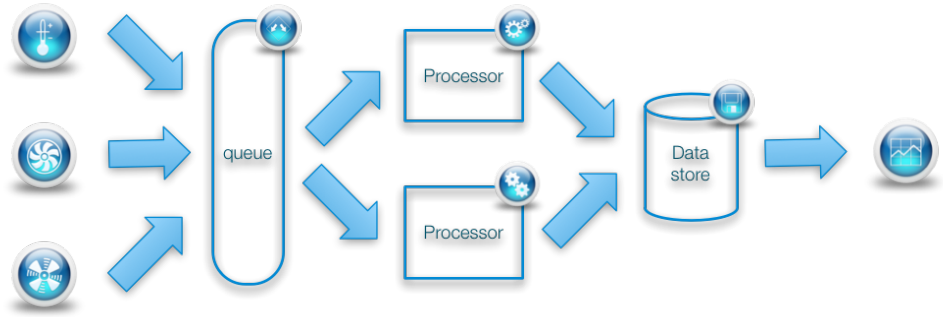
Installation and setup guide

version 1.0, last modified: 2015-02-20

This document explains how to set up the INAETICS demonstrator. For this, we use a Vagrant-based setup that boots a cluster of up to five nodes on a single machine, and with the help of a couple of shell scripts we show how you can create evolvable systems using Docker and OSGi. This document is structured as follows: first some background information about the demonstrator application itself is given after which the preparations and startup of the cluster is described. In the last section, the demonstration scenarios are described showing the dynamic aspects of the demonstrator application.

Demonstrator overview

An INAETICS system is about creating systems that can evolve in and over time. To demonstrate the scalability aspects of an INAETICS system, this demonstrator provides a sample application as denoted in the following figure:



At the utmost left, we have a number of data **producers**, which publish data onto a shared **queue**. Assuming the data needs a bit of processing, one or more **processors** picks up the data from the queue, processes it and stores the result in a single **data store**. Users can use the data from this store for reporting purposes. With these four aspects of our application, we can demonstrate the following scenarios which affect the equilibrium of the system:

1. if the producers publish *more* data than the processors can handle, in which case the queue-size will grow. In this case, we need to **scale out** by adding more processors;
2. suppose a processor fails and stops functioning, we need to handle **fail-over** to another processor;
3. in case hardware resources fail, we can dynamically reconfigure the parts across different resources.

The various parts of the application are written in both C and Java that together form a distributed application that shows the ability of how a polyglot environment can scale and handle fail-over scenarios. Each part is deployed onto one or more computing resources (virtualised Linux environments) using Apache ACE as provisioning solution and uses Fleet for the (minimalistic) scheduling and monitoring.

In the following section, the preparations and installations of the prerequisites are explained in more detail.

Vagrant-based installation

In this section, the installation of the INAETICS demonstrator is described using Vagrant. You can use it to install the INAETICS demonstrator on a single machine. This section uses CentOS-7 as host platform. Nevertheless, the installation can rather easily be ported to other (Linux-based) host platforms, as long as it provides support for Vagrant, Oracle VirtualBox, Docker and Git.

Tested configuration

For the Vagrant-based configuration, we used the following configuration:

	Configuration
Hardware	1 laptop (Core i7, 16GB RAM)
OS	CentOS 7.0.1406
CoreOS	github.com/coreos/coreos-vagrant (CoreOS Alpha 550.0.0) (Vagrant 1.6.5) (VirtualBox 4.3.20)
Etcd	Etcd v0.4.6
Fleet	Fleet v0.9.0
Docker	docker 1.3.2

In the following sections, the installation is described in more detail.

Prerequisites

For the installation of the prerequisites, a CentOS-7 installation is used. As the prerequisites are commonly available, it should be trivial to convert these instructions to the distribution of your preference. For the CentOS-7 based installation, the EPEL (Extra Packages for Enterprise Linux) and Oracle VirtualBox repositories are needed:

```
$ sudo yum repolist
Loaded plugins: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirrors.supportex.net
 * epel: ftp.rediris.es
 * extras: mirror.denit.net
 * updates: mirror.yourwebhoster.eu
repo id                           repo name
status
!base/7/x86_64                    CentOS-7 - Base
8,465
*!epel/x86_64                     Extra Packages for Enterprise Linux 7 - x86_64
6,966
!extras/7/x86_64                  CentOS-7 - Extras
102
!updates/7/x86_64                 CentOS-7 - Updates
1,531
!virtualbox/7/x86_64              Oracle Linux / RHEL / CentOS-7 / x86_64 -
VirtualBox          4
repolist: 17,068
```

To install the prerequisites, you should issue:

```
$ sudo yum install vagrant virtualbox git
```

Once the installation of the various packages is complete, we need to ensure that the `vboxnet0` network interface is properly configured. This is needed to establish proper communication between the various virtual machines that make up the cluster:

```
$ vboxmanage hostonlyif remove vboxnet0
```

Once this is done, we can clone the demonstrator-cluster repository containing everything you need to run the actual demonstrator:

```
$ git clone https://github.com/INAETICS/demonstrator-cluster.git
Cloning into 'demonstrator-cluster'...
...
$ cd demonstrator-cluster
$ export INAETICS_HOME=`pwd`
$ git submodule init && git submodule update
...
Submodule path 'docker-images/provisioning': checked out 'xyz'
```

Inside the `demonstrator-cluster` repository, you find everything that is needed for the Vagrant-based setup:

- the `bin/` folder contains a utility script to purge the state information of an Etcd cluster: `purge_etcd_discovery.sh`. This is useful in case you need to restart the compute resources;
- the `bootstrap/` folder contains the Vagrant configuration for the "bootstrap" machine. This is a Linux instance running CoreOS and provides the bare infrastructure services, like an Etcd-cluster and a Docker registry. It has a predefined IPv4 address that is used by the compute resources to obtain their state/runtime information;
- the `docker-images/` folder contain the descriptions of the various Docker images that are used in the INAETICS demonstrator application. These docker-images are provisioned automatically to the bootstrap machine;
- the `workers/` folder contains the Vagrant configuration for the compute resources, or workers. By default, this Vagrant configuration is defined as 5 Linux instances running nothing more than CoreOS.

Starting all the virtual machines on one machine is quite memory intensive. Running this demonstrator on a machine with at least 16 GB of memory is advised.

Starting the bootstrap VM

The bootstrap VM provides the basic infrastructure for the compute resources, such as an Etcd cluster and a Docker registry. The Etcd cluster is used to store and share information about the state of the system between all compute resources. The Docker registry is used by the compute resources to obtain the containerised parts of the INAETICS demonstrator application. To start the bootstrap VM, we do the following:

```
$ cd $INAETICS_HOME/bootstrap
$ vagrant up
Bringing machine 'cluster-bootstrap' up with 'virtualbox' provider...
==> cluster-bootstrap: Checking if box 'coreos-alpha' is up to date...
...
```

After a little while the bootstrap VM is up and running which can be verified by requesting the following URL:

```
$ curl http://172.17.8.2:5000/v1/search
{"query": "", "num_results": 0, "results": []}$
```

Note that it takes a little while before the Docker registry is actually started, especially if you start it the first time, so you might need to access the URL a couple of times.

Before we can build the Docker images, which make up the actual demonstrator application, we should SSH into the bootstrap VM:

```
$ vagrant ssh
CoreOS alpha (584.0.0)
Update Strategy: No Reboots
core@cluster-bootstrap ~ $
```

Building the Docker images

The compute resources use Docker images to run the various parts of the demonstrator application. The compute resources use a single shared

Docker registry to obtain these Docker images. For building and pushing the Docker images, we have created a convenience script (`docker_build.sh`) that simplify these steps into a single script call. All we need to do is issue the following command:

```
core@cluster-bootstrap ~ $ sh docker_build.sh docker-images/provisioning
localhost:5000 inaetics/provisioning
This script might need root permissions to run...
Sending build context to Docker daemon 28.43 MB
...
```

The node provisioning image contains a complete and preconfigured Apache ACE server and is used for provisioning the demonstrator application to both the "celix-agent" and "felix-agent" images.

> If during the Docker build you get an error like:
>
> ```
> FATA[0000] Error: Invalid registry endpoint "http://localhost:5000/v1/". HTTPS attempt: Get https://
> localhost:5000/v1/_ping: dial tcp 127.0.0.1:5000: connection refused. HTTP attempt: Get http://local
> host:5000/v1/_ping: dial tcp 127.0.0.1:5000: connection refused
> ```
>
> This implies that the Docker registry is not yet up and running, and will resolve itself when waiting a little longer before invoking the "docker_build" script again.
>
> You can verify whether the docker registry is running by running the command:
>
> ```
> $ docker ps -a
> CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
> 0d922ed7ef6f registry:latest "docker-registry" 21 seconds ago Up 20 seconds 0.0.0.0:5000->5000/tcp
> docker-registry-service
> ```

As explained in the introduction, the demonstrator application are developed in both C and Java. Given that the demonstrator application is modular and dynamically reconfigurable, this means that a small agent needs to run in order to get the provisioned software from the provisioning server. For the Java specific parts, this agent is based on Apache Felix and the Docker image is created by:

```
core@cluster-bootstrap ~ $ sh docker_build.sh docker-images/felix-agent localhost:5000
inaetics/felix-agent
This script might need root permissions to run...
Sending build context to Docker daemon 1.752 MB
...
```

The agent that is responsible for running the C specific parts is based on Apache Celix and its image is created by:

```
core@cluster-bootstrap ~ $ sh docker_build.sh docker-images/celix-agent localhost:5000
inaetics/celix-agent
This script might need root permissions to run...
Sending build context to Docker daemon  34.3 kB
...
```

Once all Docker images are created and pushed to the registry running on the coordinator, we can verify that they exist by asking Docker:

```
core@cluster-bootstrap ~ $ docker images
REPOSITORY                              TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
localhost:5000/inaetics/celix-agent     latest              e4493375d41e        2
minutes ago         659 MB
localhost:5000/inaetics/felix-agent     latest              7caa0c9a2a1d        6
minutes ago         398.2 MB
localhost:5000/inaetics/provisioning    latest              ca455bf3140d        8
minutes ago         424.8 MB
registry                                latest              c55308716b36        3 weeks
ago           418.6 MB
ubuntu                                  14.04               5ba9dab47459        3 weeks
ago           192.7 MB
```

If all Docker images are present, we can close the SSH session to our coordinator VM:

```
core@cluster-bootstrap ~ $ exit
Connection to 127.0.0.1 closed.
```

Starting the compute resources

The compute resources (workers) are plain CoreOS Linux distributions that are provisioned with a number of scripts for convenience of this demonstrator. By default there are *five* compute resources started, which can be controlled by the `$num_instances` variable in the `workers/Vagrantfile`. Note that if you lower the number of instances, the demonstrator scenarios might not work correctly! To start the compute resources we issue:

```
$ cd $INAETICS_HOME/workers
$ vagrant up
Vagrant configured for 5 inaetics workers
...
```

Once all five workers are booted and ready, we can SSH into the **first** compute resource (it could be any compute resource):

```
$ vagrant ssh worker-1
Vagrant configured for 5 inaetics workers
CoreOS alpha (584.0.0)
Update Strategy: No Reboots
core@worker-1 ~ $
```

It can happen sometimes that the compute resources aren't assigned the right IPv4 address, which causes the demonstrator application to fail. To verify that the compute resources are in the correct state, we check whether they have to correct IP address assigned to the `eth1` adapter:

```
core@worker-1 ~ $ ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 172.17.8.101 netmask 255.255.255.0 broadcast 172.17.8.255
...
```

If the IPv4 address is **not** `172.17.8.101`, this means that the VirtualBox image is not correctly configured by Vagrant and we need to stop and start the compute resources again (using `vagrant halt && vagrant up`)

With all preconditions in place we can proceed with running the actual INAETICS demonstrator application.

The INAETICS demonstrator

As described in the overview section, the demonstrator application consists of several parts that together form a distributed application that is running on the compute resources. To distribute the various parts across the compute resources Fleet is used. Fleet uses annotated systemd unit files and transports and invokes this unit file on a designated machine. Besides stopping and starting an application, Fleet also does minimal health-checks and restarts an application if it detects that it is no longer running.

For convenience of this demonstrator, we have created a wrapper script called `inaetics_fleet_manager` that does all the heavy-lifting with Fleet for us and is already installed on all the compute resources.

The initial state of the INAETICS demonstrator application consists of the following components:

- one provisioning server, which is used to provision the correct software to the various agents;
- 2 Celix agents, for running the C-based parts of the demonstrator;
- 2 Felix agents, for running the Java-based parts of the demonstrator.

Starting the demonstrator application

To start the INAETICS demonstrator application, we need to SSH into the first compute resource and invoke the inaetics_fleet_manager script like:

```
$ cd $INAETICS_HOME/workers && vagrant ssh worker-1
...
core@worker-1 ~ $ inaetics_fleet_manager --start
Inaetics Environment starting with 2 Celix agents and 2 Felix agents
Triggered unit provisioning.service start
Triggered unit felix@1.service start
Triggered unit felix@2.service start
Triggered unit celix@1.service start
Triggered unit celix@2.service start
```

Fleet will distribute the various unit files across our compute resources and start them for us. We can check on the progress by:

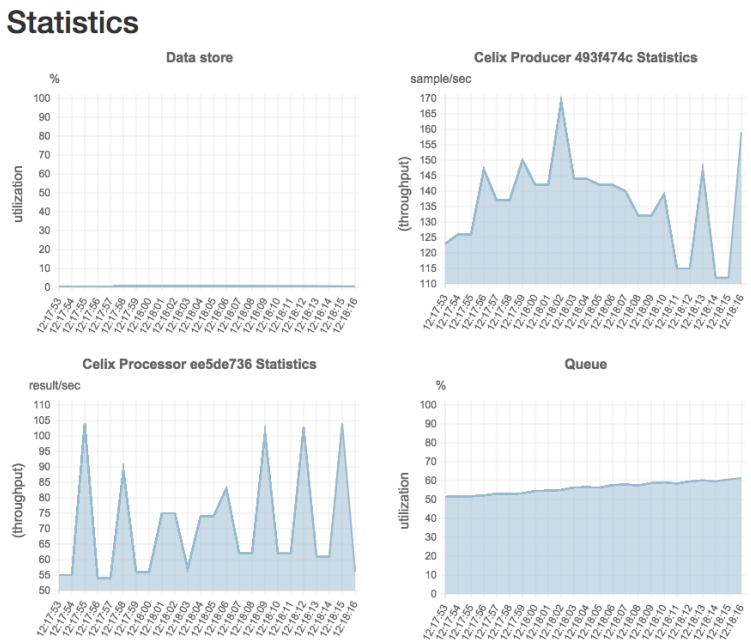
```
core@worker-1 ~ $ inaetics_fleet_manager --status
Available machines:
MACHINE   IP         METADATA
081e2194...  172.17.8.102  -
8e4c3d7b...  172.17.8.103  -
bd850c1a...  172.17.8.104  -
d191fbc3...  172.17.8.101  -
dc730136...  172.17.8.105  -


Submitted unit files:
UNIT                  HASH       DSTATE      STATE      TARGET
celix@.service        401685f    inactive    inactive   -
celix@1.service       401685f    launched    launched   d191fbc3.../172.17.8.101
celix@2.service       401685f    launched    launched   dc730136.../172.17.8.105
felix@.service        b53dfab    inactive    inactive   -
felix@1.service       b53dfab    launched    launched   8e4c3d7b.../172.17.8.103
felix@2.service       b53dfab    launched    launched   bd850c1a.../172.17.8.104
provisioning.service  237b701    launched    launched   081e2194.../172.17.8.102

Deployed units:
UNIT                  MACHINE                    ACTIVE  SUB
celix@1.service       d191fbc3.../172.17.8.101   active  running
celix@2.service       dc730136.../172.17.8.105   active  running
felix@1.service       8e4c3d7b.../172.17.8.103   active  running
felix@2.service       bd850c1a.../172.17.8.104   active  running
provisioning.service  081e2194.../172.17.8.102   active  running
```

Note that although the status reports all deployed units as "running", it takes a little while before all software is actually provisioned to the agents!

To get a little more insight into the state of the various parts of the demonstrator application, a simple web-based viewer is provisioned to the machine running the "`felix@1.service`" unit, which is `172.17.8.103` in the example above. To access it, point your webbrowser to http://172.17.8.103:8080/, which should provide you a number of graphs as shown in the following figure:



In this figure, we see that our queue has a steadily growing utilisation, which implies that the processors cannot keep up with the amount of data

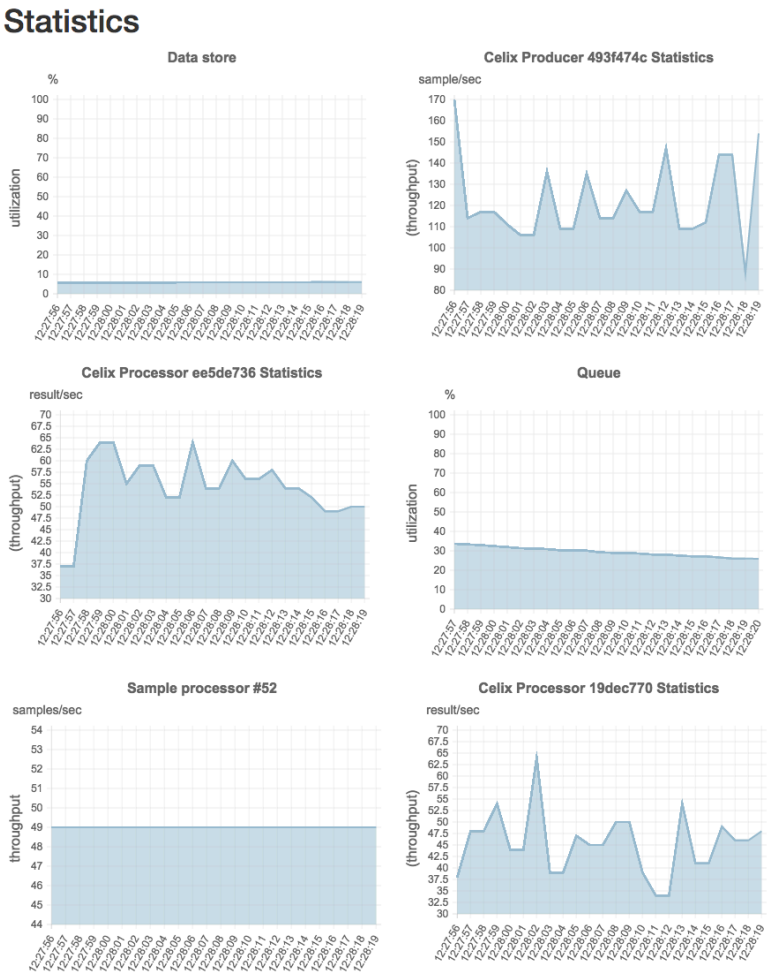
produced by the producers.

Scaling out the demonstrator

To lower the queue utilisation, we simply can add more processors. We do this by adding another Celix and Felix agent:

```
core@worker-1 ~ $ inaetics_fleet_manager --start --celixAgents=3 --felixAgents=3
Inaetics Environment starting with 3 Celix agents and 3 Felix agents
Triggered unit felix@3.service start
Triggered unit celix@3.service start
```

After a while, the additional processors are started and will cause the queue utilisation to decrease as can be seen in the following figure:



Besides scaling out, we can also scale back again, by issuing the following command:

```
core@worker-1 ~ $ inaetics_fleet_manager --start --celixAgents=2 --felixAgents=2
Inaetics Environment starting with 2 Celix agents and 2 Felix agents
Triggered unit felix@3.service stop
Triggered unit felix@3.service unload
Triggered unit celix@3.service stop
Triggered unit celix@3.service unload
```

The result of this will be that the demonstrator consists again of a single producer and consumer and a queue that has an increasing utilisation.

Handling fail-over and dynamic reconfiguration

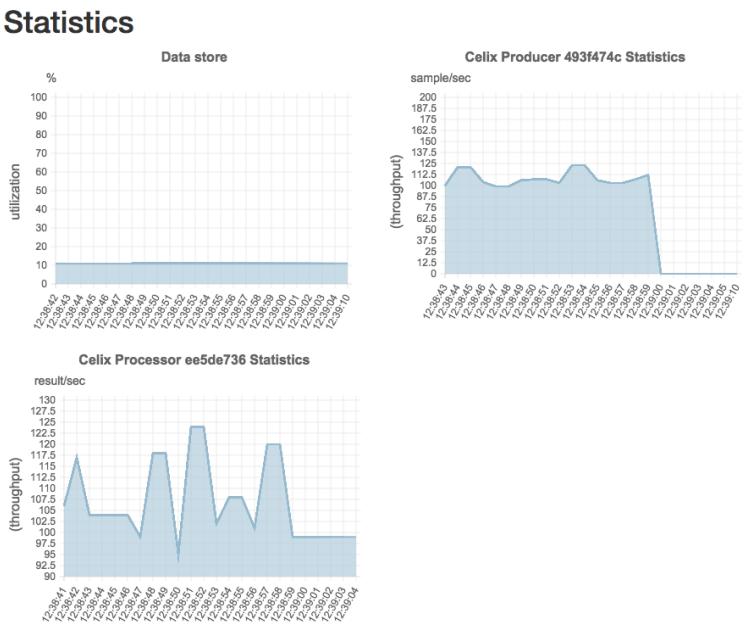
To simulate a fail-over scenario, we need to "pull the plug" on one of the agents. Suppose "`felix@2.service`" terminates representing the queue service, this causes both the producers and processors to stop functioning. Fortunately, Fleet ensures that this service is restarted after sudden termination causing the demonstrator to function properly again. To simulate the termination of the `felix@2.service`, we SSH into the compute resource on which it runs and kill the Docker container in which the Felix agent runs. By looking at the status report of the `inaetics_fleet_manager` script above, we see that it is scheduled on the compute resource with IP address `172.17.8.104`, which corresponds to the **four th** worker (the first worker is configured to `172.17.8.101`, the second to `172.17.8.102`, and so on):

```
$ vagrant ssh worker-4
...
core@worker-4 ~ $ docker ps
CONTAINER ID   IMAGE                                                COMMAND
CREATED             STATUS              PORTS     NAMES
e51592a110b4   172.17.8.2:5000/inaetics/felix-agent:latest   "/tmp/node-agent.sh    23
minutes ago      Up 23 minutes       [...]     felix-2
```

To terminate the Docker container:

```
core@worker-4 ~ $ docker kill e51592a110b4
e51592a110b4
```

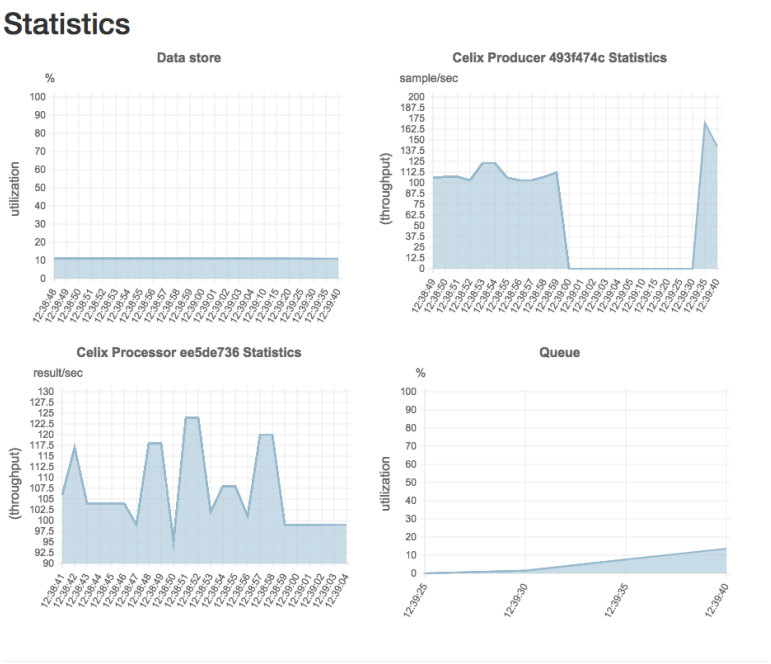
If you watch the viewer page closely, you see the following situation appear:



There is no queue present anymore, and the producer is not able to place its data onto the queue. The processor shows an average throughput which explains why its throughput is not dropped to zero. After a little while, the queue should be restarted on the same node (note the different container ID!):

```
core@worker-4 ~ $ docker ps
CONTAINER ID        IMAGE                                                COMMAND
CREATED             STATUS              PORTS     NAMES
be5b58268496        172.17.8.2:5000/inaetics/felix-agent:latest   "/tmp/node-agent.sh
16 seconds ago      Up 15 seconds       [...]     felix-2
```

And the viewer will represent this accordingly:



You see that the producer is able to push its data onto the (new) queue and that the queue utilisation starts to grow again.

A more fatal situation can be simulated by halting a compute resource's VM. To terminate the second compute resource (running a Celix processor) we issue:

```
$ vagrant halt worker-2
```

The result of this is that Fleet now has to reschedule the failing processor on to another compute resource, to compensate for the loss of the second worker. After a while, the situation should be restored, and we can see that the processor is restarted on another computing resource by running the `inaetics_fleet_manager` script on the first compute resource again (note the missing compute resource with IPv4 address `172.17.8.102`):

```
core@worker-1 ~ $ inaetics_fleet_manager --status
Available machines:
MACHINE      IP             METADATA
04be238e...  172.17.8.101   -
5ed2251e...  172.17.8.105   -
a5097953...  172.17.8.104   -
be956748...  172.17.8.103   -

Submitted unit files:
UNIT                  HASH      DSTATE     STATE      TARGET
celix@.service        401685f   inactive   inactive   -
celix@1.service       401685f   launched   launched   be956748.../172.17.8.103
celix@2.service       401685f   launched   launched   04be238e.../172.17.8.101
celix@3.service       401685f   inactive   inactive   -
felix@.service        b53dfab   inactive   inactive   -
felix@1.service       b53dfab   launched   launched   5ed2251e.../172.17.8.105
felix@2.service       b53dfab   launched   launched   a5097953.../172.17.8.104
felix@3.service       b53dfab   inactive   inactive   -
provisioning.service  237b701   launched   launched   04be238e.../172.17.8.101

Deployed units:
UNIT                  MACHINE                    ACTIVE  SUB
celix@1.service       be956748.../172.17.8.103   active  running
celix@2.service       04be238e.../172.17.8.101   active  running
felix@1.service       5ed2251e.../172.17.8.105   active  running
felix@2.service       a5097953.../172.17.8.104   active  running
provisioning.service  04be238e.../172.17.8.101   active  running
```

This concludes our last demonstrator scenario. You can play with the `inaetics_fleet_manager` script to start more or fewer agents. By default, you can start up to 4 Celix and Felix agents, but if you start more compute resources than the default of five, then you can add even more agents. If you want to see more details of the various parts of the INAETICS application, you can also take a look at the Apache ACE server. For this, you need to find the node on which the `provisioning.service` is running (`172.17.8.101` in our example) and access its web UI by pointing your web browser to: `http://172.17.8.101:8080/`. You are presented with a login dialog, for which you can use the following credentials:

user name	d
password	f

You can play around with assigning various features to different targets. Note that it is at the moment not possible to associate Felix features to Celix agents and the other way around.
For more information on how to work with Apache ACE, see its users guide.

Known issues

This demonstrator is created to demonstrate the scalability and resilience of an INAETICS system. It is not meant for production purposes and as such has a couple of issues outlined in the following list:

1. the compute resources or workers are expected to run continuously. In case they are stopped (for example, using `vagrant halt`), then the information stored in Etcd is no longer correct and needs to be purged **prior** to restarting the compute resources. For this, a simple script is available in the bin directory of the `demonstrator-cluster` repository, which should be invoked as:

```
$ sh bin/purge_etcd_discovery.sh http://172.17.8.2:4001 inaetics-cluster-1
Cleaning discovered nodes for http://172.17.8.2:4001 from inaetics-cluster-1...
...
Deleting '_state'...
```

after this script is invoked, you can start the compute resources again;
2. sometimes, the scaling out and failover scenarios take a long time before the system is in a steady state again. It will reach this situation, but can take up to a couple of minutes.

Additional debug tips and tools can be found in the README file of the `demonstrator-cluster` repository.

Summary

In this demonstrator, we have shown how an evolvable polyglot application can be run as a distributed application on top of a cluster. We have described how to scale out in case of overloads by using Fleet to start new agents, how fail-over of failing parts is handled automatically and how dynamic reconfiguration of agents can be performed.