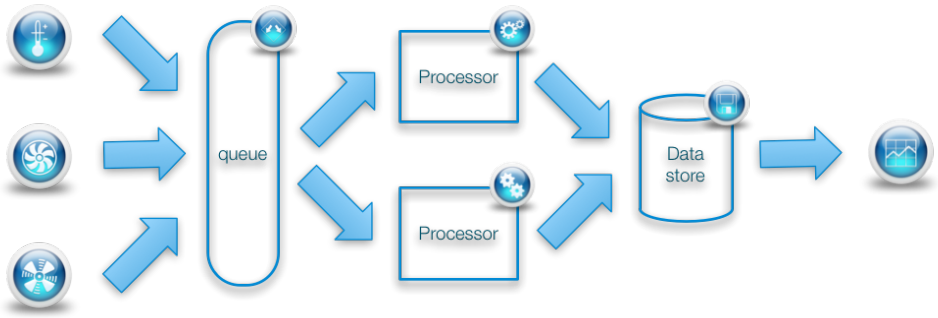
Installation and setup guide of 1.1 demonstrator

version 2.0, last modified: 2015-09-23

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 six nodes on a single machine, we show how you can create evolvable systems using Kubernetes, 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. In the appendix Under the Hood additional technical information is presented

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. The statistics of each element are available from a webpage, further the browser can be used to instrument the number of processors in the demonstrator and to instrument the sample rate of the producers. With these 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.

In an INAETICS system is a coordinator that tries to map application requirements on available resources., hereby the health of the system is monitored to decide if a re-mapping is required. This part is still under development. To be able to demonstrate the coordination concept an auto-scaler is added to the demonstrator. This auto-scaler scales the number of processors up and down dependent on the QOS of the queue. In this case, the QOS is defined as the fill rate of the queue.

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 Kubernetes 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:

	Supported Configurations		
Hardware	1 laptop (Core i7, 16GB RAM)	1 laptop (Core i7, 16GB RAM)	laptop (Core i7, 8 GB RAM)
OS	OSX 10.10	CentOS 7.0.1406	Fedora 22
CoreOS	github.com/coreos/coreos-vagrant (CoreOS Alpha 815.0.0) (Vagrant 1.7.2) (VirtualBox 4.3.26r98988)	github.com/coreos/coreos-vagrant (CoreOS Alpha 815.0.0) (Vagrant 1.6.5) (VirtualBox 4.3.20)	github.com/coreos/coreos-vagrant (CoreOS Alpha 815.0.0) (Vagrant 1.7.4) (VirtualBox 5.0.6)
Etcd	Etcd v2.0	Etcd v2.0	Etcd v2.0
Fleet	Fleet v0.9.0	Fleet v0.9.0	Fleet v0.9.0
Docker	docker 1.7.1	docker 1.7.1	docker 1.8.2

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

Prerequisites

The demonstrator is developed and tested on machines with 16 GB of internal memory and a Core i7 CPU capable of running virtual machines. Given that this configuration uses VirtualBox to instantiate six virtual machines on a single machine, each of them requiring about 1 to 1.5 GB of memory to operate correctly, it means that you need around 9 to 10 GB of **free** memory minimally available on your machine to successfully run this demonstrator.

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.

Note: 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
```

Note: On Fedora22 the packages for virtualbox and vagrant need to be installed manually (sudo dnf install <URL>)

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

```
$ git clone --branch v1.1.0 https://github.com/INAETICS/kubernetes-demo-cluster.git
Cloning into 'demonstrator-cluster'...
...
$ cd kubernetes-demo-cluster
$ export INAETICS_HOME=`pwd`
$ git submodule init && git submodule update
...
Submodule path 'docker-images/provisioning': checked out 'xyz'
```

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

- Controller, virtual machine that runs the Kubernetes master
- Cluster, virtual machines that run the cluster machines

To prevent that all virtual machines download needed packages when started, this needs to be done manually. Kubernetes needs a few docker images: flannel and pause. These are only available from insecure docker registries. For that reason, the docker configuration file (normally /etc/sysconfig/docker) needs to specify these insecure registries.

```
Edit /etc/sysconfig/docker:
Add the following option: OPTIONS=-D --insecure-registry quay.io --insecure-registry
grc.io
```

Note: for docker 1.8.2 the above was not needed, instead the executing user needs to be added to the docker group

The actual downloads are scripted:

```
$ cd Controller
$ sh bin/initial_download.sh
```

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 Kubernetes Master VM

The Kubernetes Master VM provides the basic infrastructure for the compute resources, such as an Etcd Leader, fleet units to bootstrap Kubernetes and the Kubernetes controller. The Etcd cluster is used to store and share information about the state of the system between all compute resources. On each node, docker loads the needed docker images into its local cache. This prevents problems with an overload of a single docker registry in the virtual setup. To start the Kubernetes Master VM, we do the following:

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

After a little while the Kubernetes Master VM is up and running which can be verifying the following URL (the port number is of the Kubernetes API server):

```
$ curl http://172.17.8.20:10260/api
{
  "versions": [
    "v1"
  ]
}$
```

Starting the Cluster 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/Cluster
$ vagrant up
Bringing machine 'node-1' up with 'virtualbox' provider
...
```

You might need to wait a while before all workers are up and running. It can take 5 to 10 minutes to get all workers correctly up and running depending on the speed of your host machine!

Once all five workers are booted and ready, we have the demonstrator up and running.

It can happen sometimes that the compute resources aren't assigned the right IPv4 address, which causes the demonstrator application to fail. If the IPv4 address of the Kubernetes Master VM is **not** `172.17.8.20`, 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`). You might also need to remove the VirtualBox network interface.

The INAETICS demonstrator

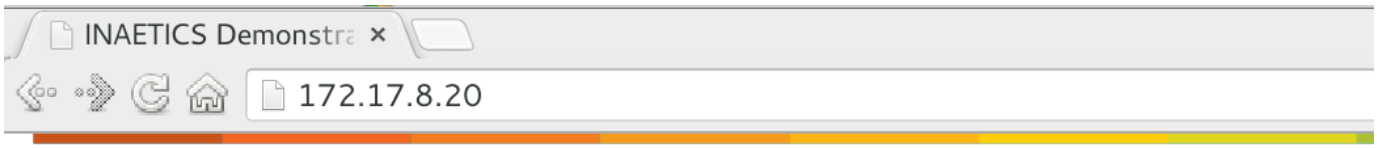
As described in the overview section, the demonstrator application consists of several parts that together form a distributed application that is running on the compute resources. To distribute the various parts across the compute resources Kubernetes is used. Kubernetes has the concept of a master node were a scheduler, controller and apiserver are running. On the cluster nodes a Kubernetes machine manager (kubelet) and a proxy is running. Kubernetes uses JSON formatted files to instrument the scheduler. The JSON files describe the grouping of docker containers in so-called PODs. These PODs can be replicated with a replication controller. Also the services offered by a POD can be proxied on to the master node.
The Kubernetes controller determines what PODS and services shall be started in the cluster. The Kubernetes scheduler checks the resources that are available in the cluster and determines where each POD is running. The Kubelet finally starts the POD.

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;
- a single producer (Felix agent) which is instrumented to run at 5% of its maximum sample rate
- a single queue (Felix agent)
- a single datastore which also runs the coordinator and the webserver (Felix agent)
- a dynamic number of processors (the coordinator alternately starts a Felix agent or a Celix agent)

Starting the demonstrator application

The demonstrator is automatically started by Kubernetes. The viewer can be accessed using a web browser at address 172.17.8.20. The main page of the demonstrator is shown below

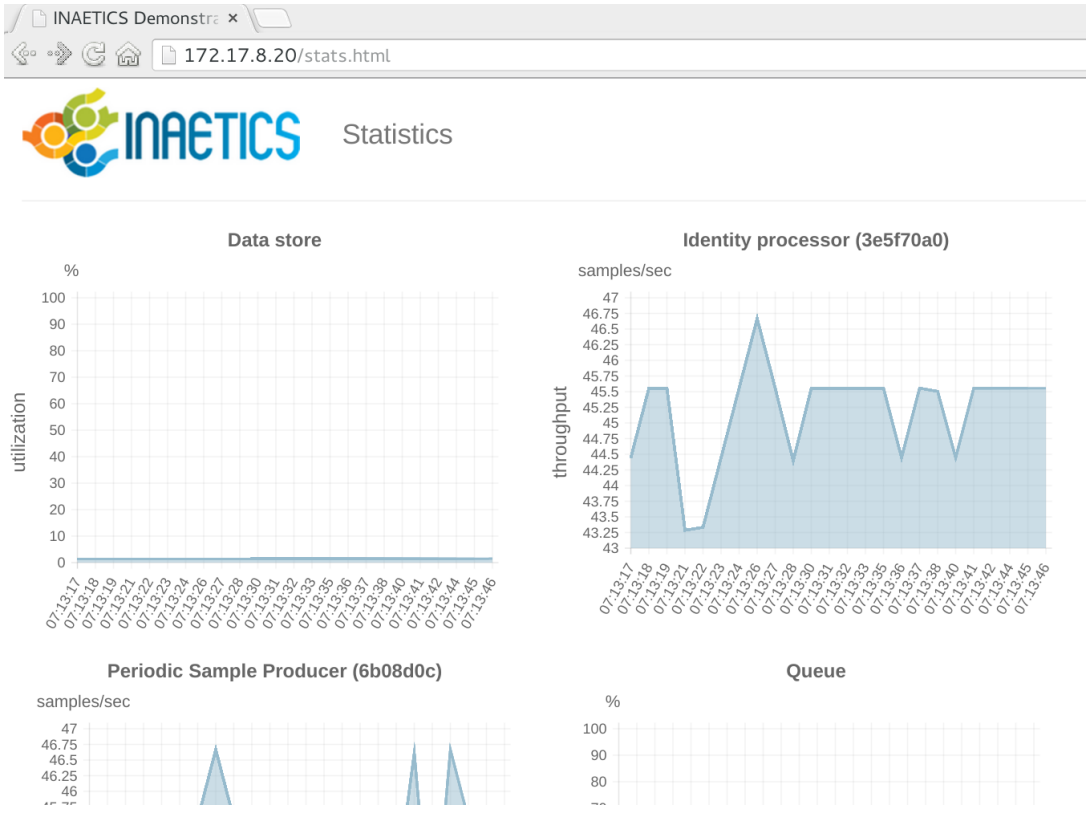


INAETICS demonstrator

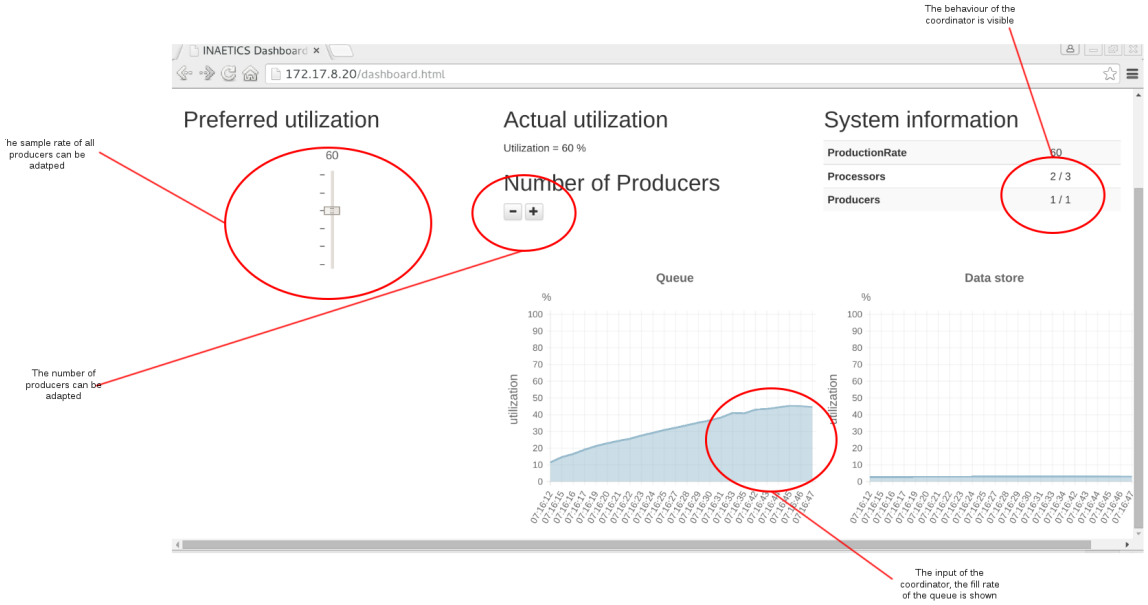
This is the landing page for the INAETICS demonstrator. You can go to the following subpages:

1. An overview with all statistics of the running system;
2. the dashboard provding a high-level view of the system.

The "all statistics" subpage of the main page is mainly for developing. It shows the statistics (number of samples) of each component running in the system.



Most interesting from a user point of view is the dashboard. It provides overview and control options



Scaling out the demonstrator

The number of samples that all producers generate can be adapted with a slider as percentage. At maximum, the number of samples per producer is approximately 400 per producer (depends on the load of the system you are running on). Also the number of producers can be increased or declined to handle change in environment.

The result of these actions will be a grow or decrease of the fill level of the queue. The coordinator running in the system uses the change in fill level during a period of ten seconds as application health metric and adds or removes processors in the system. In the shown dashboard the number of processors is shown as 2 / 3. This indicates that at the moment 2 processors are active in the system and that the coordinator has requested a third processor.

Handling fail-over and dynamic reconfiguration

To simulate a fail-over scenario, we need to "pull the plug" on one of the agents. Suppose we terminate the agent running the queue service, this causes both the producers and processors to stop functioning. Fortunately, Kubernetes ensures that this service is restarted after sudden termination causing the demonstrator to function properly again. To simulate the termination of the queue service, we SSH into the Controller.

```
$ cd $INAETICS_HOME/Controller
$ vagrant ssh
```

In the Controller the Kubernetes scheduler and api-server are running. To access Kubernetes a client application kubectl is used. It expects an environment variable with the address of the kubernetes master. The "get pods -o wide" arguments shows the Kubernetes PODS that are running on which node.

```
core@controller ~ $ export `cat /etc/kubernetes.env `
core@controller ~ $ kubectl get pods -o wide
NAME                                          READY     STATUS    RESTARTS   AGE       NODE
ace-provisioning-controller-xgcrp             1/1       Running   0          19m       172.17.8.31
inaetics-datastore-viewer-controller-6u6wo    1/1       Running   0          19m       172.17.8.32
inaetics-processor-celix-controller-b72gd     1/1       Running   0          19s       172.17.8.34
inaetics-processor-controller-dhsjb           0/1       Running   0          9s        172.17.8.33
inaetics-processor-controller-kz95r           1/1       Running   0          1m        172.17.8.35
inaetics-producer-controller-8d0qk            1/1       Running   0          19m       172.17.8.31
inaetics-queue-controller-65w74               1/1       Running   0          19m       172.17.8.33
core@controller ~ $
```

To terminate the Docker container ssh into the cluster node (node-1 is 172.17.8.31, node-2 is 172.17.8.32 etc.) and use the docker command-line to determine the container id of the queue container.

```
core@node-3 ~ $ docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS
        PORTS                    NAMES
4cee1adc34b8        inaetics/felix-agent                   "/tmp/node-agent.sh"     About an hour ago   Up About an
 hour                            k8s_inaetics-queue-container.859380f1_inaetics-queue-controller-65w74_default_0
046e6b4-6279-11e5-9cfd-08002744f5bd_4f0c13cb
ec75937cd8da        gcr.io/google_containers/pause:0.8.0   "/pause"                 About an hour ago   Up About an
 hour    0.0.0.0:8080->8080/tcp   k8s_POD.2c52e959_inaetics-queue-controller-65w74_default_0046e6b4-6279-11e5-9cf
d-08002744f5bd_1a010074
core@node-3 ~ $ docker kill 4cee1adc34b8
4cee1adc34b8
```

On the dashboard you can notice that no new statistics information of the queue is retrieved. It takes some time for Kubernetes to detect the missing POD, next the agent needs to register itself in the Cluster again and the provisioning server needs to re-install software on it.

Ater some time the statistics and queue are running as before. 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 node-4
```

The result of this is that Kubernetes 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 `kubectl get pods command.`:

This concludes our last demonstrator scenario.

Known Issues

- **vboxnet0 interface not removed**
  If you have already installed and used VirtualBox before, it might be that its network adapter (`vboxnet0`) is not correctly configured. This is needed to establish proper communication between the various virtual machines that make up the cluster. In this case, you need to remove it prior to continuing:

  ```
  $ vboxmanage hostonlyif remove vboxnet0
  ```

  You might also need to run this command after stopping and restarting the machines.

- **discovery fails with old etcd state**

The used distributed key/value store ETCD maintains a snapshot of its state on disk. When a machine is halted with the command "vagrant halt" this state will be stored in the virtual machine state. At the next start-up this information will be restored, but this causes our discovery mechanisms to fail. Therefore, we always use

```
$ vagrant halt
$ vagrant destroy
```

- vagrant up fails
  On Fedora22 sometimes libvirt is chosen as default provider, not clear yet why.

  Add the following environment variable to ~/.bashrc

  ```
  export VAGRANT_DEFAULT_PROVIDER=virtualbox
  ```

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, how fail-over of failing parts is handled automatically and how dynamic reconfiguration of agents is performed.

Appendix Under the Hood

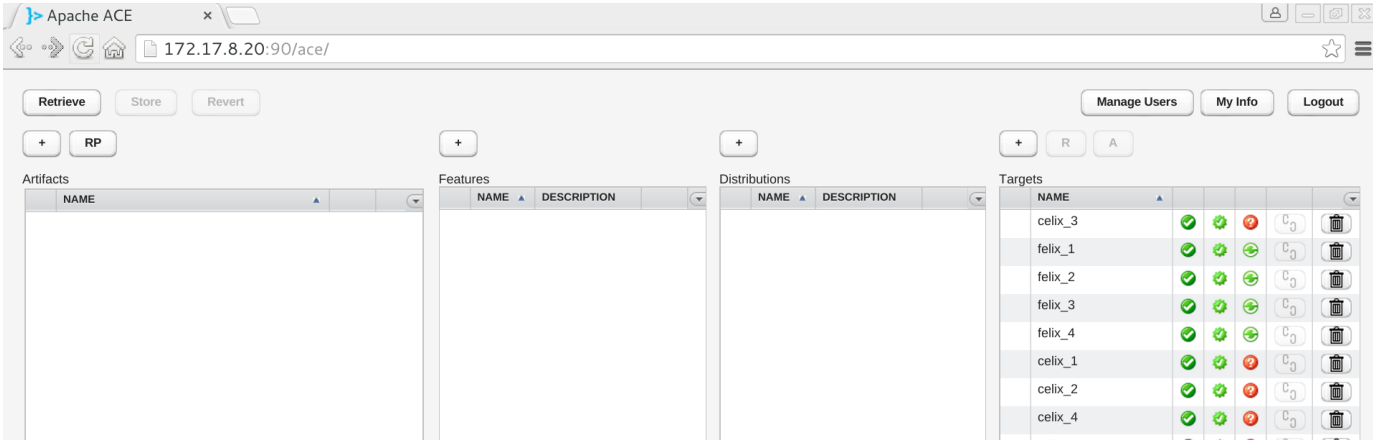
For those interested in the technical details we will present some additional information.

Software provisioning

An interesting part is the software provisioning server Apache ACE. It contains information on the OSGi software bundles present in the system and how these have to be mapped on the available computing resources. For this, you can access the viewer of Apache ACE at the following URL:

172.17.8.20:90 (a Kubernetes service is running that proxies port 8080 of the ACE server on port 90 of the Kubernetes master). 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. The available bundles and the default mapping are mounted in the node provisioning agent. These are available at $INAETICS_HOME/Controller/inaetics_demo/node-provisioning/bundles.

For more information on how to work with Apache ACE, see its users guide.

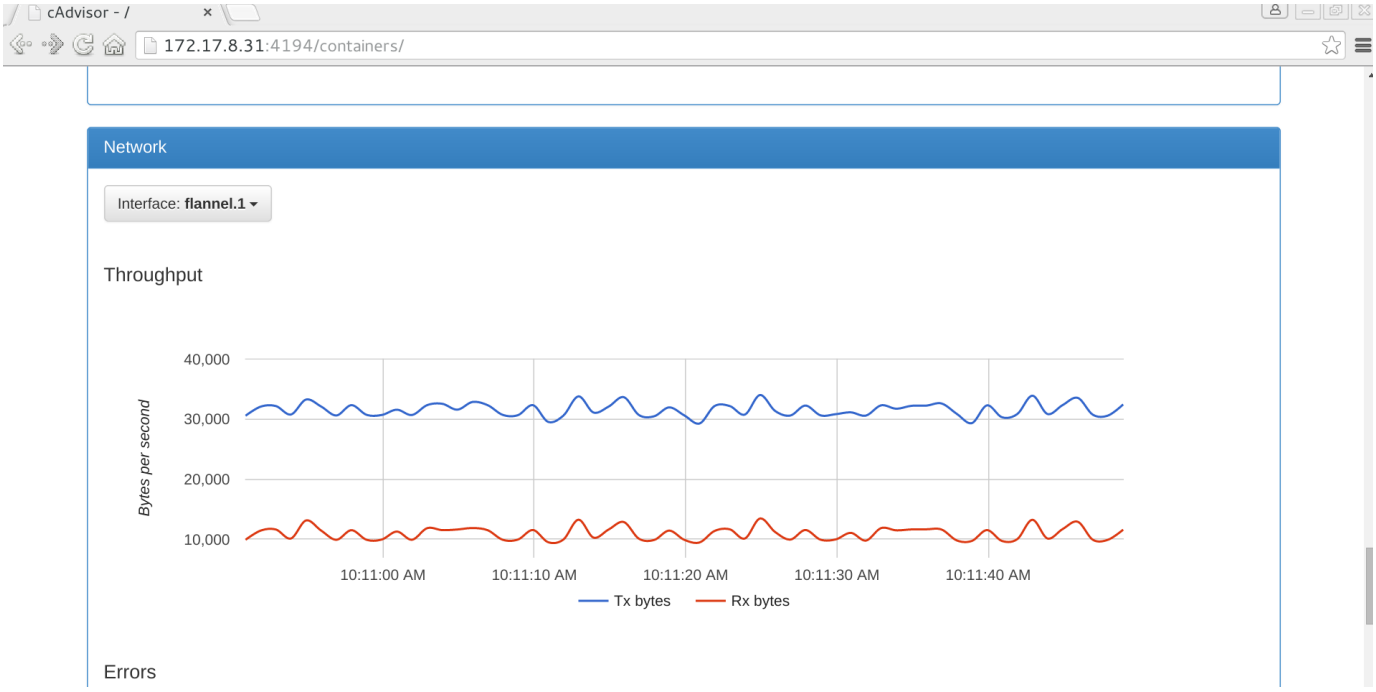
Networking

A very good overview of the networking problems with cluster environments is given by: Kubernetes Networking Model

In the demonstrator we selected Flannel, an overlay network, as solution for "one IP per POD". See Introducing Flannel As protocol to transport the overlay packets a VXLAN backend is used, see FLANNEL Readme

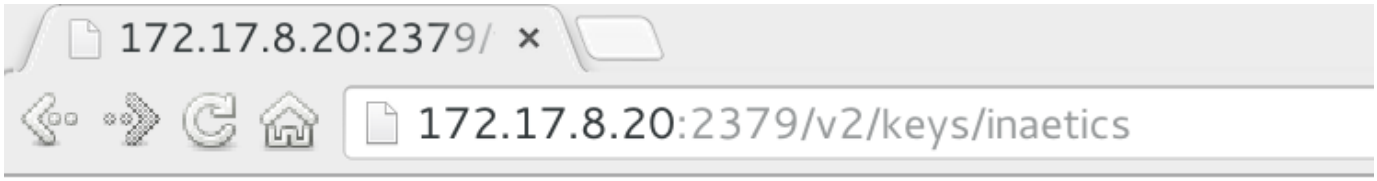
Container Monitoring

Kubernetes uses cAdvisor under the hood to obtain statistics of the running docker containers. It is only available for the Cluster nodes. You can check the available information at URL: 172.17.8.31:4194



Discovery

For the different discovery mechanisms needed in the demonstrator the distributed key-value store ETCD is used. It is part of CoreOS. So see the discovery information present check the following URL 172.17.8.20:2379/v2/keys/inaetics




```
{
    action: "get",
  - node: {
        key: "/inaetics",
        dir: true,
      - nodes: [
          - {
                key: "/inaetics/node-provisioning-service",
                dir: true,
                modifiedIndex: 1200,
                createdIndex: 1200
            },
          - {
                key: "/inaetics/node-agent-service",
                dir: true,
                modifiedIndex: 1202,
                createdIndex: 1202
            },
          - {
                key: "/inaetics/discovery",
                dir: true,
                modifiedIndex: 1287,
                createdIndex: 1287
            },
          - {
                key: "/inaetics/wiring",
                dir: true,
                modifiedIndex: 1288,
                createdIndex: 1288
            }
        ],
        modifiedIndex: 1200,
        createdIndex: 1200
    }
}
```

Logging

Logging of Celix containers is possible with the "docker logs <container_id>" command, e.g. for a Celix processor

```
core@node-4 ~ $ docker ps
CONTAINER ID        IMAGE                                  COMMAND                CREATED             STATUS
        PORTS                    NAMES
b4177abe4e64        inaetics/celix-agent                   "/tmp/node-agent.sh    12 seconds ago      Up 12 secon
ds                               k8s_inaetics-processor-celix-container.680284ec_inaetics-processor-celix-contro
ller-i2tyv_default_1ba7e293-6295-11e5-8a70-08002744f5bd_558148e5
81fa5212c3aa        gcr.io/google_containers/pause:0.8.0   "/pause"               12 seconds ago      Up 12 secon
ds       0.0.0.0:9082->8080/tcp   k8s_POD.3180e95c_inaetics-processor-celix-controller-i2tyv_default_1ba7e293-629
5-11e5-8a70-08002744f5bd_b3fbd7f6
core@node-4 ~ $ docker logs b4177abe4e64
[DEBUG] -> etcd/values - args: /inaetics/node-provisioning-service
[DEBUG] -> etcd/keys - args: /inaetics/node-provisioning-service 172.17.8.20:2379
[DEBUG] -> etcd/value - args: /inaetics/node-provisioning-service/2a48d09ef3544c52da019cff17e7d56b464ed59239386f8
43951ce3756f7ac0b 172.17.8.20:2379
Provisioning service changed:  -> 10.1.44.2:8080
Starting agent...
[INFO] CELIX Configuration
[INFO] ============================================
[INFO] RSA IP s : 10.1.62.3
[INFO] DISCOVERY_ETCD_SERVER_IP : 172.17.8.20
[INFO] DISCOVERY_ETCD_SERVER_PORT : 2379
[DEBUG] celix
agent running with provisioning 10.1.44.2:8080
[DEBUG] -> etcd/putTtl - args: /inaetics/node-agent-service/celix_3 10.1.62.3:8080 15
[DEBUG] 10.1.62.3:8080
[INFO] Pair </inaetics/node-agent-service/celix_3,10.1.62.3:8080> stored in etcd
Will update in 10 seconds...
LogWriter: BUNDLE_EVENT_STARTED from apache_celix_log_service
LogWriter: BUNDLE_EVENT_RESOLVED from apache_celix_log_writer
LogWriter: BUNDLE_EVENT_STARTED from apache_celix_log_writer
ERROR: File I/O exception [70008]: "Failed to delete tree"
       at deploymentAdmin_deleteTree(/tmp/celix/deployment_admin/private/src/deployment_admin.c:440)
LogWriter: BUNDLE_EVENT_INSTALLED from apache_celix_remote_shell
```

For Felix agents besides the docker logs command, additional information can be retrieved using the following commands

```
$ cd $INAETICS_HOME/Controller
$ vagrant ssh
core@controller ~ $export `cat /etc/kubernetes.env`
core@controller ~ $kubectl get pods -o json <name>
Check the output and determine the podIP <POD IP> where the POD is running
core@controller ~ ncat --telnet <POD IP> 2019
------------------------------------
Welcome to the Apache Felix Gogo
g! log debug
log 1
```

Gelderland & Overijssel
Gebundelde Innovatiekracht

Europese Unie
Europees Fonds voor Regionale Ontwikkeling

Hier wordt geïnvesteerd in uw toekomst!