PJE - F

Giuseppe Lipari

November 2, 2020

Contents

1 Description générale

Le projet jusqu'ici s'est déroulé en modalité *single-node* : vous pouvez gérer les vidéos disponibles sur un seul noeud Raspberry.

Maintenant, on suppose qu'il y a un réseaux local de noeuds Raspberry, chacun doté d'une caméra et d'un espace de stockage locale ou les vidéos sont enregistrées.

À partir d'un noeud, nous voulons accéder en lecture, de manière la plus transparente possible, les vidéos enregistrées sur les autres noeuds.

L'objectif final est de concevoir et programmer un systèmes de fichiers repartis sur plusieurs noeuds adapté aux exigènces de notre application.

Pour faire ça, on va travailler sur 3 étapes successives :

1. Architecture client / serveur pour lire un fichier distant ;
2. Système de notification "Publish/Subscribe" ;
3. Système de fichiers en user space (librairie FUSE).

2 Première étape : lire un fichier à distance

Pour lire le contenu d'un fichier distant, on utilise un système client/server.

Sur chaque noeud on lance un serveur, c'est à dire un programme qui attends une connexion sur un port bien connue (par exemple, on choisi le port 4044). Le serveur gères tous les fichiers contenus dans un répertoire bien connu.

De que un client se connecte au serveur, il peux lui demander une des commandes suivants :

1. Commande `open <filename>` : ouverture d'un fichier en lecture. Le client spécifie le nom du fichier: si le fichier existe et il est lisible, le serveur lui réponds avec un code "Ok" (ça peut-être un entier); si le fichier n'existe pas ou s'il n'est pas lisible, le serveur réponds avec un code d'erreur.

   Un client peut ouvrir un seul fichier à la fois. Si un fichier est déjà ouvert, le serveur renvoie un code d'erreur.

2. Commande `read <size>` : lecture de `size` octets du fichier déjà ouvert précédemment; si aucun fichier n'est pas ouvert, le serveur renvoie un code d'erreur; si non, il renvoie les informations suivants :

   - le nombre `N` d'octets à transférer (un numéro non négatif inférieur à la taille demandé `size`)
   - les `N` octets.

   S'il n'y a plus d'octets à lire (*end of file*) le serveur renvoie la valeur $N = 0$.

1. Commande `close` : fermeture du fichier; le dernier fichier ouvert est fermé et le serveurs renvoie un code "Ok". S'il n'y a aucun fichier ouvert, le serveur renvoie un code d'erreur.

2. Commande `list` : renvoie la liste de tous les fichiers dans le répertoire. Le serveur envoie un chaîne de caractères, avec un nom de fichier par ligne, et qui se termine avec 2 lignes vides.

3. Commande `stat <filename>` : le serveur renvoie les informations suivantes sur un fichier donnée :

   - taille (en octets) ;
   - date de la dernière modification ;
   - droits d'accès.

   (vous pouvez ajouter d'autre informations, si nécessaire)

Il doit être possible pour plusieurs clients de se connecter au même temps au même serveur. Pour faire ça, la structure du serveur doit être *multi-thread* : à chaque fois qu'un client se connecte, le serveur crée un thread qui gère la connexion avec le client ; le programme principale du serveur se remets en attente d'autres connexions.

Veuillez noter que le serveur garde séparément les information sur chaque connexion ouverte avec un client: donc, il est possible pour deux client de lire des fichiers en parallèle.

2.1 Implémentation

Vous pouvez utiliser soit le langage Python, soit le langage C/C++ pour implementer le serveur.

Il faut aussi implementer un programme client qui permet de tester le bon fonctionnement du serveur.

2.2 Exemple de serveur multi-threaded en Python

Le programme suivant fait l'écho des messages reçus.

```
# Echo server program
import socket
import threading

def worker(conn, addr) :
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            print("Received: ", data.decode('utf-8'))
            conn.sendall(data)

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    while True :
        conn, addr = s.accept() # accept a connection
        t1 = threading.Thread(target=worker, args=(conn, addr))
        t1.start()
```

2.3 Exemple de client

Le programme suivant est un simple client pour tester le server "écho".

```
# Echo client program
import socket

HOST = 'localhost'    # The remote host
PORT = 50007          # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', data.decode('utf-8'))
```

3 Deuxième étape : Publish / Subscribe

TODO