



LABORATORIO IOT AND NETWORKS

mTLS con ESP32 e Micropython

Damerini Jacopo

Matricola: 7125740

E-Mail: jacopo.damerini@stud.unifi.it

Filino Alessandro

Matricola: 7125518

E-Mail: alessandro.filino@stud.unifi.it

Macaluso Alberto

Matricola: 7115684

E-Mail: alberto.macaluso@stud.unifi.it

Data Consegna: 28/11/2023

Anno Accademico 2023/24

INDICE

1	Introduzione	3
2	Analisi del problema e valutazione dei possibili protocolli di autenticazione	4
2.1	Utente e Password	4
2.2	Passkey	4
2.3	VPN	5
2.4	mTLS	5
2.5	Protocolli a confronto e realizzabilità	6
3	mTLS - Implementazione	7
3.1	Configurazione	7
3.2	Generazione dei certificati	8
3.3	Caricamento dei moduli	9
3.4	Esempio di utilizzo	10

1 Introduzione

Il presente progetto si propone di affrontare la sfida della sicurezza nella connessione tra un dispositivo embedded, in questo caso un ESP32 che esegue Micropython, e il suo gestore, che può essere un broker locale o basato su cloud. L'obiettivo principale è garantire una comunicazione sicura attraverso un'autenticazione robusta. Nella sezione successiva, [2](#), si esploreranno le diverse opzioni prese in considerazione per risolvere il problema. Queste opzioni includono l'utilizzo di username e password, passkey, connessioni attraverso VPN e l'implementazione di mTLS (Mutual Transport Layer Security). Per ciascuna di queste soluzioni, verranno analizzati vantaggi e svantaggi al fine di individuare la strategia più adatta alle esigenze specifiche del progetto. Successivamente, nella sezione [3](#), verrà fornita una guida dettagliata sull'implementazione della soluzione basata su mTLS e MQTT (con un broker Mosquitto locale). La guida includerà passaggi chiave per la corretta configurazione, fornendo un supporto pratico per l'implementazione. Infine, per illustrare concretamente l'applicazione della soluzione proposta, sarà presentato un esempio pratico di utilizzo del prototipo. Questo esempio fornirà un possibile punto di partenza per l'implementazione in progetti simili.

2 Analisi del problema e valutazione dei possibili protocolli di autenticazione

Nell'ampio scenario dell'Internet degli oggetti (IoT), in cui la connettività tra i vari dispositivi è alla base dei sistemi, la sicurezza diventa un requisito fondamentale. Questo progetto si concentra sulla realizzazione di una soluzione che garantisca un'autenticazione sicura, robusta e affidabile tra dispositivi IoT ed un host.

2.1 Utente e Password

L'utilizzo di semplici credenziali come **utente e password** rappresentano un metodo di autenticazione comunemente utilizzato. Con questo approccio seguono diversi vantaggi tra cui la semplicità di implementazione e l'interoperabilità ossia garantire il funzionamento con diversi sistemi già esistenti. Questo metodo però ha lo svantaggio di essere facilmente vulnerabile. Le credenziali basate su utente e password sono infatti soggette a minacce come attacchi di forza bruta o attacchi di dizionario, rendendo il sistema vulnerabile se le password sono deboli o prevedibili. Inoltre dovremmo cifrare anche il canale di comunicazione per evitare attacchi di tipo man-in-the-middle e tenere al sicuro le password utilizzate. Infine lo schema utente-password è rischioso nel caso in cui dobbiamo gestire gruppi di utenti che condividono caratteristiche comuni: infatti l'amministratore potrebbe essere portato ad utilizzare la stessa password per un gruppo di dispositivi creando un ulteriore punto di vulnerabilità.

2.2 Passkey

L'utilizzo di **passkey** come metodo di autenticazione rappresenta un'alternativa interessante alle classiche credenziali utente e password. In passkey si utilizza una chiave generata a priori e condivisa tra il dispositivo da controllare ed un secondo dispositivo. La condivisione può avvenire attraverso varie modalità, come la visualizzazione su un display o la trasmissione attraverso un'app mobile. A prescindere da questo, l'obiettivo fondamentale è quello di garantire che la chiave raggiunga il secondo dispositivo in modo affidabile e che solo l'utente interessato sia in grado di accedervi. Ogni volta che l'utente desidererà interagire con il dispositivo IoT, dovrà fornire la propria chiave. In questo modo la comunicazione avverrà solo se la chiave fornita dall'utente corrisponderà esattamente a quella memorizzata nel dispositivo.



Figure 1: Schema funzionamento passkey

Con questo approccio troviamo diversi vantaggi tra cui:

1. **Resistenza ad attacchi a forza bruta:** Le chiavi utilizzate sono complesse e robuste
2. **Facilità di memorizzazione:** Essendo stato standardizzato, a differenza di una password complessa la chiave utilizzata può essere facilmente integrata in qualsiasi applicazione mantenendo comunque un elevato standard di sicurezza.

2.3 VPN

L'implementazione di una **Virtual Private Network (VPN)** rappresenta un approccio avanzato per garantire la sicurezza delle comunicazioni tra due dispositivi. Quello che otteniamo è appunto un canale di comunicazione diretto, sicuro e affidabile tra dispositivi. Con questo approccio troviamo diversi vantaggi tra cui:

- **Crittografia E2E:** Le VPN forniscono una crittografia end-to-end delle comunicazioni garantendo così che i dati in transito siano protetti
- **Sicurezza su reti non affidabili:** Venendosi a creare una connessione punto a punto tra due dispositivi, qualora l'utente si connettesse con reti poco affidabili, la comunicazione con il dispositivo rimarrebbe comunque sicura. Questo permette anche di utilizzare in modo sicuro servizi esterni per la gestione dei dispositivi, come per esempio dei broker in cloud.

L'utilizzo delle VPN però hanno lo svantaggio di richiedere un discreto consumo di risorse computazionali.

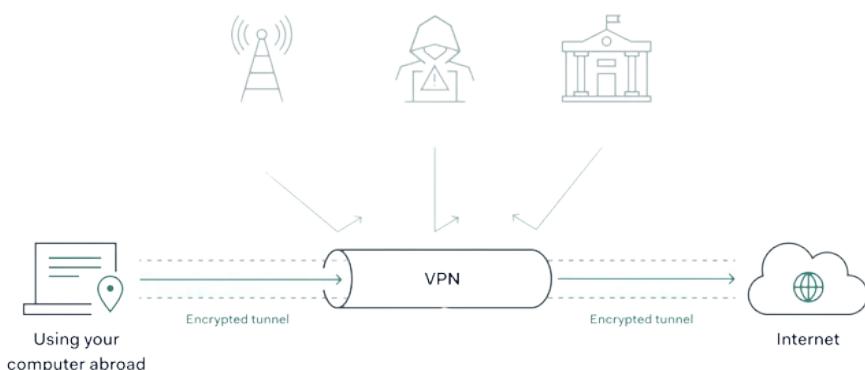


Figure 2: Schema funzionamento VPN

2.4 mTLS

Per garantire la sicurezza delle comunicazioni tra client e server può essere usato mTLS (mutual Transport Layer Security). La principale differenza tra TLS e mTLS risiede nel fatto che mTLS implica una verifica bidirezionale dell'identità durante l'handshake TLS. In un contesto mTLS, sia il client che il server devono autenticarsi reciprocamente prima che la comunicazione possa avere luogo e questo processo di autenticazione è gestito attraverso l'uso di certificati digitali. Il funzionamento generale è il seguente:

1. **Richiesta del client:** Un client che vuole stabilire una connessione sicura con un server invia una richiesta di connessione e presenta il proprio certificato al server.

2. **Risposta del server:** Il server riceve la richiesta del client e, se necessario, verifica il certificato del client. Successivamente, il server presenta il proprio certificato al client.
3. **Autenticazione reciproca:** Ora sia il client che il server si sono autenticati a vicenda, il che significa che entrambe le parti sono certe dell'identità dell'altra.
4. **Negoziazione della chiave di sessione:** Dopo l'autenticazione, le due parti possono negoziare una chiave di sessione condivisa per cifrare e decifrare i dati durante la comunicazione.

L'utilizzo di mTLS, oltre a tutti gli aspetti positivi di TLS, comporta inoltre il vantaggio di evitare attacchi man-in-the-middle poiché entrambe le parti devono autenticarsi reciprocamente. Uno dei possibili svantaggi di questo protocollo risiede nel rilascio e nella gestione dei certificati nel tempo.



Figure 3: Schema funzionamento mTLS

2.5 Protocolli a confronto e realizzabilità

Il metodo di autenticazione attraverso username-password, come ormai risaputo, presente tanti punti di vulnerabilità e quindi è stato scartato in favore di soluzioni più robuste. L'utilizzo di passkey risulta uno dei trend attualmente in maggiore espansione grazie alla forte versatilità ed efficacia di protezione. Purtroppo al momento non esistono implementazioni per micropython di tale protocollo e sarebbe quindi stato necessario sviluppare una libreria apposita. L'uso di una VPN sarebbe un ulteriore ottima soluzione, tuttavia nel caso in esame in cui sono presenti dispositivi con scarse risorse computazionali l'uso di VPN andrebbe a creare un potenziale impatto sulle prestazioni del dispositivo. Inoltre anche in questo caso non è presente un'implementazione di uso di VPN in micropython e quindi sarebbe stato necessario crearla. L'utilizzo di mTLS rappresenta una soluzione di autenticazione perfetta per il caso d'uso ed inoltre risulta più agevole da implementare in micropython. Per l'insieme di questi motivi la scelta si è ricondotta a mTLS.

3 mTLS - Implementazione

Abbiamo progettato il nostro modello con l'idea che i dispositivi comunicassero con uno o più broker tramite protocollo MQTT. Questo è un protocollo leggero e efficiente che consente lo scambio di messaggi tra client e server in modo affidabile e scalabile. La nostra implementazione quindi, è stata condotta in modo tale da rendere possibile l'utilizzo di MQTT su protocollo mTLS. In questo modo abbiamo unito l'efficienza di MQTT con la sicurezza e i vantaggi prima descritti di mTLS.

3.1 Configurazione

Per poter eseguire micropython all'interno di un ESP32, dobbiamo seguire dei passi preliminari:

1. **Installazione di esptool:** Questo è un [tool](#) messo a disposizione per poter flashare il bootloader di un dispositivo ESP32. Per poter utilizzare il tool è necessario clonare la [repo ufficiale](#) oppure tramite il comando `pip install esptool`. Dopodiché scarichiamo il [firmware precompilato di micropython](#) in base alla versione dell' ESP32.

Adesso colleghiamo il dispositivo tramite la porta seriale, per verificare la porta utilizzata dal sistema operativo fare riferimento alla [guida ufficiale](#) (Nel nostro caso è `/dev/ttyUSB0`). Eseguiamo quindi il flash della ROM del dispositivo attraverso il comando `esptool.py --port <NOME_PORTA> erase_flash` e successivamente il comando `esptool.py --chip esp32 --port <NOME_PORTA> write_flash -z 0x1000 <FIRMMWARE>`.

2. **Installazione di mpremote:** Attraverso questo tool, siamo in grado di interfacciarcoci con il dispositivo attraverso command line. Troviamo infatti i [comandi](#) :

- (a) `connect`: Per connetterci al dispositivo
- (b) `soft_reset`:
- (c) `mount <args>`: Carica all'interno del dispositivo la directory passata come argomento (*args*)
- (d) `disconnect`: Per disconnettere il dispositivo in maniera sicura
- (e) ...

È possibile installare mpremote attraverso il comando `pip install --user mpremote`.

3. **Mosquitto:** Mosquitto è un broker MQTT (Message Queuing Telemetry Transport) open source, progettato per facilitare la comunicazione tra dispositivi con limitate risorse di rete, come sensori e attuatori IoT. Mosquitto fornisce un'implementazione robusta di questo protocollo, consentendo a dispositivi distribuiti di scambiare informazioni in modo rapido ed efficiente. È possibile installare mosquitto attraverso il comando: `apt-get install mosquitto` (Linux), `brew install mosquitto` (MacOS) o attraverso il [sito web ufficiale](#).

Conclusa questa prima parte, possiamo procedere con l'implementazione di mTLS.

Cloniamo quindi la [directory del progetto](#). Per prima cosa, all'interno della directory `MQTT-over-mTLS-Micropython-main/main` creiamo un file chiamato `wifi.conf`. Questo conterrà i parametri necessari per connettere il nostro dispositivo con una rete wifi esistente. All'interno troviamo:

- `wifi_ssid`: Contiene il nome della rete wifi a cui andremo a connetterci

- **wifi_psw**: Contene la password di accesso della rete
- **server_ip**: Indirizzo del dispositivo su cui è attivo mosquitto
- **server_port**: Porta di riferimento di mosquitto (ad es. 8883)

Successivamente spostiamoci nella directory `MQTT-over-mTLS-Micropython-main/mainTask/mosquitto` ed avviamo Mosquitto attraverso il comando `mosquitto -v -c mosquitto.conf`. Questo comando avvierà il broker in modalità verbose (-v) attraverso la quale saremo in grado di verificare le connessioni dei dispositivi oltre ai messaggi scambiati.

Inoltre sarà caricata la configurazione che mosquitto dovrà eseguire:

- **per_listener_settings [true]**: Abilita le impostazioni specifiche per ciascun dispositivo connesso, consentendo configurazioni personalizzate per ogni porta o interfaccia di rete
- **persistence [true]**: Abilita la persistenza dei messaggi, consentendo a Mosquitto di archiviare i messaggi inviati e ricevuti su disco, in modo che possano essere recuperati anche dopo un riavvio del broker
- **persistence_file [mosquitto.db]**: Specifica il nome del file di archiviazione dove Mosquitto salverà i messaggi di log
- **persistence_location [./]**: Indica la directory in cui verrà creato il file di log
- **autosave_interval [300]**: Imposta l'intervallo con il quale saranno eseguiti i log dei messaggi. In questo caso è di 300 secondi (5 minuti)
- **retain_available [true]**: Abilita la gestione dei messaggi di tipo "retain", che sono messaggi memorizzati e inviati ai nuovi dispositivi quando si connettono
- **log_timestamp_format [%Y-%m-%d_%H:%M:%S]**: Specifica il formato del timestamp nei log di Mosquitto
- **listener [8883]**: Specifica la porta (8883) per le connessioni MQTT sicure tramite SSL/TLS
- **socket_domain [ipv4]**: Specifica che l'ascoltatore utilizzerà il dominio degli indirizzi IPv4
- **require_certificate [true]**: Richiede la presentazione di un certificato durante la fase di handshake SSL/TLS

3.2 Generazione dei certificati

I certificati necessari per la comunicazione vengono generati attraverso i file bash all'interno della directory `MQTT-over-mTLS-Micropython-main/mainTask/mosquitto/`. I due script [1] sono stati modificati e adattati al nostro caso d'uso:

- **ca_maker.sh**: Questo script è progettato per automatizzare il processo di creazione di una certification authority (CA) e la generazione di un certificato self-signed per il broker. Prima di generare i certificati, è necessario modificare il campo **subject_cn** inserendo l'indirizzo ip del server Mosquitto. Successivamente sono stati definiti parametri utente, come il tipo di chiave (RSA, EC, o ED25519), la curva per le chiavi EC, il numero di bit per le chiavi RSA, e l'opzione per cifrare la chiave della CA. Lo script inoltre effettua un backup di eventuali certificati già esistenti e creando una struttura di directory organizzata. Il processo è seguito da una verifica delle chiavi e dei certificati generati.

- `client_maker.sh`: Questo script semplifica il processo di creazione di certificati per i client che si dovranno connettere al borker. Essi saranno firmati con la stessa CA generata dallo script precedente. Per eseguire lo script è necessario fornire:

- Formato del certificato: Utilizziamo il formato DER
- Nome dell' utente: Ad esempio `user1`

All'interno della directory `MQTT-over-mTLS-Micropython-main/mainTask/mosquitto/nome_utente` troveremo quindi la propria chiave privata e il certificato del dispositivo firmato attraverso la Certification Authority. Lo script è inoltre in grado di generare nuovi certificati per un utente già esistente archiviando i precedenti già presenti.

3.3 Caricamento dei moduli

Per il caricamento dei moduli è stato progettato un apposito script in bash: `loader.sh`. Esso necessita come argomento il nome dell'utente (es `user1`) relativo al certificato da caricare che dovrà essere stato precedentemente generato tramite lo script `client_maker.sh` illustrato prima. Una volta verificata la presenza del certificato, lo script compie le seguenti azioni:

- Effettua la copia del certificato e della chiave selezionati nella cartella
`MQTT-over-mTLS-Micropython-main/mainTask/micropython_data/certs/`
- Tramite l'utilizzo di `mpremote`:
 - Soft-reset dell' ESP
 - Rimozione di tutti i file e le directory presenti nella memoria dell' ESP (tramite lo script `reset.py`)
 - Caricamento dei certificati, del file di configurazione e degli script python (`main.py` e `boot.py`)
 - Esegue il comando `ls` sul dispositivo così da poter verificare che il caricamento sia andato a buon fine

3.4 Esempio di utilizzo

Apriamo lo script `mainTask/mosquitto/ca_maker.sh` e modifichiamo la variabile `subject_cn` con l'indirizzo ip dell'host su cui andremo ad eseguire il broker mosquitto. Nel caso proposto l'indirizzo associato alla macchina è 192.168.1.22. Successivamente eseguiamo lo script.

```

● albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask$ cd mosquitto/
● albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ ./ca_maker.sh
Create EC Key
Certificate request self-signature ok
subject=CN = 192.168.1.22
Private-Key: (256 bit)
priv:
9e:46:01:33:5b:d8:62:b4:1d:90:12:3e:e2:b6:54:
1a:d0:f4:71:ea:5e:46:2d:bc:70:b3:3f:80:38:b5:
eb:53
pub:
04:05:dc:73:09:00:e7:35:cf:85:86:b3:f5:d4:01:
d7:92:d7:03:69:56:f7:09:9d:52:4c:0f:0d:96:24:
5f:b6:4b:9a:a0:c9:48:ed:9d:71:5e:0b:05:07:03:
3a:d1:4a:99:ba:cd:db:a8:11:bd:89:b9:bb:5f:e9:
96:01:70:5b:40
ASN1 OID: prime256v1
NIST CURVE: P-256
Certificate Request:
Data:
Version: 1 (0x0)
Subject: CN = 192.168.1.22
Subject Public Key Info:
Public Key Algorithm: id-ecPublicKey
Public-Key: (256 bit)
pub:
04:05:dc:73:09:00:e7:35:cf:85:86:b3:f5:d4:01:
d7:92:d7:03:69:56:f7:09:9d:52:4c:0f:0d:96:24:
5f:b6:4b:9a:a0:c9:48:ed:9d:71:5e:0b:05:07:03:
3a:d1:4a:99:ba:cd:db:a8:11:bd:89:b9:bb:5f:e9:
96:01:70:5b:40
ASN1 OID: prime256v1
NIST CURVE: P-256
Attributes:
Requested Extensions:
X509v3 Subject Alternative Name:
DNS:192.168.1.22, IP Address:192.168.1.22
Signature Algorithm: ecdsa-with-SHA256
Signature Value:
30:44:02:20:27:29:f2:9f:f3:08:ff:3e:e2:8d:b1:a2:3f:30:
e2:1a:59:0a:ab:e4:5d:b3:7f:00:79:9c:a1:0a:6b:3e:fa:4a:
02:20:4d:1a:ac:c1:0c:ba:b7:9f:de:d1:c4:ca:ee:b5:0c:78:
df:20:75:1c:aa:7d:0a:7f:a6:59:4d:25:df:fa:63:96
Certificate:
Data:
Version: 1 (0x0)
Serial Number:
53:8c:f6:c9:be:65:d7:77:7e:cb:ea:94:f6:29:8a:2a:cb:26:57:0a
Signature Algorithm: ecdsa-with-SHA256
Issuer: CN = MQTT CA
Validity
Not Before: Nov 27 11:26:52 2023 GMT
Not After : Nov 26 11:26:52 2024 GMT
Subject: CN = 192.168.1.22
Subject Public Key Info:
Public Key Algorithm: id-ecPublicKey
Public-Key: (256 bit)
pub:
04:05:dc:73:09:00:e7:35:cf:85:86:b3:f5:d4:01:
d7:92:d7:03:69:56:f7:09:9d:52:4c:0f:0d:96:24:
5f:b6:4b:9a:a0:c9:48:ed:9d:71:5e:0b:05:07:03:
3a:d1:4a:99:ba:cd:db:a8:11:bd:89:b9:bb:5f:e9:
96:01:70:5b:40
ASN1 OID: prime256v1
NIST CURVE: P-256
Signature Algorithm: ecdsa-with-SHA256
Signature Value:
30:46:02:21:00:cb:ab:bf:c2:9d:9d:f9:23:0b:16:1c:90:0c:
b6:04:19:1e:fa:76:fc:6d:0d:2c:7a:9c:df:0c:e2:8a:14:dd:
5a:02:21:00:90:29:71:80:3e:46:1e:34:1e:60:d5:66:bc:
67:fb:1e:8d:92:4e:10:3f:dc:e2:36:62:5c:27:68:e4:0e:8d
albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ █

```

Chiavi della certification authority appena creata

CSR del broker

Certificato rilasciato al broker

Figure 4: ca_maker.sh

Eseguiamo lo script `mainTask/mosquitto/client_maker.sh` con argomenti [der user1] in modo da generare il certificato per il client user1.

```
• albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ ./client_maker.sh der user1
Create EC Key

#####
Certificate request self-signature ok
subject=CN = user1

#####
# This is your new client certificate

Certificate:
  Data:
    Version: 1 (0x0)
    Serial Number:
      2b:b5:ec:5e:9a:f9:a9:cf:20:a8:b0:48:01:eb:60:95:da:49:28:b7
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = MQTT CA
    Validity
      Not Before: Nov 27 11:51:32 2023 GMT
      Not After : Nov 26 11:51:32 2024 GMT
    Subject: CN = user1
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (256 bit)
        pub:
          04:af:23:d4:8b:fb:9e:ab:da:e2:8b:8e:51:95:3c:
          9c:9a:d2:b2:31:63:d7:0d:cf:3d:25:ee:9a:38:cf:
          c1:04:00:de:c9:39:36:78:ce:71:c9:c7:bd:a9:84:
          60:5b:c5:19:72:8b:ad:e9:e4:fe:be:13:91:65:e9:
          ba:76:ec:b3:9d
          ASN1 OID: prime256v1
          NIST CURVE: P-256
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
      30:44:02:20:15:7d:63:b2:be:6e:fb:b9:d4:9e:34:10:4c:0c:
      e2:33:9e:2e:9c:c0:42:f5:99:b7:93:4d:e6:cb:83:74:bd:bb:
      02:20:56:80:97:ae:90:0a:41:99:bf:6b:4a:02:0b:c7:de:9f:
      5d:79:fb:9f:47:fb:aa:d0:6b:1f:e6:b1:f5:f6:b1:ed

#####
# Here are the client files
totale 12
-rw-rw-r-- 1 albe albe 381 nov 27 12:51 ca.crt.der
-rw-rw-r-- 1 albe albe 287 nov 27 12:51 user1.crt.der
-rw-r--r-- 1 albe albe 121 nov 27 12:51 user1.key.der

#####
o albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ █
└─ mosquitto
   └─ certs
      ├ CA
      └─ clients
         └─ user1
            └─ ca.crt.der
            └─ user1.crt.der
            └─ user1.key.der
            └─ ca.crt.der
               └─ ca.crt.pem
            └─ csr_files
            └─ DH
            └─ server
               └─ server.crt.pem
               └─ server.key.pem
```

Certificato user1

←

Directory contenente sia i files relativi al client che al server

←

Figure 5: client_maker.sh

Colleghiamo l'esp32 (con micropython già flashato) e creiamo il file `mainTask/micropython_data/wifi.conf` seguendo le istruzioni illustrate precedentemente nella [3.1](#).

Eseguiamo quindi `mainTask/loader.sh` indicando user1 come utente.

```
● albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ cd ..
● albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask$ ./loader.sh user1
mkdir ./certs
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/certs/esp_crt.der :certs/
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/certs/esp_key.der :certs/
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/certs/ca_crt.der :certs/
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/boot.py :
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/main.py :
cp /home/albe/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/micropython_data/wifi.conf :
ls :
    150 boot.py
    0 certs/
  2472 main.py
   115 wifi.conf
○ albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask$
```

Figure 6: loader.sh

Avviamo mosquitto con la configurazione `mainTask/mosquitto/mosquitto.conf`

```
● albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask$ cd mosquitto/
○ albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask/mosquitto$ mosquitto -v -c mosquitto.conf
2023-11-27 13:00:23: mosquitto version 2.0.11 starting
2023-11-27 13:00:23: Config loaded from mosquitto.conf.
2023-11-27 13:00:23: Opening ipv4 listen socket on port 8883.
2023-11-27 13:00:23: mosquitto version 2.0.11 running
■
```

Figure 7: avvio mosquitto

Aprimmo un'interfaccia repl sull'esp32 tramite l'utilità `mpremote` e importiamo il `main.py` per eseguire l'inizializzazione.

```
○ albe@Albe-pc:~/Scrivania/Unifi/MQTT-over-mTLS-Micropython/mainTask$ mpremove
Connected to MicroPython at /dev/ttyUSB0
Use Ctrl-] or Ctrl-x to exit this shell

>>> import main
Read Wifi Config... OK
Trying Wifi connection
Wifi Connection... OK
Read CA Certificate... OK
Read User Certificate... OK
Read User Key... OK
```

**Inizializzazione
avvenuta correttamente**

Figure 8: inizializzazione esp32

Avviamo la connessione al broker eseguendo la funzione `start_client()` sempre attraverso l' interfaccia repl.

<pre>>>> main.start_client() Connecting to MQTT Server... Sending ON Sending OFF Sending ON Sending OFF Sending ON Sending OFF Sending ON</pre> <p>Esp32</p>	<pre>2023-11-27 13:04:25: New connection from 192.168.1.44:49299 on port 8883. 2023-11-27 13:04:25: New client connected from 192.168.1.44:49299 as client1 (p2, c1, k60, u'user1'). 2023-11-27 13:04:25: No will message specified. 2023-11-27 13:04:25: Sending CONNACK to client1 (0, 0) 2023-11-27 13:04:25: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes)) 2023-11-27 13:04:27: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (3 bytes)) 2023-11-27 13:04:27: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes)) 2023-11-27 13:04:28: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (3 bytes)) 2023-11-27 13:04:30: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes)) 2023-11-27 13:04:31: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (3 bytes)) 2023-11-27 13:04:32: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes)) 2023-11-27 13:04:33: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (3 bytes)) 2023-11-27 13:04:34: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes)) 2023-11-27 13:04:35: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (3 bytes)) 2023-11-27 13:04:36: Received PUBLISH from client1 (d0, q0, r0, m0, 'test/topic01', ... (2 bytes))</pre> <p>Mosquitto</p>
--	--

Figure 9: connessione al broker

References

- [1] JustinS-B. *Mosquitto CA and Certs*. https://github.com/JustinS-B/Mosquitto_CA_and_Certs.