SYSTEMES D'INFORMATION 2

TP 2 - HTTP

Jean-Christophe ROLDAN & Samuel MERTENAT

Télécommunications
T3a & T3f

Fribourg, le 12 octobre 2015

TABLE DES MATIERES

SYSTEMES D'INFORMATION 2

TP 2 - HTTP

1 INTRODUCTION

Dans le cadre de ce deuxième travail pratique, nous sommes amenés à implémenter l'utilisation d'une communication HTTP d'un petit client Java, qui nous permettra par la suite de réaliser des requêtes de type GET, POST, OPTION ou TRACE.

2 QUESTIONS

Ce laboratoire a été réalisé à l'aide d'un ordinateur portable « APPLE » :

- Mac OS X EL Capitan, version 10.11
- Eclipse Mars (4.5.0)
- Java 1.8.0_60

2.1 P1

Implémenter et tester la classe `HTTPClient` (démontrer le bon fonctionnement au professeur) avec les méthodes GET, OPTION et TRACE.

Le code source de la classe « HTTPClient » se trouve en annexe. Afin de vérifier le bon fonctionnement de la classe, nous utilisons les méthodes « GET », « OPTION » et « TRACE ».

En effectuant un « GET » à l'adresse « rudolf.scheurer.home.hefr.ch », nous obtenons :

Status :

```
HTTP/1.1 200 OK
```

Content :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    …
</head>
  <body>
    <div id="site">
      <div id="header">
        …
      </div>
    </div>
  </body>
</html>
```

En effectuant ensuite un « TRACE » à l'adresse « rudolf.scheurer.home.hefr.ch », nous récupérons :

Content :

```
TRACE / HTTP/1.1
User-Agent: Java/1.8.0_60
Host: rudolf.scheurer.home.hefr.ch
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
X-Forwarded-For: 160.98.113.13
```

Pour finir, nous testons la méthode « OPTION » sur l'adresse « switch.ch ». Nous pouvons constater que les méthodes « OPTIONS », « GET », « HEAD » et « POST » sont autorisées.

Content :

```
Date: Mon, 05 Oct 2015 08:14:56 GMT
Server: Apache
Allow: OPTIONS,GET,HEAD,POST
Cache-Control: max-age=0
Expires: Mon, 05 Oct 2015 08:14:56 GMT
Vary: Accept-Encoding
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

2.2 P2

A quel moment (c.-à-d. à l'exécution de quelle méthode Java) la connexion TCP est-elle établie ? À quel moment (c.-à-d. à l'exécution de quelle méthode Java) la requête HTTP est-elle envoyée ?

Etablissement de la connexion TCP : conn.connect()

Documentation de la méthode « connect() » :

```
Opens a communications link to the resource referenced by this URL, if such a
connection has not already been established.
If the connect method is called when the connection has already been opened (indicated
by the connected field having the value true), the call is ignored.
URLConnection objects go through two phases: first they are created, then they are
connected. After being created, and before being connected, various options can be
specified (e.g., doInput and UseCaches). After connecting, it is an error to try to set
them. Operations that depend on being connected, like getContentLength, will implicitly
perform the connection, if necessary.
```

Envoie de la requête http : conn.getHeaderField(0)

Documentation de la méthode « getHeaderField() » :

```
Returns the value for the nth header field. Some implementations may treat the 0th
header field as special, i.e. as the status line returned by the HTTP server.
This method can be used in conjunction with the getHeaderFieldKey method to iterate
through all the headers in the message.
```

Source : https://docs.oracle.com/javase/8/docs/api/

2.3 P3

Faire une requête HTTP vers l'URL « tlabs.tic.eia-fr.ch/tinf/nodir » (sans le slash à la fin !), une fois avec le `HTTPClient`, une fois avec un navigateur normal. Documenter les échanges (diagramme en flèche niveau HTTP) et commenter les différences.

Nous débutons par réaliser une requête à l'aide du client HTTP, puis à l'aide du navigateur Safari.

Filter: http — Expression... Clear Apply Save

N	Time	Source	Destination	Protocol	Src port	Dst port	Length	Info
37	3.865227000	fe80::e5f6:886c	ff02::c	SSDP	1900	1900	514	NOTIFY * HTTP/1.1
38	3.865855000	fe80::e5f6:886c	ff02::c	SSDP	1900	1900	571	NOTIFY * HTTP/1.1
46	4.214349000	160.98.113.13	160.98.31.32	HTTP	50909	80	234	GET /tinf/nodir HTTP/1.1
48	4.217271000	160.98.31.32	160.98.113.13	HTTP	80	50909	865	HTTP/1.1 301 Moved Permanently (text/html)
56	4.220813000	160.98.113.13	160.98.31.32	HTTP	50910	80	235	GET /tinf/nodir/ HTTP/1.1
58	4.223861000	160.98.31.32	160.98.113.13	HTTP	80	50910	681	HTTP/1.1 200 OK (text/html)

Figure 1: Trames échangées lors de l'utilisation du client HTTP

Client HTTP :

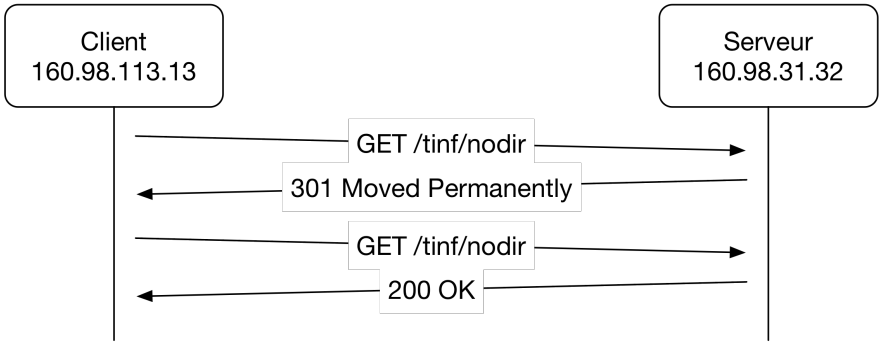

Figure 2: Diagramme en flèches de l'échange

Avec l'utilisation du client ou du browser, nous obtenons les 4 mêmes premières trames. L'idée est la suivante : le client demande la ressource « /tinf/nodir » et reçoit comme réponse du serveur un message de redirection « 301 Moved Permanently ». Le client fait alors une nouvelle requête demandant cette fois-ci la ressource au nouvel emplacement, pour finalement recevoir un « 200 OK » du serveur.

À partir de ce point, nous avons la différence suivante : le navigateur interprète le corps de la réponse pour pouvoir l'afficher correctement, alors que notre programme affiche uniquement le code source HTML. Pour ce qui est du navigateur, il découvre une autre redirection dans l'entête HTML vers « /tinf/ » et demande donc cette page et ses ressources au serveur pour finalement la réinterpréter.

Navigateur Safari :

Filter: http | Expression... | Clear | Apply | Save

No.	Time	Source	Destination	Protocol	Src port	Dst port	Length	Info
42	2.627946000	160.98.113.13	160.98.31.32	HTTP	50880	80	402	GET /tinf/nodir HTTP/1.1
44	2.631152000	160.98.31.32	160.98.113.13	HTTP	80	50880	865	HTTP/1.1 301 Moved Permanently (text/html)
46	2.634328000	160.98.113.13	160.98.31.32	HTTP	50880	80	403	GET /tinf/nodir/ HTTP/1.1
47	2.636259000	160.98.31.32	160.98.113.13	HTTP	80	50880	680	HTTP/1.1 200 OK (text/html)
49	2.691399000	160.98.113.13	160.98.31.32	HTTP	50880	80	446	GET /tinf/ HTTP/1.1
57	2.699306000	160.98.31.32	160.98.113.13	HTTP	80	50880	967	HTTP/1.1 200 OK (text/html)
61	2.755115000	160.98.113.13	160.98.31.32	HTTP	50880	80	399	GET /style.css HTTP/1.1
70	2.758140000	160.98.31.32	160.98.113.13	HTTP	80	50880	1262	HTTP/1.1 200 OK (text/css)
71	2.758149000	160.98.113.13	160.98.31.32	HTTP	50881	80	397	GET /images/Button-Home.png HTTP/1.1
73	2.758471000	160.98.113.13	160.98.31.32	HTTP	50882	80	400	GET /images/Logobloc_EIFnb.jpg HTTP/1.1
77	2.760015000	160.98.113.13	160.98.31.32	HTTP	50883	80	387	GET /tinf/date.js HTTP/1.1
83	2.762813000	160.98.31.32	160.98.113.13	HTTP	80	50881	851	HTTP/1.1 200 OK (PNG)
85	2.764369000	160.98.31.32	160.98.113.13	HTTP	80	50883	771	HTTP/1.1 404 Not Found (text/html)
144	2.775796000	160.98.31.32	160.98.113.13	HTTP	80	50882	1150	HTTP/1.1 200 OK (JPEG JFIF image)
151	2.851014000	160.98.113.13	160.98.31.32	HTTP	50884	80	386	GET /favicon.ico HTTP/1.1
156	2.856046000	160.98.31.32	160.98.113.13	HTTP	80	50884	1200	HTTP/1.1 200 OK (image/x-icon)

Figure 4: Trames échangées suite à l'utilisation d'un navigateur

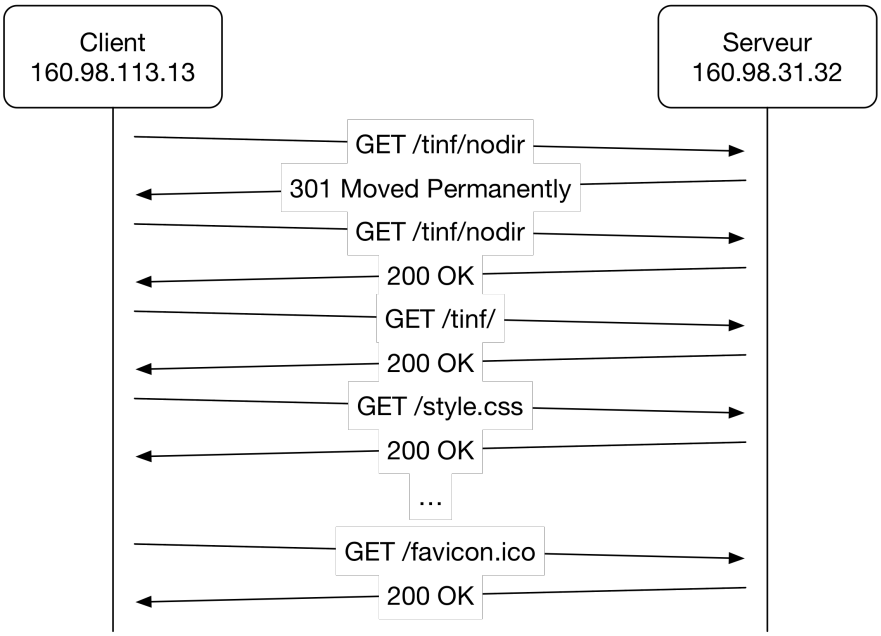

Figure 3: Diagramme en flèches de l'échange

2.4 P4

Répéter la procédure avec `setInstanceFollowRedirects(false)` dans le `HTTPClient`. Qu'est-ce qui change ?

La méthode « setInstanceFollowRedirects(false) » nous permet d'empêcher le suivi automatique d'une redirection vers un nouvel emplacement de la ressource désirée. Nous ajoutons cette méthode juste avant l'établissement de la connexion et nous effectuons à nouveau la requête. En observant la capture Wireshark, nous pouvons

observer que le client n'effectue que la première requête et s'arrête après avoir reçu la réponse du serveur au sujet de la redirection.

Documentation de la méthode :

```
If true, the protocol will automatically follow redirects. If false, the protocol
will not automatically follow redirects.
This field is set by the setInstanceFollowRedirects method. Its value is returned
by the getInstanceFollowRedirects method.
Its default value is based on the value of the static followRedirects at
HttpURLConnection construction time.
```

Filter: http | Expression... | Clear | Apply | Save

No.	Time	Source	Destination	Protocol	Src port	Dst port	Length	Info
215	7.415692000	160.98.114.234	239.255.255.250	SSDP	60980	1900	179	M-SEARCH * HTTP/1.1
219	7.503345000	160.98.113.13	160.98.31.32	HTTP	50928	80	234	GET /tinf/nodir HTTP/1.1
221	7.506403000	160.98.31.32	160.98.113.13	HTTP	80	50928	865	HTTP/1.1 301 Moved Permanently (text/html)

Figure 6: Trames échangées lors de l'utilisation du client HTTP

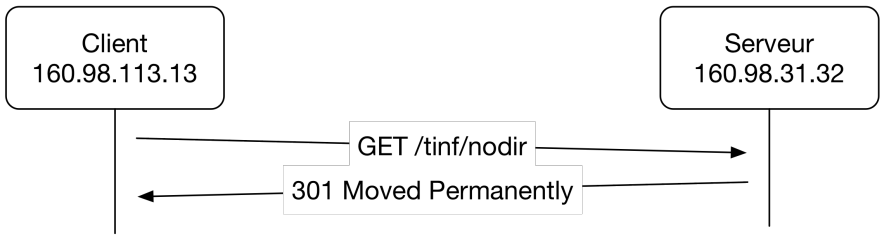

Figure 5: Diagramme en flèche de l'échange

2.5 P5

Faire une synthèse des conclusions pour les points P3 et P4.

Comme nous avons pu l'observer au point 3, il existe deux types de redirections : les redirections HTTP et les redirections HTML. Les redirections HTML ne peuvent être interprétées par l'API Java (uniquement récupération du code source), mais seulement par un navigateur WEB, disposant d'un interpréteur HTML.

2.6 P6

Quel est exactement le problème (utilisant `HTTPClient`) lors de l'affichage du corps de la réponse de la requête vers « tlabs.tic.eia-fr.ch/images/tic.png » ?

En effectuant une requête sur l'adresse mentionnée dans la consigne, nous pouvons voir qu'il a un problème lors de l'affichage du contenu. Ce problème est dû à l'interprétation faite par le client ; l'image est stockée en binaire et est interprétée en tant que caractères ASCII.

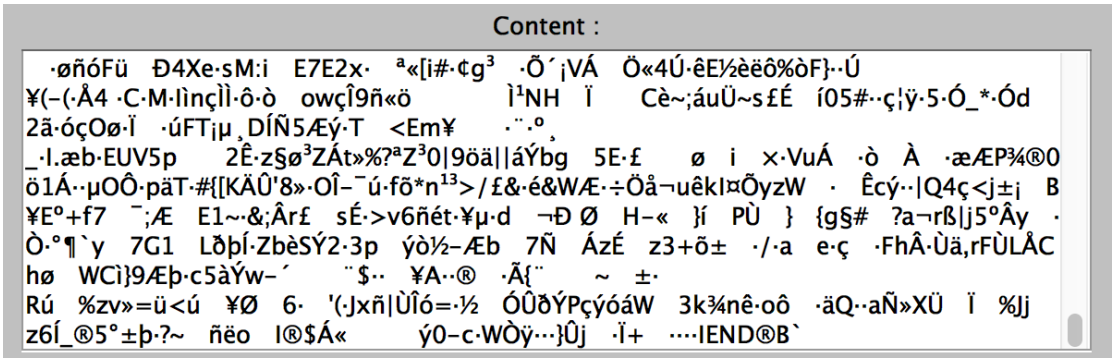

Figure 8: Affichage d'un contenu binaire en ASCII

D'autre part, en analysant la capture Wireshark à l'aide du filtre « http », nous pouvons observer que l'image a pourtant bien été chargée.

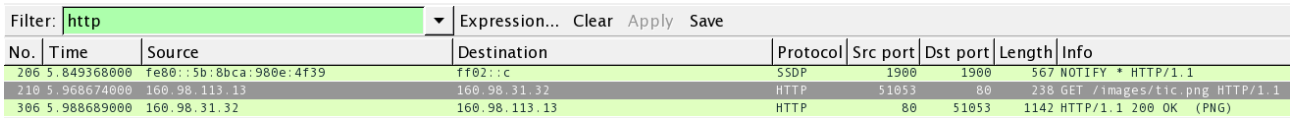

Figure 7: Récupération de l'image PNG

2.7 P7

Implémenter l'option d'envoyer une requête POST (avec deux paramètres codés en dur) en utilisant la classe `HttpURLConnection` (script de test disponible sous http://tlabs.tic.eia-fr.ch/sinf/post.php) .

Pour cette question, nous avons modifié la méthode « doConnect » afin qu'elle puisse transmettre des paramètres codés en dur, par le biais de l'utilisation de la méthode POST. La méthode « doConnect() » ressemble maintenant à ceci :

```
// set request method and perform the request
// returns status line (code & textual explanation) of http response
public String doConnect(String method) throws IOException, ProtocolException {
  conn.setRequestMethod(method);
  // conn.setInstanceFollowRedirects(false);
  if (method == "POST") {
    conn.setDoOutput(true);
    DataOutputStream out = new DataOutputStream(conn.getOutputStream());
    out.writeBytes("param1=value1&param2=value2"); // adds 2 params
    out.flush();
    out.close();
  }

  conn.connect();
  return conn.getHeaderField(0); // gets the status line in header

}
```

Nous avons employé la classe « DataOutputStream », qui permet l'envoi de caractères au travers du flux de données de notre connexion. Les paramètres à transmettre doivent être sous la forme suivante « clé = valeur » et séparés par des « & ». L'emploi de la méthode POST permet ensuite l'envoi des paramètres au sein de la requête.

Nous effectuons ensuite la requête à l'aide du client, en indiquant l'adresse http://tlabs.tic.eia-fr.ch/sinf/post.php et en précisant l'emploi de la méthode POST. Nous pouvons constater que le passage des paramètres a bien fonctionné.

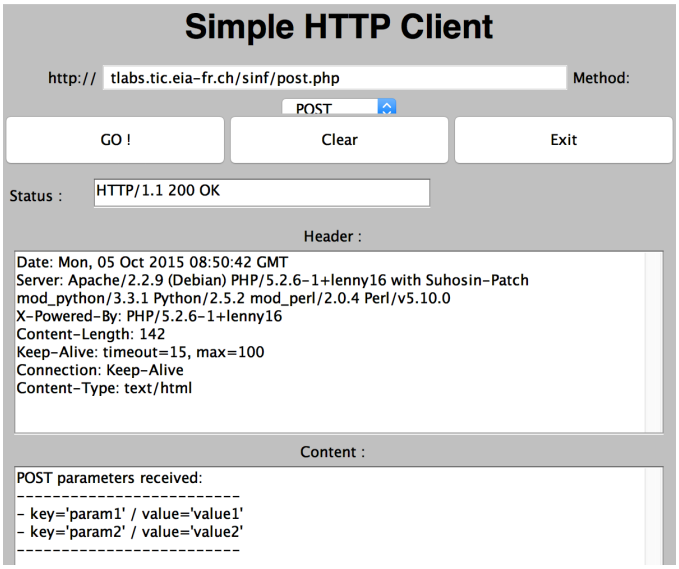

Figure 9: Requête POST à l'aide du client

En observant ensuite les paquets échangés, nous pouvons observer une trace des paramètres dans la requête, ainsi que dans la réponse.

Détails de la requête (méthode POST) :

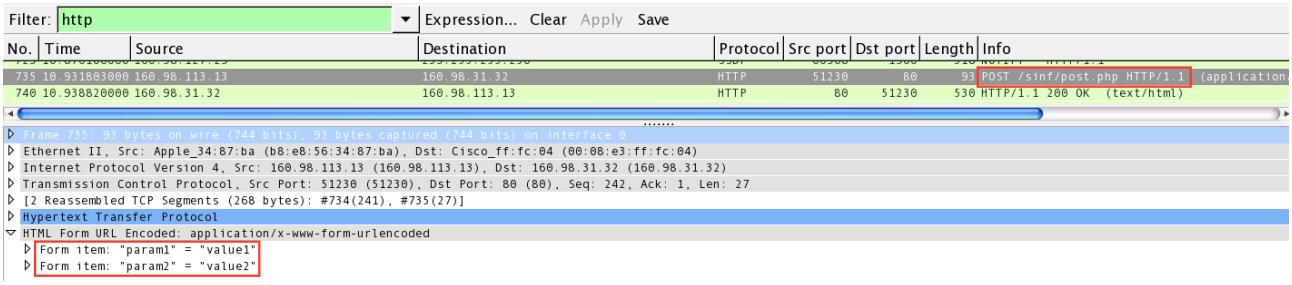

Figure 10: Méthode POST avec 2 paramètres

Suite à la requête, le serveur nous retourne un acquittement de la bonne réception (« 200 OK ») des paramètres.

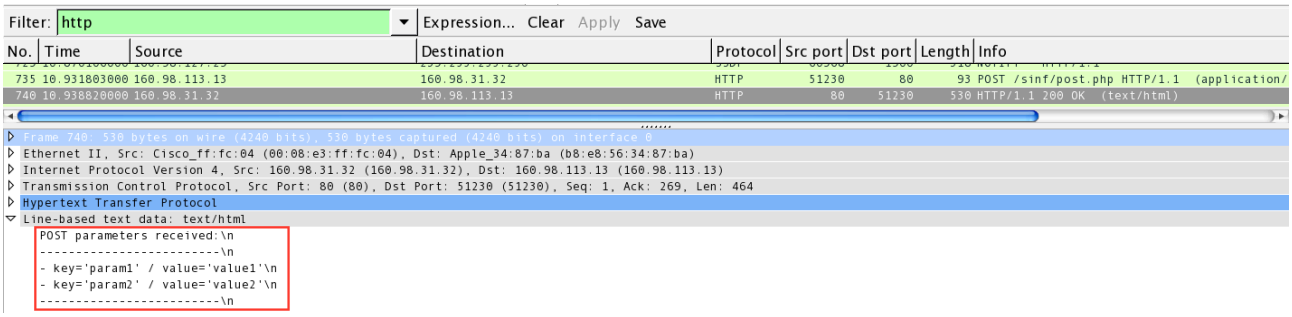

Figure 11: Réponse à la requête POST

2.8 P8

Dans quels entêtes HTTP on retrouve typiquement des paramètres comme p.ex. « *;q=0.8 » ? Expliquer la signification de ces paramètres.

Ces paramètres font référence à des champs présents dans l'entête du protocole HTTP et permettent, pour des requêtes de type GET ou HEAD, de préciser pour une même URI, le type de ressources désiré (préférence du format des images, encodage, langue, etc.).

```
Accept         = "Accept" ":"
                         #( media-range [ accept-params ] )
       media-range    = ( "*/*"
                        | ( type "/" "*" )
                        | ( type "/" subtype )
                        ) *( ";" parameter )
       accept-params  = ";" "q" "=" qvalue *( accept-extension )
       accept-extension = ";" token [ "=" ( token | quoted-string ) ]
```

*;q=0.8 :

- q=0.8 : représente le degré de préférence, qui peut aller de 0 à 1
- *; : représente le type de média

Source : http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

2.9 P9

Faire une mesure Wireshark lors du téléchargement d'une ressource transmise en mode «chunked transfer encoding». Documenter et expliquer le fonctionnement de ce mode en vous basant sur les données brutes de votre mesure.

Le mode « chunked » permet au serveur d'envoyer les données d'une ressource par le biais de petits paquets, au lieu de les envoyer en une seule fois.

Pour démontrer le fonctionnement de ce mode, nous avons effectué une requête sur « http://www.httpwatch.com/httpgallery/chunked/ », que nous filtrons alors avec le filtre suivant : « http.transfer_encoding == chunked ».

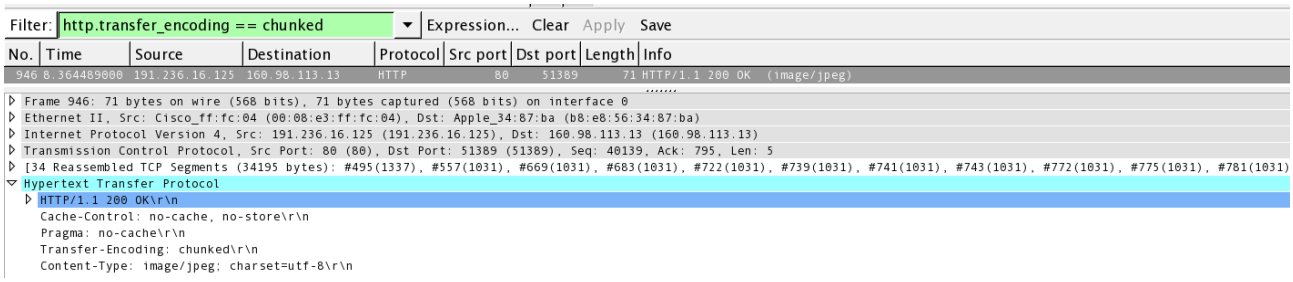

Figure 12: Illustration du mode "chunked"

Par défaut, Wireshark détecte le mode « chunked » et assemble de lui-même les segments TCP dans une même réponse. Comme nous pouvons le voir sur la figure ci-dessus, notre réponse comporte 34 segments TCP contenant les données d'une image.

Les données sont séparées en paquet de 1024 bytes :

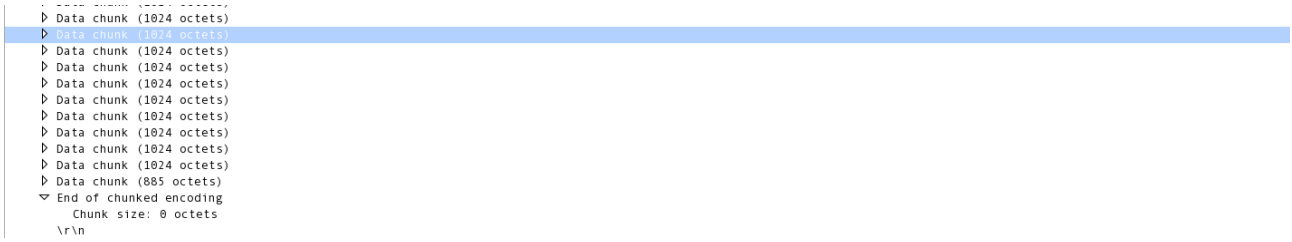

Figure 13: Illustration du mode "trunk"

2.10 P10 (FACULTATIF)

Essayer de faire une mesure qui document le « pipelining ».

Avec l'arrivée de la norme « HTTP/1.1 », il est désormais possible de réaliser plusieurs requêtes à la suite, sans avoir à attendre la réponse à chacune des requête[1].

> Normally, HTTP requests are issued sequentially, with the next request being issued only after the response to the current request has been completely received. Depending on network latencies and bandwidth limitations, this can result in a significant delay before the next request is *seen* by the server. HTTP/1.1 allows multiple HTTP requests to be written out to a socket together without waiting for the corresponding responses. The requestor then waits for the responses to arrive in the order in which they were requested. The act of *pipelining* the requests can result in a dramatic improvement in page loading times, especially over high latency connections.

Pour démontrer ce mécanisme, nous avons réalisé une capture Wireshark en se rendant sur le site http://www.ebookers.ch/.

Comme nous pouvons le constater sur la figure ci-dessous, le navigateur demande une certaine quantité de ressources à la fois, via la méthode GET, et ne reçoit les réponses que par la suite.

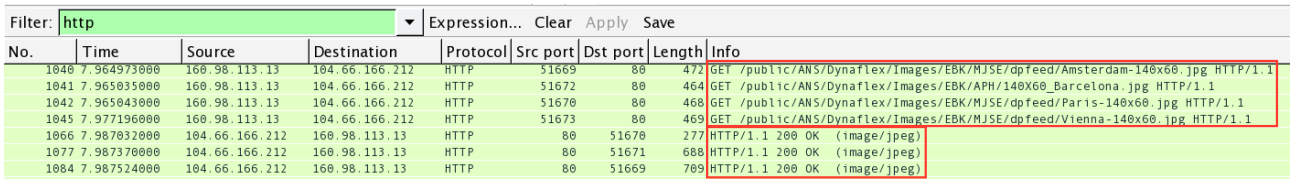

Figure 14: Démonstration du mécanisme du pipelining

[1] http://www-archive.mozilla.org/projects/netlib/http/pipelining-faq.html

2.11 P11 (FACULTATIF)

Documenter (avec Wireshark) une mesure d'une requête vers l'adresse « tlabs.tic.eia-fr.ch/tinf/digest/ » (user "TINF" et mot de passe "labo") et essayer de vérifier le calcul de la réponse au défi.

Suite à la requête sur la page protégée par un nom d'utilisateur / mot de passe (« tlabs.tic.eia-fr.ch/tinf/digest/ », « TINF », « labo ») proposée par M. Scheurer, nous tentons de calculer le défi à l'aide des formules ci-dessous.

```
HA1=MD5(username:realm:password)
HA2=MD5(method:digestURI)
response=MD5(HA1:nonce:nonceCount:clientNonce:qop:HA2)
```

Les informations nécessaires aux calculs se trouvent dans le paquet « http GET /tinf/digest/ http/1.1 ».

- username : TINF
- realm : tlabs
- password : labo
- method : GET
- digestURI : /tinf/digest/
- nonce : kh/6G1ghBQA=d5424bf624dbc2324b414004b60f16652e51f7f8
- nonceCount (nc) : 00000001
- clientNonce (cnonce) : e494a7a68d6194f1cd00a701eb1e42b9
- qop : auth

Nous calculons ensuite les valeurs de « HA1 » et de « HA » » et nous trouvons ensuite la « response ». Comme nous pouvons le constater, celle-ci correspond à la valeur présente dans le détail du paquet ci-dessous.

```
HA1=40a057f8b3549809a5380eaa2da8e3cb
HA2= 4d51f1acfc24e74862a0ed83cb41637f
response=e51febbbb46ee3d0a1c2308d501a4c77
```

Détails du paquet :

```
No.     Time           Source                Destination           Protocol Src
port Dst port Length Info
    416 13.626848000   160.98.113.13         160.98.31.32          HTTP     51852
80       671    GET /tinf/digest/ HTTP/1.1

Frame 416: 671 bytes on wire (5368 bits), 671 bytes captured (5368 bits) on
interface 0
Ethernet II, Src: Apple_34:87:ba (b8:e8:56:34:87:ba), Dst: Cisco_ff:fc:04
(00:08:e3:ff:fc:04)
Internet Protocol Version 4, Src: 160.98.113.13 (160.98.113.13), Dst: 160.98.31.32
(160.98.31.32)
Transmission Control Protocol, Src Port: 51852 (51852), Dst Port: 80 (80), Seq:
1616, Ack: 3913, Len: 605
Hypertext Transfer Protocol
    GET /tinf/digest/ HTTP/1.1\r\n
    Host: tlabs.tic.eia-fr.ch\r\n
    Connection: keep-alive\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11) AppleWebKit/601.1.56
(KHTML, like Gecko) Version/9.0 Safari/601.1.56\r\n
    Accept-Language: en-us\r\n
     [truncated]Authorization: Digest username="TINF", realm="tlabs",
nonce="kh/6G1ghBQA=d5424bf624dbc2324b414004b60f16652e51f7f8", uri="/tinf/digest/",
response="e51febbbb46ee3d0a1c2308d501a4c77", algorithm="MD5",
cnonce="e494a7a68d6194f1cd00
    Accept-Encoding: gzip, deflate\r\n
    \r\n
    [Full request URI: http://tlabs.tic.eia-fr.ch/tinf/digest/]
    [HTTP request 5/6]
    [Prev request in frame: 412]
    [Response in frame: 422]
    [Next request in frame: 427]
```

3 CONCLUSION

Ce travail pratique a été l'occasion d'appréhender et de comprendre le fonctionnement du protocole HTTP et d'utiliser la classe « HttpURLConnection » et ses différentes méthodes.

4 ANNEXE

4.1 HTTPCLIENT.JAVA

```java
/**
 * File   : HTTPClient.java
 * Author : R. Scheurer (HEIA-FR)
 * Date   : 01.10.2015
 *
 * Description - TEMPLATE for a simple client (behind the GUI)
 *
 */

package http;

import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.ProtocolException;
import java.net.URL;

public class HTTPClient {
```

```
HttpURLConnection conn;
String            method;  // http request method

public HTTPClient(String host) throws IOException {
  URL url = new URL("http://" + host);
  conn = (HttpURLConnection) url.openConnection();
}

// set request method and perform the request
// returns status line (code & textual explanation) of http response
public String doConnect(String method) throws IOException, ProtocolException {
  conn.setRequestMethod(method);
  // conn.setInstanceFollowRedirects(false);
  if (method == "POST") {
    conn.setDoOutput(true);
    DataOutputStream out = new DataOutputStream(conn.getOutputStream());
    out.writeBytes("param1=value1&param2=value2"); // adds 2 params
    out.flush();
    out.close();
  }

  conn.connect();
  return conn.getHeaderField(0); // gets the status line in header
}

// returns header fields of HTTP response
public String getHeaders() {
  String headers = "";
  for (int i = 1; i < conn.getHeaderFields().size(); i++) {
    headers += conn.getHeaderFieldKey(i) + ": " + conn.getHeaderField(i)
        + "\n";
  }
  return headers;
}

// get body of HTTP response
// NOTE: treat all exceptions locally
public String getBody() {
  String body = "";
  try {
    if (conn.getResponseCode() / 100 != 4) {
      InputStream in = conn.getInputStream();
      while (in.available() != 0) {
        body += toPrintable((char) in.read());
      }
      in.close();
    }
  } catch (IOException e) {
    e.printStackTrace();
  }
  return body;
}

// GUI user wants to exit ...
public void close() {
  conn.disconnect();
}

String toPrintable(char c) {
  if (c == 10)
    return "\n";
  if (c == 13)
    return "\r";
  if (c == 92)
    return "\\";
  if (c == 34)
```

```
      return "\"";
    if (c < 32) { // | c > 126) {
      return "" + (char) 183;
    }
    return "" + c;
  }

}
```