I. Guide de prise en main du pilote Android:

Nous allons commencer par configurer l'environnement pour la prise en main du pilote.

1. Téléchargement d'android studio :

Dans cette étape nous allons installer l'outil android studio qui permet de faire le développement d'application mobile.

Pour cela vous devez vous rendre sur le site officiel d'android studio à l'adresse ici , ensuite vous devez cliquer sur le bouton "Download Android Studio" pour télécharger l'installateur. N'oubliez pas de sélectionner la version adaptée à votre système d'exploitation.

2. Installation d'android studio :

Lancer le fichier d'installation téléchargé et suivez les instructions pour installer Android Studio sur votre système. Configurer les paramètres d'installation selon vos préférences. N'oubliez pas d'inclure l'émulateur si vous ne souhaitez pas utiliser un périphérique android extérieur. C'est sur cet émulateur (ou sur l'appareil android extérieur) que vous installerez l'application.

Une fois que celà est effectué nous allons maintenant prendre en main le code existant sur le dépôt gitlab.

3. Clone du projet :

Vous pouvez cloner le projet depuis android studio :

- Dans la barre de menus, sélectionnez "VCS" > "Get from Version Control" > "Git".
- Dans la fenêtre qui s'ouvre, collez l'URL du dépôt Git contenant votre code.
- Spécifiez le répertoire de destination où vous souhaitez cloner le projet.
- Cliquez sur "Clone" pour commencer le processus de clonage

Vous pouvez aussi le cloner dans un répertoire sur votre machine et après l'ouvrir dans Android Studio.

4. Configuration et exécution du projet :

Une fois le projet importé attendez que android studio termine l'indexation du projet et la synchronisation des dépendances.

- Vérifiez les fichiers de configuration tels que **build.gradle** pour vous assurer que les dépendances sont correctement spécifiées.

- Configurez les paramètres du projet, tels que la version de compilation, le SDK cible, etc; Ici nous utilisons la version **7.2 de gradle et un jdk 1.8** pour compiler le code.

Ensuite une fois la configuration finie vous exécuterez le projet en cliquant sur le bouton “RUN” en fonction de l’émulateur où du périphérique extérieur choisi.

5. Code java pour robot :

Maintenant ouvrez le projet pour téléverser le code java. Pour ce faire, utilisez éclipse pour l’ouvrir et exécuter. Assurez-vous d’avoir lejos installé sur votre système et que toutes les dépendances sont bien installées, de plus veillez à exécuter le code java avec un jdk 1.7 ou 1.8, et que vous avez installé le plugin LeJOS pour Eclipse ainsi que le logiciel LeJOS EV3 dans votre machine de développement.

II. Communications MQTT - Bluetooth:

1. Problèmes connus avant la mise en oeuvre de la communication:

Après avoir repris le travail TER précédent, ayant comme résultat une application android qui se connecte en Bluetooth avec le robot EV3, et un programme java qui a pour rôle de réception les communications depuis l’application.

- L’application a des problèmes de performances pour les appareils de génération moyenne (freezes).
- L’application requiert de saisir l’adresse MAC manuellement pour se connecter à un robot.
- L’application ne gère pas proprement la déconnexion du bluetooth et les problèmes de perte de connexion.
- Lors du contrôle du robot, la vitesse n’est pas synchronisée entre l’interface utilisateur et le robot.
- Le contrôle de direction par flèches est instable.
- Responsabilité non supportée. Des éléments de l’interface peuvent être masqués si on utilise un smartphone au lieu d’une tablette.
- Pas de commentaires dans le code source Android.
- Le programme du robot n’a pas de sortie, on devrait l'arrêter depuis un PC ou forcer un redémarrage du robot en cas de plantage ou de perte de connexion.
- Fonctionnement instable du capteur d’obstacles.
- Duplications dans le code source.

2. Implémentation des deux communications en parallèle:

Dans le code Java du robot, pour implémenter les deux communications en même temps, on aurait besoin pour chacun un service et l'exécuter dans un thread séparé.

a. MainMQTT_BT (classe principale):

La classe configure deux threads pour les connexions Bluetooth et MQTT.
MQTT_SERVER_IP l'adresse IP du serveur MQTT.

Le **clientId** est un identifiant unique pour le client MQTT. Il est construit en ajoutant l'adresse MAC de l'EV3 à la chaîne *"EV3_"*.

La méthode **main** est le point d'entrée du programme. Il crée d'abord un nouvel objet **Controller**.

Ensuite, il démarre les deux threads: **T1** pour la connexion Bluetooth et **T2** pour la connexion MQTT. La connexion MQTT est établie en créant un nouvel objet **MQTTConnect** avec l'adresse IP du serveur, l'ID client et le topic comme paramètres.

Le programme entre ensuite dans une boucle où il vérifie si les deux threads sont en cours d'exécution ou si le bouton DOWN de l'EV3 est enfoncé. Si ce dernier est le cas, le programme se terminera. Cela permettra de quitter le programme plus facilement si besoin.

Si une erreur se produit pendant l'exécution du programme, il imprimera le message d'erreur et la trace de la pile, puis quittera le programme. La console du robot peut être consultée depuis EV3 Control Center dans Eclipse.

---

b. BT_Connect:

Ce service est chargé d'établir et de gérer une connexion Bluetooth vers l'EV3.

La méthode **connect** est responsable de l'établissement de la connexion Bluetooth. Il imprime d'abord un message sur la console indiquant qu'il attend un client Bluetooth (qui se produira normalement depuis l'application EV3).

L'objet **NXTCommConnector** permet d'attendre une connexion depuis un client Bluetooth (A noter que son instanciation est bloquante). Une fois la connexion établie, un **DataInputStream** est ouvert à partir de la connexion. Ce flux est utilisé pour lire les données envoyées via la connexion Bluetooth depuis l'application Android.

La méthode **executeCommand** prend un objet **Controller** et une valeur int comme paramètres. On utilise une instruction switch pour exécuter différentes méthodes sur l'objet **Controller** en fonction de la valeur reçue. La réception d'une action dépend de la variable

statique **BT_disconnected** qui varie selon des messages MQTT spécifiques. (Dans ce cas, on peut désactiver temporairement la communication Bluetooth avec MQTT)

La méthode **run** est la méthode requise par l'interface **Runnable**. Il appelle simplement la méthode **connect.**

---

c. Controller:

Cette classe est chargée de contrôler les mouvements d'un robot à l'aide de deux moteurs, un moteur gauche et un moteur droit. La classe **Controller** possède plusieurs variables d'instance. **actualState** est une énumération d'état qui assure le suivi de l'état actuel du robot. **leftMotor** et **rightMotor** sont des objets Motor qui représentent les moteurs gauche et droit du robot. **ratioLeft** et **ratioRight** représentent les niveaux de vitesse des moteurs gauche et droit respectivement. **initial_speed** est la vitesse initiale des moteurs.

Le constructeur initialise les moteurs gauche et droit avec les objets **EV3LargeRegulatedMotor** attachés respectivement aux ports C et B.

La méthode **movingForward** fait avancer le robot. Il vérifie l'état actuel du robot et ajuste la vitesse des moteurs en conséquence. Si le robot est arrêté ou tourne, il règle la vitesse des deux moteurs à la vitesse initiale et change l'état en **FORWARD**.

La méthode **movingBackward** fait reculer le robot. Il vérifie si le robot est actuellement arrêté, et si c'est le cas, il règle les moteurs pour qu'ils reculent et changent l'état en **BACKWARD**.

La méthode **stop** arrête le robot en arrêtant les deux moteurs et en changeant l'état en **STOPPED**.

Les méthodes **turnLeft** et **turnRight** font tourner le robot respectivement à gauche et à droite. Ils ajustent la vitesse des moteurs pour réaliser le virage et changent l'état en **TURNING_LEFT** ou **TURNING_RIGHT**.

Les méthodes **accelerate** et **decelerate** augmentent et diminuent respectivement la vitesse des moteurs.

---

d. MQTT_Connect:

Ce service est chargé d'établir et de gérer une connexion à un broker MQTT, de s'abonner à un topic et de gérer les messages entrants à l'aide du client Eclipse Paho. Le nom d'utilisateur et le mot de passe contiennent les informations d'identification du serveur MQTT.

Pour des raisons de sécurité, l'utilisation de cette authentification est cruciale même en environnement de développement.

Le constructeur **MQTTConnect** prend trois paramètres : l'adresse IP du broker, l'ID client et le topic à laquel s'abonner. Il construit d'abord l'URL du broker en ajoutant l'adresse IP et le port à *" tcp:// "*. Ensuite, il crée un objet **MqttClient** avec l'URL du serveur et l'ID client.

L'objet **MqttConnectOptions** est utilisé pour définir les options de connexion pour le client.

Le client se connecte ensuite au serveur, imprime un message sur la console EV3 indiquant qu'il est connecté et s'abonne au topic spécifié. Il définit également un callback **ListenToMQTT**.

ListenToMQTT implémente l'interface **MqttCallback**. Il fournit trois méthodes : **connectionLost**, **messageArrived** et **deliveryComplete**. La méthode **connectionLost** est appelée lorsque la connexion au serveur est perdue. On imprime simplement un message sur la console.

La méthode **messageArrived** est appelée lorsqu'un message arrive sur le topic abonné. Ensuite, on utilise un switch pour exécuter différentes actions en fonction du message. Par exemple, si c'est *"go"*, on appelle la méthode **movingForward.**

---

3. Envoi des messages Bluetooth:

Du côté application android, Il fallait améliorer quelques parties et ajouter des fonctionnalités supplémentaires.

a. MainActivity:

La classe **MainActivity** contient une liste **activities** et une map **name2class** qui mappe le nom d'une activité à sa classe correspondante. Ceux-ci permettent de gérer les différentes activités pouvant être lancées à partir de cette activité principale.

Dans la méthode **onCreate**, appelée lors du démarrage, la méthode **setup_activities** est appelée pour remplir la liste et la map avec les activités. Un **ArrayAdapter** est ensuite défini sur **ListView** pour afficher la liste des activités.

La méthode **setup_activities** ajoute trois activités à la liste et à la carte : *"Connexion avec Bluetooth", "Connexion avec Wifi" et "Paramètres MQTT"*. Chaque activité est associée à une classe spécifique.

Un **OnItemClickListener** est défini sur **ListView** dans la méthode **onCreate**. Ce listener est déclenché lorsqu'on clique sur un élément de la liste. La méthode **onItemClick** du listener récupère le nom de l'activité cliquée, récupère sa classe correspondante de la map et la démarre.

Enfin, la méthode **onBackPressed** est implémentée pour appeler **finishAffinity()**. Cette méthode ferme non seulement l'activité en cours mais également toutes les activités faisant partie de sa tâche.

---

b. BluetoothService:

Le service **BluetoothService** est utilisé pour gérer les connexions Bluetooth dans une application Android. Il utilise la classe **BluetoothAdapter** pour interagir avec le matériel Bluetooth de l'appareil android. La classe commence par définir une chaîne **SPP_UUID** et deux variables d'instance, **localAdapter** de type **BluetoothAdapter** et **socket** de type **BluetoothSocket**.

Il existe deux constructeurs pour cette classe. Le 1er prend un objet **Context** comme paramètre et initialise le **localAdapter** à l'aide de la méthode **initBT**. Le 2eme constructeur prend à la fois un objet **Context** (pour accéder aux ressources d'une activité spécifique) et un objet **Device** comme paramètres. Il initialise **localAdapter** de la même manière, mais tente également de créer un **BluetoothSocket** pour le périphérique spécifié.

La méthode **initBT** est utilisée pour initialiser le **BluetoothAdapter**. Il vérifie la version Android et utilise la méthode appropriée pour obtenir le **BluetoothAdapter**.

La méthode **getPairedBluetoothDevices** renvoie une liste **Device** représentant tous les appareils Bluetooth couplés. Il utilise la méthode **getBondedDevices** de **BluetoothAdapter** pour obtenir un ensemble d'objets **BluetoothDevice**, puis les mappe aux objets **Device** et les ajoute à une liste. A noter qu'il faudra avoir déjà couplé l'EV3 dans les paramètres Bluetooth afin que celui-ci soit dans la liste et ainsi détecté dans l'application.

La méthode **connectToDevice** tente d'établir une connexion au robot. Il crée un **BluetoothSocket** pour l'appareil et tente de s'y connecter. Si la connexion réussit, elle renvoie **true,** sinon, il enregistre l'erreur et renvoie **false**.

La méthode **deconnect** tente de fermer le **BluetoothSocket**, en se déconnectant ainsi de l'appareil.

La méthode **writeMessage** est utilisée pour envoyer un message au robot. Il écrit le message dans le **OutputStream** du **BluetoothSocket**. C'est la source des messages **InputStream** lues dans le programme EV3 mentionné précédemment.

La méthode **checkBTPermissions** vérifie si l'application dispose des autorisations nécessaires pour utiliser Bluetooth. Sinon, il demande ces autorisations.

Enfin, la méthode **getNameOfDevice** renvoie le nom du périphérique local.

---

c. ConnectionBluetoothActivity:

Cette classe permet d'initier la connexion Bluetooth vers un robot EV3. Elle commence par définir plusieurs variables d'instance, notamment **BTConnect** de type **BluetoothService**, **LV** de type **ListView**, **btnRefresh** de type **Button…**

La méthode **onCreate** est appelée au démarrage de l'activité. Il initialise les variables d'instance, vérifie les autorisations Bluetooth et configure la connexion Bluetooth et la source de données du ListView.

La méthode **getPairedDevices** est utilisée pour obtenir une liste des appareils Bluetooth couplés. Si Bluetooth n'est pas activé, il envoie une requête à l'utilisateur pour l'activer. Sinon, il renvoie la liste des appareils couplés depuis le **BluetoothService**. Lors du click sur un item de **LV**, un **Intent** est créé pour passer l'appareil correspondant à l'activité **ControlPageActivitiy** autant qu'objet **Device**.

La méthode **updateBluetoothDevices** est utilisée pour mettre à jour ListView avec la liste actuelle des appareils Bluetooth couplés.

La méthode **onRequestPermissionsResult** est invoquée lorsque l'utilisateur répond à la boîte de dialogue de demande d'autorisation. Si l'autorisation est accordée, il met à jour ListView avec la liste actuelle des appareils Bluetooth couplés.

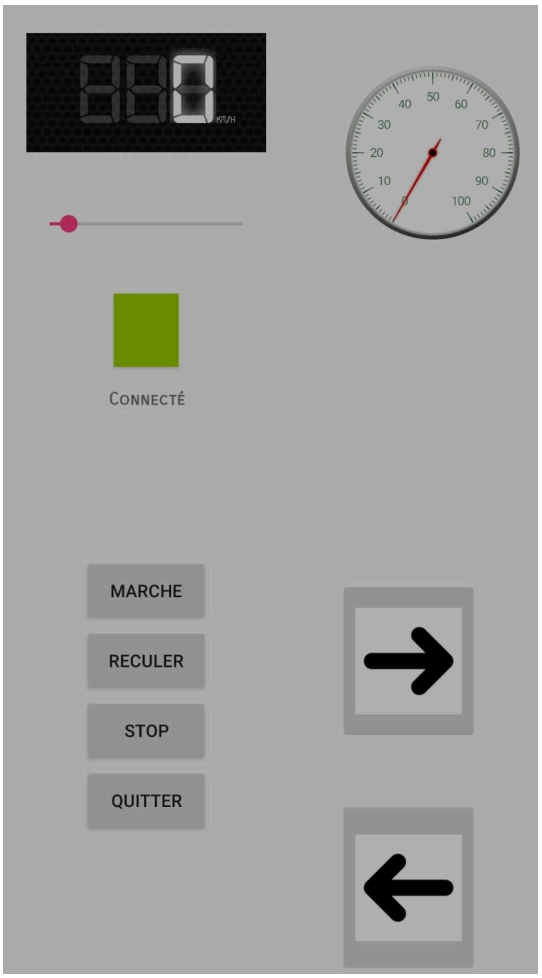
La méthode **onActivityResult** est appelée lorsqu'une activité que vous avez lancée se termine, vous donnant le **requestCode** avec lequel vous l'avez démarrée, le **resultCode** qu'elle a renvoyé et toutes les données supplémentaires qu'elle contient. Il gère le résultat de la demande d'activation Bluetooth.

Enfin, la méthode **onBackPressed** est invoquée lorsque l'activité a détecté l'appui de l'utilisateur sur la touche retour. Il appelle simplement **finish()** pour fermer l'activité.

---

d. ControlPageActivity:

Dans cette activité, le **BluetoothService** est initialisé avec un objet **Device** extrait de l' **Intent** qui a démarré cette activité. Si la connexion à l'appareil échoue ou si les autorisations Bluetooth ne sont pas accordées, un message **toast** s'affiche à l'attention de l'utilisateur et l'activité est terminée.



OnSeekBarChangeListener est défini sur l'élément **SeekBar**. Dans la méthode **onProgressChanged**, en fonction de la progression de **SeekBar**, différents messages sont écrits sur le périphérique Bluetooth et la **vitesse** actuelle est mise à jour.

Les événements **OnClickListener** sont définis sur les boutons **btnAvancer**, **btnReculer**, **btnGauche**, **btnDroite**, **btnStop** et **btnExit**. En fonction du bouton cliqué, différents messages sont écrits sur l'appareil Bluetooth et la vitesse actuelle est mise à jour.

Enfin, la méthode **onDestroy** est surchargée pour déconnecter le service Bluetooth lorsque l'activité est détruite.