

UNIVERSITÉ DE LIMOGES - FST



TERMINAUX MOBILES COMMUNICANTS

MASTER 2 CRYPTIS

RAPPORT

BRULEAUX Pierre
JEANJON Valentin

Professeur référent :
BONNEFOI Pierre-François

13 février 2023

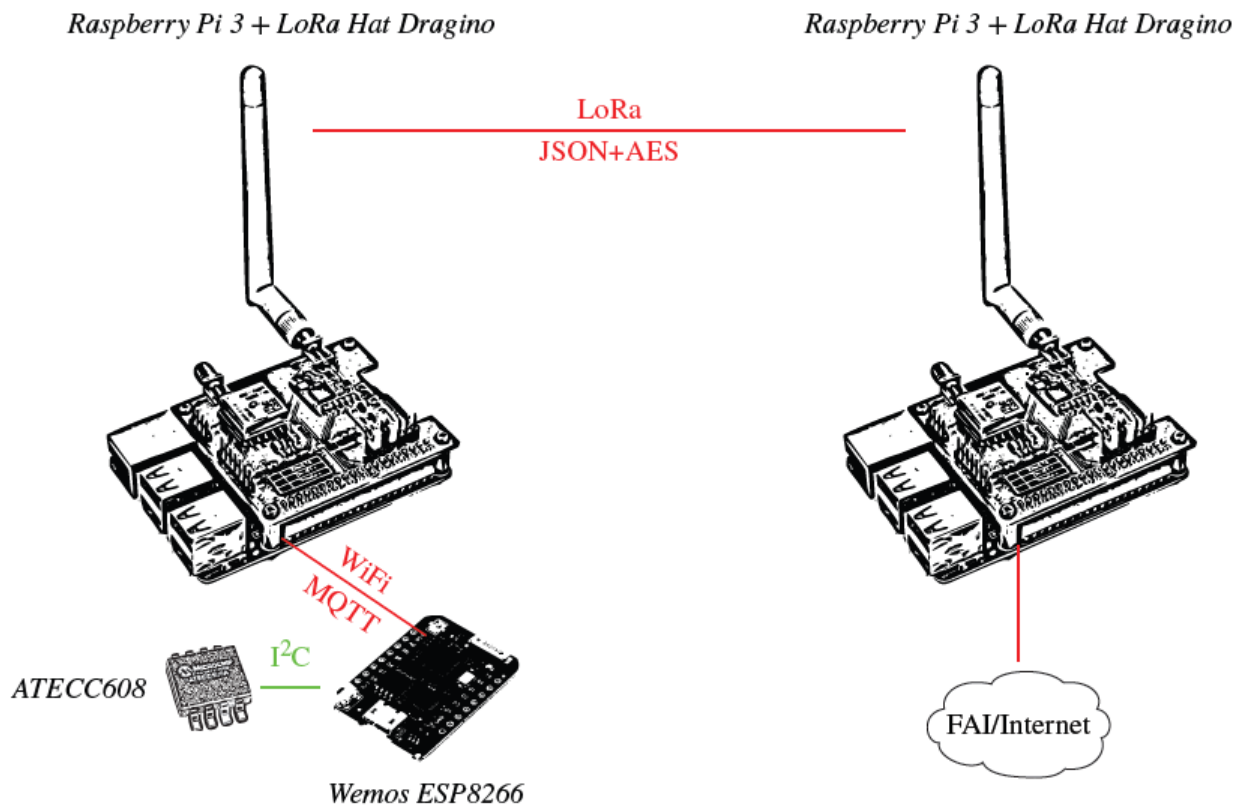
Table des matières

1	Introduction	2
2	Implémentation	3
3	Conclusion	7
3.1	Démonstration	7
3.2	GitHub	7

1 Introduction

Le but du projet était premièrement d'envoyer des messages de façon sécurisés entre une Raspberry Pi et un ESP8266 connecté en WIFI. L'ESP8266 était équipé d'un composant ATECC608 permettant de réaliser les opérations cryptographiques pour l'envoi de message avec le protocole MQTT en mode TLS.

Il fallait ensuite transférer les messages de L'ESP8266 reçu par la Raspberry Pi vers une autre Raspberry Pi. Pour ce faire nous avons utilisé une communication par LoRa. De plus, les messages étaient sous forme JSON et chiffrés avec AES.



2 Implémentation

1. Création des certificats et configuration de Mosquitto :

Pour commencer, nous avons dû paramétrer Mosquitto. Pour ce faire, il y a besoin de créer un utilisateur, générer les certificats pour mettre en place une communication TLS entre l'ESP8266 et la Raspberry Pi et configurer la sécurité de Mosquitto.

Pour créer un utilisateur reconnu par Mosquitto, on utilise la commande suivante. Les utilisateurs et les mots de passe seront stockés dans */etc/mosquitto/mosquitto_passwd*

```
1 $ mosquitto_passwd -c /etc/mosquitto/mosquitto_passwd <user_name>
```

Ensuite, on doit créer le fichier *tls.conf* dans */etc/mosquitto/conf.d* et le paramétrer en lui fournissant l'emplacement des certificats du serveur MQTT comme sur la figure ci-dessous :

```
listener 8883
cafile /home/pi/certifs/ecc.ca.cert.pem
certfile /home/pi/certifs/ecc.serveur.pem
keyfile /home/pi/certifs/ecc.key.pem
require_certificate true
```

FIGURE 1 – MOSQUITTO : *tls.conf*

Les certificats du serveur sont générés avec l'aide de openssl. Il faut faire attention à donner le bon CN au certificat serveur qui correspond au nom DNS associé lors de sa création.

Enfin, pour utiliser Mosquitto, dans notre cas, il faut désactiver son utilisation pour les utilisateurs anonymes donc on va modifier le fichier *mosquitto.conf* en ajoutant les deux lignes suivantes :

```
allow_anonymous false
password_file /etc/mosquitto/mosquitto_passwd
```

FIGURE 2 – MOSQUITTO : *mosquitto.conf*

2. Configuration de MongooseOS (mos.yml) + Installation des clefs/certificats dans l'ESP8266 :

Maintenant, on doit faire en sorte que l'ESP8266 puisse utiliser MQTT correctement.

En premier, on s'occupe de la modification du *mos.yml* pour faire fonctionner MongooseOS avec MQTT, en ajoutant :

```
- ["mqtt.enable", "b", true, {title: "Enable MQTT"}]
- ["mqtt.server", "s", "serveur.iot.com:8883", {title: "MQTT server"}]
- ["mqtt.pub", "s", "/esp8266", {title: "Publish topic"}]
- ["mqtt.user", "s", "esp", {title: "User name"}]
- ["mqtt.pass", "s", "tmctmctmc", {title: "Password"}]
- ["mqtt.ssl_ca_cert", "s", "ecc.ca.cert.pem", {title: "Verify server certificate using this CA bundle"}]
- ["mqtt.ssl_cert", "s", "ecc.esp8266.pem", {title: "Client certificate to present to the server"}]
- ["mqtt.ssl_key", "s", "ATCA:0"]
```

FIGURE 3 – mos.yml : Partie MQTT

Il faut fournir les certificats déclarés dans *mos.yml*, pour ce faire on envoie le certificat de l'AC et de l'ESP8266 dans le dossier fs de l'ESP8266 et on met la clef dans le slot 0. On vérifie si tous les fichiers ont bien été transmis à l'aide de la commande :

```
1 $ mos ls
```

On configure la connexion WiFi avec MongooseOS en faisant bien attention à mettre les mêmes paramètres que dans le fichier de configuration pour le WiFi, *script_ap*, se trouvant sur le Raspberry Pi.

Enfin, on vérifie le bon fonctionnement de MQTT entre l'ESP8266 et le Raspberry Pi, en exécutant :

```
1 $ mos console
```

```
[Feb 2 10:18:59.527] mgos_mqtt_conn.c:180 MQTT0 event: 204
[Feb 2 10:18:59.527] mgos_mqtt_conn.c:118 MQTT0 ack 98
[Feb 2 10:19:01.510] mgos_mqtt_conn.c:154 MQTT0 pub -> 99 /esp8266 @ 1 (9): [Bienvenue]
```

FIGURE 4 – MQTT : Envoi du message

Et on teste sur le Raspberry Pi la réception du message.

```
pi@serveur:~$ mosquitto_sub -C 1 -h serveur.iot.com -p 8883 -t /esp8266 --cafile /home/pi/certifs/ecc.ca.cert.pem --cert /home/pi/certifs/ecc.serveur.pem --key /home/pi/certifs/ecc.key.pem -u pi -P raspberry
Bienvenue
```

FIGURE 5 – MQTT : Réception du message

3. Configuration du client/serveur LoRa :

Il ne reste plus qu'à faire transmettre les données par LoRa. Nous nous sommes basé sur le client et le serveur en C++ déjà présent dans les fichiers RadioHead.

La première étape est de coder l'envoi du message, il faut commencer par récupérer le message MQTT envoyé par l'ESP8266. On a décidé d'intégrer la commande `mosquitto_sub` en mode terminal dans le code à l'aide d'un système de pipe. Donc on exécute :

```
1      mosquitto_sub -C 1 -h serveur.iot.com -p 8883 -t <topic> --cafile  
      <CA> --cert <cert_server> --key <key> -u <user> -P <password>
```

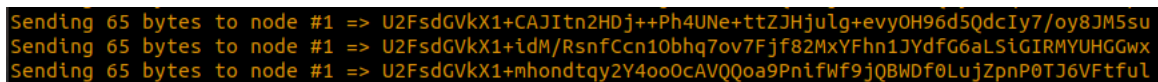
Cela permet d'attendre la réception d'un seul message envoyé par MQTT.

Une fois le message reçu, on doit le mettre au format JSON. La solution choisie a simplement été de concaténer le message pour qu'il ressemble à : `{ "message" : "<MQTTmessage>" }`. Ce JSON sera la donnée transmise.

Mais avant de transmettre, on doit le chiffrer avec AES. Pour ce faire, on a encore utilisé un pipe pour utiliser la commande `openssl` suivante depuis un terminal en C++ :

```
1      echo "<JSON>" | openssl enc -base64 -e -aes-256-cbc -salt -pass  
      pass:<password> -pbkdf2
```

En utilisant seulement un `<password>` avec l'option `-pbkdf2`, `openssl` va créer tous les paramètres nécessaires pour chiffrer et nous redonner seulement le message chiffré. On peut maintenant transmettre le message chiffré par LoRa comme le montre la figure ci-dessous :



```
Sending 65 bytes to node #1 => U2FsdGVkX1+CAJItn2HDj++Ph4UNe+ttZJHjulg+evy0H96d5QdcIy7/oy8JM5su  
Sending 65 bytes to node #1 => U2FsdGVkX1+idM/RsnfCcn10bhq7ov7Fjf82MxYFhn1JYdfG6aLSiGIRMYUHGwX  
Sending 65 bytes to node #1 => U2FsdGVkX1+mhondtqy2Y4oo0cAVQQoa9PnifWf9jQBWdf0LuJZpnP0TJ6VFtful
```

FIGURE 6 – LoRa : Envoi du message

Pour finir, il faut mettre en place la réception du message chiffré. Pour cela, nous avons fait comme pour l'envoi, on utilise une système de pipe pour utiliser la commande openssl de déchiffrement.

```
1      echo "<EncryptMessage>" | openssl enc -base64 -d -aes-256-cbc -  
      salt -pass pass:<password> -pbkdf2
```

En réutilisant le même <password> que pour chiffrer, openssl va s'occuper tout seul du déchiffrement du message chiffré et nous donner le message sous forme JSON, comme illustré sur la figure 7.

```
Packet[65] #10 => #1 -54dB: U2FsdGVkX1+CAJItn2HDj++Ph4UNe+ttZJHjulg+evyOH96d5QdcIy7/oy8JM5su  
Decode: { "message" : "Bienvenue" }  
  
Packet[65] #10 => #1 -56dB: U2FsdGVkX1+ldM/RsnfCcn10bhq7ov7Fjf82MxYFhn1JYdfG6aLSiGIRMYUHGwGwx  
Decode: { "message" : "Bienvenue" }  
  
Packet[65] #10 => #1 -54dB: U2FsdGVkX1+mhondtqy2Y4oo0cAVQOoa9PnifWf9jQBwDf0LujZpnP0TJ6VFtful  
Decode: { "message" : "Bienvenue" }
```

FIGURE 7 – LoRa : Réception du message

3 Conclusion

Le projet était un bon moyen d'interconnecter l'ensemble des solutions vu lors de TPs. Nous avons également pu mettre en pratique une solution réaliste qui pourrait être utilisée dans un contexte réel.

3.1 Démonstration

Lien vers la vidéo de démonstration de la communication : <https://youtu.be/Qb7ng5bNipc>

3.2 GitHub

Lien vers le dépôt GitHub de notre solution : <https://github.com/Jeanjon/RASPI>