# NATS Message Path Tracing

| Metadata | Value |
|---|---|
| Date | 2024-02-22 |
| Author | @ripienaar, @kozlovic |
| Status | Implemented |
| Tags | observability, server, 2.11 |

| Revision | Date | Author | Info |
|---|---|---|---|
| 1 | 2024-02-22 | @ripienaar | Initial design |

## Context and Problem Statement

As NATS networks become more complex with Super Clusters, Leafnodes, multiple Accounts and JetStream knowing the path that messages take through the system is hard to predict.

Further, when things go wrong, it's hard to know where messages can be lost, denied or delayed.

This describes a feature of the NATS Server 2.11 that allows messages to be traced throughout the NATS network.

## Prior Work

NATS supports tracking latency of Request-Reply service interactions, this is documented in ADR-3.

## Design

When tracing is activated every subsystem that touches a message will produce Trace Events. These Events are aggregated per server and published to a destination subject.

A single message published activating tracing will therefor result in potentially a number of Trace messages - one from each server a message traverse, each holding potentially multiple Trace Events.

At present the following *Trace Types* are supported

- Ingress (`in`) - The first event that indicates how the message enters the Server, client connection, route, gateway etc
- Subject Mapping (`sm`) - Indicates the message got transformed using mappings that changed it's target subject
- Stream Export (`se`) - Indicates the message traversed a Stream Export to a different account
- Service Import (`si`) - Indicates the message traversed a Service Import to a different account
- JetStream (`js`) - Indicates the message reached a JetStream Stream
- Egress (`eg`) - The final event that indicates how the message leaves a Server

## Activation

Not all messages are traced and there is no flag to enable it on all messages. Activation is by adding Headers to the message.

### Ad-hoc activation

This mode of Activation allows headers to be added to any message that declares where to deliver the traces and inhibit delivery to the final application.

| Header | Description |
|---|---|
| `Nats-Trace-Dest` | A subject that will receive the Trace messages |
| `Nats-Trace-Only` | Prevents delivery to the final client, reports that it would have been delivered (`1`, `true`, `on`) |
| `Accept-Encoding` | Enables compression of trace payloads (`gzip`, `snappy`) |

The `Nats-Trace-Only` header can be used to prevent sending badly formed messages to subscribers, the servers will trace the message to its final destination and report the client it would be delivered to without actually delivering it. Additionally when this is set messages will also not traverse any Route, Gateway or Leafnode that does not support the Tracing feature.

### Trace Context activation

Messages holding the standard `traceparent` header as defined by the Trace Context specification can trigger tracing based on the `sampled` flag.

In this case no `Nats-Trace-Dest` header can be set to indicate where the messages will flow, it requires enabling using an account setting:

```
accounts {
  A {
    users: [{user: a, password: pwd}]
    msg_trace: {
        dest: "a.trace.subj"
        sampling: "100%"
    }
  }
}
```

Here we set the `msg_trace` configuration for the `A` account, this enables support for Trace Context and will deliver all messages with the `traceparent` header and the `sampled` flag set to true.

Note the `sampling`, here set to 100% which is the default, will further trigger only a % of traces that have the `sampled` value in the `traceparent` header. This allow you to specifically sample only a subset of messages traversing NATS while your micro services will sample all.

When this feature is enabled any message holding the `Nats-Trace-Dest` header as in the previous section will behave as if the `traceparent` header was not set at all. In essence the Ad-Hoc mode has precedence.

### Cross Account Tracing

By default a trace will end at an account boundary when crossing an Import or Export. This is a security measure to restrict visibility into foreign accounts and require opt-in to allow.

```
accounts {
  B {
   exports = [
    // on a service the direction of flow is into the exporting
    // account, so the exporter need to allow tracing
    { service: "nats.add", allow_trace: true }
   ]

   imports = [
    // on a stream import the direction of flow is from exporter into
    // the importer, so the importer need to allow tracing
    {stream: {account: A, subject: ticker}, allow_trace: true}
   ]
  }
}
```

## nats CLI

The current `main` and nightly builds of `nats` includes the `nats trace` command that is built upon these features.

This uses a helper package to receive, parse, sort and present a series of related Traces, the source can be found in github.com/nats-io/jsm.go/api/server/tracing.

```
$ nats trace demo
Tracing message route to subject demo

Client "NATS CLI Version development" cid:4219 cluster:"sfo" server:"n2-sfo"
version:"2.11.0-dev"
==> Gateway "n1-lon" gid:727
    Gateway "n2-sfo" gid:735 cluster:"lon" server:"n1-lon" version:"2.11.0-dev"
    ~~> Leafnode "leaf1" lid:5391
        Leafnode "n1-lon" lid:8 server:"leaf1" version:"2.11.0-tracing8"
        --C Client "NATS CLI Version development" cid:10 subject:"demo"

Legend: Client: --C Router: --> Gateway: ==> Leafnode: ~~> JetStream: --J Error: -
-X

Egress Count:

   Gateway: 1
```

```
    Leafnode: 1
      Client: 1
```

Here we can see:

1. Message entered the `sfo` Cluster via a Client in the server `n2-sfo`.
2. The server `n2-sfo` published it to its Gateway called `n1-lon` using connection `gid:727`
3. The server `n1-lon` received from its Gateway called `n2-sfo` with connection `gid:735`
4. The server `n1-lon` published it to its Leafnode connection `leaf1` using connection `lid:5391`
5. The server `leaf1` received from its Leafnode called `leaf1` with connection `lid:8`
6. The server `leaf1` published the message to a Client with the connection name "NATS CLI Version development" over connection `cid:10`

This is a set of 3 Trace messages holding 6 Trace Events in total.

## Trace Message Formats

The full detail of the trace message types are best found in the NATS Server source code in server/msgtrace.go, here we'll call out a few specifics about these messages.

Given this sample message - a `MsgTraceEvent`:


```
{
  "server": {
    "name": "n3-lon",
    "host": "n3-lon.js.devco.net",
    "id": "NCCPZOHDJ4KZQ35FU7EDFNPWXILA7U2VJAUWPG7IFDWUADDANNVOWKRV",
    "cluster": "lon",
    "domain": "hub",
    "ver": "2.11.0-dev",
    "tags": [
      "lon"
    ],
    "jetstream": true,
    "flags": 3,
    "seq": 151861,
    "time": "2024-02-22T13:18:48.400821739Z"
  },
  "request": {
    "header": {
      "Nats-Trace-Dest": [
        "demo"
      ]
    },
    "msgsize": 36
  },
  "hops": 1,
  "events": [
    {
      "type": "in",
      "ts": "2024-02-22T13:18:48.400595984Z",
```

```
        "kind": 0,
        "cid": 5390,
        "name": "NATS CLI Version development",
        "acc": "one",
        "subj": "x"
      },
      {
        "type": "eg",
        "ts": "2024-02-22T13:18:48.40062456Z",
        "kind": 0,
        "cid": 5376,
        "name": "NATS CLI Version development",
        "sub": "x"
      },
      {
        "type": "eg",
        "ts": "2024-02-22T13:18:48.400649361Z",
        "kind": 1,
        "cid": 492,
        "name": "n1-lon",
        "hop": "1"
      }
    ]
  }
```


Lets have a quick look at some key fields:

|Key|Notes| |`server`|This is a standard Server Info structure that you will see in many of our advisories, just indicates which server sent the event| |`request`|Details about the message being traced, more details about `Nats-Trace-Hop` below| |`hops`|How many remote destination (routes, gateways, Leafnodes) this server is sending the message to| |`events`|A list of `MsgTraceEvents` that happened within the server related to this message, see below|

## Sorting

These events form a Directed Acyclic Graph that you can sort using the Hop information. The Server that sends the message to another Route, Gateway or Leafnode will indicate it will publish to a number of `hops` and each server that receives the message will have the `Nats-Trace-Hop` header set in its `request`.

The origin server - the server that receives the initial message with the `Nats-Trace-Dest` header - will have a hops count indicating to how many other servers (routes, leafs, gateways) it has forwarded the message to. This is not necessarily the total number of servers that the message will traverse.

Each time a server forwards a message to a remote server, it appends a number to its own `Nats-Trace-Hop` header value. Since the origin server does not have one, if it forwards a message to say a ROUTE, that remote server would receive the message with a new header `Nats-Trace-Hop` with the value of `1`, then the origin server forwards to a GATEWAY, and that server would receive the message with `Nats-Trace-Hop` value set to `2`.

Each of these servers in turn, if forwarding a message to another server, would add an incremental number to their existing `Nats-Trace-Hop` value, which would result in `1.1`, and `2.1`, etc..

Take a simple example of the origin server that has a LEAF server, which in turn as another LEAF server (daisy chained). If a message with tracing is forwarded, the trace message emitted from the origin server would have hops to `1` (since the origin server forwarded directly only to the first LEAF remote server). The trace emitted from the first LEAF would have a `Nats-Trace-Hop` header with value `1`. It would also have hops set to `1` (assuming there was interest in the last LEAF server). Finally, the last LEAF server would have a `Nats-Trace-Hop` of `1.1` and would have no hops field (since value would be 0).

You would associate message either by using a unique `Nats-Trace-Dest` subject or by parsing the `traceparent` to get the trace and span IDs.

### Trace Events

The `events` list contains all the events that happened in a given server. We see here that there are different types of event according to the table below:

| Type | Server Data Type | Description |
|---|---|---|
| `in` | `MsgTraceIngress` | Ingress |
| `sm` | `MsgTraceSubjectMapping` | Subject Mapping |
| `se` | `MsgTraceStreamExport` | Stream Export |
| `si` | `MsgTraceServiceImport` | Service Import |
| `js` | `MsgTraceJetStream` | JetStream |
| `eg` | `MsgTraceEgress` | Egress |

We also see a `kind` field, this holds the NATS Servers client Kind, at present these are the values:

| Kind | Server Constant | Description |
|---|---|---|
| `0` | `server.CLIENT` | Client Connection |
| `1` | `server.ROUTER` | Router Connection |
| `2` | `server.GATEWAY` | Gateway Connection |
| `3` | `server.SYSTEM` | The NATS System |
| `4` | `server.LEAF` | Leafnode Connection |
| `5` | `server.JETSTREAM` | JetStream |
| `6` | `server.ACCOUNT` | Account |

You may not see all Kinds in traces but this is the complete current list.

Using these Kinds and Types we can understand the JSON data above:

- `Ingress` from `Client` connection `cid:5390`
- `Egress` to `Client` connection `cid:5376`
- `Egress` to `Router` connection `rid:492` called `n1-lon`

This indicates a Client connected to this server (`n3-lon`) published the message it was received by a client connected to the same server and finally sent over a Route to a client on another server (`n1-lon`).

## Specific Trace Types

Most of the trace types are self explanatory but I'll call out a few specific here, the names match those in `msgtrace.go` in the Server source.

### MsgTraceEgress and MsgTraceIngress

These messages indicates the message entering and leaving the server via a `kind` connection. It includes the `acc` it's being sent for, subscription (`sub`) and Queue Group (`queue`) on the Egress.

**Note** that for non CLIENT egresses, the subscription and queue group will be omitted. This is because NATS Servers optimize and send a single message across servers, based on a known interest, and let the remote servers match local subscribers. So it would be misleading to report the first match that caused a server to forward a message across the route as the egress' subscription.

Here in cases of ACLs denying a publish the `error` will be filled in:


```
{
  "events": [
    {
      "type": "in",
      "ts": "2024-02-22T13:44:18.326658996Z",
      "kind": 0,
      "cid": 5467,
      "name": "NATS CLI Version development",
      "acc": "one",
      "subj": "deny.x",
      "error": "Permissions Violation for Publish to \"deny.x\""
    }
  ]
}
```


### MsgTraceJetStream

The values are quite obvious, the `nointerest` boolean indicates that an `Interest` type Stream did not persist the message because it had no interest.