Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Qualités de Service sur Set-Top Box sur Android
QoS for STB on Android

Cahier des charges - Rapport de projet

Filière informatique

Classe I3

Auteur :

Alan Brunetti
alan.brunetti@edu.hefr.ch

Superviseurs :

Jean-Frédéric Wagen
jean-frederic.wagen@hefr.ch

Jean-Roland Schuler
jean-roland.schuler@hefr.ch

Expert :

Benoît Piller

Responsable externe :

Olivier Macchioni

Fribourg, 12 juillet 2013

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Table des matières

Chapitre 1

Introduction

Ce chapitre introduit le travail effectué durant 7 semaines pour mon travail de Bachelor qui s'est déroulé chez Wingo SA à Fribourg. Nous allons donc voir le contexte dans lequel ce projet a été réalisé.

1.1 Contexte

Ce projet est le fruit de 7 semaines de travail dans le cadre du projet de Bachelor de l'Ecole d'Ingénieurs et d'Architectes de Fribourg, étape finale du cursus de 3 ans de la filiale Informatique.

1.1.1 Enoncé original

Pour le développement logiciel pour une set-top box TV-multimedia sous Android.
Société filiale de Swisscom (Suisse) SA située à Fribourg, WinGo a été créée en 2010 dans le but de développer de nouveaux services de télécommunication sur les réseaux fixe et mobile en Suisse. Sa philosophie se veut innovante en matière de produits et de processus de façon à répondre à une dynamique de marché actuelle et future extrêmement compétitive. Aujourd'hui, WinGo offre déjà des produits grand-public qu'elle commercialise sous la marque M-Budget, notamment un accès internet DSL, la téléphonie fixe et une TV numérique, et propose des services en marque blanche ou en partenariat. Active également dans le domaine du mobile en tant qu'opérateur, la société WinGo développe et propose des services de type MVNO (Mobile Virtual Network Operator).

Commentaire

Comme on peut le remarquer, l'énoncé n'est pas très explicite. C'est après discussion que nous avons pu voir en quoi consistait réellement le projet.

1.1.2 Nouvel énoncé

WinGo offre un service de TV sur Ip (IPTV). Il s'agit d'une Set-Top Box (STB) tournant sous Android qui reçoit le flux de données, installée chez le client. Il est actuellement possible de mesurer la Qualité de Service (QoS) entre la plateforme de streaming IPTV et le

routeur ADSL du client. Il n'est par contre pas possible de mesurer cette qualité jusqu'à la Set-Top Box. Le but du projet est donc le développement d'une solution client (STB) - serveur (WinGo) permettant de mesurer la Qualité de Service depuis Thom, un outils de supervision de Wingo, jusqu'à la Set-Top Box.

1.2 Objectifs

1.2.1 Primaires

- Analyse de produits existant permettant de définir la qualité d'un service
- Analyse sur la possibilité d'intégrer ces produits sur la STB de WinGo
- Implémentation d'un serveur recevant les données de la STB
- Implémentation de la STB sous Android
- Intégration des résultats du serveur sur la plateforme de supervision de l'entreprise

1.2.2 Secondaires

- Implémentation d'une petite interface utilisateur affichant les résultats

1.3 Spécifications

La partie cliente est la Set-Top Box. Celle-ci tourne sur Android et fait partie du réseau local de l'utilisateur. Il n'est pas possible d'atteindre directement un appareil depuis l'extérieur à cause des différentes couches de NAT. Il faudra donc que l'application soit la plus autonome possible.

Le lancement se fera donc au démarrage de la STB. Elle ne contiendra aucune interface graphique et n'aura aucun launcher. Le tout se fera sous la forme d'un service. Cela veut dire que tout se fera en "background".

C'est l'application qui se chargera de se connecter au serveur. La connexion sera permanente, car nous n'aurons pas la possibilité de lancer le service à distance. Il faut donc qu'en cas de perte de connexion, une reconnexion soit faite automatiquement.

Ensuite c'est elle aussi qui évaluera sa qualité de service. Les informations seront récoltées, puis envoyées au serveur. Elle devra juger la ligne qui se trouve entre la STB et le serveur.

Pour la partie serveur, nous devons supporter une connexion permanente avec plusieurs centaines voir milliers d'appareils dans le futur. Le serveur devra récupérer les informations générées par la STB et les envoyer à un service de supervision, nommé THOM, qui lui affichera de manière lisible pour l'humain les résultats récoltés.

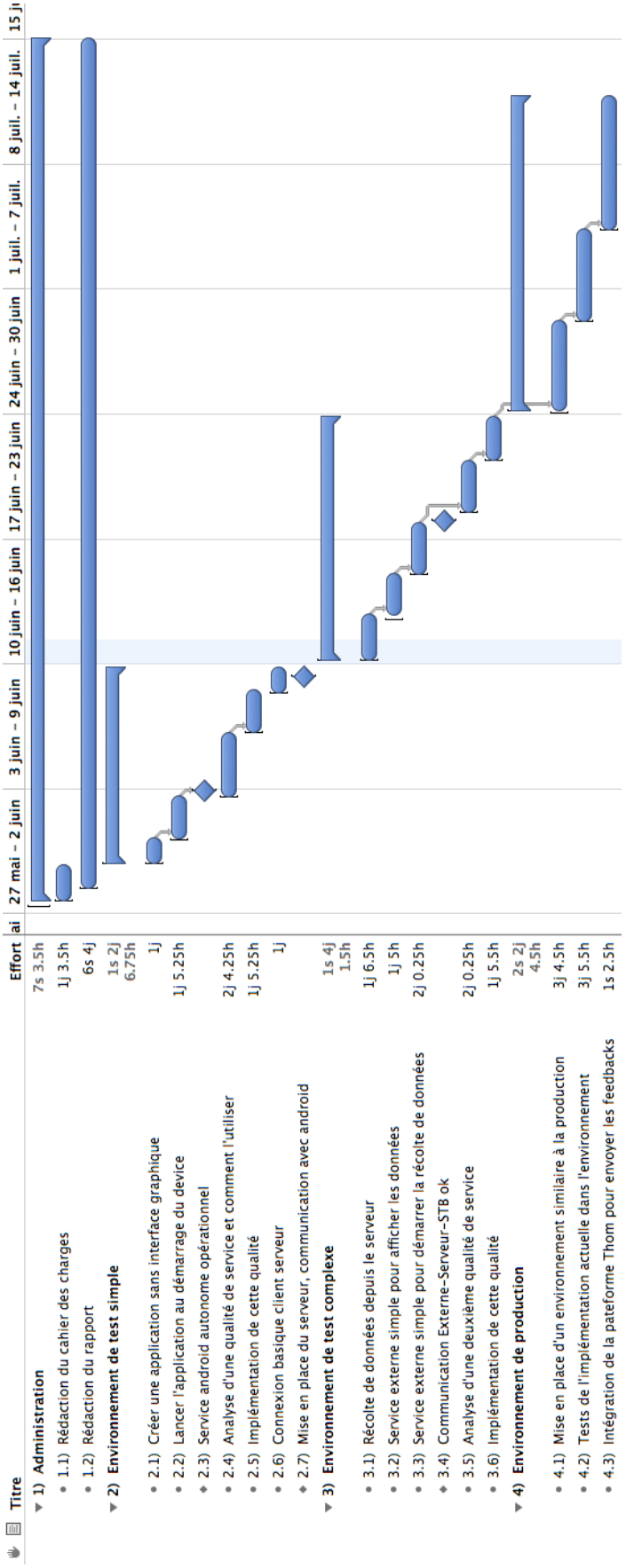
Pour résumer, voici les spécifications :

- Service Android sans interface graphique ni launcher
- Lancement automatique au démarrage de la Set-Top Box
- Connexion au serveur grâce au service Android
- Récolte d'informations sur la qualité de service depuis la Set-Top Box entre celle-ci et le serveur.

- Transmission des données au serveur
- Serveur supportant des connexions permanentes la plus fiable possible
- Scalabilité permettant la connexion de multiples appareils
- Récolte des informations et transmission de celles-ci à un service de supervision

1.4 Planification

Le déroulement aura un penchant Agile. A savoir qu'au lieu de faire les grosses phases "Analyse, conception, réalisation", nous opterons plutôt pour des cycles itératifs. C'est-à-dire que nous commencerons petit, en pure environnement de test, où nous mettrons en place une qualité de service. Lorsque ceci est prêt, nous passerons à une échelle plus proche de la production et ajouterons d'autres services et ainsi de suite. Il sera ainsi possible de rapidement se rendre compte de ce qui pourrait poser problème par la pratique et de corriger le tir plus facilement.


Titre
Effort
ai
27 mai – 2 juin
3 juin – 9 juin
10 juin – 16 juin
17 juin – 23 juin
24 juin – 30 juin
1 juil. – 7 juil.
8 juil. – 14 juil.
15 ju
1) Administration 7s 3.5h
1.1) Rédaction du cahier des charges 1j 3.5h
1.2) Rédaction du rapport 6s 4j
2) Environnement de test simple 1s 2j 6.75h
2.1) Créer une application sans interface graphique 1j
2.2) Lancer l'application au démarrage du device 1j 5.25h
2.3) Service android autonome opérationnel
2.4) Analyse d'une qualité de service et comment l'utiliser 2j 4.25h
2.5) Implémentation de cette qualité 1j 5.25h
2.6) Connexion basique client serveur 1j
2.7) Mise en place du serveur, communication avec android
3) Environnement de test complexe 1s 4j 1.5h
3.1) Récolte de données depuis le serveur 1j 6.5h
3.2) Service externe simple pour afficher les données 1j 5h
3.3) Service externe simple pour démarrer la récolte de données 2j 0.25h
3.4) Communication Externe-Serveur-STB ok
3.5) Analyse d'une deuxième qualité de service 2j 0.25h
3.6) Implémentation de cette qualité 1j 5.5h
4) Environnement de production 2s 2j 4.5h
4.1) Mise en place d'un environnement similaire à la production 3j 4.5h
4.2) Tests de l'implémentation actuelle dans l'environnement 3j 5.5h
4.3) Intégration de la pateforme Thom pour envoyer les feedbacks 1s 2.5h

Chapitre 2

Analyse

Nous allons voir ici une analyse globale de l'application, ce que l'on peut utiliser. Durant les prochains chapitres, certaines parts d'analyse sont aussi faite.

2.1 Serveur

Le serveur a besoin de deux parties, l'une permettant la communication avec la Set-Top Box, l'autre permettant d'appeler les données depuis Thom.

2.1.1 Communication avec la Set-Top Box

Comme expliqué, nous avons besoin d'un communication permanente. Après quelques recherches et après discussions, mon choix c'est porté vers les WebSockets.

Ce protocole, récent, vise à établir une communication bi-directionnelle, via TCP, entre un client et un serveur et ce de manière indépendante. Cela veut dire qu'il n'y a pas de question/réponse séquentielle, mais que le client comme le serveur peut envoyer des messages. Ce protocole a été mis en place en vue d'étendre HTTP et ainsi éviter la problématique du question/réponse. Ainsi, la première tentative de connexion vers un serveur WebSockets sera HTTP, qui ensuite sera mise à jour en WebSockets. Les WebSockets ont aussi été créés dans le but d'avoir énormément de connexions simultanées. On parle de milliers ou centaine de milliers. Il s'agit du futur de la communication asynchrone. Un autre point important est la durée de la connexion. Celle-ci est établie une seule fois, et peu importe le nombres de messages envoyés et quand, la communication reste ouverte et il n'y a pas besoin de se reconnecter. Au départ, les WebSockets ont été créés pour le Web, et sont donc supportés par les navigateurs, mais le protocole se développe à présent sur d'autres supports, le mobile notamment et Android. http ://stackoverflow.com/questions/14703627/websockets-protocol-vs-http http ://www.websocket.org/index.html Mon choix s'est porté sur Jetty, développé par Eclipse, pionnier dans les WebSockets. http ://www.eclipse.org/jetty/ Il existe bien sûr d'autres implémentations, de JBoss et Tomcat par exemple. Mais Jetty est très supporté et possède Eclipse derrière lui. De plus, sa documentation est claire et bien écrite, et les utilisateurs sont actifs concernant les problèmes rencontrés pour la recherche de solution. http ://www.eclipse.org/jetty/documentation/current/websockets.html

2.1.2 Communication avec Thom

Concernant l'appel des données depuis Thom, nous sommes face à un cas typique de communication via Web Service. Il a été d'un commun accord avec Olivier d'utiliser des Web Services REST afin de transiter les messages en JSon et ainsi simplifier la manipulation et transmission des données.

Mon choix s'est porté sur Jersey, car je l'ai déjà utilisé auparavant. Sa réputation précède cette implémentation qui n'a plus à se défendre.

https ://jersey.java.net/
https ://fr.wikipedia.org/wiki/Representational_State_Transfer

2.2 Android

Android est le système d'exploitation de la Set-Top Box. Ce qu'il faudra voir au fur et à mesure, étant donné que nous ne sommes pas face à un téléphone mobile, c'est dans quelle mesure nous pouvons développer sur cet appareil.

Mais j'ai eu la chance de rencontrer Monsieur Robert Wienecke, qui travaille chez Swisscom et a participé au développement de la box. Il m'a garanti que tout fonctionnerait et que nous avons vraiment affaire à du pure développement Android. Les modifications apportées de leur part surviennent au niveau du launcher. C'est-à-dire qu'au lieu de se retrouver avec une interface Android comme nous la connaissons, nous nous retrouvons directement avec le flux de télévision ainsi qu'une surcouche graphique faite par Swisscom. Je n'aurai donc aucun soucis au niveau du développement.

2.2.1 WebSockets sur Android

Un autre besoin est donc de pouvoir communiquer via WebSocket depuis Android.

De ce côté-là, il n'a que peu de choix, et l'implémentation principale nous vient d'Autobahn for Android.

http ://http ://autobahn.ws/android
Cette librairie est la plus utilisée du marché et d'ailleurs bon nombres d'autres librairies sont basées sur celle-ci. C'est un projet Open Source hébergé sur GitHub qui garanti l'implémentation des derniers standards du protocole WebSocket.
https ://github.com/tavendo/AutobahnAndroid/

Chapitre 3

Environnement de développement

3.1 Github

GitHub est un site web permettant l'hébergement et la gestion de logiciel. Il utilise le gestionnaire de version Git.
Mes projets sont donc disponibles sur GitHub à l'adresse suivante :

`https://github.com/brunettia/stb-qos`

On retrouve ici tout ce qui concerne mon projet :

- **01_cahiers_des_charges** : Nous retrouvons ici les différentes versions du cahier des charges, ainsi que le planning du projet.
- **02_pv** : Tous les procès verbaux des séances que j'ai eu durant le projet se retrouvent ici.
- **03_rapport** : Le rapport du projet ainsi que ses documents annexes sont disponibles dans ce dossier.
- **04_android_app** : L'application Android de la Set-Top Box avec son code source.
- **05_server_app** : La partie WebSockets et Web Services du serveur est hébergée ici.

Pourquoi avoir utilisé GitHub ? Les projets hébergés sur GitHub, à moins qu'ils soient privés, sont disponibles pour tout le monde. Il est possible de lire les sources directement en ligne ainsi que de "forker" le projet. Cela veut dire que n'importe qui peut le modifier.

Après discussion avec Olivier, il n'y avait pas de problème à ce que le projet soit Open Source, du moment que les informations relatives à l'authentification ne soient pas disponibles au grand public.

L'avantage à présent c'est que le projet est disponible et visible, et peut donc aider les utilisateurs traitant du même sujet que le mien. J'ai moi-même beaucoup utilisé GitHub pour voir certaines conceptions et des exemples de développement.

3.2 Maven

Maven est un outils d'Apache permettant la gestion et la compilation de projets Java. Je l'ai utilisé pour construire mon serveur Web.

Maven permet de créer un projet basique et d'ajouter ses dépendances. Celles-ci peuvent être des librairies externes, disponibles sur des dépôts en ligne, ou alors un autre projet.
Il se configure grâce à un fichier XML POM (Project Object Model) qui définit le nom du projet, sa version, sous quelle forme il sera compilé et contenu (war) ainsi que les dépendances.

Pour l'utiliser, il suffit de quelques commandes.
- **mvn clean install** permet de faire un clean & build des sources, en téléchargeant les librairies externes définies dans notre fichier pom.xml, puis créera notre fichier WAR à déployer dans le dossier "target" à la racine du projet. Il suffit de copier ce fichier sur notre serveur Web pour que notre application soit fonctionnelle.
- **mvn eclipse :eclipse** permet, à partir du code source, de créer un projet importable dans Eclipse. L'avantage est d'être indépendant des autres utilisateurs. Il n'y aura aucun soucis de chemins pour les fichiers de configuration ou de librairies, car il crée le projet par rapport à son utilisateur.

3.3 Déploiement

Nous allons voir ici comment générer et déployer l'exécutable .apk sur la Set-Top Box ainsi que le fichier .war pour le serveur.

3.3.1 Set-Top Box

La STB est connectée au réseau local et possède une adresse IP. L'utilitaire **adb** permet de se connecter sur la STB.

Listing 3.1– Connection à la Set-Top Box via ADB

```
1 adb connect IP_ADDR
```

A présent, la box est connectée comme n'importe quel terminal Android. Il est donc possible, via Eclipse, d'afficher les logs du terminal, ainsi que de compiler et déployer le programme.

Sauf que le déploiement ne **fonctionne pas**. En effet, sans doutes à cause d'options de compilation rajoutées par Eclipse, l'application n'est pas déployée correctement et n'est pas reconnue par la box. Ce qu'il faut donc faire, c'est construire le fichier .apk avec Eclipse, puis l'envoyer manuellement sur la box via l'utilitaire adb.

L'application est générée dans le chemin suivant :
APP_DIRECTORY/bin/app_name.apk

Il suffit alors d'utiliser adb.

Listing 3.2– Envoi du fichier apk sur la Set-Top Box

```
1 adb push APP_DIRECTORY/bin/app_name.apk /system/data
```

Le répertoire **/system/data** n'est accessible qu'en tant qu'utilisateur root.

L'application est à présent déployée sur notre STB.

3.3.2 Serveur

Le dossier **webapp** du répertoire d'installation de Jetty est le dossier de déploiement du serveur. Lorsqu'un fichier est déposé dans ce répertoire, il est automatiquement lu et déployé par Jetty s'il est démarré, ou le fera lorsqu'il sera lancé.

La compilation de la partie serveur se fait avec Maven comme déjà expliqué. En se mettant dans le répertoire racine de l'application, caractérisé par le dossier **pom.xml** que Maven va lire, il suffit de lancer la commande

Listing 3.3– Commande Maven pour construire le fichier déployable

```
1 mvn clean install
```

On remarque qu'à la première exécution de cette commande, un dossier **target** est créé.

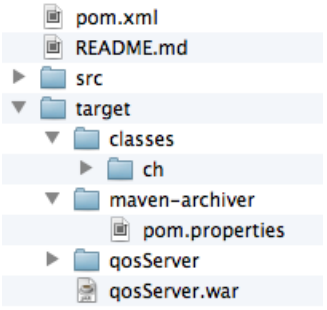


FIGURE 3.1 – Target contenu

- classes : répertoire contenant les classes compilées de l'application
- maven-archiver : contient le fichier pom.properties. Ce sont les propriétés tirées du fichier pom.xml
- qosServer : dossier contenant l'extraction du fichier war. Il s'agit du projet déployé
- qosServer.war : fichier conteneur permettant le déploiement de l'application sur le serveur

Il suffit de copier le fichier .war dans le dossier webapp de Jetty pour pouvoir le déployer.

Listing 3.4– Copie du fichier war vers Jetty

```
1    cp SERVER_DIRECTORY/target/app_name.war JETTY_DIRECTORY/webapp
```

L'application est déployée à l'adresse :
protocole ://server_name :portNumber/war_file_name

Chapitre 4

Phase de tests simple

La phase de tests dite simple va me permettre de prendre en main les éléments de base. La situation est la suivante : J'utilise une tablette Nexus 7 de Asus tournant sur Android pour simuler la Set-Top Box. Celle-ci est connectée en Wifi sur le réseau local de Wingo. L'attribution des informations du réseau est fourni via DHCP. La passerelle par défaut fera donc office de routeur du client.

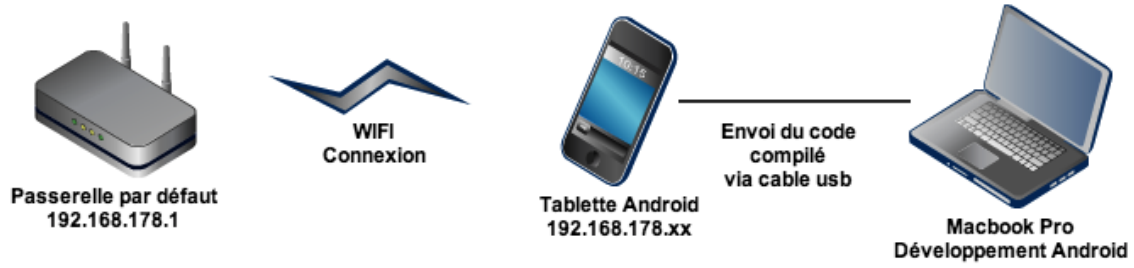


FIGURE 4.1 – Environnement de développement simple

4.1 Démarrage d'un service

Qu'est-ce qu'un service sous Android ? C'est un composant permettant à l'application de travailler sans interaction avec l'utilisateur. Il est donc exécuté en arrière-plan, à un moment donné. Ce moment est un événement. Dans notre cas il s'agira de démarrer le service lorsque la box aura démarrée.

Dans un premier temps, nous devons accorder au niveau du Manifest la permission de recevoir l'événement de fin de démarrage.

Listing 4.1– Permission de fin de démarrage

```
1 <uses-permission android:name="android.permission.
      RECEIVE_BOOT_COMPLETED" />
```

Ensuite, nous devons définir un **Broadcast Receiver**, qui permet de recevoir des **Intents**, des intentions. L'intention sera "Au démarrage". Lorsque l'intention est arrivée, on lance notre service correspondant. Tout cela se passe dans le fichier AndroidManifest.xml, qui est la structure de notre application.

Listing 4.2– Broadcast Receiver et Service dans AndroidManifest.xml

```
<receiver android:name="ch.wingo.stb.receiver.AutoStart" >
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
</receiver>
<service android:name="ch.wingo.stb.service.AutoStartService" android:enabled="true
    " />
```

La balise "receiver" permet de déclarer un BroadcastReceiver. Le nom de sa classe correspondante est "AutoStart". A l'intérieur nous avons un Intent sur l'action "BOOT_COMPLETED". Ainsi, lorsque cette action sera filtrée, la méthode "onReceive" de la classe "AutoStart" sera exécutée.
Nous voyons aussi le service "AutoStartService". Il est juste déclaré, et ne réagit pas spécialement. C'est au rôle du receiver de lancer le service.

Listing 4.3– Classe AutoStart.java

```
public class AutoStart extends BroadcastReceiver{
  private final String TAG="wingo.stb.qos.AutoStartBroadcastReceiver";
  @Override
  public void onReceive(Context context, Intent intent) {
    Intent intentToService = new Intent(context, AutoStartService.class);
    context.startService(intentToService);
    Log.i(TAG, "Service launched from AutoStart");
  }
}
```

Comme dit précédemment, lorsque l'Intent est filtrée, la méthode "onReceive" du receiver correspondant est exécutée. C'est ici que notre Service va être lancé, via la méthode "startService". A présent, AutoStartService est lancé et c'est à partir de là que notre application commence réellement son travail !

4.2 Ping

Dans un premier temps, j'ai cherché une qualité simple à effectuer, le temps de réponse. Celui-ci peut se mesurer grâce à la commande "Ping", qui pourra aussi être utilisé pour déterminer le nombre de paquets perdus.

Ce qui va être fait, c'est qu'au démarrage de la tablette, notre service sera lancé, puis il exécutera la commande ping sur la passerelle par défaut. Le résultat sera affiché via un Toast d'Android. Différents éléments sont nécessaires au fonctionnement du ping sur Android.

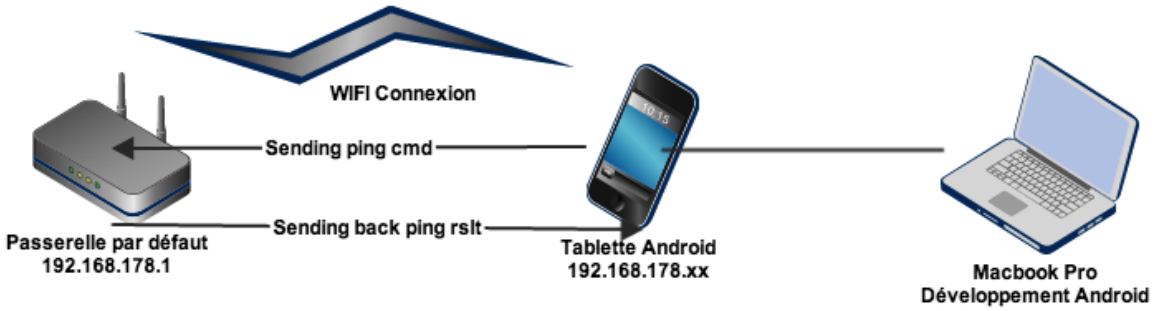


FIGURE 4.2 – Schéma de production actuel

4.2.1 Runtime and Process

La classe Runtime sur Android permet à Java d'interagir avec l'environnement sur lequel il tourne. Android est basé sur un noyau Linux, il est donc possible d'interagir avec celui-ci. La méthode "getRuntime()" retourne l'instance unique Runtime, puis sa méthode "exec" permet d'exécuter une commande UNIX, dans un process séparé. Celui-ci est récupéré par un objet de classe "Process" et nous pouvons récupérer son flux, à savoir le résultat de la commande exécutée.
Nous sommes donc capables d'exécuter la vraie commande Ping exécutée par le Linux d'Android.

4.2.2 Wifi Manager

La classe **WifiManager** va nous permettre d'accéder à différentes informations concernant la connexion Wifi établie. Cela va de la vitesse de la connexion, à son adresse ip, ou encore toutes les informations du DHCP, comme la passerelle ou les DNS. Il est donc tout à fait possible de récupérer l'adresse IP de la passerelle pour lui lancer la commande Ping.

4.2.3 Implémentation

Récupérer l'adresse IP de la passerelle par défaut

Listing 4.4– Code de récupération de la paserelle par défaut

```
1 //If connected to Wifi, return the gateway ip addr
2   public static String getWifiDefaultGateway(){
3     // getting the WifiManager from context
4     WifiManager wifi = (WifiManager) STBContext.getAppContext().
          getSystemService(Context.WIFI_SERVICE);
```

```
5       // returning the gateway converted to String, readable like an ip address
6       return intToIp(wifi.getDhcpInfo().gateway);
7   }
8
9   // Convert int format to string ip
10  // method from http://stackoverflow.com/questions/5387036/how-to-get-gateway
        -and-subnet-mask-details-in-android-programmatically
11  public static String intToIp(int addr) {
12      return ((addr & 0xFF) + "." + ((addr >>>= 8) & 0xFF) + "."
13              + ((addr >>>= 8) & 0xFF) + "." + ((addr >>>= 8) & 0xFF));
14  }
```

La méthode "wifi.getDhcpInfo().gateway" retourne un entier pour l'adresse IP. Il s'agit en fait de chaque entier de l'adresse IP, chaque fois décalé de 8 bits et converti en binaire, puis additionné. Le schéma suivant illustre la situation.

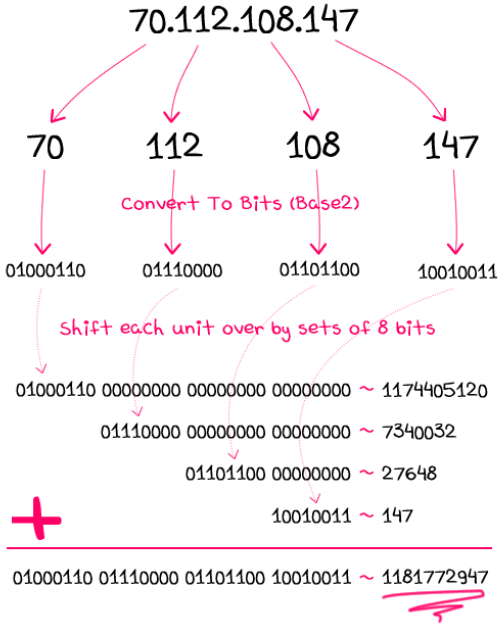


FIGURE 4.3 – Explication sur l'adresse IP format numérique

La méthode "intToIp" permet de convertir un entier en String sous la forme d'une adresse IP standard.

Ping

Listing 4.5– Code du ping

```
1 @Override
2   public void onStart(Intent intent, int startid) {
3       Log.d(TAG, "Service onStart");
```

```
4       ping();
5     }
6
7   // execute ping command
8   // code from http://learn-it-stuff.blogspot.ch/2012/01/ping-code-for-android-
        activity.html
9   public void ping(){
10    try {
11      String pingCmd = "ping -c 5 " + NetworkUtils.getWifiDefaultGateway();
12      String pingResult = "";
13      // getting the runtime
14      Runtime r = Runtime.getRuntime();
15      // executing our command ping
16      Process p = r.exec(pingCmd);
17      //getting the stream of the process
18      BufferedReader in = new BufferedReader(new
19      InputStreamReader(p.getInputStream()));
20      String inputLine = "";
21      String result = "";
22      // building the result
23      while ((inputLine = in.readLine()) != null) {
24        pingResult += inputLine;
25      }
26      in.close();
27
28      // showing the result
29      Toast.makeText(this, pingResult, Toast.LENGTH_LONG).show();
30    }
31    catch (IOException e) {
32      Toast.makeText(this,e.getLocalMessage(), Toast.LENGTH_LONG).show();
33    }
34  }
```

Comme expliqué, nous utilisons Runtime pour exécuter un processus. Nous récupérons ensuite le flux de sortie de celui-ci afin de construire le résultat, qui sera ensuite affiché sur un Toast.

4.3 Iperf

> “ Iperf est un outil pour mesurer la bande passante et la qualité d'un lien réseau. Ce dernier est délimité par deux machines sur lesquelles est installé Iperf.
> *Définition tirée de OpenManiak.com* ”

Iperf va donc être utilisé pour mesurer la bande passante. Il sera lancé en tant que serveur sur notre serveur, et en tant que client sur notre terminal Android.

Nous allons utiliser **Iperf for Android**, qui est une version recompilée et adaptée à l'architecture d'Android. Nous aurons ainsi le vrai outils Iperf, qui sera presque utilisé de la même manière que le ping.

Iperf for Android propose son code source, une interface graphique ainsi que le code permettant de lancer la commande et lire le résultat.

4.3.1 Implémentation

Voici la méthode permettant l'exécution d'Iperf.

Listing 4.6– Code d'exécution d'Iperf

```
1
2  public String iperf(){
3  Process process = null;
4      try {
5        // getting the server from the ressource file
6        String cmd = "-c " + getString(R.string.server_addr_ip)
7              + " -i 10";
8        // split the command
9        String[] commands = cmd.split(" ");
10       List<String> commandList = new ArrayList<String>(
11             Arrays.asList(commands));
12
13       // The execution command is added first in the list for the shell
14       // interface.
15       commandList.add(0, getString(R.string.iperf_data_folder));
16
17       // Creating and running the process with our iperf
18       process = new ProcessBuilder().command(commandList).
             redirectErrorStream(true).start();
19
20       // A buffered output of the stdout is being initialized so the iperf
21       // output could be displayed on the screen.
22       BufferedReader reader = new BufferedReader(new InputStreamReader(
23             process.getInputStream()));
24
25       String str1 = "";
26       String iperfResult = "";
27
28       while ((str1 = reader.readLine()) != null) {
29         iperfResult += str1+"\n";
30       }
31       reader.close();
32       process.destroy();
```

```
33          Log.d(TAG, "Iperf result: \n" + iperfResult);
34      } catch (IOException e) {
35          Log.e(TAG, e.getMessage());
36          return "";
37      }
38      return iperfResult;
39  }
```

Ce que nous remarquons c'est que cette fois nous n'utilisons pas le Runtime car la commande ne vient pas du noyau Linux mais c'est un exécutable qui se trouve dans les sources du projet. Ainsi, au lieu de récupérer le processus via le Runtime, nous allons créer celui-ci via l'objet **ProcessBuilder**. Sa méthode "command" prend en paramètre une liste. Celle-ci est en fait la séparation de toutes les options de la commande.

Position	Valeur
0	/data/data/ch.wingo.stb/iperf
1	-c
2	http ://addr_server
3	-i
4	10

TABLE 4.1 – Contenu de la liste de commandes pour iperf

Ainsi, la commande effectuée est "/data/data/ch.wingo.stb/iperf -c http ://addr_server -i 10".

A noter que le répertoire **ch.wingo.stb** correspond au nom du package et que c'est le seul endroit possible pour une application de stocker des informations qui lui sont propres. Ainsi, l'exécutable iperf pour Android est copié dans ce répertoire lors de sa première utilisation. Lors de mes premières tentatives d'exécution le répertoire était faux, ce qui empêchait la copie d'iperf.

```
root@android:/data/data # ls -l
drwxr-x--x app_0    app_0            2013-06-24 13:18 ch.wingo.stb
drwxr-x--x app_12   app_12           2009-02-14 00:32 com.android.browser
drwxr-x--x app_11   app_11           2013-06-07 10:26 com.android.defcontainer
```

FIGURE 4.4 – Liste des droits dans le répertoire data d'Android

On voit que seul l'utilisateur app_0, créé pour notre application, a accès à ce répertoire. Et il est impossible d'en utiliser d'autres.

L'option -i permet quant à elle de définir sur combien de secondes la mesure va être faite.

4.4 Hello World Serveur

La dernière partie de la phase de tests simple était de mettre en place le serveur et de communiquer de manière simple avec notre terminal Android.

4.4.1 Installation de Jetty

Jetty est téléchargeable sur son site officiel.
http ://download.eclipse.org/jetty/stable-9/dist/

Nous nous retrouvons avec une archive à dézipper qui contient le serveur.

La structure est la suivante :

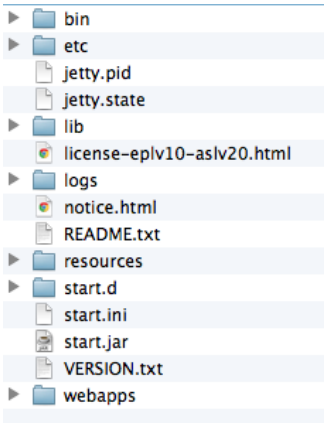


FIGURE 4.5 – Jetty structure

Répertoire	**Description**
bin	Contient les exécutables de Jetty. On y trouve notamment **Jetty.sh** qui permet, via les options start, stop et restart de démarrer, stopper et redémarrer le serveur
etc	Contient tous les fichiers de configuration de Jetty. Par défaut peu utile, mais pour la mise en place d'SSL c'est ici que nous irons.
lib	Contient toutes les librairies de Jetty
logs	Contient tous les fichiers de logs. Chaque jour un nouveau log est créé.
resources	Contient quelques fichiers de configuration en rapport aux logs notamment.
start.d	Dossier de démarrage de Jetty. Il contient par défaut le lancement de l'application démo de Jetty, à supprimer lors de la mise en production car vulnérable !
webapps	Dossier de déploiement de Jetty. Les applications sont à déposer ici avec d'être automatiquement déployées par Jetty.

TABLE 4.2 – Arborescence du répertoire de Jetty

Pour lancer Jetty, il suffit à présent de se placer depuis le terminal dans son répertoire et de lancer la commande

Listing 4.7– Commande de lancement de Jetty

```
1 bin/jetty.sh start
```

Si tout s'est bien passé, rendez-vous sur la page **http ://localhost :8080** afin de voir la page d'accueil de Jetty.

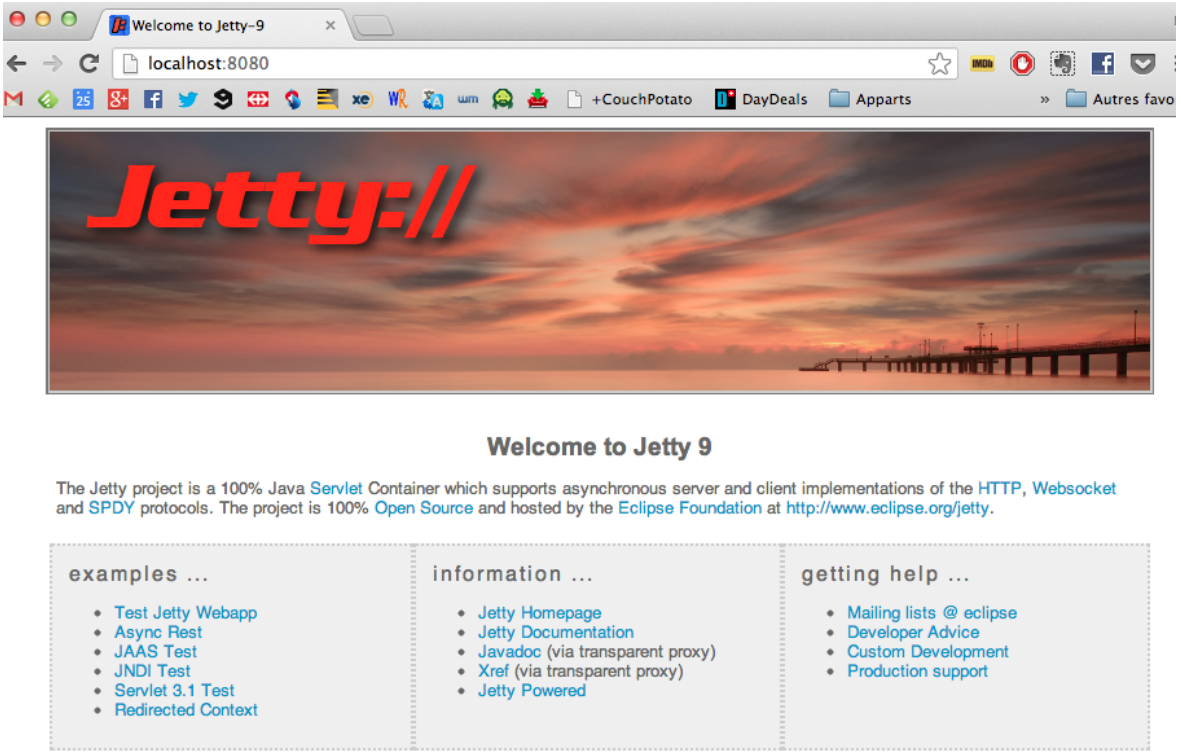


FIGURE 4.6 – Page d'accueil du serveur Jetty

Notre serveur tourne !

4.4.2 Implémentation du serveur

Nous allons voir ici ce qui est nécessaire au fonctionnement basique des WebSockets sur notre serveur.
Concrètement, nous avons besoins de deux choses :

1. Une servlet permettant la réception des requêtes de connexion et la création d'un socket dédiée
2. Un socket permettant de dialoguer avec notre terminal Android.

C'est tout ce qu'il faut. Au niveau de l'implémentation, plusieurs possibilités s'offrent à nous. Utiliser les fichiers de configuration, utiliser les annotations ou instancier chaque classe nous-mêmes.
J'ai personnellement choisi l'annotation, pour la clarté offerte. Lorsque l'on voit la classe, on sait directement à quoi elle va servir, pas besoin de vérifier dans un fichier de configuration externe, et pas besoin de créer des possibilités d'erreurs en créant le tout nous-mêmes.

Nous allons ici faire le tutoriel proposé par Eclipse permettant de faire un **echo**. Chaque message réceptionné par le serveur sera renvoyé en retour au client.
http ://www.eclipse.org/jetty/documentation/current/jetty-websocket-api-annotations.html

Servlet

Notre servlet nous sert à réceptionner les requêtes HTTP désirant établir une connexion WebSocket sur notre serveur. On devra donc lui définir une **URL**.

La réception faite, il s'agit ici de configurer quelque peu le serveur. Nous allons ici juste lui donner un **idleTimeout**, un timeout qui coupera la connexion si aucun trafic n'a été détecté durant un certain temps. Ce temps sera de **10 minutes**. Cette valeur sera stockée dans le fichier web.xml afin d'être facilement modifiable.

Ensuite, nous créerons un socket qui s'occupera de communiquer avec notre terminal.

Listing 4.8– Servlet de réception de requêtes HTTP

```
1  @SuppressWarnings("serial")
2  @WebServlet(name="PRIS WebSockets", urlPatterns="/register")
3  public class RemoteSTBServlet extends WebSocketServlet {
4
5      @Override
6      public void configure(WebSocketServletFactory factory) {
7        // getting the idleTimeout parameter from web.xml
8        int idleTimeout = Integer.parseInt(getServletContext().
             getInitParameter("idleTimeout"));
9        System.out.println(new Date()+" Servlet: configured ! Timeout set to "+
             idleTimeout+" ms");
10       // setting the timeout
11       factory.getPolicy().setIdleTimeout(idleTimeout);
12       // creating a new Socket
13         factory.register(RemoteSTBSocket.class);
14     }
15 }
```

Pour résumer les quelques paramètres

Répertoire	Description
@WebServlet	Annotation définissant la classe comme Servlet
name	Nom de la Servlet
urlPatterns	URL d'accès à la Servlet. L'URL est relative à l'adresse de l'application.
extends WebSocketServlet	Étend à la classe WebSocketServlet qui nous donne accès à la méthode "configure". Celle-ci sera ensuite exécutée automatiquement.
factory.register	Va créer un nouveau Socket. En donnant la classe de celui-ci, l'instantiation se fait automatiquement.

TABLE 4.3 – Résumé des paramètres Servlet

Socket

Les Sockets permettent de communiquer entre deux points, client et serveur, via la session qu'ils gèrent. C'est ici que l'on pourra envoyer des messages sur nos terminaux, et c'est ici que l'on réceptionnera les leurs.

Il existe 4 événements dans un Socket :

1. **On Connect** : Lorsqu'un client est connecté pour la première fois
2. **On Close** : Lorsque la connexion est fermée
3. **On Message** : Lorsque le serveur reçoit un message du client
4. **On Error** : Lorsqu'une erreur survient.

Ces événements sont interceptés par les annotations. Nous aurons par exemple **@OnWebSocketConnect** pour définir la méthode qui réceptionnera l'événement "On Connect".

Chaque méthode prend en paramètre un objet de type Session, qui est l'objet de connexion avec le client distant. Via cet objet nous pourrons donc envoyer des messages mais aussi fermer la connexion, savoir si la session est ouverte, si elle sécurisée, quelle est l'adresse IP du client etc.

Voici le code permettant de faire un "echo" d'un message reçu.

Listing 4.9– Code du Socket pour faire un echo

```
1  @WebSocket(maxMessageSize = 64 * 1024)
2  public class EchoSocket {
3
4      @OnWebSocketMessage
5      public void onText(Session session, String message) {
6          if (session.isOpen()) {
7              System.out.printf("Echoing back message [%s]%n", message);
8              session.getRemote().sendStringByFuture(message);
9          }
10     }
11 }
```

Donc ce qu'on peut lire c'est :
"Lorsque je reçois un message, je vérifie que la session soit bien ouverte. Si c'est le cas, je renvoie le message dans le futur au client".

session.getRemote() me permet de récupérer un objet "RemoteEndpoint", qui correspond à mon interlocuteur.
sendStringByFuture va envoyer un message, mais de manière **non bloquante**. C'est-à-dire qu'une pile de messages existe et que notre message y est ajouté, en étant envoyé sans qu'on ne sache à quel moment. En opposition il existe **sendString** qui lui est **bloquant**. Le message est directement envoyé et le code continue uniquement lorsque celui-ci a été réceptionné par le client.

4.4.3 Résultat du serveur

Après quelques recherches, j'ai trouvé un utilitaire se nommant **Dark WebSocket Terminal** permettant de se connecter à un serveur en utilisant les WebSockets. Il va ainsi me permettre de tester si mon serveur est bien configuré.
https ://chrome.google.com/webstore/detail/dark-websocket-terminal/

4.4.4 Implémentation du client

Du côté du client, comme expliqué en "Analyse", nous allons utiliser **Autobahn Android**. Son utilisation se veut simple : un objet **WebSocketConnection** mConnect, contenant la méthode **connect**, et prenant en paramètre l'URL du serveur ainsi qu'une instance de la classe **WebSocketHandler** qui elle réceptionnera les différents événements, comme vu avec les Sockets.

Les événements sont les suivants :

1. onOpen : Ouverture de la connexion
2. onTextMessage : Lors de la réception d'un message texte
3. onClose : Lors de la fermeture de la connexion

L'envoi de message sur le serveur se fait via la méthode **sendTextMessage** de l'objet mConnect.

Ce que nous allons simplement faire, c'est envoyer le message "Hello World" au démarrage du terminal, qui devra être renvoyé en retour par le serveur.

Listing 4.10– Envoi Hello World Android vers Serveur via WebSockets

```
1
2    private final WebSocketConnection mConnection =
3    new WebSocketConnection();
4
5    @Override
6       public void onStart(Intent intent, int startid) {
7             connectToServer();
```

```
8         }
9
10    public void connectToServer(){
11        try {
12        // connecting to the IP of my server in the local network
13        String wsuri = "ws://192.168.0.10:8080/qosServer/register";
14        mConnection.connect(wsuri, new WebSocketConnectionHandler() {
15            @Override
16            public void onOpen() {
17                Log.d(TAG, "Connected to serveur");
18                //sending hello world
19                mConnection.sendTextMessage("Hello World !");
20            }
21
22            @Override
23            public void onTextMessage(String payload) {
24                Log.d(TAG, "We have got a message:\n");
25                Log.d(TAG, payload);
26            }
27
28            @Override
29            public void onClose(int code, String reason) {
30                Log.d(TAG, "Closing the connection. Reason: "+reason);
31            }
32        });
33    } catch (WebSocketException e) {
34
35        Log.e(TAG, e.toString());
36    }
37 }
```

L'url a été construite comme expliquée :
l'adresse IP du serveur, qui sera remplacée dans le futur par un nom de domaine +
le nom du fichier déployée sans l'extension (qosServer.war) +
l'url de la servlet (/register).

Chapitre 5

Phase de tests avancée

La phase de tests dite avancé va introduire la Set-Top Box ainsi que l'implémentation du client et du serveur pour un fonctionnement final. Nous aurons donc l'introduction des Web Services sur le serveur.
La situation est la suivante :

J'utilise un switch, où sont connectés mon ordinateur, qui fait office de serveur, la Set-Top Box ainsi qu'un routeur. Les adresses IP sont ainsi distribuées via DHCP par le routeur.

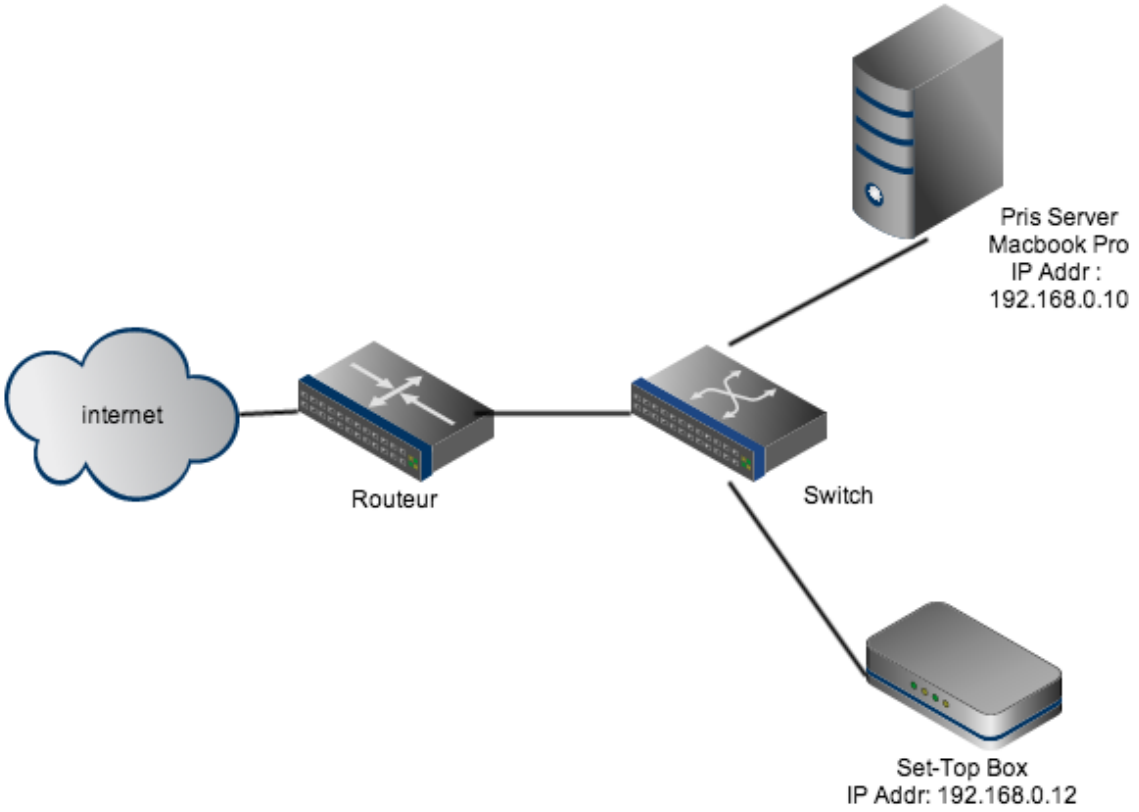


Figure 5.1 – Environnement de développement avancé

5.1 Conception

5.1.1 Identification de la Set-Top Box

Chaque Set-Top Box est identifiable par sa MAC adresse. Lorsqu'une Box se connectera sur notre serveur, la première valeur qui sera envoyée vers celui-ci est sa MAC adresse. De plus, cette information est disponible sur Thom.

Ainsi, il sera possible de répertorier et identifier chaque session par la MAC adresse, en imaginant une table ayant pour clé l'adresse et la session comme objet. Lorsque l'adresse a été ajoutée avec la session, la communication est établie et le dialogue peut commencer.

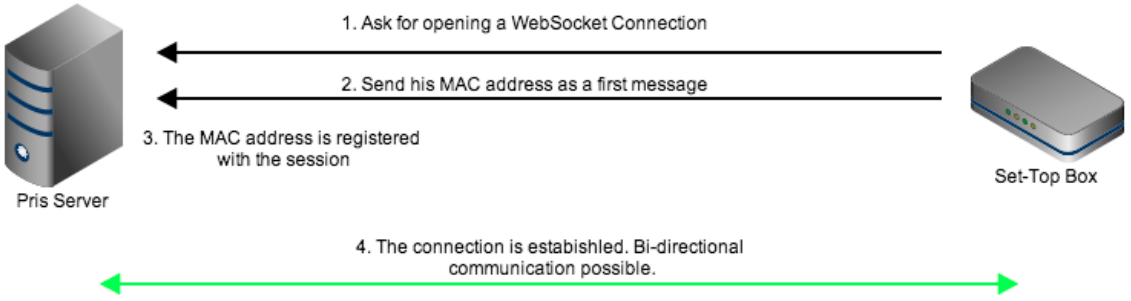


FIGURE 5.2 – Démarrage de la Set-Top Box

Nous utiliserons une **HashTable** dont la clé sera de type **String**, pour l'adresse MAC et la valeur sera de type **Session** pour stocker la session distante.

5.1.2 Centralisation des WebSockets

Afin de faciliter la communication entre les Web Services et les WebSockets, une classe passerelle sera créée. Car bien qu'il s'agisse du même serveur, il faut pouvoir transférer les requêtes reçues vers la partie WebSockets.

Le but est donc d'avoir une classe singleton, qui contiendra notre HashTable. Elle contiendra deux méthodes, **join** et **leave**, qui permettent respectivement d'ajouter une session dans la HashTable et de la retirer lorsque la connexion est fermée. De plus, chaque service aura sa méthode dans cette classe. Lorsque le Web Service recevra une requête, il la transmettre à notre WebSocketCentralisation, qui elle la transmettra au bon Socket, qui pourra communiquer avec la STB.

C'est elle aussi qui retourna la valeur de retour au Web Service, qui pourra à son tour retourner la valeur au demandeur.

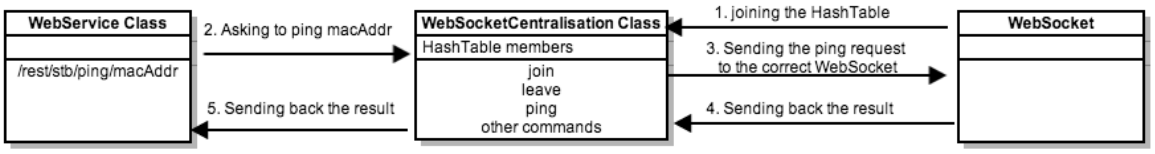


FIGURE 5.3 – Fonctionnement de la WebSocketCentralisation

5.1.3 Envoi de commandes sur la Set-Top Box

En complément de ce qui a été vu précédemment, nous allons définir le déroulement d'un envoi d'une commande en partant de Thom, jusqu'à notre Set-Top Box et le retour des données.

1. Pour commencer, Thom fait une requête à l'adresse du Web Service désiré.
2. Réception faite, Pris envoie la requête à la STB au format JSon.
3. La STB réceptionne la requête, exécute la commande, puis retourne le résultat, au format JSon, sur Pris.
4. Pris réceptionne le résultat et le retourne, toujours au format JSon, vers Thom.

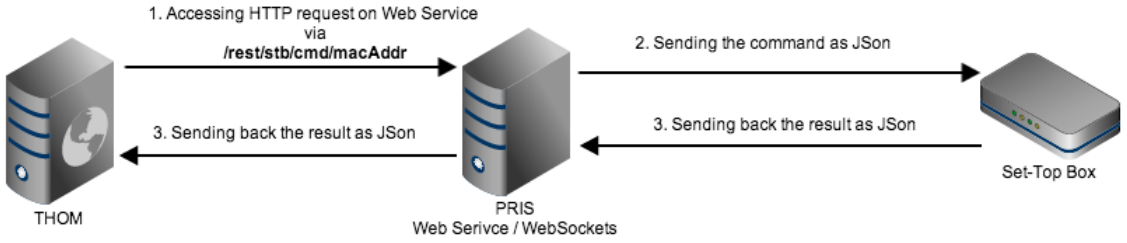


FIGURE 5.4 – Envoi de commande depuis Thom

Comme le montre la figure ci-dessus, les données transitées sont au format JSon. Il sera ainsi possible via un seul String de définir de quel type de commande il s'agit, et de transmettre des informations supplémentaires, par exemple pour un ping, choisir la destination. Voici un exemple de JSon représentant le résultat d'un ping.

Listing 5.1– Résultat au format JSon

```
{
   "type" : "ping",
   "data_size_unit" : "bytes",
   "time_unit" : "ms",
   "ip_dest" : "173.194.35.24",
   "macAddr" : "00:09:DF:1B:FC:A5",
   "data_size" : 64,
   "packet_loss" : 0.0,
   "time" : 35.2,
   "round_trip_avg" : 35.236,
   "round_trip_max" : 35.236,
   "round_trip_min" : 35.236,
   "round_trip_stddev" : 0.0,
   "packet_transmitted" : 1,
   "icmp_seq" : 1,
   "ttl" : 52,
```

```
17    "packet_received" : 1
18 }
```

Voici un JSon représentant une requête de ping envoyée sur la Set-Top Box

Listing 5.2– Résultat au format JSon

```
1  {
2     "opt":"google.ch",
3     "cmd":"ping"
4  }
```

"cmd" étant la commande à effectuer et "opt" les options possibles de ping. Ici il s'agit juste de la destination.

5.1.4 Synchronisation des données

Un problème persiste, comment synchroniser le retour des données de la Set-Top Box jusqu'au Web Service, sachant que je n'ai aucun moyen de savoir, lorsque j'envoie la requête via WebSockets sur la box, quand le résultat va m'être retourné.

Il existe un sous-protocole des WebSockets, nommé **The WebSocket Application Messaging Protocol** (WAMP). Ce que permet de faire WAMP est d'appliquer le principe de **Publish and Subscribe**, qui permet de s'inscrire sur un serveur pour recevoir des notifications, ce qui ne nous intéresse pas, et le principe de **Remote Procedure Call**(RPC). Ici, cela devient plus intéressant, car il s'agit d'appeler des méthodes et d'avoir un **callback**, c'est-à-dire que lorsque l'appel de méthode à distance est fini, le résultat est envoyé et nous pouvons **attendre** sur celui-ci. Le problème est que l'appel de méthode se fait sur ... le serveur ! Or, ce que nous aurions aimé, c'est faire appel au client.

J'ai toute fois utilisé le même principe. Lorsque le Socket envoie la commande à la STB, celui-ci va attendre sur le résultat. Tant que le résultat attendu est vide, j'attends. Dès que le résultat est arrivé, je le renvoie au demandeur.
Nous pouvons faire un **sleep** du Thread courant, qui permettra d'attendre que le résultat soit mis à jour. Si ce n'est toujours pas le cas, je le rendors.

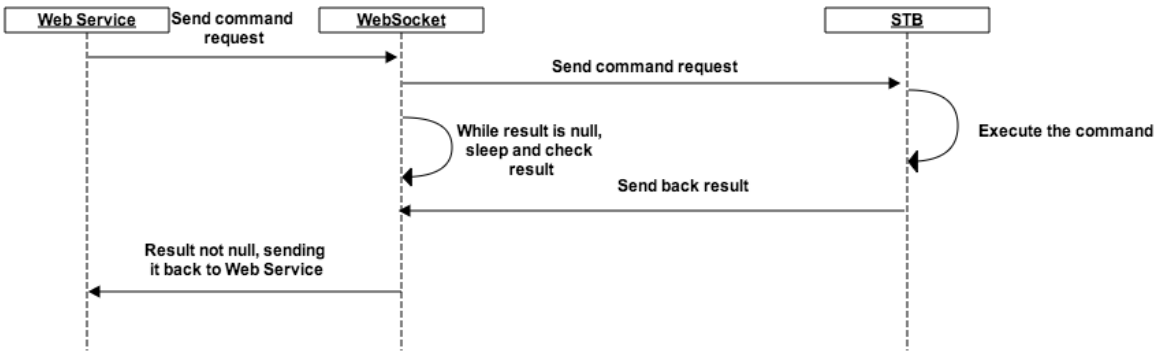


FIGURE 5.5 – Synchronisation du résultat avec le serveur

5.1.5 Gestion de la connexion

La gestion de la connexion consiste à garantir que peu importe les problèmes rencontrés, la connexion doit restée établie.

Serveur

Le serveur lui doit maintenir la connexion établie en envoyant des messages. C'est son seul travail. Nous aurons donc un timer qui toutes les 5 secondes enverra un message au client.

J'ai remarqué que lorsque la communication est interrompue, en enlevant par exemple le câble ethernet de la box, le serveur fait comme si tout se passait bien durant 60 secondes. Si la connexion est rétablie dans ce laps de temps, tout rentre dans l'ordre et les messages qui devaient être envoyés ont été bufferisés ce qui permet de les transmettre.
C'est au-delà des 60 secondes que le serveur coupe officiellement la connexion.

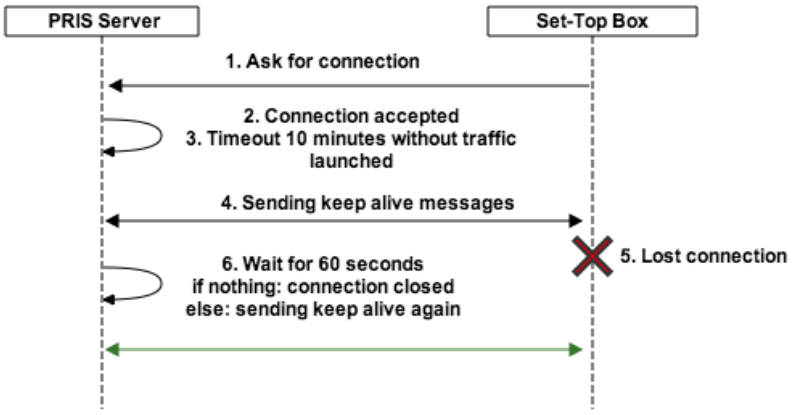


FIGURE 5.6 – Gestion de la connexion sur le serveur

Client

Le client quant à lui, aura un timer de 65 secondes pour dire : "si je n'ai pas reçu de message en 65 secondes, je coupe et relance la connexion".

Il aura un autre timer qui se connectera de manière aléatoire, entre 9 et 11 secondes, sur le serveur lorsque la connexion est perdue.

5.1.6 SSL

SSL/TLS permet de sécuriser une connexion en encryptant les données, ainsi que de certifier que le site distant est une source sûre. Dans notre cas, seule la partie de cryptage des données nous intéresse et non pas la partie certification.

Pourtant il n'est pas facile d'avoir l'un sans l'autre et j'ai dû faire quelques modifications pour que cela fonctionne.

Premièrement, il s'avère que Autobahn Android ne gère pas de base l'accès sécurisé **wss** du WebSocket. Cependant, une version non officielle existe. Il s'agit d'un projet à part modifier pour SSL. Pourquoi est-ce que ça n'est pas encore intégré ? Car en ajoutant la gestion de SSL, la librairie aurait perdu en rapidité de traitement des données. Donc ce n'est pas très grave dans notre cas puisque nous n'avons pas une charge excessive.
https ://github.com/tavendo/AutobahnAndroid/tree/tlsnio

Après installation sur Android, je m'occupe de Jetty. La documentation semble claire sur l'installation :
http ://www.eclipse.org/jetty/documentation/current/configuring-ssl.html
Mais impossible de le faire fonctionner, une exception est levée :
oeji.SelectorManager :qtp1384613607-13-selector-0 :

Et impossible de savoir de quoi cela vient. Après différents tests, c'est-à-dire refaire la configuration des clés et des certificats, essayer d'autres clients pour la connexion, toujours la même erreur.

C'est alors que j'ai remarqué que je n'étais pas à jour avec Jetty, j'utilisais la version 9.0.3 au lieu de la 9.0.4. Mais j'utilisais les librairies de la 4 ! Après mise à jour, le serveur fonctionne !

Par contre c'est à présent le client qui ne veut pas : le certification n'est pas officiellement certifié ! Après quelques recherches je ne trouve pas comment palier à ce problème. Il est écrit qu'il est possible de se connecter à un serveur non certifié, mais uniquement si l'on est depuis l'émulateur Android . J'ai donc cherché où le test se faisait et ai décidé d'enlever cette condition, puis de recompilé la librairie.

Listing 5.3– classe WebSocketConnection modifié

```
protected SSLContext getSSLContext() throws KeyManagementException,
    NoSuchAlgorithmException {
      //if (!mOptions.getVerifyCertificateAuthority()) {
         // Create a trust manager that does not validate certificate chains
         TrustManager tm = new X509TrustManager() {
            public void checkClientTrusted(X509Certificate[] chain,
                  String authType) throws CertificateException {
            }

            public void checkServerTrusted(X509Certificate[] chain,
                  String authType) throws CertificateException {
            }

            public X509Certificate[] getAcceptedIssuers() {
               return null;
            }
         };
         SSLContext ctxt = SSLContext.getInstance("TLS");
         ctxt.init(null, new TrustManager[] { tm }, null);

```

```
20              Log.d(TAG, "trusting all certificates");
21              return ctxt;
22
23          // } else {
24          // Log.d(TAG, "NOT trusting all certificates");
25          // return SSLContext.getDefault();
26          // }
27      }
```

La partie en commentaire est l'implémentation de base.

Ce n'est guère très propre, mais ce changement n'affecte en rien le fonctionnement de la librairie et c'était un besoin.

5.2 Jersey Web Services

Nous allons voir dans cette section comment a été installé Jersey, ainsi que son implémentation.

5.2.1 Installation

L'installation se fait via Maven. Il suffit d'ajouter les dépendances dans notre fichier **pom.xml** et de lancer la commande **mvn clean**.

Les dépendances se trouvent sur le site officiel de Jersey.

J'ai eu de la peine à comprendre quelles librairies utiliser au départ, car les versions ont beaucoup évoluées en peu de temps. Ainsi les exemples de configuration que l'on trouve sur Internet ne sont pas forcément à jour. Dans tous les cas, la configuration proposée est fonctionnelle en utilisation la configuration de Glassfish depuis le site de documentation cité précédemment.

Listing 5.4– Dépendances Maven pour Jersey

```
<!-- Web Services -->
  <dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>2.0</version>
  </dependency>
  <dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
```

```
14      <groupId>org.codehaus.jackson</groupId>
15      <artifactId>jackson-jaxrs</artifactId>
16      <version>1.9.12</version>
17    </dependency>
18    </dependencies>
```

5.2.2 Hello World

Le but de cette partie est de mettre en place un premier Web Service, qui une fois appelé nous retournerait le fameux message "Hello World" !

Pour ce faire, il faudra définir une URL d'accès, qui sera **http ://SERVER_ADDR :8080/qos-Server/rest/test/hello**. Lorsque cette adresse est atteinte, le texte "Hello World" est retourné.
Nous allons créer une classe **QOSResource** qui regroupera tous nos points d'entrée, dans le package **ch.wingo.pris.WS.resource**

Nous devons déclarer ce package dans notre fichier **web.xml** afin de définir sous quelle URL celui-ci sera utilisée.

Listing 5.5– Configuration web.xml pour Web Service

```
1  <servlet>
2      <servlet-name>jersey-serlvet</servlet-name>
3      <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
4      <init-param>
5        <param-name>jersey.config.server.provider.packages</param-name>
6        <param-value>ch.wingo.pris.WS.resources</param-value>
7      </init-param>
8    </servlet>
9
10   <servlet-mapping>
11     <servlet-name>jersey-serlvet</servlet-name>
12     <url-pattern>/rest/*</url-pattern>
13   </servlet-mapping>
```

Ainsi, chaque requête sur l'URL "/rest" va être redirigé dans le package "ch.wingo.pris.WS.resources".

Voyons à présent le code Java.

Listing 5.6– Implémentation Hello World Web Service

```
1  @Path("/test")
2  @Consumes(MediaType.TEXT_PLAIN)
3  public class QOSResource {
4
5    public QOSResource(){}
```

```
6
7   @Produces(MediaType.TEXT_PLAIN)
8   @Path("/hello")
9   @GET
10  public String helloWorld(){
11     return "Hello World";
12  }
```

Paramètre	Description
@Path	Permet de définir le chemin de l'URL, pour arriver au niveau de la bonne classe, puis de la bonne méthode.
@Consumes	Définit ce qui peut être pris en paramètre des méthodes. Se met au niveau de la classe, pour une valeur par défaut, ou au niveau de la méthode, ce qui écrasera le type par défaut.
@Produces	Définit ce qui va être produit en retour.
@GET	Il s'agit du GET HTTP, cela veut dire que nous allons chercher une valeur. Peut donc aussi être POST, PUT ou DELETE. Une même URL peut être différencier par sa méthode HTTP !

Table 5.1 – Résumé des paramètres Web Services

@Consumes et @Produces prennent comme valeur les différents MIME. Nous avons ici du simple texte, mais cela peut-être du JSon, du XML, du HTML etc.

Nous avons mis comme chemin, au niveau de la classe, "/test". Donc toutes les requêtes qui contiennent "/rest/test" vont arriver ici. Ensuite la méthode helloWorld sera atteinte par l'URL "/rest/test/hello".

5.2.3 Implémentation

Chaque service aura donc une URL qui lui est propre. Le pattern suivant a été choisi :

http ://SERVER_URL/qosServer/rest/stb/ :cmd/ :macAddr

Paramètre	Description
/qosServer	URL de notre web application
/rest	URL à partir de laquelle tous nos Web Services seront accessibles
/stb	URL à partir de laquelle nous aurons accès aux Set-Top Box
/ :cmd	Ce paramètre est une variable. Sa valeur dépendra de ce que l'on veut faire. En l'occurrence, nous aurons **ping**, **iperf**, **reboot**
/ :macAddr	Ce paramètre est une variable. Sa valeur sera la MAC adresse de la Set-Top Box.

Table 5.2 – Résumé des paramètres Web Services

Notre classe **QOSResource** sera donc implémentée. On y donnera en paramètre du JSon, pour le type de commande et les options, et le retour produit sera lui-aussi en JSon.

Listing 5.7– Implémentation du Web Service

```
@Path("/stb")
//@Path("/test")
@Consumes(MediaType.APPLICATION_JSON)
public class QOSResource {

  public QOSResource(){}

  @Produces(MediaType.APPLICATION_JSON)
  @Path("/ping/{macAddr}")
  @GET
  public Response getPing(@PathParam("macAddr") String macAddr,
      @QueryParam("cmd") String cmd){
    System.out.println("WebService : Ping");
    String json = WebSocketsCentralisation.getInstance().ping(macAddr, cmd
        );
    return Response.ok(json).header("Access-Control-Allow-Origin", "*").build
        ();
  }

  @Produces(MediaType.APPLICATION_JSON)
  @Path("/iperf/{macAddr}")
  @GET
  public Response getIperf(@PathParam("macAddr") String macAddr,
      @QueryParam("cmd") String cmd){
    System.out.println("WebService : Iperf");
    String json = WebSocketsCentralisation.getInstance().iperf(macAddr,
        cmd);
    return Response.ok(json).header("Access-Control-Allow-Origin", "*").build
        ();
  }

  @Path("/reboot/{macAddr}")
  @GET
  public Response reboot(@PathParam("macAddr") String macAddr){
    System.out.println("WebService : Reboot");
    WebSocketsCentralisation.getInstance().reboot(macAddr);
    return Response.ok("reboot done").header("Access-Control-Allow-Origin", "*"
        ).build();
  }

}
```

Ce que l'on peut remarquer, c'est le retour, il s'agit d'un objet de type **Response**. Il est ainsi possible de faire un retour personnalisé en donnant un code de réponse HTML. Par

exemple, si la MAC adresse est introuvable, on pourrait répondre un code 404 - not found. Le cas est ensuite traité sur la partie web, Thom.

Dans mon cas, il n'est pas pleinement exploité car je retourne systématiquement une réponse positive, seul le texte va changer. C'est donc un point d'amélioration possible.

Ensuite, si je l'utilise, car nous pourrions répondre un simple String s'il est au format JSon, c'est pour le fameux problème du **Allow-origin**. En effet, il n'est **pas possible** de télécharger des données depuis un site sur un serveur. Le navigateur bloque ce genre de requête par défaut, à l'exception d'images par exemples. Il s'agit d'un système de sécurité pour éviter des intrusions.

La solution est de réceptionner la requête, puis de modifier l'en-tête HTTP en changeant la valeur du paramètre "Access-Control-Allow-Origin". Pour l'instant, la valeur est à "*" ce qui autorise **toutes les origines**. Il serait plus judicieux de réduire au domaine hébergeant Thom afin d'améliorer la sécurité de l'application.

Ceci est un problème que je n'avais pas prévu et qui m'a fait perdre un peu de temps.

Autre remarque, la méthode **reboot** est à la base **void**. Lorsque cette requête est appelée, nous n'avons aucun retour sur le déroulement du redémarrage de la STB. Ceci est aussi une amélioration possible.

Nous voyons aussi l'utilisation de notre singleton **WebSocketCentralisation**, qui regroupe les méthodes d'accès, et qui retourne le résultat de mesure.

5.3 WebSocketCentralisation

Nous allons voir comment a été implémentée cette classe centrale, élément de transition entre les Web Services et les WebSockets. Seule la méthode iperf des commandes est montrée, les autres étant similaires.

5.3.1 Implémentation

,

```
1  // singleton of websocketcentralisation
2  private static final WebSocketsCentralisation INSTANCE = new
       WebSocketsCentralisation();
3  // HashMap regrouping all the sessions, identifiable by the MAC Addr
4  private HashMap<String, RemoteSTBSocket> members = new HashMap<String,
       RemoteSTBSocket>();
5  // Constant value if no user is found in members
6  private static final String USER_NOT_FOUND = "{\"error\":\"user not found\"}";
7
8  public static WebSocketsCentralisation getInstance(){
9    return INSTANCE;
10 }
```

```
11
12  // Join: Add the session to the HashTable
13  public void join(RemoteSTBSocket socket){
14    System.out.println(new Date()+" New socket connexion added: "+socket.
          getMacAddr());
15    // adding the socket to the hashtable
16    members.put(socket.getMacAddr(), socket);
17    // printing the connected boxes
18    for(RemoteSTBSocket sockets:members.values()){
19      System.out.println(new Date()+" Is connected: "+sockets.getMacAddr());
20    }
21  }
22
23  // Leave: Remove the session from the HashTable
24  public void leave(RemoteSTBSocket socket){
25    System.out.println(new Date()+" Socket connexion removed: "+socket.
          getMacAddr());
26    // removing the socket from the hashtable
27    members.remove(socket.getMacAddr());
28    // printing the connected boxes
29    for(RemoteSTBSocket sockets:members.values()){
30      System.out.println(new Date()+" Is connected: "+sockets.getMacAddr());
31    }
32  }
33
34  // Will ask the correct Session to iperf
35  public String iperf(String macAddr, String cmd) {
36    // if the session exists
37    if(members.containsKey(macAddr))
38      // returning the iperf value from the session
39      return members.get(macAddr).iperf(cmd);
40    // else returning error message
41    else return USER_NOT_FOUND;
42  }
```

On retrouve donc les éléments expliqués, une HashTable regroupant toutes nos connexions, les méthodes "join" et "leave" permettant de rejoindre et de quitter la HashTable, ainsi qu'une méthode de commande, iperf.

Celle-ci va donc récupérer la bonne session grâce à la MAC adresse, exécuter la commande iperf de la session et attend le résultat. Nous verrons à la prochaine étape comment se fait "l'attente".

5.4 RemoteSTBSocket

Voici la partie traitant de la communication par les WebSockets. Il s'agit de voir comment les requêtes sont envoyées, réceptionnées et comment la connexion est gérée.

5.4.1 Traitement des requêtes

Dans la classe WebSocketCentralisation, nous avons vu que nous utilisions la méthode "iperf" des websockets. Son but est d'envoyer un message via la session à la STB, et d'attendre le résultat pour le retourner.

Listing 5.8– Méthode iperf du WebSocket

```
1  // sending iperf request to the stb
2    // waiting for the result
3    // called by WebSocketCentralisation
4    public String iperf(String cmd) {
5      String result = ""; // result sends back
6      // send command to the stb via websocket
7      session.getRemote().sendStringByFuture(cmd);
8      try{
9        System.out.println(new Date()+ " Gonna wait");
10       // while the iperf result is empty
11       while (iperfResult.equals("")) {
12         // I sleep for 10 ms
13         Thread.sleep(10);
14       }
15     } catch(InterruptedException e){
16       e.printStackTrace();
17     }
18     // if i m out, i have a result! Copying the iperResult
19     result = iperfResult;
20     // reinitialising iperfResult for next time
21     iperfResult = "";
22     System.out.println(new Date()+" No more sleeping, gonna send: "+result);
23     // returning the result to WebSocketCentralisation
24     return result;
25   }
```

Nous voyons que la méthode **attend** le résultat du String **iperfResult**. Cet objet est mis à jour lors de la réception du résultat envoyé par la STB. Nous allons justement voir comment cela se passe.

Listing 5.9– Réception d'un retour de résultat STB to PRIS

```
1  @OnWebSocketMessage
```

```
2     public void onText(Session session, String message) {
3         if (session.isOpen()) {
4 ObjectMapper mapper = new ObjectMapper(); // mapper to convert a String JSon to
      a map
5       Map<String, Object> map; // map containing the result json in format key/
            value
6       try {
7         // converting the String json to a map
8         map = mapper.readValue(message, new TypeReference<Map<String,
              Object>>() {});
9         // getting the type of command
10        String type = (String) map.get("type");
11         if(type.equals("iperf")){
12           System.out.println(new Date()+" Socket: iperf result received");
13           // updating iperfResult with message, the result.
14           // the sleeping thread will now awake!
15           iperfResult = message;
16         }
17       }
```

Lorsque le résultat est reçu, nous testons s'il s'agit d'iperf. Si c'est le cas, nous allons mettre à jour "iperfResult". Il s'agit d'objet dont le Thread attend le changement de valeur pour le renvoyer. Nous sommes ainsi notifiés de manière très rapide.

5.4.2 Gestion de la connexion

Pour la partie serveur, la gestion de la connexion se limite à envoyer des **Keep Alive**. Ceci se fait via un Timer, qui toutes les **SEND_KEEP_ALIVE_MS**, va envoyer un message en disant simplement **hello**.

Listing 5.10– Code de lancement des Keep Alive

```
1 @OnWebSocketConnect
2   public void onConnect(Session session){
3     System.out.println(new Date()+" Device connected: "+session.
          getRemoteAddress().getHostString());
4     keepAliveTimer.schedule(new KeepAliveTask(), 0, SEND_KEEP_ALIVE_MS);
5     this.session = session;
6   }
7
8   // Task for timer to send hello messages to maintain
9   // the connexion
10  private class KeepAliveTask extends TimerTask{
11
12    @Override
13    public void run() {
```

```
14          session.getRemote().sendStringByFuture("hello");
15      }
16  }
```

L'utilisation de la méthode **sendStringByFuture**, qui rappelons-le, est **non bloquante**, est dans ce cas pratique puisque déjà, un nouveau thread est créé pour juste envoyer les messages, et en plus il peut le faire de manière continu. Ce n'est pas dit que toutes les 5 secondes la STB reçoive le message, mais au moins ceux-ci sont empilés et seront de toute façon envoyés.

Comme expliqué, c'est après un décompte de 60 secondes, que le serveur va stopper définitivement la connexion. Nous rentrons dans la méthode "onClose" qui va me permettre de bien vérifier que toute connexion soit arrêtée, puis de quitter la HashTable de la classe "WebSocketCentralisation".

Lorsque c'est le serveur qui reçoit à son tour un message **hello** de la part de la STB, il n'y a rien que l'on puisse faire puis que le décompte sans communication se fait au niveau du client.

Listing 5.11– Réception d'un keep alive sur le serveur

```
1 if(message.equals("hello")){
2   return;
3 }
```

5.4.3 Premier message reçu

Le premier message que le serveur reçoit, ce doit être la MAC adresse de la Set-Top Box. Nous allons donc vérifier qu'il s'agisse bien de cela puis si tel est le cas, nous enregistrer auprès de la classe "WebSocketCentralisation".

Listing 5.12– Code d'enregistrement de la Set-Top Box

```
1  @OnWebSocketMessage
2      public void onText(Session session, String message) {
3          if (session.isOpen()) {
4            if(firstConnection && isMacAddr(message)){
5              System.out.println(new Date()+" First message: should be mac addr: "
                   +message);
6              firstConnection = false;
7              this.macAddr = message;
8              WebSocketsCentralisation.getInstance().join(this);
9              return;
10           }
```

5.5 AutoStartService sur Android

Voici la partie Android. Certaines choses ont déjà été vues, les implémentations du ping et de iperf, ainsi que la connexion au serveur.
Nous allons voir comment se passe l'initialisation de la connexion, la réception de message, comment convertir un objet java en Json et surtout la gestion de la connexion du côté client, qui a la tâche de vérifier la réception des keep alive et de se reconnecter en cas de problème.

5.5.1 Première message

Le premier message envoyé par la box doit être sa MAC adresse. Il faut pour cela la récupérer et nous aurons besoin de savoir quelle interface est active sur la Set-Top Box. Pour le moment, seule la connexion via ethernet est possible, mais le code a été fait en sorte de supporter la connexion Wifi aussi.
Les interfaces sont listables via la commande **adb shell netcfg**

Ces méthodes font partie de la classe statique **NetworkUtils** qui me permet à tout moment de récupérer ses informations d'informations.

Listing 5.13– Network Utils

```
1  public static String getActiveInterface(Context context){
2      ConnectivityManager cm = (ConnectivityManager) context.
           getSystemService(Context.CONNECTIVITY_SERVICE);
3      switch (cm.getActiveNetworkInfo().getType()) {
4      case ConnectivityManager.TYPE_ETHERNET:
5        return "eth0";
6      case ConnectivityManager.TYPE_WIFI:
7        return "ra0";
8
9      default:
10       break;
11     }
12     return null;
13   }
14
15     // return the MAC address of the device
16   public static String getMACAddr(String interfaceName) {
17     try {
18       // getting all the interfaces
19       List<NetworkInterface> interfaces = Collections
20           .list(NetworkInterface.getNetworkInterfaces());
21
22       for (NetworkInterface intf : interfaces) {
23         if (interfaceName != null) {
24           // checking that the interface given exists
25           if (!intf.getName().equalsIgnoreCase(interfaceName))
```

```
                continue;
            }
            // getting the mac address on byte array format
            byte[] mac = intf.getHardwareAddress();
            if (mac == null)
              return "";
            StringBuilder buf = new StringBuilder();
            // converting byte array to readable String
            for (int idx = 0; idx < mac.length; idx++)
              buf.append(String.format("%02X:", mac[idx]));
            if (buf.length() > 0)
              buf.deleteCharAt(buf.length() - 1);
            return buf.toString();
          }
        } catch (Exception ex) {
          Log.e(TAG, ex.toString());
        }
        return "";
    }
```

La méthode getMACAddr a été trouvée sur StackOverFlow
http ://stackoverflow.com/questions/14190602/how-to-get-the-mac-address-of-an-android-devicewifi-is-switched-off-through-co

Maintenant que la mac adresse est récupérée, nous pouvons l'envoyer au serveur.

Listing 5.14– Envoi de la mac adresse au serveur

```
// when the connexion is established
        @Override
        public void onOpen(){
          Log.d(TAG, "Connected");
          Log.d(TAG, "First connexion, sending MAC @");
          Log.d(TAG, "My MAC Addr: "+ macAddr);
          mConnection.sendTextMessage(macAddr);
        }
```

5.5.2 Traitement des requêtes

Cela se passe à la méthode **onTextMessage**, à la réception d'un message. Lorsqu'il y a des options, comme avec ping et iperf, on réceptionne du JSon, et celui-ci commence toujours par le caractère **{**.
Autrement nous recevrons des String. Pour les commandes qui ne demandent pas d'option particulière (reboot) et pour les message de **Keep Alive** (hello).

Listing 5.15– Réception d'une commande avec options

```
@Override
            public void onTextMessage(String payload) {
                Log.d(TAG, "Message received: "+payload);
                // if we have JSon
                if(payload.startsWith("{")){
                    ObjectMapper mapper = new ObjectMapper(); // mapper String Json to map
                    Map<String, Object> map; // map key/value of the json recept
                    try {
                        // converting the string to map
                    map = mapper.readValue(payload, new TypeReference<Map<String, Object>>() {});
                    // if we have a an iperf
                    if(map.get("cmd").equals("iperf")){
                        Log.d(TAG, "Iperf request received");
                        // we get des options of the command
                        String opt = (String) map.get("opt");
                        // mapper Java Object to JSon
                        ObjectMapper toJson = new ObjectMapper();
                        // the result sent back
                        String json = "";
                        try {
                            // IperResult instantiate feed!
                            IperfResult ir = iperf(opt);
                            // if everything was good:
                            if(ir != null){
                                json = toJson.writerWithDefaultPrettyPrinter().writeValueAsString(ir);
                            }
                            // else the iperf server was certainly down. Creating a new JSon.
                            else {
                                JSONObject job = new JSONObject();
                                job.put("message", "Error during iperf. Check if iperf server is up!");
                                job.put("type", "error");
                                job.put("command", "iperf");
                                // make it pretty, not in a single line
                                json = job.toString(2);
                            }

                        } catch (JsonGenerationException e) {
                            e.printStackTrace();
                        } catch (JsonMappingException e) {
                            e.printStackTrace();
```

```
41                  } catch (IOException e) {
42                      e.printStackTrace();
43                  } catch (JSONException e) {
44                      e.printStackTrace();
45                  }
46                  // sending back the result to the server
47                  mConnection.sendTextMessage(json);
48              }
49          }
```

La **ligne 25** est intéressante. Car cette fois, au lieu d'utiliser un mapper pour convertir un String en map, nous convertissons un **objet java** en String sous format JSon. Regardons de plus près la classe **IperfResult**.

Listing 5.16– classe IperfResult

```
1  public class IperfResult {
2      private String type;
3      private String macAddr;
4      private double throughput;
5      private String unit;
6
7      public IperfResult(){}
8
9      public IperfResult(String type, String macAddr, double throughput, String
           unit){
10         this.type = type;
11         this.macAddr = macAddr;
12         this.throughput = throughput;
13         this.unit = unit;
14     }
15
16     public String getType() {
17         return type;
18     }
19     public void setType(String type) {
20         this.type = type;
21     }
22     public String getMacAddr() {
23         return macAddr;
24     }
25     public void setMacAddr(String macAddr) {
26         this.macAddr = macAddr;
27     }
28     public double getThroughput() {
29         return throughput;
```

```
30    }
31    public void setThroughput(double throughput) {
32      this.throughput = throughput;
33    }
34    public String getUnit() {
35      return unit;
36    }
37    public void setUnit(String unit) {
38      this.unit = unit;
39    }
40  }
```

Ce que l'on peut remarquer, c'est que chaque attribut possède un **get** et un **set** portant son nom. Ceci est une règle pour pouvoir convertir un objet directement en JSon. Ainsi que de posséder un constructeur vide.

Nous allons voir comment parser le résultat d'iperf, qui est un flux de String. Nous allons utiliser pour cela les expressions régulières.

Listing 5.17– Parsing du résultat d'une commande Iperf

```
1  String str1 = ""; // will contain each line
2          String[] arrayOfString = null; // will contain the results
3          while ((str1 = reader.readLine()) != null) {
4            Log.d(TAG, "Entering iperf result process");
5            // regex for a line with our result. We double de \ in java
6            if (str1.matches("\\[[ \\d]+\\]\\s*[\\d]+.*")) {
7              Log.d(TAG, "We got a match, filtering...");
8              // regex for the between of each value
9              Pattern localPattern = Pattern.compile("[-\\[\\]\\s]+");
10             // we split the line with our pattern. We get the results
11             arrayOfString = localPattern.split(str1);
12             break;
13           }
14           // regex to catch if the server it down
15           if (str1.matches("[a-zC:\\s]+")) {
16             Log.d(TAG, "Connection refused - no server found");
17             reader.close();
18             process.destroy();
19             return null;
20           }
21         }
```

Nous avons une première expression, qui permet de récupérer la ligne contenant les données de résultat.

FIGURE 5.7 – Expression régulière pour la ligne de résultat

```
------------------------------------------------------------
Client connecting to localhost, TCP port 5001
TCP window size:  144 KByte (default)
------------------------------------------------------------
[  5] local 127.0.0.1 port 50909 connected with 127.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  5]  0.0-10.0 sec  27.2 GBytes  23.4 Gbits/sec
```

FIGURE 5.8 – Vu de la ligne de résultat

En ce qui concerne le split entre les valeurs :

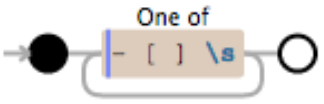


FIGURE 5.9 – Expression régulière pour la ligne de résultat

```
[  5]  0.0-10.0 sec  27.2 GBytes  23.4 Gbits/sec
```

FIGURE 5.10 – Vu de la ligne de résultat

Donc tous les entre-valeurs que l'on désire récupérer sont interceptés. Lorsque l'on split la ligne, nous aurons un tableau de valeurs. Il suffit d'instancier un objet IperfResult, de lui donner les valeurs, et de le retourner pour qu'il soit converti en JSon.

Reboot

Voici l'implémentation du reboot sous Android. Il nécessite la permission **<uses-permission android :name="android.permission.REBOOT" />** à ajouter dans le fichier **AndroidManifest.xml**, qui nécessite d'être **root** pour l'exécuter.

Listing 5.18– Code du reboot

```
1 private void reboot() { ((PowerManager)STBContext.getAppContext().
      getSystemService(Context.POWER_SERVICE)).reboot(null);
2   }
```

Nous récupérons une instance de **PowerManager** qui permet, comme son nom l'indique de gérer l'énergie du terminal sous Android. Toutefois attention :

> “ Device battery life will be significantly affected by the use of this API ”
> *Tiré de la documentation officielle d'Android*

Mais notre STB est sous alimentation nous n'avons donc pas de soucis à nous faire.
http ://developer.android.com/reference/android/os/PowerManager.html

5.5.3 Gestion de la connexion

Pour cette section, nous avons deux parties.

1. Une partie permettant de vérifier que nous avons bien reçu les keep alive
2. Une partie permettant de se reconnecter au serveur en cas de problème

Keep Alive

Chaque fois que l'on va recevoir un message "hello" du serveur, un Timer nommé **timeWithoutMessage** va être déclenché. Du côté serveur, la connexion est rompue au bout de 60 secondes s'il y a un problème de connexion. Ici, nous allons donc donné 65 secondes au Timer. Si au bout de 65 secondes aucun message n'a été réceptionné, alors le Timer permettant de se reconnecter au serveur va être déclenché.

Le timer est lancé dès la connexion. Ensuite, si je reçois un message "hello", je peux annuler le timer, réceptionner le message et renvoyer un "hello" au serveur, et relancer le timer.

Listing 5.19– Code Keep Alive Android

```
1  if(payload.equals("hello")){
2     // cancelling the timer
3     tt.cancel();
4     timeWithoutMessage.purge();
5     // sending hello back to the server
6     mConnection.sendTextMessage(payload);
7     // lauching the timer again
8     tt = createTimeWithoutMessageTask();
9     timeWithoutMessage.schedule(tt, WAIT_WITHOUT_MESSAGE);
10 }
11 // when de 65 secondes are done
12 private TimerTask createTimeWithoutMessageTask(){
13    return new TimerTask() {
14       @Override
15       public void run() {
16          Log.d(TAG, "Waiting for a minute, trying de reconnect");
17          mConnection.disconnect();
18          reconnectToServer();
```

```
19          }
20      };
21  }
```

Lorsque les 65 secondes sont écoulées, le Timer exécute sa méthode run, qui va se déconnecter du serveur afin de mieux s'y reconnecter !

Ping/Pong frames

Le protocole des WebSockets spécifient que l'on peut utiliser les frames Ping et Pong pour maintenant la connexion client serveur. Le ping est envoyé depuis le serveur, via la command **sendPingFrame**, et la librairie Autobahn va automatiquement répondre par un Pong pour confirmer qu'on est toujours là. Ce sont d'ailleurs ces requêtes qui sont utilisés par Jetty pour calculer les fameux 10 minutes de connexion.

Le problème est que nous n'avons aucun contrôle sur le Ping reçu sur Android. En effet, c'est la librairie qui gère cela sans proposer de méthode de réception à l'instar de **onTextMessage**. J'étais au début parti sur cette voie, qui fonctionnait jusqu'à un certain point, puisque je ne pouvais pas calculer combien de temps s'écoulait entre les réceptions de Ping.

La gestion du Ping/Pong sur Autobahn est une requête demandée par les utilisateurs. Si le temps m'en avait permis, j'aurais pu me pencher sur ce cas afin d'une part l'utiliser dans mon projet et d'autre par contribuer au projet open source de cette librairie. Mais j'ai privilégié l'alternative qui consistait à implémenter moi-même la solution de keep alive.
https ://github.com/tavendo/AutobahnAndroid/issues/31

Reconnexion au serveur

La reconnexion se lance depuis deux situations :

1. Lorsque la connexion est correctement fermée. Ici, nous rentrons dans l'événement **onClose**
2. Lorsque la connexion n'est pas correctement fermée. Dans ce cas, c'est le timer des keep alive qui va relancer la connexion.

Lorsque la connexion se passe bien, par exemple si le serveur est arrêté et que la fin de connexion WebSocket est faite correctement, nous passons par la méthode **onClose** proposée par la librairie.

Listing 5.20– Code onClose Android

```
1   @Override
2           public void onClose(int code, String reason) {
3               Log.d(TAG, "Connection lost. "+reason+" error code : "+code);
4               // under 4000 it is managed by the library.
5               // we can custom our own codes if we want
6               if(code<4000){
7                 reconnectToServer();
```

```
8                     }
9                 }
```

La particularité, c'est que si l'on passe dans **onClose**, la connexion est déjà interrompue. Et si l'on essaie de se reconnecter directement, on repasse dans le onClose ! Ce qui fait une boucle infinie et tue l'application. C'est une particularité de Autobahn, qui à mes yeux ne fait pas tellement sens. Si la connexion n'existe plus, pourquoi retombe-t-on dans onClose ? En tout cas je n'avais pas tout de suite compris cela, ce qui m'a fait perdre passablement de temps. Pour contrer ce problème, j'utilise un boolean **goConnect**, qui est initialisé à **true**. Lorsqu'on tente de se reconnecter pour la première fois, on est autorisé. Puis l'on change le boolean à **false**, pour éviter la boucle infini, qui ne lancera ainsi le timer qu'une seule fois.

Listing 5.21– Code de reconnexion au serveur

```
1  private void reconnectToServer() {
2      try {
3        // boolean to check that the task is launched
4        // only one time
5        if(goConnect){
6          goConnect = false;
7          Thread.sleep(1000);
8          Log.d(TAG, "ReconnectTimer Launched");
9          // launching the Task
10         new ReconnectTask().run();
11       }
12     } catch (InterruptedException e) {
13       e.printStackTrace();
14     }
15
16   }
17     // task launching one time, when connetion is lost
18   private class ReconnectTask extends TimerTask{
19     @Override
20     public void run() {
21       try{
22         // if we don t wait for an hour
23         if(totalWaitTime<HOUR_TO_MS){
24           // if we are still deconnected
25           if(!mConnection.isConnected()){
26             // random waiting time between a min and max value
27             int waitTime= random.nextInt(MAX_TO_WAIT - MIN_TO_WAIT + 1) +
                   MIN_TO_WAIT;
28             Log.d(TAG, "Next tentative to connect in "+waitTime+" ms");
29             // calculate the total waiting time
30             totalWaitTime +=waitTime;
31             // schedule a new reconnection waitTime later
```

```
32                reconnectTimer.schedule(new ReconnectTask(), waitTime);
33                // trying to connect
34                connectToServer();
35              }else{
36                // if the task was launched but we are connected now
37                Log.d(TAG, "Connected to the server again");
38                // reinitializing parameters for next interruption
39                reinitializeReconnection();
40              }
41            }else throw new InterruptedException("Attempt to connect to the server
                  during 1 hours without success");
42          }catch(InterruptedException e){
43            Log.d(TAG, e.getMessage());
44          }
45        }
46
47    }
```

J'ai décidé que c'était la Task elle-même qui allait lancer le timer. Pourquoi ? Car nous avons un **temps aléatoire**. Or, il est impossible de base de lancer le timer de manière aléatoire. Nous devons lui donner des temps fixes.
L'autre avantage est que je n'ai qu'une seule instance de **ReconnectTask**, ce qui me permet d'avoir des variables à travers les différentes tentatives de connexion, typiquement **totalWaitTime** qui va accumuler tout le temps attendu, jusqu'à atteindre une heure de reconnexion.

Chapitre 6

Phase de tests production

Cette phase finale sert à intégrer les résultats sur Thom, l'outils de supervision, ainsi que de tester le projet sur une situation équivalente à la production. Avoir un serveur avec une adresse publique, plusieurs Set-Top Box qui se connectent au serveur, ainsi qu'utiliser une ligne externe pour la connexion à Internet. Malheureusement cette phase de tests n'a pas pu être exécutée faute de temps.

6.1 Thom

Thom est donc l'outils de supervision de Wingo permettant d'accéder aux données des clients. Toutes les données permettant la supervision et le dépannage sont regroupées sur ce site. Voici la partie **IPTV** d'un client test.

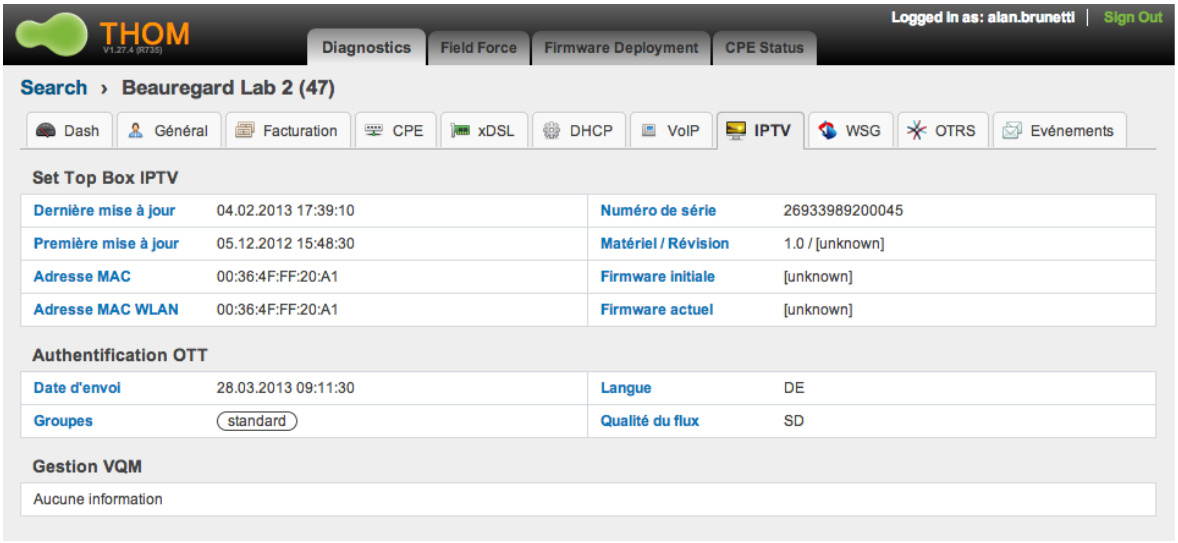


FIGURE 6.1 – Partie IPTV de Thom

Le but est de rajouter une partie **Remote Service** permettant de lancer les commandes sur la Set-Top Box via des boutons. Le résultat est le suivant :

FIGURE 6.2 – Ajout des boutons sur Thom

Nous voyons des inputs pour le ping et iperf permettant de mettre les options des commandes, alors que le reboot et isMute (que nous expliquerons au prochain chapitre) n'ont pas d'option.

Lorsque la commande est lancée, le résultat s'affiche juste en dessous.

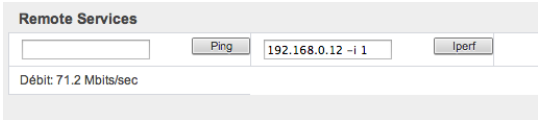


FIGURE 6.3 – Résultat iperf sur Thom

6.1.1 Conception

Thom est réalisé en HTML, avec entre autre l'aide de Slimebone, une version personnalisée par Wingo de Backbone, un framework Javascript facilitant l'utilisation de Web Services et de la manipulation de données au format JSon.
Nous allons utiliser deux éléments de Backbone, les **views** et les **models**

- **view** : Concerne tout ce qui est propre au HTML, la capture d'événements, la génération de code HTML dynamique
- **model** : Concerne la récupération de données sur un serveur à l'aide Web Service. Un model contient un paramètre URL qui est l'URL du Web Service auquel nous voulons accéder.

J'utilise 4 views et 4 models, chacun pour un service proposé. Nous allons nous concentrer sur Iperf, puis que le reste est identique au niveau de l'utilisation.

La view **Thom.Views.Stbs.Iperf** est la vue qui donc propre à Iperf. Elle réagira au clique sur le bouton correspondant, et appellera le model **Thom.Models.Stb** pour que lui aille chercher les données sur le serveur.

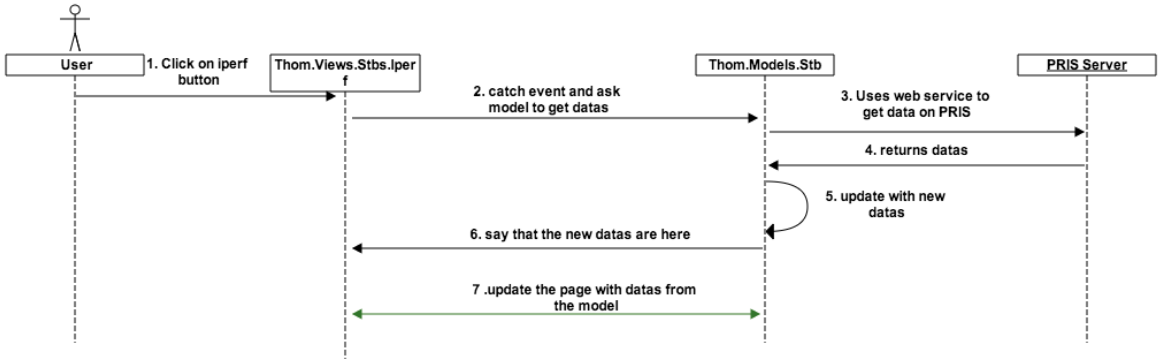


FIGURE 6.4 – Fonctionnement de Backbone

6.1.2 Implémentation

6.1.3 HTML de base

Ici, nous allons créer une nouvelle table HTML contenant les nos 4 boutons. Ceci est fait de manière statique, mais l'on pourrait tout à fait imaginer un Web Service nous retournant les commandes disponibles dans le futur !

Listing 6.1– Table HTLM

```
<table class='diagnostics vertical' id='qos_result'>
  <caption>Remote Services</caption>
  <tr>
    <th><input type='text' id='ping_input' value='ping cmd without the ping'/><a
        class='button' id='btn_ping' href='javascript:void(0);'>Ping</a></th>
    <th><input type='text' id='iperf_input' value='iperf cmd without the iperf'/><a
        class='button' id='btn_iperf' href='javascript:void(0);'>Iperf</a></th>
    <th><a class='button' id='btn_reboot' href='javascript:void(0);'>Reboot</a></th>
    <th><intput type='text' id='isMute_cell' disable/><a class='button' id='btn_isMute
        ' href='javascript:void(0);'>Is Mute</a></th>
  </tr>
</table>
```

Ce qui est important, ce sont sont les IDs. En effet, avec jQuery, nous pouvons facilement interagir avec le HTML en se basant sur l'ID des éléments, et Backbone aussi ! Par exemple, nous voyons que le clique sur un bouton ne fait rien, mais c'est la View qui va intercepter l'événement et le traiter.

Donc l'ID de la table est **qos_result**, l'ID du bouton est **btn_iperf** et l'ID du input est **iperf_input**.

Initialisation

Premièrement, il faut récupérer la valeur de la MAC adresse stockée dans la cellule dont l'ID est **stb_mac_addr**.
Ensuite, il faut instancier un Model pour Iperf, ainsi qu'une View. On passera en paramètre le Model à la View.
Tout ceci se fait dans la page HTML. Le script se lancera automatiquement au chargement de la page.

Listing 6.2– Initialisation du Model et de la Vue

```
<script type="text/javascript">
   $(function() {
    var macAddr = $('#stb_mac_addr').text();
    var iperfModel = new Thom.Models.Stb({path:'iperf/'+macAddr});
      new Thom.Views.Stbs.Iperf({model:iperfModel});
```

```
  });
</script>
```

Le paramètre **path** est le chemin relatif du Web Service par rapport à **http ://server_adresse/qosServer/rest**

View

Nous allons voir quel est le code derrière une View. Nous allons voir que cela fonctionne de manière clé/valeur, à savoir que l'on peut déclarer par exemple une fonction, dont le nom sera la clé, et la valeur sera l'implémentation de la fonction.

Listing 6.3– Code de la View

```
Thom.Views.Stbs.Iperf = Slimbone.View.extend({
  el: 'table#qos_result',

  initialize: function(){
    this.$list = this.$el.find("tbody:last");
    this.model.on('fetch:success' , this.renderSuccess , this);
  },

  events: {
    'click #btn_iperf': 'fetch'
  },

  fetch: function(){
    var value = $('#iperf_input').val();
    this.model.opt = value;
    this.model.cmd = 'iperf';
    this.model.fetch({
      success:this.renderSuccess
    });

  },
  cleanList: function() {
      this.$list.children().not("tr:first").remove();
    },
  renderSuccess:function(){
    console.log('fetch success');
    console.log(this.model);
    var values = this.model.val;
    this.cleanList();
        this.$list.append(
          '<tr>' +
     '<td> Debit: ' + values.throughput + ' '+values.unit+'</td>' +
     '</tr>'
```

```
34              );
35      }
36  });
```

Clé	Description
el	L'élément el est le tronçon HTML dont la View peut avoir accès. Cela veut dire que la View ne peut pas manipuler tous les éléments parents de el, mais les enfants oui. Nous lui donnons ici la table qos_result, dont on pourra donc manipuler tout le contenu.
initialize	Méthode appelée en tout premier lorsque l'on instancie la View. Il s'agit en quelque sorte d'un constructeur. Nous l'utilisons ici pour définir une variable **list**, qui est la fin de notre table HTML, puis nous disons d'appeler la méthode **fetch_success**lorsque le Model a correctement été chercher les données sur le serveur.
events	Permet d'interagir avec les événements produits par l'utilisateur. Ici, lorsque l'événement **click** intervient sur le bouton **btn_iperf**, on lance la méthode fetch.
fetch	Fetch veut dire "aller chercher". Lorsque l'utilisateur à fait un click, nous allons dire au Model d'aller chercher les données. Nous en profitons pour lui changer ses paramètres : on récupère les options écrites dans le input **iperf_input** et on lui précise que la commande sera bien un **iperf**, en vue de construire le Json envoyé au serveur.
cleanList	Permet de vide la table d'ancien résultat pour afficher par la suite les nouveaux.
renderSuccess	Lorsque le Model a fini de récupérer les données, cette méthode est appelée. On peut maintenant ajouter du contenu HTML avec les valeurs que le Model a récupérées sur le serveur.

TABLE 6.1 – Résumé d'une View

Model

Même principe ici. Sauf que nous n'avons pas un Model propre à chaque commande, mais un seul que nous personnalisons grâce aux paramètres différents qu'il prendra.

Listing 6.4– Code du Model

```
1   Thom.Models.Stb = Slimbone.Model.extend({
2
3     initialize: function(attributes) {
4           this.path = attributes.path;
5           this.opt = null;
6           this.cmd = null;
7           _.bindAll(this);
8       },
```

```
9       fetch: function(){
10
11        if(this.opt!==null){
12          this.jsonopt = "{\"opt\":\""+this.opt+"\",\"cmd\":\""+this.cmd+"\"}";
13          this.finalPath = 'http://localhost:8080/qosServer/rest/stb/'+this.path+'?
                cmd='+this.jsonopt;
14        }
15        else{
16          this.finalPath = 'http://server:8080/qosServer/rest/stb/'+this.path;
17        }
18        console.log('fetching to :'+this.finalPath);
19        Slimbone.ajax({
20          url:this.finalPath,
21          dataType: 'json',
22          success: this.commandSuccess,
23          error: this.commandError
24        });
25      },
26      commandSuccess: function(resp) {
27        this.val = resp;
28        this.trigger('fetch:success');
29      },
30
31      commandError: function(jqXHR, textStatus, errorThrown) {
32          console.log(textStatus);
33        }
34    });
```

Clé	Description
initialize	Comme pour la View, cette méthode permet d'initialiser le Model avec certains attributs. Dans notre cas, lors de sa création nous lui avions passé un paramètre **path** que pouvons ici récupérer. Ensuite, nous déclarons deux autres attributs : **opt**, le contenu de l'input mis à disposition pour les options s'il y en a, et **cmd** qui définira quelle commande est exécutée pour le serveur.
fetch	Cette fois, nous allons parler au Web Service. Nous construisons l'URL, en lui ajoutant le chemin **path**, puis des options en paramètre au format JSon. L'appel se fait à l'aide d'ajax et si tout se passe bien, nous allons dans la méthode commandSuccess, dans l'autre cas, la méthode commandError
commandSuccess	Lorsque le fetch se déroule correctement, nous mettons à jour l'attribut **val** avec les données récupérées depuis le serveur. Ensuite nous utilisons un trigger sur **fetch :success**. On se souvient que la View, va se mettre à jour lorsque ce trigger est déclenché (**this.model.on('fetch :success')**
commandError	Si le fetch ne se déroule pas correctement, le cas peut être traité ici.

Table 6.2 – Résumé d'un Model

L'URL finale peut donc ressembler à ceci :

```
http://server:8080/qosServer/rest/stb/iperf/00:09:DF:1B:FC:A5?cmd={"opt":"url_iperf_server","cmd":"iperf"}
```

Figure 6.5 – URL finale d'iperf pour Web Service

Chapitre 7

Conclusion

Voici le chapitre final, concluant ce travail de Bachelor. Je vais vous donner mes impressions personnelles, les problèmes rencontrer ainsi que les amélioration possibles de l'application.

7.1 Impressions personnelles

J'ai énormément aimé travailler sur ce projet et j'ai beaucoup appris.

Techniquement, j'ai pu découvrir énormément : l'utilisation des qualités de services dans une application, la communication à travers les WebSockets, la mise en place d'un serveur avec les Web Services. Bien que ce n'était pas une application Android la plus complète que l'on puisse faire, sans interface graphique, j'ai tout de même bien pu mettre en pratique les notions du développement sur ce système d'exploitation.

J'ai aussi pu réutiliser du Backbone pour la partie Web, bien qu'un tout petit peu, mais j'avais justement eu l'occasion d'en faire durant un projet de semestre et j'étais content de voir que je connaissais cette technologie, et que je n'ai pas été dépaysé lorsque j'ai dû la réutiliser !

Pour l'expérience en soit, j'ai été ravi de découvrir une entreprise telle que Wingo. J'ai vraiment été très bien accueilli et supporté tout le long du projet. Je remercie grandement Olivier qui a été mon responsable durant ce projet, ainsi que toute l'équipe avec qui je me suis bien entendu. J'ai d'ailleurs profiter des petits plaisirs en entreprise comme une petite journée sportive avec tout le monde !

C'était vraiment très intéressant et enrichissant, je recommande à tout le monde d'essayer !

Je remercie aussi Jean-Frédéric Wagen, Jean-Roland Schuler et Benoît Piller, avec qui les discussions durant les séances hebdomadaires se sont toujours bien déroulées et ont été constructives.

J'ai beaucoup redouté de commencer ce projet, car je ne savais pas chez qui j'allais atterrir et que la partie réseau des qualités de service me faisait un peu peur. Mais finalement tout s'est très bien déroulé et je ne regrette pas une seule seconde le choix que j'ai fait. Rarement un projet n'a été aussi passionnant du début à la fin

Par contre mes vieux défauts sont toujours là, notamment au niveau de la documentation que je n'ai pas tenu à jour durant toute la durée du projet. J'ai malheureusement quelques parties manquantes, notamment au niveau des résultats qui manquent ainsi que de l'analyse. Mais je sais que j'ai de quoi être fier du projet en soit, et que le travail est là.

Je suis par contre un peu déçu d'avoir finalement perdu du temps sur la partie du serveur, avec la gestion de la connexion et la connexion sécurisée, car quasiment tout le projet s'est déroulé sans encombre et dans les temps. Par contre dès qu'un peu de retard a été pris il a été difficile de le rattraper, surtout lorsque cela arrive sur la fin. J'ai pour le coup dû mettre de côté la phase finale des tests, ce qui est très dommage car j'aurais pu apprendre encore beaucoup !

7.2 Problèmes rencontrés

7.2.1 Déploiement sur la Set-Top Box

Le déploiement ne fonctionnait au début pas sur la box, alors qu'il fonctionnait sur la tablette. C'est Robert, développeur chez Swisscom, qui m'a dit de ne pas passer par Eclipse mais de le faire manuellement avec adb.

7.2.2 Installation de Jersey sur Jetty

L'installation m'a donné du fil à retordre à cause de compatibilité de versions. J'aurais dû mieux lire la documentation dès le départ, ce qui m'aurait préserver quelques heures d'arrachage de cheveux !

7.2.3 Mise en place de SSL

A cause apparemment d'un bug dans la version 9.0.3 de Jetty corrigé dans la 9.0.4 puis du fait que Autobahn Android ne faisait confiance à tous les certificats qu'en étant dans le simulateur d'ADB, j'ai perdu un précieux temps sur la fin du projet.

7.3 Améliorations

7.3.1 Récupérer automatiquement les qualités de services

Lorsque l'on arrive sur Thom, nous pourrions dynamiquement afficher les commandes que l'on peut envoyer sur notre STB, et non l'ajouter manuellement comme c'est le cas à présent.

7.3.2 Améliorer l'ergonomie de l'application

Ce qui a été implémenté sur Thom est très basique. Aucune vérification n'est faite, très peu de retour pour l'utilisateur et peu de gestion d'erreurs.

7.3.3 Voir la reconnexion d'une STB après reboot

Lorsque l'on redémarre à distance une box, nous n'avons pas de suivi. Cela veut dire que nous ne savons pas vraiment si tout s'est bien passé ou non. Il serait envisageable de pouvoir retourner lorsque la box est à nouveau connectée sur le serveur.

7.3.4 Ajout de commandes à distance

Plus qu'une simple analyse des qualités de service, c'est vraiment une prise en main à distance qui est possible. On l'a vu notamment avec le reboot. Ce n'est pas une qualité mais c'est utile de pouvoir le faire à distance. Tout un tas de possibilités s'offrent à nous pour le dépannage.

Liste des figures

Liste des tableaux

Liste des codes

Annexe A

Description de la structure du DVD

```
01_Cahier_des_charges -> contient les versions du cahier des charges
02_Rapport -> contient une version du rapport destinée à être lue
    Sources -> contient les sources latex du rapport
03_Planning -> contient toutes les versions du planning
04_PV -> contient tous les procès-verbaux des séances
06_Workspace -> contient les sources des applications
    client -> contient les sources de l'application cliente
    serveur -> contient les sources de l'application serveur
```