Bouchon de test

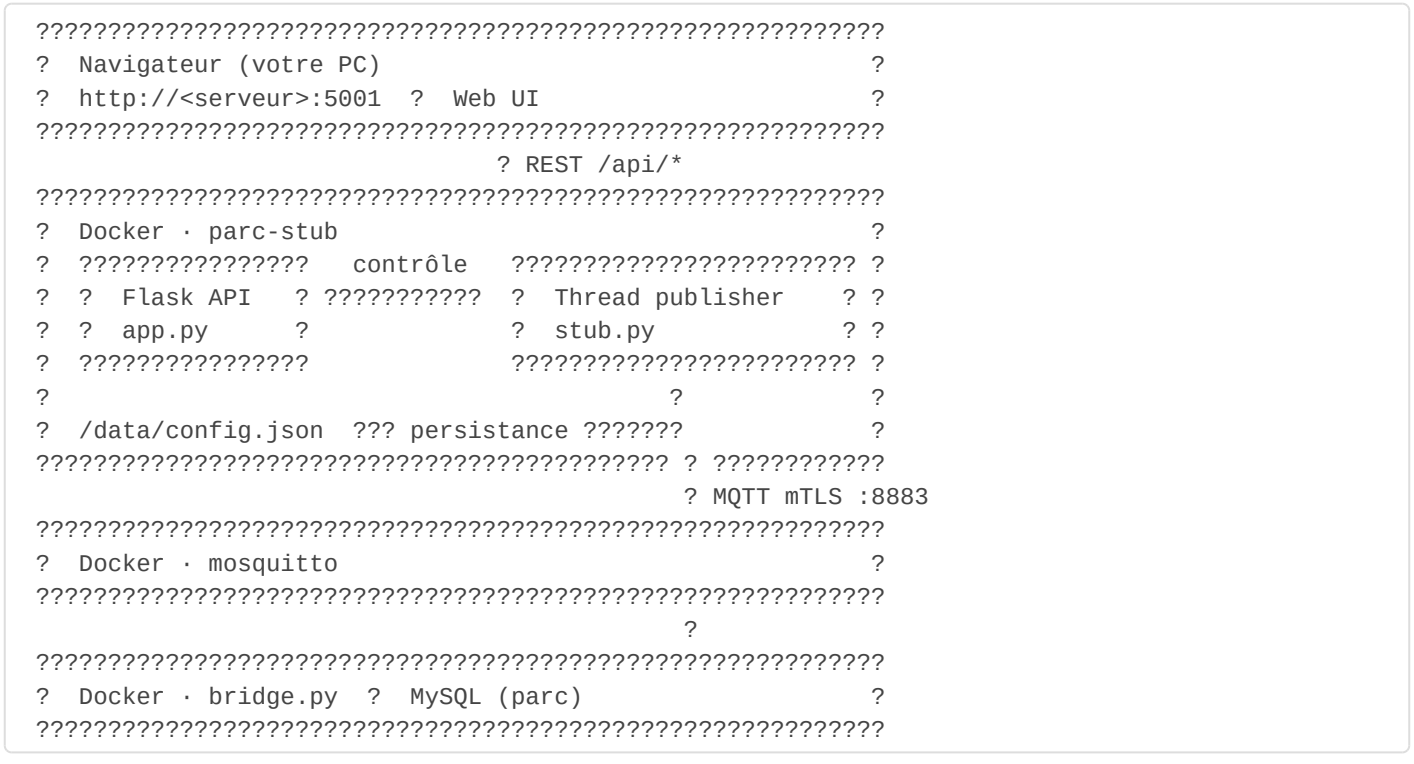
Documentation du bouchon de test mis en place afin d'envoyer de manière fiable et paramétrable.

Fonctionnement

Simulateur MQTT — Documentation

> **parc-stub** est un simulateur de capteurs IoT pour le projet SIGACS.
> Il publie des données de serre sur le broker MQTT afin de tester le pipeline complet sans matériel réel.

Aperçu de l'architecture



Données simulées — capteurs

Le stub publie trois capteurs en **onde sinusoïdale** (cycle de 2 minutes).
Chaque bac possède un déphasage aléatoire — les courbes diffèrent entre bacs.

Capteur	Clé JSON	Centre	Amplitude	Unité
Humidité ambiante	`humiditeAmbiante`	60	±20	%
Humidité du sol	`humiditeSol`	55	±15	%
Température ambiante	`temperatureAmbiante`	22	±5	°C

Les données capteurs ne sont publiées **que si le contrôleur de la serre est connecté**.

Topic et payload capteurs

```
serre/<numero_serre>/bac/<numero_bac>
```

```
{"humiditeAmbiante": 67.4, "humiditeSol": 52.1, "temperatureAmbiante": 24.3}
```

Données simulées — contrôleurs (M5Stack)

Chaque serre est associée à un contrôleur identifié par son adresse MAC.

Événement	Topic	Sens
Connexion	`controleur/<MAC>`	Stub → Broker
Déconnexion (LWT)	`controleur/<MAC>/disconnect`	Stub → Broker (retain)

Payload de connexion

```
{
  "topic":  "controleur/24:D7:EB:38:DC:38",
  "ip":     "192.168.1.101",
  "status": true
}
```

Payload de déconnexion (LWT)

Payload vide `""`, topic retain `controleur/<MAC>/disconnect`.

Injection d'erreurs

Le taux d'erreur capteurs (0–100 %) déclenche aléatoirement l'un de ces 4 types :

Type	Payload envoyé	Erreur attendue côté bridge
`out_of_range`	`{"humiditeAmbiante": 150.0}`	`VALUE_OUT_OF_RANGE`
`unknown_sensor`	`{"co2Level": 400.0}`	`UNKNOWN_SENSOR`
`malformed`	`\xff\xfe …` (non-UTF-8)	`INVALID_ENCODING`
`invalid_value`	`{"temperatureAmbiante": "hot"}`	`INVALID_VALUE`

Web UI

Accessible sur `http://<IP_serveur>:5001`

Section	Rôle
Contrôle global	Démarrer / arrêter le stub, régler l'intervalle de publication et le taux d'erreurs capteurs
Serres & Contrôleurs	Ajouter / supprimer des serres ; configurer MAC, IP, taux de déconnexion aléatoire ; connecter / déconnecter manuellement
Journal	Affichage en direct — vert = OK, orange = erreur capteur, bleu = connexion, rouge = déconnexion

Couleurs du journal

Couleur	Signification
◫ Vert	Message capteur nominal
◫ Orange	Message capteur invalide
◫ Bleu	Connexion contrôleur
◫ Rouge	Déconnexion contrôleur (LWT)

> Les numéros de serre et bac **doivent correspondre** aux colonnes `numero` dans la base de données, sinon le bridge logge `BAC_NOT_FOUND`.

Persistance de la configuration

La configuration (serres, bacs, MAC, IP, intervalle, taux d'erreur) est sauvegardée automatiquement dans `/data/config.json` à chaque modification.

- Survit aux **refreshs** de l'interface
- Survit aux **redémarrages** du conteneur Docker
- L'état `connected` est toujours remis à `false` au redémarrage

Le fichier est stocké dans le volume Docker nommé `parc-stub-data`.

API REST

Méthode	Route	Description
`GET`	`/api/state`	État complet (running, config, serres, log)
`POST`	`/api/start`	Démarre la publication
`POST`	`/api/stop`	Arrête la publication
`POST`	`/api/config`	Modifie `interval` (s) et `bad_rate` (0–1)
`POST`	`/api/serres`	Remplace la liste complète des serres/bacs
`POST`	`/api/serre/<n>/connect`	Connecte manuellement le contrôleur de la serre n°
`POST`	`/api/serre/<n>/disconnect`	Déconnecte manuellement le contrôleur de la serre n°

Exemples

```
# Changer l'intervalle et le taux d'erreur
curl -X POST http://<serveur>:5001/api/config \
  -H "Content-Type: application/json" \
  -d '{"interval": 5, "bad_rate": 0.1}'

# Connecter manuellement la serre 1
curl -X POST http://<serveur>:5001/api/serre/1/connect

# Déconnecter manuellement la serre 2
curl -X POST http://<serveur>:5001/api/serre/2/disconnect

# Remplacer la configuration des serres
curl -X POST http://<serveur>:5001/api/serres \
```

```
  -H "Content-Type: application/json" \
  -d
'[{"numero":1,"mac":"24:D7:EB:38:DC:38","ip":"192.168.1.101","disconnect_rate":0.05,"bacs":[{"
numero":1},{"numero":2}]}]'
```

Certificats mTLS

Le stub utilise un certificat client dédié généré par `setup.sh` :

```
mosquitto/stub-certs/
??? stub.crt   ? certificat client (CN=parc-stub, 720 jours)
??? stub.key   ? clé privée RSA 2048 bits (chmod 600)
```

Signé par la même CA que le reste du projet — aucun changement de config Mosquitto nécessaire.

> **Erreur fréquente :** `[Errno 21] Is a directory` — un des chemins de certificat dans le volume Docker pointe vers un dossier. Vérifier avec `docker inspect parc-stub | grep -A 20 Mounts`.

Commandes utiles

```
# Logs en direct
docker logs -f parc-stub

# Redémarrer sans rebuild
docker compose -f parc-stub/docker-compose.stub.yml restart

# Reconstruire après modification des sources
docker compose -f parc-stub/docker-compose.stub.yml up -d --build

# Arrêter et supprimer le conteneur
docker compose -f parc-stub/docker-compose.stub.yml down

# Inspecter les volumes montés
docker inspect parc-stub | grep -A 20 Mounts

# Lire la config persistée
docker exec parc-stub cat /data/config.json
```

Fichiers du projet

```
parc/
??? setup.sh                          ? configuration initiale (certificats + docker-compose)
??? mosquitto/
?   ??? stub-certs/
?       ??? stub.crt                  ? certificat client du stub
?       ??? stub.key                  ? clé privée (non committée)
??? parc-stub/
    ??? Dockerfile
    ??? docker-compose.stub.yml       ? service + volume parc-stub-data
    ??? requirements.txt
    ??? stub.py                       ? logique de publication + persistance
    ??? app.py                        ? API Flask
    ??? templates/
        ??? index.html                ? interface Web
```

Projet SIGACS — BTS CIEL · Saint Joseph LaSalle · Lorient
HERVOUET Clément · BANCQUART Alan · LE GOUALEC Titouan

Programme - API Bouchon

```
"""
parc-stub · Flask control API

GET  /api/state                    ? full state snapshot
POST /api/start                    ? start publishing
POST /api/stop                     ? stop publishing
POST /api/config                   ? update interval & bad_rate
POST /api/serres                   ? replace serres/bacs layout
POST /api/serre/<int>/connect      ? manual connect controleur
POST /api/serre/<int>/disconnect   ? manual disconnect controleur
GET  /                             ? Web UI
"""

import logging
from flask import Flask, jsonify, render_template, request
import stub

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)s %(message)s")
app = Flask(__name__)


# ?? State ????????????????????????????????????????????????????????????????
@app.get("/api/state")
def api_state():
    with stub.state_lock:
        return jsonify(dict(stub.state))


# ?? Start / Stop ?????????????????????????????????????????????????????????
@app.post("/api/start")
def api_start():
    stub.start()
    return jsonify({"ok": True})

@app.post("/api/stop")
def api_stop():
    stub.stop()
    return jsonify({"ok": True})


# ?? Global config ????????????????????????????????????????????????????????
@app.post("/api/config")
def api_config():
    data = request.get_json(force=True)
    with stub.state_lock:
        if "interval" in data:
            v = float(data["interval"])
            if v < 1:
                return jsonify({"error": "interval min 1s"}), 400
            stub.state["interval"] = v
        if "bad_rate" in data:
            v = float(data["bad_rate"])
            if not 0.0 <= v <= 1.0:
```

```
                return jsonify({"error": "bad_rate 0–1"}), 400
            stub.state["bad_rate"] = v
    stub.save_config()
    return jsonify({"ok": True})


# ?? Serres layout ????????????????????????????????????????????????????????
@app.post("/api/serres")
def api_serres():
    """
    Body: [
      {
        "numero": 1,
        "mac": "24:D7:EB:38:DC:38",
        "ip": "192.168.1.101",
        "disconnect_rate": 0.05,
        "bacs": [{"numero": 1}, {"numero": 2}]
      }, …
    ]
    """
    data = request.get_json(force=True)
    if not isinstance(data, list):
        return jsonify({"error": "liste attendue"}), 400
    for s in data:
        for field in ("numero", "mac", "ip", "bacs"):
            if field not in s:
                return jsonify({"error": f"champ manquant : {field}"}), 400
        for b in s["bacs"]:
            if "numero" not in b:
                return jsonify({"error": "chaque bac doit avoir numero"}), 400

    with stub.state_lock:
        # preserve connected state for existing serres
        existing = {s["numero"]: s.get("connected", False) for s in stub.state["serres"]}
        for s in data:
            s["connected"] = existing.get(s["numero"], False)
            s.setdefault("disconnect_rate", 0.0)
        stub.state["serres"] = data
    stub.save_config()
    return jsonify({"ok": True})


# ?? Controleur connect / disconnect ??????????????????????????????????????
@app.post("/api/serre/<int:numero>/connect")
def api_connect(numero: int):
    stub.connect_serre(numero)
    return jsonify({"ok": True})

@app.post("/api/serre/<int:numero>/disconnect")
def api_disconnect(numero: int):
    stub.disconnect_serre(numero)
    return jsonify({"ok": True})


# ?? UI ???????????????????????????????????????????????????????????????????
@app.get("/")
def index():
    return render_template("index.html")
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=False)
```

Programme - Envois MQTT

```
"""
parc-stub · MQTT sensor simulator
Publishes sine-wave sensor data + controleur connect/disconnect over mTLS.
"""

import json
import logging
import math
import os
import random
import ssl
import threading
import time

import paho.mqtt.client as mqtt

logger = logging.getLogger("stub")

CONFIG_PATH = os.getenv("CONFIG_PATH", "/data/config.json")

# ?? Persistence ??????????????????????????????????????????????????????????????
def _save(s: dict):
    """Save interval, bad_rate and serres layout (not log, not running)."""
    try:
        os.makedirs(os.path.dirname(CONFIG_PATH), exist_ok=True)
        payload = {
            "interval":  s["interval"],
            "bad_rate":  s["bad_rate"],
            "serres": [
                {k: v for k, v in serre.items() if k != "connected"}
                for serre in s["serres"]
            ],
        }
        tmp = CONFIG_PATH + ".tmp"
        with open(tmp, "w") as f:
            json.dump(payload, f, indent=2)
        os.replace(tmp, CONFIG_PATH)
    except Exception as e:
        logger.warning(f"Config save failed: {e}")

def _load() -> dict | None:
    try:
        with open(CONFIG_PATH) as f:
            return json.load(f)
    except FileNotFoundError:
        return None
    except Exception as e:
        logger.warning(f"Config load failed: {e}")
        return None


SENSORS = {
    "humiditeAmbiante":    {"min": 0,   "max": 100, "center": 60, "amp": 20},
```

```
    "humiditeSol":         {"min": 0,   "max": 100, "center": 55, "amp": 15},
    "temperatureAmbiante": {"min": -40, "max": 80,  "center": 22, "amp": 5},
}

def _default_serre(numero, mac=None, ip=None):
    return {
        "numero": numero,
        "mac": mac or _random_mac(),
        "ip": ip or f"192.168.1.{100 + numero}",
        "disconnect_rate": 0.0,   # probability per cycle of random disconnect
        "connected": False,       # current simulated connection state
        "bacs": [{"numero": 1}],
    }

def _random_mac():
    parts = [random.randint(0x00, 0xFF) for _ in range(6)]
    return ":".join(f"{p:02X}" for p in parts)

def _build_state() -> dict:
    saved = _load()
    if saved:
        serres = saved.get("serres", [])
        for s in serres:
            s["connected"] = False
            s.setdefault("disconnect_rate", 0.0)
        return {
            "running":  False,
            "interval": saved.get("interval", 10),
            "bad_rate": saved.get("bad_rate", 0.05),
            "serres":   serres,
            "log":      [],
        }
    return {
        "running": False,
        "interval": 10,
        "bad_rate": 0.05,
        "serres": [
            _default_serre(1, "24:D7:EB:38:DC:38", "192.168.1.101"),
            _default_serre(2, "24:D7:EB:AA:BB:CC", "192.168.1.102"),
        ],
        "log": [],
    }

state = _build_state()
# Per-serre manual connect/disconnect commands queued by API
# { serre_numero: "connect" | "disconnect" }
_cmd_queue: dict = {}

state_lock = threading.Lock()
_stop_event = threading.Event()
_thread = None


# ?? Logging ????????????????????????????????????????????????????????????????
def _log(msg: str, kind: str = "inf"):
    logger.info(msg)
    with state_lock:
        state["log"].append({"t": time.strftime("%H:%M:%S"), "msg": msg, "kind": kind})
        if len(state["log"]) > 150:
```

```
            state["log"].pop(0)


# ?? Controleur MQTT helpers ??????????????????????????????????????????????
def _publish_connect(client: mqtt.Client, serre: dict):
    mac = serre["mac"]
    topic = f"controleur/{mac}"
    payload = json.dumps({
        "topic":  topic,
        "ip":     serre["ip"],
        "status": True,
    })
    client.publish(topic, payload, retain=False)
    _log(f"[CONN] controleur/{mac}  ip={serre['ip']}", "conn")


def _publish_disconnect(client: mqtt.Client, serre: dict):
    mac = serre["mac"]
    lwt_topic = f"controleur/{mac}/disconnect"
    client.publish(lwt_topic, "", retain=True)
    _log(f"[DISC] controleur/{mac}/disconnect (LWT)", "disc")


# ?? Sine / bad payload ????????????????????????????????????????????????????
def _sine_value(sensor: str, t: float, phase: float = 0.0) -> float:
    cfg = SENSORS[sensor]
    val = cfg["center"] + cfg["amp"] * math.sin(2 * math.pi * t / 120 + phase)
    return round(val, 2)

def _bad_payload(_t):
    kind = random.choice(["out_of_range", "unknown_sensor", "malformed", "invalid_value"])
    if kind == "out_of_range":
        return json.dumps({"humiditeAmbiante": 150.0})
    if kind == "unknown_sensor":
        return json.dumps({"co2Level": 400.0})
    if kind == "malformed":
        return b"\xff\xfe not utf-8"
    return json.dumps({"temperatureAmbiante": "hot"})


# ?? MQTT client ???????????????????????????????????????????????????????????
def _build_client() -> mqtt.Client:
    host     = os.getenv("MQTT_HOST", "mosquitto")
    port     = int(os.getenv("MQTT_PORT", 8883))
    ca       = os.getenv("CA_CERT",     "/certs/ca.crt")
    certfile = os.getenv("CLIENT_CERT", "/certs/client.crt")
    keyfile  = os.getenv("CLIENT_KEY",  "/certs/client.key")

    client = mqtt.Client(client_id="parc-stub", protocol=mqtt.MQTTv5)
    client.tls_set(ca_certs=ca, certfile=certfile, keyfile=keyfile,
                   tls_version=ssl.PROTOCOL_TLS_CLIENT)
    client.on_connect    = lambda c, u, f, rc, p: _log(f"MQTT connecté (rc={rc})", "inf")
    client.on_disconnect = lambda c, u, rc, p:    _log(f"MQTT déconnecté (rc={rc})", "inf")
    _log(f"Connexion à {host}:{port} …")
    client.connect(host, port, keepalive=60)
    return client


# ?? Publisher loop ????????????????????????????????????????????????????????
```

```
def _publish_loop():
    try:
        client = _build_client()
    except Exception as e:
        _log(f"Échec connexion MQTT : {e}", "err")
        return

    client.loop_start()
    t0 = time.time()
    phases: dict = {}

    while not _stop_event.is_set():
        with state_lock:
            interval = state["interval"]
            bad_rate = state["bad_rate"]
            serres   = [dict(s) for s in state["serres"]]
            cmds     = dict(_cmd_queue)
            _cmd_queue.clear()

        t = time.time() - t0

        for serre in serres:
            sn  = serre["numero"]
            mac = serre["mac"]

            # ?? handle manual commands ??????????????????????????????????????
            cmd = cmds.get(sn)
            if cmd == "connect" and not serre["connected"]:
                _publish_connect(client, serre)
                with state_lock:
                    for s in state["serres"]:
                        if s["numero"] == sn:
                            s["connected"] = True
                serre["connected"] = True
            elif cmd == "disconnect" and serre["connected"]:
                _publish_disconnect(client, serre)
                with state_lock:
                    for s in state["serres"]:
                        if s["numero"] == sn:
                            s["connected"] = False
                serre["connected"] = False

            # ?? random disconnect ???????????????????????????????????????????
            if serre["connected"] and serre["disconnect_rate"] > 0:
                if random.random() < serre["disconnect_rate"]:
                    _publish_disconnect(client, serre)
                    with state_lock:
                        for s in state["serres"]:
                            if s["numero"] == sn:
                                s["connected"] = False
                    serre["connected"] = False

            # ?? sensor data — only when connected ???????????????????????????
            if not serre["connected"]:
                continue

            for bac in serre.get("bacs", []):
                bn  = bac["numero"]
                key = (sn, bn)
```

```
                if key not in phases:
                    phases[key] = {s: random.uniform(0, 2 * math.pi) for s in SENSORS}

                topic = f"serre/{sn}/bac/{bn}"
                if random.random() < bad_rate:
                    payload = _bad_payload(t)
                    _log(f"[BAD]  {topic} ? {payload!r}", "bad")
                else:
                    data    = {s: _sine_value(s, t, phases[key][s]) for s in SENSORS}
                    payload = json.dumps(data)
                    _log(f"[OK]   {topic} ? {payload}", "ok")
                try:
                    client.publish(topic, payload)
                except Exception as e:
                    _log(f"Erreur publish : {e}", "err")

        _stop_event.wait(interval)

    client.loop_stop()
    client.disconnect()
    _log("Stub arrêté.")


# ?? Public API ??????????????????????????????????????????????????????????????
def start():
    global _thread
    if _thread and _thread.is_alive():
        return
    _stop_event.clear()
    _thread = threading.Thread(target=_publish_loop, daemon=True)
    _thread.start()
    with state_lock:
        state["running"] = True
        _save(state)
    _log("Stub démarré.")

def stop():
    _stop_event.set()
    with state_lock:
        state["running"] = False
    _log("Arrêt demandé.")

def connect_serre(numero: int):
    with state_lock:
        _cmd_queue[numero] = "connect"
    _log(f"[CMD] connexion serre {numero} demandée", "inf")

def disconnect_serre(numero: int):
    with state_lock:
        _cmd_queue[numero] = "disconnect"
    _log(f"[CMD] déconnexion serre {numero} demandée", "inf")

def save_config():
    """Called by Flask after any config/serres mutation."""
    with state_lock:
        _save(state)
```

Docker compose

```
services:
  parc-stub:
    build:
      context: .
    container_name: parc-stub
    restart: unless-stopped
    networks:
      - parc-net
    ports:
      - "5001:5000"                # Web UI ? http://localhost:5001
    volumes:
      - ../mosquitto/server-certs/ca.crt:/certs/ca.crt:ro
      - ../mosquitto/stub-certs/stub.crt:/certs/client.crt:ro
      - ../mosquitto/stub-certs/stub.key:/certs/client.key:ro
      - parc-stub-data:/data             # persists config.json across restarts
    environment:
      MQTT_HOST: mosquitto
      MQTT_PORT: "8883"
      CONFIG_PATH: /data/config.json
    logging:
      driver: json-file
      options:
        max-size: "5m"
        max-file: "3"

networks:
  parc-net:
    external: true                 # already declared in your main compose

volumes:
  parc-stub-data:                  # config.json persisted here
```