Motey Documentation

Release 0.0.1

Markus Paeschke

Aug 30, 2017

Contents:

CHAPTER 1

Documentation Core

class motey.core.**Core**(*logger*, *capability_repository*, *nodes_repository*, *valmanager*, *inter_node_orchestrator*, *communication_manager*, *capability_engine*, *as_daemon=True*)

This module provides the core functionality of Motey. It can be executed as a daemon service or can be executed in foreground. It will start an API webserver and a MQTTServer which can be configured via the config.ini file. The core will also start all the necessary components like the VALManager, the InterNodeOrchestrator and the HardwareEventEngine. After it is started via self.start() it will be executed until self.stop() is executed.

restart()

Restart the core.

run()

The method is the main app loop. It starts the `Communication Manager Components` and it will be executed until `self.stop()` is executed.

start()

Start the core component. At first clean up config if necessary. If self.as_daemon is set to True, the component will be started as a daemon services. It will use the path to the pid which is configured in the config.ini. If self.as_daemon is set to False, the component will be executed in foreground.

startup_clean()

Clean up the capability and node database to remove old entries.

stop()

Clean up the started services. It will stop the `Communication Manager Components`. Finally it stops the daemon if self.as_daemon is set to True.

CHAPTER 2

Documentation cli

CHAPTER 3

Documentation Communication

class motey.communication.apiserver.**APIServer**(*logger*, *host='127.0.0.1'*, *port=5023*)

Starts a Flask webserver which acts as an REST API to control the Motey service. The webserver runs in a separate thread and will not block the main thread.

check_heartbeat()

Method to get the heartbeat. Checks if all the requirements for a healthy webserver are fulfilled. :return: True if the service is in a healthy state, otherwise False.

configure_url()

Adds all the configured api endpoints.

is_running()

Checks if the webserver is still running. :return: True if the server is running, otherwise False.

run_server()

Starts the server and add an info to the logs, that the webserver is started.

start()

Starts the execution thread.

stop()

Stops the webserver and add an info the logs, that the webserver is stopped.

class motey.communication.communication_manager.**CommunicationManager**(*api_server*, *mqtt_server*, *zeromq_server*)

This class acts as an facade for the communication endpoints like the `MQTT server`, the `API server` and the `ZeroMQ server`. It covers all method calls and can start and stop the mentioned components.

after_connect_callback()

Will be called after the MQTTServer has established a connection to the broker. Send out a request to fetch the ip from all existing nodes.

deploy_image(*image*)

Facades the `ZeroMQServer.deploy_image()` method. Will deploy an image to the node stored in the `Image.node` attribute.

Parameters **image** (motey.models.image.Image) – Image to be deployed.

Returns the id of the deployed image or None if something went wrong.

request_capabilities(*ip*)
Facades the ZeroMQServer.request_capabilities() method. Will fetch the capabilities of a specific node and will return them.

Parameters **ip** (*str*) – The ip of the node to be requested.

Returns the capabilities of a specific node

request_image_status(*image*)
Facades the ZeroMQServer.request_image_status() method. Request the status of an specific image instance or None if something went wrong.

Parameters **image** (motey.models.image.Image) – Image to be used to get the status.

Returns the status of the image or None if something went wrong

start()
Start all the connected communication components.

stop()
Stop all the connected communication components. Will send out a mqtt message to remove the current node.

terminate_image(*image*)
Facades the ZeroMQServer.terminate_image() method. Will terminate an image instance.

Parameters **image** (motey.models.image.Image) – the image instance to be terminated

class motey.communication.mqttserver.**MQTTServer**(*logger*, *nodes_repository*, *host='127.0.0.1'*, *port=1883*, *username=None*, *password=None*, *keepalive=60*)
MQTT server to register and unregister adjacent fog nodes. The webserver runs in a separate thread and will not block the main thread.

handle_nodes_request(*client*, *userdata*, *message*)
Define the node request callback implementation. Will execute the callback of the request to fetch the ip from all existing nodes.

Parameters

- **client** – the client instance for this callback
- **userdata** – the private user data as set in Client() or userdata_set()
- **message** – the data which was send

handle_on_connect(*client*, *userdata*, *flags*, *resultcode*)
Define the connect callback implementation. If the client is connected to the MQTT broker, the nodes will be registered and the _after_connect method will be executed.

flags is a dict that contains response flags from the broker:

flags['session present'] - this flag is useful for clients that are using clean session set to 0 only. If a client with clean session=0, that reconnects to a broker that it has previously connected to, this flag indicates whether the broker still has the session information for the client. If 1, the session still exists.

The value of rc indicates success or not: 0: Connection successful 1: Connection refused - incorrect protocol version 2: Connection refused - invalid client identifier 3: Connection refused - server unavailable 4: Connection refused - bad username or password 5: Connection refused - not authorised 6-255: Currently unused.

Parameters

- **client** – the client instance for this callback
- **userdata** – the private user data as set in Client() or userdata_set()
- **flags** – response flags sent by the broker
- **resultcode** – the connection result

handle_on_disconnect(*client*, *userdata*, *resultcode*)
Define the disconnect callback implementation.

Parameters

- **client** – the client instance for this callback
- **userdata** – the private user data as set in Client() or userdata_set()
- **resultcode** – the connection result

handle_register_node(*client*, *userdata*, *message*)
Define the register new node callback implementation. Adds the new node to the `NodesRepository`.

Parameters

- **client** – the client instance for this callback
- **userdata** – the private user data as set in Client() or userdata_set()
- **message** – the data which was send

publish_new_node(*ip=None*)
Publish the info that a new node is available to the all subscribers. If the `ip` is none, nothing will be send.

Parameters **ip** – The IP address of the new node. Default is None.

publish_node_request(*ip=None*)
Publish the request to fetch the ip from all existing nodes.

Parameters **ip** – the own ip to let the other nodes know where the request cames from.

register_routes()
Adds all the configured MQTT endpoints.

remove_node(*ip=None*)
Remove a specific node and publish it to all subscribers. If the `ip` is none, nothing will be send.

Parameters **ip** – The IP address of the new node. Default is None.

run_server()
Starts the server and add an info to the logs, that the MQTT server is started. Also adds an error message to the logs if the broker is not available.

start()
Starts the execution thread.

stop()
Stops the MQTT server and add an info the logs, that the server is stopped.

class motey.communication.zeromq_server.**ZeroMQServer**(*logger*, *valmanager*, *capability_repository*)

ZeroMQ server to communicate with adjacent fog nodes and to reply to requests. The different listeners will be executed in a separate thread and will not block the main thread.

deploy_image(*image*)

Will deploy an image to the node stored in the Image.node attribute.

Parameters **image** (motey.models.image.Image) – Image to be deployed.

Returns the id of the deployed image or None if something went wrong.

request_capabilities(*ip*)

Method to request all capabilities from another node. Will request via the *ZeroMQ.REQ* pattern. After the request is send, the method will wait for the response.

Parameters **ip** – the IP address of the node to request the capabilities

Returns the capabilities as a JSON object

request_image_status(*image*)

Request the status of an specific ImageState instance or ImageState.ERROR if something went wrong.

Parameters **image** (motey.models.image.Image) – Image to be used to get the status.

Returns the ImageState or ImageState.ERROR if something went wrong

start()

Starts the listening on a given port. This method will be executed on a separate thread.

stop()

Should be executed to clean up the capability engine

terminate_image(*image*)

Will terminate an image instance.

Parameters **image** (motey.models.image.Image) – the image instance to be terminated

CHAPTER 4

Documentation Communication - API Routes

class motey.communication.api_routes.capabilities.**Capabilities**

This REST API endpoint for capability handling. A capability is basically a capability for the whole node. New capabilities can be added or deleted via this endpoint or a list with the existing ones can be fetched.

delete()

Remove a list of capabilities or at least a single one from the node. The content type of the request must be application/json, otherwise the request will fail.

Returns 201 - Created if at least one capability was removed, 304 - Not Modified if non of the sent capabilities was removed because they don not exists or 400 - Bad Request if the wrong content type was sent or the json does not match the motey.models.schemas.capability_schema.

get()

Returns a list off all existing capabilities of this node.

Returns a JSON object with all the existing capabilities of this node

put()

Add a list of new capabilities or at least a single one to the node. The content type of the request must be application/json, otherwise the request will fail.

Returns 201 - Created if at least one capability was added, 304 - Not Modified if non of the sent capabilities was added or 400 - Bad Request if the wrong content type was sent or the json does not match the motey.models.schemas.capability_schema.

class motey.communication.api_routes.nodestatus.**NodeStatus**

Give information about the current hardware usage of the node. This includes the cpu, memory and disk usage.

get()

Returns the currant hardware usage of the node.

Returns a json object with the current hardware usage of the node.

get_average_cpu(*loops=5*)

Helper method to get an average cpu usage.

Parameters **loops** – the number of iterations to sum up the avarage cpu usage.

Returns The avarage cpu usage as a float.

class motey.communication.api_routes.service.**Service**

This REST API endpoint for getting service informations and also let the client upload a YAML file to the node. A service contain all the running images.

delete()

DELETE endpoint. Receive the YAMl file and validates them. The content type of the request must be application/x-yaml, otherwiese the request will end up in a HTTP status code 400 - Bad Request. If validation was successful the given service will be terminated by the InterNodeOrchestrator.

Returns HTTP status code 201 - Created, if the request is successful, otherwise 400 - Bad Request.

get()

Returns a list off all existing capabilities of this node.

Returns a JSON object with all the existing capabilities of this node

post()

POST endpoint. Receive the YAMl file and validates them. The content type of the request must be application/x-yaml, otherwiese the request will end up in a HTTP status code 400 - Bad Request. If validation was successful the given service will be instantiate by the InterNodeOrchestrator.

Returns HTTP status code 201 - Created, if the request is successful, otherwise 400 - Bad Request.

CHAPTER 5

Documentation Labeling Engine

class motey.capabilityengine.capability_engine.**CapabilityEngine**(*logger*, *capability_repository*, *communication_manager*)

This module provides a connection endpoint for third party apps like the hardware layer to add new capabilities.

parse_capability(*data*)

Parse a JSON string with capability data and transform them into an array with capability models

Parameters

- **data** – the data that should be parsed
- **data** – str

Returns an array with capability models or an empty array if something went wrong

perform_add_capability(*data*)

Adds a capability entry to the database.

Parameters **data** (*str*) – the capability entry which should be added. The entry must match the *motey.models.schemas.capability_json_schema*

perform_remove_capability(*data*)

Removes a capability entry from the database.

Parameters **data** (*str*) – the capability entry which should be removed. The entry must match the *motey.models.schemas.capability_json_schema*

start()

Subscibes to the capability event stream.

stop()

Should be executed to clean up the capability engine

CHAPTER 6

Documentation Models

class motey.models.image.**Image**(*name*, *engine*, *id=''*, *parameters={}*, *capabilities={}*, *node=None*)

Model object. Represent an image. An image can have execution parameters, required capabilities and the node where it is executed. All of them are optional.

static transform(*data*)

Static method to translate an image dict into a image model.

Parameters **data** (*dict*) – a dict with image data

Returns the translated image model, None if something goes wrong

motey.models.**schemas**

alias of *motey.models.schemas*

class motey.models.service.**Service**(*service_name*, *images*, *id='fab3aecfa54843fa88ac0320816f5570'*, *state=0*, *state_message=''*)

Model object. Represent a service. A service can have multiple states, action types and service types.

static transform(*data*)

Static method to translate the service dict data into a service model.

Parameters **data** (*dict*) – service dict to be transformed

Returns the translated service model, None if something went wrong

class motey.models.systemstatus.**SystemStatus**

Model which represents the status of the whole system. Possible status values are:

- used_memory
- used_cpu
- network_tx_bytes
- network_rx_bytes

class motey.models.valinstancestatus.**VALInstanceStatus**

Model which represents the status of an VAL instance. Possible status values are:

- name

- image_name
- created_at
- status
- ip
- used_memory
- used_cpu
- network_tx_bytes
- network_rx_bytes

CHAPTER 7

Documentation Orchestrator

class motey.orchestrator.inter_node_orchestrator.**InterNodeOrchestrator**(*logger*, *valmanager*, *service_repository*, *capabilityrepository*, *node_repository*, *communication_manager*)

This class orchestrates services. It will start and stop virtual instances of images defined in the service. It also can communicate with other nodes to start instances there if the requirements does not fit with the possibilities of the current node.

compare_capabilities(*needed_capabilities_list*, *node_capabilities_dict*)

Compares two dicts with capabilities.

Parameters

- **needed_capabilities_list** (*list*) – the capabilities to compare with
- **node_capabilities_dict** (*list*) – the capabilties to check

Returns True if all capabilities are fulfilled, otherwiese False

deploy_service(*service*)

Deploy all images of a service to the related nodes.

Parameters **service** (motey.models.service.Service) – the service which should be deployed

find_node(*image*)

Try to find a node in the cluster which can be used to deploy the given image.

Parameters **image** (motey.models.image.Image) – the image to be used

Returns the IP of the node to be used or None if it does not found a node which fulfill all capabilities

get_service_status(*service*)
Returns the service status.

Parameters **service** (motey.models.service.Service) – the service which should be used

Returns the status of the service

instantiate_service(*service*)
Instantiate a service.

Parameters **service** (motey.models.service.Service) – the service to be used.

terminate_service(*service*)
Terminates a service.

Parameters **service** (motey.models.service.Service) – the service to be used.

CHAPTER 8

Documentation Repositories

class motey.repositories.base_repository.**BaseRepository**
Base repository to wrapp database handling.

all()
Return a list of all existing entries in the database.

Returns a list of all existing entries.

clear()
Remove all entries from the database.

class motey.repositories.capability_repository.**CapabilityRepository**
Repository for all capability specific actions.

add(*capability*, *capability_type*)
Add a new capability to the database.

Parameters

- **capability** – the capability to be added.
- **capability_type** – the capability type of the capability.

has(*capability*)
Checks if the given capability exist in the database.

Parameters **capability** – the capability to search for.

Returns True if the lable exists, otherwise False

remove(*capability*, *capability_type=None*)
Remove a capability from the database.

Parameters

- **capability** – the capability to be removed.
- **capability_type** – optional. The capability must also matche the capability type to be removed.

remove_all_from_type(*capability_type*)
Remove a capabilities with a specific capability type.

Parameters **capability_type** – the capability type where all related capabilitys should be removed.

class motey.repositories.nodes_repository.**NodesRepository**
Repository for all node specific actions.

add(*ip*)
Add a new node to the database if they not exist yet.

Parameters **ip** – the ip of the new node.

has(*ip*)
Checks if the given ip exist in the database.

Parameters **ip** – the ip of the node to search for.

Returns True if the node exists, otherwise False

remove(*ip*)
Remove a node from the database.

Parameters **ip** – the ip of the node to be removed.

class motey.repositories.service_repository.**ServiceRepository**
Repository for all service specific actions.

add(*service*)
Add a new service to the database if they not exist yet.

Parameters **service** (*dict*) – a service model to be stored

has(*service_id*)
Checks if the given id exist in the database.

Parameters **service_id** – the id of the service to search for.

Returns True if the service exists, otherwise False

remove(*service_id*)
Remove a service from the database.

Parameters **service_id** – the id of the service to be removed.

update(*service*)
Update a service in the database.

Parameters **service** (*dict*) – a service model to be updated

CHAPTER 9

Documentation Utils

motey.utils.heartbeat.**check_heartbeat**()
The endpoint himself. Will be executed if the url is requested.

Returns 204 - No Content, if the node is up and running and all the registered callbacks requirements are fulfilled, otherwise 400 - Bad Request.

motey.utils.heartbeat.**register_callback**(*callback*)
Helper method to register a new callback.

Parameters **callback** – callback to be added to the queue

motey.utils.heartbeat.**register_heartbeat**(*flask_app*)
Helper method to register the heartbeat to the `Flask` webserver. Does not works with other webservers.

Parameters **flask_app** – the `Flask` instance.

class motey.utils.logger.**Logger**
Wrapper to configure the LogbookLogger.

motey.utils.network_utils.**get_own_ip**()
Reads out the ip of the device and return them.

Returns the ip of the device

CHAPTER 10

Documentation Virtualization Abstraction Layer (VAL)

class motey.val.valmanager.**VALManager**(*logger*, *capability_repository*, *plugin_manager*)

Manger for all the virtual abstraction layer plugins. Loads the plugins and wrapps the commands.

close()

Will clean up the VALManager. At first it will remove the capability from the capability engine and afterwards all the deactivate method for each plugin will be executed.

instantiate(*image*)

Instantiate an image.

Parameters **image** (motey.models.image.Image) – the image which should be executed

Returns the image id of the instantiated image

register_plugins()

Register all the available plugins. A plugin has to be located under motey/val/plugins. After all the available plugins are loaded, the activate method of the plugin will be executed and a capability with the related plugin type will be added to the capability engine.

start()

Starts the VALManager and register all the available plugins.

terminate(*image*)

Terminate a running instance.

Parameters **instance_name** (*str*) – the name of the instance

CHAPTER 11

Documentation Virtualization Abstraction Layer (VAL) Plugins

class motey.val.plugins.abstractVAL.**AbstractVAL**

An abstract implementation for a virtualization abstraction layer (VAL). Should be inherited to build an custom VAL plugin. A VAL plugin is implemented as an yapsy plugin. Each plugin must be configured via a yapsy-plugin file. See more at http://yapsy.sourceforge.net/

activate()

Called at plugin activation.

create_instance(*image_name*, *parameters={}*)

Create and start an instance of an image.

Parameters

- **image_name** – the name of the image which should be created
- **parameters** – execution parameters

Returns the id of the created instance

deactivate()

Called when the plugin is disabled.

delete_image(*image_name*)

Delete an image, but not the instance of it.

Parameters **image_name** – the image to be deleted.

get_all_instances_stats()

Returns object which is type of `Status`. Represents the status of all instances of this plugin type.

Returns object from type `Status`

get_all_running_instances()

Returns a list with all running instance in this VAL.

Returns list of instances

get_plugin_type()

Returns the specific plugin type.

Returns the specific plugin type.

get_stats(*instance_name*)
Returns object which is type of `Status`. Represents the status of an instance.

Parameters **instance_name** – the name of the instance

Returns object from type `Status`

has_image(*image_name*)
Checks if an specific images exists.

Parameters **image_name** – the name of the image to search for.

Returns True if the image exist, otherwise False.

has_instance(*instance_name*)
Checks if an image instance exists.

Parameters **instance_name** – the name of an existing instance

load_image(*image_name*)
Load the image to the device, but does not start the image himself.

Parameters **image_name** – the image to be loaded.

start_instance(*instance_name*, *parameters={}*)
Start an existing image instance.

Parameters

- **instance_name** – the name of the existing instance
- **parameters** – execution parameters

Returns should return the id of the started instance

stop_instance(*instance_name*)
Stop an existing image instance.

Parameters **instance_name** – the name of the container to be stopped

class `motey.val.plugins.dockerVAL.`**DockerVAL**
Concrete implementation of the docker virtualization abstraction layer (VAL).

create_instance(*image_name*, *parameters={}*)
Create and start an instance of an image. It is a wrapper around the `docker.containers.create` command.

Parameters

- **image_name** – the name of the image which should be created
- **parameters** – execution parameters. Same as in the `docker.create_container`.

Returns the id of the created instance

delete_image(*image_name*)
Delete an image, but not the instance of it. It is a wrapper around the `docker.images.remove` command.

Parameters **image_name** – the image to be deleted.

get_all_instances_stats()
Returns object which is type of `Status`. Represents the status of all docker container.

Returns object from type `Status`

get_all_running_instances()
Returns a list with all running instance in this VAL. It is a wrapper around the `docker.containers.list(filters={'status': 'running'})` command.

Returns list of instances

get_docker_client()
Instantiates the docker client.

Returns the docker client

get_plugin_type()
Returns the specific plugin type.

Returns the specific plugin type.

get_stats(*container_name*)
Returns object which is type of `Status`. Represents the status of a container.

Parameters **container_name** – the name of the container

Returns object from type `Status`

has_image(*image_name*)
Checks if an specific images exists.

Parameters **image_name** – the name of the image to search for. Can be the `image.id` or the `image.short_id`.

Returns True if the image exist, otherwise False.

has_instance(*container_name*)
Checks if an image instance exists. It is a wrapper around the `docker.containers.get` command.

Parameters **container_name** – the name of an existing instance

load_image(*image_name*)
Load the image to the device, but does not start the image himself. It is a wrapper around the `docker.images.pull` command.

Parameters **image_name** – the image to be loaded.

start_instance(*instance_name*, *parameters={}*)
Start an existing image instance. It is a wrapper around the `docker.containers.run` command.

Parameters

- **instance_name** – the name of the existing instance
- **parameters** – execution parameters. Same as in the `docker.create_container`.

Returns the id of the started instance

stop_instance(*container_name*)
Stop an existing image instance. It is a wrapper around the `docker.containers.stop` command.

Parameters **container_name** – the name of the container to be stopped

CHAPTER 12

What is Motey

Motey is a fog node agent which is able to start virtual containers and can act autonomous.

CHAPTER 13

Installation

13.1 Dependecies

Motey is using python 3.5 or newer. All the necessary requirements are in `motey-docker-image/requirements.txt`. A separate MQTT server is optional but recommended.

13.2 Docker

The easiest way to use Motey is to run the docker container.

```
# pull the docker container.
$ docker pull neoklosch/motey

# run the container
$ docker run -ti -v /var/run/docker.sock:/var/run/docker.sock -p 5023:5023 -p␣
↪5024:5024 neoklosch/motey

# to enter the container
$ docker exec -ti <container_name> bash
```

Motey need a MQTT broker to communicate with other nodes. Therefore a MQTT server has to started.

```
# pull the docker container.
$ docker pull toke/mosquitto

# run the container and load the config file from the scripts folder
$ docker run -p 1883:1883 -p 9001:9001 -v ./scripts/config:/mqtt/config:ro toke/
↪mosquitto
```

The ip of the server has to be configured in the `config.ini` file of Motey.

13.3 Install manually Linux

```
# clone Motey repo
$ git clone https://github.com/Neoklosch/Motey.git

# enter Motey folder
$ cd Motey

# build application
$ python3 setup.py build

# install application
$ python3 setup.py install
```

CHAPTER 14

Using Motey

By default Motey is executed as a daemon. It can be started, stopped and restarted via the cli tool.

```
# start the service
$ motey start

# stop the service
$ motey stop

# restart the service
$ motey restart
```

You also can start Motey in foreground.

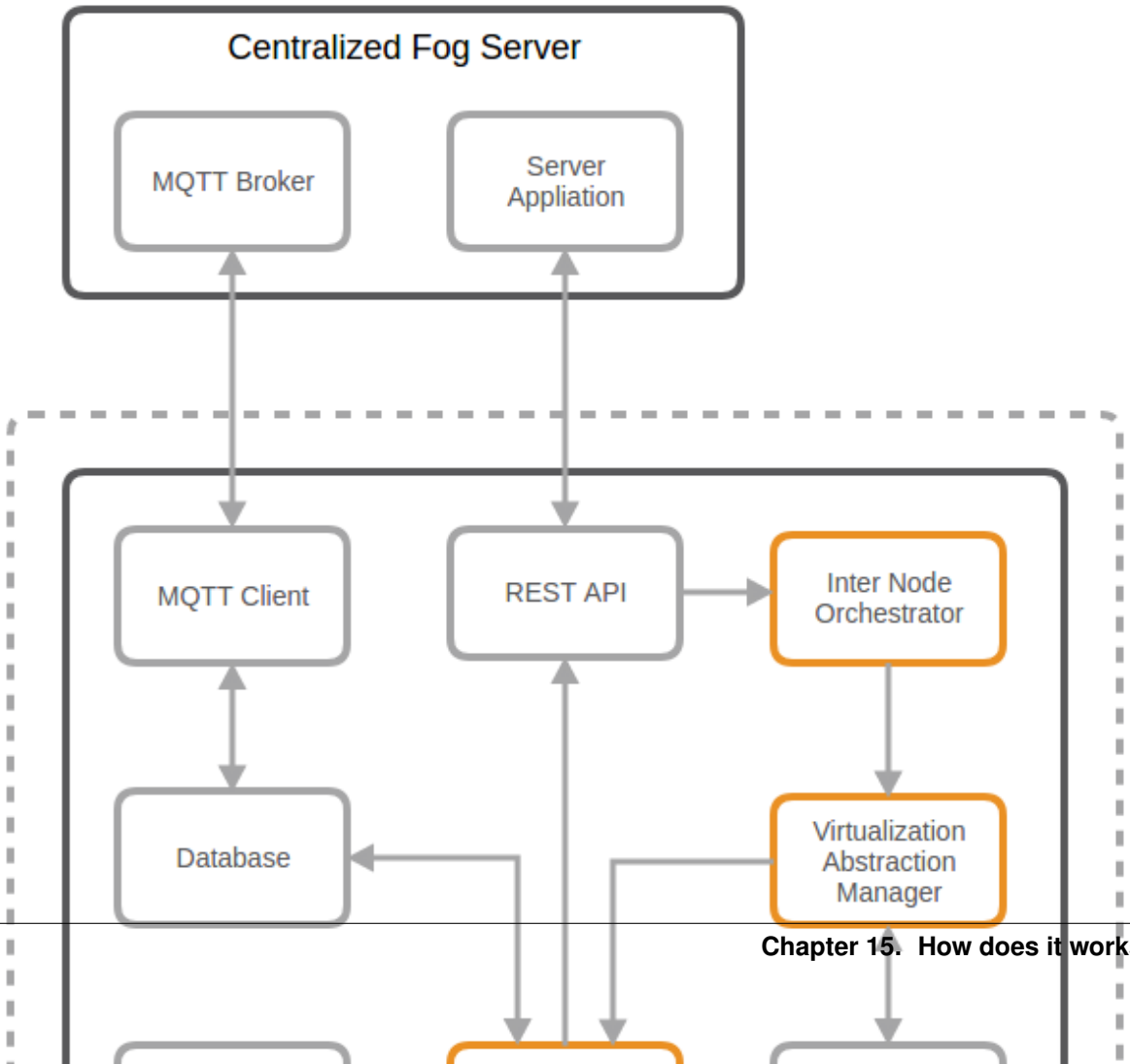
```
# start the application
$ python3 /opt/motey/main.py
```

CHAPTER 15

How does it works

15.1 Motey Architecture

no-web



15.2 Communication

Motey provide several endpoints to communicate with the system.

Capabilities Engine (only inter-process communication) You can communicate with the capabilities engine via ZeroMQ. In the default configuration port 5090 is exposed as a ZeroMQ subscriber. You can connect to them witho one ore more ZeroMQ publisher to add or remove capabilities.

REST API A REST API is provided on port 5023. Endpoints are `/v1/service` to upload a YAML blueprint and get informations about the status of a service, `/v1/capabilities` to add capabilities, which is basically another possiblity to communicate with the capabilities engine and `/v1/nodestatus` to get the current node status.

MQTT Motey will try to connect to a MQTT broker on startup. Default config is set to url `172.17.0.3` and port `1883`. This can be configured by modifing the `config.ini` file.

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

R

S

T

U

V

Z