

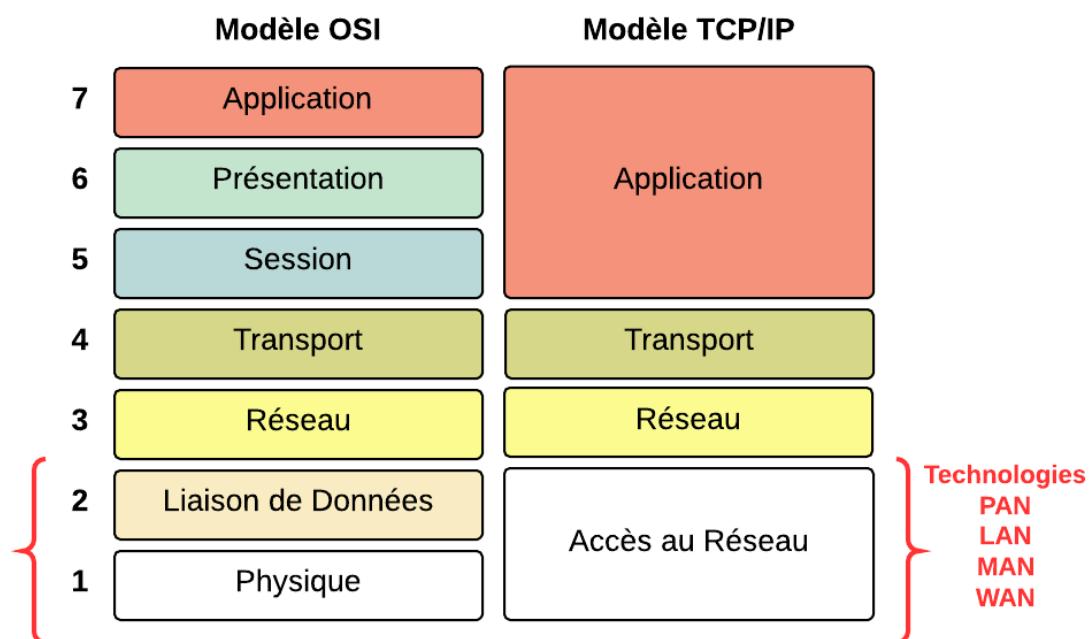
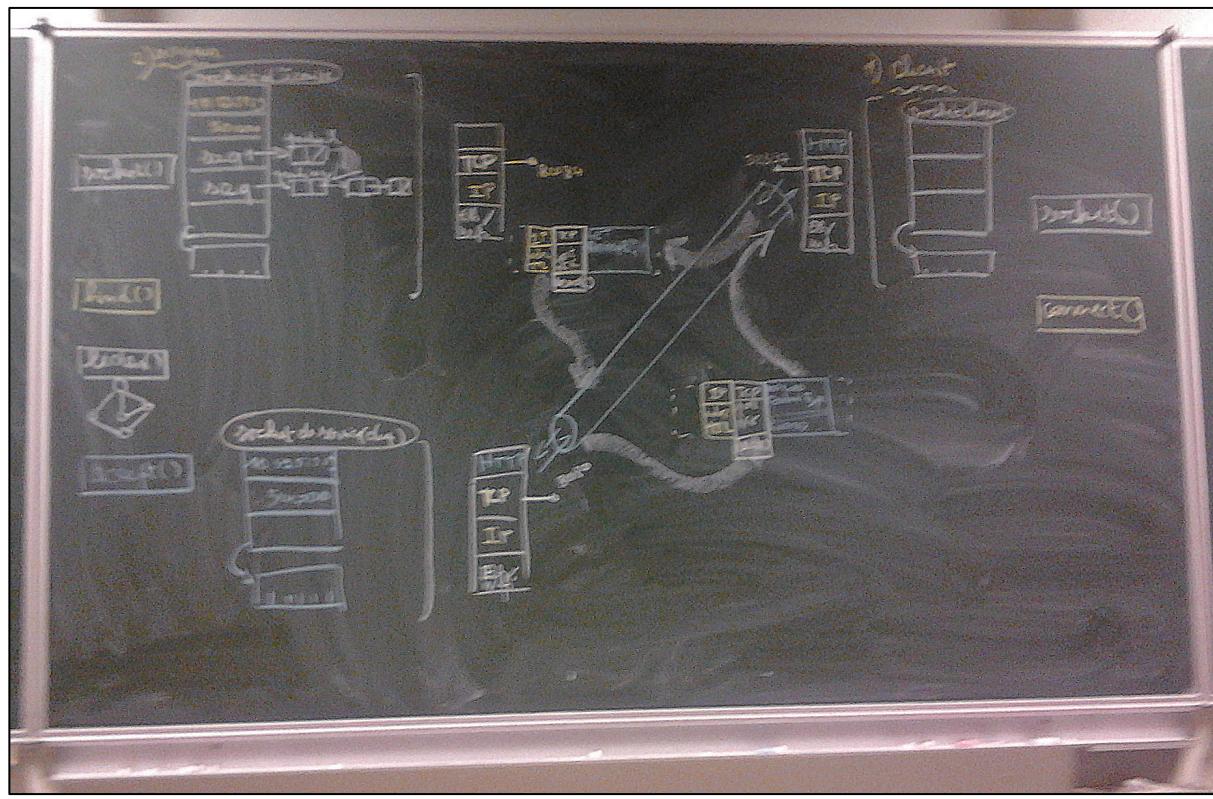
# **Programmation TCP-UDP/IP**

UE: Programmation réseaux, web et mobiles  
AA : Réseaux et technologie Internet

**Claude VILVENS**

claude.vilvens@hepl.be

<http://haute-ecole.provincedeliege.be/>



(© <https://juleshuynhvan.business.blog/2017/02/20/modele-osi-modele-tcpip/>)

## ***Sommaire***

### **Introduction**

## **I. Internet et la suite TCP/IP**

<b>1. L'idée d'Internet</b>	<b>3</b>
<b>2. L'Internet Society</b>	<b>4</b>
<b>3. Le World Wide Web</b>	<b>5</b>
<b>4. Les RFC</b>	<b>6</b>
<b>5. L'interconnexion des réseaux</b>	
5.1 Les éléments de connexion	6
5.2 La sécurité et les firewalls	7
5.3 Une hiérarchie de réseaux	7
<b>6. Le modèle TCP/IP</b>	<b>7</b>
<b>7. L'encapsulation des données selon TCP/IP</b>	<b>9</b>
<b>8. ARP et RARP : les protocoles de la couche d'interface IP</b>	<b>11</b>
<b>9. IP : le protocole de la couche réseau</b>	
9.1 La structure d'un datagramme IP	12
9.2 Le principe du routage IP	13
<b>10. Les ports des protocoles de transport</b>	<b>15</b>
<b>11. TCP : un protocole de la couche transport</b>	
11.1 Le principe d'une transmission TCP	16
11.2 La structure d'un segment TCP	16
11.3 L'automate à nombre d'états finis de TCP	18
<b>12. UDP : l'autre protocole de la couche transport</b>	<b>21</b>

## **II. Les sockets : un client-serveur TCP itératif**

<b>1. Un interface de communication</b>	<b>22</b>
<b>2. Représentation d'une socket en programmation C</b>	<b>23</b>
<b>3. La création d'une socket</b>	<b>24</b>
<b>4. La fermeture d'une socket</b>	<b>29</b>
<b>5. L'attachement d'une socket à une adresse (bind)</b>	<b>30</b>
<b>6. Un serveur monoconnexion</b>	<b>34</b>
<b>7. La mise à l'écoute sur une socket (listen)</b>	<b>35</b>
<b>8. La prise de connaissance d'une connexion (accept)</b>	<b>36</b>
<b>9. Deux sockets pour un serveur</b>	<b>39</b>
<b>10. Le client et sa connexion au serveur (connect)</b>	<b>41</b>
<b>11. L'échange de données entre le client et le serveur (send et recv)</b>	
11.1 L'émission de caractères (send)	44
11.2 La réception de caractères (recv)	45
11.3 Le serveur et le client	46
11.4 Un dialogue prolongé	53
<b>12. Si le client est interrompu</b>	<b>58</b>
<b>13. Un client Telnet</b>	<b>60</b>

<b>14. Un client Java</b>	
14.1 Faire redescendre Java au bas niveau	61
14.2 Un client Java pour le serveur C	65
<b>15. Le nombre de caractères lus</b>	68
<b>16. Le problème des chaînes de caractères de longueur variable</b>	80

### III. Un serveur TCP/IP multi-sockets

<b>1. Le principe de base</b>	88
<b>2. La structure fd_set</b>	
2.1 Un tableau masque	89
2.2 Le nombre de flags	90
2.3 Positionner un flag	91
<b>3. Le serveur multi-clients</b>	91

### IV. Un serveur TCP multi-threads POSIX

<b>1. L'idée d'un serveur concurrent</b>	101
<b>2. Un serveur multi-connexion et multi-thread à la demande</b>	110
<b>3. Un peu de programmation asynchrone</b>	121
<b>4. Une utilisation des mutex pour threads</b>	130
<b>5. Un client Java</b>	139
<b>6. Une utilisation des variables de condition : le pool de threads</b>	150

### V. L'adressage IP et ses fonctions d'utilisation

<b>1. Les adresses IP</b>	162
<b>2. L'adressage standard</b>	162
<b>3. L'adressage avec sous-réseau</b>	165
<b>4. La mise en œuvre d'une structure de sous-réseaux</b>	
4.1 La détermination du masque de sous-réseau	166
4.2 La détermination des subnetids	166
4.3 La détermination des hostids	167
<b>5. Les adresses locales et les adresses réseau</b>	167
<b>6. Les pseudo-connexions et les sockets paires</b>	
6.1 Adresse locale et adresse distante (getsockname et getpeername)	171
6.2 Pour le client	173
6.3 Pour le serveur	174
<b>7. Le DNS</b>	
7.1 Noms et adresses	175
7.2 Les domaines	175
7.3 Les serveurs de noms	176
7.4 Le resolver	176
<b>8. Les adresses dans les Intranets</b>	
8.1 Les machines publiques et privées	177
8.2 Les espaces d'adresses des intranets	178
8.3 L'intranet de l'In.Pr.E.S.	178

## VI. La programmation TCP/IP sous Windows

<b>1. En C : l'interface Winsock</b>	<b>181</b>
<b>2. En C++ : les classes sockets de MFC</b>	
2.1 La classe CAsyncSocket	186
2.2 La classe CSocket	189
<b>3. L'architecture .NET et TCP/IP</b>	<b>191</b>
<b>4. En C# : la classe Socket</b>	<b>191</b>
<b>5. Un serveur TCP synchrone et les opérations de base</b>	
5.1 L'attachement d'une socket à une adresse (bind)	193
5.2 La mise à l'écoute sur la socket (listen)	194
5.3 La prise en compte d'une connexion pendante (accept)	195
5.4 La fermeture d'une socket (close et shutdown)	195
5.5 L'envoi et la réception de données	195
5.6 La socket peer	196
5.7 Le code du serveur synchrone	197
<b>6. Un client TCP synchrone</b>	<b>201</b>
<b>7. Un client TCP asynchrone</b>	
7.1 Les méthodes asynchrones de .NET	204
7.2 La connexion asynchrone	205
7.3 La synchronisation : version avec événements	206
7.4 Les envois et réceptions asynchrones	208
7.5 La synchronisation : version simplifiée	209
7.6 Visualiser les threads sous-jacents	210
7.7 Le code du client asynchrone	210
<b>8. Un serveur TCP asynchrone</b>	
8.1 Un accept asynchrone	214
8.2 Le code du serveur asynchrone en mode console	214
8.3 Un exemple de conversation	217
<b>9. Un client TCP simplifié</b>	
9.1 La classe TCPClient	218
9.2 Un flux réseau	219
9.3 Le code du client TCP	219
<b>10. Un serveur TCP simplifié</b>	
10.1 La classe TCPLListener	222
10.2 Le code du serveur TCP	222
<b>11. Un client C# / .NET pour un serveur multithread C / UNIX</b>	
11.1 Les flux binaires de .NET	226
11.2 Le client du serveur multithread UNIX	227
11.3 Le code du client C#	228

## VII. Les sockets UDP

<b>1. Les caractéristiques d'UDP</b>	<b>234</b>
<b>2. La création d'une socket</b>	<b>235</b>
<b>3. L'échange de données entre le client et le serveur (sendto et recvfrom)</b>	
3.1 L'émission de caractères (sendto)	236

3.2 La réception de caractères (recvfrom)	237
<b>4. Une première communication UDP</b>	<b>239</b>
<b>5. Un dialogue client-serveur</b>	
5.1 Un dialogue requête-réponse	243
5.2 Un dialogue clients-serveur continu	248
<b>6. Les problèmes d'UDP</b>	<b>254</b>
<b>7. Le multicast</b>	
7.1 Les adresses multicast	258
7.2 Les options d'une socket multicast	259
<b>8. Le multicast en Java</b>	
8.1 La création d'un participant multicast	259
8.2 Un chat élémentaire	260
<b>9. UDP et le multicast en C# sous .NET</b>	
9.1 La classe UdpClient	266
9.2 La création d'un participant multicast	266
9.3 Le thread de réception des messages	267
9.4 Un chat élémentaire (bis)	268

## VIII. Le paramétrage des sockets

<b>1. Le mode non-bloquant</b>	
1.1 Le contrôle des descripteurs de socket (fcntl)	273
1.2 Le contrôle des périphériques flux (ioctl)	274
<b>2. Etre le propriétaire d'une socket</b>	<b>275</b>
<b>3. Obtenir des informations sur une socket (getsockopt)</b>	<b>276</b>
<b>4. Modifier les options d'une socket (setsockopt)</b>	<b>278</b>
<b>5. Les options booléennes</b>	
5.1 Le broadcast	280
5.2 La réutilisation d'une adresse ou d'un port	280
5.3 Les connexions perdues	281
5.4 Une option au niveau protocole	281
5.5 Les caractères urgents	282
<b>6. Les options non booléennes</b>	
6.1 Le type de socket	282
6.2 La longueur maximale d'un segment TCP	282
6.3 La taille des buffers	282
6.4 Les time-out	283
<b>7. L'utilisation des options pour le multicast</b>	<b>287</b>

## IX. Les caractères urgents

<b>1. Le principe du caractère urgent</b>	<b>292</b>
<b>2. La réception hors buffer</b>	<b>293</b>
<b>3. La réception dans le buffer</b>	<b>293</b>
<b>4. Le signal de caractère urgent</b>	<b>294</b>
<b>5. Un exemple élémentaire en non OOBINLINE</b>	<b>294</b>

## X. Quelques commandes réseaux utiles

<b>1. Une commande de diagnostic : ping</b>	
1.1 Le protocole ICMP	301
1.2 Quelques exemples d'exécution de la commande ping	302
1.3 Les options de ping	305
<b>2. Une commande de statistiques des sockets : netstat</b>	
2.1 La visualisation des sockets	307
2.2 Les options de netstat	308
<b>3. Une commande de lecture de configuration : i(plf)config</b>	316
<b>4. Une commande d'interrogation des DNS : nslookup</b>	318

## XI. Les fichiers de configuration TCP/IP

<b>1. Les fonctions de consultation machines-adresses</b>	
1.1 La recherche par adresse	322
1.2 Le nom de la machine locale	323
<b>2. Le fichier /etc/hosts</b>	
2.1 Sous UNIX	326
2.2 Sous Windows	327
<b>3. Le fichier /etc/networks</b>	
3.1 Sous UNIX	329
3.2 Sous Windows	329
<b>4. Les fonctions de consultation réseaux-adresses</b>	330
<b>5. Le fichier /etc/protocols</b>	
5.1 Sous UNIX	333
5.2 Sous Windows	333
<b>6. Les fonctions de consultation des protocoles</b>	334
<b>7. Le fichier /etc/services</b>	
7.1 Sous UNIX	337
7.2 Sous Windows	338
<b>8. Les fonctions de consultation des services</b>	340

## XII. Le protocole IPv6

<b>1. Un meilleur IP</b>	345
<b>2. La structure d'un datagramme IPv6</b>	345
<b>3. La notation des adresses d'IPv6</b>	347
<b>4. Différents types d'adresses</b>	348
<b>5. Installation du stack IPv6</b>	349
<b>6. La programmation des sockets IPv6 en C/C++</b>	
6.1 Une nouvelle famille et de nouvelles structures d'adresse	350
6.2 Attacher une socket à un interface et un service	352
6.3 La prise en compte d'une connexion	353
6.4 Pour identifier le client connecté	353
6.5 Le serveur IPv6 de base	354
6.6 Le client IPv6 de base	360

6.7 Le cas de plusieurs interfaces	365
<b>7. Le protocole ICMPv6</b>	<b>366</b>
<b>8. L'utilisation des sockets IPv6 en Java</b>	<b>367</b>

## **Annexe : La programmation TCP/IP en C/Windows**

<b>1. L'utilisation des sockets</b>	<b>369</b>
<b>2. L'utilisation des threads</b>	<b>371</b>

## **Ouvrages consultés**

## Introduction



La programmation des réseaux a toujours eu un côté fascinant mais aussi effrayant : tout y semble si complexe ... Cependant, au fil du temps, le contexte de développement a considérablement évolué.

Il y a une vingtaine d'années, il y était essentiellement question de lignes de transmissions, de signaux et de modems, bref de communications à longue distance. Le domaine était manifestement celui d'ingénieurs de télécommunications, pas tellement celui des informaticiens qui, d'ailleurs, avaient d'autres bits à fouetter (à coups d'assembleur ou, plus raffiné, de FORTRAN ou, avec plus de littérature, en utilisant COBOL).

Le succès extraordinaire des ordinateurs personnels (autrement dit, des "PC"s) a considérablement modifié la donne vers la moitié des années '80. En effet, le terrain s'est révélé extrêmement fécond pour permettre de faire communiquer ces petites unités et introduire, au niveau d'un simple bureau, la notion de ressource partagée : le réseau local s'imposait. Ethernet et Token Ring sont les éléments les plus connus de cette révolution. La programmation, dans ce monde des PCs, utilisait NetBIOS. Le langage de prédilection pour de tels réseaux était bien sur le langage C tandis que le modèle de référence qui s'imposait était le modèle OSI à 7 couches.

Depuis déjà de nombreuses années, et c'est un lieu commun que de le rappeler, Internet a bouleversé les certitudes : son, ou plutôt ses, protocoles TCP, UDP et IP se sont imposés, non seulement dans le domaine des communications mondiales, mais même dans celui des réseaux locaux. Si bien qu'à l'heure actuelle, c'est une programmation utilisant cette famille de protocoles, et le modèle à 4 couches qui l'accompagne, qui s'impose comme une bagage indispensable de la formation d'un informaticien de bon niveau.

Le présent ouvrage se propose donc de donner les bases de la programmation classique d'applications basées sur TCP-UDP/IP. Ceci se fera dans un contexte

- ◆ hétérogène : machines UNIX, Linux et Windows;
- ◆ multi-langage : programmation C et C++ classique, avec références à la programmation Java (voir notes de cours Java II) et programmation Windows C#/.NET;
- ◆ multi-clients : threads pour tout le monde.

Le cheminement proposé ici semblera peut-être peu orthodoxe aux puristes. En effet, plutôt que de présenter les différentes matières dans des chapitres aux cloisons étanches, on a choisi ici de se plonger le plus rapidement possible dans la programmation des communications TCP. Certains points, provisoirement laissés dans l'ombre, sont évoqués plus en profondeur dans les chapitres ultérieurs. Dans une telle optique, les travaux de laboratoire peuvent s'effectuer en parallèle du cours théorique en étant efficaces dès le départ.

Une autre remarque s'impose. On constatera que le traitement des erreurs, s'il est bien sûr présent, n'est pas poussé à l'extrême. La raison en est simplement un soucis de relative clarté : on ne souhaitait pas obscurcir les concepts par des routines d'erreurs complexes. Mais, que l'on en soit bien conscient, une programmation professionnelle ne peut se concevoir sans une gestion d'erreurs irréprochable.

On peut le constater en regardant la carte, cet exposé n'est donc accessible qu'à ceux qui maîtrisent les systèmes d'exploitations et les langages cités ci-dessus. Cruel prérequis exigé. Mais quelle somptueuse récompense que de pouvoir faire communiquer des ordinateurs et des applications si différents par ce standard de fait qu'est TCP/IP !

**Claude Vilvens**

## I. Internet et la suite TCP/IP

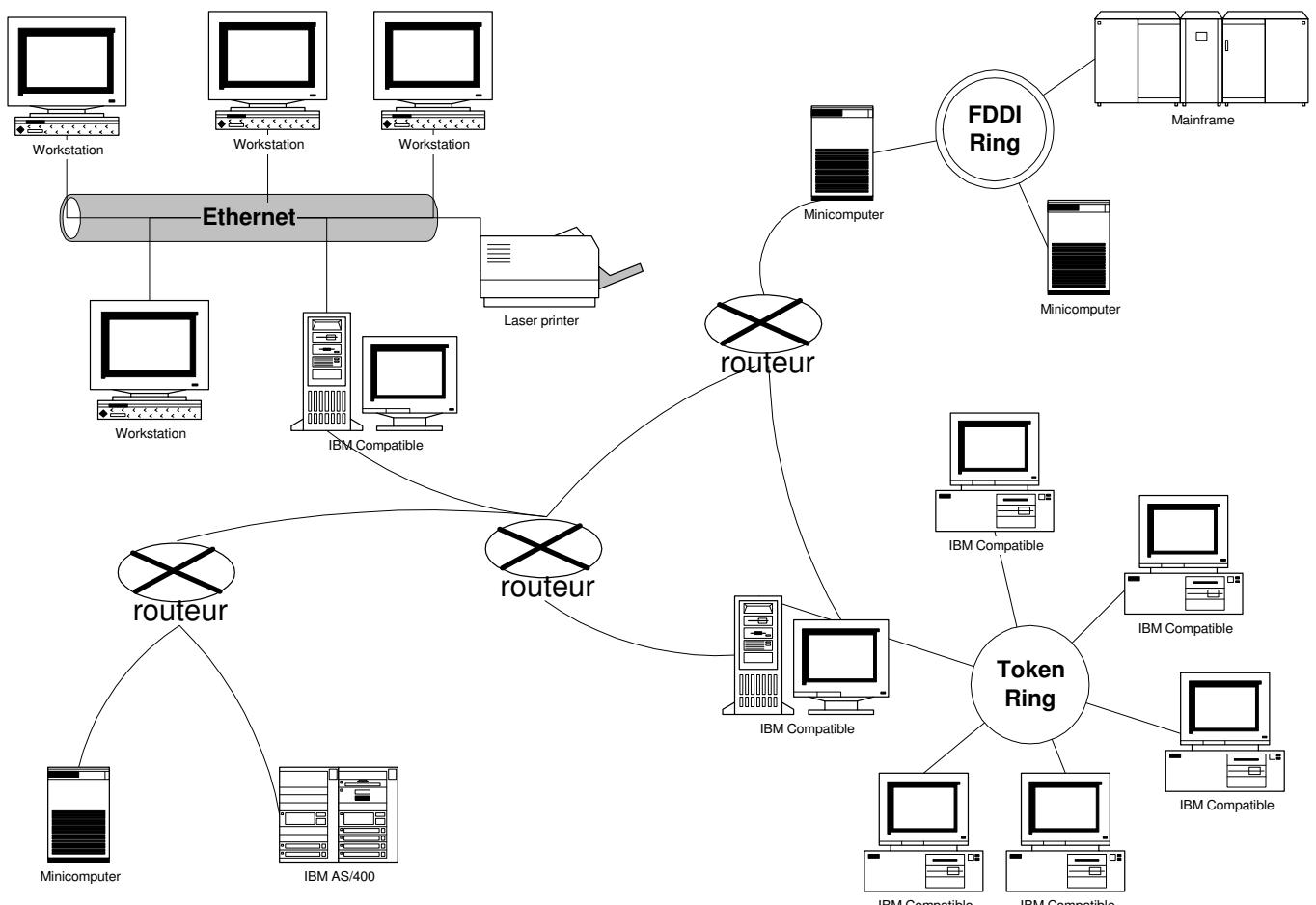


*Imbéciles : Ceux qui ne pensent pas comme nous.*

(G. Flaubert, Dictionnaire des idées reçues).

### 1. L'idée d'Internet

On admet généralement que c'est à la DARPA (Defense Advanced Research Projects Agency) que l'on doit l'initiative des recherches visant à l'interconnexion de tous les réseaux quelle que soit leur technologie. Ce réseau apparu en 1962 fut appelé **Arpanet**. On parle encore, pour désigner un tel "méga-réseau", de *réseau logique*. Il devait contribuer, dans la suite et avec d'autres expériences, à un réseau dont l'ambition est de connecter tous les réseaux existants, y compris ceux qui ne sont pas encore inventés : le "réseau des réseaux", c'est-à-dire **Internet**.



Indéniablement, le monde UNIX a apporté à ces recherches une contribution importante. Ainsi, c'est l'UNIX BSD qui a apporté l'interface classique de la communication réseau, à savoir les **sockets**. Mais il faut bien remarquer que c'est l'ensemble du monde de l'informatique qui a été impliqué dans cette mise au point d'un "graal" de la communication.

Ces recherches ont finalement abouti à la définition d'un ensemble de protocoles connus sous le vocable général de "**suite TCP/IP**", d'après le nom des deux protocoles **Transmission Control Protocol / Internet Protocol** (respectivement de niveau transport et de niveau réseau) les plus connus et les plus utilisés. Bien sûr, les réseaux interconnectés n'utilisent pas forcément les mêmes mécanismes d'adressage. Il a donc fallu attribuer une "*adresse logique*" à chaque machine du réseau. Cette adresse est la célèbre "**adresse IP**", codée<sup>1</sup> sur 32 bits. Elle comporte une composante adresse réseau et une composante adresse machine dans ce réseau. En pratique, on la désigne sous la forme **n<sub>1</sub>.n<sub>2</sub>.n<sub>3</sub>.n<sub>4</sub>**. (par exemple, 10.10.0.100). Nous y reviendrons de manière plus détaillée dans le chapitre plus spécifiquement consacré à ces adresses IP.

## 2. L'Internet Society

Une question vient immédiatement à l'esprit : qui définit les adresses des différentes machines et réseaux ?

A priori, Internet appartient à tout le monde et à personne : aucune entreprise ou organisation n'en est le propriétaire. Il n'en reste pas moins qu'il existe des organismes dont l'objectif est d'encadrer son évolution et de définir, puis de maintenir, des standards.

On désigne par "Internet Society", ou **ISOC**<sup>2</sup>, une organisation internationale qui fut créée en 1992 dans le but de promouvoir Internet et d'en assurer la gestion. Sa mission officielle est :

"D'assurer l'essor, l'évolution et l'utilisation de l'Internet pour le bienfait de toutes et tous à travers le monde."

Pour ce faire, elle définit son action selon les axes suivants :

1. faciliter le développement accessible de normes et de protocoles, l'administration et les structures techniques de l'Internet;
2. soutenir la formation dans les pays en développement et où le besoin existe;
3. soutenir le développement professionnel et encourager les occasions de contacts avec les chefs de file de l'Internet;
4. fournir des informations fiables sur l'Internet;
5. organiser des forums de discussions sur des questions touchant à l'évolution, au développement et à l'utilisation de l'Internet (sur les plans technique, commercial , social ...);
6. développer un environnement favorable à la coopération internationale, à la communauté et une culture qui rend possible l'autogestion;
7. servir de point focal pour les efforts communs de promotion de l'Internet en tant qu'outil fiable pour tous les peuples du monde;
8. donner la direction et permettre la coordination des efforts sur les plans humanitaire et de l'éducation ainsi qu'au niveau social etc.

L'organisation est dirigée par un Conseil de Gestion (**Board of Trustees**) dont les membres sont élus par les membres de l'ISOC. Elle se flatte de regrouper 150 organisations et 6000 membres individuels provenant de plus de 100 pays. Citons par exemple AT&T, IBM,

---

<sup>1</sup> du moins actuellement

<sup>2</sup> [www.isoc.org](http://www.isoc.org)

Microsoft, AOL, Cisco, Intel, Adobe, ... Les membres sont classés en fonction de leur contribution financière annuelle.

L'**IAB** (Internet Activities/Architecture Board) est un comité créé en 1983 dans le but de coordonner les recherches et travaux concernant Internet. Les communications se font par RFC (Request For Comments – nous en reparlerons plus loin). L'IAB a mis sur pieds diverses composantes permettant de gérer les évolutions des protocoles de communication TCP/IP :

- ◆ l'**ICANN** (Internet Corporation for Assigned Names and Numbers) qui gère tous les numéros et codes devant être uniques au niveau d'Internet, donc notamment les numéros de ports, de versions, de protocoles, de MIB, etc; ses membres sont essentiellement des organismes; quant aux adresses IP (v4 et v6), elles sont déléguées à des organismes particuliers : les **RIR** (Regional Internet Registries) qui distribuent les adresses dans les différentes régions du monde; on en compte 4 : le RIPE-NCC (Réseaux IP Européens – Network Coordination Center) qui a reçu la délégation de ce rôle pour l'Europe, tout comme l'APNIC (Asia Pacific Network Information Centre) pour la région Asie-Pacifique, l'ARIN (American Registry for Internet Numbers) et le LACNIC (Latin America and Caribbean Network Coordination Center) pour les Amériques; l'ICANN remplace depuis 2000 l'**IANA** (Internet Assigned Number Authority), qui n'était contrôlée que par le gouvernement américain;
- ◆ l'**IETF** (Internet Engineering Task Force) visant à gérer Internet à court terme; elle élabore les spécifications et les premières implantations des protocoles de la suite TCP/IP; on trouve en son sein des concepteurs de réseaux, des opérateurs, des vendeurs et des chercheurs de toute origine; c'est en fait le pilier d'Internet; son organe de direction, l'**IESG** (Internet Engineering Steering Group), est responsable de l'approbation finale des standards;
- ◆ l'**IRTF** (Internet Research Task Force) qui se consacre à la recherche à long terme.

### 3. Le World Wide Web

C'est au CERN (Conseil Européen pour la Recherche Nucléaire), en 1989, qu'un certain **Tim Berners-Lee** proposa un projet de gestion d'informations basé sur un interface graphique portable utilisant les hyperliens : autrement dit, il venait d'inventer le concept de browser-éditeur. Ce n'est que l'année suivante que ce projet fut accepté du bout des lèvres avec des crédits limités. Mais cela permit de développer un prototype de browser dont les premières présentations publiques eurent lieu en 1992.

Le succès fut foudroyant, d'autant que le CERN mettait le logiciel à la disposition libre de tous au moyen de l'Internet. En un an, le nombre de serveurs Web était passé de 26 à 200. On connaît la suite ...

Tim Berners-Lee quitta le CERN en 1994 pour fonder le **W3C**<sup>3</sup> (World Wide Web Consortium), basé au MIT (Massachusetts Institute of Technology). Le rôle de cet organisme est de gérer le Web et d'en définir les spécifications et recommandations pour ce qui concerne sa technologie, son interface utilisateur, son architecture, son accessibilité, ses rapports avec la société. Ses membres paient une cotisation fixe (un montant pour les sociétés, un montant beaucoup plus faible pour les ASBL). Citons, par exemple encore une fois, AT&T, IBM, Apple, Microsoft, AOL, Cisco, Intel, Adobe, Compaq, ...

On peut remarquer que la manière d'utiliser Internet pour gérer de l'information par le Web s'est confondue, dans l'usage courant, avec Internet lui-même : "Web" et "Internet" sont devenus des synonymes dans l'esprit de beaucoup de gens.

---

<sup>3</sup> [www.w3.org](http://www.w3.org)

## 4. Les RFC

Les spécifications pures et dures des protocoles de TCP/IP sont publiées sous forme de **RFC** (Requests For Comments) numérotées. Les RFC sont une série de documents qui définissent les standards de TCP/IP et, plus généralement, de tout ce qui touche l'applicatif basé sur TCP/IP, donc finalement Internet lui-même. Donc, **ils décrivent les fonctionnement internes d'Internet et les standard de TCP/IP**. Ces RFC sont accessibles via Internet<sup>4</sup>.

Un membre de l'ISOC peut soumettre un document pour qu'il soit publié sous forme de RFC. Ce document est d'abord **PS** (Proposed Standard - standard proposé). Si l'implémentation proposée est testée avec succès pendant 4 mois au moins sur deux sites différents, la proposition devient un **DS** (Draft Standard - standard brouillon). L'IAB décide finalement si elle peut devenir un **IS** (Internet Standard), c'est-à-dire une norme Internet. Les RFC sont classées en cinq niveaux (obligatoire, recommandé, facultatif, limité, non recommandé).

Lorsqu'un document est publié, un numéro de RFC lui est attribué. En cas d'évolution une nouvelle RFC sera publié sous un autre numéro.

L'IAB publie une note chaque trimestre qui permet de connaître la RFC en cours pour chaque protocole.

## 5. L'interconnexion des réseaux

### 5.1 Les éléments de connexion

Une fois les adresses attribuées, le problème est encore de faire communiquer des réseaux ou des machines de nature fort différente. Une "boîte noire" est placée à la jonction de deux réseaux afin de prendre en compte les conversions nécessaires. Cette "boîte noire" est une unité fonctionnelle d'interconnexion de réseau. Selon les cas, on distingue :

- ◆ le **répéteur** = il copie les bits en transit entre deux segments de câble, en les amplifiant ou en les regénérant. Il travaille au niveau de la couche **physique**. On l'utilise par exemple lorsque l'on dépasse la longueur tolérée des câbles de liaisons.
- ◆ le **pont** = dispositif utilisé pour relier deux réseaux. C'est une sorte de répéteur intelligent qui sait analyser les trames qu'il reçoit pour déterminer l'adresse de chacune des deux stations des deux côtés du pont. Il peut ainsi faire suivre vers le destinataire. Les ponts travaillent sur la sous-couche MAC de la couche **liaison**.
- ◆ le **routeur** = sorte de pont super-intelligent qui connaît les adresses de tous les ordinateurs du réseau ainsi que les autres ponts et routeurs. De plus, il est capable de réaliser les conversions entre les divers protocoles. Il peut donc choisir le meilleur chemin pour envoyer un message. Les routeurs opèrent au niveau de la couche **réseau**.
- ◆ la **passerelle** (gateway) = dispositif reliant des réseaux de types différents, en particulier des réseaux locaux à des mainframes ou minis. Le terme de "passerelle" est en fait un terme générique, qui peut être utilisé au niveau de différentes couches. Ainsi, un routeur peut être considéré comme une passerelle de niveau réseau. Habituellement, cependant, une passerelle est supposée agir au niveau **transport** et, surtout, au niveau **application**.

Dans le cadre de la programmation TCP/IP, seul l'existence de routeurs nous importera. Le terme de passerelle lui servira, dans ce contexte seulement, de synonyme.

---

<sup>4</sup> [www.enst.fr/~dax/services/rfc](http://www.enst.fr/~dax/services/rfc) ou [www.freesoft.org/CIE/RFC](http://www.freesoft.org/CIE/RFC)

---

## 5.2 La sécurité et les firewalls

Dès que l'on communique se pose le problème des visites non désirées, pour ne pas dire les effractions. Un **firewall** (garde-barrière) est le point de passage obligé pour les flux entrants et sortants entre un réseau d'entreprise interne et l'extérieure. Il est l'équivalent électronique du pont-levis (et de ses gardes) d'un château médiéval entouré de douves. Il comporte :

- ◆ un routeur filtreur des paquets sortants;
- ◆ un routeur filtreur des paquets entrants;
- ◆ une passerelle d'application entre les deux.

Les filtres sont gérés par des tables configurées par l'administrateur du système. Elles précisent quelles adresses peuvent être utilisées pour sortir ou entrer (mais c'est assez difficile, car pas mal d'application utilisent des adresses dynamiques). La passerelle permet en plus d'analyser les flux et de prendre une décision selon le contenu.

## 5.3 Une hiérarchie de réseaux

On peut finalement distinguer :

- ◆ les réseaux fédérateurs = **backbones** (épines dorsales) : communications à très haut débit, routeurs très rapides;
- ◆ les réseaux régionaux = souvent de grands réseaux d'opérateurs nationaux de télécommunications ou des réseaux spécialisés;
- ◆ les réseaux locaux = le plus souvent des LAN.

Ces réseaux communiquent sur Internet grâce à IP. La couche transport (typiquement, TCP) reçoit un flux de données d'un processus applicatif et le tronçonne en morceaux appelés des segments TCP qui seront ensuite décomposés, au niveau de la couche réseau, en **datagrammes**<sup>5</sup> IP (en pratique, de l'ordre de 1500 octets). Chaque datagramme est ensuite transmis au réseau Internet. Le datagramme peut lui-même être décomposé en fragments IP plus petits. Lorsque toutes les composantes parviennent à destination, la couche réseau reconstitue le datagramme original. La couche transport reconstitue ensuite le flux original à partir des différents datagrammes.

# 6. Le modèle TCP/IP

L'architecture correspondante est à **4 couches** – on parle encore du "*stack TCP/IP*"<sup>6</sup>. Par rapport à l'architecture OSI classique à 7 couches, c'est ce qui saute aux yeux immédiatement. On peut considérer que les deux niveaux les plus "physiques" ont été regroupés, de même que les 3 niveaux les plus proches des applications. Un autre façon de définir la couche la plus haute, que l'on trouve dans bon nombre de littératures, est d'affirmer que les couches OSI session et présentation n'existent pas. Cela donne schématiquement :

---

<sup>5</sup> on parle encore aussi de "**paquets IP**"; on utilise ce terme de "paquet" plus particulièrement lorsque l'on ne s'intéresse qu'aux données véhiculées et pas au protocole de livraison de ces données.

<sup>6</sup> les Français parlent d'"empilement" TCP/IP.

<b>Application</b> = application + session + présentation + données utilisateurs	Telnet (Terminal Network - login à distance), Rlogin (pour UNIX)	FTP (File Transfer Protocol – transfert de fichiers)	HTTP (Hyper-Text Transfer Protocol – gestion des pages WEB)	SMTP (Simple Mail Transfer Protocol – courrier à distance)	NNTP (Net News Transfer Protocol – gestion des articles des groupes de discussion = news)	DNS (Domain Name System - gestion du nom des machines)	BOOTP (Boot-strap Protocol - pour les machines sans disques qui doivent charger le logiciel de base à partir d'un serveur)	FTP (Trivial File Transfer Protocol - transfert de fichiers simplifié)	RPC (Remote Procedure Call)	
<b>Transport</b> [segment/ datagramme]					TCP (Transmission Control Protocol) : <b>fiable et orienté connexion</b> (comme une communication téléphonique) Il établit un circuit virtuel et logique entre deux machines : gestion des <b>segments</b> , confirmation (ACK), rémission, timeouts [RFC 793]	UDP (User Datagram Protocol) : <b>non fiable et orienté sans connexion</b> (comme une lettre) Il envoie les <b>datagrammes</b> d'une machine à l'autre, sans garantie de réception [RFC 768]				
<b>Réseau ou Internet</b> [datagramme/ paquet]					IP (Internet Protocol) : transport des <b>datagrammes</b> sans garantie [RFC 791] ICMP (Internet Control Message Protocol) : échange de messages d'erreurs et d'informations [ping] [RFC 792] IGMP (Internet Group Management Protocol) ; envoi de <b>datagrammes</b> UDP vers plusieurs hôtes	ARP (Address Resolution Protocol) / RARP (Reverse Address Resolution Protocol) : conversion adresses IP/ adresses de certains interfaces réseaux (comme Ethernet). On trouve à ce niveau les drivers et les cartes (par ex : driver Ethernet, Token Ring) [frame]				

Pour faire court, on peut citer comme différences principales de ce modèle par rapport au modèle OSI les points suivants :

- ◆ on ne trouve que 4 couches; OSI en a 7 (sans doute parce qu'à l'époque, l'architecture SNA d'IBM en avait 7 aussi) – mais, en fait, les couches session et présentation sont fort peu utilisées ou meublées;
- ◆ la distinction entre service, protocole et interface est moins claire;
- ◆ pour OSI, on a d'abord inventé le modèle, puis on a mis au point les protocoles nécessaires; pour TCP/IP, c'est l'inverse : on a d'abord utilisé les protocoles et le modèle ne fait que refléter ces derniers;
- ◆ le niveau réseau est sans connexion, mais le niveau transport offre les deux possibilités; dans OSI, c'est l'inverse : la couche réseau permet les deux types de communication tandis que la couche transport n'admet que la communication orientée connexion.

## **7. L'encapsulation des données selon TCP/IP**

Selon ce modèle à 4 couches, des données utilisateurs (*data user*) véhiculées par TCP/IP pour fournir à un réseau local une trame Ethernet (qui peut contenir de 46 à 1500 octets) seront encapsulées de la manière suivante :

header Ethernet 14 b	header IP 20 b	header TCP 20 b	header application	<b>data user</b>	trailer Ethernet 4
----------------------------	-------------------	--------------------	-----------------------	------------------	--------------------------

			<b>data user</b>	
			<b>data application</b>	
			<b>segment TCP</b>	
	<b>datagramme IP</b>			
<b>trame Ethernet</b>				

On peut déjà savoir d'emblée que :

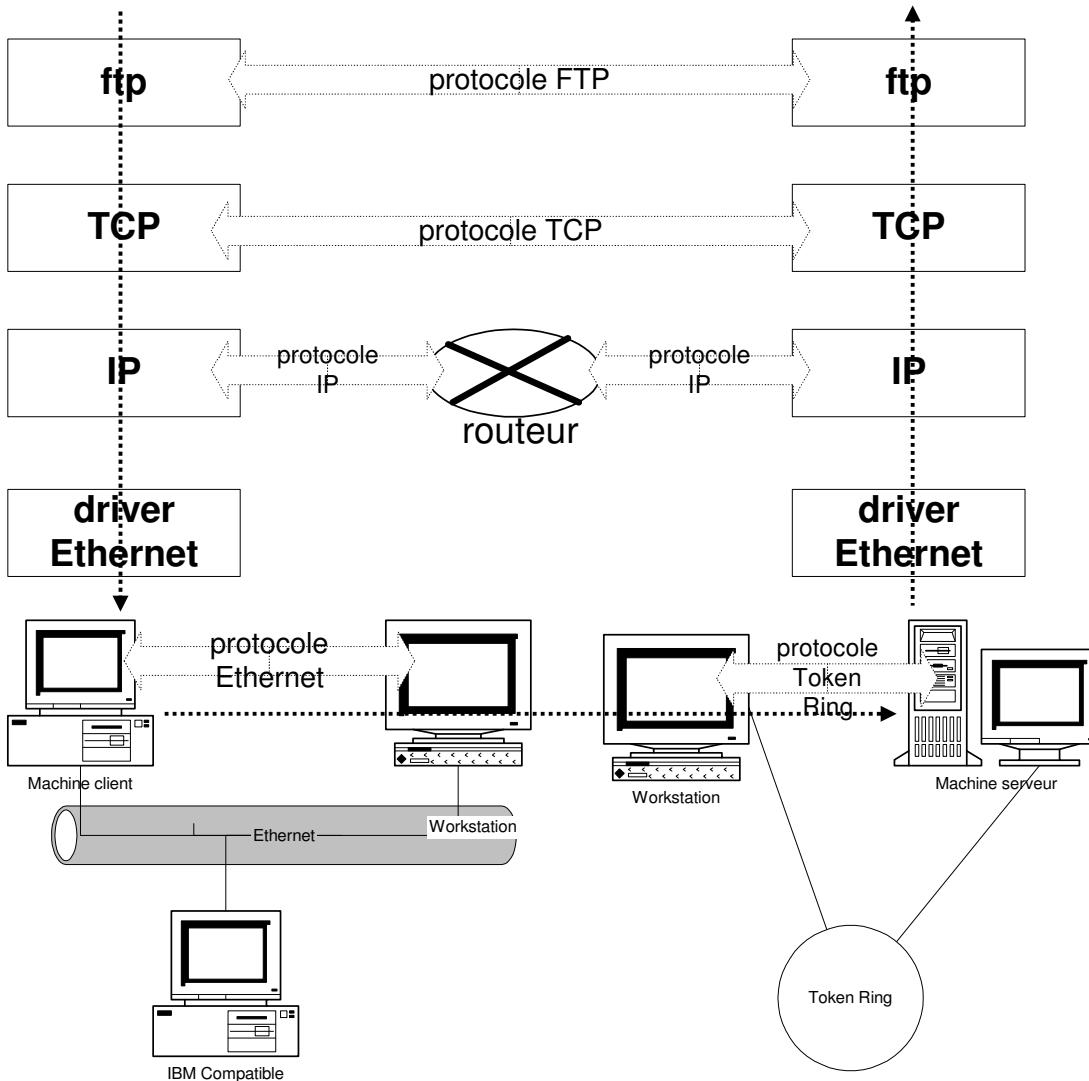
- 1) le header IP contient une valeur de 8 bits appelée le **champ du protocole** et indiquant la couche supérieure d'origine (ou de destination) des données. Sous UNIX, ce champ vaut 6 pour TCP, 17 pour UDP, etc. On trouve ces valeurs, sur une machine UNIX, dans /etc/protocol.
- 2) le header TCP code sur des nombres de 16 bits un numéro de port qui identifie l'application émettrice et réceptrice.

Pour mieux visualiser les choses, considérons l'exemple classique d'un réseau Ethernet connecté à un réseau Token ring par l'intermédiaire d'un routeur "parlant" TCP/IP. Si nous désignons par :

- ◆ ES : un "End System", donc un ordinateur host;
- ◆ IS : un "Intermediate System", donc en pratique un routeur;

on peut constater que, pour la couche application et transport (TCP), le protocole est de type ES-ES tandis que pour la couche réseau (IP), le protocole est de type ES-IS.

---



Il faut encore remarquer que chaque interface réseau possède une taille limite pour la trame lors de l'encapsulation des données à émettre; cette taille est donc fixée par la couche liaison de données. On désigne cette limite par **MTU** (**M**aximum **T**ransmission **U**nit). Elle varie selon les réseaux :

- ◆ Ethernet :1500;
- ◆ 802.3 : 1492;
- ◆ FDDI : 4352;
- ◆ X.25 : 576;
- ◆ PPP : 296.

Par conséquent, IP devra **fragmenter** un datagramme de taille supérieure à ce MTU en morceaux de taille inférieure à ce MTU diminué de la taille des en-têtes.

Voyons à présent les protocoles Internet n'appartenant pas à la couche application de manière plus précise. Les protocole d'applications, comme Telnet, HTTP, FTP ou SNMP ne seront pas appréhendés ici plus profondément.

## 8. ARP et RARP : les protocoles de la couche d'interface IP

Les protocoles de cette couche de bas niveau sont responsables

- ◆ de la **fragmentation** des messages à émettre pour les envoyer selon une **trame** physique (par exemple une trame Ethernet) et du travail inverse;
- ◆ de la **conversion** des adresses IP en adresses de l'interface réseau (comme Ethernet) et inversément.

*L'interfaçage d'IP avec la couche liaison, et notamment les réseaux Ethernet, est réalisée par deux protocoles :*

### ARP (Address Resolution Protocol)

Ce protocole transforme donc les adresses de la couche réseau en adresses de la couche liaison. Supposons ainsi qu'une machine  $M_{Orig}$  veuille communiquer, sur un réseau Ethernet, avec une machine  $M_{Dest}$  dont elle ne connaît que l'adresse IP  $adrIPDest$ . Le résultat sera alors l'envoi sur le réseau d'une trame Ethernet de diffusion, message de type particulier, comportant les trois adresses connues à ce moment :  $adrIPOrig$ ,  $adrIPDest$  et  $adrETHOrig$  (cette dernière est l'adresse Ethernet de  $M_{Orig}$ ). La machine  $M_{Dest}$  va reconnaître son adresse IP et répondra à  $M_{Orig}$  pour lui envoyer son adresse Ethernet  $adrETHDest$ .

### RARP (Reverse Address Resolution Protocol)

Ce protocole transforme donc les adresses de la couche liaison en adresses de la couche réseau. Il s'agit ici de permettre à une machine  $M_{Orig}$  de demander son adresse IP à un serveur dédié. La machine envoie pour cela un message Ethernet de diffusion. Le serveur est le seul à répondre en fournissant l'adresse IP demandée.

Signalons l'existence d'un protocole d'interfaçage d'IP avec les lignes séries asynchrones utilisant le protocole RS232 : **SLIP** (Serial Line IP). Les datagrammes IP sont encapsulés pour être véhiculés sur une ligne série RS-232 au moyen d'un modem haute vitesse. Il n'y a pas de détection ou de correction d'erreurs. Le procédé est peu intéressant pour les petits paquets IP, parce que trop d'octets sont nécessaires à chaque transmission. **CSLIP** (Compressed SLIP) est un SLIP compressé.

## 9. IP : le protocole de la couche réseau

**IP (Internet Protocol )** gère

- ◆ l'échange d'unités de transfert appelés des datagrammes, en **mode sans connexion et sans garantie** d'arrivée à bon port (ce sera le rôle de la couche supérieure d'assurer éventuellement cette fonctionnalité avec le protocole TCP);
- ◆ la **fragmentation** des datagrammes en unités plus petites si nécessaire, avec reconstitution sur le site final.

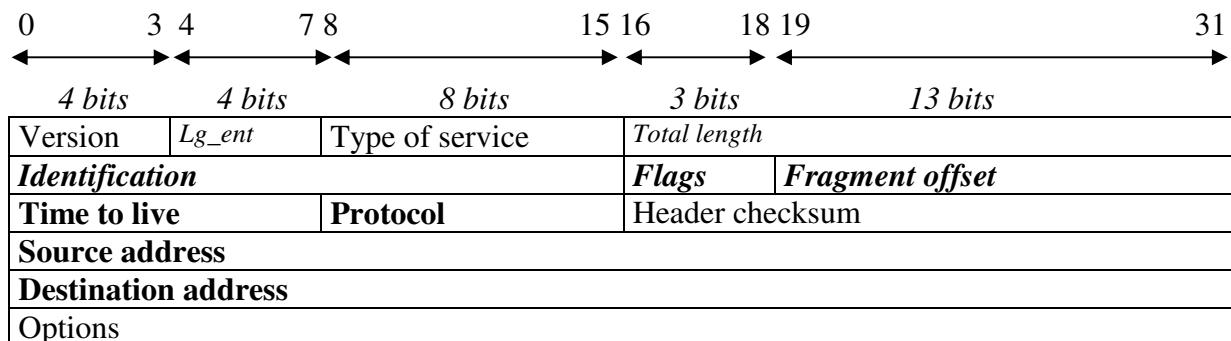
Donc, si IP reçoit un datagramme à envoyer, le protocole compare sa longueur au MTU du réseau à partir duquel il sera envoyé. En cas de nécessité, le datagramme sera fragmenté en morceaux donnant chacun naissance à un paquet différent, qui suivra son chemin propre. Les fragments peuvent très bien parvenir en désordre au destinataire puisque

---

IP les remettra en ordre pour reconstituer le datagramme initial. Il suffit qu'un seul fragment ne parvienne pas pour que tout le datagramme soit écrasé par IP.

## **9.1 La structure d'un datagramme IP**

Un datagramme IP comporte un en-tête suivi des données proprement dites (la "charge utile"). Cet en-tête est composé lui-même d'une partie fixe (20 octets ou 5 mots de 32 bits) et d'une partie optionnelle (de longueur variable, mais toujours telle que l'en-tête soit d'une longueur multiple de 32 bits). La taille maximale d'un datagramme est de 65535 bytes (sous la norme IPv4).



On peut constater que l'en-tête IP est assez complexe. On peut cependant remarquer immédiatement :

- ♦ les adresses de la source et de la destination du datagramme : **Source address** et **Destination address**;
  - ♦ le numéro conventionnel du protocole de transport à qui confier le datagramme (TCP, UDP, ...) : **Protocol**;

On se souviendra que le datagramme peut être *fragmenté* (tous les ordinateurs doivent admettre des fragments de 576 octets au moins). Le problème de la reconstitution explique l'existence des champs

- ◆ **Identification** : qui permet de savoir à quel datagramme appartient le fragment reçu (donc, on trouvera la même valeur pour tous les fragments d'un même datagramme, ce qui permettra de le reconstituer) ;
  - ◆ **Fragment offset** : ce champ fournit l'offset du fragment dans le datagramme courant; sa taille étant de 13 bits, on peut donc compter 8192 fragments par datagramme - tous les fragments (sauf le dernier) doivent avoir des longueurs multiples de 8 octets;
  - ◆ **Flags** : indicateurs relatifs à la fragmentation :  $DF = \text{Don't fragment}$ ,  $MF = \text{More Fragment}$  (fragment non unique) - tous les fragments d'un datagramme ont leur flag MF activé sauf le dernier; évidemment, si un fragment a son flag DF actif, mais que le routeur auquel il est parvenu constate que la fragmentation est pourtant nécessaire à cause du MTU local, il générera un message d'erreur ICMP.

Il ne faut cependant pas perdre de vue que le niveau réseau fait intervenir également la notion de  **routage**. En effet, dès qu'une machine doit communiquer avec une autre n'appartenant pas au même réseau, il faudra faire transiter les messages par des éléments de connexions inter-réseaux qui sont typiquement ici des  ***routeurs***. Comme IP travaille en mode non connecté, on pourrait donc courir le risque de voir certains datagrammes errer indéfiniment sur le réseau (datagrammes parasites), par exemple parce que les tables de

routage sont altérées. Ce n'est heureusement pas le cas à cause de l'existence du champ **Time to live** (TTL), qui est un compteur limitant la durée de vie du datagramme – il est décrémenté

- ◆ de 1 à chaque passage dans un routeur;
- ◆ de plusieurs unités lorsqu'il reste longtemps dans une file d'attente d'un routeur.

Sa valeur initiale est souvent de 32 ou 64 et de toute façon de 255 au maximum. S'il tombe à 0, il y aura destruction et avertissement de l'ordinateurs source (erreur ICMP).

Les tailles étant variables, on comprend aisément la raison d'être des champs :

- ◆ **Lg\_ent** qui mémorise la longueur de l'en-tête en mots de 32 bits (de 5 à 15);
- ◆ **Total length**, qui contient la longueur en octets de l'ensemble du datagramme (donc, 65635 octets au maximum).

Pour être complet, précisons encore que le champ **Type of service** permet de définir

- ◆ sur 3 bits : la priorité (0=normal, 7=supervision réseau);
- ◆ sur 3 bits comment le datagramme doit être géré, au moyen d'indicateurs *D* (Delay), *T* (Throughput) et *R* (Reliability); un 4<sup>ème</sup> bit peut servir à l'indicateur *M* (Monetary cost).

Enfin, le champ **Options** a été imaginé afin de pouvoir prendre en compte des extensions futures par rapport à la version d'origine : sécurité, horodatage, routeurs obligatoires, ... Les options ne sont pas reconnues par tous les routeurs !

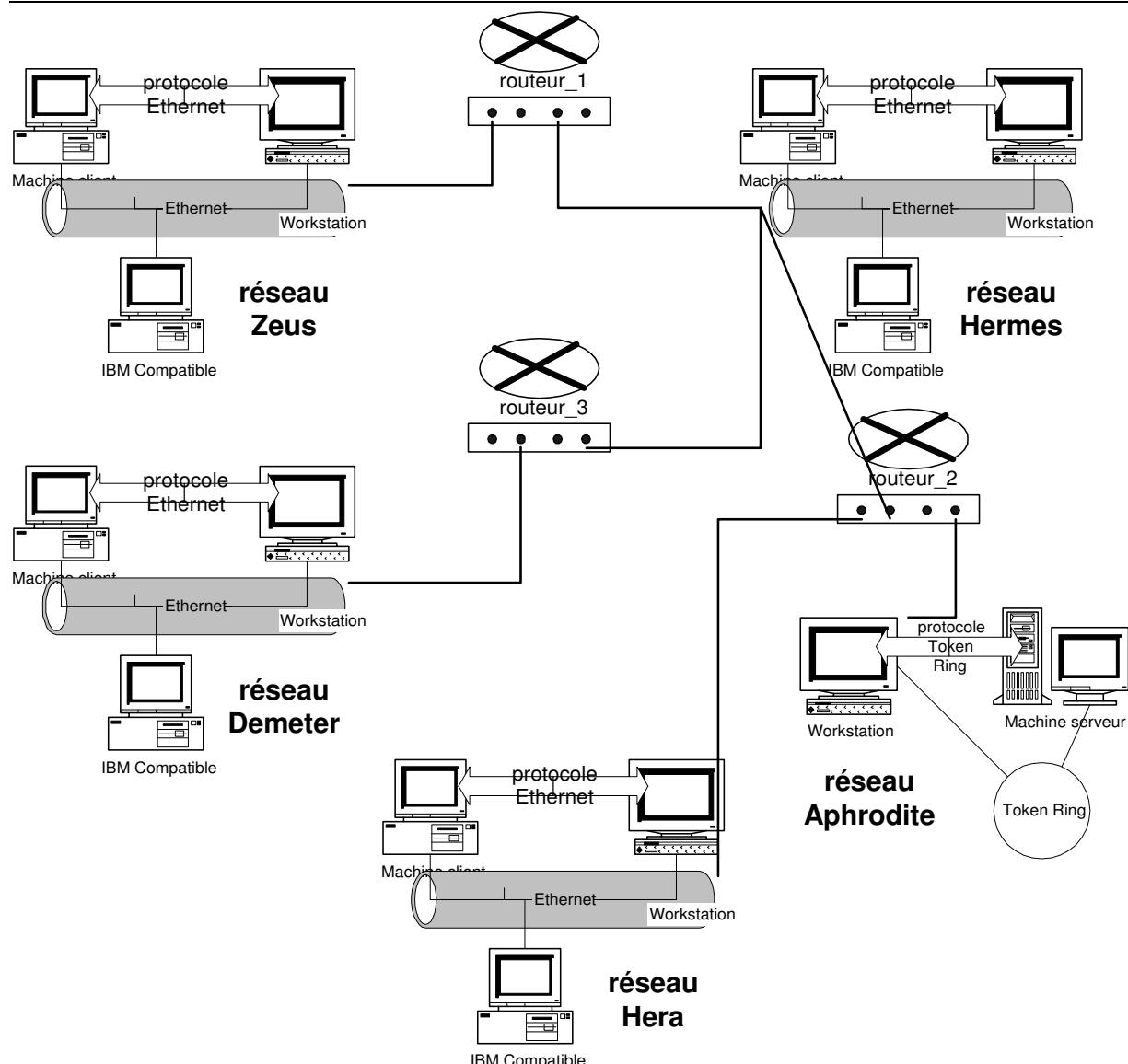
## 9.2 Le principe du routage IP

Il appartient donc au protocole IP de déterminer le chemin permettant d'acheminer un datagramme d'un point à un autre. Deux cas peuvent se présenter :

- ◆ les machines origines et destinations appartiennent au même réseau physique : il n'y a évidemment aucun problème et il suffit d'encapsuler le datagramme dans une trame physique (Ethernet par exemple);
- ◆ les machines origines et destinations appartiennent à des réseaux différents : il faudra dans ce cas trouver des "passerelles" permettant la liaison entre ces réseaux – c'est cette recherche qui est appelée "routage". Bien sûr, une "passerelle" possède plusieurs adresses IP : une pour chaque interface correspondant à un réseau distinct.

C'est donc le rôle du routeur de déterminer le chemin lorsque celui-ci ne fait pas partie intégrante du réseau. Pour ce faire, il dispose, comme toute machine d'un réseau quelconque, d'une **table de routage** contenant, pour chaque réseau interconnecté, l'interface que la machine ou le routeur utilise et l'adresse de la passerelle à utiliser. De plus, une table de routage contient toujours une route par défaut utilisée par les datagrammes pour lesquels il n'existe pas de spécification de passerelle.

Pour illustrer ces tables, considérons l'exemple suivant :



Une machine du réseau Hermès aura comme table de routage (on suppose qu'elle dispose d'un seul interface réseau) :

<i>destination</i>	<i>passerelle</i>
Hermès	la machine elle-même (même réseau)
Zeus	routeur_1
Aphrodite	routeur_2
Hera	routeur_2
Demeter	routeur_3
par défaut	routeur_3

Le routeur\_2 aura pour table de routage :

<i>destination</i>	<i>interface du routeur passerelle</i>	<i>passerelle</i>
Hermès	2	le routeur lui-même
Aphrodite	4	le routeur lui-même
Hera	1	le routeur lui-même
Zeus	1	routeur_1
Demeter	2	routeur_3
par défaut	2	routeur_3

L'*algorithme de routage* de base, non optimisé, est alors simple :

- ◆ si le réseau visé par le datagramme est accessible directement, on peut immédiatement confectionner la trame physique;
- ◆ si le réseau visé par le datagramme n'est pas accessible directement, on envoie le datagramme à la passerelle correspondant au réseau ou, s'il n'en existe pas, vers la passerelle par défaut.

Une erreur de routage sera émise en cas d'échec. Il importe de bien remarquer que toutes les étapes intermédiaires ne se passent que dans les deux couches les plus basses : les couches transport et application n'interviennent qu'aux deux extrémités.

Et puisque nous parlons de la couche transport ...

### **Remarques**

- 1) Les tables de routage sont normalement configurées manuellement (dans un but évident d'optimisation). Cependant, on peut utiliser le protocole **RIP** (**Routing Information Protocol**) pour permettre la mise à jour dynamique des tables : les machines voisines s'échangent des messages dans ce but.
- 2) La gestion des erreurs telles qu'une machine ou un port inaccessible, un réseau hors d'atteinte, la non-délivrance d'un datagramme, est assurée par le protocole **ICMP** (**Internet Control Message Protocol**). Ses informations sont encapsulées dans un datagramme IP, avec le champ Protocol à la valeur correspondant à ICMP.

## **10. Les ports des protocoles de transport**

Il s'agit ici de faire dialoguer deux processus qui correspondent chacun à un point d'entrée dans leur système respectif. Cependant, un processus peut assurer plusieurs services et chaque service peut être assuré par plusieurs processus sous des protocoles différents. Comme la désignation d'un processus varie d'un système d'exploitation à l'autre, on identifie le service demandé (le point d'entrée transport) de manière indépendante en utilisant le couple

**<application> : <numéro de port>**

Le port est, en quelque sorte, à TCP ce que l'adresse est à IP.

Certains ports sont réservés par l'IANA à des applications Internet standard : on appelle encore ces ports des ***well-known ports***; leur numéros se trouvent dans la plage [0-1023]. Ainsi, Telnet utilise par défaut le port 23, FTP le 21, HTTP le 80, SMTP le 25, le service date-heure (quand il existe) le 13.

Par opposition, on nomme encore "port éphémère" (*ephemeral port*) un port attribué à un client le temps de la communication avec le serveur visé. Ces ports se trouvent dans l'intervalle [49152-65535].

Les ports se trouvant dans la fourchette [1024-49151] sont *registered ports* : ils ne sont pas contrôlés par l'IANA, mais enregistrés par cet organisme à des fins de publication.

Sous TCP, un point de communication sera constitué de l'adresse IP de l'ordinateur considéré et du numéro de port du service visé. Ce point de communication constitue ce que l'on appelle une socket.

On peut encore remarquer que, s'il importe de connaître l'adresse et le port d'un serveur, il est indifférent pour un client de connaître le port qui lui a été attribué par le serveur : seul celui-ci doit le connaître.

## 11. TCP : un protocole de la couche transport

### 11.1 Le principe d'une transmission TCP

Pour rappel, TCP est un protocole orienté flux (c'est-à-dire transportant des flots de caractères) **orienté connexion** et **fiable** (comme une communication téléphonique). Comme la couche réseau IP n'assure pas cette fiabilité, c'est à TCP d'opérer les contrôles nécessaires en utilisant des accusés de réception (ACK pour acknowledgement). Sa démarche est la suivante :

1. les données de l'application sont morcelées en segments;
2. TCP envoie un segment avec un time-out et attend l'ACK de la cible – si aucun ACK n'est reçu après le time-out, il y a retransmission;
3. si TCP reçoit des données, il envoie un ACK à la source après un délai assez court.

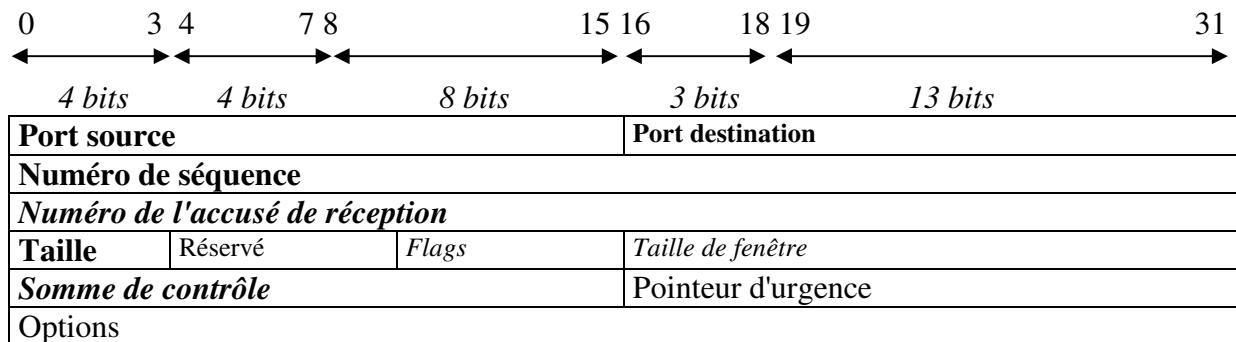
De plus

- ◆ TCP gère un checksum – si un segment arrive avec un checksum erroné, TCP le détruit et n'envoie pas d'ACK à la source;
- ◆ selon IP, les segments TCP peuvent arriver dans un ordre différent de celui d'émission; TCP est capable de les remettre en ordre;
- ◆ un datagramme IP qui serait dupliqué est supprimé;
- ◆ TCP contrôle le flux : il peut le régulariser entre deux ordinateurs de vitesse différente.

Il faut remarquer qu'un routeur peut très bien découper en plusieurs morceaux un segment trop grand. Chaque nouveau segment sera véhiculé indépendamment (donc avec son propre en-tête IP et TCP).

### 11.2 La structure d'un segment TCP

Un segment TCP, qui constituera la partie données d'un datagramme IP, comporte un en-tête suivi des données proprement dites (la "charge utile"). Cet en-tête est composé lui-même d'une partie fixe (20 octets ou 5 mots de 32 bits) et d'une partie optionnelle.



On peut constater que l'en-tête TCP est, lui aussi, assez complexe. On peut cependant remarquer immédiatement les ports de la source et de la destination du segment : **Port source** et **Port destination**.

Tout aussi naturellement, on trouve un **numéro de séquence** : l'objectif est donc de numérotier les séquences de caractères dans le but d'assurer leur mise en ordre correcte - en fait, chaque octet est numéroté et le numéro pris en compte est celui du premier byte du segment. En début de connexion, un numéro initial (*Initial Sequence Number*), choisi par l'ordinateur (en usant d'une horloge), est envoyé.

Enfin, au chapitre des évidences, on trouve également la **taille** réelle de l'en-tête, qui dépend bien sûr des options facultatives.

Ceci étant dit, il ne faut pas perdre de vue que le service doit être *fiable* : la notion d'accusé de réception (acquittement ou **ACK**), avec réémission en cas d'absence de celui-ci, doit donc être présente au sein d'un segment TCP. De fait, le champ **Numéro de l'accusé de réception** permet à un segment de véhiculer un ACK (si du moins le flag ACK de la collection des flags est à 1) : ce champ contient alors le prochain numéro de séquence que l'émetteur s'attend à recevoir (il s'agit donc du numéro du dernier byte reçu avec succès par le destinataire, augmenté de la longueur des données + 1). Dans le sens de l'émission, il s'agit du numéro du prochain segment attendu. L'émetteur enclenche alors un timer; si celui-ci arrive à expiration sans réception de l'ACK correspondant, le segment est réémis.

Toujours dans le contexte de la fiabilité, on remarquera également la **somme de contrôle** (checksum) qui se calcule sur l'en-tête et les données. L'émetteur la calcule et le récepteur la vérifie.

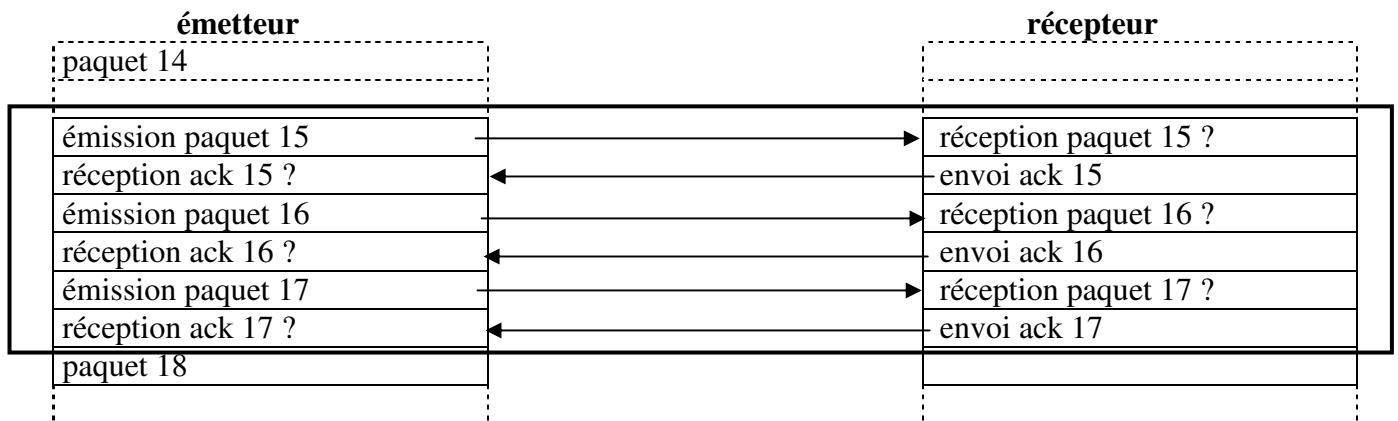
Plus techniques sont les autres champs. Ainsi, tout d'abord, les 6 *flags* disponibles sont :

<i>flag</i>	<i>signification</i>
TH_URG	Il s'agit d'un caractère urgent – voir le champ pointeur d'urgence qui est valide dans ce cas
TH_ACK	Il s'agit d'un ACK – le champ numéro d'accusé de réception est significatif
TH_PUSH	Le segment doit être transmis le plus vite possible à l'application, donc sans attendre le regroupement éventuel avec d'autres données ( <i>exemple typique</i> : un utilisateur connecté sur une machine distante; il appuie sur la touche Return; il est normal que la donnée soit envoyée immédiatement sans bufférisation).
TH_RST	Réinitialisation de la connexion – correspond à la notification d'une erreur
TH_SYN	Synchronisation des numéros de séquence pour initialiser une connexion
TH_FIN	Fin de la communication

Ensuite, TCP utilise des fenêtres d'anticipation ("fenêtres glissantes" ou *sliding windows*), qui doivent leur nom au fait que des segments sont en fait envoyés alors que l'on n'a pas encore reçu l'ACK de certains segments précédents. On peut distinguer :

- ◆ la fenêtre d'émission : elle comporte les numéros des séquences déjà transmises mais non encore acquittées. Evidemment, sa taille est le nombre de buffers d'émission disponibles, puisqu'il faudra éventuellement réémettre ces segments.
- ◆ la fenêtre de réception : elle mémorise les numéros de segments reçus et non encore acquittés (ce qui peut arriver : il suffit que le numéro inférieur n'ait pas encore été reçu avec succès). La taille de cette fenêtre est donc en fait le nombre de segments que le récepteur est capable de recevoir.

Ainsi, schématiquement, si l'émetteur et le récepteur utilisent une fenêtre ayant une taille de 3 paquets, le cycle de transmissions et d'accusés de réception peut être illustré comme suit :



Enfin, il est possible de transmettre des *caractères urgents*, c'est-à-dire de vilains tricheurs qui ne feront pas la file mais seront pris en compte selon un mécanisme d'interruption (sous UNIX, avec le signal SIGURG). Le champ *pointeur d'urgence* permet alors de repérer le dernier caractère urgent transmis. Nous reparlerons de ces caractères urgents ("*out-of-band*") plus loin.

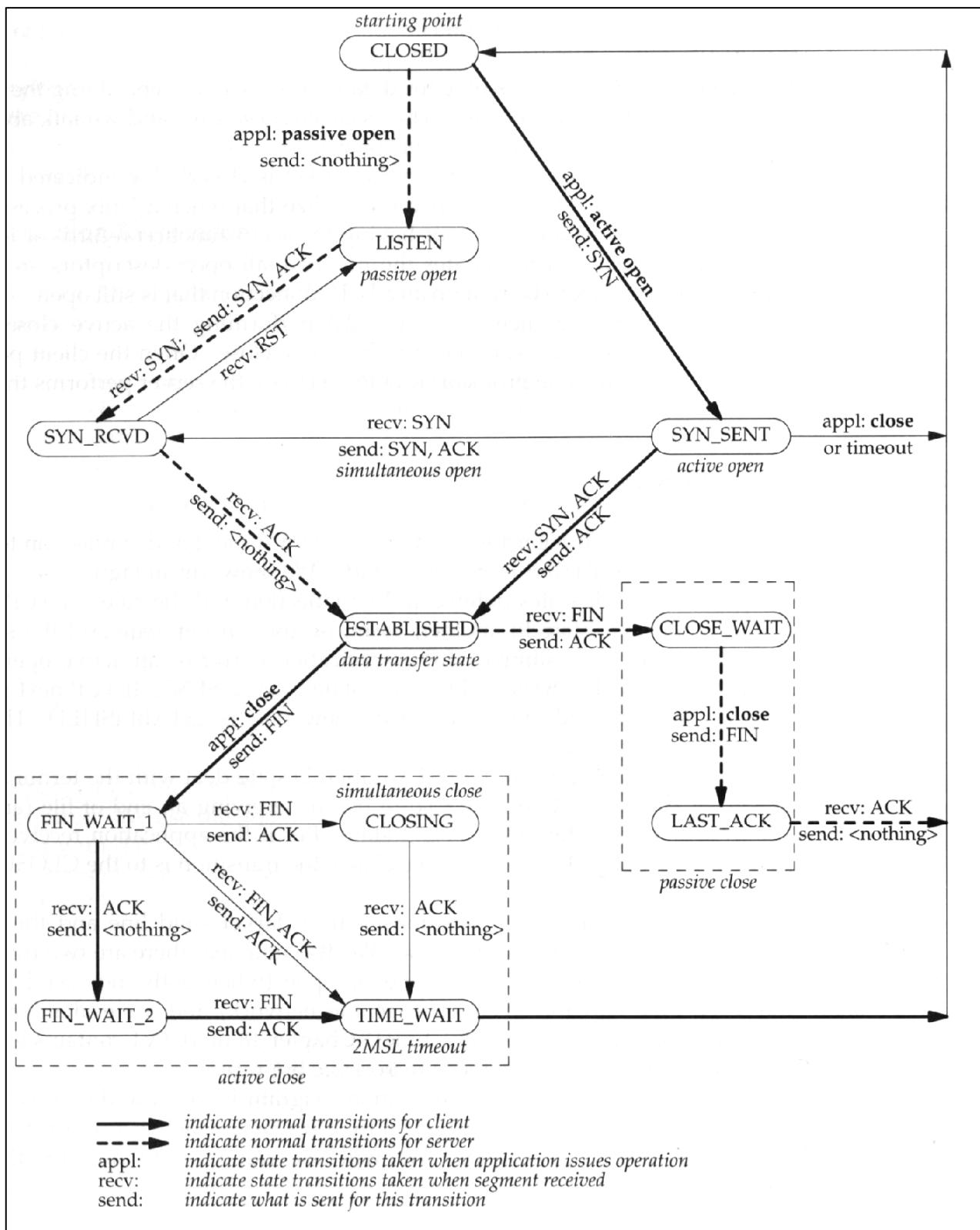
### 11.3 L'automate à nombre d'états finis de TCP

TCP est un protocole à états, c'est-à-dire qu'une connexion utilisant ce protocole se trouve toujours dans un état donné ou passe d'un état à un autre. On peut donc très bien résumer le fonctionnement de TCP dans un automate à nombre d'états finis. Ceux-ci sont (d'après le livre d'Andrew Tanenbaum) :

état	state	signification
FERMEE	CLOSED	Connexion fermée, inactive
ECOUTE	LISTEN	Le serveur attend une demande de connexion
SYN_RECUE	SYN_RCVD	Une demande de connexion est arrivée
SYN_EMIS	SYN_SENT	L'application a commencé à ouvrir une connexion
ETABLIE	ESTABLISHED	La connexion est dans son état normal pour transférer des données
FIN_ATTENTE_1	FIN_WAIT_1	L'application indique qu'elle a terminé
FIN_ATTENTE_2	FIN_WAIT_2	Le serveur indique qu'il accepte cette terminaison
TEMPORISATION	TIME_WAIT	Attente que tous les paquets aient disparu

FERMETURE_EN_COURS	CLOSING	Le client et le serveur ont essayé de fermer simultanément
ATTENTE_FERMETURE	CLOSE_WAIT	Le serveur reçoit une indication de fermeture
DERNIER_ACK	LAST_ACK	Attente que tous les paquets aient disparu

et l'automate est (d'après le livre de W. Richard Stevens) :



Comment comprendre cette horreur (;-)) ? En fait, le diagramme semble compliqué parce qu'il fait intervenir sur la connexion étudiée et le serveur (que l'on contacte) et le client (qui prend l'initiative du contact). On peut distinguer :

1) L'**établissement d'une connexion**, qui passe par les étapes suivantes :

- ◆ le serveur se met à l'écoute (instruction listen -> état ECOUTE) : on parle encore d'*ouverture passive* de la connexion;
- ◆ le client se connecte (instruction connect) : on parle encore d'*ouverture active* de la connexion; il envoie alors un premier segment TCP de type SYN, ce qui permet de communiquer au serveur le premier numéro de séquence qui sera utilisée pour les données émises par le client; la connexion est alors dans l'état SYN\_EMIS;
- ◆ le serveur envoie au client un ACK et son propre SYN qui initialise les numéros de séquence des données qu'il enverra; cet envoi se fait dans un seul paquet; la connexion est alors dans l'état SYN\_RECUE;
- ◆ le client envoie un ACK au serveur; la connexion est alors dans l'état ETABLIE; cet état correspond en fait au transfert de données entre le serveur et le client.

**Trois segments** ont donc du être échangés pour que la connexion soit établie (*three-way handshake* en anglais).

2) La **terminaison d'une connexion** qui passe par les étapes suivantes :

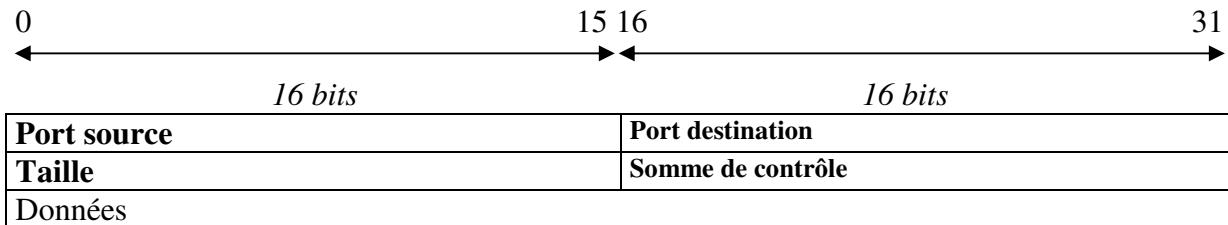
- ◆ le client ferme la connexion (instruction close) : on parle encore de *fermeture active* de la connexion; il envoie alors un premier segment TCP de type FIN; la connexion est alors dans l'état FIN\_ATTENTE\_1;
- ◆ le serveur reçoit le segment de fin et réalise une *fermeture passive*; la connexion est alors dans l'état ATTENTE\_FERMETURE; on dit encore que la connexion est à-demi fermée parce que des données peuvent encore transiter entre le client et le serveur;
- ◆ le serveur envoie alors un ACK que le client doit recevoir; la connexion est alors dans l'état FIN\_ATTENTE\_2;
- ◆ après un certain temps de latence, la connexion est définitivement fermée par le serveur; il envoie alors un segment TCP de type FIN; la connexion passe à l'état DERNIER\_ACK; lorsque le client reçoit ce segment, il envoie un ACK; la connexion est alors dans l'état TEMPORISATION pour attendre que tous les paquets aient bien disparu; si l'on désigne par **MSL** (Maximum Segment Lifetime) la durée de vie maximale d'un segment sur le réseau, alors le temps de latence sera de 2\*MSL; le MSL est fixé par l'implémentation, mais peut varier typiquement entre 30s et 2 min; il est clairement relié au TTL;
- ◆ la connexion passe donc finalement à l'état FERMEE.

On constatera que **quatre segments** ont du être échangés pour que la connexion soit finalement fermée.

## 12. UDP : l'autre protocole de la couche transport

On se souviendra donc qu'UDP (User Datagram Protocol) est **non fiable et orienté sans connexion** (comme une lettre). Il envoie des unités de transmission, malheureusement appelées des **datagrammes** ("malheureusement" parce que distincts des datagrammes d'IP, qui en restent un sur-ensemble !), d'une machine à l'autre, sans garantie de réception. Son grand intérêt est qu'il apporte la possibilité de multiplexer les informations entre différentes applications – autrement dit de mélanger les destinataires.

La structure d'un datagramme UDP, qui constitue donc les données d'un datagramme IP, est scandaleusement simple<sup>1</sup> :



On se doute évidemment du rôle des ports source et destination ... Le champ **Taille** correspond à la longueur de l'en-tête et des données UDP; sa valeur est au minimum de 8 bytes.

La **Somme de contrôle** est plus surprenante : elle se calcule non seulement sur l'en-tête et les données UDP, mais aussi sur des informations de l'en-tête IP, comme les adresses de la source et de la destination ainsi que sur le numéro du protocole; on espère ainsi réaliser un contrôle plus sûr à l'arrivée.

Il convient de signaler ici un effet pervers de la fragmentation IP. En effet, un problème apparaît si un paquet se perd :

- ◆ IP n'est pas un protocole fiable (pas de time-out, d'ACK, de réémission)<sup>2</sup>;
- ◆ UDP, à la différence de TCP, n'est pas non plus fiable .

On obtiendra donc une erreur irréparable par le stack UDP/IP. Le seul moyen d'éviter ce problème est de ne transmettre à la fois que des données d'une taille inférieure au MTU diminué de la taille des en-têtes ...



Tout cela est bien joli, mais assez théorique. Bien sûr, tout le monde se demande comment programmer une application TCP/IP. C'est par ici ...

<sup>1</sup> si on la compare à celle de TCP ou d'IP

<sup>2</sup> tout au plus pourra-t-il envoyer un message au protocole ICMP

## II. Les sockets : un client-serveur TCP itératif



*Ils croient que rien n'arrivera parce qu'ils ont fermé la porte.*

(M.Maeterlinck, Intérieur)

### 1. Un interface de communication

C'est donc l'œuvre de Berkeley d'avoir développé l'espéranto des interfaces de communication TCP/IP : les sockets.

Une **socket** est un point de communication bidirectionnel par lequel un processus peut émettre ou recevoir des informations.

Par "point de communication", il faut comprendre l'analogie d'un poste téléphonique, d'une radio, d'une boîte aux lettres, tous étant des points de contacts permettant la communication entre individus.

L'aspect "bidirectionnel" mérite d'être souligné : l'analogie avec une boîte aux lettres électronique est donc fondée, contrairement à celle avec une boîte aux lettres postale classique.



Toujours par comparaison avec les moyens de communication classiques, on peut aisément se représenter qu'une socket possède :

- ◆ **un type** : ou bien la communication se fait en *mode connecté* (comme une communication téléphonique), ou bien en *mode non connecté* (comme les lettres postales classiques que l'on se contente de poster);
- ◆ **un domaine** : on entend par là l'ensemble des sockets avec lesquelles une communication pourra être établie en utilisant cette socket; c'est l'équivalent de la notion de poste téléphonique interne (le domaine est l'entreprise) et externe (le domaine est le monde); on peut encore dire qu'un domaine définit une famille d'adresses, ou plus exactement le format des adresses possibles.

## 2. Représentation d'une socket en programmation C

A toute socket est associée une **structure socket** qui contient l'ensemble des informations la concernant. Cette structure joue pour les sockets le même rôle que les i-nodes UNIX pour les fichiers. Ainsi, si un i-node contient, pour un fichier donné, son type, ses permissions d'accès, l'identificateur du propriétaire, des dates, une table de pointeurs sur les blocs de data du fichier, une structure socket a la structure suivante (définie dans sys/socketvar.h) :

### La structure socket (sys/socketvar.h)

```
struct socket
{
    int      so_type;          /* generic type, see socket.h */
    int      so_options;       /* from socket call, see socket.h */
    short    so_linger;        /* time to linger while closing */
    short    so_timeo;         /* connection timeout */
    int      so_state;         /* internal state flags SS_*, below */
    caddr_t   so_pcb;          /* protocol control block */
    struct  protosw *so_proto; /* protocol handle */
    struct  socklocks *so_lock; /* socket structure lock(s) */
    struct  socket *so_head;   /* back pointer to accept socket */
    struct  socket *so_q0;     /* queue of partial connections */
    struct  socket *so_q;      /* queue of incoming connections */
    struct  socket *so_dq;     /* queue of defunct connections */
    short    so_q0len;         /* partials on so_q0 */
    short    so_qlen;          /* number of connections on so_q */
    short    so_qlimit;        /* max number queued connections */
    short    so_dqlen;         /* listener dequeues in progress */
    int      so_error;         /* error affecting connection */
    int      so_special;       /* special state flags SP_*, below */
    pid_t    so_pgid;          /* pgid for signals */
    u_int    so_oobmark;        /* chars to oob mark */

/*
 * Variables for socket buffering.
 */
    struct  sockbuf {
        u_int    sb_cc;           /* actual chars in buffer */
        u_int    sb_hiwat;        /* max actual char count */
        u_int    sb_mbcnt;        /* chars of mbufs used */
        u_int    sb_mbmax;        /* max chars of mbufs to use */
        u_int    sb_dbcnt;        /* DART: chars of dart-style used */
        u_int    sb_dbmax;        /* DART: max chars to use */
        int      sb_lowat;        /* low water mark */
        struct  mbuf *sb_mb; /* the mbuf chain */
        struct sbselque { /* process selecting read/write */
            struct sbselque *next, *prev;
        } sb_selq;
        int      sb_flags;         /* flags, see below */
        int      sb_timeo;        /* timeout for read/write */
    };
};
```

```

#if __STDC__
    void (*sb_wakeup)(caddr_t, int);
#else
    void (*sb_wakeup)(); /* upcall instead of sowakeup */
#endif
    caddr_t      sb_wakearg; /* (*sb_wakeup)(sb_wakearg, state) */
    sb_lock_t sb_lock; /* sockbuf lock (in socklocks) */
} so_rcv, so_snd;

```

En pratique, une socket est identifiée par un **descripteur** (un **handle**) qui est de même nature que ceux identifiant les fichiers (ce qui permettra des redirections). Tout comme pour les fichiers, ce sera ce handle qui sera essentiellement utilisé dans la suite. Il sera demandé au système d'exploitation par la primitive socket() habilement paramétrée. C'est le header sys/socket.h qui apporte les déclarations nécessaires à cette création d'une socket et à son utilisation. On peut, en passant, remarquer que son implémentation Digital-Unix, comme celle de tous ses associés, est prévue pour être utilisée éventuellement en C++<sup>1</sup> :

### Adaptation des headers TCP/IP au C++

```

/* SOCKET.H */

#ifndef _SYS_SOCKET_H_
#define _SYS_SOCKET_H_

#ifdef __cplusplus
extern "C" {
#endif
...
...
#ifdef __cplusplus
}
#endif
#endif /* _SYS_SOCKET_H_ */

```

### 3. La création d'une socket

Pour obtenir du système d'exploitation local un handle sur une socket d'un type, d'un protocole et d'un domaine donnés, un process utilise donc la primitive :

```

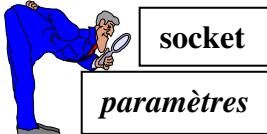
int socket ( <domaine défini par une famille de protocoles – int>,
             <type – int>,
             <protocole utilisé – int>);

```

En cas de succès, cette fonction renvoie le handle de la socket. Sinon, la valeur de retour est l'habituel -1 du monde UNIX.

---

<sup>1</sup> je dis cela et je ne dis rien ... ;-)



Pour ce qui est du domaine, on dispose de constantes **AF\_\*** (Address Family) :

#### domaines (socket.h) : familles d'adresses

```
/*
 ** Address families.
 */

#define AF_UNIX      1      /* local to host (pipes, portals) */
#define AF_INET      2      /* internetwork: UDP, TCP, etc. */
...
#define AF_ISO       7      /* ISO protocols */
#define AF_OSI        AF_ISO
...
#define AF_SNA       11     /* IBM SNA */
#define AF_DECnet    12     /* DECnet */
...
#define AF_APPLETALK 16     /* Apple Talk */
```

et de constantes **PF\_\*** (Protocol Family) équivalentes (chaque protocole a ses adresses) :

#### domaines (socket.h) : familles de protocoles

```
/*
 * Protocol families, same as address families for now.
 */

#define PF_UNIX      AF_UNIX
#define PF_INET      AF_INET
...
#define PF_SNA       AF_SNA
#define PF_DECnet   AF_DECnet
...
#define PF_APPLETALK AF_APPLETALK
#define PF_802        AF_802
#define PF_OSI        AF_OSI
#define PF_X25        AF_X25
#define PF_IPX        AF_IPX
```

Le type de la socket détermine les propriétés de la communication : mode connecté ou non connecté, fiabilité ou pas, non-duplication, possibilité d'envoyer des données prioritaires. Les constantes symboliques **SOCK\_\*** de sys/socket.h sont :

#### types (socket.h)

```
/*
 * Types
 */

#define SOCK_STREAM  1
/* stream socket - mode connecté fiable avec possibilité de données urgentes */
#define SOCK_DGRAM   2
/* datagram socket - mode non connecté et non fiable */
#define SOCK_RAW     3
/* raw-protocol interface – bas niveau, le plus souvent IP */
```

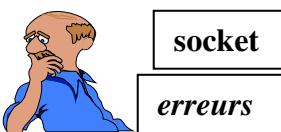
Le *protocole* peut être, lui aussi, également spécifié en utilisant une constante prédéfinie du header netinet/in.h :

<b>protocoles (in.h)</b>			
/*			
* Protocols			
*/			
#define IPPROTO_IP	0		/* dummy for IP */
#define IPPROTO_ICMP	1		/* control message protocol */
#define IPPROTO_IGMP	2		/* group mgmt protocol */
#define IPPROTO_GGP	3		/* gateway^2 (deprecated) */
#define IPPROTO_IPIP	4		/* IP inside IP */
#define IPPROTO_TCP	6		/* tcp */
#define IPPROTO_EGP	8		/* exterior gateway protocol */
#define IPPROTO_PUP	12		/* pup */
#define IPPROTO_UDP	17		/* user datagram protocol */
#define IPPROTO_IDP	22		/* xns idp */
#define IPPROTO_TP	29		/* tp-4 w/ class negotiation */
#define IPPROTO_RSVP	46		/* resource reservation proto */
#define IPPROTO_EON	80		/* ISO cnlp */
#define IPPROTO_RAW	255		/* raw IP packet */
#define IPPROTO_MAX	256		

*Mais il y a plus simple* : il arrive souvent que, dans un domaine donné, le type de socket utilisé implique automatiquement qu'un seul protocole est possible. Ainsi, dans les deux cas qui nous concernent directement :

<b>domaine</b>	<b>type</b>	<b>seul protocole utilisable</b>
AF_INET	SOCK_STREAM	TCP
AF_INET	SOCK_DGRAM	UDP

Dans ce cas, il suffit de fournir comme troisième paramètre à la primitive socket la valeur nulle : elle signifie que le système d'exploitation choisira le seul protocole adapté.



Si la valeur de retour est  $-1$ , une erreur s'est produite et la variable globale `errno`<sup>1</sup> est positionnée sur l'une des valeurs suivantes :

<b>valeur de errno</b>	<b>erreur</b>
#define EAFNOSUPPORT 47	/* Address family not supported by protocol family */ : à votre avis ? la famille d'adresses du domaine n'est pas disponible dans le noyau
#define EPROTONOSUPPORT 43	/* Protocol not supported */ : à votre avis ? le domaine et le protocole sont incompatibles

<sup>1</sup> pour rappel, une valeur nulle de `errno` correspond à l'absence d'erreur.

#define EMFILE 24	/* Too many open files */ : la table des handles est pleine – <i>il faudrait fermer des fichiers ou des sockets</i>
#define ENOBUFS 55	/* No buffer space available */ : les ressources sont insuffisantes - <i>il faudrait libérer des ressources systèmes</i>
#define ENOMEM 12	/* Not enough core */ : il n'y a plus de mémoire pour augmenter la table des descripteurs – <i>il faudrait libérer de la mémoire</i>
#define EACCES 13	/* Permission denied */ : le type SOCK_RAW n'est utilisable dans le domaine AF_INET que si l'on est utilisateur privilégié

Donc, finalement, la création d'une socket suivra le cheminement suivant :

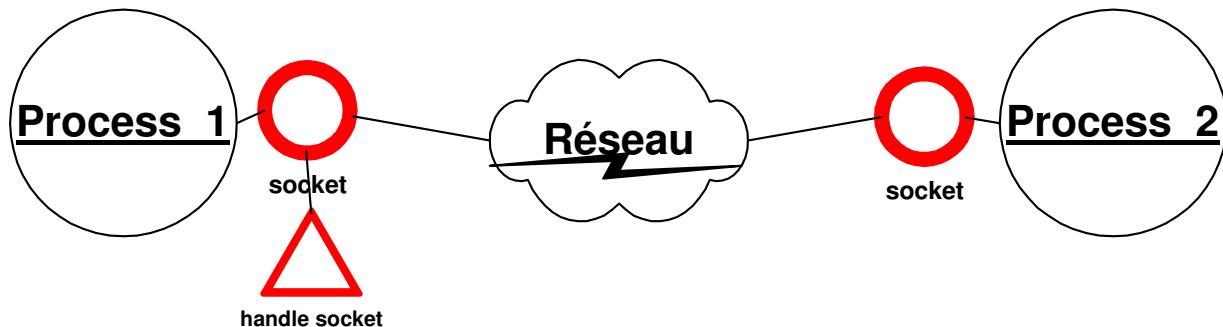
```
SOCKET01.C
/*
 * SOCKET01.C
 * - Claude Vilvens -
 */
#include <stdio.h>
#include <stdlib.h> /* pour exit */

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <errno.h>

int main()
{
    int hSocket, /* Handle de la socket */

    /* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0); /* au sens strict : PF_INET */
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

    return 0;
}
```



Précisons que, sur les machines Sun, la compilation et le linkage des programmes utilisant les sockets ci-dessus réclament les options de librairie **-lsl (ou -lns)** et **-lsocket** :

```
%> cc -o csocket socket01.c -lsl -lsocket
```

On peut encore remarquer que le programme suivant génère une erreur :

### SOCKET02.C

```
/* SOCKET02.C
- Claude Vilvens -
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */

#include <sys/socket.h> /* pour les types de socket */
#include <errno.h>
#include <netinet/in.h> /* pour les noms de protocoles */
int main()
{
    int hSocket; /* Handle de la socket */

/* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

    ... /* A suivre */

    return 0;
}
```

On obtient :

```
| Erreur de creation de la socket 13
```

ce qui correspond bien à l'erreur de droits insuffisants.

#### Remarque

Sous UNIX, on usera du traditionnel

**man <nom de la primitive>**

pour obtenir de l'aide sur une fonction.

## 4. La fermeture d'une socket

Bien logiquement, on libère un descripteur (handle) de socket comme celui d'un fichier et on utilise donc la fonction :

```
int close (<handle de la socket – int>);
```

qui renvoie -1 en cas de d'échec; dans ce dernier cas, errno prend la valeur EBADF.

Il faut cependant remarquer que, en tous cas pour une socket de type SOCK\_STREAM du domaine AF\_INET, la fermeture n'aboutira à la libération complète des ressources que lorsque les éventuelles données se trouvant encore dans le buffer d'émission de la socket auront été acheminées à bon port (ce qui, par analogie, correspond bien au comportement de close dans le cas des fichiers).

En réalité, les choses sont un peu plus subtiles. En effet, un descripteur de socket (comme d'ailleurs un descripteur de fichier sous UNIX) est accompagné, de par la structure qu'il désigne, d'un compteur de références. La fonction close() décrémente ce compteur et ce n'est que si ce compteur tombe à 0 que les ressources sont effectivement libérées.

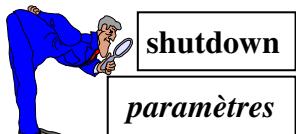
La librairie TCP/IP fournit une autre primitive qui permet un traitement plus fin de la cessation des activités sur la socket :

```
int shutdown (<handle de la socket – int>,  
               <sens de la fermeture - int>);
```

Cette primitive permet :

- ◆ de fermer la socket, quelle que soit la valeur du compteur de références;
- ◆ de fermer la socket en écriture (signifiant ainsi à l'autre extrémité que l'on a terminé d'envoyer des informations) tout en la laissant ouverte en lecture (signifiant ainsi à l'autre extrémité que l'on est encore disposé à lire des informations), ou l'inverse.

Une valeur renournée de -1 signifiera une erreur.

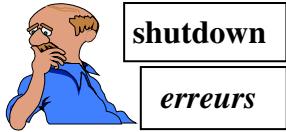


Le deuxième paramètre détermine l'extrémité de la communication sur laquelle l'opération de fermeture a lieu; il a trois valeurs possibles :

- ◆ 0<sup>1</sup> : la socket en fermée en lecture (autrement dit, un recv() donnera 0);
- ◆ 1 : la socket en fermée en écriture (autrement dit, un send() génère le signal SIGPIPE – s'il est capté, errno contiendra EPIPE );
- ◆ 2 : la socket en fermée tant en lecture qu'en écriture.

---

<sup>1</sup> la norme Posix.1g définit trois constantes SHUT\_RD, SHUT\_WR et SHUT\_RDWR qui remplacent 0,1 et 2



Si la valeur de retour est  $-1$ , une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
<code>#define EBADF 9</code>	<code>/* Bad file number */</code> : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
<code>#define ENOTSOCK 38</code>	<code>/* Socket operation on non-socket */</code> : le descripteur n'est pas associé à une socket, mais à un fichier

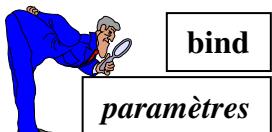
## 5. L'attachement d'une socket à une adresse (bind)

Après sa création au moyen de la primitive `socket()`, la socket n'est désignée, au moyen de son descripteur, que localement au processus. En fait, la socket n'est pour l'instant reliée à aucune adresse qui lui permettrait d'être désignée de l'extérieur, ce qui permettrait aux autres process, locaux ou distants, d'y écrire ou d'y lire des données.

La primitive `bind()` a pour rôle de réaliser cette association avec une adresse, dont le format sera déterminé par le domaine de la socket :

```
int bind      ( <handle de la socket - int>
                <adresse au format du domaine de la socket - const struct sockaddr *>,
                <longueur de l'adresse - size_t>);
```

Une valeur renvoyée de  $-1$  signifiera une erreur.



La structure `sockaddr`, définie dans `sys/socket.h` :

<b>structure sockaddr</b>
<b>struct sockaddr</b> { unsigned char sa_len;      /* total length */ sa_family_t sa_family;    /* address family */ char        sa_data[14];   /* actually longer; address value */ };

est, en fait, une structure générique : elle devra en effet être particularisée (oserais-je dire spécialisée, POO oblige ?) pour décrire l'adresse selon le domaine utilisé. Ainsi, dans le cas du domaine internet `AF_INET`, l'adresse à utiliser sera décrite au moyen d'une structure `sockaddr_in` définie dans `netinet/in.h` (le domaine `AF_UNIX` utilise une structure `sockaddr_un`) :

```
structure sockaddr_in (in.h)
struct sockaddr_in
{
    unsigned char sin_len;          /* on peut ignorer */
    sa_family_t   sin_family;       /* ici, AF_INET */
    in_port_t     sin_port;         /* numéro de port */
    struct in_addr sin_addr;       /* adresse internet de la machine considérée */
    char   sin_zero[8];            /* 8 caractères nuls */

};

/* a structure for historical reasons */
struct in_addr
{
    in_addr_t s_addr;
};
```

avec, dans types.h :

```
typedef unsigned char sa_family_t;
typedef unsigned short in_port_t;
typedef unsigned int in_addr_t;
```

Donc, après traduction :

```
struct sockaddr_in
{
    unsigned char sin_len;
    unsigned char sin_family;
    unsigned short sin_port;
    struct in_addr { unsigned int s_addr; }sin_addr;
    char   sin_zero[8];
};
```

D'où sortira *l'adresse de la machine* (donc le champ sin\_addr) ? On pourrait penser à encoder l'adresse IP telle quelle (encore que cela ne soit pas si simple) : mais on perdrat évidemment toute portabilité d'une machine à l'autre, sans parler des changements d'adresses. Il est donc beaucoup plus raisonnable de demander au système d'exploitation de fournir cette adresse en fonction du nom de la machine. Ceci se réalise au moyen de la fonction déclarée dans netdb.h :

```
struct hostent *gethostbyname (<nom de la machine selon le DNS - char*>);
```

où la structure renvoyée est :

```
structure hostent
struct hostent
{
    char   *h_name;           /* official name of host */
    char   **h_aliases;        /* alias list */
    int    h_addrtype;         /* host address type */
```

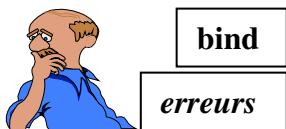
```
int h_length;           /* length of address */
char **h_addr_list;    /* list of addresses from name server */
#define h_addr h_addr_list[0]      /* address, for backward compatibility */
};
```

On peut aussi parfois utiliser la constante définie dans in.h :

```
#define INADDR_ANY (unsigned int)0x00000000
```

ce qui permettra d'associer la socket à toutes les adresses possibles de la machine - celle-ci peut en effet très bien avoir plusieurs interfaces ! De cette manière, la socket pourra être contactée sur une quelconque de ces adresses, donc par n'importe quel réseau sur lequel la machine se trouve. C'est particulièrement intéressant pour les passerelles. A l'inverse, on peut très bien associer à la socket une seule adresse bien précise, restreignant ainsi les connexions possibles à cette seule adresse.

La spécification du **port** est évidemment fondamentale pour un serveur, puisque ce port correspond à un service précis. Par contre, dans le cas d'un client, le numéro de port peut être quelconque (exactement comme un particulier peut appeler d'un numéro quelconque pour obtenir le service qui se trouve à telle extension (=le port) d'un numéro général (=l'adresse)).



Si la valeur de retour est  $-1$ , une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define EADDRNOTAVAIL 49	/* Can't assign requested address */ : l'adresse spécifiée n'est pas accessible à partir de l'ordinateur local
#define EADDRINUSE 48	/* Address already in use */ : c'est clair ...
#define EINVAL 22	/* Invalid argument */ : moins clair – c'est l'inverse du précédent, c'est-à-dire que la socket est déjà liée à une adresse
#define EACCES 13	/* Permission denied */ : l'adresse spécifiée est protégée : l'utilisateur courant ne peut l'accéder
#defineEFAULT 14	/* Bad address */ : l'adresse n'est pas accessible en lecture dans l'espace d'adressage de l'utilisateur

Donc, finalement, la liaison de notre socket à l'adresse de l'ordinateur local faisant office de serveur sera la suivante :

## TCPIITER01.C

```
....
```

```
#define PORT 50000 /* Port d'ecoute de la socket serveur */

int main()
{
    int hSocket; /* handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format réseau */
    struct sockaddr_in adresseSocket;
        /* Structure de type sockaddr contenant les infos adresses -
           ici,structure adaptée au cas de TCP */
    int tailleSockaddr_in;

/* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0);
    ...

/* 2. Acquisition des informations sur l'ordinateur local */
    if ( (infosHost = gethostbyname("sunray2v440"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);

/* 3. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */
    adresseSocket.sin_port = htons(PORT);
        /* conversion numéro de port au format réseau */
    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocket, (struct sockaddr *)&adresseSocket, sizeof(struct sockaddr_in))
        == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        exit(1);
    }
    else printf("Bind adresse et port socket OK\n");

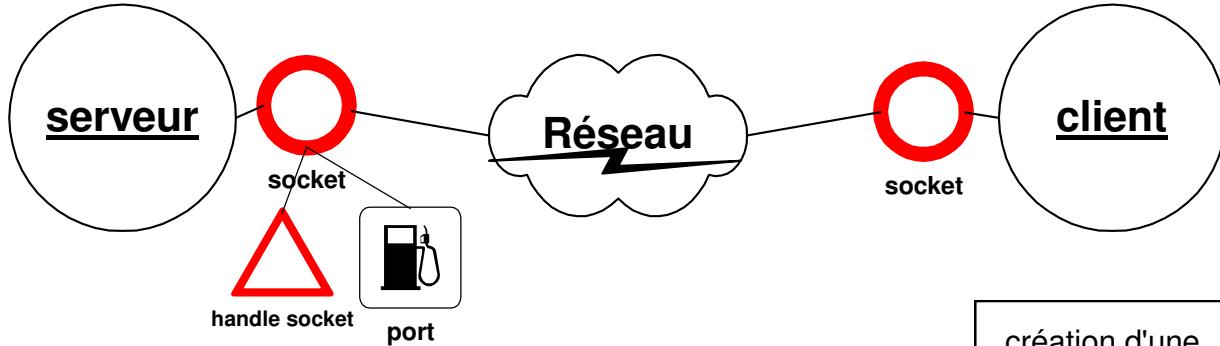
    ... /* A suivre */

    return 0;
}
```

Précisons encore que, sur les machines Sun, la compilation et le linkage du code ci-dessus réclament deux options de librairie

```
%> cc -o cser01 tcriter01.c -lsocket -lsl
```

On remarquera encore que le numéro de port doit être converti au format réseau – nous reviendrons sur les fonctions de conversion, comme **hton()**, plus loin.



#### Remarque

On pourrait être tenté (et c'est légitime et réconfortant) de vouloir afficher l'adresse obtenue par `gethostbyname()`. C'est faisable, mais par :

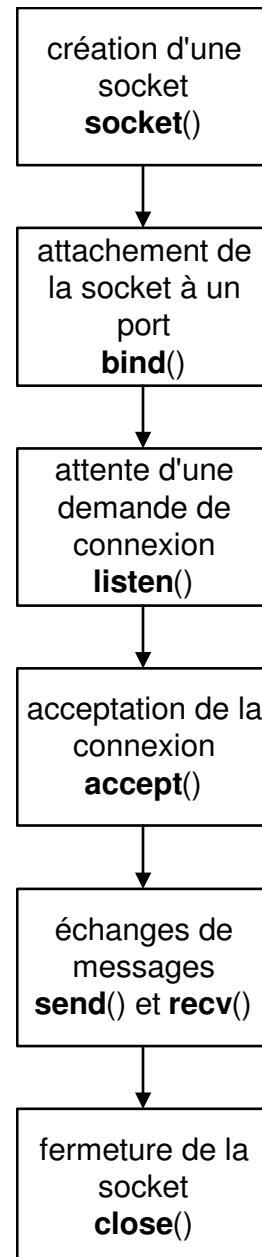
```
printf("Adresse IP = %s\n",inet_ntoa(adresseIP));
```

En fait, l'adresse est au format réseau et il faut réaliser une conversion – nous avons déjà promis d'en reparler plus loin.

## 6. Un serveur monoconnexion

Notre objectif est donc ici de d'implémenter un modèle client serveur fonctionnant selon le mode connecté (SOCK\_STREAM) dans le domaine internet (AF\_INET). Le protocole de transport sous-jacent est donc bien TCP. Nous supposerons ici qu'il s'agit d'une serveur monoconnexion : un seul client peut dialoguer avec le serveur. Nous ferons mieux plus loin, avec un serveur multi-connexions autorisant plusieurs clients à se connecter sur le même serveur.

Il appartient au serveur de se mettre au préalable en attente d'une demande de connexion (au moyen de la primitive **listen()**) ; celle-ci sera émise par un client qui réalise un appel de la primitive **connect()** et qui prend donc l'initiative de cette demande. Lorsqu'une nouvelle connexion est demandée, le serveur en prend connaissance et l'accepte au moyen de la primitive **accept()**. La socket peut donc à présent être utilisée pour la communication au moyen des primitives **send()** et **recv()**.



## 7. La mise à l'écoute sur une socket (listen)

Comme déjà précisé, le serveur prévient le système qu'il se met en attente d'une demande de connexion sur la socket, préalablement créée et reliée à un port, au moyen de la primitive :

```
int listen      (< handle de la socket – int>,
                 <nombre de connexions pendantes – int>);
```

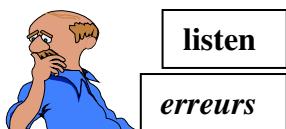
Comme d'habitude, cette fonction renvoie 0 ou –1 en cas de succès ou d'échec.



Le deuxième paramètre spécifie le nombre maximum de connexions qui peuvent être reçues par le serveur mais qui n'ont pas encore été prises en compte par celui-ci au moyen de la primitive accept() : de telles demandes sont appelées des ***connexions pendantes*** et sont enfilées dans un FIFO pointé par le champ so\_q de la structure socket. Le nombre maximum de connexions pendantes est donné par la constante (définie dans sys/socket.h) :

```
#define SOMAXCONN 1024
```

Sa valeur est variable selon les implémentations (Digital, IX, HP, Ultrix, ...). En pratique, des limites de 5 à 20 sont réalistes. Dans le cas de notre modeste exemple, les connexions pendantes autres que celle en cours sont ignorées – mais cela ne durera pas ...



Si la valeur de retour est –1, une erreur s'est produite et la variable globale errno est positionnée sur l'une des valeurs suivantes :

valeur de errno	erreur
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define EOPNOTSUPP 45	/* Operation not supported on socket */ : le type de socket utilisé ne permet pas d'appel à listen
#define EINVAL 22	/* Invalid argument */ : la socket est connectée ou fermée ou non attachée
#define ENOBUFS 55	/* No buffer space available */ : les ressources sont insuffisantes - <i>il faudrait libérer des ressources systèmes</i>

Pour notre premier essai, cela peut donner :

**TCPITER01.C**

```
.....
#define PORT 50000 /* Port d'ecoute de la socket serveur */

int main()
{
    int hSocket; /* handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format réseau */
    struct sockaddr_in adresseSocket;
        /* Structure de type sockaddr contenant les infos adresses - ici, cas de TCP */
    int tailleSockaddr_in;

/* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0);
    ...
/* 2. Acquisition des informations sur l'ordinateur local */
    ...
/* 3. Préparation de la structure sockaddr_in */
    ...
/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    ...

/* 5. Mise à l'écoute d'une requête de connexion */
    if (listen(hSocket,SOMAXCONN) == -1)
    {
        printf("Erreur sur la liste de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

    ... /* A suivre */

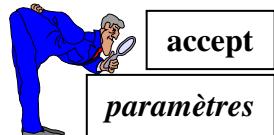
    return 0;
}
```

**8. La prise de connaissance d'une connexion (accept)**

Après l'exécution de listen(), la connexion est donc pendante (les anglo-saxons disent *partial*) : il faut encore que le processus serveur prenne en compte cette connexion, et prenne connaissance de l'adresse de la socket du client qui se connecte, au moyen de la primitive :

int <b>accept</b> (<handle de la socket – int>, <adresse reçue de la socket client - struct sockaddr *>, <longueur de la structure qui reçoit l'adresse client– int *>);
--

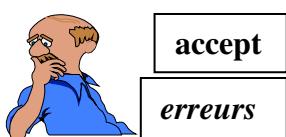
La valeur renvoyée est un fait un handle de socket dédié à la connexion acceptée : il s'agit en fait d'une *duplication de la socket initiale*. Nous en reparlerons dans le cas d'un serveur à visées multiconnexions. Bien sûr, la valeur –1 signifie qu'il y a eu erreur.



**L'adresse de la socket du client** sera écrite, lors de l'exécution de la fonction, à l'adresse de la structure passée comme deuxième paramètre. La longueur de la structure doit être passée par adresse lors de l'appel dans le troisième paramètre.

### Remarque

Par défaut, le processus serveur reste en attente s'il n'y a pas de connexion pendante : accept est donc *bloquant*.



Si la valeur de retour est –1, une erreur s'est produite et la variable globale errno est positionnée sur l'une des valeurs suivantes :

valeur de errno	erreur
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define EOPNOTSUPP 45	/* Operation not supported on socket */ : le type de socket utilisé ne permet pas d'appeler à listen
#define EINVAL 22	/* Invalid argument */ : la socket est connectée ou fermée ou non attachée et ne peut donc pas accepter de connexion
#define EFAULT 14	/* Bad address */ : l'adresse de la structure sockaddr est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)
#define EWOULDBLOCK 35 #define EAGAIN EWOULDBLOCK	/* Operation would block */ : il n'y a pas de connexion pendante et la socket n'est pas bloquante (ce qu'elle devrait être)
#define EMFILE 24	/* Too many open files */ : la table des handles est pleine – <i>il faudrait fermer des fichiers ou des sockets</i>
#define ENOMEM 12	/* Not enough core */ : il n'y a plus de mémoire pour augmenter la table des descripteurs – <i>il faudrait libérer de la mémoire</i>
#define EINTR 4	/* Interrupted system call */ : la fonction a été interrompue

Pour notre exemple :

**TCPIITER01.C**

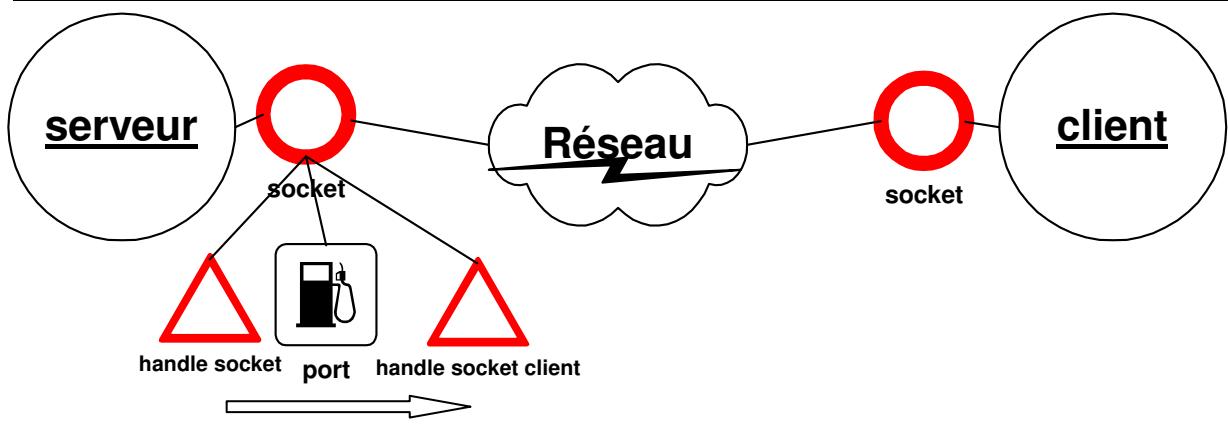
```
.....
#define PORT 50000 /* Port d'ecoute de la socket serveur */

int main()
{
    int hSocket; /* handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format réseau */
    struct sockaddr_in adresseSocket;
        /* Structure de type sockaddr contenant les infos adresses - ici, cas de TCP */
    int tailleSockaddr_in;
/* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0);
    ...
/* 2. Acquisition des informations sur l'ordinateur local */
    ...
/* 3. Préparation de la structure sockaddr_in */
    ...
/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    ...
/* 5. Mise à l'écoute d'une requête de connexion */
    ...
/* 6. Acceptation d'une connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (accept(hSocket, (struct sockaddr *)&adresseSocket,
                &tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");
    ... /* A suivre */

    return 0;
}
```

Notre socket, qui était donc initialement connectée sur le serveur pour permettre le listen, est à présent connectée sur le client. **Cette manière de faire rend impossible toute nouvelle connexion : nous devrons donc, en fait, utiliser une autre socket, dite "socket dupliquée", pour résoudre ce problème.**

De plus, si certains systèmes UNIX acceptent l'utilisation d'une seule socket du côté serveur, d'autres exigent l'usage de cette "socket dupliquée".



## 9. Deux sockets pour un serveur

A ce stade, il est donc clair que notre serveur est assez peu efficace : *il ne peut s'occuper que d'un client à la fois*, puisqu'il utilise sa **socket d'écoute** pour communiquer avec celui-ci. Il lui faudrait donc une autre socket dédiée à la connexion qui s'est établie : on l'appelle une **socket de service**. La socket d'écoute pourrait ainsi être libérée pour se remettre à l'écoute d'une demande de connexion d'un autre client.

Or, précisément, la primitive accept() renvoie comme valeur une **socket dupliquée de la socket d'écoute**. Il n'y a donc plus qu'à utiliser cette socket de service pour la suite des échanges avec le client connecté. Il convient de remarquer que la socket de service permet simplement d'envoyer et de recevoir des bytes sur la connexion : *elle ne permet pas d'accepter de nouvelles connexions*.

Dans un premier temps, cela ne change guère le code de notre serveur : on déclare deux sockets au lieu d'une et **la socket de service prend le relais de la socket d'écoute une fois la connexion établie**.

### TCPITER01D.C

```
/* TCPITER01D.C
 * serveur itératif mono-connexion *
 - Claude Vilvens -
 */
#include <stdio.h>
...
#define PORT 50000 /* Port d'écoute de la socket serveur */
#define MAXSTRING 100 /* Longeur des messages */

int main()
{
    int hSocketEcoute, /* Handle de la socket d'écoute */
        hSocketService; /* Handle de la socket de service connectée au client */
    ...
/* 1. Création de la socket */
    hSocketEcoute= socket(AF_INET, SOCK_STREAM, 0);
    if (hSocketEcoute== -1)
    {
        printf("Erreur de création de la socket %d\n", errno); exit(1);
    }
    else printf("Création de la socket OK\n");
}
```

```

/* 2. Acquisition des informations sur l'ordinateur local */

...
/* 3. Préparation de la structure sockaddr_in */

...
/* 4. Le système prend connaissance de l'adresse et du port de la socket */

...
/* 5. Mise à l'écoute d'une requête de connexion */

...
/* 6. Acceptation d'une connexion */
tailleSockaddr_in = sizeof(struct sockaddr_in);
if ( (hSocketService =
      accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) ) == -1)
{
    printf("Erreur sur l'accept de la socket %d\n", errno);
    close(hSocket); exit(1);
}
else printf("Accept socket OK\n");

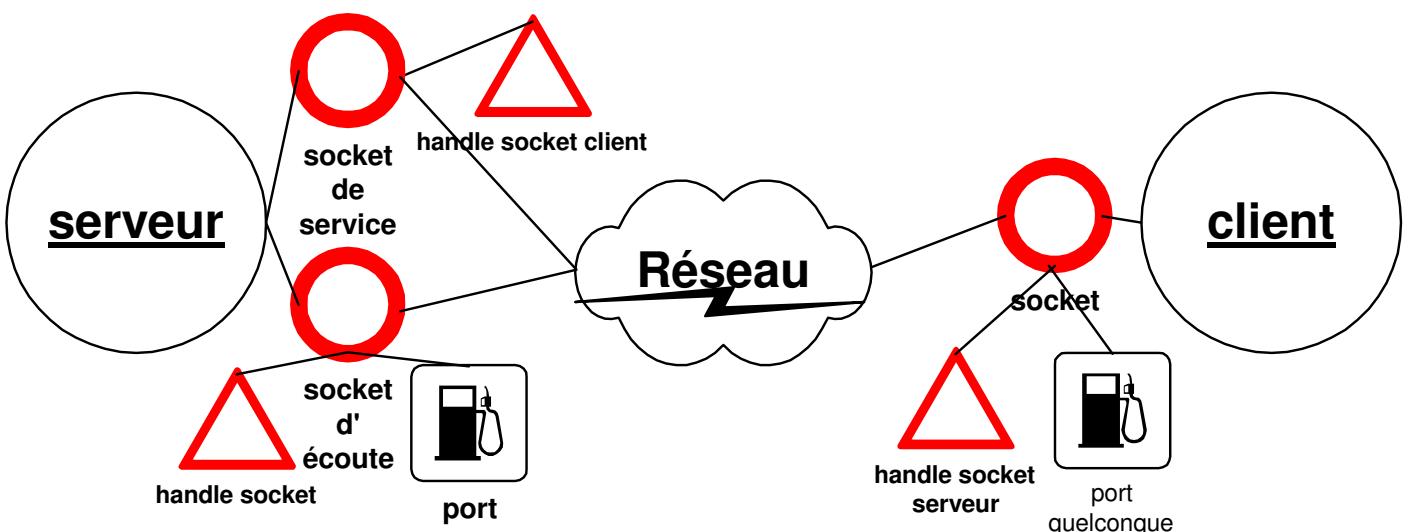
... /* A suivre */

/* ? . Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */
printf("Socket connectée au client fermée\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermée\n");

return 0;
}

```

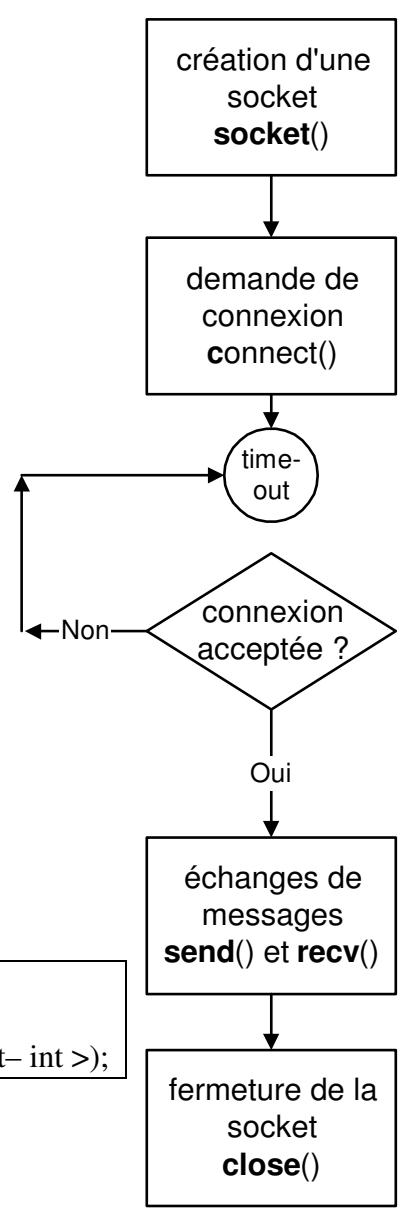
Schématiquement :



## 10. Le client et sa connexion au serveur (connect)

Un client de notre serveur va donc solliciter une connexion à celui-ci. Il le fera en utilisant la primitive **connect()**. Pour cela, il peut demander au DNS<sup>1</sup> l'adresse IP du serveur. Mais il devra tout de même savoir à quel port du serveur se connecter pour obtenir le service attendu. Par contre, le port client est quelconque, si bien qu'un appel de **bind()** n'est pas utile.

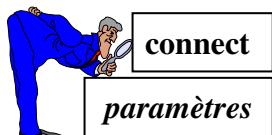
A remarquer que si la liste des connexions pendantes du serveur est pleine (et si la socket locale est bloquante), la demande de connexion sera réitérée. Ce n'est qu'à la fin d'un time-out que la demande est effectivement abandonnée.



La primitive typique du client est :

```
int connect (<handle de la socket - int>,
             <adresse du serveur - struct sockaddr *>,
             <longueur de la structure qui reçoit l'adresse client- int >);
```

On obtient 0 comme valeur de retour en cas de succès, -1 sinon.

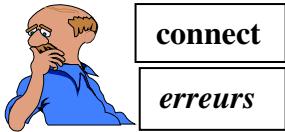


Cette fonction demande de connecter la socket(locale au client) dont on passe le handle à la socket dont les caractéristiques (dont l'adresse) se trouvent dans la structure sockaddr dont on passe l'adresse en deuxième argument. Comme déjà précisé, il n'est pas nécessaire que la socket locale ait été attachée au préalable, au moyen de bind(), à une adresse (avec son port) : connect() réalise dans ce cas un attachement à un port quelconque.

### Remarque

Par défaut, l'appel à connect() est bloquant.

<sup>1</sup> voir chapitre consacré aux adresses



Si la valeur de retour est `-1`, une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
<code>#define EBADF 9</code>	<code>/* Bad file number */</code> : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
<code>#define ENOTSOCK 38</code>	<code>/* Socket operation on non-socket */</code> : le descripteur n'est pas associé à une socket, mais à un fichier
<code>#define EOPNOTSUPP 45</code>	<code>/* Operation not supported on socket */</code> : le type de socket utilisé ne permet pas d'appel à connect
<code>#define ENETUNREACH 51</code>	<code>/* Network is unreachable */</code> : la machine cible n'a pu être atteinte [soft error – le SYN du client a occasionné une erreur ICMP]
<code>#define EISCONN 56</code>	<code>/* Socket is already connected */</code> : à votre avis ?
<code>#define ETIMEDOUT 60</code>	<code>/* Connection timed out */</code> : clair ! le client ne reçoit pas de réponse à son segment SYN dans un délai raisonnable
<code>#define ECONNREFUSED 61</code>	<code>/* Connection refused */</code> : le serveur a refusé la connexion – <i>le serveur n'a probablement pas exécuté listen() et personne n'écoute donc sur ce port</i> [hard error – le serveur envoie un segment RST]
<code>#define EADDRINUSE 48</code>	<code>/* Address already in use */</code> : la socket distante est déjà connectée
<code>#define EFAULT 14</code>	<code>/* Bad address */</code> : l'adresse de la strcuture <code>sockaddr</code> est incorrecte
<code>#define EINTR 4</code>	<code>/* Interrupted system call */</code> : la fonction a été interrompue

Pour notre exemple, le début du client sera écrit de la manière suivante :

### TCPCLI01.C

```
...
#define PORT 50000 /* Port d'écoute de la socket serveur */

int main()
{
    int hSocket; /* handle de la socket */
    struct hostent * infosHost; /* pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format réseau */
    struct sockaddr_in adresseSocket;
        /* Structure de type sockaddr contenant les infos adresses - ici, cas de TCP */
    unsigned int tailleSockaddr_in;
    int ret; /* valeur de retour */
```

```

/* 1. Création de la socket */
hSocket = socket(AF_INET, SOCK_STREAM, 0);
if (hSocket == -1)
{
    printf("Erreur de creation de la socket %d\n", errno); exit(1);
}
else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur distant */
if ( (infosHost = gethostbyname("sunray2v440"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host distant %d\n", errno); exit(1);
}
else printf("Acquisition infos host distant OK\n");
memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
printf("Adresse IP = %s\n",inet_ntoa(adresseIP));

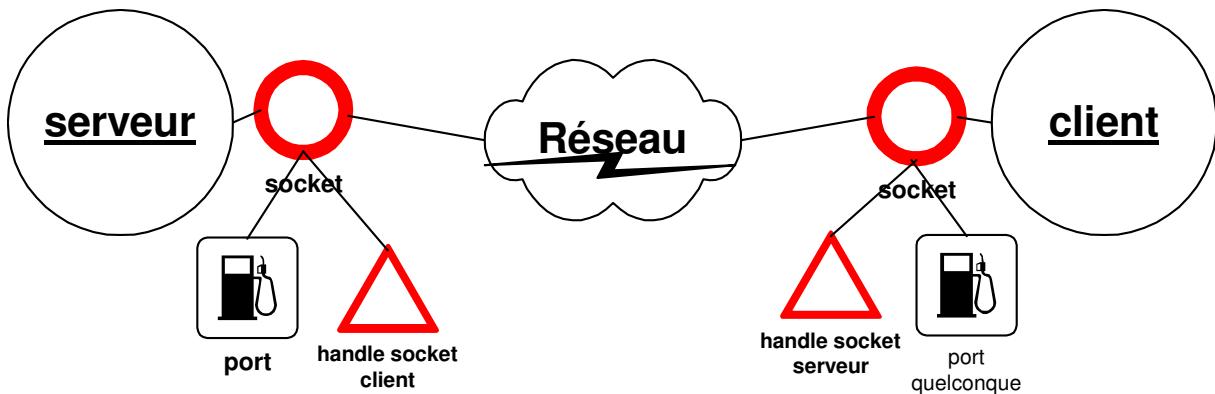
/* 3. Préparation de la structure sockaddr_in */
memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
adresseSocket.sin_family = AF_INET; /* Domaine */
adresseSocket.sin_port = htons(PORT); /* conversion port au format réseau */
memcpy(&adresseSocket.sin_addr, infosHost->h_addr,infosHost->h_length);

/* 4. Tentative de connexion */
tailleSockaddr_in = sizeof(struct sockaddr_in);
if (( ret = connect(hSocket, (struct sockaddr *)&adresseSocket, tailleSockaddr_in) )
    == -1)
{
    printf("Erreur sur connect de la socket %d\n", errno);
    close(hSocket); exit(1);
}
else printf("Connect socket OK\n");

... /* A suivre */

return 0;
}

```



## 11. L'échange de données entre le client et le serveur (send et recv)

A ce point de l'exposé, le client dispose d'une socket attachée au serveur (par `connect()`) et celui-ci dispose d'une socket attachée au client (par `accept()`). L'échange de caractères entre les deux processus distants est donc à présent possible.

Cette communication utilise, pour chaque socket de type `SOCK_STREAM`, un buffer de lecture et un autre d'écriture. TCP garantit que les caractères reçus ont été fournis dans l'ordre initial. *Les opérations de lecture et d'écriture sont bloquantes* : un buffer vide bloque l'opération de lecture, un buffer plein bloque l'opération d'écriture.

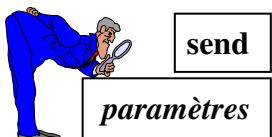
Examinons de plus près les deux primitives classiques de communication.

### 11.1 L'émission de caractères (send)

C'est le rôle de la primitive

```
int send      (<handle de la socket cible – int>,
              <adresse de la suite de caractères à envoyer - const void *>,
              <nombre de caractères à envoyer – int>,
              <flag d'urgence - int>);
```

La valeur renvoyée est le nombre de caractères transmis ou `-1` en cas d'erreur.



Seul le dernier paramètre demande une explication. Dans le domaine `AF_INET`, il vaudra toujours `0` pour des communications normales. Ce n'est que dans le cas où les données transmises sont **urgentes** et sont **prioritaires** par rapport aux autres que ce paramètre prendra la valeur (définie dans `sys/socket.h`) :

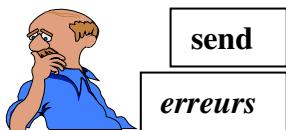
```
#define      MSG_OOB      0x1      /* process out-of-band data */
```

Les anglo-saxons diront d'ailleurs que de tels messages sont ***Out Of Band Data***, signifiant par là qu'*ils ne suivent pas le flux normal des données*.

Pour être complet, précisons que ce dernier paramètre peut également valoir :

```
#define      MSG_DONTROUTE  0x4 /* send without using routing tables */
```

pour signifier de ne pas utiliser les tables de routage durant la transmission.



Si la valeur de retour est `-1`, une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

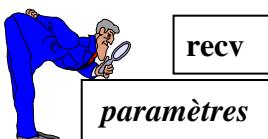
<i>valeur de errno</i>	<i>erreur</i>
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define EFAULT 14	/* Bad address */ : l'adresse de la suite de caractères est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)
#define EMSGSIZE 40	/* Message too long */ : très clair ...
#define EWOULDBLOCK 35 #define EAGAIN EWOULDBLOCK	/* Operation would block */ : il n'y a rien à envoyer et la socket n'est pas bloquante (ce qu'elle devrait être)
#define ENOTCONN 57	/* Socket is not connected */ : à votre avis ?
#define EOPNOTSUPP 45	/* Operation not supported on socket */ : MSG_OOB n'est pas utilisable – <i>on a utilisé un mauvais domaine ou un mauvais protocole</i>
#define EIO 5	/* I/O error */ : le temps limite a été dépassé
#define EPIPE 32	/* Broken pipe */ : il s'agit effectivement d'une situation analogue à celle d'un <i>pipe</i> sans lecteur – la connexion a été fermée ! Le processus qui a tenté d'écrire reçoit le signal SIGPIPE, qu'il peut capter (par défaut, il se termine).
#define EINTR 4	/* Interrupted system call */ : la fonction a été interrompue

## 11.2 La réception de caractères (recv)

C'est le rôle de la primitive de réaliser une **lecture destructive** des caractères reçus :

```
int recv    (<handle de la socket source – int>,
            <adresse où écrire la suite de caractères lus - void *>,
            <nombre maximum de caractères à lire – int>,
            <flag d'urgence et/ou de non destruction - int>);
```

La valeur renvoyée est le nombre de caractères lus, 0 si la connexion a été fermée ou –1 en cas d'erreur. A remarquer que, par défaut, recv est *bloquant* dans la situation d'un buffer vide avec une connexion non fermée.



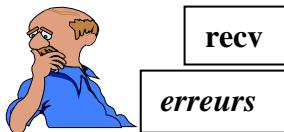
A nouveau, seul le dernier paramètre demande une explication. Il peut à nouveau valoir

```
#define      MSG_OOB      0x1      /* process out-of-band data */
```

pour lire une donnée urgente ou

```
#define      MSG_PEEK     0x2      /* peek at incoming message */
```

pour consulter les caractères lus sans les retirer du buffer.



Si la valeur de retour est  $-1$ , une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
<code>#define EBADF 9</code>	<code>/* Bad file number */</code> : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
<code>#define ENOTSOCK 38</code>	<code>/* Socket operation on non-socket */</code> : le descripteur n'est pas associé à une socket, mais à un fichier
<code>#define EFAULT 14</code>	<code>/* Bad address */</code> : l'adresse où placer la suite de caractères est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)
<code>#define EINVAL 22</code>	<code>/* Invalid argument */</code> : le nombre de caractères à lire est négatif ou l'option OOB a été spécifiée alors qu'il n'y a aucune donnée urgente.
<code>#define EWOULDBLOCK 35</code>	<code>/* Operation would block */</code> : il n'y a rien à lire et la socket n'est pas bloquante (ce qu'elle devrait être)
<code>#define EAGAIN EWOULDBLOCK</code>	
<code>#define ENOTCONN 57</code>	<code>/* Socket is not connected */</code> : à votre avis ?
<code>#define EINTR 4</code>	<code>/* Interrupted system call */</code> : la fonction a été interrompue

### Remarque

Dans le cas où l'on travaillerait en mode non bloquant (ce qui est possible par un paramétrage ad hoc de la socket – voir le chapitre consacré à ce sujet), on peut imaginer qu'une réception est en fait programmée comme une boucle qui fonctionne tant que l'appel de la fonction de réception renvoie `EWOULDBLOCK`. Lorsque cette boucle s'arrête, cela signifie que des données sont arrivées. Ce procédé de scrutation de la socket est appelé **polling**.

### **11.3 Le serveur et le client**

Nous pouvons à présent écrire intégralement un premier serveur "TCPm" (**Monoconnexion** ou **Minable**) ; il s'exécutera sur la machine UNIX **sunray2v440** :

```
TCPITER01D.C
/* TCPITER01D.C
 * serveur itératif mono-connexion *
 - Claude Vilvens -
 */

#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */
```

A diagram illustrating a network connection between two hosts. On the left, a computer monitor labeled 'Client Copernic' is connected by a line to a server tower labeled 'Serveur : Sunray'. The server tower has a small screen at the top.

```

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h>      /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h>   /* pour la conversion adresse reseau->format dot
                           ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h>   /* pour la conversion adresse reseau->format dot */

#define PORT 50000 /* Port d'ecoute de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocketEcoute, /* Handle de la socket d'écoute */
        hSocketService; /* Handle de la socket de service connectee au client */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
                           /* Structure de type sockaddr contenant les infos adresses - ici, cas de TCP */
    int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;

/* 1. Création de la socket */
    hSocketEcoute= socket(AF_INET, SOCK_STREAM, 0);
    if (hSocketEcoute== -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur local */
    if ( (infosHost = gethostbyname("sunray2v440"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
    printf("Adresse IP = %s\n",inet_ntoa(adresseIP));
                           /* Conversion de l'adresse contenue dans le structure in_addr
                           en une chaine comprehensible */
}

```

```

/* 3. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */
    adresseSocket.sin_port = htons(PORT);
    /* conversion numéro de port au format réseau */
    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocketEcoute, (struct sockaddr *)&adresseSocket,
              sizeof(struct sockaddr_in)) == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        exit(1);
    }
    else printf("Bind adresse et port socket OK\n");

/* 5. Mise à l'écoute d'une requête de connexion */
    if (listen(hSocketEcoute, SOMAXCONN) == -1)
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 6. Acceptation d'une connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
          accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) ) == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); exit(1);
    }
    else printf("Accept socket OK\n");

/* 7. Reception d'un message client */
    if ((nbreRecv = recv(hSocketService, msgClient, MAXSTRING, 0)) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        close(hSocketService); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recv socket OK\n");
    msgClient[nbreRecv]=0;
    printf("Message recu = %s\n", msgClient);

```

```

/* 8. Envoi de l'ACK du serveur au client */
    sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
    if (send(hSocketService, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        close(hSocketService); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

/* 9. Reception d'un second message client */
    if ((nbreRecv = recv(hSocketService, msgClient, MAXSTRING, 0)) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        close(hSocketService); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recv socket OK\n");
    msgClient[nbreRecv]=0;
    printf("Message recu = %s\n", msgClient);

/* 10. Envoi de l'ACK du serveur au client */
    sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
    if (send(hSocketService, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        close(hSocketService); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

/* 11. Fermeture des sockets */
    close(hSocketService); /* Fermeture de la socket */
    printf("Socket connectee au client fermee\n");
    close(hSocketEcoute); /* Fermeture de la socket */
    printf("Socket serveur fermee\n");

    return 0;
}

```

Le client, quant à lui, s'exécutera sur la machine UNIX **copernic** et aura la forme suivante :

### TCPCLI01.C

```

/* TCPCLI01.C
* client *
- Claude Vilvens -
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* conversions adresse reseau->format dot et local/ reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#define PORT 50000 /* Port d'ecoute de la socket serveur */
#define MAXSTRING 100 /* Longeur des messages */

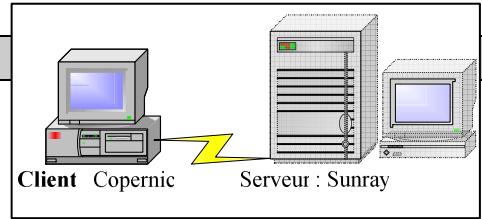
int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket; /* Structure de type sockaddr - ici, cas de TCP */
    unsigned int tailleSockaddr_in;
    int ret; /* valeur de retour */

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];

/* 1. Creation de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur distant */
    if ((infosHost = gethostbyname("sunray2v440"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host distant %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host distant OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
}

```



```

printf("Adresse IP = %s\n",inet_ntoa(adresseIP));

/* 3. Préparation de la structure sockaddr_in */
memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
adresseSocket.sin_family = AF_INET; /* Domaine */

adresseSocket.sin_port = htons(PORT);
/* conversion numéro de port au format réseau ← */
memcpy(&adresseSocket.sin_addr, infosHost->h_addr,infosHost->h_length);

/* 4. Tentative de connexion */
tailleSockaddr_in = sizeof(struct sockaddr_in);
if (( ret = connect(hSocket, (struct sockaddr *)&adresseSocket, tailleSockaddr_in) )
    == -1)
{
    printf("Erreur sur connect de la socket %d\n", errno);
    close(hSocket);
    exit(1);
}
else printf("Connect socket OK\n");

/* 5. Envoi d'un message client */
strcpy(msgClient, "Bonjour ! Nous nous connaissons ?");
if (send(hSocket, msgClient, MAXSTRING, 0) == -1) /* pas message urgent */
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");
printf("Message envoyé = %s\n", msgClient);

/* 6. Reception de l'ACK du serveur au client */
if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
{
    printf("Erreur sur le recv de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recv socket OK\n");
printf("Message reçu en ACK = %s\n", msgServeur);

/* 7. Envoi d'un deuxième message client */
printf("Message à envoyer : "); gets(msgClient);
if (send(hSocket, msgClient, MAXSTRING, 0) == -1) /* pas message urgent */
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}

```

```

else printf("Send socket OK\n");

printf("Message envoye = %s\n", msgClient);

/* 8. Reception de l'ACK du serveur au client */
if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
{
    printf("Erreur sur le recv de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recv socket OK\n");

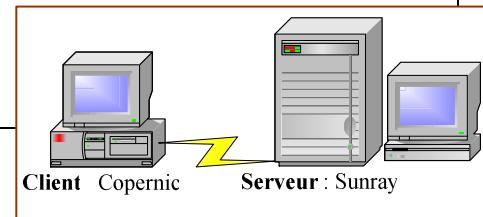
printf("Message recu en ACK = %s\n", msgServeur);

/* 9. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");

return 0;
}

```

L'exécution peut donner comme résultat :



serveur	client
sunray2v440.inpres.epl.prov-liege.be> <b>tcpiter01</b> Creation de la socket OK Acquisition infos host OK Adresse IP = 10.59.5.9 Bind adresse et port socket OK Listen socket OK Accept socket OK Recv socket OK Message recu = Bonjour ! Nous nous connaissons ? Send socket OK Recv socket 2 OK Message recu = J'habite encore chez mes parents ! Send socket 2 OK Socket connectee au client fermee Socket serveur fermee	copernic.inpres.epl.prov-liege.be> cli Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.5.9 Connect socket OK Send socket OK Message envoye = Bonjour ! Nous nous connaissons ? Recv socket OK Message recu en ACK = ACK pour votre message : <Bonjour ! Nous nous connaissons ?> Message a envoyer : <b>J'habite encore chez mes parents !</b> Send socket OK Message envoye = J'habite encore chez mes parents ! Recv socket OK Message recu en ACK = ACK pour votre message 2 : <J'habite encore chez mes parents !> Socket client fermee

Que se passe-t-il si le client ferme la connexion ? Le serveur reçoit des données erratiques ...

#### **11.4 Un dialogue prolongé**

Bien sûr, à ce stade, rien n'interdit d'imaginer un véritable dialogue entre le serveur et son client : une simple boucle suffit. Elle se termine, dans ce protocole d'une rare complexité, par l'envoie du message "END\_OF\_CONNEXION", défini par une constante dans un mini-header :

##### **TCPITER02.H**

```
/* TCPITER02.H
- Claude Vilvens -
*/
#ifndef TCPITER_H
#define TCPITER_H

#define PORT 50000 /* Port d'écoute de la socket serveur */
#define MAXSTRING 100 /* Longueur des messages */
#define EOC "END_OF_CONNEXION"

#endif
```

Le serveur sera simplement :

##### **TCPITER02.C**

```
/* TCPITER02.C
- Claude Vilvens -
*/
...
#include "tcpiter.h"

int main()
{
    ...
    int finConnexion;

    /* 1. Création de la socket */
    ...
    /* 2. Acquisition des informations sur l'ordinateur local */
    ...
    /* 3. Préparation de la structure sockaddr_in */
    ...
    /* 4. Le système prend connaissance de l'adresse et du port de la socket */
    ...
```

```

finConnexion = 0;
do
{
/* 7.Reception d'un message client */
    if ((nbreRecv =
        recv(hSocketService, msgClient, MAXSTRING, 0)) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketService); /* Fermeture de la socket */
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    printf("Recv socket OK\n");
    msgClient[nbreRecv]=0;
    printf("Message recu = %s\n", msgClient);
    if (strcmp(msgClient, EOC)==0)
    {
        finConnexion=1;
        printf("*** Le client demande la fin de la connexion ***\n");
    }
}

/* 8. Envoi de l'ACK du serveur au client */
if (!finConnexion)
{
    memset(msgServeur, 0, MAXSTRING);
    sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
    if (send(hSocketService, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketService); /* Fermeture de la socket */
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");
}
while (!finConnexion);

/* 9. Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */
printf("Socket connectee au client fermee\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");

return 0;
}

```

tandis que le client fonctionnera selon une boucle similaire :

## TCPCLI02.C

```

/* TCPCLI02.C
* client *
- Claude Vilvens -
*/
...
#include "tcpiter.h"

int main()
{
    ...
    int cpt=0;
/* 1. Création de la socket */
    ...
/* 2. Acquisition des informations sur l'ordinateur distant */
    ...
/* 3. Préparation de la structure sockaddr_in */
    ...
/* 4. Tentative de connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if (( ret = connect(hSocket, (struct sockaddr *)&adresseSocket, tailleSockaddr_in) ) == -1)
    {
        printf("Erreur sur connect de la socket %d\n", errno);
        switch(errno)
        {
            case EBADF : printf("EBADF - hSocketEcoute n'existe pas\n"); break;
            case ENOTSOCK :
                printf("ENOTSOCK - hSocketEcoute identifie un fichier\n");break;
            case EAFNOSUPPORT :
                printf("EAFNOSUPPORT - adresse ne correspond pas familie\n");break;
            case EISCONN : printf("EISCONN - socket deja connectee\n");break;
            case ECONNREFUSED :
                printf("ECONNREFUSED - connexion refusee par le serveur\n");break;
            case ETIMEDOUT : printf("ETIMEDOUT - time out sur connexion \n");break;
            case ENETUNREACH : printf("ENETUNREACH - cible hors d'atteinte\n");break;
            case EINTR : printf("EINTR - interruption par signal\n");break;
            default : printf("Erreur inconnue ?\n");
        }
        close(hSocket);
        exit(1);
    }
    else printf("Connect socket OK\n");
}

```

```

/* 5. Envoi d'un message client */
do
{
    printf("Message num %d a envoyer : ", cpt + 1); gets(msgClient);
    if (send(hSocket, msgClient, MAXSTRING, 0) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

    printf("Message envoye = %s\n", msgClient);
    if (strcmp(msgClient, EOC))
    {

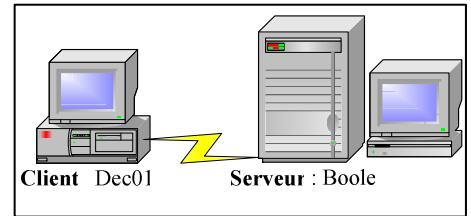
/* 6. Reception de l'ACK du serveur au client */
        if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
        {
            printf("Erreur sur le recv de la socket %d\n", errno);
            close(hSocket); /* Fermeture de la socket */
            exit(1);
        }
        else printf("Recv socket OK\n");

        printf("Message recu en ACK = %s\n", msgServeur);
        cpt++;
    }
}
while (strcmp(msgClient, EOC));

/* 7. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");
printf("%d messages envoyees !", cpt);
return 0;
}

```

Ce qui donne (avec cette fois d'autres machines : le serveur sur boole et le client sur dec01) :



serveur	client
<pre> boole&gt; tcpciter02 Creation de la socket OK Acquisition infos host OK Adresse IP = 10.59.4.1 Bind adresse et port socket OK Listen socket OK Accept socket OK Recv socket OK ← Message recu = Bonjour vous ! Send socket OK → Recv socket OK ← Message recu = Vous habitez chez vos parents ? Send socket OK → Recv socket OK ← Message recu = Et si nous allions dîner ? Send socket OK → Recv socket OK ← Message recu = On va chez toi ou chez moi ? Send socket OK → Recv socket OK ← Message recu = END_OF_CONNEXION *** Le client demande la fin de la connexion *** Socket connectee au client fermee Socket serveur fermee </pre>	<pre> % cli Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.4.1 Connect socket OK Message num 1 a envoyer : <b>Bonjour vous !</b> Send socket OK Message envoyee = Bonjour vous ! Recv socket OK Message recu en ACK = ACK pour votre message : &lt;Bonjour vous !&gt; Message num 2 a envoyer : <b>Vous habitez chez vos parents ?</b> Send socket OK Message envoyee = Vous habitez chez vos parents ? Recv socket OK Message recu en ACK = ACK pour votre message : &lt;Vous habitez chez vos parents ?&gt; Message num 3 a envoyer : <b>Et si nous allions dîner ?</b> Send socket OK Message envoyee = Et si nous allions dîner ? Recv socket OK Message recu en ACK = ACK pour votre message : &lt;Et si nous allions dîner ?&gt; Message num 4 a envoyer : On va chez toi ou chez moi ? Send socket OK Message envoyee = <b>On va chez toi ou chez moi ?</b> Recv socket OK Message recu en ACK = ACK pour votre message : &lt;On va chez toi ou chez moi ?&gt; Message num 5 a envoyer : <b>END_OF_CONNEXION</b> Send socket OK Message envoyee = <b>END_OF_CONNEXION</b> Socket client fermee 4 messages envoyees !% </pre>

## 12. Si le client est interrompu

Il y a des grossiers partout, même dans les clients ... Que se passera-t-il si le client est interrompu violemment par un CTRL-C ? Le serveur verra arriver des informations erratiques, sans signification ... Il est évidemment plus raisonnable d'orner le client d'un handler de *traitement du signal SIGINT* : ce handler enverra poliment un message de fin au serveur, lui signalant ainsi la fin de la connexion, avant de laisser ce client se terminer.

D'autre part, à titre d'exemple et de manière anticipée, le client spécifiera le serveur qu'il désire atteindre en utilisant l'adresse IP "à la dure", au lieu de la demander au DNS avec la fonction `gethostbyname()`. Cette adresse se présente ici sous forme d'une chaîne de caractère (la constante ADRESSE) et c'est au moyen de la fonction de transformation

```
int inet_addr (<adresse sous forme pointée - char *>);
/* adresse pointée -> adresse réseau */
```

que l'on obtient l'adresse réseau qu'aurait fournie le DNS<sup>1</sup>.

Tout ceci donne quelque chose de ce genre :

### TCPCLI01INT.C

```
/* TCPCLI01INT.C
- Claude Vilvens -
*/
#include <stdio.h>
...
#include <signal.h>

#define ADRESSE "10.59.4.5" /* Adresse IP du serveur */
#define PORT 50000 /* Port d'écoute de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

void handlerSigint(int sig);

int hSocket; /* Handle de la socket */

int main()
{
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format réseau */
    struct sockaddr_in adresseSocket;
    int tailleSockaddr_in;
    int ret; /* valeur de retour */

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];

    struct sigaction act;
```

---

<sup>1</sup> nous reparlerons des divers formats d'adresses dans le chapitre consacré spécifiquement à celles-ci.

```

/* 1. Création de la socket */
...
printf("Adresse IP du serveur = %s\n",ADRESSE);

/* 2. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */
    adresseSocket.sin_port = htons(PORT); /* conversion numéro port au format réseau */

    adresseSocket.sin_addr.s_addr = inet_addr(ADRESSE);

/* 3. Tentative de connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if (( ret = connect(hSocket, (struct sockaddr *)&adresseSocket,
                        tailleSockaddr_in) )
        == -1) ...

/* 4. Armement sur le signal SIGINT */
    act.sa_handler = handlerSigint;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGINT, &act, 0);

/* 5. Envoi d'un message client */
...
/* 6. Reception de l'ACK du serveur au client */
...
/* 7. Envoi d'un deuxième message client */
...
/* 8. Reception de l'ACK du serveur au client */
...
/* 9. Fermeture de la socket */
    close(hSocket); /* Fermeture de la socket */
    printf("Socket client fermee\n");

    return 0;
}

/* -----
void handlerSigint(int sig)
{
    char msgClient[MAXSTRING];
    puts("Passage dans Sigint");
    strcpy(msgClient,"CLIENT INTERRUPTED!");

    if (send(hSocket, msgClient, MAXSTRING, 0) == -1) /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); exit(1);
    }
}

```

```

else printf("Send socket OK\n");

printf("Message envoyé = %s\n", msgClient);
exit(0);
}

```

### 13. Un client Telnet

Le protocole applicatif **TELNET** (TERminaL NETwork) est un protocole standard qui permet de se connecter à un système distant en se comportant comme si le terminal local était en fait un terminal quelconque de ce système distant. On parle encore de **terminal virtuel**.

Normalement, la socket distante dédiée à Telnet travaille sur le port 23; c'est sur ce port qu'un processus démon appelé finement "telnetd" est en écoute. Cependant, rien n'interdit d'utiliser un autre port, la gestion du protocole de communication étant alors laissé à l'appréciation de l'utilisateur qui s'est ainsi connecté directement au serveur. La commande de connexion telnet possède diverses options dont nous ne retiendrons que :

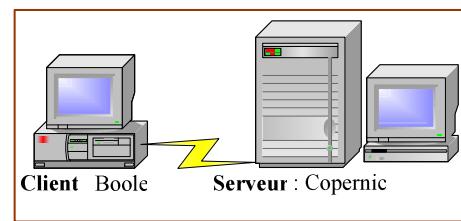
```
telnet <nom machine distante | adresse machine distante> [<port>]
```

le port par défaut étant donc 23. Dans notre cas, nous pouvons contacter depuis boole notre serveur installé sur copernic par :

```

boole.inpres.epl.prov-liege.be> telnet sunray2v440 50000
Trying 10.59.5.9...
Connected to copernic.
Escape character is '^].
Here I am !!!
ACK pour votre message : <Here I am !!!
>
Do you think I'm sexy (as Rod Stewart said ...)
ACK pour votre message : <Do you think I'm sexy (as Rod Stewart said ...)
>

```



Du côté du serveur, la réaction est :

```

copernic.inpres.epl.prov-liege.be> s
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.5.9
Bind adresse et port socket OK
Listen socket OK
Accept socket OK
Nombre de caractères reçus = 15
Message reçu = Here I am !!!

Longueur du message reçu = 15
Envoi de l'ACK au client
Msg envoyé = ACK pour votre message : <Here I am !!!
>

```

Send socket OK

Recv socket OK

Nombre de caractères recus = 49

Message reçu = Do you think I'm sexy (as Rod Stewart said ...)

Longueur du message reçu = 49

Envoi de l'ACK au client

Msg envoyé = ACK pour votre message : <Do you think I'm sexy (as Rod Stewart sa)  
>

Send socket OK

Si on arrête le serveur, on obtient du côté du client :

| Connection closed by foreign host.

## 14. Un client Java

### 14.1 Faire redescendre Java au bas niveau

On s'en doute, c'est là la beauté de l'art, le client n'est pas forcément écrit en C ! L'idée vient donc d'écrire, par exemple, un client Java (ou C#) pour notre serveur. Comme exposé en détail ailleurs<sup>1</sup>, il faut tenir compte du fait que Java ne voit pas les bytes et les caractères sous le même angle : les caractères dépendent d'une plate-forme d'encodage et peuvent occuper 2 bytes (Unicode) ou un nombre variable de 1, 2 ou 3 bytes (UTF).

Pour illustrer cela, considérons un **serveur Java élémentaire** (il reçoit des chaînes de caractères et renvoie un ACK applicatif – "END\_OF\_CONNEXION" signifiera la déconnexion du client) qui va être interrogé par un **client Telnet** (qui est bien sûr seulement capable de manipuler des bytes). Le serveur a été développé en JDK 1.4 avec Sun ONE Studio et ressemble à ceci :



#### FenServer.java

```
import java.net.*;
import java.io.*;

public class FenServer extends java.awt.Frame
{
    ServerSocket SSocket;
    Socket CSocket;
```

<sup>1</sup> voir "Langage Java (II) : Programmation avancée des applications classiques" du même auteur ... (pub ;-))

```

public FenServer() { initComponents(); }

private void initComponents()
{
    ZTServeur = new java.awt.Label(); panel1 = new java.awt.Panel();
    ...
    // initailisations du GUI – nenus intéresse guère ici
    ...
}

private void BArretActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        CSocket.close();SSocket.close();
    }
    catch (IOException e) { System.err.println("Erreur ! ? [" + e + "]"); }

    ZTServeur.setText("Serveur arrêté"); System.out.println("Serveur arrêté");
}

private void BEnMarcheActionPerformed(java.awt.event.ActionEvent evt)
{
    SSocket = null;
    try
    {
        SSocket = new ServerSocket(6000);
    }
    catch (IOException e)
    { System.err.println("Erreur de port d'écoute ! ? [" + e + "]"); System.exit(1); }

    CSocket = null;
    DataInputStream dis = null; DataOutputStream dos = null;
    System.out.println("Serveur en attente"); ZTServeur.setText("Serveur en marche");
    try
    {
        CSocket = SSocket.accept();
        dis = new DataInputStream(new BufferedInputStream(CSocket.getInputStream()));
        dos = new DataOutputStream(new BufferedOutputStream(CSocket.getOutputStream()));

        }
        catch (IOException e)
        { System.err.println("Erreur d'accept ! ? [" + e + "]"); System.exit(1); }

    System.out.println("Une connexion client acceptée");
    InetSocketAddress ad = (InetSocketAddress)CSocket.getRemoteSocketAddress();
    String idClient = ad.getHostName() + " (" + CSocket.getInetAddress().getHostAddress() +
    ")";
    ZTClient.setText(idClient);
    System.out.println("Connexion provenant de : " + idClient);
}

```

```

StringBuffer message = new StringBuffer();
boolean fini = false;
byte b=0; int cpt = 0;
do
{
    message.setLength(0);
    try
    {
        while ( (b=dis.readByte())!= (byte)'\n' )
        {
            if (b!='\n') message.append((char) b);
            System.out.println("byte reçu = "+b);
        }
    }
    catch (IOException e)
    { System.out.println("Erreur de lecture = " + e.getMessage()); }

    System.out.println("dernier byte reçu = "+b); cpt++;
    System.out.println (cpt + ". Message reçu = " + message.toString().trim());
    ZTMessage.setText(message.toString().trim());
    fini = "END_OF_CONNEXION".equals(message.toString().trim()) || cpt>5;
    System.out.println("fini = " + fini);
    String reponse = "ACK applicatif> " + message + "\n";
    try
    {
        if (!fini) dos.write(reponse.getBytes()); dos.flush();
    }
    catch (IOException e)
    { System.out.println("Erreur de lecture = " + e.getMessage()); }
}
while (!fini);

try
{
    dis.close();dos.close();
}
catch (IOException e)
{ System.out.println("Erreur de fermeture = " + e.getMessage()); }
}

private void exitForm(java.awt.event.WindowEvent evt) { System.exit(0); }

public static void main(String args[])
{
    new FenServer().show();
}

private java.awt.Label ZTMessage;
private java.awt.Label ZTClient;

```

```
private java.awt.Panel panel1;
private java.awt.Label label2;
private java.awt.Button BEnMarche;
private java.awt.Label ZTServeur;
private java.awt.Button BArret;
private java.awt.Label label1;
}
```

On se souviendra simplement que :

- ◆ les classes Socket et ServerSocket encapsulent les mécanismes fondamentaux de connexion;
- ◆ la lecture des bytes se fait byte par byte au moyen de la méthode  
`public final byte readByte() throws IOException`  
de la classe DataInputStream;
- ◆ les bytes utiles sont extraits d'un String au moyen de la méthode  
`public byte[] getBytes();`
- ◆ le message d'ACK reçu du serveur est expurgé des blancs inutiles qui sont sur le flux au moyen de la méthode `trim()`.

Un exemple de communication ressemble à ceci :

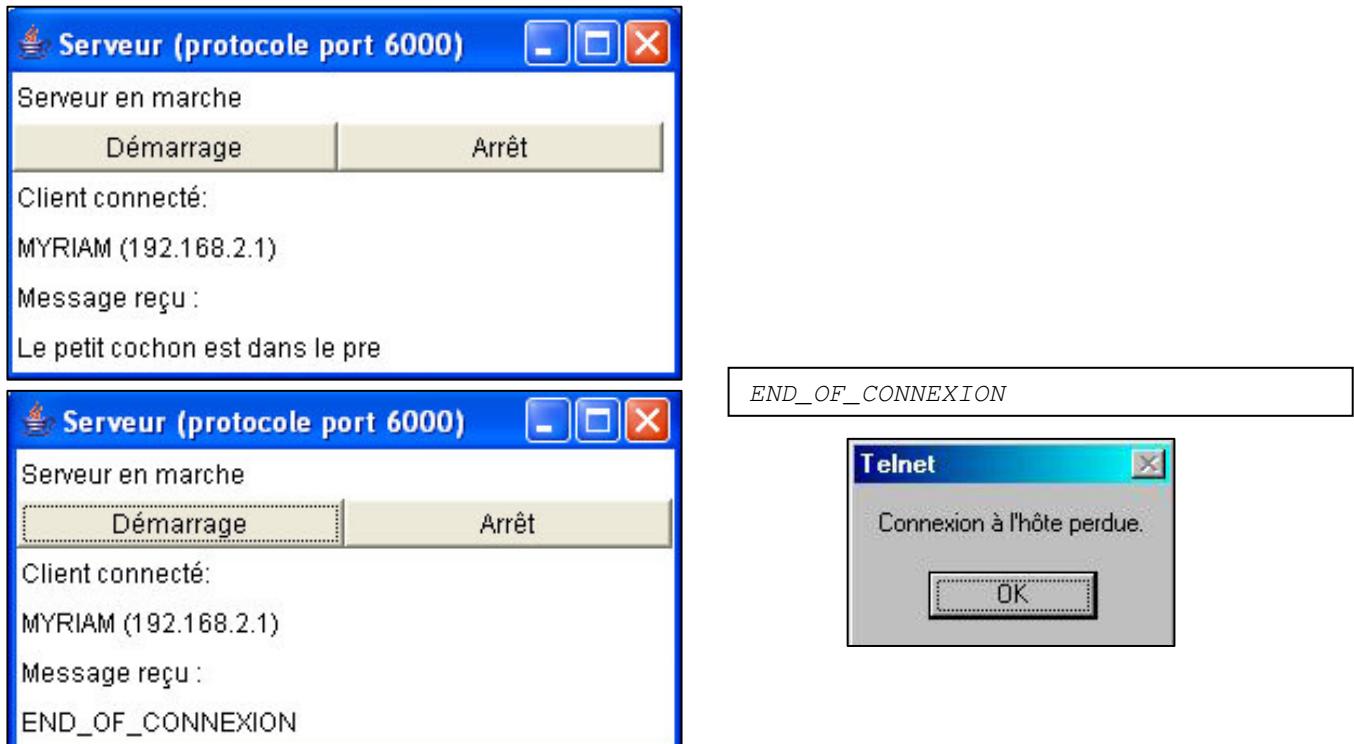
**serveur** (machine claude – Windows XP)



**client** (machine myriam – Windows 98)



*Le petit cochon est dans le pré  
ACK applicatif> Le petit cochon est  
dans le pré*



## 14.2 Un client Java pour le serveur C

Voici à présent un client Java écrit de manière similaire au serveur Java décrit ci-dessus et qui va remplacer le client TCPCLI02.C de notre serveur écrit en C (cette fois, il fonctionne sur la machine boole).

### clientJava.java

```

import java.io.*;
import java.net.*;

public class Java
{
    public final static int portEcouteServeur = 50000;
    public final static String nomServeur = "boole";

    public static void main(String args[])
    {
        Socket cliSock = null;      /* Initialisations indispensables */
        DataInputStream dis=null; DataOutputStream dos=null;
        StringBuffer msgClient = new StringBuffer();
        boolean fini = false;

        try
        {
            cliSock = new Socket(nomServeur, portEcouteServeur);
            System.out.println(cliSock.getInetAddress().toString());
            dis = new DataInputStream(cliSock.getInputStream());
            dos = new DataOutputStream(cliSock.getOutputStream());
        }
    }
}

```

```

    catch (UnknownHostException e)
    {
        System.err.println("Erreur ! Host non trouvé [" + e + "]");
    }
    catch (IOException e)
    {
        System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
    }
    if (cliSock==null || dis==null || dos==null) System.exit(1);

    try
    {
        int cpt=0, c;
        byte b;

        do
        {
            cpt++;
            System.out.println("Message numero " + cpt + " à envoyer au serveur : ");
            int cptChar = 0;
            while((c = System.in.read())!='\n')
                if (c != '\r')
                {
                    msgClient.append((char)c); cptChar++;
                }
            msgClient.setLength(cptChar);

            dos.write(msgClient.toString().getBytes());
            dos.flush();
            if (msgClient.toString().compareTo("END_OF_CONNEXION")==0)
            {
                System.out.println("Fin de connexion demandée !");
                fini = true;
            }
            else
            {
                msgClient.setLength(0);
                System.out.println("Attente de la réponse du serveur");
                StringBuffer msgServeur = new StringBuffer();
                int cptCharServeur = 0;
                while((b = dis.readByte())!=((byte)\n))
                    msgServeur.append((char)b);
                System.out.println("Reçu du serveur : " + msgServeur.toString().trim());
            }
        }
        while (!fini);
        dos.close(); dis.close();
        cliSock.close();
        System.out.println("Client déconnecté");
    }
}

```

```

        catch (UnknownHostException e)
        {
            System.err.println("Erreur ! Host non trouvé [" + e + "]");
        }
        catch (IOException e)
        {
            System.err.println("Erreur ! Pas de connexion ? [" + e + "]");
        }
    }
}

```

On remarquera qu'au sein du serveur, le message d'ACK devra être complété d'un '\n' pour permettre au client de détecter la fin du message.

Si le client s'exécute sur la console Java d'une machine Windows :

boole/10.59.4.1

Message numero **1** à envoyer au serveur :

*Bonjour Belle enfant !*

Attente de la réponse du serveur

Reçu du serveur : ACK pour votre message : <Bonjour Belle enfant !>

Message numero **2** à envoyer au serveur :

*C'est à vous ces beaux yeux ?*

Attente de la réponse du serveur

Reçu du serveur : ACK pour votre message : <C'est à vous ces beaux yeux ?>

Message numero **3** à envoyer au serveur :

*Ne me trouvez-vous pas merveilleux ?*

Attente de la réponse du serveur

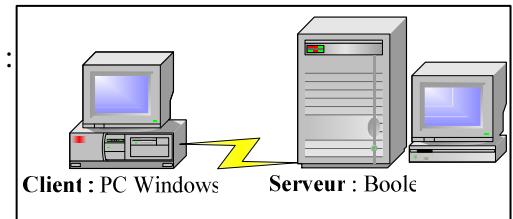
Reçu du serveur : ACK pour votre message : <Ne me trouvez-vous pas merveilleux ?>

Message numero **4** à envoyer au serveur :

***END\_OF\_CONNEXION***

Fin de connexion demandée !

Client déconnecté



le serveur (sur boole) répondra :

```

boole> tcpter02
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Listen socket OK
Accept socket OK
Recv socket OK
Message recu = Bonjour Belle enfant !
Msg envoye = ACK pour votre message : <Bonjour Belle enfant !>
Send socket OK
Recv socket OK
Message recu = C'est à vous ces beaux yeux ?
Longueur du message regu = 29

```

Msg envoyé = ACK pour votre message : <C'est à vous ces beaux yeux ?>

Send socket OK

Recv socket OK

Message recu = ***Ne me trouvez-vous pas merveilleux ?***

Msg envoyé = ACK pour votre message : <Ne me trouvez-vous pas merveilleux ?>

Send socket OK

Recv socket OK

Message recu = ***END\_OF\_CONNEXION***

\*\*\* Le client demande la fin de la connexion \*\*\*

Socket connectee au client fermee

Socket serveur fermee

boole>

Evidemment, notre serveur est toujours monoconnexion : il change effectivement de socket pour le service rendu par le serveur mais ne se remet à l'écoute que lorsque ce service est terminé ! La solution vient immédiatement à l'esprit : créons un sous-processus (genre fork() ) qui prendra en charge la transaction, laissant ainsi le processus père se remettre à l'écoute sur la socket d'écoute. On peut procéder ainsi, mais un mécanisme similaire, celui des threads, nous permettra de réaliser cet objectif à moindre frais : le coût système d'un sous-processus n'est pas négligeable, ce qui est un peu gênant dans un contexte réseau où tout temps perdu est déplorable ...

Il existe aussi une solution intermédiaire qui consiste à utiliser la primitive select(). Nous allons en parler.

Mais, avant d'attaquer ces nouveaux objectifs, il reste encore une chose à préciser .

## **15. Le nombre de caractères lus**

Il ne faut jamais perdre de vue que

le protocole TCP est susceptible de **découper** le groupe de caractères envoyés en blocs plus petits.

Dans ces conditions, si l'on peut être assuré du respect de l'ordre des caractères par TCP, **rien**, par contre, **ne permet d'affirmer en toute généralité que tous les caractères transmis seront lus en une seule opération** ! Il importe donc nos programmes lisent les caractères reçus sur une socket jusqu'à ce que le nombre de caractères reçus soit suffisant.

Reprenons les programmes serveur (fonctionnant sur une machine boole) et client (fonctionnant sur une machine dec01) développés ci-dessus. Ils vont tenter de s'échanger un message de 5000 bytes. Or, la taille maximale d'un segment (le **MTU - Maximum Transmission Unit**) est d'un certain nombre de bytes, variable selon l'environnement sur lequel on travaille (usuellement, on trouve un MTU de 1500 bytes, mais on peut rencontrer d'autres valeurs comme 1460, 4096, 8232, etc)

Le serveur va recevoir un message et le renvoyer au client tel quel.

## TCPITER01BIS.C

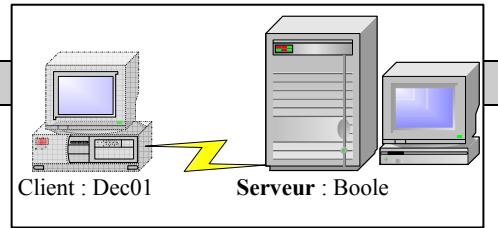
```

/* TCPITER01BIS.C
- Claude Vilvens -
*/
...
#define PORT 50000 /* Port d'ecoute de la socket serveur */
#define MAXSTRING 100 /* Longeur des messages */
#define LONG_GROS_MSG 5000 /* Taille gros message */

int main()
{
    ...
    /* 7. Reception d'un message client */
    /* tailleO=sizeof(int);
       if(getsockopt(hSocketConnectee, IPPROTO_TCP, TCP_MAXSEG,
                     &tailleS, &tailleO) == -1)
    {
        printf("Erreur sur le getsockopt de la socket %d\n", errno);
        exit(1);
    }
    else
    {
        printf("getsockopt OK\n");
        printf("Taille maximale d'un segment = %d\n", tailleS);
    }
*/
    if ((nbreRecv = recv(hSocketConnectee, msgClient, LONG_GROS_MSG, 0)) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketConnectee); /* Fermeture de la socket */
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recv socket OK");
    msgClient[nbreRecv]=0;
    printf("Longueur du message recu %d\n", nbreRecv);

    /* 8. Envoi de l'ACK du serveur au client */
    if (send(hSocketConnectee, msgClient,nbreRecv, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketConnectee); /* Fermeture de la socket */
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK");
}

```



```
/* 9. Fermeture des sockets */
...
return 0;
}
```

Remarquons, en passant, la fonction **getsockopt()** qui permet de connaître la limite de la longueur des messages – nous en reparlerons dans le chapitre consacré à la paramétrisation des sockets. Pour le client, rien de spécial :

### TCPCLI01BIS.C

```
/* TCPCLI01BIS.C
- Claude Vilvens -
*/
...

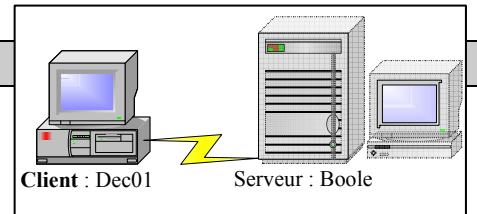
#define PORT 50000 /* Port d'écoute de la socket serveur */
#define MAXSTRING 100 /* Longeur des messages */
#define LONG_GROS_MSG 5000 /* Longeur d'un gros message */

int main()
{
    ...
    char grosMsg[LONG_GROS_MSG], msgServeur[LONG_GROS_MSG];
    ...

/* 5. Envoi d'un message client */
    for (i=0; i<LONG_GROS_MSG-1; i++) grosMsg[i]=i%2==0?'A':'B';
    grosMsg[LONG_GROS_MSG]=0;
    if ((nbreEnv=send(hSocket, grosMsg, LONG_GROS_MSG, 0)) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        printf("Send socket OK\n");
        printf("Nbre de bytes envoyés = %d\n", nbreEnv);
    }
    ...

/* 6. Reception de l'ACK du serveur au client */
    if ((nbreRecv = recv(hSocket, grosMsg, LONG_GROS_MSG, 0)) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocket);
        exit(1);
    }
    else
        printf("Recv socket OK\n");

    printf("Message reçu en ACK = %s\n", grosMsg);
    printf("Nombre de caractères reçus = %d\n", nbreRecv);
```



```
/* 7. Fermeture de la socket */
    close(hSocket); /* Fermeture de la socket */
    printf("Socket client fermee\n");

    return 0;
}
```

Cela donne côté serveur :

```
boole> tcriter01bis  
Creation de la socket OK  
Acquisition infos host OK  
Adresse IP = 10.59.4.1  
Bind adresse et port socket OK  
Listen socket OK  
Accept socket OK  
Recv socket OK
```

### 15.1.1 Longueur du message recu 1460

Send socket OK  
Socket connectee au client fermee  
Socket serveur fermee

et côté client :

```
% cli  
Creation de la socket OK  
Acquisition infos host distant OK  
Adresse IP = 10.59.4.1  
Connect socket OK  
Send socket OK
```

### 15.1.2 Nbre de bytes envoyés = 5000

Il faut donc impérativement **programmer une boucle de réception du message qui assure que le nombre de caractères attendus a bien été reçu**; les différents morceaux du message complet seront mis bout à bout pour le reconstituer.

L'exemple suivant tient compte de ce problème. De plus, au lieu de se contenter d'échanger des chaînes de caractères, le client va envoyer au serveur une requête plus élaborée sous forme de structure. Si le contexte est celui d'un client (au sens commercial du terme, du

---

genre les 3 Helvètes<sup>1</sup>) qui souhaite commander un article vu dans le catalogue, le message envoyé au serveur sera une structure du type "client", définie dans un header :

### **TCPITER03.H**

```
/* TCPITER03.H
- Claude Vilvens -
*/
#ifndef TCPITER_H
#define TCPITER_H

#define EOC "END_OF_CONNEXION"

#define PORT 50000 /* Port d'écoute de la socket serveur */

struct client
{
    char numClient[20];
    char nom[30];
    char dateDernierAchat[11];
    char numArticle[15];
    int montant;
    char coeffReduction[10];
    char fourni;
};

#define LONG_STRUCT_CLI sizeof(struct client) /* Longueur des messages */
#define LONG_MSG_SERV 100

#endif
```

Le serveur se contentera, pour l'instant, de demander la confirmation (ou la modification) du coefficient de réduction et d'envoyer un accusé de réception sous forme d'une chaîne de caractères, dont la longueur maximale est définie dans le même header. Le serveur s'écrira :

### **TCPITER03.C**

```
/* TCPITER03.C
- Claude Vilvens -
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>

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

---

<sup>1</sup> pour faire plaisir à une collègue ;-) ...

```
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#include "tcpiter03.h"

void afficheRequete(struct client *c);

int main()
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService; /* Handle de la socket de service connectee au client */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    unsigned int tailleSockaddr_in;

    char msgServeur [LONG_MSG_SERV];
    struct client *msgClient = (struct client *)malloc(sizeof(struct client));
    int tailleMsgRecu, nbreBytesRecus;
    int i;
    char buf[100];

/* 1. Creation de la socket */
    hSocketEcoute = socket(AF_INET,SOCK_STREAM,0);
    if (hSocketEcoute == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur local */
    if ( (infosHost = gethostbyname("boole"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
    printf("Adresse IP = %s\n",inet_ntoa(adresseIP));
    /* Conversion de l'adresse contenue dans le structure in_addr
       en une chaine comprehensible */
}
```

```

/* 3. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */
    adresseSocket.sin_port = htons(PORT);
    /* conversion numéro de port au format réseau ← */
    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocketEcoute, (struct sockaddr *)&adresseSocket,
              sizeof(struct sockaddr_in)) == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        exit(1);
    }
    else printf("Bind adresse et port socket OK\n");

do
{
/* 5. Mise à l'écoute d'une requête de connexion */
    if (listen(hSocketEcoute, SOMAXCONN) == -1) /* Constante définie dans
                                                sys/socket.h [de 5 à 8 selon les systèmes] */
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 6. Acceptation d'une connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
          accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");

/* 7. Reception d'un message client */
    tailleMsgRecu = 0;

```

```

do
{
    puts("Passage boucle de reception");
    if ( (nbreBytesRecus = recv(hSocketService, ((char*)msgClient) + tailleMsgRecu,
        LONG_STRUCT_CLI-tailleMsgRecu, 0))
        == -1)           /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketService); /* Fermeture de la socket */
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        printf("Taile msg recu = %d et taille attendue = %d\n",
            tailleMsgRecu, LONG_STRUCT_CLI);
        tailleMsgRecu += nbreBytesRecus;
    }
    printf("Taill msg = %d et nbreBytes = %d \n", tailleMsgRecu, nbreBytesRecus);
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
       tailleMsgRecu <LONG_STRUCT_CLI );
printf("Recv socket OK\n");

if (strcmp(msgClient->nom, EOC))
    printf("Demande recue pour le client = %s\n", msgClient->nom);
else break;

/* 8. Envoi de l'ACK du serveur au client */
sprintf(msgServeur,"Demande recue du client %s !!!", msgClient->nom);
printf("--- Coefficient de reduction = %s\n", msgClient->coeffReduction);
printf("Nouveau coeff : ");gets(msgClient->coeffReduction);
afficheRequete(msgClient);
if (send(hSocketService, msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");
}

while(1);

/* 9. Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */
printf("Socket connectee au client fermee\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");

```

```

        return 0;
}

/* ----- */

void afficheRequete(struct client *c)
{
    printf("**** Requete d'un client ****\n");
    printf("Numero de client : %s\n", c->numClient);
    printf("Nom = %s\n", c->nom);
    printf("Date du dernier achat : %s\n", c->dateDernierAchat);
    printf("Numero de l'article demande : %s\n", c->numArticle);
    printf(" et son prix : %d\n", c->montant);
    printf("Coefficient de reduction = %s\n", c->coeffReduction);
    printf("Fourni ? = %d\n", c->fourni);
}

```

Le client, de son côté, s'écrit plus simplement :

### **TCPCLI03.C**

```

/* TCPCLI03.C
- Claude Vilvens -
*/

#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#include "tcpiter03.h"

void analyseErreur(int numErreur);

struct client msgClient;

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */

```

```

struct in_addr adresseIP; /* Adresse Internet au format reseau */
struct sockaddr_in adresseSocket;
    /* Structure de type sockaddr contenant les infos adresses - ici, cas de TCP */
unsigned int tailleSockaddr_in;
int ret; /* valeur de retour */

char msgServeur[LONG_MSG_SERV], buf[100];

printf("?? Taille d'un client = %d\n", sizeof(struct client));

/* 1. Création de la socket */
hSocket = socket(AF_INET, SOCK_STREAM, 0);
if (hSocket == -1)
{
    printf("Erreur de creation de la socket %d\n", errno);
    exit(1);
}
else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur distant */
...
/* 3. Préparation de la structure sockaddr_in */
...
/* 4. Tentative de connexion */
...
/* 5. Envoi d'un message client */
printf("Nom du client (END_OF_CONNEXION) : "); gets(msgClient.nom);
if (strcmp(msgClient.nom, EOC))
{
    printf("Numero client : "); gets(msgClient.numClient);
    printf("Date du dernier achat : "); gets(msgClient.dateDernierAchat);
    printf("Numero d'article : "); gets(msgClient.numArticle);
    printf("Prix : "); gets(buf); msgClient.montant = atoi(buf);
    printf("taille de montant = %d\n", sizeof (msgClient.montant));
    printf("Coefficient de reduction : "); gets (msgClient.coeffReduction);
    msgClient.fourni=0;
}
if (send(hSocket, (char*)&msgClient, LONG_STRUCT_CLI, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");

printf("Demande envoyee pour le client = %s\n", msgClient.nom);

/* 6. Reception de l'ACK du serveur au client */
if (strcmp(msgClient.nom, EOC))
{
    if (recv(hSocket, msgServeur, LONG_MSG_SERV, 0) == -1)

```

```

    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); exit(1);
    }
    else
    {
        printf("Recv socket OK\n");
        printf("Message recu en ACK = %s\n", msgServeur);
    }
}

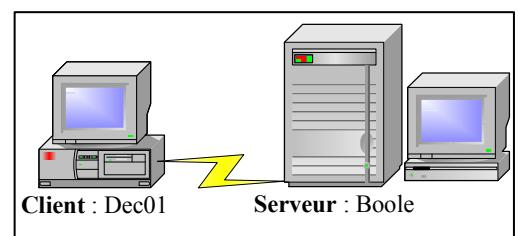
/* 7. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");

return 0;
}
/* ----- */
void analyseErreur(int numErreur)
{
    switch(numErreur)
    {
        case EBADF : printf("EBADF - hsocket n'existe pas\n"); break;
        case ENOTSOCK :
            printf("ENOTSOCK - hsocket identifie un fichier\n"); break;
        case EAFNOSUPPORT :
            printf("EAFNOSUPPORT - adresse ne correspond pas famille\n"); break;
        case EISCONN : printf("EISCONN - socket deja connectee\n"); break;
        case ECONNREFUSED :
            printf("ECONNREFUSED - connexion refusee par le serveur\n"); break;
        case ETIMEDOUT :
            printf("ETIMEDOUT - time out sur connexion \n"); break;
        case ENETUNREACH :
            printf("ENETUNREACH - cible hors d'atteinte\n"); break;
        case EINTR : printf("EINTR - interruption par signal\n"); break;
        default : printf("Erreur inconnue ?\n");
    }
}
}

```

Un exemple d'exécution d'un client (sur dec01) donne :

```
% cli
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.1
Connect socket OK
Nom du client (END_OF_CONNEXION) : Albert
Numero client : MR005
Date du dernier achat : 12/4/1998
Numero d'article : MOUS234
```



```
Prix : 2390
Coefficient de reduction : 3.98
Send socket OK
Demande envoyee pour le client = Albert
Recv socket OK
Message recu en ACK = Demande recue du client Albert !!!
Socket client fermee
%
```

Du côté du serveur (sur Boole), cela donne :

```
boole> tcriter03
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Listen socket OK
Accept socket OK
Passage boucle de reception
Taile msg recu = 0 et taille atendue = 92
Taill msg = 92 et nbreBytes = 92
Recv socket OK
Demande recue pour le client = Albert
--- Coefficient de reduction = 3.98
15.1.2.1 Nouveau coeff : 4.25
*** Requete d'un client ***
Numero de client : MR005
Nom = Albert
Date du dernier achat : 12/4/1998
Numero de l'article demande : MOUS234
    et son prix : 2390
----taille prix : 4
Coefficient de reduction = 4.25
Fourni ? = 0
Send socket OK
Listen socket OK
```

Un arrêt du serveur sera provoqué par un client s'annonçant comme "END\_OF\_CONNEXION" :

```
% cli
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.1
Connect socket OK
Nom du client (END_OF_CONNEXION) : END_OF_CONNEXION
Send socket OK
Demande envoyee pour le client = END_OF_CONNEXION
Socket client fermee
%
```

ce qui provoquera bien l'arrêt du serveur :

```
Accept socket OK
Passage boucle de reception
Taile msg recu = 0 et taille atendue = 92
Taill msg = 92 et nbrBytes = 92
Recv socket OK
Socket connectee au client fermee
Socket serveur fermee
boole>
```

## 16. Le problème des chaînes de caractères de longueur variable

Dans l'exemple précédent, il est finalement assez aisé de vérifier que l'on a reçu toute la structure attendue, puisqu'on en connaît la taille à l'avance. Cependant, il n'en est plus de même pour un message se limitant à une simple chaîne de caractères, puisque celle-ci est généralement de longueur variable. Certes, on pourrait dimensionner un tableau de caractères à une taille jugée "suffisante" (5000 par exemple); mais cela signifierait que l'on échangerait invariablement le même nombre de bytes, sans se soucier du nombre de bytes effectivement utiles (par exemple véhiculer 5000 bytes pour transmettre "Hello world !"). En fait, pour savoir si l'on a reçu toute la chaîne, il convient plutôt d'appliquer l'une des stratégies suivantes :

- ◆ si l'émetteur ferme la connexion une fois l'information demandée fournie (cas de HTTP, d'un serveur de date-heure classique), il suffit d'attendre que le `recv()` reçoive zéro;
- ◆ sinon, il faut *guetter une marque quelconque de fin de message*. TCP ne fournit pas lui-même un marqueur de fin : c'est donc au programmeur à y pourvoir. On peut par exemple :
  - fournir en tête de message le nombre de bytes à lire au niveau TCP (cas du DNS);
  - *gérer ses propres marqueurs*, comme `\r\n` (cas de FTP ou SMTP).

C'est cette dernière stratégie que nous allons développer ici : **nous allons considérer que toute chaîne de caractère envoyée au serveur doit être terminée par les deux caractères `\r\n`.** La fonction `marqueurRecu()`, comme son nom l'indique finement, sera chargée de rechercher ces marqueurs de fin dans le paquet qu'elle reçoit comme paramètre. Inutile de dire que la fonction de lecture boucle tant que cette fonction n'a pas détecté la séquence de fin, ce qu'elle indiquera par sa valeur renournée récupérée dans une variable flag `finDetectee` :

### TCPITER03CD.C

```
/* TCPITER03CD.C
```

```
- Claude Vilvens -
```

```
*/
```

```
#include <stdio.h>
```

```
...
```

```
#include "tcpiter03.h"
```

```

char marqueurRecu (char * m, int n);

int main()
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService;
        /* Handle de la socket de service connectee au client */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    unsigned int tailleSockaddr_in;

    char msgServeur [LONG_MSG_SERV];
    char msgClient [LONG_MSG_CLIENT];
    int tailleMsgRecu, nbreBytesRecus;
    int i;
    char buf[5500];
    int finDetectee;

    int tailleS, tailleO;
    printf("?? Taille d'un client = %d\n", sizeof (struct client));

/* 1. Creation de la socket */
    ...
/* 2. Acquisition des informations sur l'ordinateur local */
    ...
/* 3. Preparation de la structure sockaddr_in */
    ...
/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
    ...
/* 5. Mise a l'ecoute d'une requete de connexion */
    ...
/* 6. Acceptation d'une connexion */
    ...
/* 7. Recherche du MTU */
    tailleO=sizeof(int);
    if (getsockopt(hSocketService, IPPROTO_TCP, TCP_MAXSEG,
                   &tailleS, &tailleO) == -1)
    {
        printf("Erreur sur le getsockopt de la socket %d\n", errno);
        exit(1);
    }
    else
    {
        printf("getsockopt OK\n");
        printf("Taille maximale d'un segment = %d\n", tailleS);
    }
}

```

```

for (i=0; i<3; i++)
{
/* 8.Reception de trois messages client */
tailleMsgRecu = 0;
finDetectee = 0;
memset(buf,0,sizeof(buf));
do
{
    puts("Passage boucle de reception");
    if ( (nbreBytesRecus = recv(hSocketService, buf, tailleS, 0)) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketService); close(hSocketEcoute); exit(1);
    }
    else
    {
        finDetectee = marqueurRecu (buf, nbreBytesRecus);
        memcpy((char *)msgClient + tailleMsgRecu, buf,nbreBytesRecus);
        tailleMsgRecu += nbreBytesRecus;
        printf("finDetectee = %d\n", finDetectee);
        printf("Nombre de bytes recus = %d\n", nbreBytesRecus );
        printf("Taile totale msg recu = %d\n", tailleMsgRecu );
    }
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 && !finDetectee);

msgClient[tailleMsgRecu-2]=0; /* zero de fin de chaine */
printf("Recv socket OK\n");

if (strcmp(msgClient, EOC))
    printf("Demande recue pour le client = %s\n", msgClient);
else break;

/* 9. Envoi de l'ACK du serveur au client */
printf("Message recu = %s\n", msgClient);
sprintf(msgServeur,"Demande n° %d recue du client !!!", i+1);
printf("Message de réponse = %s\n", msgServeur);

if (send(hSocketService, msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");
}

/* 10. Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */

```

```

printf("Socket connectee au client fermee\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");

return 0;
}

/* -----
char marqueurReçu (char *m, int nc)
/* Recherche de la séquence \r\n */
{
    static char demiTrouve=0;
    int i;
    char trouve=0;
    if (demiTrouve==1 && m[0]=='\n') return 1;
    else demiTrouve=0;

    for (i=0; i<nc-1 && !trouve; i++)
        if (m[i]=='\r' && m[i+1]=='\n') trouve=1;

    if (trouve) return 1;
    else if (m[nc]=='\r')
    {
        demiTrouve=1;
        return 0;
    }
    else return 0;
}

```

Le client, de son côté, va envoyer trois messages de type chaîne de caractères : deux "normaux" et un "très gros" :

<b>TCPCLI03CD.C</b>
<pre> /* TCPCLI03CD.C - Claude Vilvens - */ #include &lt;stdio.h&gt; ... #define ADRESSE "10.59.4.1" /* Adresse IP du serveur */ #define PORT 50000 /* Port d'écoute de la socket serveur */  #define MAXSTRING 100 /* Longueur des messages */ #define LONG_GROS_MSG 5000 /* Longueur d'un gros message */ </pre>

```

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    unsigned int tailleSockaddr_in;
    int ret; /* valeur de retour */

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    char *msgClientCourt;
    int t, tailleO, nbreRecv, i, nbreEnv;
    char grosMsg[LONG_GROS_MSG], gMsgServeur[LONG_GROS_MSG];

/* 1. Création de la socket */
    ...
/* 2. Acquisition des informations sur l'ordinateur distant */
/* Utilisation de l'adresse à la dure */
    printf("Adresse IP du serveur = %s\n",ADRESSE);

/* 3. Préparation de la structure sockaddr_in */
    ...
    adresseSocket.sin_addr.s_addr = inet_addr(ADRESSE);

/* 4. Tentative de connexion */
    ...
/* 5. Envoi d'un message client */
    strcpy(msgClient,"Bonjour ! Nous nous connaissons ?\r\n");
    if (send(hSocket, msgClient, strlen(msgClient)+1, 0) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

    printf("Message envoyé = %s\n", msgClient);

/* 6. Reception de l'ACK du serveur au client */
    if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recv socket OK\n");

    printf("Message reçu en ACK = %s\n", msgServeur);

```

```

/* 7. Envoi d'un deuxième message client */
    printf("Message a envoyer : "); gets(msgClient);
    t = strlen(msgClient); printf("taille msgClient = %d\n", t);
    msgClient[t]='r';
    msgClient[t+1]='\n';
    msgClient[t+2]=0;
    free(msgClientCourt);
    if (send(hSocket, msgClient, strlen(msgClient)+1, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");
    printf("Message envoye = %s\n", msgClient);

/* 8. Reception de l'ACK du serveur au client */
    if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recv socket OK\n");
    printf("Message recu en ACK = %s\n", msgServeur);

/* 9. Envoi d'un gros message client */
    for (i=0; i<LONG_GROS_MSG-3; i++) grosMsg[i]=i%2==0?'A':'B';
    grosMsg[LONG_GROS_MSG-2]='\r';
    grosMsg[LONG_GROS_MSG-1]='\n';
    grosMsg[LONG_GROS_MSG]=0;
    printf("Longueur client envoyé == %d \n", strlen(grosMsg)+1);
    printf("Gros msg envoye = %s\n", grosMsg);

    if ( (nbreEnv=send(hSocket, grosMsg, LONG_GROS_MSG, 0)) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        printf("Send socket OK\n");
        printf("Nbre de bytes envoyees = %d\n", nbreEnv);
    }

/* 11. Fermeture de la socket */
    close(hSocket); /* Fermeture de la socket */ printf("Socket client fermee\n");
    return 0;
}

```

Du côté du serveur, on aura :

```
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Listen socket OK
Accept socket OK
getsockopt OK
Taille maximale d'un segment = 1460

Passage boucle de reception
finDetectee = 1
16.1.1.1.1 Nombre de bytes recus = 36
Taille totale msg recu = 36
Recv socket OK
Demande recue pour le client = Bonjour ! Nous nous connaissons ?
Message recu = Bonjour ! Nous nous connaissons ?
Message de réponse = Demande n° 1 recue du client !!!
Send socket OK

Passage boucle de reception
finDetectee = 1
16.1.1.1.2 Nombre de bytes recus = 17
Taille totale msg recu = 17
Recv socket OK
Demande recue pour le client = Je suis la !!!
Message recu = Je suis la !!!
Message de réponse = Demande n° 2 recue du client !!!
Send socket OK

Passage boucle de reception
finDetectee = 0
16.1.1.1.3 Nombre de bytes recus = 1460
Taille totale msg recu = 1460
Passage boucle de reception
finDetectee = 0
16.1.1.1.4 Nombre de bytes recus = 1460
Taille totale msg recu = 2920
Passage boucle de reception
finDetectee = 0
16.1.1.1.5 Nombre de bytes recus = 1460
Taille totale msg recu = 4380
Passage boucle de reception
finDetectee = 1
16.1.1.1.6 Nombre de bytes recus = 620
Taille totale msg recu = 5000
Recv socket OK
```

Demande recue pour le client =

ABA

Message recu =

AB

ABABABABABA

Message de réponse = Demande n° 3 recue du client !!!

Send socket OK

Socket connectee au client fermee

Socket serveur fermee

Cette fois , tout est arrivé ! Pour le client, le dialogue ressemble à ceci :

Creation de la socket OK

Adresse IP du serveur = 10.59.4.1

Connect socket OK

Send socket OK

Message envoyee = Bonjour ! Nous nous connaissons ?

Recv socket OK

Message recu en ACK = Demande n° 1 recue du client !!!

Message a envoyer : Je suis la !!!

taille msgClient = 14

Send socket OK

Message envoyee = Je suis la !!!

Recv socket OK

Message recu en ACK = Demande n° 2 recue du client !!!

Longueur client envoyé == 4998

Gros msg envoyee =

AB

ABABABAA

Send socket OK

Nbre de bytes envoyees = 5000

Socket client fermee



Dans le cas où le serveur est contacté par plusieurs clients, celui-ci réagit au coup par coup. N'existe-t-il pas une technique permettant de surveiller plus globalement les points d'entrée de la communication ? Et bien voici ...

### III. Un serveur TCP/IP multi-sockets



*On se persuade mieux, pour l'ordinaire,  
par les raisons qu'on a soi-même  
trouvées que par celles qui sont venues  
dans l'esprit des autres.*

(B. Pascal, Pensées)

#### 1. Le principe de base

Un **serveur concourant** est capable de se mettre en attente sur les sockets de chacun de ses clients, sans pouvoir prédire l'ordre dans lequel les demandes s'effectueront. Lorsqu'il reçoit un message de l'un de ses clients, le serveur enclenche un traitement (en pratique, crée un thread – voir chapitre suivant) qui sera chargé de traiter la demande du client. On implémente donc ainsi un système de *multiplexage*.

La fonction **select()** permet précisément à un process de se mettre en attente sur plusieurs sockets en même temps. On trouve dans select.h :

```
int select ( <nombre de descripteurs – int>,
             <indicateurs de lecture ou non pour les sockets scrutées - fd_set *>,
             <indicateurs d'écriture ou non pour les sockets scrutées - fd_set *>,
             <indicateurs out-of-band ou non pour les sockets scrutées - fd_set *>,
             <temps d'attente maximal - struct timeval * >);
```

Lorsqu'un événement se produit (par exemple, l'arrivée de données sur une socket client), select() rend la main au programme appelant. En l'absence d'erreur, cette fonction renvoie comme valeur le nombre d'événements qui se sont produits. Comme nous allons le voir, il s'agit alors de tester les paramètres 2,3 et/ou 4 pour en savoir plus sur ce qui s'est passé. Si, par contre, une erreur s'est produite, la fonction renvoie -1 et errno est positionnée.



Le premier paramètre spécifie bien sûr le nombre de descripteurs utilisés par le processus. Attention ! Il s'agit aussi bien de descripteurs de fichiers que de sockets.

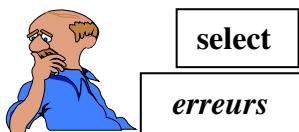
Le dernier paramètre permet de spécifier le temps maximum d'attente pour la réception d'un événement. Il s'agit de l'adresse d'une structure déclarée dans sys/time.h :

<b>struct timeval</b>
struct timeval
{
time_t tv_sec;     /* seconds */
int   tv_usec;    /* microseconds */
};

Une valeur nulle indique l'absence de time-out. Par contre, si le time-out est utilisé, la fonction renverra 0 si rien ne s'est produit avant l'expiration du délai.

Les trois paramètres de même nature sont, en première analyse, des indicateurs précisant quels types d'événements on souhaite traiter dans le processus. Ces événements sont groupés en trois catégories :

- ◆ ***lecture*** : le client effectue un send, un write, un close, un shutdown ou un connect OU une erreur est survenue sur la socket au niveau du client;
- ◆ ***écriture*** : le serveur effectue un send, un write, un close ou un shutdown OU le client effectue un close ou un shutdown OU une erreur est survenue sur la socket client au moment de l'envoi des données.
- ◆ ***exception*** : le client a effectué un send d'un message out-of-band.



Si la valeur de retour est -1, une erreur s'est produite et la variable globale errno est positionnée sur l'une des valeurs suivantes :

valeur de errno	erreur
#define EBADF 9	/* Bad file number */ : un descripteur est incorrect dans un ensemble
#define EINTR 4	/* Interrupted system call */ : l'appel a été interrompu
#define EINVAL 22	/* Invalid argument */ : nombre de descripteurs ou temps incorrect
#define EWOULDBLOCK 35	/* Operation would block */ : les sockets sont non bloquantes et il n'y a rien à lire
#define EFAULT 14	/* Bad address */ : l'adresse n'est pas accessible en lecture dans l'espace d'adressage de l'utilisateur

Précisons quelque peu la nature des trois paramètres indicateurs.

## 2. La structure fd\_set

### 2.1 Un tableau masque

Ces trois paramètres similaires sont les adresses d'une structure :

<b>struct fd_set</b>
typedef struct <b>fd_set</b>
{
fd_mask fds_bits[fds_howmany ( <b>FD_SETSIZE</b> , <b>FD_NFDBITS</b> )];
} <b>fd_set</b> ;

où

typedef int **fd\_mask**;

Il s'agit donc d'une structure ne comportant qu'un tableau d'entiers. Le rôle de chaque bit de ce tableau est d'enregistrer l'activation ou la désactivation de la surveillance de la lecture, de l'écriture ou d'une exception (c'est-à-dire d'un message out-of-band) sur la socket client correspondant au descripteur dont le numéro est la position dans le tableau. Chaque tableau est donc en fait un **masque**. Si, par exemple, le tableau suivant :

bits	0	1	2	3	4	...	n-2	n-1
	0	<b>1</b>	<b>1</b>	0	0		0	0

est le tableau de masque pour la lecture, la fonction select se déclenchera si des données sont prêtes à être lues sur la socket de descripteur 1 ou 2. Par contre, elle ne se déclenchera pas, dans une situation semblable, pour les autres sockets.

## 2.2 Le nombre de flags

Combien y a-t-il d'éléments dans ce tableau ? Ce nombre est donné par

- ◆ le nombre de descripteurs de sockets ouvrables dans un processus (nombre donné par FD\_SETSIZE et valant par défaut FD\_MAX\_NOFILE, soit 4096)
- divisé par
- ◆ le nombre de bits dans la représentation d'un entier (nombre donné par FD\_NFDBITS, donc 32).

En pratique, cela donne donc  $4096/32$ , soit 128 éléments. Evidemment, il se pourrait que la division ne donne pas un résultat entier, mais décimal. Les décimales seraient alors perdues. Le calcul que l'on trouve dans le header select.h augmente pour cela le dividende de la taille d'un entier diminuée de 1 pour être certain d'obtenir un nombre de cellules de tableaux suffisant :

```
#ifndef howmany
#define howmany(x, y)    (((x)+((y)-1))/(y))
#define fds_howmany(x, y) howmany(x, y)
#endif
```

Ici, le premier argument est :

```
#ifndef FD_SETSIZE
#define FD_SETSIZE      FD_MAX_NOFILE
#endif
```

et le deuxième :

```
#define FD_NFDBITS      (sizeof(long) * FD_NBBY) /* bits per mask */
#define FD_NBBY    8          /* number of bits in a byte */
#define FD_MAX_NOFILE   4096
```

### **2.3 Positionner un flag**

Comment positionner ces flags ? En utilisant des macros définies dans le même header select.h :

- ◆ **FD\_SET** (int hSocket, fd\_set \*tabMasque) : le bit du masque correspondant au descripteur de la socket est mis à 1 puisque

```
#define FD_SET(n, p)((p)->fds_bits[(n)/FD_NFDBITS] |= (1L << ((n) % FD_NFDBITS)))
```

- ◆ **FD\_CLR** (int hSocket, fd\_set \*tabMasque) : le bit du masque correspondant au descripteur de la socket est mis à 0 car

```
#define FD_CLR(n, p)((p)->fds_bits[(n)/FD_NFDBITS] &= ~(1L << ((n) % FD_NFDBITS)))
```

On peut tester l'état du bit d'un masque correspondant à une socket donnée par :

**FD\_ISSET** (int hSocket, fd\_set \*tabMasque)

Cette macro renvoie 0 ou pas selon que ce bit est à 0 ou 1 :

```
#define FD_ISSET(n, p) ((p)->fds_bits[(n)/FD_NFDBITS] & (1L << ((n) % FD_NFDBITS)))
```

Enfin, on peut remettre à 0 l'ensemble du masque par :

**FD\_ZERO** (fd\_set \*tabMasque)

### **3. Le serveur multi-clients**

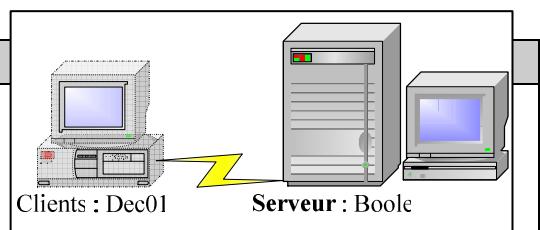
Nous allons donc mettre en place un serveur qui accepte les assauts de plusieurs clients, avec un nombre maximum fixé.

Nous allons donc déclarer un tableau de sockets d'écoute, un tableau de sockets de service (socket connectée) et une socket complémentaire pour les cas où la connexion demandée est refusée (elle permettra de notifier ce refus au client évincé). Initialement, les descripteurs sont non valides et donc initialisés à -1.

Nous ne traiterons que les événements de lecture sur les deux types de sockets (tentatives de connexion et réceptions d'un message).

#### **TCPMULT01.C**

```
/* TCPMULT01.C
- Claude Vilvens -
- Serveur multi-sockets avec select
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */
```



```
#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#include <time.h> /* pour select et timeval */

#define NB_MAX_SOCKETS 5 /* Nombre de sockets a creer au depart */
#define NB_MAX_CLIENTS 10 /* Nombre maximum de clients connectes */
#define NB_DESC_DEF 4 /* Nombre de descripteurs pris par defaut :
                      stdin, stdout, stderr et ERRNOFILE */

#define PORT 50000 /* Port d'ecoute de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket[NB_MAX_SOCKETS], /* Sockets utilisees pour l'attente */
        hSocketConnectee[NB_MAX_CLIENTS], /* Sockets pour clients*/
        hSocketRefusee; /* Socket utilisee pour exprimer le refus de connexion*/
    int i,j, /* variables d'iteration */
        retSelect, /* Code de retour du select */
        retRecv; /* Code de retour dun recv */

    fd_set ReadMask /*, WriteMask, ExceptionMask*/ ;
    /* Structures contenant un tableau d'entiers destines aux
     flags correspondant aux trois evenements possibles */

    struct hostent * infosHost;
    struct in_addr adresseIP;
    struct sockaddr_in adresseSocket;

    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];

/* 0. Initialisation des tableaux de sockets */
    /* Si la socket n'est pas utilisee, le descripteur est a -1 */
    for (i=0; i<NB_MAX_SOCKETS; i++) hSocket[i] = -1;
    for (i=0; i<NB_MAX_CLIENTS; i++) hSocketConnectee[i] = -1;
    hSocketRefusee = -1;
```

```

/* 1. Acquisition des informations sur l'ordinateur local */
if ( (infosHost = gethostbyname("boole"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    exit(1);
}
else printf("Acquisition infos host OK\n");
memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
printf("Adresse IP = %s\n",inet_ntoa(adresseIP));

/* 2. Cr閐ation de toutes les sockets */
for (i=0; i<NB_MAX_SOCKETS; i++)
{
/* 2.1 Cr閐ation de la socket numero i */
    hSocket[i] = socket(AF_INET, SOCK_STREAM,0
    if (hSocket[i] == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket num. %dOK\n",i);

/* 2.2 Pr閞paration de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET;
    adresseSocket.sin_port = htons(PORT + i);
    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 2.3. Le syst鑝e prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocket[i], (struct sockaddr *)&adresseSocket,
              sizeof(struct sockaddr_in)) == -1)
    {
        printf("Socket num. %d : Erreur sur le bind de la socket %d\n",
               i, errno);
        exit(1);
    }
    else printf("Socket num. %d : Bind adresse et port socket OK\n",i);

/* 2.4 Mise a l'ecoute d'une requete de connexion */
    if (listen(hSocket[i], SOMAXCONN) == -1)
    {
        printf("Socket num. %d: Erreur sur le listen de la socket %d\n"
               ,i , errno);
        close(hSocket[i]); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Socket num. %d :Listen socket OK\n",i);
} /* fin du for */

/* -----
 */

```

```

/* 3. Positionnement des masques */

while (1)
{
    retSelect = 0;

    FD_ZERO(&ReadMask);
    /* Si nécessaire : FD_ZERO(&WriteMask); FD_ZERO(&ExceptionMask); */

    for (i=0; i<NB_MAX_SOCKETS; i++)
    {
        if (hSocket[i] != 0) FD_SET(hSocket[i],&ReadMask);
    }
    for (i=0; i<NB_MAX_CLIENTS; i++)
    {
        if (hSocketConnectee[i] != -1) FD_SET(hSocketConnectee[i],&ReadMask);
    }

/* ----- */

/* 4. Attente du select */
    retSelect = select(NB_MAX_SOCKETS+NB_MAX_CLIENTS+NBNB_DESC_DEF,
                      &ReadMask,
                      0,      /*&WriteMask*/
                      0,      /*&ExceptionMask*/
                      0);

/* 5. Le select s'est déclenché */
/* 5.1 Erreurs ou rien */
    if (retSelect == -1)
    {
        printf("Erreur sur le select %d\n", errno);
        /* Fermeture des sockets */
        for (i=0; i<NB_MAX_SOCKETS; i++)
        {
            printf("Fermeture de la socket %d\n", hSocket[i]);
            close(hSocket[i]);
        }
        for (i=0; i<NB_MAX_CLIENTS; i++)
        {
            printf("Fermeture de la socket %d\n", hSocketConnectee[i]);
            close(hSocketConnectee[i]);
        }
        if (hSocketRefusee != -1)
        {
            printf("Fermeture de la socket %d\n", hSocketRefusee);
            close(hSocketRefusee);
        }
        exit(1);
    }
}

```

```

if (retSelect == 0)
{
    /* Il n'y a pas de socket a traiter */
    puts("--- rien a traiter - remise en attente ---");
    continue;      /* Remise en attente */
}

printf("Nombre d'evenements a traiter = %d\n", retSelect);
tailleSockaddr_in = sizeof(struct sockaddr);

/* Sockets de base */
for (i=0; i<NB_MAX_SOCKETS; i++)
{

/* 5.2 Cas du declenchement en lecture sur une socket de base */
if (hSocket[i] != -1 && FD_ISSET(hSocket[i], &ReadMask))
/* Cas d'une demande de connexion */
{
    /* 5.2.1 Recherche d'une socket connectee libre */
    printf("Recherche d'une socket connecteee libre ... \n");
    for (j=0; j<NB_MAX_CLIENTS && hSocketConnectee[j] !=-1; j++);

    /* 5.2.2 Cas ou toutes les connexions sont utilisees */
if (j == NB_MAX_CLIENTS)
{
    printf("Plus de connexion disponible \n");
    if ( (hSocketRefusee =
        accept(hSocket[i], (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d \n", errno);
        close(hSocketRefusee); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Socket refusee OK \n");

    sprintf(msgServeur,"Connexion refusee par le serveur");
    if (send(hSocketRefusee, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket refusee %d \n"
               , errno);
        close(hSocketRefusee); /* Fermeture de la socket */

        exit(1);
    }
    else printf("Send socket refusee OK");
    close(hSocketRefusee); /* Fermeture de la socket */
    hSocketRefusee = -1; /* Fermeture de la socket */
}
else

```

```

/* 5.2.3 Il y a une connexion de libre */
printf("Connexion sur la socket num. %d\n", j);
if ( (hSocketConnectee[j] =
      accept(hSocket[i], (struct sockaddr *)&adresseSocket, &tailleSockaddr_in))
     == -1)
{
    printf("Erreur sur l'accept de la socket %d : %d\n", i, errno);
    close(hSocketConnectee[j]); /* Fermeture de la socket */
    exit(1);
}
else printf("Accept Socket %d OK\n",j);
retSelect--;
if (retSelect == 0) break;
}
/* fin du for */

if (retSelect == 0) continue;
/* Toutes les actions sur les sockets ont ete traitees =>
pas la peine de tester les sockets clients */

/* Sockets clients */
for (i=0; i<NB_MAX_CLIENTS; i++)
{

/* 5.3 Cas du declenchement en lecture sur une socket client */
if (hSocketConnectee[i] != -1 &&
    FD_ISSET(hSocketConnectee[i], &ReadMask))
{
    if ((retRecv=recv(hSocketConnectee[i], msgClient, MAXSTRING,0)) == -1)
    {
        if (errno == ECONNRESET)
        {
            printf("ECONNRESET - hsocket fermee\n");
            hSocketConnectee[i] = -1;
            retSelect--;
        }
        else
        {
            printf("Erreur sur le recv de la socket connectee %d : %d\n", i, errno);
            switch(errno)
            {
                case EBADF : printf("EBADF - hsocket n'existe pas\n");
                break;
                ...
                default : printf("Erreur inconnue ?\n");
            }
            close(hSocketConnectee[i]); /* Fermeture de la socket */
            exit(1);
        }
    }
}
}

```

```

else printf("Recv socket OK");
if (retRecv == 0)
{
    printf("Deconnexion du client\n");
    close (hSocketConnectee[i]); /* Fermeture de la socket */
    hSocketConnectee[i] = -1;
}
else printf("Message recu = %s\n", msgClient);
sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
if (send(hSocketConnectee[i], msgServeur, MAXSTRING, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketConnectee[i]); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket connectee %d OK\n",i);
retSelect--;
if (retSelect == 0) break;
}
} /* fin du for */
printf("Fin de la boucle serveur\n");
printf("Valeur de retSelect = %d\n", retSelect);
} /* fin du while(1) */

/* 6. Fermeture des sockets */
for (i=0; i<NB_MAX_SOCKETS; i++)
{
    if (hSocket[i] != -1)
    {
        printf("Fermeture de la socket %d\n",hSocket[i]);
        close(hSocket[i]);
        hSocket[i] = -1;
    }
}
for (i=0; i<NB_MAX_CLIENTS; i++)
{
    if (hSocketConnectee[i] != -1)
    {
        printf("Fermeture de la socket %d\n", hSocketConnectee[i]);
        close(hSocketConnectee[i]);
        hSocketConnectee[i] = -1;
    }
}
if (hSocketRefusee != -1)
{
    printf("Fermeture de la socket %d\n",hSocketRefusee);
    close(hSocketRefusee);
}
return 0;
}

```

Le client reste un client simple du type du début du chapitre précédent (TCPCLI02.C).  
Au niveau de l'exécution, un premier client va se connecter et envoyer des messages :

```
% cli
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.1
Connect socket OK
Message num 1 a envoyer : ici Dieu !
Send socket OK
Message envoyé = ici Dieu !
Recv socket OK
Message recu en ACK = ACK pour votre message : <ici Dieu !>
Message num 2 a envoyer : Voici mes commandements :
Send socket OK
Message envoyé = Voici mes commandements :
Recv socket OK
Message recu en ACK = ACK pour votre message : <Voici mes commandements :>
Message num 3 a envoyer :
```

et deux autres clients vont interférer avec le premier :

```
% cli
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.1
Connect socket OK
Message num 1 a envoyer : ici Allah
Send socket OK
Message envoyé = ici Allah
Recv socket OK
Message recu en ACK = ACK pour votre message : <ici Allah>
Message num 2 a envoyer : Que ma volonté soit faite !
Send socket OK
Message envoyé = Que ma volonté soit faite !
Recv socket OK
Message recu en ACK = ACK pour votre message : <Que ma volonté soit faite !>
Message num 3 a envoyer :
```

et

```
% cli
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.1
Connect socket OK
Message num 1 a envoyer : ici Brahma ...
Send socket OK
Message envoyé = ici Brahma ...
Recv socket OK
```

Message recu en ACK = ACK pour votre message : <ici Brahma ...>

Message num 2 a envoyer : Médite et tais-toi !

Send socket OK

Message envoyee = *Médite et tais-toi !*

Recv socket OK

Message recu en ACK = ACK pour votre message : <Médite et tais-toi !>

Message num 3 a envoyer :

Le serveur réagira de la manière suivante :

```
boole> tcpmult01
Acquisition infos host OK
Adresse IP = 10.59.4.1
Creation de la socket num. 0OK
Socket num. 0 : Bind adresse et port socket OK
Socket num. 0 :Listen socket OK
Creation de la socket num. 1OK
Socket num. 1 : Bind adresse et port socket OK
Socket num. 1 :Listen socket OK
Creation de la socket num. 2OK
Socket num. 2 : Bind adresse et port socket OK
Socket num. 2 :Listen socket OK
Creation de la socket num. 3OK
Socket num. 3 : Bind adresse et port socket OK
Socket num. 3 :Listen socket OK
Creation de la socket num. 4OK
Socket num. 4 : Bind adresse et port socket OK
Socket num. 4 :Listen socket OK
Nombre d'evenements a traiter = 1
Recherche d'une socket connectee libre ...
Connexion sur la socket num. 0
Accept Socket 0 OK
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = ici Dieu !
Send socket connectee 0 OK
Fin de la boucle serveur
Valeur de retSelect = 0
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = Voici mes commandemants :
Send socket connectee 0 OK
Fin de la boucle serveur
Valeur de retSelect = 0
Nombre d'evenements a traiter = 1
Recherche d'une socket connectee libre ...
Connexion sur la socket num. 1
Accept Socket 1 OK
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = ici Allah
Send socket connectee 1 OK
Fin de la boucle serveur
```

---

```
Valeur de retSelect = 0
Nombre d'evenements a traiter = 1
Recherche d'une socket connectee libre ...
Connexion sur la socket num. 2
Accept Socket 2 OK
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = ici Brahma ...
Send socket connectee 2 OK
Fin de la boucle serveur
Valeur de retSelect = 0
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = Médite et tais-toi !
Send socket connectee 2 OK
Fin de la boucle serveur
Valeur de retSelect = 0
Nombre d'evenements a traiter = 1
Recv socket OKMessage recu = Que ma volonté soit faite !
Send socket connectee 1 OK
Fin de la boucle serveur
Valeur de retSelect = 0
```

etc



Au point où nous en sommes, on peut dire que nous communiquons. On ne peut pas dire pour la cause que l'utilisation des ressources est idéale : fondamentalement, tout est séquentiel. Ne serait-il pas possible de programmer des traitements parallèles ? Mais si, mais si ...

## IV. Un serveur TCP multi-threads POSIX

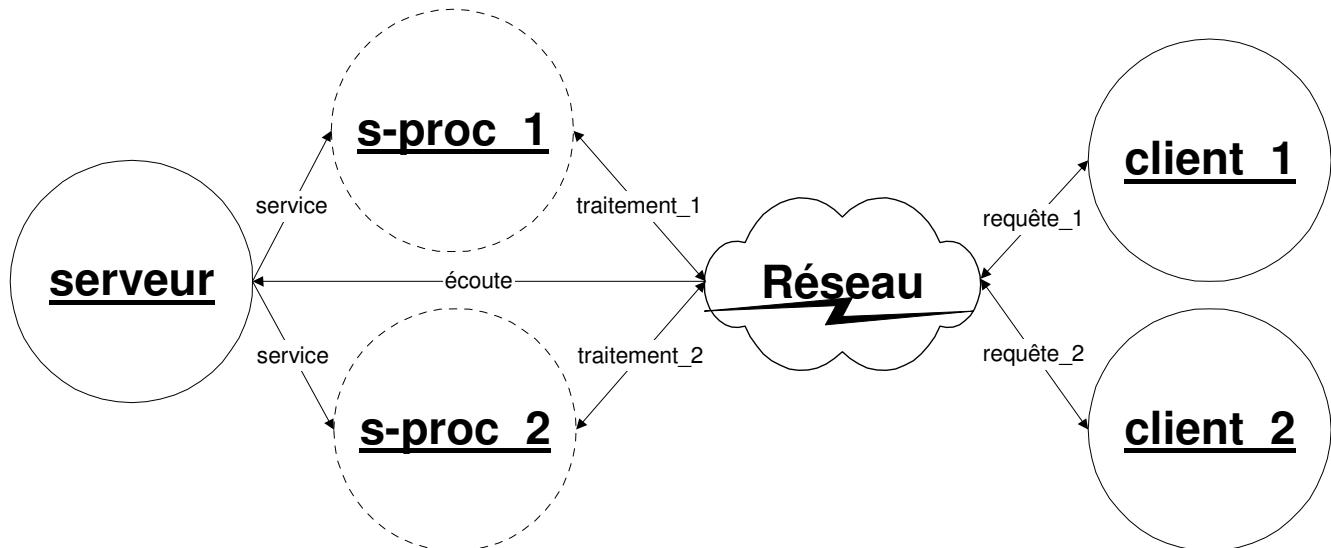


*L'insouciance ne s'improvise pas.*

(R. Radiguet, Le Bal du comte d'Orgel)

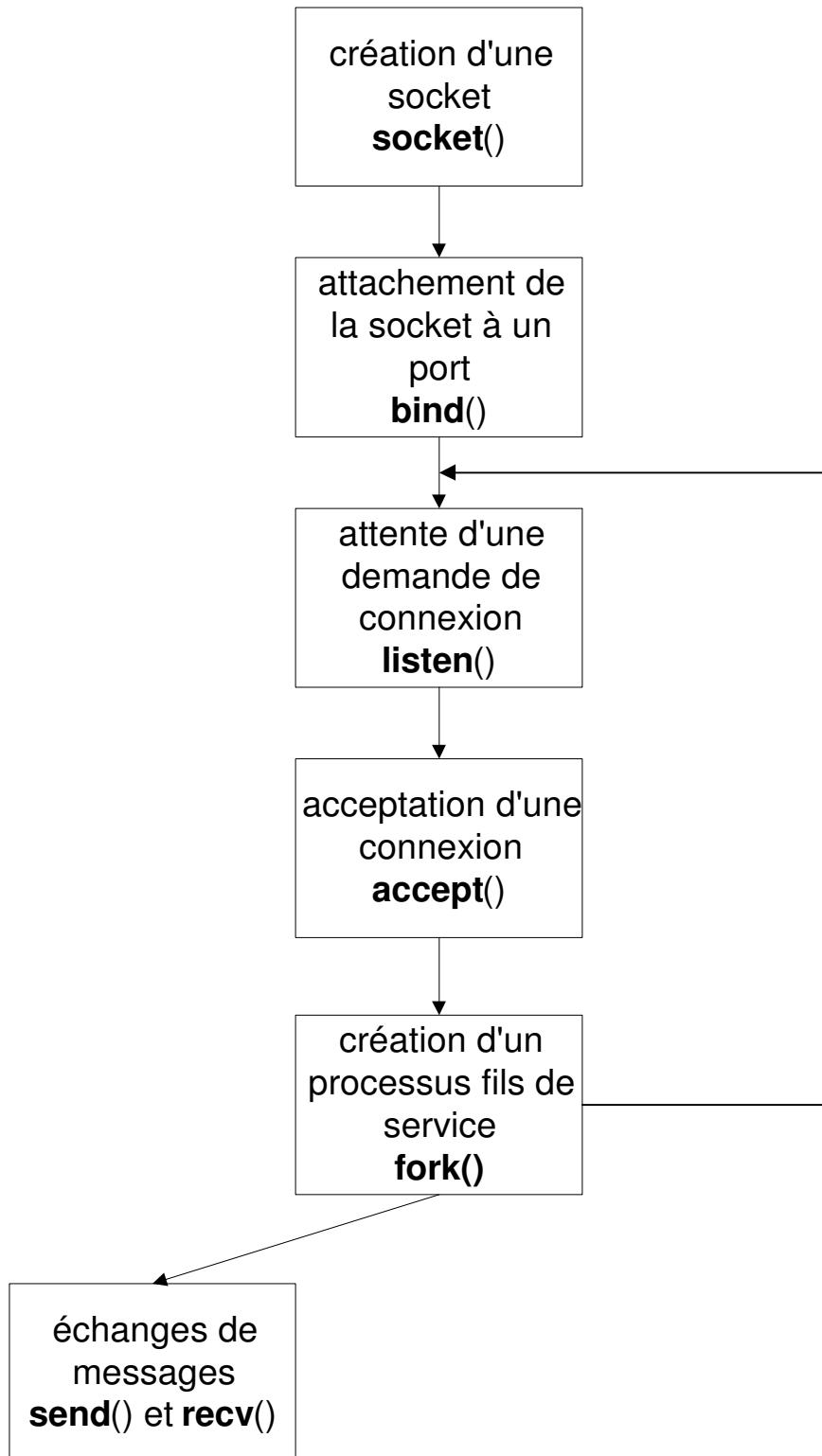
### 1. L'idée d'un serveur concurrent

Nous allons donc nous attaquer ici à la construction d'un serveur TCP multi-clients. Autrement dit nous allons rendre notre serveur effectivement multi-connexions. En bon programmeur UNIX, l'idée qui vient immédiatement à l'esprit est de créer un sous-processus pour chaque requête d'un client, ce qui permettra l'exécution en parallèle des traitements de ces requêtes.

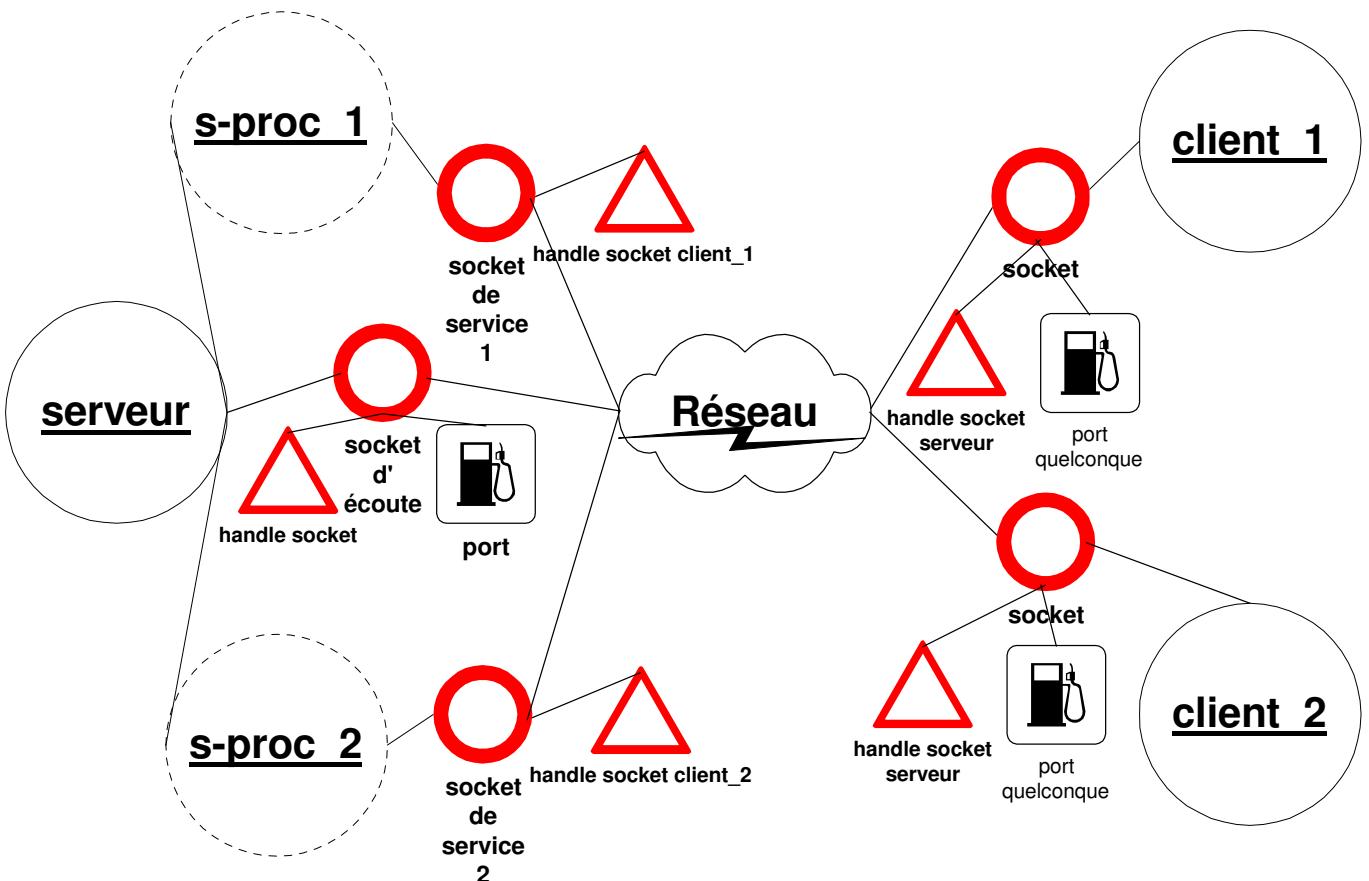


On conçoit sans peine qu'un tel mécanisme est plus efficace que la méthode classique qui consiste à placer les requêtes dans une file d'attente et à les traiter l'une après l'autre.

Le principe sera donc le suivant. Le serveur se met toujours en attente au sein d'une boucle infinie. Dès qu'il détecte une demande de connexion, il se décharge du traitement de la transaction sur un processus fils, à qui il transmet la socket de service fournie par accept(). Le processus fils va donc réaliser la suite de la transaction, tandis que le serveur peut se remettre à l'écoute :



Ce sera donc la fonction exécutée par le sous-processus qui s'occupera du dialogue avec le client. Cette fonction devra donc recevoir comme paramètre le handle de la socket de service créée par la fonction `accept()`. Schématiquement :



Le programme serveur complet s'écrit donc comme suit. On constatera que l'on n'a pas été avare de traceurs en tous genres pour bien comprendre le déroulement des opérations : un processus fils se voit affecté artificiellement un numéro (à partir de son port de service, également passé en argument à la fonction – on eut pu utiliser son pid) et la macro affProcess permet d'afficher l'état d'avancement des opérations.

### TCPIITER04FORK.C

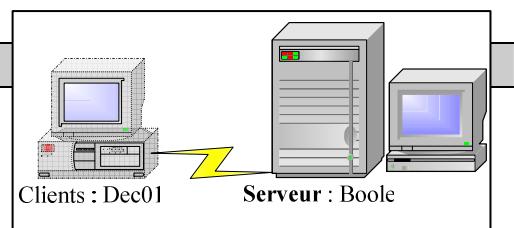
```
/* TCPIITER04FORK.C
 - Claude Vilvens -
 */
```

```
#include <stdio.h>
...
#include "tcpiter03.h"

void afficheRequete(struct client *c);
void fctSousProcess (int hSockServ, int port);

#define affSousProcess(num, msg) printf("pr_%d> %s\n", num, msg)

int main()
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService; /* Handle de la socket de service connectee au client */
```



```

struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
struct in_addr adresseIP; /* Adresse Internet au format reseau */
struct sockaddr_in adresseSocket;
unsigned int tailleSockaddr_in;
int pid;

puts("Process pere serveur demarre");
printf("identite = %d\n", getpid() );

/* 1. Creation de la socket */
...
/* 2. Acquisition des informations sur l'ordinateur local */
...
/* 3. Preparation de la structure sockaddr_in */
...
/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
...
do
{
/* 5. Mise a l'ecoute d'une requete de connexion */
    puts("Process principal : en attente d'une connexion");
    if (listen(hSocketEcoute,SOMAXCONN) == -1)
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 6. Acceptation d'une connexion et lancement d'un sous-process pour ce client */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
            accept(hSocketEcoute, (struct sockaddr *)&adresseSocket,
                   &tailleSockaddr_in) ) == -1)
    {
        if (errno == EINTR) continue;
        /* accept débloqué MAIS pas de connexion */
        else
        {
            printf("Erreur sur l'accept de la socket %d\n", errno);
            close(hSocketEcoute); /* Fermeture de la socket */
            exit(1);
        }
    }
    else printf("Accept socket OK\n");

    printf("Socket d'ecoute = %d\n", hSocketEcoute);
    printf("Socket de service attribuee = %d\n", hSocketService);
}

```

```

if ( (pid = fork()) == 0)
{
    puts("---- Dans sous-process");
    printf("Port utilise = %d\n", adresseSocket.sin_port);
    close(hSocketEcoute); /* "Fermeture" de la socket */
    fctSousProcess (hSocketService,adresseSocket.sin_port);
    close(hSocketService);
    puts("Socket connectee au client fermee");
    exit(0);
}
else
{
    puts("---- Dans process principal");
    close(hSocketService); /* "Fermeture" de la socket */
}
}

while(1);

/* 7. Fermeture des sockets */
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");
puts("Fin du process principal");
return 0;
}

/*
void afficheRequete(struct client *c)
{
    printf("*** Requete d'un client ***\n");
    printf("Numero de client : %s\n", c->numClient);
    printf("Nom = %s\n", c->nom);
    printf("Date du dernier achat : %s\n", c->dateDernierAchat);
    printf("Numero de l'article demande : %s\n", c->numArticle);
    printf(" et son prix : %d\n", c->montant);
    printf("Coefficient de reduction = %s\n", c->coeffReduction);
    printf("Fourni ? = %d\n", c->fourni);
}

/*
void fctSousProcess (int hSockServ, int port)
{
    int vr = port;
    char msgServeur [LONG_MSG_SERV];
    struct client *msgClient = (struct client *)malloc(sizeof(struct client));
    int tailleMsgRecu, nbreBytesRecus;
    char buf[100];

    printf("** vr = %d\n", vr);
    affSousProcess(vr,"Debut du sous-processus");
    sprintf(buf, "identite = %d\n", getpid()); affSousProcess(vr, buf);
    sprintf(buf, "je travaille sur la socket de service %d\n",hSockServ);
}

```

```

affSousProcess(vr, buf);

/* 1.Reception d'un message client */
tailleMsgRecu = 0;
do
{
    if ( (nbreBytesRecus = recv(hSockServ, ((char *)msgClient) + tailleMsgRecu,
        LONG_STRUCT_CLI-tailleMsgRecu, 0)) == -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSockServ); /* Fermeture de la socket */
        exit(1);
    }
    else tailleMsgRecu += nbreBytesRecus;
    printf("Taill msg = %d et nbreBytes = %d \n", tailleMsgRecu,
        nbreBytesRecus);
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
    tailleMsgRecu <LONG_STRUCT_CLI );

affSousProcess(vr, "Recv socket OK");

sprintf(buf, "Demande recue pour le client = %s\n", msgClient->nom);
affSousProcess(vr, buf);

/* 2. Envoi de l'ACK du serveur au client */
sprintf(msgServeur,"Demande recue du client %s !!!",
    msgClient->nom);
affSousProcess(vr, msgServeur);
afficheRequete(msgClient);

if (send(hSockServ, msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSockServ); /* Fermeture de la socket */
    exit(1);
}
else affSousProcess(vr, "Send socket OK");

affSousProcess(vr, "--fin du sous-process--");
}

```

Un exemple dexécution pour le serveur sera (s est l'exécutable) :

```
boole> s
Process pere serveur demarre
identite = 14895
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Process principal : en attente d'une connexion
Listen socket OK
Accept socket OK
Socket d'ecoute = 3
Socket de service attribuee = 4
---- Dans process principal
Process principal : en attente d'une connexion
Listen socket OK
---- Dans sous-process
Port utilise = 4874
** vr = 4874
pr_4874> Debut du sous-processus
pr_4874> identite = 16612

pr_4874> je travaille sur la socket de service 4

Accept socket OK
Socket d'ecoute = 3
Socket de service attribuee = 4
---- Dans process principal
Process principal : en attente d'une connexion
Listen socket OK
---- Dans sous-process
Port utilise = 5130
** vr = 5130
pr_5130> Debut du sous-processus
pr_5130> identite = 20172

pr_5130> je travaille sur la socket de service 4

Taill msg = 92 et nbreBytes = 92
pr_4874> Recv socket OK
pr_4874> Demande recue pour le client = Jules Vieux de la Vieille

pr_4874> Demande recue du client Jules Vieux de la Vieille !!!
*** Requete d'un client ***
Numero de client : 344RT
Nom = Jules Vieux de la Vieille
Date du dernier achat : 22/1/2000
Numero de l'article demande : AR9485
et son prix : 1205
```

```
Coefficient de reduction = 0.45
Fourni ? = 0
pr_4874> Send socket OK
pr_4874> --fin du sous-process--
Socket connectee au client fermee
Taill msg = 92 et nbreBytes = 92
pr_5130> Recv socket OK
pr_5130> Demande recue pour le client = Albert Letoutbeau

pr_5130> Demande recue du client Albert Letoutbeau !!!
*** Requete d'un client ***
Numero de client : 897YH
Nom = Albert Letoutbeau
Date du dernier achat : 3/2/2000
Numero de l'article demande : AR984785
    et son prix : 1299
Coefficient de reduction = 0.98
Fourni ? = 0
pr_5130> Send socket OK
pr_5130> --fin du sous-process--
Socket connectee au client fermee
boole>
```

si deux clients sont successivement connectés (c'est l'exécutable) :

% c ?? Taille d'un client = 92 Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.4.1 Connect socket OK Nom du client : Jules Vieux de la Vieille Numero client : 344RT Numero d'article : AR9485 Prix : 1205 Date du dernier achat : 22/1/2000 Coefficient de reduction : 0.45 Send socket OK Demande envoyee pour le client = Jules Vieux de la Vieille Recv socket OK Message recu en ACK = Demande recue du client Jules Vieux de la Vieille !!! Socket client fermee	% c ?? Taille d'un client = 92 Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.4.1 Connect socket OK Nom du client : Albert Letoutbeau Numero client : 897YH Numero d'article : AR984785 Prix : 1299 Date du dernier achat : 3/2/2000 Coefficient de reduction : 0.98 Send socket OK Demande envoyee pour le client = Albert Letoutbeau Recv socket OK Message recu en ACK = Demande recue du client Albert Letoutbeau !!! Socket client fermee
--	---

Deux remarques s'imposent encore :

**1)** On remarquera que l'on referme une socket dans un processus donné dès que l'on n'en a plus besoin. Donc :

- ♦ le processus père, c'est-à-dire le serveur, ferme la socket de service dès que le processus fils de traitement a été créé;
- ♦ le processus fils ferme la socket d'écoute dès son démarrage.

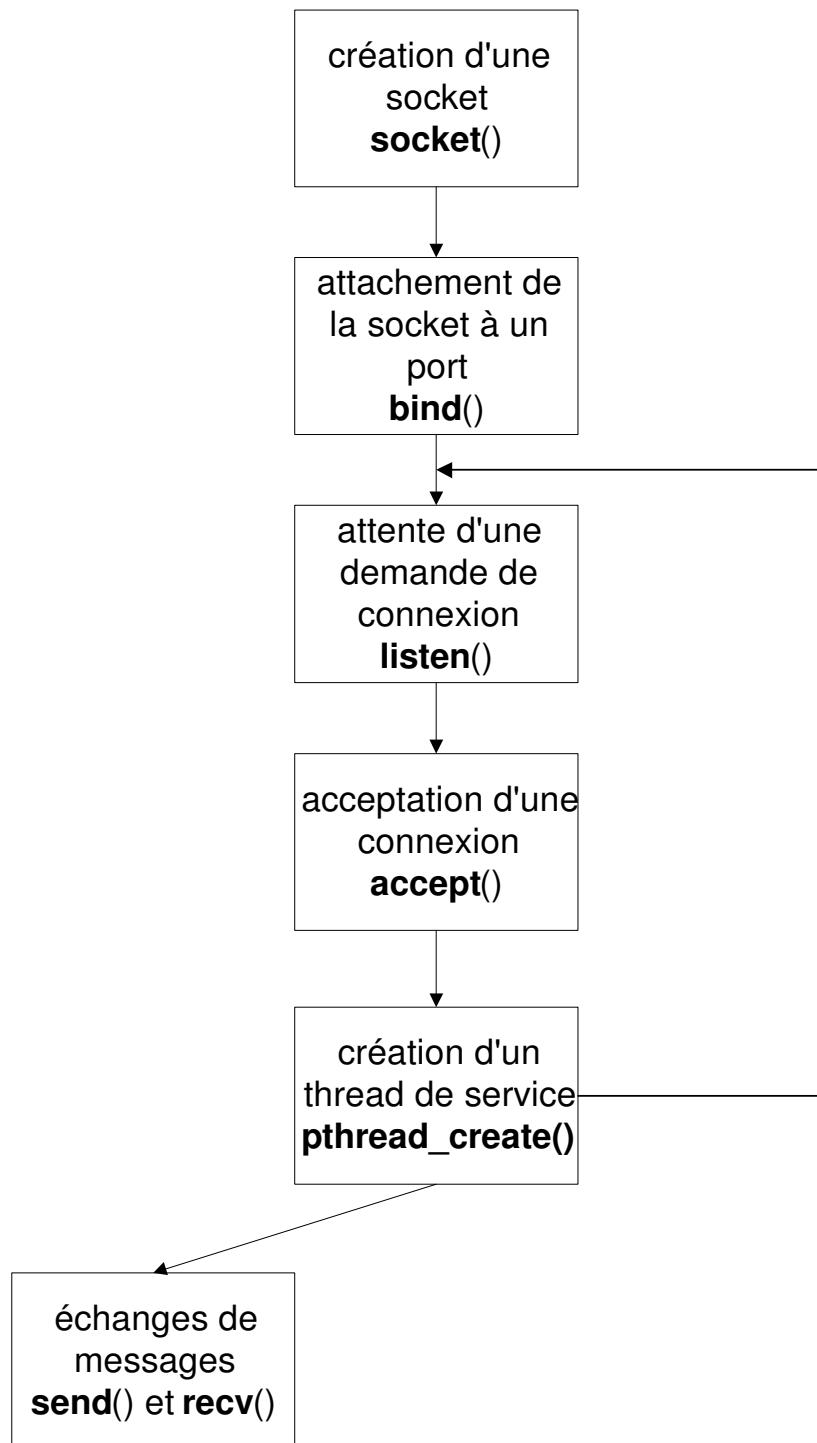
En réalité, il faut se souvenir que ***l'opération de fermeture décrémente le compteur de connexions accompagnant la socket***. Dans les deux cas, ce compteur passera ainsi de 2 à 1, si bien qu'en réalité aucune des deux sockets n'est réellement fermée ! On peut aussi remarquer que les deux processus fils utilisent le même handle de socket (4) : mais ils travaillent dans des environnements différents, si bien que *ce handle ne correspond pas à la même socket* (heureusement d'ailleurs !), comme on peut s'en convaincre en examinant le numéto de port éphémère utilisé.

**2)** Du strict point de vue du système d'exploitation, on peut regretter que chaque connexion implique la mise en place d'un processus fils, avec tout ce que cela implique comme duplication d'environnement; de plus, il faut utiliser les IPC pour qu'un process père puisse discuter avec son process fils. Tout cela est donc un peu lourd et nous fait penser à des "processus légers" ...

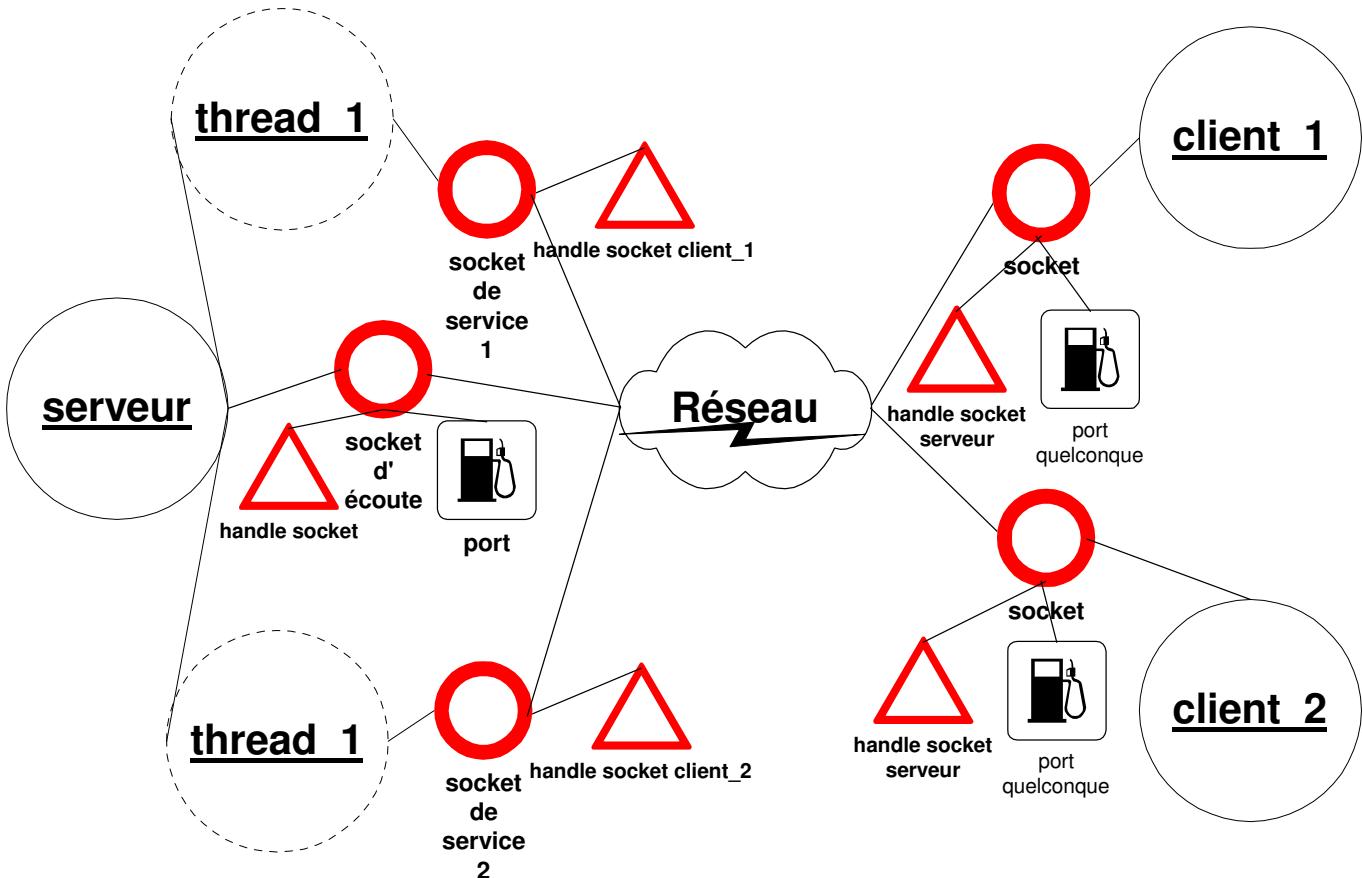
## 2. Un serveur multi-connexion et multi-thread à la demande

Nous pouvons donc aussi penser à rendre multi-connexions notre serveur TCP en le rendant multi-threads. Nous allons procéder d'une manière suivante très analogue à celle qui nous a fait utiliser les sous-processus, puisque, après tout, les threads sont des "*processus légers*".

Le serveur se met toujours en attente au sein d'une boucle infinie. Dès qu'il détecte une demande de connexion, il se décharge du traitement de la transaction sur un thread, à qu'il transmet la socket de service fournie par `accept()`. Le thread va donc réaliser la suite de la transaction, tandis que le serveur peut se remettre à l'écoute :



Ce sera donc la fonction exécutée par le thread qui s'occupera du dialogue avec le client. Cette fonction devra donc recevoir comme paramètre le handle de la socket de service créée par la fonction accept(). Schématiquement :



Le programme serveur complet s'écrit donc comme suit. Les traceurs en tous genre sont toujours bien là afin de bien comprendre le déroulement des opérations : un thread se voit affecté artificiellement un numéro (à partir de sa socket de service – on aurait pu utiliser son tid ou numéro de séquence) et la macro affThread permet d'afficher l'état d'avancement des opérations. On remarquera aussi que le thread est "ralenti" par le "sleep" propres aux threads, soit nanosleep() (ou pthread\_delay\_np() pour la machine copernic).

Voici donc notre serveur (à compiler avec **-lrt** en plus sur sunray) :

**TCPIITER04.C**

```

/* TCPIITER04.C
- Claude Vilvens -
*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>
```

Clients Copernic/Sunray Serveur : Sunray

```

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h>      /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h>    /* pour la conversion adresse reseau->format dot
                           ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h>  /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h>    /* pour la conversion adresse reseau->format dot */

#include "tcpiter03.h"

void afficheRequete(struct client *c);
void * fctThread(void * param);

#define affThread(num, msg) printf("th_%d> %s\n", num, msg)

pthread_t threadHandle;

int main()
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService; /* Handle de la socket de service connectee au client */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    unsigned int tailleSockaddr_in;
    int i;
    int ret, * retThread;

    puts("Thread principal serveur demarre");
    printf("identite = %d.%u\n", getpid(), pthread_self());

/* 1. Creation de la socket */
    hSocketEcoute = socket(AF_INET,SOCK_STREAM,0);
    if (hSocketEcoute == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur local */
    if ( (infosHost = gethostbyname("sunray2v440"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
    printf("Adresse IP = %s\n",inet_ntoa(adresseIP));
}

```

```

/* 3. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */
    adresseSocket.sin_port = htons(PORT); /* conversion numéro de port au format réseau */

    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocketEcoute, (struct sockaddr *)&adresseSocket,
              sizeof(struct sockaddr_in)) == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        exit(1);
    }
    else printf("Bind adresse et port socket OK\n");

do
{
/* 5. Mise à l'écoute d'une requête de connexion */
    puts("Thread principal : en attente d'une connexion");
    if (listen(hSocketEcoute, SOMAXCONN) == -1) /* Constante définie dans
                                                sys/socket.h [de 5 à 8 selon les systèmes] */
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 6. Acceptation d'une connexion et lancement d'un thread pour ce client */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
            accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");

    printf("Socket de service attribuée = %d\n", hSocketService);
    ret = pthread_create(&threadHandle, NULL, fctThread, (void*)hSocketService);
    puts("Thread secondaire lancé !");
    puts("Marquage pour effacement du thread secondaire");
    ret = pthread_detach(threadHandle);
}

while(1);

```

```

/* 7. Fermeture de la socket d'ecoute */
    close(hSocketEcoute); /* Fermeture de la socket */
    printf("Socket serveur fermee\n");

    puts("Fin du thread principal");
    return 0;
}

/* -----
void afficheRequete(struct client *c)
{
    printf("*** Requete d'un client ***\n");
    printf("Numero de client : %s\n", c->numClient);
    printf("Nom = %s\n", c->nom);
    printf("Date du dernier achat : %s\n", c->dateDernierAchat);
    printf("Numero de l'article demande : %s\n", c->numArticle);
    printf(" et son prix : %d\n", c->montant);
    printf("----taille prix : %d\n", sizeof (c->montant));
    printf("Coefficient de reduction = %5.3f\n", c->coeffReduction);
    printf("---- taille Coefficient de reduction = %d\n",
           sizeof(c->coeffReduction));
    printf("Fourni ? = %d\n", c->fourni);
}

/* -----
void * fctThread (void *param)
{
    int hSockServ = (int)param;
    int vr = hSockServ;
    char msgServeur [LONG_MSG_SERV];
    struct client *msgClient =
        (struct client *)malloc(sizeof(struct client));
    int tailleMsgRecu, nbreBytesRecus;
    char buf[100];
    struct timespec temps;
    temps.tv_sec = 20; temps.tv_nsec = 0;

    printf("** vr = %d\n", vr);
    affThread(vr,"Debut de thread");
    sprintf(buf, "identite = %d.%d\n", getpid(), pthread_self());
    affThread(vr, buf);
    sprintf(buf, "je travaille sur la socket de service %d\n",hSockServ);
    affThread(vr, buf);

/* 1.Reception d'un message client */
    tailleMsgRecu = 0;
}

```

```

do
{
    if ( (nbreBytesRecus = recv(hSockServ,
        ((char *)msgClient) + tailleMsgRecu,
        LONG_STRUCT_CLI-tailleMsgRecu, 0))
        == -1)           /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSockServ); /* Fermeture de la socket */
        exit(1);
    }
    else tailleMsgRecu += nbreBytesRecus;
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
       tailleMsgRecu <LONG_STRUCT_CLI );

affThread(vr, "Recv socket OK");
sprintf(buf, "Demande recue pour le client = %s\n", msgClient->nom);
affThread(vr, buf);

/* 2. Envoi de l'ACK du serveur au client */
sprintf(msgServeur,"Demande recue du client %s !!!", msgClient->nom);
affThread(vr, msgServeur);
afficheRequete(msgClient);

nanosleep(&temps, NULL); /* pthread_delay_np(&temps); */

if (send(hSockServ, msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSockServ); /* Fermeture de la socket */
    exit(1);
}
else affThread(vr, "Send socket OK");
close(hSockServ); /* Fermeture de la socket */

affThread(vr, "--fin du thread--");
pthread_exit(&vr);
return 0;
}

```

Un exemple d'exécution est évoqué ci-dessous. Trois clients (exécutables c sur copernic et sunray2) vont se connecter successivement au serveur (exécutable s sur sunray2). Leurs interactions vont s'enchevêtrer comme montré ici :

```
sunray> s
Thread principal serveur demarre
identite = 24607.414848
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.5
Bind adresse et port socket OK
Thread principal : en attente d'une connexion
Listen socket OK
Accept socket OK
Socket de service attribuee = 5
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 5
```

**th\_5> Debut de thread**

```
th_5> identite = 24607.4
```

```
th_5> je travaille sur la socket de service 5
```

```
Accept socket OK
Socket de service attribuee = 6
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 6
```

**th\_6> Debut de thread**

```
th_6> identite = 24607.5
```

```
th_6> je travaille sur la socket de service 6
```

```
Accept socket OK
Socket de service attribuee = 7
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 7
```

**th\_7> Debut de thread**

```
th_7> identite = 24607.6
```

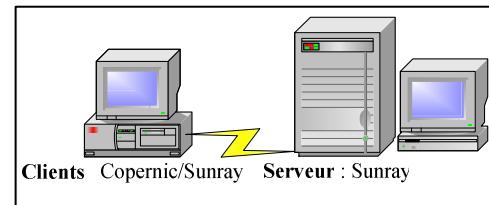
```
th_7> je travaille sur la socket de service 6
```

```
Taill msg = 88 et nbreBytes = 88
```

**th\_5> Recv socket OK**

```
th_5> Demande recue pour le client = client2
```

```
th_5> Demande recue du client client2 !!!
```



\*\*\* Requete d'un client \*\*\*

Numero de client : AS22

Nom = client2

Date du dernier achat : 5/1/1999

Numero de l'article demande : AR5554

et son prix : 3250

Coefficient de reduction = 0.000

Fourni ? = 0

Taill msg = 88 et nbreBytes = 88

**th\_6> Recv socket OK**

th\_6> Demande recue pour le client = **client3**

th\_6> Demande recue du client client3 !!!

\*\*\* Requete d'un client \*\*\*

Numero de client : AS3434

Nom = client3

Date du dernier achat : 9/12/1998

Numero de l'article demande : AR875

et son prix : 340

Coefficient de reduction = 0.000

Fourni ? = 0

**th\_5> Send socket OK**

th\_5> --fin du thread--

Taill msg = 88 et nbreBytes = 88

**th\_7> Recv socket OK**

th\_7> Demande recue pour le client = **client1**

th\_7> Demande recue du client client1 !!!

\*\*\* Requete d'un client \*\*\*

Numero de client : AS111

Nom = client1

Date du dernier achat : 12/3/1999

Numero de l'article demande : AR3243

et son prix : 2000

Coefficient de reduction = 0.000

Fourni ? = 0

**th\_7> Send socket OK**

th\_7> --fin du thread--

**th\_6> Send socket OK**

th\_6> --fin du thread--

**\*INTERRUPT\***

sunray>

### **Remarque**

On peut souhaiter maîtriser de manière très directive le nombre de threads lancés et, par voie de conséquence, le nombre de clients qui seront traités simultanément (dans le but pas exemple de ne pas surcharger le serveur). Classiquement, on peut pour cela utiliser un **tableau de handles de sockets** : tout thread créé trouvera son handle de socket dans ce tableau. Si le tableau est plein, la connexion sera refusée. Sinon, on place le handle de la

---

socket dupliquée dans la première place libre du tableau. Le thread recevra comme paramètre l'indice du tableau le concernant. Schématiquement :

### TCPITER05.C

```
/* TCPITER05.C
- Claude Vilvens -
*/
#include <pthread.h>
...
#define NBRE_MAX_CLIENTS 2
...
pthread_t threadHandle;

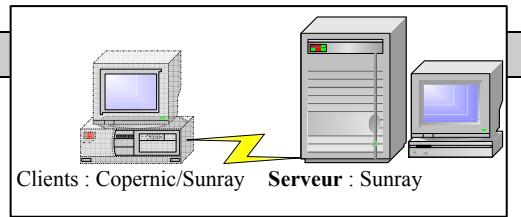
int hSocketEcoute, /* Handle de la socket d'ecoute*/
      hSocketDupliquee,
      hSocketService[NBRE_MAX_CLIENTS];
      /* Handle de la socket de service connectee au client */
int cpt = 0; /* Compteur de clients acceptés */

int main()
{
    ...
    int posLibre;

    /* 1. Creation de la socket */
    ...
    /* 2. Acquisition des informations sur l'ordinateur local */
    ...
    /* 3. Préparation de la structure sockaddr_in */
    ....
    /* 4. Le système prend connaissance de l'adresse et du port de la socket */
    ...
    /* 4bis. Initialisation du tableau des sockets de service */
    for (i=0; i<NBRE_MAX_CLIENTS; i++) hSocketService[i]=-1;

    do
    {
        /* 5. Mise à l'écoute d'une requête de connexion */
        puts("Thread principal : en attente d'une connexion");
        if (listen(hSocketEcoute,SOMAXCONN) == -1)
        {
            printf("Erreur sur le listen de la socket %d\n", errno);
            close(hSocketEcoute); /* Fermeture de la socket */
            exit(1);
        }
        else printf("Listen socket OK\n");

        /* 6. Acceptation d'une connexion et lancement d'un thread pour ce client */
        tailleSockaddr_in = sizeof(struct sockaddr_in);
    }
}
```



```

if ( (hSocketDupliquee =
      accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) ) == -1)
{
    printf("Erreur sur l'accept de la socket %d\n", errno);
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else if (cpt>=NBRE_MAX_CLIENTS)
{
    puts("Connexion refusee par le serveur - trop de clients !");
    strcpy(buf, "Connexion refusee par le serveur - trop de clients !");
    if (send(hSocketDupliquee, buf, strlen(buf)+1, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketDupliquee); /* Fermeture de la socket */
        exit(1);
    }
    else puts("Send socket refus OK");
    continue;
}
else printf("Accept socket OK\n");

printf("Socket de service attribuee = %d\n", hSocketDupliquee);

/* Recherche de la première position libre posLibre – à faire ;-) */
...
hSocketService[posLibre] = hSocketDupliquee; cpt++; /* A protéger par mutex */
ret = pthread_create(&threadHandle, NULL, fctThread, (void *)posLibre);
puts("Thread secondaire lance !");
puts("Marquage pour effacement du thread secondaire");
ret = pthread_detach(threadHandle);
}

while(1);
/* 7. Fermeture des sockets */

...
puts("Fin du thread principal");
return 0;
}

/*
void afficheRequete(struct client *c) {...}

void * fctThread (void *param)
{
    ...
    int num = (int)param;
    int vr = num+1;

/* 1.Reception d'un message client */
tailleMsgRecu = 0;

```

```

do
{
    if ( (nbreBytesRecus = recv(hSocketService[num], msgClient + tailleMsgRecu,
        LONG_STRUCT_CLI-tailleMsgRecu, 0))
        == -1)           /* pas message urgent */
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketService[num]); exit(1);
    }
    else tailleMsgRecu += nbreBytesRecus;
    printf("Taill msg = %d et nbreBytes = %d \n",tailleMsgRecu, nbreBytesRecus);
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
tailleMsgRecu <LONG_STRUCT_CLI );
...
/* 2. Envoi de l'ACK du serveur au client */
...
if (send(hSocketService[num], msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService[num]); /* Fermeture de la socket */
    exit(1);
}
else affThread(vr, "Send socket OK");

affThread(vr, "--fin du thread--"); cpt--; hSocketService[num]=-1;      /* A protéger par
mutex */
pthread_exit(&vr);
return 0;
}

```

Voici un exemple de refus :

```

Listen socket OK
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 2
th_2> Debut de thread
th_2> identite = 44917.4

th_2> je travaille sur la socket de service 04

Connexion refusee par le serveur - trop de clients !
Send socket refus OK

```

Du côté client :

```
% c
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : Albert
Numero client : RF34
Numero d'article : SL342
Prix : 340
Date du dernier achat : 2/4/1999
Coefficient de reduction : 0.34
Send socket OK
Demande envoyee pour le client = Albert
Recv socket OK
Message recu en ACK = Connexion refusee par le serveur - trop de clients !
Socket client fermee
%
```

Evidemment, le client refusé doit quand même entrer toute sa commande pour se la voir refusée ! Dur, dur – il faudrait un message préalable de test de possibilité de connexion. A faire ... ;-)

### **3. Un peu de programmation asynchrone**

L'exemple précédent l'évoque bien : le serveur boucle éternellement jusqu'à ce qu'un violent CTRL-C l'interrompe.

On peut imaginer plus élégant : un client privilégié, connu sous le nom d"**"ADMIN"**, avec un numéro de client "PASSWORD" est reconnu par le serveur comme un administrateur. Le numéro d'article correspond alors à une commande pour ce serveur. Ici, une seule sera reconnue : "SHUTDOWN!", au but fort clair. Il s'agira donc pour le thread connecté sur la socket de service d'arrêter le serveur :

- ◆ dans la violence : avec, sous UNIX, l'envoi d'un signal SIGKILL par kill();
- ◆ avec diplomatie : en utilisant une instruction arrêtant le thread principal.

Pour notre exemple de serveur, la reconnaissance d'une demande de connexion par ADMIN-PASSWORD provoquera l'émission du signal SIGUSR1. La fonction de traitement déclenchée en cas de délivrance du signal sera la fonction gérant l'administration. Ici, la commande SHUTDOWN! donnera donc lieu à l'arrêt du thread principal.

## TCPIITER06.C

```
/* TCPIITER06.C
 - Claude Vilvens -
 */
```

```
#include <pthread.h>
...
#include <signal.h>

#include "tcpiter03.h"

void afficheRequete(struct client *c);
void * fctThread(void * param);

void handlerAdmin (int sig);

#define affThread(num, msg) printf("th_%d> %s\n", num, msg)

pthread_t threadHandle, threadPrincipal;
unsigned int canal;
char msgAdminClient[100];

int main()
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService; /* Handle de la socket de service connectee au client */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;

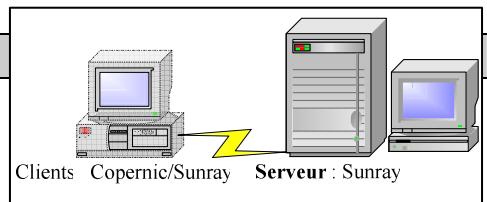
    unsigned int tailleSockaddr_in;
    int i;
    int ret;

    struct sigaction sigAct;

    puts("Thread principal serveur demarre");
    printf("identite = %d.%u\n", getpid(), pthread_self());
    threadPrincipal = pthread_self();

    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ret);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ret);

/* 1. Armement sur le signal SIGUSR1 */
    /* on enverra SIGUSR1 si on detecte un client administration */
    sigAct.sa_handler = handlerAdmin;
    sigAct.sa_flags = SA_SIGINFO;
    /* sigemptyset(&sigAct.sa_mask); */
```



```

if ( (ret=sigaction(SIGUSR1, &sigAct, 0)) == -1) perror("\nErreur de sigaction");

/* 2. Creation de la socket */
...
/* 3. Acquisition des informations sur l'ordinateur local */
...
/* 4. Préparation de la structure sockaddr_in */
...
/* 5. Le système prend connaissance de l'adresse et du port de la socket */
....
do
{
/* 6. Mise a l'ecoute d'une requete de connexion */
    puts("Thread principal : en attente d'une connexion");
    if (listen(hSocketEcoute,SOMAXCONN) == -1)
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");
/* 7. Acceptation d'une connexion et lancement d'un thread pour ce client */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
            accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");

    printf("Socket de service attribuee = %d\n", hSocketService);
    ret = pthread_create(&threadHandle, NULL, fctThread, (void*)hSocketService);
    puts("Thread secondaire lance !");
    puts("Marquage pour effacement du thread secondaire");
    ret = pthread_detach(threadHandle);
}
while(1);

/* 8. Fermeture des sockets */
    close(hSocketEcoute); /* Fermeture de la socket */
    printf("Socket serveur fermee\n");

    puts("Fin du thread principal");
    return 0;
}

/* ----- */

```

```

void afficheRequete(struct client *c) {...}

void * fctThread (void *param)
{
    int hSockServ = (int)param;
    int vr = hSockServ;
    char msgServeur [LONG_MSG_SERV];
    struct client *msgClient =
        (struct client *)malloc(sizeof(struct client));
    int tailleMsgRecu, nbreBytesRecus;
    char buf[100];

    printf("## vr = %d\n", vr);
    affThread(vr,"Debut de thread");
    sprintf(buf, "identite = %d.%d\n", getpid(), pthread_self());affThread(vr, buf);
    sprintf(buf, "je travaille sur la socket de service %d\n",hSockServ);affThread(vr, buf);

/* 1.Reception d'un message client */
    tailleMsgRecu = 0;
    do
    {
        if ( (nbreBytesRecus = recv(hSockServ, ((char*)msgClient) + tailleMsgRecu,
            LONG_STRUCT_CLI-tailleMsgRecu, 0))
            == -1)
        {
            printf("Erreur sur le recv de la socket %d\n", errno);
            close(hSockServ); /* Fermeture de la socket */
            exit(1);
        }
        else tailleMsgRecu += nbreBytesRecus;
        printf("Taill msg = %d et nbreBytes = %d \n",
            tailleMsgRecu, nbreBytesRecus);
    }
    while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
        tailleMsgRecu <LONG_STRUCT_CLI );

    affThread(vr, "Recv socket OK");

    if (strcmp(msgClient->nom, "ADMIN") == 0 &&
        strcmp(msgClient->numClient, "PASSWORD") == 0)
    {
        affThread(vr,"!!! Demande d'administration du serveur !!!");
        strcpy(msgAdminClient,msgClient->numArticle);
        kill (getpid(), SIGUSR1);
    }
    else
    {
        sprintf(buf, "Demande recue pour le client = %s\n", msgClient->nom);
        affThread(vr, buf);
    }
}

```

```

/* 2. Envoi de l'ACK du serveur au client */
    sprintf(msgServeur,"Demande recue du client %s !!!", msgClient->nom);
    affThread(vr, msgServeur);
    afficheRequete(msgClient);

    if (send(hSockServ, msgServeur, LONG_MSG_SERV, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSockServ); /* Fermeture de la socket */
        exit(1);
    }
    else affThread(vr, "Send socket OK");

    close(hSockServ); /* Fermeture de la socket */
    affThread(vr, "Socket connectee au client fermee");
    affThread(vr, "--fin du thread--");
    pthread_exit(&vr);
    return 0;
}

void handlerAdmin (int sig)
{
    puts("*** handler de terminaison");
    printf("Message admin = %s\n", msgAdminClient);
    if (strcmp(msgAdminClient, "SHUTDOWN!") == 0)
    {
        puts("**** Arret du serveur ****");
        puts("--fin immediate du process--");
        kill (getpid(), SIGKILL);
        /* pthread_cancel(threadPrincipal);*/
    }
}

```

Le client, pour sa part, ne posera pas de questions superflues si il a affaire à une demande d'accès d'administrateur :

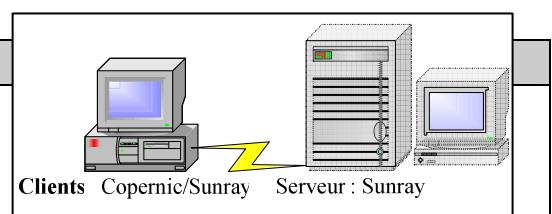
```

TCPCLI06.C
/* TCPCLI06.C
- Claude Vilvens -
*/
#include <stdio.h>
...
#include "tcpiter03.h"

void analyseErreur(int numErreur);

struct client msgClient;

```



```

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
        /* Structure de type sockaddr contenant les infos adresses -
           ici,structure adaptée au cas de TCP */
    unsigned int tailleSockaddr_in;
    int ret; /* valeur de retour */
    char msgServeur[LONG_MSG_SERV], buf[100];

/* 1. Création de la socket */
    ...
/* 2. Acquisition des informations sur l'ordinateur distant */
    ...
/* 3. Préparation de la structure sockaddr_in */
    ...
/* 4. Tentative de connexion */
    ...
/* 5. Envoi d'un message client */
    printf("Nom du client : ");gets(msgClient.nom);
        /*if (strcmp(msgClient,"FIN")==0) break;*/
    printf("Numero client : ");gets(msgClient.numClient);
    printf("Numero d'article : ");gets(msgClient.numArticle);
    if (strcmp(msgClient.nom, "ADMIN") ||
        strcmp(msgClient.numClient,"PASSWORD"))
    {
        printf("Prix : ");gets(buf);msgClient.montant = atoi(buf);
        printf("Date du dernier achat : ");
        gets(msgClient.dateDernierAchat);
        printf("Coefficient de reduction : ");
        gets (msgClient.coeffReduction);
        msgClient.fourni=0;
    }
    if (send(hSocket, &msgClient, LONG_STRUCT_CLI, 0) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

    printf("Demande envoyee pour le client = %s\n", msgClient.nom);

    if (strcmp(msgClient.nom, "ADMIN")==0 &&
        strcmp(msgClient.numClient,"PASSWORD")==0 &&
        strcmp(msgClient.numArticle, "SHUTDOWN!")==0)
        exit(0);
}

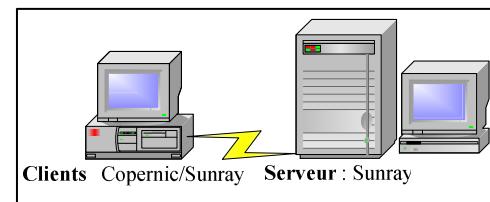
```

```
/* 6. Reception de l'ACK du serveur au client */
.....
/* 7. Fermeture de la socket */
.....
return 0;
}

/* -----
void analyseErreur(int numErreur)
{ ... }
```

Au niveau de l'exécution, supposons que nous lancions plusieurs clients, dont un sera un administrateur qui arrêtera le serveur :

```
% c
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : Albert
Numero client : 3
Numero d'article : 3
Prix : 3
Date du dernier achat : 3
Coefficient de reduction : 0.333
Send socket OK
Demande envoyee pour le client = Albert
Recv socket OK
Message recu en ACK = Demande recue du client Albert !!!
Socket client fermee
```



```
% c
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : Julien
Numero client : 2
Numero d'article : 2
Prix : 222
Date du dernier achat : 0.0222
Coefficient de reduction : 3
Send socket OK
Demande envoyee pour le client = Julien
Recv socket OK
Message recu en ACK = Demande recue du client Julien !!!
Socket client fermee
```

```
% c
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : Juliette
Numero client : 6
Numero d'article : 6
Prix : 666
Date du dernier achat : 6
Coefficient de reduction : 0.0666
Send socket OK
Demande envoyee pour le client = Juliette
Recv socket OK
Message recu en ACK = Demande recue du client Juliette !!!
Socket client fermee
```

```
% c
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : ADMIN
Numero client : PASSWORD
Numero d'article : SHUTDOWN!
Send socket OK
Demande envoyee pour le client = ADMIN
```

Le serveur réagira de la manière suivante :

```
sunray> s
Thread principal serveur demarre
identite = 11364.3223038392
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.5
Bind adresse et port socket OK
Thread principal : en attente d'une connexion
Listen socket OK
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 4
th_4> Debut de thread
```

---

```
th_4> identite = 11364.1073994112
th_4> je travaille sur la socket de service 4
Taill msg = 92 et nbreBytes = 92
th_4> Recv socket OK
th_4> Demande recue pour le client = Albert
th_4> Demande recue du client Albert !!!
*** Requete d'un client ***
Numero de client : 3
Nom = Albert
Date du dernier achat : 3
Numero de l'article demande : 3
    et son prix : 3
Coefficient de reduction = 0.333
Fourni ? = 0
Accept socket OK
Socket de service attribuee = 5
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 5
th_5> Debut de thread
th_5> identite = 11364.1074059648
th_5> je travaille sur la socket de service 5
th_4> Send socket OK
th_4> Socket connectee au client fermee
th_4> --fin du thread--
Taill msg = 92 et nbreBytes = 92
th_5> Recv socket OK
th_5> Demande recue pour le client = Julien
th_5> Demande recue du client Julien !!!
*** Requete d'un client ***
Numero de client : 2
Nom = Julien
Date du dernier achat : 0.0222
Numero de l'article demande : 2
    et son prix : 222
Coefficient de reduction = 3
Fourni ? = 0
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 4
th_4> Debut de thread
th_4> identite = 11364.1073994112
th_4> je travaille sur la socket de service 4
th_5> Send socket OK
```

---

```
th_5> Socket connectee au client fermee
th_5> --fin du thread--
Taill msg = 92 et nbreBytes = 92
th_4> Recv socket OK
th_4> Demande recue pour le client = Juliette
th_4> Demande recue du client Juliette !!!
*** Requete d'un client ***
Numero de client : 6
Nom = Juliette
Date du dernier achat : 6
Numero de l'article demande : 6
et son prix : 666
Coefficient de reduction = 0.0666
Fourni ? = 0
Accept socket OK
Socket de service attribuee = 5
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 5
th_5> Dебut de thread
th_5> identite = 11364.1074059648
th_5> je travaille sur la socket de service 5
th_4> Send socket OK
th_4> Socket connectee au client fermee
th_4> --fin du thread--
Taill msg = 92 et nbreBytes = 92
th_5> Recv socket OK
th_5> !!! Demande d'administration du serveur !!!
*** handler de terminaison
Message admin = SHUTDOWN!
**** Arret du serveur ****
--fin immediate du process--
```

**3.1.1.1.1.1 Killed**

A remarquer qu'**un port réservé à l'administrateur** est bien plus logique : une congestion du port "normal" de service n'empêcherait ainsi pas l'administration à distance !

#### **4. Une utilisation des mutex pour threads**

Nous allons imaginer que notre serveur enregistre les commandes reçues dans un fichier "commandes.clients" (tout ce qu'il y a de plus bas niveau). L'accès concurrent des différents threads est évident.

De plus, nous allons ajouter à la fonction d'administration SHUTDOWN! la fonction LIST! qui permettra au client administrateur d'obtenir les noms enregistrés dans le fichier (bien sûr, il pourrait demander les commandes complètes, avec LISTPLUS! – à faire ...).

Un mutex nous permettra ici d'assurer qu'un seul thread à la fois écrit dans le fichier des commandes. Cette écriture fera donc l'objet d'une section critique. Il faudra prévoir qu'un

client administrateur peut demander la liste des commandes. Dans ce cas, comme il faudra envoyer cette liste par le réseau, il faut que la fonction de traitement de SIGUSR1 connaisse le handle de la socket de service : un moyen (bête et méchant, mais cela fait parfois du bien ;)) consiste à écrire ce handle dans la variable globale hSockAdmin.

Petit raffinement complémentaire : si le serveur est lancé avec un argument de ligne de commande valant "recreecomm", le fichier des commandes sera effacé.

Cela donne (les tests sur les opérations mutex ont été "oubliés") :

### TCPITER07.C

```
/* TCPITER07.C
 - Claude Vilvens -
 */
```

```
#include <pthread.h>
...
#include "tcpiter03.h"

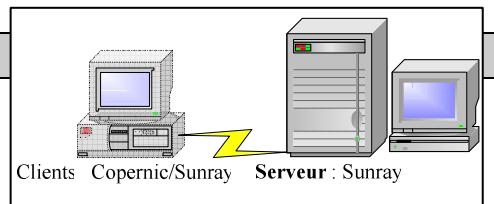
void afficheRequete(struct client *c);
void * fctThread(void * param);
void erreur (int ligne, char *typErr, int vr);
void handlerAdmin (int sig);

#define affThread(num, msg) printf("th_%d> %s\n", num, msg)

pthread_t threadHandle, threadPrincipal;
unsigned int canal;
int ret;
pthread_mutex_t mutexFichierCom;
char msgAdminClient[100];
int hSockAdmin;

int main(int argc, char **argv)
{
    int hSocketEcoute, /* Handle de la socket d'écoute*/
        hSocketService; /* Handle de la socket de service connectée au client */
    ...
    int hFile;
    ...
    pthread_mutex_init(&mutexFichierCom, NULL);

/* 1. Armement sur le signal SIGUSR1 */
    /* on enverra SIGUSR1 si on détecte un client administration */
    sigAct.sa_handler = handlerAdmin;
    sigAct.sa_flags = SA_SIGINFO;
    sigemptyset(&sigAct.sa_mask);
    if ((ret=sigaction(SIGUSR1, &sigAct, 0)) == -1)
        perror("\nErreur de sigaction");
```



```

/* 2. Reinitialisation eventuelle du fichier des commandes */
if (argc>1 && strcmp(*++argv, "recreecomm")==0)
    /* les parametres en ligne de commande sont pris en minuscules */
{
    puts("--- Recreation du fichier des commandes ---");
    hFile = open ("commandes.clients", O_CREAT | O_TRUNC,
                  S_IREAD | S_IWRITE);
    close(hFile);
    if (hFile == -1)
    {
        erreur(__LINE__, "Erreur ouverture du fichier", errno);
        exit(0);
    }
}

/* 3. Creation de la socket */
...
/* 4. Acquisition des informations sur l'ordinateur local */
...
/* 5. Preparation de la structure sockaddr_in */
...
/* 7. Le systeme prend connaissance de l'adresse et du port de la socket */
...
do
{
/* 8. Mise a l'ecoute d'une requete de connexion */
...
/* 9. Acceptation d'une connexion et lancement d'un thread pour ce client */
...
/* 10. Creation du thread de traitement */
...
}
while(1);

/* 10. Fermeture de la socket d'ecoute */
close(hSocketEcoute);
printf("Socket serveur fermee\n");

puts("Fin du thread principal");
return 0;
}

/* -----
void afficheRequete(struct client *c)
{
    printf("**** Requete d'un client ****\n");
    printf("Numero de client : %s\n", c->numClient);
    printf("Nom = %s\n", c->nom);
    printf("Date du dernier achat : %s\n", c->dateDernierAchat);
    printf("Numero de l'article demande : %s\n", c->numArticle);
}

```

```

printf(" et son prix : %d\n", c->montant);
printf("Coefficient de reduction = %s\n", c->coeffReduction);
printf("Fourni ? = %d\n", c->fourni);
}

void * fctThread (void *param)
{
/* 1. Recuperation du handle de la socket */
...
/* 2.Reception d'un message client */
...
if (strcmp(msgClient->nom, "ADMIN") == 0 &&
    strcmp(msgClient->numClient, "PASSWORD") == 0)
{
    affThread(vr, "!!! Acces aux fonctions d'administration !!!");
    strcpy(msgAdminClient, msgClient->numArticle);
    hSockAdmin = hSockServ;
    kill (getpid(), SIGUSR1);
    puts("== Apres emission du signal");
    vr = 0; pthread_exit(&vr);
    return 0;
}
else
{
    sprintf(buf, "Demande recue pour le client = %s\n", msgClient->nom);
    affThread(vr, buf);
}

/* 3. Enregistrement de la demande du client */
pthread_mutex_lock(&mutexFichierCom);
affThread(vr, "Enregistrement de la commande");
hFile = open ("commandes.clients", O_APPEND | O_RDWR);
if (hFile == -1)
{
    erreur(__LINE__, "Erreur ouverture du fichier", errno);
    exit(0);
}
if ( (ret=write(hFile, msgClient, sizeof (struct client))) == -1)
{
    erreur(__LINE__, "Erreur d'ecriture dans le fichier", errno);
    exit(0);
}
if ( (ret=close(hFile)) == -1)
{
    erreur (__LINE__, "Erreur de fermeture du fichier", errno);
    exit(0);
}

pthread_mutex_unlock(&mutexFichierCom);

```

```

/* 4. Envoi de l'ACK du serveur au client */
    ...
}

void handlerAdmin (int sig)
{
    int hFile, i, ret, cpt=0;
    char buf[100];
    struct client *msgClient = (struct client *)malloc(sizeof(struct client));
    puts("!!! Administration du serveur !!!");
    puts("/** handler d'administration **");
    printf("Message admin = %s\n", msgAdminClient);

/* 1. Shutdown ? */
    if (strcmp(msgAdminClient, "SHUTDOWN!") == 0)
    {
        puts("**** Arret du serveur ****");
        puts("--fin immediate du thread--");
        kill (getpid(), SIGKILL);
        /* ou pthread_cancel(threadPrincipal); - plus douloureux */
    }

/* 2. Liste des commandes ? Envoi au client */
    if (strcmp(msgAdminClient, "LIST!") == 0)
    {
        puts("**** Liste des commandes enregistrees ****");
        pthread_mutex_lock(&mutexFichierCom);
        hFile = open ("commandes.clients", O_RDONLY );
        if (hFile == -1)
        {
            erreur (__LINE__, "Erreur ouverture du fichier", errno);
            exit(0);
        }

        while ( ret=read(hFile, msgClient, sizeof (struct client)) )
        {
            if (ret == -1)
            {
                erreur (__LINE__, "Erreur de lecture dans le fichier",
                       errno);
                exit(0);
            }
            printf("%d.", ++cpt);
            afficheRequete(msgClient);
            if (send(hSockAdmin, msgClient->nom, LONG_MSG_SERV, 0) == -1)
            {
                printf("Erreur sur le send de la socket %d\n", errno);
                close(hSockAdmin); /* Fermeture de la socket */
                exit(1);
            }
        }
    }
}

```

```

        else puts( "Send socket OK");
    }
    if ( (ret=close(hFile)) == -1)
    {
        erreur (__LINE__, "Erreur de fermeture du fichier", errno);
        exit(0);
    }
    pthread_mutex_unlock(&mutexFichierCom);
    printf("%d. ADMINISTRATOR!\n", ++cpt);
    strcpy(msgClient->nom, "ADMINISTRATOR!");
    if (send(hSockAdmin, msgClient->nom, LONG_MSG_SERV, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSockAdmin); /* Fermeture de la socket */
        exit(1);
    }
    else puts("Send administrateur socket OK");
    close(hSockAdmin); /* Fermeture de la socket */
    puts("Socket connectee au client fermee");
}
}

void erreur (int ligne, char *typErr, int vr)
{
    printf("Erreur %s (%d) en %d\n", typErr, vr, ligne);
}

```

Le client est complété pour pouvoir recevoir la liste de clients enregistrés :

### TCPCLI07.C

```

/* TCPCLI07.C
- Claude Vilvens -
*/

```

```

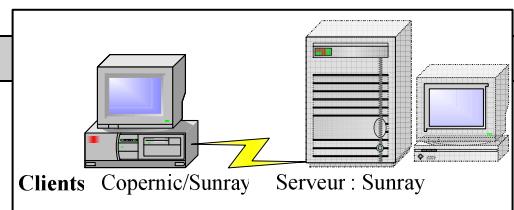
#include <stdio.h>
#include <stdlib.h> /* pour exit */
...
#include "tcpiter03.h"

void analyseErreur(int numErreur);

struct client msgClient;

int main()
{
    int hSocket; /* Handle de la socket */
    ...
    char msgServeur[LONG_MSG_SERV], buf[100];

```



```

printf("?? Taille d'un client = %d\n", sizeof(struct client));

/* 1. Création de la socket */
...
/* 2. Acquisition des informations sur l'ordinateur distant */
...
/* 3. Préparation de la structure sockaddr_in */
...
/* 4. Tentative de connexion */
...
/* 5. Envoi d'un message client */
    printf("Nom du client : "); gets(msgClient.nom);
    printf("Numero client : "); gets(msgClient.numClient);
    printf("Numero d'article : "); gets(msgClient.numArticle);
    if (strcmp(msgClient.nom, "ADMIN") ||
        strcmp(msgClient.numClient, "PASSWORD"))
    {
        printf("Prix : "); gets(buf); msgClient.montant = atoi(buf);
        printf("Date du dernier achat : "); gets(msgClient.dateDernierAchat);
        printf("Coefficient de reduction : "); gets (msgClient.coeffReduction);
        msgClient.fourni=0;
    }
    if (send(hSocket, &msgClient, LONG_STRUCT_CLI, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

    printf("Demande envoyee pour le client = %s\n", msgClient.nom);

if (strcmp(msgClient.nom, "ADMIN")==0 &&
    strcmp(msgClient.numClient,"PASSWORD")==0 &&
    strcmp(msgClient.numArticle, "SHUTDOWN!")==0)
exit(0);

if (strcmp(msgClient.nom, "ADMIN")==0 &&
    strcmp(msgClient.numClient,"PASSWORD")==0 &&
    strcmp(msgClient.numArticle, "LIST!")==0)
{
    puts(">>> Liste des clients enregistres <<<");
    do
    {
        if (recv(hSocket, msgServeur, LONG_MSG_SERV, 0) == -1)
        {
            printf("Erreur sur le recv de la socket %d\n", errno);
            close(hSocket); /* Fermeture de la socket */
            exit(1);
        }
    }
}

```

```

        else
        {
            printf("Recv socket OK\n");
            printf("Client recu = %s\n", msgServeur);
        }
    }
    while (strcmp(msgServeur,"ADMINISTRATOR!"));
    close(hSocket); /* Fermeture de la socket */
    printf("Socket client fermee\n");
    exit(1);
}

/* 6. Reception de l'ACK du serveur au client */
...
/* 7. Fermeture de la socket */
...
}

/* ----- */
void analyseErreur(int numErreur) {...}

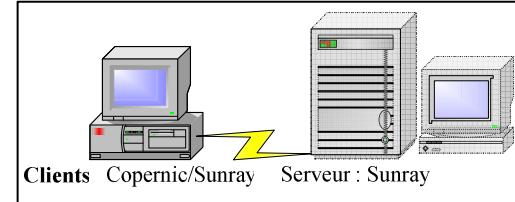
```

L'exécution du serveur donne des résultats similaires à ceux programme précédent :

```

sunray> s recreecomm
Thread principal serveur demarre
identite = 12515.3223038392
--- Recreation du fichier des commandes ---
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.5
Bind adresse et port socket OK
Thread principal : en attente d'une connexion
Listen socket OK
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 4
th_4> Debut de thread
...

```



Parmi les clients, un administrateur peut demander la liste des clients qui se sont connectés depuis la mise en route du serveur :

```

% c
?? Taille d'un client = 92
Creation de la socket OK

```

```
Acquisition infos host distant OK
Adresse IP = 10.59.4.5
Connect socket OK
Nom du client : ADMIN
Numero client : PASSWORD
Numero d'article : LIST!
Send socket OK
Demande envoyee pour le client = ADMIN
>>> Liste des clients enregistres <<<
Recv socket OK
Client recu = Julien
Recv socket OK
Client recu = Albert
Recv socket OK
Client recu = Veronique
Recv socket OK
Client recu = ADMINISTRATOR!
Socket client fermee
%
```

résultat tangible de ce qui a été fait du côté du serveur :

```
...
th_4> !!! Acces aux fonctions d'administration !!!
!!! Administration du serveur !!!
*** handler d'administration ***
Message admin = LIST!
**** Liste des commandes enregistrees ****
1.*** Requete d'un client ***
Numero de client : 6
Nom = Julien
Date du dernier achat : 6
Numero de l'article demande : 6
    et son prix : 6666
Coefficient de reduction = 0.066
Fourni ? = 0
Send socket OK
2.*** Requete d'un client ***
Numero de client : 3
Nom = Albert
Date du dernier achat : 3
Numero de l'article demande : 3
    et son prix : 333
Coefficient de reduction = 0.33
Fourni ? = 0
Send socket OK
3.*** Requete d'un client ***
Numero de client : 2
Nom = Veronique
Date du dernier achat : 2
```

```

Numero de l'article demande : 2
et son prix : 2
Coefficient de reduction = 0.22
Fourni ? = 0
Send socket OK
4. ADMINISTRATOR!
Send administrateur socket OK
Socket connectee au client fermee
==== Apres emission du signal
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
...

```

## **5. Un client Java**

Tant qu'il s'agit de faire communiquer deux applications Java, il n'y a pas de problème particulier. En effet, toutes deux utilisent les mécanismes d'encapsulation du langage, avec les mêmes conventions de représentation interne et d'optimisation.

Par contre, s'il s'agit de faire communiquer notre serveur, écrit en C, avec un client écrit en Java, des problèmes vont apparaître dès que l'on quitte le domaine de l'échange de simples chaînes de caractères. On se doute que la structure C client aura pour équivalent une classe Java. Mais il y a pire ...

En effet, d'une part, la représentation interne des champs d'une classe Java s'apparente à celle des structures C dans lesquelles les chaînes de caractères seraient représentées par des char \* au lieu de char [] – le zéro de fin de chaîne en moins ! Il faut donc, dans une optique de communication réseau, forcer la taille de chaque variable membre à ce qui est attendu du côté de la structure du C ! Ceci se traduira par l'utilisation de la méthode **setLength()** de la classe StringBuffer. On introduit ainsi artificiellement (du point de vue de Java) des caractères nuls, dont on se passe à l'affichage en usant de la méthode **trim()** de la classe String. On peut alors créer une suite de caractères concaténant l'ensemble des variables membres de la classe; c'est cette suite de caractères qui sera envoyée sur la socket.

De plus, les entiers ou, plus généralement, les grandeurs numériques, posent des problèmes de représentation interne différente de machine à machine. C'est pourquoi, de telles grandeurs sont plus sagement passées comme des chaînes de caractères, quitte à les convertir, si besoin est, en nombre.

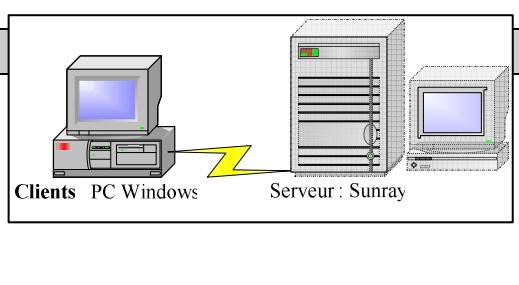
En tenant compte de ces remarques, le client Java, qui s'offrira un petit interface graphique, s'écrira :

### **commandeClient.java**

```

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

```



```

class client //implements Serializable
{
    public String numClient;
    public String nom;
    public String numArticle;
    public String dateDernierAchat;
    public String montant;
    public String coeffReduction;
    public String fourni;

    public client (String nc, String n, String na, String dda, String m, String cr, String f)
    {
        StringBuffer bufNC=new StringBuffer(20);
        bufNC.append(nc); bufNC.setLength(20); numClient = new String(bufNC);
        StringBuffer bufN=new StringBuffer(30);
        bufN.append(n); bufN.setLength(30);nom = new String(bufN);
        StringBuffer bufNA=new StringBuffer(15);
        bufNA.append(na); bufNA.setLength(15);numArticle = new String(bufNA);
        StringBuffer bufDDA=new StringBuffer(11);
        bufDDA.append(dda); bufDDA.setLength(11);dateDernierAchat = new String(bufDDA);
        StringBuffer bufM=new StringBuffer(10);
        bufM.append(m);bufM.setLength(10); montant=new String(bufM);
        StringBuffer bufCR=new StringBuffer(10);
        bufCR.append(cr); bufCR.setLength(10);coeffReduction = new String(bufCR);
        StringBuffer bufF=new StringBuffer(1);
        bufF.append(f);bufF.setLength(1);fourni = new String(bufF);
    }
    public void affiche()
    {
        System.out.println(numClient.trim() + ". " + nom.trim());
        System.out.println(dateDernierAchat.trim() + " - " + numArticle.trim());
        System.out.println(montant.trim() + " - " + coeffReduction.trim() + " - " + fourni.trim());
    }
}

// -----
public class commandeClient extends Frame implements ActionListener, WindowListener
{
    public final static int portEcouteServeur = 50000;
    public final static String nomServeur = "sunray2v440";

    TextField entreesForm[] = new TextField[7];
    Label ack;
    Button bEnvoyer;
    Socket cliSock = null; /* Initialisations indispensables */
    DataInputStream dis=null;
    DataOutputStream dos=null;

    public commandeClient (String titre, int longueur, int hauteur)
    {
        super(titre);
    }
}

```

```

setSize(longueur, hauteur);

// initConnexion(); OK pour une seule demande

setLayout(new GridLayout(8,2));

add(new Label("Numero de client : "));entreesForm[0]=new TextField("",20);
    add(entreesForm[0]);
add(new Label("Nom : "));entreesForm[1]=new TextField("",30);add(entreesForm[1]);
add(new Label("Numero d'article : "));entreesForm[2]=new TextField("",15);
    add(entreesForm[2]);
add(new Label("Date dernier achat : "));entreesForm[3]=new TextField("",11);
    add(entreesForm[3]);
add(new Label("Montant : "));entreesForm[4]=new TextField("",10);add(entreesForm[4]);
add(new Label("Coefcient de réduction : "));entreesForm[5]=new TextField("",10);
    add(entreesForm[5]);
add(new Label("Fourni ? (O/N) : "));entreesForm[6]=new TextField("",5);
    add(entreesForm[6]);
bEnvoyer = new Button("Envoyer");add(bEnvoyer); bEnvoyer.addActionListener(this);
ack = new Label("");add(ack);
addWindowListener(this);
}

void initConnexion()
{
try
{
    cliSock = new Socket(nomServeur, portEcouteServeur);
    System.out.println(cliSock.getInetAddress().toString());
    dis = new DataInputStream(cliSock.getInputStream());
    dos = new DataOutputStream(cliSock.getOutputStream());
}
catch (UnknownHostException ex)
{
    System.err.println("Erreur ! Host non trouvé [" + ex + "]");
}
catch (IOException ex)
{
    System.err.println("Erreur ! Pas de connexion ? [" + ex + "]");
}
if (cliSock==null || dis==null || dos==null) System.exit(1);
else System.out.println("---- socket et flux OK");
}

public static void main(String args[])
{
    commandeClient fenApp =
        new commandeClient("Client qui envoie une commande",400,400);
    fenApp.setVisible(true);
}

```

```

public void actionPerformed (ActionEvent e)
{
    byte b;
    initConnexion(); // on peut envoyer plusieurs commandes
    if (e.getActionCommand() == "Envoyer")
    {
        Integer entier = new Integer(entreesForm[4].getText());
        client clientCourant = new client(entreesForm[0].getText(),
                                         entreesForm[1].getText(), entreesForm[2].getText(),
                                         entreesForm[3].getText(), entreesForm[4].getText(),
                                         entreesForm[5].getText(), entreesForm[6].getText());
        System.out.println("Client créé et expédié : "); clientCourant.affiche();
        try
        {
            StringBuffer bufMsgClient = new StringBuffer(92);
            bufMsgClient.append(clientCourant.numClient);
            bufMsgClient.append(clientCourant.nom);
            bufMsgClient.append(clientCourant.numArticle);
            bufMsgClient.append(clientCourant.dateDernierAchat);
            bufMsgClient.append(clientCourant.montant);
            bufMsgClient.append(clientCourant.coeffReduction);
            bufMsgClient.append(clientCourant.fourni);
            String msgClient = new String(bufMsgClient);
            dos.write(msgClient.getBytes());
            if (clientCourant.nom.trim().compareTo("ADMIN") == 0 &&
                clientCourant.numClient.trim().compareTo("PASSWORD") == 0 &&
                clientCourant.numArticle.trim().compareTo("SHUTDOWN!") == 0 )
            {
                System.out.println("--- Arret du serveur et déconnexion du client");
                System.exit(0);
            }
        }
        catch (IOException ex)
        {
            System.err.println("Erreur d'I/O [ " + ex + "]");
        }
        System.out.println("Attente de la réponse du serveur");
        StringBuffer msgServeur = new StringBuffer();
        int cptCharServeur = 0;
        try
        {
            while((b = dis.readByte())!=((byte)'\\n')) msgServeur.append((char)b);
            System.out.println("Reçu du serveur : " + msgServeur.toString().trim());
        }
        catch (IOException ex)
        {
            System.err.println("Erreur d'I/O [ " + ex + "]");
        }
    }
}

```

```

public void windowClosing(WindowEvent e) {System.exit(0);}
public void windowOpened(WindowEvent e) { }
public void windowClosed(WindowEvent e) { }
public void windowDeiconified(WindowEvent e) { }
public void windowDeactivated(WindowEvent e) { }
public void windowActivated(WindowEvent e) { }
public void windowIconified(WindowEvent e) { }
}

```

La structure utilisée pour le serveur aura été modifiée comme suit :

### TCPITER03-JAVA.H

```

/* TCPITER03-JAVA.H
- Claude Vilvens -
*/
#ifndef TCPITER_H
#define TCPITER_H

#define EOC "END_OF_CONNEXION"

#define PORT 50000 /* Port d'écoute de la socket serveur */

struct client
{
    char numClient[20];
    char nom[30];
    char numArticle[15];
    char dateDernierAchat[11];
    char montant[10];
    char coeffReduction[10];
    char fourni;
};

#define LONG_STRUCT_CLI sizeof(struct client) /* Longueur des messages */
#define LONG_MSG_SERV 100

#endif

```

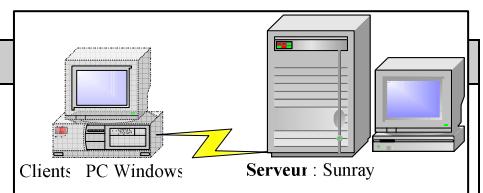
Le serveur a été ici simplifié pour ne reconnaître que la commande d'administration SHUTDOWN! :

### TCPITER07-JAVA.C

```

/* TCPITER07-JAVA.C
- Claude Vilvens -
*/
#include <pthread.h>
...

```



```
#include "tcpiter03-java.h"

void afficheRequete(struct client *c);
void * fctThread(void * param);
void erreur (int ligne, char *typErr, int vr);
void handlerAdmin (int sig);

#define affThread(num, msg) printf("th_%d> %s\n", num, msg)

pthread_t threadHandle, threadPrincipal;
unsigned int canal;
int ret;
pthread_mutex_t mutexFichierCom;
char msgAdminClient[100];
int hSockAdmin;

int main(int argc, char **argv)
{
    ...
    puts("Thread principal serveur demarre");
    printf("identite = %d.%u\n", getpid(), pthread_self());
    threadPrincipal = pthread_self();

    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &ret);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &ret);
    pthread_mutex_init(&mutexFichierCom, NULL);

/* 1. Armement sur le signal SIGUSR1 */
/* on enverra SIGUSR1 si on detecte un client administration */
    ...
/* 2. Reinitialisation eventuelle du fichier des commandes */
    ...
/* 3. Creation de la socket */
    ...
/* 4. Acquisition des informations sur l'ordinateur local */
    ....
/* 5. Preparation de la structure sockaddr_in */
    ...
/* 6. Le systeme prend connaissance de l'adresse et du port de la socket */
    ...
do
{
/* 7. Mise a l'ecoute d'une requete de connexion */
    ...
/* 8. Acceptation d'une connexion et lancement d'un thread pour ce client */
    ...
/* 10. Creation du thread de traitement */
    ret = pthread_create(&threadHandle, NULL, fctThread,
                        (void*)hSocketService);
    puts("Thread secondaire lance !");
}
```

```

        puts("Marquage pour effacement du thread secondaire");
        ret = pthread_detach(threadHandle);
    }
while(1);

/* 10. Fermeture de la socket d'ecoute */
    ...
}

/* -----
void afficheRequete(struct client *c) { ... }

void * fctThread (void *param)
{
/* 1. Recuperation du handle de la socket */
    ...
/* 2.Reception d'un message client */
    ...
    if (strcmp(msgClient->nom, "ADMIN") == 0 &&
        strcmp(msgClient->numClient, "PASSWORD") == 0)
    {
        ...
    }
    else
    {
        sprintf(buf, "Demande recue pour le client = %s\n", msgClient->nom);
        affThread(vr, buf);
    }
}

/* 3. Enregistrement de la demande du client */
    ...
/* 4. Envoi de l'ACK du serveur au client */
    sprintf(msgServeur,"Demande recue du client %s - enregistree !!!\n",
           msgClient->nom);
    affThread(vr, msgServeur);
    afficheRequete(msgClient);

    sleep(20);

    if (send(hSockServ, msgServeur, LONG_MSG_SERV, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSockServ); /* Fermeture de la socket */
        exit(1);
    }
    else affThread(vr, "Send socket OK");

    close(hSockServ); /* Fermeture de la socket */
    affThread(vr, "Socket connectee au client fermee");
    affThread(vr, "--fin du thread--");
}

```

```

pthread_exit(&vr);
return 0;
}

void handlerAdmin (int sig)
{
    int hFile, i, ret, cpt=0;
    char buf[100];
    struct client *msgClient =
        (struct client *)malloc(sizeof(struct client));
    puts("!!! Administration du serveur !!!");

    puts("*** handler d'administration ***");
    printf("Message admin = %s\n", msgAdminClient);
/* 1. Shutdown ? */
    if (strcmp(msgAdminClient, "SHUTDOWN!") == 0)
    {
        puts("**** Arret du serveur ****");
        puts("--fin immediate du thread--");
        kill (getpid(), SIGKILL);
    }

/* 2. Liste des commandes ? Envoi au client */
#ifndef _WIN32_WCE
    if (strcmp(msgAdminClient, "LIST!") == 0)
        ...
#endif
    printf("%d. ADMINISTRATOR!\n", ++cpt);
    strcpy(msgClient->nom, "ADMINISTRATOR!");
    if (send(hSockAdmin, msgClient->nom, LONG_MSG_SERV, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSockAdmin); /* Fermeture de la socket */
        exit(1);
    }
    else puts("Send administrateur socket OK");
    close(hSockAdmin); /* Fermeture de la socket */
    puts("Socket connectee au client fermee");
}

void erreur (int ligne, char *typErr, int vr)
{
    printf("Erreur %s (%d) en %d\n", typErr, vr, ligne);
}

```

Si donc un client a l'aspect suivant :

**Client qui envoie une commande**

Numero de client :	Cli4321
Nom :	Jules
Numero d'article :	AR432
Date dernier achat :	12/12/98
Montant :	23098
Coefcient de réduction :	0.65
Fourni ? (O/N) :	O
<b>Envoyer</b>	

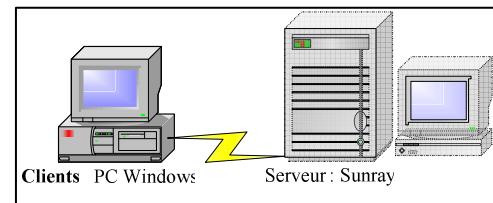
tandis qu'un deuxième se présente comme ceci :

**Client qui envoie une commande**

Numero de client :	Cli9745
Nom :	Philippe
Numero d'article :	AR874
Date dernier achat :	4/1/99
Montant :	23100
Coefcient de réduction :	0.89
Fourni ? (O/N) :	N
<b>Envoyer</b>	

on obtient sur la console Java, pour le premier client :

```
sunray2v440/10.59.4.5
---- socket et flux OK
Client créé et expédié :
Cli4321. Jules
12/12/98 – AR432
23098 - 0.65 - N
nom = Jules
numClient = Cli4321
numArticle = AR432
Attente de la réponse du serveur
Reçu du serveur : Demande recue du client Jules - enregistree !!!
```



Dans le cas d'un demande d'arrêt du serveur :

```
sunray2v440/10.59.4.5
---- socket et flux OK
Client créé et expédié :
PASSWORD. ADMIN
12 - SHUTDOWN!
1 - 1 - N
--- Arret du serveur et déconnexion du client
```

Le serveur, de son côté, réagira ainsi :

```
Thread principal serveur demarre
identite = 20199.3223038392
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.5
Bind adresse et port socket OK
Thread principal : en attente d'une connexion
Listen socket OK
Accept socket OK
Socket de service attribuee = 4
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
** vr = 4
th_4> Debut de thread
th_4> identite = 20199.1073994112
th_4> je travaille sur la socket de service 4
Accept socket OK
Socket de service attribuee = 5
Thread secondaire lance !
Marquage pour effacement du thread secondaire
Thread principal : en attente d'une connexion
Listen socket OK
```

```
** vr = 5
th_5> Debut de thread
th_5> identite = 20199.1074059648
th_5> je travaille sur la socket de service 5
Taill msg = 97 et nbreBytes = 97
th_4> Recv socket OK
----- Analyse de msgClient
th_4> Demande recue pour le client = Jules
th_4> Enregistrement de la commande
th_4> Demande recue du client Jules - enregistree !!!
*** Requete d'un client ***
Numero de client : Cli4321
Nom = Jules
Date du dernier achat : 12/12/98
Numero de l'article demande : AR432
    et son prix : 23098
Coefficient de reduction = 0.65
Fourni ? = 79
Taill msg = 97 et nbreBytes = 97
th_5> Recv socket OK
----- Analyse de msgClient
th_5> Demande recue pour le client = Philippe
th_5> Enregistrement de la commande
th_5> Demande recue du client Philippe - enregistree !!!
*** Requete d'un client ***
Numero de client : Cli9745
Nom = Philippe
Date du dernier achat : 4/1/99
Numero de l'article demande : AR874
    et son prix : 23100
Coefficient de reduction = 0.89
Fourni ? = 78
th_4> Send socket OK
th_4> Socket connectee au client fermee
th_4> --fin du thread--
th_5> Send socket OK
th_5> Socket connectee au client fermee
th_5> --fin du thread--
```

## 6. Une utilisation des variables de condition : le pool de threads

Bien que ceci ne nous apporte rien de plus au niveau de la programmation réseau, on ne peut passer sous silence un autre concept fondamental associé aux threads : nous voulons parler des *variables de condition*. Pour illustrer l'utilisation de celles-ci, nous allons envisager le cas d'un serveur multi-thread qui ne crée pas un thread à chaque connexion d'un client mais qui, au contraire, crée au démarrage un pool de threads en nombre fixé, ces threads étant en attente au moyen d'une variable de condition. L'intérêt de cette manière de faire est, comme il a déjà été écrit plus haut, de garder le contrôle sur le nombre de threads actifs, ce qui permet de limiter la charge du système de manière raisonnable (beaucoup de "légers", cela finit quand même par faire du "lourd" ...).

Les handles des threads créés initialement sont conservés dans un tableau (appelons-le *threadHandle*). Une variable globale *indiceCourant* a pour rôle de contenir la position dans ce tableau du handle du thread attribué au client qui vient de se connecter; initialement elle vaut -1. Tous les threads sont en attente sur une variable de condition (disons *condIndiceCourant*). Cette variable de condition est associée à l'événement caractérisé par le passage de *indiceCourant* à une valeur différente de -1, indiquant ainsi la tentative de prise en charge d'un client. "Tentative", car le nombre de threads est limité; si cette limite est atteinte, le client se verra envoyé pour toute réponse que sa connexion est refusée ! Inutile de dire que l'accès à la variable *indiceCourant* est réglementé par le mutex associé à la variable de condition. Les handles de sockets utilisés pour les communications sont conservées dans un tableau *hSocketConnectee*. Le dialogue, contrôlé par un flag *finDialogue*, est terminé à l'initiative du client, avec l'envoi du traditionnel *END\_OF\_CONNEXION*.

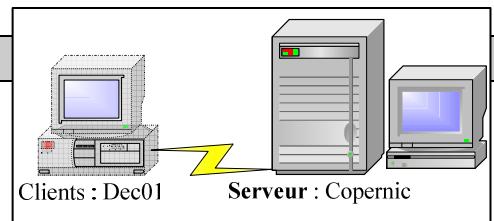
Notre serveur peut donc s'écrire ainsi :

### TCPVARCOND.C

```
/* tcpVarCond.c
-- Claude Vilvens --
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */
#include <time.h> /* pour select et timeval */
#include <pthread.h>

#define NB_MAX_CLIENTS 2 /* Nombre maximum de clients connectés */
#define EOC "END_OF_CONNEXION"
#define DOC "DENY_OF_CONNEXION"
#define PORT 50000 /* Port d'écoute de la socket serveur */
```



```

#define MAXSTRING 100 /* Longueur des messages */
#define affThread(num, msg) printf("th_%s> %s\n", num, msg)

pthread_mutex_t mutexIndiceCourant;
pthread_cond_t condIndiceCourant;
int indiceCourant=-1;
pthread_t threadHandle[NB_MAX_CLIENTS]; /* Threads pour clients*/
void * fctThread(void * param);
char * getThreadIdentity();
int hSocketConnectee[NB_MAX_CLIENTS]; /* Sockets pour clients*/

int main()
{
    int      hSocketEcoute, /* Socket d'ecoute pour l'attente */
            hSocketService;
    int      i,j,      /* variables d'iteration */
            retRecv;      /* Code de retour dun recv */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    int tailleSockaddr_in;
    int ret, * retThread;
    char msgServeur[MAXSTRING];

/* 1. Initialisations */
    puts("* Thread principal serveur demarre *");
    printf("identite = %d.%u\n", getpid(), pthread_self());

    pthread_mutex_init(&mutexIndiceCourant, NULL);
    pthread_cond_init(&condIndiceCourant, NULL);

    /* Si la socket n'est pas utilisee, le descripteur est a -1 */
    for (i=0; i<NB_MAX_CLIENTS; i++) hSocketConnectee[i] = -1;

/* 2. Creation de la socket d'ecoute */
    hSocketEcoute = socket(AF_INET,SOCK_STREAM,0);
    if (hSocketEcoute == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 3. Acquisition des informations sur l'ordinateur local */
    if ( (infosHost = gethostbyname("copernic"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
}

```

```

        memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
        printf("Adresse IP = %s\n",inet_ntoa(adresseIP));

/* 4. Préparation de la structure sockaddr_in */
        memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
        adresseSocket.sin_family = AF_INET;
        adresseSocket.sin_port = htons(PORT);
        memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 5. Le système prend connaissance de l'adresse et du port de la socket */
        if (bind(hSocketEcoute, (struct sockaddr *)&adresseSocket,
                 sizeof(struct sockaddr_in)) == -1)
        {
            printf("Erreur sur le bind de la socket %d\n", errno);
            exit(1);
        }
        else printf("Bind adresse et port socket OK\n");

/* 6. Lancement des threads */
        for (i=0; i<NB_MAX_CLIENTS; i++)
        {
            ret = pthread_create(&threadHandle[i],NULL,fctThread, (void*)i);
            printf("Thread secondaire %d lance !\n", i);
            ret = pthread_detach(threadHandle[i]);
        }

do
{
/* 7. Mise à l'écoute d'une requête de connexion */
    puts("Thread principal : en attente d'une connexion");
    if (listen(hSocketEcoute,SOMAXCONN) == -1)
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 8. Acceptation d'une connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if ( (hSocketService =
          accept(hSocketEcoute, (struct sockaddr *)&adresseSocket, &tailleSockaddr_in) )
        == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");
}

```

```

/* 9. Recherche d'une socket connectee libre */
printf("Recherche d'une socket connectee libre ... \n");
for (j=0; j<NB_MAX_CLIENTS && hSocketConnectee[j] !=-1; j++);

if (j == NB_MAX_CLIENTS)
{
    printf("Plus de connexion disponible \n");

    sprintf(msgServeur,DOC);
    if (send(hSocketService, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de refus %d \n", errno);
        close(hSocketService); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket refusee OK");
    close(hSocketService); /* Fermeture de la socket */
}
else
{
    /* Il y a une connexion de libre */
    printf("Connexion sur la socket num. %d \n", j);
    pthread_mutex_lock(&mutexIndiceCourant);
    hSocketConnectee[j] = hSocketService;
    indiceCourant=j;
    pthread_mutex_unlock(&mutexIndiceCourant);
    pthread_cond_signal(&condIndiceCourant);
}
}

while (1);

/* 10. Fermeture de la socket d'ecoute */
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee \n");

puts("Fin du thread principal");
return 0;
}

/* -----
void * fctThread (void *param)
{
    char * nomCli, *buf = (char*)malloc(100);
    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int vr = (int)(param), finDialogue=0, i, iCliTraite;
    int temps, retRecv;
    char * numThr = getThreadIdentity();
    int hSocketServ;
}

```

```

while (1)
{
/* 1. Attente d'un client à traiter */
    pthread_mutex_lock(&mutexIndiceCourant);
    while (indiceCourant == -1)
        pthread_cond_wait(&condIndiceCourant, &mutexIndiceCourant);
    iCliTraite = indiceCourant; indiceCourant=-1;
    hSocketServ = hSocketConnectee[iCliTraite];
    pthread_mutex_unlock(&mutexIndiceCourant);
    sprintf(buf,"Je m'occupe du numero %d ...", iCliTraite);affThread(numThr, buf);
/* 2. Dialogue thread-client */
    finDialogue=0;
    do
    {
        if ((retRecv=recv(hSocketServ, msgClient, MAXSTRING,0)) == -1)
        {
            printf("Erreur sur le recv de la socket connectee : %d\n", errno);
            close (hSocketServ); exit(1);
        }
        else
        if (retRecv==0)
        {
            sprintf(buf,"Le client est parti !!!"); affThread(numThr, buf);
            finDialogue=1;
            break;
        }
        else
        {
            sprintf(buf,"Message recu = %s\n", msgClient);
            affThread(numThr, buf);
        }
        if (strcmp(msgClient, EOC)==0)
        {
            finDialogue=1; break;
        }
    }

    sprintf(msgServeur,"ACK pour votre message :<%s>", msgClient);
    if (send(hSocketServ, msgServeur, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocketServ); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        sprintf(buf,"Send socket connectee OK\n");
        affThread(numThr, buf);
    }
}
while (!finDialogue);

```

```
/* 3. Fin de traitement */
    pthread_mutex_lock(&mutexIndiceCourant);
    hSocketConnectee[iCliTraite]=-1;
    pthread_mutex_unlock(&mutexIndiceCourant);
}
close (hSocketServ);
return (void *)vr;
}

char * getThreadIdentity()
{
    unsigned long numSequence;
    char *buf = (char *)malloc(30);

    numSequence = pthread_getsequence_np( pthread_self( ) );
    sprintf(buf, "%d.%u", getpid(), numSequence);
    return buf;
}
```

Le client n'est guère différent des précédents, si ce n'est qu'il détecte l'envoi d'un refus de connexion par le serveur :

## TCPVARCONDCLI.C

```
/* tcpVarCondCli.c
- Claude Vilvens -
Cr: 13/6/2001
Maj: 14/6/2008
*/
```

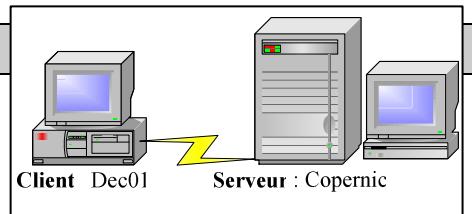
```
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

##include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format
                        reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#include "tcpiter.h"

#define DOC "DENY_OF_CONNEXION" /* → dans tcpiter.h */
```

```
int main()
```



```
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost; /* Infos sur le host : pour gethostbyname */
    struct in_addr adresseIP; /* Adresse Internet au format reseau */
    struct sockaddr_in adresseSocket;
    int tailleSockaddr_in;
    int ret; /* valeur de retour */

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int cpt=0;

/* 1. Création de la socket */
    hSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur distant */
    if ( (infosHost = gethostbyname("copernic"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host distant %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host distant OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
    printf("Adresse IP = %s\n",inet_ntoa(adresseIP));

/* 3. Préparation de la structure sockaddr_in */
    memset(&adresseSocket, 0, sizeof(struct sockaddr_in));
    adresseSocket.sin_family = AF_INET; /* Domaine */

    adresseSocket.sin_port = htons(PORT);
    memcpy(&adresseSocket.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Tentative de connexion */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    if (( ret = connect(hSocket, (struct sockaddr *)&adresseSocket, tailleSockaddr_in) ) == -1)
    {
        printf("Erreur sur connect de la socket %d\n", errno);
        switch(errno)
        {
            case EBADF : printf("EBADF - hsocket n'existe pas\n");
                          break;
            ...
            default : printf("Erreur inconnue ?\n");
        }
    }
}
```

```

        close(hSocket);
        exit(1);
    }
    else printf("Connect socket OK\n");

/* 5. Envoi d'un message client */
do
{
    printf("Message num %d a envoyer : ", cpt + 1);gets(msgClient);

    if (send(hSocket, msgClient, MAXSTRING, 0) == -1)
        /* pas message urgent */
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Send socket OK\n");

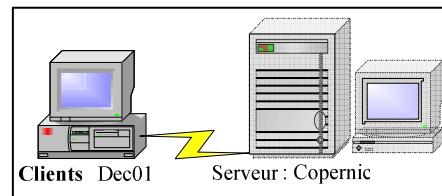
    printf("Message envoye = %s\n", msgClient);

    if (strcmp(msgClient,EOC))
    {
/* 6. Reception de l'ACK du serveur au client */
        if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
        {
            printf("Erreur sur le recv de la socket %d\n", errno);
            close(hSocket); /* Fermeture de la socket */
            exit(1);
        }
        else printf("Recv socket OK\n");
        printf("Message recu en ACK = %s\n", msgServeur);
        cpt++;
    }
}
while (strcmp(msgClient, EOC) && strcmp(msgServeur, DOC));

/* 9. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");
printf("%d messages envoyees !", cpt);
return 0;
}

```

Un exemple de dialogue multiple pourrait être (les messages sont volontairement simplistes – ils ont le mérite de trahir leur origine) :



le serveur (sur copernic)	les 3 clients (sur dec01)
<p>copernic.inpres.epl.prov-liege.be&gt; s  Thread principal serveur demarre  identite = 70230.3222999712  * Thread principal serveur demarre *  Creation de la socket OK  Acquisition infos host OK  Adresse IP = 10.59.5.9  Bind adresse et port socket OK  Thread secondaire 0 lance !  Thread secondaire 1 lance !  Thread principal : en attente d'une connexion  Listen socket OK  Accept socket OK  Recherche d'une socket connecteee libre ...  Connexion sur la socket num. 0  Thread principal : en attente d'une connexion  Listen socket OK  th_70230.2&gt; Je m'occupe du numero 0 ...  th_70230.2&gt; Message recu = <b>dec1-1</b> ←  th_70230.2&gt; Send socket connectee OK</p>	<p>% cli  Creation de la socket OK  Acquisition infos host distant OK  Adresse IP = 10.59.5.9  Connect socket OK  Message num 1 a envoyer : <b>dec1-1</b>  Send socket OK  Message envoye = dec1-1  Recv socket OK  Message recu en ACK = ACK pour votre message : &lt;dec1-1&gt;</p>
<p>Accept socket OK  Recherche d'une socket connecteee libre ...  Connexion sur la socket num. 1  Thread principal : en attente d'une connexion  Listen socket OK  th_70230.3&gt; Je m'occupe du numero 1 ...  th_70230.3&gt; Message recu = <b>dec2-1</b> ←  th_70230.3&gt; Send socket connectee OK  th_70230.3&gt; Message recu = <b>dec2-2</b> ←  th_70230.3&gt; Send socket connectee OK</p>	<p>% cli  Creation de la socket OK  Acquisition infos host distant OK  Adresse IP = 10.59.5.9  Connect socket OK  Message num 1 a envoyer : <b>dec2-1</b>  Send socket OK  Message envoye = dec2-1  Recv socket OK  Message recu en ACK = ACK pour votre message : &lt;dec2-1&gt;  Message num 2 a envoyer : <b>dec2-2</b>  Send socket OK  Message envoye = dec2-2  Recv socket OK  Message recu en ACK = ACK pour votre message : &lt;dec2-2&gt;</p>
<p>Accept socket OK  Recherche d'une socket connecteee libre ...  Plus de connexion disponible  Send socket refusee OK</p>	<p>% cli  Creation de la socket OK  Acquisition infos host distant OK  Adresse IP = 10.59.5.9  Connect socket OK  Message num 1 a envoyer : dec3-1  Send socket OK  Message envoye = dec3-1</p>

	<p>Recv socket OK Message recu en ACK = <b>DENY_OF_CONNEXION</b> Socket client fermee 1 messages envoyes !%</p>
Thread principal : en attente d'une connexion Listen socket OK th_70230.2> Message recu = <b>dec1-2</b>  th_70230.2> Send socket connectee OK th_70230.2> Message recu = <b>dec1-3</b>  th_70230.2> Send socket connectee OK	<p>Message num 2 a envoyer : dec1-2 Send socket OK Message envoye = <b>dec1-2</b> Recv socket OK Message recu en ACK = ACK pour votre message : &lt;dec1-2&gt; Message num 3 a envoyer : dec1-3 Send socket OK Message envoye = <b>dec1-3</b> Recv socket OK Message recu en ACK = ACK pour votre message : &lt;dec1-3&gt;</p>
th_70230.3> Message recu = <b>dec2-3</b>  th_70230.3> Send socket connectee OK	<p>Message num 3 a envoyer : <b>dec2-3</b> Send socket OK Message envoye = dec2-3 Recv socket OK Message recu en ACK = ACK pour votre message : &lt;dec2-3&gt;</p>
th_70230.2> Message recu = <b>END_OF_CONNEXION</b>	<p>Message num 4 a envoyer : <b>END_OF_CONNEXION</b> Send socket OK Message envoye = END_OF_CONNEXION Socket client fermee 3 messages envoyes !%</p>
th_70230.3> Message recu = <b>dec2-4</b>  th_70230.3> Send socket connectee OK th_70230.3> Message recu = <b>dec2-5</b>  th_70230.3> Send socket connectee OK	<p>Message num 4 a envoyer : <b>dec2-4</b> Send socket OK Message envoye = dec2-4 Recv socket OK Message recu en ACK = ACK pour votre message : &lt;dec2-4&gt; Message num 5 a envoyer : <b>dec2-5</b> Send socket OK Message envoye = dec2-5 Recv socket OK Message recu en ACK = ACK pour votre message : &lt;dec2-5&gt;</p>
Accept socket OK Recherche d'une socket connectee libre ... Connexion sur la socket num. 1 Thread principal : en attente d'une connexion Listen socket OK th_70230.2> <del>Je m'occupe du numero 1 ...</del> th_70230.2> Message recu = <b>dec3bis-1</b>	<p>% cli Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.5.9 Connect socket OK Message num 1 a envoyer : <b>dec3bis-1</b> Send socket OK</p>

th_70230.2> Send socket connectee OK th_70230.2> Message recu = <b><i>dec3bis-2</i></b>	Message envoye = dec3bis-1 Recv socket OK Message recu en ACK = ACK pour votre message : <dec3bis-1> Message num 2 a envoyer : <b><i>dec3bis-2</i></b> Send socket OK Message envoye = dec3bis-2 Recv socket OK Message recu en ACK = ACK pour votre message : <dec3bis-2>
th_70230.3> Message recu = <b><i>dec2-6</i></b> th_70230.3> Send socket connectee OK	Message num 6 a envoyer : <b><i>dec2-6</i></b> Send socket OK Message envoye = dec2-6 Recv socket OK Message recu en ACK = ACK pour votre message : <dec2-6>
Accept socket OK Recherche d'une socket connectee libre ... Plus de connexion disponible Send socket refusee OK	% cli Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.5.9 Connect socket OK Message num 1 a envoyer : dec1bis-1 Send socket OK Message envoye = dec1bis-1 Recv socket OK Message recu en ACK = <b>DENY_OF_CONNEXION</b> Socket client fermee 1 messages envoyees !%
Thread principal : en attente d'une connexion Listen socket OK th_70230.2> Message recu = <b><i>END_OF_CONNEXION</i></b>	Message num 3 a envoyer : <b><i>END_OF_CONNEXION</i></b> Send socket OK Message envoye = END_OF_CONNEXION Socket client fermee 2 messages envoyees !%
Accept socket OK Recherche d'une socket connectee libre ... Connexion sur la socket num. 1 Thread principal : en attente d'une connexion Listen socket OK th_70230.2> <del>J'en m'occupe du numero 1 ...</del> th_70230.2> Message recu = <b><i>dec1bis-1</i></b>	% cli Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.5.9 Connect socket OK Message num 1 a envoyer : <b><i>dec1bis-1</i></b> Send socket OK Message envoye = dec1bis-1 Recv socket OK Message recu en ACK = ACK pour votre message : <dec1bis-1> Message num 2 a envoyer : <b><i>dec1bis-2</i></b> Send socket OK

<b>END_OF_CONNEXION</b>	Message envoyé = dec1bis-2 Recv socket OK Message reçu en ACK = ACK pour votre message : <dec1bis-2> Message num 3 à envoyer : <b>END_OF_CONNEXION</b> Send socket OK Message envoyé = END_OF_CONNEXION Socket client fermée 2 messages envoyés !%
th_70230.3> Message reçu = <b>END_OF_CONNEXION</b>  copernic.inpres.epl.prov-liege.be>	Message num 7 à envoyer : <b>END_OF_CONNEXION</b> Send socket OK Message envoyé = END_OF_CONNEXION Socket client fermée 6 messages envoyés !%



Après cette pièce maîtresse, une petite récréation. Les adresses IP sont encore pour nous une notion assez vague. Remédions à cette carence !

## V. L'adressage IP et ses fonctions d'utilisation



*Rassemblons des faits pour nous donner des idées.*

(Buffon, Histoire naturelle)

### 1. Les adresses IP

Pour pouvoir communiquer, chaque machine doit posséder une adresse logique. Les adresses du protocole IP sont actuellement codées sur **32 bits** : c'est la norme **IPv4**. Une partie de ces bits est vouée à l'adresse du réseau, le reste correspond à l'adresse de la machine au sein du réseau. On s'en doute, un gros réseau devra utiliser de nombreux bits pour repérer de manière unique chacune de ses machines. Par conséquent, il ne pourra consacrer que peu de bits à son adresse elle-même : il faudra donc lui octroyer une adresse courte.

Pour faire face à l'extension d'Internet, l'IETF (Internet Engineering Task Force) prévoit une norme **IPv6** ou IPng (Next Generation). Cette norme sera utilisée sur les réseaux haute vitesse comme les réseaux ATM. Les adresses sont codées dans un tableau de 4 éléments à 32 bits, soit sur **128 bits**. La compatibilité vers le bas est assurée : les adresses IPv4 sont codées dans les 32 bits de poids le plus fort, le reste étant complété par un préfixe fixe.

### 2. L'adressage standard

Une adresse IP comporte :

- ◆ un **netid** = adresse du réseau sur 8/16/24 bits selon les classes (voir plus loin);
- ◆ un **hostid** = adresse de l'ordinateur au sein du réseau (différent de tous 0 et de tous 1, valeurs particulières réservées au broadcasting).

31-24	23-16	15-8	7-0
netid		hostid	

On obtient donc ainsi un entier codé sur 4 bytes. Mais, en pratique, on utilise la **notation décimale pointée** au lieu de l'hexadécimale. Par exemple :

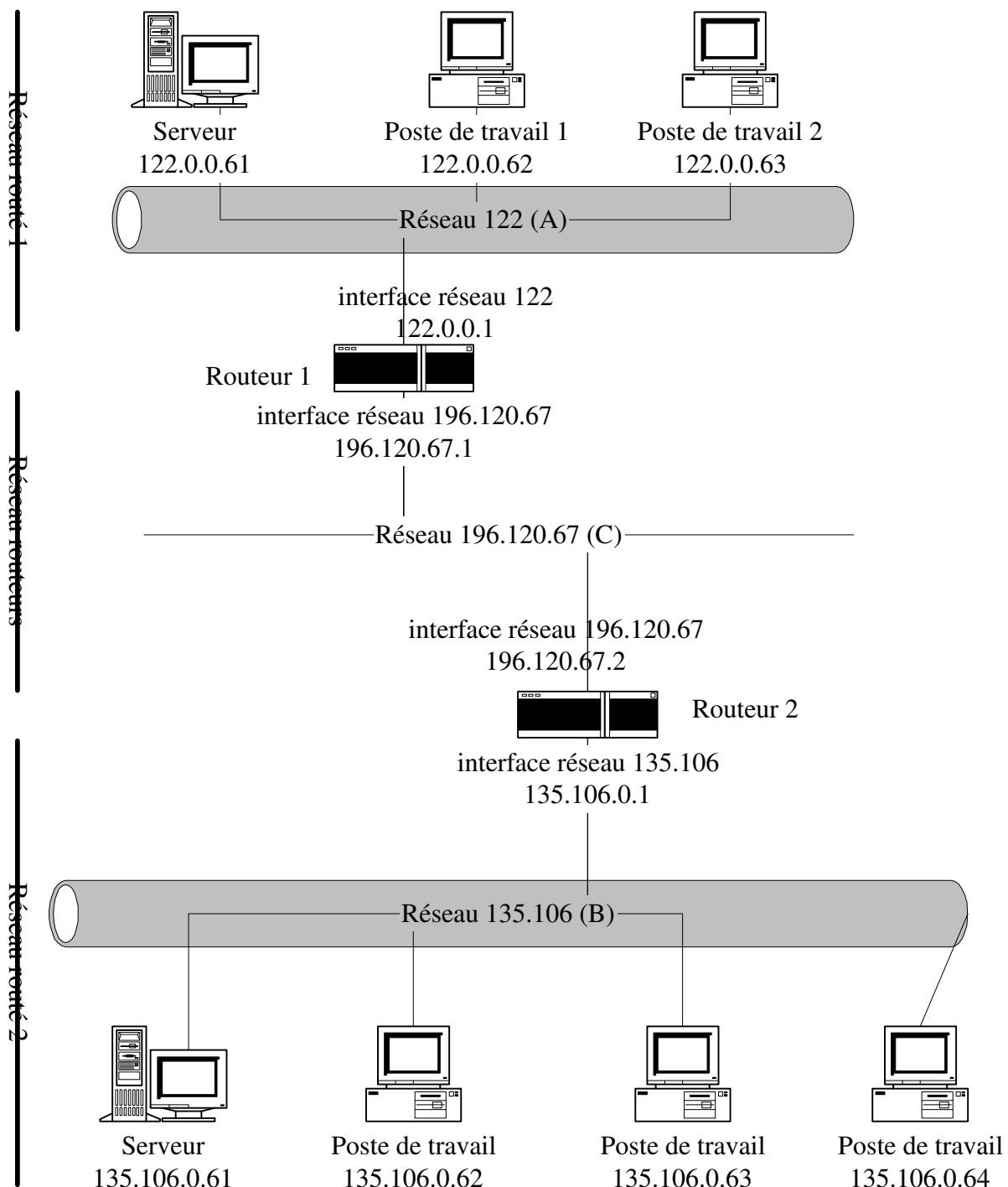
C0290614 -> 192.41.6.20

Il faut savoir que certaines adresses sont réservées à un usage particulier :

- ◆ **127.0.0.1** (127.xx.yy.zz) = l'adresse locale de l'ordinateur local (**localhost**) pour rebouclage (loopback);
- ◆ **0.0.0.0** = "cet" ordinateur = sert uniquement au démarrage d'un ordinateur, lorsqu'il communique pour obtenir son adresse IP – une fois cette adresse obtenue, l'ordinateur ne se sert plus de cette adresse 0.0.0.0.;

- ◆ **255.255.255.255** (tout à 1) : pour une diffusion limitée sur le réseau d'attachement (typiquement, le LAN).

On peut donc imaginer l'exemple suivant:



On remarquera que le réseau de routeurs, représentant la connexion étendue entre ceux-ci, nécessite un netid (ici, 196.120.67) : de cette manière, les deux routeurs peuvent y recevoir un hostid unique.

De plus, deux cas de figure sont à épingler :

- ◆ **netid = 0** : réseau d'attachement;
- ◆ **hostid = tous les bits à 1** : diffusion vers tous les ordinateurs d'un réseau particulier.

On peut, d'autre part, distinguer 3 types d'adresses IP :

- ◆ unicast : pour un seul ordinateur (TCP est orienté connexion et utilise ce type d'adresse);
- ◆ broadcast<sup>1</sup> : pour tous les ordinateurs du réseau (seulement en UDP);
- ◆ multicast : vers un groupe d'ordinateurs - il s'agit alors d'une adresse dite "de classe D" (1110 au poids le plus fort) – on utilise ce genre d'adresses dans les news par exemple (seulement en UDP – protocole IGMP).

Selon le nombre de bits consacrés au netid et aux hostids, on distingue des **classes** d'adresses IP :

	31-24	23-16	15-8	7-0
<b>A</b>	0	suite netid=7 bits		hostid = <b>24</b> bits
<b>B</b>	1	0	suite netid = 14 bits	hostid = <b>16</b> bits
<b>C</b>	1	1	0	suite netid = 21 bits      hostid = <b>8</b> bits
<b>D</b>	1	1	1	multicast group id = 28 bits
<b>E</b>	1	1	1	1
				27 bits – usages futurs

Les trois premières classes sont celles qui nous intéressent le plus, puisqu'elles sont utilisées pour désigner les réseaux et les hôtes dans un contexte TCP/IP. A l'évidence, la classe d'une adresse est attribuée selon les cas suivants :

<b>A</b>	un seul réseau avec un grand nombre d'ordinateurs
<b>B</b>	nombre moyen de réseaux avec un nombre moyen d'ordinateurs
<b>C</b>	beaucoup de réseaux avec peu d'ordinateurs par réseau

Rappelons que seuls les **RIR** (Regional Internet Registries), organismes officiels dépendant de l'**ICANN** (ex IANA), sont habilités à délivrer les numéros d'identification des réseaux (ce qui fait donc que *les adresses IP reflètent une localisation géographique*) . En termes plus quantitatifs :

	nombre de réseaux	nombre d'hôtes par réseau	plage du premier octet
<b>A</b>	126 nets	16777214 hosts	<b>1-126</b>
<b>B</b>	16384 nets	65534 hosts	<b>128-191</b>
<b>C</b>	2097152 nets	254 hosts	<b>192-223</b>

En ce qui concerne la 3<sup>ème</sup> colonne, on remarquera

- ◆ l'absence de l'adresse dont le 1<sup>er</sup> octet est 127 (localhost);
- ◆ la non utilisation des zones dont les bits sont tous à 0 ou tous à 1.

De manière plus détaillée, les plages d'adresses disponibles sont, d'après les références de CISCO<sup>1</sup> et la littérature classique :

<sup>1</sup> les Français disent "multi-diffusion"

<b>A</b>	1.0.0.0 -> 126.0.0.0 / 0.0.0.1 -> 127.255.255.254
<b>B</b>	128.1.0.0. -> 191.254.0.0 /128.0.0.1 -> 191.255.255.254
<b>C</b>	192.0.1.0. -> 223.255.254.0 / 192.0.0.1 -> 223.255.255.254
<b>D</b>	224.0.0.1 -> 239.255.255.255 / 224.0.0.1 -> 239.255.255.254
<b>E</b>	240.0.0.0 -> 254.255.255.255 / 240.0.0.1 -> 247.255.255.254

### 3. L'adressage avec sous-réseau

Dans le cas des grands réseaux, il est souvent utile de découper ce réseau en sous-réseaux, seule l'adresse du réseau étant importante pour l'extérieur. L'intérêt d'un tel découpage est que le routeur terminal doit seulement rechercher l'ordinateur visé dans le sous-réseau qu'il a déterminé dans l'adresse (grâce, comme nous allons le voir, au "masque de sous-réseau") au lieu de le rechercher dans l'ensemble du réseau. Bien sûr, la découpe en sous-réseaux reste cachée des routeurs extérieurs. Cette technique est surtout utilisée pour les réseaux de classe A et B.

Si l'on reprend l'exemple d'un réseau de classe B dont le netid est 128.1, on conçoit que le monde extérieur a juste besoin de connaître l'adresse du routeur principal, c'est-à-dire une seule entrée dans la table de routage.

A l'intérieur du réseau, on réalise l'adressage d'un ordinateur dans un sous-réseau en partageant les bits du hostid en deux parties :

- ◆ l'adresse du sous-réseau (**subnetid**);
- ◆ l'adresse de l'ordinateur (**hostid**).

Dans le cas d'un réseau de classe B, par exemple, il reste 16 bits pour la partie hostid. On peut, par exemple, la partager en deux parties de taille égale, soit 8 bits pour le subnetid et 8 bits pour le hostid proprement dit. On peut ainsi adresser :

- ◆  $2^8 - 2 = 254$  sous-réseaux;
- ◆  $2^8 - 2 = 254$  ordinateurs/réseaux.

car il faut décompter les deux adresses avec tous 0 (cet ordinateur) et tous 1 (broadcasting).

Le **masque de sous-réseau (subnet mask)** permet de définir le nombre de bits employés pour l'identification du sous-réseau et le nombre de bits utilisés pour identifier l'ordinateur.

Dans un masque de sous-réseau, les seuls bits à 0 sont ceux qui correspondent à l'identification de la machine hôte (ordinateur ou routeur).

**Un ET logique permet donc d'isoler l'adresse du sous-réseau** et donc, par conséquent, de déterminer si le paquet IP considéré est destiné à un hôte du réseau local ou d'un réseau distant. Les masques de sous-réseau par défaut sont :

A	255.0.0.0
B	255.255.0.0
C	255.255.255.0

---

<sup>1</sup> célèbre société constructrice de routeurs

Autrement dit, par défaut, il n'y a pas de sous-réseau ! Il n'existe qu'un seul masque par réseau.

exemples :

1) Si l'**adresse d'un ordinateur** est **132.1.10.100** avec pour masque de sous-réseau **255.255.255.0** ou **/24**, on peut conclure que :

- ◆ il s'agit d'une adresse de classe B;
- ◆ son adresse du réseau = 132.1;
- ◆ son adresse du sous-réseau = 10 (vu le masque précisé);
- ◆ son adresse de machine hôte = 100.

Une telle adresse sera encore désignée en abrégé par : **132.1.10.100/24**.

2) Si l'**adresse d'un ordinateur** est **132.1.225.100/18**, on peut conclure que :

- ◆ il s'agit d'une adresse de classe B;
- ◆ son adresse du réseau = 132.1;
- ◆ son adresse du sous-réseau = 3 (vu le masque précisé);
- ◆ son adresse de machine hôte = 8548.

## **4. La mise en œuvre d'une structure de sous-réseaux**

### **4.1 La détermination du masque de sous-réseau**

Pour fixer les idées, supposons devoir définir des sous-réseaux distincts au sein de notre entreprise, dont l'adresse de classe B a pour netid 135.106. Pour définir le masque de sous-réseau correspondant, il nous faut donc :

- ◆ déterminer le nombre de sous-réseaux (par exemple, 6);
- ◆ convertir ce nombre en binaire (ici, 110);
- ◆ considérer que le nombre de bits nécessaires donne le nombre de bits de poids le plus fort à positionner à 1 dans la zone hostid.

Pour notre exemple, s'il s'agit d'un réseau de classe B :

1111 1111	1111 1111	<b>1110</b> 0000	0000 0000
-----------	-----------	------------------	-----------

et le masque de sous-réseau est donc :

255.255.224.0

### **4.2 La détermination des subnetids**

Il faut se souvenir que deux netids sont à rejeter d'emblée : celui dont les bits sont à 1 (broadcast au sein du sous-réseau) et celui dont tous les bits sont à 0 ("ce sous-réseau"). Si  $n$  est le nombre de bits retenu pour le subnetid, on dispose donc potentiellement de  $2^n - 2$  subnetids. Pour notre exemple (pour rappel, le netid de notre exemple est 135.106) :

netid	subnetid + hostid		subnetid en clair
1000 0111	0110 1010	<b>0010 0000</b>	0000 0000
1000 0111	0110 1010	<b>0100 0000</b>	0000 0000
1000 0111	0110 1010	<b>0110 0000</b>	0000 0000
1000 0111	0110 1010	<b>1000 0000</b>	0000 0000
1000 0111	0110 1010	<b>1010 0000</b>	0000 0000
1000 0111	0110 1010	<b>1100 0000</b>	0000 0000

#### 4.3 La détermination des hostids

Il reste donc à déterminer les plages d'adresses pour les machines (ordinateurs ou routeurs). A nouveau, un hostid ne peut être constitué de bits tous à 0 ou tous à 1. Pour le sous-réseau 132.106.32.0, la première adresse valide pour une machine est donc :

1000 0111	0110 1010	<b>0010 0000</b>	0000 0001	135.106.32.1
-----------	-----------	------------------	-----------	--------------

tandis que la dernière est

1000 0111	0110 1010	<b>0011 1111</b>	1111 1110	135.106.63.254
-----------	-----------	------------------	-----------	----------------

ce que l'on aurait d'ailleurs pu déduire de l'id du sous-réseau suivant :

1000 0111	0110 1010	<b>0100 0000</b>	0000 0000	135.106.64.0
-----------	-----------	------------------	-----------	--------------

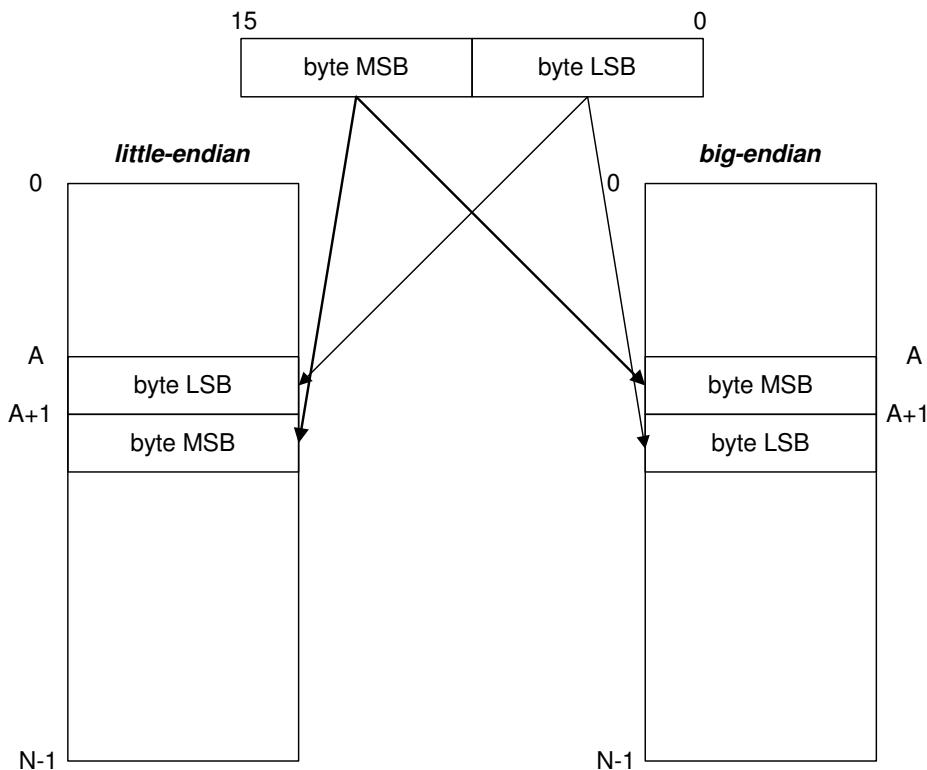
Les plages sont donc :

adresse IP de base			subnetid clair	plages
1000 0111	0110 1010	<b>0010 0000</b>	0000 0000	135.106.32.0 → 135.106.63.254
1000 0111	0110 1010	<b>0100 0000</b>	0000 0000	135.106.64.0 → 135.106.95.254
1000 0111	0110 1010	<b>0110 0000</b>	0000 0000	135.106.96.0 → 135.106.127.254
1000 0111	0110 1010	<b>1000 0000</b>	0000 0000	135.106.128.0 → 135.106.159.254
1000 0111	0110 1010	<b>1010 0000</b>	0000 0000	135.106.160.0 → 135.106.191.254
1000 0111	0110 1010	<b>1100 0000</b>	0000 0000	135.106.192.0 → 135.106.223.254

#### 5. Les adresses locales et les adresses réseau

On le sait, Internet regroupe des machines hétérogènes à tous les points de vue. L'un d'entre eux concerne la représentation des entiers. En effet,

- ◆ ou bien l'octet de poids le plus **fort** de l'entier est mémorisé en premier (en quelque sorte, on numérote de gauche à droite) – on parle encore de représentation **BigEndian** (cas des processeurs Motorola, Sun, HP);
- ◆ ou bien l'octet de poids le plus **faible** de l'entier est mémorisé en premier (en quelque sorte, on numérote de droite à gauche) – on parle encore de représentation **LittleEndian** (cas des processeurs Intel ou AXP Alpha).



Ceci pose un problème dans le cas particulier d'une adresse IP. En effet, si l'on considère l'adresse IP en notation pointée :

145.127.10.75 (=0x91.0x7F.0x0A.0x4B),

sa représentation locale sera sur une machine donnée sera :

- ◆ en Big Endian : 0x917F0A4B;
- ◆ en Little Endian : 0x4B0A7F91.

Les numéros de ports sont évidemment concernés par ces difficultés de représentations internes.

Il est clair qu'au niveau du réseau Internet, il a fallu choisir *un format commun* : le choix s'est porté sur la représentation Big Endian. Donc, le "**Network standard byte order**", c'est-à-dire l'ordre des octets standard, est un ordre "gros-boutiste" (*big endian*) : *les octets de poids fort du datagramme sont émis en premier lieu*.

Par conséquent, pour une machine Little Endian :

- ◆ la représentation **locale** est : 0x4B0A7F91
- ◆ la représentation **réseau** est : 0x917F0A4B

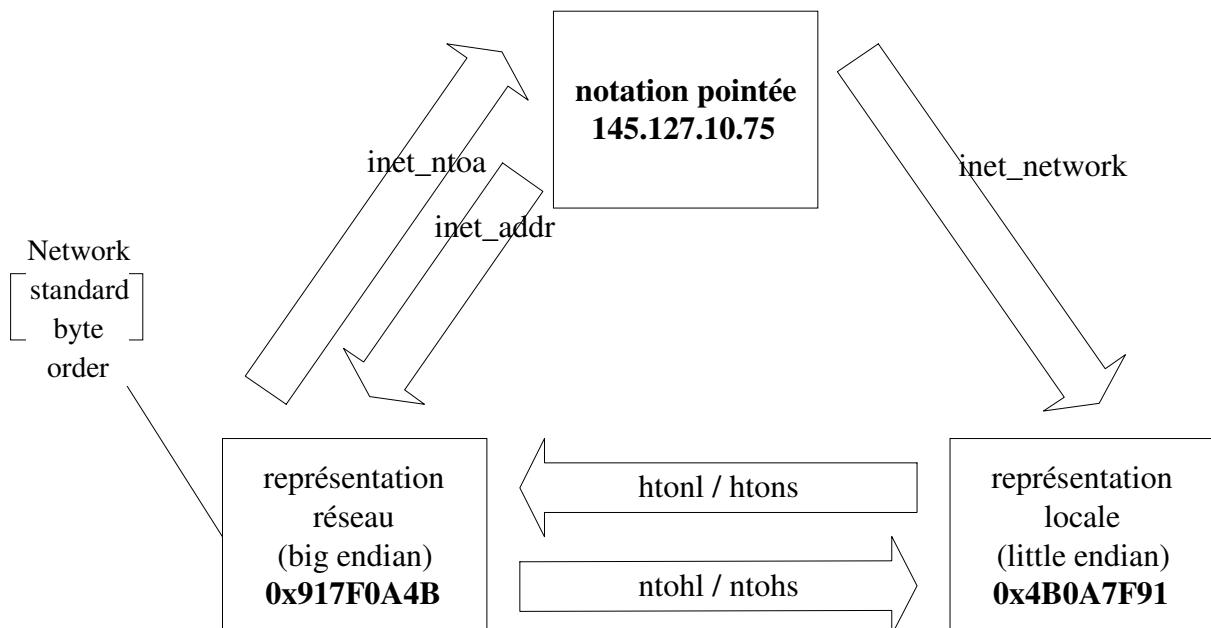
Pour un processeur Sparc de Sun, pas de problème : il est Big Endian. Mais, par contre, le processeur Pentium d'Intel est Little Endian : il faudra donc une conversion de la représentation standard (réseau) en représentation locale (ou vice-versa) pour que l'en-tête soit interprété correctement ! Des fonctions sont disponibles pour cela (prototypes dans netinet/in.h) :

```
unsigned long int htonl (<adresse locale - unsigned long int hostshort>);
unsigned short int htons (<numéro de port en local - unsigned short int>);
unsigned long int ntohl (<adresse réseau - unsigned long int>);
unsigned short int ntohs (<numéro de port en format réseau - unsigned short int>);
```

L'effet de ces fonctions se mémorise facilement si l'on pense à **network** et **host**, avec les suffixes **I** et **s** respectivement pour les adresses (long) et les ports (short).

D'autres fonctions permettent la conversion d'une chaîne de caractères en une adresse en format réseau ou local, avec éventuelle distinction de la composante réseau et de la composante machine :

```
int inet_addr (<adresse sous forme pointée - char *>);
/* adresse pointée -> adresse réseau */
int inet_network (<adresse sous forme pointée - char *>);
/* adresse pointée -> adresse locale */
char *inet_ntoa (struct in_addr);
/* adresse réseau -> adresse pointée */
int inet_netof (struct in_addr);
/* partie réseau de l'adresse réseau */
int inet_lnaof (struct in_addr);
/* partie adresse machine de l'adresse réseau */
struct in_addr inet_makeaddr (int, int);
/* adresse réseau à partir de l'adresse du réseau et de l'adresse locale de la machine
dans le réseau --- sans l'inclusion de arpa/inet.h, la valeur renournée est un int ! */
```



La constante

```
#define INADDR_NONE 0xffffffff /* -1 return */
```

est utilisée pour noter une adresse inexistante. Le petit programme suivant illustre les possibilités d'utilisation de ces diverses fonctions de conversion :

**ADRESSE01.C**

```
/* ADRESSE01.C
- Claude Vilvens -
*/
#include <netinet/in.h>
#include <arpa/inet.h>

#define TAILLE_ADRESSE 16 /* '*' + 3 points +0 de fin de chaine */

int main()
{
    char *adresseP = (char *)malloc(TAILLE_ADRESSE);
    int adresseR, adresseL;
    int b[4],i;
    struct in_addr inAdresseR;
    int composanteAdresseRMachine,composanteAdresseRReseau;

    printf("Adresse IP sous forme pointee : ");gets(adresseP);
    sscanf(adresseP, "%d.%d.%d.%d", &b[3], &b[2], &b[1], &b[0]);
    printf("Adresse pointee en hexadecimal : ");

    for (i=3; i>=0; i--)
    {
        printf("%02x", b[i]);
        if (i>0) printf(".");
        else printf("\n");
    }

    if ( (adresseR = inet_addr(adresseP)) != INADDR_NONE)
        printf("Adresse reseau = %08x\n", adresseR);
    else puts("Adresse invalide");

    if ( (adresseL = inet_network(adresseP)) != INADDR_NONE)
        printf("Adresse locale = %08x\n", adresseL);
    else puts("Adresse invalide");

    puts("** Conversions **");
    printf("Conversion reseau -> local : %08x => %08x\n", adresseR, ntohs(adresseR));
    printf("Conversion local -> reseau : %08x => %08x\n", adresseL, htons(adresseL));

    inAdresseR.s_addr = adresseR;
    if ( (adresseP = inet_ntoa(inAdresseR)) != (char *)INADDR_NONE)
        printf("Conversion reseau :-> pointee %08x => %s\n", adresseR, adresseP);
    else puts("Adresse invalide");
```

```

puts("** Composantes host et net de l'adresse **");
if ( (composanteAdresseRMachine = inet_Inaof(inAdresseR)) != INADDR_NONE)
    printf("Composante machine de l'adresse reseau : %08x\n",
           composanteAdresseRMachine);
else puts("Adresse invalide");
if ( (composanteAdresseRReseau = inet_netof(inAdresseR)) != INADDR_NONE)
    printf("Composante reseau de l'adresse reseau : %08x\n",
           composanteAdresseRReseau);
else puts("Adresse invalide");

puts("** Reconstruction de l'adresse reseau a partir de ses deux \
composantes");

inAdresseR =
    inet_makeaddr(composanteAdresseRReseau,composanteAdresseRMachine);
printf("reseau: %08x + host:%08x = %08x\n",
       composanteAdresseRReseau,composanteAdresseRMachine, inAdresseR.s_addr);
printf("      soit %s\n", (char *) inet_ntoa(inAdresseR));
return 0;
}

```

Ce qui donne sur boole (qui est little endian) :

```

boole>adr1
Adresse IP sous forme pointee : 123.45.10.34
Adresse pointee en hexadecimal : 7b.2d.0a.22
Adresse reseau = 220a2d7b
Adresse locale = 7b2d0a22
** Conversions **
Conversion reseau -> local : 220a2d7b => 7b2d0a22
Conversion local -> reseau : 7b2d0a22 => 220a2d7b
Conversion reseau :-> pointee 220a2d7b => 123.45.10.34
** Composantes host et net de l'adresse **
Composante machine de l'adresse reseau : 002d0a22
Composante reseau de l'adresse reseau : 0000007b
** Reconstruction de l'adresse reseau a partir de ses deux composantes
reseau: 0000007b + host:002d0a22 = 220a2d7b
      soit 123.45.10.34

```

## 6. Les pseudo-connexions et les sockets paires

### 6.1 Adresse locale et adresse distante (getsockname et getpeername)

Dans le cas d'un client se connectant à un serveur, on peut remarquer que la socket utilisée est en fait créée sur l'adresse<sup>1</sup> du serveur, c'est-à-dire de l'ordinateur distant. Cette adresse, avec le port utilisé, constitue ce que l'on appelle l'adresse distante ou *foreign address*. Pourtant, dans le contexte de l'échange de données, cette socket doit avoir aussi une adresse locale ou *local address*. Celle-ci peut être obtenue au moyen de la primitive :

---

<sup>1</sup> au sens large du terme, c'est-à-dire adresse IP avec port

```
int getsockname ( < handle de la socket – int>,
                  < adresse locale de la socket - struct sockaddr *>,
                  < longueur de la structure qui reçoit l'adresse– int *>);
```



**getsockname**

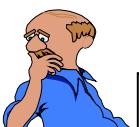
*erreurs*

Si la valeur de retour est  $-1$ , une erreur s'est produite (elle est  $0$  en cas de succès) et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define ENOBUFS 55	/* No buffer space available */ : il n'y a plus assez de ressources systèmes
#define EFAULT 14	/* Bad address */ : l'adresse de la structure <code>sockaddr</code> est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)

Implicitement, il s'établit donc une connexion entre ces deux adresses, connexion dont on se sert pour les `send()` et les `recv()`. Comme il n'est pas nécessaire de rappeler cette connexion à chaque opération de transfert, on parle encore de *pseudo-connexion* et les deux vues de la socket (locale et distante) sont appelées des "*sockets paires*" (*peer sockets*). On peut obtenir l'adresse distante associée à une socket au moyen de la primitive :

```
int getpeername ( < handle de la socket – int>,
                  < adresse distante de la socket - struct sockaddr *>,
                  < longueur de la structure qui reçoit l'adresse– int *>);
```



**getpeername**

*erreurs*

Si la valeur de retour est  $-1$ , une erreur s'est produite (elle est  $0$  en cas de succès) et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define ENOBUFS 55	/* No buffer space available */ : il n'y a plus assez de ressources systèmes
#define ENOTCONN 57	/* Socket is not connected */ : à votre avis ?
#define EFAULT 14	/* Bad address */ : l'adresse de la strcuture <code>sockaddr</code> est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)

Bien évidemment, ces deux fonctions ne fonctionnent valablement que pour des sockets connectées ...

## 6.2 Pour le client

Dans le cas du client, il connaît l'adresse distante mais pas forcément l'adresse locale de la socket qu'il utilise. Ainsi, sur dec01, on peut compléter le programme tcpcli02.c du chapitre II par :

<b>Complément à TCPCLI02.C</b>
<pre>struct sockaddr_in adresseSocket, adresseSocketLocale, adresseSocketDistante; ... /* connect */ ... printf("Adresse contenue dans la socket avec connect: %s\n",       inet_ntoa(adresseSocket.sin_addr)); if (getsockname(hSocket, &amp;adresseSocketLocale, &amp;tailleSockaddr_in) == -1) {     printf("Erreur sur le getsockname de la socket %d\n", errno);     close(hSocket); /* Fermeture de la socket */     exit(1); } else printf("Getsockname socket OK\n"); printf("Adresse de l'ordinateur local d'apres getsockname : %s /port %d\n",       inet_ntoa(adresseSocketLocale.sin_addr), ntohs(adresseSocketLocale.sin_port)); if (getpeername(hSocket, &amp;adresseSocketDistante, &amp;tailleSockaddr_in) == -1) {     printf("Erreur sur le getpeername de la socket %d\n", errno);     close(hSocket); /* Fermeture de la socket */     exit(1); } else printf("Getpeername socket OK\n"); printf("Adresse de l'ordinateur distant d'apres getpeername : %s /port %d\n",       inet_ntoa(adresseSocketDistante.sin_addr), ntohs(adresseSocketDistante.sin_port)); /* échange de données */</pre>

Cela donne :

Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.4.1 Connect socket OK Adresse du client : 10.59.4.131 Getsockname socket OK Adresse de l'ordinateur local d'apres getsockname : 10.59.4.131 /port 1894 Getpeername socket OK Adresse de l'ordinateur distant d'apres getpeername : 10.59.4.1 /port 5000
---

On peut obtenir confirmation du bien fondé des résultats en utilisant la commande **netstat**<sup>1</sup> :

---

<sup>1</sup> cette commande sera expliquée dans un chapitre ultérieur

| netstat –an

| donne :

Proto	Recv-Q	Send-Q	<b>Local Address</b>	<b>Foreign Address</b>	(state)
...					
tcp	0	0	10.59.4.131.1894	10.59.4.1.5000	ESTABLISHED

### 6.3 Pour le serveur

Le serveur a forcément du réaliser un bind() sur l'adresse locale; c'est donc cette dernière que fournira getsockname(). D'autre part, la socket de service, fournie par accept(), est connectée sur l'adresse du client qui s'est connecté : c'est donc cette dernière que fournira getpeername().

#### Complément à TCPITER02.C

```
...
struct sockaddr_in adresseSocket, adresseSocketLocale, adresseSocketDistante;

/* listen et accept */
printf("Adresse du client = %s\n", inet_ntoa(adresseSocket.sin_addr));

if (getsockname(hSocketService, &adresseSocketLocale, &tailleSockaddr == -1)
{
    printf("Erreur sur le getsockname de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    exit(1);
}
else printf("Getsockname socket OK\n");
printf("Adresse de l'ordinateur local d'apres getsockname : %s /port %d\n",
       inet_ntoa(adresseSocketLocale.sin_addr), ntohs(adresseSocketLocale.sin_port));

if (getpeername(hSocketService, &adresseSocketDistante, &tailleSockaddr) == -1)
{
    printf("Erreur sur le getpeername de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    exit(1);
}
else printf("Getpeername socket OK\n");
printf("Adresse de l'ordinateur distant d'apres getpeername : %s /port %d\n",
       inet_ntoa(adresseSocketDistante.sin_addr), ntohs(adresseSocketDistante.sin_port));
...
/* échange de données */
```

Sur boole, cela donne :

| Adresse du client = 10.59.4.131

| Getsockname socket OK

| Adresse de l'ordinateur local d'apres getsockname : 10.59.4.1 /port 5000

| Getpeername socket OK

| Adresse de l'ordinateur distant d'apres getpeername : 10.59.4.131 /port 1894

## 7. Le DNS

### 7.1 Noms et adresses

En pratique, il est évidemment peu ais  pour les utilisateurs d'utiliser les adresses num riques, m me sous forme point e. Aussi a-t-on tr s vite imagin  de pouvoir d signer les machines par un nom. Historiquement (c'est- -dire au d but de l'Internet), un simple fichier **Hosts.txt** suffisait   m m riser les correspondances noms/adresses, comme un annuaire t l phonique. Cependant, l'extension foudroyante du r seau des r seaux a tr s vite conduit ce fichier   grossir d mesur m nt, au point de le rendre quasiment inutilisable. C'est   ce moment que imagin  le DNS :

Le **DNS** (Domain Name System) est une bases de donn es distribu e qui associe   chaque adresse IP le nom de l'ordinateur ou du n ud.

L'utilisation de noms au lieu d'adresses reste ainsi possible sur l'ensemble d'Internet. Cette base de donn es impl mente en fait un sch ma de nommage hi rarchique qui se base sur la notion de domaine.

### 7.2 Les domaines

Au sommet de la hi rarchie se trouvent les grands domaines, dont le nom,   connotation g n rique ou g ographique, correspond :

- ◆ soit au domaine d'activit  au niveau international [g n rique 1] : les sigles sont
  - .com [commercial],
  - .org [organisations   but non lucratif],
  - .int [organisations internationales],
  - .net [fournisseurs de services r seau];
  - .store [magasin virtuel];
- ◆ soit   des r seaux d pendant de l'administration am ricaine [g n rique 2] : les sigles sont
  - .gov [gouvernement et administrations],
  - .mil [militaire],
  - .edu [ ducatif - universit s];
- ◆ soit au pays pour les r seaux nationaux [g ographique] : on utilise les codes ISO 3166   deux lettres du pays (.be, .fr, .uk, .us, .jp, ...).

Il existe aussi un domaine particulier nomm  ARPA. Il r pertorie la transformation des adresses   points en noms de domaines ....

Chaque domaine est ensuite divis  en sous-domaines, eux-m mes pouvant  tre divis s en sous-sous-domaines, et ainsi de suite jusqu'  parvenir effectivement   une machine h te. Ainsi, le premier niveau interm diaire identifie un sous-domaine qui correspond   une entit  conomique ou scientifique, par exemple microsoft, cern, nasa, **prov-liege**, ... ou **ac**. Un exemple de sous-domaine est epl.prov-liege.be et m me inpres.epl.prov-liege.be. Le dernier niveau d signe un sous-domaine qui n'est plus subdivis . Il comporte une machine au minimum, mais peut aussi bien correspondre au point la machine elle-m me, par un nom quelconque : **www**, web, ulyss, ...

Ceci donne comme nom de machine, par exemple, [www.prov-liege.be](http://www.prov-liege.be) ou [www.freesoft.org](http://www.freesoft.org). Chaque composante peut avoir 63 caract res maximum, le nom complet ne pouvant exc der 255 caract res.

On peut remarquer que ce sch ma de nommage est bas  sur les organisations et pas sur les r seaux sous-jacents.

### 7.3 Les serveurs de noms

Un *serveur de noms* [*name server*] est un logiciel qui traduit les noms de machines et de domaines en adresses IP. Un tel programme est encore appelé, sans doute avec un clin d'œil, un BIND (Berkeley Internet Name Domain). Dès qu'un réseau acquiert une certaine importance, son administrateur développe son propre serveur de noms. Comme aucune communication n'est possible tant que l'on ne dispose pas de l'adresse IP pure et dure, on peut dire que les services d'un serveur de noms sont critiques.

Bien sûr, un serveur de noms donné ne connaît pas toutes les adresses Internet, mais seulement celles de sa "juridiction", sa "**zone d'autorité**". En fait, le plus souvent, la zone d'autorité d'un serveur de noms un domaine, avec éventuellement certains sous-domaines (mais pas forcément tous); le domaine de base est appelé le "domaine racine".

En réalité, une zone doit normalement disposer deux serveurs de noms :

- ◆ un serveur de noms principal : il obtient effectivement ses informations à partir de fichiers locaux;
- ◆ un serveur de noms secondaire (voire plusieurs) : il obtient au contraire ses informations à partir d'un autre serveur de noms, qui est serveur principal pour la zone qui contient l'information recherchée; on parle dans ce cas d'un **transfert de zone**.

Les serveurs sont structurés selon une hiérarchie, le principe étant qu'un serveur de niveau donné sait quel serveur de niveau supérieur est susceptible de parvenir à résoudre un nom donné. On comprend dès lors mieux ce que l'on veut dire lorsque l'on affirme que le DNS d'Internet est une base de données distribuées : l'ensemble de l'informations sur la correspondance noms-adresses est effectivement disséminé sur un grand nombre de sites.

Les 13 machines suivantes sont les serveurs primaires de noms de domaines sur Internet :

198.41.0.4	192.5.5.241	198.41.0.10
128.9.0.107	192.112.36.4	193.0.14.129
192.33.4.12	128.63.2.53	198.32.64.12
128.8.10.90	192.36.148.17	202.12.27.33
192.203.230.10		

### 7.4 Le resolveur

Lorsqu'une application a besoin de l'adresse IP correspondant à un nom, elle utilise des primitives comme `gethostbyname()`, lesquelles font appel à des procédures système se trouvant dans une librairie particulière appelée le *solveur* (*resolver*); par abus de langage, le terme "solveur" a fini par désigner l'ensemble du processus de recherche. Le solveur lit alors des fichiers de configuration pour déterminer l'emplacement du ou des serveurs de noms disponibles sur le réseau local. Le plus souvent, sous Unix, c'est le rôle du fichier **resolv.conf** de contenir les adresses IP des serveurs locaux de noms. Ainsi, pour la machine boole, on y trouve :

<b>etc/resolv.conf (machine UNIX boole)</b>	
domain	INPRES.EPL.PROV-LIEGE.BE
search	INPRES.EPL.PROV-LIEGE.BE
nameserver	10.7.0.100

Le solveur envoie alors un paquet UDP au serveur DNS local :

- ◆ ou bien le nom cherché figure dans le fichier de ressources local;

- ♦ ou bien elle ne s'y trouve pas; le solveur s'adresse alors à un DNS de domaine de plus haut niveau (les adresses de ces domaines sont chargées dans la mémoire cache au démarrage du système et n'en sont jamais purgées).

Dans les deux cas, le serveur DNS renvoie l'adresse au solveur, qui la transmet à l'application appelante.

### **Remarques**

- 1) A priori, on a supposé jusqu'ici que les adresses IP des différentes machines sont fixées une fois pour toutes. Cependant, en pratique, il n'est pas toujours possible de donner une adresse fixe à chaque machine d'un grand réseau (pas assez d'adresses disponibles, nombre limité d'adresses utilisée simultanément, etc). Dans ce cas, on installera un serveur DHCP (**Dynamic Host Configuration Protocol**) dont le rôle sera d'attribuer et de communiquer une adresse IP temporaire à la machine client qui le contacte. Dans un tel cas, cette machine ne doit évidemment pas être configurée avec une adresse fixe.
- 2) Un réseau Windows peut mettre en place un serveur WINS (Windows Internet Name Service), dont le rôle est de gérer la correspondance entre les adresses IP et les adresses NetBIOS. Mais ceci n'est jamais qu'un redoublement du rôle du DNS.
- 3) On peut aussi obtenir la correspondance nom-adresse par consultation du fichier etc/hosts (voir chapitre consacré à la configuration TCP/IP).

## **8. Les adresses dans les intranets**

### **8.1 Les machines publiques et privées**

On assiste de plus en plus à l'apparition de réseaux privés fonctionnant selon les principes d'Internet. On parle alors d'"**intranets**" (les anglo-saxons parlent de "**private internets**"). A priori, le gestionnaire d'un tel intranet est libre de distribuer les adresses comme bon lui semble. Il faut cependant remarquer que toutes les machines du réseau ne sont pas équivalentes :

- ♦ certaines ne sont pas du tout connectées au monde extérieur; elles se limitent uniquement à communiquer avec les autres machines de l'intranet; leur adresse IP est unique au sein de l'Intranet, mais est bien sûr ambiguë au niveau d'Internet (plusieurs machines de par le monde ont la même adresse au sein de leur intranet respectif);
- ♦ d'autres machines n'utilisent que certains services bien précis en rapport avec le monde extérieur, comme l'e-mail ou ftp; leur situation est assez similaire à celles des précédentes, puisqu'elles accèdent à ces services par une passerelle de niveau application;
- ♦ enfin, il se trouve dans l'intranet des machines qui doivent être connues sur Internet; elles nécessitent alors une adresse non ambiguë au niveau d'Internet.

Les machines des deux premiers types de sont donc à considérer comme "private" tandis que celles du dernier groupe sont "public"<sup>1</sup>. Seules ces dernières nécessitent une connexion sur Internet qui s'effectue au niveau de la couche réseau (les anglo-saxons utilisent

---

<sup>1</sup> comme dirait un prof de P.O.O. ... ;-)

le terme de "*network layer connectivity*"). Et, donc, seules ces adresses intéressent les routeurs extérieurs à l'intranet. En pratique, le routeur qui gère le trafic vers Internet "fait du NAT" (**Network Address Translation**), c'est-à-dire qu'il traduit les adresses privées et non routables d'un Intranet en une adresse unique et routable. Il existe donc une seule adresse externe publique visible sur [Internet](#) associée à toutes les adresses d'un réseau privé,

Un problème est évident : une entreprise peut donner à ses machines des adresses IP sans se soucier de ce qui se passe dans Internet. Elle donnera alors inévitablement à l'une ou l'autre de ces machines une adresse déjà attribuée à une autre machine dans un autre réseau.

Que se passe-t-il si une certaine machine de l'Intranet passe du statut de machine privée à celui de machine publique ? Il faudra :

- ◆ changer son adresse;
- ◆ modifier son entrée dans le DNS;
- ◆ changer la configuration des machines qui communiquent avec elle en utilisant son adresse IP et pas son nom.

Et si plusieurs Intranets sont fusionnés ? Il est donc apparu qu'il convenait de définir les espaces d'adresses des Intranets,

## **8.2 Les espaces d'adresses des intranets**

Le Request for Comments 1918 a défini une politique plus stricte pour l'attribution des adresses dans les Intranets en préconisant des espaces d'adresses propres aux Intranets. Les adresses IP appartenant à ces espaces ne sont donc jamais attribuées comme adresses publiques.

L'ICANN a réservé les trois blocs d'adresses suivants pour les espaces d'adresses des "internets privés" :

réseau privé	"classe"	plage
10.0.0.0/8	A	10.0.0.0 → 10.255.255.254
172.16.0.0/12	B	172.16.0.0 → 172.31.255.254
192.168.0.0/16	C	192.168.0.0 → 192.168.255.254

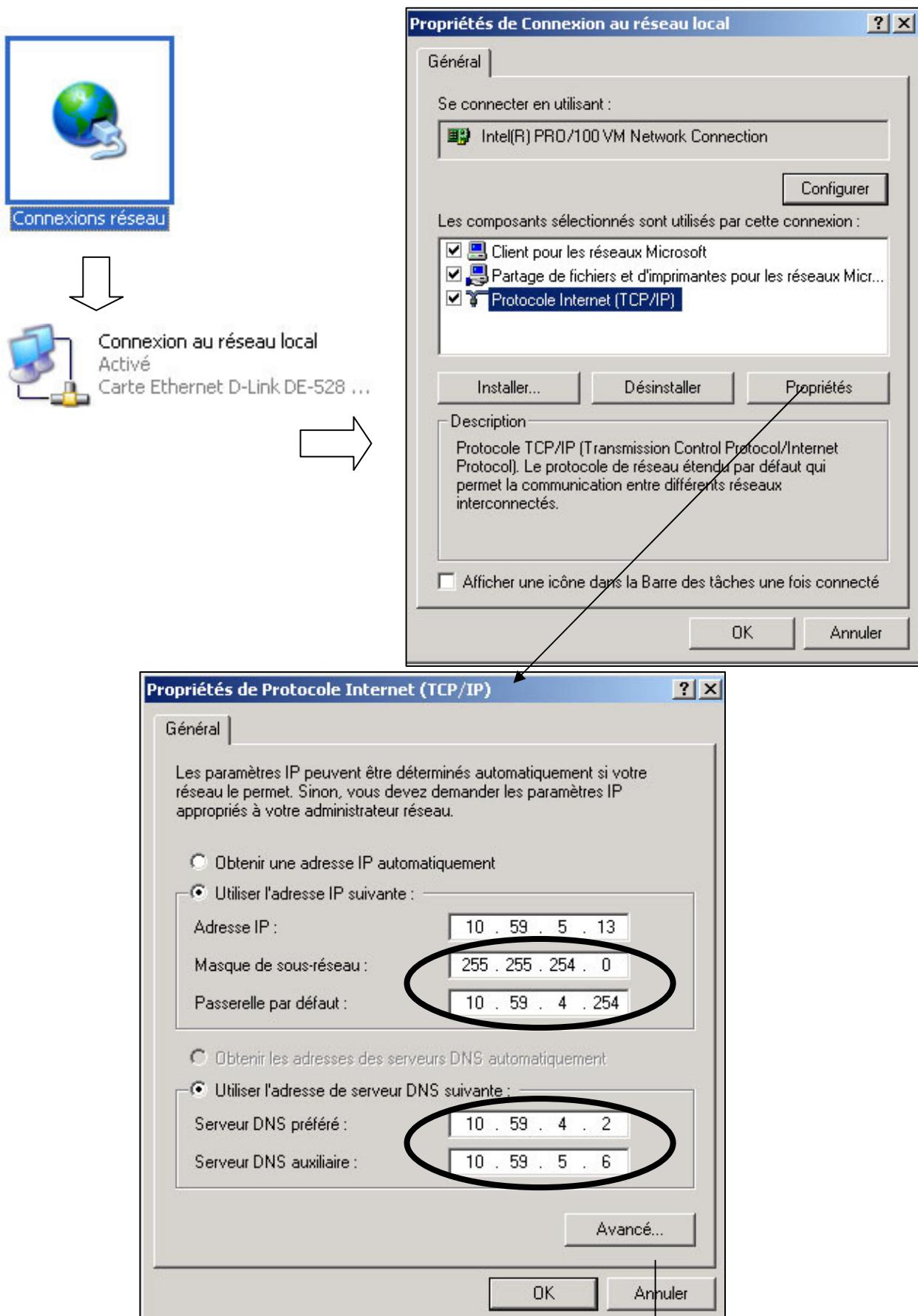
## **8.3 L'intranet de l'In.Pr.E.S.**

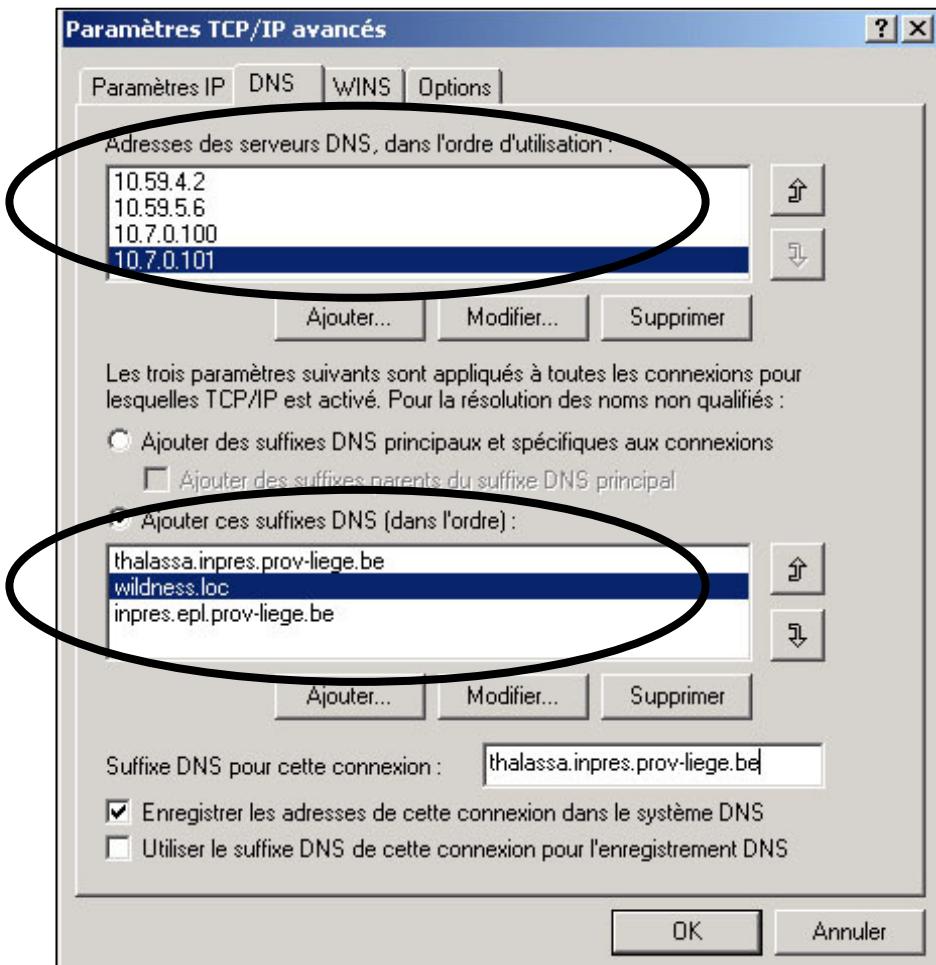
Dans le cas de l'Intranet de la Province de Liège (dont les réseaux de l'In.Pr.E.S. font partie), c'est le groupe d'adresse débutant par 10 qui est utilisé. Plusieurs domaines sont utilisés. Quelques serveurs actuellement en service (en 2012 – mais tout ceci peut être modifié) sont :

machine	adresse	domaine
sunray1	10.59.4.6	inpres.epl.prov-liege.be
u2	10.59.5.219	wildness.loc
indochine	10.59.5.3	inpres.epl.prov-liege.be

Le firewall est donc le seul à être connu du monde extérieur avec une adresse de classe C : 193.190.122.7.

Toute machine de type PC doit avoir une configuration du type suivant :





Nous pouvons raisonnablement considérer que nous possédons à présent de solides bases de programmation TCP. Cependant, tout se passe en mode texte (monde UNIX typique) ou alors, c'est le rouleau compresseur de Java qui se met en route avec ses GUIs et ses listeners. Mais qu'en est-il du monde Windows ?

## VI. La programmation TCP/IP Windows



*Qu'importe que l'on vive plus vite, pourvu que l'on soit plus heureux !*

(P. Mérimée, Lettres à Jenny Dacquin)

### 1. En C : l'interface Winsock

On désigne sous le nom de **Winsock** (Windows Sockets) un ensemble d'APIs, en *langage C*, permettant la programmation des sockets TCP/IP sous Windows. On y retrouve la plupart des APIs classiques de la programmation TCP/IP, mais adaptées aux spécificités du système d'exploitation de Microsoft, c'est-à-dire essentiellement le *mécanisme des messages*. Il faut remarquer que Winsock n'est pas la propriété de Microsoft : il a été développé par des programmeurs issus de différentes sociétés. Ce n'est pas une surprise : Winsock est implémenté sous la forme d'une DLL (oui, oui – elle s'appelle bien Winsock.dll).

En première approche, on peut dire que Winsock comporte :

- ◆ les fonctions de sockets bloquantes : accept(), connect(), recv(), send(), select(), ...
- ◆ les fonctions de sockets non bloquantes : bind(), listen(), socket(), fonctions de conversions de format d'adresses, ...
- ◆ les fonctions d'accès aux informations réseaux : gethostbyname(), gethostbyaddr(), getprotobynumber(), ...
- ◆ les fonctions asynchrones correspondant aux précédentes – elles peuvent donc être utilisées dans le style de programmation classique de Windows, soit une programmation événementielle basée sur les messages : WSAAsyncGetHostByName, WSAAsyncGetHostByAddr, ...
- ◆ les fonctions propres à Windows : WSASStartup (initialisation de la DLL Winsock), WSAGetLastError, WSASyncSelect (select asynchrone), ...

Un seul mega-header est nécessaire à leur utilisation : **socket.h**, dont voici quelques extraits.

```
socket.h (Windows)
#ifndef _WINSOCKAPI_
#define _WINSOCKAPI_

/*
 * Basic system type definitions, taken from the BSD file sys/types.h.
 */

typedef unsigned char u_char;
...
```

```

/*
 * The new type to be used in all
 * instances which refer to sockets.
 */
typedef u_int      SOCKET;

...
struct hostent { ... }
...

/*
 * Protocols
 */
#define IPPROTO_IP      0          /* dummy for IP */
...
#define IPPROTO_TCP     6          /* tcp */
#define IPPROTO_UDP    17          /* user datagram protocol */
...
#define IPPROTO_RAW    255         /* raw IP packet */
#define IPPROTO_MAX    256

/*
 * Port/socket numbers: network standard functions
 */
...
#define IPPORT_FTP      21
#define IPPORT_TELNET   23
#define IPPORT_SSMTP    25
...

/*
 * Internet address (old style... should be updated)
 */
struct in_addr
{
    union
    {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;

#define s_addr S_un.S_addr           /* can be used for most tcp & ip code */
#define s_host S_un.S_un_b.s_b2      /* host on imp */
#define s_net  S_un.S_un_b.s_b1      /* network */
...
};

/*
 * Socket address, internet style.
*/

```

```

struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};

...
/*
 * This is used instead of -1, since the
 * SOCKET type is unsigned.
*/
#define INVALID_SOCKET (SOCKET)(~0)
#define SOCKET_ERROR      (-1)

/*
 * Types
 */
#define SOCK_STREAM     1      /* stream socket */
#define SOCK_DGRAM      2      /* datagram socket */
...

/*
 * TCP options.
*/
#define TCP_NODELAY    0x0001
#define TCP_BSDURGENT  0x7000

/*
 * Address families.
*/
#define AF_UNSPEC      0      /* unspecified */
#define AF_UNIX        1      /* local to host (pipes, portals) */
#define AF_INET        2      /* internetwork: UDP, TCP, etc. */
...

/*
 * Structure used by kernel to store most
 * addresses.
*/
struct sockaddr
{
    u_short sa_family;      /* address family */
    char sa_data[14];       /* up to 14 bytes of direct address */
};

...
/*
 * Maximum queue length specifiable by listen.
*/
#define SOMAXCONN      5

```

```

#define MSG_OOB      0x1      /* process out-of-band data */
#define MSG_PEEK     0x2      /* peek at incoming message */
#define MSG_DONTROUTE 0x4      /* send without using routing tables */
...
/*
 * Windows Sockets definitions of regular Berkeley error constants
 */
#define WSAEWOULDBLOCK      (WSABASEERR+35)
...
#define WSAENOTSOCK         (WSABASEERR+38)
...
#define WSAENOPROTOOPT      (WSABASEERR+42)
#define WSAEPROTONOSUPPORT   (WSABASEERR+43)
...
#define WSAECONNREFUSED      (WSABASEERR+61)
...
/*
 * Extended Windows Sockets error constant definitions
 */
#define WSASYSNOTREADY      (WSABASEERR+91)
#define WSAVERNOTSUPPORTED    (WSABASEERR+92)
#define WSANOTINITIALISED     (WSABASEERR+93)
...
/* Socket function prototypes */

#ifndef __cplusplus
extern "C" {
#endif

SOCKET PASCAL FAR accept (SOCKET s, struct sockaddr FAR *addr, int FAR *addrlen);
int PASCAL FAR bind (SOCKET s, const struct sockaddr FAR *addr, int nameLEN);
int PASCAL FAR connect (SOCKET s, const struct sockaddr FAR *name, int nameLEN);
...
u_long PASCAL FAR htonl (u_long hostlong);
u_short PASCAL FAR htons (u_short hostshort);
...
int PASCAL FAR listen (SOCKET s, int backlog);
...
int PASCAL FAR recv (SOCKET s, char FAR * buf, int len, int flags);
...
int PASCAL FAR select (int nfds, fd_set FAR *readfds, fd_set FAR *writefds,
                      fd_set FAR *exceptfds, const struct timeval FAR *timeout);
int PASCAL FAR send (SOCKET s, const char FAR * buf, int len, int flags);
...
SOCKET PASCAL FAR socket (int af, int type, int protocol);
...
/* Microsoft Windows Extension function prototypes */
int PASCAL FAR WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData);
int PASCAL FAR WSACleanup(void);
...

```

```

int PASCAL FAR WSAGetLastError(void);
...
HANDLE PASCAL FAR WSAAsyncGetHostByName(HWND hWnd, u_int wMsg,
                                         const char FAR * name, char FAR * buf, int buflen);
...
#endif __cplusplus
}
#endif
#endif /* _WINSOCKAPI_ */

```

Un certain nombre de différences méritent d'être épinglees :

- ◆ Les descripteurs de socket de Windows sont du type SOCKET (soit un entier); la valeur -1 (0xFFFF) identifie une socket non valide. Les handles de sockets classiques sont des cousins des handles de fichiers : on peut les utiliser pour des E/S; mais sous Windows, il n'en est plus ainsi : les descripteurs de sockets Windows ne sont pas assimilables à de tels handles.
- ◆ Tout programme désirant utiliser les fonctions Winsock doit au préalable appeler la fonction :

```
int WSASStartup(WORD wVersionRequired, LPWSADATA lpWSAData);
```

Elle permet de spécifier la version de Winsock qui est utilisée et d'ajuster la suite avec la version maximale supportée par la DLL (usuellement, 1.1 ou 2.0); elle fournit aussi des informations sur l'implémentation utilisée (mais le contenu de la structure WSADATA ne nous intéresse pas vraiment ici). La fonction renvoie 0 en cas de succès.

Tout nouvel appel à WSASStartup doit être précédé d'un appel à

```
int WSACleanup(void);
```

Un compteur interne gère le nombre d'appels start et clean. Lorsqu'il s'agit du dernier, les ressources sont libérées.

Un exemple classique de code d'initialisation est :

```

WORD version;
WSADATA infoImpl;
int err;

version = MAKEWORD( 1, 1 );

err = WSASStartup(version, & infoImpl);
if ( err != 0 ) ... ; /* Faire quelque chose d'intelligent pour signaler l'erreur ;-), notamment un
                      WSACleanup */

```

- ♦ En cas d'erreur, les fonctions de Winsock ne positionnent pas une variable du genre errno, mais retournent la valeur :

```
#define SOCKET_ERROR (-1)
```

La nature de l'erreur peut être obtenue par l'appel de la fonction :

```
int WSAGetLastError(void);
```

qui fournit les erreurs classiques dont l'identificateur est préfixé de WSA (par exemple, WSAEAFNOSUPPORT au lieu de EAFNOSUPPORT).

Il est donc possible de développer sous Windows une programmation TCP/IP assez semblable à celle qui est développée sous UNIX (donc synchrone) mais en tenant compte de certains messages de Windows. En pratique, cependant, il est de plus en plus fréquent d'utiliser intégralement les mécanismes de la programmation événementielle (donc asynchrone) et surtout la Programmation Orientée Objets (quel que soit le langage) avec des classes réseaux appropriées.

## 2. En C++ : les classes sockets de MFC

La bibliothèque MFC (Microsoft Foundation Classes), bien connue des programmeurs Windows, comporte des classes dévolues à l'encapsulation des mécanismes de la programmation TCP/IP. Même si leur emploi est tombé en désuétude pour les nouvelles applications .NET, il est utile de les évoquer eu égard aux nombreuses applications existantes développées avec ces classes.

### 2.1 La classe CAAsyncSocket

La classe de base est **CAAsyncSocket** : avec un nom pareil, son rôle semble clair mais, en fait, elle apporte tout ce qui est nécessaire pour travailler avec les sockets

- ♦ selon le paradigme classique de Windows, soit la gestion d'événements, donc asynchrone;
- ♦ selon le paradigme classique des sockets, soit de manière synchrone.

On peut donc parler d'asynchronisme, mais pas comme une obligation, plutôt comme d'une possibilité. Tous les problèmes classiques de gestion des sockets (fonctions bloquantes, network byte order, etc) sont ici laissés aux bons soins des programmeurs. La classe CAAsyncSocket est déclarée dans Afxsock.h :

<b>classe CAAsyncSocket</b>
class CAAsyncSocket : public CObject
{
DECLARE_DYNAMIC(CAAsyncSocket);
...
public:
<b>CAAsyncSocket()</b> ;
BOOL <b>Create</b> (UINT nSocketPort = 0, int nSocketType=SOCK_STREAM,
long lEvent = FD_READ   FD_WRITE   FD_OOB   FD_ACCEPT   FD_CONNECT
FD_CLOSE, LPCTSTR lpszSocketAddress = NULL);

```

// Attributes
public:
    SOCKET m_hSocket;
```

operator SOCKET() const;

...

BOOL SetSockOpt(int nOptionName, const void\* lpOptionValue,  
 int nOptionLen, int nLevel = SOL\_SOCKET);

BOOL GetSockOpt(int nOptionName, void\* lpOptionValue,  
 int\* lpOptionLen, int nLevel = SOL\_SOCKET);

...

static int PASCAL GetLastError();

```

// Operations
public:
    virtual BOOL Accept(CAsyncSocket& rConnectedSocket,
                         SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL);
```

BOOL Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress = NULL);  
 BOOL Bind (const SOCKADDR\* lpSockAddr, int nSockAddrLen);

virtual void **Close**();

BOOL **Connect**(LPCTSTR lpszHostAddress, UINT nHostPort);  
 BOOL Connect(const SOCKADDR\* lpSockAddr, int nSockAddrLen);

...

BOOL **Listen**(int nConnectionBacklog=5);

virtual int **Receive**(void\* lpBuf, int nBufLen, int nFlags = 0);

...

enum { receives = 0, sends = 1, both = 2 };

BOOL ShutDown(int nHow = sends);

virtual int **Send**(const void\* lpBuf, int nBufLen, int nFlags = 0);

...

BOOL AsyncSelect(long lEvent =  
 FD\_READ | FD\_WRITE | FD\_OOB | FD\_ACCEPT | FD\_CONNECT | FD\_CLOSE);

```

// Overridable callbacks
protected:
    virtual void OnReceive(int nErrorCode);
    virtual void OnSend(int nErrorCode);
    virtual void OnOutOfBandData(int nErrorCode);
    virtual void OnAccept(int nErrorCode);
    virtual void OnConnect(int nErrorCode);
    virtual void OnClose(int nErrorCode);
```

```

// Implementation
public:
    virtual ~CAsyncSocket();
```

...

```

BOOL Socket(int nSocketType=SOCK_STREAM, long lEvent =
            FD_READ | FD_WRITE | FD_OOB | FD_ACCEPT | FD_CONNECT | FD_CLOSE,
            int nProtocolType = 0, int nAddressFormat = PF_INET);
...
protected:
    friend class CSocketWnd;
    ...
};
```

On remarquera :

- ◆ la variable membre ***m\_hSocket*** qui est le handle de la socket;
- ◆ le constructeur qui est un constructeur par défaut : il se contente de créer un objet socket vide. En fait, la socket sera effectivement créée au niveau du système par l'appel de la méthode:

<b>BOOL Create(</b>	UINT nSocketPort = 0, int nSocketType = SOCK_STREAM, long lEvent = FD_READ   FD_WRITE   FD_OOB   FD_ACCEPT   FD_CONNECT   FD_CLOSE, LPCTSTR lpszSocketAddress = NULL );
---------------------	---

La première valeur par défaut concerne le port utilisé : elle laisse le choix du port utilisé à la discréption du système. Ceci convient bien pour un client qui va se connecter à un serveur (par la méthode Connect), mais pas pour un serveur qui devra impérativement spécifier ici le port sur lequel il a l'intention d'écouter. On remarquera aussi les flags permettant de préciser quels événements réseaux seront pris en compte.

- ◆ les méthodes du style de la programmation TCP/IP classique :

<b>BOOL Listen(</b>	int nConnectionBacklog = 5 );
---------------------	-------------------------------

<b>virtual BOOL Accept(</b>	CAsyncSocket& rConnectedSocket, SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL );
-----------------------------	---

<b>BOOL Connect(</b>	LPCTSTR lpszHostAddress, UINT nHostPort );
----------------------	---

<b>virtual int Send(</b>	const void* lpBuf, int nBufLen, int nFlags = 0 );
--------------------------	---

<b>virtual int Receive(</b>	void* lpBuf, int nBufLen, int nFlags = 0 );
-----------------------------	---

<b>virtual void Close( );</b>
-------------------------------

- ♦ les méthodes de réaction aux **événements réseaux** - chacun de ces événements provoque une notification à l'objet socket concerné :

```
virtual void OnAccept( int nErrorCode );
virtual void OnConnect( int nErrorCode );
virtual void OnReceive( int nErrorCode );
...
```

Inutile de préciser que ces méthodes virtuelles demandent à être redéfinies pour chaque application particulière. A remarquer aussi c'est le paramètre qui renseigne sur la nature d'une erreur réseau éventuelle, et pas le code de retour de la méthode associée (par exemple, Connect et OnConnect) !

- ♦ la méthode de détection d'erreur, qui ressemble comme une sœur à l'API correspondante :

```
static int GetLastError( );
```

## 2.2 La classe CSocket

MFC propose une classe dérivée **CSocket** qui encapsule un certain nombre de questions laissées au bon vouloir des programmeurs dans sa classe mère : on peut donc parler, comme le fait l'aide de Microsoft, de "niveau d'abstraction plus élevé" des sockets ... La classe CSocket fournit

- ♦ un mécanisme de blocage, très utile en programmation synchrone, pour les méthodes Receive, Send et Accept - évidemment, pour qui connaît le dessous des cartes, la paramétrisation des sockets n'est pas étrangère à tout cela.
- ♦ une communication réseau basée sur la sérialisation des données (dans un sens ou dans l'autre) en associant à la socket des objets CSocketFile et CArchive – bien sûr, pour qui connaît la programmation Java, cela n'est pas neuf ;-)

La classe est déclarée dans afxsock.h :

<b>classe CSocket</b>
<pre>class <b>CSocket</b> : public <b>CAsyncSocket</b> {     DECLARE_DYNAMIC(CSocket); private:     CSocket(const CSocket&amp; rSrc);      // no implementation     void operator=(const CSocket&amp; rSrc); // no implementation  // Construction public:     <b>CSocket</b>();     BOOL <b>Create</b>(UINT nSocketPort = 0, int nSocketType=SOCK_STREAM,                  LPCTSTR lpszSocketAddress = NULL);  // Attributes public:     BOOL <b>IsBlocking</b>();</pre>

```
static CSocket* PASCAL FromHandle(SOCKET hSocket);
BOOL Attach(SOCKET hSocket);

// Operations
public:
    void CancelBlockingCall();

// Overridable callbacks
protected:
    virtual BOOL OnMessagePending();

// Implementation
public:
    int m_nTimeOut;
    virtual ~CSocket();
    static int PASCAL ProcessAuxQueue();

    virtual BOOL Accept(CAsyncSocket& rConnectedSocket,
                      SOCKADDR* lpSockAddr = NULL, int* lpSockAddrLen = NULL);
    virtual void Close();

    virtual int Receive(void* lpBuf, int nBufLen, int nFlags = 0);
    virtual int Send(const void* lpBuf, int nBufLen, int nFlags = 0);

    int SendChunk(const void* lpBuf, int nBufLen, int nFlags);

protected:
    friend class CSocketWnd;
    ...
};
```

On peut constater que les seuls changements notables sont les redéfinitions des méthodes typiques de communication, ceci afin de tenir compte que du schéma classique de communication synchrone TCP/IP avec blocage vu sous Unix.

Comme Microsoft recommande chaudement de développer les nouvelles applications en se basant sur l'architecture .NET, nous n'investirons pas plus dans l'étude de ces classes ... en ayant une pensée émue pour les nombreuses entreprises qui ont investi dans des applications basées sur ces classes ☺. Ces applications sont néanmoins toujours utilisables en mode non managé ☺ !

### 3. L'architecture .NET et TCP/IP

C'est peu dire que l'avènement de l'architecture .NET a considérablement modifié le travail des programmeurs du monde Windows. Maints ouvrages se sont fait un devoir de le montrer. En particulier, C#, clône microsoftien de Java ;-), se devait de proposer au développeur des moyens efficaces de mettre au point des communications réseaux fiables.



Cependant, contrairement à ce qui se passe en Java avec la machine virtuelle et quoi qu'on en dise, l'architecture .NET reste fondamentalement Windows. Les outils de programmation réseau correspondants se devaient donc de combiner deux éléments :

- ◆ programmation **synchrone** (à la Unix) ou **asynchrone** (à la Windows);
- ◆ programmation au **niveau des sockets** (à la Unix) ou programmation de plus **haut niveau** (à la Java).

Nous allons voir ici que les différentes possibilités sont offertes avec plus ou moins de bonheur. Le langage support sera C#.

### 4. En C# : la classe Socket

La classe **Socket** (qui n'est pas liée à C# mais à .NET), relevant de l'espace de nom **System.Net.Sockets**, est présentée comme l'implémentation de l'interface des sockets Berkeley que nous connaissons à présent très bien. Cet espace de noms System.Net.Sockets fournit en fait une implémentation managée de l'interface Windows Sockets (Winsock) pour les développeurs qui doivent travailler à un niveau relativement bas.

A priori, elle n'est pas dédiée obligatoirement à un protocole de transport précis. Le constructeur de cette classe a pour prototype en C# :

```
public Socket  
(  
    AddressFamily addressFamily,  
    SocketType socketType,  
    ProtocolType protocolType  
)
```

On voit ainsi apparaître de nouvelles classes dont le rôle semble assez clair pour qui connaît la programmation conventionnelle (c'est-à-dire UNIX) des sockets. Ainsi :

#### 1) public enum AddressFamily

Un membre de AddressFamily spécifie le domaine et aussi le modèle d'adresse utilisé par la socket pour résoudre une adresse. Les valeurs les plus courantes sont :

<b>InterNetwork</b>	Adresse IP version 4.
<b>InterNetworkV6</b>	Adresse IP version 6.
<b>Unix</b>	Adresse Unix locale vers hôte

#### 2) public enum SocketType

Les membres de cette énumération caractérisent le type de socket. Bien normalement, les deux valeurs utilisées couramment sont :

<b>Stream</b>	Il s'agit d'une communication en <i>mode connecté</i> qui, concrètement, utilise le protocole TCP et le domaine InterNetwork.
<b>Dgram</b>	Il s'agit d'une communication en <i>mode non connecté</i> qui, concrètement, utilise le protocole UDP et le domaine InterNetwork.

### 3) public enum ProtocolType

Les membres de cette énumération désignent évidemment le protocole utilisé. Parmi les valeurs possibles, relevons :

Icmp	Internet Control Message Protocol.
IP	Internet Protocol.
Raw	Protocole de paquets Raw UP.
Tcp	Transmission Control Protocol.
Udp	User Datagram Protocol.

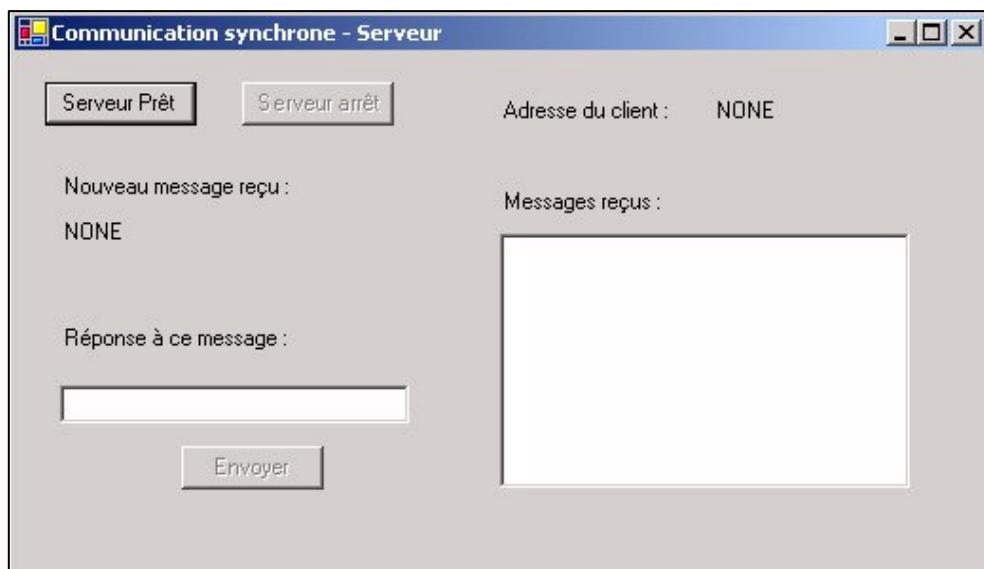
Donc, en pratique et par exemple, nous créerons une socket pour les communications TCP/IP au moyen de :

```
Socket s = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp)
```

A priori, les objets instances de cette classe socket peuvent aussi bien être des sockets d'écoute que de service, synchrone ou asynchrone de surcroît. Explorons donc les diverses possibilités avec méthode ;-).

## 5. Un serveur TCP synchrone et les opérations de base

Nous retrouvons ici les étapes habituelles de la programmation d'un serveur : bind, listen et accept avant le balai des receive et send. Pour illustrer nos propos, nous allons imaginer un serveur qui présente un interface graphique indiquant les opérations effectuées. En pratique, un tel serveur synchrone avec un GUI est cependant peu plaisant à utiliser, à cause du caractère bloquant de plusieurs opérations réseau – l'utilisation de threads est donc une voie à envisager, à moins que l'on se contente d'un serveur sans GUI tournant en mode texte. On envisage donc ici pour lui l'aspect suivant :



## **5.1 L'attachement d'une socket à une adresse (bind)**

La classe Socket possède la méthode attendue :

```
public void Bind(EndPoint localEP);
```

où la classe **EndPoint** identifie une adresse réseau avec son port. En fait, il s'agit d'une classe abstraite, dont la classe dérivée

public class **IPEndPoint** : EndPoint

fournit une réelle implémentation et deux constructeurs effectivement utilisables :

```
public IPEndPoint(long address, int port);
public IPEndPoint(IPAddress address, int port);
```

On peut donc spécifier l'adresse à la dure, sous la forme de l'entier associé. En pratique, bien entendu, on préférera passer l'adresse en format pointé ou, mieux encore, utiliser l'adresse de la machine hôte fournie par le DNS (ce qui assure une portabilité de bon aloï).

**1)** Dans le premier cas, on utilisera la deuxième forme du constructeur utilisant la classe **IPAddress** qui matérialise la notion d'adresse IP. Ce n'est pas tant le constructeur de cette classe qui nous intéresse que la méthode

```
public static IPAddress Parse(string ipString);
```

On peut en effet passer l'adresse pointée sous forme d'une chaîne de caractères à cette méthode de classe pour obtenir l'objet IPAddress correspondant.

**2)** Dans le second cas, on utilise les services d'une classe

```
public sealed class Dns
```

Cette classe permet d'obtenir les informations réseaux sur une machine donnée en utilisant son nom DNS. Ainsi, la méthode de classe :

```
public static string GetHostName();
```

permet bien entendu de récupérer le nom de la machine hôte. De même, la méthode de classe :

```
public static IPHostEntry GetHostByName (string hostName);
```

permet de rechercher une machine par son nom DNS. Les informations ainsi obtenues au cours de la requête DNS sont renvoyées dans une instance de la classe

```
public class IPHostEntry
```

Cette classe contient notamment une propriété :

```
public IPAddress[] AddressList {get; set;}
```

---

destinée évidemment à contenir la(les) adresse(s) IP correspondant à la machine visée. S'il existe donc plusieurs entrées dans la base de données DNS pour l'hôte spécifié, **IPHostEntry** contient plusieurs adresses IP et alias.

Concrètement, notre serveur devra donc posséder une première variable membre :

```
private Socket socketEc;
```

qui lui servira de socket d'écoute :

#### **serveur synchrone : bind()**

```
String nomServeur = Dns.GetHostName();
IPHostEntry he = Dns.GetHostByName(nomServeur);
IPAddress[] listeAdresses = he.AddressList;
socketEc.Bind (new IPEndPoint(listeAdresses[0], 5000));
```

Au niveau du traitement des erreurs, il faut savoir que cette méthode Bind () est susceptible de lancer une exception instance de la classe **SocketException**, héritée de Win32Exception et qui comporte donc la propriété :

```
public override int ErrorCode {get;}
```

Ceci permet d'obtenir le code de la dernière erreur de réseau survenue.

Type d'exception	Interprétation
ArgumentNullException	Le paramètre <i>buffer</i> est une référence null (Nothing dans Visual Basic).
ArgumentOutOfRangeException	Le paramètre <i>offset</i> ou <i>size</i> est supérieur à la taille de <i>buffer</i> .
SocketException	Une erreur du système d'exploitation s'est produite lors de l'accès au socket.
ObjectDisposedException	Socket a été fermé.

#### **5.2 La mise à l'écoute sur la socket (listen)**

L'ouverture passive d'une connexion TCP s'effectue sans surprise au moyen de l'instruction :

```
public void Listen(int backlog);
```

où le paramètre représente la longueur maximale de la file d'attente des connexions en attente. Dans notre cas :

#### **serveur synchrone : listen()**

```
socketEc.Listen (100);
```

### **5.3 La prise en compte d'une connexion pendante (accept)**

Bien en accord avec nos habitudes, la méthode

```
public Socket Accept();
```

retourne bien une socket de service. Notre serveur aura donc une deuxième variable membre socket

```
private Socket socketServ;
```

qui sera une socket de service :

<b>serveur synchrone : accept()</b>
<i>socketServ = socketEc.Accept();</i>

### **5.4 La fermeture d'une socket (close et shutdown)**

On dispose bien sûr de :

```
public void Close();
```

L'application doit cependant, de préférence, appeler auparavant la méthode Shutdown pour s'assurer que toutes les données en attente seront envoyées ou reçues avant la fermeture de Socket. Cette méthode a pour syntaxe :

```
public void Shutdown(SocketShutdown how);
```

et son argument est un élément de l'énumération :

```
public enum SocketShutdown
```

dont les valeurs possibles sont :

<b>Both</b>	Arrête une socket pour l'envoi et la réception.
<b>Receive</b>	Arrête une socket pour la réception.
<b>Send</b>	Arrête une socket pour l'envoi.

### **5.5 L'envoi et la réception de données**

La socket de service va bien entendu permettre la communication, selon un protocole applicatif, avec un client. Les méthodes nécessaires sont :

1) la méthode de réception :

```
public int Receive(byte[] buffer);
```

la valeur de retour représentant le nombre d'octets reçus. Comme ce sont des bytes qui sont reçus, il faudra éventuellement les transformer en chaînes de caractères si le message reçu est

textuel. Or, il existe une classe **ASCIIEncoding** de l'espace de noms System.Text, dérivée de la classe **Encoding** qui représente un codage de caractères, qui code les caractères Unicode en caractères ASCII simples de 7 bits. Une propriété membre de classe intéressante est un objet **Encoding** :

```
public static Encoding Default {get;}
```

qui fournit un codage pour la page de codes ANSI du système hôte. Un tel objet possède la méthode dont nous avons besoin :

```
public override string GetString(byte[] bytes);
```

ou

```
public override string GetString(byte[] bytes, int byteIndex, int byteCount);
```

où le deuxième paramètre *byteIndex* désigne la position du premier byte à convertir tandis que le troisième représente le nombre de bytes à convertir. Si notre serveur laisse au client l'initiative de la conversation, nous aurons donc :

#### **serveur synchrone : receive()**

```
byte[] buf = new byte[200];
int n = socketServ.Receive(buf);
String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
Console.WriteLine("Message reçu = " + msgClient);
ZTMessageRecu.Text = msgClient;
```

si ZTMessageRecu est une zone de texte quelconque.

**2) la méthode d'envoi :**

#### **public int Send(byte[] buffer, int offset, int size, SocketFlags socketFlags);**

la valeur de retour représentant le nombre d'octets envoyés – une version polymorphe se contente du premier paramètre. Le serveur pourra donc répondre :

#### **serveur synchrone : send()**

```
String msgServeur = "ACK : " + msgClient;
byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
int n = socketServ.Send(tb);
```

## **5.6 La socket peer**

Qui est à l'autre bout ? Le serveur peut répondre à la question au moyen de la propriété :

```
public EndPoint RemoteEndPoint {get;}
```

qui représente le point de terminaison distant que la socket utilise pour les communications. Il nous faudra caster le résultat en **IPEndPoint**. Une propriété de ce dernier nous intéresse :

```
public IPAddress Address {get; set;}
```

avec le fait que la classe IPAddress comporte la méthode :

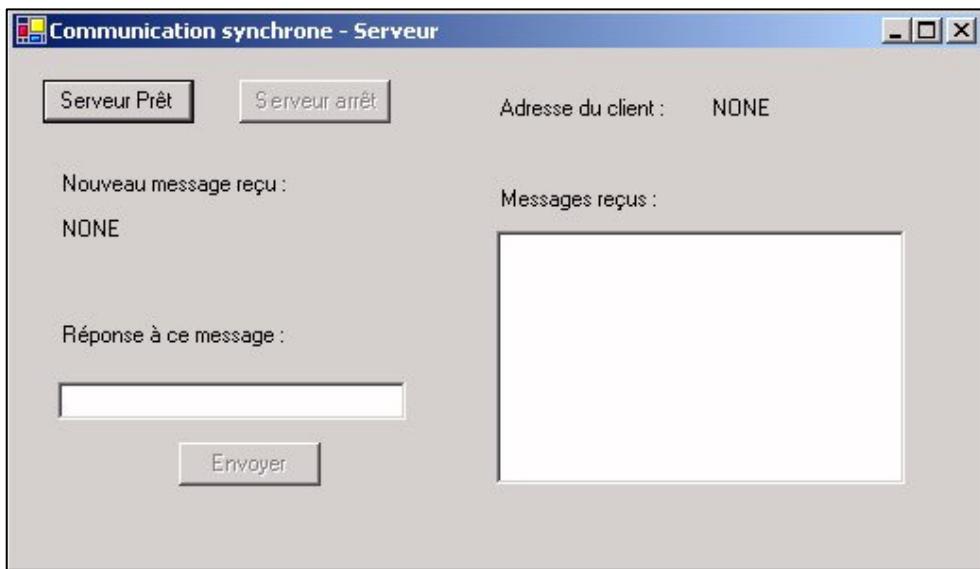
```
public override string ToString();
```

Donc, si ZTClient est unz zone de texte :

```
serveur synchrone : socket peer  
IPEndPoint ep = (IPPEndPoint)socketServ.RemoteEndPoint;  
ZTClient.Text = ep.ToString();
```

## 5.7 Le code du serveur synchrone

Pour nous résumer, voici donc un serveur se présentant avec le GUI du type annoncé :



Voici le code de ce serveur TCP synchrone, allégé du code concernant le GUI qui ne nous intéresse guère ici :

### ServeurReseauSync.cs

```
using System;  
...  
using System.Net;  
using System.Net.Sockets;  
using System.Text;  
using System.Threading;  
  
namespace ServeurRéseau  
{  
    public class ServeurReseauSync : System.Windows.Forms.Form  
    {  
        private Socket socketEc;  
        private Socket socketServ;
```

```

private System.Windows.Forms.TextBox ZEMsgReponse;
...

public ServeurReseauSync ()
{
    InitializeComponent();
    socketEc = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
    BDemarrer.Enabled=true; BArreter.Enabled=false;
    BEnvoyer.Enabled=false;
}

protected override void Dispose( bool disposing ) { ... }

private void InitializeComponent()
{
    this.BDemarrer = new System.Windows.Forms.Button();
    this.BArreter = new System.Windows.Forms.Button();
    this.BEnvoyer = new System.Windows.Forms.Button();
    ...
    //
    // BDemarrer
    //
    this.BDemarrer.Location = new System.Drawing.Point(16, 16);
    this.BDemarrer.Name = "BDemarrer";
    this.BDemarrer.Text = "Serveur Prêt";
    this.BDemarrer.Click += new System.EventHandler
        (this.BDemarrer_Click);
    //
    // BArreter
    //
    this.BArreter.Location = new System.Drawing.Point(120, 16);
    this.BArreter.Name = "BArreter";
    ...
}

[STAThread]
static void Main()
{
    Application.Run(new ServeurReseauSync());
}

private void BDemarrer_Click(object sender, System.EventArgs e)
{
    BDemarrer.Enabled=false; BArreter.Enabled=true;
    BEnvoyer.Enabled=false;

    String nomServeur = Dns.GetHostName();
    IPHostEntry he = Dns.GetHostByName(nomServeur);
    IPAddress[] listeAdresses = he.AddressList;
    //if (listeAdresses.Length > 0)ZENomServeur.Text = "ERREUR !!!";
}

```

```
socketEc.Bind (new IPEndPoint(listeAdresses[0], 5000));
Console.WriteLine("Bind effectué sur le port 5000");
socketEc.Listen (100);
Console.WriteLine("Listen effectué");
socketServ = socketEc.Accept();

IPEndPoint ep = (IPEndPoint)socketServ.RemoteEndPoint;
ZTClient.Text = ep.ToString();

byte[] buf = new byte[200];
int n = socketServ.Receive(buf);
String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
Console.WriteLine("Message reçu = " + msgClient);
ZTMessageRecu.Text = msgClient;
LMsgRecus.Items.Add(msgClient);
BEnvoyer.Enabled=true;
}

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgServeur = ZEMsgReponse.Text;
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    int n = socketServ.Send(tb);

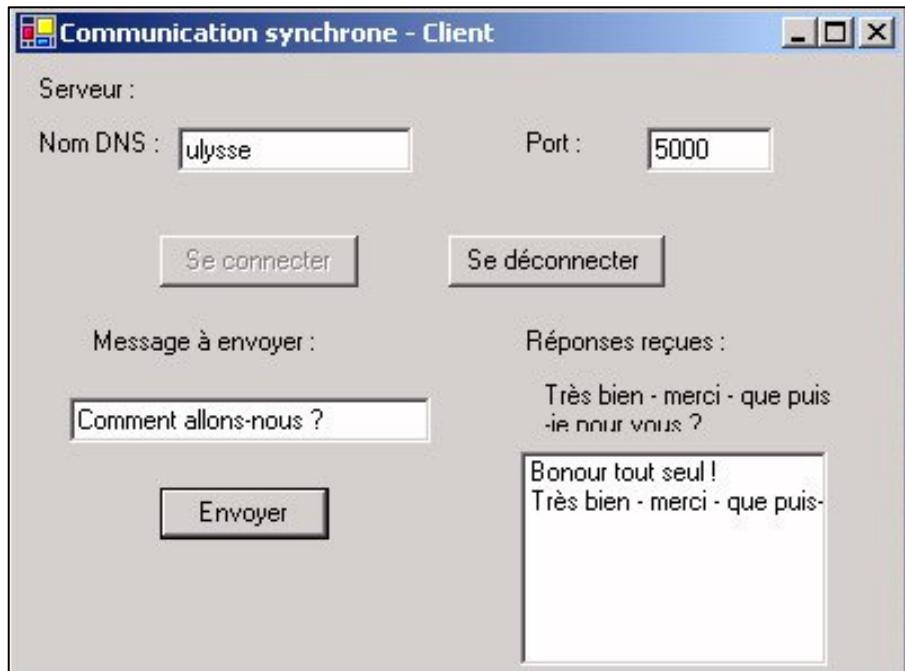
    byte[] buf = new byte[100];
    n = socketServ.Receive(buf);
    String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
    Console.WriteLine("Message reçu = " + msgClient);
    ZTMessageRecu.Text = msgClient;
    LMMsgRecus.Items.Add(msgClient);

    if (msgClient.Equals("EOC"))
    {
        BEnvoyer.Enabled=false;
        socketServ.Shutdown(SocketShutdown.Both);
        socketServ.Close();

        socketServ = socketEc.Accept();
        IPEndPoint ep = (IPEndPoint)socketServ.RemoteEndPoint;
        ZTClient.Text = ep.ToString();
        buf = new byte[200];
        n = socketServ.Receive(buf);
        msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
        Console.WriteLine("Message reçu = " + msgClient);
        ZTMessageRecu.Text = msgClient;
        LMMsgRecus.Items.Add(msgClient);
        BEnvoyer.Enabled=true;
    }
}
```

```
private void BArreter_Click(object sender, System.EventArgs e)
{
    BDemarrer.Enabled=true; BArreter.Enabled=false;
    BEnvoyer.Enabled=false;
    socketEc.Close();
}
}
```

Une interaction avec un client (que nous allons écrire) donne quelque chose de ce genre :



Le client envoie la chaîne de caractères EOC pour se déconnecter; le serveur est alors prêt à prendre en compte une nouvelle connexion. Résultat sur la console :

```
Bind effectué sur le port 5000
Listen effectué
Message reçu = Bonjour !
Message reçu = Comment, allons-nous ?
Message reçu = Avez vous la marchandise ?
Message reçu = EOC
```

L'écriture du client correspondant est assez similaire ...

## 6. Un client TCP synchrone

Un client du serveur élémentaire décrit ci-dessus possèdera une socket variable membre :

```
private Socket socketC;
```

La méthode de connexion au serveur est simplement :

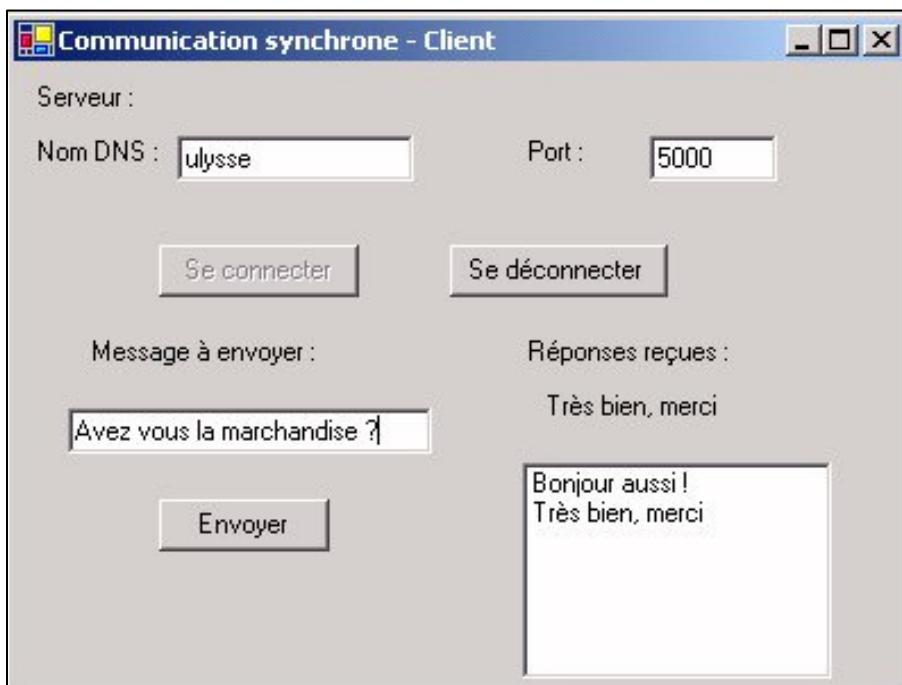
```
public void Connect(EndPoint remoteEP);
```

Cela donnera ici, si ZENomServeur est une zone de texte :

### client synchrone : connect()

```
String nomServeur = ZENomServeur.Text;  
  
IPHostEntry he = Dns.GetHostByName(nomServeur);  
  
IPAddress[] listeAdresses = he.AddressList;  
try  
{  
    socketC.Connect(new IPEndPoint(listeAdresses[0], 5000));  
}  
catch (Exception ex)  
{  
    Console.WriteLine("Erreur : " + ex.ToString());  
}
```

Si donc notre client possède un GUI analogue à celui du serveur, la conversation évoquée ci-dessus a l'aspect suivant de son point de vue :



Le code du client, expurgé de ses instructions GUIs, est :

---

### ClientReseauSync.cs

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace ClientReseau
{
    public class ClientReseauSync : System.Windows.Forms.Form
    {
        Socket socketC;

        private System.Windows.Forms.Button BConnecter;
        private System.Windows.Forms.TextBox ZENomServeur;
        private System.Windows.Forms.Button BDeconnecter;
        private System.Windows.Forms.Button BEnvoyer;
        ...

        public ClientReseauSync()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing ) { ... }

        private void InitializeComponent()
        {
            this.BConnecter = new System.Windows.Forms.Button();
            this.BDeconnecter = new System.Windows.Forms.Button();
            this.BEnvoyer = new System.Windows.Forms.Button();
            ...
            //
            // BConnecter
            //
            this.BConnecter.Location = new System.Drawing.Point(64, 80);
            this.BConnecter.Name = "BConnecter";
            this.BConnecter.Text = "Se connecter";
            this.BConnecter.Click += new System.EventHandler(
                this.BConnecter_Click);
            ...
        }

        [STAThread]
        static void Main()
        {
            Application.Run(new ClientReseauSync());
        }
    }
}
```

```

private void BConnecter_Click(object sender, System.EventArgs e)
{
    socketC = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
    String nomServeur = ZENomServeur.Text;
    IPHostEntry he = Dns.GetHostByName(nomServeur);
    IPAddress[] listeAdresses = he.AddressList;
    if (listeAdresses.Length == 0) ZENomServeur.Text = "ERREUR !!!";
    else
    {
        try
        {
            socketC.Connect(new IPEndPoint(listeAdresses[0], 5000));
            BConnecter.Enabled=false;
            BDeconnecter.Enabled=true;
            BEnvoyer.Enabled=true;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Erreur : " + ex.ToString());
        }
    }
}

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgClient = ZEMsg.Text;
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgClient);
    int n = socketC.Send(tb);

    if (msgClient.Equals("EOC"))
    {
        BConnecter.Enabled=true; BDeconnecter.Enabled=false;
        BEnvoyer.Enabled=false;
        socketC.Shutdown(SocketShutdown.Both);
        socketC.Close();
        return;
    }

    byte[] buf = new byte[100];
    n = socketC.Receive(buf);
    String msgServeur = ASCIIEncoding.Default.GetString(buf, 0, n);
    LMsgRecus.Items.Add(msgServeur);
    ZTMsg.Text = msgServeur;
}

private void BDeconnecter_Click(object sender, System.EventArgs e)
{
    String msgClient = "EOC";
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgClient);
}

```

```

        int n = socketC.Send(tb);

        BConnecter.Enabled=true; BDisconnecter.Enabled=false;
        BEnvoyer.Enabled=false;
        socketC.Shutdown(SocketShutdown.Both);
        socketC.Close();
    }
}
}

```

## 7. Un client TCP asynchrone

Le problème dans les programmes précédents est bien entendu le fait que certaines méthodes réseaux sont bloquantes. Il en résulte que, par exemple, les GUIs éventuels restent bloqués ou ont un comportement indésirable durant ces opérations, ou encore tout simplement que les clients et les serveurs sont totalement liés au réseau. En utilisant le paradigme de programmation asynchrone de .NET, on peut résoudre ce problème.

### 7.1 Les méthodes asynchrones de .NET

On peut en effet remplacer une opération synchrone à priori monolithique et bloquante par une opération asynchrone basée sur la logique événementielle et divisée en deux composantes :

- ◆ la méthode qui constitue la première partie (typiquement appelée **BeginXXX()**) obtient les données à traiter et commence l'opération asynchrone; éventuellement, elle peut aussi recevoir
  - une fonction callback (plus précisément un délégué [*delegate*] au sens .NET, soit une espèce de pointeur de fonction) à appeler quand l'opération asynchrone est achevée;
  - un objet de type quelconque (donc un *object*) qui sera le paramètre de la fonction callback;
 cette méthode **BeginXXX()** renvoie un objet implémentant l'interface **IAsyncResult** qui fournit des informations sur la fonction en cours d'exécution (nous allons y revenir pour parler de la synchronisation);
- ◆ la seconde méthode (typiquement appelée **EndXXX()**) reçoit comme paramètre la valeur renvoyée par le **BeginXXX()** associé (c'est par ce biais que les couples Begin-End se reconnaissent) et retourne les résultats de l'opération asynchrone lorsque celle-ci est terminée, autrement dit la valeur renvoyée par la fonction callback; elle est typiquement bloquante puisqu'elle attend la fin de l'opération asynchrone; .

Le résultat concret de cette manière de programmer est que le programme principal poursuit son exécution une fois l'opération lancée par **BeginXXX()**; en fait,

.NET démarre un thread, choisi parmi un pool de threads en attente, qui se charge de l'exécution de l'opération commandée par **BeginXXX()**

Le programme principal peut toujours se mettre en attente de la fin de l'exécution du thread.

## 7.2 La connexion asynchrone

Pour en revenir à nos communications réseau, un client asynchrone (pour commencer par le plus simple) utilisera la méthode de la classe Socket :

```
public IAsyncResult BeginConnect (EndPoint remoteEP, AsyncCallback cback, object state);
```

- ◆ le 1<sup>er</sup> paramètre permettra évidemment de désigner le serveur (en pratique, il s'agira donc d'un IPPEndPoint);
- ◆ le 3<sup>ème</sup> paramètre représente une information quelconque passée à la fonction de terminaison désignée par le 2<sup>ème</sup> paramètre (dans notre cas, ce sera tout simplement la socket utilisée);
- ◆ le 2<sup>ème</sup> paramètre est un délégué :

```
public delegate void AsyncCallback( IAsyncResult ar );
```

qui désignera cette fonction de terminaison; celle-ci est donc une méthode de classe recevant comme paramètre un objet qui implémente l'interface **IAsyncResult**; cet objet a pour rôle de conserver des informations sur l'état d'une opération asynchrone et fournit un objet de synchronisation permettant aux threads d'être prévenus une fois l'opération terminée : c'est la raison d'être des propriétés spécifiées dans cet interface :

- ◆ object **AsyncState** {get;}  
qui sera un objet associé par l'utilisateur à l'opération asynchrone; en fait, c'est le paramètre passé à la fonction callback;
- ◆ **WaitHandle AsyncWaitHandle** {get;}  
qui est une instance de la classe WaitHandle utilisé pour attendre qu'une opération asynchrone se termine; on trouve dans cette classe des méthodes polymorphes comme :

```
* public virtual bool WaitOne();
```

bloque le thread en cours jusqu'à ce que le WaitHandle en cours reçoive un signal.

```
* public static int WaitAny
```

```
(  
    WaitHandle[] waitHandles  
)
```

Attend qu'un des éléments du tableau spécifié reçoive un signal

```
* public static bool WaitAll
```

```
(  
    WaitHandle[] waitHandles  
)
```

Attend que tous les éléments du tableau spécifié reçoivent un signal.

```
bool IsCompleted {get;}
```

Obtient une indication précisant si l'opération asynchrone est terminée.

Comme on l'a dit, cet objet IAsyncResult identifie la méthode, puisque la méthode soeur :

```
public void EndConnect( IAsyncResult asyncResult);
```

réclame l'objet en question; ceci revêt de l'importance dans le contexte multi-threads sous-jacent, puisque plusieurs threads pourraient utiliser la même méthode en même temps – il faut donc distinguer la méthode à terminer.

Concrètement, si notre client possède une variable membre :

```
private Socket socketC;
```

sa connexion peut se programmer, dans un premier temps, comme suit :

```
client asynchrone : connect()
socketC = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
String nomServeur = ZENomServeur.Text;
IPHostEntry he = Dns.GetHostByName(nomServeur);
IPAddress[] listeAdresses = he.AddressList;
try
{
    socketC.BeginConnect (new IPPEndPoint(listeAdresses[0], 5000),
                         new AsyncCallback(ConnexionRealisee),
                         socketC);
    ...
}
catch (Exception ex)
{
    Console.WriteLine("Erreur : " + ex.ToString());
}
```

avec

```
static void ConnexionRealisee (IAsyncResult ar)
{
    Socket socketC = (Socket)ar.AsyncState;
    socketC.EndConnect(ar);
    ...
}
```

### 7.3 La synchronisation : version avec événements

Une fois que l'on a bien mesuré que l'on travaille dans un environnement multithread, on n'est pas vraiment étonné de voir apparaître les outils classiques de synchronisation comme les variables de condition qui permettent le synchronisation des threads Posix. Ces variables de condition se retrouvent dans les classes (de l'espace de noms System.Threading) :

- ◆ public sealed class **ManualResetEvent** : WaitHandle

Se produit lorsqu'un ou plusieurs threads en attente sont avertis qu'un événement a eu lieu.  
Cette classe ne peut pas être héritée.

- ◆ public sealed class **AutoResetEvent** : WaitHandle  
Avertit un ou plusieurs threads en attente qu'un événement s'est produit. Cette classe ne peut pas être héritée.

.NET préfère parler d'objets "événements" – ils peuvent se trouver dans l'état "non signalé" (false) ou "signalé" (true), ce qui revient à dire que l'événement attendu respectivement ne s'est pas encore produit ou au contraire vient d'avoir lieu. La notification, c'est-à-dire le passage à l'état "signalé", se fait au moyen de la méthode :

```
public bool Set();
```

La différence entre les deux types d'objets de synchronisation réside dans le fait que les objets "Auto" sont automatiquement remis dans un l'état "non signalé" une fois qu'un thread en attente sur eux s'est libéré (méthode **WaitOne()** ci-dessous) – c'est le comportement de base des variables de condition. Les objets "Manual" réclament l'utilisation de la méthode

```
public bool Reset();
```

qui retourne true si la fonction aboutit ; sinon false.

L'attente sur la variable de condition – pardon, sur l'événement - se programme par la méthode :

```
public virtual bool WaitOne();
```

Cette attente est bloquante et indéfinie (encore qu'il existe des méthodes à time-out).

Notre client réalisera donc les opérations suivantes pour une connexion à un serveur, si on a déclaré la variable membre statique :

```
public static AutoResetEvent ConnexionEffectuee = new AutoResetEvent(false);
```

<b>client asynchrone (2) : connect()</b>
socketC = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp); String nomServeur = ZENomServeur.Text; IPHostEntry he = Dns.GetHostByName(nomServeur); IPAddress[] listeAdresses = he.AddressList; try { socketC.BeginConnect (new IPEndPoint(listeAdresses[0], 5000), new AsyncCallback( <u>ConnexionRealisee</u> ), socketC); ConnexionEffectuee.WaitOne(); } catch (Exception ex) { Console.WriteLine("Erreur : " + ex.ToString()); }

avec

```
static void ConnexionRealisee (IAsyncResult ar)
{
    Socket socketC = (Socket)ar.AsyncState;
    socketC.EndConnect(ar);
    ConnexionEffectuee.Set();
}
```

### **Remarque**

On peut parfaitement se passer des fonctions callback en passant des arguments null comme 2<sup>ème</sup> et 3<sup>ème</sup> paramètres à la méthode BeginConnect().

### **7.4 Les envois et réceptions asynchrones**

D'une manière similaire, on peut programmer les envois et les réceptions de messages de manière asynchrone au moyen des méthodes :

```
public IAsyncResult BeginSend (byte[] buffer, int offset, int size, SocketFlags socketFlags,
                             AsyncCallback callback, object state);
public IAsyncResult BeginReceive (byte[] buffer, int offset, int size,
                                 SocketFlags socketFlags, AsyncCallback callback, object state);
```

associées aux méthodes :

```
public int EndSend (IAsyncResult asyncResult );
public int EndReceive ( IAsyncResult asyncResult);
```

Les deux derniers paramètres de ces méthodes BeginXXX() sont déjà bien connus. Le premier paramètre désigne évidemment les bytes à envoyer ou l'emplacement pour les bytes reçus; les deux suivants désignent le déplacement éventuel dans le tableau de bytes et le nombre de bytes à envoyer ou à recevoir. Le 4<sup>ème</sup> paramètre est une variable enumération de type **SocketFlags** dont les valeurs possibles ont un lourd relent de protocoles TCP/IP; citons simplement :

DontRoute	Pour ne pas utiliser de table de routage.
OutOfBand	Pour les caractères urgents
Peek	Pour une lecture non destructive du message entrant.

Notre client gèrera donc ses communications de la manière suivante, si on a déclaré les deux variables membres de classe :

```
public static AutoResetEvent EnvoieEffectue = new AutoResetEvent (false);
public static AutoResetEvent ReceptionEffectue = new AutoResetEvent (false);
```

#### **client asynchrone : send() et receive()**

```
String msgServeur = ZEMsg.Text;
byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
socketC.BeginSend(tb, 0, tb.Length,0, new AsyncCallback(EnvoyerRealise), socketC);
EnvoieEffectue.WaitOne();
```

```
byte[] buf = new byte[100];
socketC.BeginReceive(buf,0,buf.Length, 0 , new AsyncCallback(ReceptionRealisee),
socketC);
ReceptionEffectue.WaitOne();
String msgClient = ASCIIEncoding.Default.GetString(buf, 0, buf.Length);
LMsgRecus.Items.Add(msgClient);
```

avec

```
static void EnvoieRealise (IAsyncResult ar)
{
    Socket socketC = (Socket)ar.AsyncState;
    socketC.EndSend(ar);
    EnvoieEffectue.Set();
}
static void ReceptionRealisee (IAsyncResult ar)
{
    Socket socketC = (Socket)ar.AsyncState;
    socketC.EndReceive(ar);
    ReceptionEffectue.Set();
}
```

## 7.5 La synchronisation : version simplifiée

Dans le code présenté ci-dessus, il est clair que le passage par les objets événements donne une impression de luxe inutile. De fait, on peut très bien, dans les cas simples de synchronisation, se passer de ces objets en se souvenant que les méthodes BeginXXX() renvoient un objet implémentant IAsyncResult et que les méthodes EndXXX() peuvent se mettre en attente sur cet objet qui comporte toutes les armes de la synchronisation.

Ce qui nous donne donc :

<b>client asynchrone (3) : connect()</b>
<pre>socketC = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp); String nomServeur = ZENomServeur.Text; IPHostEntry he = Dns.GetHostByName(nomServeur); IPAddress[] listeAdresses = he.AddressList; try {     IAsyncResult ar = socketC.BeginConnect(new IPEndPoint(listeAdresses[0], 5000),  new AsyncCallback(<u>ConnexionRealisee</u>),  socketC);     <b>socketC.EndConnect(ar);</b> } catch (Exception ex) {     Console.WriteLine("Erreur : " + ex.ToString()); }</pre>

avec

```
static void ConnexionRealisee (IAsyncResult ar)
{
    Socket sock = (Socket) ar.AsyncState;
```

## 7.6 Visualiser les threads sous-jacents

Ainsi que nous l'avons déjà signalé, les fonctions callback lancées par les méthodes asynchrones sont exécutées par un thread appartenant à un pool de threads en attente. Pour nous en convaincre, nous demanderons à ces méthodes d'afficher une identification du thread qui les exécute. Pour cela, nous utiliserons la méthode de classe :

```
public static int GetCurrentThreadId();
```

La valeur renournée est un entier signé sur 32 bits qui est l'identificateur du thread en cours.  
La classe définissant cette méthode est

```
public sealed class AppDomain : MarshalByRefObject, _AppDomain, IEvidenceFactory
```

Cette classe représente un domaine d'application (of course !), c'est-à-dire l'environnement dans lequel s'exécutent l'application considérée; en Java, on aurait parlé d'un "contexte". Le travail des instances de cette classe est en fait de charger et d'exécuter les assemblages .NET, d'isoler les applications les unes des autres et de pouvoir les doter de caractéristiques de sécurité différentes. Lorsqu'un thread s'exécute, il le fait obligatoirement dans un domaine d'application unique.

Concrètement, nous laisserons donc dans notre code des callbacks des appels du type

```
Console.WriteLine("Connect effectué - thread " + AppDomain.GetCurrentThreadId());
```

pour visualiser la présence effective de différents threads.

## 7.7 Le code du client asynchrone

Pour nous résumer, voici le code de notre client écrit selon le paradigme asynchrone (son GUI est analogue à celui de la version synchrone) :

### ClientReseauAs.cs

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;
```

```
namespace ClientReseauAsync
{
    public class ClientReseauAs : System.Windows.Forms.Form
    {
        Socket socketC;

        /*public static ManualResetEvent ConnexionEffectuee =
           new ManualResetEvent(false);
        public static ManualResetEvent EnvoieEffectuee =
           new ManualResetEvent(false);
        public static ManualResetEvent ReceptionEffectuee =
           new ManualResetEvent(false);*/

        private System.Windows.Forms.Button BConnecter;
        private System.Windows.Forms.Button BEnvoyer;
        private System.Windows.Forms.Button BDeconnecter;
        ...
        public ClientReseauAs()
        {
            InitializeComponent();
        }

        protected override void Dispose( bool disposing ) { ... }

        private void InitializeComponent()
        {
            this.BConnecter = new System.Windows.Forms.Button();
            this.BDeconnecter = new System.Windows.Forms.Button();
            this.BEnvoyer = new System.Windows.Forms.Button();
            ...
            //
            // BConnecter
            //
            this.BConnecter.Location = new System.Drawing.Point(280, 8);
            this.BConnecter.Name = "BConnecter";
            this.BConnecter.Text = "Se connecter";
            this.BConnecter.Click += new System.EventHandler
                (this.BConnecter_Click);
            ...
            //
            // BEnvoyer
            //
            this.BEnvoyer.Location = new System.Drawing.Point(56, 128);
            this.BEnvoyer.Name = "BEnvoyer";
            this.BEnvoyer.Text = "Envoyer";
            this.BEnvoyer.Click += new System.EventHandler
                (this.BEnvoyer_Click);
            ...
        }
    }
}
```

```

[STAThread]
static void Main()
{
    Application.Run(new ClientReseauAs());
}

private void BConnecter_Click(object sender, System.EventArgs e)
{
    socketC = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
    String nomServeur = ZENomServeur.Text;
    IPHostEntry he = Dns.GetHostByName(nomServeur);
    IPAddress[] listeAdresses = he.AddressList;
    if (listeAdresses.Length == 0) ZENomServeur.Text = "ERREUR !!!";
    else
    {
        try
        {
            IAsyncResult ar = socketC.BeginConnect(
                new IPEndPoint(listeAdresses[0], 5000),
                new AsyncCallback(ConnexionRealisee),
                socketC);
            Console.WriteLine("BeginConnect - thread " +
                AppDomain.GetCurrentThreadId());
            BConnecter.Enabled=false;
            BDeconnecter.Enabled=true;
            BEnvoyer.Enabled=false;

            socketC.EndConnect(ar);
            BEnvoyer.Enabled=true;
            Console.WriteLine("Connect effectué - thread " +
                AppDomain.GetCurrentThreadId());
        }
        catch (Exception ex)
        {
            Console.WriteLine("Erreur : " + ex.ToString());
        }
    }
}

static void ConnexionRealisee (IAsyncResult ar)
{
    Console.WriteLine("Callback connexion - thread " +
        AppDomain.GetCurrentThreadId());
    Socket sock = (Socket) ar.AsyncState;
    Console.WriteLine("Connexion avec " +
        sock.RemoteEndPoint.ToString());
}

```

```

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgServeur = ZEMsg.Text;
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    Console.WriteLine("Avant BeginSend - thread "
        + AppDomain.GetCurrentThreadId());
    IAsyncResult ar = socketC.BeginSend(tb, 0, tb.Length,0,
        new AsyncCallback(EnvoieRealise), socketC);
    Console.WriteLine("Avant EndSend - thread " +
        AppDomain.GetCurrentThreadId());
    socketC.EndSend(ar);
    if (msgServeur == "EOC")
    {
        BConnecter.Enabled=true;
        BDeconnecter.Enabled=false;
        BEnvoyer.Enabled=false;
        socketC.Shutdown(SocketShutdown.Both);
        socketC.Close();
        return;
    }

    byte[] buf = new byte[100];
    Console.WriteLine("Avant BeginReceive - thread " +
        AppDomain.GetCurrentThreadId());
    ar = socketC.BeginReceive(buf,0,buf.Length, 0 ,
        new AsyncCallback(ReceptionRealisee), socketC);
    Console.WriteLine("Avant EndReceive - thread " +
        AppDomain.GetCurrentThreadId());
    socketC.EndReceive(ar);
    String msgClient = ASCIIEncoding.Default.GetString(buf, 0,
        buf.Length);
    LMsgRecus.Items.Add(msgClient);
}

public static void EnvoieRealise (IAsyncResult ar)
{
    Console.WriteLine("Callback envoi - thread " +
        AppDomain.GetCurrentThreadId());
}

public static void ReceptionRealisee (IAsyncResult ar)
{
    Console.WriteLine("Callback reception - thread " +
        AppDomain.GetCurrentThreadId());
}
}
}

```

Reste évidemment à écrire le serveur correspondant de manière asynchrone ...

## 8. Un serveur TCP asynchrone

### 8.1 Un accept asynchrone

De manière analogue à ce qui est décrit ci-dessus, le serveur peut se mettre en attente de connexion par une "accept asynchrone" usant des méthodes prévisibles :

```
public IAsyncResult BeginAccept ( AsyncCallback callback, object state);  
public Socket EndAccept ( IAsyncResult asyncResult );
```

C'est donc cette dernière méthode qui fournit la socket de service. Notre serveur organise donc sa prise en compte d'une connexion par (non, il n'y a pas de BeginBind() ou de BeginListen() – quel serait l'intérêt ?) en utilisant les variables membres :

```
public Socket socketServ;  
public static AutoResetEvent AcceptEffectue = new AutoResetEvent(false);
```

dans

#### serveur asynchrone : accept()

```
Socket socketEc = new Socket (AddressFamily.InterNetwork, SocketType.Stream,  
ProtocolType.Tcp);  
socketEc.BeginAccept ( new AsyncCallback(AcceptRealise), socketEc);  
AcceptEffectue.WaitOne();
```

avec

```
static void AcceptRealise (IAsyncResult ar)  
{  
    Socket socketC = (Socket)ar.AsyncState;  
    appGui.socketServ = socketC.EndAccept(ar);  
    AcceptEffectue.Set();  
    ...  
}
```

### 8.2 Le code du serveur asynchrone en mode console

Très prudemment, nous allons nous contenter ici d'un serveur sans GUI. La version avec GUI réclamerait simplement de confier les tâches réseau à un thread créé par nos soins, laissant ainsi le thread de base gérer le GUI – à faire ;-) ...

#### ServeurReseauAsCons.cs

```
using System;  
using System.Net;  
  
using System.Net.Sockets;  
using System.Text;  
using System.Threading;
```

```

namespace ServeurAsyncConsole
{
    class ServeurReseauAsCons
    {
        public static byte[] buf = new byte[1024];
        public static AutoResetEvent serveurArrete = new AutoResetEvent(false);

        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("main> démarrage du serveur - thread " +
                AppDomain.GetCurrentThreadId());
            Socket socketEc = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);

            String nomServeur = Dns.GetHostName();
            IPHostEntry he = Dns.GetHostByName(nomServeur);
            IPAddress[] listeAdresses = he.AddressList;
            Console.WriteLine("main> adresse du serveur = " + listeAdresses[0] +
                " / thread " + AppDomain.GetCurrentThreadId());

            socketEc.Bind (new IPEndPoint(listeAdresses[0], 5000));

            socketEc.Listen (100);

            IAsyncResult ar = socketEc.BeginAccept(
                new AsyncCallback(AcceptRealise),
                socketEc);
            Console.WriteLine("main> BeginAccept lancé - thread " +
                AppDomain.GetCurrentThreadId());
            Console.WriteLine("main> Avant attente dans main()");
            ar.AsyncWaitHandle.WaitOne();
            Console.WriteLine("main> Après attente dans main()");
            serveurArrete.WaitOne();
        }

        static void AcceptRealise (IAsyncResult ar)
        {
            Console.WriteLine("callback : AcceptRealise> thread " +
                AppDomain.GetCurrentThreadId());

            Socket socketC = (Socket)ar.AsyncState;
            Socket sockC = socketC.EndAccept(ar);

            IPEndPoint ep = (IPEndPoint)sockC.RemoteEndPoint;

            bool fin = false;
        }
    }
}

```

```

while (!fin)
{
    IAsyncResult e = sockC.BeginReceive(buf, 0, buf.Length,
        0, new AsyncCallback(ReceptionRealisee), sockC);

    Console.WriteLine("callback : AcceptRealise> BeginReceive
        lancé - thread " + AppDomain.GetCurrentThreadId());
    Console.WriteLine("callback : AcceptRealise> Avant attente
        dans AcceptRealise");
    e.AsyncWaitHandle.WaitOne();
    Console.WriteLine("callback : AcceptRealise> Après attente
        dans AcceptRealise");
    String msgCli = ASCIIEncoding.Default.GetString (buf, 0, 3);

    if ( msgCli == "EOC")
    {
        Console.WriteLine("callback : AcceptRealise> fin EOC
            – thread " + AppDomain.GetCurrentThreadId());
        fin = true;
    }
}
Console.WriteLine("callback : AcceptRealise> fin de la boucle –
    thread" + AppDomain.GetCurrentThreadId());
sockC.Shutdown(SocketShutdown.Both);
sockC.Close();
Console.WriteLine("callback : AcceptRealise> fermeture de la socket –
    thread" + AppDomain.GetCurrentThreadId());
serveurArrete.Set();
}

static void ReceptionRealisee (IAsyncResult ar)
{
    Console.WriteLine("callback : ReceptionRealisee> thread " +
        AppDomain.GetCurrentThreadId());
    Socket socketServ = (Socket)ar.AsyncState;
    int n = socketServ.EndReceive(ar);
    String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
    Console.WriteLine("callback : ReceptionRealisee> msg reçu = " +
        msgClient);
    Console.WriteLine("callback : ReceptionRealisee> Entrez la réponse du
        serveur : ");
    String msgServeur = Console.ReadLine();
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    IAsyncResult e = socketServ.BeginSend(tb, 0, tb.Length, 0,
        new AsyncCallback(EnvoiRealise), socketServ);
    Console.WriteLine("callback : ReceptionRealisee> BeginSend lancé –
        thread " + AppDomain.GetCurrentThreadId());
    Console.WriteLine("callback : ReceptionRealisee> Avant attente dans
        ReceptionRealise");
    e.AsyncWaitHandle.WaitOne();
}

```

```
Console.WriteLine("callback : ReceptionRealise> Après attente dans  
ReceptionRealise");  
}  
  
static void EnvoiRealise (IAsyncResult ar)  
{  
    Console.WriteLine("callback : EnvoiRealise> thread " +  
        AppDomain.GetCurrentThreadId());  
    Socket socketServ = (Socket)ar.AsyncState;  
    int n = socketServ.EndSend(ar);  
}  
}  
}
```

### 8.3 Un exemple de conversation

Si le client ressemble à ceci :



le serveur peut répondre de la manière suivante :

```
main> démarrage du serveur - thread 1688  
main> adresse du serveur = 10.59.5.13 / thread 1688  
main> BeginAccept lancé - thread 1688  
main> Avant attente dans main()  
main> Après attente dans main()  
callback : AcceptRealise> thread 1336  
callback : AcceptRealise> BeginReceive lancé - thread 1336  
callback : AcceptRealise> Avant attente dans AcceptRealise  
callback : ReceptionRealisee> thread 1692  
callback : ReceptionRealisee> msg reçu = Bonjour !  
callback : ReceptionRealisee> Entrez la réponse du serveur :  
callback : AcceptRealise> Après attente dans AcceptRealise
```

```
callback : AcceptRealise> BeginReceive lancé - thread 1336
callback : AcceptRealise> Avant attente dans AcceptRealise
Hello !!!
callback : ReceptionRealisee> BeginSend lancé - thread 1692
callback : ReceptionRealisee> Avant attente dans ReceptionRealise
callback : EnvoiRealise> thread 268
callback : ReceptionRealisee> Après attente dans ReceptionRealise
callback : AcceptRealise> Après attente dans AcceptRealise
callback : AcceptRealise> BeginReceive lancé - thread 1336
callback : AcceptRealise> Avant attente dans AcceptRealise
callback : ReceptionRealisee> thread 1692
callback : ReceptionRealisee> msg reçu = Vous ici ?
callback : ReceptionRealisee> Entrez la réponse du serveur :
Mais oui - pour 3 semaines !
callback : ReceptionRealisee> BeginSend lancé - thread 1692
callback : ReceptionRealisee> Avant attente dans ReceptionRealise
callback : ReceptionRealisee> Après attente dans ReceptionRealise
callback : EnvoiRealise> thread 268
...
...
```

## 9. Un client TCP simplifié

Dans le contexte de .NET, nous avons essentiellement travaillé au niveau de la couche réseau, puisque les sockets apparaissaient explicitement dans notre programmation. Fort heureusement, le kit de développement de .NET fournit aussi le moyen de faire abstraction des sockets pour ne travailler qu'à partir de la couche transport.

### 9.1 La classe **TCPClient**

Ainsi, il propose la classe **TCPClient** qui gère une connexion client pour des services réseau TCP. Le constructeur le plus simple est celui "à la java" ;-), qui réalise automatiquement une connexion :

```
public TcpClient ( string hostname, int port );
```

Mais il existe aussi :

```
public TcpClient ( IPEndPoint localEP );
```

et même

```
public TcpClient();
```

Dans ce dernier cas, il faut alors programmer dans la foulée un appel à :

```
public void Connect ( string hostname, int port );
ou
public void Connect( IPEndPoint remoteEP );
```

Bien sûr, en cas de problème de connexion, une exception **SocketException** est lancée.

## 9.2 Un flux réseau

Une fois la connexion réalisée, il reste à acquérir le moyen de faire transiter des informations entre le client et le serveur atteint : ce sera bien entendu un flux qui sera chargé de ce travail – comme en Java, donc ;-) ... On peut obtenir ce flux sous-jacent au moyen de la méthode de TcpClient :

```
public NetworkStream GetStream();
```

La classe instanciée par l'objet ainsi obtenu

public class **NetworkStream** : Stream

comporte les méthodes attendues :

```
public override void Write ( byte[] buffer, int offset, int size );
public override int Read ( in byte[] buffer, int offset, int size);
```

Autrement dit, ce flux travaille essentiellement avec des bytes !

## 9.3 Le code du client TCP

Point n'est sans doute nécessaire de disposer plus longtemps pour pouvoir écrire un client TCP classique, avec un interface graphique similaire à ceux des exemples précédents :

### ClientReseauTCP.cs

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;
using System.IO;

namespace ClientReseauTCPClient
{
    public class ClientReseauTCP : System.Windows.Forms.Form
    {
        TcpClient cli;
        NetworkStream ns;

        private System.Windows.Forms.Button BConnecter;
        private System.Windows.Forms.Button BEnvoyer;
        private System.Windows.Forms.Button BDeconnecter;
        ...
        public ClientReseauTCP()
        {
            InitializeComponent();
            BConnecter.Enabled=true; BDeconnecter.Enabled=false;
            BEnvoyer.Enabled=false;
        }
    }
}
```

```

protected override void Dispose( bool disposing ) { ... }

private void InitializeComponent()
{
    this.BConnecter = new System.Windows.Forms.Button();
    this.BDeconnecter = new System.Windows.Forms.Button();
    this.BEnvoyer = new System.Windows.Forms.Button();
    ...
    //
    // BConnecter
    //
    this.BConnecter.Location = new System.Drawing.Point(280, 8);
    this.BConnecter.Name = "BConnecter";
    this.BConnecter.Text = "Se connecter";
    this.BConnecter.Click += new System.EventHandler
        (this.BConnecter_Click);
    //
    // BDeconnecter
    //
    this.BDeconnecter.Location = new System.Drawing.Point(280, 40);
    this.BDeconnecter.Name = "BDeconnecter";
    this.BDeconnecter.Text = "Se déconnecter";
    this.BDeconnecter.Click += new System.EventHandler
        (this.BDeconnecter_Click);
    //
    // BEnvoyer
    //
    this.BEnvoyer.Location = new System.Drawing.Point(56, 128);
    this.BEnvoyer.Name = "BEnvoyer";
    this.BEnvoyer.Text = "Envoyer";
    this.BEnvoyer.Click += new System.EventHandler
        (this.BEnvoyer_Click);
    ...
}

[STAThread]
static void Main()
{
    Application.Run(new ClientReseauTCP());
}

private void BConnecter_Click(object sender, System.EventArgs e)
{
    Console.WriteLine("Tentative de connexion au serveur");
    String nomServeur = ZENomServeur.Text;
    int portServeur = int.Parse(ZEPortServeur.Text.ToString());
}

```

```

try
{
    cli = new TcpClient(nomServeur, portServeur);
}
catch (Exception ex )
{
    Console.WriteLine(ex.ToString());
}
ns = cli.GetStream();

BConnecter.Enabled=false;BDisconnecter.Enabled=true;
BEnvoyer.Enabled=true;
}

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgServeur = ZEMsg.Text;
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    ns.Write(tb, 0, tb.Length);

    if (msgServeur.Equals("EOC"))
    {
        BConnecter.Enabled=true; BDisconnecter.Enabled=false;
        BEnvoyer.Enabled=false;
        ns.Close();cli.Close();
        return;
    }

    byte[] buf = new byte[100];
    int n = ns.Read(buf, 0, 100);
    String msgClient = ASCIIEncoding.Default.GetString(buf, 0,
        buf.Length);
    LMsgRecus.Items.Add(msgClient);
}

private void BDisconnecter_Click(object sender, System.EventArgs e)
{
    BConnecter.Enabled=true; BDisconnecter.Enabled=false;
    BEnvoyer.Enabled=false;
    String msgServeur = "EOC";
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    ns.Write(tb, 0, tb.Length);
    ns.Close(); cli.Close();
    Console.WriteLine("close effectué");
}
}
}

```

Un point mérite néanmoins d'être remarqué. En effet, la fermeture de la connexion réclame successivement

- ◆ la fermeture du flux réseau;
- ◆ la fermeture de la socket elle-même.

Oublier de fermer le flux a comme effet que la fermeture de la socket est sans effet !

## 10. Un serveur TCP simplifié

Nous allons retrouver ici le même niveau d'encapsulation.

### 10.1 La classe TCPListener

Tout comme pour le client, .NET nous propose une classe **TCPListener** incorporant les fonctionnalités serveur. Cette classe réalise donc l'écoute et la prise en compte des connexions de clients réseau TCP. Les constructeurs sont assez prévisibles :

```
public TcpListener ( int Erreur ! Référence de lien  
hypertexte non valide. );  
public TcpListener ( IPPEndPoint Erreur ! Référence  
de lien hypertexte non valide. );  
public TcpListener ( IPAddress Erreur ! Référence  
de lien hypertexte non valide., int Erreur !  
Référence de lien hypertexte non valide.);
```

La première version suppose évidemment que c'est l'adresse de la machine hôte qui doit être utilisée pour le futur serveur. Cependant, rien n'est encore fait du point de vue de la socket sous-jacente. C'est le rôle de la méthode

```
public void Start();
```

de réaliser toutes les opérations préliminaires nécessaires à la prise en compte d'un client :

- ◆ initilisation de l'objet Socket;
- ◆ réalisation du bind en utilisant un IPPEndPoint sous-jacent;
- ◆ mise à l'écoute de demandes de connexions.

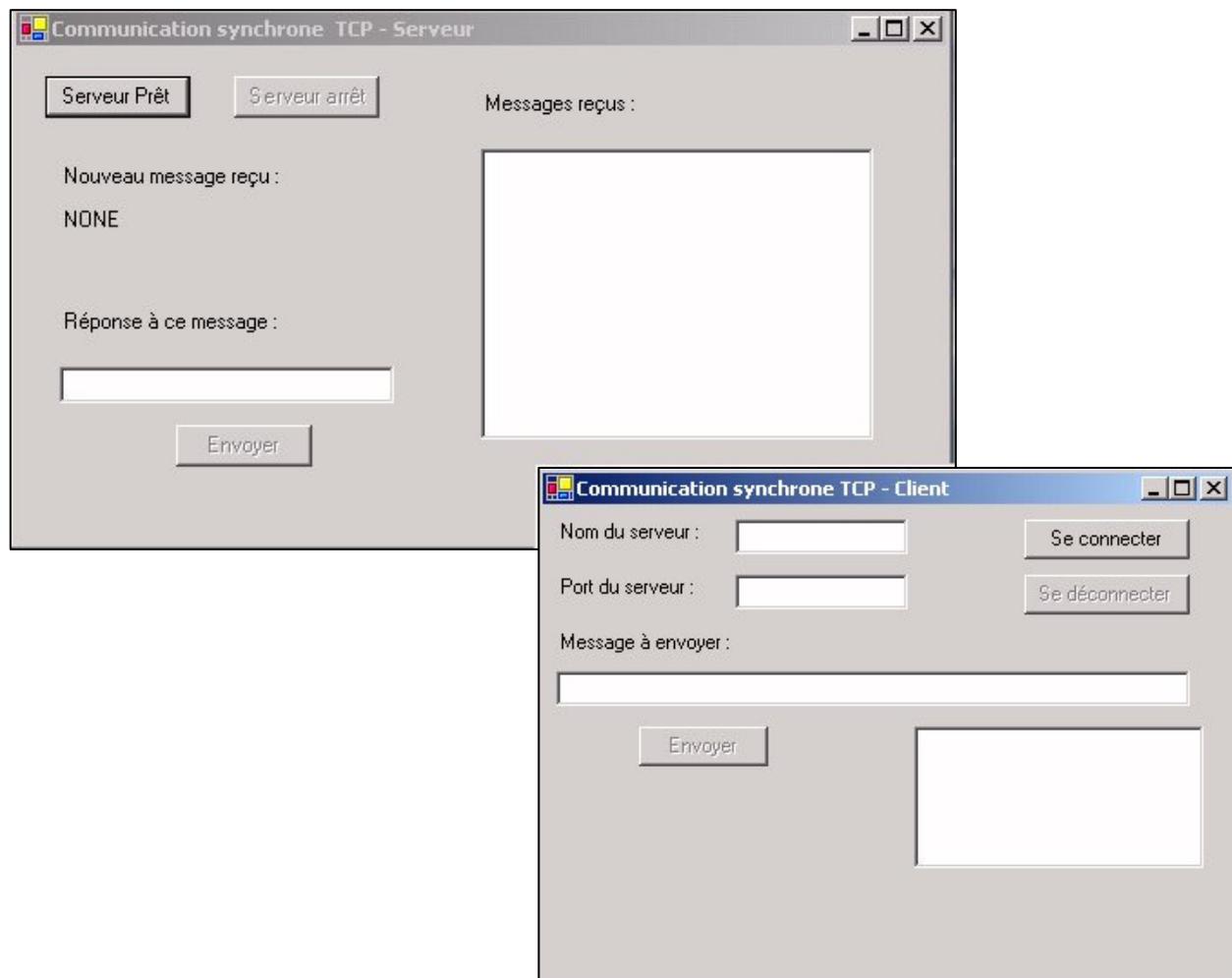
Une connexion est prise en compte au moyen de la méthode :

```
public TcpClient AcceptTcpClient();
```

et il suffit d'utiliser cet objet TcpClient comme dans le cas du client.

### 10.2 Le code du serveur TCP

Nous allons donc imaginer que le client du point précédent se connecte sur un serveur ayant le look suivant :



### ServeurReseauTCP.cs

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.IO;

namespace ServeurReseauTCPListener
{
    public class ServeurReseauTCP : System.Windows.Forms.Form
    {
        TcpListener ser;
        NetworkStream ns;
        TcpClient cli;

        private System.Windows.Forms.Button BDemarrer;
        private System.Windows.Forms.Button BEnvoyer;
        private System.Windows.Forms.Button BArreter;
```

```
public ServeurReseauTCP()  
{  
    InitializeComponent();  
    BDemarrer.Enabled=true; BArreter.Enabled=false;  
    BEnvoyer.Enabled=false;  
}  
  
protected override void Dispose( bool disposing ) { ... }  
private void InitializeComponent()  
{  
    this.BDemarrer = new System.Windows.Forms.Button();  
    this.BArreter = new System.Windows.Forms.Button();  
    this.BEnvoyer = new System.Windows.Forms.Button();  
    ...  
    //  
    // BDemarrer  
    //  
    this.BDemarrer.Location = new System.Drawing.Point(16, 16);  
    this.BDemarrer.Name = "BDemarrer";  
    this.BDemarrer.Text = "Serveur Prêt";  
    this.BDemarrer.Click += new System.EventHandler  
        (this.BDemarrer_Click);  
    //  
    // BArreter  
    //  
    this.BArreter.Location = new System.Drawing.Point(120, 16);  
    this.BArreter.Name = "BArreter";  
    this.BArreter.Text = "Serveur arrêté";  
    this.BArreter.Click += new System.EventHandler(this.BArreter_Click);  
    //  
    // BEnvoyer  
    //  
    this.BEnvoyer.Location = new System.Drawing.Point(88, 208);  
    this.BEnvoyer.Name = "BEnvoyer";  
    this.BEnvoyer.Text = "Envoyer";  
    this.BEnvoyer.Click += new System.EventHandler  
        (this.BEnvoyer_Click);  
    ...  
}  
  
[STAThread]  
static void Main()  
{  
    Application.Run(new ServeurReseauTCP());  
}  
  
private void BDemarrer_Click(object sender, System.EventArgs e)  
{  
    BDemarrer.Enabled=false; BArreter.Enabled=true;  
    BEnvoyer.Enabled=true;
```

```

ser = new TcpListener(5000);
ser.Start();
cli = ser.AcceptTcpClient();
ns = cli.GetStream();

byte[] buf = new byte[100];
int n = ns.Read(buf, 0, 100);
String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
ZTMessageRecu.Text = msgClient; LMMsgRecus.Items.Add(msgClient);
}

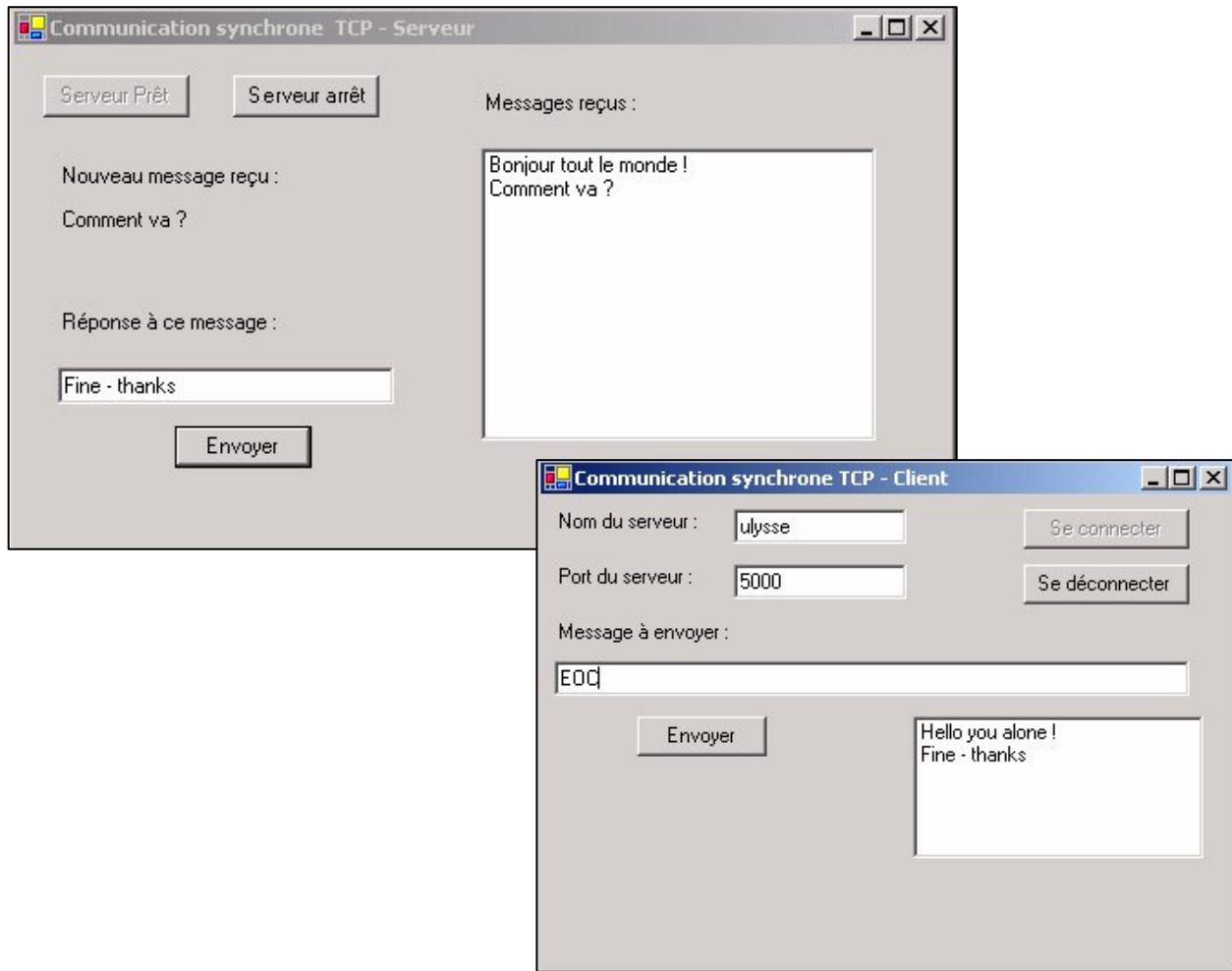
private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgServeur = ZEMsgReponse.Text;
    byte[] tb = ASCIIEncoding.Default.GetBytes(msgServeur);
    ns.Write(tb, 0, tb.Length);

    byte[] buf = new byte[100];
    int n = ns.Read(buf, 0, 100);
    String msgClient = ASCIIEncoding.Default.GetString(buf, 0, n);
    ZTMessageRecu.Text = msgClient; LMMsgRecus.Items.Add(msgClient);
    if (msgClient == "EOC")
    {
        Console.WriteLine("Message de fin de communication reçu");
        try
        {
            ns.Close(); cli.Close();
            String msg = "fermeture de la socket réalisée";
            ser.Stop();
            BDemarrer.Enabled=true; BArreter.Enabled=false;
            BEnvoyer.Enabled=false;
        }
        catch (SocketException ex)
        {
            String msg = "Erreur de fermeture de la socket";
            ZTMessageRecu.Text = msg;
            LMMsgRecus.Items.Add(msgClient);
        }
    }
}

private void BArreter_Click(object sender, System.EventArgs e)
{
    BDemarrer.Enabled=true; BArreter.Enabled=false;
    BEnvoyer.Enabled=false;
    ns.Close(); cli.Close(); ser.Stop();
}
}

```

Un exemple d'exécution serait :



## 11. Un client C# / .NET pour un serveur multithread C / UNIX

Tout comme nous avons connecté un client écrit en Java avec le serveur multithread tournant sur la machine Unix Copernic et développé dans les premiers chapitres, nous allons à présent connecter à ce même serveur (nommé `tcpiter07-java.c` – voir chapitre IV paragraphe 5) un client écrit en C#.

### 11.1 Les flux binaires de .NET

Lorsqu'il s'agit d'échanger d'autres informations que des bytes, il faut utiliser des flux réseaux appropriés qui reconnaissent les types de données et tiennent compte d'un schéma d'encodage. En fait, en accord avec ce que nous avons réalisé pour le client Java, nous allons passer toutes les données sous forme de leur "image" en chaîne de caractères. Néanmoins, nous avons ici l'occasion de rencontrer ces flux .NET assez proches des `DataOutputStream` et `DataInputStream` de Java. Ces flux sont des instances des classes **BinaryWriter** et **BinaryReader**. Leur constructeur réclame le flux TCP sur lesquels ils vont s'appuyer :

```
public BinaryWriter (Stream output);
public BinaryReader (Stream input);
```

le flux de base étant d'ailleurs une propriété de la classe :

```
public virtual Stream BaseStream { get; }
```

Leurs méthodes polymorphes Write() et Read() permettent d'écrire ou de lire n'importe quel type de données, sans problèmes particulier de manipulations de tableaux de bytes. Ainsi, par exemple :

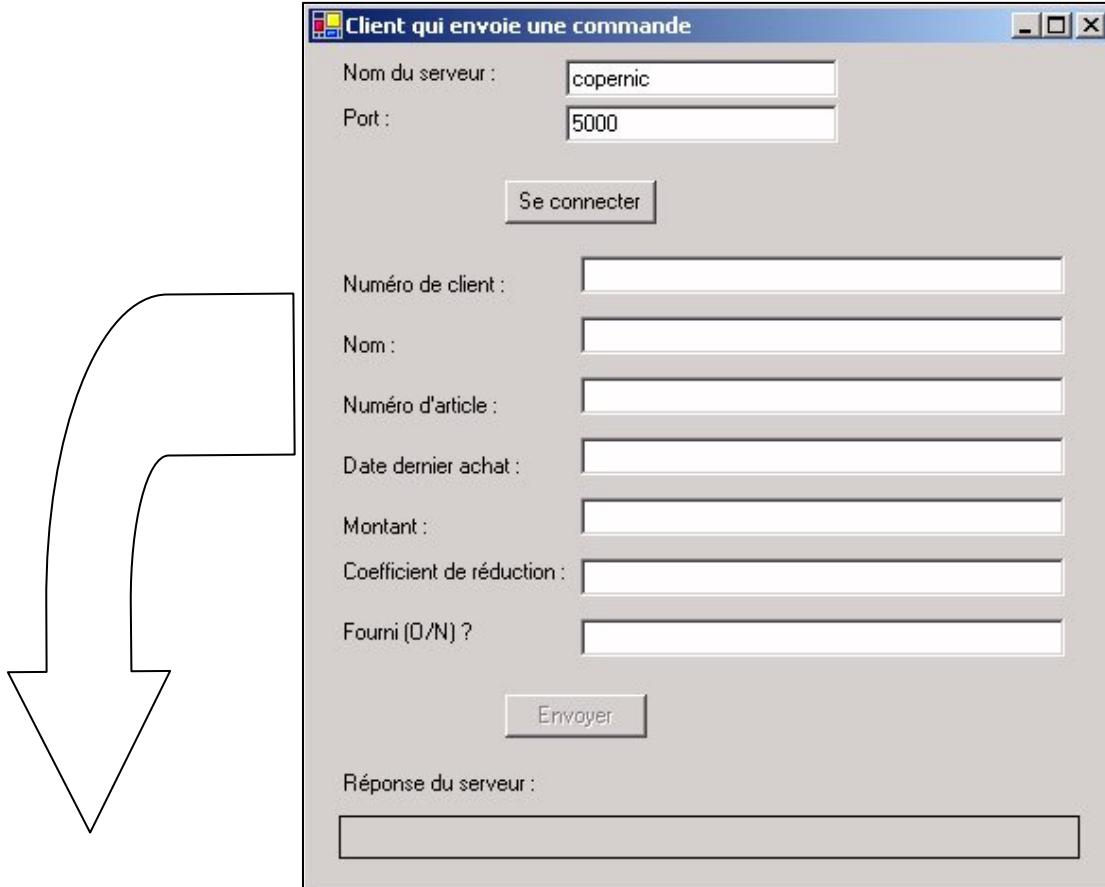
<b>BinaryWriter</b>	
public virtual void <b>Write</b> ( byte[] <i>buffer</i> )	Écrit le tableau de bytes et avance la position dans le flux du nombre de bytes correspondants
public virtual void <b>Write</b> ( int <i>value</i> )	Écrit un entier signé de 4 octets dans le flux et avance la position dans le flux de 4 octets.
public virtual void <b>Write</b> ( string <i>value</i> )	Écrit dans le flux une chaîne (préfixée par sa longueur) en utilisant le schéma d'encodage déterminé par l'objet Encoding courant et avance la position actuelle dans le flux en fonction de ce schéma et des caractères spécifiques écrits.
<b>BinaryReader</b>	
public virtual byte[] <b>ReadBytes</b> ( int <i>n</i> )	Lit <i>n</i> octets dans le flux, les place dans un tableau d'octets et avance la position dans le flux de <i>n</i> octets.
public virtual int <b>ReadInt32()</b>	Lit un entier signé de 4 octets dans le flux et avance la position dans le flux de 4 octets.
public virtual string <b>ReadString()</b>	Lit un String dans le flux et avance la position dans le flux du nombre d'octets correspondant.

## 11.2 Le client du serveur multithread UNIX

Comme nous l'avons dit, nous allons donc reprendre notre serveur multithread dans sa version "structure". L'information envoyée par le client est donc de la forme (en syntaxe C) :

```
struct client
{
    char numClient[20];
    char nom[30];
    char numArticle[15];
    char dateDernierAchat[11];
    char montant[10];
    char coeffReduction[10];
    char fourni;
};
```

Pour faire court, cette structure ne donnera pas naissance à une classe C# - mais rien n'empêche de la faire comme en Java afin d'encapsuler les divers mécanismes de transformation (à faire ...). Car mécanismes il y aura : en fait, nous allons simplement récupérer les informations fournies par le client dans son GUI et construire une chaîne de bytes respectant le découpage de la structure C attendue par le serveur :



numClient	nom	numArticle	montant	fourni
			dateDernierAchat	coeffReduction
20	30	15	11	10
0	20	50	65	76

Pour en arriver là, il nous faudra donc, pour chaque champ, créer un tableau de bytes de la taille correspondante au champ de la structure C et y recopier la données lue, afin de ne pas modifier la taille du tableau, au moyen de la méthode

```
public virtual void CopyTo (Array array, int index);
```

appelée par le tableau de bytes que l'on recopie (on se souviendra que la classe abstraite `Array` est la mère des tableaux du CLR de .NET).

Ensuite, il restera à recopier ces tableaux intermédiaires dans un tableau dont la taille est la longueur de la structure C (ici, 97).

### 11.3 Le code du client C#

Nous pouvons dès lors utiliser le petit client suivant :

**ClientReseauUnix.cs**

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;
using System.IO;

namespace WindowsApplication1
{
    public class ClientReseauUnix: System.Windows.Forms.Form
    {
        TcpClient cli;
        NetworkStream ns;
        BinaryWriter bw;
        BinaryReader br;

        private System.Windows.Forms.Label label1;
        ...
        private System.Windows.Forms.Button BEnvoyer;
        private System.Windows.Forms.Button BConnecter;
        private System.Windows.Forms.TextBox ZENomServeur;
        private System.Windows.Forms.TextBox ZEPortServeur;
        private System.Windows.Forms.TextBox ZENumeroClient;
        private System.Windows.Forms.TextBox ZENom;
        ...

        public ClientReseauUnix
```

```

this.BEnvoyer.Click += new System.EventHandler
    (this.BEnvoyer_Click);
...
//
// BConnecter
//
this.BConnecter.Location = new System.Drawing.Point(104, 72);
this.BConnecter.Name = "BConnecter";
this.BConnecter.Text = "Se connecter";
this.BConnecter.Click += new System.EventHandler
    (this.BConnecter_Click);
...
}

[STAThread]
static void Main()
{
    Application.Run(new ClientReseauUnix());
}

private void BConnecter_Click(object sender, System.EventArgs e)
{
    Console.WriteLine("Tentative de connexion au serveur");
    String nomServeur = ZENomServeur.Text;
    int portServeur = int.Parse(ZEPortServeur.Text.ToString());

    try
    {
        cli = new TcpClient(nomServeur, portServeur);
    }
    catch (Exception ex )
    {
        Console.WriteLine("Echec de la connexion : " + ex.ToString());
    }
    ns = cli.GetStream();
    bw = new BinaryWriter(ns);
    br = new BinaryReader(ns);

    BConnecter.Enabled=false;
    BEnvoyer.Enabled=true;
}

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    // création des tableaux de bytes de chaque champ
    String numeroClient = ZENumeroClient.Text;
    byte[] tr = ASCIIEncoding.Default.GetBytes(numeroClient);
    byte[] nc = new byte[20];
    tr.CopyTo(nc, 0);
    Console.WriteLine("Numéro de client : " + nc + " (" + tr.Length);
}

```

```

String nom = ZENom.Text;
tr = ASCIIEncoding.Default.GetBytes(nom);
byte[] n = new byte[30];
tr.CopyTo(n, 0);
Console.WriteLine("Nom de client : " + n + " (" + tr.Length);

String numeroArticle = ZENumeroArticle.Text;
tr = ASCIIEncoding.Default.GetBytes(numeroArticle);
byte[] na = new byte[15];
tr.CopyTo(na, 0);
Console.WriteLine("Numéro d'article : " + na + " (" + tr.Length);

String dateDernierAchat = ZEDateDernierAchat.Text;
tr = ASCIIEncoding.Default.GetBytes(dateDernierAchat);
byte[] dda = new byte[11];
tr.CopyTo(dda, 0);
Console.WriteLine("Date dernier achat : " + dda + " (" + tr.Length);

String montant = ZEMontant.Text;
tr = ASCIIEncoding.Default.GetBytes(montant);
byte[] m = new byte[10];
tr.CopyTo(m, 0);
Console.WriteLine("Montant : " + m + " (" + tr.Length);

String coeffReduction = ZECoeffReduc.Text;
tr = ASCIIEncoding.Default.GetBytes(coeffReduction);
byte[] cr = new byte[10];
tr.CopyTo(cr, 0);
Console.WriteLine("Coefficient de réduction : " + cr + " (" + tr.Length);

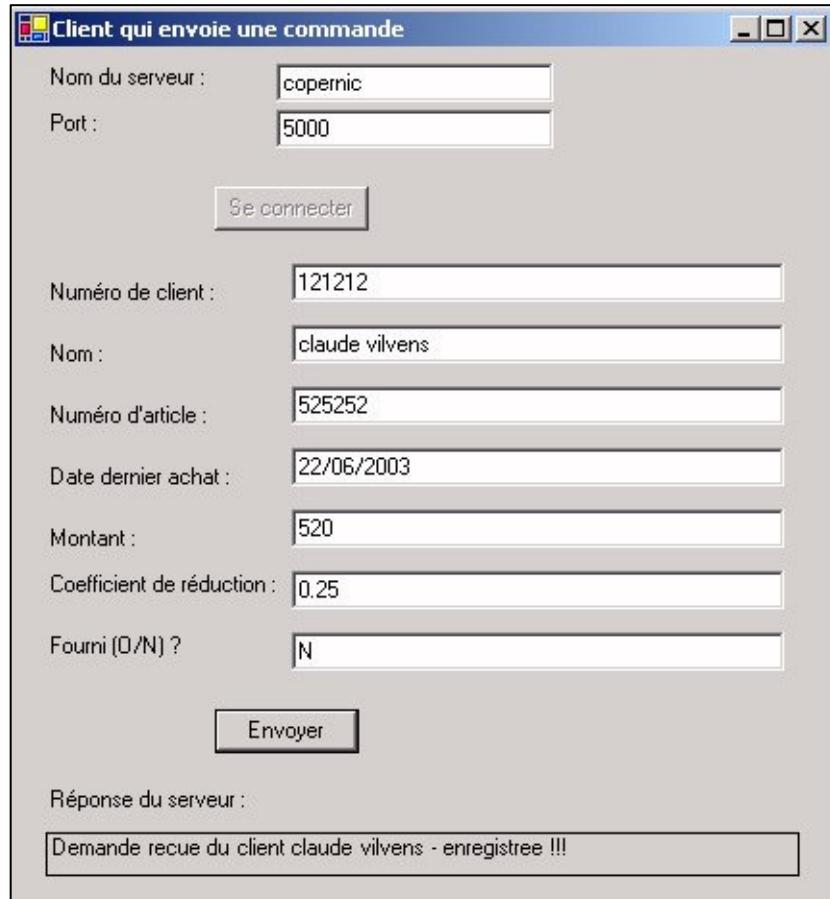
String fourni = ZEFourni.Text;
tr = ASCIIEncoding.Default.GetBytes(fourni);
byte[] f = new byte[1];
tr.CopyTo(f, 0);
Console.WriteLine("Fourni : " + f + " (" + tr.Length);

// création du tableau de bytes qui sera envoyé
byte[] msgClient = new byte [97];
nc.CopyTo(msgClient, 0);
n.CopyTo(msgClient, 20);
na.CopyTo(msgClient, 50);
dda.CopyTo(msgClient,65);
m.CopyTo(msgClient, 76);
cr.CopyTo(msgClient, 86);
f.CopyTo(msgClient, 96);
Console.WriteLine("msgClient : " + f + " (" + msgClient.Length);
bw.Write(msgClient);
Console.WriteLine("msgClient envoyé !");

```

```
        byte[] msgServeur = br.ReadBytes(100);
        Console.WriteLine("msgServeur reçu !");
        String reponse =
            ASCIIEncoding.Default.GetString(msgServeur,0,100);
        Console.WriteLine("Réponse du serveur : " + reponse);
        ZTReponseServeur.Text = reponse;
    }
}
```

Si le client s'exécute avec les données suivantes :



le serveur répondra ainsi :

```
21 /prof/vilvens/tcp#c recreecom  
Thread principal serveur demarre  
identite = 162204.3222999712  
--- Recreation du fichier des commandes ---  
Creation de la socket OK  
Acquisition infos host OK  
Adresse IP = 10.59.5.9  
Bind adresse et port socket OK  
Thread principal : en attente d'une connexion  
Listen socket OK  
Accept socket OK  
Socket de service attribuee = 4  
Thread secondaire lance !  
Marquage pour effacement du thread secondain  
Thread principal : en attente d'une connexion
```

```
Listen socket OK
** vr = 4
th_4> Debut de thread
th_4> identite = 162204.15825024
th_4> je travaille sur la socket de service 4
Taill msg = 97 et nbreBytes = 97
th_4> Recv socket OK
----- Analyse de msgClient
th_4> Demande recue pour le client = claude vilvens
th_4> Enregistrement de la commande
th_4> Demande recue du client claude vilvens - enregistree !!!
*** Requete d'un client ***
Numero de client : 121212
Nom = claude vilvens
Date du dernier achat : 22/06/2003
Numero de l'article demande : 525252
et son prix : 520
Coefficient de reduction = 0.25
Fourni ? = N
th_4> Send socket OK
th_4> Socket connectee au client fermee
th_4> --fin du thread--
```



La suite TCP/IP comporte un autre protocole de transport : UDP. Le moment est venu de s'en souvenir ...

## VII. Les sockets UDP



*Prenez garde à vous : si vous continuez à être de bonne foi, nous allons être d'accord.*

(Stendhal, Racine et Shakespeare)

### 1. Les caractéristiques d'UDP

Pour rappel, UDP est un protocole **non fiable et orienté sans connexion** (comme une lettre qui est déposée dans la boîte de son destinataire, mais sans garantie que celui-ci relève sa boîte aux lettres).

UDP envoie donc des **datagrammes** d'une machine à l'autre *sans garantie de réception*. Ceci implique une complexité moindre, puisque qu'il n'y a plus de mécanisme d'accusés de réception à gérer, et un trafic réseau réduit (8 bytes d'en-tête au lieu de 20 pour TCP – voir p.21). Par contre, UDP est si simple qu'il ne garantit pas non plus l'ordre de réception des données ni leur longueur. On parle encore de "livraison au mieux" [*best effort*]. L'application basée sur une programmation UDP devra donc

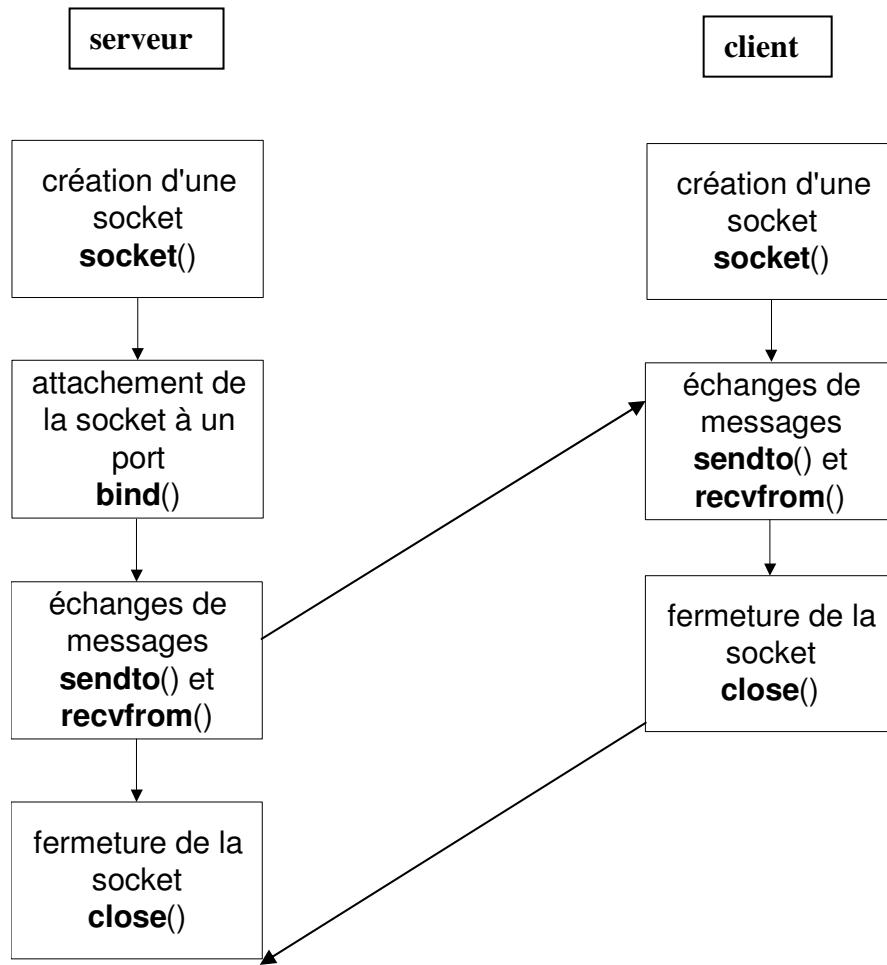
- ◆ connaître le MTU afin d'éviter une fragmentation par IP en n'utilisant que des données de taille inférieure;
- ◆ numéroter ses données si l'ordre de réception a une importance.

La fait d'être orienté sans connexion permet évidemment le **multicast**, c'est-à-dire le fait de viser tout un groupe d'adresses destinataires (une espèce de broadcast limité). UDP simplifie aussi la programmation, puisque les applications désirant échanger des informations doivent simplement chacune

- ◆ créer une socket;
- ◆ la faire reconnaître par le système, du moins dans le cas du serveur;
- ◆ échanger les données (avec les précautions évoquées ci-dessus);
- ◆ fermer leur socket.

Il importe donc de remarquer la relative symétrie existante dans la programmation réseau d'un serveur et d'un client : tous deux ont besoin d'un numéro de port que l'autre interlocuteur connaît. Il ne peut être question de laisser le système choisir un port quelconque pour le serveur, puisque le client doit connaître celui-ci explicitement pour l'utiliser dans ses requêtes. Par contre, le serveur recevra avec chaque message l'adresse de son émetteur et pourra donc utiliser celle-ci pour lui répondre.

En général, les serveurs utilisant les datagrammes, donc UDP, sont itératifs : ils fonctionnent selon un schéma question/réponse.



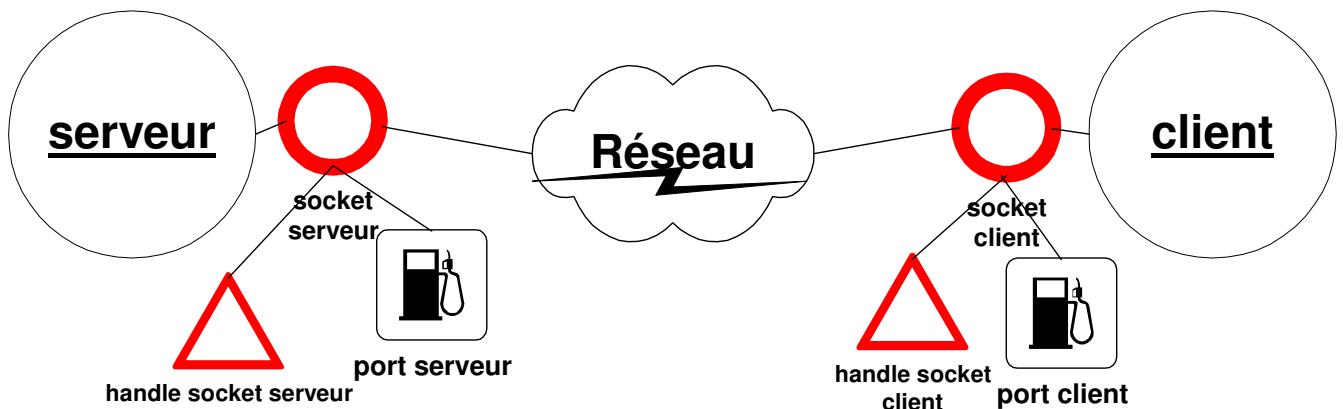
## 2. La création d'une socket

La création d'une socket UDP est très similaire à celle d'une socket UDP. Il faut tout d'abord obtenir du système d'exploitation local un handle sur une socket au moyen de la primitive `socket()`. Cette socket sera

- ◆ de domaine **AF\_INET**;
- ◆ de type **SOCK\_DGRAM** (type du mode sans connexion);
- ◆ de protocole UDP (c'est le protocole par défaut de ce mode – il s'agit donc de **IPPROTO\_UDP**).

Il s'agit ensuite de faire connaître du système l'adresse et le port utilisés. Ici non plus, il n'y a pas de véritable nouveauté : on utilise les fonctions `gethostbyname()`, `hton()` et `bind()` avec une structure `sockaddr_in`.

La situation est donc celle-ci :



### **3. L'échange de données entre le client et le serveur (sendto et recvfrom)**

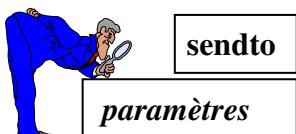
Le transfert de datagrammes peut à présent être réalisé sans autres préparatifs au moyen des deux primitives suivantes.

#### **3.1 L'émission de caractères (sendto)**

C'est le rôle de la primitive

```
int sendto    (<handle de la socket – int>,
              <adresse de la suite de caractères à envoyer - char *>,
              <nombre de caractères à envoyer – int>,
              <flag d'urgence - int>,
              <adresse et port du destinataire – struct sockaddr*>,
              <taille de la structure adresse – int>);
```

La valeur renvoyée est le nombre de caractères écrits sur la socket ou –1 en cas d'erreur.

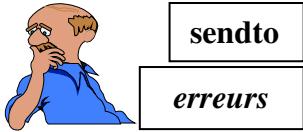


Les trois premiers paramètres sont similaires à ceux de la primitive **send()** de TCP. Le quatrième paramètre vaudra toujours 0 pour des communications normales. Ce n'est que dans le cas où l'on souhaite ne pas utiliser les tables de routage durant la transmission ce paramètre prendra la valeur

```
#define      MSG_DONTROUTE0x4 /* send without using routing tables */.
```

A remarquer qu'il n'est pas question d'utiliser ici **MSG\_OOB** des caractères urgents (voir chapitres suivants).

Le 5<sup>ème</sup> paramètre (adresse et port du destinataire) montre bien que le protocole est **non connecté** : une même socket peut en effet servir pour envoyer des données à des destinataires différents !



Si la valeur de retour est  $-1$ , une erreur s'est produite et la variable globale `errno` est positionnée sur l'une des valeurs suivantes :

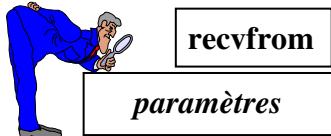
<i>valeur de errno</i>	<i>erreur</i>
<code>#define EBADF 9</code>	<code>/* Bad file number */</code> : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
<code>#define ENOTSOCK 38</code>	<code>/* Socket operation on non-socket */</code> : le descripteur n'est pas associé à une socket, mais à un fichier
<code>#define EFAULT 14</code>	<code>/* Bad address */</code> : l'adresse de la suite de caractères est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)
<code>#define EDESTADDRREQ 39</code>	<code>/* Destination address required */</code> : l'adresse de destination n'est pas spécifiée
<code>#define EMSGSIZE 40</code>	<code>/* Message too long */</code> : très clair ...
<code>#define EWOULDBLOCK 35</code>	<code>/* Operation would block */</code> : il n'y a rien à envoyer et la socket n'est pas bloquante (ce qu'elle devrait être)
<code>#define EAGAIN EWOULDBLOCK</code>	
<code>#define EOPNOTSUPP 45</code>	<code>/* Operation not supported on socket */</code> : <code>MSG_OOB</code> n'est pas utilisable
<code>#define EINTR 4</code>	<code>/* Interrupted system call */</code> : la fonction a été interrompue

### 3.2 La réception de caractères (`recvfrom`)

C'est le rôle de la primitive de réaliser une **lecture destructive ou pas** des caractères reçus :

```
int recvfrom (<handle de la socket - int>,
              <adresse où écrire la suite de caractères lus - char *>,
              <nombre maximum de caractères à lire - int>,
              <flag de non destruction - int>,
              <adresse et port de l'émetteur - struct sockaddr*>,
              <taille de la structure adresse - int *>);
```

La valeur renvoyée est le nombre de caractères lus ou  $-1$  en cas d'erreur. Comme on en a pris l'habitude, `recvfrom` est *bloquant* sur un buffer de lecture vide (sauf si l'on a paramétré la socket pour qu'il n'en soit pas ainsi – voir le chapitre parlant de cette paramétrisation). Il faut encore remarquer qu'une valeur nulle comme valeur de retour ne signifie pas, et pour cause, que l'autre extrémité a fermé la connexion (alors qu'il en est ainsi en TCP) : l'envoi d'un datagramme dont les data sont de longueur nulle est donc possible (on ne reçoit donc que les headers).



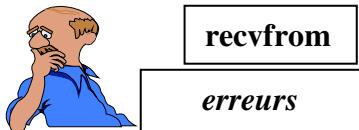
Le quatrième paramètre permet de spécifier une éventuelle *lecture non destructive* : il suffit de lui donner la valeur (définie dans sys/socket.h) :

```
#define MSG_PEEK 0x2           /* peek at incoming message */
```

Le message reçu reste alors dans le buffer de lecture. A remarquer que l'adresse émettrice n'est alors pas recopiée.

A nouveau, le 5<sup>ème</sup> paramètre (adresse et port de l'expéditeur) montre bien que le protocole est **non connecté** : une même socket peut en effet servir pour recevoir des données de provenances différentes.

Le dernier paramètre désigne l'emplacement la valeur de la longueur de la zone adresse. Au retour, elle contiendra la taille effective.



Si la valeur de retour est -1, une erreur s'est produite et la variable globale errno est positionnée sur l'une des valeurs suivantes :

valeur de errno	erreur
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define EFAULT 14	/* Bad address */ : l'adresse où placer la suite de caractères est incorrecte (probablement hors de l'espace d'adressage en lecture du processus)
#define EINVAL 22	/* Invalid argument */ : le nombre de caractères à lire est négatif.
#define EWOULDBLOCK 35	/* Operation would block */ : il n'y a rien à lire et la socket n'est pas bloquante (ce qu'elle devrait être)
#define EINTR 4	/* Interrupted system call */ : la fonction a été interrompue

## 4. Une première communication UDP

Commençons par une simple communication unidirectionnelle : un client envoie un message à un serveur en attente sur le port 5000 de boole. Le serveur s'écrit très simplement :

### UDPITER01.C

```
/* UDPITER01.C
- Claude Vilvens -
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format
                        reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#define PORT 5000 /* Port de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost;
    struct in_addr adresseIPServer, adresseIClient;
    struct sockaddr_in adresseSocketServeur, adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServer[MAXSTRING];
    int nbreRecv;

    /* 1. Creation de la socket */
    hSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");
}
```

```

/* 2. Acquisition des informations sur l'ordinateur local */
if ( (infosHost = gethostbyname("boole"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    exit(1);
}
else printf("Acquisition infos host OK\n");
memcpy(&adresseIPServeur, infosHost->h_addr, infosHost->h_length);
printf("Adresse IP = %s\n",inet_ntoa(adresseIPServeur));

/* 3. Preparation de la structure sockaddr_in */
tailleSockaddr_in = sizeof(struct sockaddr_in);
memset(&adresseSocketServeur, 0, tailleSockaddr_in);
adresseSocketServeur.sin_family = AF_INET;
adresseSocketServeur.sin_port = htons(PORT);
memcpy(&adresseSocketServeur.sin_addr, infosHost->h_addr,infosHost->h_length);

/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
if (bind(hSocket, (struct sockaddr *)&adresseSocketServeur,
        sizeof(struct sockaddr_in)) == -1)
{
    printf("Erreur sur le bind de la socket %d\n", errno);
    exit(1);
}
else printf("Bind adresse et port socket OK\n");

/* 5.Reception d'un message client */
if ((nbreRecv = recvfrom(hSocket,
                         msgClient, MAXSTRING,
                         0,      /* lecture destructive */
                         &adresseSocketClient,
                         &tailleSockaddr_in)
      ) == -1)
{
    printf("Erreur sur le recvfrom de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recvfrom socket OK\n");
msgClient[nbreRecv+1]=0;
printf("Message recu par le serveur = %s\n", msgClient);
printf("Adresse de l'emetteur = %u\n", adresseSocketClient.sin_addr.s_addr);
adresseIPClient = adresseSocketClient.sin_addr;
printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPClient));

/* 6. Fermeture des sockets */
close(hSocket); printf("Socket serveur fermee\n"); return 0;
}

```

Le client est assez similaire. En fait, il peut émettre dès qu'il possède les informations nécessaires sur sa cible :

**UDPCLI01.C**

```
/* UDPCLI01.C
- Claude Vilvens -
*/

#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format
                        reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#define PORT_SERVEUR 5000 /* Port de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost, * infosOther;
    struct in_addr adresseIP, adresseIPOther;
    struct sockaddr_in adresseSocketServeur, adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;

/* 1. Creation de la socket */
    hSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");
}
```

```

/* 2. Acquisition des informations sur le serveur */
if ( (infosOther = gethostbyname("boole"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    exit(1);
}
else printf("Acquisition infos other OK\n");
memcpy(&adresseIPOther, infosOther->h_addr, infosOther->h_length);
printf("Adresse IP other = %s\n",inet_ntoa(adresseIPOther));

/* 3. Preparation de la structure sockaddr_in du serveur */
tailleSockaddr_in = sizeof(struct sockaddr_in);
memset(&adresseSocketServeur, 0, tailleSockaddr_in);
adresseSocketServeur.sin_family = AF_INET;
adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr,infosOther->h_length);

/* 4. Envoi d'un message au serveur */
printf("Message a envoyer au serveur : ");
gets(msgClient);
if (sendto(hSocket,
        msgClient, MAXSTRING,
        0,
        &adresseSocketServeur,
        tailleSockaddr_in)
    == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");

/* 5. Fermeture des sockets */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");

return 0;
}

```

Si le client (sur dec01) agit de la manière suivante :

```

% cli
Creation de la socket OK
Acquisition infos other OK
Adresse IP other = 10.59.4.1
Message a envoyer au serveur : Ennemi en vue !
Send socket OK
Socket client fermee
%

```

le serveur (sur boole) réagira de la manière suivante :

```
boole>s
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Recvfrom socket OK
Message recu par le serveur = Ennemi en vue !
Adresse de l'emetteur = 2198092554
Adresse de l'emetteur = 10.59.4.131
Socket serveur fermee
```

Et le port ? Si l'on ajoute au serveur :

```
printf("Port de l'emetteur = %d\n", ntohs(adresseSocketClient.sin_port));
```

on obtient en plus (par exemple) :

```
| Port de l'emetteur = 2574
```

c'est-à-dire le port attribué par défaut au client.

Evidemment, il faudrait que le serveur réponde quelques chose au client ...

## **5. Un dialogue client-serveur**

### **5.1 Un dialogue requête-réponse**

Il suffit évidemment d'ajouter les instructions permettant la communication dans l'autre sens. Le serveur utilise pour son sendto() la structure adresseSocketClient complaisamment initialisée par le recvfrom() :

#### **UDPITER01.C (2)**

```
/* UDPITER01.C
- Claude Vilvens -
*/
#include <stdio.h>
...
#define PORT 5000 /* Port de la socket serveur */
#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost;
    struct in_addr adresseIPServeur, adresseIPClient;
    struct sockaddr_in adresseSocketServeur, adresseSocketClient;
    unsigned int tailleSockaddr_in;
    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;
/* 1. Creation de la socket */
```

```

...
/* 2. Acquisition des informations sur l'ordinateur local */

...
/* 3. Preparation de la structure sockaddr_in */

...
/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */

...
/* 5. Reception d'un message client */
if ((nbreRecv = recvfrom(hSocket,
                      msgClient, MAXSTRING,
                      0,      /* lecture destructive */
&adresseSocketClient,
&tailleSockaddr_in)
    ) == -1)
{
    printf("Erreur sur le recvfrom de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recvfrom socket OK\n");
msgClient[nbreRecv+1]=0;
printf("Message recu par le serveur = %s\n", msgClient);
printf("Adresse de l'emetteur = %u\n", adresseSocketClient.sin_addr.s_addr);
adresseIPClient = adresseSocketClient.sin_addr;
printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPClient));
printf("Port de l'emetteur = %d\n", ntohs(adresseSocketClient.sin_port));

/* 6. Reponse du serveur au client */
memset(msgServeur, 0, MAXSTRING);
sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
printf("Message de reponse : %s\n", msgServeur);
if (sendto(hSocket,
          msgServeur, MAXSTRING,
          0,
&adresseSocketClient,
          tailleSockaddr_in
    ) == -1)
{
    printf("Erreur sur le sendto de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Sendto socket OK\n");

/* 7. Fermeture de la socket */
...
}

```

Le client va se doter d'un port fixe (ici, 6000). Il devra donc disposer d'une structure `sockaddr_in` dans `adresseSocketClient` pour effectuer son `bind()` et d'une structure `sockaddr_in` dans `adresseSocketServeur` pour réaliser son `sendto()` :

### UDPCLI01.C (2)

```
/* UDPCLI01.C
- Claude Vilvens -
*/
#include <stdio.h>
...
#define PORT_SERVEUR 5000 /* Port de la socket serveur */
#define PORT_CLIENT 6000 /* Port de la socket client */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent *infosHost, *infosOther;
    struct in_addr adresseIP, adresseIPOther;
    struct sockaddr_in adresseSocketServeur;
    struct sockaddr_in adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;

/* 1. Creation de la socket */
    ...

/* 2. Acquisition des informations sur l'ordinateur local */
    if ((infosHost = gethostbyname("dec01"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos host OK\n");
    memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
    printf("Adresse IP host = %s\n", inet_ntoa(adresseIP));

/* 3. Preparation de la structure sockaddr_in du client */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    memset(&adresseSocketClient, 0, tailleSockaddr_in);
    adresseSocketClient.sin_family = AF_INET;
    adresseSocketClient.sin_port = htons(PORT_CLIENT);
    memcpy(&adresseSocketClient.sin_addr, infosHost->h_addr, infosHost->h_length);
```

```

/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
if (bind(hSocket, (struct sockaddr *)&adresseSocketClient,
tailleSockaddr_in) == -1)
{
    printf("Erreur sur le bind de la socket %d\n", errno);
    exit(1);
}
else printf("Bind adresse et port socket OK\n");

/* 5. Acquisition des informations sur l'autre ordinateur */
if ( (infosOther = gethostbyname("boole"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    exit(1);
}
else printf("Acquisition infos other OK\n");
memcpy(&adresseIPOther, infosOther->h_addr, infosOther->h_length);
printf("Adresse IP other = %s\n",inet_ntoa(adresseIPOther));

/* 6. Preparation de la structure sockaddr_in du serveur */
memset(&adresseSocketServeur, 0, tailleSockaddr_in);
adresseSocketServeur.sin_family = AF_INET;
adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr,infosOther->h_length);

/* 7. Envoi d'un message au serveur */
printf("Message a envoyer au serveur : ");
gets(msgClient);

if (sendto(hSocket,
            msgClient, MAXSTRING,
            0,
            &adresseSocketServeur,
            tailleSockaddr_in)
    == -1)
{
    printf("Erreur sur le sendto de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Sendto socket OK\n");

/* 8.Reception d'un message serveur */
memset(msgServeur, 0, MAXSTRING);
if ((nbreRecv = recvfrom(hSocket,
                         msgServeur, MAXSTRING,
                         0,      /* lecture destructive */
                         &adresseSocketServeur,
                         &tailleSockaddr_in)
    ) == -1)

```

```
{
    printf("Erreur sur le recvfrom de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recvfrom socket OK\n");
msgClient[nbreRecv+1]=0;
printf("Message envoyé par le serveur = %s\n", msgServeur);

printf("Adresse de l'émetteur = %u\n", adresseSocketServeur.sin_addr.s_addr);
adresseIPOther = adresseSocketServeur.sin_addr;
printf("Adresse de l'émetteur = %s\n", inet_ntoa(adresseIPOther));
printf("Port de l'émetteur = %d\n", ntohs(adresseSocketServeur.sin_port));

/* 9. Fermeture de la socket */
...
}
```

Pour cet exemple, le serveur reçoit un message :

```
boole>s
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Recvfrom socket OK
Message recu par le serveur = Alea jacta est
Adresse de l'émetteur = 2198092554
Adresse de l'émetteur = 10.59.4.131
Port de l'émetteur = 6000
Message de reponse : ACK pour votre message : <Alea jacta est>
Sendto socket OK
Socket serveur fermee
boole>
```

et envoie donc un ack au client :

```
% cli
Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.4.131
Bind adresse et port socket OK
Acquisition infos other OK
Adresse IP other = 10.59.4.1
Message à envoyer au serveur : Alea jacta est
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <Alea jacta est>
Adresse de l'émetteur = 17054474
Adresse de l'émetteur = 10.59.4.1
```

Port de l'emetteur = 5000  
Socket client fermee  
%

### **Remarque**

Si le client ne réalise pas un bind() pour fixer son numéro de port, le système lui attribuera un numéro qu'il conservera tout au long de la communication. Nous allons procéder ainsi, pour l'exemple, dans le cas de plusieurs clients. En pratique, la question est de savoir s'il est important qu'un client possède un port fixe ou pas.

### **5.2 Un dialogue clients-serveur continu**

Nous allons ici simplement modifier le client et le serveur pour permettre un dialogue continu. Il suffit pour cela d'enfermer les instructions sento() et recvfrom() dans une boucle qui se terminera sur un message SHUTDOWN!. Pour le serveur, la modification est minime :

#### **UDPITER02.C**

```
/* UDPITER02.C
- Claude Vilvens -
Maj: 17/8/99
*/
#include <stdio.h>
...
#define PORT 5000 /* Port de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost;
    struct in_addr adresseIPServeur, adresseIPClient;
    struct sockaddr_in adresseSocketServeur, adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;

/* 1. Creation de la socket */
.....
/* 2. Acquisition des informations sur l'ordinateur local */
.....
/* 3. Preparation de la structure sockaddr_in */
.....
/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
.....
```

```

do
{
/* 5.Reception d'un message client */
    if ((nbreRecv = recvfrom(hSocket, msgClient, MAXSTRING, 0,
        &adresseSocketClient, &tailleSockaddr_in) ) == -1)
    {
        printf("Erreur sur le recvfrom de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Recvfrom socket OK\n");
    msgClient[nbreRecv+1]=0;
    printf("Message recu par le serveur = %s\n", msgClient);
    printf("Adresse de l'emetteur = %u\n", adresseSocketClient.sin_addr.s_addr);
    adresseIPClient = adresseSocketClient.sin_addr;
    printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPClient));
    printf("Port de l'emetteur = %d\n", ntohs(adresseSocketClient.sin_port));

/* 6. Reponse du serveur au client */
    memset(msgServeur, 0, MAXSTRING);
    sprintf(msgServeur,"ACK pour votre message : <%s>", msgClient);
    printf("Message de reponse : %s\n", msgServeur);
    if (sendto(hSocket, msgServeur, MAXSTRING, 0,
        &adresseSocketClient, tailleSockaddr_in) == -1)
    {
        printf("Erreur sur le sendto de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Sendto socket OK\n");
}
while (strcmp(msgClient,"SHUTDOWN!"));

/* 7. Fermeture de la socket */
...
}

```

Le client n'est gère différent de sa version précédente, mais il n'effectue pas de bind et laisse donc le système choisir un port pour lui :

### **UDPCLI02.C**

```

/* UDPCLI02.C
- Claude Vilvens -
*/
#include <stdio.h>
...
#define PORT_SERVEUR 5000 /* Port de la socket serveur */

```

```
#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost, * infosOther;
    struct in_addr adresseIP, adresseIPOther;
    struct sockaddr_in adresseSocketServeur;
    struct sockaddr_in adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv;

/* 1. Creation de la socket */
...
/* 2. Acquisition des informations sur l'ordinateur local */
....
/* 3. Preparation de la structure sockaddr_in du client */
    tailleSockaddr_in = sizeof(struct sockaddr_in);
    memset(&adresseSocketClient, 0, tailleSockaddr_in);
    adresseSocketClient.sin_family = AF_INET;
    /* Pas de port fixe ... */
    memcpy(&adresseSocketClient.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Acquisition des informations sur l'autre ordinateur */
...
/* 5. Preparation de la structure sockaddr_in du serveur */
    memset(&adresseSocketServeur, 0, tailleSockaddr_in);
    adresseSocketServeur.sin_family = AF_INET;
    adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
    memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr, infosOther->h_length);

do
{
/* 6. Envoi d'un message au serveur */
    printf("Message a envoyer au serveur : ");
    gets(msgClient);
    if (sendto(hSocket, msgClient, MAXSTRING, 0,
                &adresseSocketServeur, tailleSockaddr_in) == -1)
    {
        printf("Erreur sur le sendto de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Sendto socket OK\n");

/* 7. Reception d'un message serveur */
    memset(msgServeur, 0, MAXSTRING);
    if ((nbreRecv = recvfrom(hSocket, msgServeur, MAXSTRING, 0,
                            &adresseSocketServeur,&tailleSockaddr_in) ) == -1)
```

```

{
    printf("Erreur sur le recvfrom de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Recvfrom socket OK\n");
msgClient[nbreRecv+1]=0;
printf("Message envoyé par le serveur = %s\n", msgServeur);
printf("Adresse de l'emetteur = %u\n", adresseSocketServeur.sin_addr.s_addr);
adresseIPOther = adresseSocketServeur.sin_addr;
printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPOther));
printf("Port de l'emetteur = %d\n", ntohs(adresseSocketServeur.sin_port));
}
while (strcmp(msgClient, "SHUTDOWN!") != 0);

/* 8. Fermeture de la socket */
...
}

```

La construction de la structure adresseSocketClient dans le serveur est indispensable – ce n'est que le numéro de port qui est pris par défaut !

Dans notre exemple d'exécution, un premier client envoie des requêtes au serveur :

```

% cli
Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.4.131
Acquisition infos other OK
Adresse IP other = 10.59.4.1
Message a envoyer au serveur : Coucou !
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <Coucou !>
Adresse de l'emetteur = 17054474
Adresse de l'emetteur = 10.59.4.1
Port de l'emetteur = 5000
Message a envoyer au serveur : C'est moi !!!
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <C'est moi !!!>
Adresse de l'emetteur = 17054474
Adresse de l'emetteur = 10.59.4.1
Port de l'emetteur = 5000
Message a envoyer au serveur : Heureuse de me voir sans doute ?
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <Heureuse de me voir s>
Adresse de l'emetteur = 17054474
Adresse de l'emetteur = 10.59.4.1
Port de l'emetteur = 5000

```

Le serveur répond gentiment:

```
boole>s
Creation de la socket OK
Acquisition infos host OK
Adresse IP = 10.59.4.1
Bind adresse et port socket OK
Recvfrom socket OK
Message recu par le serveur = Coucou !
Adresse de l'emetteur = 2198092554
Adresse de l'emetteur = 10.59.4.131
Port de l'emetteur = 2595
Message de reponse : ACK pour votre message : <Coucou !>
Sendto socket OK
Recvfrom socket OK
Message recu par le serveur = C'est moi !!!
Adresse de l'emetteur = 2198092554
Adresse de l'emetteur = 10.59.4.131
Port de l'emetteur = 2595
Message de reponse : ACK pour votre message : <C'est moi !!!>
Sendto socket OK
Recvfrom socket OK
Message recu par le serveur = Heureuse de me voir sans doute ?
Adresse de l'emetteur = 2198092554
Adresse de l'emetteur = 10.59.4.131
Port de l'emetteur = 2595
Message de reponse : ACK pour votre message : <Heureuse de me voir sans doute ?>
Sendto socket OK
```

Mais un deuxième client intervient :

```
% cli
Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.4.131
Acquisition infos other OK
Adresse IP other = 10.59.4.1
Message a envoyer au serveur : Ici boy2
Sendto socket OK
Recvfrom socket OK
Message envoyee par le serveur = ACK pour votre message : <Ici boy2>
Adresse de l'emetteur = 17054474
Adresse de l'emetteur = 10.59.4.1
Port de l'emetteur = 5000
```

Le serveur lui a répondu sur le port qui lui a été attribué :

```
Recvfrom socket OK
Message recu par le serveur = Ici boy2
Adresse de l'emetteur = 2198092554
```

---

Adresse de l'emetteur = 10.59.4.131

Port de l'emetteur = **2596**

Message de reponse : ACK pour votre message : <Ici boy2>

Sendto socket OK

Chaque client continue alors sa conversation. Ainsi, le deuxième :

Message a envoyer au serveur : *Je suis le trouble-fete ...*

Sendto socket OK

Recvfrom socket OK

Message envoyee par le serveur = ACK pour votre message : <Je suis le trouble-fe>

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = **1278**

Message a envoyer au serveur : *Je suis mieux que le premier client !*

Sendto socket OK

Recvfrom socket OK

Message envoyee par le serveur = ACK pour votre message : <Je suis mieux que le >

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = **1278**

Message a envoyer au serveur : *SHUTDOWN!*

Sendto socket OK

Recvfrom socket OK

Message envoyee par le serveur = ACK pour votre message : <SHUTDOWN!>

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = **1278**

Socket client fermee

%

et le premier :

Message a envoyer au serveur : *Je t'ai manqué ?*

Sendto socket OK

Recvfrom socket OK

Message envoyee par le serveur = ACK pour votre message : <Je t'ai manqué ?>

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = **1276**

Et le serveur sera, très naïvement d'ailleurs, arrêté par le deuxième client :

Recvfrom socket OK

Message recu par le serveur = *Je suis le trouble-fete ...*

Adresse de l'emetteur = 2198092554

Adresse de l'emetteur = 10.59.4.131

Port de l'emetteur = **2596**

Message de reponse : ACK pour votre message : <Je suis le trouble-fete ...>

Sendto socket OK

Recvfrom socket OK

Message recu par le serveur = *Je suis mieux que le premier client !*

Adresse de l'emetteur = 2198092554

Adresse de l'emetteur = 10.59.4.131

Port de l'emetteur = **2596**

Message de reponse : ACK pour votre message : <Je suis mieux que le premier client !>

Sendto socket OK

Recvfrom socket OK

Message recu par le serveur = *SHUTDOWN!*

Adresse de l'emetteur = 2198092554

Adresse de l'emetteur = 10.59.4.131

Port de l'emetteur = **2596**

Message de reponse : ACK pour votre message : <SHUTDOWN!>

Sendto socket OK

Socket serveur fermee

boole>

si bien que lorsque le premier voudra faire la même chose, il arrivera trop tard et restera bloqué sur son sendto :

Message a envoyer au serveur : *Shutdown Oh oh ...*

Sendto socket OK

## 6. Les problèmes d'UDP

L'exemple ci-dessus nous inspire quelques remarques :

- ◆ si un serveur a entamé un dialogue avec un client, un autre client connaissant le port du serveur peut parfaitement s'immiscer dans la conversation; ***la cohérence ne pourra être maintenue que si le serveur vérifie***, à chaque réception, que l'adresse IP du client est bien celle du client primitif;
- ◆ comme UDP n'est pas fiable, ***il se peut très bien que le serveur soit arrêté sans que le client puisse s'en rendre compte***; en effet, le message d'erreur ICMP n'est pas envoyé au client; celui-ci restera alors bloqué sur un recvfrom(); une telle erreur est encore appelé une "*erreur asynchrone*";
- ◆ toujours parce qu'UDP n'est pas fiable, ***un datagramme peut être perdu***, provoquant un blocage sur la fonction recvfrom();

Il serait donc sage d'accompagner l'appel de recvfrom() d'un time-out; ceci peut se faire par la paramétrisation ad hoc de la socket (voir chapitre consacré à ce sujet) ou encore en utilisant le signal SIGALRM. On utilisera pour ce faire la fonction

unsigned int **alarm**(unsigned int seconds)

qui permet, pour rappel, de *demande au système une notification de time-out* après un intervalle de temps correspondant au nombre de secondes spécifié par le paramètre (une valeur nulle pour celui-ci désactive le timer). Le processus se voit ensuite délivré le signal

SIGALRM. La valeur de retour vaut normalement 0, à moins qu'un autre timer ait déjà été enclenché, auquel cas la valeur de retour vaut le nombre de secondes à attendre pour voir arriver la fin de cet autre timer.

Nous pouvons de cette manière, du côté du client, soumettre l'attente d'une réception d'ACK à un time-out :

### UDPCLI03.C

```
/* UDPCLI03.C
- Claude Vilvens -
*/
...
#include <signal.h>
#include <unistd.h>

#define PORT_SERVEUR 5000 /* Port de la socket serveur */
#define MAXSTRING 100 /* Longueur des messages */

void handlerSignalAlarme(int sig);
struct sigaction sigAct;

int main()
{
    ...
    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv, cpt=0;

/* 1. Creation de la socket */
    ....
/* 2. Acquisition des informations sur l'ordinateur local */
    ....
/* 3. Preparation de la structure sockaddr_in du client */
    ....
/* 4. Acquisition des informations sur l'autre ordinateur */
    if ( (infosOther = gethostbyname("boole"))==0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        exit(1);
    }
    else printf("Acquisition infos other OK\n");
    memcpy(&adresseIPOther, infosOther->h_addr, infosOther->h_length);
    printf("Adresse IP other = %s\n",inet_ntoa(adresseIPOther));

/* 5. Preparation de la structure sockaddr_in du serveur */
    memset(&adresseSocketServeur, 0, tailleSockaddr_in);
    adresseSocketServeur.sin_family = AF_INET;
    adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
    memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr,
           infosOther->h_length);
```

```

sigAct.sa_handler = handlerSignalAlarme;
sigaction(SIGALRM, &sigAct, NULL);

do
{
/* 6. Envoi d'un message au serveur */
    printf("Message a envoyer au serveur : ");gets(msgClient);
    if (sendto(hSocket, msgClient, MAXSTRING,
               0, &adresseSocketServeur, tailleSockaddr_in)
        == -1)
    {
        printf("Erreur sur le sendto de la socket %d\n", errno);
        close(hSocket); exit(1);
    }
    else
    {
        printf("Sendto socket OK\n");
    }

/* 7.Reception d'un message serveur */
    memset(msgServeur, 0, MAXSTRING);
    alarm(5);
    if ((nbreRecv = recvfrom(hSocket, msgServeur, MAXSTRING,
                            0, /* lecture destructive */
                            &adresseSocketServeur, &tailleSockaddr_in)
        ) == -1)
    {
        if (errno == EINTR)
        {
            cpt++;
            puts("Time-out sur le recvfrom !");
            if (cpt<3) continue;
            else
            {
                puts("Le serveur semble etre mort ... ");
                break;
            }
        }
        else
        {
            printf("Erreur sur le recvfrom de la socket %d\n", errno);
            close(hSocket); exit(1);
        }
    }
    else
    {
        if (cpt>0) cpt=0;
        printf("Recvfrom socket OK\n");
        alarm(0);
    }
}

```

```

msgClient[nbreRecv+1]=0;

printf("Message envoyé par le serveur = %s\n", msgServeur);
printf("Adresse de l'émetteur = %u\n", adresseSocketServeur.sin_addr.s_addr);
adresseIPOther = adresseSocketServeur.sin_addr;
printf("Adresse de l'émetteur = %s\n", inet_ntoa(adresseIPOther));
printf("Port de l'émetteur = %u\n", ntohs(adresseSocketServeur.sin_port));
}

while (strcmp(msgClient, "SHUTDOWN!"));

/* 9. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermée\n");

return 0;
}

void handlerSignalAlarme(int sig)
{
    puts("Pas de réponse du serveur ...");
    return; /* Pourrait suffir : juste pour interrompre le recvfrom */
}

```

Avec le même serveur que précédemment, un exemple d'exécution du client est :

```

Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.4.131
Acquisition infos other OK
Adresse IP other = 10.59.4.1
Message à envoyer au serveur : Bonjour, c'est LULU!
Sendto socket OK
Pas de réponse du serveur ...
Time-out sur le recvfrom !
Message à envoyer au serveur : Bonjour, c'est LULU !
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <Bonjour, c'est LULU !>
Adresse de l'émetteur = 17054474
Adresse de l'émetteur = 10.59.4.1
Port de l'émetteur = 5000
Message à envoyer au serveur : Je suis prête ...
Sendto socket OK
Recvfrom socket OK
Message envoyé par le serveur = ACK pour votre message : <Je suis prête ...>
Adresse de l'émetteur = 17054474
Adresse de l'émetteur = 10.59.4.1
Port de l'émetteur = 5000
Message à envoyer au serveur : Tu m'écoutes ?
Sendto socket OK

```

Le serveur a été arrêté ! Le client va-t-il en prendre conscience ?

**Pas de reponse du serveur ...**

Time-out sur le recvfrom !

Message a envoyer au serveur : Dis, tu vas ma repondre ?

Sendto socket OK

**Pas de reponse du serveur ...**

Time-out sur le recvfrom !

Message a envoyer au serveur : Mais enfin ???

Sendto socket OK

**Pas de reponse du serveur ...**

Time-out sur le recvfrom !

***Le serveur semble etre mort ...***

Socket client fermee

**Remarque**

Les erreurs asynchrones peuvent être prises en compte en utilisant la fonction connect() sur la socket UDP; cet appel n'a pas le même effet qu'avec une socket TCP, puisqu'il se limite à l'enregistrement de l'adresse IP et du port. On utilise alors les primitives send() et recv().

## 7. Le multicast

Toutes les adresses IP que nous avons utilisées jusqu'à présent sont des adresses désignant une seule machine (plus précisément un seul interface). Cependant, il existe des situations où l'on souhaite envoyer un message à plusieurs destinataires. Bien sûr, on peut envoyer successivement le message à chacun d'entre eux, mais on préférerait évidemment le faire en une seule opération. C'est possible en utilisant une adresse multicast de **classe D** et le protocole de transport UDP.

### 7.1 Les adresses multicast

Pour rappel, une adresse de classe D a la structure suivante:

D	1	1	1	0	multicast group id = 28 bits
---	---	---	---	---	------------------------------

si bien que l'espace d'adresses utilisables est [224.0.0.0 ; 239.255.255.255]. Une telle adresse n'identifie pas un seul interface, mais un ensemble d'interfaces qui, au travers de l'application qu'il exécutent, ont manifesté leur souhait de recevoir les données adressées à cette adresse : on dit encore que ces interfaces participent à la **session multicast** ou qu'ils ont rejoint le **groupe multicast**. Lorsque l'on envoie un message à une telle *adresse de groupe multicast*, ce sont donc tous les membres du groupe qui le recoivent. Bien sûr, le protocole de transport doit donc être UDP (et non pas TCP), puisque seules les machines intéressées (dont l'adresse unicast est inconnue) accepteront le message (c'est l'aspect "*resource discovery*" du multicast et d'ailleurs aussi du broadcast).

En pratique, certaines adresses sont affectées à un usage particulier:

- ◆ 224.0.0.1 : adresse du groupe "*all-hosts*", c'est-à-dire du groupe que toutes les machines hôtes capables de recevoir un message multicast doivent rejoindre;
- ◆ 224.0.0.2 : adresse du groupe "*all-routers*", c'est-à-dire du groupe que tous les routeurs capables de recevoir un message multicast doivent rejoindre;

- ◆ 224.0.0.0 -> 224.0.0.255 (donc 224.0.0.0/24) : groupe des adresses "*link local*" pour lesquelles les datagrammes ne sont jamais renvoyés par un routeur multicast – elles permettent donc un multicast au plus bas niveau topologique du réseau;
- ◆ 239.255.255.255 : comme d'habitude, elle désigne toutes les machines du réseau.

IPv6 a introduit en plus la notion de portée pour une adresse : il s'agit d'un champ de 4 bits qui permet de spécifier dans quelles limites un datagramme multicast doit être expédié. IPv4 ne possède pas la notion de portée, mais

- ◆ la simule en utilisant la valeur du TTL des paquets IP associés : intuitivement, plus le TTL est grand, et plus le paquet "peut aller loin";
- ◆ définit des plages d'adresses correspondant à la portée prévue.

Les valeur correspondantes les plus courantes sont :

portée	portée IPv6	IPv4	
		TTL	plage d'adresses
machine [ <i>node-local</i> ]	1	0	
sous-réseau proche [ <i>link-local</i> ]	2	1	224.0.0.0 → 224.0.0.255
sous-réseau local [ <i>subnet-local</i> ]	3	2	
administrateur local [ <i>admin-local</i> ]	4	3	
site [ <i>site-local</i> ]	5	<32	239.255.0.0 → 239.255.255.255
entreprise [ <i>organization-local</i> ]	8	<128	239.192.0.0 → 239.195.25.255
global	14	<255	224.0.1.0 → 238.255.255.255

## 7.2 Les options d'une socket multicast

Pour rejoindre [*join*] un groupe multicast (et pour le quitter par après), une socket doit être paramétrée. Nous reparlerons du paramétrage des sockets dans le chapitre suivant. Qu'il nous suffise pour l'instant de savoir que cela peut se faire au moyen de la fonction **setsockopt()**, cousine de la fonction **getsockopt()** déjà rencontrée au chapitre II. Cette fonction utilise les constantes et types de données suivants (définis dans netinet/in.h) :

constantes pour setsockopt()	types de données utilisés	action	
IP_ADD_MEMBERSHIP (12)	struct ip_mreq	pour rejoindre un groupe multicast	R
IP_DROP_MEMBERSHIP (13)	struct ip_mreq	pour quitter un groupe multicast	E C V
IP_MULTICAST_TTL (10)	u_char	pour spécifier le TTL d'un message multicast sortant – à placer sur une valeur supérieure à 1 pour pouvoir sortir de son sous-réseau	S E
IP_MULTICAST_IF (9)	struct in_addr	pour spécifier l'adresse à utiliser pour les messages multicasts sortants	N D
IP_MULTICAST_LOOP (11)	u_char	pour spécifier si un message multicast sortant doit être aussi envoyé à la machine émettrice elle-même (loopback)	

Les 3 dernières options concernent l'envoi de datagrammes multicast; les valeurs correspondantes peuvent être prises par défaut (TTL de 1, adresse de l'hôte et loop-back activé), si bien que *l'on peut envoyer un message multicast dans son sous-réseau sans aucune précaution particulière.*

Les 2 premières options concernent la réception de datagrammes multicast; la structure

```
struct ip_mreq
{
    struct in_addr imr_multiaddr;      /* IP multicast address of group */
    struct in_addr imr_interface;     /* local IP address of interface */
};
```

permet bien entendu de mémoriser respectivement l'adresse de classe D caractérisant le groupe et l'adresse (unicast) à utiliser effectivement sur la machine pour les communications réseaux. Il est possible d'adhérer à plusieurs groupes multicast distincts (c'est-à-dire d'adresses de classe D différentes). A la différence de l'envoi de messages multicast, *la réception de tels messages implique* donc :

- ◆ un **bind** de la socket locale au port utilisé pour le groupe multicast;
- ◆ l'**adhésion** au groupe multicast proprement dit.

En résumé, lorsqu'un datagramme multicast arrive sur une machine,

- ◆ la couche IP reconnaît une adresse multicast et vérifie si certaines applications ont joint ce groupe multicast;
- ◆ si oui, la couche UDP cherche une socket liée au port multicast spécifié;
- ◆ si il la trouve, le message y est délivré.

Comme le paramétrage des sockets en C sera vu plus tard, nous allons réaliser ces opérations avec des langages de plus haut niveau qui encapsuleront donc les détails techniques du niveau socket : Java et C#.

## 8. Le multicast en Java

### 8.1 La création d'un participant multicast

Le package `java.net` fournit une classe **MulticastSocket**, dérivée de la classe **DatagramSocket** dont on se sert en Java pour les communications basées sur UDP<sup>1</sup>. Le constructeur

```
public MulticastSocket (int port) throws IOException
```

réalise le bind sur le port spécifié (ici, ce sera par exemple 5001). Pour se joindre à un groupe multicast, on utilisera la méthode :

```
public void joinGroup(InetAddress mcastaddr) throws IOException
```

dont le paramètre permet de préciser l'adresse (de classe D) du groupe multicast considéré. Par conséquent, si on déclare :

---

<sup>1</sup> voir "Langage Java (II) : programmation avancée", chapitre XII (du même auteur ;-)

---

**MulticastSocket** *socketGroupe*;  
**InetAddress** *adresseGroupe*;

on peut alors adhérer valablement à un groupe multicast de chat d'adresse 234.5.5.9 en programmant :

```
try
{
    adresseGroupe = InetAddress.getByName("234.5.5.9");
    socketGroupe = new MulticastSocket(5001);
    socketGroupe.joinGroup(adresseGroupe);
}
catch (UnknownHostException e) {System.out.println("Erreur :-( : " + e.getMessage()); }
catch (IOException e) { System.out.println("Erreur :-( : " + e.getMessage()); }
```

Il est dès lors possible d'envoyer et de recevoir des datagrammes au moyen des méthodes *send()* et *receive()* qui utilise des objets instances de la classe **DatagramPacket**. Inutile de dire qu'il existe une méthode pour quitter le groupe :

public void **leaveGroup**(InetAddress mcastaddr) throws IOException

Précisons encore qu'il existe une méthode

public void **setTimeToLive** (int ttl) throws IOException

permettant de fixer le nombre de "hops" définissant la portée du datagramme multicast.

## 8.2 Un chat élémentaire

Supposons donc, pour illustrer nos propos, vouloir programmer un mini-chat de conversation par réseau. Chaque participant aura à sa disposition le modeste interface suivant :



Le constructeur de cette fenêtre ou encore la méthode réponse à l'appui sur le bouton "Démarrer" réalisera les opérations de bind et de join décrites ci-dessus. L'appui sur le bouton

"Envoyer" se contentera d'avoir pour effet de créer un objet **DatagramPacket** basé sur le texte entré et de l'envoyer à l'ensemble du groupe.

Reste la réception des messages. En effet, plusieurs participants peuvent simultanément envoyer des messages au groupe – il n'y a plus ici une simple communication requête-réponse. Comme ces messages peuvent donc arriver de manière imprévisible, le plus raisonnable est de créer un thread, instance de la classe **ThreadReception** que nous allons créer ici, qui sera responsable de leur réception et de leur affichage dans une boîte de liste. Tout naturellement, ce thread recevra donc, par son constructeur, le nom du client, la socket d'accès au groupe et la boîte de listes dans laquelle il placera les messages reçus. Finalement, on arrive ainsi aux deux composants Java suivants :

### **ClientChat.java**

```
/*
 * ClientChat.java
 */

import java.net.*;
import java.io.*;

public class ClientChat extends java.awt.Frame
{
    private String nomCli;
    private InetSocketAddress adresseGroupe;
    MulticastSocket socketGroupe;
    ThreadReception thr;

    public ClientChat()
    {
        initComponents();
        BDemarrer.setEnabled(true);BArreter.setEnabled(false);BEnvoyer.setEnabled(false);
    }

    private void initComponents()
    {
        BDemarrer = new java.awt.Button();
        ZTNom = new java.awt.TextField();
        BArreter = new java.awt.Button();
        ZTMessage = new java.awt.TextField();
        BEnvoyer = new java.awt.Button();
        LMsgRecus = new java.awt.List();

        ...
        setLayout(new java.awt.GridLayout(2, 1));
        setTitle("Client multicast");
        ...
        /* Instructions pour le GUI ... */
    }
}
```

```

private void BDemarrerActionPerformed(java.awt.event.ActionEvent evt)
{
    try
    {
        adresseGroupe = InetAddress.getByName("234.5.5.9");
        socketGroupe = new MulticastSocket(5001);
        socketGroupe.joinGroup(adresseGroupe);
        thr = new ThreadReception (nomCli, socketGroupe, LMsgRecus );
        thr.start();

        nomCli = ZTNom.getText();
        String msgDeb = nomCli + " rejoint le groupe";
        DatagramPacket dtg = new DatagramPacket(msgDeb.getBytes(), msgDeb.length(),
            adresseGroupe, 5001);
        socketGroupe.send(dtg);
    }
    catch (UnknownHostException e){ System.out.println("Erreur :-( :" + e.getMessage());}
    catch (IOException e){ System.out.println("Erreur :-( :" + e.getMessage()); }

    BDemarrer.setEnabled(false);BArreter.setEnabled(true);BEnvoyer.setEnabled(true);
}

private void BEnvoyerActionPerformed(java.awt.event.ActionEvent evt)
{
    String msg = nomCli + "> " + ZTMessage.getText();
    DatagramPacket dtg = new DatagramPacket(msg.getBytes(), msg.length(),
        adresseGroupe, 5001);
    try
    {
        socketGroupe.send(dtg);
    }
    catch (IOException e) { System.out.println("Erreur :-( :" + e.getMessage()); }
}

private void BArreterActionPerformed(java.awt.event.ActionEvent evt)
{
    String msg = nomCli + " quitte le groupe";
    DatagramPacket dtg = new DatagramPacket(msg.getBytes(), msg.length(),
        adresseGroupe, 5001);
    try
    {
        socketGroupe.send(dtg);
        thr.stop();
        BDemarrer.setEnabled(true);BArreter.setEnabled(false);BEnvoyer.setEnabled(false);
        socketGroupe.leaveGroup(adresseGroupe); System.out.println("Après leaveGroup");
        socketGroupe.close();
        System.out.println("Après close");
    }
    catch (IOException e){ System.out.println("Erreur :-( :" + e.getMessage()); }
}

```

```

public static void main(String args[])
{
    new ClientChat().show();
}

private java.awt.Button BDemarrer;
private java.awt.TextField ZTNom;
private java.awt.Button BArreter;
private java.awt.TextField ZTMessage;
private java.awt.Button BEnvoyer;
private java.awt.List LMsgRecus;
...
}

```

avec

### ThreadReception.java

```

/*
 * ThreadReception.java
 */

import java.net.*;
import java.awt.*;
import java.io.*;

public class ThreadReception extends Thread
{
    private String nom;
    private MulticastSocket socketGroupe;
    private List LMsgRecus;

    public ThreadReception (String n, MulticastSocket ms, List l)
    {
        nom = n; socketGroupe = ms; LMsgRecus = l;
    }

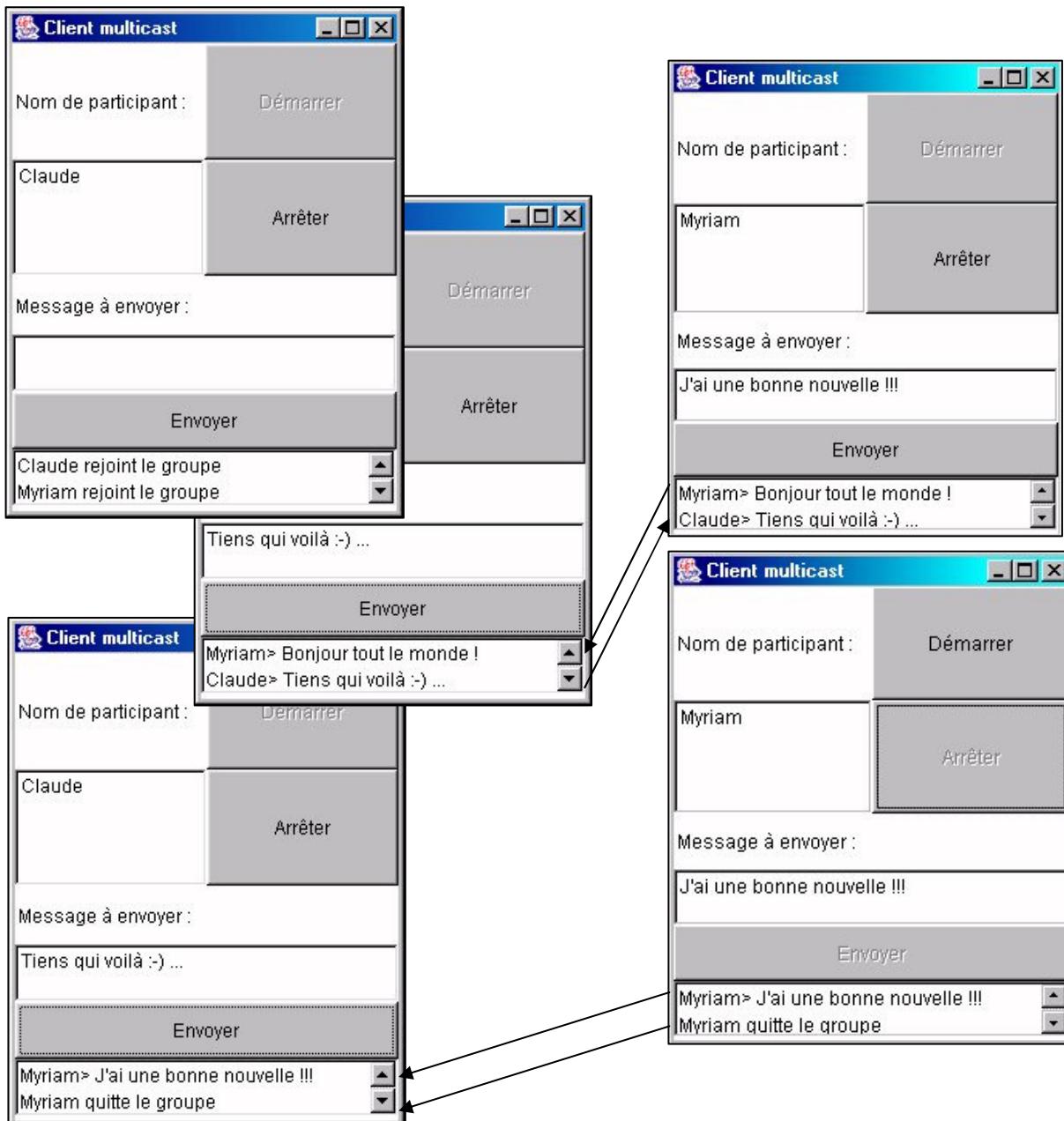
    public void run()
    {
        boolean enMarche = true;

        while (enMarche)
        {
            try
            {
                byte[] buf = new byte[1000];
                DatagramPacket dtg = new DatagramPacket(buf, buf.length);
                socketGroupe.receive(dtg);
                LMsgRecus.add(new String (buf).trim());
            }
        }
    }
}

```

```
        catch (IOException e)
        {
            System.out.println("Erreur dans le thread :-( :" + e.getMessage());
            enMarche = false; // fin
        }
    }
}
```

Un exemple d'exécution pourrait être :



## 9. UDP et le multicast en C# sous .NET

### 9.1 La classe UdpClient

A l'image de la classe TcpClient, .NET fournit une classe **UdpClient** dont la manipulation est fort semblable à celle de sa consoeur. Ainsi, les constructeurs sont :

```
public UdpClient ( int );
public UdpClient ( IPEndPoint );
```

Les paramètres dont il est ici question sont des éléments locaux pour une communication UDP.

On peut encore s'étonner de l'existence de méthodes comme

```
public void Connect ( IPEndPoint endPoint );
public void Connect ( IPAddress addr, int port );
public void Connect ( string hostname, int port );
```

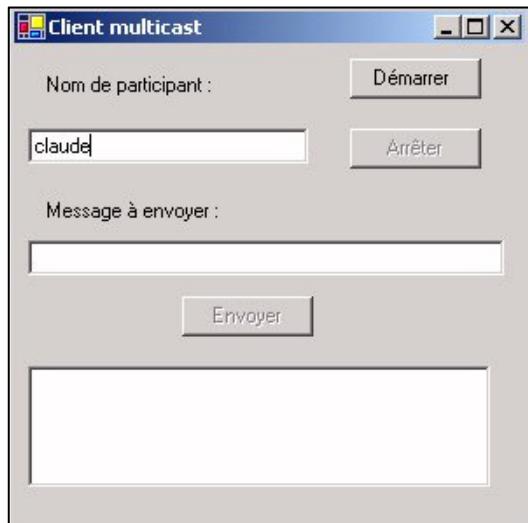
qui semblent parler de connexion pour un *protocole non connecté* ;-) ... En fait, il s'agit de mémoriser l'adresse et le port vers lesquels on va *probablement* envoyer les messages. On trouve donc dans cette classe les méthodes attendues d'envoi et de réception d'un tableau de bytes :

```
public int Send ( byte[] dgram, int nbreBytes);
public int Send ( byte[] dgram, int nbreBytes, IPEndPoint remoteEP);
public byte[] Receive ( ref IPEndPoint remoteEP );
```

avec une méthode Send polymorphe dont la deuxième version permet de préciser un destinataire autre que celui prévu.

### 9.2 La création d'un participant multicast

Supposons à présent vouloir à nouveau programmer un mini-chat de conversation par réseau analogue à celui construit en Java. Chaque participant aura donc à sa disposition le modeste interface suivant :



Outre la création d'un objet UdpClient lié à la machine hôte, il nous faudra aussi une méthode pour rejoindre et quitter le groupe multicast. Celles-ci se prototypent notamment sous la forme :

```
public void JoinMulticastGroup ( IPAddress multicastAddr );  
public void DropMulticastGroup ( IPAddress multicastAddr );
```

Dans notre cas, nous allons donc déclarer :

```
IPAddress adresseGroupe;  
UdpClient cli;
```

puis réaliser l'adhésion au groupe :

```
adresseGroupe = IPAddress.Parse("234.5.5.9");  
cli = new UdpClient(5001);  
cli.JoinMulticastGroup(adresseGroupe);
```

Pour envoyer des messages au groupe, nous créerons :

```
IPEndPoint socketGroupe = new IPEndPoint(adresseGroupe, 5001);
```

et nous utiliserons la méthode d'envoi en multicast :

```
public int Send ( byte[] dgram, int nbreBytes, IPEndPoint remoteEP);
```

### **9.3 Le thread de réception des messages**

Comme en Java, nous allons charger un thread de recevoir les multiples messages et de les afficher dans une boîte de liste. Pour rappel, un thread de .NET est une instance de la classe **Thread**, dont le constructeur est

```
public Thread ( ThreadStart start);
```

où le paramètre est un délégué qui désigne la méthode à appeler au début de l'exécution de ce thread :

```
public delegate void ThreadStart();
```

Le thread est effectivement lancé lorsque l'on exécute sa méthode start(). De plus, nous allons le faire tourner en arrière-plan en utilisant la propriété :

```
public bool IsBackground {get; set;}
```

#### **9.4 Un chat élémentaire (bis)**

Nous pouvons donc à présent rédiger le code d'un client de notre chat multicast :

##### **ClientMulticast.cs**

```
using System;
...
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Text;
using System.IO;

namespace ClientMulticast
{
    public class ClientMulticast : System.Windows.Forms.Form
    {
        UdpClient cli;
        IPAddress adresseGroupe;
        IPEndPoint socketGroupe;
        int portGroupe; int portEnvoi;
        bool enMarche;
        String nomCli;
        public static AutoResetEvent FinChat = new AutoResetEvent(false);

        private System.Windows.Forms.TextBox ZENomClient;
        private System.Windows.Forms.Button BDemarrer;
        private System.Windows.Forms.Button BArreter;
        private System.Windows.Forms.TextBox ZEMessage;
        private System.Windows.Forms.Button BEnvoyer;
        public System.Windows.Forms.ListBox LMsgRecus;
        ...

        public ClientMulticast()
        {
            InitializeComponent();
            BDemarrer.Enabled = true; BArreter.Enabled = false;
            BEnvoyer.Enabled = false;

            adresseGroupe = IPAddress.Parse("234.5.5.9");
            Console.WriteLine("adresse groupe OK");
            portGroupe = 5001; portEnvoi = 5001;
        }

        private void InitializeComponent()
        {
            this.ZENomClient = new System.Windows.Forms.TextBox();
            this.BDemarrer = new System.Windows.Forms.Button();
            this.BArreter = new System.Windows.Forms.Button();
            this.ZEMessage = new System.Windows.Forms.TextBox();
            this.BEnvoyer = new System.Windows.Forms.Button();
        }
    }
}
```

```

this.LMsgRecus = new System.Windows.Forms.ListBox();
this.SuspendLayout();
...
//
// BDemarrer
//
this.BDemarrer.Location = new System.Drawing.Point(192, 8);
this.BDemarrer.Name = "BDemarrer";
this.BDemarrer.Text = "Démarrer";
this.BDemarrer.Click += new System.EventHandler
    (this.BDemarrer_Click);
//
// BArreter
//
this.BArreter.Location = new System.Drawing.Point(192, 48);
this.BArreter.Name = "BArreter";
this.BArreter.Text = "Arrêter";
this.BArreter.Click += new System.EventHandler(this.BArreter_Click);
//
// BEnvoyer
//
this.BEnvoyer.Location = new System.Drawing.Point(96, 144);
this.BEnvoyer.Name = "BEnvoyer";
this.BEnvoyer.Text = "Envoyer";
this.BEnvoyer.Click += new System.EventHandler
    (this.BEnvoyer_Click);
...
}

[STAThread]
static void Main()
{
    Application.Run(new ClientMulticast());
}

private void BDemarrer_Click(object sender, System.EventArgs e)
{
    cli = new UdpClient (portGroupe);
    Console.WriteLine("UdpClient créé");
    cli.JoinMulticastGroup(adresseGroupe);
    Console.WriteLine("Adhésion au groupe multicast OK");

    socketGroupe = new IPEndPoint(adresseGroupe, portEnvoi);
    Console.WriteLine("Socket groupe créée");

    Thread recepteur = new Thread(new ThreadStart(recevoir));
    Console.WriteLine("Thread créé");
    recepteur.IsBackground = true;
    recepteur.Start();
    Console.WriteLine("Thread lancé");
}

```

```

BDemarrer.Enabled = false; BArreter.Enabled = true;
BEnvoyer.Enabled = true;

nomCli = ZENomClient.Text;
String msgDeb = nomCli + " rejoint le groupe";
byte[] msg = ASCIIEncoding.Default.GetBytes(msgDeb);
cli.Send(msg, msg.Length, socketGroupe);
}

private void recevoir()
{
    Console.WriteLine("Début du Thread ");
    enMarche = true;

    while (enMarche)
    {
        IPEndPoint ep = null;
        byte[] buf = cli.Receive(ref ep);
        String msg = ASCIIEncoding.Default.GetString(buf);
        Console.WriteLine("Msg reçu = " + msg);
        LMsgRecus.Items.Add(msg);

        if (!enMarche) FinChat.Set();
    }
}

private void BEnvoyer_Click(object sender, System.EventArgs e)
{
    String msgCli = nomCli + "> " + ZEMessage.Text;
    byte[] msg = ASCIIEncoding.Default.GetBytes(msgCli);
    cli.Send(msg, msg.Length, socketGroupe);
}

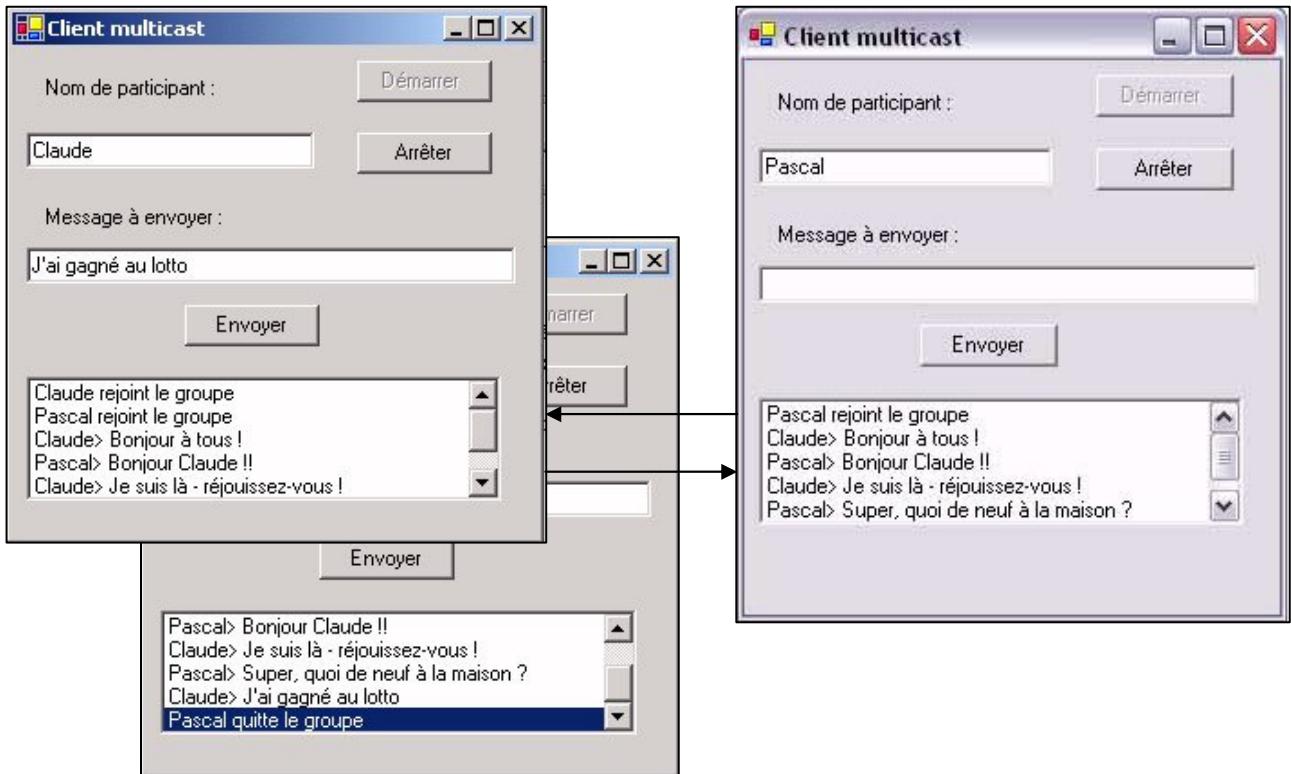
private void BArreter_Click(object sender, System.EventArgs e)
{
    enMarche = false;
    byte[] msgFin = ASCIIEncoding.Default.GetBytes
        (nomCli + " quitte le groupe");
    cli.Send(msgFin, msgFin.Length, socketGroupe);

    FinChat.WaitOne();
    cli.DropMulticastGroup(adresseGroupe);
    cli.Close();

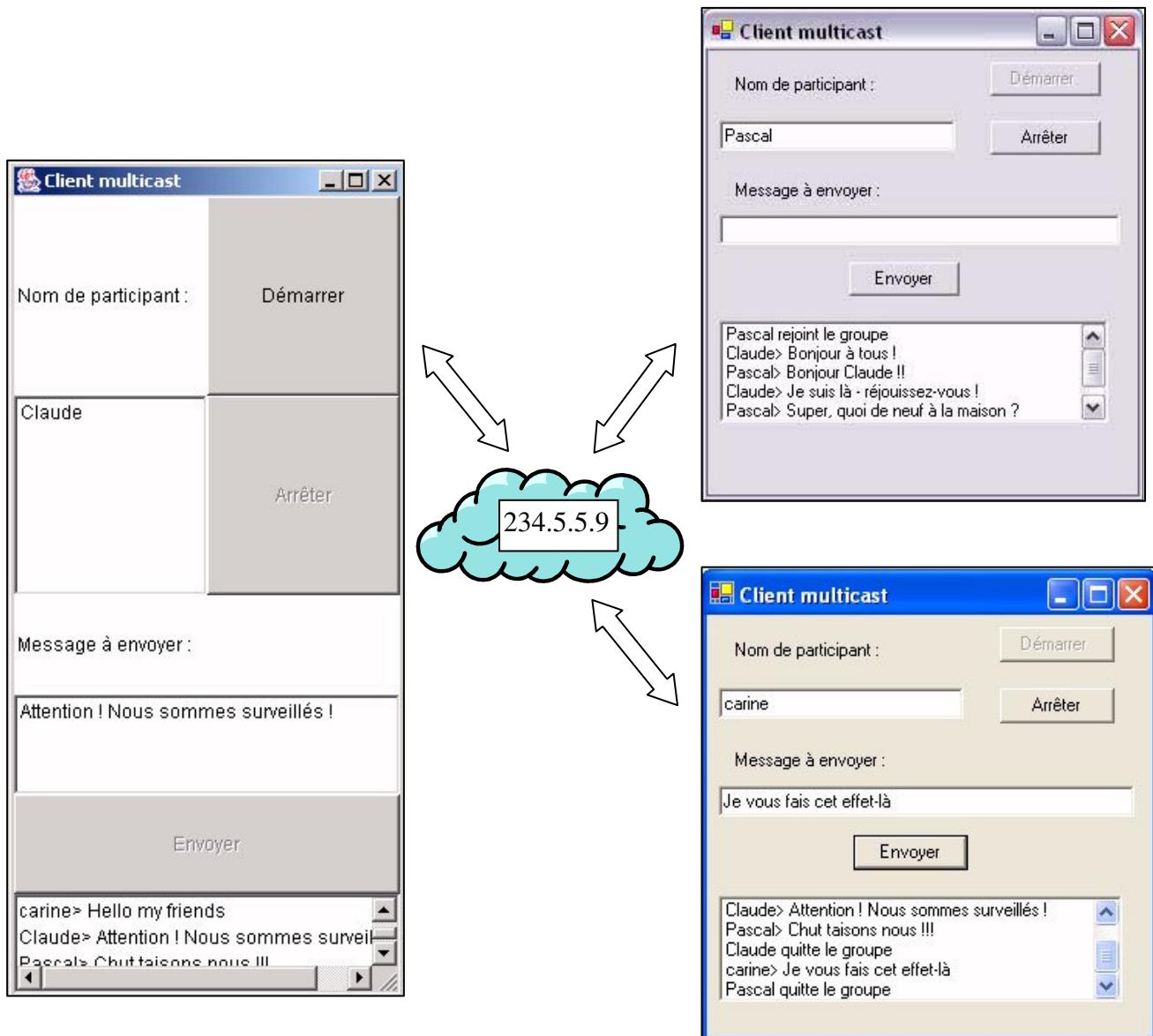
    BDemarrer.Enabled = true; BArreter.Enabled = false;
    BEnvoyer.Enabled = false;
}
}
}

```

On remarquera la synchronisation lors de l'arrêt, afin d'éviter de tenter de manipuler une socket déjà libérée ... Un petit exemple de discussion entre un client Windows 2000 et un autre client Windows XP :



Il est bien sûr possible d'envisager une conversation avec plusieurs participants, certain Java et d'autre C#/NET : ô interopérabilité !



Il faudrait tout de même penser à apprendre à paramétriser les sockets pour qu'elles ne soient pas forcément bloquantes ou encore pour qu'elles soient soumises à des time-out ... Courage ! ➔

## VIII. Le paramétrage des sockets



*On a raison d'exclure les femmes des affaires publiques et civiles : rien n'est plus opposé à leur vocation naturelle que tout ce qui leur donnerait des rapports de rivalité avec les hommes.*

(Madame de Staël, De l'Allemagne)

Il faut bien l'admettre, nous ne nous sommes guère posé de question au sujet des caractéristiques des sockets que nous utilisons. En fait, leur comportement par défaut a presque toujours convenu à nos schémas de résolution des problèmes rencontrés. Il peut cependant ne pas en être toujours ainsi, par exemple pour les messages urgents. Voyons donc comment mieux maîtriser le comportement de nos points de communication.

### 1. Le mode non-bloquant

On peut donner à une socket des caractéristiques non bloquantes pour les connexions et les échanges de données en utilisant deux fonctions qui, au demeurant, ne sont pas propres aux sockets mais concernent en fait tout fichier au sens d'UNIX : il s'agit de fcntl() et ioctl().

#### 1.1 Le contrôle des descripteurs de socket (fcntl)

Utilisant les headers fcntl.h, sys/types.h et unistd.h, cette fonction utilisée classiquement pour les fichiers a pour prototype :

```
int fcntl (    <descripteur de fichier ou de socket – int>,
                <requête – int>,
                <argument de la requête – int ou struct flock *> );
```

L'utilisation de cette fonction, au spectre plus large rappelons-le, est ici celle qui emploie comme requête (dans fcntl.h) :

```
#define F_SETFL     4      /* Set file flags           */
```

avec les indicateurs combinables :

indicateurs	signification
#define O_NONBLOCK 00000004	/* non-blocking I/O, <b>POSIX</b> style */
#define O_NDELAY     00100000	/* Non-blocking I/O */ - System V style
#define FNDELAY     O_NDELAY	BSD style

On peut donc passer en mode non bloquant mais selon des indicateurs variables selon le système hôte. Quel est l'effet de ce mode sur les opérations de base sur les sockets ?

opération	blocage réalisé par	effet
envoi de caractères	O_NONBLOCK	<ul style="list-style-type: none"> <li>◆ tout ce qui peut être écrit l'est; les caractères en excès par rapport à la taille du buffer sont perdus;</li> <li>◆ si il n'y a aucun caractère, la fonction retourne -1 et errno vaut EAGAIN</li> </ul>
	O_NDELAY	idem, mais le retour vaut 0
réception de caractères	O_NONBLOCK	si il n'y a rien à lire, la fonction retourne -1 et errno vaut EAGAIN
	O_NDELAY	si il n'y a rien à lire, la fonction retourne 0
acceptation d'une connexion	O_NONBLOCK	si il n'y a pas de connexion pendante, la fonction retourne -1 et errno vaut EAGAIN
	O_NDELAY	si il n'y a pas de connexion pendante, la fonction retourne -1 et errno vaut EWOULDBLOCK
demande de connexion	O_NONBLOCK	<ul style="list-style-type: none"> <li>◆ la demande provoque un retour immédiat avec -1 et EINPROGRESS – la demande connexion n'est pas perdue, elle est simplement en cours;</li> <li>◆ de nouvelles tentatives peuvent donner : <ul style="list-style-type: none"> <li>-1 et ETIMEOUT : échec;</li> <li>0 : connexion établie;</li> <li>-1 et EALREADY : la tentative est toujours en cours.</li> </ul> </li> </ul>
	O_NDELAY	

## 1.2 Le contrôle des périphériques flux (ioctl)

Prototypée dans stropts.h, cette célèbre fonction d'I/O d'UNIX s'écrit :

```
int ioctl(    <descripteur de fichier ou de socket – int>,
            <requête – int>,
            <argument de la requête>);
```

Pour ce qui nous intéresse ici, le troisième argument sera un pointeur d'entier (int \*). La requête concernant l'aspect bloquant ou non correspond à l'indicateur :

```
#define FIONBIO      _IOW('f', 126, int) /* set/clear non-blocking i/o */
```

On passe en mode non bloquant si le troisième paramètre est non nul.

La requête de passage en mode asynchrone (permettant l'utilisation du signal SIGIO) est celle qui utilise :

```
#define FIOASYNC     _IOW('f', 125, int) /* set/clear async i/o */
```

Le non-bloquage avec ioctl() a des effets similaires à ceux de fcntl() :

opération	blocage réalisé par	effet
envoi de caractères	FIONBIO	<ul style="list-style-type: none"> <li>◆ tout ce qui peut être écrit l'est; les caractères en excès par rapport à la taille du buffer sont perdus;</li> <li>◆ si il n'y a aucun caractère, la fonction retourne -1 et errno vaut EWOULDBLOCK</li> </ul>
réception de caractères	FIONBIO	si il n'y a rien à lire, la fonction retourne -1 et errno vaut EWOULDBLOCK
acceptation d'une connexion	FIONBIO	si il n'y a pas de connexion pendante, la fonction retourne -1 et errno vaut EWOULDBLOCK
demande de connexion	FIONBIO	<ul style="list-style-type: none"> <li>◆ la demande provoque un retour immédiat avec -1 et EINPROGRESS – la demande connexion n'est pas perdue, elle est simplement en cours;</li> <li>◆ de nouvelles tentatives peuvent donner : <ul style="list-style-type: none"> <li>-1 et ETIMEOUT : échec;</li> <li>0 : connexion établie;</li> <li>-1 et EALREADY : la tentative est toujours en cours.</li> </ul> </li> </ul>

## 2. Etre le propriétaire d'une socket

Il peut être utile (comme nous le verrons dans le chapitre des caractères urgents) qu'un processus se rende propriétaire d'une socket. Ceci lui permet ainsi de pouvoir être avisé de l'un ou l'autre signal (comme SIGURG ou SIGIO). L'opération peut se faire

- ◆ avec fcntl : en utilisant la requête

```
#define F_SETOWN    6    /* set async I/O owner      */
```

le troisième paramètre étant le pid du process (ou groupe de process). On obtient, à l'inverse, le pid du propriétaire dans ce troisième argument avec la requête :

```
#define F_GETOWN    5    /* get async I/O owner      */
```

- ◆ avec ioctl : les opérations équivalentes aux précédentes se réalisent avec les requêtes

```
#define FIOSETOWN   _IOW('f', 124, int) /* set owner */
#define FIOGETOWN    _IOR('f', 123, int) /* get owner */
```

ou

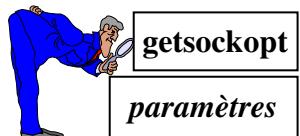
```
#define SIOCSPGRP   _IOW('s', 8, pid_t)    /* set process group */
#define SIOCGPGRP    _IOR('s', 9, pid_t)    /* get process group */
```

### 3. Obtenir des informations sur une socket (getsockopt)

La primitive à utiliser pour se faire une idée des caractéristiques d'une socket est prototypée dans socket.h et a pour forme :

```
int getsockopt (<handle de la socket – int>,
                <niveau (socket ou protocole) – int>,
                <option – int>,
                <valeur correspondante reçue - void *>,
                <longueur de la zone pointée - int *>);
```

Une valeur retournée de –1 ou 0 signifiera une erreur ou un succès.



Les informations obtenues peuvent être interprétées à différents niveaux; ce niveau est spécifié par le deuxième paramètre. Ainsi, dans le domaine AF\_INET, deux niveaux sont le plus fréquemment utilisés :

- ◆ le niveau socket, désigné par la constante (de sys/socket.h) :

```
#define SOL_SOCKET      0xffff          /* options for socket
level */
```

- ◆ le niveau protocole : les informations obtenues s'appliquent donc en correspondance avec le protocole spécifié au moyen des constantes définies dans netinet/in.h (comme **IPPROTO\_IP**, **IPPROTO\_TCP** ou **IPPROTO\_UDP**).

L'option, fixée par le troisième paramètre, sur laquelle on se renseigne peut être

- ◆ booléenne : elle s'applique à la socket ou pas à l'instant considéré.
- ◆ non booléenne : l'information reçue est donc cette fois une véritable valeur qui est celle d'une caractéristique de la communication dont la socket est chargée.

La nature exacte de l'avant-dernier paramètre dépend évidemment de la nature du renseignement demandé. Le dernier paramètre contient la longueur de la zone recevant l'information : il est initialisé avec la longueur attendue, mais rectifié éventuellement par la primitive sur la longueur de son résultat.



Si la valeur de retour est –1, une erreur s'est produite et la variable globale errno est positionnée sur l'une des valeurs suivantes :

<i>valeur de errno</i>	<i>erreur</i>
#define EBADF 9	/* Bad file number */ : le descripteur est invalide, c'est-à-dire qu'il n'est pas associé à une socket
#define ENOTSOCK 38	/* Socket operation on non-socket */ : le descripteur n'est pas associé à une socket, mais à un fichier
#define ENOPROTOOPT 42	/* Protocol not available */ : option booléenne non applicable au protocole
#define EOPNOTSUPP 45	/* Operation not supported on socket */ : l'option n'est pas applicable à ce niveau
#define EINVAL 22	/* Invalid argument */ : moins clair – l'option n'est pas applicable au niveau SOL_SOCKET
#define EFAULT 14	/* Bad address */ : le paramètre d'option ou celui de sa longueur est incorrect

A titre d'exemple, demandons nous ce qu'est la longueur maximale d'un segment TCP. Nous travaillons donc au niveau du protocole TCP (constante **IPPROTO\_TCP**) et l'information demandée peut être obtenue en utilisant la constante :

```
#define TCP_MAXSEG 0x02 /* maximum segment size */
```

Ainsi, on peut programmer, dans le contexte d'un serveur ou d'un client classique :

<b>Utilisation de getsockopt (TCP_MAXSEG)</b>
<pre>... int tailleS, tailleO; ... tailleO = sizeof(int); if (getsockopt(hSocketConnectee, IPPROTO_TCP, TCP_MAXSEG, &amp;tailleS, &amp;tailleO) == -1) {     printf("Erreur sur le getsockopt de la socket %d\n", errno);     exit(1); } else {     printf("getsockopt OK\n");     printf("Taille maximale d'un segment = %d\n", <b>tailleS</b>); }</pre>

ce qui donne (sur la machine UNIX Boole) :

| Taille maximale d'un segment = 1460

Sans vouloir prétendre à l'exhaustivité, passons à présent en revue les options les plus utiles ...

#### 4. **Modifier les options d'une socket (setsockopt)**

Ceci est du ressort de la primitive jumelle

```
int setsockopt (<handle de la socket – int>,
                <niveau (socket ou protocole) – int>,
                <option – int>,
                <valeur à donner à l'option - void *>,
                <longueur de la zone pointée - int>);
```

Une valeur retournée de  $-1$  ou  $0$  signifiera une erreur ou un succès. On se doute bien que la signification des paramètres ainsi que celle des codes d'erreur est celle de la primitive `getsockopt()`.

La tableau donné page suivante (extrait du livre de R. Stevens) donne un panorama général des options disponibles selon le niveau.

level	opname	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	enable debug tracing	•	int
	SO_DONTROUTE	•	•	bypass routing table lookup	•	int
	SO_ERROR	•		get pending error and clear	int	
	SO_KEEPALIVE	•	•	periodically test if connection still alive	•	int
	SO_LINGER	•	•	linger on close if data to send	linger()	
	SO_OOBINLINE	•	•	leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	receive buffer size	int	
	SO_SNDBUF	•	•	send buffer size	int	
	SO_RCVLOWAT	•	•	receive buffer low-water mark	int	
	SO SNDLOWAT	•	•	send buffer low-water mark	int	
	SO_RCVTIMEO	•	•	receive timeout	timeval()	
	SO SNDTIMEO	•	•	send timeout	timeval()	
	SO_REUSEADDR	•	•	allow local address reuse	•	int
	SO_REUSEPORT	•	•	allow local address reuse	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options	•	(see text)
	IP_RECVDSTADDR	•	•	return destination IP address	•	int
	IP_RECVIF	•	•	return received interface index	•	int
	IP_TOS	•	•	type-of-service and precedence	int	
	IP_TTL	•	•	time-to-live	int	
	IP_MULTICAST_IF	•	•	specify outgoing interface	in_addr()	
	IP_MULTICAST_TTL	•	•	specify outgoing TTL	u_char	
	IP_MULTICAST_LOOP	•	•	specify loopback	u_char	
	IP_ADD_MEMBERSHIP	•	•	join a multicast group	ip_mreq()	
	IP_DROP_MEMBERSHIP	•	•	leave a multicast group	ip_mreq()	
	ICMP6_FILTER	•	•	specify ICMPv6 message types to pass	icmp6_filter()	
IPPROTO_IPV6	IPV6_ADDRFORM	•	•	change address format of socket	int	
	IPV6_CHECKSUM	•	•	offset of checksum field for raw sockets	int	
	IPV6_DSTOPTS	•	•	receive destination options	•	int
	IPV6_HOPLIMIT	•	•	receive unicast hop limit	•	int
	IPV6_HOPOPTS	•	•	receive hop-by-hop options	•	int
	IPV6_NEXTHOP	•	•	specify next-hop address	•	sockaddr()
	IPV6_PKTINFO	•	•	receive packet information	•	int
	IPV6_PKTOPTIONS	•	•	specify packet options	•	(see text)
	IPV6_RTHDR	•	•	receive source route	•	int
	IPV6_UNICAST_HOPS	•	•	default unicast hop limit	int	
	IPV6_MULTICAST_IF	•	•	specify outgoing interface	in6_addr()	
	IPV6_MULTICAST_HOPS	•	•	specify outgoing hop limit	u_int	
	IPV6_MULTICAST_LOOP	•	•	specify loopback	u_int	
	IPV6_ADD_MEMBERSHIP	•	•	join a multicast group	ipv6_mreq()	
	IPV6_DROP_MEMBERSHIP	•	•	leave a multicast group	ipv6_mreq()	
IPPROTO_TCP	TCP_KEEPALIVE	•	•	idle time in seconds before probing	int	
	TCP_MAXRTT	•	•	TCP maximum retransmit time	int	
	TCP_MAXSEG	•	•	TCP maximum segment size	int	
	TCP_NODELAY	•	•	disable Nagle algorithm	•	int
	TCP_STDURG	•	•	interpretation of urgent pointer	•	int

 Les différentes options pour getsockopt() et setsockopt()  
 (© R. Stevens)

Toutes les options ne présentent pas le même intérêt : nous allons donc nous limiter aux plus intéressantes d'un point de vue pratique courant.

## 5. Les options booléennes

Il s'agit donc ici d'options qui s'appliquent ou pas à la socket concernée.

### 5.1 Le broadcast

En utilisant le protocole UDP (donc en utilisant des sockets du domaine AF\_INET et du type SOCK\_DGRAM), il est possible de diffuser un message à toutes les machines d'un réseau écoutant sur un port donné. Il se peut cependant que, selon les implémentations, cette diffusion soit soumise à la possession de priviléges ou ne soit pas du tout disponible (bien sûr, c'est le cas habituel sur les machines UNIX).

L'adresse utilisée dans ce cas est l'adresse de diffusion, dont on peut obtenir la valeur par l'appel de la fonction (déclarée dans stropts.h mais qui réclame aussi quand on l'utilise l'inclusion de net/if.h) :

```
int ioctl(      <descripteur du fichier ou de la socket – int>,
               <type de la requête – int>,
               <argument éventuel>);
```

en utilisant la constante définie dans sys/ioctl.h :

```
#define SIOCGIFBRDADDR _IOWR('i',35, struct ifreq) /* get broadcast addr */
```

Une socket pourra être utilisée pour un broadcast si elle bien sûr attachée à cette adresse de diffusion, mais aussi si elle est paramétrée avec un appel de setsockopt() utilisant la constante de sys/socket.h :

```
#define SO_BROADCAST 0x0020 /* permit sending of broadcast msgs */
```

pour le niveau SOL\_SOCKET.

### 5.2 La réutilisation d'une adresse ou d'un port

En principe, une adresse locale ne peut correspondre qu'à une seule socket. Autrement dit, la primitive bind() donne une erreur si l'on tente de l'invoquer pour lier une socket à une adresse déjà liée avec le même numéro de port à une autre socket.

Il est possible de contourner cet interdit pour les sockets du domaine AF\_INET en positionnant l'option correspondante au moyen des constantes :

```
#define SO_REUSEADDR 0x0004 /* allow local address reuse */
#define SO_REUSEPORT 0x0200 /* allow local addr and port
```

- seule la première option existe et est nécessaire sur les machines Sun.

Ainsi, si l'on souhaite lier plusieurs sockets (dont les handles sont par exemple dans un tableau hSocket) à un même couple adresse-port, il suffit de programmer :

```

int flagReuse = 1;
for (i=0; i<nSoket; i++)
{
    if (setsockopt(hSocket[i], SOL_SOCKET, SO_REUSEADDR,
                  (char *)&flagReuse, sizeof(flagReuse)))
    {
        printf("Erreur sur le setsockopt ADDR de la socket %d\n", errno);
        exit(1);
    }
    /* PAS NECESSAIRE ET INEXISTANTE SUR SUNRAY */
    if (setsockopt(hSocket[i], SOL_SOCKET, SO_REUSEPORT,
                  (char *)&flagReuse, sizeof(flagReuse)))
    {
        printf("Erreur sur le setsockopt iPORT de la socket %d\n", errno);
        exit(1);
    }
}

```

Evidemment, "pourquoi pas ?", mais à quoi cela peut-il servir ? On peut s'en faire une idée avec un exemple de type FTP, où un client introduit une requête sur un port d'adresse donnée, mais reçoit sa réponse (ici, son ou ses fichiers) par un autre port, mais sur la même adresse. En effet, un client FTP se connecte à un serveur FTP d'adresse aaa.bbb.ccc.ddd sur le port 21 typique. Le serveur répond par cette connexion, jusqu'à ce qu'il soit question d'un transfert : le serveur initialise alors une nouvelle double connexion, dédiacée au transfert, entre la socket client et une socket serveur attachée au port 20. Pour cela, le serveur doit donc créer deux nouvelles sockets attachées à la même adresse du serveur, mais avec le port 20. Seulement, selon les options par défaut, le bind() nécessaire est impossible ...

### **5.3 Les connexions perdues**

Lorsqu'une connexion n'est plus utilisée pour échanger des données pendant un certain temps, il est possible que cette connexion se soit rompue sans que l'on s'en soit rendu compte. L'utilisation, au sein de getsockopt(), de la constante

```
#define      SO_KEEPALIVE  0x0008          /* keep connections alive */
```

pour les sockets SOCK\_STREAM du domaine AF\_INET permet de tester la validité de cette connexion. En effet, le constat de la perte de connexion provoque, dans un système UNIX, le déclenchement du signal SIGPIPE; le handler correspondant peut alors entreprendre les actions appropriées.

### **5.4 Une option au niveau protocole**

Par défaut, les petits paquets TCP ne sont pas envoyés immédiatement sur le réseau, mais sont plutôt regroupés pour en former un plus important. Dans le cas de réseaux lents, cette politique se justifie, puisque cela permet de limiter un certain engorgement de petits paquets pénalisants. Evidemment, un certain délai peut s'écouler entre la collecte du premier et du dernier paquet. On peut éviter cette manière de faire, autrement dit forcer l'envoi immédiat de tous les segments au fur et à mesure de leur production, en usant de l'option **TCP\_NODELAY** pour une socket SOCK\_STREAM utilisée au niveau IPPROTO\_TCP.

## 5.5 Les caractères urgents

Les sockets SOCK\_STREAM du domaine AF\_INET peuvent être positionnées pour accepter dans leur buffer de réception un caractère urgent. On utilise pour cela la constante :

```
#define SO_OOBINLINE 0x0100 /* leave received OOB data in line */
```

Mais nous en reparlerons au chapitre suivant ...

---

## 6. Les options non booléennes

Il s'agit donc ici d'options qui représentent des valeurs et non pas la présence ou l'absence d'une caractéristique.

### 6.1 Le type de socket

Utilisée uniquement avec un appel de getsockopt() au niveau SOL\_SOCKET, la constante :

```
#define SO_TYPE 0x1008 /* get socket type */
```

permet d'obtenir en réponse le type de la socket concernée :

```
#define SOCK_STREAM 1 /* stream socket */  
#define SOCK_DGRAM 2 /* datagram socket */  
#define SOCK_RAW 3 /* raw-protocol interface */
```

### 6.2 La longueur maximale d'un segment TCP

Pour rappel (voir exemple du paragraphe 3), on utilise la constante :

```
#define TCP_MAXSEG 0x02 /* maximum segment size */
```

pour obtenir ou diminuer (jamais augmenter – évidemment) la taille du segment. On ne peut jamais utiliser ici que des sockets de type SOCK\_STREAM. Il faut remarquer que la valeur obtenue n'est pas la même selon que la socket utilisée est connectée ou pas. En fait, si elle est connectée, on obtiendra la valeur du MTU diminuée de la taille des en-têtes IP et TCP.

### 6.3 La taille des buffers

Il est possible de traiter la taille des buffers d'émission et de réception d'une socket au moyen des constantes de sys/socket.h :

```
#define SO_SNDBUF 0x1001 /* send buffer size */  
#define SO_RCVBUF 0x1002 /* receive buffer size */
```

Outre l'aspect informatif, on peut être amené à modifier ces valeurs selon la densité des envois ou des réceptions. Le petit programme suivant affiche les valeurs courantes de la taille de ces buffers :

### Utilisation de getsockopt (SO\_RCVBUF et SO\_SNDBUF)

```

int tailleBufRecv, taille;
...
if (getsockopt(hSocketConnectee, SOL_SOCKET, SO_RCVBUF, &tailleBufRecv, &taille)
    == -1)
{ printf("Erreur sur le getsockopt de la socket %d\n", errno); exit(1); }
else
{
    printf("getsockopt OK\n"); printf("Taille du buffer de reception = %d\n", tailleBufRecv);
}
if (getsockopt(hSocketConnectee, SOL_SOCKET, SO_SNDBUF, &tailleBufRecv, &taille)
    == -1)
{ printf("Erreur sur le getsockopt de la socket %d\n", errno); exit(1); }
else
{
    printf("getsockopt OK\n");
    printf("Taille du buffer d'emission = %d\n", tailleBufRecv);
}

```

avec pour résultat pour TCP :

```

getsockopt OK
Taille du buffer de reception = 33580
getsockopt OK
Taille du buffer d'emission = 33580

```

Une utilisation classique de ces options est d'aligner la taille des buffers au MTU de la machine considérée.

#### 6.4 Les time-out

On se souviendra que le risque de blocage sur un recvfrom() UDP pouvait être circonvenu en utilisant un time-out; celui-ci était programmé par le traitement du signal SIGALRM. On peut ainsi paramétriser la connexion au niveau socket (SOL\_SOCKET) pour qu'elle soit soumise à une telle contrainte de temps en utilisant dans un appel de setsockopt() la constante :

#### SO\_RCVTIMEO

Il faut alors préciser le temps du time-out au moyen d'un variable de type struct timeval définie dans sys/time.h :

### struct timeval (sys/time.h)

```

struct timeval
{
    time_t tv_sec;      /* seconds */
    int    tv_usec;     /* microseconds */
};

```

Les deux paramètres spécifiques de setsockopt() sont alors l'adresse d'une telle structure et sa longueur.

Le time-out est détecté si un appel de recvfrom(), en principe bloqué sur l'attente d'une réception, s'est débloqué sans que des données soient parvenues à l'application : c'est ce que reflète la valeur EWOULDBLOCK de errno.

### UDPCLI04.C

```
/* UDPCLI04.C
- Claude Vilvens -
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <sys/time.h> /* pour les types temps */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#define PORT_SERVEUR 5000 /* Port de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

struct timeval temps;

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost, * infosOther;
    struct in_addr adresseIP, adresseIPOther;
    struct sockaddr_in adresseSocketServeur;
    struct sockaddr_in adresseSocketClient;
    unsigned int tailleSockaddr_in;

    char msgClient[MAXSTRING], msgServeur[MAXSTRING];
    int nbreRecv, cpt=0;

    /* 1. Creation de la socket */
    hSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");
}
```

```

/* 2. Acquisition des informations sur l'ordinateur local */
if ( (infosHost = gethostbyname("dec01"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    exit(1);
}
else printf("Acquisition infos host OK\n");
memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
printf("Adresse IP host = %s\n",inet_ntoa(adresseIP));

/* 3. Preparation de la structure sockaddr_in du client */
tailleSockaddr_in = sizeof(struct sockaddr_in);
memset(&adresseSocketClient, 0, tailleSockaddr_in);
adresseSocketClient.sin_family = AF_INET;
memcpy(&adresseSocketClient.sin_addr, infosHost->h_addr, infosHost->h_length);

/* 4. Acquisition des informations sur l'autre ordinateur */
if ( (infosOther = gethostbyname("boole"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);exit(1);
}
else printf("Acquisition infos other OK\n");
memcpy(&adresseIPOther, infosOther->h_addr, infosOther->h_length);
printf("Adresse IP other = %s\n",inet_ntoa(adresseIPOther));

/* 5. Preparation de la structure sockaddr_in du serveur */
memset(&adresseSocketServeur, 0, tailleSockaddr_in);
adresseSocketServeur.sin_family = AF_INET;
adresseSocketServeur.sin_port = htons(PORT_SERVEUR);
memcpy(&adresseSocketServeur.sin_addr, infosOther->h_addr,
       infosOther->h_length);

/* 6. Paramétrage de la socket sur un time-out de 5 secondes */
temps.tv_sec = 5; temps.tv_usec = 0;
setsockopt(hSocket, SOL_SOCKET, SO_RCVTIMEO, &temps, sizeof(temps));

do
{
/* 7. Envoi d'un message au serveur */
printf("Message a envoyer au serveur : ");gets(msgClient);
if (sendto(hSocket, msgClient, MAXSTRING, 0,
           &adresseSocketServeur, tailleSockaddr_in) == -1)
{
    printf("Erreur sur le sendto de la socket %d\n", errno);
    close(hSocket); exit(1);
}
else
{
    printf("Sendto socket OK\n");
}
}

```

```

/* 8.Reception d'un message serveur */
    memset(msgServeur, 0, MAXSTRING);
    if ((nbreRecv = recvfrom(hSocket, msgServeur, MAXSTRING, 0,
        &adresseSocketServeur, &tailleSockaddr_in) ) == -1)
    {
        if (errno == EWOULDBLOCK)
        {
            cpt++;
            puts("Time-out sur le recvfrom !");
            if (cpt<3) continue;
            else
            {
                puts("Le serveur semble etre mort ..."); break;
            }
        }
        else
        {
            printf("Erreur sur le recvfrom de la socket %d\n", errno);
            close(hSocket); /* Fermeture de la socket */
            exit(1);
        }
    }
    else
    {
        if (cpt>0) cpt=0;
        printf("Recvfrom socket OK\n");
    }
    msgClient[nbreRecv+1]=0;

    printf("Message envoye par le serveur = %s\n", msgServeur);
    printf("Adresse de l'emetteur = %u\n", adresseSocketServeur.sin_addr.s_addr);
    adresseIPOther = adresseSocketServeur.sin_addr;
    printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseIPOther));
    printf("Port de l'emetteur = %u\n", ntohs(adresseSocketServeur.sin_port));
}
while (strcmp(msgClient, "SHUTDOWN!"));

/* 9. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");

return 0;
}

```

Un exemple d'exécution du côté client pourrait être ceci :

Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.4.131
Acquisition infos other OK

Adresse IP other = 10.59.4.1

Message a envoyer au serveur : *Bonjour ! C'est Mimi !*

Sendto socket OK

Recvfrom socket OK

Message envoyé par le serveur = ACK pour votre message : <Bonjour ! C'est Mimi >

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = 5000

Message a envoyer au serveur : *J'avais hate de te voir !*

Sendto socket OK

Recvfrom socket OK

Message envoyé par le serveur = ACK pour votre message : <J'avais hate de te voir>

Adresse de l'emetteur = 17054474

Adresse de l'emetteur = 10.59.4.1

Port de l'emetteur = 5000

Message a envoyer au serveur : *Pourquoi ne dis-tu rien ?*

Sendto socket OK

**Time-out sur le recvfrom !**

Message a envoyer au serveur : *Dis quelque chose !*

Sendto socket OK

**Time-out sur le recvfrom !**

Message a envoyer au serveur : *Y a quelqu'un ?*

Sendto socket OK

**Time-out sur le recvfrom !**

**Le serveur semble etre mort ...**

Socket client fermee

Le serveur  
vient d'être  
arrêté !

## 7. L'utilisation des options pour le multicast

Nous avons évoqué dans le chapitre précédent les options nécessaires à la réception de datagrammes multicast ainsi que celles envisageables pour l'envoi de tels datagrammes. Pour rappel :

constantes pour setsockopt()	action
IP_ADD_MEMBERSHIP (12)	pour rejoindre un groupe multicast
IP_DROP_MEMBERSHIP (13)	pour quitter un groupe multicast
IP_MULTICAST_TTL (10)	pour spécifier le TTL d'un message multicast sortant
IP_MULTICAST_IF (9)	pour spécifier l'adresse à utiliser pour les messages multicasts sortants
IP_MULTICAST_LOOP (11)	pour spécifier si un message multicast sortant doit être aussi envoyé à la machine émettrice elle-même (loopback)

Supposons donc vouloir créer un receveur Unix pour notre chat multicast du chapitre précédent, chat auquel des clients Java et C#/.NET peuvent déjà participer. Par "receveur", on entend que notre client va se contenter de réceptionner les messages sans envoyer de réponse à la conversation. Nous avons donc besoin d'une socket de réception

- ◆ liée à l'adresse multicast (bind() classique);
- ◆ qui va se joindre au groupe multicast en utilisant

```
setsockopt(hSocket, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

si mreq est une structure du type

```
struct ip_mreq
{
    struct in_addr imr_multiaddr;      /* IP multicast address of group */
    struct in_addr imr_interface;     /* local IP address of interface */
};
```

Le premier champ de cette structure contiendra l'adresse multicast, tandis que le deuxième contiendra l'adresse utilisée localement (que nous nous fournirons avec htonl(INADDR\_ANY) pour obtenir l'adresse par défaut pour le système hôte). Cela donne donc pour notre auditeur de chat :

### MULTICAST01.C

```
/* MULTICAST01.C
- Claude Vilvens -
Cr: 7/7/2003
Maj: 37/7/2003
*/
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <sys/time.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                        ainsi que le conversion format local/format
                        reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#define PORT_MULTI 5001 /* Port de la socket serveur */

#define MAXSTRING 100 /* Longueur des messages */

int main()
{
    int hSocket; /* Handle de la socket */
    struct hostent * infosHost;
    struct in_addr adresseIP;
```

```

struct sockaddr_in adresseSocketServeur;
unsigned int tailleSockaddr_in;

char msg[MAXSTRING];
int nbreRecv, cpt=0;

struct ip_mreq mreq;
int flagReuse = 1;

/* 1. Creation de la socket */
hSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (hSocket == -1)
{
    printf("Erreur de creation de la socket %d\n", errno); exit(1);
}
else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur local */
if ((infosHost = gethostbyname("copernic"))==0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno); exit(1);
}
else printf("Acquisition infos host OK\n");
memcpy(&adresseIP, infosHost->h_addr, infosHost->h_length);
printf("Adresse IP host = %s\n", inet_ntoa(adresseIP));

/* 3. Preparation de la structure sockaddr_in du serveur */
tailleSockaddr_in = sizeof(struct sockaddr_in);
memset(&adresseSocketServeur, 0, tailleSockaddr_in);
adresseSocketServeur.sin_family = AF_INET;
adresseSocketServeur.sin_port = htons(PORT_MULTI);
adresseSocketServeur.sin_addr.s_addr = inet_addr("234.5.5.9");
puts("adresseSocket prete");

/* 4. Le systeme prend connaissance de l'adresse et du port de la socket */
if (bind(hSocket, (struct sockaddr *)&adresseSocketServeur,
        tailleSockaddr_in) == -1)
{
    printf("Erreur sur le bind de la socket %d\n", errno); exit(1);
}
else printf("Bind adresse et port socket OK\n");

/* 5. Parametrage de la socket */
memcpy(&mreq.imr_multiaddr, &adresseSocketServeur.sin_addr,
        tailleSockaddr_in);
mreq.imr_interface.s_addr = htonl(INADDR_ANY);
printf("Utilisation de l'adresse %s\n", inet_ntoa(mreq.imr_interface));
setsockopt (hSocket, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq,
        sizeof(mreq));

```

```

do
{
/* 6.Reception d'un message serveur */
    memset(msg, 0, MAXSTRING);
    if ((nbreRecv = recvfrom(hSocket, msg, MAXSTRING, 0,
                            (struct sockaddr *)&adresseSocketClient,&tailleSockaddr_in)) == -1)
    {
        printf("Erreur sur le recvfrom de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        printf("Recvfrom socket OK\n");
    }
    msg[nbreRecv+1]=0;

    printf("Message recu = %s\n", msg);
    printf("Adresse de l'emetteur = %s\n", inet_ntoa(adresseSocketClient.sin_addr));
}
while (strcmp(msg, "Arret du chat !"));

/* 9. Fermeture de la socket */
    close(hSocket); /* Fermeture de la socket */
    printf("Socket client fermee\n");
    return 0;
}

```

Une exécution de notre programme donne :

```

71 /prof/vilvens/tcp# c
Creation de la socket OK
Acquisition infos host OK
Adresse IP host = 10.59.5.9
adresseSocket prete
Bind adresse et port socket OK
Utilisation de l'adresse 0.0.0.0
Recvfrom socket OK
Message recu = Claude rejoint le groupe
Adresse de l'emetteur = 234.5.5.9
Recvfrom socket OK
Message recu = Claude> Bonjour à tous !
Adresse de l'emetteur = 234.5.5.9
Recvfrom socket OK
Message recu = Albert rejoint le groupe
Adresse de l'emetteur = 234.5.5.9
Recvfrom socket OK
Message recu = Albert> Hellooooooooo !
Adresse de l'emetteur = 234.5.5.9
...

```

Si nous voulons que notre client soit plus actif, donc envoie des messages à son tour, nous allons sans doute utiliser la même socket pour les envois. Cependant, un datagramme IP ne peut être utilisé par une socket liée à une adresse multicast. Nous serons donc obligé d'utiliser deux sockets, une d'envoi et une de réception ...

On ne perdra pas non plus de vue que, par défaut, les datagrammes multicast sont envoyés avec un TTL (time-to-live) de 1. Autrement dit, ils sont limités sous-réseau sur lequel est physiquement connectée la machine par son interface Ethernet et ils ne passeront pas au-delà du premier routeur rencontré. Il conviendra donc, si nécessaire, de modifier ce TTL :

```
unsigned char ttl = 2;  
setsockopt (hSocket, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```



On se rend bien compte que l'on n'utilise ces options ni tous les jours, ni n'importe comment. Néanmoins, elles trouvent une illustration dans le traitement de messages particuliers : les messages urgents.

## IX. Les caractères urgents



*Le plus lent à promettre est toujours le plus fidèle à tenir.*

(J.J. Rousseau, Du contrat social)

### 1. Le principe du caractère urgent

Leur nom le dit bien, les sockets du type SOCK\_STREAM se lisent de manière séquentielle : on ne peut lire une donnée sans avoir lu au préalable celles qui la précédent.

Cependant, il est possible, dans le domaine AF\_INET notamment et avec le protocole TCP uniquement, d'envoyer un caractère urgent vers la socket à l'autre bout de la connexion pour que celui-ci soit lu immédiatement : on appelle très logiquement ce type de caractère resquilleur un "*caractère urgent*" et le paquet éventuellement associé un "*message urgent*". En anglais, ces caractères "hors norme" sont qualifiés de "**out of band**" : ils ne font pas partie du flux normal d'envoi. De tels caractères sont utilisés dans les protocoles applicatifs Telnet (caractères de contrôle sur un terminal virtuel) et FTP (arrêt de transfert dans un téléchargement de fichiers).

On se souviendra que le protocole TCP prévoit l'existence de tels caractères puisque l'en-tête TCP comporte :

- ◆ le flag TH\_URG signalant un caractère urgent;
- ◆ le champ *pointeur d'urgence* qui permet de repérer le dernier caractère urgent transmis (en fait, il pointe la position suivant celle du caractère urgent au sein du buffer d'émission).

Un émetteur peut provoquer l'envoi d'un caractère urgent en paramétrant l'appel de la primitive send() du flag MSG\_OOB. Il faut cependant remarquer que :

- ◆ si l'on envoie plusieurs caractères au moyen de ce send(), seul le dernier caractère sera considéré comme urgent;
- ◆ le caractère n'est effectivement envoyé que lorsque ceux qui le précèdent l'ont été (il resquille à la lecture, pas à l'écriture); le buffering peut donc retarder l'envoi en question.

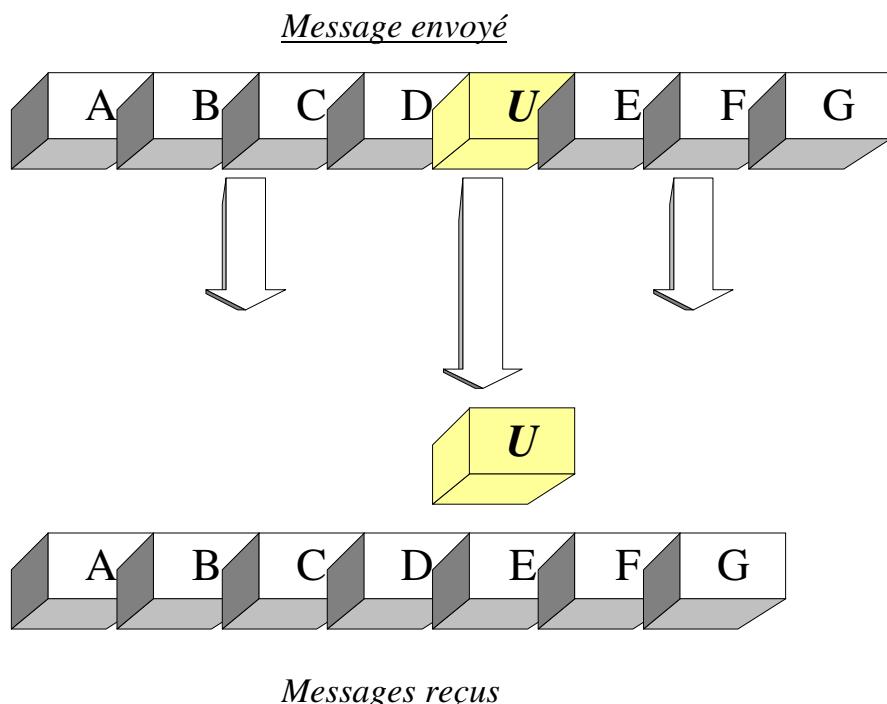
En particulier, la fragmentation (cas d'un buffer plein) peut avoir pour résultat qu'un paquet est envoyé avec le flag urgent alors que le caractère urgent proprement dit fera partie du paquet suivant ☺ ...

La manière dont le récepteur va recevoir et détecter le caractère urgent peut suivre deux politiques différentes.

## 2. La réception hors buffer

Par défaut, le caractère urgent n'est pas intégré dans le buffer de la socket cible mais dans un buffer particulier de 1 byte, le buffer out-of-band; on parle encore de mode "**non OOBINLINE**". En fait, ce n'est que sa position dans la séquence envoyée qui est transmise. La lecture du seul caractère urgent se fait au moyen d'un recv() muni du flag MSG\_OOB. Si l'on programme un appel de recv() sans ce flag, les caractères normaux sont lus jusqu'à la position du caractère urgent. Un autre appel du même genre lirait la suite des caractères normaux, laissant le caractère urgent toujours disponible.

En cas de tentative de lecture de caractère urgent alors qu'il n'y en a pas, le recv() donne une erreur EINVAL. En cas de tentative de lecture d'un caractère urgent annoncé mais non encore arrivé, le recv() donne une erreur EWOULDBLOCK.

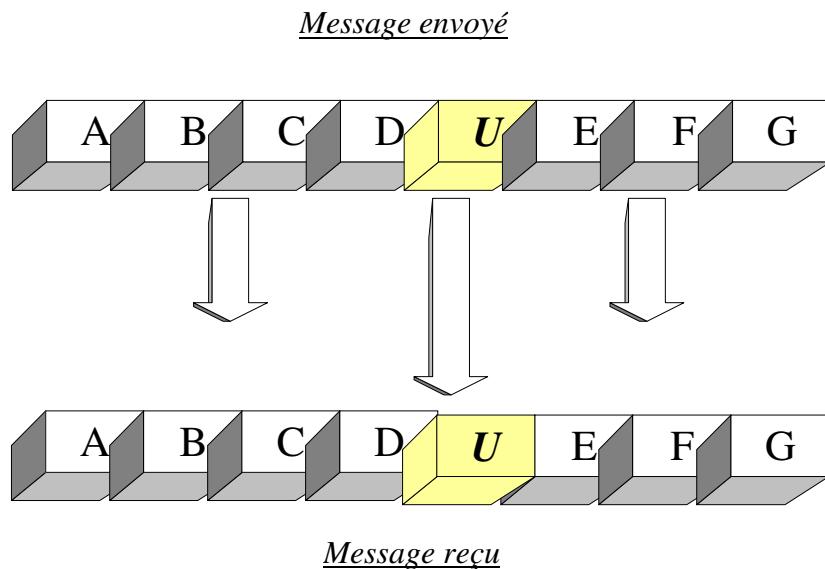


## 3. La réception dans le buffer

Le caractère urgent peut être intégré au message reçu, sa position étant toujours notée; on parle cette fois de mode "**OOBINLINE**". Il faut pour cela paramétriser la socket récepteur au moyen de setsockopt() muni de l'option SO\_OOBINLINE. La lecture s'effectue alors au moyen d'un recv normal (c'est-à-dire sans le flag MSG\_OOB). Mais comme le caractère urgent est à présent noyé dans le message, il faut en repérer la position en détectant *une marque de caractère urgent*, ce qui se fait en utilisant la fonction ioctl() avec la requête correspondant à la constante (définie dans ioctl.h) :

```
#define SIOCATMARK _IOR('s', 7, int) /* at oob mark? */
```

Le troisième paramètre de la requête fournira la position recherchée.



#### 4. Le signal de caractère urgent

Evidemment, tout ceci peut très bien fonctionner parce que le destinataire sait qu'il doit recevoir des caractère urgents. Mais s'il n'en sait rien à priori, il faut qu'il puisse être avisé, de manière asynchrone, de l'imminence de l'arrivée d'un tel caractère. Dans le monde UNIX, cette situation correspond bien sûr à un signal, nommé **SIGURG**, *envoyé au propriétaire de la socket*. Pour être capable de réagir à ce signal (ce qui n'est pas le cas par défaut), le récepteur devra :

- ◆ installer un handler au moyen de la fonction signal (prototypée dans signal.h) :
- ◆ se rendre propriétaire de la socket de réception en utilisant **ioctl()** avec la requête FIOSETOWN ou SIOCSPGRP flanquée de son identificateur (donnée par **getpid()**).

#### 5. Un exemple élémentaire en non OOBINLINE

Pour illustrer notre propos de manière simple, nous allons traiter des caractères urgents en mode de réception hors buffer (non oobinline). Le serveur, des plus élémentaires, se trouve sur boole et, après réception d'un message d'invitation à agir, envoie quelques messages dont l'un sera urgent et les autres pas :

##### **OOBEMETTEUR01.C**

```
/*
 * OOBEMETTEUR01.C
 * - Claude Vilvens -
 */
...
int main()
{
    int hSocketEcoute, /* Handle de la socket */
        hSocketService; /* Handle de la socket connectee au client */
    ...
    int urgentBuffer = 1;
```

```

/* 1. Création de la socket */
...
/* 2. Acquisition des informations sur l'ordinateur local */
...
/* 3. Préparation de la structure sockaddr_in */
...
/* 4. Le système prend connaissance de l'adresse et du port de la socket */
...
/* 5. Mise à l'écoute d'une requête de connexion */
...
/* 6. Acceptation d'une connexion */
if ( (hSocketService =
      accept(hSocketEcoute, (struct sockaddr *)&adresseSocket,
             &tailleSockaddr_in) )
     == -1)
{
    printf("Erreur sur l'accept de la socket %d\n", errno);
    close(hSocketEcoute);exit(1);
}
else printf("Accept socket OK\n");
printf("Adresse du client = %s\n", inet_ntoa(adresseSocket.sin_addr));
finConnexion = 0;

/* 7. Paramétrage de la socket pour qu'elle puisse recevoir un caractère urgent dans son buffer */
if (setsockopt(hSocketService, SOL_SOCKET, SO_OOBINLINE, &urgentBuffer,
               sizeof(urgentBuffer)) == -1 )
    puts("Erreur sur setsockopt");
else puts("setsockopt OK");

/* 8. Reception d'un message client */
if ((nbreRecv =
      recv(hSocketService, msgClient, MAXSTRING, 0)) == -1)
/* pas message urgent */
{
    printf("Erreur sur le recv de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
printf("Recv socket OK\n");
printf("Nombre de caractères reçus = %d\n", nbreRecv);
msgClient[nbreRecv]=0;
printf("Message reçu = %s\n", msgClient);
printf("Longueur du message reçu = %d\n", strlen(msgClient));
if (strcmp(msgClient, EOC)==0)
{
    finConnexion=1;
    printf("*** Le client demande la fin de la connexion ***\n");
}

```

```

/* 9. Envoi de messages successifs par le serveur au client */
if (send(hSocketService, "abcdefg", 7, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Send abcdefg socket OK\n");

if (send(hSocketService, "XYZ", 3, MSG_OOB) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Send XYZ <U> socket OK\n");

if (send(hSocketService, "1234", 4, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Send 1234 socket OK\n");

while (1); /* Pour éviter que le serveur ne s'arrête trop vite – petit petit ;-)
... */

/* 9. Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */
printf("Socket connectee au client fermee\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");

return 0;
}

```

Le récepteur (qui semble très au courant de ce qu'il va recevoir) se trouve sur dec01. Il détournera le signal SIGURG pour lire dans son handler les caractères urgents; pour rappel, ce détournement n'est effectif que si le récepteur se rend propriétaire de la socket. Le signal SIGINT est également détourné pour éviter toute terminaison impromptue.

## OOBRECEPTEUR01.C

```

/* OOBRECEPTEUR01.C
- Claude Vilvens -
*/
...

```

```
#include <signal.h>
#include <fcntl.h>

#include "tcpriter.h"

void traiteUrgent();
void traiteInt();

int hSocket; /* Handle de la socket */
int acceptOob;

int main()
{
    ...
    /* 1. Création de la socket */
    ...
    /* 2. Acquisition des informations sur l'ordinateur distant */
    ...
    /* 3. Préparation de la structure sockaddr_in */
    ...

    /* 4. Détournement de SIGURG et de SIGINT */
    if (signal(SIGURG, traiteUrgent) == BADSIG)
    {
        puts("Erreur sur le signal SIGURG");
        exit(1);
    }
    else puts("SIGURG detourne");

    if (signal(SIGINT, traiteInt) == BADSIG)
    {
        puts("Erreur sur le signal SIGINT");
        exit(1);
    }
    else puts("SIGINT detourne");

    /* 5. Le process se rend propriétaire de la socket */
    if (fcntl(hSocket, F_SETOWN, getpid()) == -1)
    {
        puts("Erreur sur l'appropriation de la socket");
        exit(1);
    }
    else puts ("fcntl - owner OK");

    /* 6. Envoi d'un message client au serveur */
    printf("Message num %d a envoyer : ", cpt + 1);gets(msgClient);
    if (send(hSocket, msgClient, MAXSTRING, 0) == -1)
    {
        printf("Erreur sur le send de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
    }
}
```

```

        exit(1);
    }
    else printf("Send socket OK\n");
    printf("Message envoyé = %s\n", msgClient);

/* 7. Reception des messages normaux du serveur */
    memset(msgServeur, 0, MAXSTRING);
    if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
        if (errno != EINTR)
    {
        printf("1. Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("EINTR\n");
else
{
    printf("1. Recv socket OK\n");
    cpt++;
    printf("%d. Message reçu = %s\n", cpt, msgServeur);
}

memset(msgServeur, 0, MAXSTRING);
if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
    if (errno != EINTR)
    {
        printf("2. Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("EINTR");
else
{
    printf("2. Recv socket OK\n");
    cpt++;
    printf("%d. Message reçu = %s\n", cpt, msgServeur);
}

memset(msgServeur, 0, MAXSTRING);
if (recv(hSocket, msgServeur, MAXSTRING, 0) == -1)
    if (errno != EINTR)
    {
        printf("3. Erreur sur le recv de la socket %d\n", errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("EINTR");
else
{
    printf("3. Recv socket OK\n");
}

```

```

        cpt++;
        printf("%d. Message recu = %s\n", cpt, msgServeur);
    }

/* 9. Fermeture de la socket */
close(hSocket); /* Fermeture de la socket */
printf("Socket client fermee\n");
printf("%d messages envoyes !", cpt);
return 0;
}

/* -----
void traiteUrgent()
/* Réception des caractères urgents */
{
    char caracUrg;
    static cptUrg = 0;

    puts("Passage dans le handler de SIGURG");

    if (recv(hSocket, &caracUrg, 1, MSG_OOB) == -1)
    {
        printf("u%d. Erreur sur le recv de la socket %d\n", ++cptUrg,
               errno);
        close(hSocket); /* Fermeture de la socket */
        exit(1);
    }
    else printf("u%d. Recv socket OK\n Caractere urgent recu = %c\n",
               ++cptUrg, caracUrg);
}

void traiteInt()
{
    puts("Passage dans le handler de SIGINT");
}

```

L'exécution des deux programmes donne :

Du côté de l'émetteur	Du côté du récepteur
<pre> boole&gt; s Creation de la socket OK Acquisition infos host OK Adresse IP = 10.59.4.1 Bind adresse et port socket OK Listen socket OK Accept socket OK Adresse du client = 10.59.4.131 setsockopt OK Recv socket OK ← Nombre de caractères recus = 100 Message recu = Tire je suis pret Longueur du message regu = 17 Send abcdefg socket OK → Send XYZ &lt;U&gt; socket OK → Send 1234 socket OK → </pre>	<pre> % c Creation de la socket OK Acquisition infos host distant OK Adresse IP = 10.59.4.1 Connect socket OK SIGURG detourne SIGINT detourne fcntl - owner OK Message num 1 a envoyer : Tire je suis pret Send socket OK Message envoyé = Tire je suis pret <b>Passage dans le handler de SIGURG</b> u1. Recv socket OK <b>Caractère urgent recu = Z</b> <b>EINTR</b> 2. Recv socket OK 1. Message recu = abcdefgXY 3. Recv socket OK 2. Message recu = 1234 Socket client fermée 2 messages envoyés !% </pre>

On visualise ainsi très bien la "butée" que constitue le caractère urgent ainsi que le fait que les caractères ont été bufférissés.

### Remarques

1) L'envoi de caractères urgents conserve un caractère relativement exceptionnel. Si le récepteur constate son apparition (ou du moins sa tentative d'envoi) de manière asynchrone, il doit en être de même pour l'émetteur qui l'envoie à un moment imprévisible séquentiellement. Le procédé classique consiste pour cela à ajouter au code de l'émetteur la possibilité d'introduction de caractères urgents au moyen d'un CTRL-D.

2) Les sockets de Java connaissent également les caractères urgents puisque l'on trouve dans la classe Socket la méthode :

```
public void sendUrgentData(int data) throws IOException
```



La gestion d'un réseau TCP/IP ne se contente pas seulement de programmation. Il est évidemment vital de disposer d'outils de diagnostic fiables. Cependant, nous n'avons pas ici de prétentions au niveau de l'administration réseau. Nous allons donc nous contenter de quelques commandes : mais l'une d'entre elles est très célèbre ...

## X. Quelques commandes réseaux utiles



*J'en passe, et des meilleurs.*

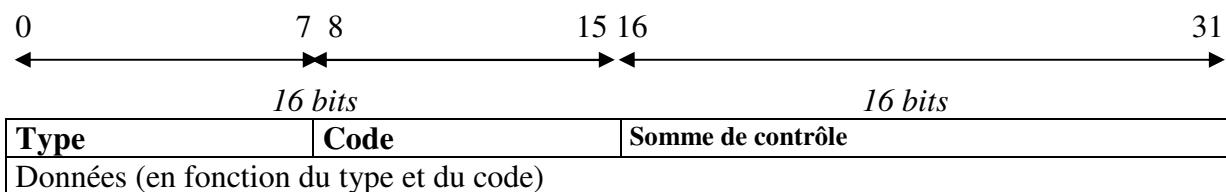
(V. Hugo, Hernani)

### 1. Une commande de diagnostic : ping

#### 1.1 Le protocole ICMP

La commande ping (pour Packet InterNet Groper) est probablement la commande de diagnostic la plus immédiatement et la plus fréquemment utilisée par les programmeurs et gestionnaires réseaux. Elle permet en effet de tester si une machine donnée (ordinateur ou routeur) est actuellement en service sur le réseau ou pas, permettant ainsi d'orienter les recherches dans la résolution d'un problème réseau. Pour ce faire, la commande envoie un datagramme à la machine et en attend une réponse.

En fait, ping utilise le protocole **ICMP**, qui permet principalement aux passerelles de s'échanger des informations de contrôle ou d'erreur. Il se trouve au même niveau que le protocole IP. Les informations traitées par ce protocole sont encapsulées dans un datagramme IP (le champ du protocole dans l'en-tête IP est égal à 1). Une entité ICMP proprement dite ne comporte que 4 champs :



Les valeurs 8 et 0 du champ type correspondent respectivement à une requête et à une réponse d'écho envoyée par ping. Dans le cas où la machine visée est inaccessible, le champ type vaut 3 et le champ code permet de compléter l'information :

- 0 : réseau inaccessible;
- 1 : machine inaccessible;
- 2 : protocole inaccessible;
- 3 : port inaccessible;
- 4 : fragmentation nécessaire, mais non autorisée;
- etc.

Le cas d'un compteur **Time to live** (TTL – voir chapitre 1) qui tombe à 0 correspond à la valeur 11 du champ type (avec un code de 0 ou de 1 selon que cette mise à 0 se soit effectuée durant le transfert ou durant l'assemblage).

La structure C correspondante est définie dans netinet/ip\_icmp.h :

### **structure icmp**

```
struct icmp
{
    u_char icmp_type;          /* type of message, see below */
    u_char icmp_code;          /* type sub code */
    u_short icmp_cksum;        /* ones complement cksum of struct */
    union
    {
        ...
        } ih_idseq;
    ...
};
```

tandis que les valeurs utiles de code et de type correspondent aux constantes :

```
#define ICMP_ECHOREPLY      0      /* echo reply */
#define ICMP_ECHO            8      /* echo service */

#define ICMP_UNREACH         3      /* dest unreachable, codes: */
#define ICMP_UNREACH_NET     0      /* bad net */
#define ICMP_UNREACH_HOST    1      /* bad host */
#define ICMP_UNREACH_PROTOCOL 2      /* bad protocol */
#define ICMP_UNREACH_PORT    3      /* bad port */
#define ICMP_UNREACH_NEEDFRAG 4      /* IP_DF caused drop */
#define ICMP_UNREACH_SRCFAIL  5      /* src route failed */

#define ICMP_TIMXCEED        11     /* time exceeded, code: */
#define ICMP_TIMXCEED_INTRANS 0      /* ttl==0 in transit */
#define ICMP_TIMXCEED_REASS   1      /* ttl==0 in reass */
```

## **1.2 Quelques exemples d'exécution de la commande ping**

La commande s'utilise, dans le cas le plus simple, selon la syntaxe :

<b>ping &lt;nom de la machine cible   adresse IP de la machine cible&gt;</b>
--

### a) sous Unix

La commande ping se trouve dans le répertoire /usr/sbin (chemin qu'il suffit donc d'ajouter au PATH) :

```
bash-2.05b$ /usr/sbin/ping 10.59.4.5
PING 10.59.4.5 (10.59.4.5): 56 data bytes
64 bytes from 10.59.4.5: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 10.59.4.5: icmp_seq=1 ttl=255 time=10 ms
64 bytes from 10.59.4.5: icmp_seq=2 ttl=255 time=10 ms
64 bytes from 10.59.4.5: icmp_seq=3 ttl=255 time=10 ms
64 bytes from 10.59.4.5: icmp_seq=4 ttl=255 time=10 ms
64 bytes from 10.59.4.5: icmp_seq=5 ttl=255 time=10 ms
64 bytes from 10.59.4.5: icmp_seq=6 ttl=255 time=10 ms
<CTRL-C>
```

```
----sunray2v440 PING Statistics----
```

```
7 packets transmitted, 7 packets received, 0% packet loss  
round-trip (ms) min/avg/max = 0/8/10 ms  
bash-2.05b$
```

En cas d'échec, il faut également interrompre la commande par CTRL-C :

```
% /usr/sbin/ping 10.10.0.199  
PING 10.10.0.199 (10.10.0.199): 56 data bytes  
<blocage – il faut un CTRL-C>
```

```
----10.10.0.199 PING Statistics----
```

```
9 packets transmitted, 0 packets received, 100% packet lost  
%
```

Il est aussi possible d'utiliser le nom de la machine cible :

```
copernic.inpres.epl.prov-liege.be> /usr/sbin/ping sunray2v440  
PING sunray2v440 (10.59.4.5): 56 data bytes  
64 bytes from 10.59.4.5: icmp_seq=0 ttl=255 time=0 ms  
64 bytes from 10.59.4.5: icmp_seq=1 ttl=255 time=10 ms  
64 bytes from 10.59.4.5: icmp_seq=2 ttl=255 time=10 ms  
64 bytes from 10.59.4.5: icmp_seq=3 ttl=255 time=10 ms  
64 bytes from 10.59.4.5: icmp_seq=4 ttl=255 time=10 ms  
64 bytes from 10.59.4.5: icmp_seq=5 ttl=255 time=10 ms  
  
----sunray2v440 PING Statistics----  
6 packets transmitted, 6 packets received, 0% packet loss  
round-trip (ms) min/avg/max = 0/8/10 ms  
copernic.inpres.epl.prov-liege.be>
```

Un cas particulier est de tester le bien-fondé réseau de sa propre machine (loopback) :

```
% /usr/sbin/ping localhost  
PING localhost (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=1 ms  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0 ms  
<CTRL-C>  
  
----localhost PING Statistics----  
5 packets transmitted, 5 packets received, 0% packet loss  
round-trip (ms) min/avg/max = 0/0/1 ms  
%
```

Sur Sunray, il faut utiliser le commutateur **-s** (voir paragraphe suivant – un paquet est émis chaque seconde) :

```
bash-2.03$ /usr/sbin/ping -s copernic  
PING copernic: 56 data bytes  
64 bytes from copernic (10.59.5.9): icmp_seq=0. time=0. ms
```

---

```
64 bytes from copernic (10.59.5.9): icmp_seq=1. time=0. ms
64 bytes from copernic (10.59.5.9): icmp_seq=2. time=0. ms
64 bytes from copernic (10.59.5.9): icmp_seq=3. time=0. ms
64 bytes from copernic (10.59.5.9): icmp_seq=4. time=0. ms
64 bytes from copernic (10.59.5.9): icmp_seq=5. time=0. ms
^C
----copernic PING Statistics----
6 packets transmitted, 6 packets received, 0% packet loss
round-trip (ms) min/avg/max = 0/0/0
bash-2.03$
```

b) **sous Windows**

En lançant ping depuis une fenêtre DOS, cela peut donner :

```
C:\WINDOWS>ping 10.10.20.3
Envoi d'une requête 'ping' sur 10.10.20.3 avec 32 octets de données :
Réponse de 10.10.20.3 : octets=32 temps=160 ms TTL=254
Réponse de 10.10.20.3 : octets=32 temps=115 ms TTL=254
Réponse de 10.10.20.3 : octets=32 temps=106 ms TTL=254
Réponse de 10.10.20.3 : octets=32 temps=104 ms TTL=254
Statistiques Ping pour 10.10.20.3:
    Paquets : envoyés = 4, reçus = 4, perdus = 0 (perte 0%),
Durée approximative des boucles en milli-secondes :
    minimum = 104ms, maximum = 160ms, moyenne = 121ms

C:\WINDOWS>
```

Dans le cas d'une adresse inutilisée, on obtiendra :

```
C:\WINDOWS>ping 10.10.0.199
Envoi d'une requête 'ping' sur 10.10.0.199 avec 32 octets de données :
Délai d'attente de la demande dépassé.

Statistiques Ping pour 10.10.0.199:
    Paquets : envoyés = 4, reçus = 0, perdus = 4 (perte 100%),
Durée approximative des boucles en milli-secondes :
    minimum = 0ms, maximum = 0ms, moyenne = 0ms

C:\WINDOWS>
```

Les noms de machines sont toujours disponibles :

```
C:\WINDOWS> ping boole
Envoi d'une requête 'ping' sur boole.inpres.epl.prov-liege.be [10.59.4.1] avec 3
```

2 octets de données :

Réponse de 10.59.4.1 : octets=32 temps=130 ms TTL=62  
Réponse de 10.59.4.1 : octets=32 temps=111 ms TTL=62  
Réponse de 10.59.4.1 : octets=32 temps=108 ms TTL=62  
Réponse de 10.59.4.1 : octets=32 temps=111 ms TTL=62

Statistiques Ping pour 10.59.4.1:

Paquets : envoyés = 4, reçus = 4, perdus = 0 (perte 0%),

Durée approximative des boucles en milli-secondes :

minimum = 108ms, maximum = 130ms, moyenne = 115ms

Et le loopback est bien possible :

C:\WINDOWS>**ping localhost**

Envoi d'une requête 'ping' sur cvilvens.inpres.epl.prov-liege.be [127.0.0.1] avec 32 octets de données :

Réponse de 127.0.0.1 : octets=32 temps<10 ms TTL=128  
Réponse de 127.0.0.1 : octets=32 temps<10 ms TTL=128  
Réponse de 127.0.0.1 : octets=32 temps<10 ms TTL=128  
Réponse de 127.0.0.1 : octets=32 temps<10 ms TTL=128

Statistiques Ping pour 127.0.0.1:

Paquets : envoyés = 4, reçus = 4, perdus = 0 (perte 0%),

Durée approximative des boucles en milli-secondes :

minimum = 0ms, maximum = 0ms, moyenne = 0ms

C:\WINDOWS>

### **1.3 Les options de ping**

Il est donc possible de modifier le comportement par défaut de la commande, selon une syntaxe variable selon la machine hôte.

#### a) **sous Unix**

A la mode UNIX, les options de la commande se spécifient avec "-" suivi d'une lettre et, éventuellement, d'une donnée complémentaire :

**-c <nombre de paquets>**

En cas d'échec, on interrompt la commande par CTRL-C.

```
% /usr/sbin/ping -c 3 10.10.0.100
PING 10.10.0.100 (10.10.0.100): 56 data bytes
64 bytes from 10.10.0.100: icmp_seq=0 ttl=255 time=1 ms
64 bytes from 10.10.0.100: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 10.10.0.100: icmp_seq=2 ttl=255 time=0 ms
```

----10.10.0.100 PING Statistics----

3 packets transmitted, 3 packets received, 0% packet loss  
round-trip (ms) min/avg/max = 0/0/1 ms  
%

**-s <taille des paquets>**

```
% /usr/sbin/ping -c 3 -s 30 10.10.0.100
PING 10.10.0.100 (10.10.0.100): 30 data bytes
38 bytes from 10.10.0.100: icmp_seq=0 ttl=255 time=1 ms
38 bytes from 10.10.0.100: icmp_seq=1 ttl=255 time=0 ms
38 bytes from 10.10.0.100: icmp_seq=2 ttl=255 time=0 ms
```

----10.10.0.100 PING Statistics----

3 packets transmitted, 3 packets received, 0% packet loss  
round-trip (ms) min/avg/max = 0/0/1 ms  
%

**-i <temps en secondes entre l'émission de deux paquet>**

b) **sous Windows**

Sous Windows, on obtient la liste des possibilités par :

C:\WINDOWS>**ping**

Utilisation : ping [-t] [-a] [-n échos] [-l taille] [-f] [-i vie] [-v TypServ]  
[-r NbSauts] [-s NbSauts] [[-j ListeHôtes] | [-k ListeHôtes]]  
[-w Délai] ListeDestination

Options :

- t Envoie la requête ping sur l'hôte spécifié jusqu'à interruption. Entrez Ctrl-Arrêt pour afficher les statistiques et continuer, Ctrl-C pour arrêter.
- a Recherche les noms d'hôte à partir des adresses.
- n échos Nombre de requêtes d'écho à envoyer.
- l taille Envoie la taille du tampon.
- f Active le signal Ne pas fragmenter dans le paquet.
- i vie Durée de vie.
- v TypServ Type de service.
- r NbSauts Enregistre l'itinéraire pour le nombre de sauts.
- s NbSauts Dateur pour le nombre de sauts.
- j ListeHôtes Itinéraire source libre parmi la liste d'hôtes.
- k ListeHôtes Itinéraire source strict parmi la liste d'hôtes.
- w Délai Délai d'attente pour chaque réponse, en millisecondes.

Par exemple :

C:\WINDOWS>**ping -n 3 10.10.0.102**

Envoi d'une requête 'ping' sur 10.10.0.102 avec 32 octets de données :

Réponse de 10.10.0.102 : octets=32 temps=102 ms TTL=63

Réponse de 10.10.0.102 : octets=32 temps=105 ms TTL=63

Réponse de 10.10.0.102 : octets=32 temps=102 ms TTL=63

Statistiques Ping pour 10.10.0.102:

Paquets : envoyés = 3, reçus = 3, perdus = 0 (perte 0%),

Durée approximative des boucles en milli-secondes :

minimum = 102ms, maximum = 105ms, moyenne = 103ms

C:\WINDOWS>

## 2. Une commande de statistiques des sockets : netstat

### 2.1 La visualisation des sockets

Cette commande, disponible tant sous Windows que sous Unix, permet une analyse plus fouillée du réseau que ne le permet la commande ping. Il est notamment possible de voir l'état des sockets (tant UDP que TCP), l'état des connexions pour les protocoles à mode connecté ou de connaître la table de routage utilisée. Dans sa version la plus simple, la syntaxe se limite à l'usage de la commande :

**netstat**

#### a) sous Unix

Sous Unix, la commande se trouve dans le répertoire **/usr/sbin**. Cela peut donner :

```
boole.INPRES.EPL.PROV-LIEGE.BE> /usr/sbin/netstat
Active Internet connections
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp    0      2   boole.inpres.epl.telne  10.10.10.12.1033  ESTABLISHED
tcp    0      0   localhost.1521           localhost.1068    ESTABLISHED
tcp    0      0   localhost.1068           localhost.1521    ESTABLISHED
tcp    0      0   boole.inpres.epl.1521  boole.inpres.epl.1069 ESTABLISHED
tcp    0      0   boole.inpres.epl.1069  boole.inpres.epl.1521 ESTABLISHED
boole.INPRES.EPL.PROV-LIEGE.BE>
```

On interprète un tel écran assez facilement :

colonne	information
1 : Proto	protocole utilisé par la socket
2 : Recv-Q	nombre de bytes dans la file de réception
3 : Send-Q	nombre de bytes dans la file d'envoi
4 : Local Address	adresse locale (avec port) associée à la socket
5 : Foreign Address	sous tcp, adresse distante (avec port) de la socket paire; sous udp, colonne non utilisée.
6 : (state)	état de la socket selon les états possibles du diagramme de transition de TCP (voir chapitre I)

Sous Sunray, les choses sont analogues (l'ordre des colonnes n'est pas le même et il y a des colonnes en plus) mais beaucoup plus populaires :

```
sh-2.03$ netstat
```

**UDP: IPv4**

Local Address	Remote Address	State
sunray2v440.inpres.epl.prov-liege.be.34109	10.59.5.175.54664	Connected
sunray2v440.inpres.epl.prov-liege.be.34117	10.59.5.170.55814	Connected
sunray2v440.inpres.epl.prov-liege.be.34125	10.59.5.165.42537	Connected

...

**TCP: IPv4**

Local Address	Remote Address	Swind	Send-Q	Rwind	Recv-Q	State
sunray2v440..65525	10.59.13.252.40519	130736	0	24616	0	ESTABLISHED
sunray2v440.inpres..10.59.5.144.52405		8095	0	24820	0	ESTABLISHED

...

**Active UNIX domain sockets**

Address	Type	Vnode	Conn	Local Addr	Remote Addr
3000931ea58	stream-ord	00000000	00000000		
3000931ec08	stream-ord	300088ec960	00000000	/tmp/.X11-unix/X22	

...

b) **sous Windows**

De manière similaire :

```
C:\>netstat
```

Connexions actives

Proto	Adresse locale	Adresse distante	Etat
TCP	claude:1024	localhost:3006	ESTABLISHED
TCP	claude:3006	localhost:1024	ESTABLISHED
TCP	claude:3080	localhost:3081	ESTABLISHED
TCP	claude:3081	localhost:3080	ESTABLISHED

```
C:\>
```

## 2.2 Les options de netstat

A nouveau, nous nous contenterons des options les plus courantes ou les plus utiles.

a) **Obtenir les adresses en notation à point**

```
netstat -n
```

Un exemple sur boole donne :

```
| boole.INPRES.EPL.PROV-LIEGE.BE> /usr/sbin/netstat -n
```

## Active Internet connections

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	10.59.4.1.1196	10.7.0.100.53	TIME_WAIT
tcp	0	2	10.59.4.1.23	10.10.10.12.1033	ESTABLISHED
tcp	0	0	127.0.0.1.1521	127.0.0.1.1068	ESTABLISHED
tcp	0	0	127.0.0.1.1068	127.0.0.1.1521	ESTABLISHED
tcp	0	0	10.59.4.1.1521	10.59.4.1.1069	ESTABLISHED
tcp	0	0	10.59.4.1.1069	10.59.4.1.1521	ESTABLISHED

boole.INPRES.EPL.PROV-LIEGE.BE>

b) Afficher toutes les sockets

Autrement dit, il s'agit d'obtenir des informations sur toutes les sockets, quel que soit leur état :

**netstat -a**

Sur boole :

boole.INPRES.EPL.PROV-LIEGE.BE> /usr/sbin/netstat -a					
printing 1 hashtable with 32 buckets					
Active Internet connections (including servers)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	boole.inpres.epl.1193	dns.epl.prov-lie.domai	TIME_WAIT
tcp	0	0	boole.inpres.epl.1194	dns.epl.prov-lie.domai	ESTABLISHED
tcp	0	244	boole.inpres.epl.telne	10.10.10.12.1033	ESTABLISHED
tcp	0	0	localhost.1521	localhost.1068	ESTABLISHED
tcp	0	0	localhost.1068	localhost.1521	ESTABLISHED
tcp	0	0	boole.inpres.epl.1521	boole.inpres.epl.1069	ESTABLISHED
tcp	0	0	boole.inpres.epl.1069	boole.inpres.epl.1521	ESTABLISHED
tcp	0	0	*.2481	.*.*	LISTEN
tcp	0	0	*.1521	.*.*	LISTEN
tcp	0	0	*.1064	.*.*	LISTEN
tcp	0	0	*.6000	.*.*	LISTEN
tcp	0	0	*.1025	.*.*	LISTEN
tcp	0	0	*.printer	.*.*	LISTEN
tcp	0	0	*.1024	.*.*	LISTEN
tcp	0	0	*.dtspc	.*.*	LISTEN
tcp	0	0	*.cfgmgr	.*.*	LISTEN
tcp	0	0	*.kdebug	.*.*	LISTEN
tcp	0	0	*.finger	.*.*	LISTEN
tcp	0	0	*.exec	.*.*	LISTEN
tcp	0	0	*.login	.*.*	LISTEN
tcp	0	0	*.shell	.*.*	LISTEN
tcp	0	0	*.telnet	.*.*	LISTEN
tcp	0	0	*.ftp	.*.*	LISTEN
tcp	0	0	*.AdvFS	.*.*	LISTEN
tcp	0	0	*.smtp	.*.*	LISTEN
tcp	0	0	*.auditd	.*.*	LISTEN
tcp	0	0	*.2049	.*.*	LISTEN

```

tcp    0    0 *.897      *.*      LISTEN
tcp    0    0 *.111      *.*      LISTEN
udp    0    0 *.177      *.*      
udp    0    0 *.1047     *.*      
udp    0    0 *.1046     *.*      
udp    0    0 *.time     *.*      
udp    0    0 *.ntalk    *.*      
udp    0    0 *.biff     *.*      
udp    0    0 *.1045     *.*      
udp    0    0 *.advfsd-s  *.*      
udp    0    0 *.*        *.*      
udp    0    0 *.*        *.*      
udp    0    0 *.*        *.*      
udp    0    0 *.snmp     *.*      
udp    0    0 *.2049     *.*      
udp    0    0 *.*        *.*      
udp    0    0 *.1031     *.*      
udp    0    0 *.111      *.*      
udp    0    0 *.1027     *.*      
udp    0    0 *.route    *.*      
udp    0    0 *.*        *.*      
udp    0    0 *.syslog   *.*      
boole.INPRES.EPL.PROV-LIEGE.BE>

```

et sur un PC Windows en réseau local :

C:\>**netstat -a**

Connexions actives

Proto	Adresse locale	Adresse distante	Etat
TCP	claude:epmap	0.0.0.0:0	LISTENING
TCP	claude:microsoft-ds	0.0.0.0:0	LISTENING
TCP	claude:1024	0.0.0.0:0	LISTENING
TCP	claude:1025	0.0.0.0:0	LISTENING
TCP	claude:1035	0.0.0.0:0	LISTENING
TCP	claude:3006	0.0.0.0:0	LISTENING
TCP	claude:3039	0.0.0.0:0	LISTENING
TCP	claude:3081	0.0.0.0:0	LISTENING
TCP	claude:5000	0.0.0.0:0	LISTENING
TCP	claude:1024	localhost:3006	ESTABLISHED
TCP	claude:3001	0.0.0.0:0	LISTENING
TCP	claude:3002	0.0.0.0:0	LISTENING
TCP	claude:3003	0.0.0.0:0	LISTENING
TCP	claude:3006	localhost:1024	ESTABLISHED
TCP	claude:3009	0.0.0.0:0	LISTENING
TCP	claude:3080	0.0.0.0:0	LISTENING
TCP	claude:3080	localhost:3081	ESTABLISHED
TCP	claude:3081	localhost:3080	ESTABLISHED
TCP	claude:5180	0.0.0.0:0	LISTENING

```
TCP  claude:netbios-ssn  0.0.0.0:0      LISTENING
UDP  claude:epmap        *:*
UDP  claude:snmp         *:*
UDP  claude:microsoft-ds  *:*
UDP  claude:isakmp       *:*
UDP  claude:1030          *:*
UDP  claude:1034          *:*
UDP  claude:3017          *:*
UDP  claude:ntp           *:*
UDP  claude:1900          *:*
UDP  claude:2234          *:*
UDP  claude:3082          *:*
UDP  claude:ntp           *:*
UDP  claude:netbios-ns    *:*
UDP  claude:netbios-dgm   *:*
UDP  claude:1900          *:*
```

C:\>

Dans le cas des adresses, on constate que la commande peut utiliser le joker (\*) pour signifier une écoute sur une adresse quelconque et un port fixé (\*.1023), l'inverse (10.10.20.3.\*) ou même les deux (\*.\*). Il n'y a évidemment pas d'état pour udp.

### c) Les informations sur un protocole donné

Il suffit de spécifier le protocole visé dans l'option -p :

```
netstat -p <nom du protocole>
```

Par exemple, pour ip sur boole :

```
boole.INPRES.EPL.PROV-LIEGE.BE> /usr/sbin/netstat -p ip
ip:
```

```
1096623 total packets received
0 bad header checksums
0 with size smaller than minimum
0 with data size < data length
0 with header length < data size
0 with data length < header length
0 fragments received
0 fragments dropped (dup or out of space)
0 fragments dropped after timeout
0 packets forwarded
1405 packets not forwardable
0 packets denied access
0 redirects sent
0 packets with unknown or unsupported protocol
1095218 packets consumed here
760786 total packets generated here
2 lost packets due to resource problems
```

```
0 total packets reassembled ok
0 output packets fragmented ok
0 output fragments created
0 packets with special flags set
boole.INPRES.EPL.PROV-LIEGE.BE>
```

Pour tcp :

```
boole.INPRES.EPL.PROV-LIEGE.BE> /usr/sbin/netstat -p tcp
tcp:
    172100 packets sent
        142750 data packets (20238985 bytes)
        1662 data packets (1703100 bytes) retransmitted
        17557 ack-only packets (9096 delayed)
        300 URG only packets
        90 window probe packets
        4056 window update packets
        5685 control packets
    233818 packets received
        138574 acks (for 20270195 bytes)
        4342 duplicate acks
        0 acks for unsent data
        130908 packets (13453168 bytes) received in-sequence
        180 completely duplicate packets (174928 bytes)
        38 packets with some dup. data (10167 bytes duped)
        3284 out-of-order packets (2279208 bytes)
        0 packets (0 bytes) of data after window
        0 window probes
        513 window update packets
        0 packets received after close
        19 discarded for bad checksums
        0 discarded for bad header offset fields
        0 discarded because packet too short
    2626 connection requests
    821 connection accepts
    3433 connections established (including accepts)
    4089 connections closed (including 92 drops)
    526 embryonic connections dropped
    139337 segments updated rtt (of 140670 attempts)
    971 retransmit timeouts
        1 connection dropped by rexmit timeout
    10 persist timeouts
    7 keepalive timeouts
        7 keepalive probes sent
        0 connections dropped by keepalive
boole.INPRES.EPL.PROV-LIEGE.BE>
```

Sur sunray, l'option **-P** (majuscule) permet simplement de filtrer les lignes de netstat en ne conservant que celles qui correspondent au protocole spécifié.

---

d) **Le contenu de la table de routage**

Combinée avec l'option –n (parce que certaines informations concernent des réseaux et pas des machines), l'option –r permet de consulter la table de routage utilisée par la machine :

**netstat –rn**

boole.INPRES.EPL.PROV-LIEGE.BE> **/usr/sbin/netstat -rn**

Routing tables

Destination	Gateway	Flags	Refs	Use	Interface	Netmasks:
Inet	255.255.254.0					

Route Tree for Protocol Family 2:

default	10.59.4.254	UGS	1	175793	tu0
10.59.4/23	10.59.4.1	U	3	425427	tu0
127.0.0.1	127.0.0.1	UH	3	159614	lo0

boole.INPRES.EPL.PROV-LIEGE.BE>

On obtient tout d'abord le masque de sous-réseau : 255.255.254.0. Ensuite viennent des informations sous forme de tableau :

colonne	information
1 : Destination	adresse ou groupe d'adresses
2 : Gateway	route à utiliser
3 : Flags	U=Up (la route fonctionne) – G=Gateway (routeur – si ce flag est absent, il s'agit d'une connexion directe) – H=Host (l'adresse de routage est en fait un ordinateur - si ce flag est absent, il s'agit d'un réseau ou sous-réseau) – S=Static – D=Dynamic (route créée par une redirection) – M=Modified (route modifiée par une redirection) – R=Reject.
4 : Refs	nombre de connexions actives sur cette route (cela n'a de sens que pour un protocole orienté connexion)
5 : Use	nombre de paquets ayant transité sur cette route
6 : Interface	lo0 représente toujours l'adresse de loopback (le flag correspondant est donc toujours un H) – tu0 est un interface Ethernet – fta0 est un interface FDDI – sl0 est un interface slip – ppp0 est un interface point à point.

Sur une machine Windows, en étant connecté à distance sur boole et zeus, cela donnerait :

Claude : Win> **netstat -rn**

Table de routage

Itinéraires actifs :

Adresse réseau	Masque réseau	Adr. passerelle	Adr. interface	Métrique
0.0.0.0	0.0.0.0	10.10.10.12	10.10.10.12	1
10.0.0.0	255.0.0.0	10.10.10.12	10.10.10.12	1
10.10.10.12	255.255.255.255	127.0.0.1	127.0.0.1	1
10.255.255.255	255.255.255.255	10.10.10.12	10.10.10.12	1
127.0.0.0	255.0.0.0	127.0.0.1	127.0.0.1	1
224.0.0.0	224.0.0.0	10.10.10.12	10.10.10.12	1
255.255.255.255	255.255.255.255	10.10.10.12	10.10.10.12	1

#### Connexions actives

Proto	Adresse locale	Adresse distante	État
TCP	10.10.10.12:1044	10.59.4.1:23	ESTABLISHED
TCP	10.10.10.12:1048	10.10.0.100:23	ESTABLISHED

Claude : Win>

#### e) L'état des interfaces

On peut obtenir des informations sur les interfaces évoqués ci-dessus par :

**netstat -i**

On obtient sur boole :

boole.INPRES.EPL.PROV-LIEGE.BE> **/usr/sbin/netstat -i**

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
tu0	1500	<Link>	00:00:f8:1b:37:d2	1022764	0	378038	662	1898
tu0	1500	DLI	none	1022764	0	378038	662	1898
tu0	1500	10.59.4	boole.inpres.epl.p	1022764	0	378038	662	1898
s10*	296	<Link>		0	0	0	0	0
lo0	4096	<Link>		427977	0	427977	0	0
lo0	4096	loop	localhost	427977	0	427977	0	0
ppp0*	1500	<Link>		0	0	0	0	0

boole.INPRES.EPL.PROV-LIEGE.BE>

On interprète ce tableau comme suit :

colonne	information
1 : Name	interface (voir ci-dessus)
2 : Mtu	Maximum Transmission Unit
3 : Network	réseau
4 : Address	adresse du réseau
5 : Ipkts	nombre de paquets reçus
6 : Ierrs	nombre de paquets reçus avec erreur
7 : Opkts	nombre de paquets
8 : Oerrs	nombre de paquets envoyés avec erreur
9 : Coll	nombre de collisions

#### Remarque

L'option –i n'existe pas sous Windows : on obtient à la place le mode d'emploi !

#### f) Les statistiques par protocole

Sur sunray, on peut obtenir des informations sur l'utilisation des protocoles courants par :

**netstat -s**

```
bash-2.03$ netstat -s
```

### RAWIP

```
rawipInDatagrams = 11 rawipInErrors = 0  
rawipInCksumErrs = 0 rawipOutDatagrams = 10  
rawipOutErrors = 0
```

### UDP

```
udpInDatagrams =5382207 udpInErrors = 0  
udpOutDatagrams =2005252 udpOutErrors = 0
```

### TCP

```
tcpRtoAlgorithm = 4 tcpRtoMin = 400  
tcpRtoMax = 60000 tcpMaxConn = -1  
tcpActiveOpens = 10698 tcpPassiveOpens = 3925  
tcpAttemptFails = 2078 tcpEstabResets = 110
```

...

```
tcpOutAck =753512 tcpOutAckDelayed =360945  
tcpOutUrg = 0 tcpOutWinUpdate = 1385
```

...

```
tcpInUnorderSegs = 95 tcpInUnorderBytes =117595  
tcpInDupSegs = 2774 tcpInDupBytes =575925
```

...

```
tcpListenDrop = 0 tcpListenDropQ0 = 0  
tcpHalfOpenDrop = 0 tcpOutSackRetrans = 40339
```

### IPv4

```
ipForwarding = 2 ipDefaultTTL = 255  
ipInReceives =9987277 ipInHdrErrors = 5  
ipInAddrErrors = 0 ipInCksumErrs = 0
```

...

```
ipFragOKs = 0 ipFragFails = 0  
ipFragCreates = 0 ipRoutingDiscards = 0  
tcpInErrs = 7 udpNoPorts = 51526  
udpInCksumErrs = 0 udpInOverflows = 0  
rawipInOverflows = 0 ipsecInSucceeded = 0  
ipsecInFailed = 0 ipInIPv6 = 0  
ipOutIPv6 = 0 ipOutSwitchIPv6 = 5900
```

### IPv6

```
ipv6Forwarding = 2 ipv6DefaultHopLimit = 255  
ipv6InReceives = 0 ipv6InHdrErrors = 0
```

...

### ICMPv4

```
icmpInMsgs = 392 icmpInErrors = 0  
icmpInCksumErrs = 0 icmpInUnknowns = 0  
icmpInDestUnreachs = 346 icmpInTimeExcds = 0
```

...

### ICMPv6

```
icmp6InMsgs = 0 icmp6InErrors = 0  
icmp6InDestUnreachs = 0 icmp6InAdminProhibs = 0
```

...

**IGMP:**

```
5 messages received
0 messages received with too few bytes
0 messages received with bad checksum
0 membership queries received
0 membership queries received with invalid field(s)
4 membership reports received
0 membership reports received with invalid field(s)
4 membership reports received for groups to which we belong
5 membership reports sent
```

```
bash-2.03$
```

### **3. Une commande de lecture de configuration : ipconfig**

La littérature décrit de ce type de commande, **ipconfig** sous windows et **ifconfig** sous Unix, comme permettant d'afficher la configuration du stack IP de la machine hôte. En plus clair, disons qu'elle fournit tous les renseignements concernant la machine hôte du point de vue réseau, qu'il s'agisse de la couche physique (adresse MAC), réseau (adresse IP) ou encore de références DNS.

a) **sous Windows**

La commande est donc

```
ipconfig
```

Dans une fenêtre DOS d'un PC classique doté d'une carte réseau LAN et d'une carte d'accès à distance, cela peut donner, avec le commutateur /all :

```
C:\VILVENS>ipconfig /all
```

Configuration IP de Windows 98

```
Nom d'hôte ..... : CLAUDE
Serveur DNS. .... : 195.238.2.21
                           195.238.2.22
Type de noeud. .... : Diffuser
ID étendue NetBIOS .... :
Routage IP activé. .... : Non
Proxy WINS activé. .... : Non
Résolution NetBIOS par DNS : Non
```

0 - Carte Ethernet :

```
Description ..... : PPP Adapter.
Adresse physique. .... : 47-45-33-59-00-00
DHCP activé ..... : Oui
Adresse IP. .... : 195.238.12.168
Masque de sous-réseau . . . : 255.255.255.0
Passerelle par défaut . . . : 195.238.12.168
Serveur DHCP. .... : 255.255.255.255
```

Serveur WINS principal . . . :  
Serveur WINS secondaire . . . :  
Bail obtenu . . . . . : 01 01 80 0:00:00  
Bail expirant . . . . . : 01 01 80 0:00:00

1 - Carte Ethernet :

Description . . . . . : **ELNK3 Ethernet Adapter**  
Adresse physique. . . . . : 00-60-08-4D-ED-DE  
DHCP activé . . . . . : Non  
Adresse IP. . . . . : 192.168.2.2  
Masque de sous-réseau . . . . . : 255.255.255.0  
Passerelle par défaut . . . . . :  
Serveur WINS principal . . . . . :  
Serveur WINS secondaire . . . . . :  
Bail obtenu . . . . . :  
Bail expirant . . . . . :

La commande ipconfig possède d'autres options, comme :

/release N	libère l'adresse IP attribuée par DHCP à la carte réseau N.
/renew N	renouvelle l'adresse IP attribuée par DHCP à la carte réseau N.

b) **sous Unix**

La commande

**ifconfig**

réalise un travail analogue, avec le commutateur -a. Ainsi, sur la machine Sunray2 :

```
bash-2.03$ /usr/sbin/ifconfig -a
lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1
    inet 127.0.0.1 netmask ff000000
ce0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2
    inet 10.59.4.5 netmask fffffe00 broadcast 10.59.5.255
bash-2.03$
```

Pour Copernic :

```
19 /info/prof/vilvens/tcp# /usr/sbin/ifconfig -a
lo0: flags=100c89<UP,LOOPBACK,NOARP,MULTICAST,SIMPLEX,NOCHECKSUM>
    inet 127.0.0.1 netmask ff000000 ipmtu 4096

sl0: flags=10<POINTOPOINT>

tu0: flags=c63<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST,SIMPLEX>
    inet 10.59.5.9 netmask fffffe00 broadcast 10.59.5.255 ipmtu 1500

20 /info/prof/vilvens/tcp#
```

---

et Indochine :

```
vilvens@indochine:~$ ifconfig -a
dummy0  Link encap:Ethernet HWaddr 92:87:C3:19:8E:7D
        BROADCAST NOARP MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

eth0    Link encap:Ethernet HWaddr 00:0B:CD:C9:CA:04
        inet addr:10.59.5.3 Bcast:10.59.5.255 Mask:255.255.254.0
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:118563648 errors:42 dropped:0 overruns:0 frame:27
        TX packets:105858888 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:978752386 (933.4 MiB) TX bytes:3372041479 (3.1 GiB)
        Interrupt:28

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:6910880 errors:0 dropped:0 overruns:0 frame:0
        TX packets:6910880 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:2162241114 (2.0 GiB) TX bytes:2162241114 (2.0 GiB)

vilvens@indochine:~$
```

#### 4. Une commande d'interrogation des DNS : nslookup

Il s'agit ici d'une commande fort utile pour résoudre les problèmes de résolutions de noms selon le système de DNS. La commande

**nslookup**

est en fait une application interactive qui affiche à son démarrage le nom et l'adresse IP du serveur DNS configuré pour le système local puis une invite de commande. Il n'y a plus alors qu'à entrer les noms des machines recherchées : ces noms seront recherchés par le serveur DNS par défaut, à moins qu'on en définisse un autre au moyen de la commande :

**>server <nom\_autre\_serveur\_DNS>**

Ce serveur restera alors celui pris en considération. Le suffixe DNS utilisé pour rechercher une machine est celui de l'ordinateur local. En cas d'échec, le suffixe est dégradé d'un niveau (donc, on passe de `inpres.prov-liege.be` à `prov-liege.be`) et la requête est renouvelée. Précisons encore que la commande

**>set debug**

permet de faire exécuter les recherches de nslookup avec une visualisation de tous les détails.  
On quitte nslookup par la commande **exit**.

a) **sous Windows**

Cela peut donner :

```
C:\Documents and Settings\vilvens>nslookup
Serveur par défaut : neptune.thalassa.inpres.prov-liege.be
Address: 10.59.4.2
```

> **indochine**

```
Serveur: neptune.thalassa.inpres.prov-liege.be
Address: 10.59.4.2
```

```
Nom : indochine.inpres.epl.prov-liege.be
Address: 10.59.5.3
```

> **10.59.5.6**

```
Serveur : neptune.thalassa.inpres.prov-liege.be
Address: 10.59.4.2
```

```
Nom : sysiphe.thalassa.inpres.prov-liege.be
Address: 10.59.5.6
```

> **set debug**

> **copernic**

```
Serveur : neptune.thalassa.inpres.prov-liege.be
Address: 10.59.4.2
```

-----  
Got answer:

HEADER:

```
opcode = QUERY, id = 3, rcode = NOERROR
header flags: response, auth. answer, want recursion, recursion avail.
questions = 1, answers = 1, authority records = 0, additional = 0
```

QUESTIONS:

```
copernic.inpres.epl.prov-liege.be, type = A, class = IN
```

ANSWERS:

```
-> copernic.inpres.epl.prov-liege.be
internet address = 10.59.5.9
ttl = 867600 (10 days 1 hour)
```

-----  
Nom : copernic.inpres.epl.prov-liege.be
Address: 10.59.5.9

> **u2**

```
Serveur : neptune.thalassa.inpres.prov-liege.be
Address: 10.59.4.2
```

---

-----  
Got answer:

HEADER:

opcode = QUERY, id = 4, rcode = NXDOMAIN  
header flags: response, auth. answer, want recursion, recursion avail.  
questions = 1, answers = 0, authority records = 1, additional = 0

QUESTIONS:

u2.inpres.epl.prov-liege.be, type = A, class = IN

AUTHORITY RECORDS:

-> inpres.epl.prov-liege.be

ttl = 3600 (1 hour)

primary name server = neptune.thalassa.inpres.prov-liege.be

responsible mail addr = admin.thalassa.inpres.prov-liege.be

serial = 462

refresh = 900 (15 mins)

retry = 600 (10 mins)

expire = 86400 (1 day)

default TTL = 3600 (1 hour)

-----  
-----

Got answer:

HEADER:

opcode = QUERY, id = 5, rcode = NOERROR

header flags: response, auth. answer, want recursion, recursion avail.

questions = 1, answers = 1, authority records = 0, additional = 0

QUESTIONS:

u2.wildness.loc, type = A, class = IN

ANSWERS:

-> ***u2.wildness.loc***

***internet address = 10.59.5.219***

ttl = 867600 (10 days 1 hour)

-----

Nom : u2.wildness.loc

Address: 10.59.5.219

>exit

b) **sous Unix**

Dans le même genre, par exemple sur sunray2

bash-2.03\$ **/usr/sbin/nslookup**

Default Server: neptune.thalassa.inpres.prov-liege.be

Address: 10.59.4.2

> **ulysse**

Server: neptune.thalassa.inpres.prov-liege.be

Address: 10.59.4.2

---

Name: ulysse.inpres.epl.prov-liege.be  
Address: 10.59.5.13

> **u2**

Server: neptune.thalassa.inpres.prov-liege.be  
Address: 10.59.4.2

Name: u2.wildness.loc  
Address: 10.59.5.219

> **indochine**

Server: neptune.thalassa.inpres.prov-liege.be  
Address: 10.59.4.2

Name: indochine.inpres.epl.prov-liege.be  
Address: 10.59.5.3

> **machiavel**

Server: neptune.thalassa.inpres.prov-liege.be  
Address: 10.59.4.2

\*\*\* neptune.thalassa.inpres.prov-liege.be can't find machiavel: Server failed

> **sysiphe**

Server: neptune.thalassa.inpres.prov-liege.be  
Address: 10.59.4.2

Name: sysiphe.thalassa.inpres.prov-liege.be  
Address: 10.59.5.6

>exit



Historiquement, les sockets doivent leur existence au monde UNIX.  
Nous allons nous en souvenir en traitant des primitives de consultation des fichiers de configuration d'une machine UNIX travaillant en TCP/IP, fichiers que l'on retrouve dans le monde Windows ...

## XI. Les fichiers de configuration TCP/IP



*L'homme a créé les dieux, l'inverse reste à prouver.*

(S. Gainsbourg)

Les renseignements concernant la configuration TCP/IP d'une machine constituent une base de données au sens large du terme. Les fonctions qui permettent d'obtenir ces renseignements sont, pour cette raison, prototypés dans un header appelé très justement netdb.h. Voyons cela plus en détail.

### 1. Les fonctions de consultation machines-адresses

D'un point de vue programmatique, on peut considérer qu'une fonction comme **gethostbyname()** est en fait un appel au solveur. Nous l'avons déjà utilisée afin de réaliser l'appel de la primitive bind(). Mais il existe en fait un certain nombre de fonctions de consultations apparentées.

#### 1.1 La recherche par adresse

On peut obtenir des informations sur une machine dont on connaît l'adresse IP par :

```
struct hostent *gethostbyaddr ( <l'adresse réseau – const void *>,
                                <longueur de la zone contenant l'adresse – size_t>,
                                <domaine – int>);
```

Cette fonction, prototypée dans netdb.h, fournit l'adresse d'une structure hostent qui, pour rappel, est définie par :

#### structure hostent

```
struct hostent
{
    char  *h_name;      /* official name of host */
    char  **h_aliases;  /* alias list */
    int   h_addrtype;   /* host address type */
    int   h_length;     /* length of address */
    char  **h_addr_list; /* list of addresses from name server */
#define h_addr h_addr_list[0]    /* address, for backward compatibility */
};
```

La fonction retourne NULL si l'adresse ne correspond pas à une machine connue.

## 1.2 Le nom de la machine locale

Comme une sorte de cas particulier de gethostbyname(), on peut aussi utiliser :

```
int gethostname (char *name, size_t namelen);
```

### CONFIG01.C

```
/* CONFIG01.C
- Claude Vilvens -
*/
...
#include <netdb.h>      /* pour la structure hostent */
...
int main()
{
    char nomMachine[40], adresseMachine[16];
    int adresseReseau;
    struct hostent * infosMachine;
    struct in_addr adresseIP; /* Adresse Internet au format reseau */

/* 1. Acquisition du nom de la machine locale */
    if (gethostname(nomMachine, sizeof(nomMachine)) == -1)
    {
        printf("Erreur d'acquisition d'infos sur le host local %d\n", errno);
    }
    else
    {
        printf("Acquisition infos host local OK\n");
        printf("Nom de la machine = %s\n", nomMachine);
    }

/* 2. Acquisition des informations sur un ordinateur de nom donne */
    do
    {
        printf("Nom de l'ordinateur cherche (FIN) : "); gets(nomMachine);
        if (strcmp(nomMachine, "FIN") == 0) break;          /* Denys-like */
        if ((infosMachine = gethostbyname(nomMachine)) == 0)
        {
            printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        }
        else
        {
            printf("Acquisition infos host OK\n");
            memcpy(&adresseIP, infosMachine->h_addr, infosMachine->h_length);
            printf("Adresse IP = %s\n", inet_ntoa(adresseIP));
            printf("Type de l'adresse = %d\n", infosMachine->h_addrtype);
        }
    }
    while (1);
```

```

/* 3. Acquisition des informations sur un ordinateur d'adresse donnee */
do
{
    printf("Adresse de l'ordinateur cherche (0) : ");
    gets(adresseMachine);
    if (strcmp(adresseMachine, "0") == 0) break;
    adresseReseau = inet_addr(adresseMachine);
    if ((infosMachine = gethostbyaddr(&adresseReseau,
                                      sizeof(adresseReseau), AF_INET)) == 0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    }
    else
    {
        printf("Acquisition infos host OK\n");
        printf("Nom = %s\n", infosMachine->h_name);
        printf("Type de l'adresse = %d\n", infosMachine->h_addrtype);
    }
}
while (1);

return 0;
}

```

Sur boole, cela donne (il y longremps) :

```

boole> s
Acquisition infos host local OK
Nom de la machine = boole.INPRES.EPL.PROV-LIEGE.BE
Nom de l'ordinateur cherche (FIN) : zeus
Acquisition infos host OK
Adresse IP = 10.10.0.100
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : dec01
Acquisition infos host OK
Adresse IP = 10.59.4.131
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : boole
Acquisition infos host OK
Adresse IP = 10.59.4.1
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : zues
Erreur d'acquisition d'infos sur le host 1
Nom de l'ordinateur cherche (FIN) : FIN
Adresse de l'ordinateur cherche (0) : 10.59.4.1
Acquisition infos host OK
Nom = boole.INPRES.EPL.PROV-LIEGE.BE
Type de l'adresse = 2
Adresse de l'ordinateur cherche (0) : 10.10.0.100
Acquisition infos host OK

```

```
Nom = zeus
Type de l'adresse = 2
Adresse de l'ordinateur cherche (0) : 10.59.4.131
Acquisition infos host OK
Nom = dec01
Type de l'adresse = 2
Adresse de l'ordinateur cherche (0) : 10.10.20.0
Erreur d'acquisition d'infos sur le host 1
Adresse de l'ordinateur cherche (0) : 0
boole.INPRES.EPL.PROV-LIEGE.BE>
```

tandis que sur dec01 :

```
% c
Acquisition infos host local OK
Nom de la machine = dec01
Nom de l'ordinateur cherche (FIN) : zeus
Acquisition infos host OK
Adresse IP = 10.10.0.100
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : boole
Acquisition infos host OK
Adresse IP = 10.59.4.1
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : osfI
Acquisition infos host OK
Adresse IP = 10.59.4.135
Type de l'adresse = 2
Nom de l'ordinateur cherche (FIN) : zues
Erreur d'acquisition d'infos sur le host 2
Nom de l'ordinateur cherche (FIN) : FIN
Adresse de l'ordinateur cherche (0) : 10.59.4.1
Acquisition infos host OK
Nom = boole
Type de l'adresse = 2
Adresse de l'ordinateur cherche (0) : 10.10.0.100
Acquisition infos host OK
Nom = zeus.inpres.epl.prov-liege.be
Type de l'adresse = 2
Adresse de l'ordinateur cherche (0) : 0
%
```

## 2. Le fichier /etc/hosts

### 2.1 Sous UNIX

Sur les machines UNIX, le fichier **/etc/hosts** contient une liste d'adresses avec le nom des machines correspondantes. Ainsi, par exemple, sur sunray :

```
/etc/hosts (monde Unix)
#
# Internet host table
#
127.0.0.1      localhost
10.59.4.5      sunray2v440.inpres.epl.prov-liege.be  sunray2v440  loghost
10.59.4.6      sunray1v440
10.59.4.4      mgmtsb150
#
10.59.5.9      copernic    # serveur Compaq
10.59.4.1      diderot     # serveur Compaq(pour test)
10.59.4.8      machiavel   # serveur Compaq(pour ldap)
```

Les fonctions évoquées ci-dessus parcourront évidemment ce fichier (*avant d'ailleurs de s'adresser au DNS pour ce qui n'a pas été trouvé*). Il existe d'autres fonctions (header netdb.h) pour parcourir ce même fichier de manière exhaustive :

```
void sethostent( <flag indiquant si le fichier doit rester ouvert – int>);
struct hostent *gethostent();
void endhostent();
```

La fonction sethostent() ouvre le fichier au début, en le laissant éventuellement ouvert si le flag qui lui sert d'argument est non nul. La fonction gethostent() lit la ligne suivante, en ouvrant le fichier si nécessaire, puis se positionne sur la ligne suivante. Et, bien sûr, endhostent() ferme le fichier. Illustration sur boole :

### CONFIG02.C

```
/* CONFIG02.C
- Claude Vilvens -
*/
#include <netdb.h>
...
int main()
{
    char nomMachine[40];
    int adresseReseau;
    struct hostent * infosMachine;
    struct in_addr adresseIP;

    if (sethostent(0)==NULL) puts("Ouverture de /etc/hosts impossible");
    else
    {
        while ( (infosMachine = gethostent()) != NULL)
        {
```

```

        printf("Nom = %s\n",infosMachine->h_name);
        memcpy(&adresseIP, infosMachine->h_addr, infosMachine->h_length);
        printf("Adresse IP = %s\n",inet_ntoa(adresseIP));
        printf("Type de l'adresse = %d\n", infosMachine->h_addrtype);
    }
endhostent();
}
return 0;
}

```

L'exécution de ce programme donnait, sur la machine Unix Boole (il y a longtemps ;-)) :

```

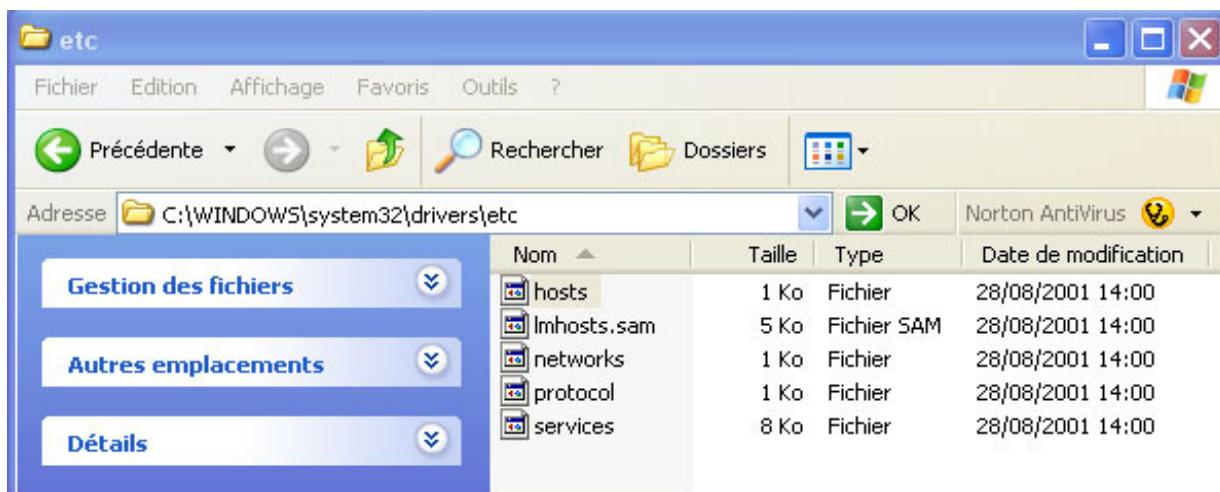
boole> c
Nom = localhost
Adresse IP = 127.0.0.1
Type de l'adresse = 2
Nom = zeus
Adresse IP = 10.10.0.100
Type de l'adresse = 2
Nom = dec01
Adresse IP = 10.59.4.131
Type de l'adresse = 2
Nom = boole.INPRES.EPL.PROV-LIEGE.BE
Adresse IP = 10.59.4.1
Type de l'adresse = 2
boole.INPRES.EPL.PROV-LIEGE.BE>

```

Comme nous allons le voir, l'histoire se répète pour les noms des sous-réseaux, les services et les protocoles ...

## 2.2 Sous Windows

Sur les machines Windows98, 2000, XP et 7, on trouve un répertoire **WINDOWS\system32\drivers\etc** qui contient le fichier **hosts** équivalent de celui d'UNIX. Il en est d'ailleurs de même pour les autres fichiers de configuration dont il sera question dans les paragraphes suivants :



Ainsi, par exemple, sur un banal PC Windows, on peut trouver :

```
WINDOWS\system32\drivers\etc\hosts (monde Windows)
# Copyright (c) 1993-1999 Microsoft Corp.
#
# Ceci est un exemple de fichier HOSTS utilisé par Microsoft TCP/IP
# pour Windows.
#
# Ce fichier contient les correspondances des adresses IP aux noms d'hôtes.
# Chaque entrée doit être sur une ligne propre. L'adresse IP doit être placée
# dans la première colonne, suivie par le nom d'hôte correspondant. L'adresse
# IP et le nom d'hôte doivent être séparés par au moins un espace.
#
# De plus, des commentaires (tels que celui-ci) peuvent être insérés sur des
# lignes propres ou après le nom d'ordinateur. Ils sont indiqué par le
# symbole '#'.
#
# Par exemple :
#
#   102.54.94.97  rhino.acme.com      # serveur source
#   38.25.63.10    x.acme.com          # hôte client x

127.0.0.1      localhost
127.0.0.1      serveurParDefautLocal
192.168.2.2    serveurParDefaut
192.168.2.1    clientParDefaut
```

Indépendamment de tout DNS, on peut donc commander avec succès :

```
C:\>ping serveurParDefaut
```

Envoi d'une requête 'ping' sur serveurParDefaut [192.168.2.2] avec 32 octets de données :

```
Réponse de 192.168.2.2 : octets=32 temps<1ms TTL=128
```

Statistiques Ping pour 192.168.2.2:

Paquets : envoyés = 4, reçus = 4, perdus = 0 (perte 0%),

Durée approximative des boucles en millisecondes :

Minimum = 0ms, Maximum = 0ms, Moyenne = 0ms

### 3. Le fichier /etc/networks

#### 3.1 Sous UNIX

Ce fichier, cousin de /etc/hosts, contient les composantes adresses réseaux avec les noms symboliques qui leur sont associés. Sur sunray, son contenu est le suivant (c'est le strict minimum) :

<b>/etc/networks (monde Unix)</b>
#ident "@(#)"networks 1.4 92/07/14 SMI" /* SVr4.0 1.1 */ # # The networks file associates Internet Protocol (IP) network numbers # with network names. The format of this file is: # # network-name network-number nicnames ... # # # The loopback network is used only for intra-machine communication # loopback 127  # # Internet networks # arpanet 10 arpa # Historical

Tout comme /etc/hosts, ce fichier n'est pas manipulé directement (sauf, évidemment, par l'administrateur réseau) : les fonctions du paragraphe suivant permettent de le manipuler.

#### 3.2 Sous Windows

L'équivalent sur une machine Windows XP est du genre :

<b>WINDOWS\system32\drivers\etc\networks (monde Windows)</b>
# Copyright (c) 1993-1999 Microsoft Corp. # # Ce fichier contient les correspondances des noms et des num,ros de r,seau # pour les r,seaux locaux. Les num,ros de r,seau sont reconnus en forme # d,cimale s,par,e par des points. # # Format: # # <nom réseau> <numéro réseau> [alias...] [#<commentaires>] # # Par exemple: # # loopback 127 # campus 284.122.107 # londres 284.122.108  loopback 127

#### 4. Les fonctions de consultation réseaux-adresses

Par analogie, il n'est guère difficile de saisir le rôle des fonctions suivantes :

```
struct netent *getnetbyname (<nom du réseau - const char *>);
struct netent *getnetbyaddr (      <adresse du réseau - in_addr_t>,
                                <domaine - int>);
void setnetent (<flag indiquant si le fichier doit rester ouvert - int>);
struct netent *getnetent ();
void endnetent ();
```

où la structure netent cousine de hostent est :

##### structure netent

```
struct netent
{
    char *n_name;      /* official name of net */
    char **n_aliases;  /* alias list */
    int n_addrtype;   /* net address type */
    in_addr_t n_net;   /* Replaced unsigned int type to      */
                        /* typedef as specified by Spec1170. */
                        /* in_addr_t defined as unsigned int */
                        /* in netinet/in.h           */
};
```

Les deux programmes suivants, tout à fait similaires aux deux programmes concernant le fichier etc/hosts, illustrent ces fonctions :

##### CONFIG03.C

```
/* CONFIG03.C
- Claude Vilvens -
*/
...
int main()
{
    char nomReseau[40], adresseReseau[16];
    struct netent *infosReseau;

/* 1. Acquisition des informations sur un reseau de nom donne */
    do
    {
        printf("Nom du reseau cherche (FIN) : "); gets(nomReseau);
        if (strcmp(nomReseau, "FIN") == 0) break;
        if ((infosReseau = getnetbyname(nomReseau)) == 0)
        {
            puts("Nom de reseau inconnu");
            printf("Erreur d'acquisition d'infos sur le reseau %d\n", errno);
        }
        else
        {
            printf("Acquisition infos reseau OK\n");
        }
    }
}
```

```

        printf("Reseau %s trouve !\n", infosReseau->n_name);
        printf("Numero de reseau = %s\n" ,inet_ntoa(ntohs(infosReseau->n_net)));
        printf("Type de l'adresse = %d\n", infosReseau->n_addrtype);
    }
}
while (1);

/* 2. Acquisition des informations sur un reseau d'adresse donnee */
do
{
    printf("Adresse du reseau cherche (0) : "); gets(adresseReseau);
    if (strcmp(adresseReseau, "0") == 0) break;
    if ( (infosReseau=getnetbyaddr(ntohs(inet_addr(adresseReseau)),AF_INET)) == 0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    }
    else
    {
        printf("Acquisition infos host OK\n");
        printf("Nom = %s\n",infosReseau->n_name);
    }
}
while (1);
return 0;
}

```

Ce premier programme donne, exécuté sur dec01 (là aussi, il y a longtemps) :

```

% conf
Nom du reseau cherche (FIN) : inpres.net
Acquisition infos reseau OK
Reseau inpres.net trouve !
Numero de reseau = 128.1.0.0
Type de l'adresse = 2
Nom du reseau cherche (FIN) : acp.net
Acquisition infos reseau OK
Reseau acp.net trouve !
Numero de reseau = 128.2.0.0
Type de l'adresse = 2
Nom du reseau cherche (FIN) : vilvens.net
Nom de reseau inconnu
Erreur d'acquisition d'infos sur le reseau 3
Nom du reseau cherche (FIN) : FIN
Adresse du reseau cherche (0) : 128.1
Erreur d'acquisition d'infos sur le host 3
Adresse du reseau cherche (0) : 128.1.0.0
Acquisition infos host OK
Nom = inpres.net
Adresse du reseau cherche (0) : 128.3.0.0
Acquisition infos host OK

```

```
Nom = dgacp.net
Adresse du reseau cherche (0) : 0
%
```

Ce deuxième programme réalise une boucle analogue à celle parcourant le fichier etc/hosts (config02.c) :

### CONFIG04.C

```
/* CONFIG04.C
- Claude Vilvens -
*/
...
int main()
{
    char nomReseau[40];
    struct netent * infosReseau;
    struct in_addr adresseIP; /* Adresse Internet au format reseau */

    if (setnetent(0)==NULL)
        puts("Ouverture de /etc/networks impossible");
    else
    {
        while ( (infosReseau = getnetent()) != NULL)
        {
            printf("Nom = %s\n",infosReseau->n_name);
            adresseIP.s_addr = infosReseau->n_net;
            printf("Adresse IP = %s\n",inet_ntoa(ntohs(adresseIP)));
            printf("Type de l'adresse = %d\n", infosReseau->n_addrtype);
        }
        endnetent();
    }

    return 0;
}
```

L'exécution sur dec01 donnait, à l'époque ;-) :

```
% conf
Nom = loop
Adresse IP = 0.127.0.0
Type de l'adresse = 2
Nom = inpres.net
Adresse IP = 128.1.0.0
Type de l'adresse = 2
Nom = dgacp.net
Adresse IP = 128.3.0.0
Type de l'adresse = 2
Nom = acp.net
Adresse IP = 128.2.0.0
```

```
Type de l'adresse = 2
Nom = eplnet.net
Adresse IP = 128.100.0.0
Type de l'adresse = 2
Nom = esej.net
Adresse IP = 128.200.0.0
Type de l'adresse = 2
%
```

## 5. Le fichier /etc/protocols

Ce fichier contient, quelle surprise, les informations sur les protocoles connus et utilisés par l'ordinateur local.

### 5.1 Sous UNIX

Ainsi, une machine Unix supporte un fichier dont voici quelques morceaux choisis :

etc/protocols (monde Unix)			
#ident	"@(#)protocols	1.5	99/03/21 SMI" /* SVr4.0 1.1 */
#			
#	# Internet (IP) protocols		
#			
ip	0	IP	# internet protocol, pseudo protocol num
ber			
icmp	1	ICMP	# internet control message protocol
ggp	3	GGP	# gateway-gateway protocol
tcp	6	TCP	# transmission control protocol
egp	8	EGP	# exterior gateway protocol
pup	12	PUP	# PARC universal packet protocol
udp	17	UDP	# user datagram protocol
hmp	20	HMP	# host monitoring protocol
xns-idp	22	XNS-IDP	# Xerox NS IDP
rdp	27	RDP	# "reliable datagram" protocol
#			
#	# Internet (IPv6) extension headers		
#			
hopopt	0	HOPOPT	# Hop-by-hop options for IPv6
ipv6	41	IPv6	# IPv6 in IP encapsulation
ipv6-route	43	IPv6-Route	# Routing header for IPv6

### 5.2 Sous Windows

Une machine Windows disposera d'un fichier du type suivant :

WINDOWS\system32\drivers\etc\protocols (monde Windows)	
# Copyright (c) 1993-1999 Microsoft Corp.	
#	
# Ce fichier contient les protocoles Internet tels qu'ils sont définis	

```
# dans le document officiel RFC 1700 (Assigned Numbers).
#
# Format:
#
# <nom de protocole> <num,ro assign,> [alias...] [#<commentaire>]

ip      0   IP    # Protocole Internet
icmp     1   ICMP   # Protocole Internet de contrôle de message
ggp      3   GGP    # Protocole passerelle-passerelle
tcp      6   TCP    # Protocole de contrôle de transmission
egp      8   EGP    # Protocole de passerelle externe
pup     12   PUP    # Protocole de paquet universel PARC
udp     17   UDP    # Protocole de datagramme utilisateur
hmp     20   HMP    # Protocole de surveillance d'hôte
xns-idp 22   XNS-IDP # IDP Xerox NS
rdp     27   RDP    # Protocole de "datagramme fiable"
rvd     66   RVD    # Disque virtuel distant MIT
```

## 6. Les fonctions de consultation des protocoles

Au risque de lasser (et ce n'est pas fini !), on retrouve ici des fonctions au pouvoir bien clair. Et, bien sûr, elles utilisent une structure :

<b>structure protoent</b>
struct protoent
{
char *p_name; /* official protocol name */
char **p_aliases; /* alias list */
int p_proto; /* protocol # */
};

Les primitives se déclinent comme suit :

struct protoent * <b>getprotobynumber</b> (<nom du protocole - const char *>);
struct protoent * <b>getprotobyname</b> (<numéro du protocole - int>);
void <b>setprotoent</b> (<flag indiquant si le fichier doit rester ouvert – int>);
struct protoent * <b>getprotoent</b> (void);
void <b>endprotoent</b> (void);

Les deux programmes suivants illustrent leur utilisation. Le premier passe en revue les protocoles disponibles :

### CONFIG05.C

/* CONFIG05.C
- Claude Vilvens -
*/
...
int main()
{

```

char nomProto[40], adresseMachine[16];
int adresseReseau;
struct protoent * infosProto;

if (setprotoent(0)==NULL) puts("Ouverture de /etc/protocols impossible");
else
{
    while ( (infosProto = getprotoent()) != NULL)
    {
        printf("Nom = %s\n",infosProto->p_name);
        printf("Protocole = %d\n", infosProto->p_proto);
    }
    endprotoent();
}

return 0;
}

```

Sur dec01, le résultat obtenu est :

```

% conf
Nom = ip
Protocole = 0
Nom = icmp
Protocole = 1
Nom = igmp
Protocole = 2
Nom = ggp
Protocole = 3
Nom = tcp
Protocole = 6
Nom = pup
Protocole = 12
Nom = udp
Protocole = 17
%

```

Le second programme permet les habituelles interrogations :

### **CONFIG06.C**

```

/* CONFIG06.C
- Claude Vilvens -
...
int main()
{
    char nomProtocole[40];
    int numProtocole;
    struct protoent * infosProtocole;
```

/\* 1. Acquisition des informations sur un protocole de nom donne \*/

```

do
{
    printf("Nom du protocole cherche (FIN) : "); gets(nomProtocole);
    if (strcmp(nomProtocole, "FIN") == 0) break;
    if ((infosProtocole = getprotobynumber(nomProtocole)) == 0)
    {
        puts("Nom de protocole inconnu");
        printf("Erreur d'acquisition d'infos sur le protocole %d\n", errno);
    }
    else
    {
        printf("Acquisition infos protocole OK\n");
        printf("Protocole %s trouve !\n", infosProtocole->p_name);
        printf("Numero de protocole = %d\n", infosProtocole->p_proto);
    }
}
while (1);

```

```

/* 2. Acquisition des informations sur un protocole de numéro donne */
do
{
    char buf[20];
    printf("Numero du protocole cherche (-1) : "); gets(buf);
    if (strcmp(buf, "-1") == 0) break;
    if ((infosProtocole = getprotobynumber(atoi(buf))) == 0)
    {
        printf("Erreur d'acquisition d'infos sur le protocole %d\n", errno);
    }
    else
    {
        printf("Acquisition infos protocole OK\n");
        printf("Nom = %s\n", infosProtocole->p_name);
    }
}
while (1);
return 0;
}

```

Un exemple d'exécution sur dec01 sera alors :

```

% conf
Nom du protocole cherche (FIN) : tcp
Acquisition infos protocole OK
Protocole tcp trouve !
Numero de protocole = 6
Nom du protocole cherche (FIN) : udp
Acquisition infos protocole OK
Protocole udp trouve !

```

Numero de protocole = 17  
Nom du protocole cherche (FIN) : **ppp**  
Nom de reseau inconnu  
Erreur d'acquisition d'infos sur le protocole 3  
Nom du protocole cherche (FIN) : FIN  
Numero du protocole cherche (-1) : **1**  
Acquisition infos protocole OK  
Nom = icmp  
Numero du protocole cherche (-1) : **0**  
Acquisition infos protocole OK  
Nom = ip  
Numero du protocole cherche (-1) : **12**  
Acquisition infos protocole OK  
Nom = pup  
Numero du protocole cherche (-1) : **129**  
Erreur d'acquisition d'infos sur le protocole 3  
Numero du protocole cherche (-1) : -1  
%

## 7. Le fichier /etc/services

### 7.1 Sous UNIX

Ce fichier contient la liste des services standards disponibles, avec le port et le protocole correspondant. Son contenu sur une machine Unix de type Sunray est du type :

/etc/services (monde Unix)		
#ident "@(#)"services 1.27 00/11/06 SMI"	/* SVr4.0 1.8 */	
#		
#		
# Copyright (c) 1999-2000 by Sun Microsystems, Inc.		
# All rights reserved.		
#		
# Network services, Internet style		
#		
tcpmux	1/tcp	
echo	7/tcp	
echo	7/udp	
discard	9/tcp	sink null
discard	9/udp	sink null
systat	11/tcp	users
daytime	13/tcp	
daytime	13/udp	
netstat	15/tcp	
chargen	19/tcp	ttytst source
chargen	19/udp	ttytst source
ftp-data	20/tcp	
ftp	21/tcp	
telnet	23/tcp	
smtp	25/tcp	mail

On peut constater, à la lecture de ce fichier, que l'administrateur peut ajouter de nouveaux services (à l'usage du démon inetd<sup>1</sup>). Ainsi, on a ajouté des services correspondants à des serveurs réalisés par des étudiants.

## 7.2 Sous Windows

Le fichier similaire existe sous Windows :

<b>WINDOWS\system32\drivers\etc\services (monde Windows)</b>			
# Copyright (c) 1993-1999 Microsoft Corp.			
#			
# Ce fichier contient les numéros de port des services les plus connus définis par IANA			
#			
# Format:			
#			
# <nom de service> <numéro de port/<protocole> [alias...] [#<commentaire>]			
#			
echo	7/tcp		
echo	7/udp		
discard	9/tcp	sink null	
discard	9/udp	sink null	
systat	11/tcp	users	#Utilisateurs actifs
systat	11/udp	users	#Utilisateurs actifs
daytime	13/tcp		
daytime	13/udp		
qotd	17/tcp	quote	#Citation du jour
qotd	17/udp	quote	#Citation du jour
chargen	19/tcp	ttytst source	#Générateur de caractères
chargen	19/udp	ttytst source	#Générateur de caractères
ftp-data	20/tcp		#FTP, données
ftp	21/tcp		#FTP. contrôle
telnet	23/tcp		
smtp	25/tcp	mail	#Format SMTP (Simple Mail Transfer Protocol)
time	37/tcp	timserver	
time	37/udp	timserver	
rlp	39/udp	resource	#Protocole d'emplacement des ressources
nameserver	42/tcp	name	#Serveur de nom d'hôte
nameserver	42/udp	name	#Serveur de nom d'hôte
nicname	43/tcp	whois	
domain	53/tcp		#Serveur de nom de domaine
domain	53/udp		#Serveur de nom de domaine
bootps	67/udp	dhcp	#Serveur de protocole d'amorçage
bootpc	68/udp	dhcp	#Serveur de protocole d'amorçage
tftp	69/udp		#Transfert de fichiers trivial
gopher	70/tcp		
finger	79/tcp		
http	80/tcp	www www-http	#World Wide Web
kerberos	88/tcp	krb5 kerberos-sec	#Kerberos

<sup>1</sup> il s'agit d'un démon qui réalise automatiquement les listen() et les accept() des services dont il a connaissance.

kerberos	88/udp	krb5	kerberos-sec	#Kerberos
hostname	101/tcp	hostnames		#Serveur de nom d'hôte NIC
iso-tsap	102/tcp			#ISO-TSAP Classe 0
rlogin	107/tcp			#Service Telnet distant
pop2	109/tcp	postoffice		#Protocole Bureau de poste - Version 2
pop3	110/tcp			#Protocole Bureau de poste - Version 3
sunrpc	111/tcp	rpcbind	portmap	#Appel de procédure distante SUN
sunrpc	111/udp	rpcbind	portmap	#Appel de procédure distante SUN
auth	113/tcp	ident	tap	#Protocole d'identification
uucp-path	117/tcp			
nntp	119/tcp	usenet		#Protocole de transfert de nouvelles par Internet
ntp	123/udp			#Protocole d'heure du réseau
epmap	135/tcp	loc-srv		#Résolution de point de sortie DCE
epmap	135/udp	loc-srv		#Résolution de point de sortie DCE
netbios-ns	137/tcp	nbname		#Service de nom NETBIOS
netbios-ns	137/udp	nbname		#Service de nom NETBIOS
netbios-dgm	138/udp	nbdatagram		#Service de datagramme NETBIOS
netbios-ssn	139/tcp	nbsession		#Service de session NETBIOS
imap	143/tcp	imap4		#Protocole d'accès de messagerie Internet
pcmail-srv	158/tcp			#Serveur PCMail
snmp	161/udp			#SNMP
snmptrap	162/udp	snmp-trap		#Printége SNMP
print-srv	170/tcp			#PostScript réseau
bgp	179/tcp			#Protocole de passerelle de frontière
irc	194/tcp			#Protocole IRC (Internet Relay Chat)
ipx	213/udp			#IPX par IP
ldap	389/tcp			#Protocole allégé d'accès aux répertoires
https	443/tcp	MCom		
https	443/udp	MCom		
microsoft-ds	445/tcp			
microsoft-ds	445/udp			
kpasswd	464/tcp			# Kerberos (v5)
kpasswd	464/udp			# Kerberos (v5)
isakmp	500/udp	ike		#Echange de clés Internet
exec	512/tcp			#Execution de processus ... distance
biff	512/udp	comsat		
login	513/tcp			#Connexion ... distance
who	513/udp	whod		
cmd	514/tcp	shell		
syslog	514/udp			
printer	515/tcp	spooler		
talk	517/udp			
ntalk	518/udp			
efs	520/tcp			#Serveur de noms de fichiers ,tendus
router	520/udp	route	routed	
timed	525/udp	timeserver		
tempo	526/tcp	newdate		
courier	530/tcp	rpc		
conference	531/tcp	chat		
netnews	532/tcp	readnews		

netwall	533/udp		#Pour diffusions en urgence
uucp	540/tcp	uucpd	
klogin	543/tcp		#Ouverture de session Kerberos
kshell	544/tcp	krcmd	#Interpr,teur distant Kerberos
new-rwho	550/udp	new-who	
remoteefs	556/tcp	rfs rfs_server	
rmonitor	560/udp	rmonitord	
monitor	561/udp		
ldaps	636/tcp	sldap	#LDAP par TLS/SSL
doom	666/tcp		#Logiciel ID Doom
doom	666/udp		#Logiciel ID Doom
kerberos-adm	749/tcp		#Administration Kerberos
kerberos-adm	749/udp		#Administration Kerberos
kerberos-iv	750/udp		#Kerberos version IV
kpop	1109/tcp		#POP Kerberos
phone	1167/udp		#Appel de conf,rence
ms-sql-s	1433/tcp		#Microsoft-SQL-Server
ms-sql-s	1433/udp		#Microsoft-SQL-Server
ms-sql-m	1434/tcp		#Microsoft-SQL-Moniteur
ms-sql-m	1434/udp		#Microsoft-SQL-Moniteur
wins	1512/tcp		#Microsoft Windows Internet Name Service (WINS)
wins	1512/udp		#Microsoft Windows Internet Name Service (WINS)
ingreslock	1524/tcp	ingres	
l2tp	1701/udp		#Protocole de tunneling couche 2
pptp	1723/tcp		#Protocole de tunneling de point ... point
radius	1812/udp		#Protocole d'authentification RADIUS
radacct	1813/udp		#Protocole de gestion de comptes RADIUS
nfsd	2049/udp	nfs	#Serveur NFS
knetd	2053/tcp		#D,multiplexeur Kerberos
man	9535/tcp		#Serveur MAN distant

## 8. Les fonctions de consultation des services

On s'en doute, une ligne de ce fichier correspond en programmation à une structure définie dans netdb.h :

### structure servent

```
struct servent
{
    char *s_name;      /* official service name */
    char **s_aliases;  /* alias list */
    int   s_port;      /* port # */
    char *s_proto;    /* protocol to use */
};
```

Les fonctions de manipulation sont bien conformes à ce que l'on attend :

---

```

struct servent *getservbyname (      <nom du service - const char *>,
                                    <nom du protocole - const char *>);
struct servent *getservbyport (     <port codé en format réseau - int>,
                                    <nom du protocole - const char *>);
void setservent(<flag indiquant si le fichier doit rester ouvert – int>);
struct servent *getservent(void);
void endservent(void);

```

Commençons par parcourir le fichier :

### CONFIG07.C

```

/* CONFIG07.C
- Claude Vilvens -
*/
...
#include <netdb.h>      /* pour la structure hostent */
...
int main()
{
    char nomService[40];
    struct servent *infosService;
    char * pTrav; /* Pour la cours du tableau des alias */
    int i = 0;

    if (setservent(0)==NULL) puts("Ouverture de /etc/services impossible");
    else
    {
        while ( (infosService = getservent()) != NULL)
        {
            printf("Nom = %s\n", infosService->s_name);
            printf("\tPort = %d\n", ntohs(infosService->s_port));
            printf("\tProtocole = %s\n", infosService->s_proto);
            pTrav = (infosService->s_aliases)[i];
            while (pTrav)
            {
                printf("\t%s\n", pTrav);
                pTrav = (infosService->s_aliases)[++i];
            }
        }
        endservent();
    }

    return 0;
}

```

Ce qui donne :

```

Nom = echo
    Port = 7
    Protocole = tcp
Nom = echo
    Port = 7
    Protocole = udp
Nom = discard
    Port = 9
    Protocole = tcp
    sink
    null
Nom = discard
    Port = 9
    Protocole = udp
Nom = daytime
    Port = 13
    Protocole = tcp
Nom = daytime
    Port = 13
    Protocole = udp
Nom = netstat
    Port = 15
    Protocole = tcp

```

Pour les fonctions ciblées, on peut écrire :

### CONFIG08.C

```

/* CONFIG08.C
- Claude Vilvens -
*/
...
int main()
{
    char nomService[40], nomProtocole[20], numPort[20];
    struct servent * infosService;
/* 1. Acquisition des informations sur un service de nom donne */
    do
    {
        printf("Nom du service cherche (FIN) : "); gets(nomService);
        if (strcmp(nomService, "FIN") == 0) break;
        printf("Nom du protocole cherche : "); gets(nomProtocole);
        if ((infosService = getservbyname(nomService, nomProtocole)) == 0)

        {
            printf("Erreur d'acquisition d'infos sur le service %d\n", errno);
        }
    else
    {
        printf("Acquisition infos service OK\n");
    }
}

```

```

        printf("Port = %d\n", ntohs(infosService->s_port));
    }
}
while (1);

/* 2. Acquisition des informations sur un service de port donne */
do
{
    printf("Port du service cherche (0) : "); gets(numPort);
    if (strcmp(numPort, "0") == 0) break;
    printf("Nom du protocole cherche : "); gets(nomProtocole);
    if ((infosService = getservbyport(htons(atoi(numPort)), nomProtocole)) == 0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    }
    else
    {
        printf("Acquisition infos host OK\n");
        printf("Nom = %s\n", infosService->s_name);
    }
}
while (1);

return 0;
}

```

Cela peut donner :

```

% conf
Nom du service cherche (FIN) : telnet
Nom du protocole cherche : tcp
Acquisition infos service OK
Port = 23
Nom du service cherche (FIN) : telnet
Nom du protocole cherche : udp
Erreur d'acquisition d'infos sur le service 3
Nom du service cherche (FIN) : echo
Nom du protocole cherche : tcp
Acquisition infos service OK
Port = 7
Nom du service cherche (FIN) : ftp
Nom du protocole cherche : tcp
Acquisition infos service OK
Port = 21
Nom du service cherche (FIN) : FIN
Port du service cherche (0) : 21
Nom du protocole cherche : tcp
Acquisition infos host OK
Nom = ftp
Port du service cherche (0) : 23

```

Nom du protocole cherche : ***tcp***

Acquisition infos host OK

Nom = telnet

Port du service cherche (0) : ***25***

Nom du protocole cherche : ***tcp***

Acquisition infos host OK

Nom = smtp

Port du service cherche (0) : 0

%



Le nombre d'adresses IP disponibles devient trop petit pour un Internet en pleine explosion ☹ ! La solution : IPv6 ...

## XII. Le protocole IPv6



*L'âge amènera tout, et ce n'est pas le temps,  
Madame, comme on sait, d'être prude à vingt ans.*

(Molière, Le Misanthrope).

### 1. Un meilleur IP

On le sait : avec une adresse définie sur **32 bits** seulement, le protocole IP courant (appelé encore **IPv4** pour insister sur les 4 bytes utilisés pour cette adresse) possède un espace d'adressage assez limité et, en fait, peu en rapport avec la taille actuelle d'Internet et le nombre de machines qui y interviennent. Pour contourner le problème potentiel d'une pénurie d'adresses, on a donc inventé le concept d'adresses publiques et privées et l'utilisation du NAT. Cependant, cette manière de faire ralentit (à cause de la surcharge de stables de routage) et obscurcit (pas de mode "*end to end*" puisque les vraies adresses des deux extrémités de la communication sont le plus souvent cachées).

Le protocole **IPv6** (encore appelé initialement **IPng - IP new generation**) s'est donné pour objectif de résoudre ces divers problèmes, en utilisant des adresses définies sur **128 bits** (16 bytes) et en ajoutant de plus une diffusion multicast plus facile et une meilleure qualité de service (notamment vis-à-vis du temps réel).

Le premier point est évidemment l'utilisation d'adresses plus longues, permettant d'adresser

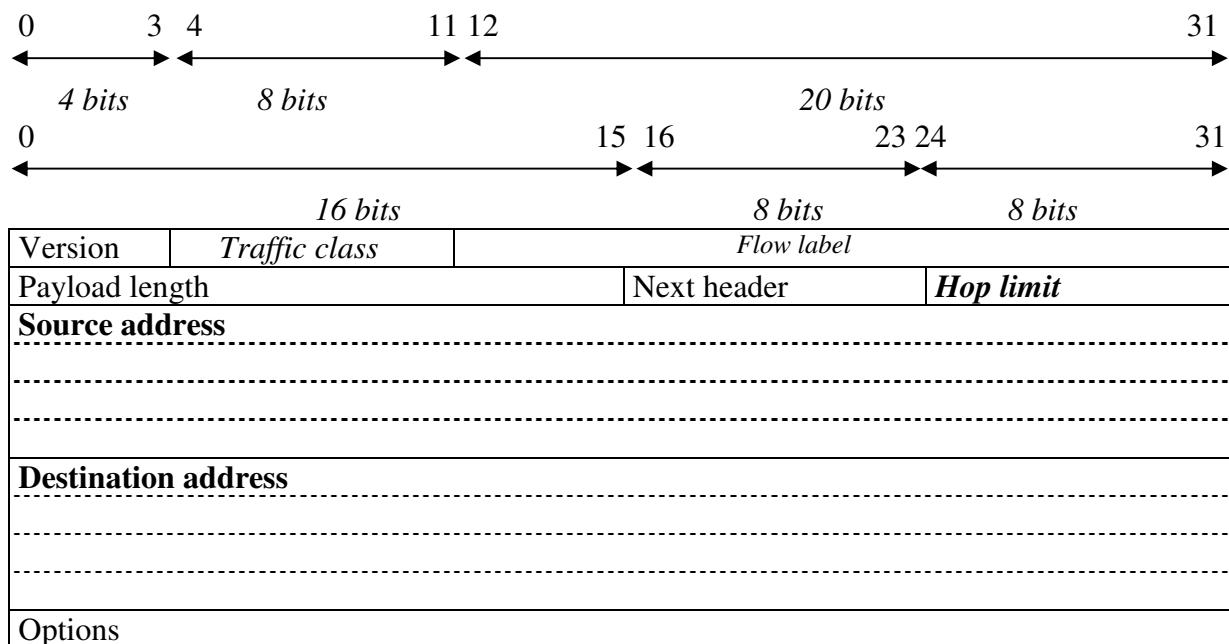
$$2^{128} = 3.4 \cdot 10^{38} \text{ adresses distinctes.}$$

Mais il n'y a pas que cela :

- ◆ l'en-tête est simplifié (8 champs au lieu de 14), ce qui permet aux routeurs de traiter les datagrammes plus rapidement;
- ◆ les options d'envoi n'apparaissent que si nécessaire dans les champs optionnels, au lieu d'être systématiquement obligatoires;
- ◆ le champ de type de service a été conservé pour faire face à la montée en puissance du multimédia – IPv6 implémente nativement QoS (Quality of Service);
- ◆ le multicast est implémenté nativement;
- ◆ IPv6 se préoccupe d'authentification et de confidentialité et utilise IPsec;
- ◆ le protocole est capable de s'auto-configurer (par exemple sur base de l'adresse MAC).

### 2. La structure d'un datagramme IPv6

Un datagramme IPv6 comporte au moins un en-tête suivi des données proprement dites (la "charge utile"). Cet en-tête est composé de 40 octets ou 5 mots de 64 bits. Si il y a des options, elles apparaissent sous la forme d'en-têtes supplémentaires de même format que le premier en-tête (donc, de 40 octets). Schématiquement, l'en-tête IPv6 a la structure suivante :



On peut constater que cet en-tête moins complexe que son prédecesseur. On peut remarquer immédiatement, outre le champ de **version** sur 4 bits forcément incontournable pour les routeurs (4 ou 6), les adresses de la source et de la destination du datagramme : **Source address** et **Destination address**, chacune sur 128 bits.

Le champ **Traffic class** est en fait le correspondant du **Type of service** d'IPv4 : il permet de définir la priorité (0=normal, 7=supervision réseau) pour que les routeurs capables de contrôler leur débit en tiennent compte. C'est le concept de **QoS** (Quality of Service) : certaines applications sont négativement sensibles à un temps de latence trop important (typiquement, les flux vidéos) et réclameront donc une priorité élevée, tandis que d'autres le sont beaucoup moins (typiquement, un banal transfert de fichiers). Les niveaux de priorité peuvent se schématiser ainsi :

0	pas de priorité spécifique	4	transfert prévisible (FTP)
1	arrière-plan (news)	5	réservé (?)
2	données imprévisibles (mail)	6	trafic interactif (telnet, SSL, fenêtre)
3	réservé (?)	7	contrôle du trafic (routage)

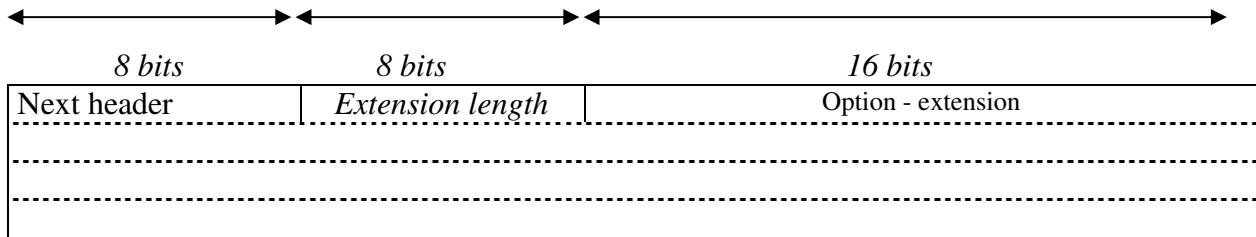
Associé à cette idée de service, le champ **Flow label** (identificateur de flux) contient un identificateur unique (en fait, un nombre) choisi par l'émetteur et destiné aux routeurs : ceux-ci l'utiliseront pour appliquer leur stratégie de qualité de service selon la source.

Le champ **Hop limit** évoque la notion de portée évoquée pour le multicast. C'est un compteur limitant la durée de vie du datagramme puisqu'il est décrémenté de 1 à chaque passage dans un noeud (routeur); s'il tombe à 0, il y aura destruction et avertissement de l'ordinateur source (erreur ICMP6). Bien clairement, c'est l'équivalent du champ TTL d'IPv4.

Le champ **Payload length** contient la taille des données, les champs d'en-tête n'étant pas comptés puisque de longueur toujours fixe.

Enfin, le champ **Next header** rappelle le champ Protocol de IPv4 : il identifie le prochain en-tête que l'on trouvera dans le datagramme, précisant s'il s'agit d'un en-tête de protocole encapsulé (TCP, UDP, ICMP) ou d'extensions, autrement dit d'**options** d'IPv6 (typiquement IPsec). Les **extensions** se présentent donc en fait comme des espèces d'en-têtes IPv6 supplémentaires : on parle encore d'encapsulation d'IP dans IP. Leur longueur doit être un multiple de 8 octets. Après un champs d'en-tête sur 8bits, on trouve un deuxième octet qui définit ce qui suit l'extension : une autre extension ou un en-tête de protocole (TCP, UDP,

ICMP); éventuellement le 3<sup>ème</sup> octet contient la longueur de l'extension en unités de 8 octets si l'extension est à longueur variable :



Les extensions prévues sont principalement :

- ◆ proche-en-proche (hop-by-hop) : octets de bourrage (padding) pour respecter les alignements sur des mots de 32 ou 64 bits, jumbogramme pour des paquets très gros au point que le champ de longueur de données d'IPv6 ne suffit pas, etc; cette extension doit toujours se trouver en 1<sup>ère</sup> position et est la seule à être prise en compte par les routeurs intermédiaires (les autres sont ignorées par ceux-ci);
- ◆ routage : pour imposer à un paquet une route différente de celles prévues par la politique de routage prévue;
- ◆ sécurité; soit
- ◆ AH (Authentication header) : HMAC-MD5, HMAC-SHA1, signature numérique
- ◆ ESP (Encapsulating Security Payload) : données cryptées par DES ou AES
- ◆ fragmentation – voir ci-dessous ...

On peut évidemment se demander où sont passés les champs d'IPv4 relatifs à la fragmentation. En fait, ils ont disparu parce que ***IPv6 n'accepte pratiquement plus la fragmentation*** ! Ainsi, la RFC précise que tous les intervenants (ordinateurs, routeurs) doivent supporter des datagrammes avec un MTU de 1280 bytes (en pratique, celui-ci est le plus souvent de 1500 bytes). L'idée est donc d'adapter la taille des paquets à l'émission plutôt que de fragmenter puis de reconstituer à l'arrivée. *Si la fragmentation s'avère malgré tout incontournable* sous peine de devoir réécrire le protocole applicatif (cas de NFS sur UDP qui produit des messages de grande taille), IPv6 utilisera une extension rassemblant les mêmes champs relatifs à la fragmentation que ceux d'IPv4.

On remarquera encore que le champ Checksum d'IPV4 a disparu : celui-ci posait problème puisqu'il devait être recalculé à chaque décrémentation du TTL. C'est donc le protocole de niveau supérieur (typiquement TCP et UDP) qui doivent s'en charger.

### 3. La notation des adresses d'IPv6

La notation des adresses codées sur 128 bits se base sur un découpage de celles-ci en 8 groupe de 16 bits, que l'on peut donc noter avec 4 chiffres hexadécimaux, chaque groupe étant séparé du suivant par le symbole ":". Donc, par exemple :

1FFF:0000:0000:0000:1234:6A69:0B11:A113

Il est assez fréquent que des adresses contiennent plusieurs 0 ou des groupes de 0 et on a imaginé d'alléger dans ce cas la notation :

- ◆ les premiers 0 d'un groupe peuvent être omis;
- ◆ un (ou plusieurs groupes) de 4 zéros peut (peuvent) être remplacé(s) par "::"

L'adresse ci-dessus peut donc encore s'écrire :

1FFF::1234:6A69:B11:A113

Cependant, le remplacement de groupes de 0 par "::" ne peut se faire que pour un seul groupe; pour les autres, on peut seulement remplacer le groupe de 4 zéros par un seul – ceci afin d'éviter toute ambiguïté.

**Les adresses IPv4 ont leur 6 premiers groupes (donc 96 bits) à 0** tandis que les 2 derniers contiennent le codage de l'adresse IPv4, qui peut être notée avec la traditionnelle notation décimale pointée – on parle encore d'**adresses IPv4compatibles**. Par exemple, l'adresse IPv4 192.168.1.8 devient en IPv6 :

::192.168.1.8

De manière plus générale, il est permis (mais pas conseillé) d'utiliser la notation décimale pointée pour les deux derniers groupes d'une adresse IPv6 quelconque.

Enfin, IPv6 connaît la notion de **sous-réseau** et identifie les adresses d'un tel sous-réseau par une même séquence binaire initiale, le nombre de bits utilisés étant signalé après "/" – on parle encore de "**préfixe**". Par exemple :

C800:523A:1:1A0::/59

#### **Remarque**

Si on utilise une adresse IPv6 comme nom d'hôte, dans une URL par exemple, il faut l'entourer de crochets :

[http://\[1FFF::1234:6A69:B11:A113\]/index.html](http://[1FFF::1234:6A69:B11:A113]/index.html)

## **4. Différents types d'adresses**

IPv6 connaît trois types d'adresses :

- ◆ **les adresses unicast** : elles désignent un seul interface, soit sur Internet soit à l'intérieur d'un intranet; elles comportent un préfixe et un identifiant de l'interface, avec éventuellement un identifiant du sous-réseau :
  - adresse lien local : préfixe = FE80::/64 + id interface sur 64 bits
  - adresse site local : préfixe = FEC0::/48 + id sous-réseau sur 16 bits + id interface sur 64 bits
- ◆ **les adresses multicast** : une telle adresse désigne un groupe d'interfaces pouvant se trouver n'importe où; le concept est bien connu : un paquet envoyé vers cette adresse sera en fait acheminé à tous les interfaces du groupes; à remarquer qu'un "multicast général", c'est-à-dire un broadcast, n'existe pas en IPv6 afin de ne pas risquer de saturer le réseau; le format général d'une telle adresse, dérivée du préfixe FF00::/8, est le suivant :

FF + 4 bits flags + 4 bits de portée + identifiant du groupe sur 112 bits

où les 4 bits flags sont, de celui du poids le plus fort au plus faible :

- non attribué;
- R et P : utilisés pour indiquer l'inclusion d'une adresse de point de rendez-vous;
- T (transient) : à 0 pour une adresse multicast permanente;

tandis que les bits de portée (scope) nous sont connus (voir chapitre consacré à UDP, section multicast);

---

- ♦ **les adresses anycast** : une telle adresse désigne à nouveau un groupe d'interfaces pouvant se trouver n'importe où, mais un paquet envoyé vers cette adresse sera en fait seulement acheminé à l'un des interfaces du groupes (c'est un moyen aisément d'identifier un groupe de serveurs dévolus au même service); de telles adresses comportent un préfixe sur n bits qui est une adresse unicast et une partie identifiant sur 128-n bits.

Bien sûr, certaines adresses jouent un rôle particulier :

0:0:0:0:0:0:0:0 : pour les sockets d'écoute  
0:0:0:0:0:0:0:1 : adresse de loopback, équivalente de 127.0.0.1 d'IPv4

## 5. Installation du stack IPv6

Une machine peut posséder la double pile IPv4 et IPv6. Si le stack IPv6 n'existe pas encore (il est présent par défaut sous Windows 7), on peut l'installer sous Windows XP par la ligne de commande `ipv6 install`. Ainsi, si au préalable :

```
C:\Documents and Settings\vilvens.THALASSA>ipconfig /all
```

Configuration IP de Windows

Nom de l'hôte ..... : ulysssevil  
Suffixe DNS principal . . . : thalassa.inpres.prov-liege.be  
Type de noeud ..... : Inconnu  
Routage IP activé ..... : Non  
Proxy WINS activé ..... : Non  
Liste de recherche du suffixe DNS : thalassa.inpres.prov-liege.be  
                                  wildness.loc  
                                  inpres.epl.prov-liege.be

Carte Ethernet Connexion au réseau local:

Suffixe DNS propre à la connexion :  
Description ..... : Realtek RTL8169/8110 Family Gigabit Ethernet NIC  
Adresse physique . . . . . : 00-18-F3-EB-C8-C9  
DHCP activé. .... . : Non  
Adresse IP..... . . . . . : 10.59.5.17  
Masque de sous-réseau : 255.255.254.0  
Passerelle par défaut . . . : 10.59.4.254  
Serveurs DNS ..... . . . . . : 10.59.4.2  
                                  10.59.5.6  
                                  10.7.0.100

Puis :

```
C:\Documents and Settings\vilvens.THALASSA>ipv6 install  
Installation en cours...  
Opération réussie.
```

On peut vérifier le résultat avec un nouveau `ipconfig /all` :

---

C:\Documents and Settings\vilvens.THALASSA>**ipconfig /all**

Configuration IP de Windows

Nom de l'hôte ..... : ulyssevil  
Suffixe DNS principal .... : thalassa.inpres.prov-liege.be  
Type de noud ..... : Inconnu  
Routage IP activé ..... : Non  
Proxy WINS activé ..... : Non  
Liste de recherche du suffixe DNS : thalassa.inpres.prov-liege.be  
                                  wildness.loc  
                                  inpres.epl.prov-liege.be

Carte Ethernet Connexion au réseau local:

Suffixe DNS propre à la connexion :  
Description ..... : Realtek RTL8169/8110 Family Gigabit Ethernet NIC  
Adresse physique ..... : **00-18-F3-EB-C8-C9**  
DHCP activé ..... : Non  
Adresse IP..... : 10.59.5.17  
Masque de sous-réseau . : 255.255.254.0  
Adresse IP..... : fe80::218:f3ff:feeb:c8c9%5  
Passerelle par défaut . . . : 10.59.4.254  
Serveurs DNS ..... : 10.59.4.2  
                          10.59.5.6  
                          10.7.0.100  
                          fec0:0:0:ffff::1%1  
                          fec0:0:0:ffff::2%1  
                          fec0:0:0:ffff::3%1  
....

On peut donc remarquer que l'adresse IPv6 a été construite à partir de l'adresse MAC de la carte réseau. Le symbole "%5" à la fin de l'adresse désigne en fait le numéro de l'interface utilisée.

## **6. La programmation des sockets IPv6 en C/C++**

### **6.1 Une nouvelle famille et de nouvelles structures d'adresse**

L'apparition d'IPv6 a évidemment entraînée l'apparition d'une nouvelle famille d'adresses AF\_INET6 et d'une nouvelle famille de protocoles PF\_INET6 :

dans **sys/socket.h**

```
...
#define      AF_INET6   26          /* Internet Protocol, Version 6 */
...
#define      PF_INET6   AF_INET6
```

Les structures sockaddr\_in et in\_addr se déclinent aussi en IPv6 :

---

```
structure sockaddr_in6 (in.h)
struct sockaddr_in6
{
    sa_family_t    sin6_family;
    in_port_t      sin6_port;
    uint32_t       sin6_flowinfo;
    struct in6_addr    sin6_addr;
    uint32_t sin6_scope_id;      /* Depends on scope of sin6_addr */
    uint32_t __sin6_src_id;     /* Impl. specific - UDP replies */
};

/* a structure for historical reasons */
struct in6_addr
{
    union
    {
#define _KERNEL
        uint32_t      _S6_u32[4]; /* IPv6 address */
        uint8_t      _S6_u8[16]; /* IPv6 address */
#endif
        uint8_t      _S6_u8[16]; /* IPv6 address */
        uint32_t      _S6_u32[4]; /* IPv6 address */
#endif
        uint32_t      __S6_align; /* Align on 32 bit boundary */
    } _S6_un;
};
```

La création d'une socket s'effectue alors d'une manière similaire à celle utilisée en IPv4 :

```
int hSocket, /* Handle de la socket */
hSocket = socket (AF_INET6 SOCK_STREAM, 0); /* au sens strict : PF_INET6 */
```

Cependant, la structure sockaddr\_in6 est plus longue que la structure générique sock\_addr et elle ne peut donc plus remplacer celle-ci telle quelle (dans une instruction **accept()** par exemple). Une autre structure nommée **sockaddr\_storage** a donc été définie pour se rendre indépendant de ce problème :

```
strcture sockaddr_storage
struct sockaddr_storage
{
    short ss_family;
    char __ss_pad1[6]; /* pad to 8 */
    __int64 __ss_align; /* force alignment */
    char __ss_pad2[112]; /* pad to 128 */
};
```

## 6.2 Attacher une socket à un interface et un service

Dans le cas d'un serveur, nous allons réaliser un bind() des plus classiques, mais, au lieu d'utiliser gethostbyname() pour découvrir l'adresse de l'hôte, nous allons utiliser :

```
int getaddrinfo (<nom de la machine selon le DNS - const char *>,
                 <nom du service - const char *>,
                 <choix famille, type et protocole pour réponse - const struct addrinfo *>,
                 <liste d'adresses - struct addrinfo **>);
```

où la structure addrinfo est définie dans netdb.h comme

### structure addrinfo

```
struct addrinfo
{
    int ai_flags;           /* AI_PASSIVE, AI_CANONNAME, ... */
    int ai_family;          /* PF_XXX */
    int ai_socktype;        /* SOCK_XXX */
    int ai_protocol;         /* 0 or IPPROTO_XXX for IPv4 and IPv6 */
    socklen_t ai_addrlen;
    char *ai_canonname; /* canonical name for hostname */
    struct sockaddr *ai_addr; /* binary address */
    struct addrinfo *ai_next; /* next structure in linked list */
};
```

Les deux premiers paramètres de getaddrinfo() ne peuvent être NULL simultanément :

- ◆ le "nom" de la machine peut-être un nom DNS (ou défini dans etc/hosts); si il est à NULL, le champ adresse du résultat ne sera pas initialisé (on aura un INADDRANY ou son équivalent IPv6, soit IN6ADDRANY\_INIT);
- ◆ le "service" peut être un nom ou un numéro, faisant référence au fichier etc/services, par exemple :

cliser	50000/tcp
--------	-----------

- si il est à NULL, le champ port du résultat ne sera pas initialisé.

Le troisième paramètre permet de donner des indications sur le type de réponse attendue dans le quatrième paramètre (d'où son nom de "*hints*" dans la documentation) : famille, protocole, type; la valeur NULL signalant que l'on attend une réponse pour toutes les adresses et tous les protocoles. Le bit **AI\_PASSIVE** du champ flags est classiquement positionné dans le cas où le résultat doit permettre à un serveur de réaliser un bind(); il est au contraire non positionné pour une socket destinée à un connect() (TCP) ou un sendTo() (UDP). Si le bit **AI\_NUMERICHOST** du champ flags est positionné, le "nom" de la machine est en fait une adresse IPv4 ou IPv6 (donc, il n'y a pas d'appel DNS).

Une fois le travail de bind() réalisé, on pourra libérer l'espace mémoire avec

<b>void freeaddrinfo ( &lt;adresse - struct addrinfo *&gt;);</b>
--

Concrètement, on écrira donc quelque chose du genre :

```

struct addrinfo indications, *infosHost;
int errgetaddr;

memset(&indications, 0, sizeof indications);
indications.ai_flags = AI_PASSIVE;
indications.ai_socktype = SOCK_STREAM;
indications.ai_family = AF_INET6;

getaddrinfo("localhost", "cliser", &indications, &infosHost);
bind(hSocketEcoute, infosHost->ai_addr, infosHost->ai_addrlen);
freeaddrinfo(infosHost);

```

### **6.3 La prise en compte d'une connexion**

Côté serveur, une fois le bind() réalisé, on peut faire démarrer la machine à état de TCP (listen()) puis prendre en compte une connexion pendante avec accept(). C'est ici qu'intervient la nouvelle structure sockaddr\_storage, afin de réceptionner l'adresse distante : l'utilisation d'une sockaddr\_in classique provoquerait une erreur, puisque trop courte pour recevoir une adresse IPv6. Donc :

```

struct sockaddr_storage rem;
socklen_t remlen = sizeof (rem);

listen(hSocketEcoute,SOMAXCONN) ;
hSocketService = accept(hSocketEcoute, (struct sockaddr *)&rem, &remlen);

```

Côté client, on réalise un connect() au moyen d'une socket préalablement initialisée en utilisant les résultats obtenus par un getaddrinfo() qui, cette fois, n'a évidemment pas le bit AI\_PASSIVE positionné (dans certaines configurations, on positionne cependant le bit AI\_ADDRCONFIG).

```

struct addrinfo indications, *infosHost;
hSocket = socket(AF_INET6, SOCK_STREAM, 0);

memset(&indications, 0, sizeof indications);
indications.ai_flags = 0;
indications.ai_socktype = SOCK_STREAM;
indications.ai_family = AF_INET6;

getaddrinfo("localhost", "cliser", &indications, &infosHost);
connect(hSocket, infosHost->ai_addr, infosHost->ai_addrlen);

```

### **6.4 Pour identifier le client connecté**

On peut savoir que les fonctions inet\_addr() et inet\_ntoa() de conversion "adresse réseau"/"adresse chaîne" ont des équivalents permettant de traiter IPv6 :

```

int inet_pton (<famille AF_INET ou AF_INET6 – int >,
               <adresse sous forme de chaîne - const char *>,
               <adresse où placer le résultat binaire de la conversion - void *>);

```

```
const char *inet_ntop (<famille AF_INET ou AF_INET6 – int >,
                      <adresse binaire à convertir - const void>,
                      <adresse où placer la chaîne obtenue - char *>,
                      <taille de la zone réceptrice - size_t size);
```

Et on pourra utiliser les deux constantes (dans in.h) :

```
#define INET_ADDRSTRLEN    16     /* max len IPv4 addr in ascii dotted */
                                /* decimal notation. */
#define INET6_ADDRSTRLEN 46 /* max len of IPv6 addr in ascii */
```

Cependant, il est beaucoup plus simple d'utiliser :

```
int getnameinfo (<structure reçue par accept pour le client - const struct sockaddr *>,
                 <longueur structure client - socklen_t>,
                 <nom ou adresse - char *>,
                 <longueur de la zone de stockage - size_t>,
                 <nom du service ou NULL - char *>,
                 <longueur de la zone service ou 0 - size_t>
                 <flag pour indiquer si on désir l'adresse ou le nom - int>
);
```

où le dernier paramètre peut prendre la valeur

- ◆ NI\_NUMERICHOST pour obtenir l'adresse sous forme de chaîne ;
- ◆ NI\_NAMEREQD pour obtenir le nom de la machine

On ajoutera dans le code :

```
char bufAdresse[100];
getnameinfo ((struct sockaddr *) &rem, remlen, bufAdresse, sizeof (bufAdresse),
              NULL, 0, NI_NUMERICHOST);
printf("Connexion de %s\n", bufAdresse);
char bufNom[100];
getnameinfo ((struct sockaddr *) &rem, remlen, bufNom, sizeof (bufNom), NULL, 0, 0);
printf("Connexion de %s\n", bufNom);
```

## **6.5 Le serveur IPv6 de base**

Nous pouvons donc résumer les opérations classiques d'un serveur dans le code suivant (en localhost). A remarquer que, sous Windows, le header nécessaire pour IPv6 est **winsock2.h**, agrémenté de **ws2tcpip.h**. De plus, dans les paramètres du linker, il faudra ajouter à

-lwsock32 -lwinmm -lmswsock

la ligature avec

**-lws2\_32**

Cela donne :

### TCPv6ITER03.C

```
/* TCPv6ITER03.C
SOCKETSERVEUR IPv6
- Claude Vilvens -
Cr: 19/6/2008
*/
#ifndef SocketsUnix
#include <stdio.h>
#include <stdlib.h> /* pour exit */
#include <string.h> /* pour memcpy */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h> /* pour les types de socket */
#include <netdb.h> /* pour la structure hostent */
#include <errno.h>
#include <netinet/in.h> /* pour la conversion adresse reseau->format dot
                         ainsi que le conversion format local/format
                         reseau */
#include <netinet/tcp.h> /* pour la conversion adresse reseau->format dot */
#include <arpa/inet.h> /* pour la conversion adresse reseau->format dot */

#include "tcpiter03.h"
#endif

#ifndef SocketsWindows
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#include <windows.h>
#include <winsock2.h> /* au lieu de #include <winsock.h> */
#include <ws2tcpip.h>

#include <io.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#endif

#include "tcpv6iter03.h"

void afficheRequete(struct client *c);
```

```

int main(int argc, char *argv[])
{
    int hSocketEcoute, /* Handle de la socket d'ecoute*/
        hSocketService; /* Handle de la socket de service connectee au client */

    struct addrinfo indications, *infosHost;
    int erregetaddr;

    struct sockaddr_storage rem;
    socklen_t remlen = sizeof (rem);

    char bufNom[100];
    char bufAdresse[100];

    char msgServeur [LONG_MSG_SERV];
    struct client *msgClient = (struct client *)malloc(sizeof(struct client));
    int tailleMsgRecu, nbreBytesRecus;
    int i;
    char buf[100];

    printf("?? Taille d'un client = %d\n", sizeof (struct client));

#define SocketsWindows
// Pour windows
WORD version;
WSADATA infoImpl;
int err;

version = MAKEWORD( 1, 1 );

err = WSAStartup(version, & infoImpl);
if ( err != 0 )
{
    printf ("Problème au startup de Windows");
    exit(1);
}
// Fin Pour Windows
#endif

/* 1. Creation de la socket */
    hSocketEcoute = socket(AF_INET6,SOCK_STREAM,0);
    printf("Apres creation de la socket : hSocketEcoute = %d\n", hSocketEcoute);
    if (hSocketEcoute == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        system("PAUSE");
        exit(1);
    }
    else printf("Creation de la socket OK\n");

```

```

/* 2. Préparation de la structure adddrinf */
    memset(&indications, 0, sizeof indications);
    indications.ai_flags = AI_PASSIVE;
    indications.ai_socktype = SOCK_STREAM;
    indications.ai_family = AF_INET;

/* 3. Acquisition des informations sur l'ordinateur local */
    if ( (errgetaddr = getaddrinfo("localhost", "cliser", &indications, &infosHost))!=0)
    {
        printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
        system("PAUSE");
        exit(1);
    }
    else printf("Acquisition infos host OK\n");

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
    if (bind(hSocketEcoute, infosHost->ai_addr, infosHost->ai_addrlen) == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        fprintf(stderr, "bind : %s\n", strerror(errno));
        system("PAUSE");
        exit(1);
    }
    else printf("Bind adresse et port socket OK\n");
    freeaddrinfo(infosHost);

do
{
/* 5. Mise a l'ecoute d'une requete de connexion */
    if (listen(hSocketEcoute,SOMAXCONN) == -1)
    {
        printf("Erreur sur le listen de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Listen socket OK\n");

/* 6. Acceptation d'une connexion */
    if ( (hSocketService = accept(hSocketEcoute, (struct sockaddr *)&rem, &remlen) ) == -1)
    {
        printf("Erreur sur l'accept de la socket %d\n", errno);
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else printf("Accept socket OK\n");

    getnameinfo ((struct sockaddr *) &rem, remlen, bufAdresse, sizeof (bufAdresse),
                NULL, 0, NI_NUMERICHOST);
    printf("Connexion de %s\n", bufAdresse);
}

```

```

getnameinfo((struct sockaddr *)&rem, remlen, bufNom, sizeof (bufNom), NULL,
0, 0);
printf("Connexion de %s\n", bufNom);

/* 7.Reception d'un message client */
tailleMsgRecu = 0;
do
{
    puts("Passage boucle de reception");
    if ((nbreBytesRecus = recv(hSocketService, ((char*)msgClient) +
tailleMsgRecu, LONG_STRUCT_CLI-tailleMsgRecu, 0))
== -1)
    {
        printf("Erreur sur le recv de la socket %d\n", errno);
        close(hSocketService); /* Fermeture de la socket */
        close(hSocketEcoute); /* Fermeture de la socket */
        exit(1);
    }
    else
    {
        printf("Taile msg recu = %d et taille attendue = %d\n",
tailleMsgRecu, LONG_STRUCT_CLI);
        tailleMsgRecu += nbreBytesRecus;
    }
    printf("Taill msg = %d et nbreBytes = %d \n", tailleMsgRecu,
nbreBytesRecus);
}
while (nbreBytesRecus != 0 && nbreBytesRecus != -1 &&
tailleMsgRecu <LONG_STRUCT_CLI );

printf("Recv socket OK\n");

if (strcmp(msgClient->nom, EOC))
    printf("Demande recue pour le client = %s\n", msgClient->nom);
else break;

/* 8. Envoi de l'ACK du serveur au client */
sprintf(msgServeur,"Demande recue du client %s !!!",
msgClient->nom);
printf("--- Coefficient de reduction = %s\n",
msgClient->coeffReduction);
printf("Nouveau coeff : ");gets(msgClient->coeffReduction);
afficheRequete(msgClient);
if (send(hSocketService, msgServeur, LONG_MSG_SERV, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocketService); /* Fermeture de la socket */
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}

```

```

        else printf("Send socket OK\n");
    }
while(1);

/* 9. Fermeture des sockets */
close(hSocketService); /* Fermeture de la socket */
printf("Socket connectee au client fermee\n");
close(hSocketEcoute); /* Fermeture de la socket */
printf("Socket serveur fermee\n");

system("PAUSE");
return EXIT_SUCCESS;
}

/* -----
void afficheRequete(struct client *c)
{
    printf("**** Requete d'un client ****\n");
    printf("Numero de client : %s\n", c->numClient);
    printf("Nom = %s\n", c->nom);
    printf("Date du dernier achat : %s\n", c->dateDernierAchat);
    printf("Numero de l'article demande : %s\n", c->numArticle);
    printf(" et son prix : %d\n", c->montant);
    printf("----taille prix : %d\n", sizeof(c->montant));
    printf("Coefficient de reduction = %s\n", c->coeffReduction);
    printf("Fourni ? = %d\n", c->fourni);
}

```

avec (aucun changement par rapport à IPv4) :

### **TCPIITER03.H**

```

/* TCPIITER03.H
- Claude Vilvens -
Cr: 19/4/99
Maj: 25/2/2000
*/
#ifndef TCPIITER_H
#define TCPIITER_H

#define EOC "END_OF_CONNEXION"

#define PORT 50000 /* Port d'ecoute de la socket serveur */

struct client
{
    char numClient[20];
    char nom[30];
    char dateDernierAchat[11];

```

```
char numArticle[15];
int montant;
char coeffReduction[10];
char fourni;
};

#define LONG_STRUCT_CLI sizeof(struct client) /* Longeur des messages */
#define LONG_MSG_SERV 100

#endif
```

## 6.6 Le client IPv6 de base

Les mêmes headers et options de link sont évidemment nécessaires :

### TCPv6CLI03.C

```
/* TCPv6CLI03.C
SOCKETSERVEUR IPv6
- Claude Vilvens -
Cr: 19/6/2008
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

#ifndef SocketsUnix
#define HOST "sunray2v440"
#include <sys/socket.h>
#include <netdb.h>
#include <errno.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <string.h>
#else
#define HOST "localhost"
#include <cstdlib>
#include <iostream>
#include <time.h>
#include <io.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#endif

#include "tcpv6iter03.h"
```

```

void analyseErreur(int numErreur);

struct client msgClient;

int main()
{
    int hSocket; /* Handle de la socket */

    struct addrinfo indications, *infosHost;
    int ergetaddr;
    int ret; /* valeur de retour */

    char msgServeur[LONG_MSG_SERV], buf[100];

    printf("?? Taille d'un client = %d\n", sizeof(struct client));

#define SocketsWindows
// Pour windows
WORD version;
WSADATA infoImpl;
int err;

version = MAKEWORD( 1, 1 );

err = WSAStartup(version, & infoImpl);
if ( err != 0 )
{
    printf ("Problème au startup de Windows");
    exit(1);
}
// Fin Pour Windows
#endif

/* 1. Création de la socket */
    hSocket = socket(AF_INET6, SOCK_STREAM, 0);
    if (hSocket == -1)
    {
        printf("Erreur de creation de la socket %d\n", errno);
        exit(1);
    }
    else printf("Creation de la socket OK\n");

/* 2. Acquisition des informations sur l'ordinateur distant */
    memset(&indications, 0, sizeof indications);
    indications.ai_flags = 0 /*| AI_ADDRCONFIG*/ ;
    indications.ai_socktype = SOCK_STREAM;
    indications.ai_family = AF_INET6;

    if ( (ergetaddr = getaddrinfo("localhost", "cliser", &indications, &infosHost))!=0)

```

```

{
    printf("Erreur d'acquisition d'infos sur la cible %d\n", errno);
    system("PAUSE");
    exit(1);
}
else printf("Acquisition infos cible OK\n");

/* 4. Tentative de connexion */
if (( ret = connect(hSocket, infosHost->ai_addr, infosHost->ai_addrlen) )
    == -1)
{
    printf("Erreur sur connect de la socket %d\n", errno);
}

#ifndef SocketsWindows
    analyseErreur(WSAGetLastError());
#else
    analyseErreur(errno);
#endif
close(hSocket);
exit(1);
}
else printf("Connect socket OK\n");
freeaddrinfo(infosHost);

/* 5. Envoi d'un message client */
printf("Nom du client : ");gets(msgClient.nom);
printf("Numero client : ");gets(msgClient.numClient);
printf("Numero d'article : ");gets(msgClient.numArticle);
if (strcmp(msgClient.nom, "ADMIN") ||
    strcmp(msgClient.numClient,"PASSWORD"))
{
    printf("Prix : ");gets(buf);msgClient.montant = atoi(buf);
    printf("Date du dernier achat : ");
    gets(msgClient.dateDernierAchat);
    printf("Coefficient de reduction : ");
    gets (msgClient.coeffReduction);
    msgClient.fourni=0;
}
if (send(hSocket, (char *)&msgClient, LONG_STRUCT_CLI, 0) == -1)
{
    printf("Erreur sur le send de la socket %d\n", errno);
    close(hSocket); /* Fermeture de la socket */
    exit(1);
}
else printf("Send socket OK\n");

printf("Demande envoyee pour le client = %s\n", msgClient.nom);

if (strcmp(msgClient.nom, "ADMIN")==0 &&
    strcmp(msgClient.numClient,"PASSWORD")==0 &&

```

```

        strcmp(msgClient.numArticle, "SHUTDOWN!") == 0)
        exit(0);

/* 6. Reception de l'ACK du serveur au client */
        if (strcmp(msgClient.nom, EOC))
        {
            if (recv(hSocket, msgServeur, LONG_MSG_SERV, 0) == -1)
            {
                printf("Erreur sur le recv de la socket %d\n", errno);
                close(hSocket); /* Fermeture de la socket */
                exit(1);
            }
            else
            {
                printf("Recv socket OK\n");
                printf("Message recu en ACK = %s\n", msgServeur);
            }
        }

/* 7. Fermeture de la socket */
        close(hSocket); /* Fermeture de la socket */
        printf("Socket client fermee\n");

#endif SocketsWindows
        system("PAUSE");
#endif
        return EXIT_SUCCESS;
}

/* -----
void analyseErreur (int numErreur)
{
#endif SocketsWindows
    switch(numErreur)
    {
        case WSAEBADF : printf("EBADF - hsocket n'existe pas\n");
                        break;
        case WSAENOTSOCK :
                        printf("ENOTSOCK - hsocket identifie un fichier\n");
                        break;
        case WSAEAFNOSUPPORT :
                        printf("EAFNOTSUPPORT - adresse ne correspond pas famille\n");
                        break;
        case WSAEISCONN : printf("EISCONN - socket deja connectee\n");
                        break;
        case WSAECONNREFUSED :
                        printf("ECONNREFUSED - connexion refusee par le serveur\n");
                        break;
        case WSAETIMEDOUT :
                        printf("ETIMEDOUT - time out sur connexion \n");
    }
}

```

```

        break;
    case WSAENETUNREACH :
        printf("ENETUNREACH - cible hors d'atteinte\n");
        break;
    default : printf("Erreur inconnue ?\n");
    }

#else
    switch(numErreur)
    {
    case EBADF : printf("EBADF - hsocket n'existe pas\n");
        break;
    case ENOTSOCK :
        printf("ENOTSOCK - hsocket identifie un fichier\n");
        break;
    case EAFNOSUPPORT :
        printf("EAFNOTSUPPORT - adresse ne correspond pas famille\n");
        break;
    case EISCONN : printf("EISCONN - socket deja connectee\n");
        break;
    case ECONNREFUSED :
        printf("ECONNREFUSED - connexion refusee par le serveur\n");
        break;
    case ETIMEDOUT :
        printf("ETIMEDOUT - time out sur connexion \n");
        break;
    case ENETUNREACH :
        printf("ENETUNREACH - cible hors d'atteinte\n");
        break;
    case EINTR :
        printf("EINTR - interruption par signal\n");
        break;
    default : printf("Erreur inconnue ?\n");
    }
#endif
}

```

On peut donc lancer le serveur, qui se met en attente :

```
C:\Documents and Settings\wilvens.THALASSA>netstat -an | find "50000"
TCP  [::1]:50000      [::]:0          LISTENING      0
```

puis lancer le client qui va se connecter :

```
C:\Documents and Settings\wilvens.THALASSA>netstat -an | find "50000"
TCP  [::1]:1027      [::1]:50000      ESTABLISHED  0
TCP  [::1]:50000      [::]:0          LISTENING      0
TCP  [::1]:50000      [::1]:1027      ESTABLISHED  0
```

Côté serveur, on obtient :

```
?? Taille d'un client = 92
Apres creation de la socket : hSocketEcoute = 120
Creation de la socket OK
Acquisition infos host OK
Bind adresse et port socket OK
Listen socket OK
Accept socket OK
Connexion de ::1
Connexion de ulyssevil.thalassa.inpres.prov-liege.be
Passage boucle de reception
Taile msg recu = 0 et taille atendue = 92
Taill msg = 92 et nbreBytes = 92
Recv socket OK
Demande recue pour le client = Vilvens
--- Coefficient de reduction = 0.54
Nouveau coeff : 084
```

et côté client :

```
?? Taille d'un client = 92
Creation de la socket OK
Acquisition infos cible OK
Connect socket OK
Nom du client : Vilvens
Numero client : C125436
Numero d'article : A5241
Prix : 52.31
Date du dernier achat : 12/5/2008
Coefficient de reduction : 0.54
Send socket OK
Demande envoyee pour le client = Vilvens
```

## 6.7 Le cas de plusieurs interfaces

Il ne faut cependant pas sous-estimer le fait que la machine serveur peut posséder plusieurs interfaces. Dans ce cas, la recherche de l'adresse par `getaddrinfo()` donne en réalité une liste d'adresses, et rien en prouve que celle à laquelle on pense sera en première position : en fait, selon la norme POSIX, elles sont triées selon les spécifications de la RFC 3484, usant de considérations sur les adresses les plus probables à ne pas poser de problèmes. On peut imaginer de parcourir la liste fournie par `getaddrinfo()` :

### TCPv6ITER03.C – parcours de la liste des interfaces

```
struct addrinfo *elemInfosHost; /* Pour la version à plusieurs interfaces */
...
/* VERSION A PLUSIEURS INTERFACE */
/* 3. Acquisition des informations sur l'ordinateur local */
if ( (errgetaddr = getaddrinfo(NULL, "cliser", &indications, &infosHost))!=0)
{
    printf("Erreur d'acquisition d'infos sur le host %d\n", errno);
    system("PAUSE");exit(1);
}
```

```

else printf("Acquisition infos host OK\n");

/* 4. Le système prend connaissance de l'adresse et du port de la socket */
puts("Bind sur les interfaces trouvés");
for (elemInfosHost = infosHost; elemInfosHost;
     elemInfosHost=elemInfosHost->ai_next)
{
    puts("interface trouvé !");
    if (bind(hSocketEcoute, elemInfosHost->ai_addr, elemInfosHost->ai_addrlen)
        == -1)
    {
        printf("Erreur sur le bind de la socket %d\n", errno);
        fprintf(stderr, "bind : %s\n", strerror(errno));
        system("PAUSE");exit(1);
    }
    else printf("Bind adresse et port socket OK\n");
/* 5. Mise à l'écoute d'une requête de connexion */
if (listen(hSocketEcoute,SOMAXCONN) == -1)
{
    printf("Erreur sur le listen de la socket %d\n", errno);
    close(hSocketEcoute); /* Fermeture de la socket */
    exit(1);
}
else printf("Listen socket OK\n");
}
freeaddrinfo(infosHost);
...

```

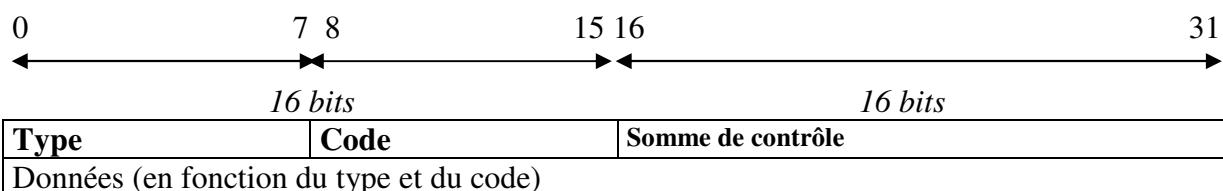
Bien sûr, il n'y aura ici pas de problème parce qu'un seul interface est détecté. Sinon, il faudra paramétriser la socket en SO\_REUSEADDR ;-) ...

## 7. Le protocole ICMPv6

Défini dans la RFC 2463, ICMP reste le protocole de contrôle d'IP. Ses rôles sont :

- ◆ la détection d'erreurs;
- ◆ les tests (ping);
- ◆ la configuration automatique du réseau (découverte des routeurs);
- ◆ mais aussi la gestion des groupes multicast (qui était prise en charge par le protocole **IGMP** (Internet Group Message Protocol) en IPv4).

La structure de la trame ICMPv6 n'est pas neuve :



mais *la numérotation du champ type a été rationnalisée* puisque les valeurs inférieures à 127 sont associées à des erreurs tandis les valeurs supérieures correspondent à des messages d'information. Les valeurs les plus utiles sont :

type	code	nature
1	0	pas de route vers la cible
	1	franchissement du firewall interdit ("raison administrative")
	2	destination hors de portée
	3	adresse inaccessible
	4	port inaccessible, port non attribué
2		paquet trop grand
3	0	temps dépassé : le nombre de sauts est tombé à 0
128		demande d'écho (ping)
129		réponse d'écho (retour de ping)
130		gestion des groupes multicast : enquête d'abonnement
131		gestion des groupes multicast : rapport d'abonnement
132		gestion des groupes multicast : fin d'abonnement
133-137		intégration à l'environnement avec découverte des voisins (neighbor discovery)

## 8. L'utilisation des sockets IPv6 en Java

Les classes **Socket** et **ServerSocket** fonctionnent aussi bien avec des adresses IPv6 que des adresses IPv4. Au moment de l'exécution, le stack utilisé est choisi en fonction du système d'exploitation de la machine hôte et aussi éventuellement des préférences de l'utilisateur si il les a exprimées (au moyen des properties `java.net.preferIPv4Stack` et `java.net.preferIPv6Addresses`). En fait, ces classes manipulent de manière générique des objets **InetAddress** : selon le stack réellement utilisé, il s'agit en fait d'objets instances de l'une des deux classes dérivées **Inet4Address** et **Inet6Address**. On remarquera au sein de cette dernière classe la méthode :

```
public static Inet6Address getByAddress(String host, byte[] addr, int scope_id)
throws UnknownHostException
```

permettant de créer un objet adresse IPv6.

A partir du JDK 1.5, on dispose de la méthode

```
public void setPerformancePreferences (int connectionTime, int latency, int bandwidth)
```

qui permet clairement de faire intervenir la notion de QoS puisque l'on voit apparaître l'expression des **préférences en termes de latence** mais aussi de largeur de bande et de temps de connexion. En fait, ce n'est pas la valeur absolue des trois paramètres qui importe, mais leurs valeurs comparatives : les plus élevées correspondent à ce qui doit être le plus privilégié. Par exemple, (1,0,0) indique que le temps de connexion est considéré comme plus important que les deux autres facteurs, alors que (0,2,1) indique une préférence pour de faibles temps de latence, le temps de connexion étant sans grande importance.



**FIN ?**

Il y aurait encore bien des choses à dire sur la Suite TCP/IP : serveur inetd, programmation dans d'autres protocoles, utilisation des protocoles du niveau application (comme http ou ftp), les "raw sockets", multicast en IPv6. Mais à chaque jour suffit sa peine, à chaque cours ses objectifs. "Ce sera tout pour aujourd'hui" ;-)

## Annexe :

### La programmation TCP/IP en C/Windows

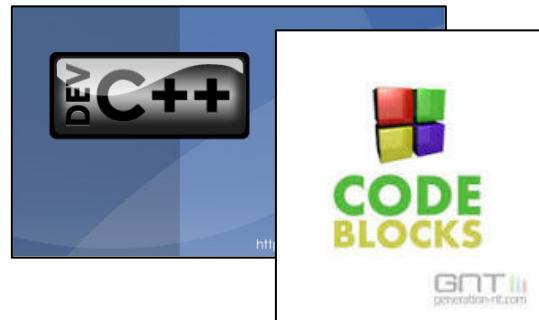


*C'est un métier que de faire un livre,  
comme de faire une pendule.*

(La Bruyère, Les Caractères – Des ouvrages de l'esprit)

#### 1. L'utilisation des sockets

Code::Blocks et l'ancien Dev-C++ sont fournis d'origine avec les librairies des sockets. Par rapport à une plate-forme Unix, seuls quelques aménagements sont nécessaires pour pouvoir développer sous Windows des applications utilisant des sockets.



##### A. Les fichiers headers

Le fichier winsock.h est évidemment nécessaire, comme le windows.h. Les headers TCP/IP sont beaucoup moins répartis dans des répertoires que sous UNIX. Une liste typique d'inclusions dans une application TCP/IP est du style suivant :

```
#include <cstdlib>
// #include <iostream> si nécessaire

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <io.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <windows.h>
#include <winsock.h>           // winsowlk2.h pour IPv6
```

Pour réaliser des portages UNIX-Windows, l'utilisation de directives #if sera la bienvenue :

```
#ifdef SocketsUnix
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <errno.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <netdb.h>      /* getaddrinfo d'IPv6*/
#endif
```

### **B. L'initialisation de Winsock**

On se souviendra (chapitre VI) que l'api Winsock doit être initialisée :

#### **Code d'initialisation de Winsock**

```
...
#ifndef SocketsWindows
// Pour windows
WORD version;
WSADATA infoImpl;
int err;

version = MAKEWORD( 1, 1 );

err = WSAStartup(version, & infoImpl);
if ( err != 0 )
{
    printf ("Problème au startup de Windows");
    exit(1);
}
// Fin Pour Windows
#endif
...
```

### **C. Les paramètres du projet Dev-C++**

Vu ce qui précède, il faudra préciser l'option

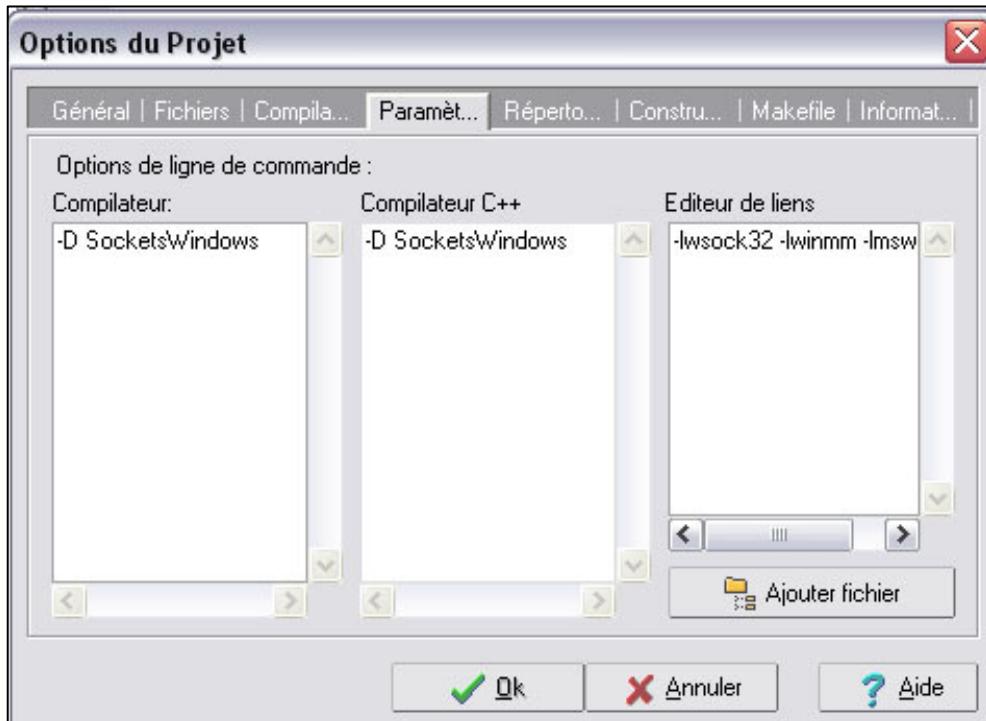
| -D SocketsWindows

pour les compilateurs (C et C++). Il faudra aussi charger les librairies nécessaires pour le linker :

| -lwsock32 -lwinmm -lmswsock

avec en plus, pour IPv6 :

| -lws2\_32



## 2. L'utilisation des threads

### A. Une librairie externe

Il faut tout d'abord se procurer la librairie des threads pour la plate-forme Win32, par exemple sur <http://mirror.facebook.com/sourceware/pthreads-win32/>. Le fichier **pthreads-w32-2-8-0-release.exe** obtenu peut être décompressé pour donner

Adresse	C:\Dev-Cpp\thread-download\Pre-built.2\lib		
Dossiers	Nom	Taille	Type
Pre-built.2	libpthreadGC2.a	88 Ko	Fichier A
include	libpthreadGCE2.a	88 Ko	Fichier A
lib	pthreadGC2.dll	59 Ko	Extension de l'applic...
pthreads.2	pthreadGCE2.dll	110 Ko	Extension de l'applic...
manual	pthreadVC2.dll	85 Ko	Extension de l'applic...
tests	pthreadVC2.lib	29 Ko	Fichier LIB
mkdir	pthreadVCE2.dll	77 Ko	Extension de l'applic...
QueueUserAPCEx	pthreadVCE2.lib	29 Ko	Fichier LIB
driver	pthreadVSE2.dll	85 Ko	Extension de l'applic...
execs	pthreadVSE2.lib	29 Ko	Fichier LIB
testapp			
user			

### B. Modification de l'environnement Dev-C++

Il faut alors ajouter les librairies statiques suivantes au répertoire lib de Dev+C++ :

```
./lib/pthreadVSE2.lib
./lib/libpthreadGC2.a
./lib/libpthreadGCE2.a
./lib/pthreadVC2.lib
```

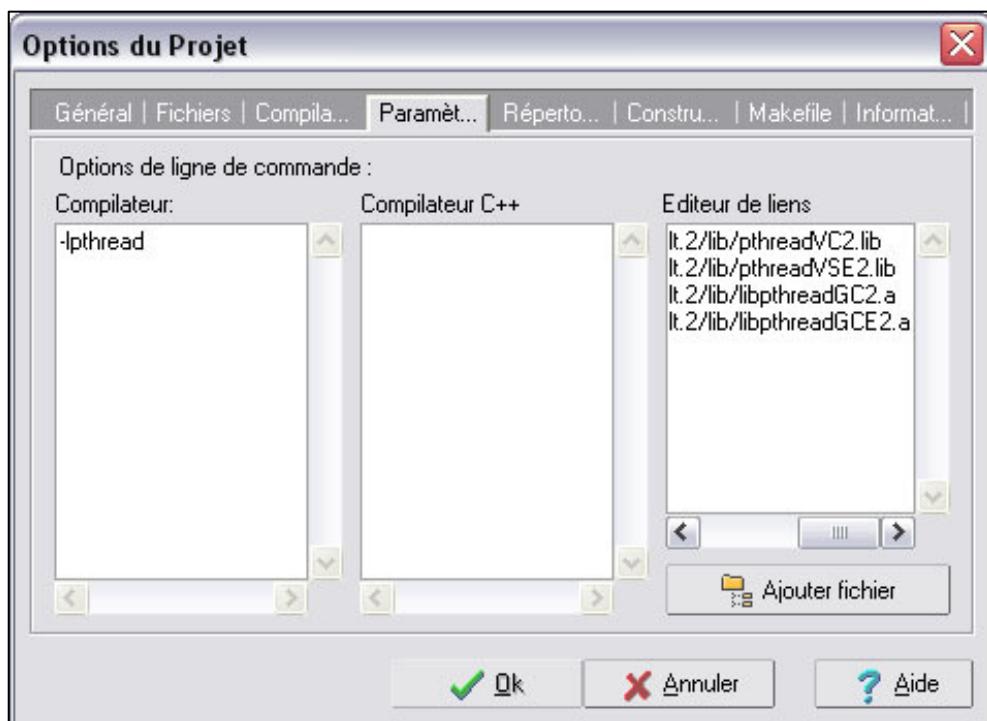
et les librairies dynamiques

pthreadGC2.dll  
pthreadGCE2.dll  
pthreadVC2.dll  
pthreadVCE2.dll  
pthreadVSE2.dll

au répertoire bin de Dev-C++. Enfin, au minimum, il faut ajouter le header **pthread.h** dans le répertoire include de Dev-C++ et évidemment l'inclure dans le source.

### **C. Les paramètres du projet Dev-C++**

Vu ce qui précède, il faudra charger les librairies nécessaires pour le linker :



## Ouvrages consultés

### Ouvrages imprimés

**Arnold, K. & Gosling, J.** The Java Programming Language - Second Edition / The Java Series. Reading, Massachussettes, U.S.A. Addison-Wesley Publishing Company. 1997.

**Cizault, G.** 2007. IPv6 - Théorie et pratique. Paris, Ed. O'Reilly. 2005.

**Campione, M. & Walrath, K.** The Java Tutorial - Object-oriented Programming for the Internet / The Java Series. Reading, Massachussettes, U.S.A. Addison-Wesley Publishing Company. 1997.

**Harold, E. R.** Java I/O. Sebastopol, California, U.S.A. O'Reilly and Associates, Inc. 1999.

**Jamsa, K. & Cope, K.** Programmation Internet C et C++. Paris, France. International Thomson Publishing Company. 1996.

**Janssens, A.** UNIX sous tous les angles. Paris, France. Ed. Eyrolles. 1992.

**Orfali, R., Harkey, D. & Edwards, J.** The Essential Client/Server Survival Guide. New York. Wiley Computer Publishing. 1996.

**Rifflet, J.M.** La communication sous UNIX – Applications réparties. Paris, France. Ediscience international. 1995.

**Ronvaux, A.** Etude des protocoles réseaux implémentés dans .NET. Seraing, Belgique. TFE In.Pr.E.S. 2003.

**Stevens, W.R.** UNIX networking programming – Networking APIs : Sockets and XTI (Volume 1). U.S.A. Prentice Hall Pub. 1998.

**Tanenbaum, R.** Réseaux. Paris/London, France/United Kingdom. InterEditions & Prentice Hall International. 1997.

**Vilvens, C.** Langage Java (I) : Programmation de base. Seraing, Belgique. 2021.

**Vilvens, C.** Langage Java (II) : Programmation avancée des applications classiques et cryptographie. Seraing, Belgique. 2021.

**Vilvens, C.** Les threads POSIX. Seraing, Belgique. 2021.

**Vilvens, C.** Le protocole HTTP et le langage HTML. Seraing, Belgique. 2021.

### Sites Internet

<https://broux.developpez.com/articles/c/sockets/>  
B. Roux. Les sockets en C sur developpez.com

---

**<https://bousk.developpez.com/cours/reseau-c++/TCP/01-premiers-pas/>**

Cours programmation réseau en C++ sur developpez.com

**<https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c/1902751-communiquez-en-reseau-avec-son-programme>**

Programmation client-serveur C/C++ avec interface Qt