RAPPORT DE PROJET SAR 2015-2016

MALDEME Alexandre

MOUTON Benoît

LICENCE PRO ADMINISTRATION SYSTEMES ET RESEAUX

Table des matières

Introduction

L'objectif de ce sujet est la conception et l'implémentation d'un mini serveur proxy web, réalisant un certain nombre de fonctions. Un serveur proxy est un ordinateur qui se place entre le routeur (ou modem) et les stations de travail. Il assure un rôle de relai entre le réseau local et internet. Le serveur web réalise cette fonction dans le cadre de la réception des pages/requêtes internet, en utilisant le protocole HTTP.

Les fonctions principales de ce serveur sont :

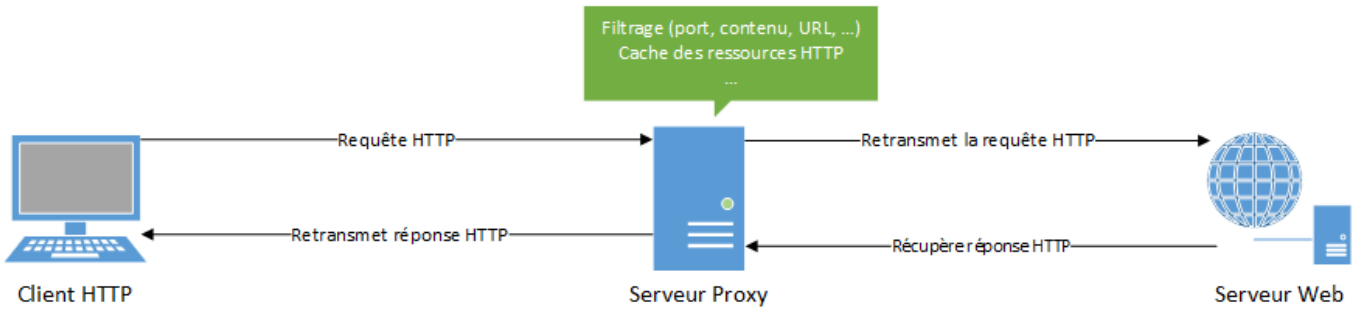
- Le cache.
- Le filtrage de certains sites.
- L'interdiction de télécharger.
- La protection du réseau.
- Limiter la congestion externe.
- Limiter la congestion interne.

Ce Rapport se compose de quatre parties.

La première portera sur l'analyse du projet et des fonctions à implémenter, la seconde sur les algorithmes utilisés lors de la réalisation de ce projet, la troisième partie sur la justification de l'implémentation de chaque composante du projet, et enfin la dernière sur les tests unitaires.

Analyse du projet

Un serveur proxy, (ou en français serveur mandataire), est un serveur servant à faire suivre les requêtes informatiques entre un ordinateur client et un serveur. Il peut avoir de nombreuses utilités, comme par exemple filtrer les requêtes, garantir l'anonymat, chiffrer les échanges ou encore contourner les restrictions imposées sur un réseau donné.



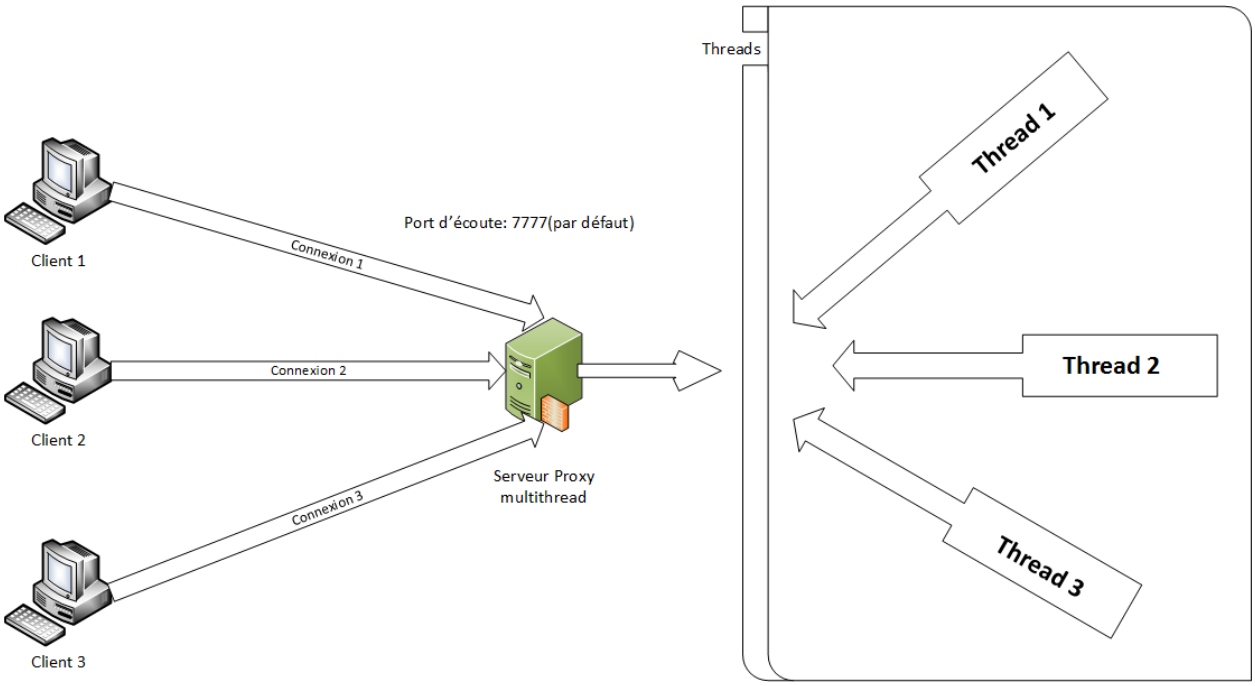
L'objectif de ce projet est d'implémenter un serveur proxy en utilisant les sockets ou RMI. Nous avons choisi ici d'utiliser les sockets. De plus, il devra également supporter :

- **Le cache** : Si une station visite un site donné, le résultat de la requête est stocké. Une deuxième station qui souhaiterait accéder à ce site trouvera les pages déjà chargées dans le cache. Le cache apport plusieurs avantages : le débit sera alors plus proche de 10 Mb ou 100 Mb suivant votre installation et non pas celle de la connexion internet. De plus, pendant ce temps la connexion internet ne sera pas (ou peu) sollicitée, ce qui génère ainsi une très importante économie et une grande augmentation de la vitesse globale.
- **Le filtrage de certains sites** : Le proxy peut choisir d'autoriser ou non les requêtes vers certains sites. On peut interdire certaines adresses internet avec les proxys actuels (chats ou autres contenus illégaux). Le filtrage peut être discriminant. Certaines machines peuvent être autorisées à accéder à certains

contenus que d'autres se verront refuser. On peut interdire l'accès internet à certaines machines sans pour autant interdire l'accès à votre intranet.

- **L'interdiction de télécharger** : Si certains utilisateurs tentent de télécharger des programmes ou des jeux : il est possible de l'interdire avec le proxy. Le client est notifié de l'échec par l'apparition d'un message particulier sur son navigateur.
- **La protection du réseau** : Le proxy vous permet de sécuriser les machines qui se trouvent sur le réseau local. En effet, le proxy permet d'interdire certains ports particulièrement sensibles quant à la sécurité.
- **Limiter la congestion externe** : Le proxy devra limiter le nombre de connections simultanées vers internet. En effet, le nombre de stations accédant en même temps à des pages n'existant pas dans le cache devra être limité à un maximum de n stations. Cela permettra d'éviter les congestions. On doit pouvoir modifier la valeur de n.
- **Limiter la congestion interne** : Le proxy devra limiter le nombre de connections simultanées. En effet, le nombre de stations accédant en même temps au proxy doit être limité à un maximum de m stations. On doit pouvoir modifier la valeur de m.

L'une des conditions sine qua non de ce projet est la création d'un serveur multithread. Ce qui signifie que le serveur doit être capable d'accepter et de gérer plusieurs clients simultanément :

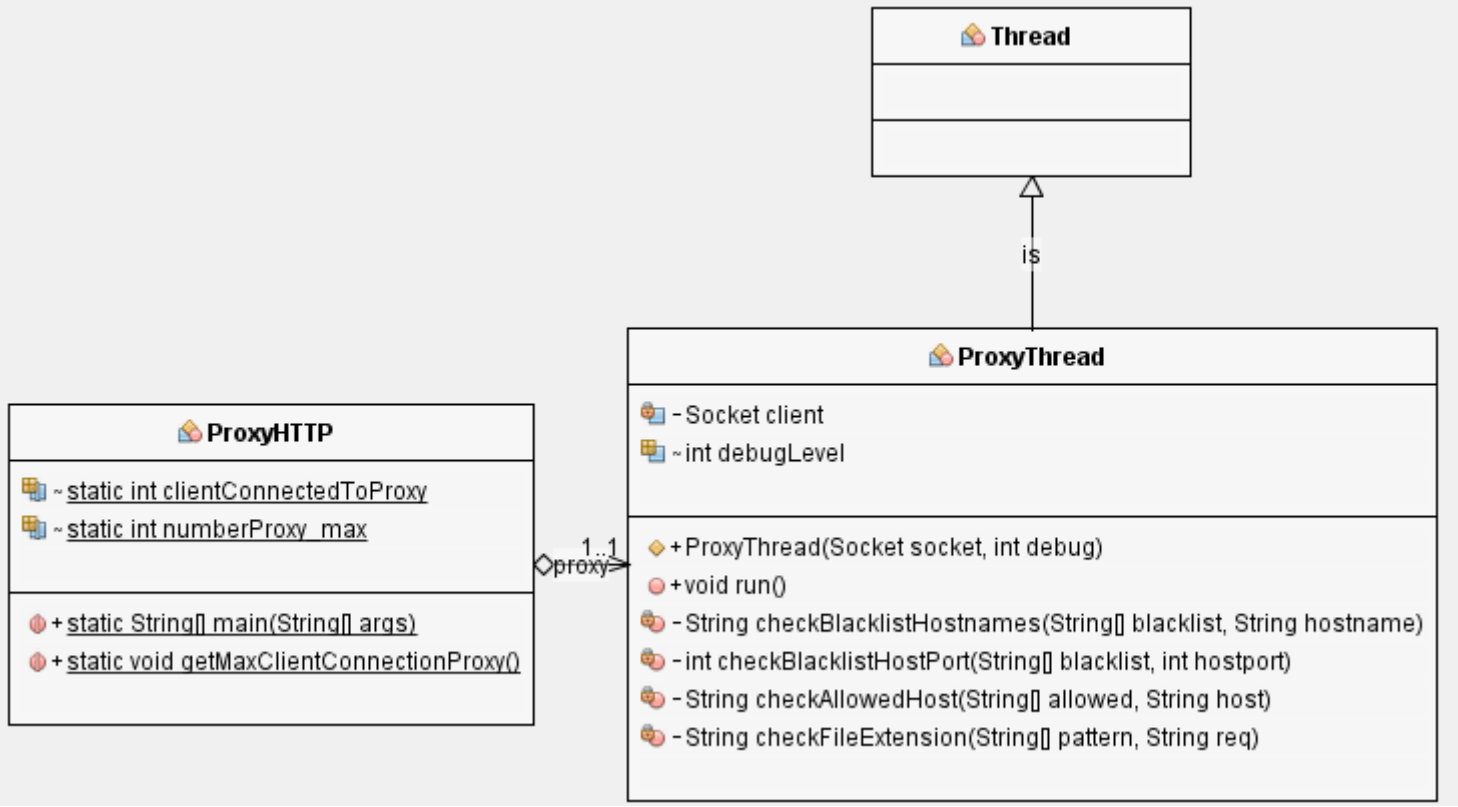


Les requêtes de chaque client s'effectueront dans des Threads séparés créés lors de la connexion des clients au proxy. Ainsi chaque requête pourra être traitée en parallèle à d'autres.

Nous aurons deux types de thread :

- Un thread principal qui va écouter sur le port entré en paramètre du programme. A chaque nouvelle connexion d'un client, il va créer un nouveau thread client.
- Des threads clients créés par le thread principal. C'est ce thread qui va s'occuper de traiter les requêtes HTTP des clients.

Analyse UML : Diagramme de classe



Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci.

La classe `ProxyHTTP()` est donc la classe contenant le thread principal. Elle est composée de deux attributs statiques :

- `clientConnectedToProxy` et
- `numberProxy_max`.

Le premier contient le nombre de clients actuellement connectés au proxy, et le deuxième contient le nombre maximal de clients connectable au serveur. La méthode `getMaxClientConnectionProxy()` sert à récupérer le nombre `numberProxy_max` dans un fichier de configuration détaillé plus bas.

La classe `ProxyThread()` quant à elle est appelée lors de chaque connexion d'un nouveau client. Elle est composée de deux attributs privés :

- `client` de type Socket() et
- *debugLevel* de type Integer.

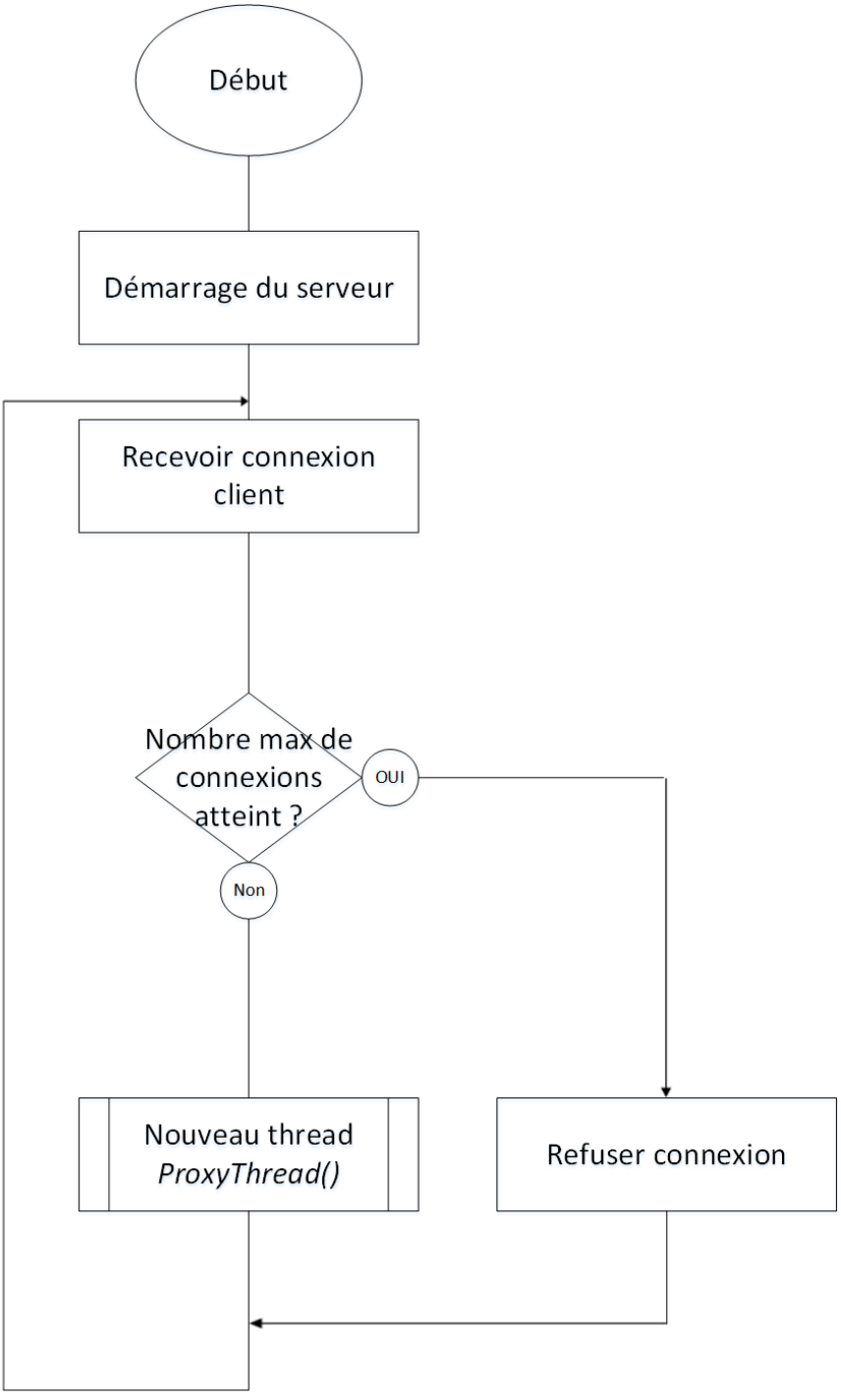
client est la socket utilisée pour la communication avec le client. `debugLevel` est une variable définissant le niveau de verbosité du serveur. En effet, lorsqu'elle est mise à 1, le serveur renvoie beaucoup d'informations utiles au débogage sur la sortie standard.

La classe *ProxyThread()* contient également six méthodes :

- *ProxyThread(Socket socket, int debug)* **:** le constructeur de la classe,
- *void run()* **:** qui contient le code exécuté dans le thread,
- *boolean checkAllowedHost(String[] allowed, String host):* permet de verifier si le client est autorisé à accéder aux sites placés en liste noire,
- *boolean checkBlacklistHostPort(String[] blacklist, int hostport):* Permet de vérifier si le port demandé par le client est sur liste noire ou non,
- *boolean checkBlacklistHostnames(String[] blacklist, String hostname):* permet de vérifier si le client essaye de se connecter à un site sur liste noire,
- *boolean checkFileExtension(String[] pattern, String req):* permet de vérifier si l'extension de l'objet demandé par le client est sur liste noire.

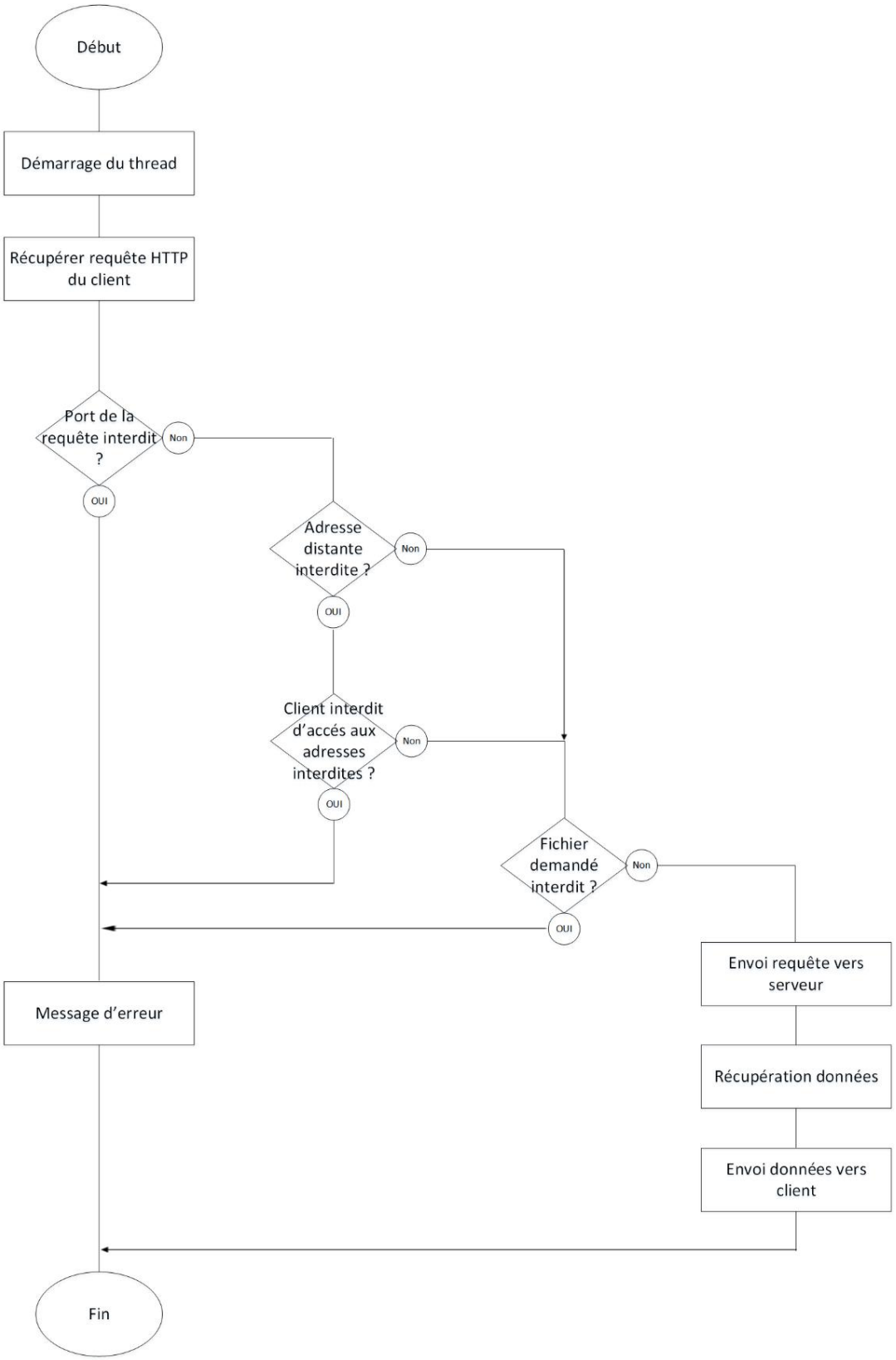
Algorithmes

Analyse du projet : Algorigrammes



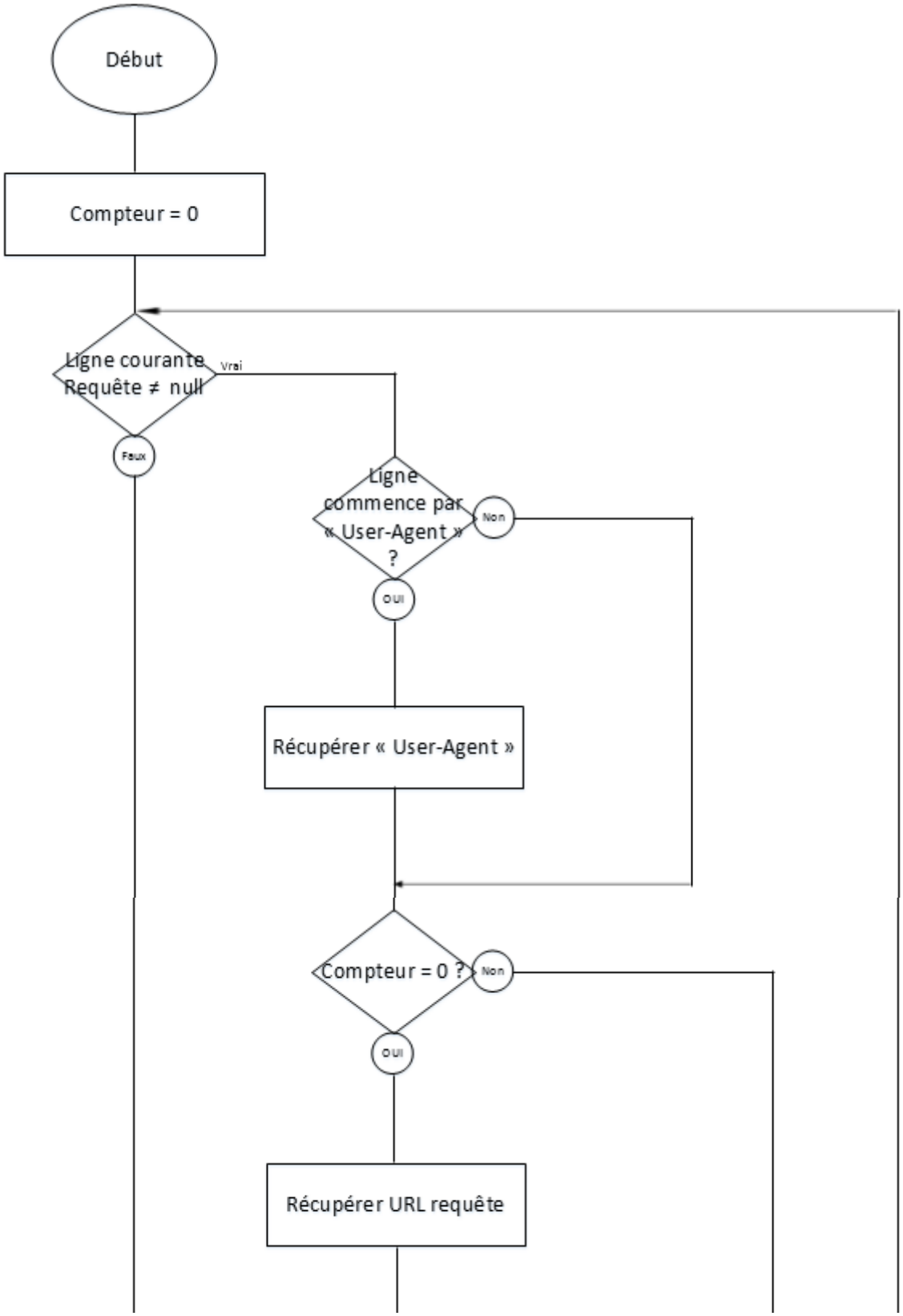
Ce schéma représente l'algorigramme du thread principal.

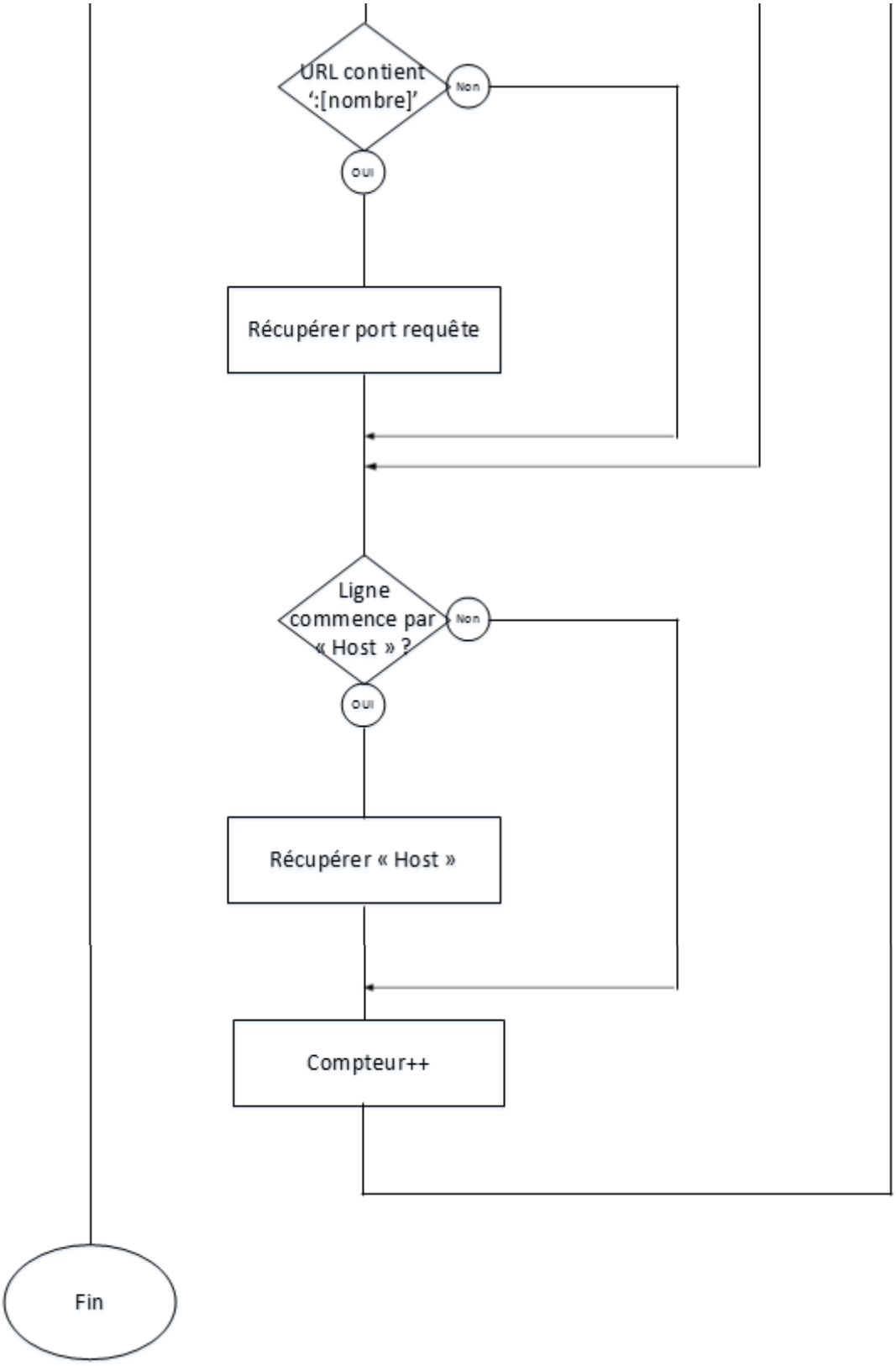
- Une fois le serveur démarré, il est en attente de connexions sur le port passé en paramètre.
- Lorsqu'un client désire se connecter, il vérifie que le nombre maximal de connexions n'est pas atteint. Si c'est le cas, un nouveau thread client est commencé. En revanche si le nombre maximal de connexions est atteint, la connexion venant du client est refusée.


Début
Démarrage du thread
Récupérer requête HTTP du client
Port de la requête interdit ?
Non
OUI
Adresse distante interdite ?
Non
OUI
Client interdit d'accés aux adresses interdites ?
Non
OUI
Fichier demandé interdit ?
Non
OUI
Envoi requête vers serveur
Récupération données
Envoi données vers client
Message d'erreur
Fin

Ce second schéma représente quant à lui l'algorigramme du thread client. Une fois le thread démarré et la requête HTTP du client récupérée, le proxy se doit d'en filtrer le contenu :

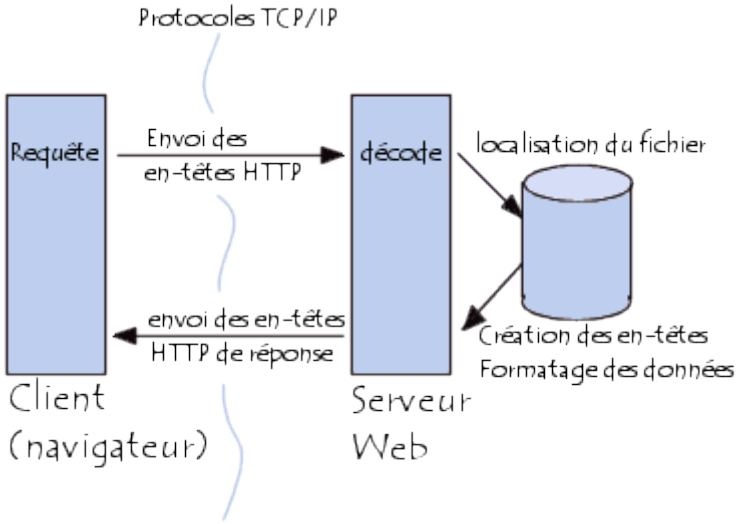
- Si le port sur lequel le client souhaite se connecter est sur liste noire, alors on envoie un message d'erreur au client pour l'en informer et la connexion est terminée.
- Si le port n'est pas sur liste noire mais que le nom du site vers lequel le client émet une requête y est, il y a deux solutions :
    - Soit le client est autorisé à se connecter aux sites sur liste noire, et le cas échéant le programme continue.
    - Soit le client n'est pas autorisé à accéder aux sites sur liste noire et dans ce cas on en informe le client et la connexion se termine à ce moment.
- Si le fichier demandé par le client a son extension sur liste noire, le téléchargement n'a pas lieu. On peut de ce fait interdire le téléchargement de fichier exécutables (.exe) ou des scripts (.js, .sh, .pl, …).


Début
Compteur = 0
Ligne courante Requête ≠ null
Vrai
Faux
Ligne commence par « User-Agent » ?
Non
OUI
Récupérer « User-Agent »
Compteur = 0 ?
Non
OUI
Récupérer URL requête


URL contient ':[nombre]'
Non
oui
Récupérer port requête
Ligne commence par « Host » ?
Non
oui
Récupérer « Host »
Compteur++
Fin

Ce troisième algorigramme est celui du traitement de la requête HTTP du client. En effet, il intervient dans le thread client lorsque le client envoie sa requête HTTP au serveur proxy. Pour analyser cet algorigramme, il va falloir se pencher sur les spécifications du protocole HTTP.

Le but du protocole HTTP est de permettre un transfert de fichiers (essentiellement au format HTML) localisés grâce à une chaîne de caractères appelée URL entre un navigateur (le client) et un serveur Web. La communication entre le navigateur et le serveur se fait en deux temps :



- Le navigateur effectue une requête HTTP
- Le serveur traite la requête puis envoie une réponse HTTP

Une requête HTTP est un ensemble de lignes envoyé au serveur par le navigateur. Elle comprend :

- Une ligne de requête: c'est une ligne précisant le type de document demandé, la méthode qui doit être appliquée, et la version du protocole utilisée. La ligne comprend trois éléments devant être séparés par un espace :

    - La méthode
    - L'URL
    - La version du protocole utilisé par le client (généralement *HTTP/1.0*)

- Les champs d'en-tête de la requête: il s'agit d'un ensemble de lignes facultatives permettant de donner des informations supplémentaires sur la requête et/ou le client (Navigateur, système d'exploitation, ...). Chacune de ces lignes est

composée d'un nom qualifiant le type d'en-tête, suivi de deux points (:) et de la valeur de l'en-tête

- Le corps de la requête: c'est un ensemble de lignes optionnelles devant être séparées des lignes précédentes par une ligne vide et permettant par exemple un envoi de données par une commande POST lors de l'envoi de données au serveur par un formulaire.

Voici un exemple d'une requête HTTP avec le navigateur Mozilla Firefox demandant la page d'accueil du site de météo France :

GET http://www.meteofrance.com/accueil HTTP/1.1
Host: www.meteofrance.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:43.0) Gecko/20100101 Firefox/45.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/*;q=0.8*
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: https://www.google.fr
Connection: keep-alive
Cache-Control: max-age=0

Dans notre cas, dès la réception de la requête du client nous allons la parcourir ligne par ligne pour en récupérer certaine informations :

- L'URL de la page demandée,
- Le nom d'hôte du serveur distant,
- L'User-Agent (indique quel logiciel est utilisé par le client),
- Le port de la connexion s'il est différent de 80.

Implémentation

Configuration du proxy : la classe `Properties()`

Avant de transmettre la requête du client vers le serveur Web distant, le proxy va d'abord analyser cette requête et y appliquer au besoin certains filtres (comme les listes noires pour les ports ou les noms d'hôtes).

Ces listes noires sont stockées dans un fichier *.properties* accessible grâce à la classe `Properties()`. Voici un exemple d'un tel fichier :

```
#Wed Feb 17 17:17:13 CET 2016

# Sites webs interdits.
blacklisted_websites=korben.info;kernel.org

# Clients autorisés à accéder aux sites webs interdits
allowed_clients=172.16.108.140;127.0.0.1;172.16.127.1

# Fichiers dont l'extension est interdite.
forbidden_extensions=.json;.png;.jpg;.jpeg;.exe

# Ports de communication interdits.
blacklisted_port=443;9091;8080

# Nombre maximal de connexions simultanées à Internet.
nbr_connection_internet=256

#Nombre maximal de connexions simultanées au Proxy.
nbr_connection_proxy=256
```

La méthode `split()` permet de séparer les champs des valeurs. Ici notre séparateur est le caractère point-virgule.

Voici un exemple d'une utilisation basique de la classe `Properties()` :

Soit le fichier config.properties suivant :

```
#Wed Feb 17 17:17:13 CET 2016
blacklist=site1;site2;site3
```

On désire accéder aux valeurs de la clef *blacklist* séparées par des points-virgules :

```
Properties prop = new Properties();
// Charge le fichier 'config.properties'.
InputStream input = new FileInputStream("config.properties");
prop.load(input);

String ligne;
String[] tab_values;

// Récupère la clef "blacklist".
ligne = prop.getProperty("blacklist");

System.out.println("blaccklist=" + ligne + "\n");

// Stocke les champs séparés par des ";".
tab_values = ligne.split(";");
for (String val : tab_values)    // Parcourt le tableau et affiche les champs séparés par les ";".
{
    System.out.println(val);
}
```

Et en sortie on obtient :

```
run:
blaccklist=site1;site2;site3

site1
site2
site3
```

L'utilisation d'un fichier de configuration pour stocker différents paramètres paraît justifiée : il est facile de le modifier et un administrateur peut simplement le mettre à jour. De plus, grâce à la classe `Properties()` nativement implémentée dans Java dans la collection `Java.util`, il est facile de venir lire les valeurs de chaque clef.

En revanche, il peut vite devenir fastidieux de maintenir le fichier si de nombreux champs sont présents à chaque clef (par exemple si on met une centaine de sites en liste noire) et il pourrait être préférable soit de créer un fichier par clef (un fichier pour la liste noire des ports, un fichier pour la liste noire des sites web …) soit directement stocker les paramètres dans une base de données (telle que SQLite) mais c'est hors de la portée pédagogique de ce projet.

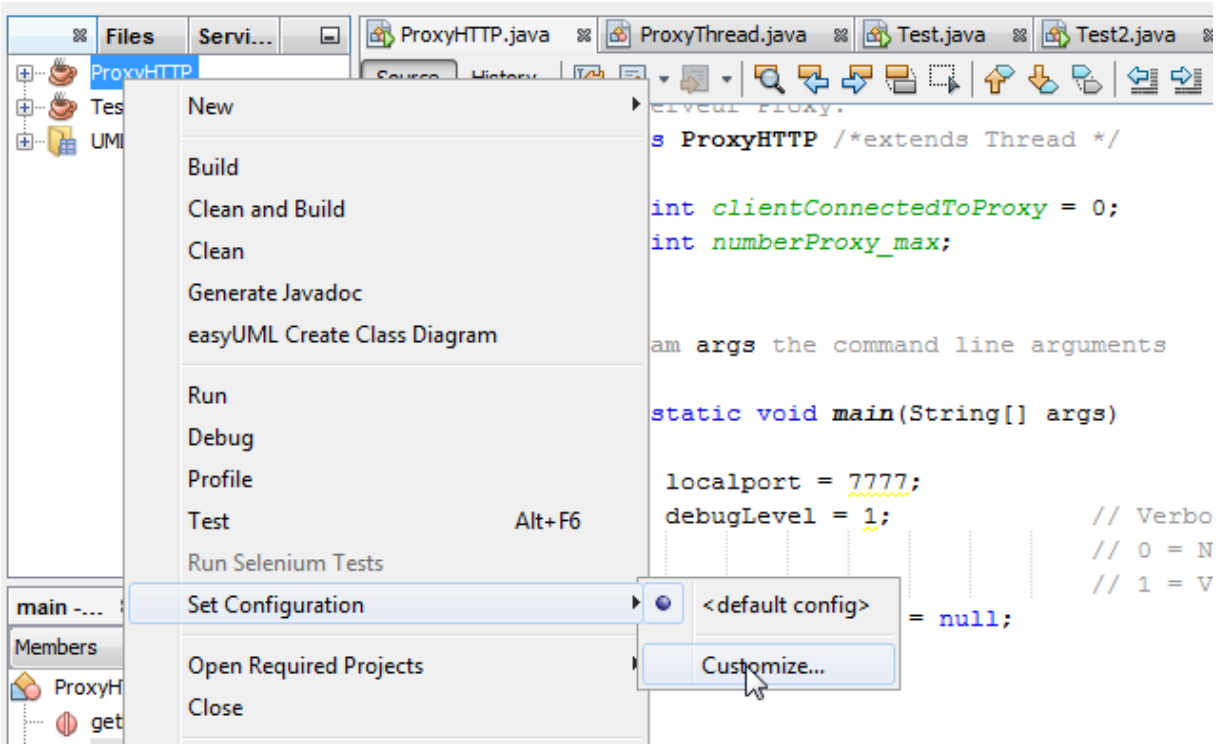
La classe `ProxyHTTP()`

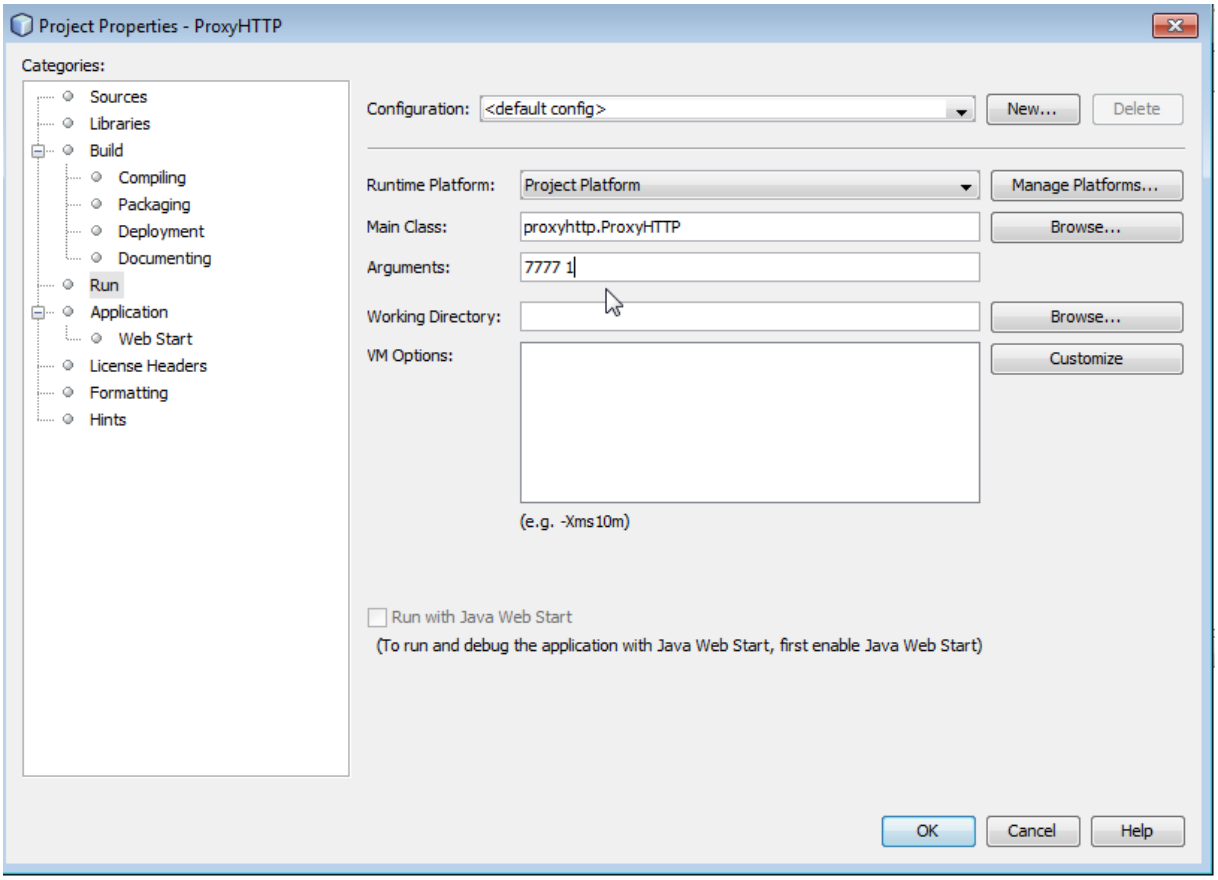
La classe *`ProxyHTTP()`* est donc le thread principal qui écoute sur un port donné et attend les connexions. Pour ce faire, nous utiliserons la classe *`ServerSocket()`*. Elle permet de créer un socket d'écoute qui ne servira en revanche pas à l'échange de données. La méthode *`accept()`* va attendre une connexion et va créer une socket d'échange pour la communication avec le client.

La méthode *`main()`* prend deux arguments :

- Le port d'écoute du serveur proxy,
- Le niveau de verbosité du proxy.

Pour modifier les arguments à passer à la ligne de commande dans un IDE tel que NetBeans, il faut :





Ici le port d'écoute est 7777 et le niveau de verbosité est fixé à 1.

```
try
{
    // Create a listening TCP socket on the local port given in parameters.
    server = new ServerSocket(localport);

    // Get the maximal number of clients allowed to connect to the server.
    getMaxClientConnectionProxy();

    System.out.println("Server is running on port " + localport);
    System.out.println("Debug verbosity level: " + debugLevel);
    System.out.println("Max number of connections to the proxy = " + numberProxy_max);
    System.out.println("\n\n");
}
catch(IOException ex)
{
    System.err.println(ex);
}
```

Voici le morceau de code qui instancie `ServerSocket()` sur le port local donné en argument du `main()`.

```
while(true)
{
    try
    {
        // If number of connected clients to the proxy is bigger than the maximal number of allowed clients.
        if( !(clientConnectedToProxy > numberProxy_max) )
        {
            // Run a new Thread.
            new ProxyThread(server.accept(), debugLevel).start();

            // Incrementing the number of connected clients
            clientConnectedToProxy++;
            if(debugLevel > 0)
                System.out.println("New connection to the proxy. " +  clientConnectedToProxy +
                    " client(s) connected to the proxy.");
        }
        else
        {
            System.err.println("The connection to the proxy from " + server.getInetAddress().getHostAddress() +
                " cannot be established due to numberProxy_max reached.");
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

Ici le serveur écoute si une connexion est demandée à la condition que le nombre maximal de connexions au proxy ne soit pas atteint. Si ce n'est pas le cas, le proxy accepte la connexion et lance un nouveau thread *`ProxyThread()`*.

La classe `ProxyThread()`

La classe *`ProxyThread()`* est donc appelée lorsqu'un client est accepté. On récupère les flux d'entrée et de sortie pour ce socket.

```
// Create the output stream on the client's socket.
OutputStream client_out = client.getOutputStream();
PrintWriter client_pout = new PrintWriter(new OutputStreamWriter(client_out), true);

// Create the reader stream on the client's socket.
BufferedReader client_in = new BufferedReader(new InputStreamReader(client.getInputStream()));
```

Ces flux sont « encapsulés » dans un *`PrintWriter()`* et un *`BufferedReader()`* afin de gérer les flux en tant que chaînes de caractères pour l'écriture et la lecture.

Comme il a été vu dans le troisième algorigramme, la requête HTTP du client contenue dans le flux d'entrée va être traitée ligne par ligne afin d'en récupérer certaines informations.

```java
// Read the stream of the request line by line.
while ((inputLine = client_in.readLine()) != null)
{
    if(debugLevel > 0)
        System.out.println("* * * " + inputLine);

    // Get the User-Agent.
    if ( inputLine.startsWith("User-Agent") )
        UserAgent = inputLine;

    try
    {
        // Get the first line of the HTTP request (ex: GET http://alexasr.tk/ HTTP/1.1)
        StringTokenizer tok = new StringTokenizer(inputLine);
        tok.nextToken();
    }
    catch (Exception e)
    {
        break;
    }
    // Get the URL of the first line in the HTTP request. (ex: http://kernel.org/)
    if (count == 0)
    {
        String[] tokens = inputLine.split(" ");
        urlAsked = tokens[1];

        // If URL contains ':[nombre]', get the port number following.
        Matcher m = Pattern.compile(":[0-9]+").matcher(urlAsked);
        if(m.find())
        {
            // '\\D' means numerical string.
            hostPort = Integer.parseInt(urlAsked.substring(urlAsked.lastIndexOf(":") + 1).replaceAll("\\D+", ""));
        }
    }
    // Get the hostname of the remote web server. (ex: kernel.org)
    if( inputLine.startsWith("Host"))
    {
        String[] tokens = inputLine.split(" ");
        hostName = tokens[1];
    }
    count++;
}
```

Ce morceau de code est la traduction du troisième algorigramme vers le langage Java.

Pour rappel, voici à quoi ressemble une requête HTTP :

GET http://www.meteofrance.com/accueil HTTP/1.1
Host: www.meteofrance.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:43.0) Gecko/20100101 Firefox/45.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,/*;q=0.8*
Accept-Language: fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: https://www.google.fr
Connection: keep-alive
Cache-Control: max-age=0

- Si la ligne commence par “User-Agent”, alors on récupère cette ligne.
- La classe *`StringTokenizer()`* fait partie du package `java.util`. Elle permet de décomposer une chaîne de caractères en une suite de mots (tokens) séparés par des délimiteurs. On l’utilise ici pour récupérer la première ligne de la requête HTTP.
- Ensuite si le compteur vaut 0 (donc on travaille sur la première ligne de la requête), on récupère l’URL demandé grâce à la méthode *`split()`* qui permet de découper une chaîne d’après un délimiteur (qui ici est un espace : ‘’ ‘’).
    - D’après la RFC-2616-14.23: "Hypertext Transfer Protocol -- HTTP/1.1", le port distant n’est présent dans la requête HTTP que s’il diffère de celui par défaut (soit 80). Si ce n’est pas le cas, le port est spécifié après le nom d’hôte séparé par un point-virgule : <hôte>:<port>
      Exemple : *GET http://kernel.org:8080/ HTTP/1.1*
      On va donc vérifier grâce à une Regex si l’URL contient un numéro après un double-point.

```
// If URL contains ':[nombre]', get the port number following.
Matcher m = Pattern.compile(":[0-9]+").matcher(urlAsked);
```

Dans l’exemple précédent, on récupérerait le numéro de port : *http://kernel.org:**8080**/*

Si une telle occurrence est trouvée, on récupère le numéro du port dans la variable *`hostPort`*.

```
if(m.find())
{
    // '\\D' means numerical string.
    hostPort = Integer.parseInt(urlAsked.substring(urlAsked.lastIndexOf(":") + 1).replaceAll("\\D+", ""));
}
```

La méthode *`ParseInt()`* permet de « traduire » un `String` vers un `Integer`.

C'est après le traitement de la requête du client que le proxy va filtrer en fonction des listes noires. C'est le rôle des methodes *checkAllowedHost(String[] allowed, String host)*, *checkBlacklistHostPort(String[] blacklist, int hostport)*, *checkBlacklistHostnames(String[] blacklist, String hostname)* et *checkFileExtension(String[] pattern, String req)*

```
// Return true if host port is blacklisted.
private boolean checkBlacklistHostPort(String[] blacklist, int hostport)
{
    for(String port : blacklist)
    {
        // True if equals, false if not.
        if(Integer.toString(hostport).equals(port))
            return true;
    }
    return false;
}
```

Cette méthode retourne *true* si le port est sur liste noire, et *false* le cas contraire. Elle prend en argument un tableau de *String* contenant tous les ports sur liste noire et le port demandé par le client.

Le tableau de *String* est parcouru et si le port demandé par le client est égal à un des ports présents dans le tableau, alors la méthode renvoie *true*.

La structure `for(String port : blacklist)` permet de parcourir une collection d'objets ou un tableau. Il est de la forme

```
for (type variable : <expression>) {
    instruction(s)
 }
```

Où <**expression**> est ici un tableau de String. C'est-à-dire qu'à chaque tour de boucle le String *port* prend pour valeur la valeur suivante dans le tableau de `String blacklist`.

Les autres méthodes reprennent le même principe :

```
// Return true if hostname is blacklisted.
private boolean checkBlacklistHostnames(String[] blacklist, String hostname)
{
    for(String domain : blacklist)
    {
        // True if equals, false if not.
        if(hostname.equals(domain))
            return true;
    }
    return false;
}

// Return true if @IP is an allowed client.
private boolean checkAllowedHost(String[] allowed, String host)
{
    for(String address : allowed)
    {
        if(host.equals(address))
        {
            System.out.println(address + " autorisé à accéder au site.");
            return true;
        }
    }
    return false;
}

// Return true if pattern matches.
private boolean checkFileExtension(String[] pattern, String req)
{
    for(String match : pattern)
    {
        Matcher m = Pattern.compile(match).matcher(req);

        if(m.find())
        {
            System.err.println("/!\\ " + match + " file detected.");
            return true;
        }
    }
    return false;
}
```

checkFileExtension() en revanche va filtrer le String *req* à la recherche d'une extension de fichier sur liste noire en utilisant les expressions régulières.

Le morceau de code suivant représente le deuxième algorigramme. C'est l'implémentation du filtrage par le proxy :

```
// If hostport is blacklisted.
if(checkBlacklistHostPort(hostPortBlacklisted, hostPort))
{
    System.err.println("/!\\ Port " + hostPort + " is blacklisted. Cancelling request.");
    client_pout.println("/!\\ Port " + hostPort + " is blacklisted. Cancelling request.");
    client.close();
    ProxyHTTP.clientConnectedToProxy--;
    System.out.println("End of connection to the proxy from " + client.getInetAddress().getHostAddress());
    System.out.println("Number of connected client(s) : " + ProxyHTTP.clientConnectedToProxy);
    return;
}

// If hostName isn't blacklisted.
if(!checkBlacklistHostnames(hostBlacklisted, hostName))
{
    url_ = new URL(urlAsked);
    if(debugLevel > 0)
        System.out.println("Connection from " + client.getInetAddress().getHostAddress() + " for " + hostName + " accepted.\n");
}
else // Hostname is blacklisted.
{
    // If client is allowed to access blacklisted hostnames.
    if(checkAllowedHost(allowedClients, client.getInetAddress().getHostAddress()))
    {
        url_ = new URL(urlAsked);
        if(debugLevel > 0)
            System.out.println("Connection from " + client.getInetAddress().getHostAddress() + " accepted.");
    }

    else
    {
        System.err.println("/!\\ " + hostName + " is blacklisted. Cancelling request.");
        client_pout.println("/!\\ " + hostName + " is blacklisted. Cancelling request.");
        client.close();
        ProxyHTTP.clientConnectedToProxy--;
        System.out.println("End of connection to the proxy from " + client.getInetAddress().getHostAddress());
        System.out.println("Number of connected client(s) : " + ProxyHTTP.clientConnectedToProxy);
        return;

    }
}
```

Ainsi, lorsque le filtrage est effectif et qu'une requête ne peut aboutir car elle essaye de demander une ressource sur liste noire (port, hôte …), on affiche un message d'erreur sur la sortie d'erreur standard 'stderr' et on envoie également un message d'erreur au client en utilisant le *`PrintWriter client_pout`* :


```
/!\ Port 8080 is blacklisted. Cancelling request.
```


```
/!\ perdu.com is blacklisted. Cancelling request.
```

Si la requête peut continuer et qu'aucun filtrage ne s'applique, on instancie la classe *URL()* dans l'objet *url_*.

```
url_ = new URL(urlAsked);
```

La classe *URL()* permet d'encapsuler une URL. La validité syntaxique de l'URL est assurée mais l'existence de la ressource représentée par l'URL ne l'est pas. Le constructeur de la classe lève une exception du type *MalformedURLException* si la syntaxe de l'URL n'est pas correcte.

La classe URL possède des getters pour obtenir les différents éléments qui la composent : *getProtocole(), getHost(), getPort(), getFile().*

La méthode *openStream()* ouvre un flux de données en entrée pour lire la ressource et renvoie un objet de type *InputStream*.

La méthode *openConnection()* ouvre une connexion vers la ressource et renvoie un objet de type *URLConnexion()*

Pour envoyer la requête du client vers le serveur Web distant, nous allons justement utiliser la méthode *openConnection()* et donc la classe *URLConnection()* grâce à *HttpURLConnection()* qui en hérite.

```
if(debugLevel > 0)
    System.out.println("Sending a request for URL: " + urlAsked);

// Connection to the website asked by the client.
HttpURLConnection connection = (HttpURLConnection)url_.openConnection();
HttpURLConnection httpConn = (HttpURLConnection)connection;
connection.setRequestMethod("GET");
connection.setDoOutput(false);
httpConn.setRequestProperty("Accept","*/*");

// Some servers won't respond if User-Agent isn't informed.
if ( ! UserAgent.isEmpty())
{    String[] a = UserAgent.split(":", 2);
    httpConn.setRequestProperty(a[0], a[1]);
}
```

.

La classe *URLConnection()* est abstraite et encapsule une connexion vers une ressource désignée par une URL pour obtenir un flux de données ou des informations sur la ressource.

La classe *HttpURLConnection()* hérite de *URLConnection()*. Elle implémente en plus le support des fonctionnalités HTTP.

La méthode *setRequestMethod(String method)* prépare la méthode pour la requête. Les valeurs peuvent être GET POST HEAD OPTIONS PUT DELETE TRACE.

La méthode *setDoOutput(boolean dooutput)* va modifier la valeur du champ *DoOutPut* de l'*URLConnection*. Le mettre à *true* indique que l'application a l'intention d'écrire des données au travers de l'*URLConnection* ce qui n'est pas le cas ici.

La méthode *SetRequestProperty()* quand a elle permet de paramétrer la requête HTTP. Le premier paramètre est le nom du champ (ex : Accept), le second la valeur dudit champ.

Puisque certains serveurs ne répondent pas aux requêtes si le champ « User-Agent » n'est pas rempli, on le rempli avec le User-Agent récupéré dans la requête du client.

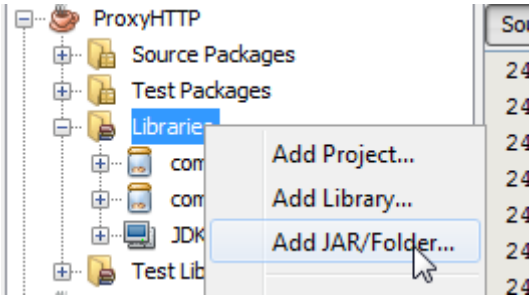
```
httpConn.setRequestProperty("Accept","*/*");

// Some servers won't respond if User-Agent isn't informed.
if ( ! UserAgent.isEmpty())
{    String[] a = UserAgent.split(":", 2);
    httpConn.setRequestProperty(a[0], a[1]);
}
```

Ensuite nous copions le flux de réponse du serveur Web dans le flux de sortie du client grâce à la méthode *org.apache.commons.io.IOUtils.copy()*. Cette méthode fait partie du package *commons-io-2.4* téléchargeable sur le site :

http://commons.apache.org/proper/commons-io/download_io.cgi

Il faut importer ce package au projet. En utilisant l'IDE Netbeans, il faut faire :



Et ensuite aller chercher le package dernièrement téléchargé.

La méthode `org.apache.commons.io.IOUtils.copy()` permet donc de copier un flux dans un autre en créant un buffer interne pour ne pas avoir à utiliser `BufferedInputStream()`.

```
try
{
    // Copy response stream (input) of the remote server to the output stream of the client.
    // This method create an intern buffer to not use "BufferedInputStream"
    org.apache.commons.io.IOUtils.copy(connection.getInputStream(), client_out);
}
catch(FileNotFoundException e) // In case of unavailable file on the server (ex: 404 error).
{
    System.err.println("FILENOTFOUNDEXCEPTION: " + e);
    e.printStackTrace();
}
```

La classe `ProxyThread()` : choix d'implémentations

Le choix d'utiliser les classes `URL()` et `HttpURLConnection()` est simple. Elles permettent une grande souplesse et une grande facilité et fournissent un ensemble complet de méthodes pour la création et réception de requête HTTP. Il est aisé d'écrire des requêtes GET, POST ou autres. L'une des limites est en revanche la non prise en charge du protocole HTTPS (mais l'objectif du projet est de créer un serveur proxy HTTP, et non pas HTTPS).

Ensuite, il a été choisi d'utiliser le package *commons-io-2.4* car, non seulement il est bien documenté (http://commons.apache.org/proper/commons-io/javadocs/api-2.4/), mais il est aussi beaucoup utilisé et dans de nombreux projets. De plus la méthode

`copy()` fera sûrement mieux que nous la copie d'un flux entier vers un autre : rien ne sert de réinventer la roue.

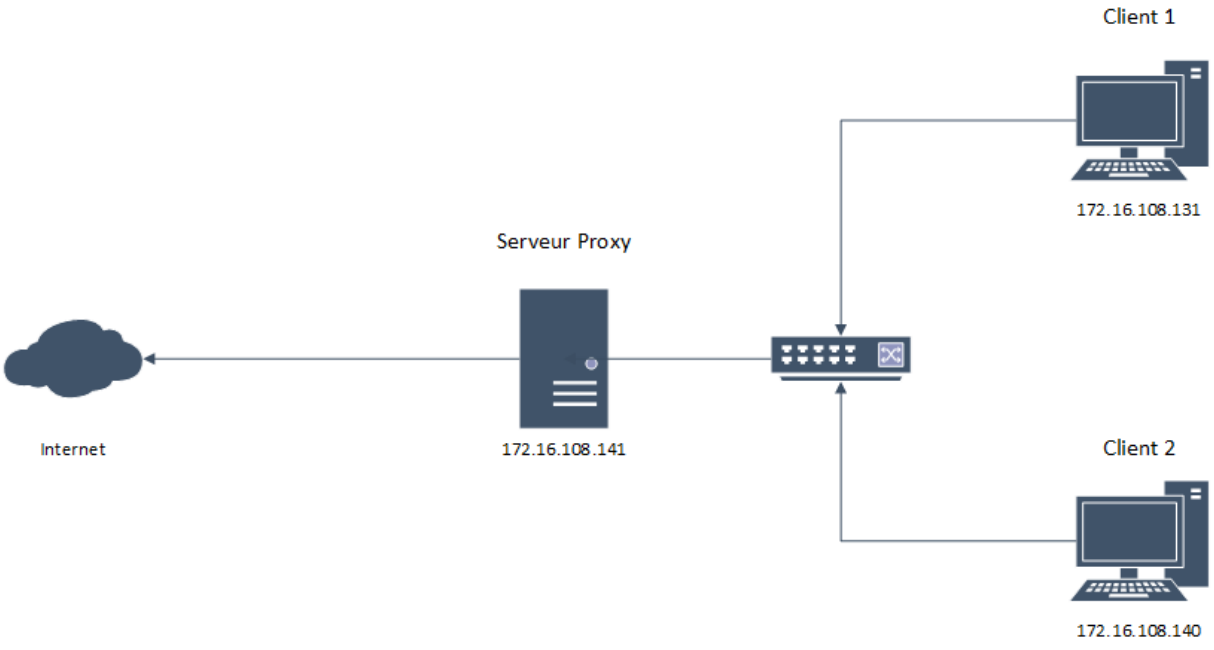
Initialement le serveur proxy devait pouvoir mettre en cache les pages Web des clients. Il a été choisi de ne pas implémenter cette fonctionnalité par manque de savoir-faire. Le cache aurait pu être stocké soit dans des fichiers textes, soit dans des fichiers binaires (sérialisés) ou encore dans une base de données (afin de bénéficier de l'indexation ou encore du tri par dates).

Tests unitaires

Dans cette partie nous allons tester les principales fonctions du serveur proxy. A savoir sa capacité à relayer les requêtes des clients vers les sites web demandés ou encore les différents filtrages.

Architecture réseau

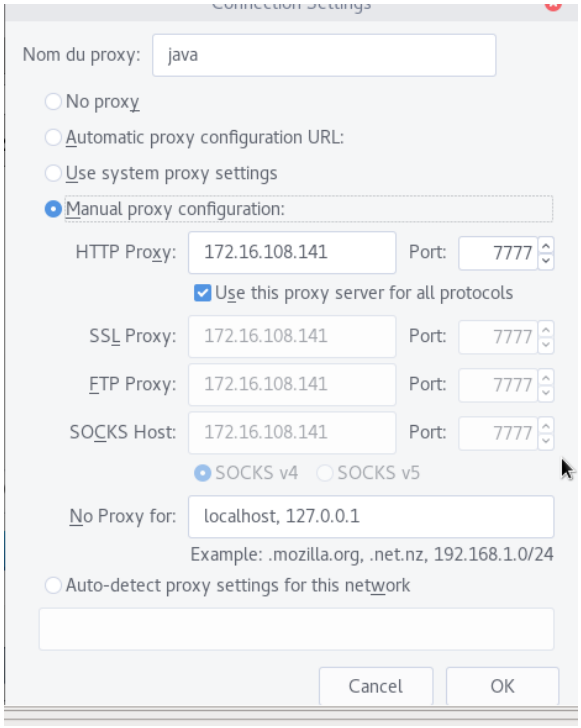
Nous allons réaliser cette série de tests dans un environnement virtuel en utilisant l'hyperviseur VMware Workstation Pro v12. De ce fait, une machine virtuelle fera tourner le serveur proxy, et une ou plusieurs autres feront office de clients :



Note : Il est aussi possible de n'utiliser qu'une seule machine, qui jouerait à la fois le rôle de serveur proxy et de client. Mais il est plus pédagogique de séparer les rôles.

Configuration des clients et du serveur

La configuration proxy du navigateur Mozilla Firefox des deux clients est la suivante :



Pour le serveur proxy, le fichier « config.properties » est le suivant :

```
#Wed Feb 17 17:17:13 CET 2016

# Sites webs interdits.
blacklisted_websites=korben.info;kernel.org;alexasr.tk;perdu.com

# Clients autorisés à accéder aux sites webs interdits
allowed_clients=172.16.108.140;172.16.127.1

# Fichiers dont l'extension est interdite.
forbidden_extensions=.json;.png;.jpg;.jpeg;.exe

# Ports de communication interdits.
blacklisted_port=443;9091;8080

# Nombre maximal de connexions simultanées à Internet.
nbr_connection_internet=256

#Nombre maximal de connexions simultanées au Proxy.
nbr_connection_proxy=256
```

Les sites sur liste noire sont :

- korben.info
- kernel.org
- alexasr.tk
- perdu.com

Les clients autorisés à accéder aux sites sur liste noire sont :

- 172.16.108.140 (soit le client 2)
- 172.16.127.1 (soit une machine d'un autre réseau qui ne nous intéresse pas ici)

Les extensions de fichiers sur liste noire sont :

- json
- png
- jpg
- jpeg
- exe

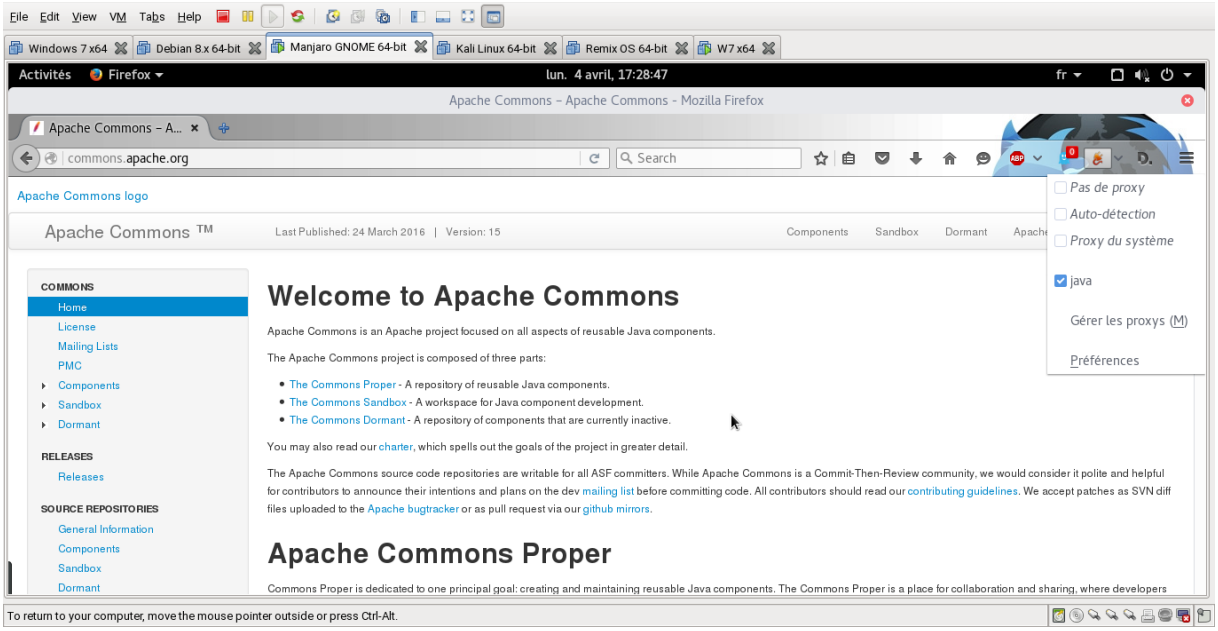
Les ports de communications sur liste noire sont :

- 443
- 9091
- 8080

Et 256 clients sont autorisés à se connecter simultanément au serveur proxy et à Internet.

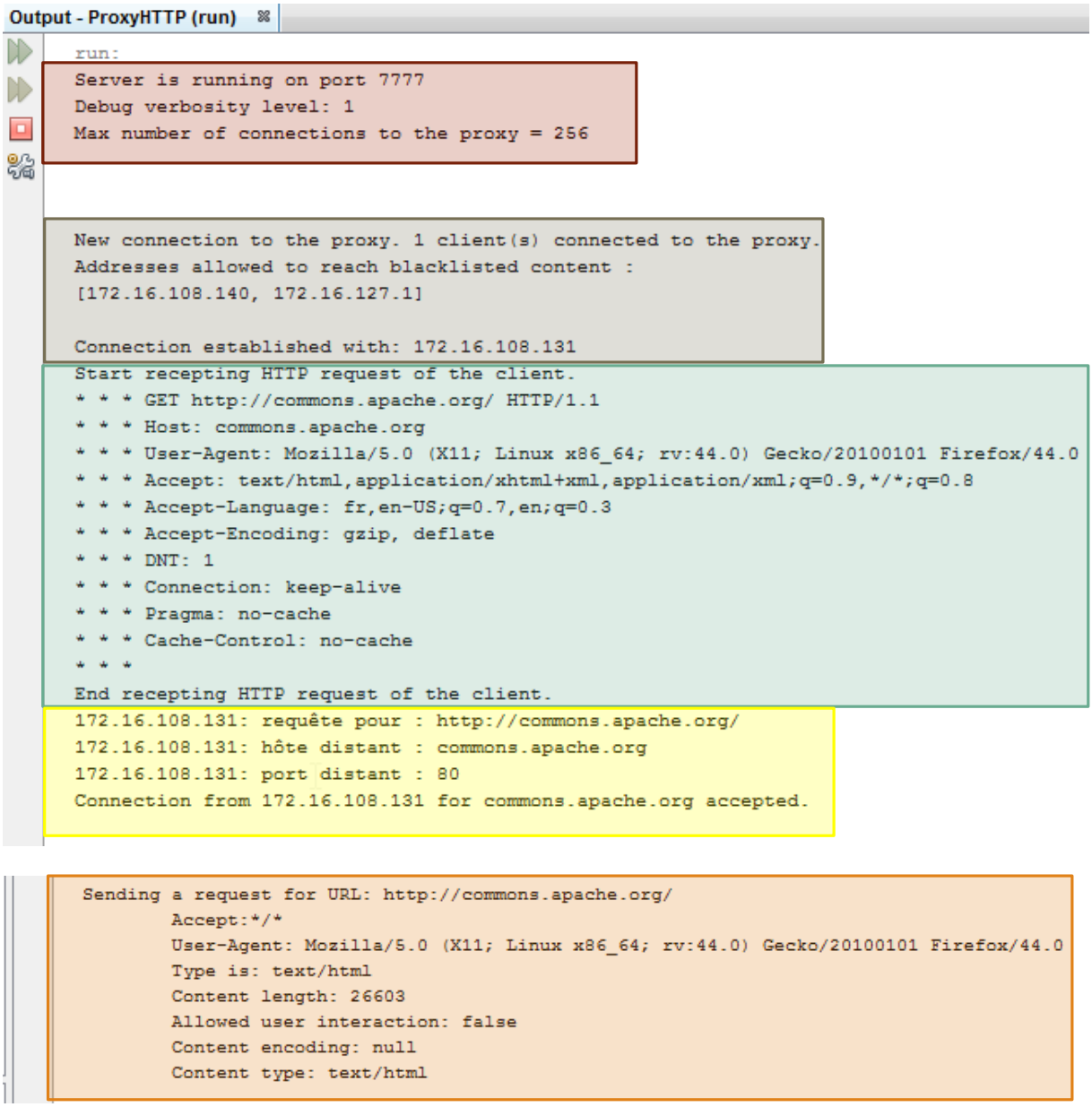
Relai de requêtes HTTP

Pour ce test, le client va tenter de se connecter à un site web non présent en liste noire : *http://commons.apache.org/*



On voit bien sur cette capture d'écran que le client a réussi à obtenir la page qu'il a demandé.

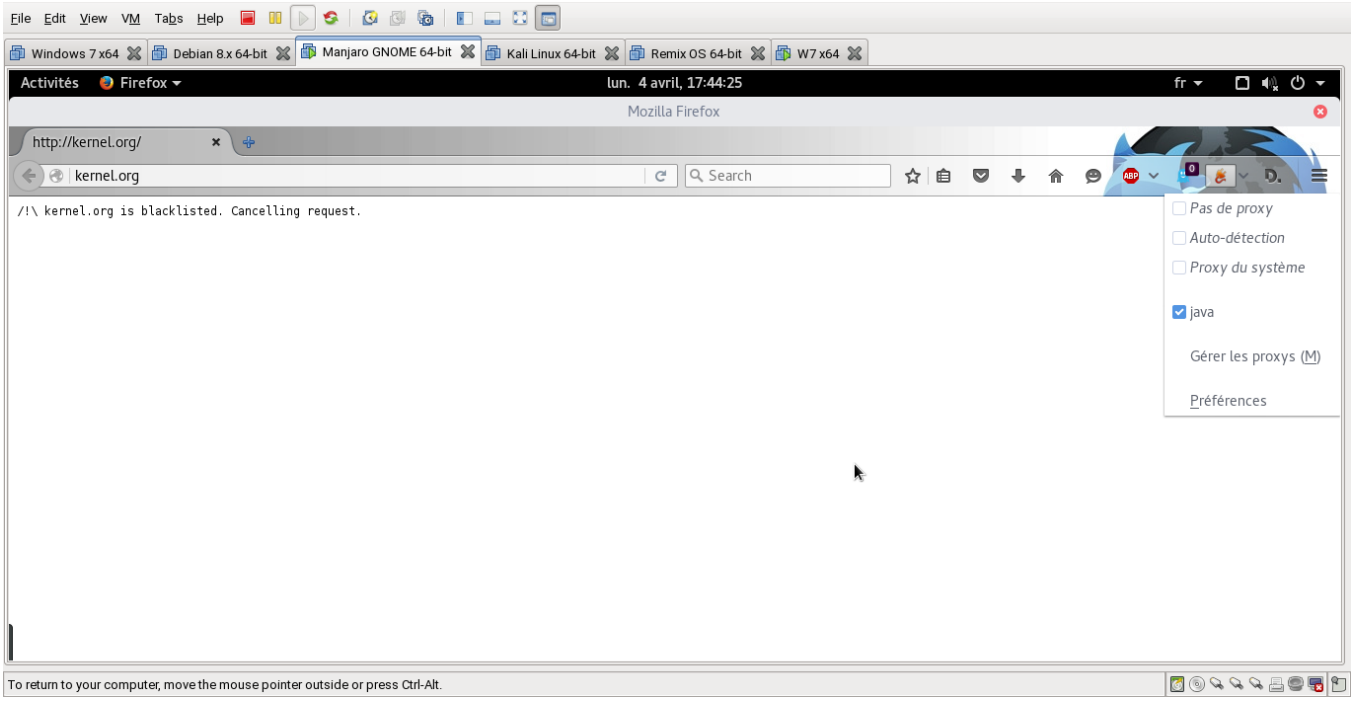
Regardons les premières lignes de la sortie du serveur proxy :



- Les trois premières lignes retournent des informations sur la configuration du proxy.
- Ensuite le proxy reçoit une nouvelle connexion de 172.16.108.131 qui est acceptée.
- La troisième partie affiche la requête HTTP du client en détail.
- La partie suivante affiche un court récapitulatif de la requête du client (URL demandée, hôte distant, port distant).
- Et enfin on affiche la requête HTTP du serveur proxy vers le site web distant

Accès à un site sur liste noire (1)

Maintenant, le client va essayer de se connecter à un site présent dans la liste noire du proxy (et ce client n'est pas autorisé à consulter les sites blacklistés).



Le serveur proxy a refusé de se connecter au site *kernel.org* et en a averti le client en lui affichant un message d'erreur sur son navigateur.

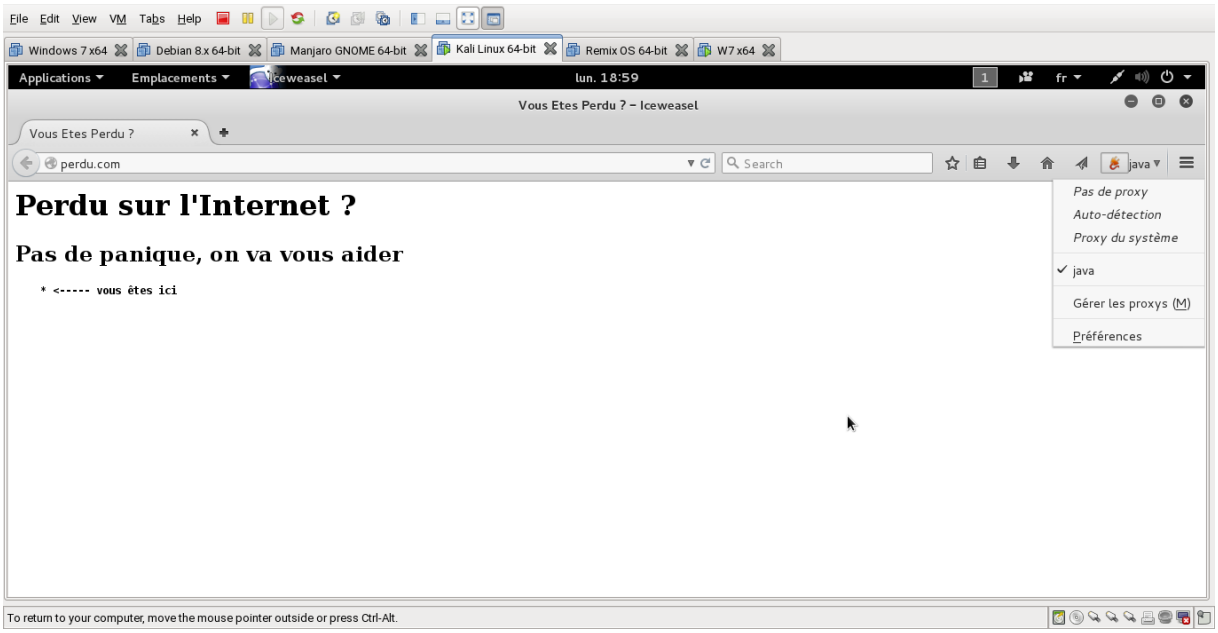
Regardons les lignes de la sortie du serveur proxy :

```
/!\ kernel.org is blacklisted. Cancelling request.
```

Contrairement à précédemment, pendant que le proxy parcourt ligne par ligne la requête HTTP du client, il va détecter que l'hôte distant est sur liste noire et va afficher un message d'erreur sur la sortie standard. Le client n'aura pas accès à sa page Web.

Accès à un site sur liste noire (2)

Ici, le client 172.16.108.140 (client 2) va essayer de se connecter au site *perdu.com* qui est présent dans la liste noire. Mais le client 2 est autorisé à accéder à ces sites :



Et la sortie du serveur proxy :

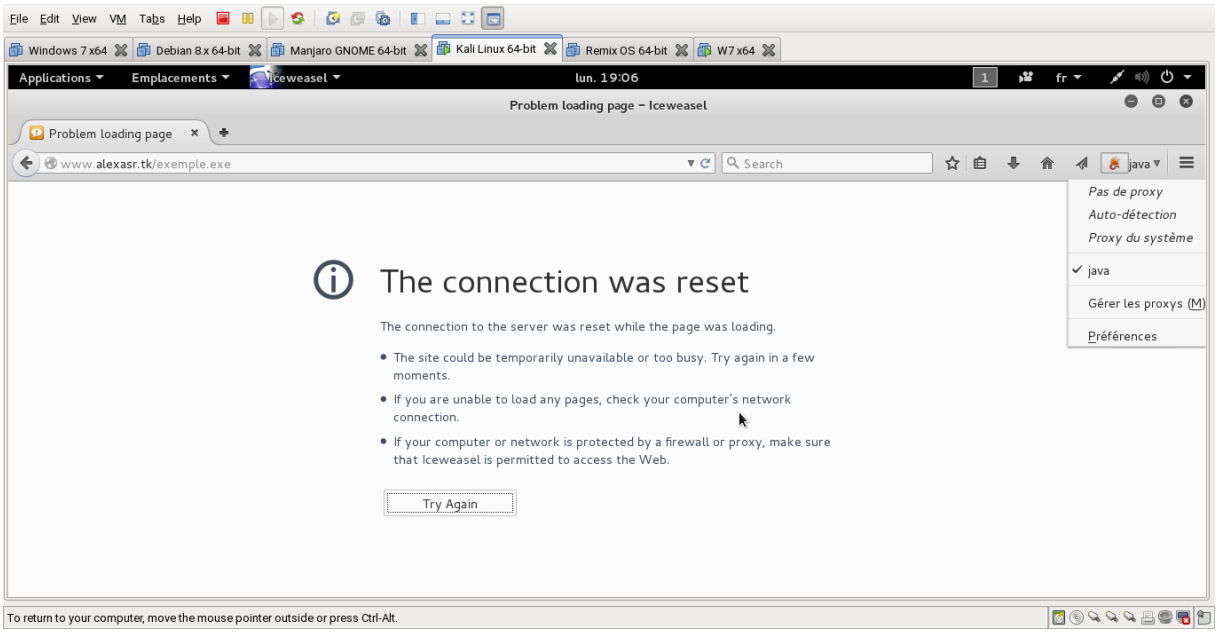
```
New connection to the proxy. 1 client(s) connected to the proxy.
Addresses allowed to reach blacklisted content :
[172.16.108.140, 172.16.127.1]

Connection established with: 172.16.108.140
Start recepting HTTP request of the client.
* * * GET http://perdu.com/ HTTP/1.1
* * * Host: perdu.com
* * * User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.7.1
* * * Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
* * * Accept-Language: en-US,en;q=0.5
* * * Accept-Encoding: gzip, deflate
* * * Connection: keep-alive
* * *
End recepting HTTP request of the client.
172.16.108.140: requête pour : http://perdu.com/
172.16.108.140: hôte distant : perdu.com
172.16.108.140: port distant : 80
172.16.108.140 autorisé à accéder au site.
Connection from 172.16.108.140 accepted.
Sending a request for URL: http://perdu.com/
        Accept:*/*
        User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:38.0) Gecko/20100101 Firefox/38.0 Iceweasel/38.7.1
        Type is: text/html
        Content length: 204
        Allowed user interaction: false
        Content encoding: null
        Content type: text/html


End of connection to the proxy from 172.16.108.140
Number of connected client(s) : 0
```

Accès à un fichier sur liste noire

Pour ce test, le client va demander à télécharger un fichier *exemple.exe*, dont l'extension est sur la liste noire des fichiers non-autorisés à être téléchargés :

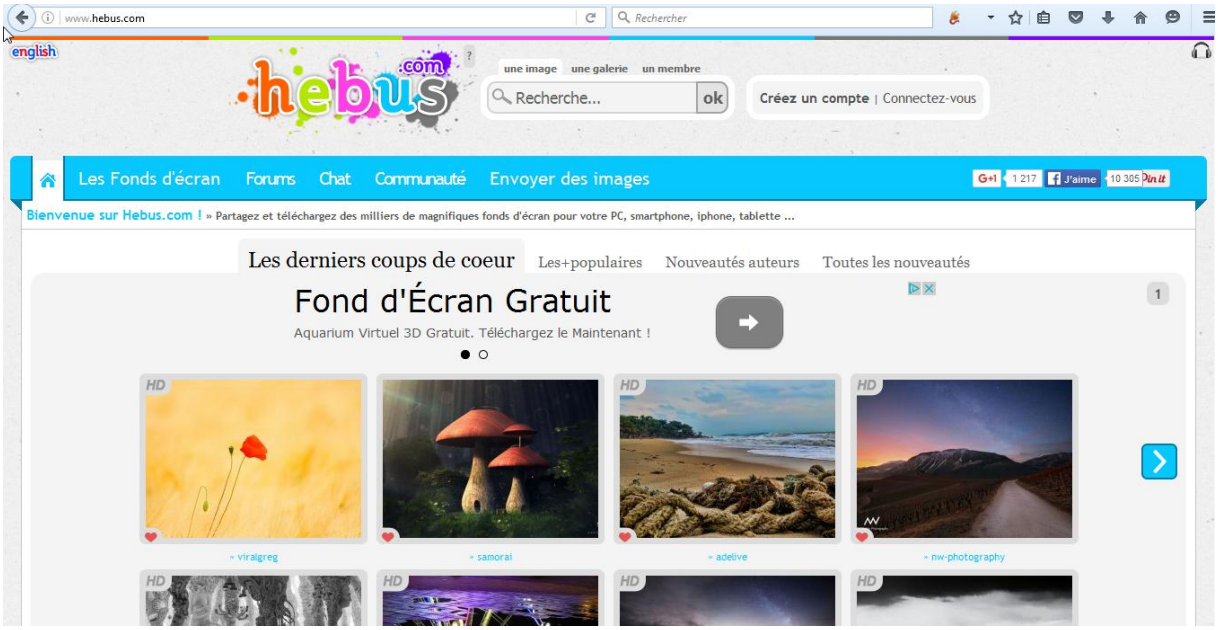


Et la sortie du serveur proxy :

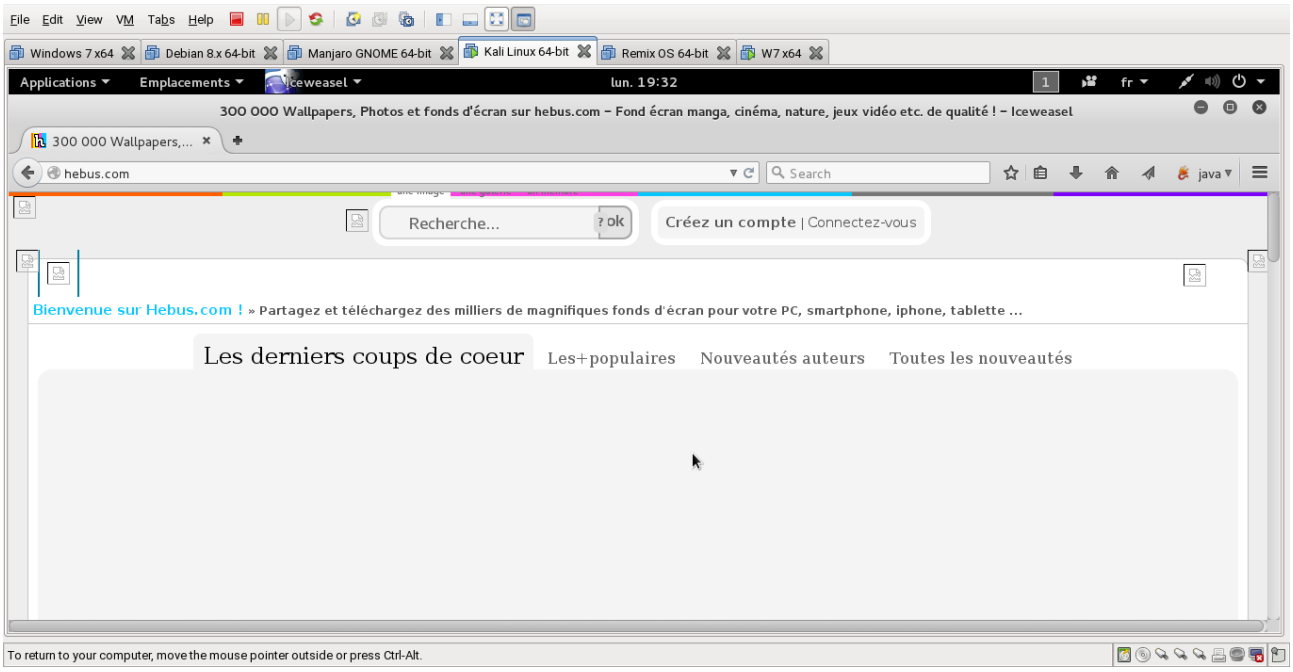
```
/!\ .exe file detected.
Downloading [.json, .png, .jpg, .jpeg, .exe] is forbidden.
```

Le serveur proxy a détecté que le client a demandé un fichier dont l'extension est sur liste noire. Une erreur apparaît sur la sortie standard mais pas sur l'écran de l'utilisateur. En effet, lorsque le client demande une page web, il n'y a en général pas qu'un seul type de ressource (il peut y avoir du HTML, du CSS, du JavaScript, du PHP, des images, des scripts CGI etc …). Si le proxy bloque certains fichiers, les autres sont néanmoins téléchargés.

Par exemple, voici une capture d'écran du *hebus.com* (site d'hébergement d'images de fonds d'écran):



Et maintenant une capture d'écran du même site avec tous les fichiers PNG, JPG et JPEG sur liste noire :



Les images n'ont pas été téléchargées mais le reste du site si. Donc il apparaît absurde d'envoyer un message d'erreur sur le navigateur du client en plein écran comme précédemment pour lui informer que certains fichiers ne peuvent pas être téléchargés.

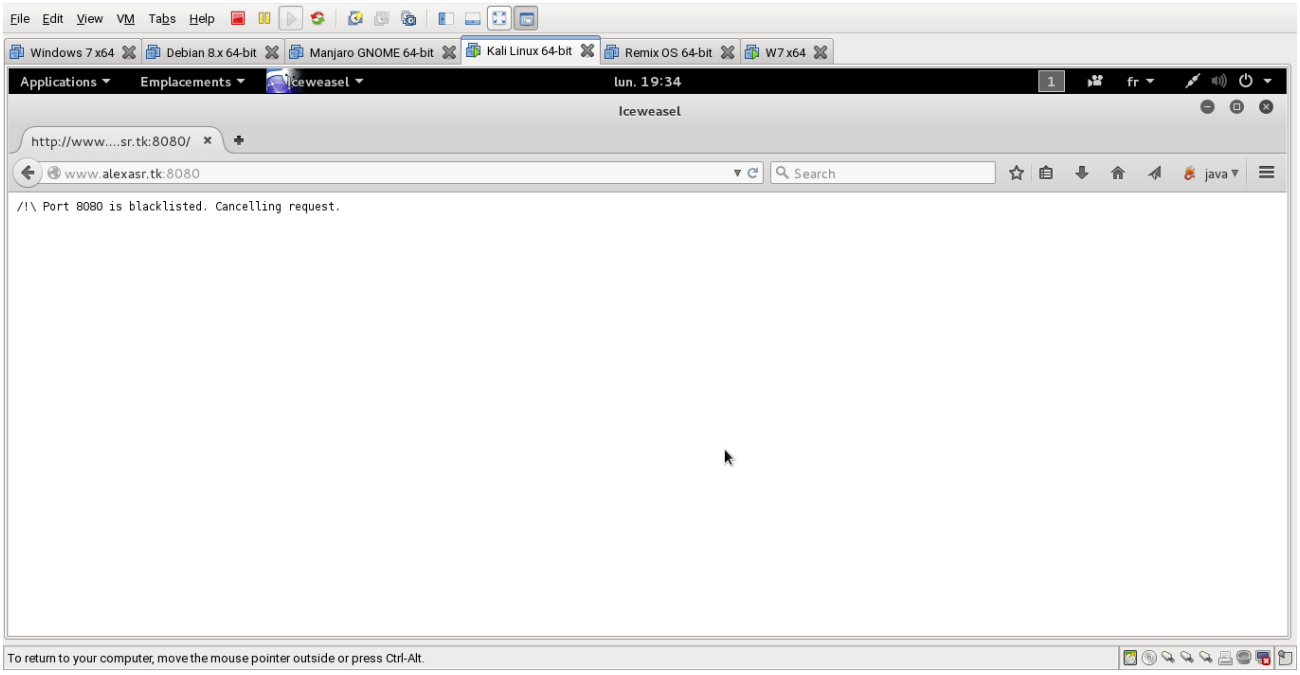
En revanche des messages d'erreurs apparaissent sur stderr :

```
/!\ .png file detected.
Downloading [.json, .png, .jpg, .jpeg, .exe] is forbidden.

/!\ .jpg file detected.
Downloading [.json, .png, .jpg, .jpeg, .exe] is forbidden.
```

Accès à un port sur liste noire

Notre client va maintenant essayer de se connecter à un site en utilisant un port sur liste noire (le 8080) :



Le serveur proxy a refusé de se connecter au port 8080 et en a averti le client en lui affichant un message d'erreur sur son navigateur.

Et sur stderr :

```
/!\ Port 8080 is blacklisted. Cancelling request.
```

Conclusion

Ce projet nous a permis de redécouvrir le fonctionnement des sockets de communication mais cette fois dans un nouveau langage qu'est Java.

Le projet est aussi formateur dans la mesure où nous travaillons en équipe, donc il oblige le dialogue et la communication entre les membres. Nous avons en plus étés contraints de respecter un calendrier, donc il nous revenait de gérer efficacement le temps qui nous était imparti.