Université de Strasbourg
Réseaux
CSMI M2 — Année 2019-2020
Hadetskyi Vadym

Rapport du Projet

Bibliothèque distribuée tolérante aux pannes

1. Description et objectifs

L'objectif principal du projet est de concevoir une bibliothèque distribuée tolérante aux pannes. La bibliothèque contient des documents électroniques, représentés sous forme de fichiers: un document est constitué d'un et un seul fichier. On remarque que la bibliothèque peut contenir n'importe quel type de fichier (par exemple ".txt", ".pdf", ".mp3", etc.). De plus, le programme doit permettre le transfert de fichiers de toute taille.

Ces fichiers sont stockés sur plusieurs serveurs pour des problèmes de redondance et de tolérance aux pannes. On considère que le nombre maximum de serveurs constituant la bibliothèque est faible.

Un lecteur souhaitant accéder à la bibliothèque commence par obtenir le catalogue, c'est-à-dire la liste de tous les fichiers offerts par les serveurs connus. Il est possible qu'un ou plusieurs serveurs ne répondent pas (ils sont alors considérés comme en panne), et l'obtention du catalogue est limitée dans le temps à un certain délai (par exemple 30 secondes).

Pendant cette période de temps, le lecteur tentera de se reconnecter un nombre prédéfini de fois et abandonnera à l'expiration du temps.

Par la suite, si le catalogue est obtenu, le client peut procéder au téléchargement du fichier. Lorsque le lecteur souhaite accéder à un document, il le

télécharge depuis un des serveurs l'ayant annoncé. À ce stade-ci on considère que le serveur n'est pas en panne.

2. Structure de l'implémentation

L'implémentation proposée de ce projet consiste en 3 fichiers de programme qui produisent des exécutables, un fichier d'en-tête et un Makefile :

- serveur.c — le programme qui définit tout le travail effectué par le serveur ;
- lire.c — le programme qui définit le travail de récupération d'un fichier;
- catalogue.c — le programme qui définit le travail d'obtention du catalogue;
- jobs.h contient les fonctions utilitaires, les fonctions d'actions, les paramètres globaux et la structure « catalog_record » utilisées par les deux clients et le serveur;
- Makefile — utilisé pour compiler les fichiers de programme et produire les executables qui fournissent le travail du client ou du serveur.

```
//=======================Globales=======================
// Les variables globales communes
#define MAXLEN  1024
#define MAX_NAME_LEN 255
#define DGRAM_SIZE 258
#define SEGM_SIZE 1500
#define MAX_FPATH 4096

// structure pour stocker le catalogue
struct catalog_record
{
  uint16_t id ;
  uint8_t oct_len ;
  char name[MAX_NAME_LEN] ;
} ;
```

Figure 1. Les paramètres globaux et la structure « catalog_record »

Il a été décidé de définir une structure globale, utilisée lors du transfert du catalogue, et quelques variables globales communes au serveur et au client pour les maintenir en consensus sur les paramètres de communication et le format du message envoyé via UDP (Fig. 1).

Ces variables définissent également les limites de la solution proposée:

- *MAXLEN* — définit la taille maximale du message lors d'une communication simple client -> serveur ; Utilisé uniquement en mode test, débugage.
- *MAX_NAME_LEN* — définit la longueur maximale du nom du fichier.
- *DGRAM_SIZE* — définit la taille maximale d'une structure et d'une datagram UDP par conséquent.
- *SEGM_SIZE* — définit la taille maximale d'un segment TCP.
- *MAX_PATH* — définit la longueur maximale du chemin vers un fichier. Normalement, sa valeur correspond à celle des systèmes basés sur Unix.

La structure « catalog_record » contient id d'un fichier dans catalogue (2 octets), la longueur du nom (1 octet) et le nom lui-même du fichier. Ainsi, si le nom du fichier est plus long que *MAX_NAME_LEN*, il n'est pas inclus dans le catalogue. Ces variables peuvent être modifiées en fonction des besoins de l'utilisateur (programmeur), mais la congruence doit être sauvegardée. Par exemple, le programme échouera si *MAX_NAME_LEN* est supérieur à *DGRAM_SIZE*.

3. Le serveur

Le server est un programme tournant à l'infini qui attend la connexion du client, fait son travail et continue d'attendre d'autres connexions.

Il peut être lancé par la commande suivante :

```
./serveur port répertoire
```

Une fois lancé, il commence par analyser le répertoire donné. Il trouve le nombre de fichiers dans le dossier, sa taille totale et il forme le catalogue qui est

conservé pour le reste de la vie du programme et jamais modifié (Fig. 2). Si le dossier est modifié, le serveur doit être redémarré.

Figure 2. Le démarrage du serveur

Il procède par la création et la configuration des *MAXSOCK* sockets TCP et UDP. *MAXSOCK* est aussi une variable globale, qui n'est définie que dans le cadre de serveur, puisque le client n'en a jamais besoin.

On remarque que, bien que *MAXSOCK* sockets soient créé, le serveur ne supporte pas plus que *MAXCON* connexions simultanées. Si la file d'attente est remplie et qu'une nouvelle file d'attente arrive, selon la page de manuel « listen », le client peut recevoir une erreur avec l'indication *ECONNREFUSED* ou, si le protocole sous-jacent supporte la retransmission, la requête peut être ignorée pour qu'une nouvelle tentative ultérieure à la connexion réussisse. Par ailleurs *MAXCON* ne peut être plus grande que 128 et elle va être tronquée autrement.

Puisque le travail du serveur va probablement devenir assez lourd, on parallélise la gestion des requêtes TCP et UDP. Le processus fils est crée avec fork() pour la réception des connexions TCP. Quant au processus initial, il gère l'envoi du catalogue sur UDP, comme c'est un travail relativement léger.

Dans la boucle infinie, il vérifie s'il y a une nouvelle connexion entrante. Si c'est bien le cas, il l'accepte et il suit le protocole (correspond à la fonction « send_record »):

1. Le serveur attend l'identifiant du fichier;
2. Il reçoit un message et il effectue la conversion de *Network Byte Order;*
3. Il envoie la ligne correspondante du catalogue ;
4. Lorsque le serveur reçoit une demande avec un numéro $p \geq n$, il répond avec un nom de document vide.

On remarque que cela se passe dans la boucle commune pour toutes les sockets. Il n'est probablement pas optimal pour la mise à l'échelle, mais dans le cas donné, ayant un nombre assez limité de connexions et seulement 258 octets à envoyer, il fait parfaitement son travail, aucun délai n'a été observé.

En même temps, dans la boucle infinie, la partie TCP du serveur vérifie s'il y a une nouvelle connexion TCP entrante. Si c'est bien le cas, le nouveau processus-fils est crée, parce que le transfert de données peut être une tache assez lourde et longe. Il accepte la connexion en recevant également l'identifiant du fichier en NBO sur 2 octets et continue dans la fonction « send_file ». On décrit l'implementation de « 3-way handshake », qui est destiné à établir la connexion entre deux agents dans TCP:

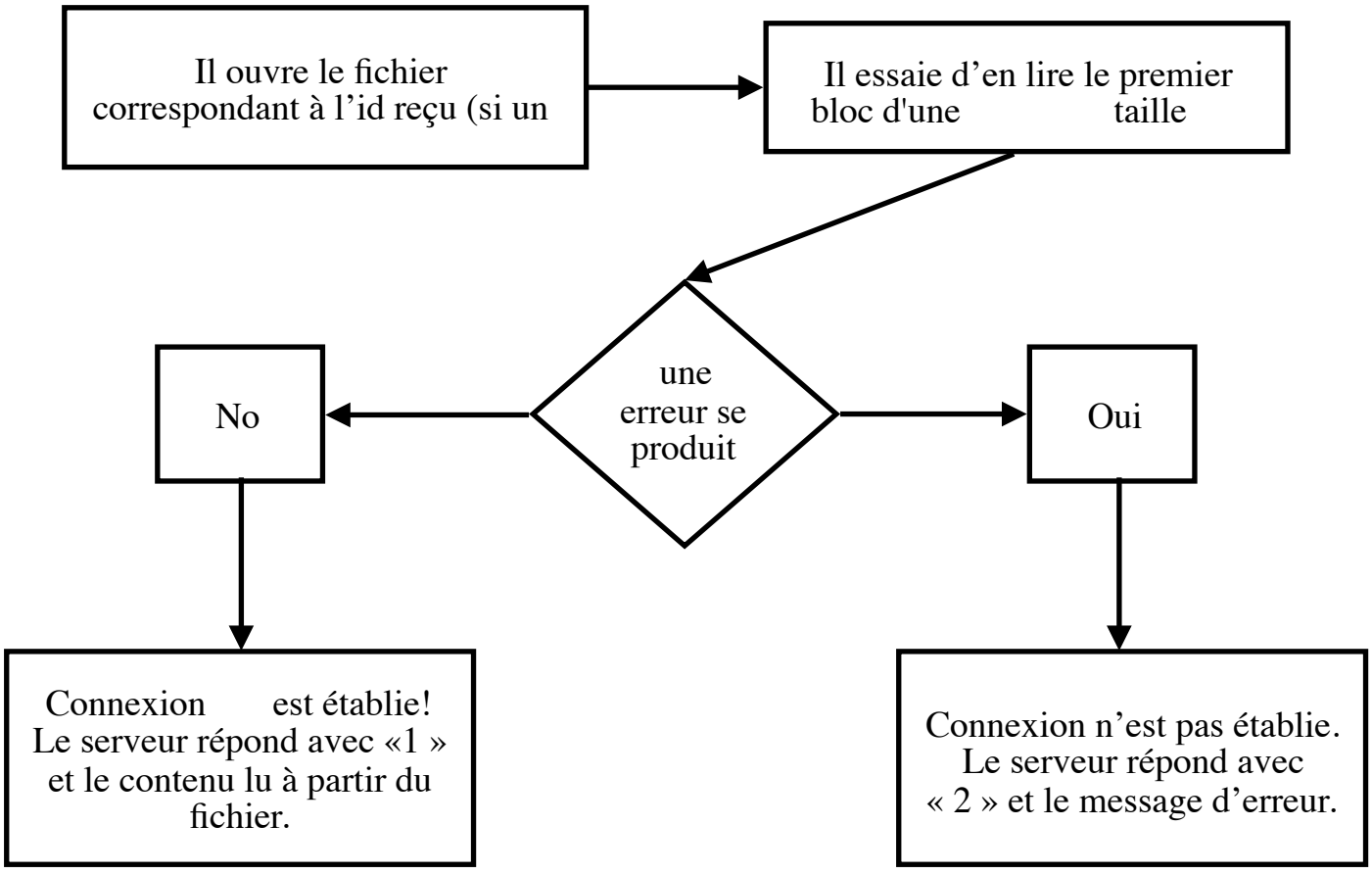

Figure 3. Etablissement de la connexion TCP

Si la connexion est établie le serveur continue à lire le fichier et à envoyer son contenu tant que "read" ne renvoie pas 0 pour un nombre arbitraire d'itérations (3). Il s'agit d'une simple mesure de sécurité, car "read" peut lire toute quantité d'octets inférieure ou égale à celle demandée.

Il est à noter que la lecture se fait dans une buffer de mémoire qui, après avoir été envoyée, est effacée à l'aide de la fonction « memset ». Cette fonction permet de « annuler » le buffer, en remplissant la mémoire d'un octet constant.

Lorsque la fin du fichier est atteinte, son descripteur se ferme, le processus enfant se termine, la socket, utilisée pour cette connexion est également fermée.

4. Protocoles utilisés — Obtention du catalogue

Le protocole UDP est utilisé pour l'obtention du catalogue. Cette operation est effectué en appelant:

```
./catalogue délai serveur port serveur port …
```

Plusieurs serveurs peuvent être donnés comme sources de catalogues. Alors, pour rendre le programme plus efficace on crée un processus fils par serveur&port combinaison. Chaque processus-fils récupère son adresse IPv4 ou IPv6 et port, crée le socket et ajuste ses caractéristiques. On remarque l'utilisation de flags *SO_SNDTIMEO* et *SO_RCVTIMEO* dans setsockopt. Ils servent de timers pour l'envoi et la réception respectivement. Pour les définir on a également besoin de définir une structure « timeval » (Fig. 4) :

```
// delai
struct timeval timeout;
float rem = atoi(argv[1]) % MAX_COMM_TRIES ;
// 1 tv_usec = 1/1000000 de 1 tv_sec
rem = rem / MAX_COMM_TRIES * 1000000 ;
rem = (int)rem ;
timeout.tv_sec = atoi(argv[1]) / MAX_COMM_TRIES ;
timeout.tv_usec = rem;
```

Figure 4. Définition de timer

Les operations d'envoi et de la réception peuvent potentiellement être bloquantes. Les flags mentionnés plus haut fixent la durée maximale pendant laquelle ils peuvent l'être. Cependant, nous aimerions essayer de nous reconnecter plusieurs fois en cas de blocage, donc le délai réel pour chaque opération est calculé comme délai / *MAX_COMM_TRIES*. Chaque fois on va réessayer d'obtenir meme fichier ou le blocage est arrivé.

Il convient également de mentionner que les deux programmes clients sont capables de travailler avec les adresses IPv4 et IPv6. C'est fait grâce à la fonction de la bibliothèque standard associée aux sockets Berkeley « inet_pton »:

```
//=======================IPv4||IPv6=======================
if (inet_pton (AF_INET6, padr, & server_adr6->sin6_addr) == 1)
{
  family = PF_INET6 ;
  server_adr6->sin6_family = AF_INET6 ;
  server_adr6->sin6_port = port ;
  len = sizeof *server_adr6 ;
}
else if (inet_pton (AF_INET, padr, & server_adr4->sin_addr) == 1)
{
  family = PF_INET ;
  server_adr4->sin_family = AF_INET ;
  server_adr4->sin_port = port ;
  len = sizeof *server_adr4 ;
}
else
{
  fprintf (stderr, "%s: adresse '%s'non reconnue\n",\
   argv [0], padr) ;
  exit (1) ;
}
```

Figure 5. Traitement des adresses IPv4 & IPv6

Le même bloc de code est présent dans « lire.c » pour atteindre le même but.

Après avoir fini de traiter les adresses et de configurer les sockets on commence la boucle jusqu'a *ENTRY_LIMIT* — le nombre maximum de fichiers dans un catalogue. Bien sur, il y aura pas forcement toujours ce nombre des

fichiers, alors suivre les règles proposées, la boucle va arrêter en cas d'une réception d'un nom du fichier vide.

A l'intérieur de la boucle on sauvegarde le contenu de chaque message reçu dans l'instance de la structure catalog_record et on l'affiche sur la sortie standard. Finalement, on obtient un résultat suivant:

```
Vadyms-MacBook-Pro:Projet_BDTAP thewaveorthemountain$ ./catalogue 1 ::1 1221
Obtention du catalogue...
Le serveur || Port || L'enregistrement || La taille || Le nom
::1 || 1221 || 0 || 8 || Lava.jpg
::1 || 1221 || 1 || 6 || jobs.h
::1 || 1221 || 2 || 6 || little
::1 || 1221 || 3 || 10 || Sluggy.png
::1 || 1221 || 4 || 15 || cours1_insa.pdf
::1 || 1221 || 5 || 14 || 04 Gravity.mp3
::1 || 1221 || 6 || 9 || some_text
::1 || 1221 || 7 || 20 || TurtleMartinique.mov
```

Figure 6. Obtention du catalogue

Quand le processus-fils fini l'obtention du catalogue à partir de son serveur, il ferme son socket et il quitte avec le code 0.

Le processus-père attend attend que chaque fils finisse son travail.

5. Protocoles utilisés — Récupération d'un document

La récupération d'un document s'effectue en suivant le protocole TCP. On peut le faire en appelant commande suivante:

```
./lire serveur port document
```

Au début de son travail, le programme crée un dossier « Downloaded_From_Port » dans lequel enregistrer les fichiers téléchargés, s'il n'existe pas déjà. Là, on va stocker tous les fichiers téléchargés à partir du meme serveur. Ils vont être sauvegardés sous le nom « File_document », où document est le paramètre qui a été donné pour le programme et qui correspond a l'id du document dans le catalogue obtenue.

Le programme met au point les sockets et les adresses à utiliser de meme façon comme décrit dans une partie « Obtention du catalogue ». Par ailleurs, on note, que on ne va pas attendre le délai donné et on ne va pas essayer de rétablir la connexion en cas d'échec, alors les flags *SO_SNDTIMEO* et *SO_RCVTIMEO* ne sont pas précisés. Puisque le protocole TCP est utilisé pour la récupération d'un document, on considère qu'après l'établissement d'une connexion la communication est sécurisé et fiable.

Le travail principal est fait dans la fonction « download », définie en « jobs.h ». D'abord on essaie d'établir une connexion avec le « 3-way handshake », décrit déjà dans la parti « Le serveur » (Fig. 3). Si la connexion échoue en raison d'une défaillance du serveur, le client obtient code 2 avec l'erreur système produit. Alors, on l'affiche sur la sortie standard et on quitte avec 0.

Sinon, tout va bien, le fichier est crée à partir du descripteur donné à la fonction et l'écriture commence. Tant qu'il reçoit quelque chose du serveur, le client l'écrit dans le fichier. On n'oubli pas également d'annuler le buffer utilisé pour la récupération des messages lus avec « recv ».Enfin, on ferme le descripteur du fichier, et on libère le memoire de buffer.

Il faut remarquer, que les fichiers téléchargés ne gardent pas leur extension d'origin. On peut les verifier en affichant avec « ./catalogue … » et ajouter manuellement. Pareil pour les noms : « File_2 » n'est pas très explicite, cependant, dans le monde réel, l'utilisateur final obtiendrait très probablement un menu déroulant avec des emplacements possibles pour enregistrer le fichier et un champ pour donner le nom sous lequel il préfère que le fichier soit enregistré.

6. Les tests

Pour s'assurer dans la performance des programmes de multiples tests ont été effectués.

La première étape du test a été effectuée sur des fichiers texte simples de tailles différentes, supérieures et inférieures à la taille définie du segment.

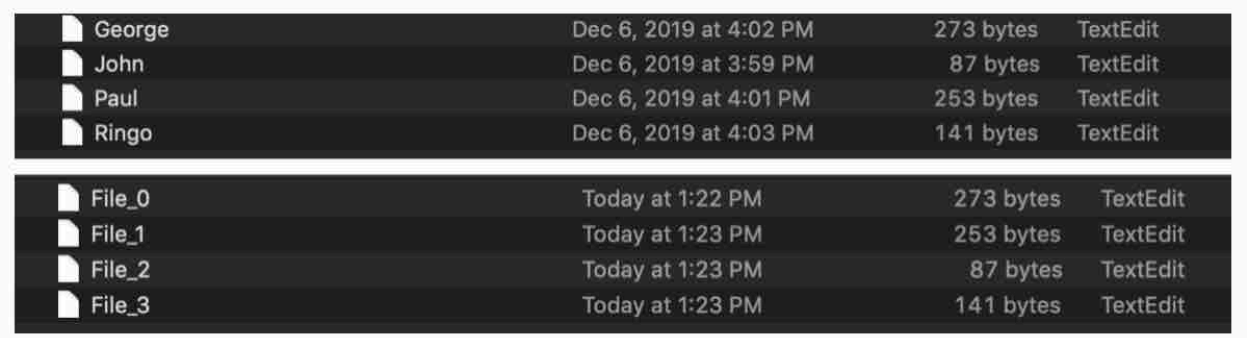

Figure 7. Tests simples txt

La première capture d'écran est faite à partir du dossier du catalogue du serveur, la seconde dans le dossier créé par le client pour les fichiers téléchargés. On constate que les fichier gardent leur taille, ce qui est plutôt la condition nécessaire.

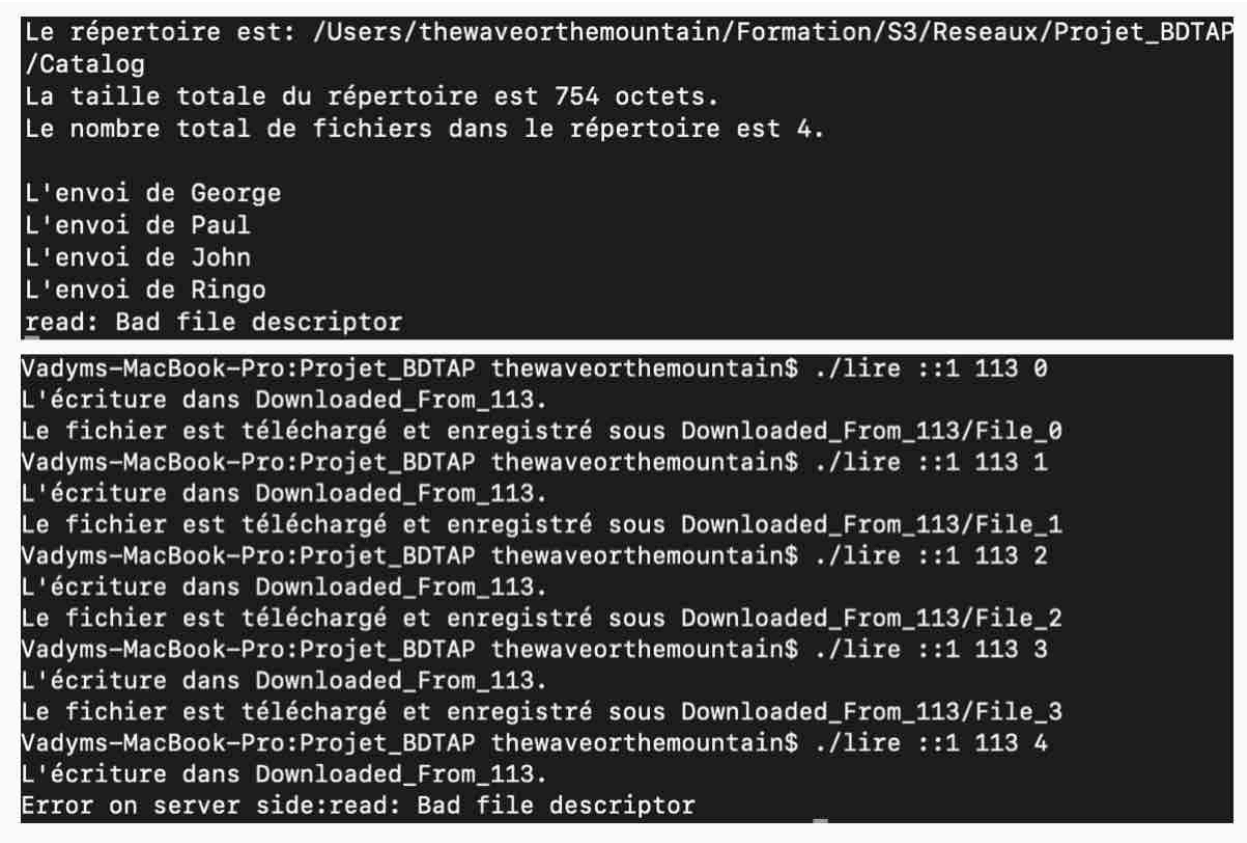

Figure 8. Tests simples — affichage du serveur et du client

Ces opérations sont soutenues par l'affichage montré sûr la Fig. 8. On voit que le serveur envoie les fichier correspondantes aux identifiants reçus et affiche l'erreur après avoir lu « 4 », qui n'existe pas dans le catalogue de 4 fichiers.

On remarque, que malgré l'erreur obtenue, le serveur n'a aucun problème de continuer son travail et traiter les requêtes suivantes, puisque c'était le fils qui a fini son travail et a quitté avec 1.

La deuxième phase d'essais a été menée sur différents types de fichiers, comme le montre la Fig. 9.

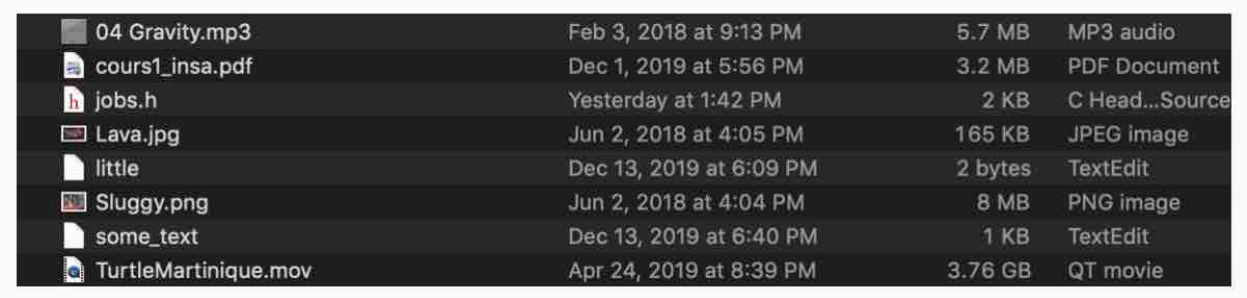

Figure 9. Tests avancés — plusieurs format différents

On commence par l'obtention du catalogue:

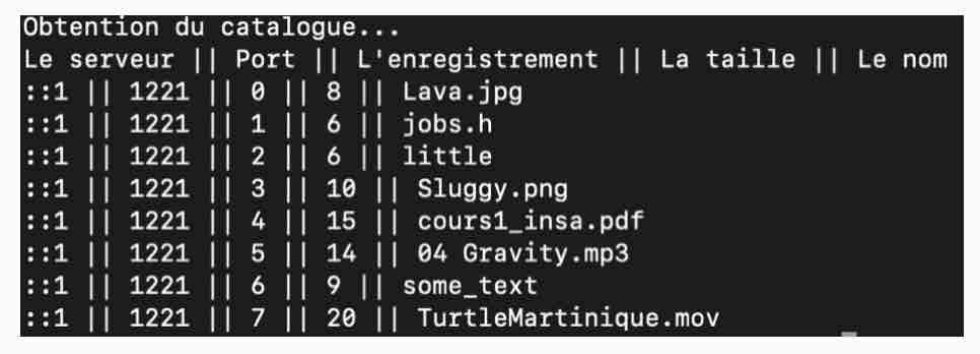

Figure 10 (6). Obtention du catalogue

On récupère les fichiers un par un (Fig. 11). On constate que les fichier téléchargés ont gardé leur propre taille (Fig. 12) et on peut verifier leur fonctionnalité et ouvrant avec l'application destiné.

```
La taille totale du répertoire est 3776216870 octets.
Le nombre total de fichiers dans le répertoire est 8.

L'envoi de Lava.jpg
L'envoi de jobs.h
L'envoi de little
L'envoi de Sluggy.png
L'envoi de cours1_insa.pdf
L'envoi de 04 Gravity.mp3
L'envoi de some_text
L'envoi de TurtleMartinique.mov
```

Figure 11. Récupération des fichier

File_0	Today at 1:48 PM	165 KB	TextEdit
File_1	Today at 1:48 PM	2 KB	TextEdit
File_2	Today at 1:48 PM	2 bytes	TextEdit
File_3	Today at 1:48 PM	8 MB	TextEdit
File_4	Today at 1:48 PM	3.2 MB	TextEdit
File_5	Today at 1:48 PM	5.7 MB	TextEdit
File_6	Today at 1:48 PM	1 KB	TextEdit
File_7	Today at 1:49 PM	3.76 GB	TextEdit

Figure 12. Dossier avec les fichiers téléchargés

On voit que tout est également juste pour les fichier des format plus sophistiques. Finalement, on ouvre chaque fichier avec une propre application pour voir si tout fonctionne correctement. Donnons un exemple d'un grand fichier vidéo sous le nom « TurtleMartinique.mov » ou bien « File_7 » (Fig.13).

Figure 13. Verification d'un transfert de video

On ouvre le « File_7 » avec VLC et on voit que tout fonctionne correctement; on peut regarder la vidéo dans son intégralité sans pertes d'informations.

Enfin, on vérifie que les programmes marchent toujours sur « turing » et on fait des testes similaires y étant connecté.

Par ailleurs, plusieurs tentatives de lancement d'un serveur et d'un client sur deux machines différentes ont été faites, bien que la connexion n'ait jamais été autorisée. Certaines sources suggèrent que cela peut souvent se produire à cause des paramètres du réseau et que dans ce cas, il n'y a aucun moyen de le corriger dans le programme. Pour cette raison, il n'était pas possible de tester l'accès au serveur par son nom, mais cela ne devrait pas être un problème, car "getaddrinfo" devrait normalement le traiter correctement.

7. Conclusion

En conclusion, l'implementation propose l'implémentation proposée de la bibliothèque distribuée est fiable et efficace et permet d'obtenir des catalogues de plusieurs serveurs à la fois, ainsi que de télécharger les fichiers qu'ils contiennent. On confirme que les fichiers téléchargés conservent leur structure et ne perdent aucune information.

Diverses optimisations de performance peuvent être introduites en cas de besoin de mettre le programme en échelle à de plus grandes bibliothèques ; de même que certaines extensions d'utilisation peuvent être faites pour rendre les programmes plus pratique en utilisation, surtout pour l'utilisateur final.

Appendix — Gestion du projet

Lorsque vous travaillez sur un grand projet, il est essentiel de le diviser en tâches plus petites pour les réaliser une par une. Bien que le plan donné n'ait pas été suivi avec précision, il a permis de définir les principaux éléments du projet et d'en suivre l'avancement.