DropBophinex.

Projet Applications réseau M1 Miage Apprentissage Paris Dauphine 2015/2016

F. Sikora (D'après un sujet de M. Chilowicz)

1 Dates importantes

- Rendu de la RFC: 10 juin 2016, 23h.
- Rendu du projet: 4 juillet, 23h.
- Soutenances: 6 juillet 2016, à partir de 13h30. (à confirmer)

2 Versions

Ce sujet est susceptible d'être modifié.

- 12 mai 2016: Mise en ligne du sujet.

3 En quelques mots

La synchronisation d'un répertoire à distance est utile si l'on désire faire une sauvegarde de secours (backup) distante. On travaillera de manière unidirectionnelle, c'est-à-dire que la copie des fichiers s'effectue à partir d'une source locale (resp. distante) vers une destination distance (resp. locale). Rien n'empêche de lancer la commande deux fois (une fois dans chaque sens) pour avoir deux répertoires identiques.

Le but du projet est de proposer un protocole de communication entre un client et un serveur afin de synchroniser un ou des répertoire locaux et à distance, ainsi qu'une implémentation en Java d'un tel client et d'un tel serveur. Afin de minimiser le temps de transfert, le protocole essayera de transférer uniquement l'information ajoutée plutôt que de tout retransférer à chaque modification.

Pour qu'une copie puisse s'effectuer, il faut qu'un serveur (un démon) existe déjà sur l'hôte distant. On le lancera via la commande:

```
java -jar DropBophinexServeur.jar repertoire [PORT]
```

`Repertoire` est le répertoire père où les répertoires seront synchronisés. Par exemple, s'il vaut `/home/tartampion/backups`, on pourra synchroniser le répertoire `/home/tartampion/backups/vacances2017` (si le client indique vacances2017 comme répertoire) mais pas `/home/tartampion/photosHonteuses`. `PORT` est le port d'écoute du serveur (optionnel, par défaut le serveur écoute sur le port défini par le protocole).

Une synchronisation se fera au moyen d'une de ces commandes:

```
java -jar DropBophinexClient.jar [OPTIONS...]  SRC HOTE[:PORT]/DEST
java -jar DropBophinexClient.jar [OPTIONS...]  HOTE[:PORT]/SRC DEST
```

La première commande envoie sur la machine `HOTE` (dont le port d'écoute peut éventuellement être spécifié, sinon port par défaut du protocole) le contenu du répertoire `SRC` de la machine locale vers le répertoire `DEST` du serveur (plus précisement, la concaténation du répertoire d'écoute du serveur avec `DEST`). La seconde fait l'opération inverse et rapatrie le répertoire distant vers le répertoire local.

On veut proposer au moins les options suivantes (qui ne font sens que si le répertoire existe déjà dans `DEST`):

- `-d` ou `--delete` supprime de l'hôte destination les fichiers présents dans `DEST` mais pas dans `SRC` (cela peut arriver si un fichier a été supprimé entre 2 backups). Attention, cela implique qu'un effacement par erreur sur la source supprime également le fichier sauvegardé après la synchronisation !
- `-i` ou `--ignore` n'écrase pas les fichiers déjà présents dans `DEST`.

Pour simplifier, on considère qu'aucun processus externe ne peut modifier le contenu des répertoires source et destination pendant un transfert.

4 Protocole

L'hôte de la source communique à l'hôte de la destination le contenu du répertoire à transférer, i.e. l'ensemble des informations (chemin, taille, date de modification) sur les fichiers présents dans le répertoire, et de manière récursive sur les répertoires.

L'hôte destination reçoit ces informations et détermine les fichiers dont il a besoin de recevoir le contenu afin de synchroniser le répertoire. Il ignore donc les fichiers qu'il possède déjà et dans le même état (même chemin dans le répertoire, même taille, même date de modification). Il ignore également les fichiers ayant le même chemin (mais une taille ou une date de modification différente) si et seulement si l'option `ignore` est indiquée. Les autres fichiers devront être transférés (donc créés ou écrasés). Pour cela, l'hôte destination envoie à l'hôte source la liste des fichiers qu'il souhaite obtenir.

L'hôte source reçoit une liste de fichiers demandés par l'hôte destination. Pour chaque fichier demandé, l'hôte va le découper en morceaux (de la manière précisée dans le protocole) et calculer un hash (aussi appelé somme de contrôle) selon une fonction de hachage (précisée dans le protocole) pour chaque morceau. L'hôte envoie ensuite l'ensemble des hashs à l'hôte destination.

L'hôte destination reçoit l'ensemble des hashs et recherche s'il possède déjà certains de ces morceaux en local (afin d'éviter de transférer véritablement le morceau de fichier) parmi les fichiers présents dans le répertoire père (à vous de voir s'il vaut mieux rechercher dans un seul fichier ou dans tout le répertoire). L'hôte destination compare alors les hashs des morceaux déjà présents avec ceux envoyés par l'hôte source et peut ainsi déterminer et envoyer à l'hôte source la liste des morceaux qu'il ne connait pas et dont il souhaite le transfert.

L'hôte source reçoit cet ensemble de hashs et envoie le contenu de ces morceaux. L'hôte destination peut alors assembler les morceaux déjà connus avec les morceaux reçus.

Découpage et calcul des hash

Il faut donc découper les fichiers en morceaux et calculer un hash pour chacun d'entre eux. On utilisera pour cela une fonction de hachage prenant un tableau d'octets (correspondant au morceau du fichier) et retournant un hash. Pour un hash de taille raisonnable, le risque de collision est de probabilité epsilonesque. Ces fonctions renvoient la même valeur étant donnée la même entrée: ainsi, si le même morceau existe sur 2 hôtes différents, leur hash sera le même. Il existe de telles fonctions (telles que MD5 ou SHA 256) déjà implémentées dans le JDK dans la classe MessageDigest. La méthode `digest(byte[] input)` devrait suffire (une fois l'objet `MessageDigest` obtenu avec `getInstance`). Faites attention aux cas de fichiers de très grande taille dans la gestion de vos buffers.

Plusieurs manières de découper les fichiers sont envisageable. La plus simple consiste à fixer une taille dans le protocole et de découper les fichiers en morceaux de cette taille (sauf le dernier morceau). Attention au choix de la taille (si les morceaux sont plus petits que la taille des hash, il n'y a aucun intérêt à faire ces tests, si les morceaux sont trop grands, on diminue la probabilité d'avoir des morceaux déjà existants). Si cette méthode est employée par l'hôte source ET l'hôte destination, les modifications en fin de fichier impliqueront le transfert uniquement des derniers morceaux et on pourra garder les morceaux de début de fichier sans problème (ces premiers morceaux non modifiés auront gardé le même hash). Par contre, l'ajout d'un seul octet en début

de fichier décale la valeur de tous les morceaux et aucun hash ne sera alors identique: tout devra être retransféré.

On pourrait donc aussi envisager que l'hôte destination découpe de manière différente les fichiers déjà connus afin d'éviter ces inconvénients. Attention cependant, le calcul des hash est une opération coûteuse en temps (un cache serait à envisager pour une application réelle).

5 Implémentation

On demande l'implémentation en Java d'un client et d'un serveur, selon le protocole que vous aurez défini (ils devront fonctionner sur les machines où Java est disponible (attention notamment au séparateurs des répertoires différents sous windows et linux).

Le serveur devra être capable de gérer plusieurs demandes de synchronisations en simultané. Pour simplifier, on fera en sorte d'autoriser l'accès concurrent à un répertoire uniquement si on est dans un mode lecture. Si on est en mode écriture (le serveur est l'hôte destination au moment du transfert), les demandes devront être traitées les unes après les autres, les autres demandes seront mises en attente.

On veut pouvoir annuler (proprement) une opération de synchronisation en cours.

On désire également avoir des statistiques après une opération (nombre de fichiers et d'octets échangés, nombre d'octets économisés grâce au mécanisme, etc.).

Des améliorations au projet sont possibles, si les fonctionnalités de bases demandées sont parfaitement implémentées (option pour faire un essai ne faisant aucun changement, filtre d'exclusions de fichiers, historique des versions de chaque fichier, etc.). Vous pouvez également modifier le protocole proposé par le sujet si vous le souhaitez. **Aucune** interface graphique n'est demandée.

6 Conditions de rendu

Le projet est à effectuer en binôme, i.e. par 2 personnes. Comme vous êtes un nombre impair d'élèves, un seul groupe sera autorisé à effectuer le projet en trinôme (noté plus sévèrement).

Première étape: description du protocole. Une première RFC de votre protocole est à rendre avant la date définie plus haut sur l'espace MyCourse dédié. Les noms des deux personnes du binôme devront apparaitre dans la RFC, dans le nom du fichier ainsi que dans les commentaires de l'espace MyCourse. Attention, l'espace est fermé automatiquement. La RFC doit être dans le même format que les autres RFC, c'est-à-dire un document texte ASCII rédigé en anglais (voir la RFC 2223 pour le format d'une RFC..., voir aussi ce lien `http://www.rfc-editor.org/formatting.html`). La RFC doit seulement décrire les méthodes de communications (format des messages (tel entier, tel code, tels caractères encodé de telle manière, etc), ordres des messages (avant, après, pendant), protocoles utilisés...) entre les différentes entités intervenant dans votre protocole. Aucun détail d'implémentation ne doit être donné. Elle doit contenir toutes les informations nécessaires par une personne désirant implémenter votre protocole, mais pas d'informations superflues. Typiquement, le nombre de threads utilisés par un serveur est une information inutile, mais un message où on précise l'ordre "chemin - date - taille" plutôt que "chemin - taille - date" est une information importante. La RFC 1350 (TFTP) est un exemple de RFC simple.

Seconde étape: implémentation pour le protocole en langage Java. Théoriquement, deux implémentations de différents binômes pourraient communiquer entre elles si elles suivent la même RFC (théoriquement toujours, une implémentation en langage C++ lancée sur une machine personnelle pourrait communiquer avec une implémentation Java sur un téléphone androïd).

Votre projet est à rendre avant la date précisée plus haut, sur l'espace MyCourse dédié. Un seul membre du binôme fait une soumission (plusieurs essais possibles). Une fois votre fichier envoyé, vous devez avoir un écran de confirmation avec votre fichier et la date de l'envoi. **Il y aura un point en moins par heure de retard, une heure entamée est due.**

Le format de rendu est une archive au format ZIP contenant le code source de votre projet, une doc utilisateur indiquant comment utiliser votre projet, une doc développeur, votre RFC première version et une autre éventuellement mise à jour (avec explication des changements dans la doc développeur).

Votre projet doit pouvoir s'exécuter sans utiliser eclipse et bien évidement en réseau (serveur et client sur des machines différentes).

L'archive aura pour nom Nom1Nom2.zip, où Nom1 et Nom2 sont les noms des membres du binôme par ordre alphabétique. L'extraction de l'archive devra créer un dossier Nom1Nom2.

On préférera un projet qui fonctionne bien avec peu de fonctionnalités qu'un projet bancal avec plus de fonctionnalités.

Vous pouvez utiliser les dépôts GIT créés pour le projet Java.

Soutenance

Une soutenance de 10 minutes aura lieu binôme par binôme à la date précisée plus haut. Elle doit être préparée et menée par le binôme (i.e. fonctionnant parfaitement du premier coup, en réseau, etc). Des jeux de tests permettant de voir que votre projet fonctionne bien (synchronisation, détection de contenu déjà transféré etc.) sont à prévoir.