Horizon 2020 PROGRAMME *ICT-01-2014: Smart Cyber-Physical Systems*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 643924

D3.1

Desktop middleware API reference documentation

1. DOCUMENT INFORMATION

Deliverable Number	D3.1
Deliverable Name	Desktop middleware API reference documentation
Authors	O. Deniz(UCLM), J.L. Espinosa-Aranda(UCLM), N. Vallez(UCLM)
Responsible Author	O. Deniz (UCLM) e-mail: Oscar.Deniz@uclm.es phone: +34 926 29 53 00 ext: 6286
Keywords	MQTT, broker, API, Java
WP	WP3
Nature	R
Dissemination Level	PU
Planned Date	31.01.2016
Final Version Date	28.01.2016
Reviewed by	O. Deniz (UCLM), A. Pagani (DFKI)
Verified by	C. Fedorczak (TCS)

2. DOCUMENT HISTORY

Person	Date	Comment	Version
J.L. Espinosa-Aranda	08.01.2016	Initial version	0.1
J.L. Espinosa-Aranda	19.01.2016	Sections 6, 7 and 10	0.2
N. Vallez	19.01.2016	Section 8	0.3
J.L Espinosa-Aranda and N. Vallez	20.01.2016	Sections 5 and 9	0.4
C. Fedorczak	28/01/2016	Verification	

3. ABSTRACT

The Eyes of Things (EoT) project envisages a computer vision platform that can be used both standalone and embedded into more complex artefacts, particularly for wearable applications, robotics, home products, surveillance etc. The core hardware will be based on a number of technologies and components that have been designed for maximum performance of the always-demanding vision applications while keeping the lowest energy consumption.

An important functionality is to be able to communicate with other devices that we use every day. In EoT, a middleware is developed to allow configuration and basic control of the device from an external computer like a desktop/laptop PC or a tablet/smartphone. The wireless communication on which this middleware is based is additional to the existing wired debug capability of the Myriad SoC.

Apart from low-power hardware components, an efficient wireless communication protocol is necessary. Text-oriented protocols like HTTP are not appropriate in this context. Instead, the lightweight publish/subscribe message-based MQTT protocol was selected. With MQTT, the typical scenario is that of a device that sends/receives messages, the messages being forwarded by a cloud-based message broker. In the EoT project we propose a novel approach in which each EoT device acts as an MQTT broker instead of the typical cloud-based architecture. This eliminates the need for an external Internet server, which not only makes the whole deployment more affordable and simpler but also more secure by default.

This document describes the desktop middleware API implemented, which includes:

1. The WiFi module used.
2. Bases of the MQTT protocol and the approach followed.
3. Pulga, a tiny open-source MQTT broker for flexible and secure IoT deployments.
4. The MQTT client application developed in Java for controlling the EoT device.

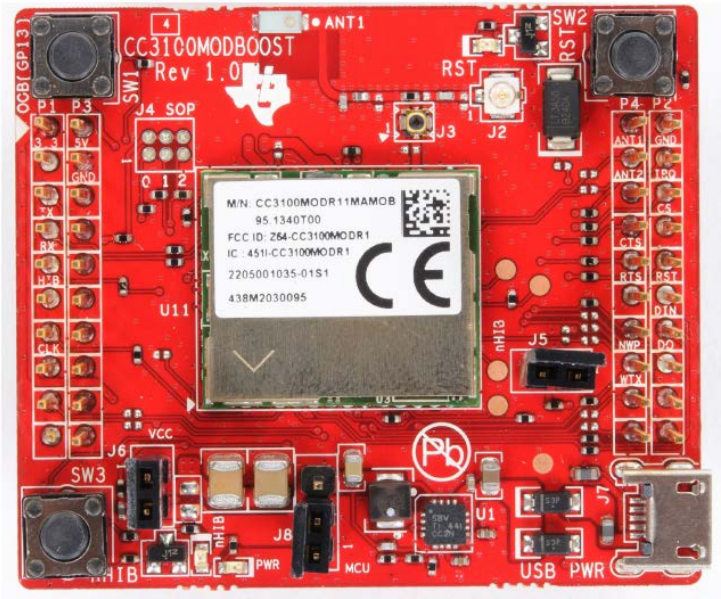
These elements work together to give the EoT a minimal control mode, allowing external access from a desktop/laptop computer.

4. TABLE OF CONTENTS

1. Document Information 2
2. Document History 3
3. Abstract 4
4. Table of Contents 5
5. WiFi module 6
6. MQTT Protocol and Approach 7
6.1. Approach 8
7. Pulga, a tiny open-source MQTT broker for flexible and secure IoT deployments 12
7.1. API (MQTT Restricted Topics) 14
7.2. Problems Found/Known Issues 22
7.3. Test Cases 22
7.3.1. Tests implemented 22
7.3.2. Expected output of the tests 23
7.4. Using the Application/Screenshots 25
7.5. To Do 32
8. MQTT Client Application Developed in Java (JAVA API) 34
8.1. Paho Java Client 34
8.2. API Specification 35
8.3. User Interface/Use of the Application 43
8.1. Problems Found/Known Issues 49
8.2. To Do 49
9. Code 50
10. Conclusion 51
11. Annex: Python tests 52
12. References 57
13. Glossary 58

5. WIFI MODULE

The EoT device configuration (‘control mode’) is performed through a wireless connection. The device incorporates a WiFi module (CC3100MOD from Texas Instruments [1]). Since the CC3100 module allows the creation of an ad-hoc WiFi, the connection with the external configuration device can be done even without an existing WiFi infrastructure. Therefore, a computer, a mobile phone, or a tablet can establish a connection with the EoT device.



The W*ifiFunctions* module has been developed over the CC3100 driver (provided by Movidius) in order to provide a convenient wrapper of common WiFi functions. This module provides a layer of functions for creating an ad-hoc WiFi, establishing a connection with another device, sending/receiving data and closing a connection. In addition, functions for managing WiFi connection profiles have been also included. The SSID and the password of the ad-hoc WiFi or the local WiFi infrastructure can be stored in the flash memory of the CC3100 as a connection profile.

6. MQTT PROTOCOL AND APPROACH

In the last few years cognitive applications and services have been acknowledged as key drivers of innovation and demand. The particular case of computer vision represents a fundamental challenge. While image analysis and inference requires massive computing power, the sheer volume of visual information that can be potentially generated by mobile devices cannot be transferred to the cloud for processing. This problem becomes even worse when we consider emerging sensing technologies like 3D and hyperspectral cameras. One of the most recent attempts at alleviating this problem is the cloudlets approach [2], which essentially proposes offloading computation to local computers that are within one wireless hop of the mobile device. These computers would play a role similar to those in data centers. In particular, streams of image data would be processed and analyzed in those computers, providing relatively fast responses to the mobile device.

While the cloudlets approach is certainly an efficient way to manage increasing demands of computing power, it falls short in a number of aspects. First, streaming of raw sensor data out of the mobile device is still being assumed. Power efficiency then becomes a major issue, since wirelessly transmitting data for remote computation can cost up to one million times more energy per operation compared to processing locally in a device. Second, it also assumes that the end-user will have to purchase and manage the local computer. Another scenario is when this computer is part of some service provided in the premises (say, within a Hospital), but then the problem becomes one of security, for raw sensor data would be streamed to an externally-managed device. The philosophy behind EoT is precisely focused on maximizing the mobile device's processing power vs energy consumption ratio as well as ensuring secure use by individual users.

Within the overall aim of optimizing energy consumption, an important component of the EoT device is the low-power WiFi chip. The specific model selected for EoT is the CC3100 from Texas Instruments, which provides basic TCP/IP communication. A firmware in the device allows sending/receiving of images, metadata and control/config commands. Apart from hardware, an efficient communication layer protocol is necessary. This protocol shall be used for sending the results of computer vision processing, including text, images or other types of data. HTTP is widely used, but its text-oriented nature is not appropriate for resource-limited devices. Instead, the MQTT protocol was selected early on [3]. MQTT is a lightweight publish/subscribe protocol designed for use over TCP/IP networks which provides an efficient 1-to-n communication mechanism. MQTT has been designed for low bandwidth and unreliable or intermittent connections, thus being a strong candidate in the Internet of Things scenario. MQTT-enabled devices can open a connection and keep it open using very little power.

The typical scenario is that of an embedded client which connects to an MQTT server (message broker) in the cloud [4] (Figure 1).

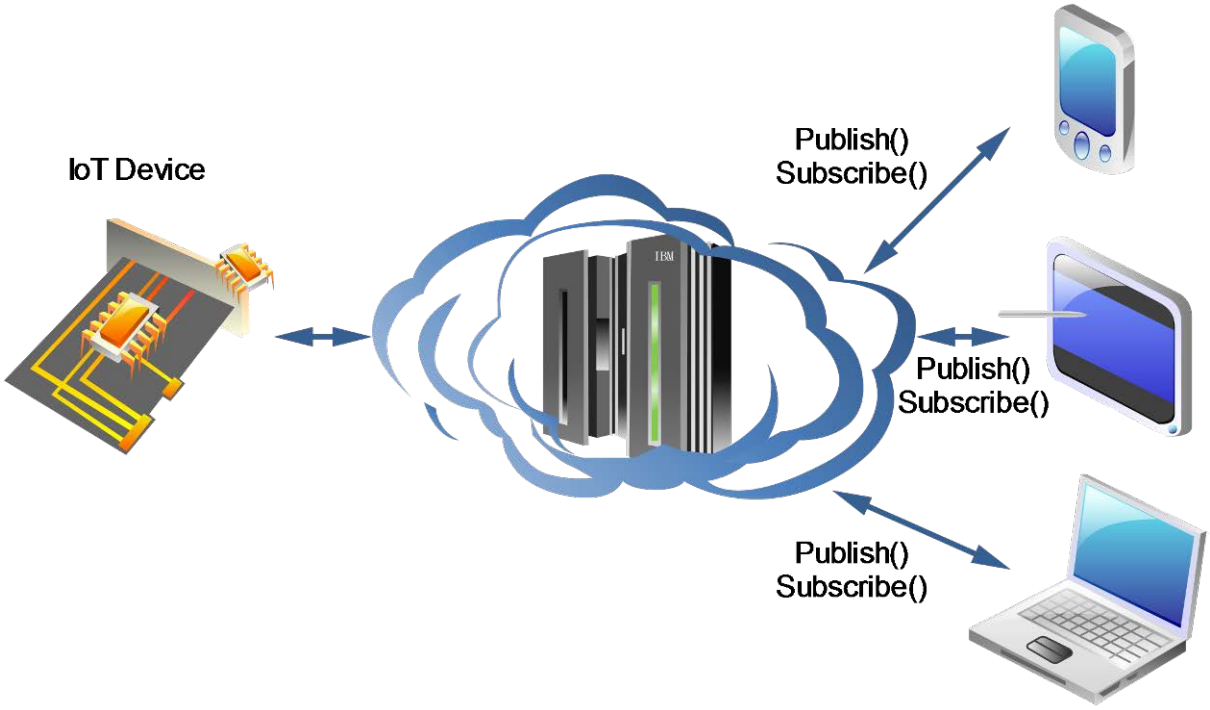


Fig. 1: Typical IoT MQTT communication model

Again, in this scenario the brokering service has to be purchased by the user, or else it has to be installed on a locally-managed server. In the EoT project a novel architecture in which each EoT device can act as a broker is proposed. This way no external server will be required, and data will not be initially sent through the Internet. In fact, the configuration device (another device as smartphone, tablet or PC) and the EoT node do not need to be in a WiFi network infrastructure, since an ad hoc network is created by the EoT device by default. This allows setting up applications in which only the EoT and another device are involved. That is in fact the default mode upon boot, with the additional possibility of connecting to an existing WiFi network. As a result, depending on the final application an EoT device can be configured to work at 3 levels: 1) Single device mode (with the only requirement of a configuration device, typically a smartphone, tablet or laptop, connecting to the EoT-generated ad-hoc WiFi), 2) Home, i.e. EoT device connecting to a local WiFi infrastructure, and 3) EoT device connecting to the cloud (through the WiFi infrastructure).

Approach

By default, EoT devices create an ad-hoc WiFi. This is necessary to allow connection to the configuration device even without WiFi infrastructure. Note also that the EoT device cannot by itself connect to an existing WiFi since it does not have means for specifying SSIDs or network passwords. Each EoT device creates a WiFi with univocal SSID and password. The configuration device will have to enter those to establish communication with it. This method allows a configuration device to connect and configure devices one by one. During configuration, an EoT device can be also made to connect to a given existing WiFi (either from other EoT or from infrastructure). This allows EoTs to connect to each other (with or without infrastructure) and also allows EoTs to connect to the

Internet. Low-level security is handled by an encryption protocol used in the ad-hoc WiFi (typically WPA). Horizontal arrows in Figure 2 represent data communication to/from EoT devices to/from a) Desktop computers and mobile devices such as smartphones and tablets and b) Cloud services.

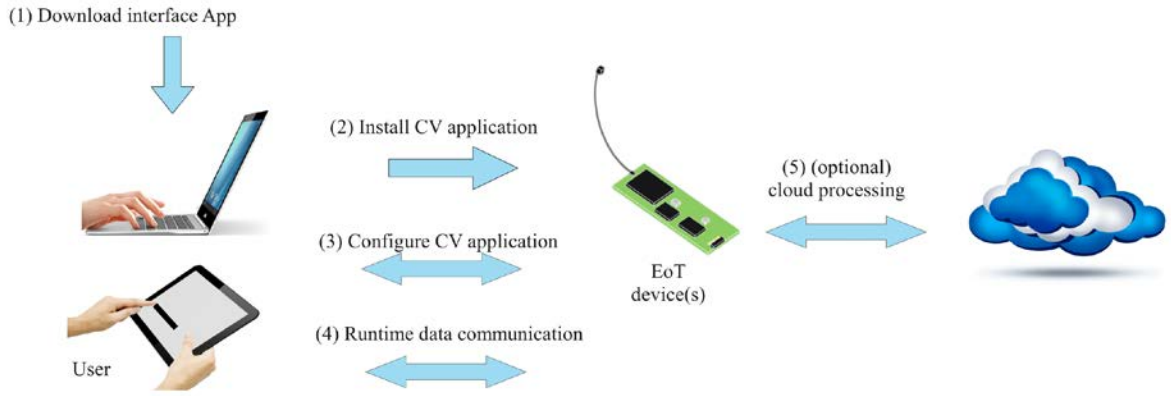


Fig. 2: User view

All Internet of Things implementations must consider low-powered devices which need to function for months or years without getting any power recharge. This makes the as-is use of some of the existing Internet protocols to be sub-optimal. Some protocols that are heavily used in Internet add substantial overheads and a large number of device-to-cloud network technologies and protocols are being developed by researchers and start-ups. This has led to an enormous fragmentation, described in [5].

Currently efficient options exist for low-power connectivity such as Zigbee and the more recent Bluetooth LE (low-energy). The proposed approach will use TCP/IP over WiFi, since low-power features present in the latest WiFi modules prepared for the Internet of Things will be leveraged, having low-power standby modes and short wake-up times.

As mentioned above, the proposed EoT device approach uses the TCP/IP stack and the MQTT protocol [3] for communicating with other devices. MQTT-enabled devices can open a connection, keep it open using very little power and receive events or commands with as little as 2 bytes of overhead. While HTTPS is slightly more efficient in terms of establishing connection, MQTT is much more efficient during transmission.

MQTT v3.1.1 has become recently an OASIS Standard [6]. One of the key aspects of MQTT is an extremely efficient and scalable data distribution model. While HTTP is point-to-point, MQTT can distribute 1-1 or 1-to-n via the publish/subscribe mechanism. For these reasons MQTT is being increasingly used in mobile Apps (it is notably used by Facebook Messenger) instead of existing unreliable push notification mechanisms, see Figure 3. Devices can publish data on a “topic”. Other devices can subscribe to a given topic and they will receive the corresponding published data. The broker is typically hosted on an enterprise server.

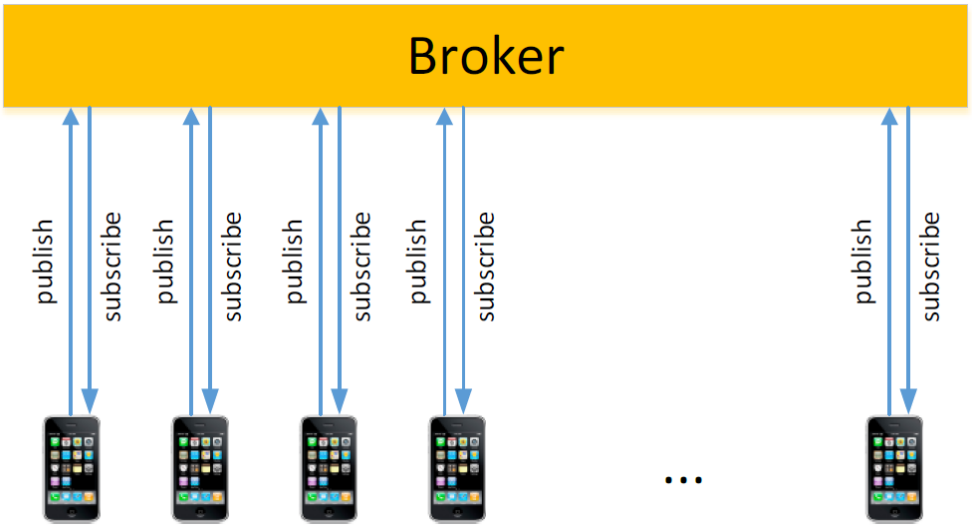


Fig. 3: MQTT publish/subscribe

MQTT is at the time of writing one of the strongest contenders in the IoT and M2M “protocol wars”. MQTT has been selected in the EoT project for two reasons: a) it is a low-power protocol and b) it provides an efficient 1-to-n communication mechanism. 1-to-n communication is fundamental since it allows multiple viewers, multiple (additional) processors and cooperation between sensors. The typical MQTT scenario would need a broker in the cloud. Alternatively, the smartphone/tablet/PC used for configuration could be used as a broker, but this would mean that such device (typically our personal smartphone or tablet) would have to be continuously functioning as a gateway. Thus, an architecture in which each EoT device can act as a broker is proposed. This way other EoT devices can subscribe to a given EoT. The configuration device can subscribe to the device and receive data too, although it can also 'publish' configuration commands for the device. In this architecture each EoT device can effectively act as both client and server, and the configuration device will only be woken up by the appropriate EoT (see Figure 4).

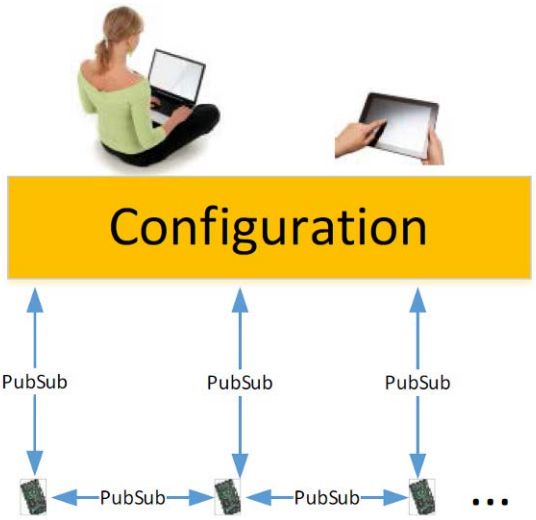


Fig. 4: Proposed EoT MQTT architecture

Apart from efficient peer-to-peer cooperation, this model allows building hierarchical processing networks. An example is extracting salient features on a first level and recognition on another. The first level of EoTs would compute salient features. The second level would be subscribed to their results and would perform object recognition. Finally, the configuration device would be subscribed to second-tier devices to get only the final result. Another possible hierarchy is object detection, followed by object recognition and object tracking. Finally, neural network-based techniques could be staged using multiple EoT devices, both in parallel and sequentially. Note also that the huge fan-out capabilities of MQTT also make it ideal for massively parallel processing.

7. PULGA, A TINY OPEN-SOURCE MQTT BROKER FOR FLEXIBLE AND SECURE IOT DEPLOYMENTS

Pulga, which means flea in Spanish, is the proposed tiny MQTT broker implementation for EoT devices. Its name derives from Mosquitto [7], which is a widely-used Open Source MQTT v3.1 message broker written by Roger Light. As opposed to Mosquitto, Pulga is a lightweight broker designed to be run in embedded systems. While Mosquitto requires at least 3MB RAM, Pulga has been tested using only 512KB and probably can still run using less memory. Figure 5 shows an example of the typical configuration that the proposed approach will have.

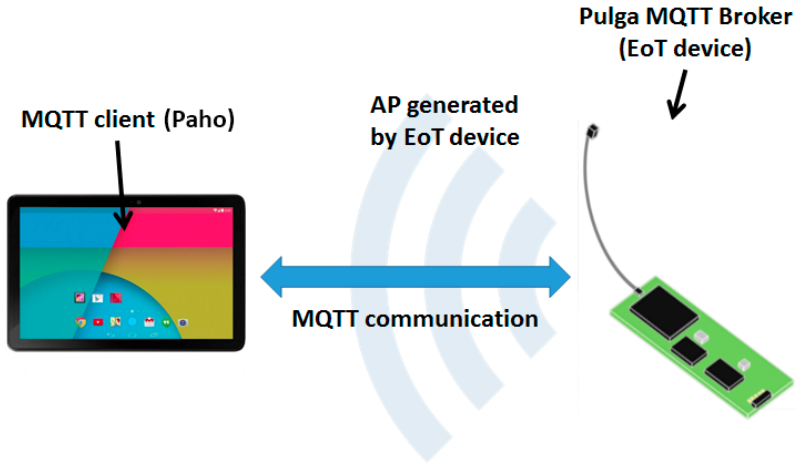


Fig. 5: Pulga broker configuration

To develop Pulga it was necessary to start from the ground-up implementing only the minimal parts of the protocol. It uses the MQTTPacket library of the Eclipse Paho MQTT C/C++ client library for embedded platforms [8]. This library contains the lowest level C library which supplies simple serialization/deserialization routines depending on the type of message sent/received.

In particular, Pulga has the following MQTT protocol functionality implemented:

1) Manage connection/disconnection of a client.
2) Publish message.
3) Multiple clients can connect to Pulga and subscribe/unsubscribe to topics.
4) Topics/subscriptions management. Currently Pulga does not consider hierarchical topics (subtopics), only basic topics.
5) Keep alive functionality through the ping request.

With respect to other typical features of a MQTT broker as defined by the protocol, there are some of them that are not implemented because of the main focus of Pulga broker. More specifically Pulga does not manage retained messages, the definition of a session as clean or durable, or the “last will” option,

which allows a client to send a message that it wishes the broker to forward when it disconnects unexpectedly.

Moreover MQTT defines three levels of Quality of Service (QoS). The QoS defines how hard the broker/client will try to ensure that a message is received. There are three levels of QoS defined by the MQTT protocol, 0: The broker/client will deliver the message once, with no confirmation, 1: The broker/client will deliver the message at least once, with confirmation required, and 2: The broker/client will deliver the message exactly once by using a four step handshake. The current Quality of Service (QoS) implemented in Pulga is the QoS 0.

As used in EoT, Pulga includes other functionality that a typical MQTT broker does not have. The most important is that it adds the possibility of defining 'restricted' topics. This option allows to include new uses for the MQTT broker. When a Publish message is received, Pulga first detects if the topic is a 'restricted' one using a simple parser on the broker, changing the typical publish behaviour as the programmer defines. This approach allows the user to have additional functionality:

- Send/receive files. Considering that the MQTT protocol sends binary messages (the text of an MQTT message is always serialized) it is possible to use an MQTT message to send binary data without serializing/deserializing it. This process would allow us to upload the applications we want to run into the device or send the captured images in the EoT device to a client that requests them.
- Configuration of the EoT device. By defining different configuration topics, Pulga is able to manage the different EoT parameters and to configure the WiFi connection of the device. A simple parser is used to manage the messages published.
- Configuration of access to a WiFi infrastructure. The configuration device can send the required connection data to the EoT device. This data can be stored in the device and used by preloaded applications to connect to an existing WiFi.

Concretely, the control mode, which will be activated through a dipswitch, will receive and respond to this control commands:

a. Establish wireless communication.
b. Change from AP mode to Station Mode and connect to another existing AP.
c. Upload and flash an application (a payload).
d. Upload data to SD card in EoT.
e. Run application (payload).
f. Download data from SD card in EoT.
g. Change network password.
h. Remove network password.
i. Request a camera snapshot.
j. Update Myriad's clock/time with the client's clock/time (clock/time is strictly needed, for example to store timestamps on events).

It is worth noting that these commands depend on other modules which will be described in Deliverable 3.3 Firmware documentation.

As a way to test communications, the interpreter will be on when in AP mode (and off when in station mode), and flashing upon receiving every command. The SD Card will be used to store application data (audio files, cascade classifier data…), configuration parameters, snapshots, application results, etc.

Pulga will be always resident in the device Flash. The application to run will be stored also in the Flash. If for some reason the payload does not work properly then the device will have to be booted in Control mode to flash another payload. If for some reason the module stops working then the device will have to be connected to a PC (with the JTAG cable) and Pulga will have to be reflashed.

[redacted]API (MQTT Restricted Topics)

When a message is received by Pulga in one these topics, the behavior of the broker changes from a typical MQTT broker as explained in Table 1.

The client must subscribe to each reserved topic before sending the first message. The unsubscribing is managed afterwards automatically by the broker after finishing the command, so it is not necessary for the client to send the unsubscribe command.

Some of the reserved topic must answer to the client if the operation has been performed correctly in the same reserved topic. In case that an error occurs, Pulga will send “-1” to the client. Otherwise, it will send “0”.

The package described corresponds to a packet of information with a maximum size of 1024 bytes. This fact is because of the limitation imposed by the CC3100 WiFi device when receiving a message through sockets.

Basic Description	Topic	Publish Message	Description
Upload file to SD	EOTUploadFileSD	NumberOfPackages PathToFile	Parameters: NumberOfPackages: Number of packages to be sent. PathToFile: Path to the file to be written. Functionality: The client application must send the first message including the defined information. After that the client must send the file divided in parts of maximum 1024 bytes using publish messages with the same topic (EOTUploadFile). The message for each part of the file must be: PartOfFIleXXXXXX Where XXXXXX is the number of the package sent using 6 digits (it must include zeros at the left). The complete size of the message would be 1024+6 at maximum. Answer to the client: The EoT device must send a message to the client in the same topic indicating if the file has been uploaded correctly or an error.
Create directory in SD	EOTMakeDirSD	PathToDir	Parameters: PathToDir: Path to the directory to be created. Functionality: Create the indicated directory in the SD card.

			Answer to the client: The EoT device must send a message to the client in the same topic indicating that the directory has been created in the SD card.
List files from SD	EOTListFilesSD	PathToDir	Parameters: PathToDir: Path to the directory to be listed. Functionality: Send a list with the name of the files on a folder of the SD. Answer to the client: The EoT device must send a message to the client in the same topic indicating the files of the device SD.
Download file from SD	EOTDownloadFileSD	PathToFile	Parameters: PathToFile: Path to the file to be downloaded. Functionality: Send the indicated file to the client. Answer to the client: First the EoT device must send a message in the same topic as follows: NumberOfPackages This message indicates the number of packages to be received by the client. The size of each package is at maximum 1024 bytes. After that the EoT device must send NumberOfPackages messages partitioning the file

			using the same topic. The client should combine these packages to obtain the complete file.
Delete selected file from SD	EOTDeleteFileSD	PathToFile	Parameters: PathToFile: Path to the file to be deleted. Functionality: Delete the indicated file from the SD card. Answer to the client: The EoT device must send a message to the client in the same topic indicating that the selected file of the SD card has been removed or an error.
Delete selected directory from SD	EOTDeleteDirSD	PathToDir	Parameters: PathToDir: Path to the directory to be deleted. Functionality: Delete the indicated directory from the SD card. Answer to the client: The EoT device must send a message to the client in the same topic indicating that the selected directory of the SD card has been removed or an error.
Delete all files from SD	EOTDeleteContentSD	PathToDir	Parameters: PathToDir: Path to the directory which contents will be deleted. Functionality: Delete all the files inside the indicated directory. Answer to the client: The EoT device must send a message to the client in the same topic indicating that the files of the SD

			have been removed or an error.
Upload and flash an application (ELF)	EOTUploadElf	NumberOfPackages AppName	Parameters: NumberOfPackages: Number of packages to be sent. AppName: Name of the app to be written. Functionality: The client application must send the first message including the defined information. After that the client must send the ELF file divided in packages using publish messages with the same topic (EOTUploadElf). The message for each part of the file must be: PartOfFIleXXXXXX Where XXXXXX is the number of the package sent using 6 digits (it must include zeros at the left). The complete size of the message would be 1024+6 at maximum. Answer to the client: The EoT device must send a message to the client in the same topic indicating if the ELF has been uploaded correctly or an error.
List ELF files	EOTListElf	-	Parameters: No parameters Functionality: Send to the client a list with the name of the ELF files uploaded to the device. Answer to the client:

			The EoT device must send a message to the client in the same topic indicating the ELF files of the device.
Request a camera snapshot	EOTSnapshot	-	Parameters: No parameters Functionality: Send to the client a snapshot taken by the camera of the device. Answer to the client: First the EoT device must send a message in the same topic as follows: NumberOfPackages This message indicates the number of packages to be received by the client. After that the EoT device must send NumberOfPackages messages partitioning the image using the same topic. The client should combine these packages to obtain a complete image.
Create access point	EOTCreateAP	Ssid Security [Pass] [Channel]	Parameters: Ssid: SSID of the access point to be created. Security: Security of the access point. The values of this parameter must be Open, WEP or WPA. Pass: Password of the WiFi network if necessary (only on WEP and WPA networks). Channel: Channel in which the access point will be emitting. If not defined, it will be selected among the less congested channels with other access

			points. Functionality: Create the indicated access point. Answer to the client: Since all the connections previously created are reset, it is not possible to send an answer to the client, which must be restarted.
Connect to access point	EOTConnectToAP	Ssid Security [Pass]	Parameters: Ssid: SSID of the network to be connected to. Security: Security of the network. The values of this parameter must be Open, WEP or WPA. Pass: Password of the WiFi network if necessary (only on WEP and WPA networks). Functionality: Connect to the indicated access point (if it exists). Answer to the client: Since all the connections previously created are reset, it is not possible to send an answer to the client, which must be restarted.
Reset WiFi configuration	EOTDisconnectFromAP	-	Parameters: No parameters Functionality: Reset the WiFi configuration and restart to default. Answer to the client: Since all the connections previously created are reset, it is not possible to send an answer to the client, which must be restarted.

Update date	EOTUpdateDate	Year Month Day Hour Min Sec	Parameters: Year: Year to be updated. Month: Month to be updated. Day: Day to be updated. Hour: Hour to be updated. Min: Minute to be updated. Sec: Second to be updated. Functionality: Update the date and time of the device. Answer to the client: The EoT device must send a message to the client in the same topic indicating that the date has been updated correctly.
Get date	EOTGetDate	-	Parameters: No parameters. Functionality: Get the date of the device. Answer to the client: The EoT device must send a message to the client in the same topic indicating the date of the device in the following format: Www Mmm dd hh:mm:ss yyyy Where Www is the weekday, Mmm the month (in letters), dd the day of the month, hh:mm:ss the time, and yyyy the year.

Table 1: Pulga restricted topics

Problems Found/Known Issues

- There is a simple version of Pulga (only MQTT broker) which includes the possibility of sending a snapshot from the camera installed on the initial prototype of the EoT device board. This is not yet implemented in the last version until the NanEye camera is included in the board.

- The CC3100 has a limitation of receiving a maximum of 1472 bytes in a socket. Pulga splits every file/binary it needs to send. We selected a maximum chunk size of 1024 bytes, including at the end 6 bytes which will include the number of the package sent. This has been tested with files and pictures of 4MB at the moment.

- Memory problems: in custom.ldscript, the parameter _RAM_SIZE_LOS must be configured as the maximum size of a file expected to be received. It can be improved in future versions of Pulga by saving the file not at the end of reception, but when each part is received.

- Topic EOTUploadElf is not implemented because in the initial hardware prototype there are known conflicts between the SPI of the WiFi chip and the flash memory. Currently this topic works receiving the file and saving it in a folder called Flash in the SD card.

Test Cases

Several tests have been implemented in Python to test Pulga (desktop\unittest\test_pulga). The Python package dependences are:

Unittest, wifiUtils (the files are included), paho.mqtt.client, logging, os, filecmp, datetime, dbus, time.

The steps for executing them are:

1) Run myriad/apps/pulga_control_app in the EoT device. It will create an AP (SSID = Myriad2Wifi, password = visilabap, if the WifiFunctions library has not been modified) and will start the MQTT broker.
2) When Pulga shows "Waiting" in the command line, run the test.py file. This test will create a MQTT client which will connect to the generated AP and will test several of the functionalities of Pulga.

See in the Annex some support info to get the Python test running.

7.3.1. Tests implemented

test00ConnectToBroker

The client tests if it is able to connect to and disconnect from the broker. It also tests the connection of the client if it is already connected.

test01SubscribeUnsubscribe

The client tests the subscription and unsubscription from a topic.

test02UploadFile

The client tests the upload file feature.

test03DownloadFile

The client tests the download file feature

test04UploadDownloadFileAreEqual

The test compares the downloaded file with the uploaded file.

test05RequestSnapshot

The client requests a snapshot from Pulga.

test06UpdateDate

The client tests the update date functionality.

test07GetDate

The client tests the get date functionality.

test08EOTMakeDirSD

The test creates a dir “Test” in the SD card.

test09EOTListFilesSD

The client obtains the complete list of files contained in the SD card.

test10EOTDeleteFileSD

The test deletes the file uploaded previously to the SD card.

test11EOTDeleteDirSD

The test deletes the dir “Test” from the SD card.

test12UploadElf

The test uploads an Elf binary to the flash memory of the EoT device.

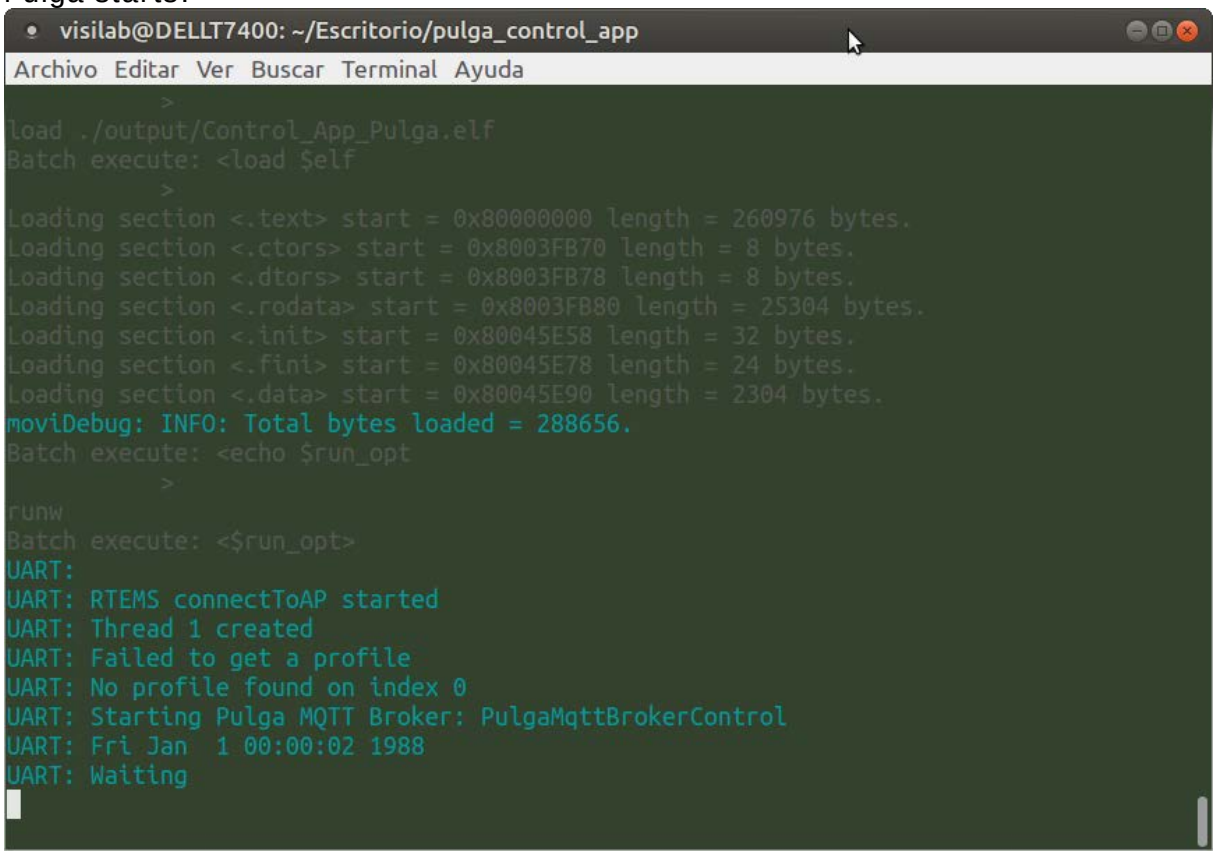
7.3.2. Expected output of the tests

The expected output of the tests must be similar to (currently test05 will fail, because the snapshot feature is not implemented in this version):

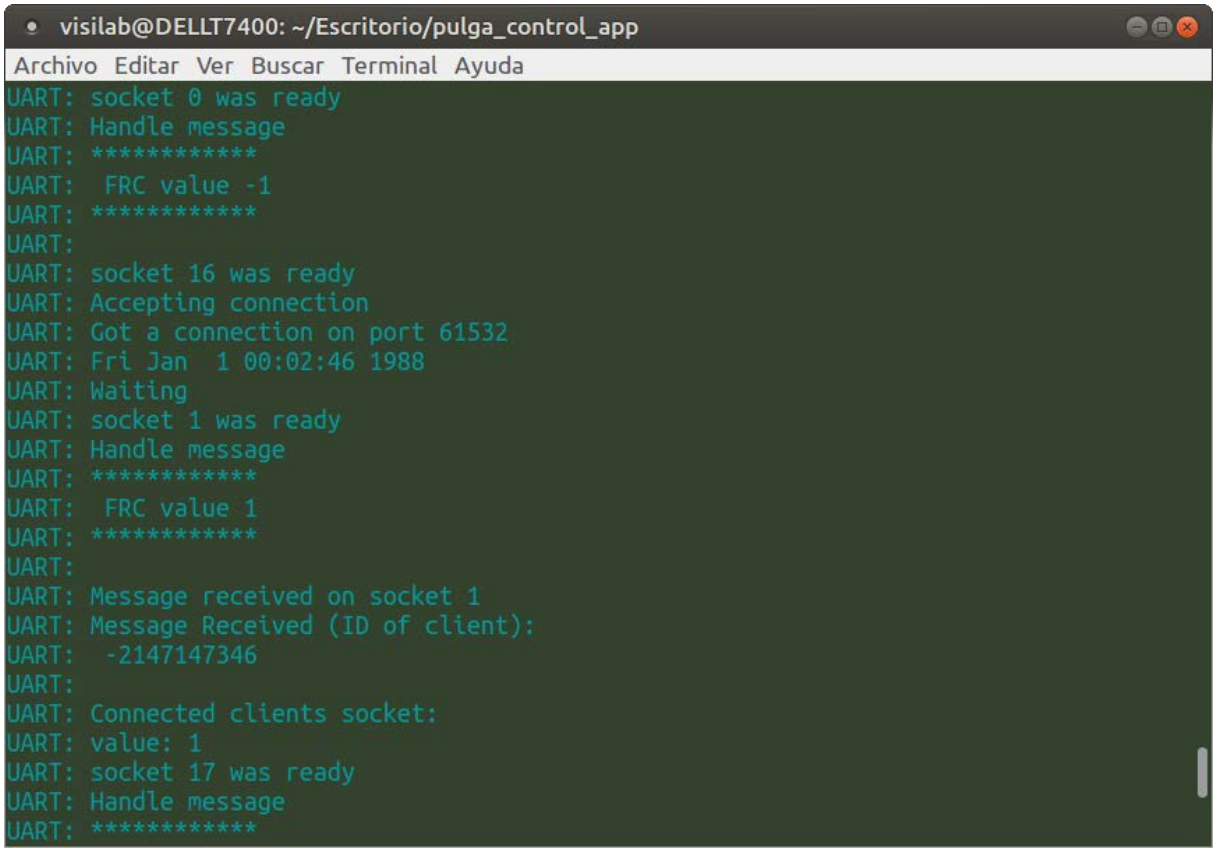
```
test00ConnectToBroker (__main__.PulgaTests) ... Waiting for connection to reach NM_ACTIVE_CONNECTION_STATE_ACTIVATED state ...
Connection established!
ok
test01SubscribeUnsubscribe (__main__.PulgaTests) ... Check if unsubscription was successful
ok
test02UploadFile (__main__.PulgaTests) ... Uploading 000000 chunk
Uploading 000001 chunk
......
Uploading 000013 chunk
ok
test03DownloadFile (__main__.PulgaTests) ... Downloading chunk
Downloading chunk
......
Downloading chunk
ok
test04UploadDownloadFileAreEqual (__main__.PulgaTests) ... ok
test05RequestSnapshot (__main__.PulgaTests) ... FAIL
test06UpdateDate (__main__.PulgaTests) ... ok
test07GetDate (__main__.PulgaTests) ... ok
test08EOTMakeDirSD (__main__.PulgaTests) ... ok
test09EOTListFilesSD (__main__.PulgaTests) ... OpenCVTests;0;Thu Dec  3 12:04:36 2015
Thu Dec  3 12:04:36 2015
Thu Dec  3 12:04:36 2015

test.py;1;Fri Jan  1 00:00:18 1988
Fri Jan  1 00:00:18 1988
Fri Jan  1 00:00:18 1988
......
ok
test10EOTDeleteFileSD (__main__.PulgaTests) ... ok
test11EOTDeleteDirSD (__main__.PulgaTests) ... ok
test12UploadElf (__main__.PulgaTests) ... Uploading 000000 chunk
Uploading 000001 chunk
......
Uploading 000013 chunk
ok

======================================================================
FAIL: test05RequestSnapshot (__main__.PulgaTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 178, in test05RequestSnapshot
    self.assertTrue(False, "Error on received message with size of snapshot")
AssertionError: Error on received message with size of snapshot

----------------------------------------------------------------------
Ran 13 tests in 18.286s
```

Using the Application / Screenshots

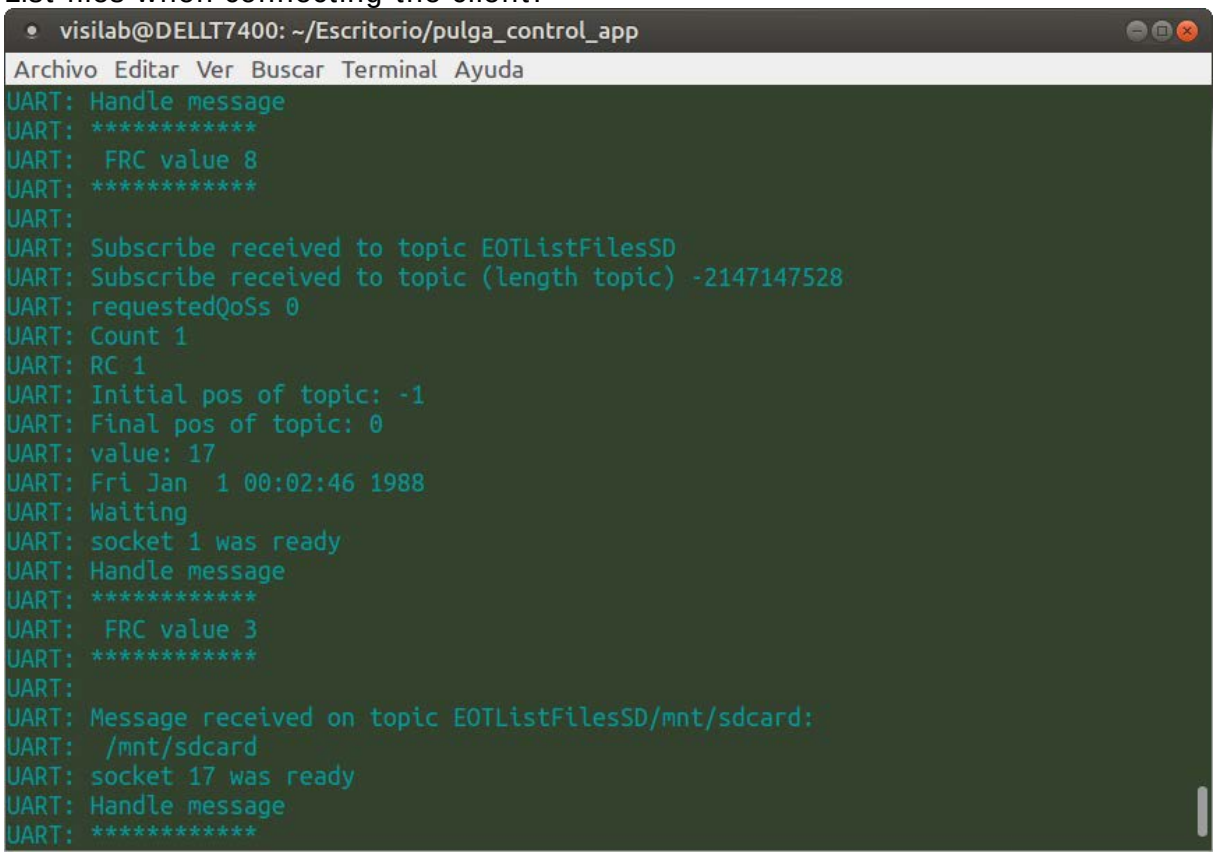
Pulga starts:



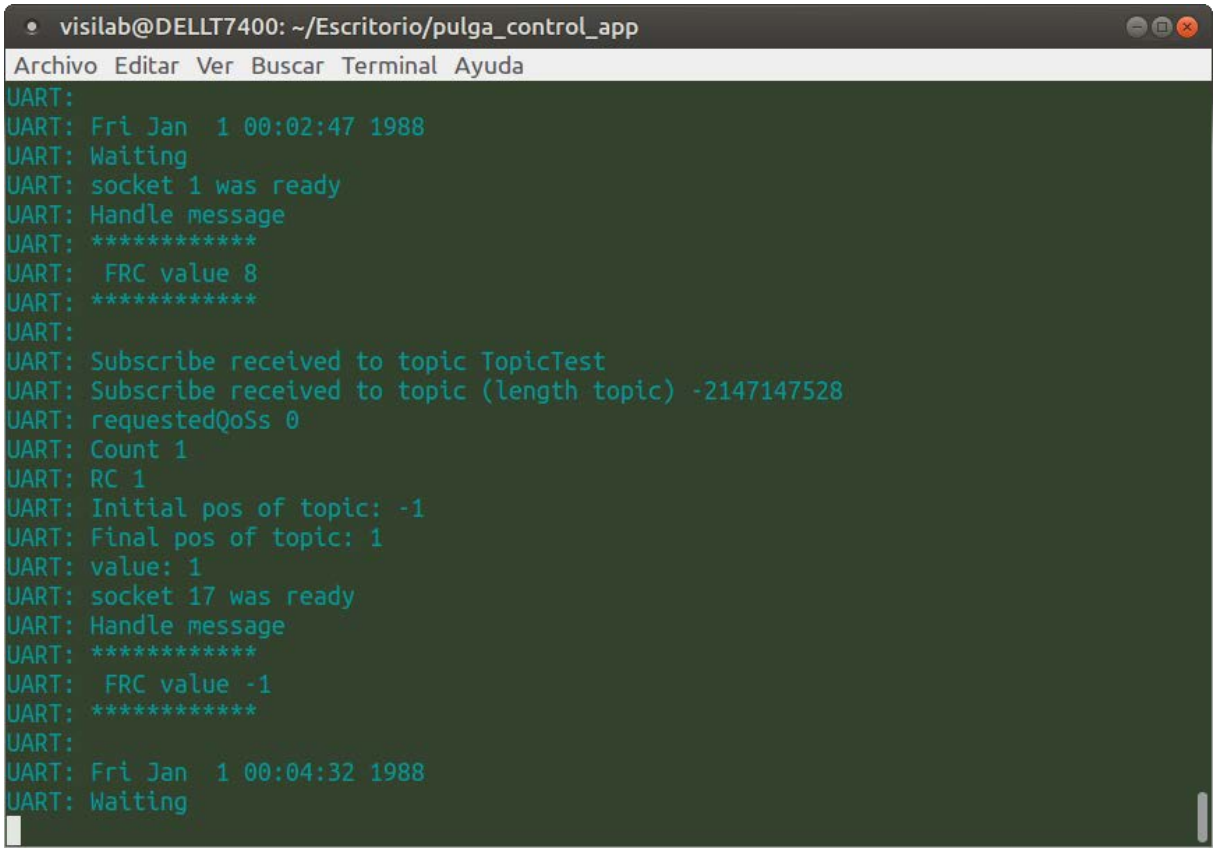
Client connection:



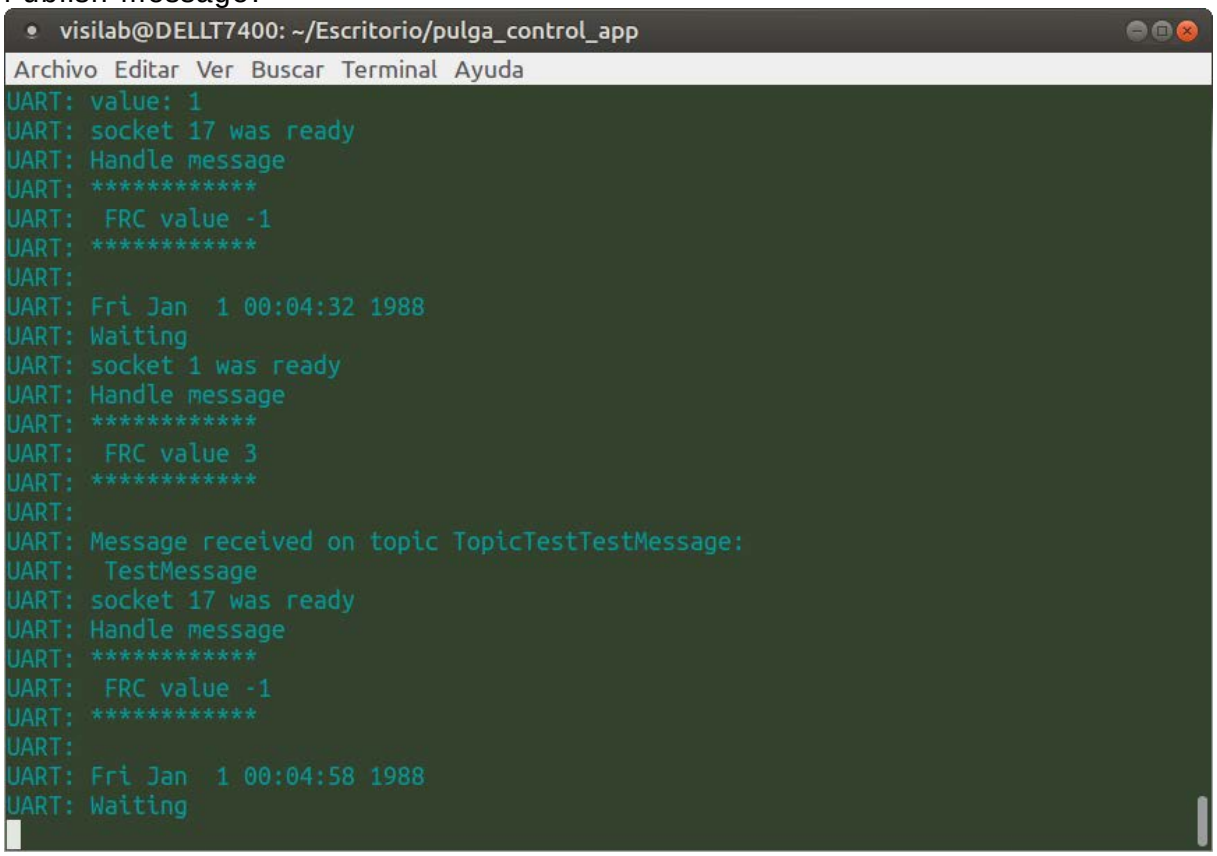
List files when connecting the client:



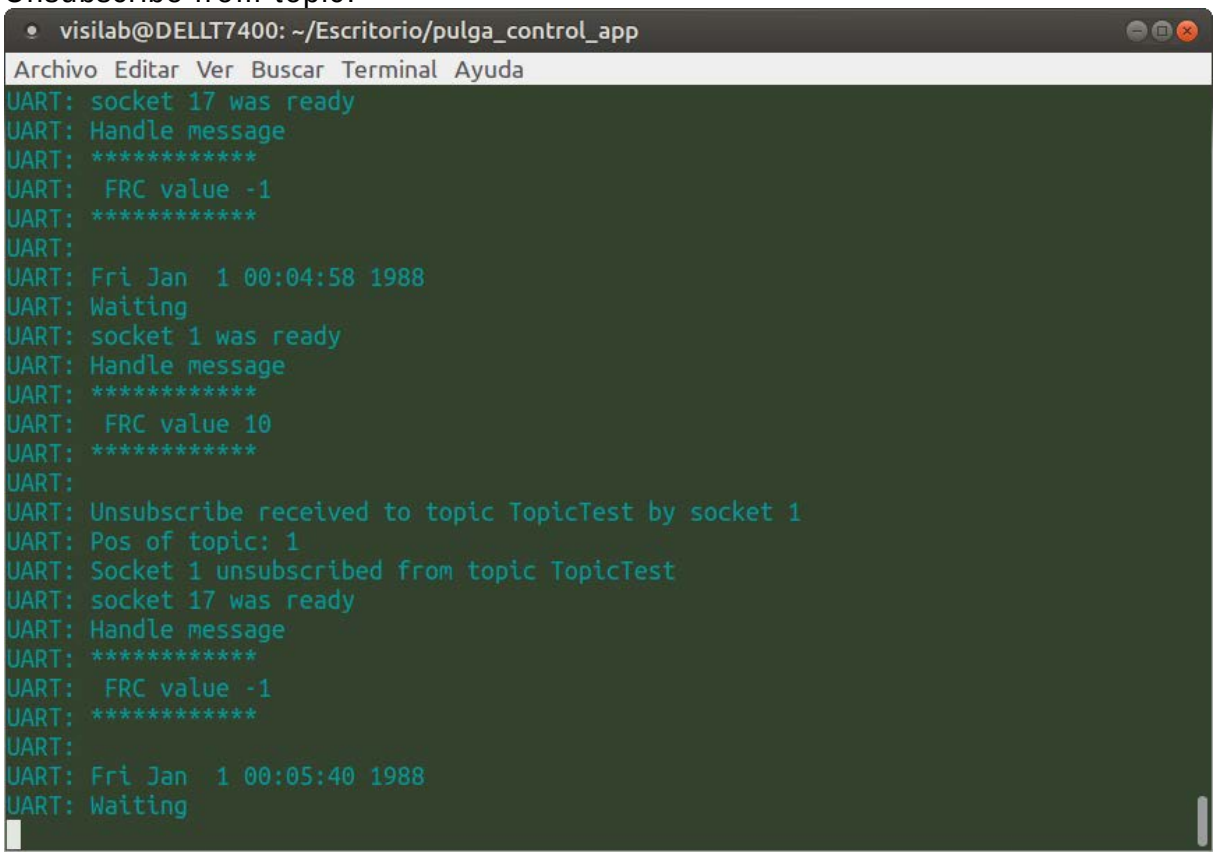
Topic subscription:



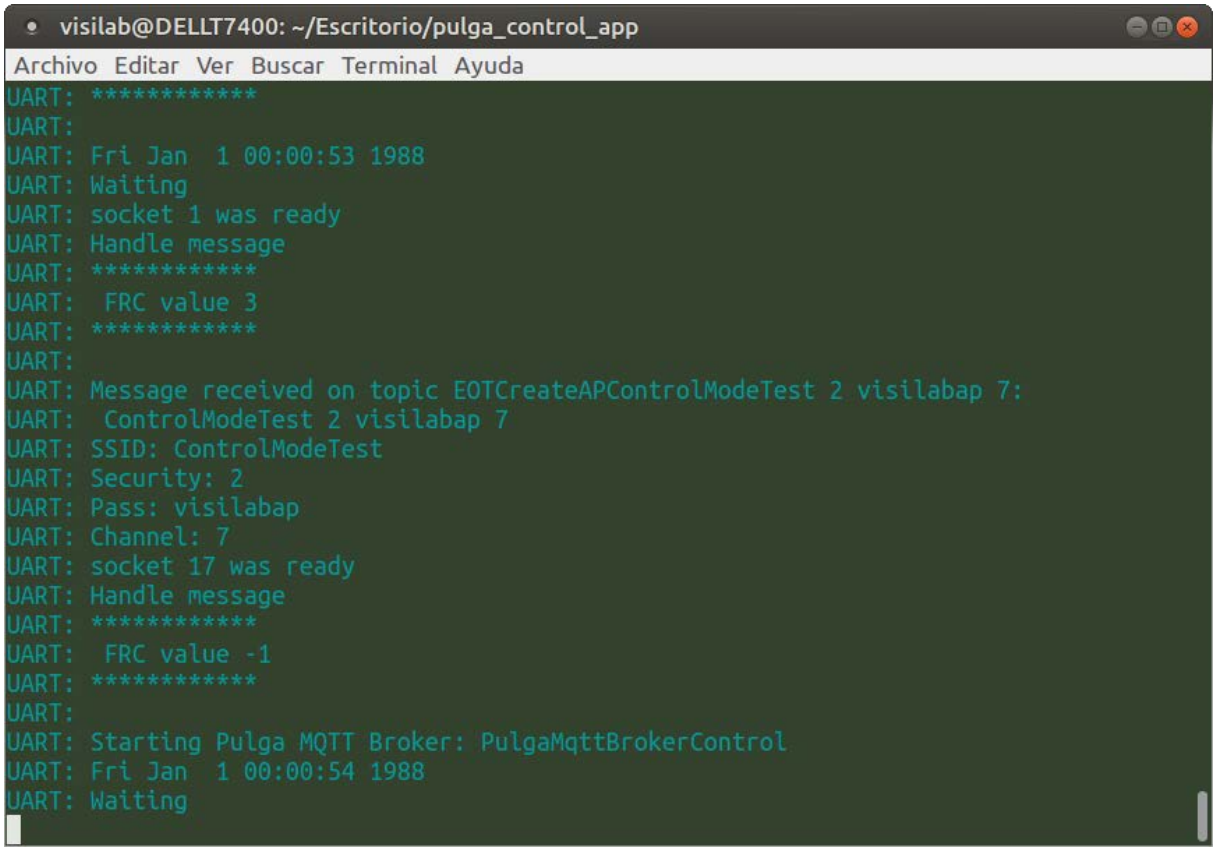
Publish message:



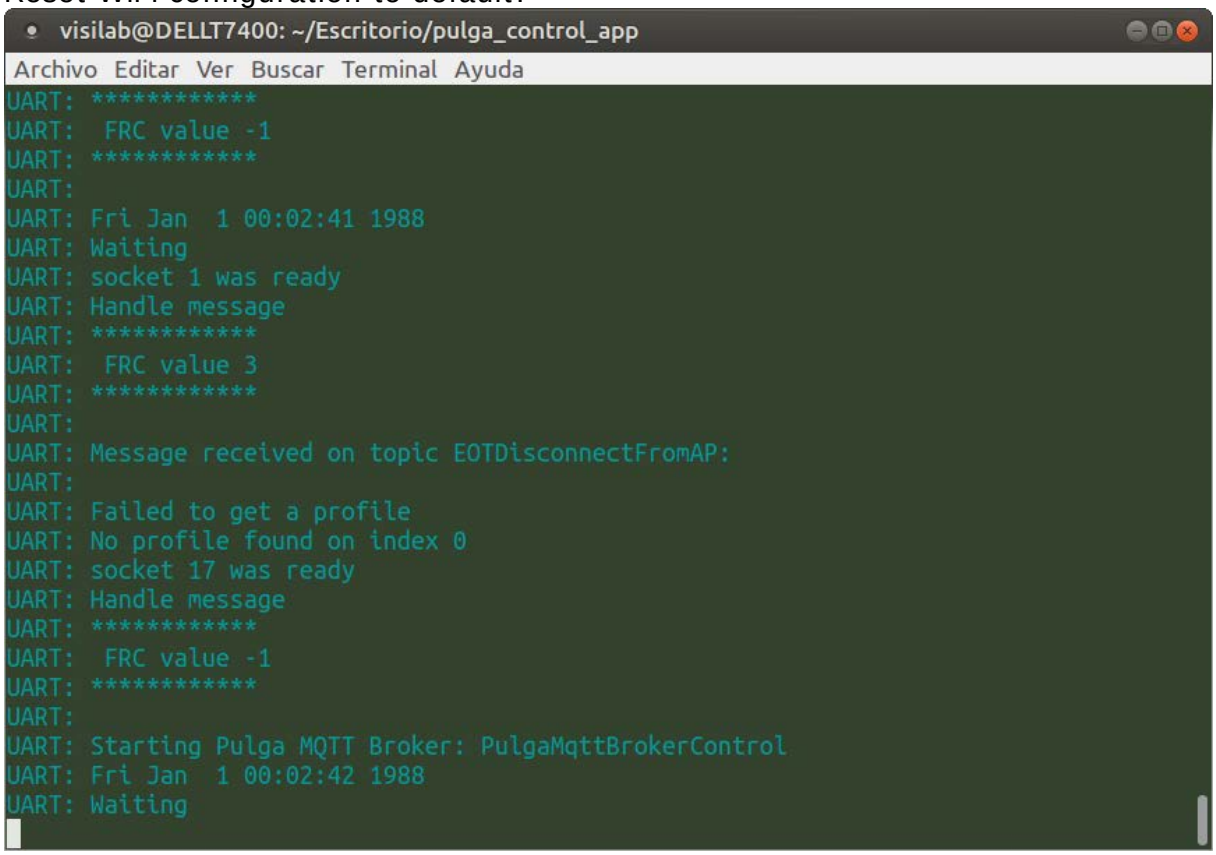
Unsubscribe from topic:



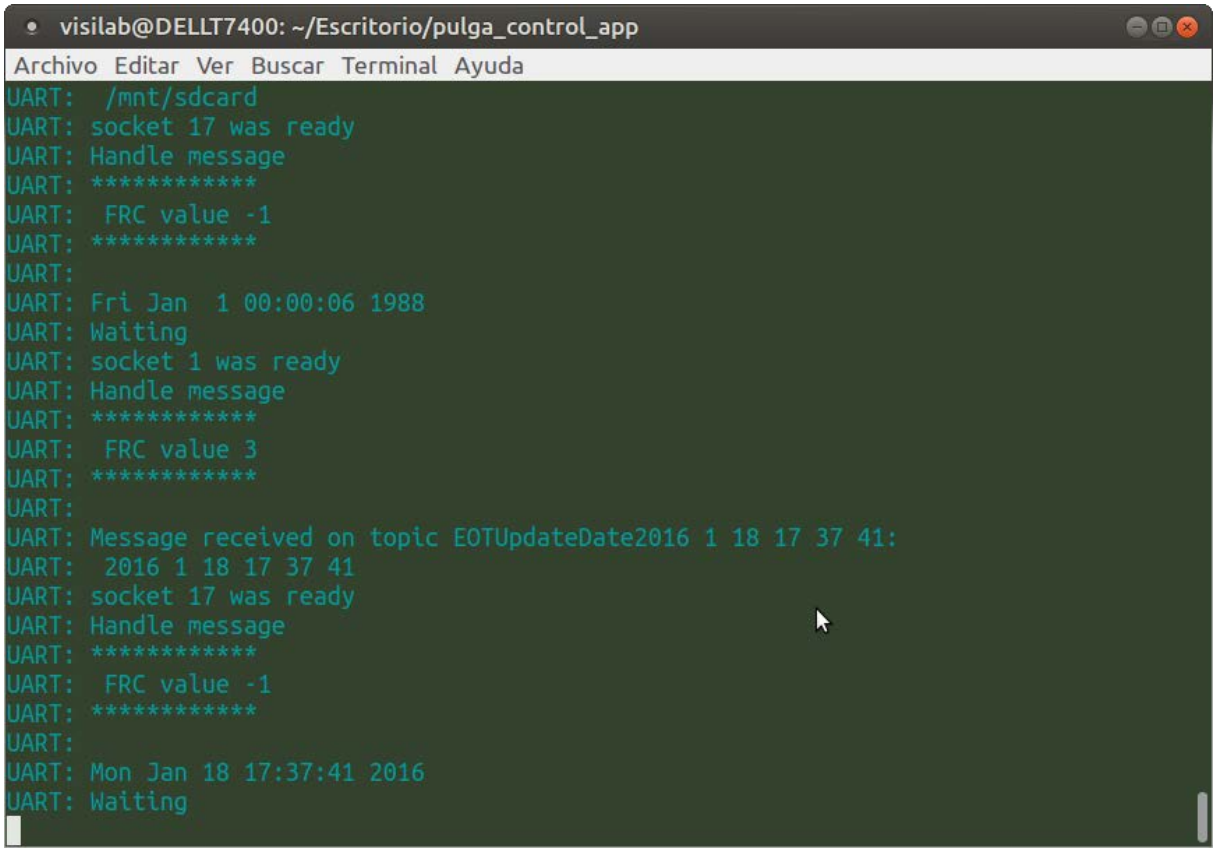
Create Access Point:



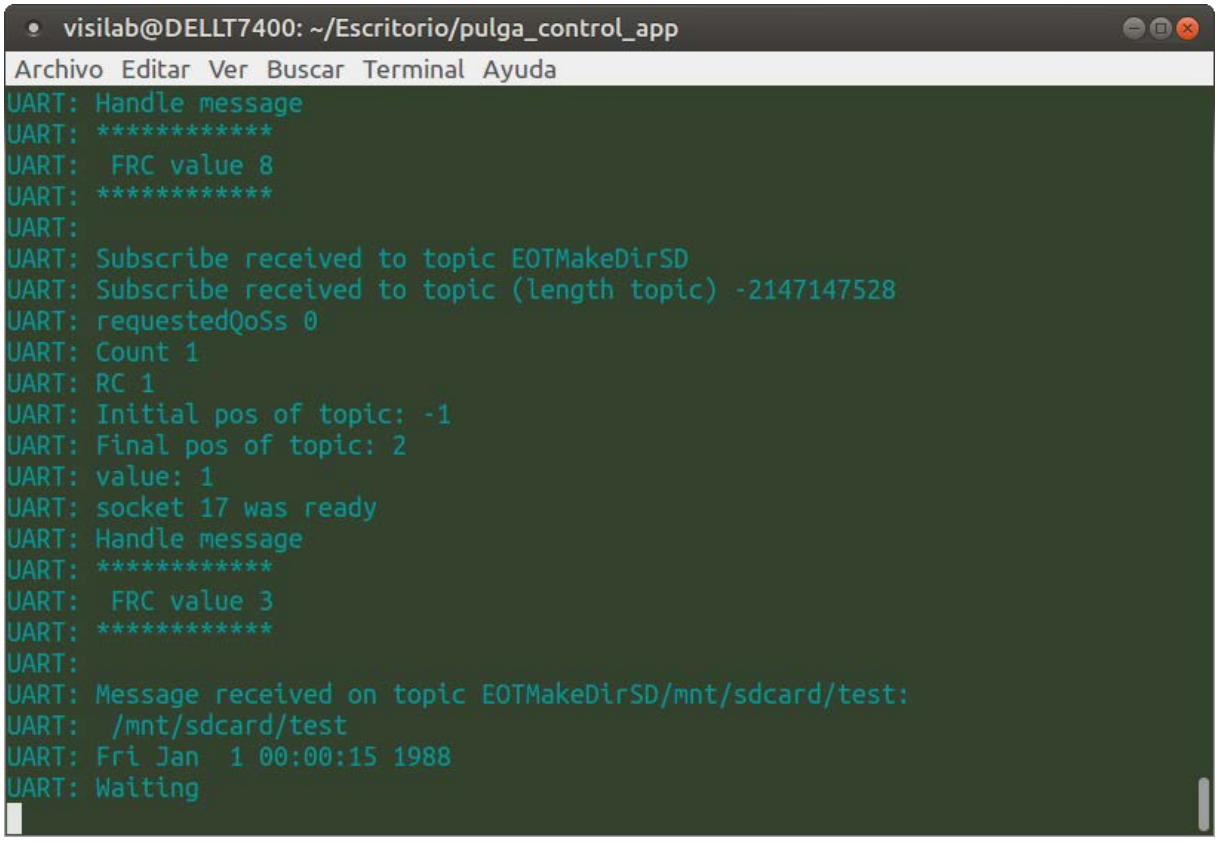
Reset WiFi configuration to default:



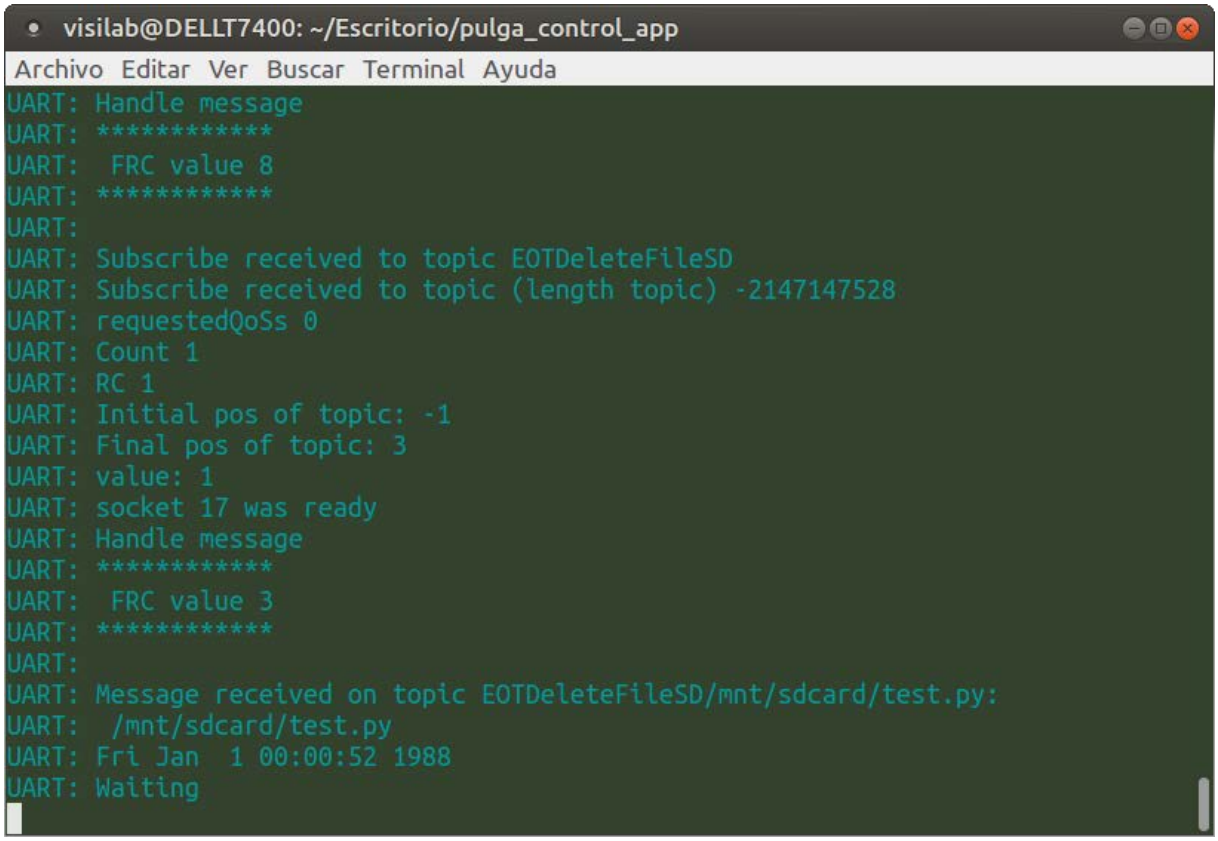
Update date of EoT device:



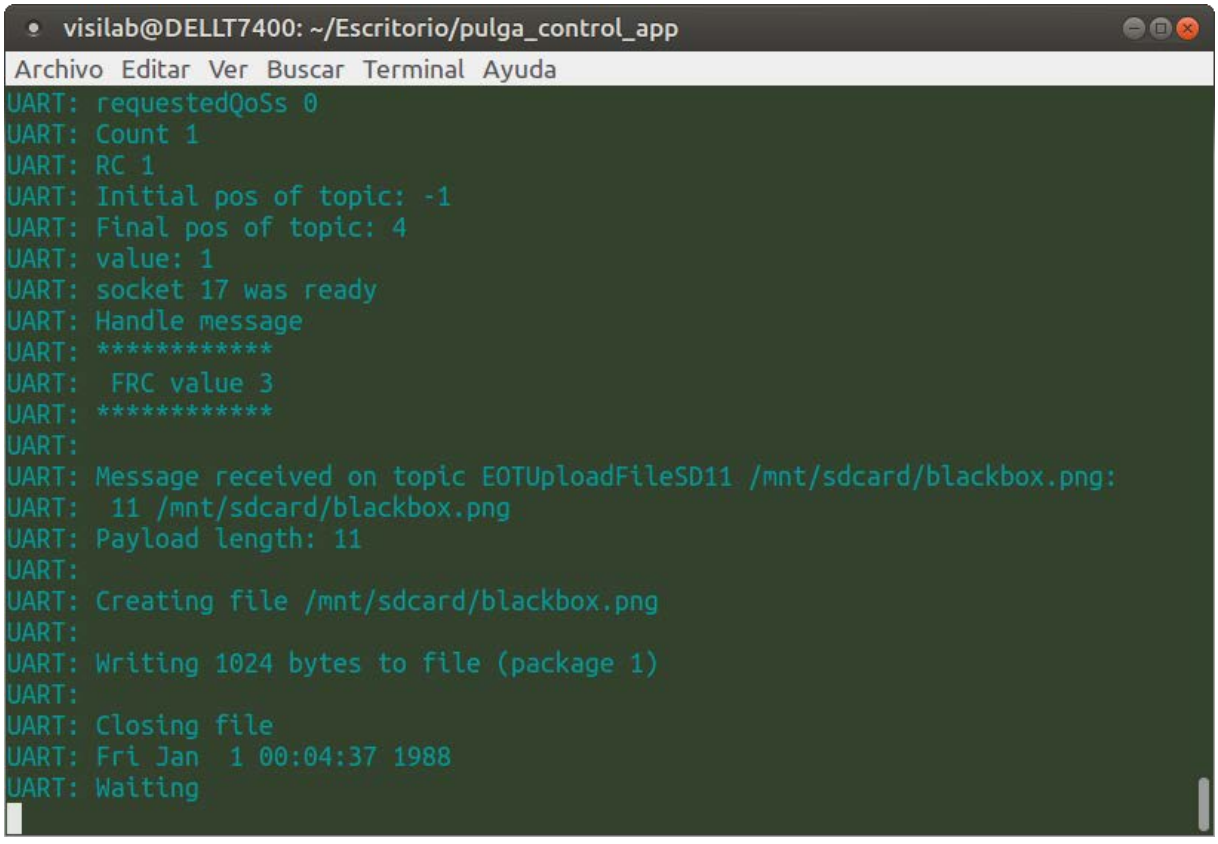
Create folder:



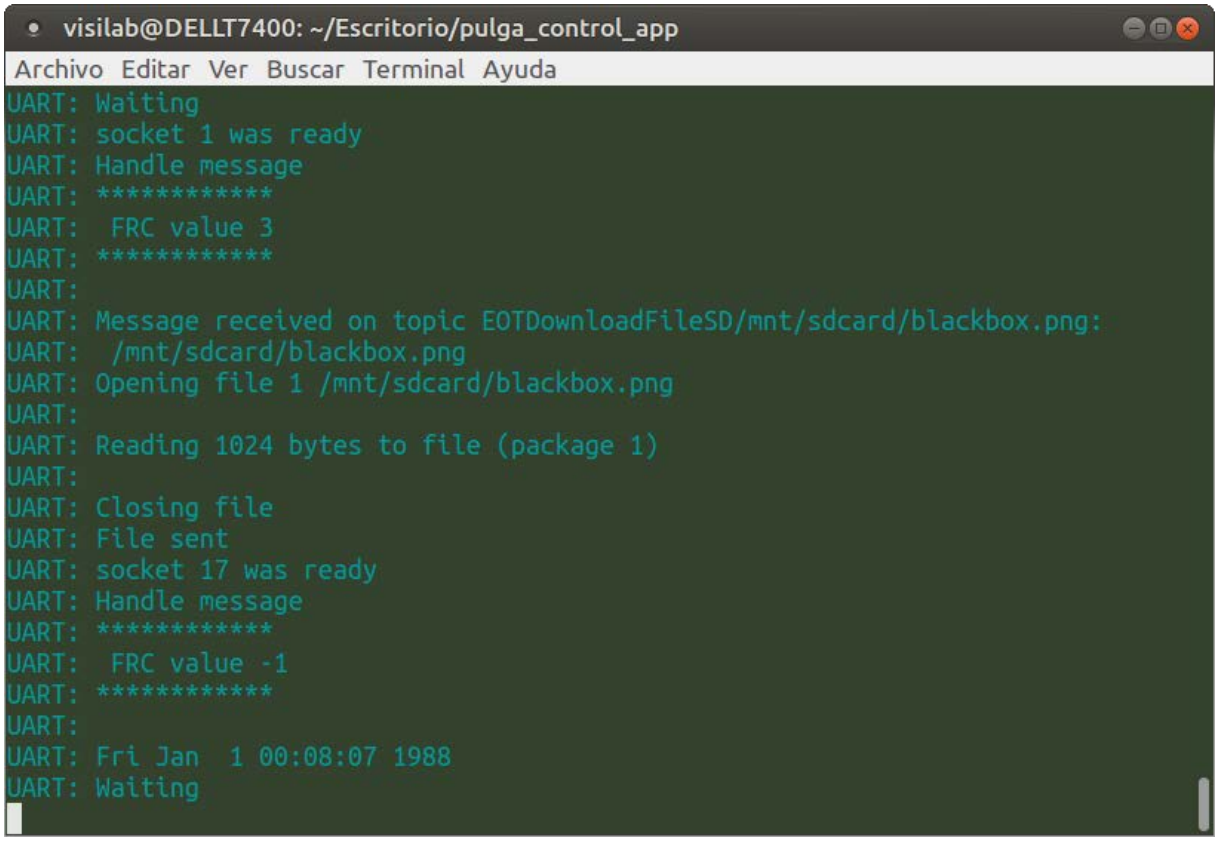
Remove file from SD card:



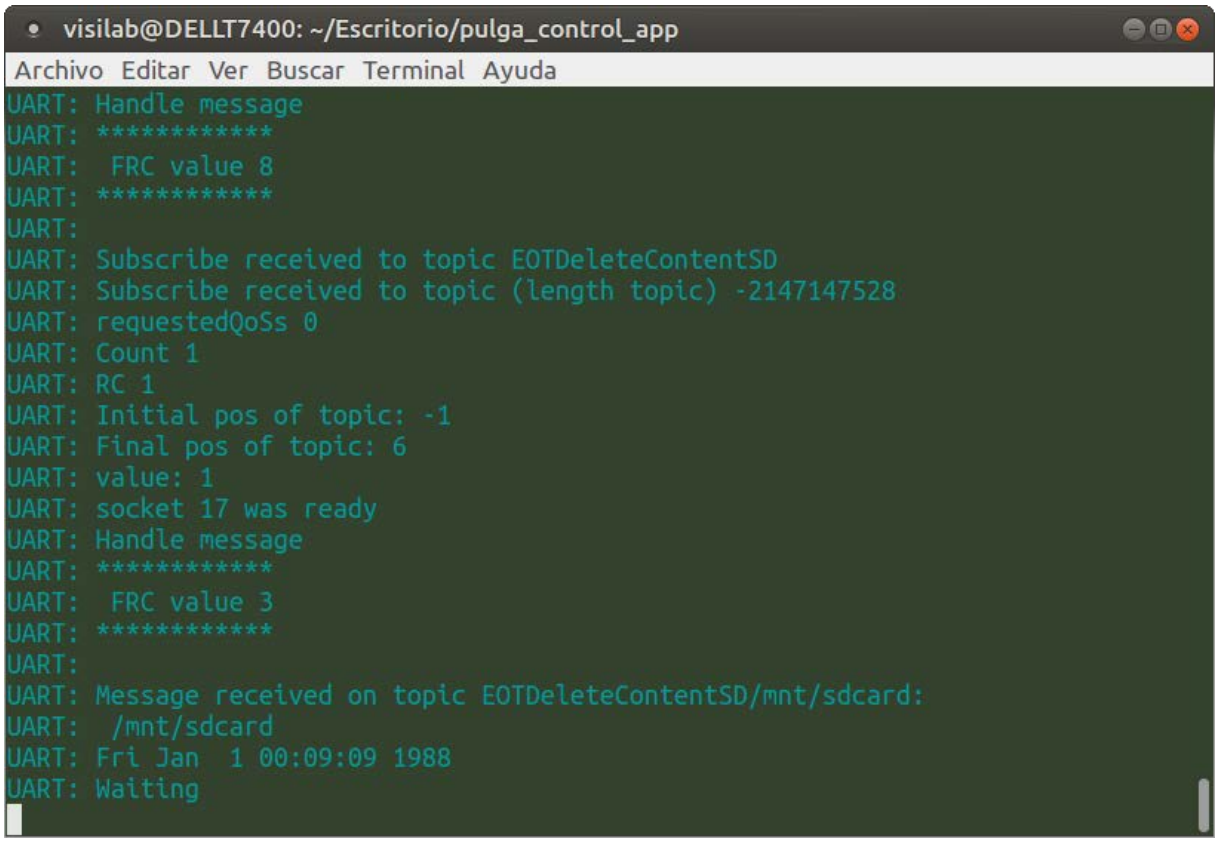
Receive file and store it on SD card:



Send file from SD card to client:



Remove all content from SD card:



To Do

- Implement the snapshot retrieval capability when the new camera is ready.

- Implement all the functionality of the flash when conflicts between the WiFi chip and the flash memory are solved in the new hardware board.

8. MQTT CLIENT APPLICATION DEVELOPED IN JAVA (JAVA API)

This section describes the counterpart of Pulga for a desktop/laptop computer. An MQTT client can act as a publisher, a subscriber or both. Due to the small resources needed by the MQTT protocol, an MQTT client may run in any device from a micro controller up to a server. Basically any device that has a TCP/IP stack can use MQTT over it using:

- A plain TCP socket
- A secure SSL/TLS socket

The MQTT application only requires an MQTT library that connects the client with the broker through a network connection in order to send and receive small messages. There are many open-source MQTT client libraries available for a variety of programming languages such as Java, JavaScript, C, C++, C#, Go, iOS, .NET, Android, or Arduino.

Here, the EoT Desktop MQTT client has been developed in Java using the Paho Java Client library [7]. That way, the same code base can be used for both desktop and Android applications.

Paho Java Client

Paho Java Client is an MQTT client library written in Java for developing applications that runs on the Java Virtual Machine, JVM. Moreover, it can be used under Android through the Paho Android Service.

Paho provides two APIs: MqttAsyncClient and MqttClient.

- MqttAsyncClient provides a fully asychronous API where completion of activities is notified via registered callbacks.
- MqttClient is a lightweight client that blocks the application until an operation is complete. This class implements the blocking IMqttClient client interface.

The EoT MQTT application is divided into two clases: the EoT_MainFrame and the EoT_MQTT_Client. The first one only contains the code of the graphical user interface whereas the second one, the EoT_MQTT_Client manages the Paho client and provides all the functionalities needed.

API Specification

Class EoT_MQTT_Client

The EoT_MQTT_Client.

1 Declaration

public class EoT_MQTT_Client
extends java.lang.Object

2 Fields

public final java.lang.String **topicEOTConnectToAP**
public final java.lang.String **topicEOTContentSD**
public final java.lang.String **topicEOTCreateAP**
public final java.lang.String **topicEOTDeleteDirSD**
public final java.lang.String **topicEOTDeleteFileSD**
public final java.lang.String **topicEOTDisconnectFromAP**
public final java.lang.String **topicEOTDownloadFileSD**
public final java.lang.String **topicEOTGetDate**
public final java.lang.String **topicEOTListFilesSD**
public final java.lang.String **topicEOTMakeDirSD**
public final java.lang.String **topicEOTUpdateDate**
public final java.lang.String **topicEOTUploadElf**
public final java.lang.String **topicEOTUploadFileSD**
public final java.lang.String **topicSnapshot**

3 Constructor summary

EoT_MQTT_Client(String,int) Gets an instance of EoT_MQTT_Client

4 Method summary

- **askSnapshot()**
  Sends a message in the topicSnapshot topic to get the image from the broker
- **connect()**
  Connects the client to the MQTT server
- **connectionLost(Throwable)**
- **connectToAP(String, String, String)**
  Connects the EoT device to an external AP
- **createAP(String, String, String, String)**
  Creates a new AP configuration profile
- **createFolder(String)**
  Makes a new folder in the SD card
- **deliveryComplete(IMqttDeliveryToken)**
- **disconnect()**
  Disconnects the client
- **downloadFile(String, String)**

Downloads a file from the SD card

- **getDate()**
  Gets the current EoT device time/date
- **getFileSystemStructure(String)**
  Gets the paths of the SD card content
- **isConnected()**
  Checks if the client is connected
- **messageArrived(String, MqttMessage)**
- **publish(String, int, byte[])**
  Publishes / sends a message to an MQTT server
- **removeAll(String)**
  Removes a folder and its content recursively
- **removeContent(String)**
  Removes the content of a folder (or the SD card if /mnt/sdcard is used)
- **removeFile(String)**
  Removes a file from the SD card
- **resetAPConfig()**
  Resets the AP configuration to the default profile
- **setMainFrame(EoT_MainFrame)**
  Sets the main frame where results are displayed
- **subscribe(String, int)**
  Subscribes the client to a topic on an MQTT server
- **unsubscribe(String)**
  Unsubscribes the client from a topic
- **updateDate(String, String, String, String, String, String)**
  Changes the EoT device time/date
- **uploadFile(String, String)**
  Sends a file to the SD card

6 Constructor

public EoT_MQTT_Client(java.lang.String brokerip , int brokerport)

– Description
  Gets an instance of EoT_MQTT_Client

– Parameters
  i. brokerip – IP where the broker is running
  ii. brokerport – port used by the broker

7 Methods

- **askSnapshot**

public javax.swing.ImageIcon askSnapshot() throws MqttException

– Description
  Sends a message in the topicSnapshot topic to get the image from the broker

– Throws

* MqttException

- **connect**

public void connect () throws MqttException

  - Description
    Connects the client to the MQTT server

  - Throws
    * MqttException

- **connectionLost**

  - Parameters
    * cause

  - See also
    public void connectionLost (java.lang.Throwable cause)
    * MqttCallback#connectionLost(Throwable)

- **connectToAP**

public void connectToAP(java.lang.String SSID, java.lang.String security, java.lang.String pass) throws MqttException

  - Description
    Connects the EoT device to an external AP

  - Parameters
    * SSID
    * security
    * pass

  - Throws
    * MqttException

- **createAP**

public void createAP(java.lang.String SSID, java.lang.String security, java.lang.String pass, java.lang.String channel) throws MqttException

  - Description
    Creates a new AP configuration profile

  - Parameters
    * SSID
    * security
    * pass
    * channel

  - Throws

* MqttException

- **createFolder**

  public int createFolder(java.lang.String path) throws MqttException

  – Description
    Makes a new folder in the SD card

  – Parameters
    * path – Path of the new folder

  – Returns – 0 if the operation was successfully completed

- **deliveryComplete**

  public void deliveryComplete (IMqttDeliveryToken token)

  – Parameters
    * token

  – See also
    * MqttCallback#deliveryComplete(IMqttDeliveryToken)

- **disconnect**

  public void disconnect () throws MqttException

  – Description
    Disconnects the client

  – Throws
    * MqttException

- **downloadFile**

public void downloadFile(java.lang.String srcDir, java.lang.String dstDir) throws java.lang.Exception

  – Description
    Downloads a file from the SD card

  – Parameters
    * srcDir – SD card path of the file
    * dstDir – Path where the file should be store

  – Throws
    * java.lang.Exception

- **getDate**

  public java.util.Calendar getDate() throws MqttException

- Description
  Gets the current EoT device time/date

- Returns
  Calendar. The device current time/date

- Throws
  * MqttException

- **getFileSystemStructure**

public java.lang.String[] getFileSystemStructure(java.lang.String path) throws MqttException

- Description
  Gets the paths of the SD card content

- Returns
  A String[] with all the file and folder paths

- **isConnected**

public boolean isConnected ()

- Description
  Checks if the client is connected

- Returns
  true if the client is connected

- **messageArrived**

public void messageArrived(java.lang.String topic, MqttMessage messageArrived) throws java . lang . Exception

- Parameters
  * topic
  * messageArrived

- Throws
  * java.lang.Exception

- See also
  * MqttCallback#messageArrived(String, MqttMessage)

- **publish**

public void publish(java.lang.String topicName, int qos, byte [] payload) throws MqttException

- Description
  Publishes / sends a message to an MQTT server

- Parameters
  * topicName – the name of the topic to publish to
  * qos – the quality of service to deliver the message at (0,1,2) (0 in this case)
  * payload – the set of bytes to send to the MQTT server

- Throws
  * MqttException

- **removeAll**

  public int removeAll(java.lang.String path) throws MqttException

  - Description
    Removes a folder and its content recursively

  - Parameters
    * path – Path of the folder

  - Returns
    0 if the operation was successfully completed

- **removeContent**

  public int removeContent(java.lang.String path) throws MqttException

  - Description
    Removes the content of a folder (or the SD card if /mnt/sdcard is used)

  - Parameters
    * path – Path of the folder

  - Returns – 0 if the operation was successfully completed

- **removeFile**

  public int removeFile(java.lang.String path) throws MqttException

  - Description
    Removes a file from the SD card

  - Parameters
    * path – Path of the file to be removed

  - Returns

0 if the operation was successfully completed

- **resetAPConfig**

  public void resetAPConfig () throws MqttException

  - Description
    Resets the AP con.guration to the default profile

  - Throws
    * MqttException

- **setMainFrame**

  public void setMainFrame(EoT_MainFrame frame)

  - Description
    Sets the main frame where results are displayed

  - Parameters
    * frame

- **subscribe**

public void subscribe(java.lang.String topicName, int qos) throws MqttException

  - Description
    Subscribes the client to a topic on an MQTT server

  - Parameters
    * topicName – to subscribe to (can be wild carded)
    * qos – the maximum quality of service to receive messages at for this subscription

  - Throws
    * MqttException

- **unsubscribe**

public void unsubscribe(java.lang.String topicName) throws MqttException

  - Description
    Unsubscribes the client from a topic

  - Parameters
    * topicName

  - Throws
    * MqttException

- **updateDate**

public int updateDate(java.lang.String year, java.lang.String month, java.lang.String day, java.lang.String hour, java.lang.String mins, java.lang.String secs) throws MqttException

  - Description
    Changes the EoT device time/date

  - Parameters
    * year
    * month
    * day
    * hour
    * mins
    * secs

  - Throws
    * MqttException

- **uploadFile**

public int uploadFile(java.lang.String srcDir, java.lang.String dstName) throws java.lang.Exception

  - Description
    Sends a file to the SD card

  - Parameters
    * srcDir – Path of the file
    * dstName – SD card path where the file should be store

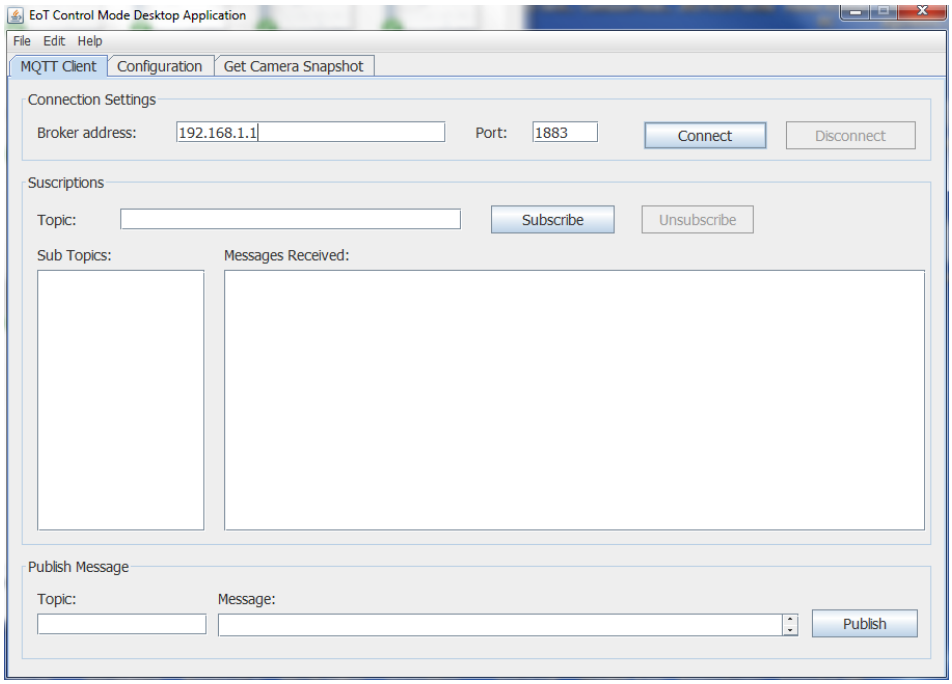
  - Returns
    0 if all is OK

  - Throws
    * java.lang.Exception

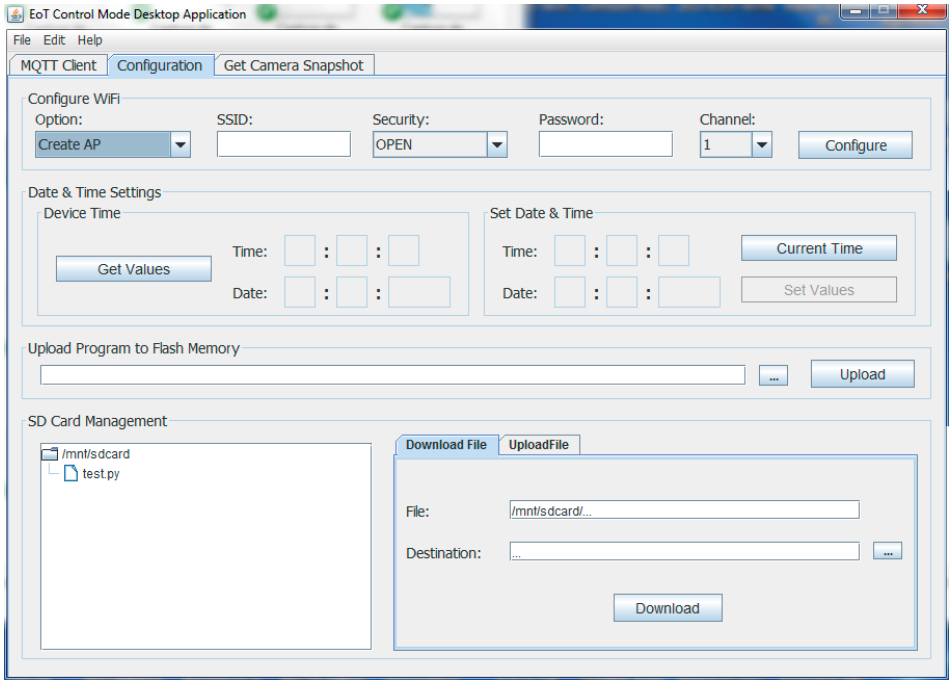
User Interface / Use of the Application

The application is divided into three tabbed panels. The first panel contains the functionalities of a typical MQTT client. The second panel allows the user to configure the EoT device and manage the files stored in the SD card. Finally, in the last panel the user can request a snapshot from the EoT device camera.

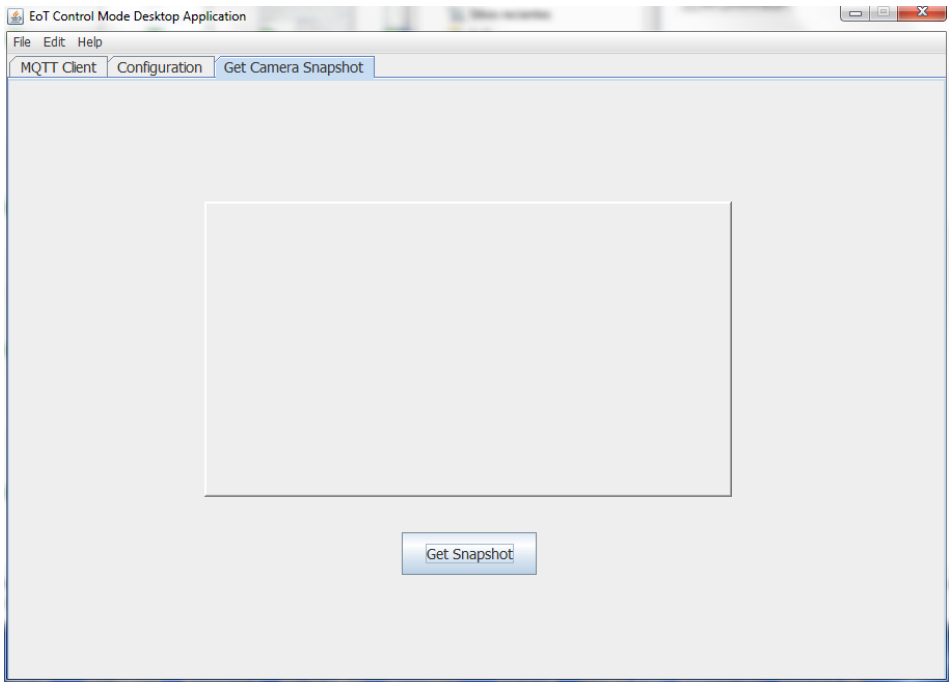
MQTT Client panel



Configuration panel

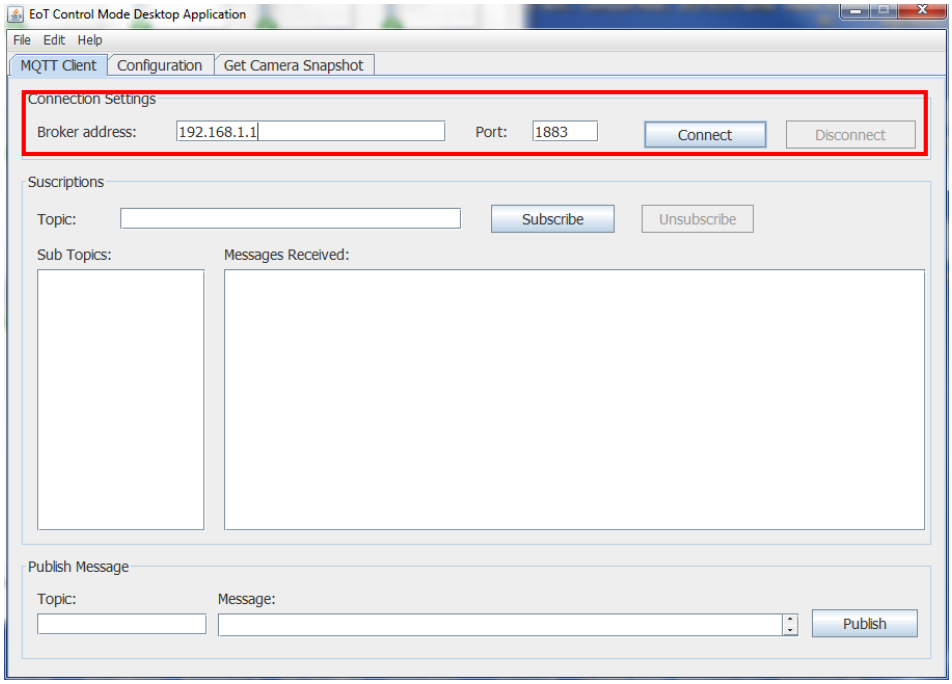


Snapshot panel

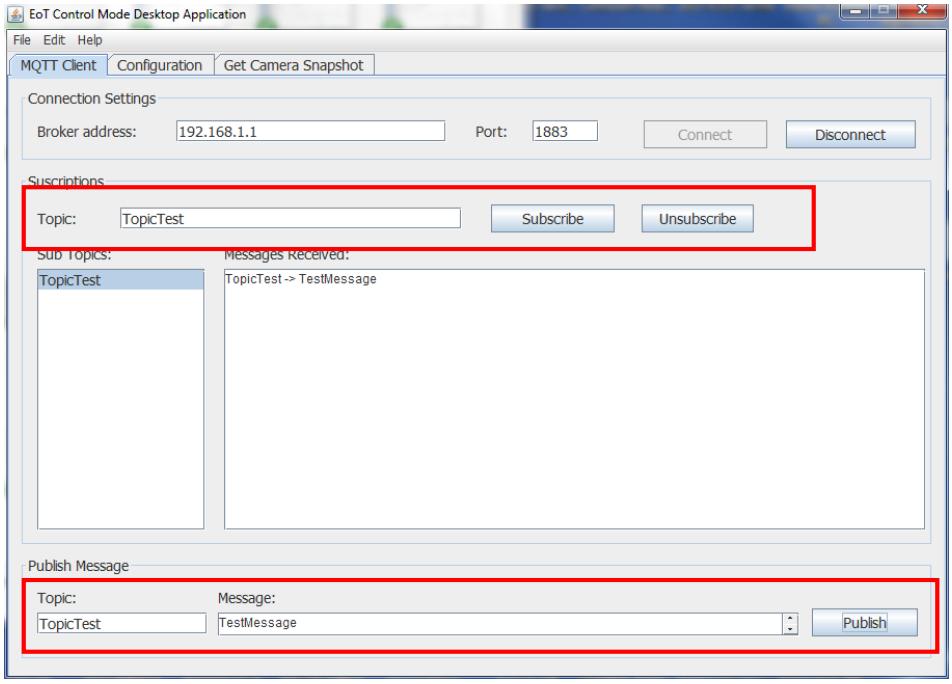


Before connecting the EoT Control Mode Desktop application to the EoT device the computer should be connected to the EoT device AP.

Once the computer is connected to the EoT device AP, the MQTT client can be connected to the Pulga broker using the correct IP address and Port.



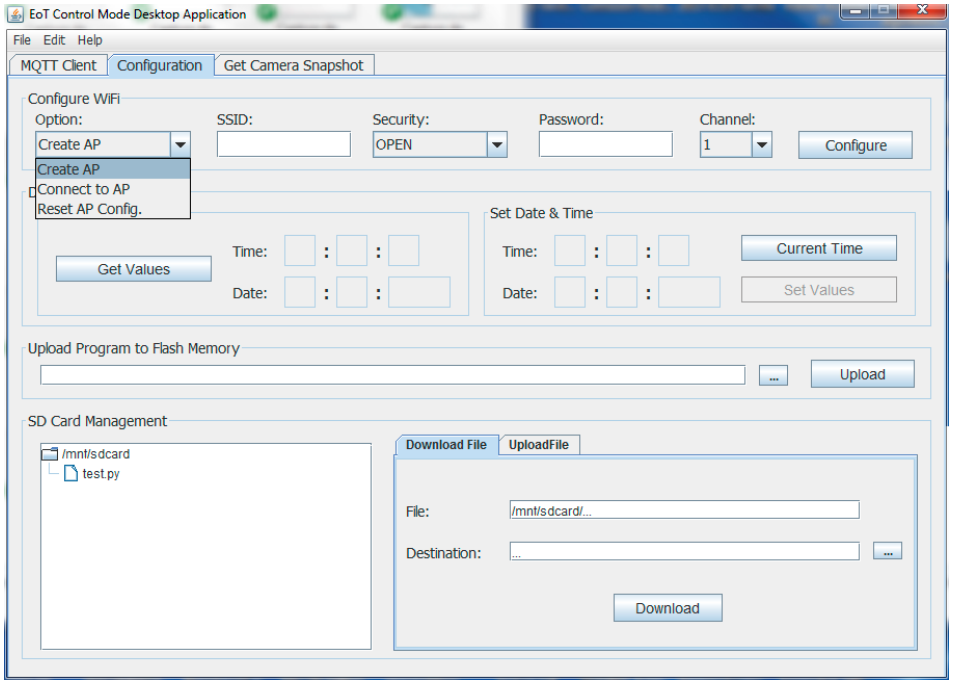
After that, it is possible to use the application as a common MQTT client, performing topic subscriptions and publishing messages to topics.



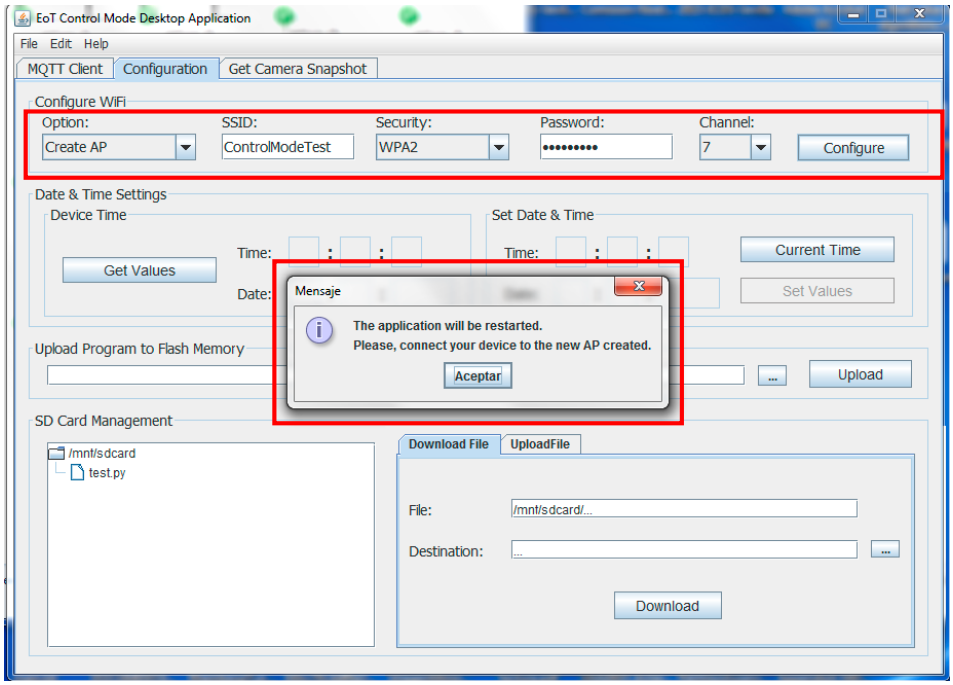
In the Configuration panel there are options that allow the user to configure WiFi and time and manage Flash and SD card memories.

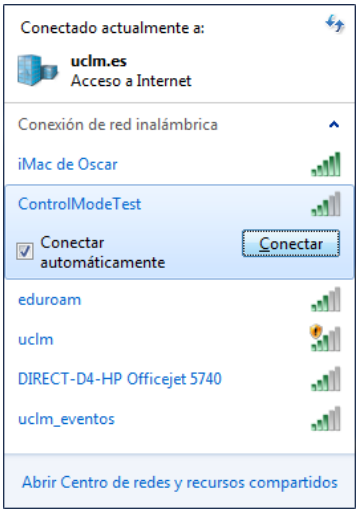
The EoT device WiFi configuration includes options to:

- create an AP with new parameters,
- connect the device with an existing AP and
- reset the device AP settings to the default profile.

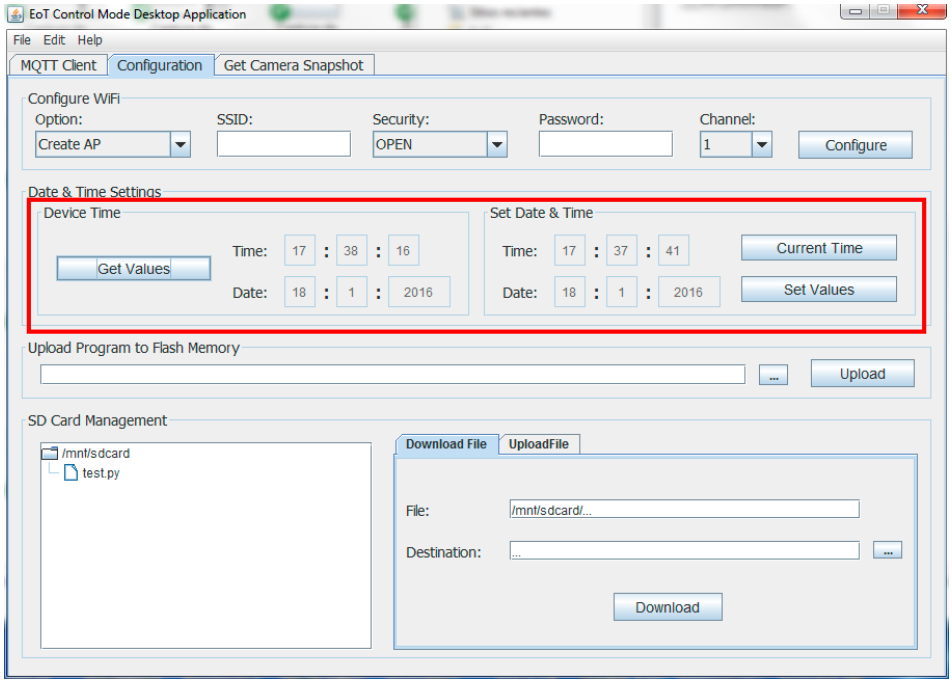


If the device's WiFi configuration is changed, the desktop application is restarted and the user needs to connect the computer to the new device AP or the same wireless network the device is connected to. Then, the client-broker connection should be established again.

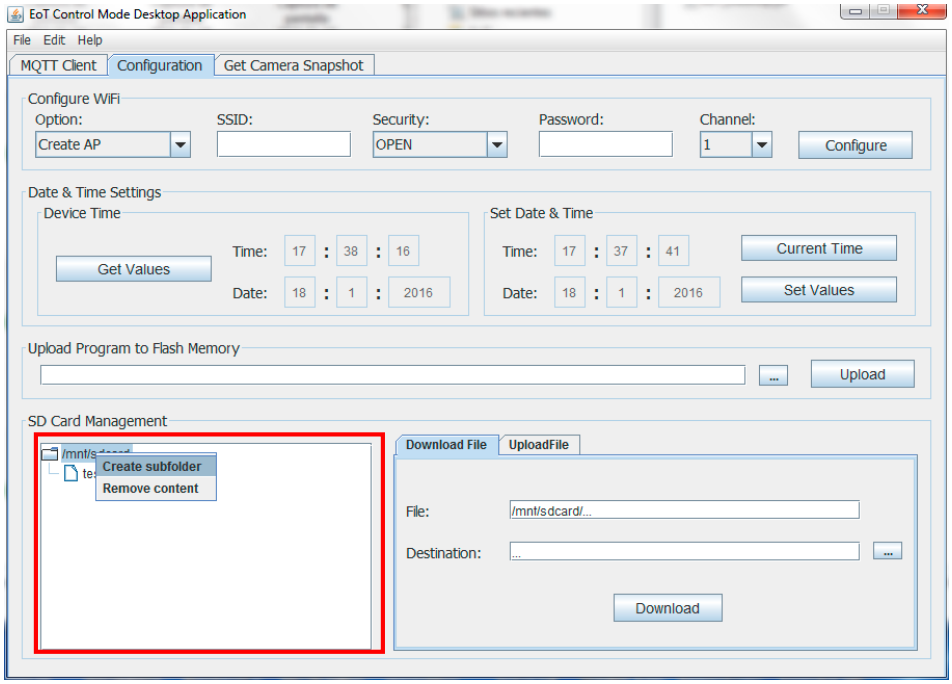


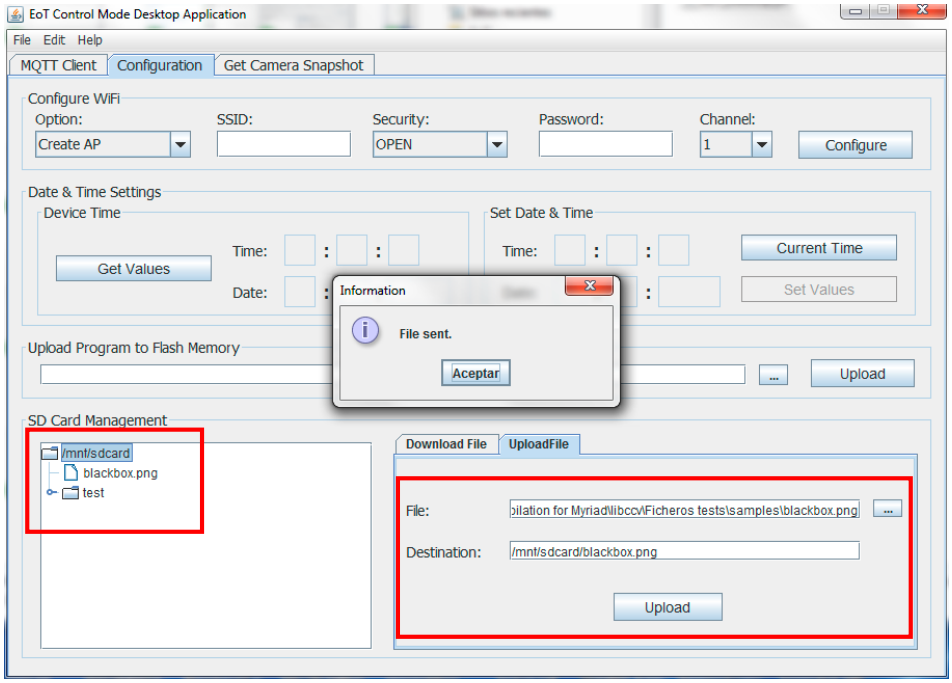


In order to set the current time values in the EoT device it is possible to use the Date & Time settings provided in the desktop application. This allows the user to get the current time of the computer and set them in the device. In addition, it is possible to check the current time of the EoT device.



Finally, the SD card management options are divided into two parts. The first part shows the directory tree of the SD card. Through the mouse right click the user can delete the content of a directory (including the root directory), delete a directory and its content, delete a file and create new directories. On the other hand, the second part allows to upload and download files to and from the SD card.





Note that when the EoT device is in AP mode (by default) only one client can be connected to it. This is considered a desiarable feature in terms of security. On the other hand, the Java application was tested with up to three clients. In order to do this, a device (Android smartphone) was used to generate a Wifi AP, and then the EoT device plus other two devices connected to that same Wifi and exchanged messages that were brokered by the EoT device.

Problems Found / Known Issues

In some cases, the MQTT client sends the unsubscribe message through a different socket than the socket used during connection. Since the broker uses the socket number to identify each client, an unsubscribe message through a different socket cannot be managed by it.

This behaviour only occurs when the client tries to send the unsubscribe message to a reserved topic. Therefore, the broker unsubscribes a client from a reserved topic when the operation is completed. This is not considered a problem in practice but still it is reflected here for the sake of providing a more comprehensive description.

To Do

The following two functionalities are implemented in the described EoT Control Mode Desktop application but not in Pulga:

- Implement the snapshot retrieval when the new camera is ready.
- Implement all the functionality of the flash when the conflicts between the WiFi chip and the flash memory are solved in hardware.

These functions will be implemented as hardware evolves.

9. CODE

The code of the EoT project can be found in the following GitLab repository:

https://gitlab.com/espiaran/EoT

The Pulga code can be found in the myriad applications directory of the WP3:

WorkPackage_3/myriad/apps/pulga_control_app

Pulga depends on *Crypto*, *SDCardIO*, *WifiFunctions*, and *TimeFunctions* modules. These modules can be found under the *WorkPackage_3/myriad/libs* folder.

The Java Control Mode Desktop application is stored in the following directory:

WorkPackage_3/desktop/apps/EoT_control_mode_java

This application needs the Paho library that can be found in *WorkPackage_3/desktop/libs.*

10. CONCLUSION

EoT focuses on developing an open platform for mobile embedded computer vision. The building elements have been all optimized for size and cost. Particularly, the device optimizes the processing power vs energy consumption ratio. Apart from hardware and architectural elements, software and protocols used have been also optimized. The publish/subscribe MQTT protocol has been selected early on because of its low-power profile. While typical scenarios involve (mobile) clients sending/receiving messages to/from a cloud-based broker, a novel architecture is proposed in which each EoT device can act as a broker itself. This provides a minimal way of communication that does not require any cloud-based broker. In this way no data is initially sent through the Internet which is also an advantage in terms of security. This basic form of communication can be in turn used to establish additional modes of communication should the application require it. It can, for example, be used to configure the device and the embedded application to run on it, including connection to an existing WiFi.

The proposed embedded MQTT broker, Pulga, offers the opportunity to install and configure applications in the EoT device using a computer or a mobile device with any MQTT client.

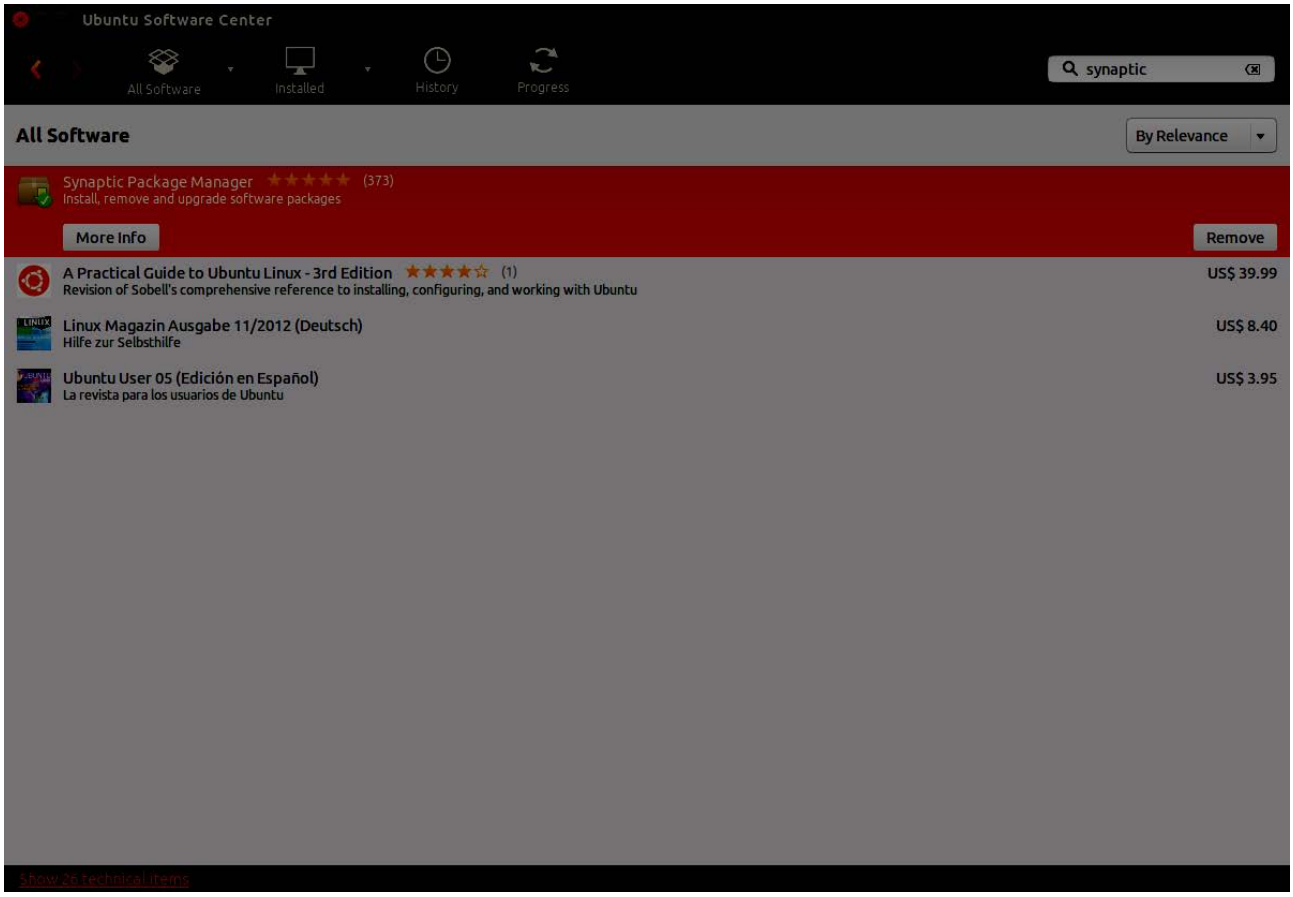
Finally, a client and the JAVA API for interacting with Pulga has been also developed including all the main functionalities of a classic MQTT broker and the new possibilities required by EoT.

11. ANNEX: PYTHON TESTS

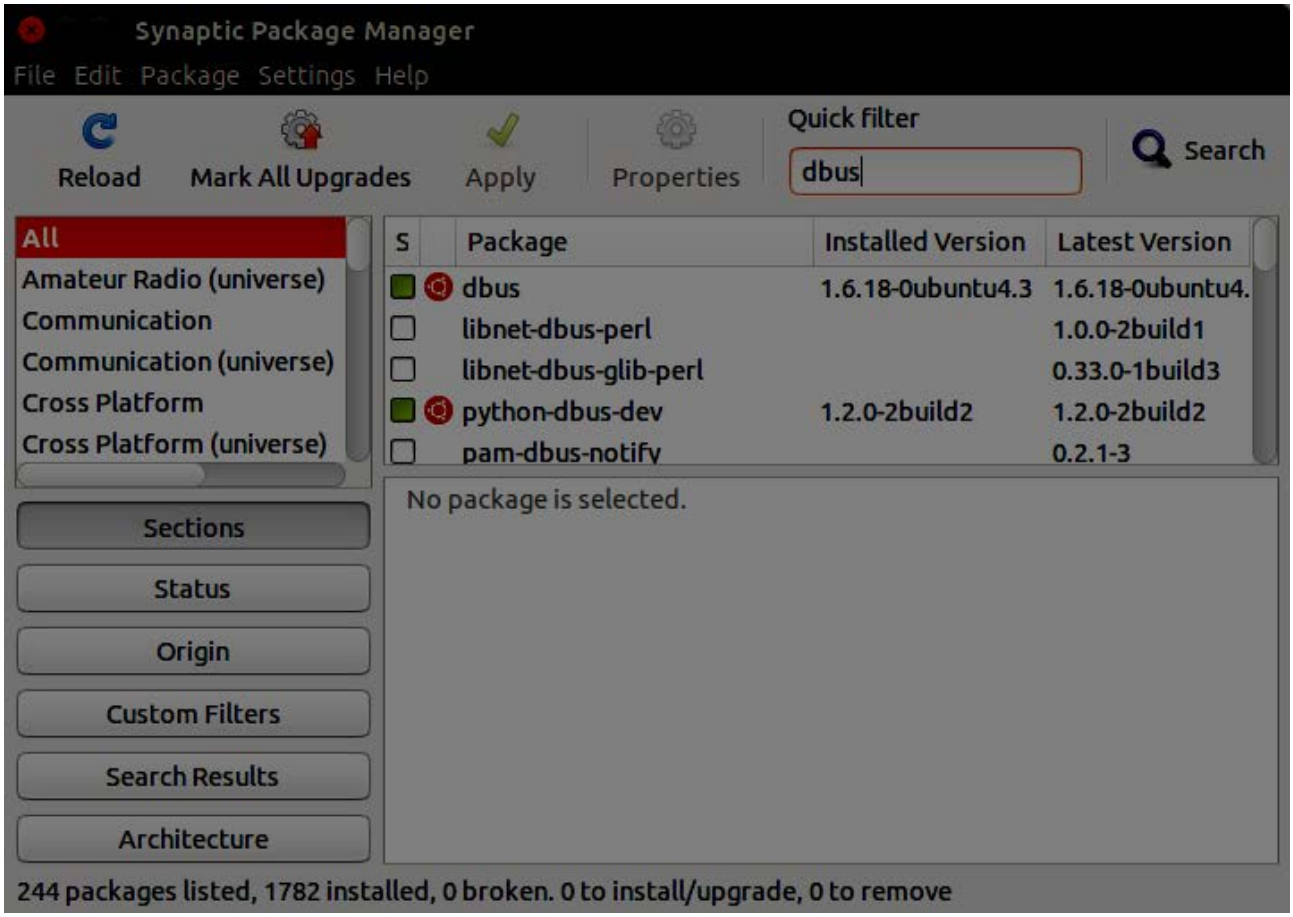
The following shows how to use an Ubuntu-live-CD distribution to run the Python tests. Note also that if the tests are carried out from inside a virtual machine the WiFi connection will fail because different interface names are used.

Run Ubuntu 14.04 LTS Live CD

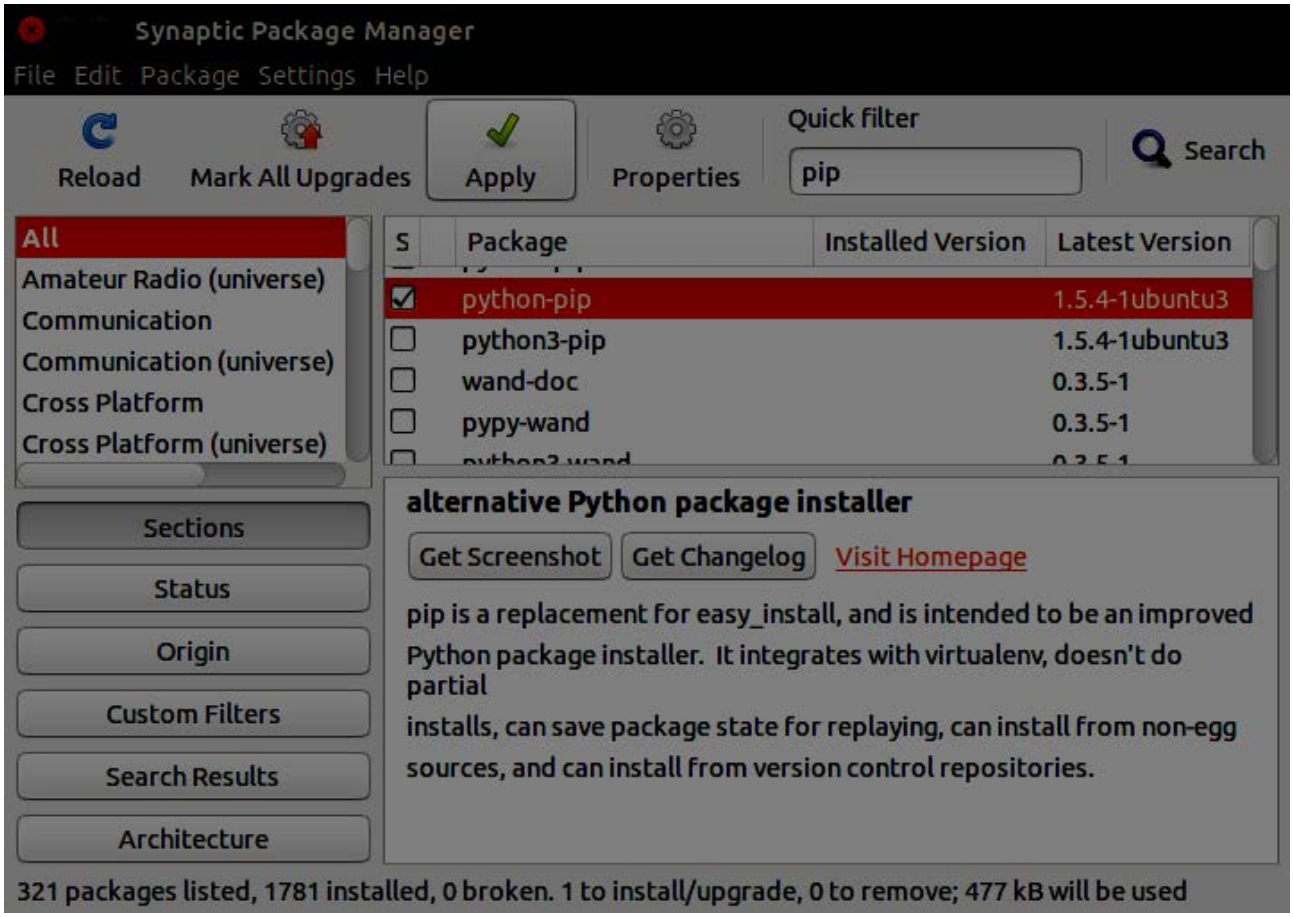
Install “Synaptic Package Manager” through the “Ubuntu Software Center”



Open Synaptic and see that dbus is installed by default



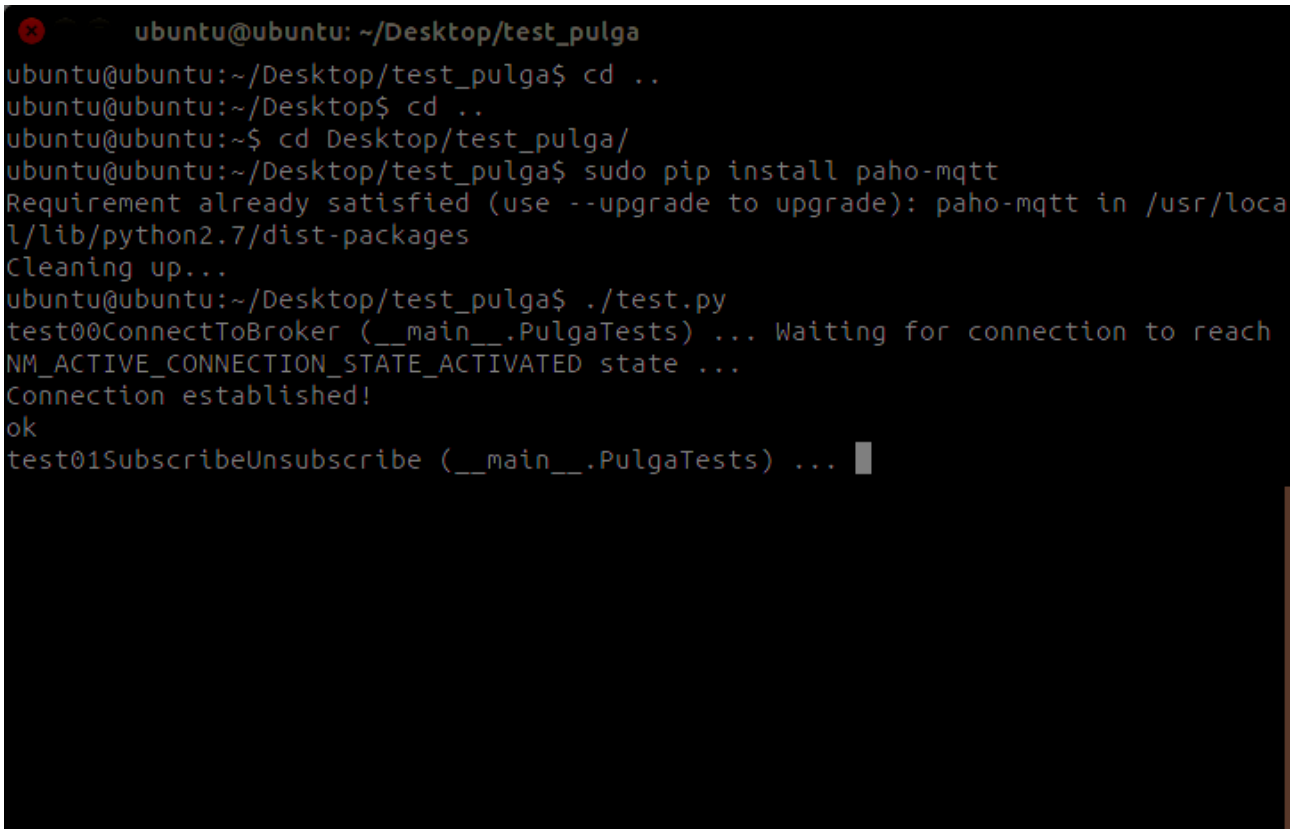
Install python-pip and all its dependencies using Synaptic



Copy the “test_pulga” folder with the python code and install “paho-mqtt” for python using:

>> sudo pip install paho-mqtt

Finally, run the test



12. REFERENCES

[1]. http://www.ti.com/product/cc3100mod?keyMatch=CC3100MOD&tisearch=Search-EN. Last accessed: 12th of January 2016.

[2]. Satyanarayanan, M., Bahl, P., Caceres, R., & Davies, N. (2009). The case for vm-based cloudlets in mobile computing. Pervasive Computing, IEEE, 8(4), 14-23.

[3]. Banks, A., & Gupta, R. (2014). MQTT Version 3.1. 1. OASIS Standard. https://www.oasis-open.org/standards. Last accessed: 12th of January 2016.

[4]. Belli, L., Cirani, S., Ferrari, G., Melegari, L., & Picone, M. (2014). A graph-based cloud architecture for big stream real-time applications in the internet of things. In Advances in Service-Oriented and Cloud Computing (pp. 91-105). Springer International Publishing.

[5]. Sutaria, R., & Govindachari, R. (2013). Making sense of interoperability: Protocols and Standardization initiatives in IOT. 2nd International Workshop on Computing and Networking for Internet of Things.

[6]. MQTT: a machine-to-machine (M2M)/Internet of Things connectivity protocol. http://mqtt.org. Last accessed: 12th of January 2016.

[7]. Mosquitto: an open source message broker that implements the MQ Telemetry Transport protocol. http://mosquitto.org/. Last accessed: 12th of January 2016.

[8]. http://eclipse.org/paho . Last accessed: 12th of January 2016.

13. GLOSSARY

AP	Access Point
API	Application Programming Interface
EoT	Eyes of Things
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
MQTT	Message Query Telemetry Transport
M2M	Machine to Machine
OASIS	Organization for the Advancement of Structured Information Standards
PC	Personal Computer
QoS	Quality of Service
SD	Secure Digital
SoC	System on a Chip
SSID	Service Set Identifier
SSL/TLS	Secure Sockets Layer / Transport Layer Security
TCP/IP	Transmission Control Protocol / Internet Protocol
WEP	Wired Equivalent Privacy
WPA	WiFi Protected Access

- End of document -