TP Programmation socket

Sommaire

Les objectifs de ce tp sont de mettre en oeuvre sous Linux la programmation réseau en utilisant l'interface socket.

Remarque : les TP *ont pour but d'établir ou de renforcer vos compétences pratiques. Vous pouvez penser que vous comprenez tout ce que vous lisez ou tout ce que vous a dit votre enseignant mais la répétition et la pratique sont nécessaires pour développer des compétences en programmation. Ceci est comparable au sport ou à la musique ou à tout autre métier demandant un long entraînement pour acquérir l'habileté nécessaire. Imaginez quelqu'un qui voudrait disputer une compététion dans l'un de ces domaines sans pratique régulière. Vous savez bien quel serait le résultat.*

Questions de révision

Question 1. Qu'est-ce qu'une *socket* ?

Question 2. Quelles sont les trois caractéristiques d'une *socket* ?

Question 3. Quelles sont les deux informations qui définissent un point de communication en IPv4 ?

Question 4. Comment le serveur connaît-il le port utilisé par le client ?

Question 5. Comment le client connaît-il le port utilisé par le serveur ?

Question 6. À quelle couche du modèle DoD est reliée l'interface de programmation *socket* ?

Question 7. Quel protocole de niveau Transport permet d'établir une communication en mode connecté ?

Question 8. Quel protocole de niveau Transport permet d'établir une communication en mode non connecté ?

Question 9. Quel est l'ordre des octets en réseau ?

Question 10. À quels protocoles correspond le domaine `PF_INET` ? Est-ce le seul utilisable avec l'interface socket ? En citer au moins un autre.

Travail demandé

La programmation réseau nécessite l'utilisation de l'interface **socket** pour faire communiquer des processus. Une socket est un point de communication par lequel un processus peut donc émettre et recevoir des données. Ce point de communication devra être relié à une adresse IP et un numéro de port dans le cas des protocoles Internet.

Dans les séquences 1 et 2, on utilisera l'outil `netcat`.

`netcat`, également abrégé `nc`, est un utilitaire permettant d'ouvrir des connexions réseau, que ce soit UDP ou TCP. Il est conçu pour être incorporé aisément dans un large panel d'applications. En raison de sa polyvalence, netcat est aussi appelé le « couteau suisse du TCP/IP ». Il existe sur plusieurs systèmes d'exploitation et s'utilise en ligne de commande. Mais la flexibilité de cet outil permet des usages plus exotiques : transferts de fichiers, backdoor, serveur proxy basique, ou encore messagerie instantanée. Aide : `nc.traditional -h` ou `man nc`

Séquence 1 : client/serveur TCP

On vous fournit le code source de base d'un client et d'un serveur TCP (`http://tvaira.free.fr/bts-sn/reseaux/tp-sockets/src-tp-sockets.zip`).

Les explications détaillées sur le code source de base du client et du serveur TCP sont fournies dans le support de cours (`http://tvaira.free.fr/bts-sn/reseaux/cours/cours-programmation-sockets.pdf`).

Question 1. Analyser et compléter le client `clientTCP1.c` afin qu'il puisse créer une socket TCP en mode connecté. Donner les modifications apportées.

Question 2. Tester le client `clientTCP1.c` fourni en utilisant `netcat` (`nc` ou `nc.traditional`) en serveur TCP. Donner les deux commandes utilisées. *Remarque : Le client attend une réponse du serveur sous la forme d'une chaîne de caractères inférieure à 256.*

Question 3. Modifier dans le client `clientTCP1.c` l'adresse IP et le numéro de port du serveur distant. Tester le nouveau client en utilisant `netcat` en serveur. Donner les deux commandes utilisées et les lignes modifiées dans le programme.

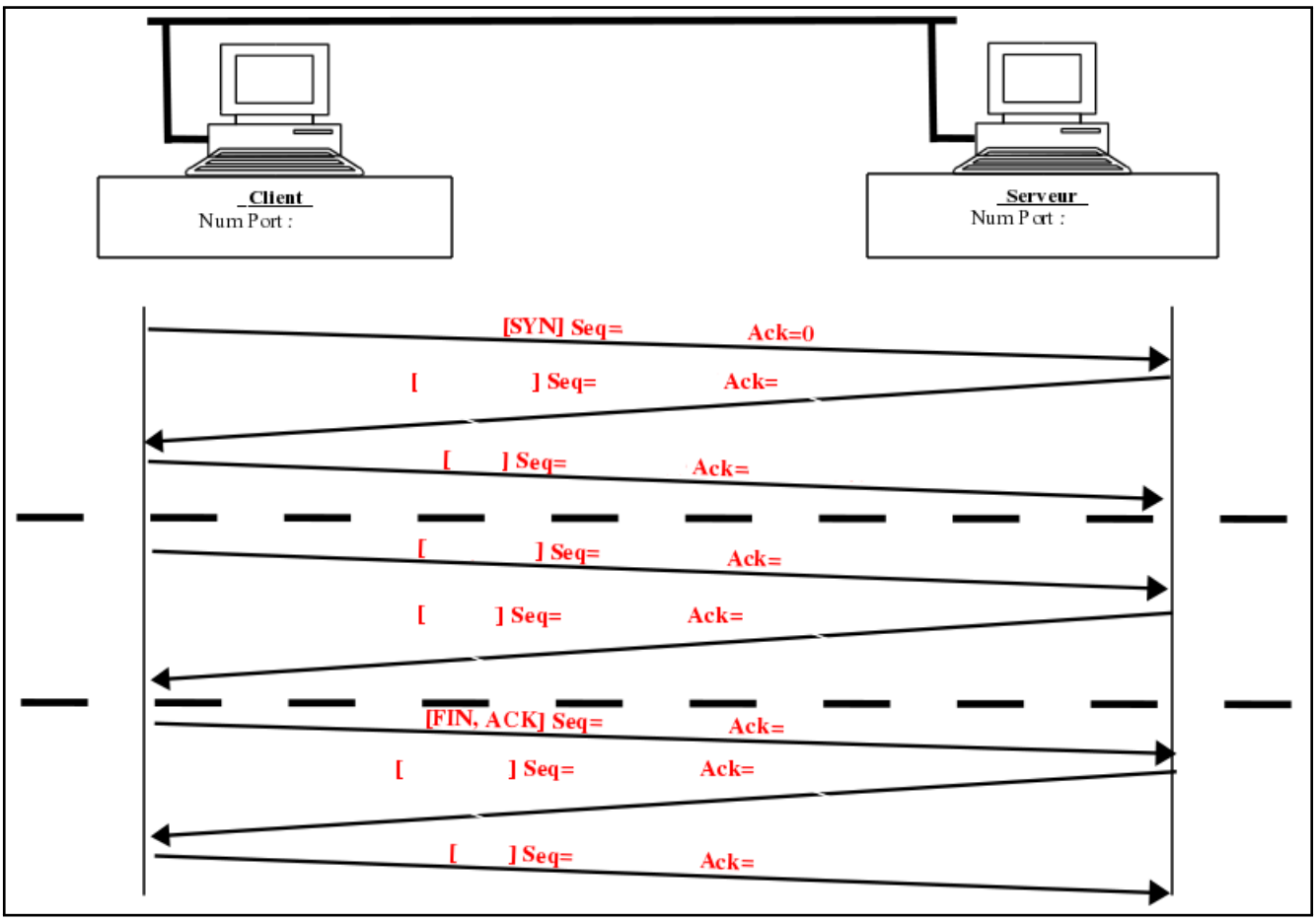
Question 4. Tester ensuite le serveur `serveurTCP1.c` fourni en utilisant `netcat` en client. Donner les deux commandes exécutées. *Remarque : Le serveur attend une « requête »du client sous la forme d'une chaîne de caractères inférieure à 256.*

Question 5. Lancer deux serveurs `serveurTCP1.c` successivement. Que se passe-t-il ? Expliquer.

Question 6. Tester maintenant le client `clientTCP2.c` fourni avec le serveur `serveurTCP2.c` fourni. Tester le client avec une adresse IP puis avec un nom de machine. Donner les commandes exécutées côté client.

Question 7. Quelle est la différence entre les serveurs `serveurTCP1.c` et `serveurTCP2.c` fournis ?

Question 8. Observer le dialogue entre le client et le serveur avec `wireshark`. Donner les numéros de séquence utilisés côté client et serveur. Compléter le diagramme d'échange ci-dessous entre le client et le serveur.



Séquence 2 : client/serveur UDP

On vous fournit le code source de base d'un client et d'un serveur UDP (`http://tvaira.free.fr/bts-sn/reseaux/tp-sockets/src-tp-sockets.zip`).

Les explications détaillées sur le code source de base du client et du serveur UDP sont fournies dans le support de cours (`http://tvaira.free.fr/bts-sn/reseaux/cours/cours-programmation-sockets.pdf`).

Question 9. Tester le client `clientUDP.c` fourni en utilisant `netcat` (`nc` ou `nc.traditional`) en serveur UDP. Donner les deux commandes utilisées. Que se passe-t-il pour le client si il n'y a pas de serveur UDP en attente ? Obtient-on une erreur côté client ?

Question 10. Analyser le serveur `serveurUDP.c` fourni. Quel est son numéro de port d'écoute ?

Question 11. Tester ensuite le serveur `serveurUDP.c` fourni en utilisant `netcat` en client. Donner la commande `netcat` exécutée.

Question 12. Dans quel fichier trouve-t-on les noms de service et leur numéro de port ?

Question 13. Tester maintenant le client `clientUDP.c` fourni avec le serveur `serveurUDP.c` fourni. Tester le client avec le numéro de port distant puis avec son nom de service. Donner les deux commandes utilisées côté client.

Question 14. Observer le dialogue entre le client et le serveur avec `wireshark`. Qui a choisi les numéros de port du client et du serveur ?

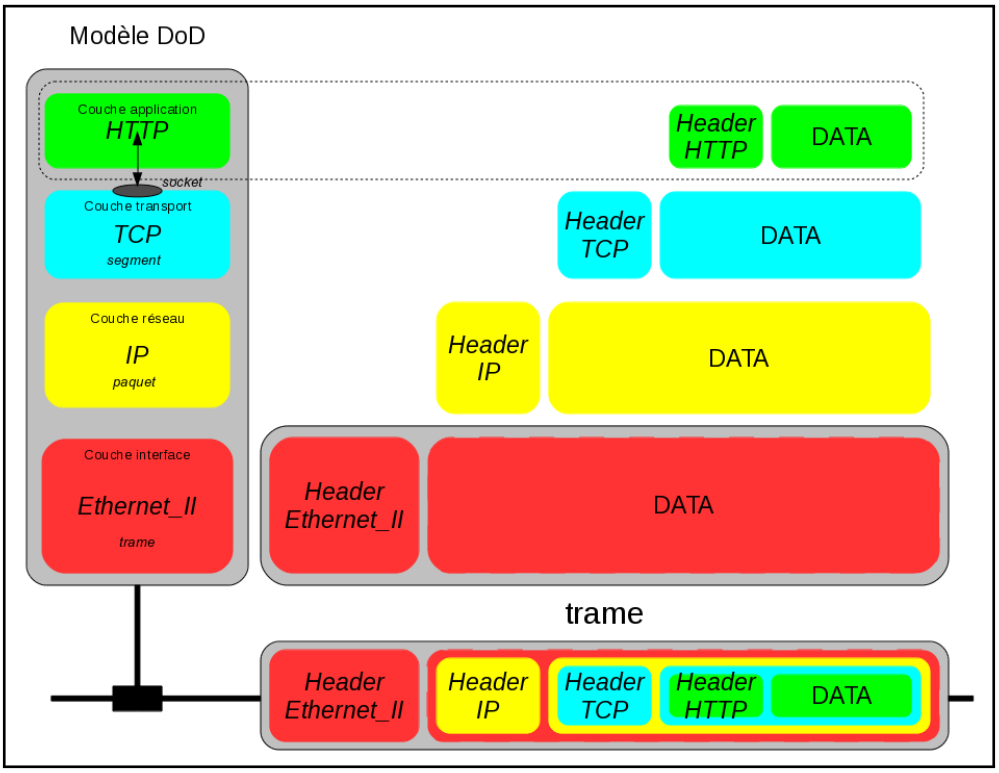
Bonus

Le *framework* Qt fournit le module `QtNetwork` qui contient de nombreuses classes dont certaines permetttant de mettre en œuvre les sockets dans un programme.

– Liste des classes Qt4 : `http://doc.qt.io/archives/qt-4.8/qtnetwork-module.html`
– Liste des classes Qt5 : `https://doc.qt.io/qt-5.10/qtnetwork-module.html`

Séquence 3 : client Qt HTTP

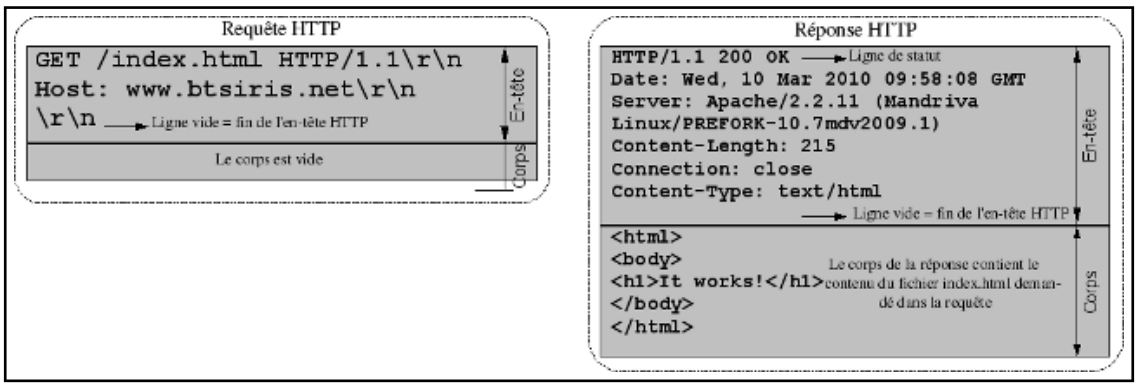
L'objectif de cette séquence est d'écrire un client Qt utilisant le protocole HTTP de couche APPLICATION.



Exemple d'encapsulation du protocole HTTP

Les protocoles de la couche APPLICATION privilégie l'échange de données sous forme ASCII (caractère sur 8 bits soit 1 octet) notamment parce qu'il n'y pas de risque d'inversion dans l'ordre des octets. La plupart des protocoles de la couche APPLICATION sur Internet utilise des délimiteurs "$\backslash r \backslash n$" pour structurer leurs en-têtes (*header*) de protocole.

Voici un exemple de requête (et de réponse) HTTP :



Exemple de requête et réponse HTTP version 1.1

Pour simplifier la requête émise, on utilisera côté client la version 1.0 du protocole HTTP. La requête sera alors la suivante : "GET / HTTP/1.0\r\n\r\n". Présentation du protocole HTTP : https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol.

Qt fournit un module QtNetwork qu'il faut activer dans son fichier de projet .pro :

```
QT += network
```

Dans ce module, Qt fournit de nombreuses classes dont la classe QTcpSocket :

- La classe QTcpSocket Qt4 (en) : http://doc.qt.io/qt-4.8/qtcpsocket.html
- La classe QTcpSocket Qt5 (en) : http://doc.qt.io/qt-5/qtcpsocket.html

Le module QtNetwork fournit aussi la classe QHostAddress pour gérer les adresses IPv4 ou IPv6. On peut l'utiliser avec les classes QTcpSocket, QTcpServer et QUdpSocket pour se connecter à un hôte ou pour configurer un serveur. QHostAddress ne fait pas de recherche DNS. QHostInfo sera alors nécessaire pour une résolution de nom.

La création d'une socket TCP et sa connexion vers un serveur TCP suit la procèdure suivante :

```
#include <QtNetwork>
#include <QDebug>

QTcpSocket *socket; // une socket TCP
socket = new QTcpSocket(this);

QString adresse; // l'adresse IP ou le nom du serveur distant
int port; // numéro de port du serveur distant

// Connexion vers le serveur distant
socket->connectToHost(adresse, port);

connect(socket, SIGNAL(connected()), this, SLOT(estConnecte()));

qDebug() << Q_FUNC_INFO << QString::fromUtf8("Connexion en cours ...");
if(!socket->waitForConnected(5000)) // on attendra 5 secondes
{
   qDebug() << Q_FUNC_INFO << QString::fromUtf8("Erreur impossible de se connecter au
       serveur (") << adresse << QString::fromUtf8(":") << QString::number(port) << QString::
       fromUtf8(")") << QString::fromUtf8(" : ") << socket->errorString();
```



```
   return false;
}

connect(socket, SIGNAL(disconnected()), this, SLOT(estDeconnecte()));
connect(socket, SIGNAL(readyRead()), this, SLOT(recevoir()));

qDebug() << Q_FUNC_INFO << socket->isOpen();
```

Pour envoyer des données, on utilisera par exemple la méthode `write()` de la classe `QTcpSocket` :

```
QString requete = "Hello world !\n";

socket->write(qPrintable(requete));
```

Pour recevoir des données, on pourra utiliser la méthode `readAll()` de la classe `QTcpSocket` :

```
QByteArray donnees;
donnees = socket->readAll();

qDebug() << Q_FUNC_INFO << QString::fromUtf8("Réception d'une réponse du serveur (") <<
   socket->peerAddress().toString() << QString::fromUtf8(":") << QString::number(socket->
   peerPort()) << QString::fromUtf8(")");

QString reponse(donnees.constData());
qDebug() << Q_FUNC_INFO << reponse;
```

La réception se fera dans un *slot* connecté au *signal* `readyRead()`.

Pour tester le client HTPP, nous avons besoin de joindre un serveur HTTP. Il est possible qu'un serveur HTTP (Apache) s'exécute déjà en local.

```
$ netstat -tan | grep 80
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
```

Voici un exemple de ce que vous pouvez obtenir en local :

```
$ ./clientHTTP 127.0.0.1 80
bool Client::demarrer(QString, int) "Connexion en cours ..."
void Client::estConnecte() "Connexion réussie au serveur (" "127.0.0.1" ":" "80" ")"
void Client::envoyer(const QString&) "Envoi d'une requête au serveur (" "127.0.0.1" ":" "80"
     ")"
void Client::envoyer(const QString&) "GET / HTTP/1.0

"
void Client::recevoir() "Réception d'une réponse du serveur (" "127.0.0.1" ":" "80" ")"
void Client::recevoir() "HTTP/1.1 200 OK
Date: Sun, 18 Mar 2018 08:49:10 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Mon, 25 Nov 2013 15:26:25 GMT
Accept-Ranges: bytes
Content-Length: 212
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```



```
<html>
<body>
<h1>It works!</h1>
</body>
</html>
"
void Client::estDeconnecte() "Déconnexion du serveur (" "127.0.0.1" ":" "80" ")"
```

On peut tester avec un serveur web Internet :

```
$ ./clientHTTP www.google.fr 80
bool Client::demarrer(QString, int) "Connexion en cours ..."
void Client::estConnecte() "Connexion réussie au serveur (" "216.58.213.67" ":" "80" ")"
void Client::envoyer(const QString&) "Envoi d'une requête au serveur (" "216.58.213.67" ":"
    "80" ")"
void Client::envoyer(const QString&) "GET / HTTP/1.0

"
void Client::recevoir() "Réception d'une réponse du serveur (" "216.58.213.67" ":" "80" ")"
void Client::recevoir() "HTTP/1.0 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Location: http://www.google.fr/?gfe_rd=cr&dcr=0&ei=PiiuWqnOFJPS8AeD3pOACQ
Content-Length: 268
Date: Sun, 18 Mar 2018 08:50:06 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.fr/?gfe_rd=cr&amp;dcr=0&amp;ei=PiiuWqnOFJPS8AeD3pOACQ">here</A>.
</BODY></HTML>
"
void Client::estDeconnecte() "Déconnexion du serveur (" "216.58.213.67" ":" "80" ")"
```

On placera l'envoi de la requête HTTP dans le *slot* `estConnecte()`. Dans le *slot* `estDeconnecte()`, on mettra fin au programme avec `qApp->quit()`. Le client peut demander la déconnexion en appelant la méthode `disconnectFromHost()` ou `close()`.

Question 15. Compléter le client `clientHTTP` fourni et tester le.

Question 16. Tester le client avec une requête `"GET / HTTP/1.1`$\backslash r\backslash n\backslash r\backslash n$`"`. Quelle réponse obtient-on ?

Ici, nous avons utilisé directement la classe `QTcpSocket` pour mettre en oeuvre une socket TCP. Qt fournit aussi des classes de niveau Application comme : `QWebView` pour afficher un document HTML, `QNetworkAccessManager` pour envoyer des requêtes et recevoir des réponses, ...

Séquence 4 : serveur Qt HTTP

L'objectif de cette séquence est d'écrire un serveur Qt HTTP (basique).

Qt fournit la classe `QTcpServer` :

– La classe `QTcpServer` Qt4 (en) : `http://doc.qt.io/qt-4.8/qtcpserver.html`
– La classe `QTcpServer` Qt5 (en) : `http://doc.qt.io/qt-5/qtcpserver.html`

La création du serveur TCP et sa mise en attente de demande suit la procèdure suivante :

```
QTcpServer *serveur = new QTcpServer(this);

// Démarrage du serveur sur toutes les IP locales disponibles et sur le port PORT_SERVEUR
if (!serveur->listen(QHostAddress::Any, PORT_SERVEUR))
{
   qDebug() << Q_FUNC_INFO << QString::fromUtf8("Erreur démarrage serveur : ") << serveur->
      errorString();
}
else
{
   qDebug() << Q_FUNC_INFO << QString::fromUtf8("Démarrage du serveur sur le port ") <<
      QString::number(serveur->serverPort());
   connect(serveur, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
}
```

Le *slot* pour gérer la connexion d'un nouveau client devra récupérer la *socket* de dialogue de la manière suivante :

```
QTcpSocket *nouveauClient = serveur->nextPendingConnection();

qDebug() << Q_FUNC_INFO << QString::fromUtf8("Connexion client (") << nouveauClient->
   peerAddress().toString() << QString::fromUtf8(":") << QString::number(nouveauClient->
   peerPort()) << QString::fromUtf8(")");

// TODO : si besoin stocker la socket client dans une QList

connect(nouveauClient, SIGNAL(readyRead()), this, SLOT(recevoir()));
connect(nouveauClient, SIGNAL(disconnected()), this, SLOT(deconnexionClient()));
```

Dans le *slot* connecté au signal `readyRead()`, on pourra récupérer la *socket* client de la manière suivante :

```
// On détermine quel client envoie le message (recherche du QTcpSocket du client)
QTcpSocket *client = qobject_cast<QTcpSocket *>(sender());

if (client == 0) // Si par hasard on n'a pas trouvé le client à l'origine du signal, on sort
   return;

// ...
```

Concernant la gestion du protocole HTTP par notre serveur, on va se contenter d'envoyer systèmatiquement une réponse basique en HTTP :

```
HTTP/1.1 200 OK
Server: (le nom et la version de notre serveur !)
Connection: close
Content-Type: text/html
Content-Length: 55
```



```
<html><body><h1>Un super serveur !</h1></body></html>
```

On peut ici utiliser un vrai client HTTP (`firefox` par exemple) pour tester notre serveur et on obtiendra ceci :

Notre serveur HTTP en action !

Comme vous pouvez le constatez ici, le serveur HTTP écoute sur le port 5000. Normalement, il devrait le faire sur le port 80 qui est le port réservé pour le service HTTP. Il faut savoir que l'utilisation des ports en-dessous de 1024 n'est possible que pour des processus "administrateurs".

Question 17. Écrire le serveur `serveur` et tester le sur le port 5000.

Question 18. Que faudrait-il faire pour l'exécuter sur le port 80 ? Tester.

Question 19. Ici le serveur renvoie la valeur 200 dans la réponse HTTP. À quoi correspond cette valeur 200 dans le protocole HTTP ?

Question 20. Le client émet une requête GET pour obtenir un document hébergé sur le serveur. Si ce document n'existe pas sur le serveur, quelle valeur doit-il alors renvoyer dans la réponse HTTP ?

