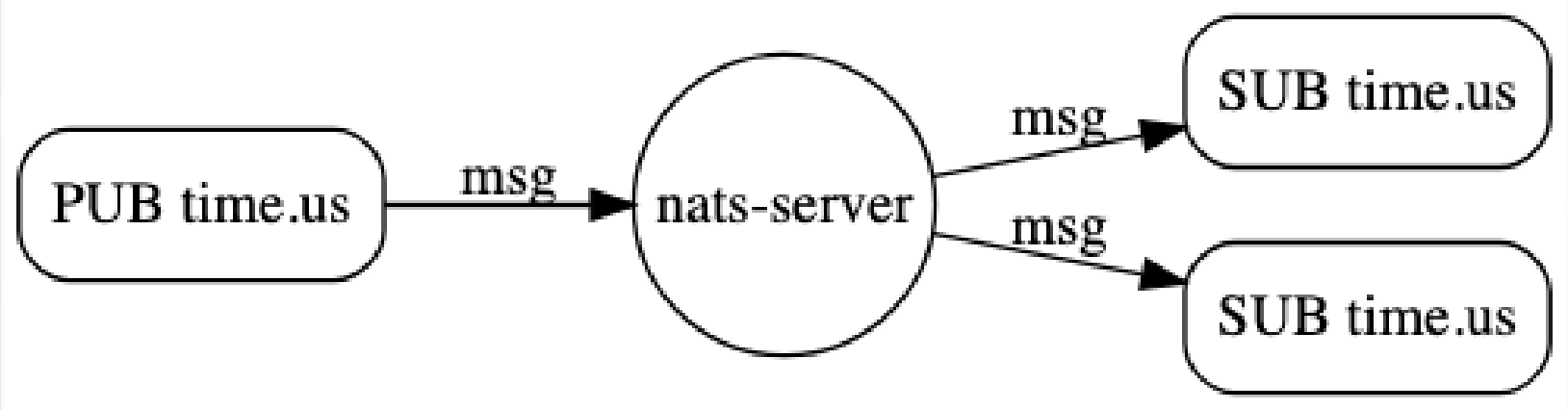
Basic PUB/SUB



PUB/SUB: Messages

- **A subject**
- **A byte array as Payload**
- **Any number of header fields**
- **An optional *reply*** address field
- **Maximum size can be configured**
  - **1 MB default**
  - **64 MB limit**
  - **8 MB recommended**

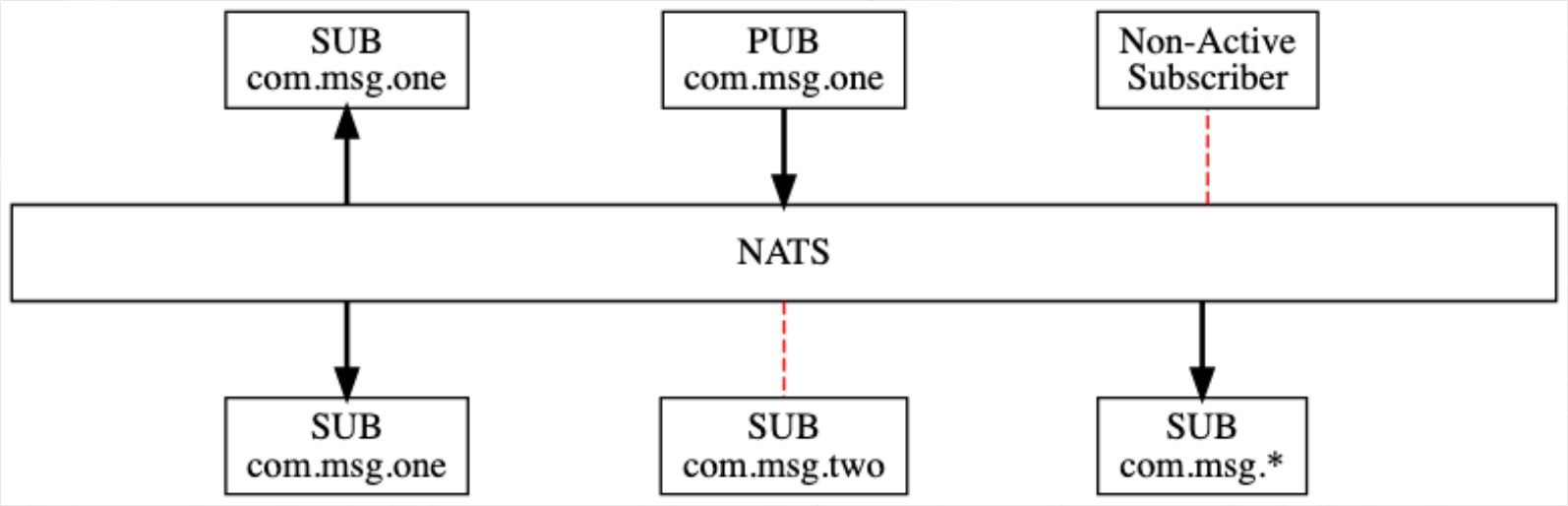
PUB/SUB: subject addressing



PUB/SUB: Subject names

- Tokenized, separated by “.”
- Token allowed characters
- Any Unicode, except “.”, “*”, and “>”
- Reserved names start with “$”
  - $SYS
  - $JS
  - $KV
  - …

Exercise


SUB
com.msg.one
PUB
com.msg.one
Non-Active
Subscriber
NATS
SUB
com.msg.one
SUB
com.msg.two
SUB
com.msg.*

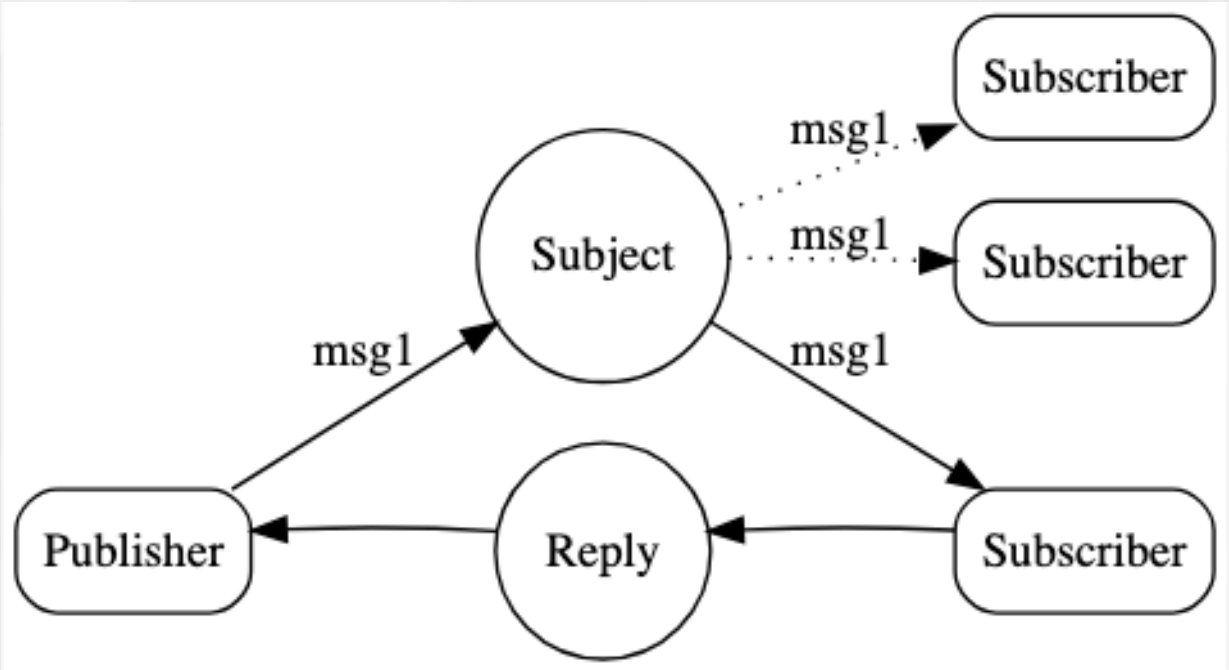
Exercise

- Create subscriber 1 (on its own terminal)
  - nats sub com.msg.one
- Create a publisher and publish message
  - nats pub com.msg.one “This is a simple message”
  - nats pub com.msg.one “This is another simple message”
- More subscribers (use different terminals)
  - nats sub com.msg.one
  - nats sub com.msg.one

Exercise: subject wildcards

- Create simple wildcard subscriber
  - nats sub com.msg.*
- Create a publisher and publish message
  - nats pub com.msg.one “Hi”
  - nats pub com.msg.one.two “Do you see me?”
- Single-token wildcard (many)
  - nats sub com.msg.*.two
  - nats pub com.msg.one.two “Do you see me now?”
- Multi-token wildcard (only at the end)
  - nats sub com.msg.>

Request/Reply



Request/Reply

- Built on top of basic PUB/SUB
- Requests sent on a request subject
  - Reply subject dynamically created: _INBOX.<random>
  - Requester subscribes to reply subject
- Requests sent on a request subject
  - Replier subscribes to request subject
  - Replier publishes to dynamic reply subject
    - Reply subject is found in headers

Request/Reply

- Can have multiple repliers
  - Only the first arriving reply is considered
- Repliers can be grouped
  - Only one in the group receives each request
  - Can scale up/down
    - Repliers must drain messages to avoid request loss on downscale

Exercise

- Create replier (on its own terminal)
  - nats reply com.request.hello world
- Perform a request
  - nats request com.request.hello world
- Subscribe to all (in its own terminal)
  - nats sub ”>”
- Look at the traffic generated
  - nats request com.request.hello world

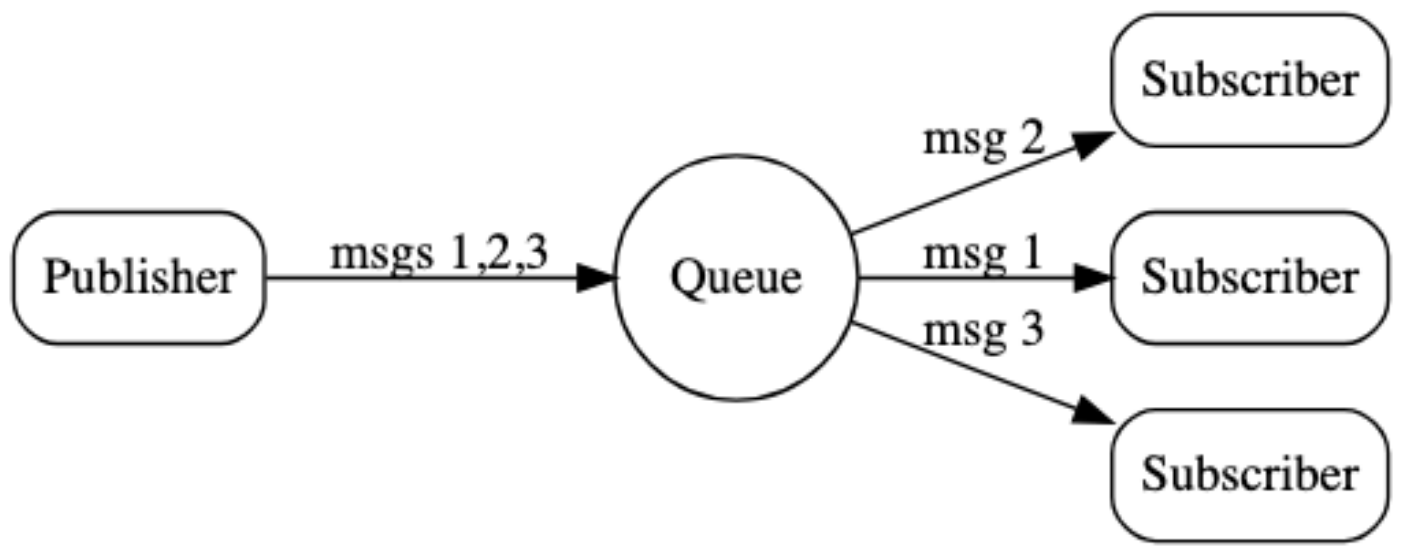
Exercise

- Create replier with command
  - nats reply ‘hello.*’ —command “echo Hello {{1}}”
- Perform a request
  - nats request hello.peter ‘’
- Inspect body
  - nats reply ‘hello.*’ –command “echo {{.Request}} {{1}}”
- Perform another request
  - nats request hello.Don Hello

Queue Groups

- **Subscribers subscribe as part of a named “queue”**
  - ***Queue Group*** = all subscribers part of a queue
  - Names follow same rules as subject names
- **Only one subscriber in the group gets a given message**
  - Randomly selected
- **All queue groups get all messages to the subject**
  - Load balancing within each group
- **Also applies to repliers**

Queue group


Publisher
msgs 1,2,3
Queue
msg 2
msg 1
msg 3
Subscriber
Subscriber
Subscriber

Exercise

- Create two subscribers (each on its own terminal)
  - nats sub –queue test hello
  - nats sub –queue test hello
- Create a subscriber on a different group
  - nats sub –queue notest hello
- Send several requests
  - nats pub –count hello “Hi {{.Count}}”

Exercise

- Create two repliers (each on its own terminal)
  - nats reply –queue test hello –command “echo 1 {{.Request}}”
  - nats reply –queue test hello –command “echo 2 {{.Request}}”
- Create a replier on a different group
  - nats reply –queue notest hello –command “echo 0 {{.Request}}”
- Send several requests
  - nats request –count hello “Hi {{.Count}}”

01

Jetstream

Persistence in NATS

Overview

- NATS persistence engine
  - Messages can be stored and replayed at a later time
  - Late arriving consumers can still receive messages
- In a cluster of NATS servers, data can be replicated
  - Helps handle failures
- Key Value store can be built on top
  - At client library level (no server changes)
- Object Store can be built on top
  - At client library level (no server changes)

Streams

- Stream == Named and ordered message store
  - Consumes normal NATS Subjects
  - Defines means of storage
    - In memory
    - In File
    - Degree of replication (1 .. 5) among servers
      - By means of Raft consistency protocol
  - Imposes limits
    - Retention policy
    - Discard Policy (old out or new blocked)
  - Message de-duplication
    - Within a sliding window

Details: https://docs.nats.io/nats-concepts/jetstream/streams#configuration

Retention Policies

- Limits Policy
  - MaxBytes, MaxAge, MaxMsgsPerSubject
    - When hit, automatic deletion of message
- Work Queue Policy
    - Each message can be consumed only once
      - ➔ Only one consumer per subject
      - Once acked, the message is deleted
- Interest Policy
    - If no consumers exist, published messages are deleted
    - Once a message is acked by all consumers, it is deleted

Consumer

- Stateful view of a stream
  - Lives in server
  - Interface for clients to consume
    - Keeps track of which messages are ACKed by clients
      - Responsible of tracking delivery and acks
      - Automatic redeliver attempt if not delivered
      - Various ack types and policies
- Dispatch types
- Persistence

Dispatch type: Push/Pull

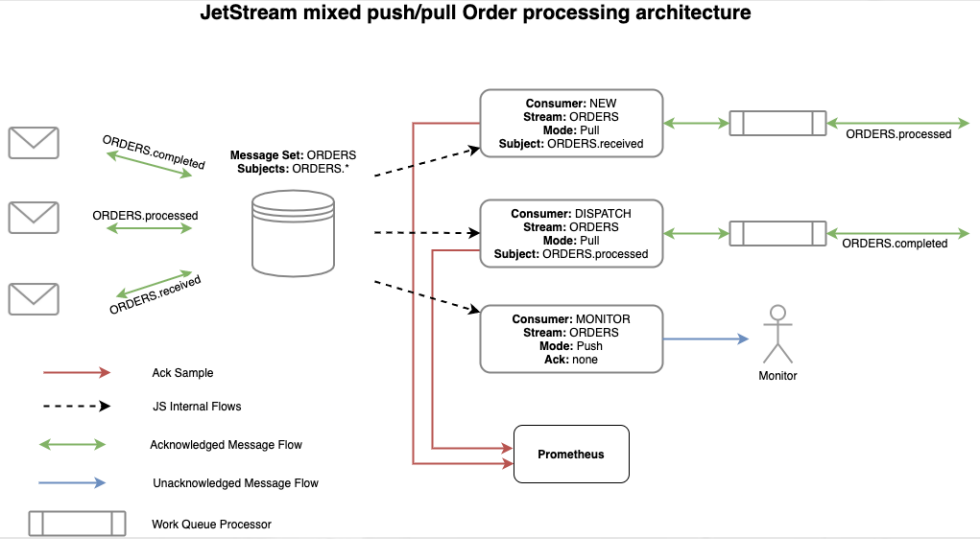
- Push - based
  - Consumer delivers message immediately
  - Use case:
    - Simple app that need to access all messages in order
- Pull - based
  - Client requests messages from consumer
    - Potentially in batches
  - Use case:
    - Application controls, and can scale
    - Also above use case

Persistence: ephemeral/durable

- Ephemeral
  - Server memory only: not persisted.
    - Automatically deleted after inactivity (no subscribers)
- Durable
  - Same replication factor as their stream
    - Can recover from server/client failures
  - Can be cleaned up if inactive
    - If ***InactiveThreshold*** is set
  - When ***Durable*** field is set
    - Or ***InactiveThreshold*** is set

Details: https://docs.nats.io/nats-concepts/jetstream/consumers#configuration

Example




```
nats stream add ORDERS --subjects "ORDERS.*" --ack --max-msgs=-1 --max-bytes=-1 --max-age=1y --storage file --retention limits --max-msg-size=-1 --discard=old
nats consumer add ORDERS NEW --filter ORDERS.received --ack explicit --pull --deliver all --max-deliver=-1 --sample 100
nats consumer add ORDERS DISPATCH --filter ORDERS.processed --ack explicit --pull --deliver all --max-deliver=-1 --sample 100
nats consumer add ORDERS MONITOR --filter '' --ack none --target monitor.ORDERS --deliver last --replay instant
```

Example

- **Create a stream**
  - nats stream add sogreat
    - **Set subjects fantastic, horrible.>**
- **Publish to the subjects of the stream**
  - nats pub fantastic thatis
  - nats pub horrible.my.gosh thatis
- **Verify info**
  - nats info sogreat
    - **We can see there are two messages in stream**
- **Normal subscribers cannot access the stream**
  - **Only new messages**

Example

- **Create a consumer**
  - nats consumer add
    - **Set the name: *marvel***
    - **Accept the defaults**
- **Subscribe**
  - nats consumer next sogreat marvel –count 100
    - **We can see we are retrieving the messages we sent**
  - nats consumer next sogreat marvel –count 100
    - **No new messages appear**

Example

- **We can replay: creating another consumer**
  - nats consumer add …
  - **Or removing old consumer and recreating it**
    - nats consumer rm marvel
    - nats consumer add …
- **Clean up**
  - nats stream purge sogreat
  - nats stream rm sogreat

Key Value Store

- Built on top of Jetstream
- A KV bucket corresponds to a stream
  - Builds an immediately consistent map
- Operations on a bucket
  - ***put*** associates a value with a key
  - ***get*** retrieves the value associated with a key
  - ***delete*** *remove any value associated with* a key
  - ***purge*** *remove all values associated with* all keys
  - ***keys*** *get all the keys with the operations associated*

KV Store: concurrency control

- Operations that verify a condition and mutate atomically
  - *create* associates a value with a key
    - Only if the key does not exist
  - *update* compare and set the value for a key
    - Fails if the revision provided is not the current one for the key

KV Store: stream operations

- Getting streams
  - *watch* receives changes for a key (with wildcards)
    - Similar to a subscription
  - *watch all* receive changes for all the keys in the bucket
  - *history* list of the values and deletes for each key over time
    - By default, history of buckets set to 1.
      - Only the latest value/operation is stored

Example

- **Creating a KV Bucket**
  - nats kv add almacen
- **Setting a value**
  - nats kv create almacen pala.troca grande
  - nats kv create almacen pala.troca grande
- **Delete a key**
  - nats kv del almacen pala.troca
- **Watch a store or key**
  - nats kv watch almacen
  - nats kv watch almacen “pala.>”

Object Store

- Built on top of Jetstream
- An Object bucket corresponds to a stream
  - Builds a set of files or arbitrary size
    - Stored as a collection of chunks
  - All files in a bucket must fit in a file system
    - Not a distributed storage system
- Operations on a bucket
  - ***put*** adds a file to a bucket
  - ***get*** retrieves a file and stores in a designated location
  - ***del*** removes a file
  - ***watch*** informs of changes in a bucket

Example

- **Creating an Object Bucket**
  - nats object add pueblo
- **Storing a file**
  - nats object put pueblo archie.zip
  - nats object put –name archivo.zip pueblo archie.zip
- **Retrieving a file**
  - nats object get pueblo archivo.zip
  - nats object get –output archie.zip pueblo archivo.zip

Q & A