UNIVERSITÉ PARIS SUD | IUT DE CACHAN

université PARIS-SACLAY

Advanced Market

Solutions d'aides à la préparation de commande

Antoine Braut et Bastien Duprey | Advanced Market | version : 1.0 | 21/06/19

Table des matières

Résumé

Advanced Market est un projet comportant 3 solutions à un même problèmes : concevoir un systèmes permettant de simplifier la préparation des paniers client dans des structures tels que des entrepôts.

Le projet est divisé en trois parties : un serveur de gestion des paniers appelé PSaMOS, un microcontrôleur embarqué dans un « bracelet » nommé MegaMarket2 et une application Android : MarketDroid.

Abstract

Advanced Market is 3 solutions for one problematic: how to ease the process of preparing client's card.

The project is divided into 3 parts: A card manager server which is called PSaMOS, a microcontroller embedded in a bracelet named MegaMarket2 and an Android application: MarketDroid.

Introduction

Le projet Advanced Market répond à la problématique suivante : Comment faciliter la préparation de paniers dans un entrepôt à l'aide de solution simple d'utilisation ?

Pour cela il était demandé dans un premier temps de réaliser deux IHM[1]. La première sous forme d'une application Android pouvant être utilisé sur n'importe quelle smartphone ou tablette Android. La deuxième sous la forme d'un microcontrôleur équipés d'un écran (cette partie sera nommée « le Bracelet » dans la suite du rapport). Afin de valider les articles dans le panier, chaque IHM est relié par Bluetooth à un scanner de code barre. Il a ensuite été imaginé un serveur permettant de récupérer les commandes passées par les clients et de les redistribué sous formes de « paniers[2] ».

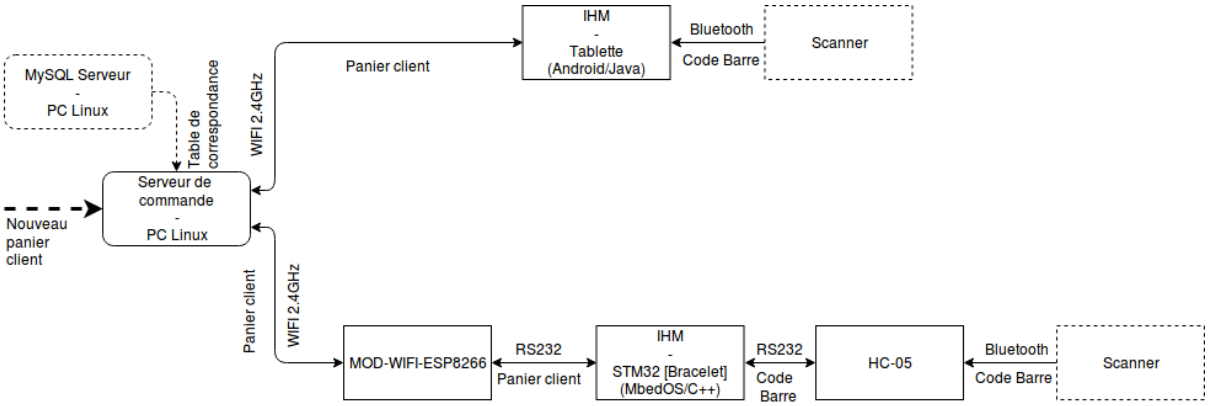

Figure 1. Schéma synoptique d'Advanced Market

Le serveur est conçu pour être un serveur multi-client. Chaque IHM, aussi appelé terminal, peut venir s'y connecter pour échanger des paniers avec lui. Une fois l'échange de panier effectué, chaque terminal est capable d'opérer en mode hors-ligne jusqu'à la validation d'un panier remplie. Cette fonctionnalité est prévue afin de faciliter le déploiement d'Advanced Market dans des zones contraignante en termes de couverture Wi-Fi, comme par exemple de grand entrepôt. De plus chaque partie à été conçue pour être indépendante. Il est tout à fait possible de ne déployer qu'une partie des solutions proposé par Advanced Market, comme par exemple uniquement le serveur et le bracelet ou juste l'application avec un serveur personnalisé. Le protocole de communication étant « Open Source[3] », il est possible de substituer une des partie avec sa propre solution.

[1] Interface Homme-Machine

[2] Le terme « commande » étant source de confusion, le terme « panier » est privilégié une fois arrivé au serveur.

[3] L'ensemble du projet est sous licence GPLV3

1. Le serveur de gestion des paniers : PSaMOS

Afin de distribuer les paniers vers les différents terminaux, un serveur a été imaginé. Ce serveur sert de "point d'entré" à tous le système d'Advanced Market, autrement il a été pensé de sorte que n'importe qu'elle source indépendante (site de e-commerce, borne de commande, logiciel de gestion ...) puissent créer un panier et le mettre en préparation en passant par le serveur. Il se charge ensuite de redistribuer tous paniers en attente à la demande d'un terminal. Le serveur se nomme PSaMOS (Powerfull Server and Market Oriented System) en clin d'oeil à GLaDOS (Genetical Lifeform and Disk Operating System) du jeu Portal.

1.1 L'architecture du serveur et la lib easyTcpSockeet

Afin de pouvoir répondre aux demandes de plusieurs terminaux en même temps, le serveur se doit d'être multi-client et multi-tâche. PSaMOS n'ayant pas été conçu pour être "cross-platform" mais pour tourner spécifiquement sur une plate-forme Linux[4], il a été choisi de passer par des sockets POSIX afin de mettre en place le serveur TCP/IP. Les sockets POSIX sont une implémentation au sein du noyeau du système (ici le Kernel Linux) d'un "module" permettant de gérer plusieurs protocole de communications. Ces sockets sont capable de gérer plusieurs protocoles de communications, le TCP/IP dans le cadre de PSaMOS (AF_INET + SOCK_STREAM), mais aussi par exemple les communications locales (AF_UNIX ou AF_LOCAL), le Bluetooth (AF_BLUETOOTH) ou même le CAN (AF_CAN).

Plusieur Bibliothéques C permettent de faire des appels systèmes ou "ioctl", permettant de d'utiliser les sockets en passant par le Kernel. Lorsqu'un socket est créée suite à un appel système, il crée un fichier qui servira de "driver" entre lui et le programme qui l'a appeler.

```
28  void tcp_server::_start_listening()
29  {
30    server_socket = socket(AF_INET,SOCK_STREAM,0);
31
32    memset(&_server,0,sizeof(_server));
33
34    _server.sin_family=AF_INET;
35    _server.sin_addr.s_addr=htonl(INADDR_ANY);
36    _server.sin_port=htons(server_port);
37
38    bind(server_socket,(struct sockaddr *)&_server, sizeof(_server));
39    listen(server_socket,easyTCP::MAXCONNECTIONS);
40
41    listener_task = std::thread(&tcp_server::_listen_for_client,this);
42
43
44  }
```

Figure 2. Création du server et du socket (easyTcpServer.cpp)

Une fois le socket créée il suffit de venir lire ou écrire ce fichier, toujours à l'aide d'ioctl, pour effectuer une communication.

```
113                 if ((valread = read( sd , buffer, 1024)) == 0)
```

Figure 3. Lecture du socket (easyTcpServer.cpp)

[4] Ubuntu 18.04 lors de la phase de test

Les sockets peuvent opérer avec 2 modes différents. Le mode bloquant dans lequel toute lecture du socket bloque le programme jusqu'à l'obtention d'une réponse. Et le mode non bloquant qui à l'inverse ne bloque pas le programme et donne juste une réponse vide lorsqu'il n'y en a pas. PSaMOS devant être capable de gérer plusieurs clients en même temps, il est donc nécessaire de ne pas être bloquer à chaque attente de réponse d'un client. Pour cela il est possible d'utiliser le mode non bloquant des socket, cependant le passage du socket en Asynchrone rends la synchronisation avec le client plus compliqué, et donc les séquences d'échanges aussi. Afin de pouvoir garder les sockets en mode bloquant tout en assurant le caractére multi-client du serveur, il a été choisi d'implémenter les actions liée au serveur sur des « tâches parallèle ». Pour cela une bibliothéque en C++ a été réalisé : easyTcpSocket[5]. Cette bibliothéque englobe toute les fonctions C nécessaire à la gestion du serveur dans une classe. Elle automatise aussi le démarrage des threads pour la reception et l'émission. Afin de ne pas démarrer un thread de réception pour chaque client, la fonction « select[6]() » est préféré à la fonction « Read() », elle permet de venir scruter l'activité de tous les sockets enregistré et ainsi de tous les gérer sur une seule tâche.

1.2 Le fonctionnement de PSaMOS

PSaMOS fonctionne autour de deux tâches principales, Une tâche dédiée aux actions du serveur (Envoyer, recevoir, enregistrer de nouveau client …). Et une tâche dédiée à l'échange avec un client selon les protocoles définies.

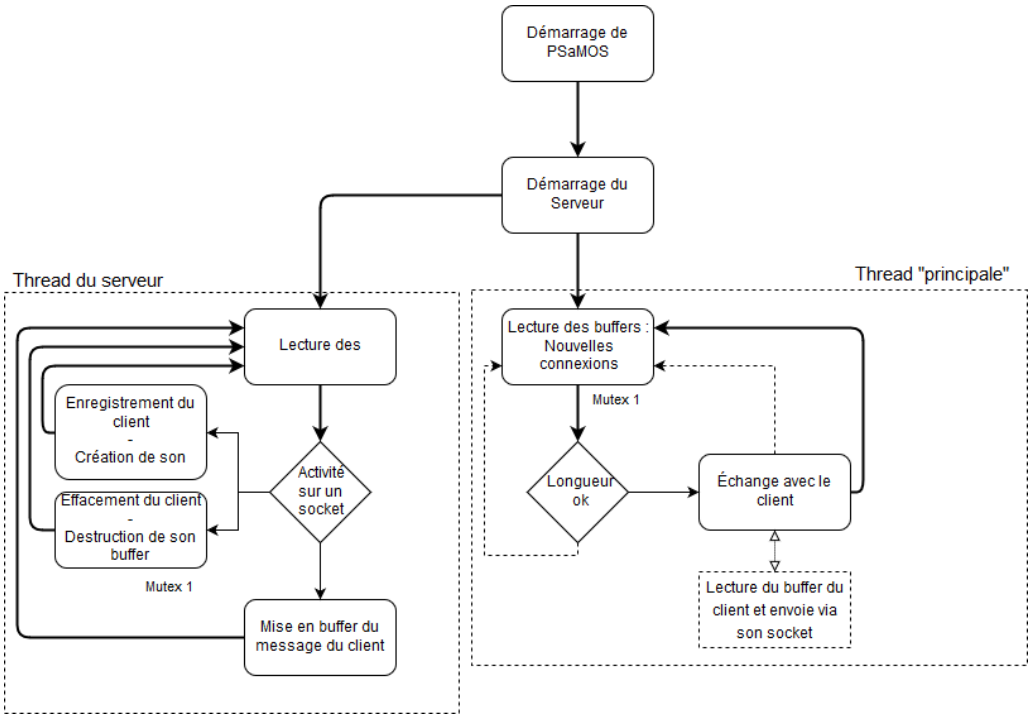

Figure 4. Architecture simplifié de PSaMOS

[5] Plus d'information : https://github.com/Brautantoine/easyTcpSocket
[6] Plus d'information sur Select : https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-clients-on-server-without-multi-threading/(en) et http://manpagesfr.free.fr/man/man2/select.2.html(fr)

Au démarrage, PSaMOS se sépare en deux parties. La partie dédiée au serveur va venir boucler sur le lecture de l'activité des Socket. Pour chaque action sur un Socket (Connection d'un nouveau client, réception d'un nouveau message, déconnection d'un client …), la tâche remplie un buffer (FIFO) correspondant à l'action avec la nouvelle donnée.

En parallèles dans le Thread « principale », le programme va boucler sur la lecture de tous les buffers remplie par la tâche du serveur. Lorsqu'un message avec une longueur suffisante est détecté, le client qui a émis ce message devient prioritaire[7]. Si le message reçus est une demande d'échange selon l'un des protocoles définies, un échange commence entre le client et le serveur jusqu'à la fin du protocole ou la détection d'une erreur. Pour mener à bien cette échange, la tâche principale utilise les buffers d'émission et de réception de la tâche du serveur.

1.3 Les protocoles d'échanges

Pour pouvoir faire communiquer le serveur avec les clients, des protocoles ont été mis en place pour chaque action que le serveur peut réaliser.

- Envoie d'un nouveau panier :

Ce protocole permet à un terminal de demander un nouveau panier au serveur. Il commence avec une trame de demande de panier (ID_NPANIER [0x1001]). Cette identifiant se retrouvera au début de toutes les trames correspondant à cette échange.

Une fois la demande de panier reçue par le serveur, celui-ci renvoie une trame avec la composition du nouveau panier. Cette trame est sous le format suivant : ID_NPANIER –[8] id_panier – qt_panier – X data_panier – STOP avec ID_NPANIER le marqueur d'en-tête (0x1001), id_panier l'id du nouveau panier envoyé, qt_panier la quantité d'article dans ce panier, data_panier le description d'un article du panier sous la forme id_article.qt_article/, et STOP le marqueur de fin (0xFF).

Le client renvoie ensuite une nouvelle trame avec l'en-tête ID_NPANIER, puis la quantité d'article dans le panier reçue afin de confirmer la bonne réception du panier.

Si tout est ok, le serveur envoie une dernière trame de confirmation (ID_NPANIER – data_validation) et l'échange est terminé avec succès, sinon il renvoie une trame de non validation (ID_NPANIER - !data_validation) et l'échange est abandonné.

[7] Pour la phase de test, dans la version définitive cette partie sera détacher sur un nouveau Thread.
[8] Les « - » marque la séparation des différents champs de la trame, ils ne sont pas présents sur le vrai trame.

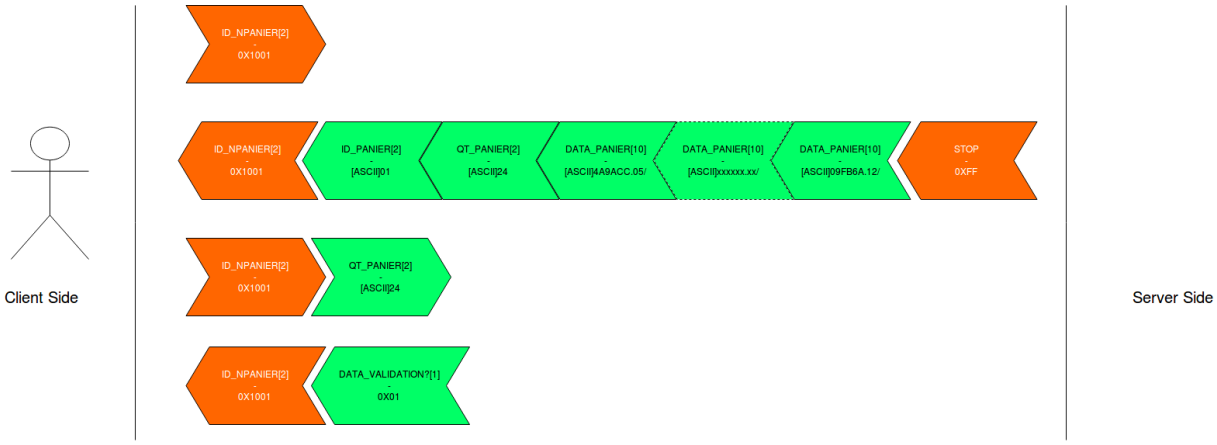

Figure 5. Échange de nouveau panier

- Validation d'un nouveau panier :

Lorsqu'un opérateur a terminé de préparer un panier, il effectue grâce à son terminal la validation de celui-ci auprès du serveur. Pour cela il envoie, dans un premier temps, une trame de demande de validation de panier (ID_VALIDATION [0x1100] – id_panier). Le serveur lui répond alors pour lui confirmer la validation du panier (ID_VALIDATION – data_validation | !data_validation).

- Mise à jour de la table d'un terminal

Les terminaux utilisant des tables de correspondances stocké en local[9], il est nécessaire de pouvoir transmettre les changements fait sur les produits de la base de données aux tables de correspondances des terminaux. Pour cela les terminaux peuvent suivre le protocole de mise à jour de la table. Ils doivent d'abord envoyer une trame de demande (HEADER_TABLE [0x0110]). Dès que le serveur commence à traiter cette demande, il renvoie une trame contenant le nombre de ligne de la nouvelle table (HEADER_TABLE – nb_ligne). Le serveur envoie ensuite une par une les nouvelles lignes du tableau, en respectant le format suivant :

HEADER_TABLE – num_ligne – size_id – id_produit[10] – size_nom – nom_produit - emplacement – STOP

Chaque ligne est suivie d'un acquittement de la part du client (HEADER_TABLE – num_ligne_reçue). Si toutes les lignes ont bien été acquittées par le client, le serveur envoie une trame de fin de table après toutes les lignes (HEADER_TABLE – nb_de_ligne_total) que le client vient acquitter afin de confirmer le bon déroulement de l'échange (HEADER_TABLE – validation_ok | !validation_ok).

[9] Cette partie n'est pas encore implémentée dans le version de démo.
[10] L'id du produit étant obtenue en prenant les 6 premiers caractères du sha1 du nom du produit, il peut être recalculé par le client afin de blinder la communication.

2. Le bracelet : MegaMarket2

L'un des deux terminaux disponible se présente sous la forme d'un « bracelet ». Conçue autour d'une STM32F746-DISCO, il utilise l'écran de celui-ci afin de proposer un IHM pouvant s'intégrer autour du poignet d'où le nom de code « bracelet ». Le nom du produit « MegaMarket2 » n'a pas de signification précise, le choix du chiffre 2 provient de l'abandon du projet MegaMarket(1) à cause d'un problème de linkage d'une variable.

2.1 Les modules de communications

Pour pouvoir communiquer avec le serveur et le scanner, le microcontrôleur dispose de deux modules externes. Un pour le Wi-Fi : l'ESP-8266 et l'autre pour le Bluetooth : l'HC-05

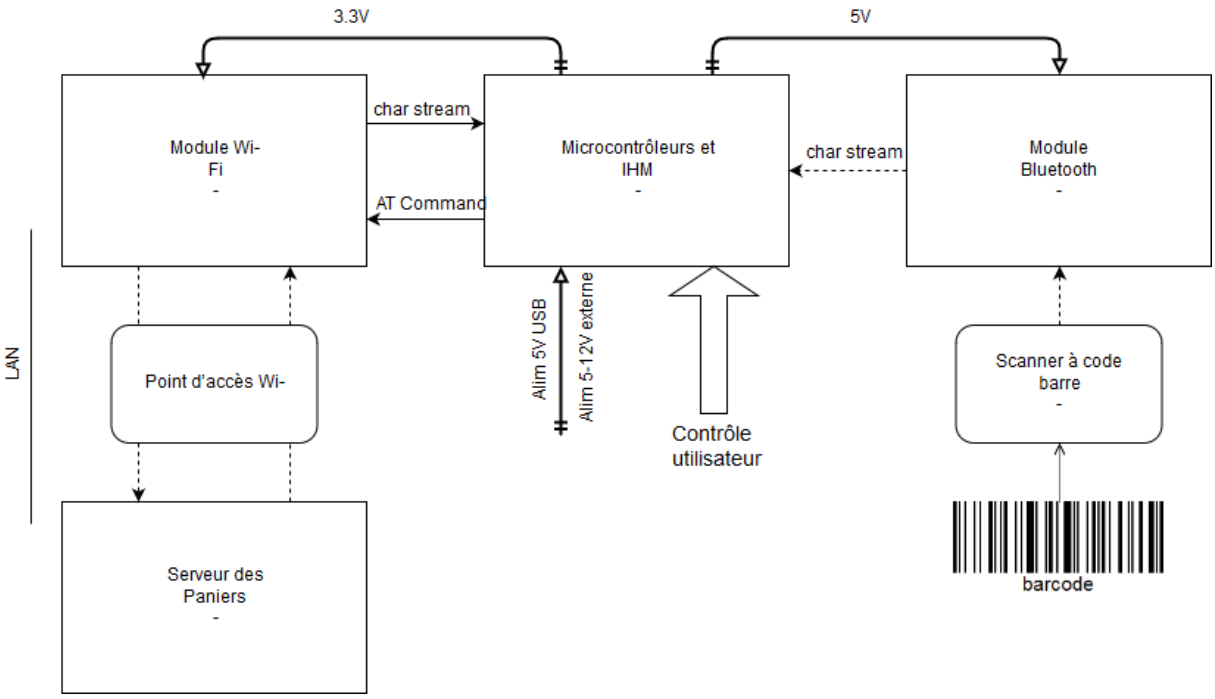

Figure 6. Synoptique hardware de MegaMarket2

Le module Bluetooth à d'abord été préprogrammé une première fois grâce à des commandes AT, afin d'opérer en maitre et de se connecter automatiquement au scanner Bluetooth grâce à son adresse MAC. Il se comporte ensuite comme un simple pont série entre le microcontrôleur et le scanner. Il suffit d'écrire ou de lire sur la liaison série pour communiquer avec l'appareil connecté[11].

Le module Wi-Fi est un peu plus compliqué puisqu'il nécessite d'utiliser des commandes AT pour faire appel à ses fonctions. Afin de pouvoir l'utiliser simplement une classe a été réalisé[12] afin d'automatiser l'écriture des commandes AT sur la ligne série.

[11] Dans le cadre du scanner seul la lecture est importante.
[12] Disponible dans ESP_lib dans le dépôt

2.2 L'architecture du logiciel

L'interface graphique est réalisée grâce à la bibliothèque STM32F7_GUI qui permet la création d'interface « Bloat free » de manière simplifié. Le programme s'articule autour de 2 fonctions principales. Un pour créer et gérer l'interface de la fenêtre d'accueil/connexion et l'autre pour la fenêtre de scan. La fenêtre d'accueil va permettre de remplir le FIFO global « panier » avec des structures « ligne_panier » décrivant le contenu d'une ligne d'un panier. Le remplissage de cette FIFO ce fait grâce à une méthode d'ESP_lib (get_card()) qui implémente le protocole de demande d'un nouveau panier auprès du serveur.

Une fois la fenêtre de scan lancé, le programme va sonder la liaison série du module Bluetooth pour venir capturer les ID lu depuis les codes-barres. A chaque nouveau produit scanné, le programme va avancer dans la FIFO afin d'obtenir les éléments suivant et rajouter des lignes dans l'affichage avec les produits déjà scanné.

Figure 7. Interface de scan

3. L'application Android : MarketDroid

MarketDroid est le nom donné à l'application Android qui répond au projet initial. Cette partie à le même but que la partie 2 mais elle est destinée à une utilisation sur tablette ou smartphone. Nous utilisons donc Android Studio pour développer en Android 8.0 (Oréo) avec l'API 28.

L'application se décompose en deux fenêtres qui sont détaillées dans les deux parties suivantes.

3.1 Le Wi-Fi

Figure 8. Interface Wi-Fi

La première fenêtre permet la connexion au serveur PSaMOS. En effet pour recevoir les paniers il faut que l'application soit connectée au serveur par lequel transite ses informations. Il a donc fallu mettre en place une communication TCP IP entre MarketDroid et PSaMOS.

Pour pouvoir connecter le mobile (Android) Nous avons laissé la possibilité à l'utilisateur de choisir l'adresse IP à laquelle il souhaite se connecter, ce champ est prérempli par l'adresse IP que nous utilisons le plus souvent : 192.168.43.125. Le deuxième paramètre impératif pour la connexion est le port du serveur, nous l'avons imposé : 4242.

La fenêtre de l'application dispose de deux boutons le premier (Connexion WIFI) permet de lancer/relancer une nouvelle connexion au serveur dans les cas où la première tentative à échouer ou si elle a simplement été interrompue ; le deuxième bouto0n (Nouvelle Commande) permet de demander au serveur un nouveau panier.

Afin de gérer la connexion TCP/IP, une tache asynchrone (ConnectTask) est utilisée. Celle-ci permet de créer un client qui initialise la liaison TCP/IP et récupère les messages du serveur en boucle pour les transmettre au reste de l'application. Une fois la connexion établie elle est interrompue uniquement

lors de la demande d'un nouveau panier qui entraine le changement de fenêtre. Une classe TcpClient.java a spécialement été créée pour l'application et regroupe toutes les méthodes nécessaires au fonctionnement de la connexion WIFI.

Le deuxième bouton, la demande d'un nouveau panier, entraine quant à lui le traitement de la réponse du serveur qui correspond à la réception du panier. Les messages respectent un protocole précis et commencent, donc, tous par des caractères définis : 0x1001. En recherchant ces deux caractères dans la réponse du serveur il est alors possible de séparer la liste de produits et leurs quantités pour générer une liste utilisable par la suite du programme (Partie 2.2 Le Bluetooth)

Exemple de réponse du serveur à la suite d'une requête de nouveau panier :

Panier reçu = "**2b034a9acc.02/0e3120.05/aecd97.01\0\n" ;

Composition de la réponse :

** : les 2 caractères fixes : 0x1001, retiré avant d'être transmis à la suite du programme

2b : ID du panier

03 : nombre de produits différents dans le panier (toujours sur 2 caractères)

4a9acc : premier produit du panier, ici : STM32F7 (toujours sur 6 caractères)

. : séparateur entre le produit et sa quantité, défini dans le protocole de communication

02 : quantité du premier produit à prélever (toujours sur 2 caractères)

/ : séparateur entre le premier et le deuxième produit dans le panier, défini dans le protocole de communication

0e3120 : deuxième produit (Saturne V) suivi de sa quantité : 5

aecd97 : troisième produit (Robe de Soirée) attendu 1 seule fois

\0\n : fin de trame, défini dans le protocole de communication

Pour éviter de recevoir des caractères illisibles, il faut que les réponses du serveur se terminent par un « \n »

3.2 Le Bluetooth

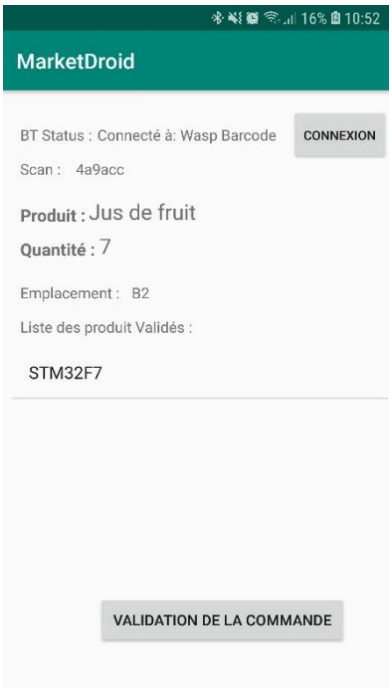

Figure 9. Interface Bluetooth

La deuxième fenêtre gère, quant à elle, la communication Bluetooth avec le scanner.

Lors de l'ouverture de la fenêtre une connexion automatique s'exécute pour s'appairer au scanner en Bluetooth. Des informations sur l'état de la connexion sont transmisse à l'utilisateur dans la partie haute de l'écran, ainsi qu'un bouton de secours pour relancer la connexion en cas de problème.

En passant par une table de correspondance L'ID d'un produit est traduit en un nom compréhensible pour l'utilisateur. Dans cette table chaque produit à un ID fixe et unique ainsi qu'un emplacement.

1	Nom du produit	id du produit	Emplacement	Rangée	Position
2	Aston Martin de James	f65eb4	D8	D	8
3	Avion de chasse	6f9a81	D4	D	4
4	Bague téléporteur	74bb6c	E3	E	3

Figure 10. Exemple de la table de correspondance

Le produit en attente d'être scanné et sa quantité sont donc traduit puis affichés juste en dessous de l'état de la connexion Bluetooth, grâce aux données issues du serveur et de la fenêtre précédente. Pour plus de réalisme nous avons choisis d'afficher également un emplacement

correspondant au lieu où trouver le produit dans l'entrepôt (voir *Rangée* et *Position* sur la figure précédente). Lorsqu'un produit est scanné il est comparé au produit attendu s'il est différent un message d'erreur (Pop-up rouge) apparait sinon il est ajouté à une liste dynamique qui se remplie à chaque nouveau produit attendu scanné. Dans la dernière partie de l'écran ce situ le bouton de validation de fin de commande, ce bouton affiche un nouveau message d'erreur si tous les produits du panier n'ont pas été scannés ou retourne un message de validation dans le cas où tous les produit sont correctement scannés.

Détection du scanner :

La première étape est de vérifier la compatibilité Bluetooth de l'appareil, vérifier si l'appareil dispose bien du Bluetooth. Une fois ce test, rapidement, fait il faut activer le Bluetooth. Pour accéder au Bluetooth sur un appareil Android il faut définir/utiliser des permissions dans le fichier *Manifest.xml*.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Figure 11. Permissions utilisées dans le projet (fichier Manifest.xml)

Les permissions de localisations sont indispensables car elles permettent à l'application d'exécuter une recherche de nouveau appareils disponibles dans les alentours.

Une fois le Bluetooth activé on peut commencer les recherches des appareils disponibles. Les appareils déjà appairés sont testé en premiers. Une méthode existe dans Android Studio pour récupérer cette liste : « *.getBondedDevices();* ». Si l'appareil n'est pas détecté ici il faut alors tester tous les appareils disponibles dans les alentours. Ici encore une méthode existe : « *.isDiscovering()* », pour récupérer les informations des nouveaux appareils il faut utiliser un « *BroadcastReceiver* ». Dans les deux types de recherche puis tester tous les appareils détectés avec leurs adresses MAC, sachant que nous recherchons toujours le même scanner (*"00:1C:97:14:35:02"*) et s'il est trouvé on débute une communication.

Pour recevoir les informations scannées nous utilisons un gestionnaire (*Handler*). Ce dernier englobe les fonctions de vérification du produit et changement de l'affichage (produits scannés, nouveaux produits en attentes).

Lorsque la liste du panier est entièrement scannée, cela signifie que l'utilisateur à récupérer l'intégralité de la commande du client, il peut alors valider la commande ce qui entraine la déconnexion du scanner Bluetooth.

Conclusion

Ce projet a été très formateur, car il nous a permis de mélanger plusieurs technologies au sein d'un même projet : Bluetooth, Wi-Fi, TCP/IP, développement Linux, développement Microcontrôleur, Développement Android, RTOS, Python, Excel …

Aujourd'hui, le serveur WIFI permet de connecter plusieurs clients en simultané et gère le protocole de demande de nouveau panier. En revanche, la validation de ce dernier n'est pas encore disponible. Les IHM sur la STM32F7 et le mobile Android sont fonctionnelles et permettent une gestion globale de la communication Bluetooth avec le scanner ainsi que la gestion des différents produits présents demandés dans les paniers.

Les points d'amélioration restant sont la mise en place des protocoles de communication restant, la mise en place de la base de données SQL ainsi que la réalisation du support « bracelet » pour la STM32.

Annexes

Lien du dépôt github : https://github.com/Brautantoine/Advanced-market

Révisions

Première édition : version 1.0 – Braut Antoine et Bastien Duprey – 21/06/2019