

Compte-rendu de TP

ARAR – HTTP 1.1.

BENALI Myriam, BESSON Cécile, DELPLANQUE Thibaut,
NAAJI Dorian INFO 3A Groupe 2
POLYTECH LYON

TABLE DES MATIERES

1. ENJEUX DU TP	3
1.1. REALISATION D'UN CLIENT/SERVEUR HTTP 1.1.....	3
1.2. FONCTIONNALITES D'ECRITURE ET DE LECTURE	3
2. FONCTIONNEMENT INTERNE DU LOGICIEL	4
1.1. SCENARIO DE LECTURE ET D'ECRITURE	4
1.2. LA LECTURE, ALGORITHME CLIENT : « GET ».....	5
1.3. L'ECRITURE, ALGORITHME CLIENT : « PUT »	9
1.4. ALGORITHMES DU SERVEUR, TRAITEMENT DES REQUETES	12
3. RESULTATS ET APPRENTISSAGES.....	14
3.1. ENSEIGNEMENTS SUR LA PROGRAMMATION RESEAU	15
3.2. DIFFICULTES RENCONTREES.....	15
4. MANUEL UTILISATEUR.....	16
4.1. REQUETE GET :	16
4.1.1. ➔ Si nous voulons récupérer un fichier d'une autre machine	16
4.1.2. Exemple d'une requête GET afin de récupérer une image d'une machine hébergeant le serveur :	16
4.1.3. ➔ Si nous voulons récupérer un fichier d'un navigateur :	18
4.1.4. Exemple d'une requête GET afin de récupérer une page WEB :.	19
4.2. REQUETE PUT :	20
4.2.1. Exemple d'une requête PUT afin d'envoyer un fichier au serveur distant :	21
4.3. VALEURS DE RETOUR DE LA METHODE GET ET DE LA METHODE PUT (ÉVENTUELLES ERREUR DU PROGRAMME) :	21

1. ENJEUX DU TP

1.1. REALISATION D'UN CLIENT/SERVEUR HTTP 1.1.

Ce TP a consisté en la conception et la réalisation d'un client/serveur WEB reposant sur le protocole HTTP 1.1., basé sur la norme [RFC 2616](#). HTTP 1.1. se base sur le protocole de transport TCP/IP. Pour utiliser ce protocole, nous avons utilisé les sockets Inet, notamment via l'interface de programmation que propose JAVA, grâce à un ensemble de méthodes et d'objets permettant la communication TCP/IP entre plusieurs applications.

1.2. FONCTIONNALITES D'ECRITURE ET DE LECTURE

HTTP 1.1. propose différentes méthodes, encapsulées dans des « requêtes ». Le logiciel envoie ces requêtes côté clients et les traite et y répond côté serveur. Ce dernier propose en effet deux fonctionnalités :

- La lecture d'un fichier contenu sur le serveur, via la méthode HTTP « **GET** ». Le client va en effet pouvoir récupérer un fichier stocké sur le serveur, en envoyant une requête à ce dernier. Le serveur va ensuite répondre au client en donnant le fichier spécifié.
- L'écriture d'un fichier appartenant au client sur le serveur, via la méthode HTTP « **PUT** ». Le client peut placer un fichier qu'il détient en local sur le serveur, par le biais d'une requête qui sera envoyée au serveur puis traitée.

Ce logiciel est divisé en deux parties distinctes, indépendantes l'une de l'autre. Chacune des parties communique grâce au protocole HTTP 1.1. et TCP/IP :

D'un côté, le **serveur**, qui est éveillé en permanence et est en écoute permanente sur le port TCP 1026. Chaque client va pouvoir établir une connexion avec le serveur sur ce même port puis se verra attribuer un port de communication spécifique, réservé au client. La connexion est ensuite fermée sur demande du client. Pour résumer, le serveur fonctionne en **mode concurrent**.

De l'autre côté, le **client**, qui va pouvoir **communiquer avec des serveurs HTTP 1.1. quelconques**, que ce soit le serveur que nous avons programmé ou d'autres serveurs WEB, via une URL.

2. FONCTIONNEMENT INTERNE DU LOGICIEL

1.1. SCENARIO DE LECTURE ET D'ECRITURE

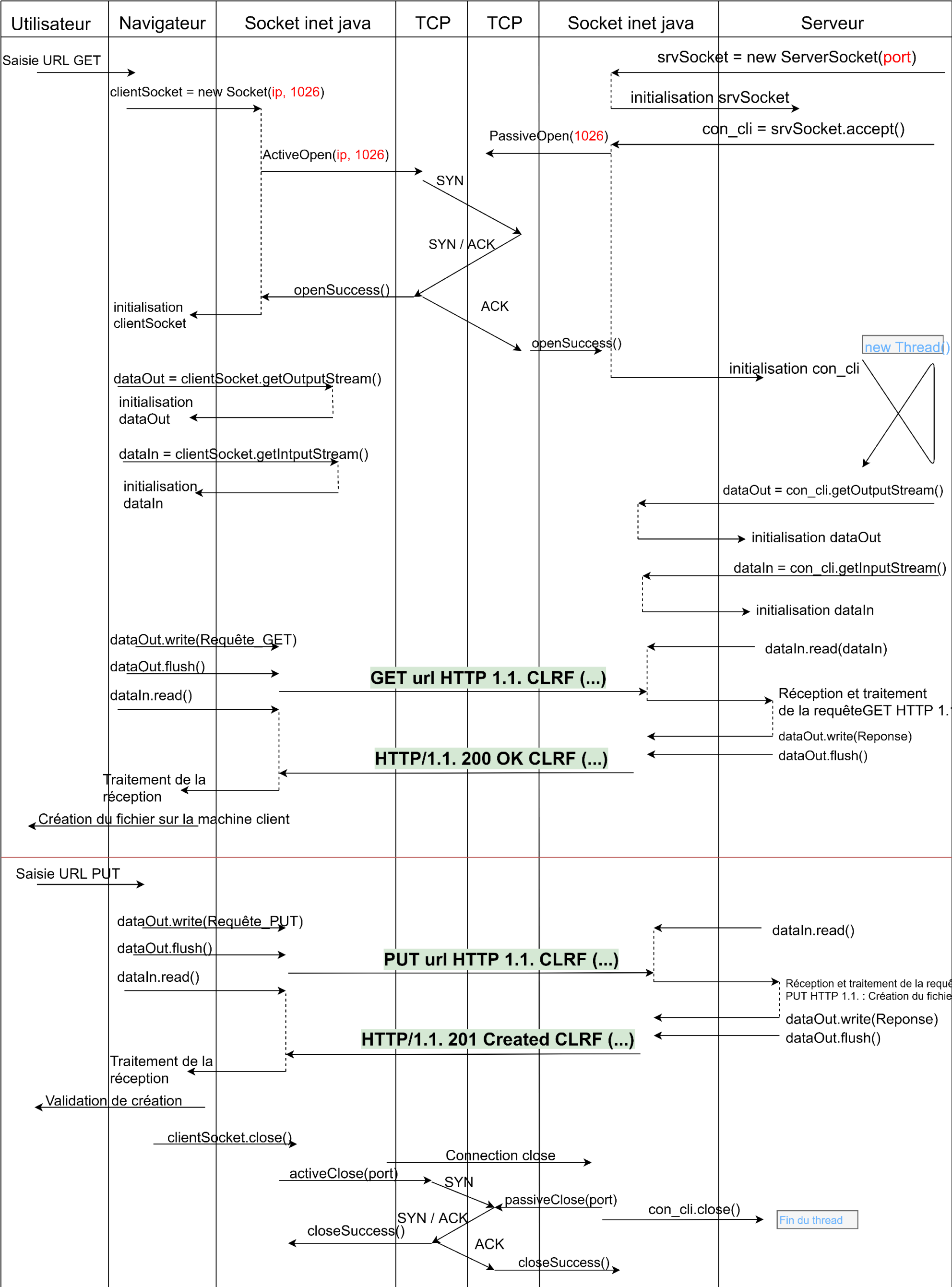


FIGURE 1 : CE DIAGRAMME MONTRE L'ENCHAINEMENT DES DIFFERENTES OPERATIONS, COTE SERVEUR ET COTE CLIENT APRES L'ENVOI D'UNE REQUETE GET PUIS PUT PAR LE CLIENT.

1.2. LA LECTURE, ALGORITHME CLIENT : « GET »

L'algorithme ci-après permet la récupération d'un fichier, côté client, par l'envoi d'une requête GET à un serveur et un port spécifié en paramètre.

```
2.  /**
3.   *
4.   * @param ipServer
5.   * @param port
6.   * @param fileName
7.   * @return int :
8.   * 0 : Tout s'est bien passé.
9.   * -1 : Problème de flux
10.  * -2 : erreur de fermeture de socket
11.  * -3 : Erreur lors de l'ouverture du socket (Methode OpenSocket())
12.  * -5 : Impossible de lire/ouvrir le fichier
13.  * -6 : Impossible d'écrire le fichier spécifié dans un flux
14.  * -7 : Impossible d'écrire le fichier local
15.  * -8 : Erreur HTTP
16.  */
17. public static int GET(String ipServer, int port, String fileName)
18. {
19.     byte[] fileInBytes = null;
20.     String fileInString = null;
21.     boolean isImg = strUtils.isImg(fileName);
22.     // ouverture du socket client
23.     Socket sock = OpenSocket(ipServer, port);
24.     try
25.     {
26.         if (sock == null)
27.         {
28.             // -3 en cas d'erreur d'ouverture
29.             System.out.println("Erreur lors de l'ouverture du socket (Methode
30.                               OpenSocket())");
31.             return -3;
32.         }
33.     } catch (NullPointerException ex)
34.     {
35.         System.out.println("Exception : " + ex);
36.         return -3;
37.     }
38.     try
39.     {
40.         // outputStream contient les données pouvant être envoyées côté
41.         // serveur
42.         OutputStream outputStream = sock.getOutputStream();
43.         // indata représente l'objet InputStream et l'objet inputStream permet
44.         // d'utiliser la zone de stockage (buffer).
45.         InputStream indata = sock.getInputStream();
46.         // un bufferedInputStream permet des échanges plus optimisés
47.         BufferedInputStream inputStream = new BufferedInputStream(indata);
48.         // String url = "http://" + ipServer + ":" + port;
49.         String httpGetRequest = "GET " + fileName + " HTTP/1.1\r\n";
50.         httpGetRequest += "Host: " + ipServer + "\r\n\r\n";
```

```

50. //httpGETRequest += "Accept : */*\r\n\r\n";
51.
52. // écriture et envoi
53. outputStream.write(httpGETRequest.getBytes());
54. outputStream.flush();
55.
56. //On récupère les données
57. FileOutputStream fileb;
58. if (isImg)
59. {
60.     String extension = strUtils.getFileExtension(fileName);
61.     fileb = new FileOutputStream("reception." + extension);
62. }
63. else
64. {
65.     fileb = new FileOutputStream("reception.html");
66. }
67. int byteCourant;
68. int count = 0;
69. String responseCode = "";
70. boolean headerEnded = false;
71. try
72. {
73.     // tant que des données sont envoyées par le serveur
74.     while ((inputStream.available()) != 0)
75.     {
76.         // on récupère le byte courant
77.         byteCourant = inputStream.read();
78.         // on récupère le code HTTP de la réponse du serveur qui est
79.         // toujours codé du 9ème au 11ème octet
80.         if (count == 9 || count == 10 || count == 11)
81.         {
82.             responseCode += (char) byteCourant;
83.         }
84.         // si on n'est pas encore sorti du header, on n'a pas eu
85.         // CRLF dans la réponse du serveur, on teste les bytes
86.         if (!headerEnded)
87.         {
88.             System.out.print((char) byteCourant);
89.             if (byteCourant == 13)
90.             {
91.                 int byteSuivant0 = inputStream.read();
92.                 if (byteSuivant0 == 10)
93.                 {
94.                     int byteSuivant1 = inputStream.read();
95.                     if (byteSuivant1 == 13)
96.                     {
97.                         int byteSuivant2 = inputStream.read();
98.                         if (byteSuivant2 == 10)
99.                         {
100.                            // on a bien trouvé le premier CRLF
101.                            headerEnded = true;
102.                        }
103.                    }
104.                }
105.            }
106.        }
107.    }
108. }
109. catch (IOException e)
110. {
111.     e.printStackTrace();
112. }
113. }

```

```

106.          // Le header a bien déjà été rencontré, on a affaire à des
données brutes.
107.          else if (headerEnded)
108.          {
109.              fileb.write(byteCourant);
110.          }
111.          count++;
112.      }
113.      indata.close();
114.      fileb.close();
115.      int responseInt = Integer.parseInt(responseCode);
116.      System.out.println(responseCode);
117.      if (responseInt != 200 || responseInt != 201)
118.      {
119.          // code d'erreur
120.          _errCode = responseInt;
121.          return -8;
122.      }
123.  }
124.  catch (Exception e)
125.  {
126.      System.out.println(e);
127.  }
128.  }
129.  catch (IOException ex)
130.  {
131.      // -1 en cas de problème de flux
132.      System.out.println("Erreur lors de l'utilisation d'un flux "
133.          + "sortant.\nRapport d'exception : " + ex);
134.      try
135.      {
136.          sock.close();
137.      }
138.      catch (IOException exSock)
139.      {
140.          // -2 en cas d'erreur de fermeture de socket
141.          System.out.println("Erreur lors de la fermeture du Socket. "
142.              + "\nIP serveur : " + ipServer + "\nPort serveur : "
143.              + port + " \nRapport d'exception complet : " + exSock);
144.          return -2;
145.      }
146.      return -1;
147.  }
148.  // Fermeture de la connexion si tout s'est bien passé
149.  try
150.  {
151.      sock.close();
152.  }
153.  catch (IOException ex)
154.  {
155.      // -2 en cas d'erreur de fermeture de socket
156.      System.out.println("Erreur lors de la fermeture du Socket. "
157.          + "\nIP serveur : " + ipServer + "\nPort serveur : "
158.          + port + " \nRapport d'exception complet : " + ex);
159.      return -2;
160.  }
161.  // Le GET s'est bien passé.

```

```
162.         return 0;
163.     }
```


1.3. L'ECRITURE, ALGORITHME CLIENT : « PUT »

L'algorithme suivant permet l'écriture d'un fichier, situé chez le client, sur la machine serveur spécifiée (IP et port)

```
2.  /**
3.   *
4.   * @param ipServer
5.   * @param port
6.   * @param fileName
7.   * @param localFilePath
8.   * @return int :
9.   * 0 : Tout s'est bien passé.
10.  * -1 : Problème de flux
11.  * -2 : erreur de fermeture de socket
12.  * -3 : Erreur lors de l'ouverture du socket (Methode OpenSocket())
13.  * -5 : Impossible de lire/ouvrir le fichier
14.  * -6 : Impossible d'écrire le fichier spécifié dans un flux
15.  */
16. public static int PUT(String ipServer, int port, String fileName, String
    localFilePath)
17. {
18.     byte[] fileInBytes = null;
19.     String fileInString = null;
20.     boolean isImg = strUtils.isImg(fileName);
21.
22.     if (isImg)
23.     {
24.         try
25.         {
26.             File file = new File(localFilePath);
27.             fileInBytes = readFileToByteArray(file);
28.         }
29.         catch (Exception e)
30.         {
31.             // impossible d'ouvrir / lire le fichier
32.             System.out.println("Exception : Impossible de lire/ouvrir le
                fichier.\n" + e);
33.             return -5;
34.         }
35.     }
36.     else
37.     {
38.         try
39.         {
40.             fileInString = strUtils.readFileAsString(localFilePath);
41.         }
42.         catch (IOException ex)
43.         {
44.             // impossible d'ouvrir / lire le fichier
45.             System.out.println("Exception : Impossible de lire/ouvrir le
                fichier.\n" + ex);
46.             return -5;
47.         }
48.     }
```

```

49. // ouverture du socket
50. Socket sock = OpenSocket(ipServer, port);
51. try
52. {
53.     if (sock == null)
54.     {
55.         // -3 en cas d'erreur d'ouverture
56.         System.out.println("Erreur lors de l'ouverture du socket
(Methode OpenSocket())");
57.         return -3;
58.     }
59. }
60. catch (NullPointerException ex)
61. {
62.     System.out.println("Exception : " + ex);
63.     return -3;
64. }
65.
66. try
67. {
68.     OutputStream outputStream = sock.getOutputStream();
69.     String url = "http://" + ipServer + ":" + port;
70.     String httpPUTRequest = "PUT /" + fileName + " HTTP/1.1 \r\n";
71.     httpPUTRequest += "Host: " + ipServer + "\r\n";
72.     if (isImg && fileInBytes != null)
73.     {
74.         // (+1 car on compte le retour chariot après le header
Content-length)
75.         int length = fileInBytes.length + 1;
76.         httpPUTRequest += "Content-length: " + length + "\r\n\r\n";
77.         for (int i = 0; i < fileInBytes.length; i++)
78.         {
79.             httpPUTRequest += fileInBytes[i];
80.         }
81.     }
82.     else if (!isImg && fileInString != null)
83.     {
84.         int length = fileInString.length() + 1;
85.         httpPUTRequest += "Content-length: " + length + "\r\n\r\n";
86.         httpPUTRequest += fileInString;
87.     }
88.     else
89.     {
90.         System.out.println("Exception : Impossible d'écrire le
fichier"
91.             + "spécifié dans un flux.");
92.         sock.close();
93.         return -6;
94.     }
95.
96. // écriture et envoi
97. System.out.println(httpPUTRequest);
98. outputStream.write(httpPUTRequest.getBytes());
99. outputStream.flush();
100. }
101. catch (IOException ex)
102. {

```

```

103.         // -1 en cas de problème de flux
104.         System.out.println("Erreur lors d'utilisation d'un flux "
105.             + "sortant.\nRapport d'exception : " + ex);
106.         try
107.         {
108.             sock.close();
109.         }
110.         catch (IOException exSock)
111.         {
112.             // -2 en cas d'erreur de fermeture de socket
113.             System.out.println("Erreur lors de la fermeture du Socket."
114.                 + "\nIP serveur : " + ipServer + "\nPort serveur : "
115.                 + port + "\nRapport d'exception complet: "+exSock);
116.             return -2;
117.         }
118.         return -1;
119.     }
120.     // Fermeture de la connexion si tout s'est bien passé
121.     try
122.     {
123.         sock.close();
124.     }
125.     catch (IOException ex)
126.     {
127.         // -2 en cas d'erreur de fermeture de socket
128.         System.out.println("Erreur lors de la fermeture du Socket. "
129.             + "\nIP serveur : " + ipServer + "\nPort serveur : "
130.             + port + " \nRapport d'exception complet : " + ex);
131.         return -2;
132.     }
133.     // Le PUT s'est bien passé. Le fichier a été écrit sur le serveur.
134.     return 0;
135. }

```

1.4. ALGORITHMES DU SERVEUR, TRAITEMENT DES REQUETES

L'algorithme suivant permet au serveur de lire un fichier, à la suite de la réception d'une requête de type GET.

```
2.  /**
3.   * La méthode ReadFile permet la lecture d'un fichier
4.   *
5.   * @param filePath
6.   */
7.  private void readFile(String filePath)
8.  {
9.
10.     try
11.     {
12.         FileInputStream fileInputStream = new FileInputStream(WWW_PATH +
filePath);
13.         BufferedReader reader = new BufferedReader(new
InputStreamReader(fileInputStream));
14.         StringBuilder result = new StringBuilder();
15.         String line;
16.         try
17.         {
18.             while ((line = reader.readLine()) != null)
19.             {
20.                 result.append(line + "\r\n");
21.             }
22.             reader.close();
23.             result.append("\r\n");
24.         }
25.         catch (IOException e)
26.         {
27.             _body = null;
28.         }
29.         _body = result.toString();
30.     }
31.     catch (FileNotFoundException e)
32.     {
33.         _body = "Error 404";
34.         _code = "404";
35.         _codeMessage = "resource not found";
36.     }
37. }
```

Cet algorithme permet cette fois-ci au serveur d'exécuter le traitement à la suite de la réception d'une requête « PUT » : le serveur va créer le fichier.

```
1. /**
2.  * Permet l'écriture d'un fichier après réception PUT
3.  * @param filePath le chemin vers le fichier à créer
4.  */
5. private void writeFile(String filePath)
6. {
7.     try
8.     {
9.         File file = new File(WWW_PATH + filePath);
10.        file.createNewFile();
11.        FileOutputStream fileOutputStream = new FileOutputStream(file, false);
12.
13.        fileOutputStream.write(_body.getBytes());
14.        fileOutputStream.close();
15.        _contentLocation = filePath;
16.        _code = "201";
17.        _codeMessage = "Created";
18.    }
19.    catch (FileNotFoundException e)
20.    {
21.        {
22.            _body = "Error 404";
23.            _code = "404";
24.            _codeMessage = "resource not found";
25.        }
26.    catch (IOException e)
27.    {
28.        e.printStackTrace();
29.    }
30. }
```

Finalement, ce dernier algorithme permet au serveur de répondre au client :

```
1.  /**
2.   * Permet de répondre au client
3.   */
4.  public void sendAnwser()
5.  {
6.      if (_command.equalsIgnoreCase("GET"))
7.      {
8.          _out.write(_version + " " + _code + " " + _codeMessage + "\r\n");
9.          _out.write("Content-Length: " + _contentLength + "\r\n");
10.         _out.write("Content-Type: " + _contentType + "\r\n");
11.         _out.write("\r\n");
12.         _out.write(_body + "\r\n");
13.         _out.write("\r\n");
14.     }
15.     else if (_command.equalsIgnoreCase("PUT"))
16.     {
17.         _out.write(_version + " " + _code + " " + _codeMessage + "\r\n");
18.         _out.write("Content-location: " + _contentLocation + "\r\n");
19.         _out.write("\r\n");
20.     }
21.     else
22.     {
23.         // traitement éventuel d'autres requêtes HTTP
24.     }
25.     _out.flush();
26. }
27.
```

3. RESULTATS ET APPRENTISSAGES

3.1. ENSEIGNEMENTS SUR LA PROGRAMMATION RESEAU

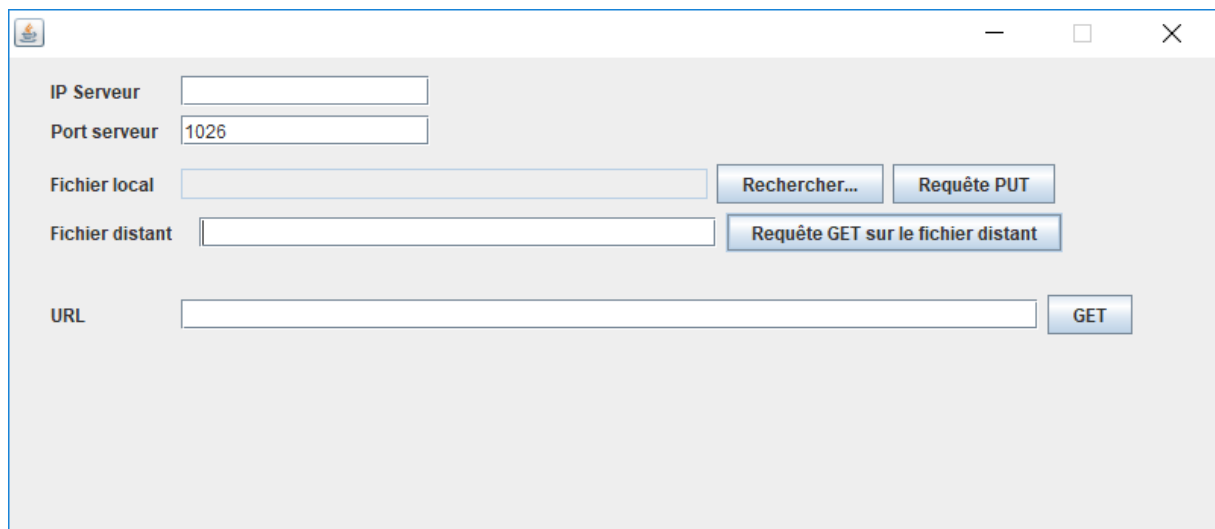
Ce TP sur la création d'un client/serveur reposant sur HTTP 1.1. nous a permis de confirmer nos connaissances acquises sur le protocole de transport TCP/IP et d'expérimenter la programmation réseaux JAVA via notamment les Sockets Inet. Il nous a permis de comprendre les rouages du fonctionnement des serveurs WEB reposants sur HTTP 1.1. et pose des bases pour nos apprentissages futurs en matière de technologies WEB.

3.2. DIFFICULTES RENCONTREES

La syntaxe HTTP 1.1. (RFC 2616) demande une rigueur de taille car sensible à la casse et également aux retours chariots et sauts de lignes (CRLF pour Carriage Return, Line Feed). De ce fait, nombreux ont été les blocages en raison de cela. Ce TP nous a permis d'apprendre à maîtriser cette syntaxe et ce protocole.

4. MANUEL UTILISATEUR

Notre interface ressemble à ceci :



The screenshot shows a web application interface with the following elements:

- IP Serveur**: A text input field.
- Port serveur**: A text input field containing the value "1026".
- Fichier local**: A text input field.
- Fichier distant**: A text input field.
- URL**: A text input field.
- Buttons**:
 - Rechercher...**: A button next to the "Fichier local" field.
 - Requête PUT**: A button next to the "Fichier distant" field.
 - Requête GET sur le fichier distant**: A button below the "Fichier distant" field.
 - GET**: A button next to the "URL" field.

Pour exécuter les requêtes, il faut, tout d'abord, lancer le serveur, puis lancer le client.









4.1. REQUETE GET :

4.1.1. ➔ Si nous voulons récupérer un fichier d'une autre machine

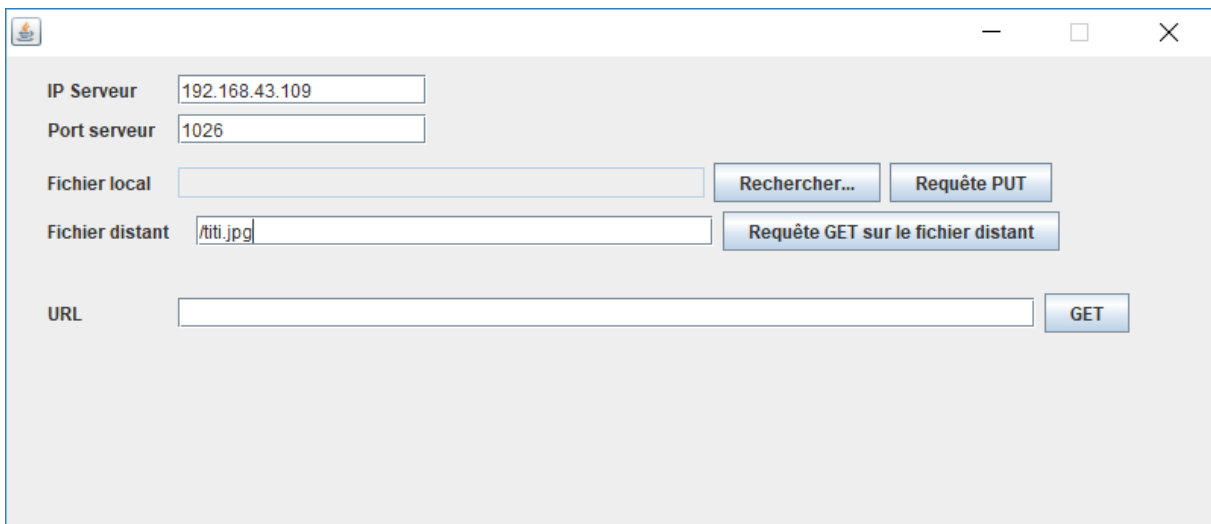
Les champs à renseigner du côté du client sont « **IP Serveur** » (l'adresse IP de la machine hébergeant le serveur), « **Port serveur** » et « **Fichier distant** » (le nom du fichier distant, fichier que l'on veut récupérer chez le serveur). Une fois ces champs remplis, le client recevra le fichier distant récupéré chez le serveur. Le fichier reçu s'appellera « Reception » et se trouvera à la racine du projet chez le client.

4.1.2. Exemple d'une requête GET afin de récupérer une image d'une machine hébergeant le serveur :

L'image que l'on veut récupérer se trouve dans le dossier « www » dans le disque C (C:\www) chez le serveur :

> Ce PC > Windows (C:) > Utilisateurs > Myriam > eclipse-workspace > ClientServerHTTP			
> Ce PC > Windows (C:) > www			
Nom	Modifié le	Type	Taille
 titi.jpg	30/05/2019 19:46	Fichier JPG	31 Ko
 aoc	23/05/2019 11:06	Dossier de fichiers	
 src	30/05/2019 18:45	Dossier de fichiers	
 .classpath	22/05/2019 19:18	Fichier CLASSPATH	1 Ko
 .gitignore	30/05/2019 18:45	Document texte	1 Ko
 .project	22/05/2019 19:18	Fichier PROJECT	1 Ko
 README.md	22/05/2019 19:21	Markdown file	1 Ko
 titi.jpg	30/05/2019 19:46	Fichier JPG	31 Ko

Du côté du client, nous remplissons les 3 champs nécessaires (« IP Serveur », « Port Serveur » et « Fichier distant ») :



The screenshot shows a client application window with the following fields and buttons:

- IP Serveur**: Text field containing "192.168.43.109"
- Port serveur**: Text field containing "1026"
- Fichier local**: Text field (empty), with a **Rechercher...** button to its right.
- Fichier distant**: Text field containing "/titi.jpg", with a **Requête GET sur le fichier distant** button to its right.
- URL**: Text field (empty), with a **GET** button to its right.

Nous cliquons ensuite sur « Requête GET sur le fichier distant » et le client reçoit bien l'image :

En ouvrant l'image « titi.jpg », nous obtenons bien l'image voulue :

4.1.3. ➔ Si nous voulons récupérer un fichier d'un navigateur :

Seul le champ « **URL** » sera à renseigner du côté du client. Une fois ce champ rempli, nous pourrons cliquer sur « **GET** » pour récupérer le fichier. Le fichier reçu s'appellera « Reception » et se trouvera à la racine du projet.



4.1.4. Exemple d'une requête GET afin de récupérer une page WEB :



IP Serveur

Port serveur

Fichier local

Fichier distant

URL

Le client a bien reçu la page à la racine du projet :

📁 > Ce PC > Windows (C:) > Utilisateurs > Myriam > eclipse-workspace > ClientServerHTTP

Nom	Modifié le	Type	Taille
📁 .settings	22/05/2019 19:18	Dossier de fichiers	
📁 bin	30/05/2019 18:08	Dossier de fichiers	
📁 doc	23/05/2019 11:06	Dossier de fichiers	
📁 src	30/05/2019 18:45	Dossier de fichiers	
📄 .classpath	22/05/2019 19:18	Fichier CLASSPATH	1 Ko
📄 .gitignore	30/05/2019 18:45	Document texte	1 Ko
📄 .project	22/05/2019 19:18	Fichier PROJECT	1 Ko
📄 README.md	22/05/2019 19:21	Markdown file	1 Ko
📄 reception.html	30/05/2019 19:29	Fichier HTML	56 Ko

En ouvrant « reception.html » grâce à un navigateur, nous obtenons bien la page voulue :



4.2. REQUETE PUT :

Le serveur doit, au préalable, créer un dossier « **www** » dans le disque C (C:\www). Les champs à renseigner du côté du client sont « **IP Serveur** » (l'adresse IP de la machine hébergeant le serveur), « **Port Serveur** » et « **Fichier local** » (le nom du fichier que l'on souhaite envoyer au serveur, grâce au bouton Rechercher, nous pouvons directement accéder aux documents de notre PC et nous pouvons alors choisir un fichier). Une fois les 3 champs remplis, nous pouvons cliquer sur « **Requête PUT** » pour envoyer le fichier au serveur.

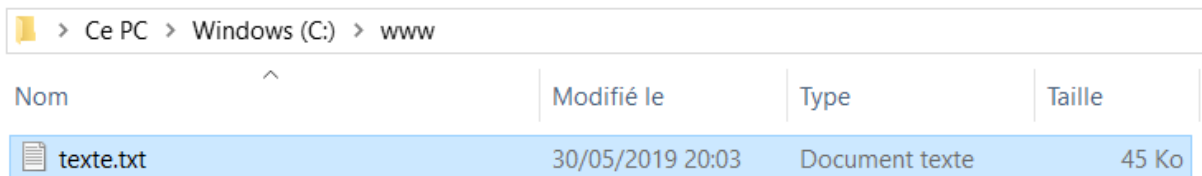
4.2.1. Exemple d'une requête PUT afin d'envoyer un fichier au serveur distant :

Du côté du client, nous remplissons les 3 champs nécessaires (« IP Serveur », « Port serveur » et « Fichier local ») :



IP Serveur: 192.168.43.109
Port serveur: 1026
Fichier local: C:\Users\Myriam\Desktop\texte.txt
Fichier distant:
URL:
Buttons: Rechercher..., Requête PUT, Requête GET sur le fichier distant, GET

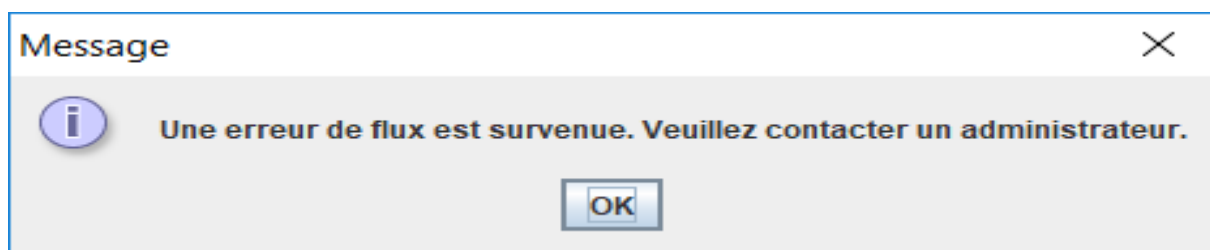
Nous cliquons ensuite sur « Requête PUT » et le serveur reçoit bien le fichier voulu dans le dossier « www » (C:\www) :



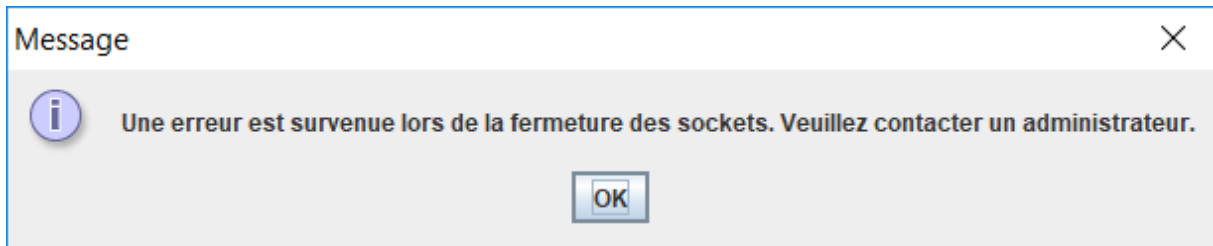
Ce PC > Windows (C:) > www			
Nom	Modifié le	Type	Taille
texte.txt	30/05/2019 20:03	Document texte	45 Ko

4.3. VALEURS DE RETOUR DE LA METHODE GET ET DE LA METHODE PUT (EVENTUELLES ERREURS DU PROGRAMME) :

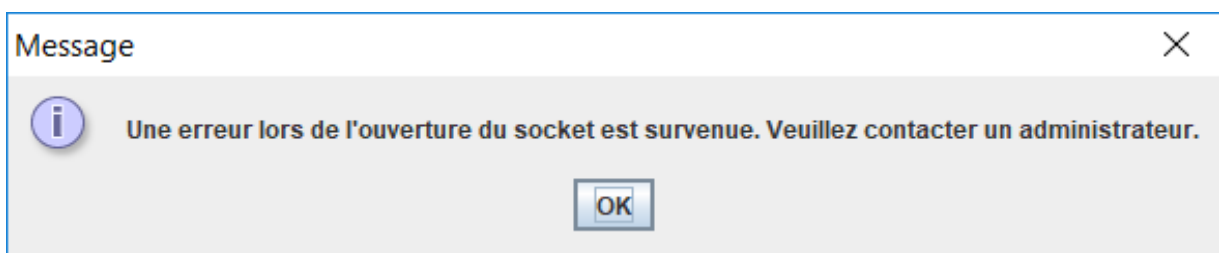
Si la valeur de retour est égale à -1 : il y a eu une erreur de flux (IOException)



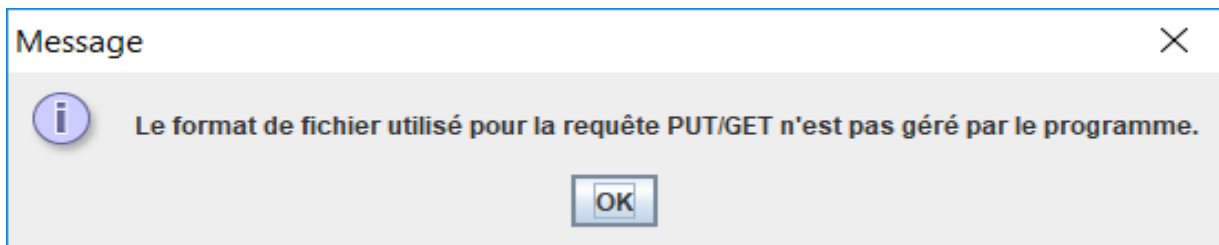
Si la valeur de retour est égale à -2 : il y a eu une erreur lors de la fermeture des sockets (IOException)



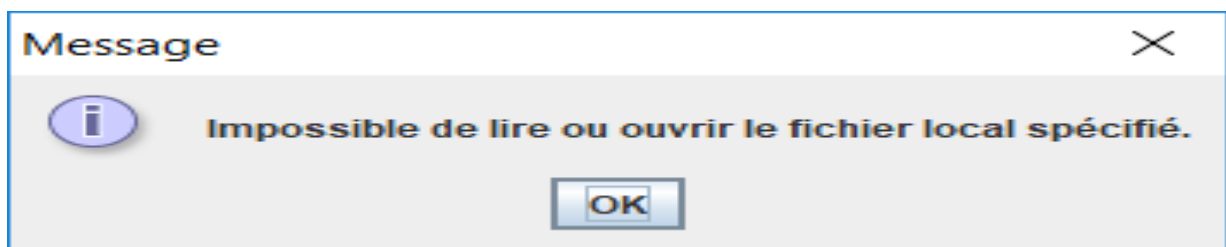
Si la valeur de retour est égale à -3 : il y a eu une erreur lors de l'ouverture du socket (NullPointerException)



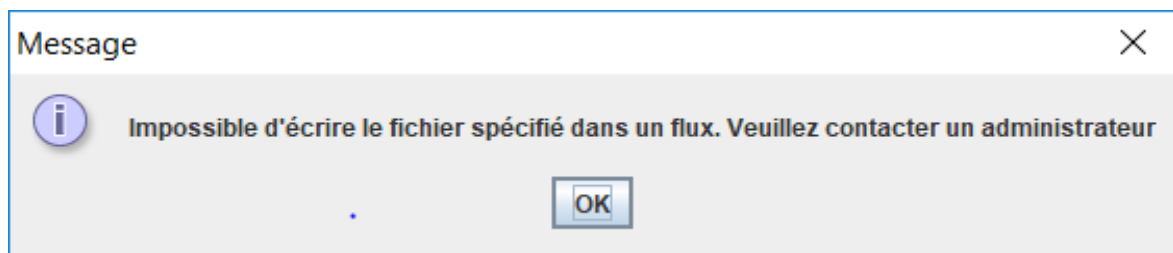
Si la valeur de retour est égale à -4 : le format de fichier utilisé pour la requête PUT ou GET n'est pas géré par le programme



Si la valeur de retour est égale à -5 : il est impossible de lire ou d'ouvrir le fichier local spécifié (Exception)



Si la valeur de retour est égale à -6 : il est impossible d'écrire le fichier spécifié dans un flux



Si la valeur de retour est égale à -8 : le serveur a retourné une erreur

