Micro on NATS - microservices with messaging

11 APR 2016

In this post we're going to discuss using **Micro** on **NATS**. It includes discussion around service discovery, synchronous and asynchronous communication for microservices.

If you would like to learn more about Micro first, check out the blog post detailing the toolkit here.

Let's get down to business.

What is NATS?

NATS is an open source cloud native messaging system or more simply a message bus. NATS was created by Derek Collison, the founder of Apcera. It originated within VMWare and began life as a ruby based system. It's long since been rewritten in Go and is steadily gaining adoption amongst those looking for a highly scalable and performant messaging system.

If you want to learn more about NATS itself, visit nats.io or join the community here.

Why NATS?

Why not NATS? Having worked with many message buses in the past, it was very quickly clear that NATS stood apart. Over the years messaging has been hailed as the saviour of the enterprise, resulting in systems that attempted to be everything to everyone. It led to a lot of false promises, significant feature bloat and high cost technologies that created more problems than they solved.

NATS, in contrast, takes the approach of being very focused, solving the problems of performance and availability while staying incredibly lean. It speaks of being “always on and available”, and using a “fire and forget” messaging pattern. It’s simplicity, focus and lightweight characteristics make it a prime candidate for the microservices ecosystem. We believe it will soon become the primary candidate as a transport for communication between services where messaging is concerned.

What NATS provides:

- High performance and scalability
- High availability
- Extremely lightweight
- At most once delivery

What NATS does not provide:

- Persistence
- Transactions
- Enhanced delivery modes
- Enterprise queueing

That briefly covers the what’s and why’s of NATS. So how does it fit in with Micro? Let’s discuss.

Micro on NATS

Micro is a microservice toolkit built with a pluggable architecture allowing the underlying dependencies to be swapped out with minimal changes. Each interface of the **Go-Micro** framework provides a building block for microservices; the registry for service discovery, transport for synchronous communication, broker for asynchronous messaging, etc.

Creating a plugin for each component is as simple as implementing the interface. We’ll spend more time detailing how to write plugins in a future blog post. If you want to

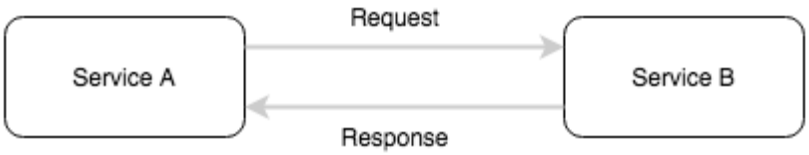
check out the plugins for NATS or any other systems such as etcd discovery, kafka broker, rabbitmq transport, you can find them here github.com/micro/go-plugins.

Micro on NATS is essentially a set of go-micro plugins that can be used to integrate with the NATS messaging system. By providing plugins for the various interfaces of go-micro, we create a number of integration points which allow a choice of architecture patterns.

In our experience one size does not fit all and the flexibility of Micro on NATS allows you to define the model that works for you and your team.

Below we'll discuss implementations of the NATS plugins for the transport, broker and registry.

Transport



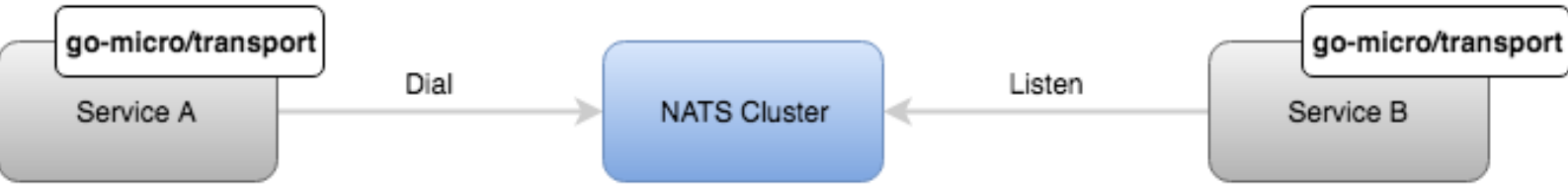
The transport is the go-micro interface for synchronous communication. It uses fairly common Socket semantics similar to other Go code with `Listen`, `Dial` and `Accept`. These concepts and patterns are well understood for synchronous communication using tcp, http, etc but it can be somewhat more difficult to adapt to a message bus. A connection is established with the message bus rather than with a service itself. To get around this we use the notion of a pseudo connection with topics and channels.

Here's how it works.

A service uses `transport.Listen` to listen for messages. This will create a connection to NATS. When `transport.Accept` is called, a unique topic is created and subscribed to. This unique topic will be used as the service address in the go-micro registry. Every message received will then be used as the basis for a pseudo socket/connection. If an existing connection exists with the same reply address we'll simply put the message in the backlog for that connection.

A client which wants to communicate with this service will use `transport.Dial` to create a connection to the service. This will connect to NATS, create it's own unique topic and subscribe to it. The topic is used for responses from the service. Anytime a message is sent by the client to the service, it will set the reply address to this topic.

When either side wants to close the connection, they simply call `transport.Close` which will terminate the connection to NATS.



Using the transport plugin

Import the transport plugin

```
import _ "github.com/micro/go-plugins/transport/nats"
```

Start with the transport flag

```
go run main.go --transport=nats --
transport_address=127.0.0.1:4222
```

Alternatively use the transport directly

```
transport := nats.NewTransport()
```

...

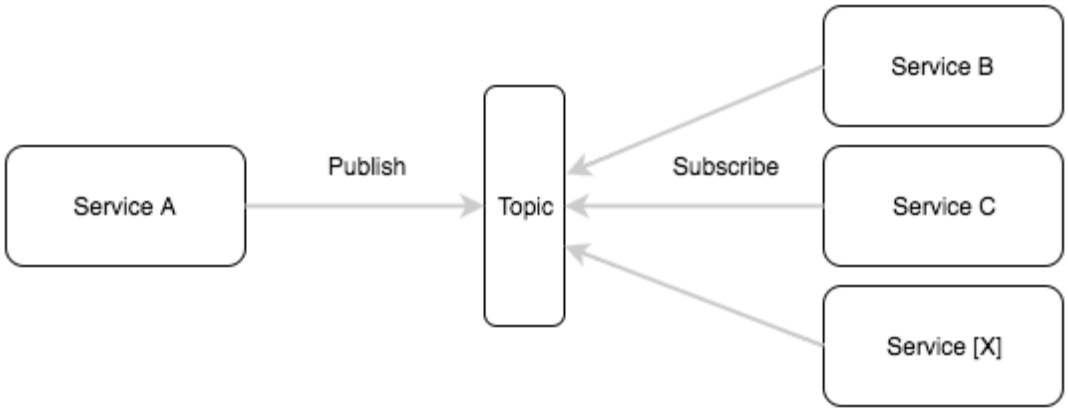
The go-micro transport interface:

```
type Transport interface {
    Dial(addr string, opts ...DialOption) (Client, error)
    Listen(addr string, opts ...ListenOption) (Listener, er-
```

```
ror)
    String() string
}
```

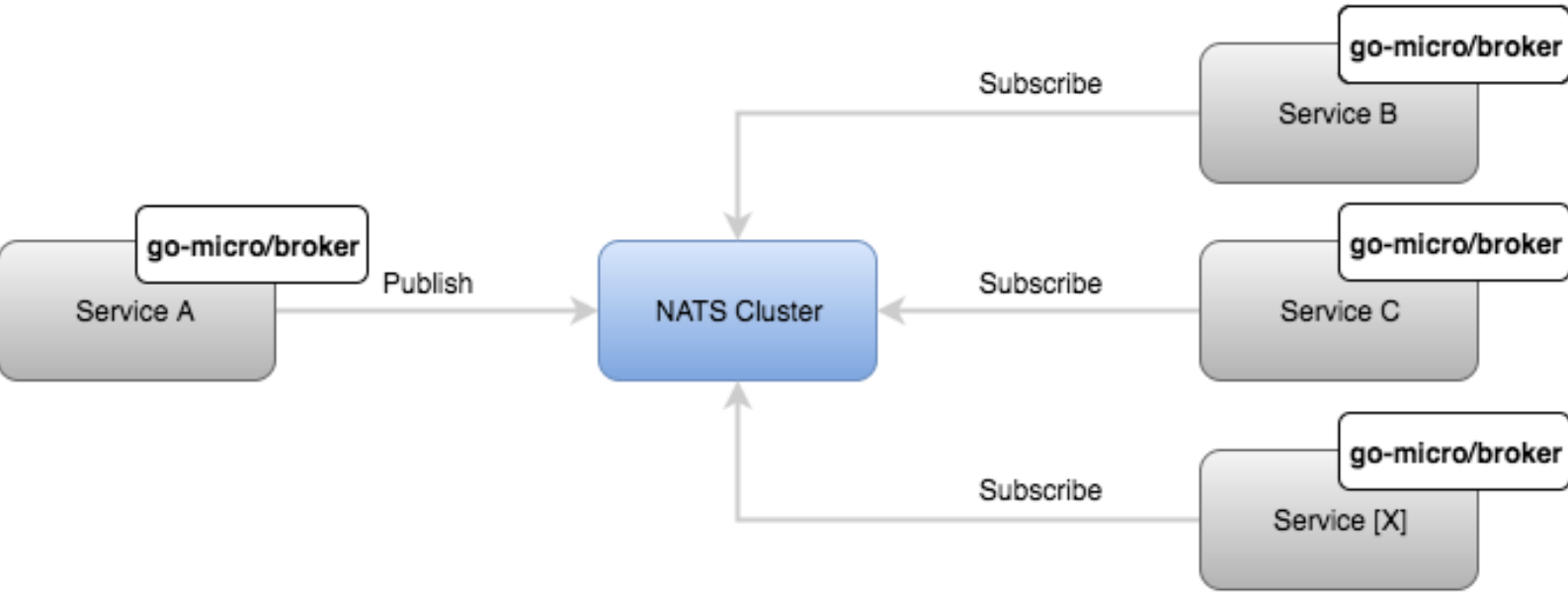
Transport

Broker



The broker is the go-micro interface for asynchronous messaging. It provides a high level generic implementation that applies across most message brokers. NATS, by its very nature, is an asynchronous messaging system, it's made for use as a message broker. There's only one caveat, NATS does not persist messages. While this may not be ideal for some, we still believe NATS can and should be used as a broker with go-micro. Where persistence is not required, it allows for a highly scalable pub sub architecture.

NATS provides a very straight forward Publish and Subscribe mechanism with the concepts of Topics, Channels, etc. There was no real fancy work required here to make it work. Messages can be published in an asynchronous fire and forget manner. Subscribers using the same channel name form a Queue Group in NATS which will then allow messages to automatically be evenly distributed across the subscribers.



Using the broker plugin

Import the broker plugin

```
import _ "github.com/micro/go-plugins/broker/nats"
```

Start with the broker flag

```
go run main.go --broker=nats --broker_address=127.0.0.1:4222
```

Alternatively use the broker directly

```
broker := nats.NewBroker()
```

...

The go-micro broker interface:

```
type Broker interface {
    Options() Options
    Address() string
    Connect() error
    Disconnect() error
```

```
    Init(...Option) error
    Publish(string, *Message, ...PublishOption) error
    Subscribe(string, Handler, ...SubscribeOption) (Sub-
scriber, error)
    String() string
}
```

Broker

Registry

The registry is the go-micro interface for service discovery. You might be thinking. Service discovery using a message bus? Does that even work? Indeed it does and rather well. Many people using a message bus for their transport will avoid using any kind of separate discovery mechanism. This is because the message bus itself can handle routing via topics and channels. Topics defined as service names can be used as the routing key, automatically load balancing between instances of a service that subscribe to the topic.

Go-micro treats the service discovery and transport mechanisms as two separate concerns. Anytime a client makes a request to another service, beneath the covers, it looks up the service in the registry by name, picks the address of a node and then communicates with it via the transport.

Usually the most common way of storing service discovery information is via a distributed key-value store like zookeeper, etcd or something similar. As you may have realised, NATS is not a distributed key-value store, so we're going to do something a little different…

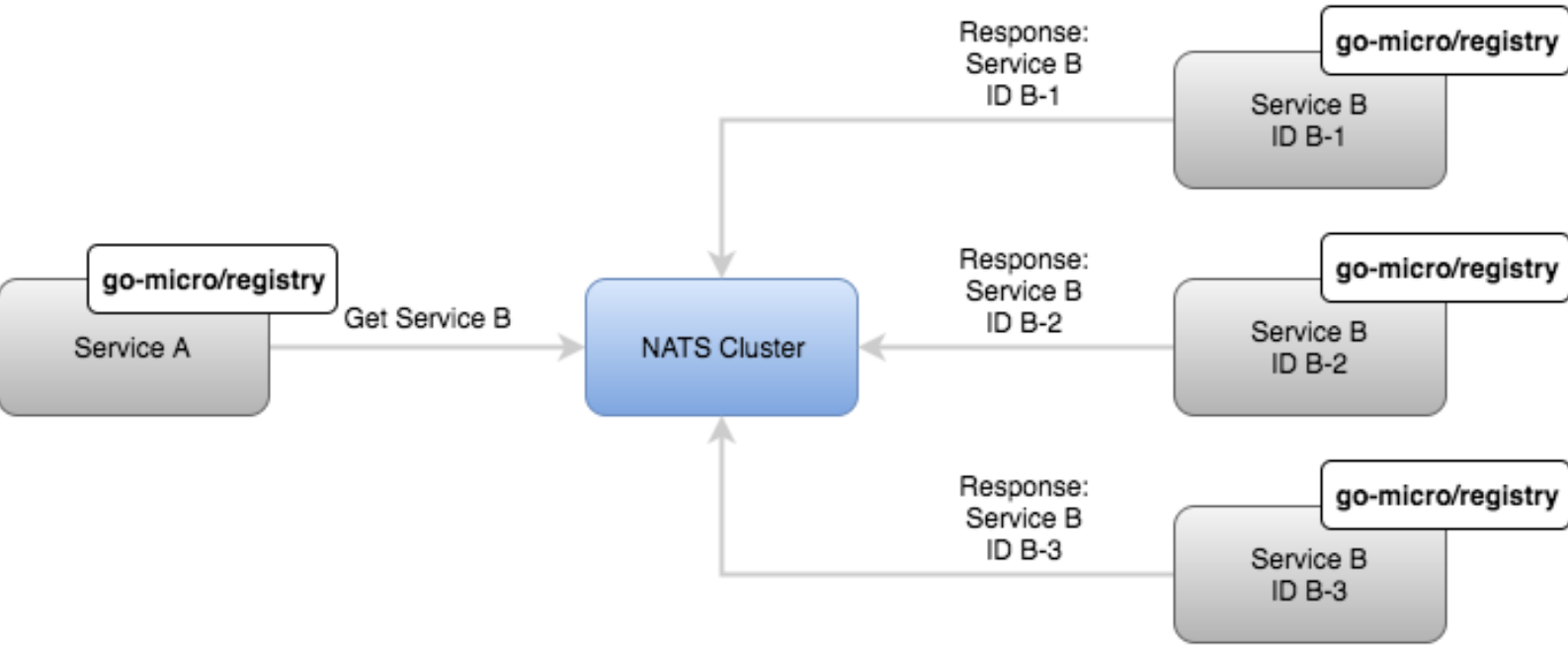
Broadcast queries!

Broadcast queries are just as you may imagine. Services listen on a particular topic we deem for broadcasting queries. Anyone who wants service discovery information will first create a reply topic which it subscribes to, then make its query on the broadcast topic with their reply address.

Because we don't actually know how many instances of a service are running or how many responses will be returned, we set an upper bound on the time we're willing to wait for a response. It's a crude mechanism of scatter gather for discovery but because of the scalable and performant nature of NATs, it actually works incredibly well. It also indirectly provides a very simple way of filtering services with higher response times. In the future we'll look to improve the underlying implementation.

So to sum up how it works:

1. Create reply topic and subscribe
2. Send query on broadcast topic with reply address
3. Listen for responses and unsubscribe after a time limit
4. Aggregate response and return result



Using the registry plugin

Import the registry plugin

```
import _ "github.com/micro/go-plugins/registry/nats"
```

Start with the registry flag

```
go run main.go --registry=nats --
registry_address=127.0.0.1:4222
```

Alternatively use the registry directly

```
registry := nats.NewRegistry()
```

...

The go-micro registry interface:

```
type Registry interface {
    Register(*Service, ...RegisterOption) error
    Deregister(*Service) error
    GetService(string) ([]*Service, error)
    ListServices() ([]*Service, error)
    Watch() (Watcher, error)
    String() string
}
```

 Registry

Scaling Micro on NATS

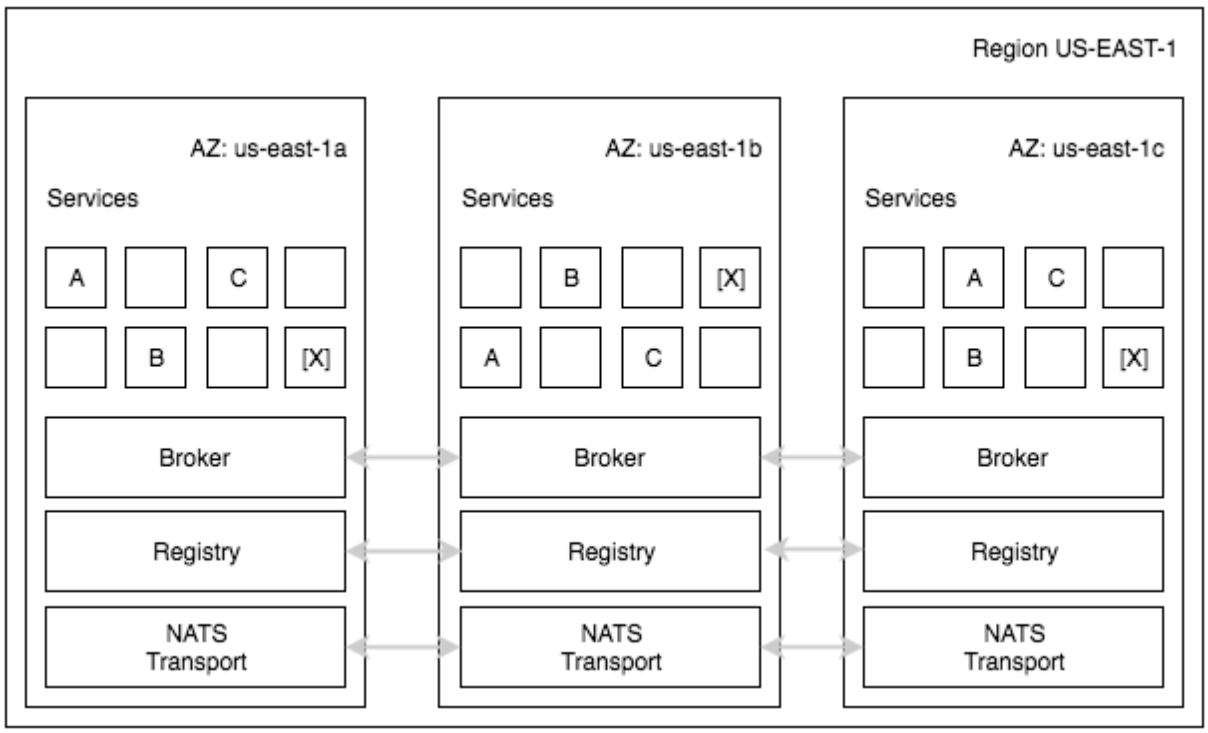
In the examples above we're only specifying a single NATS server on localhost but our recommended practice for real world use is to setup a NATS cluster for highly availability and fault tolerance. To learn more about NATs clustering checkout the NATS documentation here.

Micro accepts a comma separated list of addresses as the flags mentioned above or optionally the use of environment variables. If you're using the client libraries directly it

also allows a variadic set of hosts as an option for initialisation of the registry, transport and broker.

In terms of architecture in a cloud native world, our past experiences suggest that clusters per AZ or per region are ideal. Most cloud providers have relatively low (3-5ms) latency between AZs which allows for regional clustering without issue. When running a highly available configuration, it's important to ensure that you're system is capable of tolerating an AZ failure and in more mature configurations an entire region failure. We do not recommend clustering across regions. Ideally higher level tools should be used to manage multi-cluster and multi-region systems.

Micro is an incredibly flexible runtime agnostic microservices system. It's designed to run anywhere and in any configuration. It's view of the world is guided by the service registry. Clusters of services can be localised and namespaced within a pool of machines, AZs or regions based entirely on which registry you provide the service access to. In combination with NATS clustering it allows you to build a highly available architecture to serve your needs.



Summary

NATS is a scalable and performant messaging system which we believe fits nicely into the microservice ecosystem. It plays extremely well with Micro and as we've demon-

strated can be used as a plugin for the Registry, Transport or Broker. We've implemented all three to highlight just how flexible NATS can be.

Micro on NATS is an example of Micro's powerful pluggable architecture. Each of the go-micro packages can be implemented and swapped out with minimal changes. In the future look to see more of examples of Micro on [X]. The next most likely to be Micro on Kubernetes.

Hopefully this will inspire you to try out Micro on NATS or even write some plugins for other systems and contribute back to the community.

Find the source for the NATS plugins at github.com/micro/go-plugins.

If you want to learn more about the services we offer or microservices, check out the blog, the website micro.mu or the github repo.

Follow us on Twitter at @MicroHQ or join the Slack community here.

Micro

Asim Aslam

The founder of Micro. Former Cloud Architect at Hailo and SRE at Google.

Share this post