

**ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA**

**SCUOLA DI INGEGNERIA E ARCHITETTURA**

**Dipartimento di Informatica - Scienze e Ingegneria**

*Corso di Laurea Magistrale in INGEGNERIA INFORMATICA*

*Esame di MOBILE SYSTEMS M*

# **Relazione Attività Progettuale**

*Analisi protocollo MQTT in base a QoS e Payload Encryption*

**Professore:** Paolo Bellavista

**Studente:** Luigi Maccallini

# Indice

<b>1. Introduzione.....</b>	<b>3</b>
<b>2. Broker MQTT.....</b>	<b>4</b>
2.1 Configurazione e avvio del broker Mosquitto.....	4
<b>3. Android Client .....</b>	<b>6</b>
3.1 Struttura dell'applicazione .....	6
3.2 Funzionamento.....	7
3.2.1 MainViewModel .....	8
3.2.2 FirstFragment .....	9
3.2.3 SecondFragment.....	11
3.3 Payload Encryption e QoS .....	12
<b>4. Test .....</b>	<b>15</b>
<b>5. Conclusioni .....</b>	<b>18</b>
<b>6. Bibliografia .....</b>	<b>19</b>

# 1.Introduzione

Il progetto si base sull'utilizzo di **MQTT**, protocollo: semplice, leggero, basato sui broker (responsabili della distribuzione dei messaggi ai client interessati), basato sul modello publish/subscribe (distribuzione di messaggi uno a molti) e aperto, ideale per l'uso in dispositivi molto limitati oppure quando la rete risulta costosa, con poca larghezza di banda o è inaffidabile. MQTT si posiziona sopra a TCP/IP, crea connessioni durevoli con lo scopo di prevenire la mancata consegna dei messaggi in caso di disconnessioni dei client e presenta tre semantiche di consegna: *at least once* (non c'è garanzia di consegna), *at most once* (garantisce che un messaggio venga consegnato almeno una volta ai destinatari), *exactly once* (garantisce che ogni messaggio venga ricevuto una sola volta dai destinatari previsti).

L'obiettivo del progetto è quello di partire dalla realizzazione di un'**applicazione Android** che si connetta ad un **broker MQTT**. Lo scopo dell'app deve essere quello di permettere all'utente di poter effettuare la sottoscrizione ad un topic, per poi mostrare a video tutti i messaggi che verranno pubblicati su di esso. Nella fase finale del progetto, è stata effettuata un'analisi dei messaggi scambiati tra il client ed il broker, analizzando, nello specifico, il consumo delle risorse e tempo di latenza in base alla *QoS* (Qualità del Servizio) utilizzata e all'utilizzo o meno della *crittografia del payload del messaggio*.

Partendo dalla presentazione del broker scelto e presentando la configurazione dello stesso, si passerà poi alla descrizione dell'applicazione Android realizzata, descrivendo il funzionamento della stessa e le scelte fatte per quanto riguarda la crittografia del payload e la qualità del servizio. Infine, verranno descritti i test effettuati e fatte delle considerazioni finali sul lavoro svolto.

## 2. Broker MQTT

La scelta del broker MQTT è ricaduta su Eclipse Mosquitto, broker di messaggi open source che implementa le versioni del protocollo MQTT 5.0, 3.1.1 e 3.1 (EclipseMosquitto™). Tra le diverse utility proposte, quelle utilizzate nel progetto sono:

- ✓ `mosquitto.conf` → configurare il broker Mosquitto
- ✓ `mosquitto` → gestire e avviare un broker Mosquitto
- ✓ `mosquitto_passwd` → generazione di file di password
- ✓ `mosquitto_pub` → pubblicare messaggi su un topic
- ✓ `mosquitto_sub` → effettuare una sottoscrizione ad un topic o crearne uno

### 2.1 Configurazione e avvio del broker Mosquitto

Prima di poter avviare il broker Mosquitto, bisogna effettuare la configurazione dello stesso. Per farlo si può o intervenire direttamente nel file `mosquitto.conf`, oppure possiamo creare un file di configurazione a parte. La scelta è ricaduta sulla seconda opzione, così da poter effettuare una configurazione adeguata, evitando quindi di modificare i parametri di default del broker.

Per la configurazione è stato creato il file **broker.conf**, in cui sono stati specificati i seguenti parametri:

- **listener 1883** → ascoltare la connessione di rete in entrata sulla porta specificata (1883 in questo caso)
- **per\_listener\_settings true** → impostato a *true* indica che le impostazioni di autenticazione e controllo dell'accesso (in questo caso la `allow_anonymous` e la `password_file`) saranno applicate solo sul listener che si sta configurando
- **allow\_anonymous false** → impostato a *false* indica che i client non si possono connettere senza autenticazione
- **password\_file passwd.txt** → indica la posizione ed il nome del file delle password
- **log\_type all** → specifica i tipi di messaggio da inserire nei log (in questo caso tutti)
- **log\_dest file mosquitto.log** → specifica dove salvare i log (in questo caso in un file)
- **log\_timestamp true** → impostato a *true* indica che bisogna attribuire un timestamp ad ogni log
- **log\_timestamp\_format %d/%m/%Y %H:%M:%S** → imposta il formato del timestamp

Per quanto riguarda la creazione delle credenziali, che i client dovranno utilizzare per connettersi al broker, bisogna specificare nel file `password.txt` (indicato nel `broker.conf`) la coppia username e password, una per riga, nel formato `"username:password"`. Una volta specificate le credenziali valide, si sfrutta l'utility `mosquitto_passwd` per crittografare il file e rendere quindi illeggibili le credenziali.

Una volta create le credenziali, si procede con l'avvio del broker Mosquitto tramite il comando *"mosquitto -c broker.conf"* eseguito da riga di comando (previa corretta configurazione delle variabili d'ambiente in Windows). Da questo momento in poi il broker sarà in ascolto sulla porta 1883, come specificato nel file di configurazione, in attesa di ricevere sottoscrizioni o pubblicazioni. Per testare il corretto funzionamento del broker, sono state sfruttate le due utility *mosquitto\_sub* e *mosquitto\_pub*.

Tramite il comando *"mosquitto\_sub -h [hostname] -p [port-number] -u [username] -P [password] -t [topic]"* si effettua la sottoscrizione al topic specificato (se non esistente, esso viene creato dal broker) e si rimane in ascolto dei possibili messaggi che verranno pubblicati su di esso.

Tramite il comando *"mosquitto\_pub -h [hostname] -p [port-number] -u [username] -P [password] -t [topic] -m [message]"* si effettua la pubblicazione di un messaggio su un topic (anche in questo caso, se il topic non esiste viene creato).

Specificando lo stesso nome del topic nei due comandi precedenti, possiamo testare la corretta pubblicazione del messaggio e successiva ricezione da parte del subscriber.

## 3.Android Client

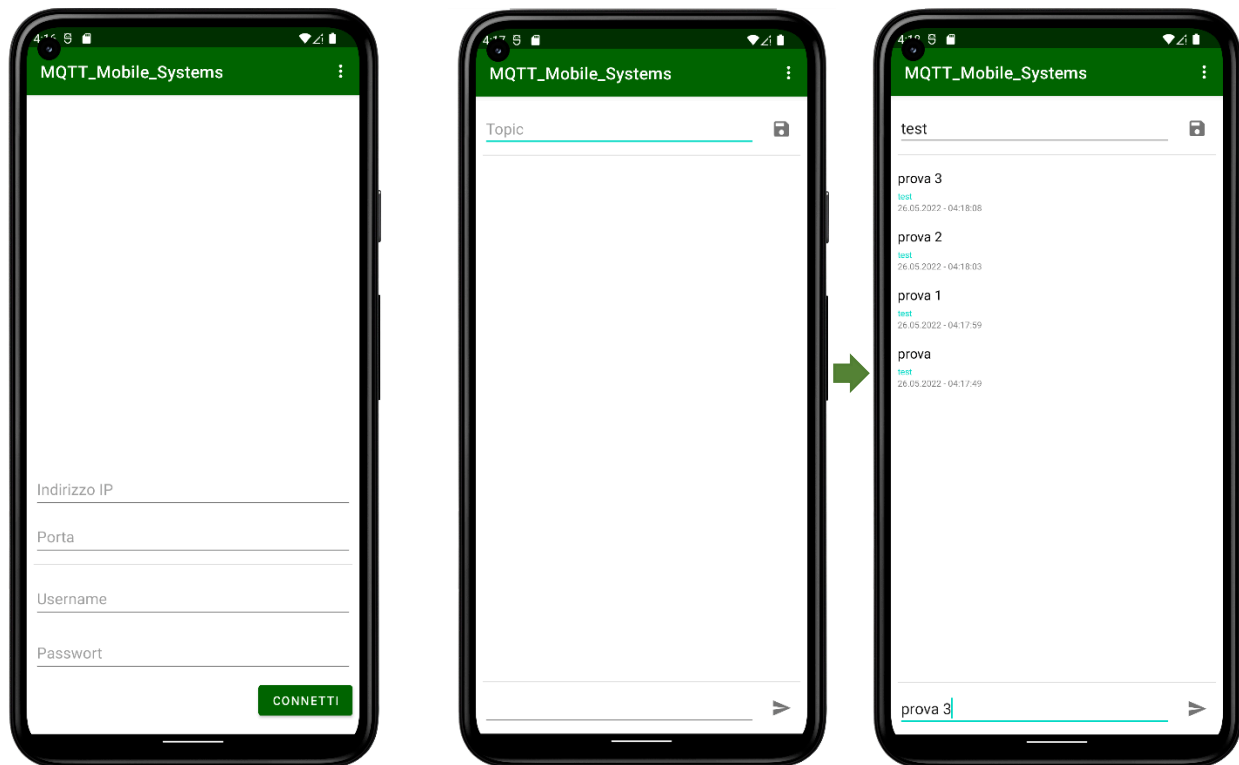
L'applicazione Android ha lo scopo di fungere da client, permettendo innanzitutto una connessione al broker, per poi mostrare un'interfaccia in cui si possa: specificare il topic a cui si vuole effettuare la sottoscrizione, mostrare i vari messaggi pubblicati sul topic ed effettuare una pubblicazione sul topic specificato.

Di seguito verrà presentata la struttura dell'applicazione, il funzionamento della stessa e, infine, l'impostazione della *payload encryption* e della *QoS*.

### 3.1 Struttura dell'applicazione

L'applicazione presenta due schermate:

1. Una prima schermata (schermata di sinistra in **Figura 1**) costituita da un pulsante per avviare la connessione e quattro campi testuali (obbligatori), in cui bisogna specificare i dati necessari per la connessione al broker Mosquitto:
  - *IP*
  - *Porta*
  - *Username*
  - *Password*
2. Effettuata la connessione, viene mostrata una seconda schermata (schermate di destra in **Figura 1**) costituita da:
  - Campo testuale per l'inserimento del nome del *topic* e relativo pulsante per l'invio della richiesta di sottoscrizione;
  - *RecyclerView* che mostra i messaggi che sono stati pubblicati sul topic scelto (ogni elemento della lista è costituito da: *testo del messaggio*, *nome del topic* e *timestamp*);
  - Campo testuale per l'inserimento del messaggio da pubblicare nel topic specificato e relativo pulsante per l'invio della richiesta di pubblicazione.



**Figura 1:** Schermate d'esempio dell'applicazione Android

## 3.2 Funzionamento

In questa sezione verranno analizzate le classi ed i metodi principali, che vanno a realizzare la logica di base dell'app e tutte le funzionalità necessarie per l'interazione con il broker MQTT.

Per poter implementare il protocollo MQTT nell'app è stato configurato il file gradle, specificando tra le dipendenze quanto segue:

```
implementation 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.1.0'
implementation 'org.eclipse.paho:org.eclipse.paho.android.service:1.1.1'
```

Grazie all'inserimento di queste due dipendenze è possibile utilizzare l'*Eclipse Paho Java Client* (Eclipse, s.d.), una libreria per la creazione di client MQTT scritta in Java.

### 3.2.1 MainViewModel

La classe *MainViewModel* si occupa di gestire i dati relativi all'interfaccia utente, ovvero i messaggi ricevuti dal broker, organizzandoli in una "MutableLiveData", proponendo, inoltre, le funzioni necessarie per la comunicazione con il broker, tra cui:

- **initClient**: funzione che si occupa dell'inizializzazione di *MqttAndroidClient* (classe di *Eclipse Paho Java Client* per l'integrazione del protocollo MQTT);

```
fun initClient(serverUri:String, clientId:String) {  
    mqttAndroidClient = MqttAndroidClient(context, serverUri, clientId)  
}
```

- **connectClient**: funzione che si occupa della connessione dell'app al broker MQTT e dell'impostazione della funzione di callback per l'aggiornamento della lista dei messaggi ricevuti, tramite la funzione *addNewMessageToList*;

```
fun connectClient(username:String, pwd:String, callback:(status:Boolean)->Unit) {  
    mqttAndroidClient.setCallback(object:MqttCallbackExtended {  
        override fun connectionLost(cause: Throwable?) {  
        }  
  
        override fun messageArrived(topic: String?, message: MqttMessage?) {  
            addNewMessageToList(message.toString(),topic!!, getStringFromDate())  
        }  
  
        override fun deliveryComplete(token: IMqttDeliveryToken?) {  
        }  
  
        override fun connectComplete(reconnect: Boolean, serverURI: String?) {  
        }  
    })  
  
    val options = MqttConnectOptions()  
    options.password = pwd.toCharArray()  
    options.userName = username  
    try {  
        mqttAndroidClient.connect(options, null, object : IMqttActionListener {  
            override fun onSuccess(asyncActionToken: IMqttToken?) {  
                callback(true)  
            }  
  
            override fun onFailure(asyncActionToken: IMqttToken?, exception: Throwable?) {  
                exception?.printStackTrace()  
                callback(false)  
            }  
        })  
    } catch (e:MqttException) {  
        e.printStackTrace()  
        callback(false)  
    }  
}  
  
private fun addNewMessageToList(msg:String, topic:String, time:String) {  
    val temp = Message(msg, topic, time)  
    val tempData = messages.value!!  
    tempData.add(0,temp)  
  
    messages.value = tempData  
}
```



- **subscribe**: funzione per la sottoscrizione ad un topic;

```

fun subscribe(topic:String, qos:Int = 0, callback: (status: Boolean) -> Unit) {
    try {
        mqttAndroidClient.subscribe(topic, qos, null, object : IMqttActionListener {
            override fun onSuccess(asyncActionToken: IMqttToken?) {
                callback(true)
            }

            override fun onFailure(asyncActionToken: IMqttToken?, exception: Throwable?) {
                exception?.printStackTrace()
                callback(false)
            }
        })
    } catch (e:MqttException) {
        e.printStackTrace()
    }
}

```

- **publish**: funzione per la pubblicazione di un messaggio sul topic specificato;

```

fun publish(topic:String, msg:String, qos:Int = 0, callback: (status: Boolean) -> Unit) {
    try {
        val message = MqttMessage()
        message.payload = msg.toByteArray()
        message.qos = qos
        message.isRetained = false
        mqttAndroidClient.publish(topic, message, null, object : IMqttActionListener {
            override fun onSuccess(asyncActionToken: IMqttToken?) {
                callback(true)
            }

            override fun onFailure(asyncActionToken: IMqttToken?, exception: Throwable?) {
                callback(false)
            }
        })
    } catch (e:MqttException) {
        e.printStackTrace()
    }
}

```

### 3.2.2 FirstFragment

La classe *FirstFragment* si occupa dell'interazione e scambio di dati tra la prima view (ovvero la prima schermata mostrata in precedenza) ed il *MainViewModel*, catturando quindi i dati inseriti dall'utente e sfruttando le funzioni messe a disposizione dal *MainViewModel* per avviare la connessione al broker. Tra le funzioni presenti, le principali sono:

- **onViewCreate:** funzione, eseguita una volta creata la view, che si occupa di ottenere il ViewModel ed avviare la connessione al broker (se tutti i campi sono stati compilati) tramite la funzione *connectToBroker*;

```

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    viewModel = ViewModelProvider(requireActivity(), defaultViewModelProviderFactory).get(
        MainViewModel::class.java)

    first_btn_login.setOnClickListener {

        if(checkInputs()) {
            connectToBroker()
        } else {
            printToast(requireContext(), "Si prega di compilare tutti i campi...")
        }
    }
}

```

- **connectToBroker:** funzione che: blocca gli input, estrae i dati dai campi testuali della view (IP, porta, username, password), inizializza il client tramite la funzione *initClient* del ViewModel e richiede la connessione al broker tramite la funzione *connectClient* del ViewModel. Se la connessione va a buon fine richiede lo spostamento alla seconda schermata, altrimenti vengono sbloccati gli input e specificato l'errore di connessione.

```

private fun connectToBroker() {

    showViews(first_pb, first_tv_pb)
    blockInputs()

    val ip = "tcp://" + first_et_ip.text.toString()
    val port = first_et_port.text.toString()
    val serverUri = "$ip:$port"
    val clientId = "MobylyeSystemsMQTT"
    viewModel.initClient(serverUri, clientId)

    val pwd = first_et_pwd.text.toString()
    val username = first_et_name.text.toString()
    viewModel.connectClient(username, pwd) { status->

        if(status) {
            findNavController().navigate(R.id.action_first_sec2nd)
        } else {
            printToast(requireContext(), "Impossibile stabilire una connessione...")

            first_btn_login.setOnClickListener {
                if(checkInputs()) {
                    connectToBroker()
                } else {
                    printToast(requireContext(), "Si prega di compilare tutti i campi...")
                }
            }
        }
    }

    unblockInputs()
    hideViews(first_pb, first_tv_pb)
}

```

### 3.2.3 SecondFragment

La classe *SecondFragment*, invece, si occupa dell'interazione e scambio di dati tra la seconda view (ovvero la seconda schermata mostrata in precedenza) ed il *MainViewModel*, gestendo quindi la sottoscrizione ad un topic, la pubblicazione di un messaggio nel topic e la visualizzazione di tutti i messaggi del topic. Tra le funzioni presenti, la principale è:

- **onViewCreate**: funzione, eseguita una volta creata la view, che si occupa di ottenere il *ViewModel*, impostare il *Listener* per avviare la richiesta di sottoscrizione al topic quando l'utente lo richiede, impostare il *Listener* per avviare la pubblicazione del messaggio sul topic quando l'utente lo richiede, inizializzare ed aggiornare la *RecyclerView* prelevando i messaggi dal *ViewModel*.

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    viewModel = ViewModelProvider(requireActivity(), defaultViewModelProviderFactory).get(
        MainViewModel::class.java)

    initRecyclerView()
    second_btn_save.setOnClickListener {

        if (!second_et_topic.text.isNullOrEmpty()) {
            viewModel.subscribe(second_et_topic.text.toString()) { status->
                if(status)
                    printToast(requireContext(), "La sottoscrizione al Topic ha avuto successo")
                else
                    printToast(requireContext(), "La sottoscrizione al Topic NON ha avuto successo")
            }
        }
    }

    second_btn_send.setOnClickListener {

        if (!second_et_publish.text.isNullOrEmpty() && !second_et_topic.text.isNullOrEmpty()) {
            viewModel.publish(second_et_topic.text.toString(), second_et_publish.text.toString()) { status->
                if(status)
                    printToast(requireContext(), "La pubblicazione ha avuto successo")
                else
                    printToast(requireContext(), "La pubblicazione NON ha avuto successo")
            }
        }
    }

    viewModel.getLiveMessages().observe(viewLifecycleOwner, Observer { messages ->
        adapter.updateContent(messages)
        rv.scrollToPosition(0)
    })
}
```

### 3.3 Payload Encryption e QoS

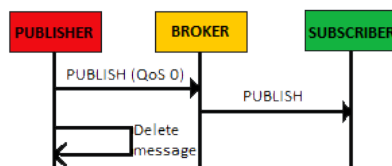
Prima di passare ai test effettuati sull'intero "sistema", verranno presentate le tecniche utilizzate per l'implementazione della crittografia del payload del messaggio e per la scelta della qualità del servizio.

Per quanto riguarda la **QoS** (*Qualità del Servizio*), il protocollo MQTT prevede tre diverse semantiche di consegna del messaggio:

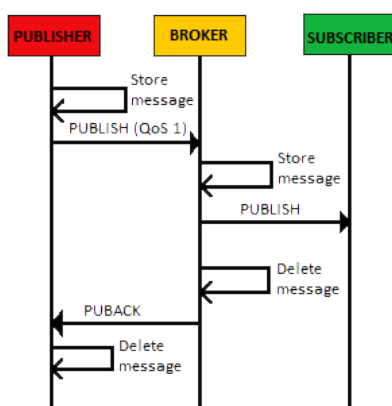
0. **at least once**: non c'è garanzia di consegna;
1. **at most once**: garantisce che un messaggio venga consegnato almeno una volta ai destinatari;
2. **exactly once**: garantisce che ogni messaggio venga ricevuto una sola volta dai destinatari previsti.

In base alla qualità scelta (**Figura 2**), il *client* ed il *broker* manterranno o meno il messaggio in memoria (per una possibile ritrasmissione in caso di problemi) e, inoltre, si verificherà o meno un maggiore scambio di messaggi tra le parti.

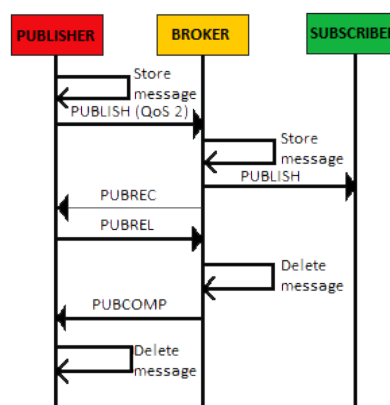
**QoS 0: Al massimo una volta**



**QoS 1: Almeno una volta**



**QoS 2: Esattamente una volta**



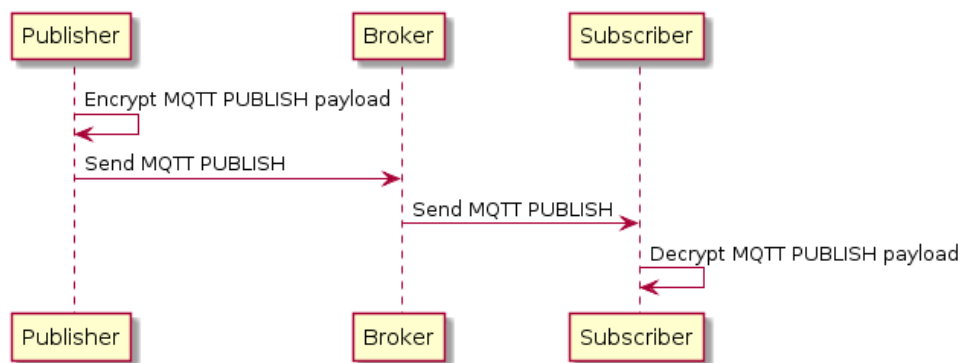
**Figura 2:** Livelli QoS (lato publisher)

Nell'applicazione, la scelta può essere specificata nei due metodi del *MainViewModel*, visti in precedenza, che si occupano della sottoscrizione e della pubblicazione:

```
fun subscribe(topic:String, qos:Int = "...", callback: (status: Boolean) -> Unit) {...}
fun publish(topic:String, msg:String, qos:Int = "...", callback: (status: Boolean) -> Unit) {...}
```

Inserendo al posto di “...” la QoS desiderata (0 | 1 | 2).

Per quanto riguarda la **crittografia del payload del messaggio**, *MQTT Payload* (HiveMQ, s.d.) è la crittografia dei dati specifici dell'applicazione a livello di applicazione (in genere, il payload del pacchetto MQTT PUBLISH o il payload CONNECT). Questo approccio consente la *crittografia end-to-end* (**Figura 3**) dei dati dell'applicazione anche in ambienti non attendibili. Mentre i metadati del messaggio, come l'argomento MQTT, rimangono intatti, il payload del messaggio viene crittografato. Questo tipo di crittografia non è definito nella specifica MQTT e va definito esplicitamente nell'applicazione.



**Figura 3:** Crittografia end-to-end

Nell'app Android realizzata, la crittografia del payload è stata implementata seguendo l'approccio a chiave simmetrica e sfruttando il package **javax.crypto**, che fornisce differenti classi e interfacce per la generazione di chiavi, per la crittazione, per la decrittazione ecc. (Oracle, s.d.).

La crittografia del payload è stata implementata nel *MainViewModel*, inserendo una funzione **initCryptography** alla fine della *connectClient*, vista in precedenza, che si occupa dell'inizializzazione dei componenti necessari (es. la generazione della chiave di cifratura).

```

private fun initCryptography(pwd: String) {
    val random = SecureRandom()
    val salt = ByteArray(256)
    random.nextBytes(salt)

    val pbKeySpec = PBEKeySpec(pwd.toCharArray(), salt, 1324, 256)
    val secretKeyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1")
    val keyBytes = secretKeyFactory.generateSecret(pbKeySpec).encoded
    keySpec = SecretKeySpec(keyBytes, "AES")
    val ivRandom = SecureRandom()

    val iv = ByteArray(16)
    ivRandom.nextBytes(iv)
    ivSpec = IvParameterSpec(iv)
}

```

La **cifratura** del payload del messaggio che si vuole pubblicare nel topic, viene realizzata nella funzione *publish* vista in precedenza, sostituendo la riga “*message.payload = msg.toByteArray()*” con quanto segue:

```

cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec)
message.payload = cipher.doFinal(msg.toByteArray())

```

Per quanto riguarda la **decifrazione** del messaggio, essa viene effettuata all’interno della funzione *messageArrived* (funzione del *callback listener* della funzione *connectClient* vista in precedenza), prima della chiamata della funzione che aggiorna la lista dei messaggi ricevuti (*addNewMessageToList*), inserendo quanto segue:

```

val temp = message?.payload
cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec)
val decrypted = cipher.doFinal(temp)
message?.payload = decrypted

```

## 4. Test

Una volta configurato correttamente il broker e realizzata l'applicazione Android, sono stati effettuate diversi test dell'intero "sistema", con lo scopo di analizzare le conseguenze dell'utilizzo della crittografia del payload del messaggio e della scelta di una determinata qualità nel servizio.


Prima di passare a tale verifica è stata effettuato un test sul funzionamento generale. Partendo dall'avvio del broker, è stato effettuato il test sulla connessione del client (app Android) al broker, sulla sottoscrizione del client ad un topic, sulla ricezione dei messaggi del topic sull'app (messaggi inviati da terminale, sfruttando la *mosquitto\_pub*) e sulla pubblicazione di messaggi sul topic dal client Android.

Con lo scopo di mostrare il funzionamento generale, vengono riportati di seguito alcuni screen che ne indicano i vari step da seguire.

1

```
Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto -c broker.conf -v
1653748063: mosquitto version 2.0.14 starting
1653748063: Config loaded from broker.conf.
1653748063: Opening ipv6 listen socket on port 1883.
1653748063: Opening ipv4 listen socket on port 1883.
1653748063: mosquitto version 2.0.14 running
```


2



3

```
Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto -c broker.conf -v
1652260008: mosquitto version 2.0.14 starting
1652260008: Config loaded from broker.conf.
1652260008: Opening ipv6 listen socket on port 1883.
1652260008: Opening ipv4 listen socket on port 1883.
1652260008: mosquitto version 2.0.14 running
1652260283: New connection from 192.168.178.32:65332 on port 1883.
1652260283: New client connected from 192.168.178.32:65332 as MobySystemsMQTT (p2, c1, k60, u'mqtt').
1652260283: No will message specified.
1652260283: Sending CONNACK to MobySystemsMQTT (0, 0)
```

4



5

```
1652260326: Received SUBSCRIBE from MobySystemsMQTT
1652260326:   temperatura/C (Qos 1)
1652260326: MobySystemsMQTT 1 temperatura/C
1652260386: Sending SUBACK to MobySystemsMQTT
1652260386: Received PINGREQ from MobySystemsMQTT
1652260386: Sending PINGRESP to MobySystemsMQTT
```

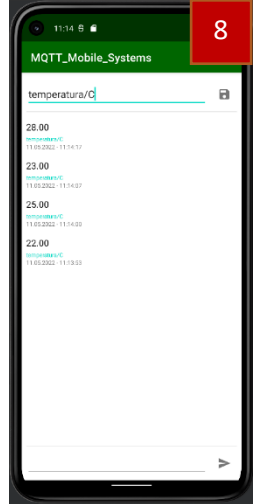
6

```
Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto_pub -h 192.168.178.32 -u mqtt_pub -P 123456789 -t temperatura/C -m "22.00"
Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto_pub -h 192.168.178.32 -u mqtt_pub -P 123456789 -t temperatura/C -m "25.00"
Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto_pub -h 192.168.178.32 -u mqtt_pub -P 123456789 -t temperatura/C -m "23.00"
C:\Users\Luigi\Desktop\Progetto\mosquitto_script>mosquitto_pub -h 192.168.178.32 -u mqtt_pub -P 123456789 -t temperatura/C -m "28.00"
```

7

```
1652260451: New connection from 192.168.178.32:65376 on port 1883.
1652260451: New client connected from 192.168.178.32:65376 as auto-E5D02FB2-6C9A-6702-AB89-0918B2A5E4AB (0, 0)
1652260451: No will message specified.
1652260451: Sending CONNACK to auto-E5D02FB2-6C9A-6702-AB89-0918B2A5E4AB (0, 0)
1652260451: Received PUBLISH from auto-E5D02FB2-6C9A-6702-AB89-0918B2A5E4AB (0, 0)
1652260451: Sending PUBLISH to MobySystemsMQTT (0, 0, 0, 0, 0, 0, 'temperatura/C')
1652260451: Received DISCONNECT from auto-E5D02FB2-6C9A-6702-AB89-0918B2A5E4AB
1652260451: Client auto-E5D02FB2-6C9A-6702-AB89-0918B2A5E4AB disconnected.
1652260443: New connection from 192.168.178.32:65380 on port 1883.
1652260443: New client connected from 192.168.178.32:65380 as auto-F421B476-0EAC-7167-A923-14ABC0AE284A (0, 0)
1652260443: No will message specified.
1652260443: Sending CONNACK to auto-F421B476-0EAC-7167-A923-14ABC0AE284A (0, 0)
1652260443: Received PUBLISH from auto-F421B476-0EAC-7167-A923-14ABC0AE284A (0, 0)
1652260443: Sending PUBLISH to MobySystemsMQTT (0, 0, 0, 0, 0, 0, 'temperatura/C')
1652260443: Received DISCONNECT from auto-F421B476-0EAC-7167-A923-14ABC0AE284A
1652260443: Client auto-F421B476-0EAC-7167-A923-14ABC0AE284A disconnected.
1652260446: Received PINGREQ from MobySystemsMQTT
1652260446: Sending PINGRESP to MobySystemsMQTT
1652260451: New connection from 192.168.178.32:65385 on port 1883.
1652260451: New client connected from 192.168.178.32:65385 as auto-3C958AB7-E29D-C572-338B-A6656B43518B (0, 0)
1652260451: No will message specified.
1652260451: Sending CONNACK to auto-3C958AB7-E29D-C572-338B-A6656B43518B (0, 0)
1652260451: Received PUBLISH from auto-3C958AB7-E29D-C572-338B-A6656B43518B (0, 0)
1652260451: Sending PUBLISH to MobySystemsMQTT (0, 0, 0, 0, 0, 0, 'temperatura/C')
1652260451: Received DISCONNECT from auto-3C958AB7-E29D-C572-338B-A6656B43518B
1652260451: Client auto-3C958AB7-E29D-C572-338B-A6656B43518B disconnected.
1652260461: New connection from 192.168.178.32:65390 on port 1883.
1652260461: New client connected from 192.168.178.32:65390 as auto-93988CD7-977A-530C-D5B8-E1D659271C79 (0, 0)
1652260461: No will message specified.
1652260461: Sending CONNACK to auto-93988CD7-977A-530C-D5B8-E1D659271C79 (0, 0)
1652260461: Received PUBLISH from auto-93988CD7-977A-530C-D5B8-E1D659271C79 (0, 0)
1652260461: Sending PUBLISH to MobySystemsMQTT (0, 0, 0, 0, 0, 0, 'temperatura/C')
1652260461: Received DISCONNECT from auto-93988CD7-977A-530C-D5B8-E1D659271C79
1652260461: Client auto-93988CD7-977A-530C-D5B8-E1D659271C79 disconnected.
1652260506: Received PINGREQ from MobySystemsMQTT
1652260506: Sending PINGRESP to MobySystemsMQTT
```

8





Una volta effettuato il test di funzionamento del sistema, sono stati effettuati dei test modificando la **Qualità del Servizio** e sfruttando o meno la **Payload Encryption**. Di seguito vengono riportati i log prodotti dal broker in base alla configurazione scelta:

#### SI ENCRYPTION

```
QoS 0
26/05/2022 16:09:05: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:09:05:   test (QoS 0)
26/05/2022 16:09:05: MobyleSystemsMQTT 0 test
26/05/2022 16:09:05: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:09:11: Received PUBLISH from MobyleSystemsMQTT (d0, q0, r0, m0, 'test', ... (16 bytes))
26/05/2022 16:09:11: Sending PUBLISH to MobyleSystemsMQTT (d0, q0, r0, m0, 'test', ... (16 bytes))

QoS 1
26/05/2022 16:08:00: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:08:00:   test (QoS 1)
26/05/2022 16:08:00: MobyleSystemsMQTT 1 test
26/05/2022 16:08:00: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:08:07: Received PUBLISH from MobyleSystemsMQTT (d0, q1, r0, m2, 'test', ... (16 bytes))
26/05/2022 16:08:07: Sending PUBLISH to MobyleSystemsMQTT (d0, q1, r0, m1, 'test', ... (16 bytes))
26/05/2022 16:08:07: Sending PUBACK to MobyleSystemsMQTT (m2, rc0)
26/05/2022 16:08:07: Received PUBACK from MobyleSystemsMQTT (Mid: 1, RC:0)

QoS 2
26/05/2022 16:04:43: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:04:43:   test (QoS 2)
26/05/2022 16:04:43: MobyleSystemsMQTT 2 test
26/05/2022 16:04:43: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:04:51: Received PUBLISH from MobyleSystemsMQTT (d0, q2, r0, m2, 'test', ... (16 bytes))
26/05/2022 16:04:51: Sending PUBREC to MobyleSystemsMQTT (m2, rc0)
26/05/2022 16:04:51: Received PUBREL from MobyleSystemsMQTT (Mid: 2)
26/05/2022 16:04:51: Sending PUBLISH to MobyleSystemsMQTT (d0, q2, r0, m1, 'test', ... (16 bytes))
26/05/2022 16:04:51: Sending PUBCOMP to MobyleSystemsMQTT (m2)
26/05/2022 16:04:51: Received PUBREC from MobyleSystemsMQTT (Mid: 1)
26/05/2022 16:04:51: Sending PUBREL to MobyleSystemsMQTT (m1)
26/05/2022 16:04:51: Received PUBCOMP from MobyleSystemsMQTT (Mid: 1, RC:0)
```

#### NO ENCRYPTION

```
QoS 0
26/05/2022 16:21:35: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:21:35:   test (QoS 0)
26/05/2022 16:21:35: MobyleSystemsMQTT 0 test
26/05/2022 16:21:35: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:21:41: Received PUBLISH from MobyleSystemsMQTT (d0, q0, r0, m0, 'test', ... (5 bytes))
26/05/2022 16:21:41: Sending PUBLISH to MobyleSystemsMQTT (d0, q0, r0, m0, 'test', ... (5 bytes))

QoS 1
26/05/2022 16:20:43: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:20:43:   test (QoS 1)
26/05/2022 16:20:43: MobyleSystemsMQTT 1 test
26/05/2022 16:20:43: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:20:51: Received PUBLISH from MobyleSystemsMQTT (d0, q1, r0, m2, 'test', ... (5 bytes))
26/05/2022 16:20:51: Sending PUBLISH to MobyleSystemsMQTT (d0, q1, r0, m1, 'test', ... (5 bytes))
26/05/2022 16:20:51: Sending PUBACK to MobyleSystemsMQTT (m2, rc0)
26/05/2022 16:20:51: Received PUBACK from MobyleSystemsMQTT (Mid: 1, RC:0)

QoS 2
26/05/2022 16:17:46: Received SUBSCRIBE from MobyleSystemsMQTT
26/05/2022 16:17:46:   test (QoS 2)
26/05/2022 16:17:46: MobyleSystemsMQTT 2 test
26/05/2022 16:17:46: Sending SUBACK to MobyleSystemsMQTT
26/05/2022 16:17:54: Received PUBLISH from MobyleSystemsMQTT (d0, q2, r0, m2, 'test', ... (5 bytes))
26/05/2022 16:17:54: Sending PUBREC to MobyleSystemsMQTT (m2, rc0)
26/05/2022 16:17:54: Received PUBREL from MobyleSystemsMQTT (Mid: 2)
26/05/2022 16:17:54: Sending PUBLISH to MobyleSystemsMQTT (d0, q2, r0, m1, 'test', ... (5 bytes))
26/05/2022 16:17:54: Sending PUBCOMP to MobyleSystemsMQTT (m2)
26/05/2022 16:17:54: Received PUBREC from MobyleSystemsMQTT (Mid: 1)
26/05/2022 16:17:54: Sending PUBREL to MobyleSystemsMQTT (m1)
26/05/2022 16:17:54: Received PUBCOMP from MobyleSystemsMQTT (Mid: 1, RC:0)
```



Come si può notare dai log, in base alla QoS utilizzata si avrà un maggior o minor scambio di messaggi tra il client ed il broker (messaggi di controllo). La scelta della qualità del servizio è quindi da considerare con attenzione in base alle caratteristiche del dispositivo che eseguirà il broker, in quanto più si sale di qualità, più si rischia di sovraccaricare il broker e, di conseguenza, di aumentare il tempo di latenza sulla consegna dei messaggi.

Per quanto riguarda l'uso della *crittografia del payload*, si può notare che il suo utilizzo comporta un notevole aumento delle dimensioni del payload (più del triplo rispetto ai messaggi senza crittografia → valori in rosso nei log) e provoca uno sforzo computazionale maggiore lato client (riscontro ottenuto analizzando l'esecuzione dell'emulatore Android tramite il servizio di Windows "procexp"). La scelta di utilizzare o meno la crittografia del payload del messaggio dipende, quindi, dalle caratteristiche del dispositivo (CPU, memoria ecc.) che invia e riceve il messaggio (nel caso della crittografia end-to-end).

## 5. Conclusioni

Il protocollo MQTT è utilizzato in particolar modo in ambito IoT, sfruttando il broker per lo smistamento di messaggi tra varie tipologie di sensori, che raccolgono informazioni e le inviano al topic dedicato, ed i client, che si occupano di presentare all'utente le informazioni di suo interesse tramite la sottoscrizione ai relativi topic (Esempio: un sensore di temperatura invia, ad intervalli di tempo prestabiliti, un messaggio ad un topic "temperatura/C"; il broker riceve il messaggio e lo inoltra ai client che hanno effettuato la sottoscrizione al topic; il client riceve i messaggi e mostra a video i dati ricevuti).

Nella parte iniziale del progetto è stato realizzato uno scenario simile a quanto detto, con l'unica differenza che, in questo caso, la sensoristica è stata simulata con l'invio di messaggi al broker da terminale. Una volta realizzato il client che si occupa della visualizzazione grafica dei dati ricevuti dal broker (app Android) e configurato adeguatamente il broker mosquitto, sono stati effettuati dei test con lo scopo di analizzare l'effetto della **crittografia del payload del messaggio** e della **qualità del servizio** sull'intero sistema.

L'utilizzo della *Payload Encryption* con crittografia end-to-end a chiave simmetrica, implementata in questo progetto, comporta un maggior sforzo computazionale lato client, di conseguenza bisogna decidere se implementarla o meno in base alle risorse (CPU, ram ecc.) dei dispositivi che eseguiranno il client.

Per quanto riguarda la QoS (Qualità del Servizio), all'aumentare del suo livello ( $0 \rightarrow 1 \rightarrow 2$ ) si verifica un maggior scambio di messaggi sia tra publisher (es. sensore) e broker, sia tra broker e subscriber (es. client); maggiore nel primo caso. Tutto ciò comporta un maggior sforzo del broker, soprattutto in situazioni in cui i dispositivi publisher e subscriber sono numerosi, in quanto più si sceglie un livello alto di qualità, più il broker consuma risorse per mantenere in memoria i messaggi da consegnare e per inviare i vari messaggi di controllo. Di conseguenza, la scelta va effettuata con attenzione, analizzando le risorse disponibili nei dispositivi che implementeranno i vari elementi del sistema. Da notare che, pur essendo il broker il componente maggiormente influenzato dalla qualità del servizio scelta, anche i sensori ed i client sono soggetti ad un maggior consumo di risorse se si sceglie un livello di qualità alto.

Per concludere, essendo MQTT un protocollo leggero, nato per implementare sistemi IoT costituiti da dispositivi e sensori con capacità computazionale ridotta, l'usabilità e la stabilità generale del sistema devono risultare ottimali, di conseguenza bisogna scegliere il livello di QoS e l'utilizzo o meno di un sistema di crittografia del payload (da applicare soprattutto in base alla tipologia di dati trasmessi) in modo adeguato. A mio parere, meglio impiegare più risorse sull'implementazione di un algoritmo di crittografia del payload che protegga i dati trasmessi rispetto ad una qualità del servizio elevata.

## 6. Bibliografia

- Eclipse. (s.d.). *Eclipse Paho Java Client*. Tratto da <https://www.eclipse.org/paho/index.php?page=clients/java/index.php>
- EclipseMosquitto™. (s.d.). *Eclipse Mosquitto*. Tratto da <https://mosquitto.org>
- HiveMQ. (s.d.). *Payload Encryption - MQTT Security Fundamentals*. Tratto da <https://www.hivemq.com/blog/mqtt-security-fundamentals-payload-encryption/>
- Oracle. (s.d.). *Package javax.crypto*. Tratto da <https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html>