Slovenská technická univerzita
Fakulta elektrotechniky a informatiky

Mosquitto Telemetry Transport protokol pre IoT

Tímové zadanie z predmetu Inteligentné Mechatronické Systémy

Zimný semester 2017
2. ročník ING
ÚAMT FEI STU

Andrej Bielik, 72780
Michal Grega, 72815
Peter Halász, 72816

Obsah

1 Základné vlastnosti MQTT

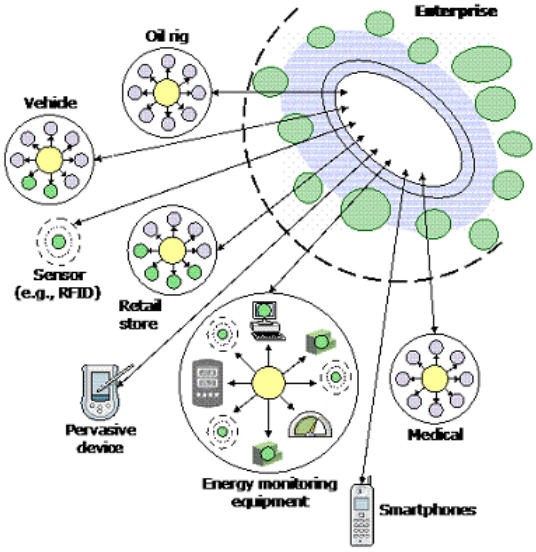
Mosquitto Telemetry Transport protokol (*MQTT*) je sieťový, komunikačný protokol, ktorý na výmenu dát medzi zariadeniami používa princíp *publish/subcribe*. V minulosti sa *MQTT* protokol nazýval *MQ Integrator SCADA Device* protokol. *MQTT* protokol je navrhnutý tak, aby bol ľahký, jednoduchý a nenáročný na implementáciu. Charakteristickou vlastnosťou protokolu *MQTT* je to, že jeden server môže obsluhovať veľké množstvo klientov, ktorí so serverom komunikujú a sú s ním spojení. Vďaka týmto vlastnostiam je možné protokol *MQTT* používať v obmedzených prostrediach a tiež v sieťach, ktoré disponujú nízkou šírkou pásma, obmedzenými možnosťami spracovania, malými pamäťovými kapacitami, a vysokou latenciou. Návrh *MQTT* minimalizuje požiadavky na šírku pásma siete a zabezpečuje spoľahlivosť doručenia pri posielaní dát medzi serverom a klientom.

Postupom času, ako sa technológia vyvíja, stále viac a viac zariadení sa medzi sebou komunikuje cez internet, čo sa v súčasnosti nazýva *Internet Of Things* (Internet vecí alebo skrátene *IoT*). Snahou je vytvoriť plne autonómnu komunikáciu medzi zariadeniami pre zvýšenie ich efektivity.

Vďaka *MQTT* protokolu je možné spojiť všetky vzdialené zariadenia, medzi ktorými je možná výmena informácií a dát. Vlastnosť, ktorú MQTT podporuje, *XR Component of WebSphere MQ*, dovoľuje zariadeniam, ktoré bežia na okraji sieťového pripojenia, komunikovať s ďaleko dostupnými zariadeniami efektívne a bezpečne. Patria sem napríklad zariadenia ako: inteligentné merače energie, telefóny, autá, vlaky, satelitné stanice a zdravotnícke zariadenia. Tieto zariadenia je možné prepojiť a sú schopné posielať si dáta cez senzory do centrálnych systémov.

Vstavané senzorové prístroje dokážu merať stav zariadení a sú schopné vymieňať si tieto dáta medzi inými zariadeniami. Vďaka tomu je možné analyzovať a reagovať na dáta z rôznych prístrojov v reálnom čase. Týmito zariadeniami môžu byť napríklad inteligentné zariadenia, ktoré môžu mať nízku šírku pásma. Typickým zariadením sú čipy rádio-frekvenčnej identifikácie (skrátene *RFID*). Tieto čipy sú bežne zakomponované napríklad v ID kartách. Používanie takéhoto malého hardvéru, ktorý môže využívať protokol *MQTT* na pripojenie k internetu, umožňuje každodenným objektom prenášať dáta z jedného do druhého. Tieto drobné zariadenia potom publikujú správy, ktoré používajú protokol *MQTT* a sú prijaté *subscriber*-om a môžu spustiť nejakú udalosť na inom zariadení.

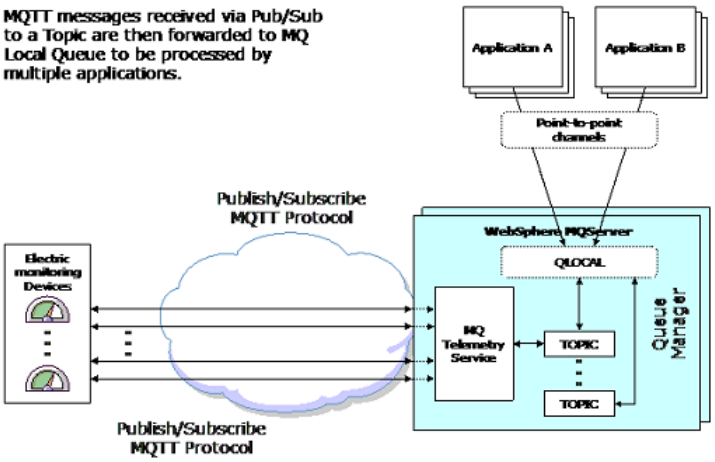
Niektoré veľmi užitočné príklady, ktoré využívajú protokol *MQTT* môžu byť používané na hardvérových čipov na prenos informácií a monitorovanie zdravotnej starostlivosti. Protokol *MQTT* funguje na princípe *machine-to-machine* (skrátene *M2M*) komunikácie. Na obrázku 1 je vidno príklad tejto komunikácie.



Obr. 1: Príklad využitia *MQTT* protokolu

Robustný produkt *WebSphere MQ* poskytuje bezpečné funkcie pre zasielanie správ, ktoré môžu byť posielané medzi veľkým množstvom klientov a inými zariadeniami v rámci nejakého podniku, alebo tam, kde to prostredie dovoľuje. *WebSphere MQ* s *MQTT* umožňuje škálovateľnosť , spoľahlivosť a bezpečnosť na širokej škále platforiem.

Základná architektúra *MQTT* je architektúra *Pub/Sub*, zatiaľ čo *WebSphere MQ* môže poskytovať buď správu typu *Pub/Sub* alebo doručenie správy z jedného miesta na druhé. Správy prijaté od klientov *MQTT* je možné presmerovať na fronty správ, ktoré sú následne použité na spracovanie nejakých požiadaviek. Začlenením funkcie *MQ* do protokolu *MQTT*, sa správy z *MQTT* protokolu *prekonvertujú* na správy *MQ* s využitím bezpečného doručenia *MQ* na overenie, či sa správy doručujú raz, a raz bez ohľadu na chyby alebo problémy so sieťou. Architektúru *WebSphere* je možné vidieť na obrázku 2.

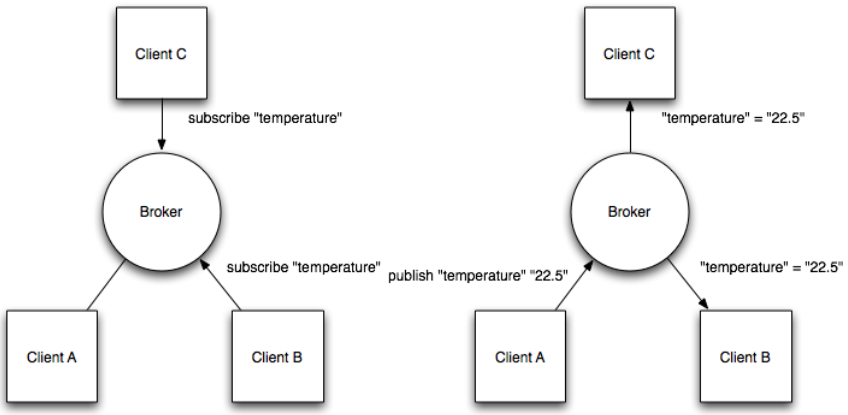


Obr. 2: *WebSphere* architektúra

1.1 Princíp komunikácie pomocou MQTT protokolu

Mosquitto Telemetry Transport protokol (skrátene *MQTT*) je komunikačný protokol využívajúci *Publisher/Subscriber* hierarchiu na báze posielania správ (*messages*), ktoré su priradené témam (*topic*).

Princíp komunikácie pomocou MQTT protokolu je postavený na publikovaní správy *publisher*-om a odoberaní celých *topick*-ov *subscriber*-om. Všetci klienti, či publikovatelia alebo odberatelia sú pripojený k takzvanému sprostredkovateľovi (*broker*), ktorý pôsobí ako jednoduché rozhranie ku ktorému je jednoduché sa pripojiť. Preto je jednoduché pridávať *publisher*-ov. Príklad takejto hierarchie môžeme vidieť na obrázku 3.



Obr. 3: Hierarchia MQTT

Ako vidno, klient A je *publisher*, ktorý publikuje informáciu o teplote a klienti B a C sú odberatelia, ktorý túto informáciu dostanú.

1.2 Ideológia pri navrhovaní MQTT komunikácie

MQTT protokol bol navrhnutý pre siete s malou šírkou pásma a vysokou latenciou, preto je jeho dizajn zameraný na jednoduchosť a minimalizáciu potrebných dát. Preto boli zadefinované nasledujúce princípy, ktoré je odporúčané dodržiavať.

1. Jednoduchosť, je potrebné aby celá štruktúra bola jednoduchá a jednoducho implementovateľná, aby sa dala ľahko zakomponovať do iných riešení.

2. *Publish/Suscribe messaging.* Užitočné pre senzorové aplikácie, umožňuje zariadeniam prísť online a publikovať dáta bez toho aby boli predom definované v štruktúre.

3. Nulová administrácia. Treba sa správať rozumne pri odpovediach na nečakané udalosti a treba zabezpečiť aby stačilo, že aplikácie fungujú. Na príklad dynamicky vytvárať *topick*-y keď je treba.

4. Minimalizovať svoju prezenciu na sieti. Treba posielať len tie dáta, ktoré sú potrebné.

5. Treba očakávať časté prerušenia alebo nestabilitu spojenia a ošetriť to. Na príklad použitím závete.

6. Je potrebné zadefinovať čo je a čo nieje permanentné v štruktúre siete a podľa toho nastaviť daných klientov.

7. Treba očakávať, že klientske aplikácie majú k dispozícií veľmi málo zdrojov na spracovanie údajov.

8. Tam kde je to možné, je dobré zadefinovať tradičné *Quality of service*.

9. *Data agnostic*. netreba presne definovať formát dát, dá sa ostať flexibilným.

2 Technické vlastnosti MQTT

2.1 Topics/Subscritions

Všetky správy sú publikované v takzvaných témach, alebo *topic*-och. Tieto *topic*-y nepotrebujeme nakonfigurovať alebo inak dopredu definovať, publikovanie správy je dostatočné.
Topick-y majú podobnú štruktúru ako *filesystem*-y, kde sú jednotlivé dáta kategorizované a delené pomocou znamienka /.
Na príklad ak máme viacero počítačov, ktoré pôsobia ako *publisher*-i a posielajú správy o aktuálnej teplote ich diskov, tak hierarchia tohoto *topic*-u vyzerá následovne.

sensor/COMPUTER_NAME/temperature/HARDDRIVE_NAME

Odberatelia si vedia vytvoriť *subscrition*, ktorá môže byť na konkrétny *topic*, v takom prípade budú dostávať len správy s danými dátami. Taktiež je možné vytvoriť *subscription* obsahujúce viacero informácií a to pomocou špeciálnych znakov. Špeciálne znaky sú dva.

- \+ - je prvý znak, využíva sa na získanie viacerých informácií z rovnakej úrovne. Napríklad ak chceme z predošlého príkladu získať teploty všetkých diskov v nami zvolenom počítači, tak hierarchia odberateľa bude vizerať následovne.

  sensor/COMPUTER_NAME/temperature/+

  Ak chceme získavať túto informáciu zo všetkých počítačov, ktoré publikujú do tohoto *topic*-u tak použijeme takúto hierarchiu odberateľa.

  sensor/+/temperature/+

  Je potrebné definovať celý *topic*. Týmto sa myslí, že v našom príklade reťazce ako *sensor/+/temperature* alebo *sensor/+* nefungujú.

- # - je druhý znak, ktorý sa používa na označenie všetkých zostávajúcich levelov hierarchie a tým pádom musí vždy byť použitý na konci reťazca. Čiže ak chceme z daného počítača v našom príklade všetky údaje ktoré poskytuje, nielen teplotu disku, tak reťazec bude vyzerať následovne.

  sensor/COMPUTER_NAME/#

  V takom prípade je jedno aké ďalšie informácie poskytuje alebo koľko úrovní daná informácia má, pomocou tohoto reťazca si ich vyžiadame všetky.

2.2 Quality of sevice

V MQTT si vieme vybrať z troch úrovní *Quality of Service* (skrátene QoS), ktoré nám definujú ako veľmi sa *broker/client* snažia o to, aby bola správa správne prijatá. Úrovne QoS sú následovné.

- 0: *Broker/Client* doručí správu raz a bez potvrdenia o doručení.
- 1: *Broker/Client* doručí správu aspoň raz a je potrebné potvrdenie o doručení.
- 2: *Broker/Client* doručí správu práve raz s využitím štvor-krokovej inicializácie komunikácie (*Four step handshake*)

Správa má *publisher*-om danú maximálnu úroveň QoS, ktorú si vie odberateľ vyžiadať. Čo znamená, že ak je správa publikovaná v úrovni QoS1, tak si ju vedia odberatelia od *broker*-a vyžiadať v úrovni QoS0 a QoS1. Ak si odberateľ predsa len vyžiada túto správu vo vyššej úrovni, v našom príklade na príklad QoS2, tak predanie informácie prebehne, ale len v najvyššej publikovanej úrovni, v našom prípade QoS1.

2.3 Retained messages (Zachované správy)

Každá správa sa dá nastaviť tak, aby po odoslaní danej správy všetkým odberateľom ostala táto správa v pamäti *broker*-a a to pomocou *retain status*-u. Tým pádom ak príde nový klient, ktorý sa stane odberateľom *topic*-u, ktorý obsahuje túto správu, tak po vyžiadaní tohoto *topic*-u bude poskytnutá aj táto uložená správa.

2.4 Clean session/Durable connection

Keď sa pripojí klient nastaví si údaj o *clean session*.

Ak je tento údaj nastavený na *false*, tak je toto pripojenie brané ako stále (*durable*). To znamená, že ak sa tento klient odpojí, tak informácie o jeho odberoch a všetky správy s QoS1 a QoS2 budú automaticky uložené v pamäti *broker*-a pokým sa tento klient nepripojí späť a neprevezme tieto správy.

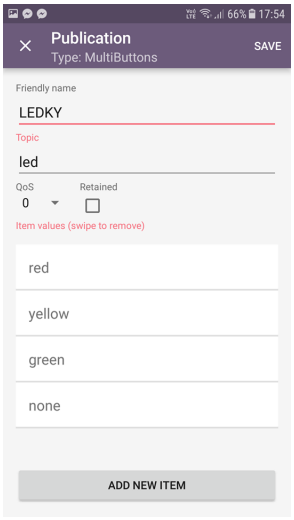
Ak je tento údaj nastavený na *true*, tak sú všetky údaje o tomto klientovi po jeho odpojení odstránené.

2.5 Wills (Závete)

Keď sa klient pripojí k *broker*-ovi, vie ho informovať, že má závet (*will* alebo *last will*). *Will* je správa, ktorá sa publikuje v prípade, že sa tento klient nečakane odpojí. Táto správa má takú istú štruktúru ako ostatné spravy a taktiež ako ostatné správy má *topic*, *Qos* a *retain status*.

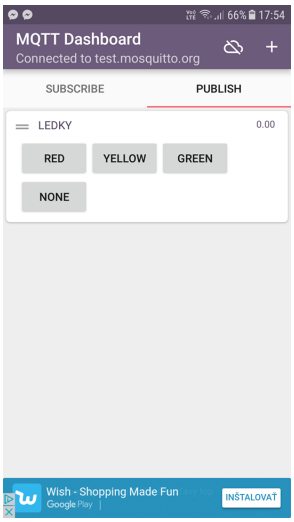
3 Praktický príklad 1: Rozsvecovanie LED-ky

V tomto príklade využívame aplikáciu a free hosting *broker*-a zo stránky *https://test.mosquitto.org/*. Aplikácia je volne dostupná na Google Store pod názvom *IoT MQTT Dashboard*. Ďalej využívame *Raspberry Pi* a LED-ky v troch rôznych farbách (červenú, žltú a zelenú).
Vytvorili sme *MQTT* komunikáciu, ktorá má dvoch klientov a *broker*-a. Prvým klientom je aplikácia v ktorej sme si vytvorili publikovanie *topic*-u, ktorý sme nazvali *led*. Nastavili sme, že tento *topic* môže mať hodnoty *red*, *yellow*, *green* alebo *none*. Tvorbu možno vidno na obrázku 4.



Obr. 4: Vytvorenie publikácie *topic*-u v aplikácií *IoT MQTT Dashboard*

Ako vidno, *QoS* sme nastavili na 0, a zachovanie správy (*retained messages*) na false, čiže správa sa po odoslaní odberateľom nezachová. Vytvoril sa nám teda klient, ktorému vieme nastaviť čo publikuje. Interface na používanie tohoto klienta vidno na obrázku 5.



Obr. 5: Publikovanie správy v aplikácií IoT MQTT Dashboard

Vieme teda publikovať nami žiadanú správu jednoduchým stlačením tlačidla zodpovedajúcemu našej požadovanej hodnote.

Ako už bolo spomenuté, náš *broker* beží na serveri poskytovaným na stránke *https://test.mosquitto.org/* a táto aplikácia sa k nemu automaticky pripája.

Ako klienta, ktorý pôsobí ako odberateľ využívame *Raspberry Pi*. Preňho sme vytvorili skript v jazyku *Python*.
V tomto prípade nám stačia dve funkcie a to na pripojenie na server a na prijatie správy. Toto je funkcia na pripojenie klienta k *broker*-ovi:

```
# The callback for when the client receives a CONNACK
# response from the server.
def on_connect(client, userdata, rc,self):
        print("Connected with result code "+str(rc))
        # Subscribing in on_connect() means that if we lose the
        # connection andreconnect then subscriptions will be
        # renewed.
        client.subscribe("led")
```

Ako vidno, definovali sme si funkciu *on_connect*, ktorá pri spustení odberateľa aby počúval na *topic led*.

Ďalšia funkcia nám definuje ako sa spracuje už prijatá správa.

```
# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
        print(msg.topic+" "+str(msg.payload))
        if msg.payload == "yellow":
                led.on()
        if msg.payload == "red":
        led1.on()
        if msg.payload == "green":
                led2.on()
        if msg.payload == "none":
            led.off()
            led1.off()
            led2.off()
```

V tejto funkcií na základe prijatej správy zapíname alebo vypíname LED-ky, ktoré sme si dopredu definovali a to nasledovne.

```
import paho.mqtt.client as mqtt
from gpiozero import LED

led = LED(4)
led1 = LED(5)
led2 = LED(6)
```

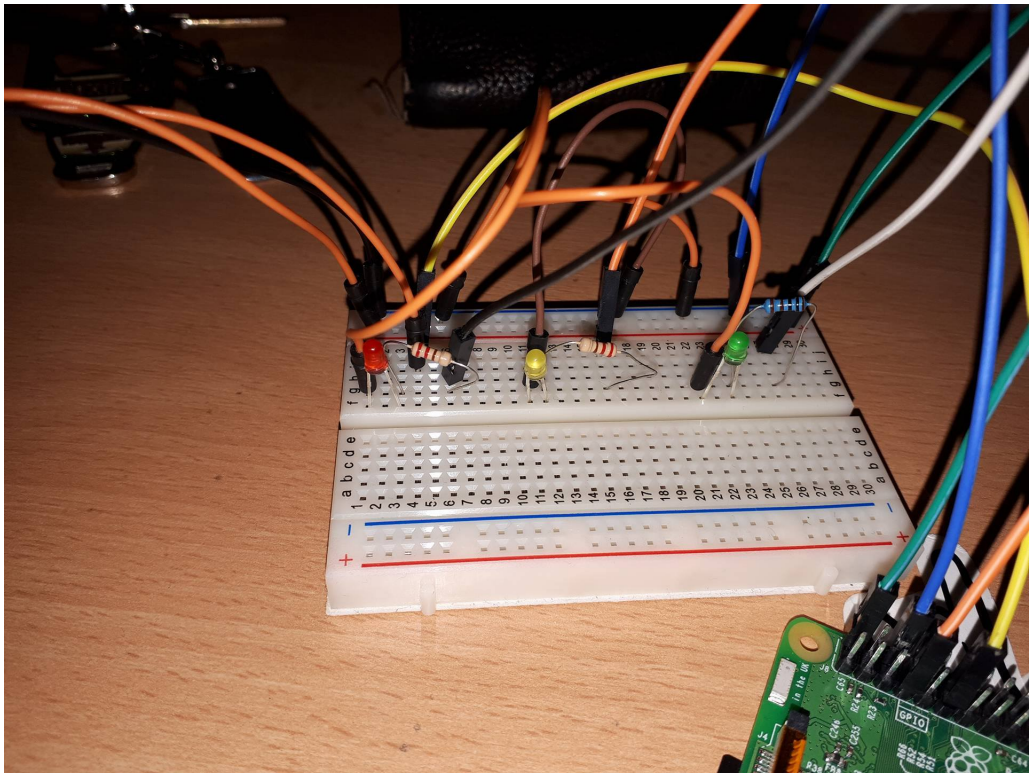
Tu vidno, že sme si importovali *paho.mqtt.client* ako *mqtt* a zadefinovali *GPIO* pre LED-ky.
Teraz nám len zostáva to celé spustiť.

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("test.mosquitto.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

Najprv sme si zadefinovali všetky potrebné parametre ako *client* a potom sme sa cez *client.connect()* pripojili k *broker*-ovi, ktorý beží na serveri, tak ako sme to už vyššie spomínali. Nakoniec sme to ešte celé zaciklili, aby sme vedeli prijímať akúkoľvek zmenu pokým tento skript beží.
Zapojenie LED-iek vyzeralo následovne.



Obr. 6: Zapojenie Raspberry Pi a LED-iek

4 Praktický príklad 2: Riadenie servo-motora

Druhým príkladom ktorý sme si pripravili je riadenie polohy servo-motora. V tomto prípade sme nevyužívali server ale všetko bežalo lokálne.
Ako *publisher*-a sme používali len jednoduchý príkaz v príkazovom riadku a to nasledovne.

mosquitto_pub -h localhost -t servo -m servo

Tento príkaz nám odoslal správu obsahujúcu slovo *servo* na *topic servo*. Tým pádom *broker* bežiaci na localhoste rozpoznal nový *topic* a správu, lebo ako bolo spomenuté, jediná potrebná inicializácia *topic*-u je poslanie správy do daného *topic*-u.

Aj tentokrát využívame *Raspberry Pi* ako odberateľa a beží na ňom skript pozostávajúci z troch funkcií.
Prvou funkciu je už v prvom príklade spomenutý *on_connect*, ktorý nám vytvorí odoberanie *topic*-u *servo*.

```
# The callback for when the client receives a CONNACK
# response from the server.
def on_connect(client, userdata, rc, self):
        print("Connected with result code "+str(rc))
        # Subscribing in on_connect() means that if we lose
        # the connection and reconnect then subscriptions
        # will be renewed.
        client.subscribe("servo")
```

Druhou funkciu je spravovanie prijatej správy, v tomto prípade ak príde správa obsahujúca slovo *servo* tak sa spustí funkcia *servo*.

```
# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
        print(msg.topic+" "+str(msg.payload))
        if msg.payload == "servo":
            servo()
```

Tretia funkcia, ktorú sme si definovali je teda funkcia *servo()* Tá povolí užívateľovi desať krát zadať požadovaný uhol, do ktorého sa má servo-motor vychýliť.

```
def servo():
    for i in range(1,10):
        a = input("Write an angle: (0-180)")
        angle = a/18 + 3
        pwm.start(angle))
```

Teraz je potrebné definovať si výstupy na ovládanie serva.

```
import paho.mqtt.client as mqtt
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(11,GPIO.OUT)
pwm = GPIO.PWM(11,50)
pwm.start(3)
```

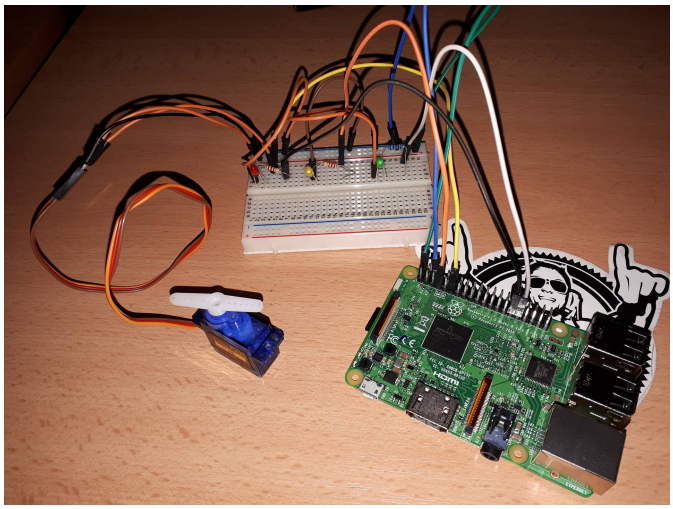
A na koniec to celé spustíme a taktiež uvedieme do cyklického chodu.

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

Zapojenie taktiež možno vidno na obrázku 7.



Obr. 7: Zapojenie Raspberry Pi a serva

5 Zdroje

https://github.com/mqtt/mqtt.github.io/wiki
http://mosquitto.org/man/mqtt-7.html
https://test.mosquitto.org/
https://play.google.com/store/apps/details?id=com.thn.iotmqttdashboard

Zdrojové kódy

server-led.py

```
import paho.mqtt.client as mqtt
from gpiozero import LED

led = LED(4)
led1 = LED(5)
led2 = LED(6)

# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, rc,self):
        print("Connected with result code "+str(rc))
        # Subscribing in on_connect() means that if we lose the connection and
        # reconnect then subscriptions will be renewed.
        client.subscribe("led")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
        print(msg.topic+" "+str(msg.payload))
        if msg.payload == "yellow":
                led.on()
        if msg.payload == "red":
        led1.on()
        if msg.payload == "green":
                led2.on()
        if msg.payload == "none":
            led.off()
            led1.off()
            led2.off()


client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("test.mosquitto.org", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```

servo.py

```
import paho.mqtt.client as mqtt
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BOARD)
GPIO.setup(11,GPIO.OUT)
pwm = GPIO.PWM(11,50)
pwm.start(3)

def servo():
    for i in range(1,10):
        a = input("Write an angle: (0-180)")
        angle = a/18 + 3
        pwm.start(angle)



# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, rc,self):
        print("Connected with result code "+str(rc))
        # Subscribing in on_connect() means that if we lose the connection and
        # reconnect then subscriptions will be renewed.
        client.subscribe("servo")

# The callback for when a PUBLISH message is received from the server.
def on_message(client, userdata, msg):
        print(msg.topic+" "+str(msg.payload))
        if msg.payload == "servo":
            servo()



client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect("localhost", 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks and
# handles reconnecting.
# Other loop*() functions are available that give a threaded interface and a
# manual interface.
client.loop_forever()
```