TP Développement Réseau n°1 : Socket TCP

© 2012-2016 tv <tvaira@free.fr> v.1.0

Sommaire

Travail demandé **2**

Séquence 0 : client/serveur TCP 2

Séquence 1 : client TCP amélioré 4

Séquence 2 : serveur TCP amélioré 5

Séquence 3 : client HTTP 6

Séquence 4 : serveur HTTP 9

Annexe 1 : le résolveur nom → adresse **11**

Les objectifs de ce tp sont de mettre en oeuvre sous Linux la programmation réseau en utilisant l'interface socket pour le mode connecté (TCP).

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.*

Travail demandé

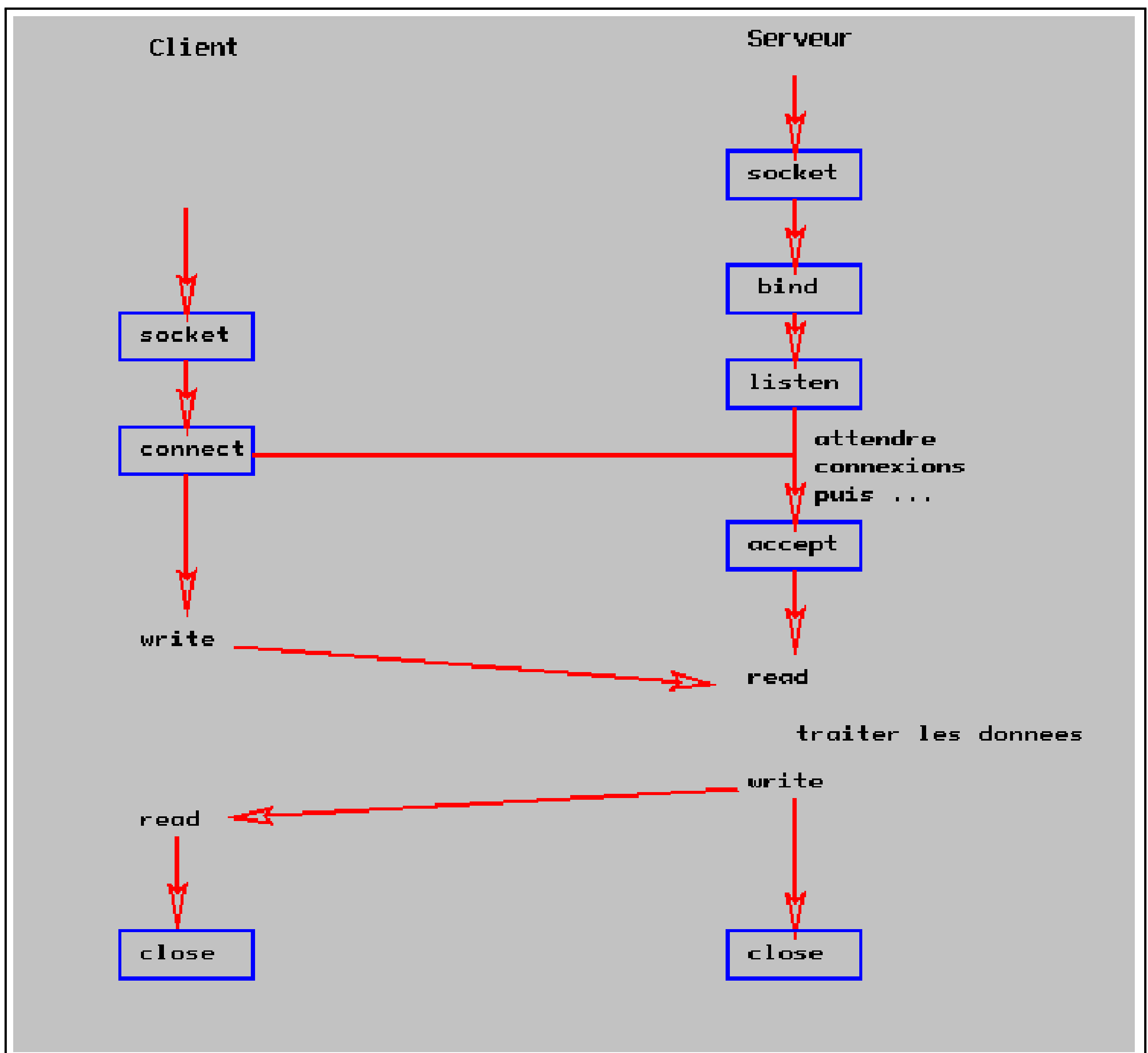
La programmation réseau nécessite l'utilisation de l'interface **socket** pour faire communiquer des processus. Une socket est un point de communication par lequel un processus peut donc émettre et recevoir des données.

Ce point de communication devra être relié à une adresse IP et un numéro de port dans le cas des protocoles Internet.

Une socket est communément considérée comme un point d'entrée initial au niveau TRANSPORT du modèle à couches DoD dans la pile de protocole.

Séquence 0 : client/serveur TCP

L'échange entre un processus client et un processus serveur en TCP peut être schématisé de la manière suivante :



Les appels systèmes utilisés dans un échange TCP

 © 2012-2016 tv <tvaira@free.fr>

On rappelle :
– le processus client a l'initiative de l'échange en faisant une demande de connexion vers l'adresse IPv4 et un numéro de port d'un serveur
– le processus serveur attend des demandes de connexion en provenance de processus client

Le processus client doit connaître au moment de la connexion le numéro de port d'écoute (et l'adresse IPv4) du serveur. Le serveur connaîtra le numéro de port et l'adresse IPv4 du client en reception d'une demande de connexion qu'il aura acceptée.

En résumé, il faudra côté **client** :
– créer une socket en mode connecté (0=TCP) dans le domaine `PF_INET` pour les protocoles Internet IPv4 : `socket(PF_INET, SOCK_STREAM, 0);`
– initialiser le point de communication distant à partir d'une structure `sockaddr_in` : le champ `sin_port` à renseigner avec le numéro de port du serveur et le champ `sin_addr` avec l'adresse IPv4 du serveur
– connecter la socket créée vers le serveur : `connect(...);`
– échanger des données avec `read()` et `write()` ou `recv()` et `send()` sur la socket connectée au serveur

Parallèlement, il faudra côté **serveur** :
– créer une socket d'écoute en mode connecté (0=TCP) dans le domaine `PF_INET` pour les protocoles Internet IPv4 : `socket(PF_INET, SOCK_STREAM, 0);`
– initialiser le point de communication local à partir d'une structure `sockaddr_in` : le champ `sin_port` à renseigner avec le numéro de port d'écoute du serveur et le champ `sin_addr` avec l'adresse IPv4 du serveur
– attacher la socket créée au point de communication local : `bind(...);`
– initialiser la socket d'écoute en attente passive : `listen(...);`
– attendre la demande de connexion en provenance d'un client : `accept(...);`
– échanger des données avec `read()` et `write()` ou `recv()` et `send()` sur la socket connectée au serveur
– éventuellement attendre de nouvelle demande de connexion en provenance d'un client : `accept(...);`

On vous fournit le code source de base d'un client et d'un serveur TCP.

Les explications détaillées sur le code source de base du client et du serveur TCP sont fournies dans le support de cours `cours-programmation-sockets.pdf`.

Question 1. Tester le client `clientTCP.c` fourni en utilisant `netcat` en serveur. Donner la commande `netcat` exécutée.

Question 2. Tester ensuite le serveur `serveurTCP.c` fourni en utilisant `netcat` en client. Donner la commande `netcat` exécutée.

Question 3. Tester maintenant le client `clientTCP.c` fourni avec le serveur `serveurTCP.c` fourni. Donner l'affichage obtenu.

Question 4. Qu'est-ce qu'une *socket* ?

Question 5. Quelles sont les trois caractéristiques d'une *socket* ?

Question 6. Quelles sont les deux informations qui définissent un point de communication en IPv4 ?

Question 7. Comment le serveur connaît-il le port utilisé par le client ?

Question 8. Comment le client connaît-il le port utilisé par le serveur ?

Question 9. À quelle couche du modèle DoD est reliée l'interface de programmation *socket* ?

 © 2012-2016 tv <tvaira@free.fr>

Question 10. Quel protocole de niveau Transport permet d'établir une communication en mode connecté ?

Question 11. Quel est l'ordre des octets en réseau ?

Question 12. À quels protocoles correspond le domaine `PF_INET` ? Est-ce le seul utilisable avec l'interface socket ? En citer au moins un autre.

Séquence 1 : client TCP amélioré

L'objectif de cette séquence est de modifier le client TCP afin qu'il puisse se connecter vers n'importe quel serveur TCP.

Étant donné qu'un serveur est identifié par une adresse IPv4 et un numéro de port, il suffit que le client TCP recoive ces deux informations en arguments comme ceci :

```
$ ./clientTCP1 adresse_ip_serveur numero_port_serveur
```

Si le client ne reçoit pas ces deux arguments, il affichera son "usage" :

```
$ ./clientTCP1
Erreur : argument manquant !
Usage : ./clientTCP1 adresse_ip_serveur numero_port_serveur
```

Pour vous aider : lorsqu'on saisit des arguments après le nom d'un programme, ceux-ci sont passés en paramètres de la fonction principal `main()`. Le prototype classique de la fonction `main()` d'un programme C est le suivant :

```
int main(int argc, char *argv[]); // ou char **argv

Le paramètre argc contient le nombre d'arguments (%d pour l'afficher).

Le paramètre argv contient l'ensemble des arguments sous la forme de chaînes de caractère :
    argv[0] contient le nom du programme, argv[1] contient le premier argument, etc ... (%s
    pour afficher un argv[x]).
```

Vous pouvez tester le principe des arguments en ligne de commande avec ce programme :

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int i;

   printf("nb d'arguments = %d\n", argc);

   // les arguments du programme :
   for(i=0;i<argc;i++)
      printf("argv[%d] = %s\n", i, argv[i]); // les arguments reçus sont toujours sous la
          forme de chaînes de caractères

   // Exemple : 2 arguments ?
   if (argc != 3)
      return 1; // indique une erreur (echo $? dans le shell)
   return 0; // ok
}
```

testArg.c

 © 2012-2016 tv <tvaira@free.fr>

Récupérer l'adresse IPv4 en notation décimale pointée ne pose pas de problème. Par contre, il vous faudra convertir (avec `atoi` ou `strtol`) le numéro de port reçu sous la forme d'une chaîne de caractères en un entier court sur 16 bits avant de l'utiliser.

Question 13. Réaliser le client `clientTCP1.c` demandé et tester le.

Aller plus loin : Parfois, on ne connaît pas l'adresse IPv4 d'un serveur mais seulement son nom (par exemple `localhost` ou `www.google.fr`). Il faut alors un **résolveur** pour obtenir son adresse IPv4.

Avant on utilisait la fonction `gethostbyname()` mais, celle-ci est maintenant obsolète et, les programmes doivent utiliser `getaddrinfo()` à la place. Un exemple d'utilisation de cette fonction est donné en Annexe.

Séquence 2 : serveur TCP amélioré

L'objectif de cette séquence est de modifier le serveur TCP afin qu'il affiche les informations (adresse IPv4 et numéro de port) identifiant le client qui vient de se connecter.

Il vous faut utiliser la fonction `getnameinfo()` pour obtenir ces informations.

Avant on utilisait la fonction `gethostbyaddr()` mais, celle-ci est maintenant obsolète.

Question 14. Réaliser le serveur `serveurTCP1.c` demandé et tester le.

Vous devriez obtenir ceci (ici ma machine se nomme "alias") :

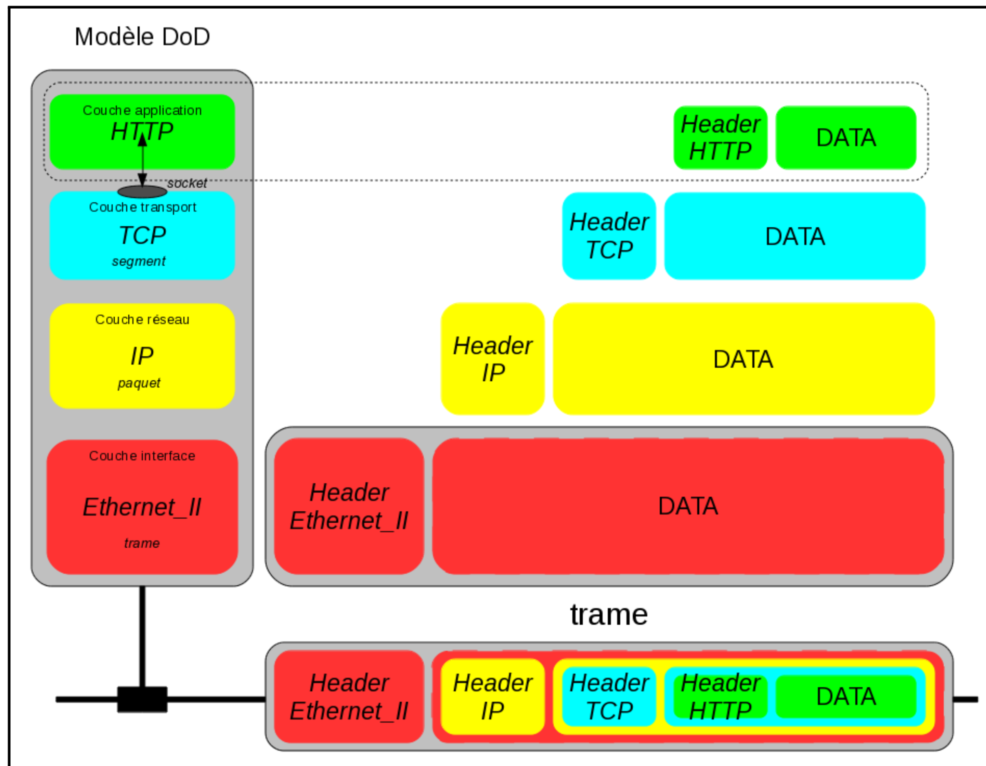
```
$ ./serveurTCP1
Socket créée avec succès ! (3)
Socket attachée avec succès !
Socket placée en écoute passive ...
Attente d'une demande de connexion (quitter avec Ctrl-C)

Connexion client alias:56748
Attente d'une demande de connexion (quitter avec Ctrl-C)
...
$
```

Rappel : le numéro de port du client a été choisi librement (parmi les numéros disponibles) par le système d'exploitation.

 © 2012-2016 tv <tvaira@free.fr>

Séquence 3 : client HTTP

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



Exemple d'encapsulation du protocole HTTP

Il est important que la gestion de la couche APPLICATION réalisée par le processus (client ou serveur) soit basée sur des fonctions d'émission et réception optimales.

Les protocoles de la couche APPLICATION privilégie l'échange de données sous forme ASCII (caractère sur 8 bits soit 1 octet) car 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. Cela signifie que ces protocoles orientés caractères utilisent la notion de "ligne".

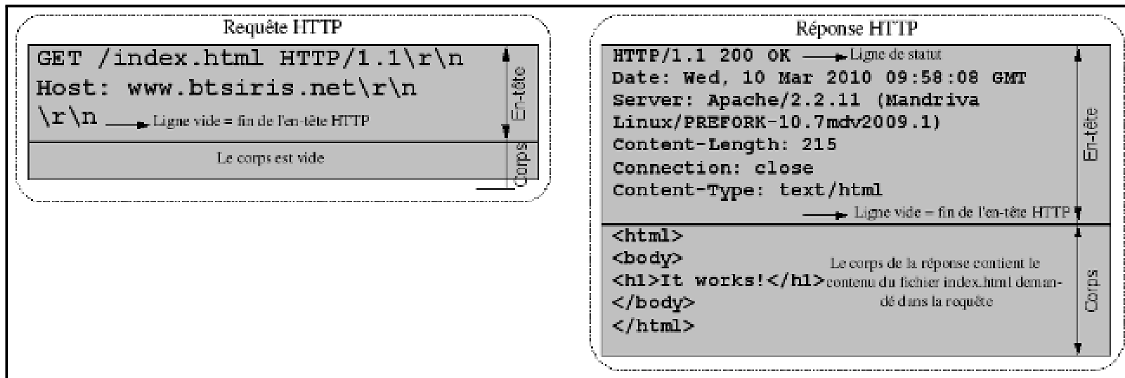
D'autre part, il est très important que nos fonctions d'émission et réception d'octets dans la *socket* tiennent compte du nombre d'octets réellement envoyés et reçus dans la *socket*.

Pour faciliter la programmation de client/serveur en mode connecté, on vous fournit les fonctions suivantes (cf. `echange-tcp.h`) :
- `ecrireLigne()` et `lireLigne()` pour l'émission et la réception des en-têtes (*header*)
- `ecrireDonnees()` et `lireDonnees()` pour l'émission et la réception des données (*data*)

Les délimiteurs "$\backslash r \backslash n$" seront retirés des données reçues.

 © 2012-2016 tv <tvaira@free.fr>

On va réaliser un **client HTTP basique**. Voici un exemple de requête (et de réponse) HTTP :



Exemple de requête et réponse HTTP version 1.1

En reprenant votre client TCP de la séquence 1, ajouter la partie de code propre à la gestion du protocole HTTP de la couche APPLICATION :

```
// Envoie une requête HTTP au serveur
sprintf(messageEnvoi, "GET / HTTP/1.0");
ecrit = ecrireLigne(descripteurSocket, messageEnvoi, strlen(messageEnvoi));
printf("Message %s envoyé (%d)\n\n", messageEnvoi, ecrit);
sprintf(messageEnvoi, "%s", ""); //une ligne vide indique la fin de l'en-tête HTTP
ecrit = ecrireLigne(descripteurSocket, messageEnvoi, strlen(messageEnvoi));

// On réceptionne la réponse HTTP du serveur
// Affichage des lignes d'en-têtes HTTP
/* TODO */
// Affichage du corps de la réponse HTTP
/* TODO */
```

Envoi d'une requête HTTP 1.0 avec un client TCP

Pour simplifier la requête émise, on utilisera côté client la version 1.0 du protocole HTTP.

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 devez obtenir :

```
$ ./clientHTTP 127.0.0.1 80
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !

Message GET / HTTP/1.0 envoyé avec succès !

Message reçu du serveur (en-tête) : HTTP/1.1 200 OK
Message reçu du serveur (en-tête) : Date: Sat, 03 Nov 2012 14:09:56 GMT
Message reçu du serveur (en-tête) : Server: Apache/2.2.22 (Mandriva Linux/PREFORK-0.1mdv2010
    .2)
Message reçu du serveur (en-tête) : Last-Modified: Wed, 01 Feb 2012 12:23:51 GMT
Message reçu du serveur (en-tête) : ETag: "1cad-d4-4b7e626f823c0"
```

 © 2012-2016 tv <tvaira@free.fr>

```
Message reçu du serveur (en-tête) : Accept-Ranges: bytes
Message reçu du serveur (en-tête) : Content-Length: 212
Message reçu du serveur (en-tête) : Connection: close
Message reçu du serveur (en-tête) : Content-Type: text/html
Message reçu du serveur (en-tête) -> fin

Message reçu du serveur (données) :
<html>
<body>
<!-- $Id: index.html 92365 2007-09-23 14:04:27Z oden $ -->
<!-- $HeadURL: http://svn.mandriva.com/svn/packages/cooker/apache-conf/current/SOURCES/index
    .html $ -->
<h1>It works!</h1>
</body>
</html>
$
```

On peut tester avec un serveur web Internet :

```
$ ./clientHTTP www.google.fr 80
Socket créée avec succès ! (3)
Connexion au serveur réussie avec succès !

Message GET / HTTP/1.0 envoyé avec succès !

Message reçu du serveur (en-tête) : HTTP/1.0 302 Found
Message reçu du serveur (en-tête) : Cache-Control: private
Message reçu du serveur (en-tête) : Content-Type: text/html; charset=UTF-8
Message reçu du serveur (en-tête) : Location: http://www.google.fr/?gfe_rd=cr&ei=
    hUSrVvi5BoaLoQewsIyAAg
Message reçu du serveur (en-tête) : Content-Length: 258
Message reçu du serveur (en-tête) : Date: Fri, 29 Jan 2016 10:52:53 GMT
Message reçu du serveur (en-tête) : Server: GFE/2.0
Message reçu du serveur (en-tête) -> fin

Message reçu du serveur (données) : <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;ei=hUSrVvi5BoaLoQewsIyAAg">here</A>.
</BODY></HTML>
```

Question 15. Écrire le client `clientHTTP.c` et tester le.

Question 16. Quel est le format des données renvoyées par le serveur ?

 © 2012-2016 tv <tvaira@free.fr>

Séquence 4 : serveur HTTP

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

L'écriture d'un serveur est souvent plus complexe qu'un simple client car il se doit de rendre du mieux possible le service qu'il offre.

Ici, la principale difficulté est de répondre à un ensemble de clients demandant le service. Pour cela, on distingue les deux particularités suivantes :

- l'acceptation (écoute) des demandes
- le traitement (dialogue) des demandes

Pour accepter les demandes, on peut procéder de différentes manières :

- **une seule socket d'écoute en mode bloquant (cas classique)**.
- plusieurs sockets d'écoute : il est important de ne pas rester bloqué en attente avec l'appel `accept()` sur une seule d'entre elles. Il est alors possible d'utiliser l'appel système `select()` ou `poll()` sur l'ensemble des sockets pour déterminer la disponibilité des données à lire, et d'effectuer ensuite l'appel `accept()` sur celles qui ont effectivement reçu des demandes de connexion.
- On peut déléguer l'attente de connexion à un service standard (par exemple `inetd`) qui assure l'écoute pour tous les serveurs qui lui sont déclarés.

Plusieurs situations de traitement des demandes sont possibles :

- le traitement séquentiel : ceci a un sens si le traitement est rapide, car pendant ce temps la file d'attente des demandes de connexions peut s'allonger ... et saturer pour obtenir un DoS (*Deny of Service*)
- **le traitement parallèle (ou multi-tâche)** : c'est la méthode classique, on se dépêche de créer un processus fils avec l'appel `fork()` qui traitera lui-même l'échange avec le client. Le processus père lui continue l'attente des demandes de connexion.

Le temps de création des processus fils peut devenir une charge importante pour le système hébergeant le serveur. Dans ce cas, on peut envisager les solutions suivantes : une création anticipée des processus ou l'utilisation des *threads*.

On va implémenter la solution classique pour notre serveur : une seule socket d'écoute et création de processus fils avec `fork()` pour le dialogue avec les clients.

```
...
// boucle d'attente de connexion : en théorie, un serveur attend indéfiniment :( !
while(1)
{
   // c'est un appel bloquant
   socketDialogue = accept(socketEcoute, (struct sockaddr *)&pointDeRencontreDistant, &
       longueurAdresse);
   ...
   /* Création du processus de dialogue */
   switch(fork())
   {
   case -1 :  perror("fork:"); exit(-7);
   case 0 :   /* processus fils */
              /* le fils n'utilise pas la socket d'écoute */
              close(socketEcoute);

              /* ici on dialogue avec le client sur la socket de dialogue */
```

 © 2012-2016 tv <tvaira@free.fr>

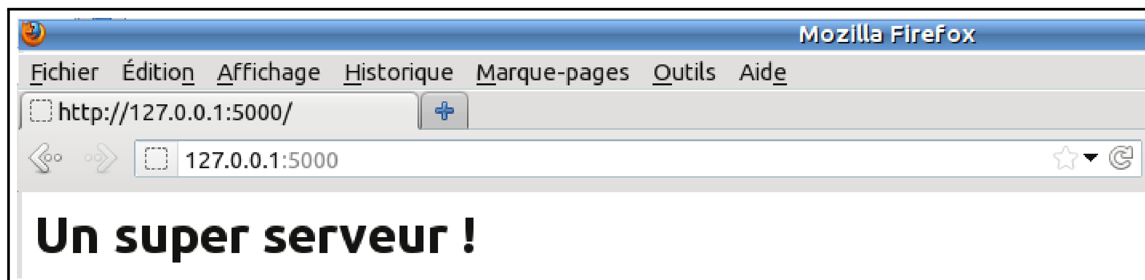
```
                /* TODO */

                /* à la fin du dialogue, on ferme la socket de dialogue */
                close(socketDialogue);
                exit(0); /* fin du processus fils */
    default :   /* processus pere */
                /* le pere n'utilise pas la socket de dialogue */
                close(socketDialogue);
                /* le pere se replace en attente des demandes de connexion */
    }
  }
```

Solution classique pour un serveur TCP multi-clients

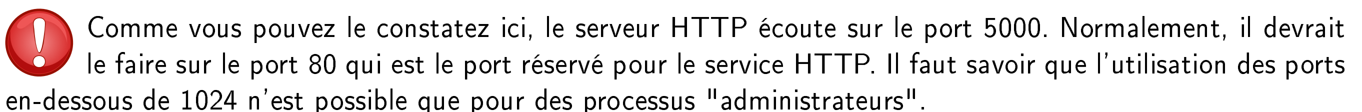
Concernant la gestion du protocole HTTP par notre serveur, on va se contenter d'envoyer systèmatiquement une réponse basique en HTTP :

```
HTTP/1.1 200 OK
Server: (le nom du serveur !)
Connection: close
Content-Type: text/html

<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 `serveurHTTP.c` et tester le sur le port 5000.

Question 18. Que faut-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 ?

 © 2012-2016 tv <tvaira@free.fr>

Annexe 1 : le résolveur nom → adresse

Parfois, on ne connaît pas l'adresse IPv4 d'un serveur mais seulement son nom (par exemple `localhost` ou `www.google.fr`). Il faut alors un **résolveur** pour obtenir son adresse IPv4.

Avant on utilisait la fonction `gethostbyname()` mais, celle-ci est maintenant obsolète et, les programmes doivent utiliser `getaddrinfo()` à la place.

```
#include <stdio.h>
#include <stdlib.h> /* pour exit et atoi */
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h> /* pour memset */
#include <netinet/in.h> /* pour struct sockaddr_in */
#include <arpa/inet.h> /* pour htons et inet_aton */
#include <netdb.h> /* pour getaddrinfo */

int main(int argc, char *argv[])
{
   int descripteurSocket;
   int retour;
   struct addrinfo hints;
   struct addrinfo *result, *rp;

   // Il faut 2 arguments à ce programme
   if (argc != 3)
   {
      fprintf(stderr, "Erreur : argument manquant !\n");
      fprintf(stderr, "Usage : %s adresse_ip numero_port\n", argv[0]);
      exit(-1); // On sort en indiquant un code erreur
   }

   // Obtenir un point de communication distant correspondant au couple nom/port fourni en
       argument
   memset(&hints, 0, sizeof(struct addrinfo));
   hints.ai_family = PF_INET; // sinon AF_UNSPEC pour IPv4 ou IPv6
   hints.ai_socktype = SOCK_STREAM; // mode connecté
   hints.ai_flags = 0; // options supplémentaires, ici aucune
   hints.ai_protocol = 0; // n'importe quel type

   retour = getaddrinfo(argv[1], argv[2], &hints, &result);
   if (retour != 0)
   {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(retour)); // Affiche le message d'
          erreur
      exit(-2); // On sort en indiquant un code erreur
   }

   // getaddrinfo() retourne une liste de structures d'adresses
   // on cherche la première qui permettra un connect()
   for (rp = result; rp != NULL; rp = rp->ai_next)
   {
      // Crée un socket de communication
      descripteurSocket = socket(rp->ai_family, rp->ai_socktype, rp->ai_protocol);
```

 © 2012-2016 tv <tvaira@free.fr>

```
      // Échec ?
      if (descripteurSocket == -1)
         continue; // alors on essaye la suivante

      // Débute la connexion vers le processus serveur distant
      // Ok ?
      if((connect(descripteurSocket, rp->ai_addr, rp->ai_addrlen)) != -1)
         break; // on a trouvé une adresse valide !

      // On ferme cette ressource avant de passer à la suivante
      close(descripteurSocket);
   }

   if (rp == NULL)
   {
      // on a donc trouvé aucune adresse valide !
      fprintf(stderr, "Impossible de se connecter à cette adresse !\n");
      exit(-3); // On sort en indiquant un code erreur
   }

   freeaddrinfo(result); // on libère les résultats obtenus et on continue ...

   printf("Socket créée avec succès ! (%d)\n", descripteurSocket);
   printf("Connexion au serveur réussie avec succès !\n");

   // ...

   // On ferme la ressource avant de quitter
   close(descripteurSocket);

   return 0;
}
```

 © 2012-2016 tv <tvaira@free.fr>