The Zen of High Performance Messaging with NATS

Waldemar Quevedo / @wallyqs

Strange Loop 2016

ABOUT

- Waldemar Quevedo / @wallyqs
- Software Developer at Apcera in SF
  - Development of the Apcera Platform
- Past: PaaS DevOps at Rakuten in Tokyo
- NATS client maintainer (Ruby, Python)

ABOUT THIS TALK

- What is NATS
- Design from NATS
- Building systems with NATS

What is NATS?

NATS

- High Performance Messaging System
- Created by Derek Collison
- First written in Ruby in 2010
  - Originally built for Cloud Foundry
- Rewritten in Go in 2012
  - Better performance
- **Open Source, MIT License**
  - https://github.com/nats-io

THANKS GO TEAM

Small binary → Lightweight Docker image

```
$ docker images
REPOSITORY          TAG                              IMAGE ID            CREATED             SIZE
nats                0.9.4                            6057c5dae1e2        13 days ago         7.538 MB
$ docker run nats:0.9.4
[1] 2016/09/01 15:03:05.514978 [INF] Starting nats-server version 0.9.4
[1] 2016/09/01 15:03:05.515040 [INF] Starting http monitor on 0.0.0.0:8222
[1] 2016/09/01 15:03:05.515111 [INF] Listening for client connections on 0.0.0.0:4222
[1] 2016/09/01 15:03:05.515158 [INF] Server is ready
[1] 2016/09/01 15:03:05.515297 [INF] Listening for route connections on 0.0.0.0:6222
```

No deployment dependencies

Acts as an always available *dial-tone*

Performance

```
NATS @ ~/repos/nats-dev/src/github.com/nats-io/nats/examples (master) $ ./nats-bench -n 100000000 -np 20 -ms 1 a
Starting benchmark [msgs=100000000, msgsize=1, pubs=20, subs=0]
```

single byte message

```
 0 bash
NATS server version 0.9.4 (uptime: 1h6m25s)
Server:
  Load: CPU:  216.5%  Memory: [illegible]  Slow Consumers: 0
  In:   Msgs: 670.0M  Bytes: 670.0M  Msgs/Sec: 8789133.9  Bytes/Sec: 8.4M
  Out:  Msgs: 0  Bytes: 0  Msgs/Sec: 0.0  Bytes/Sec: 0
```

Around 10M messages/second

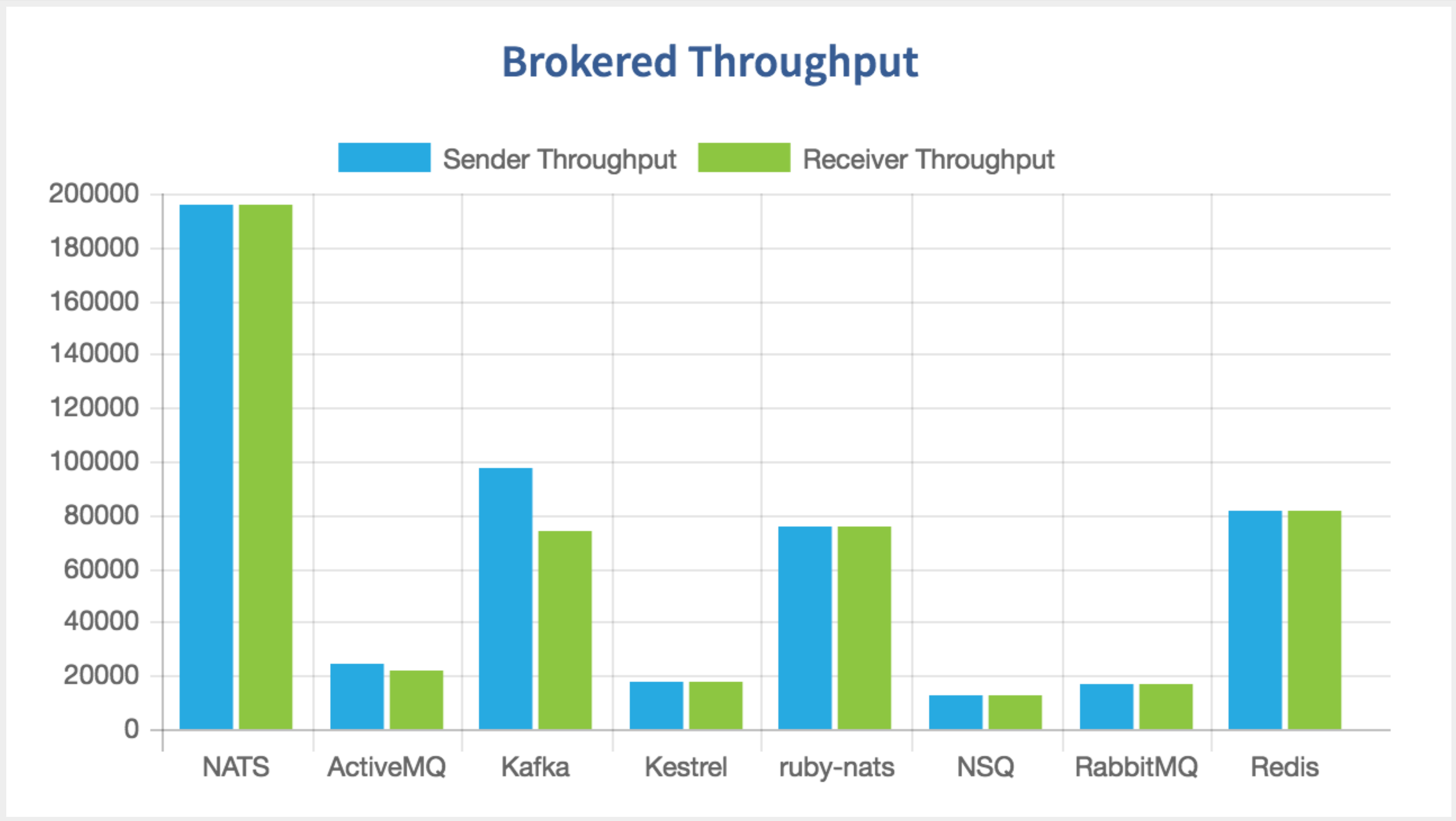
Connections: 20

HOST	CID	NAME	SUBS	PENDING	MSGS_TO	MSGS_FROM	BYTES_TO	BYTES_FRO
::1:55145	627		0	0	0	456.7K	0	456.7K
::1:55146	628		0	0	0	504.2K	0	504.2K
::1:55148	629		0	0	0	530.3K	0	530.3K
::1:55147	630		0	0	0	552.7K	0	552.7K
::1:55149	631		0	0	0	393.7K	0	393.7K

1 bash

MUCH BETTER BENCHMARK

From @tyler_treat's awesome blog



http://bravenewgeek.com/dissecting-message-queues/ (2014)

NATS = Performance + Simplicity

Design from NATS

NATS

Design constrained to keep it as **operationally simple** and **reliable** as possible while still being both **performant** and **scalable**.

Simplicity Matters!

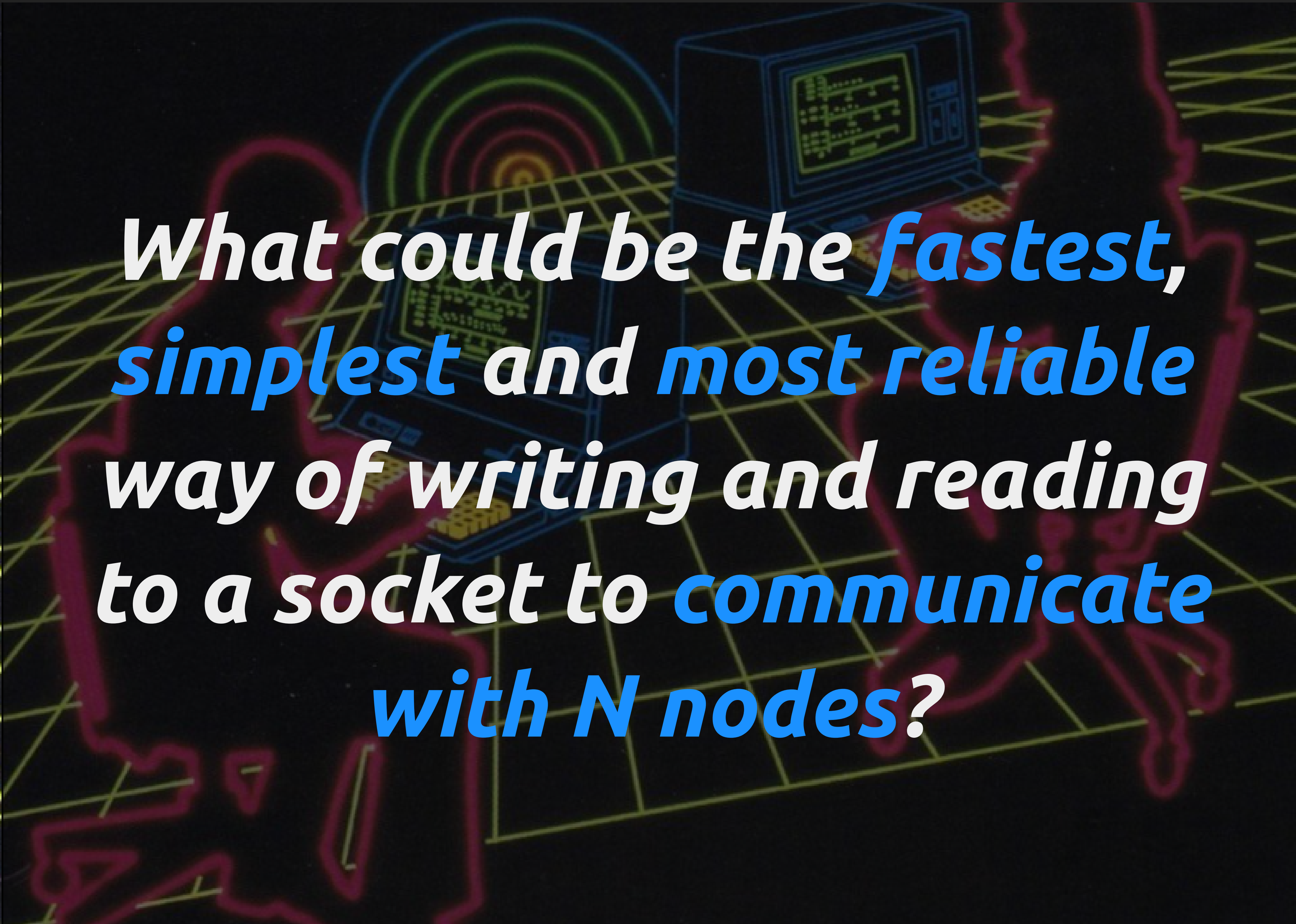
> *Simplicity buys you opportunity.*
>
> *— Rich Hickey, Cognitect*

Link: https://www.youtube.com/watch?v=rI8tNMsozo0

LESS IS BETTER

Concise feature set (pure pub/sub)
No built-in persistence of messages
No exactly-once-delivery promises either
Those concerns are *simplified* away from **NATS**

THOUGHT EXERCISE



What could be the **fastest**, **simplest** and **most reliable** way of writing and reading to a socket to **communicate with N nodes**?

DESIGN

- **TCP/IP** based
- Plain text protocol
- Pure pub/sub
  - *fire and forget*
  - *at most once*

PROTOCOL

```
PUB
SUB
UNSUB
MSG
PING
PONG
INFO
CONNECT
-ERR
+OK
```

EXAMPLE

Connecting to the public demo server…

```
telnet demo.nats.io 4222

INFO {"tls_required":false,"max_payload":1048576,...}
```

Optionally giving a name to the client

```
connect {"name":"nats-strangeloop-client"}
+OK
```

Pinging the server, we should get a pong back

```
ping
PONG
```

Not following ping/pong interval, results in server disconnecting us.

```
INFO {"auth_required":false,"max_payload":1048576,...}
PING
PING
-ERR 'Stale Connection'
Connection closed by foreign host.
```

Subscribe to the `hello` subject identifying it with the arbitrary number 10

```
sub hello 10
+OK
```

Publishing on `hello` subject a payload of 5 bytes

```
sub hello 10
+OK
pub hello 5
world
```

Message received!

```
telnet demo.nats.io 4222

sub hello 10
+OK
pub hello 5
world
MSG hello 10 5
world
```

Payload is opaque to the server!

It is just bytes, but could be json, msgpack, etc…

REQUEST/RESPONSE

is also pure pub/sub

PROBLEM

How can we send a request and expect a response back with pure pub/sub?

NATS REQUESTS

Initially, client making the request creates a subscription with a unique identifier string:

```
SUB _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 2
+OK
```

NATS clients libraries have helpers for generating these:

```
nats.NewInbox()
// => _INBOX.ioL1Ws5aZZf5fyeF6sAdjw
```

NATS REQUESTS

Then it expresses limited interest in the topic:

```
SUB _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 2
UNSUB 2 1
```

tells the server to *unsubscribe from subscription with sid=2 after getting 1 message*

NATS REQUESTS

Then the request is published to a subject (`help`), tagging it with the ephemeral inbox just for the request to happen:

```
SUB _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 2
UNSUB 2 1
PUB help _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 6
please
```

NATS REQUESTS

Then *iff* there is another subscriber connected and interested in the `help` subject, it will receive a message with that inbox:

```
# Another client interested in the help subject
SUB help 90

# Receives from server a message
MSG help 90 _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 6
please

# Can use that inbox to reply back
PUB _INBOX.ioL1Ws5aZZf5fyeF6sAdjw  11
I can help!
```

NATS REQUESTS

Finally, *iff* the client which sent the request is still connected and interested, it will be receiving that message:

```
SUB _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 2
UNSUB 2 1
PUB help _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 6
please
MSG _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 2 11
I can help!
```

Simple Protocol == Simple Clients

Given the protocol is simple, NATS clients libraries tend to have a very small footprint as well.

CLIENTS

RUBY

```ruby
require 'nats/client'

NATS.start do |nc|
  nc.subscribe("hello") do |msg|
    puts "[Received] #{msg}"
  end

  nc.publish("hello", "world")
end
```

GO

```go
nc, err := nats.Connect()
// ...
nc.Subscribe("hello", func(m *nats.Msg){
        fmt.Printf("[Received] %s", m.Data)
})
nc.Publish("hello", []byte("world"))
```

MANY MORE AVAILABLE

C	C#	Java
Python	NGINX	Spring
Node.js	Elixir	Rust
Lua	Erlang	PHP
Haskell	Scala	Perl

Many thanks to the community!

ASYNCHRONOUS IO

Note: Most clients have asynchronous behavior

```
nc, err := nats.Connect()
// ...
nc.Subscribe("hello", func(m *nats.Msg){
        fmt.Printf("[Received] %s", m.Data)
})
for i := 0; i < 1000; i ++ {
        nc.Publish("hello", []byte("world"))
}
// No guarantees of having sent the bytes yet!
// They may still just be in the flushing queue.
```

ASYNCHRONOUS IO

In order to guarantee that the published messages have been processed by the server, we can do an **extra ping/pong** to confirm they were consumed:

```
nc.Subscribe("hello", func(m *nats.Msg){
        fmt.Printf("[Received] %s", m.Data)
})
for i := 0; i < 1000; i ++ {
        nc.Publish("hello", []byte("world"))
}
// Do a PING/PONG roundtrip with the server.
nc.Flush()
```

```
SUB hello 1\r\nPUB hello 5\r\nworld\r\n..PING\r\n
```

Then flush the buffer and wait for `PONG` from server

ASYNCHRONOUS IO

Worst way of measuring NATS performance

```go
nc, _ := nats.Connect(nats.DefaultURL)
msg := []byte("hi")
nc.Subscribe("hello", func(_ *nats.Msg) {})
for i := 0; i < 100000000; i++ {
        nc.Publish("hello", msg)
}
```

```
NATS @ ~ () $ go run /tmp/example.go


 0 bash
NATS @ ~ () $ gnatsd -m 8222
[45216] 2016/09/04 18:31:22.207437 [INF] Starting nats-server version 0.9.4
[45216] 2016/09/04 18:31:22.207651 [INF] Starting http monitor on 0.0.0.0:8222
[45216] 2016/09/04 18:31:22.208061 [INF] Listening for client connections on 0.0.0.0:4222
[45216] 2016/09/04 18:31:22.208096 [INF] Server is ready


 2 bash
NATS server version 0.9.4 (uptime: 31s)
Server:
  Load: CPU:  0.0%  Memory: 5.8M  Slow Consumers: 0
  In:   Msgs: 0  Bytes: 0  Msgs/Sec: 0.0  Bytes/Sec: 0
  Out:  Msgs: 0  Bytes: 0  Msgs/Sec: 0.0  Bytes/Sec: 0

Connections Polled: 0
  HOST             CID    SUBS    PENDING     MSGS_TO     MSGS_FROM    BYTES_TO     BYTES_FROM  LANG     VERSION  UPTIME   LAST ACTIVITY


 1 bash
```

The client is a *slow consumer* since it is not consuming the messages which the server is sending fast enough.

Whenever the server cannot flush bytes to a client fast enough, **it will disconnect the client** from the system as this consuming pace could affect the whole service and rest of the clients.

NATS Server is protecting itself

NATS = Performance + Simplicity + Resiliency

ALSO INCLUDED

- Subject routing with wildcards
  - Authorization
- Distribution queue groups for balancing
- Cluster mode for high availability
  - Auto discovery of topology
- Secure TLS connections with certificates
- `/varz` monitoring endpoint
  - used by nats-top

SUBJECTS ROUTING

Wildcards: `*`

```
SUB foo.*.bar 90
PUB foo.hello.bar 2
hi
MSG foo.hello.bar 90 2
hi
```

e.g. subscribe to all NATS requests being made on the demo site:

```
telnet demo.nats.io 4222
INFO {"auth_required":false,"version":"0.9.4",...}

SUB _INBOX.* 99
MSG _INBOX.ioL1Ws5aZZf5fyeF6sAdjw 99 11
I can help!
```

SUBJECTS ROUTING

Full wildcard: >

```
SUB hello.> 90
PUB hello.world.again 2
hi
MSG hello.world.again 90 2
hi
```

Subscribe to all subjects and see whole traffic going through the server:

```
telnet demo.nats.io 4222
INFO {"auth_required":false,"version":"0.9.4",...}
sub > 1
+OK
```

SUBJECTS AUTHORIZATION

Clients are not allowed to publish on `_SYS` for example:

```
PUB _SYS.foo 2
hi
-ERR 'Permissions Violation for Publish to "_SYS.foo"'
```

SUBJECTS AUTHORIZATION

Can customize disallowing pub/sub on certain subjects via server config too:

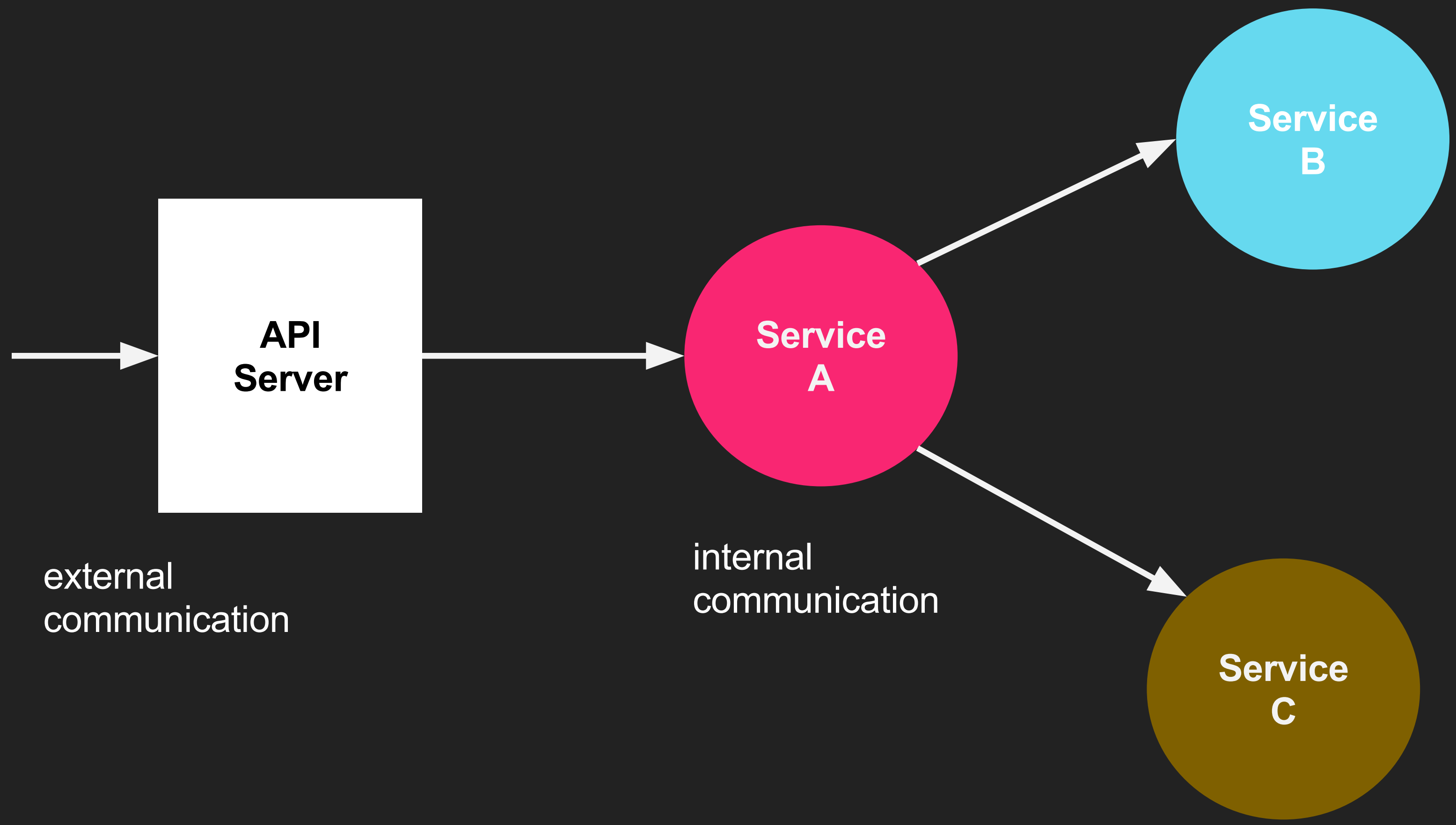
```
authorization {
 admin = { publish = ">", subscribe = ">" }
 requestor = {
  publish = ["req.foo", "req.bar"]
  subscribe = "_INBOX.*"
 }

 users = [
  {user: alice, password: foo, permissions: $admin}
  {user: bob,   password: bar, permissions: $requestor}
 ]
}
```

Building systems with NATS

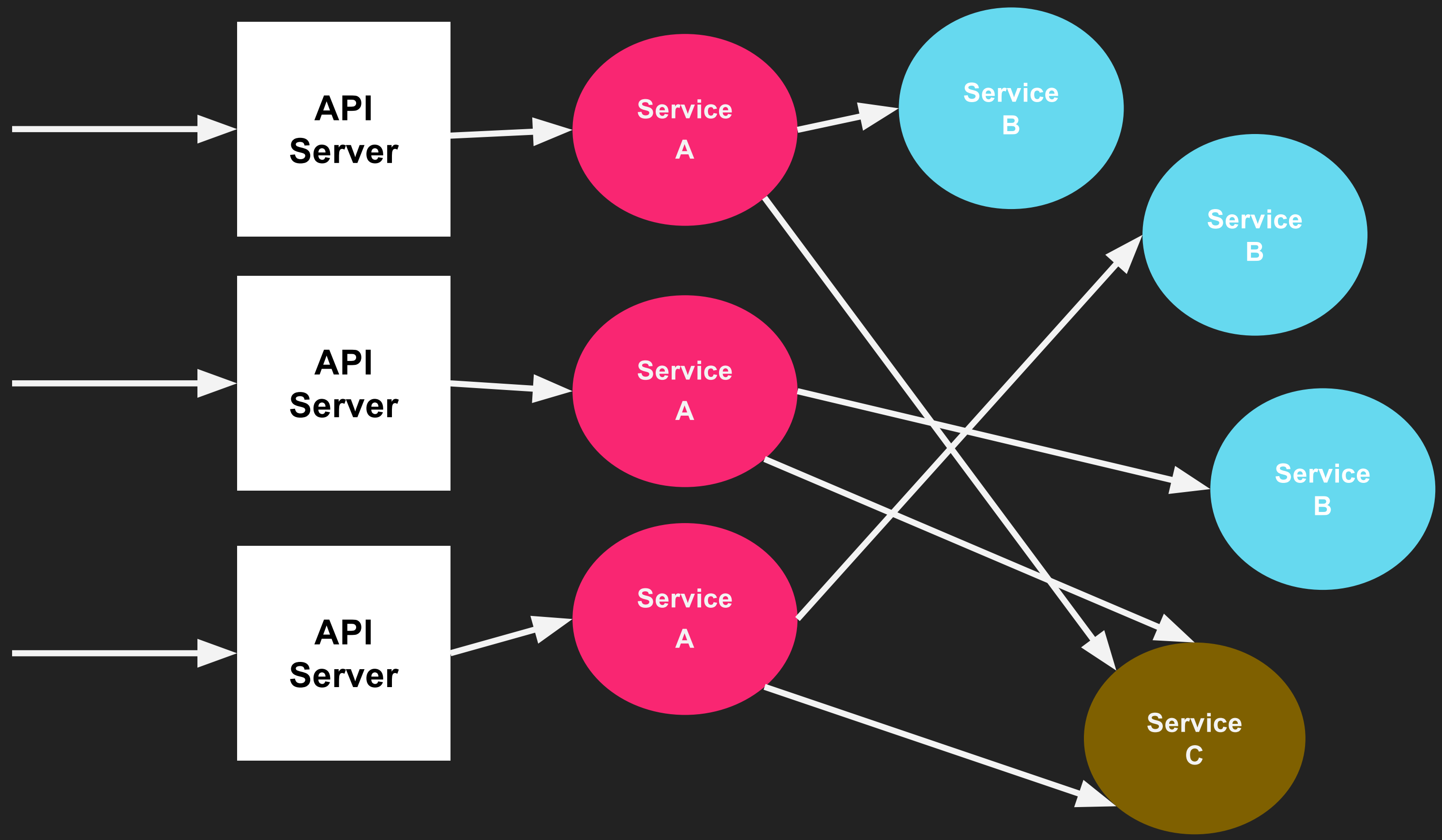
FAMILIAR SCENARIO

Service A needs to talk to services B and C



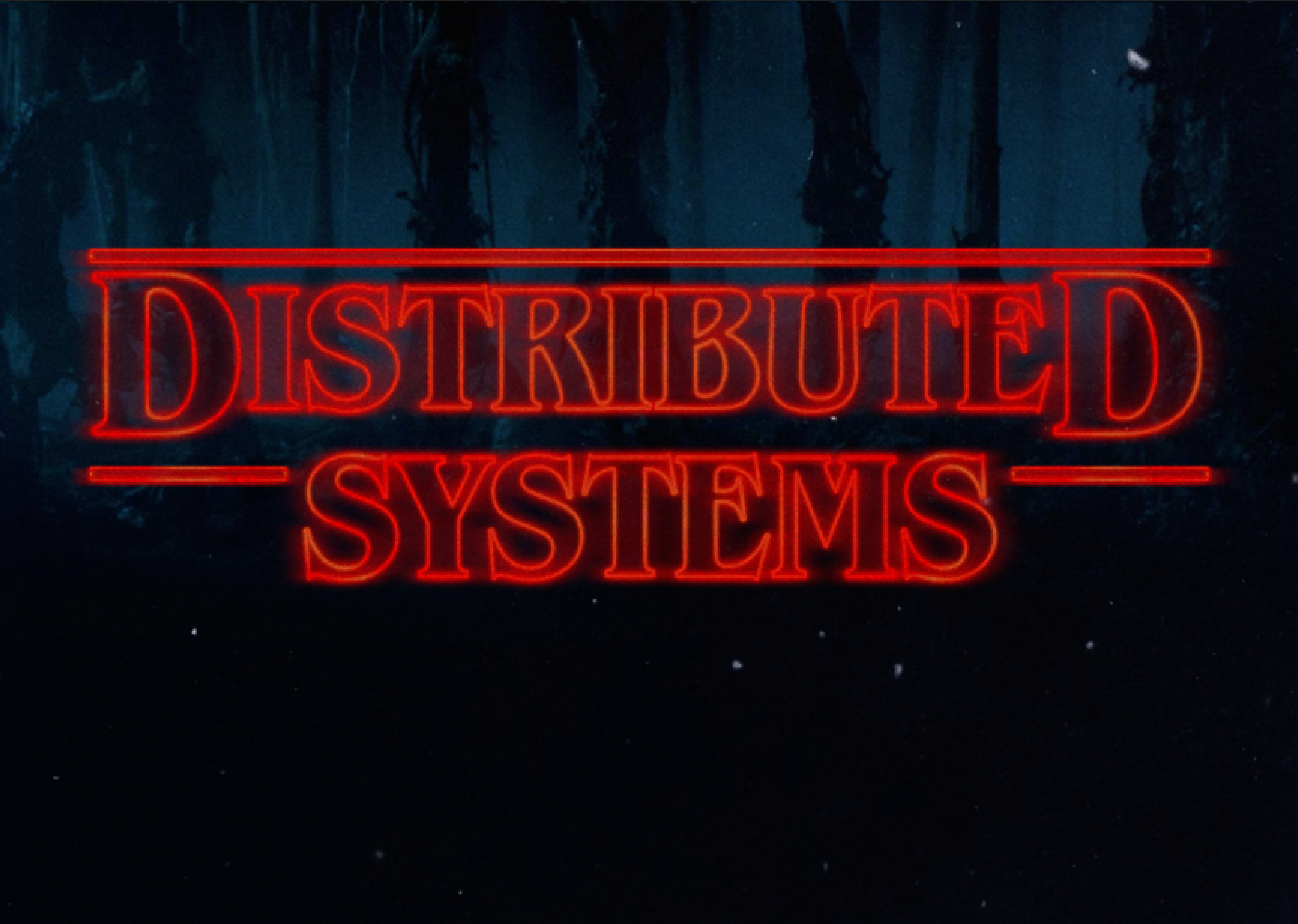
FAMILIAR SCENARIO

Horizontally scaled…



COMMUNICATING WITHIN A DISTRIBUTED SYSTEM

Just use HTTP everywhere?
Use some form of point to point RPC?
What about service discovery and load balancing?
What if sub ms latency performance is required?


DISTRIBUTED
SYSTEMS

WHAT NATS GIVES US

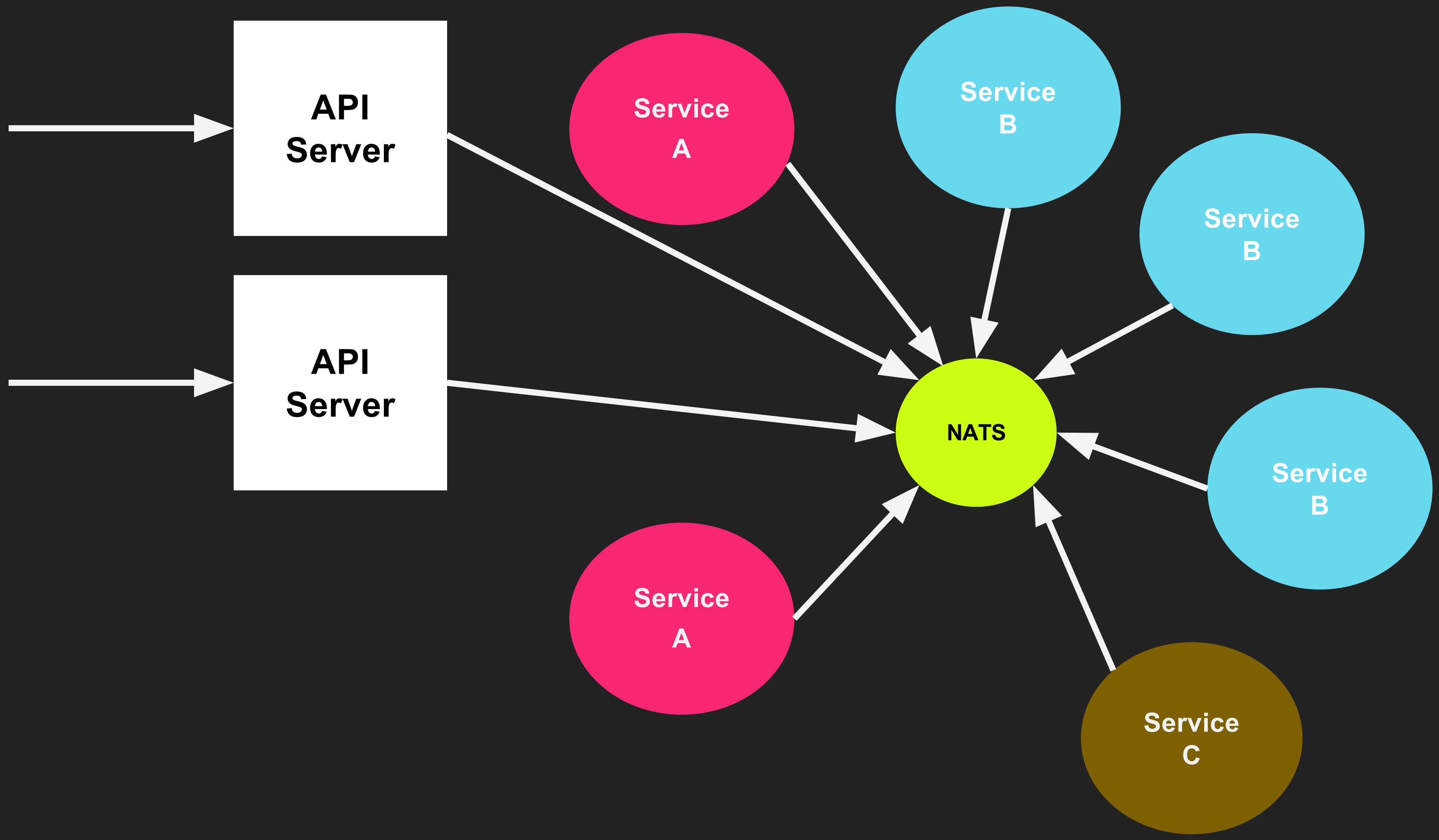
publish/subscribe based low latency mechanism for communicating with 1 to 1, 1 to N nodes

An established TCP connection to a server

A *dial tone*

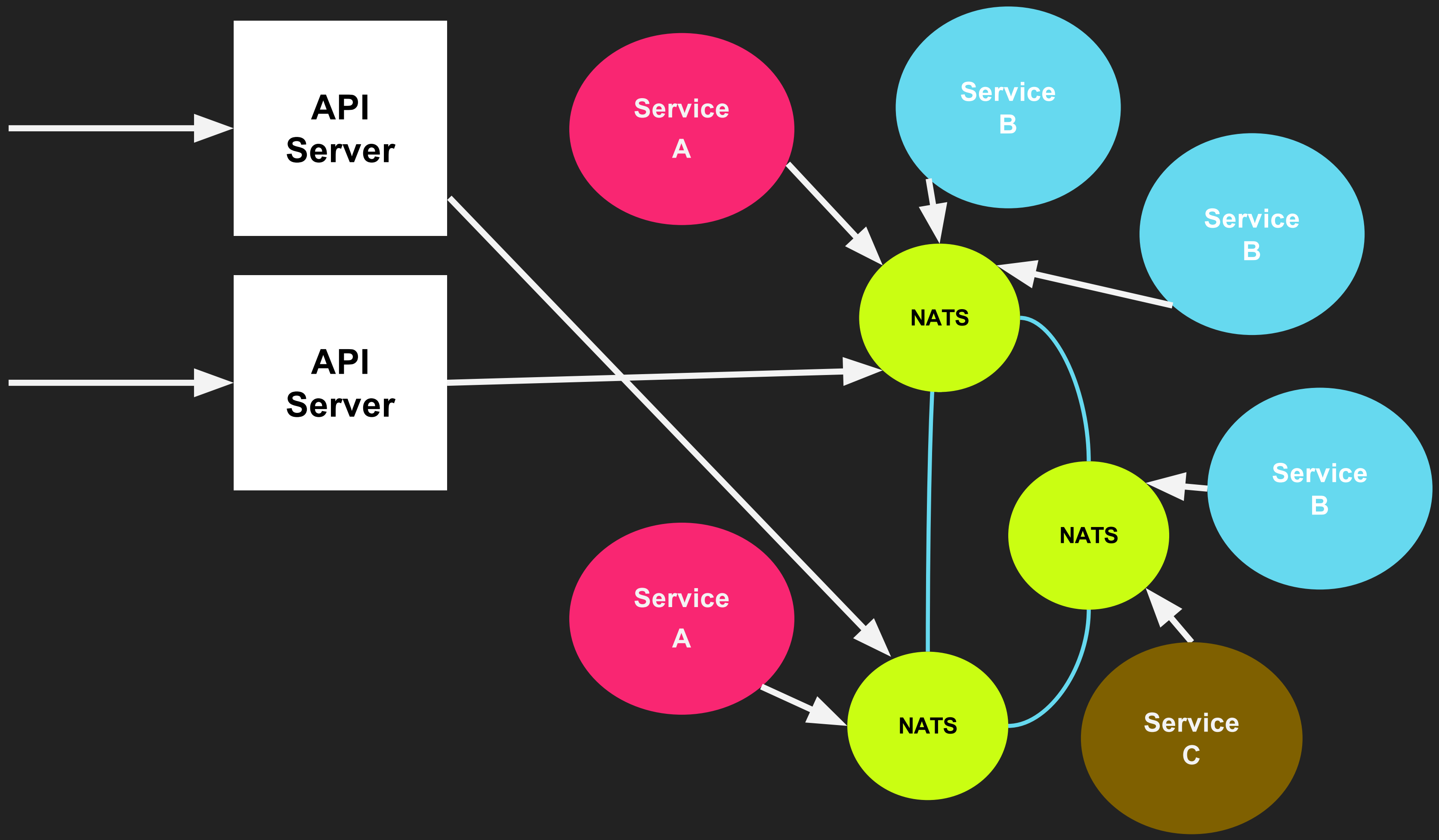
COMMUNICATING THROUGH NATS

Using NATS for internal communication



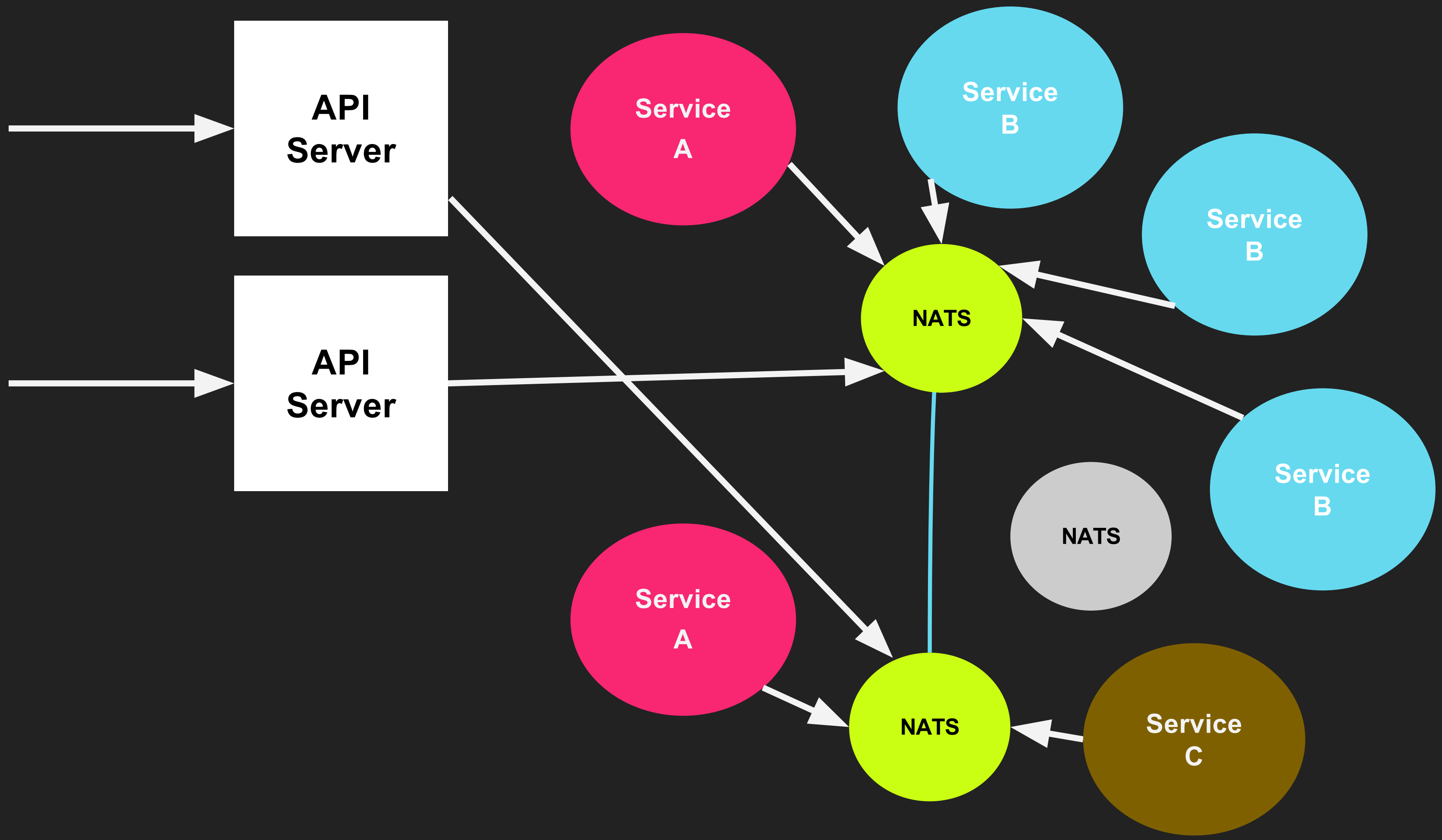
HA WITH NATS CLUSTER

Avoid *SPOF* on NATS by assembling a full mesh cluster



HA WITH NATS CLUSTER

Clients reconnect logic is triggered



HA WITH NATS CLUSTER

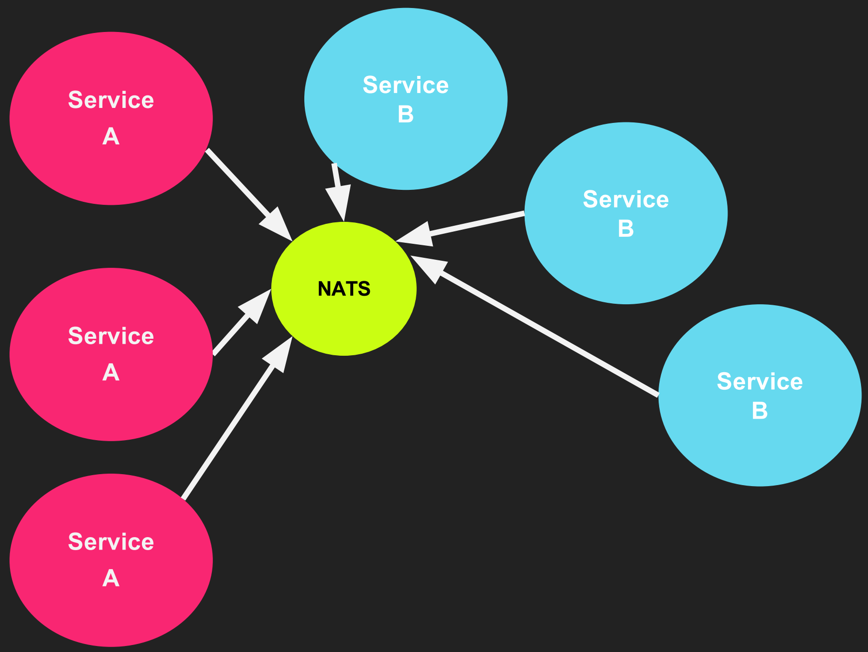
Connecting to a NATS cluster of 2 nodes explicitly

```
srvs := "nats://10.240.0.11:4222,nats://10.240.0.21:422
nc, _ := nats.Connect(srvs)
```

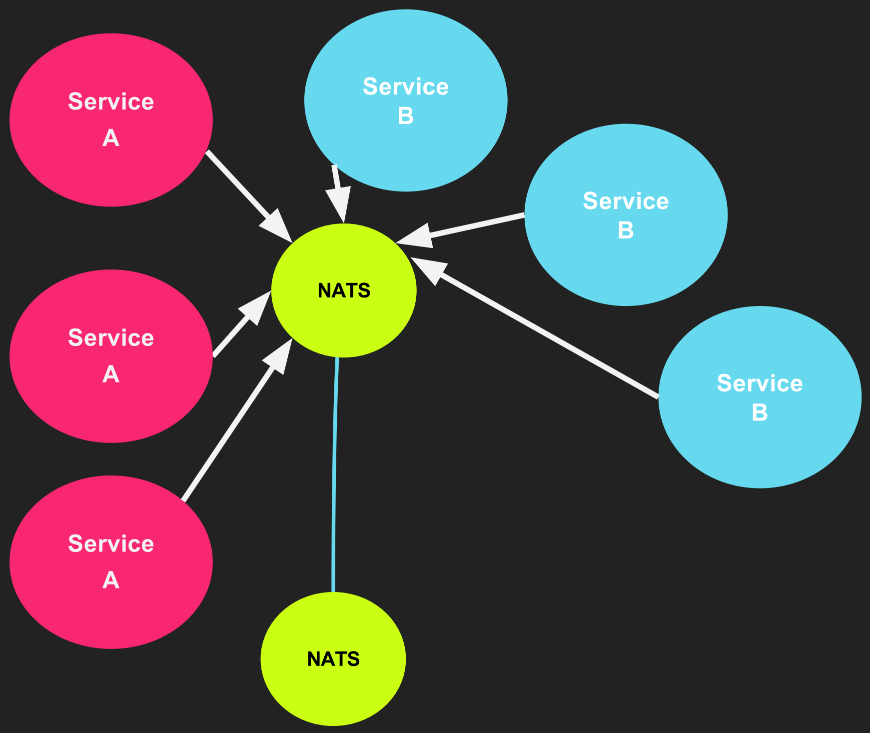
Bonus: Cluster topology can be discovered dynamically too!

CLUSTER AUTO DISCOVERY

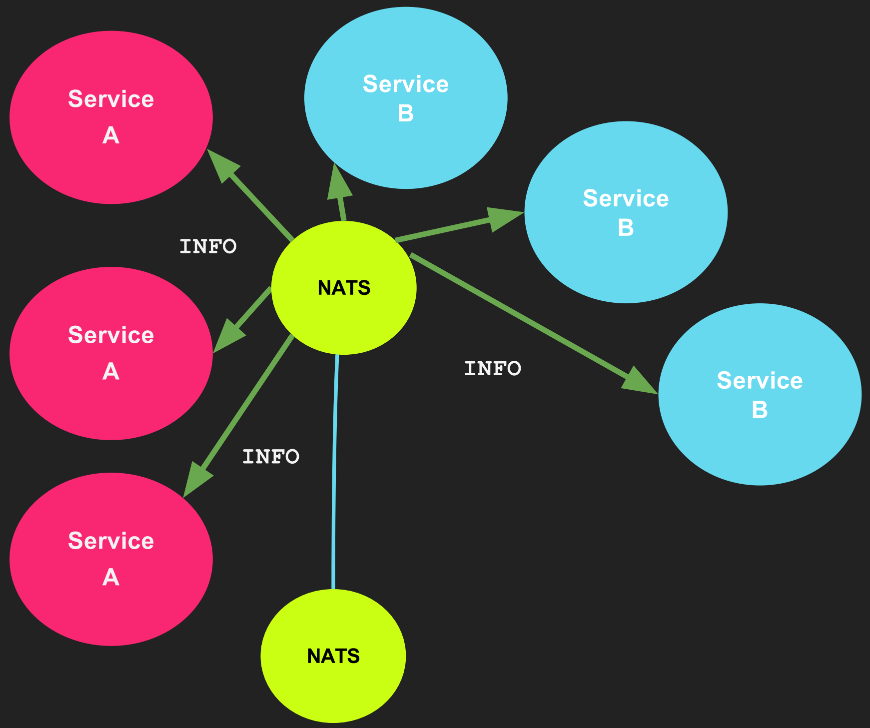
We can start with a single node…



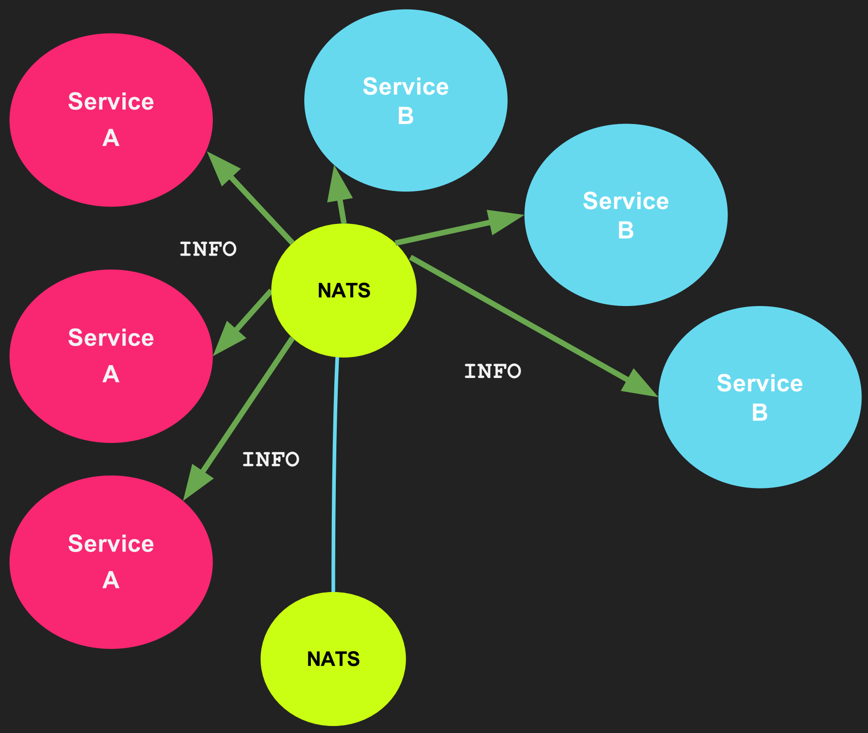
Then have new nodes join the cluster…



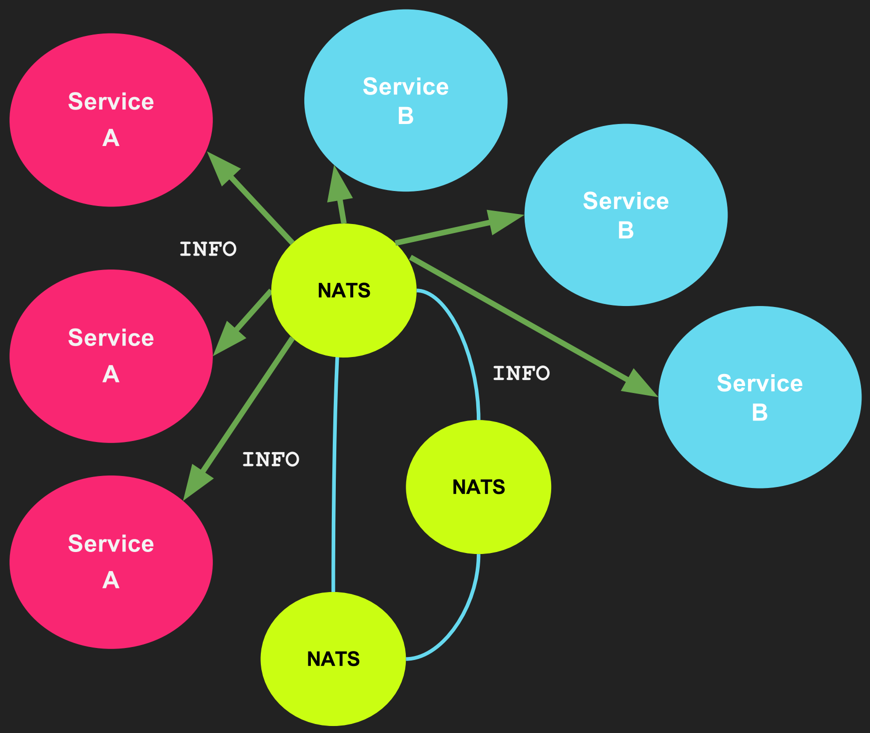
As new nodes join, server announces `INFO` to clients.



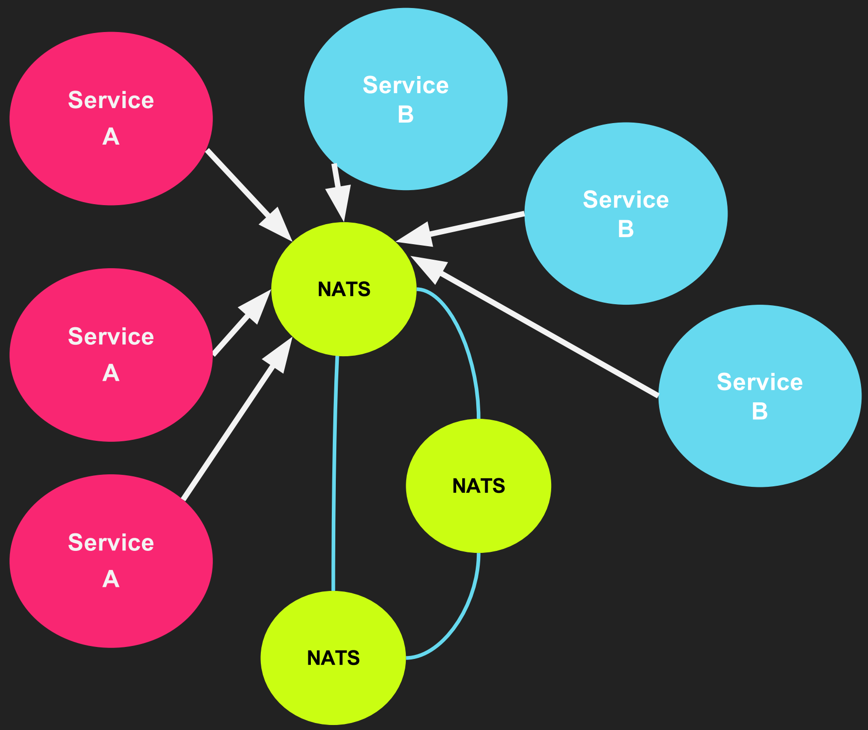
Clients auto reconfigure to be aware of new nodes.



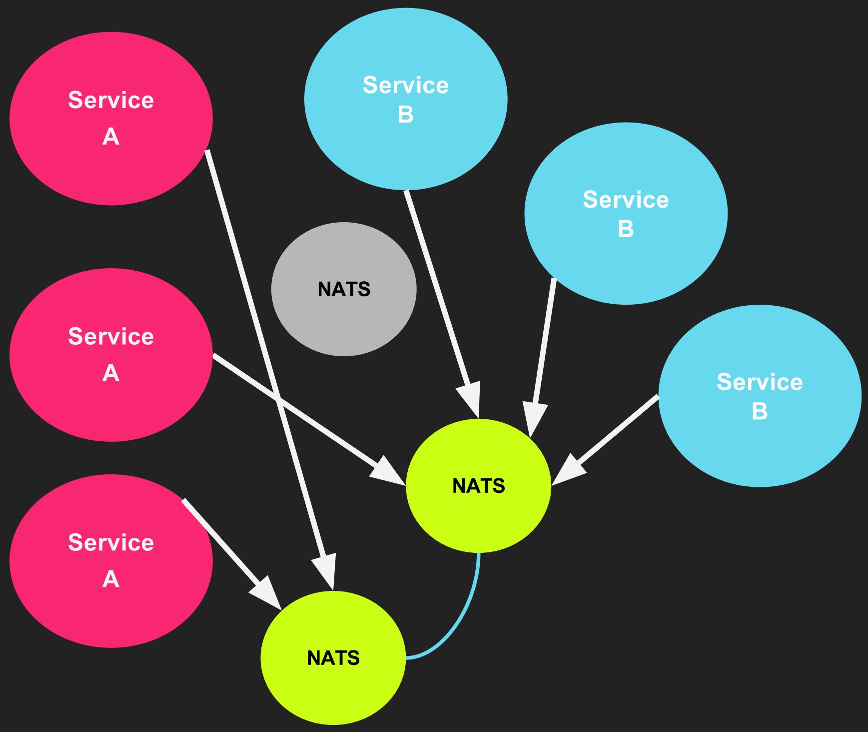
Clients auto reconfigure to be aware of new nodes.



Now fully connected!



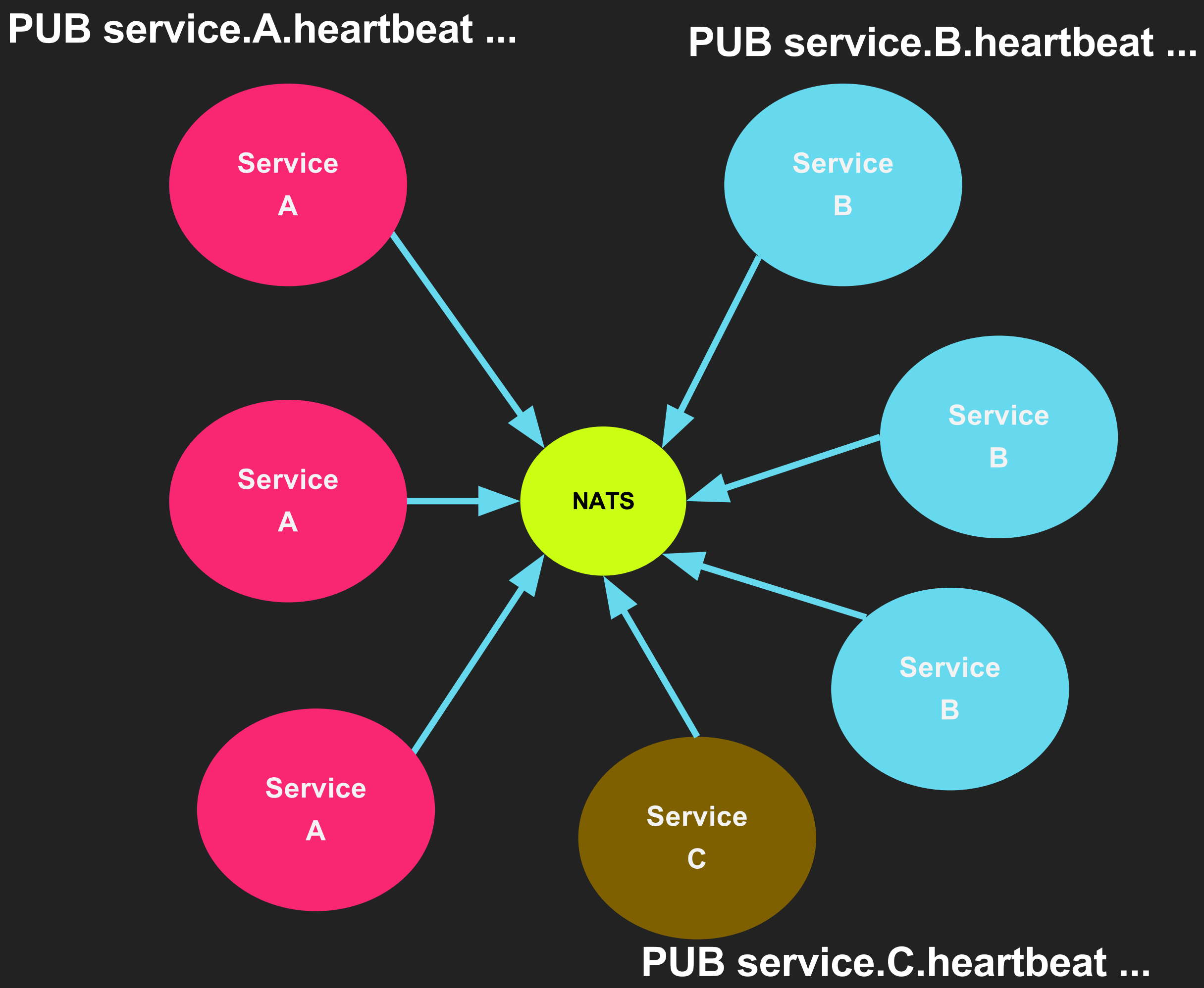
On failure, clients reconnect to an available node.



COMMUNICATING USING NATS

HEARTBEATS

For announcing liveness, services could publish heartbeats



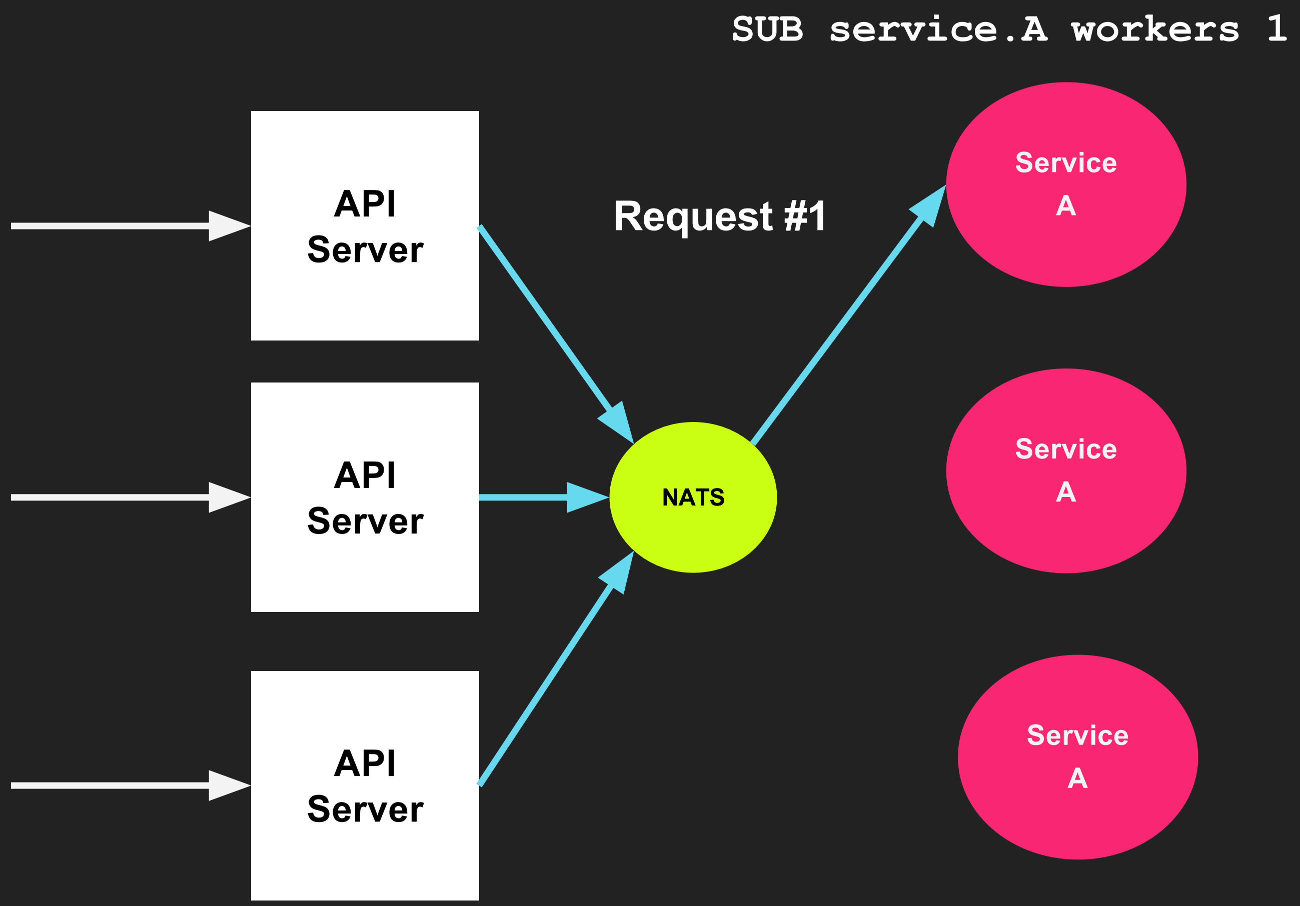
HEARTBEATS → DISCOVERY

Heartbeats can help too for discovering services via wildcard subscriptions.

```
nc, _ := nats.Connect(nats.DefaultURL)
// SUB service.*.heartbeats 1\r\n
nc.Subscribe("service.*.heartbeats", func(m *nats.Msg)
        // Heartbeat from service received
})
```

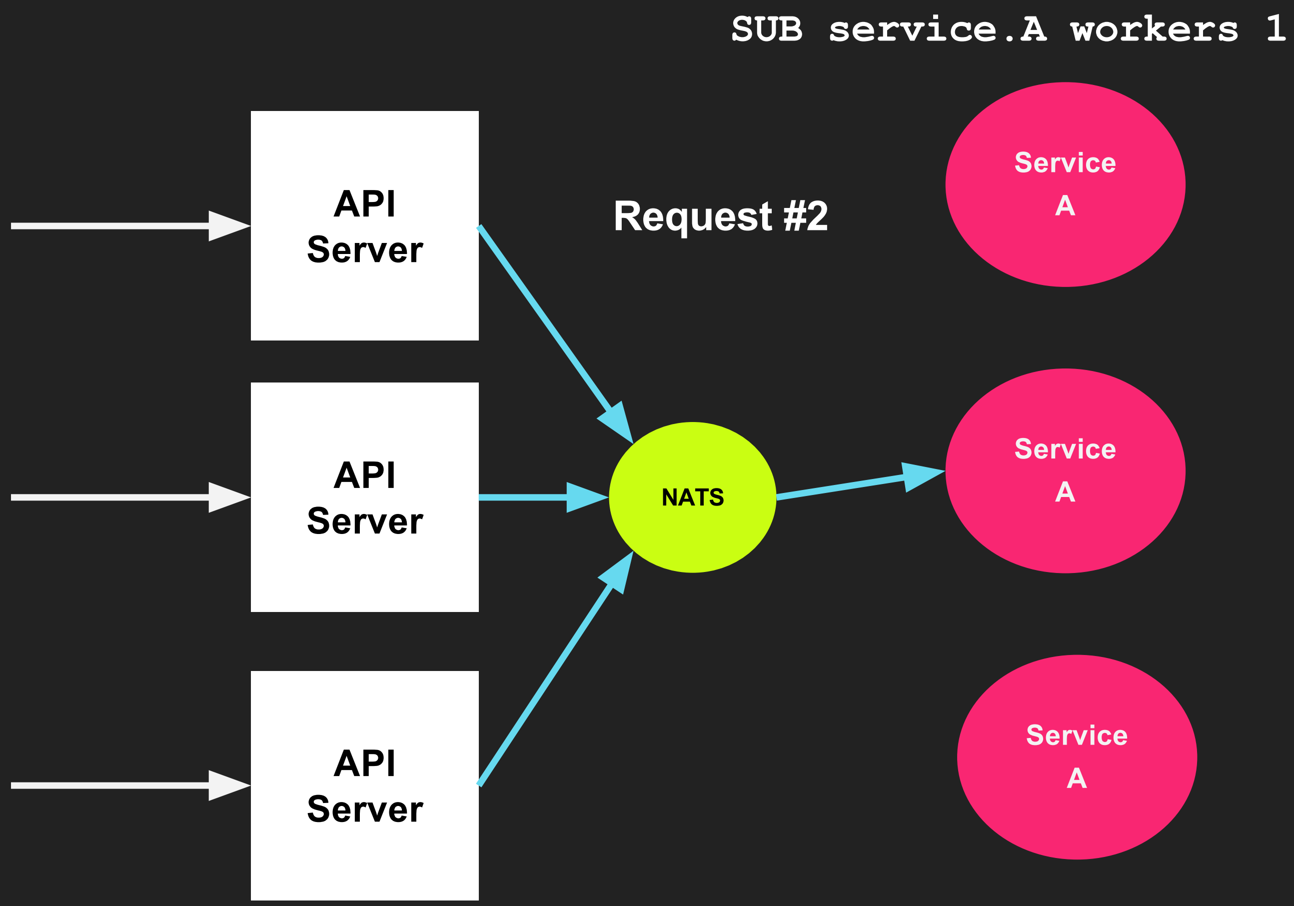
DISTRIBUTION QUEUES

Balance work among nodes randomly



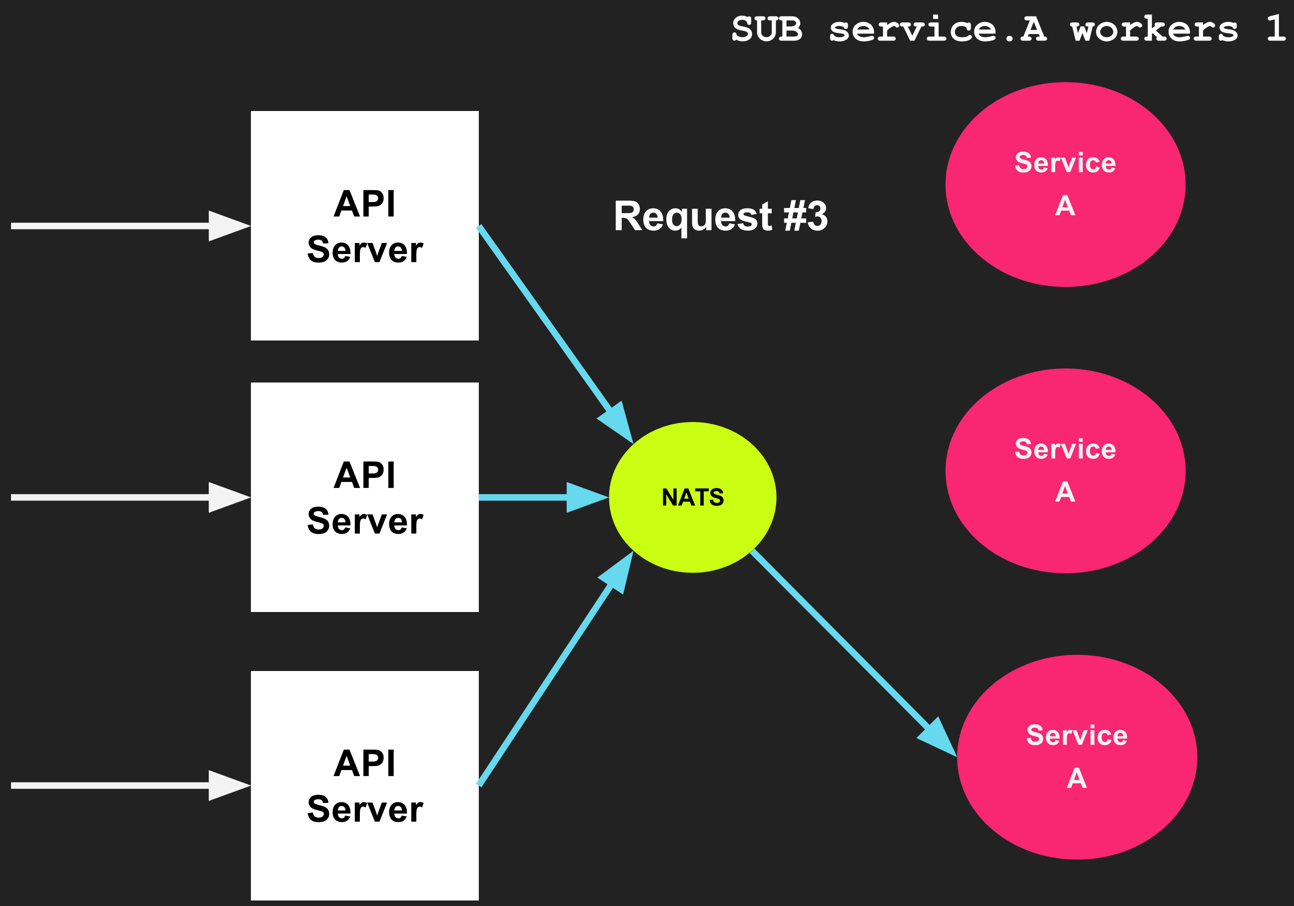
DISTRIBUTION QUEUES

Balance work among nodes randomly



DISTRIBUTION QUEUES

Balance work among nodes randomly



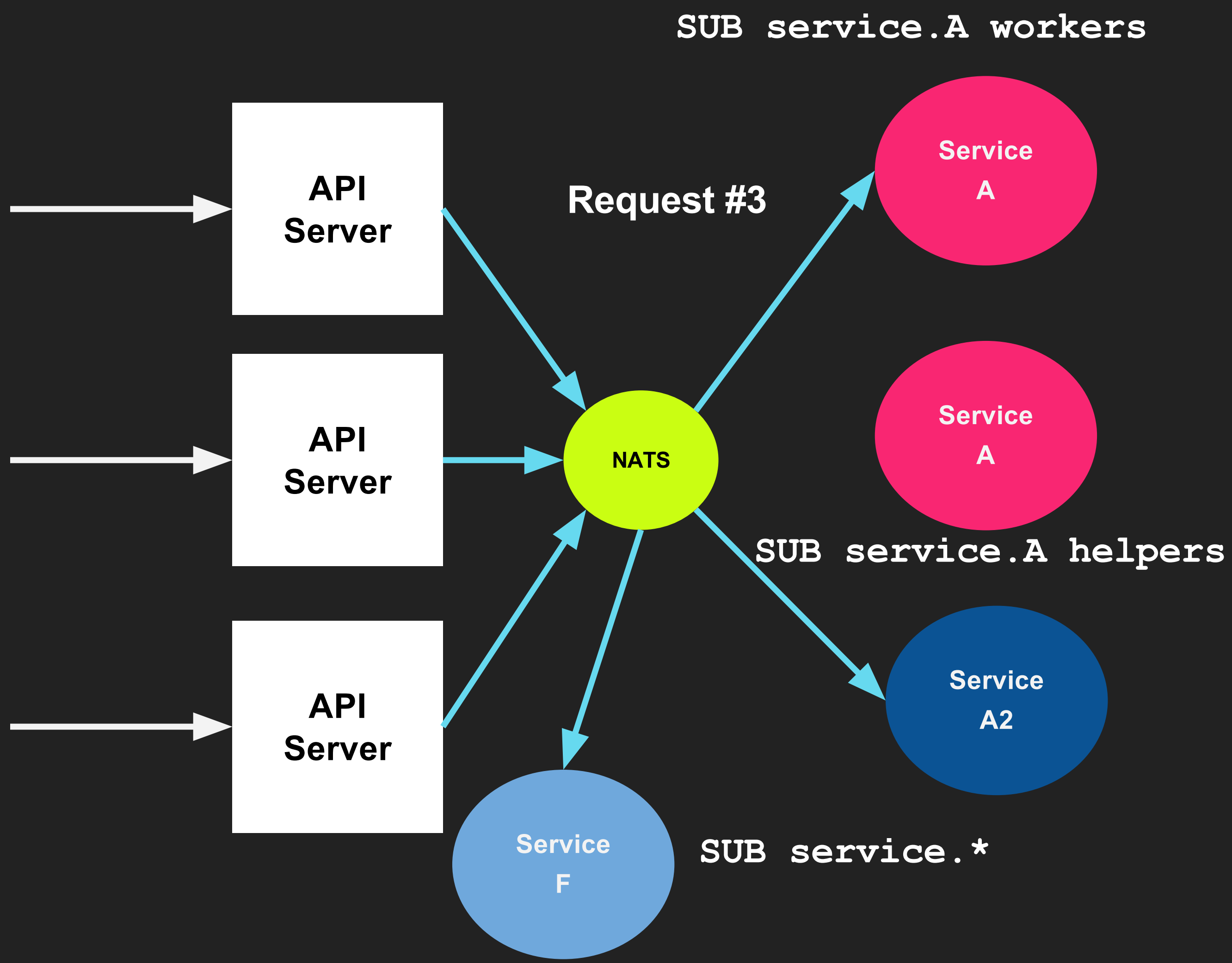
DISTRIBUTION QUEUES

Service A workers subscribe to `service.A` and create `workers` distribution queue group for balancing the work.

```
nc, _ := nats.Connect(nats.DefaultURL)
// SUB service.A workers 1\r\n
nc.QueueSubscribe("service.A", "workers",
        func(m *nats.Msg) {
                nc.Publish(m.Reply, []byte("hi!"))
        })
```

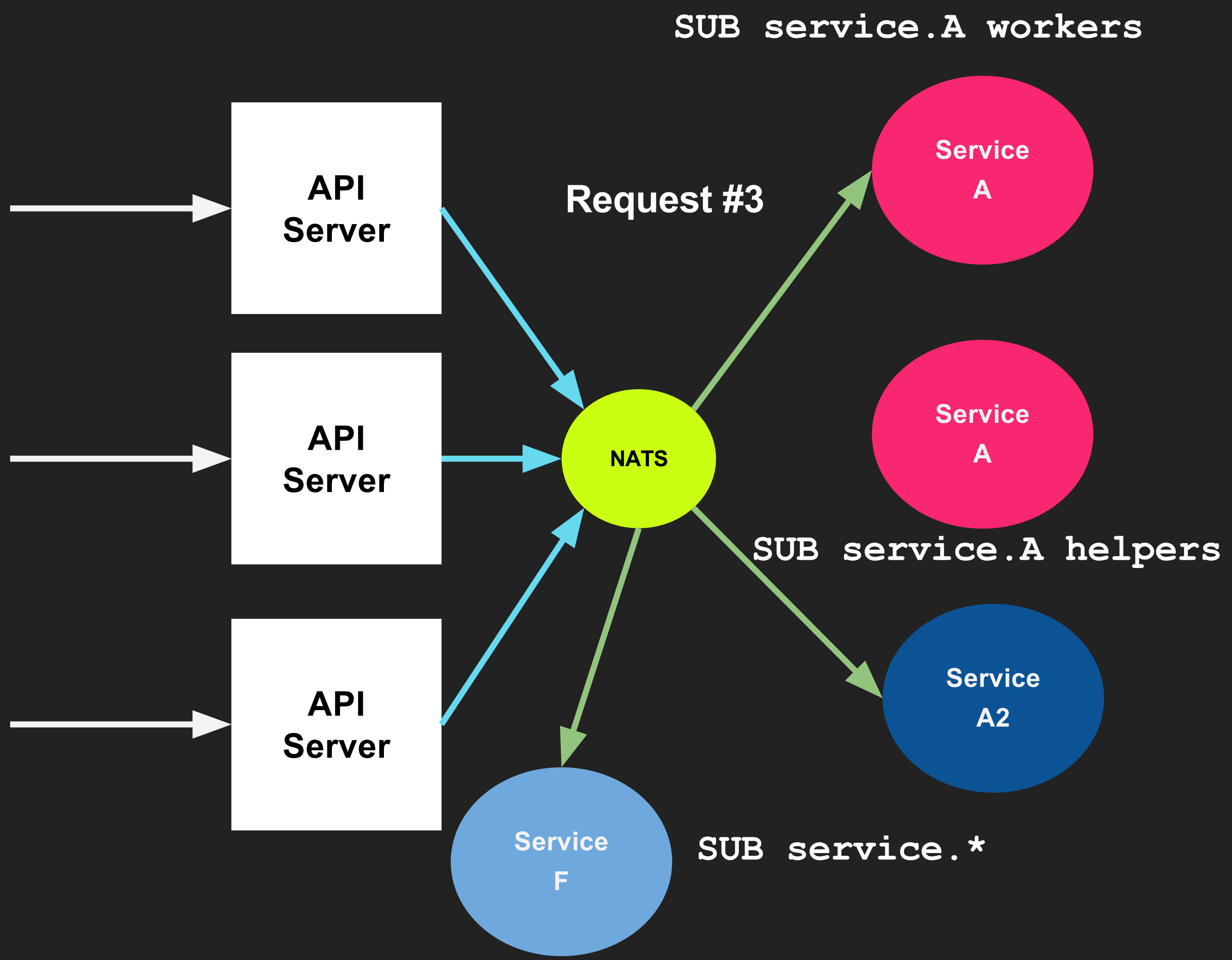
DISTRIBUTION QUEUES

Note: NATS does not assume the audience!



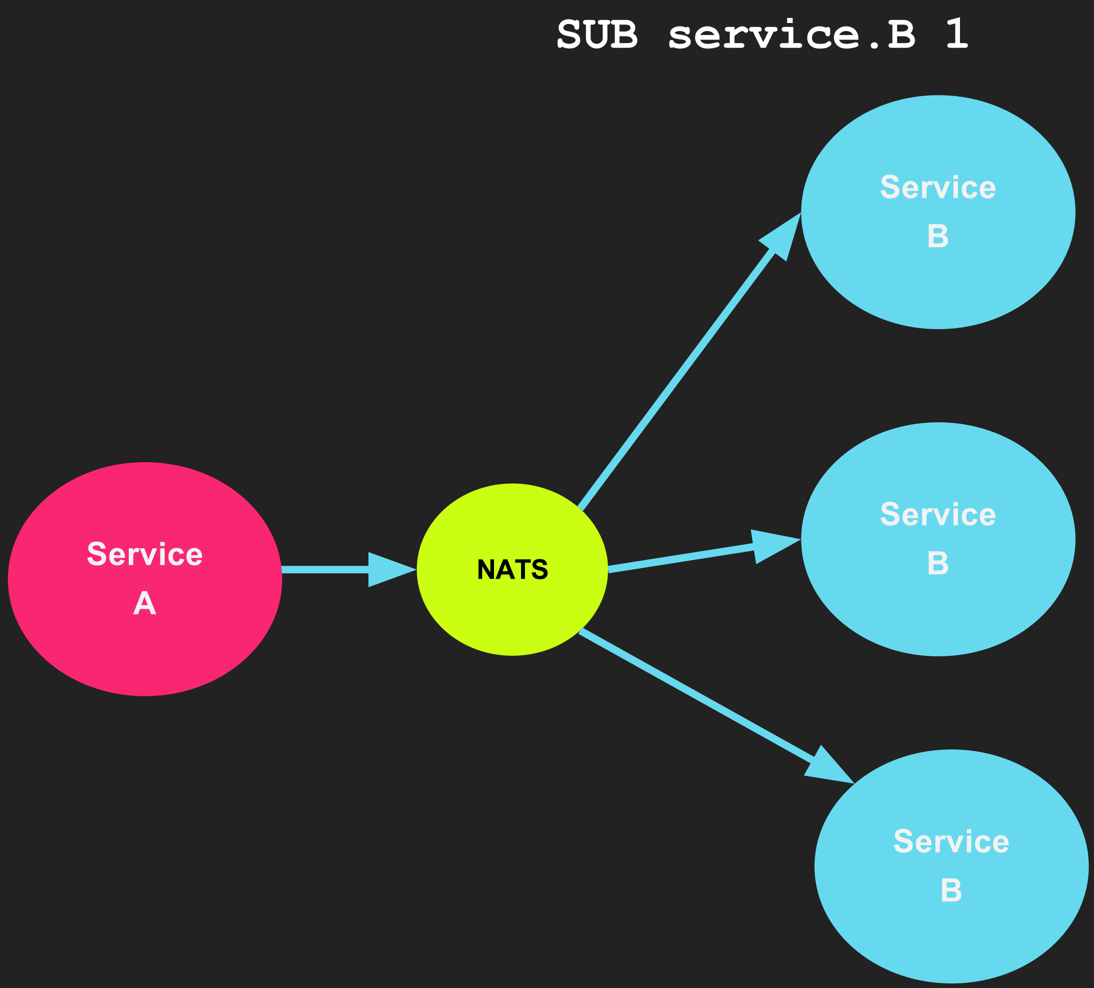
DISTRIBUTION QUEUES

All interested subscribers receive the message



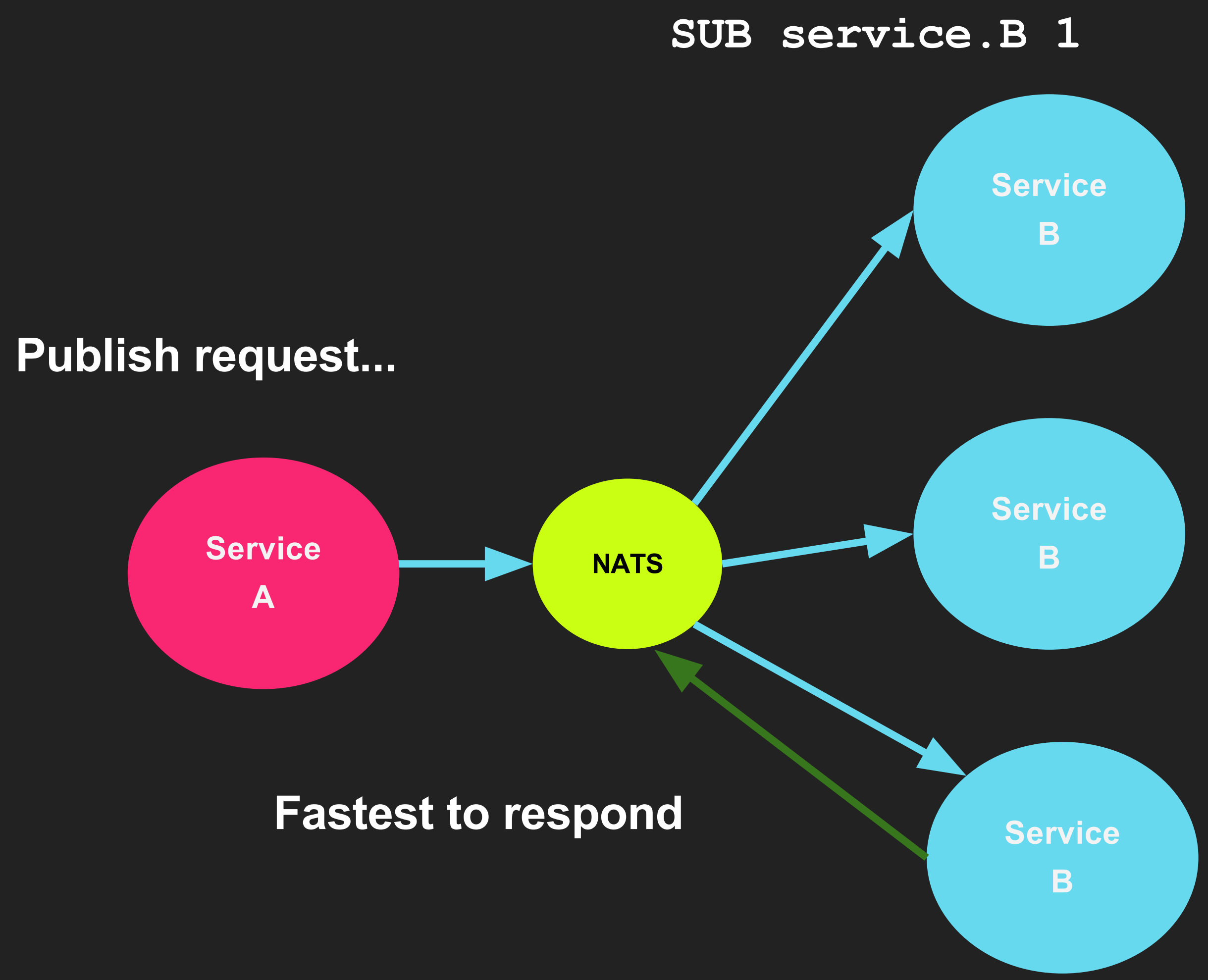
LOWEST LATENCY RESPONSE

Service A communicating with fastest node from Service B



LOWEST LATENCY RESPONSE

Service A communicating with fastest node from Service B



LOWEST LATENCY RESPONSE

NATS requests were designed **exactly** for this

```
nc, _ := nats.Connect(nats.DefaultURL)
t := 250*time.Millisecond
// Request sets to AutoUnsubscribe after 1 response
msg, err := nc.Request("service.B", []byte("help"), t)
if err == nil {
        fmt.Println(string(msg.Data))
        // => sure!
}
```

```
nc, _ := nats.Connect(nats.DefaultURL)

nc.Subscribe("service.B", func(m *nats.Msg) {
        nc.Publish(m.Reply, []byte("sure!"))
})
```

UNDERSTANDING NATS TIMEOUT

Note: Making a request involves establishing a client timeout.

```
t := 250*time.Millisecond
_, err := nc.Request("service.A", []byte("help"), t)
fmt.Println(err)
// => nats: timeout
```

This needs special handling!

UNDERSTANDING NATS TIMEOUT

NATS is *fire and forget*, reason for which a client times out could be many things:

- No one was connected at that time
  - *service unavailable*
- Service is actually still processing the request
  - *service took too long*
- Service was processing the request but crashed
  - *service error*

CONFIRM AVAILABILITY OF SERVICE NODE WITH REQUEST

- Each service node could have its own inbox

```
SUB _INBOX.123available 90
```

- A *request* is sent to `service.B` to get a single response, which will then reply with its own inbox, (no payload needed).
- If there is not a fast reply before client times out, then most likely the service is *unavailable* for us at that time.
- If there is a response, then use that *inbox* in a request

```
PUB _INBOX.123available _INBOX.456helpplease...
```

Summary

NATS is a **simple**, **fast** and **reliable** solution for the internal communication of a distributed system.

It chooses **simplicity** and **reliability** over **guaranteed delivery**.

Though this does not necessarily mean that guarantees of a system are constrained due to NATS!

→ https://en.wikipedia.org/wiki/End-to-end_principle

> *We can always build strong guarantees on top, but we can't always remove them from below.*
>
> - *Tyler Treat, Simple Solutions to Complex Problems*

Replayability can be better than **guaranteed delivery**

Idempotency can be better than **exactly once delivery**

Commutativity can be better than **ordered delivery**

Related NATS project: NATS Streaming

REFERENCES

High Performance Systems in Go **(Gophercon 2014)**

Derek Collison (link)

Dissecting Message Queues (2014)

Tyler Treat (link)

Simplicity Matters **(RailsConf 2012)**

Rich Hickey (link)

Simple Solutions for Complex Problems **(2016)**

Tyler Treat (link)

End To End Argument **(1984)**

J.H. Saltzer, D.P. Reed and D.D. Clark (link)

THANKS!

github.com/nats-io / @nats_io

Play with the demo site!

```
telnet demo.nats.io 4222
```