UNIVERSITÉ DE SHERBROOKE

Projet SDF

Dans le cadre du cours ift630, nous avons réalisé un mini projet qui consiste en un système de partage de fichiers distribué (Système de Diffusion de Fichiers). Il offre aux utilisateurs des services de bases en ce qui concerne la manipulation des fichiers.

Préparé par:
- Ternisien d'Ouville Matthieu 11 114 199
- Youcef Hammou Ahmed

29 avril 2012

Table des matières

Architecture de l'application 1

Architecture générale 1

Thread UDP 1

Thread TCP 2

L'IHM 2

FileSys (FS), FileSysNetwork (FSN) et FileSysOper (FSO) 2

Groupe et liste de fichiers 2

Action possible sur le système de fichier. 3

Configuration 3

Conclusion 4

Evolution possibles 4

Difficulté rencontrée 4

1. Architecture de l'application

1. Architecture générale

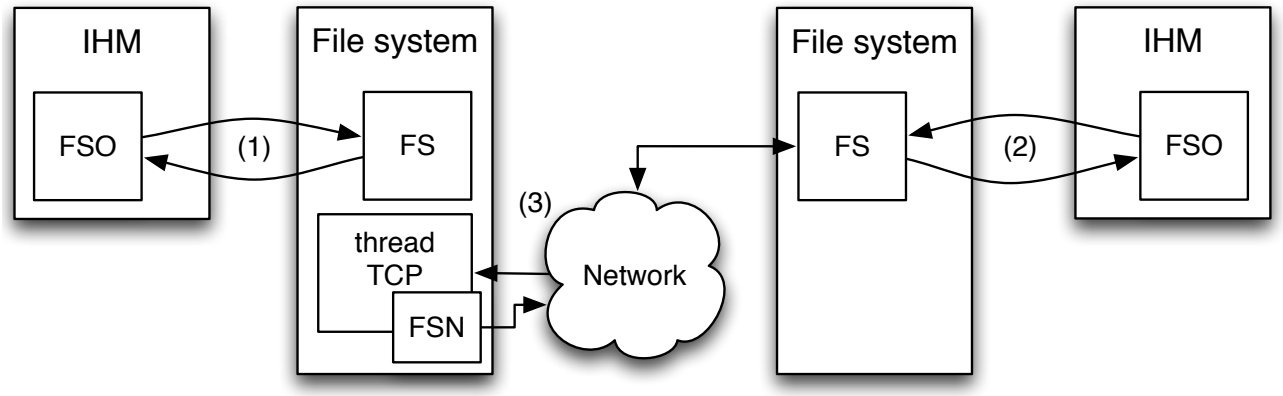

Figure 2.1.1 : architecture générale de l'application

L'IHM permet la communication entre l'utilisateur et le système de fichier. Le système de fichier est constituer de 3 fils d'exécutions.
- Le 1er, le fil UDP, permet de réceptionner la connexion et la déconnexion des systèmes ce trouvant sur d'autres machines.
- Le seconde, fil TCP, permet de gérer les différentes demandes qui peuvent être faites sur le système de fichiers (3).
- Le dernier, le fil principal, gère la communication avec l'interface (1 et 2).

(1) L'IHM fait une demande au système de fichier. Dans certain cas (ex. : aucune autre machine connectée) le système de fichier répond uniquement avec les données dont il a accès localement.
(2) Dans la majorité des cas, le système de fichiers fera des demandes aux autres machines connectées. Elle utilisera le protocole TCP pour ce connecter au autres machines.
(3) Dans ce cas, une connexion TCP aura lieu entre la machine cherchant l'information est les autres machines du réseau. Lors-ce que le thread TCP reçoit une demande de connexion il créait un objet *FileSysNetwork* qui s'occupera de répondre à la demande avec les informations du système de fichier local.

2. Thread UDP

Le thread UDP gère la synchronisation avec les systèmes qui se connectes. Ainsi que de leurs déconnexions. Pour cela il tient à jour une liste de tout les systèmes de fichiers présents sur le réseau.

3. Thread TCP

Ce fil d'exécution à pour but de recevoir des demandes d'informations sur son système de fichier local. Ces demandes peuvent être :

- Vérifier l'existence d'un fichier,
- Récupérer la liste des fichiers présents sur la machine,
- Transférer un fichier,
- Supprimer un fichier.

Nous avons choisie d'utiliser le protocole TCP pour la communication entre les machines plutôt que le protocole UDP pour simplifier la programmation et avoir un système plus résistant aux pannes.

4. L'IHM

L'interface homme-machine permet aux utilisateurs d'interagir avec le système de fichier. L'IHM propose de :

- Supprimer un fichier,
- Ajouter/créer un fichier,
- Afficher la liste des fichiers du système,
- Upload/Download des fichiers.

Toute ces fonctionnalitées seront détaillées plus bas.

5. FileSys (FS), FileSysNetwork (FSN) et FileSysOper (FSO)

C'est 3 classes ont toutes un objectif différent mais restent très proches les unes des autres.

FileSysOper permet à l'IHM de communiquer avec le système de fichier. *FileSys* récupère les demandes faites par *FileSysOper* (l'IHM) et y répond. *FileSysNetwork* permet de répondre à une demande faites par *FileSys*. Ce dernier et *FileSysOper* communique grâce à des tubes (pipe) de communication. *FileSys* et *FileSysNetwork* communique par des sockets TCP.

6. Groupe et liste de fichiers

Chaque système de fichiers à un groupe et une liste de fichiers. Le groupe correspond à une liste d'adresse des autres systèmes de fichiers présents sur le réseaux.
Le système de fichiers à aussi une liste des fichiers présent en local. Ce permet d'éviter un accès au disque dur toutes les fois où on souhaites connaitre la liste des fichiers.

L'accès aux données du groupe et de la liste de fichier peut être faites par plusieurs fils d'exécution au même moment. Pour assurer l'exclusion mutuelle, nous utilisons des mutex.

2. Action possible sur le système de fichier.

Pour le moment 7 actions sont possibles sur le système de fichiers :

- Vérifier qu'un fichier existe : Cette action est la plus utilisée, car la plus part des autres actions s'en servent. Elle vérifie d'abord en local puis sur les autres machines du groupe si le fichier existe. La méthode permettant de vérifier l'existence d'un fichier, retourne l'adresse de la machine qui contient ce fichier. Ce qui permet d'optimiser les autres fonctions, en communiquant uniquement avec celle qui détient le fichier.
- Ajouter un fichier : Le système vérifie si le fichier existe, puis créer un fichier vierge en local.
- Récupérer la liste des fichiers complète : Copie la liste des fichiers présents sur la machine, puis la concatène avec la liste de chacune des autres machines.
- Upload un fichier : Vérification de l'existence du fichier puis ajout de ce dernier dans le système de fichier local, s'il n'existe pas. Si le fichier existe, il peut être remplacé. Dans ce cas, le système demandera la suppression du fichier, puis il le copiera sur sont système de fichier local.
- Download un fichier : Le système vérifie l'existence du fichier puis, si besoin, le télécharge sur la machine, ou le copie.
- Supprimer un fichier : Vérification de l'existence du fichier puis suppression.
- Arrêter le système de fichiers : Le système de fichier ce déconnecte et le processus est détruit.

3. Configuration

Pour permettre le fonctionnement de l'application, il faut modifier certains paramètres dans le fichier main.cpp :

- La variable *ipBroad* correspond à l'adresse broadcast du réseau.
- La variable *ip* correspond à l'adresse IP de la machine.
- La variable *path* correspond à l'emplacement de la racine du système de fichiers.

Une fois ces variables modifiées, il reste à compiler le programme grâce à la commande suivante :

```
$ g++ -pthread fileSys.cpp fileSysNetwork.cpp fileSysOper.cpp fileList.cpp group.cpp mySocket.cpp myUdpSocket.cpp tube.cpp main.cpp -o sdf
```

Aucun paramètre n'est demandé pour exécuter le programme.

4. Conclusion

1. Evolution possibles

Beaucoup de fonctionnalitées, n'ont pas été implémentées. Par exemple, pour le moment le système de fichiers ne gère qu'un dossier (la racine). Il n'est pas possible de créer des dossiers. Il aurait par exemple pu être intéressant d'ajouter une base de données réparti pour pouvoir gérer une arborescence de fichier en graphe.

Cette première version du système ne permet pas plusieurs actions simultanées sur le système de fichier. Chaque demande faites par un système de fichiers sur un autre est traitée séquentiellement. Une solution parallèle pause de nombreux problème, Tel que le fait qu'un fichier ne doit pouvoir être modifier par 2 machines simultanément ou encore, un fichier ne doit pas pouvoir être supprimé si il est en cours de téléchargement. Une table des fichiers utilisés aurait permit d'empêcher ces problèmes.

Nous n'avons pas pu développer l'IHM. Il aurait par exemple pu être intéressant de mettre en place une interface proche des clients FTP. Ou encore un shell, ce qui permettrait au utilisateur de développer leur propre commandes.

2. Difficulté rencontrée

Lors du développement de ce projet nous avons rencontré quelques difficultés. La première à été les différents choix d'implémentation que nous avons du faire. Par exemple le choix du protocole TCP plutôt que UDP pour les échanges d'informations entre les machines.

Nous avons aussi du trouver un moyen d'éviter les inter-blocages lorsqu'un fils d'exécution entre dans une section critique et qu'il y a une exception non gérée par le système. Nous avons, pour résoudre ce problème, suivie l'idiome RAII (Resource Acquisition Is Initialization). Cette technique de programmation permet d'être sûr que le mutex est bien relâché. Le principe et de créer un objet qui verrouille le mutex lors de sa création, et le libère à sa destruction.

Un autre problème que nous avons rencontré, concerne la partie réseau. La taille de certain type de données n'est pas la même en fonction des systèmes (Linux/Unix, système 32/64 bites). Dans notre cas, il s'agissait notamment du type *size_t* qui avait une taille mémoire de 8 octets sur un système BSD 64 bites (Mac OS X 10.7) et de 4 octets sur un système Linux (Ubuntu 11.04). La définition de traits auraient permis d'éviter ce type de problèmes.