Master 1 - Ingénierie Informatique

Projet Android : DLNApp

Benoit Gallet
Emmanuel Herrmann
Bryce Marbois
Martin Rety

Table des matières

1 Introduction

1.1 Explications de DLNA et du protocole UPnP

Le DLNA est un service permettant de partager des fichiers multimédias entre différents périphériques sur un réseau domestique. Nous avons pour cela un côté serveur : un ordinateur ou un disque dur réseau par exemple, et un côté client : le périphérique sur lequel sera lu le flux comme un smartphone ou un téléviseur connecté.
Le service DLNA repose sur le protocole UPnP pour interagir avec le réseau. Ce protocole permet de connecter des périphériques entre eux afin de simplifier les échanges entre ces derniers. La première étape est la découverte des services présents sur le réseau et de permettre au périphérique d'indiquer ses propres services au serveur. La seconde est de récupérer les informations des périphériques. Nous pouvons à partir de cela communiquer avec les services des différents périphériques tout en informant les changements effectués.
En Android, il existe une bibliothèque implémentant ces fonctionnalités et donc permettant à un smartphone de développer une application utilisant ce protocole.

1.2 Bibliothèque Cling-Core

La bibliothèque Cling-Core fournit une architecture complète qui aide l'utilisateur à implémenter toutes les fonctionnalités nécessaires à UPnP. Celle-ci inclut un grand nombre de classes et de méthodes pour gérer la découverte, la connexion, la gestion de l'arborescence et la lecture d'un flux.
Cling-Core gère de façon interne les opérations réseau entre les périphériques (client et serveur) de manière asynchrone. Cela permet à l'utilisateur de ne pas avoir à se soucier de l'implémentation via une `AsyncTask` de ces tâches. L'utilisation de cette bibliothèque est très répendue, de nombreuses applications telles que BubbleUPnP l'emploient en effet pour la gestion du protocole UPnP. Ceci implique l'existence d'exemples et de supports aidant à sa compréhension.

2 Structure du projet

2.1 Structure DIDL et liste de devices : Package MyObject

La classe MyObject permet la mise en commun des informations générales aux trois catégories de MyObject que nous aurons : Device qui vient de la bibliothèque Cling-Core, Item et Container qui proviennent de la classe DIDL de Cling-Core. Ces trois sous-classes possèdent une icône, un titre et une description, informations communes aux affichages de n'importe quel MyObject.

— MyObjectDevice : possède deux attributs supplémentaires, urlIcon et Device. Etant donné qu'un serveur peut posséder sa propre icône, nous la récupérons si elle existe dans urlIcon afin de l'afficher à la place de l'icône

par défaut. Quant à Device, ce dernier permet de regrouper toutes les informations relatives à un serveur ainsi que différents services tels que le ContentDirectory qui permet de naviguer dans le système de fichier du Device.
- MyObjectContainer : possède un attribut container de type DIDLObject, et un attribut service de type Service. Ces derniers servent à parcourir une arborescence de type DIDL.
- MyObjectItem : possède également un attribut de type DIDLObject qui permet d'obtenir les informations d'une ressource, et d'une chaîne de caractère extension afin d'en connaître son type.

Le type DIDLObject (Digital Item Declaration Language) sus-mentionné, est une architecture décrivant une arborescence d'objets complexes. Celle-ci se compose de la manière suivante :
- DIDL : contient un ou plusieurs containers
- Container : contient d'autres containers ou des items
- Item : contient des items ou des components
- Component : contient des resources
- Resource : flux de données

Notre modélisation reprend le même schéma, mais en omettant deux niveau de profondeur (Item et Component). Ainsi le DIDL est représenté par MyObject-Device, Container par MyObjectContainer et Resource par MyObjectitem.

2.2 Gestion du protocole UPnP : Browser, MyHandler et RegistryListener

Les classes Browse et RegistryListener gèrent toutes les interactions liées au protocole UPnP et donc les appels des fonctions de Cling-Core.
Ces trois classes regroupent tous les appels liés au protocole UPnP : communications entre le client et le serveur, l'écoute de ces transactions et la gestion de l'arborescence DIDL.
- Browser : Cette classe possède un Service nommé serviceConnection, servant à communiquer avec le Service Cling-Core androidUpnpService. Ce dernier gère une pile de Device, et en lui ajoutant un listener ceci nous permet d'en ajouter, supprimer ou de la rafraîchir. La classe permet au final d'initialiser, de gérer et de terminer les différentes connexions, que ce soit à des Devices ou à des objets DIDL. On peut par exemple naviguer à travers l'arborescence d'un serveur grâce à browseDirectory.
- MyHandler : Sert à créer l'arborescence DIDl suite à une demande du Browser puis à en demander l'affichage.
- RegistryListener : Gère les évènements qui surviennent sur le réseau tels que la connexion ou la déconnexion d'un serveur, etc.

Pour effectuer différentes actions sur AndroidUpnpService nous utilisons la méthode getControlPoint(), donnant accès à différentes actions, comme search() qui recherche de manière asynchrone la liste des Device.

2.3 Lecteurs multimédias : Package MediaActivities

Lors du parcours de l'arborescence, si l'utilisateur sélectionne une ressource alors il a le choix de sélectionner une application tierce présente sur son téléphone, ou à défaut d'ouvrir le fichier sur un des lecteurs multimédia que nous fournissons.
L'activité MyImageViewActivity permet ainsi d'ouvrir toutes les ressources image présentes dans le dossier courant tout en pouvant les faire défiler.
MyVideoMusicViewActivity sert quant à elle à ouvrir les fichiers audio et vidéo. Nous rencontrons malheureusement quelques problèmes avec certaines extensions de fichiers qui ne sont pas compatibles de base avec Android. Ainsi un fichier dont l'extension est .mkv n'aura pas de son par exemple.

2.4 Algorithmique générale

La classe MainActivity implémente notre interface Notification. Cette dernière permet de communiquer entre l'activité principale et les différentes classes de notre structure. Elle permet de rajouter, supprimer, mettre à jour ou obtenir des informations sur les éléments de type MyObject de l'ArrayAdapter. Cette classe permet également de faire l'affichage et la gestion des actions de l'utilisateur. Elle crée un objet Browser qui, lors de son initialisation, recherche tous les périphériques présents sur le réseau. Le RegistryListener demande alors à l'interface leur affichage.
Quand un utilisateur veut accéder au contenu d'un Device ou d'un dossier, une action browseDirectory() est demandée par MainActivity. Celle-ci, en fonction du type d'objet envoyé par l'évènement, crée un nouvel objet MyHandler générant l'arborescence correspondante et en demande son affichage.
Sinon, lorsqu'un utilisateur sélectionne une ressource, une activité est créée dans l'appel de browseDirectory en fonction du type de l'objet choisi.

2.5 Autres fonctionnalités

Rafraîchissement de la page Nous avons ajouté deux manières de rafraîchir la page : un SwipeRefreshLayout et un bouton présent dans le menu. Ces deux actions déclenchent la même méthode refresh() de Browser.

Bibliothèque Picasso Dans le but d'afficher les images présentes sur le serveur, il nous a fallu utiliser une bibliothèque externe. Cette dernière peut ainsi les afficher à partir de leur URL.

Gestion du bouton "back" Par défaut le bouton Back d'Android réduit l'application en la mettant en arrière-plan. Nous l'avons modifié pour que celui-ci remonte dans l'arborescence DIDL.

3 Analyse du travail

3.1 Universalité

3.2 Distribution du travail

3.3 Problèmes rencontrés

3.4 Améliorations possibles

4 Exécution

Un jar est fourni, permettant l'exécution de l'algorithme. Le seul paramètre nécessaire est le nom du fichier .graphe (ainsi que son chemin s'il n'est pas situé dans le même répertoire que le jar.

Exemple : `java -jar ensemble_stable_maximum.jar test.graphe`

Si la syntaxe du fichier .graphe est exactement celle donnée en exemple, le fichier pourra être lu. Si cependant un graphe plus conséquent est donné, il est possible de retirer les virgules et les crochets de la liste de voisins pour avoir une écriture plus condensée. Cela est du au fait que nous découpons chanque ligne via le délimiteur ' '. Il est donc indispensable de garder les espaces et sauts de ligne entre chaque sommet. Les caractères '[', ']' et ',' sont automatiquement supprimés de la ligne, mais si d'autres caractères sont rajoutés dans le fichier, il ne seront pas supprimés par le parseur et pourront entraîner une erreur de lecture. Il est par contre obligatoire de noter les noms des sommets de 0 à n-1.

5 Implémentation

5.1 Structure du projet

Notre projet est organisé de la façon suivante : Une classe réservée à l'exécution de l'algorithme à proprement parler, quatre classes correspondant aux quatre briques, et une dernière classe graphe pour notre sutructure de données.

Le main appelle l'algorithme qui appelle à son tour les quatre briques suivant l'algorithme récursif de Fomin, Grandoni et Kratsch. Il retourne un entier en sortie (correspondant à la taille du plus grand sous-ensemble trouvé) qui est ensuite affiché dans la méthode correspondante.

Un objet graphe a été créé pour contenir la structure de données ainsi que plusieurs méthodes utilitaires. Un graphe est composé d'une Hashmap, dont la clé est un entier représentant le sommet, et la valeur une ArrayList d'entiers contenant les voisins de ce sommet. On a ainsi un accès très simple au sommet et à ses voisins, ce qui a beaucoup facilité l'implémentation de l'algorithme. De plus, l'ajout de plusieurs méthodes utiles permettent de manipuler cet objet très simplement. On peut trouver par exemple un textttoString() pour afficher proprement le graphe (ce qui a été particulièrement utile dans la phase de debug), des fonctions pour supprimer ou ajouter un sommet, voir son voisinage, ... Ce sont des méthodes reprises par les différentes briques, ce qui permet de factoriser le code.

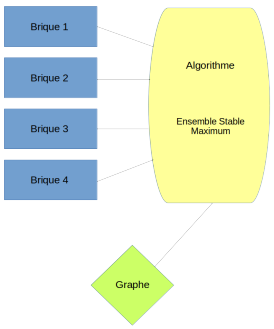

Figure 1. Structure

Brique 1 La Brique 1 est construite de façon à retourner une composante connexe du graphe et le graphe sans cette composante. Ainsi, on ne parcourt le graphe qu'une seule fois (via un parcours en largeur), car il suffit de tester si le graphe sans composante connexe est vide ou non. S'il l'est, alors il n'y a pas de composante connexe et on peut poursuivre l'algorithme avec la brique 2.

Pseudo-code de la Brique 1 :

```
test(Graphe g)
Liste fifo;
sommetCourant = prendreUnSommet();
Graphe composanteConnexe;
Graphe gSansConnexe;
gSansConnexe.supprimerSommet(sommetCourant);

TantQue fifo n'est pas vide Faire
    sommetCourant=defiler(fifo);
    Pour tous les voisins v de sommetCourant Faire
        Si v n'a pas encore été rencontré
            Ajouter v à fifo;
            composanteConnexe.ajouterSommet(v);
            gSansConnexe.supprimerSommet(v);

Retourner composanteConnexe, gSansConnexe;
```

Brique 2 La brique 2 sest composée de deux méthodes : une de test pour voir s'il existe un sommet dominant, et une d'exécution pour retourner le graphe transformé.

Le test vérifie s'il existe un sommet possédant au moins le même voisinage qu'un autre sommet, et s'il sont voisin entre eux.

Pseudo-code de Test :

```
test(Graphe G){
    Pour tous les sommets s1 de G Faire
        Pour tous les sommets s2 de G Faire
            Si s1 != s2 Alors
                Si s1 et s2 sont voisins Alors
                    Pour tous les voisins v de s1 Faire
                        Si v est n'est pas voisin de s2 Alors
                            Retourner -1;
                    Retourner s2;
    Retourner -1;
}
```

Brique 3 Comme la brique précédente, celle-ci est composée de deux méthodes : une de test et une d'exécution. Le test vérifie si le graphe possède un sommet 2-pliable grâce à un parcours du graphe et de vérification du voisinage de chaque sommet. Run retourne le graphe transformé par la brique 3.

Pseudo-code :

```
Test(Graphe G){
Pour tous les sommets s de G Faire
    Si s est de degré 2 Alors
        Si les deux sommets voisins de s ne sont pas voisins entre eux Alors
            Retourner s;
        }
    }
    Retourner -1;
}
```

Brique 4 La brique 4 est séparée en deux parties : celle qui renvoie le graphe moins v et ses miroirs, et l'autre qui renvoie le graphe moins v est ses voisins.

Pseudo-code de miroir :

```
Miroir(sommet s, Graphe G){
    Liste sommetsASupprimer;
    sommetsASupprimer.ajouter(s);
    Liste voisin_distance2;
    Pour tous les voisins u de v Faire
        Pour tous les voisons w de u Faire
            Si w != v, v n'est pas un voisin de w et voisin_distance2 ne contient pas w
                voisin_distance.ajouter(w);
    Liste clique_a_tester;
```

```
    Pour tous les u dans voisin_distance2 Faire
        Pour tous voisins w de v Faire
            Si w n'est pas un voisin de u Alors
                clique_a_tester.ajouter(w);
        Si clique_a_tester est une clique Alors
            Si sommetsASupprimer ne contient pas u Alors
                sommetsASupprimer.ajouter(u);
    Pour tous les sommets u dans sommetsASupprimer Faire
        G.supprimerSommet(u);
    Retourner G;
}
```

6 Justification de la brique 3 (pliage)

Nous noterons S l'ensemble stable maximum de G.

Cas 1 : v $\in$ S

Si v appartient à S, alors ses deux voisins u_1 et u_2 n'appartiennent pas à S (de par la définition). On peut alors retirer v du graphe et dire que la taille de S sera la taille de l'ensemble stable maximum du nouveau graphe $\tilde{G} + 1$. De plus, les voisins de v ne peuvent pas faire partie de S (comme v y est déjà), on peut donc les fusionner et travailler sur $\tilde{G}$ comportant notamment ce sommet ainsi que ses voisins (le sommet nouvellement créé u_{12} ne fera pas partie de S car les 2 sommets fusionnés n'en faisait pas partie).

Cas 2 : u_1 ou u_2 $\in$ S

Ce cas revient au cas 1. En effet, lors de la transformation, la suppression de v ainsi que la la fusion entre u_1 et u_2 fait que le nouveau sommet u_{12} ne fait pas partie de S. Or u_1 ou u_2 en faisait partie, il est donc logique de dire que la taille de S est de 1 + la taille de l'ensemble stable maximal de $\tilde{G}$.

Cas 3 : u_1 et u_2 $\in$ S Comme u_1 et u_2 font tous les deux partie de S, on peut supprimer v car il n'est relié qu'à ces deux sommets et n'a donc aucune incidence sur l'ensemble stable maximum. Par contre, comme on a fusionné deux sommets faisant partie de S en un seul, il est alors logique de compenser la perte d'un de ces deux sommets en disant que la taille de S sera de 1 plus celle de l'ensemble stable du nouveau graphe $\tilde{G}$.