PlanteHub ved brug af MQTT-broker

Lavet af: Simon Skødt Holm Nielsen
Vejleder: Martin Kierkegaard
Afleveringsdato: 29. Maj 2020
Antal tegn: 24015 med mellemrum excl. indholdsfortegnelse, forside og litteraturliste

Indholdsfortegnelse

Indledning

Introduktion + Motivation

Denne synopsis er den grundlæggende del af min mundtlige 4. semester eksamen på datamatiker-uddannelsen i valgfaget IoT. Jeg har valgt at arbejde med og undersøge transport af data inden for Internet of Things. Med specielt fokus på Home Automation. Hvor Home Automation forstås som automatisering af hjemmet via IoT enheder som sensorer, lamper etc. Til dette har jeg valgt at lave et mindre system der skal påvise hvordan man kan lave en simpel ”Plante Central”, som skal være en samling af sensorere der holder øje på helbredet af ens planter i hjemmet. Samt hvilke fordele og ulemper der kan stødes på undervejs.

Min motivation for at arbejde med dette, er at jeg i forvejen er meget interesseret i Home Automation delen af IoT. Jeg har selv dannet en del erfaring med Raspberry Pi + Home Assistant og brugen af lignende teknologier til at lave simple Home Automation systemer.
Derudover har der undervejs på uddannelsen været stor fokus på FN’s 17 verdensmål og dér syntes jeg det kunne være en idé at indføre noget fokus på genopretning af biodiversitet i verdenen og muligvis kunne IoT hjælpe på dette område.

Problemformulering

Den grundlæggende problemstilling vil jeg beskrive via et hovedspørgsmål (markeret med **fed** skrift) samt flere tilhørende underspørgsmål (markeret med *italics*)

Hvordan bliver MQTT brugt i Home Automation?
Hovedsageligt vil jeg gerne kigge på hvordan data bliver håndteret i et lokalt hjemmenetværk med brugen af MQTT protokollerne.

Hvordan holder en MQTT-broker styr på sender og modtager

- At kigge generelt på hvordan en MQTT-broker er sat op og hvordan og hvorledes man sender og modtager til de korrekte destinationer.

Hvad er fordelene og ulemperne ved at bruge en MQTT-broker

- Da jeg har lagt mærke til at i Home Automation netværker er det ofte man ser brugen af en MQTT-broker så kunne jeg godt tænke mig at vide hvorfor de er fordelagtige at bruge i forhold til andre protokoller.

Kan man lave sin egen MQTT-broker og forbinde den til nogle sensorer?

- Ved at kunne forbinde brokeren til nogle sensorer ville jeg kunne teste praktikaliteterne af den samt komme et skridt videre i at lave et produkt der omhandler det at kunne observerer planternes helbred i ens eget hjem via en tablet, pc, mobil etc.

Metode

De metoder jeg har valgt at gøre brug af, har primært afspejlet sig af egen erfaring samt researcharbejde. Research har været via generel dokumentation på internettet.

Eksperimenter i dette forløb vil have til formål at teste teknologien og ville kunne bekræftes/afkræftes via nogle tests af MQTT-brokeren, primært ved brug af brokeren med en række forskellige sensorere.

Planlægning

Som del af forløbet har jeg skulle planlægge hvordan mit arbejdsforløb har skulle se ud.
Jeg har valgt ikke at arbejde med nogle systemudviklingsmodeller som SCRUM eller XP, men derudover fokuseret på at overordnet få stablet en uge til uge basis af aktiviteter. Dette skyldes at jeg forudsér forløbet som en anelse ustabilt og at meget kan komme til at ændre sig under forløbet.

Under alle uger vil jeg løbende skrive synopsen hvilket er derfor den ikke er inkluderet i tabellen forneden.

Uge	**Aktivitet**
18	Projektafgrænsning, Problemformulering, Planlægning, Research
19	Problemformulering, Research, Arkitektur-overvejelser, MQTT-broker
20	Anskaffe sensorer, MQTT-broker, Test af MQTT-broker
21	Samle sensorer, Research plantedatabase, Lokal hjemmeside
22	Lokal hjemmeside, Test af system

Research

Indledning

I det følgende afsnit vil jeg gennemgå den research der har været nødvendig i mit tilfælde. Jeg vil derudover også undersøge hvilke typer af datatransport der kan være en mulighed for min prototype. Jeg vil yderligere specificere hvilke dele af MQTT-brokeren jeg har tænkt mig at fokusere på, da dette kan have en direkte indflydelse på hvor lang tid processen kommer til at foregå afhængig af hvor mange funktioner af MQTT jeg vil påvise.

MQTT

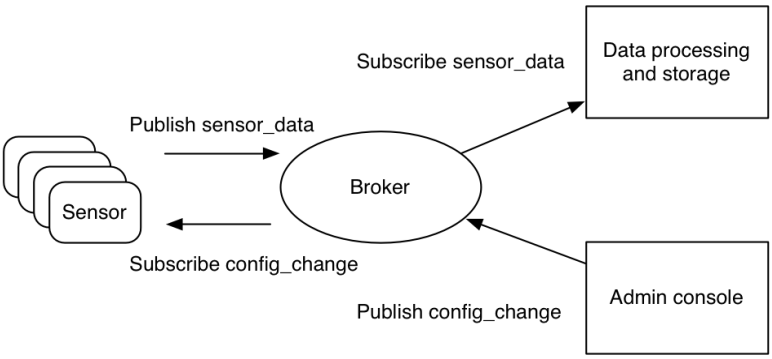
Overview

Overordnet set består MQTT-protokollen af to typer forbundne enheder, en broker (server) og en række klienter. MQTT-brokerens opgave er at modtage en publiceret besked fra en klient og sende denne besked ud til alle klienter der har abonneret på det "topic" som beskeden blev publiceret på. Denne type kommunikationsmodel kan også kaldes en publish/subscribe model[1].

Man kan sammenligne modellen med et postkontor. Hvor postkontoret er brokeren der sørger for at blade/reklamer kommer ud til modtagere der har abonneret på bladene/reklamerne.

Broker/Klient Egenskaber

Som nævnt tidligere er brokeren en server der håndterer beskeder ind og ud af systemet. Dette gøres ikke ved brug af klienternes ip-adresser men i stedet ved brug af det såkaldte "topic".
Et topic består primært af en tekst-streng hvorpå der opdeles lidt ligesom en mappestruktur på et standard URI. Dette kan se således ud:
"hovedemne/underemne/yderligere-underemne/osv.."
Derudover skal URI'et også følge nogle bestemte retningslinjer som bestemt indenfor MQTT-protokollen. Dette er fordi protokollen også kan indeholde specielle karaktere kaldet "wildcards". Disse karakterer kan indføres hvis man gerne vil abonnere på en hel række abonnementer indenfor et underemne, f.eks: "hovedemne/#" hvor der



Billede 1: MQTT Sensor Model

indikeres at man gerne vil abonnere på alle topics i mappen hovedemne.
Hvad der publiceres er op til klienten der publicerer på det emne, men i mange tilfælde består det af JSON, XML, etc.

En klient kan både publicere og abonnere på topics. Tag f.eks. et system hvorpå en broker er forbundet til en række sensorer og en dataopsamler/behandler. Dette kan ses på billedet 1[3].
Her vil sensorerne både publicere deres egne målinger men udover dette også abonnere på eventuelle konfigurationsændringer fra administratoren.
Dette gør det muligt at nemt kunne ændre sensorernes måleegenskaber således de passer til det formål de nu skal bruges til.
Man kan kigge lidt på det som et flow af information sendt ud til de nødvendige modtagere som så kan videregive deres egne informationer til andre modtagere.

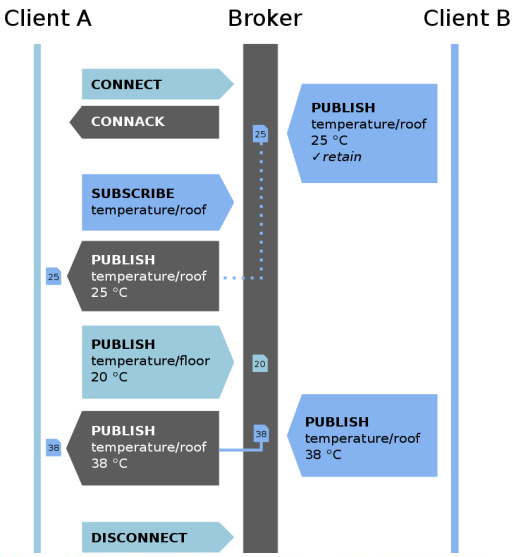
En ting der også er vigtig at pointere er at protokollen ses som værende letvægts, dvs. at der ikke bliver transporteret store mængder af data i mellem klienter og broker. Dette gør det nemt at opskalere systemer med flere enheder[2].

Transport

Måden dataen bliver transporteret i mellem broker og klienter er via TCP/IP protokoller eller andre typer bidirektionale transportmetoder.
Derudover indeholder protokollen en række datapakker (packets) der skal sikre forbindelsen mellem broker og klient og sørge for at dataen bliver sendt og modtaget succesfuldt.
Nogle af datapakkerne er som følgende:

- CONNECT: Sendes når klient anmoder om tilladelse til at forbinde til brokeren.
- CONNACK: Broker skal sende dette tilbage om den har godkendt klienten.
- SUBSCRIBE: Sendes til broker når en klient vil abonnere på et bestemt emne.
- UNSUBSCRIBE: Sendes til broker hvis en klient vil fjerne et abonnement.fjernelse.
- PUBLISH: Sendes fra klient til broker og derefter fra broker til modtager klient
- DISCONNECT: Sendes til broker hvis klienten vil fjerne forbindelse til brokeren.

En mere visuel måde at vise dette på kunne være ligesom på billede 2[10].
Her kan det ses at en klient A forbinder til en broker ved at sende en CONNECT pakke til brokeren. Brokeren skal så svare tilbage med en CONNACK pakke og fortælle klient A at den succesfuldt er blevet forbundet til brokeren.
Når dette så er sket kan klient A få lov til at abonnere på et emne (via SUBSCRIBE pakken), i dette tilfælde "temperature/roof", og brokeren skal så tilføje klienten som abonnent til emnet.



Billede 2: Eksempel på procedure af sendte pakker

Her kan det være at klient B har publiceret (via PUBLISH pakken) noget data på emnet tidligere end klient A har abonneret, men klient B valgte at bede brokeren holde beskeden liggende ved hjælp af flaget "retain". Dermed kan brokeren så publicere beskeden til klient A når klient A abonnerer på emnet på trods af at klient B ikke har publiceret noget efter abonnement fandt sted.

Derudover viser billedet også at klient A publicerer sin egen besked til emnet "temperature/floor". Denne besked bliver dog ikke sendt til klient B, da klient B ikke har valgt at abonnere på det emne og da klient A ikke bad om at lade brokeren gemme beskeden, bliver den i stedet bare slettet/glemt

Til sidst publicerer klient B en ny besked og klient A modtager den fra brokeren stort set lige efter hvorefter klient beslutter sig for at sende en DISCONNECT pakke og afslutte forbindelsen.

Prototype

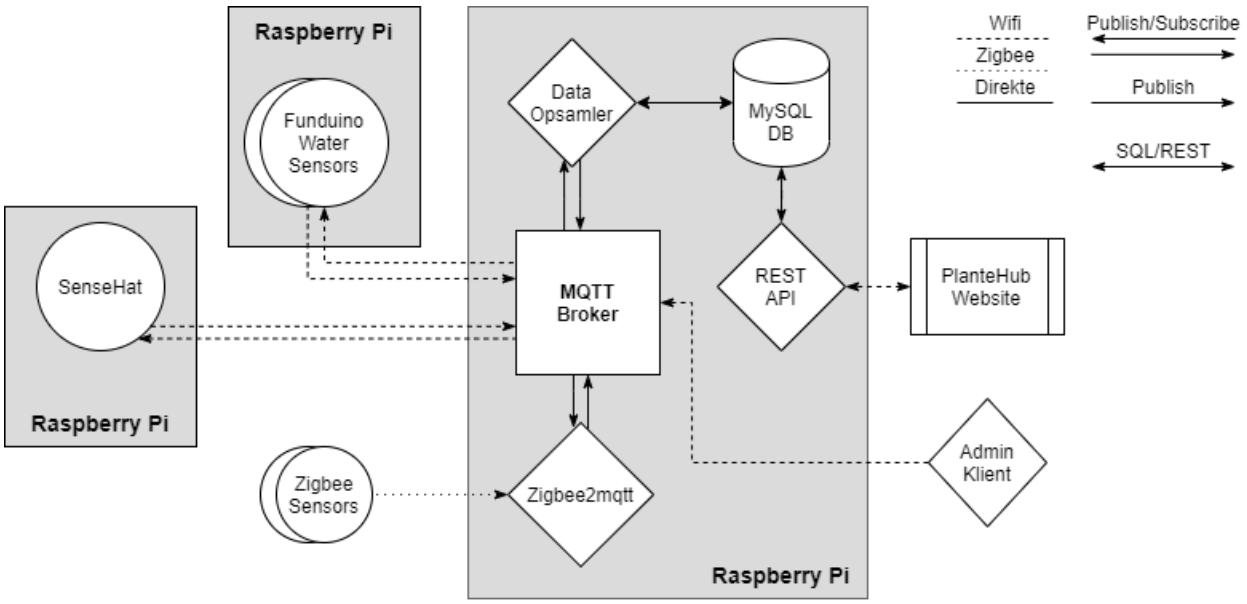
For at afprøve mulighederne med MQTT har jeg lavet en prototype af et system der kan afprøve nogle af de mere basale funktioner af MQTT protokollen. Med dette mener jeg publish, subscribe, unsubscribe, connection og disconnection. Derudover vil jeg også gennemgå hvordan jeg har introduceret nogle sensorer til systemet samt hvilke overvejelser der har været undervejs i udviklingsprocessen.

Arkitektur

Arkitekturen af systemet er bygge op med MQTT-brokeren som et centralt punkt. Det er den jeg har valgt at lægge vægt på og man kan dermed sige at nogle af de inkluderede enheder og programmer er noget jeg har valgt at tilbygge for at påvise hvordan man f.eks. kunne bygge et system op der kan bruges i praktiske sammenhænge.
Det har som beskrevet tidligere i synopsen skulle være en løsning til at overvåge op til mange forskellige planters helbred, for at give brugeren et nemmere overblik over sine planter.

Kigger man på billede 3, kan det ses at på en Raspberry Pi, er der inkluderet både MQTT-broker, MySQL database og en REST API. Derudover er der også to MQTT-klienter; en Dataopsamler hvis opgave er at abonnere på alle sensorer der publicerer til brokeren og via SQL-kald lægge publikationerne ind i databasen. Dermed er der også inkluderet en Zigbee2mqtt-klient, som skal oversætte fra et Zigbee signal til MQTT pakker, inklusiv nogle andre funktioner som jeg vil uddybe lidt senere.

Billede 3: Domænemodel



Alle sensorerne er forbundet via en trådløs forbindelse, enten via WiFi eller via Zigbee. I det nuværende system findes der 3 typer sensorer; Zigbee, SenseHat og en Funduino vandsensor forbundet til en Raspberry Pi.

Derudover er der også forbundet en hjemmeside til REST API'en (bilag 4) og en administrator klient. Hjemmesiden havde jeg egentlig den intention skulle være host'et på Pi'en, men er på nuværende tidspunkt bare en visuel bonus til at vise det funktionelle system (bilag 5).
Administrator klienten skal fungere som en ekstern klient der kun kan publicere konfigurationsændringer. Således kan man styre intervallerne af sensorernes målinger ligesom på billede 1.

MQTT-broker

Min broker er bygget op med inspiration og enkelte klasser fra Beerfactory's HBMQTT bibliotek[4]. Dette bibliotek er bygget oven på "Asyncio" som er et asynkront modul til python. Min broker er delt op i to forskellige klasser; selve brokeren og en session-klasse (session skal forstås som, når en klient forbinder sig til brokeren).

Broker-klassen består generelt af to asynkrone loop i dens start-metode. Et loop der lytter efter nye klienter og håndterer de klienter når de forbinder til brokeren, samt et loop der venter på publicerede beskeder fra de forbundne klienter.

Håndtering af Klienter

Som beskrevet tidligere skal brokeren opfylde nogle regler ifølge MQTT-protokollen. Dette betyder at brokeren skal håndtere modtagne pakker på en bestemt måde. Her gøres der brug af "start_server" metoden i asyncio modulet. Denne metode inkluderer både en StreamReader og en StreamWriter. Disse gør jeg brug af for at lytte efter input fra klienten. Når en klient forbinder sig til brokeren vil den starte metoden "handle_client". Denne metode er en såkaldt coroutine hvilket gør at den kan køres asynkront og er nødvendig for at flere end en klient kan forbinde sig til brokeren.

```
connect = yield from MQTTConnect.ConnectPacket.from_stream(reader)
if (connect.payload.client_id is None):
    generate_id = True
else:
    generate_id = False

#send a connack packet to client depending on correct credentials
connack = None
if (connect.username == self._username and connect.password == self._password):
    connack = MQTTConnack.ConnackPacket.build(0, MQTTConnack.CONNECTION_ACCEPTED)
    connected = True
else:
    connack = MQTTConnack.ConnackPacket.build(0, MQTTConnack.BAD_USERNAME_PASSWORD)
    connected = False

writer.write(connack.to_bytes())
```

For at opfylde flowet som kravet fra MQTT-protokollen, venter brokeren først efter input fra klienten. En asyncio coroutine kræver så at man bruger "yield from" for at vente på inputtet, i stedet for await som andre typer af asynkrone procedurer gør brug af.
Når en klient har sendt en CONNECT pakke kontrollerer brokeren også at klienten har lov til at få adgang til brokeren via et brugernavn og kodeord. Afhængig af hvad resultatet af denne kontrol er, vil brokeren uanset hvad sende en CONNACK pakke tilbage og samtidig i den pakke fortælle klienten om den blev godkendt.

Der bliver så lavet et objekt af klassen Session til at holde relevant information fra klienten med specielt fokus på klientens id, klientens abonnementer, og derudover reader og writer.
Dette objekt bliver så tilføjet til en liste af alle igangværende sessioner, hvor der derefter startes et while loop der konstant venter på input fra klienten og sammenligner inputtet med de forskellige typer af pakker, som nævnt tidligere i Research afsnittet, dog kun så længe klienten stadig er forbundet.

Subscribe:

```
def add_subscription(self, subscription, session):
    if (subscription not in self._subscriptions):
        self._subscriptions[subscription] = []

    match = []
    for s in self._subscriptions[subscription]:
        if (s.client_id == session.client_id):
            match.append(s)
            s.reader = session.reader
            s.writer = session.writer


    if (len(match) == 0):
        self._subscriptions[subscription].append(session)
        print("Client %s subscribed to %s" % (session.client_id, subscription))
    else:
        print("Client %s has already subscribed to %s" % (session.client_id, subscription))
```

Når en klient beder om at abonnere på et nyt emne skal brokeren håndtere det ved at tjekke først om abonnementet allerede eksisterer. Hvis ikke, så laver den det nye abonnement i brokerens liste over abonnementer.
Derefter skal brokeren kontrollere om klienten ikke allerede har abonneret på emnet. Hvis det forekommer at klienten allerede har abonneret på emnet skal det tilføjes match-listen samt reader og writer overføres til klientens nye session.
Hvis klienten så ikke allerede har abonneret ved at tjekke om længden af match-listen er større end 0, så tilføjes sessionen til listen af abonnementer på lige netop det ønskede emne.
Det kan så forstås som at brokeren har en liste over alle emner som klienter har abonneret til. Her har hver enkelt emne så en tilsvarende liste over sessioner der abonnerer på emnet.

Unsubscribe:

Hvis en klient ikke længere vil abonnere på et specifikt emne, skal brokeren så fjerne klienten fra listen over abonnenter på det emne.

```
def del_subscription(self, topic, session):
    deleted = False
    subscriptions = self._subscriptions[topic]

    for index, sub_session in enumerate(subscriptions):
        if (sub_session.client_id == session.client_id):
            subscriptions.pop(index)
            deleted = True
            break

    return deleted
```

Dette gøres ved få listen over alle abonnenter og derefter lede efter klientens id i listen. Derudover skal der bruges indekset af id'et for at kunne fjerne det, så derfor bruges "enumerate()".
Hvis klientens id eksisterer i listen skal den fjernes via ".pop" metoden og der indikeres så at det lykkedes at slette abonnementet ved at sætte "deleted = True" og returnere denne værdi. Dette er især brugbart fordi metoden gøres også brug af når klienten sender en DISCONNECT pakke og der vil man gerne holde styr på at det lykkedes at fjerne alle abonnementer klienten havde således at brokeren ikke kommer til at prøve at publicere en pakke til en abonnent der ikke længere er forbundet.

Publish:

Den simpleste af metoderne er publicering af beskeder under specifikke emner.
Når en klient vil have publiceret sin besked til de brokeren så bliver den relevante information lavet om til en dictionary og derefter bliver det tilføjet til brokerens broadcast kø.

```
@asyncio.coroutine
def broadcast_message(self, session, topic, payload):
    broadcast = {
        'session': session,
        'topic': topic,
        'data': payload
    }

    yield from self._broadcast_queue.put(broadcast)
```

Disconnect:

```
def del_session(self, client_id):
    session = self._sessions[client_id]

    self.del_all_subscriptions(session)

    del self._sessions[client_id]

def del_all_subscriptions(self, session):
    topic_queue = deque()

    for topic in self._subscriptions:
        if self.del_subscription(topic, session):
            topic_queue.append(topic)

    for topic in topic_queue:
        if not self._subscriptions[topic]:
            del self._subscriptions[topic]
```

Når en klient ikke længere vil være forbundet til brokeren skal serveren håndtere det ved at først slette alle abonnementer som klienten har abonneret på. Dette gøres ved at gå igennem alle emnerne i brokerens liste over emner. Derved gør vi brug af den tidligere funktion og kontrollerer hvorvidt det lykkedes at slette abonnementet fra emnet. Hvis det lykkedes tilføjes emnet til en kø af slettede emner.
Derefter kan brokeren så gå igennem alle de slettede emner og kontrollere hvorvidt emnerne har andre abonnenter, hvis ikke så slettes emne helt fra listen, ellers bliver den beholdt.

Broadcast-loop:

Tidligere nævnte jeg at brokeren består af to asynkrone loops. Det andet loop er her broadcast-loop'et. Dette loop er nødvendigt i tilfælde af at flere klienter publicerer noget samtidigt. Dette har jeg forudset som værende meget sandsynligt at kunne ske. Derved har brokeren en tilknyttet asynkron kø kaldet "asyncio.Queue". Denne kø kan man så hente instanser ud fra via ".get()" metoden når en klient har sendt sin publicering til køen.

```
@asyncio.coroutine
def broadcast_loop(self):
    while True:
        broadcast = yield from self._broadcast_queue.get()

        for sub in self._subscriptions:
            if (broadcast['topic'].startswith('&') and (sub.startswith("+") or sub.startswith("#"))):
                print("[MQTT-4.7.2-1] - Ignoring broadcasting $ topic to subscriptions starting with + or #")
            elif (self.match(broadcast['topic'], sub)):
                subscriptions = self._subscriptions[sub]

                for match_sub in subscriptions:
                    task = asyncio.ensure_future(match_sub.publish(broadcast['topic'], broadcast['data']), loop=self._loop)
```

Når brokeren har hentet en publikation fra køen, skal der så kontrolleres om der er nogle der abonnerer på emnet i publikationen. Dermed går brokeren gennem hele listen over abonnementer. Hvis der i publikationens emne starter med karakteren "$" og samtidig indeholder abonnement-emnet en wildcard karakterer som "#" eller "+" skal der ikke broadcastes til dette abonnement. Dette er specificeret ifølge MQTT-protokollen da det ikke er meningen at klienter skal kunne tale med hinanden via emner indeholdende "$". Derimod kan en broker implementere en brug af "$" på anden vis[1].
Hvis brokeren ikke finder nogle problemer med specielle karakterer så kan der ledes efter abonnenter der skal have videregivet beskeden på det emne.

```python
def match(self, topic, match_topic):
    if ("#" not in match_topic and "+" not in match_topic):
        #if no wildcards are in matched topic, return if topics are equal (i.e match)
        match = topic == match_topic
        return match
    else:
        #regex to match patterns of strings preceding the topic and including wildcards
        pattern = match_topic.replace('#', '.*').replace('$', '\$').replace('+', '[/\$\s\w\d]')
        match = re.match(pattern, topic)
        return match
```

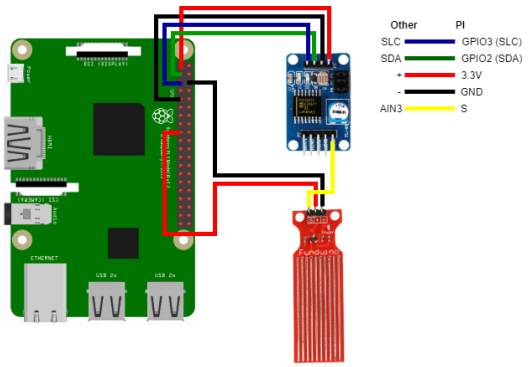
Metoden her hedder “match” og skal finde ud af om hvorvidt et givent emne passer til et andet emne i listen af abonnements-emner. Her skal man huske at wildcards også spiller ind. Men det mest simple eksempel ville være hvis der ingen wildcards befinder sig i abonnement-emnet. Så skal der bare kontrolleres om emnerne stemmer overens og hvis de gør returnér “True”, hvis ikke returnér “False”.
Det mere komplekse tilfælde ville være et abonnement-emne der indeholder wildcards. Her bruges der python modulet regex[12] til at finde et mønster i emne-strengen. Først skal strengen dog klargøres til regex “match()” metoden. “#” erstattes af “.*” for at få regex til at matche alle karakterer på det givne index og alle efterfølgende karakterer.
Karakteren “$” erstattes af “\$” for at fortælle regex at karakteren skal ignoreres da den har sin egen funktion i modulet.
Til sidst skiftes “+” ud med “[/\&\s\w\d]” for at indikere at regex skal matche strengene indtil en ny “/” hvor de resterende karakterer er indikatorer for at den matcher alle typer Unicode karakterer, både bogstaver og decimalværdier.
Regex returnerer så enten “True” eller “False” afhængig af om emnerne stemmer overens inklusiv de specielle karakterer. Dette kan broadcast-loop’et så bruge til vide om den skal sende publikationen ud til den korresponderende klient ved at publicere med inkluderingen af både emne og data.

Sensorer

Da MQTT-brokeren gør brug af simpelt JSON format kan jeg forbinde mange forskellige typer af enheder til brokeren. Det vil jeg prøve at påvise både ved at lave mine egne sensorer, men også ved at prøve at forbinde sensorer der gør brug af Zigbee protokollen.
Jeg har opdelt mine emner som sensorerne publicerer på således:

- “wifi/sensor/sensor_navn_her” :
  Til alle sensorer der bruger WiFi som transport.
- “zigbee2mqtt/sensor_navn_her” :
  Til sensorer der bruger Zigbee som transport

Mest grundet at jeg ikke har særlig mange sensorere. Man kunne evt. inkludere underemner der fortæller om sensorens placering i hjemmet: “wifi/sensor/stue/navn”



Billede 4: Schematic af hjemmelavet sensor

Raspberry Pi + Funduino Water Sensor

For at påvise hvordan man laver en MQTT-klient der kan publicere data fra en sensor har jeg valgt at lave en sensor der kan måle fugtigheden i plantejord. Det har jeg ved brug en Raspberry Pi, en AD/DA converter[7][8] og en Funduino Water Sensor. Da Raspberry Pi'en ikke har mulighed for at måle analoge signaler har det været nødvendigt at bruge en converter der kan konvertere det analoge signal til en digital repræsentation (i form af intervallet 0x00 - 0xff).
Det har jeg så kunne læse ved brug af python modulet SMBus2 og ved at slå I2C til på Raspberry Pi'en. Converteren har så adressen 0x48 som jeg har så også fortæller at den fremtidigt skal læse signalet på channel 3 på converter'en hvor sensoren er forbundet[9].

```
PCF8591 = smbus2.SMBus(1)
PCF8591.write_byte(address, 0x03)
```

Til at forbinde til MQTT-brokeren har jeg anvendt et python modul kaldet "Paho-mqtt"[5]. Dette er et klient baseret modul der gør det nemt at forbinde til MQTT-brokere ved at lave nogle simple start konfigurationer. Disse konfigurationer inkluderer brugernavn og kodeord samt hvad der skal ske når man er forbundet og når man modtager en besked fra emne man er abonneret til. Til min sensor har jeg valgt at abonnere på "admin/config_changes" for at nemt kunne ændre på tidsintervallerne mellem målinger fra en enkelt ekstern administrator enhed.

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.username_pw_set('mqtt', password='mqtt')

client.connect(ip, port, 60)
client.subscribe('admin/config_changes')
client.loop_start()
while_connected(client, interval, 0, 256)
```

Derudover skal også foretage målinger imens den er forbundet, her gøres der brug af metoden "PCF8591.read_byte()" og denne værdi omdannes til den tilsvarende procentvise fugtighed. Når sensoren så skal publicere signalet gøres det via ".publish()" i MQTT-klienten. Dette inkluderer emnet, samt data i JSON-format og der skal også specificeres en QoS (læs bilag 2 for forklaring af denne funktion) samt om beskeden skal gemmes lokalt hos brokeren.

```
client.publish(topic, str(message), qos=0, retain=False)
```

Zigbee Sensor

Når det gælder Zigbee kan jeg ikke lave en klient der kan modtage signaler direkte fra en zigbee sensor. Det kræver også lidt ekstra udstyr. For at kunne forbinde en Zigbee sensor til brokeren kræver det to yderligere ting[6].

- Zigbee2mqtt software:
  Klientbaseret program der omdanner zigbee signaler til MQTT brugbart data
- CC2531 USB Zigbee 'sniffer':
  USB antenne der opfanger Zigbee signaler.

Her fungerer Zigbee2mqtt som klient og kan publicere og abonnere på forskellige topics. Dermed har man også mulighed for at ændre på dens konfiguration så det tilpasses ens Zigbee netværk afhængig af hvordan det er opsat og hvilke enheder det gør brug af.
Klienten publicerer så på vegne af sensorerne når de udsender en måling til USB-antenne og videre til Zigbee2mqtt klienten.

Dette fungerer glimrende da man nemt kan forbinde nye sensorer til netværket bare ved at "parre" dem med antennen og klienten. Dermed hvis man har brug for nogle nye sensorer til flere planter kan man nemt forbinde dem til resten af netværket. Dette kunne være et glimrende redskab til at opskalere en Plante Central på.

Konklusion

Jeg kan hermed konkludere at grundlæggende bliver MQTT-protokollen brugt fordi den er fleksibel og letvægtig når det kommer til at holde styr på mange enheder. Dette gør det muligt at kombinere mange forskellige typer af systemer på samme broker. Det kan nemt opskaleres da det er pointen med at være en letvægts protokol.
Protokollens fleksibilitet skal forstås som værende dens evne til at kunne koble forskellige typer af systemer sammen. Dette kan være lige som i mit tilfælde WiFi og Zigbee, men også andre typer af protokoller da der findes klientbaseret biblioteker til stort set alle typer af enheder og protokoller. Dette kan være f.eks. BlueTooth, Z-Wave etc.
Derudover kan det også være fordelagtigt at bruge protokollen hvis der i systemet er inkluderet sensorer der er batteridrevne og dermed helst skulle spare så meget strøm som muligt.

Ulemperne ved at bruge protokollen er ikke så fremtrædende, det kræver dog lidt forståelse for hvorfor det er nødvendigt at bruge i visse situationer. Det skal ikke bruges som løsning til alle systemer, hvis et system kræver at to klienter skal have kendskab til hinanden via direkte forbindelse kan det ikke rigtig bruges, da protokollen specifikt undgår dette ved at have brokeren midtpunkt.

Det kan godt lade sig gøre at lave sin egen broker og forbinde den til nogle sensorer. Det har lykkedes mig at forbinde den til både WiFi og Zigbee sensorere. Her kan jeg dog konkludere at praktisk ville det være Zigbee der ville være den bedste løsning til brug i en Plante Central. Dette skyldes at det er nemmere at sætte op og holde ved lige end WiFi sensorerne.

Refleksion

Når det kommer til stykket så er min hjemmelavede broker ikke i nærheden af at indeholde alle de ting som en broker der følger MQTT-protokollen til punkt og prikke normalt har.
Jeg mangler f.eks. Inklusion af forskellige QoS, at kunne håndtere at brokeren skal kunne gemme publikationer hvis klienten vil have det, samt at inkludere et kø system til nye abonnementer da man i større systemer godt kunne forudse at enheder ville abonnere på en ny

enhed på nogenlunde samme tid hvilket med min nuværende iteration af brokeren ikke ville kunne håndtere.
En ting jeg også fandt frem til som gør MQTT-brokere utroligt nyttige er at bruge dem sammen med WebSockets[11]. Dette gør det muligt at forbinde en browser direkte med brokeren og dermed kan browseren modtage informationer og opdatere lige så snart den modtager ny information fra brokeren.
Processen har været meget lærerig, den har både givet mig en lidt mere hardware orienteret forståelse for visse typer af teknologier som Raspberry Pi og sensorere, men også en bedre forståelse for generelle server-klient baserede systemer.

Litteraturliste

1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
2. http://mqtt.org/faq
3. https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/
4. https://github.com/beerfactory/hbmqtt/tree/07c4c70f061003f208ad7e510b6e8fce4d7b3a6c
5. https://pypi.org/project/paho-mqtt/#id3
6. https://www.zigbee2mqtt.io/getting_started/running_zigbee2mqtt.html
7. https://www.waveshare.com/wiki/Raspberry_Pi_Tutorial_Series:_PCF8591_AD/DA
8. https://brainfyre.wordpress.com/2012/10/25/pcf8591-yl-40-ad-da-module-review/
9. http://www.diyblueprints.net/measuring-voltage-with-raspberry-pi/
10. https://en.wikipedia.org/wiki/MQTT#/media/File:MQTT_protocol_example_without_QoS.svg
11. https://www.hivemq.com/blog/mqtt-essentials-special-mqtt-over-websockets/
12. https://docs.python.org/3/library/re.html