Projet L3 Réseau

1 Objectif

Le but du projet est de produire deux outils permettant l'échange de fichiers entre un client et un serveur. Il faudra donc développer:

- Un client C capable de récupérer et d'envoyer des fichiers à distance
- Un serveur de fichier écrit en Java capable de suivre les ordres autant du client précédent que d'un navigateur web classique.

Quelques exemples d'utilisations:

- Lancement du serveur : **./server 7654** Lance le serveur sur le port 7654
- Envoie d'un fichier : **./client put 192.168.0.1 7654 bidule.jpg** envoie le fichier **bidule.jpg** sur le serveur lancé sur la machine **192.168.0.1** sur le port **7654**
- Afficher un fichier : **./client get 192.168.0.1 7654 machin.txt** télécharge le fichier **machin.txt** qui est sur le serveur **192.168.0.1** en passant par le port **7654** et l'affiche
- Récupération d'un fichier : **./client get 192.168.0.1 7654 machin.jpg bidule.jpg** télécharge le fichier **machin.jpg** qui est sur le serveur **192.168.0.1** en passant par le port **7654** et le sauvegarde dans le fichier **bidule.jpg** du répertoire courant
- Une fonction de test sera rajoutée au client : **./client test 192.168.0.1 7654 "coucou"** affiche (puis quitte) à l'écran la réponse du serveur quand le client lui envoi la chaine de caractère **coucou**

Tous les outils (client et serveur) devront suive le protocole HTTP décrit en annexe (Section 7).

2 Déroulement du projet

Le projet se déroulera en 5 séances de présentiel. Ce temps de présentiel sera principalement destiné à débloquer les problèmes rencontrés, une majorité du projet sera à faire entre ces séances.

Les sources concernant l'outil en C seront à envoyer le vendredi suivant la seconde séance à votre encadrant de TP. Les sources totales du projet seront à envoyer le vendredi suivant la dernière séance.

Il faudra toujours joindre aux sources le *makefile* utilisé pour la compilation. Il est obligatoire !

3 Client en C

Le client en C est composé de l'outil en ligne de commande présenté précédemment, ainsi qu'un outil de test 'miroir'.

Le client est capable de trois fonctions:

- Récupérer un fichier
- Envoyer un fichier
- Faire un test

La récupération et l'envoie d'un fichier devront suivre le protocole http. Vous trouverez une description du protocole http en annexe.

Le code de l'outil de test 'miroir' est en annexe (section 6). Il faudra le modifier pour pouvoir passer en paramètre le port sur lequel le lancer. Il faudra aussi le modifier pour qu'il puisse recevoir la connexion successive de plusieurs clients. (Optionnel: On pourra faire une version capable de recevoir plusieurs connexions en parallèle)

La première fonction à implanter pour le client consistera à faire la partie **test**. On pourra partir de l'exemple en Section 5. Elle permettra d'envoyer une chaine de caractère à un serveur (dans notre cas l'outil de test miroir) et d'afficher la réponse.

La seconde fonction à implanter sera la fonction **get** et sera à tester sur un site web tel que `www.irit.fr`. La fonction get lorsqu'elle est utilisée avec deux noms de fichiers devra sauvegarder le contenu du fichier téléchargé dans le second nom, mais affichera à l'écran la partie entête renvoyée par le serveur.

La dernière fonction à implanter sera la fonction **put**. La méthode de test de cette fonction sera à expliquer.

4 Serveur en Java

Le serveur en Java sera à tester avec deux outils diffèrent, l'outil développé dans la partie précédentes, mais aussi avec un navigateur web.

Le but du serveur en Java est de pouvoir manipuler les fichiers présents dans un sous-répertoire (par exemple **www-data**). Ainsi si le fichier **bidule.txt** est demandé, le fichier **./www-data/bidule.txt** sera envoyé.

Si un fichier demandé n'est pas présent, il faudra envoyer une réponse de type **404** cohérente

Si le serveur est lancé sur le port **6543** alors pour y accéder en utilisant un navigateur web il faudra aller à l'adresse suivante `http://localhost:6543`

Vous pourrez utiliser les exemples en Section 8 et en Section 9

4.1 Requête GET

- Analyser la requête par exemple avec *split* de la classe *String* pour récupérer la méthode (ex. GET), le nom du fichier (ex. /index.html) et la version du protocole (ex. HTTP/1.0) ;
- Vérifier l'existence et récupérer la taille du fichier en utilisant la classe *File*
- Si le fichier existe :
  - Envoyer le début de la réponse : `HTTP/1.0 200 OK\n`
  - Envoyer l'entête HTTP `Content-length : <taille_fichier>\n`
  - Envoyer une ligne vide `\n`
  - Envoyer le fichier partie par partie on utilisera par exemple les classes FileInputStream ou BufferedInputStream (paquetage java.io).
- Si le fichier n'existe pas : (essayer un telnet www.irit.fr 80 et demandez un fichier inexistant)
  - Envoyer le début de la réponse : `HTTP/1.0 404 Not Found\n`
  - Envoyer une ligne vide `\n`
  - Envoyer une petite page web similaire à celle que renvoie le site web de l'IRIT.

4.2 Requête PUT

- Vérifier qu'on a la requête PUT lors de l'analyse
- Rechercher l'entête Content-length: et récupérer la longueur du fichier
- Sauter toutes les autres entêtes jusqu'à la ligne vide
- Ouvrir le fichier en écriture, et écrire les données en provenance du serveur
- Renvoyer une réponse 201 Created (si le fichier a été créé) et/ou 200 OK (s'il a été modifié).

Pour de plus amples informations, lisez la partie de la RFC 2616 (sur HTTP/1.1) sur la requête PUT (http://rfc.sunsite.dk/rfc/rfc2616.html)

Voici un exemple de requête PUT :

```
C→S  PUT /toto.html HTTP/1.1
C→S  Host: www
C→S  Content-length: 10
C→S
C→S  test 1234
C←S  HTTP/1.1 201 Created
C←S  Date: Mon, 26 Feb 2001 10:44:06 GMT
C←S  Server: Apache/1.3.14 (Win32)
C←S  Transfer-Encoding: chunked
C←S  Content-Type: application/octet-stream
C←S
```

5 Exemple de client C

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg) {
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[]) {
    int portno, sockfd;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char *msg="Message d'exemple";

    portno = 7;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname("localhost");
    if (server == NULL) {
```

```
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr*)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");

    char *buffer=malloc(sizeof(char)*(strlen(msg)+1));

    write(sockfd,msg,strlen(msg));
    read(sockfd,buffer,strlen(msg));

    // we receive an array of characters, so it must be converted
    // to a string by adding a \0 at the end
    buffer[strlen(msg)]='\0';
    printf("%s\n", buffer);

    return 0;
}
```

6 Exemple de server C

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void error(char *msg) {
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer;
     struct sockaddr_in serv_addr, cli_addr;
     int n;

     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     portno = 7000;
     serv_addr.sin_family = AF_INET;
```

```
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);

     newsockfd = accept(sockfd,
                 (struct sockaddr *) &cli_addr,
(socklen_t*) &clilen);
     if (newsockfd < 0)
          error("ERROR on accept");

     do {
       n = read(newsockfd,&buffer,1);
       write(newsockfd,&buffer, 1);
     } while(n>-1);

     return 0;
}
```

7 Annexe HTTP

Le client envoie une requête au serveur. Le format de la requête est le suivant :

Requête	=	Commande "uri" VersionDuProtocole "CR, LF" [Entêtes] <CR, LF> [CorpsDeLaRequête]
VersionDuProtocole	=	HTTP/1.0 | HTTP/1.1 ...

URI (Uniform Resource Identifier) : chaîne identifiant l'objet (par exemple une URL)

principales commandes

GET : demande l'envoi des données identifiées par l'URI.

HEAD : même chose que GET mais retourne seulement les entêtes HTTP sans le corps du document.

POST : permet l'envoi des données d'un formulaire.

PUT : transmet un fichier au serveur.

Les entêtes ont le format :

Mot-clé : Valeur

Par exemple le mot-clé Accept permet de préciser les types de données que sait gérer le navigateur Web, le mot-clé User-Agent permet de préciser le nom et la version du navigateur.

Exemple de requête :

```
GET /index.html HTTP/1.0
Accept : text/html
Accept : text/plain
User-Agent : Lynx/2.4 libwww/2.1.4
```

La réponse envoyée par le serveur ce compose d'une première ligne précisant la version du protocole, un code de réponse et un message précisant la réponse, suivent éventuellement des entêtes (avec le même format que pour la requête), une ligne vide et le corps de la réponse.

Code	Type	Signification
1xx	Informationnel	La commande a été reçue – le traitement est en cours.
2xx	Succès	La commande a été reçue correctement, comprise et acceptée.
3xx	Redirection	La commande ne peut pas être exécutée telle qu'elle il faudra faire autre chose pour la gé
4xx	Erreur client	La commande n'est pas correcte syntaxiquement ou ne peut pas être traitée.
5xx	Erreur serveur	Le serveur n'a pu traiter correctement une commande apparemment correcte.

Exemple de réponse :

```
HTTP/1.0 200 OK
Server : NCSA/1.4.2
MIME-Version : 1.0
Content-type : text/html
Content-length : 97

<HTML>
<HEAD>
<TITLE>
Exemple de document HTML
</TITLE>
<HEAD>
<BODY>
(Corps du document)
</BODY>
</HTML>
```

Pour plus de détail sur HTTP vous pouvez consulter la RFC 1945 qui décrit la spécification de HTTP/1.0 (http://abcdrfc.free.fr/rfc-vf/rfc1945.html).

Pour faire un test vous pouvez utiliser la commande telnet dans un terminal:

```
telnet www.debian.org 80
GET /index.html HTTP/1.0
HOST www.debian.org
```

La ligne vide en fin de commande est nécessaire

8 Exemple de client Java

```
import java.io.*;
import java.net.*;
public class ClientEcho extends Object {
  public static void main (String args[]) {
    String reponse;
    Socket leSocket;
    PrintStream fluxSortieSocket;
    BufferedReader fluxEntreeSocket;

    try {
      // creation d'une socket et connexion à la machine marine sur le port numéro 7
      leSocket = new Socket("marine.edu.ups-tlse.fr", 7);
      System.out.println("Connecté sur : "+leSocket);
      // création d'un flux de type PrintStream lié au flux de sortie de la socket
```

```
      fluxSortieSocket = new PrintStream(leSocket.getOutputStream());
      // creation d'un flux de type BufferedReader lié au flux d'entrée de la socket
      fluxEntreeSocket = new BufferedReader(new InputStreamReader(leSocket.getInputStream()));
      // envoi de données vers le serveur
      fluxSortieSocket.println("Bonjour le monde!");
      // attente puis réception de données envoyées par le serveur
      reponse = fluxEntreeSocket.readLine();
      System.out.println("Reponse du serveur : " + reponse);
      leSocket.close();
    } // try
    catch (UnknownHostException ex)
    {
      System.err.println("Machine inconnue : "+ex);
      ex.printStackTrace();
    }
    catch (IOException ex)
    {
      System.err.println("Erreur : "+ex);
      ex.printStackTrace();
    }
  } // main
} // class
```

9 Exemple de serveur Java

```
import java.io.*;
import java.net.*;
public class ServeurEcho extends Object {
  public static void main (String args[]) {
    ServerSocket socketEcoute;
    Socket socketService;
    InputStream entreeSocket;
    OutputStream sortieSocket;
    try {
      // création du socket d'écoute (port numéro 7)
      socketEcoute = new ServerSocket(7);
      while (true) {
        // attente d'une demande de connexion
        socketService = socketEcoute.accept();
        System.out.println("Nouvelle connexion : " + socketService);
        // récupération des flux d'entrée/sortie de la socket de service
        entreeSocket = socketService.getInputStream();
        sortieSocket = socketService.getOutputStream();
        try {
          int b = 0;
          while (b != -1) {
            b = entreeSocket.read();
            sortieSocket.write(b);
          } // while
          System.out.println("Fin de connexion");
        } // try
        catch (IOException ex)
        {
```

```
          // fin de connexion
          System.out.println("Fin de connexion : "+ex);
          ex.printStackTrace();
        }
        socketService.close();
      } // while (true)
    } // try
    catch (Exception ex)
    {
      // erreur de connexion
      System.err.println("Une erreur est survenue : "+ex);
      ex.printStackTrace();
    }
  } // main
} // class
```