NATS · JETSTREAM · GO

Building Apps on JetStream.

// best practices

Production patterns from **nats.go** and the **orbit.go** extensions.

01 · FOUNDATIONS

One connection, context-first

- Open **one** connection and share it — it multiplexes everything.
- Configure with a named **nats context**, not hard-coded URLs or creds.
- Every JetStream call takes a **context** — give it a timeout.

connect.go

```go
// One connection per process — reuse it.
nc, settings, err := natscontext.Connect(
    "production", nats.Name("orders-svc"))
if err != nil {
    log.Fatal(err)
}
defer nc.Close()

// Context-first API: always pass a deadline.
js, _ := jetstream.NewWithDomain(nc, settings.JSDomain)
ctx, cancel := context.WithTimeout(
    context.Background(), 10*time.Second)
defer cancel()
```

02 · DESIGN

Declare streams, idempotently

- Model your **subject space** first — **orders.>** scales to many event types.
- Set retention, limits and **Replicas: 3** up front, not later.
- **CreateOrUpdateStream** is declarative — your config is the source of truth.

stream.go

```go
// Declarative + idempotent: safe on every boot.
stream, err := js.CreateOrUpdateStream(ctx,
    jetstream.StreamConfig{
        Name:       "ORDERS",
        Subjects:   []string{"orders.>"},
        Retention:  jetstream.LimitsPolicy,
        Storage:    jetstream.FileStorage,
        Replicas:   3,
        MaxAge:     72 * time.Hour,
        Duplicates: 2 * time.Minute,
    })
```

02 · DESIGN · NACK

…or declare it in Git

- **nack** — the NATS controller for Kubernetes — reconciles JetStream from **CRDs**.
- The same **StreamConfig** fields, now under **spec:** — GitOps-friendly.
- The controller creates & updates the stream to match — declarative, like the Go call.

orders-stream.yaml

```yaml
apiVersion: jetstream.nats.io/v1beta2
kind: Stream
metadata:
  name: orders
spec:
  name: ORDERS
  subjects: ["orders.>"]
  retention: limits
  storage: file
  replicas: 3
  maxAge: h
  duplicateWindow: m
```

04 · CONSUME

Durable pull consumers

- Prefer **pull** consumers — flow control lives with the client.
- One **Durable**, many workers — they share the load automatically.
- Bound retries with **MaxDeliver** + **BackOff** so poison messages stop.

consumer.go

```go
cons, err := stream.CreateOrUpdateConsumer(ctx,
    jetstream.ConsumerConfig{
        Durable:       "order-workers",
        AckPolicy:     jetstream.AckExplicitPolicy,
        AckWait:       30 * time.Second,
        MaxDeliver:    5,
        BackOff: []time.Duration{
            time.Second, 5 * time.Second, 30 * time.Second,
        },
        FilterSubject: "orders.created",
    })
```

04 · CONSUME

Acknowledge with intent

- **Ack** on success, **Nak** to retry, **Term** to drop a poison message.
- Use **InProgress** to extend **AckWait** on slow work.
- **DoubleAck** waits for server confirmation — survive a client crash.

consume.go

```go
cc, err := cons.Consume(func(msg jetstream.Msg) {
    if err := handle(msg); err != nil {
        msg.NakWithDelay(5 * time.Second) // retry
        return
    }
    msg.Ack() // DoubleAck(ctx) for effectively-once
})
defer cc.Stop()
```

CONSUME · BATCHES

Pull in batches.

Fetch in bulk for throughput — but never out-fetch the connection.

`Fetch / FetchBytes` `PullMaxBytes` `64 MB / connection`

BATCHES · BOUND BY BYTES

Cap the batch, not just the count

- A pull buffers messages in the client; a huge **Fetch** can blow past the **64 MB** per-connection buffer.
- Bound by bytes — **FetchBytes(n)** or **PullMaxBytes** — and stay well under 64 MB.
- **PullMaxMessagesWithBytesLimit(batch, bytes)** sets **both** batch size and the byte cap.

fetch.go

```go
// ANTI-PATTERN — a giant count can ask for more
// than the connection's 64 MB buffer can hold
// → stalls, or a slow-consumer disconnect.
msgs, _ := cons.Fetch(100000)

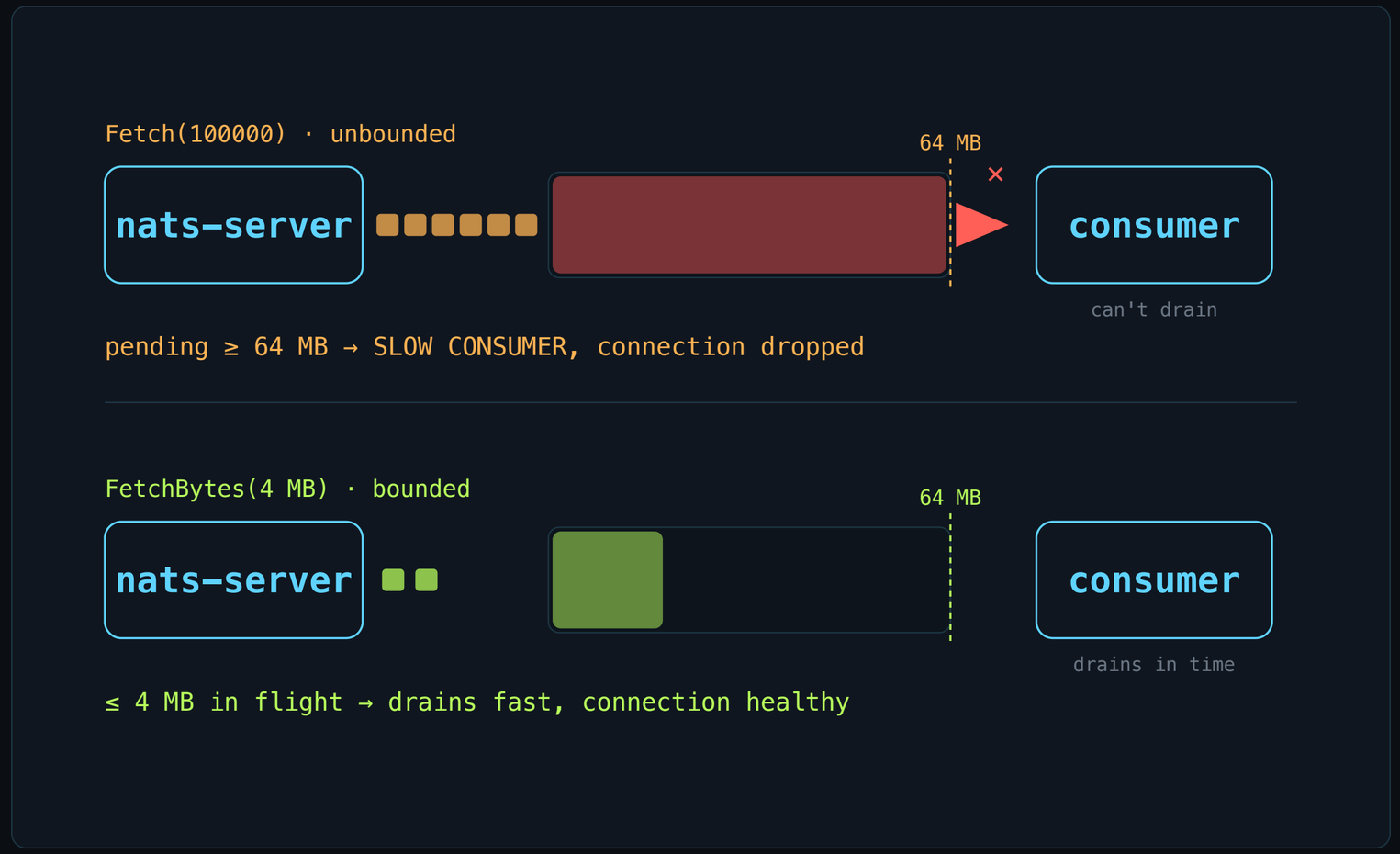
// FIX — bound the pull by BYTES, not just count:
msgs, _ = cons.FetchBytes(4 * 1024 * 1024) // ≤ 4 MB

// Continuous consume, bounded client buffer:
cc, _ := cons.Consume(handle,
    jetstream.PullMaxMessagesWithBytesLimit(
        500, 4*1024*1024)) // batch + max bytes
```

BATCHES · SLOW CONSUMER

Out-fetch the buffer, get dropped

- The server queues your pull's messages in the connection's **64 MB** outbound buffer.
- Fetch faster than you process and it fills — the server flags a **slow consumer** and **closes the connection**.
- Bounding **max bytes** keeps in-flight data small, so the client always drains in time.



05 · CAPACITY

Scaling JetStream.

Plan capacity **before** the pager goes off — accounts, replica cost, ingest limits.

- accounts & routes
- R3 vs R1 cost
- max_ha_assets
- meta snapshots
- ingest limits
- read paths

SCALING · ACCOUNTS

Separate accounts, separate routes

- An account is ultimately bound to **one** TCP route — a noisy tenant blocks the rest.
- Split workloads into accounts to isolate their subject space and blast radius.
- Widen with **pool_size**, or pin hot accounts to their own route.

cluster.conf

```
// Each account hashes to ONE route connection
// in the pool — busy accounts can saturate it.
cluster {
    name: "nats"
    port: 6222

    pool_size: 5   // route connections per peer

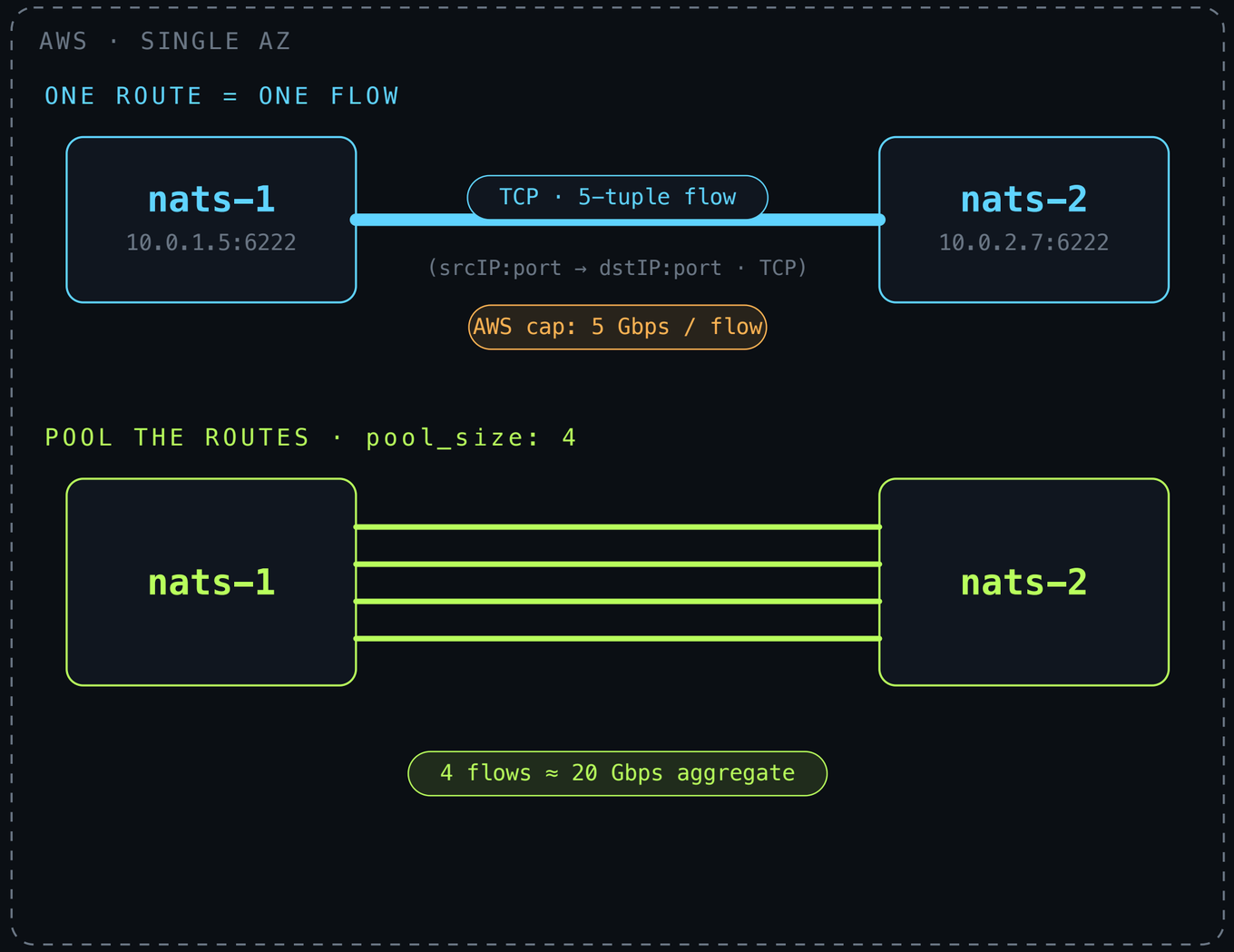
    // Pin hot accounts to a dedicated route.
    accounts: [ "ORDERS", "ANALYTICS" ]

    routes: [ nats://n2:6222, nats://n3:6222 ]
}
```

SCALING · NETWORK

Routes hit the 5 Gbps flow cap

- A route is one **TCP connection** — a single 5-tuple flow.
- On AWS one flow caps at **~5 Gbps**, even on a 25–100 Gbps instance.
- Widen with **pool_size** (more flows), or pin hot accounts to their own route.



SCALING · CONSUMERS

Replicas are not free

- An **R3** consumer costs far more than **R1** — a whole Raft group vs none.
- **MemoryStorage: true** keeps consumer state in RAM — cheap for hot, ephemeral work.
- Cap server-side Raft groups with **max_ha_assets**; mind snapshot size.

consumer.go

```go
// Every R>1 consumer is an "HA asset" — its own
// Raft group: leader, replication, snapshots.

// A cheap consumer: pin to R1 + memory storage,
// decoupled from the stream's replicas and disk.
cons, err := stream.CreateOrUpdateConsumer(ctx,
    jetstream.ConsumerConfig{
        Durable:       "fast-workers",
        AckPolicy:     jetstream.AckExplicitPolicy,
        Replicas:      1,    // no Raft group
        MemoryStorage: true, // state stays in RAM
    })
```

SCALING · META LAYER

Snapshots that block the cluster

⛔ Blocks the apply loop — **every request stalls** for the whole snapshot.

- The meta layer Raft-logs **every** stream & consumer assignment, then snapshots to compact it.
- Async by default — a runaway log forces a **synchronous, blocking** snapshot.
- A stall > 2s also logs **Internal subscription … took too long**.

jetstream.conf

```
// Cluster meta = every stream + consumer assignment.
jetstream {
    meta_compact:      8192    // snapshot every N entries
    meta_compact_size: MB     // ...or N bytes of Raft log
    meta_compact_sync: false   // default: async, non-blocking
}
```

nats-server.log

```
[WRN] metalayer log size has exceeded async threshold (80 MB), will fall back to blocking snapshot

[WRN] Metalayer blocking snapshot starting

[WRN] Metalayer blocking snapshot took 3.214s (streams: 1200, consumers: 327000, compacted: 64 MB)

[WRN] Internal subscription on '$JS.API.STREAM.INFO.ORDERS' took too long: 3.21s
```

SCALING · STREAMS

Mind the ingest queue

- Producers outrun replication — writes buffer in the IPQ, waiting to commit.
- **high message lag** warns; past the limit it **drops** & replies **429**.
- Constant drops = one shard isn't enough — **partition the stream**.

jetstream.conf

```
// Inbound buffers in an IPQ before it commits via Raft.
jetstream {
    max_buffered_msgs: 100000   // IPQ limit
    max_buffered_size: MB    // IPQ limit
}

// Behind by > 10,000 — the server warns:
//   JetStream stream 'APP > ORDERS' has high message lag

// Past the IPQ limit it DROPS (publishers get 429):
//   Dropping messages due to excessive stream
//   ingest rate on 'APP' > 'ORDERS'
```

SCALING · OBSERVE

Watch the queues: /ipqueuesz

- **/ipqueuesz** reports every internal queue's depth — **pending** & in-progress.
- **stream 'X' messages** is the ingest backlog behind "high message lag".
- RAFT **entry** / **committedEntry** = replication & apply lag; consumer queues = ack/pull backlog.

GET /ipqueuesz?queues=ORDERS

```
// pending = queued · in_progress = being applied
{
  "[ACC:APP] stream 'ORDERS' messages": { "pending": 8421, "in_progress": 64 },
  "[ACC:APP] RAFT 'ORDERS' entry": { "pending": 2310 },
  "[ACC:APP] RAFT 'ORDERS' committedEntry": { "pending": 512 },
  "[ACC:APP] consumer 'workers' ackMsgs": { "pending": 73 }
}
```

SCALING · PAYLOADS

Payloads over 8K get compressed

- R>1 streams encode each message into a Raft entry — over **8K** it is S2-compressed.
- Fewer bytes on the wire & disk, but extra CPU: compress on the leader, **decompress on every follower**.
- The 8K counts **subject + reply + headers + body** — and it isn't configurable.
- Keep latency-critical messages under 8K; park big blobs in the **object store**.

jetstream_cluster.go

```
// nats-server, clustered (R>1) stream encoding:
const compressThreshold = 8192 // 8k, fixed

total := len(subject) + len(reply) +
         len(hdr) + len(msg)

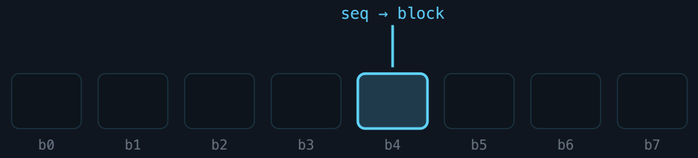
// Over 8K → S2-compress the Raft entry.
if total > compressThreshold {
    encoded = s2Compress(encoded)
}
```

SCALING · READ PATH

Every message load has a worst case

A LoadMsg(seq)

msgForSeq → fetchMsg




```
// replay from a sequence
DeliverPolicy: DeliverByStartSequencePolicy,
OptStartSeq:   1000,
```

WORST cold block fault — read a full block from disk

B LoadLastMsg(subj)

loadLast · psim walk




```
// latest value per subject (KV-style)
DeliverPolicy: DeliverLastPerSubjectPolicy,
FilterSubject: "kv.config.>",
```

WORST wildcard across a huge subject space

C LoadNextMsg(filter)

firstMatching / block

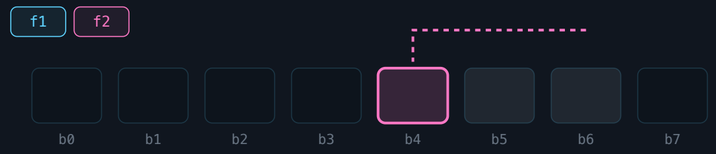



```
// single-filter streaming consumer
DeliverPolicy: DeliverAllPolicy, // or DeliverNew
FilterSubject: "orders.created",
```

WORST sparse filter — scan block after block to the tail

D Multi / Prev

firstMatchingMulti · prev




```
FilterSubjects: []string{"events.a.>","events.b.>"},
DeliverPolicy:  DeliverLastPerSubjectPolicy, // all|new|lps
// stream caps history → MaxMsgsPerSubject: 1
```

WORST sparse multi-filter; backward block walk

READ PATH · A

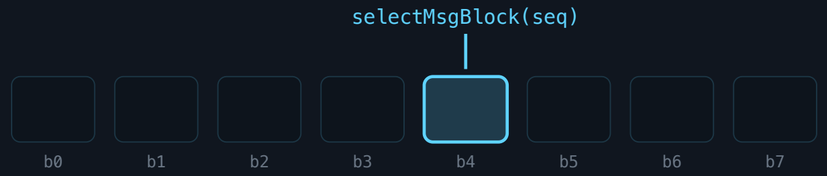
LoadMsg(seq)

msgForSeq → fetchMsg

1. **selectMsgBlock(seq)** finds the block holding the sequence — O(1).
2. **fetchMsg**: on a cold cache, the **whole block** is read from disk.
3. **cacheLookup(seq)** returns it; a linear scan may expire the cache.

WORST random-by-seq on a cold stream → a disk read per message.

FIX read sequentially to keep blocks warm; range with GetBatch.



fetchMsg loads the whole block on a cache miss

O(1) block select · O(block) load when cold

consumer.go

```
jetstream.ConsumerConfig{
    DeliverPolicy: DeliverByStartSequencePolicy,
    OptStartSeq:   142,
}
```

READ PATH · A · FIX

Read sequentially, not randomly

- Random by-seq loads a **different block** each read — most fault to disk.
- A range touches each block **once**; **GetBatch** amortizes the load over thousands of msgs.

by-seq.go

```
msgs, _ := jetstreamext.GetBatch(ctx, js,
    "ORDERS", 1000, jetstreamext.GetBatchSeq(142))
for msg, err := range msgs {
    if err != nil { break }
    handle(msg)
}
```



READ PATH · B

LoadLastMsg(subj)

loadLast · psim walk

1. **psim.Find(subj)** → the subject's last block (fblk / lblk).
2. Walk blocks backward, loading each block's per-subject **fss**.
3. **cacheLookup(last)** returns the newest message for the subject.

WORST a wildcard matches a huge subject space → psim.Match walks them all.

FIX prefer literal subjects; cap cardinality with MaxMsgsPerSubject.



literal subject jumps straight to the last block

literal → 1 block · wildcard → psim over N subjects

consumer.go

```
jetstream.ConsumerConfig{
    DeliverPolicy: DeliverLastPerSubjectPolicy,
    FilterSubject: "kv.config.>",
}
```

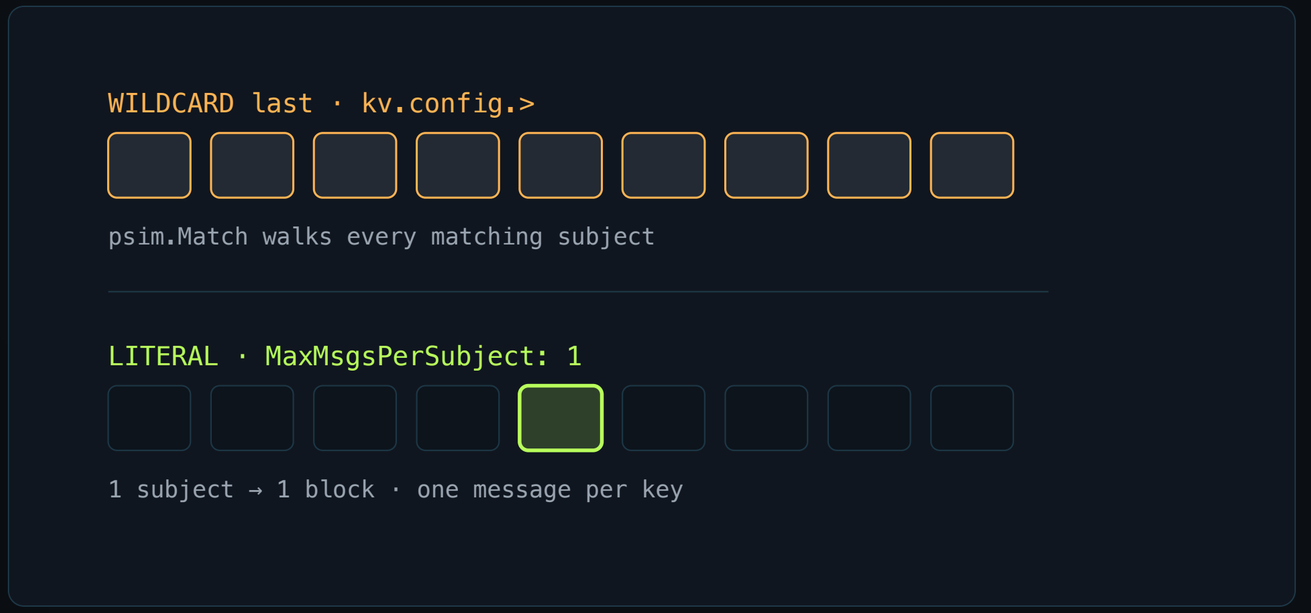
READ PATH · B · FIX

Literal subjects, bounded history

- A literal subject is a **psim.Find** → one block; a wildcard **Match**es every subject.
- **MaxMsgsPerSubject** keeps one message per key — last-per-subject is trivial, storage stays flat.

stream.go

```
js.CreateOrUpdateStream(ctx, jetstream.StreamConfig{
    Name:              "KV_config",
    Subjects:          []string{"kv.config.>"},
    MaxMsgsPerSubject: 1, // newest per key only
})
```



READ PATH · C

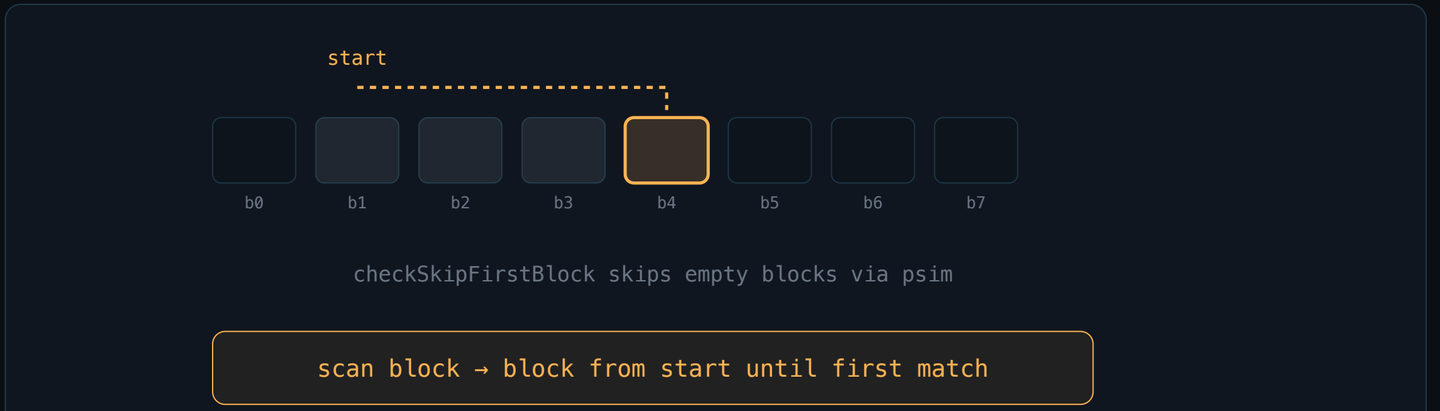
LoadNextMsg(filter)

firstMatching / block

1. Start at the block for **start**, then **firstMatching** per block.
2. Each block loads its **fss** and scans for the first seq ≥ start.
3. **checkSkipFirstBlock** uses psim to jump past empty blocks.

WORST a sparse filter on a busy stream scans block after block to the tail.

FIX dense filters; or a dedicated / sourced stream per consumer.



consumer.go

```
jetstream.ConsumerConfig{
    DeliverPolicy: DeliverAllPolicy, // or DeliverNew
    FilterSubject: "orders.created",
}
```

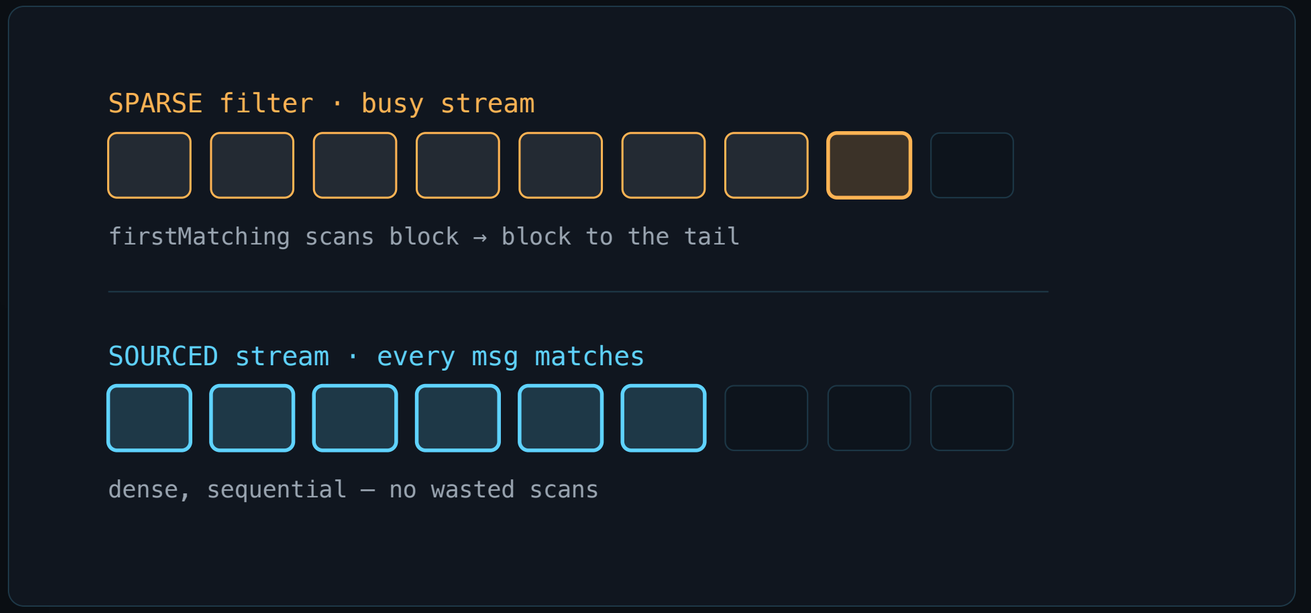
READ PATH · C · FIX

Give the consumer its own stream

- A sparse filter scans past every non-matching message on a busy stream.
- A **sourced** stream copies just those subjects — **every** message matches, so reads are dense.

source.go

```
js.CreateOrUpdateStream(ctx, jetstream.StreamConfig{
    Name: "ORDERS_created",
    Sources: []*jetstream.StreamSource{{
        Name:          "ORDERS",
        FilterSubject: "orders.created",
    }},
})
```



READ PATH · D

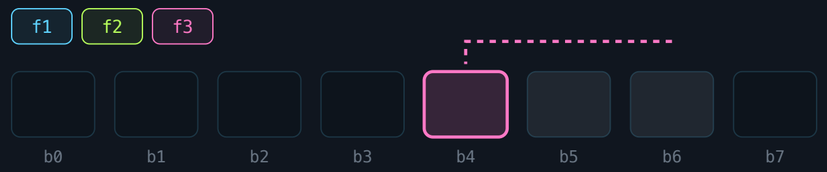
Multi / Prev

firstMatchingMulti · prevMatching

1. **FilterSubjects** compile into a sublist matched per block.
2. **firstMatchingMulti** checks each block's subjects against it.
3. **prevMatching** walks blocks **backward** for reverse / last.

WORST a sparse multi-filter plus a backward walk = the heaviest scan.

FIX keep FilterSubjects under ~100; shard by subject; avoid reverse scans.



firstMatchingMulti matches each block to the sublist

multi-filter scan per block · prev walks backward

consumer.go

```
jetstream.ConsumerConfig{
    FilterSubjects: []string{"events.a.>", "events.b.>"},
    DeliverPolicy:  DeliverLastPerSubjectPolicy,
}
```

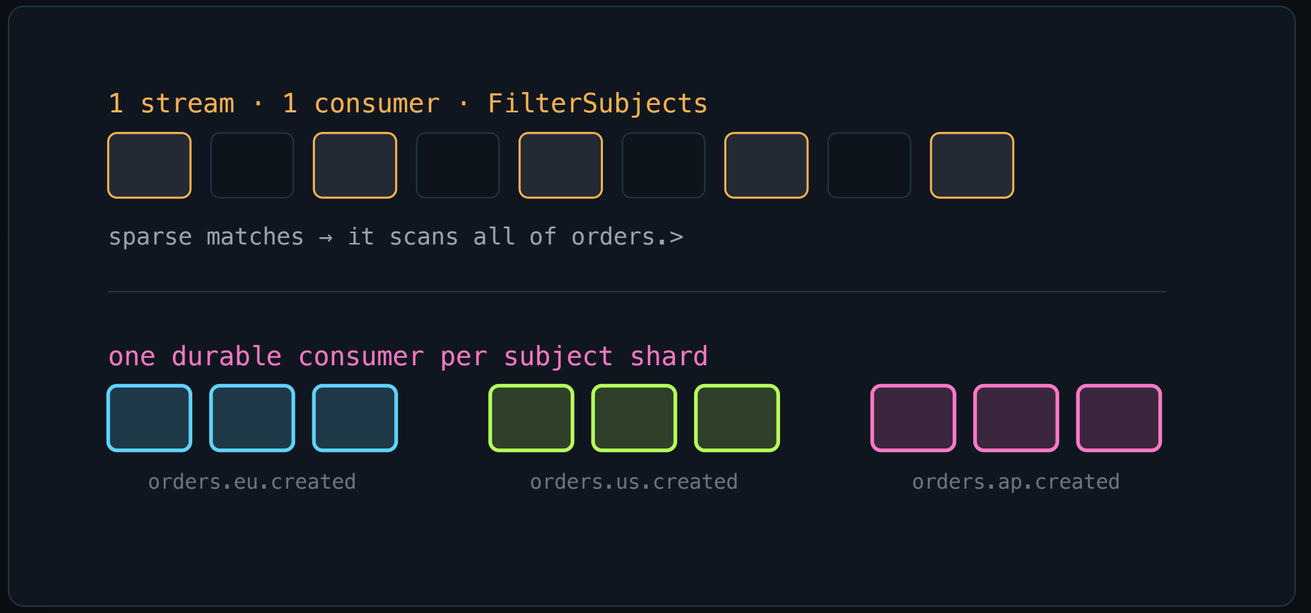
READ PATH · D · FIX

One consumer per subject shard

- Keep **FilterSubjects** reasonable — under **~100**; past that each read scans more.
- Put a shard key in the subject; give **each shard its own durable consumer** — small, dense, parallel.

shard.go

```go
// Shard key lives in the subject: orders.<shard>.created
for _, shard := range []string{"eu", "us", "ap"} {
    cons, _ := stream.CreateOrUpdateConsumer(ctx,
        jetstream.ConsumerConfig{
            Durable:       "fulfil-" + shard,
            FilterSubject: "orders." + shard + ".created",
            AckPolicy:     jetstream.AckExplicitPolicy,
        })
    cc, _ := cons.Consume(handle)
    defer cc.Stop()
}
```



EXTEND · 06

orbit.go/

Battle-tested extensions to **nats.go** — reach for these before you roll your own.

- jetstreamext
- natsext
- counters
- kvcodec
- pcgroups
- natscontext
- natssysclient

EXTEND · jetstreamext

Fetch ranges without a consumer

- **GetBatch** pulls N messages by seq, subject or time — as an iterator.
- **GetLastMsgsFor** returns the head of each subject in one round-trip.
- Perfect for snapshots, backfills and audits — no durable to manage.

getbatch.go

```go
// Replay a precise range — no consumer needed.
msgs, err := jetstreamext.GetBatch(ctx, js,
    "ORDERS", 100, jetstreamext.GetBatchSeq(5000))
for msg, err := range msgs {
    if err != nil {
        break
    }
    fmt.Printf("seq %d: %s\n", msg.Sequence, msg.Subject)
}

// Latest message for each subject, one call.
last, _ := jetstreamext.GetLastMsgsFor(ctx, js,
    "ORDERS", []string{"orders.*.status"})
```

EXTEND · natsext

One request, many replies

- Collect responses from **every** subscriber with a single request.
- Stop on a stall timer, a max count, or a custom **Sentinel**.
- Built for service discovery and fan-out queries.

requestmany.go

```go
// Scatter / gather over core NATS.
replies, err := natsext.RequestMany(ctx, nc,
    "service.ping", nil,
    natsext.RequestManyStall(50*time.Millisecond),
    natsext.RequestManyMaxMessages(16))
for msg, err := range replies {
    if err != nil {
        break
    }
    register(msg.Data)
}
```

EXTEND · counters

Atomic distributed counters

- Counters live in the stream itself — set **AllowMsgCounter**.
- **big.Int** precision, aggregated across every source.
- Increments are atomic — no read-modify-write races.

counters.go

```go
// Stream created with AllowMsgCounter: true.
counter, err := counters.GetCounter(ctx, js, "METRICS")

// Atomic, arbitrary precision, multi-source.
_, err = counter.AddInt(ctx, "events.clicks", 1)

total, _ := counter.Load(ctx, "events.clicks")
fmt.Printf("clicks = %s\n", total.String())
```

EXTEND · kvcodec

KV keys, your way

- Use **/path/style** or escaped keys the raw KV API would reject.
- Codecs wrap **any** KeyValue store — the API stays identical.
- Chain key and value codecs, e.g. path + base64.

kvcodec.go

```go
kv, _ := js.CreateKeyValue(ctx,
    jetstream.KeyValueConfig{Bucket: "config"})

// Familiar /path keys → NATS subjects.
store := kvcodec.NewForKey(kv, kvcodec.PathCodec())

store.Put(ctx, "/app/db/host", []byte("10.0.0.5"))
e, _ := store.Get(ctx, "/app/db/host")
fmt.Println(string(e.Value())) // 10.0.0.5
```

OPERATE · natssysclient

Health-check from your app

- **Healthz** checks a server — or a specific **Account / Stream / Consumer**.
- Get back **Status: "ok"** or a list of **Errors** (no leader, not current…).
- Also read **Varz** / **Jsz** / **Connz**; one server or ping the cluster.

sysclient.go

```go
sys, _ := natssysclient.NewSysClient(nc)

// Health of a specific stream on a server.
h, _ := sys.Healthz(ctx, serverID,
    natssysclient.HealthzOptions{
        Account: "APP", Stream: "ORDERS",
    })
fmt.Println(h.Healthz.Status) // "ok"

// Health of a consumer (Account is required).
h, _ = sys.Healthz(ctx, serverID,
    natssysclient.HealthzOptions{
        Account:  "APP",
        Stream:   "ORDERS",
        Consumer: "workers",
    })
```

SHIP IT

The JetStream checklist

- ✓ One connection per process; every call takes a **context**.
- ✓ Declare streams with **CreateOrUpdateStream** — idempotent config.
- ✓ Stable **Msg-ID** + **Duplicates** window → retries are safe.
- ✓ Durable pull consumers, **explicit ack**, bounded **MaxDeliver**.
- ✓ Need order at scale? **Partitioned consumer groups**.
- ✓ Plan capacity: separate **accounts**, mind **R3** cost, cap buffered writes.
- ✓ Don't reinvent it — **orbit.go** probably already has it.

OVER TO YOU

Q&A?

Streams, consumers, scaling, the read paths, orbit.go — ask away.

COUNTERS · HISTORY

Read a counter's full history

- The **counters** API only returns the **latest** value (GetLastMsgForSubject).
- Each increment is a message: **Time** + **Nats-Incr** delta + running total in the body.
- The full series is yours — as long as the stream still **retains** the messages.

counter-history.go

```go
// The counters API returns only the latest value.
// Full history → read the increment messages:
msgs, _ := jetstreamext.GetBatch(ctx, js, "METRICS",
    1000, jetstreamext.GetBatchSubject("events.clicks"))

for msg, err := range msgs {
    if err != nil { break }
    var b struct{ Val string `json:"val"` } // body
    json.Unmarshal(msg.Data, &b)
    fmt.Printf("%s  +%s → %s\n",
        msg.Time.Format(time.RFC3339),       // timestamp
        msg.Header.Get("Nats-Incr"), b.Val)  // delta, total
}
```

THANK YOU

Build it reliably.

nats.go	github.com/nats-io/nats.go
orbit.go	github.com/synadia-io/orbit.go
demos	./js-apps

THE PATH

From connect to production

01	Connect & configure	one connection, context-first
02	Design your streams	subjects, retention, dedup
03	Produce safely	idempotency & throughput
04	Consume reliably	durable pulls, ack discipline
05	Scaling JetStream	accounts, replica cost, ingest limits
06	Reach for orbit.go	extensions you shouldn't rebuild

03 · PRODUCE

Make publishes idempotent

- Give every message a stable **Nats-Msg-Id** via **WithMsgID**.
- Pair it with the stream's **Duplicates** window to dedupe.
- Check **ack.Duplicate** — now retries are **safe**, not double-writes.

publish.go

```go
// Same Msg-ID inside the window = stored once.
ack, err := js.Publish(ctx, "orders.created",
    payload, jetstream.WithMsgID(order.ID))
if err != nil {
    return err
}
if ack.Duplicate {
    log.Printf("dup at seq %d", ack.Sequence)
}
```

03 · PRODUCE

Batch with async publish

- Async pipelines the acks — far higher throughput than sync.
- Bound in-flight messages with **WithPublishAsyncMaxPending**.
- Always wait on **PublishAsyncComplete** before you exit.

publish_async.go

```go
js, _ := jetstream.New(nc,
    jetstream.WithPublishAsyncMaxPending(4096))

for _, o := range orders {
    _, err := js.PublishAsync("orders.created",
        o.Bytes(), jetstream.WithMsgID(o.ID))
    if err != nil {
        return err
    }
}
select {
case <-js.PublishAsyncComplete():
case <-ctx.Done():
    return ctx.Err()
}
```

SCALING · ORDERING

Ordered and parallel

- Partition by subject token: ordered **per key**, throughput across keys.
- Each member runs **MaxAckPending: 1** without the global bottleneck.
- Elastic groups add and drop members live. **orbit.go/pcgroups**, nats-server 2.11+.

pcgroups.go

```go
// Order per key, parallel across keys.
cfg := jetstream.ConsumerConfig{
    MaxAckPending: 1, // strict order in a partition
    AckPolicy:     jetstream.AckExplicitPolicy,
}
cc, err := pcgroups.ElasticConsume(ctx, js,
    "ORDERS", "fulfillment", "worker-1",
    func(msg jetstream.Msg) {
        process(msg)
        msg.Ack()
    }, cfg)
defer cc.Stop()
```

OPEN SOURCE · 2026 · NATS CLIENT UTILITIES

Synadia Orbit.

// the NATS extensions

Independent client utilities for NATS — in **Go, Java, .NET, JS, Python, Rust & C**. Every module stands alone: **go get** just the ones you need.

go net

ORBIT · CONNECT

natscontext

- Connect using the same named **Contexts** the **nats** CLI uses.
- Reuse contexts from **nats context add** — creds, JS domain, TLS and more.
- Accept a friendly context name **or** a full path to a context JSON file.
- Returns the connection **plus** parsed settings (e.g. the JS domain).
- Supports every context setting except Windows Certificate Store options.

natscontext.go

```go
// Connect with a named context — same as the nats CLI.
nc, settings, err := natscontext.Connect("production")
if err != nil {
    log.Fatal(err)
}

// Settings carry the JS domain, creds, TLS…
js, _ := jetstream.NewWithDomain(nc, settings.JSDomain)
```

go java rs net

ORBIT · CORE NATS

natsext

one request, many replies

- **RequestMany**: one request, **many** responses — scatter-gather & streaming replies.
- Responses arrive through an **iterator** you range over.
- The iterator closes itself when a termination condition is met.

stall timer | max count | sentinel message

requestmany.go

```go
// One request, many replies — range the iterator.
it, _ := natsext.RequestMany(ctx, nc, "service.ping", nil,
    natsext.RequestManyStall(50*time.Millisecond), // stall timer
    natsext.RequestManyMaxMessages(16))           // max count
for msg, err := range it {
    if err != nil { break }
    register(msg.Data)
}
```

go

ORBIT · SYSTEM API

natssysclient

- Client for the server monitoring / system-account APIs.
- Query a **single** server by ID, or the **whole cluster** at once.
- Cluster queries are request-many: wait for a timeout or N replies, then aggregate.
- Exposes **VARZ** and the other sys-account endpoints.
- Response structs track future server additions.

sysclient.go

```go
sys, _ := natssysclient.NewSysClient(nc)

// One server by ID…
varz, _ := sys.Varz(ctx, serverID,
    natssysclient.VarzEventOptions{})

// …or ping the whole cluster and aggregate.
all, _ := sys.VarzPing(ctx,
    natssysclient.VarzEventOptions{})
```

go java rs net js c

ORBIT · JETSTREAM EXT

jetstreamext

fetch ranges without a consumer

- **GetBatch**: fetch N messages from a stream — with or without a subject filter.
- From a sequence **or** a point in time; bounded by count **or** max bytes.
- **GetLastMsgsFor**: the last message(s) for matching subject filter(s).
- **GetBatch** goes **forward** in time; **GetLastMsgsFor** goes **backward**.

getbatch.go

```go
// Forward: a batch from a sequence — no consumer.
msgs, _ := jetstreamext.GetBatch(ctx, js, "ORDERS",
    100, jetstreamext.GetBatchSeq(5000))

// Backward: the last message per matching subject.
last, _ := jetstreamext.GetLastMsgsFor(ctx, js,
    "ORDERS", []string{"orders.*.status"})
```

go java rs net js

ORBIT · JETSTREAM EXT

jetstreamext — atomic batch publishing

BatchPublisher / **PublishMsgBatch** — commit a set of messages as a single unit.

ATOMIC

- Pass a whole batch, or open-ended — signal **Commit** on/after the last message.
- Atomicity is guaranteed; messages can still carry **Compare-And-Set** options.
- Messages are actually stored on **Commit**.

FAST INGRESS

- A **fast-ingress** path for high-throughput writes — **no** atomicity guarantee.
- Messages are stored as they arrive; **Commit** just asks for the final ack.
- Speed comes from one PubAck every **n** messages (also flow control).

go java net

ORBIT · KV CODECS

kvcodec

- Transparent encode/decode wrapper around JetStream KV stores.
- Full KV API compatibility — codecs are **invisible** to calling code.
- Separate, independent codecs for **keys** and **values**.
- Escape invalid key chars, store **/path/style** keys as subjects, base64 values.
- Chain multiple codecs together, or write your own.

kvcodec.go

```go
kv, _ := js.CreateKeyValue(ctx,
    jetstream.KeyValueConfig{Bucket: "config"})

// /path keys -> NATS subjects, transparently.
store := kvcodec.NewForKey(kv, kvcodec.PathCodec())
store.Put(ctx, "/app/db/host", []byte("10.0.0.5"))
e, _ := store.Get(ctx, "/app/db/host")
```

go java rs net js c

ORBIT · DISTRIBUTED COUNTERS

counters

- Distributed counters on streams configured with **AllowMsgCounter**.
- Works with **sourcing** too; each subject is its own independent counter.
- Arbitrary-precision values (**big.Int**).
- Operations: **Add**, **Load**, **Get** (source tracking), **GetMultiple**.

counters.go

```go
c, _ := counters.GetCounter(ctx, js, "METRICS")

// Atomic increment, arbitrary precision.
c.Add(ctx, "events.clicks", big.NewInt(1))

v, _ := c.Load(ctx, "events.clicks")  // current value
all, _ := c.GetMultiple(ctx,          // several at once
    []string{"events.clicks", "events.views"})
```

go java net

ORBIT · PARTITIONED CONSUMER GROUPS

pcgroups

Client-side partitioned consumer groups for JetStream (nats-server 2.11). Ordered **per partition key** — effectively per-key **MaxAckPending: 1** — with deterministic distribution across consuming instances.

STATIC GROUPS

- For streams that already carry the partition number as the first subject token.
- Fixed membership; consumers created directly on the stream.

DISTRIBUTION & HA

- Balanced distribution or explicit partition mappings.
- Multiple instances per member — one pinned/active, rest hot standby.
- **ElasticMemberStepDown()** / **StaticMemberStepDown()** force failover.

ELASTIC GROUPS

- Work on **any** stream; add / drop members administratively at any time.
- The library manages a sourced work-queue stream and inserts partition tokens.

TRANSPARENT TO APP CODE

- Join with a member name, a consumer config and a callback.
- The library strips the partition token; a **cg** CLI handles admin / testing.

DEEP CUTS · SOURCING & MIRRORING

Things you may not know — sourcing

- Each source can have a **filter** plus a subject-transform destination.
- A source can have **multiple** filter+transform pairs — as long as the filters don't overlap.
- Source from many streams, or from the **same** stream more than once.
- Only one filter+transform combination per stream being sourced.
- Create **duplicates** of a stream's messages with different destination transforms.
- e.g. source **foo.>** with no transform, and **foo.>** → **bar.>**.
- Source from a **durable consumer** on a stream (2.14) — enables sourcing Interest/WQ streams.
- The stream's own subject transform still applies after sourcing.
- **Upgrade** a mirror stream to a sourcing stream (2.14).
- The deduplication window now works with sourcing — and is **persistent**.

DEEP CUTS · CONSUMER PRIORITY GROUPS

Things you may not know — priority groups

Pull-consumer priority groups change how messages are handed out across client instances.

PINNING

- Many app instances pull from one consumer, but only one receives at a time.
- Active-passive failover across the group.

PRIORITIZED

- Higher-priority workers get messages immediately — no threshold wait.
- Enables regional awareness and new design patterns.

OVERFLOW

- By pending messages or pending acks, overflow spills to other clients.
- Keep work regional; fail out only when the primary falls behind.

LIMITS

- Pull consumers only — and always will be.
- One priority group per consumer (currently).

POSITIONING · VS KAFKA

Comparing with Kafka

Apache Kafka

- One topic = one stream; no server-side filtering.
- Pure write-ahead-log persistence.
- “Just streaming” — everything else means more JVMs: MirrorMaker, KStreams, Flink, connectors…

NATS JetStream

- One stream, hierarchies of subjects; consumer filters run on the **server**.
- A **NoSQL** data store: delete any message anywhere (GDPR), per-subject limits, limits-as-constraints.
- Compare-And-Set concurrency; distributed “increment-in-place” counters.
- More than streaming: low-latency Core NATS request/reply, multi-tenant security, super-clustering & leaf nodes, partition only when you need it.

ANTI-PATTERNS · AVOID

“Bad ideas” — things to avoid

The pending count is already in fetched-message metadata — you rarely need to ask.

- ✗ Sending too many **asynchronous** publications to a stream.
- ✗ Fetching with **no wait**.
- ✗ Fetching **more than one** message with an infinite wait.
- ✗ Spamming consumer **info**.

ANTI-PATTERNS · GREY AREA

“Bad ideas” — pay attention to…

- Creating / deleting consumers all the time.
  Can you use a batched get instead?
- Many streams with a single subject each.
  Can you aggregate via subject-based addressing?
- One giant stream holding everything.
  Will it become a bottleneck?
- Subjects on stored messages you don't need.
  Lots of subjects is fine, but indexes cost server memory.
- Edge R1 JetStream on power that can fail.
  Consider sync=always.
- Explicitly acking every consumed message.
  Higher quality of service, but costlier.
- max bytes vs max messages limits.
  max bytes is a little more expensive; mind too many sources into one stream.

DESIGN · KEY VALUE

Key Value — when to use KV (or not)

KV is an **opinionated view** of a stream — not everything a stream does is exposed, but everything KV does you can do on the Stream API directly.

KV MAPS ONTO STREAMS

- put = publish · get = direct get.
- watch = ordered consumer.
- delete = roll-up: delete the message + publish a delete marker.

THE REAL DIFFERENCE: DELETES & HISTORY

- Care **when/if** keys are deleted? Use the KV API.
- Never-repeating keys? Maybe skip KV — or TTL the delete markers.
- Want per-key history? KV covers it (hard cap 64) — or batched-get on the stream.
- KV will add atomic batching later (not yet).