BUILDING HIGH-PERFORMANCE EVENT-DRIVEN SYSTEMS USING NATS

Edwin van Wijk

Principal Architect at Info Support
Solid Innovator

Microsoft® MVP Most Valuable Professional

@evanwijk

SESSION OUTLINE:

- NATS
- NATS Streaming
- Demo: event-driven distributed application with NATS


NATS INTRODUCTION

NATS is a **high performant** and **lightweight** messaging infrastructure for building **cloud-native distributed systems**.

Designed to be **resilient** and
highly available.

Entirely written in GO.

Open Source and part of the **CNCF**.

https://github.com/nats-io

NATS is **mature** (version 2.0) and **battle tested**.

It was originally built as the messaging infrastructure for **Cloud Foundry**.

There is a **client library** available for most modern programming languages.

Supported clients:

C / C# / Elixir / Go / Java / NGINX / Node.js / Pure Ruby / Python Asyncio / Python Tornado / Ruby / TypeScript

Demos in this session are all built using .NET Core and use the supported C# client.

Community clients:

.NET / Arduino / Clojure / Elixir / Elm / Erlang / Haskell / Java Android / Lua / MicroPython / PHP / Perl / Python / Python Twisted / Qt5 C++ / Rust / Scala / Spring API / Swift

NATS FEATURES

NATS is designed to favor **simplicity**, **performance** and **availability** over a rich feature-set.

At most once delivery guarantees.

No message **persistence**.

Pure **publish/subscribe** messaging semantics.

Message-routing using **subjects**.

Several **authentication** and **authorization** features:

Security token
Username / Password
NKeys challenge / response (Ed22519 signing)
JWT (using credentials file)
Mutual TLS using X509 certs


RUNNING A NATS SERVER

The NATS server is distributed as a **single executable** and a **docker container**.

Configure NATS using **command-line flags** or by using a **configuration file**.


NATS CLUSTERING

NATS servers can be **clustered** to form a highly available **mesh** of NATS servers.

NATS clusters **automatically adapt** to newly started or stopped servers.

Updates to the **cluster topology** are automatically distributed to all connected clients.


NATS MONITORING

A **monitoring port** can be configured
when starting a server:

```
nats-server -p 4222 -m 8222
```

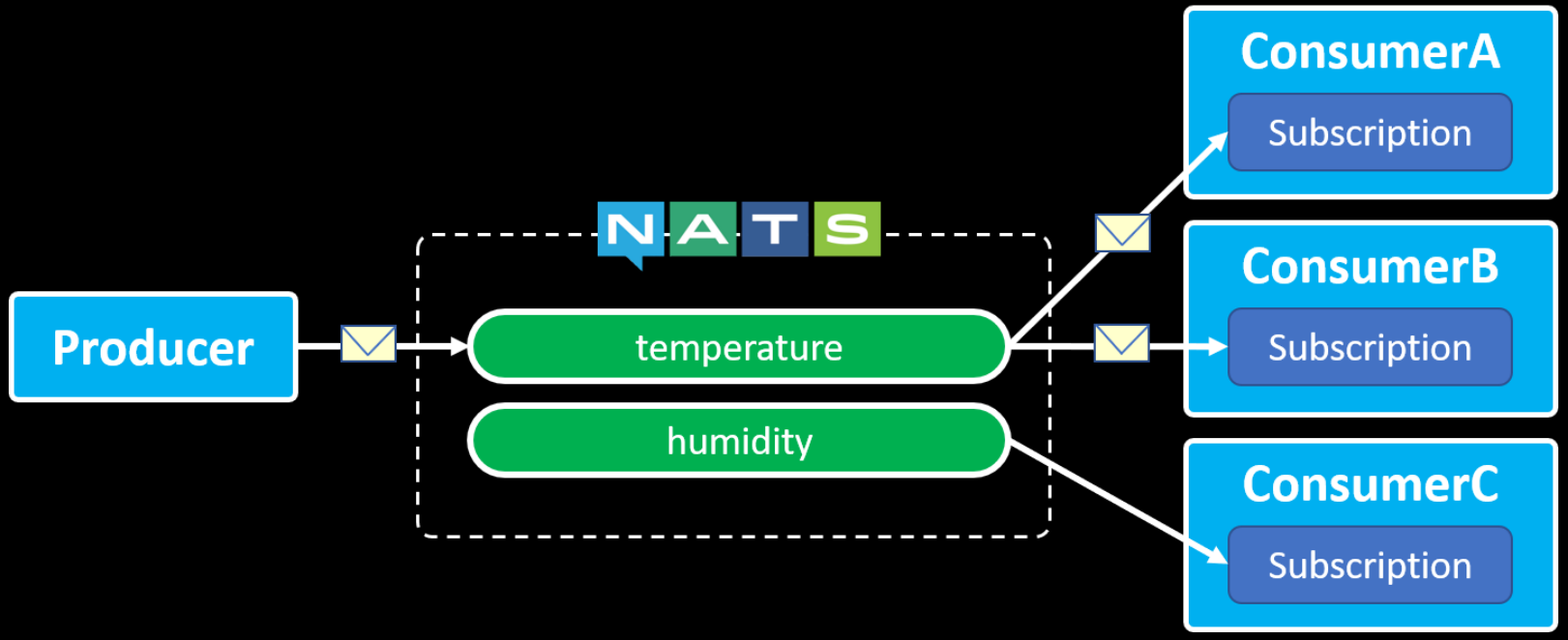
The monitoring endpoint offers a **RESTful API** for retrieving server information:

NATS	NATS Streaming
/varz	/streaming/serverz
/connz	/streaming/storez
/routez	/streaming/clientsz
/gatewayz	/streaming/clientsz
/subz	/streaming/channelsz


SUBJECT BASED MESSAGING

Producers send messages to a subject.

Consumers subscribe to a subject to receive the messages sent to this subject.


Producer
NATS
temperature
humidity
ConsumerA
Subscription
ConsumerB
Subscription
ConsumerC
Subscription

A subject is a simple **string** that can only contain **alpha-numeric** characters and the **.** character.

(Subject names are case-sensitive!)

The `.` can be used to create **subject-hierarchies**:

- `store.us.seattle`
- `store.us.newyork`
- `store.eu.amsterdam`
- `store.eu.berlin.1`
- `store.eu.berlin.2`

Wild-cards (***** and **>**) enable consumers to **listen to multiple subjects** with a single subscription.

The * wildcard matches a single element:

```
store.us.*
store.eu.*.*
dept.*.report
```

The **>** wildcard matches **multiple elements**,
but only at the **end of the subject**:

`store.eu.>`

SENDING AND RECEIVING MESSAGES

You **send** messages to a subject
using the ***Publish*** method:

```
ConnectionFactory factory = new ConnectionFactory();
IConnection conn = factory.CreateConnection();

string message = "NATS Rulez!!";
byte[] data = Encoding.UTF8.GetBytes(message);

conn.Publish("nats.demo.pubsub", data);

// ...
```

You receive messages by creating a
subscription on a subject.

Using a synchronous subscription:

```
ConnectionFactory factory = new ConnectionFactory();
IConnection conn = factory.CreateConnection();

ISyncSubscription sub =
    conn.SubscribeSync("nats.demo.pubsub");

while (true)
{
    var message = sub.NextMessage(1000);
    string data = Encoding.UTF8.GetString(message.Data);

    // handle incoming data
}
```

Using an asynchronous subscription:

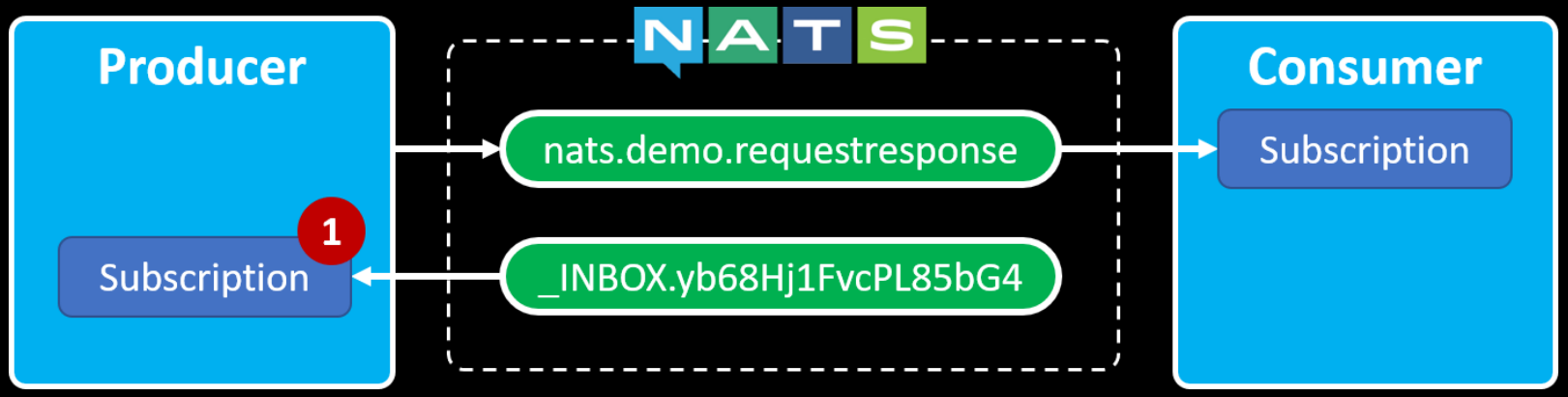
```
ConnectionFactory factory = new ConnectionFactory();
IConnection conn = factory.CreateConnection();

EventHandler<MsgHandlerEventArgs> handler = (sender, args) =>
{
    string data = Encoding.UTF8.GetString(args.Message.Data);

    // handle incoming data
};

IAsyncSubscription sub =
    conn.SubscribeAsync("nats.demo.pubsub", handler);
```

REQUEST / RESPONSE

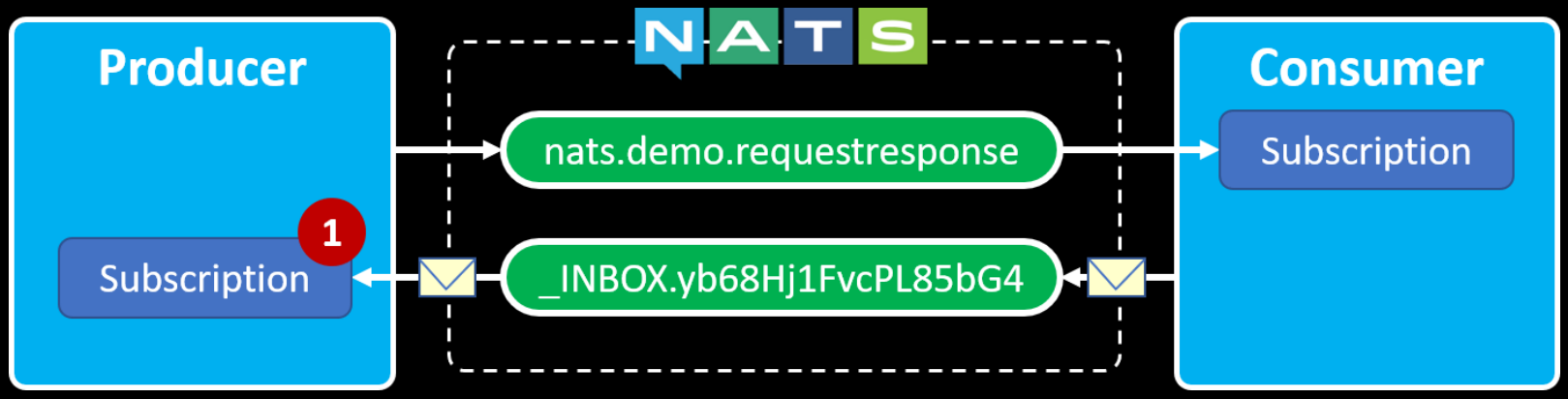
Out of the box, NATS offers pure **publish / subscribe** semantics (fire & forget).

Request / response can be implemented
using the **reply-subject** feature.


Producer
NATS
nats.demo.requestresponse
Consumer
Subscription


Producer
Subscription
1
NATS
nats.demo.requestresponse
_INBOX.yb68Hj1FvcPL85bG4
Consumer
Subscription


Producer
1
Subscription
NATS
Reply:_INBOX.yb68Hj1FvcPL85bG4
nats.demo.requestresponse
_INBOX.yb68Hj1FvcPL85bG4
Consumer
Subscription


Producer
NATS
Consumer
nats.demo.requestresponse
Subscription
1
Subscription
_INBOX.yb68Hj1FvcPL85bG4


Producer
NATS
nats.demo.requestresponse
Consumer
Subscription

Reply-subjects can be used to implement **acknowledgements** on application-level.

Most client implementations offer a *Request* method:

```
string message = "Request Payload";
byte[] data = Encoding.UTF8.GetBytes(message);

var response =
    conn.Request("nats.demo.requestresponse", data, 5000);

var responseMsg = Encoding.UTF8.GetString(response.Data);

// handle response
```

The consumer can use the **reply-subject** to reply:

```
EventHandler<MsgHandlerEventArgs> handler = (sender, args) =>
{
    string data = Encoding.UTF8.GetString(args.Message.Data);

    // handle the request

    string replySubject = args.Message.Reply;
    if (replySubject != null)
    {
        byte[] responseData = Encoding.UTF8.GetBytes($"Response")
        conn.Publish(replySubject, responseData);
    }
};
```


NATS STREAMING
(STAN)

NATS Streaming is a **streaming server**
built on top of NATS.

https://github.com/nats-io/nats-streaming-server

Messages sent to a **"channel"** are persisted
in a FIFO **message-log**.

Messages are **not removed** from the channel once they're delivered to the consumer.

Offers **at least once** messaging semantics
through **persistence** and **acknowledgements***.

* automatic or manual acks

Offers **replay** functionality with the ability
to specify the **starting point** in the message-log.

Offers multiple **subscription-types**:

- Regular
- Durable
- Queue Group

Wildcard subscriptions are not supported!

EVENT-DRIVEN DISTRIBUTED SYSTEM WITH NATS AND NATS STREAMING

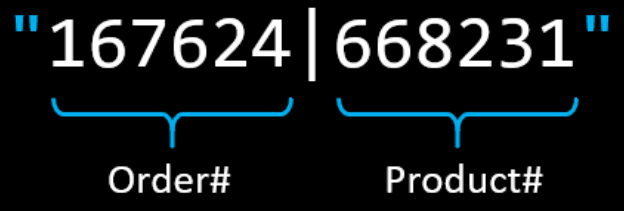
Simplified demo of a **microservices** based system for an online **book-store**.

Leverages an **event-driven** approach and
CQRS to create loosely coupled services.

Subjects for **commands** and **queries**
in NATS include the message-type:

`store.commands.CreateOrder`	`store.commands.ShipOrder`
`store.commands.OrderProduct`	`store.commands.CancelOrder`
`store.commands.RemoveProduct`	`store.queries.OrdersOverview`
`store.commands.CompleteOrder`	

The **payloads** for **commands** and **queries** are simple strings, with each field separated by a '|':

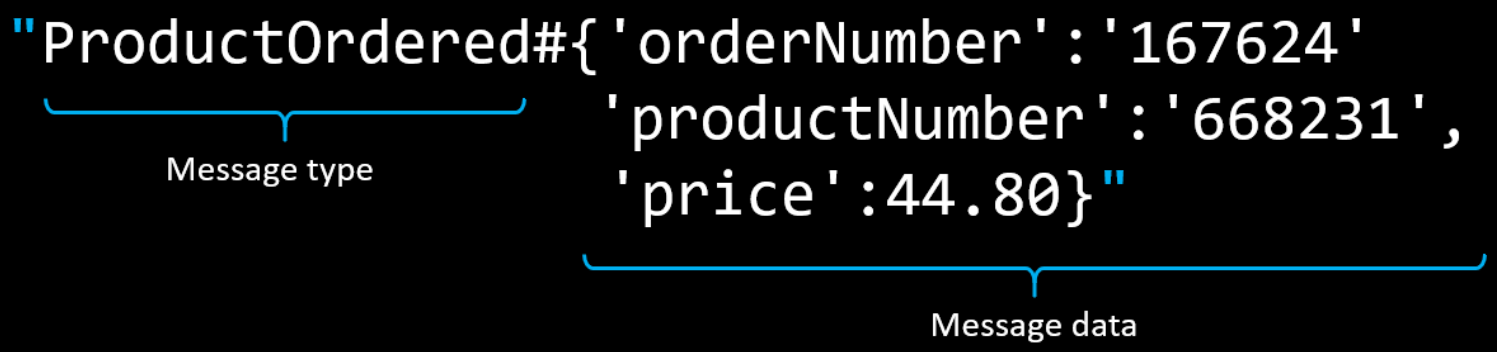


(OrderProduct command payload)

All **events** are sent to a single **subject** in NATS Streaming:

`store.events`

The **payloads** for **events** contain the **message-type** and the event serialized to **JSON** (separated by a '#'):

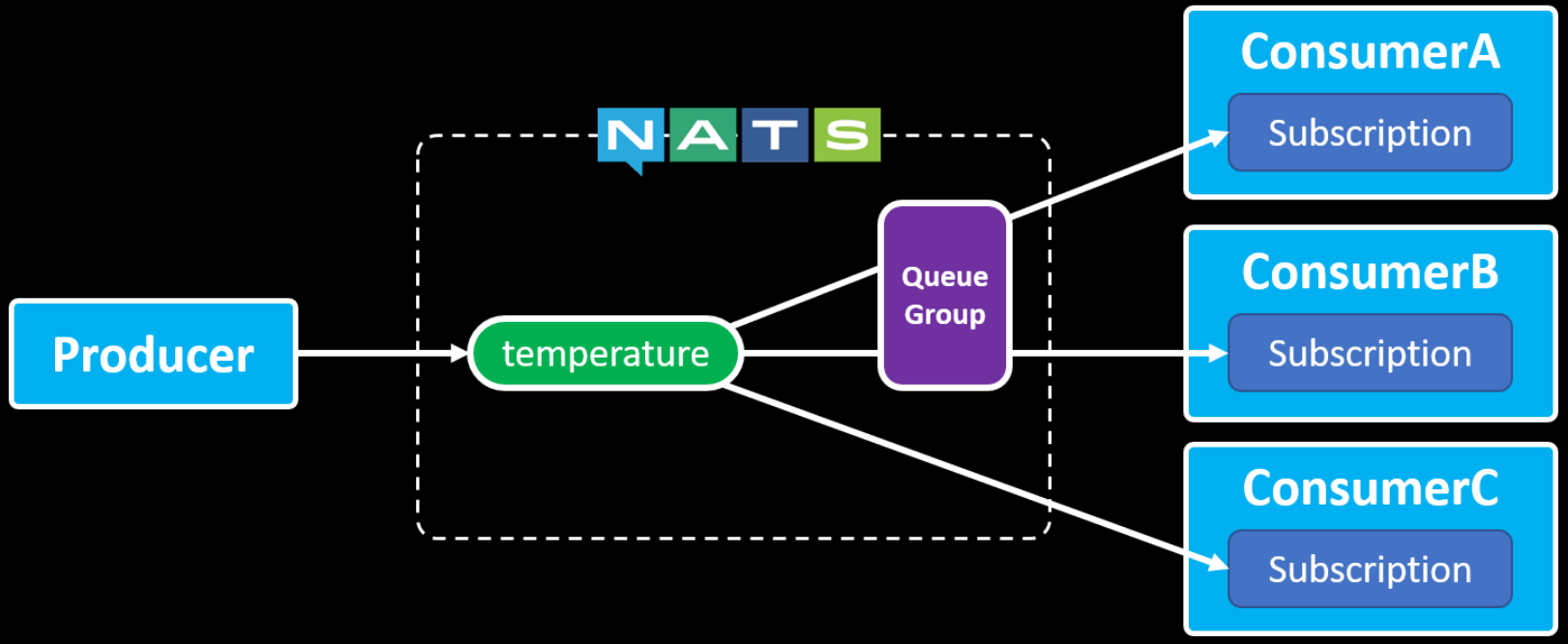


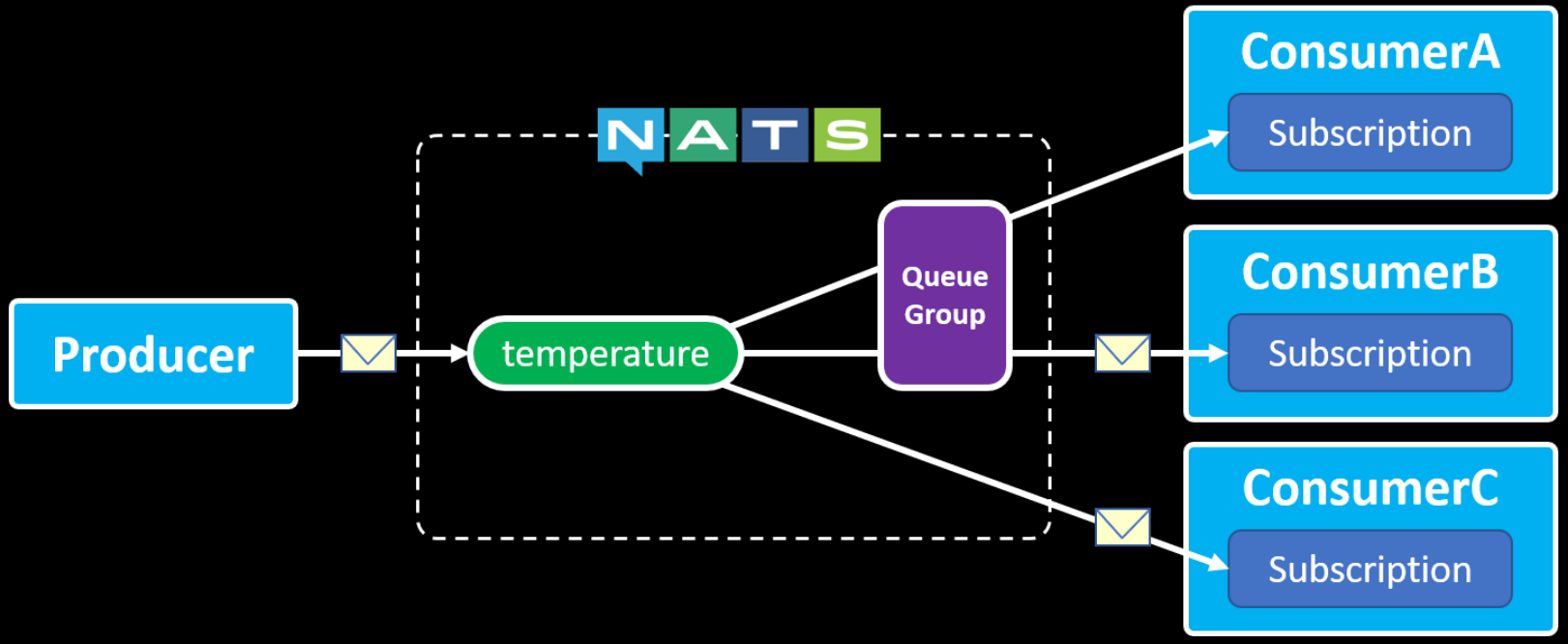
LOAD BALANCING

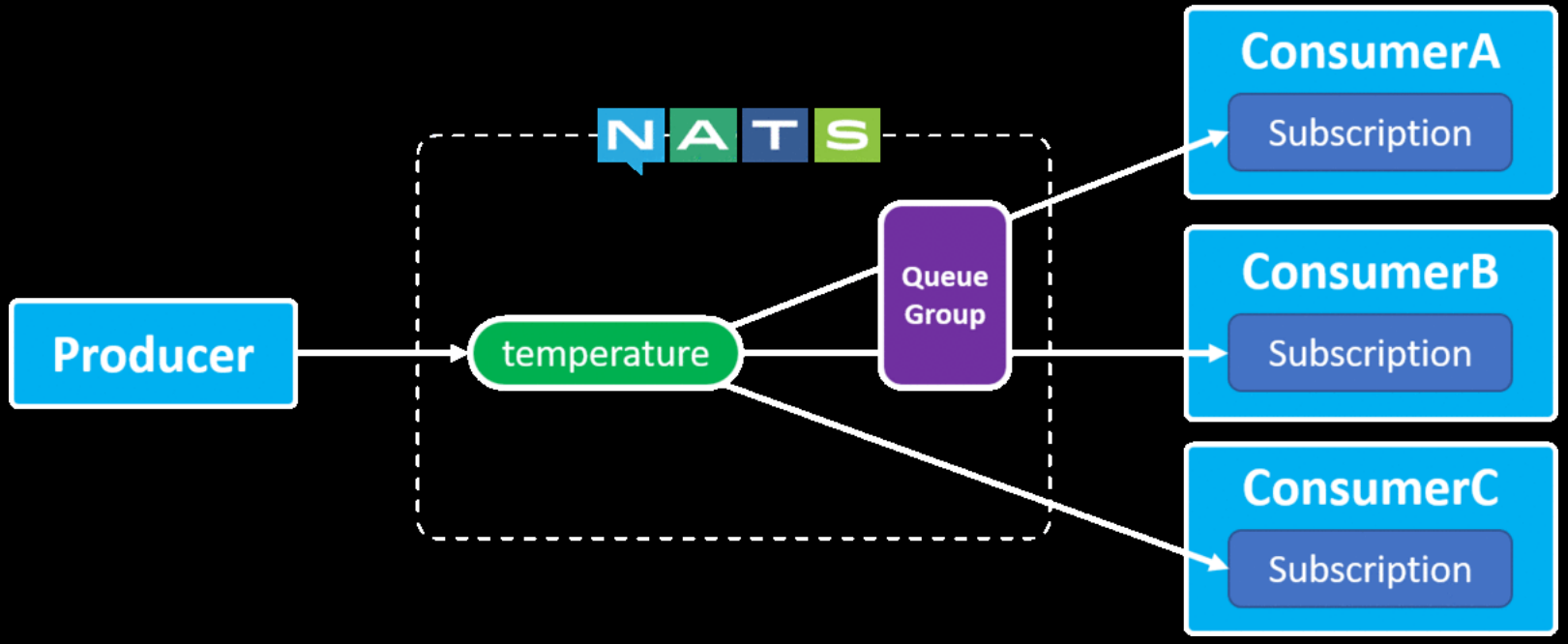
NATS offers a **load-balancing** capability out of the box.

Subscribers can specify a **queue-group**
when creating a subscription.

Messages sent to the subject will be **randomly divided** over the subscribers in the queue group.


NATS
Producer
temperature
Queue Group
ConsumerA
Subscription
ConsumerB
Subscription
ConsumerC
Subscription


NATS
Producer
temperature
Queue Group
ConsumerA
Subscription
ConsumerB
Subscription
ConsumerC
Subscription

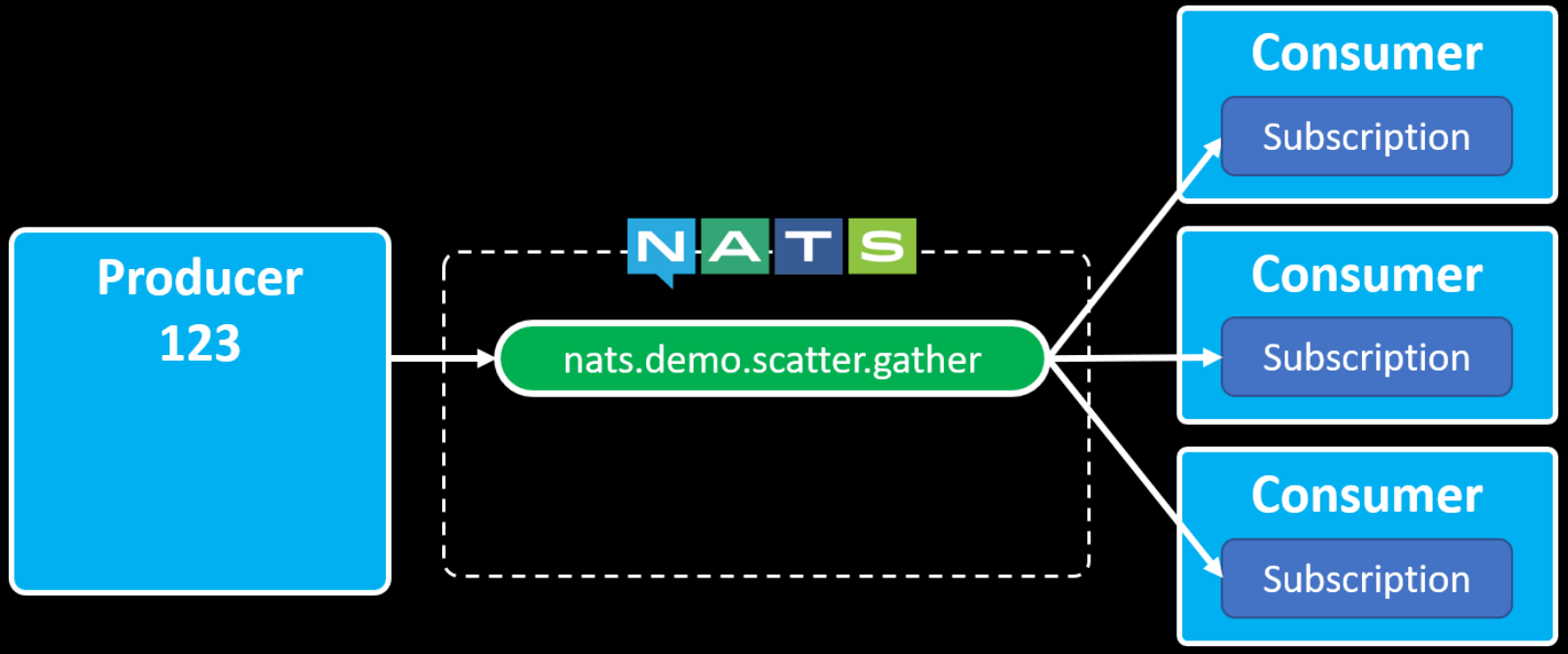

NATS
Producer
temperature
Queue Group
ConsumerA
Subscription
ConsumerB
Subscription
ConsumerC
Subscription

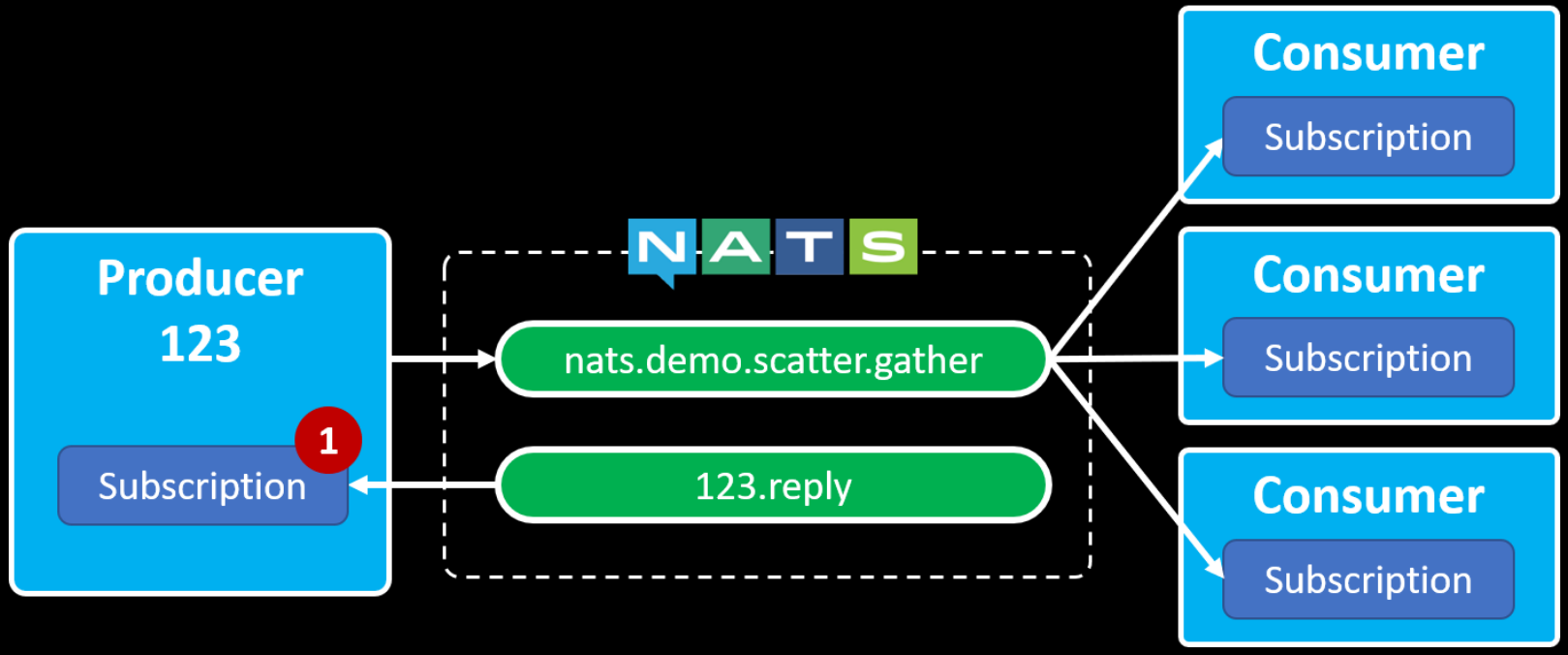
Subscribers can specify a **queue-group** when creating a subscription:

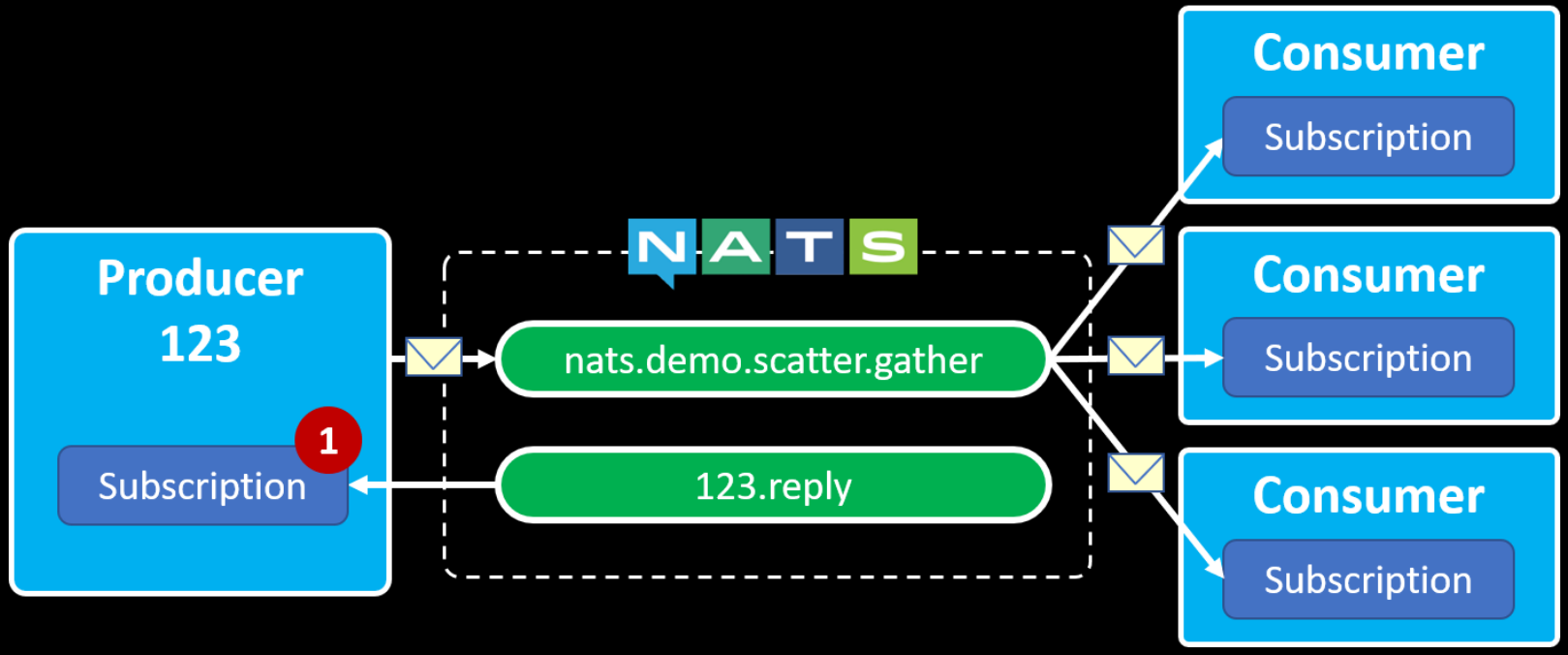
```
ConnectionFactory factory = new ConnectionFactory();
IConnection conn = factory.CreateConnection();

EventHandler<MsgHandlerEventArgs> handler = (sender, args) =>
{
    string data = Encoding.UTF8.GetString(args.Message.Data);

    // handle incoming data
};

IAsyncSubscription sub =
    conn.SubscribeAsync("nats.demo.pubsub", "lbqueue", handler);

// ...
```

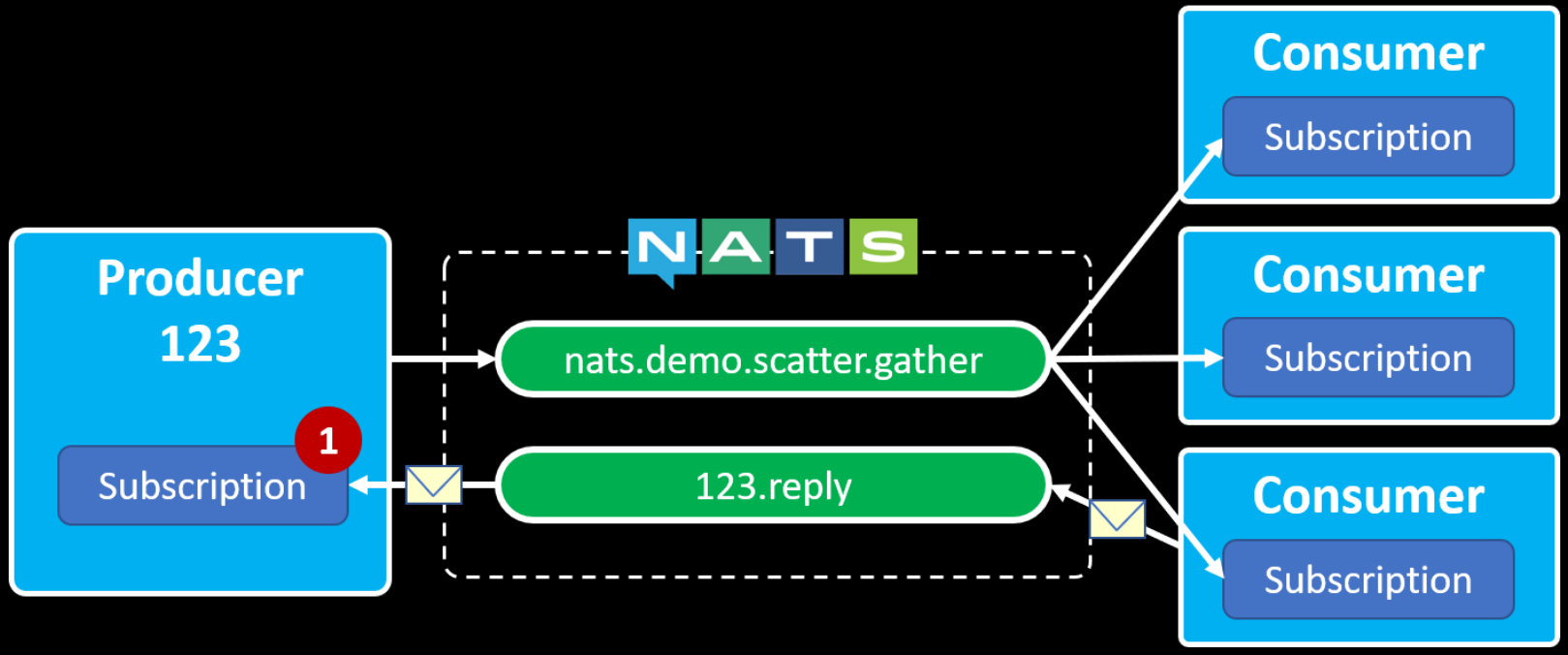
SCATTER GATHER PATTERN

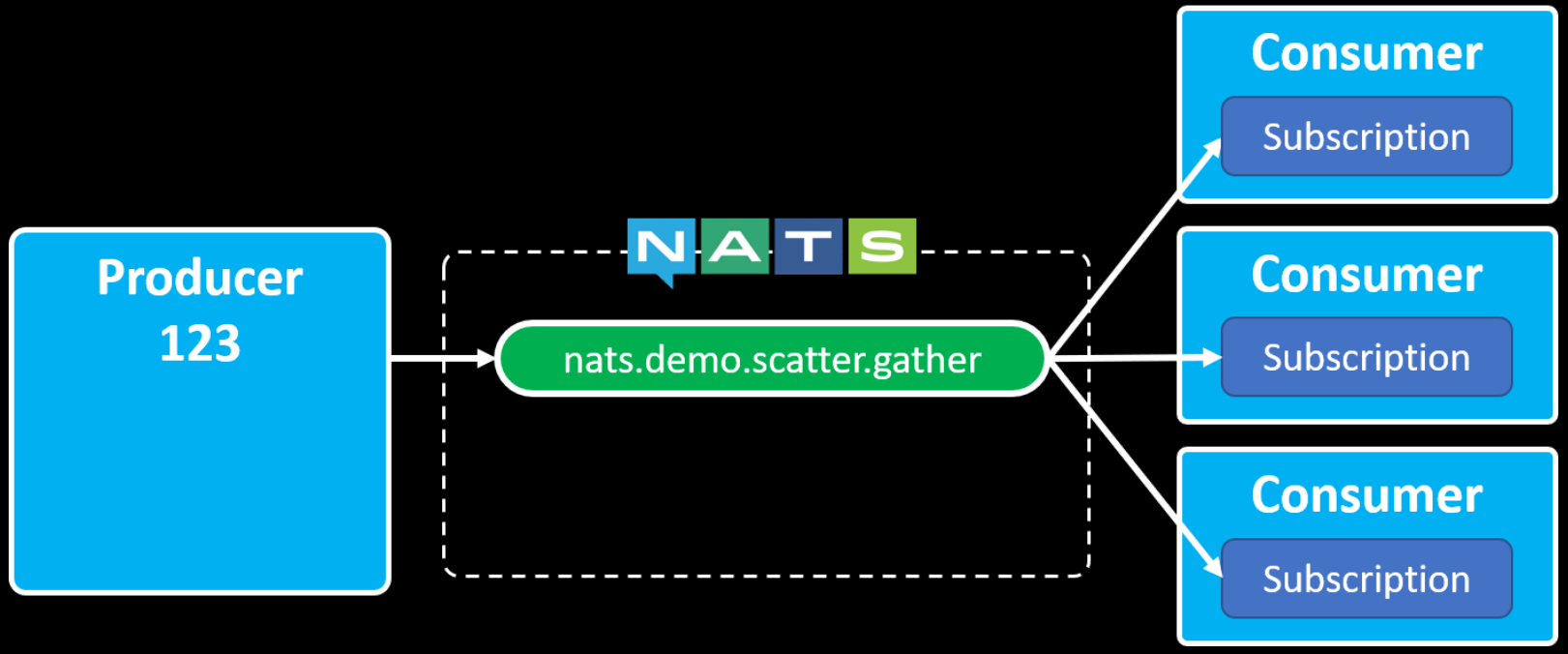
The **auto-unsubscribe** feature of NATS can be used to implement the **scatter / gather** pattern.

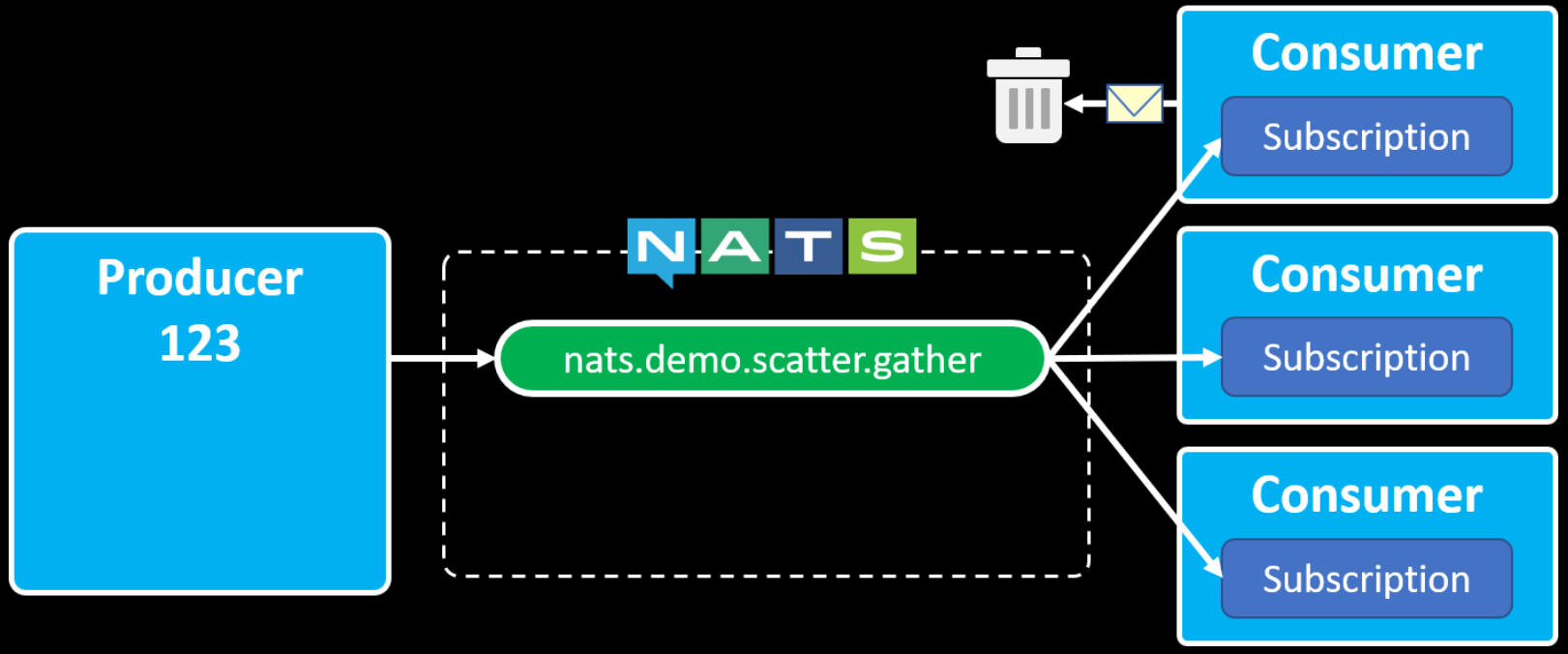

Producer
123
NATS
nats.demo.scatter.gather
Consumer
Subscription
Consumer
Subscription
Consumer
Subscription


Producer
123
1
Subscription
NATS
nats.demo.scatter.gather
123.reply
Consumer
Subscription
Consumer
Subscription
Consumer
Subscription


Producer
123
1
Subscription
NATS
nats.demo.scatter.gather
123.reply
Consumer
Subscription
Consumer
Subscription
Consumer
Subscription


Producer
123
1
Subscription
NATS
nats.demo.scatter.gather
123.reply
Consumer
Subscription
Consumer
Subscription
Consumer
Subscription


Producer
123
NATS
nats.demo.scatter.gather
Consumer
Subscription
Consumer
Subscription
Consumer
Subscription


Consumer
Subscription
Consumer
Subscription
Consumer
Subscription
NATS
Producer
123
nats.demo.scatter.gather

Scatter gather can be used to get a response from the **fastest service** in a pub/sub scenario.


NATS COMMUNICATION PROTOCOL

NATS uses a very **small** and **simple** text-based communication protocol over **TCP**.

INFO	Sent to client after initial TCP/IP connection
CONNECT	Sent to server to specify connection information
PUB	Publish a message to a subject, with optional reply subject
SUB	Subscribe to a subject (or subject wildcard)
UNSUB	Unsubscribe (or auto-unsubscribe) from subject
MSG	Delivers a message payload to a subscriber
PING	PING keep-alive message
PONG	PONG keep-alive response
+OK	Acknowledges well-formed protocol message in verbose mode
-ERR	Indicates a protocol error. May cause client disconnect

You can test the protocol using **telnet**:

```
$ telnet demo.nats.io 4222
INFO {"server_id":"NAA...EFS", ... ,"client_id":36837}
connect {"name":"telnet"}
+OK
sub test 5
+OK
pub test 12
NATS Rulez!!
+OK
MSG test 5 12
NATS Rulez!!
unsub 5
+OK
```

WRAP UP

NATS is a performant messaging infrastructure for building **cloud-native distributed systems**.

It is **simple**, **lightweight** and **resilient**.

You can choose for **NATS** or **NATS streaming** (or both) based on your messaging **requirements**:

At most once **vs.** At least once delivery

Raw performance **vs.** Message persistence and replay

Make sure you understand (and accept) the **trade-offs** that come with NATS' simplicity and performance.

Use **clustering** to make your NATS infrastructure resilient.

Check out the **documentation** and start **experimenting**!

https://nats.io

Get the **demo code** from this session on **GitHub**:

https://github.com/edwinvw/nats-demos

THANK YOU!

Edwin van Wijk


@evanwijk