NATS AT THE EDGE

The power of Leaf Nodes in edge architectures

Samuel Attwood (Senior Software Engineer) & Borja Etxebarria (SE Director)



NATS AT THE EDGE

Agenda.

- ⇉ Distributed edge architectures
- ⇉ Leaf nodes, do’s and don'ts
- ⇉ Hands-on lab
- ⇉ Q&A



Hands-on Requirements

https://github.com/synadia-labs/2026-RethinkConn-EdgeWorkshop

- **nats-server 2.14**
  - https://docs.nats.io/running-a-nats-service/introduction/installation
  - Mac OS: `brew install nats-server`
  - Arch Linux: `yay -S nats-server`
  - Windows: `scoop install main/nats-server`
- **nats-cli 0.4.0**
  - https://github.com/nats-io/natscli?tab=readme-ov-file#installation
- terminal (**`bash`** / **`zsh`** / ...)
- **`jq`** (https://jqlang.org)

The Edge

What makes it unique & challenging

Diverse Physical Footprint

Device, vehicle, robot, factory, warehouse, store...
Distributed in many locations
Decentralized operations with security challenges

Scale

Massive number of deployments & endpoints

The Cloud vs. Edge Gap

- Fragmented ecosystem & tools
- Extreme hardware diversity
- Resource constrained (CPU/RAM/storage/network)
- Intermittent & costly connectivity
- Requirement for local autonomy

Why NATS at the edge ?

The little server that could

Lightweight & Flexible

Small binary (**~15MB**).
No dependencies.
Deploys seamlessly from high-scale cloud to resource-constrained edge devices.

Advanced Topologies

Interconnected **fabric** supporting diverse architectures: Clusters, Superclusters, and Leaf Nodes for total connectivity.

Feature Rich

- Pub/Sub & Request/Reply
- Microservices & Streaming
- KV/Obj Store
- Built-in Multi-tenancy & Security
- MQTT

Proven Deployments

- Mission-critical telemetry collection
- **Command & Control** across extreme environments
- & more!

- Vehicles & Stores
- Factories & Robots
- Satellites & Airplanes

NATS Topologies

Cluster (routes)

```
cluster: {
  name: CLUSTER_A
  port: 6222
  routes: ["nats://a-1:6222"]
}
```

SuperCluster (gateways)

```
gateway: {
  name: "CLUSTER_A"
  port: 7222
  gateways: [
    {  name: "CLUSTER_A"
       urls: [ "nats://a-1:7222" ] }
    {  name: "CLUSTER_B"
       urls: [ "nats://b-1:7222" ] }
    {  name: "CLUSTER_C"
       urls: [ "nats://c-1:7222" ] }
}
```

LeafNodes (remotes)

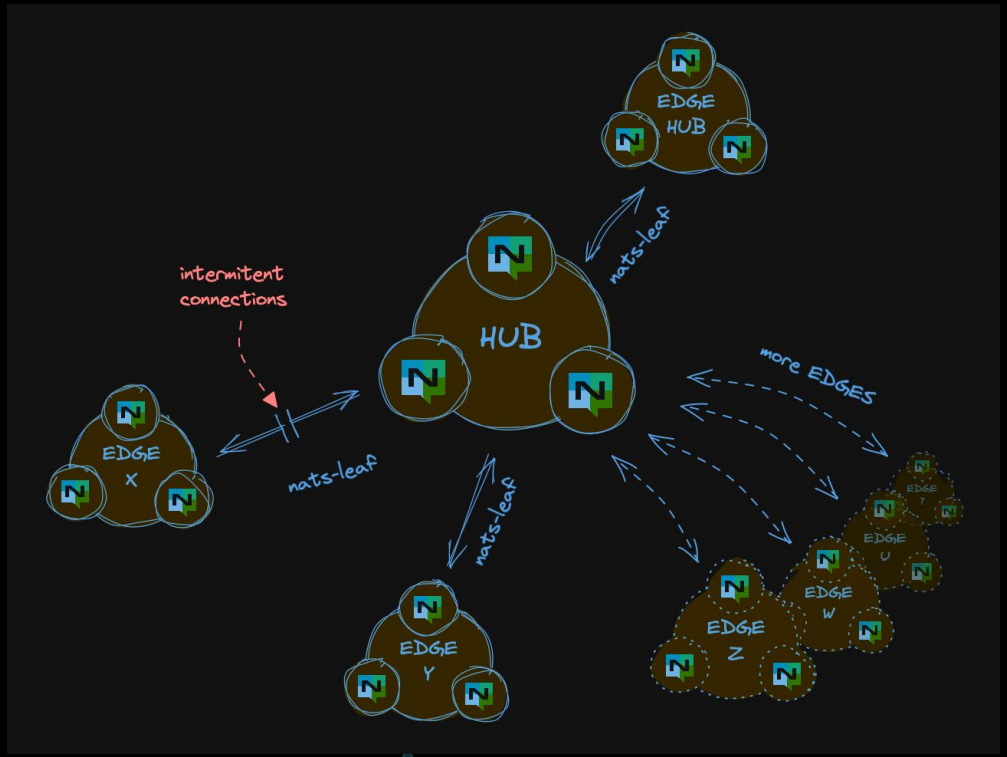
```
leafnodes {     ### hub
  port: 7422
  authorization {
    users = [
      { user: u1, password: p1, account: A1 }
...
```

```
leafnodes {     ### leaf
  remotes [
    { urls: ["nats://u1:p1@a-1:7422" ]
      account: L1
    }
    ...
```

Leaf Nodes

What makes them unique

- Security boundary
  - edge_account→to→hub_account
  - can use different security model
    - noauth, token, user/pwd, nkeys, jwt...
- Network boundary & topology
  - Detachable from HUB
  - Support daisy chains
  - Can fence Subjects & Interest propagation
  - Connect across firewalls
- JetStreams
  - Fencing RAFT
  - JS Domains



Terminology

Just in case...

- **Leaf node** → can refer to a leaf cluster

- Hub-Leaf traffic types:

Intra-cluster

Standard communication within the same local cluster environment.

North-South

hub-to-leaf & leaf-to-hub

Daisy-chained:

leaf-to-leaf-to-leaf... extending the reach of the hub connection.

East-West

leaf-to-hub-to-leaf

- Leafs not directly connected
- Brokered by the hub (south→north→south)

Cluster name

Same for all leafs or unique?

Unique names → correctness and better observability

Same name

Will **disable east-west** traffic.

Note: it is a workaround (mostly not needed anymore)

NATS 2.12+

Different cluster names → Disable east-west interest propagation using:

`isolate:true`

Subject Interest Propagation (I)

A relevant detour

`nats subscribe foo` ← triggers propagation of interest in `foo` to ***all nodes*** in the cluster

Protocol Mechanics

Check NATS Cluster Protocol

RS+ / RS- messages (**LS+ / LS-** for leafs)

Source: nats-server-protocol

Blocking interest != Permissions (blocking msgs.)

Interest Propagation Traffic

- Exchanged by NATS nodes
- Triggered on **`SUB/UNSUB/DISCONNECT`**
- Reverse direction to regular traffic
- Lower cardinality than payload traffic...
- *... except on Edge:*
  address one of many devices occasionally

Subject Interest Propagation (II)

A relevant detour

Network load correlates to:

- SUB/UNSUB/disconnects rates
- Subject cardinality (not message rate)
- Nodes in the clusters

- *Beware leafnode churn!*

Memory pressure (interest graph) correlates to:

- Subject cardinality, length, token structure

- *Affects all nodes equally (no sharding)*

Leaf Nodes can fence subjects & propagation! → Leverage to save resources

Related pattern → micro cell clusters for both overall load *and* interest sharding

Leaf interconnected [through a hub] with *tight control* on north-south subject sharing

Subject Muxing

One Subscription Per Handler:

- Simple pattern
- Works at small scale
- Easy to lose track of subscription growth
- Overhead exacerbated by leaf interest propagation
    - Each additional subscription is multiplied by the number of leaf nodes with interest

Muxing:

- One subscription prefix (`<id>.foo`)
- Still enables per-client subject addressing
- Enables same functionality with reduced interest propagation overhead

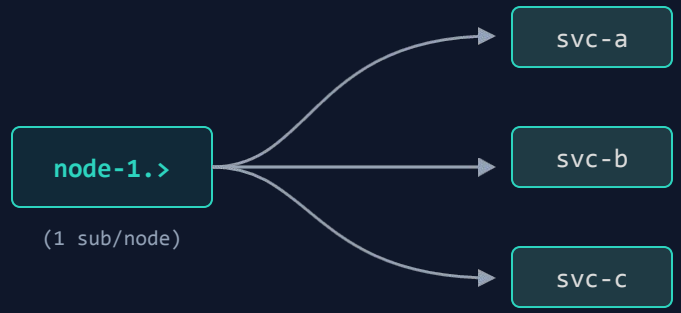
Subject Muxing Visualization

One Subscription Per Handler



Linear growth: Total of 3 unique subscriptions propagated across all cluster nodes (1 per handler).

Muxed (Wildcard) Pattern



Constant overhead: Only 1 unique subscription propagated per node, regardless of handler count.

Fencing subjects/interest on Leaf Nodes

Security + wisely save resources

Hub-Side Control (Option 1)

Permissions applied to the `leaf-user` on the Hub.

- Centralized governance
- `pub/sub allow/deny` rules
- Avoid unnecessary interest propagation bidirectionally

```
CONFIG EXAMPLE (leaf-user at Hub):
...
permissions: {
  publish: { deny: ["nopub.>"] },
  subscribe: { deny: ["nosub.>"] }
}
```



Leaf-Side Control (Option 2)

Fencing defined in the Leaf Node `remotes` block.

- Local autonomy for the edge
- `deny_exports`: blocks North-bound msgs
- `deny_imports`: blocks South-bound msgs
- `deny_exports` is less efficient
  (wasted interest prop. msgs from the hub)
- Requires NATS 2.12+

```
CONFIG EXAMPLE (Leaf remote config):
remotes: [{
  url: "nats-leaf://hub:7422",
  deny_exports: ["nopub.>"],
  deny_imports: ["nosub.>"],
...
```

SYSTEM account on Leaf Nodes

Should we leaf-connect it?

Benefits

- Centralized management & observability
- System Events: Audit events (core & JetStream)
- Remote server management & observability
- NATS Surveyor & Insights support

Ref: NATS Sys Accounts Docs

Considerations (Addressable)

- Security exposure (mitigated via fencing)
- Extra management traffic on the WAN

Note: System account is not strictly needed from a functional point of view.

Leaf Nodes allow subject fencing!

Not everything has to be shared. You can precisely control what system data propagates north-bound.

Fencing JetStreams RAFT with Leaf Nodes

Infrastructure & Assets

- Independent RAFT groups per leaf cluster
- Local JetStream assets (streams, consumers)
- Asset partitioning

Performance & WAN Optimization

- Support for detachable clusters & slow links
- Reduced cross-WAN consensus chatter
- Prevention of global quorum impact

Locality, Reliability & Scale

- **Local-first persistency:** Superior data locality & compliance vs. placement tags
- **Failure boundaries:** Isolate issues to local clusters without affecting the Hub
- **Next-level for scale-wide:** Foundations for distributed micro-cell architectures

JetStream Domains

Should we use them?

Service API Formats

Domain-less:

`$JS.API.STREAM.CREATE.FOO`

Domain-name:

`$JS.mydomain.API.STREAM.CREATE.FOO`

Architecture Requirements

Leaf-to-hub/hub-to-leaf stream API access (including mirror/sourcing) **requires domains**

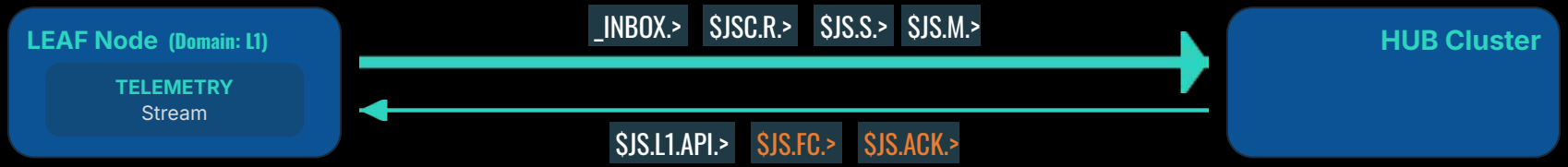
- **Routing:** API call sent to leaf/hub with that domain
- **Local:** Domain-less still works for local cluster

Best Practices

- General rule → always **use Domains**
- Unique Domain in **each leaf cluster**

JetStream Domain Interaction (hub → leaf)

JS API & Control Flow - Hub accessing streams in Leaf



- Check the JetStreams API reference
- $JS.FC.> and $JS.ACK.> are not Domain aware → ensure stream_name + consumer_name is unique!
  - NATS 2.14 supports v2 format, domain/account aware. It will be active in NATS 2.15
- _INBOX.> → or whatever inbox prefix used by the caller

JetStream Domain Interaction (leaf → hub)

JS API & Control Flow - Leaf accessing streams in Hub



- Check the JetStreams API reference
- Beware of _INBOX.> interest graph up to the hub (high cardinality devices in the leaf). Block accordingly

Other topics

Subscription creep

- mux subscriptions

Consumers & KV

(→ watchers are expensive)

- stream republish + direct get

Future

- Subject mux
- All-accounts leaf

PING-PONG / LDM / Reconnect

- `PingInterval(), MaxPingOut()`
- `lame_duck_duration`
- `ReconnectJitter()...`

Auth Architecture

- Decentralized **JWT** Authx
- **Auth Callout** on leaf nodes

NATS AT THE EDGE

Hands-on *Lab*



Thank you +

