Mini-projet de développement client/serveur

L3 MIAGE

Décembre 2010

Intervenants : Vincent Danjean, Guillaume Huard

Résumé

Ce projet de développement a pour objectif la mise en place d'un système informatique pour la gestion d'un stock de marchandises d'un vendeur et de la préparation des factures des ventes effectuées. Le système doit fournir un ensemble de fonctions (Partie 1) et doit être architecturée selon le modèle client-serveur (Partie 2). Il doit gérer les informations de stock et de facturation de manière persistante (Partie 3) et doit utiliser un protocole de communication bien défini entre les clients et les serveurs (Partie 4). La communication entre clients et serveurs doit être basée sur le protocole de transport TCP.

Table des matières

1 Fonctionnalités du système automatisé

Le système informatique doit gérer un stock de composants électroniques et savoir combien doit payer chaque client.

1.1 Données manipulées

1.1.1 Gestion du stock

Le système informatique dispose d'une base de données où sont décris tous les composants électroniques de la boutique. Pour chacun des composants, on dispose des informations suivantes :

- la référence précise du composant (ex : Résistance de 100 Ω) ;
- la famille du composant (ex : Résistances) ;
- le prix unitaire de ce composant (ex : 0,12 EUR)
- le nombre total d'exemplaires en stock (ex : 15).

Deux composants distincts diffèrent par leur référence (ils peuvent faire partie de la même famille par contre). Le nombre maximum de composants (`NombreMaximumDeComposant`) gérés par la boutique est de 1000. On considère dans le cadre de ce projet que le stock préexiste.

1.1.2 Facturation des clients

La boutique dispose d'un fichier qui décrit l'ensemble des factures que les clients devront acquitter. Ce fichier contient pour chaque client :

- le nom du client (ex : Maurice Dupont) ;
- le total de la facture en cours pour ce client (ex : 104,5 EUR)

Dans le cadre de ce projet, on ne demande pas le détail des factures. On ne s'occupe pas non plus du payement effectif qui acquitterait la facture.

1.2 Opérations possibles sur les données

Le système informatique doit permettre de réaliser les opérations suivantes :

consulter le stock d'un composant : en donnant la référence d'un composant, on doit pouvoir récupérer les informations le concernant (quantité en stock, prix unitaire, etc.) ;

rechercher un composant : en donnant une famille de composant, on doit pouvoir récupérer toutes les références des composants de cette famille. Seules les références dont le stock n'est pas nul doivent être retournées ;

acheter un composant : un client doit pouvoir acheter un composant en stock. S'il demande plus d'exemplaires qu'il n'y en a en stock, la vente doit être complètement refusée ;

payer une facture : un client peut payer ce qu'il doit à la boutique ;

consulter une facture : il doit être possible de voir la facture correspondant à un client ;

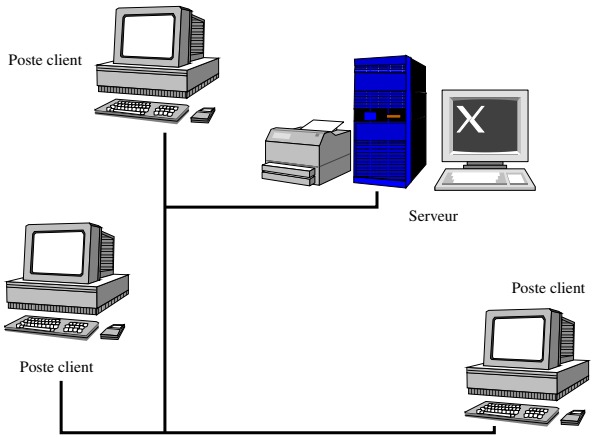
ajouter un produit : on peut ajouter un certain nombre d'exemplaires d'un produit dans le catalogue (la référence du produit doit déjà exister).

2 Architecture du système informatisé

Notre vendeur possédant plusieurs boutiques, on souhaite mettre en place un système informatique composé des éléments suivants :

- un serveur central situé dans une zone non accessible au public ;
- des postes clients répartis dans chacune des boutiques.

Les postes clients doivent permettre, grâce à une interface homme-machine appropriée (qui sera simulée ici sous forme d'une zone de saisie en mode caractères), de réaliser les différentes opérations prévues.



Le fonctionnement sera le suivant :

1. Selon la saisie de l'utilisateur, le client préparera une requête à envoyer au serveur.
2. La requête sera envoyée au serveur et le client se mettra en attente de la réponse.
3. Le serveur réceptionnera la requête et la traitera pour comprendre la demande du client.
4. Il effectuera ensuite le traitement associé,
5. et enverra le résultat de ce traitement au client.
6. Le client réceptionnera le résultat et pourra enchaîner sur une nouvelle requête.

3 Gestion des données

Le serveur central a la charge de l'ensemble des données de la boutique (stock et facturation). Les données manipulées par le serveur sont de nature persistante, c'est-à-dire qu'elles doivent survivre à l'exécution de l'application. Pour assurer la persistance de ces données, nous les stockerons dans un fichier. Le principe consiste alors à charger ce fichier en mémoire pour traiter chaque requête client, puis de sauvegarder ce fichier après chaque mise à jour. Un mécanisme de synchronisation devra donc être utilisé pour gérer la concurrence d'accès au fichier. Des détails techniques sur la synchronisation des accès aux fichiers sont fournis en annexe B.

4 Gestion des échanges client/serveur

Les échanges entre les clients et le serveur doivent suivre un protocole bien défini pour que le serveur comprenne les requêtes des clients et pour que les clients comprennent les résultats renvoyés par le serveur. Les échanges doivent être considérés au niveau applicatif et au niveau transport.

Au niveau applicatif, des structures décrivant les messages vous sont proposées. Vous pouvez les modifier si vous le désirez (mais ce n'est pas nécessaire). Ces structures contiennent de nombreux champs. Seule une partie de ceux-ci seront utilisés (cela dépendra du type requête et de réponse par exemple).

Au niveau transport, il est demandé d'utiliser le protocole TCP/IP. Le projet contient des exemples de communication UDP, l'annexe D donne des détails techniques sur le protocole TCP/IP.

5 Consignes

5.1 Travail de base demandé

Vous disposez d'un squelette de programme. Après lecture des différents documents, il faudra le compléter pour fournir :

1. **Aspect réseau :** un client et un serveur en mode TCP ;
2. **Aspect système :** un serveur parallèle. Il s'agit ici de modifier votre serveur TCP pour qu'il puisse gérer des requêtes en parallèle. Le serveur doit donc créer un nouveau processus chaque fois qu'il reçoit une requête (voir l'annexe C) ;
3. **Aspect programmation :** une sauvegarde/rechargement des données dans des fichiers. Cela permet que les informations (stocks, etc.) soient persistantes et non pas limitées à la session en cours. Attention, le format des fichiers est partiellement imposé (cf Annexe A.2).

5.2 Travaux complémentaires

Après la réalisation du travail demandé dans la section précédente, vous pourrez choisir un ou plusieurs points présentés ici pour améliorer votre programme.

1. **Aspect réseau :** Faites en sorte que votre serveur et votre client puissent fonctionner indifféremment en TCP ou en UDP. Le protocole TCP ou UDP sera choisi selon la valeur d'un argument de la ligne de commande ;
2. **Aspect programmation :** Optimisez (en taille et en efficacité) les structures de données utilisées, en particulier celles utilisées pour le protocole de communication entre le serveur et le client. Faites en sorte qu'il n'y ait plus de limite dans les requêtes à réponse multiple (recherche/liste) ;
3. **Evolution fonctionnelle :** Introduisez la notion de profil en distinguant un profil administrateur d'un profil vendeur. Désormais, l'administrateur sera le seul à avoir le droit d'effectuer une requête listant les clients et récupérant les factures.

Dans tous les cas, avant de commencer à coder ces questions, faites une analyse des problèmes que vous voulez résoudre et des solutions que vous allez proposer. Si cette analyse est bien faite, le codage sera facile. Une bonne analyse (même sans implémentation) dans le rapport et la soutenance sera fortement valorisée.

5.3 Organisation du travail

Le travail s'effectuera sur la machine `jpp`, `mandelbrot` ou `imasrv-compil`. **Restez toujours sur la même machine** pour éviter des problèmes de recompilation partielle. Les groupes travailleront en quadrinôme. La répartition des tâches entre les quatre est libre. Nous vous conseillons de :

– bien respecter le cahier des charges ;
– bien gérer le temps qui vous est imparti ;
– bien discuter dans le groupe ;
– réfléchir avant de programmer.

Pour la réalisation logicielle, nous conseillons vivement de suivre les étapes suivantes :

1. chargement et sauvegarde des données ;
2. réalisation d'une communication en TCP ;
3. réalisation d'une requête (au choix, par exemple la consultation de livres par titre) ;
4. traitement de l'ensemble des requêtes possibles ;
5. ajout du mode parallèle ;
6. ajout des fonctionnalités supplémentaires.

Note : certaines étapes peuvent être faites en parallèle. Pensez à vous répartir le travail.

Support

Les enseignants seront présents pendant une partie seulement du temps qui vous est réservé pour ce projet. Le planning de présence vous sera donné lors au début du projet (lors de la formation des groupes). Vous pouvez aussi poser des questions par mail [1] à :

– `Guillaume.Huard@imag.fr`
– `Vincent.Danjean@imag.fr`
– `Goran.Frehse@imag.fr`
– `Nicolas.Palix@imag.fr`

5.4 Évaluation du travail

Soutenance

Chaque groupe aura 15 minutes pour présenter son travail. Le groupe expliquera brièvement la répartition du travail dans le quadrinôme, l'avancement du projet et les problèmes rencontrés. L'enseignant posera ensuite des questions et demandera également des démonstrations.

Rapport

Le quadrinôme doit rendre juste avant la soutenance un rapport de 3 pages maximum donnant :
– l'organisation du travail dans le groupe ;
– la méthodologie utilisée dans le développement ;
– la pertinence de certains choix ;
– l'état courant du projet et ce qu'il reste à réaliser ;
– les difficultés rencontrées ;
– un rapide bilan de ce que vous a apporté ce projet.

Le rapport ne doit pas redonner des informations présentes dans ce document.

1. Attentions : les enseignants ne sont pas 24h sur 24 devant leur mail, voire même sont en déplacement sans accès à leur mail. Plus la question est précise (nécessitant donc une réponse courte), plus la réponse sera rapide.

Code source

Vous devrez également fournir l'intégralité de votre code source sous format tar compressé. N'oubliez pas de documenter fortement vos programmes. L'archive doit être nettoyée i.e ne doit pas contenir des fichiers objets ou des exécutables.

Annexes

Annexe A Lecture et sauvegarde des données

A.1 Principe général

Les données gérées par le serveur sont sauvées dans un fichier. Afin de permettre à plusieurs processus de fonctionner en même temps (voir la section B suivante pour plus de détails), à chaque requête, les données sont entièrement chargées depuis le disque vers la mémoire, puis elles sont sauvées à la fin de la requête.

Afin de simplifier le travail demandé, les opérations de chargement et sauvegarde sont globales : on charge et on sauve la totalité des données. En particulier, vous ne devrez pas tenter de ne réécrire dans le fichier que ce qui a été modifié.

Quand les données sont chargées, on peut y accéder en utilisant les tableaux globaux définis dans le fichier `BdD.h`. Le code de chargement doit lire les fichiers et écrire dans ces tableaux. Le code de sauvegarde doit lire ces tableaux et écrire dans les fichiers.

Vous êtes libre de choisir le ou les noms des fichiers contenant les données.

A.2 Format des fichiers

Afin d'être compatible avec d'autres logiciels (remplissage des stocks, payement des factures, etc.), le format des fichiers est en partie imposé.

Fichier des stocks. Le fichier contenant les stocks doit contenir une suite d'enregistrements. Chaque enregistrement correspond à un des composants. Tous les enregistrements doivent occuper le même nombre d'octets sur le disque.

Fichier de facturation. Le fichier contenant les sommes dues par les clients doit être lisible/modifiable avec un éditeur de texte quelconque (i.e. les informations doivent être au format texte).

Annexe B Accès aux fichiers et synchronisation

Lorsque plusieurs programmes (ou plusieurs instances d'un même programme) tentent d'accéder simultanément à un même fichier, il peut se produire des conflits. En effet, chaque processus ouvrant un fichier reçoit du système un *file descriptor* (*fd*). Ces descripteurs sont utilisés par les processus pour chaque accès disque. Ils permettent au système de savoir, en particulier, où en est le processus dans la lecture ou l'écriture du fichier.

Si plusieurs programmes tentent, par exemple, d'écrire en même temps dans le même fichier, le système va simplement stocker dans le fichier les informations écrites par le dernier processus à effectuer ses modifications. **Il y a un risque d'écrasement de données.** Cela est illustré sur la figure 1.

La solution est de limiter l'accès au fichier grâce à un verrou. Il existe deux types de verrous :

le verrou partagé : plusieurs processus peuvent avoir ce verrou en même temps. Il est utile par exemple si on veut simplement consulter le fichier : d'autres processus peuvent le consulter en parallèle, ça ne posera pas de problème ;

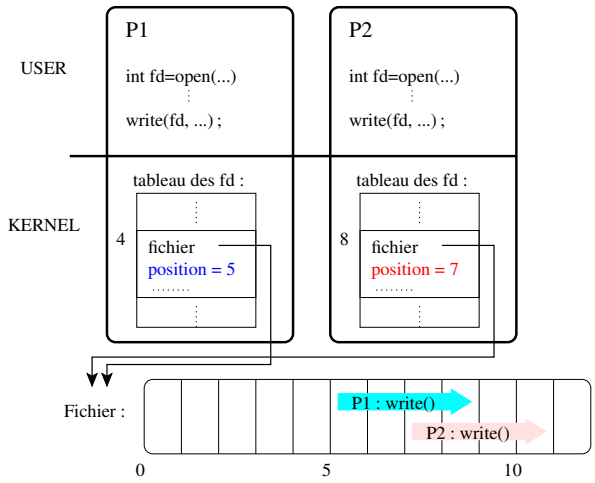


FIGURE 1 – Accès à un fichier par deux processus en parallèle

le verrou exclusif : un seul processus peut avoir un tel verrou. Il est utile si l'on veut modifier le fichier : il ne faut pas que d'autres processus soient en cours de lecture ou de modification du fichier pendant que l'on fait notre modification.

Bien évidemment, si un processus B tente d'acquérir un verrou exclusif alors qu'un autre processus A a déjà un verrou (exclusif ou partagé), alors le processus B est mis en attente par le système d'exploitation tant que le processus A n'a pas relâché le verrou.

De manière symétrique, si un processus B tente d'acquérir un verrou partagé alors qu'un autre processus A a déjà un verrou exclusif, alors, là encore, le processus B est mis en attente par le système d'exploitation tant que le processus A n'a pas relâché le verrou.

Par contre, si un processus B tente d'acquérir un verrou partagé alors qu'un autre processus A a déjà un verrou partagé, alors B obtient immédiatement le verrou qu'il a demandé. On a alors maintenant deux processus possédant chacun un verrou partagé sur le fichier.

Pour prendre ou relâcher un verrou, on utilisera la fonction système `flock()` (voir la page de manuel pour son emploi).

Pour assurer un fonctionnement correct d'un serveur multiprocessus, il faut que l'accès aux fichiers soit verrouillé pendant toute la durée du traitement de la requête. Cela vous est donné en commentaire dans le fichier `serveur.c`.

Annexe C Architecture serveur

C.1 Serveur itératif (ou accès sérialisés)

Les requêtes sont traitées séquentiellement par le même processus en mode FIFO (premier entré/premier sorti). Mais pour éviter de perdre des requêtes qui arriveraient pendant un traitement, l'attente et la mémorisation des requêtes dans la file d'attente continuent à être gérées par le système (dans la limite indiquée lors du *listen*) durant le déroulement de chaque traitement.

Il peut y avoir plusieurs clients qui dialoguent en même temps avec le serveur, il y a dans ce cas toujours une seule file d'attente et le serveur différenciera les requêtes au moment de leur traitement. La gestion de la file d'attente des requêtes est faite par le système (le réseau joue le même rôle qu'un

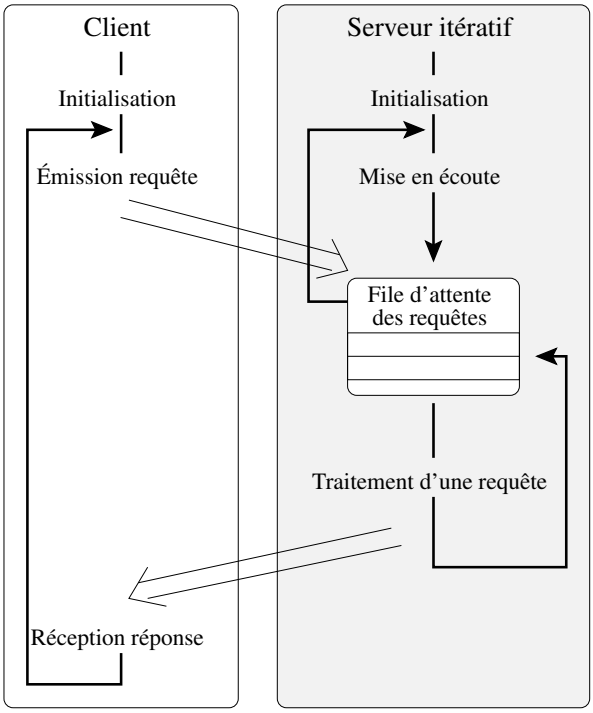


FIGURE 2 – Serveur itératif

tube de communication au sens Unix). Le nombre de requêtes de connexion pouvant être mis en attente est fixé par un paramètre donné à la commande listen (par défaut 5).

Ce type de serveur est facile de conception, mais il se peut que certaines requêtes dont le temps de traitement est très élevé, créent un délai inacceptable pour le traitement de requêtes d'autres clients. On utilisera dans ce cas les serveurs dits parallèles.

C.2 Serveur parallèle (ou accès concurrents)

Les différentes requêtes sont traitées en parallèle, chacune nécessitant la création d'un processus dédié à son traitement.Lorsqu'une requête est acceptée et le service correspondant identifié, le serveur crée en général un processus fils (ou un thread) qui prend en charge la communication et le traitement applicatif, alors que le serveur retourne à l'écoute des autres requêtes.

Une requête "trop longue" ne pourra ainsi plus retarder d'une façon trop importante d'autres traitements.

Annexe D Protocole de transport

Vous devez assurer le dialogue entre les environnements client et serveur. Les fonctions minimales à assurer sont :

- gérer la boucle sur la socket permettant l'arrivée de nouvelles connexions ;
- assurer la partie réception et envoi des messages sur les connexions ouvertes ;
- assurer l'encodage et le décodage des messages ;
- orienter les traitements en fonction des messages reçus.

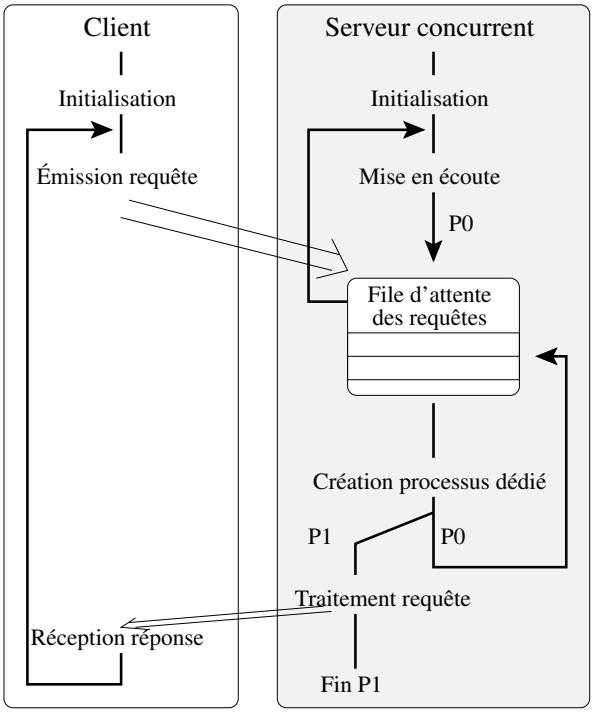


FIGURE 3 – Serveur parallèle

Vous devez permettre au client et au serveur de communiquer dans le cadre d'une connexion persistante et ne pas couper la connexion après chaque donnée fournie par le serveur au client, sauf à la fin de la transaction initiée par le client.

Nous nous baserons sur les protocoles de la famille TCP/IP (correspondant aux couches 3 et 4 de la norme OSI) et l'interface d'accès à ces protocoles que nous utiliserons s'appelle les Sockets BSD (Berkley Software Distribution). Ce sont les mêmes primitives que vous avez utilisées avec SOCKLAB dans le TP2 réseaux.

Pour vous simplifier le travail, vous aurez à votre disposition une boîte à outils (fon.c) où vous pourrez retrouver les procédures de manipulation des sockets d'UNIX auxquelles quelques simplifications ont été apportées (se référer à la documentation technique sur l'utilisation des sockets).

Le concept client/serveur est devenu la méthode incontournable pour la communication au niveau application quand le protocole utilisé est de la famille TCP/IP. En effet les protocoles de la famille TCP/IP ne fournissent aucun moyen de lancer un programme à distance grâce à l'envoi d'un message. De ce fait dans une communication entre 2 parties, il faut forcément qu'il y en ait une qui soit en attente permanente d'une requête éventuelle de l'autre. Les applications sont donc classées en 2 catégories :

les clients : ils prennent l'initiative du début de la communication (demande d'ouverture de connexion, envoi de requête, attente de réponse)

les serveurs : ils sont en permanence à l'écoute, en attente de demande de communications (attente de demande d'ouverture de connexion, de requête, émission de réponse).

La procédure générale de dialogue entre un serveur et son ou ses clients est la suivante :

1. le serveur initialisé se place en écoute (attente de requête client) ;
2. le client émet une requête à destination du serveur. Il demande un service ou l'accès à une ressource ;

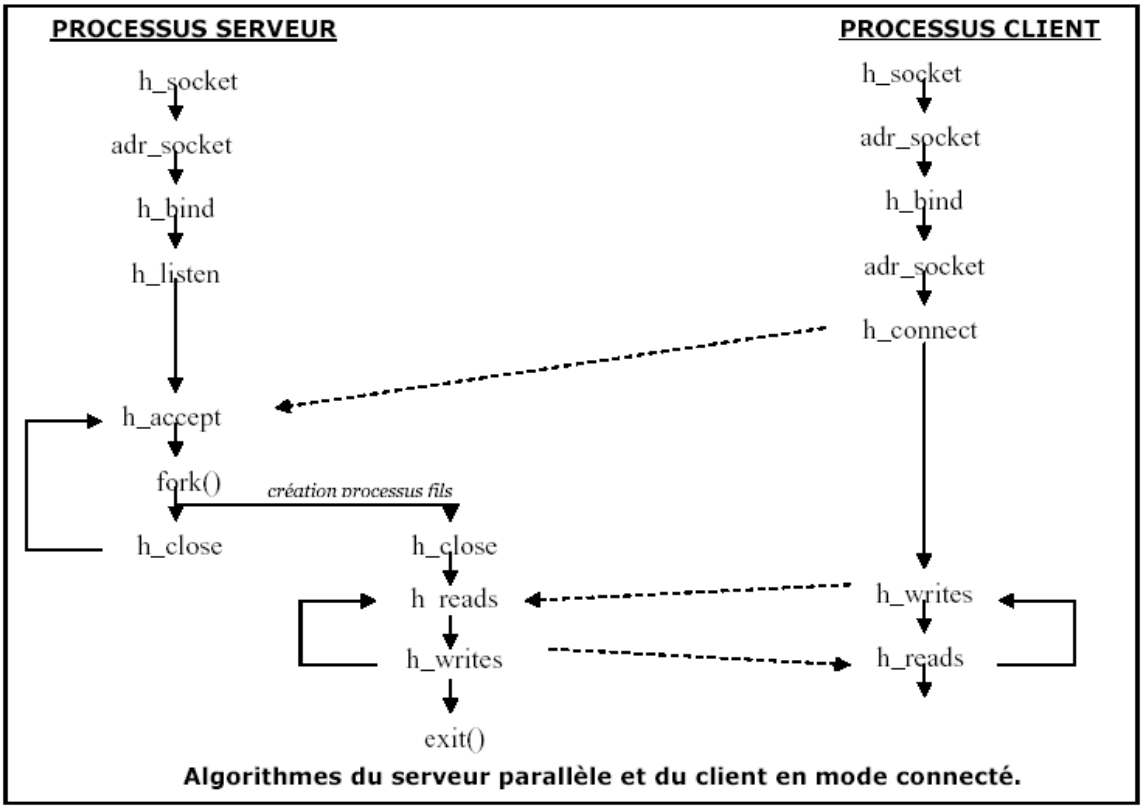


FIGURE 4 – Algorithmes du serveur parallèle et du client en mode connecté.

3. le serveur renvoie une réponse au(x) client(s). Il rend le service ou octroie la ressource, éventuellement il engage même un dialogue assez conséquent avec son (ses) client(s) ;
4. le serveur se replace en écoute (attente de nouvelle requête client).

C'est toujours le client qui a l'initiative de l'échange !

Remarque : Il est possible d'utiliser une option de compilation `DEBUG` du programme fon.c qui vous donne le statut des communications.

On peut écrire des applications dites *orientées connexion ou non*, suivant le protocole transport utilisé. Par définition, un serveur utilisant TCP est dit orienté connexion et un serveur utilisant UDP est dit orienté sans connexion. Suivant l'application à réaliser, la fiabilité et la performance attendues des communications et la gestion des reprises sur erreurs, le choix de l'un ou l'autre de ces algorithmes de base sera plus ou moins adéquat. **Pour les différents algorithmes et les différentes fonctions utilisées se référer à la documentation sur les sockets**.

Annexe Références

[1] *Manuels UNIX*. `man socket`.

[2] Douglas Comer. *TCP/IP : Architecture, protocoles, applications*. InterEditions.

Documentation technique sur l'utilisation des sockets

Table des matières

1. INTRODUCTION

L'interface d'accès au réseau que nous utiliserons s'appelle les Sockets. L'interface *socket* d'accès au transport Internet a été développée dans la version BSD du système Unix (Berkeley Sofware Distribution) sous l'impulsion de l'*Advanced Research Project Agency* (ARPA) à l'université de Californie à Berkeley. Elle est de fait au cours des années devenue un standard de communication dans le monde Unix et donc dans le monde des stations de travail. En fait les architectes de Berkeley en définissant les *sockets* ont choisi d'offrir une interface qui permette d'accéder non seulement au transport d'Internet, mais également à des à des communications utilisant d'autres protocoles. Les primitives disponibles sur les *sockets* permettent au programmeur de spécifier le type de service qu'il demande plutôt que le nom d'un protocole spécifique qu'il désire utiliser.

Pour vous simplifier le travail (programmation de la gestion des erreurs, des aides à la mise au point ...), pour chaque procédure de l'interface a été développé une procédure que vous pourrez utiliser.

Ainsi, chaque procédure à votre disposition (voir fon.h)⊡

- encapsule une primitive UNIX en conservant la même sémantique⊡la procédure de nom ***h_ primitive()*** encapsule la procédure Unix de nom ***primitive()*** *(sauf adr_socket)*.
- réalise les manipulations de données fastidieuses.
- réalise des contrôles sur les paramètres et rend des messages d'erreur.
- permet d'obtenir avec une option de compilation (voir makefile) des traces à l'exécution des primitives

*** Le concept de socket**

Un *socket* est un point d'accès a un service de communication qui peut être réalisé par différentes familles[1] de protocole. Il est créé dynamiquement par un processus et est désigné par ce processus par un nom local qui est l'index de son descripteur dans la table des descripteurs du processus.

Lorsque l'on crée un socket, on doit préciser la famille de protocoles de communication que l'on désire utiliser, ainsi que la type de service que l'on veut obtenir.

[1] On dit également domaine, on parle ainsi du domaine Internet, du domaine ISO etc...

2. PARAMETRES UTILISÉS DANS LES PROCEDURES

Ces paramètres apparaissent dans les différentes procédures. Pour chacun d'eux, nous donnons la signification, le type en C et les différentes valeurs qu'il peut prendre.

- *Le domaine d'utilisation (int domaine)*

Il indique la famille de protocoles de communication⊡ pour TCP/IP, il prendra la valeur **AF_INET**.

- *Le mode de communication (int mode)*

C'est un entier qui représente le type de protocole transport utilisé sur une socket⊡

  - mode connecté⊡ **SOCK_STREAM** (TCP⊡ flot d'octets)
  - mode non connecté⊡**SOCK_DGRAM** (UDP⊡datagrammes)

- *L'identificateur de socket (int num_soc)*

L'ensemble des points d'accès aux diverses communications en cours pour un processus donné est accessible à travers un tableau de pointeurs de descripteurs. Chacun de ces descripteurs de Socket contient les différents paramètres propres à une communication⊡mode de communication, adresses IP source et destination ...

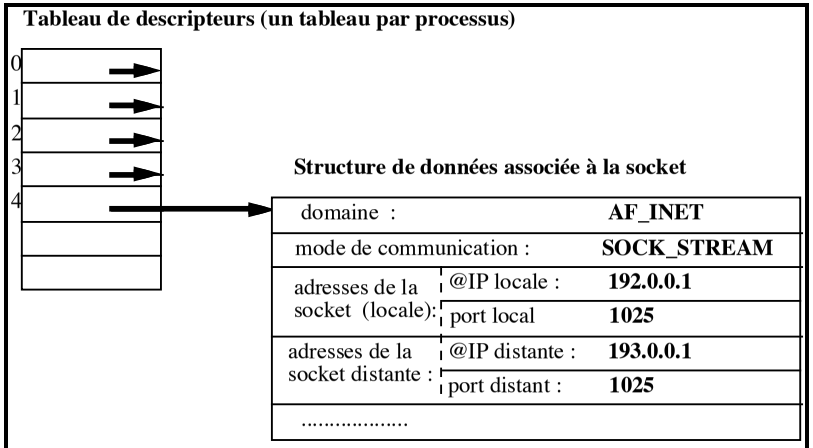


Figure: Descripteur de Socket

Les primitives de l'interface manipulent l'indice dans ce tableau pour faire référence à un point d'accès donné, cet indice est l'identificateur de socket.

Remarque L'identificateur de socket de valeur nulle correspond au clavier et peut être utilisé de la même manière que toutes les autres sockets.

- *Les adresses socket* (struct sockaddr_in *p_adr_socket)

Le couple d'adresses (@IP, numéro de port) permet d'identifier la socket au niveau réseau INTERNET. Ce couple sera appelé par la suite adresse socket. Ce couple (@IP, numéro de port) doit être contenu dans une structure *sockaddr_in*. (dans le cas particulier d'Internet) qui sera passée en paramètre aux différentes procédures de l'interface

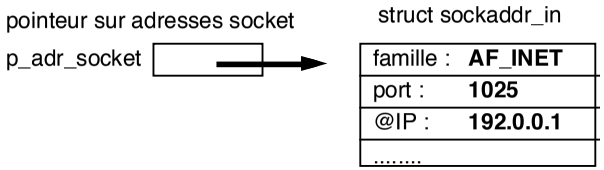


Figure: structure d'adresses d'une socket

La convention de notation des différents pointeurs de cette structure sera

p_adr_client : pointeur sur l'adresse socket du processus client
p_adr_distant : pointeur sur l'adresse socket du processus distant
p_adr_local : pointeur sur l'adresse socket du processus local

Remarque Vous n'aurez normalement pas à manipuler directement cette structure, une procédure d'instanciation est à votre disposition pour cela (voir adr_socket plus loin).

- *name* (char *name)

C'est une chaîne de caractère correspondant à un nom de machine (alias de @IP dans le fichier /etc/hosts). Pour une utilisation sans modification du fichier /etc/hosts, on pourra passer ce type de paramètre sous la forme adresse IP en décimale (ex : "192.0.0.1").

- *service* (char *service)

C'est le nom du service (alias du n° de port dans /etc/services) il s'agit d'une chaîne de caractères qui identifie le numéro de port. Pour une utilisation sans modification du fichier

/etc/services, on pourra passer ce type de paramètre sous la forme numéro de port en décimal (ex : "1234").

3. LES PROCEDURES DE l'INTERFACE

3.1. Renseignement des adresses d'une socket : *adr_socket*

Cette procédure est spécifique à la boîte à outils, il n'y a pas de primitive UNIX équivalente. Elle permet d'affecter les différents champs d'un descripteur d'adresse de socket (structure sockaddr_in) avant que celui-ci soit passé à différentes procédures (bind , sendto , connect).

void ***adr_socket*** *(service, name, protocole, p_adr_socket, type_proc)*

```
        char *service;          /* nom du  service lié au port socket à renseigner */
        char *name; /* chaine de caractère correspondant à une adresse IP, soit figurant
dans le fichier /etc/hosts soit sous la forme décimal pointé (« 192.0.0.1 ») */
        char *protocole;        /* protocole de communication  */
        struct sockaddr_in  *p_adr_socket;/* pointeur sur les adresses socket  à renseigner
*/
        int type_proc;          /* type du processus  local CLIENT ou SERVEUR ) */
```

Cette procédure renseigne les adresses de la socket (@IP, port) à partir

- du nom du service : nom dans /etc/services ou numéro de port en décimal
- du nom: nom dans /etc/hosts ou @ IP en décimal pointé
- du protocole de la socket : « udp » ou « tcp »
- Le type du processus : CLIENT ou SERVEUR qui va utiliser localement cette socket est nécessaire. Dans le cas d'un programme client (resp. serveur) ce sera CLIENT (resp. SERVEUR) pour toutes les structures manipulées. (voir source de **fon.c** pour plus de détails).

La structure sockaddr_in se présente comme suit

```
struct sockaddr_in

        {
                short sin_family;       /* famille d'adresse : AF_INET*/
                ushort sin_port;        /* numéro de port */
                ulong sin_addr;         /* adresse de niveau 3 : IP*/
                char sin_zero [8];      /* inutilisé  (mis à zéro) */
        }
```

Elle est en fait la forme particulière d'un champs de la structure plus générale **sockaddr** qui est prévue pour différentes familles de protocole et d'adresse.

Dans le cas d'un processus client, on peut laisser le choix de numéro de port au système (allocation dynamique) qui gère l'affectation de ces numéros, il faut alors donner la valeur «□» au numéro de port.

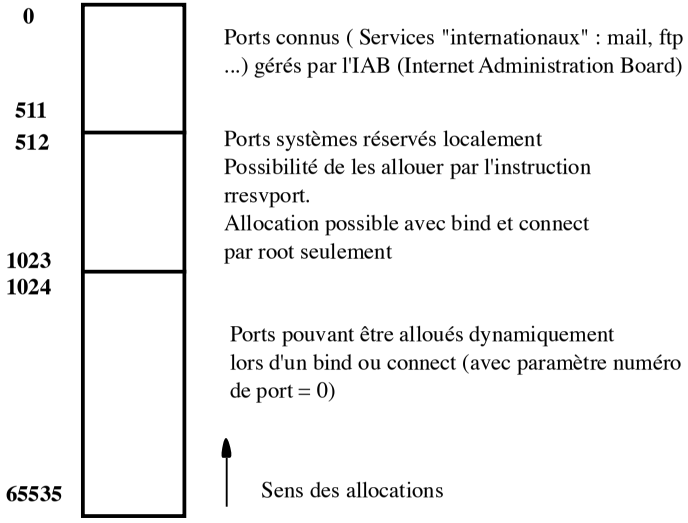
Si l'on veut récupérer le nom de la machine sur laquelle s'exécute le programme, on peut utiliser la procédure **gethostname** (voir plus loin sur l'utilisation de cette procédure).

Exemple: gethostname (myname, lg);
adr_socket (“□□, myname, “□dp□, p_adr_socket, CLIENT);

Dans le cas d'un processus serveur, le port du service ne doit pas changer afin d'être toujours connu par les clients potentiels. Il doit donc être défini une fois pour toute par le processus serveur. Dans ce cas la machine doit être accessible sur tous les réseaux auxquels elle est directement reliée (cas de plusieurs interfaces physiques)□la socket est donc définie pour l'ensemble des @IP de la machine serveur. Dans ce cas, l'adresse IP est définie de façon particulière par le processus serveur (paramètre *type_proc = SERVEUR*)□Le paramètre *name* n'est alors pas pris en compte.

Exemple: adr_socket (“□111□, myname, “□dp□, p_adr_socket, SERVEUR);

La figure suivante donne l'affectation des différentes plages de numéros de port. (on peut avoir la liste des connexions (adresses Internet et ports) ouvertes sur une machine par la commande **netstat -an**).



Figure□affectation des numéros de port

3.2. Création d'une socket□“*h_socket*

int ***h_socket*** *(domaine, mode)*

int domaine□ /* *AF_INET* */
int mode□ /* *SOCK_STREAM ou SOCK_DGRAM* */

Cette procédure crée une nouvelle structure de données de socket, et une nouvelle entrée dans la table des descripteurs, qui pointe sur cette structure.

Elle retourne l'entier qui est l'identificateur de la nouvelle socket. Il correspond à l'indice dans le tableau des pointeurs de descripteurs de socket.

Les adresses de la socket locale ne sont pas instanciées. A ce stade, la socket n'est donc pas identifiée au niveau du réseau. En particulier, les adresses de la socket distante ne sont pas renseignées□la communication n'est donc pas encore définie.

Ces adresses seront renseignées ultérieurement, par l'appel d'autres fonctions (“□CONNECT□,□BIND□, ...).

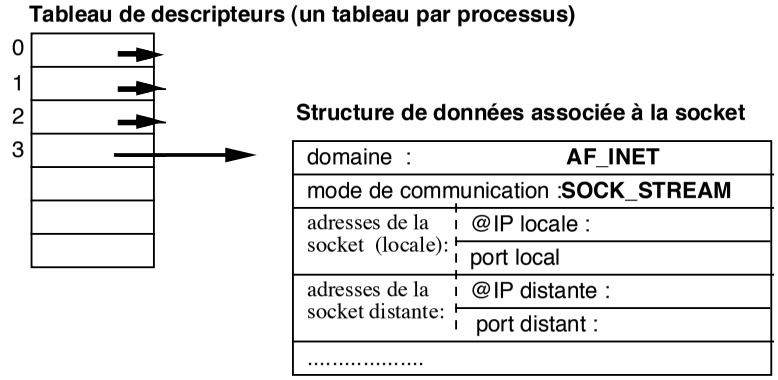


Figure: Structure de données créée par la procédure h_socket.

3.3. Association d'une socket à ses adresses□“*h_bind* “□

void ***h_bind*** *(num_soc, p_adr_local)*

int num_soc□ /* *n° de socket* */
*struct sockaddr_in *p_adr_local*□ /* *adresses (locales) de la socket (locale)* */

Le **BIND** réalise l'instanciation des adresses locales dans le descripteur de socket dont l'identificateur lui est passé en paramètre.

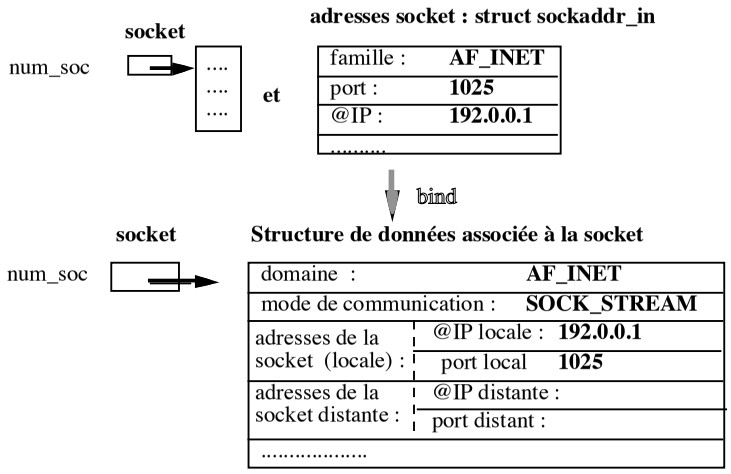


Figure : la procédure h_bind

3.4. Demande de connexion "*h_connect*"

void ***h_connect*** *(num_soc, p_adr_distant)*

```
int num_soc;                        /* n° de socket */
struct sockaddr_in *p_adr_distant;  /* adresses de la socket distante */
```

Cette procédure est utilisée dans les applications clients qui fonctionnent en mode connecté. Elle réalise la connexion TCP entre la socket d'identificateur *num_sock* d'un processus client et une socket d'un processus serveur dont l'adresse est fournie par *p_adr_distant*..

Il faut donc que le serveur soit en attente de connexion. Elle effectue l'instanciation de l'adresse distante dans la socket locale avec les paramètres fournis.

Remarque Dans le cas de TCP (l'interface socket peut servir à d'autres protocoles), l'instanciation de l'adresse locale est faite automatiquement avec l'adresse de la machine locale et le bind est donc inutile. Pour avoir une version "standard" il est donc recommandé d'effectuer le bind dans tous les cas.

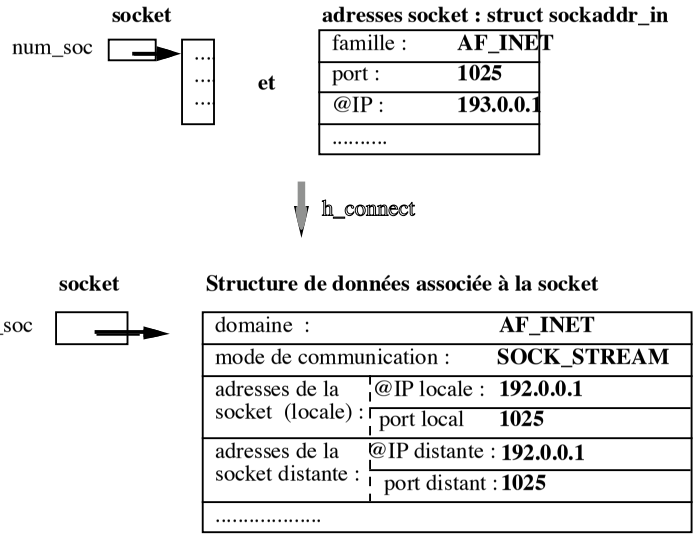


Figure: la procédure h_connect

3.5. Mise en état d'écoute "*h_listen*"

void ***h_listen****(num_soc, nb_req_att)*

```
int num_soc; /* n° de socket */
int nb_req_att;      /* nombre maximum de requêtes en attente */
```

Cette procédure est utilisée en mode connecté pour mettre la socket du serveur en état d'écoute de demandes de connexion des clients, et elle est dite passive. Les demandes de connexion sont mises en file d'attente, et *nb_req_att* est le nombre maximum de requêtes qui peuvent être mises en attente de traitement.

Cette procédure n'est pas bloquante et les réceptions des demandes de connexion se feront en parallèle avec le reste du programme.

La figure suivante résume les conséquences au niveau du réseau des procédures h_connect et h_listen.

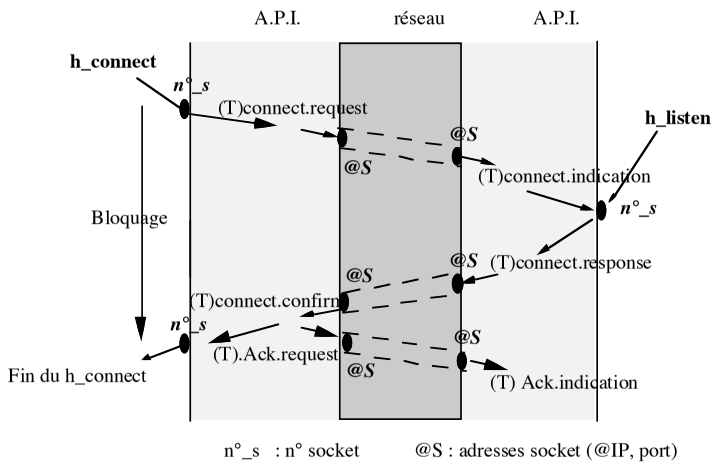


Figure: Etablissement d'une connexion TCP grâce à h_connect et h_listen

Le h_connect provoque au niveau de l'interface du réseau un *(T)connect.request* (service TCP).

Du coté du serveur le *(T)connect.indication* arrive à l'API qui si un *h_listen* est présent sur cette socket génère un *(T)connect.response* au niveau du service TCP. La mémorisation dans une liste d'attente de la demande de connexion TCP se fera au moment de l'ouverture définitive de la connexion (réception du (T)Ack.indication). A l'arrivée, côté client du *(T) connect.confirm* la connexion est supposée établie et un dernier acquittement est envoyé pour finir l'ouverture de connexion TCP. Dans le cas d'une impossibilité d'ouverture de connexion TCP, un code d'erreur est retourné par *h_connect*..

Remarque : il peut y avoir des réceptions de données côté serveur alors que la connexion n'a pas encore été acceptée au niveau de l'application (voir procédure accept plus loin).

3.6. Acceptation de connexion : "*h_accept*"

int ***h_accept*** *(num_soc, p_adr_client)*

```
int num_soc ;                       /* n° de socket */
struct sockadrr_in *p_adr_client ;  /* adresses (port, @IP) du distant (client) */
```

Elle est utilisée en mode connecté par le processus serveur pour accepter une demande de connexion qui a été faite par un processus client. Elle retourne le descripteur d'une nouvelle

socket qui sera utilisée par le processus serveur pour l'échange des données avec le processus client :

- la socket initiale du processus serveur est utilisée pour recevoir les demandes de connexion des processus clients
- pour chaque connexion acceptée avec un processus client, une nouvelle socket est créée pour l'échange des données.

S'il n'y a aucune demande de connexion dans la file d'attente, "accept" est bloquant : il attend une demande de connexion pour réaliser la connexion qui lui est demandée.

3.7. Lecture de données sur une socket en mode connecté : "*h_reads* "

int ***h_reads*** *(num_soc, tampon, nb_octets)*

```
int num_soc ;       /* n° de socket */
char *tampon ;      /* pointeur sur les données reçues par le processus */
int nb_octets ;     /*  nb octets du tampon */
```

Cette procédure est utilisée en mode connecté par les processus applicatifs pour récupérer les octets transmis par TCP au niveau du port de la socket.

Cette procédure effectue autant d'appels que nécessaire à la primitive UNIX "read" et retourne le nombre d'octets réellement lus. *nb_octets* indique le nombre d'octets que l'on veut recevoir dans la chaîne *tampon*. Elle fonctionne donc avec les mêmes particularités que le read sur fichier standard Unix (idem pour le write).

Si le buffer de réception du port de la socket est vide, le "read" est bloquant : il attend que le buffer soit remplis (par TCP) pour pouvoir réaliser la lecture demandée. La procédure **h_reads** effectuant plusieurs appels à *read* , elle est aussi bloquante tant que le nombre de caractères spécifié n'est pas arrivé.

3.8. Ecriture de données sur une socket en mode connecté : "h_writes"

int ***h_writes*** *(num_soc, tampon, nb_octets)*

```
int num_soc ;       /* n° de socket */
char *tampon ;      /* pointeur sur les données émises par le processus */
int nb_octets ;     /* nombre d'octets émis = nb octets du tampon */
```

Cette procédure est utilisée en mode connecté par les processus applicatifs pour envoyer des octets à travers une socket donnée. Cette procédure effectue autant d'appels que nécessaire à la primitive UNIX "*write*" et elle retourne le nombre d'octets effectivement écrits.

Si le buffer d'émission (géré par le système) du port de la socket est plein, le "write" est bloquant: il attend que le buffer se vide pour pouvoir réaliser l'écriture demandée. La procédure **h_writes** effectuant plusieurs appels à write peut de la même façon être bloquante.

nb_octets indique le nombre d'octets que l'on veut envoyer, contenus dans la chaîne *tampon*.

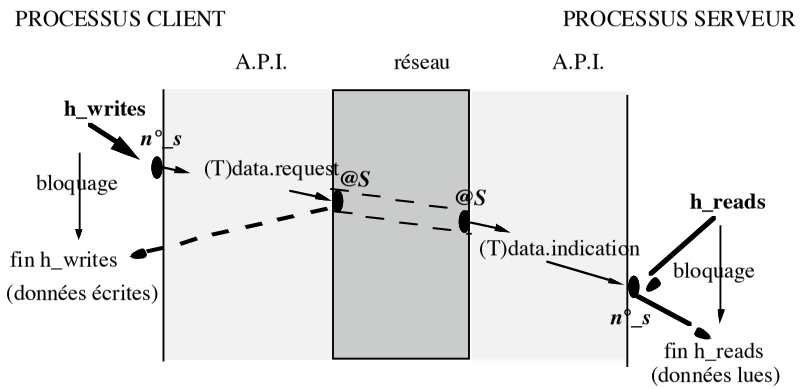


Figure: Echange de données

"write" génère une transmission de données ((T)data.request au niveau des services TCP)

"write" est bloquant si, et tant que le buffer d'émission de TCP est plein.

"read" lit le buffer de réception TCP de la socket. "read" est bloquant si, et tant que celui-ci est vide.

"read" est débloqué à la réception d'un '(T)data.indication' par l'A.P.I.

Remarque: On peut échanger des données en incluant des options avec les primitives *"recv"* et *"send"*.

"send" (respectivement "recv") est identique à "write" (respectivement "read"), dans son fonctionnement, il a seulement un argument supplémentaire d'options, qui permet par exemple d'envoyer une donnée prioritaire.

3.9. Lecture d'une socket en mode non connecté: *"h_recvfrom"*

int ***h_recvfrom*** *(num_soc, tampon, nb_octets, p_adr_distant)*


```
int num_soc;          /* n° de socket */
char *tampon;         /* pointeur sur les données reçues par le processus */
int nb_octets;        /* nombre d'octets reçus = nb octets du tampon */
struct sockaddr_in *p_adr_distant; /* pointeur sur adresses socket distante */
```

Elle est utilisée en mode déconnecté, de la même façon que le "read". Elle retourne le nombre d'octets effectivement reçus, ou un résultat négatif en cas d'erreur.

Contrairement au read qui fonctionne en mode connecté et donc pour une connexion donnée, cette procédure permet de connaître les adresses de la socket distante, pour lui répondre éventuellement. Pour cela elle instancie le paramètre *p_adr_distant* .

3.10. Ecriture sur une socket en mode non connecté: *"h_sendto"*

int ***h_sendto*** *(num_soc, tampon, nb_octets, p_adr_distant)*

```
int num_soc;  /* n° de socket */
char *tampon;         /* pointeur sur les données reçues par le processus */
int nb_octets;/* nombre d'octets à envoyer = nb octets du tampon */
struct sockaddr_in *p_adr_distant;

/* pointeur sur adresses socket distante */
```

Elle est utilisée en mode déconnecté, de la même façon que le "write". Elle retourne le nombre d'octets effectivement émis, ou un résultat négatif en cas d'erreur.

p_adr_distant indique les adresses de la socket distante vers laquelle les données sont émises: ces adresses socket doivent être renseignées avant le "sendto" . On peut utiliser la procédure **adr_socket** précédemment vue.

3.11. Désallocation d'une socket: *"h_close"*

void ***h_close****(num_soc)*

```
int num_soc;          /* n° de socket */
```

Cette fermeture en cas de mode connecté assure que les messages en attente d'émission seront correctement envoyés. Puis elle désalloue et supprime la srtucture de donnée associée à la socket. Cette procédure n'est pas bloquante, c'est à dire que l'utilisateur retrouve la main toute de suite (avant l'envoi d'éventuels messages en attente d'émission).

Au niveau du réseau une demande de déconnexion est envoyé à la couche transport distante, qui ferme la connexion dans un sens. Pour fermer une connexion TCP dans les deux sens il faut donc deux "close", un à chaque extrémité de la connexion.

En pratique, une socket peut être partagée par plusieurs processus : dans ce cas, si n processus partagent la socket, il faut n "close" pour supprimer la socket. C'est le nième "close" qui réalise la désallocation de la socket.

3.12. Fermeture d'une socket : "*h_shutdown*"

void ***h_shutdown****(num_soc, sens)*

```
int num_soc ;        /* n° de socket */
int sens ;           /* sens dans lequel se fait la fermeture :
                     0 : entrées fermées, 1 : sorties fermées, 2 : entrées et sorties fermées  */
```

Cette procédure permet de moduler la fermeture complète d'une socket. Elle ferme la socket seulement dans le sens indiqué par *sens*.. La fermeture de la socket est brutale, il y a perte immédiate des données en attente d'émission ou de réception dans le buffer concerné.

Par exemple la fermeture de la socket dans le *sens sortie* stoppe l'envoi de requêtes et envoie au processus serveur distant un 'EOF'' qui indique la fin des requêtes. Après avoir lu cet EOF, le processus serveur peut donc envoyer les dernières réponses au processus client, puis

fermer la connexion d'échange des données avec celui-ci (par un shutdown dans le sens sorties aussi).

Elle permet aussi de rendre une socket mono-directionnelle et de l'utiliser comme flot d'entrée ou comme flot de sortie.

4. CLIENT - SERVEUR : algorithmes de programmation

4.1. Modes de communication

Comme nous l'avons vu, il existe deux modes de communication client/serveur :

- le mode connecté : protocole de transport TCP
- le mode non connecté : protocole de transport UDP

Le mode connecté sera utilisé pour des échanges de données fiables (mise à jour d'une base de données par exemple) et pour des échanges de type 'flux d'octets' (échanges de gros fichiers par exemple).

Le mode déconnecté est utilisé pour des échanges de messages courts, dont le contenu peut être facilement contrôlé par les processus applicatifs avant d'être traité. Il s'agit d'échanges de données non sensibles aux pertes, que l'on peut facilement répéter, où les erreurs non détectées ne génèrent pas d'incohérences, ni de catastrophes.

4.2. Types de serveurs

Deux types de serveur pourront être utilisés :

- le serveur itératif : Le processus applicatif traite lui-même toutes les requêtes, les unes après les autres. Ce type de serveur présente de forts risques d'engorgement, surtout en mode connecté. (on peut supposer que les réponses sont courtes et traitées rapidement dans le cas du mode non connecté).

- le serveur parallèle, ou à accès concurrent : Il sous-traite à des processus fils (lancés en parallèle) chacune des requêtes. Ce type de serveur est plus employé en mode connecté, car il offre de meilleures performances (temps de réponse plus réguliers). Par contre, son utilisation en mode déconnecté est discutable, car les traitements et réponses aux requêtes sont à priori plus courts et ne justifient pas le coût de gestion des processus fils et de leur évolution en parallèle.

4.3. Serveur itératif

Voici les algorithmes de bases écrits à partir des procédures de la boite à outils dans le cas d'un seul client avec un traitement itératif du serveur.

4.3.1. Mode connecté

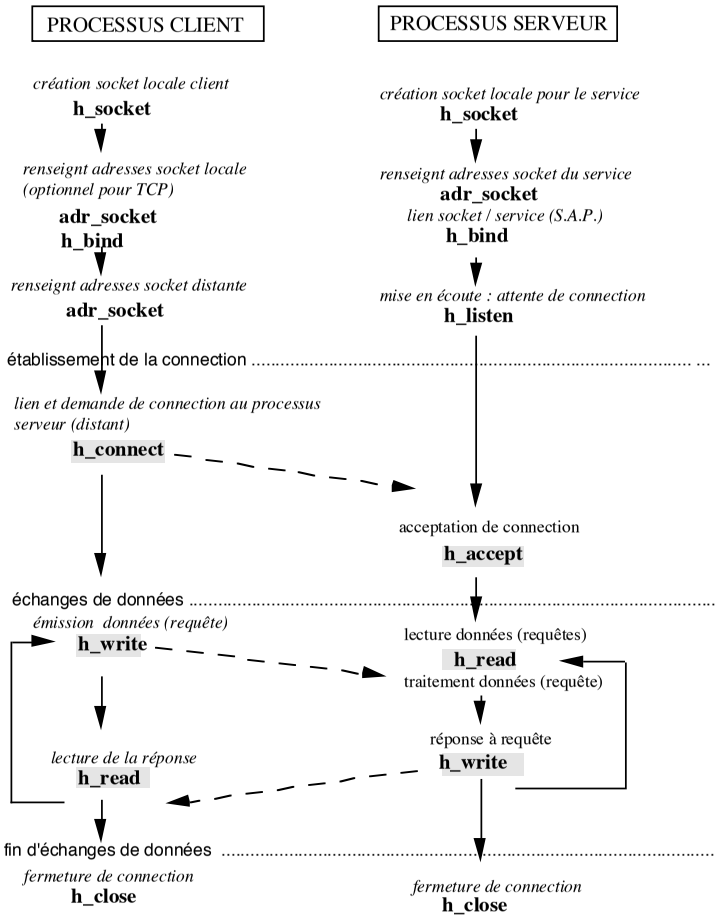


Figure: Algorithmes client et serveur itératif,
en mode connecté (un seul client par serveur)

4.3.2. Mode déconnecté

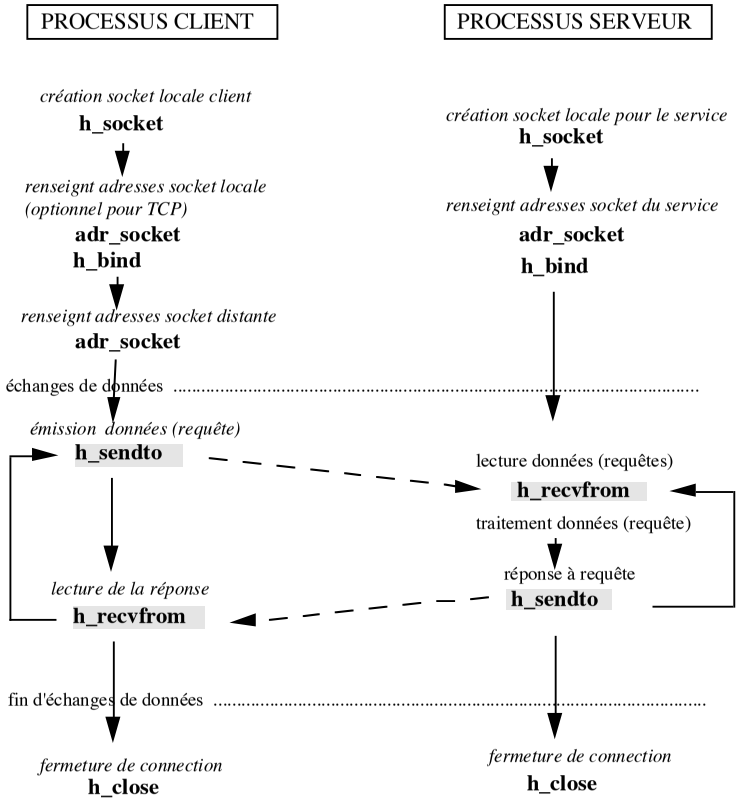


Figure: Algorithmes client et serveur itératif, en mode non connecté

4.4.Serveur parallèle

4.4.1. Mode connecté

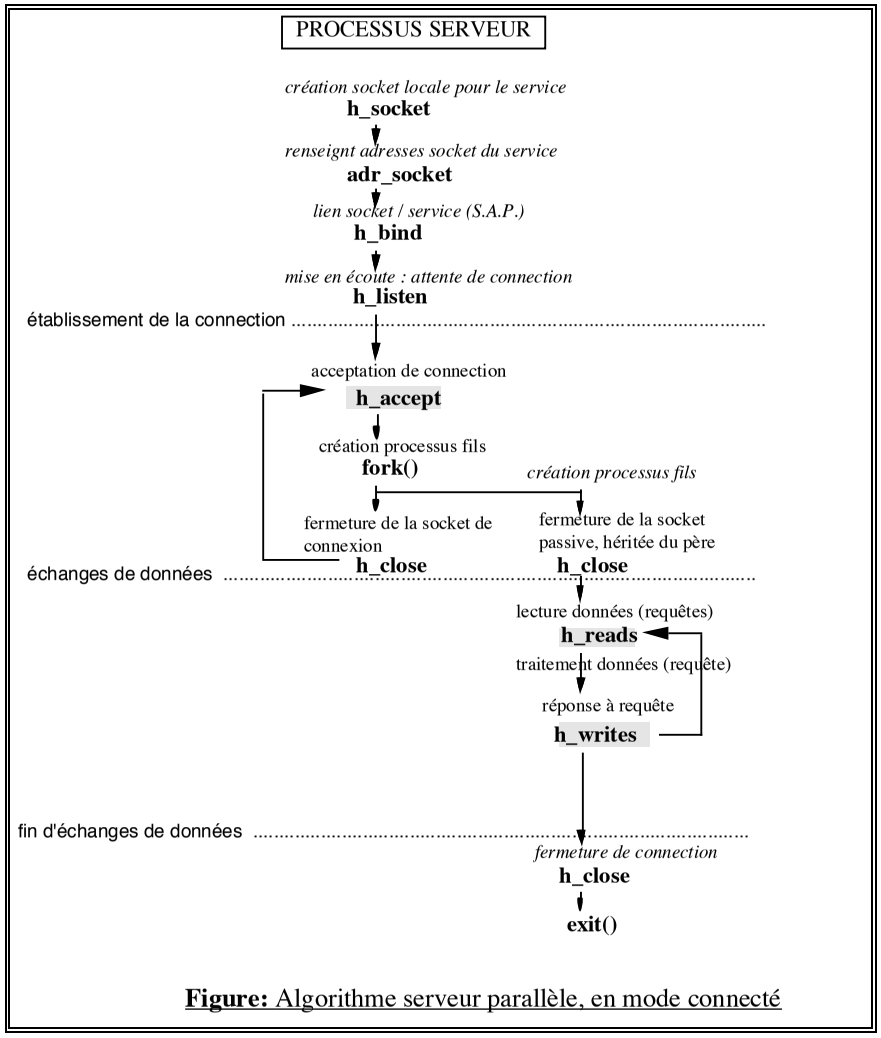


Figure: Algorithme serveur parallèle, en mode connecté

4.4.2. Mode déconnecté

On peut très facilement déduire ce cas du précédent.

4.5. Cas multi-clients

En mode déconnecté les requêtes étant traitée individuellement sur une seule socket, on peut se ramener soit au cas "un seul client", soit générer « à la main » la nouvelle socket (créer par le accept dans le cas du mode connecté).

On peut donc ensuite dans les deux cas connecté ou non connecté, soit lancer des processus fils s'occupant des dialogues avec chaque client, soit gérer un ensemble de requête sur différentes sockets dans le même processus.

Pour ce dernier cas, voici les primitives permettant de gérer un ensemble de sockets (qui sont assimilées à des descripteurs de fichiers).

fd_set type C définissant un ensemble de descripteurs de fichier (ou de socket)

FD_ZERO (fd_set *set))
permet la remise à vide d'un ensemble de socket.

FD_SET (int idsocket, fd_set *set)
ajoute l'identificateur de socket idsocket à l'ensemble set.

FD_CLR (int idsocket, fd_set *set)
supprime l'identificateur de socket idsocket de l'ensemble set.

FD_ISSET (int idsocket, fd_set *set)
retourne 1 si l'identificateur de socket idsocket appartient à l'ensemble set.

int select (int maxfdpl, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

Supprime de l'ensemble de socket readfds les sockets qui ne sont pas en attente de lecture (dont le buffer de lecture est vide).

maxfdpl contient l'identificateur de socket maximum qui sera testé lors du select.

La fonction C **getdtablesize()** permet d'obtenir ce descripteur maximum.

Les trois autres paramètres ne nous serviront pas ici et seront forcés à zéro.

Remarque : l'utilisation de ces fonctions et types nécessite l'inclusion du fichier <sys/types.h>

Exemple d'utilisation

```
main ()
{
fd_set set, setbis;
int idsock1, idsock2, maxsock;

        FD_ZERO(&set);
        maxsock=getdtablesize();
        FD_SET(idsock1, &set); /* ajout de idsock1 à l'ensemble set */
        FD_SET(idsock2, &set); /* ajout de idsock2 à l'ensemble set */

        bcopy ( (char*) &set, (char*) &setbis, sizeof(setbis));
        /* copie de l'ensemble set dans setbis */
        select (maxsock,&set, 0, 0, 0);
        if (FD_ISSET(idsock1, &set))        /* Test si idsock1 appartient à l'ensemble set */
                ...
if (FD_ISSET(idsock2, &set))
                ...
}
```


ANNEXES

PROCEDURES SECONDAIRES RELATIVES AUX SOCKETS

Vous n'aurez normalement pas besoin d'utiliser ces procédures, elles sont utilisées dans la procédure (addr_socket) d'instantiation de la structure sock_addr_in (voir fon.c).

- GESTION des ADRESSES IP

La procédure suivante permet de récupérer le nom (alias de l'adresse internet) de la machine locale. Elle peut être utiliser pour renseigner l'adresse locale dans la structure **sockaddr_in** avant un bind.

int **gethostname** (char *nom_hote, int longueur_nom)

```
Exemple:
char myname[MAXHOSTNAMELEN +1];
/* MAXHOSTNAMELEN est une constante predefinie*/
gethostname(myname, MAXHOSTNAMELEN);
```

La procédure suivante permet de récupérer la valeur numérique correspondant à l'adresse IP à partir de l'adresse sous forme de nom existant dans le fichier /etc/hosts.

struct hostent *__gethostbyname__ (char *nom_hote)

```
struct hostent {
    char *h_name;        /* nom du service */
    char **h_aliases;    /* liste des alias */
    int h_addrtype;      /* type de famille de protocole */
    int h_length;        /* longueur de l'adresse en octets */
    char * *h_addr_list; /* liste des adresses */
    }
```

0 <=> échec de la recherche

Dans le cas de la famille de protocoles Internet on aura :

h_addr_type = AF_INET (famille AF_INET)
h_length = 4 (adresse sur 4 octets)

@IP = h_addr_list [0]

Si l'adresse IP est donné sous forme "décimale pointée" (ex : "192.0.0.1") la procédure suivante permet de récupérer la valeur numérique :

unsigned long **inet_addr** (char *adr_ip)

0 <=> échec de la recherche

- GESTION des PROTOCOLES

La primitive suivante permet de déterminer la valeur numérique correspondant à un protocole donné :

struct protoent ***getprotobyname** (char *nom_hote)

```
struct protoent {
    char *p_name;       /* nom du protocole */
    char **p_aliases;   /* liste des alias */
    int p_proto;        /* protocole transport */
}
```

0 <=> échec de la recherche

- GESTION des NUMEROS de PORT

Les services standards tels que "mail", "ftp",... ont des numéros de ports réservés (voir le fichier /etc/services) qui peuvent être déterminés en utilisant la primitive suivante :

servent ***getservbyname** (char *service, char *protocole)


```
struct servent {
    char *s_name;       /* nom du service */
    char **s_aliases;   /* liste des alias */
    int s_port;         /* numéro de port */
    char *s_proto;      /* protocole utilisé */
    }
```

0 <=> échec de la recherche

- RECUPERATION DES ADRESSES DE LA SOCKET LOCALE

Cette procédure permet de récupérer les adresses locales de la socket donnée en paramètre en cas d'allocation dynamique.

int **getsockname** (int socket, struct sockaddr_in *name, int *namelen)

1 <=> échec de la recherche

ATTENTION : le paramètre **namelen** est un paramètre «donnée/résultat».

- RECUPERATION DES ADRESSES DE LA SOCKET DISTANTE

Cette procédure permet de récupérer les adresses socket du processus distant connecté à la socket donnée en paramètre.

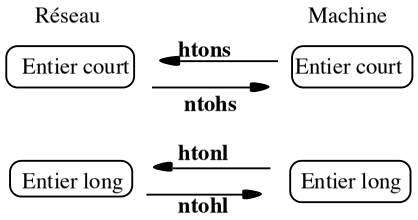
int **getpeername** (int socket, struct sockaddr_in *name, int *namelen)

1 <=> échec de la recherche

- REPRESENTATION DES ENTIERS (réseau / machine)

ATTENTION !! Les représentations des valeurs des entiers sur plusieurs octets diffèrent d'un ordinateur à l'autre (poids forts en premier ou en dernier). TCP/IP a choisi de passer les poids forts en dernier (comme XNS et SNA) et chaque constructeur livre, avec sa version de système d'exploitation, les procédures de transformation de format dont l'emploi garantit la portabilité des applications.

Il existe quatre procédures qui permettent de passer de la représentation machine à la représentation réseau pour les différents types entiers :



Par exemple, htons permet de passer de la machine (**h**ost) vers (**to**) le réseau (**n**etware) pour un entier court (**s**hort).

Ainsi par exemple, les deux octets correspondant au paramètre *s_port* de la procédure *getservbyname* sont dans l'ordre réseau et il faudra les remettre dans l'ordre machine avec la primitive *ntohs*.

RAPPELS CONCERNANT LES CREATIONS DE PROCESSUS ET LEURS SYNCHRONISATIONS DANS UNIX

Pour créer un processus à partir d'un autre processus, on utilise la primitive ***fork()*** d'UNIX. Le processus créé est appelé "processus fils", vis à vis du processus initial appelé "processus père".

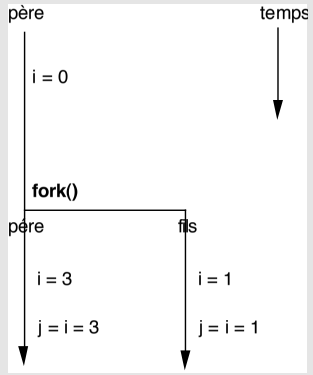
Le contexte de ce nouveau processus est créé par duplication du contexte du processus père, c'est à dire qu'il possède le même code exécutable et les mêmes variables (en particulier les descripteurs des sockets) dans une zone mémoire séparée. Son exécution démarre au point même où s'est réalisé le fork(). Le processus père se distingue du processus fils par son identifiant ou PID, dont la valeur est zéro (0) pour le processus fils.

Voici un exemple de code et son exécution dans le temps:

```
pere()
{
        int pid,int i,j;

        i=0;
        pid=fork();
        printf("i= %d",i);
        if (pid==0) /* processus fils */
                { i=1;printf("i= %d",i);}
        else    /* processus père */
                { i=3;printf("i= %d",i);}

        j=i;
        printf ("j= i = %d",j);
}
```



Les deux processus fonctionnent de façon indépendante. Si l'on souhaite à un moment donné, que le père attende la fin d'exécution du fils, il faudra le synchroniser par la primitive UNIX *wait()*.

Il faut bien avoir en tête qu'il se passe la même chose avec les descripteurs de sockets:

- le père crée un processus fils après "ACCEPT" : il possède à ce moment-là au moins 2 sockets: la socket passive et la socket de connexion au client

- le processus fils hérite donc des 2 sockets du père au moment du fork()

Il faut donc, après le fork()□

- dans le cas du fils□fermer la socket passive, parce qu'il ne s'en sert pas. Il conserve la socket de connexion au client, pour gérer les échanges de données avec celui-ci.

- dans le cas du père, fermer la socket de connexion au client, puisqu'il ne s'en sert pas.

Sans entrer dans les détails, signalons également que pour éviter les processus 'zombies', il faut que le processus serveur (père) ignore le signal SIGCHLD que lui fait le processus fils à sa mort□ce qui s'écrit en début de programme par *signal (SIGCHLD, SIG_IGN)*□

GENERATION D'EXECUTABLE

Tous les fichiers de prototypage nécessaires sont inclus dans le fichier **fon.h** , qui contient aussi le prototype de toutes les fonctions décrites ci-dessus. Il faudra donc inclure ce fichier en en-tête des programmes application client et serveur (#include "□on.h□). Le corps des fonctions se trouve dans le fichier **fon.c**. Fichier qu'il est donc nécessaire de compiler et lier avec chacune des applications (voir le makefile qui vous est fourni).

EXEMPLE DE MAKEFILE□

```
OBJ1 = fon.o  serveur.o
OBJ2 = fon.o  client.o
fon.o□      fon.h  fon.c
gcc -DDEBUG -c fon.c
serveur.o□fon.h serveur.c
gcc -c serveur.c
client.o□   fon.h client.c
gcc -c client.c
serveur□   {OBJ1}
gcc {OBJ1} -o serveur -lsocket -lnsl
client□     {OBJ2}
gcc {OBJ2} -o client -lsocket -lnsl
```


PROCEDURES DE GESTION DU SON SUR LES STATIONS SUN

Les stations que vous utilisez sont équipées d'une **carte son** et d'un **driver UNIX** pour les piloter. L'interface physique de sortie son est intégrée à la machine, c'est un haut-parleur. L'interface physique d'entrée étant externe, il s'agit d'un petit micro que l'on connecte à la prise mini-jack en face arrière de la station (attention, il est muni d'un petit **bouton ON/OFF**).

OUTILS AUDIO INTERACTIFS

Parmi les outils fournis en standard par le constructeur, on trouve un utilitaire interactif d'emploi du driver SON, **AudioTool**. Sous openwindows cliquez sur le bouton de droite de la souris et accédez, par l'option "□rogram□, à l'outil audio tool. La fenêtre comporte des boutons de type magnétophone (Ecoute, Enregistrement, Stop, Avance avant et arrière ....).

FICHIERS SON

En outre, il est possible de stocker sous forme binaire le son enregistré dans un fichier (extension .**au**). Fichier manipulé comme tout autre fichier UNIX par les primitives de lecture / écriture.

INTEGRATION A UNE APPLICATION C

Il est possible d'envoyer du son sur le haut-parleur ou d'enregistrer du son depuis le micro grâce au primitive read et write classique sur le fichier /dev/audio (pour plus d'information faire man audio). Attention plusieurs processus ne peuvent pas ouvrir en même temps et dans le même sens (lecture ou écriture) ce fichier.

Voici un petit exemple de programme d'enregistrement et d'écoute (voir le fichier essaison.c qui vous est fourni)□

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/fcntl.h>
#define TAILLE_BUF  80500
/* taille du buffer de son numérise correspond a 10 secondes */
/* 1 seconde de son est équivalent à 8 Koctets */
main()
{
    char buf [TAILLE_BUF];    /* Buffer de son numérisé */
    int taille_enr, taille_joue;
    int fsp;                              /* descripteur de fichier */
/* Ouverture en lecture de /dev/audio */
/* l'enregistrement commence des l'ouverture */
fsp= open ("/dev/audio", O_RDONLY);
/* Récupération dans buf du son enregistre */
taille_enr= read (fsp, buf, TAILLE_BUF);
printf ("%d octets ont ete enregistrés\n", taille_enr);
/* fermeture du fichier */
close (fsp);
/* Ouverture en écriture de /dev/audio */
/* l'enregistrement commence des l'ouverture */
fsp= open ("/dev/audio", O_WRONLY);
/* Envoie du contenu de buf sur le haut-parleur  */
taille_joue= write (fsp, buf, TAILLE_BUF);
printf ("%d octets ont ete auditionnes \n", taille_joue);
/* fermeture du fichier */
close(fsp);
}
```


RESUME DES PROCEDURES DE LA BOITE A OUTILS

Elles sont différenciées des primitives UNIX par un préfixe (**h_** pour highlevel). Notez bien pour les fonctions de lecture et écriture le **s** final qui indique clairement plusieurs appels de la primitive UNIX associée.

EXEMPLE:

h_socket qui encapsule un appel socket

h_writes qui encapsule plusieurs appels write

Une exception toutefois, la fonction adr_socket qui permet d'alléger la programmation.

ADR_SOCKET

void adr_socket (char *service, char *name, char *protocole,

struct sockaddr_in *p_adr_socket, int type)

Renseigne les différents champs de la structure d'adresses pointée par p_adr_socket, en fonction des paramètres service, name, protocole et du type d'application (constantes prédéfinies CLIENT ou SERVEUR).

```
char            *bip_bip = "2222"
int         num_socket;
struct      sockaddr_in adr_interne;
struct      sockaddr_in adr_distante;
adr_socket ( bip_bip, "frege ", "tcp" , &adr_interne, CLIENT);
adr_socket ( «1111», "boole ", "tcp" , &adr_distante, CLIENT );
```

H_SOCKET

int h_socket (int domaine, int mode)

Réserve un descripteur socket pour un mode de transmission et une famille de protocoles.

```
int num_socket;
num_socket = h_socket ( AF_INET, SOCK_STREAM );
```

H_BIND

void h_bind (int num_soc, struct sockaddr_in *p_adr_local)

Définit un point d'accès (@IP, numéro de port) local pour la socket.

```
int num_socket;
struct sockaddr_in adr_locale;
h_bind (num_socket, &adr_locale );
```

H_CONNECT

void h_connect (int num_soc, struct sockaddr_in *p_adr_distant)

Utilisée côté client, en mode connecté, ouvre une connexion entre la socket locale (num_sock) et la socket serveur distante (référencée par la structure sockaddr_in que pointe p_adr_distant).

EXEMPLE:

```
int num_socket;
struct sockaddr_in adr_serv;
h_connect (num_socket, &adr_serv );
```

H_LISTEN

void h_listen (int num_soc, int nb_req_att)

Utilisée côté serveur, en mode connecté, place le processus en attente de connexion d'un client. Définit la taille de la file d'attente, c'est-à-dire le nombre maximum de requêtes client qui peuvent être stockées en attente de traitement.

EXEMPLE:

```
int num_socket;
int nb_requetes = 6;
h_listen (num_socket, nb_requetes );
```


H_ACCEPT

int h_accept (int num_soc, struct sockaddr_in *p_adr_client)

Utilisée côté serveur, en mode connecté. Accepte, sur la socket (qui était en écoute) , la connexion d'un client et alloue à cette connexion une nouvelle socket qui supportera tous les échanges. Retourne l'identité du client (sockaddr_in que pointe p_adr_client) qui pourra être traitée par l'application, si aucun traitement n'est envisagé utiliser la constante prédéfinie TOUT_CLIENT .

```
int num_socket;
h_accept (num_socket, TOUT_CLIENT );
```

H_READS

int h_reads (int num_soc, char *tampon, int nb_octets)

Transfert les octets du buffer réception socket vers le tampon du processus applicatif jusqu'à détection d'une fin de transfert ou atteinte de la capacité du buffer. Utilisé en mode connecté, elle est bloquante tant que le buffer de réception socket est vide. Retourne le nombre de caractères effectivement reçus.

```
int taille, num_socket;
char *message;
message = ( char * ) malloc ( taille, sizeof (char) );
lg_message = h_reads (num_socket, message, taille );
```

H_WRITES

int h_writes (int num_soc, char *tampon, int nb_octets)

Utilisée en mode connecté. Bloquante tant que le buffer d'émission socket est plein. Transfert le nombre d'octets spécifié du tampon vers le buffer réception socket. Retourne le nombre de caractères effectivement émis.

```
int taille, lg_message, num_socket;
char *message;
message = ( char * ) malloc ( taille, sizeof (char) );
h_writes (num_socket, message, lg_message );
```

H_RECVFROM

int h_recvfrom (int num_soc, char *tampon, int nb_octets, struct sockaddr_in *p_adr_distant)

Utilisée en mode non connecté, bloquante tant que le buffer de réception socket est vide.

Transfert les octets provenant d'une socket distante (affecte la structure sockaddr_in que pointe p_adr_distant) du buffer réception socket vers le tampon.

Retourne le nombre de caractères effectivement reçus.

```
int taille, lg_message, num_socket;
char *message;
struct sockaddr_in adr_distante
message = ( char * ) malloc ( taille, sizeof (char) );
lg_message = h_recvfrom (num_socket, message, taille, &adr_distante );
```

H_SENDTO

int h_sendto (int num_soc, char *tampon, int nb_octets, struct sockaddr_in *p_adr_distant)

Utilisée en mode non connecté, elle est bloquante tant que le buffer d'émission socket est plein. Transfert le nombre d'octets spécifié du tampon vers le buffer réception socket à destination d'une socket distante spécifique (structure sockaddr_in que pointe p_adr_distant).

Retourne le nombre de caractères effectivement émis.


```
int taille, lg_message, num_socket;
char *message;
struct sockaddr_in adr_distante
message = ( char * ) malloc ( taille, sizeof (char) );
h_sendto (num_socket, message, lg_message, &adr_distante);
```

H_CLOSE

void h_close (int num_soc)

Libère le **descripteur socket** après avoir achevé les transactions en cours.

```
int num_socket;
h_close ( num_socket );
```

H_SHUTDOWN

void h_shutdown (int num_soc, int controle)

Met fin "violemment" à toute transaction sur une socket, y compris celles en attente, dans le ou les sens spécifiés.

```
#define FIN_RECEPTION 0
#define FIN_EMISSION 1
#define FIN_ECHANGES 2

    int num_socket ;
    h_shutdown ( num_socket, FIN_RECEPTION );
```