



Dossier Technique

Version 1.0

Foucault Clémentine, Mhadi Zakariya

Saint Jean-Baptiste de Lasalle

Rue Notre Dame des 7 Douleurs

84000 Avignon

Sommaire

Sommaire	1
Présentation générale	3
Expression des besoins	3
Les cas d'utilisation	6
Diagramme de déploiement	7
MQTT (Message Queuing Telemetry Transport)	8
Partie application mobile (Foucault Clémentine)	9
Informations	9
Présentation du système	10
Diagramme de cas d'utilisation	11
Diagrammes de séquences	12
Démarrer l'application	12
Recevoir les données	13
Traiter les données d'environnement	14
Diagramme de classes	15
Description des classes	16
Description classe IHMMobile	17
Description classe Communication	18
Description classe MesureRuche	19
Description classe Ruche	20
Description classe StockageRuche	21
Description de la classe Alertes	22
Description de la classe Historique	24
Description de la classe IHMGraphique	24
Maquette IHM	25
IHM v1.0	26
Fenêtre de paramétrage des seuils d'alertes	27
Fenêtre d'affichage pour ajouter une ruche	28
Fenêtre d'affichage pour la suppression	29
Communication MQTT	32
Création d'un objet client MQTT	32
Connexion au serveur TTN	32
Installation des fonctions de rappel	32
Décodage de trame en JSON (Java Script Objet Nation)	33

Stockage des ruches	35
Partie physique	36
Tests de validation	37
Partie application pc (Mhadi Zakariya)	38
Introduction	38
Présentation de l'IHM	39
Planification	40
Diagramme de GANTT	40
Diagramme de cas d'utilisation	41
Diagramme de séquence et code	42
Démarrage de l'IHM	42
Consulter les données sur l'IHM	47
JSON	52
Diagramme de classes	54
Diagramme de la classe Ruche	56
Diagramme de la classe MesureRuche	59
Diagramme de la classe Communication	64
Diagramme de la classe IHMPc	70
Maquette IHM	77
Tests de validation	78
Informations	79
GLOSSAIRE	80

Présentation générale

Le projet **Bee-Honey't** est un concept de ruche connectée. A partir d'une application, l'utilisateur pourra obtenir les données (température intérieure/extérieure, humidité intérieure/extérieure, pression, poids, ensoleillement et la charge de la batterie) d'une ruche. L'utilisateur pourra aussi gérer le paramétrage des alertes et visualiser des graphiques à partir des relevés de mesure sur sept jours d'une ruche.

Expression des besoins

La ruche connectée devra effectuer les tâches suivantes :

- L'envoi à intervalles réguliers (15 min) des mesures effectuées suivantes :
 - Température intérieure et extérieure,
 - Humidité relative intérieure et extérieure,
 - Pression atmosphérique,
 - Poids de la ruche,
 - Ensoleillement,
 - Niveau de charge, tension et courant de la batterie
- L'affichage des mesures et des alertes en temps réel
- L'affichage sous forme de vues graphiques des mesures (récapitulatifs journaliers : moyennes, min, max pour chaque heure sur les 7 derniers jours) et éventuellement l'historique des alertes
- L'alerte en cas de variation brutale d'une grandeur mesurée (perte de poids soudaine) ou de dépassement de seuils. L'alerte pourra être signalée sous la forme d'un *email* ou d'un message SMS envoyé sur le *smartphone* de l'apiculteur.

Contraintes :

- Le système ne doit pas perturber les abeilles. Une attention particulière doit être portée aux technologies employées, aux ondes et aux fréquences utilisées.
- Le système ne doit pas entraver le travail de l'apiculteur. Les capteurs doivent pouvoir être déconnectés simplement.
- Le système doit être le plus longtemps possible autonome en énergie afin de pouvoir être installé dans un endroit isolé. (autonomie 15 jours sans soleil)

Le développement du système doit répondre aux exigences des utilisateurs :

- ❑ simplicité d'utilisation,
- ❑ correspondre aux contraintes définies,
- ❑ réalisable dans un délai de 200 heures (IR) et 170 heures (EC).

Répartition des tâches

Etudiant 1 (EC) : PERRIN Edgar

- Mesurer les grandeurs température, humidité, pression atmosphérique, et ensoleillement de la ruche (option)
- Mesurer le poids de la ruche
- Transmettre les données au PC

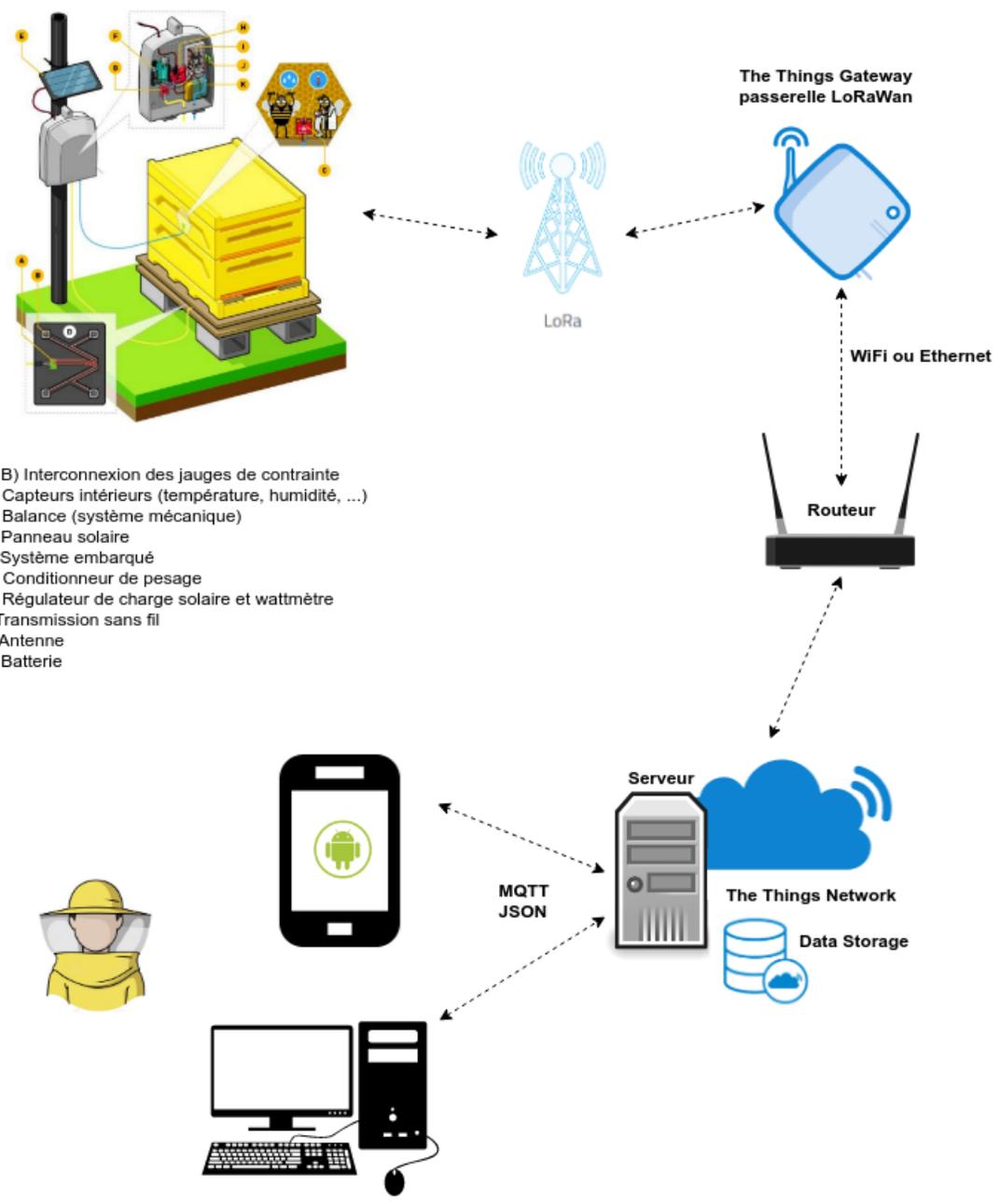
Etudiant 2 (IR) : Version Android

- Consulter les données d'une ruche (vue "temps réel" + description + affichage des alertes)
- Recevoir les données actuelles des ruches (MQTT/Json)
- Récupérer les données enregistrées (HTTP/Json)
- Éditer les ruches
- Paramétrer une nouvelle ruche et les alertes
- Option : Déclencher les alertes (par email)

Etudiant 3 (IR) : Version PC

- Consulter les données d'une ruche (vue "temps réel" + description + affichage des alertes)
- Recevoir les données actuelles des ruches (MQTT/Json)
- Récupérer les données enregistrées (HTTP/Json)
- Éditer les ruches
- Paramétrer une nouvelle ruche et les alertes
- Option : Déclencher les alertes (par email)

Présentation du système



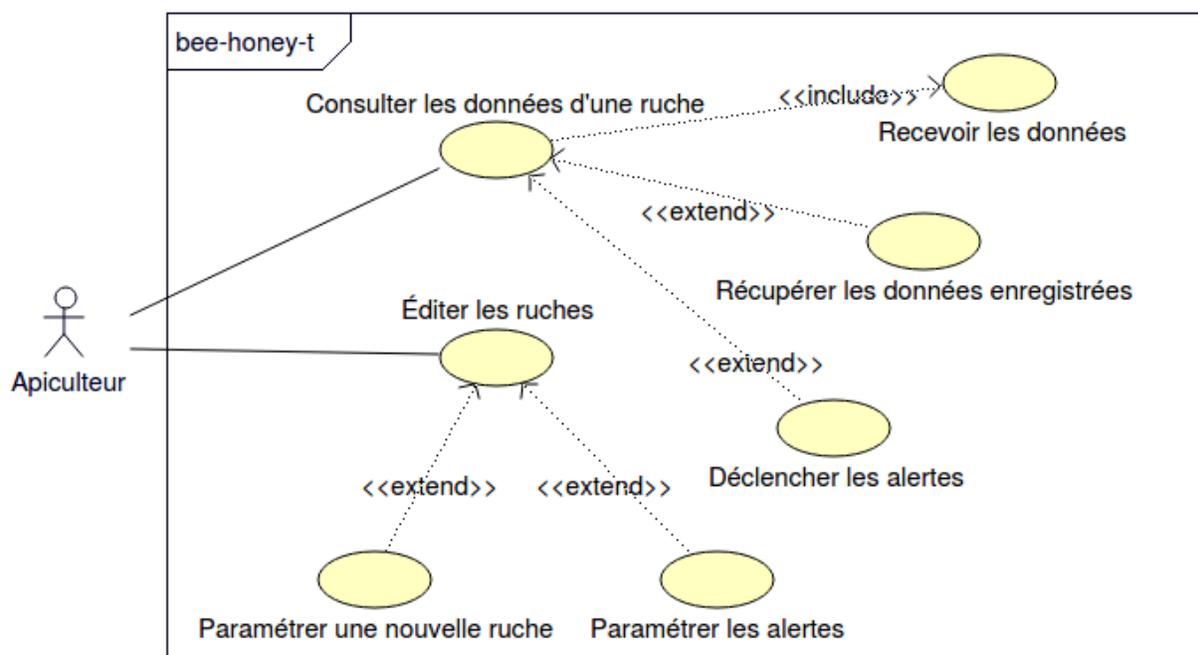
- A) B) Interconnexion des jauges de contrainte
- C) Capteurs intérieurs (température, humidité, ...)
- D) Balance (système mécanique)
- E) Panneau solaire
- F) Système embarqué
- G) Conditionneur de pesage
- H) Régulateur de charge solaire et wattmètre
- I) Transmission sans fil
- J) Antenne
- K) Batterie

Les cas d'utilisation

L'apiculteur peut éditer ses ruches (ajouter, paramétrer ou supprimer une ruche). Les paramètres d'une ruche sont un nom, une localisation, une description optionnelle et une date de mise en service. On lui associe un DeviceID (identifiant de la carte embarquée *The Things Uno*) et un ApplicationID (représentant l'ensemble des ruches gérées par *The Things Network*¹). L'ensemble des paramètres des ruches seront enregistrés localement dans l'application (base de données SQLite par exemple).

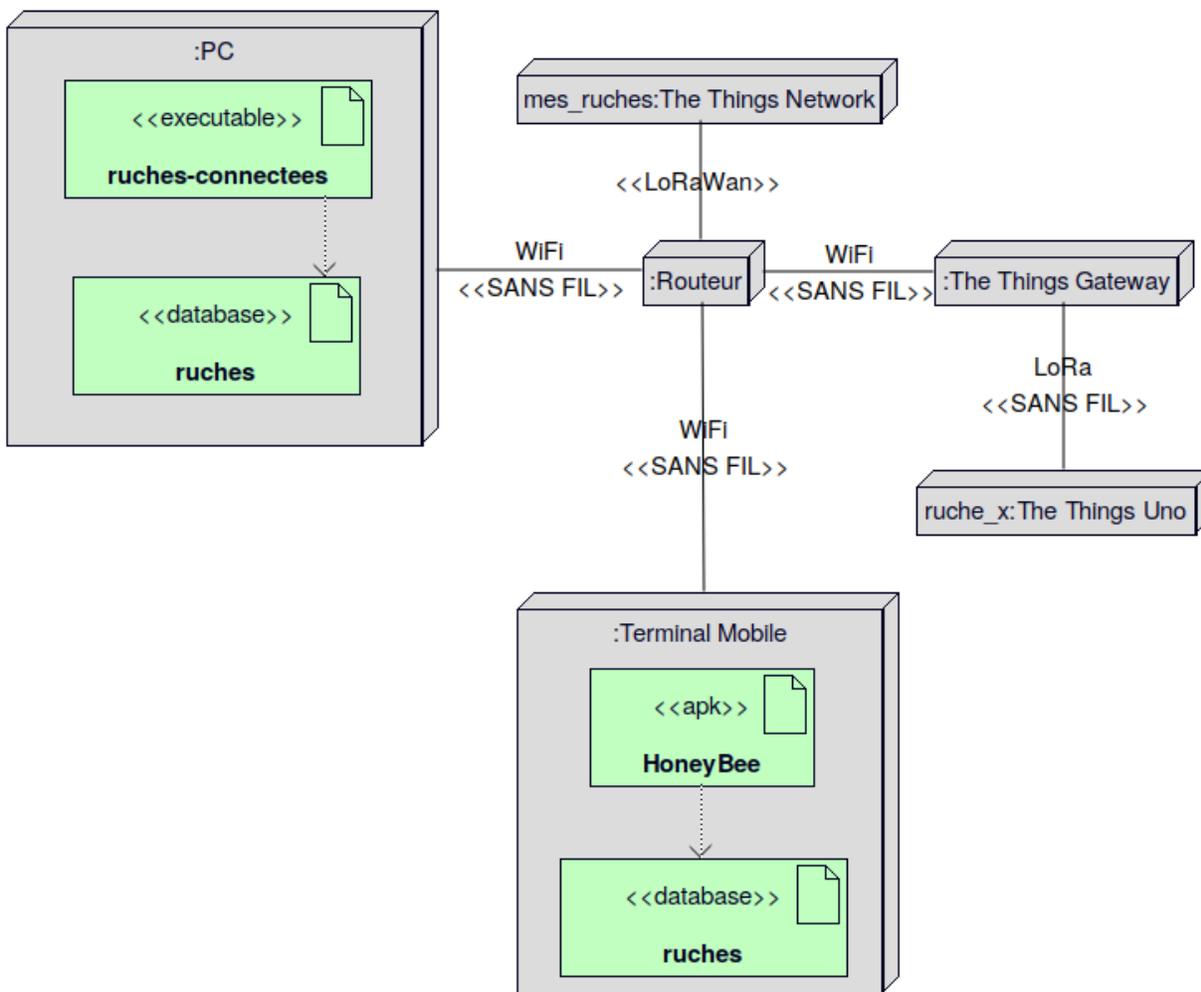
L'apiculteur peut paramétrer les seuils d'alerte (humidité, température et poids) d'une ruche et le type de notification (visuel/email/sms) et le destinataire de l'alerte si besoin. L'apiculteur reçoit une alerte par email et/ou SMS ou tout simplement sur son interface lorsqu'une condition anormale est détectée.

Il peut choisir une ruche pour consulter les données actuelles et/ou enregistrées. Les données sont reçues au format JSON par le protocole MQTT via le réseau *The Things Network*. *The Things Network* propose aussi un service *Data Storage* assurant la sauvegarde de données pour les 7 derniers. Les données enregistrées reçues via le protocole HTTP sont au format JSON.



¹ The Things Network est un réseau LoRaWAN open source qui est disponible dans 86 pays, basé sur une communauté de plus de 17 000 membres. Il peut être utilisé sans contrainte commerciale ou privée.

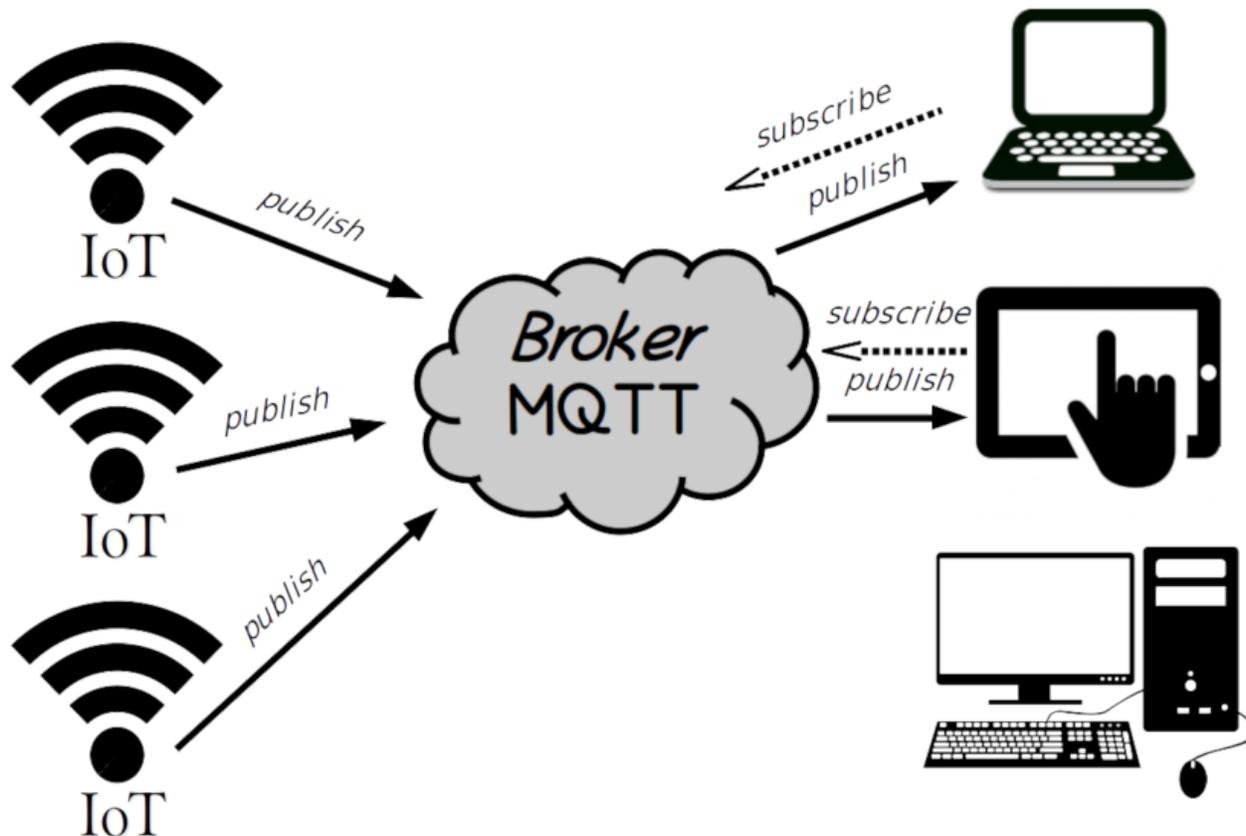
Diagramme de déploiement



MQTT (Message Queuing Telemetry Transport)

MQTT est un protocole de messagerie publish-subscribe basé sur le protocole TCP/IP. [cf. fr.wikipedia.org/wiki/MQTT]

Tous les acteurs doivent se connecter au broker MQTT (un serveur) pour communiquer entre eux en envoyant des messages (publish) où ils se sont abonnés (subscribe) auparavant à des sujets (topic) qui les intéressent.



MQTT a été utilisé lors du projet pour recevoir les trames de données envoyées par les objets connectés des ruches.

Partie application mobile (Foucault Clémentine)

Informations

Auteur	Foucault Clémentine : foucaultclementine@gmail.com
Date	2021
Version	1.0
SVN	https://svn.riouxsvn.com/beeoneyt-2021

Présentation du système

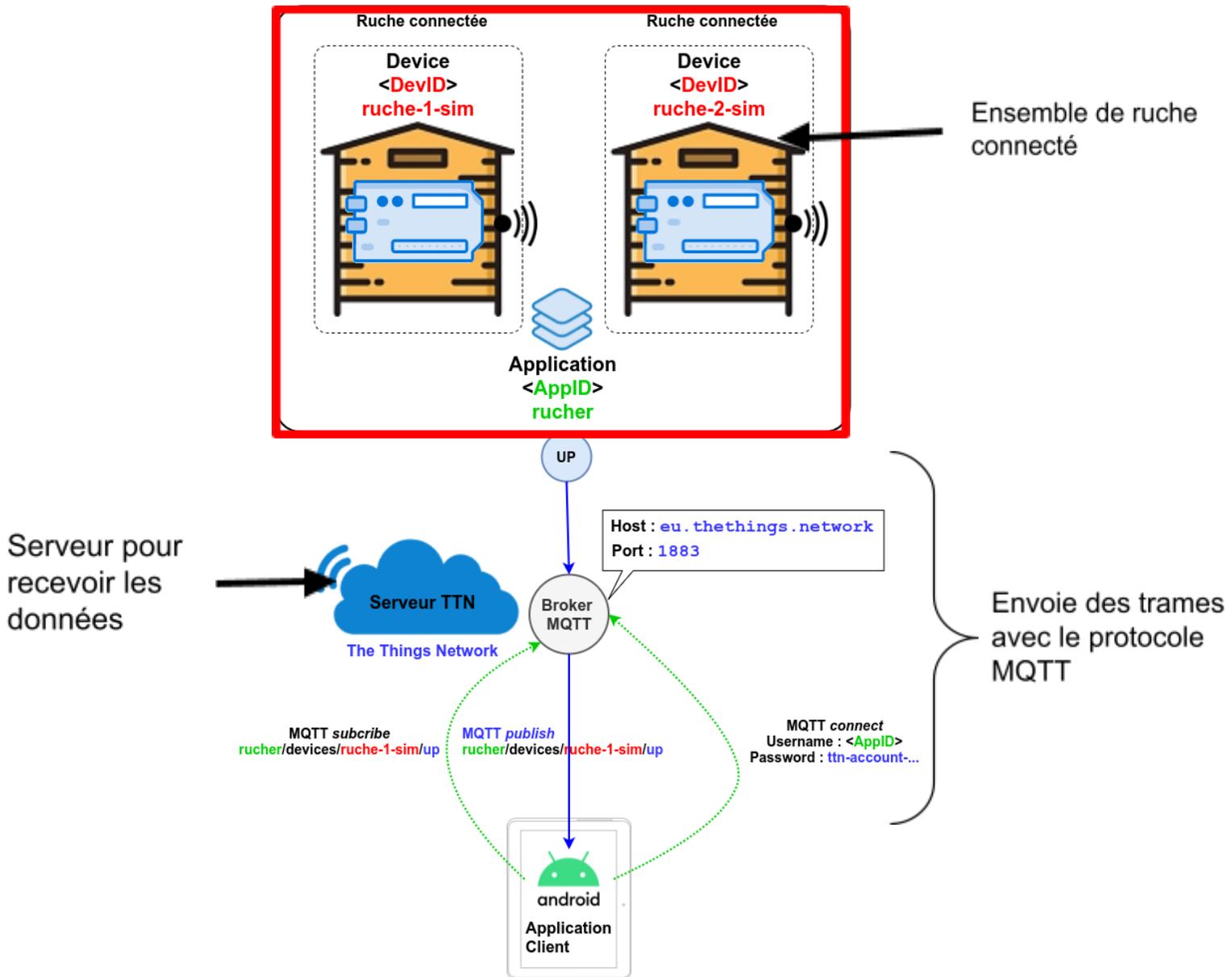
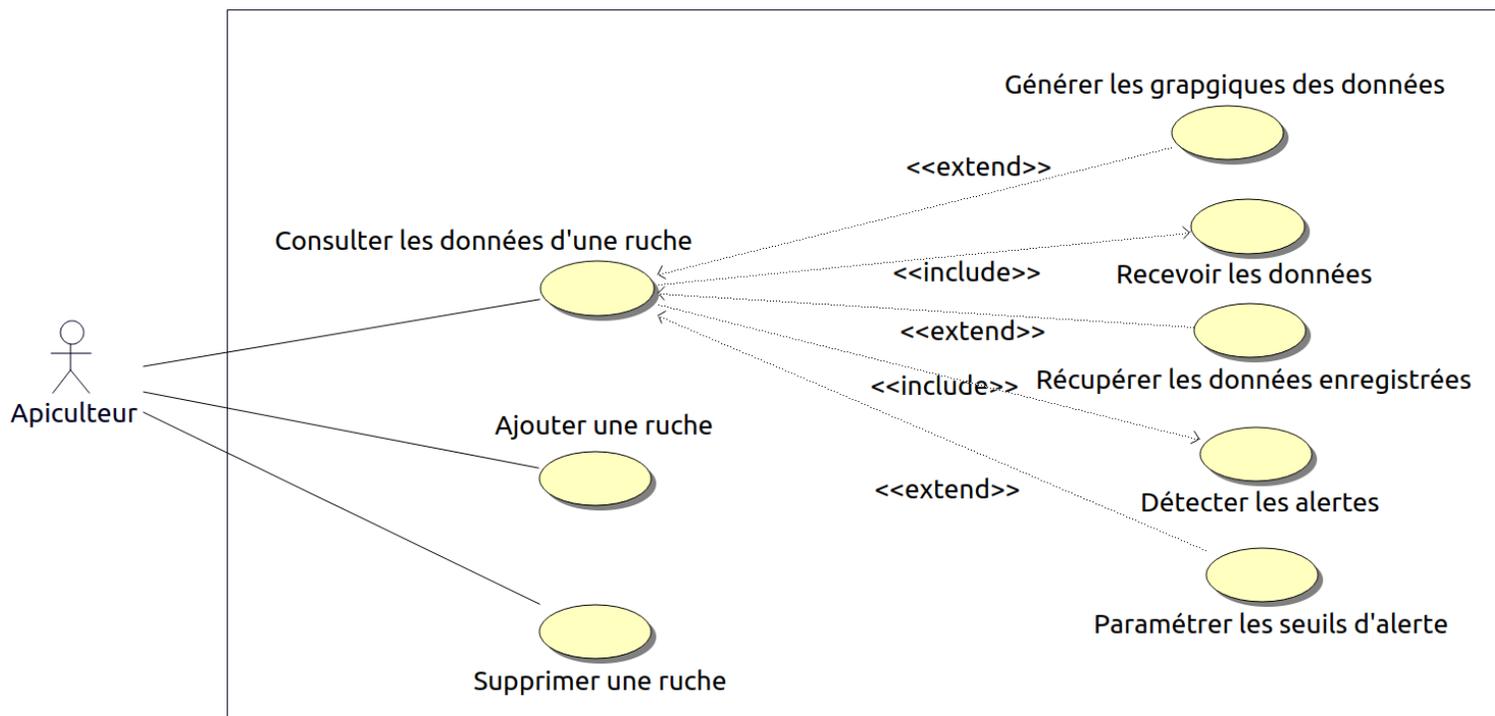


Diagramme de cas d'utilisation

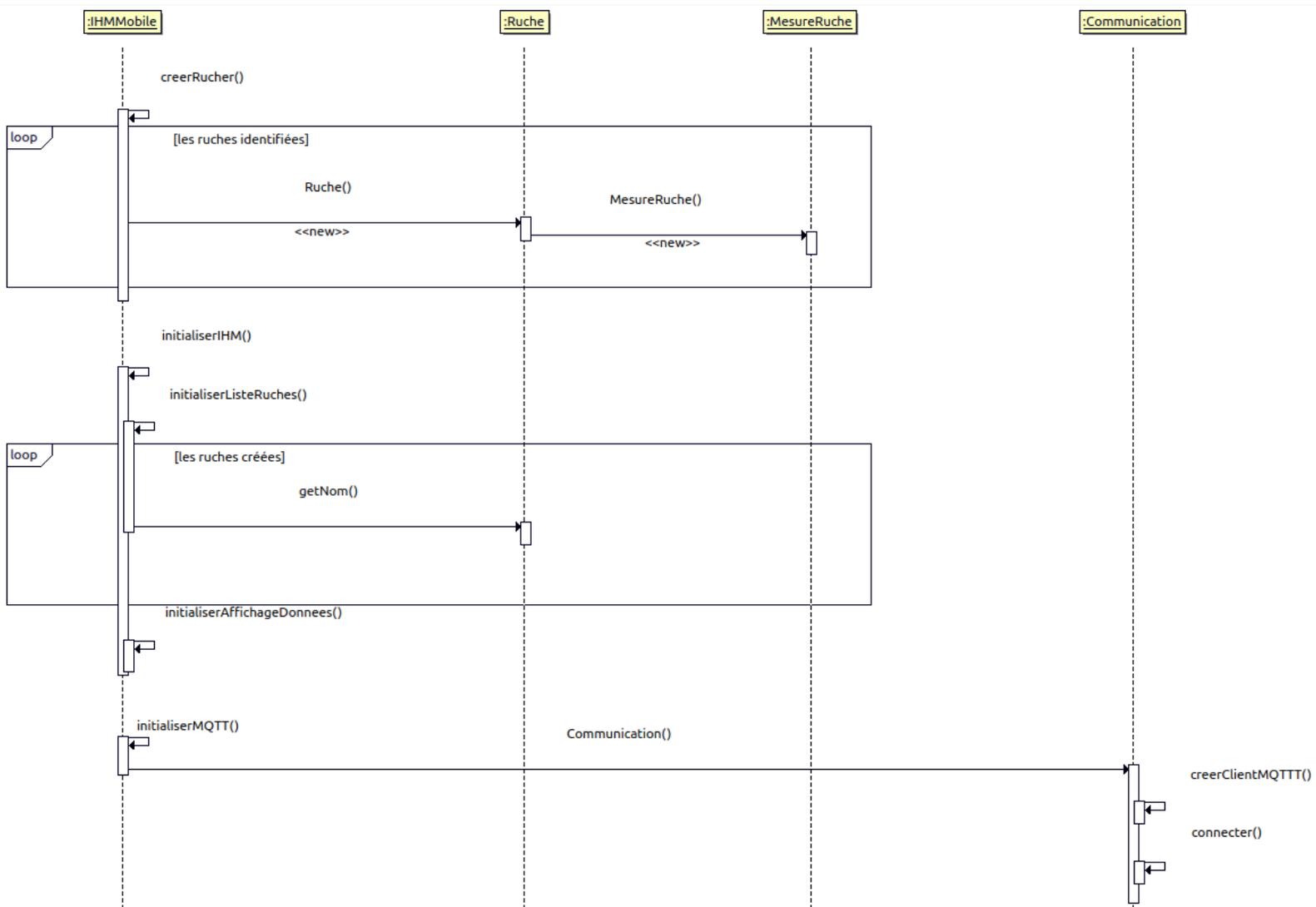
- L'apiculteur peut consulter les données d'une ruche : température intérieure/extérieure, humidité intérieure/extérieure, pression, poids, ensoleillement et la charge de la batterie.
- L'apiculteur peut éditer des ruches : les ajouter ou les supprimer.
- L'apiculteur peut sélectionner une ruche dans une liste déroulante
- L'apiculteur peut visualiser les alertes sur l'IHM
- L'application sauvegarde les ruches



Les données sont reçues au format JSON par le protocole MQTT via le réseau *The Things Network*. *The Things Network* propose aussi un service *Data Storage* assurant la sauvegarde de données pour les 7 derniers.

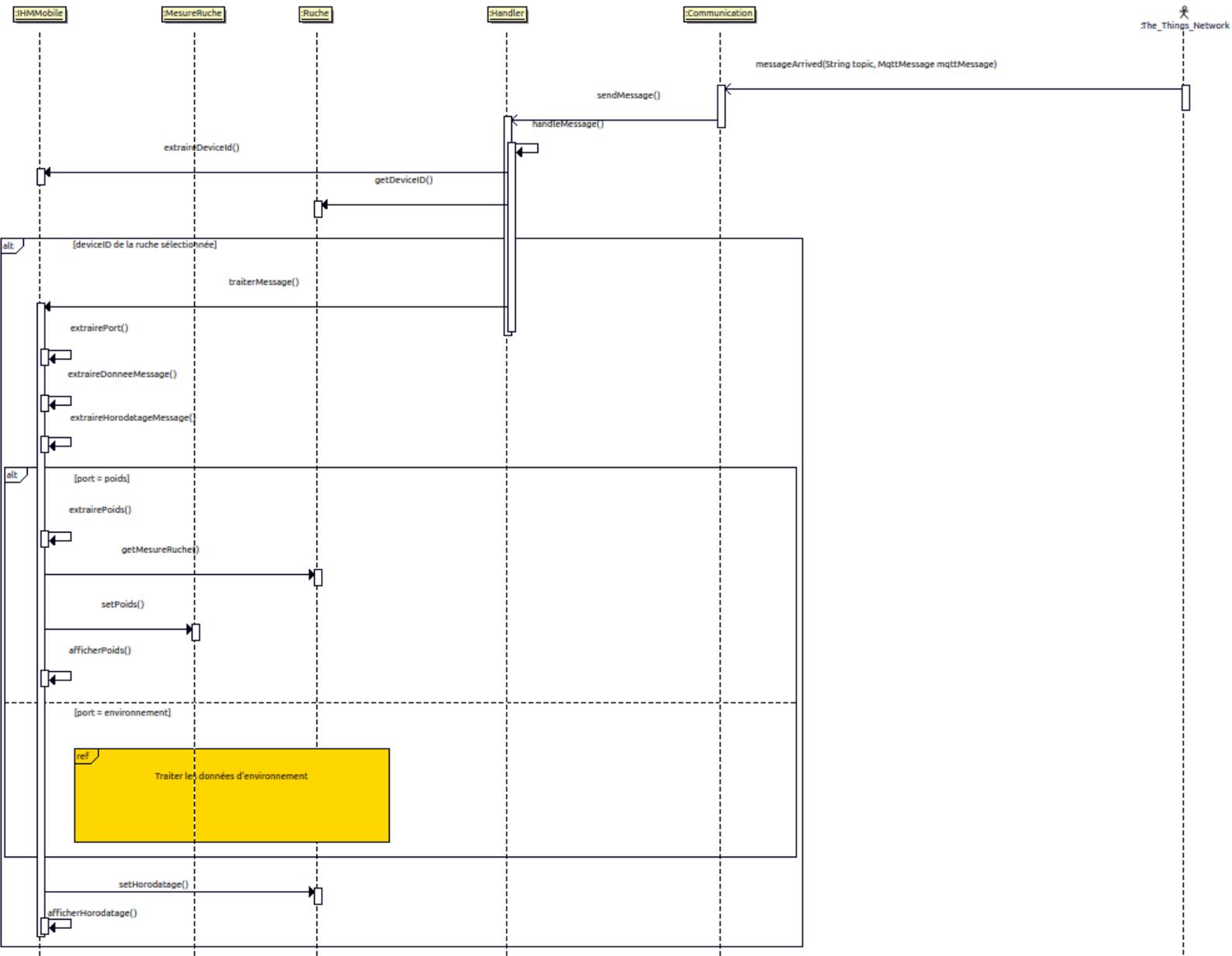
Diagrammes de séquences

Démarrer l'application



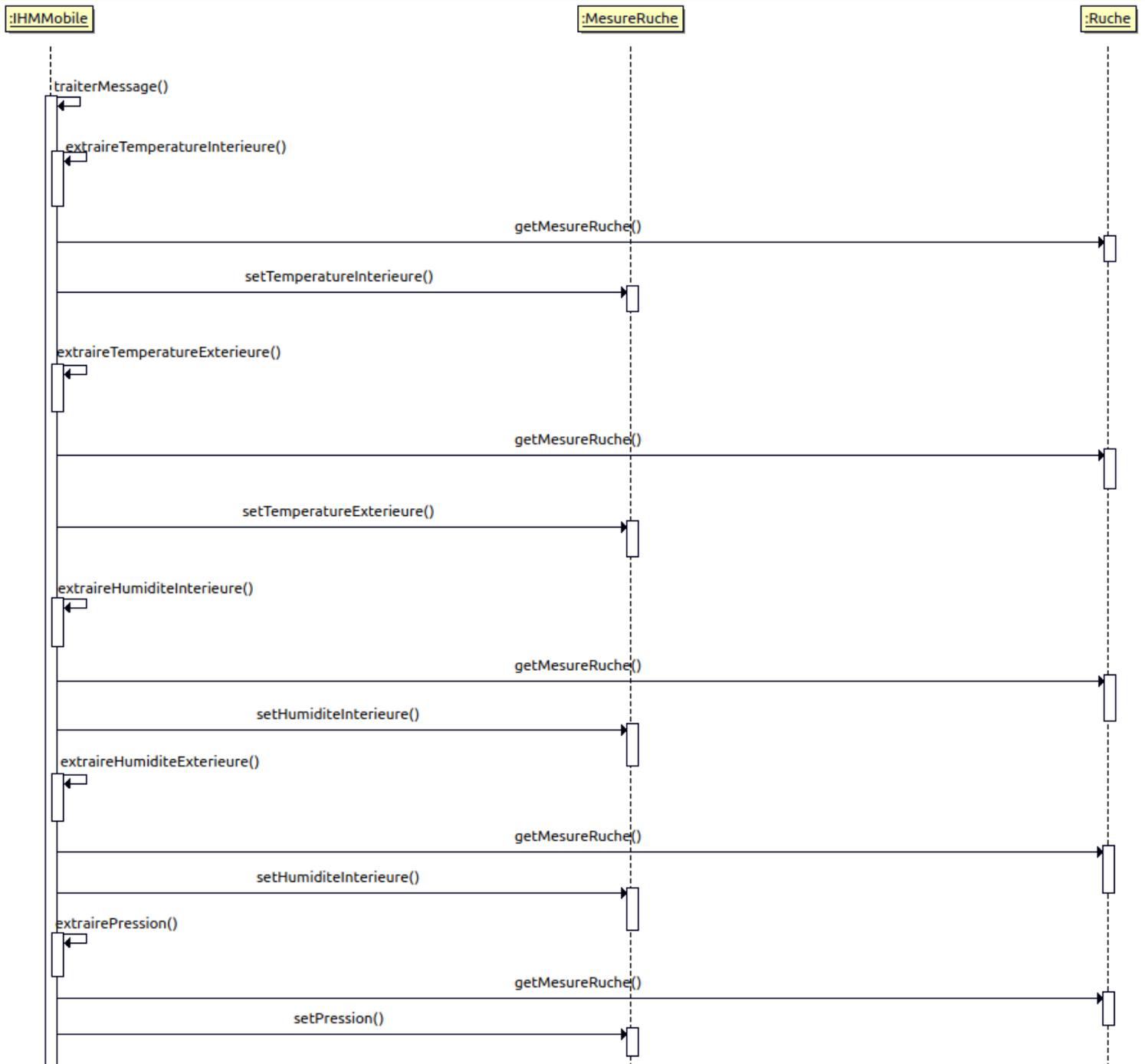
Ce schéma nous explique comment l'application démarre.

Recevoir les données



Ce schéma nous explique comment recevoir les données.

Traiter les données d'environnement



Ce schéma nous explique comment est géré le traitement des données de l'environnement.

Diagramme de classes



Diagramme de classes complet :

- IHMMobile
- Ruche
- MesureRuche
- Communication
- Historique
- IHMGraphique
- Alertes
- AppCompatActivity
- StockageRuche

Description des classes

Classe	Description
IHMMobile	Permet d'afficher les valeurs dans l'IHM(Interface Homme Machine).
MesurerRuche	Contient les mesures d'une ruche sélectionnée.
Ruche	Contient les informations d'une ruche.
Communication	permet la communication MQTT avec le serveur The Things Network.
Historique	Permet la récupération de l'historique Data Storage TTN sur 7 jours max
IHMGraphique	Permet d'afficher les graphiques
Alertes	Permet de gérer les alertes
AppCompatActivity	Classe mère des activités
StockageRuche	Permet de stocker les ruches et le paramétrage des alertes

- La classe en gras les activités de l'application et en rouge la classe principale

Description classe IHMMobile

IHMMobile
- TAG
- EXTRAIRE_DONNEE_POIDS
- EXTRAIRE_DONNEES_ENVIRONNEMENT
- APPLICATION_ID
- CHAMP_TOPIC_APPLICATION_ID
- CHAMP_TOPIC_DEVICE_ID
- indexRucheSelectionnee
- adapterRuche
- reponseNom
- reponseDeviceID
- reponseTemperatureInterieureMax
- reponseTemperatureInterieureMin
- reponseTemperatureExterieurMax
- reponseTemperatureExterieurMin
- reponseHumiditeInterieureMax
- reponseHumiditeInterieureMin
- reponseHumiditeExterieurMax
- reponseHumiditeExterieurMin
- reponsePoidsMax
- reponsePoidsMin
- listeRuches
- afficheTemperatureInterieur
- afficheTemperatureExterieur
- afficheHumiditeInterieur
- afficheHumiditeExterieur
- affichePoids
- affichePression
- afficheHorodatage
- afficheConnexion
- afficheDeconnexion
- listeNomsRuches
- alerteTemperatureInterieure
- alerteTemperatureExterieur
- alerteHumiditeInterieure
- alerteHumiditeExterieur
- alertePoids
- handler
onCreate()
- initialiserStockage()
- initialiserMQTT()
- initialiserHistorique()
- creerRucher()
onStart()
onResume()
onPause()
onStop()
onDestroy()
- initialiserIHM()
- initialiserAffichageDonnees()
- remettreAZeroAlertes()
- initialiserListeRuches()
+ afficherTemperatureInterieure()
+ afficherTemperatureExterieur()
+ afficherHumiditeInterieure()
+ afficherHumiditeExterieur()
+ afficherPoids()
+ afficherPression()
+ afficherHorodatage()
+ afficherConnexion()
+ afficherDeconnexion()
- traiterMessage()
- afficherAlertesEnvironnement()
- extrairePression()
- extraireHumiditeExterieur()
- extraireHumiditeInterieure()
- extraireTemperatureExterieur()
- extraireTemperatureInterieure()
- extrairePort()
- abonnerRuches()
- desAbonnerRuches()
- decoderTopic()
- extraireDonneeMessage()
- extraireHorodatageMessage()
+ extraireHorodatage()
+ extraireDeviceID()
- extrairePoids()
+ afficherFenetreSupprimer()
+ afficherFenetreAjouter()
+ ajouterRuche()
+ supprimerRuche()
+ afficherFenetreParametrageAlertes()
+ afficherGraphiques()
+ ajouterParametreAlerte()
+ afficherAlerteTemperatureInterieure()
+ afficherAlerteTemperatureExterieur()
+ afficherAlerteHumiditeInterieure()
+ afficherAlerteHumiditeExterieur()
+ afficherAlertePoids()

Attributs

Méthodes

Description classe Communication

Communication
<ul style="list-style-type: none"> - TAG : String - mqttAndroidClient : MqttAndroidClient - handler : Handler + TTN_CONNECTE : int + TTN_DECONNECTE : int + TTN_MESSAGE : int - serverUri : String - clientId : String - username : String - password : String
<ul style="list-style-type: none"> + Communication(inout context : Context, in handler : Handler) + getUsername() : String + getPassword() : String - creerClientMQTT(inout context : Context, inout handler : Handler) : void + setCallback(inout callback : MqttCallbackExtended) : void - connecter() : void + reconnecter() : void + deconnecter() : void + estConnecte() : boolean + souscrireTopic(in deviceId : String) : boolean + unsubscribe(in deviceId : String) : boolean

Les attributs de la classe communication servent à la communication MQTT avec le serveur The Things Network (TTN).

- `Communication ()` est notre constructeur
 - `creerClientMQTT()` sert a creer un client MQTT
- les méthodes `connecter()`, `reconnecter()`, `deconnecter()` et `estConnecter()` sont les méthode pour gérer la connexion au serveur TTN.
- `souscrireTopic()` et `unsubscribe()` sont des méthodes pour s'abonner ou se désabonner des topic.
 - `getUsername()`,`getPassword()` sont les accesseurs des attributs username et password.

Description classe MesureRuche

MesureRuche
<ul style="list-style-type: none"> - TAG : String - temperatureInterieure : double - temperatureExterieur : double - humiditeInterieure : int - humiditeExterieur : int - pression : int - poids : double
<ul style="list-style-type: none"> + MesureRuche() + MesureRuche(in temperatureInterieure : double, in temperatureExterieur : double, in humiditeInterieure : int, in humiditeExterieur : int, in pression : int, in poids : double) + getTemperatureInterieure() : double + setTemperatureInterieure(in temperatureInterieure : double) : void + getTemperatureExterieur() : double + setTemperatureExterieur(in temperatureExterieur : double) : void + getHumiditeInterieure() : int + setHumiditeInterieure(in humiditeInterieure : int) : void + getHumiditeExterieur() : int + setHumiditeExterieur(in humiditeExterieur : int) : void + getPression() : int + setPression(in pression : int) : void + getPoids() : double + setPoids(in poids : double) : void

Les attributs :

- temperatureInterieure
- temperatureExterieur
- humiditeInterieure
- humiditeExterieur
- pression
- poids

Sont les variables pour stocker les mesures reçues.

- MesureRuche() est le constructeur par défaut
- MesureRuche(double temperatureInterieure, double temperatureExterieur, int humiditeInterieure, int humiditeExterieur, int pression, double poids) est le constructeur de la classe.

Et par la suite, il y a tous les accesseurs et mutateurs des attributs.

Description classe Ruche

Ruche
<ul style="list-style-type: none"> - TAG : String - nom : String - deviceID : String - horodatage : String - longitude : double - latitude : double
<ul style="list-style-type: none"> + Ruche(in nom : String, in deviceID : String) + Ruche(in nom : String, in deviceID : String, in alertesJSON : String) + getNom() : String + setNom(in nouveauNom : String) : void + getDeviceID() : String + setDeviceID(in deviceID : String) : void + getHorodatage() : String + setHorodatage(in horodatage : String) : void + getMesureRuche() : MesureRuche + setMesureRuche(inout mesureRuche : MesureRuche) : void + getAlerteRuche() : Alertes + setAlerteRuche(inout alerteRuche : Alertes) : void + getLongitude() : double + setLongitude(in longitude : double) : void + getLatitude() : double + setLatitude(in latitude : double) : void

Les attributs :

- nom
- deviceID
- horodatage
- longitude
- latitude

Sont les variables pour stocker le nom, le deviceID pour reconnaître une ruche et l'horodatage ainsi que la longitude et latitude.

- `Ruche()` est le constructeur par défaut
- `Ruche(String nom, String deviceID, String alertesJSON)` est le constructeur

Et par la suite, il y a tous les accesseurs et mutateurs des attributs.

Description classe StockageRuche

StockageRuche
- TAG : String
- stockage : SharedPreferences
+ StockageRuche(inout ihmMobile : IHMMobile)
+ obtenir(in cle : String) : String
+ obtenirRuches() : Ruche
+ editerRuche(in nom : String, in deviceID : String) : void
+ supprimerRuche(in deviceID : String) : void
+ contient(in cle : String) : boolean
+ obtenirNombreRuches() : int
+ editerAlertes(in nomRuche : String, in alertesJSON : String) : void
+ supprimerAlertes(in nomRuche : String) : void

L'attribut :

- stockage est un objet SharedPreferences qui nous permettrait de stocker les ruches dans un fichier txt

Les méthodes :

- `StockageRuche()` est le constructeur de la classe
- `obtenir()` est une méthode pour obtenir toute les données du stockage
- `obtenirRuches()` est une méthode pour obtenir les ruches
- `editerRuche()` est une méthode pour éditer une ruche
- `supprimerRuche()` est une méthode pour supprimer une ruche
- `contient()` est une méthode pour savoir ce que contient le stockage
- `obtenirNombreRuches()` est une méthode pour obtenir le nombre de ruche dans le stockage

Description de la classe Alertes

Alertes
<pre> TAG : String + TEMPERATURE_INTERIEURE_MAX : double + TEMPERATURE_INTERIEURE_MIN : double + TEMPERATURE_EXTERIEURE_MIN : double + TEMPERATURE_EXTERIEURE_MAX : double + HUMIDITE_INTERIEURE_MIN : int + HUMIDITE_INTERIEURE_MAX : int + HUMIDITE_EXTERIEURE_MIN : int + HUMIDITE_EXTERIEURE_MAX : int + POIDS_MAX : double + POIDS_MIN : double - temperatureInterieureMin : double - temperatureInterieureMax : double - temperatureExterieurMin : double - temperatureExterieurMax : double - humiditeInterieureMin : int - humiditeInterieureMax : int - humiditeExterieurMin : int - humiditeExterieurMax : int - poidsMin : double - poidsMax : double + Alertes() + Alertes(in temperatureInterieureMin : double, in temperatureInterieureMax : double, in temperatureExterieurMin : double, in temperatureExterieurMax : double, in humiditeInterieureMin : int, in humiditeInterieureMax : int, in humiditeExterieurMin : int, in humiditeExterieurMax : int, in poidsMin : double, in poidsMax : double) + Alertes(in fromJSON : String) + getTemperatureInterieurMin() : double + setTemperatureInterieurMin(in temperatureInterieurMin : double) : void + getTemperatureInterieurMax() : double + setTemperatureInterieurMax(in temperatureInterieurMax : double) : void + getTemperatureExterieurMin() : double + setTemperatureExterieurMin(in temperatureExterieurMin : double) : void + getTemperatureExterieurMax() : double + setTemperatureExterieurMax(in temperatureExterieurMax : double) : void + getHumiditeInterieurMin() : int + setHumiditeInterieurMin(in humiditeInterieurMin : int) : void + getHumiditeInterieurMax() : int + setHumiditeInterieurMax(in humiditeInterieurMax : int) : void + getHumiditeExterieurMin() : int + setHumiditeExterieurMin(in humiditeExterieurMin : int) : void + getHumiditeExterieurMax() : int + setHumiditeExterieurMax(in humiditeExterieurMax : int) : void + getPoidsMin() : double + setPoidsMin(in poidsMin : double) : void + getPoidsMax() : double + setPoidsMax(in poidsMax : double) : void + toJSON() : String </pre>

Les attributs :

Les attributs en majuscule sont des constantes, ce sont les seuils par défaut pour la gestion des alertes :

```

+ TEMPERATURE_INTERIEURE_MAX : double
+ TEMPERATURE_INTERIEURE_MIN : double
+ TEMPERATURE_EXTERIEURE_MIN : double
+ TEMPERATURE_EXTERIEURE_MAX : double
+ HUMIDITE_INTERIEURE_MIN : int
+ HUMIDITE_INTERIEURE_MAX : int
+ HUMIDITE_EXTERIEURE_MIN : int
+ HUMIDITE_EXTERIEURE_MAX : int
+ POIDS_MAX : double
+ POIDS_MIN : double

```

Les autres attributs sont les variables pour les seuils définis par l'utilisateur :

```

- temperatureInterieureMin : double
- temperatureInterieureMax : double
- temperatureExterieurMin : double
- temperatureExterieurMax : double
- humiditeInterieureMin : int
- humiditeInterieureMax : int
- humiditeExterieurMin : int
- humiditeExterieurMax : int
- poidsMin : double
- poidsMax : double

```

Il y a les constructeurs de la classe :

- `Alertes();`
- `Alertes(double temperatureInterieureMin, double temperatureInterieureMax, double temperatureExterieurMin, double temperatureExterieurMax, int humiditeInterieureMin, int humiditeInterieureMax, int humiditeExterieurMin, int humiditeExterieurMax, double poidsMin, double poidsMax);`
- `Alertes(String fromJSON)`

Les mutateurs et accesseurs des attributs :

```
+ getTemperatureInterieurMin() : double
+ setTemperatureInterieurMin(in temperatureInterieurMin : double) : void
+ getTemperatureInterieurMax() : double
+ setTemperatureInterieurMax(in temperatureInterieurMax : double) : void
+ getTemperatureExterieurMin() : double
+ setTemperatureExterieurMin(in temperatureExterieurMin : double) : void
+ getTemperatureExterieurMax() : double
+ setTemperatureExterieurMax(in temperatureExterieurMax : double) : void
+ getHumiditeInterieurMin() : int
+ setHumiditeInterieurMin(in humiditeInterieureMin : int) : void
+ getHumiditeInterieurMax() : int
+ setHumiditeInterieurMax(in humiditeInterieureMax : int) : void
+ getHumiditeExterieurMin() : int
+ setHumiditeExterieurMin(in humiditeExterieurMin : int) : void
+ getHumiditeExterieurMax() : int
+ setHumiditeExterieurMax(in humiditeExterieurMax : int) : void
+ getPoidsMin() : double
+ setPoidsMin(in poidsMin : double) : void
+ getPoidsMax() : double
+ setPoidsMax(in poidsMax : double) : void
```

Description de la classe Historique

Historique	
-	TAG : String
-	chargement : boolean
+	Historique()
+	setCallback(inout callback : HistoriqueEventListener) : void
+	estCharge() : boolean
+	charger(in strUrl : String, in key : String, in duree : String) : void

Cette classe permet la récupération de l'historique Data Storage TTN sur 7 jours max

Description de la classe IHMGraphique

<<activity>> IHMGraphique	
-	TAG : String
-	MAX_POIDS : int
~	contenu : String
-	graphique : GraphView
-	seriesPoids : DataPoint
-	boutonTempInt : Button
-	boutonTempExt : Button
-	boutonHumInt : Button
-	boutonHumExt : Button
-	boutonPoids : Button
-	boutonPression : Button
-	valeurCourante : TextView
-	donneesMin : TextView
-	donneesMoyenne : TextView
-	donneesMax : TextView
#	onCreate(inout savedInstanceState : Bundle) : void
-	initialiserIHM() : void
-	initialiserGraphique() : void
-	extraireDonnees() : void

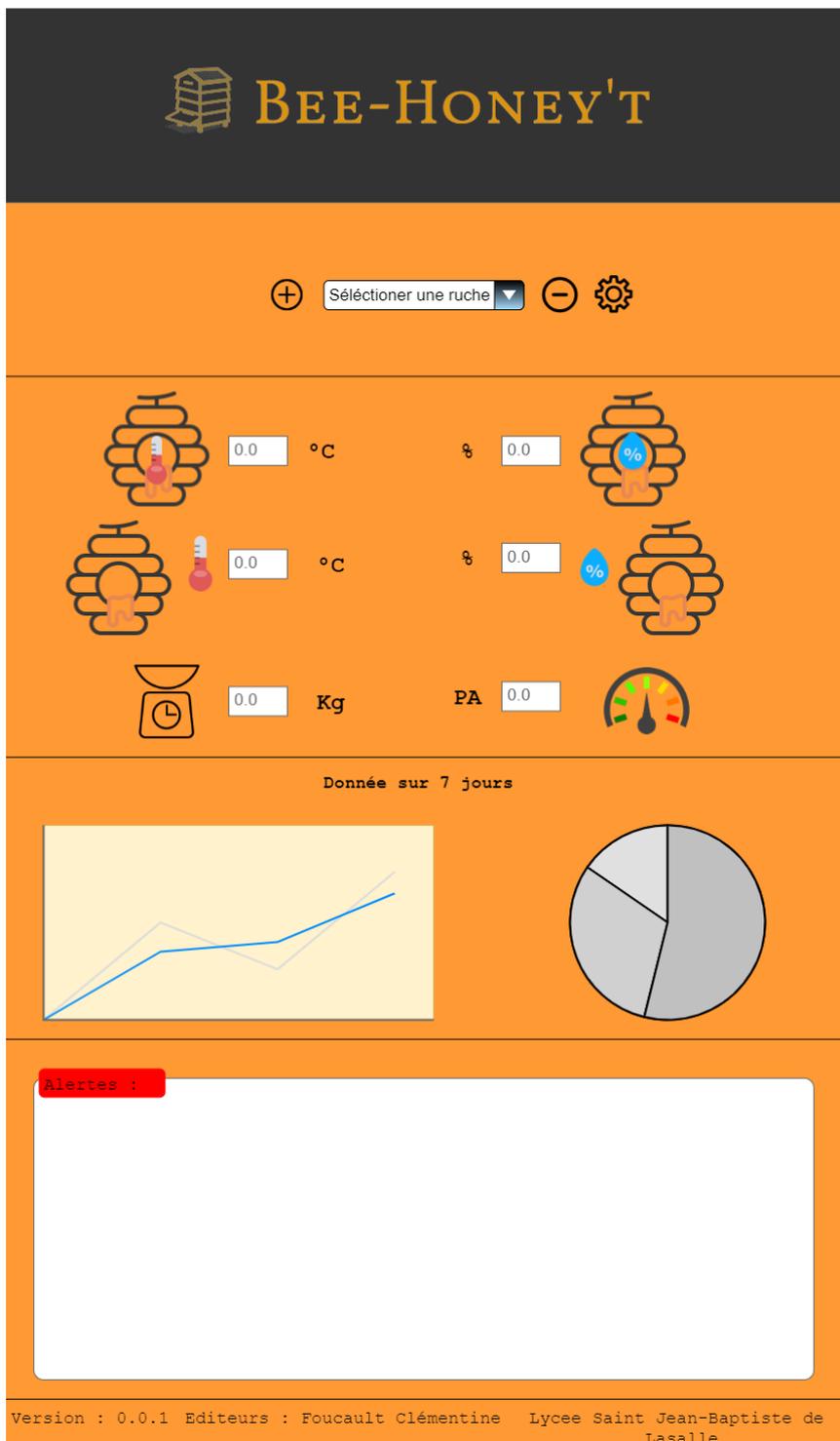
Les attributs :

- contenu
- graphique
- seriesPoids
- boutonTempInt
- boutonTempExt
- boutonHumInt
- boutonHumExt
- boutonPoids
- boutonPression
- valeurCourante
- donneesMin
- donneesMoyenne
- donneesMax

Les méthodes :

- `initialiserIHM()` permet de créer l'IHM
- `initialiserGraphique()` permet de créer les graphiques
- `extraireDonnees()` permet d'extraire les données

Maquette IHM



IHM v1.0

Bee-Honey't



Ruche : Ruche1 Connectée

AJOUTER RUCHE +	SUPPRIMER RUCHE 🗑️	PARAMÉTRER ALERTES 🔧	AFFICHER GRAPHIQUES 📊
--------------------	-----------------------	-------------------------	--------------------------

Température int. : 27.7 °C
Température ext. : 16.4 °C
Poids : 13.6 Kg

Humidité int. : 34 %
Humidité ext. : 34 %
Pression : 1016 hPa

28/05/2021 11:04

Alertes

- Humidité intérieure trop élevée !
- Humidité extérieure trop élevée !
- Poids trop bas !

Fenêtre de paramétrage des seuils d'alertes

Bee-Honey't

Paramétrer les seuils d'alerte

Température intérieure maximale :
28.0

Température intérieure minimale :
12.0

Température extérieure maximale :
35.0

Température extérieure minimale :
10.0

Humidité intérieure maximale :
30

Humidité intérieure minimale :
12

Humidité extérieure maximale :
23

Humidité extérieure minimale :
13

Poids maximal :
42.0

Poids minimal :
41.0

ANNULER AJOUTER

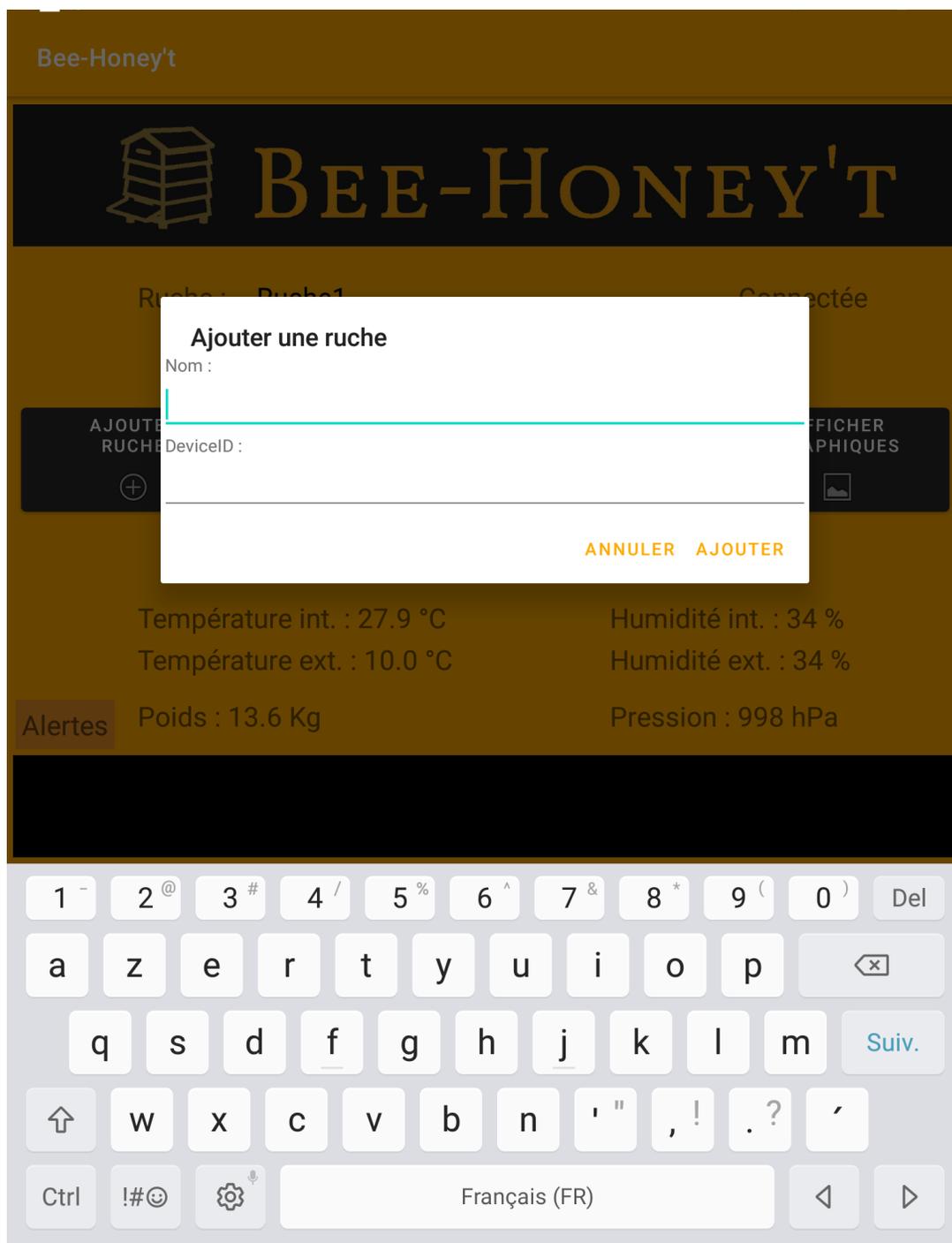
Alertes

Humidité intérieure trop élevée !

Humidité extérieure trop élevée !

Poids trop bas !

Fenêtre d'affichage pour ajouter une ruche



Fenêtre d'affichage pour la suppression

The screenshot displays the Bee-Honey't mobile application interface. At the top, the app name "Bee-Honey't" is visible. Below it is a dark header with a beehive icon and the text "BEE-HONEY'T". The main content area shows "Ruche : Ruche1" and "Connectée". A row of four buttons is present: "AJOUTER RUCHE" (with a plus icon), "SUPPRIMER RUCHE" (with a trash icon), "PARAMÉTRER ALERTES" (with a wrench icon), and "AFFICHER GRAPHIQUES" (with a chart icon). A white dialog box is overlaid on the "SUPPRIMER RUCHE" button, containing the text "Supprimer une ruche" and "Etes-vous sur de vouloir supprimer Ruche1 ?". Below the dialog, the words "NON" and "OUI" are visible. The background shows some data: "Poids : 13.6 Kg" and "Pression : 998 hPa", along with a timestamp "28/05/2021 11:04". At the bottom, there is an "Alertes" section with a dark background and red text: "Humidité intérieure trop élevée !" and "Humidité extérieure trop élevée !", and blue text: "Poids trop bas !".

Bee-Honey't

BEE-HONEY'T

Ruche : Ruche1 Connectée

AJOUTER RUCHE SUPPRIMER RUCHE PARAMÉTRER ALERTES AFFICHER GRAPHIQUES

Supprimer une ruche
Etes-vous sur de vouloir supprimer Ruche1 ?

NON OUI

Poids : 13.6 Kg Pression : 998 hPa

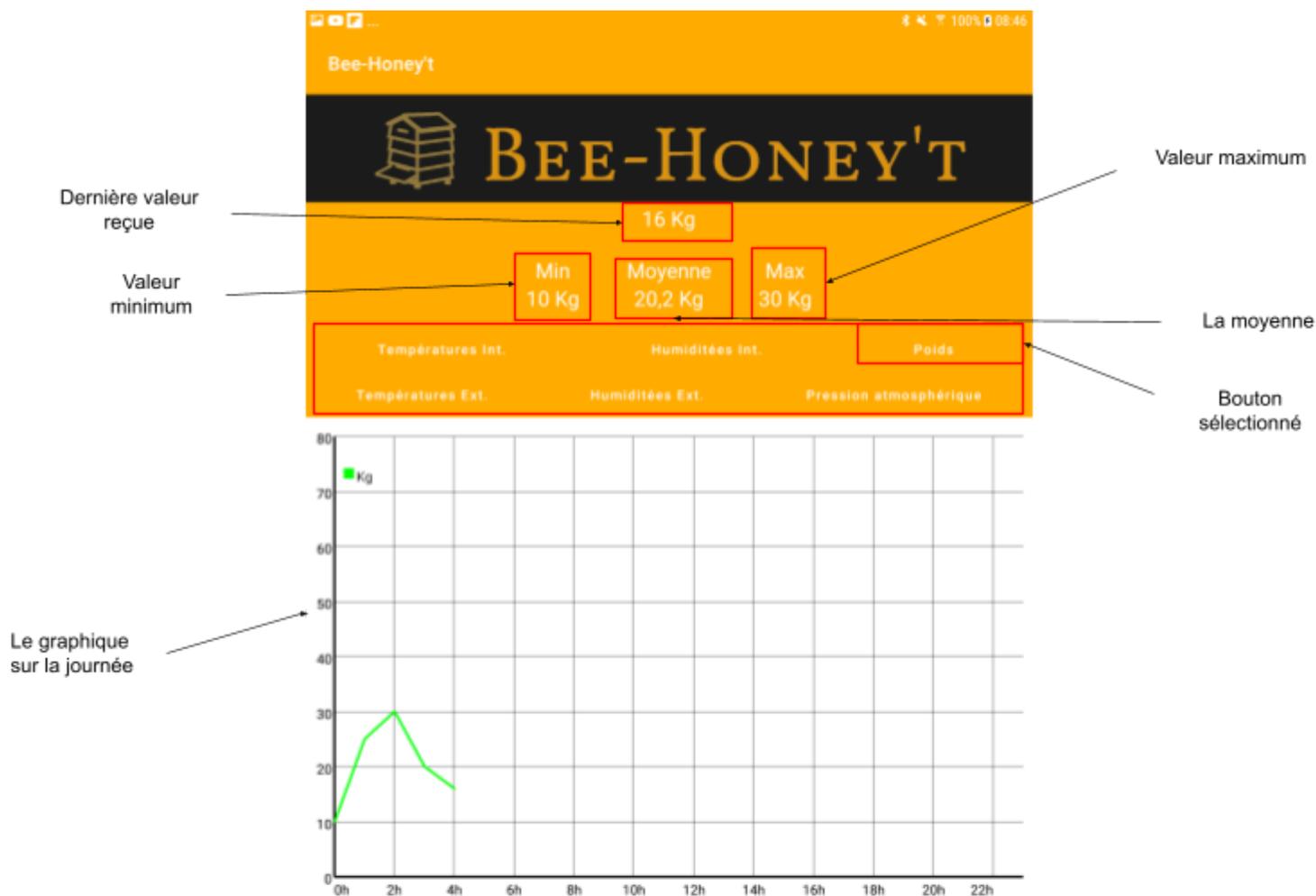
28/05/2021 11:04

Alertes

Humidité intérieure trop élevée !
Humidité extérieure trop élevée !
Poids trop bas !

Deuxième activité pour afficher les graphiques avec la moyenne, le minimum et le maximum ainsi que la dernière valeur reçue. Ici pour la valeur du poids.





Communication MQTT

Création d'un objet client MQTT

```
mqttAndroidClient = new MqttAndroidClient(context,  
"tcp://eu.thethings.network:1883", "rucher");
```

Connexion au serveur TTN

```
MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();  
mqttConnectOptions.setUsername("rucher");  
mqttConnectOptions.setPassword("ttn-account-v2.a4GRsjloPzQ...");  
mqttAndroidClient.connect(mqttConnectOptions, null, null);
```

Installation des fonctions de rappel

```
mqttAndroidClient.setCallback(new MqttCallbackExtended() {  
    public void connectComplete(boolean b, String s) { ... }  
    public void connectionLost(Throwable throwable) { ... }  
    public void messageArrived(String topic, MqttMessage mqttMessage)  
throws Exception  
    { // Traitement du message }  
    public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken)  
{ ... }  
});
```

Décodage de trame en JSON (Java Script Objet Nation)

JSON est un format de données textuelle dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.

JSON a deux types d'éléments structurels avec un système de paires Clé/Valeur et un système de liste ou chaque valeur est séparée par une virgule.

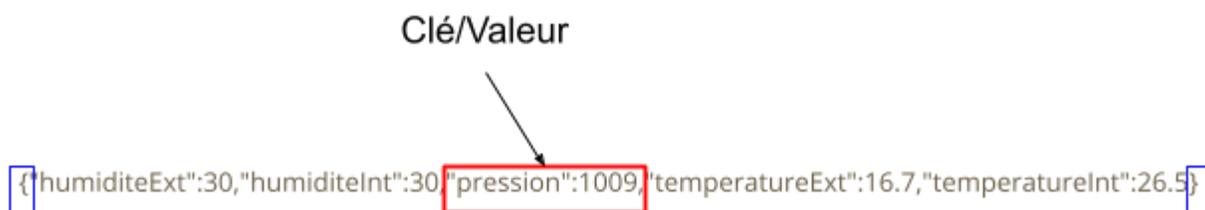
certaines éléments représentent des types de données comme :

- Les objets désignés par des { ... }
- Les tables représentées par des [...]
- Des valeurs générées de type tableau.

Dans notre cas nous utiliserons le système de clé/valeur.

Exemple de JSON avec la trames pour les données d'environnement :

Clé/Valeur



```
{ "humiditeExt":30,"humiditeInt":30,"pression":1009,"temperatureExt":16.7,"temperatureInt":26.5 }
```

 = Objet

Exemple de méthode pour extraire les données du format JSON :

```
/**  
 * payloadFields =  
 {"humiditeExt":44,"humiditeInt":44,"pression":1021,"temperatureExt":14.  
 5,"temperatureInt":23.1}  
 */
```

```
private int extraireHumiditeExterieur(String payloadFields)  
{  
    int humiditeExterieur = 0;  
  
    try  
    {  
        JSONObject json = null;  
        Iterator<String> it = null;  
        json = new JSONObject(payloadFields);  
        humiditeExterieur = json.getInt("humiditeInt");  
    }  
    catch (JSONException e)  
    {  
        e.printStackTrace();  
    }  
  
    return humiditeExterieur;  
}
```

Création d'un nouvelle objet JSON avec notre payloadFields (Notre trames de données) :

```
json = new JSONObject(payloadFields);
```

Récupérer l'humidité extérieure en récupérant la clé :

```
humiditeExterieur = json.getInt("humiditeInt");
```

Ensuite on retourne l'humidité extérieure :

```
return humiditeExterieur;
```

Tout ça est bien sûr encapsulé dans un bloc Try/Catch pour gérer les exceptions.

Stockage des ruches

Pour le projet, on va stocker les informations sur les "ruches" dans un objet stockage de type SharedPreferences

```
private SharedPreferences stockage; //!< Création du stockage
```

On va créer un conteneur d'objets Ruche

```
public Vector<Ruche> obtenirRuches()  
{  
    Vector<Ruche> ruches = new Vector<Ruche>();  
    Map<String,?> donnees = stockage.getAll();  
    for (String id : donnees.keySet())  
    {  
        String nom = "";  
        if(stockage.contains(id))  
        {  
            nom = stockage.getString(id, "");  
        }  
        Ruche rucheActuelle = new Ruche(nom, id);  
        ruches.add(rucheActuelle);  
    }  
    return ruches;  
}
```

Un méthode pour éditer une ruche dans le stockage :

```
public void editerRuche(String nom,String deviceID )  
{  
    stockage.edit().putString(deviceID,nom).apply();  
}
```

Une méthode pour supprimer une ruche du stockage :

```
public void supprimerRuche(String deviceID)  
{  
    stockage.edit().remove(deviceID).apply();  
}
```

Partie physique

Sur notre projet nous utilisons le réseau LoraWAN, celui-ci a une portée rurale entre 15 et 20km avec une fréquence de 868 Mhz en Europe. Son principal concurrent est Sigfox.

Nous avons choisi LoRa car on peut envoyer des messages en illimité alors que pour Sigfox on peut en envoyer seulement 140 par jour. De plus on peut envoyer des message de 242 octets avec LoRa alors que seulement 12 octets avec Sigfox. Certes Sigfox a une plus grande portée mais pour notre projet on n'en aura pas l'utilité.

Même famille LPWA (Low Power, Wide Area)
Réseaux dédiés spécifiquement aux applications IoT à longue portée et faible consommation

	 sigfox	VS	 LoRa
Type de réseau :	Ouvert (open source)		Propriétaire
Origine :	Start-up Française toulousaine		Technologie et Protocole développés par SMETECH (société Américaine) à partir d'une technologie Française (Cycleo)
Portée urbaine :	3 à 10 kms		3 à 8 kms
Portée rurale :	30 à 50 kms		15 à 20kms
Modulation :	SS chip Ultra Narrow Band (UNB)		UNB/GFSK/BOSK
Abonnement par appareil :	1 et 15€ /an		1 et 15€ /an
Bidirectionnel :	Oui		Oui
Messages max :	12 octets		242 octets
Messages par jour :	140		Illimité
Débit :	100bits/s		0,3 à 50kbits/s
Couverture :	93% de la population française (supporté par Bouygues et Orange)		94% de la population française (supportée par SFR)
Sécurité :	Appareil et station protégés par id unique		Encryptage AES128
Immunité aux bruits :	Basse		Très haute
Autonomie batterie 2000mAh :	90 mois		105 mois

Tests de validation

Désignation	Démarche à suivre	Résultat attendu	Oui / Non
S'authentifier	Nom d'utilisateur et mot de passe permettant un accès sécurisé	Pouvoir s'authentifier	Non
Créer une ruche	Cliquer sur le bouton "+" et configurer les paramètres de la ruche	Création d'une ruche	Oui
Sélectionner une ruche	Cliquer sur la liste déroulante et sélectionner la ruche souhaitée	Avoir le nom de la ruche sélectionné affiché	Oui
Recevoir les données d'une ruche	Une fois la ruche sélectionnée	Les visualisé sur l'IHM	Oui
Supprimer une ruche	Cliquer sur le bouton "-" et confirmer la suppression de la ruche	Enlever une ruche	Oui
Gestion des alertes	Visualiser les alertes en bas de l'IHM	Recevoir des notification	Oui

Partie application pc (Mhadi Zakariya)

Introduction

Le développement de l'interface PC s'est réalisé à l'aide d'une bibliothèque logicielle orientée objet développée en C++ : Qt. La bibliothèque logicielle fournie par Qt regroupe un ensemble de classes. Qt offre notamment des composants (widgets) permettant de concevoir des interfaces graphiques.

source de la documentation : <https://doc.qt.io/>

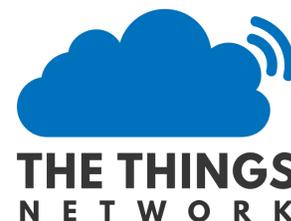
Les données des capteurs sont envoyées sous formes de Trame, dans un format de JSON (*JavaScript Object Notation*) : un format de données textuelles dérivé de la notation des objets du langage JavaScript.

source de la documentation : <https://json.org/json-fr.html>).

Le protocole de messagerie est le protocole MQTT, un protocole *publish-subscribe* basé sur le protocole TCP/IP. Les données sont reçues au format JSON par le protocole MQTT via le réseau TTN.

voir la partie [MQTT](#).

Les définitions et explications de toutes les méthodes différentes utilisées au cours de cette réalisation seront définies par la suite (voir [Glossaire](#)).



Présentation de l'IHM

L'Interface Homme Machine (IHM) se présente sous cette forme :



La présence de quatre boutons : Ajouter, Supprimer, Connecter et Afficher informations.

Dans un premier temps, le bouton Connecter servira à se connecter au serveur TTN (The Things Network) à l'aide des paramètres de connexions stockés dans un fichier de configuration : beehoneyt.ini.

La présence d'un bouton Ajouter et Supprimer, permet l'édition d'une ruche nouvelle ou sélectionnée. Une liste de ruches, qui contiendra toutes les ruches du rucher.

La présence de trois onglets différents : Mesures, Graphiques propre à la ruche sélectionnée et Journal qui contient tous les messages de journalisation.

L'interface comprend également des notifications visuelles (images), qui changeront d'état si les mesures reçues sont considérées comme anormales. L'utilisateur pourra régler les niveaux d'alertes (maximum et minimum).

La version de la livrable est la 1.0

Planification

Voici sous forme de tableau les fonctionnalités attendues dans l'itération 1.

Fonctionnalités	Itération
Création interface	1
Affichage des mesures des capteurs (température int. & ext., humidité int. & ext., poids et pression)	1
Recevoir les données des capteurs	1
Gestion de la ruche : nomenclature, paramétrage, ajout-suppression	1
Actualisation des données en temps réels	1

Diagramme de GANTT

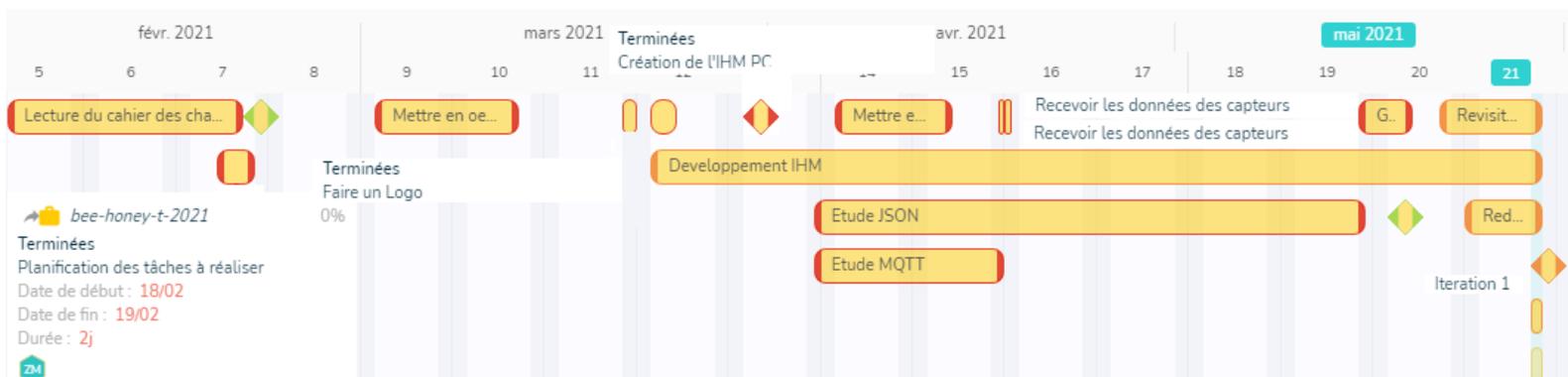
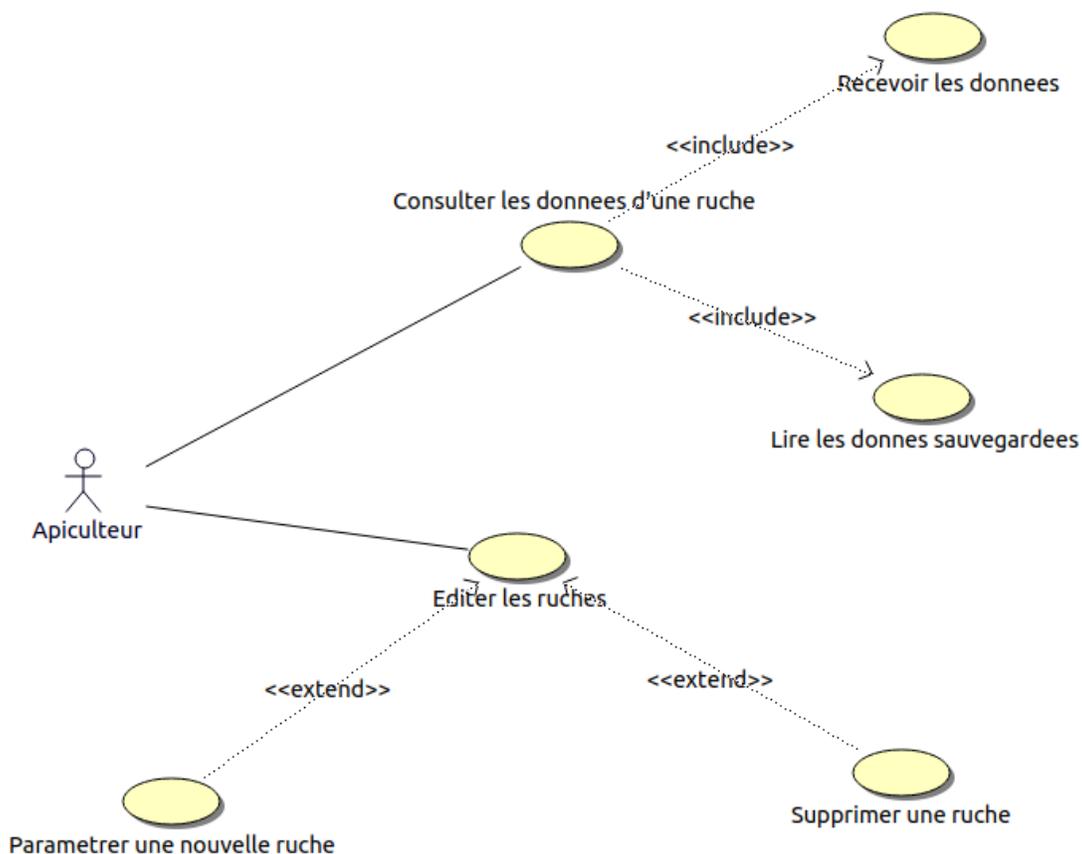


Diagramme de cas d'utilisation

L'apiculteur peut consulter les données d'une ruche : température intérieure/extérieure, humidité intérieure/extérieure, pression, poids.



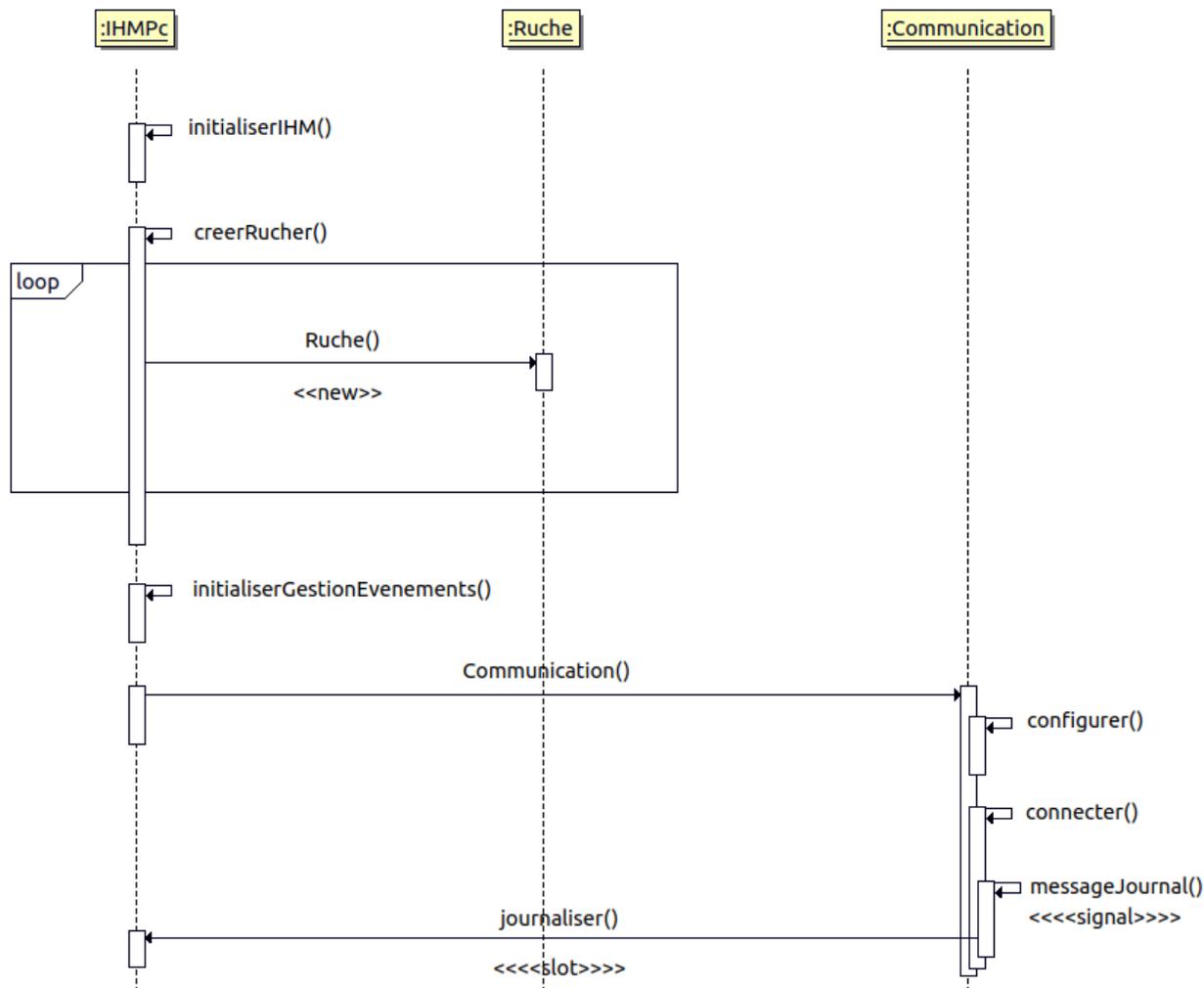
L'utilisateur peut ajouter une ruche en lui affectant un Nom : qui sera visible dans la liste de ruches, et qui fait office d'identifiant. L'apiculteur doit également ajouter le deviceID de sa ruche, qui permettra la récupération des mesures de celle-ci sur le TTN : *The Things Network*.

The Things Network propose aussi un service *Data Storage* assurant la sauvegarde de données pour les 7 derniers.

L'apiculteur est alerté en cas de mesures anormales (auparavant défini par l'apiculteur ou par défaut) par une notification visuelle sur l'IHM.

Diagramme de séquence et code

Démarrage de l'IHM



Voici le diagramme de séquence du démarrage de notre application. Dans un premier temps l'IHM sera initialisé :

```

void IHMPc::initialiserIHM()
{
    ui->setupUi(this);
    ui->ongletIHM->setCurrentIndex(OngletIHM::ONGLET_MESURES);
    showMaximized();
}
  
```

L'interface s'ouvre sur l'onglet Mesures par défaut. La fenêtre d'application s'affiche en plein écran.

Un rucher est également créé dans un premier temps :

```
void IHMPc::creerRucher()
{
    // Tests d'un rucher composé de deux ruches (simulateur)
    rucher.push_back(new Ruche("Ruche 1", "ruche-1-sim", this));
    rucher.push_back(new Ruche("Ruche 2", "ruche-2-sim", this));
    //rucher.push_back(new Ruche("Ruche BeeHoneyT", "bee_honeyt", this));
    for(int i=0;i<rucher.size();++i)
    {
        ui->listeRuches->addItem(rucher[i]->getNom());
    }

    indexRucheSelectionnee = 0;
    ui->listeRuches->setCurrentIndex(indexRucheSelectionnee);
    ui->labelRucheSelectionnee->setText("Ruche sélectionnée : " +
    ui->listeRuches->currentText());
}
```

Les deux ruches simulées ont été ajoutées. Une ruche se construit à l'aide de grâce à cette méthode : [Ruche\(QString nom, QString deviceID\)](#):

Le premier argument initialisera l'attribut nom de la ruche en QString et le deuxième le deviceID.

Les deux ruches seront ensuite ajoutées par leur nom grâce à la méthode [Ruche::getNom\(\)](#) dans la liste de ruche **listeRuches** de type QComboBox à l'aide d'une boucle for qui parcourt le QVector **rucher**.

La liste sélectionne la première ruche par défaut (**indexRucheSelectionnee = 0**), donc la première ruche du rucher. Le champ de texte **labelRucheSelectionnee** de type QLabel, nous indique par un message construit la ruche sélectionnée.

La méthode [IHMPc::initialiserGestionEvenements\(\)](#) :

```
void IHMPc::initialiserGestionEvenements()
{
    qRegisterMetaType<MesureRuche>();
    connect(communicationTTN, SIGNAL(ttnConnecte()), this, SLOT(afficherEtatConnecte()));
    connect(communicationTTN, SIGNAL(ttnDeconnecte()), this, SLOT(afficherEtatDeconnecte()));
    connect(communicationTTN, SIGNAL(messageJournal(QString)), this,
    SLOT(journaliser(QString)));
    qRegisterMetaType<MesureRuche>();
    connect(communicationTTN, SIGNAL(nouvellesMesures(MesureRuche)), this,
    SLOT(afficherNouvellesMesures(MesureRuche)));
}
```

```

connect(historique, SIGNAL(messageJournal(QString)), this, SLOT(journaliser(QString)));

connect(ui->boutonTTN, SIGNAL(clicked(bool)), this, SLOT(gererConnexionTTN()));
connect(ui->boutonAjouter, SIGNAL(clicked(bool)), this, SLOT(ouvrirFenetreAjouter()));
connect(ui->boutonSupprimer, SIGNAL(clicked(bool)), this, SLOT(supprimerRuche()));
connect(ui->boutonAfficher, SIGNAL(clicked(bool)), this,
        SLOT(afficherInformationsRuche()));
connect(ui->listeRuches, SIGNAL(currentIndexChanged(int)), this,
        SLOT(selectionnerRuche()));
connect(ui->ongletIHM, SIGNAL(currentChanged(int)), this,
        SLOT(gererChangementOnglet(int)));
}

```

Cette méthode réalise la connexion entre signal/slot. Les signaux et les slots forment un mécanisme de communication entre objets Qt :

- Le signal représente un événement
- Le slot représente le gestionnaire d'événement

Un **signal** se déclare comme une méthode mais il n'aura pas définition. On utilisera un nom pour préciser l'événement qu'il représente. Un slot est une méthode (il se déclare et se définit). On utilisera un verbe pour préciser l'action qui sera déclenchée en réponse à un signal.

L'association d'un signal à un slot est réalisée par une connexion avec l'appel **connect()** (une méthode statique de la classe QObject).

Un pointeur vers un objet de type Communication est maintenant déclaré et construit à l'aide du constructeur `new Communication()`.

La méthode configurer() est appelé par la suite :

```

void Communication::configurer()
{
    QSettings settings(QDir::currentPath() + "/beehoneyt.ini", QSettings::IniFormat);

    settings.beginGroup("TTN");
    hostname = settings.value("Hostname").toString();
    port = settings.value("Port").toInt();
    username = settings.value("Username").toString();
    password = settings.value("Password").toString();
    settings.endGroup();
    settings.sync();
}

```

```

QDebug() << Q_FUNC_INFO << hostname << port << username << password;

connect(clientMqtt, SIGNAL(stateChanged(ClientState)), this, SLOT(changerEtat()));
}

```

Cette méthode ouvre le fichier de configuration beehoneyt.ini. Elle crée un objet de type QSettings, par la méthode **QSettings(const QString & fileName, QSettings::Format format, QObject * parent = nullptr)**.

Le premier argument est défini par le chemin absolu de **beehoneyt.ini** à l'aide de la classe QDir, et de sa méthode **currentPath()**, qui retourne le chemin absolu de l'application qui l'exécute, voir <https://doc.qt.io/qt-5/qdir.html#currentPath>.

Le deuxième argument est défini par **QSettings::iniFormat** : cette valeur indique que les informations de type ne sont pas conservées lors de la lecture des paramètres à partir de fichiers .ini; toutes les valeurs seront renvoyées sous forme de QString, voir <https://doc.qt.io/qt-5/qsettings.html#Format-enum>.

Voici le contenu du fichier beehoneyt.ini :

```

[TTN]
Hostname=eu.thethings.network
Password=ttn-account-.....
Port=18..
Username=rucher

```

L'objet settings est ensuite utilisé afin d'accéder au fichier et permet la lecture de celui-ci sans oublier le type de retour souhaité : celui sera directement affecté à l'attribut de l'objet Communication.

La méthode connecter() sera ensuite appelée :

```

void Communication::connecter()
{
    if(clientMqtt->state() != QMqttClient::Connected)
    {
        qDebug() << Q_FUNC_INFO << hostname << port << username << password;

        clientMqtt->setHostname(hostname);
        clientMqtt->setPort(port);
        clientMqtt->setUsername(username);
        clientMqtt->setPassword(password);

        QString message = "Connexion " + hostname;
    }
}

```

```
emit messageJournal(message);
clientMqtt->connectToHost();

connect(clientMqtt, SIGNAL(messageReceived(QByteArray,QMqttTopicName)), this,
SLOT(recevoirMessage(QByteArray,QMqttTopicName)));
}
}
```

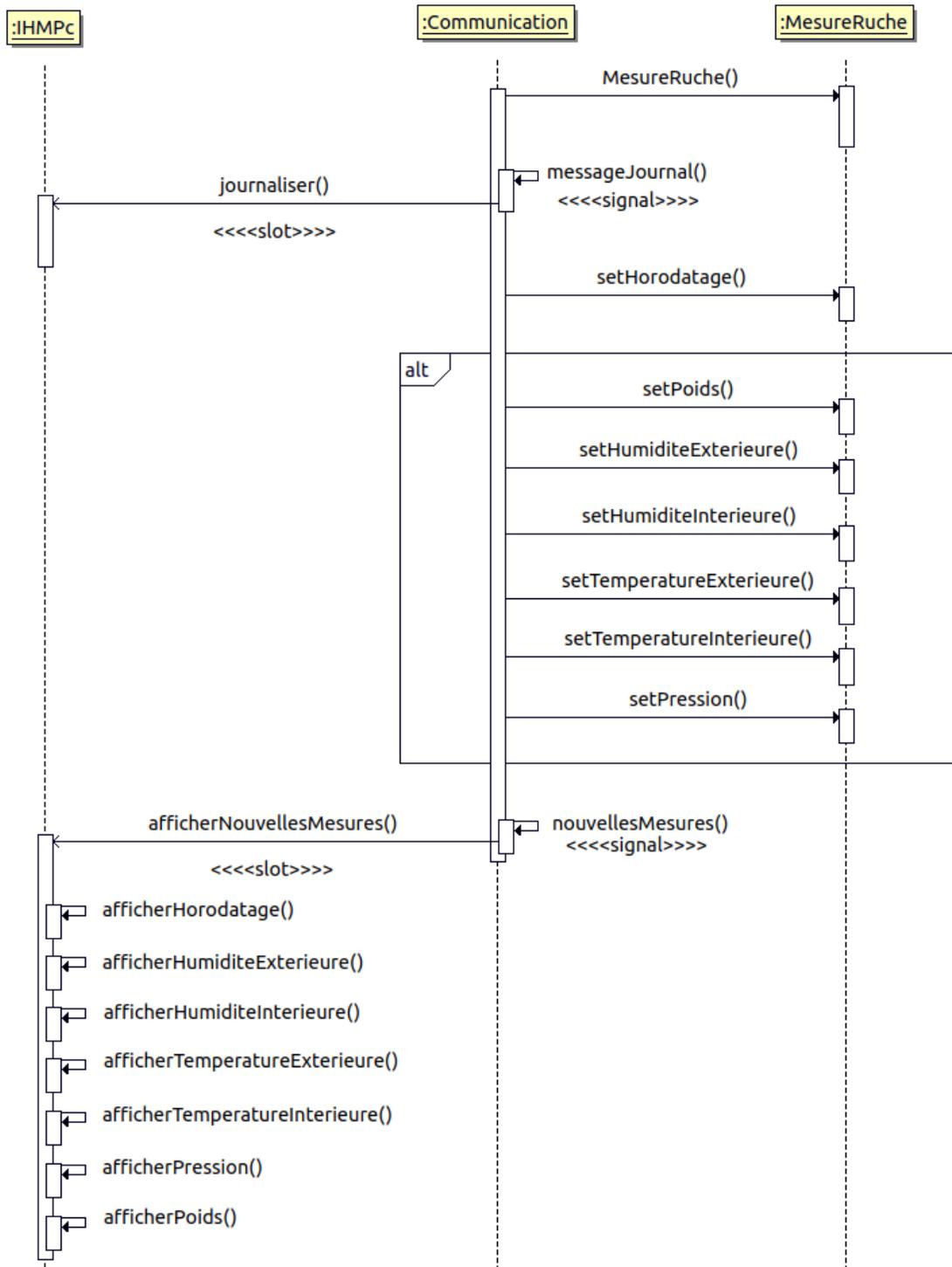
Dans un premier temps, la boucle for s'assurera que le **clientMqtt** (attribut de la classe [Communication](#)) est bien déconnecté.

Par la suite, le **clientMqtt** (qui est un pointeur vers un objet de type [QMqttClient](#)) appellera quatre méthodes de type mutateurs (méthode qui permet l'accès en écriture des attributs de l'objet [Communication](#)).

Ces attributs qui ont été auparavant modifiés par la méthode **configurer()** de l'objet [Communication](#) sont utilisés pour la connexion du **clientMqtt**.

Une journalisation par le signal [messageJournal\(\)](#), connecté au slot [journaliser\(\)](#). Le **clientMqtt** se connecte ensuite par la méthode `connectToHost()`.

Consulter les données sur l'IHM



Voici ci dessus, le diagramme de séquence illustrant la consultation des données. Ce diagramme est défini en parti par la méthode [recevoirMessage\(const QByteArray &messageRecu, const QMqttTopicName &topic\)](#) :

```
void Communication::recevoirMessage(const QByteArray &messageRecu, const QMqttTopicName
&topic)
{
    QJsonDocument documentJSON = QJsonDocument::fromJson(messageRecu);
    QJsonObject objetJSON = documentJSON.object();

    // Extraction du deviceID et du port
    QString deviceID = objetJSON.value(QString("dev_id")).toString();
    int port = objetJSON.value(QString("port")).toInt();

    qDebug() << Q_FUNC_INFO << "topic" << topic;
    qDebug() << Q_FUNC_INFO << "DeviceID " << deviceID;
    qDebug() << Q_FUNC_INFO << "message" << messageRecu;
    qDebug() << Q_FUNC_INFO << "port" << port;

    // Journalisation

    QString message = "Message reçu de " + deviceID + " :\n" + messageRecu;
    emit messageJournal(message); // Extrait d'un message reçu

    // Extraction de l'horodatage
    QJsonObject messageMetadata = objetJSON.value(QString("metadata")).toObject();
    QString valeurHorodatage = messageMetadata.value(QString("time")).toString();
    qDebug() << Q_FUNC_INFO << "valeurHorodatage" << valeurHorodatage;
    QDateTime horodatage = QDateTime::fromString(valeurHorodatage,
Qt::ISODate).toLocalTime();
    qDebug() << Q_FUNC_INFO << "horodatage" << horodatage.toString("dd/MM/yyyy HH:mm:ss");
    mesure.setHorodatage(horodatage.toString("dd/MM/yyyy HH:mm:ss"));

    // Extraction des mesures
    QJsonObject messageMesures = objetJSON.value(QString("payload_fields")).toObject();
    QJsonValue valeurPoids, valeurHumiditeExt, valeurHumiditeInt, valeurTemperatureExt,
valeurTemperatureInt, valeurPression;

    switch (port)
    {
    case PORT_TTN_POIDS:
        valeurPoids = messageMesures.value(QString("poids")).toDouble();

        mesure.setPoids(valeurPoids.toDouble());

        emit nouvellesMesures(mesure);
        break;
    case PORT_TTN_ENVIRONNEMENT:
```

```

    valeurHumiditeExt = messageMesures.value(QString("humiditeExt")).toDouble();
    valeurHumiditeInt = messageMesures.value(QString("humiditeInt")).toDouble();
    valeurTemperatureExt = messageMesures.value(QString("temperatureExt")).toDouble();
    valeurTemperatureInt = messageMesures.value(QString("temperatureInt")).toDouble();
    valeurPression = messageMesures.value(QString("pression")).toDouble();

    mesure.setHumiditeExterieur(valeurHumiditeExt.toDouble());
    mesure.setHumiditeInterieur(valeurHumiditeInt.toDouble());
    mesure.setTemperatureExterieur(valeurTemperatureExt.toDouble());
    mesure.setTemperatureInterieur(valeurTemperatureInt.toDouble());
    mesure.setPression(valeurPression.toDouble());
    qDebug() << Q_FUNC_INFO << "mesure" << mesure.getTemperatureInterieur() <<
    mesure.getTemperatureExterieur() << mesure.getHumiditeInterieur() <<
    mesure.getHumiditeExterieur() << mesure.getPression();

    emit nouvellesMesures(mesure);
    break;
default:
    break;
}
}

```

Premièrement, la méthode [recevoirMessage\(const QByteArray &messageRecu, const QMqttTopicName &topic\)](#) possède deux paramètres : un objet constant **messageRecu** de type QByteArray par référence, et un objet constant **topic** de type QMqttTopicName par référence aussi. Il faut garder en tête que l'objet **messageRecu** se présente sous le format JSON.

messageRecu est directement utilisé à l'initialisation de l'objet **documentJSON** de type QJsonDocument par la méthode [fromJson\(const QByteArray &messageRecu\)](#) : cette méthode reçoit un objet de type QByteArray encodé en UTF-8 et retourne un objet un QJsonDocument à partir de celui-ci.

documentJSON appelle ensuite la méthode [object\(\) const](#), qui retourne l'objet JSON contenu dans le document : le type de retour est par conséquent un QJsonObject.

Remarque : lire [JSON](#)

De ce fait, la méthode [value\(const QString &cle\) const](#), retourne un QJsonValue, qui correspondra à la valeur de **clé** de type QString, correspondant à la clé.

L'extraction d'une donnée se généralise sous cette forme pour un double par exemple :

```
double nomValeur = objetJSON.value(QString("clé")).toDouble();
```

Dans ce projet, les données des capteurs ne sont pas envoyées sur le même port : un switch est mis en place, en fonction du port (au préalable extrait) afin de traiter la trame reçue.

Les mutateurs de la classe [MesureRuche](#) sont ensuite appelés par l'objet, afin de modifier les attributs de l'objet **mesureRuche** par les valeurs extraites.

Une journalisation par le signal [messageJournal\(\)](#), connecté au slot [journaliser\(\)](#). Un signal est enfin émis, le signalant à l'IHM par [nouvellesMesures\(MesureRuche mesure\)](#) connecté au slot [afficherNouvellesMesures\(MesureRuche mesure\)](#).

Un objet `MesureRuche` est alors envoyé, placé en paramètre, à travers la connexion signal/slot. Le transfert de l'objet permet donc l'utilisation de celui-ci dans les méthodes de l'IHM : [afficherHumiditeExterieur\(double humiditeExterieur, QString uniteHumidite\)](#) par exemple.

```
void IHMPc::afficherHumiditeExterieur(double humiditeExterieur, QString uniteHumidite)
const
{
    ui->lcdHumiditeExterieur->display(humiditeExterieur);
    ui->labelUniteHumiditeExterieur->setText(uniteHumidite);
}
```

Le changement de valeur du `QLCDNumber` (ici nommé **lcdHumiditeExterieur**) de l'Humidité Extérieure par la méthode [QLCDNumber::display\(double humiditeExterieur\)](#).

La récupération de la valeur de l'humidité extérieure se fait dans le slot [afficherNouvellesMesures\(MesureRuche mesure\)](#) :

```
void IHMPc::afficherNouvellesMesures(MesureRuche mesure)
{
    if(!mesure.getHorodatage().isEmpty())
    {
        afficherHorodatage(mesure.getHorodatage());
        afficherPoids(mesure.getPoids());
        afficherTemperatureInterieur(mesure.getTemperatureInterieur());
        afficherTemperatureExterieur(mesure.getTemperatureExterieur());
        afficherHumiditeInterieur(mesure.getHumiditeInterieur());
        afficherHumiditeExterieur(mesure.getHumiditeExterieur());
        afficherPression(mesure.getPression());
    }
}
```

Elle se fera par l'utilisation d'un accesseur de l'objet **mesure**, dans la méthode vu précédemment : [mesure.getHumiditeExterieur\(\)](#) , qui retourne une valeur de type double.

JSON

JSON est un format de données textuelle dérivé de la notation des objets du langage JavaScript. Il permet de représenter de l'information structurée comme le permet XML par exemple.

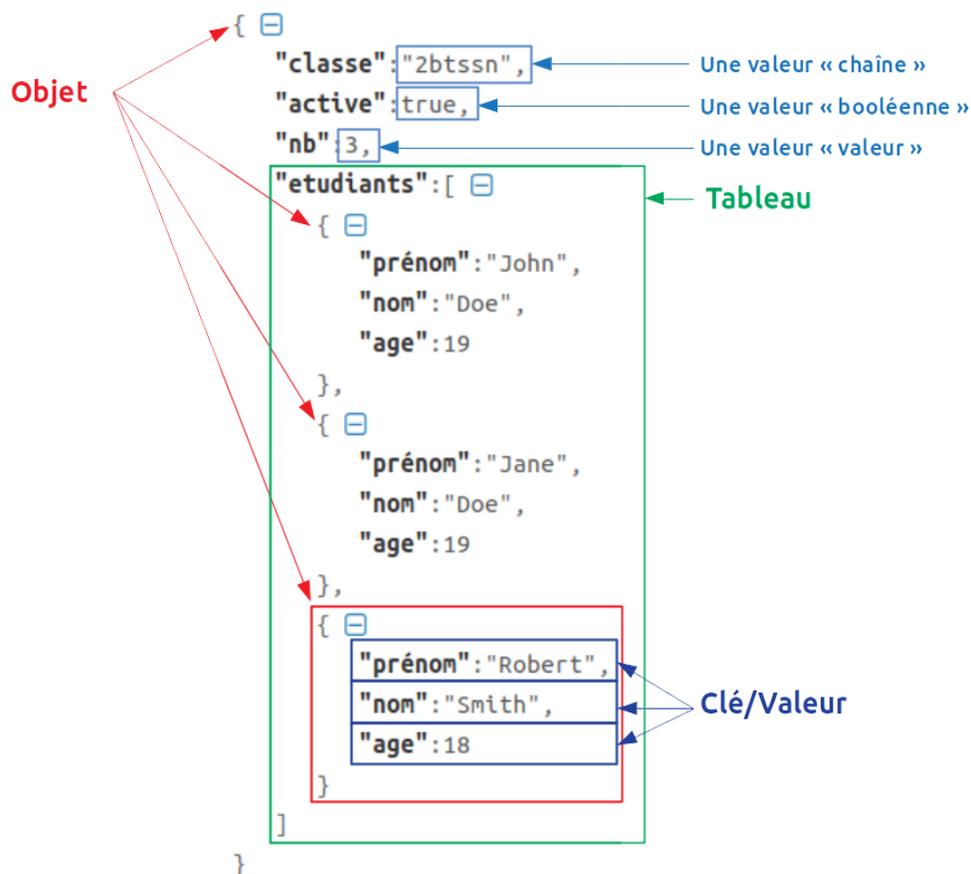
JSON a deux types d'éléments structurels avec un système de paires Clé/Valeur et un système de liste ou chaque valeur est séparée par une virgule.

certaines éléments représentent des types de données comme :

- Les objets désignés par des { ... }
- Les tables représentées par des [...]
- Des valeurs générées de type tableau.

Dans notre cas nous utiliserons le système de clé/valeur.

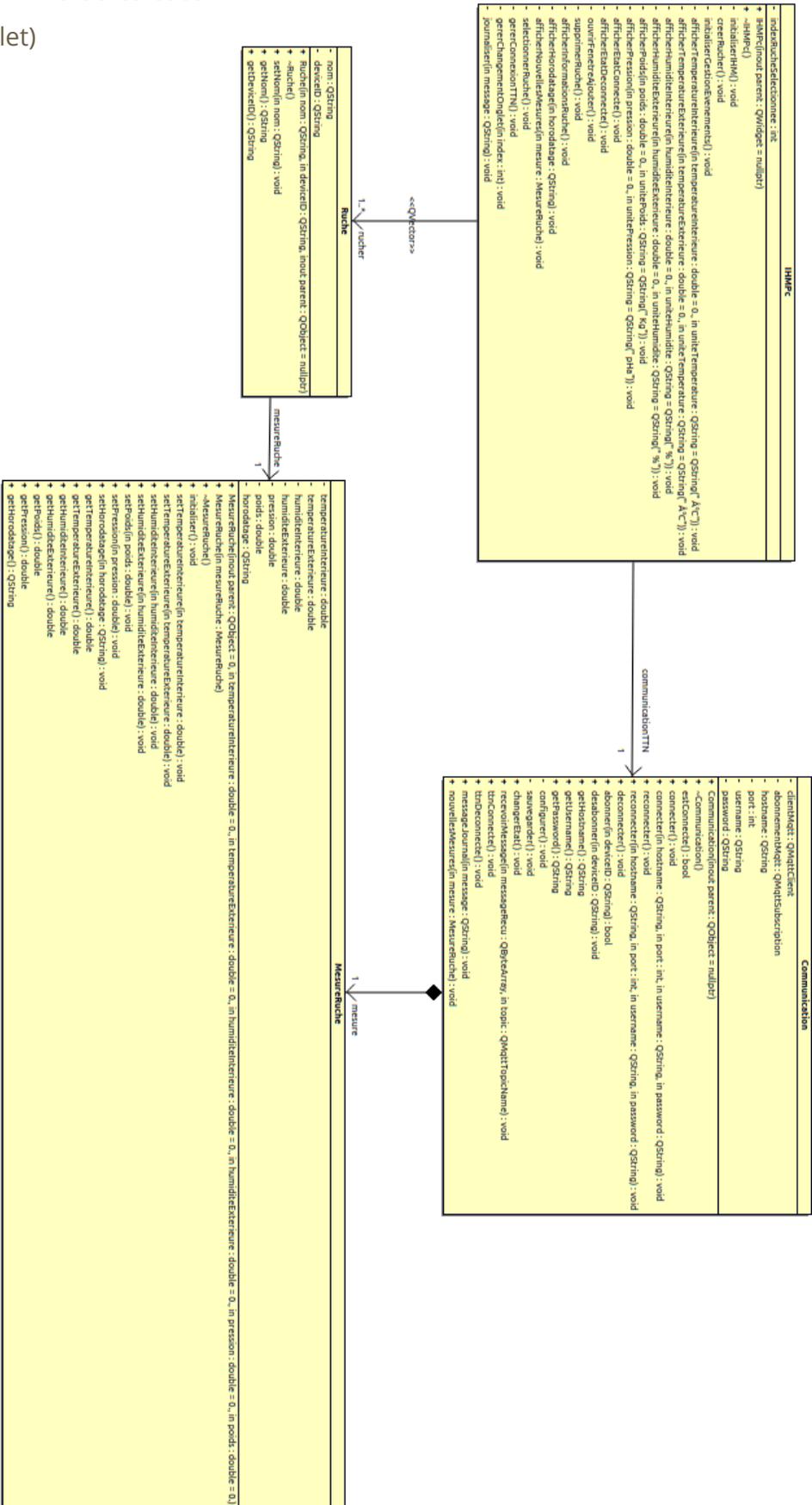
Voici un exemple :



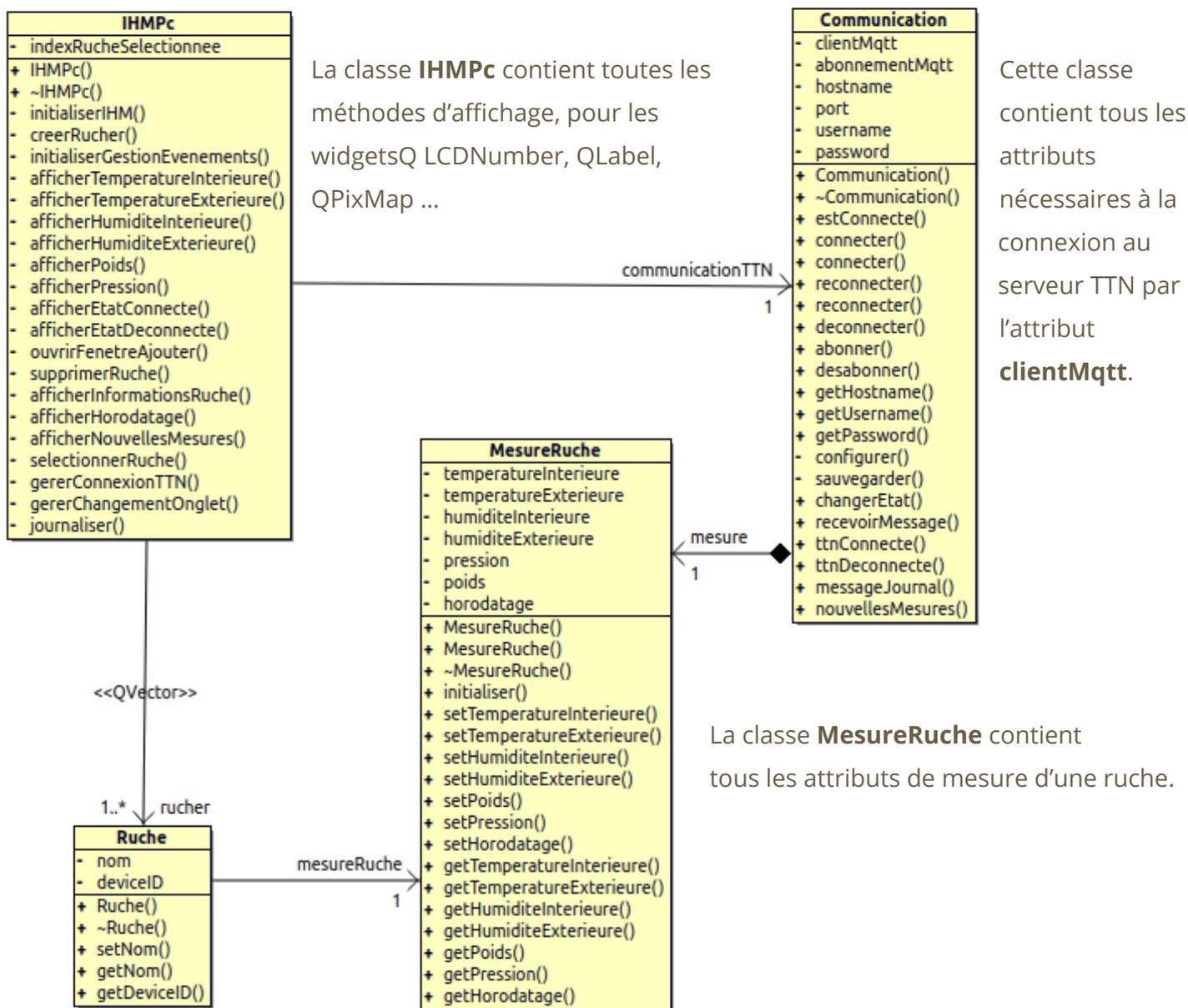
Voici concrètement un message reçue sous format JSON :

```
{
  "app_id":"rucher",
  "dev_id":"ruche-1-sim",
  ...,
  "port":2,
  "payload_fields":
  {
    "humiditeExt":32,
    "humiditeInt":32,
    "pression":1003,
    "temperatureExt":17.4,
    "temperatureInt":23.1
  },
  "metadata":
  {
    "time":"2021-04-18T08:11:35.714925102Z",
    "frequency":868.3,
    "modulation":"LORA",
    ...
    "gateways":
    [
      {
        "gtw_id":"btssn-lasalle-84",
        "timestamp":3459190155,
        "time":"2021-04-18T08:11:35Z",
        ...
      }
    ]
  }
}
```

Diagramme de classes (complet)

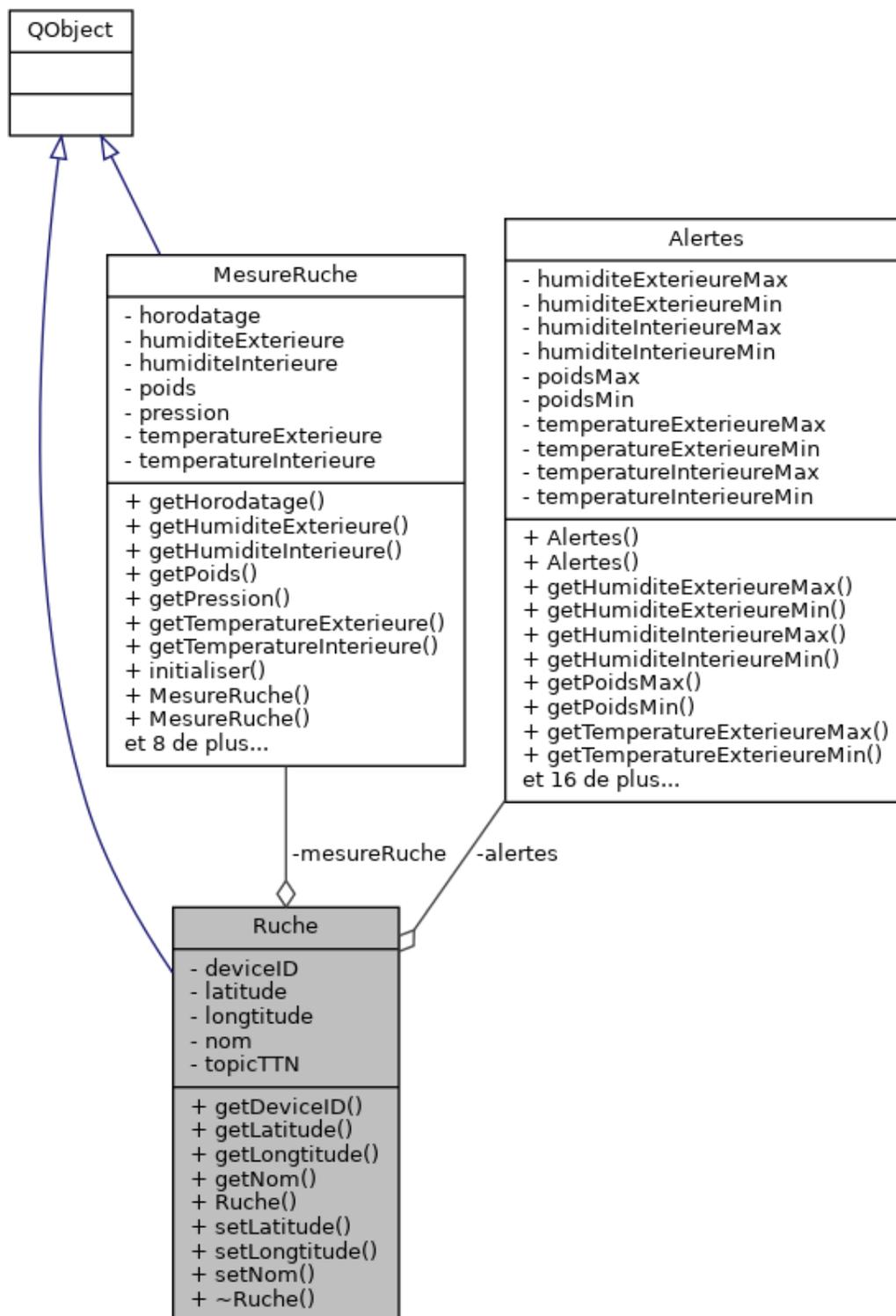


(simplifié)



Un objet de type **Ruche** se définit par un nom et un deviceID.

Diagramme de la classe Ruche



Les fonctions membre de la classe

QString **getDeviceID** () const

Méthode qui retourne le deviceID.

QString **getNom** () const

Méthode qui retourne le nom.

Ruche (QString **nom**, QString **deviceID**, **QObject** *parent=nullptr)

Constructeur de la classe **Ruche**.

void **setNom** (QString **nom**)

Méthode qui change le nom.

~Ruche ()

Destructeur de la classe **Ruche**.

Attributs privés

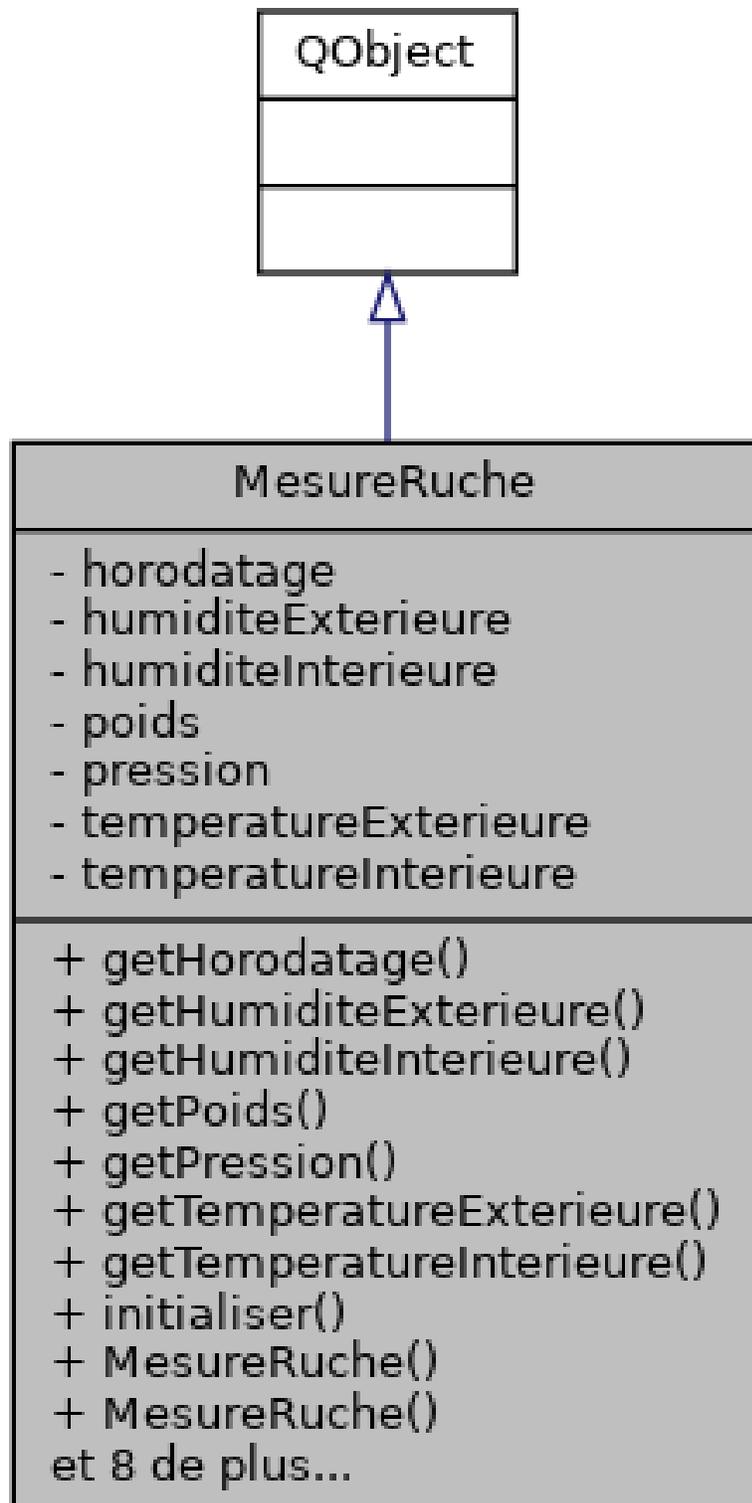
Alertes *	alertes
------------------	----------------

QString	deviceId
---------	-----------------

MesureRuche *	mesureRuche
----------------------	--------------------

QString	nom
---------	------------

Diagramme de la classe **MesureRuche**



Les fonctions membres de la classe

publiques

QString **getHorodatage** () const

Méthode qui retourne la valeur de l'horodatage.

double **getHumiditeExterieur** () const

Méthode qui retourne la valeur de l'humidité extérieure.

double **getHumiditeInterieur** () const

Méthode qui retourne la valeur de l'humidité intérieure.

double **getPoids** () const

Méthode qui retourne la valeur du poids.

double **getPression** () const

Méthode qui retourne la valeur de la pression.

double **getTemperatureExterieur** () const

Méthode qui retourne la valeur de la température extérieure..

double **getTemperatureInterieur** () const

Méthode qui retourne la valeur de la température intérieure.

void **initialiser** ()

Méthode qui initialise les attributs.

MesureRuche (const **MesureRuche** &mesureRuche)

MesureRuche (QObject *parent=0, double **temperatureInterieure**=0., double **temperatureExterieur**=0., double **humiditeInterieure**=0., double **humiditeExterieur**=0., double **pression**=0., double **poids**=0.)

void **setHorodatage** (QString **horodatage**)

Méthode qui change la valeur de l'horodatage.

void **setHumiditeExterieur** (double **humiditeExterieur**)

Méthode qui change la valeur de l'humidité extérieure.

void **setHumiditeInterieure** (double **humiditeInterieure**)

Méthode qui change la valeur de l'humidité intérieure.

void **setPoids** (double **poids**)

Méthode qui change la valeur du poids.

void **setPression** (double **pression**)

Méthode qui change la valeur de la pression.

void **setTemperatureExterieur** (double **temperatureExterieur**)

Méthode qui change la valeur de la température extérieure.

void **setTemperatureInterieur** (double **temperatureInterieur**)

Méthode qui change la valeur de la température intérieure.

~MesureRuche ()

Destructeur de la classe **MesureRuche**.

Attributs privés

QString	horodatage
---------	-------------------

double	humiditeExterieur
--------	--------------------------

double	humiditeInterieur
--------	--------------------------

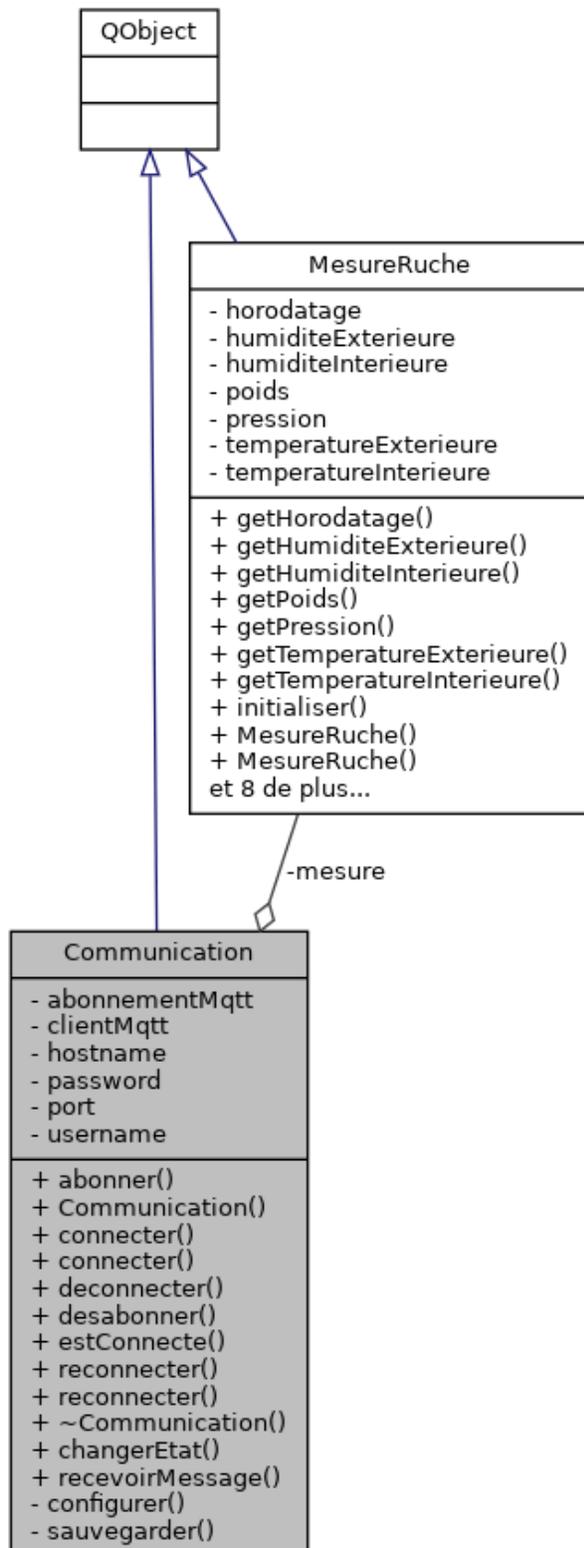
double	poids
--------	--------------

double	pression
--------	-----------------

double	temperatureExterieur
--------	-----------------------------

double	temperatureInterieur
--------	-----------------------------

Diagramme de la classe **Communication**



Les fonctions membres de la classe

publiques

bool **abonner** (QString deviceId)

Méthode qui abonne le client Mqtt à un topic , à partir du deviceId.

Communication (QObject *parent=nullptr)

Constructeur de la classe **Communication**.

void **connecter** ()

Méthode qui connecte le client Mqtt (par défaut) et permet la récupération du message (à l'aide de la connection)

void **connecter** (QString hostname, int port, QString username, QString password)

void **deconnecter** ()

Méthode qui déconnecte le client Mqtt.

void **desabonner** (QString deviceId)

Méthode qui désabonne le client Mqtt , à partir du deviceId.

bool **estConnecte** () const

Méthode qui retourne une valeur booléenne true si le client Mqtt est connecté

void **reconnecter** ()

Méthode qui déconnecte puis connecte le client Mqtt.

void **reconnecter** (QString **hostname**, int **port**, QString **username**, QString **password**)

~Communication ()

Destructeur de la classe **Communication**.

privées

void **configurer ()**

Méthode qui configure le client Mqtt , à partir du fichier beehoneyt.ini.

void **sauvegarder ()**

Méthode qui sauvegarde la configuration du client Mqtt , dans le fichier beehoneyt.ini.

Attributs privés

QMqttSubscription * **abonnementMqtt**

QMqttClient * **clientMqtt**

QString **hostname**

MesureRuche **mesure**

QString **password**

int **port**

QString **username**

Les connecteurs et signaux

Connecteurs publics

void **changerEtat** ()

Méthode qui signale l'état du client Mqtt à l'IHM. Plus de détails...

void **recevoirMessage** (const QByteArray &**message**, const QMqttTopicName &**topic**)

Méthode qui reçoit et traite le message du topic, et signale les nouvelles mesures à l'IHM. Plus de détails...

Signaux

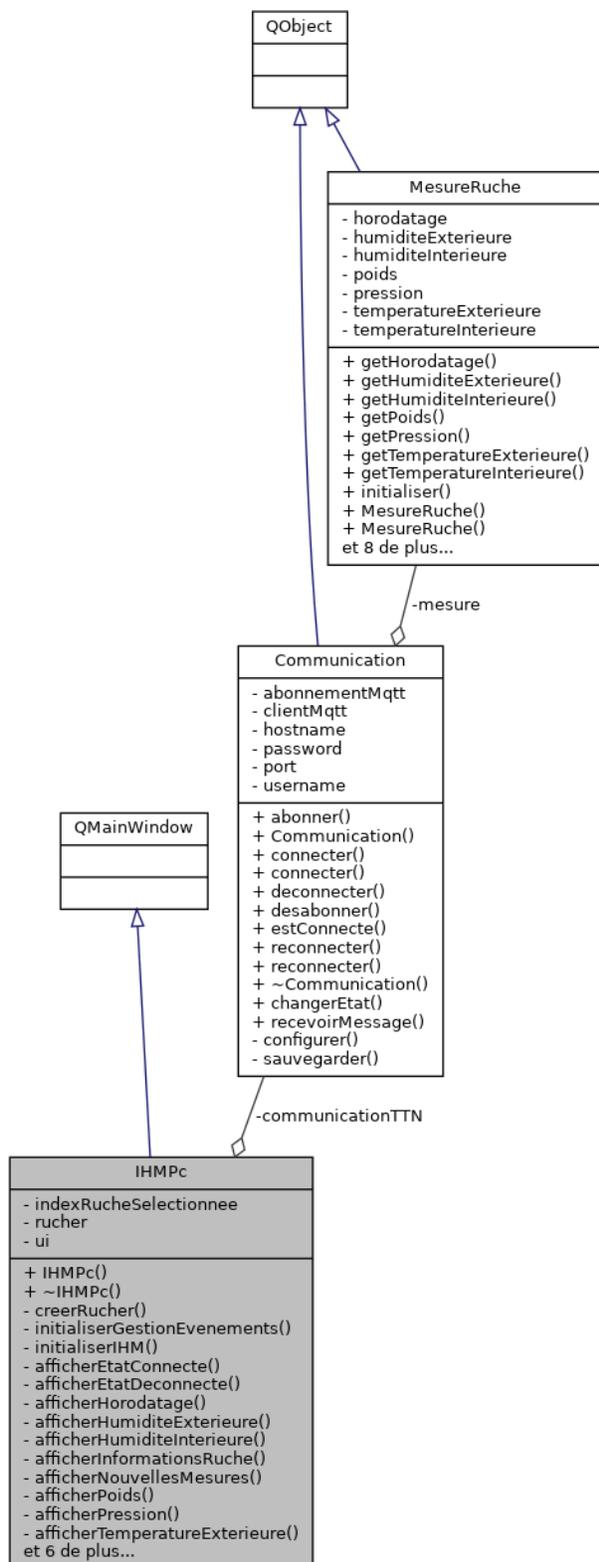
void **messageJournal** (QString **message**)

void **nouvellesMesures** (MesureRuche **mesure**)

void **ttnConnecte** ()

void **ttnDeconnecte** ()

Diagramme de la classe IHMPc



Les fonctions membres de la classe

publiques

IHMPC (QWidget *parent=nullptr)

Constructeur de la classe **IHMPC**.

~IHMPC ()

Destructeur de la classe **IHMPC**.

privées

void **creerRucher** ()

Méthode qui crée un rucher.

void **initialiserGestionEvenements** ()

Méthode qui initialise les événements signals/slots.

void

initialiserIHM ()

Méthode qui initialise l'IHM.

Les attributs privées

Communication *

communicationTTNPointeur sur l'objet **Communication**.

int

indexRucheSelectionnee

L'index courant de la ruche sélectionnée.

QVector< Ruche * >

rucher

Les ruches de l'apiculteur. Plus de détails...

Ui::IHMPc *

ui

Association vers l'interface utilisateur (Qt Designer)

Les connecteurs privés

void **afficherEtatConnecte** ()

Méthode qui affiche l'état de la connexion (connecté), à partir d'une image en haut à droite.

void **afficherEtatDeconnecte** ()

Méthode qui affiche l'état de la connexion (déconnecté), à partir d'une image en haut à droite.

void **afficherHorodatage** (QString **horodatage**)

Méthode qui affiche l'horodatage.

void **afficherHumiditeExterieur** (double **humiditeExterieur**=0., QString **uniteHumidite**=QString("%")) const

Méthode qui affiche l'humidité intérieure.

void **afficherHumiditeInterieur** (double **humiditeInterieur**=0., QString **uniteHumidite**=QString("%")) const

Méthode qui affiche l'humidité intérieure.

void **afficherInformationsRuche** ()

Méthode qui affiche les informations de la ruche.

void **afficherNouvellesMesures** (**MesureRuche** **mesure**)

Méthode qui affiche les nouvelles mesures (slot).

void **afficherPoids** (double **poids**=0., QString **unitePoids**=QString(" Kg")) const

Méthode qui affiche le poids.

void **afficherPression** (double **pression**=0., QString **unitePression**=QString(" pHa")) const

Méthode qui affiche la pression.

void **afficherTemperatureExterieur** (double **temperatureExterieur**=0., QString **uniteTemperature**=QString(" °C")) const

Méthode qui affiche la température extérieure.

void **afficherTemperatureInterieur** (double **temperatureInterieur**=0., QString **uniteTemperature**=QString(" °C")) const

Méthode qui affiche la température intérieure.

void **gererConnexionTTN** ()

void **journaliser** (QString message)

void **ouvrirFenetreAjouter** ()

Méthode qui ouvre une fenêtre de dialogue pour ajouter une ruche : nom et deviceID.

void **selectionnerRuche** ()

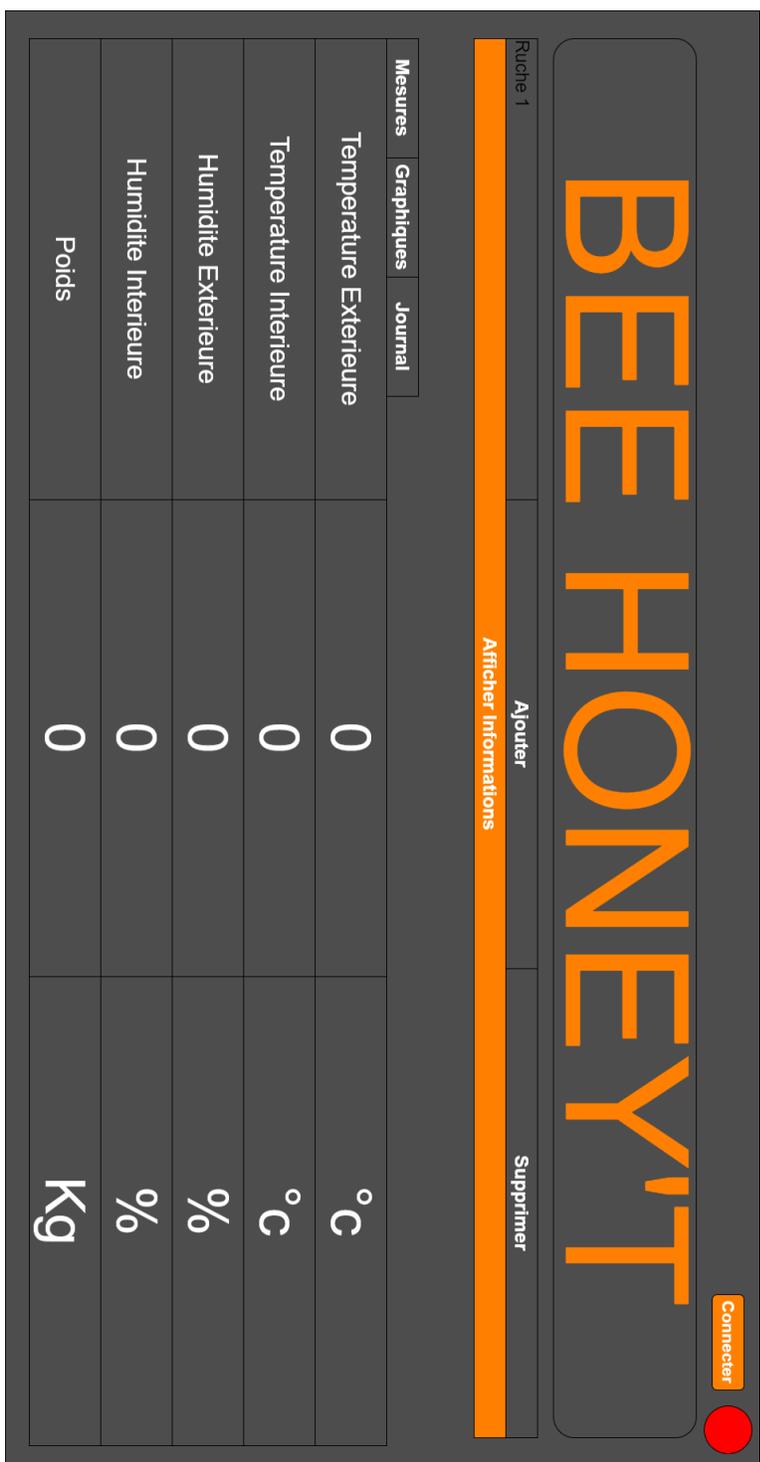
Méthode qui sélectionne la ruche (s'abonne) et se désabonne de l'ancienne : réinitialisation des valeurs.

void **supprimerRuche** ()

Méthode qui supprime une ruche, dans la **listeDeRucheSelectionnee** et dans le vector **rucher<>**

Maquette IHM

Voici la maquette de l'interface :



Tests de validation

Désignation	Démarche à suivre	Résultat attendu	oui/non	Remarques
S'authentifier	Nom d'utilisateur et mot de passe permettant un accès sécurisé		NON	
Créer une ruche	Cliquer sur le bouton "Ajouter" et configurer les paramètres de la ruche		OUI	
Sélectionner une ruche	Cliquer sur la ruche que nous voulons consulter (choix)		OUI	
Recevoir les données d'une ruche	Une fois la ruche sélectionnée et afficher les mesures		OUI	
Supprimer une ruche	Cliquer sur le bouton "Supprimer" et confirmer la suppression de la ruche sélectionnée		OUI	
Gestion des alertes	Visualiser les alertes à l'aide de plusieurs icônes		NON	

Informations

Auteur	Mhadi Zakariya : zakariya.mhadi@gmail.com
Date	2021
Version	1.0
SVN	https://svn.riouxsvn.com/beeoneyt-2021

GLOSSAIRE

Général

une méthode : une méthode est une routine membre d'une classe.

un attribut : un attribut n'est ni plus ni moins qu'une variable.

un objet : un objet est créé à partir d'un modèle appelé classe ou prototype, dont il hérite les comportements et les caractéristiques.

POO : La programmation objet consiste à définir des objets logiciels et les faire interagir entre eux.

une classe : la déclaration d'une classe regroupe des membres, méthodes et propriétés (attributs) communs à un ensemble d'objets. La classe déclare, d'une part, des attributs représentant l'état des objets et, d'autre part, des méthodes représentant leur comportement.

un pointeur : un pointeur est une variable qui stocke l'adresse d'une autre variable.

une méthode statique : est une méthode qui appartient à la classe mais pas aux objets instanciés à partir de la classe.

un constructeur : le constructeur est la fonction membre appelée automatiquement lors de la création d'un objet (en statique ou en dynamique). Cette fonction membre est la première fonction membre à être exécutée, il s'agit donc d'une fonction permettant l'initialisation des variables.

bibliothèque logicielle : une bibliothèque logicielle est une collection de routines, qui peuvent être déjà compilées et prêtes à être utilisées par des programmes.

Trame: une trame est la structure de base d'un ensemble de données encadré par des bits de début et des bits de fin.

protocole publish-subscribe: Publish-subscribe (littéralement : publier-s'abonner) est un mécanisme de publication de messages et d'abonnement à ces derniers dans lequel les diffuseurs (publisher, littéralement éditeurs) ne destinent pas a priori les messages à des destinataires (subscriber, littéralement abonné).

fichier de configuration : Un fichier possédant l'extension .INI (abréviation d'*initialization*) est généralement un fichier de configuration au format texte définissant les paramètres d'une application ou d'une partie du système d'exploitation.

diagramme de séquence : les diagrammes de séquences sont la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique dans la formulation Unified Modeling Language.

diagramme de classe : le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que leurs relations. Ce diagramme fait partie de la partie statique d'UML, ne s'intéressant pas aux aspects temporels et dynamiques.

diagramme de cas d'utilisation : Les diagrammes de cas d'utilisation sont des diagrammes UML utilisés pour une représentation du comportement fonctionnel d'un système logiciel. Ils sont utiles pour des présentations auprès de la direction ou des acteurs d'un projet, mais pour le développement, les cas d'utilisation sont plus appropriés.

Spécifique

TCP/IP : La suite des protocoles Internet est l'ensemble des protocoles utilisés pour le transfert des données sur Internet. Elle est souvent appelée TCP/IP, d'après le nom de ses deux premiers protocoles : TCP et IP.

Data Storage : "stockage d'information"

UTF-8 : est un codage de caractères. Il attribue à chaque caractère Unicode existant une séquence de bits précise que l'on peut également lire comme un nombre binaire.