Last modified 3 days ago

Tutorial for Secure Gateway

Introduction

This tutorial is mainly written for running on a Ubuntu desktop Linux machine. It is 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.

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.

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.

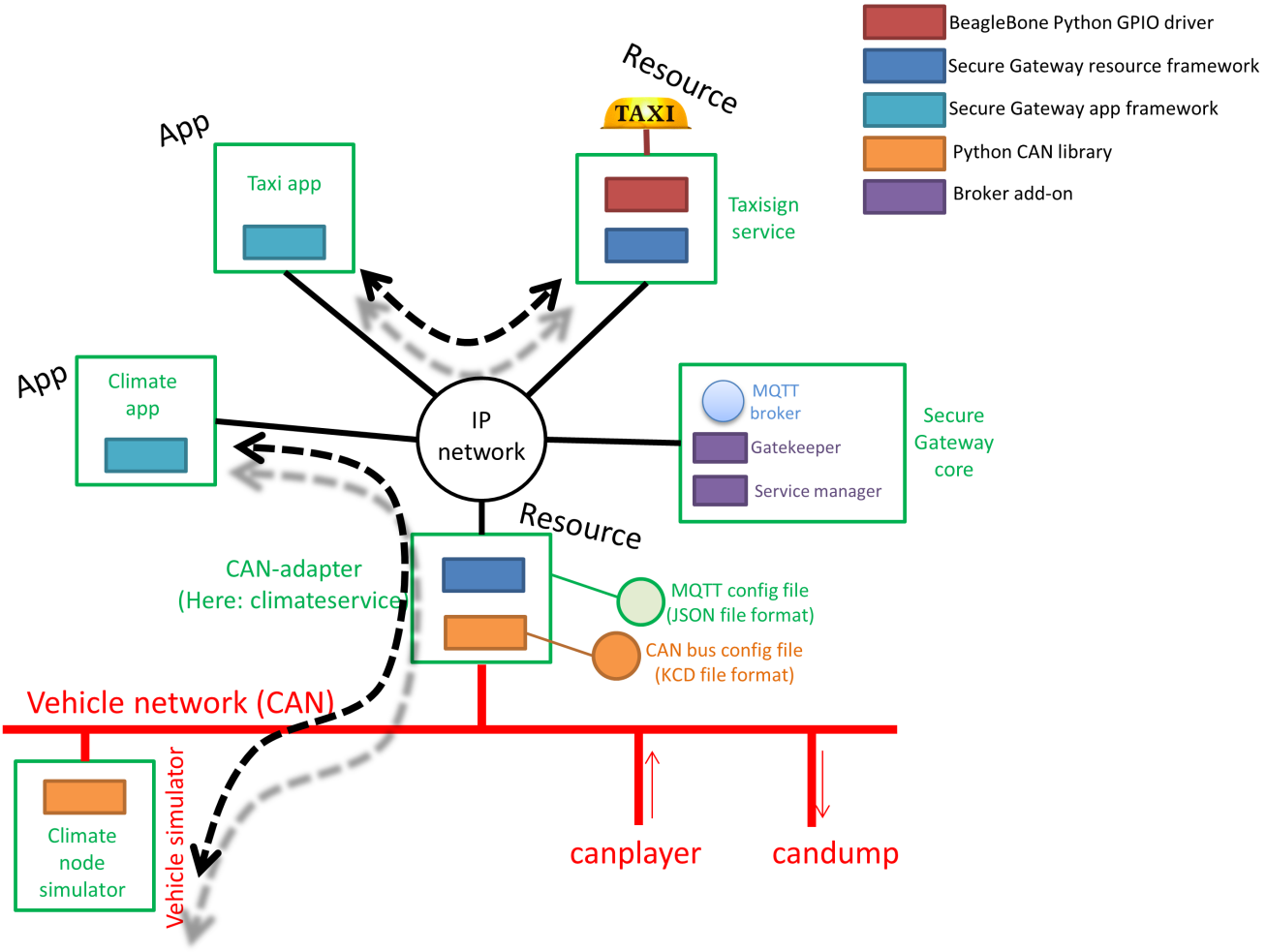
Secure Gateway is a concept for an Internet Protocol (IP) based in-vehicle infotainment network, with a secure connection to the safety-critical vehicle network. This allows the user to install infotainment apps on the IHU, without compromising the vehicle safety. The concept is also useful for example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

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 a SG app. Note that both resources and apps both sends and receives MQTT messages.

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.
- 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.

The image below gives an overview of the Secure Gateway components, and the distributed examples.



The MQTT protocol uses a publish-subscribe pattern. A MQTT broker (server) acts as the central communication point. Clients registers 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

Using plain MQTT

In order to be able to try this, you need to install the Mosquitto broker and the Mosquitto command line tools. See the section in the end of this tutorial 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
+ 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 start and stop it then.

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" 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.

Secure Gateway core (security disabled)

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 send to a resource (from an app or from another resource)

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

- dataavailable: Indicate that certain data is available
- commandavailable: Indicate that a certain command is available

So if a sensor publish 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 connect.

(There is also the resourceavailable messagetype, see below).

The signalnames should be unique among commands and data. The command typically echoed back as data, why there not can be some other data topic having the same signalname as a command.

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

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). In the previous example the last will should 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):

```
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 (Note: Python2):

```
SecureGateway/scripts$ python servicemanager.py localhost
```

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
```

Taxi sign resource

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!

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

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

```
python3 taxisignservice.py -h
```

The taxi sign service listens to this command:

```
command/taxisignservice/signstatus True
```

and it will respond on the corresponding data topic hierarchy.

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

```
mosquitto_sub -t +/#  -v
python3 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 be sent from the broker. The service manager will send out presence information for individual signals.

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

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 taxisignapp.py -v -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
```

Listen to all MQTT commands, and press the buttons on the taxi sign app:

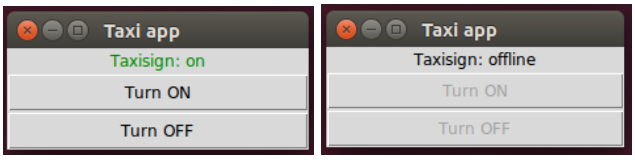
```
mosquitto_sub -t +/#  -v
```

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 taxisignapp.py -v -mode graphical
python3 taxisignservice.py -v -mode graphical
```

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

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 embedded Linux computer board. Use the command:

```
python3 taxisignservice.py -mode hardware -host BROKER_IP_NUMBER
```

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.

Secure Gateway core (with security)

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 (Certificat 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.

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.

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_certificate.pem
```

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_request.csr
openssl req -new -key server_private_key.pem -subj "/C=SE/O=TEST/CN=localhost"   -out server_request.csr
```

The CA issues a server certificate:

```
openssl x509 -req -in server_request.csr -CA ca_public_certificate.pem -CAkey ca_private_key.pem -days 3650 -CAcreates
```

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

Now the .csr file can be removed.

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_request.csr
```

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 -days 3650 -CAseria
```

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.csr
```

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 -days 3650 -CAserial
```

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
```

Testing the certificates from command line

For the clients, put the key+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 --cafile ca_public_c
SecureGateway/examples/taxisignservice/certificates$ mosquitto_sub -v -t +/# -h localhost -p 8883 --cafile ca_public_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 --cafile ca_public
SecureGateway/examples/taxisignservice/certificates$ mosquitto_sub -v -t +/# -h 192.168.0.3 -p 8883 --cafile ca_public
```

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 public_certificate.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.")

Run the taxisignservice and taxisign app using certificates

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

```
SecureGateway/examples/taxisignapp$     python3 taxisignapp.py     -mode graphical -host localhost -port 8883 -cert ce
SecureGateway/examples/taxisignservice$ python3 taxisignservice.py -mode graphical -host localhost -port 8883 -cert ce
```

All distributed example apps and resources can use certificates.

Introduction to access control lists

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

Authorization is about defining which client should be allowed to 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 is 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 defined above enabled. 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 the two usernames swapped.

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

Dynamically change access to applications

TODO

Dynamically change access to application

CAN communication, 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 in the end of this tutorial, and name it vcan0.

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. See below for installation description.

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.

Simulated climate node (CAN 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 on 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).

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

```
python3 vehiclesimulator.py -v
candump vcan0
```

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 changes in the simulated in-car temperature.

Canadapter between CAN and MQTT

The CAN-adapter convert CAN signals to MQTT messages, and MQTT messages to CAN signals. All MQTT communication is handled by the MQTT broker, which controls the access. 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

These MQTT topics are used by the Can-adapter:

```
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 data on the vehiclespeed, enginespeed etc, and listens to 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:

```
python3 vehiclesimulator.py -v
python3 canadapter.py ../examples/configfilesForCanadapter/climateservice_cansignals.kcd -mqttfile ../examples/configt
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
```

Note that for this specific signal, the command is echoed back 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 from the vehiclesimulator in a separate CAN message, and convert that information to the mentioned MQTT message. This can easily 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 limited CAN definition files as possible (in the KCD file format), as CAN message filtering in the Linux kernel is used for the messages defined in the file.

TODO: KERNEL FILTERING NOT YET IMPLEMENTED!

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 climateapp.py -mode graphical
```

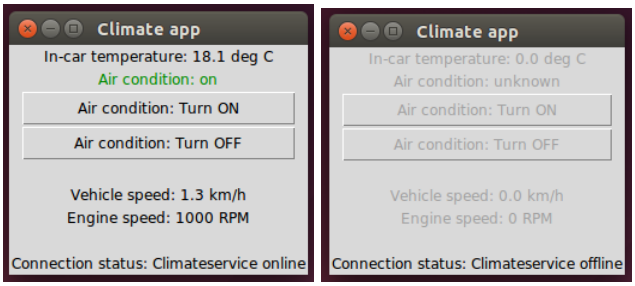
To test the climate app standalone, with a broker only:

```
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 True

mosquitto_sub -t +/#  -v
```

Then, 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 shown.

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



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

CAN communication, using real CAN bus

TODO!

Setting up CAN communication between two Embedded Linux boards

- Raspberry Pi
- Beaglebone
- CAN interface boards
- 500 kbit/s

Test communication using command line tools

- Install CAN adapter software for Secure Gateway.
- Adjust configuration

Test run

- CAN player on other node
- Simulated climate node on other node

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 searches for this information.

Description

installation

Command line usage

Taxi sign app with startup script for node discovery.

Developing your own Resources

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

The main object is the `Resource()`, and it has these public methods:

- register_command()
- register_data()
- start()
- stop()
- loop()
- send_data()
- possibly use register_inputsignal()

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 resource can be implemented in any programming language having a proper MQTT library with TLS support.

Developing your own Apps

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

The main object is the `App()`, and it has these public methods:

- register_inputsignal()
- send_command()
- start()
- stop()
- loop()

The taxisign app source code should be used for inspiration.

Making apps for Android

TODO!

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

Dependency installation details

General

In order to be able to use apt-get installation tools, make sure that you have proper internet connection from your (embedded) Linux machine. Test by:

```
ping www.google.com
```

If it not is working, verify that the DNS settings are OK.

In general, you need the 'make' command. Install it with:

```
sudo apt-get install make
```

Mosquitto broker

The broker is described here; http://mosquitto.org/

How to install a recent version of Mosquitto is described on http://mosquitto.org/download/

Minimum recommended version: 1.4.1

```
sudo apt-get remove mosquitto
sudo apt-get install -y libc-ares-dev
wget http://mosquitto.org/files/source/mosquitto-1.4.1.tar.gz
(see deploy script) <-- ###################################################################
```

This is installing an old version on Linux machines:

```
sudo apt-get install mosquitto
```

The broker will get started automatically, listening on port 1883.

To start and stop the broker, use:

```
sudo service mosquitto start
sudo service mosquitto stop
```

The configuration file for the Mosqitto broker is typically located at:

```
/etc/mosquitto/mosquitto.conf
```

Mosquitto command line tools

The tools mosquitto_pub and mosquitto_sub are very useful.

To install it on an Ubunto Linux machine:

```
sudo apt-get install mosquitto-clients
```

Enabling a virtual CAN interface

To create a virtual (simulated) CAN bus named vcan0 on Desktop Ubuntu:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ifconfig vcan0 up
```

Installing Linux CAN command line tools (can-utils)

To install can-utils:

```
sudo apt-get install can-utils
```

or:

```
sudo apt-get install git make
git clone https://github.com/linux-can/can-utils.git
cd can-utils/
make
sudo make install
```

The source code is found on https://github.com/linux-can/can-utils

Some usage ideas:

- http://fabiobaltieri.com/2013/07/23/hacking-into-a-vehicle-can-bus-toyothack-and-socketcan/
- http://www.cowfishstudios.com/blog/canned-pi-part1

Python CAN library

The CAN library is distributed with Secure Gateway.

It reads a CAN definition file in the KCD file format. Use .get_descriptive_ascii_art() to print out the location of the signals in the messages.

```
canbus = CanBus(interfacename, timeout, filename='CANdefinition.kcd')
print(canbus.get_descriptive_ascii_art())
```

Here is an example output, where the most significant and least significant bits are indicated with 'M' and 'L' respectively. The upper row of numbers is the 'normal' bit numbering, and the lower row of numbers is the 'backward' bit numbering.

```
CAN bus 'Mainbus' on CAN interface: vcan0, having 3 messageIDs defined.
    Messages:

    CAN message definition ID: 7 (0x007), name: climatecontrolsignals, DLC: 8, contains 1 signals
        Signal details:
        ---------------


        Signal 'acstatus' Startbit 7, bits 1 (min DLC 1) big endian, scalingfactor 1, unit:
             valoffset(smallest) 0.0 (max of range 1.0) min None, max None, default 0.0.

             Startbit normal bit numbering, least significant bit: 7
             Startbit normal bit numbering, most significant bit: 7
             Startbit backward bit numbering, least significant bit: 63

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

    CAN message definition ID: 8 (0x008), name: vehiclesimulationdata, DLC: 8, contains 2 signals
        Signal details:
        ---------------


        Signal 'vehiclespeed' Startbit 8, bits 16 (min DLC 2) big endian, scalingfactor 0.01, unit:
             valoffset(smallest) 0.0 (max of range 655.4) 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

(etc)
```

Installing Wireshark on Ubuntu

This will give an old version:

```
sudo apt-get install wireshark
```

It is better to download the sourcecode, and compile it as described on
https://www.wireshark.org/docs/wsug_html_chunked/ChBuildInstallUnixBuild.html

You might need to run this the download its dependencies:

```
sudo apt-get install bison flex libqt4-core qt4-dev-tools libgtk-3-dev libpcap-dev
```

It seems that the resulting program ends up in the source directory. How to change this?

In the source directory, run:

```
./wireshark
```

Installing Avahi

```
sudo apt-get install -y avahi-daemon avahi-discover libnss-mdns avahi-utils uuid-dev
```

Installing Python 3.3

Desktop Ubuntu

Raspbian

BeagleBone?

Installing Python3 TK graphics library

```
sudo apt-get install python3-tk
```

Create a root account on Raspberry Pi (Raspbian)

By default you can not log in as root.

To create a root password:

```
sudo password root
```

CAN controller 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 a 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 MCP25215 chip uses a crystal oscillator, most often 10 MHz or 16 MHz. The crystal frequency is entered when installing the kernel modules for the CAN controller.

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

To use the CAN bus on Raspberry Pi, you need some compiled kernel modules (rpi-can). Those modules seems to be available for Raspbian 3.12.28+, with a date of 2014-09-09.

- Download that version of Raspbian from http://downloads.raspberrypi.org/raspbian/images/
- Download the pre-compiled CAN kernel modules, and install it according to this site: http://www.cowfishstudios.com/blog/canned-pi-part1

Explanation of spi-config usage: https://github.com/msperl/spi-config

To enable the CAN interface, and set the bitrate:

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

To disable, and re-enable the CAN interface:

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

Sometimes it seems better to run these commands as superuser (su) instead of using sudo. Unclear why.

CAN controller on Beaglebone

There are two CAN controllers (dcan0 and dcan1) in the Beaglebone processor. One of them (dcan0) share pins with EEPROMS on the capes (expansion boards), so it most often not possible to use.

Beaglebone pin	Signal	Linux pin	GPIO number	Device tree offset number
P9.19	dcan0_rx	31	GPIO0[13]	0x17C
P9.20	dcan0_tx	94	GPIO0[12]	0x178
P9.24	dcan1_rx	97	GPIO0[15]	0x184
P9.26	dcan1_tx	96	GPIO0[14]	0x180

- Pin voltage 3.3 V
- dcan0 pins are typically used by the capes I2C communication
- The device tree offset number is calculated from the pinmux base address 0x44e10800.

There are CAN expansion boards (capes) for Beaglebone, for example this product from Towertech: http://www.towertech.it/en/products/hardware/tt3201-can-cape/

This cape has 3 CAN channels, of which one uses a CAN controller in the CPU and the other two channels use external MCP2515 chips. The dip-switch S1 allows enabling 120 Ohm termination resistors for each of the CAN channels. The resistor is enabled when the corresponding switch is in "on" position. A table of the different connection pins and corresponding CAN interface names is available in the user manual.

Linux distributions for Beaglebone Black

The Linux distribution by default on the Beaglebone Black NAND memory is Debian with a 3.8 kernel version.

http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

- Username: debian
- Password: temppwd

- Username: root
- Password: (none)

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

To see which Linux version you are running:

```
uname -a
```

To install a Ubuntu image on a SD-card, see http://elinux.org/BeagleBoardUbuntu It uses an image built by Robert C Nelson.

- Username: ubuntu
- Password: temppwd

- Username: root
- Password: root

The Debian and Ubuntu images handle the CAN hardware slightly differently. You need to enable the CAN hardware in the processor, and connect it to the physical pin of the processor (pin muxing). How this is done for the different distributions is described below.

Enabling CAN interface on Beaglebone Black running Debian

In the Debian distribution, the dcan1 CAN controller hardware in the processor will be named can0.

The pin muxing using Debian for dcan1 is described here: http://www.embedded-things.com/bbb/enable-canbus-on-the-beaglebone-black/

This is necessary for DCAN1 to work. When inside the code there is a code. Please copy the code and put it in a file inside the beaglebone. The file name is displayed in the top of the code, give the file the same name. please continue with the guide on the web page.

If you encounter the error "Failed to load firmware", please restart the beaglebone and try again.

To see whether dcan1 is enabled:

```
cat /proc/device-tree/ocp/d_can\@481d0000/status
```

You must do this as root (sudo does not work):

```
echo BB-DCAN1 > /sys/devices/bone_capemgr.*/slots
```

Start up the can0 interface in Debian by running this as normal user:

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

Enabling CAN interface on Beaglebone Black running Ubuntu

Robert C Nelsons Ubuntu

In the Ubuntu distribution, the dcan1 CAN controller hardware in the processor will be named can1.

The pin muxing is done like this:

```
su
echo can > /sys/devices/ocp.3/P9_24_pinmux.51/state
echo can > /sys/devices/ocp.3/P9_26_pinmux.53/state
exit
```

This will enable the can1 interface.

To list the current pin mux settings, run this as su:

```
cat /sys/kernel/debug/pinctrl/44e10800.pinmux/pinmux-pins
```

To start up the CAN interface, as a normal user run:

```
sudo ip link set can1 up type can bitrate 500000
```

Test the CAN interface on Beaglebone Black

Test sending a CAN message:

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

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

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
```

Installing Paho Python MQTT client library

https://eclipse.org/paho/clients/python/

```
sudo apt-get install python3-setuptools
sudo easy_install3 pip
sudo apt-get install python3-pip
sudo pip3 install paho-mqtt
```

Autostart taxisign resource on Unbuntu

TODO: Change address, and explain startupscript!

To automatically run the startscript at power up and to have plug and play feature enabled, you need to put the following code to you "/etc/rc.local" file:

```
cd /home/ubuntu/SG/tutorial/taxisign; ## Could be different in your case
sudo ./start
```

Then, next time when you connect the taxi sign box to the Secure Gateway network and power it on, it will automatically discover the SG IP and connect to it.