FalconEye

Prototype fonctionnel de supervision de systèmes embarqués critiques

Supervision temps réel pour environnements critiques.

Auteur : **Arthur Morain**

Août 2025

Table des matières

1 Introduction **3**
1.1 Présentation . . . 3
1.2 Motivations . . . 3
1.3 Contexte de la simulation . . . 3

2 Modules du projet **3**
2.1 Module `sensor-engine` . . . 3
2.2 Module `mqtt-broker` . . . 4
2.3 Module `data-collector` . . . 5
2.4 Module `anomaly-detector` . . . 6
2.5 Module `grafana` . . . 7
2.6 Module `security-layer` . . . 8
2.7 Module `tests` . . . 9
2.8 Module `documentation` . . . 9

3 Architecture **10**
3.1 Architecture - Simulateurs . . . 10
3.2 Architecture - Communication des données . . . 10
3.3 Architecture – Gestion des données . . . 11
3.4 Architecture – Système d'alertes . . . 12
3.5 Architecture – Tableau de bord . . . 12
3.6 Architecture – Sécurité . . . 13

4 Mise en place des infrastructures **14**
4.1 Réseau Falconnet . . . 14
4.2 Fonctionnement des simulateurs . . . 15
4.3 Déploiement du broker MQTT . . . 16
4.4 Listes de Contrôle d'Accès (ACL) . . . 17
4.5 Déploiement du collecteur de données . . . 18
4.6 Déploiement de Grafana et InfluxDB . . . 19

5 Détection des anomalies **20**
5.1 Choix mathématiques . . . 20
5.2 Comparaison des approches . . . 21
5.3 Limitations et optimisations possibles . . . 21
5.4 Surveillance de la présence des capteurs (script Bash) . . . 22

6 Intégration d'un reverse proxy NGINX **22**
6.1 Rôle d'un reverse proxy . . . 23
6.2 Pertinence dans notre contexte . . . 23

6.3 Exemple de mise en place avec Docker 23
6.4 Conclusion 24

7 Automatisation du déploiement 25

8 Sécurisation des communications et des accès 26
8.1 TLS sur Mosquitto 26
8.2 Pare-feu pour réduire la surface d'attaque 26
8.3 Fail2ban pour réagir dynamiquement aux attaques 27
8.4 Choix retenu dans ce projet 27

9 Tests et Validation 28
9.1 Test d'intégrité des messages 28
9.2 Test d'intrusion réseau 28
9.3 Test des certificats et ACLs 28
9.4 Test de connectivité réseau 28
9.5 Test End-to-End 29

10 Conclusion 29
10.1 Améliorations possibles 29
10.2 Dashboards Grafana 30

1 Introduction

1.1 Présentation

FalconEye est un prototype fonctionnel de supervision de systèmes embarqués critiques, destiné à simuler la surveillance en temps réel des composants sensibles d'un avion de chasse : moteur, hydraulique, structure, et navigation. Il repose sur une architecture distribuée, sécurisée et modulaire, inspirée des contraintes industrielles rencontrées dans les secteurs de l'aéronautique et de la défense.

1.2 Motivations

Après deux projets orientés sur les thématiques des réseaux et de la cryptographie, j'ai souhaité m'engager dans une démarche plus ambitieuse, avec une approche systémique et industrielle.

L'objectif de ce projet est de respecter les exigences d'un développement technique en environnement professionnel : conception d'architecture, mise en place de la sécurité, tests unitaires et d'intégration, documentation technique complète et démonstration reproductible.

1.3 Contexte de la simulation

Dans le cadre de ce projet, l'avion de chasse est modélisé comme un système embarqué constitué de multiples capteurs critiques. Ces capteurs transmettent en continu leurs données sur un bus réseau interne sécurisé, reflétant les contraintes réelles des communications embarquées.

Les données collectées par ces capteurs sont ensuite :

— collectées et centralisées via un protocole de communication sécurisé,
— stockées dans une base de données temporelle pour un accès performant,
— visualisées en temps réel à travers des tableaux de bord dynamiques,
— analysées afin de détecter automatiquement les comportements anormaux ou les défaillances potentielles,
— gérées par un système d'alerting automatique permettant la notification immédiate des opérateurs via des canaux tels que Discord ou email.

2 Modules du projet

2.1 Module `sensor-engine`

Le module `sensor-engine` gère la simulation des systèmes embarqués critiques d'un avion de chasse. Il modélise en temps réel plusieurs sous-systèmes essentiels, en respectant

les niveaux de criticité définis par les normes DO-178C (logiciel) et DO-254 (matériel).

Systèmes critiques simulés et niveaux de criticité Selon la classification des niveaux de criticité (Design Assurance Levels, DAL) des normes DO-178C et DO-254, les systèmes suivants sont simulés :

- **Système moteur (Engine Control Unit - ECU)**
  - *DAL A* : contrôle moteur vital pour la sécurité.
  - Paramètres simulés : température (40°C à 1200°C), pression d'huile (0 à 10 bars), vibrations (0 à 50 g).
- **Systèmes hydrauliques**
  - *DAL B* : commande des gouvernes et freins.
  - Paramètres simulés : pression hydraulique (0 à 350 bars), état des pompes.
- **Structure et intégrité mécanique**
  - *DAL C* : surveillance des vibrations et contraintes mécaniques.
  - Paramètres simulés : accélérations (0 à 20 g).
- **Systèmes avioniques de navigation**
  - *DAL A* : données GPS, vitesse air essentielles au vol.
  - Paramètres simulés : coordonnées GPS, vitesse air (0 à 2500 km/h).
- **Systèmes de contrôle de vol**
  - *DAL A* : capteurs d'angle d'attaque, position des gouvernes.
  - Paramètres simulés : angle d'attaque (15° à +25°), positions gouvernes (30° à +30°).

Références normatives Les niveaux de criticité s'appuient sur :

- RTCA DO-178C, pour les logiciels embarqués.
- RTCA DO-254, pour les composants électroniques matériels.

2.2 Module `mqtt-broker`

Le module `mqtt-broker` constitue l'élément central de communication entre les capteurs simulés, les modules d'analyse, et l'interface de visualisation. Il repose sur le protocole MQTT (Message Queuing Telemetry Transport), reconnu pour sa légèreté, sa faible latence, et son adéquation aux systèmes embarqués distribués, comme ceux que l'on retrouve dans les environnements aéronautiques.

Fonctionnalité

- **Réception des messages** : les capteurs simulés (`sensor-engine`) publient périodiquement des messages sur différents topics (ex. `aircraft/engine/temp`).
- **Routage vers les consommateurs** : les modules abonnés aux topics correspondants (visualisation, analyse d'anomalies) reçoivent les données en temps réel.

— **Gestion fine des clients** : le broker gère l'authentification, les autorisations (ACL), la rétention de messages, et la gestion des sessions.
— **Support des niveaux de QoS (Quality of Service)** :
  — QoS 0 : At most once (meilleur effort)
  — QoS 1 : At least once (accusé de réception)
  — QoS 2 : Exactly once (sécurité maximale)

Sécurisation Étant donné la sensibilité des données échangées (simulées comme critiques pour un avion de chasse), le module implémente plusieurs mesures de cybersécurité :
— **Authentification des clients** via certificats X.509 ou login/mot de passe.
— **Chiffrement TLS** pour sécuriser les flux MQTT (port 8883).
— **Contrôle d'accès** (ACL) pour restreindre la publication et la souscription à certains topics uniquement.
— **Protection contre les abus** : limitation du débit, du nombre de connexions et surveillance des logs.

Implémentation Le module repose sur un broker open-source (Mosquitto ou EMQX), configuré localement et scriptable pour être intégré dans la chaîne de simulation. Il est également possible de visualiser les messages en direct via des outils comme MQTT Explorer ou Node-RED.

Exemple de topics utilisés

```
aircraft/engine/temp
aircraft/hydraulics/pressure
aircraft/structure/vibrations
aircraft/nav/gps
alerts/critical
```

Conclusion Le protocole MQTT, bien que simple, se révèle particulièrement adapté à la communication efficace entre capteurs et analyseurs dans un environnement embarqué simulé. Couplé à une sécurisation adéquate, il permet de respecter les exigences des systèmes critiques militaires, tout en gardant une grande souplesse d'intégration. Cependant, ce n'est pas un protocole envisageable dans un cas réel.

2.3 Module `data-collector`

Le module `data-collector` a pour rôle de centraliser, valider et stocker de manière persistante toutes les données émises par les capteurs simulés et reçues via le `mqtt-broker`. Il constitue un point d'entrée critique dans la chaîne de supervision, permettant l'analyse ultérieure, la visualisation ou le déclenchement d'alertes.

Fonctionnalité

- **Souscription aux données** : le `data-collector` est un client MQTT qui s'abonne à tous les topics simulant des données capteurs (`aircraft/#`).
- **Validation des messages** : il vérifie la structure JSON reçue, les champs obligatoires (`timestamp`, `value`, `unit`, etc.) et leur cohérence.
- **Stockage des séries temporelles** : les mesures sont persistées dans une base de données InfluxDB, spécialisée dans les données horodatées.
- **Structuration intelligente** : chaque capteur devient une "mesure" InfluxDB, enrichie de *tags* comme `source`, `system`, `unit`, etc.
- **Exposition des données** : les données collectées peuvent ensuite être consultées par l'interface graphique ou des modules d'analyse en aval.

Choix d'InfluxDB InfluxDB a été choisi pour sa capacité à :

- gérer efficacement des milliers de points par seconde avec une faible latence,
- compresser automatiquement les données pour un stockage long terme optimisé,
- offrir un langage de requête performant (`Flux`) pour les agrégations temporelles,
- fournir une API HTTP native permettant une intégration simple avec d'autres composants.

Exemple de données stockées

```
measurement: engine_temperature
tags:
    source: sensor-01
    unit: Celsius
    aircraft_zone: rear
fields:
    value: 725.0
time: 2025-07-26T10:30:00Z
```

Le `data-collector` permet une historisation fiable et performante des mesures critiques simulées. Il s'inscrit dans une logique de traçabilité, d'analyse et de visualisation à long terme, tout en respectant la structure d'un système de supervision industriel moderne.

2.4 Module `anomaly-detector`

Le module `anomaly-detector` est responsable de la détection automatique des comportements anormaux à partir des données stockées dans InfluxDB. Son objectif est de détecter toute dérive critique, de classifier son niveau de gravité et de déclencher un mécanisme d'alerte en conséquence.

Fonctionnement général Le module interroge régulièrement InfluxDB pour analyser les dernières mesures collectées par le `data-collector`. Ces analyses sont réalisées selon des règles prédéfinies, basées sur :

— des **seuils statiques** (ex : température moteur ¿ 800°C),
— des **écarts dynamiques** (ex : montée trop rapide de la température),
— des **fenêtres temporelles glissantes** (analyse de tendance).

Fréquence adaptative selon la criticité Chaque sous-système de l'avion est associé à un *niveau de criticité* (inspiré du DO-178C/DO-254). Le module adapte sa fréquence d'analyse à ce niveau :

— **Criticité A (ex : moteur)** : analyse toutes les 2 secondes.
— **Criticité B (ex : hydraulique)** : toutes les 5 secondes.
— **Criticité C-D (structure, navigation)** : toutes les 10 à 15 secondes.

Cela permet d'optimiser les performances du système tout en respectant les exigences de sûreté de fonctionnement.

Détection et alerte Lorsqu'une anomalie est détectée, un message est immédiatement publié sur un topic MQTT dédié au système concerné. Exemple :

```
Topic : alerts/engine/overheat
Payload : {
    "timestamp": "2025-07-26T10:30:00Z",
    "system": "engine",
    "type": "temperature",
    "value": 825,
    "level": "CRITICAL",
    "message": "Engine temperature too high"
}
```

Ces messages peuvent être consommés par le module `grafana` pour être relayés vers Discord, email ou autres canaux d'alerte.

2.5 Module `grafana`

Le module `grafana` constitue l'interface principale de visualisation et de supervision des données collectées et des alertes générées par le système. Connecté à la base de données InfluxDB, il permet de créer des dashboards interactifs en temps réel, facilitant la surveillance des paramètres critiques simulés.

Visualisation des données Grafana récupère les données historiques et en temps réel stockées dans InfluxDB. Il offre une grande variété de panneaux graphiques (courbes,

jauges, histogrammes) permettant de suivre l'évolution des mesures telles que la température moteur, la pression hydraulique, ou les vibrations structurelles.

Gestion des alertes Bien que Grafana ne puisse pas directement consommer les messages MQTT, il reçoit les alertes émises par le module `anomaly-detector` via un canal intermédiaire (par exemple, insertion d'une entrée dans InfluxDB dédiée aux alertes ou un webhook HTTP). Ces alertes sont alors affichées dans les dashboards et peuvent déclencher des notifications vers différents canaux (email, Discord, Slack).

Avantages et limites

- **Avantages :**
  - Interface graphique riche et configurable,
  - Support natif des bases de données séries temporelles comme InfluxDB,
  - Possibilité de définir des seuils d'alerte simples avec notifications intégrées,
  - Accès facile aux historiques et à la corrélation des données.
- **Limites :**
  - Dépend d'un module tiers pour l'injection des alertes dans la base de données.

Conclusion Le module `grafana` joue un rôle central dans la supervision visuelle et la gestion simple des alertes. Associé au module `anomaly-detector`, il permet une surveillance efficace des systèmes critiques simulés tout en offrant une interface utilisateur accessible et moderne.

2.6 Module `security-layer`

Le module `security-layer` assure la sécurisation des communications et des accès aux différents services du système FalconEye, en particulier ceux exposés via des interfaces web comme Grafana. Ce module est essentiel pour garantir la confidentialité, l'intégrité, et l'authentification dans un environnement critique.

Sécurisation des accès à Grafana Un reverse proxy NGINX est déployé devant Grafana afin de :

- Assurer la terminaison TLS/SSL, garantissant le chiffrement des échanges entre les utilisateurs et Grafana.
- Mettre en place une authentification préalable (exemple : Basic Auth, OAuth ou LDAP) pour restreindre l'accès aux dashboards.
- Appliquer des règles de filtrage, comme le blocage d'adresses IP suspectes ou la limitation du nombre de requêtes (rate limiting), afin de protéger contre les attaques par déni de service ou d'autres menaces réseau.
- Gérer de manière centralisée les certificats TLS, simplifiant la maintenance et le déploiement de la sécurité.

Sécurisation transverse Outre Grafana, le module `security-layer` protège également l'ensemble des services critiques du système (broker MQTT, API, bases de données) via des mécanismes similaires, assurant une politique de sécurité homogène et centralisée.

Complémentarité avec la sécurité intégrée au MQTT broker Bien que le broker MQTT intègre ses propres mécanismes de sécurité (TLS, authentification des clients), le module `security-layer` apporte une couche supplémentaire et globale, notamment via le reverse proxy NGINX, qui agit comme un point d'entrée sécurisé unique pour tous les composants exposés.

Conclusion La mise en place du module `security-layer` garantit que les communications sensibles sont chiffrées, que les accès sont contrôlés, et que le système dans son ensemble est mieux protégé contre les attaques et les intrusions, ce qui est indispensable dans un contexte de systèmes embarqués critiques.

2.7 Module `tests`

Le module `tests` regroupe l'ensemble des tests automatisés permettant de valider la qualité et la fiabilité du système FalconEye. Il comprend :

- **Tests unitaires** : vérification du bon fonctionnement des fonctions et composants individuels (simulateurs, collecteur, détecteur d'anomalies).
- **Tests d'intégration** : contrôle des échanges entre modules, notamment la communication MQTT, l'écriture dans InfluxDB, et la génération des alertes.
- **Tests fonctionnels** : simulation de scénarios complets visant à valider le comportement global et la conformité aux exigences.
- **Tests de performance** : évaluation des performances en conditions de charge élevée, garantissant la capacité du système à traiter les données en temps réel.

Ces tests sont automatisés à l'aide d'outils adaptés (ex. pytest, JUnit) et exécutés régulièrement pour assurer la stabilité du code lors des évolutions.

2.8 Module `documentation`

Le module `documentation` regroupe tous les documents liés au projet, essentiels pour la compréhension, l'utilisation, et la maintenance du système :

- **Architecture** : description détaillée des modules, de leurs interactions et des choix techniques.
- **Spécifications** : définition des exigences fonctionnelles et non fonctionnelles.
- **Guides utilisateurs** : instructions pour l'installation, la configuration et l'utilisation des interfaces (ex. dashboards Grafana).
- **Documentation des tests** : protocoles de tests, résultats et procédures de validation.

3 Architecture

3.1 Architecture - Simulateurs

La figure 1 présente l'architecture générale du module `sensor-engine`, qui simule les différents systèmes embarqués critiques de l'avion de chasse.

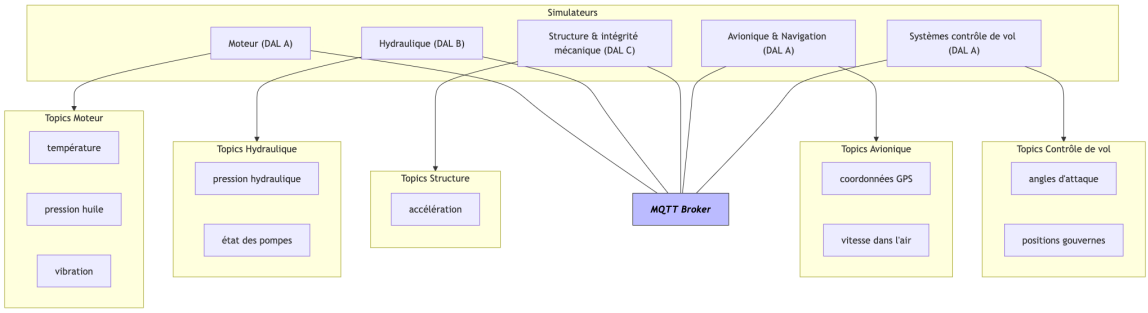

FIGURE 1 – Architecture du module `sensor-engine` simulant les systèmes embarqués critiques

3.2 Architecture - Communication des données

La figure 2 illustre l'architecture sécurisée du module `mqtt-broker`. Ce module centralise la communication entre les simulateurs (`sensor-engine`) et les composants d'analyse, de visualisation et d'alerte. La sécurité est assurée par une couche intégrant TLS et un système d'authentification.

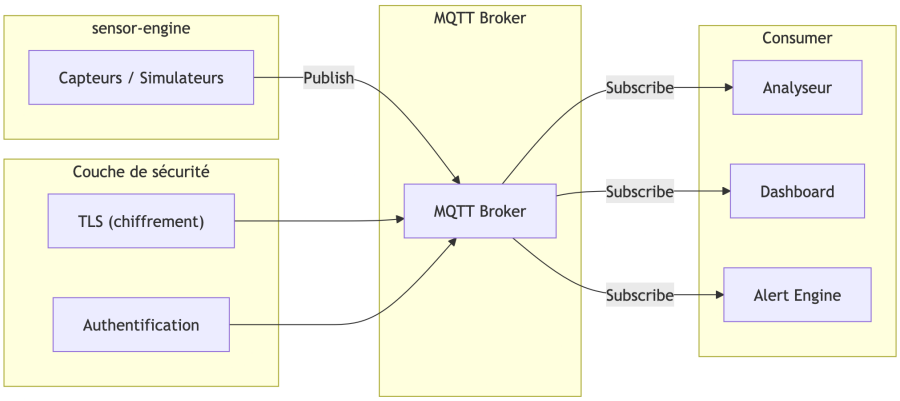

FIGURE 2 – Architecture sécurisée du module MQTT Broker avec publish/subscribe

3.3 Architecture – Gestion des données

La figure 3 illustre le fonctionnement du module `data-collector` et son interaction avec l'infrastructure de stockage et d'analyse de données critiques. Ce module joue un rôle central dans la réception, la validation et la persistance des mesures simulées issues des capteurs embarqués.

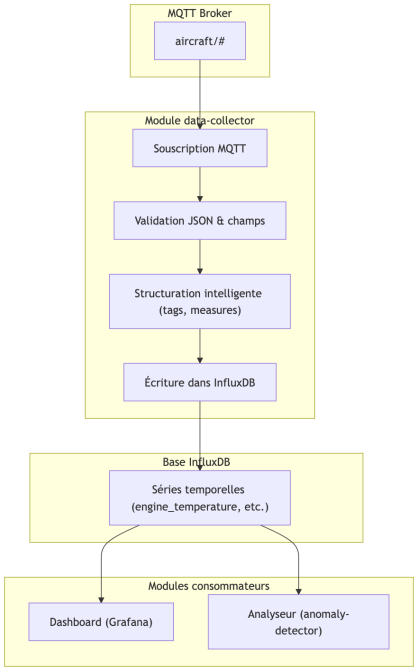

FIGURE 3 – Architecture de la gestion des données via le module `data-collector` et InfluxDB

3.4 Architecture – Système d'alertes

La figure 4 présente le fonctionnement du module `anomaly-detector`, chargé de la surveillance automatique des paramètres critiques simulés. Ce composant interroge régulièrement la base de données InfluxDB pour analyser les mesures collectées, selon trois types de règles : seuils statiques, variations dynamiques et tendances temporelles.

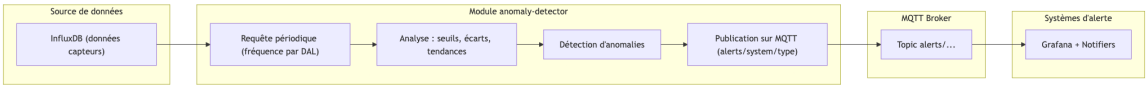

FIGURE 4 – Architecture du module `anomaly-detector` pour la détection d'anomalies

3.5 Architecture – Tableau de bord

La figure 5 illustre le rôle du module `grafana` dans l'interface de supervision de FalconEye. Ce composant permet la visualisation en temps réel des données critiques et des alertes émises par le système.

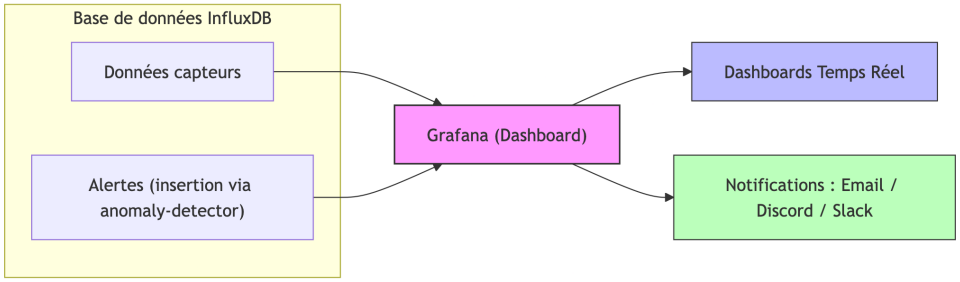

FIGURE 5 – Architecture d'intégration du module `grafana` pour la supervision

3.6 Architecture – Sécurité

La figure 6 présente l'architecture du module `security-layer`, qui assure la sécurisation des communications et des accès aux composants critiques du système FalconEye. Il repose sur un reverse proxy NGINX, configuré pour jouer le rôle de point d'entrée sécurisé unique.

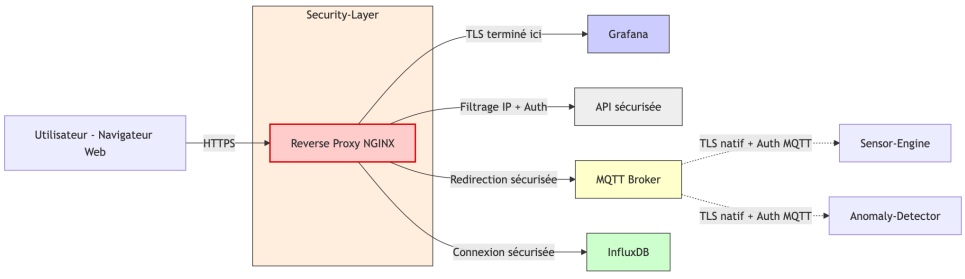

FIGURE 6 – Architecture de la couche de sécurité via le module `security-layer` (NGINX)

4 Mise en place des infrastructures

4.1 Réseau Falconnet

Afin d'isoler efficacement les services et d'assurer une communication fiable, un réseau Docker dédié nommé *falconnet* a été mis en place.

Les trois services principaux sont :

— **Mosquitto** : le broker MQTT, essentiel pour la communication entre les capteurs et les consommateurs de données ;

— **InfluxDB** : la base de données temporelle dans laquelle sont stockées toutes les valeurs transmises par les capteurs ;

— **Grafana** : la plateforme de visualisation utilisée pour concevoir les tableaux de bord de supervision.

Chaque système embarqué de l'avion (cf. figure 1) est encapsulé dans un conteneur Docker distinct, ce qui garantit une isolation logicielle renforcée et améliore la maintenabilité de l'ensemble.

Un conteneur spécifique, nommé *mqtt-collector-influxdb*, assure la réception, la validation et le transfert des messages JSON envoyés par les capteurs vers la base de données. Il agit ainsi comme une passerelle de confiance entre le monde des capteurs et le stockage des données.

Un script de test d'intégration, `check_network.sh`, permet de vérifier à la fois la présence de tous les conteneurs dans le réseau *falconnet*, ainsi que la connectivité entre les services critiques, notamment le chemin **Mosquitto → MQTT Collector → InfluxDB**.

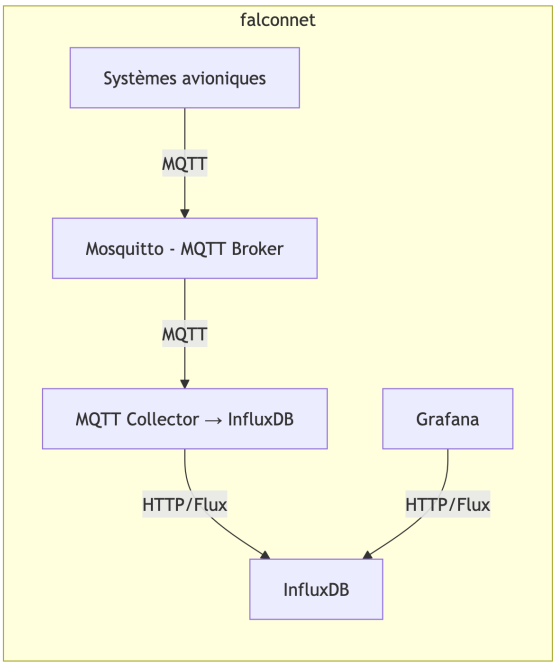

FIGURE 7 – Architecture réseau FalconEye

4.2 Fonctionnement des simulateurs

Les simulateurs jouent un rôle crucial dans la génération de données pour les différents systèmes avioniques. Leur but principal est de reproduire, de manière réaliste, le comportement des capteurs et des instruments embarqués afin de tester l'intégration et le fonctionnement global de la plateforme.

Chaque simulateur est encapsulé dans un conteneur Docker distinct, ce qui permet une isolation stricte entre les systèmes et facilite la maintenance ainsi que l'évolution indépendante des modules. Ces simulateurs génèrent en continu des données de navigation, telles que la latitude, la longitude, la vitesse, ou encore la température, avec un intervalle de publication configurable.

Le processus de simulation repose sur des modèles d'évolution réalistes des paramètres mesurés : par exemple, la position GPS est modifiée à chaque cycle en fonction de la vitesse simulée, ce qui assure un déplacement cohérent dans l'espace. Des anomalies aléatoires sont également introduites pour tester la robustesse du système face aux imprévus, tels que des sauts brusques dans la position ou des variations extrêmes de vitesse.

Les messages produits par les simulateurs sont formatés en JSON et publiés sur des topics MQTT dédiés via le broker Mosquitto. Ce protocole léger et asynchrone permet une communication efficace entre les simulateurs et le système de collecte de données.

Cette architecture modulaire assure non seulement une simulation réaliste, mais aussi une grande flexibilité pour intégrer de nouveaux capteurs ou ajuster les modèles de simulation selon les besoins.

Extrait de code de simulation :

```
import time
import json
import random
import paho.mqtt.client as mqtt
from datetime import datetime, timezone

mqtt_client = mqtt.Client()
mqtt_client.connect("mosquitto", 1883)

latitude = 48.8566
longitude = 2.3522
airspeed = 800.0

def evolve_position(lat, lon, speed_kmh, interval_s=1):
    # Approximate movement in degrees (1 deg  111 km)
    distance_km = speed_kmh * (interval_s / 3600)
    delta_deg = distance_km / 111
```

```
    # Simulate movement north-east for simplicity
    lat += delta_deg * (random.uniform(0.8, 1.2))
    lon += delta_deg * (random.uniform(0.8, 1.2))
    return round(lat, 6), round(lon, 6)

while True:
    latitude, longitude = evolve_position(latitude, longitude, airspeed)
    message = {
        "timestamp": datetime.now(timezone.utc).isoformat(),
        "latitude": latitude,
        "longitude": longitude,
        "airspeed": airspeed,
        "unit": "degrees",
        "source": "nav-sensor-01"
    }
    mqtt_client.publish("aircraft/navigation/position/1", json.dumps(message))
    time.sleep(1)
```

4.3 Déploiement du broker MQTT

Le broker MQTT est indispensable pour la communication des capteurs vers la base de données *InfluxDB*. Pour assurer cette communication, les ports **8883** (MQTT sur TLS) et **9001** (MQTT via WebSocket sur TLS) ont été ouverts.

Pour la mise en place de **TLS**, une autorité de certification interne a été créée afin de signer à la fois le certificat du serveur et ceux des différents clients (capteurs, base de données, etc.). Le script *create_client_cert.sh* automatise la création de ces certificats clients.

La configuration impose que tout client souhaitant se connecter au broker présente un certificat signé par la même autorité de certification que le serveur, grâce au paramètre *require_certificate*.

Le chiffrement des communications utilise la suite de chiffrement **ECDHE** (Elliptic Curve Diffie-Hellman Ephemeral), qui permet un échange de clés éphémères à chaque connexion. Cette approche garantit la *Perfect Forward Secrecy* (PFS) : même si une clé de session venait à être compromise, les communications passées resteraient protégées.

En complément de cette sécurisation par chiffrement, un mécanisme de **Listes de Contrôle d'Accès** (ACL) a été mis en place pour restreindre les actions que chaque client est autorisé à effectuer sur des topics précis, renforçant ainsi la sécurité globale du système.

4.4 Listes de Contrôle d'Accès (ACL)

En plus de l'authentification par certificat TLS, le broker MQTT utilise des **Listes de Contrôle d'Accès** (ACL) pour limiter les actions des différents clients sur des topics spécifiques. Les ACL permettent par exemple de restreindre un capteur de moteur à publier uniquement sur les topics `aircraft/engine/#` et d'empêcher tout accès aux topics réservés à d'autres sous-systèmes.

La configuration d'origine visait à utiliser l'option *use_identity_as_username* de Mosquitto, qui associe automatiquement l'identité (CN) contenue dans le certificat client au nom d'utilisateur pour l'ACL. Cependant, nous n'avons pas pu mettre en place ce mécanisme car le broker ne parvenait pas à extraire correctement le CN lors de la connexion TLS.

Face à cette limitation, nous avons opté pour une authentification par nom d'utilisateur et mot de passe via un fichier `passwd` généré avec l'outil `mosquitto_passwd`. Chaque client se connecte ainsi avec des identifiants spécifiques, et les ACL sont appliquées sur la base de ce nom d'utilisateur.

Bien que cette solution assure toujours un contrôle d'accès efficace, la combinaison de l'identité issue des certificats et du contrôle par mot de passe aurait permis une **double authentification** renforçant encore la sécurité.

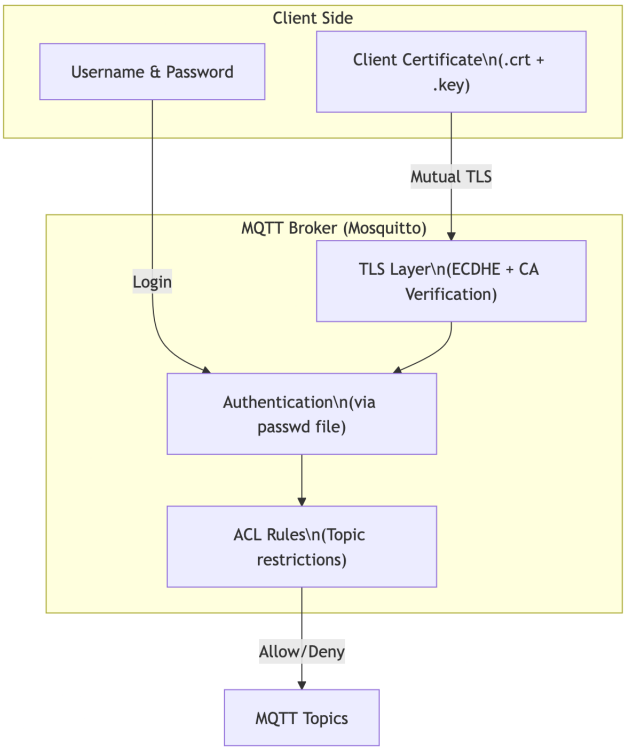

FIGURE 8 – ACL/TLS du système Mosquitto

4.5 Déploiement du collecteur de données

Le collecteur de données a pour rôle de faire le lien entre le **broker MQTT Mosquitto** et la base de données **InfluxDB**. Il s'abonne aux différents sujets (*topics*) liés à l'avion et insère les messages reçus sous forme de points exploitables par Grafana.

Connexion au broker MQTT. Le collecteur utilise la bibliothèque `paho-mqtt` pour établir une connexion sécurisée avec le broker Mosquitto. L'authentification est réalisée par certificat TLS et le collecteur s'abonne au sujet global `aircraft/#`, ce qui lui permet de recevoir tous les messages publiés par les capteurs embarqués.

Réception et traitement des messages. Lorsqu'un message est reçu, il est décodé depuis le format `JSON`. Le collecteur détermine ensuite le type de données en analysant la structure du sujet MQTT :

- **Alertes** : les messages de type `aircraft/alerts/...` contiennent des informations critiques (ex. température moteur, anomalies). Ils sont enrichis de métadonnées comme la sévérité, la zone de l'appareil et l'unité de mesure.
- **Position GPS** : les messages associés au sujet `aircraft/position/...` fournissent la latitude et la longitude. Ces données sont stockées avec des champs adaptés (coordonnées géographiques et unité en degrés).
- **Données capteurs standards** : pour les autres capteurs, le collecteur enregistre la valeur mesurée, l'unité et les informations contextuelles (source, zone de l'avion).

Écriture dans InfluxDB. Chaque message est transformé en un `Point` (format natif d'InfluxDB) comportant :

- des *tags* : informations qualitatives utilisées pour filtrer les données (nom du capteur, unité, zone de l'avion, sévérité, etc.) ;
- des *fields* : valeurs numériques ou textuelles mesurées (par exemple température, latitude, taux d'alerte) ;
- un *timestamp* : instant précis fourni par le capteur.

Ces points sont ensuite insérés de manière synchrone dans le **bucket InfluxDB**, garantissant la persistance et la cohérence temporelle des données.

Boucle de fonctionnement. Une fois initialisé, le collecteur reste en écoute permanente sur le broker MQTT. Il enregistre automatiquement toute nouvelle donnée publiée par les capteurs de l'avion et assure ainsi un flux continu d'informations vers la base de données temporelle.

Résumé. Ce composant est essentiel pour transformer les messages MQTT en données exploitables par InfluxDB. Il assure la **collecte fiable, la normalisation et le stockage**

sécurisé des informations de vol, permettant à Grafana de générer des tableaux de bord temps réel à partir de ces données.

4.6 Déploiement de Grafana et InfluxDB

Le déploiement de l'infrastructure de visualisation et de stockage des données de **FalconEye** repose sur Docker Compose. Les services principaux sont :

- **InfluxDB** : stockage temporel des données brutes des capteurs et des alertes générées par le collector. L'instance est configurée via les variables d'environnement suivantes :
  - `DOCKER_INFLUXDB_INIT_USERNAME`, `DOCKER_INFLUXDB_INIT_PASSWORD` : utilisateur administrateur (ici stocké en clair pour simplification)
  - `DOCKER_INFLUXDB_INIT_ORG`, `DOCKER_INFLUXDB_INIT_BUCKET` : organisation et bucket initial
  - `DOCKER_INFLUXDB_INIT_ADMIN_TOKEN` : token d'accès

  L'authentification est donc activée, mais les mots de passe étant stockés en clair, ce système ne doit pas être utilisé en production.
- **Grafana** : visualisation des données et création de dashboards. Chaque simulateur dispose d'un tableau de bord dédié pour suivre ses variables, et un tableau de bord supplémentaire est prévu pour les alertes. L'accès est sécurisé par un utilisateur administrateur et un mot de passe configurés via :
  - `GF_SECURITY_ADMIN_USER`
  - `GF_SECURITY_ADMIN_PASSWORD`

  Toutefois, le tableau de bord des alertes n'est pas fonctionnel pour le moment, car la gestion des alertes multi-capteurs et la limitation du nombre de faux positifs n'ont pas encore été entièrement implémentées.
- **Collector** : service intermédiaire qui récupère les messages MQTT, valide leur format et les injecte dans InfluxDB. Il dépend des services `mosquitto` et `influxdb`.

Architecture réseau Tous les services communiquent via le réseau Docker nommé `falconnet`, assurant l'isolation et la connectivité entre simulateurs, broker MQTT, collector, InfluxDB et Grafana.

Volumes et persistance Les données InfluxDB et Grafana sont montées sur des volumes locaux pour assurer la persistance au redémarrage des conteneurs.

Limites actuelles

- Les mots de passe et tokens sont stockés en clair dans le Docker Compose, ce qui constitue un risque de sécurité.
- Le tableau de bord des alertes n'est pas encore opérationnel, et les alertes basées sur les valeurs des capteurs n'ont pas été mises en place, afin d'éviter un flux massif

de faux positifs dû au comportement aléatoire des simulateurs.
— Grafana permet d'observer les données historiques et les valeurs associées aux capteurs, mais les alertes doivent être consultées directement dans InfluxDB.

5 Détection des anomalies

Le système **FalconEye** intègre deux modes d'analyse complémentaires pour la génération d'alertes à partir des données de capteurs avioniques :

— **Analyse en temps réel via MQTT** : chaque message publié par les capteurs est traité dès sa réception. Une logique de filtrage (EMA + règles de seuils + détection de dérivée) est appliquée immédiatement. Les alertes sont publiées vers un canal dédié (`aircraft/alerts/...`) si une anomalie est détectée.
— **Analyse par backfill (InfluxDB)** : les données brutes sont d'abord stockées dans InfluxDB. Un script périodique interroge l'historique récent (par exemple les 10 dernières minutes) et applique les mêmes règles de détection. Les alertes sont émises avec un léger décalage temporel, mais de façon plus robuste et moins sensible aux variations ponctuelles.

5.1 Choix mathématiques

1. Lissage par moyenne exponentielle (EMA) Les deux scripts utilisent une *moyenne exponentielle mobile* pour réduire le bruit :

$$EMA_t = \alpha \cdot x_t + (1 - \alpha) \cdot EMA_{t-1}$$

où :
— x_t : valeur brute au temps t
— α : facteur de lissage ($\alpha = 0.25$ dans notre cas)
— EMA_t : valeur lissée

Ce choix permet d'atténuer les fluctuations aléatoires tout en restant réactif aux tendances réelles.

2. Détection de dérivée (vitesse de variation) On calcule le **taux de variation** entre deux instants successifs :

$$rate = \frac{EMA_t - EMA_{t-\Delta t}}{\Delta t}$$

Une alerte est générée si $|rate|$ dépasse un seuil prédéfini ($rate_{thresh}$).

Exemple : pour la vibration moteur ($max = 50$, $rate_{thresh} = 5$), une augmentation de 20 unités en 3 secondes donne :

$$rate = \frac{20}{3} \approx 6.67 > 5 \quad \Rightarrow \quad alerteslope$$

3. Vérification de seuils Chaque mesure possède des bornes $[min, max]$. Si une valeur sort de cet intervalle, une alerte `threshold` est générée :

$$alert = (x_t < min) \ \vee \ (x_t > max)$$

Exemple : si la pression dépasse 350 PSI, une alerte immédiate est émise.

5.2 Comparaison des approches

Critère	**Temps réel (MQTT)**	**Différé (InfluxDB)**
Réactivité	Très élevée (alerte instantanée)	Moins réactive (fenêtre de 10 min)
Robustesse au bruit	Faible (risque de faux positifs)	Meilleure (analyse historique)
Consommation	Continue (loop infinie)	Périodique (requêtes plus lourdes)
Déploiement	Simple (MQTT + règles locales)	Plus complexe (InfluxDB requis)
Cas d'usage	Sécurité en vol (temps réel)	Validation / analyse post-vol

5.3 Limitations et optimisations possibles

— **Trop d'alertes en sortie** : les simulateurs produisent des signaux bruités ⇒ de nombreux faux positifs. *Solutions possibles* : ajuster α, augmenter $rate_{thresh}$, introduire une notion de persistance temporelle (par ex. alerte seulement si anomalie persiste n secondes).

— **Cooldown par clé trop simple** : actuellement, une alerte est bloquée pendant $30s$ après émission pour un capteur donné. *Optimisation* : rendre le cooldown adaptatif, proportionnel à l'amplitude de l'anomalie.

— **Absence de corrélations inter-capteurs** : les règles sont mono-capteur. *Optimisation* : introduire des règles multi-sources (ex. température moteur + pression d'huile).

— **Fusion des deux approches** : utiliser **MQTT temps réel** pour la détection immédiate et **InfluxDB différé** pour confirmer ou invalider les alertes a posteriori.

5.4 Surveillance de la présence des capteurs (script Bash)

En complément des deux approches (temps réel via MQTT et différée via InfluxDB), un **script Bash** a été mis en place afin de vérifier que tous les conteneurs simulateurs de capteurs sont bien présents dans le réseau `falconnet`.

Le script s'exécute en boucle toutes les 5 secondes, interroge l'état du réseau Docker et compare la liste des services attendus (moteur, hydraulique, navigation, etc.) avec ceux réellement connectés. Si un service est manquant, une notification est envoyée directement sur l'ordinateur de l'opérateur (via `notify-send` sur Linux ou `osascript` sur macOS).

Avantages :

- Simplicité de mise en œuvre (script léger, aucune dépendance complexe).
- Réactivité immédiate (alerte visuelle/lokale en quelques secondes).
- Indépendant d'InfluxDB et de Grafana (fonctionne même si la stack de monitoring est arrêtée).

Inconvénients :

- Notifications locales uniquement : les alertes ne sont pas intégrées dans le **tableau de bord Grafana**, donc non visibles dans l'interface principale de l'avion.
- Pas d'historique centralisé : contrairement aux alertes InfluxDB, aucune trace n'est stockée pour analyse ultérieure.
- Risque de surcharge visuelle si plusieurs capteurs sont absents en même temps.

Discussion : Ce mécanisme répond à un besoin de supervision basique mais manque d'intégration dans l'écosystème FalconEye. Idéalement, ces alertes devraient être émises vers InfluxDB puis affichées dans Grafana. Actuellement :

- Les **alertes de présence** (capteurs manquants) sont visibles uniquement via ce script.
- Les **alertes associées aux valeurs** (dépassements de seuils, dérivées) ne sont pas affichées dans Grafana, car le tableau de bord est encore incomplet et les faux positifs liés au bruit rendaient la solution inutilisable.
- Néanmoins, il est toujours possible de consulter les **alertes brutes dans InfluxDB**, qui centralise les événements détectés.

Une amélioration future consisterait à **fusionner ce script Bash avec la logique d'alerting Grafana**, afin d'obtenir un tableau de bord complet (valeurs anormales + capteurs absents) dans l'interface de supervision.

6 Intégration d'un reverse proxy NGINX

Dans l'architecture actuelle, les capteurs envoient leurs données vers un *broker* Mosquitto sécurisé par TLS, puis un collecteur traite les informations et les insère dans **In-**

fluxDB. Enfin, **Grafana** permet de visualiser les tableaux de bord construits à partir de ces données.

6.1 Rôle d'un reverse proxy

Un **reverse proxy** comme **NGINX** se place devant un ou plusieurs services afin de servir de point d'entrée unique. Il est principalement utilisé pour :

— Terminer les connexions SSL/TLS (certificats centralisés).
— Appliquer des règles de sécurité supplémentaires (authentification, filtrage IP).
— Unifier l'accès aux services (*routing* basé sur l'URL ou le sous-domaine).
— Cacher ou compresser certaines réponses pour améliorer les performances.

6.2 Pertinence dans notre contexte

Dans notre projet, les communications *MQTT* entre capteurs et Mosquitto sont déjà sécurisées par TLS. De plus, l'utilisation de Grafana est prévue uniquement dans un réseau restreint (tableau de bord d'un avion de chasse), ce qui limite l'intérêt immédiat d'un reverse proxy.

Toutefois, si l'on souhaitait exposer Grafana ou l'API InfluxDB à des utilisateurs distants (par exemple dans un centre de contrôle au sol), alors NGINX pourrait être inséré comme *point de sortie unique*. Il assurerait ainsi la sécurisation des accès et la centralisation de la configuration SSL.

6.3 Exemple de mise en place avec Docker

Une configuration typique en `docker-compose` permettrait de définir trois services : InfluxDB, Grafana et NGINX. NGINX serait le seul à exposer ses ports vers l'extérieur (80/443), et redirigerait ensuite les requêtes vers les services internes.

Listing 1 – Exemple simplifié de docker-compose.yml

```
version: "3.9"
services:
  influxdb:
    image: influxdb:2.7
    networks: [backend]

  grafana:
    image: grafana/grafana:10.0.0
    networks: [backend]

  nginx:
    image: nginx:latest
```

```
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on: [influxdb, grafana]
    networks: [backend]

networks:
  backend:
    driver: bridge
```

Listing 2 – Exemple simplifié de configuration NGINX

```
http {
    server {
        listen 80;

        location /grafana/ {
            proxy_pass http://grafana:3000/;
        }

        location /influx/ {
            proxy_pass http://influxdb:8086/;
        }
    }
}
```

Dans ce schéma, l'utilisateur externe ne communique qu'avec NGINX, qui se charge de relayer la requête vers Grafana ou InfluxDB en fonction de l'URL. Ce modèle rend l'architecture plus modulable et sécurisée en cas d'ouverture vers l'extérieur.

6.4 Conclusion

L'intégration d'un reverse proxy NGINX n'est donc pas indispensable dans le périmètre actuel du projet, mais constitue une amélioration pertinente pour toute extension vers un accès distant ou multi-utilisateurs. Dans ce cas, il centraliserait la gestion des certificats TLS et renforcerait la sécurité de l'ensemble de la plateforme.

7 Automatisation du déploiement

Dans le cadre du projet, l'architecture (Mosquitto, collecteur, InfluxDB, Grafana) a été déployée via `docker-compose`. Une alternative aurait été d'automatiser ce déploiement avec un outil comme **Ansible**.

Concrètement, un playbook Ansible aurait permis de :

— installer automatiquement Docker et Docker Compose sur la machine cible ;

— créer les répertoires nécessaires et copier les fichiers de configuration (certificats TLS, `docker-compose.yml`, etc.) ;

— lancer la stack complète en une seule commande ;

— redéployer facilement l'ensemble en cas de panne ou de mise à jour.

Un exemple de playbook Ansible minimal aurait pu ressembler à ceci :

```
---
- hosts: avionics_hosts
  become: true
  tasks:
    - name: Installer Docker
      apt:
        name: docker.io
        state: present
        update_cache: yes

    - name: Installer Docker Compose
      apt:
        name: docker-compose
        state: present

    - name: Créer les répertoires pour la stack FalconEye
      file:
        path: "/opt/falconeye/{{ item }}"
        state: directory
        mode: '0755'
      loop:
        - docker/mqtt
        - docker/influxdb
        - docker/grafana
        - config/certs

    - name: Copier les fichiers de configuration
      copy:
```

```
      src: "{{ item.src }}"
      dest: "{{ item.dest }}"
      owner: root
      group: root
      mode: '0644'
    loop:
      - { src: 'docker-compose.yml', dest: '/opt/falconeye/docker-compose.yml' }
      - { src: 'config/certs/', dest: '/opt/falconeye/config/certs/' }

  - name: Lancer la stack Docker Compose
    command: docker-compose up -d
    args:
      chdir: /opt/falconeye
```

Cette approche aurait été particulièrement utile dans un contexte aéronautique, où la **reproductibilité**, la **rapidité de redéploiement** et la **réduction des erreurs humaines** sont essentielles. Bien que non implémentée ici (le code étant déjà versionné sur GitHub), elle constitue une piste d'amélioration pour professionnaliser et fiabiliser le déploiement.

8 Sécurisation des communications et des accès

La sécurité est un enjeu central dans toute architecture distribuée, en particulier lorsque des services sont exposés à l'extérieur d'un réseau local. Plusieurs mécanismes complémentaires peuvent être envisagés pour protéger l'infrastructure.

8.1 TLS sur Mosquitto

Dans ce projet, nous avons mis en place du chiffrement **TLS** sur le broker **Mosquitto**. Ce mécanisme permet de garantir la confidentialité et l'intégrité des données échangées entre les capteurs et le collecteur, en empêchant qu'un tiers puisse intercepter ou modifier les messages MQTT. L'utilisation de certificats permet également d'assurer une authentification entre les différentes entités du système.

8.2 Pare-feu pour réduire la surface d'attaque

Une autre mesure de sécurité classique consiste à mettre en place un **pare-feu** (par exemple `ufw` sur Linux) afin de restreindre les flux entrants. Dans un déploiement réel, il serait pertinent de :

— Autoriser uniquement les ports strictement nécessaires (par exemple, 8883 pour Mosquitto, 3000 pour Grafana).

— Bloquer tout autre port pour réduire la surface d'attaque.
— Limiter l'accès à certaines adresses IP de confiance lorsque c'est possible.

Dans le cadre de ce projet, cette configuration n'a pas été mise en place, car l'environnement reste expérimental et les services tournent en local. Néanmoins, dans un contexte de production, un pare-feu constitue une barrière essentielle pour limiter les risques d'intrusion.

8.3 Fail2ban pour réagir dynamiquement aux attaques

Un second mécanisme qui pourrait être ajouté est **Fail2ban**. Cet outil analyse les logs des services (par exemple les échecs de connexion à Mosquitto ou Grafana) et met automatiquement en place des règles de filtrage pour bloquer les adresses IP suspectes pendant un temps donné.

Dans une architecture réelle, Fail2ban permettrait de :

— Bloquer temporairement ou définitivement les adresses IP effectuant des tentatives de connexion répétées.
— Réduire l'impact d'attaques par force brute sur les mots de passe.
— Ajouter une couche de défense adaptative par rapport à un pare-feu statique.

8.4 Choix retenu dans ce projet

Seul le **TLS sur Mosquitto** a été mis en œuvre dans ce projet, car il représentait la mesure de sécurité la plus pertinente pour protéger la confidentialité des données transmises par les capteurs. L'ajout d'un pare-feu et d'un système comme Fail2ban aurait peu d'intérêt dans un environnement restreint et contrôlé comme celui du projet.

Cependant, dans un déploiement opérationnel (par exemple sur un serveur en ligne), la combinaison des trois mécanismes — chiffrement TLS, restriction des ports via pare-feu, et défense dynamique via Fail2ban — constituerait une approche robuste pour sécuriser l'infrastructure.

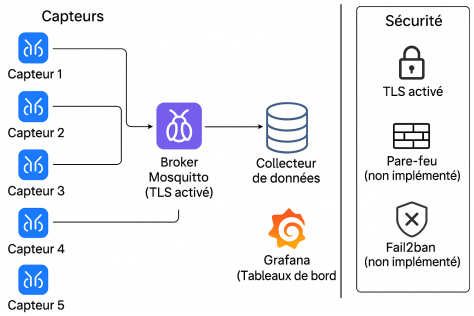

FIGURE 9 – Sécurité des communications

9 Tests et Validation

Afin de garantir la robustesse et la sécurité de l'architecture **FalconEye**, plusieurs catégories de tests ont été mises en place. Ces tests permettent de vérifier à la fois la **résilience du système face aux erreurs**, la **sécurité des communications**, ainsi que la **continuité du flux de données** entre les composants.

9.1 Test d'intégrité des messages

Un conteneur spécifique a été ajouté afin d'envoyer volontairement des messages malformés sur le broker MQTT (par exemple, en remplaçant une valeur numérique par une chaîne de caractères). Le **collecteur** est alors chargé de rejeter ces messages invalides et de consigner une erreur dans ses logs. Ce test valide la **protection contre la corruption de données** et le **respect du format attendu**.

9.2 Test d'intrusion réseau

Un conteneur de type `Kali Linux` a été intégré dans l'infrastructure pour simuler des attaques réseau. Les principaux scénarios testés sont :

- **scan de ports** (via `nmap`) afin d'identifier les services exposés ;
- **tests TLS** avec `openssl` pour vérifier la validité des certificats ;
- **tentatives de publication non autorisée** sur Mosquitto avec de faux identifiants.

Ces attaques simulées permettent de confirmer que l'infrastructure **rejette tout accès non autorisé**.

9.3 Test des certificats et ACLs

Un script dédié vérifie le bon fonctionnement de la chaîne de confiance et des règles d'accès :

1. Publication et souscription avec un certificat valide sur un topic autorisé (succès attendu).
2. Publication avec un certificat valide mais sur un topic interdit (échec attendu).
3. Connexion avec un mot de passe invalide (refus attendu).
4. Connexion sans certificat client (refus attendu).

Ces tests garantissent que la **mutual TLS**, les **ACLs** et l'authentification par mot de passe fonctionnent conformément aux exigences.

9.4 Test de connectivité réseau

Un second script automatise les tests suivants :

— vérification que tous les services (`mosquitto`, `influxdb`, `grafana`, simulateurs, etc.) sont bien reliés au réseau Docker `falconnet` ;
— tests de **ping** entre les conteneurs critiques (par ex. collecteur → broker et collecteur → InfluxDB) ;
— vérification de la **santé du service InfluxDB** via son endpoint HTTP `/health` ;
— test des **ports ouverts** sur l'hôte (8883 pour MQTT, 8086 pour InfluxDB, 3000 pour Grafana, 9001 pour WebSockets MQTT).

Ces tests confirment que l'**infrastructure est correctement interconnectée** et que les services essentiels sont joignables.

9.5 Test End-to-End

Enfin, un test complet a été mis en place : un simulateur envoie une donnée vers le broker MQTT, celle-ci est collectée par le collecteur puis stockée dans InfluxDB. Une requête HTTP interroge InfluxDB pour s'assurer que la donnée a bien été persistée. Ce test valide l'**intégrité de la chaîne de traitement des données** de bout en bout, depuis les capteurs simulés jusqu'aux dashboards Grafana.

10 Conclusion

Ce projet a permis de mettre en place une architecture complète de simulation avionique, basée sur des conteneurs Docker et orchestrée avec `docker-compose`. L'objectif principal a été atteint : comprendre et déployer une infrastructure intégrant un broker MQTT sécurisé (Mosquitto), un collecteur de données, une base de données temporelle (InfluxDB) et un outil de visualisation (Grafana).

L'aspect le plus formateur a été la **mise en place de l'architecture réseau et logicielle** ainsi que la **sécurisation des communications** (TLS, certificats, ACLs). Le code des simulateurs a constitué une partie secondaire : l'essentiel du travail résidait dans la configuration, l'apprentissage de Docker, et la structuration du projet (organisation des dossiers, scripts d'automatisation).

10.1 Améliorations possibles

Plusieurs pistes d'amélioration ont été identifiées :

— **Documentation** : enrichir la documentation utilisateur et développeur, afin de faciliter la prise en main du projet et son déploiement.
— **Tests** : compléter les tests d'intégrité et de sécurité existants par des tests unitaires sur les simulateurs. L'usage de `pytest` permettrait par exemple de vérifier automatiquement la validité des messages générés (types, unités, plages de valeurs).
— **Intégration continue** : mettre en place une branche dédiée aux tests automatisés,

exécutés via GitHub Actions. Cette approche garantirait la fiabilité de l'infrastructure et éviterait l'introduction de régressions.

10.2 Dashboards Grafana

Un élément central du projet est la création de tableaux de bord dans Grafana, permettant de visualiser en temps réel les données issues des capteurs simulés : navigation, commandes de vol, hydraulique, moteur et structure. Ces dashboards apportent une vue d'ensemble claire et dynamique du fonctionnement de l'avionique.

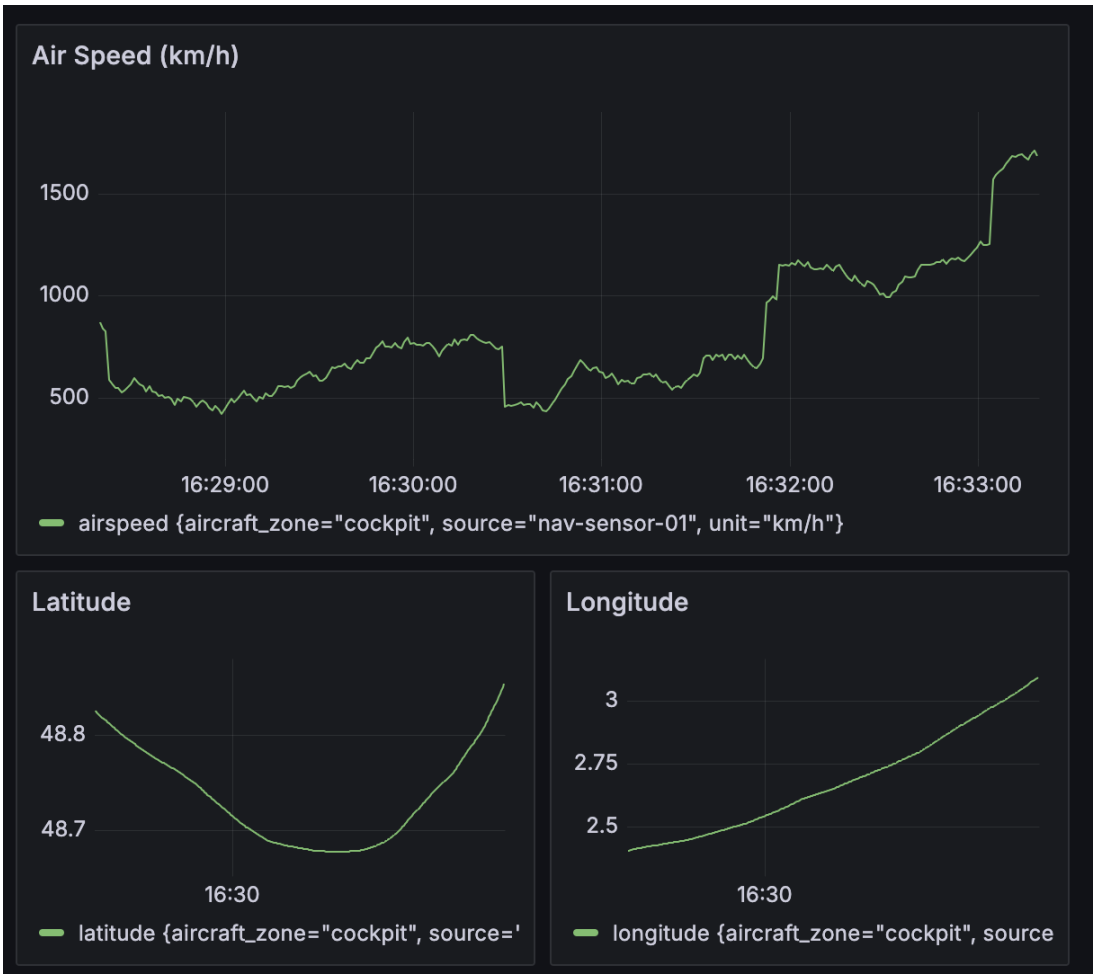

FIGURE 10 – Exemple de dashboard Grafana pour la surveillance moteur.

En conclusion, ce projet a permis d'acquérir une expérience complète dans la mise en place d'une architecture distribuée, sécurisée et visualisable, en partant de zéro. Il constitue une base solide pouvant être étendue par des fonctionnalités de tests automatisés, une documentation enrichie et une intégration continue pour tendre vers un environnement encore plus professionnel.