Prénom, Nom :	*Groupe :*

Test de Réseau
le 19 janvier 2007

Conseils et demandes

N'utilisez pas de rouge. Écrivez lisiblement et assez grand. Prenez votre temps et regardez tout le document. N'oubliez-pas que même en `C++` les `int` et les `char *` ne sont pas de « classes », n'ayant pas de méthode du style `.size()`. Utilisez des wrappers PARTOUT. La syntaxe d'un ***appel correct*** de `Write()` est

```
Write(descripteur, tampon, nombreDOctetsATransfererDepuisleTampon);
```

où `descripteur` est un **entier** et tampon est un ***pointeur***. Pour utiliser un **`string maChaine`** il faut faire

```
Write(descripteur, maChaine . c_str(), maChaine . size());
```

Pour lire, vous allez utiliser une fonction spéciale `LireN()`, avec le même type de paramètres, mais là, pas de `string`s du tout. Ces fonctions seront suffisantes pour rédiger tout ce qui est demandé concernant le transfert de données.

Commencez par les questions qui suivent, et passez juste cinq ou dix minutes dessus.

1 Questions – une seule réponse correcte

1. Une *socket* est
   (a) un processus spécial, nommé aussi *serveur*
   (b) un type particulier de mutex
   (c) un point de communication entre processus
2. Un *protocole de communication* est
   (a) un programme écrit en C++ utilisant des sockets
   (b) un ensemble de règles gérant tous les cas possibles pour le dialogue réseau
   (c) un processus qui écoute pour les connexions entrantes
3. Le *protocole IP* est un protocole
   (a) couche liaison de données
   (b) couche transport
   (c) couche physique
   (d) couche réseau
4. Une *adresse IP* contient
   (a) trois octets
   (b) quatre octets
   (c) cinq octets
5. Un *serveur DNS* sert
   (a) à faire arriver les packets IP à destination – Direct Node Selection
   (b) à distribuer les news – Distributed News Service
   (c) à obtenir la correspondance entre noms de sites et leurs adressses IP – Domain Name System
6. La fonction `htons()`
   (a) convertit un nombre entier long du format hôte vers le format réseau
   (b) convertit un nombre entier court du format réseau vers le format hôte
   (c) convertit un nombre entier court du format hôte vers le format réseau
7. On a besoin des fonctions `htons()`, `htonl()`, `ntohs()` et `ntohl()`
   (a) pour passer d'ASCII à UNICODE
   (b) pour pouvoir transférer les caractères accentués par ftp
   (c) parce que ce qui transite sur le réseau est en big endian, alors que différentes architectures ont leurs ordres particuliers (little endian, etc,), et on dispose ainsi d'une convention uniforme pour transmettre des valeurs numériques

2 À votre service

Complétez le code de la fonction suivante, qui met en place une socket côté serveur et l'enregistre pour l'écoute

```
int ServerInit(::uint16_t & NumPort, int fileAttente = 5) {
    const int sd = //_// CREATION socket INET STREAM avec socket()
    if(-1 == sd) throw CExcFctSyst ("socket()");
    ::sockaddr_in Addr;
    memset (& Addr, 0, sizeof (Addr));
    Addr.sin_family      = //_// famille Internet
    Addr.sin_port        = NumPort;
    Addr.sin_addr.s_addr = INADDR_ANY;
    //_// APPEL de bind() (attention au cast vers sockaddr *)
    //_// SI ECHEC
        Close (sd);
        throw CExcFctSystFile ("bind()", sd);
    } // finsi
    ::socklen_t Lg = sizeof (Addr);
    //_// APPEL  getsockname()
    //_// SI ECHEC
        Close (sd);
        throw nsSysteme::CExcFctSystFile ("getsockname()", sd);
    } // finsi
    NumPort = //_// RECUPERATION DEPUIS Addr, maintenant rempli par getsockname()
    //_// APPEL DE listen()
    return(sd);
}
```

3 Exercice – client et serveur FTP simpliste

Explications

Le but de cet exercice est de vous amener à rédiger les parties essentielles d'un client et d'un serveur FTP – file transfer protocol – d'une manière (grandement) simplifiée. Le protocole FTP est principialement simple, mais néanmoins assez complexe à mettre en œuvre en prenant soin de tous les détails, tous les cas de figure, etc. Ici nous allons en ébaucher le squelette, aboutissant à quelque chose qui fonctionne réellement, mais avec peu de commandes, et sans nécessairement garantir le traitement de toutes les erreurs ou cas extrèmes.

Comment ça marche, réellement, le FTP ?

Présentation FTP

Le client est (sous Unix/Linux, par exemple) invoqué sur la ligne de commande avec `ftp <serveur>`, et s'identifie comme lors d'une connexion (username, et puis mot de passe). Ensuite, on peut manipuler les fichiers et répertoires de `<serveur>` (comme dans un shell normal), et on peut récupérer en local des fichiers depuis le serveur (avec `get`), en télécharger depuis le local sur le serveur (avec `put`), et même changer de répertoire courant local (avec `lcd`).

Détails de structure et fonctionnement

Concrètement, le serveur et le client communiquent par une paire de sockets dite « de commande » et par une autre paire de sockets, dite « de données ». Par exemple, pour un `get`, le client envoie la commande et le nom du fichier à obtenir par sa socket de commande, et reçoit le contenu du fichier sur sa socket de données. Ici les deux connexions seront établies avec les wrappers vus en TD/TP, le serveur se mettant à l'écoute et le client venant se connecter.

Protocole

- le serveur se met à l'écoute sur un port « bien connu » (le vrai FTP le fait sur le port 21).
- le client demande sur ce port une connexion – ce sera celle de commande
- le serveur accepte cette demande, et se duplique, pour que le fils la traite (le père revenant à l'écoute)
- le client et le fils du serveur (qu'on appellera serveur quand même) ont un bref échange initial (nom d'utilisateur, mot de passe, etc. finissant par le port de l'écoute pour la connexion de données trasmis par le serveur au client)
- le client se met en boucle pour
  - lire une commande de l'utilisateur au clavier et l'envoyer au serveur par la socket de commande
  - recevoir une réponse initiale depuis la socket de commande (par exemple « commande non-comprise », etc.)
  - le cas échéant (si pas d'erreur, etc.), ouvrir une autre socket, celle de données (sur le port obtenu à la fin de l'échange initial)
  - récupérer tout ce qui y arrive (et le déposer dans un fichier, ou bien l'afficher sur la sortie standard, etc.)
- le serveur (i.e. le fils) se met également en boucle, car il est construit assez symétriquement par rapport au client.

Cadre général de programmation – code fourni et à respecter

```
namespace {
    enum Command { CMD_CD, CMD_LCD, CMD_LS, CMD_GET, CMD_PUT, CMD_BYE, CMD_UNKNOWN };
    const unsigned int PATIENCE(10), CMDCOUNT(6);
    const char *cmdStr[CMDCOUNT] = {"cd2", "lcd", "lsl", "get", "put", "bye"};
    const unsigned int CMDLENGTH(3), CMDARGLENGTH(256);
    const unsigned int MAXMSGLENGTH(1024), ERRORTXTLENGTH(1024);
    const unsigned int USERNAMELENGTH(8), PASSWDLENGTH(20);
    int socketEcoute;
    const string invldUsrTxt  ("ERROR Invalid username/password.\n");
    const string greetingTxt  ("Ok, welcome.");
    const string dataPortTxt  ("DataPort is ");
    const string cmdUnknTxt   ("ERROR Unknown command ");
    const string processingTxt("Processing...");
    const string byebyeTxt    ("Good bye.");
}
// ppal(), traitants et autres fonctions auxiliaires deja definis
// ClntFTP() est donne en entier pour reference, apres ``Travail a faire''
// le char * cmdStr est donne juste pour ``info'' -- il n'est pas a utiliser par vous
```

3.1 Travail à faire

3.1.1 Écrivez la fonction auxiliaire `LireN()`

```
int LireN(int socket, char *buff, int count = 0,
          bool qSetNull = true);
```

qui lit au plus `count` *octets, les déposant dans* `buff`*, s'arrêtant au premier "\n" et rendant le nombre d'octets lus (**attention**, le "\n" **n'est pas** à mettre dans* `buff`*) ; si* `qSetNull` *est vrai, alors avant de finir, cette fonction rajoute un "\0" dans* `buff`*, à la fin de ce qu'elle vient d'y déposer.*

selon les pas suivants :

- si `count` est strictement positif
  - définir deux entiers, `nbTotLus`, initialisé à zéro, et `patience`, initialisé à `PATIENCE`
  - dans une boucle tant qu'on a de la patience, qu'on n'a pas encore lu le nombre `count` d'octets, et qu'on n'est pas arrivé au "\n" (codifié par le booléen `qNotEOL` vrai)
    - tenter de lire avec `Read()` un seul octet depuis `socket` en le deposant au bon endroit dans `buff`
    - si jamais la tentative de lecture revient avec zéro, décrémentez seulement `patience` et dormez une seconde.
    - autrement, tout mettre à jour (`qNotEOL`, `nbTotLus`, etc.)
  - si le paramètre `qSetNull` est vrai, mettre un '\0'à la fin
  - rendre `nbTotLus` (qui n'est pas incrémenté encore une fois même si `qSetNull` est vrai)

3.1.2 Complétez l'authentification serveur selon les indications mises en commentaire avec //_//

```
void ServFTP(::uint16_t NumPort) {
    const int socketEcoute (ServerInit (NumPort));
    cout << "Le numero de port principal d'ecoute est " << ::ntohs (NumPort) << endl;
    Signal(SIGCHLD,TraiterZombies);Signal(SIGTERM,Quitter);
    while(1) {
        int socketCommande;
        //_// ECOUTE avec Accept() pour obtenir socketCommande
        if(Fork()) {
           //_// FERMETURE socket de commandes
           continue;
        }
        //_// FERMETURE socket Ecoute (donc on est dans le fils)
        for(bool qValidUser = false; !qValidUser;) {
            const string loginTxt("login: \n");
            const string passwdTxt("Password: \n");
            char userName[USERNAMELENGTH+1], passwd[PASSWDLENGTH+1];
            //_// DIALOGUE D'AUTHENTIFICATION SUR socketCommande (PAS DE stdin/stdout!!!)
            //_// POUR RECUPERER userName ET passwd
            qValidUser = validerUser(userName,passwd);
            if(!qValidUser)  //_// ECRITURE du message de invldUsrTxt sur socketCommand
        }
        //_// ECRITURE du greetingTxt sur socketCommande
        // (car le client vient d'etre reconnu et donc accepte)
        // (et maintenant la derniere partie -- celle du port de donnees :)
        uint16_t dataPort(0);
        const int socketDataEcoute(//_// A OBTENIR AVEC ServerInit()
                                   //_// TOUT EN REMPLISSANT dataPort AUSSI
        ostringstream buff;
        buff << //_// LA BONNE VALEUR DE dataPort
        const string socketDataEcouteMsgTxt(dataPortTxt + buff . str() + "\n");
        //_// ECRITURE DE socketDataEcouteMsgTxt SUR socketCommande
        // (et nous sommes maintenant a la fin de l'initialisation de la connexion)
```

3.1.3 Complétez la partie traitement commande du serveur selon ce schéma (donc suite au 3.1.2)

Dans une boucle infinie (considérant disposer de la fonction `analyserCmd()`)

- déclarez `char receivedCmdArg[MAXMSGLENGTH+1];` et lisez dedans la commande à traiter depuis `socketCommande`
- définissez `Command commande(analyserCmd(receivedCmdArg,&receivedArgTxt))` et traitez les cas :
  - commande inconnue `CMD_UNKNOWN` – écrivez `cmdUnknTxt` sur `socketCommande` et faites `continue`
  - commande de fin (`CMD_BYE` – écrivez `byebyeTxt` sur `socketCommande` et faites `return`
- écrivez `processingTxt` sur `socketCommande`
- si la commande n'est pas `CMD_CD`
  - dupliquez le processus
  - dans le père faites `Wait()` et puis `continue`
- acceptez sur `socketDataEcoute` une connexion avec `Accept()`, pour obtenir `socketData` (qui est la deuxième liaison avec le client) – revoyez 3.1.2 pour comprendre.
- faites un `switch(commande)` et dedans
  - mettez le traitement de `CMD_GET` suivant :

```
case CMD_GET:  Dup2(socketData, STDOUT_FILENO);
               execl("/bin/cat","cat",receivedArgTxt . c_str(),0);
               throw CExcFctSyst ("execl()"); break;
```

  - traitez similairement le `case CMD_LS` :
    - dupliquez `socketData` sur la sortie standard,

  - faites `execl` de `ls -l`, mais cette fois **sans `receivedArgTxt`**
  - levez une exception similairement
- traitez le `case CMD_PUT` :
  - ouvrez le fichier de nom `receivedArgTxt` en écriture (avec `Open()`) sur le descripteur `outFile` ,
  - dupliquez `socketData` sur **entrée standard**
  - dupliquez `outFile` sur la sortie standard,
  - faites `execl` du `cat`, comme pour le `CMD_GET`, mais **sans `receivedArgTxt`**.
  - levez une exception similairement
- traitez le `case CMD_CD :` – duplication similaire à celle de `CMD_LS` et appel de `ChDir()` avec `receivedArgTxt` (**sans `execl()`** !) dans `try-catch`, avec confirmation/annonce erreur sur sortie standard (qui ira chez le client, de par la duplication) et fermeture de `socketData` et de la sortie standard (si elle est différente)

Avant de reboucler, faites un `return` si la commande n'était pas `CMD_CD`.

Supplément – code fourni – côté client – pour mieux comprendre le serveur

```
int Connexion(const ::in_addr & AddrIP,::uint16_t NumPort)throw (CException) {
    const int sd = ::socket (PF_INET, SOCK_STREAM, 0);
    if(-1 == sd) throw CExcFctSyst ("socket()");
    ::sockaddr_in Addr;memset (& Addr, 0, sizeof (Addr));
    Addr.sin_family = AF_INET;Addr.sin_port = NumPort; Addr.sin_addr = AddrIP;
    if (::connect(sd, reinterpret_cast <const ::sockaddr *> (& Addr), sizeof (Addr)))
                throw CExcFctSystFile ("connect()", sd);
    return sd;
}
::in_addr AddrIP (const char * Nom); // renvoie la bonne adresse IP a partir du nom
void ClntFTP(const char *host, const ::uint16_t NumPort) {
    const int socketCommande (Connexion(AddrIP(host),NumPort));
    const string prompt (string("miniFTP (") + host + "): "); string userLine;
    char servAnswMsg[MAXMSGLENGTH+1];
    while(true) {
        LireN(socketCommande, servAnswMsg, MAXMSGLENGTH);
        cout << prompt << servAnswMsg << flush; // "login: "
        getline(cin,userLine); userLine += "\n"; // user name
        Write(socketCommande, userLine . c_str(), userLine . size());
        LireN(socketCommande, servAnswMsg, MAXMSGLENGTH);
        cout << prompt << servAnswMsg << flush; // "Password: "
        getline(cin,userLine); userLine += "\n";  // password
        Write(socketCommande, userLine . c_str(), userLine . size());
        LireN(socketCommande, servAnswMsg, MAXMSGLENGTH);
        cout << servAnswMsg << "\n"; if(string(servAnswMsg) == greetingTxt) break;
    }
    LireN(socketCommande, servAnswMsg, MAXMSGLENGTH);
    const string servAnswData(servAnswMsg); ::uint16_t NumPortData (0);
    if(servAnswData . size() >=  dataPortTxt . size() &&
       servAnswData . substr(0,  dataPortTxt . size()) == dataPortTxt) {
        istringstream buff(servAnswData . substr(dataPortTxt . size()));
        buff >> NumPortData;
        cout << "  << Server data connection request on port " << NumPortData << "\n";
    }
    for(bool qLoop (true); qLoop;) {
        cout << prompt << flush;
        getline(cin,userLine);
        string fileArgTxt;
        Command commande(analyserCmd(userLine, &fileArgTxt));
        if(commande == CMD_UNKNOWN) {
            cout << " OOPS: Commands are\n" << "    lcd <dir>\n"
                 << "    cd2 <dir>\n"    << "    lsl\n"
                 << "    get <file>\n"   << "    put <file>\n" << "    bye\n"; continue;
        }
```

```
        if(commande == CMD_LCD) {
            try { ChDir(fileArgTxt . c_str());
                  cout << "  << Locally changed to " << fileArgTxt << "\n"; }
            catch(CException &exc) {
                cout << "ERROR Could not locally cd to "
                     << fileArgTxt << " " << strerror(exc . GetCodErr()) << "\n";
            }
            catch(...) { cout << "FATAL Could not locally cd to "<<fileArgTxt
                              << " "<<strerror(errno)<< "\n"; }
            continue;
        }
        userLine += "\n";Write(socketCommande, userLine . c_str(), userLine . size());
        cout << "  << Sent to server, waiting for reply...\n";
        LireN(socketCommande, servAnswMsg, MAXMSGLENGTH);
        cout << "  << Got reply from server, analyzing...\n";
        const string servAnsw(servAnswMsg);
        if(servAnsw . size() >= cmdUnknTxt . size() &&
           servAnsw . substr(0, cmdUnknTxt . size()) == cmdUnknTxt) {
            cout << "  << OOPS: Server said " <<  servAnsw << "\n";
            continue;
        }
        if(servAnsw . size() >= byebyeTxt . size() &&
           servAnsw . substr(0, byebyeTxt . size()) == byebyeTxt) {
            cout << "  << Server is closing, and says " << byebyeTxt << "\n"; return;
        }
        cout << "  << Server said \n" << servAnsw << "\n  <<_____\n";
        const int socketData (Connexion(AddrIP(host),htons(NumPortData)));
        if(Fork()) { Close(socketData); Wait(); continue; }
        switch(commande) {
            case CMD_GET: {
                cout << "  << Getting " << fileArgTxt  << " from server...\n";
                if(Fork()) Wait();
                else {
                    const int outFile (Open(fileArgTxt . c_str(),
                                            O_RDWR | O_CREAT, 0600));
                    Dup2(outFile,    STDOUT_FILENO); Dup2(socketData, STDIN_FILENO);
                    execl("/bin/cat","cat",0);throw CExcFctSyst ("execl()");
                }
                cout << "  << End of file downloading...\n";break;
            }
            case CMD_PUT:
                cout << "  << Sending " << fileArgTxt << " to server...\n";
                if(Fork()) Wait();
                else {
                    Dup2(socketData, STDOUT_FILENO);
                    execl("/bin/cat","cat",fileArgTxt . c_str(),0);
                    throw CExcFctSyst ("execl()");
                }
                cout << "  << End of file uploading...\n";break;
            default:
                Dup2(socketData, STDIN_FILENO);
                cout << "  << Here's what the server gives us...\n";
                if(Fork()) Wait();
                else { execl("/bin/cat","cat",0);throw CExcFctSyst("execl()"); }
                cout << "  << End of what server said...\n";break;
        } // switch(commande)
        return;
    } // loop
}
```

Man

```
#include <netinet/in.h>
uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);

#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

socket cree un point de communication, et renvoie un descripteur. Le parametre domain indique le domaine de communication pour le dialogue ; ceci selectionne la famille de protocole a employer. Elles sont definies dans le fichier ¡linux/socket.h¿. Les formats actuellement proposes sont :

```
Nom     Utilisation     Page
PF_UNIX,PF_LOCAL        Communication locale    unix(7)
PF_INET         IPv4 Protocoles Internet        ip(7)
PF_INET6        IPv6 Protocoles Internet
...etc...
```

Les sockets ont le type, indique, ce qui fixe la semantique des communications. Les types definis actuellement sont :

- `SOCK_STREAM` Support de dialogue garantissant l'integrite, fournissant un flux de donnees binaires, et integrant un mecanisme pour les transmissions de donnees hors-bande.
- `SOCK_DGRAM` Transmissions sans connexion, non garantie, de datagrammes de longueur fixe, generalement courte.

Le protocole a utiliser sur la socket est indique par l'argument protocol. Normalement il n'y a qu'un seul protocole par type de socket pour une famille donnee, auquel cas l'argument protocol peut etre nul.

Une socket de type `SOCK_STREAM` est un flux d'octets full-duplex, similaire aux tubes (pipes). Elle ne preserve pas les limites d'enregistrements. Une socket `SOCK_STREAM` doit être dans un état connecté avant que des données puissent y être lues ou écrites. Une connexion sur une autre socket est etablie par l'appel systeme connect(2). Une fois connectee les donnees y sont transmises par read(2) et write(2) ou par des variantes de send(2) et recv(2). Quand une session se termine, on referme la socket avec close(2).

Les protocoles de communication qui implementent les sockets SOCK_STREAM garantissent qu'aucune donnee n'est perdue ou dupliquee. Si un bloc de donnees, pour lequel le correspondant a suffisament de place dans son buffer, n'est pas transmis correctement dans un delai raisonnable, la connexion est consideree comme inutilisable.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

`bind()` fournit a la socket sockfd, l'adresse locale `my_addr`. `my_addr` est longue de `addrlen` octets. Traditionnellement cette operation est appelee "affectation d'un nom a une socket" (Quand une socket est créée, par l'appel-système socket(2), elle existe dans l'espace des noms mais n'a pas de nom assigne).

Il est normalement necessaire d'affecter une adresse locale avec bind avant qu'une socket SOCK_STREAM puisse recevoir des connexions (voir accept(2)).

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

Pour accepter des connexions, une socket est d'abord creee avec socket(2), puis le desir d'accepter des connexions entrantes, et la limite de la file d'entree sont indiques avec listen(2), ensuite les connexions seront acceptees avec accept(2). L'appel systeme listen(2) s'applique seulement aux sockets de type `SOCK_STREAM` ou `SOCK_SEQPACKET`.

Le parametre backlog definit une longueur maximale pour la file des connexions en attente. Si une nouvelle connexion arrive alors que la file est pleine, le client recoit une erreur indiquant ECONNREFUSED, ou, si le protocole sous-jacent supporte les retransmissions, la requete peut etre ignoree afin qu'un nouvel essai reussisse.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sock, struct sockaddr *adresse, socklen_t *longueur);
```

accept() est utilise generalement avec des processus serveurs orientes-connexion. Cet appel systeme est employe avec les sockets utilisant un protocole en mode connecte (SOCK_STREAM, SOCK_SEQPACKET et SOCK_RDM) Il extrait la premiere connexion de la file des connexions en attente, cree une nouvelle socket avec essentiellement les memes proprietes que sock et alloue pour cette socket un nouveau descripteur de fichier qu'il renvoie. La nouvelle socket n'est plus en etat d'ecoute. La socket originale sock n'est pas modifiee par l'appel-systeme. Remarquez que les attributs du descripteur de fichier (tout ce qu'on peut configurer avec l'option F_SETFL de fcntl() comme l'etat non-bloquant ou asynchrone), ne sont pas herites durant un accept().

L'argument sock est une socket qui a ete creee avec la fonction socket(2). attachee a une adresse avec bind(2). et attend des connexions apres un appel listen(2).

L'argument adresse est un pointeur sur une structure sockaddr. La structure sera remplie avec l'adresse du correspondant se connectant, telle qu'elle est connue par la couche de communication. Le format exact du parametre adresse depend du domaine dans lequel la communication s'etablit. (Voir socket(2) et la page de manuel correspondant au protocole). L'argument longueur est un parametre-resultat : il doit contenir initialement la taille de la structure pointee par adresse, et est renseigne au retour par la longueur reelle (en octet) de l'adresse remplie. Quand adresse vaut NULL, rien n'est rempli.

S'il n'y a pas de connexion en attente dans la file, et si la socket n'est pas marquee comme non-bloquante, accept() se met en attente d'une connexion. Si la socket est non-bloquante, et qu'aucune connexion n'est presente dans la file, accept retourne une erreur EAGAIN.

```
#include <sys/socket.h>

int getsockname(int s, struct sockaddr *name, socklen_t *namelen)
```

Getsockname renvoie le nom name de la socket indiquee. Le parametre namelen doit etre initialise pour indiquer la taille de la zone memoire pointee par name. En retour, il contiendra la taille effective (en octets) du nom renvoye.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen);
```

Le parametre sockfd est une socket. Si la socket est du type SOCK_DGRAM, alors serv_adr est l'adresse a laquelle les datagrammes seront envoyes par defaut, et la seule adresse depuis laquelle ils seront recus. Si la socket est du type SOCK_STREAM ou SOCK_SEQPACKET, cette fonction tente de se connecter a une autre socket. L'adresse de l'autre socket est indiquee par serv_addr, qui doit etre une adresse (de longueur addrlen) dans le meme domaine que la socket.

```
#include <stdlib.h>
int atoi (const char *nptr);

#include <unistd.h>
int dup2(int oldfd, int newfd);
int chdir(const char *path);
```

dup2() transforme newfd en une copie de oldfd, fermant auparavant newfd si besoin est.
chdir() change le repertoire courant du processus.

```
#include <unistd.h>
int execl (const char *path, const char *arg, ...);
```

Le premier char *arg PAR CONVENTION repete le NOM de la commande, et les autres contiennent les eventuels arguments. Le DERNIER des char *arg DOIT etre obligatoirement nul.