SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TD 7 : Système de fichiers réparti

Objectifs pédagogiques :

- **Interfaces et communication RMI**
- **Service réparti**
- **Interaction entre objets RMI**

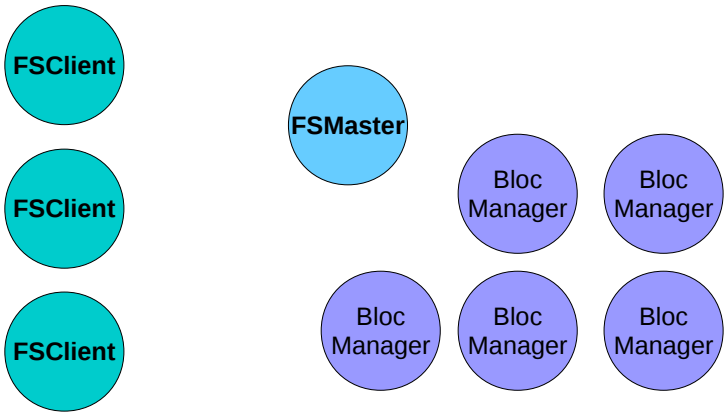
Exercice 1 – Gestion répartie des blocs

Nous souhaitons coder en RMI un système de fichiers réparti. Les fichiers manipulés ne sont pas stockés sur une seule machine mais sont répartis en blocs sur un ensemble de machines appelées `BlocManager`. Pour gérer les méta-données du système de fichiers (par exemple la liste de l'ensemble des `BlocManager`, l'emplacement des blocs de chaque fichier, la liste des fichiers, etc.) nous aurons besoin d'une machine maître que l'on appellera `FSMaster`. Le `FSMaster` ne stockera aucun bloc.

Afin de fournir l'API classique d'un système de fichier (création, lecture, écriture, suppression de fichier, etc.), on définit la classe `FSClient` qui décrit les objets hébergés sur les machines clientes et qui interagiront avec les avec le `FSMaster` et les `BlocManager`. Ainsi, cette classe offre les méthodes suivantes :

- `boolean create(String name)` :
  - ◇ ***Spécification*** : créer un nouveau fichier pour un nom de fichier donné. Si la création est impossible (ex : le fichier existe déjà), la méthode renverra faux, vrai sinon.
  - ◇ ***Implémentation*** : On enregistrera auprès du `FSMaster` un nouveau fichier. Puisqu'un fichier est vide à sa création, aucun bloc ne sera alloué sur les `BlocManager`.
- `boolean remove(String name)` :
  - ◇ ***Spécification*** : efface un fichier pour un nom de fichier donné. Si la suppression est impossible (ex : le fichier n'existe pas), la méthode renverra faux, vrai sinon.
  - ◇ ***Implémentation*** : on déclarera au `FSMaster` que le fichier n'existe plus. Le `FSMaster` devra en conséquence communiquer avec les `BlocManager` pour que ces derniers effacent les blocs du fichier correspondant.
- `boolean read(String name, int offset, byte[] buf)` :
  - ◇ ***Spécification*** : lit `buf.length` octets à partir de l'octet `offset` dans le fichier de nom `name`. Elle renverra faux si on a lu moins de `buf.length` octets, vrai sinon.
  - ◇ ***Implémentation*** : on s'adressera dans un premier temps au `FSMaster` pour connaître la liste des `BlocManager` qui contiennent les blocs du fichier. Cette liste sera triée dans l'ordre croissant des blocs du fichier c'est-à-dire que le `BlocManager` à l'indice i héberge le bloc i du fichier. Une fois cette liste obtenue, on récupérera le contenu de chaque bloc voulu pour remplir le buffer.
- `void write(String name, int offset, byte[] buf)` :
  - ◇ ***Spécification*** : écrit les données de `buf` à partir de l'octet `offset` dans le fichier de nom `name`. Pour simplifier, on considérera que l'offset est borné par la taille du fichier.
  - ◇ ***Implémentation*** : comme la lecture, on récupérera la liste des `BlocManager` qui contiennent les blocs du fichier auprès du `FSMaster`. En fonction de l'offset et de la quantité de données à écrire, le client s'adressera au fur et à mesure aux `BlocManager` concernés pour récupérer

une copie des blocs à modifier. Une fois qu'un bloc est modifié en local, le client renverra ce bloc au `BlocManager` correspondant. Si l'écriture de données concerne des blocs inexistants, alors il y a nécessité d'allouer de nouveaux blocs dans le système. Pour allouer un nouveau bloc on s'adressera alors au `FSMaster` pour que celui-ci choisisse un `BlocManager` pour ensuite lui communiquer d'allouer un nouveau bloc. La méthode d'allocation du `FSMaster` renverra une référence sur le `BlocManager` sur lequel le nouveau bloc a été alloué. Ce processus d'écriture se répétera tant qu'il restera des données à écrire.



Nous considérerons que l'ensemble des `BlocManager` est statique et que le système est fiable, c'est-à-dire qu'aucune machine ne peut tomber en panne. Nous poserons la taille d'un bloc à 1024 octets. Chaque bloc est stocké sous forme de fichier local sur le système de fichier du `BlocManager`. Pour simplifier, nous ne gérerons pas la synchronisation des clients pour les concurrents à un même fichier.

Question 1

En supposant qu'un fichier "toto" ait une taille de 4050 octets, combien de messages seront envoyés dans le système si on souhaite écrire 750 octets à la fin de ce fichier ? Vous justifierez votre réponse en précisant la séquence de messages échangés dans le système.

Question 2

Donner l'interface RMI des services offerts par le `FSMaster`.

Question 3

Donner l'interface RMI des services offerts par un `BlocManager`.

Question 4

Quel intérêt pour les accès en lecture ou écriture de s'adresser directement aux `BlocManager` pour le transfert de données

Question 5

Quel intérêt y a-t-il à passer par le `FSMaster` pour allouer ou supprimer des blocs sur le système ?

Question 6

Les protocoles entre les différentes entités du système sont-ils avec ou sans état ? Justifiez.

Question 7

Quel(s) attribut(s) la classe `FSClient` doit-elle avoir ?

Question 8

Donner le code (ou le pseudo-code) de la classe `FSClient`.

Question 9

Nous supposons qu'un `BlocManager` peut tomber en panne.

1. Quel mécanisme pourriez-vous mettre en place pour détecter la panne du `BlocManager` ?
2. Quel mécanisme pourriez-vous mettre en place pour assurer la disponibilité des données ?

Donner les éventuelles modifications sur les interfaces ou les nouvelles interfaces à définir.

Exercice 2 – Ajout d'un mécanisme de cache

On se replace dans un monde où les `BlocManager` ne peuvent pas tomber en panne. Par conséquent, vous ne prendrez pas en compte les modifications apportées à la dernière question de l'exercice précédent.

On souhaiterait ajouter un mécanisme de cache entre les `BlocManager` et les `FSClient` pour accélérer les accès sur les données d'un bloc.

Chaque client possède donc un cache local de bloc. Les lectures et écritures se font directement dans le cache et non plus sur l'appel distant au `BlocManager`. Dans un premier temps les écritures sont directement envoyées sur le serveur.

Pour assurer une cohérence forte, le `BlocManager` diffuse chaque mise à jour d'un bloc qu'il héberge à tous les `FSClient` possédant une copie du bloc.

Question 1

Pourquoi l'objet gérant le cache local doit-il offrir une interface distante ?

Question 2

Modifier l'interface du `BlocManager` pour prendre en compte les accès en cache.

Question 3

Donnez l'interface offerte par le cache ainsi qu'une classe d'implémentation.

Question 4

Quelle hypothèse faut-il faire sur les canaux de communication (justifiez) ?

Question 5

Sachant qu'un bloc est utilisé en moyenne chez M `FSClient`, quelle est la complexité moyenne (en nombre de messages) de l'algorithme ?

Question 6

Pour réduire la charge des `BlocManager`, lorsqu'un client modifie un bloc, il diffuse directement à tous les autres clients un message invalidant le bloc dans leur cache. La donnée modifiée n'est pas envoyée au `BlocManager`. Quel est l'avantage de cette stratégie par rapport à la précédente ?

Question 7

Décrire un algorithme permettant à un client de charger la dernière version d'un bloc sans effectuer aucune diffusion. Vous définirez clairement les structures de données et variables utiles sur chaque site. Vous considérerez les hypothèses suivantes :

- H1 : les mises à jour ne sont pas transmises au `BlocManager`
- H2 : pour réduire la charge du `BlocManager` celui-ci ne connaît que le premier site client possédant une copie du bloc.

Question 8

De manière générale, quelle est la complexité moyenne en nombre de messages de votre algorithme ?

Question 9

Peut-on réduire le nombre de messages lorsqu'un client réclame à nouveau un bloc qu'il possédait préalablement ? Si oui, modifiez votre algorithme en conséquence.