RX-M Cloud Native Consulting

Microservices

Lab 5 - Messaging and Loosely Coupled Systems

NATS is a family of open source, MIT License products that are tightly integrated but can be deployed independently. NATS is used in a range of systems including mobile, IoT, microservices, and cloud native applications. The core NATS Server acts as a central nervous system for building distributed applications. There are dozens of clients including Java, .NET and Go. NATS Streaming extends the platform to provide for real-time streaming & big data use-cases.

In this lab we will deploy a NATS message broker in a networked container. Then we will simulate IoT gateway traffic using NATS messages, as if the gateways were forwarding trash can level reports from the city streets. Then we will create a report microservice the rest of the application can access to recover trash can levels at any time.

1. Create a Private Network for the IoT Message Traffic

Docker provides support for software defined networks (SDN). Because we are interested in managing thousands of dynamically scaled microservices in any given application, the ability to create arbitrary L2 networks is key to our success with microservices.

Imagine we want to isolate all of the IoT gateway trash can level traffic. We could ask Docker to create a network used only by the IoT gateway agents and the NATS message broker.

Tell Docker to create an IoT net:

```
user@ubuntu:~/trash-can/levels$ cd ~

user@ubuntu:~$ docker network create iot-net

64702fb0fed619cb4833202a8801f9ab8f1398eab6cbba2273c93b79c2412df7
user@ubuntu:~$
```

Run the `docker network` subcommand by itself see the commands available:

```
user@ubuntu:~$ docker network

Usage:  docker network COMMAND

Manage networks

Options:
      --help   Print usage

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.
user@ubuntu:~$
```

List out the networks on your machine using the `docker ls` subcommand:

```
user@ubuntu:~$ docker network ls

NETWORK ID          NAME                DRIVER              SCOPE
913a0e892a2c        bridge              bridge              local
c4fd5236b924        host                host                local
64702fb0fed6        iot-net             bridge              local
8dddac639dd4        none                null                local
user@ubuntu:~$
```

The network we created is a Linux Bridge based network (the default) and will be isolated to this host computer. Docker includes the overlay driver which, when configured, can be used to create multi host networks. Docker also supports many CNM network plugins which supply a range of SDN solutions.

Inspect the network's metadata:

```
user@ubuntu:~$ docker network inspect iot-net
```

```
[
    {
        "Name": "iot-net",
        "Id": "64702fb0fed619cb4833202a8801f9ab8f1398eab6cbba2273c93b79c2412df7",
        "Created": "2017-03-22T03:34:45.670235078-07:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
user@ubuntu:~$
```

Our network received the subnet 172.18.0.0/16 (yours may differ.) Docker will automatically resolve container names on user defined networks, like this one, to IP addresses via a proxy DNS agent. We'll use this feature to allow our microservices to discover the NATS broker without knowing its IP in advance.

Next lets start the NATS service.

2. Starting a NATS Message Broker

In this lab we will run everything in containers but we will need to create two programs, a simulator to generate trash levels and a report server to capture and store them.

Create a working directory for your report code.

```
user@ubuntu:~$ cd ~/trash-can/

user@ubuntu:~/trash-can$ mkdir report

user@ubuntu:~/trash-can$ cd report
```

Next run a NATS server (in a container of course) connected to the iot-net:

```
user@ubuntu:~/trash-can/report$ docker container run -d --name iot-msg --network-alias iot-msg --net iot-net nats

Unable to find image 'nats:latest' locally
latest: Pulling from library/nats
2d3d00b0941f: Pull complete
24bc6bd33ea7: Pull complete
Digest: sha256:47b825feb34e545317c4ad122bd1a752a3172bbbc72104fc7fb5e57cf90f79e4
Status: Downloaded newer image for nats:latest
44a6070943a84f3c25cefb0bdf17327676542c640e6c200724e18eb6f7062446
user@ubuntu:~/trash-can/report$
```

List out the containers on your system:

```
user@ubuntu:~/trash-can/report$ docker container ls

CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
44a6070943a8        nats                "/gnatsd -c gnatsd..."   26 seconds ago      Up 23 seconds       4222/tcp,
6222/tcp, 8222/tcp   iot-msg
user@ubuntu:~/trash-can/report$
```

Our NATS server is running and listening on three ports. Port 4222 is the messaging port. The other ports are used for control and management.

We can use `docker container inspect` to discover the IP address of our server:

```
user@ubuntu:~/trash-can/report$ docker container inspect iot-msg | grep IPAddress

            "SecondaryIPAddresses": null,
            "IPAddress": "",
                    "IPAddress": "172.18.0.2",
user@ubuntu:~/trash-can/report$
```

Docker created the server on the 172.18/16 network as we requested. However we do not need the IP to connect to the server. Try running a container on the same network and then use the name "iot-msg" to ping the NATS server. We can use the alpine image (a light weight Alpine based Linux environment with basic command line tools) to run our `ping` command:

```
user@ubuntu:~/trash-can/report$ docker container run --net=iot-net alpine ping -c 1 iot-msg

Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
88286f41530e: Pull complete
Digest: sha256:f006ecbb824d87947d0b51ab8488634bf69fe4094959d935c0c103f4820a417d
Status: Downloaded newer image for alpine:latest
PING iot-msg (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.074 ms

--- iot-msg ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.074/0.074/0.074 ms
user@ubuntu:~/trash-can/report$
```

This works because Docker configures the containers it runs with a `resolv.conf` configuration that directs the container IP stack to use the Docker engine as the name server for user defined networks. The 127.0.0.0/8 address range is reserved for loopback operations and Docker hooks the .11 address on port 53 (in iptables) in order to supply containers with name resolution. Names unknown to Docker are passed on to the normal DNS resources configured on the network.

```
user@ubuntu:~/trash-can/report$ docker container run --net=iot-net busybox cat /etc/resolv.conf

search localdomain
nameserver 127.0.0.11
options ndots:0
user@ubuntu:~/trash-can/report$
```

Great, our NATS server is up. Now we can simulate trash can level IoT traffic.

3. Creating a Gateway Traffic Simulator

Imaging that in the trash level reporting bounded context we have decided to use JavaScript. So we'll create our trash can level report simulator using NodeJS.

First create a simple program simulation IoT messages from trash cans containing their ID and percentage full:

```
user@ubuntu:~/trash-can/report$ vim sim.js

user@ubuntu:~/trash-can/report$ cat sim.js
```

```javascript
var servers = ['nats:iot-msg:4222'];
var topic = 'trash-level';
var nats = require('nats').connect({'servers':servers});

function trash_level() {
    var lvl = Math.random() * 100;
    var can_id = (Math.random() * 1000).toFixed();
    msg = '{"can_id":"' + can_id + '", "level":"' + lvl + '"}';
    console.log('publishing: ' + msg);
    nats.publish(topic, msg);
}

setInterval(trash_level, 4000);
```

```
user@ubuntu:~/trash-can/report$
```

This simple program generates a random trash level and a random 3 digit can ID and then packs it into a JSON message for transmission to the NATS broker. The NodeJS *setInterval()* function calls our message generator every 4 seconds creating a constant stream of messages.

The NATS server we connect to is the one on our network with the name "iot-msg". As you can see we could list several servers This allows you to create a cluster of NATS servers for HA or scaling purposes. We only write to one server with each message but clients can listen to all of the servers in the cluster.

Now we can use a Node JS container to test our program:

```
user@ubuntu:~/trash-can/report$ docker container run -it -v ~/trash-can/report:/app --net=iot-net --name=sim node
/bin/bash

Unable to find image 'node:latest' locally
latest: Pulling from library/node
693502eb7dfb: Already exists
081cd4bfd521: Already exists
5d2dc01312f3: Already exists
54a5f7da9a4f: Already exists
f6ebc0704397: Pull complete
cd487f72ac9f: Pull complete
a93aceb4d4b2: Pull complete
eb9bbc6304ca: Pull complete
Digest: sha256:7639cf1f253af2a3a9a213d83c4e6dca61a8b75f13499aead75f17dc8679e0b8
Status: Downloaded newer image for node:latest
root@5eb903eacb10:/#
```

The command we issued runs the node image from Docker hub and mounts our working directory with our `sim.js` program into the container using the *-v* switch. We also connect to the iot-net (--net) where our NATS server is running and launch a bash shell with an interactive tty (-it).

Now use the node package manager (npm) to install the NodeJS NATS library in the container:

```
root@fec514a7c43a:/# npm install nats

npm WARN saveError ENOENT: no such file or directory, open '/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/package.json'
npm WARN !invalid#1 No description
npm WARN !invalid#1 No repository field.
npm WARN !invalid#1 No README data
npm WARN !invalid#1 No license field.

+ nats@1.0.1
added 2 packages from 1 contributor and audited 2 packages in 0.439s
found 0 vulnerabilities

root@fec514a7c43a:/#
```

We can ignore the warnings, and the installer has not indicated any errors so we can proceed.

Finally run your simulator:

```
root@bc271661b3d5:/# node /app/sim.js

publishing: {"can_id":"135", "level":"42.930236081376314"}
publishing: {"can_id":"826", "level":"73.63959586423843"}
publishing: {"can_id":"538", "level":"28.598132886884354"}
...
```

Perfect! Leave the simulator running. Because we are using messaging, our microservices are loosely coupled and atomically deployable. The simulator has no idea whether the report server is running or not, nor does it care.

4. Creating a Trash Level Report Service

Now we can create a trash reporting service. Open a new terminal, create the following `rep.js` file to implement the report server:

```
user@ubuntu:~/trash-can/report$ vim rep.js

user@ubuntu:~/trash-can/report$ cat rep.js
```

```javascript
var servers = ['nats:iot-msg:4222'];
var topic = 'trash-level';
var nats = require('nats').connect({'servers':servers});

nats.subscribe(topic, function(msg) {
  console.log('Received report: ' + msg);
});
```

```
user@ubuntu:~/trash-can/report$
```

Our report server will connect to the same NATS broker cluster as the simulator and will subscribe to the topic the simulator is publishing, trash-level. Let's give it a try!

Run another Node container, install NATS, and run the report server code:

```
user@ubuntu:~/trash-can/report$ docker container run -it -v ~/trash-can/report:/app --net=iot-net --name=rep node
/bin/bash
```

```
root@26a64b293469:/# npm install nats

npm WARN saveError ENOENT: no such file or directory, open '/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/package.json'
npm WARN !invalid#1 No description
npm WARN !invalid#1 No repository field.
npm WARN !invalid#1 No README data
npm WARN !invalid#1 No license field.

+ nats@1.0.1
added 2 packages from 1 contributor and audited 2 packages in 0.439s
found 0 vulnerabilities

root@26a64b293469:/#
```

```
root@26a64b293469:/# node /app/rep.js

Received report: {"can_id":"734", "level":"20.297191921596024"}
Received report: {"can_id":"225", "level":"55.85403489938945"}
Received report: {"can_id":"540", "level":"30.82060976169345"}
Received report: {"can_id":"45", "level":"57.25747108133685"}
```

Great, we now have a trash level simulator and a reporting server.

5. Clustering the messaging system

To test the cloud native resilience of NATS we should try running a cluster of NATS brokers and then intentionally kill one to see if messages are still delivered. The "iot-msg" container created above is the only NATS container running presently, so we will join its cluster.

It is easy to add additional NATS brokers to a cluster. The command we will use looks like this:

`docker run -d --name=iot-msg2 --net=iot-net nats -c gnatsd.conf --routes=nats-route://ruser:T0pS3cr3t@iot-msg:6222`

Let's examine the command piece by piece:

- docker container run :This is the docker command to create and start a new container
- -d :This is the switch used to "detach" the container from the terminal (run it in the background)
- --name=iot-msg2 :This switch names the container
- --net=iot-net :This swtich connects us to the IOT network (the net with all of our containers for this lab)
- nats :This specifies the image to run, NATS
- -c gnatsd.conf :Arguments after the image are passed to the image exe, this tells NATS to use the default config file
- --routes=nats-route://ruser:T0pS3cr3t@iot-msg:6222 :This switch tells NATS to connect (nats-route) as the user "ruser" with the password "T0pS3cr3t" (both NATS defaults that you would override in production) to the server "iot-msg" on the default port (6222).

Try it:

```
ubuntu@ip-172-31-6-251:~/trash-can/report$ docker run -d --name=iot-msg2 --net=iot-net  nats -c gnatsd.conf --
routes=nats-route://ruser:T0pS3cr3t@iot-msg:6222

3b8dffc6f14c8ddaafdd8e4d462b408185713502e1dacd391fa9e9f3e10e8ff3

ubuntu@ip-172-31-6-251:~/trash-can/report$
```

Check the log output of the new broker:

```
ubuntu@ip-172-31-6-251:~/trash-can/report$ docker container logs iot-msg2

[1] 2018/07/25 20:31:56.081249 [INF] Starting nats-server version 1.2.0
[1] 2018/07/25 20:31:56.081306 [INF] Git commit [6608e9a]
```

```
[1] 2018/07/25 20:31:56.081394 [INF] Starting http monitor on 0.0.0.0:8222
[1] 2018/07/25 20:31:56.081420 [INF] Listening for client connections on 0.0.0.0:4222
[1] 2018/07/25 20:31:56.081429 [INF] Server is ready
[1] 2018/07/25 20:31:56.081580 [INF] Listening for route connections on 0.0.0.0:6222
[1] 2018/07/25 20:31:56.084084 [INF] 172.19.0.2:6222 - rid:1 - Route connection created

ubuntu@ip-172-31-6-251:~/trash-can/report$
```

The last line in the log tells us that the new broker we executed found the iot-msg broker at IP 172.19.0.2 (your system may produce a different IP).

Verify that your report server is still receiving messages (it should be).

Now let's kill the original message broker to see if our NATS cluster is truely resilient.

```
ubuntu@ip-172-31-6-251:~/trash-can/report$ docker container kill iot-msg

iot-msg

ubuntu@ip-172-31-6-251:~/trash-can/report$
```

Take a look at your level reporting service:

```
Received report: {"can_id":"684", "level":"32.868171242209556"}
Received report: {"can_id":"567", "level":"13.845117254267025"}
Received report: {"can_id":"455", "level":"30.997351503611824"}
Received report: {"can_id":"430", "level":"39.141507789554495"}
Received report: {"can_id":"104", "level":"67.65542216909583"}
```

Note that does not seem to care at all that the first broker was just killed. This is because the NATS brokers and client libraries allow the entire NATS cluster to act like a single unit. The cluster of NATS brokers is resilient, allowing clients to ignore individual node failures. As long as at least one node is still running, the cluster is operational. In production you might run 10 or more NATS brokers so that if one fails you only lose 10% of your total capacity.

6. Cleanup

Stop your service and applications and commit the code.

To stop the node applications you can simply press ^C in their terminals:

```
Received report: {"can_id":"401", "level":"15.852232630313324"}
^C
root@79a70de42da1:/#
```

To exit the containers just `exit` the root shells they are running:

```
root@5eb903eacb10:/# exit

exit
user@ubuntu:~/trash-can/report$
```

Now create a Git repo for your trash reporting applications:

```
user@ubuntu:~/trash-can/report$ git init

Initialized empty Git repository in /home/user/trash-can/report/.git/

user@ubuntu:~/trash-can/report$ git status

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        rep.js
        sim.js

nothing added to commit but untracked files present (use "git add" to track)
user@ubuntu:~/trash-can/report$
```

Add your Javascript source files to the index and commit them:

```
user@ubuntu:~/trash-can/report$ git add *.js

user@ubuntu:~/trash-can/report$ git commit -m "initial NATS commit"

[master (root-commit) be5ecb9] initial NATS commit
 2 files changed, 20 insertions(+)
 create mode 100644 rep.js
 create mode 100644 sim.js
user@ubuntu:~/trash-can/report$
```

Congratulation you have successfully completed Lab 5!

[OPTIONAL] Run Multiple Simulators in Separate Containers with a Single Subscriber Container.

[OPTIONAL] Run Multiple Simulators in Separate Containers with a Multiple Subscribers in Separate Containers.