Internet des Objets (IoT)

Chapitre 2 : Le protocole MQTT

Aimen Bouchhima

Plan

Introduction

"MQTT est un protocole client serveur de transport de message en mode "publish/subscribe" (publier/souscrire). Il est léger, ouvert, simple et conçu pour être facile à mettre en œuvre. Ces caractéristiques le rendent idéal pour une utilisation dans de nombreuses situations, y compris dans des environnements à ressources restreintes tels que la communication dans des contextes M2M (Machine to Machine) et l'Internet des objets (IoT) dans lesquels une taille de code réduite est requise et / ou la bande passante réseau n'est pas garantie. "

Introduction

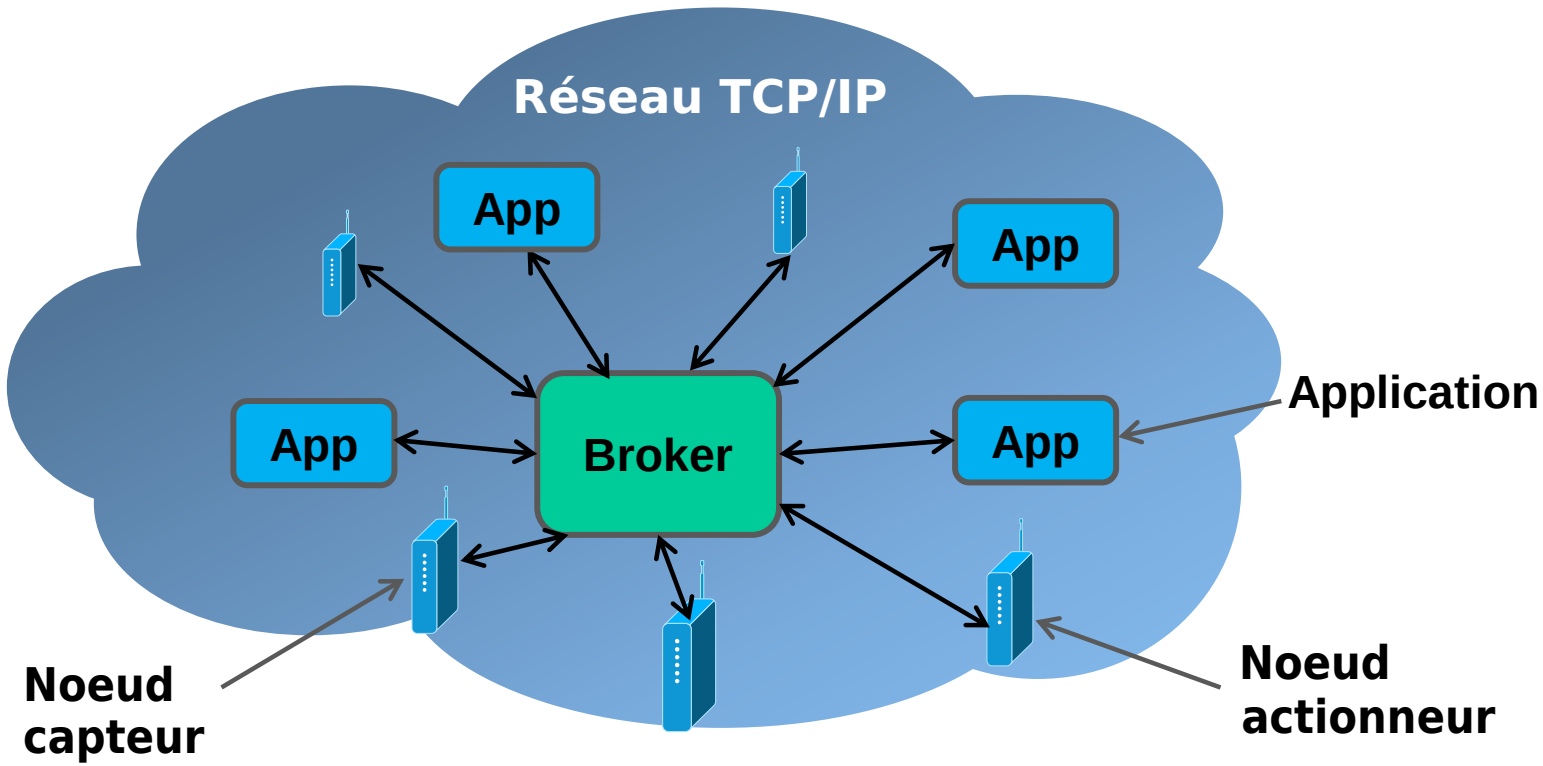
- initialement développé (1999) par Andy Stanford-Clark (IBM) et Arlen Nipper (EuroTech).
- se base sur le protocole TCP/IP
- Standart OASIS : versions 5 et 3.1

Origine de MQTT

- A l'origine, MQTT a été conçu pour interconnecter des noeuds capteurs (sensor nodes) dans des réseaux peu fiables et/ou à latences élevées.
- Exemple : Contrôle et suivi des caduc de pétrole et de gaz dans des endroits peu accessibles
  - Connectivité restreinte (communication par satellite à faible bande passate et fortes latences).
  - Les noeuds capteurs, opérant avec des batteries, devaient consommer très peu d'énergie et donc sont restreint en terme de puissance de calcul et de mémorisation
- Aujourd'hui l'utilisation de MQTT s'est élargie au delà de sa raison d'être. Par exemple Facebook utilise MQTT pour la partie messagerie de son application Facebook Messenger

MQTT : les concepts de base

Le protocole MQTT repose sur plusieurs concepts de base, tous destinés à assurer la livraison du message tout en gardant les messages eux-mêmes aussi légers que possible.

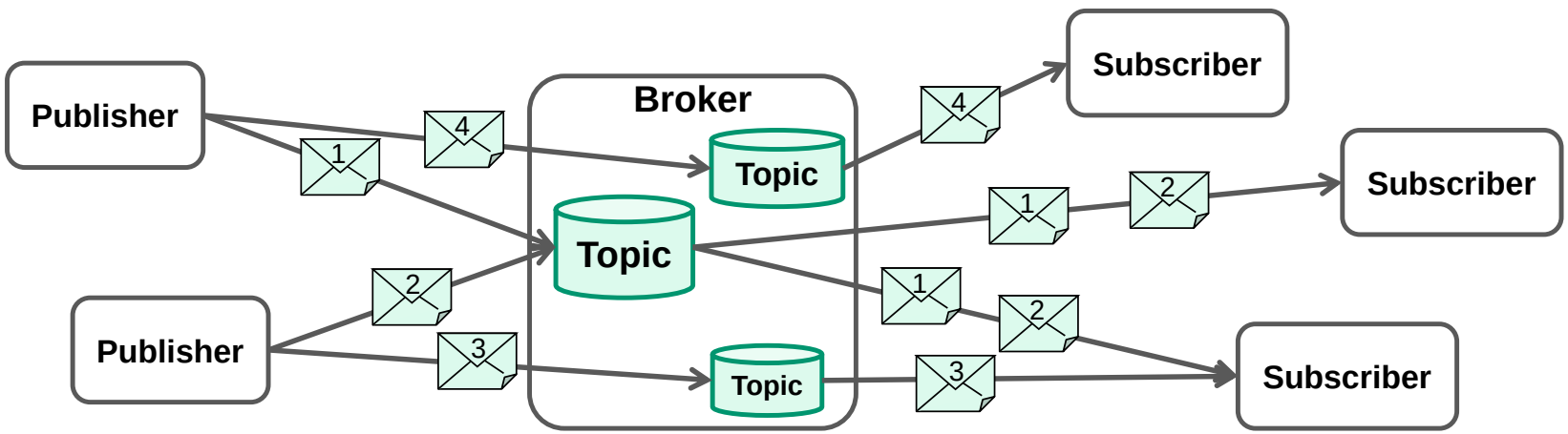


Le modèle Publish/Subscribe (publier/souscrire)

- Le modèle publish/subscribe (connu aussi sous le nom pub/sub) est un modèle de communication qui repose sur l'envoi de messages entre des entités faiblement couplées appelées clients
- les messages sont classés en `topics` (sujets ou thèmes)
- Pour recevoir un message postés dans à un `topic` donné, un client doit se souscrire (subscribe) à ce `topic`.
- A tout moment, un client peut publier (publish) un message dans un `topic` particulier, rendant ce message disponible pour tous les clients qui se sont souscrits à ce `topic`.

Le modèle Publish/Subscribe (publier/souscrire)

- Une entité centrale assure le filtrage et l'acheminement des messages entre les différents clients : le `broker`
- Les `topics` sont maintenus au niveau du `broker`.
  - ne nécessitent pas une définition préalable
  - créés dynamiquement au fur et à mesure de la publication des messages



Le modèle Publish/Subscribe : conséquences

- La communication ne se fait pas de manière directe entre deux entités de l'application (comme c'est le cas du modèle classique client/serveur)
- Le modèle découple la partie qui envoie le message (`publisher`) de celle qui le reçoit (`subscriber`). Les deux parties ne sont jamais en contact direct. En effet, une partie n'est même pas au courant de l'existance de l'autre partie
- Le modèle `pub/sub` est aisni facilement extensible (`scalable`) pour supporter un très grand nombre de clients
- Le `broker` devient le point sensible du système (single point of failure). Des mécanismes existent pour rendre cet élement plus robuste (duplication dans un système réparti)

les Topics MQTT

- Un `topic` est une simple chaîne de caractères ayant eventuellement des niveaux hiérarchiques séparés par des slash (/)
- Exemple: un `topic` utilisé pour communiquer la température dans le séjour d'une maison pourra être `home/living-room/temperature`
- En une souscription donnée, un client peut s'abonner à un `topic` particulier, ou il peut spécifier plusieurs `topics` en utilisant les joker (`wildcard`)
- Deux types de joker : joker à un seul niveau (+) et joker multiniveau (#)

Joker à un seul niveau : +

- Single level wildcard
- Correspond à n'importe quelle valeur à l'intérieur d'un seul niveau de la hiérachie
- Exemple : la souscription à `home/+/temperature` permet de recevoir les températues de toutes les pièces de la maison
  - Couvre :
    - `home/living-room/temperature`
    - `home/room1/temperature`
  - Ne couvre pas :
    - `home/room1/main-light`
    - `home/room1/part1/temperature`
    - `work/desk/temperature`

Joker multiniveau :

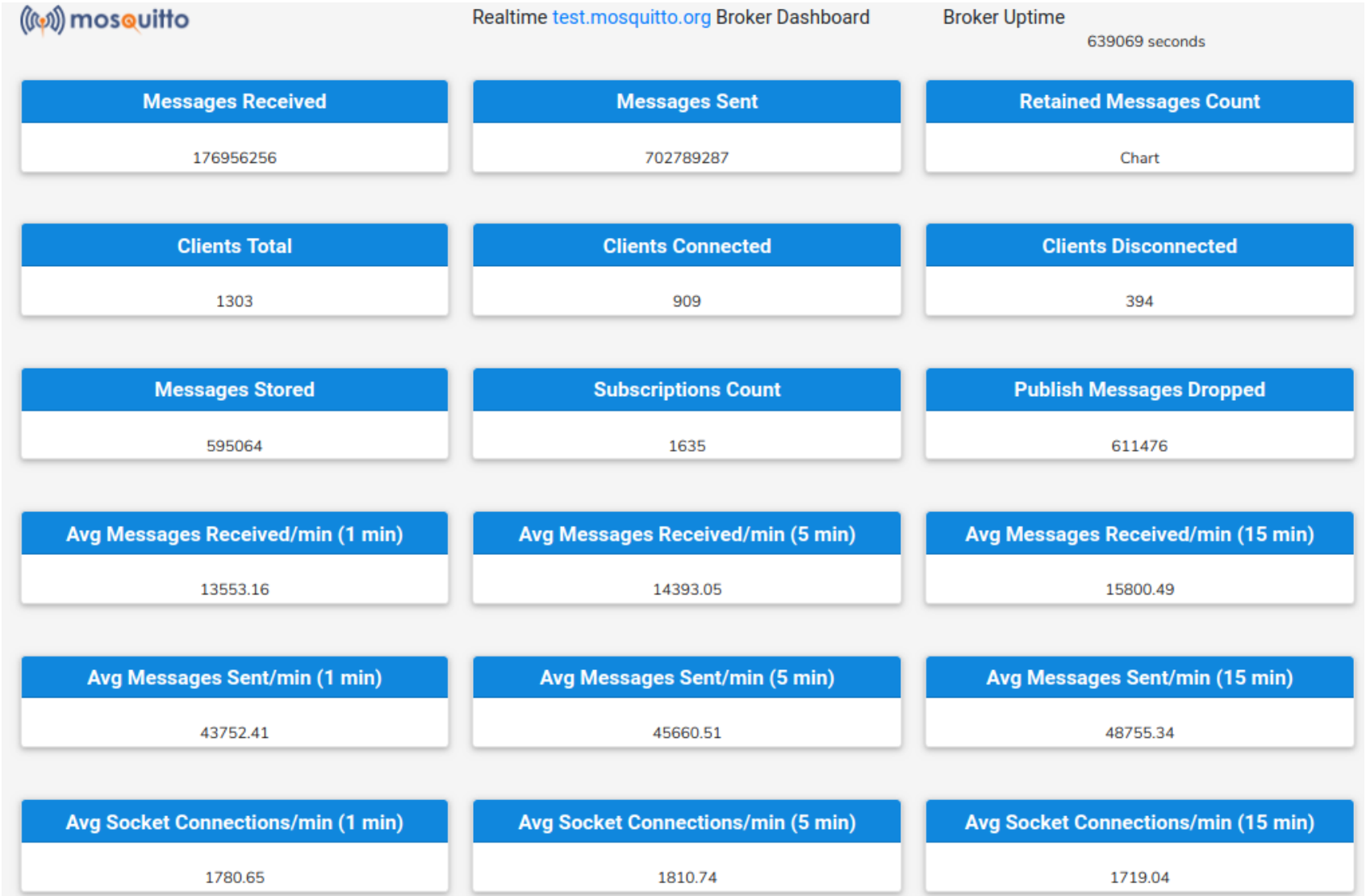
- Multi-level wildcard
- Correspond à n'importe quelle valeur dans n'importe quel niveau sous-jacent de la sous-arborescence
- Exemple : la souscription à `home/#` permet de s'abonner à tous les `topic` se trouvant sous le premier niveau hiérachique `home`
  - Couvre :
    - `home/living-room/temperature`
    - `home/room1/main-light`
    - `home/room1/part1/temperature`
  - Ne couvre pas :
    - `work/desk/temperature`

Topic $SYS

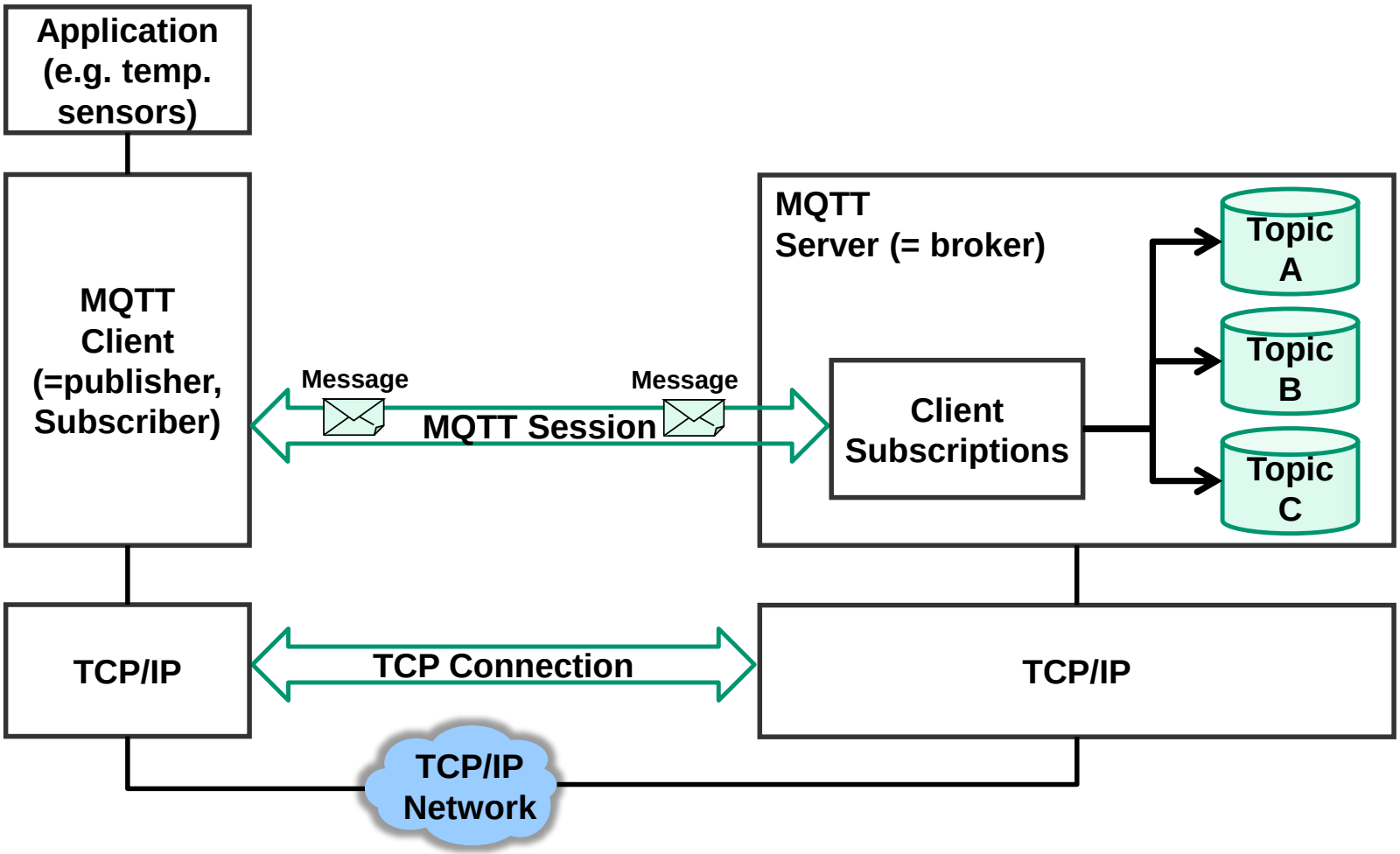
- Topic réservé utilisé par la plupart des `brokers` pour publier des information (statistiques) internes
- Le topic `$SYS` et les topics sous-jacent sont accessibles en lecture seule depuis les clients.
- Exemple : le tableau de bord du `broker` Mosquitto (IBM)[1] visualise les `topics` sous `$SYS`

[1] `http://rtsdp.eu5.org/dashboard/TestMosquittoOrgDashboard.html`

Topic $SYS

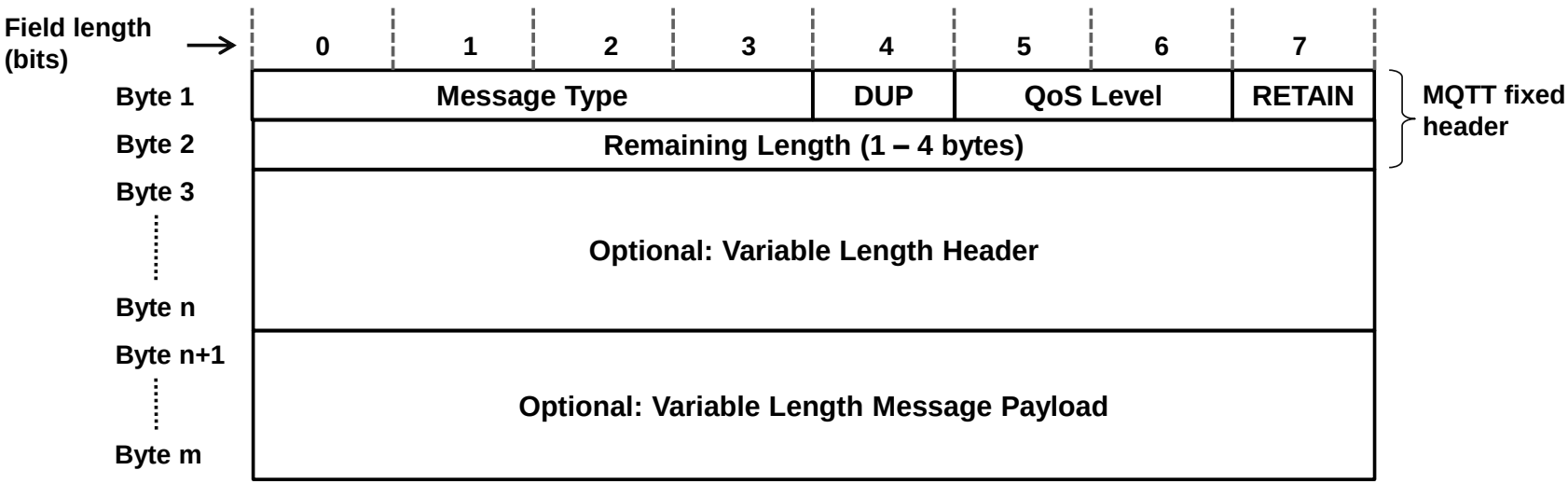

mosquitto
Realtime test.mosquitto.org Broker Dashboard
Broker Uptime
639069 seconds
Messages Received
176956256
Messages Sent
702789287
Retained Messages Count
Chart
Clients Total
1303
Clients Connected
909
Clients Disconnected
394
Messages Stored
595064
Subscriptions Count
1635
Publish Messages Dropped
611476
Avg Messages Received/min (1 min)
13553.16
Avg Messages Received/min (5 min)
14393.05
Avg Messages Received/min (15 min)
15800.49
Avg Messages Sent/min (1 min)
43752.41
Avg Messages Sent/min (5 min)
45660.51
Avg Messages Sent/min (15 min)
48755.34
Avg Socket Connections/min (1 min)
1780.65
Avg Socket Connections/min (5 min)
1810.74
Avg Socket Connections/min (15 min)
1719.04

Position dans le modèle OSI



Format d'un message MQTT

- Trois parties :
  - Entête fixe (obligatoire) de taille 2 octets
  - Entête variable (optionelle) de taille variable selon le type de message
  - Corps du message ou `payload` (optionel) de taille variable selon le type de message



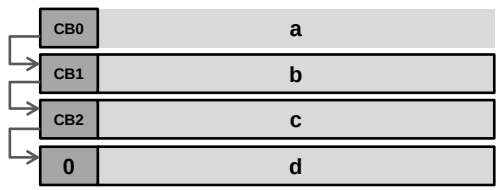
Format d'un message MQTT

- Champs de l'entête fixe :

Message fixed header field	Description / Values	
Message Type	**0: Reserved**	**8: SUBSCRIBE**
	1: CONNECT	**9: SUBACK**
	2: CONNACK	**10: UNSUBSCRIBE**
	3: PUBLISH	**11: UNSUBACK**
	4: PUBACK	**12: PINGREQ**
	5: PUBREC	**13: PINGRESP**
	6: PUBREL	**14: DISCONNECT**
	7: PUBCOMP	**15: Reserved**
DUP	**Duplicate message flag. Indicates to the receiver that this message may have already been received.** **1: Client or server (broker) re-delivers a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message (duplicate message).**	
QoS Level	**Indicates the level of delivery assurance of a PUBLISH message.** **0: At-most-once delivery, no guarantees, «Fire and Forget».** **1: At-least-once delivery, acknowledged delivery.** **2: Exactly-once delivery.** **Voir section sur le niveau de QoS**	
RETAIN	**1: Instructs the server to retain the last received PUBLISH message and deliver it as a first message to new subscriptions.** **Voir section sur le retention de message**	
Remaining Length	**Indicates the number of remaining bytes in the message, i.e. the length of the (optional) variable length header and (optional) payload.**	

Codage de la taille restante

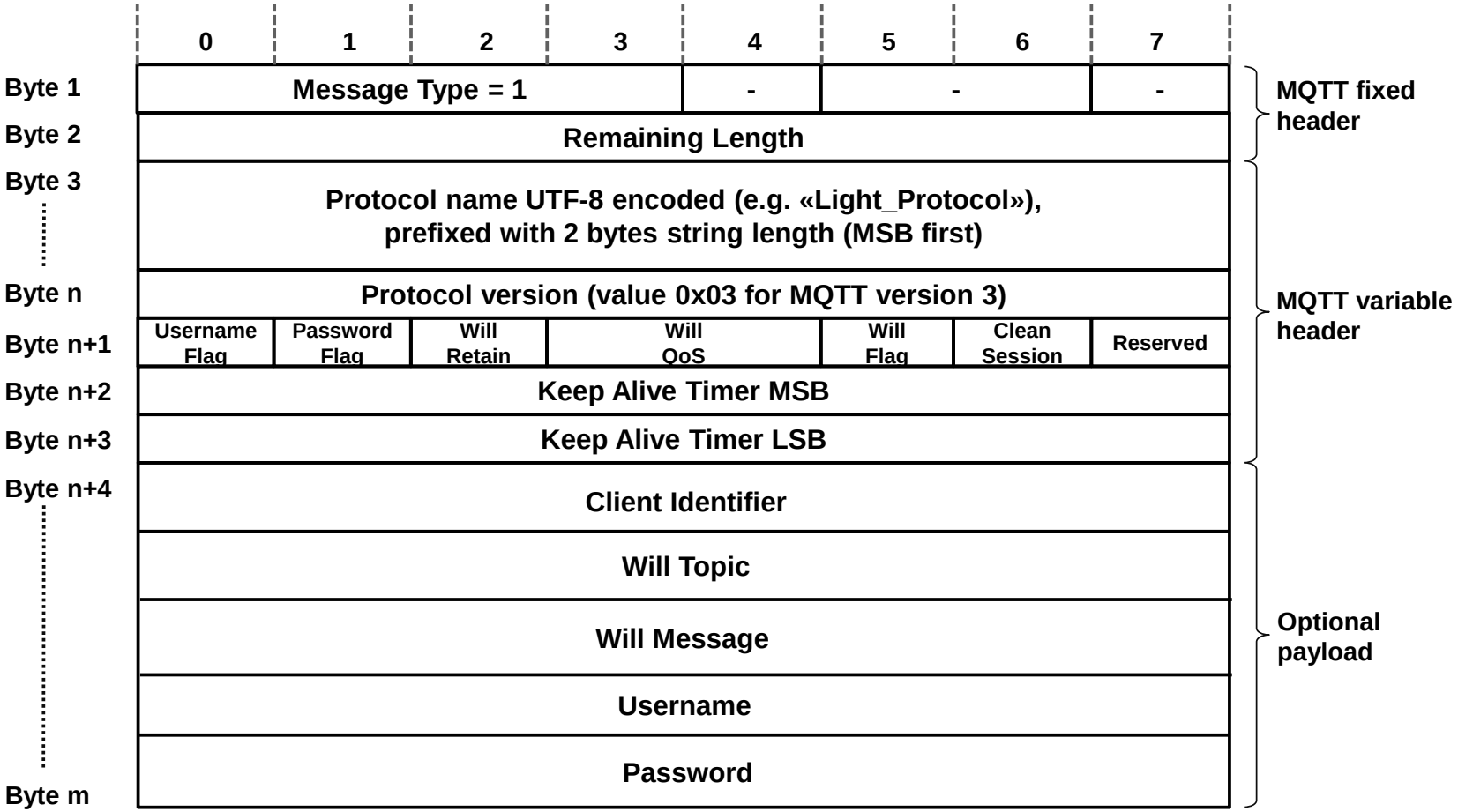
- Champ `remaining length RL` :
- RL = taille de l'entête variable + taille du payload :
- RL sur 1 à 4 octets (pour optimiser la taille totale du message)
  - Premier octet se trouve dans l'entête fixe
  - Les autres octets (s'il y en a) se trouvent dans l'entête variable
  - Le bit de poid le plus fort de chaque octet (appelé bit de continuation ou `continuation bit CB`) indique s'il y a ou pas un octet suivant
    $RL = a + CB0 \times b \times 128^1 + CB1 \times c \times 128^2 + CB2 \times d \times 128^3$



- Exercice : déterminer l'encodage de RL dans le cas où la taille totale du message MQTT est 358

Message de type CONNECT

- Envoyé par le client pour se connecter au `broker`

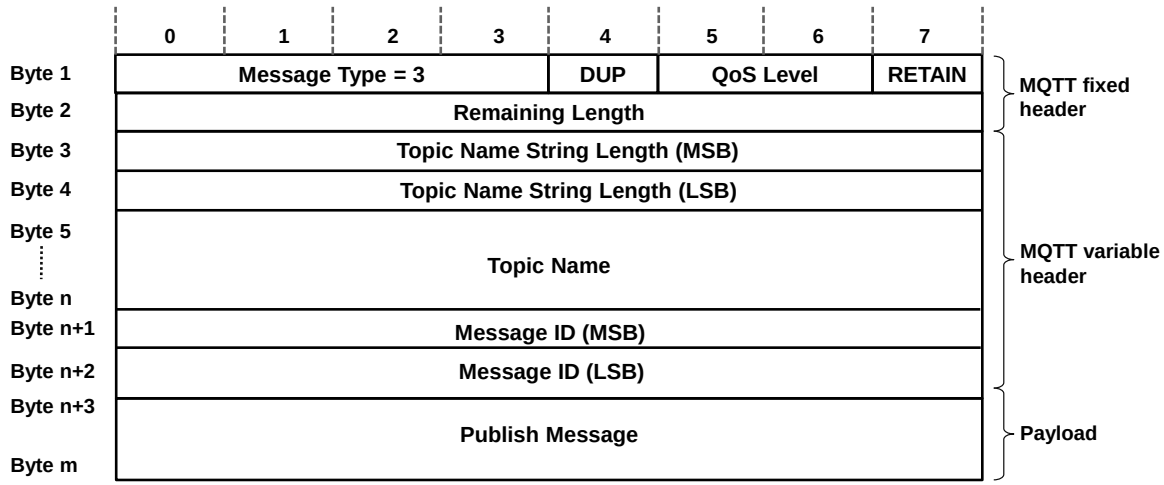


Message de type CONNECT

CONNECT message field	Description / Values
Protocol Name	**UTF-8 encoded protocol name string.** **Example: «Light_Protocol»**
Protocol Version	**Value 3 for MQTT V3.**
Username Flag	**If set to 1 indicates that payload contains a username.**
Password Flag	**If set to 1 indicates that payload contains a password.** **If username flag is set, password flag and password must be set as well.**
Will Retain	**If set to 1 indicates to server that it should retain a Will message for the client which is published in case the client disconnects unexpectedly.**
Will QoS	**Specifies the QoS level for a Will message.**
Will Flag	**Indicates that the message contains a Will message in the payload along with Will retain and Will QoS flags.** **Voir section Last will**
Clean Session	**If set to 1, the server discards any previous information about the (re)-connecting client (clean new session).** **If set to 0, the server keeps the subscriptions of a disconnecting client including storing QoS level 1 and 2 messages for this client. When the client reconnects, the server publishes the stored messages to the client.**
Keep Alive Timer	**Used by the server to detect broken connections to the client.** **Voir section Keep alive**
Client Identifier	**The client identifier (between 1 and 23 characters)uniquely identifies the client to the server. The client identifier must be unique across all clients connecting to a server.**
Will Topic	**Will topic to which a will message is published if the will flag is set.**
Will Message	**Will message to be puslished if will flag is set.**
Username and Password	**Username and password if the corresponding flags are set.**

Message de type PUBLISH

- Envoyé par le client `publisher` au `broker` ou par le `broker` au client `subscriber` :
  - MSB : most significant byte (octet le plus significatif)
  - LSB : least significant byte (octet le moins significatif)



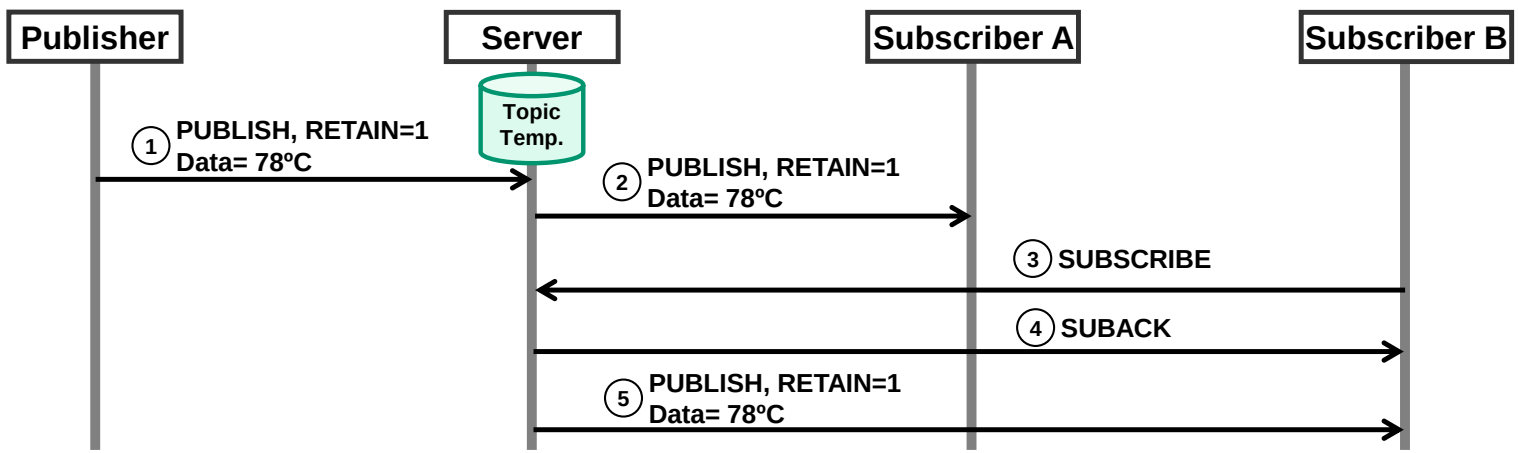
- Exercice 1 : Quelle est la taille maximale du nom du `topic`
- Exercice 2 : Quelle est la taille maximale du payload (pour le cas de la figure) si on suppose que le nom du `topic` est de taille 8 caractères (un caractère = 1 octet)

Message retenu

- Normalement, lorsqu'un message est délivré aux clients concernés (qui sont souscrits au `topic` du message), le message en question est effacé sur le `broker`
- Cependant un client peut envoyer un message avec le bit (flag) `RETAIN=1`.
- Dans ce cas, le message sera traité par le `broker` comme étant un message retenu
  - Le `broker` sauvegardera le message retenu même après l'avoir délivré aux clients concernés
  - Le message sera délivré à toute nouvelle souscription à son `topic` par un client
- Un seul message retenu par `topic` est accepté
  - Si le `broker` reçoit un autre message retenu non vide ayant le même `topic`, le nouveau message remplace l'ancien
  - Pour effacer, au niveau du `broker`, un message retenu dans un `topic` donné, il suffit d'envoyer un message retenu vide au même `topic`

Message retenu

- Scénario typique d'utilisation : le `publisher` met à jour l'état d'une variable dans le système. Ainsi, les `subscribers` reçoivent toujours la dernière bonne valeur connue (last known good value)
- Exemple : le client `subscriber B` vient de se connecter et se souscrit au `topic temperature`. il reçoit immédiatement la dernière bonne valeur connue de la température

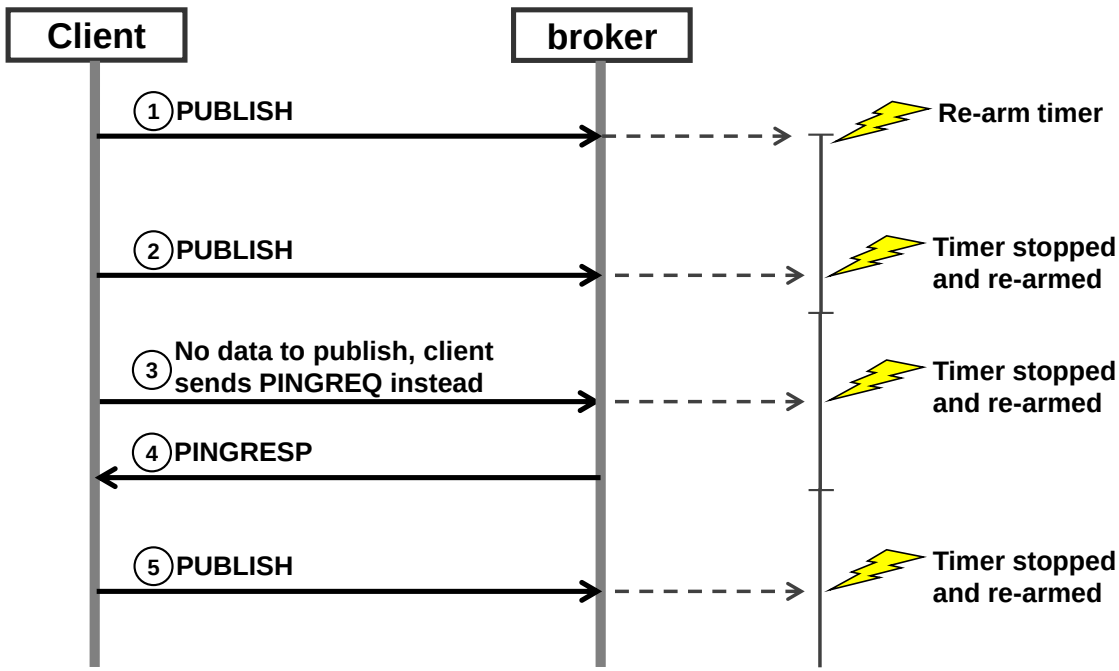


Le mécanisme "Keep alive"

- mécanisme utilisé pour que le `broker` soit notifié lorsqu'un client se déconnecte d'une façon accidentelle
- Exercice : citer quelques raisons derrière une déconnexion accidentelle dans le contexte IoT
- Principe : Si le `broker` ne reçoit aucun message de la part d'un client pendant une période de temps T, il peut conclure que le client est déconnecté
  - La période T = 1.5 * Keep Alive Timer
  - Keep Alive Timer : paramètre utilisé par le client lors de la connexion au `broker` (voir message de type CONNECT). Sa valeur par défaut est 60 secondes
  - En déconnectant le client, le `broker` publie le message `Last Will` du client (s'il y en a un) : voir section suivante
  - Le client peut désactiver le mécanisme de déconnexion en mettant `Keep Alive Timer = 0` lors de la connexion
- Exercice: Déterminer la valuer maximale du `Keep Alive Timer`

Le mécanisme "Keep alive"

- Pour qu'un client "reste en vie", il doit envoyer au moins un message au `broker` tout les T secondes. En l'absence de données utiles, un message PINGREQ sera envoyé [2]



[2] l'envoi du message PINGREQ est géré automatiquement par la plupart des bilbliothèques qui implémentent le protocole MQTT coté client

Le mécanisme "Keep alive"

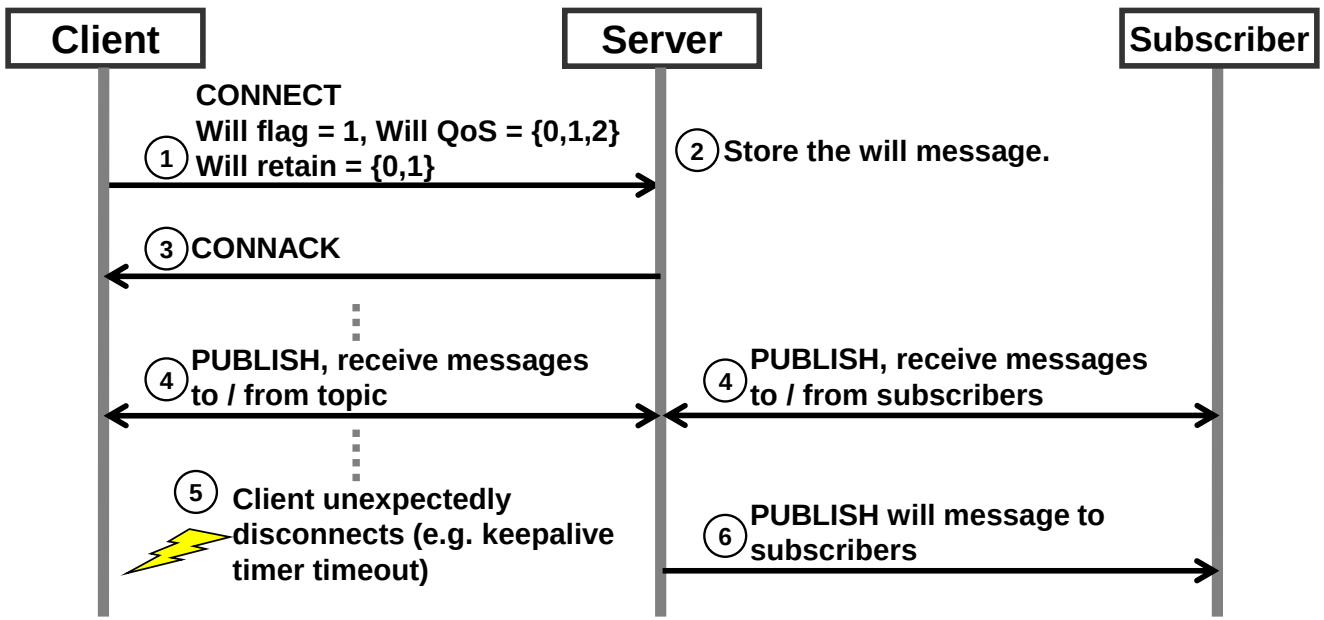
- Le client doit adapter le paramètre `Keep Alive Timer` à son activité
  - Une période trop courte encombre le réseau et augmente la consommation d'énergie coté client
  - Une période trop longue ne permettra pas de notifier rapidement le reste du système de la déconnexion du client
- Exercice
  - Cas 1 : un `device` capteur de température sans fil (`wireless sensor`) opérant sur batterie envoie la température périodiquement chaque 15 min. Entre deux envois successives, le `device` doit passer en mode veille à très faible consommation d'énergie pour faire prolonger l'autonomie de sa batterie à plusieurs années.
  - Cas 2 : un `device` capteur de mouvement opérant sur secteur envoie un message alertant le système d'une éventuelle intrusion
  - Comment faut-il régler le paramètre `Keep Alive Timer` dans les deux cas ?

Le mécanisme "Last Will"

- Ce mecanisme est utilisé conjointement avec le mécanisme `Keep Alive` pour alerter les intervenants du système de la déconnexion accidentelle d'un client.
- Principe
  - En se connectant au `broker` (message CONNECT), un client peut indiquer qu'il a un "Last Will" (dernier voeux) qui doit être "honoré" s'il perd la connexion subitement (mort!).
  - Si le client se déconnecte accidentellement (détecté dans le `broker` via le mécanisme `Keep Alive`), le `broker` publie le message "Last Will" qui sera alors reçu par les clients ayant souscrit au `topic` du message en question
  - Si le client se déconnecte normalement (via le message DISCONNECT), il n'y aura pas envoi du message "Last Will" et le `broker` enlève le message "Last Will" correspondant.

Le mécanisme "Last Will"

- Exemple illustrant le mécanisme "Last Will"
  - Remarque : Le message "Last Will" bénificie des mêmes paramètres que les messages normaux (i.e le flag RETAIN et la qualité de service QoS). Ces paramètres seront spécifiés dans le message de connexion (CONNECT)



Exercice

Dans un système SmartHome, on associe à chaque device un `topic` qui indique son état de connexion (ONLINE ou OFFLINE).
Par exemple, le `topic` myHome/devices/device1/status correspond à l'état du device `device 1`.
Tout client ayant souscrit à ce topic devra être immédiatement notifié de l'état courant ansi que des changements futurs de l'état du (des) device(s) spécifié(s) dans la souscription.

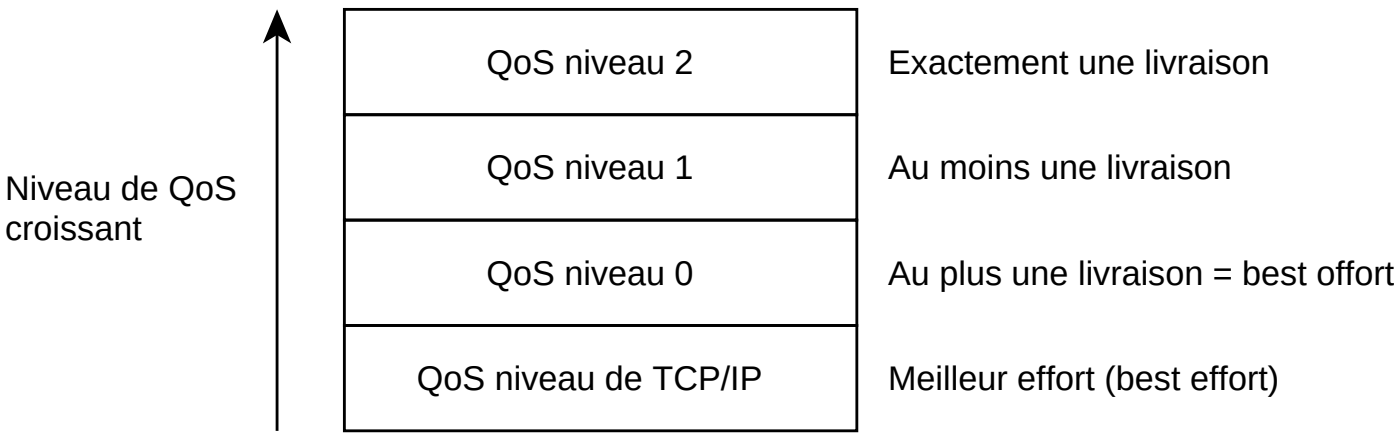
1. Comment une application cliente peut connaître l'état de tout les devices dans le système
2. Expliquer comment ces `topics` sont crées et quelles mécansimes faut-il utiliser pour les mettre à jour conformément à la spécification

Niveaux de qualité de service

- Le niveau de qualité de service (Quality of Service ou QoS) concerne la communication point-à-point entre deux entités : *publisher* $\rightarrow$ *broker* ou *broker* $\rightarrow$ *subscriber*
- Il s'agit d'un niveau d'assurance que le message envoyé est réellement arrivé à destination
- Un niveau de qualité de service plus élevé assure un envoi plus fiable du message, mais consomme plus de bande passante réseau et entraîne un délai supplémentaire

Niveaux de qualité de service

- Le standard MQTT définit trois niveaux de QoS :
  - QoS 0 : Au plus une livraison (at most once). C'est le niveau de qualité de service offert par défaut par la couche TCP/IP (meilleur effort)
  - QoS 1 : Au moin une livrison (at least once)
  - QoS 2 : Exactement une livraison (Exactly once)

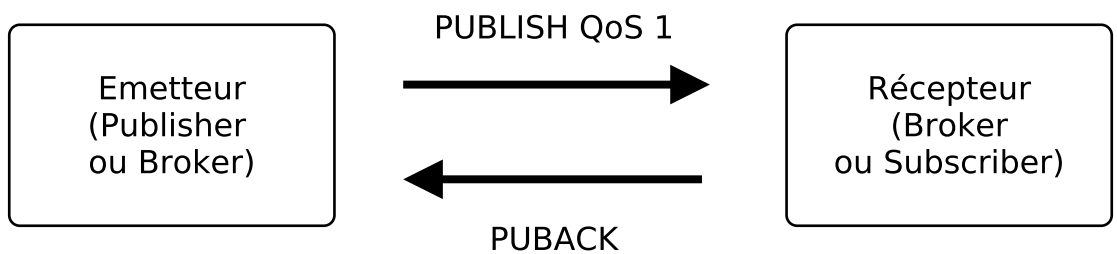


QoS niveau 0

- c'est le moyen le plus rapide pour publier un message, mais qui offre le moins de guarantie sur sa livraison effective
- Les messages sont délivrés selon les garanties fournies par le protocole TCP/IP sous-jacent.
  - Bien que TCP/IP garantisse un transport fiable, une perte du message peut quand même se produire s'il y a une perte de la connexion (exemple: la destination se déconnecte d'une manière accidentelle)
- Aucun aquitement de réception du message n'est envoyé par le récepteur.
- L'émetteur ne garde pas de copie du message envoyé dans se file d'émission. Donc le message ne sera pas ré-envoyé sous aucun cas.

QoS niveau 1

- Le niveau 1 garantie que le message PUBLISH est délivré au moins une fois au récepteur
- L'émetteur garde le message dans sa file d'émission jusqu'à réception d'un message d'aquitement (PUBACK) l'informant que le message a été bien reçu par le récepteur
  - L'identificateur du message PUBLISH (packetId) est utilisé pour reconnaitre le message PUBACK correspondant
  - Si aucun message PUBACK n'est reçu pendant une période de temps raisonnable, l'émetteur ré-envoit le même message PUBLISH en mettant le bit DUB à 1



QoS niveau 1

- Lorsqu'un récepteur reçoit un message avec QoS = 1, il le traite immédiatement et supprime le message de sa file de réception.
  - Si le récepteur est le `broker`, il envoit le message pour tous les `subscribers` et répond par PUBACK
  - Si le `broker` reçoit un message dupliqué (DUB=1), il le traite normalement (en ignorant le champ DUB)

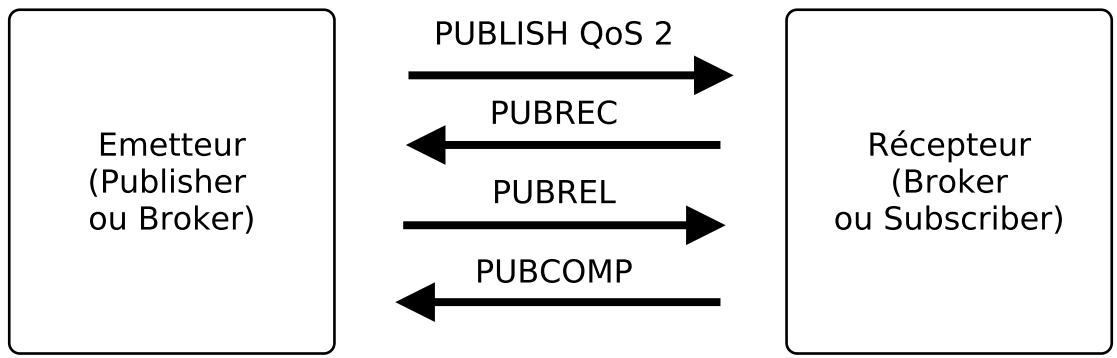
Exercice : Est ce qu'un `subscriber` peut se baser sur le champ DUB d'un message reçu de la part du `broker` pour écarter le message ? justifier.

QoS niveau 2

- Le niveau 2 garantie que le message PUBLISH est délivré exactement une fois au récepteur.
- Lorsque le récepteur reçoit un message PUBLISH pour la première fois (DUP=0), il le traite et répond par un message d'aquitement PUBREC. Cependant, contrairement au niveau QoS 1, le message n'est pas effacé de la file de réception.
- Lorsqu'un émetteur reçoit l'aquitement PUBREC, il efface le message initial de la file d'émission et répond par un message PUBREL
  - Si aucun message PUBREC n'est reçu pendant une période de temps raisonnable, l'émetteur ré-envoit le même message PUBLISH en mettant le bit DUB à 1

QoS niveau 2

- Lorsque le récepteur reçoit PUBREL, il efface le message PUBLISH initial de sa file de réception et répond par un message PUBCOMP pour indiquer la fin de la transaction.
  - Si le récepteur reçoit un message PUBLISH dupliqué, il l'ignore et répond simplement par un PUBREC



QoS de bout-en-bout

- La qualité de service de bout-en-bout est définie entre deux clients : le `publisher` d'un coté et le `subscriber` de l'autre
- C'est la composition de deux niveaux de qualités élémentaires:
  - la QoS du chemin `publisher` $\rightarrow$ `broker`: le niveau de qualité QoS est défini dans chaque message PUBLISH envoyé
  - la QoS du chemin `broker` $\rightarrow$ `subscriber`: le niveau de qualité QoS est défini dans le message de souscription SUBSCRIBE préalablement envoyé par le `subscriber` concernant un `topic` donné et sera appliqué à tous les messages PUBLISH envoyés par le `broker` concernant le même `topic`
- Le niveau de qualité de service résultant (de bout-en-bout) sera le minimum des deux QoS élémentaires

Exercice: illustrer la dernière affirmation par un exemple

Scénario d'utilisation des QoS

- Utiliser le niveau QoS 0 quand :
  - La perte d'un message n'est pas aussi critique sur le fonctionnement du système
  - La connection est très fiable (filaire) et le premier critère est la performance
  - On n'a pas besoin de stocker les messages pendant que les clients sont déconnectés (offline) [3]

Exemple d'application: un capteur de température qui envoit régulièrement des valeurs. La perte d'une valeur n'est pas critique pour l'application puisque les clients vont intégrer les valeurs reçus sur une longue période de temps

[3] le stockage des messages dans le `broker` s'effectue seulement pour les messages de type QoS 1 et 2 et en présence d'une session persistante (voir section suivante)

Scénario d'utilisation des QoS

- Utiliser le niveau QoS 1 quand :
  - On a besoin de garantir la réception de chaque message
  - La réception d'un message duppliqué n'est pas un problème : par exemple, l'application à la possibilié d'écarter un message duppliqué en se basant sur l'identifiant du message
  - On a besoin de stocker les messages pendant que les clients sont déconnectés (offline)
  - On ne supporte pas la dégradation de performance induite par le QoS 2
  - En pratique, le niveau QoS 1 est le plus utilisé

Scénario d'utilisation des QoS

- Utiliser le niveau QoS 2 quand :
  - Il est critique à l'application de recevoir le message exactement une fois. C'est souvent le cas lorsque la réception d'un message dupliqué peut causer le disfonctionnement d'un client
    - exemple, le client déclanche une alarme à la réception d'un message et n'implémente pas un mécanisme de détection de messages dupliqués
  - Il faut noter cependant que ce niveau implique un surcoût important en terme de délai de communication

Session persistante

- Une session est un enregistrement maintenu coté `boker` contenant l'état de la connexion avec chaque client. la session inclut
  - L'ensemble des souscriptions du clients au différents `topics`
  - La liste des messages de type QoS 1 et 2 non encore aquités par le client
  - d'autres information tels ques le compteur utilisé dans le mécansime "Keep Alive", etc.
- Quand un client se connecte à un `broker` (message CONNECT), il a la possibilié d'indiquer le type de session voulue en positionnant le bit "Clean Session"
  - Clean Session = 1 : la session sera effacé à la déconnexion du client
  - Clean Session = 0 : la session est considérée comme durable et sera persistée même si le client est déconnecté. lorsque le client se re-connecte la prochaine fois, la session sera rétablie et les messages QoS 1 et 2 non reçus par le client seront délivrés

Les implementations MQTT

- Plusieurs implémentations libres et commerciales du protocole MQTT existent
- Une liste assez exhaustive est disponible sur le site : mqtt.org/software
- Dans le domaine libre, l'implémentation la plus connue est celle fournie par la fondation Eclipse (IBM)
  - Le `broker` open source Mosquitto
  - La bibliothèque coté client Eclispe Paho qui fournit des interfaces avec la majorité des langages de programmation (C / C++ / Java / Python / Go /JavaScript / C# / Lua)
- Un service cloud gratuit exposant le `broker` Mosquitto est accessible sous : iot.eclipse.org

Cadre expériemental

- Dans la suite du module, on se basera sur l'implémentation (Mosquitto/Paho) comme cadre expérimental
- On utilisera le serice cloud iot.eclipse.org comme `broker`
- On utilisera le binding Python de la bibliothèque client Paho sous Linux
  - La bibliothèque est installé par : pip install paho-mqtt

Exemple de client MQTT utilisant Python/Paho

```python
import paho.mqtt.client as mqtt
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("$SYS/#")

def on_message(client, userdata, msg):
    print(msg.topic+" "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect("iot.eclipse.org", 1883, 60)
client.loop_forever()
```