sgframework Documentation

Release 0.2.3

Jonas Berg

2016-10-17

Documentation built using Sphinx 2016-10-17 for sgframework version 0.2.3.

Contents:

CHAPTER 1

Introduction to sgframework

Secure Gateway is a concept for an Internet Protocol (IP) based in-vehicle infotainment network, with a secure connection to the safety-critical vehicle network. This allows the user to install infotainment apps on the infotaiment head unit (IHU), without compromising the vehicle safety. The concept is also useful for example in industrial applications, where serial communication to industrial equipment must be protected, or in general for Internet-of-things (IoT) applications.

This sgframework library is a Python package intended for implementing demonstration nodes in the suggested architecture.

There are several examples available, including examples running CAN (Controller Area Network) communication on embedded Linux boards (for example Raspberry Pi and Beaglebone).

1.1 Web resources

- Free software: BSD license
- Source code on GitHub: https://github.com/caran/SecureGateway
- Documentation: https://sgframework.readthedocs.org
- Python Package Index (PyPI): https://pypi.python.org/pypi/sgframework

1.2 Features

- Examples use the can4python package for CAN communication.
- Implements Apps and Resources.
- Supports Python 3.3 and later.

1.3 Installation and usage

See separate documentation pages.

Installation

At the command line:

```
$ pip3 install sgframework
```

This will also install the dependencies `paho-mqtt` and `can4python`. In order to actually use it, you need to install an MQTT broker, etc. See the "Dependencies" chapter.

Run the canadapter:

```
$ canadapter -h
```

To download the example files and test files, you need to use:

```
$ git clone https://github.com/caran/sgframework.git
```

In order to run tests, you might need to enable the virtual CAN bus. On Debian:

```
$ sudo make vcan
```

Run tests:

```
$ make test
```

If running on a desktop machine, it is possible to test also the graphical apps etc (more dependencies required):

```
$ make test-all
```

CHAPTER 3

Background

Secure Gateway (SG) is an architecture concept, using Internet technology in automotive and industrial environments. As a concept, it is not intended for inclusion in products and is not production ready. It is our hope that it can give inspiration for architectures of future embedded systems. The Secure Gateway concept is the result of the automotive research project PLINTA, but has since found its use also in industrial environments. It is further developed in the "Second Road - Open Innovation Lab" project.

Vehicles have several buses for vehicle data, often of the types Controller Area Network (CAN) or Flexray. These vehicle buses contain safety-critical data, and must be protected from malicious impact.

Vehicles also have infotainment electronics, for example the Infotainment Head Unit (IHU) and Rear Seat Entertainment (RSE). Also smart-phone apps for direct communication with the vehicles can be considered part of the infotainment system, as the user for example would like to transfer destination information from her smartphone to the vehicle navigation system. Thus there is a need for a separate infotainment network, separated from the safety-critical vehicle network. Nevertheless, there must be some connection between the two networks, as the IHU should control for example the climate settings.

3.1 Architecture and MQTT background

Secure Gateway utilizes the MQTT (Message Queue Telemetry Transport) protocol for communication via a broker (a MQTT server), which handles access control. Authentication and encryption is handled by Transport Layer Security (TSL). The MQTT clients don't have any open ports to the IP network, and they connect to the broker.

In the Secure Gateway we have defined resources and apps. For example the in-car climate node can be represented by an SG resource, while an application in the IHU sending climate control signals corresponds to an SG app. Note that both resources and apps send and receive MQTT messages.

The MQTT protocol uses a publish-subscribe pattern. A MQTT broker (server) acts as the central communication point. Clients register the message types (MQTT topics) they are interested in (to the broker). A client publishes a message to the broker, which re-transmits the message to the clients that previously have subscribed to the topic. The publishers does not know the identity (or existence) of the subscribers.

Details on the MQTT protocol are found on:

http://mqtt.org/ http://en.wikipedia.org/wiki/MQTT

The Secure Gateway concept was presented at the conference "ESCAR USA" (embedded security in cars) 2015. The full paper "Secure Gateway – A concept for an in-vehicle IP network bridging the infotainment and the safety critical domains" is available for download through the ESCAR website (registration required).

3.2 Tutorial overview

This documentation contains tutorials, mainly written for running on a Ubuntu desktop Linux machine. They are also suitable for running on embedded Linux boards, for example the Beaglebone and the Raspberry Pi. This is especially relevant when it comes to sending and receiving CAN messages, as there are CAN expansion boards available for those platforms.

This tutorial will show you how to:

- Install the necessary components.
- Test MQTT communication from command line tools.
- Use the Secure Gateway topic hierarchy.
- Use a simulated resource (a taxi sign) and an app to control it.
- Build your own Secure Gateway enabled taxi sign hardware.
- Work with client-side and server-side certificates and encryption.
- Send and receive CAN messages on a simulated CAN bus.
- Run a real CAN-bus between a CAN vehicle simulator and the Secure Gateway, implemented on two embedded Linux boards (Raspberry Pi or Beaglebone).
- Develop your own resources and apps for the Secure Gateway.

3.3 MQTT tutorial

In order to be able to try this, you need to install the Mosquitto broker and the Mosquitto command line tools. See another section of this documentation for installation details.

The Secure Gateway concept builds on using the MQTT protocol over an IP network.

Each MQTT message has a payload and a topic, which both are strings. The MQTT topics are arranged in a hierarchy, for example `A/B/C/D`.

These wildcard are used: `*` to listen to everything in a message hierarchy and `+` to allow anything on that particular message hierarchy level.

For example the actual in-car temperature reading might be published on this topic:

```
data/climateservice/actualindoortemperature
```

To listen to all data from the climateservice, use this topic:

```
data/climateservice/*
```

Similarly, to listen to all data from any node, use this topic:

```
data/*
```

To listen to everything related to the climateservice, use:

```
+/climateservice/*
```

Mosquitto does not require a config file, if all default settings are accepted. Then certificates are not used. For some Linux distributions the Mosquitto broker is started automatically after installation. See below for how to then start and stop it.

In general, start the Mosquitto broker by running:

```
$ mosquitto
```

To subscribe to all topics (and print out the topic for each message), use this tool:

```
$ mosquitto_sub -t +/#  -v
```

It connects to `localhost` on port 1883, by default.

Open one terminal window, and run the command above.

To send one message on the `data/climateservice/actualindoortemperature` topic, use something like this:

```
$ mosquitto_pub -t data/climateservice/actualindoortemperature -m 27.4
```

Run this command in a second terminal window, and look at the message appearing in the first terminal window.

It is possible to send messages as "retained", which means that the broker is sending the last known value to any new subscriber.

Try out this retained message:

```
$ mosquitto_pub -t data/climateservice/actualindoortemperature -m 27.4 -r
```

Then open a new terminal window, and subscribe to all topics using the command above. Note that the retained message will appear!

To "delete" a retained message from the broker, send a message with a NULL payload:

```
mosquitto_pub -t data/climateserviceactualindoortemperature -n -r
```

The "Quality of Service" (QOS) setting is defining how hard the broker is trying to ensure that messages have been delivered. It ranges from "fire and forget" to a four-step handshake.

Also a "last will" can be defined for each client. That is a message sent out by the broker, if the client connection is unexpectedly lost.

3.4 MQTT topic structure for Secure Gateway

In order to handle `presence` information, a number of additional topic structures are defined. With presence information it is possible to have a plug-and-play behavior of new resources and apps.

Secure Gateway uses a topic hierarchy with three levels:

```
messagetype/servicename/signalname payload
```

The message types are basically:

- `data` Data sent from a resource (to an app or to another resource)
- `command` Command sent to a resource (from an app or from another resource)

In addition there are `messagetypes` for indication of `presence` of the above functionality:

- `dataavailable` Indicates that certain data is available.
- `commandavailable` Indicates that a certain command is available.

So if a sensor publishes data on the topic:

```
data/sensor1/temperature 29.3
```

then the sensor is expected to send this message at startup:

```
dataavailable/sensor1/temperature True
```

Applications connecting later are receiving the dataavailable message, as it was published as “retained”. This instructs the broker to send the last known value to new clients on connection.

(There is also the resourceavailable messagetype, as discussed in a subsequent section).

The signalnames should be unique among commands and data. The command is typically echoed back as data. There should not be some data topic having the same signalname as a command.

Note that resources and apps are both subscribers and publishers of messages.

3.5 Abbreviations

- ACL - Access Control List
- CA - Certificate Authority
- CN - Common Name. For certificates.
- CAN - Controller Area Network
- CSR - Certification Signature Request. A file format for sending requests to the certificate authority (CA).
- DBC - Database for CAN. A file format for CAN configuration, owned by Vector Informatik GmbH.
- DLC - Data Length Code. Part of a CAN message.
- DNS - Domain Name System
- IP - Internet Protocol
- KCD - Kayak CAN Definition. A file format used by the open-source Kayak application for displaying CAN data.
- MQTT - Message Queue Telemetry Transport
- PEM - Text file format for keys and certificates
- PKI - Public Key Infrastructure
- SSL - Secure Sockets Layer
- TLS - Transport Layer Security

CHAPTER 4

Usage introduction for the 'sgframework' package

With the sgframework you can implement the 'apps' and 'resources' described in the previous section. Below are some minimal examples to get you started.

4.1 Minimal 'resource' example

This is an example of a 'resource' implemented using the Secure Gateway framework. It is a taxi sign listening to commands on the MQTT topic `command/taxisignservice/state`. When the received payload is `True` the text "Turning on my taxi sign." is printed.

Listing 4.1: ../examples/minimal/minimaltaxisign.py

```python
import time

import sgframework


def on_taxisign_state_command(resource, messagetype, servicename,
                              commandname, commandpayload):
    if commandpayload.strip() == 'True':
        print("Turning on my taxi sign.", flush=True)
        return 'True'
    else:
        print("Turning off my taxi sign.", flush=True)
        return 'False'


resource = sgframework.Resource('taxisignservice', 'localhost')
resource.register_incoming_command('state',
                                   on_taxisign_state_command,
                                   defaultvalue='False',
                                   send_echo_as_retained=True)
resource.start(use_threaded_networking=True)
while True:
    time.sleep(1)
```

This example resource is using threaded networking, meaning that the MQTT communication is done in a separate thread.

It is also possible to run the resource in a single thread, but then you need to call `resource.loop()` regularly.

4.2 Minimal 'app' example

This example 'app' controls the taxi sign by sending MQTT commands on the appropriate topic. It also listens to the echo from the taxi sign, and shows the current taxi sign state.

Listing 4.2: ../examples/minimal/minimaltaxiapp.py

```python
import time

import sgframework


def on_taxisign_state_data(app, messagetype, servicename, signalname, payload):
    if payload.strip() == 'True':
        print("The taxi sign is now on.", flush=True)
    else:
        print("The taxi sign is now off.", flush=True)


app = sgframework.App('taxiapp', 'localhost')
app.register_incoming_data('taxisignservice', 'state', on_taxisign_state_data)
app.start()   # Not using threaded networking

starttime = time.time()
command_sent = False
while True:
    app.loop()

    if time.time() - starttime > 4 and not command_sent:
        print("Turning on the taxi sign from script...", flush=True)
        app.send_command('taxisignservice', 'state', 'True')
        command_sent = True
```

This app is an example implementation not using threaded networking. Apps can also use threaded networking, as described in the API documentation (see another section).

4.3 Usage recommendations

It is recommended to use the threaded networking. However when using for example the TK graphics library, it is not possible to run it in another thread. Instead you need to call the `loop()` method regularly.

4.4 Developing your own Resources

If you are to develop your own resource for the Secure Gateway network, you could use the available resource framework (which runs under Python3).

The main API object is the `Resource`, and you should use these public methods to get your resource up and running:

- `register_incoming_command()`
- `register_outgoing_data()`
- `start()`
- `send_data()`

and:

- `stop()`
- `loop()` (if not using the threaded networking interface)

Have a look on the source code for the taxisign service (in the distributed examples) to have inspiration for the usage of the Secure Gateway resource framework.

Of course, Secure Gateway resources can be implemented in any programming language having a proper MQTT library with TLS support.

4.5 Developing your own Apps

Similarly, there is Secure Gateway app framework, for usage when implementing custom Python3 apps.

The main API object is the `App`, and use these public methods to get it running:

- `register_incoming_availability()`
- `register_incoming_data()`
- `start()`
- `send_command()`

and:

- `stop()`
- `loop()` (if not using the threaded networking interface)

The taxisign app source code should be used for inspiration.

CHAPTER 5

API for sgframework

This page shows the public part of the API. For a more detailed documentation on all objects, see the sgframework page: `sgframework.framework`

class `sgframework.`**`App`**`(`*name*, *host*, *port=1883*, *certificate_directory=None*`)`

App framework for the Secure Gateway

Sends commands to any resource. Handles incoming MQTT messages (data) from any resource.

It does not have any 'last will'. Typically sends (non retained=non persistent) commands to:

`command/`*resource_to_be_controlled*`/`*signalname*

and listens to data on topic:

`data/`*dataproducing_resource*`/`*signalname*

Parameters

- **`name`** (*`str`*) – Name of the app/resource. For resources, it is also used in the MQTT topic hierarchy.
- **`host`** (*`str`*) – Broker host name.
- **`port`** (*`int`*) – Broker port number.
- **`certificate_directory`** (*`str or None`*) – Full path to the directory of the certificate files.

`protocol`

enum in the Paho module

MQTT protocol version, defaults to `MQTTv31`, as older versions of the Mosquitto broker can not handle `MQTTv311`.

`tls_version`

enum in the ssl module

SSL protocol version, defaults to `ssl.PROTOCOL_TLSv1`

`qos`

int

MQTT quality of service. 0, 1 or 2. See Paho documentation. Default value `DEFAULT_QOS` is set in `sgframework.constants`.

`timeout`

numerical

MQTT socket timeout, when running the `loop()` method. Default value `DEFAULT_TIMEOUT`.

keepalive
numerical

MQTT keepalive message interval. Default value `DEFAULT_KEEPALIVE_TIME`.

Also the parameters appear as attributes. The public attributes are used when calling `start()`. Any changes are valid from next `start()`.

References to sub-objects:

- **mqttclient** (object): See Paho documentation
- **logger** (object): See Python standard library documentation
- **userdata** (whatever): Convenience object that is available for user code in callbacks. Not used by the framework itself.
- **on_broker_connectionstatus_info**: Implement this callback if you would like notifications on broker connection status changes. See below.

Callback to the user application on changed broker connection status:

```
on_broker_connectionstatus_info(app_or_resource, broker_connected)
```

Where *app_or_resource* is the app or resource object, and *broker_connected* (**bool**) is `True` if the user application is connected to the broker.

Callbacks to the user application on incoming information are registered using separate methods. The callbacks should have this interface:

```
callbackname(resource_or_app, messagetype, servicename, signalname, inputpayload)
```

where *messagetype*, *servicename*, *signalname* and *inputpayload* are strings. The callback is protected by try/except. The strings to the callback have been through `.strip()`.

When using echo and the returnvalue of the callback is `None`, the command payload is used in the echo. For returnvalues other then `None`, the echo payload will be `str(returnvalue)`. More than one input signal can use the same callback.

The certificate files should be named according to `CA_CERTS`, `CERTFILE` and `KEYFILE`.

get_descriptive_ascii_art()
Display an overview with registered incoming and outgoing topics.

Returns A multi-line string.

loop()
Run network activities.

This function needs to be called frequently to keep the network traffic alive, if not using threaded networking. It will block until a message is received, or until the self.timeout value.

If not connected to the broker, it will try to connect once.

Do not use this function when running threaded networking.

register_incoming_availability(*prefix*, *servicename*, *signalname*, *callback*)
Register a callback for incoming availability information (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data etc from other resources).

Parameters

- **prefix** (*str*) – one of PREFIX_COMMANDAVAILABLE, PREFIX_DATAAVAILABLE (or maybe PREFIX_RESOURCEAVAILABLE)

- **`servicename`** (*`str`*) – name of the service sending the availability info
- **`signalname`** (*`str`*) – name of the data or command
- **`callback`** (*`function`*) – Callback that will be used when availability information is received.

When registering a callback for RESOURCEAVAILABLE the actual value of the signalname is not used. Just pass in any string.

For details on the callback, see the class documentation.

Subscribes to: *prefix/servicename/signalname*

for example: `dataavailable/climateservice/actualindoortemperature`.

`register_incoming_data`(*servicename*, *signalname*, *callback*, *callback_on_change_only=False*)
Register a callback for incoming data (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data from other resources).

Parameters

- **`servicename`** (*`str`*) – name of the service sending the data
- **`signalname`** (*`str`*) – name of the signal
- **`callback`** (*`function`*) – Callback that will be used when data is received.
- **`callback_on_change_only`** (*`bool`*) – Trigger callback only for changed payload.

For details on the callback, see the class documentation.

Subscribes to: `data/`*servicename/signalname*

for example: `data/climateservice/actualindoortemperature`.

`send_command`(*servicename*, *signalname*, *value*, *send_command_as_retained=False*)
Send a command.

Primarily useful for apps (but is useful for resources to control other resources).

Parameters

- **`servicename`** (*`str`*) – destination service name
- **`signalname`** (*`str`*) – destination signal name
- **`value`** – Value to be sent. Is converted to a string before sending.
- **`send_command_as_retained`** (*`bool`*) – Publish the command as retained.

Sends messages on topic: `command/`*servicename/signalname*

for example `command/climateservice/aircondition`.

Most often commands are sent as non-retained messages.

`start`(*use_threaded_networking=False*, *use_clean_session=True*)
Connect to the broker.

Parameters

- **`use_threaded_networking`** (*`bool`*) – Start MQTT networking activity in a separate thread.
- **`use_clean_session`** (*`bool`*) – Connect to broker using a clean session.

If not using threaded networking, you need to call the `loop()` method frequently.

If using a clean session, also the client name is changed to include the process ID. This in order to avoid client name collisions in the broker.

stop()

Disconnect from the broker

class `sgframework.`**`Resource`**(*name*, *host*, *port=1883*, *certificate_directory=None*)

Resource framework for the Secure Gateway

Receives commands from apps (incoming MQTT messages). Sends data to apps (outgoing MQTT messages).

It can also recieve incoming data from other resources, and can send commands to other resources.

The resource name is typically part of incoming and outgoing message topics:

`command/`*myresourcename*/*signalname*

`data/`*myresourcename*/*signalname*

Also publishes availability of commands and data, using retained messages:

`commandavailable/`*myresourcename*/*signalname*

`dataavailable/`*myresourcename*/*signalname*

When starting up, it sends 'True' to the 'last will' topic in a retained message:

`resourceavailable/`*myresourcename*`/presence`

The broker is automatically broadcasting 'False' on 'last will' topic at lost connection.

Parameters

- **name** (*str*) – Name of the app/resource. For resources, it is also used in the MQTT topic hierarchy.
- **host** (*str*) – Broker host name.
- **port** (*int*) – Broker port number.
- **certificate_directory** (*str or None*) – Full path to the directory of the certificate files.

protocol

enum in the Paho module

MQTT protocol version, defaults to `MQTTv31`, as older versions of the Mosquitto broker can not handle `MQTTv311`.

tls_version

enum in the ssl module

SSL protocol version, defaults to `ssl.PROTOCOL_TLSv1`

qos

int

MQTT quality of service. 0, 1 or 2. See Paho documentation. Default value `DEFAULT_QOS` is set in *`sgframework.constants`*.

timeout

numerical

MQTT socket timeout, when running the `loop()` method. Default value `DEFAULT_TIMEOUT`.

keepalive
numerical

MQTT keepalive message interval. Default value `DEFAULT_KEEPALIVE_TIME`.

Also the parameters appear as attributes. The public attributes are used when calling `start()`. Any changes are valid from next `start()`.

References to sub-objects:

- **mqttclient** (object): See Paho documentation
- **logger** (object): See Python standard library documentation
- **userdata** (whatever): Convenience object that is available for user code in callbacks. Not used by the framework itself.
- **on_broker_connectionstatus_info**: Implement this callback if you would like notifications on broker connection status changes. See below.

Callback to the user application on changed broker connection status:

```
on_broker_connectionstatus_info(app_or_resource, broker_connected)
```

Where *app_or_resource* is the app or resource object, and *broker_connected* (**bool**) is `True` if the user application is connected to the broker.

Callbacks to the user application on incoming information are registered using separate methods. The callbacks should have this interface:

```
callbackname(resource_or_app, messagetype, servicename, signalname, inputpayload)
```

where *messagetype*, *servicename*, *signalname* and *inputpayload* are strings. The callback is protected by try/except. The strings to the callback have been through `.strip()`.

When using echo and the returnvalue of the callback is `None`, the command payload is used in the echo. For returnvalues other then `None`, the echo payload will be `str(returnvalue)`. More than one input signal can use the same callback.

The certificate files should be named according to `CA_CERTS`, `CERTFILE` and `KEYFILE`.

get_descriptive_ascii_art()
Display an overview with registered incoming and outgoing topics.

Returns A multi-line string.

loop()
Run network activities.

This function needs to be called frequently to keep the network traffic alive, if not using threaded networking. It will block until a message is received, or until the self.timeout value.

If not connected to the broker, it will try to connect once.

Do not use this function when running threaded networking.

register_incoming_availability(*prefix*, *servicename*, *signalname*, *callback*)
Register a callback for incoming availability information (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data etc from other resources).

Parameters

- **prefix** (*str*) – one of PREFIX_COMMANDAVAILABLE, PREFIX_DATAAVAILABLE (or maybe PREFIX_RESOURCEAVAILABLE)

- **`servicename`** (*`str`*) – name of the service sending the availability info
- **`signalname`** (*`str`*) – name of the data or command
- **`callback`** (*`function`*) – Callback that will be used when availability information is received.

When registering a callback for RESOURCEAVAILABLE the actual value of the signalname is not used. Just pass in any string.

For details on the callback, see the class documentation.

Subscribes to: *prefix/servicename/signalname*

for example: `dataavailable/climateservice/actualindoortemperature`.

`register_incoming_command`(*signalname*, *callback*, *callback_on_change_only=False*, *echo=True*, *send_echo_as_retained=False*, *defaultvalue=None*)

Register a callback for an incoming command (incoming MQTT message).

Parameters

- **`signalname`** (*`str`*) – command name
- **`callback`** (*`function`*) – Callback that will be used when a command is received.
- **`callback_on_change_only`** (*`bool`*) – Trigger callback only for changed payload.
- **`echo`** (*`bool`*) – True if the incoming command should be echoed back (as "data")
- **`send_echo_as_retained`** (*`bool`*) – True if the echo should be published as retained.
- **`defaultvalue`** – Value to be echoed on startup and reconnect. Set to None to avoid sending. The value is converted to a string before sending. It will be updated by the internal *`_on_incoming_message()`* callback for incoming MQTT messages.

For details on the callback, see the class documentation.

Subscribes to: `command/`*myresourcename/signalname*

When the resource is starting, it is publishing a retained message to:

`commandavailable/`*myresourcename/signalname*

`register_incoming_data`(*servicename*, *signalname*, *callback*, *callback_on_change_only=False*)

Register a callback for incoming data (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data from other resources).

Parameters

- **`servicename`** (*`str`*) – name of the service sending the data
- **`signalname`** (*`str`*) – name of the signal
- **`callback`** (*`function`*) – Callback that will be used when data is received.
- **`callback_on_change_only`** (*`bool`*) – Trigger callback only for changed payload.

For details on the callback, see the class documentation.

Subscribes to: `data/`*servicename/signalname*

for example: `data/climateservice/actualindoortemperature`.

`register_outgoing_data`(*signalname*, *defaultvalue=None*, *send_data_as_retained=False*)

Pre-register information on a outgoing data topic (MQTT messages). Note that the actual data sending is later done with the *`send_data()`* method.

Parameters

- **`signalname`** (*`str`*) – signal name
- **`defaultvalue`** – Value to be sent on startup and reconnect. Set to None to avoid sending. The value is converted to a string before sending. It will be updated by send_data().
- **`send_data_as_retained`** (*`bool`*) – Whether the data should be published as retained

When the resource is starting, it is publishing a retained message to:

`dataavailable/`*myresourcename*`/`*signalname*

Upon sending data, the topic is: `data/`*myresourcename*`/`*signalname*

for example: `data/climateservice/actualindoortemperature`.

Typically the data is published using non-retained messages.

`send_command`(*servicename*, *signalname*, *value*, *send_command_as_retained=False*)
Send a command.

Primarily useful for apps (but is useful for resources to control other resources).

Parameters

- **`servicename`** (*`str`*) – destination service name
- **`signalname`** (*`str`*) – destination signal name
- **`value`** – Value to be sent. Is converted to a string before sending.
- **`send_command_as_retained`** (*`bool`*) – Publish the command as retained.

Sends messages on topic: `command/`*servicename*`/`*signalname*

for example `command/climateservice/aircondition`.

Most often commands are sent as non-retained messages.

`send_data`(*signalname*, *value*)
Send data on a pre-registered topic.

Parameters

- **`signalname`** (*`str`*) – signal name
- **`value`** – Value to be sent. Is convered to a string before sending.

Sends to the topic: `data/`*myresourcename*`/`*signalname*

for example: `data/climateservice/actualindoortemperature`.

Updates the defaultvalue for this signal.

Whether the signal should be sent as retained or not is set already during registration.

`start`(*use_threaded_networking=False*, *use_clean_session=True*)
Connect to the broker.

Parameters

- **`use_threaded_networking`** (*`bool`*) – Start MQTT networking activity in a separate thread.
- **`use_clean_session`** (*`bool`*) – Connect to broker using a clean session.

If not using threaded networking, you need to call the `loop()` method frequently.

If using a clean session, also the client name is changed to include the process ID. This in order to avoid client name collisions in the broker.

stop()

Disconnect from the broker

Service Manager broker add-on

Each client can register a last will, that is sent by the broker if the client connection is unexpectedly lost. We use it to indicate the presence of resources (not apps). For example, the last will could be:

```
dataavailable/sensor1/temperature False
```

Unfortunately it is possible only to send one message per client, why some trick is required to handle more than one signal per client (resource). Therefore the resource instead register:

```
resourceavailable/sensor1/presence False
```

as the last will, and sends this message at startup (along with the `dataavailable` message):

```
resourceavailable/sensor1/presence True
```

A separare component, the Service Manager, is keeping track of the connected services. It will send the individual `datavailable/x/y False` when resource x disconnects.

Start up the Service Manager:

```
$ python3 scripts/servicemanager.py
```

Test it from command line by using one subscribe window and one publish window. Subscribe in one terminal:

```
$ mosquitto_sub -t +/#  -v
```

In the other window:

```
$ mosquitto_pub -t resourceavailable/foo/presence -m True
$ mosquitto_pub -t dataavailable/foo/bar -m True
$ mosquitto_pub -t commandavailable/foo/baz -m True

$ mosquitto_pub -t resourceavailable/foo/presence -m False
```

The Service Manager will then automatically send these messages:

```
dataavailable/foo/bar False
commandavailable/foo/baz False
```

6.1 Service Manager helptext

```
$ python3 scripts/servicemanager.py -h
usage: servicemanager.py [-h] [-v] [-version] [-host HOST] [-port PORT]
                         [-cert CERT] [-qos {0,1,2}]

optional arguments:
  -h, --help    show this help message and exit
  -v            Increase verbosity level. Can be repeated.
  -version      show program's version number and exit
  -host HOST    Broker host name. Defaults to 'localhost'.
  -port PORT    Broker port number. Defaults to 1883.
  -cert CERT    Directory for certificate files. Defaults to not using
                certificates.
  -qos {0,1,2}  MQTT quality-of-service setting. Defaults to '0'.

A Service Manager for the Secure Gateway concept architecture

It requires a MQTT broker (for example Mosquitto), and a MQTT client library (Paho).

It registers on the Secure Gateway network, and can connect to
the broker in a secure or insecure way.
The settings of the broker determines what is allowed. To connect in the secure way,
the directory of the certificate files must be specified.

The certificate files should be named:
  CA file:          ca_public_certificate.pem
  Certificate file: public_certificate.pem
  Key file:         private_key.pem
```

CHAPTER 7

Canadaper usage

The canadaper is a script distributed as part of the sgframework. It converts CAN signals to MQTT messages and MQTT messages to CAN signals. All MQTT communication is handled by the MQTT broker which controls the access. Most often only a specific subset of all available vehicle CAN signals is implemented in the canadapter. Typically the 'DeployAirbag' CAN signal is excluded from the implementation in the canadapter.

The configuration of the canadapter consists of providing it with information about the signals on the CAN bus, and a configuration file telling which of those that should be possible to send/receive over MQTT. The Canadapter uses the KCD fileformat for configuring the CAN signals. This file-format is an open-source alternative to the Vector DBC files. An DBC-to-KCD conversion tool is available online: CAN-Babel

The usecase for this prototype implementation is to extract a few signals from a CAN bus and possibly send a few CAN signals. Due to the use of the MQTT protocol (over TCP/IP) it will not fulfill any hard realtime requirements.

7.1 Configuration files for canadapter

There are two configuration files for the canadapter. One describes the frames and signals on the CAN bus, and is in the KCD file format. The other type of configuration file defines the names each of the CAN signals should have when sent as MQTT messages and is in JSON file format.

7.1.1 KCD file

The KCD file format is described here: https://github.com/julietkilo/kcd

Canadapter is using the KCD file format via the can4python library, which is found here: https://github.com/caran/can4python

Example of a KCD configuration file:

Listing 7.1: ../examples/configfilesForCanadapter/climateservice_cansignals.kcd

```
<NetworkDefinition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns="http://kayak.2codeornot2code.org/1.0"
                   xsi:noNamespaceSchemaLocation="Definition.xsd">
    <Document name="Vehicle simulator"></Document>
    <Bus name="Mainbus">
        <Message id="0x007" name="climatecontrolsignals">
            <Signal name="acstatus" offset="7" length="1" endianess="big"/>
            <Producer>
                <NodeRef id="1"/>
            </Producer>
        </Message>
        <Message id="0x008" name="vehiclesimulationdata">
            <Signal name="vehiclespeed" offset="8" length="16" endianess="big">
                <Value type="unsigned" slope="0.01"/>
            </Signal>
            <Signal name="enginespeed" offset="26" length="14" endianess="big"/>
        </Message>
        <Message id="0x009" name="climatesimulationdata">
            <Signal name="indoortemperature" offset="8" length="11" endianess="big">
                <Value type="unsigned" slope="0.1" intercept="-50"/>
            </Signal>
        </Message>
    </Bus>
</NetworkDefinition>
```

Edit the KCD file to only contain the CAN frames and signals you are interested in, otherwise CPU capacity is used to parse CAN frames of data not is used.

7.1.2 JSON file

The JSON file defines the names of each of the CAN signals should be sent as MQTT messages. It also represents which of the CAN frames should be sent on MQTT and which MQTT messages should be allowed to be forwarded to the CAN bus.

Note however which messages are actually allowed to be sent to the CAN bus (they are also controlled by the CAN configuration file (KCD)) and which of the nodes on the bus are enacted by the canadapter (ego node id). The KCD file could hold information about all nodes and signals in the system.

For example, consider a KCD file defining that node 1 sends CAN frame id 6 and 7, and node 2 sends CAN frame 8 and 9. If the canadapter enacts node 2 (ego node id, set by a command line argument), then it is allowed to send CAN frame 8 and 9 according to the KCD file. The JSON file further defines that this instance of the canadapter only is allowed to send CAN frame 9.

It is possible to adjust the configuration so several CAN signals to be sent in a single MQTT message. Here is an example of such an MQTT wire message:

```
{"values":
   {
      "ADAS_Seg_MsgType": 1.0,
      "ADAS_Seg_Offset": 2.0,
      "ADAS_Seg_CycCnt": 3.0,
      "ADAS_Seg_EffSpdLmt": 4.0,
      "ADAS_Seg_EffSpdLmtType": 5.0
   }
}
```

If an MQTT message contains one CAN signal (extracted from a CAN frame), this is named a “signal” in the JSON configuration file.

If an MQTT message contains information about several CAN signals (all extracted from the same CAN frame)then it is named an “aggregate” in the JSON configuration file.

The top structure of the JSON configuration file is like this:

```
{"entities":
    {
        "signals": [],
        "aggregates": []
    }
}
```

Each "signal" object (in the list of signals) should have a structure like this:

```
{"canName": "indoortemperature",
 "canMultiplier": 1.0,
 "mqttName": "actualindoortemperature",
 "mqttType": "float",
 "mqttEcho": false,
 "toCan": false,
 "fromCan": true
}
```

Only the "canName" field of a signal is mandatory. The "mqttName" defaults to be the same as the "canName". The "mqttType" is float by default, but could also be "int". The field "mqttEcho" sets whether an incoming MQTT command should be echoed back as data on MQTT, and defaults to false. By default a signal is allowed to be converted from CAN (to MQTT), but not to CAN (from MQTT). The multiplier is used when converting a CAN signal to an MQTT signal. In the other direction is 1/multiplier used. Defaults to 1.0.

Example of a JSON configuration file containing only "signals":

Listing 7.2: ../examples/configfilesForCanadapter/climateservice_mqttsignals.json

```
{"entities":
    {"signals":[
        {"canName":"vehiclespeed"},
        {"canName":"enginespeed"},
        {"canName":"indoortemperature",
         "canMultiplier": 1.0,
         "mqttName":"actualindoortemperature",
         "mqttType":"float",
         "mqttEcho":false,
         "toCan":false,
         "fromCan":true
        },
        {"canName":"acstatus",
         "mqttName":"aircondition",
         "mqttEcho":true,
         "toCan":true,
         "fromCan":false}
    ]}
}
```

Each "aggregate" object (in the list of aggregates) should have a structure like this:

```
{"mqttName": "ADAS_Seg",
 "toCan": true,
 "fromCan": false,
 "signals": [
      {"canName": "ADAS_Seg_MsgType", "mqttType": "int"},
      {"canName": "ADAS_Seg_Offset"},
      {"canName": "ADAS_Seg_CycCnt", "mqttType": "int"},
      {"canName": "ADAS_Seg_EffSpdLmt"},
      {"canName": "ADAS_Seg_EffSpdLmtType"}
   ]
}
```

For aggregates, only the MQTTname key is mandatory. The “signals” part of an aggregate are the same as defined above for an individual signal. Only the top level “toCan” and “fromCan” fields of an aggregate are used (not the fields inside the signals, if set). Defaults to allow conversion from CAN (but not to CAN).

Example of a JSON configuration file containing “signals” and “aggregates”:

Listing 7.3: ../examples/configfilesForCanadapter/ADASIS_mqttsignals.json

```
{"entities":{
    "signals":[
            {"canName":"ADAS_Posn_MsgType", "mqttType":"int"},
            {"canName":"ADAS_Posn_Offset"},
            {"canName":"ADAS_ProfShort_CtrlPoint", "mqttType":"int"}],
    "aggregates":[
        {"mqttName":"ADAS_Posn_1",
         "signals":[
            {"canName":"ADAS_Posn_MsgType", "mqttType":"int"},
            {"canName":"ADAS_Posn_Offset", "mqttName":"Position_offset"},
            {"canName":"ADAS_Posn_CycCnt", "mqttType":"int"},
            {"canName":"ADAS_Posn_Spd"}]},
        {"mqttName":"ADAS_Posn_2",
         "signals":[
            {"canName":"ADAS_Posn_PosProbb"},
            {"canName":"ADAS_Posn_CurLane"}]},
        {"mqttName":"ADAS_Seg",
         "toCan":true,
         "fromCan":false,
         "signals":[
            {"canName":"ADAS_Seg_MsgType", "mqttType":"int"},
            {"canName":"ADAS_Seg_Offset"},
            {"canName":"ADAS_Seg_CycCnt", "mqttType":"int"},
            {"canName":"ADAS_Seg_EffSpdLmt"},
            {"canName":"ADAS_Seg_EffSpdLmtType"}]}
        ]
    }
}
```

7.2 Resulting MQTT messages

This is a part of the interpretation of the climateservice_cansignals.kcd file:

```
CAN frame definition. ID=8 (0x008, standard) 'vehiclesimulationdata', DLC=8, cycletime None ms, produ
    Signal details:
    ---------------



    Signal 'vehiclespeed' Startbit 8, bits 16 (min DLC 2) big endian, unsigned, scalingfactor 0.01, u
         valoffset 0.0 (range 0 to 7e+02) min None, max None, default 0.0.

         Startbit normal bit numbering, least significant bit: 8
         Startbit normal bit numbering, most significant bit: 7
         Startbit backward bit numbering, least significant bit: 48

                  111111   22221111 33222222 33333333 44444444 55555544 66665555
         76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
         Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
```

```
          MXXXXXXX XXXXXXXL
          66665555 55555544 44444444 33333333 33222222 22221111 111111
          32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210


     Signal 'enginespeed' Startbit 26, bits 14 (min DLC 4) big endian, unsigned, scalingfactor 1, unit
          valoffset 0.0 (range 0 to 2e+04) min None, max None, default 0.0.

          Startbit normal bit numbering, least significant bit: 26
          Startbit normal bit numbering, most significant bit: 23
          Startbit backward bit numbering, least significant bit: 34

                   111111   22221111 33222222 33333333 44444444 55555544 66665555
          76543210 54321098 32109876 10987654 98765432 76543210 54321098 32109876
          Byte0    Byte1    Byte2    Byte3    Byte4    Byte5    Byte6    Byte7
                            MXXXXXXX XXXXXL
          66665555 55555544 44444444 33333333 33222222 22221111 111111
          32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210
```

Run the Canadaper:

```
$ python3 scripts/canadapter.py examples/configfilesForCanadapter/climateservice_cansignals.kcd \
-mqttfile examples/configfilesForCanadapter/climateservice_mqttsignals.json -mqttname climateservice
```

Send a CAN frame with ID=8:

```
$ cansend vcan0 008#00FF00FF00000000
```

Study the traffic on the CAN bus:

```
$ candump vcan0
```

Resulting output:

```
vcan0  008   [8]  00 FF 00 FF 00 00 00 00
```

Study the resulting MQTT messages using this command:

```
$ mosquitto_sub -v -t +/#
```

Resulting MQTT messages:

```
data/climateservice/vehiclespeed 2.5500000000000003
data/climateservice/enginespeed 63.0
```

7.3 CAN adapter helptext

```
$ python3 scripts/canadapter.py -h
usage: canadapter.py [-h] [-mqttfile MQTTFILE] [-v] [-version] [-i INTERFACE]
                     [-host HOST] [-port PORT] [-qos {0,1,2}] [-k KEEPALIVE]
                     [-cert CERT] [-busname BUSNAME] [-mqttname MQTTNAME]
                     [-listentoallcan] [-bcm] [-t THROTTLINGTIME]
                     [-ego EGO [EGO ...]]
                     kcdfile

positional arguments:
  kcdfile             File name for CAN bus definition (in KCD file format).
```

```
optional arguments:
  -h, --help          show this help message and exit
  -mqttfile MQTTFILE  File name for mqtt-and-CAN signal name and permission
                      translation (JSON). Defaults to listen to all CAN
                      signals (no JSON file used).
  -v                  Increase verbosity level. Can be repeated.
  -version            show program's version number and exit
  -i INTERFACE        CAN interface name. Defaults to 'vcan0'.
  -host HOST          Broker host name. Defaults to 'localhost'.
  -port PORT          Broker port number. Defaults to 1883.
  -qos {0,1,2}        MQTT quality-of-service setting. Defaults to '0'.
  -k KEEPALIVE        Set keepalive time for MQTT communication to the broker,
                      in seconds. Defaults to 10 seconds.
  -cert CERT          Directory for certificate files. Defaults to not using
                      certificates.
  -busname BUSNAME    CAN Bus name in the KCD file. Defaults to use the first
                      busname (alphabetically).
  -mqttname MQTTNAME  Resource name for MQTT topics. Defaults to 'canadapter'.
  -listentoallcan     Listen to all CAN signals, and send them on MQTT using
                      the same signalname. Will be overrided by -mqttfile
                      option.
  -bcm                Use broadcast manager (BCM) for periodic sending of CAN
                      frames by the Linux kernel. Defaults to not using the
                      broadcast manager. See documentation for can4python.
                      Requires Python 3.4 or later.
  -t THROTTLINGTIME   Set throttling time (max update rate) for incoming
                      frames, in milliseconds. Is automatically setting the
                      '-bcm' option. Defaults to not throttle incoming frame
                      rate.
  -ego EGO [EGO ...]  Set ego node id (string), for defining which of the
                      frames in the KCD file should be sent. By default other
                      frames are received. Several ids can be given. Defaults
                      to '['1']'. See KCD file definition documentation.

A CAN adapter for the Secure Gateway concept architecture

This is a "Resource" according to the Secure Gateway nomenclature. It registers
on the Secure Gateway network, and accepts commands that will be sent over the
CAN network. Signals received from the CAN network are forwarded to
the Secure Gateway MQTT (over IP) network.

It is intended for running on a Linux machine having a CAN interface,
for example a Beaglebone or Raspberry Pi with appropriate expansion boards
(capes/HATs). It can also be used on simulated (virtual) CAN buses on Linux.

It requires a MQTT broker (for example Mosquitto), and a MQTT client library (Paho).

This resource can connect to the broker in a secure or insecure way.
The settings of the broker determines what is allowed. To connect in the secure way,
the directory of the certificate files must be specified.

The certificate files should be named:
  CA file:          ca_public_certificate.pem
  Certificate file: public_certificate.pem
  Key file:         private_key.pem
```

CHAPTER 8

Security usage

Certificates (signed public keys) are used in the Secure Gateway to provide authentication. The certificates are signed by a Certificate Authority (CA). Typically that is a trusted third party, but here we will create a self-signed CA.

When a client connects to the broker, the client needs three files:

- Client certificate. This is the public key that will be sent to the broker, so that the broker can encrypt messages when sending to the client.
- Client private key. This is used by the client to unlock encrypted messages that it receives.
- CA (Certificate Authority) certificate. This is used by the client to verify that the broker is the one it is claiming to be.

The broker will use a corresponding set of files.

See the separate tutorial in the examples section.

CHAPTER 9

Examples for sgframework

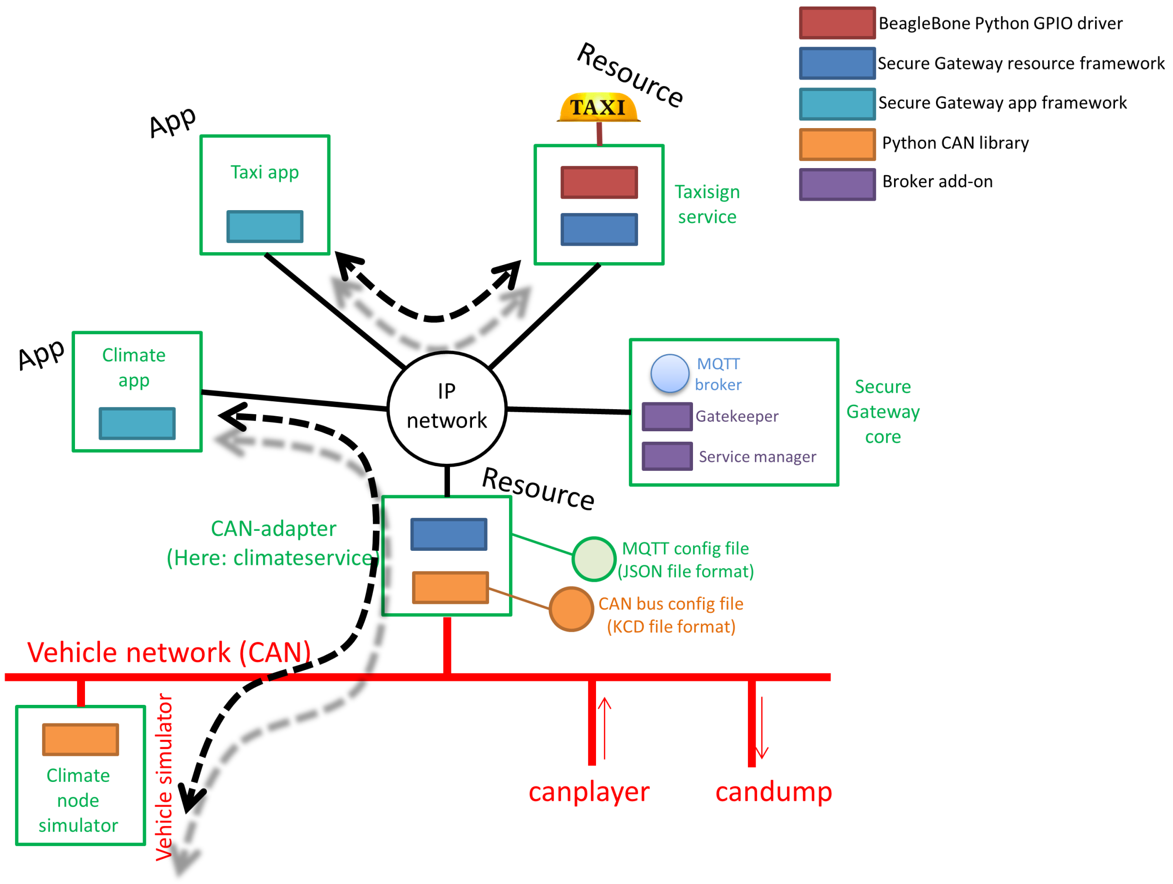
This section describes the examples distributed with the source for sgframework.

The first example describes an after-market hardware attached to the Secure Gateway IP-based network. It is a taxisign (resource) controlled by an app.

Second example shows the usage of certificates for encrypted communication discussed before.

Finally, in the third example is a CAN-bus connected vehicle simulator (for a climate node) controlled by an infotainment app.

An overview of the network and software components is shown in the image below.



9.1 Taxi sign and corresponding app

As an example resource, a taxi sign service has been created. It is one rather naive example of what could be added to a passenger car and would benefit from having a user interface in the infotainment head unit.

This taxi sign resource is a graphical application for running on Ubuntu, and will show a taxi sign lit up or turned off. The resource can also be used in command-line only mode (for use on Linux machines without graphics) or even connected to a real taxi sign with a light bulb. See below how to build your own taxi sign hardware controlled by a Beaglebone!

9.1.1 Taxi sign resource

Minimum Python 3.3 should be used for this software to run properly.

To find usage information on this (and other scripts mentioned in this section), use the -h command line switch:

```
$ python3 examples/taxisignservice/taxisignservice.py -h
```

The taxi sign service listens to this command:

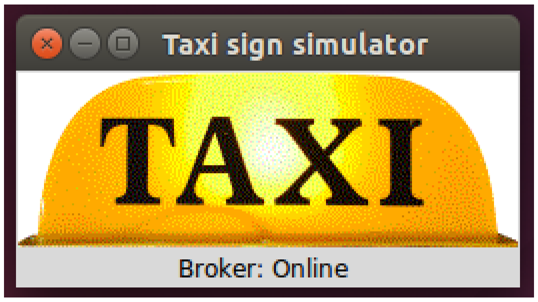
```
command/taxisignservice/state True
```

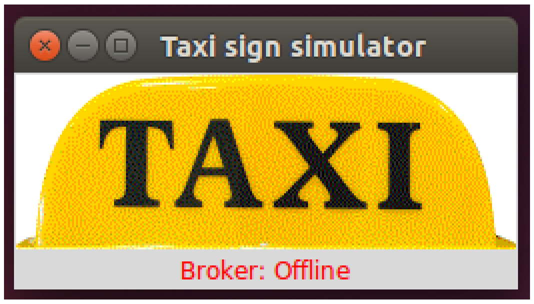
and it will respond on the corresponding data topic.

To test the taxisign resource, run this in three different terminal windows:

```
$ mosquitto_sub -t +/#  -v
$ python3 examples/taxisignservice/taxisignservice.py -v -mode graphical
$ mosquitto_pub -t command/taxisignservice/state -m True
```

The last command will turn on the taxi sign. Using 'False' as payload will turn it off. This is how the taxi sign looks like when on, and off (and in addition disconnected from the broker) respectively:





As seen in the mosquitto_sub window, also availability data is sent upon startup:

```
resourceavailable/taxisignservice/presence True
commandavailable/taxisignservice/state True
dataavailable/taxisignservice/state True

data/taxisignservice/state False
```

Kill the taxisignservice process to see the last will being sent from the broker. The service manager will send out the `presence` information for the individual signals.

Try it using:

```
$ ps -ef
$ kill <PID for taxisignservice>
```

9.1.2 Taxi sign application

The taxi sign app (application) is a graphical program that is turning on or off the taxi sign (simulated or real hardware). The taxi sign app can also be used in command-line only mode (for use on Linux machines without graphics).

First test the taxi sign app standalone, with a broker only (make sure the broker is running). Start the taxi sign app in graphical mode:

```
$ python3 examples/taxisignapp/taxisignapp.py -mode graphical
```

In a separate terminal window, send information to the taxi sign app that the taxi sign resource is online and that the sign is lit up:

```
$ mosquitto_pub -t resourceavailable/taxisignservice/presence -m True
$ mosquitto_pub -t data/taxisignservice/state -m True
```

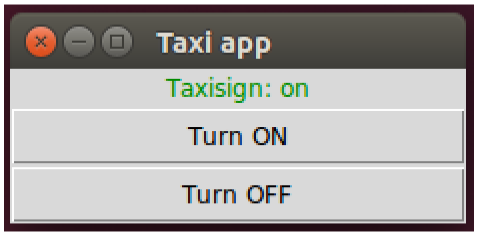
In yet another terminal window, listen to all MQTT commands:

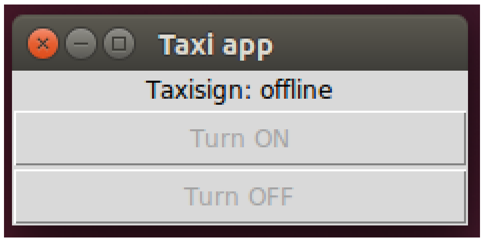
```
$ mosquitto_sub -t +/#  -v
```

Then press the buttons on the taxi sign app. It will show something like this:

```
command/taxisignservice/state True
command/taxisignservice/state False
```

This is how the taxi sign app looks like, when the taxi sign is on, and when the taxi sign is disconnected from the broker:





Now it is time to test the taxi sign app and the taxi sign resource together. Run these in separate terminal windows:

```
$ python3 examples/taxisignapp/taxisignapp.py -v -mode graphical
$ python3 examples/taxisignservice/taxisignservice.py -v -mode graphical
```

Try for example to kill the broker and then start it again.

9.1.3 Help texts for the taxi sign resource and app

Details of usage for the taxi sign related examples are found here:

```
$ python3 examples/taxisignservice/taxisignservice.py -h
usage: taxisignservice.py [-h] [-v] [-host HOST] [-port PORT] [-cert CERT]
                          [-mode {hardware,commandline,graphical}]

optional arguments:
  -h, --help            show this help message and exit
  -v                    Increase verbosity level. Can be repeated.
  -host HOST            Broker host name. Defaults to localhost.
  -port PORT            Broker port number. Defaults to 1883.
  -cert CERT            Directory for certificate files. Defaults to not using
                        certificates.
  -mode {hardware,commandline,graphical}
                        Type of simulator to use. Depends on graphical display
                        or taxi sign hardware. Defaults to 'commandline'.

A taxi sign service example for the Secure Gateway concept architecture.

This is a "Resource" according to the Secure Gateway nomenclature. It registers on
the Secure Gateway network, and accepts commands to turn on or off a
hardware taxi sign. It is intended for running on Beaglebone with appropriate
electronics connected to a GPIO output pin to contol the taxi sign. Note that
root/sudo permissions typically are required to control GPIO pins.

This resource sends out on start-up:
  resourceavailable/taxisignservice/presence True
  commandavailable/taxisignservice/state True
  datavailable/taxisignservice/state  True

This resource is listening for:
  command/taxisignservice/state True/False

This resource sends out on state change:
  data/taxisignservice/state True/False

It can also be used in two different simulation modes. This is handy when no
taxisign hardware is available (or no sudo/root permission is available). The
command line mode simulator should always be available. The graphical mode
simulator requires Tk installed on the machine. This is typically installed with:
  sudo apt-get install python3-tk

This resource can connect to the broker in a secure or insecure way. The settings
of the broker determines what is allowed. To connect in the secure way,
the directory of the certificate files must be specified.

The certificate files should be named:
  CA file:          ca_public_certificate.pem
  Certificate file: public_certificate.pem
  Key file:         private_key.pem
```

```
$ python3 examples/taxisignapp/taxisignapp.py -h
usage: taxisignapp.py [-h] [-v] [-host HOST] [-port PORT] [-cert CERT]
                      [-mode {commandline,graphical}]

optional arguments:
  -h, --help            show this help message and exit
  -v                    Increase verbosity level. Can be repeated.
  -host HOST            Broker host name. Defaults to localhost.
  -port PORT            Broker port number. Defaults to 1883.
  -cert CERT            Directory for certificate files. Defaults to not using
                        certificates.
  -mode {commandline,graphical}
                        Type of use interface. Depends on graphical display.
                        Defaults to 'commandline'.

A taxi sign app example for the Secure Gateway concept architecture.

This is an "App" according to the Secure Gateway nomenclature. It registers on
the Secure Gateway network, and sends commands to turn on or off a hardware
taxi sign (or a simulated sign).

The corresponding taxisign resource must be online. This app listens for:
  resourceavailable/taxisignservice/presence True

This app is sending:
  command/taxisignservice/state True

This app is receiving:
  data/taxisignservice/state True

It can be used in two different modes. The command line mode should always
be available. The graphical mode requires Tk installed on the machine.
This is typically installed with:
  sudo apt-get install python3-tk

This app can connect to the broker in a secure or insecure way. The settings
of the broker determines what is allowed. To connect in the secure way,
the directory of the certificate files must be specified.

The certificate files should be named:
  CA file:          ca_public_certificate.pem
  Certificate file: public_certificate.pem
  Key file:         private_key.pem
```

9.1.4 Build your own Secure Gateway enabled taxi sign hardware

The taxisignservice resource software is handling turning on and off a digital output pin on a BeagleBone or Raspberry Pi embedded Linux computer board. Use the command:

```
$ sudo python3 taxisignservice.py -v -mode hardware -host 192.168.0.93
```

Replace the IP number with your broker's IP number. Due to the hardware manipulation on the Beaglebone/Pi, you must run the program in 'sudo' mode.

Connect the output GPIO pin to a relay driver. The GPIO pin number to use is defined in the taxisigndriver file, and there is also more technical information in that file. A power supply and a lightbulb are connected to the relay. Cheap taxi sign fixtures are available on several of the common low-price marketplaces on the Internet.

Set the broker IP number manually when starting the taxisignservice, or write a small startscript that uses Avahi to find the broker IP number.

To test the GPIO output driver:

```
$ sudo python3 examples/taxisignservice/drivers/taxisign_driver.py
```

9.2 Certificate usage tutorial

Below is the procedure to generate test certificates, and similar example certificates are distributed among the examples. The example server certificate holds the host name, and assumes that the broker is running on 'localhost'. Note that the settings are optimized for being an easy-to-use example, rather than production strength security.

9.2.1 Certificate Authority

Generate a private key for the CA (certificate authority):

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out ca_private_key.pem
```

Generate the public CA certificate:

```
$ openssl req -new -x509 -days 3650 -key ca_private_key.pem -subj "/C=SE/O=TEST" -out ca_public_certi
```

9.2.2 Server certificate

Generate a private key for the server:

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out server_private_key.pem
```

The server certificate will have the host name as its CN (Common Name), and this must be the address that clients connects to. It can be for example "www.example.com", "11.22.33.44" or "localhost". When deploying the server in different locations you need to adapt this (the certificate must be re-generated if the broker IP-number is changed). A few examples are given below.

Request a server certificate from the CA:

```
$ openssl req -new -key server_private_key.pem -subj "/C=SE/O=TEST/CN=192.168.0.3" -out server_reques
$ openssl req -new -key server_private_key.pem -subj "/C=SE/O=TEST/CN=localhost"   -out server_reques
```

The CA issues a server certificate:

```
$ openssl x509 -req -in server_request.csr -CA ca_public_certificate.pem -CAkey ca_private_key.pem -c
```

Note that is the first time the CA generates a certificate. We ask it to create a file for keeping track of serial numbers (using the -CAcreateserial flag).

Now the `.csr` file can be removed.

9.2.3 Taxisign resource certificate

Generate a private key for the taxi sign:

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out taxisign_private_key.pem
```

Request a (taxi sign) client certificate from the CA. The Common Name (CN) must be unique among the clients to the broker, as we use it as the username:

```
$ openssl req -new -key taxisign_private_key.pem -subj "/C=SE/O=TEST/CN=taxisign" -out taxisign_reque
```

The CA issues a (taxi sign) client certificate:

```
$ openssl x509 -req -in taxisign_request.csr -CA ca_public_certificate.pem -CAkey ca_private_key.pem
```

9.2.4 Taxisign app certificate

Generate a private key for the taxi app:

```
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out taxiapp_private_key.pem
```

Request a (taxi app) client certificate from the CA:

```
$ openssl req -new -key taxiapp_private_key.pem -subj "/C=SE/O=TEST/CN=taxiapp" -out taxiapp_request
```

The CA issues a (taxi app) client certificate:

```
$ openssl x509 -req -in taxiapp_request.csr -CA ca_public_certificate.pem -CAkey ca_private_key.pem -
```

9.2.5 Force the Mosquitto broker to use certificates

Use a modified mosquitto.conf file:

```
port 8883
cafile ca_public_certificate.pem
certfile server_public_certificate.pem
keyfile server_private_key.pem
require_certificate true
use_identity_as_username true

#acl_file acl.txt

#listener 1883
#allow_anonymous true
```

(If you un-comment the two last lines also non-encrypted connections are accepted)

Start Mosquitto from the directory with the configutation and certificate files:

```
SecureGateway/examples/servercertificates$ mosquitto -c mosquitto.conf
```

9.2.6 Testing the certificates from command line

For the clients, put the key and certificate files in a subfolder. Rename the files to:

- public_certificate.pem

- private_key.pem

Also put a copy of the ca_public_certificate.pem in each clients' subfolders.

Test that the broker rejects communication without certificates:

```
$ mosquitto_sub -v -t +/#
```

The moquitto_sub command is accepting certificate files. When using it with certificates, the host IP address must be given exactly as in the certificate file. If not given, the mosquitto_sub will assume that the host is 'localhost' and thus the certificate must have been generated for this host name. Otherwise you must give the hostname/IPnumber explicitly to mosquitto_sub.

You can connect two clients to the broker like this, if you have 'localhost' as CN in the server/broker certificate:

```
SecureGateway/examples/taxisignapp/certificates$     mosquitto_sub -v -t +/# -h localhost -p 8883 --c
SecureGateway/examples/taxisignservice/certificates$ mosquitto_sub -v -t +/# -h localhost -p 8883 --c
```

With a broker running on a server with IP 192.168.0.3, and the server certificate has been generated for that IPnumber:

```
SecureGateway/examples/taxisignapp/certificates$     mosquitto_sub -v -t +/# -h 192.168.0.3 -p 8883 -
SecureGateway/examples/taxisignservice/certificates$ mosquitto_sub -v -t +/# -h 192.168.0.3 -p 8883 -
```

You can also use one mosquitto_pub and one mosquitto_sub to send command line messages between terminal windows.

If you do not want mosquitto_sub to check the server certificate to the server hostname, give the –insecure flag to mosquitto_sub. For example:

```
$ mosquitto_sub -v -t +/# -h localhost --insecure -p 8883 --cafile ca_public_certificate.pem --cert p
```

With the setting "require_certificate false" in the mosquitto.conf file, do not give the –cafile –cert –key options to mosquitto_sub. (Otherwise it will give "Connection Refused: bad user name or password.")

9.2.7 Run the taxisignservice and taxisign app using certificates

After starting the certificate-enabled broker, run this in two separate terminal windows:

```
SecureGateway/examples/taxisignapp$     python3 taxisignapp.py     -mode graphical -host localhost -p
SecureGateway/examples/taxisignservice$ python3 taxisignservice.py -mode graphical -host localhost -p
```

All distributed example apps and resources can use certificates.

9.3 CAN communication tutorial, using simulated CAN bus

It is possible to create a virtual (simulated) CAN bus on Linux systems. This can be used to simulate the activity of a real CAN bus, and for testing CAN software. Install a virtual CAN bus as described elsewhere in this documentation, and name it vcan0 (look at DEPENDENCIES.rst file for installation commands).

9.3.1 Linux CAN command-line tools

In order to test the CAN communication, we are using the can-utils command line CAN tools. These are used similarly on real and simulated CAN buses. For example, one of the tools is `candump` which allows you to print all data that is being received by a CAN interface.

In order to test this facility, start it in a terminal window:

```
$ candump vcan0
```

From another terminal window, send a CAN frame with identifier 0x1A (26 dec) and 8 bytes of data:

```
$ cansend vcan0 01a#11223344AABBCCDD
```

This will appear in the first terminal window (running candump):

```
vcan0  01A   [8]  11 22 33 44 AA BB CC DD
```

To send large amount of random CAN data, use the cangen tool:

```
$ cangen vcan0 -v
```

In order to record this type of received CAN data to file (including timestamp), use:

```
$ candump -l vcan0
```

The resulting file will be named like: candump-2015-03-20_123001.log

In order to print logfiles in a user friendly format:

```
$ log2asc -I candump-2015-03-20_123001.log vcan0
```

Recorded CAN log files can also be re-played back to the same or another CAN interface:

```
$ canplayer -I candump-2015-03-20_123001.log
```

If you need to use another can interface than defined in the logfile, use the expression `CANinterfaceToUse=CANinterfaceInFile`. This example also prints the frames:

```
$ canplayer vcan0=can1 -v -I candump-2015-03-20_123001.log
```

The cansniffer command line application is showing the latest CAN messages. Start it with:

```
$ cansniffer vcan0
```

It shows one CAN-ID (and its data) per line, sorted by CAN-ID, and shows the cycle time per CAN-ID. The time-out until deleting a CAN-ID row is 5 seconds by default.

There is an example CAN log file distributed with the Secure Gateway. Download it, replay it, and study the result using cansniffer.

Also the Wireshark program can be used to analyse CAN frames.

There is a description on how to analyze CAN using Wireshark: https://libbits.wordpress.com/2012/05/07/capturing-and-analyzing-can-frames-with-wireshark/ Make sure to enable the CAN interface before starting the program.

9.3.2 Setting up CAN communication between two embedded Linux boards

In order to test real CAN communication, you need two embedded Linux machines, for example Raspberry Pi and Beaglebone. Both should be equipped with CAN interface boards, set to a speed of 500 kbit/s. The installation of software and hardware is described elsewhere in this documentation.

Test the communication using command line tools. Send a CAN frame from one of the machines:

```
$ cansend can0 01a#01020304
```

This will be repeatedly sent on the CAN bus until there is an acknowledgement from at least one other node.

For the CAN controller to be able to send a message, a CAN transceiver must be connected (as it senses the CAN bus voltage). Otherwise it will stop immediately after the first try.

To cancel this sending, you need to disable and re-enable the can0:

```
$ sudo ip link set can0 down
$ sudo ip link set can0 up
```

On the other machine receive the CAN frames using:

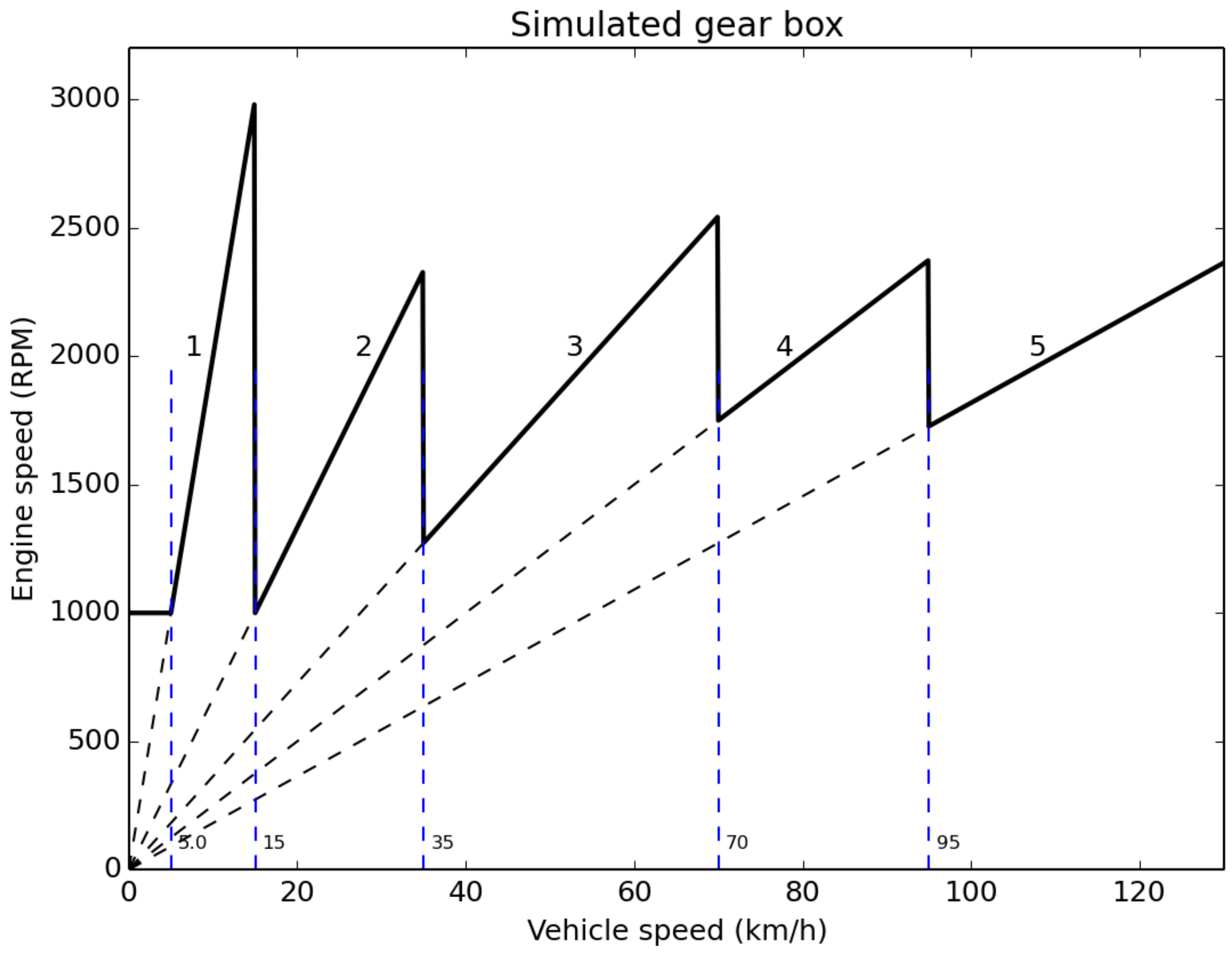
```
$ candump can0
```

9.4 Vehicle simulator and corresponding app

9.4.1 Simulated climate node (which is a CAN simulator or vehicle simulator)

There is a vehicle simulator available among the Secure Gateway examples. It will send out CAN messages on a real or simulated CAN bus. The CAN messages contain signals describing the vehicle speed, the engine speed and the in-car temperature. The simulator listens to control signals on the CAN bus to turn on or off the simulated air condition (affects the in-car temperature).

The vehicle speed is increased in small, random steps. When reaching the maximum speed, it is instead decreased in small, random steps. The engine speed is calculated from the vehicle speed, as seen in this figure:



To test the vehicle simulator, run these commands in two different terminal windows:

```
$ candump vcan0
$ python3 examples/vehiclesimulator/vehiclesimulator.py -v
```

This will print out the simulated values, and the send CAN frames, respectively.

In order to turn on the simulated air condition, run this command from yet another terminal window:

```
$ cansend vcan0 007#8000000000000000
```

To turn off the air condition:

```
$ cansend vcan0 007#0000000000000000
```

Note the resulting changes in the simulated in-car temperature.

9.4.2 Climate resource (converts CAN messages to MQTT)

The climate resource converts data from CAN frames, and sends it to MQTT messages. It also converts commands in MQTT messages to CAN frames. The climate resource is implemented using the canadapter script with appropriate configuration files.

These MQTT topics are used by the canadapter (configured to run as a climate resource named climateservice):

```
command/climateservice/aircondition 1
data/climateservice/aircondition 1
data/climateservice/actualindoortemperature 27.5
data/engineservice/vehiclespeed 67.5
data/engineservice/enginespeed 2354
```

Basically it sends out MQTT data describing the vehiclespeed, enginespeed etc, and listens to MQTT commands to turn the air condition on and off.

In addition these availability topics are sent at startup:

```
resourceavailable/climateservice/presence True

dataavailable/climateservice/actualindoortemperature True
dataavailable/climateservice/vehiclespeed True
dataavailable/climateservice/enginespeed True

commandavailable/climateservice/aircondition True
dataavailable/climateservice/aircondition True
```

To test the canadapter, run these commands in three different terminal windows (from sgframework project root directory, after installation):

```
$ python3 examples/vehiclesimulator/vehiclesimulator.py -v
$ canadapter examples/configfilesForCanadapter/climateservice_cansignals.kcd -mqttfile examples/conf
$ mosquitto_sub -t +/# -v
```

In order to turn on and off the air condition, run this command in yet another terminal window:

```
$ mosquitto_pub -t command/climateservice/aircondition -m 1
```

An example of data on the canbus (as printed by `candump vcan0`):

```
vcan0  009   [8]  03 32 00 00 00 00 00 00
vcan0  008   [8]  0F E9 17 24 00 00 00 00
vcan0  007   [8]  80 00 00 00 00 00 00 00
vcan0  009   [8]  03 2A 00 00 00 00 00 00
vcan0  008   [8]  10 0B 17 54 00 00 00 00
```

Note that the aircondition command is echoed back (as data) from the canadapter, not from the vehiclesimulator itself. This means that the `data/climateservice/aircondition` signal might be out of sync with the state of the vehicle simulator. A more advanced usage would be to echo the command in a separate CAN message from the vehiclesimulator, and convert that information to the mentioned MQTT message. This can easily be set for the canadapter by using the configuration file.

Several canadapters can be running in parallel, each handling a few CAN messages. These canadapters are then different resources on the Secure Gateway network, and could require different permission levels for usage.

Use as few CAN definition files as possible (in the KCD file format), as canadapter uses message filtering in the Linux kernel for the messages defined in the file.

Generate data in the 'climatesimulationdata' CAN frame (id 0x009, once per 100 ms):

```
$ cangen vcan0 -I 009 -L 8 -D i -g 100
```

Generate data in the 'vehiclesimulationdata' CAN frame (id 0x008, once per 100 ms):

```
$ cangen vcan0 -I 008 -L 8 -D i -g 100
```

9.4.3 Climate MQTT app

Distributed with the Secure Gateway is a climate app listening to the climate service and controlling the air condition. It has a graphical user interface, but can also be used from the command line (useful for embedded Linux boards).

Start the app:

```
$ python3 examples/climateapp/climateapp.py -mode graphical
```

To test the climate app standalone, with a broker only:

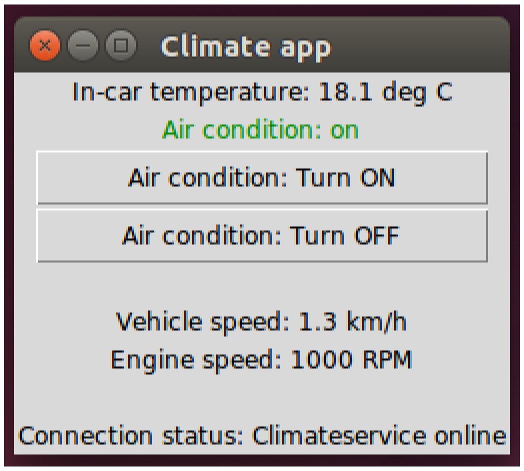
```
$ mosquitto_pub -t resourceavailable/climateservice/presence -m True
$ mosquitto_pub -t data/climateservice/vehiclespeed -m 27.1
$ mosquitto_pub -t data/climateservice/enginespeed -m 1719
$ mosquitto_pub -t data/climateservice/actualindoortemperature -m 31.6
$ mosquitto_pub -t data/climateservice/aircondition -m 1


$ mosquitto_sub -t +/#  -v
```

Or you can use a one-liner for the above mosquitto commands:

```
$ mosquitto_pub -t resourceavailable/climateservice/presence -m True && mosquitto_pub -t data/climate
```

Next, the climate app should be tested together with the vehicle simulator and the canadapter. Then it will show the present in-car temperature and can control the air condition. Also the vehicle speed, engine speed, connection status to the broker and the presence information for the climateservice will be displayed.

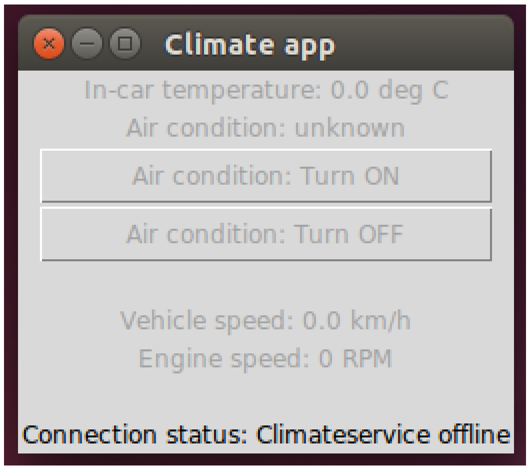
This is how the climate app looks like when the air condition is on, and when the climate service is disconnected from the broker:



In-car temperature: 18.1 deg C
Air condition: on

Vehicle speed: 1.3 km/h
Engine speed: 1000 RPM

Connection status: Climateservice online



In-car temperature: 0.0 deg C
Air condition: unknown

Vehicle speed: 0.0 km/h
Engine speed: 0 RPM

Connection status: Climateservice offline

Instead of using the vehiclesimulator, the climateapp can be tested with the recorded CAN log file and the canadapter.

9.4.4 CAN communication using real CAN bus

Set up the CAN communication between the two embedded Linux machines as described earlier.

On the first machine run the vehicle simulator using the real CAN interface:

```
$ python3 examples/vehiclesimulator/vehiclesimulator.py -v -i can0
```

On the second machine, verify that it receives CAN data:

```
$ candump can0
```

To turn on the air condition in the vehicle simulator (on the first machine) run this on the second machine:

```
$ cansend can0 007#8000000000000000
```

Then on the second machine run the canadapter using the real CAN interface:

```
$ canadapter examples/configfilesForCanadapter/climateservice_cansignals.kcd -mqttfile examples/conf
```

Note that the broker must be running.

On the second machine, verify that MQTT messages are sent:

```
$ mosquitto_sub -t +/# -v
```

Run the climate app (non-graphical mode) on the second machine (press and hold Enter key for continuous updates):

```
$ python3 examples/climateapp/climateapp.py
```

To run the climate app in graphical mode, use a laptop connected to the same network as the machine running the canadapter. Verify that the laptop receives the MQTT messages (use the IP number of the broker, typically on the second machine):

```
$ mosquitto_sub -t +/# -v -h IPNUMBER_OF_BROKER
```

Run the climate app in graphical mode on the laptop:

```
$ python3 examples/climateapp/climateapp.py -mode graphical -host IPNUMBER_OF_BROKER
```

9.4.5 Help texts for the vehicle simulator and the climate app

```
$ python3 examples/vehiclesimulator/vehiclesimulator.py -h
usage: vehiclesimulator.py [-h] [-v] [-i INTERFACE]

optional arguments:
  -h, --help    show this help message and exit
  -v            Increase verbosity level. Can be repeated.
  -i INTERFACE  CAN interface name. Defaults to vcan0.
```

```
$ python3 examples/climateapp/climateapp.py -h
usage: climateapp.py [-h] [-v] [-host HOST] [-port PORT] [-cert CERT]
                     [-mode {commandline,graphical}]

optional arguments:
  -h, --help            show this help message and exit
  -v                    Increase verbosity level. Can be repeated.
  -host HOST            Broker host name. Defaults to localhost.
  -port PORT            Broker port number. Defaults to 1883.
  -cert CERT            Directory for certificate files. Defaults to not using
                        certificates.
  -mode {commandline,graphical}
                        Type of use interface. Depends on graphical display.
                        Defaults to 'commandline'.

A vehicle app example for the Secure Gateway concept architecture.

This is an "App" according to the Secure Gateway nomenclature. It registers on
the Secure Gateway network, and receives vehicle data. It can also send commands
to the CAN-adapter to turn on the air condition.

It can be used in two different modes. The command line mode should always be
available. The graphical mode requires Tk installed on the machine.
This is typically installed with:
  sudo apt-get install python3-tk

This app can connect to the broker in a secure or insecure way. The settings
of the broker determines what is allowed. To connect in the secure way,
the directory of the certificate files must be specified.

The certificate files should be named:
```

```
CA file:          ca_public_certificate.pem
Certificate file: public_certificate.pem
Key file:         private_key.pem
```

CHAPTER 10

Other info

10.1 Network discovery

TODO! For the client to find the IP number of the broker, the Avahi system can be used. The broker publishes its connection information, and the clients search for this information.

Description

installation

Command line usage

Taxi sign app with startup script for node discovery.

10.2 Making apps for Android

Making a simple Android application to control a hardware taxi sign using MQTT protocol

10.2.1 Create a new Android project

Create a simple Android application from the editor of your choice. A simple app with a MainActivity is fine.

10.2.2 Setup necessary Android permissions

Navigate to the Android Manifest file of your project and add the following permissions:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.BROADCAST_STICKY"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
```

10.2.3 MainActivity

A number of steps is required to the source code of the MainActivity file.

1. Extend the Activity to implement MQTTCallBack

```
public class MainActivity extends Activity implements MqttCallback {
```

The editor will automatically add 3 unimplemented methods

```
@Override
 public void connectionLost(Throwable throwable) {

 }

@Override
 public void messageArrived(final String topic, MqttMessage message) {

}
@Override
 public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {

}
```

2. Declare all the necessary variables

```
/* MQTT  required variables */
String clientId = "MobileApp";
MemoryPersistence persistence = new MemoryPersistence();
MqttClient mqttClient;

/* MQTT subscribe and publish topics */
String taxi_subscribe_mqtt_topic_presence = "resourceavailable/taxisignservice/presence";
String taxi_subscribe_mqtt_topic = "data/taxisignservice/state";
String publish_mqtt_topic = "command/taxisignservice/state";

/* Android UI Elements */
ToggleButton toggleTaxiSighSwitch;
ImageView imageViewTaxiSign;
TextView tvSecureGateAway;
ImageView imView_semcon_icon_bottom;
```

3. Create the Client

```
public void create_MQTT_Client() {
   ApplicationLayer app = (ApplicationLayer) getApplicationContext();
   String brokers_ip = app.getBrokers_ip();
   String secure_gateway_broker = "tcp://"+ brokers_ip + ":1883";

         try {
             mqttClient = new MqttClient(secure_gateway_broker, clientId, persistence);
             MqttConnectOptions mqttConnOpt = new MqttConnectOptions();
             mqttConnOpt.setCleanSession(true);
             mqttClient.setCallback(this);
             mqttClient.connect();
             mqttClient.subscribe(taxi_subscribe_mqtt_topic);
             mqttClient.subscribe(taxi_subscribe_mqtt_topic_presence);
         }
         catch (MqttException e) {
             e.printStackTrace();
         }
 }
```

Note: The client should be called inside the Activity function so it will be bound with the same UI Thread.

4. Create the methods to send MQTT messages

```
public void send_MQQT_Message(String topic, String payLoad) {
       try {
           MqttMessage message = new MqttMessage(payLoad.getBytes());
           mqttClient.publish(topic, message);
       } catch (MqttException me) {
           me.printStackTrace();
       }
   }
```

5. Create a switch statement to handle the different received messages. The topics that the client is subscribed to are the ones declared on step 2. The statement must be inside the messageArrived function we added at step 1.

```
@Override
public void messageArrived(final String topic, MqttMessage message) {
    final String received_message = message.toString();
    MainActivity.this.runOnUiThread(new Runnable() {
        @Override
        public void run() {
            if (topic.equals(taxi_subscribe_mqtt_topic_presence)) {
                if(received_message.equals("True")){
                imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_off);
                toggleTaxiSighSwitch.setEnabled(true);
                toggleTaxiSighSwitch.setText("\n" + "Taxi Sign");
                }
                else if(received_message.equals("False")){
                imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_offline);
                toggleTaxiSighSwitch.setEnabled(false);
                toggleTaxiSighSwitch.setText("\n" + "Taxi Sign");
                }
             }
            else if(topic.equals(taxi_subscribe_mqtt_topic)){
                if(received_message.equals("False")) {
                    imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_off);
                    toggleTaxiSighSwitch.setChecked(false);
                    toggleTaxiSighSwitch.setText("\n" + "Taxi Sign");
                }
                else if(received_message.equals("True")) {
                    imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_on);
                    toggleTaxiSighSwitch.setChecked(true);
                    toggleTaxiSighSwitch.setText("\n" + "Taxi Sign");
                }
            }
        }

    });
}
```

6. Final step in the MainActivity is to add the on-click Listeners on the buttons in order to control the sign.

```
toggleTaxiSighSwitch = (ToggleButton) findViewById(R.id.switch1);
       toggleTaxiSighSwitch.setText("\n" + "Taxi Sign");
       toggleTaxiSighSwitch.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener()
           public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
               if (isChecked) {
                   toggleTaxiSighSwitch.setOnClickListener(new View.OnClickListener() {
                       @Override
                       public void onClick(View v) {
                           send_MQQT_Message(publish_mqtt_topic, "True");
                           imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_on);
```

```
                        }
                    });
                } else {
                    toggleTaxiSighSwitch.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                            send_MQQT_Message(publish_mqtt_topic, "False");
                            imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_off);
                        }
                    });
                }
            }
        });

        imageViewTaxiSign = (ImageView) findViewById(R.id.imTop);
        imageViewTaxiSign.setBackgroundResource(R.drawable.taxi_sign_offline);

        tvSecureGateAway = (TextView) findViewById(R.id.tvsecureGateway);
        tvSecureGateAway.setText(getResources().getString(R.string.txView));

        imView_semcon_icon_bottom = (ImageView) findViewById(R.id.imageView2);
        imView_semcon_icon_bottom.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(getApplicationContext(), SettingsActivity.class);
                startActivity(intent);
            }
        });
```

7. Finally create the simple layout that contains all the buttons.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/white"
    android:orientation="vertical"
    android:weightSum="1">

    <ImageView
        android:id="@+id/imTop"
        android:layout_gravity="center"
        android:layout_width="540dp"
        android:layout_height="180dp"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="130dp" />

    <ToggleButton
        android:layout_width="180dp"
        android:layout_height="320dp"
        android:id="@+id/switch1"
        android:enabled="false"
        android:textSize="25dp"
        android:textColor="@color/gray"
        android:gravity="center|bottom"
        android:background="@drawable/toggle_button_selector"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        />

    <ImageView
        android:layout_width="250dp"
        android:layout_height="120dp"
        android:src="@drawable/sg_transparent"
        android:id="@+id/imageView"
        android:layout_below="@+id/switch1"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginTop="45dp" />

    <TextView
        android:layout_width="500dp"
        android:layout_height="wrap_content"
        android:gravity="left"
        android:textSize="20dp"
        android:layout_marginLeft="200dp"
        android:id="@+id/tvsecureGateway"
        android:layout_alignTop="@+id/imageView"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true" />

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="20dp"
        android:src="@drawable/semocon_logo_buttom"
        android:layout_alignParentBottom="true"
        android:layout_alignParentLeft="true"
        android:layout_alignParentStart="true"
        android:layout_marginBottom="40dp"
        android:id="@+id/imageView2" />

</RelativeLayout>
```

10.3 Introduction to access control lists

The certificates discussed above handle the authentication, which identifies each client (and the server/broker).

Authorization is about defining which client should be allowed to what and is handled by access control lists (ACL) in the Mosquitto broker.

To test the ACL functionality add this line in the mosquitto.conf file:

```
acl_file acl.txt
```

With this ACL file, the only valid topic is `a/b/c`:

```
topic readwrite a/b/c
```

The permission can be `read`, `write` or `readwrite`.

If a username is given, the `topic` rows below it are valid for that user only. For example:

```
user foo
topic readwrite a/b/c
```

```
user bar
topic read a/b/c
```

Start mosquitto without certificates, but with the ACL functionality enabled that was defined above. Run this in two separate windows:

```
$ mosquitto_sub -t +/# -v -u bar
$ mosquitto_pub -t a/b/c -m 123 -u foo
```

Then try to run the same again, but with two usernames swapped.

It can be useful to run the mosquitto broker with the `-v` flag, to see the details of the communication.

10.4 Dynamically change access to applications

TODO!

Dynamically change access to application

CHAPTER 11

Approximate Canadaper performance

To get an idea of how many CAN frames per second the Canadapter is able to unpack, measurements have been done by running it on various embedded Linux boards.

Methods for measuring the performance are described in another section.

11.1 Summary

The maximum CAN frame unpacking rate is heavily dependent on which embedded Linux hardware the Canadapter is running on. The results are approximately:

- Raspberry Pi version 1: 60 frames/second
- Beaglebone Black: 130 frames/second
- Raspberry Pi version 3: 580 frames/second

The measurements were done with a CAN bitrate of 500 kbit/second.

Note that this is how many of the frames that are actually unpacked. It is perfectly fine to use the Canadapter on CAN buses with higher total frame rates, just adjust the KCD configuration file to select the CAN frames you are interested in.

CAN frames are most often sent peridocally, for example, the CAN frame with information on the vehicle speed might be sent with a period of 10 ms. If it is sufficient for you to have 10 MQTT messages per seconds informing about the vehiclespeed, set the throttling for that CAN frame to 100 ms. This reduces the processing power required by 90%.

11.2 Handle performance issues

The upper limit on the CAN frame unpacking rate is the available processing power. When reaching the upper limit there will be a lag (delay) between the CAN frame input and the MQTT message output, and eventually there will be loss of CAN frames.

Here are a few ideas to improve the performance:

- Unpack only the CAN frames you actually need (controlled by the KCD configuration file).
- Throttle the frequency of incoming CAN frames (achieve a low lag by dropping some of the periodic CAN frames).
- Use MQTT QoS level 0.
- Make sure your hardware is running at max processor frequency.

- Disable the graphical user interface (GUI) in order to save processing power.
- Use a powerful hardware, for example, the Raspberry Pi 3.

11.3 Performance on Beaglebone Black

The CPU and memory consumption have been measured for the different processes on a Beaglebone Black (running without a GUI).

It is seen that the Canadapter process is the most CPU consuming, and the CPU consumption increases linearly with the CAN frame unpacking rate. Also the CPU consumption for the Mosquitto MQTT broker is increasing with the MQTT message rate.

The CPU consumption of the Canadapter process increases until there is no idle CPU power available.

None of the processes seem to consume much memory, but the total memory usage is increasing over time.

The memory usage doesn't seem to limit the CAN frame unpacking rate.

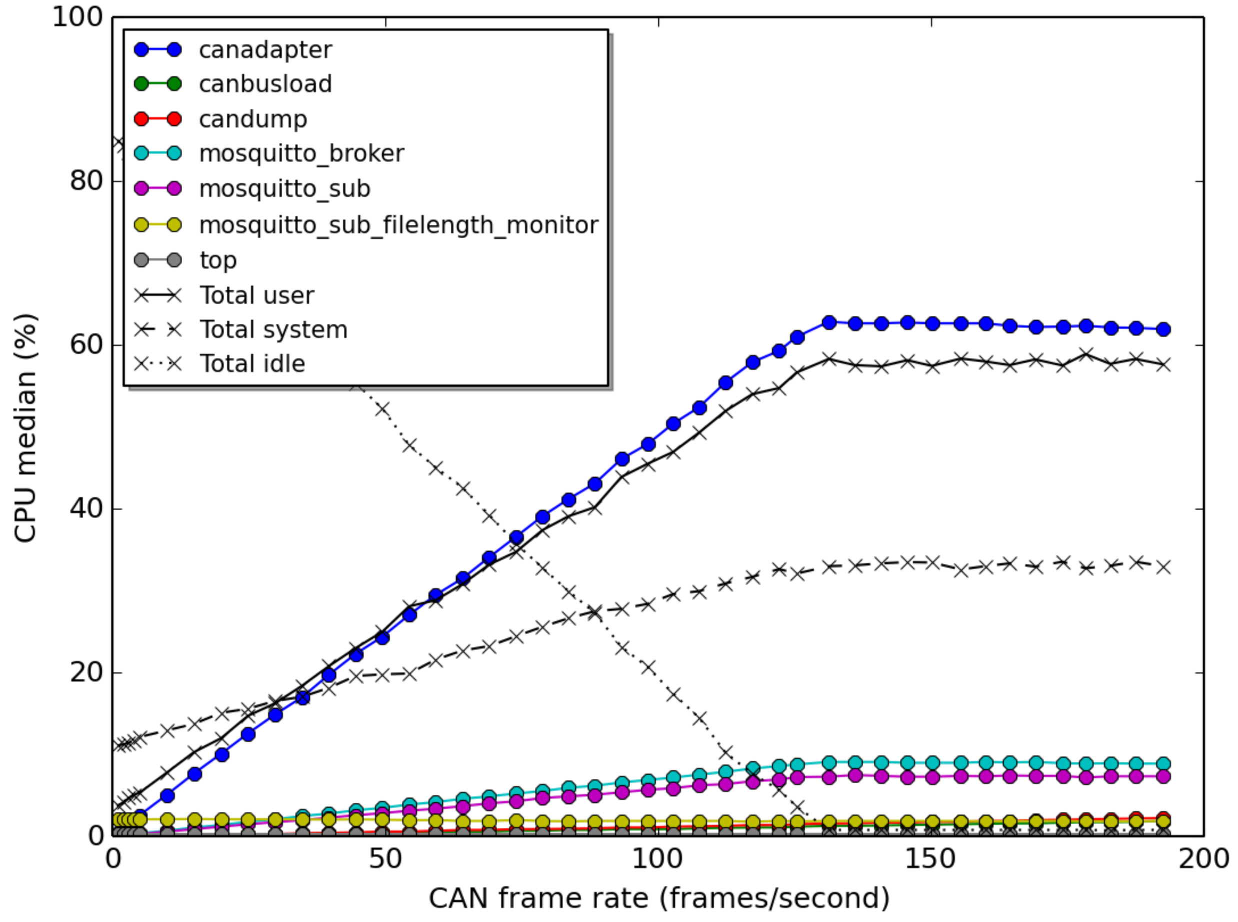


Fig. 11.1: CPU usage for a Beaglebone Black running without a GUI.

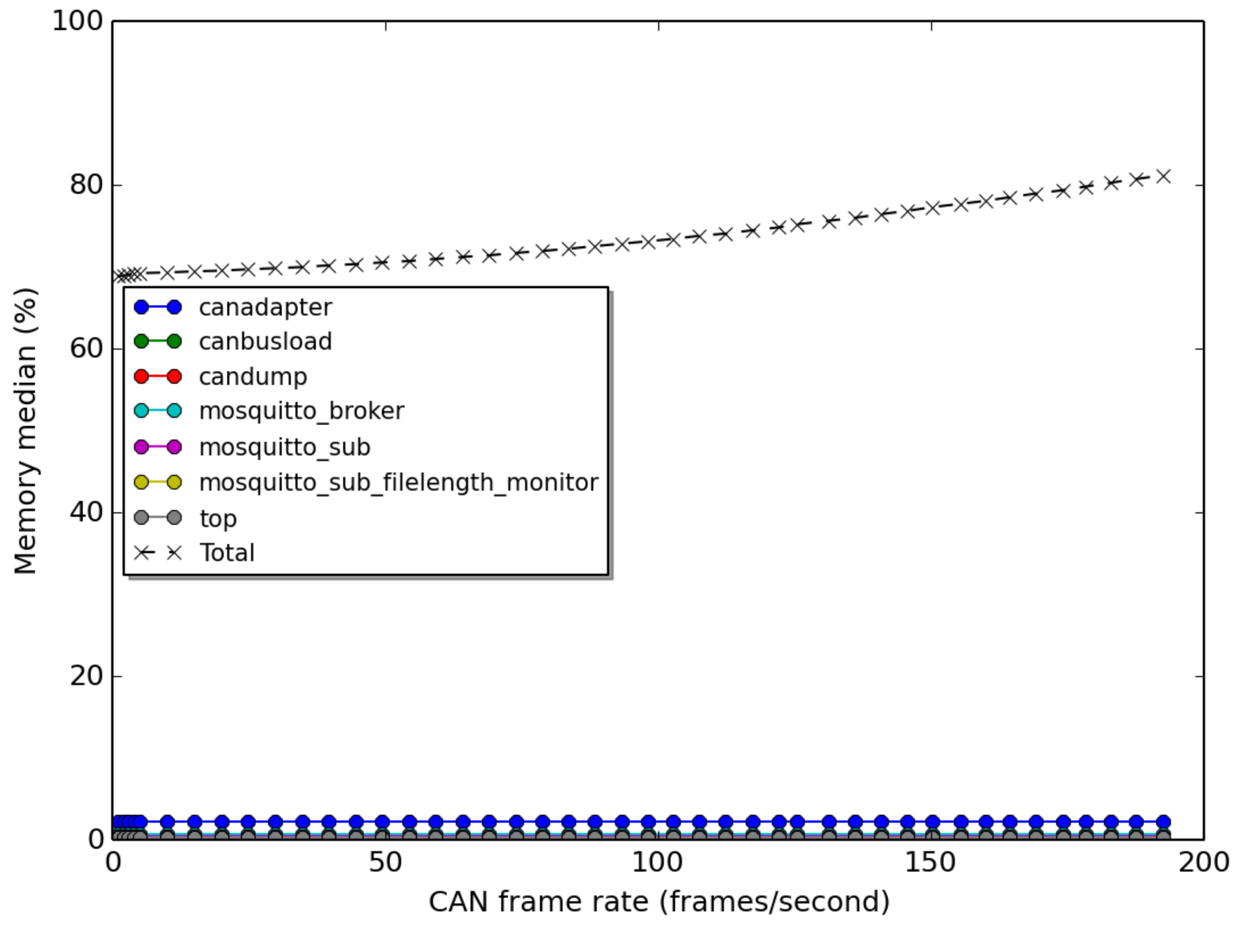


Fig. 11.2: Memory usage for a Beaglebone Black running without a GUI.

11.4 Comparison of boards

These embedded Linux boards have been compared:

- Beaglebone Black (BBB)
- Raspberry Pi 1 (RPi1)
- Raspberry Pi 3 (RPi3)

As running a graphical user interface (GUI) is consuming processor power, the measurements have been done both with the GUI enabled and disabled.

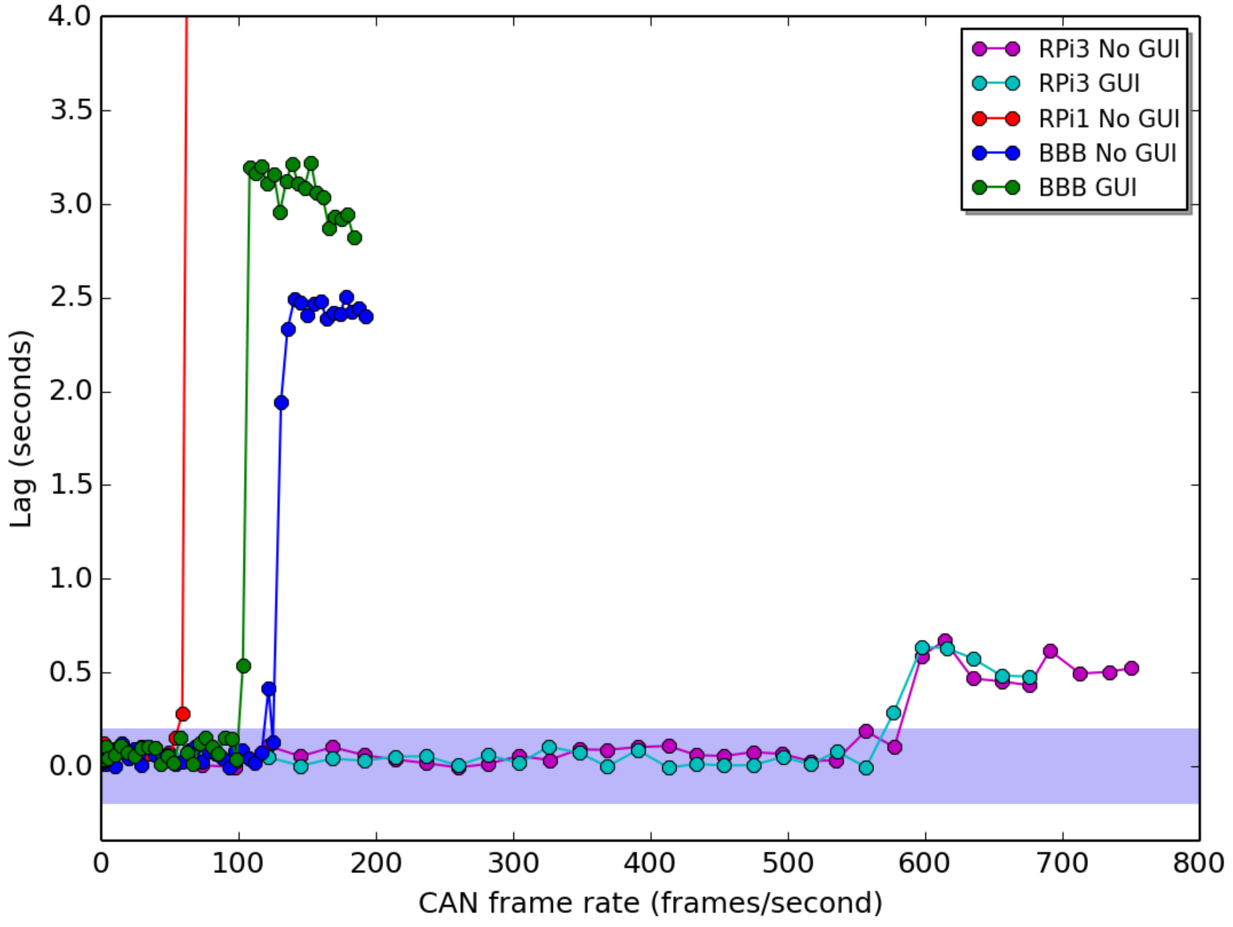


Fig. 11.3: Lag versus CAN frame rate for different embedded Linux boards.

It is seen in the lag and loss graphs that when it starts to be a lag in the processsing of the CAN frames, there is also a message loss.

The Beaglebone Black is faster than the Raspberry Pi1, but the fastest board is the Raspberry Pi3. For Beaglebone Black, which is having a single core, running the GUI reduces the processing power available for the Canadapter.

From the CPU usage graph it is clear that the lag appears when there is no more processing power available (the CPU idle drops to zero for single core machines).

For Raspberry 3, which has four cores, the lag appears when the core running the Canadapter is fully utilized. This is seen more clearly in the graph for the Raspberry Pi 3 CPU usage. The curve for Canadapter CPU usage reached 100%

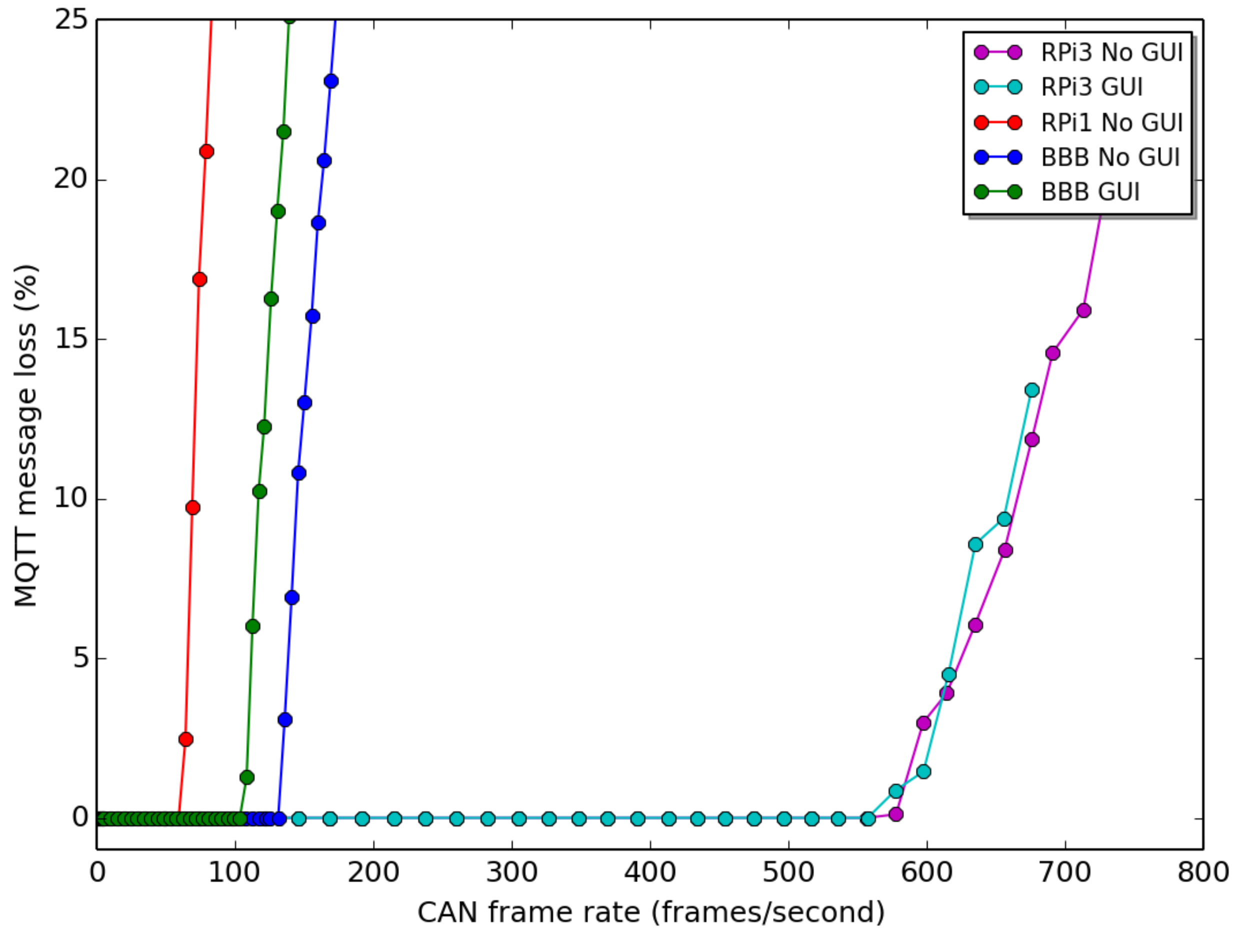


Fig. 11.4: Loss versus CAN frame rate for different embedded Linux boards.

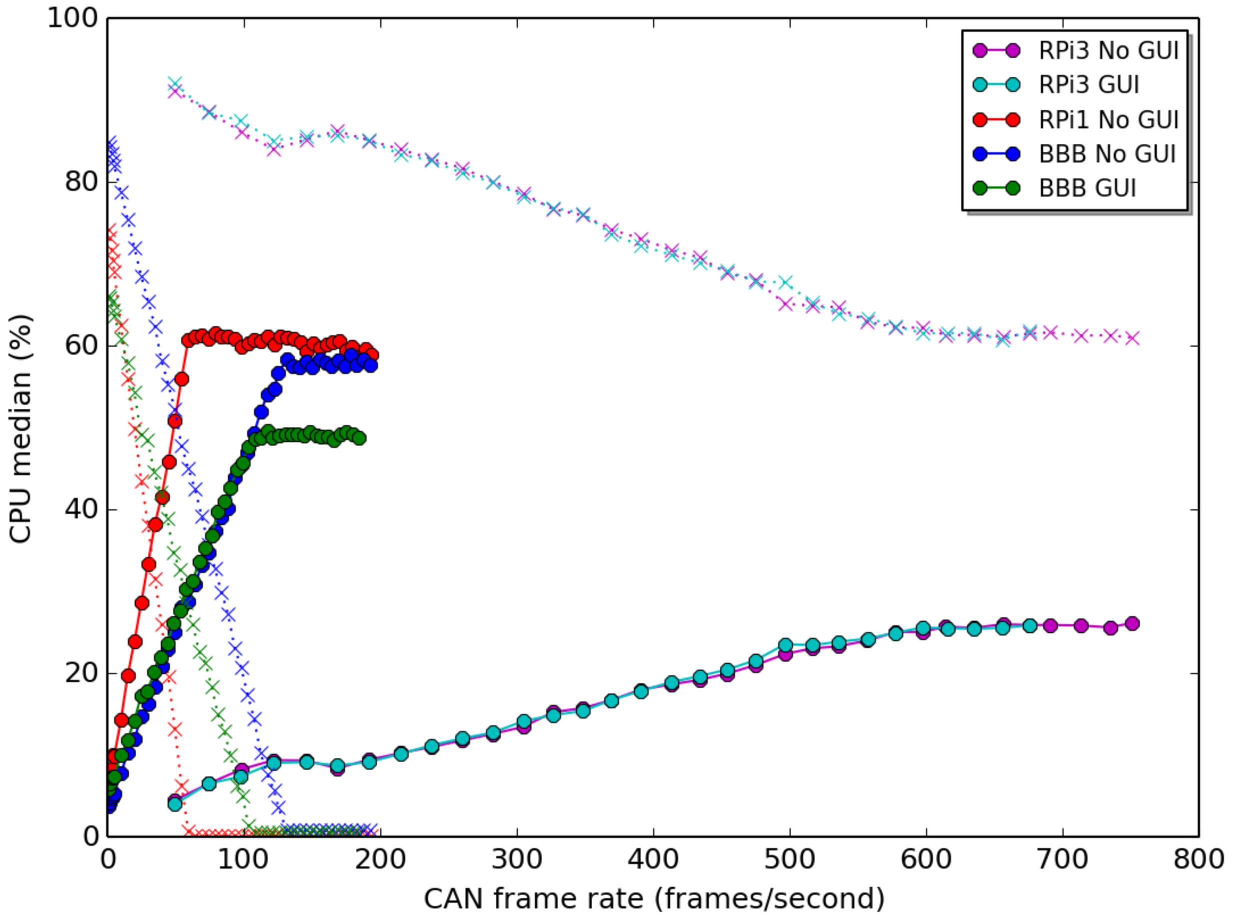


Fig. 11.5: CPU usage. The solid lines are 'total user CPU usage', and the dotted lines are 'CPU idle'.

(of a single core) at the CAN frame rate at which the lag appears. As there are four cores, the 'user's total CPU usage' reaches approximately 25% at this point.

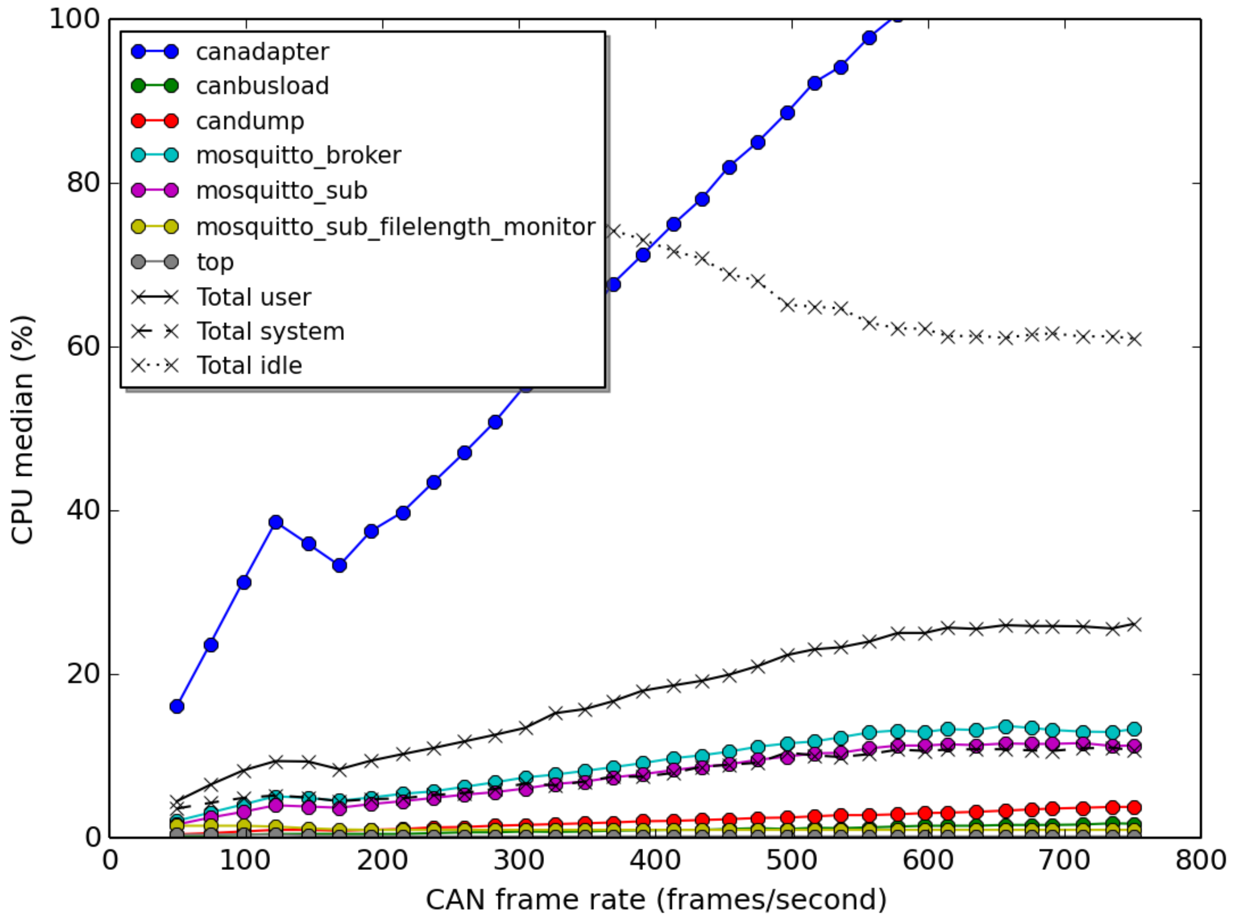


Fig. 11.6: CPU usage for Raspberry Pi 3.

11.5 Processor frequency

The effect of the CPU frequency on the Canadapter performace has been studied on Beaglebone Black. The result is:

- 1000 MHz: 130 frames/second
- 800 MHz: 110 frames/second
- 600 MHz: 90 frames/second
- 300 MHz: 50 frames/second

It is obvious that higher CPU frequency allows higher CAN frame unpacking rates.

The total memory usage is increasing between different runs. It is not clear whether this is related to the CPU frequency.

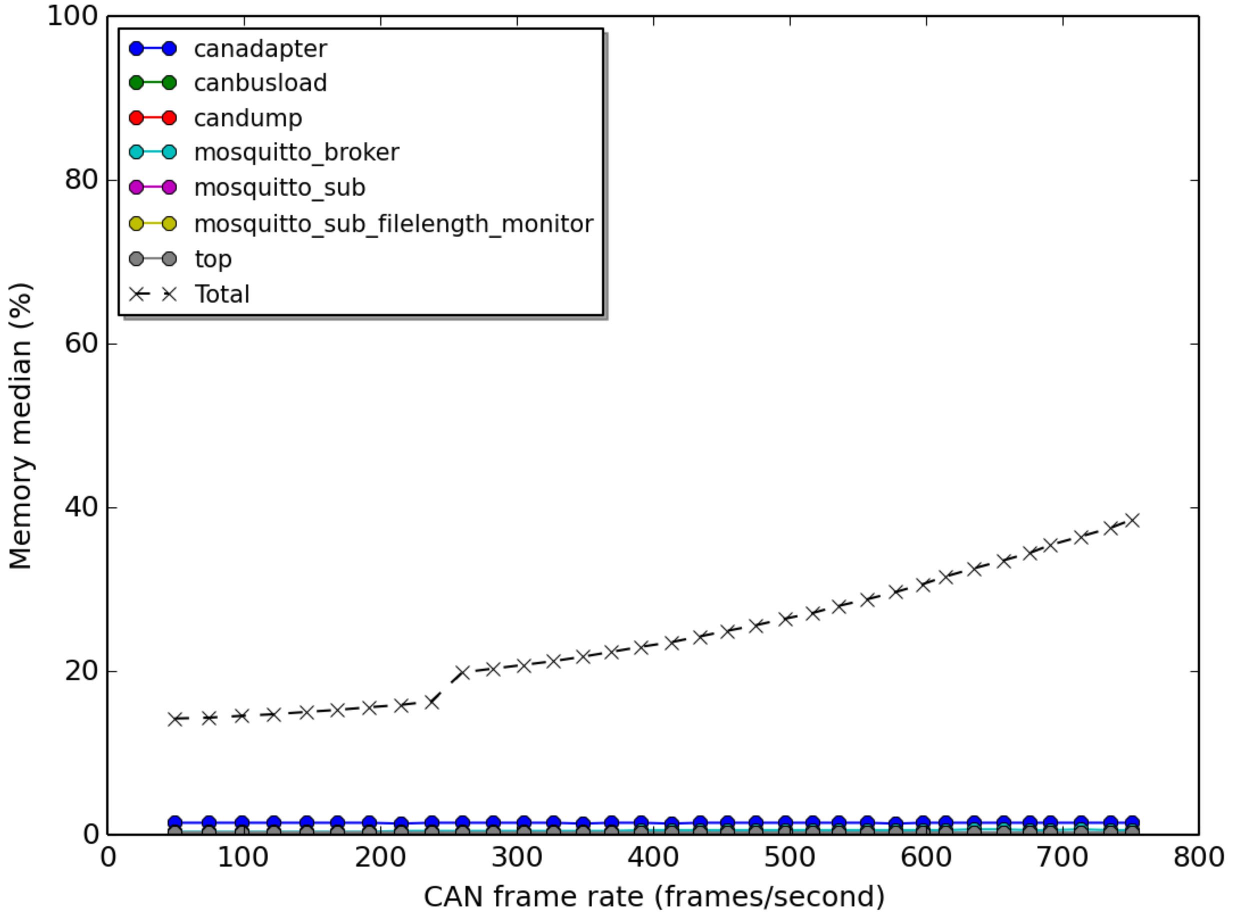


Fig. 11.7: Memory usage for Raspberry Pi 3.

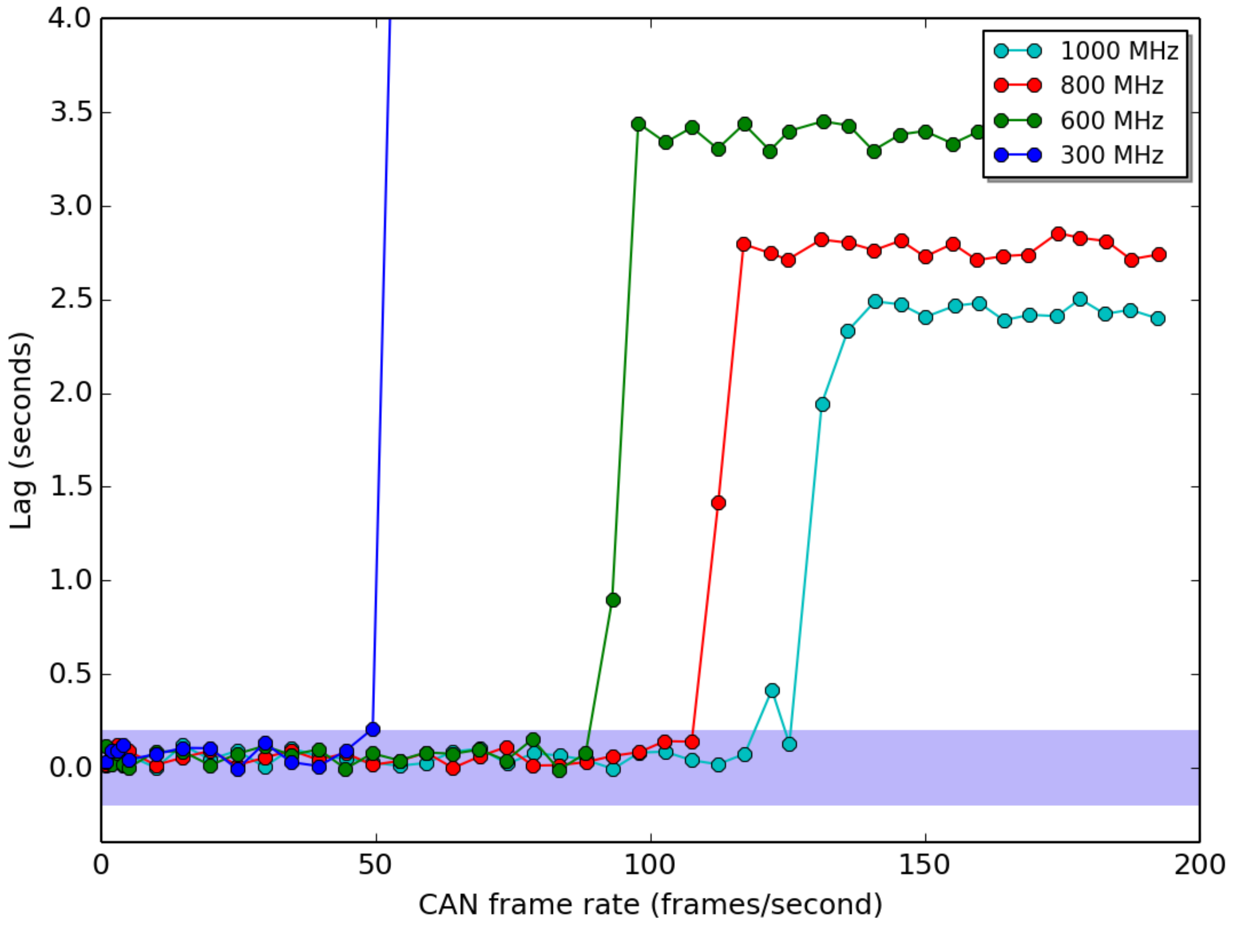


Fig. 11.8: Lag for different CPU frequencies on Beaglebone Black.

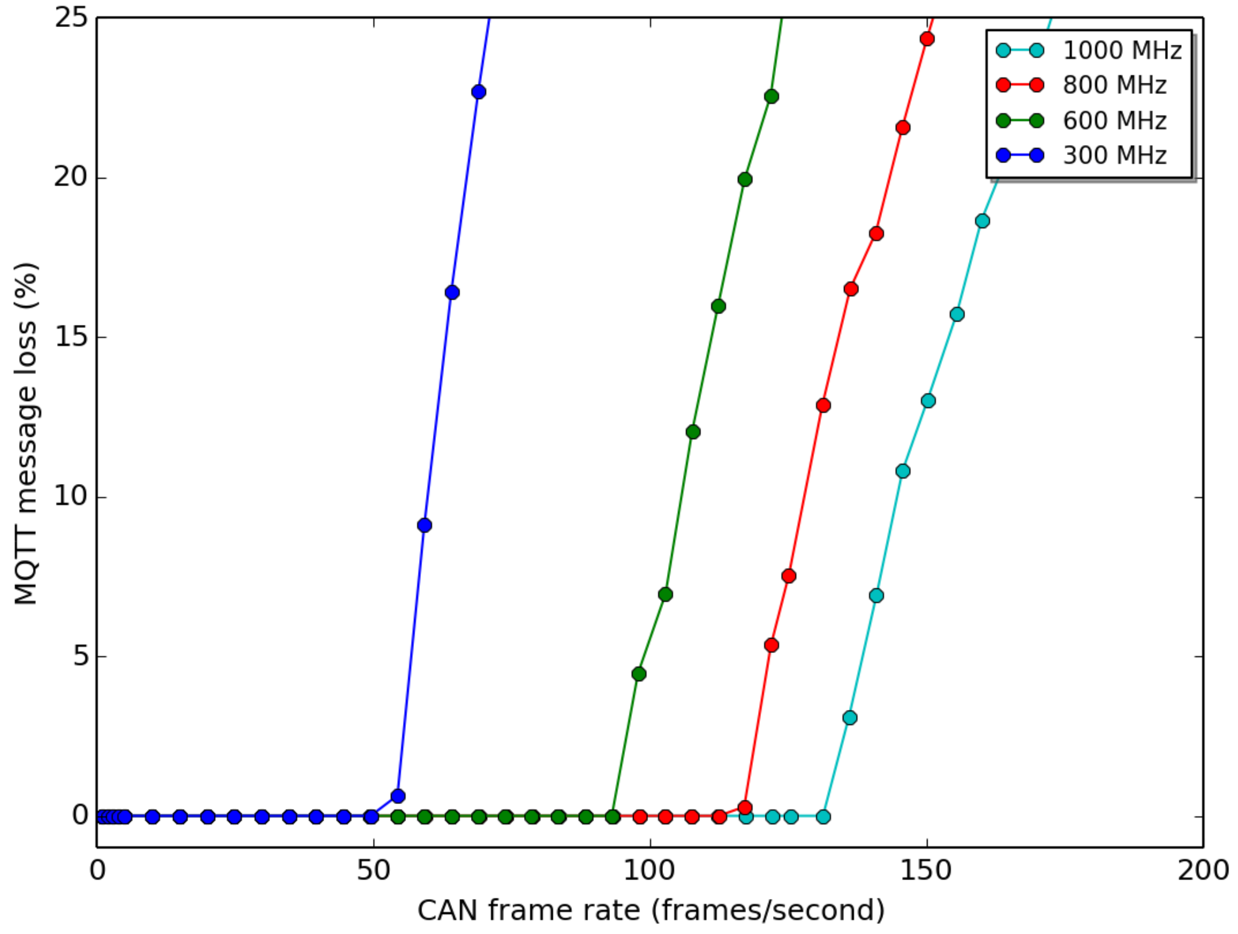


Fig. 11.9: Message loss for different CPU frequencies on Beaglebone Black.

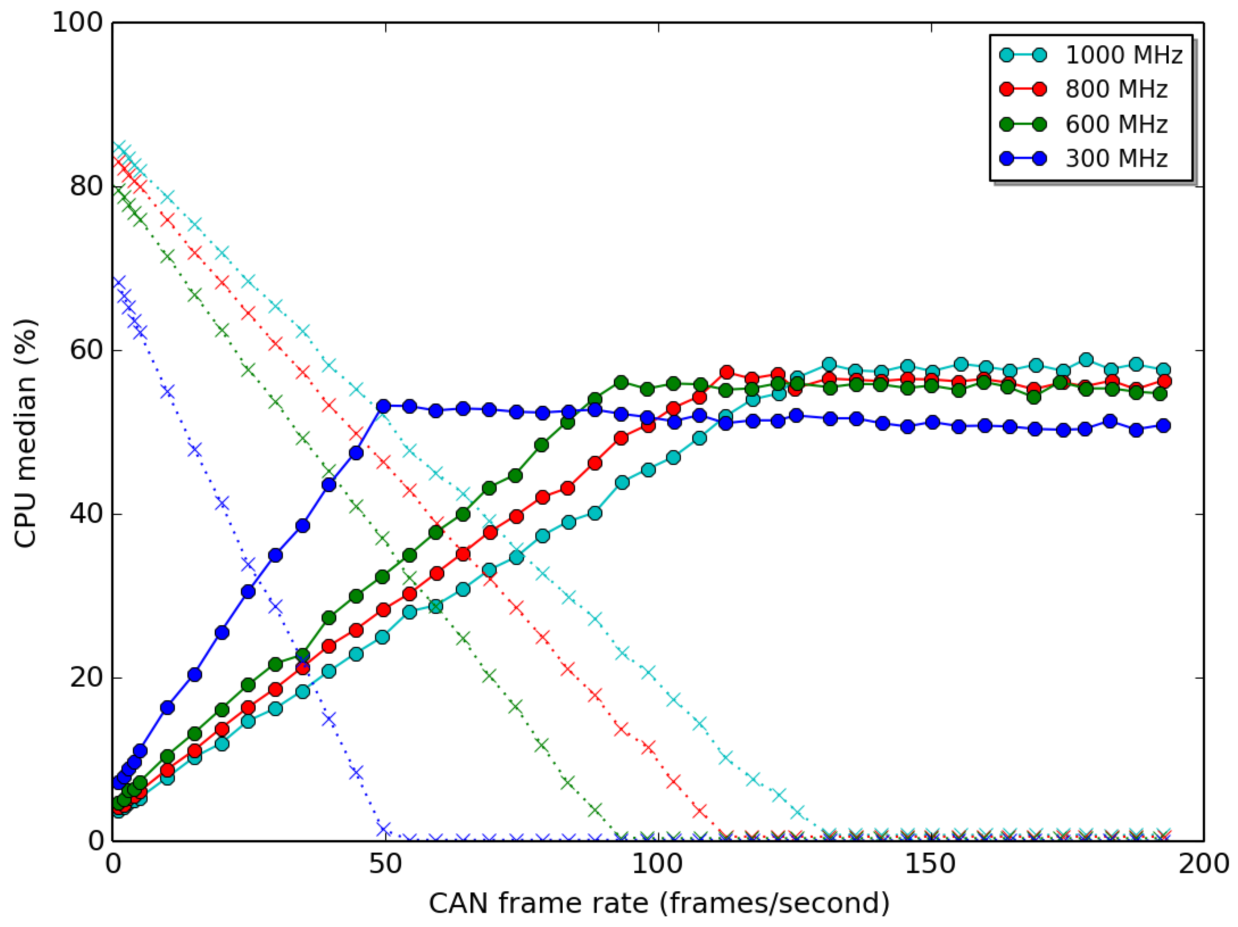


Fig. 11.10: Total user CPU usage (solid line) and CPU idle (dotted line) for different CPU frequencies on Beaglebone Black.

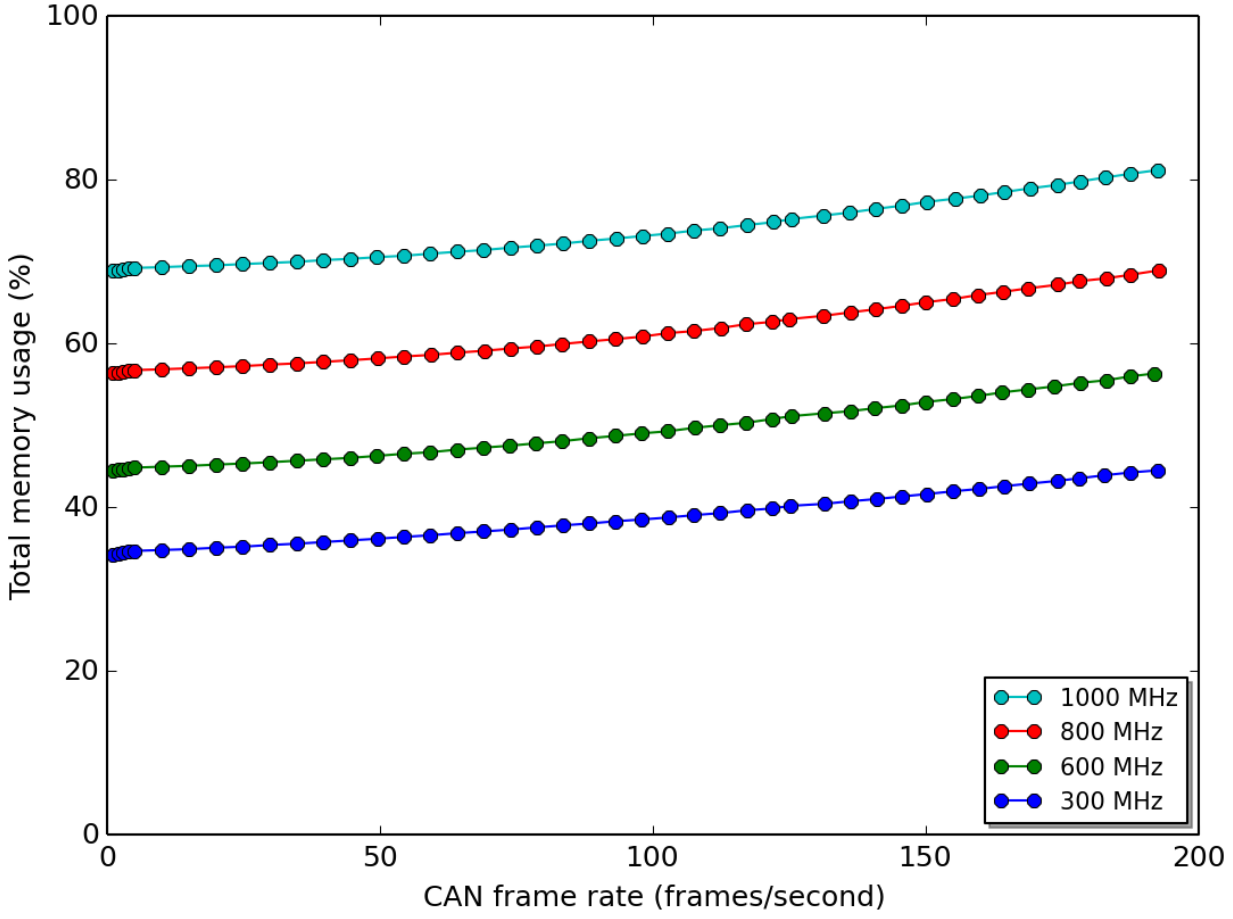


Fig. 11.11: Total memory usage for different CPU frequencies on Beaglebone Black.

11.6 MQTT Quality-of-Service (QoS)

A higher QoS level requires more processing power to handle the MQTT communication, and thus lowers the maximum CAN frame's unpacking rate. These are approximate numbers for Beaglebone Black:

- QoS 0: 130 frames/second
- QoS 1: 105 frames/second
- QoS 2: 110 frames/second

11.7 Throttling of incoming CAN frame rate

One way to avoid lag (delay) in the processing of incoming CAN frames is to use throttling. This means that, for example, a CAN frame with a period of 10 ms is downsampled to a period of 100 ms, in order to save processing power.

Measurements have been done on Beaglebone Black.

It is seen in the figures below that throttling is completely solving the lag problem, given that the period is long enough. Of course this leads to message loss.

The CAN frame rate downsampling is done in the Linux kernel, which saves on the processing power (as seen in the last graph).

11.8 Measurement data

Date	Time	Board	CPU	QoS	GUI	Throttling	Result
2016-08-26	071458	BBB	1000 MHz	2	No	No	110 Hz
2016-08-29	075740	BBB	300 MHz	0	No	No	50 Hz
2016-08-29	095732	BBB	600 MHz	0	No	No	90 Hz
2016-08-29	121348	BBB	800 MHz	0	No	No	110 Hz
2016-08-29	134655	BBB	1000 MHz	0	No	No	130 Hz
2016-08-30	113319	BBB	1000 MHz	1	No	No	105 Hz
2016-08-30	131155	BBB	1000 MHz	0	No	10 ms	100 Hz
2016-08-31	052316	BBB	1000 MHz	0	No	20 ms	50 Hz
2016-08-31	070806	BBB	1000 MHz	0	No	5 ms	130 Hz
2016-08-31	085031	BBB	1000 MHz	0	No	50 ms	20 Hz
2016-08-31	103256	BBB	1000 MHz	0	No	100 ms	10 Hz
2016-08-31	134958	BBB	1000 MHz	0	Yes	No	100 Hz
2016-09-05	144112	RPi1	Default	0	No	No	60 Hz
2016-09-12	070555	RPi3	Default	0	Yes	No	560 Hz
2016-09-13	060714	RPi3	Default	0	No	No	580 Hz

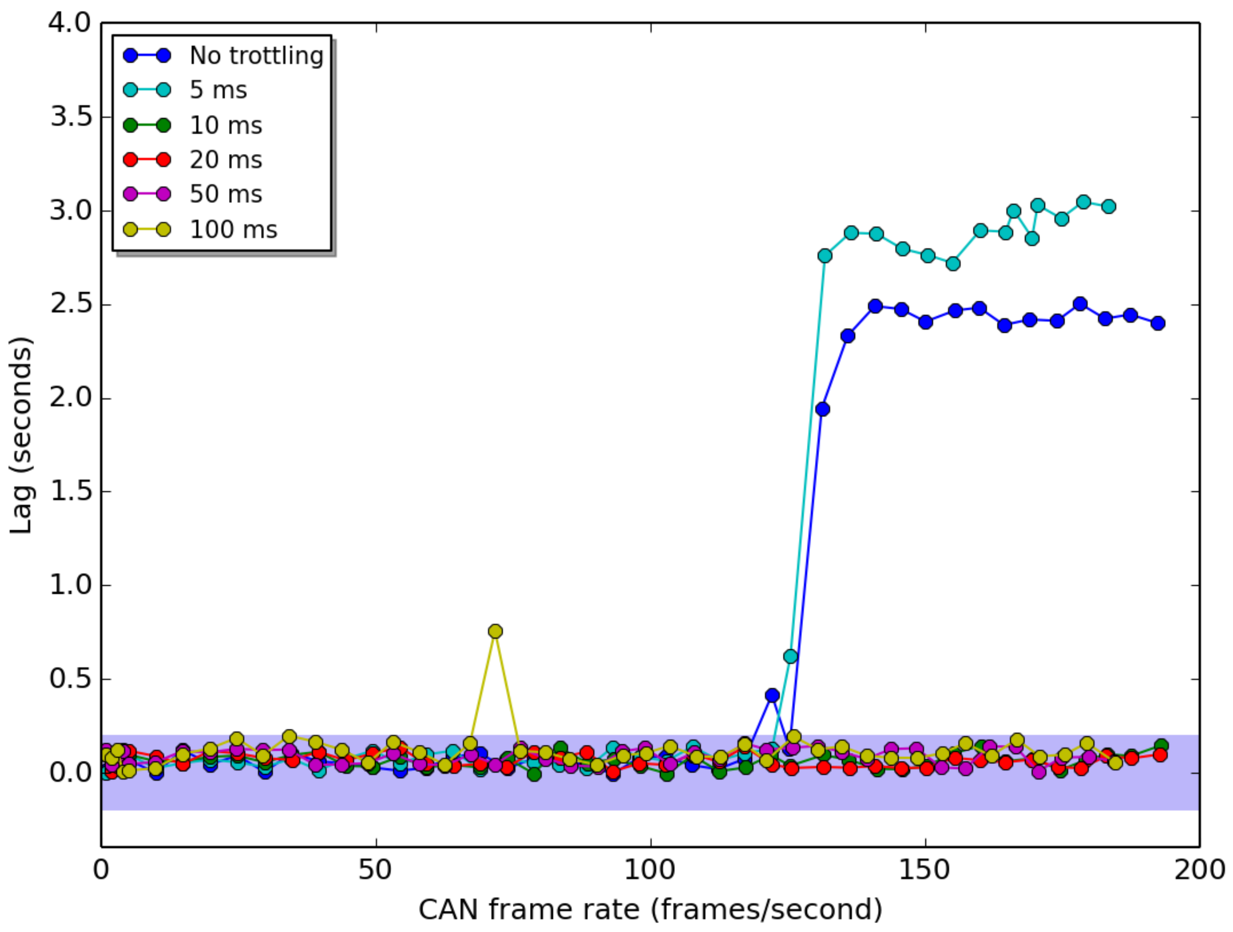


Fig. 11.12: Lag for different throttle settings on Beaglebone Black.

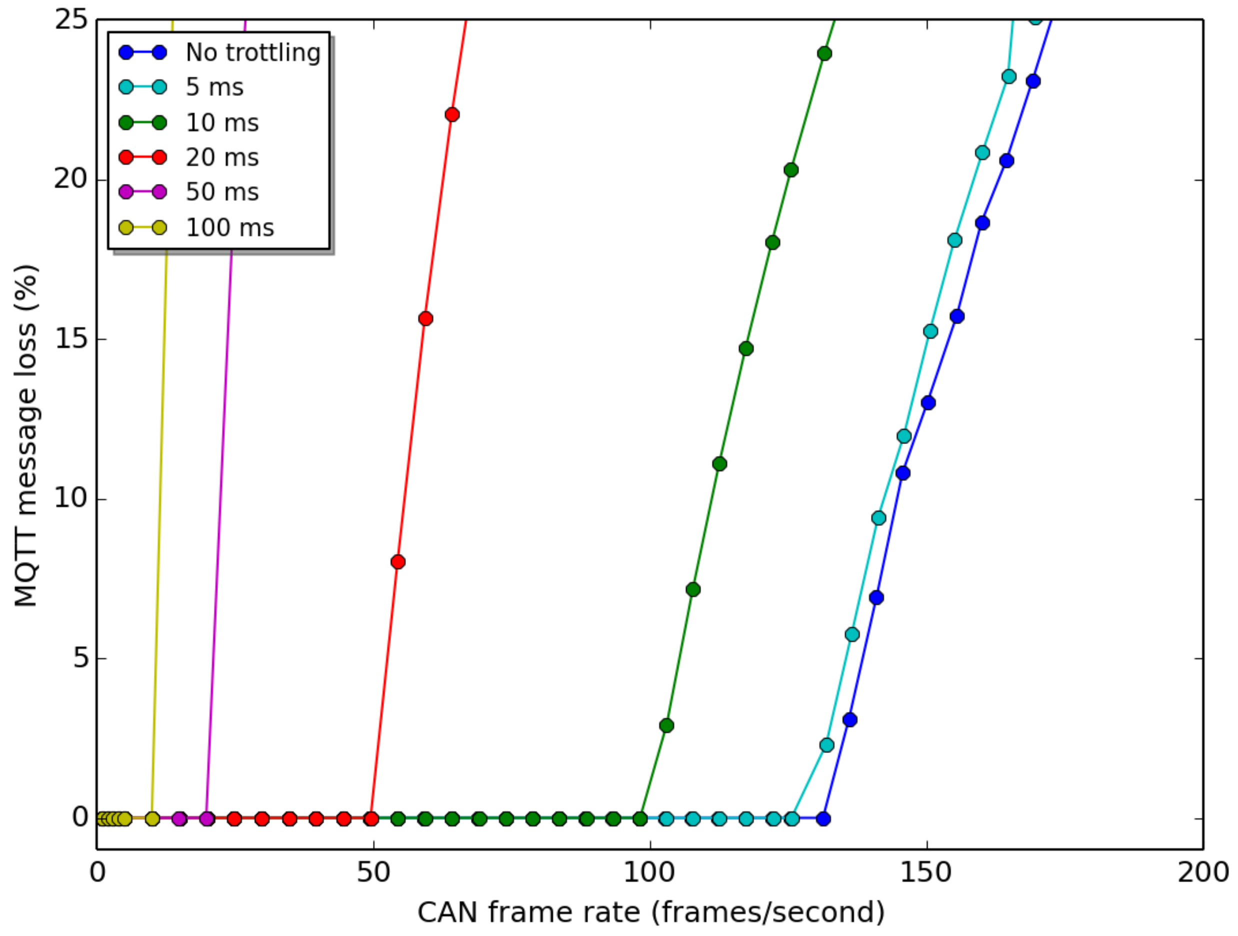


Fig. 11.13: Message loss for different throttle settings on Beaglebone Black.

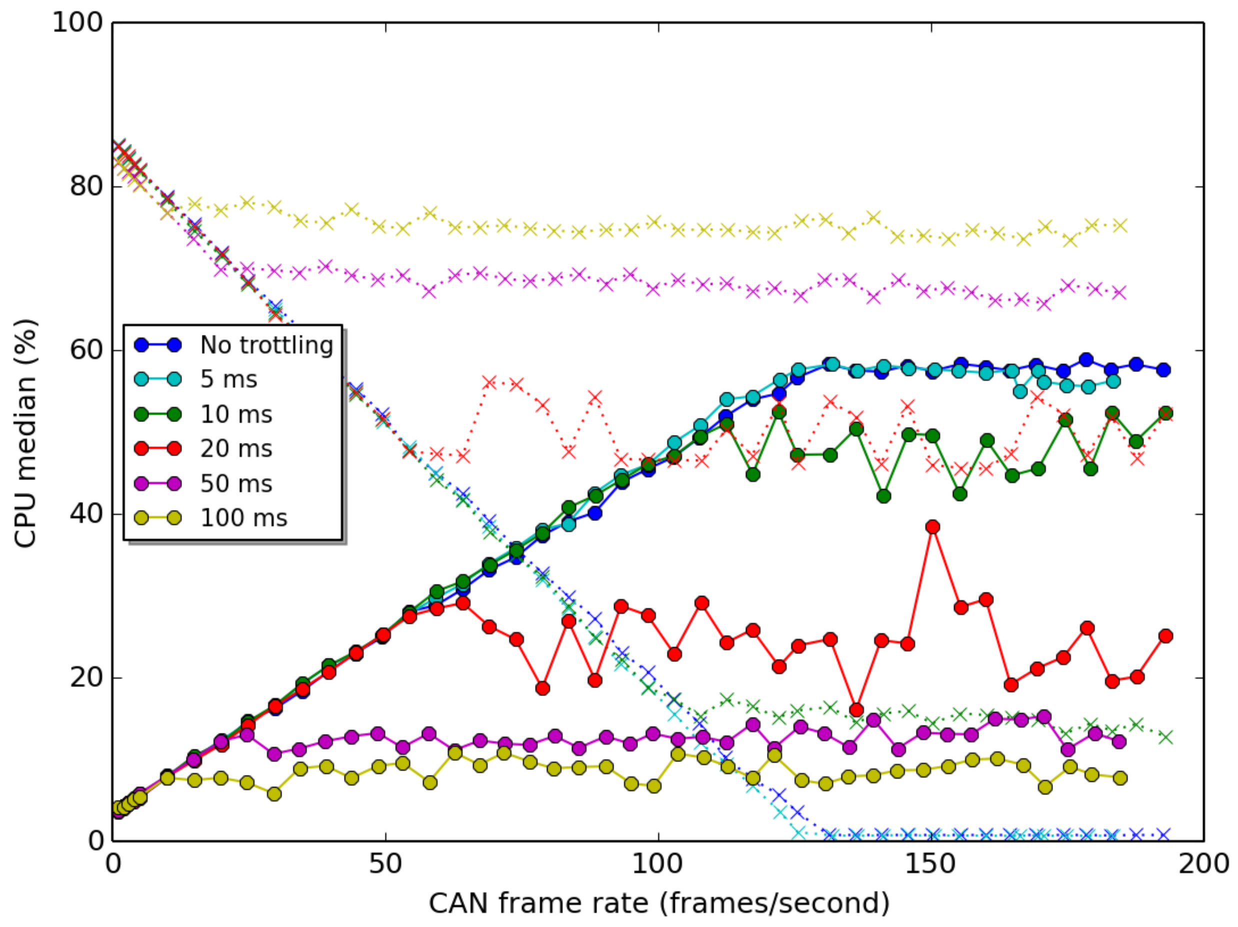


Fig. 11.14: Total user CPU usage (solid line) and CPU idle (dotted line) for different throttle settings on Beaglebone Black.

CHAPTER 12

Canadaper performance measurements

12.1 Introduction

In order to measure the performance of the canadapter, we send CAN frames to it and measure the output of MQTT messages.

Before running the measurements, some preparation needs to be done.

- Use a separate embedded Linux machine (A) sending generated CAN messages, and an embedded Linux machine (B) running the canadapter.
- The CAN bus between them should be set to 500 kbit/s.

The basic theory of the test is to:

- Send CAN frames with incrementing data values using cangen from a Linux machine (A).
- Measure CPU load and memory usage on the machine (B) running the canadapter.
- Save raw incoming CAN frames to file with timestamps using shell commands on machine B.
- Save MQTT messages to file with timestamps (also using shell commands) on machine B.

The reason shell commands are used to save measurement data to file is to minimize unnecessary CPU load and to have a reliable environment to extract data from.

12.2 Hardware setup example

One hardware setup could be:

CAN sender machine (machine A) consists of a Raspberry PI running Raspbian and including a hardware CAN interface. The CAN interface is set to send at 500 kbit/s. This hardware supports a maximum CAN busload of 45%

Candapter machine (B) is running the canadapter and the MQTT broker. It consists of a Beaglebone Black running Debian and is fitted with a CAN-cape (expansion board).

Make sure that your embedded Linux machine is running at full speed. It can be viewed using this command:

```
$ cpufreq-info
```

The result is for example:

```
cpufrequtils 008: cpufreq-info (C) Dominik Brodowski 2004-2009
Report errors and bugs to cpufreq@vger.kernel.org, please.
analyzing CPU 0:
  driver: cpufreq-voltdm
  CPUs which run at the same hardware frequency: 0
  CPUs which need to have their frequency coordinated by software: 0
  maximum transition latency: 300 us.
  hardware limits: 275 MHz - 720 MHz
  available frequency steps: 275 MHz, 500 MHz, 600 MHz, 720 MHz
  available cpufreq governors: conservative, ondemand, userspace, powersave, performance
  current policy: frequency should be within 275 MHz and 720 MHz.
                  The governor "ondemand" may decide which speed to use
                  within this range.
  current CPU frequency is 275 MHz.
  cpufreq stats: 275 MHz:0.53%, 500 MHz:0.01%, 600 MHz:0.04%, 720 MHz:99.41%  (88)
```

To change the CPU frequency:

```
$ sudo cpufreq-set -f 720MHz
```

12.3 Collecting measurement data

12.3.1 Generating CAN frames

On your CAN sender machine (A), set interface can0 to 500 kbit/s:

```
$ sudo ip link set can0 type can bitrate 500000
```

Using cangen to send frames (observe the g flag that is used to set the gap):

```
$ cangen can0 -g 50 -I 008 -L 8 -D i -n 1200
```

The frame rate is defined by the gap/sleep between messages in milliseconds. It is given by the `-g` flag. Invert the number to have the number of messages per second (f), so `f = 1/g` For example `g=10ms` gives approximately 100 messages per second. The `-I` flag sets the frame ID, the `-L` flag sets the number of bytes in each frame and the `-D` flag sets the frame to have incrementing values.

The total number of CAN frames sent is set by the `-n` flag. Adjust it so that the test is running for, for example, 100 seconds, which is the nominal duration time.

Avoid using the `-v` flag to print the sent frames, as it might slow down the process.

12.3.2 Receiving CAN frames to file

Setup interface can0 to 500 kbit/s on machine B:

```
$ sudo ip link set can0 type can bitrate 500000
```

Save incoming CAN frames to file with timestamps (`-ta` flag):

```
$ candump can0 -ta -l
```

See the following example excerpt from a candump log file:

```
(1461075290.584362) can0 008#A404000000000000
(1461075290.634635) can0 008#A504000000000000
(1461075290.684860) can0 008#A604000000000000
(1461075290.735140) can0 008#A704000000000000
(1461075290.785364) can0 008#A804000000000000
(1461075290.835666) can0 008#A904000000000000
(1461075290.885870) can0 008#AA04000000000000
(1461075290.936019) can0 008#AB04000000000000
(1461075290.986196) can0 008#AC04000000000000
(1461075291.036391) can0 008#AD04000000000000
(1461075291.086552) can0 008#AE04000000000000
```

12.3.3 Measuring CAN bus load

To measure the CAN bus load and save the data to file use:

```
$ canbusload can0@500000 -t > canbusload_data.log
```

See the following example excerpt from a canbusload log file:

```
canbusload 2016-04-19 14:16:07 (worst case bitstuffing)
 can0@500000  1595  215325 102080  43%

canbusload 2016-04-19 14:16:08 (worst case bitstuffing)
 can0@500000  1594  215190 102016  43%

canbusload 2016-04-19 14:16:09 (worst case bitstuffing)
 can0@500000  1582  213570 101248  42%
```

12.3.4 Starting the broker

The broker usually runs automatically as a service in the background. However, if it’s not running it can be started as a process by typing “mosquitto” in the terminal like so:

```
$ mosquitto
```

To run stop, start or restart mosquitto as a service the following command can be used:

```
$ sudo service mosquitto stop/start/restart
```

12.3.5 Run canadapter to convert CAN frames to MQTT messages

The canadapter needs to be started with parameters. These parameters are the path to the KCD file that interprets the CAN signals, path to the the MQTT json file that has the CAN signal names and their corresponding MQTT names. The other parameters that are good to use (but not obligatory but highly recommended) are the MQTT name and can interface.

Use the configuration files included in the project source. Run this command from the project root directory:

```
$ python3 scripts/canadapter.py \
examples/configfilesForCanadapter/climateservice_cansignals.kcd \
-mqttfile examples/configfilesForCanadapter/climateservice_mqttsignals.json \
-mqttname climateservice -i can0
```

Note that there should be no space after the backslashes. If getting problems, remove the backslashes and put the command on a single long line.

12.3.6 Receiving MQTT messages to file

To subscribe to a mosquitto topic and save the data to a file the following command can be used:

```
$ mosquitto_sub -t "data/climateservice/#" -v -R > mosquitto_sub_data.log
```

To subscribe to all topics and add timestamps to the data, this command can be used:

```
$ mosquitto_sub -v -R -t "+/#" | while IFS= read -r line; do printf '(%s) %s\n' "$(date '+%s.%N')" "$
```

Note that at high data rates, the timestamping introduces a significant delay.

See the following example excerpt from a log file to see how the result is presented:

```
(1457617715.839584510) data/climateservice/enginespeed 0.0
(1457617715.863914727) data/climateservice/vehiclespeed 0.0
(1457617715.884274556) data/climateservice/enginespeed 0.0
(1457617715.915510919) data/climateservice/vehiclespeed 2.56
(1457617715.941249622) data/climateservice/enginespeed 0.0
(1457617715.964192934) data/climateservice/vehiclespeed 5.12
(1457617715.988655770) data/climateservice/enginespeed 0.0
(1457617716.015374476) data/climateservice/vehiclespeed 7.68
(1457617716.033834838) data/climateservice/enginespeed 0.0
(1457617716.063329853) data/climateservice/vehiclespeed 10.24
(1457617716.087545615) data/climateservice/enginespeed 0.0
(1457617716.110456886) data/climateservice/vehiclespeed 12.8
(1457617716.134399576) data/climateservice/enginespeed 0.0
(1457617716.159300270) data/climateservice/vehiclespeed 15.36
```

It is better to write the MQTT messages to file, and write the number of recived messages (for each second) to another file:

```
while true; do echo "(`date +%s.%N`)  `wc -l mosquitto_sub_data.log`" \
>> mqtt_log_length.txt; sleep 1; done
```

See the following example excerpt from a filelength measurement file (showing number of total messages so far):

```
(1461076433.847400400)  16442 mosquitto_sub_data.log
(1461076434.901527780)  16442 mosquitto_sub_data.log
(1461076435.978809565)  16555 mosquitto_sub_data.log
(1461076437.134160771)  16789 mosquitto_sub_data.log
(1461076438.292547163)  17023 mosquitto_sub_data.log
(1461076439.446994608)  17257 mosquitto_sub_data.log
```

12.3.7 Measuring processor load on the machine running canadapter

Use the program `top` to monitor the CPU load and memory usage with this command:

```
$ top -cb -n 3 -d 3 > top_data.log
```

where the `-n` flag is number of samples and `-d` is the delay between each sample.

See the following example excerpt from a top log file:

```
top - 14:36:13 up 5 days, 23:16,  4 users,  load average: 1.29, 0.69, 0.35
Tasks: 130 total,   2 running, 125 sleeping,   3 stopped,   0 zombie
%Cpu(s):  0.6 us,  0.7 sy,  0.0 ni, 98.6 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem:    244088 total,   237832 used,     6256 free,    11960 buffers
KiB Swap:        0 total,        0 used,        0 free.   106688 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
16541 debian    20   0   21404  10312   5112 R 37.0  4.2   1:29.27 python3 scripts/canadapter.py exar
16703 debian    20   0    2988   1604   1300 R 15.8  0.7   0:00.15 top -cb -n 3 -d 3
  521 root     -51   0       0      0      0 S 10.6  0.0   3:28.04 [irq/199-can0]
  930 debian    20   0    4400   2504   1692 S  7.9  1.0  17:08.93 mosquitto
16525 debian    20   0    9224   3256   2592 S  5.3  1.3   0:05.71 sshd: debian@pts/3
   63 root     -51   0       0      0      0 S  2.6  0.0   3:15.87 [irq/176-4a10000]
14924 www-data  20   0  228736   2796   1432 S  2.6  1.1   0:12.28 /usr/sbin/apache2 -k start
16696 root      20   0       0      0      0 S  2.6  0.0   0:00.58 [kworker/0:2]
16698 debian    20   0    3432   2436   2120 S  2.6  1.0   0:02.98 mosquitto_sub -v -t +/#
    1 root      20   0   22012   3292   2048 S  0.0  1.3   0:56.49 /sbin/init
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.02 [kthreadd]
    3 root      -2   0       0      0      0 S  0.0  0.0  51:10.41 [ksoftirqd/0]
    6 root      20   0       0      0      0 S  0.0  0.0   0:04.12 [kworker/u2:0]
```

12.3.8 Run the actual measurement

Typically you would like to run the measurements for different CAN frame rates and number of sent frames.

Use the CAN frame ID=8, and the KCD file included in the project source.

1. On machine B start (in this order):

- Mosquitto broker
- canadapter
- candump (to file)
- mosquitto_sub (to file)

2. Start cangen with appropriate CAN framerate and number of frames on machine A.
3. Start top (to file) and canbusload (to file) on machine B.
4. Keep the logs running for an additional 10s to give the system time to work through the buffer.

12.4 Analyzing the measurement data

12.4.1 Calculating lost messages

First, make sure that all CAN frames sent from machine A are received in machine B. This is done by verifying that the number of CAN frames (lines) in the candump_data.log file equals the number of frames sent by cangen.

To calculate the number of lines in a file, use:

```
$ wc -l filename
```

Depending on the contents of the KCD file, each incoming CAN frame might be converted into several MQTT messages:

```
Expected number of MQTT messages = (candump_data.log lines)*(MQTT messages per CAN frame)
```

Calculate the number of lost MQTT messages:

```
Number of lost MQTT messages = (expected number of MQTT messages) - (mosquitto_sub_data.log lines)
```

Calculate the fraction of lost MQTT messages:

```
Message loss ratio = (number of lost MQTT messages) / (expected number of MQTT messages)
```

12.4.2 Calculating lag for the canadapter

We define the lag for the canadapter as the time it takes for an arriving CAN frame to be converted to MQTT (and received as a MQTT message). Note that this included any delay for example in the MQTT broker.

By comparing the timestamps for the first message in each of the candump_data.log and mosquitto_sub_data.log files, we calculate the start lag.

If there are no lost messages, we also calculate the end lag by comparing the last timestamps in the two files.

Additionaly calculate the incoming CAN frame rate by dividing the number of CAN frames with the duration time (measured as the difference between the first and the last time stamp).

Also calculate the MQTT duration time by comparing the first and last timestamps in the mosquitto_sub_data.log file.

Compare the CAN and MQTT duration times to the nominal duration time.

Compare the CAN frame rate to the given cangen delay flag (-g) value.

Compare the calculated CAN frame rate to the canbusload measurement data.

12.4.3 Calculating the processor load for the machine running the canadapter

From the top_data.log file calculate the minimum and maximum processor load and memory consumption for the relevant processes. Those include:

- candump
- Mosquitto broker
- canadapter
- canbusload
- mosquitto_sub
- top

Also extract the total processor load (minimum and maximum) from the log file. Compare this to the sum of the relevant processes.

12.5 Plotting graphs

Use your favourite plotting tool to plot the measuring data.

12.5.1 Processor load

Processor load is measured in `%` of the total capacity of the CPU. It's plotted as a function of the incoming CAN frame rate `(frames/sec)`. Plot a line for each of the relevant processes.

12.5.2 Lag

Lag is measured in `seconds` and is plotted as a function of the incoming frame rate `(frames/sec)`.

12.5.3 Lost messages

Lost messages are plotted as `data loss (%)` as function of the incoming frame rate `(frames/sec)`.

12.6 Analyzing the resulting graphs

Study the resulting graphs to find the maximum number of CAN frames per second that could be handled.

12.7 Running measurements automatically

There are scripts that will run the measurements automatically, for different CAN frame rates.

Before running the automated measurements make sure that:

- The broker is running
- No unwanted processes are running, for example, other instances of 'canadapter'
- Adjust the CPU frequency accordingly

In the directory `sgframework/tests/automated_measurements` run:

```
$ python3 measurement_script.py
```

You will most likely need to change some of the settings in the `measurement_settings.py` and otherwise to have it running properly.

The measurement scripts will create a measurement directory (named by the start time) with subdirectories, each representing a datapoint (a specific CAN frame rate).

Analyze data with this command. You should point it to the measurement directory (having the subdirectories). It is creating a JSON file for each datapoint. In the directory `sgframework/tests/automated_measurements` (also on the embedded Linux machine) run:

```
$ python3 parse_canadapter_measurements.py _measurementdata/md-20160425-151125
```

In order to compare different measurement, put several measurement directories (each having JSON files) in a top directory. As the measurement directory names are used in the legend, they should have informative names. Plot graphs (preferably on a Linux desktop machine) using this in the directory `sgframework/tests/automated_measurements`:

```
$ python3 dataplotter.py top_directory_name
```

12.8 Measuring the speed of subsystems - Reception of CAN frames

In order to find any bottlenecks is the subsystems made by the canadapter, it can be useful to measure the speed of the subsystems separately.

Run the reception of incoming CAN frames with increasing incoming CAN frame rates, until there is loss of data. Calculate the data loss by counting lines in the resulting file.

This will give the maximum CAN frame rate that each subsystem can handle.

Send the CAN frames using 'cangen' as described above.

12.8.1 Reception of raw CAN frames using Python

Run the measurement using this command:

```
$ python3 speedmeasurement_rawcan_receive.py
```

An example of the resulting log file:

```
(1458653867.093)   CAN Id:    8 (Hex   8)    Data: 00 00 00 00 00 00 00 00
(1458653867.095)   CAN Id:    8 (Hex   8)    Data: 01 00 00 00 00 00 00 00
(1458653867.097)   CAN Id:    8 (Hex   8)    Data: 02 00 00 00 00 00 00 00
(1458653867.098)   CAN Id:    8 (Hex   8)    Data: 03 00 00 00 00 00 00 00
(1458653867.099)   CAN Id:    8 (Hex   8)    Data: 04 00 00 00 00 00 00 00
(1458653867.100)   CAN Id:    8 (Hex   8)    Data: 05 00 00 00 00 00 00 00
(1458653867.101)   CAN Id:    8 (Hex   8)    Data: 06 00 00 00 00 00 00 00
(1458653867.103)   CAN Id:    8 (Hex   8)    Data: 07 00 00 00 00 00 00 00
(1458653867.106)   CAN Id:    8 (Hex   8)    Data: 08 00 00 00 00 00 00 00
(1458653867.107)   CAN Id:    8 (Hex   8)    Data: 09 00 00 00 00 00 00 00
```

12.8.2 can4python, which is used by the canadapter

Run the measurement using this command:

```
$ python3 can4python_receiver.py
```

Each line in the log file contains the extracted CAN signals from one CAN frame. An example of the resulting log file:

```
(1458654459.307)  Data: dict_items([('vehiclespeed', 0.0), ('enginespeed', 0.0)])
(1458654459.308)  Data: dict_items([('vehiclespeed', 2.56), ('enginespeed', 0.0)])
(1458654459.309)  Data: dict_items([('vehiclespeed', 5.12), ('enginespeed', 0.0)])
(148654459.310)   Data: dict_items([('vehiclespeed', 7.68), ('enginespeed', 0.0)])
(1458654459.311)  Data: dict_items([('vehiclespeed', 10.24), ('enginespeed', 0.0)])
(1458654459.313)  Data: dict_items([('vehiclespeed', 12.8), ('enginespeed', 0.0)])
(1458654459.314)  Data: dict_items([('vehiclespeed', 15.36), ('enginespeed', 0.0)])
(1458654459.315)  Data: dict_items([('vehiclespeed', 17.92), ('enginespeed', 0.0)])
(1458654459.316)  Data: dict_items([('vehiclespeed', 20.48), ('enginespeed', 0.0)])
(1458654459.316)  Data: dict_items([('vehiclespeed', 23.04), ('enginespeed', 0.0)])
```

12.9 Measuring the speed of subsystems - Sending MQTT messages

The idea is to send a number of MQTT messages as fast as possible and to measure data loss and reception time. Calculate the effective MQTT message rate.

Receive the MQTT messages as described in an earlier section using mosquitto_sub.

Repeat the mesurements with increasing number of MQTT messages sent, until there is data loss in the broker or in the receiving mosquitto_sub.

This will give the maximum MQTT number of messages we can send at max speed for each subsystem.

12.9.1 mosquitto_pub

Send the MQTT messages using:

```
$ time for i in `seq 1 1000`; do mosquitto_pub -t mosquitto/test -m testmessage$i; done;
```

This is a rather slow method, as the mosquitto_pub needs to connect a new client to the broker for each message sent.

Create a file with a large number of payloads:

```
for i in {1..1000}; do echo messagenumber$i ; done > payloads.txt
```

Send each line in the file as an individual MQTT message (all on same topic):

```
$ time cat payloads.txt | mosquitto_pub -t TODO/testroot/a/b/c -l -q 1
```

12.9.2 Paho, which is the Python MQTT library

Send the MQTT messages using the command:

```
$ python3 speedmeasurement_paho_send.py -n [NUMBER OF MESSAGES]
```

12.9.3 sgframework

Send the MQTT messages using:

```
$ python3 speedmeasurement_sgframework_send.py -n [NUMBER OF MESSAGES]
```

12.10 Using graphviz and pycallgraph to visualize line profiling

Install pycallgraph:

```
$ sudo apt-get install pygraphcall
```

Install graphviz (dot):

```
$ sudo apt-get install graphviz
```

To collect data from your python file and plot it to a picture use this command:

```
$ pycallgraph graphviz -f svg -o ./output_picture_name_here.svg -- your_script_here.py
```

SVG is used because an error occurs with bitmapping sometimes which leads to an extremely small picture that is not readable. SVG is based on vector graphics and can be resized as you please.

12.11 Converting can4python python modules to cython modules

- Make a backup of the can4python lib

Create a file called setup_c.py in the installed can4python lib (/usr/local/lib/python3.4/dist-packages/can4python) it should contain:

```
$ from distutils.core import setup
 from Cython.Build import cythonize

 setup(
   ext_modules = cythonize(["*.pyx"]),
 )
```

Change canbus.py and caninterface_raw.py to the file extension .pyx

Run “python3 setup_c.py build_ext –inplace”

12.12 Suggested improvements

The sgframework and the canadapter demonstrate an architecture concept, and is also useful for prototyping needs.

The usecase is to extract a small number of CAN signals from a CAN bus for applications without any real-time requirements.

A production implementation should be optimized for speed, for example, by implementing the software in the C programming language. Note, as the MQTT publish-subscribe protocol (running on TCP/IP) is used, it is not intended for usecases with hard real-time requirements.

Dependencies

13.1 Core dependencies for sgframework module

Dependency	Description	License	Debian/pip package
Python 3.3+	Python	PSFL	
Mosquitto 1.4.1+	MQTT broker	BSD	D: mosquitto
Paho Python 1.0.2+	MQTT client library	EPL 1.0 and EDL 1.0	P: paho-mqtt

13.2 Dependencies for the scripts and examples

Not all dependencies are required for all examples.

Dependency	Description	License	Debian/pip package
CAN interface	Should support SocketCAN	Part of Linux kernel	
vcan0	Virtual CAN bus interface	Part of Linux kernel	
can4python 0.1+	CAN bus library	BSD 3-clause	P: can4python
mosquitto-clients	MQTT command line tools	BSD 3-clause	D: mosquitto-clients
can-utils	CAN bus command line tools	GPL or BSD 3-clause	D: can-utils
TK	Graphics library	Tcl/Tk license (BSD)	D: python3-tk

13.3 Dependencies for testing

Dependency	Description	License	Debian/pip package
coverage	Test coverage measurement	Apache 2.0	P: coverage
paramiko	Remote control via SSH	LGPL	P: paramiko
libssl	For paramiko	Apache 1.0	D: libssl-dev
libffi	For paramiko	MIT	D: libffi-dev

13.4 Documentation dependencies

Dependency	Description	License	Debian/pip package
texlive	Latex library (for PDF creation)	“Knuth”	D: texlive-full
Matplotlib	Plotting library	PSFL based	D: python3-matplotlib
Sphinx 1.3+	Documentation tool	BSD 2-cl	P: sphinx
programoutput	Spinx add-on for program output	BSD 2-cl	P: sphinxcontrib-programoutput
Sphinx rtd theme	Theme for Sphinx	MIT	P: sphinx_rtd_theme

13.5 Installation commands for the dependencies

Core, and tools for usage on embedded Linux machines:

```
$ sudo apt-get install can-utils
$ sudo apt-get install mosquitto
$ sudo apt-get install mosquitto-clients
$ sudo apt-get install python3-pip
$ sudo pip3 install paho-mqtt
$ sudo pip3 install can4python
```

Desktop examples, documentation and testing:

```
$ sudo apt-get install python3-tk
$ sudo apt-get install python3-matplotlib
$ sudo apt-get install build-essential libssl-dev libffi-dev python-dev  # For Paramiko
$ sudo pip3 install sphinx
$ sudo pip3 install sphinxcontrib-programoutput
$ sudo pip3 install sphinx_rtd_theme
$ sudo pip3 install coverage
$ sudo pip3 install paramiko
```

For PDF, you also need to install (3 GByte):

```
$ sudo apt-get install texlive-full
```

Temporary requirements (before publishing on github):

```
$ sudo apt-get install subversion
```

CHAPTER 14

Details on installing dependencies and tools

14.1 CAN hardware on BeagleBone

BeagleBone ("white") and BeagleBone Black has built-in CAN-controllers. In order to use it you must add CAN transceivers, to adapt the electrical levels to the CAN bus.

There are expansion cards ("capes") with CAN transceivers, for example the "TT3201 CAN Cape" from Towertech. The cape also contains two additional CAN controllers.

The DCAN1 controller inside the BeagleBone processor is named CAN0 interface in Debian, and is connected to the pins 1 (GND), 3 (CAN_L) and 4 (CAN_H) on the cape.

The Debian distribution has support for the CAN interface.

It is possible to just add the CAN transceiver chip to your BeagleBone. Then use this pinout for DCAN1:

Signal	Pin number on P9
GND	1, 2, 43-46
VDD_3V3EXP (500 mA)	3, 4
SYS_5V	7, 8
dcan1_rx	24
dcan1_tx	26

14.2 Install Debian on BeagleBone, and get the CAN interface running

Have an empty SD-micro card ready.

Download the disk image appropriate for your version of the board from http://beagleboard.org/latest-images For this installation Debian 8.4 was the latest version.

Follow the instructions on https://beagleboard.org/getting-started to give your SD-card a fresh installation of the Debian OS.

It seems that the Beaglebone Black will boot from the SD card default. The documentation says that it boots from NAND by default and that you need to press and hold button S2 when powering on to boot from SD-card. It will stay in this boot mode until power down (so you can use the reset button without any problem).

On the download web page there is a description how to flash the image to NAND.

By default the Beaglebones IP through the USB connection is `192.168.7.2`.

If you are running Linux or OSX ssh to the Beaglebone:

```
$ ssh debian@192.168.7.2
```

The default password is `temppwd`. For the root account the password is `''` (none).

If you are running Windows which have no ssh client included in the default OS, use Putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html) or another SSH client (http://www.openssh.com/windows.html) to connect to your Beaglebone.

You might resize the partition on your SD card. Follow this guide: http://elinux.org/Beagleboard:Expanding_File_System_Partition_On_A_microSD

Make sure the Beaglebone is connected to the Internet.

At the command prompt:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install can-utils
```

The `upgrade` step will take more than one hour to finish.

If you need to change the host name, edit the files `/etc/hostname` and `/etc/hosts`. Replace 'beaglebone' with your new hostname.

To enable the CAN interface, add the line `cape_enable=bone_capemgr.enable_partno=BB-CAN1` to the end of the `/boot/uEnv.txt` file. Add it through nano:

```
$ sudo nano /boot/uEnv.txt
```

If you like to run your Beaglebone without a GUI, add also this line to the file: `cmdline=systemd.unit=multi-user.target`

Possibly this could be used instead from command line:

```
$ sudo systemctl set-default multi-user.target
```

If the GUI is running, this line will appear when running `ps -ef`:

/usr/sbin/lightdm

Reboot the Beaglebone and verify that the CAN interface hardware is enabled:

```
$ cat /sys/devices/platform/bone_capemgr/slots
```

and start the CAN interface with correct speed:

```
$ sudo ip link set can0 up type can bitrate 500000
```

List available network interfaces (including CAN):

```
$ ifconfig
```

If you need to disable the CAN interface:

```
$ sudo ip link set can0 down
```

14.3 CAN hardware on Raspberry Pi

In order to run CAN on a Raspberry Pi, you need a separate CAN controller chip that handles the CAN protocol details. Also a CAN transceiver is necessary to adapt the voltage levels to the CAN bus. There are several CAN expansion boards available for Raspberry Pi.

Typically an MCP2515 CAN controller chip is used and it is connected to the Raspberry Pi via the SPI bus having 3.3 V voltage levels. The MCP2515 chip uses a crystal oscillator, most often 10 MHz or 16 MHz.

Typically these pins are used to connect a CAN controller to the Raspberry Pi:

Pin number on Pi	Pi pin description	MCP2515 description	Comments
1	+3.3 V	+3.3 V	
6	GND	GND	
19	SPI_MOSI	SI	SDI
21	SPI_MISO	SO	SDO
22	GPIO25	int	Interrupt
23	SPI_SCLK	SCK	
24	SPI_CE0	CS	
(none)	(none)	reset	Use pull-up to 3.3 V

14.4 Install Raspbian on Raspberry Pi and get the CAN interface running

Download the latest Raspbian image, and install it on an SD card according to instructions: https://www.raspberrypi.org/downloads/raspbian/

It seems sufficient to use the "Raspbian Jessie Lite" version.

Plug in the SD-card in your Raspberry Pi, and give it some time to boot.

Find the IP address of the Raspberry Pi by using NMAP:

```
$ nmap -Pn -p 22 192.168.*.*
```

Log in to using SSH:

```
$ ssh pi@IPNUMBER
```

Default login credentials:

- User: `pi`
- Password: `raspberry`

Using the `raspi-config` tool, expand the filesystem and change the hostname:

```
$ sudo raspi-config
```

Also make sure to set it to boot in console mode, in order to save CPU resources.

Update Raspbian and install CAN tools:

```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
$ sudo apt-get install can-utils
```

The `upgrade` step is very time consuming.

Edit the `/boot/config.txt file` using the Nano editor:

```
$ sudo nano /boot/config.txt
```

Add these lines:

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=10000000,interrupt=25
```

Adapt the oscillator frequency to the crystal of your MCP2515 board and the GPIO pin used for interrupt. The most common oscillator frequencies seem to be 10 MHz and 16 MHz.

Reboot:

```
$ sudo reboot
```

After reboot the CAN interface is enabled by:

```
$ sudo ip link set can0 up type can bitrate 500000
```

List available network interfaces (including CAN):

```
$ ifconfig
```

Verify that the GUI not is running. There should be no line 'lightdm' when running `ps -ef`.

For Raspberry Pi 3 follow all the above steps, however, in the `/boot/config.txt file` add the these lines:

```
dtparam=spi=on
dtoverlay=mcp2515-can0,oscillator=10000000,interrupt=25
dtoverlay=spi-bcm2835
```

Cables on both Pis match the same pins.

14.5 Verify CAN hardware on two boards

Use two embedded Linux boards with CAN hardware and enable the CAN interface on each of them. The interface 'can0' should be listed as 'UP' on both boards when running ifconfig on them.

Connect a twisted pair CAN wire between them and make sure there is proper CAN line termination.

On one of the boards print out received CAN frames:

```
$ candump can0
```

On the other board generate random CAN frames and print them:

```
$ cangen can0 -v
```

The same CAN frames should now be seen in both boards.

14.6 Install Secure Gateway and other dependencies on BeagleBone or Raspberry Pi

Install the dependencies listed in the "core and tools" section in the "Dependencies" chapter.

Install sgframework and canadapter as described in the "Install" chapter.

14.7 Autostarting software using systemd

In order to automatically start the canadapter on booting of the BeagleBone, use a settings file for the systemd daemon. The contents of the file should be (adjust paths accordingly):

```
[Unit]
Description=CANadapter - CAN to MQTT adapter
Requires=network.target

[Service]
ExecStart=/usr/local/bin/canadapter \
  /home/debian/sgframework/examples/configfilesForCanadapter/climateservice_cansignals.kcd \
  -mqttfile /home/debian/sgframework/examples/configfilesForCanadapter/climateservice_mqttsignals.jso
  -mqttname canadapter -i can0
Restart=always

[Install]
WantedBy=multi-user.target
```

Put this file in this folder:

```
/etc/systemd/system/
```

Go to the same folder and type in this command to create a symbolic link and enable autostart for your service file:

```
$ sudo systemctl enable "filename".service
```

The service will not be started yet. To start it reboot the system or type the following command:

```
$ sudo systemctl restart "filename".service
```

Simularly, to automatically enable CAN interface *can0*:

```
[Unit]
Description=CAN hardware startup
After=network.target
Requires=network.target

[Service]
Type=oneshot
RemainAfterExit=true
ExecStart=/bin/sleep 5
ExecStart=/sbin/ip link set can0 up type can bitrate 500000
ExecStop=/sbin/ip link set can0 down

[Install]
WantedBy=multi-user.target
```

Note that systemd files installed by operating system packages typically end up in `/usr/lib/systemd/system` or `lib/systemd/system`.

CHAPTER 15

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

15.1 Types of Contributions

15.1.1 Report Bugs

Report bugs at https://github.com/caran/sgframework/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

15.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

15.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

15.1.4 Write Documentation

sgframework could always use more documentation, whether as part of the official sgframework docs, in docstrings, or even on the web in blog posts, articles, and such.

15.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/caran/sgframework/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

15.2 Get Started!

Ready to contribute? Here's how to set up *sgframework* for local development.

1. Fork the *sgframework* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/sgframework.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv sgframework
$ cd sgframework/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests (described elsewhere). Also flake8 and testing other Python versions with tox should be done.

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

15.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.3 and 3.4. Check https://travis-ci.org/caran/sgframework/pull_requests and make sure that the tests pass for all supported Python versions.

15.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_cansignal
```

CHAPTER 16

Credits

16.1 Development Lead

- Jonas Berg <caranopensource@semcon.com>

16.2 Contributors

- Basim Ali
- Markus Johansson
- Chuan Jin
- Gabriele Kasparaviciute

16.3 Acknowledgements

The Python file structure is set up using the Cookiecutter tool: https://github.com/audreyr/cookiecutter

Documentation is generated using the Sphinx tool: http://sphinx-doc.org/

CHAPTER 17

History

17.1 0.2.1 - 0.2.3 (2016-10-17)

- Documentation dependency update for readthedocs.org
- Documentation adjustment to display nicely on PyPI.

17.2 0.2.0 (2016-10-17)

- Improved documentation
- Examples included

17.3 0.1.0 (2016-07-26)

- First release on GitHub.

CHAPTER 18

Detailed documentation

18.1 sgframework package

18.1.1 sgframework.framework module

class `sgframework.framework.`**`App`**(*name*, *host*, *port=1883*, *certificate_directory=None*)

Bases: *`sgframework.framework.BaseFramework`*

App framework for the Secure Gateway

Sends commands to any resource. Handles incoming MQTT messages (data) from any resource.

It does not have any 'last will'. Typically sends (non retained=non persistent) commands to:

`command/`*resource_to_be_controlled*`/`*signalname*

and listens to data on topic:

`data/`*dataproducing_resource*`/`*signalname*

Parameters

- **`name`** (*`str`*) – Name of the app/resource. For resources, it is also used in the MQTT topic hierarchy.
- **`host`** (*`str`*) – Broker host name.
- **`port`** (*`int`*) – Broker port number.
- **`certificate_directory`** (*`str or None`*) – Full path to the directory of the certificate files.

`protocol`

enum in the Paho module

MQTT protocol version, defaults to `MQTTv31`, as older versions of the Mosquitto broker can not handle `MQTTv311`.

`tls_version`

enum in the ssl module

SSL protocol version, defaults to `ssl.PROTOCOL_TLSv1`

`qos`

int

MQTT quality of service. 0, 1 or 2. See Paho documentation. Default value `DEFAULT_QOS` is set in *`sgframework.constants`*.

timeout
numerical

MQTT socket timeout, when running the `loop()` method. Default value `DEFAULT_TIMEOUT`.

keepalive
numerical

MQTT keepalive message interval. Default value `DEFAULT_KEEPALIVE_TIME`.

Also the parameters appear as attributes. The public attributes are used when calling `start()`. Any changes are valid from next `start()`.

References to sub-objects:

- **mqttclient** (object): See Paho documentation
- **logger** (object): See Python standard library documentation
- **userdata** (whatever): Convenience object that is available for user code in callbacks. Not used by the framework itself.
- **on_broker_connectionstatus_info**: Implement this callback if you would like notifications on broker connection status changes. See below.

Callback to the user application on changed broker connection status:

```
on_broker_connectionstatus_info(app_or_resource, broker_connected)
```

Where *app_or_resource* is the app or resource object, and *broker_connected* (**bool**) is `True` if the user application is connected to the broker.

Callbacks to the user application on incoming information are registered using separate methods. The callbacks should have this interface:

```
callbackname(resource_or_app, messagetype, servicename, signalname, inputpayload)
```

where *messagetype*, *servicename*, *signalname* and *inputpayload* are strings. The callback is protected by try/except. The strings to the callback have been through `.strip()`.

When using echo and the returnvalue of the callback is `None`, the command payload is used in the echo. For returnvalues other then `None`, the echo payload will be `str(returnvalue)`. More than one input signal can use the same callback.

The certificate files should be named according to `CA_CERTS`, `CERTFILE` and `KEYFILE`.

class `sgframework.framework.`**`BaseFramework`**(*name*, *host*, *port=1883*, *certificate_directory=None*)

Bases: `object`

Parameters

- **name** (*str*) – Name of the app/resource. For resources, it is also used in the MQTT topic hierarchy.
- **host** (*str*) – Broker host name.
- **port** (*int*) – Broker port number.
- **certificate_directory** (*str or None*) – Full path to the directory of the certificate files.

protocol
enum in the Paho module

MQTT protocol version, defaults to `MQTTv31`, as older versions of the Mosquitto broker can not handle `MQTTv311`.

`tls_version`
enum in the ssl module

SSL protocol version, defaults to `ssl.PROTOCOL_TLSv1`

`qos`
int

MQTT quality of service. 0, 1 or 2. See Paho documentation. Default value `DEFAULT_QOS` is set in *`sgframework.constants`*.

`timeout`
numerical

MQTT socket timeout, when running the `loop()` method. Default value `DEFAULT_TIMEOUT`.

`keepalive`
numerical

MQTT keepalive message interval. Default value `DEFAULT_KEEPALIVE_TIME`.

Also the parameters appear as attributes. The public attributes are used when calling *`start()`*. Any changes are valid from next *`start()`*.

References to sub-objects:

- **mqttclient** (object): See Paho documentation
- **logger** (object): See Python standard library documentation
- **userdata** (whatever): Convenience object that is available for user code in callbacks. Not used by the framework itself.
- **on_broker_connectionstatus_info**: Implement this callback if you would like notifications on broker connection status changes. See below.

Callback to the user application on changed broker connection status:

```
on_broker_connectionstatus_info(app_or_resource, broker_connected)
```

Where *app_or_resource* is the app or resource object, and *broker_connected* (**bool**) is `True` if the user application is connected to the broker.

Callbacks to the user application on incoming information are registered using separate methods. The callbacks should have this interface:

```
callbackname(resource_or_app, messagetype, servicename, signalname, inputpayload)
```

where *messagetype*, *servicename*, *signalname* and *inputpayload* are strings. The callback is protected by try/except. The strings to the callback have been through `.strip()`.

When using echo and the returnvalue of the callback is `None`, the command payload is used in the echo. For returnvalues other then `None`, the echo payload will be `str(returnvalue)`. More than one input signal can use the same callback.

The certificate files should be named according to `CA_CERTS`, `CERTFILE` and `KEYFILE`.

`CA_CERTS` = 'ca_public_certificate.pem'

`CERTFILE` = 'public_certificate.pem'

`KEYFILE` = 'private_key.pem'

`PAYLOAD_FALSE` = 'False'

`PAYLOAD_TRUE` = 'True'

`PREFIX_RESOURCEAVAILABLE` = 'resourceavailable'

`_on_connect` (*mqttclient*, *userdata*, *flags*, *rc*)
MQTT callback at connection attempts.

This callback is responsible for doing the subscriptions, and to publish capabilities and default values.

Method signature according to Paho documentation.

`_on_disconnect` (*mqttclient*, *userdata*, *rc*)
MQTT callback at disconnect.

Method signature according to Paho documentation.

`_on_incoming_message` (*mqttclient*, *userdata*, *message*)
MQTT callback at incoming messages.

Executes a preregistered callback, and publishes an echo (if configured).

Updates the default value for echoed signals if configured.

Method signature according to Paho documentation.

`_on_mqttclient_log_event` (*mqttclient*, *userdata*, *level*, *buf*)
MQTT callback at log event.

Method signature according to Paho documentation.

`_on_publish` (*mqttclient*, *userdata*, *mid*)
MQTT callback at publication confirmation.

Method signature according to Paho documentation.

`_on_subscribe` (*mqttclient*, *userdata*, *mid*, *granted_qos*)
MQTT callback at subscribe.

Method signature according to Paho documentation.

`_on_unsubscribe` (*mqttclient*, *userdata*, *mid*)
MQTT callback at unsubscribe.

Method signature according to Paho documentation.

`_publish_capablities_and_defaultvalues` ()
To be overrided

`_register_inputsignal` (*messagetype*, *servicename*, *signalname*, *callback*, *callback_on_change_only=False*, *echo=False*, *send_echo_as_retained=False*, *defaultvalue=None*)
Register a callback for an incoming MQTT message.

Parameters

- **`messagetype`** (`str`) – One of the predefined message types (also known as prefix).
- **`servicename`** (`str`) – service name
- **`signalname`** (`str`) – signal name
- **`callback`** (`function`) – Callback that will be used when a signal is received.
- **`callback_on_change_only`** (`bool`) – Trigger callback only for changed payload.
- **`echo`** (`bool`) – True if the incoming signal should be echoed back (as "data")

- **send_echo_as_retained** (*bool*) – True if the echo should be published as retained.
- **defaultvalue** – Value to be echoed on startup. Set to None to avoid sending. The value is converted to a string before sending. It will be updated by _on_incoming_message().

For details on the callback, see the class documentation.

Subscribes to: *messagetype*/*servicename*/*signalname*

The *messagetype* can be `data`, `dataavailable`, `command`, `commandavailable`, `resourceavailable`.

For example: `data/climateservice/actualindoortemperature`.

_register_outputsignal(*messagetype*, *servicename*, *signalname*, *defaultvalue*, *send_as_retained*)

Registering outgoing MQTT messages.

This is typically used for automatically send availability information.

Parameters

- **messagetype** (*str*) – One of the predefined message types (also known as prefix), most often `data`.
- **servicename** (*str*) – service name, most often `self.name`.
- **signalname** (*str*) – signal name
- **defaultvalue** – Value to be sent on startup. Set to None to avoid sending. The value is converted to a string before sending.
- **send_as_retained** (*bool*) – True if the signal should be published as retained.

Publishes to: *messagetype*/*servicename*/*signalname*

for example: `data/climateservice/actualindoortemperature`.

There is also a mechanism to automatically publish availability topics, for example: `dataavailable/climateservice/actualindoortemperature`.

_set_broker_connectionstatus(*broker_connected*)

Set information whether the broker is connected. This is triggering a callback to the user script.

The information is not stored in the framework, it is the responsibility of the user script.

Parameters **broker_connected** (*bool*) – Indicates whether the broker is connected or not

_subscribe_to_inputsignals()

Do the subscription to input signals

get_descriptive_ascii_art()

Display an overview with registered incoming and outgoing topics.

Returns A multi-line string.

loop()

Run network activities.

This function needs to be called frequently to keep the network traffic alive, if not using threaded networking. It will block until a message is received, or until the self.timeout value.

If not connected to the broker, it will try to connect once.

Do not use this function when running threaded networking.

`register_incoming_availability`(*prefix*, *servicename*, *signalname*, *callback*)
Register a callback for incoming availability information (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data etc from other resources).

Parameters

- **`prefix`** (*`str`*) – one of PREFIX_COMMANDAVAILABLE, PREFIX_DATAAVAILABLE (or maybe PREFIX_RESOURCEAVAILABLE)
- **`servicename`** (*`str`*) – name of the service sending the availability info
- **`signalname`** (*`str`*) – name of the data or command
- **`callback`** (*`function`*) – Callback that will be used when availability information is received.

When registering a callback for RESOURCEAVAILABLE the actual value of the signalname is not used. Just pass in any string.

For details on the callback, see the class documentation.

Subscribes to: *prefix/servicename/signalname*

for example: `dataavailable/climateservice/actualindoortemperature`.

`register_incoming_data`(*servicename*, *signalname*, *callback*, *callback_on_change_only=False*)
Register a callback for incoming data (incoming MQTT message).

Primarily useful for apps (but is useful for resources to receive data from other resources).

Parameters

- **`servicename`** (*`str`*) – name of the service sending the data
- **`signalname`** (*`str`*) – name of the signal
- **`callback`** (*`function`*) – Callback that will be used when data is received.
- **`callback_on_change_only`** (*`bool`*) – Trigger callback only for changed payload.

For details on the callback, see the class documentation.

Subscribes to: `data/`*servicename/signalname*

for example: `data/climateservice/actualindoortemperature`.

`send_command`(*servicename*, *signalname*, *value*, *send_command_as_retained=False*)
Send a command.

Primarily useful for apps (but is useful for resources to control other resources).

Parameters

- **`servicename`** (*`str`*) – destination service name
- **`signalname`** (*`str`*) – destination signal name
- **`value`** – Value to be sent. Is converted to a string before sending.
- **`send_command_as_retained`** (*`bool`*) – Publish the command as retained.

Sends messages on topic: `command/`*servicename/signalname*

for example `command/climateservice/aircondition`.

Most often commands are sent as non-retained messages.

start(*use_threaded_networking=False*, *use_clean_session=True*)
Connect to the broker.

Parameters

- **use_threaded_networking** (*bool*) – Start MQTT networking activity in a separate thread.
- **use_clean_session** (*bool*) – Connect to broker using a clean session.

If not using threaded networking, you need to call the `loop()` method frequently.

If using a clean session, also the client name is changed to include the process ID. This in order to avoid client name collisions in the broker.

stop()
Disconnect from the broker

class sgframework.framework.**Inputsignalinfo**(*messagetype*, *servicename*, *signalname*, *callback*, *callback_on_change_only*, *echo*, *send_echo_as_retained*, *defaultvalue*)
Bases: `object`

Object for storing configuration information about incoming MQTT messages.

Storage of incoming signal definitions that should be subscribed to. It can be data, dataavailable, command, commandavailable, resourceavailable. Also holds the callback to be used at incoming signals, and possibly a copy of last received payload.

Arguments are described in the `BaseFramework._register_inputsignal()` method.

For details on the callback signature, see the `BaseFramework` documentation.

TODO: .messagetype should be a property.

class sgframework.framework.**Outputsignalinfo**(*messagetype*, *servicename*, *signalname*, *defaultvalue*, *send_as_retained*)
Bases: `object`

Object for storing configuration information about (some of the) outgoing MQTT messages, typically outgoing data (not outgoing commands).

Arguments are described in the `BaseFramework._register_outputsignal()` method.

TODO: .messagetype should be a property.

class sgframework.framework.**Resource**(*name*, *host*, *port=1883*, *certificate_directory=None*)
Bases: `sgframework.framework.BaseFramework`

Resource framework for the Secure Gateway

Receives commands from apps (incoming MQTT messages). Sends data to apps (outgoing MQTT messages).

It can also recieve incoming data from other resources, and can send commands to other resources.

The resource name is typically part of incoming and outgoing message topics:

`command/`*myresourcename*/*signalname*

`data/`*myresourcename*/*signalname*

Also publishes availability of commands and data, using retained messages:

`commandavailable/`*myresourcename*/*signalname*

`dataavailable/`*myresourcename*/*signalname*

When starting up, it sends 'True' to the 'last will' topic in a retained message:

`resourceavailable/`*myresourcename*`/presence`

The broker is automatically broadcasting 'False' on 'last will' topic at lost connection.

Parameters

- **name** (*str*) – Name of the app/resource. For resources, it is also used in the MQTT topic hierarchy.
- **host** (*str*) – Broker host name.
- **port** (*int*) – Broker port number.
- **certificate_directory** (*str or None*) – Full path to the directory of the certificate files.

protocol

enum in the Paho module

MQTT protocol version, defaults to `MQTTv31`, as older versions of the Mosquitto broker can not handle `MQTTv311`.

tls_version

enum in the ssl module

SSL protocol version, defaults to `ssl.PROTOCOL_TLSv1`

qos

int

MQTT quality of service. 0, 1 or 2. See Paho documentation. Default value `DEFAULT_QOS` is set in `sgframework.constants`.

timeout

numerical

MQTT socket timeout, when running the `loop()` method. Default value `DEFAULT_TIMEOUT`.

keepalive

numerical

MQTT keepalive message interval. Default value `DEFAULT_KEEPALIVE_TIME`.

Also the parameters appear as attributes. The public attributes are used when calling `start()`. Any changes are valid from next `start()`.

References to sub-objects:

- **mqttclient** (object): See Paho documentation
- **logger** (object): See Python standard library documentation
- **userdata** (whatever): Convenience object that is available for user code in callbacks. Not used by the framework itself.
- **on_broker_connectionstatus_info**: Implement this callback if you would like notifications on broker connection status changes. See below.

Callback to the user application on changed broker connection status:

```
on_broker_connectionstatus_info(app_or_resource, broker_connected)
```

Where *app_or_resource* is the app or resource object, and *broker_connected* (**bool**) is `True` if the user application is connected to the broker.

Callbacks to the user application on incoming information are registered using separate methods. The callbacks should have this interface:

```
callbackname(resource_or_app, messagetype, servicename, signalname, inputpayload)
```

where *messagetype*, *servicename*, *signalname* and *inputpayload* are strings. The callback is protected by try/except. The strings to the callback have been through `.strip()`.

When using echo and the returnvalue of the callback is `None`, the command payload is used in the echo. For returnvalues other then `None`, the echo payload will be `str(returnvalue)`. More than one input signal can use the same callback.

The certificate files should be named according to `CA_CERTS`, `CERTFILE` and `KEYFILE`.

`_publish_capablities_and_defaultvalues``()`

Sends 'True' to the topic: `resourceavailable/`*myresourcename*`/presence`

For data in outputsignal storage:

- Sends `dataavailable/`*myresourcename*/*signalname*
- If configured, sends defaultvalue `data/`*myresourcename*/*signalname*

For commands in inputsignal storage:

- Sends `commandavailable/`*myresourcename*/*signalname*
- If echo configured, sends `dataavailable/`*myresourcename*/*signalname*
- If configured, sends defaultvalue `data/`*myresourcename*/*signalname*

`register_incoming_command``(`*signalname*, *callback*, *callback_on_change_only=False*, *echo=True*, *send_echo_as_retained=False*, *defaultvalue=None*`)`

Register a callback for an incoming command (incoming MQTT message).

Parameters

- **`signalname`** (*`str`*) – command name
- **`callback`** (*`function`*) – Callback that will be used when a command is received.
- **`callback_on_change_only`** (*`bool`*) – Trigger callback only for changed payload.
- **`echo`** (*`bool`*) – True if the incoming command should be echoed back (as "data")
- **`send_echo_as_retained`** (*`bool`*) – True if the echo should be published as retained.
- **`defaultvalue`** – Value to be echoed on startup and reconnect. Set to None to avoid sending. The value is converted to a string before sending. It will be updated by the internal *`_on_incoming_message()`* callback for incoming MQTT messages.

For details on the callback, see the class documentation.

Subscribes to: `command/`*myresourcename*/*signalname*

When the resource is starting, it is publishing a retained message to:

`commandavailable/`*myresourcename*/*signalname*

`register_outgoing_data``(`*signalname*, *defaultvalue=None*, *send_data_as_retained=False*`)`

Pre-register information on a outgoing data topic (MQTT messages). Note that the actual data sending is later done with the *`send_data()`* method.

Parameters

- **`signalname`** (*`str`*) – signal name
- **`defaultvalue`** – Value to be sent on startup and reconnect. Set to None to avoid sending. The value is converted to a string before sending. It will be updated by send_data().

- **`send_data_as_retained`** (*`bool`*) – Whether the data should be published as retained

When the resource is starting, it is publishing a retained message to:

`dataavailable/`*myresourcename*`/`*signalname*

Upon sending data, the topic is: `data/`*myresourcename*`/`*signalname*

for example: `data/climateservice/actualindoortemperature`.

Typically the data is published using non-retained messages.

`send_data`(*signalname*, *value*)
Send data on a pre-registered topic.

Parameters

- **`signalname`** (*`str`*) – signal name
- **`value`** – Value to be sent. Is convered to a string before sending.

Sends to the topic: `data/`*myresourcename*`/`*signalname*

for example: `data/climateservice/actualindoortemperature`.

Updates the defaultvalue for this signal.

Whether the signal should be sent as retained or not is set already during registration.

18.1.2 sgframework.exceptions module

exception `sgframework.exceptions.`**`SGFrameworkException`**
Bases: `Exception`

Base exception for the SG framework

18.1.3 sgframework.constants module

18.1.4 sgframework.version module

18.1.5 Module contents

18.2 canadapter script

18.2.1 canadapter

`canadapter.`**`init_canadapter`**()
Initialize the canadapter.

Returns a 'resource' in the sgframework terminology.

`canadapter.`**`loop_canadapter`**(*resource*)

`canadapter.`**`main`**()

`canadapter.`**`on_send_can_data`**(*resource*, *messagetype*, *servicename*, *command_name_mqtt*, *command_payload_mqtt*)
Callback for use when receiving a MQTT message. Sends a CAN message.

For callback interface, see sgframework.BaseFramework() documentation.

`canadapter.`**`signal_handler`**(*signum*, *frame*)

18.2.2 canadapterlib

class `canadapterlib.`**`Converter`**(*can_config*, *mqttfile_path=None*, *sort_json_keys=False*)

Converter between CAN and MQTT.

Does not store any CAN or MQTT messages.

Parameters

- **`can_config`** (*`can4python.Configuration`*) –
- **`mqttfile_path`** (*`str or None`*) – Full path to the configuration (JSON) file
- **`sort_json_keys`** (*`bool`*) – Sort keys in resulting JSON strings.

If the mqttfile_path argument not is given, it listens to all CAN signals (without name or value conversion).

`canframe_to_mqtt`(*frame*)

Parameters **`frame`** (*`can4python.CanFrame`*) – Incoming CAN frame with data.

Returns a list of MQTT messages, each represented as the tuple (MQTT signalname, MQTT payload). The payload is later converted to a string before sending.

`mqtt_to_cansignals`(*command_name*, *command_payload*)

Parameters

- **`command_name`** (*`str`*) – Incoming MQTT command name (from the splitted MQTT topic)
- **`command_payload`** (*`str`*) – Incoming MQTT payload

Returns a dictionary with the signal values to send. The keys are the CAN signalnames (*str*), and the items are the CAN values (*numerical* or *None*). If the CAN value is *None* the default value is used.

`get_definitions_incoming_mqtt_command`()

Find the incoming MQTT command signalnames etc.

Returns a list of dictionaries, each having the arguments for the resource.register_incoming_command() call.

Note that the 'callback' keyword argument not is set in the output from this function, but needs to be set separately.

`get_definitions_outgoing_mqtt_data`()

Find the outgoing MQTT data signalnames etc.

Returns a list of dictionaries, each having the arguments for the resource.register_outgoing_data() call.

`get_descriptive_ascii_art`()

Return a string describing the conversion between CAN and MQTT.

`_get_incoming_cansignal_names`()

Return a list (str) of incoming cansignal names (according to the CAN configuration).

`_get_node_incoming_can_frameids`()

Return a list (int) of all incoming CAN frame ids for this node, according to the CAN configutation.

`_get_node_outgoing_can_frameids`()

Return a list (int) of all outgoing CAN frame ids for this node, according to the CAN configutation.

`_get_canframe_id`(*can_signal_name*)

Find the CAN frame id for a CAN signal name.

Parameters **`can_signal_name`** (`str`) –

Returns the frame_id (int).

Raises KeyError if the can_signal_name not is found.

`_precalculate_frame_id`(*translationinfo*)

Set the frame_id field of the translationinfo.

Does not return anything.

class `canadapterlib.`**`IndividualInfo`**(*can_name*, *mqtt_name=None*, *send_can=False*, *echo_mqtt=False*, *receive_can=True*, *multiplier=1.0*, *frame_id=None*, *mqtt_type=<class 'float'>*)

A class for describing the translation between a CAN signal and an individual MQTT signal.

`can_name`

str

Signal name in the KCD file for CAN

`mqtt_name`

str or None

Signal name on MQTT (part of the topic). Defaults to None (use same name on MQTT as on CAN.

`mqtt_type`

type

Conversion function used to convert the value to correct type inside the MQTT message (which itself is a string). Defaults to float.

`send_can`

bool

Whether the signal should be sent to CAN. Defaults to False.

`echo_mqtt`

bool

Whether the incoming MQTT message should be echoed back. Defaults to False.

`receive_can`

bool

Whether the signal should be sent to MQTT. Defaults to True.

`multiplier`

float

Multiplier when converting a CAN signal to an MQTT signal. In the other direction is 1/multiplier used. Defaults to 1.0.

`frame_id`

int or None

The id of the CAN frame the signal is using

class `canadapterlib.`**`AggregateInfo`**(*mqtt_name*, *send_can=False*, *receive_can=True*, *echo_mqtt=False*, *frame_id=None*)

A class for describing the translation between a CAN signal aggregate and a MQTT signal.

All CAN signals must be located in the same CAN frame. The send_can and receive_can fields of the subsignals are not used.

`mqtt_name`
str

Signal name on MQTT (part of the topic).

`send_can`
bool

Whether the subsignals should be sent to CAN. Defaults to False.

`receive_can`
bool

Whether the aggregate should be sent to MQTT. Defaults to True.

`echo_mqtt`
bool

Whether the incoming MQTT message should be echoed back. Defaults to False.

`frame_id`
int or None

The id of the CAN frame the aggregate data is using

`subsignals`
list of IndividualInfo

Definitions for the signals within the aggregate

`canadapterlib.`**`translationfile_read`**(*filename*)

Read a translation file, having the CAN signal names and the MQTT signal names etc.

Parameters **`filename`** (`str`) – Full path to the input file.

The input file should be a valid JSON file, having a top object with the name 'entities' and the objects 'aggregates' and 'signals'. Item inside 'signals' is a list of signaldefinition objects. Item in aggregates are several 'signals'.

See the sgframework documentation for a more detailed discussion on the file format.

Returns a single list containing AggregateInfo and IndividualInfo. The order in the list is not relevant.

`canadapterlib.`**`parse_signal`**(*json_signal*, *filename*)

Parse signal information from a JSON dict, and convert to an instance of IndividualInfo.

Parameters

- **`json_signal`** – a dict from a (part of a) parsed JSON file
- **`filename`** (`str`) – Filename (for use in error messages)

Returns An IndividualInfo instance

`canadapterlib.`**`parse_aggregate`**(*json_aggregate*, *filename*)

Parse signal information from a JSON dict, and convert to an instance of AggregateInfo.

Parameters

- **`json_aggregate`** – a dict from a (part of a) parsed JSON file
- **`filename`** (`str`) – Filename (for use in error messages)

Returns An AggregateInfo instance

18.3 servicemanager script

18.3.1 servicemanager

`servicemanager.`**`main`**`()`

`servicemanager.`**`on_incoming_message`**`(`*self*, *mqttclient*, *userdata*, *message*`)`

MQTT callback at incoming messages.

Method signature according to Paho documentation.

`servicemanager.`**`signal_handler`**`(`*signum*, *frame*`)`

`servicemanager.`**`subscribe_to_inputsignals`**`(`*self*`)`

Override the _subscribe_to_inputsignals method in sgframework.framework.BaseFramework

18.4 climateapp script

18.4.1 climateapp

class `climateapp.`**`CommandlineAppDisplay`**`(`*app*`)`

Bases: `object`

`_initialize_values``()`

`aircondition_state`

`broker_connectionstatus`

`close``()`

`enginespeed`

`indoortemperature`

`loop``()`

`redraw``()`

`resource_online`

`vehiclespeed`

class `climateapp.`**`GraphicalAppDisplay`**`(`*app*`)`

Bases: *`climateapp.CommandlineAppDisplay`*

`DISPLAY_TITLE` **= 'Climate app'**

`TEMPLATE_AIRCONDITION` **= 'Air condition: {}'**

`TEMPLATE_CONNECTION` **= 'Connection status: {}'**

`TEMPLATE_ENGINESPEED` **= 'Engine speed: {:.0f} RPM'**

`TEMPLATE_TEMPERATURE` **= 'In-car temperature: {:.1f} deg C'**

`TEMPLATE_VEHICLESPEED` **= 'Vehicle speed: {:.1f} km/h'**

`_button_off_handler``(`*event*`)`

`_button_on_handler``(`*event*`)`

close()
Close the GUI

loop()
Update the GUI

redraw()

climateapp.**init_climateapp**()

climateapp.**loop_climateapp**(*app*)

climateapp.**main**()

climateapp.**on_broker_connectionstatus_info**(*app*, *broker_connected*)
Callback for use when the broker connection status info is available.

climateapp.**on_incoming_data**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)
Callback for use when receiving a MQTT message.

Sets display fields.

climateapp.**on_resource_presence**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)
Callback for use when receiving a MQTT message.

Sets the presence information (on the display) about the resource in use.

18.5 vehiclesimulator script

18.5.1 vehiclesimulator

vehiclesimulator.**init_vehiclesimulator**()
Initialize the vehicle simulator.

Returns the tuple (temperature_simulator, speed_simulator, canbus)

vehiclesimulator.**loop_vehiclesimulator**(*temperature_simulator*, *speed_simulator*, *canbus*)

vehiclesimulator.**main**()

18.5.2 vehiclesimulation utilities

class vehiclesimulationutilities.**CabinTemperatureSimulator**
Bases: `object`

Simulate the indoor temperature of a vehicle.

The temperature will increase to TEMPERATURE_HOT, but will decrease to TEMPERATURE_COLD when the air conditioner is turned on.

The warming and cooling speeds are affected by GAIN_WARMING and GAIN_COOLING. A random temperature noise is added to the measurement.

This class holds no timer or timing information, so the average rate of temperature change is dependent on how frequently you call getNewTemperature().

temperature
float

Indoor temperature in deg C

aircondition_state
bool

Boolean indicating whether the AC is on

GAIN_COOLING = 0.05

GAIN_WARMING = 0.02

TEMPERATURE_COLD = 18.0

TEMPERATURE_HOT = 32.0

TEMPERATURE_NOISE_DEVIATION = 0.03

aircondition_state

get_new_temperature()
Calculate a new temperature.

Returns the new temperature in deg C.

class vehiclesimulationutilities.**VehicleSpeedSimulator**
Bases: object

Simulate the speed of a vehicle.

The step size in the vehicle speed is calculated with a normalized distribution using the values:

- SPEED_STEPSIZE (Should be >0)
- SPEED_STEPDEVIATION (Should be >0)

When the speed reaches SPEED_MAX, the speed will start to decrease instead. Similarly it starts to increase again at SPEED_MIN.

This class holds no timer or timing information, so the average rate of speed change is dependent on how frequently you call getNewRandomizedSpeed().

currentspeed
float

Vehicle speed in km/h

SPEED_MAX = 110.0

SPEED_MIN = 3.0

SPEED_STEPDEVIATION = 0.9

SPEED_STEPSIZE = 0.3

get_new_randomized_speed()
Calculate a new vehicle speed.

Returns the new value of currentspeed in km/h.

vehiclesimulationutilities.**calculate_engine_speed**(*vehiclespeed*)
Calculate the simulated enginespeed.

Parameters **vehiclespeed** (*float*) – Speed in km/h

Returns: Engine speed (float) in RPM (revolutions per minute)

18.6 taxisignapp script

18.6.1 taxisignapp

class `taxisignapp.`**`CommandlineAppDisplay`**(*app*)

Bases: `object`

`TAXISIGN_STATUSTEXT_TEMPLATE` = **'Taxi sign status: {:<8}'**

`broker_connectionstatus`

`close`()

`loop`()

`redraw`()

`resource_online`

`taxisign_state`

class `taxisignapp.`**`GraphicalAppDisplay`**(*app*)

Bases: *`taxisignapp.CommandlineAppDisplay`*

`DISPLAY_TITLE` = **'Taxi app'**

`_button_off_handler`(*event*)

`_button_on_handler`(*event*)

`close`()

Close the GUI

`loop`()

Update the GUI

`redraw`()

`taxisignapp.`**`init_taxisignapp`**()

`taxisignapp.`**`loop_taxisignapp`**(*app*)

`taxisignapp.`**`main`**()

`taxisignapp.`**`on_broker_connectionstatus_info`**(*app*, *broker_connected*)

Callback for use when the broker connection status info is available.

`taxisignapp.`**`on_resource_presence`**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)

Callback for use when receiving a MQTT message.

Sets the presence information for the taxi sign.

`taxisignapp.`**`on_taxisign_state_data`**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)

Callback for use when receiving a MQTT message.

Sets the state of the taxi sign.

18.7 taxisignservice script

18.7.1 taxisignservice

class `taxisignservice.`**`CommandlineDummyTaxisign`**

Bases: `object`

Taxi sign simulator for command line.

Prints out whether the taxi sign is on or off.

* **`state`**

  bool

  Set to True to turn on the taxi sign.

`broker_connectionstatus`

`close``()`

`loop``()`

`redraw``()`

`state`

class `taxisignservice.`**`GraphicalDummyTaxisign`**

Bases: *`taxisignservice.CommandlineDummyTaxisign`*

Graphical taxi sign simulator.

Shows a dark or bright taxi sign image in the GUI.

* **`state`**

  bool

  Set to True to turn on the taxi sign.

`BROKERTEXT_TEMPLATE` **= 'Broker: {:<7s}'**

`DISPLAY_TITLE` **= 'Taxi sign simulator'**

`FILENAME_TAXISIGN_OFF` **= '/home/docs/checkouts/readthedocs.org/user_builds/sgframework/checkouts/latest/examp**

`FILENAME_TAXISIGN_ON` **= '/home/docs/checkouts/readthedocs.org/user_builds/sgframework/checkouts/latest/example**

`close``()`

Close the GUI

`loop``()`

Update the GUI

`redraw``()`

`taxisignservice.`**`init_taxisignservice`**`()`

Initialize the taxi sign service.

Returns ..

`taxisignservice.`**`loop_taxisignservice`**`(`*resource*`)`

`taxisignservice.`**`main`**`()`

`taxisignservice.`**`on_broker_connectionstatus_info`**`(`*resource*, *broker_connected*`)`

Callback for use when the broker connection status info is available.

taxisignservice.**on_taxisign_state_command**(*resource*, *messagetype*, *servicename*, *commandname*, *commandpayload*)

Callback for use when receiving a MQTT message.

Sets the state of the taxi sign.

Returns the payload (str) that should be included in the command echo MQTT message.

18.8 Drivers for taxi sign hardware

18.8.1 taxisign_driver

class taxisign_driver.**Taxisign**

Bases: object

Taxi sign representation.

For controlling a Taxi sign from a Beaglebone. This is basically the Beaglebone output_pin_driver, together with hardware-describing constants.

Module constants describing the hardware:

- GPIO_NUMBER (int): GPIO number for pin connected to relay driver circuit.
- IS_INVERTED (bool): Set to True if the relay and its driver circuit are connected such that the taxi sign turns off for GPIO=high.

*** state**

bool

Set to True to turn on the taxi sign.

loop()

state

18.8.2 output_pin_driver

class output_pin_driver.**Outputpin**(*GPIO_number*)

Bases: object

GPIO output pin representation.

For controlling a GPIO output pin on a Beaglebone. Note that root permissions are required.

*** state**

bool

Turn on the GPIO output pin if the value is True.

state

CHAPTER 19

Unittests and integration tests

In order to run the tests, you need to install sgframework, for example in development mode (symbolic links to source):

```
$ sudo make develop
```

All dependencies, also for the examples must be installed.

Enable the virtual can bus:

```
$ sudo make vcan
```

Make sure that Mosquitto MQTT broker is running.

Run the core tests (also on an embedded Linux board, for example a Beaglebone):

```
$ make test
```

In order to also test the graphical example apps, use:

```
$ make test-all
```

To run individual test files, execute them from the project root directory (for example):

```
$ python3 tests/test_framework_resource.py
```

If you would like to run a single test case in a test file, adjust the `if __name__ == '__main__':` section of the file.

19.1 Test documentation for core framework

19.1.1 test_framework_app

Tests for the app part of the sgframework.

class `tests.test_framework_app.`**`TestFrameworkApp`**(*methodName='runTest'*)

`OUTPUT_FILE_SUBSCRIBER` = **'temporary-sub.txt'**

`setUp`()

`tearDown`()

`testConstructor`()

testLoop()

testRepr()

tests.test_framework_app.**on_testservice_command_availability**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)

tests.test_framework_app.**on_testservice_state_data**(*app*, *messagetype*, *servicename*, *signalname*, *payload*)

19.1.2 test_framework_resource

Tests for the resource part of the sgframework.

class tests.test_framework_resource.**TestBaseFramework**(*methodName='runTest'*)

testRepr()

class tests.test_framework_resource.**TestFrameworkResource**(*methodName='runTest'*)

OUTPUT_FILE_SUBSCRIBER = 'temporary-sub.txt'

setUp()

tearDown()

testConstructor()

testLoop()

testLoopBeforeStart()

testLoopNoCleanSession()

testLoopQos2()

testLoopThreaded()

testRepr()

testSendCommandBeforeStart()

testSendDataBeforeStart()

testStartTwice()

testStopBeforeStart()

class tests.test_framework_resource.**TestSignalInfoObject**(*methodName='runTest'*)

testConstructorInput()

testConstructorOutput()

testWrongConstructorInput()

testWrongConstructorOutput()

19.2 Test documentation for scripts

19.2.1 test_canadapter

Tests for the canadapter

class `tests.test_canadapter.`**`TestCanAdapter`**`(`*methodName='runTest'*`)`

> **`OUTPUT_FILE_CANDUMPER`** = **'temporary-candump.txt'**
>
> **`OUTPUT_FILE_SUBSCRIBER`** = **'temporary-sub.txt'**
>
> **`setUp`**`()`
>
> **`tearDown`**`()`
>
> **`testConstructor`**`()`
>
> **`testConstructorJSON`**`()`
>
> **`testConstructorVerbose`**`()`
>
> **`testConstructorVerbose2`**`()`
>
> **`testConstructorWrongArguments`**`()`
>
> **`testConstructorWrongCanInterface`**`()`
>
> **`testHelpText`**`()`
>
> **`testLoopAggregates`**`()`
>
> **`testLoopIndividualsignals`**`()`
>
> **`testLoopThrottleCanFrames`**`()`

class `tests.test_canadapter.`**`TestConverter`**`(`*methodName='runTest'*`)`

> **`test_converter_aggregate`**`()`
>
> **`test_converter_individual`**`()`

19.2.2 test_servicemanager

Tests for the servicemanager

class `tests.test_servicemanager.`**`TestServicemanager`**`(`*methodName='runTest'*`)`

> **`OUTPUT_FILE_SUBSCRIBER`** = **'temporary-sub.txt'**
>
> **`setUp`**`()`
>
> **`tearDown`**`()`
>
> **`testServicemanager`**`()`

19.3 Test documentation for minimal examples

19.3.1 test_minimal_taxisign

Tests for the minimal resource example

class `tests.test_minimal_taxisign.`**`TestMinimalTaxisign`**(*methodName='runTest'*)

> **`OUTPUT_FILE_SUBSCRIBER`** = **'temporary-sub.txt'**
>
> **`OUTPUT_FILE_TAXISIGN`** = **'temporary-taxisign.txt'**
>
> **`setUp`**()
>
> **`tearDown`**()
>
> **`testMinimalTaxisign`**()

19.3.2 test_minimal_taxiapp

Tests for the minimal app example

class `tests.test_minimal_taxiapp.`**`TestMinimalTaxiapp`**(*methodName='runTest'*)

> **`OUTPUT_FILE_APP`** = **'temporary-app.txt'**
>
> **`OUTPUT_FILE_SUBSCRIBER`** = **'temporary-sub.txt'**
>
> **`setUp`**()
>
> **`tearDown`**()
>
> **`testMinimalTaxiapp`**()

19.4 Test documentation for elaborate examples

19.4.1 test_vehiclesimulator

Tests for the vehicle simulator

class `tests.test_vehiclesimulator.`**`TestVehicleSimulator`**(*methodName='runTest'*)

> **`OUTPUT_FILE_CANDUMPER`** = **'temporary-candump.txt'**
>
> **`setUp`**()
>
> **`tearDown`**()
>
> **`testConstructor`**()
>
> **`testConstructorVerbose`**()
>
> **`testConstructorVerboser`**()
>
> **`testConstructorWrongArguments`**()
>
> **`testConstructorWrongCanInterface`**()

testHelpText()

testLoop()

tests.test_vehiclesimulator.**disable_virtual_can_bus**()

tests.test_vehiclesimulator.**enable_virtual_can_bus**()

19.4.2 test_climateapp

Tests for the app controlling the climate service.

class tests.test_climateapp.**TestClimateApp**(*methodName='runTest'*)

OUTPUT_FILE_SUBSCRIBER = 'temporary-sub.txt'

setUp()

tearDown()

testConstructor()

testConstructorCommandLine()

testConstructorGraphical()

testConstructorWrongArguments()

testHelpText()

testLoop()

19.4.3 test_taxisignservice

Tests for the taxi sign.

class tests.test_taxisignservice.**TestTaxisignService**(*methodName='runTest'*)

OUTPUT_FILE_SUBSCRIBER = 'temporary-sub.txt'

setUp()

tearDown()

testConstructor()

testConstructorCommandLine()

testConstructorGraphical()

testConstructorWrongArguments()

testHelpText()

testLoop()

19.4.4 test_taxisignapp

Tests for the app controlling the taxi sign.

class `tests.test_taxisignapp.`**`TestTaxisignApp`**(*methodName='runTest'*)

> **`OUTPUT_FILE_SUBSCRIBER`** = **'temporary-sub.txt'**
>
> **`setUp`**()
>
> **`tearDown`**()
>
> **`testConstructor`**()
>
> **`testConstructorCommandLine`**()
>
> **`testConstructorGraphical`**()
>
> **`testConstructorWrongArguments`**()
>
> **`testHelpText`**()
>
> **`testLoop`**()

CHAPTER 20

Developer information

20.1 Measuring test coverage

In order to have the 'coverage' program to measure also the Python scripts that are running in subprocesses, you need to adjust `sitecustomize.py` file. That will start the 'coverage' program when the appropriate environment variable is set.

Add this to your `sitecustomize.py` file:

```
try:
    import coverage
    coverage.process_startup()
except ImportError:
    pass
```

For more details, see http://coverage.readthedocs.org/en/latest/subprocess.html

Note that the Python programs running in the subprocess must handle SIGTERM. For example, to measure the test coverage for the `minimaltaxisign.py` file, insert this code:

```
import signal
import sys

def signal_handler(signum, frame):
    print('Handled Linux signal number:', signum)
    sys.exit()

signal.signal(signal.SIGTERM, signal_handler)
```

Alternatively, put it in some other code that imports your program.

It seems important to terminate the process gently:

```
myprocess.send_signal(signal.SIGINT)
```

20.2 TODO-list

Improve documentation:

- Update history release list.

Verify that the canadapter script is installed.

20.3 For next release

Implement:

- JSON schema verification of settings file
- Add support for another conversion parameter in the JSON file. As the "canMultiplier" is used to multiply a CAN signal when converting to MQTT messages, we should have a parameter to add a constant value (offset) when converting to MQTT messages.

Documentation and test improvements:

- Schema documentation.
- Describe logging in systemd.
- Network discovery using Avahi.
- Dynamically change access to applications.
- Test output_pin_driver using http://stackoverflow.com/questions/8166633/mocking-file-objects-or-iterables-in-python .
- Try running several canadapters on a single Beaglebone.

CHAPTER 21

Indices and tables

- genindex
- modindex
- search

Python Module Index