



**UNIVERSITATEA TEHNICĂ "GH ASACHI" IAȘI FACULTATEA  
AUTOMATICĂ ȘI CALCULATOARE SPECIALIZAREA CALCULATOARE ȘI  
TEHNOLOGIA INFORMAȚIEI**

**DISCIPLINA REȚELE DE CALCULATOARE - PROIECT**

# **Client MQTT - Aplicație pentru monitorizarea resurselor SO**

**Coordonator:**

Prof. Nicolae-Alexandru Botezatu

**Studenti:**

Balan Paul-Eusebiu - 1308B  
Batalan Vlad - 1308A  
Luca Răzvan Ionut - 1308B

**Iasi, 2020**

# Cuprins documentație

## Capitolul 1: Descriere generala

- 1.1. MQTT
- 1.2. Concepte de baza
- 1.3.1 Mosquitto MQTT Broker
- 1.3.2 Autentificare și autorizare cu Mosquitto

## Capitolul 2: Client MQTT

- 2.1. Informații generale
- 2.2. Implementare pachete MQTT
- 2.3. Implementare mecanism trimitere si receptionare pachete
- 2.4. Implementare mecanism Keep Alive
- 2.5. Implementare Quality of Service (QoS)
  - 2.5.1. QoS 0
  - 2.5.2. QoS 1
  - 2.5.3. QoS 2
- 2.6. Implementare mecanism Last Will
- 2.7. Autentificare cu nume utilizator și parola

## Capitolul 3: Aplicație de monitorizare a resurselor SO

- 3.1. Extragere resurse SO (psutil)
- 3.2. Date interfata (pyqt)
  - 3.2.1. View-uri
  - 3.2.2. Publish
  - 3.2.3. Subscribe
  - 3.2.4. Login
- 3.3. Publicare manuala si automata configurabile
- 3.4. Lista de abonare configurabila
- 3.5. Legătură cu clientul MQTT

## Capitolul 4: Bibliografie

# Capitolul 1: Descriere generală

## 1.1. MQTT

MQ Telemetry Transport (MQTT) este un protocol de rețea de tip publish-subscribe care facilitează transportul de mesaje între diverse dispozitive. Datorită construcției sale lightweight este ideal pentru conectarea la distanță a dispozitivelor a căror limitări impun dimensiuni reduse ale codului sursă sau utilizarea unei rețele cu o lățime de bandă limitată.

## 1.2. Concepte de bază

Protocolul MQTT definește câteva concepte de bază:

- Mesajele: sunt informația care este trimisă între dispozitive;
- Publish/Subscribe: este conceptul de bază din spatele protocolului care presupune existența a cel puțin două dispozitive, unul care să publice mesaje către un topic și un altul care să se aboneze la un anumit topic pentru a recepționa mesajele trimise;



- Topics: reprezintă o modalitate de a specifica locul unde este publicat un mesaj în cazul în care facem publish și de a selecta ce mesaje dorim să recepționăm în momentul în care facem subscribe;
- Broker: este un intermediar responsabil de primirea, sortarea și trimiterea mai departe a mesajelor către clienții abonați la diverse topicuri;

### 1.3.1. Mosquitto MQTT Broker

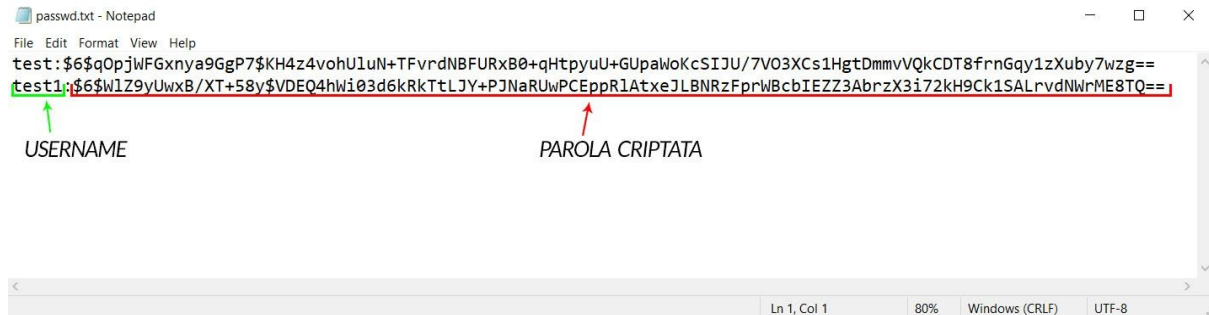
Mosquitto este un broker de mesaje care implementează protocolul MQTT. Construcția sa facilitează utilizarea broker-ului pe orice tip de device, indiferent dacă discutăm despre sisteme embeded sau servere din datacenter-uri.

### 1.3.2. Autentificare și autorizare cu Mosquitto

Broker-ul Mosquitto oferă posibilitatea configurării unor metode de autentificare și autorizare prin nume de utilizator și parolă.

În mod implicit Mosquitto nu are activată opțiunea de autentificare, fiecare utilizator care se conectează fiind un utilizator anonim. Prin modificarea anumitor parametri (*allow\_anonymous false*, *password\_file C:\Program Files\mosquitto\passwd.txt*) ai fișierului "mosquitto.conf" putem obliga Mosquitto să solicite date de autentificare în momentul conectării unui nou utilizator.

Numele utilizatorilor și parolele sunt ulterior stocate într-un fișier .txt (nume utilizator:parola). Parola este ulterior criptată în mod automat prin apelarea unei comenzi în terminal.



(fișier cu nume de utilizatori și parole)

Dacă un utilizator dorește să se conecteze folosind o parolă, trebuie să setăm pe 1 flag-ul pentru parolă din CONNECT și să așteptăm recepționarea unui pachet CONNACK care să confirme sau să infirme autentificarea. În cazul în care nu se poate confirma faptul că acea conectare a avut loc cu succes atunci utilizatorul respectiv nu are acces la serverul MQTT.

## Capitolul 2: Client MQTT

### 2.1. Informații generale

Un Client MQTT reprezintă o instanță ce se poate conecta și comunica cu un Broker MQTT utilizând diferite tipuri de pachete, fiecare având un rol distinct.

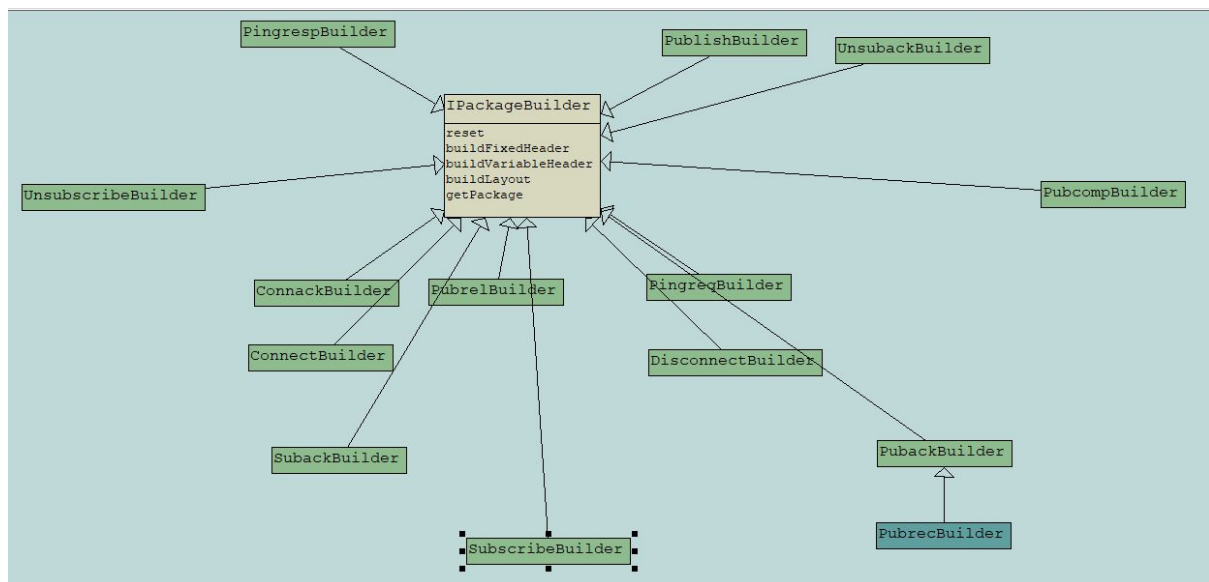
- CONNECT: utilizat pentru a loga utilizatorul la server
- CONNACK: este răspunsul serverului la primirea unui pachet de CONNECT
- PUBLISH: utilizat pentru trimiterea unui mesaj spre un anumit topic
- PUBACK: răspunsul la primirea unui pachet de PUBLISH cu QOS 1
- PUBREC: partea întâi a răspunsului la primirea unui pachet PUBLISH cu QOS 2
- PUBREL: partea a doua a răspunsului la primirea unui pachet PUBLISH cu QOS 2
- PUBCOMP: partea a treia a răspunsului la primirea unui pachet PUBLISH cu QOS 2
- SUBSCRIBE: reprezintă pachetul trimis pentru abonarea la diferite topicuri
- SUBACK: este răspunsul la primirea unui pachet de tip SUBSCRIBE
- UNSUBSCRIBE: reprezintă pachetul trimis pentru dezabonarea de la diferite topicuri
- UNSUBACK: este răspunsul la primirea unui pachet de UNSUBSCRIBE
- PINGREQ: trimis pentru a face ping spre server (utilizat și pentru sistemul keep alive)
- PINGRESP: răspunsul serverului la primirea unui pachet PINGREQ
- DISCONNECT: pachet de deconectare de la broker

Cliantul MQTT trebuie sa furnizeze un set de functionalitati recunoscute de către Broker precum: un sistem de keep alive, permiterea utilizarii celor trei nivele de calitate ale serviciilor (QOS 0,1,2), un mod de gestionare ale primirii și trimiterii pachetelor, mecanism de tip Last Will și mod de logare ca utilizator.

## 2.2. Implementare pachete MQTT

Fiecare pachet are o forma standard și este compusă din trei blocuri structurale: fixed header, variable header si payload. Conținutul acestor blocuri depinde de tipul pachetului.

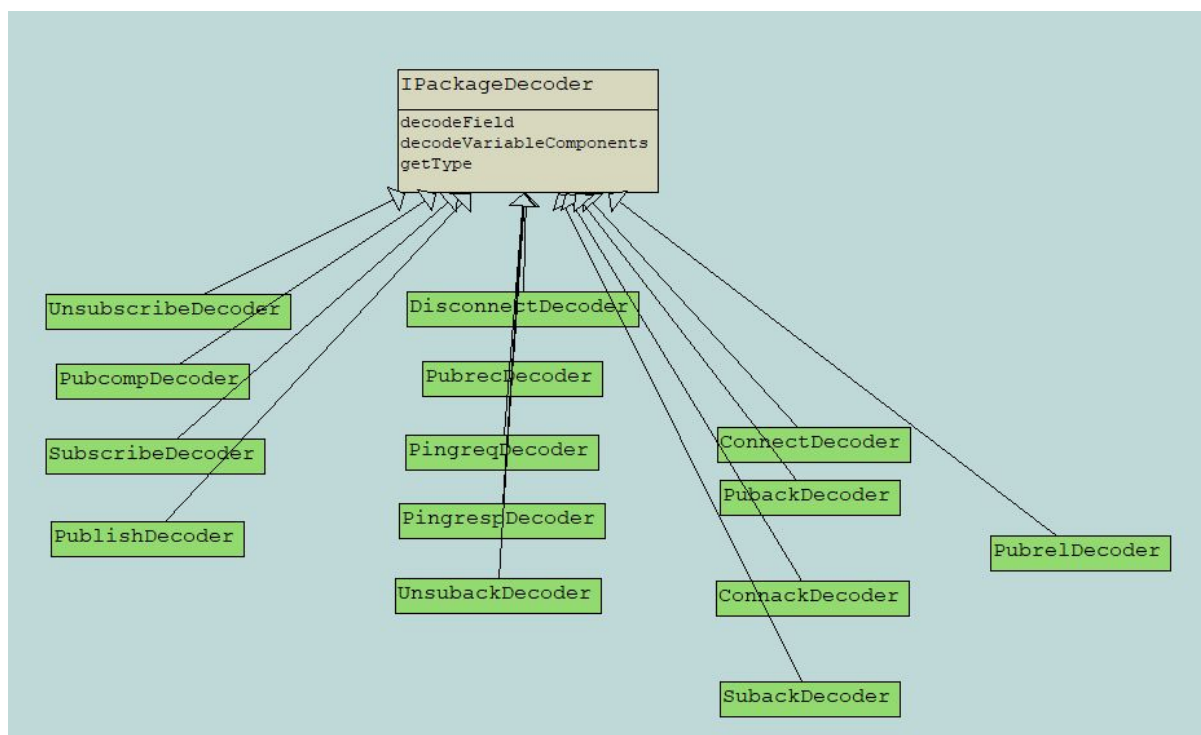
În realizarea proiectului, am decis sa utilizăm Programarea Orientată Obiect pentru a descrie fiecare tip de pachet în parte și pentru a abstractiza, pe masura ce avansam în nivel aplicație, tot ceea ce înseamnă lucru cu pachete. Pentru crearea fiecărui pachet individual, a fost utilizat Builder ca design pattern, astfel fiecare pachet are un Builder propriu care usureaza construirea acestuia.



(uml Builder pachete MQTT)

## 2.3. Mecanism de trimitere si receptionare pachete

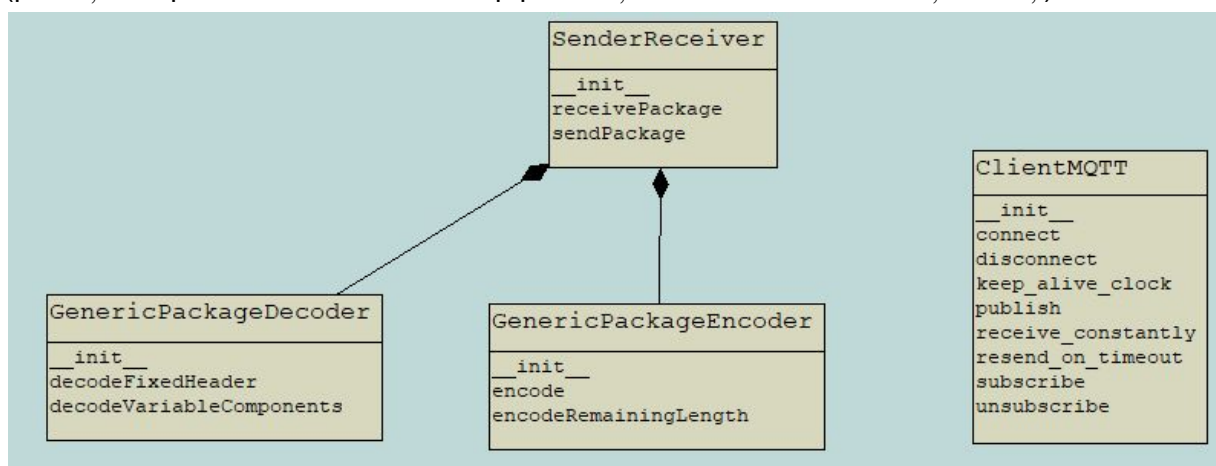
Avand in vedere ca în procesul de trimitere primire de pachete intervine și procesul de encodare, respectiv decodare a pachetului dintr-un sir binar, s-au realizat clasele GenericPackageEncoder (pentru fiecare pachet în parte realizează transformarea acestuia într-un șir de biți) și GenericPackageDecoder (pentru un șir de biți primit, acesta va alege builder-ul potrivit pentru a genera un pachet MQTT valid). Pentru partea de decodare de pachete, s-a realizat pentru fiecare tip de pachet un decoder personalizat.



(uml Decoder pachete MQTT)

Prin intermediul acestor clase, pachetele și procedeele de encodare și decodare ale acestora au fost abstractionate. Acestea vor fi folosite ca fundație pentru realizarea clasei care va fi utilizată de către programator, care îi pune la dispoziție acestuia toate funcționalitățile unui Client MQTT, numita ClientMQTT. Realizarea pachetelor și lucrul cu serverul vor fi complet invizibile pentru utilizator.

Recepția și trimiterea pachetelor este realizată prin intermediul socket-urilor în Python și procesele sunt realizate de clasa SenderReceiver care dispune de două metode: receivePackage (primește un șir de biți și returnează un obiect de tip pachet) și sendPackage (primește ca parametru un obiect de tip pachet și îl trimite brokerului ca șir de biți).



(uml ClientMQTT și clase encoder și decoder)

Trimiterea pachetelor nu reprezintă o problemă pentru clientul MQTT, totuși, primirea pachetelor a reprezentat o problemă. Inițial, primirea era gestionată în funcțiile care

realizeaza și trimiterea. De exemplu, pentru funcția de connect, după trimiterea pachetului se aștepta și primirea pachetului CONNACK în cadrul aceluiasi thread. Problema aparea atunci cand se primea receive la alt tip pachet în loc de cel așteptat.

Soluția abordată pentru primirea pachetelor este un thread separat care primește toate pachetele de la Broker și, în funcție de tipul acestuia, va lua deciziile necesare cu privire la acțiunile care trebuie făcute. În cadrul clasei ClientMQTT, funcția care realizeaza primirea pachetelor se numește **receive\_constantly** și este rulat într-un thread separat care este activ pe toata durata de viata a obiectului ClientMQTT.

```
def receive_constantly(self):
    while self.loop_flag is True:
        if self.loop_flag is True:
            # print("Prepare to receive ...")
            try:
                package_recv = self.transmitter.receivePackage()
                # print("Received Package Type = " + package.getType())
                # displayControlPackageBinary(package)

                if not isinstance(package_recv, IControlPackage):
                    continue

                package_type = package_recv.getType()
```

(funcția care primește toate pachetele)

## 2.4. Implementare mecanism Keep Alive

Mecanismul de Keep Alive este unul specific pentru protocolul MQTT și presupune un timeout ce se reseteaza de fiecare data cand clientul trimite un pachet. Dacă între doua pachete trimise a trecut mai mult timp decat indicatorul Keep Alive, Brokerul va interpreta inactivitatea clientului și îl va deconecta.

Keep Alive este transmis serverului prin intermediul unui pachet de tip CONNECT și este localizat în partea de variable header a pachetului, byte-ul 9 și 10, reprezentand un număr de secunde. Dacă este setat cu valoarea 0, atunci Brokerul va ignora functionalitatea aceasta.

Pentru a menține conexiunea, clientul trebuie sa comunice constant cu Brokerul prin pachete de tip PINGREQ.

În realizarea acestei funcționalități, clientul MQTT prezintă o funcție membru numită **keepAliveClock**, care, cât timp flag-ul `keep_alive_flag` este `True`, va trimite regulat la intervale de timp mai mici decât indicatorul Keep Alive asociat clientului pachete de tip PINGREQ către Broker. Clientul MQTT realizat, în momentul când a primit un pachet de confirmare conectare (CONNACK), va inițializa un nou thread și îl va rula cât timp clientul va rămâne conectat la Broker.

```
107     def keep_alive_clock(self):
108         while self.keep_alive_flag is True:
109             wait_time = self.keep_alive / 2
110             step = wait_time / 10
111             while wait_time > 0 and self.keep_alive_flag is True:
112                 time.sleep(step)
113                 wait_time -= step
114
115             if self.keep_alive_flag is True:
116                 # print("Ping sent!")
117                 # send ping
118                 builder = PingreqBuilder()
119                 builder.reset()
120                 builder.buildFixedHeader()
121                 builder.buildVariableHeader()
122                 builder.buildPayload()
123                 ping = builder.getPackage()
124                 self.transmitter.sendPackage(ping)
```

(cod funcție responsabilă cu menținerea Keep Alive)

## 2.5. Implementare Quality of Service (Qos)

### 2.5.1. Qos 0

Reprezintă cel mai de jos ca nivel tip de serviciu oferit. În acest caz, un mesaj este transmis odată și o singură dată. Nu se așteaptă nicio confirmare din partea receptorului, astfel pierderea acestuia nu reprezintă un punct de interes.

În program, se realizează prin simpla transmitere a unui pachet de tip PUBLISH cu QoS 0 către Broker.



```

def publish(self, topic, message, QoS, retain=0):
    self.packedId += 1

    # create connect package
    builder = PublishBuilder()
    builder.reset()
    builder.buildFixedHeader(DUP=0, QoS=QoS, RETAIN=retain)
    builder.buildVariableHeader(topic=topic, packetId=self.packedId)
    builder.buildPayload(message)

    publishPackage = builder.getPackage()

    if self.logs_flag is True:
        print("Publish[" + str(self.packedId) + "] sent:")
        print("\t[" + topic + "]: \" + message + "\""
        self.transmitter.sendPackage(publishPackage)

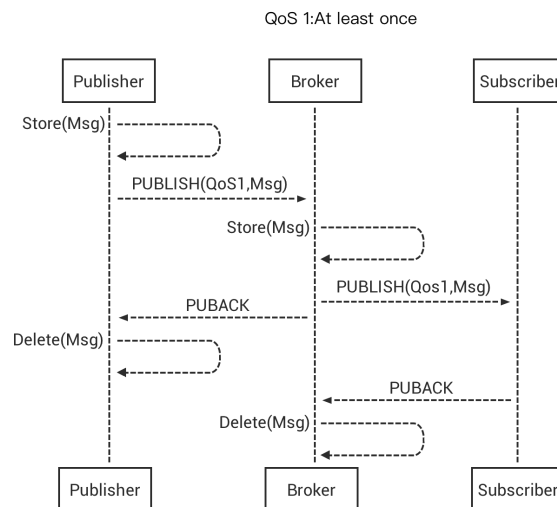
    # preparing the thread that assures the resending of the message if there was a problem
    if QoS > 0:
        self.unconfirmed[self.packedId] = publishPackage
        threading.Thread(target=self.resend_on_timeout, args=[publishPackage]).start()

```

(cod trimitere pachet PUBLISH de catre client spre server)

## 2.5.2. QoS 1

Trimiterea unui pachet va fi reluata în cazul în care se atinge un timeout, iar pachetul de acknowledge (PUBACK) nu a fost receptat.



(schema trimitere/receptie pachet PUBLISH cu QoS 1)

Mecanismul de retransmitere al unui pachet pana la recepția unui acknowledge potrivit, indiferent ca pachetul retrimis este un PUBLISH, PUBREL sau un PUBREC (ultimele 2 fiind discutate în partea de QoS 2) este realizat în cod prin intermediul funcției **resend\_on\_timeout**, acesta primind ca parametru pachetul care trebuie sa fie confirmat (depozitarea pachetului cat timp nu a fost recepționat pachetul potrivit de confirmare, care are același packet\_id). Funcția va verifica mereu dicționarul de pachete neconfirmate pentru a verifica daca id\_package a pachetului depozitat încă exista. În caz contrar, flag-ul done va fi setat pe True și va realiza ieșirea din bucla.

La primirea sau trimiterea unui pachet care necesita confirmare, în lipsa acesteia, se construiește un thread separat în care este rulată funcția **resend\_on\_timeout**.

```
# preparing the thread that assures the resending of the message if there was a problem
if QoS > 0:
    self.unconfirmed[self.packedId] = publishPackage
    threading.Thread(target=self.resend_on_timeout, args=[publishPackage]).start()
```

(thread de retransmitere a pachetului pana la primirea confirmării)

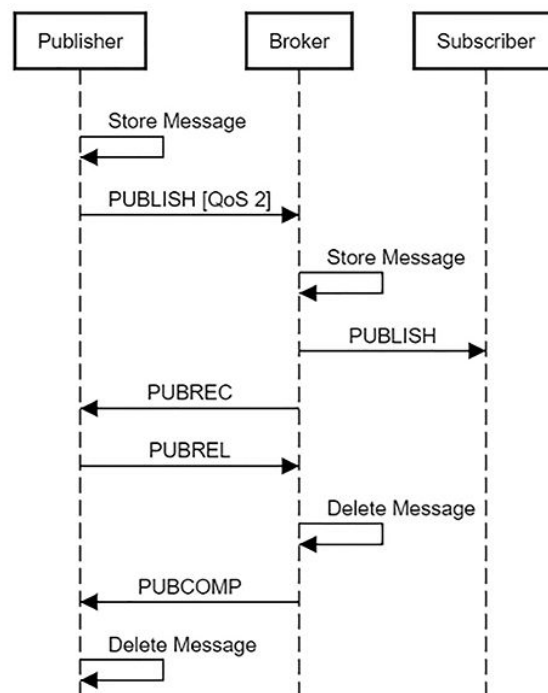
```
126 # this method is used for qos purpose
127 # it gets a packet_id, and checks every some seconds the
128 # unconfirmed dictionary
129 # if after the timeout, packet_id key still exists, resend the package
130 def resend_on_timeout(self, unconfirmed_package):
131     packet_id = unconfirmed_package.getVariableHeader().getField("packet_id")
132     packet_type = unconfirmed_package.getType()
133
134     # in case of publish, we change the package: dup = 1
135     if packet_type == 3:
136         current_flags = unconfirmed_package.getFixedHeader().getFlags()
137         unconfirmed_package.getFixedHeader().setFlags(current_flags | 8)
138
139     # set the timeout
140     done_flag = False
141     timeout = 3
142     step = timeout / 10
143
144     while done_flag is False:
145         index = timeout
146         exist_flag = True
147
148         while index > 0:
149             time.sleep(step)
150
151             # check if package is not in unconfirmed
152             if packet_id not in self.unconfirmed.keys():
153                 exist_flag = False
154                 break
155             else:
156                 # packet_id exists, but has another type
157                 existing_packet_type = self.unconfirmed[packet_id].getType()
158
159                 if packet_type != existing_packet_type:
160                     # special for qos2
161                     exist_flag = False
162
163             index -= step
164
165     # check the exist_flag
166     if exist_flag is True:
167         # we need to resend the package
168         if self.logs_flag is True:
169             print("Due to timeout, resend: " + str(unconfirmed_package.getType()) + " package type")
170         self.transmitter.sendPackage(unconfirmed_package)
171     else:
172         done_flag = True
```

(cod retransmitere pachete neconfirmate)

### 2.5.3. QoS 2

QoS 2 reprezinta cel mai inalt serviciu de trimitere a mesajelor. Acesta se oferă cea mai mare siguranta ca mesajul a ajuns la destinatie. Sistemul de retransmitere a mesajelor, prezentat la subcapitolul 2.5.2, este refolosit pentru a susține și QoS 2.

În funcția de primire a tuturor pachetelor de la Broker, sunt luate în considerare și pachetele necesare realizării unei comunicări cu QoS 2: PUBREC, PUBREL, PUBCOM. Indiferent de direcția comunicării, primirea unui astfel de pachet are ca efect stergerea pachetului de același id din lista de pachete neconfirmate, fapt care oprește thread-ul de retransmitere a pachetelor (**resend\_on\_timeout**).



(schema trimitere pachet PUBLISH cu QoS 2)

## 2.6. Implementare mecanism Last Will

Protocolul MQTT are prevăzut un mecanism care, în cazul când un dispozitiv (client) are o problemă și se deconectează involuntar (fără a trimite un pachet DISCONNECT), dacă acesta a setat la conectare un mesaj de Last Will, acesta va fi acum afișat. Sistemul este foarte util deoarece unui complex de senzori, este dificil de identificat cel care nu mai funcționează manual și atunci este mult mai simplu să se consulte fișierul de log-uri pentru a afla direct senzorul cu probleme, utilizând mesajul său de Last Will.

În cazul proiectului, sistemul este implementat în funcția de connect. În parametrii pe care îi primește funcția, utilizatorul va specifica și flagurile necesare, iar dacă acesta dorește Last Will, trebuie să seteze bitul notat în poza de mai jos cu (5).

```
# connect flags
# username(0) | password(1) | will retain(2) | will Qos(3 - 4) | will flag(5) | cleanSession(6) | reserved(7)
self.variableHeader.setField("connect_flags", int(connectFlags, 2), 1)
```

(package\_builders.py setarea flagurilor pachetului Connect)

## 2.7. Autentificare cu nume de utilizator și parola

În cazul autentificare, funcția de **connect** a clientului va primi ca parametrii numele și parola acestuia și va genera un pachet de CONNECT care va fi trimis la server. Dacă exista conexiune, serverul va trimi un pachet CONNACK care va conține codul de return al cererii de conectare.

Valoare cod return	Descriere
0x00	Pachetul de conectare a fost acceptat
0x01	Serverul nu suporta nivelul de MQTT
0x02	Client ID-ul este corect UTF-8 dar nu este primit pe server
0x03	A reușit conectarea, dar serviciul MQTT nu este valabil
0x04	Data din numele utilizatorului sau parola nu are un format corespunzător
0x05	Clientul nu este autorizat sa se conecteze pe server

Pentru a putea fi înștiințat utilizatorul clientului MQTT de codul primit de la pachetul CONNACK receptat prin intermediul funcției **recv\_constantly**, s-a utilizat un obiect de tip Condition care oprește temporar thread-ul care a apelat funcția **connect** din client în așteptarea unui semn de la **recv\_constantly** care atestă primirea CONNACK-ului și da notify la thread-urile care așteaptă.

```
447         self.condition.acquire()
448
449         # waiting for an answer
450         self.condition.wait(timeout=10)
451
452         tmp_return = self.connack_return_code
453         self.connack_return_code = None
454         self.condition.release()
455
456         # returning the answer
457         if tmp_return is None:
458             return -1
459         return tmp_return
```

(functia connect: așteptarea codului de return din alt thread)

```
188 # CONNACK PACKAGE
189 if package_type == 2:
190     self.condition.acquire()
191     return_code = package_recv.getVariableHeader().getField("connect_return_code")
192
193     if return_code == 0:
194         if self.logs_flag is True:
195             print("Connected successfully!")
196         # set keep alive thread
197         if self.keep_alive != 0:
198             # self.conn.settimeout(self.keep_alive)
199             self.keep_alive_flag = True
200             self.ping_thread.start()
201
202     else:
203         if self.logs_flag is True:
204             print("Connection failed! Return code = " + str(return_code))
205
206     self.connack_return_code = return_code
207     self.condition.notify()
208     self.condition.release()
```

(preluarea codului de return din CONNACK)

## Capitolul 3: Aplicație de monitorizare a resurselor SO

### 3.1.Extragere resurse SO (psutil)

Psutil (process and system utilities) este o bibliotecă cross-platform utilizată pentru a primi informații despre diferiți parametri ai sistemului de operare , precum procesele curente și CPU, memorie,disk-uri,network-uri , în Python. Este folosit în principal pentru monitorizarea sistemului și limitarea resurselor proceselor dar și pentru managementul proceselor ce rulează.

Psutil poate fi folosit și pe Windows dar nu are toate funcționalitățile .

Din biblioteca psutil am folosit funcțiile :

- **Cpu\_percent** - returnează date despre utilizarea procesorului la momentul apelării
- **Cpu\_freq** - folosim componenta current pentru a găsi frecvența curentă a procesorului
- **Virtual\_memory** - returnează mai multe date legate de memoria calculatorului, dar noi vom folosi doar procentul
- **Disk\_usage** - funcția disk usage returnează date despre un disk dat ca parametru. Noi folosim această funcție pe o listă ce reprezintă toate disk-urile curente , astfel putem face o sumă ce reprezintă utilizarea totală a disk-urilor.



## 3.2.Date interfata (pyqt)

Am folosit PyQt5 pentru a crea interfața. PyQt5 este un modul foarte puternic ce include multe clase pentru a face construirea interfeței ușoară. În clasa GUI am inițializat toate variabilele ce compun interfața. Acestea sunt de mai multe categorii , dintre care cele mai importante sunt:

- Widgetul principal ce reprezintă fereastra inițială
- Widgeturile secundare , ferestrele de Login , Subscribe și Publish
- Butoane , liste , label-uri folosite pentru afișarea datelor și apelarea funcțiilor corespunzătoare

Widgetul principal a fost folosit ca și container pentru toate componentele interfeței . Acesta este mereu vizibil și nu are atribuții speciale legate de aplicație.

Pentru a crea view-urile am folosit funcțiile hide si show de care dispun widget-urile secundare. Astfel în funcție de starea aplicației vom vedea doar componentele utile din interfață.

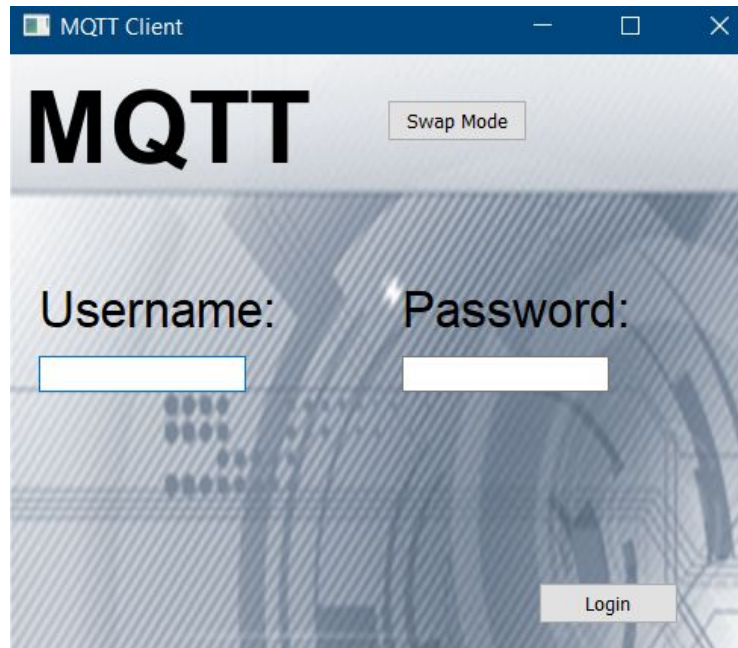
### 3.2.1.View-uri

În aplicație avem 3 view-uri Login ,Publish și Subscribe. Fiecare view corespunde unei stări în care se poate afla aplicația . Aceste view-uri sunt reprezentate de 3 widgeturi fiecare cu componentele sale utile după cum urmează.



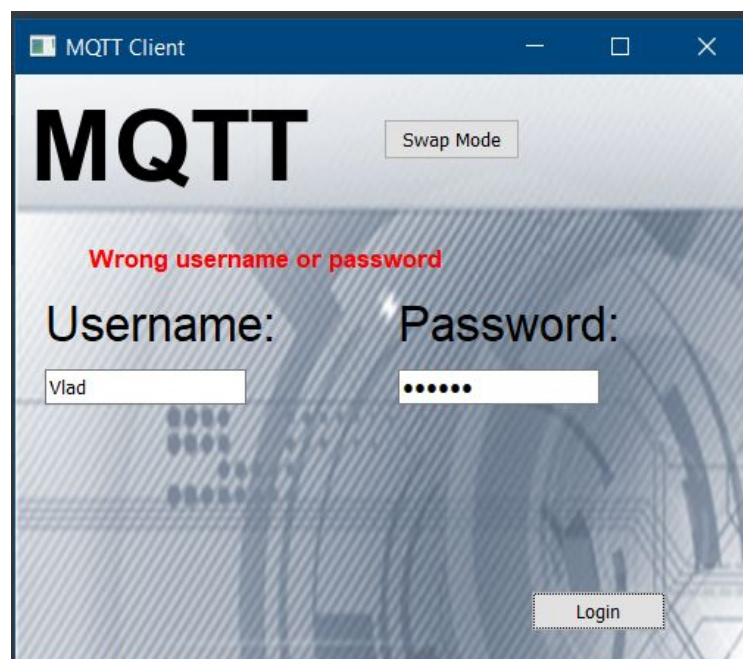
### 3.3.2.Login

Fereastra de Login este prima stare în care se poate afla aplicația. Aceasta are un buton de login care ne va duce în starea următoare (by default Subscribe) , 2 căsuțe pentru introducerea datelor și label-uri pentru datele introduse.

The image shows a window titled "MQTT Client" with a blue header bar. Below the header, the word "MQTT" is displayed in large, bold, black letters. To the right of "MQTT" is a button labeled "Swap Mode". Below this, there are two labels: "Username:" and "Password:". Under "Username:" is a text input field. Under "Password:" is a text input field. At the bottom right of the form area is a button labeled "Login". The background of the window has a faint, abstract pattern.

(pagina Login)

Datele introduse sunt verificate de aplicație iar în cazul în care acestea sunt greșite un label va apărea textul de eroare în funcție de cazul eronării.

The image shows the same "MQTT Client" window as before, but with an error message. Above the "Username:" and "Password:" labels, the text "Wrong username or password" is displayed in red. The "Username:" input field now contains the text "Vlad". The "Password:" input field contains seven dots. The "Login" button is still present at the bottom right.

(pagina Login eroare username sau parola gresita)

Fereastra de Login conține și un buton pentru schimbarea modului din Light în Dark. Diferența dintre cele două moduri constă în schimbarea culorii fundalului și a culorii label-urilor.



(pagina Login varianta Dark Mode)

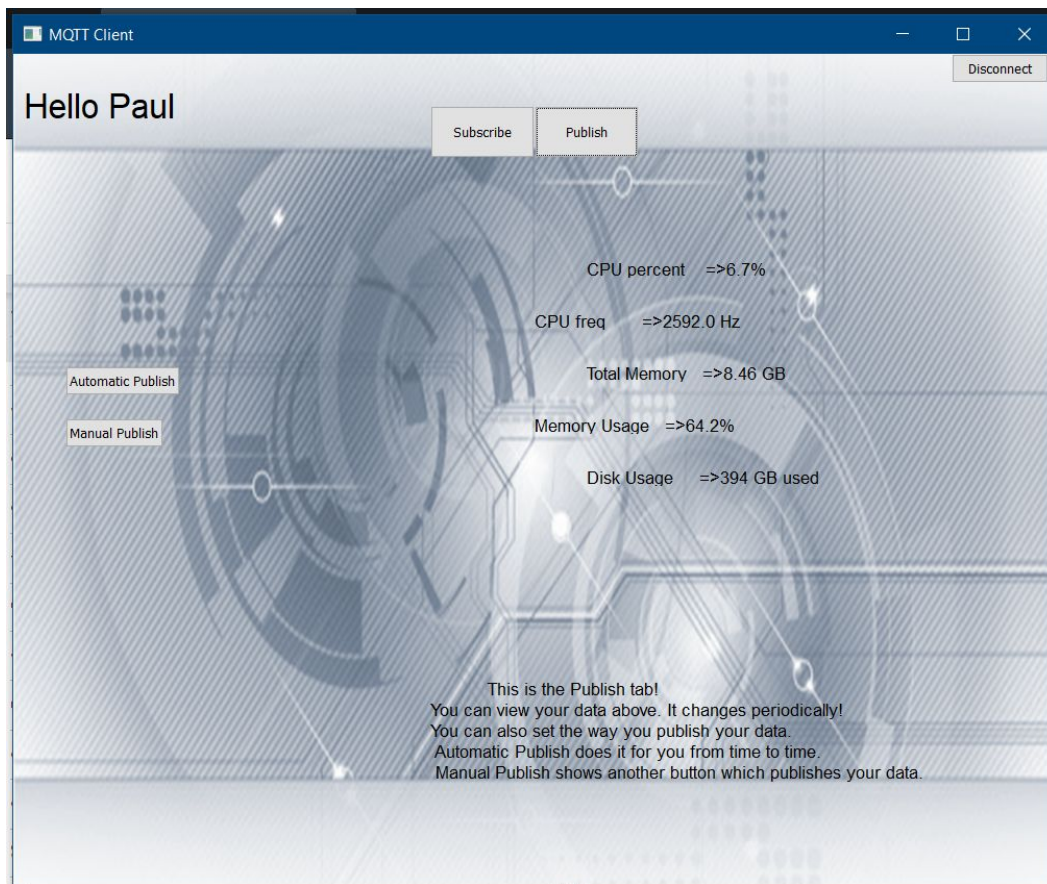
### 3.2.3.Publish

Fereastra de Publish conține 2 zone de interes. Prima zonă conține labelurile care afișează datele clientului. Aceste date sunt updatate la un interval de timp dat (by default 5 secunde). A doua zonă conține butoanele pentru setarea tipului de publish. Acesta poate fi automatic sau manual. Publish automatic va trimite date la un interval de timp iar cel Manual face posibilă apăsarea butonului "Manual publish" ce va trimite datele o singură dată. După activarea unui mod acel buton va fi colorat pentru a semnaliza tipul de publish și a face mai ușoară folosirea aplicației.

Tab-ul de Publish conține și un label Tips care arată câteva idei despre flow-ul de utilizare.

În tabul de publish avem și butonul Disconnect, din trunchiul comun al aplicației, care va deconecta utilizatorul.





(pagina Publish)

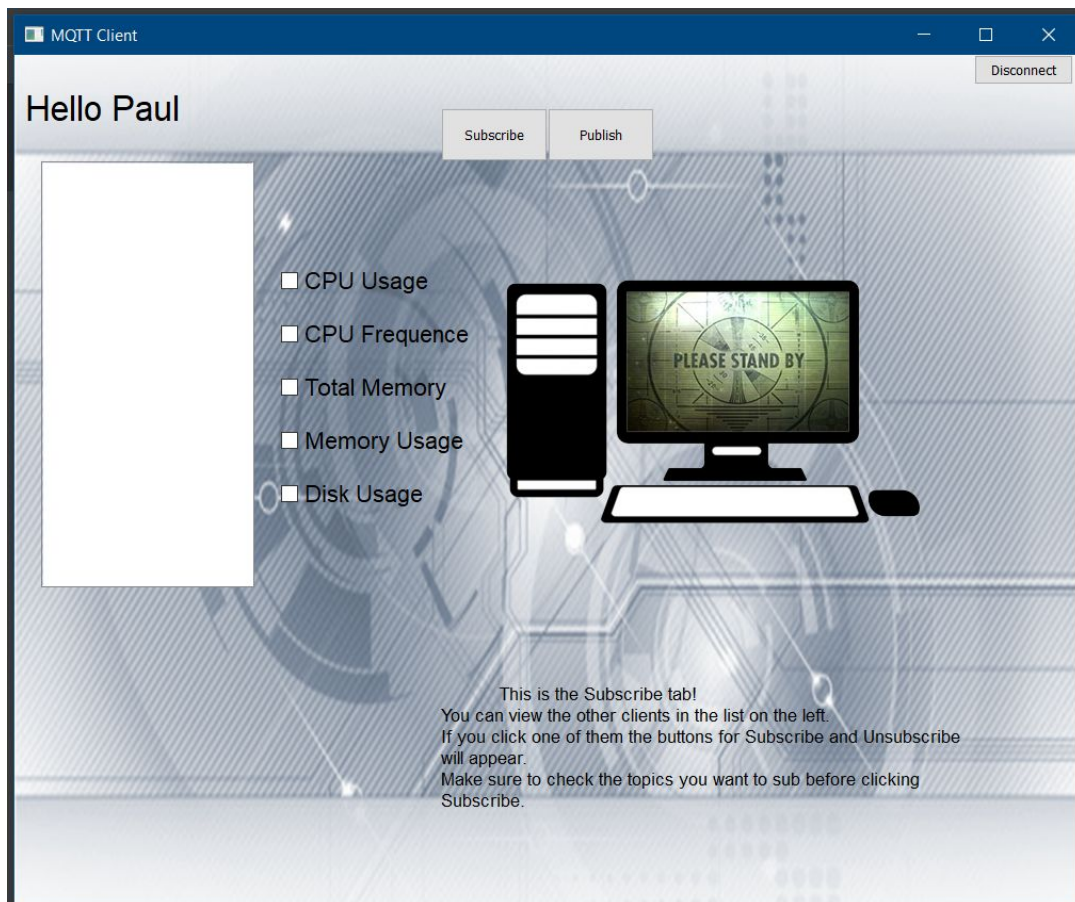
### 3.2.4.Subscribe

Fereastra de Subscribe conține 3 zone de interes și zona de Tips care prezintă flow-ul aplicației.

Prima zonă constă în afișarea listei de clienți activi în aplicație . Această listă se updateaza la conectarea sau deconectarea utilizatorilor . În listă clienții vor apărea cu roșu dacă utilizatorul aplicației nu este subscribed la acel client și cu verde în caz contrar.

A doua zonă pune la dispoziție o serie de checkbox-uri care vor fi folosite pentru a marca subtopic-urile la care vrem să dăm subscribe.

A treia zonă este diferită în funcție de ultimul client pe care am dat click. Dacă suntem subscribed la cel puțin un subtopic al acelui client vom vedea datele cerute . În caz contrar vom vedea o imagine.



(pagina Subscribe)

### 3.3.Publicare manuala si automata configurabile

Tab-ul publish conține butoanele pentru schimbarea tipului de publicare a datelor. În cazul publish-ului.

Automatic ne folosim de un alt thread care va folosi o funcție ce implementează un timer. Funcția autoPublish se folosește de doua variabile în rularea ei. Avem flag-ul de mod publish care poate întrerupe acest timer dacă este nevoie . În cazul în care acest flag rămâne activ funcția va rula cu un pas de 0.1 iar o dată la 5 secunde dacă flag-ul este încă activ se va face publish.

```

# functie care trimite datele clientului o data la x secunde
def autoPublish(self):
    timer = 5 # timer original 30
    step=0.1
    while self.autoFlag:
        index = timer
        while index > 0 and self.autoFlag:
            time.sleep(step)
            index -= step
        if self.autoFlag:
            self.sendSpecs()
    # updates your specs from "data" widget every 30 s

```

(cod thread de auto-publish)

Atunci când acest timer trebuie resetat va da publish la datele clientului prin funcția sendSpecs ce generează un pachet de Publish .Tipul Manual apelează funcția de publish o singură dată a datelor .

Aceste două tipuri nu pot exista simultan deoarece folosim un flag pentru a stabili modul de publish.



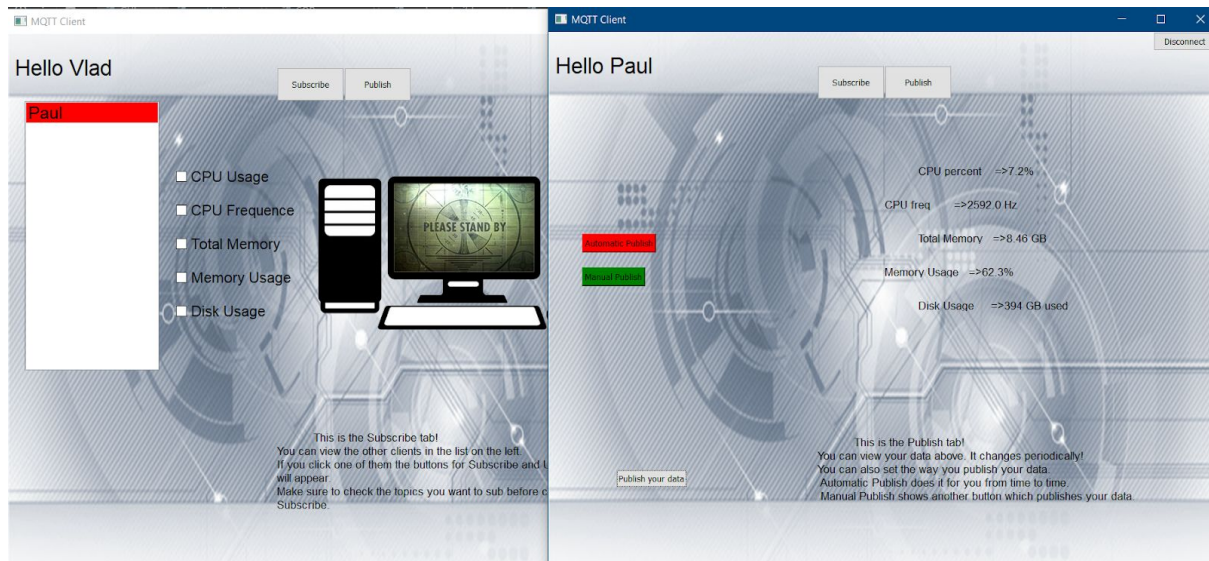
(flag-uri pentru setarea modului de publish)

### 3.4.Lista de abonare configurabilă

Atunci când un client dă Publish (de orice tip) devine vizibil în listele clienților activi. Inițial unsubscribed.Când un client dorește să dea subscribe unui alt client va da click pe numele acestuia , va bifa topic-urile de interes iar apoi poate apăsa pe butonul de subscribe. După aceasta se va genera un pachet Subscribe în funcție de lista de bife .

Dacă nu s-a bifat nici un topic atunci un label de eroare va apărea și nici un pachet nu va fi generat.

Atunci când un client se deconectează neașteptat acesta va dispărea din listele tuturor clienților .



(populare lista de abonare)

Lista de clienți este reținută și updatată cu ajutorul unei clase secundare numită User . Această clasă conține numele utilizatorului , o serie de flag-uri și variabile ce rețin datele utilizatorului și un flag de subscribe. Pe lângă acestea există și funcții ajutătoare precum get și set pentru date , unsub\_all care resetează toate datele și flag-urile utilizatorului sau isSubbed care returnează starea de subscribe a utilizatorului față de clientul respectiv.

### 3.5. Legătură cu clientul MQTT

Clasa GUI conține o variabilă de tip Client prin care se face legătura cu brokerul cât și apelarea de funcții precum publish sau subscribe , ce generează pachetele aferente în mod corespunzător. De asemenea folosim un flag de conectare prin care monitorizăm conexiunea clientului la broker.

## Capitolul 4: Bibliografie

- <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>
- <https://mosquitto.org/documentation/>
- <https://psutil.readthedocs.io/en/latest/>
- <https://doc.qt.io/qtforpython/>
- <https://docs.python.org/3/library/socket.html>

•