Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Domaine Sciences de l'ingénieur
Rte du Rawyl 47
CH- 1950 Sion 2
Tél. +41 27 606 85 11
Fax +41 27 606 85 75
info@hevs.ch

www.hevs.ch

Filière Systèmes industriels

Orientation Infotronics

Diplôme 2011

Romain Cherix

ActorDroid

Professeur	Pierre-André Mudry
Expert	David Jilli

Sion, le 15 juillet 2011

HES-SO Valais

SI	TV
X	X

Données du travail de diplôme
Daten der Diplomarbeit

FO 1.2.02.07.AB
pof/31/01/2009

☒ FSI ☐ FTV	Année académique / *Studienjahr* **2010/2011**	No TD / *Nr. DA* **it/2011/46**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*	Etudiant / *Student* **Romain Cherix** Professeur / *Dozent* **Pierre-André Mudry**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja [1] ☒ non / nein	Expert / *Experte* *(données complètes)* **David Jilli** Montagibert 8, 1000 Lausanne	

Titre / *Titel*

ActorDroid

Description et Objectifs / *Beschreibung und Ziele*

Les acteurs sont une technique de concurrence ayant été redécouverte récemment avec l'essor de langages tels que Scala. Toutefois, en dépit de leur puissance d'expressivité, ils restent largement absents des plateformes embarquées. Le but de ce travail de diplôme est de démontrer si le paradigme des acteurs est exploitable sur une plateforme comme Android notamment, afin de réaliser des systèmes distribués dans le domaine du *smart metering*.

En se basant sur du hardware existant (tablettes), les objectifs à atteindre dans ce travail sont :

- Discuter, documenter et analyser l'usage du langage Scala sur Android, notamment au niveau des outils de développement, des interfaces graphiques ou encore de l'accès aux différents services de l'OS
- Discuter, documenter et analyser l'usage du paradigme des acteurs sur Android, principalement au niveau des *remote actors*
- Réaliser un démonstrateur consistant en un petit système distribué utilisant des acteurs et mettant à disposition des informations de *metering*.

Délais / *Termine*

Attribution du thème / *Ausgabe des Auftrags:* 16.05.2011	Exposition publique / *Ausstellung Diplomarbeiten:* 02.09.2011
Remise du rapport / *Abgabe des Schlussberichts:* 15.07.2011 | 16h00	Défense orale / *Mündliche Verteidigung:* dès la semaine 36 / *ab Woche 36*

Signature ou visa / *Unterschrift oder Visum*

Responsable de l'orientation
Leiter der Vertiefungsrichtung: .....

[1] Etudiant/*Student*: .....

[1] *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive et le caractère confidentiel du travail de diplôme qui lui est confié et des informations mises à sa disposition.*
Durch seine Unterschrift verpflichtet sich der Student, die Richtlinie einzuhalten sowie die Vertraulichkeit der Diplomarbeit und der dafür zur Verfügung gestellten Informationen zu wahren.

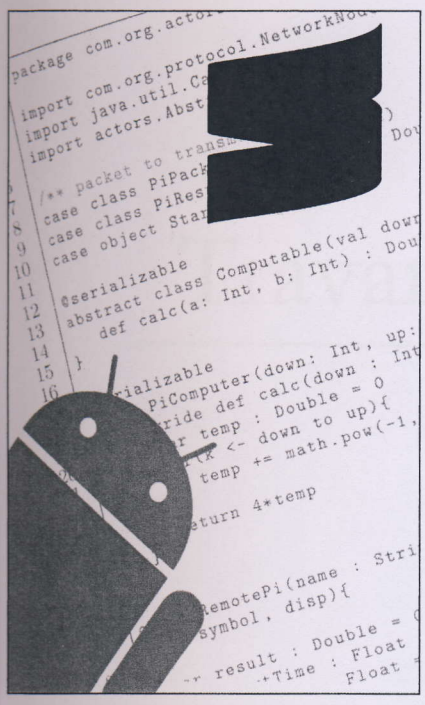
Rapport reçu le / *Schlussbericht erhalten am* 15 JUIL. 2011 Visa du secrétariat / *Visum des Sekretariats*

Hes·so VALAIS WALLIS
Rte du Rawyl 47 - 1950 Sion 2

Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

ActorDroid

Diplômant Romain Cherix



Travail de diplôme
| édition 2011 |

Filière
Systèmes industriels

Domaine d'application
Infotronics

Professeur responsable
Dr. Mudry Pierre-André
pandre.mudry@hevs.ch

Objectif du projet

Ce projet de diplôme a pour but de tester les possibilités d'utilisation du langage de programmation Scala sur un environnement Android, notamment pour l'utilisation des « *remote actors* ». L'objectif final était de créer un démonstrateur constitué d'appareils mobiles sous Android au sein d'un système distribué mettant à disposition des services basé sur des acteurs, permettant d'une part de faire du calcul distribué parallèle et, d'autre part, de partager des informations de *metering* en temps réel.

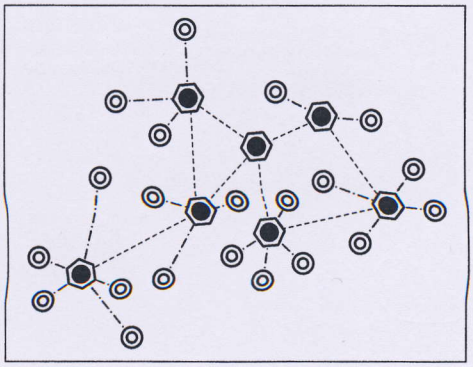
Méthodes | Expériences | Résultats

Ce travail exploratoire se compose essentiellement d'implémentation. Par le biais de différents programmes réalisés, nous avons pu démontrer le fonctionnement des outils mis à disposition par Android, notamment au niveau des interfaces graphiques lorsque celles-ci sont programmées en Scala.

Une *toolchain* semblable à celle de Google a été mise en place afin de compiler du code Scala en application compatible Android. Grâce à cet outil, nous avons montré qu'il était possible de réaliser dans ce langage des programmes utilisant l'API Android, l'API Google ainsi que d'intégrer le paradigme des acteurs qui font partie des librairies Scala.

Dans un deuxième temps, un protocole a été créé afin de pouvoir implémenter des programmes distribués utilisant des acteurs à distance. Ce protocole permet de créer un réseau auto-organisé d'appareils sous Android réalisant différents services basés sur des acteurs, permettant ainsi d'implémenter des applications s'échangeant des données ou de faire du calcul distribué.

Au final, ce projet a permis de démontrer que le paradigme des acteurs en Scala était viable sur Android et que son utilisation permettait de créer des programmes complexes et pertinents.



HES-SO Valais
Route du Rawyl 47
1950 Sion

Tél. 027 606 85 11
URL www.hevs.ch

HES-SO
SYSTEME INDUSTRIELS
INFOTRONICS

Travail de diplôme 2011

ActorDroid

Romain Cherix

Table des matières

1 Introduction

De nos jours, les systèmes embarqués possèdent de plus en plus de puissance et se répandent à grande échelle, notamment aux travers des téléphones portables. Ces appareils, dont certains possèdent plusieurs cœurs et des centaines de Mo de RAM[1], donnent des possibilités inenvisageables dans le passé comme l'utilisation de machines virtuelles gourmandes en ressources. Ainsi, le système d'exploitation Android de Google, équipant de plus en plus d'appareils mobiles[2], met à disposition une machine virtuelle spécialement conçue pour les systèmes embarqués afin de pouvoir exécuter des programmes codés en Java. Les avantages liés à l'utilisation d'une machine virtuelle sont nombreux : ramasse-miettes (*garbage collector*), *hotspot compiling*, abstraction du matériel, etc. et l'on comprend son intérêt dans le contexte d'un OS embarqué.

Dans un même temps, les abstractions offertes par les langages de programmation de haut niveau mettent à disposition des fonctions complexes aisément exploitables. C'est le cas de Scala[3], un langage de très haut niveau fonctionnant sur une machine virtuelle Java[4]. Grâce à cet héritage, ce langage possède l'une des plus grandes librairies parmi les langages existants. Les acteurs [Haller, 11] font partie de ces librairies. "Redécouverts" récemment, ils sont une technique de concurrence, très puissants de par leur expressivité, mais à notre connaissance complètement absents des plateformes embarquées.

Le but de ce projet est de démontrer si le paradigme des acteurs, notamment dans le domaine du calcul distribué et parallèle, est réalisable sur un appareil embarqué.

Ce rapport possède quatre parties distinctes traitant des différents points explorés durant le projet. La première partie traite des différentes solutions possibles afin de mettre en œuvre une toolchain complète capable de transformer du code source Scala en application fonctionnant sur Android. La seconde partie aborde l'utilisation des différentes API d'Android (système, GUI, …) et plus particulièrement leur programmation en Scala. La troisième partie traite de l'utilisation des acteurs sur un système d'exploitation Android. Finalement, la dernière partie aborde l'usage des acteurs *remote* au travers d'une application fonctionnant sur un protocole implémenté spécialement pour les appareils mobiles, utilisé pour mettre des acteurs *remote* à disposition afin de les utiliser dans des applications tel que le calcul distribué ou encore le smart metering.

[1] http://www.techyou.fr/2011/05/smartphones-les-nouvelles-collections-2011
[2] http://www.memoclic.com/1217-android/12645-succes-android.html
[3] http://www.scala-lang.org/
[4] http://infoscience.epfl.ch/record/52211

2 Objectifs

- Discuter, documenter et analyser l'utilisation du langage de programmation « Scala » sur la plateforme Android, notamment au niveau des outils de développement, des interfaces graphiques ou encore à l'accès aux différents services de l'OS ;
- Discuter, documenter et analyser l'usage du paradigme des acteurs sur Android ;
- Discuter, documenter et analyser l'usage des acteurs *remote* sur Android au travers d'une application de démonstration.

Ce rapport contient tous les éléments décrivant les démarches et réflexions effectuées durant l'exécution de ce travail. Il contient également des explications de tous les programmes implémentés, ainsi que leurs codes en annexe.

Certaines des applications implémentées durant ce travail ont été créées afin de tester le bon fonctionnement de différentes particularités directement liées à Android. Les interfaces graphiques, l'ergonomie ainsi que leurs mises en œuvre sont celles de proof-of-concept et ne se veulent pas sans failles.

3 Le langage Scala

3.1 Introduction

Scala est un langage de programmation récent, à typage statique, qui compile vers du bytecode Java et qui est exécuté sur la machine virtuelle Java (ou JVM). Son nom résulte de la contraction de l'expression anglaise « **SC**alable **LA**nguage », qui signifie langage évolutif ou « langage qui peut être mis à l'échelle ».

Ce langage est conçu pour être polyvalent, puissant et utilisable tant pour des petits que pour des grands logiciels. Son créateur, Martin Odersky, est professeur à l'École Polytechnique Fédérale de Lausanne. Ses travaux visent notamment à unifier la programmation orientée objet et la programmation fonctionnelle, l'idée sous-tendant cette approche étant que ces deux paradigmes sont complémentaires et non pas contradictoires. Afin de démontrer la validité du concept, il a réalisé plusieurs langages de programmation comme *Pizza*, *GenericJava* et *Functional Nets*. Ces développements ont donné naissance à Scala en 2001 qui a vu sa première version finale être publiée en 2003.

Au contraire des langages s'appuyant sur la *complexité du langage* afin de pouvoir offrir le plus de souplesse aux utilisateurs-trices, Scala propose une approche différente. En effet, Scala se base sur l'idée d'un langage relativement succinct (le nombre de mots-clés vis-à-vis du C++ est beaucoup plus petit par exemple) mais très puissant et *extensible*. Mélangeant l'approche orienté objet et la programmation fonctionnelle, ce langage à écriture concise reprend le meilleur de Java, C#, Eiffel, Haskell, Erlang, ML ou même SmallTalk. Récemment, il a été utilisé pour des projets de grandes envergures par des sociétés, tel que Twitter, Linkedin, SAP, eBay ou Siemens.

Scala possède deux particularités notables. D'une part, Scala est un langage objet pur, ou il n'existe pas de type primitif comme les `int`, les `char` ou `byte`, mais tout est objet. Grâce à ceci, il est possible de surcharger n'importe quel opérateur afin de simplifier le code. Sa syntaxe simple et élégante est également beaucoup moins verbeuse que Java, C# ou C++, les ";" ne sont pas indispensables et les types n'ont pas besoin d'être déclarés explicitement grâce à l'inférence de types.

D'autre part, Scala supporte le paradigme fonctionnel, les fonctions sont des objets comme les autres. Elles peuvent ainsi être passées en arguments à d'autres méthodes ou encore être stockées dans des variables. Il est possible de compiler du Scala pour être exécuté sur une machine virtuelle .NET ou en bytecode afin de l'utiliser sur une machine virtuelle Java. Il est ainsi transportable sur de multiples plateformes et entièrement interopérable avec du Java. Grâce à ceci, il est concevable de programmer certaines classes en Java et d'autres en Scala dans un même programme de manière totalement transparente [Mudry, 11].

Ce langage propose également une des plus grandes librairies mise à disposition parmi tous les langages existant. Grâce à ceci, Scala permet par exemple de programmer avec une vue très haut niveau tout en utilisant des fonctions très complexes. Les librairies récentes de Scala exploitent au maximum les collections parallèles ce qui optimise considérablement le temps d'exécution[5]. Ces collections parallèles sont directement implémentées dans les libraires et prennent en compte l'architecture du processeur afin d'accélérer les opérations. C'est par exemple le cas des `List` en Scala, qui est un élément couramment utilisé.

[5] http://www.scala-lang.org/node/9483

4 Android

4.1 Introduction

Android[6] est un système d'exploitation *open source* basé sur un noyau Linux racheté par Google en 2005. Principalement utilisé sur des téléphones mobiles, ses atouts, tels que sa gratuité, son code ouvert et la possibilité de programmer en Java le rendent extrêmement puissant et lui permettent d'évoluer rapidement. Ce système d'exploitation met également des librairies à disposition, permettant d'interagir directement avec l'appareil (nommé « terminal » dans la suite de ce rapport). Les services Google, regroupé sous le nom « d'API Google » permettent d'utiliser des outils comme Google Maps directement sur l'appareil.

La plupart des applications développées sont programmées en Java et fonctionnent sur une machine virtuelle, appelée « Dalvik Virtual Machine » (DVM). Cette machine virtuelle est conçue à la base pour des appareils mobiles et est particulièrement adaptée pour des systèmes contenant peu de mémoire et peu de puissance de calcul. Cette machine virtuelle interprète du bytecode de type *dex*, comparable à du bytecode de type *class*, mais optimisé lors de sa compilation. Le nombre d'opérations pour un programme sont réduits de 20 % par rapport à un fichier *class*, notamment en réduisant le nombre d'accès à la mémoire. Il est ainsi plus rapide à l'exécution et moins gourmand en ressources.

Android amène une particularité avec l'utilisation de la DVM car chaque processus possède sa propre instance de machine virtuelle, ce qui à l'avantage d'augmenter la stabilité du système en cas de crash d'une application. Cependant la communication entre classes de différentes applications ne peut pas se faire comme pour une machine virtuelle standard mais doit se faire avec les outils mis à disposition par Google.

[6] http://en.wikipedia.org/wiki/Android_(operating_system)

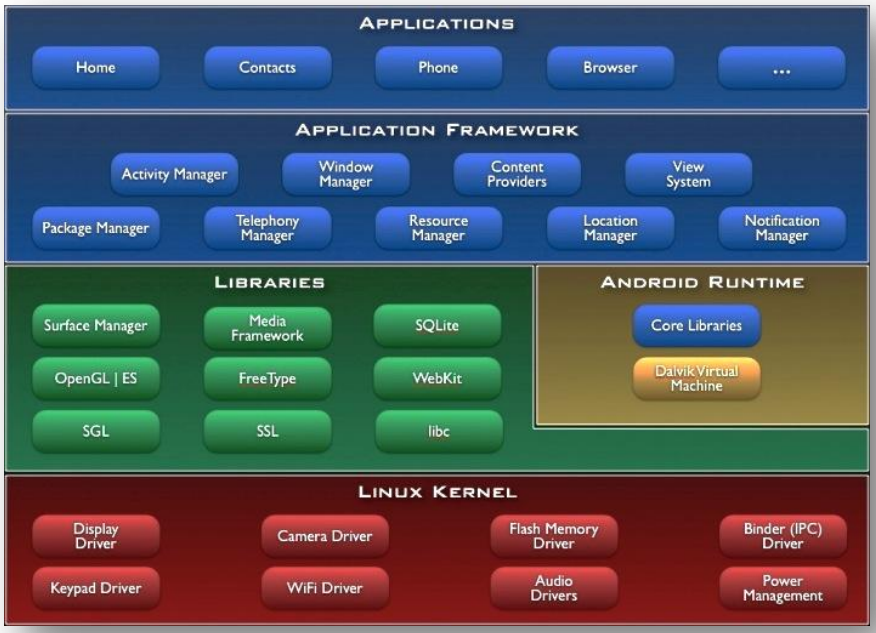


Figure 1 – Architecture Android
Tiré de http://developer.android.com/guide/basics/what-is-android.html

La figure ci dessus représente l'architecture du système Android avec ses différentes couches. La couche la plus basse est constituée du noyau basé sur Linux en version 2.6. Les couches supérieures sont constituées des libraires mises à disposition par Android ainsi que l'environnement d'exécution. Au dernier niveau se situent enfin les applications fonctionnant sur les machines virtuelles. Durant ce projet, nous allons travailler uniquement à ce niveau.

4.2 Applications Android

Il existe deux familles d'applications Android, les `Activity` et les `Service`. D'un point de vue de la programmation, ce sont deux super classes qui permettent d'exécuter un programme mais de manière différentes et dont il est nécessaire d'hériter lors de la création d'une application. Elles comportent entre autres les méthodes `onCreate` et `onDestroy` permettant de définir les opérations à effectuer lors de la création et de la destruction du thread. Au niveau de la différence de ces classes, une activité gère la partie graphique de l'application alors qu'un service s'occupe de toutes les opérations qui doivent être effectués en arrière plan.

4.2.1 Activités

Une activité est un concept simple, porté sur ce que l'utilisateur peut faire. En générale, une activité donne la possibilité à l'utilisateur d'interagir avec l'application au travers d'une fenêtre sur lequel il est possible de disposer des widgets. C'est elle qui contient la structure d'un programme Android et qui permet de le faire vivre au travers de différents états. En effet, le cycle de vie d'une activité est assez particulier comme nous allons le voir.

4.2.1.1 Cycle de vie d'une activité

Sur Android, les activités possèdent différents état afin d'optimiser le fonctionnement sur les systèmes embarqués. Ceci permet d'utiliser le minimum de ressources lorsque les applications fonctionnent et ainsi optimiser le fonctionnement des applications nécessitant des ressources. Cela permet également de minimiser les besoins en termes de mémoire et d'énergie.

Le diagramme d'états ci-dessous illustre de manière claire les états qu'une `Activity` peut prendre depuis sa création jusqu'à sa mort.

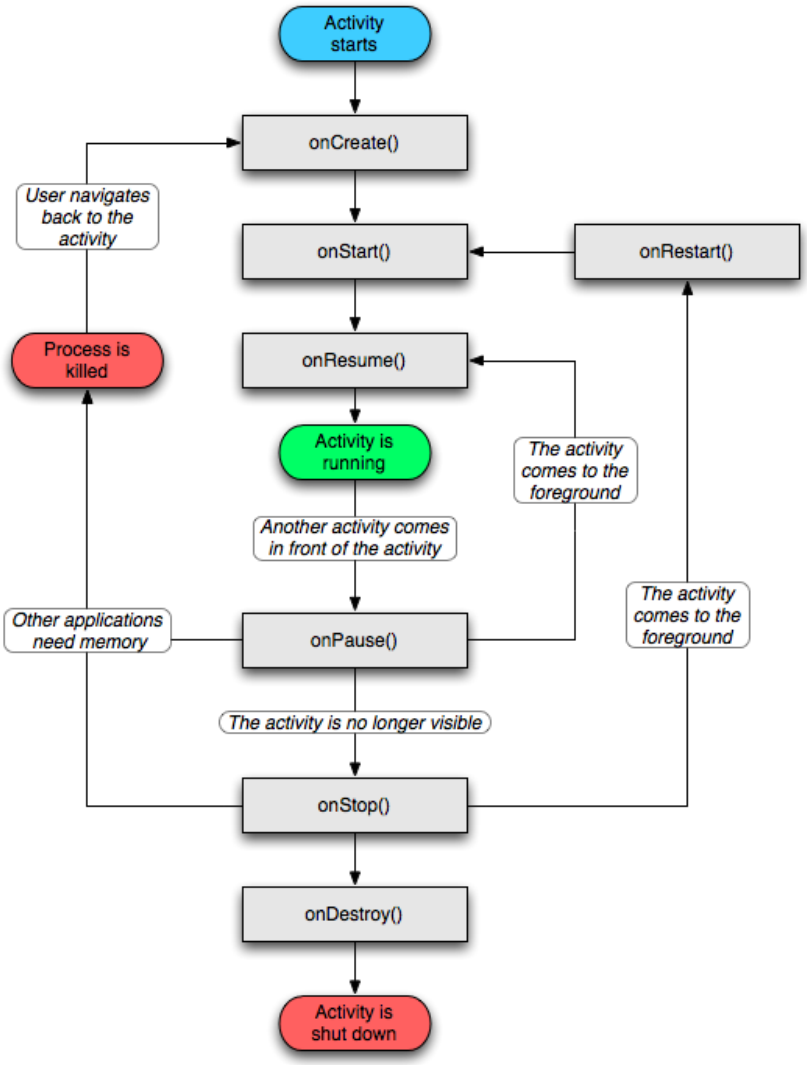


Figure 2 - Cycle de vie d'une activité
Tiré de http://developer.android.com/reference/android/app/Activity.html

Tous les états sont décrits ci-dessous.

- `onCreate()` :
  - Exécuté lorsque l'utilisateur lance une application ;
  - Initialisation de la vue XML, des fichiers/données temporaires;
- `onRestart()` :
  - Exécuté lorsque l'application sort de onStop() ;
- `onStart()` :
  - Exécuté après chaque onCreate() ou onRestart(), recharge les données sauvegardées avant le dernier arrêt.
- `onResume()` :
  - Exécuté après chaque onStart() et à chaque passage en premier plan de l'activité;
  - Initialisation et mise à jour des données qui auraient été modifiées entre temps.
- `onPause()` :
  - Exécuté avant chaque onStop() ;
  - Exécuté en cas de changement d'activité, ou lors d'un « finish() » sur l'activité ;
  - Libération des ressources et sauvegarde des données susceptibles d'être perdues.
- `onStop()` :
  - Exécuté avant chaque mise en sommeil ;
  - Exécuté avant chaque onDestroy() ;
  - Libération des ressources.
- `onDestroy()` :
  - Exécuté lors de la destruction de l'activité. ;
  - libération des ressources et des fichiers temporaires.

4.2.2 Services

Un service est un composant d'application, qui ne comporte pas d'interface graphique, servant à effectuer des opérations en arrière-plan. Des services peuvent également être implémentés afin de mettre à disposition des fonctionnalités à d'autres programmes. En général, un service est démarré depuis une activité grâce à une `Intent` (voir chapitre 8) afin de séparer la vue du traitement dans une application. Il est également possible de démarrer un service de différentes manières :

- Au démarrage du téléphone;
- A l'arrivée d'un événement (arrivée d'un appel, SMS, mail, etc.);
- Au lancement d'une application.

4.2.2.1 Cycle de vie d'un service

Le cycle de vie d'un service comporte uniquement deux états, `onCreate` et `onDestroy` appelé lors de sa création, respectivement de sa destruction. Il n'est possible de n'avoir qu'une seule instance d'un service, même si celui-ci est démarré plusieurs fois.

5 Matériel de développement

Android met à disposition un émulateur[7] pouvant fonctionner sur PC mais il est plus rapide est aisé de s'équiper d'un terminal afin de pouvoir tester toutes les applications développées dans des conditions réelles. Nous avons déterminé en début de projet que le terminal doit en outre avoir les caractéristiques suivantes :

- Connectivité sans fil Wifi;
- Android 2.1 minimum avec suivi de mise à jour si possible;
- Driver de débogage intégré dans la ROM;
- Prix raisonnable.

Le choix d'une tablette à été privilégié à un Smartphone, un écran plus large augmentant la surface de travail et permettant ainsi un meilleur confort. En prenant compte de ces caractéristiques, les tablettes suivantes ont été retenues :

- Samsung Galaxy tab[8];
- Toshiba Folio 100[9];
- Archos 7[10].

Les différences entre ces trois tablettes se situent au niveau du hardware (taille de l'écran, processeur, RAM). Notre choix s'est finalement porté sur la Toshiba Folio 100, sa configuration avancée (processeur Tegra 2, 1Go de RAM) et son faible coût la différencie des autres terminaux.

Cette tablette à été flashée et rootée avec une ROM personnalisée. Cette opération va nous permettre d'avoir accès à différents éléments du système normalement bloqué pour un simple utilisateur. Ceci est nécessaire afin d'avoir accès à différents éléments du système qui seront utilisés sur certain programmes, tel que l'accès au fichier /proc/stat affichant les informations liées au hardware (CPU, RAM, etc.).

[7] http://developer.android.com/guide/developing/tools/emulator.html
[8] http://www.samsung.com/global/microsite/galaxytab/
[9] http://www.toshiba-multimedia.com/fr/media-tablet/
[10] http://www.archos.com/products/ta/archos_7/index.html?country=mc&lang=fr

6 Toolchain

6.1 Introduction

Dans cette première partie, nous allons expliquer comment nous avons mis au point une toolchain complète afin de pouvoir compiler et exécuter des applications Scala sur Android. La compilation complète d'une application Android est complexe car de multiples fichiers entrent en jeux et de nombreuses opérations doivent être effectuées afin de pouvoir transformer un code source en application exécutable sur un terminal Android.

Le but de cette étape est de crée une toolchain semblable à celle mise à disposition par Google. Elle est capable d'effectuer des opérations redondantes complètes afin d'accélérer de développement d'une application lorsque celle-ci doit être testée. Il est ainsi possible, grâce à une seule commande, de compiler le code, le convertir en bytecode *dex*, d'en faire une apk et de l'exécuter sur un terminal.

6.2 Structure d'un projet Android

Un projet Android est organisé sous la forme d'une arborescence de répertoires spécifiques à un projet [Murphy, 10], comme tout projet Java. Il est indispensable de respecter cette arborescence pour pouvoir utiliser la toolchain mise à disposition par Google ou pour créer des programmes en utilisant des IDE tel qu'Eclipse, Netbeans ou IntelliJ. La structure d'un projet est illustrée ci-dessous :

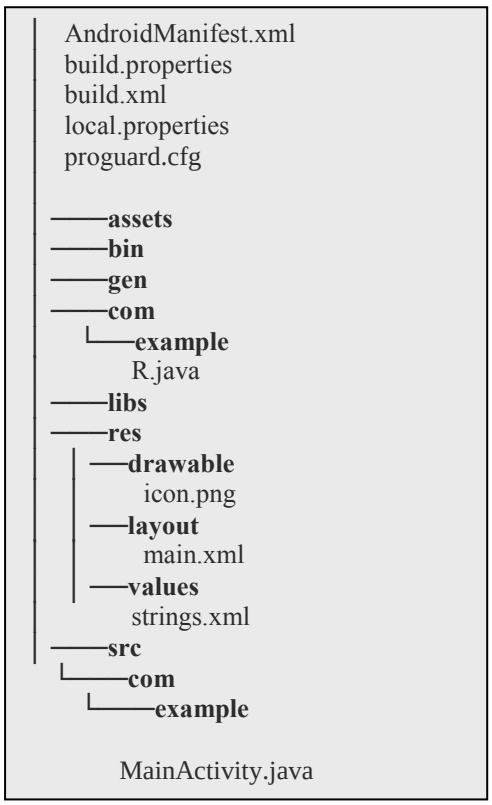


Figure 3 - structure projet Android

- **AndroidManifest.xml**

L'AndroidManifest est un fichier XML décrivant l'application à construire et les composants de celle-ci, tel que les activités, les services, les permissions.

- **Build.xml, default.properties, local.properties**

Ces fichiers sont des scripts utilisés par ANT pour la compilation ou pour toutes les opérations liées à la toolchain. Le fichier Build.xml est important, c'est notamment grâce à lui qu'il va être possible de créer une toolchain complète, car il comporte toutes les informations nécessaires pour mener un code sources à travers les différentes étapes.

- **Proguard.cfg**

Ce fichier contient différents paramètres du programme Proguard

- **Assets/**

  Contient les fichiers statiques fournis avec l'application pour son exécution sur le terminal

- **Bin/** contient l'application compilée ainsi que le .apk
- **Gen/** contient le code source produit par la compilation du projet
  - **R.java** est un fichier contenant des références vers différents éléments du projet liés à l'interface graphique ou aux variables externes, il est obtenu à partir de la compilation du projet
- **Libs/** contient les fichiers JAR extérieurs nécessaires à l'application
- **Src/** contient le code source Java de l'application
  - **MainActivity.java** est le code source du projet Android
- **Res/** contient les ressources (icônes, layout, variables externes) assemblées avec le code Java compilé
- **Tests/** contient un projet Android entièrement distinct, utilisé pour tester le projet créé.

6.3 Description et illustration

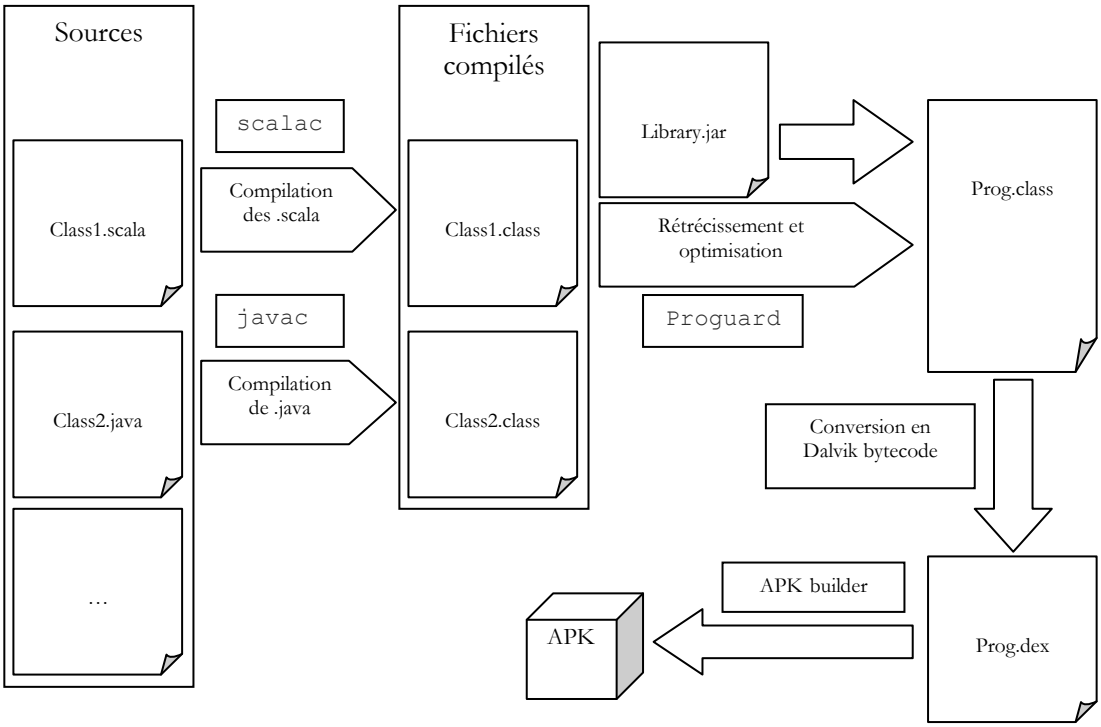


Figure 4 - Toolchain

Tout les fichiers source Java et scala seront compilés grâce au compilateur Javac [11] respectivement Scalac[12] afin d'en créer des fichiers *class*. Les fichiers *class* sont des fichiers de type bytecode, exécutable sur une JVM. Ils vont par la suite être optimisés grâce à Proguard. Les fichiers *class* optimisés vont être traduits en *dex* afin de pouvoir fonctionner sur une machine Dalvik. La dernière opération consiste à créer une archive apk qui contient tous les éléments nécessaires pour pouvoir fonctionner correctement sur une machine Android.

6.4 Création de la toolchain

Il existe plusieurs programmes capables d'automatiser des opérations redondantes définies à l'avance afin de créer une toolchain. Dans ce projet, nous avons étudié ANT et SBT. Deux outils d'automatisation de processus afin de crée une toolchain complète.

[11] http://download.oracle.com/javase/1,5.0/docs/tooldocs/solaris/javac.html
[12] http://www.scala-lang.org/docu/files/tools/scalac.html

6.5 Another Neat Tool (ANT)

ANT [13] (**A**nother **N**eat **T**ool) est un outil open source créé par la fondation Apache. Implémenté en Java, il est utilisé pour automatiser des opérations répétitives lors du développement d'un logiciel. Son fonctionnement est proche de celui de Make[14], il est donc capable de gérer les dépendances entre les tâches et de recompiler uniquement les fichiers nécessaires. Ce logiciel peut également être utilisé pour générer des documents Javadoc, des rapports et même utiliser des outils annexes.

ANT se base sur un fichier XML (build.xml) pour définir les différentes opérations à exécuter sur des fichiers sources. Il est ainsi possible de l'utiliser pour de multiples langages car l'utilisateur à la possibilité d'éditer son propre fichier de configuration.

Différents scripts XML permettant de créer cette toolchain sont disponibles sur internet. Pour ce travail, je me suis basé sur la toolchain créée par l'EPFL disponible sur le site assembla[15]. Son fonctionnement est le suivant : le build.xml comportera une dépendance sur un autre fichier nommé build-scala.xml qui décrira toutes les opérations nécessaires à la compilation du code Scala, mais également des opérations nécessaires pour créer une interaction avec un terminal Android, tel que la génération d'une apk, la signature d'une apk, l'installation du programme sur un terminal, etc. Ce script est également capable de compiler des projets mixtes Java et Scala.

Une fois tous les paramètres de la toolchain mis en place, il est possible de compiler des applications codées en Scala. Cet outil est souple d'utilisation et permet de modifier ses paramètres très simplement. Il conserve également la structure de base d'un projet Android ce qui rend possible l'exportation ou l'importation d'un projet depuis un IDE.

[13] http://ant.apache.org/
[14] http://www.gnu.org/software/make/
[15] http://www.assembla.com/wiki/show/scala-ide/Developing_for_Android

6.6 Simple Build Tool

Le deuxième outil considéré dans ce travail a été SBT [16](**S**imple **B**uild **T**ool). SBT est un outil open source, développé par un particulier, entièrement codé en Scala et conçu afin de gérer le développement de projets Scala. Il est en outre capable de :

- Compiler, tester et publier différents types de programmes Scala ;
- Gérer ou télécharger des dépendances à un projet si celles-ci ne sont pas présentes ;
- Compiler des projets mixtes Java/Scala.

De nombreux plugins sont disponibles permettant d'apporter des fonctions supplémentaires, comme la création d'archive jar, l'intégration de langages supplémentaires ou même des outils simplifiant le débogage. Parmi ces plugins, il en existe un capable de gérer des projets en Scala pour Android, il s'agit de « jberkel» [17]. Ce plugin permet de compiler du code Scala et d'en faire une application compatible Android. Il met également à disposition une série de commandes permettant de gérer le projet (compilation, création d'une apk, installation du programme sur une cible, etc.…). C'est grâce à ce plugin que nous pourrons coder des applications en Scala sur Android avec SBT. Une fois installé et paramétré, ce plugin s'utilise très simplement au travers de commande utilisé avec SBT.

SBT utilise son propre système de hiérarchisation. Le projet père contient les plugins ainsi que les dépendances du projet et le projet fils qui contient les fichiers du projet. Le seul outil pré-requis afin d'utiliser SBT est Android-SDK[18], tous les outils nécessaires à la création et à la mise en œuvre d'un projet étant automatiquement téléchargés par SBT.

6.6.1 Arborescence du projet Android avec SBT

```
|
|  ---project
|     |  ---build
|     |     |---Foo.scala
|     |  ---build.properties
|     |  ---plugins
|     |        Plugins.scala
|  ---src
|     |  ---main
|     |     |  ---AndroidManifest.xml
|     |     |  ---assests
|     |     |  ---java
|     |     |  ---res
|     |     |        |---drawable
|     |     |        |---layout
|     |     |        |---values
|     |     |        |---xml
|     |     |  ---scala
|     |     |        |---Activity.scala
|  ---tests
```

L'arborescence est repartie dans deux dossiers distincts. Dans le dossier project se trouve tous les éléments nécessaires à SBT pour la compilation et la publication (version d'Android, version de Scala, version de Proguard, clés pour le market, etc.) le dossier src quant à lui contient l'ensemble des éléments définissant le projet Android.

---

[16] http://code.google.com/p/simple-build-tool/
[17] https://github.com/jberkel/giter8
[18] http://developer.android.com/sdk/index.html

6.7 Proguard

Proguard[19] et un programme Java sous License GPL. Il permet d'optimiser un code prévu pour fonctionner sur une JVM en le rétrécissant. Cette manipulation effectue une analyse des classes et librairies incluses dans les différents fichiers afin d'en utiliser que le strict nécessaire. Il permet également d'optimiser le code en analysant s'il existe du « dead code », des variables ou classes non utilisées, et même de simplifier des opérations.

Proguard utilise un fichier de configuration externe permettant de spécifier des options lors d'un traitement de code. Il est en outre possible de spécifier les optimisations à effectuer (optimisation algorithmique, des variables/constantes, etc.…) et le nombre de passes à exécuter. Proguard est également capable de repérer des fonctions effectuant le même type d'opération et de les factoriser.

En plus de cela, il possède également une fonction pour brouiller les fichiers compilés afin d'éviter le reverse engineering du code. Ce brouillage ne rend pas le code illisible, mais le déchiffrage est beaucoup plus complexe.

L'utilisation de Proguard est hautement recommandée afin de réduire la taille des applications Android codée Scala. Sans cette étape, l'application prendrait beaucoup de place sur un appareil mobile, réduisant ainsi son efficacité. Différents tests ont été effectués permettant de vérifier la pertinence de l'utilisation de Proguard. Ces tests sont visibles sur le programme de l'utilisation des API [chapitre 7.3] ainsi que le programme utilisant les acteurs *remote* [chapitre 11.11.2].

6.8 Conclusion

Deux outils existent pour compiler du Scala sur Android, ANT et SBT. Ces deux outils ne peuvent pas faire ces opérations nativement, mais par le biais de plugins ou en modifiant des fichiers de configuration. Chacun de ces deux outils possèdent des avantages et des inconvénients. SBT par exemple est capable de modifier la version du compilateur à utiliser de manière dynamique tout en continuant à développer le projet. ANT quant à lui permet de garder la structure des projets Android définie par Google, ce qui permet de facilement transformer un projet Java existant en projet Scala.

De manière générale, les deux outils proposent les mêmes opérations, fonctionnent à la même vitesse et le résultat final est équivalent. Toutefois, ANT a été choisi pour les implémentations. En effet il est plus souple que SBT et la modification des paramètres se fait facilement dans des fichiers de configuration externe à ANT. Il conserve également la structure de base d'un projet Android, permettant ainsi d'exporter un projet sur un IDE ou d'importer des projets depuis des fichiers sources.

[19] http://proguard.sourceforge.net/index.html

7 Utilisation des API

7.1 Introduction

Dans cette deuxième partie, nous allons discuter de l'utilisation des différentes API lorsque celles-ci sont exploitées en Scala. Google et Android mettent à disposition de multiples outils permettant d'exploiter de manière optimale l'appareil sur lequel le système fonctionne. Ces outils étendent de façon considérable les possibilités pour un programmeur, car tout est accessible à travers les API. Il est ainsi possible, en quelques lignes de codes de passer un appel téléphonique, envoyer un SMS ou encore faire une recherche sur Google.

Afin de démontrer que ces services fonctionnent avec une application codée en Scala, des programmes de localisation GPS ont été implémentés. Ce logiciel permet de tester de multiples outils Android, tel que la localisation par GPS/A-GPS faisant partie de l'API Android, ainsi que l'affichage de la position sur Google Maps en utilisant l'API Google.

Deux programmes ont été implémentés. Le premier est directement traduit d'un programme Java disponible dans un tutorial[20] [annexe 4]. Le second [annexe 5] est basé sur le premier programme avec différentes modifications, dont l'affichage des coordonnées sur une carte Google Maps. C'est sur ce dernier que les discussions ont été faites. Les différences de code entre ces deux programmes est minime et se situe qu'au niveau de quelques lignes. C'est pour cette raison que seul le deuxième programme est annexé.

7.2 Implémentation

Le programme fonctionne sur deux activités. L'activité principale initialise l'interface graphique permettant de lancer la localisation et de l'afficher sous forme de coordonnées GPS. La deuxième activité, appelée au moyen d'un `URI`, affiche sur une carte Google Maps la position localisée auparavant.

L'activité principale utilise un `LocationListener` [annexe 5, ligne 20] afin que l'application puisse être avertie lors d'un changement d'état du GPS (activation / désactivation du GPS, changement du statut de la source, changement de position géographique). Lorsque le GPS à détecté la position, le programme entre dans la fonction `onLocationChanged` qui prend en paramètre la position géographique [annexe 5, ligne 120]. Il est alors possible dès ce moment d'afficher la carte Google Maps en utilisant une `URI` [annexe 5, ligne 68].

[20] http://www.tutomobile.fr/geolocalisation-grace-au-gps-ou-au-reseau-mobile-tutoriel-android-n%C2%B015/13/08/2010/

L'image ci-dessous représente la vue principale du programme où sont affichées toutes les informations intéressantes pour l'utilisateur. Les deux boutons "Afficher adresse" et "Afficher map" sont utilisable uniquement lorsqu'une position à été trouvée.

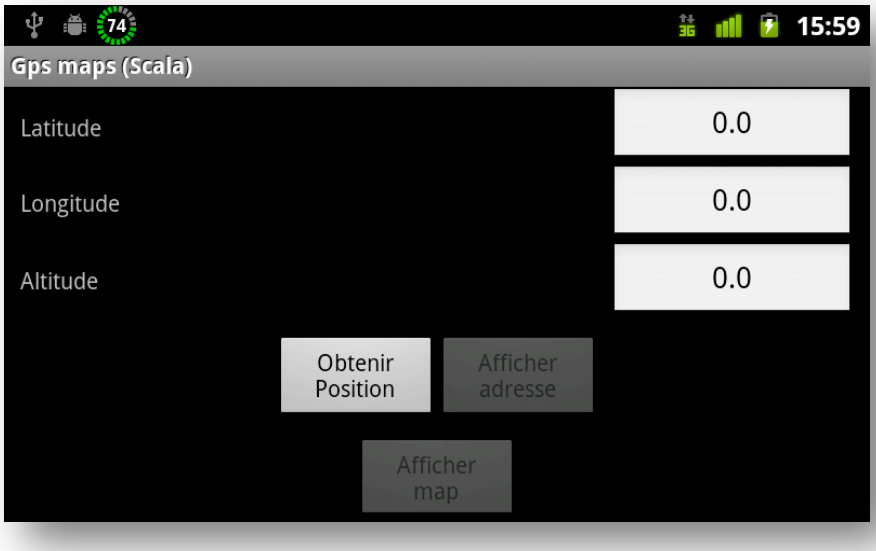


Figure 5.1 - Vue principale

Cette deuxième image est affichée sur le mobile lorsque l'utilisateur clique sur le bouton "Afficher map". Cette vue de Google Maps est démarrée grâce à une `URI`.

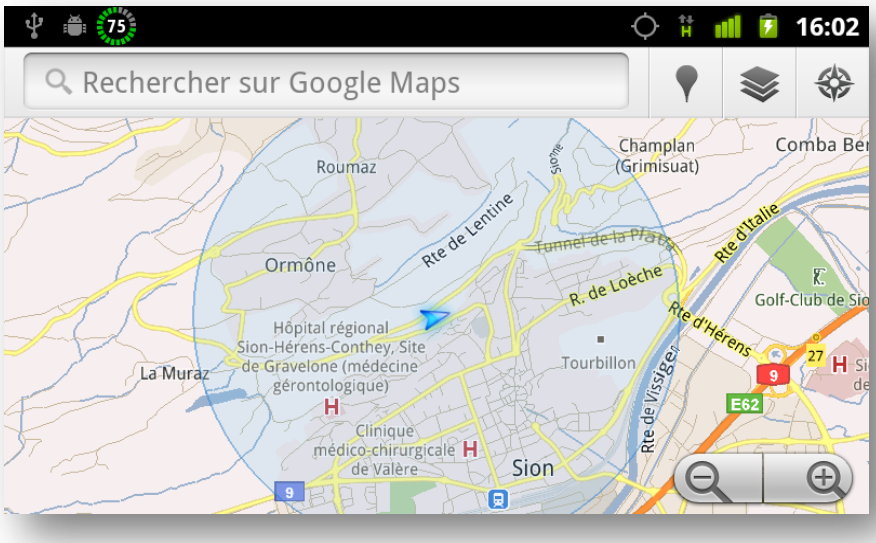


Figure 5.2 - Google Maps

7.3 Tests et mesures

Grâce à ce premier programme codé en Scala, il est possible de faire différents tests liés à la toolchain et aux différents paramètres. Pour ce test, les deux programmes utilisés sont le programme Java [annexe 4] et le programme Scala directement traduit du tutorial.

L'application possède une taille relativement petite, il n'est pas donc possible de tirer des conclusions sur cet unique test, mais il est déjà possible de se faire une idée de l'utilité de Proguard et des différents paramètres. Trois mesures ont été effectuées :

- Taille programme en Java;
- Taille programme en Scala sans optimisation mais avec rétrécissement;
- Taille programme en Scala avec optimisations maximum et rétrécissement.

Pour chaque programme, deux mesures ont été faites. La première représente la taille de l'apk, la seconde la taille de l'application une fois installé sur un appareil Android.

	taille apk [Ko]	**sur Android [Ko]**
programme Java	**8**	**16**
programme Scala sans optimization, avec rétrécissement	**16**	**52**
programme Scala avec optimisations et rétrécissment	**8**	**38**

Table 1 - optimisation Proguard sur GPS Maps

Pour différentes raisons encore indéterminées, il n'est pas possible de créer une apk sans que Proguard effectue un rétrécissement du code. En effet, le programme effectuant la transformation des fichiers *class* en *dex* retourne un message d'erreur et arrête les opérations.

7.4 Conclusion

Le programme codé en Scala a pu être directement traduit du Java. Tous les fonctions utilisées qu'ils fassent parties de l'API Android ou Google fonctionnent sans autres.

Avec ce programme, le code Scala ne fonctionne ni plus ni moins que comme du code Java. Le programmeur ne possède pas d'avantage pour une telle application d'utiliser du Scala si ce n'est que le code est plus lisible car sa syntaxe permet quelques libertés qui évitent une surcharge visuelle du code.

L'exemple ci-dessous montre un petit exemple très simple de l'utilisation du *pattern matching* de Scala face à un *switch* de Java (exemple tiré des programmes implémentés). Ce code permet de relier une fonction à un bouton de l'interface graphique.

Code Java :

```
public void onClick(View v) {
	switch (v.getId()) {
	case R.id.obtenir_position:
		obtenirPosition();
		break;
	case R.id.afficherAdresse:
		afficherAdresse();
		break;
	default:
		break;
	}
}
```

Code Scala :

```
def onClick(v: View){
	v.getId match {
		case R.id.obtenir_position => obtenirPosition()
		case R.id.afficherAdresse => afficherAdresse()
	}
}
```

Le deuxième programme apportant une fonction supplémentaire grâce aux `URI` fonctionne également sans encombre.

Aucune conclusion sur le poids de l'application installé ne peut être faite, l'application étant bien trop petite. Cependant il est possible d'observer que Proguard permet une réduction significative de la taille du code de base, qui perd dans ce cas là presque 30 % une fois l'application installée sur un terminal.

8 Utilisation des Handlers et Intents en Scala

8.1 Introduction

Android met à disposition un outil permettant de transférer des informations à travers les machines virtuelles (et donc entre différents programmes), cet outil est appelé `Intent`. Les `Intent` sont des objets dans lesquels il est possible déposer des informations ayant de l'intérêt pour le récepteur. Ils peuvent aussi bien être émis pas le système que par un programme. Lorsque cet outil est utilisé avec des `IntentFilter`, il est possible de filtrer des messages ou événements directement émis par le système.

L'API met également à disposition un gestionnaire permettant d'envoyer ou de traiter des messages associés à des threads, cet outil est appelé `Handler`. En effet, la gestion des états d'une activité sur Android ne permet pas d'effectuer certaines actions si le programme n'est pas en mode « running ». Il est donc nécessaire d'utiliser les `Handler` afin d'y déposer des actions, que le programme pourra exécuter lorsqu'il sera en fonctionnement.

Grâce à ces outils, un programme pourra être averti lorsque le système envoie un événement, comme la réception d'un SMS, d'un appel téléphonique ou même des événements liés au hardware comme la gestion de l'USB ou de la batterie.

Afin de vérifier le fonctionnement de ces mécanismes en Scala, un petit programme de test a été conçu. Ce programme a pour but de se protéger d'un éventuel vol d'appareil lorsque celui-ci est en charge. En effet, beaucoup de personnes rechargent leur appareil dans des endroits publics, tel que leur lieu de travail ou même dans le train et l'oublie lorsqu'ils sont distraits. Si une personne mal intentionnée retire le chargeur alors que le programme est enclenché, une alarme retenti.

Lorsque le programme est lancé, un mot de passe est demandé à l'utilisateur, puis, par un simple clic sur un bouton, la protection est enclenchée. À partir de ce moment, lorsque le câble de chargement est retiré sans que le mot de passe ait été correctement entré, l'appareil sonne. Afin de couper l'alarme, l'utilisateur peut entrer le mot de passe ou remettre le cordon d'alimentation.

8.2 Implémentation et fonctionnement

L'activité principale s'occupe d'initialiser l'interface graphique, d'instancier et d'initialiser les éléments de réceptions d'événement.

Une seconde classe, directement reprise depuis une source sur internet[21], `SoundManager`, est utilisée afin de générer le signal sonore. Elle implémente des méthodes pour la gestion de son (`initSound`, `playSound`, `stopSound`).

Afin de comprendre le fonctionnement de ce mécanisme, le paragraphe ci-dessous décrit les étapes nécessaires pour l'implémentation et renvoi chaque opération au programme disponible en annexe 6. Un extrait de code condensé est également exposé ci-dessous.

Fonctionnement : les actions doivent ont été déclarée dans des variables [annexe 6, lignes 15 - 16] et ont été ajoutée à l'`IntentFilter` [annexe 6, lignes 39-41]. Lorsque l'utilisateur démarre la protection, un `Handler` ainsi qu'un `BroadcastReceiver` décrivant le comportement en cas de réception des messages sont instanciés [annexe 6, lignes 72-81]. La dernière étape consiste à

[21] http://www.droidnova.com/creating-sound-effects-in-android-part-2,695.html

enregistrer l'activité, l'`IntentFilter` et le `Handler` auprès du système afin de recevoir les `Intent` [annexe 6, ligne 86].
Lorsque l'utilisateur désactive le programme, il suffit de désinscrire le `BroadcastReceiver` du système [annexe 6, ligne 93], de cette manière, le programme ne recevra plus les `Intent` émis par le système.

```
/** déclaration des événements susceptibles de nous intéresser */
private val ACTION_DIS = new String(Intent.ACTION_POWER_DISCONNECTED)
private val ACTION_CON = new String(Intent.ACTION_POWER_CONNECTED)

/** création de l'IntentFilter et ajout des actions */
intentToRecieveFilter = new IntentFilter()
intentToRecieveFilter addAction(ACTION_DIS)
intentToRecieveFilter addAction(ACTION_CON)

/** création du Handler */
mHandler = new Handler()

/** déclaration et implémentation des actions à effecuter lors de la récéption
d'événements */
mIntentReceiver = new BroadcastReceiver(){
  override def onReceive(context: Context, intent: Intent){
    intent.getAction() match {
      case ACTION_DIS => //action à effectuer lors de ACTION_DIS
      case ACTION_CON => // action à effectuer lors de ATION_CON
    }
  }
}

/** enregistrement du context auprès du système afin de recevoir les événements */
this.registerReceiver(mIntentReceiver, intentToRecieveFilter, null, mHandler)

/** désinscrit le context afin de ne plus recevoir les événements */
this.unregisterReceiver(mIntentReceiver)
```

8.3 Conclusion

Les `Intent` sont un outil très utilisé sur Android. C'est grâce à eux qu'il est possible au système et aux applications d'envoyer des messages contenant des informations à toutes les instances machines virtuelles. Nous avons démontré avec ce programme que les `Handler`, les `Intent` et `IntentFilter` fonctionnent parfaitement en Scala également. Cette étape importante a ainsi validé notre approche et prouvé que Scala était complètement interopérable avec le Java.

9 Utilisation des acteurs sur Android

9.1 Introduction

Dans cette troisième partie, nous allons discuter de l'utilisation des acteurs lorsque ceux-ci sont utilisés dans un environnement Android.

Scala possède de très nombreuses librairies dont certaines permettent de faire des fonctions particulièrement intéressantes. Parmi celle-ci, on y trouve les `Actors`[22] de Scala. Un acteur est un modèle de calcul concurrent qui encapsule les données, le code et son propre thread de contrôle. Ils communiquent de manière asynchrone en utilisant la technique du passage de messages immuables. Chaque acteur possède sa propre boîte aux lettres pour stocker les messages évitant ainsi les problèmes d'interblocage et de synchronisation. C'est en partie grâce au DSL de Scala qu'il est possible travailler élégamment avec cette librairie. Différents types d'acteurs sont utilisables, les plus connu sont les acteurs de Haller [Haller, 11] et les acteurs de Akka[23]. Pour ce projet, nous utiliseront les acteurs de Haller.

Ce paradigme de programmation concurrente, pourtant très intéressante, est à notre connaissance complètement absente des plateformes embarquées. Afin de tester son fonctionnement sur Android avec une machine virtuelle de type Dalvik, un programme simulant un `chat` utilisant les acteurs a été implémenté. Ce programme est inspiré de l'exemple mis à disposition dans le « pre-print Actors in Scala » [Haller, 11]. Ce programme va permettre de déterminer si les acteurs sont une méthode adaptée pour une plateforme Android (qui ne met pas à disposition toutes les classes de Sun par exemple) qui propose une machine virtuelle légèrement différentes de l'implémentation de référence, ou si des restrictions s'opposent au bon fonctionnement de ceux-ci.

9.2 Fonctionnement des acteurs

Chaque classe voulant utiliser les acteurs doit mixer le trait `Actor` et doit implémenter la méthode `act` correspond à la méthode `run` d'un thread. Il est également nécessaire de créer une boucle principale permettant de recevoir les messages lorsque ceux-ci arrivent dans la boîte aux lettres. Il existe différentes manières de définir ces boucles, avec `loop` ou `while` et `react` ou `receive`. De manière générale, `loop` et `while` définissent la manière dont le thread est exécuté et `react` ou `receive` la manière dont le message est attendu.

L'extrait de code ci-dessus permet d'illustrer le fonctionnement décrit ci-dessus:

```
def act{              // surcharge la méthode exécutée lors du lancement de l'acteur
  loop{              // crée la boucle principale de l'acteur
    react{           // définit la manière dont les message sont attendus
      case _ =>      // actions à effectuer lors de la réception d'un message
    }
  }
}
```

[22] http://java.dzone.com/articles/scala-threadless-concurrent
[23] http://akka.io/

La boucle `while` exécute le thread dans l'acteur lui-même alors que `loop` exécute dans un thread externe, ce qui permet d'exécuter plusieurs acteurs thread dans le même acteur.

La méthode `react` est basée sur une stratégie événement alors que la méthode `receive` est basée sur une stratégie thread. `React` n'est pas bloquant et n'occupe pas de thread tant qu'aucun message n'est reçu, il est donc possible de faire fonctionner un programme sans que celui-ci soit bloqué par la méthode de réception. La méthode `receive` est bloquante, elle occupe donc un thread même lorsque celui-ci ne fait qu'attendre un message entrant.

Une fois l'acteur instancié il est possible de lui transmettre des messages au travers de différentes méthodes. Ici, seules deux fonctions seront utilisées, l'envoi asynchrone et l'envoi avec attente de réponse. Ces deux méthodes s'utilisent comme ceci :

```
case class Message(msg : String)     // déclaration de la classe à transmettre
var actor = new MyActor              // instanciation de l'acteur
actor ! Message("bonjour")           // envoi du message à actor
rep = actor !? Message("bonjour")    // envoi du message et attente de la réponse
```

L'acteur permettant de gérer la réception peut être défini comme ceci :

```
def act() {
  while (true) {
    receive {
      case Message(msg : String) => println("recu un message: " + msg )
    }}}
}
```

9.3 Implémentation du programme de chat

Le programme est implémenté de la manière suivante : l'activité principale initialise l'interface graphique et crée une instance de la classe `ChatRoom` qui joue le rôle de serveur [annexe 7, lignes 51-84]. Chaque utilisateur ou instance de `Person` doit s'inscrire auprès de la `ChatRoom` afin de pouvoir recevoir des messages et en envoyer [annexe 7, ligne 105]. L'enregistrement des utilisateurs se fait dans une `Map` de type `User -> Actor` [annexe 7, ligne 189]. De cette manière, lorsque l'on veut contacter un utilisateur il suffit de l'appeler par son nom et son acteur nous est retourné automatiquement.

Lorsqu'un utilisateur désire envoyer un message, un objet de type `UserPost` est envoyé à la `ChatRoom` [annexe 7, ligne 138]. Cet objet contient le nom de la personne qui envoie le message et le message à poster. Une fois cette classe interceptée par la `ChatRoom`, elle va être redistribuée à toutes les `Person` inscrites dans la `Map` [annexe 7, lignes 198-199]. Chaque message reçu par `Person` est affiché sur l'interface graphique dans sa texte box correspondant [annexe 7, ligne 238].

Afin de pouvoir interagir avec les éléments de l'interface graphique depuis des classes autre que l'activité principale, il est nécessaire d'utiliser des outils. Différentes solutions sont possibles, ici il a été question d'utiliser les `Handler` (documentation disponible en annexe 3). Lorsque qu'une `Person` désire écrire sur l'interface, il envoi dans le `Handler` un objet contenant les actions à effectuer [annexe 7, lignes 212-216]. Lorsque l'activité principale sera dans l'état « running », le `Handler` pourra récupérer le message et exécuter les opérations nécessaires [annexe 7, lignes 73-77].

Les différentes classes passées par les acteurs sont :

`Subscribe(user: User)`	utilisée pour inscrire un acteur auprès de la `ChatRoom`
`User(name: String)`	classe contenant uniquement le nom de l'utilisateur
`Unsubscribe(user: User)`	utilisée pour désinscrire une `Person` de la `ChatRoom`
`Post(msg: String)`	classe contenant le message à envoyer
`UserPost(user: User, post: Post)`	utilisée pour envoyer un message à la `ChatRoom`

Le diagramme UML ci-dessous représente les différentes classes utilisées dans ce programme ainsi que les relations entre elles.

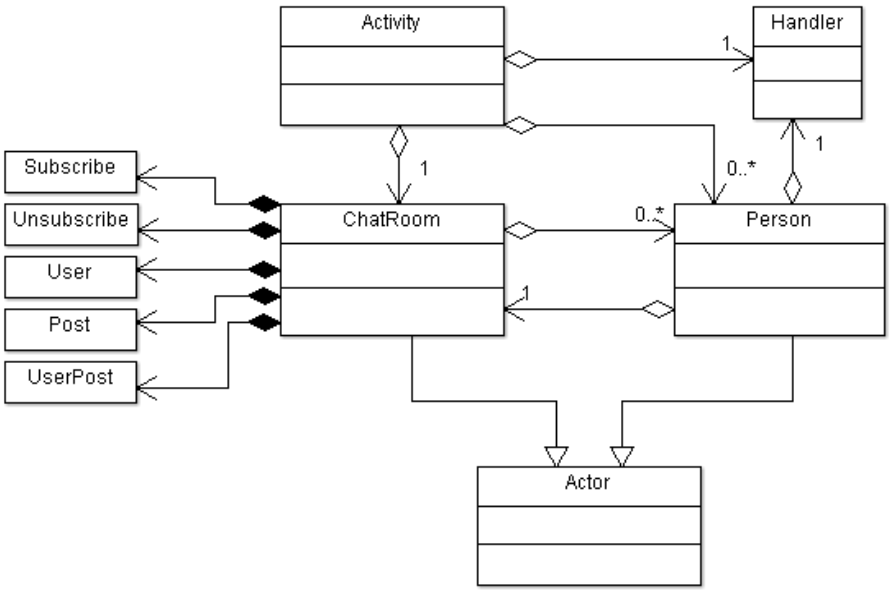


Figure 5 - diagramme UML

9.4 Illustrations et explications

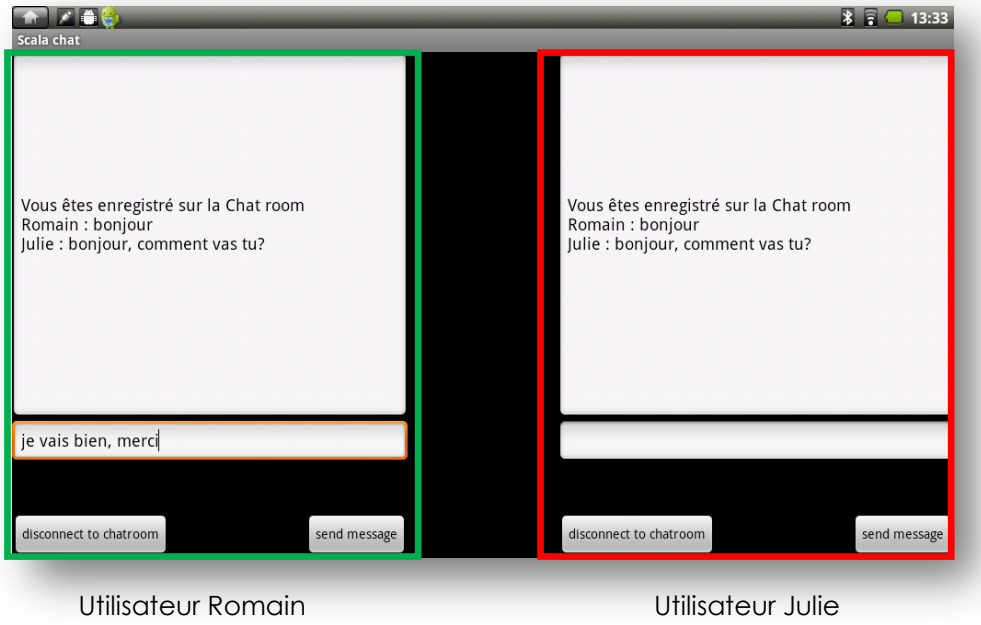


Figure 6 - Application Chat

Comme il est possible de le voir, les deux utilisateurs se partage l'écran et chacun possède sa propre interface de communication. Deux boutons suffisent pour réaliser les différentes actions, le premier est utilisé pour la connexion / déconnexion de la `ChatRoom` et le second pour envoyer le message saisi dans le champ de texte.

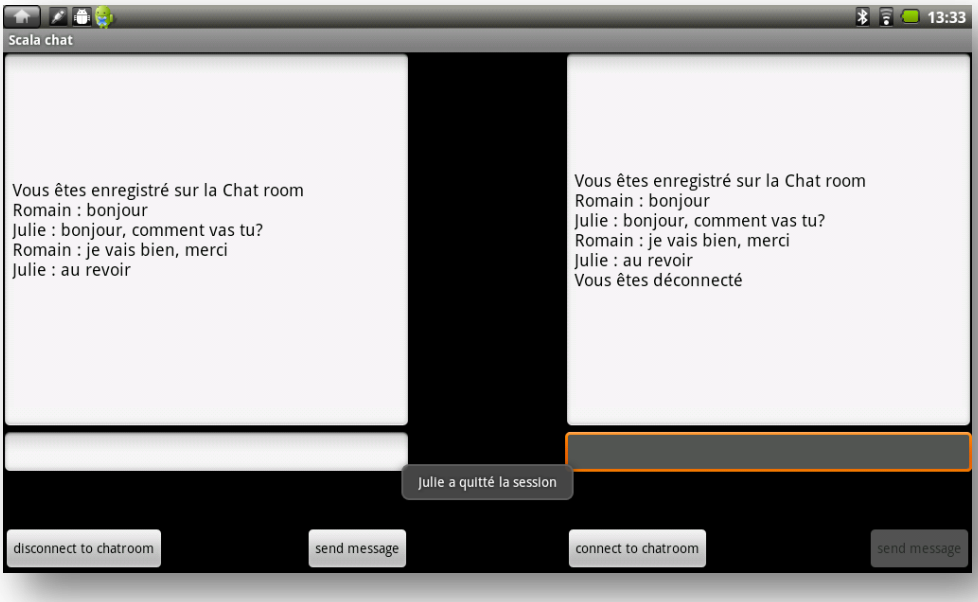


Figure 7 - Application Chat

Lorsqu'un utilisateur à quitté la session, il ne lui est plus possible d'envoyer ou de recevoir des messages.

9.5 Conclusion

Ce programme mettant en scène les acteurs de Scala est un exemple classique permettant de voir à quel point il est facile et transparent pour le programmeur de communiquer de manière efficace entre différentes classes d'un programme et d'exécuter des tâches en parallèles. Grâce à la syntaxe épurée de Scala (sucres syntaxiques, constructeur créé automatiquement, *pattern matching*, etc.…) le programme possède une structure extrêmement propre et très concentrée. L'exemple ci-dessous illustre un extrait de code Java et son équivalent en Scala.

Code Java:

```
/** déclaration de la classe message */
class Message{
  String msg;
  public Message(String msg) {
    this.msg = msg;
  }
}

/** instanciation de la class */
Message message = new Message("bonjour");
```

Code Scala:

```
/** déclaration de la classe message */
case class Message(msg : String)
/** instanciation de la class */
var message = Message("bonjour")
```

Le mot clé `case` est un sucre syntaxique, il permet non seulement de rendre la classe sérialisable mais également de se passer du mot clé `new` lors de l'instanciation de la classe. Ceci nous permet donc de rendre le code plus épuré comme le montre l'exemple ci-dessous:

```
/** classe Message sans sucre syntaxique */
monActeur ! new Message("bonjour")

/** classe Message avec sucre syntaxique */
monActeur ! Message("bonjour")
```

L'utilisation des acteurs sur un OS Android ne pose pas de problèmes, aucune adaptation n'est nécessaire et fonctionne de manière native. Tout ce qui est fait dans ce programme Scala pourrait être fait en Java, mais sa complexité et son nombre de lignes de code serait bien plus grand que les 210 lignes (commentaires inclus) du programme en Scala implémenté ici.

10 Acteurs remote sur Android

10.1 Introduction

Cette dernière partie aborde l'utilisation des acteurs *remote* lorsque ceux-ci sont utilisés dans un environnement Android.

Comme vu précédemment, les acteurs de Scala sont un modèle de concurrence très intéressant, permettant de faire des programmes complexes en très peu de lignes et relativement simple à mettre en œuvre grâce à un niveau d'abstraction élevé. Les acteurs *remote* apportent les mêmes fonctionnalités que les acteurs mais ils sont en plus capable de communiquer à distance au travers d'un réseau en utilisant le protocole TCP/IP, ce qui nous permettra d'exécuter du code distribué sur des différentes machines.

10.2 Fonctionnement

Tout d'abord, tous les objets passant d'un acteur à un autre se doivent d'être sérialisables, sans quoi les messages ne pourront être transmis. Ceci peut se faire de deux manières différentes en Scala, soit en mixant le trait `Serializable` ou avec l'annotation `@serializable`.

Lors de l'implémentation d'un acteur *remote* il est possible de distinguer deux parties :

- L'acteur qui attend un message afin de pouvoir exécuter une action (classe `Actor`);
- L'acteur qui envoie un message à un autre acteur (classe `AbstractActor`).

10.2.1 Classe Actor

L'acteur qui attend un message est une classe mixant un trait de type `Actor` dans laquelle la méthode `act` est surchargée afin d'exécuter les actions souhaitées. Il est également nécessaire de spécifier à cet acteur différents paramètres afin de le rendre accessible depuis un réseau.

Ces paramètres sont :

- Le port sur lequel vit l'acteur;
- Le nom de l'acteur (type `Symbol`, qui est un `String` précédé d'une apostrophe);
- L'acteur à exécuter (en général lui-même, `self`).

```
def act {                              // méthode exécutée lors d'un start
    alive(9000)                        // vit sur le port 9000
    register('theActor, self)          // se nomme 'theActor
    loop {
        react {
            // affiche bonjour lors de la réception d'une classe "MyClass"
            case MyClass => println("bonjour")
        }
    }
}
```

10.2.2 Classe AbstractActor

Un `AbstractActor` est une classe contenant les méthodes nécessaires pour atteindre un acteur au travers d'un réseau. Lors de sa création il est nécessaire de lui spécifier les paramètres d'accès à un acteur. Ces 3 paramètres sont les mêmes que ceux utilisés lors de la création d'un `Actor`.

L'instanciation d'un `AbstratActor` se fait comme ceci :

```
var monActeur : AbstractActor = select(Node(ip, Symbol), port)
```

Cette classe comporte plusieurs méthodes afin d'envoyer des messages, ici il sera question de décrire les plus couramment utilisées lors de l'implémentation du programme de test. Il s'agit de :

```
case class Message(msg : String)    // déclaration de la classe Message

monActeur ! Message("salut")        // envoi Message de manière asynchrone

monActeur !? Message("salut")       // envoi Message et retourne la réponse
```

11 Implémentation du programme de test

11.1 Introduction

Un programme a été implémenté afin de démontrer que les acteurs *remote* sont possibles et viables sur Android. Le but de ce programme est de pouvoir utiliser les acteurs *remote* afin d'exécuter des fonctions ou méthodes à distance sur d'autres appareils équipés d'Android. Nous verrons que le Framework mit en place dans cette partie nous permettra par exemple de faire du calcul parallèle distribué, un vrai système de chat ou encore un *sensor network* pour mesurer la consommation électrique d'appareils. Pour ceci il est nécessaire de créer un protocole permettant à tout appareil connecté sur le même réseau Wifi de partager son identité ainsi que les acteurs qu'il met à disposition sur le réseau (que je nommerais aService dans la suite de ce document). Tous les appareils qui se connectent sur un réseau Wifi doivent pouvoir s'intégrer dans le réseau de nœuds créé par les appareils présents auparavant, et ainsi tout connaître sur les utilisateurs déjà présents. Le fonctionnement du protocole permettant cela est décrit ci-dessous.

11.2 Description du protocole

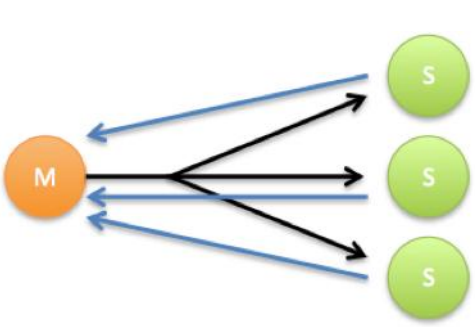
Chaque appareil est un nœud qui peut posséder deux états différents : master ou slave. Chaque appareil entrant dans le réseau écoute sur un de ses ports si un message broadcast UDP est émit par un autre appareil. Si rien n'est reçu après un temps aléatoire t (`RandomTimer`), l'appareil émet un message broadcast et devient un master node. Ce message broadcast contient l'adresse IP du master node, chaque appareil recevant ce message est donc en mesure de lui répondre, au travers d'un `AbstractActor`, avec un message de présence (classe de type `RepetedIdentifiy`) ou de lui transmettre toutes les informations nécessaires, si celui-ci n'est pas connu (classe de type `IdentifyMessage`). Lorsque le master node connaît tout le réseau, il renvoi, si nécessaire, les informations à tous les slaves. De cette manière, tous les nœuds se connaissent.

Lorsqu'un slave node sort du réseau, le master node ne recevra plus les messages de présence et le sortira de la liste des nœuds présents après deux tentatives. Ces deux tentatives sont nécessaires car les messages broadcast sont transmis en UDP et la réception ne peut donc pas être garantie. Chaque slave node réinitialise leur `RandomTimer` avec un temps aléatoire t lorsqu'un message broadcast est reçu. Lorsque le timer est à zéro (pas de message broadcast reçu, ce qui signifie plus de master), le slave node émet un message broadcast et devient ainsi le master node.

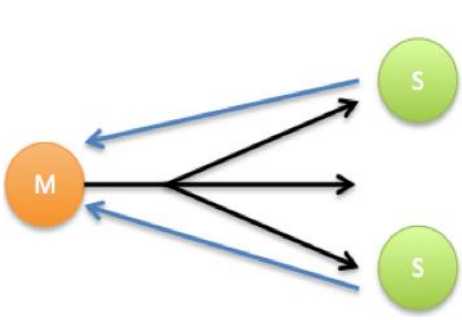
Si le nœud master disparaît, il est possible que deux nœuds slave tentent d'envoyer un message broadcast en même temps. Dans ce cas, les deux nœuds reviendront en mode slave, c'est alors le premier slave node ayant son `RandomTimer` à zéro qui deviendra master.

Les temps ont été déterminés comme ceci : le master envoie un broadcast toutes les 3 secondes, les slave ont un timer initialisé avec des valeurs aléatoire entre 7 et 10 secondes. Ces valeurs permettent de prédire le meilleur et le pire des cas lors de la disparition d'un master node. Dans le meilleurs des cas, un slave deviendra master après seulement 4 secondes. Dans le pire des cas, c'est après 10 secondes que le réseau aura de nouveau un master.

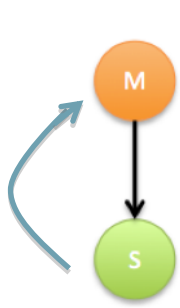
11.3 Illustration du protocole



Le master envoi un broadcast UDP (flèche noir), les slaves initialisent leur random timer et quittance leur présence au travers d'un `AbstractActor` (flèche bleu)



Un slave est sorti du réseau. Le master envoi un broadcast, les slaves initialisent leur random timer et quittancent leur présence au travers d'un `AbstractActor`, le nœud qui n'a pas quittancé sera sorti du réseau après deux non quittances consécutives.



Le master est sorti du réseau, le premier slave node ayant son timer à 0 passe en mode master et envoi un message broadcast

11.4 Diagramme de classe du protocole

La figure ci-dessous représente les quatre classes utilisées pour l'implémentation du protocole. Toutes ces classes sont contenues dans le même paquetage «protocol ». La classe `BroadcastReceptor` [annexe 8.2, p.4] utilise des éléments de `ReloadTimer` [annexe 8.2 p.5] (pour réinitialiser le `randomTimer` lorsqu'un message broadcast arrive) et de `BroadcastEmitter` [annexe 8.2, p.4] (afin de stopper l'envoi de message Broadcast). La classe `ReloadTimer` utilise la classe `BroadcastEmitter` pour redémarrer l'émission de message Broadcast si aucun master node ne se manifeste.

La communication entre le service Android qui gère tout les `Actors` et le protocole se fait par un `AbstractActor` et ceci afin de découpler au maximum le lien entre les classes représentant les couches du programme.

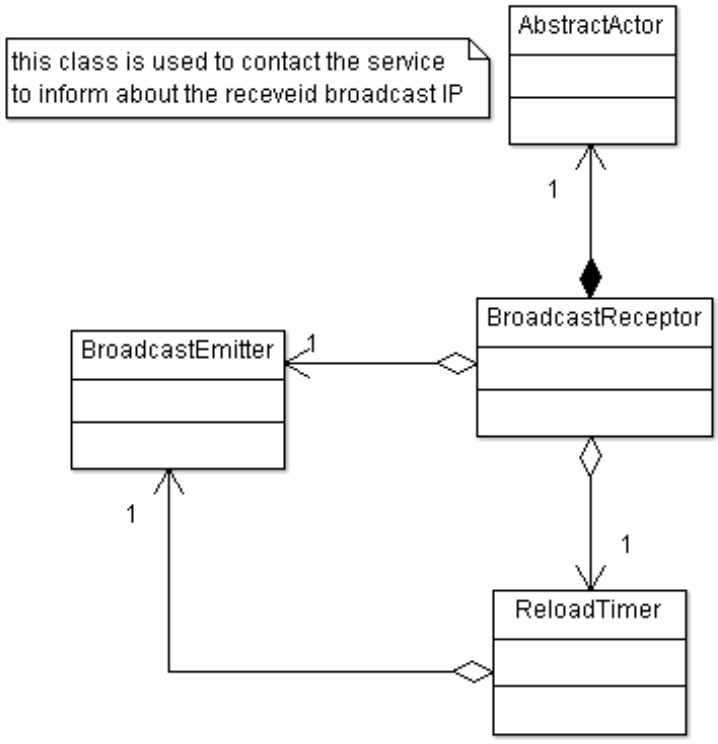


Figure 8 - diagramme de classe du protocole

11.5 Mesures

Différentes mesures ont été effectuées grâce à Wireshark[24] afin de déterminer la charge du réseau lorsque le protocole fonctionne. Lorsqu'un appareil s'identifie au moyen d'un `IdentifyMessage`, il transmet 1614 bytes, 1415 bytes pour l'`IdentifyMessage` et 199 bytes pour la connexion TCP. Lorsqu'un appareil est déjà connu sur le réseau, son message indiquant sa présence pèse 811 bytes, 612 bytes pour le `RepetedIdentify` et 199 bytes pour la connexion TCP.

Il est possible d'observer que lorsqu'un nœud n'est pas connu, il transmet deux fois plus de données sur le réseau qu'un nœud connu. Au moment où le réseau est stable et que tous les nœuds ne changent pas d'état, la quantité de données transférée est faible.

11.6 Fonctionnement et implémentation du programme

Le protocole décrit ci-dessus va être utilisé dans un programme permettant d'exploiter des ressources des terminaux présents sur le réseau. Grâce à ce protocole il est donc possible de :

- Visualiser les utilisateurs mettant à disposition des acteurs sur un réseau;
- D'activer ou de désactiver des aServices;
- D'utiliser les ressources des terminaux présents sur le réseau;
- Tous les nœuds connaissent tous les nœuds.

Toutes les informations à l'intérieur d'un nœud concernant le réseau et son architecture sont stockées dans la classe `NetworkNode` [annexe 8.2, p.7] dans une `Map` de type `String -> User` (un nom d'utilisateur faisant correspondre un `User`). La classe `User` contient toutes les informations d'un nœud. Elle contient entre autre deux `Map`, l'une contenant tous les aServices implémentés sur le terminal (`allService`) et l'autre les aServices en fonction (`service`). Les `Map` ont comme clé un `String` contenant le nom de l'aService et retourne soit leur état (activé ou désactivé) soit leur `AbstractActor` permettant de les contacter.

```
Object NetworkNode {
  var friends = Map(String, User)          /** Contient tous les nœuds amis */
…
}
class User(actor : AbstractActor){
  /** contient tous les acteurs implémentés et retourne leur disponibilité */
  var allServices = new HashMap[String, Boolean]
  /** contient tous les acteurs activés sur mon terminal */
  var service = new HashMap[String, AbstractActor]
…
}
```

[24] http://www.wireshark.org/

11.7 Diagramme de classes du programme

Le diagramme UML ci-dessous représente la hiérarchie des différentes classes implémentées dans le programme. La « gestion des onglets » ainsi que les « vues du programme » sont des classes contenues dans le paquetage « view ».

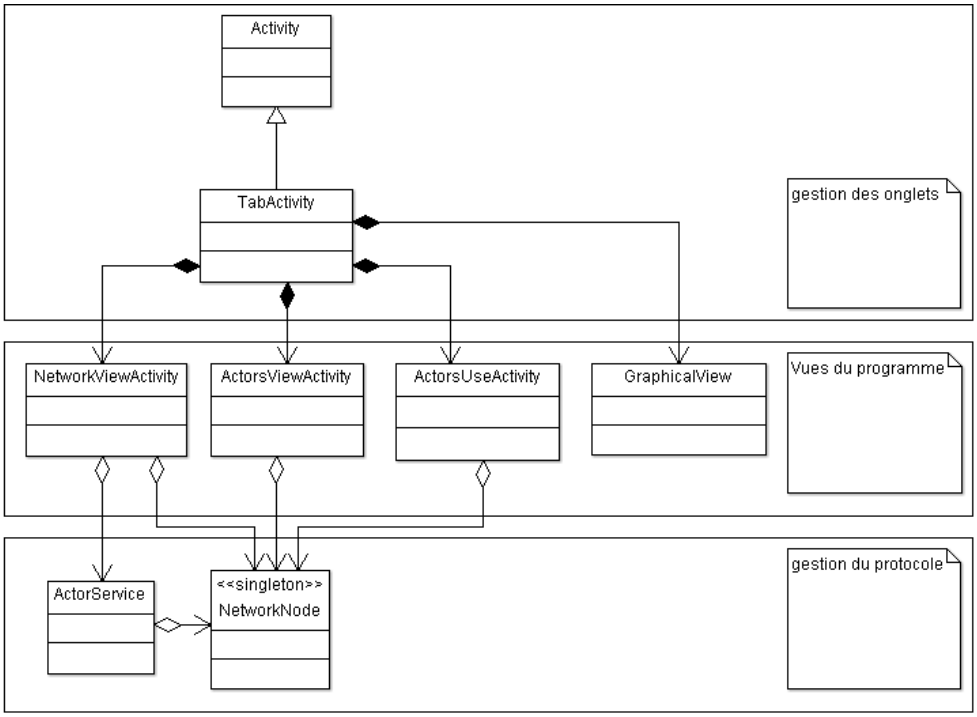


Figure 9 - diagramme UML du programme remote actors

Lors du lancement du programme, la classe `TabActivity` [annexe 8.3, p.1] initialise les différents onglets, charge toutes les activités représentant les différentes vues du programme et démarre l'activité `NetworkView`. La classe `NetworkView` [annexe 8.3, p.2] démarre, grâce à une `Intent`, le `Service ActorService` [annexe 8.2, p.1] qui permet de gérer le protocole.

La classe `NetworkNode` est une classe qui doit être accessible depuis différents endroits, car c'est elle qui contient les différentes spécifications des nœuds dans le réseau ainsi que toutes les informations concernant les acteurs. Elle est de type `Object` afin d'éviter tout conflit si plusieurs classes tentent d'y accéder en même temps. Le type `Object` en Scala permet de créer une classe statique de type singleton.

11.8 Interface graphique

L'interface graphique, disponible sur chaque nœud du réseau est répartie sur quatre onglets afin de séparer au maximum les fonctions et ainsi avoir une interface ergonomique.

11.8.1 Onglet « Network View »

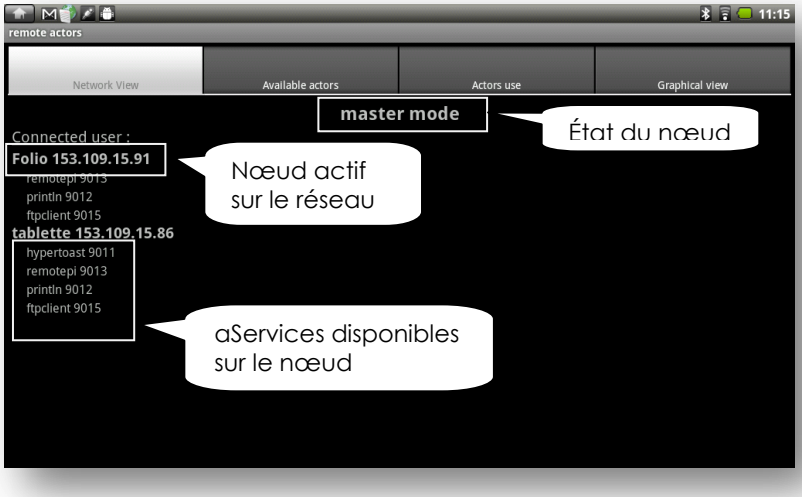


Figure 10 - Onglet Network view

Cette onglet donne une vue globale sur le réseau de nœuds formé par les terminaux. Chaque nœud présent affiche son nom de réseau et son adresse IP. Sous chaque nœud est listé l'ensemble des aServices actifs disponibles sur ces appareils.

11.8.2 Onglet « Available actors »

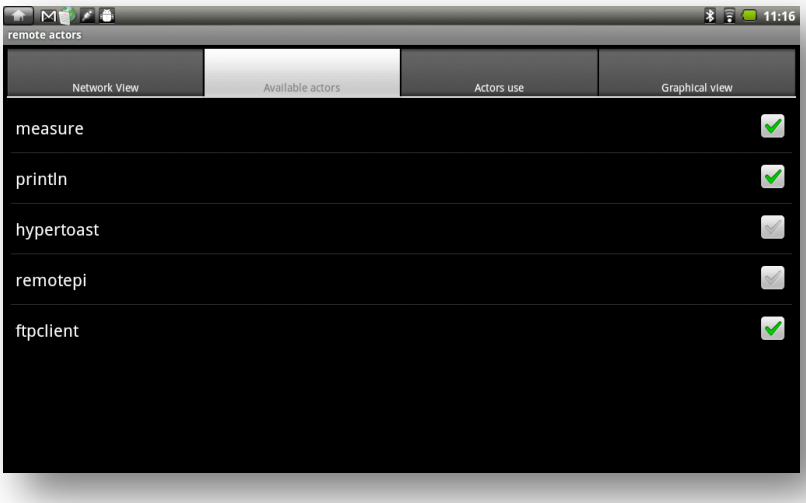


Figure 11 - Onglet "Available actors"

Cette vue permet de voir les différents aServices locaux disponibles sur le terminal afin de les activer / désactiver. Lorsqu'un acteur est activé, le master node est averti puis une confirmation est renvoyée. C'est uniquement à ce moment là que réseau de nœud est informé de l'activation d'un nouvel aService.

11.8.3 Onglet « Actors use »

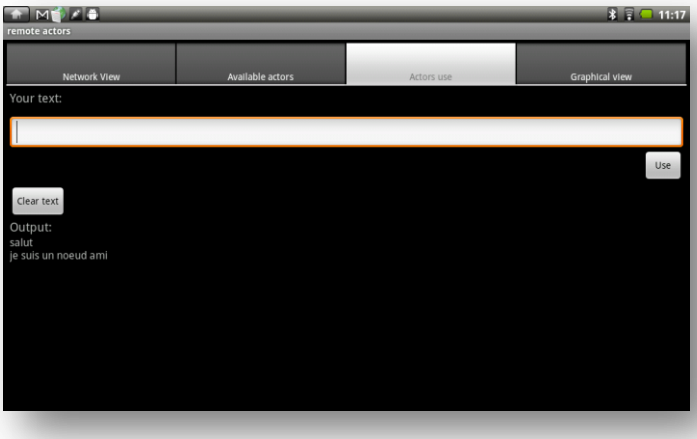


Figure 12.1 - Onglet "Actor use"

Cette vue met à disposition les éléments nécessaires pour utiliser les acteurs. Le champ de texte permet de spécifier les paramètres lors de l'utilisation des acteurs. Un `TextView` permet d'afficher les informations retournées. Le bouton « use » ouvre le menu visible ci-dessous.

Le menu affiche une liste tous les utilisateurs disponibles sur le réseau. Lorsqu'un appareil est sélectionné, un menu affiche tous les acteurs disponibles sur l'appareil.

Figure 12.2 - Onglet "Actor use", select target.

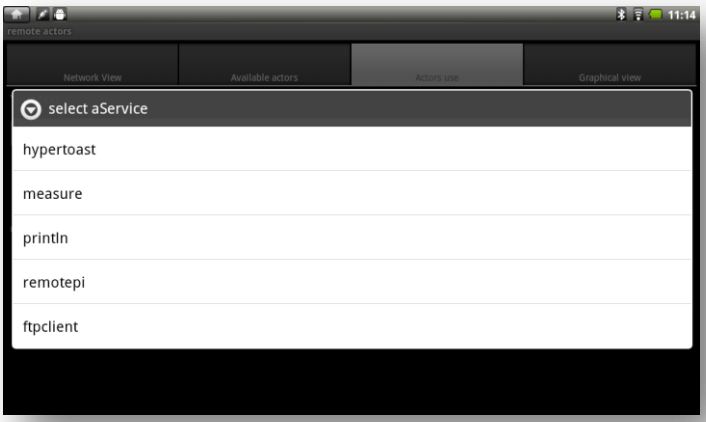


Figure 12.3 - Onglet "Actor use", select aService

Ce menu affiche tous les acteurs qu'un utilisateur met à disposition sur le réseau. Lors de leur sélection, une requête est envoyée afin de démarrer l'acteur.

11.8.4 Onglet « Graphical View »

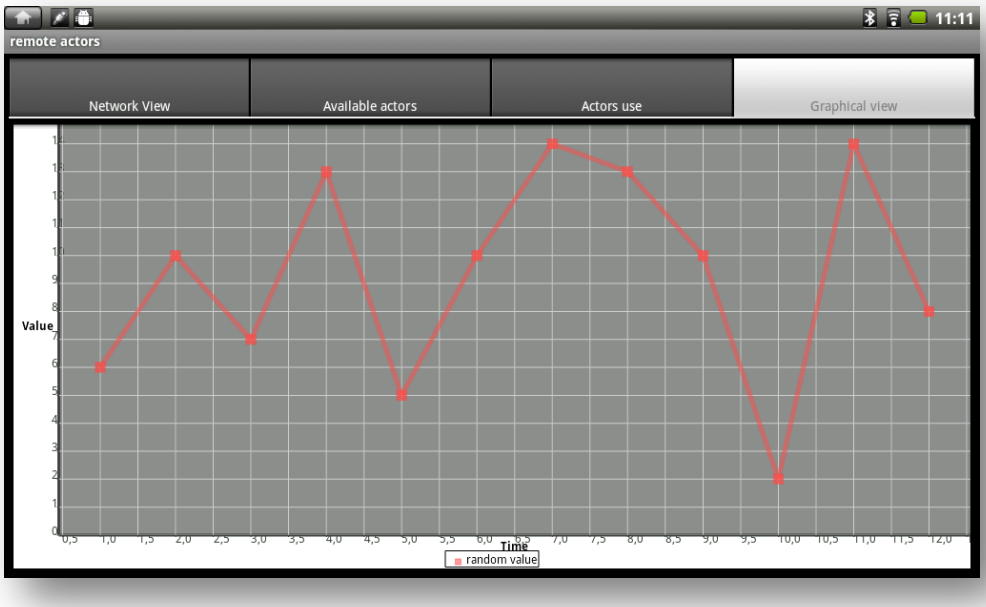


Figure 133 - Graphique DroidChart

Cet onglet permet de visualiser de manière graphique les données transmises par l'aService `Measure`. Ce graphique est généré à partir de l'outil DroidCharts[25], qui se base sur le moteur JfreeChart[26]. Grâce à cet outil, il est possible de mettre à jour sur le graphique les différentes valeurs en temps réel. Afin de garder une bonne visualisation du graphique, seul les vingt derniers points sont affichés.

[25] http://code.google.com/p/droidchart/
[26] http://www.jfree.org/jfreechart/

11.9 Implémentation de « SuperActor »

Sur un système Android, la classe mixant ce trait doit impérativement être une `Activity` ou un `Service` sans quoi les messages transmis entre acteurs se perdent.

Une super classe a été créée afin d'implémenter tous les aServices utilisables sur le réseau. Ceci permet de factoriser le code car chaque aService dans ce programme possède une seule et même structure, seule sa manière de gérer les messages est différente. Cette super classe prend en paramètre tous les éléments nécessaires à l'exécution d'un acteur *remote* [annexe 8, p.1]. Elle va permettre de démarrer l'aService, mais également d'enregistrer celui-ci auprès des acteurs actifs afin de pouvoir l'exploiter au travers du réseau [annexe 8, ligne 18]. Aucune instance de ce type n'existe, ils sont uniquement joignables depuis le réseau. Ainsi, pour les stopper il suffit de leur envoyer un message de type `Exit` [annexe 8.1, p. 1, ligne 29].

Un extrait de la classe `SuperActor` est visible ci-dessous.

```
abstract class SuperActor(name: String, port: Int, symbol: Symbol) extends Actor {

  def startActor {
    /** démarre le thread de l'acteur */
    start
    /** enregistre l'acteur dans la liste des acteurs activés */
    NetworkNode.me.enableService(name, select(Node(NetworkNode.myIp,port),symbol))
  }

  def act {
    alive(port)
    register(symbol, self)
    loop {
      react {
        /** retire l'acteur de la liste des acteurs activé et stope le thread */
        case Exit = > NetworkNode.me.removeService(name) ; exit
        /** envoi tout type de classe autre que Exit à la sous-classe */
        case any : Any = > theAction(any)
      }
    }
  }
  /** action implémentée dans la sous classe, appelée grâce au polymorphisme */
  def theAction(any: Any)
}
```

11.10 Implémentation des aServices

Différents types d'aService ont été implémentés. Ils dérivent tous de la super classe `SuperActor`. Leurs fonctionnements ainsi que leurs implémentations sont décrits ci-dessous dans l'ordre dans lequel ils ont été créés, du plus simple au plus complexe. Toutes les sous classes implémentant un aService possèdent le même prototype.

```
class aService(name : String, port: Int, symbol : Symbol, disp : (AnyRef) => Unit)
      extends(name, port, symbol disp)
```

Lors de leur instanciation, tous les paramètres nécessaires au fonctionnement sont dans le constructeur. Le `nom`, le `port` ainsi que le `symbol` sont trois éléments directement lié aux acteurs, le dernier paramètre `disp` est une fonction permettant de spécifier à l'acteur la fonction à utiliser pour afficher les résultats. De cette manière les acteurs sont indépendants du système sur lequel ils fonctionnent. Chaque aService implémente les deux faces d'un acteur: la partie qui envoie les données aux nœuds et la partie qui reçoit les données. Ainsi, chaque nœud désirant utiliser un aService doit avoir son propre aService correspondant activé. Toutes les classes utilisées pour le transfert d'informations entre acteurs sont déclarées dans les sous-classes d'acteur.

11.10.1 aService Println

Cet aService [annexe 8.1, p.2] permet de recevoir tout type de classe et de l'afficher sous forme de `String`. Sur Android, la fonction d'affichage passée en paramètre dans `SuperActor` est un `Handler`, permettant ainsi d'afficher le texte sur un `TextView` dans l'onglet « ActorUse ». Les `Handler` sont capables de recevoir tout type de classe, afin d'interagir avec l'interface graphique. Pour afficher un texte dans le `TextView` il suffit d'envoyer un objet de type `String` [annexe 8.3, p. 5, lignes 54-58]

11.10.2 aService Hypertoast

Cet aService [annexe 8.1, p.2] n'est valide que sur Android car il utilise un widget disponible uniquement sur ce système, les `Toast`. Les `Toast` sont des petits messages s'affichant en premier plan sous forme de pop-up et permettent ainsi d'informer l'utilisateur de différents événements. Cet acteur sera utilisé afin d'avertir de certains changements d'état du programme. Tout comme l'acteur `Println`, un `Handler` est utilisé afin de communiquer avec l'interface graphique. Le message passé au `Handler` est une classe de type `ToastMessage`. Lorsque le `Handler` reçoit un `ToastMessage` contenant un `String`, l'interface graphique affiche un `Toast` [annexe 8.3, p. 5, lignes 54-58].

La classe `ToastMessage` est définie ci-dessous.

```
case class ToastMessage(msg : String)
```

11.10.3 aService RemotePI

L'aService `RemotePI` permet de calculer π de manière distribuée sur un réseau. L'algorithme de PI utilisé est une somme d'approximation tendant vers π/4. De cette manière, il est aisé de séparer la somme en portions, ce qui permet de distribuer les opérations mathématiques sur plusieurs appareils.

Différentes classes permettent de faire fonctionner cet acteur. La classe `Computable` [annexe 8.1, p.3] est une classe générique permettant de calculer n'importe quel algorithme. Elle contient entre autre un prototype de fonction nommé `calc` prenant deux `Int` (représentant les bornes supérieur et inférieur de l'intervalle à calculer) et retournant un `Double`. La fonction `PiComputer` [annexe 8.1, p.3] dérive de la classe `Computable` et surcharge ainsi la méthode de calcul.

L'objet `StartRequest` permet de donner l'ordre à un acteur de préparer les portions de somme qu'il pourra distribuer à tous les acteurs `RemotePI` actifs du réseau. Cette portion de somme est définie entre deux bornes (`down` et `up`) de la classe Computable et est transférée à un autre acteur, avec l'algorithme, le tout contenu dans un `PiPacket`. Lorsque l'opération mathématique est effectuée, la réponse est retournée au travers d'un `PiResponse` à l'acteur qui à soumis l'opération.

L'extrait de code ci-dessous représente la classe générique permettant de faire des opérations mathématiques (`Computable`) ainsi que la sous-classe définissant l'algorithme de calcul pour π.

```
case object StartRequest
case class PiPacket(a: Computable)
case class PiResponse(response : Double)

@serializable
abstract class Computable(val down: Int, val up: Int) {
      def calc(a: Int, b: Int) : Double
}

@serializable
class PiComputer(down: Int, up:Int) extends Computable(down, up) {
      override def calc(down : Int, up : Int) : Double = {
              var temp : Double = 0
              for(k <- down to up){
                      temp += math.pow(-1,k)/(2*k+1)
              }
              return 4*temp
      }
}
```

11.10.4 aService Measure

L'aService `Measure` [annexe 8.1, p.4] met à disposition des valeurs mesurée sur un terminal. Lorsqu'un appareil s'inscrit auprès de cet acteur, il reçoit toutes les secondes la valeur instantanée d'une mesure. Toutes ces valeurs reçues sont affichées dans un graphique en temps réel. Deux types de messages sont envoyés entre des acteurs `Measure`, des `Token` et des `Values`. Leur prototype est décrit ci-dessous.

```
case class Token(name : String, actor : AbstractActor)
case class Value(value : Float)
```

Lorsqu'un terminal souhaite s'inscrire auprès de l'aService, il doit envoyer un `Token` contenant son nom ainsi que son `AbstractActor` par lequel il est possible de le contacter. Pour se désinscrire, la procédure est la même. Ces `Token` permettent à l'aService qui envoie les valeurs de stocker dans une `HashMap` tous les terminaux désirant recevoir des mesures. Ainsi, à chaque seconde, l'aService n'a cas envoyer à toute la `HashMap` une classe `Value` contenant la valeur mesurée [annexe 8.1, p. 4, lignes 56-68].

Toutes les valeurs reçues par les terminaux sont directement affichées en temps réel dans un graphique de l'onglet « Graphical View ».

11.10.5 aService FtpClient

Cet aService [annexe 8.1, p.5] est utilisé pour faire des relevés de mesures sur un compteur électrique. Toutes les valeurs mesurées sur le compteur sont stockées dans un fichier XML qui est accessible par FTP. Cet aService implémente deux classes permettant la communication :

```
case class GetValue(actor : AbstractActor)
case class FtpValue(value : HashMap[String, Float])
```

Lorsque l'acteur reçoit une classe `GetValue` contenant un `AbstractActor`, il se connecte au compteur électrique en FTP et récupère le fichier XML contenant les valeurs mesurées. Scala possède le support directe des balise XML, il est donc très simple de stocker ce fichier dans une variable afin de la parser par la suite.

Code permettant stocker dans une variable un fichier XML stocké sur un serveur:

```
var mesuredValue = xml.XML.load("ftp://"+ login +":"+ pass +"@"+ host +"/file")
```

Le fichier XML contenu dans `measuredValue` retourné par le compteur électrique se présente sous la forme suivante :

```
<Device DeviceID="39296907" P2LPCIdent="48065696">
<Results>
 <Register Ident="1.8.3" Value="31.028" DateTime="2011-04-01T21:51:31Z" Status="0"/>
 <Register Ident="1.8.0" Value="30.858" DateTime="2011-04-01T21:51:31Z" Status="0"/>
 <Register Ident="2.8.0" Value="00.000" DateTime="2011-04-01T21:51:31Z" Status="0"/>
</Results>
</Device>
```

Les valeurs qui nous intéressent sont stockées dans le registre `Ident` « 1.8.0 » qui correspond à la consommation totale. Nous souhaitons récupérer la `Value` ainsi que `DateTime` et les insérer dans une `HashMap`, la clé représentant le timestamp.

```
var valueMap = new HashMap[String, Float]

/** insère tous les DateTime et Value avec Ident = 1.8.0 dans valueMap */

for(reg <- (mesuredValue \\ "Register");if((reg \ "@Ident").toString == "1.8.0")){
  valueMap += (reg \ "@DateTime").toString -> (reg \ "@Value").toString.toFloat
}
```

La date se présente sous la forme : 2011-04-01T21:51:31Z. Il est encore nécessaire de la mettre en forme afin de l'afficher correctement. Ceci peut être fait avec une expression régulière définie comme ceci:

```
val timestamp = """([0-9-]+)T([0-9:]+)Z""".r
```

Cette expression nous permet de sortir dans deux variables différentes la date ainsi que l'heure. Afin d'afficher correctement ces données sur l'interface utilisateur, nous devons parcourir toute la `map` contenant les valeurs à l'aide d'une boucle `for`.

```
for((key : String, value : Float) <- valueMap){
  var timestamp(date, hour) = key
  display("date : " + date + " hour : " + hour + " value = " + value + "kWh" )
}
```

La méthode display va permettre d'afficher ces valeurs sur la sortie visible sur l'onglet *ActorUse*.

11.11 Tests et mesures

11.11.1 Test de performance

Un test permettant de déterminer la pertinence du « remote computing » a été effectué sur quatre appareils différents équipé d'Android. Les tests ont été effectués à l'aide de l'aService `RemotePI`. En effet, cet acteur utilise les ressources du processeur de manière intensive et a permis de déterminer la limite du système de calcul à distance.

Le test à tout d'abord été effectué avec un terminal. Puis l'opération a été répétée avec deux, trois et pour finir avec quatre terminaux. Le matériel à disposition ne nous permet pas de faire des tests dans d'excellentes conditions. En effet, nous disposons de deux tablettes Folio100, d'un Samsung Galaxy S et d'un Samsung Galaxy S II, qui sont quatre appareils différents, notre système n'est donc pas homogène. Ce test à permis de tracer un graphique représentant le temps de calcul de la constante avec 36'000'000 approximations en fonction du nombre d'appareils fonctionnant sur le réseau.

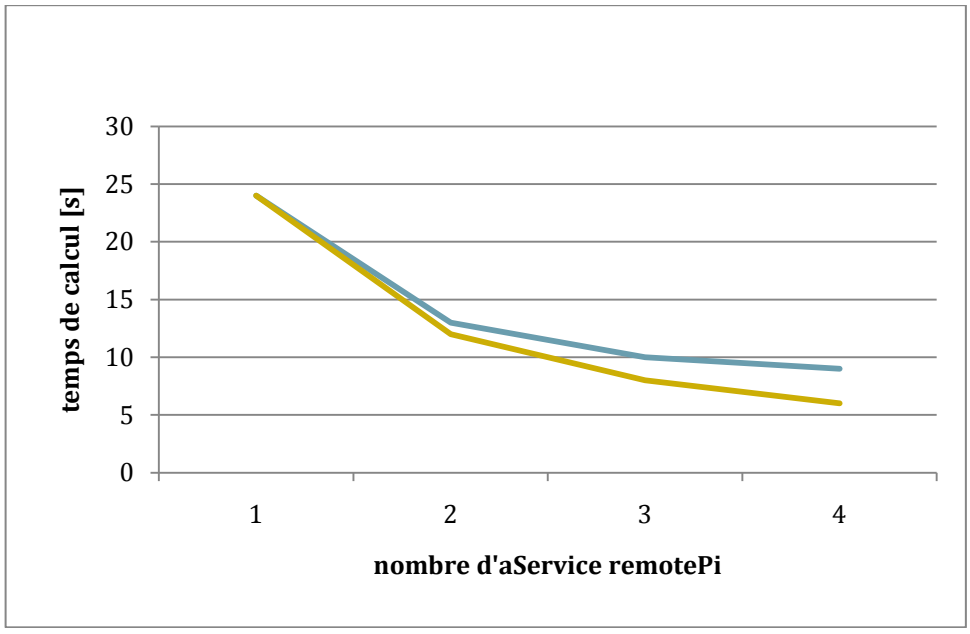


Figure 144 - relation temps de calcul / nombre d'aService, programme Android

Bien qu'il ne soit pas possible de tirer des conclusions définitives faute d'un nombre suffisant de terminaux, des observations peuvent être faites. La courbe jaune représente le temps de calcul idéal en négligeant les temps de propagation des informations au travers des acteurs et la courbe bleu les temps mesurés.

Cette fonction permet de voir que le temps de propagation des classes au travers des acteurs n'est pas négligeable. Plus le nombre de terminaux augmente, plus la réponse met de temps à être retournée au travers du réseau. Ceci peut être dû au fait que toutes les classes contenant la valeur de retour sont retournées en même temps au même terminal via Wifi, ou simplement du à l'*overhead* de la communication.

11.11.2 Test de l'optimisation avec Proguard

Avec ce programme utilisant en grande partie les librairies Scala, il est intéressant d'observer l'utilité de Proguard ainsi que les optimisations apportées au programme final. Trois tests ont été effectués :

- Sans optimisation ni rétrécissement;
- Sans optimisation, avec rétrécissement;
- Avec optimisation maximum et rétrécissement.

Le premier test n'a pas peu être effectué pour différentes raisons. En effet l'application transformant les *jar* en *dex* retourne une erreur. Il n'est pas donc possible de déterminer le poids de l'application sans rétrécissement.

Pour chaque programme, deux mesures ont été faites. La première représente la taille de l'apk, la seconde la taille de l'application une fois installé sur un appareil Android.

	taille apk [Ko]	**sur Android [Ko]**
sans optimization, avec rétrécissement	1300	1580
avec optimisations et rétrécissment	762	1110

Table 2 - optimisations Proguard sur « Remote Actors »

Il est possible d'observer que Proguard optimise de manière considérable la taille des applications.

11.12 Problèmes rencontrés

Sur Android, pas n'importe quelle classe n'est apte à exécuter des acteurs *remote*. Seuls les `Activity` et les `Service` peuvent le faire. Les autres classes retournent, à travers l'acteur, un message d'erreur informant que le message transmit s'est perdu. La raison de ceci n'as pas encore été déterminée, bien que quelques pistes nous mène à croire que les cycles de vie des programmes sur Android, optimisés pour minimiser la consommation, en seraient responsable.

Certains terminaux Android ne supportent pas la manière dont ce protocole fonctionne. Après quelques recherches, il s'avère que certains mobiles de la marque HTC ne peuvent pas recevoir de messages multicast. La ROM Wireless serait cause de ce bogue[27].

11.13 Conclusion

Le protocole implémenté permet aux différents terminaux sur le réseau de partager leurs ressources au travers des acteurs *remote*. Son implémentation lui donne une stabilité à toute épreuve pour autant qu'il ait du temps pour réévaluer le réseau.

La super classe `SuperActor` permet de développer des aServices de manière très simple et se charge de faire le nécessaire afin les inscrire ou désinscrire auprès du réseau en tant qu'aService disponible ou indisponible. Son modèle le rend indépendant de la plateforme sur laquelle il fonctionne.

[27] http://www.teleal.org/projects/mailinglists-cling.html#nabble-td2530182

12 Exportation du programme sur PC

12.1 Introduction

Le protocole implémenté permet de mettre en réseau des terminaux Android et ainsi partager les aServices. Il est intéressant d'étendre ce système et de pouvoir l'utiliser en dehors d'un environnement Android. L'intérêt de ceci est de permettre à différents systèmes d'exploitation de se partager des aServices de manière transparente quelque soit l'architecture ou l'OS de la machine. Avec ceci, il est donc possible d'utiliser des ressources d'une machine fonctionnant sur Windows depuis un terminal Android à faible puissance.

12.2 Implémentation

L'implémentation du protocole est faite dans quatre classes différentes qui sont contenues dans le paquetage "protocol". Ces classes peuvent donc être directement utilisées sur n'importe quelle plateforme car elles ne possèdent pas de dépendances liées à Android.

Toutes les classes utilisant des éléments d'Android ont du être adaptées. Il s'agit des classes utilisant des éléments de l'interface graphique ou qui interagissent avec le Wifi. Sur pc, l'interface graphique a été remplacée par un mode console, beaucoup moins intuitif mais qui permet de tout de même d'utiliser toutes les fonctions. Il serait néanmoins possible d'implémenter une interface graphique, mais nous avons décidé de ne pas y consacrer trop de temps au profit des tests et mesures.

La classe s'occupant des éléments du Wifi (démarrage / arrêt du Wifi, récupération de l'adresse IP) à été modifiée pour s'adapter sur des plateformes qui ne sont pas équipées d'Android.

Tous les aServices possèdent leur propre classe et sont dans un seul paquetage. Pour la plupart, leur implémentation à été créée de manière à ce qu'ils ne contiennent pas de composant Android. Chaque aService possède donc un paramètre dans son constructeur spécifiant la fonction à utiliser pour afficher les résultats. Sur pc, cette fonction est un simple `println`.

Deux acteurs n'ont pas été portés sur PC car ils utilisent des éléments directement liés à Android. Ce sont les aServices `Hypertoast` et `Measure`. `Hypertoast` utilise les `Toast` d'Android, qui sont un outil mis à disposition du système. Le `Mesaure` quant à lui affiche les données sous forme graphique grâce à DroidCharts. Il aurait été possible de porter toutes ces fonctions sur PC, car un `Toast` aurait pu être remplacé par un simple pop-up et l'affichage des valeurs mesurées aurait pu être fait avec JFreeChart[28].

[28] http://www.jfree.org/jfreechart/

12.3 Tests et mesures

Un test permettant de déterminer la pertinence du « remote computing » a été effectué sur 21 machines semblables. Tous ces postes, présents dans la même salle informatique, ont été équipés du programme précédemment archivé dans un jar. Chaque ordinateur à été inséré dans le réseau les un après les autres, permettant ainsi de tester la stabilité du protocole. Le test effectué consistait à mesurer le temps que mettait le système (de 1 à 21 pc) pour calculer pi avec 360'000'000 approximations, soit 10 fois plus que sur les appareils Android.

Le graphique ci-dessous représente le temps de calcul en fonction du nombre de pc.

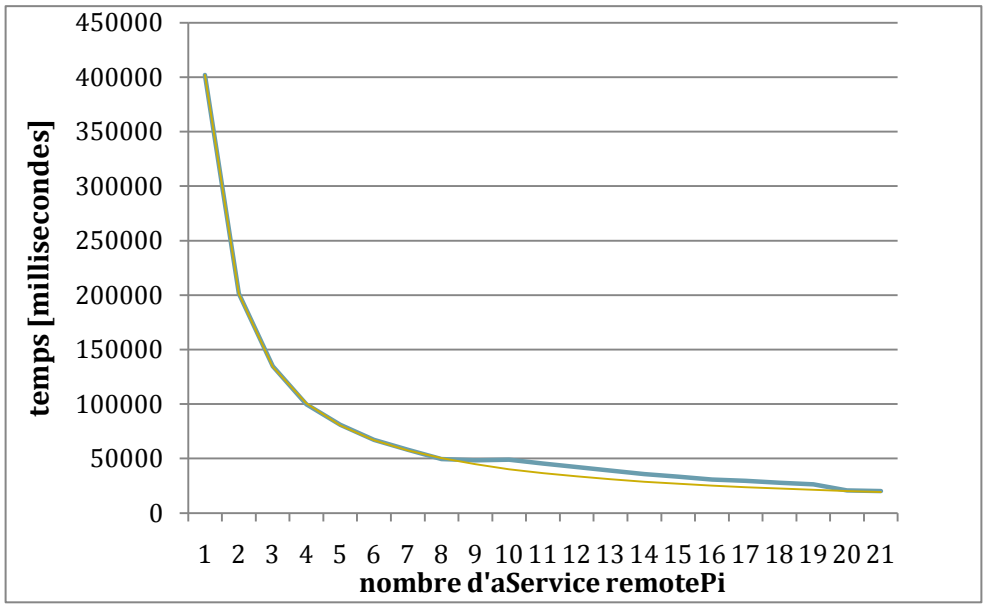


Figure 15 - relation temps de calcul / nombre d'aService, programme PC

La courbe jaune représente la fonction attendue, la bleu les valeurs mesurées. Les valeurs mesurées sont très proche des valeurs attendues. Un petit décalage est visible à partir de neuf aService, ceci peut être dû au fait que la 9ème machine soit plus lente que les autres et se fait donc attendre.

La fonction caractérisant la courbe attendue est la suivante :

$$temps = \frac{temps\ de\ calcul\ pour\ 1\ appareil}{nombre\ d'aService\ pi\ disponibles} * (nombre\ d'aServices - 1)Tprop$$

Le temps de propagation n'est pas pris en compte lorsqu'il n'y a qu'un appareil, car les informations passent par la *loopback* et non par le réseau. Idéalement, tous les appareils mettent le même temps pour exécuter l'opération mathématique, à cause de ceci la machine qui attend les valeurs retournées reçoit tout en même temps, les données vont donc arriver sur le port TCP les une après les autres.

12.4 Compatibilité avec le protocole Android

Par défaut, le protocole n'est pas compatible entre Android et PC. Ceci est du au fait qu'une classe utilisée par les acteurs *remote* ne possède pas le même UID sur les différentes plateformes. Cette classe, `Node`, permet d'identifier une machine en tant que nœud.

Pour remédier à ceci, il suffirait de fixer dans la classe `Node` l'UID et de recompiler les librairies sources de Scala. La modification consiste à ajouter une annotation. La classe ressemblerait à ceci :

```
@SerialVersionUID(12548l)
case class Node(address: String, port: Int)
```

12.5 Conclusion

Le portage du programme sur pc s'est effectué sans problèmes particuliers car seul une classe était dépendante du système Android (il s'agit de la classe s'occupant de la communication Wifi). Toutefois la compatibilité avec Android ne peut pas se faire sans modification de la classe `Node`.

Les valeurs mesurées grâce à l'aService `RemotePI` permettent d'observer que la distribution d'opérations arithmétiques permet d'accélérer considérablement le temps de calcul et que le temps de propagation peut être négligé. Nous disposons ainsi grâce à ce programme d'une petite plateforme de calcul distribué fonctionnel sur pc ou sur Android.

13 Conclusion

Des outils de développement pour coder en Scala sur Android existent, mais leurs mise en place demande différentes adaptations. D'un côté, SBT implémente au travers d'un plugin, toutes les fonctions nécessaires pour coder en Scala sur Android. Son utilisation est simple et efficace, mais très fermée. En effet, il devient vite compliqué de modifier les fichiers ou l'arborescence de dossiers crée par défaut par SBT. D'un autre côté, ANT est beaucoup plus souple car il se base sur différents fichiers extérieurs décrivant les différents paramètres ou éléments à utiliser lors de la construction du programme. Il ne permet par contre pas de télécharger automatiquement les dépendances comme c'est le cas pour SBT.

Dans la deuxième partie de ce travail, nous avons démontré que tous les composants d'Android sont utilisables au travers de Scala, sans restriction. Malgré tout, certains outils graphiques ne sont utilisables qu'avec des classes Java, comme les menus déroulants où il est nécessaire d'utiliser des tableaux Java. Il est donc impossible d'implémenter des menus en utilisant des tableaux de Scala, mais la simple conversion de tableau Scala en tableau Java suffit à remédier à ce problème.

L'utilisation de Scala permet une plus grande liberté et la programmation fonctionnelle peut, dans certain cas, faciliter l'implémentation. Le constructeur de `SuperActor` [annexe 8.1, p.1] permet d'illustrer les avantages de la programmation fonctionnelle. Il serait bien sûr possible de créer une fonction semblable en orienté objet, mais la syntaxe ne serait pas aussi simple. Sur Android, Scala apporte une manière de coder très concise et plus propre qu'en Java.

Les acteurs permettent d'implémenter des modèles de concurrence très puissants avec un niveau d'abstraction élevé. Leur implémentation sur Android permet de mettre en place un système de communication entre classes efficace. Il est possible de transmettre tout type de classes, contenant aussi bien des données que des fonctions, pour autant qu'elles soient sérialisables. Sur Android, n'importe quelle classe peut dériver d'un acteur, posséder son propre thread de communication et ainsi partager des informations aux travers d'acteurs.

Nous avons également montré dans ce travail que les acteurs *remote* fonctionnent sur Android. Leurs utilisations peuvent être multiples: ils peuvent être aussi bien un simple outil de communication entre différents terminaux, un outil de communication entre machines virtuelles (remplaçant ainsi les `Intent`, unique moyen de communication entre programme mis à disposition par Google) ou alors un outil permettant de distribuer des fonctions sur différents appareils (*distributed computing*).

La classe permettant de contacter un acteur *remote*, de type `AbstractActor`, peut être instanciée et utilisée n'importe où dans un programme Android. En revanche, la classe recevant les messages, dérivant d'`Actor`, ne peut pas être implémentée n'importe où : seules les classes `Activity` et `Services` sont capables de gérer correctement la réception des messages.

Les tests de « remote computing » démontrent qu'il est intéressant de distribuer des opérations mathématiques en fonction du nombre d'appareil afin d'accélérer les processus. Malgré tout, cette technique n'est pas optimale. En effet, si un appareil du réseau est moins puissant, il va ralentir tout le système. Pour remédier à cela, il faudrait distribuer les opérations en fonction de la puissance des appareils. Ainsi, chaque appareil recevra un calcul en fonction de sa puissance et ne se verra pas surcharger par un nombre d'opérations trop importantes. Cette méthode va être implémentée lors d'un prochain projet nommé « InduScala ».

Finalement, nous avons démontré dans ce travail la validité de l'approche et prouvé qu'il était raisonnable et possible d'écrire des applications Scala sous Android. L'utilisation des acteurs et des acteurs *remote* permettent d'implémenter des applications pertinentes dans le cadre des systèmes distribués ou du smart metering.

14 Dates et signatures

Sion, le 8 juillet 2011 Romain Cherix

15 Bibliographie

1. **Philipp Haller, Frank Sommers.** *Actors in Scala - Concurrent programming for the multi-core era.* Lausanne : s.n., 2011.
2. **Murphy, Mark.** *L'art du développement Android.* s.l. : Pearson, 2010.
3. **Mudry, Pierre-André.** *A tour of Scala - Student's overview.* 2011.