



What if your message broker can
also manage your workloads ?

About me

Luc Juggery

Developer Relations Engineer



Previous

- Co-founder energy efficiency / IoT startups
- Devops / Cloud architect
- Docker / Kubernetes trainer

Personal website: <https://luc.run>



European Cloud Provider



 Compute	 Kubernetes
 Object Storage	 Block Storage
 GPU Servers	 DNS
 Virtual Private Cloud	 DBaaS



Disclaimer : why a talk about NATS ?

- I'm not working at Synadia
- I'm not a NATS contributor

... but...

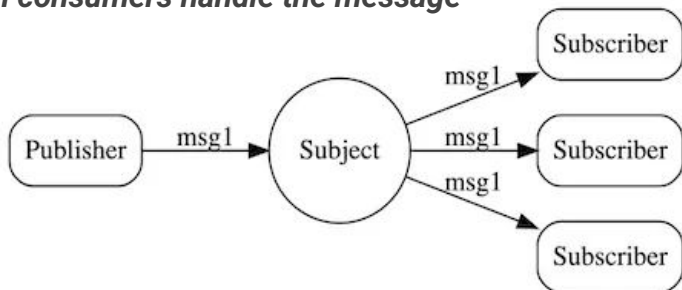
- I'm a user who likes this piece of technology
- I use it in various projects (pro and personal)
- I like to explore new ways to manage workloads (promise of **Nex**)

NATS : a quick history

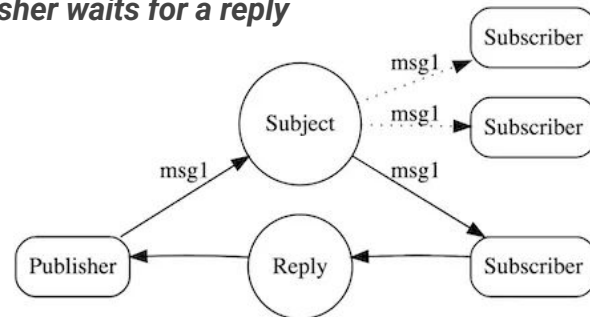
- Message broker (Publish / Subscribe, ...)
- Created in **2011**
- Donated to the CNCF in **2018** (currently in **Incubating**)
- Used by many companies
- Evolution of the feature set
 - communication layer (message broker)
 - storage layer (data stream / Jetstream)
 - **workload layer (execution engine / Nex) ⇒ Experimental**

Message Broker

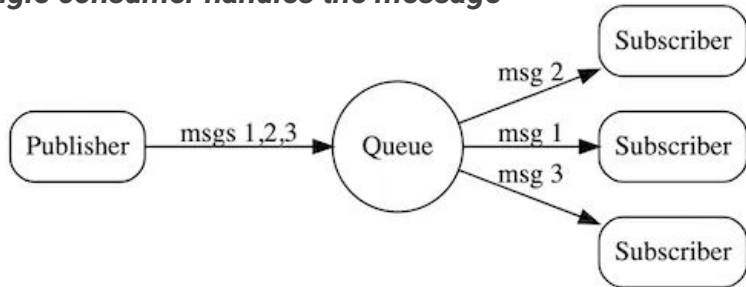
Several consumers handle the message



Publisher waits for a reply



A single consumer handles the message



```
# Commands in nats CLI
$ nats pub ...
$ nats sub ...
$ nats reply ...
$ nats req ...
```

JetStream Persistent Engine

- Built into NATS Server
- Store and replay messages
 - great in environment with connectivity issues
 - **streams** store messages
 - **consumers** allows clients to consume a subset of messages from a stream
- Storage solutions built on top
 - Key Value store
 - Object store

```
# Using nats CLI  
$ nats stream ...  
$ nats consumer ...
```

```
# Using nats CLI  
$ nats kv  
$ nats object ...
```

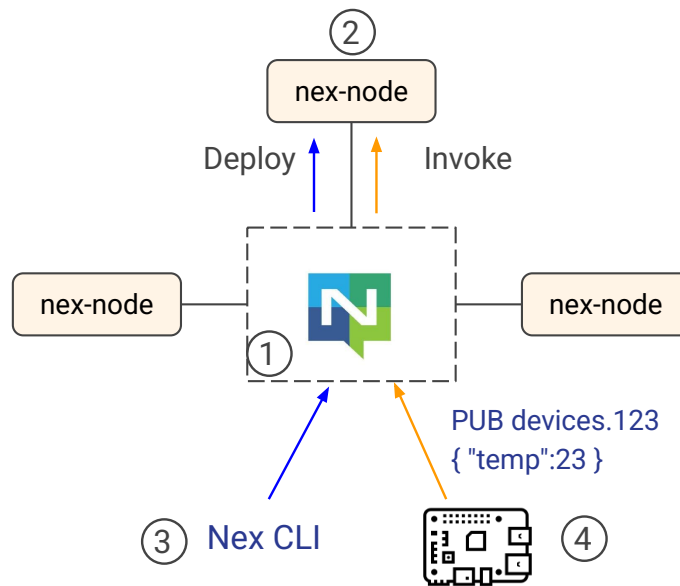
workload
storage
communication

NATS Execution Engine (Nex)

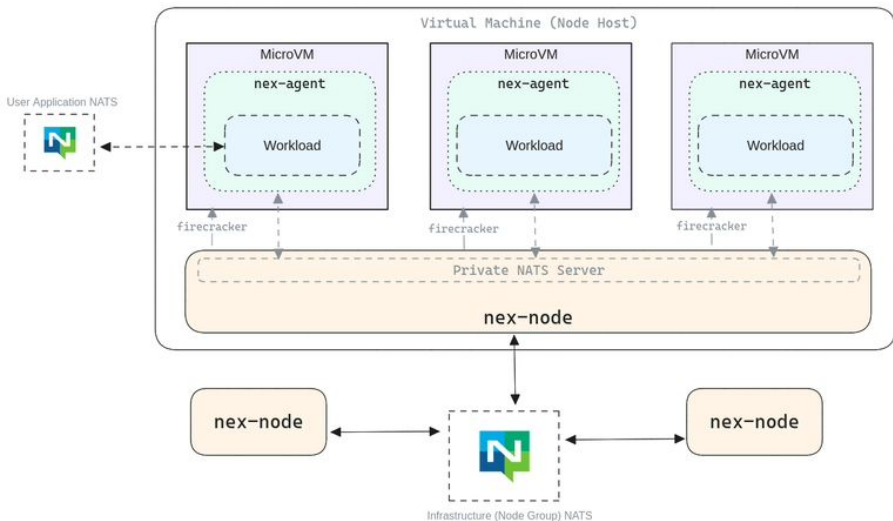
- Deploy and run workloads on nex nodes through NATS
- Specific **nex** CLI
- Services : long-running processes
- Functions : short-lived event-triggered processes

Getting started with Nex

1. Run NATS with JetStream enabled
2. Run Nex nodes connecting to NATS
3. Deploy a function using Nex CLI
4. Invoke the function



Nex : architecture overview



nex-node

- run your workloads in isolated environments
- exposes a control api over NATS

nex-agent

- manages one workload
- bridge between NATS and the workload
- receives messages / runs workload / sends results back

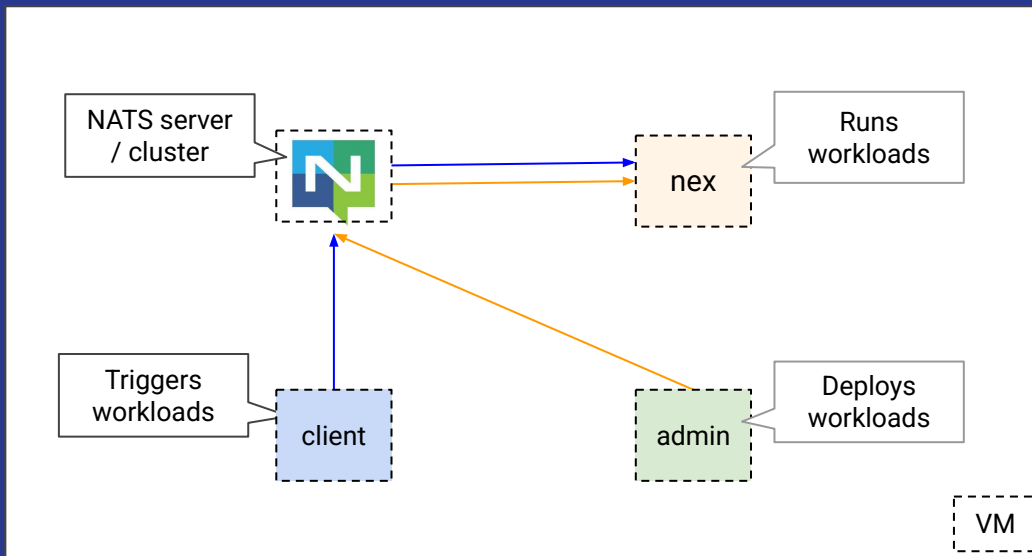
Nex : service

- Long running process (api, ...)
- In **sandbox** mode
 - a service runs in a Firecracker microVM
 - statically-linked linux binary
- In **no sandbox** mode, a service runs on the host
 - Linux / macOS / Windows

Nex : function

- Short lived process similar to AWS Lambda
- Triggered by a message on a specific NATS subject
- In **sandbox** mode, a function runs in a Firecracker microVM
- In **no sandbox** mode, a function runs on the host
- Implemented in JavaScript or compiled in WebAssembly

Demo - Nex



Deploying a Javascript function

```
(subject, payload) => {  
  console.log(subject);  
  return {  
    triggered_on: subject,  
    payload: String.fromCharCode(...payload)  
  }  
};
```

echo.js



`nex devrun ./echo.js --trigger_subject=echo --type v8`

Deploys the function and links it to the "echo" subject



`nats req echo 'hello nex!'`

Invokes the function sending a message on "echo" subject

Deploying a WASM function

```
use std::{io::{self, Read, Write}};  
fn main() {  
    let mut buf = Vec::new();  
    io::stdin().read_to_end(&mut buf).unwrap();  
    io::stdout().write_all(&mut buf).unwrap();  
}
```

main.rs

```
[package]  
name = "echo"  
version = "0.1.0"  
edition = "2021"  
  
[[bin]]  
name = "echo"  
path = "src/main.rs"
```

Cargo.toml

↳ `cargo build --target wasm32-wasi --release` Builds the function for webassembly

↳ `nex devrun echo.wasm --trigger_subject=echo.wasm --type wasm` Deploys the function

↳ `nats req echo.wasm 'hello nex!'` Invokes the function

Host Services

Javascript functions can access host services

- NATS communication layer
- KV Store
- Object Store
- HTTP Client

Access to host Services (1/2)

```
(subject, payload) => {  
  let data = JSON.parse(String.fromCharCode(...payload));  
  const deviceId = data.deviceId || "unknown";  
  const temp = data.temperature;  
  
  this.kv("iot_devices").set(deviceId, payload);  
  this.objectStore("iot_history").put(deviceId + "-" + Date.now() + ".json", payload);  
  
  let httpAlert = null;  
  if (typeof temp === "number" && temp > 30) {  
    const body = JSON.stringify({ deviceId, temperature: temp, ts: Date.now(), subject });  
    const bytes = new Uint8Array(body.length);  
    for (let i = 0; i < body.length; i++) bytes[i] = body.charCodeAt(i) & 0xff;  
    const resp = this.http.post("https://webhooks.app/data", bytes);  
    httpAlert = { status: resp?.status, length: resp?.response?.length };  
  }  
  return {  
    ok: true, subject, device: deviceId, temperature: temp, kv_bucket: "iot_devices",  
    object_bucket: "iot_history", http_alert: httpAlert};  
};
```

KV: store the incoming payload

Append history item in Object Store

Send HTTP alert if temp > 30

telemetry.js

Access to host Services (2/2)

```
(subject, payload) => {  
  // Get jokes from KV store  
  const jokesData = this.kv("dad_jokes").get("jokes");  
  const jokes = JSON.parse(String.fromCharCode(...jokesData));  
  
  // Pick a random joke  
  const randomIndex = Math.floor(Math.random() * jokes.length);  
  const joke = jokes[randomIndex];  
  
  // Return the joke  
  return {  
    joke: joke.setup,  
    punchline: joke.punchline  
  };  
};
```

Get content from
KV store

jokes.js

Deploying a service

```
package main
```

```
import (  
    "context"  
    "os"  
    "github.com/nats-io/nats.go"  
    services "github.com/nats-io/nats.go/micro"  
)
```

```
func main() {  
    ctx := context.Background()  
    natsUrl := os.Getenv("NATS_URL")  
    if natsUrl == "" { natsUrl = "nats://192.168.127.1:4222" }
```

URL to connect to
NATS from within
the Firecracker VM

```
    nc, err := nats.Connect(natsUrl)  
    if err != nil { panic(err) }  
  
    echoHandler := func(req services.Request) { req.Respond(req.Data()) }
```

```
    _err = services.AddService(nc, services.Config{  
        Name: "EchoService",  
        Version: "1.0.0",  
        Endpoint: &services.EndpointConfig{  
            Subject: "svc.echo",  
            Handler: services.HandlerFunc(echoHandler),  
        },  
    })  
    if err != nil { panic(err) }  
    <-ctx.Done()  
}
```

echo.go



Static compilation

```
CGO_ENABLED=0 go build -tags netgo -ldflags '-extldflags "-static"' -o echo
```



```
nex devrun ./echo
```

Deploying the service



```
nats req svc.echo hello
```

Invoking the service

Echoes the payload
received on **svc.echo**

Nex status

- Current Status
 - open source: experimental, APIs may change
 - commercial: available as Workloads in Synadia cloud (containers + JS functions)
- Current Limitations of OSS (*using 0.3.0-rc1*)
 - manual node management
 - limited JavaScript runtime (v8go constraints)
 - not easy to debug workloads
- Coming next
 - OCI image support to run workloads from container images

When might Nex be worth exploring?

- Worth exploring if you are
 - comfortable with experimental technology
 - already using NATS
 - building event-driven or IoT applications
 - looking for Lambdas triggered by NATS messages instead of HTTP
- Probably not the right fit if you have
 - monolithic applications
 - simple CRUD over HTTP
 - no streaming requirements

Takeaways

- NATS is evolving: communication → storage → execution
- Nex runs services (long-running processes) and functions (event driven)
- Experimental but promising
- Easy to try

Want to try Nex ?

- NATS Documentation : <https://docs.nats.io/using-nats/nex>
- NATS Slack Channel: <https://slack.nats.io> 👍
- Nex GitHub repository : <https://github.com/synadia-io/nex>



Questions ?



Thank you !

<https://luc.run/scnd25.pdf>