UNIVERSITÉ TOULOUSE III PAUL SABATIER — UPSSITECH ÉCOLE D'INGÉNIEURS

Programmation Sockets

Ph. Truillet

Septembre 2021 – v. 2.4

0. déroulement du TP

1. comprendre le fonctionnement global des Sockets et la notion de port de communication ;
2. appréhender l'API sockets en java (pour TCP et UDP)
3. écrire et utiliser un client TCP ;
4. écrire des clients et des serveurs en utilisant TCP pour des protocoles connus
5. écrire une application utilisant UDP.

1. introduction

1.1 API socket

Unix est un système multi-tâches et multi-utilisateurs. Les processus peuvent coopérer entre eux. Différents outils de communication inter-processus ont été développés : les IPC (Inter Process Communication). La communication inter-processus peut se faire à travers les signaux et les pipes. Mais tandis que la version d'ATT était orientée gestion de ressources (sémaphores, messages, mémoire partagée), l'université de Berkeley a développé une version orientée réseaux en implémentant les protocoles Internet ARPA/DOD et en offrant l'interface des « **sockets** ».

Les sockets forment une bibliothèque (appelée API : **A**pplication **P**rogramming **I**nterface). Elles offrent aux programmeurs une interface entre le programme d'application et les protocoles de communication. En aucun cas, les sockets ne forment une norme de communication ou une couche de protocole à l'instar de TCP/IP. Les sockets (*prises de raccordement*) forment un mécanisme de communication bidirectionnel inter-processus dans un environnement distribué ce qui n'est pas le cas des autres outils tels que les pipes. Ils permettent évidemment la communication inter-processus à l'intérieur d'un même système.

L'interface des « sockets » n'est pas liée à une pile de protocoles spécifique. Dans ce document, nous nous intéresserons à l'utilisation des **sockets dans le monde IP avec le langage java (mais vous pourrez écrire vos programmes en Python si vous le souhaitez)**.

En java, l'API est contenue dans le paquetage *java.net*. Il existe bien entendu des fonctions permettant d'utiliser les sockets soit dans le monde connecté **TCP** (le protocole établit alors une connexion virtuelle et se charge alors de maintenir l'intégrité de la communication et de gérer les erreurs de transmission) soit dans le monde non connecté **UDP** (Ce protocole fait l'envoi au mieux [« *best effort* »]. Dans ce cadre, c'est à l'application de maintenir la qualité de la transmission.
Java implémente aussi des sockets **de plus haut niveau** (`URLConnection`, `HttpURLConnection`) permettant d'abstraire les communications de plus bas niveau.

1.2 notion de port de communication

Un service est rendu par un programme appelé « **serveur** » sur une machine. Ce service est accessible à travers un réseau TCP/IP par un port. Un port est identifié par un entier (codé sur 16 bits).

- Les ports numérotés de **0** à **511** sont les « *well known ports* » (ports bien connus) de l'architecture TCP/IP. Ils donnent accès aux services standards de l'interconnexion : transfert de fichiers (ftp port 21), terminal (telnet port 23), shell sécurisé (ssh, port 22), courrier (SMTP port 25), serveur web (http port 80), …
- De **512** à **1023**, on trouve les services Unix.
- Au delà, (**1024** ...) ce sont les ports « *utilisateurs* » disponibles pour placer un service applicatif quelconque.
- Un service est souvent connu par un nom (ftp, …). La correspondance entre nom et numéro de port est donnée par le fichier `/etc/services` sous unix.

2. sockets TCP (Transmission Control Protocol) – un peu d'API Java

Le protocole TCP fait partie de la couche transport de la pile de protocoles TCP/IP, fiable en mode connecté et est détaillé dans la **RFC 793** (`http://tools.ietf.org/html/rfc793`).
Programmer avec les sockets implique généralement la programmation d'un « **client** » (qui se connectera à un serveur) et/ou d'un « **serveur** » (qui rendra des services à un ou des clients.)
De manière plus basique, on utilisera la classe **`Socket`** pour les clients et **`SocketServer`** + **`Socket`** pour les serveurs.

2.1 partie « client »

La classe **`Socket`** représente en Java uniquement les sockets utilisant le protocole TCP (garantissant la fiabilité du service) et fonctionnant en mode client (pas d'attente de connexion via un `accept`).
Pour se connecter à un serveur, le client ouvre une **socket** (sur une machine à un port bien identifié), créé les flux en entrée et en sortie sur cette socket puis lit et écrit comme on le ferait dans un fichier jusqu'à la fermeture de la socket.

2.1.1 constructeurs

- **`public Socket (String hote, int port) throws UnknownHostException, IOException`**
  Ce constructeur crée une socket TCP sur le port indiqué de l'hôte visé et tente de s'y connecter. Le premier paramètre de ce constructeur représente le nom de la machine serveur. Si l'hôte est inconnu ou que le serveur de noms de domaine est inopérant, le constructeur générera une `UnknownHostException`. Les autres causes d'échec, qui déclenchent l'envoi d'une IOException sont multiples : machine cible refusant la connexion sur le port précisé ou sur tous les ports, problème lié à la connexion Internet, erreur de routage des paquets, etc.
  Voici un exemple d'utilisation de ce constructeur :
  ```
  Socket laSocket = new Socket("www.irit.fr", 80);
  ```

- **`public Socket(InetAddress addresse, int port) throws IOException`**
  Ce constructeur fonctionne comme le premier, mais prends comme premier paramètre une instance de la classe `InetAddress`. Ce constructeur provoque une IOException lorsque la tentative de connexion échoue mais il ne renvoie pas d'`UnknownHostException` puisque cette information est connue lors de la création de l'objet `InetAddress`.

- **`public Socket(String hote, int port, InetAddress addresseLocale, int portLocal) throws IOException`**
  Comme les deux précédents, ce constructeur crée une socket sur le port spécifié de l'hôte visé et tente de s'y connecter. Le numéro du port et le nom de la machine sont fournis dans les deux premiers paramètres et la connexion s'établit à partir de l'interface réseau physique (choix d'une carte réseau sur un système possédant plusieurs accès réseau) ou virtuelle (système multi-adresse) et du port local indiqués dans les deux derniers paramètres. Si *portLocal* vaut **0**, la méthode (comme les deux premières d'ailleurs) choisit un port disponible (parfois appelé port anonyme) compris entre 1024 et 65 535.
  Comme la première méthode cette méthode peut générer une `IOException` en cas de problème de connexion et lorsque le nom d'hôte donné n'est pas correct (il s'agit dans ce cas d'une `UnknownHostException`).

- **`public Socket(InetAddress address, int port, InetAddress addresseLocale, int portLocal) throws IOException`**
  Cette méthode est identique à la précédente à la seule différence que le premier paramètre est, comme pour la seconde méthode, un objet `InetAddress`.

- **`protected Socket()`**
- **`protected Socket(SocketImpl impl) throws SocketException`**
  Ces deux derniers constructeurs sont protégés. Ils créent une socket sans le connecter. On utilise ces constructeurs pour implanter une socket originale qui permettra par exemple de chiffrer les transactions ou d'interagir avec un proxy.

Il existe de plus deux autres constructeurs qui permettent le choix du protocole via un booléen mais ces méthodes sont dépréciées et ne doivent pas être utilisées. Pour créer une socket utilisant UDP on utilisera **`DatagramSocket`**.

2.1.2. méthodes informatives

- **`public InetAddress getInetAddress ()`**
- **`public int getPort ()`**
  Ces méthodes renvoient l'adresse Internet et le port distant auquel la socket est connectée.

- **`public InetAddress getLocalAddress ()`**
- **`public int getLocalPort ()`**
  Ces méthodes renvoient l'adresse Internet et le port local sur lequel la socket est connectée.

- **`public InputStream getInputStream () throws IOException`**
  Cette méthode renvoie un flux d'entrées brutes grâce auquel un programme peut lire des informations à partir d'un socket. Il est d'usage de lier cet `InputStream` à un autre flux offrant davantage de fonctionnalités (un `DataInputStream` par exemple) avant d'acquérir les entrées.
  Voici un exemple d'utilisation de cette méthode :
  ```
  DataInputStream fluxEnEntree = new DataInputStream(leSocket.getInputStream());
  ```

- **`public OutputStream getOutputStream () throws IOException`**
  Cette méthode renvoie un flux de sortie brutes grâce auquel un programme peut écrire des informations sur un socket. Il est d'usage de lier cet `OutputStream` à un autre flux offrant davantage de fonctionnalités (un `DataOutputStream` par exemple) avant d'émettre des données.
  Voici un exemple d'utilisation de cette méthode :
  ```
  DataOutputStream fluxEnSortie=new DataOutputStream(leSocket.getOutputStream());
  ```

2.1.3. fermeture d'une socket

- **`public void close() throws IOException`**

Bien que Java ferme tous les sockets ouverts lorsqu'un programme se termine ou bien lors d'un « *garbage collector* », il est fortement conseillé de fermer explicitement les sockets dont on n'a plus besoin à l'aide de la méthode `close`.
Une fois une socket fermée, on peut toujours utiliser les méthodes informatives, par contre toute tentative de lecture ou écriture sur les « input/output streams » provoque une `IOException`.

2.1.4. options des sockets

Java permet l'accès à un certain nombre d'options qui modifient le comportement par défaut des sockets. Ces options correspondent à celles que l'on manipule en C via `ioctl` (ou `ioctlsocket` avec Windows).

- **`public boolean getTcpNoDelay() throws SocketException`**

- **`public void setTcpNoDelay(boolean valide) throws SocketException`**
  Valide/invalide l'option `TCP_NODELAY`. Valider `TCP_NODELAY` garantit que les paquets seront expédiés aussi rapidement que possible quelle que soit leur taille.
  Si l'on programme une application très interactive, on pourra valider l'option `TCP_NODELAY`.

- **`public int getSoLinger() throws SocketException`**

- **`public void setSoLinger(boolean valide, int secondes) throws SocketException`**
  L'option `SO_LINGER` indique le comportement à adopter lorsqu'il reste des paquets à expédier au moment de la fermeture de la socket. Par défaut, la méthode `close()` rend la main immédiatement et le système tente d'envoyer le reliquat de données. Si la durée des prolongations secondes vaut 0 et que valide est à vrai, les éventuels paquets restants sont supprimés lorsque la fermeture à lieu. Si la durée est positive et que valide est à vrai, la méthode close se fige pendant le nombre de secondes spécifiées en attendant l'expédition des paquets et la réception de l'acquittement. Une fois les prolongations écoulées, la socket est close et les données restantes ne sont pas transmises.
  Si cette option n'est pas validée (valide est à faux) on obtient le comportement par défaut et l'appel de la méthode `getSoLinger()` retourne –1, sinon la méthode renvoie la durée des prolongations.

- **public int getSoTimeout() throws SocketException**
- **public void setSoTimeout(int ms) throws SocketException**
  Par défaut, lors d'une tentative de lecture sur le flux d'entrée d'un socket, `read()` se bloque jusqu'à la réception d'un nombre d'octets suffisant. Lorsque `SO_TIMEOUT` est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une `InterruptedException`. Le socket reste malgré tout connecté.
  La valeur par défaut est 0 qui signifie, ici, un laps de temps infini. Il peut être préférable de choisir une durée raisonnable (>0) pour éviter que le programme se bloque en cas de problème.

  L'option doit être validée avant l'appel à l'opération bloquante pour être prise en compte. Le paramètre doit être >= 0 sans quoi une exception `SocketException` est générée.

- **public int getSendBufferSize() throws SocketException**
- **public void setSendBufferSize(int size) throws SocketException**
- **public int getReceiveBufferSize() throws SocketException**
- **public void setReceiveBufferSize(int size) throws SocketException**
  Ces méthodes permettent, grâce aux options `SO_SNDBUF` et `SO_RCVBUF`, de préciser une indication sur la taille à allouer aux tampons d'entrée/sortie bas niveaux. Il s'agit uniquement d'une indication donc il est conseillé après l'appel d'une méthode `set...` d'appeler la méthode `get...` correspondante pour vérifier la taille allouée.

2.1.5. méthode surchargée

- **public String toString()**
  La seule méthode surchargée de la classe `Object` est `toString`(). Cette méthode provoque l'affichage d'une chaîne ressemblant à cela :
  `Socket[addr=amber/192.11.4.2,port=80,localport=1167]`

2.2. partie « serveur »

La classe **`ServerSocket`** permet de créer des sockets qui attendent des connections sur un port spécifié et lors d'une connexion retournent une Socket qui permet de communiquer avec l'appelant.

Nota** : En clair, le serveur **écoute l'arrivée d'un client sur un port spécifié. Dès qu'un client arrive, il a deux choix.
*Le premier est de traiter le client et, à la fin de la requête, se remettre à attendre un autre client (cas simple). Le second est de lancer un thread sur une socket dite « **de service** » permettant le traitement du client et se remettre immédiatement à l'écoute (un exemple imagé est une secrétaire d'une grande entreprise qui passe les appels à un correspondant).*

2.2.1. constructeurs

- **public ServerSocket (int port) throws IOException**
  Ce constructeur crée une socket serveur qui attendra les connexions sur le port spécifié. Lorsque l'entier port vaut 0, le port est sélectionné par le système. Ces ports anonymes sont peu utilisés car le client doit connaître à l'avance le numéro du port de destination. Il faut donc un mécanisme, comme le portmapper des RPC, qui permet d'obtenir ce numéro de port à partir d'une autre information.
  L'échec de la création se solde par une IOException (ou à partir de Java 1.1 une `BindException` qui hérite de `IOException`) qui traduit soit l'indisponibilité du port choisi soit, sous UNIX, un problème de droits (sous Unix les ports inférieurs à 1024 ne sont disponibles que pour le super utilisateur).

- **public ServerSocket (int port, int tailleFile) throws IOException**
  Ce constructeur permet en outre de préciser la longueur de la file d'attente des requêtes de connexion. Si une demande de connexion arrive et que la file est pleine la connexion sera refusée sinon elle sera stockée dans la file pour être traitée ultérieurement. La longueur de la file d'attente par défaut (pour le premier constructeur) est 50.
  Lorsque la valeur donnée est supérieure à la limite fixée par le système, la taille maximale est affectée à la file.

- **public ServerSocket (int port, int tailleFile, InetAddress adresseLocale) throws IOException**

Ce constructeur permet en outre de préciser l'adresse Internet locale sur laquelle attendre des connexions. Ainsi on pourra choisir l'une des interfaces réseau de la machine locale si elle en possède plusieurs. Si `adresseLocale` est à `null`, la socket attendra des connexions sur toutes les adresses locales (ce qui est aussi le cas quand on utilise l'un des deux autres constructeurs).

2.2.2. accepter et clore une connexion

- **`public Socket accept () throws IOException`**
  Cette méthode bloque l'exécution du programme serveur dans l'attente d'une demande de connexion d'un client. Elle renvoie un objet `Socket` une fois la connexion établie.
  Si vous ne voulez pas bloquer l'exécution du programme il suffit de placer l'appel de `accept` dans un thread spécifique.

- **`public void close () throws IOException`**
  Cette méthode ferme la socket serveur en libérant le port. Les sockets serveurs sont eux aussi fermés automatiquement par le système à la fermeture de l'application mais il est fortement conseillé de les fermer explicitement.

2.2.3. méthodes informatives

- **`public InetAddress getInetAddress ()`**
- **`public int getLocalPort ()`**
  Ces méthodes renvoient l'adresse Internet et le port locaux sur lequel la socket attend les connexions.

2.2.4. options des sockets serveurs

Seule l'option `SO_TIMEOUT` est disponible pour les sockets serveurs.

- **`public int getSoTimeout() throws SocketException`**
- **`public void setSoTimeout(int ms) throws SocketException`**
  Par défaut, l'appel à `accept()` se bloque jusqu'à la réception d'une demande de connexion. Lorsque SO_TIMEOUT est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une `InterruptedException`. Le socket reste malgré tout connecté. La valeur par défaut est 0 qui signifie, ici, un laps de temps infini ce qui convient à la plupart des serveurs, conçus en général pour s'exécuter indéfiniment. L'option doit être validée avant l'appel de `accept()` pour être prise en compte. Le paramètre doit être >= 0 sans quoi une exception `SocketException` est générée.

2.2.5. méthode surchargée

- **`public String toString()`**
  La seule méthode surchargée de la classe `Object` est `toString()`. Cette méthode provoque l'affichage d'une chaîne ressemblant à cela :

  ```
  ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=80]
  ```

  Sous Java 1.1 et les précédentes versions, `addr` et `port` sont toujours nulles. La seule information utile est donc `localport` qui indique le numéro du port d'attente des connexions. Cette méthode est ici aussi plutôt destinée au débogage.

3. sockets UDP (User Datagram Socket)

Le protocole UDP fait partie de la couche transport de la pile de protocoles TCP/IP et est détaillé dans la **RFC 768** (`http://tools.ietf.org/html/rfc768`)

3.1. la classe DatagramSocket

Ici, tout est simple. Une fois la socket déclarée, il suffit de lire et écrire dedans …

3.1.1. constructeurs

- **`public DatagramSocket() throws SocketException`**
  Ce constructeur crée une socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise cette socket est choisi par le système d'exploitation. De même l'adresse IP qu'utilise cette socket est choisie automatiquement par le système.

- **`public DatagramSocket(int port) throws SocketException`**
  Ce constructeur crée une socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise cette socket est indiqué par le paramètre port. L'adresse IP qu'utilise cette socket est choisie automatiquement par le système.

- **`public DatagramSocket(int port, InetAddress adresse) throws SocketException`**
  Ce constructeur crée une socket UDP qui permet d'envoyer et recevoir des datagrammes UDP. Le port qu'utilise cette socket est indiqué par le paramètre port. L'adresse IP qu'utilise cette socket est indiqué par le paramètre adresse.

Ces trois constructeurs peuvent générer des exceptions de type `SocketException` s'il y a un problème de configuration réseau, si le port choisi est déjà utilisé ou si l'adresse choisie n'est pas l'une des adresses de la machine locale.

3.1.2. émission et réception de Datagrammes

- **`public void send(DatagramPacket p) throws IOException`**
  Envoie un datagramme UDP qui contiendra toutes les informations référencées par l'objet p.

- **`public void receive(DatagramPacket p) throws IOException`**
  Attends un datagramme UDP et lors de sa réception stocke ses informations dans l'objet p.
  Des exceptions de type `IOException` peuvent être générées si un problème d'entrée/sortie survient.

3.1.3. méthodes informatives

- **`public InetAddress getLocalAddress ()`**
- **`public int getLocalPort ()`**
  Ces méthodes renvoient l'adresse Internet et le port locaux que le socket utilise.

3.1.4. fermeture du socket

- **`public void close() throws IOException`**
  Bien que Java ferme tous les sockets ouverts lorsqu'un programme se termine ou bien lors d'un « *garbage collector* », il est fortement conseillé de fermer explicitement les sockets dont on n'a plus besoin à l'aide de la méthode **close**.
  Une fois une socket fermée, on peut toujours utiliser les méthodes informatives, par contre toute tentative d'émission ou de réception de datagrammes UDP provoque une *IOException*.

3.1.5. options

Comme pour les sockets TCP, on peut positionner un certain nombre d'options qui modifient le comportement par défaut des sockets.

- **`public int getSoTimeout() throws SocketException`**
- **`public void setSoTimeout(int ms) throws SocketException`**

Par défaut, lors d'une tentative de réception d'un datagramme UDP, **`receive`** se bloque jusqu'à la réception d'un datagramme. Lorsque `SO_TIMEOUT` est initialisée, cette attente ne dépasse pas le temps imparti, exprimé en millisecondes. Tout dépassement de temps se solde par une **`java.net.SocketTimeoutException`**. Le socket reste malgré tout connectée.
La valeur par défaut est 0 qui signifie, ici, un laps de temps infini. Il peut être préférable de choisir une durée raisonnable (>0) pour éviter que le programme se bloque en cas de problème.
L'option doit être validée avant l'appel à l'opération bloquante pour être prise en compte.
Le paramètre doit être >= 0 sans quoi une exception *`SocketException`* est générée.

- **`public int getSendBufferSize() throws SocketException`**
- **`public void setSendBufferSize(int size) throws SocketException`**
- **`public int getReceiveBufferSize() throws SocketException`**
- **`public void setReceiveBufferSize(int size) throws SocketException`**
  Ces méthodes permettent, grâce aux options `SO_SNDBUF` et `SO_RCVBUF`, de préciser une indication sur la taille à allouer aux tampons d'entrée/sortie bas niveaux. Il s'agit uniquement d'une indication donc il est conseillé après l'appel d'une méthode `set…` d'appeler la méthode `get…` correspondante pour vérifier la taille allouée.
- **`public void setReuseAddress(boolean on) throws SocketException`**
- **`public boolean getReuseAddress() throws SocketException`**
  Ces méthodes permettent, grâce à l'option `SO_REUSEADDR`, de permettre à plusieurs sockets d'utiliser le même numéro de port. L'option est désactivée par défaut. Cette option peut être utilisée lors de diffusions (broadcast ou multicast – voir aussi la classe suivante).

3.2. la classe MulticastSocket

Cette classe permet d'utiliser le multicasting IP pour envoyer des datagrammes UDP à un ensemble de machines repéré grâce à une adresse multicast (classe D dans IP version 4 : de 224.0.0.1 à 239.255.255.255).

3.2.1. constructeurs

Les constructeurs de cette classe sont identiques à ceux de la classe `DatagramSocket`.

3.2.2. abonnement/résiliation

Pour pouvoir recevoir des datagrammes UDP envoyés grâce au multicasting IP il faut s'abonner à une adresse multicast (de classe D pour IP version 4). De même lorsqu'on ne souhaite plus recevoir des datagrammes UDP envoyés à une adresse multicast on doit indiquer la résiliation de l'abonnement.

- **`public void joinGroup(InetAddress adresseMulticast) throws IOException`**
  Cette méthode permet de s'abonner à l'adresse multicast donnée.
  Note : une même socket peut s'abonner à plusieurs adresses multicast simultanément. D'autre part, il n'est pas nécessaire de s'abonner à une adresse multicast si on veut juste envoyer des datagrammes à cette adresse.
  Une `IOException` est générée si l'adresse n'est pas une adresse multicast ou si il y a un problème de configuration réseau.
- **`public void leaveGroup(InetAddress adresseMulticast) throws IOException`**
  Cette méthode permet de résilier son abonnement à l'adresse multicast donnée.
  Une `IOException` est générée si l'adresse n'est pas une adresse multicast, si la socket n'était pas abonnée à cette adresse ou si il y a un problème de configuration réseau.

3.2.3. choix du TTL (Time to Live)

Dans les datagrammes IP se trouve un champ spécifique appelé TTL (Time To Live – durée de vie) qui est normalement initialisé à 255 et décrémenté par chaque routeur que le datagramme traverse. Lorsque ce champ atteint la valeur 0 le datagramme est détruit et une erreur ICMP est envoyé à l'émetteur du datagramme. Cela permet d'éviter que des datagrammes tournent infiniment à l'intérieur d'un réseau IP.
Le multicasting IP utilise ce champ pour limiter la portée de la diffusion. Par exemple avec un TTL de 1 la diffusion d'un datagramme est limitée au réseau local.

- **`public int getTimeToLive() throws IOException`**
- **`public void setTimeToLive(int ttl) throws IOException`**
  Ces méthodes permettent la consultation et la modification du champ TTL qui sera écrit dans les datagrammes envoyés par cette socket.

4. exercices

4.0 des outils utiles

Récupérer les outils suivants intéressants pour la suite du TP :

- **KiTTY** : **`http://www.9bis.net/kitty`**, « fork » du célèbre PuTTY, client telnet et **ssh** pour windows
- **cURL** : **`http://curl.haxx.se`**, interface en ligne de commande permettant de récupérer des ressources accessibles par une URL.
- **HTTPie** : **`https://httpie.io`**,

4.1 un exemple pour débuter en TCP

Récupérer le serveur **Daytime** pour windows (**RFC 867, mai 1983,** `https://datatracker.ietf.org/doc/html/rfc867`) à l'adresse suivante :
`https://github.com/truillet/upssitech/tree/master/SRI/3A/ID/TP/Outils`
Lancez-le.

Récupérer l'exemple de **client** Daytime à l'adresse :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/ClientDaytime.java`

Compiler l'exemple et le tester.

Le service **Daytime** est un service <u>particulier **très basique**</u> (cf. Figure 1). En effet, la connexion au serveur « *vaut* » requête (pas besoin d'envoyer une requête sur la socket). Le serveur, dès la connexion acceptée, renvoie la date et l'heure courante du serveur.

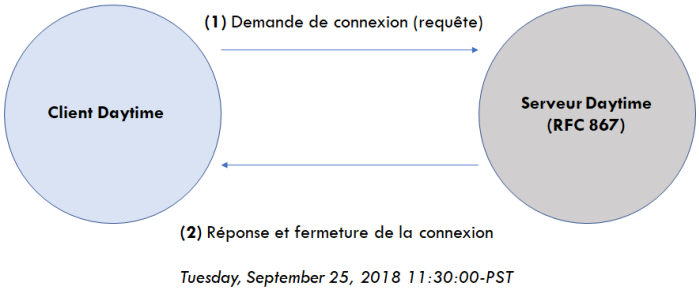


Figure 1 - Requête/Réponse – protocole **Daytime**

4.2 un (premier) client et un serveur

Modifier l'exemple précédent de manière à écrire un **client** « *echo* » [**RFC 862, mai 1983,** `https://datatracker.ietf.org/doc/html/rfc862`] *(qui se connecte sur le port « bien connu » 7)* de telle sorte que l'on puisse entrer le numéro de port et le nom de la machine cible au lancement.

Par exemple : `java Echo` *`IP_machine`* **7** (se connecte à *`IP_machine`* sur le port *7*)

Nota : Vous pouvez tester un serveur **echo** à l'adresse suivante : `upssitech.fr` sur le port **7**

Enfin, écrire un **serveur** « <u>*echo*</u> » qui choisit au lancement automatiquement son numéro de port et l'affiche à l'écran.

4.3 un mini-serveur web

Lire la spécification du protocole **http** [**RFC 1945 pour le protocole http/1.0, mai 1996**] (sur le site du W3C : http://w3c.org) et utiliser l'outil **cURL** (https://curl.se) **ou httpie** (https://httpie.io) pour comprendre le fonctionnement de la requête **GET** :

```
% curl -v http://google.com
*   Trying 216.58.215.46:80...
* TCP_NODELAY set
* Connected to google.com (216.58.215.46) port 80 (#0)
> GET / HTTP/1.1
> Host: google.com
> User-Agent: curl/7.68.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 301 Moved Permanently
< Location: http://www.google.com/
< Content-Type: text/html; charset=UTF-8
< Date: Sun, 26 Sep 2021 14:22:54 GMT
< Expires: Tue, 26 Oct 2021 14:22:54 GMT
< Cache-Control: public, max-age=2592000
< Server: gws
< Content-Length: 219
< X-XSS-Protection: 0
< X-Frame-Options: SAMEORIGIN
<
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="http://www.google.com/">here</A>.
</BODY></HTML>
* Connection #0 to host google.com left intact
```

Ecrire un **mini-serveur web** qui permet la gestion de la commande **GET** du protocole http (sans chiffrement) [voir la RFC correspondante]. Tester votre serveur avec un navigateur « courant » (firefox, IE, safari, chrome, …).

Attention ! En java, l'utilisation de `BufferedReader` peut ne pas fonctionner avec les navigateurs « standards » (*effacement* des retour-chariot), il faut préférer travailler directement avec les classes bas-niveau comme `InputStream` et `OutputStream`.

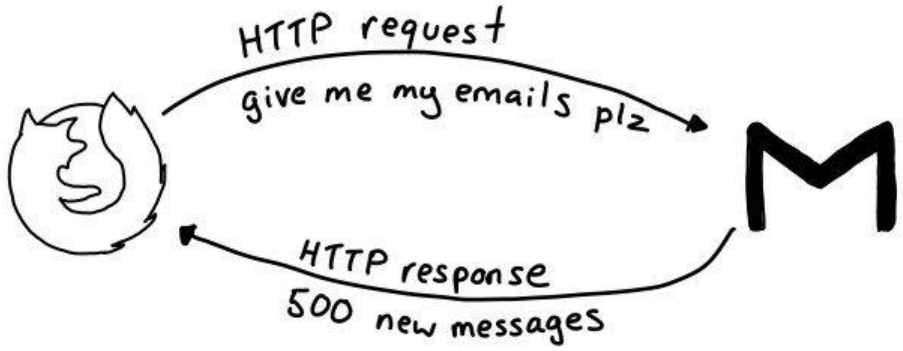
Nota : Deux flux devront être gérés – celui en sortie pour renvoyer le fichier demandé par le client et un en entrée permettant de trouver et de lire le fichier situé sur le disque dur local.

Pour le protocole *http*, vous pouvez consulter la RFC 2616 :
http://www.w3.org/Protocols/rfc2616/rfc2616.html

Julia Evans
@b0rk

What's HTTP?

HTTP is the protocol (HyperText Transfer Protocol) that's used when you visit literally any website in your browser.



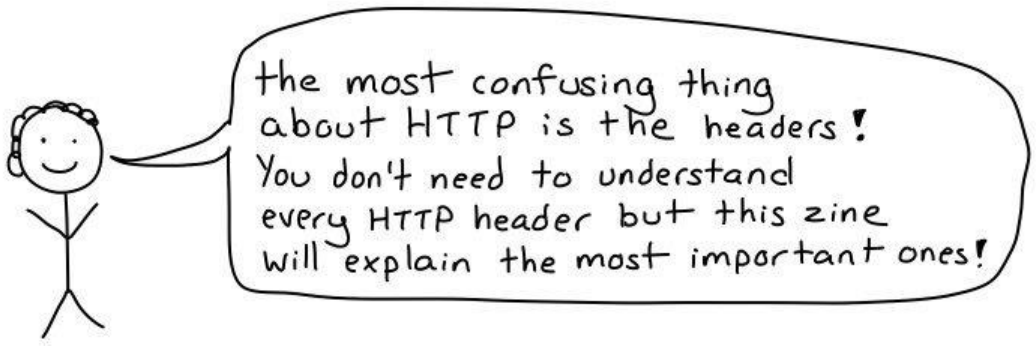
The exciting thing about HTTP is that even though it literally powers the whole internet, HTTP messages are just plain text that you can read!

HTTP requests and responses have 3 parts: the first line, the headers, and the body.

Here's an example of a valid HTTP response:

```
HTTP/1.1 200 OK                                  } first line
Content-Type: text/plain; charset=UTF-8          }
Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT     } headers
Content-Length: 24                               } (Key: value pairs)

hello I'm a HTTP response                        } body
```



4.4 un client pour lire des images

Ecrire un programme java qui récupère et affiche une image récupérée depuis une adresse IP donnée en paramètre dans une **JFrame** (ceci inclut le décodage et l'affichage de l'image JPEG).

4.6 « un cran au dessus »

Réécrivez votre client d'images en utilisant la classe **`HttpURLConnection`** (`https://docs.oracle.com/javase/8/docs/api/java/net/HttpURLConnection.html`)

4.5 un client d'un serveur d'images

Installer un logiciel de streaming de caméra comme **YawCam** (`http://www.yawcam.com`) permettant d'exposer un ensemble d'images JPG via le protocole http.

Ecrire un programme java qui récupère et affiche une image récupérée depuis le serveur web dans une **JFrame** (ceci inclut le décodage et l'affichage de l'image JPEG). Vous pouvez commencer par comprendre ce qui est lu sur la socket en utilisant l'outil **cURL** (attention aux entêtes !)

La partie représentant l'image dans le flux http est précédée d'entêtes … pensez à les supprimer ☺. En java, la classe `javax.imageio.ImageIO` permet par exemple de décoder des flux d'octets.

Etendre votre code pour afficher un flux vidéo constitué de suites d'images JPG.

4.7 s'amuser avec UDP

Récupérer l'exemple de **client et serveur** UDP à l'adresse :
`https://github.com/truillet/upssitech/blob/master/SRI/3A/ID/TP/Code/UDP.zip`

Le compiler et le tester.
Modifier le code du serveur et du client de l'exemple de manière à ce que le serveur puisse exécuter localement les commandes système **`ls et cd`** envoyées depuis n'importe quel client (bien évidemment, ceci n'est pas (vraiment pas) très sécurisé 😊).