Application d'écriture en BDD

Parc – MQTT ? MySQL Bridge

Service Python qui fait le pont entre un broker MQTT Mosquitto et une base MySQL pour le système de monitoring de serres **SIGACS / Parc**.

Architecture

```
[ESP32 / M5Stack / autres contrôleurs]
        ?  mTLS port 8883
        ?
 ????????????????????         ??????????????????????????????
 ?   Mosquitto 2.x ? ?????????   bridge.py (Python 3.12) ? ???  MySQL (externe)
 ?   (broker)      ?         ?   subscribe + persist     ?
 ????????????????????         ??????????????????????????????
   Conteneur Docker                 Conteneur Docker
        réseau partagé parc-net
```

Les deux conteneurs partagent le réseau Docker `parc-net`. Le serveur MySQL est externe (conteneurisé sur le web_server).

Stack

Composant	Technologie
Broker MQTT	Eclipse Mosquitto 2.x (Docker)
Bridge	Python 3.12 (Docker)
Base de données	MySQL
Bibliothèque MQTT	paho-mqtt 2.1.0
Bibliothèque DB	PyMySQL 1.1.1
TLS	mTLS — CA auto-signé, certificats par client
Conteneurisation	Docker Compose

Topics MQTT

Mesures capteurs

Pattern	Description
`serre/<numero>/bac/<numero>`	Mesures du bac `<numero>` dans la serre `<numero>`

`<numero>` correspond à la colonne `numero` dans les tables `serre` et `bac` (pas la clé primaire).

Payload JSON :

```
{"humiditeAmbiante": 65.3, "temperatureAmbiante": 22.1, "humiditeSol": 45.0}
```

Plusieurs capteurs peuvent coexister dans un même payload.

Capteurs supportés :

Clé JSON	`id_capteur`	Unité	Min	Max
`humiditeAmbiante`	1	%	0	100
`humiditeSol`	2	%	0	100
`temperatureAmbiante`	3	°C	-40	80

Contrôleurs

Pattern	Direction	Déclencheur
`controleur/<MAC>`	Contrôleur → Broker	À chaque connexion du contrôleur
`controleur/<MAC>/disconnect`	Broker → Bridge	Déconnexion du contrôleur (LWT Mosquitto)

`<MAC>` est l'adresse MAC Wi-Fi du contrôleur au format `XX:XX:XX:XX:XX:XX`. Elle sert d'identifiant unique (`id_controleur`) en base de données.

Payload de connexion (`controleur/<MAC>`) :

```
{"ip": "192.168.42.85", "status": true}
```

Payload de déconnexion (`controleur/<MAC>/disconnect`) :

```
(payload vide)
```

Le bridge passe automatiquement `status = false` en base à la réception de ce message.

Gestion des erreurs

Toutes les erreurs sont persistées dans la table `error` :

`type_erreur`	**Déclencheur**
`BAC_NOT_FOUND`	Numéro de serre ou bac introuvable en base
`UNKNOWN_SENSOR`	Clé JSON absente de la liste des capteurs connus
`INVALID_PAYLOAD`	Erreur de décodage JSON, payload non-objet, ou champ manquant
`INVALID_VALUE`	Valeur non numérique pour un capteur, ou type de champ invalide
`VALUE_OUT_OF_RANGE`	Valeur clampée au min/max du capteur
`INVALID_ENCODING`	Payload non UTF-8
`MQTT_DISCONNECT`	Déconnexion inattendue du broker
`DB_ERROR`	Erreur transitoire de requête base de données
`DB_INSERT_ERROR`	Échec d'insertion en base
`UNEXPECTED_ERROR`	Exception non gérée (attrape-tout)

Quand une valeur est hors limites : la valeur clampée (min ou max) est écrite dans `mesure` **et** une erreur est insérée.

TLS / mTLS

Architecture des certificats

```
ESP32 / M5Stack / autres clients
      ?  mTLS
      ?
 Mosquitto :8883  ??? server.crt (CN=mosquitto, SAN=DNS:mosquitto, IP:x.x.x.x)
      ?
      ?  mTLS
      ?
 bridge.py ????? présente client.crt (CN=mqtt-bridge)
```

Fichiers de certificats

Fichier	Utilisé par	Monté à
`server-certs/ca.crt`	Mosquitto + bridge + clients ESP32	`/mosquitto/certs/ca.crt` et `/certs/ca.crt`
`server-certs/server.crt`	Mosquitto	`/mosquitto/certs/server.crt`
`server-certs/server.key`	Mosquitto	`/mosquitto/certs/server.key`
`client-certs/client.crt`	Bridge	`/certs/client.crt`
`client-certs/client.key`	Bridge	`/certs/client.key`

Exigences critiques

- Le certificat serveur **doit** avoir `CN=mosquitto` **et** `subjectAltName=DNS:mosquitto,IP:<IP_LAN>` (Python SSL ignore le CN et ne vérifie que le SAN — les ESP32 connectent via IP LAN)
- Chaque nouveau client (ESP32, M5Stack, service…) obtient son propre certificat signé par le même CA
- Aucun changement de configuration Mosquitto nécessaire pour de nouveaux clients
- Taille des clés : 2048 bits (compatible ESP32 / M5Stack)
- Format des clés : PKCS#1 (`-----BEGIN RSA PRIVATE KEY-----`) → Sur OpenSSL 3.x, utiliser `openssl genrsa -traditional` pour forcer ce format

`v3.ext` (requis pour la signature du certificat serveur)

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names

[alt_names]
DNS.1 = mosquitto
IP.1  = 192.168.x.x
```

Permissions

```
sudo chown 1883:1883 mosquitto/server-certs/server.key && chmod 600 mosquitto/server-
certs/server.key
sudo chown 1001:1001 mosquitto/client-certs/client.key  && chmod 600 mosquitto/client-
certs/client.key
```

Démarrage rapide

1. Prérequis

- Docker ≥ 24 avec le plugin Compose
- Serveur MySQL / MariaDB accessible avec le schéma `parc` appliqué
- Certificats générés (voir section TLS)

2. Configuration

```
cp .env.example .env
# Renseigner DB_HOST, DB_USER, DB_PASSWORD dans .env
```

3. Démarrage

```
docker compose up -d --build
```

4. Vérification

```
# Suivre les logs du bridge
docker compose logs -f bridge

# Tester une mesure capteur
docker run --rm --network parc-net eclipse-mosquitto:2 \
  mosquitto_pub -h mosquitto -p 8883 \
  -t "serre/1/bac/1" \
  -m '{"humiditeAmbiante": 65, "temperatureAmbiante": 22}'

# Tester une connexion contrôleur
docker run --rm --network parc-net eclipse-mosquitto:2 \
  mosquitto_pub -h mosquitto -p 8883 \
  -t "controleur/24:D7:EB:38:DC:38" \
  -m '{"ip": "192.168.42.85", "status": true}'
```

Variables d'environnement

Variable	Défaut	Description
`DB_HOST`	*(requis)*	Hôte MySQL
`DB_PORT`	`3306`	Port MySQL
`DB_NAME`	`parc`	Nom de la base

Variable	Défaut	Description
`DB_USER`	*(requis)*	Utilisateur MySQL
`DB_PASSWORD`	*(requis)*	Mot de passe MySQL
`MQTT_HOST`	`mosquitto`	Nom d'hôte du broker
`MQTT_PORT`	`8883`	Port du broker (TLS)
`MQTT_KEEPALIVE`	`60`	Keepalive MQTT (s)
`MQTT_USER`	*(vide)*	Utilisateur broker (optionnel)
`MQTT_PASSWORD`	*(vide)*	Mot de passe broker (optionnel)
`DB_RETRY_DELAY`	`5`	Secondes entre tentatives DB
`MQTT_RETRY_DELAY`	`5`	Secondes entre tentatives MQTT
`LOG_LEVEL`	`INFO`	`DEBUG` / `INFO` / `WARNING` / `ERROR`

Décisions de conception — bridge.py

- **Reconnexion DB** : `ensure_db()` appelé avant chaque requête, reconnecte transparemment en cas de coupure
- **Reconnexion MQTT** : `reconnect_delay_set(min=5, max=60)` + `loop_forever(retry_first_connection=True)`
- **Cache mémoire** : `_bac_cache` et `_capteur_cache` évitent les allers-retours DB sur chaque message ; invalidés à chaque reconnexion DB
- **Clampge de valeurs** : valeurs hors limites clampées au min/max, mesure insérée avec valeur clampée, erreur également insérée
- **Attrape-tout** : chaque callback `on_message` est enveloppé dans try/except pour qu'un seul message mal formé ne fasse jamais planter le bridge
- **Routage de topics** : `on_message()` route vers `process_controleur_message()`, `set_controleur_offline()` ou `process_message()` selon le topic reçu
- **LWT contrôleur** : le bridge écoute `controleur/+/disconnect` — Mosquitto publie ce message automatiquement si un client se déconnecte brutalement

---

Génération d'un nouveau certificat client

Pour chaque nouveau client (ESP32, M5Stack, service...) :

```
cd mosquitto
```

```
# Sur OpenSSL 3.x, utiliser -traditional pour forcer le format PKCS#1
openssl genrsa -traditional -out newclient.key 2048

openssl req -new -out newclient.csr -key newclient.key
# Common Name : nom descriptif du client (ex: m5-serre-1)

openssl x509 -req -in newclient.csr \
  -CA server-certs/ca.crt \
  -CAkey server-certs/ca.key \
  -CAcreateserial -out newclient.crt -days 720

mkdir client-certs-newclient
mv newclient.* client-certs-newclient/
```

Vérifier que le cert et la clé correspondent :

```
openssl x509 -noout -modulus -in newclient.crt | openssl md5
openssl rsa  -noout -modulus -in newclient.key | openssl md5
# Les deux hashes doivent être identiques
```

Commandes de debug fréquentes

```
# Suivre tous les logs
docker compose logs -f

# Logs d'un seul service
docker compose logs -f bridge
docker compose logs mosquitto

# Vérifier le SAN du certificat serveur
openssl x509 -in mosquitto/server-certs/server.crt -noout -ext subjectAltName

# Vérifier la chaîne de confiance du certificat serveur
openssl verify -CAfile mosquitto/server-certs/ca.crt mosquitto/server-certs/server.crt

# Vérifier la connexion TLS depuis le serveur
openssl s_client -connect 127.0.0.1:8883 -CAfile mosquitto/server-certs/ca.crt

# Vérifier correspondance cert / clé client
openssl x509 -noout -modulus -in mosquitto/client-certs/client.crt | openssl md5
openssl rsa  -noout -modulus -in mosquitto/client-certs/client.key | openssl md5

# Vérifier l'exposition des ports Docker
docker ps | grep mosquitto
```

Structure des fichiers

```
mqtt-bridge/
??? bridge.py                          # Programme Python principal
```

```
??? Dockerfile                          # Image du bridge
??? docker-compose.yml                  # Définition du stack complet
??? requirements.txt                    # Dépendances Python
??? .env.example                        # Modèle de variables d'environnement
??? mosquitto/
    ??? config/
    ?   ??? mosquitto.conf              # Configuration du broker
    ??? server-certs/                   # CA + certificats serveur
    ?   ??? ca.crt
    ?   ??? ca.key
    ?   ??? server.crt
    ?   ??? server.key
    ??? client-certs/                   # Certificat client du bridge
    ?   ??? client.crt
    ?   ??? client.key
    ??? server-certs.sh                 # Script de génération CA + serveur
    ??? clients-certs.sh                # Script de génération d'un cert client
    ??? v3.ext                          # Extension SAN pour la signature serveur
```

Intégration ESP32 / M5Stack

- Se connecter à l'**IP LAN du serveur hôte** (pas au nom de service Docker `mosquitto`)
- Le port 8883 doit être exposé sur l'hôte dans `docker-compose.yml`
- Chaque appareil obtient son propre certificat client (même CA, nouveau key+CSR+crt, CN différent)
- Utiliser des clés 2048 bits (les clés 4096 bits peuvent dépasser la RAM de l'ESP32)
- Format clé privée : PKCS#1 (`-----BEGIN RSA PRIVATE KEY-----`) — générer avec `-traditional` sur OpenSSL 3.x
- Embarquer les certificats avec la syntaxe `R"EOF(...)EOF"` et `PROGMEM` pour stocker en flash
- Configurer le **Last Will Testament** sur `controleur/<MAC>/disconnect` avant `connect()`
- Publier les infos contrôleur sur `controleur/<MAC>` immédiatement après chaque connexion réussie

---

Revision #3
Created 2026-05-22 06:51:18 UTC by Clément
Updated 2026-06-03 08:18:19 UTC by Clément