Cours d'introduction à TCP/IP

François Laissus
<fr . laissus (à) laissus . fr>

Version du 11 septembre 2005

Les sources de ce document sont éditées sous Unix (*FreeBSD*) à l'aide de l'éditeur de texte `vi`, et gérés avec `cvs`. L'ensemble du processus de fabrication est décrit dans un fichier `Makefile` (commande `make`).

La mise en forme s'effectue grâce au logiciel LaTeX. Les figures sont dessinées sous X Window (X11) à l'aide du logiciel `xfig` et intégrées directement dans le document final sous forme de *PostScript* encapsulé. Les listings des exemples de code C ont été fabriqués à l'aide du logiciel `a2ps` et inclus dans le document final également en *PostScript* encapsulé.

La sortie papier a été imprimée en *PostScript* sur une imprimante de type laser, avec `dvips`. La version *pdf* est une transformation du format *PostScript* à l'aide du logiciel `pspdfm`, enfin la version HTML est traduite directement en HTML à partir du format LaTeX à l'aide du logiciel `latex2html`.

Tous les outils ou formats cités dans ce paragraphe sont en accès ou usage libre, sans versement de droit à leurs auteurs respectifs. Qu'ils en soient remerciés !

Je remercie également **Jean-Jacques Dhénin** et les nombreux lecteurs que je ne connais qu'au travers de leur e-mails, d'avoir bien voulu prendre le temps de relire l'intégralité de ce cours et de me faire part des innombrables erreurs et coquilles typographiques qu'il comportait, merci encore !

Ce support de cours est en accès libre au format HTML à l'url :

http ://www.laissus.fr/cours/cours.html

Où encore au format *dvi*, compressé avec `gzip` :

ftp ://ftp.laissus.fr/pub/cours/cours.dvi.gz

Où encore au format *PostScript* (600 dpi), compressé avec `gzip` :

ftp ://ftp.laissus.fr/pub/cours/cours.ps.gz

Le même, mais compressé avec `bzip2` :

ftp ://ftp.laissus.fr/pub/cours/cours.ps.bz2

Où encore au format *pdf* :

ftp ://ftp.laissus.fr/pub/cours/cours.pdf

Le même, mais compressé avec `zip` :

ftp ://ftp.laissus.fr/pub/cours/cours.pdf.zip

`$Revision: 1.9 $ --- $Date: 2005/02/11 17:27:28 $ --- $Author: fla $`

Table des matières

Table des figures

Liste des tableaux

Préface

Attention ! Ce document n'est qu'un support de cours, c'est à dire qu'il ne remplace pas les documents cités dans la bibliographie qui termine chacun des chapitres qui le composent.

Évidement imparfaites, pleines d'erreurs involontaires, et surtout incomplètes, ces pages réclament avant tout votre indulgence de lecteur bienveillant : " Rien n'est constant, tout change " comme le disait déjà Lao Tseu, 400 ans avant JC. Que dirait-il alors aujourd'hui, concernant des réseaux ! !

Ces cours s'accompagnent de travaux pratiques dont le texte ne figure pas ici, ils sont principalement destinés au Mastère SIO, (Systèmes Informatiques Ouverts — `http ://www.sio.ecp.fr/`), et sont orientés " Unix ".

Ne pas hésiter à me contacter (<fr.laissus(à)laissus.fr>) en cas de doute sur l'usage.

En aucun cas l'auteur ne pourra être tenu responsable des conséquences de l'usage de ce document, qui est fourni tel quel et sans garantie d'aucune sorte. L'usage des informations contenues est donc placé sous la responsabilité pleine et entière du lecteur.

Enfin, si vous pensez que la lecture de ce support vous a apporté quelque chose, que vous avez une remarque à me faire, ou tout simplement me complimenter (ça fait toujours plaisir quoi que l'on puisse en dire ! :) sentez-vous libres de m'envoyer un courrier électronique, je suis toujours ravi d'apprendre que ce travail a pu servir !

Première partie

Introduction à la pile ARPA

Chapitre I

Réseaux locaux

1 Préambule

Ce cours n'est pas un cours général sur les réseaux mais une présentation minimale de cette technologie pour pouvoir aborder le cours de concepts et programmation TCP/IP sous UNIX.

TCP/IP est le protocole le plus répandu dans le monde grâce à l'Internet.

En 1980 il ne comptait que quelques dizaines d'hôtes, en juin 1996 ce nombre était de 12 millions de machines, réparties en près de 500 000 réseaux (Par comparaison, en février 1995, les mêmes chiffres étaient 4 850 000 machines pour plus de 71 000 réseaux locaux).

Actuellement dans le monde (janvier 2003), le nombre de machines[1] directement accessibles sur le réseau serait de 180 000 000 selon l'ISC[2]. Pour la france l'AFNIC propose également quelques statisques[3]...Il n'existe pas de " botin " général du réseau, par contre Bill Cheswick des Bell labs l'a cartographié :

`http ://www.cs.bell-labs.com/who/ches/map/gallery/index.html`

2 Généralités - LANs

2.1 Généralités

Un réseau informatique met en relation des ordinateurs, comme un réseau téléphonique met en relation des personnes.

Des ordinateurs sont dits " en réseaux " dès lors qu'ils partagent une technologie qui leur permet de communiquer ensemble.

Le plus souvent cette technologie se matérialise physiquement par une liaison avec un câble conducteur. Sur ce type de support, un signal électrique véhicule les messages informatiques. Il existe d'autres types de supports en

[1] Source `http ://www.isc.org/ds/`

[2] Internet Software consortium

[3] `http ://www.nic.fr/statistiques/`

pleine expansion comme les liaisons par ondes hertziennes, rayon laser, infrarouge...

Sans connaissance préalable concernant les réseaux informatiques on peut imaginer quantité d'interrogations à partir de cette hypothèse de raccordement :

- Comment reconnaitre un correspondant ?
- Comment dialoguer avec ?
- Comment diffuser l'information à plusieurs correspondants ?
- Comment éviter la cacophonie ?
- Il y a t–il une hiérarchie des machines ?
- Il y a t–il un chef d'orchestre ?
- ...

Toutes ces questions (et bien d'autres) trouveront une réponse dans ce cycle de cours. Ces réponses sont généralement formulées dans un " protocole ", une sorte de mode d'emploi des réseaux. Il y a des centaines de protocoles différents sur l'Internet, certains sont très populaires, d'autres absolument pas.

2.2 Modèle de communication OSI

Le concept de base de tout ce cours est celui de la " commutation de paquets ", une vieille idée de l'informatique[4] contrairement à l'approche par circuits virtuels plus utilisée en téléphonie.

Les données à transmettre d'une machine à une autre sont fragmentées à l'émission en petit blocs de quelques centaines d'octets munis de l'adresse du destinataire, envoyées sur le réseau et ré-assemblées à la réception pour reproduire les données d'origine.

Ce concept facilite le partage des possibilités physiques du réseaux (bande passante) et est parfaitement adapté pour une implémentation sur machines séquentielles travaillant en temps partagé (plusieurs communications peuvent alors avoir lieux simultanément et sur une même machine).

Partant de ce concept, un modèle d'architecture pour les protocoles de communication a été développé par l'ISO (International Standards Organisation) entre 1977 et 1984. Ce modèle sert souvent de référence pour décrire la structure et le fonctionnement des protocoles de communication, mais n'est pas une contrainte de spécification.

Ce modèle se nomme OSI comme " Open Systems Interconnection Reference Model ". Les constituants de ce modèle sont si largement employés qu'il est difficile de parler de réseaux sans y faire référence.

Le modèle OSI est constitué de sept couches. À chaque couche est associée une fonction bien précise, l'information traverse ces couches, chacune y apporte sa particularité.

Cette forme d'organisation n'est pas dûe au hasard, c'est celle sur laquelle les informaticiens ont beaucoup travaillé dans les années soixantes

[4] Conçu par l'Américain Paul Baran et publié en 1964

pour définir les caractéristiques des systèmes d'exploitation.

Une couche ne définit pas un protocole, elle délimite un service qui peut être réalisé par plusieurs protocoles de différentes origines. Ainsi chaque couche peut contenir tous les protocoles que l'on veut, pourvu que ceux-ci fournissent le service demandé à ce niveau du modèle.

Un des intérêts majeurs du modèle en couches est de séparer la notion de communication, des problèmes liés à la technologie employée pour véhiculer les données.

Pour mémoire (*figure I.01*) :

7 La couche application (Application layer) est constituée des programmes d'application ou services, qui se servent du réseau. Ils ne sont pas forcément accessibles à l'utilisateur car ils peuvent être réservés à un usage d'administration.

6 La couche de présentation (Présentation layer) met en forme les données suivant les standards locaux ou particuliers à l'application. Comme, par exemple passer d'une représentation " big endian " ou à une représentation " little endian " ou encore plus complexe comme celle décrite pas les " XdR " (eXternal Data Representation) et qui autorise la transmission de types abstraits de données (structures complexes, arbres, listes chainées, la liste n'est pas limitative...).

De nos jour c'est de plus en plus le XML[5] qui occupe cet espace finalement assez peu normé.

5 La couche de session (Session layer) effectue l'aiguillage entre les divers services (7) qui communiquent simultanément à travers le même ordinateur connecté et le même réseau. Deux utilisateurs d'une même machine peuvent utiliser la même application sans risque d'inter-actions parasites.

4 La couche de transport (Transport layer) garantie que le destinataire obtient exactement l'information qui lui a été envoyée. Cette couche met par exemple en œuvre des règles de renvoi de l'information en cas d'erreur de réception.

3 La couche réseau (Network layer) isole les couches hautes du modèle qui ne s'occupent que de l'utilisation du réseau, des couches basses qui ne s'occupent que de la transmission de l'information.

2 La couche de donnée (Data link layer) effectue le travail de transmission des données d'une machine à une autre.

1 La couche Physique (Physical layer) définit les caractéristiques du matériel nécessaire pour mettre en oeuvre le signal de transmission, comme des tensions, des fréquences, la description d'une prise...

[5] `http ://www.w3.org/XML/`

Modèle en 7 couche de l'OSI

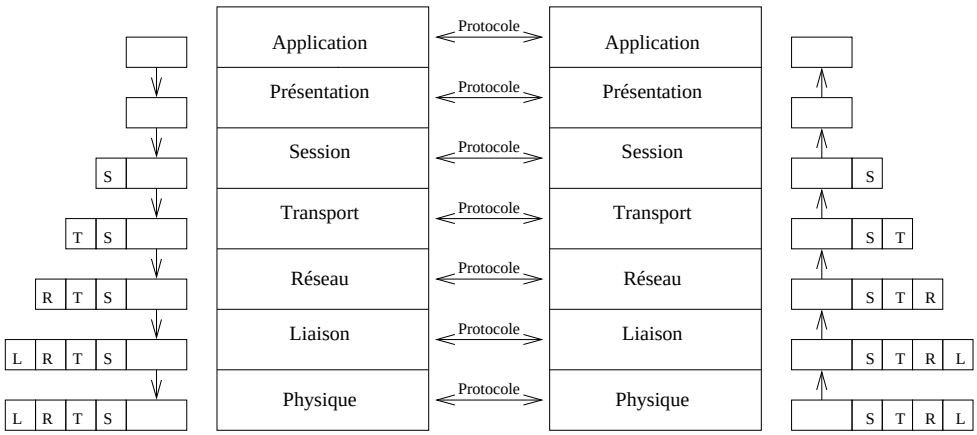


figure I.01

Du niveau 7 de l'application, au niveau 4 du transport, l'information circule dans ce que l'on appelle un " message ", au niveau 3 elle se nomme " packet ", puis " frame " au niveau 2 et " signal " au niveau 1.

Chaque couche ne voit et ne sait communiquer qu'avec la couche qui la précède et celle qui la suit, avec le cas particulier des couches 1 et 7.

L'intérêt de travailler en couches est que lorsque les modalités d'échanges entre chacune d'entres elles sont précisément décrites, on peut changer l'implémentation et les spécificités de la couche elle-même sans que cela affecte le reste de l'édifice.

C'est sur ce principe qu'est bâtie la suite de protocoles désignée par TCP/IP

Quand deux applications A et B discutent entre-elles via le réseau, les informations circulent de la couche 7 vers la couche 2 quand l'application A envoie de l'information sur le réseau, et de la couche 2 vers la couche 7 pour que l'application B reçoive l'information de A.

Le principe de base de cette discussion repose sur le fait que chaque couche du modèle de la machine A est en relation uniquement avec son homologue du même niveau de la machine B.

Quand l'information descend de la couche 7 vers la couche 1, chaque couche " en-capsule " les données reçues avant de les transmettre. Ainsi le volume d'informations s'est accrû de quelques centaines d'octets arrivé à la couche 1.

De manière symétrique, quand l'information remonte de la couche physique vers la couche Application, chaque couche prélève les octets qui lui sont propres, ainsi l'application B ne voit-elle que les octets envoyés par l'application A, sans le détail de l'acheminement.

3 Réseaux locaux

Le problème intuitif et pratique qui se pose est de relier entre elles par un câble toutes les machines qui veulent communiquer : c'est impossible d'abord pour des raisons techniques, le monde est vaste, puis de politique d'emploi des ressources du réseau, tel réseau qui sert à l'enseignement ne doit pas pas perturber le fonctionnement de tel processus industriel.

La conséquence est que les réseaux se développent d'abord en local, autour d'un centre d'intérêt commun, avant de se tourner (parfois) vers l'extérieur.

3.1 Qu'est-ce qu'un LAN ?

Le terme " réseau local " n'est pas clairement défini, cependant tout le monde s'accorde à baptiser de la sorte un réseau, dès lors qu'on lui reconnait les caractéristiques suivantes :

- Cohabitation de plusieurs protocoles,
- Un même média (même câble par exemple) qui raccorde de multiples machines, peut être de caractéristiques différentes,
- Une bande passante élevée, partagée par tous les hôtes
- La capacité de faire du " broadcasting " et du " multicasting ",
- Une extension géographique de moins en moins limitée,
- Un nombre de machines raccordées limité,
- Des relations entre les machines placées sur un mode d'égalité, (et non par exemple sur un mode Maître/Esclave comme dans un réseau dont la topologie serait en étoile),
- Une mise en œuvre qui reste du domaine privé, c'est à dire qui ne dépend pas d'un opérateur officiel de télécommunications.

Notez que les notions de " bande passante " et " nombre limité " (etc...) sont volontairement qualitatives. Elles évoluent rapidement avec le temps.

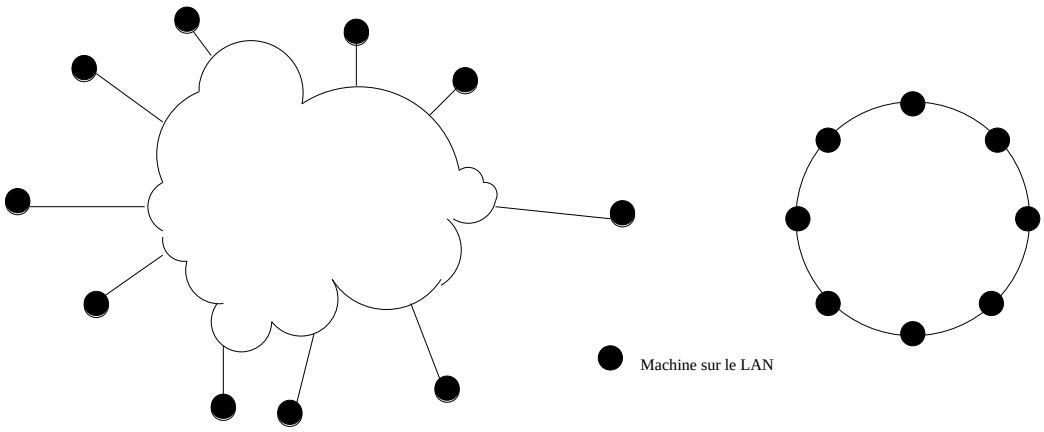


figure I.02

Exemple de types de technologies utilisées dans les LANs :

- Token ring

- IEEE 802 LANs
- Ethernet et Fast-Ethernet
- FDDI (anneau en fibre optique)
- ATM
- 802.11(a,b,g,...)
- ...

3.2 WAN - MAN

Un WAN (Wide Area Network) désigne des ordinateurs connectés entre différentes villes (Metropolitan Area Network) ou pays. La technologie utilisée est traditionnellement moins performante que celle d'un LAN, c'est par exemple une ligne téléphonique louée fonctionnant à 64 kbps, une liaison RNIS, ou encore une liaison transatlantique à 1Mbits/secondes.

Les améliorations technologiques apportées aux LANs permettent de les étendre de plus en plus géographiquement, celles apportées aux WAN augmentent considérablement les bandes passantes, ces deux tendances font que la distinction entre ces deux types de réseaux est de moins en moins claire.

3.3 Communications inter-réseaux

Les réseaux sont appelés à communiquer entres eux et quand cela se produit on parle de communications inter-réseaux (“ internetworking ”).

Le rôle d'une communication inter-réseaux est de gommer les éventuelles différences de technologie d'échange pour permettre à deux réseaux, ou plus, le partage de ressources communes, l'échange d'informations.

Un moyen de faire communiquer deux réseaux distincts passe par l'utilisation de “ gateway ” ou passerelle.

Un tel dispositif est parfois appelé routeur (router), mais c'est un abus de langage.

Les hommes se connectent sur les ordinateurs
Les ordinateurs se connectent sur un réseau
Les réseaux s'inter-connectent dans un “ internet ”

4 Couche 2 - Liaison (Data Link)

La couche 2 la plus populaire est surement celle que l'on nomme abusivement " Ethernet ", du nom du standard publié en 1982 par DEC, Intel Corp. et Xerox. Cette technique repose sur une méthode d'accès et de contrôle dite CSMA/CD (" Carrier Sense, Multiple Access with Collision Detection ").

Elle est devenue tellement populaire qu'on parle d'un câble Ethernet, d'une adresse Ethernet, d'une liaison Ethernet...

Plus tard l'IEEE (" Institute of Electrical and Electronics Engineers ") [6] sous l'instance de son commité 802, publia un ensemble de standards légèrement différents, les plus connus concernant la couche 2 sont 802.2 (Contrôle logique de la liaison – LLC[7]) et 802.3 (CSMA/CD)

Dans le monde TCP/IP, l'encapsulation des datagrammes IP est décrite dans la RFC 894 [Hornig 1984] pour les réseaux Ethernet et dans la RFC 1042 [Postel et Reynolds 1988] pour les réseaux 802.

En règle générale, toute machine utilisant TCP/IP sur ce type de réseaux doit :

1. être capable d'envoyer et de recevoir un paquet conforme à la RFC 894,
2. être capable de recevoir des paquets conformes aux deux standards,
3. Par contre il est seulement souhaitable que cette machine soit capable d'envoyer des paquets conformes à la RFC 1042.

Par défaut le standard est donc celui de la RFC 894, si une machine peut faire les deux, cela doit être configurable.

De nos jours la couche 802.11 (réseau sans fil - wifi) voit sa popularité croître très vite. Elle est basée sur une méthode d'accès assez proche, le CSMA/CA (" Carrier Sense, Multiple Access with Collision Avoidance "). En effet les collisions ne peuvent pas toujours être détectées car les hôtes ne sont pas nécessairement à portée radio directe. Les échanges, quand ils ne sont pas de type " point à point ", passent par un intermédiaire nommé en général " point d'accès " ce qui complique le protocole, et donc la trame, par rapport au CSMA/CD.

4.1 Caractéristiques d'Ethernet

Quelques principes fondamentaux

1. Le support de transmission est un Segment = bus = câble coaxial. Il n'y a pas de topologie particulière (boucle, étoile, etc...).
2. Un équipement est raccordé sur un câble par un " transceiver " : " Transmitter + receiver = transceiver " (coupleur ou transducteur). On parle alors d'une station Ethernet, celle-ci a une adresse unique.

[6] `http ://www.ieee.org/`
[7] " Logical Link Control "

3. Sur le cable circulent des trames, autant de paquets de bits. Il n'y a pas de multiplexage en fréquence, pas de " full duplex " [8]. Une trame émise par une station est reçue par tous les coupleurs du réseau Ethernet, elle contient l'adresse de l'émetteur et celle du destinataire.
4. Un coupleur doit être à l'écoute des trames qui circulent sur le câble. Un coupleur connait sa propre adresse, ainsi si une trame lui est destinée il la prend, sinon il n'en fait rien.
5. Une station qui veut émettre attend que toutes les autres stations se taisent. Autrement dit, si le câble est libre elle envoie sa trame, sinon elle attend.

   Si deux stations émettent en même temps il y a collision. Les deux trames sont alors inexploitables, les deux (ou plus) stations détectent ce fait et reémettent ultérieurement leur paquet en attente.
6. Un réseau Ethernet est donc un réseau à caractère probabiliste car il n'y a pas de chef d'orchestre pour synchroniser les émissions. Cette absence conduit à dire que c'est un réseau égalitaire, une sorte de réunion sans animateur entre personnes polies

En conclusion, la technologie Ethernet est simple, sa mise en œuvre se fait à faible coût. Points à retenir :

- Simplicité et faible coût
- Peu de fonctions optionnelles
- Pas de priorité
- Pas de contrôle sur l'attitude des voisins
- Débit d'au moins 10Mb/s (jusqu'à 1000Mb/s théorique).
- Performances peu dépendantes de la charge, sauf en cas de collisions trop importantes.

Format d'une " Frame Ethernet "

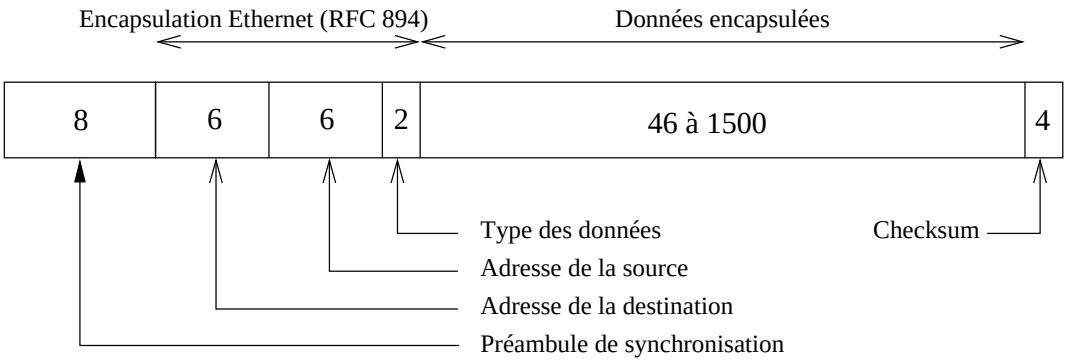


figure I.03

[8] les cartes Ethernet modernes utilisent 4 fils au lieu de deux et offrent ansi des possibilités de " full duplex " que n'avaient pas leurs ancêtres des années 80

Quelques considérations en vrac :
- Dû au débit global de 10Mbits/seconde, le débit est de 10 bits par micro-seconde (en gros un facteur 1000 avec un cpu).
- Une trame a une longueur minimale (72) et une longueur maximale (1526). Si les données ne sont pas assez longues (46 octets) des caractères de remplissage sont ajoutés (“ padding ”).
- Les octets circulent du premier octet du préambule au dernier octet du CRC.
  A l'intérieur de chaque octet le premier bit envoyé est celui de poids faible, etc..
- Le préambule et le SFD (“ Start Frame Delimiter ”) servent à la synchronisation.
- Adresses d'origine et de destination sont celles respectivement de la machine émettrice et de la machine destinatrice.
  Remarque importante : il faut connaître l'adresse de son correspondant pour pouvoir lui envoyer un paquet ! À ce stade de l'exposé on ne sait pas encore comment faire quand on ignore cette information.
- Le champ “ type ” est deux octets qui désignent le type des données encapsulées :

Type	*Données*
0800	IP
0806	ARP
0835	RARP
6000	DEC
6009	DEC
8019	DOMAIN
...	...

Adresses IEEE 802.3 ou Ethernet

Pour ces deux standards, l'adresse est codée sur 6 octets soit 48 bits. Pour un hôte sur un réseau, cette adresse est ce que l'on appelle son adresse physique (“ hardware addresse ”) par opposition à son adresse logique qui interviendra lors de l'examen de la couche 3.

En fait cette adresse est divisée en deux parties égales, les trois premiers octets désignent le constructeur, c'est le OUI (“ Organizationally Unique Identifier ”) distribué par l'IEEE [9] les trois derniers désignent le numéro de carte, dont la valeur est laissée à l'initiative du constructeur qui possède le préfixe.

L'IEEE assure ainsi l'unicité de l'attribution des numéros de constructeurs, par tranches de 2^{24} cartes[10]

Chaque constructeur assure l'unicité du numéro de chaque carte fa-

[9] `http ://standards.ieee.org/regauth/oui/index.shtml`

[10] La liste à jour est accessible à cette url `http ://standards.ieee.org/regauth/oui/oui.txt` ou à la fin de la RFC 1700 (page 172) “ Ethernet vendors address components ”

briquée. Il y a au maximum 2^{24} cartes par classe d'adresses.

Cette unicité est primordiale car le bon fonctionnement d'un LAN requiert que toutes les stations aient une adresse physique différente. Dans le cas contraire le réseau et les applications qui l'utilisent auront un comportement imprévisible le rendant impraticable.

Nous aurons l'occasion de rencontrer à nouveau ce soucis d'unicité de l'adresse physique lorsque nous examinerons les protocoles ARP et RARP (cf cours ARP/RARP pages 52 et 55) et avec CARP (“ Common Address Redundancy Protocol ”) lorsque nous parlerons des hôtes virtuels, page 177.

Exemple d'adresse physique en représentation hexadécimale :

`08:00:09:35:d5:0b`	`08:00:09` est attribué à la firme Hewlett-Packard `35:d5:0b` est l'adresse de la carte

D'autres constructeurs, saisis au hasard des réseaux :

`00:11:24`	Apple Computer
`00:00:0C`	Cisco Systems, Inc.
`00:06:5B`	Dell Computer Corp.
`08:00:20`	Sun Microsystems
`AA:00:04`	Digital Equipment Corporation
`00:10:5A`	3Com Corporation
...	...

Attention, si tous les bits de l'adresse sont à 1, c'est une adresse de broadcast. Dans ce cas toutes les stations d'un réseau sont destinatrices du paquet. Il existe une adresse particulière `01:00:5E`, non dédiée à un constructeur car dite de “ multicast ” que nous examinerons page 40. Toutes les autres adresses qui ne sont ni du type broadcast ni du type multicast sont du type “ unicast ”.

4.2 Différences Ethernet - 802.2/802.3

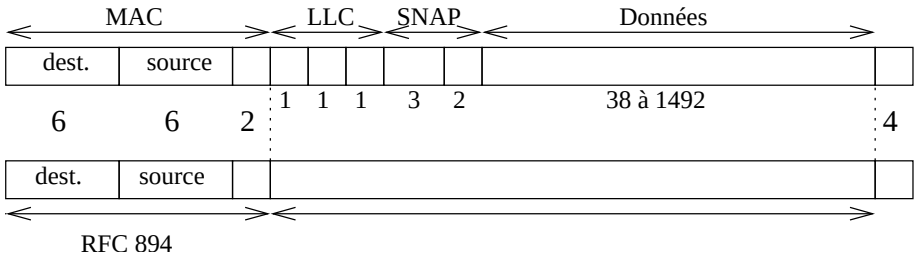


figure I.04

– On remarque que le champ " taille " de la trame 802.3 est à la place du champ " type " de la trame Ethernet. La différenciation s'effectue à partir de la valeur de ces deux octets. On remarque également que le commité 802 a choisi de subdiviser la couche 2 ISO en deux sous couches : MAC et LLC.
– Tous les numéros de protocole sont supérieurs à 1500[11] qui est la longueur maximale des données encapsulées. Donc une valeur inférieure ou égale à ce seuil indique une trame 802.3.

MAC = " Medium Access Control "

LLC = " Logical Link Control "

5 Interconnexion - Technologie élémentaire

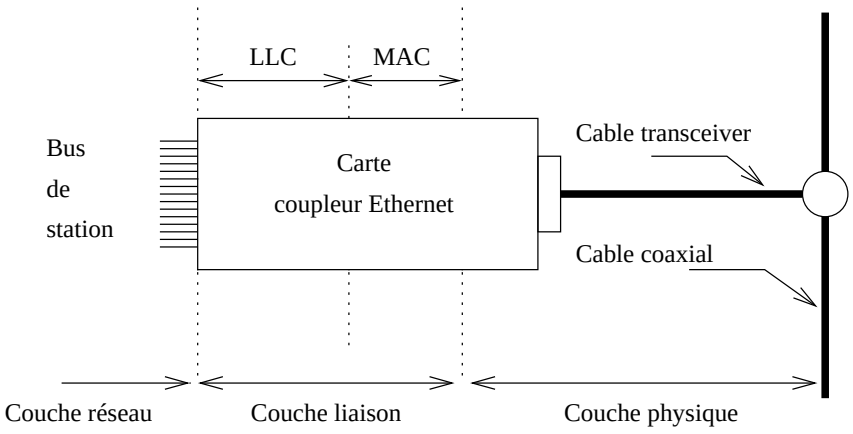


figure I.05

[11]Le plus petit numéro de protocole est celui d'IP : 0800 hexadécimal. Ce qui fait en décimal : $8 \times 16^2 + 0 \times 16^1 + 0 \times 16^0 = 2048$

L'interconnexion ne se limite pas au niveau Ethernet.

Quelques notions de technologie de base et donc très succintes sont nécessaires pour bien comprendre la suite de ce cours.

5.1 Raccordement

Figure I.06 l'hôte est raccordé à l'aide d'une prise de type " vampire " et d'un " transceiver ".

Dans cette technologie de raccordement, le support est un gros câble jaune, dit encore " Thick Ethernet " ou Ethernet standard, ou encore 10Base5 (10 comme 10Mbits/s, Base comme " Baseband ", 5 comme 500 mètres).

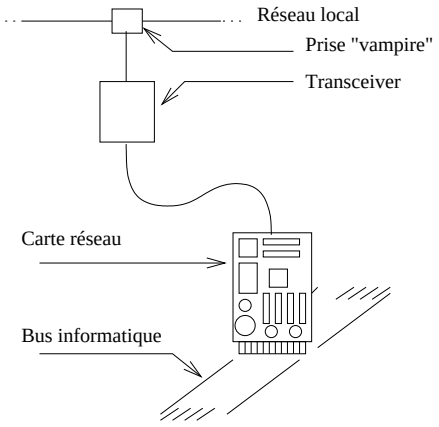


figure I.06

10Base5

Quelques particularités du 10Base5 :

- Longueur maxi est 500 mètres, pour un maximum de 100 stations.
- C'est une " vieille " technologie très bien normalisée mais dépassée.
- Pas de perturbation quand on ajoute une station : la pose d'une nouvelle prise n'interrompt pas la continuité du réseau.
- Coût non négligeable.
- Déplacement d'une station non aisé, en plus on perd la prise vampire, elle reste sur le câble.

Pour les câblages rapides on préfère le 10Base2 ou " Thin Ethernet " ou encore Ethernet fin (2 comme 200 mètres).

10Base2

Quelques particularités du 10Base2 :

- Longueur maxi de 185 mètres avec un maximum de 30 stations.
- La topologie impose de mettre les stations en série avec un minimum de 0.5 mètre entre chaque.
- Le raccord se fait avec un " transceiver " en T (BNC bien connu des électroniciens).
- Il faut un bouchon de 50 ohms à chaque extrémité du réseau (2).
- Technique très bon marché, souple, les cartes intègrent le transducteur.
- Il faut rompre la continuité du réseau pour ajouter une nouvelle station, ce qui l'empêche de fonctionner durant l'opération. C'est un in-

convénient de taille sur un réseau très utilisé.
- Cette technique est en outre assez sensible aux perturbations électromagnétiques.

Les désavantages du 10Base2 imposent généralement l'usage du 10BaseT dans toute structure dépassant quelques machines (5 à 10). Le 10BaseT règle définitivement le problème de l'ajout ou du retrait d'une machine sur le LAN (T comme " Twisted Pair " ou paires torsadées).

Cette technique impose l'usage d'une boite noire réseau nommée " HUB "[12] ou moyeu. Celle-ci simule la continuité dans le cas du retrait d'une station.

10BaseT

Quelques particularités du 10BaseT :
- Une double paire torsadée de câble suffit.
- La longueur maximale entre le moyeu et la station est de 100 mètres.
- Le moyeu impose une architecture en étoile.
- Le raccordement au transducteur se fait à l'aide d'une prise du type RJ45, très fragile (ne pas marcher dessus ! :). Le raccordement du HUB (page 17) au reste du réseau se fait par 10Base2, en fibre optique, ou tout simplement par chaînage avec un autre HUB (" Daisy chain ").
- Cette technique est d'une très grande souplesse d'utilisation elle impose néanmoins l'acquisiton de HUB, très peu onéreux de nos jours.
- Cette technique des paires torsadées est très sensible aux perturbations électromagnétiques. électromagnétiques.

Aujourd'hui le 100BaseT équipe la majeur partie des équipements professionnels, 100 comme 100 Mbits/s.

Enfin la fibre optique est utilisée de plus en plus souvent pour effectuer les liaisons point à point.

Fibre optique

Quelques particularités de la fibre optique :
- La plus utilisée est la fibre multimode 62.5/125.0 μm
- Usage d'un transducteur optique pour assurer la transformation entre le signal lumineux (un laser) et le signal électrique.
- La distance maximale entre deux points est 1,5 km.
- La fibre est insensible aux perturbations électromagnétiques, elle permet en outre le câblage de site important (plusieurs km^2).
- La fibre permet d'atteindre des vitesses de transmission supérieures aux 10Mbits/100Mbits/1000Mbits maintenant courants sur des paires de fils en cuivre.
- Les nouvelles technologies issues des recherches les plus récentes promettent des fibres multifréquences (1024 canaux par fibre) avec pour

[12] Voir au paragraphe 5.3 page 17

chaque canal une bande passante de plusieurs giga-octets. Ces nouveaux médias auront une bande passante de plusieurs téra-octets par secondes...

– Son principal désaventage est un coût élevé au mètre (de l'ordre d'une dizaine d'€ pour un câble d'un mètre cinquante) et la nécessité d'avoir des transducteurs au raccordement de tous les appareils contenant de l'électronique (serveur, switch, routeur). Un tel module peut coûter de l'ordre de 500 à 1000 € ...

Conclusion

Construire un réseau local consiste à juxtaposer des composants de base très bien maitrisé, une sorte de mécano car tous les supports sont mixables.

Ne plus installer les technologies les plus anciennes 10Base5, 10Base2 ou même 10BaseT, préférer l'usage du 100BaseT ou du 1000BaseT qui sont devenus un standard courant du précablage.

En effet le câblage constitue les fondations d'un réseau, le faire proprement d'emblé évite une source continuelle d'ennuis pas la suite ! Les besoins en bande passante d'aujourd'hui ne préfigurent sans doute pas encore les besoins de demain (vidéo haute définition sur tous les postes...), il faut donc prévoir très large dès la conception initiale.

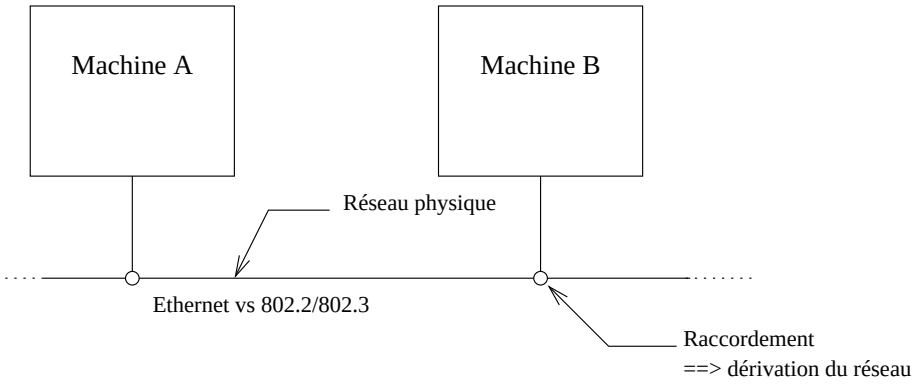


figure I.07

5.2 Répéteur

À une technologie particulière correspond forcément des limitations dues aux lois de la physique. Par exemple en technologie Ethernet la longueur maximale d'un brin ne peut pas excéder 180 mètres. Pour pallier à cette déficience on utilise des répéteurs (“ repeaters ”).

Répéteurs :

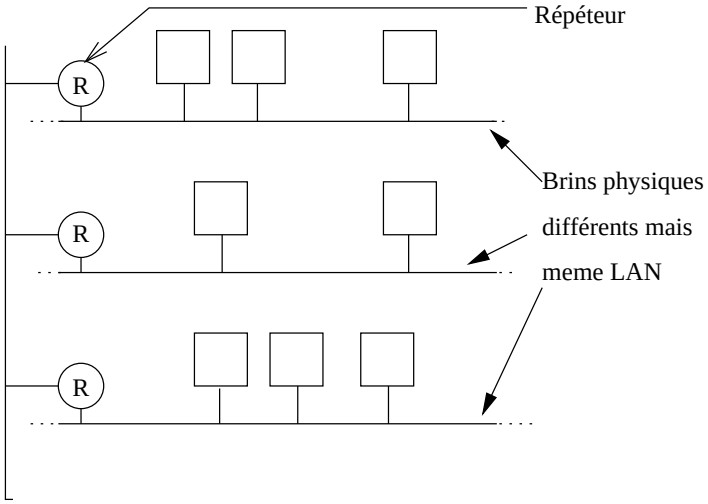


figure I.08 — Plusieurs répéteurs mais toujours le même lan

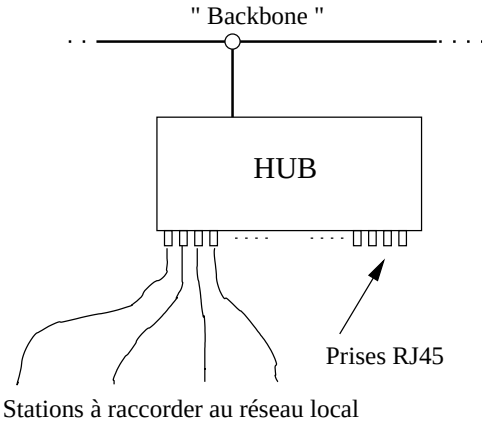
- Agit uniquement au niveau de la couche 1 ISO, c'est un " amplificateur de ligne " avec ses avantages et aussi l'inconvénient de transmettre le bruit sans discernement : il n'y a aucun filtrage sur le contenu.
- Relie deux brins d'une même technologie en un seul LAN car les trames sont reproduites à l'identique.
- En 10Base5, l'usage d'un répéteur fait passer la limite des 500 mètres à 1000 mètres...
- Il n'y a aucune administration particulière, sinon de brancher la boite noire à un emplacement jugé pertinent.
- C'est un élément " bon marché ".

5.3 Concentrateur

Un concentrateur (ou " HUB ", moyeu) :

- Est aussi nommé étoile ou multirépéteur.
- Les HUB n'ont pas d'adresse Ethernet, sauf certains modèles évolués, gérables à distance (TELNET,SNMP,...). On parle alors de " hubs intelligents " parcequ'ils permettent d'associer des ports entres-eux.

figure I.09 — Concentateur



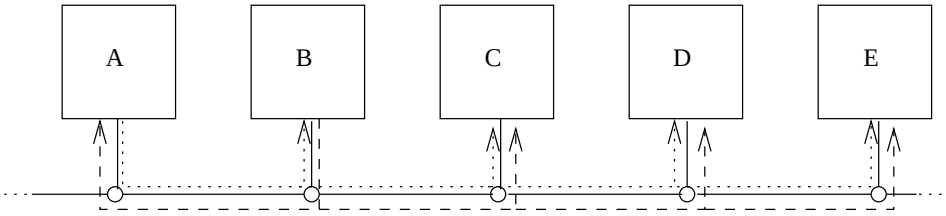
Un hub assure la continuité du réseau sur chacune de ses prises, que l'on y branche ou pas un hôte. En cela il agit uniquement au niveau de la couche 1 ISO. Il ne limite pas le nombre de collisions et n'améliore pas l'usage de la bande passante. Son seul intérêt est de donc permettre le branchement ou le débranchement des stations sans perturber le fonctionnement global du réseau.

Les hubs peuvent être chaînés entres-eux ; souvent ils sont reliés au backbone local par une autre technologie que la paire torsadée (fibre optique...).

Dans le cas de " hubs intelligents " les ports sont associés les uns aux autres par groupes de fonctionnement.

5.4 Ponts

La technologie CSMA/CD atteint vite ses limites quand le réseau est encombré. Une amélioration possible quand on ne peut pas changer de technologie (augmentation du débit) est d'utiliser un ou plusieurs ponts (" bridges ") pour regrouper des machines qui ont entre-elles un dialogue privilégié.
Dialogue entre deux stations, sans pont :



Le dialogue entre A et B perturbe l'éventuel dialogue entre D et E.

figure I.10 — Dialogue sans pont

De nos jours le pont en tant que tel est de moins en moins utilisé par contre le principe de son fonctionnement se retrouve, entres autres, dans les commutateurs (paragraphe suivant) et dans les points d'accès sans fil (" wireless ").
Dialogue entre deux stations, avec pont :

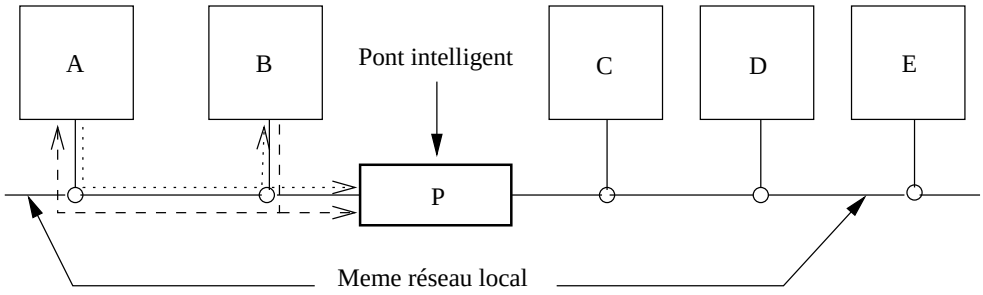


figure I.11 — Dialogue avec pont

Un pont :

- Agit au niveau de la couche 2 ISO, donc au niveau de la trame physique. Son action est plus que physique elle est aussi logique puisqu'il y a lecture et interprétation des octets véhiculés. Le résultat de ce travail logique (apprentissage) consiste à isoler le trafic sur certains tronçons d'un LAN. À cause de ce travail on parle généralement de " ponts intelligents " ou de " ponts transparents " car la phase d'apprentissage est automatique !
- Réduit le taux de collisions en réduisant le trafic inutile, donc améliore l'usage de la bande passante. Sur la *figure I.11* les machines A et B peuvent dialoguer sans pertuber le dialogue entre les machines D et E. Par contre dans le cas d'un dialogue entre A et E le pont ne sert à rien.
- Moins cher qu'un routeur et plus rapide (services rendus moins complets).
- Relie deux segments (ou plus) en un seul LAN, les trames transmises sont reproduites à l'identique.
- Un pont contient un cpu, il est en général administrable à distance car on peut agir sur la table de filtrages (ajout, contraintes de filtrages, etc...). Dans ce cas un pont a une adresse Ethernet.
- Les ponts interdisent que les réseaux aient des boucles, un protocole nommé STP (" Spanning Tree Protocol ") désactive automatiquement le ou les ponts qui occasionne(nt) un bouclage des trames.
- Il existe des ponts entre Ethernet et Token-ring, on parle alors de " ponts à translations ".
- Attention, un pont ne s'occupe que des adresses de type unicast, il filtre ne pas les types broadcast et multicast.
- On peut remarquer que dans le cas de figure ou le trafic est strictement contenu d'un coté et de l'autre du pont, alors la bande passante globale du LAN est multipliée par deux. Bien sûr cette remarque n'est plus valable dès lors qu'une trame franchit le pont.

5.5 Commutateurs

Aligner des stations sur un même réseau local constitue une première étape simple et de faible coût pour un réseau local d'entreprise. Le revers d'une telle architecture est que le nombre de collisions croît très vite avec le trafic, d'où une baisse très sensible de la rapidité des échanges dûe à ce gaspillage de la bande passante.

L'usage de ponts peut constituer une première solution mais elle n'est pas totalement satisfaisante dans tous les cas de figure, comme nous avons pu le remarquer au paragraphe précédent.

Depuis plus d'une dizaine d'années est apparue une technologie nommée " Intelligent Switching Hub " (ISH) – commutateur intelligent – qui utilise le concept de commutation parallèle et qui a révolutionné l'organisation des réseaux locaux.

D'aspect extérieur ces équipements se présentent comme un hub mais ont en interne un cpu suffisamment puissant et un bus interne de données suffisamment rapide pour mettre en œuvre une logique de commutation raffinée.

Lorsqu'une trame se présente sur l'un des ports du commutateur elle est (ou n'est pas) re-routée vers un autre port en fonction de l'adresse physique du destinataire. Il existe plusieurs différences entre un pont et un commutateur :

- Un commutateur peut mettre simultanément plusieurs ports en relation, sans que le débit de chacun en souffre. Par exemple un commutateur de 4 ports en 10BaseT peut supporter deux connexions port source/port destination simultanées à 10 Mbit/s chacune, ce qui donne un débit global de 20 Mbit/s.
  D'un point de vue plus théorique, un commutateur à N ports à 10 Mbit/s chacun a un débit maximum de $N \times 10/2 = 20 Mbit/s$.
- Si une trame est à destination d'un port déjà occupé, le commutateur la mémorise pour la délivrer sitôt le port disponible.
- Un commutateur fonctionne comme un pont pour établir sa carte des adresses mais il peut aussi travailler à partir d'une table préconfigurée.
- Un commutateur peut fonctionner par port (une seule station Ethernet par port) ou par segment (plusieurs stations Ethernet par port).

Avec un commutateur, il est aisé d'organiser un réseau en fonction de la portée des serveurs des postes clients associés. La *figure I.12* illustre ce principe :

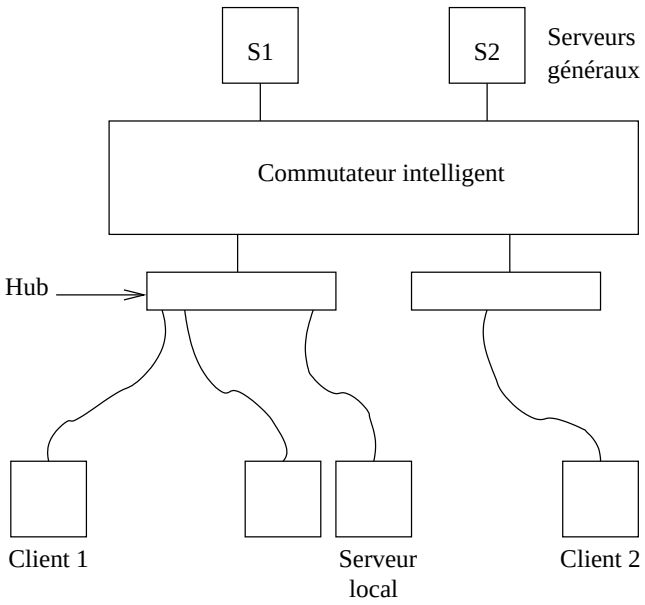


figure I.12 — Commutateur

Le trafic réseau entre le " client 1 " et le serveur " S2 " ne perturbe pas le trafic entre le " client 2 " et le serveur " S1 ". De même le trafic entre le " client 1 " et le " serveur local " n'est pas vu du " client 2 ".

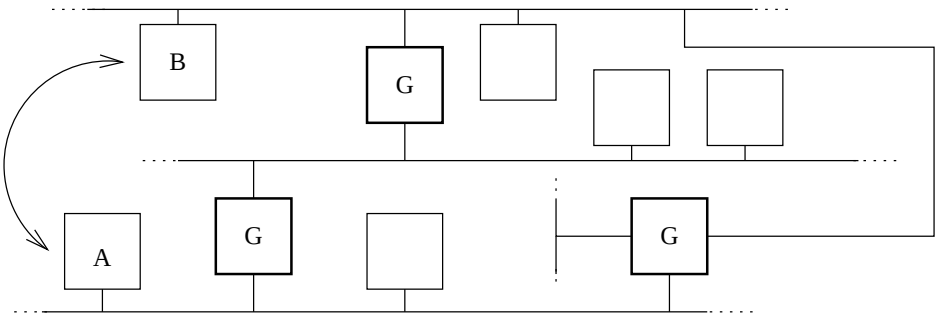
Les commutateurs étiquettent les trames avec un identificateur du VLAN auquel elles appartiennent. Cette étiquette se résume par deux octets ajoutés dans la trame, selon les recommandations du comité 802 (norme 802.1Q).

5.6 Passerelles — Routeurs

Pour raccorder deux LANs non forcément contigus il faut faire appel à ce que l'on désigne " une passerelle " (" gateway "). Son rôle est de prendre une décision sur la route à suivre et de convertir le format des données pour être compatible avec le réseau à atteindre (en fonction de la route).

Souvent, et c'est le cas avec TCP/IP, la fonction de conversion n'est pas utilisée, la fonction de routage donne alors son nom à l'appareil en question (éponyme), qui devient un " routeur " (" router ").

Le problème du routage entre A et B :



Plusieurs chemins sont possibles pour aller de A à B, d'où la nécessité d'une stratégie.

figure I.13 — Fonction routage

La fonction passerelle consiste aussi en traduction de protocoles :

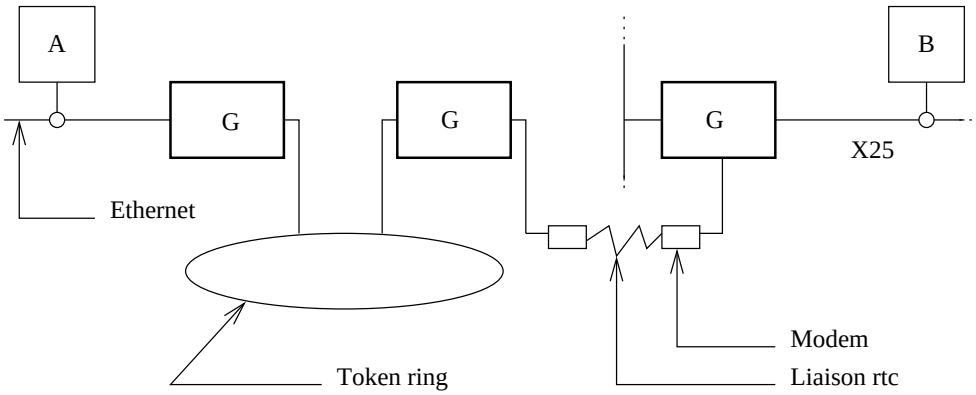


figure I.14 — Traduction de protocoles

Un routeur :

- Agit au niveau de la couche 3. Il prend des décisions de destination.
- Possède au moins deux interfaces réseau (pas forcément identiques).
- Contient un cpu et un programme très évolué, il est administrable à distance.

– Remplit également les fonctions d'un pont (B-routeur) mais les brins ainsi reliés ne forment en général plus un LAN car les adresses physiques contenues dans les trames ne servent plus à identifier le destinataire. Il faut une autre adresse qui dépend de la pile au-dessus (exemple adresse IP). Il existe cependant des possibilités de simuler un même LAN bien que les trame traversent un routeur (cf cours ARP (page 52)).

6 Bibliographie

Pour en savoir plus :

RFC 0894 S C. Hornig, " Standard for the transmission of IP datagrams over
Ethernet networks ", 04/01/1984. (Pages=3) (Format=.txt)

RFC 1042 S J. Postel, J. Reynolds, " Standard for the transmission of IP datagrams over IEEE 802 networks ", 02/01/1988. (Pages=15) (Format=.txt) (Obsoletes RFC0948)

– Radia Perlman — " Interconnections – Briges and Routeurs " — Addison–Wesley
– Radia Perlman — " Interconnections Second Edition " – Briges, Routeurs, Switches, and Internetworking Protocoles — Addison–Wesley

Chapitre II

Introduction à IP

1 TCP/IP et l'Internet - Un peu d'histoire

En 1969 aux États Unis, l'agence gouvernementale DARPA lance un projet de réseau expérimental, basé sur la commutation de paquets. Ce réseau, nommé ARPANET, fut construit dans le but d'étudier les technologies de communications, indépendamment de toute contrainte commerciale[1]

Un grand nombre de techniques de communication par modems datent de cette époque.

L'expérience d'ARPANET est alors si concluante que toutes les organisations qui lui sont rattachées l'utilisent quotidiennement pour pour leurs messages de service.

En 1975, le réseau passe officiellement du stade expérimental au stade opérationnel.

Le développement d'ARPANET ne s'arrête pas pour autant, les bases des protocoles TCP/IP sont développés à ce moment, donc après que ARPANET soit opérationnel.

En Juin 1978 Jon Postel[2] définit IPv4 et en 1981 IP est standardisé dans la RFC 791 [J. Postel 1981].

En 1983 les protocoles TCP/IP sont adoptés comme un standard militaire et toutes les machines sur le réseau commencent à l'utiliser. Pour faciliter cette reconversion, la DARPA demande à l'université de Berkeley d'implémenter ces protocoles dans leur version (BSD) d'unix. Ainsi commence le mariage entre Unix et les protocoles TCP/IP.

L'apport de l'Université de Berkeley est majeur, tant au niveau théorique (concept des sockets) qu'au niveau de l'utilisateur, avec des utilitaires très homogènes avec ceux déjà existants sous Unix (rcp, rsh, rlogin...).

---

[1] Lancé en France en 1972, le projet " Cyclades ", sous la responsabilité de Louis Pouzin, était également basé sur la commutation de paquets et l'usage de datagrammes. Il reliait quelques grands centres universitaires en France (Lille, Paris, Grenoble,...) et en Europe. Il est resté opérationnel jusqu'en 1978, date à laquelle faute de crédit il a été abandonné au profit de X25, préféré par les opérateurs de télécoms nationaux.

[2] Jon Postel est décédé le 16 Octobre 1998 à l'âge de 55 ans, c'est le premier pionner de l'Internet décédé, on peut consulter par exemple : `http ://www.isi.edu/postel.html`

Depuis cette époque, un nouveau terme est apparu pour désigner cette interconnexion de réseaux, l'Internet, avec un " i " majuscule.

Le succès de cette technologie est alors très important et suscite un intérêt croissant de la part d'acteurs très divers, et en particulier La " National Science Foundation " qui y voit un intérêt majeur pour la recherche scientifique et soutient donc ce nouveau moyen de mettre en communication tous les chercheurs.

Depuis 1990, ARPANET n'est plus, pourtant le terme Internet demeure il désigne maintenant un espace de communication qui englobe la planète tout entière. Des centaines de milliers de sites aux États Unis, en Europe et en Asie y sont connectés.

Depuis 1994, l'Internet s'est ouvert au commerce, surtout avec l'apparition en 1991 d'un nouvel outil de consultation, le " World Wide Web " ou " Web " et ses interfaces populaires : Mosaic[3], Netscape, Mozilla, Firefox, Konqueror...La liste n'est pas exhaustive !

Depuis 1995, pour faire face à sa popularité fortement croissante et aux demandes de transactions sécurisées, le protocole évolue et une nouvelle version, la version 6 (IPng), est définie et en cours de déploiement expérimental[4].

Les protocoles désignés par TCP/IP ont également envahi les réseaux locaux eux-mêmes, car il est plus facile d'utiliser les mêmes protocoles en interne et en externe.

Pour les utilisateurs, l'accès à l'Internet est possible à l'aide d'une collection de programmes spécialisés si faciles à utiliser que l'on peut ignorer tout (ou presque) de leur fonctionnement interne.

Seul les programmeurs d'applications réseaux et les administrateurs de systèmes ont besoin d'en connaître les arcanes.

Les services réseaux les plus populaires sont principalement :

- Le courrier électronique qui permet l'échange de messages entres usagers.
- Les dizaines de milliers de forums de discussion (" news ").
- Le transfert de fichiers entre machines (" ftp " et ses dérivés comme " fetch ", " wget ", " curl "...).
- Le "remote login ", ou ses équivalents cryptés (" ssh ", qui permet à un utilisateur de se connecter sur un site distant, depuis son poste local.
- Les serveurs inter-actifs. Les " anciens " se nommaient archie, gopher, veronica, wais... Désormais ils sont rendus obsolètes par le " web " (protocole `http`).
- Puis maintenant la radio, la vidéoconférence, la réalité virtuelle avec le VRML, le " chat ", les bourses d'échanges point à point...

En conclusion de ce paragraphe sur l'historique on peut dire que l'Internet est une collection apparemment anarchique (il n'y a pas de structure hiérarchique et centralisée) de réseaux inter-connectés et appartenant à divers

[3] `http ://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html`

[4] Un discours de ministre a déjà tenté d'ailleurs de lui donner un impulsion nouvelle :`http ://www.recherche.gouv.fr/discours/2002/dhourtin.htm`

propriétaires.

On distingue trois niveaux : les réseaux au sein des organisations (lans), les réseaux régionaux et les réseaux de transit.

Le site de l'association Fnet indique quelques pointeurs intéressants sur l'historique de l'Internet[5] (en anglais).

2 Caractéristiques de TCP/IP

Le succès de TCP/IP, s'il vient d'abord d'un choix du gouvernement américain, s'appuit ensuite sur des caractéristiques intéressantes :

1. C'est un protocole ouvert, les sources (C) en sont disponibles gratuitement et ont été développés indépendamment d'une architecture particulière, d'un système d'exploitation particulier, d'une structure commerciale propriétaire. Ils sont donc théoriquement transportables sur n'importe quel type de plate-forme, ce qui est prouvé de nos jours.
2. Ce protocole est indépendant du support physique du réseau. Cela permet à TCP/IP d'être véhiculé par des supports et des technologies aussi différents qu'une ligne série, un câble coaxial Ethernet, une liaison louée, un réseau token-ring, une liaison radio (satellites, " wireless " 802.11a/b/g), une liaison FDDI 600Mbits, une liaison par rayon laser, infrarouge, xDSL, ATM, fibre optique, la liste des supports et des technologies n'est pas exhaustive...
3. Le mode d'adressage est commun à tous les utilisateurs de TCP/IP quelle que soit la plate-forme qui l'utilise. Si l'unicité de l'adresse est respectée, les communications aboutissent même si les hôtes sont aux antipodes.
4. Les protocoles de hauts niveaux sont standardisés ce qui permet des développements largement répandus et inter-opérables sur tous types de machines.

La majeurs partie des informations relatives à ces protocoles sont publiées dans les RFCs (Requests For Comments). Les RFCs contiennent les dernières versions des spécifications de tous les protocoles TCP/IP, ainsi que bien d'autres informations comme des propositions d'améliorations des outils actuels, la description de nouveaux protocoles, des commentaires sur la gestion des réseaux, la liste n'est pas exhaustive.

[5] `http ://www.fnet.fr/history/`

3 Comparaison TCP/IP — ISO

La suite de protocoles désignée par TCP/IP, ou encore " pile ARPA ", est construite sur un modèle en couches moins complet que la proposition de l'ISO. Quatre couches sont suffisantes pour définir l'architecture de ce protocole.

4 Couche Application (Application layer).

3 Couche Transport (Transport layer).

2 Couche Internet (Internet layer).

1 Couche interface réseau (Network access layer).

0 Matériel (n'est pas une couche comprise dans le protocole).

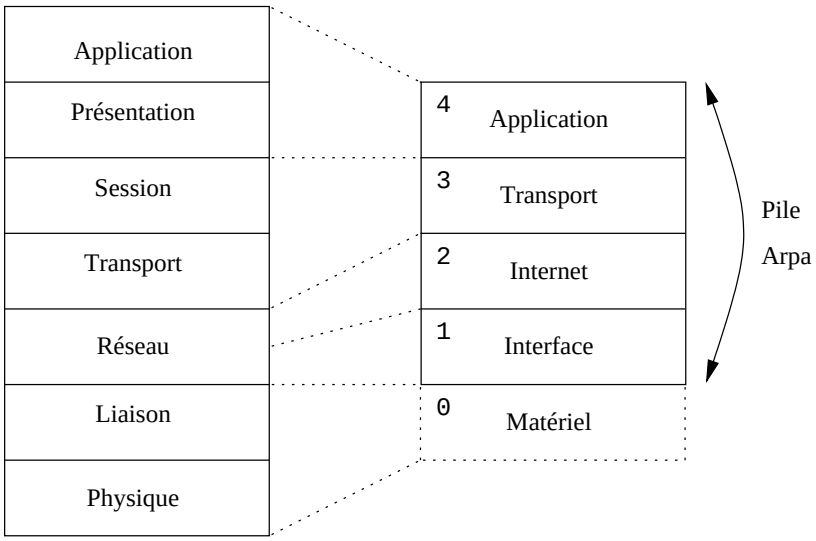


figure II.01 — Comparaison ISO-ARPA

La *figure II.01* met en comparaison les fonctionnalités des couches du modèle OSI et celles des protocoles TCP/IP.

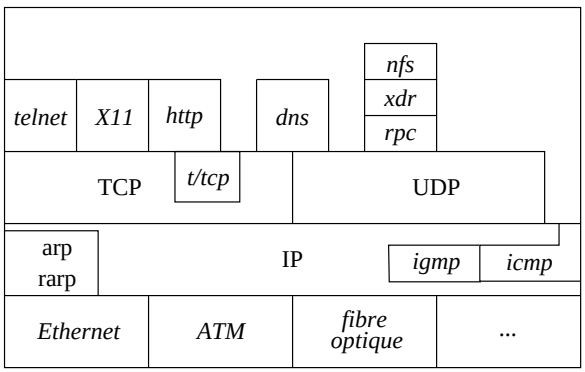


figure II.02 — Architecture logicielle

La *figure II.02* donne une vue d'ensemble de l'architecture logicielle avec protocoles d'applications de la famille IP. Ils sont très nombreux, non tous ici représentés et il s'en faut de beaucoup.

IP " Internet Protocol "

TCP " Transport Control Protocol "

UDP " User Datagram Protocol "

Les autres protocoles ne sont pas détaillés, nous aurons l'occasion d'en parler ultérieurement.

3.1 Couche " Application Layer "

Au plus haut niveau les utilisateurs invoquent les programmes qui permettent l'accès au réseau.

Chaque programme d'application interagit avec la couche de transport pour envoyer ou recevoir des données. En fonction des caractéristiques de l'échange le programme a choisi un mode de transmission à la couche de transport.

La plus grande proportion des applications laissent à la couche de transport le soin d'effectuer le travail de " Session ", néanmoins il est possible pour certaines applications de court-circuiter cette fonctionnalité pour agir directement au niveau " Réseau ", comme on peut l'observer sur la *figure II.02* à droite.

3.2 Couche " Transport Layer "

La principale tâche de la couche de transport est de fournir la communication d'un programme d'application à un autre. Une telle communication est souvent qualifiée de " point à point ".

Cette couche peut avoir à réguler le flot de données et à assurer la fiabilité du transfert : les octets reçus doivent être identiques aux octets envoyés. C'est pourquoi cette couche doit gérer des " checksums " et savoir re-émettre des paquets mal arrivés.

Cette couche divise le flux de données en paquets (terminologie de l'ISO) et passe chacun avec une adresse de destination au niveau inférieur.

De plus, et c'est surtout valable pour les systèmes d'exploitation multi-tâches multi-utilisateurs (Unix,...), de multiples processus appartenant à des utilisateurs différents et pour des programmes d'applications différents, accèdent au réseau au même moment, ce qui implique la capacité de multiplexer et de démultiplexer les données, suivant qu'elles vont vers le réseaux ou vers les programmes d'application (" Session ").

3.3 Couche " Internet Layer "

Cette couche reçoit des data-grammes en provenance de la couche réseau, qu'elle doit analyser pour déterminer s'ils lui sont adressés ou pas. Dans

le premier cas elle doit " décapsuler " son en-tête du data-gramme pour transmettre les données à la couche de transport et au bon protocole de cette couche (TCP, UDP...), dans le deuxième cas elle les ignore.

Cette couche prend aussi en charge la communication de machine à machine. Elle accepte des requêtes venant de la couche de transport avec une identification de la machine vers laquelle le paquet doit être envoyé.

Elle utilise alors l'algorithme de routage (page 67) pour décider si le paquet doit être envoyé vers une passerelle ou vers une machine directement accessible.

Enfin cette couche gère les datagrammes des protocoles `ICMP` et `IGMP`.

3.4 Couche " Network Access "

Le protocole dans cette couche définit le moyen pour un système de délivrer l'information à un autre système physiquement relié. Il définit comment les data-grammes IP sont transmis. La définition de ceux-ci reste indépendante de la couche réseau, ce qui leur permet de s'adapter à chaque nouvelle technologie au fur et à mesure de leur apparition.

Avant de s'intéresser au détail des data-grammes IP, nous allons examiner le problème de l'adressage IP, dans le chapitre suivant.

4 Encapsulation d'IP

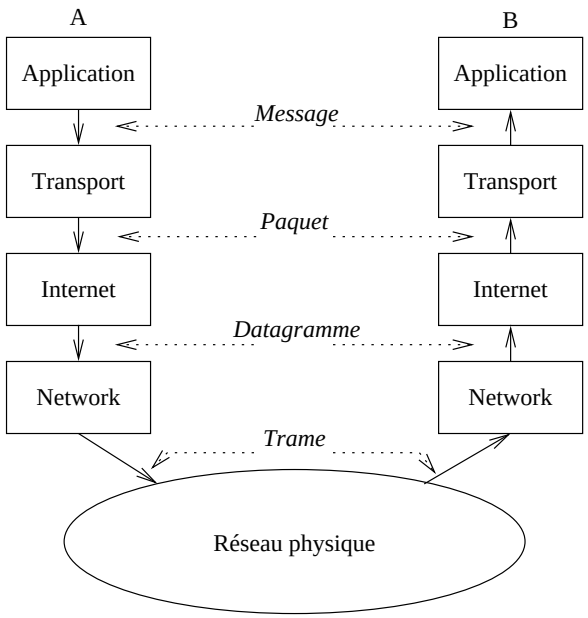


figure II.03 — Encapsulation d'IP

Comme nous l'avons décrit avec le modèle des couches OSI, les couches IP fonctionnent par encapsulations progressives.

Chaque couche en-capsule la précédente avec les informations de contrôle qu'elle destine à la couche de même niveau sur la machine distante.

Cet ajout est nommé " header " (en-tête) parce-qu'il est placé en tête des données à transmettre.

```
Application                                  | datas |
Transport                          | Header | datas |
Internet                  | Header | Header | datas |
Network      | Header | Header | Header | datas |
```

La taille des " headers " dépend des protocoles utilisés. Pour la couche IP le protocole comporte en standard 5 mots de 32 bits, même chose pour la couche TCP[6].

5 Bibliographie

Pour en savoir plus :

RFC 0791 S J. Postel, " Internet Protocol ", 09/01/1981. (Pages=45) (Format=.txt) (Obsoletes RFC0760)

La Recherche Numéro spécial Internet numéro 238 de février 2000

[6] 5 mots de 32 bits = 5×4 *octets* = 20 *octets*

Chapitre III

Anatomie d'une adresse IP

1 Adressage IP

Nous avons dit que l'Internet est un réseau virtuel, construit par interconnexion de réseaux physiques via des passerelles. Ce chapitre parle de l'adressage, le maillon essentiel des protocoles TCP/IP pour rendre transparents les détails physiques des réseaux et faire apparaitre l'Internet comme une entité uniforme.

1.1 Unicité de l'adresse

Un système de communication doit pouvoir permettre à n'importe quel hôte de se mettre en relation avec n'importe quel autre. Afin qu'il n'y ait pas d'ambiguïté pour la reconnaissance des hôtes possibles, il est absolument nécéssaire d'admettre un principe général d'identification.

Lorsque l'on veut établir une communication, il est intuitivement indispensable de posséder trois informations :

1. Le nom de la machine distante,
2. Son adresse,
3. La route à suivre pour y parvenir.

Le nom dit " qui " est l'hôte distant, l'adresse nous dit " où " il se trouve et la route " comment " on y parvient.

En règle générale les utilisateurs préfèrent des noms symboliques pour identifier les machines tandis que les processeurs des machines sont plus à l'aise avec les nombres.

Les adresses IP (version 4) sont standardisées sous forme d'un nombre de 32 bits qui permet à la fois l'identification de chaque hôte et du réseau auquel il appartient. Le choix des nombres composants une adresse IP n'est pas laissé au hasard, au contraire il fait l'objet d'une attention particulière notamment pour faciliter les opérations de routage.

Nous remettons à plus tard la correspondance entre ce nombre et une éventuelle représentation symbolique.

Chaque adresse IP contient donc deux informations basiques, une adresse de réseau et une adresse d'hôte. La combinaison des deux désigne de manière unique une machine et une seule sur l'Internet.

1.2 Délivrance des adresses IPv4

On distingue deux types d'adresses IP. Les adresses privées que tout administrateur de réseau peut s'attribuer librement pourvu qu'elle ne soient pas routées sur l'Internet, et les adresses publiques, délivrées par une structure mondiale qui en assure l'unicité. Ce dernier point est capital pour assurer l'efficience du routage, comme nous le verrons au chapitre suivant.

Les adresses à utiliser sur les réseaux privés sont décrites par la RFC 1918 [Y. Rekhter,...1996] :

`10.0.0.0`	`10.255.255.255` (ancien Arpanet !)
`172.16.0.0`	`172.31.255.255`
`192.168.0.0`	`192.168.255.255`

Les adresses publiques (souvent une seule), sont le plus généralement fournies par le FAI[1]. Qu'elles soient délivrées de manière temporaire ou attribuées pour le long terme, elles doivent être uniques sur le réseau. La question est donc de savoir de qui le FAI les obtient.

C'est L'ICANN (Internet Corporation for Assigned Names and Numbers)[2] qui est chargé au niveau mondial de la gestion de l'espace d'adressage IP. Il définit les procédures d'attribution et de résolution de conflits dans l'attribution des adresses, mais délègue le détail de la gestion de ces ressources à des instances régionales puis locales, dans chaque pays, appelées " Regional Internet Registries " ou RIR.

Il y a actuellement cinq " Regional Internet Registries " opérationnels : l'APNIC[3] pour la région Asie-Pacifique, l'ARIN[4] pour l'Amérique, le RIPE NCC[5] pour l'Europe, l'AfriNIC[6] pour l'Afrique enfin LACNIC[7] pour l'Amérique Latine.

Pour ce qui nous concerne en Europe c'est donc le RIPE NCC (Réseaux IP européen Network Coordination Centre) qui délivre les adresses que nous utilisons.

Les adresses IP sont allouées à l'utilisateur final qui en fait la demande par un " Local Internet Registry ", ou LIR, autorisé par le RIPE NCC. Un LIR est généralement un FAI ou une grande organisation (entreprise multinationale). Il est sous l'autorité de l'instance régionale de gestion de

[1] Fournisseur d'Accès Internet
[2] `http ://www.icann.org`
[3] `http ://www.apnic.net`
[4] `http ://www.arin.net/`
[5] `http ://www.ripe.net/`
[6] `http ://www.afrinic.net/`
[7] `http ://lacnic.net/`

l'adressage. Ainsi pour un utilisateur (quelle que soit sa taille) changer de FAI implique aussi de changer de plan d'adressage IP, lorsque celles-ci ont été allouées statiquement par le LIR. Les adresses IP sont alors restituées puis ré-attribuées à d'autres utilisateurs.

On compte plus de 2000 de LIRs offrant leurs services en Europe selon le RIPE NCC[8]

2 Anatomie d'une adresse IP

Une adresse IP est un nombre de 32 bits que l'on a coutume de représenter sous forme de quatre entiers de huit bits, séparés par des points (comme au paragraphe 1.2 ci-dessus).

La partie réseau de l'adresse IP vient toujours en tête, la partie hôte est donc toujours en queue.

L'intérêt de cette représentation est immédiat quand on sait que la partie réseau et donc la partie hôte sont presque toujours codées sur un nombre entier d'octets. Ainsi, on a principalement les trois formes suivantes :

Classe A Un octet réseau, trois octets d'hôtes.

Classe B Deux octets réseau, deux octets d'hôtes.

Classe C Trois octets réseau, un octet d'hôte.

2.1 Décomposition en classes

Classe	*Nombre de réseaux/machines*
A	*1.x.y.z à 127.x.y.z* *127 réseaux* *16 777 216 machines (2^24)*
B	*128.0.x.y à 191.255.x.y* *16 384 réseaux (2^14)* *65536 machines (2^16)*
C	*192.0.0.z à 223.255.255.z* *2 097 152 réseaux (2^21)* *256 machines (2^8)*
D	*224.0.0.0 à 239.255.255.255*
E	*240.0.0.0 à 247.255.255.255*

figure III.01 —
Décomposition en classes

Pour distinguer les classes A, B, C, D et E il faut examiner les bits de poids fort de l'octet de poids fort :

Si le premier bit est 0, l'adresse est de classe A. On dispose de 7 bits pour identifier le réseau et de 24 bits pour identifier l'hôte. On a donc les réseaux de 1 à 127 et 2^{24} hôtes possibles, c'est à dire 16 777 216 machines différentes (de 0 à 16 777 215).

Les lecteurs attentifs auront remarqué que le réseau 0 n'est pas utilisé, il a une signification particulière (“ tous les réseaux ”) comme on le précisera au paragraphe suivant.

De même, la machine 0 n'est pas utilisée, tout comme la machine ayant le plus fort numéro dans le réseau (tous les bits de la partie hôte à 1, ici

[8] `Voir http ://www.ripe.net/ripe/docs/ar2003.html`

16 777 215), ce qui réduit de deux unités le nombre des machines nommables. Il reste donc seulement 16 777 214 machines adressables dans une classe A !

Si les deux premiers bits sont 10 , l'adresse est de classe B. Il reste 14 bits pour identifier le réseau et 16 bits pour identifier la machine. Ce qui fait $2^{14} = 16\ 384$ réseaux (128.0 à 191.255) et 65 534 ($65\ 536 - 2$) machines.

Si les trois premiers bits sont 110 , l'adresse est de classe C. Il reste 21 bits pour identifier le réseau et 8 bits pour identifier la machine. Ce qui fait $2^{21} = 2\ 097\ 152$ réseaux (de 192.0.0 à 223.255.255) et 254 ($256 - 2$) machines.

Si les quatre premiers bits de l'adresse sont 1110 , il s'agit d'une classe d'adressage spéciale, la classe D. Cette classe est prévue pour faire du " multicast ", ou multipoint. (RFC 1112 [S. Deering, 1989]), contrairement aux trois premières classes qui sont dédiées à l'" unicast " ou point à point.

Ces adresses forment une catégorie à part, nous en reparlons au paragraphe 3.

Si les quatre premiers bits de l'adresse sont 1111 , il s'agit d'une classe expérimentale, la classe E. La RFC 1700 précise " Class E addresses are reserved for future use " mais n'indique pas de quel futur il s'agit...

Enfin, pour conclure ce paragraphe, calculons le nombre d'hôtes adressables théoriquement à l'aide des classes A, B et C :

$$127 \times 16777212 + 16384 \times 65534 + 2097152 \times 254 = 3737091588$$ [9]

Ce total, pour être plus exact, doit être amputé des 17 890 780 hôtes des réseaux privés prévus dans la RFC 1918[10], soit donc tout de même au total **3 719 200 808** hôtes adressables en utilisant IPv4 !

[9] Sous shell, tapez : `echo "127*16777212+16384*65534+2097152*254"|bc`

[10] $16777212 + 16 \times 65534 + 256 \times 254 = 17890780$

2.2 Adresses particulières

Certaines adresses IP ont une signification particulière !

Par convention le numéro 0 d'hôte n'est pas attribué. Si une adresse IP contient cette zone nulle cela signifie que l'on adresse le réseau lui-même et aucun hôte en particulier, donc en règle générale l'hôte lui-même. Par exemple, sur toutes les machines, l'adresse 127.0.0.1 indique la machine elle-même (" localhost " – Cf chapitre IP page 45), indépendamment de l'adresse réseau sur lequel elle est connectée.

À l'inverse, si tous les bits de la partie hôte sont à 1, cela désigne toutes les machines du réseaux, c'est ce que l'on appele une adresse de " broadcast ", c'est à dire une information adressée à tout le monde.

On évite au maximum l'usage d'une telle adresse IP sur les réseaux, pour des raisons d'efficacité (encombrement de la bande passante).
Quelques exemples d'adresses avec une signification particulière :

`0.0.0.0`	Hôte inconnu, sur ce réseau
`0.0.0.1`	L'hôte 1 de ce réseau
`255.255.255.255`	Tous les hôtes
`138.195.52.1`	L'hôte 52.1 du réseau 138.195.0.0
`138.195.0.0`	Cet hôte sur le 138.195.0.0
`193.104.1.255`	Tous les hôtes du 193.104.1.0
`127.0.0.1`	Cet hôte (boucle locale).

Remarque : les deux premières adresses, avec un numéro de réseau égal à 0, ne peuvent figurer que comme adresse source dans des cas bien particuliers comme le démarrage d'une station (cf chapitre IP page 45 et les travaux pratiques associés).

2.3 Sous-réseaux

En 1984 un troisième niveau de hiérarchie est mis en place : le " subnet " ou sous-réseau. pour permettre aux administrateurs de gérer plus finement de grands réseaux. La RFC 950 [J. Mogul, J. Postel, 1985] donne plus de précisions, la RFC 1878 [T. Pummill & B. Manning, 1995] est une table de tous les sous-réseaux possibles.

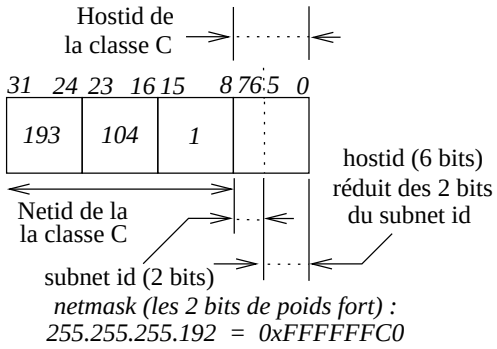


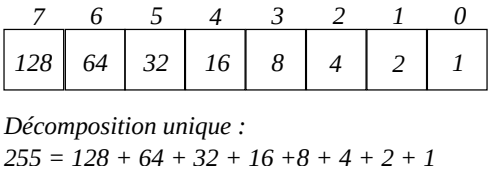
figure III.02 — Sous-réseaux

Le " subnet " utilise les bits de poids fort de la partie hôte de l'adresse IP, pour désigner un réseau. Le nombre de bits employés est laissé à l'initiative de l'administrateur.

Dans la *figure III.02* ci-contre, les bits 6 et 7 de la partie " host " sont utilisés pour caractériser un sous-réseau.

Quelques révisions des propriétés des puissances de 2[11] sont souvent nécessaires pour bien assimiler ce paragraphe. La figure suivante en rappelle les valeurs pour les huit premiers exposants :

figure III.03 — Puissances de 2



Nous avons d'une part $2^7 + 2^6 = 192$, et d'autre part $2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 63$[12]. Ce qui permet de caractériser 4 sous-réseaux de 62 machines (63 moins l'adresse de broascast, le " 0 " n'étant pas compté). Le calcul des masques et des adresses de diffusion est expliqué dans le tableau suivant :

Numéro du réseau	" Netmask "	" Broadcast "	Adressage hôte
`193.104.1.00`	`255.255.255.192`	$00 + 63 = 63$	.1 à .62
`193.104.1.64`	`255.255.255.192`	$64 + 63 = 127$	.65 à .126
`193.104.1.128`	`255.255.255.192`	$128 + 63 = 191$	.129 à .190
`193.104.1.192`	`255.255.255.192`	$192 + 63 = 255$	.193 à .254

[11] et plus généralement de la décomposition d'un nombre en ses facteurs premiers...
[12] Donc 64 valeurs possibles de 0 à 63

Soit un total de $62 \times 4 = 248$ hôtes possibles pour cette classe C avec un masque de sous-réseau[13], au lieu des 254 hôtes sans.

La machine d'adresse 1 sur chaque sous-réseau, aura comme adresse IP :

Sous-réseau	Adresse	Décomposition
00	`193.104.1.1`	$00 + 1 = 1$
01	`193.104.1.65`	$64 + 1 = 65$
10	`193.104.1.129`	$128 + 1 = 129$
11	`193.104.1.193`	$192 + 1 = 193$

Si vous pensez avoir tout compris, le remplissage du tableau suivant dans le cas de la classe C `192.168.192.0` et avec 3 bits pour définir les sous-réseaux ne devrait pas vous poser de problème...

Numéro du subnet	Numéro du réseau	Adresse de broadcast	Première machine	Dernière machine
000(0)	`192.168.192.`	`192.168.192.`		
001(1)	`192.168.192.`	`192.168.192.`		
010(2)	`192.168.192.`	`192.168.192.`		
011(3)	`192.168.192.`	`192.168.192.`		
100(4)	`192.168.192.`	`192.168.192.`		
101(5)	`192.168.192.`	`192.168.192.`		
110(6)	`192.168.192.`	`192.168.192.`		
111(7)	`192.168.192.`	`192.168.192.`		

À toutes ces adresses il faudra appliquer le masque de sous-réseau :

0xFFFFFF__ soit encore 255.255.255.___

Remarque : On pourra vérifer que la perte d'espace d'adressage pour des hôtes se calcule avec la relation $(2^n - 1) \times 2$, où n est le nombre de bits du masque. Ainsi avec 3 bits de masque de sous-réseau, la perte d'espace d'adressage s'élève à 14 hôtes ! Les 254 possibilités (256 moins 0 et 255) de nommage de la classe C se réduisent à 240, amputées de 31, 32, 63, 64, 95, 96, 127, 128, 159, 160, 191, 192, 223 et 224.

[13] « netmask »

2.4 CIDR

En 1992 la moitié des classes B étaient allouées, et si le rythme avait continué, au début de 1994 il n'y aurait plus eu de classe B disponible et l'Internet aurait bien pu mourrir par asphyxie ! De plus la croissance du nombre de réseaux se traduisait par un usage " aux limites " des routeurs, proches de la saturation car non prévus au départ pour un tel volume de routes (voir les RFC 1518 et RFC 1519).

Deux considérations qui ont conduit l'IETF a mettre en place le CIDR (" Classless InterDomain Routing " ou routage Internet sans classe) basé sur une constatation de simple bon sens :

- S'il est courant de rencontrer une organisation ayant plus de 254 hôtes, il est moins courant d'en rencontrer une de plus de quelques milliers.
  Les adresses allouées sont donc des classes C contigües, attribuées par région ou par continent. En générale, 8 à 16 classes C mises bout à bout suffisent pour une entreprise. Ces blocs de numéros sont souvent appellés " supernet ".
  Ainsi par exemple il est courant d'entendre les administrateurs de réseaux parler d'un " slash 22 " (/22) pour désigner un bloc de quatre classes C consécutives...
- Il est plus facile de prévoir une table de routage pour un bloc d'adresses contigües qu'adresse par adresse, en plus cela allège les tables.

Plus précisement, trois caractéristiques sont requises pour pouvoir utiliser ce concept :

1. Pour être réunies dans une même route, des adresses IP multiples doivent avoir les mêmes bits de poids fort.
2. Les tables de routages et algorithmes doivent prendre en compte un masque de 32 bits, à appliquer sur les adresses.
3. Les protocoles de routage doivent ajouter un masque 32 bits pour chaque adresse IP (Cet ajout double le volume d'informations) transmise. OSPF, RIP-2, BGP-4 le font.

   Ce masque se manifeste concrêtement comme dans la reécriture du tableau du paragraphe 1.2 :

   ```
   10.0.0.0      10.255.255.255   10/8
   172.16.0.0    172.31.255.255   172.16/12
   192.168.0.0   192.168.255.255  192.168/16
   ```

   Le terme " classless " vient de ce fait, le routage n'est plus basé uniquement sur la partie réseau des adresses.

Les agrégations d'adresses sont ventilées selon le tableau suivant[14] :

Multirégionales	192.0.0.0	193.255.255.255
Europe	194.0.0.0	195.255.255.255
Autres	196.0.0.0	197.255.255.255
Amérique du Nord	198.0.0.0	199.255.255.255
Amérique centrale, Amérique du Sud	200.0.0.0	201.255.255.255
Zone Pacifique	202.0.0.0	203.255.255.255
Autres	204.0.0.0	205.255.255.255
Autres	206.0.0.0	207.255.255.255

2.5 Précisions sur le broadcast

Tout d'abord il faut préciser qu'une adresse de broadcast est forcément une adresse de destination, elle ne peut jamais apparaître comme une adresse source dans un usage normal des réseaux.

Il y a quatre formes possibles de broadcast :

" **Limited broadcast** " (255.255.255.255) Une telle adresse ne peut servir que sur le brin local et ne devrait jamais franchir un routeur. Ce n'est malheureusement pas le cas (précisions en cours).

L'usage de cette adresse est normalement limitée à un hôte en phase d'initialisation, quand il ne connait rien du réseau sur lequel il est connecté.

" **Net-directed broadcast** " Tous les bits de la partie hôte sont à 1. Un routeur propage ce type de broadcast, sur option.

" **Subnet-directed broadcast** " C'est le même cas que ci-dessus mais avec une adresses IP comportant des subnets.

" **All-subnets-directed broadcast** " C'est le cas où tous les bits des subnets et hôtes sont à 1. Ce cas possible théoriquement est rendu obsolète depuis la RFC 922 (1993).

[14] Ce tableau est très synthétique, pour une information plus détaillée et à jour consultez le site de l'IANA http ://www.iana.org/assignments/ipv4-address-space

3 Adressage multicast

En règle générale l'adressage multicast est employé pour s'adresser en une seule fois à un groupe de machines.

Dans le cas d'un serveur vidéo/audio, cette approche induit une économie de moyen et de bande passante évidente quand on la compare à une démarche " unicast " : un seul datagramme est routé vers tous les clients intéressés au lieu d'un envoi massif d'autant de datagrammes qu'il y a de clients.

Les adresses de type " multicast " ont donc la faculté d'identifier un groupe de machines qui partagent un protocole commun par opposition à un groupe de machines qui partagent un réseau commun.

La plupart des adresses multicast allouées le sont pour des applications particulières comme par exemple la découverte de routeurs (que nous verrons ultérieurement lors du routage IP) ou encore la radio ou le téléphone/vidéo sur Internet (" Mbone "). Les plus couramment utilisées[15] sur un lan sont :

`224.0.0.1`	Toutes les machines sur ce sous-réseau
`224.0.0.2`	Tous les routeurs sur ce sous-réseau

3.1 Adresse de groupe multicast

Si une adresse multicast démarre avec les bits 1110 par contre pour les 28 bits suivants son organisation interne diffère de celle des classes A, B et C.

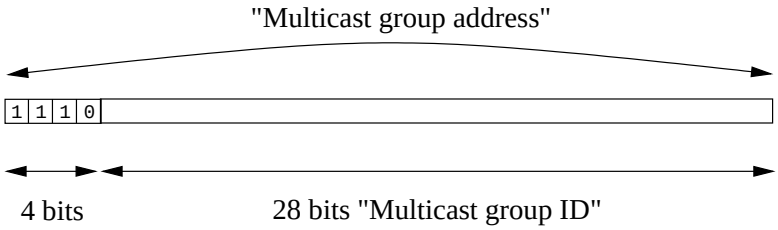


figure III.04 — Adresses de multicast

- Les 28 bits n'ont pas de structure particulière par contre on continue à utiliser la notation décimale pointée : `224.0.0.0` à `239.255.255.255`.
- Un groupe d'hôtes qui partagent un protocole commun utilisant une adresse multicast commune peuvent être répartis n'importe où sur le réseau.
- L'appartenance à un groupe est dynamique, les hôtes qui le désirent rejoignent et quittent le groupe comme ils veulent.
- Il n'y a pas de restriction sur le nombre d'hôtes dans un groupe et un hôte n'a pas besoin d'appartenir à un groupe pour lui envoyer un message.

[15] Pour plus de précisions on pourra se reporter page 56 de la RFC 1700

3.2 Adresse multicast et adresse MAC

Une station Ethernet quelconque doit être configurée pour accepter le multicast, c'est à dire pour accepter les trames contenant un datagramme munis d'une adresse IP de destination qui est une adresse multicast.

Cette opération sous entend que la carte réseau sait faire le tri entre les trames. En effet les trames multicast ont une adresse MAC particulière : elles commencent forcément par les trois octets `01:00:5E`[16]. Ceux-ci ne désignent pas un constructeur en particulier mais sont possédés par l'ICANN (ex IANA).

Restent trois octets dont le bit de poids fort est forcément à 0 pour désigner les adresses de multicast (contrainte de la RFC 1700), ce qui conduit au schéma suivant :

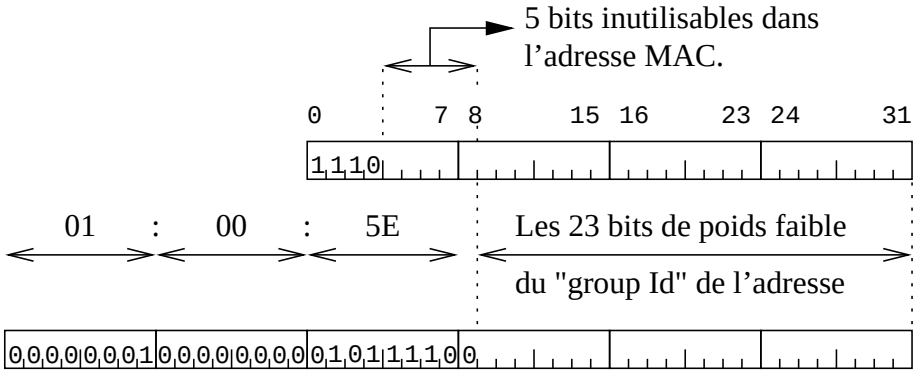


figure III.05 — Adresse physique de multicast

Du fait qu'il n'y a pas assez de place dans l'adresse MAC pour faire tenir les 28 bits du groupe multicast, cette adresse n'est pas unique. On peut même préciser que pour chaque trame comportant une adresse multicast il y a 2^5 adresses IP de groupes multicast possibles !

Ce qui signifie que si les 23 bits de poids faible ne suffisent pas à discriminer la trame il faudra faire appel au pilote de périphérique ou à la couche IP pour lever l'ambiguïté.

Quand une trame de type multicast est lue par la station Ethernet puis par le pilote de périphérique, si l'adresse correspond à l'une des adresses de groupe multicast préalablement configurées, le datagramme franchit la couche IP et une copie des données est délivrée aux processus qui ont " joint le groupe multicast ".

La question est de savoir comment les trames de type multicast atteignent justement cette station Ethernet ? La réponse se trouve dans un protocole nommé `IGMP` et que nous examinerons dans le prochain chapitre concernant `IP`, page 60.

[16] Cf RFC 1700 page 171

4 Conclusion et bibliographie

Pour conclure ce chapitre sur l'adressage IP, il faut nous donner quelques précisions supplémentaires.

Jusqu'à présent nous avons désigné un hôte par son adresse IP. Cette démarche n'est pas exacte si on considère par exemple le cas d'une passerelle, connectée physiquement à au moins deux réseaux différents, avec une adresse IP dans chacun de ces réseaux.

On dira donc maintenant qu'une adresse IP identifie non pas un hôte mais un interface. La réciproque n'est pas vraie car un même interface peut collectionner plusieurs adresses IP. Toutes permettent d'atteindre cet interface, on parle alors d'" alias IP ", d'" hôtes virtuels " et de " réseaux virtuels "...Nous aurons l'occasion de revenir sur ces notions à la fin de ce cours (page 177).

On dit d'une machine ayant au moins deux adresses IP qu'elle est du type " multi-homed ".

En général une passerelle qui met en relation n réseaux possède n adresses IP différentes (une dans chaque réseau), mais ce n'est pas une obligation (nous verrons quelle peut en être l'utilité à la fin de ce cours).

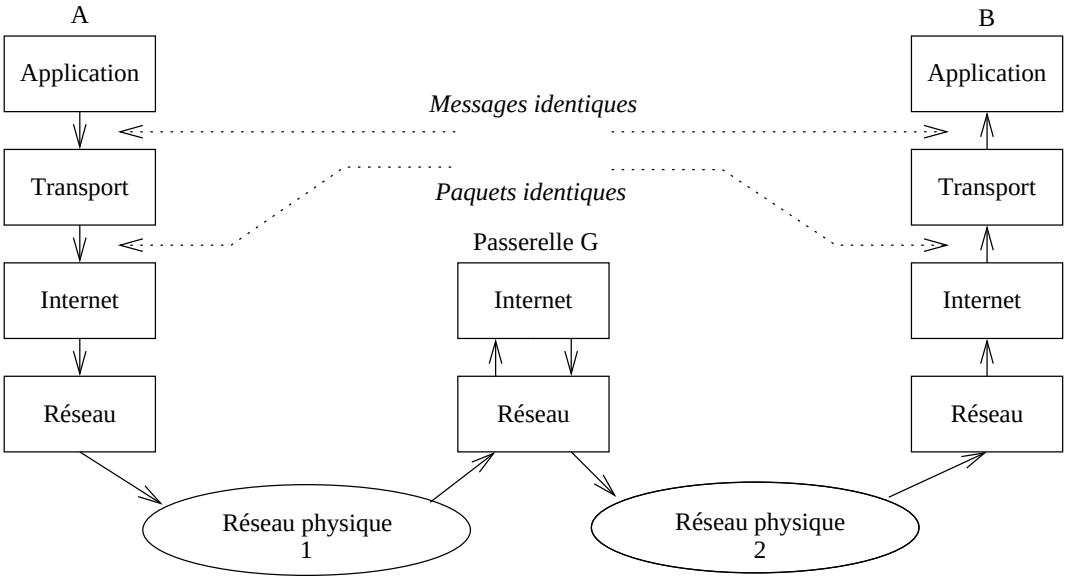


figure III.06 — Usage combiné des adresses logique et physique

La *figure III.06* met en situation deux hôtes, A et B, en relation via une passerelle G. Si les " messages " et les " paquets " sont identiques, par contre les " datagrammes " et les " trames " diffèrent puisqu'il ne s'agit plus du même réseau physique. Dès que nous aurons examiné le fonctionnement de la couche IP nous reviendrons sur cette figure pour en expliquer le fonctionnement (voir page 73).

Pour en savoir plus :

RFC 0950 S J. Mogul, J. Postel, “ Internet standard subnetting procedure ”,
08/01/1985. (Pages=18) (Format=.txt) (STD 5)

RFC 1112 S S. Deering, “ Host extensions for IP multicasting ”, 08/01/1989.
(Pages=17) (Format=.txt) (Obsoletes RFC0988) (STD 5)

RFC 1518 “ An Architecture for IP Address Allocation with CIDR ”
Y. Rekhter, T. Li. September 1993. (Format : TXT=72609 bytes) (Status : PROPOSED STANDARD)

RFC 1519 PS V. Fuller, T. Li, J. Yu, K. Varadhan, “ Classless Inter-Domain
Routing (CIDR) : an Address Assignment and Aggregation Strategy ”, 09/24/1993. (Pages=24) (Format=.txt) (Obsoletes RFC1338)

RFC 1466 I E. Gerich, “ Guidelines for Management of IP Address Space ”, 05/26/1993. (Pages=10) (Format=.txt) (Obsoletes RFC1366)

RFC 1467 “ Status of CIDR Deployment in the Internet. ”
C. Topolcic. August 1993. (Format : TXT=20720 bytes)
(Obsoletes RFC1367) (Status : INFORMATIONAL)

RFC 1700 “ Assigned Numbers. ” J. Reynolds, J. Postel. October 1994.
(Format : TXT=458860 bytes) (Obsoletes RFC1340)
(Also STD0002) (Status : STANDARD)

RFC 1878 “ Variable Length Subnet Table For IPv4. ”
T. Pummill & B. Manning. December 1995.
(Format : TXT=19414 bytes) (Obsoletes RFC1860) (Status : INFORMATIONAL)

RFC 1918 “ Address Allocation for Private Internets. ” Y. Rekhter, B. Moskowitz,
D. Karrenberg, G. J. de Groot & E. Lear. February 1996.
(Format : TXT=22270 bytes) (Obsoletes RFC1627, RFC1597) (Also BCP0005)
(Status : BEST CURRENT PRACTICE)

Quelques ouvrages qui font autorité :

– W. Richard Stevens - TCP/IP Illustrated, Volume 1 - The protocols - Addison-Wesley
– Douglas Comer - Internetworking with TCP/IP - Principles, protocols, and architecture - Prentice–Hall
– Christian Huitema - Le routage dans l’Internet - EYROLLES

Chapitre IV

Protocole IP

1 Datagramme IP

IP est l'acronyme de " **I**nternet **P**rotocol ", il est défini dans la RFC 791 et a été conçu en 1980 pour remplacer NCP (" Network Control Protocol "), le protocole de l'Arpanet.

Plus de vingt ans après sa première implémentation, ses limitations se font de plus en plus pénalisantes pour les nouveaux usages sur les réseaux. Avant de le jeter aux orties, posons-nous la question de qui pouvait prévoir à cette époque où moins de mille ordinateurs étaient reliés ensembles, que deux décennies plus tard des dizaines de millions d'hôtes l'utiliseraient comme principal protocole de communication ?

Sa longévité est donc remarquable et il convient de l'analyser de près avant de pouvoir le critiquer de manière constructive.

1.1 Structure de l'en-tête

Les octets issus de la couche de transport et encapsulés à l'aide d'un en-tête IP avant d'être propagés vers la couche réseau (Ethernet par exemple), sont collectivement nommés " datagramme IP ", datagramme Internet ou datagramme tout court. Ces datagrammes ont une taille maximale liée aux caractéristiques de propagation du support physique, c'est le MTU (" Maximum Transfert Unit ").

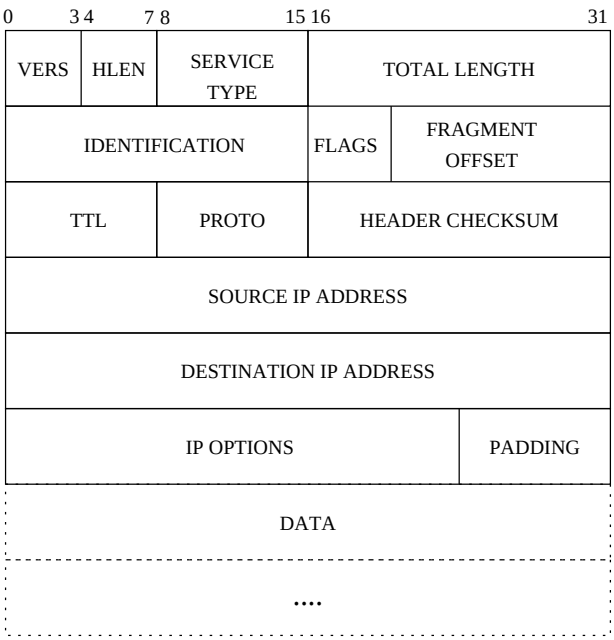


figure IV.01 — Structure du datagramme IP

Quelques caractéristiques en vrac du protocole IP :

- IP est le support de travail des protocoles de la couche de transport, TCP, UDP.
- IP ne donne aucune garantie quant au bon acheminement des données qu'il envoie. Il n'entretient aucun dialogue avec une autre couche IP distante, on dit aussi qu'il délivre les datagramme " au mieux ".
- Chaque datagramme est géré indépendamment des autres datagrammes même au sein du transfert des octets d'un même fichier. Cela signifie que les datagrammes peuvent être mélangés, dupliqués, perdus ou altérés !
  Ces problèmes ne sont pas détectés par IP et donc il ne peut en informer la couche de transport.
- Sur la *figure IV.01* les bits les plus significatifs de chaque mot de quatre octets sont à gauche (31...). Ils sont d'ailleurs transmis sur le réseau dans cet ordre, c'est un standard, c'est le " Network Byte Order ".
  Toutes les architectures de CPU ne sont pas bâties sur le même modèle :

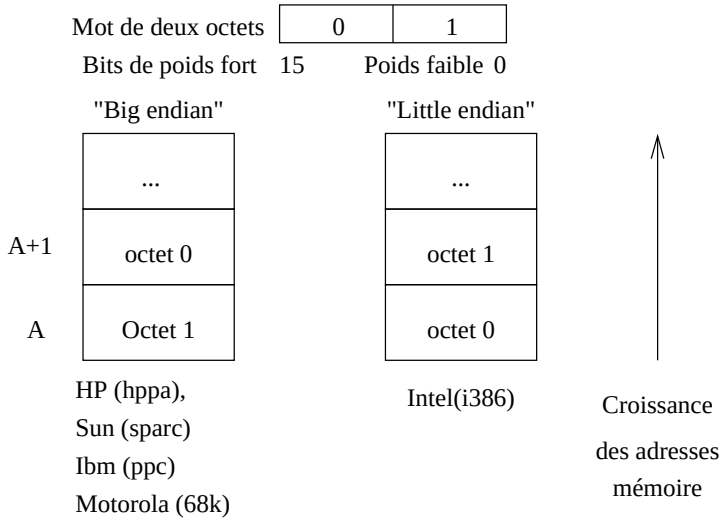


figure IV.02 — " Big endian " vs " Little endian "

- L'en-tête IP minimale fait 5 mots de 4 octets, soit 20 octets. S'il y a des options la taille maximale peut atteindre 60 octets.

1.2 Description de l'en-tête

`VERS` 4 bits qui spécifient la version du protocol IP. L'objet de ce champ est la vérification que l'émetteur et le destinataire des datagrammes sont bien en phases avec la même version. Actuellement c'est la version 4 qui est principalement utilisé sur l'Internet, bien que quelques implémentations de la version 6 existent et soient déjà en expérimentation[1].

[1] Nous examinerons les caractéristiques de la version 6 d'IP à la fin de ce cycle de cours

`HLEN` 4bits qui donnent la longueur de l'en-tête en mots de 4 octets. La taille standard de cette en-tête fait 5 mots, la taille maximale fait : $(2^3 + 2^2 + 2^1 + 2^0) \times 4 = 60$ octets[2]

`TOTAL LENGTH` Donne la taille du datagramme, en-tête plus données. S'il y fragmentation (voir plus loin) il s'agit également de la taille du fragment (chaque datagramme est indépendant des autres).

La taille des données est donc à calculer par soustraction de la taille de l'en-tête.

16 bits autorisent la valeur 65535...La limitation vient le plus souvent du support physique (MTU) qui impose une taille plus petite, sauf sur les " hyperchannel ".

`TYPE OF SERVICE` **vs** `DSCP/ECN` Historiquement dans la RFC 791 ce champ est nommé `TYPE OF SERVICE` et joue potentiellement deux rôles selon les bits examinés (préséance et type de service). Pratiquement, la préséance ne sert plus et la RFC 1349 définit 4 bits utiles sur les huit (3 à 6). Ceux-ci indiquent au routeur l'attitude à avoir vis à vis du datagramme.

Par exemple, des datagrammes d'un transfert de fichier (`ftp`) peuvent avoir à laisser passer un datagramme repéré comme contenant des caractères frappés au clavier (session telnet).

`0x00`	-	Service normal	Transfert banal
`0x10`	bit 3,D	Minimiser le délai	Session telnet
`0x08`	bit 4,T	Maximiser le débit	Transfert ftp
`0x04`	bit 5,R	Maximiser la qualité	ICMP
`0x02`	bit 6,C	Minimiser le coût	" news " (nntp)

L'usage de ces bits est mutuellement exclusif.

Les nouveaux besoins de routage on conduit l'IETF a revoir la définition de ce champ dans la RFC 3168. Celle ci partage les huit bits en deux parties, les premiers bits définissent le DSCP ou " Differentiated Services CodePoints " qui est une version beaucoup plus fine des quatre bits ci-dessus. Les deux derniers bits définissent l'ECN ou " Explicit Congestion Notification " qui est un mécanisme permettant de prévenir les congestions, contrairement au mécanisme plus ancien basé sur les messages ICMP de type " source quench " (voir page 58) qui tente de régler le flux en cas de congestion.

Il faut noter que les protocoles de routage qui tiennent compte de l'état des liaisons (OSPF,IS-IS...) sont susceptibles d'utiliser ce champ.

Enfin la RFC 3168 indique que ces deux écritures du champ ne sont pas compatibles entre elles...

`IDENTIFICATION, FLAGS et FRAGMENT OFFSET` Ces mots sont prévus pour contrôler la fragmentation des datagrammes. Les données sont frag-

[2] On encore plus simple $(2^4 - 1) \times 4$

mentées car les datagrammes peuvent avoir à traverser des réseaux avec des MTU plus petits que celui du premier support physique employé.

Consulter la section suivante *Fragmentation IP*.

`TTL` " Time To Live " 8 bits, 255 secondes maximum de temps de vie pour un datagramme sur le net.

Prévu à l'origine pour décompter un temps, ce champ n'est qu'un compteur décrémenté d'une unité à chaque passage dans un routeur.

Couramment la valeur de départ est 32 ou même 64. Son objet est d'éviter la présence de paquets fantômes circulant indéfiniment...

Si un routeur passe le compteur à zéro avant délivrance du datagramme, un message d'erreur — ICMP (consultez le paragraphe 4) — est renvoyé à l'émetteur avec l'indication du routeur. Le paquet en lui-même est perdu.

`PROTOCOL` 8 bits pour identifier le format et le contenu des données, un peu comme le champ " type " d'une trame Ethernet. Il permet à IP d'adresser les données extraites à l'une ou l'autre des couches de transport.

Dans le cadre de ce cours, nous utiliserons essentiellement ICMP, IGMP, UDP et TCP.

`HEADER CHECKSUM` 16 bits pour s'assurer de l'intégrité de l'en-tête. Lors du calcul de ce " checksum " ce champ est à 0.

A la réception de chaque paquet, la couche calcule cette valeur, si elle ne correspond pas à celle trouvée dans l'en-tête le datagramme est oublié (" discard ") sans message d'erreur.

`SOURCE ADDRESS` Adresse IP de l'émetteur, à l'origine du datagramme.

`DESTINATION ADDRESS` Adresse IP du destinataire du datagramme.

`IP OPTIONS` 24 bits pour préciser des options de comportement des couches IP traversées et destinatrices. Les options les plus courantes concernent :

- Des problèmes de sécurité
- Des enregistrements de routes
- Des enregistrements d'heure
- Des spécifications de route à suivre
- ...

Historiquement ces options ont été prévues dès le début mais leur implémentation n'a pas été terminée et la plupart des routeurs filtrants bloquent les datagrammes IP comportant des options.

`PADDING` Remplissage pour aligner sur 32 bits...

En conclusion partielle que peut-on dire du travail de la couche IP ?

1. Il consiste à router les datagrammes en les acheminant " au mieux ", on verra plus loin de quelle manière. C'est son travail principal.
2. Il peut avoir à fragmenter les données de taille supérieure au MTU du support physique à employer.

1.3 Fragmentation IP

La couche de liaison (Couche 2 “ Link ”) impose une taille limite, le “ Maximum Transfert Unit ”. Par exemple cette valeur est de 1500 pour une trame Ethernet, elle peut être de 256 avec SLIP (“ Serial Line IP ”) sur liaison série (RS232...).

Dans ces conditions, si la couche IP doit transmettre un bloc de données de taille supérieure au MTU à employer, il y a fragmentation !

Par exemple, un bloc de 1481 octets sur Ethernet sera décomposé en un datagramme de 1480 ($1480 + 20 = 1500$) et un datagramme de 1 octet !

Il existe une exception à cette opération, due à la présence active du bit “ Don't Fragment bit ” du champ `FLAGS` de l'en-tête IP. La présence à 1 de ce bit interdit la fragmentation dudit datagramme par la couche IP qui en aurait besoin. C'est une situation de blocage, la couche émettrice est tenue au courant par un message ICMP (consultez le paragraphe 4) *Fragmentation needed but don't fragment bit set* et bien sûr le datagramme n'est pas transmis plus loin.

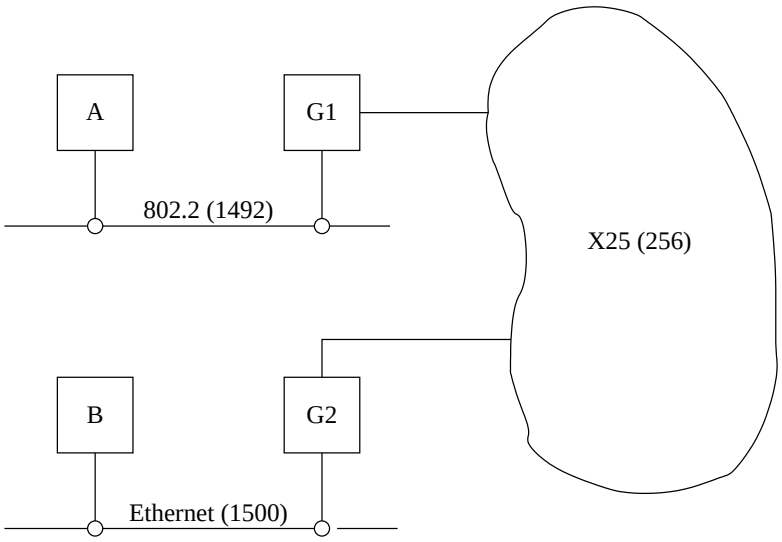


figure IV.03 — Fragmentation IP

Fragmentation

- Quand un datagramme est fragmenté, il n'est réassemblé que par la couche IP destinatrice finale. Cela implique trois remarques :
  1. La taille des datagrammes reçus par le destinataire final est directement dépendante du plus petit MTU rencontré.
  2. Les fragments deviennent des datagrammes à part entière.
  3. Rien ne s'oppose à ce qu'un fragment soit à nouveau fragmenté.
- Cette opération est absolument transparente pour les couches de transport qui utilisent IP.

– Quand un datagramme est fragmenté, chaque fragment comporte la même valeur de champ `IDENTIFICATION` que le datagramme initial.
S'il y a encore des fragments, un des bits du champ `FLAGS` est positionné à 1 pour indiquer " More fragment " !
Ce champ a une longueur de 3 bits.
`FRAGMENT OFFSET` contient l'offset du fragment, relativement au datagramme initial.
Cet offset est codé sur 13 bits.

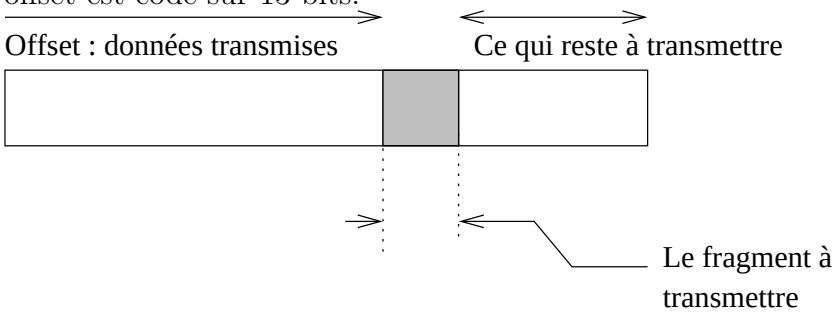


figure IV.04 — Fragment à transmettre

Pour tous les fragments :
– Les données doivent faire un multiple de 8 octets, sauf pour le dernier fragment, évidement.
– Le champ `TOTAL LENGTH` change.
– Chaque fragment est un datagramme indépendant, susceptible d'être à son tour fragmenté.

Pour le dernier fragment :
– `FLAGS` est remis à zéro.
– Les données ont une taille quelconque.

Réassemblage

– Tous les datagrammes issus d'une fragmentation deviennent des datagrammes IP comme (presque) les autres.
– Ils arrivent à destination, peut être dans le désordre, dupliqués. IP doit faire le tri.
– il y a suffisamment d'information dans l'en-tête pour réassembler les fragments épars.
– **Mais** si un fragment manque, la totalité du datagramme est perdu car aucun mécanisme de contrôle n'est implémenté pour cela dans IP.

La *figure IV.05* résume l'opération de fragmentation d'un datagramme IP.

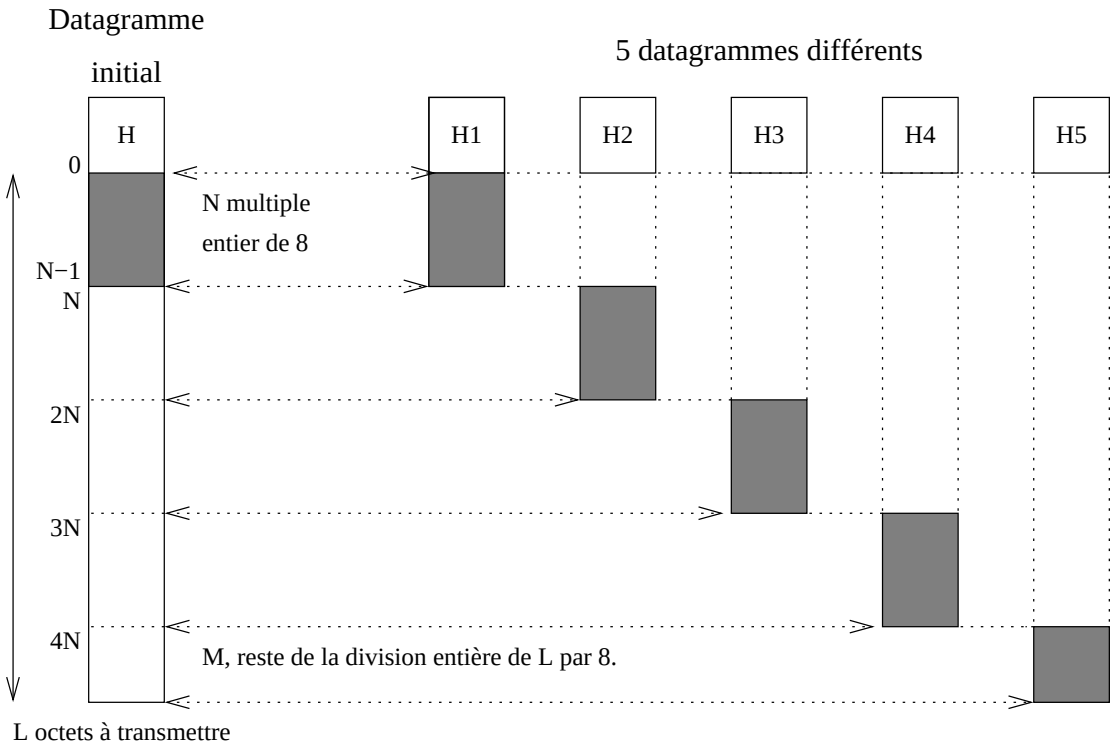


figure IV.05 — Résumé de la fragmentation

	H1	H2	H3	H4	H5
`IDENTIFICATION`	I	I	I	I	I
`FLAG`	`MF`	`MF`	`MF`	`MF`	0
`OFFSET`	0	N	$2 \times N$	$3 \times N$	$4 \times N$
`TOTAL LENGTH`	$H + N$	$H + N$	$H + N$	$H + N$	$H + M$
`HEADER CHECKSUM`	C_1	C_2	C_3	C_4	C_5

Notez les variations de certains champs de l'en-tête :

1. `IDENTIFICATION` est le même pour tous
2. `FLAG` est 0 pour le dernier datagramme
3. `OFFSET` croît de la taille du fragment, ici N.
4. `TOTAL LENGTH` est généralement différent pour le dernier fragment, sauf cas particulier.
5. `HEADER CHECKSUM` change à chaque fois car l'`OFFSET` change (rappel : il ne tient pas compte des données).

2 Protocole ARP

`ARP` est l'acronyme de " Address Resolution Protocol ", il est définie dans la RFC 826.

- Le problème à résoudre est issu de la constatation qu'une adresse IP n'a de sens que pour la suite de protocole TCP/IP ; celle-ci étant indépendante de la partie matérielle il faut avoir un moyen d'établir un lien entre ces deux constituants.
- La norme Ethernet (vs IEEE) suppose l'identification unique de chaque carte construite et vendue[3].
- Sur une même liaison physique (lire plus loin " même LAN "), Ethernet par exemple, deux machines peuvent communiquer $\Longleftrightarrow$ elles connaissent leurs adresses physiques respectives.
  On suppose qu'une machine connait sa propre adresse physique par un moyen qui n'est pas décrit ici (ne fait pas partie du protocole).
  Remarque importante : Cette information n'a pas de sens dans le cadre d'une liaison de type " point à point " avec un protocole tel que `ppp`.
- Lors du premier échange entre 2 machines d'un même LAN, si les adresses physiques ne sont pas déjà connues (on verra pourquoi plus loin), la solution à ce problème passe par l'usage du protocole `ARP`.
- L'usage de `ARP` est complètement transparent pour l'utilisateur.

2.1 Fonctionnement

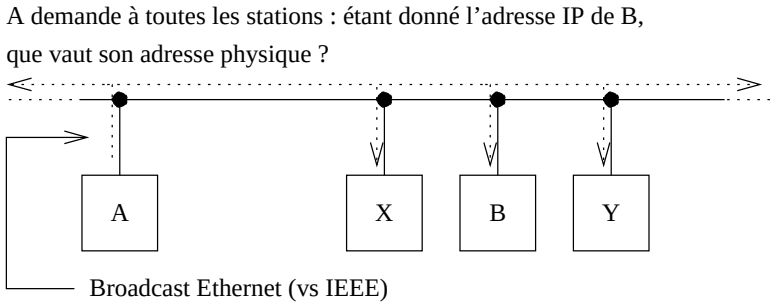


figure IV.06 — Question ARP

Sur la *figure IV.06* la station Ethernet A (I_A, P_A) a besoin de connaitre l'adresse physique de la station Ethernet B (I_B, P_B), pour ce faire elle envoie un datagramme de format spécial (cf paragraphe suivant), dédié à `ARP`, qui lui permet de poser la question (" Arp question ") à l'ensemble des machines actives. L'adresse de la machine qui doit répondre étant l'objet de la question, son adresse (champ destinataire) est donc remplacée par une adresse de " broadcast " (48 bits à 1).

Toutes les machines du LAN écoutent cet échange et peuvent mettre à jour leur table de conversion (adresse IP adresse Ethernet) pour la machine

[3] cf chapitre I " Réseaux locaux "

A.

Le “ broadcast ”, coûteux en bande passante, est ainsi utilisé au maximum de ses possibilités. Sur la *figure IV.07* la réponse de B est du type “ unicast ”.

Remarque : quand une station Ethernet ne répond plus (cf `ICMP`) il y a suppression de l'association adresse IP - adresse MAC.

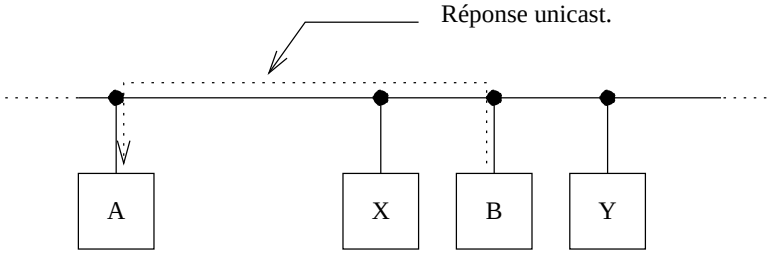


figure IV.07 — Réponse ARP

Si la station B ne répond pas, la station continuera à poser la question à intervals réguliers pendant un temps infini...

Il n'est pas besoin d'utiliser `ARP` préalablement à chaque échange, car heureusement le résultat est mémorisé.

En règle générale la durée de vie d'une adresse en mémoire est de l'ordre de 20 minutes et chaque utilisation remet à jour ce compteur.

La commande `arp -a` sous Unix permet d'avoir le contenu de la table de la machine sur laquelle on se trouve, par exemple :

```
$ arp -a
soupirs.chezmoi.fr  (192.168.192.10) at 8:0:9:85:76:9c
espoirs.chezmoi.fr  (192.168.192.11) at 8:0:9:85:76:bd
plethore.chezmoi.fr (192.168.192.12) at 8:0:9:a:f9:aa
byzance.chezmoi.fr  (192.168.192.13) at 8:0:9:a:f9:bc
ramidus.chezmoi.fr  (192.168.192.14) at 0:4f:49:1:28:22 permanent
desiree.chezmoi.fr  (192.168.192.33) at 8:0:9:70:44:52
pythie.chezmoi.fr   (192.168.192.34) at 0:20:af:2f:8f:f1
ramidus.chezmoi.fr  (192.168.192.35) at 0:4f:49:1:36:50 permanent
gateway.chezmoi.fr  (192.168.192.36) at 0:60:8c:81:d5:1b
```

Enfin, et c'est un point très important, du fait de l'utilisation de “ broadcast ” physiques, les messages `ARP` ne franchissent pas les routeurs. Il existe cependant un cas particulier : le proxy ARP, que nous évoquerons succintement à la fin de ce paragraphe.

2.2 Format du datagramme

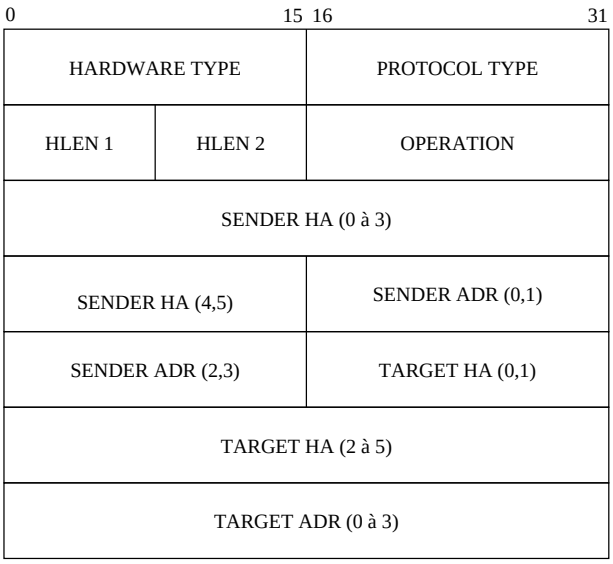


figure IV.08 — Datagramme ARP

Le datagramme ci-dessus est encapsulé dans une trame physique du type `0x0806`[4].

`HARDWARE TYPE` pour spécifier le type d'adresse physique dans les champs `SENDER HA` et `TARGET HA`, c'est 1 pour Ethernet.

`PROTOCOL TYPE` pour spécifier le type d'adresse logique dans les champs `SENDER ADR` et `TARGET ADR`, c'est `0x0800` (même valeur que dans la trame Ethernet) pour des adresses IP.

`HLEN 1` pour spécifier la longueur de l'adresse physique (6 octets pour Ethernet).

`HLEN 2` pour spécifier la longueur de l'adresse logique (4 octets pour IP).

`OPERATION` ce champ précise le type de l'opération, il est nécessaire car la trame est la même pour toutes les opérations des deux protocoles qui l'utilisent.

	Question	Réponse
`ARP`	1	2
`RARP`	3	4

`SENDER HA` adresse physique de l'émetteur

`SENDER ADR` adresse logique de l'émetteur

`TARGET HA` adresse physique du destinataire

`TARGET ADR` adresse logique du destinataire

[4] voir ou revoir la *figure II.02* du chapitre d'introduction à IP (page 23)

2.3 Proxy ARP

Le proxy ARP permet l'extension du lan à des hôtes qui ne lui sont pas directement physiquement reliés, mais qui s'y rattachent par exemple au travers d'une passerelle.

Un exemple très courant est celui d'un hôte qui accède à un réseau via un dialup (rtc, numéris,...). Le NetID de son adresse IP peut alors être le même que celui du réseau rejoint, comme s'il y était physiquement raccordé. Ce subterfuge est rendu possible après configuration adéquate de la passerelle de raccordement.

3 Protocole RARP

RARP est l'acronyme de " Reverse Address Resolution Protocol ", il est défini dans la RFC 903 (BOOTP et DHCP en sont des alternatives avec plus de possibilités).

- Normalement une machine qui démarre obtient son adresse IP par lecture d'un fichier sur son disque dur (ou depuis sa configuration figée dans une mémoire non volatile).
- Pour certains équipements cette opération n'est pas possible voire même non souhaitée par l'administrateur du réseau :
  - Terminaux X Windows
  - Stations de travail " diskless "
  - Imprimante en réseau
  - " Boites noires " sans capacité autonome de démarrage
  - PC en réseau
  - ...
- Pour communiquer en TCP/IP une machine a besoin d'au moins une adresse IP, l'idée de ce protocole est de la demander au réseau.
- Le protocole RARP est adapté de ARP : l'émetteur envoie une requête RARP spécifiant son adresse physique dans un datagramme de même format que celui de ARP et avec une adresse de " broadcast " physique. Le champ OPERATION contient alors le code de " RARP question "
- Toutes les stations en activité reçoivent la requête, celles qui sont habilités à répondre (serveurs RARP) complètent le datagramme et le renvoient directement (" unicast ") à l'émetteur de la requête puisqu'elle connaissent son adresse physique.

Sur une machine Unix configurée en serveur RARP les correspondances entres adresses IP et adresses physiques sont enregistrées dans un fichier nommé généralement /etc/bootptab.

4 Protocole ICMP

ICMP est l'acronyme de " Internet Control Message Protocol ", il est historiquement défini dans la RFC 950.

Nous avons vu que le protocole IP ne vérifie pas si les paquets émis sont arrivés à leur destinataire dans de bonnes conditions.

Les paquets circulent d'une passerelle vers un autre jusqu'à en trouver une qui puisse les délivrer directement à un hôte. Si une passerelle ne peut router ou délivrer directement un paquet ou si un évenement anormal arrive sur le réseau comme un trafic trop important ou une machine indisponible, il faut pouvoir en informer l'hôte qui a émis le paquet. Celui-ci pourra alors réagir en fonction du type de problème rencontré.

ICMP est un mécanisme de contrôle des erreurs au niveau IP, mais la *figure II.02* du chapitre d'introduction à IP (page 23) montre que le niveau *Application* peut également avoir un accès direct à ce protocole.

4.1 Le système de messages d'erreur

Dans le système que nous avons décrit, chaque passerelle et chaque hôte opère de manière autonome, route et délivre les datagrammes qui arrivent sans coordination avec l'émetteur.

Le système fonctionne parfaitement si toutes les machines sont en ordre de marche et si toutes les tables de routage sont à jour. Malheureusement c'est une situation idéale...

Il peut y avoir des rupture de lignes de communication, des machines peuvent être à l'arrêt, en pannes, déconnectées du réseau ou incapables de router les paquets parcequ'en surcharge.

Des paquets IP peuvent alors ne pas être délivrés à leur destinataire et le protocol IP lui-même ne contient rien qui puisse permettre de détecter cet échec de transmission.

C'est pourquoi est ajouté systématiquement un mécanisme de gestion des erreurs connu sous le doux nom de ICMP. Il fait partie de la couche IP[5] et porte le numéro de protocole **1**.

Ainsi, quand un message d'erreur arrive pour un paquet émis, c'est la couche IP elle-même qui gère le problème, la plupart des cas sans en informer les couches supérieures (certaines applications utilisent ICMP[6]).

Initialement prévu pour permettre aux passerelles d'informer les hôtes sur des erreurs de transmission, ICMP n'est pas restreint aux échanges passerelles-hôtes, des échanges entres hôtes sont tout à fait possibles.

Le même mécanisme est valable pour les deux types d'échanges.

[5] voir ou revoir la *figure II.02* du chapitre d'introduction à IP (page 23)

[6] Même figure qu'au point précédent

4.2 Format des messages ICMP

Chaque message ICMP traverse le réseau dans la partie DATA d'un datagramme IP :

figure IV.09 — Message ICMP

La conséquence directe est que les messages `ICMP` sont routés comme les autres paquets IP au travers le réseau. Il y a toutefois une exception : il peut arriver qu'un paquet d'erreur rencontre lui-même un problème de transmission, dans ce cas on ne génère pas d'erreur sur l'erreur !

Il est important de bien voir que puisque les messages `ICMP` sont encapsulés dans un datagramme `IP`, `ICMP` n'est pas considéré comme un protocole de niveau plus élevé.

La raison de l'utilisation d'`IP` pour délivrer de telles informations, est que les messages peuvent avoir à traverser plusieurs réseaux avant d'arriver à leur destination finale. Il n'était donc pas possible de rester au niveau physique du réseau (à l'inverse de `ARP` ou `RARP`).
La *figure IV.10* décrit le format du message `ICMP` :

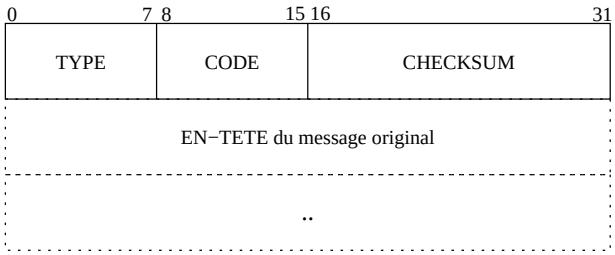


figure IV.10 — Format d'un message ICMP

Chaque message `ICMP` a un type particulier qui caractérise le problème qu'il signale. Un en-tête de 32 bits est composé comme suit :

`TYPE` contient le code d'erreur.

`CODE` complète l'information du champ précédent.

`CHECKSUM` est utilisé avec le même mécanisme de vérification que pour les datagrammes `IP` mais ici il ne porte que sur le message `ICMP` (rappel : le checksum de l'en-tête IP ne porte que sur son en-tête et non sur les données véhiculées).

En addition, les messages `ICMP` donnent toujours l'en-tête `IP` et les 64 premiers bits (les deux premiers mots de quatre octets) du datagramme qui est à l'origine du problème, pour permettre au destinataire du message d'identifier quel paquet est à l'origine du problème.

4.3 Quelques types de messages `ICMP`

Ce paragraphe examine quelques uns des principaux types de messages `ICMP`, ceux qui sont le plus utilisés. Il existe onze valeurs de `TYPE` différentes.

“ Echo Request (8), Echo reply (0) ” Une machine envoie un message `ICMP` “ echo request ” pour tester si son destinataire est accessible. N'importe quelle machine qui reçoit une telle requête doit formuler un message `ICMP` “ echo reply ” en retour[7]

Ce mécanisme est extrêmement utile, la plupart des implémentations le propose sous forme d'un utilitaire (`ping` sous Unix).

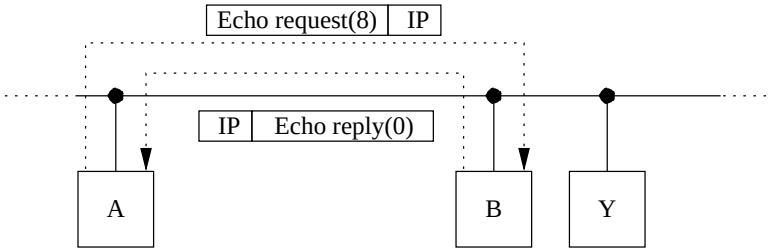


figure IV.11 — “ Echo request ” vs “ Echo reply ”

“ Destination Unreachable (3) ” Quand une passerelle ne peut pas délivrer un datagramme IP, elle envoie un message `ICMP` “ destination unreachable ” à l'émetteur.

Dans ce cas le champ `CODE` complète le message d'erreur avec :

0 “ Network unreachable ”

1 “ Host unreachable ”

2 “ Protocol unreachable ”

3 “ Port unreachable ”

4 “ Fragmentation needed and DF set ”

5 “ Source route failed ”

“ Source Quench (4) ” Quand un datagramme IP arrive trop vite pour une passerelle ou un hôte, il est rejeté.

Un paquet arrive “ trop vite ” quand la machine qui doit le lire est congestionnée, trop de trafic à suivre....

Dans ce cas la machine en question envoie un paquet `ICMP` “ source quench ” qui est interprété de la façon suivante :

L'émetteur ralenti le rythme d'envoi de ses paquets jusqu'à ce qu'il cesse de recevoir ce message d'erreur. La vitesse est donc ajustée par une sorte d'apprentissage rustique. Puis graduellement il augmente le débit, aussi longtemps que le message “ source quench ” ne revient pas .

[7] Pour des raisons de sécurité certaines machines peuvent ne pas répondre.

Ce type de paquet `ICMP` a donc tendance à vouloir réguler le flux des datagrammes au niveau IP alors que c'est une fonctionnalité de la couche de transport (`TCP`).

C'est donc une sérieuse entorse à la règle d'indépendance des couches.

“ Redirect (5) ” Les tables de routage (Voir le paragraphe 6) des stations restent assez statiques durant de longues périodes. Le système d'exploitation les lit au démarrage sur le système de fichiers et l'administrateur en change de temps en temps les éléments.

Si entre deux modifications une destination change d'emplacement, la donnée initiale dans la table de routage peut s'avérer incorrecte.

Les passerelles connaissent de bien meilleures routes que les hôtes eux-mêmes, ainsi quand une passerelle détecte une erreur de routage, elle fait deux choses :

1. Elle envoie à l'émetteur du paquet un message ICMP “ redirect ”
2. Elle redirige le paquet vers la bonne destination.

Cette redirection ne règle pas les problèmes de routage car elle est limitée aux interactions entres passerelles et hôtes directement connectés.

La propagation des routes aux travers des réseaux multiples est un autre problème.

Le champ `CODE` du message `ICMP` peut avoir les valeurs suivantes :

0 “ Redirect datagram for the Net ”

1 “ Redirect datagram for the host ”

2 ...

“ Router solicitation (10) vs Router advertisement (9) ” Il s'agit d'obtenir ou d'annoncer des routes, nous verrons cela plus en détail dans le paragraphe 6.4.

“ Time exceeded (11) ” Chaque datagramme contient un champ TTL dit “ TIME TO LIVE ” appellé aussi “ hop count ”.

Afin de prévenir le cas ou un paquet circulerait à l'infini d'une passerelle à une autre, chaque passerelle décrémente ce compteur et rejette le paquet quand le compteur arrive à zéro et envoie un message `ICMP` à l'émetteur pour le tenir au courant.

5 Protocole IGMP

IGMP, l'acronyme de " Internet Group Management Protocol ", est historiquement défini dans l'Annexe I de la RFC 1112.

Sa raison d'être est que les datagrammes ayant une adresse multicast[8] sont à destination d'un groupe d'utilisateurs dont l'émetteur ne connait ni le nombre ni l'emplacement. L'usage du multicast étant par construction dédié aux applications comme la radio ou la vidéo sur le réseau[9], donc consommatrices de bande passante, il est primordial que les routeurs aient un moyen de savoir s'il y a des utilisateurs de tel ou tel groupe sur les LANs directement accessibles pour ne pas encombrer les bandes passantes associées avec des flux d'octets que personne n'utilise plus !

5.1 Description de l'en-tête

IGMP est un protocole de communication entre les routeurs susceptibles de transmettre des datagrammes multicast et des hôtes qui veulent s'enregistrer dans tel ou tel groupe. IGMP est encapsulé dans IP[10] avec le protocole numéro **2**. Comme le montre la *figure IV.12*, sa taille est fixe (contrairement à ICMP) : seulement 2 mots de 4 octets.

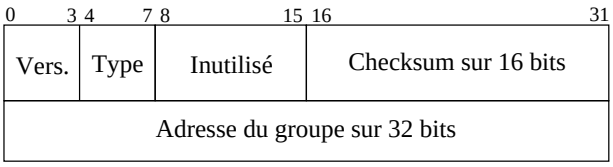


figure IV.12 — En-tête IGMP

Version Version 1.

Type Ce champ prend deux valeurs, **1** pour dire qu'il s'agit d'une question (`query` d'un routeur), **2** pour dire qu'il s'agit de la réponse d'un hôte.

Inutilisé ...

Checksum Le checksum est calculé comme celui d'ICMP.

Adresse C'est l'adresse multicast (classe D) à laquelle appartient l'hôte qui répond.

[8] Voir page 40

[9] La première expérience à grande échelle du multicast fut sans doute la conférence de l'IETF en mars 1992. Le papier `ftp ://venera.isi.edu/ietf-audiocast-article.ps` relate cette expérience.

[10] voir ou revoir la *figure II.02* du chapitre I d'introduction à IP (page 23)

5.2 Fonctionnement du protocole

La RFC 1112 précise que les routeurs multicast envoient des messages de questionnement (`Type=Queries`) pour reconnaître quels sont les éventuels hôtes appartenant à quel groupe. Ces questions sont envoyées à tous les hôtes des LANs directement raccordés à l'aide de l'adresse multicast du groupe `224.0.0.1`[11] encapsulé dans un datagramme IP ayant un champ `TTL=1`. Tous les hôtes susceptibles de joindre un groupe multicast écoutent ce groupe par hypothèse.

Les hôtes, dont les interfaces ont été correctement configurées, répondent à une question par autant de réponses que de groupes auxquels ils appartiennent sur l'interface réseau qui a reçu la question. Afin d'éviter une " tempête de réponses " chaque hôte met en œuvre la stratégie suivante :

1. Un hôte ne répond pas immédiatement à la question reçue. Pour chaque groupe auquel il appartient, il attend un délais compris entre 0 et 10 secondes, calculé aléatoirement à partir de l'adresse IP unicast de l'interface qui a reçu la question, avant de renvoyer sa réponse. La *figure IV.13* montre un tel échange, remarquez au passage la valeur des adresses.
2. La réponse envoyée est écoutée par tous les membres du groupe appartenant au même LAN. Tout ceux qui s'apprétaient à envoyer une telle réponse au serveur en interrompent le processus pour éviter une redite. Le routeur ne reçoit ainsi qu'une seule réponse pour chaque groupe, et pour chaque LAN, ce qui lui suffit pour justifier le routage demandé.

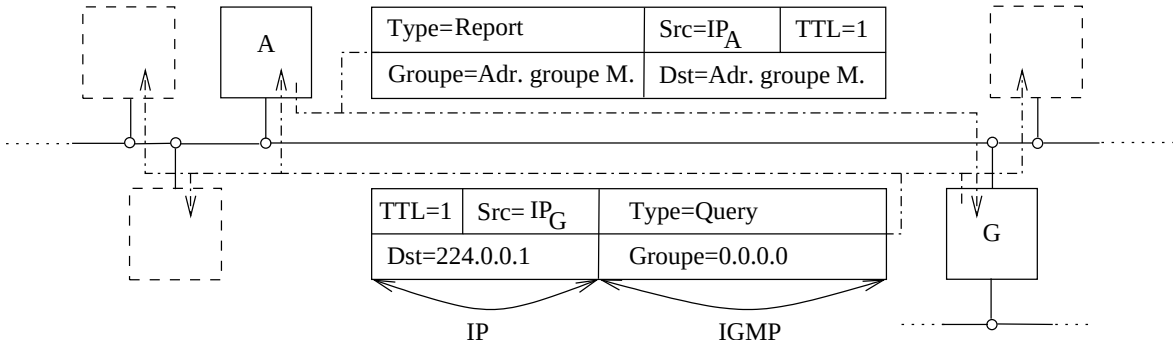


figure IV.13 — Fonctionnement IGMP

Il y a deux exceptions à la stratégie ci-dessus. La première est que si une question est reçue alors que le compte à rebours pour répondre à une réponse est en cours, il n'est pas interrompu.

La deuxième est qu'il n'y a jamais de délai appliqué pour l'envoi de datagramme portant l'adresse du groupe de base `224.0.0.1`.

Pour rafraîchir leur connaissance des besoins de routage les routeurs envoient leurs questions avec une fréquence très faible de l'ordre de la minute,

[11] " tous les hôtes du LAN "

afin de préserver au maximum la bande passante du réseau. Si aucune réponse ne leur parvient pour tel ou tel groupe demandé précédement, le routage s'interrompt.

Quand un hôte rejoint un groupe, il envoie immédiatement une réponse (`type=report`) pour le groupe (les) qui l'intéresse, plutôt que d'attendre une question du routeur. Au cas où cette réponse se perdrait il est recommandé d'effectuer une réémission dans un court délai.

Remarques :

1. Sur un LAN sans routeur pour le multicast, le seul trafic `IGMP` est celui des hôtes demandant à rejoindre tel ou tel groupe.
2. Il n'y a pas de `report` pour quitter un groupe.
3. La plage d'adresses multicast entre `224.0.0.0` et `224.0.0.225` est dédiée aux applications utilisant une valeur de 1 pour le champ `TTL` (administration et services au niveau du LAN). Les routeurs ne doivent pas transmettre de tels datagrammes.
4. Il n'y a pas de message `ICMP` sur les datagrammes ayant une adresse de destination du type multicast.

   En conséquence les applications qui utilisent le multicast (avec une adresse supérieure à `224.0.0.225`) pour découvrir des services, doivent avoir une stratégie pour augmenter la valeur du champ `TTL` en cas de non réponse.

5.3 Fonctionnement du Mbone

Précisions en cours...

6 Routage IP

Ce paragraphe décrit de manière succincte le routage des datagrammes. Sur l'Internet, ou au sein de toute entité qui utilise IP, les datagrammes ne sont pas routés par des machines Unix, mais par des routeurs dont c'est la fonction par définition. Ils sont plus efficaces et plus perfectionnés pour cette tâche par construction, et surtout autorisent l'application d'une politique de routage (" routing policy ") ce que la pile IP standard d'une machine Unix ne sait pas faire. Toutefois il est courant dans les " petits réseaux ", ou quand le problème à résoudre reste simple, de faire appel à une machine Unix pour ce faire[12].

Nous examinerons donc le problème (simplifié) du routage dans ce dernier cas. Les lecteurs soucieux d'en savoir plus sur ce problème très vaste pourront se reporter à la bibliographie en fin de chapitre.

Le routage des datagrammes se fait au niveau de la couche `IP`, et c'est son travail le plus important. Toutes les machines multiprocessus sont théoriquement capables d'effectuer cette opération.

La différence entre un " routeur " et un " hôte " est que le premier est capable de transmettre un datagramme d'un interface à un autre et pas le deuxième.

Cette opération est délicate si les machines qui doivent dialoguer sont connectées à de multiples réseaux physiques.

D'un point de vue idéal établir une route pour des datagrammes devrait tenir compte d'éléments comme la charge du réseau, la taille des datagrammes, le type de service demandé, les délais de propagation, l'état des liaisons, le trajet le plus court... La pratique est plus rudimentaire !

Il s'agit de transporter des datagrammes aux travers de multiples réseaux physiques, donc aux travers de multiples passerelles.

On divise le routage en deux grandes familles :

Le routage direct Il s'agit de délivrer un datagramme à une machine raccordée au même LAN.

L'émetteur trouve l'adresse physique du correspondant (`ARP`), encapsule le datagramme dans une trame et l'envoie.

Le routage indirect Le destinataire n'est pas sur le même LAN comme précédement. Il est absolument nécessaire de franchir une passerelle connue d'avance ou d'employer un chemin par défaut.

En effet, toutes les machines à atteindre ne sont pas forcément sur le même réseau physique. C'est le cas le plus courant, par exemple sur l'Internet qui regroupe des centaines de milliers de réseaux différents.

Cette opération est beaucoup plus délicate que la précédente car il faut sélectionner une passerelle.

[12] on peut consulter par exemple `http ://www.freebsd.org/~picobsd/`, où le site du projet Zebra de GNU `http ://www.zebra.org/`

Parceque le routage est une opération fondamentalement orientée “ réseau ”, le routage s'appuie sur cette partie de l'adresse `IP` du destinataire. La couche `IP` détermine celle-ci en examinant les bits de poids fort qui conditionnent la classe d'adresse et donc la segmentation “ network.host ”. Dans certain cas (`CIDR`) le masque de sous réseau est aussi employé.

Muni de ce numéro de réseau, la couche `IP` examine les informations contenues dans sa table de routage :

6.1 Table de routage

Sous Unix toutes les opérations de routage se font grâce à une table, dite “ table de routage ”, qui se trouve dans le noyau lui-même. La *figure IV.14* résume la situation :

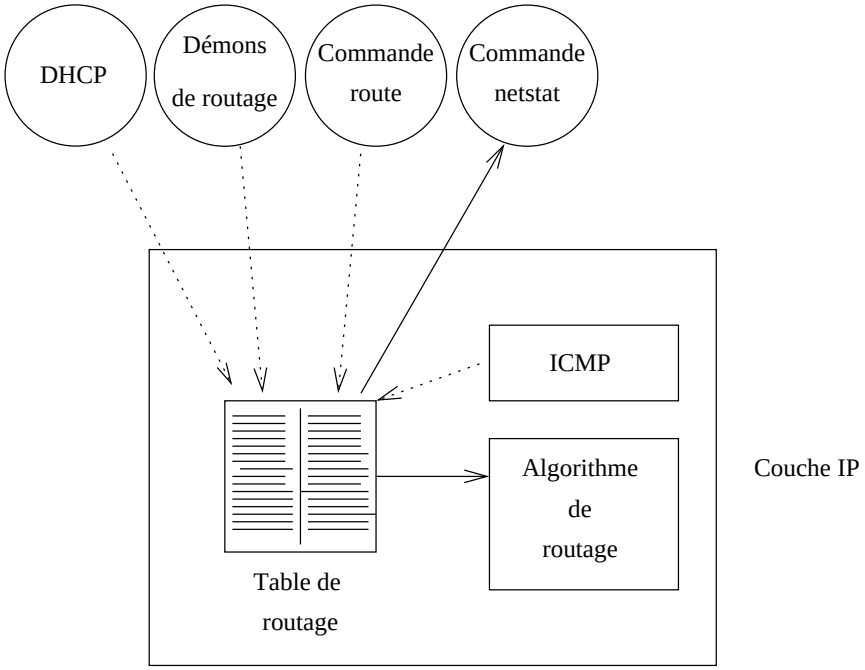


figure IV.14 — Table de routage

Cette table est très fréquemment utilisée par `IP` : sur un serveur plusieurs centaines de fois par secondes.

Comment est-elle crée ?

Au démarrage avec la commande `route`, invoquée dans les scripts de lancement du système, et en fonctionnement :

- Au coup par coup avec la commande `route`, à partir du shell (administrateur système uniquement).
- Dynamiquement avec les démons de routage “ `routed` ” ou “ `gated` ” (la fréquence de mise à jour est typiquement de l'ordre de 30 sec.).

– Par des messages “ ICMP redirect ”.

La commande `netstat -rn` permet de la visualiser au niveau de l'interface utilisateur (“ Application layer ”) :

```
$ netstat -rn
Routing tables

Internet:
Destination          Gateway            Flags
default              192.168.192.36     UGS
127.0.0.1            127.0.0.1          UH
192.168.192/27       link#1             UC
192.168.192.10       8:0:9:85:76:9c     UHLW
192.168.192.11       8:0:9:85:76:bd     UHLW
192.168.192.12       8:0:9:88:8e:31     UHLW
192.168.192.13       8:0:9:a:f9:bc      UHLW
192.168.192.14       0:4f:49:1:28:22    UHLW
192.168.192.15       link#1             UHLW
192.168.192.32/27    link#2             UC
192.168.192.33       8:0:9:70:44:52     UHLW
192.168.192.34       0:20:af:2f:8f:f1   UHLW
192.168.192.35       0:4f:49:1:36:50    UHLW
192.168.192.36       link#2             UHLW
```

On peut mémoriser cette table comme étant essentiellement composée d'une colonne origine, d'une colonne destination.

De plus, chaque route qui désigne une passerelle (ici la route par défaut) doit s'accompagner d'un nombre de sauts (“ hop ”), ou encore métrique, qui permet le choix d'une route plutôt qu'une autre en fonction de cette valeur. Chaque franchissement d'un routeur compte pour un saut. Dans la table ci-dessus, la métrique de la route par défaut est 1.

Remarque : la sortie de la commande `netstat -rn` ci-dessus a été simplifiée.[13]

Les drapeaux (“ flags ”) les plus courants :

`C c`	La route est générée par la machine, à l'usage.
`D`	La route a été crée dynamiquement (démons de routage).
`G`	La route désigne une passerelle, sinon c'est une route directe.
`H`	La route est vers une machine, sinon elle est vers un réseau.
`L`	Désigne la conversion vers une adresse physique (cf `ARP`).
`M`	La route a été modifiée par un “ redirect ”.
`S`	La route a été ajoutée manuellement.
`U`	La route est active.
`W`	La route est le résultat d'un clônage.

[13] Des colonnes `Refs`, `Use` et `Netif`

La *figure 15* précise l'architecture du réseau autour de la machine sur laquelle a été exécuté le `netstat`.

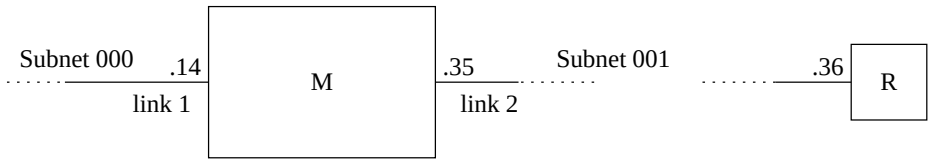


figure IV.15 — Situation réseau lors du `netstat`

6.2 Routage statique

Comme nous avons pu le deviner au paragraphe précédent, les routes statiques sont celles crées au démarrage de la machine ou ajoutées manuellement par l'administeur système, en cours de fonctionnement.

Le nombre de machines possibles à atteindre potentiellement sur l'Internet est beaucoup trop élevé pour que chaque machine puisse espérer en conserver l'adresse, qui plus est, même si cela était concevable, cette information ne serait jamais à jour donc inutilisable.

Plutôt que d'envisager la situation précédente on préfère restreindre l'étendue du " monde connu " et utiliser la " stratégie de proche en proche " précédement citée.

Si une machine ne peut pas router un datagramme, elle connait (ou est supposée connaître) l'adresse d'une passerelle supposée être mieux informée pour transmettre ce datagramme.

Dans l'exemple de sortie de la commande `netstat` du paragraphe 6.1, on peut reconnaître que l'administrateur système n'a configuré qu'une seule route " manuellement "[14], toutes les autres lignes ont été déduites par la couche `IP` elle-même.

La *figure IV.16* met en situation plusieurs réseaux et les passerelles qui les relient.

[14]Ce n'est pas tout à fait exact, nous verrons pourquoi au paragraphe concernant l'interface de " `loopback` " (6.6).

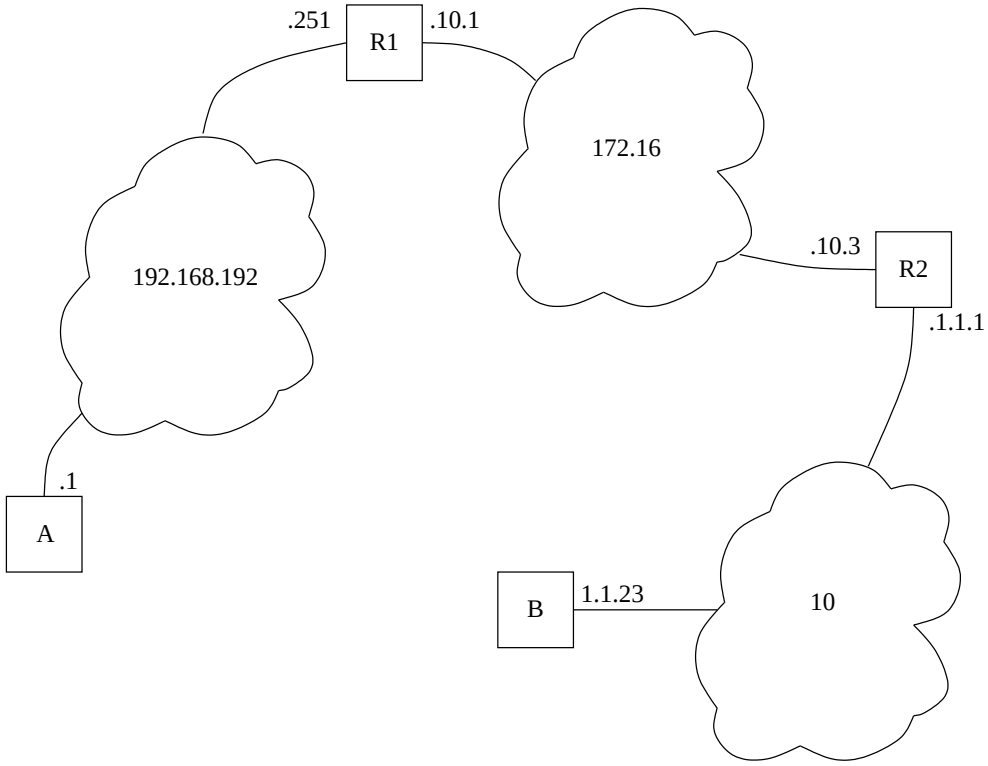


figure IV.16 — Exemple pour routage statique

Et voici une version très simplifiée des tables de routage statiques :

Machine A default : 192.168.192.251

Machine B default : 10.1.1.1

Routeur R1 10 : 172.16.10.3

Routeur R2 192.168.192 : 172.16.10.1

Algorithme de routage

Cet algorithme simplifié résume les opérations de la couche `IP` pour choisir une destination, en fonction de sa table de routage. Cette opération est essentiellement basée sur le numéro de réseau, I_N, extrait de l'adresse `IP`, I_D. M désigne la machine sur laquelle s'effectue le routage.

Si I_N est un numéro de réseau auquel M est directement reliée :
- Obtenir l'adresse physique de la machine destinatrice
- Encapsuler le datagramme dans une trame physique et l'envoyer directement.

Sinon Si I_D apparait comme une machine à laquelle une route spéciale est attribuée, router le datagramme en fonction.

Sinon Si I_N apparait dans la table de routage, router le datagramme en fonction.

Sinon S'il existe une route par défaut router le datagramme vers la passerelle ainsi désignée.

Sinon Déclarer une erreur de routage (`ICMP`).

6.3 Routage dynamique

Si la topologie d'un réseau offre la possibilité de plusieurs routes pour atteindre une même destination, s'il est vaste et complexe, sujet à des changements fréquents de configuration...Le routage dynamique est alors un bon moyen d'entretenir les tables de routages et de manière automatique.

Il existe de nombreux protocoles de routage dynamique dont certains sont aussi anciens que l'Internet. Néanmoins tous ne conviennent pas à tous les types de problème, il en existe une hiérarchie.

Schématiquement on peut imaginer l'Internet comme une hiérarchie de routeurs. Les routeurs principaux (" core gateways ") de cette architecture utilisent entres-eux des protocoles comme GGP (" Gateway to Gateway Protocol "), l'ensemble de ces routeurs forment ce que l'on nomme l'" Internet Core ".

En bordure de ces routeurs principaux se situent les routeurs qui marquent la frontière avec ce que l'on nomme les " Autonomous systems ", c'est à dire des systèmes de routeurs et de réseaux qui possèdent leurs mécanismes propres de propagation des routes. Le protocole utilisé par ces routeurs limitrophes est souvent EGP (" Exterior Gateway Protocol ") ou BGP (" Border Gateway Protocol ").

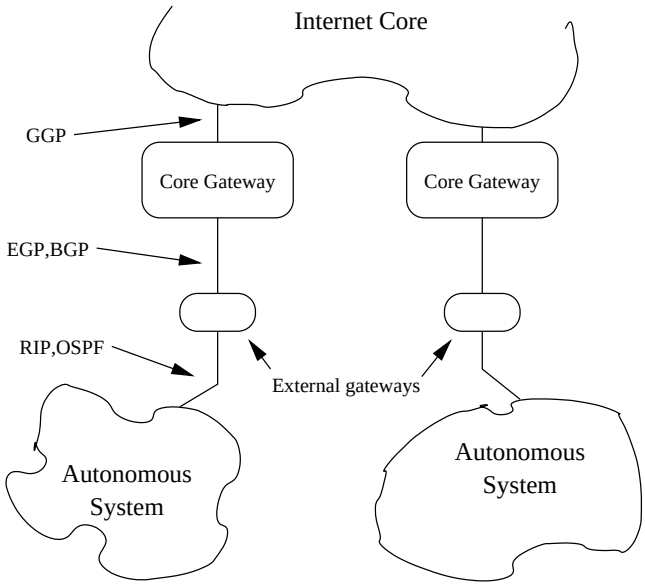


figure IV.17 — Exemple pour routage dynamique

Au sein d'un système autonome on utilise un IGP (" Interior Gateway Protocol ") c'est à dire un " protocole de gateways intérieurs ". Les protocoles

les plus couramment employés sont RIP (“ Routing Information Protocol ”) qui est simple à comprendre et à utiliser, ou encore OSPF (“ Open Shortest Path First ”) plus récent, plus capable mais aussi beaucoup plus complexe à comprendre dans son mode de fonctionnement, ou encore IS-IS de la couche ISO de l'OSI.

RIP — “ Routing Information Protocol ”

RIP est apparu avec la version BSD d'Unix, il est documenté dans la RFC 1058 (1988 - Version 1 du protocole) et la RFC 1388 (1993 - Version 2 du protocole). Ce protocole est basé sur des travaux plus anciens menés par la firme Xerox.

RIP utilise le concept de “ vecteur de distance ”, qui s'appuie sur un algorithme de calcul du chemin le plus court dans un graphe. Le graphe est celui des routeurs, la longueur du chemin est établie en nombre de sauts (“ hop ”), ou métrique, entre la source et la destination, c'est à dire en comptant toutes les liaisons. Cette distance est exprimée comme un nombre entier variant entre 1 et 15 ; la valeur 16 est considérée comme l'infini et indique une mise à l'écart de la route.

Chaque routeur émet dans un datagramme portant une adresse IP de broadcast, à fréquence fixe (environ 30 secondes), le contenu de sa table de routage et écoute celle des autres routeurs pour compléter sa propre table. Ainsi se propagent les tables de routes d'un bout à l'autre du réseau. Pour éviter une “ tempêtes de mises à jours ”, le délais de 30 secondes est augmenté d'une valeur aléatoire comprise entre 1 et 5 secondes.

Si une route n'est pas annoncée au moins une fois en trois minutes, la distance devient “ infinie ”, et la route sera retirée un peu plus tard de la table (elle est propagée avec cette métrique).

L'adresse IP utilisée est une adresse de multipoint (“ multicast ”) comme nous verrons au paragraphe 6.4

Depuis la définition de RIPv2 les routes peuvent être accompagnées du masque de sous réseau qui les caractérise. Ainsi on peut avoir la situation suivante :

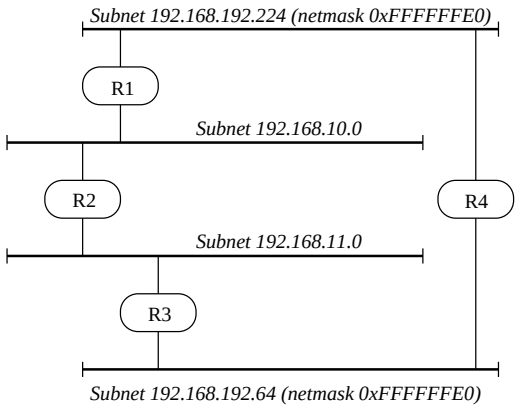


figure IV.18 — Topologie pour routage dynamique

Après propagation des routes, la table de routage du routeur R1 pourrait bien ressembler à :

Source	Destination	Coût
192.168.192.224	R1	1
192.168.10.0	R1	1
192.168.11.0	R2	2
192.168.192.64	R3	3

Avec une route par défaut qui est le routeur R2. La constitution de cette table n'est possible qu'avec RIPv2 étant donné l'existence des deux sous-réseaux de la classe C 192.168.192.

OSPF — " Open Shortest Path First "

Contrairement à RIP, OSPF n'utilise pas de vecteur de distances mais base ses décisions de routage sur le concept d'" état des liaisons ". Celui-ci permet un usage beaucoup plus fin des performances réelles des réseaux traversés, parceque cette métrique est changeante au cours du temps. Si on ajoute à cela une méthode de propagation très rapide des routes par inondation, sans boucle et la possibilité de chemin multiples, OSPF, bien que beaucoup plus complexe que RIP, a toutes les qualités pour le remplacer, même sur les tous petits réseaux.

OSPF doit son nom à l'algorithme d'Edsger W. Dijkstra[15] de recherche du chemin le plus court d'abord lors du parcours d'un graphe. Le " Open " vient du fait qu'il s'agit d'un protocole ouvert de l'IETF, dans la RFC 2328...

Des travaux pratiques viendront compléter l'information sur RIP et OSPF !

6.4 Découverte de routeur et propagation de routes

Au démarrage d'une station, plutôt que de configurer manuellement les routes statiques, surtout si elle sont susceptibles de changer et que le nombre de stations est grand, il peut être intéressant de faire de la " découverte automatique de routeurs " (RFC 1256).

À intervals réguliers les routeurs diffusent des messages `ICMP` de type 9 (" router advertisement ") d'annonces de routes. Ces messages ont l'adresse multicast 224.0.0.1, qui est a destination de tous les hôtes du LAN.

Toutes les stations capables de comprendre le multicast (et convenablement configurées pour ce faire) écoutent ces messages et mettent à jour leur table.

Les stations qui démarrent peuvent solliciter les routeurs si l'attente est trop longue (environ 7 minutes) avec un autre message `ICMP`, de type 10 (" router sollicitation ") et avec l'adresse multicast 224.0.0.2 (à destination

[15] `http ://www.cs.utexas.edu/users/EWD/`

de tous les routeurs de ce LAN). La réponse du ou des routeurs est du type " unicast ", sauf si le routeur s'apprêtait à émettre une annonce.

À chaque route est associé un niveau de préférence et une durée de validité, définis par l'administrateur du réseau. Une validité nulle indique un routeur qui s'arrête et donc une route qui doit être supprimée.

Si entre deux annonces une route change, le mécanisme de " `ICMP` redirect ", examiné au paragraphe suivant, corrige l'erreur de route.

La découverte de routeur n'est pas un protocole de routage, son objectif est bien moins ambitieux : obtenir une route par défaut.

6.5 Message ICMP " redirect "

La table de routage peut être modifiée dynamiquement par un message `ICMP` (IV).

La situation est celle de la *figure IV.21.*

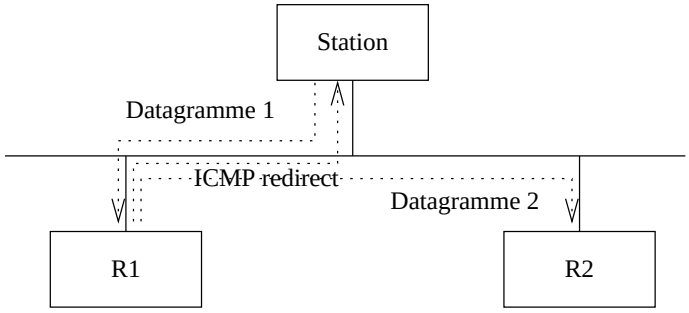


figure IV.21 — ICMP " redirect "

- La station veut envoyer un datagramme et sa table de routage lui commande d'utiliser la route qui passe par le routeur R1.
- Le routeur R1 reçoit le datagramme, scrute sa table de routage et s'apperçoit qu'il faut désormais passer par R2. Pour se faire :

  1. Il re-route le datagramme vers R2, ce qui évite qu'il soit émis deux fois sur le LAN.
  2. Il envoie un message " `ICMP` redirect " (type 5) à la station, lui indiquant la nouvelle route vers R2.

  Ce travail s'effectue pour chaque datagramme reçu de la station.
- Dès que la station reçoit le message " `ICMP` redirect " elle met à jour sa table de routage. La nouvelle route est employée pour les datagrammes qui suivent (vers la même direction).
  La route modifiée est visible avec la commande `netstat -r`, elle figure avec le drapeau 'M' (modification dynamique).

Pour des raisons évidentes de sécurité, cette possibilité n'est valable que sur un même LAN.

6.6 Interface de “ loopback ”

Toutes les implémentations d'IP supportent une interface de type “ loopback ”. L'objet de cette interface est de pouvoir utiliser les outils du réseau en local, sans passer par un interface réseau réel (associé à une carte physique).

La *figure IV.22* ci-contre, montre que la couche IP peut utiliser, selon le routage, l'interface standard du réseau, où l'interface de loopback.

Le routage est ici bien sûr basé sur l'adresse IP associée à chacune des interfaces. Cette association est effectuée sur une machine Unix à l'aide de la commande ifconfig, qui établit une correspondance entre un pilote de périphérique (repéré par son fichier spécial) et une adresse IP.

Dans le cas du pilote de loopback, l'adresse est standardisée à n'importe quelle adresse valide du réseau 127 (page 35).

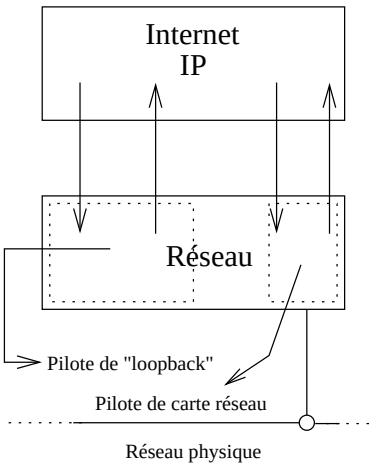


figure IV.22 — Interface de “ loopback ”

La valeur courante est 127.0.0.1, d'où l'explication de la ligne ci-dessous déjà rencontrée (page 64) dans le cadre de la table de routage :

```
Routing tables
Internet:
Destination        Gateway            Flags  Netif
...
127.0.0.1          127.0.0.1          UH      lo0
...
```

Dans toutes les machines Unix modernes cette configuration est déjà prévue d'emblée dans les scripts de démarrage.

Concrètement, tout dialogue entre outils clients et serveurs sur une même machine est possible et même souhaitable sur cet interface pour améliorer les performances et parfois la sécurité[16].

L'exemple d'usage le plus marquant est sans doute celui du serveur de noms (voir page 105) qui tient compte explicitement de cet interface dans sa configuration.

[16]Nous verrons ultérieurement (cf chapitre X) que le filtrage IP sur le 127/8 est aussi aisé que sur n'importe quel autre réseau

7 Finalement, comment ça marche ?

Dans ce paragraphe nous reprenons la *figure III.06* (page 42) et nous y apportons comme était annoncé une explication du fonctionnement qui tienne compte des protocoles principaux examinés dans ce chapitre. Pour cela nous utilisons deux réseaux privés de la RFC 1918 : 192.168.10.0 et 192.168.20.0 et nous faisons l'hypothèse que la passerelle fonctionne comme une machine Unix qui ferait du routage entre deux de ses interfaces !

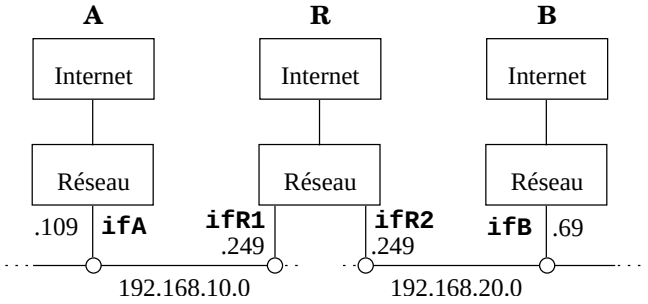


figure IV.23 — Illustration du routage direct et indirect

Ce tableau résume l'adressage physique et logique de la situation :

Interface	Adresse MAC	Adresse IP
`ifA`	`08:00:20:20:cf:af`	192.168.10.109
`ifB`	`00:01:e6:a1:07:64`	192.168.20.69
`ifR1`	`00:06:5b:0f:5a:1f`	192.168.10.249
`ifR2`	`00:06:5b:0f:5a:20`	192.168.20.249

Nous faisons en outre les hypothèses suivantes :

1. Les caches " arp " des machines **A**, **B** et **R** sont vides
2. La machine **A** a connaissance d'une route vers le réseau 192.168.20 passant par 192.168.10.249 et réciproquement la machine **B** voit le réseau 192.168.10.0 via le 192.168.20.249
3. La machine **A** a connaissance de l'adresse IP de la machine **B**

La machine A envoie un datagramme à la machine B, que se passe t-il sur le réseau ?

Étape 1 La machine **A** applique l'algorithme de routage (page 67) et s'aperçoit que la partie réseau de l'adresse de **B** n'est pas dans le même LAN (192.168.10/24 et 192.168.20/20 diffèrent).

L'hypothèse 2 entraine qu'une route route existe pour atteindre ce réseau, passant par **R**. L'adresse IP de **R** est dans le même LAN, **A** peut donc atteindre **R** par un routage direct. La conséquence de l'hypothèse 1 implique que pour atteindre **R** directement il nous faut d'abord déterminer son adresse physique. Le protocole ARP (page 52) doit être utiliser.

A envoie en conséquence une trame ARP (page 54 comportant les éléments suivants :

SENDER HA	08:00:20:20:cf:af
SENDER ADR	192.168.10.109
TARGET HA	ff:ff:ff:ff:ff:ff
TARGET ADR	192.168.10.249

Avec un champ OPERATION qui contient la valeur 1, comme " question ARP ".

Remarquez qu'ici l'adresse IP destination est celle de **R** !

Étape 2 **R** répond à la " question ARP " par une " réponse ARP " (OPERATION contient 2) et un champ complèté :

SENDER HA	00:06:5b:0f:5a:1f
SENDER ADR	192.168.10.249
TARGET HA	08:00:20:20:cf:af
TARGET ADR	192.168.10.109

Étape 3 **A** est en mesure d'envoyer son datagramme à **B** en passant par **R**. Il s'agit de routage indirect puisque l'adresse de **B** n'est pas sur le même LAN. Les adresses physiques et logiques se répartissent maintenant comme ceci :

IP SOURCE	192.168.10.109
IP TARGET	192.168.20.69
MAC SOURCE	08:00:20:20:cf:af
MAC TARGET	00:06:5b:0f:5a:1f

Remarquez qu'ici l'adresse IP destination est celle de **B** !

Étape 4 **R** a reçu le datagramme depuis **A** et à destination de **B**. Celle-ci est sur un LAN dans lequel **R** se trouve également, un routage direct est donc le moyen de transférrer le datagramme. Pour la même raison qu'à l'étape 1 **R** n'a pas l'adresse MAC de **B** et doit utiliser ARP pour obtenir cette adresse. Voici les éléments de cette " question ARP " :

SENDER HA	00:06:5b:0f:5a:20
SENDER ADR	192.168.20.249
TARGET HA	ff:ff:ff:ff:ff:ff
TARGET ADR	192.168.20.69

Étape 5 Et la " réponse ARP " :

SENDER HA	00:01:e6:a1:07:64
SENDER ADR	192.168.20.69
TARGET HA	00:06:5b:0f:5a:20
TARGET ADR	192.168.20.249

Étape 6 Enfin, dans cette dernière étape, **R** envoie le datagramme en provenance de **A**, à **B** :

IP SOURCE	192.168.10.109
IP TARGET	192.168.20.69
MAC SOURCE	00:06:5b:0f:5a:20
MAC TARGET	00:01:e6:a1:07:64

Remarque, comparons avec le datagramme de l'étape 3. Si les adresses IP n'ont pas changé, les adresses MAC, diffèrent complètement !

Remarque : Si **A** envoie un deuxième datagramme, les caches ARP ont les adresses MAC utiles et donc les étape 1, 2, 4 et 5 deviennent inutiles...

8 Conclusion sur IP

Après notre tour d'horizon sur IPv4 nous pouvons dire en conclusion que son espace d'adressage trop limité n'est pas la seule raison qui a motivé les travaux de recherche et développement d'IPv6 :

1. Son en-tête comporte deux problèmes, la somme de contrôle (checksum) doit être calculée à chaque traitement de datagramme, chaque routeur doit analyser le contenu du champ option.
2. Sa configuration nécessite au moins trois informations que sont l'adresse, le masque de sous réseau et la route par défaut.
3. Son absence de sécurité est insupportable. Issu d'un monde fermé où la sécurité n'était pas un problème, le datagramme de base n'offre aucun service de confidentialité, d'intégrité et d'authentification.
4. Son absence de qualité de servive ne répond pas aux exigences des protocoles applicatifs modernes (téléphonie, vidéo, jeux interactifs en réseau, ...). Le champ TOS n'est pas suffisant et surtout est interprété de manière inconsistante par les équipements.

9 Bibliographie

Pour en savoir plus :

RFC 791 “Internet Protocol.” J. Postel. Sep-01-1981. (Format : TXT=97779 bytes) (Obsoletes RFC0760) (Status : STANDARD)

RFC 826 “Ethernet Address Resolution Protocol : Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware.” D.C. Plummer. Nov-01-1982. (Format : TXT=22026 bytes) (Status : STANDARD)

RFC 903 “Reverse Address Resolution Protocol.” R. Finlayson, T. Mann, J.C. Mogul, M. Theimer. Jun-01-1984. (Format : TXT=9345 bytes) (Status : STANDARD)

RFC 950 “Internet Standard Subnetting Procedure.” J.C. Mogul, J. Postel. Aug-01-1985. (Format : TXT=37985 bytes) (Updates RFC0792) (Status : STANDARD)

RFC 1112 “Host extensions for IP multicasting.” S.E. Deering. Aug-01-1989. (Format : TXT=39904 bytes) (Obsoletes RFC0988, RFC1054) (Updated by RFC2236) (Also STD0005) (Status : STANDARD)

RFC 1256 “ICMP Router Discovery Messages. S. Deering.” Sep-01-1991. (Format : TXT=43059 bytes) (Also RFC0792) (Status : PROPOSED STANDARD)

- W. Richard Stevens - TCP/IP Illustrated, Volume 1 - The protocols - Addison-Wesley
- Douglas Comer - Internetworking with TCP/IP - Principles, protocols, and architecture - Prentice–Hall
- Craig Hunt - TCP/IP Network Administration - O´Reilly & Associates, Inc.
- Christian Huitema - Le routage dans l’Internet - EYROLLES

Chapitre V

Protocole UDP

1 UDP – User Datagram Protocol

`UDP` est l'acronyme de "**U**ser **D**atagram **P**rotocol", il est défini dans la RFC 768 [Postel 1980]. Les données encapsulées dans un en-tête `UDP` sont des "paquets `UDP`".

1.1 Identification de la destination

Rappel : Au niveau de la couche *Internet* les datagrammes sont routés d'une machine à une autre en fonction des bits de l'adresse `IP` qui identifient le numéro de réseau. Lors de cette opération aucune distinction n'est faite entre les services ou les utilisateurs qui émettent ou recoivent des datagrammes, ie tous les datagrammes sont mélangés.

La couche `UDP` ajoute un mécanisme qui permet l'identification du service (niveau *Application*). En effet, il est indispensable de faire un tri entre les divers applications (services) : plusieurs programmes de plusieurs utilisateurs peuvent utiliser simultanément la même couche de transport et il ne doit pas y avoir de confusion entre eux.

Pour le système `Unix` les programmes sont identifiés de manière unique par un numéro de processus, mais ce numéro est éphémère, non prévisible à distance, il ne peut servir à cette fonction.

L'idée est d'associer la destination à la fonction qu'elle remplie. Cette identification se fait à l'aide d'un **entier positif** que l'on baptise **port**.

- Le système d'exploitation local a à sa charge de définir le mécanisme qui permet à un processus d'accéder à un port.
- La plupart des systèmes d'exploitation fournissent le moyen d'un accès synchrone à un port. Ce logiciel doit alors assurer la possibilité de gérer la file d'attente des paquets qui arrivent, jusqu'à ce qu'un processus (*Application*) les lise. A l'inverse, l'OS[1] bloque un processus qui tente de lire une donnée non encore disponible.

[1] "Operating System"

Pour communiquer avec un service distant il faut donc avoir connaissance de son numéro de port, en plus de l'adresse `IP` de la machine elle-même.

On peut prévoir le numéro de port en fonction du service à atteindre, c'est l'objet du paragraphe 1.3.

La *figure 1* explicite la notion de port. La couche `IP` sépare les datagrammes `TCP` et `UDP` grâce au champ `PROTO`[2] de son en-tête, l'association du protocole de transport et du numéro de port identifie un service sans ambiguïté.

Conceptuellement on s'apperçoit alors que rien ne s'oppose à ce qu'un même service (Numéro de port) soit attribué conjointement aux deux protocoles (pointillé sur la figure). Cette situation est d'ailleurs courante dans la réalité des serveurs.

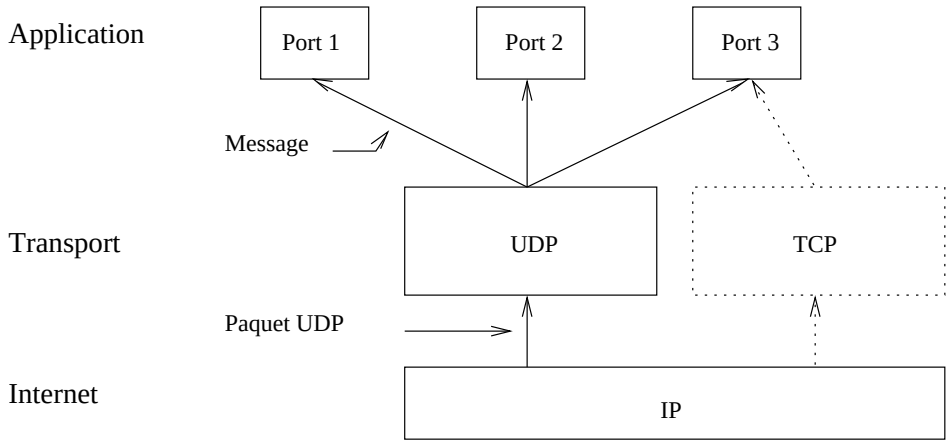


figure V.01 — Numéro de port comme numéro de service

[2]Cf "Protocole IP" page 45

1.2 Description de l'en-tête

Un paquet UDP est conçu pour être encapsulé dans un datagramme IP et permettre un échange de données entre deux applications, sans échange préliminaire. Ainsi, si les données à transmettre n'obligent pas IP à fragmenter (cf page 49), un paquet UDP génère un datagramme IP et c'est tout !

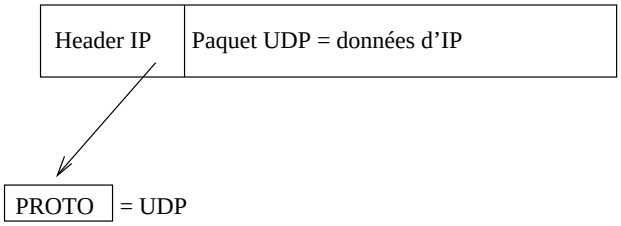


figure V.02 — UDP encapsulé dans IP

- UDP apporte un mécanisme de gestion des ports, au dessus de la couche Internet.
- UDP est simplement une interface au dessus d'IP, ainsi l'émission des messages se fait-elle sans garantie de bon acheminement. Plus généralement, tous les défauts d'IP recensés au chapitre précédent sont applicables à UDP.
  Plus particulièrement, les paquets à destination d'une application UDP sont conservés dans une pile de type FIFO. Si l'application destinatrice ne les "consomme" pas assez rapidement, les plus anciens paquets risquent d'être écrasés par les plus récents...Un risque supplémentaire (par rapport aux propriétés d'IP déjà connues) de perte de données.
- Il n'y a aucun retour d'information au niveau du protocole pour apporter un quelconque moyen de contrôle sur le bon acheminement des données.
  C'est au niveau applicatif qu'il convient de prendre en compte cette lacune.
- UDP est aussi désigné comme un mode de transport "non connecté", ou encore mode datagramme, par opposition à TCP que nous examinerons dans le chapitre suivant.

Parmis les utilisations les plus courantes d'UDP on peut signaler le serveur de noms[3], base de données répartie au niveau mondial, et qui s'accomode très bien de ce mode de transport.

En local d'autres applications très utiles comme **tftp** ou **nfs** sont également susceptibles d'employer UDP.

[3]DNS — RFC 1035— Ce service utilise UDP dans le cas d'échanges de petits paquets d'informations ($\leq$ 512 octets) sinon il utilise TCP

La *figure V.03* décrit la structure de l'en-tête.

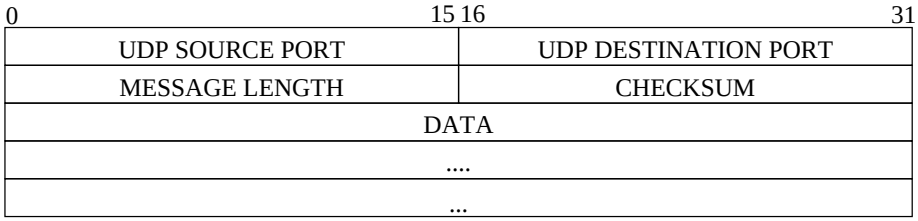


figure V.03 — Structure de l'en-tête UDP

`UDP SOURCE PORT` Le numéro de port de l'émetteur du paquet. Ce champ est optionnel, quand il est spécifié il indique le numéro de port que le destinataire doit employer pour sa réponse. La valeur zéro (0) indique qu'il est inutilisé, le port 0 n'est donc pas celui d'un service valide.

`UDP DESTINATION PORT` Le numéro de port du destinataire du paquet.

`MESSAGE LENGTH` C'est la longueur du paquet, donc comprennant l'en-tête et le message.

- La longueur minimal est 8
- La longueur maximale est $65\,535 - H(IP)$. Dans le cas courant (`IP` sans option) cette taille maximale est donc de 65 515.

`CHECKSUM` Le checksum est optionnel et toutes les implémentations ne l'utilisent pas. S'il est employé, il porte sur un pseudo en-tête constitué de la manière suivante :

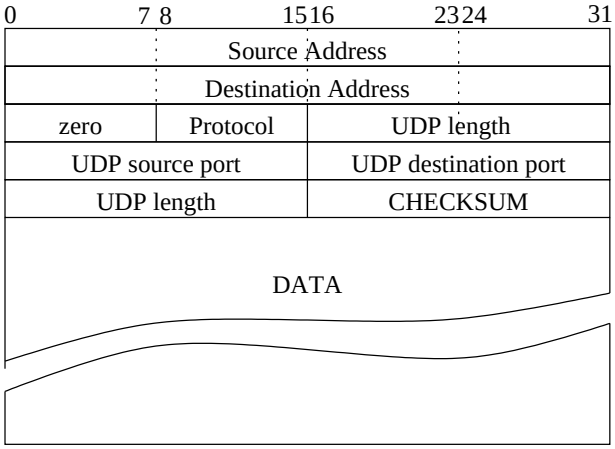


figure V.04 — Cas du checksum non nul

Ce pseudo en-tête est prévu initialement pour apporter une protection en cas de datagrammes mal routés !

1.3 Ports réservés — ports disponibles

Le numéro de port est un entier 16 bits non signé, les bornes sont donc $[0, 65535]$, par construction. Nous avons vu précédement que le port 0 n'est pas exploitable en tant que désignation de service valide, donc le segment réellement exploitable est $[1, 65535]$.

Toute machine qui utilise la pile `TCP/IP` se doit de connaître un certain nombre de services bien connus, repérés par une série de ports bien connus ou "well known port numbers", pour pouvoir dialoguer avec les autres machines de l'Internet (vs Intranet). Sur une machine `Unix`, cette liste de services est placée dans le fichier `/etc/services` et lisible par tous les utilisateurs et toutes les applications.

En effet, comme nous l'examinerons en détail dans le cours de programmation, un service (comprendre un programme au niveau applicatif) qui démarre son activité réseau (et qui donc est considéré comme ayant un rôle de serveur) s'attribue le (les) numéro(s) de port qui lui revient (reviennent) conformément à cette table.

Nom	Port	Proto	Commentaire
echo	7	tcp	
echo	7	udp	
ftp-data	20	tcp	`#File Transfer [Default Data]`
ftp-data	20	udp	`#File Transfer [Default Data]`
ftp	21	tcp	`#File Transfer [Control]`
ftp	21	udp	`#File Transfer [Control]`
telnet	23	tcp	
telnet	23	udp	
smtp	25	tcp	`mail #Simple Mail Transfer`
smtp	25	udp	`mail #Simple Mail Transfer`
domain	53	tcp	`#Domain Name Server`
domain	53	udp	`#Domain Name Server`
http	80	tcp	`www www-http #World Wide Web HTTP`
http	80	udp	`www www-http #World Wide Web HTTP`
pop3	110	tcp	`#Post Office Protocol - Version 3`
pop3	110	udp	`#Post Office Protocol - Version 3`
sunrpc	111	tcp	`rpcbind #SUN Remote Procedure Call`
sunrpc	111	udp	`rpcbind #SUN Remote Procedure Call`
nntp	119	tcp	`usenet #Network News Transfer Protocol`
nntp	119	udp	`usenet #Network News Transfer Protocol`

figure V.05

Le tableau de la *figure V.05* indique quelques uns des plus connus et plus utilisés ports bien connus, il y en a quantité d'autres...

Une autorité, l'IANA[4], centralise et diffuse l'information relative à tous les nombres utilisés sur l'Internet via une RFC. La dernière en date est la RFC 1700, elle fait plus de 200 pages !

Par voie de conséquence cette RFC concerne aussi les numéros de ports.

Attribution des ports "ancienne méthode"

Historiquement les ports de 1 à 255 sont réservés aux services bien connus, plus récemment, ce segment à été élargi à $[1, 1023]$. Aucune application ne peut s'attribuer durablement et au niveau de l'Internet un numéro de port dans ce segment, sans en référrer à l'IANA, qui en contrôle l'usage.

À partir de 1024 et jusqu'à 65535, l'IANA se contente d'enregistrer les demandes d'usage et signale les éventuels conflits.

Attribution des ports "nouvelle méthode"

Devant l'explosion du nombre des services enregistrés l'IANA a modifié la segmentation[5] qui précède. Désormais les numéros de ports sont classés selon les trois catégories suivantes :

1. **Le segment** $[1, 1023]$ est toujours réservés aux services bien connus.
   Les services bien connus sont désignés par l'IANA et sont mis en œuvre par des applications qui s'exécutent avec des droits privilégiés (**root** sur une machine `Unix`)
2. **Le segment** $[1024, 49151]$ est celui des services enregistrés.
   Ils sont énumérés par l'IANA et peuvent être employés par des processus ayant des droits ordinaires.
   Par exemple :

Nom	Port	Proto	Commentaire
bpcd	13782	tcp	VERITAS NetBackup
bpcd	13782	udp	VERITAS NetBackup

3. **Le segment** $[49152, 65535]$ est celui des attributions dynamiques et des services privés ; nous en examinerons l'usage dans le cours de programmation.

[4] "Internet Assigned Numbers Authority"

[5] `http ://www.iana.org/assignments/port-numbers`

2 Bibliographie

Pour en savoir plus :

RFC 768 “User Datagram Protocol.” J. Postel. Aug-28-1980. (Format : TXT=5896 bytes) (Status : STANDARD)

RFC 1035 “Domain names - concepts and facilities.” P.V. Mockapetris. Nov-01-1987. (Format : TXT=129180 bytes) (Obsoletes RFC0973, RFC0882, RFC0883) (Obsoleted by RFC1065, RFC2065) (Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC2065, RFC2181) (Status : STANDARD)

RFC 1700 “ASSIGNED NUMBERS.” J. Reynolds,J. Postel. October 1994. (Format : TXT=458860 bytes) (Obsoletes RFC1340) (Also STD0002) (Status : STANDARD)

RFC 1918 “Address Allocation for Private Internets.” Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot & E. Lear. February 1996. (Format : TXT=22270 bytes) (Obsoletes RFC1627, RFC1597) (Also BCP0005) (Status : BEST CURRENT PRACTICE)

Sans oublier :

– W. Richard Stevens - TCP/IP Illustrated, Volume 1 - The protocols - Addison-Wesley — 1994
– Douglas Comer - Internetworking with TCP/IP - Principles, protocols, and architecture - Prentice–Hall

Chapitre VI

Protocole TCP

1 TCP – Transport Control Protocol

TCP est l'acronyme de " **T**ransport **C**ontrol **P**rotocol ", il est défini dans la RFC 793 [Postel 1981c]. Les données encapsulées dans un en-tête TCP sont des " paquets TCP ".

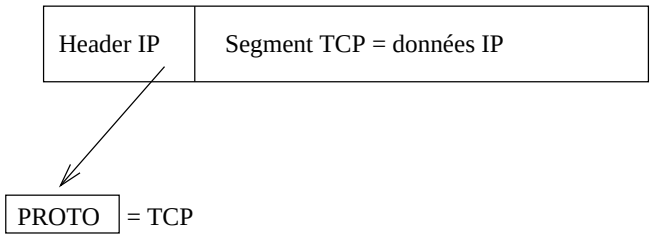


figure VI.01 — TCP encapsulé dans IP

1.1 Caractéristiques de TCP

TCP est bien plus compliqué[1] qu'UDP examiné au chapitre précédent, il apporte en contrepartie des services beaucoup plus élaborés.
Cinq points principaux caractérisent ce protocole :

1. TCP contient un mécanisme pour assurer **le bon acheminement des données**. Cette possibilité est absolument indispensable dès lors que les applications doivent transmettre de gros volumes de données et de façon fiable.

   Il faut préciser que les paquets de données sont acquittés de bout en bout et non de point en point. D'une manière générale le réseau assure l'acheminement et les extrémités le contrôle (Dave Clark).

2. Le protocole TCP permet l'établissement d'**un circuit virtuel** entre les deux points qui échangent de l'information. On dit aussi que TCP

[1] Une simple comparaison du volume des RFC initiales est parlante : 85 pages pour TCP, 3 pour UDP !

fonctionne en mode connecté (par opposition à `UDP` qui est en mode non connecté ou encore mode datagramme).

- Avant le transfert les 2 applications se mettent en relation avec leurs OS[2] respectifs, les informent de leurs désirs d'établir ou de recevoir une communication.
- Pratiquement, l'une des deux applications doit effectuer un appel que l'autre doit accepter.
- Les protocoles des 2 OS communiquent alors en s'envoyant des messages au travers du réseau pour vérifier que le transfert est possible (autorisé) et que les deux applications sont prêtes pour leurs rôles.
- Une fois ces préliminaires établis, les modules de protocole informent les applications respectives que la connexion est établie et que le transfert peut débuter.
- Durant le transfert, le dialogue entre les protocoles continue, pour vérifier le bon acheminement des données.

Conceptuellement, pour établir une connexion — un circuit virtuel — il faut avoir réunis les éléments du quintuplet :

Le protocole C'est `TCP` mais il y pourrait y avoir d'autres transports qui assurent le même service...

IP locale Adresse de la machine qui émet.

Port local Le numéro de port associé au processus. Il est imposé ou est déterminé automatiquement comme nous le verrons dans le cours de programmation.

IP distante Adresse de la machine distante.

Port distant Le numéro de port associé au service à atteindre. Il est obligatoire de le connaître précisement.

L'ensemble de ces cinq éléments définit un circuit virtuel unique. Que l'un d'eux change et il s'agit d'une autre connexion !

3. `TCP` a la capacité de **mémoriser[3] des données** :
   - Aux deux extrémités du circuit virtuel, les applications s'envoient des volumes de données absolument quelconques, allant de 0 octet à des centaines (ou plus) de Mo.
   - À la réception, le protocole délivre les octets exactement comme ils ont été envoyés.
   - Le protocole est libre de fragmenter le flux de données en paquets de tailles adaptées aux réseaux traversés. Il lui incombe cependant d'effectuer le réassemblage et donc de stocker temporairement les fragments avant de les présenter dans le bon ordre à l'application.
4. `TCP` est **indépendant vis à vis des données transportées**, c'est un flux d'octets non structuré sur lequel il n'agit pas.

[2] " Operating System "
[3] dans un buffer

5. TCP simule une connexion en " **full duplex** ". Pour chacune des deux applications en connexion par un circuit virtuel, l'opération qui consiste à lire des données peut s'effectuer indépendamment de celle qui consiste à en écrire.

   Le protocole autorise la clôture du flot dans une direction tandis que l'autre continue à être active. Le circuit virtuel est rompu quand les deux parties ont clos le flux.

1.2 Description de l'en-tête

La figure suivante montre la structure d'un en-tête TCP. Sa taille normale est de 20 octets, à moins que des options soient présentes.

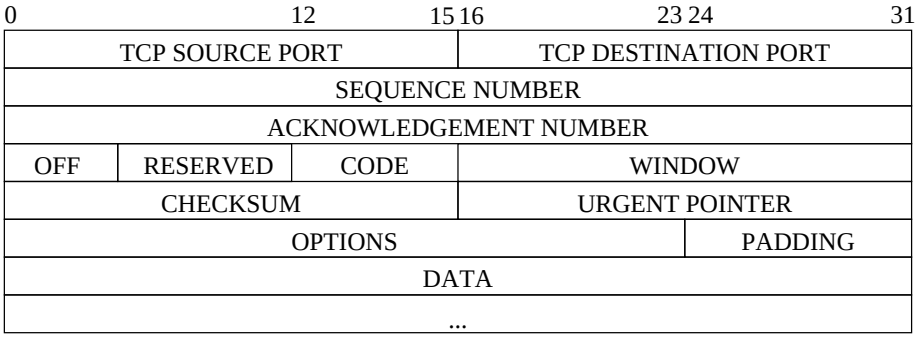


figure VI.02 — Structure de l'en-tête TCP

`TCP SOURCE PORT` Le numéro de port de l'application locale.

`TCP DESTINATION PORT` Le numéro de port de l'application distante.

`SEQUENCE NUMBER` C'est un nombre qui identifie la position des données **à transmettre** par rapport au segment original. Au démarrage de chaque connexion, ce champ contient une valeur non nulle et non facilement prévisible, c'est la séquence initiale ou ISN[4]

TCP numérote chaque octet transmis en incrémentant ce nombre 32 bits non signé. Il repasse à 0 après avoir atteint $2^{32} - 1$ (4 294 967 295).

Pour le premier octet des données transmis ce nombre est incrémenté de un, et ainsi de suite...

`ACKNOWLEDGEMENT NUMBER` C'est un numéro qui identifie la position du **dernier octet reçu** dans le flux entrant.

Il doit s'accompagner du drapeau `ACK` (voir plus loin).

`OFF` pour `OFFSET`, il s'agit d'un déplacement qui permet d'atteindre les données quand il y a des options. Codé sur 4 bits, il s'agit du nombre de mots de 4 octets qui composent l'en-tête. Le déplacement maximum est donc de 60 octets ($2^4 - 1 \times 4$ octets). Dans le cas d'un en-tête sans option, ce champ porte la valeur 5. 10 mots de 4 octets sont donc possibles pour les options.

[4] " Initial Sequence Number "

`RESERVED` Six bits réservés pour un usage futur !

`CODE` Six bits pour influer sur le comportement de `TCP` en caractérisant l'usage du segment :

URG	Le champ " `URGENT POINTER` " doit être exploité.
ACK	Le champ " `ACNOWLEDGMENT NUMBER` " doit être exploité.
PSH	C'est une notification de l'émetteur au récepteur, pour lui indiquer que toutes les données collectées doivent être transmises à l'application sans attendre les éventuelles données qui suivent.
RST	Re-initialisation de la connexion
SYN	Le champ " `SEQUENCE NUMBER` " contient la valeur de début de connexion.
FIN	L'émetteur du segment a fini d'émettre.

En fonctionnement normal un seul bit est activé à la fois mais ce n'est pas une obligation. La RFC 1024 [Postel 1987] décrit l'existence de paquets tcp dénommés " Christmas tree " ou " paquet kamikaze " comprenant les bits SYN+URG+PSH+FIN !

`WINDOW` Le flux `TCP` est controlé de part et d'autre pour les octets compris dans une zone bien délimitée et nommée " fenêtre ". La taille de celle-ci est définie par un entier non signé de 16 bits, qui en limite donc théoriquement la taille à 65 535 octets (ce n'est pas complètement exact, voir plus loin l'option **wscale**).

Chaque partie annonce ainsi la taille de son buffer de réception. Par construction, l'émetteur n'envoie pas plus de données que le récepteur ne peut en accepter.

Cette valeur varie en fonction de la nature du réseau et surtout de la bande passante devinée à l'aide de statistiques sur la valeur du RTT. Nous y reviendrons au paragraphe 4.

`CHECKSUM` Un calcul qui porte sur la totalité du segment, en-tête et données.

`URGENT POINTER` Ce champ n'est valide que si le drapeau `URG` est armé. Ce pointeur contient alors un offset à ajouter à la valeur de `SEQUENCE NUMBER` du segment en cours pour délimiter la zone des données urgentes à transmettre à l'application.

Le mécanisme de transmission à l'application dépend du système d'exploitation.

`OPTIONS` C'est un paramètrage de `TCP`. Sa présence est détectée dès lors que l'`OFFSET` est supérieur à 5.

Les options utilisées :

mss La taille maximale du segment[5] des données applicatives que l'émetteur accepte de recevoir. Au moment de l'établissement d'une connexion (paquet comportant le flag SYN), chaque partie annonce sa taille de `MSS`. Ce n'est pas une négociation. Pour de l'Ethernet la valeur est 1460 ($= MTU - 2 \times 20$).

timestamp pour calculer la durée d'un aller et retour (RTT ou " round trip time ").

wscale Facteur d'échelle (" shift ") pour augmenter la taille de la fenêtre au delà des 16 bits du champ `WINDOW` (> 65535).

Quand cette valeur n'est pas nulle, la taille de la fenêtre est de 65535×2^{shift}. Par exemple si " shift " vaut 1 la taille de la fenêtre est de 131072 octets soit encore 128 ko.

nop Les options utilisent un nombre quelconque d'octets par contre les paquet `TCP` sont toujours alignés sur une taille de mot de quatre octets ; à cet effet une option " `No Operation` " ou `nop`, codée sur 1 seul octet, est prévue pour compléter les mots.

`PADDING` Remplissage pour se caler sur un mot de 32 bits.

`DATAS` Les données transportées. Cette partie est de longueur nulle à l'établissement de la connexion, elle peut également être nulle par choix de l'application.

[5] `MSS`=" Maximum Segment Size "

2 Début et clôture d'une connexion

2.1 Établissement d'une connexion

L'établissement d'une connexion `TCP` s'effectue en trois temps, comme le schéma de la *figure 3* l'explicite.

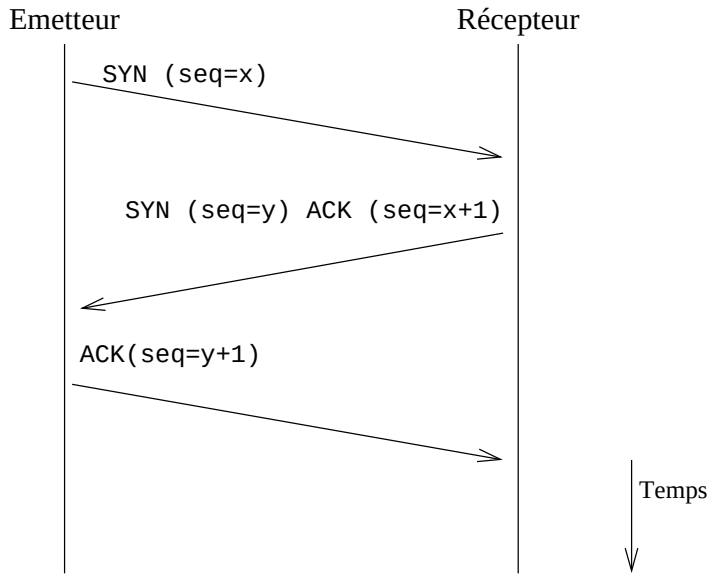


figure VI.03 — Établissement d'une connexion

On suppose que l'émetteur du premier paquet avec le bit `SYN` a connaissance du couple (adresse IP du récepteur, numéro de port du service souhaité).

L'émetteur du premier paquet est à l'origine de l'établissement du circuit virtuel, c'est une attitude généralement qualifiée de " cliente ". On dit aussi que le client effectue une " ouverture active " (*active open*).

Le récepteur du premier paquet accepte l'établissement de la connexion, ce qui suppose qu'il était prêt à le faire **avant** que la partie cliente en prenne l'initiative. C'est une attitude de " serveur ". On dit aussi que le serveur effectue une " ouverture passive " (*passive open*).

1. Le client envoie un segment comportant le drapeau `SYN`, avec sa séquence initiale ($ISN = x$).
2. Le serveur répond avec sa propre séquence ($ISN = y$), mais il doit également acquitter le paquet précédent, ce qu'il fait avec `ACK` ($seq = x + 1$).
3. Le client doit acquitter le deuxième segment avec `ACK` ($seq = y + 1$).

Une fois achevée cette phase nommée " three-way handshake ", les deux applications sont en mesure d'échanger les octets qui justifient l'établissement de la connexion.

2.2 Clôture d'une connexion

Clôture canonique

Un échange de trois segments est nécessaire pour l'établissement de la connexion ; il en faut quatre pour qu'elle s'achève de manière canonique (" orderly release ").

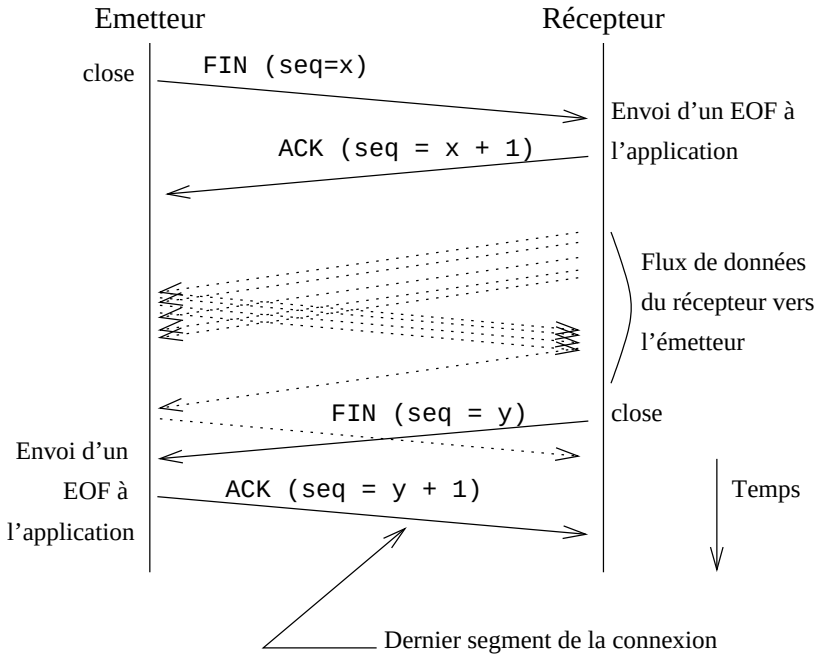


figure VI.04 — Clôture d'une connexion

La raison est qu'une connexion TCP est " full-duplex ", ce qui implique que les données circulent indépendamment dans un sens et dans l'autre. Les deux directions doivent donc pouvoir être interrompues indépendamment l'une de l'autre.

L'application qui envoie un paquet avec le drapeau FIN indique à la couche TCP de la machine distante qu'elle n'enverra plus de donnée. La machine distante doit acquitter ce segment, comme il est indiqué sur la *figure VI.04*, en incrémentant d'une unité le " sequence number ".

La connexion est véritablement terminée quand les deux applications ont effectué ce travail. Il y a donc échange de 4 paquets pour terminer la connexion.

Au total, sans compter les échanges propres au transfert des données, les deux couches `TCP` doivent gérer 7 paquets, il faut en tenir compte lors de la conception des applications !

Sur la figure on constate que le serveur continue d'envoyer des données bien que le client ait terminé ses envois. Le serveur a détecté cette attitude par la réception d'un caractère de `EOF` (en C sous `Unix`).

Cette possibilité a son utilité, notamment dans le cas des traitements distants qui doivent s'accomplir une fois toutes les données transmises, comme par exemple pour un tri.

Clôture abrupte

Au lieu d'un échange de quatre paquets comme précédement, un mécanisme de reset est prévu pour terminer une connexion au plus vite (*abortive release*).

Ce type d'arrêt est typiquement géré par la couche `TCP` elle-même quand l'application est brutalement interrompue sans avoir effectué un appel à la primitive `close(2)`, comme par exemple lors d'un appel à la primitive `abort(2)`, ou après avoir rencontré une exception non prise en compte (“ core dump ”...).

L'extremité qui arrête brutalement la connexion émet un paquet assorti du bit `RST`, après avoir (ou non) envoyé les derniers octets en attente[6]. Ce paquet clôt l'échange. Il ne reçoit aucun acquittement.

L'extrémité qui reçoit le paquet de reset (bit `RST`), transmet les éventuelles dernières données à l'application et provoque une sortie d'erreur du type “ Connection reset par peer ” pour la primitive de lecture réseau. Comme c'est le dernier échange, si des données restaient à transmettre à l'application qui a envoyé le `RST` elles peuvent être détruites.

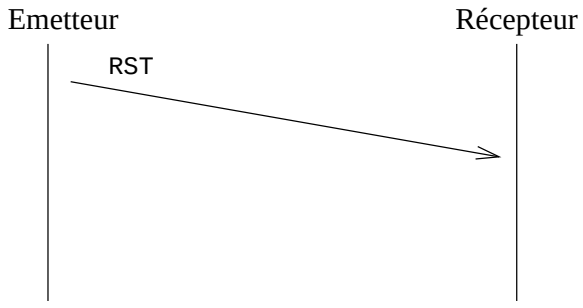


figure VI.05 — Émission d'un `rst`

[6] Voir dans le cours de programmation, l'option `SO_LINGER`

3 Contrôle du transport

Le bon acheminement des données applicatives est assuré par un mécanisme d'acquittement des paquets, comme nous avons déjà pu l'examiner partiellement au paragraphe précédent.

3.1 Mécanisme de l'acquittement

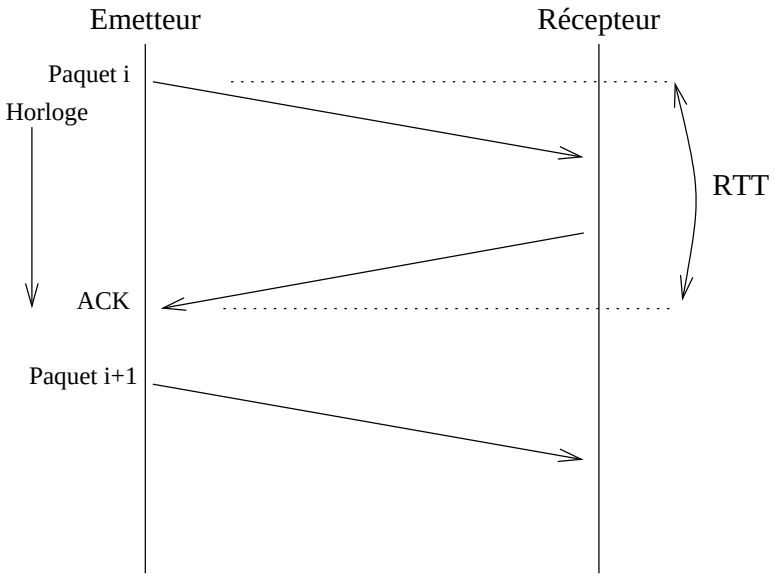


figure VI.06 — Mécanisme de l'acquittement

- Au départ du *Paquet i* une horloge se déclenche. Si cette horloge dépasse une valeur limite avant réception de l'`ACK` le *Paquet i* est retransmis Cette valeur limite est basée sur la constante MSL[7] qui est un choix d'implémentation, généralement de 30 secondes à 2 minutes. Le temps maximum d'attente est donc de $2 \times MSL$.
- Le temps qui s'écoule entre l'émission d'un paquet et la réception de son acquittement est le RTT[8], il doit donc être inférieur à $2 \times MSL$. Il est courant sur l'Internet actuel d'avoir un RTT de l'ordre de la seconde. Il faut noter que le RTT est la somme des temps de transit entre chaque routeur et du temps passé dans les diverses files d'attente sur les routeurs.
- L'émetteur conserve la trace du *Paquet i* pour éventuellement le renvoyer.

Si on considère des délais de transmission de l'ordre de 500 ms (voire plus), un tel mécanisme est totalement inadapté au transfert de flux de données. On peut aussi remarquer qu'il sous-emploi la bande passante du réseau.

[7] " Maximum Segment Lifetime "

[8] " Round Trip Time ", calculé à l'aide de l'option " timestamp " - voir page 89

3.2 Fenêtres glissantes

Cette attente de l'acquittement est pénalisante, sauf si on utilise un mécanisme de " fenêtres glissantes[9] ", comme le suggère la *figure VI.07* :

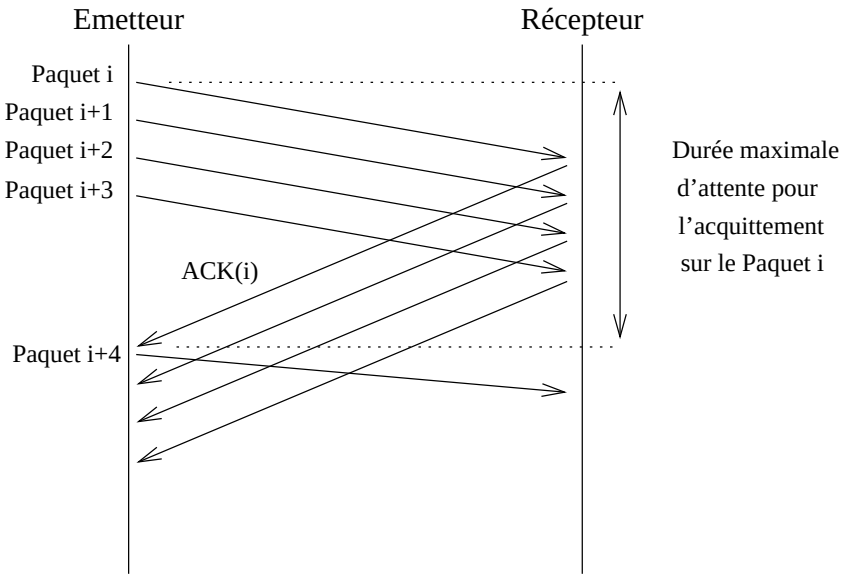


figure VI.07 — Principe de la fenêtre glissante

- Avec ce principe, la bande passante du réseau est beaucoup mieux employée.
- Si l'un des paquets doit être reémis, la couche `TCP` du destinataire aura toute l'information pour le replacer dans le bon ordre.
- À chaque paquet est associée une horloge comme sur la *figure VI.06*.
- Le nombre de paquets à envoyer avant d'attendre le premier acquittement est fonction de deux paramètres :
  1. La largeur de la fenêtre précisée dans le champ `WINDOW` de l'en-tête. Des valeurs courantes sont de l'ordre de 4096, 8192 ou 16384. Elle change dynamiquement pour deux raisons :
     (a) L'application change la taille du " buffer de la socket "[10] qui correspond à la taille de cette fenêtre.
     (b) Chaque acquittement `ACK` envoyé est assorti d'une nouvelle valeur de taille de la fenêtre, permettant ainsi à l'émetteur d'ajuster à tout instant le nombre de segment qu'il peut envoyer simultanément. Celle valeur peut être nulle, comme par exemple lorsque l'application cesse de lire les données reçues. C'est ce mécanisme qui assure le contrôle de flux de TCP.

[9] " sliding windows "
[10] voir le cours de programmation

2. La taille maximale des données, ou MSS[11] vaut 512 octets par défaut. C'est la plus grande taille du segment de données que `TCP` enverra au cours de la session.

   Le datagramme `IP` a donc une taille égale au `MSS` augmentée de 40 octets (20 + 20), en l'absence d'option de `TCP`.

   Cette option apparait uniquement dans un paquet assorti du drapeau `SYN`, donc à l'établissement de la connexion.

   Comme de bien entendu cette valeur est fortement dépendante du support physique et plus particulièrement du MTU[12].

   Sur de l'Ethernet la valeur maximale est $1500 - 2 \times 20 = 1460$, avec des trames l'encapsultation 802.3 de l'IEEE un calcul similaire conduit à une longueur de 1452 octets.

   Chaque couche `TCP` envoie sa valeur de MSS en même temps que le paquet de synchronisation, comme une option de l'en-tête. Cette valeur est calculée pour éviter absolument la fragmentation de `IP` au départ des datagrammes.

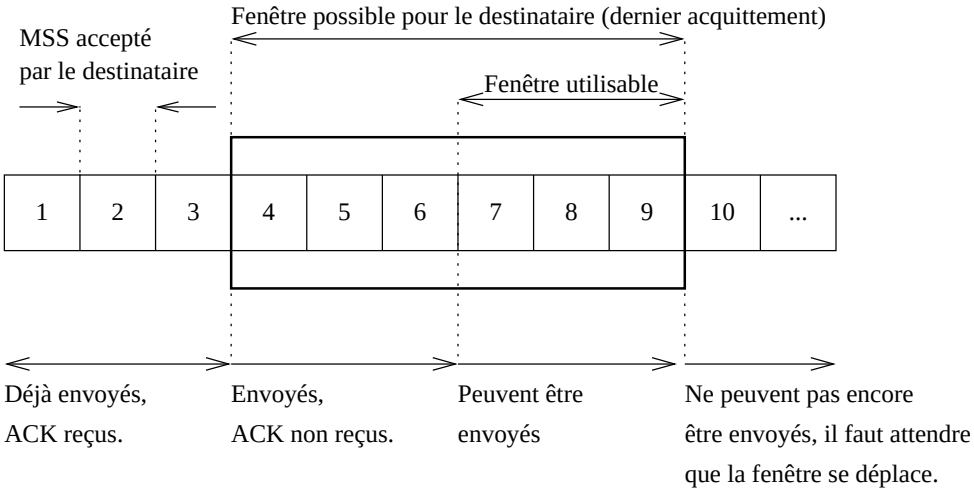


figure VI.08 — Détail de la fenêtre glissante

Le débit obtenu dépend de la taille de la fenêtre et bien sûr de la bande passante disponible. On conçoit aisément qu'entre la situation de la *figure VI.06* et celle de la *figure VI.07* l'usage de la bande passante s'améliore. Par contre l'agrandissement de la taille de la fenêtre ne se conçoit que jusqu'à une limite optimale au dela de laquelle des paquets sont perdus parcequ'envoyés trop rapidement pour être reçus par le destinataire. Or, pour fonctionner de manière optimale, `TCP` se doit de limiter au maximum la perte de paquets et donc leur reémission.

[11] " Maximum Segment Size " - Voir page 89
[12] " Maximum Transfert Unit "

Cette taille limite optimale de la largeur de la fenêtre est, comme on peut le deviner, fonction de la bande passante théorique du réseau et surtout de son taux d'occupation instantanné. Cette dernière donnée est fluctuante, aussi `TCP` doit-il asservir continuement les tailles de fenêtre pour en tenir compte.

4 Compléments sur le fonctionnement de TCP

L'usage du protocole TCP diffère considérablement en fonction des applications mises en œuvre et des réseaux à parcourir.

D'après [W. Richard Stevens], 10% des données échangées sur le réseau concernent des applications interactives et 90% des applications qui échangent des flux de données.

Si le protocole TCP reste le même pour tous, les algorithmes qui le pilotent s'ajustent en fonction de l'usage.

Pour le trafic en volume (" bulk data "), TCP tente d'utiliser des paquets les plus larges possibles pour maximiser le débit, alors que le trafic interactif utilise des paquets quasiment vides émis le plus souvent à la fréquence de frappe des utilisateurs ou au rythme des mouvements d'une souris.

Un exemple typique est celui de l'application `telnet` pour laquelle les caractères sont envoyés un à un dans un paquet différent, chaque caractère étant à l'origine de quatre paquets : émission d'un caractère, acquittement, retour de l'écho du caractère, acquittement.

Si ce comportement n'est absolument pas pénalisant sur un réseau rapide (LAN) par contre dès que la bande passante commence à être staturée il est préférable de regrouper un maximum d'octets (deux ou trois en pratique) dans un seul paquet pour en diminuer le nombre. C'est ce que fait l'algorithme de Nagle.

4.1 Algorithme de Nagle

Pour réduire le trafic de ces " tinygrams " (RFC 896), l'algorithme de Nagle (1984) dit qu'une connexion TCP ne peut pas attendre plus d'un acquittement. Deux cas se présentent donc :

1. Le réseau est lent. Dans ce cas TCP accumule dans un même buffer les octets en partance. Dès réception de l'acquittement il y a émission du contenu du buffer en un seul paquet.

2. Le réseau est rapide. Les acquittements arrivent rapidement les agrégats d'octets peuvent tendre vers un seul caractère par paquet.

La qualité lent/rapide du réseau est calculée à partir du " timestamp " envoyé dans les options de TCP et qui est établi dès le premier échange (puis reévaluée statistiquement par la suite).

L'élégance de cet algorithme est qu'il est très simple et qu'il s'auto-régule suivant le délais de propagation.

Certaines applications désactivent cet algorithme[13] comme le serveur *Apache* ou le système de multi-fenêtrage *X11*.

[13] à l'aide de l'option TCP_NODELAY

4.2 Départ lent

Un paquet est reémis parcequ'il arrive corrompu ou parcequ'il n'arrive jamais. Une reémission entraine un blocage de l'avancement de la " fenêtre glissante ", pénalisant pour le débit.

`TCP` considère qu'un paquet perdu est la conséquence d'un routeur (ou plus) congestionné, c'est à dire pour lequel les files d'attente ne sont pas assez larges pour absorber tous les paquets entrants[14]

Dans ce contexte, on comprend bien qu'il vaut mieux ne pas envoyer la totalité du contenu de la fenêtre dès le début de la connexion. Au contraire, `TCP` utilise un algorithme nommé " slow start " qui asservit l'émission des paquets au rythme de la réception de leurs acquittements, plutôt que de les émettre d'un coup aussi rapidement que l'autorise le système ou le débit théorique du réseau.

Ainsi, au début de chaque connexion ou après une période de calme (" idle ") l'émetteur envoie un premier paquet de taille maximale (le " mss " du destinataire), et attend son acquittement. Quand celui-ci est reçu, il envoie deux paquets, puis quatre, et ainsi de suite jusqu'à atteindre l'ouverture maximale de la fenêtre.

Durant cette progression, si des paquets sont perdus, il y a congestion supposée sur l'un des routeurs franchis et l'ouverture de la fenêtre est réduite pour retrouver un débit qui minimise le taux de retransmission.

L'ouverture de la fenêtre est nommée fenêtre de congestion ou encore " congestion window ".

4.3 Évitement de congestion

Le contrôle du flux évoqué précédement, pour éviter la congestion des routeurs, est implémenté à l'aide d'une variable (`cwnd`) nommée " congestion window " que l'on traduit par fenêtre de congestion.

Concrètement, le nombre maximum de segments de données ($\times MSS$ en octets) que l'émetteur peut envoyer avant d'en recevoir le premier acquittement est le minimum entre cette variable (`cwnd`) et la taille de la fenêtre annoncée par le récepteur à chaque acquittement de paquet.

Le contenu de cette variable est piloté par les algorithmes de départ lent — " slow start ", voir 4.2 — et d'évitement de congestion (" congestion avoidance ") examiné ici.

La limite de croissance de la variable `cwnd` est la taille de la fenêtre annoncée par le récepteur. Une fois la capacité de débit maximale atteinte, si un paquet est perdu l'algorithme d'évitement de congestion en diminue linéairement la valeur (contrairement au " slow start " qui l'augmente exponentiellement).

[14] Ce cas arrive fréquemment quand un routeur sépare un réseau rapide d'un réseau lent

5 Paquets capturés, commentés

Le premier exemple montre un échange de paquets de synchronisation (`SYN`) et de fin (`FIN`) entre la machine `clnt.chezmoi` et la machine `srv.chezmoi`. L'établissement de la connexion se fait à l'aide de la commande `telnet` sur le port `discard` (9) du serveur `srv.chezmoi`. La machine qui est à l'origine de l'établissement de la connexion est dite cliente, et celle qui est supposée prête à répondre, serveur. Pour information, le service `discard` peut être considéré comme l'équivalent du fichier `/dev/null` sur le réseau : les octets qu'on lui envoie sont oubliés (“ discard ”).

L'utilisateur tape :

Simultanément la capture des paquets est lancée, par exemple dans une autre fenêtre `xterm`.

```
$ telnet srv discard
  Trying...
  Connected to srv.chezmoi.
  Escape character is '^]'.

  telnet> quit
  Connection closed.
```

Et l'outil d'analyse réseau[15] permet la capture pour l'observation des échanges suivants. Le numéro qui figure en tête de chaque ligne a été ajouté manuellement, le nom de domaine “ chezmoi ” a été retiré, le tout pour faciliter la lecture :

```
0  13:52:30.274009 clnt.1159 > srv.discard: S 55104001:55104001(0)
                                                    win 8192 <mss 1460>
1  13:52:30.275114 srv.discard > clnt.1159: S 2072448001:2072448001(0)
                                          ack 55104002 win 4096 <mss 1024>
2  13:52:30.275903 clnt.1159 > srv.discard: . ack 1 win 8192
3  13:52:33.456899 clnt.1159 > srv.discard: F 1:1(0) ack 1 win 8192
4  13:52:33.457559 srv.discard > clnt.1159: . ack 2 win 4096
5  13:52:33.458887 srv.discard > clnt.1159: F 1:1(0) ack 2 win 4096
6  13:52:33.459598 clnt.1159 > srv.discard: . ack 2 win 8192
```

Plusieurs remarques s'imposent :

1. Pour améliorer la lisibilité les numéros de séquences “ vrais ” ne sont indiqués qu'au premier échange. Les suivants sont relatifs. Ainsi le `ack 1` de la ligne 2 doit être lu 2072448002 (2072448001 + 1).
   À chaque échange la valeur entre parenthèse indique le nombre d'octets échangés.
2. Les tailles de fenêtre (`win`) et de segment maximum (`mss`) ne sont pas identiques. Le `telnet` du client fonctionne sur HP-UX alors que le serveur `telnetd` fonctionne sur une machine BSD.
3. La symbole > qui marque le sens du transfert.

[15] `tcpdump` que nous aurons l'occasion d'utiliser en TP

4. Le port source `1159` et le port destination `discard`.
5. Les flags `F` et `S`. L'absence de flag, repéré par un point.

Le deuxième exemple montre une situation de transfert de fichier avec l'outil `ftp`[16].

Il faut remarquer que l'établissement de la connexion `TCP` est ici à l'initiative du serveur, ce qui peut laisser le lecteur perplexe...L'explication est simple. En fait le protocole `ftp` fonctionne avec deux connexions `TCP`, la première, non montrée ici, est établie du client vers le serveur, supposé à l'écoute sur le port 21. Elle sert au client pour assurer le contrôle du transfert. Lorsqu'un transfert de fichier est demandé via cette première connexion, le serveur établit une connexion temporaire vers le client. C'est cette connexion que nous examinons ici. Elle est cloturée dès que le dernier octet demandé est transférré.
Extrait du fichier `/etc/services`, concernant `ftp` :

```
ftp-data        20/tcp    #File Transfer [Default Data]
ftp-data        20/udp    #File Transfer [Default Data]
ftp             21/tcp    #File Transfer [Control]
ftp             21/udp    #File Transfer [Control]
```

Dans cette exemple nous pouvons suivre le fonctionnement du mécanisme des fenêtres glissantes. Les lignes ont été numérotées manuellement et la date associée à chaque paquet supprimée.

```
0   srv.20 > clnt.1158: S 1469312001:1469312001(0)
                                    win 4096 <mss 1024> [tos 0x8]
1   clnt.1158 > srv.20: S 53888001:53888001(0) ack 1469312002
                                    win 8192 <mss 1460>
2   srv.20 > clnt.1158: . ack 1 win 4096 [tos 0x8]
3   srv.20 > clnt.1158: P 1:1025(1024) ack 1 win 4096 [tos 0x8]
4   clnt.1158 > srv.20: . ack 1025 win 8192
5   srv.20 > clnt.1158: . 1025:2049(1024) ack 1 win 4096 [tos 0x8]
6   srv.20 > clnt.1158: . 2049:3073(1024) ack 1 win 4096 [tos 0x8]
7   clnt.1158 > srv.20: . ack 3073 win 8192
8   srv.20 > clnt.1158: . 3073:4097(1024) ack 1 win 4096 [tos 0x8]
9   srv.20 > clnt.1158: P 4097:5121(1024) ack 1 win 4096 [tos 0x8]
10  srv.20 > clnt.1158: P 5121:6145(1024) ack 1 win 4096 [tos 0x8]
11 clnt.1158 > srv.20: . ack 5121 win 8192
12  srv.20 > clnt.1158: P 6145:7169(1024) ack 1 win 4096 [tos 0x8]
13  srv.20 > clnt.1158: P 7169:8193(1024) ack 1 win 4096 [tos 0x8]
14  clnt.1158 > srv.20: . ack 7169 win 8192
15  srv.20 > clnt.1158: P 8193:9217(1024) ack 1 win 4096 [tos 0x8]
16  srv.20 > clnt.1158: P 9217:10241(1024) ack 1 win 4096 [tos 0x8]
17  clnt.1158 > srv.20: . ack 9217 win 8192
18  srv.20 > clnt.1158: P 10241:11265(1024) ack 1 win 4096 [tos 0x8]
19  srv.20 > clnt.1158: P 11265:12289(1024) ack 1 win 4096 [tos 0x8]
20  clnt.1158 > srv.20: . ack 11265 win 8192
... ...  ...
```

[16]du nom du protocole applicatif utilisé : " File Transfert Protocol "

```
21  srv.20 > clnt.1158: P 1178625:1179649(1024) ack 1 win 4096 [tos 0x8]
22  clnt.1158 > srv.20: . ack 1178625 win 8192
23  srv.20 > clnt.1158: P 1212417:1213441(1024) ack 1 win 4096 [tos 0x8]
24  srv.20 > clnt.1158: P 1213441:1214465(1024) ack 1 win 4096 [tos 0x8]
25  srv.20 > clnt.1158: P 1214465:1215489(1024) ack 1 win 4096 [tos 0x8]
26  clnt.1158 > srv.20: . ack 1213441 win 8192
27  clnt.1158 > srv.20: . ack 1215489 win 8192
28  srv.20 > clnt.1158: P 1215489:1215738(249) ack 1 win 4096 [tos 0x8]
29  srv.20 > clnt.1158: F 1215738:1215738(0) ack 1 win 4096 [tos 0x8]
30  clnt.1158 > srv.20: . ack 1215739 win 8192
31  clnt.1158 > srv.20: F 1:1(0) ack 1215739 win 8192
32  srv.20 > clnt.1158: . ack 2 win 4096 [tos 0x8]
```

Remarques :

1. Le `P` symbolise le drapeau `PSH`. La couche `TCP` qui reçoit un tel paquet est informée qu'elle doit transmettre à l'application toutes les données reçues, y compris celles transmises dans ce paquet.

   Le positionnement de ce drapeau est à l'initiative de la couche `TCP` émettrice et non à l'application.

2. Le type de service (" Type Of service " `tos 0x8`) est demandé par l'application pour maximiser le débit (consulter le cours `IP` page 47).

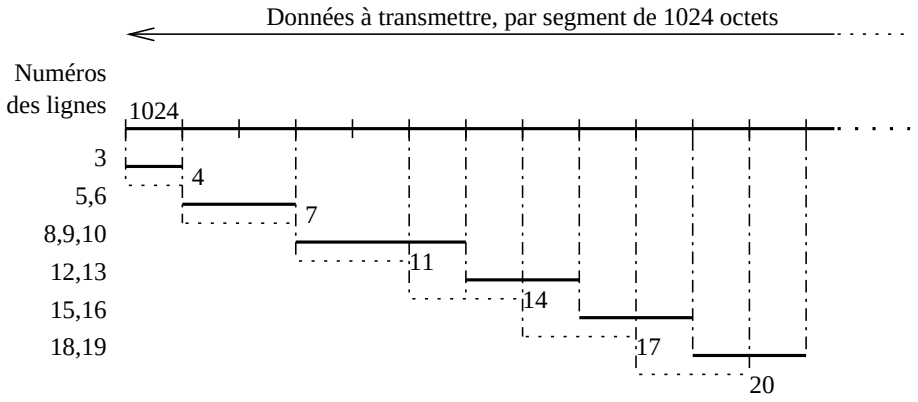


figure VI.09 — Exemple de fenêtre glissante

6 Bibliographie

RFC 793 “ Transmission Control Protocol. ” J. Postel. Sep-01-1981. (Format : TXT=177957 bytes) (Status : STANDARD)

RFC 1025 “ TCP and IP bake off. ” J. Postel. Sep-01-1987. (Format : TXT=11648 bytes) (Status : UNKNOWN)

RFC 1700 “ ASSIGNED NUMBERS. ” J. Reynolds,J. Postel. October 1994. (Format : TXT=458860 bytes) (Obsoletes RFC1340) (Also STD0002) (Status : STANDARD)

Sans oublier :

- W. Richard Stevens - TCP/IP Illustrated, Volume 1 - The protocols - Addison-Wesley
- Douglas Comer - Internetworking with TCP/IP - Principles, protocols, and architecture - Prentice–Hall
- McKusick – Bostic – Karels – Quaterman — “ The Design and implementation of the 4.4 BSD Operating System ” (chapitre 13) — Addison–Wesley — 1996

Deuxième partie

Protocoles applicatifs

Chapitre VII

Serveur de noms - DNS

1 Généralités sur le serveur de noms

S'il est obligatoire d'attribuer au moins une adresse IP à une pile ARPA pour pouvoir l'interconnecter en réseau avec d'autres piles de même nature, en revanche, lui attribuer un nom symbolique n'est absolument pas nécessaire au bon fonctionnement de ses trois couches basses.

Ce nommage symbolique est simplement beaucoup plus naturel pour les humains que la manipulation des adresses IP, même sous forme décimale pointée (adresses IP page 31). Il n'intervient donc qu'au niveau applicatif, ainsi la majeure partie des applications réseaux font usage de noms symboliques avec, de manière sous-jacente, une référence implicite à leur(s) correspondant(s) numérique(s).

Ce chapitre explore les grandes lignes du fonctionnement de ce que l'on nomme le " serveur de noms ", lien entre cette symbolique et l'adressage IP qui lui est associé .

Pour terminer cette introduction, il n'est pas innocent de préciser que le serveur de noms est en général le premier service mis en route sur un réseau, tout simplement parceque beaucoup de services le requièrent pour accepter de fonctionner (le courrier électronique en est un exemple majeur). C'est la raison pour laquelle l'usage d'adresses IP sous la forme décimale pointée reste de mise lors de la configuration des éléments de commutation et de routage[1].

1.1 Bref historique

Au début de l'histoire de l'Internet, la correspondance entre le nom (les noms s'il y a des synonymes ou " alias ") et l'adresse (il peut y en avoir plusieurs associées à un seul nom) d'une machine est placée dans le fichier `/etc/hosts`, présent sur toutes les machines unix dotées d'une pile Arpa.

Ci-après le fichier en question, prélevé (et tronqué partiellement) sur une machine FreeBSD[2] à jour. On y remarque qu'il ne contient plus que l'adresse

[1] Éviter un blocage dû à l'interrogation des serveurs de noms

[2] www.freebsd.org

de “ loopback ” en ipv6 et ipv4.

```
# $FreeBSD: src/etc/hosts,v 1.11.2.4 2003/02/06 20:36:58 dbaker Exp $
#
# Host Database
#
# This file should contain the addresses and aliases for local hosts that
# share this file.  Replace 'my.domain' below with the domainname of your
# machine.
#
# In the presence of the domain name service or NIS, this file may
# not be consulted at all; see /etc/host.conf for the resolution order.
#
#
::1                     localhost localhost.my.domain
127.0.0.1               localhost localhost.my.domain
```

Au début des années 1980 c'est le NIC[3] qui gère la mise à jour continuelle de cette table (`HOSTS.TXT`), avec les inconvénients suivants :

- Absence de structure claire dans le nommage d'où de nombreux conflits entre les noms des stations. Par exemple entre les dieux de la mythologie grecque, les planètes du système solaire, les héros historiques ou de bandes dessinées...
- Centralisation des mises à jour, ce qui entraine :
  1. Une lourdeur du processus de mise à jour : il faut passer par un intermédiaire pour attribuer un nom à ses machines.
  2. Un trafic réseau (`ftp`) en forte croissance (N^2 si N est le nombre de machines dans cette table) et qui devient rapidement ingérable au vu des bandes passantes de l'époque (quelques kilo bits par seconde), et surtout jamais à jour compte tenu des changements continuels.

D'après Douglas E. Comer, au milieu des années 1980 (1986) la liste officielle des hôtes de l'Internet contient 3100 noms et 6500 alias !

La forte croissance du nombre des machines, a rendu obsolète cette approche.

1.2 Système hiérarchisé de nommage

L'espace de noms, préalablement non structuré, est désormais réorganisé de manière hiérarchique, sous forme d'un arbre (et non d'un graphe).

Cette organisation entraine une hiérarchisation des noms de machines et des autorités qui ont le pouvoir de les nommer, de les maintenir.

Chaque nœud de l'arbre porte un nom, la racine n'en a pas. Les machines, feuilles de l'arbre, sont nommées à l'aide du chemin parcouru de la feuille (machine) à la racine (non nommée).

[3] “ Network Information Center ” (http ://www.internic.net/)

Le séparateur entre chaque embranchement, ou nœud, est le point décimal. Voici un exemple de nom de machine :

`www.sio.ecp.fr`

Derrière ce nom il faut imaginer un point (.) qui est omis la plupart du temps car il est implicite[4]. La lecture s'effectue naturellement de gauche à droite, par contre la hiérarchie de noms s'observe de droite à gauche.

Domaine & zone

Le réseau peut être considéré comme une hiérarchie de domaines. L'espace des noms y est organisé en tenant compte des limites administratives ou organisationnelles. Chaque nœud, appelé un domaine, est baptisé par une chaîne de caractères et le nom de ce domaine est la concaténation de toutes les étiquettes de nœuds lues depuis la racine, donc de droite à gauche. Par exemple :

fr	Domaine `fr`
ecp.fr	Domaine `ecp.fr` sous domaine du `fr`
sio.ecp.fr	Domaine `sio.ecp.fr` sous domaine de `ecp.fr`

Par construction, tout nœud est un domaine, même s'il est terminal, c'est à dire n'a pas de sous domaine. Un sous domaine est un domaine à part entière et, exceptée la racine, tout domaine est un sous domaine d'un autre.

Bien que le serveur de noms, " **D**omain **N**ame **S**erver " fasse référence explicitement au concept de domaine, pour bien comprendre la configuration d'un tel service il faut également comprendre la notion de " zone ".

Une zone est un point de délégation dans l'arbre DNS, autrement dit une zone concerne un sous arbre du DNS dont l'administration lui est propre. Ce sous arbre peut comprendre plusieurs niveaux, c'est à dire plusieurs sous domaines. Une zone peut être confondue avec le domaine dans les cas les plus simples.

Dans les exemples ci-dessus, on peut parler de zone `sio.ecp.fr` puisque celle-ci est gérée de manière autonome par rapport à la zone `ecp.fr`.

Le serveur de noms est concerné par les " zones ". Ses fichiers de configuration[5] précisent la portée de la zone et non du domaine.

Chaque zone doit avoir un serveur principal (" primary server ") qui détient ses informations d'un fichier configuré manuellement ou semi manuellement (DNS dynamique). Plusieurs serveurs secondaires (" secondary server ") recoivent une copie de la zone via le réseau et pour assurer la continuité du service (par la redondance des serveurs).

Le fait d'administrer une zone est le résultat d'une déléguation de pouvoir de l'administrateur de la zone parente et se concrétise par la responsabilité de configurer et d'entretenir le champ `SOA` (" start of authority ", page 120) de la-dite zone.

[4] Sauf justement dans les fichiers de configuration du serveur de noms, voir plus loin

[5] named.boot vs named.conf

Hiérarchie des domaines

Cette organisation du nommage pallie aux inconvénients de la première méthode :

- Le NIC gère le plus haut niveau de la hiérarchie, appelé aussi celui des " top levels domains " (TLD).
- Les instances régionales du NIC gèrent les domaines qui leur sont dévolus. Par exemple le " NIC France " [6] gère le contenu de la zone `.fr`. Le nommage sur deux lettres des pays est issu de la norme ISO 3166[7].
- Chaque administrateur de domaine (universités, entreprises, associations, entités administratives,...) est en charge de son domaine et des sous domaines qu'il crée. Sa responsabilité est nomminative vis à vis du NIC. On dit aussi qu'il a l'autorité sur son domaine (" authoritative for the domain ")[8]

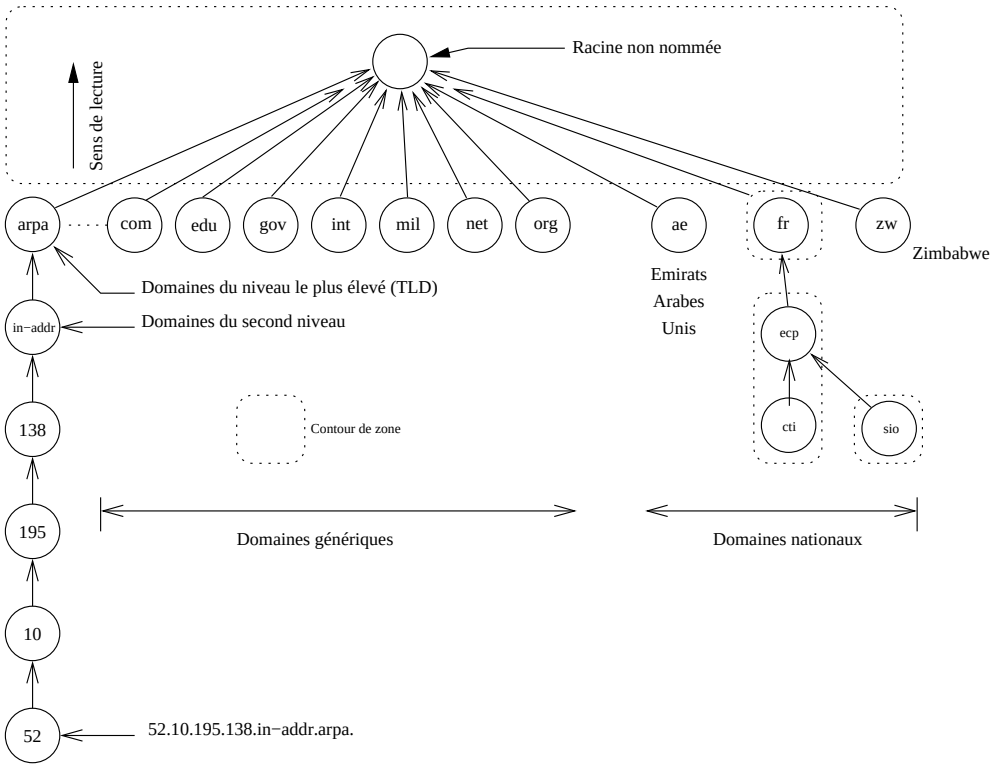


figure VII.01

[6] http ://www.nic.fr/

[7] On peut les voir en détail à cette adresse `http ://www.nw.com/zone/iso-country-codes`

[8] Une base de données sur les administrateurs de DNS est entretenue par les NICs, c'est la base " whois ", interrogeable par l'utilitaire du même nom. Consulter le site `http ://www.ripe.net/` pour plus d'informations, également " `man whois` " sur une machine unix

– Les éventuels conflits de nommage sont à la charge des administrateurs de domaine. Du fait de la hiérarchisation, des machines de même nom peuvent se trouver dans des domaines différents sans que cela pose le moindre problème.
  Le nom “ `www` ” est de loin le plus employé [9], pourtant il n'y a aucune confusion possible entres ces machines, comme par exemple entres les machines `www.ecp.fr` et `www.sio.ecp.fr`.
– Chaque domaine entretient une base de données sur le nommage de ses machines. Celle-ci est mise à disposition de tous les utilisateurs du réseau.
  Chaque site raccordé de manière permanente procède de cette manière, ainsi il n'y a pas une base de données pour l'Internet mais un ensemble structuré de bases de données reliées entres elles et formant une gigantesque **base de données distribuée**.

2 Fonctionnement du DNS

2.1 Convention de nommage

La RFC 1034 précise que les noms de machines sont développés un peu comme les noms de fichiers d'un système hiérarchisé (Unix,...).

– Le “ . ” est le séparateur
– Chaque nœud ne peut faire que 63 caractères au maximum ; “ le bon usage ” les limite à 12 caractères et commençant par une lettre.
– Les majuscules et minuscules sont indifférenciées.
– Les chiffres [0-9] et le tiret peuvent être utilisés, le souligné (_) est un abus d'usage.
– Le point “ . ” et le blanc “ ” sont proscrits.
– Les chaînes de caractères comme “ NIC ” ou d'autres acronymes bien connus sont à éviter absolument, même encadrées par d'autres caractères.
– Les noms complets ne doivent pas faire plus de 255 caractères de long.

Il y a des noms “ relatifs ” et des noms “ absolus ”, comme des chemins dans un système de fichiers. L'usage du “ . ” en fin de nom, qui indique un nommage absolu[10], est réservé à certains outils comme `ping` ou `traceroute` et aux fichiers de configuration du serveur de noms. En règle générale il n'est pas utile de l'employer.

[9] 901 961 instances en janvier 2003 contre 1 203 856 instances en janvier 2002, selon le site du “ Network Wizards Internet Domain Suvey ” (www.nw.com), “ Top 100 Host Names ”

[10] FQDN, comme “ Fully Qualified Domain Name ”

“ Completion ”

Sur un même réseau logique on a coutume de ne pas utiliser le nom complet des machines auxquelles on s'adresse couramment et pourtant ça fonctionne !

La raison est que le “ resolver ”, partie du système qui est en charge de résoudre les problèmes de conversion entre un nom de machine et son adresse IP, utilise un mécanisme de complétion (“ domain completion ”) pour compléter le nom de machine simplifié, jusqu'à trouver un nom plus complet que le serveur de noms saura reconnaître dans sa base de données.

Le “ resolver ” connait par hypothèse le nom de domaine courant (lu dans le fichier de configuration `/etc/resolv.conf`).
Exemple d'un tel fichier :

```
domain      sio.ecp.fr
search      sio.ecp.fr., ecp.fr.
nameserver  138.195.52.68
nameserver  138.195.52.69
nameserver  138.195.52.132
```

Plus généralement ce nom de domaine se présente sous forme $d_1.d_2...d_n$. Ainsi, en présence d'un nom symbolique x, le “ resolver ” teste pour chaque i, $i \in \{1, 2, \ldots, n\}$ l'existence de $x.d_i.d_{i+1}...d_n$ et s'arrête si celle-ci est reconnue. Dans le cas contraire la machine en question n'est pas atteignable.
Exemple, avec le domaine ci-dessus :

a) machine = `www` (requête)
`www.sio.ecp.fr` $\Longrightarrow$ Succès !

b) machine = `www.sio` (requête)
`www.sio.sio.ecp.fr` $\Longrightarrow$ Échec !
`www.sio.ecp.fr` $\Longrightarrow$ Succès !

2.2 Le “ Resolver ”

Le “ resolver ” désigne un ensemble de fonctions[11] placées dans la bibliothèque standard (`gethostbyname` vu en cours de programmation invoque les fonctions du “ resolver ”) qui font l'interface entre les applications et les serveurs de noms. Par construction les fonctions du “ resolver ” sont compilées avec l'application qui les utilise (physiquement dans la `libc`, donc accessibles par défaut).

Le “ resolver ” applique la stratégie locale de recherche, définie par l'administrateur de la machine, pour résoudre les requêtes de résolution de noms. Pour cela il s'appuie sur son fichier de configuration `/etc/resolv.conf` et sur la stratégie locale (voir paragraphe suivant) d'emploi des possibilités (serveur de noms, fichier `/etc/hosts`, NIS,...).

[11] `res_query, res_search, res_mkquery, res_send, res_init, dn_comp, dn_expand` - Faire “ man resolver ” sur une machine unix

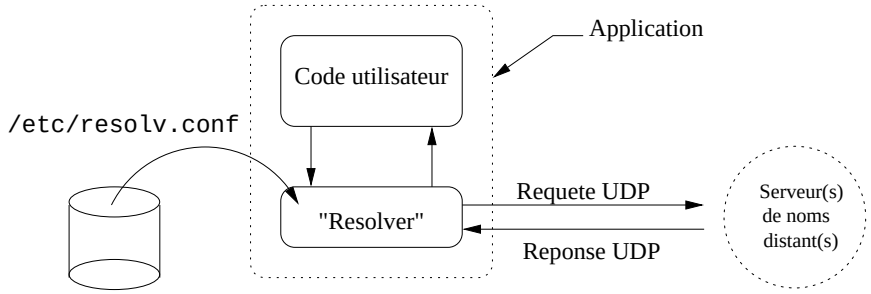


figure VII.02

Le fichier `/etc/resolv.conf` précise au moins le domaine local assorti de directives optionnelles.

2.3 Stratégie de fonctionnement

La figure *VII.03* illustre le fait que chaque serveur de noms a la maîtrise de ses données mais doit interroger ses voisins dès qu'une requête concerne une zone sur laquelle il n'a pas l'autorité de nommage.

Ainsi, un hôte du domaine " R2 " qui veut résoudre une adresse du domaine " R1 " doit nécessairement passer par un serveur intermédiaire pour obtenir l'information. Cette démarche s'appuie sur plusieurs stratégies possibles, que nous examinons dans les paragraphes suivants.

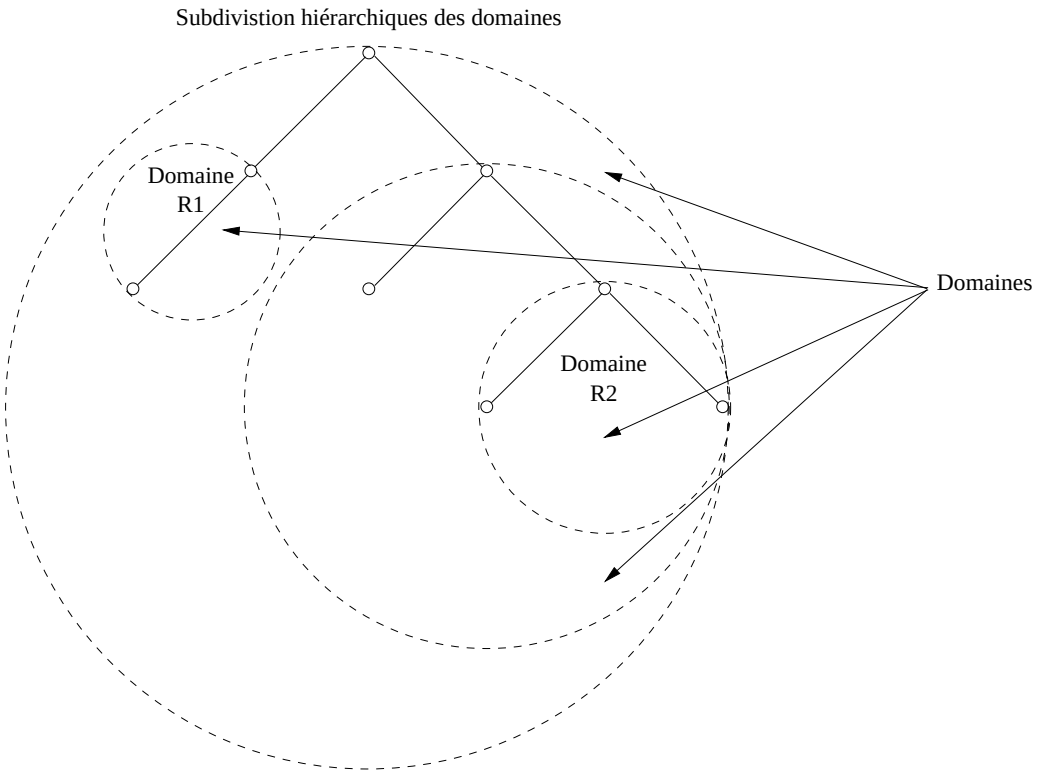


figure VII.03

Interrogation locale

La figure ci-dessous illustre la recherche d'un nom dans le domaine local.

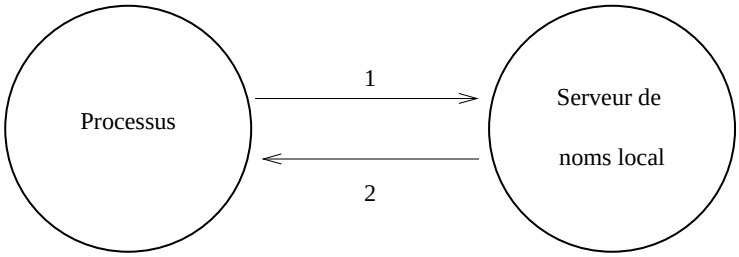


figure VII.04

Un processus (" browser " `http` par exemple) recherche l'adresse d'un nom de serveur. Sur les machines Unix cela se traduit par l'appel à la fonction `gethostbyname`. Cette fonction est systématiquement présente dans la bibliothèque standard (`libc`) et est donc accessible potentiellement à tout exécutable lors d'une compilation.

La fonction `gethostbyname` fait systématiquement appel au " resolver " déjà cité. C'est donc toujours en passant par ce mécanisme que les processus accèdent à l'espace de noms. Le " resolver " utilise une stratégie générale à la machine (donc qui a été choisie par son administrateur) pour résoudre de telles requêtes :

1. Interrogation du serveur de noms (DNS) si présent
2. Utilisation des services type " YP " (NIS) si configurés
3. Utilisation du fichier `/etc/hosts`

Cette stratégie est paramètrable en fonction du constructeur. Le `nsswitch` sous HP-UX[12] ou Solaris[13] permet de passer de l'un à l'autre en cas d'indisponibilité, le fichier `/etc/host.conf` sous FreeBSD effectue un travail assez proche.

Enfin, quelle que soit l'architecture logicielle le " resolver " est configuré à l'aide du fichier `/etc/resolv.conf`.
Sur la *figure VII.04* :

1. Le processus demande l'adresse IP d'un serveur. Le " resolver " envoie la demande au serveur local.
2. Le serveur local reçoit la demande, parcequ'il a l'autorité sur le domaine demandé (le sien), il répond directement au " resolver ".

Interrogation distante

1. Un processus demande l'adresse IP d'une machine. Le " resolver " envoie sa requête au serveur local.

[12]Unix de " Hewlett-Packard "
[13]Unix de " Sun Microsystem "

2. Le serveur local reçoit la requête et dans ce deuxième cas il ne peut pas répondre directement car la machine n'est pas dans sa zone d'autorité. Pour lever l'indétermination il interroge alors un serveur racine pour avoir l'adresse d'un serveur qui a l'autorité sur la zone demandée par le processus.
3. Le serveur racine renvoie l'adresse d'un serveur qui a officiellement l'autorité sur la zone
4. Le serveur local interroge ce nouveau serveur distant.
5. Le serveur distant renvoie l'information demandée au serveur local.
6. Le serveur local retourne la réponse au " resolver "

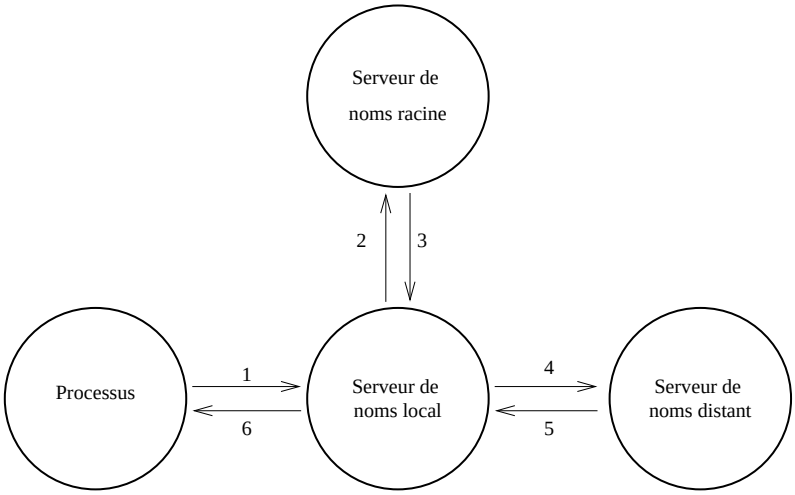


figure VII.05

Remarques importantes :

- Un mécanisme de cache accélère le processus ci-dessus : Si un processus redemande la même machine distante on se retrouve dans le cas d'une interrogation " locale ", du moins pendant la durée de validité des données (cf page 123).
- Si un processus demande une machine du même domaine que la précédente (mais pas du même nom ! :), les étapes 2 et 3 deviennent inutiles et le serveur local interroge alors directement le serveur distant. La durée de vie de l'adresse du serveur distant est elle aussi assortie d'une date limite d'utilisation.
- Dans le cas général les serveurs racines ne voient pas plus de 1 ou deux niveaux en dessous. Ainsi, si un processus demande A.B.C.D.net :
  1. Le serveur racine donne l'adresse d'un serveur pour D
  2. Le serveur pour D donnera peut être l'adresse d'un serveur pour C et ainsi jouera le rôle de serveur racine de l'étape précédente.

Interrogation par " procuration "

Le processus de recherche décrit au paragraphe précédent ne convient pas dans tous les cas, notamment vis à vis des deux critères suivants :

1. Sécurité d'un domaine
2. Conservation de la bande passante

1. La *figure VII.05* montre le serveur local qui interroge directement les serveurs distants, cette démarche pose des problèmes de sécurité dans le cas d'un domaine au sein duquel seuls un ou deux serveurs sont autorisés.

Par exemple le serveur de noms du domaine **sio.ecp.fr** n'interroge pas directement le serveur racine, il passe par le serveur officiel qui est `piston.ecp.fr` (138.195.33.3).

2. Le trafic destiné au serveur de noms peut consommer une partie non négligeable de la bande passante, c'est pourquoi il peut être stratégique de concentrer les demandes vers un seul serveur régional et donc de bénéficier au maximum de l'effet de cache décrit précédement.

2.4 Hiérarchie de serveurs

Si tous les serveurs de noms traitent de données d'un format identique, leur position dans l'arborescence leur confère un statut qui se nomme :

serveur racine (" root name server ") Un serveur ayant autorité sur la racine de l'espace de nommage. Actuellement il y a 13 serveurs de ce type, nommés [A-M].ROOT-SERVERS.NET[14]

serveur primaire (" primary server ") Un serveur de noms qui a l'autorité pour un ou plusieurs domaines (est détenteur d'autant de `SOA` – Voir page 120). Il lit ses données dans un fichier stocké sur disque dur, à son démarrage.

L'administrateur du (des) domaine(s) met à jour les informations des domaines concernés depuis cette machine.

serveur secondaire (" secondary server ") Dans le cas d'une panne ou d'un engorgement du serveur primaire, les serveurs secondaires reçoivent en prévision une copie de la base de données.

- Stratégiquement il est préférable de les placer en dehors du domaine, sur le réseau d'un autre FAI. Il peut y avoir autant de serveurs secondaires que souhaité, de l'ordre de trois ou quatre est souvent recontré.
- Au démarrage ils reçoivent les informations du serveur primaire, ou ils les lisent sur leur disque dur s'ils ont eu le temps de les y stocker au précédent arrêt du serveur, et si elles sont encore valides.

Par exemple, le serveur PISTON.ECP.FR a comme serveurs secondaires NS2.NIC.FR, SOLEIL.UVSQ.FR, NADIA.IRCAM.FR.

2.5 Conversion d'adresses IP en noms

On dit aussi questions inverses (" inverse queries ").

[14] fichier `named.root`, par exemple dans le répertoire /etc/namedb

Cette possibilité est indiquée comme optionnelle dans la RFC 1034 mais est néanmoins bien commode voire même fréquement requise pour le client réseau de services comme le courrier électronique ou même les serveurs de fichiers anonymes (`ftp`). Si une machine est enregistrée dans le TLD `in-addr.arpa`, cela signifie cette idresse ip est administrée, ce qui ne prouve rien quant aux bonnes intentions de son utilisateur mais est un gage de " bonne conduite " au niveau du réseau.

Il faut ajouter que le bon usage sur les réseaux est de prévoir une entrée dans la zone reverse pour toutes les machines utiles et utilisées d'un réseau accessible de l'Internet. Le contraire provoque bien souvent la grogne (à juste titre) des administrateurs.

Il faut reconsidérer la *figure VII.01*. À gauche de la figure on distingue un domaine un peu particulier " in-addr.arpa ". Toutes les adresses sont exprimées dans le " top level domain " :

in-addr.arpa

Du fait de la lecture inverse de l'arbre, les adresses IP sont exprimées en " mirroir " de la réalité. Par exemple pour la classe B de l'ecp :

195.138.in-addr.arpa (Classe B 138.195)

Le fonctionnement par délégation est calqué sur celui utilisé pour les noms symboliques (c'est la justification de son insertion dans la *figure VII.01*). Ainsi on peut obtenir quels sont les serveurs qui ont autorité sur le **195.138.in-addr.arpa** en questionnant d'abord les serveurs du TLD **in-addr.arpa** puis ceux pour la zone **138.in-addr.arpa**,...

Chaque administrateur de zone peut aussi être en charge de l'administration des " zones reverses ", portion du domaine " arpa ", des classes d'adresses dont il dispose pour nommer ses machines, s'il en reçoit la délégation (celle-ci est indépendante des autres noms de domaines).

Il faut noter que la notion de sous réseau (cf page 36) n'est pas applicable au domaine " in-addr.arpa ", ce qui signifie que les adresses selon les contours naturels des octets.

Ainsi, pour les clients de fournisseurs d'accès n'ayant comme adresses IP officielles que celles délimitées par un masque de sous réseau large seulement que de quelques unités (< 254), la gestion de la zone reverse reste du domaine du prestataire et non de son client.

2.6 Conclusion

Qu'est-ce qu'un DNS ?

Un serveur de noms repose sur trois constituants :

1. Un espace de noms et une base de données qui associe de manière structurée des noms à des adresses IP.
2. Des serveurs de noms, qui sont compétents pour répondre sur une ou plusieurs zones.
3. Des " resolver " capables d'interroger les serveurs avec une stratégie définie par l'administrateur du système.

TCP ou UDP ?

Le port 53 " bien connu " pour le serveur de noms est prévu pour fonctionner avec les deux protocoles.

- Normalement la majeure partie du trafic se fait avec UDP, mais si la taille d'une réponse dépasse les 512 octets, un drapeau de l'en-tête du protocole l'indique au client qui reformule sans question en utilisant TCP.
- Quand un serveur secondaire démarre son activité, il effectue une connexion TCP vers le serveur primaire pour obtenir sa copie de la base de données. En général, toutes les trois heures (c'est une valeur courante) il effectue cette démarche.

3 Sécurisation du DNS

Le serveur de noms est la clef de voûte des réseaux, et c'est en même temps un de ses talons d'Achille parceque les programmes que nous employons quotidiennement utilisent sans discernement l'information acquise du réseau. En effet, qu'est-ce qui vous assure que le site web de votre banque sur lequel vous venez de taper votre mot de passe est bien le vrai site officiel de cet établissement ? L'apparence de la page de garde ?

Typiquement il y a deux situations de vulnérabilité :

1. Dialogue serveur à serveur, notament lors de transferts de zones.
2. Interrogation d'un serveur par un resolveur

Pour faire confiance en ce que vous dit le serveur de noms interrogé il faut d'une part que vous soyez certains d'interroger la bonne machine et d'autre part que celle-ci soit détentrice d'une information incontestable.

C'est une chaîne de confiance, comme toujours en sécurité, qui remonte par construction du fonctionnement du serveur de noms interrogé par votre application (comme nous l'avons examiné dans les paragraphes qui précèdent) jusqu'aux serveurs racines.

La version ISC (consultez le paragraphe 6) du programme BIND utilise deux stratégies différentes, selon les cas ci-dessus. Dans le premier cas il s'agit d'un mécanisme nommé TSIG/TKEY, dans le deuxième DNSSEC.

TSIG/TKEY utilisent une clef symétrique, donc partagée par les deux serveurs (cette clef leur est connue par des mécanismes différents). DNSSEC utilise un mécanisme basé sur le principe d'un échange de clefs publiques.

3.1 TSIG/TKEY pour sécuriser les transferts

L'usage d'une clef symétrique indique qu'il s'agit d'un secret partagé entre 2 serveurs. La même clef sert au chiffrement et au déchiffrement des données. Le bon usage veut que l'on dédie une clef à un certain type de transaction (par exemple le transfert d'une zone) entre deux serveurs donnés. Cette manière de procéder se traduit donc rapidement par un grand nombre de clefs à gérer ce qui interdit un déploiement généralisé sur l'Internet.

Pour éviter de trop longs temps de chiffrement, ce ne sont pas les données à transférer qui sont chiffrées (ce ne sont pas des données confidentielles), mais leur signature avec un algorithme de type MD5[15]

Le serveur qui reçoit un tel paquet signé, calcule la signature du paquet avec le même algorithme (MD5), déchiffre la signature jointe avec la clef secrète partagée et compare les deux signatures. Le résultat de cette comparaison dit si les données sont valides ou non.

L'intérêt de ces transferts signés est que les serveurs secondaires sont certains de mettre à jour leur zones avec des données qui proviennent bien du détenteur du SOA et qui sont absolument semblables à ce qui a été émis.

TSIG

TSIG comme " Transaction SIGnature " est la méthode décrite dans la RFC 2845 et basée sur l'usage d'une clef symétrique. La génération de cette clef peut être manuelle ou automatisée avec le programme " `dnssec-keygen` ".

La propagation de cette clef est manuelle (`scp`...Éviter absolument l'usage de tout protocole diffusant un mot de passe en clair sur le réseau), donc mise en place au coup par coup.

TSIG sert également à la mise à jour dynamique (" dynamic update "), la connaissance de la clef par le client sert à la fois à l'authentifier et à signer les données [16].

TKEY

TKEY, décrit dans la RFC 2930, rend les mêmes services que TSIG tout en évitant le transport du secret (TSIG). Cette caractéristique est basée sur le calcul la clef symétrique automatiquement avec l'algorithme de Diffie-Hellman plutôt que par un échange " manuel "[17].

[15] Ne pas hésiter à faire un `man md5` sur une machine Unix pour en savoir plus !

[16] cf le programme `nsupdate` et la RFC 2136

[17] On peut trouver une explication de cet algorithme sur ce site : `http ://www.rsasecurity.com/rsalabs/faq/3-6-1.html`

Par contre, cet algorithme à base du tandem clef publique – clef privée suppose l'ajout d'un champ `KEY` dans les fichiers de configuration du serveur. Comme d'ailleurs le mécanisme suivant...

3.2 DNSSEC pour sécuriser les interrogations

DNSSEC décrit dans la RFC 2535 permet :

1. La distribution d'une clef publique (champ `KEY`)
2. La certification de l'origine des données
3. L'authentification des transactions (transferts, requêtes)

Mis en place, le DNSSEC permet de construire une chaîne de confiance, depuis le " top level " jusqu'au serveur interrogé par votre station de travail.

4 Mise à jour dynamique

La mise à jour dynamique de serveur de noms (RFC 2136) est une fonctionnalité assez récente sur le réseau, elle permet comme son nom l'indique de mettre à jour la base de donnée répartie.

Aussi bien au niveau du réseau local qu'à l'échelle de l'Internet il s'agit le plus souvent de faire correspondre un nom de machine fixe avec une adresse ip changeante. C'est typiquement le cas d'un tout petit site qui a enregistré son domaine chez un vendeur quelconque et qui au gré des changements d'adresse ip (attribuée dynamiquement par exemple avec DHCP[18]) par son fournisseur d'accès, met à jour le serveur de noms pour être toujours accessible.

Avec comme mot clef " dyndns ", les moteurs de recherche indiquent l'existence de sites commerciaux ou à caractère associatif, qui proposent cette fonctionnalité.

[18] Cf `http ://www.isc.org/products/DHCP/`

5 Format des “ Resource Record ”

Comme pour toute base de données, le serveur de nom a un format pour ses champs, ou “ Resource Record ”, `RR` dans la suite de ce texte, défini à l'origine dans la RFC 1035.

En pratique toutes les distributions (commerciales ou libres) du serveur de noms conservent ce format de base de données, la mise en œuvre du serveur seule change (fichier de configuration du `daemon`).

Un serveur de noms a autorité (responsabilité du `SOA`) sur une ou plusieurs zones, celles-ci sont repérées dans ses fichiers de configuration (`named.conf` ou `named.boot` selon les versions). S'il est serveur primaire d'une ou plusieurs zones, le contenu de ces zones est inscrit dans des fichiers ASCII ; leur syntaxe est succintement décrite dans le paragraphe suivant.

S'il est serveur secondaire, le fichier de configuration indique au server de quelle(s) zone(s) il est secondaire (il peut être secondaire d'un grand nombre de zones) et donc où (adresse IP) il devra aller chercher cette information. Cette action se traduit par ce que l'on nomme un “ transfert de zone ”. Ce transfert est effectué automatiquement à la fréquence prévue par l'administrateur du champ `SOA` et donc connue dès le premier transfert. En cas de changement sur le serveur principal, celui-ci avertit (“ Notify ”) ses serveurs secondaires de la nécessité de recharger la zone pour être à jour.

Le propos de ce qui suit n'est pas de se substituer à une documentation nombreuse et bien faite sur le sujet, mais d'apporter quelques éléments fondamentaux pour en aider la lecture.

Le constituant de base d'un serveur de noms est une paire de fichiers ASCII contenant les enregistrements, les “ Resource Record ”.

Ceux-ci sont en général écrits sur une seule ligne de texte (sauf pour le champ `SOA` qui s'étale sur plusieurs lignes. Le marqueur de fin de ligne (CR+LF) est aussi celui de la fin de l'enregistrement. Le contenu général d'un tel enregistrement a la forme suivante (les accolades indique des données optionnelles) :

```
{name}   {ttl}   addr-class   Record Type   Record Specific data
```

Cinq enregistrements, ou “ Resource Record ”, ou en RR, sont absolument fondamentaux pour faire fonctionner un serveur de noms : `SOA`, `NS`, `A`, `MX` et `PTR`.

5.1 RR de type SOA

```
$(ORIGIN) sio.ecp.fr.
name   {ttl}   addr-class   SOA      Origin       Person in charge
@              IN           SOA      sio.ecp.fr.  hostmaster.sio.ecp.fr. (
                                          2003110301   ; Serial
                                          10800        ; Refresh (3h)
                                          3600         ; Retry (1H)
                                          3600000      ; Expire (5w6d16h)
                                          86400 )      ; Minimum ttl (1D)
```

SOA est l'acronyme de **“ Start Of Authority ”** et désigne le début obligé et unique d'une zone. Il doit figurer dans chaque fichier db.domain et db.adresse Le nom de cette zone est ici repéré par le caractère @ qui signifie la zone courante, repérée par la ligne au dessus “ $(ORIGIN) sio.ecp.fr. ”. La ligne aurait également pu s'écrire :

```
sio.ecp.fr.    IN          SOA      sio.ecp.fr.  hostmaster.sio.ecp.fr. (
                                                 ...
```

Un problème concernant cette zone devra être signalé par e-mail à hostmaster@sio.ecp.fr (notez le “.” qui s'est transformé en “@”).

Les paramètres de ce SOA sont décrits sur plusieurs lignes, regroupées entres parenthèses. Le caractère “ ;” marque le début d'un commentaire, qui s'arrête à la fin de ligne.
Les points en fin de noms sont nécessaires.

Le numéro de série doit changer à chaque mise à jour de la zone (sur le serveur principal). Le “ Refresh ” indique la fréquence, en secondes, à laquelle les serveurs secondaires doivent consulter le primaire pour éventuellement lancer un transfert de zone (si le numéro de série est plus grand). Le “ Retry ” indique combien de secondes un serveur secondaire doit attendre un transfert avant de le déclarer impossible. Le “ Expire ” indique le nombre de secondes maximum pendant lesquelles un serveur secondaire peut se servir des données du primaire en cas d'échec du transfert. “ Minimum ttl ” est le nombre de secondes par défaut pour le champ TTL si celui-ci est omis dans les RR.

5.2 RR de type NS

Il faut ajouter une ligne de ce type (**“ Name Server ”**) pour chaque serveur de noms pour le domaine. Notez bien que rien dans la syntaxe ne permet de distinguer le serveur principal de ses secondaires.
Dans le fichier db.domaine :

```
          {name}   {ttl}   addr-class   NS   Name servers name
                                IN      NS   ns-master.sio.ecp.fr.
                                IN      NS   ns-slave1.sio.ecp.fr.
                                IN      NS   ns-slave2.sio.ecp.fr.
```

Dans le fichier qui renseigne la zone “ reverse ”, par exemple `db.adresse`, on trouvera :

```
52.195.138.in-addr.arpa. IN   NS   ns-master.sio.ecp.fr.
52.195.138.in-addr.arpa. IN   NS   ns-slave1.sio.ecp.fr.
52.195.138.in-addr.arpa. IN   NS   ns-slave2.sio.ecp.fr.
```

5.3 RR de type A

Le RR de type `A`, ou encore **“ Address record ”** attribue une ou plusieurs adresses à un nom, c'est donc celui qui est potentiellement le plus fréquement utilisé. Il doit y avoir un RR de type `A` pour chaque adresse d'une machine.

```
{name}    {ttl}   addr-class   A   address
gw-sio            IN           A   138.195.52.2
                  IN           A   138.195.52.33
                  IN           A   138.195.52.65
```

5.4 RR de type PTR

Le RR de type `PTR`, ou encore **“ PoinTeR record ”** permet de spécifier les adresses pour la résolution inverse, donc dans le domaine spécial `IN-ADDR.ARPA`. Notez le “.” en fin de nom qui interdit la complétion (il s'agit bien du nom FQDN).

```
name   {ttl}   addr-class   PTR   real name
2               IN          PTR   gw-sio.sio.ecp.fr.
33              IN          PTR   gw-sio.sio.ecp.fr.
65              IN          PTR   gw-sio.sio.ecp.fr.
```

5.5 RR de type MX

Le RR de type `MX`, ou encore **“ Mail eXchanger ”** concerne les relations entre le serveur de noms et le courrier électronique. Nous examinerons son fonctionnement ultérieurement dans le chapitre sur le courrier électronique (cf page 139).

```
sio.ecp.fr.            IN      MX      10      smtp.ecp.fr.
sio.ecp.fr.            IN      MX      20      mailhost.laissus.fr.
```

5.6 RR de type CNAME

Le RR de type CNAME, ou encore **“ canonical name ”** permet de distinguer le nom officiel d'une machine de ses surnoms.

```
www.sio.ecp.fr.          IN      CNAME          srv.laissus.fr.
```

Dans l'exemple ci-dessus, la machine www.sio.ecp.fr est un surnom de la machine srv.laissus.fr. On notera que les domaines n'ont rien à voir mais que ça fonctionne quand même parfaitement.

Cette possibilité est très employée pour constituer des machines virtuelles, comme nous le verrons au chapitre X.

5.7 Autres RR...

Il existe d'autres RR, entres autres HINFO , TXT, WKS et KEY, non traités dans cette présentation parcequ'ils n'apportent rien à la compréhension du fonctionnement du serveur de noms. Le lecteur est fortement incité à se reporter au “ Name Server Operations Guide ” pour plus d'informations.

6 BIND de l'ISC

L'Internet Software Consortium[19] est une organisation non commerciale qui développe et favorise l'emploi de l'outil " Open Source " comme BIND (acronyme de " Berkeley Internet Name Domain ").

Cette version libre du serveur de nom est la plus employée sur les machines Unix du réseau, ce qui justifie que l'on s'y intéresse. Elle fournit une version du `daemon` " named " et un " resolver " intégré dans la `libc`. On peut aisément installer ce logiciel sur à peu près toutes les implémentations d'unix connues (cf le fichier INSTALL du répertoire `src`).

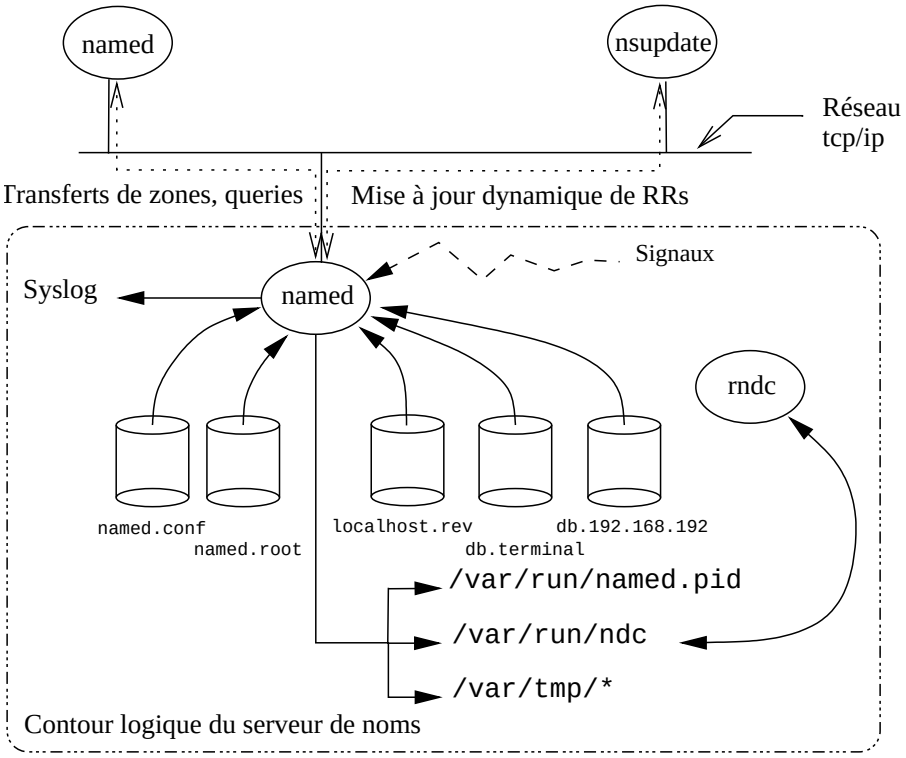


figure VII.06

6.1 Architecture du daemon " named "

La *figure VII.06* montre le schéma général de l'organisation logicielle du daemon " named ".

Au démarrage celui-ci lit sa configuration dans un fichier qui peut se nommer `named.boot` ou `named.conf` selon que l'on est en version 4.9.11, 8.3.6 ou 9.2.3 et les suivantes du logiciel[20].

`named.conf` C'est le fichier de configuration lu au démarrage. Sa structure dépend de la version du logicielle, heureusement dans les deux cas la sémantique reste proche !

[19] `http ://www.isc.org/`

[20] taper " `named -v` " pour les discerner entre-elles

`named.root` Ce fichier contient la liste des serveurs de la racine, leur nom et adresse IP.

`localhost.rev` Ce fichier contient la base de donnée du " localhost ". Personne n'a la responsabilité du réseau 127, donc chacun doit le gérer pour lui-même. L'absence de ce fichier n'empêche pas le serveur de fonctionner, mais ne lui permet pas de résoudre 127.0.0.xx (où xx est le numéro de la machine courante, souvent 1).

`db.terminal` Exemple de fichier de base de données pour le domaine factice `terminal.fr` qui est utile durant les travaux pratiques. Ce fichier permet la convertion des noms en adresses IP.

`db.192.168.192` Ce fichier contient la base de données de la zone " reverse " pour le domaine `terminal.fr`, c'est à dire le fichier qui permet au logiciel de convertir les adresses IP en noms.

rndc ou " **name server control utility** " est comme son nom l'indique un outil d'administration du programme `named` lui-même. C'est une alternative à l'usage direct des signaux. Le canal de communication entre les deux programmes est une socket unix `AF_UNIX` vs `AF_LOCAL` (cf cours de programmation page 201).

Un certain nombre de signaux modifient le comportement du serveur, ils seront examinés en travaux pratiques, tout comme les fichiers lus ou écrits dans les répertoires `/var/run` et `/var/tmp/`.

Enfin la flèche vers `syslog` signifie que `named` utilise ce service pour laisser une trace de son activité (cf cours sur l'architecture des serveurs).

Enfin, le BOG, c'est à dire le " Bind Operations Guide ", détaille le contenu des champs de la base de données.

7 Bibliographie

Quand on " sait déjà ", la page de man de " `named` " suffit à vérifier un point obscur ! Sinon il existe une documentation très fournie sur le sujet, avec notamment :

- Kein J. Dunlap & Michael J. Karels — " Name Server Operations Guide " — Ce document est accessible sur le serveur de l'Internet Software Consortium [21]
- By the Nominum BIND Development Team — " BIND 9 Administrator Reference Manual " — Version 9.1.3 [22]
- Douglas E. Comer — " Internetworking with TCP/IP – Volume I" (chapter 18) — Prentice All — 1988
- Paul Albitz & Cricket Liu — " DNS and BIND " — O'Reilly & Associates, Inc. — 1992

[21] On trouve le BOG dans la distribution de " bind " à cette adresse : `http ://www.isc.org/products/BIND/`

[22] on trouve également la dernière version de ce document sur le site de l'ISC

– “ Installing and Administering ARPA Services ” — Hewlett Packard — 1991
– W. Richard Stevens — “ TCP/IP Illustrated Volume I ” (chapter 14) — Prentice All — 1994

Et pour en savoir encore plus...

RFC 1034 “ Domain names - concepts and facilities ”. P.V. Mockapetris. Nov-01-1987. (Format : TXT=129180 bytes) (Obsoletes RFC0973, RFC0882, RFC0883) (Obsoleted by RFC1065, RFC2065) (Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC2065, RFC2181) (Status : STANDARD)

RFC 1035 “ Domain names - implementation and specification ”. P.V. Mockapetris. Nov-01-1987. (Format : TXT=125626 bytes) (Obsoletes RFC0973, RFC0882, RFC0883) (Obsoleted by RFC2065) (Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC1995, RFC1996, RFC2065, RFC2181, RFC2136, RFC2137) (Status : STANDARD)

RFC 1101 “ DNS encoding of network names and other types ”. P.V. Mockapetris. Apr-01-1989. (Format : TXT=28677 bytes) (Updates RFC1034, RFC1035) (Status : UNKNOWN)

RFC 1123 “ Requirements for Internet hosts - application and support ”. R.T. Braden. Oct-01-1989. (Format : TXT=245503 bytes) (Updates RFC0822) (Updated by RFC2181) (Status : STANDARD)

RFC 1713 “ Tools for DNS debugging ”. A. Romao. November 1994. (Format : TXT=33500 bytes) (Also FYI0027) (Status : INFORMATIONAL)

RFC 2136 “ Dynamic Updates in the Domain Name System (DNS UPDATE) ”. P. Vixie, Ed., S. Thomson, Y. Rekhter, J. Bound. April 1997. (Format : TXT=56354 bytes) (Updates RFC1035) (Updated by RFC3007) (Status : PROPOSED STANDARD)

RFC 2535 “ Domain Name System Security Extensions ”. D. Eastlake 3rd. March 1999. (Format : TXT=110958 bytes) (Obsoletes RFC2065) (Updates RFC2181, RFC1035, RFC1034) (Updated by RFC2931, RFC3007, RFC3008, RFC3090, RFC3226, RFC3445) (Status : PROPOSED STANDARD)

RFC 2845 “ Secret Key Transaction Authentication for DNS (TSIG) ” P. Vixie, O. Gudmundsson, B. Wellington. May 2000. (Format : TXT=32272 bytes) (Updates RFC1035) (Status : PROPOSED STANDARD)

RFC 2930 “ Secret Key Establishment for DNS (TKEY RR) ” D. Eastlake 3rd. September 2000. (Format : TXT=34894 bytes) (Status : PROPOSED STANDARD)

Chapitre VIII

Courrier électronique

1 Généralités sur le courrier électronique

Le courrier électronique, ou "mail" est l'un des deux services les plus populaires sur le réseau, avec le web.

C'est aussi l'un des plus vieux services du réseau, bien avant que le réseau existe sous la forme que l'on pratique aujourd'hui[1]. La préface de la [**RFC 822**], document fondamental parmi les documents fondamentaux pour ce chapitre, laisse supposer l'existence de nombreux formats d'échanges électroniques sur l'Arpanet, et ce avant 1977.

Sa popularité repose sur sa grande souplesse et rapidité d'emploi. Il permet aussi bien les échanges professionnels que les échanges privés ; son mode d'adressage donne la possibilité d'envoyer un courrier à une personne comme à une liste de personnes ou encore à un automate capable de rediffuser vers un groupe ("mailing-list").

De nombreux outils développés, à l'origine essentiellement sur le système Unix, autour de ce concept ouvrent un vaste champs de possibilités aux utilisateurs de tous les systèmes d'exploitation, comme la ventilation des courriers par thème, le renvoi automatique, le répondeur (pendant les absences), l'accès à sa boite aux lettres depuis des endroits différents, la réception de fax,...La liste ne peut pas être exhaustive !

C'est souvent pour avoir l'usage du courrier électronique que les entreprises accèdent à l'Internet. Les autres services viennent après.

1.1 Métaphore du courrier postal

Un courrier postal (ou de surface, "s-mail") a besoin de l'adresse du destinataire, de l'adresse de l'émetteur (pour la réponse), d'un timbre et d'une enveloppe.

Une fois dans la boîte aux lettres, l'enveloppe est routée de la poste locale vers la poste la plus proche du destinataire, pour être finalement délivrée par un facteur.

[1] Un historique intéressant `http ://www.fnet.fr/history/`

Pour un courrier électronique c'est la même chose !

Il existe de très nombreux outils pour lire/écrire un mail, des outils pour jouer le rôle du bureau de poste et/ou du facteur. Sous Unix le facteur est le système lui-même, le bureau de poste un programme nommé " sendmail[2] ". Il existe deux autres alternatives non abordées dans ce document, à savoir le programme " qmail[3] " ou encore le programme " postfix[4] ".

1.2 Adresse électronique

Tous les courriers électroniques ont un destinataire précisé par son adresse électronique, ou " E-mail[5] ". Celle-ci précise le nom du destinataire et le site où il reçoit son courrier électronique.

Le nom du destinaire est une chaîne de caractères. Traditionnellement et pour des raisons techniques, sur le système Unix, le login de l'utilisateur peut être également le nom de sa boite aux lettres. Cette possibilité est de moins en moins vraie à mesure que d'autres systèmes avec d'autres logiques de fonctionnement font leur apparition sur le réseau (notamment la lecture du mail via un interface `html` ou encore lorsque le mail est géré par une base de données).

Par exemple, il est assez fréquent de voir employer le nom complet (prénom et nom de famille) pour désigner l'interlocuteur distant. La conversion ultime entre cette convention et la boîte aux lettres de l'utilisateur est l'affaire du "bureau de poste le plus proche", c'est à dire le programme "sendmail" pour ce document (voir plus loin au paragraphe 4).

Le caractère " @ " (lire " at ") sépare l'identificateur du destinataire de la destination.

La destination est peut être vide (il s'agit alors d'un destinataire sur la machine courante, ou d'un "alias" que le sendmail local sait traiter), être un nom de machine du domaine local, le nom d'un autre domaine ou d'une machine sur un autre domaine.
Les adresses suivantes ont un format valide :

user1 Destinataire local.

user2@nom_de_machine Destinataire sur une machine du domaine courant (rappellez-vous, il existe un mécanisme de complétion dans le " resolver " page 110 !).

user3@nom_de_machine.domaine Destinataire sur une machine particulière d'un domaine particulier (non forcément local).

user4@domaine Destinataire sur un domaine particulier (même remarque que ci-dessus).

[2] Version 8.12.9 en avril 2003 — `http ://www.sendmail.org/`
[3] `http ://www.qmail.org/`
[4] `http ://postfix.eu.org/start.html`
[5] Terme francisé en " mèl ", ou " couriel " pour les documents administratifs... `;-)`

On devine aisément que le fonctionnement du courrier électronique sur une machine distante est fortement liée au bon fonctionnement du serveur de noms (chapitre VII).

Qui plus est, lorsque seul un nom de domaine est précisé à droite du caractère " @ ", une information manque apparemment quant à la machine susceptible de recevoir le mail.

Le lecteur en quête de plus de précisions trouvera une description exhaustive de la syntaxe d'une adresse au paragraphe 6 de la [**RFC 822**].

2 Format d'un "E-mail" - RFC 822

Les octets qui composent un courrier électronique obéissent à une structure bien définie par la [**RFC 822**] de David H. Crocker : un en-tête et un corps de message, séparés par une ligne blanche (deux CRLF[6] qui se suivent).

Le contenu de l'en-tête dans son intégralité n'est pas toujours spontanément montré par les outils qui nous permettent de lire et d'envoyer du courrier électronique. Une option est toujours accessible pour ce faire, comme " **h** " sous `mutt` [7].

Une partie de l'en-tête est générée automatiquement par le programme qui se charge du transfert (le paragraphe suivant nous dira qu'il s'agit d'un MTA), une autre est ajoutée par le programme qui permet de composer le mail, le MUA, une autre enfin est tapée par l'utilisateur lui-même.

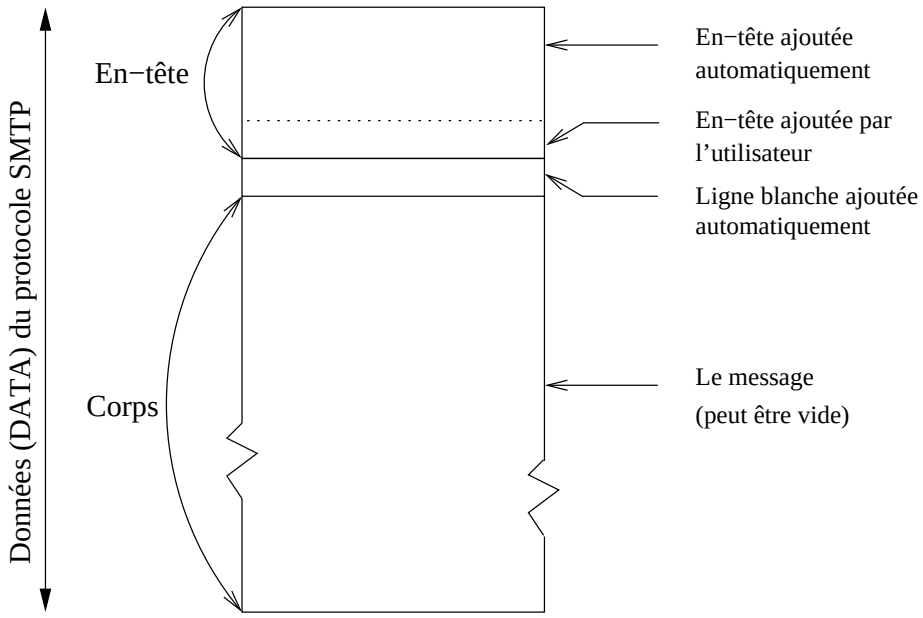


figure VIII.01

[6] Il s'agit respectivement des caractères 13 et 10 de la table ASCII - cf " `man ascii` "

[7] `http ://www.mutt.org/` — le MUA, voir paragraphe suivant, préféré de l'auteur

L'en-tête est constitué de lignes construites sur le modèle :

identificateur : [valeur] CRLF

L'identificateur ne peut pas contenir le caractère " : " parcequ'il sert de séparateur avec la partie droite. Il est constitué de caractères ASCII codés sur 7 bits et imprimables (c'est à dire $\in [33, 126]$), excepté l'espace. " Valeur " est optionnelle. L'usage des majuscules ou des minuscules est indifférencié.

L'ordre d'apparition de ces champs est quelconque. Seule l'organisation de la *figure VIII.01* doit être globalement respectée. Le lecteur soucieux d'une description exhaustive de ce en quoi peut être constitué un en-tête pourra se repporter au paragraphe 4.1 de la RFC (SYNTAX).

L'en-tête d'un mail donne des informations de nature variée sur le message lui-même. On peut les apparenter, pour certaines d'entres elles, aux informations qu'on trouve sur les enveloppes des courriers postaux.

Type d'information	Noms des champs
Cheminement du courrier	`Received, Return-Path`
Origine du courrier	`From, Sender, Reply-To`
Destinataire(s) du courrier	`To, Cc, Bcc`
Identification du courrier	`Message-Id, In-Reply-To`
Renvoi	`Resent-`
Autres	`Subject, Date`
Champs étendus	`X-?`

Nous aurons l'occasion d'examiner des en-têtes, notamment lors des travaux pratiques qui suivront ce cours.

3 Protocole SMTP - RFC 821

Le protocole SMTP, ou " Simple Mail Transfert Protocol " a comme objet le transport du courrier électronique de manière fiable et efficace. Il est défini dans la [**RFC 821**] de Jonathan B. Postel.

Indépendant par sa conception d'un quelconque sous-système de transport, il est principalement aujourd'hui encapsulé dans des paquets `TCP` à destination du port 25 (cf le fichier /etc/services). Dans un passé pas si lointain l'accès réseau de beaucoup de sites se résumait au courrier électronique encapsulé dans des trames du protocole `UUCP`, donc sur liaison série via modem !

3.1 Protocole `SMTP`

SMTP est un protocole ASCII (7 bits, "human readable"). La partie cliente de la transaction se connecte sur le port 25 du serveur et envoie des commandes auxquelles le serveur répond par des codes numériques qui indiquent le statut de la prise en compte de la commande.

C'est pourquoi il est aisé de se connecter sur un MTA avec un simple `telnet`[8] :

```
$ telnet localhost 25
Connected to localhost.
Escape character is '^]'.
220 host.mondomain.fr ESMTP Sendmail 8.12.6; Mon, 20 Jan 2003 15:34:45 +0100 (CET)
NOOP
250 2.0.0 OK
QUIT
221 2.0.0 host.mondomain.fr closing connection
Connection closed by foreign host.
```

Dans cet exemple le MTA est le programme `Sendmail` [9], qui répond à la connexion par un code 220 pour dire que le service est opérationnel ("service ready"), suivi du nom de la machine, de la bannière du programme, de la version de sa configuration, et de sa date courante.

Puis l'utilisateur a tapé la commande `NOOP` qui n'a d'autre effet que de forcer le serveur à répondre et renvoyer un code (250) pour dire que tout va bien.

Enfin L'utilisateur a tapé `QUIT` pour finir proprement la transaction. La réponse du serveur est un code 221 pour signifier la fin canonique de la connexion.

[8] Attention toutefois de ne pas abuser de cette pratique car de nombreuses attaques de sites ont démarré par le passé à l'aide un détournement de `sendmail`. Les administrateurs réseaux sont donc attentifs au trafic sur le port 25 ; il est préférable de réserver ce genre de tests uniquement sur son propre site.

[9] `http ://www.sendmail.org/`

Dans un deuxième essai nous utilisons l'option `-v` du programme `mail`, pour visualiser les échanges entre le MUA (machine `athome.mondomain.fr`) et le MTA local (machine `mailhub.mondomain.fr`).
Essayons :

```
$ sendmail -v user@mondomain.fr
Subject: test
Ca passe ?

.          <<<--- A taper pour marquer la fin du mail dans ce mode.
EOT
user@mondomain.fr... Connecting to mailhub.mondomain.fr. via relay...
220 mailhub.mondomain.fr HP Sendmail (1.37.109.4/user-2.1) ready at Mon,
26 Jan 98 14:08:57 +0100
>>> HELO athome.mondomain.fr
250 mailhub.mondomain.fr Hello athome.mondomain.fr, pleased to meet you
>>> MAIL From:<user@mondomain.fr>
250 <user@mondomain.fr>... Sender ok
>>> RCPT To:<user@mondomain.fr>
250 <user@mondomain.fr>... Recipient ok
>>> DATA
354 Enter mail, end with "." on a line by itself
>>> .
250 Ok
user@mondomain.fr... Sent (Ok)
Closing connection to mailhub.mondomain.fr.
>>> QUIT
221 mailhub.mondomain.fr closing connection
```

Et le courrier reçu, lu aussi avec `mail` :

```
Message 208/208 User Lambda                    Jan 26, 98 02:09:06 pm +0100

From user@mondomain.fr Mon Jan 26 14:09:08 1998
Received: from mailhub.mondomain.fr by athome.mondomain.fr with SMTP (8.8.7/8.8.7/f
la.2.1) id OAA27655; Mon, 26 Jan 1998 14:09:08 +0100 (CET)
Received: from athome.mondomain.fr by mailhub.mondomain.fr with SMTP (1.37.109.4/fl
a-2.1) id AA06996; Mon, 26 Jan 98 14:08:57 +0100
From: User Lambda <user@mondomain.fr>
Received: by athome.mondomain.fr
Date: Mon, 26 Jan 1998 14:09:06 +0100 (CET)
Message-Id: <199801261309.OAA27653@athome.mondomain.fr>
To: user@mondomain.fr
Subject: test

Ca passe ?
```

Manifestement, ça passe ! `:)`

Il est également intéressant d'observer à ce niveau que les caractères du courrier ont été considérablement enrichis par un en-tête volumineux (relativement).

En effet, chaque nœud traversé (MTA), ajoute un champ "`Received`" permettant après coup de suivre le trajet du courrier. Les autres champs

comme `Date :`, `Subject :` ou `Message-Id :` sont ajoutés dans l'en-tête par le MUA de l'écrivain du message.

Cette partie de l'en-tête ajoutée par le MUA d'origine est souvent destinée à piloter le comportement du MUA du destinataire du message plus que pour être lue. Cette attitude s'est généralisée au point de devenir assez compliquée et être formalisée dans un ensemble de règles baptisées MIME comme "Multipurpose Internet Mail Extensions" ([**RFC 2184**]).

La fonction la plus répandue et la plus simple de MIME est d'autoriser l'usage des caractères accentués (codage sur 8 bits ou plus) à l'intérieur du corps du message (l'en-tête SMTP reste codée sur 7 bits). L'utilisateur voit alors apparaître des lignes supplémentaires comme celles-ci :

```
X-Mime-Autoconverted: from 8bit to quoted-printable by bidule.domain id SAA23150
X-MIME-Autoconverted: from quoted-printable to 8bit by mamachine.ici id QAA29283
```

Le "`quoted-printable`" est une forme possible du codage des caractères accentués, définie dans la [**RFC 822**]. Le plus souvent on trouve des lignes comme celles-ci :

```
Mime-version: 1.0
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfert-Encoding: quoted-printable
Content-ID: Content-ID: <Pine.FBSD.3.14.1592654.19971998X.domain>
Content-Description: pouet.c
```

D'autres formes de MIME peuvent conduire à l'exécution d'un programme extérieur au MUA, ce qui constitue une dangereuse faille potentielle dans la sécurité des réseaux, donc à éviter.

3.2 Principales commandes de SMTP

Expérimentalement nous avons découvert quelques uns des mots réservés du protocole : `HELO, MAIL, RCPT, DATA, QUIT`. Une implémentation minimale de `SMTP` en comprend deux autres en plus : `RSET` et `NOOP`. C'est donc un protocole assez simple, du moins dans sa version de base.

Les codes de retour sont organisés sur trois chiffres, le premier chiffre donne le sens général de la réponse, très succintement ce qui débute par 1,2 ou 3 a une signification positive, 4 ou 5 signifie une erreur. Une information plus détaillée se trouve à l'annexe E de la RFC.

Les cinq commandes découvertes précédement s'utilisent toujours dans cet ordre. Examinons succintement leur usage :

Commande HELO

Synopsis : HELO <espace> <domaine> <CRLF>

Cette commande est utilisée pour identifier l'émetteur du mail. L'argument qui suit, **domain** est le nom de la machine ou du domaine d'où provient la connexion.

En réponse le serveur envoie une bannière dans laquelle il s'identifie et donne la date courante. Cette information est optionnelle, ce qui compte c'est le code de retour pour confirmer l'aptitude au travail du serveur !

Commande MAIL

Synopsis : MAIL <espace> FROM : <chemin inverse> <CRLF>

Cette commande débute un transfert de mail. En argument sont transmis (**chemin inverse**) l'adresse e-mail de l'émetteur et la liste des machines qui ont relayé le mail courant. La première machine est la plus récente. Cette information est utilisée pour renvoyer, s'il y a lieu, une notification de problème à l'émetteur du mail.
Par exemple :

MAIL FROM :<@mailhub.ici :@mail-host.labas :Lambda@mondomain.fr>

Commande RCPT

Synopsis : RCPT <espace> TO : <destinataire> <CRLF>

Cette commande est la deuxième étape dans la procédure d'envoi d'un mail. Il y a une commande de ce type par destinataire du courrier (“ recipient ”).
Par exemple :

RCPT TO :<Lambda@mondomain.fr>

Il est intéressant de noter que les arguments de cette commande et ceux de la précédente (MAIL) forment **l'enveloppe du mail** (expéditeur et destinataire).

Commande DATA

Synopsis : DATA <CRLF>

Après réception de la commande, le serveur lit les lignes de texte en provenance du client jusqu'à rencontrer la séquence **<CRLF>.<CRLF>** qui marque la fin du message. Il faut remarquer que celui-ci comprend l'intégralité de la *figure VIII.01.*

Commande QUIT

Synopsis : QUIT <CRLF>

Marque la fin de la session et entraine la clôture de la connexion.

3.3 Propagation du courrier électronique

SMTP est défini comme un protocole de transfert, donc un moyen pour router et délivrer le message à son (ses) destinataire final (finaux).

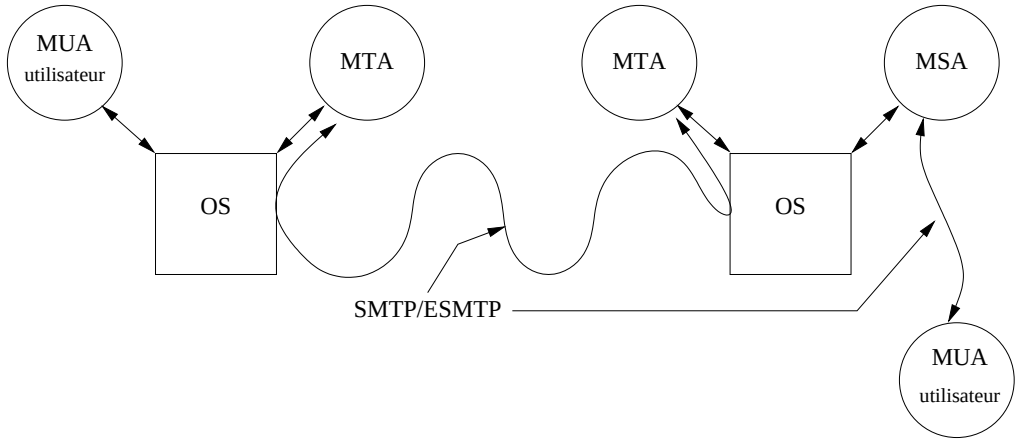


figure VIII.02

MTA " Mail Transfert Agent ", c'est le programme que prend en charge le transfert du courrier. Sous Unix c'est un `daemon`. Par exemple : `MMDF, Smail, Zmailer, sendmail, postfix, qmail...`

Les MTAs écoutent le réseau sur le port 25 et dialoguent entre-eux avec le protocole SMTP (ESMTP[10]).

MSA " Mail Submission Agent ", c'est une " nouveauté " définie par la [**RFC 2476**] et qui joue le rôle d'interface entre le MUA et le MTA.

L'objet du MSA est de séparer les fonctions de transfert du courrier et d'acceptation de nouveaux courriers émis depuis les MUA. Cette séparation des tâches améliore deux aspects :

La sécurité Les nouveaux mails sont soumis à un `daemon` qui ne n'exécutent pas avec les droits du `root`[11].

La conformité aux standards Les messages proviennent de MUAs qui ne respectent pas forcément tous les prérequis de formulation des en-têtes.

Le rôle du MSA est de vérifier et de complèter ces en-têtes avant de soumettre les mails au MTA pour le routage.

MUA "Mail User Agent" ou encore "mailer", c'est le programme qui permet de lire et écrire le corps du courrier et de paramétrer quelques éléments de l'en-tête, principalement l'adresse du destinaire, et le sujet du message.

[10]Extented SMTP
[11]SUID bit

Il existe un très grand nombre de MUAs sous Unix, il est courant de rencontrer `mail, mailx, elm, pine, mutt, mh, eudora, kmail, sylpheed...` Il y en a pour tous les goûts !

OS "Operating System", le système d'exploitation sur lequel fonctionnent ces programmes.

La *figure 2* illustre la possibilité la plus simple d'échange entre deux MTA : la connexion directe. Cela signifie que le MTA de la station émettrice contacte le MTA de la station réceptrice et lui délivre directement le message.

La vie " réelle " est plus compliquée car elle tient compte de l'organisation hiérarchique des réseaux et surtout de la sécurité qui est un aspect devenu important sur l'Internet. Cela se traduit par un emploi généralisé de machines relais ou " mailhub ".

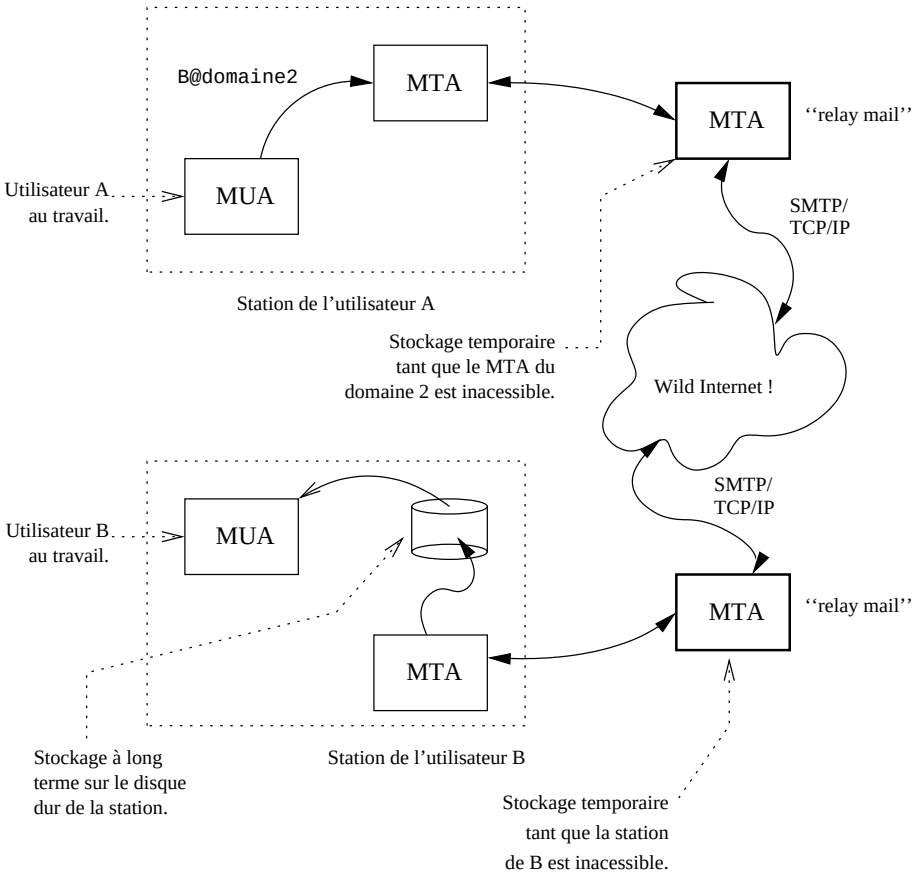


figure VIII.03

Une telle machine concentre tous les courriers électroniques d'un site, vers l'extérieur et inversement. Elle a des avantages, notamment :

– Avoir une politique de sécurité concentrée sur une machine plutôt que sur toutes les stations du réseau : le routeur filtrant n'autorise les accès extérieurs sur le port smtp que sur cette machine.

- Avoir une politique centralisée de filtrage des contenus indésirables (virus) et émetteurs suspects (spam).
- Limiter le nombre de configurations compliquées de `sendmail` à un petit nombre de machines. Les stations des utilisateurs peuvent se contenter d'une configuration standard plus facile à distribuer et à adapter automatiquement.
- Permettre de masquer plus facilement les machines internes du réseau vis à vis de l'extérieur. En clair, les courriers auront l'air de provenir de cette machine plutôt que de la station d'un utilisateur sur le réseau interne.
  L'adresse de l'émetteur aura la forme :
  `user@domaine`
  au lieu de :
  `user@machine.domaine`.
- Permettre le stockage intermédiaire du courrier en attente d'une délivrance : les stations des utilisateurs ne sont pas toujours en fonctionnement.

Cette architecture est théorique, en pratique il peut y avoir une hiérarchie de " relay mail " plus compliquée. Par exemple une machine distincte suivant que le courrier entre ou sort du site, une arborescence de machines relais quand l'entreprise est elle-même répartie sur plusieurs sites géographiques et ne possède qu'une liaison vers l'extérieur,...

3.4 Courriers indésirables - Le spam

Le spam est l'aspect très désagréable du courrier électronique.

Par " spam " on désigne ces innombrables courriers, le plus souvent à caractère commercial, qui envahissent nos boites aux lettres électroniques. Certaines estimations tablent sur 30% de spam dans le trafic mail.

Deux questions se posent, comment le caractériser et surtout comment l'éviter.

Caractéristiques du spam

- Un contenu commercial (ou financier d'une manière ou une autre)
- Une importante liste de destinataires
- En-tête du message truquée
- Un grand nombre d'exemplaires du même message envoyé dans un court laps de temps
- Utilisation de l'adresse d'un destinataire sans son consentement explicite
- Usage d'un site " open mail relay " pour l'émission
- Champs `To` et `From` de l'en-tête invalides

Pour mémoire, un site " open mail relay " autorise un `RCPT` qui ne désigne pas un destinataire local. Le site sert alors en quelque sorte de tremplin pour

le mail, avec un effet de dissimulation du site émetteur véritable. Les versions modernes des MTA interdisent cette possibilité.

Éviter le spam

C'est une question ouverte...

S'il est évident que l'œil humain reconnait un tel courrier de manière quasi infaillible, il n'en est pas de même pour la machine.

Néanmoins des outils existent[12] qui établissent un pré-filtrage souvent très efficace mais laissant toujours un pourcentage de mails qui franchissent les protections.

Ces outils sont toujours basés sur un filtrage. Plus celui-ci est complexe, plus le temps de traitement à consacrer pour 1 courrier électronique est important. L'idéal étant évidement de reconnaître la qualité d'un e-mail en très peu de cycles machines...

Il y a deux parties à examiner dans un courrier, l'en-tête et le corps (*figure VIII.01*).

Tout ce qui est vérifiable dans l'en-tête et qui concerne le réseau se résoud bien (détection des sites " open mail relay " ou liste des utilisateurs indésirables). Cette approche n'est pas suffisante car rien n'oblige dans le protocole à ce que les champs `From :` et `To :` de l'en-tête MIME soient en concordance avec les éléments du protocole SMTP. On parle alors de " fake mail ".
L'exemple qui suit illustre cette faille ô combien navrante :

```
telnet mailhost.mondomain.fr 25
Connected to mailhost.mondomain.fr.
Escape character is '^]'.
220 mailhost.mondomain.fr ESMTP Sendmail 8.11.0/fla-2001-04-10; ...
HELO UnSiteQuelconque.com
250 mailhost.mondomain.fr Hello UnSiteQuelconque.com, pleased to meet you
MAIL From:<>
250 2.1.0 <>... Sender ok
RCPT To:<tartempion@mondomain.fr>
250 2.1.5 <tartempion@mondomain.fr>... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
From:<NePasRepondre@XXX.com>
To:<TuPeuxToujoursEssayer@YYY.com>

Unsollicited Bulk Email (UBE) or
Unsollicited Commercial Email (UCE).
.
```

Et le mail sera délivré dans la boite aux lettres de l'utilisateur " tartempion " avec des champs `From :` et `To :` complètement inexploitables.

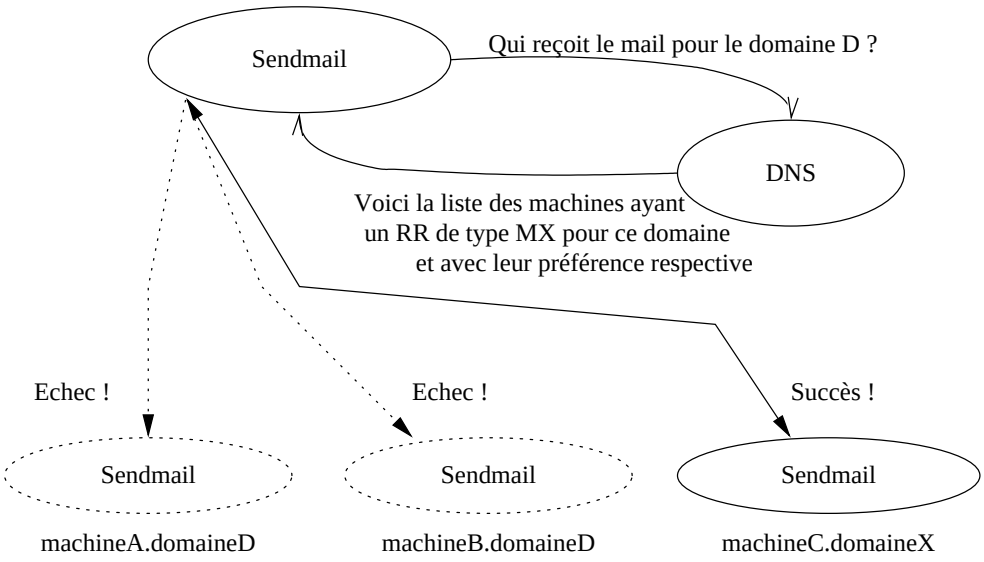
[12] Cf documentation complémentaire distribuée en cours

Le filtrage sur le contenu du message est une autre possibilité mais qui est beaucoup plus consommatrice de ressouces cpu, proportionnellement à sa taille et à la complexité des parties (pièces jointes) qui peuvent le composer. Les outils les plus performants restent à écrire... L'outil ultime sera celui qui comprendra le sens du texte et rejettera le courrier parcequ'indésirable.

4 Exemple de MTA - “Sendmail” et son environnement

4.1 Relations avec le DNS

Comme nous l'avons évoqué au paragraphe **??**, la relation entre le MTA et le DNS est étroite. Sendmail a besoin du serveur de noms pour les opérations suivantes :



MTA pouvant récupérer le courrier pour le domaine D

figure VIII.04

1. Transformer le nom de la machine distante en adresse IP
2. Canoniser le nom de la machine locale.
3. Canoniser le nom de la machine qui se connecte
4. Déterminer quelles sont les machines susceptibles de recevoir du courrier pour le domaine à atteindre.

Le quatrième point est le plus crucial. Si le DNS du domaine à atteindre (une adresse est toujours mise sous la forme “nom@domain”) ne désigne pas de machine capable de recevoir le courrier, le mail ne passera jamais pour ce domaine.

Le champ RR (“Resource Record”) correspondant est du type MX (“Mail Exchanger”). Il spécifie une liste d'hôtes pondérés par des préférences, à qui on peut envoyer du courrier. La pondération indique l'ordre à suivre pour les tentatives de connexions : il faut commencer par la valeur la plus basse. Si cette liste est explorée de bout en bout sans succès il y a échec de la transmission du courrier.

S'il y a échec de la reémission, le mail est conservé un certain temps, puis est finalement rejeté s'il y a persistance de l'échec. Le résultat est matérialisé dans un fichier nommé `dead.letter`

Figure 4, Le contenu du champ RR renvoyé par le DNS pourrait avoir la constitution suivante :

```
;
;[name]     [ttl][class] MX preference      mail exchanger
;
domaineD           IN     MX         10      machineA.domaineD
                   IN     MX         20      machineB.domaineD
                   IN     MX         30      machineC.domaineX
```

Il est important de remarquer qu'une machine baptisée “MX” par le DNS n'est pas forcément localisée dans le domaine pour lequel elle reçoit le courrier, c'est même souvent le cas pour les machines “ relay ”. C'est le cas de la troisième ligne, `machineC.domaineX`

4.2 Relations avec l'OS

`Sendmail` a de multiples relations avec le système d'exploitation, comme la *figure 5* tente de le résumer :

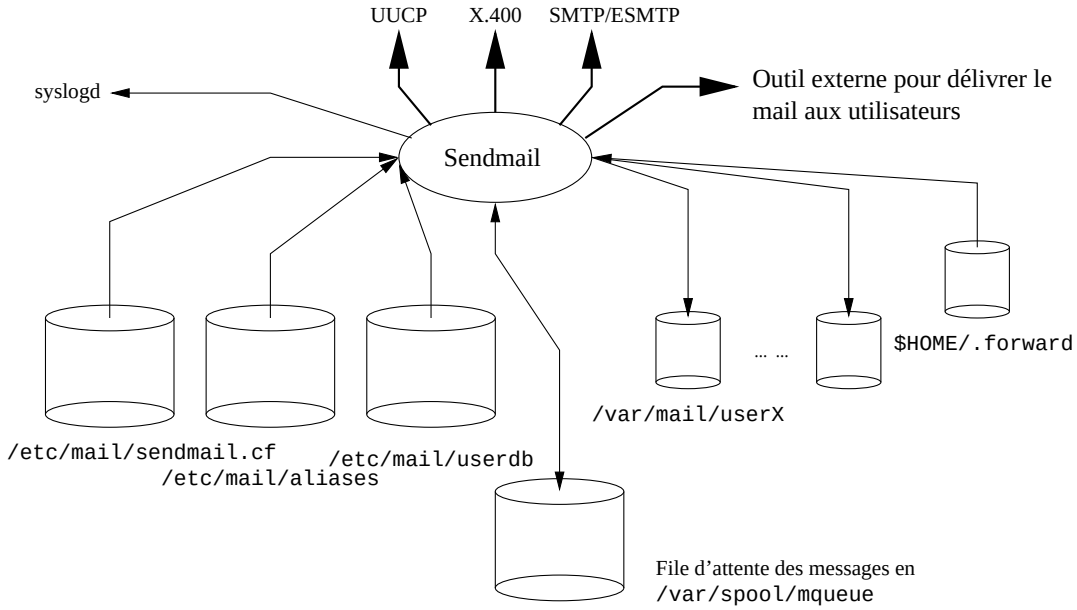


figure VIII.05

– L'activité opérationnelle de `sendmail` est consignée à l'aide de **syslog**[13], ce qui explique la présence de la ligne suivante trouvée dans le fichier `/etc/syslog.conf` :

```
mail.info                                    /var/log/maillog
```

– `UUCP`, `X.400`, `SMTP` sont autant de moyens de propager le courrier. Ces supports peuvent cohabiter au sein d'une même configuration ; autant de "Mailer" sélectionnés en fonction de l'adresse du destinataire (cf `sendmail.cf`).
– `/etc/sendmail.cf` est le fichier de configuration de `sendmail`. Tout système Unix est livré avec une configuration standard qu'il faut adapter à la situation locale ; c'est en général à cet instant que les administrateurs systèmes deviennent pensifs…(voir pourquoi au paragraphe 5).
– `/etc/aliases` est une base de synonymes. Quand `sendmail` reçoit un courrier, il tente de reconnaître le nom du destinataire dans cette base et si c'est le cas de lui appliquer la transformation prescrite.
Un certain nombre d'alias sont requis par la [**RFC 1123**], d'autres sont conseillés par la [**RFC 2142**]. Un court extrait du-dit fichier :

[13] Chapitre III du cours de programmation — Éléments de serveurs

```
# Basic system aliases -- these MUST be present
MAILER-DAEMON: postmaster
postmaster: root
# Well-known aliases -- these should be filled in!
root:         user
info:         root
marketing:    root
sales:        root
support:      root
www:          webmaster
webmaster:    root
...
```

Cela signifie par exemple que chaque fois q'un courrier est envoyé au "postmaster" de ce site, sendmail transforme "postmaster" en "root", puis "root" en "user". Si cette dernière chaîne ne fait pas l'objet d'une autre transformation par cette table, il s'agit d'un utilisateur de la machine courante.
L'entretien de la table des alias est de la responsabilité de l'administrateur de la machine. La table des alias d'un domaine est un fichier stratégique qu'il convient de mettre à jour soigneusement (droits d'accès, utilisateurs inexistants, boucles...).
À chaque changement dans cette table l'administrateur doit fabriquer une table d'accès rapides ("hash table") à l'aide de la commande "`sendmail -bi`" souvent liée à "`newaliases`"

– `/etc/userdb` Cette table, "User Database" permet au sendmail d'effectuer une traduction de chaîne sur les noms des utilisateurs pour les courriers sortants. Cette disposition permet de traduire un nom de login en "Prenom.Nom", donc d'avoir une adresse de retour de la forme "Prenom.Nom@domaine" ce qui fait toujours plus chic !
– `/var/spool/mqueue`. Dans ce répertoire sont stockés les mails en attente d'une délivrance. Il peuvent y rester plusieurs jours (c'est un paramètre de la configuration du sendmail), on peut visualiser cette file d'attente avec la commande `mailq`.
Si un grand nombre de courriers sont en attente, et ça peut être le cas pour les machines relais, la section du disque dur qui supporte cette partition (ici `/var`) doit faire l'objet d'un dimensionnement en conséquence, sous peine d'obliger `sendmail` à refuser les mails faute de place disque.
– `/var/mail/userX`. Chaque utilisateur de la machine locale (il peut ne pas y en avoir sur un serveur) a une boîte aux lettres ("mail box") repérée comme un fichier ayant comme nom son login. Par exemple `/var/spool/mail/root`.
Ce fichier est mis à jour automatiquement par le MTA local en cas d'arrivée de courrier.
De même que pour le répertoire de file d'attente, le répertoire des boîtes aux lettres des utilisateurs doit faire l'objet d'une attention par-

ticulière de la part de l'administrateur ; la prolifération des "documents attachés" aux courriers électroniques est un fléaux contre lequel il est difficile de se prémunir sauf à agrandir perpétuellement la taille de la partition `/var... ! :(`

– `${HOME}/.forward` : avant d'être finalement délivré dans la boîte aux lettres de l'utilisateurs, `sendmail` lit le contenu de ce fichier, `${HOME}` étant le répertoire racine des fichiers de l'utilisateur en question.
Le fichier `.forward` est la base personnelle d'alias pour chaque utilisateur, ça permet de renvoyer son courrier vers d'autres sites, voire aussi d'effectuer des transformations avant de stocker les mails (`procmail`).
Si le `.forward` contient la chaîne suivante :

```
moi@un-autre-site.ailleurs
```

Tous les courriers envoyé à cette utilisateurs sont renvoyés à l'adresse indiquée.
Ou encore :

```
"|exec /usr/local/bin/procmail"
```

Qui permet d'invoquer l'usage du programme `procmail`, celui-ci est un puissant filtre qui permet de faire un tri des courriers électroniques par mots clefs (indispensable pour gérer les "mailing-lists"abondantes).

Enfin on peut remarquer qu'aucun signal n'est prévu pour indiquer à `sendmail` qu'il faut relire son fichier de configuration, c'est voulu par le concepteur. Lors de la mise au point de ce fichier, il faut arrêter (" `kill -TERM pid-du-sendmail`") puis relancer (`/usr/sbin/sendmail -bd -q30m`) manuellement...

4.3 Le cas de POP

POP est l'acronyme de " Post Office Protocol ", il permet l'accès à un serveur de courrier depuis des clients PC sous Windows, voire même des stations unix distantes, par exemple via `ppp`, qui ne sont pas configurées pour faire du SMTP. POP dans sa version 3 est défini par la [**RFC 1939**].

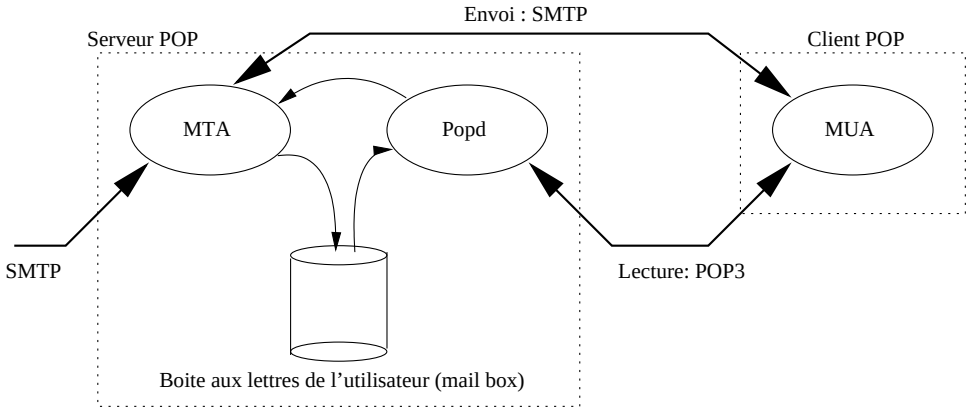


figure VIII.06

Les clients POP sont légions sur les PCs et sur les stations de travail sous Unix. Pour celles-ci citons : kmail[14], mh, pine, elm, mutt, gnus pour emacs, ou encore sylpheed. La liste n'est pas exhaustive, loin s'en faut.

POP est un protocole très simple qui fonctionne parfaitement mais qui n'est pas dénué de défauts :

1. L'authentification (login/password) est échangée en "clair" sur le réseau
2. Sur l'architecture Unix, l'utilisateur doit avoir un compte sur la machine serveur
3. Les messages doivent être récupérés sur le poste client pour être manipulés (en POP3 un double peut rester sur le serveur)
4. La boite aux lettres ne peut être consultée que par un seul client à la fois

Ces points deviennent vite rédhibitoires quand le poste client doit accéder au serveur au travers d'un réseau non sûr, et surtout lorsque le détenteur de la boite aux lettres veut consulter ses mails depuis des postes différents. Ce cas de figure est de plus la réalité des professionnels en déplacements.

C'est pourquoi d'autres solutions se sont développées et sont de plus en plus utilisées : les messageries accessibles via un browser web (webmail), donc qui utilisent comme support le protocole HTTP (voir page 151), ou un remplaçant du procole POP, le protocole IMAP !

[14] De l'environnement de bureau `KDE` - `http ://www.kde.org/`

4.4 Le cas de IMAP

Bien que largement déployé que depuis quelques années, le protocole IMAP — " Internet Message Access Protocol " — a été développé à l'université de Stanford en 1986. C'est actuellement la version 4rev1 qui est utilisée, définie dans la [**RFC 3501**].

L'architecture présentée *figure VIII.06* reste valable, pratiquement il suffit d'y remplacer le mot " POP " par " IMAP[15] " !

Les fonctionnalités sont plus riches et surtout pallient aux inconvénients listés pour POP. En clair, les points négatifs listés pour POP sont tous réglés. Imap est conçu pour pouvoir accéder à ses boites aux lettres depuis de multiples machines, n'importe où sur le réseau, alors que POP est plus adapté à une machine unique.
Voici ses objectifs principaux :

1. Être compatible avec les standards, MIME par exemple
2. Permettre l'accès et la gestion des mails depuis plus d'une machine
3. Fournir un mode de fonctionnement en-ligne et hors-ligne
4. Supporter les accès concourrants aux mêmes boites aux lettres
5. Être indépendant du stockage des mails (fichiers ou base de données, par exemple)

Un excellent comparatif des deux protocoles est accessible ici :

`http ://www.imap.org/papers/imap.vs.pop.brief.html`

5 Configuration du Sendmail

Cette section est une extraction libre et incomplète du paragraphe 5 du document intitulé "Sendmail Installation and Operation Guide", disponible dans toute distribution de la V8. On peut également trouver ce document dans la section "System Manager's Manual" (SMM) des systèmes BSD[16].

La configuration de `sendmail` est contenue essentiellement dans le fichier `/etc/mail/sendmail.cf` comme on peut le voir sur la *figure 5*.

Le fichier de configuration est organisé comme une série de lignes, le premier caractère de chacune d'elles en précise le type. Une ligne vide ou qui débute par un `#` est ignorée (commentaire), une ligne qui débute par un espace ou une tabulation est la continuation de la précédente.

[15] Consultez `http ://www.imap.org/` pour plus d'informations

[16] pour FreeBSD, NetBSD ou OpenBSD ça se trouve dans le répertoire /usr/share/doc/smm/08.sendmailop/

5.1 Règles de reécriture

Les règles de reécritures sont repérables comme ces lignes qui démarrent par un **S** (début d'un paquet de règles) ou un par un **R** (simple règle).

Les paquets de règles (organisation d'ensemble *figure 7*) ont comme but essentiel d'analyser puis de prendre les bonnes décisions en fonction des adresses trouvées dans l'en-tête. C'est une démarche purement formelle.

Ces règles utilisent une syntaxe dense qui rebute généralement les administrateurs. Il faut imaginer qu'elles sont intégralement analysées chaque fois que le programme `sendmail` est invoqué, c'est à dire en gros pour chaque e-mail entrant ou sortant. Elles doivent donc être faciles à analyser, même par un cpu de modeste performance (ou très chargé, ce qui revient finalement au même).

L'ensemble fonctionne comme un système de production, c'est à dire qui consomme des règles lues séquentiellement et les applique à un jeux de données initiales, ici une ou plusieurs adresses électroniques.

La partie gauche (ou **lhs**[17]) sert de déclencheur ("pattern matching") pour une règle.

La partie droite (ou **rhs**[18]) est déclenchée si le motif de la partie gauche est reconnu dans le flux de données. Le résultat produit, s'il existe, est reinjecté dans le système de production et ce jusqu'à épuisement des possibilités.

La *figure 7* donne un aperçu du fonctionnement de l'automate, les chiffres (0,1,2,3,4) sont autant de "ruleset", comprendre des paquets de règles qui sont regroupées ensembles parcequ'elles traitent d'un objectif commun.

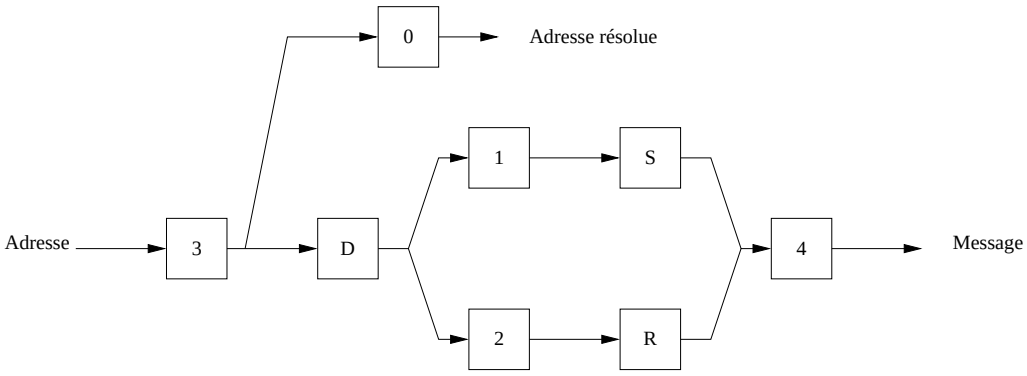


figure VIII.07

Les règles sont numérotées, le premier chiffre dit à quel paquet elles appartiennent.

[17] "left hand side"
[18] "right hand side"

Le paquet de règles 0 sert essentiellement à déterminer le “mailer” c'est à dire le moyen d'envoyer le courrier (SMTP, ESMTP, UUCP...).

Les paquets de règles 1 et 2 sont appliqués respectivement à l'en-tête des messages qui sortent (“send”) ou qui entrent (“receive”).

Le paquet de règles 3 est appliqué systématiquement à toutes les adresses.

Le paquet de règles 4 est appliqué à toutes les adresses dans le message, typiquement pour passer d'une forme interne à une forme externe.

Le paquet de règles 5, non représenté sur l'automate, sert à traiter les adresses locales après que les aliases aient été appliqués.

Les paquets de règles sont repérés par la lettre **S**, qui en balise le nom, comme dans :

```
######################################
###   Ruleset 0 -- Parse Address   ###
######################################

Sparse=0
```

Quant aux règles, elles démarrent toutes avec la lettre **R**, comme dans :

```
R$*                     $: $>Parse0 $1          initial parsing
R<@>                    $#local $: <@>          special case error msgs
R$*                     $: $>ParseLocal $1      handle local hacks
R$*                     $: $>Parse1 $1          final parsing
```

Deux remarques s'imposent :

1. Chaque ligne forme une règle, sur le modèle :

   R*lhs rhs commentaire*

2. Le séparateur de champ entre les trois partie de ces règles est la tabulation (une au minimum)

L'adresse **postmaster@mondomain.fr**, par exemple, quand elle se présente devant le paquet de règles 0 qui démarre ci-dessus, a été mise sous une forme canonique par d'autres règles appliquées préalablement (notament dans la “ruleset 3”) :

postmaster < @ mondomain . fr . >

Le “pattern matching” va tenter de rapprocher cette suite de mots ou tokens de la partie gauche des règles (**lhs**) pour déterminer celles qui peuvent être déclenchées.

Dans la partie gauche **$⋆** s'applique à toute suite de tokens, même vide. Donc la première ligne convient. La deuxième ne le pourrait pas car la chaîne “**postmaster**” précède le caractère “<” et le “@” est suivi de “**mondomain . fr .**”.

La troisième et la quatrième règle sont également déclenchables mais présentement l'ordre d'apparition est également l'ordre de déclenchement, cette possiblité sera donc examinée éventuellement plus tard.

La partie droite de la première règle commence par **$: $>Parse0** ce qui signifie l'appel d'un autre paquet de règles que l'on pourra trouver plus loin dans le fichier `sendmail.cf` :

```
#
#  Parse0 -- do initial syntax checking and eliminate local addresses.
#       This should either return with the (possibly modified) input
#       or return with a #error mailer.  It should not return with a
#       #mailer other than the #error mailer.
#


SParse0
```

Le **$1** signifie que l'on ne transmet que le premier des tokens reconnus dans la partie gauche ("**postmaster**" dans l'exemple)...

5.2 Exemple de sortie de debug

Il est utile d'avoir ce schéma en tête quand on débogue les règles : avec l'option `-bt` de `sendmail` on peut suivre la progression de la transformation, règle par règle.

```
ADDRESS TEST MODE (ruleset 3 NOT automatically invoked)
Enter <ruleset> <address>
> 3,0 postmaster@mondomain.fr
rewrite: ruleset   3   input: postmaster @ mondomain . fr
rewrite: ruleset  96   input: postmaster < @ mondomain . fr >
rewrite: ruleset  96 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset   3 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset   0   input: postmaster < @ mondomain . fr . >
rewrite: ruleset 199   input: postmaster < @ mondomain . fr . >
rewrite: ruleset 199 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset  98   input: postmaster < @ mondomain . fr . >
rewrite: ruleset  98 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset 198   input: postmaster < @ mondomain . fr . >
rewrite: ruleset  90   input: < mondomain . fr > postmaster < @ mondomain . fr . >
rewrite: ruleset  90   input: mondomain . < fr > postmaster < @ mondomain . fr . >
rewrite: ruleset  90 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset  90 returns: postmaster < @ mondomain . fr . >
rewrite: ruleset  95   input: < mailhub . mondomain . fr > postmaster < @ mondomain . fr . >
rewrite: ruleset  95 returns: $# relay $@ mailhub . mondomain . fr $: postmaster < @ mondomain . fr . >
rewrite: ruleset 198 returns: $# relay $@ mailhub . mondomain . fr $: postmaster < @ mondomain . fr . >
rewrite: ruleset   0 returns: $# relay $@ mailhub . mondomain . fr $: postmaster < @ mondomain . fr . >
>
```

Plus en TP...

6 Bibliographie

Pour en savoir davantage, on pourra consulter “les bons auteurs” suivants :

- Eric Allman — “Sendmail Installation and Operation Guide” — document au format PostScript jointe à toutes les distributions de la `V8.xx`.
- Bryan Costales with Eric Allman & Neil Rickert — “Sendmail” — OŔeilly & Associates Inc. — 1994
- “Installing and Administering ARPA Services” — Hewlett–Packard — 1991
- Douglas E. Comer — “Internetworking with TCP/IP – Volume I” (chapter 19) — Prentice All — 1988
- W. Richard Stevens — “TCP/IP Illustrated Volume I” (chapter 28) — Prentice All — 1994

Et pour en savoir encore plus...

RFC 821 “Simple Mail Transfer Protocol.” J. Postel. Aug-01-1982. (Format : TXT=124482 bytes) (Obsoletes RFC0788) (Also STD0010) (Status : STANDARD)

RFC 822 “Standard for the format of ARPA Internet text messages.” D. Crocker. Aug-13-1982. (Format : TXT=109200 bytes) (Obsoletes RFC0733) (Updated by RFC1123, RFC1138, RFC1148, RFC1327) (Also STD0011) (Status : STANDARD)

RFC 974 “Mail routing and the domain system.” C. Partridge. Jan-01-1986. (Format : TXT=18581 bytes) (Status : STANDARD)

RFC 1123 “Requirements for Internet hosts - application and support. R.T.” Braden. Oct-01-1989. (Format : TXT=245503 bytes) (Updates RFC0822) (Updated by RFC2181) (Status : STANDARD)

RFC 1652 “SMTP Service Extension for 8bit-MIMEtransport.” Klensin, N. Freed, M. Rose, E. Stefferud & D. Crocker. July 1994. (Format : TXT=11842 bytes) (Obsoletes RFC1426) (Status : DRAFT STANDARD)

RFC 1939 “Post Office Protocol - Version 3.” J. Myers & M. Rose. May 1996. (Format : TXT=47018 bytes) (Obsoletes RFC1725) (Updated by RFC1957) (Also STD0053) (Status : STANDARD)

RFC 2060 “ Internet Message Access Protocol - Version 4rev1.” M. Crispin. December 1996. (Format : TXT=166513 bytes) (Obsoletes RFC1730) (Status : PROPOSED STANDARD)

RFC 2184 “MIME Parameter Value and Encoded Word Extensions : Character Sets, Languages, and Continuations.” N. Freed, K. Moore. August 1997. (Format : TXT=17635 bytes) (Updates RFC2045, RFC2047, RFC2183) (Status : PROPOSED STANDARD)

RFC 2476 “Message Submission.” R. Gellens, J. Klensin. December 1998. (Format : TXT=30050 bytes) (Status : PROPOSED STANDARD)

RFC 2821 “Simple Mail Transfer Protocol.” J. Klensin, Ed.Àpril 2001. (Format : TXT=192504 bytes) (Obsoletes RFC0821, RFC0974, RFC1869) (Status : PROPOSED STANDARD)

RFC 2822 “Internet Message Format.” P. Resnick, Ed.Àpril 2001. (Format : TXT=110695 bytes) (Obsoletes RFC0822) (Status : PROPOSED STANDARD)

Chapitre IX

Anatomie d'un serveur Web

1 Le protocole HTTP

Ce document est une présentation succincte du protocole HTTP 1.0 et du serveur Apache qui l'utilise.

Le protocole HTTP[1] est le protocole d'application utilisé pour véhiculer, entres autres, de l'HTML[2] sur l'Internet.

C'est le protocole le plus utilisé en volume depuis 1995, devant FTP, NNTP et SMTP ; il accompagne l'explosion de l'utilisation du système global d'information "World–Wide Web".

Depuis 1990, date d'apparition du "Web", le protocole HTTP évolue doucement mais surement. Il est longtemps resté sous forme de "draft". La première version déployée largement a été la 1.0, définie dans la RFC 1945 de mai 1996. Depuis le début du mois de janvier 1997 est apparue la version 1.1, deux fois plus volumineuse pour tenir compte des nouvelles orientations de l'usage du service.

Aujourd'hui ce protocole est tellement répandu que pour bon nombre de nouveaux utilisateurs du réseau, l'Internet c'est le "web" !

Techniquement, l'usage de ce protocole se conçoit comme une relation entre un client et un serveur. Le client, appelé génériquement un "browser", un "User Agent", ou encore *butineur de toile*, interroge un serveur connu par son "url[3]" dont la syntaxe est bien décrite dans la RFC 1738.

Par exemple la chaîne de caractères `http ://www.sio.ecp.fr/` est une url ; il suffit de la transmettre en tant qu'argument à un quelconque outil d'exploration et celui-ci vous renverra (si tout se passe comme prévu !) ce qui est prévu sur le serveur en question pour répondre à cette demande (car il s'agit bien d'une requête comme nous le verrons plus loin dans ce chapitre).

Le serveur, supposé à l'écoute du réseau au moment où la partie cliente l'interroge, utilise un port connu à l'avance. Le port 80 est dédié officiellement

[1] "Hypertext Transfer Protocol"

[2] "Hypertext Markup Language" — Consulter les "technical reports and publications" du site : `http ://www.w3.org/pub/WWW/TR/`

[3] "Uniform Resource Locator"

au protocole http[4], mais ce n'est pas une obligation (cette décision est prise à la configuration du serveur). L'url qui désigne un serveur peut contenir dans sa syntaxe le numéro de port sur lequel il faut l'interroger, comme dans :

`http ://www.sio.ecp.fr :11235/.`

1.1 Exemple d'échange avec `http`

Le transport des octets est assuré par TCP[5] et le protocole est "human readable", ce qui nous autorise des essais de connexion avec le client tcp à tout faire : `telnet` ! Bien entendu on pourrait utiliser un "browser" plus classique, mais celui-ci gérant pour nous le détail des échanges il ne serait plus possible de les examiner.

```
$ telnet localhost 80
Trying...
Connected to localhost.
Escape character is '^]'.
GET / HTTP/1.0


HTTP/1.1 200 OK
Date: Fri, 01 Mar 2002 10:59:06 GMT
Server: Apache/1.3.23 (Unix)
Last-Modified: Sat, 10 Nov 2001 16:13:02 GMT
ETag: "1381-8b-3bed520e"
Accept-Ranges: bytes
Content-Length: 79
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Ceci est un titre</TITLE>
</HEAD>
<BODY>
</BODY>
</HTML>
Connection closed by foreign host.
```

Ce qui est tapé par l'utilisateur et la réponse du serveur.

La requête, suivie d'une ligne vide.

Enfin la réponse du serveur, que l'on peut décomposer en trois parties :

1. Un code de retour (HTTP)
2. Un en-tête MIME
3. Des octets, ici ceux d'une page écrite en HTML.

Notons également la deconnexion à l'initiative du serveur, en fin d'envoi de la page HTML.

1.2 Structure d'un échange

L'exemple qui précède est typique d'un échange entre le client et le serveur : une question du client génère une réponse du serveur, le tout lors d'une connexion TCP qui se termine lors de l'envoi du dernier octet de la réponse (clôture à l'initiative du serveur).

Le serveur ne conserve pas la mémoire des échanges passés, on dit aussi qu'il est sans état, ou "stateless".

[4] `http ://www.iana.org/assignments/port-numbers`
[5] page 85

La question et la réponse sont bâties sur un modèle voisin : le message HTTP.

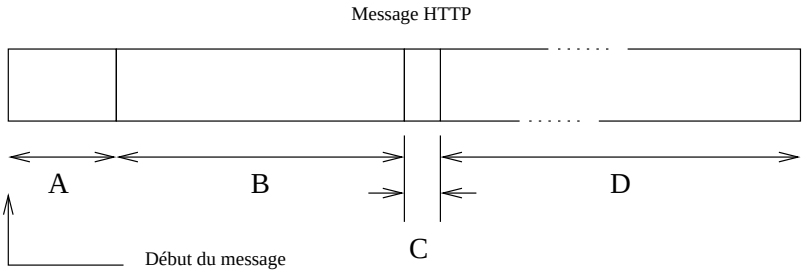


figure IX.01

Les parties A, B et C forment l'en-tête du message, et D le corps.

A La première ligne du message, est soit la question posée ("request-line"), soit le statut de la réponse ("status-line").

– La question est une ligne terminée par `CRLF`, elle se compose de trois champs :

Une méthode à prendre dans `GET, HEAD`, ou `POST`.

`GET` Plus de 99% des requêtes ont cette méthode, elle retourne l'information demandée dans l'URL (ci-dessous).

`HEAD` La même chose que GET, mais seul l'en-tête du serveur est envoyé. Utile pour faire des tests d'accessibilité sans surcharger la bande passante. Utile également pour vérifier de la date de fraicheur d'un document (information contenue dans l'en-tête).

`POST` Cette méthode permet d'envoyer de l'information au serveur, c'est typiquement le contenu d'un formulaire rempli par l'utilisateur.

Une ressource que l'on désigne par une URL[6].

Par exemple `http ://www.site.org/`.

La version du protocole , sous forme `HTTP-`*Numéro de version*. Par exemple `HTTP/1.1` !

– La réponse. Cette première ligne n'est que le statut de la réponse, les octets qui la détaillent se trouvent plus loin, dans le corps du message. Trois champs la composent, elle se termine par `CRLF` :

La version du protocole , sous forme `HTTP-`*Numéro de version*, comme pour la question.

Statut C'est une valeur numérique qui décrit le statut de la réponse. Le premier des trois digits donne le sens général :

[6] "Uniform Resource Locator", consulter la RFC 1630

- $1xx$ N'est pas utilisé "Futur Use"
- $2xx$ Succès, l'action demandée a été comprise et exécutée correctement.
- $3xx$ Redirection. La partie cliente doit reprendre l'interrogation, avec une autre formulation.
- $4xx$ Erreur coté client. La question comporte une erreur de syntaxe ou ne peut être acceptée.
- $5xx$ Erreur coté serveur. Il peut s'agir d'une erreur interne, due à l'OS ou à la ressource devenue non accessible.

Phrase C'est un petit commentaire ("Reason- Phrase") qui accompagne le statut, par exemple le statut 200 est suivi généralement du commentaire "OK" !

B C'est une partie optionnelle, qui contient des informations à propos du corps du message. Sa syntaxe est proche de celle employée dans le courrier électronique, et pour cause, elle repecte aussi le standard MIME[7].

Un en-tête de ce type est constitué d'une suite d'une ou plusieurs lignes (la fin d'une ligne est le marqueur `CRLF`) construite sur le modèle :

`Nom_de_champ` : *Valeur_du_champ* `CRLF`

Éventuellement le marqueur de fin de ligne peut être omis pour le séparateur ";".

Exemple d'en-tête MIME :

```
Date: Fri, 01 Mar 2002 10:59:06 GMT
Server: Apache/1.3.23 (Unix)
Last-Modified: Sat, 10 Nov 2001 16:13:02 GMT
ETag: "1381-8b-3bed520e"
Accept-Ranges: bytes
Content-Length: 79
Connection: close
Content-Type: text/html
```

`Date` : C'est la date à laquelle le message a été envoyé. Bien sûr il s'agit de la date du serveur, il peut exister un décalage incohérent si les machines ne sont pas synchronisées (par exemple avec XNTP).

`Server` : Contient une information relative au serveur qui a fabriqué la réponse. En générale la liste des outils logiciels et leur version.

`Content-type` : Ce champ permet d'identifier les octets du corps du message.

`Content-length` : Désigne la taille (en octets) du corps du message, c'est à dire la partie D de la *figure 1.*

`Last-modified` : Il s'agit de la date de dernière modification du fichier demandé, comme l'illustre le résultat de la commande `ll` (voir aussi la coïncidence de la taille du fichier et la valeur du champ précédent).

[7] "Multipurpose Internet Mail Extension", consulter la RFC 1521

```
-rw-r--r--  1 web   doc    139 Nov 10 17:13 index.html
```

`ETag` : C'est un identificateur du serveur, constant lors des échanges. C'est un moyen de maintenir le dialogue avec un serveur en particulier, par exemple quand ceux-ci sont en grappe pour équilibrer la charge et assurer la redondance.

C Une ligne vide (`CRLF`) qui est le marqueur de fin d'en-tête. Il est donc absolument obligatoire qu'elle figure dans le message. Son absence entraine une incapacité de traitement du message, par le serveur ou par le client.

D Le corps du message. Il est omis dans certains cas, comme une requête avec la méthode `GET` ou une réponse à une requête avec la méthode `HEAD`.

C'est dans cette partie du message que l'on trouve par exemple les octets de l'HTML, ou encore ceux d'une image...

Le type des octets est intimement lié à celui annoncé dans l'en-tête, plus précisement dans le champ `Content-Type`.

Par exemple :

`Content-Type : text/html` $\Longrightarrow$ Le corps du message contient des octets à interpréter comme ceux d'une page écrite en HTML.

`Content-Type : image/jpg` $\Longrightarrow$ Le corps du message contient des octets à interpréter comme ceux d'une image au format `jpeg`

2 URIs et URLs

Le succès du "web" s'appuie largement sur un système de nommage des objets accessibles, qui en uniformise l'accès, qu'ils appartiennent à la machine sur laquelle on travaille ou distants sur une machine en un point quelconque du réseau (mais supposé accessible). Ce système de nommage universel est l'**url** ("Uniform Resource Locator" — RFC 1738) dérivé d'un système de nommage plus général nommé **uri** ("Universal Resource Identifier" — RFC 1630).
La syntaxe générale d'un(e) url est de la forme :

<scheme> :<scheme-specific-part>

Succintement la "scheme" est une méthode que l'on sépare à l'aide du caractère " :" d'une chaîne de caractères ascii 7 bits dont la structure est essentiellement fonction de la "scheme" qui précède et que l'on peut imaginer comme un argument.

Une "scheme" est une séquence de caractères 7bits. Les lettres "a" à "z", les chiffres de "0" à "9", le signe "+", le "." et le "-" sont admis. Majuscules et minuscules sont indifférenciés.

Exemples de "schemes" : **http, ftp, file, mailto**, ...Il en existe d'autres (cf la RFC) non indispensables pour la compréhension de cet exposé.

Globalement une url doit être encodée en ascii 7 bits[8] sans caractère de contrôle (c'est à dire entre les caractères 20 et 7F), ce qui a comme conséquence que tous les autres caractères doivent être encodés.

La méthode d'encodage transforme tout caractère non utilisable directement en un triplet formé du caractère "%" et de deux caractères qui en représente la valeur hexadécimale. Par exemple l'espace (20 hex) doit être codé **%20**.

Un certain nombre de caractères, bien que théoriquement représentables, sont considérés comme non sûrs ("unsafe") et devraient être également encodés de la même manière que ci-dessus, ce sont :

```
% < > " # { } | \ ^ ~ [ ] ‘
```

Pour un certain nombre de "schemes" (**http**...) certains caractères sont réservés car ils ont une signification particulière. Ce sont :

```
; / ? : @ = &
```

Ainsi, s'ils apparaissent dans l'url sans faire partie de sa syntaxe, ils doivent être encodés.

2.1 Scheme http

Une url avec la "scheme" **http** bien formée doit être de la forme :

[8] `man ascii` sur toute machine unix

http ://<host> :<port>/<path> ?<searchpath>

"path" et "searchpath" sont optionnels.

host C'est un nom de machine ou une adresse IP.

port Le numéro de port. S'il n'est pas précisé, la valeur 80 est prise par défaut.

path C'est un sélecteur au sens du protocole **http**.

searchpath C'est ce que l'on appelle la "query string", autrement dit la chaîne d'interrogation.

À l'intérieur de ces deux composantes, les caractères **/ ; ?** sont réservés, ce qui signifie que s'ils doivent être employés, ils doivent être encodés pour éviter les ambiguïtés.

Le **?** marque la limite entre l'objet interrogeable et la "query string". À l'intérieur de cette chaîne d'interrogation le caractère **+** est admis comme raccourci pour l'espace (ascii 20 hex). Il doit donc être encodé s'il doit être utilisé en tant que tel.

De même, à l'intérieur de la "query string" le caractère **=** marque la séparation entre variable et valeur, le **&** marque la séparation entre les couples **variable = valeur**.

Exemple récapitulatif :

http ://www.google.fr/search ?q=cours+r%E9seaux&hl=fr&start=10&sa=N

Notez le "**é**" codé **%E9**, c'est à dire le caractère de rang $14 \times 16 + 9 = 233$. On peut également observer quatre variables **q, hl, start** et **sa** dont la signification peut être partiellement devinée, mais dont le remplissage reste à la charge du serveur en question.

Le rôle de la chaîne "**search**" est celui de ce que l'on appelle une CGI ou "Common Gateway Interface", c'est à dire un programme qui effectue le lien entre le serveur interrogé et des programmes d'application, ici une recherche dans une base de données. Nous examinons succinctement le principe de fonctionnement de tels programmes page 171.

3 Architecture interne du serveur Apache

Cette partie se consacre à l'étude du fonctionnement du serveur Apache[9].

Pour comprendre les mécanismes internes d'un tel serveur il est absolument nécessaire de pouvoir en lire le code source, cette contrainte exclu les produits commerciaux.

Au mois de mars 2002, d'après le "Netcraft Web Server Survey[10]" le serveur le plus utilisé (55%) est très majoritairement celui du projet Apache.

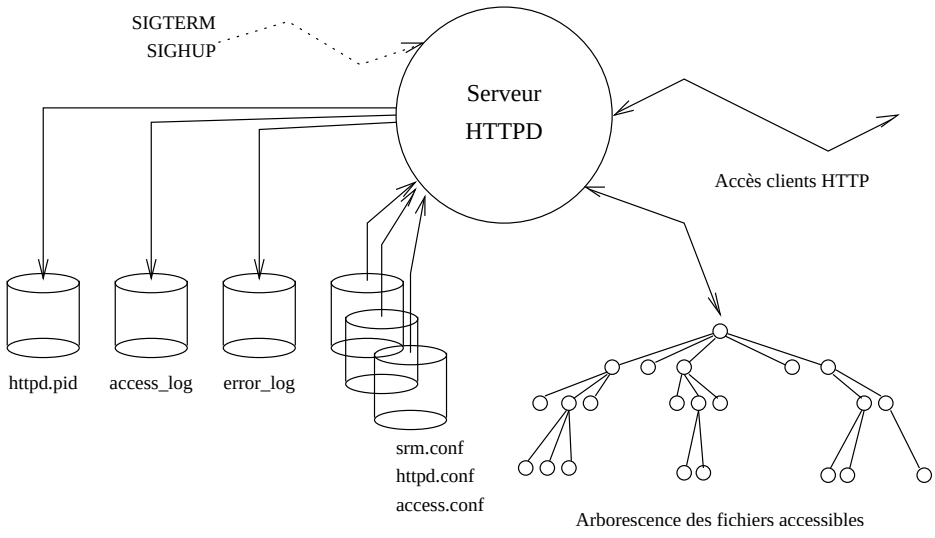
D'après ses auteurs, le serveur Apache est une solution de continuité au serveur du NCSA[11]. Il corrige des bugs et ajoute de nombreuses fonctionnalités, particulèrement un mécanisme d'API pour permettre aux administrateurs de sites de développer de nouveaux modules adaptés à leurs besoins propres.

Plus généralement, tout ce qui n'est pas strictement dans les attributions du serveur (gestion des processus, gestion mémoire, gestion du réseau) est traité comme un module d'extension. Le fichier `apache_1.3.23/htdocs/manual/misc/API.html` de la distribution standard apporte des précisions, le serveur `http ://www.apacheweek.com/` pointe également sur grand nombre de documents très utiles.

Le serveur Apache introduit également la possibilité de serveurs multi-domaines (domaines virtuels), ce qui est fort apprécié des hébergeurs de sites.

3.1 Environnement d'utilisation

La *figure 2* qui suit, synthétise l'environnement d'utilisation.



[9] "Apache HTTP server project", `http ://www.apache.org`, `http ://www.apacheweek.com/` est également une source intéressante d'informations

[10] `http ://www.netcraft.com/survey/`

[11] "National Center for Supercomputing Applications" – Université de l'Illinois, aux États Unis

Le serveur se met en œuvre simplement. La compilation fournit un exécutable, `httpd`, qui, pour s'exécuter correctement, a besoin des trois fichiers ASCII de configuration : `srm.conf, access.conf, httpd.conf`. C'est en fait dans celui-ci que sont effectués l'essentiel des ajustements locaux à la configuration standard.
Lors de l'exécution, trois fichiers sont modifiés[12] :

`httpd.pid` (`PidFile`) contient le "process ID" du "leader" de groupe ; à utiliser avec le signal `SIGHUP` (cf *figure 2*) pour provoquer la re-lecture et le re-démarrage "à chaud" du serveur, ou avec `SIGTERM` pour mettre fin à son activité.

`access_log` (`CustomLog`) Qui contient le détail des accès clients. Ce fichier peut devenir très volumineux.

`error_log` (`ErrorLog`) Qui contient le détail des accès infructueux et des éventuels problèmes de fonctionnement du serveur.

Le "daemon" **httpd** est soit du type (`ServerType`) "standalone" ou si son invocation est occasionnelle, à la demande piloté par `inetd` (cf page 4).

Il démarre son activité par ouvrir le port (`Port`) désigné dans la configuration, puis s'exécute avec les droits de l'utilisateur `User` et du groupe `Group`. Sa configuration se trouve dans le répertoire `conf` sous-répertoire de `ServerRoot`. Les fichiers accessibles par les connexions clientes, eux, se situent dans le répertoire `DocumentRoot` qui en est la racine, c'est à dire vue comme "/" par les browsers clients.

Le fichier `httpd.pid` contient un numéro de processus "leader" de groupe car en fait le serveur Apache, dès son initialisation, démarre un certain nombre de processus, autant de serveurs capables de comprendre les requêtes des clients. En voici un exemple :

`MinSpareServers 5`	C'est le nombre minimum d'instances du serveur (non compris le processus maître) en attente d'une connexion. S'il en manque, elles sont créées.
`MaxSpareServers 10`	C'est le nombre maximum d'instances du serveur (non compris le processus maître) en attente d'une connexion. S'il y en a de trop elles sont supprimées.
`StartServers 5`	C'est le nombre minimum d'instances du serveur (non compris le processus maître) au démarrage.
`MaxClients 150`	C'est le nombre maximum de clients (requêtes HTTP) simultanés. Cette constante peut être augmentée en fonction de la puissance de la machine.

Un processus joue le rôle de régulateur, du point de vue Unix c'est un processus chef de groupe ("leader"). La commande `ps` permet de visualiser une situation opérationnelle :

[12]Entre parenthèses le nom de la variable du le fichier `httpd.conf` qui permet d'en modifier le chemin

```
  web 17361  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
 root  2794     1  0  Feb 23  ? 0:06 /usr/local/bin/httpd -d /users/web
  web 17363  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
  web 17270  2794  0  Mar 13  ? 0:01 /usr/local/bin/httpd -d /users/web
  web 17362  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
  web 17401  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
  web 17313  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
  web 17312  2794  0  Mar 13  ? 0:01 /usr/local/bin/httpd -d /users/web
  web 17355  2794  0  Mar 13  ? 0:00 /usr/local/bin/httpd -d /users/web
  web 17314  2794  0  Mar 13  ? 0:01 /usr/local/bin/httpd -d /users/web
```

Ici il y a 9 instances du serveurs et le processus maître, qu'il est aisé de reconnaitre (2794) car il est le père des autres processus (ce qui n'implique pas qu'il en est le chef de groupe, mais la suite de l'analyse nous le confirmera).

La commande `netstat -f inet -a | grep http` ne donne qu'une ligne :

```
tcp        0      0  *.http                 *.*                    LISTEN
```

Cela signifie qu'aucune connexion n'est en cours sur ce serveur, et qu'une seule "socket" est en apparence à l'écoute du réseau. C'est une situation qui peut sembler paradoxale eu égard au nombre de processus ci-dessus, le code nous fournira une explication au paragraphe suivant.

La commande `tail -1 logs/access.log` fournit la trace de la dernière requête :

```
www.chezmoi.tld - - [01/Mar/2002:17:13:28 +0100] "GET / HTTP/1.0" 200 79
```

Il s'agit de la trace de notre exemple d'interrogation du début de ce chapitre !

3.2 Architecture interne

Attention, ce paragraphe concerne la version 1.1.1 du logiciel. Le fonctionnement de la version courante, 1.3.23, reste similaire mais le code ayant beaucoup changé, les numéros de lignes sont devenus de fait complètement faux.

Ce qui nous intéresse plus particulièrement pour expliquer le fonctionnement du serveur se trouve dans le répertoire `src/`, sous répertoire du répertoire principal de la distribution :

```
$ ll apache_1.1.1/
total 19
-rw-------  1 fla  users  3738 Mar 12  1996 CHANGES
-rw-------  1 fla  users  2604 Feb 22  1996 LICENSE
-rw-------  1 fla  users  3059 Jul  3  1996 README
drwxr-x---  2 fla  users   512 Feb  7 22:14 cgi-bin
drwxr-x---  2 fla  users   512 Feb  7 22:14 conf
drwxr-x---  2 fla  users   512 Feb  7 22:14 htdocs
drwxr-x---  2 fla  users  2048 Feb  7 22:14 icons
drwxr-x---  2 fla  users   512 Jul  8  1996 logs
drwxr-x---  2 fla  users  2048 Mar 12 10:42 src
drwxr-x---  2 fla  users   512 Feb  7 22:15 support
```

Dans ce répertoire qui compte au moins 75 fichier (`wc -l *.[ch]` ⇒ 27441 lignes) nous nous restreignons aux fichiers essentiels décrits dans le "README" soit encore une petite dizaine d'entres eux (`wc -l` ⇒ 6821 lignes) : `mod_cgi.c, http_protocol.c, http_request.c, http_core.c, http_config.c, http_log.c, http_main.c, alloc.c`

Dans un premier temps nous allons examiner le fonctionnement de la gestion des processus, du mode de relation entre le père et ses fils.

Puis, dans un autre paragraphe, nous examinerons plus particulièrement ce qui se passe lors d'une connexion, avec le cas particulier de l'exécution d'une CGI[13] qui comme son nom l'indique est le programme qui effectue l'interface entre le serveur HTTP et un programme d'application quelconque. Dans la suite de ce document le terme "cgi" employé seul désigne ce type d'application.

Gestion des processus

Au commencement est le `main`, et celui-ci se trouve dans le fichier `http_main.c`, comme il se doit !

```
 ...          ...
1035 pool *pconf;                    /* Pool for config stuff */
1036 pool *ptrans;                   /* Pool for per-transaction stuff */
 ...          ...
1472 int
1473 main(int argc, char *argv[])
1474 {
 ...          ...
1491     init_alloc();
1492     pconf = permanent_pool;
1493     ptrans = make_sub_pool(pconf);
```

La fonction `init_alloc` appelle `make_sub_pool` qui initialise un intéressant mécanisme de buffers chaînés, utilisé tout au long du code dès lors qu'il y a besoin d'allouer de la mémoire.

```
 ...          ...
1523     setup_prelinked_modules();
```

Les différents modules du serveurs sont ajoutés dans une liste chaînée.

```
1524
1525     server_conf = read_config (pconf, ptrans, server_confname);
1526
1527     if(standalone) {
1528         clear_pool (pconf);     /* standalone_main rereads... */
```

[13] "Common Gateway Interface"

```
1529          standalone_main(argc, argv);
1530      }
1531      else {
 ...            ...
1580      }
1581      exit (0);
1582 }
```

"Standalone" est à 0 si le serveur est invoqué depuis `inetd` [14]. Son mode de fonctionnement le plus efficace reste avec ce paramètre à 1 (voir le cours sur `inetd`), que nous supposons ici.

La fonction `standalone_main` (ligne 1362) prépare le processus à son futur rôle de serveur. Pour bien comprendre le cette fonction, il faut imaginer qu'elle est invoquée au démarrage, et "à chaud", pour lire et relire la configuration.

Ligne 1369 , la variable `one_process = 0` (sinon le processus est en mode debug) occasionne l'appel à la fonction `detach` (ligne 876). Celle-ci transforme le processus en "leader" de groupe avec la succession bien connue `fork` + `setsid` (pour plus de détails, voir le cours sur les `daemons`).

Ligne 1374 , la fonction `sigsetjmp` enregistre la position de la pile et l'état du masque d'interruption (deuxième argument non nul) dans `restart_buffer`. Tout appel ultérieur à `siglongjmp` forcera la reprise de l'exécution en cette ligne.

Ligne 1377 On ignore tout signal de type `SIGHUP`, cela bloque toute tentative cumulative de relecture de la configuration.

Ligne 1382 (`one_process = 0`) on envoie un signal `SIGHUP` à tous les processus du groupe. Cette disposition n'est utile que dans le cas d'un re-démarrage "à chaud". La variable `pgrp` est initialisée par la fonction `detach` (ligne 876), elle contient le PID du chef de groupe.

L'intéret d'avoir un groupe de processus est que le signal envoyé à son "leader" est automatiquement envoyé à tous les processus y appartenant.

Chaque processus qui reçoit ce signal, s'il est en mesure de le traiter, appelle la fonction `just_die` qui exécute un `exit(0)` (ligne 943). Donc tous les fils meurent, sauf le chef de groupe.

Ligne 1390 , l'appel à la fonction `reclaim_child_processes()` effectue autant de `wait` qu'il y avait de processus fils, pour éviter les zombis.

Ligne 1398 Relecture des fichiers de configuration.

Ligne 1400 `set_group_privs` (ligne 960) change le "user id" et le "group id" si ceux-ci ont été modifiés.

[14] Cette alternative est décidée dans le fichier `httpd.conf`, configuration du paramètre `ServerType`

Ligne 1401 `accept_mutex_init` (ligne 166) fabrique un fichier temporaire (`/usr/tmp/htlock.XXXXXX`), l'ouvre, réduit son nombre de lien à 0, de telle sorte qu'il sera supprimé dès la fin du processus. Ce fichier est le verrou d'accès à la socket principale, comme nous l'examinerons un peu plus loin.

Ligne 1402 `reinit_scoreboard` (ligne 596) Cette fonction remet à zéro, ou crée (`setup_shared_mem` ligne 432) la zone de mémoire commune entre le chef de groupe et ses processus fils. Cette zone mémoire est, soit un segment de mémoire partagée (IPC du système V), soit de la mémoire rendue commune par un appel à la primitive `mmap` (Le choix est effectué par `configure` qui analyse les possibilités du système, avant compilation du serveur).

La taille de cette zone est de `HARD_SERVER_LIMIT` × `sizeof(short_score)` octets, la structure `short_score`, définie dans `scoreboard.h`, recueille les statistiques d'utilisation de chaque serveur et son statut (par exemple `SERVER_READY, SERVER_DEAD`...). C'est par ce moyen que le serveur maître contrôle les serveurs fils.

`HARD_SERVER_LIMIT` définit le nombre maximum de connexions actives, donc d'instances de serveur, à un instant donné. En standard cette valeur est 150, elle peut être augmentée dans le fichier de configuration `httpd.conf` (voir ci-dessus au paragraphe II.1)

Enfin la *figure 4* montre son rôle stratégique.

Ligne 1413 (on suppose `listeners` = NULL), après avoir initialisé une structure d'adresse, appel à la fonction `make_sock` (ligne 1298). Celle-ci crée et initialise une socket, et, détail important, appelle par deux fois `setsockopt`, ce qui mérite un commentaire :

```
...      ...
1312     if((setsockopt(s, SOL_SOCKET,SO_REUSEADDR,(char *)&one,sizeof(one)))
1313        == -1) {
...      ...
1318     if((setsockopt(s, SOL_SOCKET,SO_KEEPALIVE,(char *)&keepalive_value,
1319         sizeof(keepalive_value))) == -1) {
...      ...
```

`SO_REUSEADDR` Indique que la règle d'exclusivité suivie par `bind`(2) pour l'attribution d'un port ne s'applique plus : un processus peut se servir d'un même port pour plusieurs usages différents (comme par exemple le client `ftp` qui attaque les port 21 et 20 du serveur avec le même port local), voire même des processus différents (c'est le cas ici) peuvent faire un `bind` avec le même port sans rencontrer la fatidique erreur 48 ("Address already in use") !

Vue des clients HTTP, le serveur est accessible uniquement sur le port 80, ce que nous avons remarqué au paragraphe II.1 (`netstat`) sans l'expliquer, voila qui est fait !

`SO_KEEPALIVE` Indique à la couche de transport, ici TCP, qu'elle doit émettre à interval régulier (non configurable) un message à destination de la socket distante. Que celle-ci n'y réponde pas et la prochaine tentative d'écriture sur la socket se solde par la réception d'un `SIGPIPE`, indiquant la disparition de la socket distante (voir plus loin la gestion de ce signal dans la fonction `child_main`, ligne 1139).

Ligne 1430 `set_signals` (1007) prévoit le comportement lors de la réception des signaux :

`SIGHUP`	Appel de `restart`
`SIGTERM`	Appel de `sig_term`

Ligne 1438 et la suivante, création d'autant de processus fils qu'il en est demandé dans le fichier de configuration (`StartServers`). La fonction `make_child` (1275) appelle `fork`, puis dans le fils modifie le comportement face aux signaux `SIGHUP` et `SIGTERM` (`just_die` appelle `exit`) avant d'exécuter `child_main`.

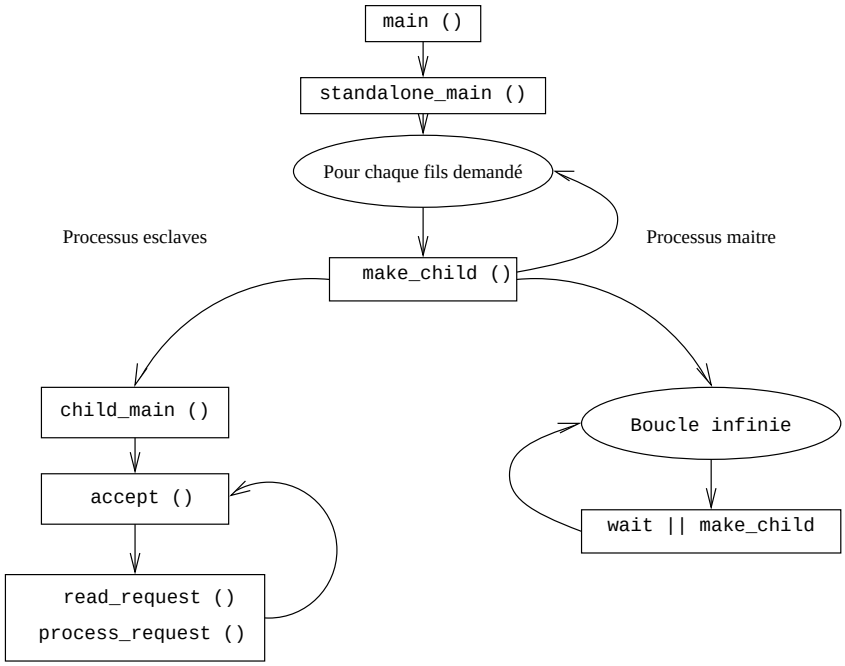


figure IX.03

Arrivés à ce stade, il nous faut analyser l'attitude des deux types de processus.

Le processus maître

Ligne 1444 démarre une boucle infinie de surveillance. Seule la réception et le traitement d'un signal peut l'interrompre.

Ligne 1458 Ici, si le nombre de serveurs disponibles est inférieur au nombre minimal requis, il y regénération de ceux qui manquent avec la fonction `make_child`

Les processus esclaves

Ligne 1139 La gestion de ces processus, autant de serveurs Web opérationnels, débute avec l'appel de la fonction `child_main`.

Ligne 1167 Début de la boucle infinie qui gère cette instance du serveur. Au cours d'une boucle le serveur gère l'acceptation d'une requête et son traitement.

Ligne 1174 Gestion du `SIGPIPE` donc des clients qui déconnectent avant l'heure !

Ligne 1180 Si le nombre de serveurs libres (`count_idle_servers`) est supérieur à la limite configurée, ou si

Ligne 1182 le nombre de requêtes traitées par ce processus a atteint la limite `max_requests_per_child`, le processus s'arrête de lui-même. C'est l'auto-régulation pour libérer des ressources occupées par un trop grand nombre de processus serveurs inutiles.

Ligne 1190 L'appel à la fonction `accept_mutex_on` vérouille l'accès à la ressource définie précédement avec `accept_mutex_init` (ligne 166). Ce vérouillage est bloquant et exclusif. C'est à dire qu'il faut qu'un autre processus en dévérouille l'accès pour que le premier processus sur la liste d'attente (gérée par le noyau Unix) y accède.

Ce fonctionnement est assuré suivant la version d'Unix par la primitive `flock` ou par la primitive `fcntl`.

Les sémaphore du Système V (`semget, semctl, semop`...) assurent la même fonctionnalité, en plus complet, ils sont aussi plus complexes à mettre en œuvre.

Cette opération est à rapprocher de l'option `SO_REUSEADDR` prise ligne 1312. Il faut éviter que plusieurs processus serveurs disponibles ne répondent à la requête. Il n'y a qu'un seul processus prêt à répondre à la fois et dès que le `accept` (ligne 1217) retourne dans le code utilisateur la ressource est dévérouillée (on suppose toujours `listeners = 0`).

Ligne 1221 La fonction `accept_mutex_off` dévérouille l'accès à la socket.

Ligne 1245 `read_request` lit la requête du client, donc un message HTTP.

Ligne 1247 `process_request` fabrique la réponse au client, donc un autre message HTTP.

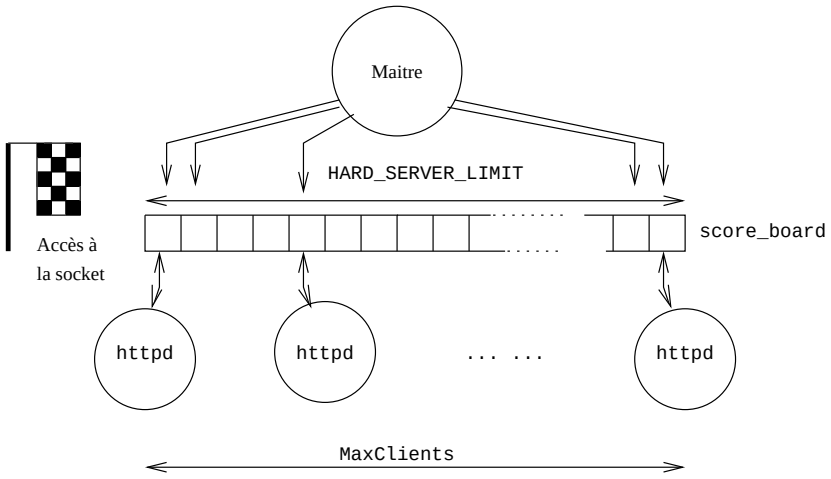


figure IX.04

Prise en main des requêtes

Le fichier concerné par la lecture de la requête est `http_protocol.c`.

La lecture de la première ligne du message HTTP est assurée par la fonction `read_request_line`, ligne 329[15].

La lecture de l'en-tête MIME est assurée par la fonction `get_mime_headers`, ligne 356. Attention, seule l'en-tête lue le corps du message dans le cas de a méthode POST est lu plus tard, lors du traitement du message, par le programme d'application (CGI).

Le fichier concerné par la formulation de la réponse est `http_request.c` et la fonction principale `process_request`, ligne 772. Celle-ci appelle en cascade `process_request_internal`, ligne 684.

Cette dernière effectue un certain nombre de tests avant de traiter effectivement la requête. Parmi ceux-ci on peut relever,

Ligne 716 La fonction `unescape_url` (util.c, ligne 593) assure le décodage des caractères réservés et transcodés comme il est spécifié par la RFC 1738.

Ligne 723 La fonction `getparents` filtre les chemins ("pathname") qui prêtent à confusion.

Ligne 768 La fonction `invoke_handler` est le point d'entrée dans le traitement de la requête. Cette fonction (`http_config.c`, ligne 267) invoque le programme (module) qui se charge de fabriquer la réponse, au vue du contenu du champ `content_type` de la requête. Si celui est inexistant, comme dans notre exemple du paragraphe I, il est positionné par défaut à la valeur `html/text`.

[15] La méthode, l'URL, et le protocole demandé

Deux types de CGIs

Pour la suite nous supposons que la prise en main de la requête est faite par le module "cgi", défini dans le fichier `mod_cgi.c`. Le point d'entrée est la fonction `cgi_handler` (ligne 207), c'est à dire celle qui appellée par `invoke_handler`, vue au paragraphe ci-dessus.

La lecture du code permet de déduire qu'il y a deux types de CGI, la distinction est faite parle nom de la cgi elle-même.

La *figure 5* résume les 2 situations possibles d'exécution d'une CGI.

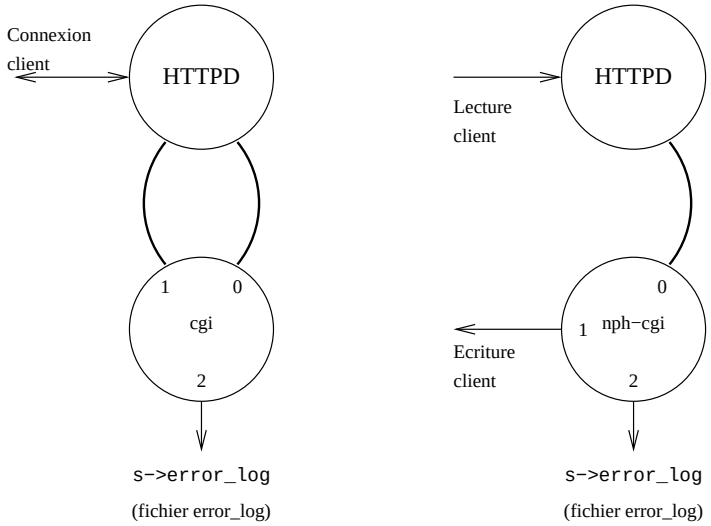


figure IX.05

Si le nom de l'exécutable commence par `nph-`[16] le comportement du serveur http change. Dans le cas ordinaire (nom quelconque) les données transmises depuis le client vers la cgi et réciproquement, passent par le processus httpd et via un tube non nommé ("pipe").

Dans le cas d'une cgi "nph", seules les données lues depuis le client (par exemple avec la méthode POST) transitent par le processus httpd, la réponse, elle, est émise directement, ce qui améliore les performances en évitant une séquence lecture/écriture inutile. Ce comportement est justifié dès que de gros volumes d'octets sont à transmettre au client (de grandes pages HTML, des images...).

Attention, dans ce dernier cas, c'est à la CGI de fabriquer l'intégralité du message HTTP, y compris l'en-tête MIME. A elle également de gérer la déconnexion prématurée du client (`SIGPIPE`).

Ces deux modes de fonctionnement ne sont pas clairement documentés, en fait il s'agit d'une caractéristique du serveur du *CERN*, maintenue pour assurer sans doute la compatibilité avec les applicatifs déjà écrits. Il n'est pas assuré que cette possibilité existera dans les futures versions du serveur

[16] "Non Parse Header"

Apache, notamment celles qui utiliseront la version 1.1 d'HTTP.

Examinons le code du fichier `mod_cgi.c` :

```
207 int cgi_handler (request_rec *r)
208 {
209     int nph;
...     ...
222     nph = !(strncmp(argv0,"nph-",4));
...     ...
```

La variable `nph` vaut 1 si la cgi est de ce type.

```
...     ...
251     add_common_vars (r);
...     ...
```

Ici on commence à fabriquer l'environnement d'exécution de la cgi Cette fonction (fichier `util_script.c`, ligne 126) complète les variables qui ne dépendent pas du contenu de la requête, par exemple `SERVER_SOFTWARE`, `REMOTE_HOST`,...

```
...     ...
277      if (!spawn_child (r->connection->pool, cgi_child, (void *)&cld,
278                        nph ? just_wait : kill_after_timeout,
279                        &script_out, nph ? NULL : &script_in)) {
280         log_reason ("couldn't spawn child process", r->filename, r);
281         return SERVER_ERROR;
282     }
...     ...
```

L'appel de cette fonction provoque la création d'un processus fils, celui qui finalement va exécuter la cgi. Il faut remarquer le deuxième argument qui est un pointeur de fonction (`cgi_child`), et le sixième qui est nul dans le cas d'une cgi du type "nph".

`script_in` et `script_out` sont respectivement les canaux d'entrée et sortie des données depuis et vers le processus qui exécute la cgi. Il parait donc logique que dans le cas d'une cgi de type "nph" `script_in` soit nul. Un mécanisme non encore analysé duplique la sortie de ce processus vers le client plutôt que vers le processus serveur.

Nous continuons la description du point de vue du processus père, donc httpd.

Ligne 295 et les suivantes, jusqu'à la ligne 332, le processus lit ce que le client lui envoie, si la méthode choisie est du type POST. Le contenu est renvoyé vers le processus fils, sans transformation :

```
311             if (fwrite (argsbuffer, 1, len_read, script_out) == 0)
312                 break;
...     ...
335     pfclose (r->connection->pool, script_out);
```

Il est donc clair que c'est à celui-ci d'exploiter ce qu'envoie le client, par exemple le résultat d'une forme de saisie.

```
337     /* Handle script return... */
338     if (script_in && !nph) {
...         ...
373         send_http_header(r);
374         if(!r->header_only) send_fd (script_in, r);
375         kill_timeout (r);
376         pfclose (r->connection->pool, script_in);
377     }
```

Ce test traite le cas d'une cgi normale, dont la sortie est lue par le serveur, puis renvoyée au client (ligne 374).

Examinons maintenant comment se prépare et s'exécute le processus fils :

```
101 void cgi_child (void *child_stuff)
102 {
...         ...
126     add_cgi_vars (r);
127     env = create_environment (r->pool, r->subprocess_env);
```

Ces deux dernières lignes préparent l'environnement du futur processus fils. Il s'agit du tableau de variables accessibles depuis la variable externe `environ`, pour tout processus. La fonction `add_cgi_vars` (fichier `util_script.c`, ligne 192) ajoute, entres autres, les variables `REQUEST_METHOD` et `QUERY_STRING` à l'environnement.

Cette dernière variable joue un rôle majeur dans la transmission des arguments à la cgi quand la méthode choisie est GET. En effet, dans ce cas, le seul moyen pour le client d'envoyer des arguments à la cgi est d'utiliser l'URL, comme par exemple dans :

```
http ://monweb.chez.moi/cgi-bin/nph-qtp ?base=datas&mot=acacia&champ=.MC
```

La syntaxe de l'URL prévoit le caractère " ?" comme séparateur entre le nom et ses arguments. Chaque argument est ensuite écrit sous la forme :

```
nom = valeur
```

Les arguments sont séparés les uns des autres par le caractère "&".

```
135     error_log2stderr (r->server);
136
...         ...
138     if (nph) client_to_stdout (r->connection);
```

Ligne 135, la sortie d'erreur est redirigée vers le fichier `error_log`, et ligne 138, la sortie standard du processus, c'est la socket, donc envoyée directement vers le client !

```
184     if((!r->args) || (!r->args[0]) || (ind(r->args,'=') >= 0))
185         execle(r->filename, argv0, NULL, env);
186     else
187         execve(r->filename, create_argv(r->pool, argv0, r->args), env
```

Puis finalement on exécute la cgi ! Bien sûr, si le programme va au delà de la ligne 187, il s'agit d'une erreur...

4 Principe de fonctionnement des CGIs

4.1 CGI — Méthode GET, sans argument

Dans ce paragraphe nous examinons ce qui se passe lors de l'exécution d'une CGI très simple (shell script, le source suit) que l'on suppose placée dans le répertoire idoine, pour être exécutée comme lors de la requête suivante :

```
$ telnet localhost 80
Trying ...
Connected to localhost.
Escape character is '^]'.
GET /cgi-bin/nph-test-cgi HTTP/1.0
```

La requête tapée par l'utilisateur.

```
HTTP/1.0 200 OK
Content-type: text/plain
Server: Apache/1.1.1
```

L'en-tête du message HTTP renvoyé par le serveur. La ligne de statut est générée par la cgi car il s'agit d'une cgi de type `nph-`

```
CGI/1.0 test script report:

argc is 0. argv is .

SERVER_SOFTWARE = Apache/1.1.1
SERVER_NAME = www.chezmoi.tld
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = GET
SCRIPT_NAME = /cgi-bin/nph-test-cgi
QUERY_STRING =
REMOTE_HOST = labas.tresloin.tld
REMOTE_ADDR = 192.168.0.1
REMOTE_USER =
CONTENT_TYPE =
CONTENT_LENGTH =
Connection closed by foreign host.
```

Le corps du message. Ces octets sont générés dynamiquement par le programme `nph-test-cgi`.

Et voici le source de cet exemple (une modification du script de test livré avec Apache) :

```
#!/bin/sh

echo HTTP/1.0 200 OK
echo Content-type: text/plain
echo Server: $SERVER_SOFTWARE
echo

echo CGI/1.0 test script report:
echo

echo argc is $#. argv is "$*".
echo
```

Remarquez la fabrication de l'entête MIME, réduite mais suffisante. Le `echo` seul génère une ligne vide, celle qui marque la limite avec le corps du message.

```
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo QUERY_STRING = $QUERY_STRING
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

Toutes ces variables sont celles de l'environnement généré par le module `cgi_handler` avant d'effectuer le `exec`.

4.2 CGI — Méthode GET, avec arguments

Examinons maintenant un cas plus compliqué, avec des arguments transmis, ou "query string". Celle-ci est transmise à la cgi via la variable d'environnement `QUERY_STRING`. C'est à la cgi de l'analyser puisque son contenu relève de l'applicatif et non du serveur lui-même. Essayons de coder la cgi de l'exemple :

```
http ://www.chezmoi.tld/cgi-bin/query ?base=datas&mot=acacia&champ=.MC
```

Première version :

```
#!/bin/sh
echo Content-type: text/plain
echo
echo "QUERY_STRING=>"$QUERY_STRING "<="
exit 0
```

L'interrogation avec un `telnet` produit la sortie :

```
QUERY_STRING=>base=datas&mot=acacia&champ=.MC<=
```

Se trouve très facilement sur les sites `ftp` le source d'un outil nommé `cgiparse`[17], parfaitement adapté à l'analyse de la chaîne transmise. D'où la **deuxième version :**

```
#!/bin/sh

CGIBIN=~web/cgi-bin
BASE=‘$CGIBIN/cgiparse -value base‘
MOT=‘$CGIBIN/cgiparse -value mot‘
CHAMP=‘$CGIBIN/cgiparse -value champ‘
```

Cette partie du code montre l'analyse du contenu de la variable `QUERY_STRING` avec l'outil `cgiparse`.

[17] Il est dans la distribution standard du serveur `httpd-3.0.tar.gz` du *CERN*

```
echo Content-type: text/plain
echo

echo "BASE="$BASE
echo "MOT="$MOT
echo "CHAMP="$CHAMP

exit 0
```

Et là, la fabrication du message renvoyé. La cgi renvoie ses octets via un tube au serveur, c'est donc celui-ci qui se charge de fabriquer un en-tête MIME.

Puis le résultat de l'exécution :

```
BASE=datas
MOT=acacia
CHAMP=.MC
```

4.3 CGI — Méthode POST

La méthode POST autorise un client à envoyer au serveur une liste de couples `variable=valeur`. Chaque couple est séparé des autres par une fin de ligne, c'est à dire (`CR`,`LF`).

Cette méthode est bien adaptée à la transmission d'informations collectées coté client dans une forme[18]de saisie, et dont le volume est variable.

La liste des couples est écrite dans le corps du message, par le programme client, et ne fait pas partie de l'URL, comme c'est le cas pour la méthode GET. Il y a évidement une limite au nombre maximum d'octets envoyé par le client, c'est le serveur qui en fixe la valeur[19]. Du coté du client, il faut prévenir le serveur, dans l'en-tête du message HTTP, de la taille du corps du message. Cela s'effectue avec le champ `Content-length :`.

L'exemple suivant ne fait que renvoyer la liste des couples lus, vers le client. Attention il ne fonctionne qu'avec `telnet`.

```
$ telnet www.chezmoi.tld 80
Trying 192.168.0.2...
Connected to www.chezmoi.tld.
Escape character is '^]'.
POST /cgi-bin/test-post HTTP/1.0
Content-length:14

areuh=tagada
HTTP/1.0 200 OK
Date: Mon, 24 Mar 1997 14:41:26 GMT
Server: Apache/1.1.1
Content-type: text/plain

REQUEST_METHOD = POST
CONTENT_LENGTH = 14
Couple lu : areuh=tagada
Connection closed by foreign host.
```

Le corps du message fait bien 14 caractères si on compte la fin de ligne (CR+LF).

La réponse du serveur liste les couples lus, ici un seul ! La variable globale `REQUEST_METHOD` pourrait être utilisée pour adapter le comportement de la cgi en fonction de la méthode demandée.

[18]Voir le "tag" `FORM` de l'HTML

[19]`HUGE_STRING_LEN` qui vaut 8192 en standard, définie dans le fichier `httpd.h`

Et voici le source de la cgi :

```
#!/bin/sh
#
# Ce script teste la methode POST
#
echo Content-type: text/plain
echo
echo REQUEST_METHOD = $REQUEST_METHOD
echo CONTENT_LENGTH = $CONTENT_LENGTH
while read l
do
    echo "Couple lu :" $l
done
exit 0
```

4.4 Ecriture d'une CGI en Perl

Voir cours de *Jean-Jacques Dhenin...*

5 Conclusion – Bibliographie

Rien n'est constant, tout change...Et il faut être constamment à l'affut de la nouveauté dans ce domaine très réactif qu'est celui du "World Wide Web".

Les quelques références bibliographique qui suivent illustrent ce cours, mais il est évident qu'elles sont bien insuffisantes pour couvrir tous les aspects que le terme "web" sous-entend !

RFC 1521 N. Borenstein, N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One : Mechanisms for Specifying and Describing the Format of Internet Message Bodies", 09/23/1993. (Pages=81) (Format=.txt, .ps) (Obsoletes RFC1341) (Updated by RFC1590)

RFC 1590 J. Postel, "Media Type Registration Procedure", 03/02/1994. (Pages=7) (Format=.txt) (Updates RFC1521)

RFC 1630 T. Berners-Lee, "Universal Resource Identifiers in WWW : A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web", 06/09/1994. (Pages=28) (Format=.txt)

RFC 1738 T. Berners-Lee, L. Masinter, M. McCahill, "Uniform Resource Locators (URL)", 12/20/1994. (Pages=25) (Format=.txt)

RFC 1945 T. Berners-Lee, R. Fielding, H. Nielsen, "Hypertext Transfer Protocol – HTTP/1.0", 05/17/1996. (Pages=60) (Format=.txt)

RFC 2068 R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee,"Hypertext Transfer Protocol – HTTP/1.1", 01/03/1997. (Pages=162) (Format=.txt)

TCP/IP Illustrated Volume 3 W. Richard Stevens – Addison-Wesley – Janvier 1996.

Chapitre X

Éléments de réseaux

1 Hôtes ou services virtuels

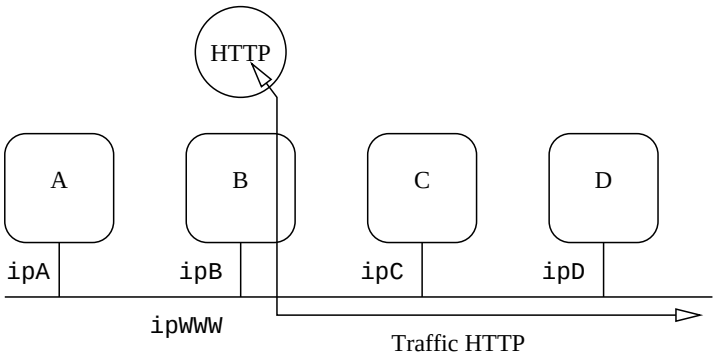


figure X.01 : Serveur HTTP virtuel

La machine B (d'adresse IP fixe ipB) héberge par exemple le service HTTP d'adresse ip ipWWW. Cette opération est rendue possible si le système d'exploitation de B autorise la notion d'alias IP.

Si les machines A, C et D exécutent également un serveur HTTP, elles peuvent potentiellement prendre le relais de la machine B, dès lors que l'adresse ipWWW aura été retirée de la machine B pour être reconfigurée sur l'une d'entres elles.

Cette opération peut se faire manuellement ou via un outil d'administration. Elle permet de faire transiter très rapidement des services d'une machine à une autre, sans rupture ou presque de la continuité. Vu des clients il s'agit toujours de la même machine, mais elle est virtuelle puisqu'elle n'existe pas physiquement.

Section en chantier, précisions en cours...

2 Tunnel IP

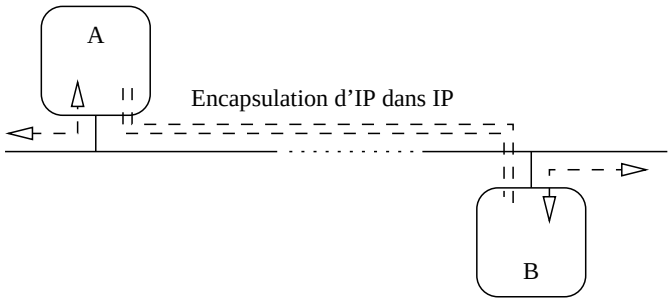


figure X.02 : Tunnel IP - Principe

Le tunnel permet d'encapsuler un protocole dans un autre de même niveau ou supérieur. Précédemment, page 28, nous avons déjà analysé l'encapsulation des couches de la pile Arpa selon une progression naturelle de fonctionnalités. Ici, si le principe d'encapsulation est conservé, la logique initiale de construction, elle, est bousculée. Par exemple on peut envisager d'encapsuler IP dans de multiple protocole autres qu'Ethernet, comme PPP[1], IP, dans une couche de transport comme TCP, voire même dans une couche applicative comme HTTP. Ce dernier exemple peut paraître " contre nature " et pourtant cela fonctionne...

Construire un tunnel a un coût : d'une part celui de la perte d'espace de données dans le datagramme (il faut loger un ou plusieurs en-têtes supplémentaires et le MTU reste constant lui !) et d'autre part celui du traitement supplémentaire (décapsulation, analyse) engendré par l'ajout de ces nouveaux en-têtes.

En résumé, construire un tunnel se traduit par une perte d'espace pour les données dans les datagrammes et par une consommation accrue de cycles cpus pour traiter les en-têtes supplémentaires. Heureusement le gain en fonctionnalités pour le réseau est substanciel, comme les exemples qui vont suivre l'illustrent !

Les tunnels qui transitent par une couche de transport sont gérés par une application (par exemple `sshd` ou `httptunnel`). Aussi le trafic de datagrammes remonte au niveau applicatif pour redescendre au niveau IP, ce qui a l'avantage de pouvoir être mis en œuvre par un utilisateur n'ayant pas nécessairement les droits de l'administrateur[2], mais par contre, outre une consommation supplémentaire de cycles cpu et des changements de contexte inhérents à l'architecture logicielle[3], a l'inconvénient d'être dédié à un seul

[1] " Point to Point Protocol ", lui -même éventuellement encapsulé dans de l'Ethernet (PPPoE RFC 2516) ou de l'ATM (PPPoA pour l'ADSL, RFC 2364)

[2] Pour encapsuler IP dans IP par exemple, il faut pouvoir écrire directement dans IP ce qui nécessite une socket en mode raw et donc un uid 0 à l'exécution

[3] Rappelons que les processus applicatifs standards s'exécutent en mode utilisateur, et que les transferts entre la couche de transport (dans le noyau) et la couche applicative s'effectuent via un jeu de primitives du système, voir la description des sockets de Berkeley page 201

port (par exemple celui d'une application non cryptée comme `pop` au travers une liaison `ssh`)[4].

Encapsuler IP dans IP a l'avantage de rester généraliste du point de vue des applications. Sur la figure ci-dessus le tunnel IP encapsule donc de l'IP dans lui même. Pour les routeurs qui acheminent ce trafic il s'agit de datagrammes IP avec le type 4 (cf le fichier `/etc/protocols` au lieux des types 1 (icmp) 6 (tcp) ou 17 (udp) plus habituels.

2.1 Tunnel IP avec l'interface `gif`

La *figure X.03* illustre l'encapsulation d'IP dans IP grâce à l'usage du " `generic tunnel interface` "[5]. Il s'agit d'un pseudo-device (pas d'interface réel associé, comme cela pourrait être le cas avec une carte réseau) très généraliste, qui permet d'encapsuler de l'IP (version 4 ou 6) dans de l'IP (version 4 ou 6)[6].

Le but de cet exemple de tunnel est de montrer un routage de datagrammes issus d'un réseau non officiellement routé, le 192.168.2.0/24 (RFC 1918), depuis la machine B (IP_B), vers la machine A (IP_A) et au travers d'un réseau routé quelconque.

- Par hypothèse la machine A sait comment router vers le 192.168.2.0/24.
- Le réseau 192.168.249.0/30 (RFC 1918) sert de réseau d'interconnexion entre les deux machines. Concrètement, il s'agit d'attribuer une adresse IP à chacun des pseudo-devices, qui ne soit pas déjà dans l'un des réseaux attachés à chacune des machines.
  Conceptuellement, il serait parfaitement possible d'utiliser, par exemple, des adresses dans le 192.168.2.0/24.
  L'auteur préfère l'usage d'un réseau d'interconnexion qui permet de bien séparer ce qui concerne le tunnel de ce qui utilise le tunnel. De plus, si on souhaite (et c'est une quasi obligation quand on utilise des tunnels) ajouter un filtrage IP sur la machine B, il est beaucoup plus aisé pour la conception des règles de filtrage de considérer l'origine des datagrammes ayant une adresse source dans le 192.168.2.0/24 uniquement derrière le filtre.

Examinons maintenant quelle pourrait être la configuration spécifique à ce tunnel, Sur la machine A :

```
ifconfig create gif0
ifconfig gif0 inet tunnel IP(A) IP(B)
ifconfig gif0 inet 192.168.249.1 192.168.249.2 netmask 0xfffffffc
route add -net 192.168.2.0 192.168.249.2
```

[4] La mise en œuvre d'un tunnel au travers http pour contourner le filtrage en sortie d'un site n'est absolument pas recommandée par l'auteur de ces pages et laissée à la complète responsabilité du lecteur

[5] " `man gif` " sous FreeBSD, NetBSD ou OpenBSD

[6] Nous avons déjà rencontré un tel interface virtuel avec l'interface de " loopback " `lo0` page 72

Puis sur la machine B :

```
ifconfig create gif0
ifconfig gif0 inet tunnel IP(B) IP(A)
ifconfig gif0 inet 192.168.249.2 192.168.254.1 netmask 0xfffffffc
```

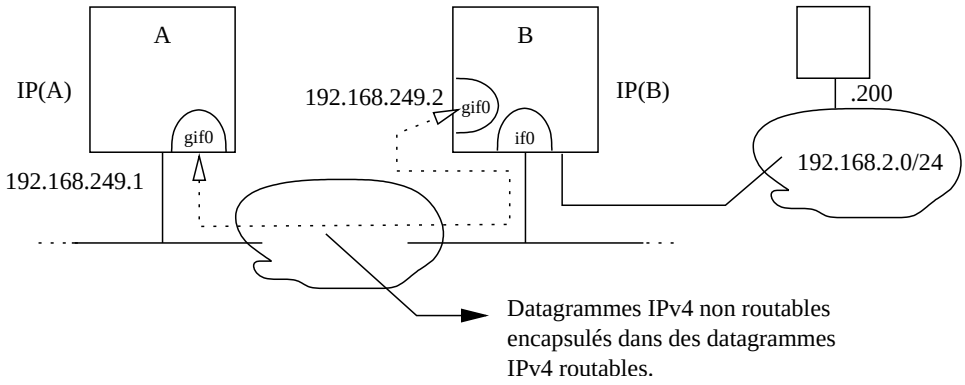


figure X.03 : Tunnel IP - cas concrêt

Notez que la première ligne de configuration précise la source et la destination réelle des datagrammes alors que la deuxième indique l'adresse locale et distante des extrémités du tunnel. C'est une écriture particulière, adaptée au pilote de l'interface `gif0` pour la configuration des tunnels.
Sur la machine B, on peut voir le résultat de la configuration comme ça :

```
$ ifconfig -a
...
gif0: flags=8011<UP,POINTOPOINT,MULTICAST> mtu 1280
    tunnel inet IP(B) --> IP(A)
    inet 192.168.249.2 -> 192.168.249.1 netmask 0xfffffffc
```

Et sur la machine A :

```
$ ifconfig -a
...
gif0: flags=8011<UP,POINTOPOINT,MULTICAST> mtu 1280
    tunnel inet IP(A) --> IP(B)
    inet 192.168.249.1 -> 192.168.249.2 netmask 0xfffffffc
```

Enfin, si on examine[7] sur les interfaces de B le passage de datagrammes d'un `ping`, par exemple, envoyé depuis A vers 192.168.2.200, on relève les en-têtes IP du tableau qui suit.

Ainsi l'observation pratique rejoint la théorie : on retrouve bien sur le l'interface du tunnel (`gif0`) l'en-tête 2, décapsulé de son en-tête 1. Le datagramme est alors disponible au niveau de la pile IP de B pour être routé (routage direct[8] ici) vers 192.168.2.200.

[7] Avec `tcpdump -i if0` puis `tcpdump -i gif0` par exemple
[8] Cf page 63

Sur l'interface `if0`

En-tête 1		En-tête 2
Src	IP_A	192.168.249.1
Dst	IP_B	192.168.2.200
Code	ipencap(4)	icmp

Sur l'interface `gif0`

En-tête	
Src	192.168.249.1
Dst	192.168.2.200
Code	icmp

Remarques :

Attention, les routeurs filtrants doivent être spécialement configurés pour laisser passer ce type de trafic[9]. Les " core gateway " le laisse passer sans problème.

Il est intéressant de noter que le déploiement d'IPv6 est d'abord basé sur l'encapsulation de trames de la version 6 dans des trames de la version 4, en attendant que tous les routeurs soient capables de router IPv6 nativement.

Enfin pour conclure, est-il nécessaire de préciser que même encapsulés dans IP, nos datagrammes n'en sont pas moins lisibles par des yeux indiscrêts ? Pour s'en prémunir il nous faut examiner une technologie complétementaire. . .Dans le paragraphe suivant !

[9]Pour un routeur de type cisco qui protégerait la machine B, il faudrait ajouter des règles du genre "`permit ipinip host IPB IPA`" et "`permit ipinip host IPA IPB`"

2.2 IPsec et VPN

IPsec est un protocole de sécurité inclus dans la couche IP elle-même. Il est défini dans la RFC 2401. Ce paragraphe aborde brièvement la question du point de vue du protocole et de sa place dans la pile Arpa, tout en laissant volontairement dans un certain flou les aspects liés à la cryptographie, clairement absents des objectifs de ce cours[10].

IPsec dans quel but ?

IPsec est un point de passage obligé pour tout administrateur de réseau qui souhaite mettre en place une politique de sécurité. D'ailleurs, pour faire le lien avec le paragraphe qui précède, précisons qu'IPsec encapsulé dans IP (formellement, un tunnel) porte un nom, le VPN (“ Virtual Private Network ”) ! Nous avons examiné comment un tunnel accroît l'étendue d'un réseau au travers d'autres réseaux. Ajouter Ipsec introduit, entre autres, une dimension de sécurité très utilisée pour relier des machines - ou des réseaux - physiquement localisés n'importe où il y a un accès IP, en réseaux virtuels sécurisés ! C'est pour cette raison qu'Ipsec est un artefact incontournable de la panoplie sécuritaire sur les réseaux.

Nous aurions pu conclure le chapitre sur IP page 45 par cette constatation que le protocole IP est lisible par tout le monde y compris par les indiscrêts et que quasiment n'importe quel “ bricoleur ” peut forger de faux datagrammes (“ fake datagram ”) pour empoisonner un réseau, voire détourner les services d'une machine. Ainsi, tout ce qui renforce la sécurité IP est une bonne chose, surtout à l'heure des réseaux “ wifi ” dont les limites de portée ne sont pas maîtrisables.

IPsec renforce la sécurité d'IP sur plusieurs points :

Confidentialité Les données d'IP (protocole de transport et données applicatives) sont cryptées, donc normalement non inspectables avec tout outil d'analyse de réseau.

Authentification La source du datagramme ne peut être qu'un seul émetteur, et non un intermédiaire non prévu.

Intégrité La totalité des données est protégée par une somme de contrôle (checksum), travail normalement dévolu à la couche de transport mais qui au niveau d'IP permet d'écarter tout datagramme qui aurait été modifié pendant son transport.

Dispositif “ anti-rejeux ” pour éviter les attaques du type “ man-in-the-middle ” consistants à capturer un ou plusieurs datagrammes (cryptés) dans le but de les envoyer à nouveau pour bénéficier du même effet produit que l'envoi initial.

[10] Les RFCs données page 197 sont le bon point de départ pour se documenter sur les aspects cryptographiques d'IPsec

IPsec en résumé

Ipsec (RFC 2401) est un assemblage de quatre protocoles :

ESP (“ Encapsulating Security Payload ”) est défini par la RFC 2406. Il assure la **confidentialité** par l'usage d'algorithmes de cryptage comme “ DES ” (RFC 2405) , “ 3DES ” (RFC 2451), “ CAST-128 ” (RFC 2144) ou encore “ blowfish ” (RFC 2451), la liste n'est pas exhaustive... Il faut juste noter qu'il s'agit d'algorithmes basés sur l'existence un secret partagé (manuellement dans un fichier ou crée dynamiquement avec IKE, voir plus bas) entre les parties qui échangent des messages, et non sur l'échange d'une clef publique. Cette remarque a un impact sur la manière avec laquelle on doit les configurer !

AH (“ Authentication Header ”) est défini par la RFC 2402. Il assure l'**authentification**, c'est à dire qu'il cherche à certifier que les deux couches IP qui dialoguent sont bien celles qu'elles prétendent être, puis l'**intégrité** des données par le calcul d'un checksum. Il faut noter que ce dernier travail empiète largement sur les attributions de la couche de transport mais se justifie compte-tenu des exigences inhérentes au fonctionnement d'IPsec.

IPcomp (“ IP payload compression ”) sert à compresser les données avant de les crypter. Son action est rendue nécessaire pour tenter de compenser la perte de la place occupée par les en-têtes ajoutés. Bien entendu IPcomp peut être utilisé seul.

IKE (“ Internet Key Exchange ”) est défini par la RFC 2409. Ce protocole n'est pas formellement indispensable au bon fonctionnement d'IPsec mais lui apporte un mécanisme d'échange de clefs, au démarrage des échanges et au cours du temps. Ainsi la clef de chiffrement n'est plus définie de manière statique dans un fichier mais change continuellement au cours du temps, ce qui est meilleur du point de vue de la sécurité.

Du point de vue de l'administration système et réseau, la mise en place de ce protocole passe par l'usage d'un *daemon*[11], par exemple `racoon`, et par une ouverture de port UDP (`isakmp`/500) supplémentaire dans le filtrage du réseau. Une négociation décrite dans la RFC 2409 se déroule entre les hôtes qui doivent dialoguer, ce qui entraine une certaine latence au début des échanges.

Les 32 bits de l'adresse IP de destination[12] permettent théoriquement d'exprimer un adressage de type unicast ou multicast ou broadcast. Si ces cas de figures sont tous théoriquement possibles, les implémentations d'IPsec ne supportent que l'unicast. La conséquence est importante sur le déploiement d'IPsec, il est effectué “ point à point ” plutôt que généralisé pour tout un réseau.

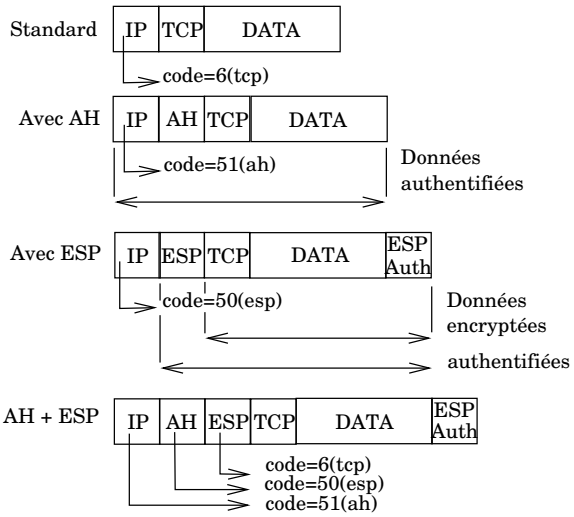
[11] Voir page 252 pour le fonctionnement des *daemons*
[12] Révision possible page 33

Ce travail est inclus dans ce qui est nommé “ politique de sécurité ” dans la RFC 2401.

Pour AH comme pour ESP, l'ajout de données vient se placer entre l'en-tête IP et les données. Les deux protocoles peuvent être utilisés ensembles ou séparement, c'est un choix qui relève de la politique de sécurité. Pour en tenir compte, la négociation qui a lieu lors de l'établissement d'IPsec repose sur ce que la RFC appelle des SA (“ Security Association ”).

Une SA est formellement un triplet unique constitué d'un index unique , le SPI

figure X.04 : En-têtes d'IP-sec



(“ Security Parameter Index ”) sorte de numéro d'identification IP supplémentaire[13] inclus dans l'en-tête AH ou ESP, de l'adresse IP du destinataire et du protocole ESP ou d'AH. Si les deux protocoles doivent être utilisés, il faut négocier deux SAs.

Comment utiliser IPsec ?

Aux deux protocoles AH et ESP, s'ajoutent deux manières d'utiliser IPsec, soit directement d'une machine à une autre, on parle alors de “ mode transport ” soit encore en l'encapsulant dans un tunnel comme vu au paragraphe 2 page 178 et on parle de “ mode tunnel ”, plus connu sous le vocable “ VPN ”.

La RFC 2401 indique que toute implémentation se réclamant d'IPsec doit supporter les 4 associations qui suivent.

La sécurité entre deux hôtes qui supporte IPsec, au travers l'Internet, en mode transport ou en mode tunnel. Les datagrammes peuvent avoir les structures suivantes :

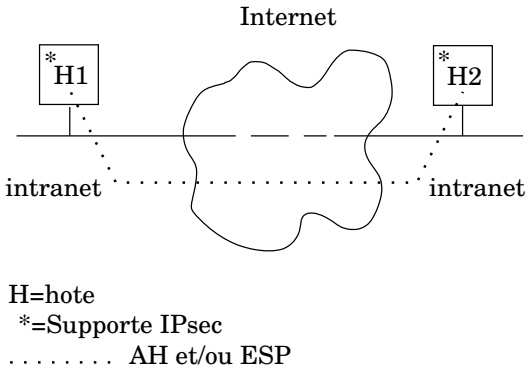
Mode transport

```
[IP1][AH][Transport][Data]
[IP1][ESP][Transport][Data]
[IP1][AH][ESP][Transport][Data]
```

Mode tunnel

```
[IP2][AH][IP1][Transport][Data]
[IP2][ESP][IP1][Transport][Data]
```

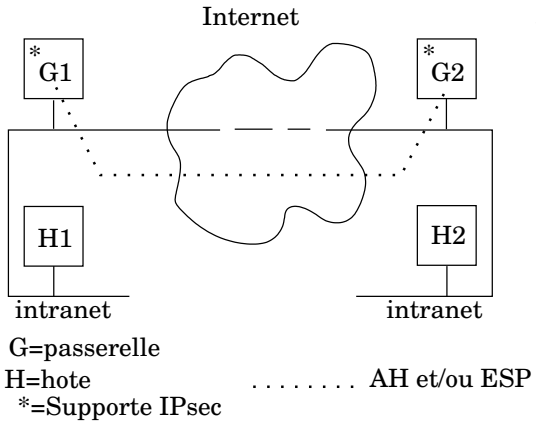
figure X.05 : Association 1



[13] Voir page 48

Remarque : En mode tunnel pour ce premier cas il n'y a pas d'obligation du support d'AH et ESP simultanément. Quand ils sont appliqués tous les deux, il faut d'abord appliquer ESP, puis AH aux datagrammes.

figure X.06 : Association 2



Le mode tunnel est le seul requis ici. Nous avons donc une structure de datagramme qui a ces formes possibles :

Mode tunnel

```
[IP2][AH][IP1][Transport][Data]
[IP2][ESP][IP1][Transport][Data]
```

C'est la combinaison des deux premiers cas, on ajoute la sécurité entre les hôtes terminaux à celle déjà apportée par les routeurs.

La propagation du trafic de type ISAKMP (protocole IKE) au travers les routeurs est un plus.

figure X.07 : Association 3

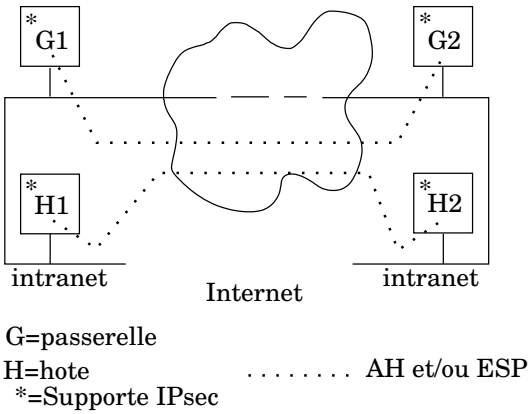
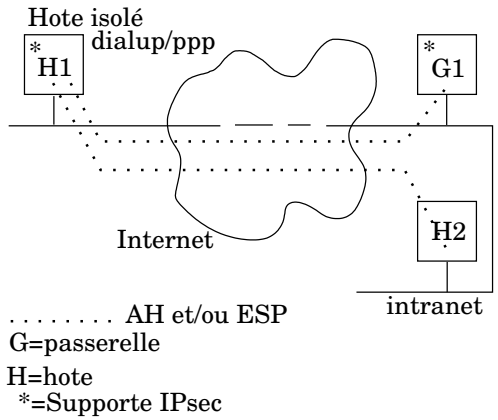


figure X.08 : Association 4



Ce dernier cas est celui d'un poste isolé qui se raccorde par exemple à l'intranet de son entreprise via un modem ou un accès IP non sûr, et qui utilise un protocole non crypté comme `AppleTalk`, ou `PoP`, par exemple.

Le mode tunnel est seul requis entre l'hôte H1 et la passerelle G1. Ensuite, entre H1 et H2 on revient au premier cas.

Implémentation d'IPsec

L'implémentation d'IPsec sur les machines FreeBSD et NetBSd est issue du projet KAME[14] et est ainsi fortement lié au développement de la pile IPv6.

Les protocoles AH, ESP et IPcomp sont inclus dans le noyau directement. La gestion des clefs s'effectue via une commande externe, `setkey` qui pilote une table de gestion des clefs située elle aussi dans le noyau, grâce à des socket de type `PF_KEY`

Les clefs sont soit placées de manière semi-définitive dans le fichier de configuration d'ipsec lui-même (par exemple `/etc/ipsec.conf` soit confiée aux bons soins d'un programme externe qui se charge de les crée et de les propager à l'aide du protocole IKE. Quelques `daemons` savent faire cela, notamment `racoon` du projet KAME.

Si nous reconsidérons la *figure X.03* les machines A et B jouent le rôle des passerelles G1 et G2 de la *figure X.06* (association 2). Les fichiers de configuration IPsec pourraient être :

Sur la machine A

```
spdadd IP(A) IP(B) any -P out ipsec esp/tunnel/192.168.249.1-192.168.249.2/require;
spdadd IP(B) IP(A) any -P in  ipsec esp/tunnel/192.168.249.2-192.168.249.1/require;
```

`spdadd` est une instruction de la commande `setkey`. Il faut définir sa politique de sécurité, c'est à dire ce que l'on souhaite en entrée (`in`), en sortie (`out`) puis un choix de protocole (esp, ah, ipcomp), un mode (tunnel ici) avec l'entrée et la sortie du tunnel, enfin un niveau d'usage (`require` ici indique que tous échanges doivent utiliser IPsec).

Sur la machine B

```
spdadd IP(B) IP(A) any -P out ipsec esp/tunnel/192.168.249.2-192.168.249.1/require;
spdadd IP(A) IP(B) any -P in  ipsec esp/tunnel/192.168.249.1-192.168.249.2/require;
```

La clef de cryptage ne figure pas dans ce fichier car l'exemple utilise IKE pour cela, à l'aide de l'outil `racoon`.

Enfin, un excellent document de configuration se trouve sur le site du projet NetBSD :

`http ://www.netbsd.org/Documentation/network/ipsec/`

[14] `http ://www.kame.net/`

3 Proxy

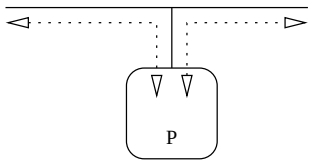


figure X.09 : Proxy

Le propos d'un proxy est de concentrer le trafic réseau via une seule machine pour toute une variété de protocoles (telnet, http, smtp, ...). Il existe des proxy spécialisés sur tel ou tel protocole, qui effectuent donc des tâches potentiellement très complexes (par exemple `squid` pour `http`) ou très généraux et donc moins performants sur chaque protocole (cf `nat` au paragraphe suivant).

Tout le trafic réseau qui passe par un proxy s'y arrête pour en repartir. Les conditions de ce " rebond " peuvent être paramètrées par des règles d'accès, ce qui en fait un élément utile en sécurité des réseaux (voir la RFC 1919).
Section en chantier, précisions en cours...

4 Translation d'adresses

La pénurie d'adresses IP est à l'origine du besoin de translation des adresses. Son principe se trouve décrit dans la RFC 1631.

Le propos d'un automate de translation d'adresses est de convertir les adresses IP à la volée au passage d'un interface.

Dans une configuration classique un tel automate isole un réseau comportant des adresses IP non officielles (RFC 1918) d'un autre comportant des adresses officielles et donc potentiellement routées sur l'Internet.

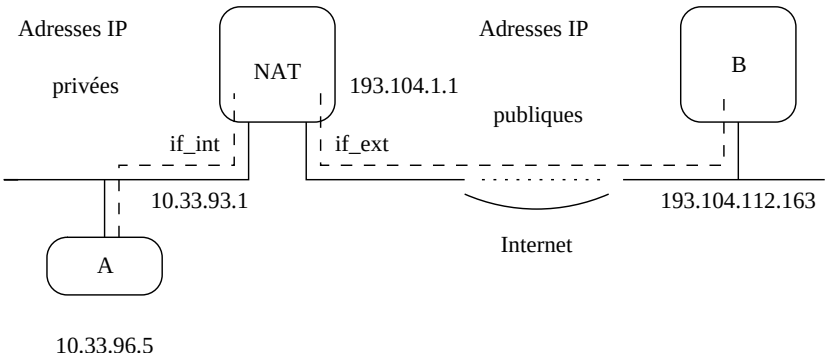


figure X.10 : Machine NAT en routeur

Dans la *figure 10*, la machine " NAT " est par hypothèse configurée en routeur. On suppose également que pour la machine A c'est le routeur par défaut pour atteindre la machine B.

Pour la machine A

- La machine A s'adresse directement à la machine `193.104.112.163` en utilisant son routeur par défaut.
- L'utilisateur de la machine A " voit " la connexion soit la forme :

{tcp, IP Hôte A, port A, IP Hôte B, port B}

Pour la machine B

- La machine B voit une connexion arriver en provenance de " NAT ".
- La machine B n'a pas connaissance de la machine A, elle dialogue avec la machine NAT selon :

{tcp, IP Hôte B, port B, IP Hôte NAT, port A}

Pour la machine NAT

- La machine NAT a connaissance des 2 réseaux, elle translate dynamiquement les adresses dans les deux sens.
- La machine NAT fait le travail d'un proxy transparent puisque chaque connexion s'y arrête puis en repart sans configuration particulière de la part de l'administrateur de A ou de B.
- La translation (ou "IP masquerading") s'effectue dynamiquement selon l'adresse demandée.
- La translation d'adresse s'effectue pour les datagrammes qui traversent l'interface `if_ext`. Le dialogue entre cette machine et les autres machines du réseau " privé ", via l'interface `if_int` ne fait pas l'objet d'une translation d'adresse.
- La situation de la machine A est plutôt celle d'un poste client car non vu de l'extérieur de son réseau. Être serveur est toutefois possible comme il l'est expliqué avec l'usage de `natd` au paragraphe suivant.

Les routeurs modernes ont la fonction de translation d'adresses incluse dans leurs fonctionnalités standards.

4.1 Cas de `NATD`

`Natd` est l'outil logiciel libre bien connu des administrateurs réseaux. Il fonctionne selon le modèle de la *figure 11*[15].

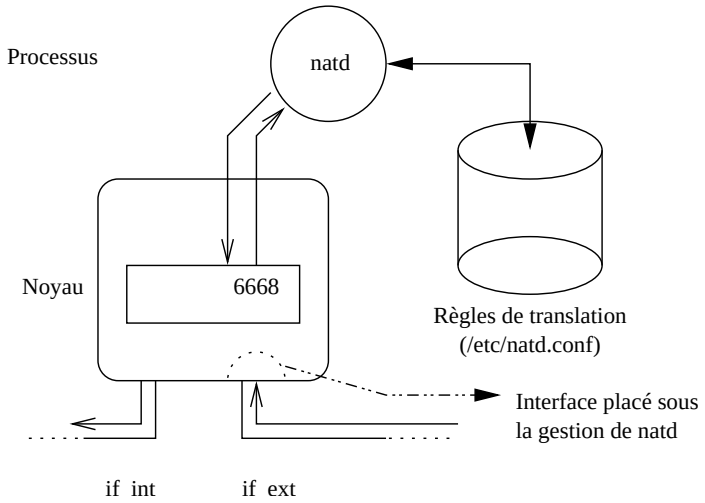


figure 11 : Natd sous FreeBSD

Le processus `natd` ouvre une socket en mode raw pour communiquer directement avec la couche IP :

```
divertInOut = socket (PF_INET, SOCK_RAW, IPPROTO_DIVERT) ;
```

Le noyau IP, muni du mécanisme adéquat [16] redirige tout le trafic entrant et sortant d'un interface réseau, vers un numéro de port convenu à la configuration, par exemple le port 6668, à ajouter dans `/etc/services` :

```
natd 6668/divert # Network Address Translation socket
```

`Natd` lit tous les datagrammes et ne retient pour traitements que ceux qui sont à destination du port dédié.

Compte tenu du fichier de configuration, les adresses IP des datagrammes ainsi que les numéros de ports sont reécrits et reinjectés dans le noyau IP qui les traite comme d'autres datagrammes (routage, filtrage...).

`Natd` convertit les adresses IP à la volée, un datagramme entrant peut ainsi par jeux de reécriture être redirigé sur une toute autre machine, sans que son émetteur puisse le soupçonner !

On distingue deux attitudes, soit tout le flux entrant est redirigé sur une seule machine, soit il est analysé plus finement et la redirection (résultat de la reécriture) est effectuée en fonction du port, donc du service demandé.

La littérature appelle ce cas le "static nat". À l'insu des utilisateurs de la machine "NAT" du réseau publique, tout le trafic IP (c'est ainsi qu'il faut comprendre l'adresse IP 0.0.0.0) est renvoyé sur la machine "S", et celle-ci n'est pas "visible" du réseau public.

[15] Les implémentation commerciales que l'on trouve dans les routeurs, si elles ne se configurent pas de la même manière, ont des propriétés très voisines en fonctionnement

[16] Par exemple pour FreeBSD il faut ajoute l'option `IPDIVERT` dans la configuration du noyau

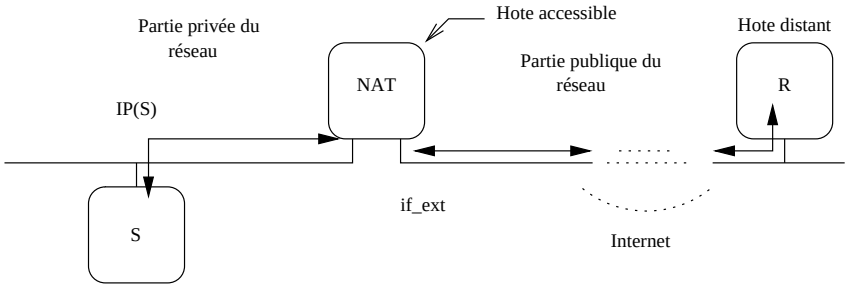


figure X.12 : "Static Nat"

La configuration du `daemon` pourrait être :

```
natd -interface if_ext -redirect_address IP(S) 0.0.0.0
```

La *figure 13* nous montre un trafic éclaté en fonction du service demandé, ce qui permet une gestion beaucoup fine des ressources du réseau.

Une demande de connexion de l'hôte distant R sur la machine NAT et au port 80 va être réacheminée vers la machine interne HTTP et sur le port que l'on souhaite, par exemple 8080.

Même remarque pour les deux autres services présentés.

La machine HTTP voit la connexion en provenance de la machine R sous sa forme exacte :

{tcp, IP Hôte R, Port R, IP Hôte HTTP, 8080}

La machine R ne voit que la partie "publique" de la connexion :

{tcp, IP Hôte R, Port R, IP Hôte NAT, 80}

La configuration de NAT pourrait ressembler à :

```
#
# Configuration multiservices
#
redirect_port tcp http:8080 80
redirect_port tcp smtp:25 25
redirect_port tcp dns:domain domain
redirect_port udp dns:domain domain
```

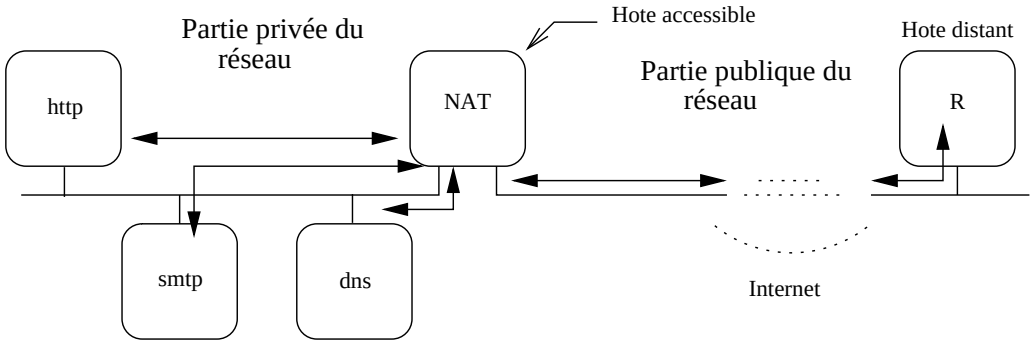


figure 13 : Configuration multiservices

5 Filtrage IP

Le propos du filtrage IP est d'appliquer des règles de filtrage à un flux de datagrammes IP afin de prendre une décision qui est le plus souvent binaire : laisser passer ou ne pas laisser passer avec en option de conserver une trace de passage (des logs).

Par son usage on cherche à protéger un site ou une machine d'un flux de datagrammes pour lesquels on suspecte que tous ne sont pas envoyés par des utilisateurs bienveillants. Le filtre s'efforce d'éliminer le trafic indésirable à partir de considérations **à priori**, mais il ne représente pas la panacée en matière de sécurité sur les réseaux, autrement dit il ne faut pas penser qu'un filtre IP suffit à règler tous les problèmes de sécurité d'un site ou d'un hôte.

En effet, à partir du moment où le filtre laisse passer certains datagrammes, même **à priori** innocents, une porte est ouverte au détournement de l'usage initial du service offert. Dans ces conditions il faut se rendre à l'évidence : il n'y a pas de sécurité absolue sur le réseau[17] !

Dans la littérature, un routeur filtrant est nommé " FireWall ", qu'il faut traduire en " pare-feux ".

Le filtrage IP est une caractéristique essentielle de tout routeur digne de ce nom !

Il est aussi possible de faire du filtrage IP avec les Unix libres, c'est cette approche qui est choisie dans les paragraphes qui suivent parcequ'accessible à toutes les bourses...

Si les détails de mise en œuvre diffèrent d'une implémentation à une autre, le fond du problème reste le même. L'implémentation choisie ici est `ipfw`, le filtre natif de FreeBSD[18]. Il existe d'autres filtre, notamment `ipf`, encore une fois le principe reste toujours le même.

5.1 Le cas d'`ipfw` de FreeBSD

Le filtre IP en question est implémenté dans le noyau, il est activé dès lors que l'options `IPFIREWALL` est ajoutée dans le noyau. On peut également y adjoindre l'option `IPFIREWALL_VERBOSE` pour le rendre bavard[19], ce qu'apprécient par dessus tout les administrateurs réseaux, soucieux de tout connaître sur la nature du trafic...

Le filtre est un module du noyau, chargé au démarrage, et qui se paramètre à l'aide de la commande `ipfw`. Celle-ci est utilisée dans les scripts de démarrage pour dicter au noyau les règles de filtrage, lues dans un fichier nommé par defaut `/etc/rc.firewall`. les scripts de démarrage pour dicter au noyau les règles de filtrage,

Établir des règles de filtrage IP sous-entend avoir une connaissance exhaustive de tous les éléments qui s'y rattachent :

[17] Les seuls réseaux sûrs sont isolés du monde extérieur dans une cage de Faraday...

[18] `http ://www.freebsd.org`

[19] Via `syslogd`

– Nom des interfaces réseaux impliquées
– Protocoles réseaux employés (tcp, udp, icmp,...)
– Protocoles applicatifs autorisés (smtp, domain, http...)
– Adresses IP, numéro de ports, masque de sous-réseaux
– Sens du trafic par rapport aux interfaces ci-dessus

Il est assez aisé de mettre en place un filtrage simple, par contre cette opération peut devenir complexe dès lors qu'on utilise des protocoles applicatifs compliqués, mettant en jeux une stratégie d'utilisation des ports et des adresses non triviale.

Considérons un site simple, comme celui de la *figure X.14*.

La machine "C" accède depuis l'extérieur à la machine "S", protégée par le filtrage IP activé sur la machine "R", qui agit donc comme un routeur filtrant.

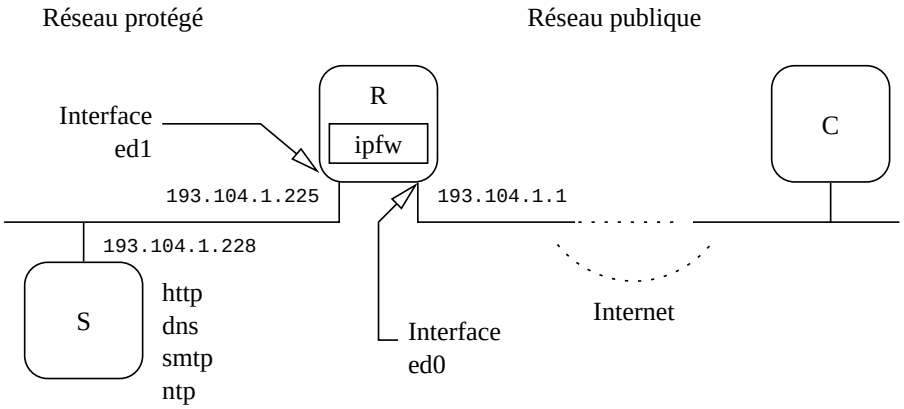


figure X.14

Adaptons-y les règles du modèle de base, extraites du fichier `/etc/rc.firewall` de la configuration standard d'une machine FreeBSD récente (c'est un script shell). L'examen de ces règles nous permet de découvrir la nature du trafic autorisé ou non.

```
# Interface externe
oif="ed0"
onet="193.104.1.0"
omask="255.255.255.224"
oip="193.104.1.1"

# Interface interne
iif="ed1"
inet="193.104.1.224"
imask="255.255.255.224"
iip="193.104.1.225"

# Ne pas laisser passer le ‘‘spoofing’’
ipfw add deny all from ${inet}:${imask} to any in via ${oif}
ipfw add deny all from ${onet}:${omask} to any in via ${iif}

# Ne pas router les adresses de la RFC1918
ipfw add deny all from 192.168.0.0:255.255.0.0 to any via ${oif}
ipfw add deny all from any to 192.168.0.0:255.255.0.0 via ${oif}
ipfw add deny all from 172.16.0.0:255.240.0.0 to any via ${oif}
ipfw add deny all from any to 172.16.0.0:255.240.0.0 via ${oif}
ipfw add deny all from 10.0.0.0:255.0.0.0 to any via ${oif}
ipfw add deny all from any to 10.0.0.0:255.0.0.0 via ${oif}

# Laisser passer les connexions TCP existantes
ipfw add pass tcp from any to any established

# Permettre l’arrivee du courrier SMTP
ipfw add pass tcp from any to 193.104.1.228 25 setup

# Permettre l’acces au serveur HTTP
ipfw add pass tcp from any to 193.104.1.228 80 setup

# Rejetter et faire des logs de toutes les autres demandes de connexion
ipfw add deny log tcp from any to any in via ${oif} setup

# Permettre l’etablissement des autres connexions (via $iif).
ipfw add pass tcp from any to any setup

# Permettre le trafic UDP/DOMAIN vers/depuis les serveurs DNS externes
ipfw add pass udp from any 53 to any 53

# Permettre le trafic NTP vers/depuis les serveurs de dates
ipfw add pass udp from any 123 to any 123

# Permettre le passage de tous les paquets ICMP
ipfw allow icmp from any to any

# Tout ce qui n’est pas explicitement autorise est
# implicitement interdit ;-)
ipfw deny ip from any to any
```

Quelques considérations :

- Les règles sont parcourues de la première à la dernière, si aucune convient, l'action par défaut consiste à bloquer le trafic (peut être changée).
- Dès qu'une règle convient, elle est appliquée et le filtrage s'arrête.
- Le filtrage IP consomme des ressources cpu, donc pour améliorer les performances il vaut mieux placer en tête de liste les règles employées le plus couramment.

Il faut remarquer que la machine 193.104.1.228 est visible depuis l'extérieure et utilise une adresse officiellement routée.

Une tentative d'accès sur un service non autorisé se traduit par un message d'erreur (`syslogd`). Par exemple supposons que l'utilisateur de la station "C" exécute la commande suivante :

telnet 193.104.1.228

Il va obtenir le message :

```
telnet : Unable to connect to remote host : Connection timed out
```

Tandis que l'administrateur du réseau 193.104.1.0 va constater la tentative d'intrusion par le message :

```
ipfw : 3310 Deny TCP Adr.IP Hôte C :2735 193.104.1.228 :23 in via
                                ed0
```

Par contre, si l'intrusion consiste à détourner l'usage du service SMTP, l'administrateur du réseau 193.104.1.0 ne verra rien par ce biais puisque l'accès SMTP est autorisé sur la machine 193.104.1.228[20]

6 Exemple complet

Dans cette partie nous examinons le cas de la configuration de la *figure X.15*, synthèse des figures X.12, X.13 et X.14. C'est une configuration très employée du fait de la distribution parcimonieuse d'adresses IP par les fournisseurs d'accès.

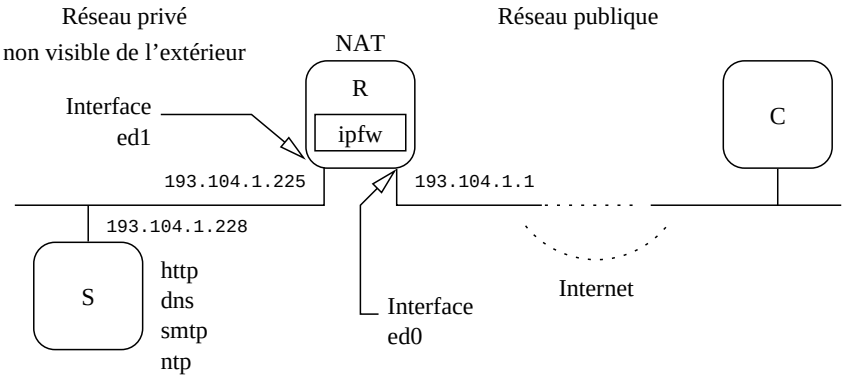


figure X.15 : Translation d'adresse et filtrage IP

Le propos est de mettre en place un routeur filtrant effectuant en plus la translation d'adresses IP. La juxtaposition des deux services induit peu de changements dans la configuration des règles de filtrage.

[20]Toute ressemblance avec la configuration réelle de ce réseau ne peut être que fortuite

Commençons par les règles de filtrage :

```
# Interface externe
oif="ed0"
onet="193.104.1.0"
omask="255.255.255.224"
oip="193.104.1.1"

# Interface interne
iif="ed1"
inet="193.104.1.224
imask="255.255.255.224"
iip="193.104.1.225"
#
# Usage de 'natd' pour transformer tout ce qui passe sur l'interface
# "ed0" donc le subnet public.
ipfw add divert 8668 all from any to any via ${oif}

# Ne pas laisser passer le ''spoofing''
ipfw add deny all from ${inet}:${imask} to any in via ${oif}
ipfw add deny all from ${onet}:${omask} to any in via ${iif}

# Ne pas router les adresses de la RFC1918
ipfw add deny all from 192.168.0.0:255.255.0.0 to any via ${oif}
ipfw add deny all from any to 192.168.0.0:255.255.0.0 via ${oif}
ipfw add deny all from 172.16.0.0:255.240.0.0 to any via ${oif}
ipfw add deny all from any to 172.16.0.0:255.240.0.0 via ${oif}
ipfw add deny all from 10.0.0.0:255.0.0.0 to any via ${oif}
ipfw add deny all from any to 10.0.0.0:255.0.0.0 via ${oif}

# Laisser passer les connexions TCP existantes
ipfw add pass tcp from any to any established

# Permettre l'arrivee du courrier SMTP
ipfw add pass tcp from any to 193.104.1.228 25 setup

# Permettre le trafic TCP/DOMAIN
ipfw add pass tcp from any to 193.104.1.228 53 setup

# Permettre l'acces au serveur HTTP
ipfw add pass tcp from any to 193.104.1.228 80 setup

# Rejetter et faire des logs de tout autre demande de connexion
ipfw add deny log tcp from any to any in via ${oif} setup

# Permettre l'etablissement des autres connexions (via $iif).
ipfw add pass tcp from any to any setup

# Permettre le trafic UDP/DOMAIN vers/depuis les forwarders
ipfw add pass udp from any 53 to 193.104.1.228 53
ipfw add pass udp from 193.104.1.228 53 to any 53

# Permettre le trafic DTP/NTP
ipfw add pass udp from any 123 to 193.104.1.228 123
ipfw add pass udp from 193.14.1.228 123 to any 123

# Permettre le passage des paquets ICMP (ping, traceroute...)
ipfw add pass icmp from any to any via ${oif} icmptype 0,3,8,11
ipfw add pass udp from any 32768-65535 to any 32768-65535 out xmit ${oif}
ipfw add log icmp from any to any in recv ${oif}
ipfw add pass icmp from any to any via ${iif}

# Tout ce qui n'est pas explicitement autorise est
# implicitement interdit ;-)
```

Dans le principe l'hôte 193.104.1.228 n'est plus visible de l'extérieur, les services sont en apparence hébergés par la machine "R" qui se charge de

re-router les datagrammes en modifiant dynamiquement l'adresse de destination.

Dans l'ordre des opérations, la translation d'adresses est effectuée avant le filtrage IP. Ce sont donc des adresses IP modifiées qui sont introduites dans les règles de filtrage !

Terminons avec la configuration de `natd`. Voici le contenu du fichier `/etc/natd.conf` pour cette situation :

```
redirect_port tcp 193.104.1.228:80 80
redirect_port tcp 193.104.1.228:25 25
redirect_port tcp 193.104.1.228:53 53
redirect_port udp 193.104.1.228:53 53
redirect_port udp 193.104.1.228:123 123
```

Où l'on s'apperçoit que la configuration n'a pratiquement pas changé fondamentalement ormis par l'introduction de la règle :

ipfw add divert 6668 all from any to any via ${oif}

Qui indique au filtre que tout ce qui provient de l'interface "oif" est à lire sur le port 6668, donc a déjà subit la translation d'adresse avant d'être soumis au filtrage IP. Ainsi une demande de connexion sur le port 25 de la machine 193.104.1.1 sera transformée en une demande de connexion sur le port 25 de la machine 193.104.1.228, qui est autorisé.

Pour l'utilisateur de la station "C" la machine 193.104.1.228 n'est plus visible, seule la machine d'adresse 193.104.1.1 semble cumuler tous les services !

7 Bibliographie

RFC 1631 “ The IP Network Address Translator (NAT). ” K. Egevang & P. Francis. May 1994. (Format : TXT=22714 bytes) (Status : INFORMATIONAL)

RFC 1918 “ Address Allocation for Private Internets. ” Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J.d̃e Groot & E. Lear. February 1996. (Format : TXT=22270 bytes) (Obsoletes RFC1627, RFC1597) (Also BCP0005) (Status : BEST CURRENT PRACTICE)

RFC 1825 “ Security Architecture for the Internet Protocol. ” R. Atkinson. August 1995. (Format : TXT=56772 bytes) (Obsoleted by RFC2401) (Status : PROPOSED STANDARD)

RFC 2364 “ PPP Over AAL5. ” G. Gross, M. Kaycee, A. Li, A. Malis, J. Stephens. July 1998. (Format : TXT=23539 bytes) (Status : PROPOSED STANDARD)

RFC 2401 “ Security Architecture for the Internet Protocol. ” S. Kent, R. Atkinson. November 1998. (Format : TXT=168162 bytes) (Obsoletes RFC1825) (Updated by RFC3168) (Status : PROPOSED STANDARD)

RFC 2402 “ IP Authentication Header. ” S. Kent, R. Atkinson. November 1998. (Format : TXT=52831 bytes) (Obsoletes RFC1826) (Status : PROPOSED STANDARD)

RFC 2406 “ IP Encapsulating Security Payload (ESP). ” S. Kent, R. Atkinson. November 1998. (Format : TXT=54202 bytes) (Obsoletes RFC1827) (Status : PROPOSED STANDARD)

RFC 2409 “ The Internet Key Exchange (IKE). ” D. Harkins, D. Carrel. November 1998. (Format : TXT=94949 bytes) (Status : PROPOSED STANDARD)

RFC 2516 “ A Method for Transmitting PPP Over Ethernet (PPPoE). ” L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, R. Wheeler. February 1999. (Format : TXT=32537 bytes) (Status : INFORMATIONAL)

RFC 3168 “ The Addition of Explicit Congestion Notification (ECN) to IP. ” K. Ramakrishnan, S. Floyd, D. Black. September 2001. (Format : TXT=170966 bytes) (Obsoletes RFC2481) (Updates RFC2474, RFC2401, RFC0793) (Status : PROPOSED STANDARD)

RFC 1919 “Classical versus Transparent IP Proxies.” M. Chatel. March 1996. (Format : TXT=87374 bytes) (Status : INFORMATIONAL)

Sans oublier :

- W. Richard Stevens - TCP/IP Illustrated, Volume 1 - The protocols - Addison-Wesley
- “Firewalls and Internet Security” - William R. Cheswick, Steven M. Bellovin - Addison-Wesley 1994.
- “Building Internet Firewalls” - D. Brent Chapman and Elizabeth D. Zwicky - O´Reilly - 1995. Steven M. Bellovin - Addison-Wesley 1994.

Troisième partie

Programmation des sockets de Berkeley

Chapitre XI

Généralités sur les sockets de Berkeley

1 Généralités

La version BSD 4.1c d'Unix pour VAX, en 1982, a été la première à inclure TCP/IP dans le noyau du système d'exploitation et à proposer une interface de programmation de ces protocoles : les sockets[1].

Les sockets sont ce que l'on appelle une API ("Application Program Interface") c'est à dire une interface entre les programmes d'applications et la couche transport, par exemple TCP ou UDP. Néanmoins les sockets ne sont pas liées à TCP/IP et peuvent utiliser d'autres protocoles comme AppleTalk, Xérox XNS, etc...

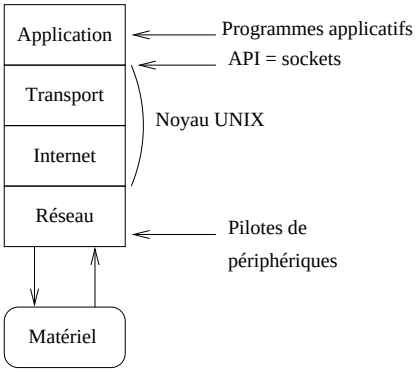


figure XI.01

Les deux principales API pour Unix sont les sockets Berkeley et les TLI System V. Ces deux interfaces ont été développées en C.

Les fonctionnalités des sockets que nous allons décrire, sont celles apparues à partir de la version 4.3 de BSD Unix, en 1986. Il faut noter que les

[1]Pour un historique très intéressant de cette période on pourra consulter `http ://www.oreillynet.com/pub/a/network/2000/03/17/bsd.html`

constructeurs de stations de travail comme HP, SUN[2], IBM, SGI, ont adopté ces sockets, ainsi sont-elles devenues un standard de fait et une justification de leur étude.

Pour conforter ce point de vue il n'est pas sans intérêt d'ajouter que toutes les applications majeures (`named, dhcpd, sendmail, ftpd`, serveurs `httpd`,...) de l'Internet, et dont l'utilisateur peut lire le code source, sont programmées avec de telles sockets.

Enfin, et avant d'entrer dans le vif du sujet, le schéma ci-dessous rappelle les relations entre pile *ARPA*, N° de port et processus.

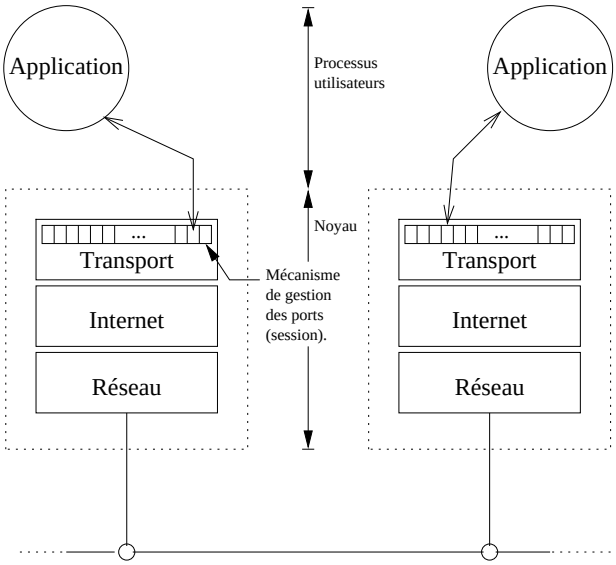


figure XI.02

2 Présentation des sockets

Les créateurs des sockets ont essayé au maximum de conserver la sémantique des primitives systèmes d'entrées/sorties sur fichiers comme `open`, `read`, `write`, et `close`. Cependant, les mécanismes propres aux opérations sur les réseaux les ont conduits à développer des primitives complémentaires.

Quand un processus ouvre un fichier (`open`), le système place un pointeur sur les structures internes de données correspondantes dans la table des descripteurs ouverts de ce processus et renvoie l'indice utilisé dans cette table. Par la suite, l'utilisateur manipule ce fichier uniquement par l'intermédiaire de l'indice, aussi nommé descripteur de fichier.

Comme pour un fichier, chaque socket active est identifiée par un petit entier appelé descripteur de socket. Unix place ce descripteur dans la même table que les descripteurs de fichiers, ainsi une application ne peut-elle pas avoir un descripteur de fichier et un descripteur de socket identique.

[2]Les applications de ce constructeur utilisent les TLI, par contre il est possible d'utiliser les sockets dans toutes les contributions locales

Pour créer une socket, une application utilisera la primitive `socket` et non `open`, pour les raisons que nous allons examiner. En effet, il serait très agréable si cette interface avec le réseau conservait la sémantique des primitives de gestion du système de fichiers sous Unix, malheureusement les entrées/sorties sur réseau mettent en jeux plus de mécanismes que les entrées/sorties sur un système de fichiers.

Par exemple, il faut considérer les points suivants :

- Dans une relation du type client-serveur les relations ne sont pas symétriques. Démarrer une telle relation suppose que le programme sait quel rôle il doit jouer.
- Une connexion réseau peut être du type connecté ou non. Dans le premier cas, une fois la connexion établie le processus émetteur discute uniquement avec le processus destinataire. Dans le cas d'un mode non connecté, un même processus peut envoyer plusieurs data-grammes à plusieurs autres processus sur des machines différentes.
- Une connexion est définie par un quintuplet (cf cours TCP page 85) qui est beaucoup plus compliqué qu'un simple nom de fichier.
- L'interface réseau supporte de multiples protocoles comme XNS, APPLETALK[3], la liste n'est pas exhaustive. Un sous système de gestion de fichiers sous Unix ne supporte qu'un seul format.

En conclusion de ce paragraphe on peut dire que le terme *socket* désigne, d'une part un ensemble de primitives, on parle des *sockets de Berkeley*, et d'autre part l'extrémité d'un canal de communication (point de communication) par lequel un processus peut émettre ou recevoir des données. Ce point de communication est représenté par une variable entière, similaire à un descripteur de fichier.

[3]L'inspection du fichier `/usr/include/sys/socket.h` sous `Unix` en explicite une trentaine

3 Étude des primitives

Ce chapitre est consacré à une présentation des primitives de base pour programmer des applications en réseaux. Pour être bien complet il est fortement souhaitable de doubler sa lecture avec celle des pages de `man` correspondantes.

4 Création d'une socket

La création d'une socket se fait par l'appel système `socket`.

```
#include  <sys/types.h>   /* Pour toutes les primitives */
#include  <sys/socket.h>  /* de ce chapitre il faut */
#include  <netinet/in.h>  /* inclure ces fichiers. */


          int socket(int PF, int TYPE, int PROTOCOL);
```

PF Spécifie la famille de protocole ("**P**rotocol **F**amily") à utiliser avec la socket. Sur tout système qui permet l'usage des sockets, on doit trouver au moins les familles :

`PF_INET`	:	Pour les sockets IPv4
`PF_INET6`	:	Pour les sockets IPv6
`PF_CCITT`	:	Pour l'interface avec X25
`PF_LOCAL`	:	Pour rester en mode local (`pipe`)...
`PF_UNIX`	:	Idem `AF_LOCAL`
`PF_ROUTE`	:	Accès à la table de routage
`PF_KEY`	:	Accès à une table de clefs (IPsec)

Mais il existe d'autres implémentations notamment avec les protocoles :

`PF_APPLETALK`	:	Pour les réseaux Apple
`PF_NS`	:	Pour le protocole Xerox NS
`PF_ISO`	:	Pour le protocole de l'OSI
`PF_SNA`	:	Pour le protocole SNA d'IBM
`PF_IPX`	:	Protocole Internet de Novell
...	:	...

Le préfixe `PF` est la contraction de "**P**rotocol **F**amily" On peut également utiliser le préfixe **AF**, pour "**A**ddress **F**amily". Les deux nommages sont possibles ; l'équivalence est définie dans le fichier d'entête `socket.h`.

TYPE Cet argument spécifie le type de communication désiré. En fait avec la famille PF_INET, le type permet de faire le choix entre un mode connecté, un mode non connecté ou une intervention directe dans la couche IP :

SOCK_STREAM	:	Mode connecté	Couche transport
SOCK_DGRAM	:	Mode non connecté	Idem
SOCK_RAW	:	Dialogue direct avec la couche IP	

Faut-il repréciser que seules les sockets en mode connecté permettent les liaisons "full-duplex".

PROTOCOL Ce troisième argument permet de spécifier le protocole à utiliser. Il peut être du type UDP ou TCP sous Unix.

IPPROTO_TCP	:	TCP
IPPROTO_UDP	:	UDP
IPPROTO_RAW, IPPROTO_ICMP	:	uniquement avec SOCK_RAW

PROTOCOL est typiquement mis à zéro car l'association de la famille de protocole et du type de communication définit explicitement le protocole de transport :

PF_INET + SOCK_STREAM $\Longrightarrow$ TCP = IPPROTO_TCP
PF_INET + SOCK_DGRAM $\Longrightarrow$ UDP = IPPROTO_UDP

La primitive socket retourne un entier qui est le descripteur de la socket nouvellement créée par cet appel.

Par rapport à la connexion future cette primitive ne fait que donner le premier élément du quintuplet :

{**protocole**, port local, adresse locale, port éloigné, adresse éloignée}

Si la primitive renvoie -1, la variable globale errno donne l'indice du message d'erreur idoine dans la table sys_errlist, que la bibliothèque standard sait parfaitement exploiter [4].

Remarque importante :

Comme pour les entrées/sorties sur fichiers, un appel système fork duplique la table des descripteurs de fichiers ouverts du processus père dans le processus fils. Ainsi les descripteurs de sockets sont également transmis.

Le bon usage du descripteur de socket partagé entre les deux processus incombe donc à la responsabilité du programmeur.

Enfin, quand un processus a fini d'utiliser une socket il appelle la primitive close avec en argument le descripteur de la socket :

close(*descripteur de socket*);

Si un processus ayant ouvert des sockets vient à s'interrompre pour une raison quelconque, en interne la socket est fermée et si plus aucun processus n'a de descripteur ouvert sur elle, le noyau la supprime.

[4] cf la page de manuel de perror(3)

5 Spécification d'une adresse

Il faut remarquer qu'une `socket` est créée sans adresse de l'émetteur ni celle du destinataire.

Pour les protocoles de l'Internet cela signifie que la création d'une `socket` est indépendante d'un numéro de port.

Dans beaucoup d'applications, surtout des clients, on a pas besoin de s'inquiéter du numéro de port, le protocole sous-jacent s'occupe de choisir un numéro de port pour l'application.

Cependant un serveur qui fonctionne suivant un numéro de port "bien connu" doit être capable de spécifier un numéro de port bien précis.

Une fois que la socket est créée, la primitive `bind` permet d'effectuer ce travail, c'est à dire d'associer une adresse IP et un numéro de port à une socket :

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

`socket` Est le descripteur de la socket, renvoyé par `socket`.

`myaddr` Est une structure qui spécifie l'adresse locale avec laquelle `bind` doit travailler.

`addrlen` Est un entier qui donne la longueur de l'adresse, en octets.

La description de l'adresse change d'une famille de protocole (cf `socket`) à une autre, c'est pourquoi les concepteurs ont choisi de passer en argument une structure plutot qu'un entier. Cette structure générique, est nommée généralement `sockaddr`.

Elle commence généralement par deux octets qui rappellent la famille de protocole, suivis de 14 octets qui définissent l'adresse en elle-même :

```
struct sockaddr {
    unsigned short  sa_family ;
    char sa_data[14] ;
} ;
```

Pour la famille PF_INET cette structure se nomme `sockaddr_in`, elle est définie de la façon suivante :

```
struct sockaddr_in {
    short sin_family ; /* PF_INET */
    unsigned short sin_port ; /* No de port. */
    struct in_addr sin_addr ; /* 32 bits. */
    char sin_zero[8] ; /* Inutilises. */
} ;

struct in_addr {
    unsigned long s_addr ; /* 32b Internet */
} ;
```

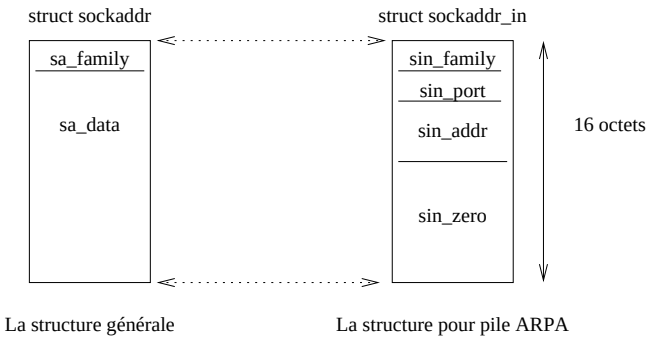


figure XI.03

La primitive `bind` ne permet pas toutes les associations de numéros de port, par exemple si un numéro de port est déjà utilisé par un autre processus, ou si l'adresse Internet est invalide.

Il y a trois utilisations de la primitive :

1. En général les serveurs ont des numéros de port bien connus du système (cf `/etc/services`. Bind dit au système "c'est mon adresse, tout message reçu à cette adresse doit m'être renvoyé". Que cela soit en mode connecté ou non, les serveurs ont besoin de le dire avant de pouvoir accepter les connexions.
2. Un client peut préciser sa propre adresse, en mode connecté ou non.
3. Un client en mode non connecté a besoin que le système lui assigne une adresse particulière, ce qui autorise l'usage des primitives `read` et `write` traditionnellement dédiées au mode connecté.

`Bind` retourne 0 si tout va bien, -1 si une erreur est intervenue. Dans ce cas la variable globale errno est positionnée à la bonne valeur.

Cet appel système complète l'adresse locale et le numéro de port du quintuplet qui qualifie une connexion. Avec `bind+socket` on a la moitié d'une connexion, à savoir un protocole, un numéro de port et une adresse IP :

{**protocole, port local, adresse locale**, port éloigné, adresse éloignée}

6 Connexion à une adresse distante

L'initiative de la demande de connexion est typiquement la préoccupation d'un processus client.

Quand une socket est créée, elle n'est pas associée avec une quelconque destination. C'est la primitive `connect` qui permet d'établir la connexion, si le programmeur choisit un mode de fonctionnement "connecté", sinon elle n'est pas très utile (voir plus loin).

Pour les protocoles orientés connexion, cet appel système rend la main au code utilisateur une fois que la liaison entre les deux systèmes est établie. Durant cette phase des messages sont échangés ; les deux systèmes échangent des informations sur les caractéristiques de la liaison future (cf cours TCP page 85).

Tant que cette liaison n'est pas définie au niveau de la couche de transport, la primitive `connect` ne rend pas la main. Dans le cas ou une impossibilité se présente elle retourne -1 et positionne `errno` à la bonne valeur. Dans le cas ou tout va bien elle retourne 0.

Les clients n'ont pas besoin de faire appel à la primitive `bind` avant d'invoquer `connect` car la connexion renseigne d'un coup les éléments qui manquent.

Dans le cas d'un client en mode non connecté un appel à `connect` n'est pas faux mais il ne sert à rien et redonne aussitôt la main au code utilisateur. Le seul intérêt que l'on peut voir est que l'adresse du destinataire est fixée et que l'on peut alors utiliser les primitives `read, write, recv` et `send`, traditionnellement réservées au mode connecté.
La primitive connect a le prototype suivant :

```
int connect(int sockfd,const struct sockaddr *servaddr,socklen_t addrlen) ;
```

`sockfd` Est le descripteur de socket renvoyé par la primitive `socket`.

`servaddr` est une structure qui définit l'adresse du destinataire, la même structure que pour `bind`.

`addrlen` Donne la longueur de l'adresse, en octets.

Le quintuplet est maintenant complet :

{protocole, port local, adresse locale, port éloigné, adresse éloignée}

7 Envoyer des données

Une fois qu'un programme d'application a créé une socket, il peut l'utiliser pour transmettre des données. Il y a cinq primitives possibles pour ce faire :

`send, write, writev, sendto, sendmsg`

`Send, write` et `writev` fonctionnent uniquement en mode connecté, parce-qu'elles n'offrent pas la possibilité de préciser l'adresse du destinataire.

Les différences entre ces trois primitives sont mineures.

```
ssize_t write(int descripteur, const void *buffer, size_t longueur) ;
```

Quand on utilise `write`, le `descripteur` désigne l'entier renvoyé par la primitive `socket`. Le `buffer` contient les octets à transmettre, et `longueur` leur cardinal.

Tous les octets ne sont pas forcément transmis d'un seul coup, et ce n'est pas une condition d'erreur. En conséquence il est absolument nécessaire de tenir compte de la valeur de retour de cette primitive, négative ou non.

La primitive `writev` est sensiblement la même que `write` simplement elle permet d'envoyer un tableau de structures du type `iovec` plutot qu'un simple buffer, l'argument `vectorlen` spécifie le nombre d'entrées dans `iovector` :

```
ssize_t writev(int descriptor, const struct iovec *iovector, int
                                vectorlen) ;
```

La primitive `send` à la forme suivante :

```
int send(int s, const void *msg, size_t len, int flags) ;
```

`s` Désigne l'entier renvoyé par la primitive `socket`.

`msg` Donne l'adresse du début de la suite d'octets à transmettre.

`len` Spécifie le nombre de ces octets.

`flags` Ce drapeau permet de paramètrer la transmission du data-gramme, notamment si le buffer d'écriture est plein ou si l'on désire, par exemple et avec TCP, faire un envoi en urgence (`out-of-band`) :

0	:	Non opérant, c'est le cas le plus courant.
`MSG_OOB`	:	Pour envoyer ou recevoir des messages `out-of-band`.
`MSG_PEEK`	:	Permet d'aller voir quel message on a reçu sans le lire, c'est à dire sans qu'il soit effectivement retiré des buffers internes (ne s'applique qu'à `recv` (page 210).

Les deux autres primitives, `sendto` et `sendmsg` donnent la possibilité d'envoyer un message via une socket en mode non connecté. Toutes deux réclament que l'on spécifie le destinataire à chaque appel.

```
ssize_t sendto(int s,const void *msg,size_t len,int flags, const struct
                    sockaddr *to, socklen_t tolen) ;
```

Les quatre premiers arguments sont exactement les mêmes que pour `send`, les deux derniers permettent de spécifier une adresse et sa longueur avec une structure du type `sockaddr`, comme vu précédemment avec `bind`.

Le programmeur soucieux d'avoir un code plus lisible pourra utiliser la deuxième primitive :

```
ssize_t sendmsg(int sockfd, const struct msghdr *messagestruct,int flags) ;
```

Où `messagestruct` désigne une structure contenant le message à envoyer sa longueur, l'adresse du destinataire et sa longueur. Cette primitive est très commode à employer avec son pendant `recvmsg` car elle travaille avec la même structure.

8 Recevoir des données

Symétriquement aux cinq primitives d'envoi, il existe cinq primitives de réception : `read, readv, recv, recvfrom, recvmsg`.

La forme conventionnelle `read` d'Unix n'est possible qu'avec des sockets en mode connecté car son retour dans le code utilisateur ne s'accompagne d'aucune précision quant à l'adresse de l'émetteur. Sa forme d'utilisation est :

```
ssize_t read(int descripteur, void *buffer,size_t longueur) ;
```

Bien sur, si cette primitive est utilisée avec les sockets BSD, le descripteur est l'entier renvoyé par un appel précédent à la primitive `socket`. `buffer` et `longueur` spécifie respectivement le buffer de lecture et la longueur de ce que l'on accepte de lire.

Chaque lecture ne renvoie pas forcément le nombre d'octets demandés, mais peut être un nombre inférieur.
Mais le programmeur peut aussi employer le `readv`, avec la forme :

```
ssize_t readv(int descripteur, const struct iovec *iov, int vectorlen) ;
```

Avec les même caractéristiques que pour le `readv`.

En addition aux deux primitives conventionnelles, il y a trois primitives nouvelles pour lire des messages sur le réseau :

```
ssize_t recv(int s, void *buf, size_t len, int flags) ;
```

`s` L'entier qui désigne la socket.

`buf` Une adresse où l'on peut écrire, en mémoire.

`len` La longueur du buffer.

`flags` Permet au lecteur d'effectuer un contrôle sur les paquets lus.

```
ssize_t recvfrom(int s, void *buf,size_t len, int flags,struct sockaddr
                    *from, socklen_t *fromlen) ;
```

Les deux arguments additionnels par rapport à `recv` sont des pointeurs vers une structure de type `sockaddr` et sa longueur. Le premier contient l'adresse de l'émetteur. Notons que la primitive `sendto` fournit une adresse dans le même format, ce qui facilite les réponses.

La dernière primitive `recvmsg` est faite pour fonctionner avec son homologue `sendmsg` :

```
ssize_t recvmsg(int sockfd, struct msghdr *messagestruct,int flags) ;
```

La structure `messagestruct` est exactement la même que pour `sendmsg` ainsi ces deux primitives sont faites pour fonctionner de paire.

9 Spécifier une file d'attente

Imaginons un serveur en train de répondre à un client, si une requête arrive d'un autre client, le serveur étant occupé, la requête n'est pas prise en compte, et le système refuse la connexion.

La primitive `listen` est là pour permettre la mise en file d'attente des demandes de connexions.

Elle est généralement utilisée après les appels de `socket` et de `bind` et immédiatement avant le `accept`.

```
int listen(int sockfd, int backlog) ;
```

`sockfd` l'entier qui décrit la socket.

`backlog` Le nombre de connexions possibles en attente (quelques dizaines). La valeur maximale est fonction du paramètrage du noyau. Sous FreeBSD la valeur maximale par défaut est de 128 (sans paramètrage spécifique du noyau), alors que sous Solaris 9, " There is currently no backlog limit ".

Le nombre de fois où le noyau refuse une connexion est comptabilisé et accessible au niveau de la ligne de commande via le résultat de l'exécution de la commande `netstat -s -p tcp` (chercher " listen queue overflow "). Ce paramètre est important à suivre dans le cas d'un serveur très sollicité.

10 Accepter une connexion

Comme nous l'avons vu un serveur utilise les primitives `socket`, `bind` et `listen` pour se préparer à recevoir les connexions. Il manque cependant à ce trio le moyen de dire au protocole "j'accepte désormais les connexions entrantes". Ce moyen existe, il passe par l'usage de la primitive `accept`.

Quand le serveur invoque cette primitive il se met en attente bloquante d'une connexion, la forme de la primitive est :

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen) ;
```

Qui s'utilise comme suit :

```
int newsock ;
newsock = accept(sockfd, addr, addrlen) ;
```

`sockfd` descripteur de la socket, renvoyé par la primitive du même nom.

`addr` Un pointeur sur une structure du type `sockaddr`.

`addlen` Un pointeur sur un entier.

Quand une connexion arrive, les couches sous-jacentes du protocole de transport remplissent la structure `addr` avec l'adresse du client qui fait la demande de connexion. `Addrlen` contient alors la longueur de l'adresse.

Puis le système crée une nouvelle socket dont il renvoie le descripteur, récupéré ici dans `newsock`. Ainsi la socket originale reste disponible pour d'éventuelles autres connexions.

Donc, quand une connexion arrive, l'appel à la primitive `accept` retourne dans le code utilisateur.

11 Terminer une connexion

Dans le cas du mode connecté on termine une connexion avec la primitive `close` ou `shutdown`.

```
int close(descripteur) ;
```

La primitive bien connue sous Unix peut être aussi employée avec un descripteur de socket. Si cette dernière est en mode connecté, le système assure que tous les octets en attente de transmission seront effectivement transmis dans de bonnes conditions. Normalement cet appel retourne immédiatement, cependant le kernel peut attendre le temps de la transmission des derniers octets (transparent).

Le moyen le plus classique de terminer une connexion est de passer par la primitive `close`, mais la primitive `shutdown` permet un plus grand contrôle sur les connexions en "full-duplex".

```
int shutdown(int sockfd, int how) ;
```

`Sockfd` est le descripteur à fermer, `how` permet de fermer partiellement le descripteur suivant les valeurs qu'il prend :

0 Aucune donnée ne peut plus être reçue par la socket.

1 Aucune donnée ne peut plus être émise.

2 Combinaison de 0 et de 1 (équivalent de `close`).

Enfin, pour une socket en mode connecté, si un processus est interrompu de manière inopinée (réception d'un signal de fin par exemple), un " reset " (voir page 88) est envoyé à l'hôte distant ce qui provoque la fin brutale de la connexion. Les octets éventuellement en attente sont perdus.

12 Schéma général d'une connexion

Il est temps pour nous de donner un aperçu de la structure d'un serveur et d'un client, puisque c'est ainsi que nous appelerons désormais les destinataire et émetteur.

Schéma d'une relation client–serveur en mode connecté :

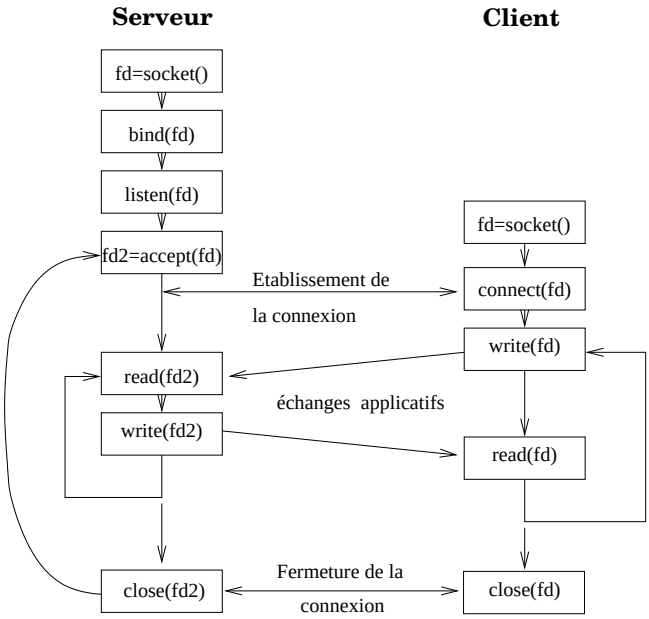


figure XI.04

Schéma d'une relation client–serveur en mode non connecté :

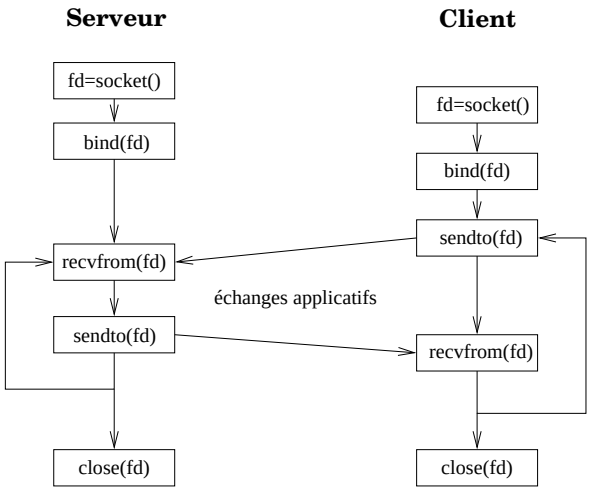


figure XI.05

13 Exemples de code “ client ”

L'objectif de ces deux exemples est de montrer le codage en C et le fonctionnement d'un client du serveur de date (RFC 867, “ daytime protocol ”) présent sur toute machine unix[5].

Ce serveur fonctionne en mode connecté ou non, sur le port 13 qui lui est réservé (`/etc/services`). Ici le serveur est une machine portant l'adresse IP `192.168.52.232`. La connaissance de l'adresse IP du client n'est absolument pas utile pour la compréhension de l'exemple.

En mode TCP le simple fait de se connecter provoque l'émission de la chaîne ascii contenant la date vers le client et sa déconnexion.

En mode UDP il suffit d'envoyer un datagramme quelconque (1 caractère) au serveur puis d'attendre sa réponse.

13.1 Client TCP “ DTCPcli ”

Exemple d'usage :

```
$ ./DTCPcli 192.168.52.232
Date(192.168.52.232) = Wed Dec 10 20:59:46 2003
```

Une capture des trames réseau échangées lors de l'exécution de cette commande se trouve page 217.

ligne 28 Déclaration de la structure `saddr` du type `sockaddr_in`, à utiliser avec IPv4. Attention, il s'agit bien d'une structure et non d'un pointeur de structure.

ligne 34 La variable `sfd`, reçoit la valeur du descripteur de socket. Celle-ci est dédiée au protocole TCP.

ligne 38 Le champ `sin_family` de la structure `saddr` indique que ce qui suit (dans la structure) concerne IPv4.

ligne 39 Le champ `sin_port` est affecté à la valeur du numéro de port sur lequel écoute le serveur. Il faut remarquer l'usage de la fonction `htons` (en fait une macro du pré-processeur `cpp`) qui s'assure que ce numéro de port respecte bien le NBO (“ network Byte Order ”), car cette valeur est directement recopiée dans le champ PORT DESTINATION du protocole de transport employé (voir page 80 pour UDP et page 87 pour TCP).

Nous en dirons plus sur `htons` au chapitre suivant.

Si le processeur courant est du type “ little endian ” (architecture Intel par exemple) les octets sont inversés (le NBO est du type “ big endian ”). Vous croyez écrire 13 alors qu'en réalité pour le réseau vous avez écrit 3328 (0x0D00) ce qui bien évidement ne conduit pas au même résultat, sauf si un serveur de date écoute également sur le port 3328, ce qui est très peu probable car non conventionnel.

[5] Son activation est éventuellement à faire à partir du serveur de serveur **inetd**, page 257

En résumé, si le programmeur n'utilise pas la fonction `htons`, ce code n'est utilisable que sur les machines d'architecture “ big endian ”.

```
1  /* $Id: DTCPcli.c,v 1.1 2002/11/24 16:52:50 fla Exp $
2   *
3   * Client TCP pour se connecter au serveur de date (port 13 - RFC 867).
4   * La syntaxe d'utilisation est : DTCPcli <adresse ip sous forme décimale>
5   *
6   */
7
8  #include        <stdio.h>
9  #include        <string.h>
10 #include        <unistd.h>
11 #include        <sysexits.h>
12
13 #include        <sys/types.h>
14 #include        <sys/socket.h>
15 #include        <sys/param.h>
16 #include        <netinet/in.h>
17 #include        <arpa/inet.h>
18
19 #define         USAGE   "Usage:%s adresse_du_serveur\n"
20 #define         MAXMSG  1024
21 #define         NPORT   13
22
23 int
24 main(int argc, char *argv[])
25 {
26     int         n, sfd ;
27     char        buf[MAXMSG] ;
28     struct      sockaddr_in saddr ;
29
30     if (argc != 2) {
31         (void)fprintf(stderr,USAGE,argv[0]) ;
32         exit(EX_USAGE) ;
33     }
34     if ((sfd = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP)) < 0) {
35         perror("socket") ;
36         exit(EX_OSERR) ;
37     }
38     saddr.sin_family      = AF_INET ;
39     saddr.sin_port        = htons(NPORT) ;      /* Attention au NBO !   */
40     saddr.sin_addr.s_addr = inet_addr(argv[1]) ;
41     if (connect(sfd,(struct sockaddr *)&saddr,sizeof saddr) < 0) {
42         perror("connect") ;
43         exit(EX_OSERR) ;
44     }
45     if ((n = read(sfd, buf,MAXMSG-1)) < 0) {
46         perror("read") ;
47         exit(EX_OSERR) ;
48     }
49     buf[n] = '\0' ;
50     (void)printf("Date(%s) = %s\n",argv[1],buf) ;
51
52     exit(EX_OK) ;                                 /* close(sfd) implicite */
53 }
```

DTCPcli.c

ligne 40 Le champ `s_addr` de la structure `sin_addr` se voit affecté de l'adresse IP. C'est donc l'écriture de quatre octets (IPv4), pouvant comporter un caractère ascii 0, donc interprétable comme le caractère de fin de chaîne du langage C.

C'est pourquoi à cet endroit on ne peut pas employer les habituelles fonctions de la bibliothèque standard (celles qui commencent par `str`).

Ici le problème se complique un peu dans la mesure où l'on dispose au départ d'une adresse IP sous sa forme décimale pointée.

La fonction `inet_addr` gère parfaitement ce cas de figure. Nous en dirons plus à son sujet au chapitre suivant.

ligne 41 Appel à la primitive `connect` pour établir la connexion avec le serveur distant. Quand celle-ci retourne dans le code du programme, soit la connexion a échoué et il faut traiter l'erreur, soit la connexion est établie. L'échange préliminaire des trois paquets s'est effectué dans de bonnes conditions (page 90).

Du point de vue TCP, les cinq élément du quintuplet qui caractérise la connexion sont définis (page **??**).

Sur la capture des paquets de la page 217 nous sommes arrivés à la ligne 6, c'est à dire l'envoi de l'acquittement par le client du paquet de synchronisation envoyé par le serveur (ligne 3 et 4).

Il faut noter que bien que nous ayons transmis la structure `saddr` par adresse (caractère &) et non par valeur, la primitive `connect` ne modifie pas son contenu pour autant.

Notons également l'usage du " cast " du C pour forcer le type du pointeur (le prototype de la primitive exige à cet endroit un pointeur de type `sockaddr`).

ligne 45 Appel à la primitive `read` pour lire le résultat en provenance du serveur, à l'aide du descripteur `sfd`.

Sur la capture d'écran on voit ligne 8 (et 9) l'envoi de la date en provenance du serveur, d'une longueur de 26 caractères.

Ce nombre de caractères effectivement lus est affecté à la variable `n`.

Ce nombre ne peut excéder le résultat de l'évaluation de $MAXMSG - 1$, qui correspond à la taille du buffer `buf` moins 1 caractère prévu pour ajouter le caractère 0 de fin de chaîne.

En effet, celui-ci fait partie de la convention de représentation des chaînes de caractères du langage C. Rien ne dit que le serveur qui répond ajoute un tel caractère à la fin de sa réponse. Le contraire est même certain puisque la RFC 867 n'y fait pas mention.

Remarque : le buffer `buf` est largement surdimensionné compte tenu de la réponse attendue. La RFC 867 ne prévoit pas de taille maximum si ce n'est implicitement celle de l'expression de la date du système en anglais, une quarantaine d'octets au maximum.

ligne 49 Ajout du caractère de fin de chaîne en utilisant le nombre de caractères renvoyés par `read`.

ligne 52 La sortie du programme induit une clôture de la socket coté client. Coté serveur elle est déjà fermée (séquence `write` + `close`) comme on peut le voir ligne 8 (flag FP, page 88) ci-après dans la capture du trafic entre le client et le serveur.

Remarque : rien n'est explicitement prévu dans le code pour établir la socket coté client, à savoir l'association d'un numéro de port et d'une adresse IP. En fait c'est la primitive `connect` qui s'en charge. L'adresse IP est celle de la machine. Le numéro de port est choisi dans la zone d'attribution automatique comme nous l'avons examiné page 81.

Il existe bien entendu une possibilité pour le programme d'avoir connaissance de cette information : la primitive `getsockname`.

```
1  23:03:29.465183 client.2769 > serveur.daytime: S 2381636173:2381636173(0)
2      win 57344 <mss 1460,nop,wscale 0,nop,nop,timestamp 299093825 0> (DF)
3  23:03:29.465361 serveur.daytime > client.2769: S 3179731077:3179731077(0)
4      ack 2381636174 win 57344 <mss 1460,nop,wscale 0,nop,nop,timestamp
5      4133222 299093825> (DF)
6  23:03:29.465397 client.2769 > serveur.daytime: . ack 1 win 57920
7                  <nop,nop,timestamp 299093826 4133222> (DF)
8  23:03:29.466853 serveur.daytime > client.2769: FP 1:27(26) ack 1 win 57920
9                  <nop,nop,timestamp 4133223 299093826> (DF)
10 23:03:29.466871 client.2769 > serveur.daytime: . ack 28 win 57894
11                 <nop,nop,timestamp 299093826 4133223> (DF)
12 23:03:29.467146 client.2769 > serveur.daytime: F 1:1(0) ack 28 win 57920
13                 <nop,nop,timestamp 299093826 4133223> (DF)
14 23:03:29.467296 serveur.daytime > client.2769: . ack 2 win 57920
15                 <nop,nop,timestamp 4133223 299093826> (DF)
```

Trafic client-serveur TCP, capturé avec `tcpdump`

13.2 Client UDP “ DUDPcli ”

Exemple d'usage :

```
$ ./DUDPcli 192.168.52.232
Date(192.168.52.232) = Wed Dec 10 20:56:58 2003
```

ligne 33 Ouvertude d'une socket UDP, donc en mode non connecté.

ligne 41 Envoit d'un caractère (NULL) au serveur, sans quoi il n'a aucun moyen de connaître notre existence.

ligne 37, 38 et 39 Le remplissage de la structure `saddr` est identique à celui de la version TCP.

ligne 45 Réception de caractères en provenance du réseau.

Il faut remarquer que rien n'assure que les octets lus sont bien en provenance du serveur de date interrogé.

Nous aurions pu utiliser la primitive `recvfrom` dont un des arguments est une structure d'adresse contenant justement l'adresse de la socket qui envoie le datagramme (ici la réponse du serveur).

Le raisonnement sur la taille du buffer est identique à celui de la version TCP.

La capture de trames suivante montre l'extrême simplicité de l'échange en comparaison avec celle de la version utilisant TCP !

```
23:23:17.668689 client.4222 > serveur.daytime: udp 1
23:23:17.670175 serveur.daytime > client.4222: udp 26
```

Trafic client-serveur UDP, capturé avec `tcpdump`

```
/* $Id: DUDPcli.c,v 1.1 2002/11/24 16:52:50 fla Exp $
 *
 * Client UDP pour se connecter au serveur de date (port 13 - RFC 867).
 * La syntaxe d'utilisation est : DUDPcli <adresse ip sous forme décimale>
 *
 */
#include        <stdio.h>
#include        <string.h>
#include        <unistd.h>
#include        <sysexits.h>

#include        <sys/types.h>
#include        <sys/socket.h>
#include        <sys/param.h>
#include        <netinet/in.h>
#include        <arpa/inet.h>

#define         USAGE   "Usage:%s adresse_du_serveur\n"
#define         MAXMSG  1024
#define         NPORT   13

int
main(int argc, char *argv[])
{
    int         n, sfd ;
    char        buf[MAXMSG] ;
    struct      sockaddr_in saddr ;

    if (argc != 2) {
        (void)fprintf(stderr,USAGE,argv[0]) ;
        exit(EX_USAGE) ;
    }
    if ((sfd = socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP)) < 0) {
        perror("socket") ;
        exit(EX_OSERR) ;
    }
    saddr.sin_family      = AF_INET ;
    saddr.sin_port        = htons(NPORT) ;      /* Attention au NBO !   */
    saddr.sin_addr.s_addr = inet_addr(argv[1]) ;
    buf[0] = '\0' ;
    if (sendto(sfd,buf,1,0,(struct sockaddr *)&saddr, sizeof saddr) != 1) {
        perror("sendto") ;
        exit(EX_OSERR) ;
    }
    if ((n=recv(sfd,buf,MAXMSG-1,0)) < 0) {
        perror("recv") ;
        exit(EX_OSERR) ;
    }
    buf[n] = '\0' ;
    (void)printf("Date(%s) = %s\n",argv[1],buf) ;

    exit(EX_OK) ;                               /* close(sfd) implicite */
}
```

DUDPcli.c

14 Conclusion et Bibliographie

En conclusion on peut établir le tableau suivant :

	Protocole	Adresses locale et N° de port.	Adresse éloignée et N° de port.
Serveur orienté connexion	`socket`	`bind`	`listen, accept`
Client orienté connexion	`socket`	`connect`	
Serveur non orienté connexion	`socket`	`bind`	`recvfrom`
Client non orienté connexion	`socket`	`bind`	`sendto`

Concernant les exemples de code client, consulter cette RFC :

RFC 867 “ Daytime Protocol ”. J. Postel. May-01-1983. (Format : TXT=2405 bytes) (Also STD0025) (Status : STANDARD)

Pour en savoir davantage, outre les pages de `man` des primitives citées dans ce chapitre, on pourra consulter les documents de référence suivants :

- Stuart Sechrest — “An Introductory 4.4BSD Interprocess Communication Tutorial” — Re imprimé dans “Programmer’s Supplementary Documents” — O’Reilly & Associates, Inc. — 1994[6]
- W. Richard Stevens — “Unix Network Programming” — Prentice All — 1990
- W. Richard Stevens — “Unix Network Programming” — Second edition — Prentice All — 1998
- Douglas E. Comer – David L. Stevens — “Internetworking with TCP/IP – Volume III” (BSD Socket version) — Prentice All — 1993
- Stephen A. Rago — “Unix System V Network Programming” — Addison–Wesley — 1993

Et pour aller plus loin dans la compréhension des mécanismes internes :

- W. Richard Stevens — “TCP/IP Illustrated Volume 2” — Prentice All — 1995
- McKusick – Bostic – Karels – Quaterman — “The Design and implementation of the 4.4 BSD Operating System” — Addison–Wesley — 1996

[6] On peut également trouver ce document dans le répertoire `/usr/share/doc/psd/20.ipctut/` des OS d’inspiration Berkeley 4.4

Chapitre XII

Compléments sur les sockets Berkeley

1 Réservation des ports

Au chapitre précédent nous avons utilisé la primitive `bind` pour assigner une adresse à une socket, dans ce paragraphe nous précisons comment choisir le numéro de port qui va bien, selon le type d'application envisagé. Nous avons déjà examiné ce point dans les grandes lignes page 81.
Il y a deux manières d'assigner un N° de port à une socket :

1. Le processus spécifie le numéro. C'est typiquement ce que fait un serveur. On suppose bien évidement que les clients sont au courant ou qu'ils disposent d'un mécanisme qui peut les renseigner (cf cours sur les `RPC`).
2. Le processus laisse le système lui assigner automatiquement un numéro de port. C'est typiquement ce que fait un client, sauf cas contraire exigé par le protocole d'application (cf cours sur les "remote execution").

En règle générale le développeur d'application ne s'attribue pas au hasard un (ou plus) numéro de port. Il doit respecter quelques contraintes comme ne pas utiliser les ports déjà attribués. Ceux-ci figurent dans une RFC particulière. La dernière en date est la RFC 1700 [Reynolds & Postel 1994]) au paragraphe "WELL KNOWN PORT NUMBERS". Plus simplement, sur toute machine Unix à jour, une liste de ces ports se trouve dans le fichier `/etc/services` [1].

Codé sur deux octets non signés, le numéro de port permet 65536 possibilités de 0 à 65535. Cette échelle est fragmentée de deux manières, l'ancienne ou la nouvelle méthode. Toutes les applications sur tous les systèmes d'exploitation n'ont pas encore adopté la nouvelle approche, les deux vont donc cohabiter un certain temps, ne serait-ce qu'à cause de "vieilles" applications non mises à jour.

[1] `http ://www.iana.org/assignments/port-numbers` pour se tenir au courant des évolutions de cette liste

1.1 Réservation de port — Ancienne méthode

Port N° 0 Ce numéro n'est pas utilisable pour une application, c'est une sorte de "jocker" qui indique au système que c'est à lui de compléter automatiquement le numéro (voir plus loin **de 1024 à 5000**).

Port de 1 à 255 Pour utiliser cette zone il faut avoir les droits du `root` à l'exécution pour que le `bind` ne retourne pas une erreur. Les serveurs "classiques" (`domain, ftp, smtp, telnet, ...`) se situent tous dans cette partie.

Ports de 256 à 511 Jadis considéré comme une "réserve" des serveurs officiels commencent à s'y installer, faute de place dans la zone précédente. Il faut également avoir les droits du `root` pour utiliser un numéro de port dans cette zone.

Port de 512 à 1023 Une fonction `rresvport` permet l'attribution automatique d'un numéro de port dans cette zone, pour des applications ayant un $UID = 0$. Par exemple, c'est dans cette zone qu'`inetd` (cf cours sur les serveurs) attribue des ports automatiquement pour les outils en "r" de Berkeley (`rlogin, rcp, rexec, rdist,...`).

Port de 1024 à 5000 Zone d'attribution automatique par `bind`. Lorsque l'utilisateur (non `root`) utilise 0 comme numéro, c'est le premier port libre qui est employé. Si tous les utilisateurs peuvent s'attribuer "manuellement" un numéro dans cette zone, il vaut mieux éviter de le faire, la suivante est prévue pour cela.

5001 à 65535 Zone "libre" attention cependant car de très nombreux serveurs y ont un port réservé, et pas des moindres comme le serveur `X11` sur le port 6000 !

1.2 Réservation de port — Nouvelle méthode

Port N° 0 Ce numéro n'est pas utilisable pour une application, c'est une sorte de "jocker" qui indique au système que c'est à lui de compléter automatiquement le numéro (voir plus loin **de 49152 à 65535**).

Port de 1 à 1023 Pour utiliser cette zone il faut avoir les droits du `root` à l'exécution pour que le `bind` ne retourne pas une erreur. Les serveurs "classiques" (`domain, ftp, smtp, telnet, ...`) se situent tous dans cette partie.

Port de 1024 à 49151 est la zone des services enregistrés par l'IANA et qui fonctionnent avec des droits ordinaires.

Port de 49152 à 65535 est la zone d'attribution automatique des ports, pour la partie cliente des connexions (si le protocole n'impose pas une valeur particulière) et pour les tests de serveurs locaux.

2 Ordre des octets sur le réseau

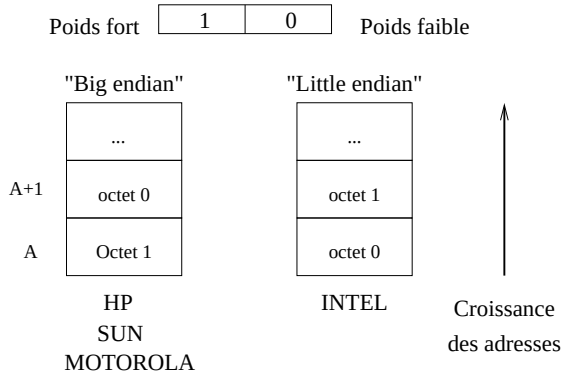


figure XII.01

Le problème de l'ordre des octets sur le réseau est d'autant plus crucial que l'on travaille dans un environnement avec des architectures hétérogènes.

La couche *Réseau* (page 28) ne transforme pas les octets de la couche *Internet* (page 27) qui elle même ne modifie pas ceux de la couche de *Transport*[2] (page 27).

Pour cette raison, le numéro de port inscrit dans l'en-tête *TCP* (vs *UDP*) de l'émetteur est exploité tel quel par la couche de transport du récepteur et donc il convient de prendre certaines précautions pour s'assurer que les couches de même niveau se comprennent.

D'un point de vue plus général, les réseaux imposent le "poids fort" avant le "poids faible", c'est le "Network Byte Order". Les architectures qui travaillent naturellement avec cette représentation n'ont donc théoriquement pas besoin d'utiliser les fonctions qui suivent, de même pour les applications qui doivent dialoguer avec d'autres ayant la même architecture matérielle. Néanmoins écrire du code "portable" consiste à **utiliser ces macros dans tous les cas**[3] **!**

Pour se souvenir de ces fonctions, il faut connaître la signification des quatre lettres utiles :

s "short" — Entier court sur 16 bits, un numéro de port par exemple.

l "long" — Entier long sur 32 bits, une adresse IP par exemple.

h "host" — La machine sur laquelle s'exécute le programme.

n "network" — Le réseau sur lequel on envoie les données.

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

[2]Nous n'abordons pas ici la question de la transmission de données hétérogènes au niveau applicatif, elle sera examinée dans le cours sur les *XDR*

[3]Pour les machines qui respectent naturellement le *NBO*, comme les stations *HP* ou *SUN*, ces fonctions sont des macros "vides".

```
u_long   htonl  (u_long) ;   /* host to network --- long */
u_short  htons  (u_short) ;  /* host to network --- short */
u_long   ntohl  (u_long) ;   /* network to host --- long */
u_short  ntohs  (u_short) ;  /* network to host --- short */
```

Par exemple, pour affecter le numéro de port 13 (service "daytime") au champ `sin_port` d'une structure de type `sockaddr_in` :

```
saddr.sin_port = htons(13) ;
```

Cette écriture est valable quelle que soit l'architecture sur laquelle elle est compilée. S'il avait fallu se passer de la macro `htons` sur une architecture Intel ("little endian"), pour l'affection du même numéro de port, il eut fallu écrire :

```
saddr.sin_port = 0x0D00 ; /* 0D hexadécimal == 13 décimal */
```

3 Opérations sur les octets

Dans le même ordre d'idée qu'au paragraphe précédent, les réseaux interconnectent des architectures hétérogènes et donc aussi des conventions de réprésentation des chaînes de caractères différentes. Pour être plus précis, le caractère `NULL` marqueur de fin de chaîne bien connu des programmeurs C, n'est pas valide partout, voire même est associé à une autre signification !

En conséquence, pour toutes les fonctions et primitives qui lisent et écrivent des octets sur le réseau, les chaînes de caractères sont toujours associées au nombre de caractères qui les composent.

Le corollaire est que les fonctions "classiques" de manipulation de chaînes en C (`strcpy, strcat, ...`) ne sont à utiliser qu'avec une extrême prudence.

Pour copier des octets d'une zone vers une autre il faut utiliser `bcopy`, pour comparer deux buffers, `bcmp`, enfin pour mettre à zéro (remplir d'octet `NULL`) une zone, `bzero`.

```
#include  <string.h>
void  bcopy (const void *src, void *dst, size_t len) ;
int   bcmp (const void *b1, const void *b2, size_t len) ;
void  bzero (const void *b, size_t len) ;
```

bcopy Attention, `len` octets de `src` sont copiés dans `dst` et non l'inverse, comme dans `strcpy`.

bcmp Compare `b1` et `b2` et renvoie 0 si les `len` premiers octets sont identiques, sinon une valeur non nulle qui n'est pas exploitable vis à vis d'une quelconque relation d'ordre (à la différence de `strcmp` qui suppose les caractères dans la table *ASCII*).

bzero Met des octets `NULL` (0) `len` fois à l'adresse `b`.

Il exite des outils similaires, issus du système V : `memcpy, memcmp, memset,...`.

4 Conversion d'adresses

La plupart du temps les adresses IP sont fournies par l'utilisateur dans le format "décimal pointé", or la structure d'adresse (`sockaddr_in`) a besoin d'un entier non signé sur 32 bits. Une conversion est donc nécessaire pour passer d'une représentation à une autre.

La fonction `inet_addr` converti une adresse décimale pointée en un entier long non signé et qui respecte le *NBO*. La fonction `inet_ntoa` effectue le travail inverse.

```
#include  <sys/socket.h>
#include  <netinet/in.h>
#include  <arpa/inet.h>

unsigned long  inet_addr  (char *) ;
char *         inet_ntoa  (struct in_addr) ;
```

Remarque : Ces deux fonctions ne sont pas symétriques, le fait que `inet_addr` ne renvoie pas une structure du type `in_addr` semble être une inconsistance.

Exemple d'utilisation :

```
struct sockaddr_in saddr ;
saddr.sin_addr.s_addr = inet_addr("138.195.52.130") ;
printf("Adresse IP = %s\n",inet_ntoa(saddr.sin_addr.s_addr)) ;
```

5 Conversion hôte – adresse IP

Se pose régulièrement le problème de convertir un nom symbolique en une adresse IP et inversement. L'origine de cette information dépend de la configuration de la station de travail : c'est un serveur de noms (*DNS*), c'est un fichier (`/etc/hosts`) ou tout autre système de propagation des informations (*NIS*...). Dans tous les cas l'information arrive à un programme via une entité nommée le *resolver*, qui unifie les sources d'informations.

5.1 Une adresse IP à partir d'un nom d'hôte

```
#include  <netdb.h>
struct hostent *  gethostbyname  (char *name) ;
struct hostent {
```

```
    char *h_name ;       /* Le nom officiel */
    char **h_aliases ;   /* Tableau de synonymes */
    int h_addrtype ;    /* PF_INET pour ARPA */
    int h_length ;      /* Long. de l'adresse */
    char **h_addr_list ; /* Adresses possibles */
} ;
#define h_addr h_addr_list[0]
```

la macro `h_addr` sert à assurer la compatibilité avec les premières versions dans lesquelles il n'y avait qu'une seule adresse IP possible par hôte.

Le nom "officiel" s'oppose aux noms synonymes. Par exemple, soit une machine officiellement baptisée `pasglop.mon-domain.fr` ; si pour répondre au besoin d'une certaine application l'administrateur du réseau lui donne le surnom `www.mon-domain.fr`, celui-ci sera considéré comme un "alias" vis à vis du nom officiel et donc lu dans `h_aliases`. (voir page 122)

La fin du tableau de pointeurs est marquée par un pointeur `NULL`.

La liste des adresses est un tableau de pointeurs, le marqueur de fin de liste est également un pointeur `NULL`. Chaque adresse est une zone de `h_length` octets (cf fonction `impnet` dans l'exemple ci-après).

Le programme d'usage qui suit affiche toutes les informations contenues dans cette structure pour les hôtes dont le nom est passé en argument (le code source de cet exemple, `gethostbyname.c`, est à la page suivante).

```
$ gethostbyname puma.cti.ecp.fr isabelle.cti.ecp.fr
Nom officiel     : puma.cti.ecp.fr
Type d'adresse   : 2 - Longueur : 4
Adresse Internet : 138.195.34.2
Nom officiel     : isabelle.cti.ecp.fr
Type d'adresse   : 2 - Longueur : 4
Adresse Internet : 138.195.33.49

$ gethostbyname  anna.cti.ecp.fr
anna.cti.ecp.fr : hote inconnu !
```

```
/*
 * $Id: gethostbyname.c,v 1.3 2001/12/02 15:08:24 fla Exp $
 * Exemple d'utilisation de la fonction "gethostbyname".
 */
#include        <stdio.h>
#include        <sysexits.h>
#include        <sys/types.h>
#include        <sys/socket.h>  /* AF_INET              */
#include        <netinet/in.h>  /* struct in_addr       */
#include        <netdb.h>       /* gethostbyname        */
#include        <arpa/inet.h>   /* inet_ntoa            */

#define USAGE   "Usage:%s liste de machines distantes\n"

void
impnet(struct in_addr **list)
{
    struct      in_addr *adr ;
    while ((adr = *list++))
        (void)printf("Adresse Internet : %s\n",inet_ntoa(*adr)) ;
}

int
main(int argc,char *argv[])
{
    register    char    *ptr ;
    register    struct  hostent *pth ;
    if (argc <2) {
        (void)fprintf(stderr,USAGE,argv[0]) ;
        exit(EX_USAGE) ;
    }
    while (--argc > 0) {
        ptr = *++argv ;
        if (!(pth = gethostbyname(ptr))) {
            (void)fprintf(stderr,"%s : hote inconnu !\n",ptr) ;
            exit(EX_SOFTWARE) ;
        }
        printf ("Nom officiel    : %s\n",pth->h_name) ;
        while ((ptr = *(pth->h_aliases))!=NULL) {
            (void)printf("\talias : %s\n",ptr) ;
            pth->h_aliases++ ;
        }
        (void)printf("Type d'adresse   : %d - Longueur : %d\n",
                        pth->h_addrtype, pth->h_length) ;
        if (pth->h_addrtype == PF_INET)
            impnet((struct in_addr **)pth->h_addr_list) ;
        else
            (void)printf("Type d'adresse non reconnu !\n") ;
    }
    exit(EX_OK) ;
}
```

gethostbyname.c

5.2 Un nom d'hôte à partir d'une adresse IP

le problème symétrique se résoud avec la fonction `gethostbyaddr`. La définition du prototype se trouve au même endroit que précédement, la fonction renvoie un pointeur sur une structure du type `hostent`.

```
#include  <netdb.h>
struct hostent *  gethostbyaddr  (char *addr, int len, int type) ;
```

addr Pointe sur une structure du type `in_addr`.

len Est la longueur de `addr`.

type `PF_INET` quand on utilise la pile ARPA.

6 Conversion N° de port – service

Couramment les ports bien connus sont donnés par leur nom plutôt que par leur valeur numérique, comme par exemple dans les sorties de la commande `tcpdump`.

6.1 Le numéro à partir du nom

Un tel programme a besoin de faire la conversion symbolique — numérique, la fonction `getservbyname` effectue ce travail. L'utilisateur récupère un pointeur sur une structure du type `servent`, `NULL` dans le cas d'une impossibilité. La source d'informations se trouve dans le fichier `/etc/services`.

```
#include  <netdb.h>
struct servent *  getservbyname  (char *name, char *proto) ;

              struct servent {
                  char        *s_name ;
                  char        **s_aliases ;
                  int         s_port ;
                  char        *s_proto ;
              } ;
```

s_name Le nom officiel du service.

s_aliases Un tableau de pointeurs sur les aliases possibles. Le marqueur de fin de tableau est un pointeur à `NULL`.

s_port Le numéro du port (il respecte le *Network Byte Order*).

s_proto Le nom du protocole à utiliser pour contacter le service (*TCP* vs *UDP*).

Voici un programme de mise en application de la fonction, le code source de l'exemple, `getservbyname.c` se trouve à la page suivante.

```
$ getservbyname domain
Le service domain est reconnu dans /etc/services
protocole :tcp - N de port :53

$ getservbyname domain udp
Le service domain est reconnu dans /etc/services
protocole :udp - N de port :53
```

```
/*
 * $Id: getservbyname.c,v 1.3 2001/12/02 14:17:32 fla Exp $
 *  Exemple d'utilisation de la fonction "getservbyname".
 */
#include        <stdio.h>
#include        <sysexits.h>
#include        <sys/types.h>
#include        <netinet/in.h> /* Pour "ntohs"          */
#include        <netdb.h>      /* Pour "getservbyname" */

#define USAGE "Usage:%s <nom de service> [<nom de protocole>]\n"
#define MSG1  "Le service %s est reconnu dans /etc/services\nprotocole :%s − N° de port :%d\n"
#define MSG2  "Le service %s (%s) est introuvable dans /etc/services !\n"

int
main(int argc,char *argv[])
{
    struct      servent *serv ;

    if (argc <2) {
        (void)fprintf (stderr,USAGE,argv[0]) ;
        exit (EX_USAGE) ;
    }
    if (serv = getservbyname(argv[1],argv[2]?argv[2]:"tcp"))
        (void)printf(MSG1,serv->s_name,serv->s_proto,ntohs(serv->s_port)) ;
    else
        (void)printf(MSG2,argv[1],argv[2]?argv[2]:"") ;
    exit(EX_OK) ;
}
```

getservbyname.c

6.2 Le nom à partir du numéro

Symétriquement la fonction `getservbyport` effectue le travail inverse. Elle renvoie aussi un pointeur sur une structure `servent`, `NULL` dans le cas d'une impossibilité.

```
#include  <netdb.h>
struct servent *  getservbyport  (int port, char *proto) ;
```

Exemples d'usage :

```
$ getservbyport 53
Le port 53 correspond au service "domain" (protocole tcp).
$ getservbyport 53 udp
Le port 53 correspond au service "domain" (protocole udp).
```

Exemple de programmation :

```
/*
 * $Id: getservbyport.c,v 1.3 2001/12/02 14:26:01 fla Exp $
 * Exemple d'utilisation de la fonction "getservbyport".
 */
#include        <stdio.h>
#include        <stdlib.h>      /* Pour "atoi".         */
#include        <sysexits.h>
#include        <sys/types.h>
#include        <netinet/in.h> /* Pour "ntohs"         */
#include        <netdb.h>      /* Pour "getservbyport" */

#define USAGE   "Usage:%s <numéro de port> [<nom de protocole>]\n"
#define MSG1    "Le port %d correspond au service \"%s\" (protocole %s).\n"
#define MSG2    "Le port %s (%s) est introuvable dans /etc/services !\n"

int
main(int argc,char *argv[])
{
    struct servent *serv ;
    if (argc <2) {
        (void)fprintf(stderr,USAGE,argv[0]) ;
        exit(EX_USAGE) ;
    }
    if ((serv = getservbyport(atoi(argv[1]),argv[2]?argv[2]:"tcp")))
        (void)printf(MSG1,ntohs(serv->s_port),serv->s_name,serv->s_proto) ;
    else
        (void)printf(MSG2,argv[1],argv[2]?argv[2]:"") ;
    exit(EX_OK) ;
}
```

getservbyport.c

7 Conversion nom de protocole – N° de protocole

Les fonctions `getservbyname` et `getservbyport` délivrent un nom de protocole, donc un symbole.

Lors de la déclaration d'une `socket` le troisième argument est numérique, il est donc nécessaire d'avoir un moyen pour convertir les numéros de protocoles (`IPPROTO_UDP, IPPROTO_TCP, IPPROTO_IP,IPPROTO_ICMP,...`) en symboles et réciproquement.

Le fichiers `/etc/protocols` contient cette information, et la paire de fonctions `getprotobyname` et `getprotobynumber` l'exploitent.

```
#include  <netdb.h>
struct protoent *getprotobyname (char *name) ;
struct protoent *getprotobynumber (char *proto) ;
struct protoent {
    char *p_name ;
    char **p_aliases ;
    int p_proto ;
} ;
```

p_name Le nom officiel du protocol.

p_aliases La liste des synonymes, le dernier pointeurs est `NULL`.

p_proto Le numéro du protocole, dans `/etc/services`.

```
1  /*
2   *  $Id: getprotobyname.c,v 1.2 2001/11/29 18:45:39 fla Exp $
3   *  Exemple d'utilisation de la fonction "getprotobyname".
4   */
5  #include        <stdio.h>
6  #include        <sysexits.h>
7  #include        <netdb.h> /* Pour getprotobyname        */
8
9  #define USAGE   "Usage:%s <nom du protocole>\n"
10 #define MSG1    "Le protocole %s est reconnu dans /etc/protocols − N° : %d\n"
11 #define MSG2    "Le protocole %s est introuvable dans /etc/protocols !\n"
12
13 int
14 main(int argc,char *argv[])
15 {
16     struct protoent *proto ;
17
18     if (argc <2) {
19         (void)fprintf(stderr,USAGE,argv[0]) ;
20         exit(EX_USAGE) ;
21     }
22     if (proto = getprotobyname(argv[1]))
23         (void)printf(MSG1,proto->p_name,proto->p_proto) ;
24     else
25         (void)printf(MSG2,argv[1]) ;
26     exit(EX_OK) ;
27 }
```

getprotobyname.c

Le source qui précède donne un exemple de programmation de la fonction `getprotobyname`. Usage de ce programme :

```
$ getprotobyname ip
Le protocole ip est reconnu dans /etc/protocols - N : 0
$ getprotobyname tcp
Le protocole tcp est reconnu dans /etc/protocols - N : 6
$ getprotobyname vmtp
Le protocole ipv6 est reconnu dans /etc/protocols - N : 41
```

8 Diagnostic

Chaque retour de primitive devrait être soigneusement testé, le code généré n'en est que plus fiable et les éventuels disfonctionnements plus aisés à détecter.

Quelques unes des erreurs ajoutées au fichier d'en-tête `errno.h`

erreur	Description de l'erreur
ENOTSOCK	Le descripteur n'est pas celui d'une socket
EDESTADDRREQ	Adresse de destination requise
EMSGSIZE	Le message est trop long
EPROTOTYPE	Mauvais type de protocole pour une socket
ENOPROTOOPT	Protocole non disponible
EPROTONOSUPPORT	Protocole non supporté
ESOCKTNOSUPPORT	Type de socket non supporté
EOPNOTSUPP	Opération non supportée
EAFNOSUPPOR	Famille d'adresse non supportée
EADDRINUSE	Adresse déjà utilisée
EADDRNOTAVAIL	L'adresse ne peut pas être affectée
ENETDOWN	Réseau hors service
ENETUNREACH	Pas de route pour atteindre ce réseau
ENETRESET	Connexion coupée par le réseau
ECONNABORTED	Connexion interrompue
ECONNRESET	Connexion interrompue par l'hôte distant
ENOBUFS	Le buffer est saturé
EISCONN	La socket est déjà connectée
ENOTCONN	La socket n'est pas connectée
ESHUTDOWN	Transmission après un `shutdown`
ETIMEDOUT	"time-out" expiré
ECONNREFUSED	Connexion refusée
EREMOTERELEASE	L'hôte distant a interrompue sa connexion
EHOSTDOWN	L'hôte n'est pas en marche
EHOSTUNREACH	Pas de route vers cet hôte

9 Exemples de mise en application

Deux exemples de fonctions de connexion à un serveur sont jointes à ce cours : l'une `tcp_open` et l'autre `udp_open`. Toutes les deux renvoient un descripteur de `socket` prêt à l'emploi, ou -1 en cas d'erreur (le message d'erreur doit être généré par les fonctions elles mêmes). En voici les prototypes :

```
int tcp_open(char *host, char *service, int port) ;
int udp_open(char *host, char *service, int port, int conn) ;
```

Description des arguments :

host Une chaîne de caractères qui est l'adresse de l'hôte distant. Cette adresse est soit sous forme décimale pointée, soit c'est un nom symbolique. Elle ne peut pas être nulle.

service Si cette chaîne n'est pas nulle, elle contient le nom symbolique du service distant sur lequel il faut se connecter.

port Le numéro du port distant. S'il est négatif alors `service` est obligatoirement non nulle. Dans le cas où `service` est non nulle et port > 0, cette dernière valeur l'emporte sur celle trouvée dans le fichier `/etc/services`.

conn Cet argument n'a d'usage que pour la fonction `udp_open`. Égal à un, il précise que la socket crée est dédiée au serveur `host` (avec un `bind`), ie on pourra employer `send` et `recv` au lieu de `sendto` et `recvfrom`.

```
1  /* $Id: open_tcp.c,v 1.4 2002/11/24 17:26:11 fla Exp $
2   *
3   *  Fonction "tcp_open" + exemple d'utilisation avec "daytime"
4   *  et "time".
5   */
6  #include        <stdio.h>
7  #include        <unistd.h>
8  #include        <string.h>
9  #include        <errno.h>
10 #include        <sysexits.h>
11 #include        <sys/types.h>
12 #include        <sys/socket.h>
13 #include        <netinet/in.h>
14 #include        <netdb.h>
15 #include        <arpa/inet.h>
16
17 #define         USAGE   "Usage:%s <nom de machine distante>\n"
18
19 struct sockaddr_in      tcp_serv_addr ; /* Adresse Internet du serveur. */
20 struct servent          tcp_serv_info ; /* Infos de "getservbyname".    */
21 struct hostent          tcp_host_info ; /* Infos de "gethostbyname".    */
```

```
1  int
2  tcp_open(char *host,char *service,int port)
3  {
4      int                 fd ;
5      unsigned long       inaddr ;
6      struct hostent      *hp ;
7      struct servent      *sv ;
8
9      bzero ((char *)&tcp_serv_addr, sizeof tcp_serv_addr) ;
10     tcp_serv_addr.sin_family = AF_INET ;
11     if (*service) {
12         if (!(sv = getservbyname(service,"tcp"))) {
13             (void)fprintf(stderr,"tcp_open:service inconnu:%s/tcp\n",service) ;
14             return -1 ;
15         }
16         tcp_serv_info = *sv ;
17         if (port > 0)
18             tcp_serv_addr.sin_port = htons(port) ;
19         else
20             tcp_serv_addr.sin_port = sv->s_port ;
21     }
22     else {
23         if (port <= 0) {
24             (void)fprintf(stderr,"tcp_open:numéro de port non spécifié !\n") ;
25             return -1 ;
26         }
27         tcp_serv_addr.sin_port = htons(port) ;
28     }
29
30     if ((inaddr = inet_addr(host)) != INADDR_NONE) { /* netid.hostid ?   */
31         bcopy((char *)&inaddr,(char *)&tcp_serv_addr.sin_addr,sizeof inaddr)
32         tcp_host_info.h_name = (char *)NULL ;
33     }
34     else {
35         if (!(hp = gethostbyname(host))) {
36           (void)fprintf(stderr,"tcp_open: erreur de nom de machine : %s : %s\n",
37                         host,strerror(errno)) ;
38             return -1 ;
39         }
40         tcp_host_info = *hp ;
41         bcopy(hp->h_addr,(char *)&tcp_serv_addr.sin_addr,hp->h_length) ;
42     }
43
44     if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
45         (void)fprintf(stderr,"tcp_open: impossible de créer la socket !\n") ;
46         return -1 ;
47     }
48     if (connect(fd,(struct sockaddr *)&tcp_serv_addr,sizeof tcp_serv_addr)<0)
49         (void)fprintf(stderr,"tcp_open: impossible de se connecter !\n") ;
50         (void)close(fd) ;
51         return -1 ;
52     }
53
54     return fd ;
55 }
56
57 int
58 main(int argc,char *argv[])
59 {
60     int sfd ;
61     int n ;
62     unsigned long temps ;
63     char buf[256] ;
64
```

```
1       if (argc < 2) {
2           (void)fprintf(stderr,USAGE,argv[0]) ;
3           exit (EX_USAGE) ;
4       }
5
6       /*
7        * Connexion au serveur de date
8        */
9       if ((sfd = tcp_open(argv[1],"daytime",0)) < 0)
10          exit(EX_SOFTWARE) ;
11      if ((n=read(sfd,(void *)buf, sizeof(buf)-1))< 0)
12          perror("read/daytime") ;
13      else {
14          buf[n]='\0';
15          (void)printf("Date(%s)=%s",argv[1],buf) ;
16      }
17      (void)close(sfd) ;
18
19      /*
20       * Connexion au serveur de temps
21       */
22      if ((sfd = tcp_open(argv[1],"",37)) < 0)
23          exit (EX_SOFTWARE) ;
24      if (read(sfd,(void *)&temps, sizeof temps) < 0)
25          perror("read/port 37") ;
26      else
27          (void)printf("Temps(%s)=%lu\n",argv[1],ntohl(temps)) ;
28
29      exit (EX_OK) ;
30  }
```

open_tcp.c

Exemple d'usage :

```
$ ./open_tcp localhost
Date  : Sun Dec  2 16:12:57 2001
Temps : 3216294777
```

Remarque : Pour transmettre la structure d'adresse du serveur à la fonction appelante, La fonction `udp_open` complète une structure globale.

```
1   /* $Id: open_udp.c,v 1.2 2001/12/02 15:08:24 fla Exp $
2    *
3    * Fonction "udp_open" + exemple d'utilisation avec "daytime"
4    * et "time".
5    */
6   #include        <stdio.h>
7   #include        <unistd.h>
8   #include        <string.h>
9   #include        <errno.h>
10  #include        <sysexits.h>
11  #include        <sys/types.h>
12  #include        <sys/socket.h>
13  #include        <netinet/in.h>
14  #include        <netdb.h>
15  #include        <arpa/inet.h>
16  #define         USAGE   "Usage:%s <nom de machine distante>\n"
17  struct sockaddr_in      udp_serv_addr ; /* Adresse Internet du serveur. */
18  struct sockaddr_in      udp_cli_addr ;  /* Adresse Internet du client.  */
19  struct servent          udp_serv_info ; /* Infos de "getservbyname".    */
```

```
1
2   int
3   udp_open(char *host,char *service,int port,int conn)
4   {
5       int fd ;
6       unsigned long inaddr ;
7       struct hostent *hp ;
8       struct servent *sv ;
9
10      bzero ((char *)&udp_serv_addr, sizeof udp_serv_addr) ;
11      udp_serv_addr.sin_family = AF_INET ;
12      if (*service) {
13          if (!(sv = getservbyname(service,"udp"))) {
14              (void)fprintf(stderr,"udp_open:service inconnu:%s/udp\n",service) ;
15              return -1 ;
16          }
17          udp_serv_info = *sv ;
18          if (port > 0)
19              udp_serv_addr.sin_port = htons(port) ;
20          else
21              udp_serv_addr.sin_port = sv->s_port ;
22      }
23      else {
24          if (port <= 0) {
25              (void)fprintf(stderr,"udp_open:numéro de port non spécifié !\n") ;
26              return -1 ;
27          }
28          udp_serv_addr.sin_port = htons(port) ;
29      }
30
31      if ((inaddr = inet_addr(host)) != INADDR_NONE) { /* netid.hostid ?   */
32          bcopy ((char *)&inaddr,(char *)&udp_serv_addr.sin_addr,sizeof inaddr)
33          udp_host_info.h_name = (char *)NULL ;
34      }
35      else {
36          if (!(hp = gethostbyname(host))) {
37              (void)fprintf(stderr,"udp_open:erreur de nom de machine:%s:%s\n",
38                          host,strerror(errno)) ;
39              return -1 ;
40          }
41          udp_host_info = *hp ;
42          bcopy(hp->h_addr,(char *)&udp_serv_addr.sin_addr,hp->h_length) ;
43      }
44      if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
45          (void)fprintf(stderr,"udp_open: impossible de créer la socket !\n") ;
46          return -1 ;
47      }
48      bzero ((char *)&udp_cli_addr,sizeof udp_cli_addr) ;
49      udp_cli_addr.sin_family      = AF_INET ;
50      udp_cli_addr.sin_addr.s_addr = htonl(INADDR_ANY) ;
51      udp_cli_addr.sin_port        = htons(0) ;
52      if (bind(fd,(struct sockaddr *)&udp_cli_addr,sizeof udp_cli_addr) < 0) {
53          (void)fprintf(stderr,"udp_open:erreur avec l'adresse locale !\n") ;
54          (void)close(fd) ;
55          return -1 ;
56      }
57      if (conn == 1) {                    /* Figer l'adresse du serveur.  */
58          if (connect(fd,(struct sockaddr *)&udp_serv_addr,
59                                                  sizeof udp_serv_addr)<0) {
60              (void)fprintf(stderr,"udp_open: connexion impossible avec %s !\n",\
61                                                                      host)
62              (void)close(fd) ;
63              return -1 ;
64          }
65      }
66      return fd ;
67  }
```

```
1
2   int
3   main(int argc,char *argv[])
4   {
5       int sfd ;
6       int n ;
7       unsigned long temps ;
8       char buf[256] ;
9
10      if (argc < 2) {
11          (void)fprintf(stderr,USAGE,argv[0]) ;
12          exit (EX_USAGE) ;
13      }
14
15      /*
16       * Connexion au serveur de date
17       */
18      if ((sfd = udp_open(argv[1],"daytime",0,1)) < 0)
19          exit(EX_SOFTWARE) ;
20      if (send(sfd,(void *)"",1,0) < 0)
21          perror("send") ;
22      else
23          if ((n=recv(sfd,(void *)buf, sizeof buf,0)) < 0)
24              perror("recv") ;
25          else {
26              buf[n]='\0' ;
27              (void)printf("Date(%s)=%s",argv[1],buf) ;
28          }
29      (void)close(sfd) ;
30
31      /*
32       * Connexion au serveur de temps
33       */
34      if ((sfd = udp_open(argv[1],"",37,0)) < 0)
35          exit(EX_SOFTWARE) ;
36
37      n = sizeof udp_serv_addr ;
38      if (sendto(sfd,(void *)"",1,0,(void *)&udp_serv_addr, n) < 0)
39          perror("sendto") ;
40      else
41          if (recvfrom(sfd,(void *)&temps,sizeof temps,0,\
42                                          (void *)&udp_serv_addr, &n) < 0)
43              perror("recvfrom") ;
44          else
45              (void)printf("Temps(%s)=%lu\n",argv[1],ntohl(temps)) ;
46      exit(EX_OK);
47  }
```

open_udp.c

Exemple d'usage :

```
$ ./open_udp localhost
Date  : Sun Dec  2 16:12:17 2001
Temps : 3216294737
```

10 Conclusion et bibliographie

L'usage des fonctions vues dans ce chapitre peut se résumer autour d'une structure d'adresse de `socket` :

	Fonction	Structure	champ
`short sin_family ;`	`gethostbyname`	`hostent`	`h_addrtype`
`u_short sin_port ;`	`getservbyname`	`servent`	`s_port`
`struct in_addr sin_addr ;`	`gethostbyname`	`hostent`	`h_addr`
`char sin_zero[8] ;`	`bzero`		

Outre les pages de manuel (`man`) le lecteur pourra consulter avec grand profit les ouvrages suivants :

RFC 1700 J. Reynolds, J. Postel, "ASSIGNED NUMBERS", 10/20/1994. (Pages=230) (Format=.txt) (Obsoletes RFC1340) (STD 2)

- Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley — "An Advanced 4.4BSD Interprocess Communication Tutorial" — CSRG University of California, Berkeley Berkeley, California 94720. Ce document est reédité dans "Programmer's Supplementary Documents" éditeur O'Reilly, ou sous forme de fichier ascii dans le répertoire :
  `/usr/share/doc/psd/21.ipc/paper.ascii.gz`
- W. Richard Stevens — "Unix Network Programming" — Prentice All — 1990
- W. Richard Stevens — "Unix Network Programming" — Second edition — Prentice All — 1998
- Douglas E. Comer – David L. Stevens — "Internetworking with TCP/IP – Volume III" (BSD Socket version) — Prentice All — 1993

Chapitre XIII

Éléments de serveurs

Dans ce chapitre nous abordons quelques grands principes de fonctionnement des logiciels serveurs. D'abord nous tentons de résumer leurs comportements selon une typologie en quatre modèles génériques, puis nous examinons quelques points techniques remarquables de leur architecture logicielle comme la gestion des taches multiples, des descripteurs multiples, le fonctionnement en arrière plan (les fameux " daemon "), la gestion des logs...

Enfin nous concluons ce chapitre avec une petite présentation du " serveur de serveurs " sous Unix, c'est à dire la commande `inetd`!

1 Type de serveurs

L'algorithme intuitif d'un serveur, déduit des schémas (revoir la page 213) d'utilisation des sockets, pourrait être celui-ci :

1. Créer une socket, lui affecter une adresse locale avec un numéro de port connu des clients potentiels.
2. Entrer dans une boucle infinie qui accepte les requêtes des clients, les lit, formule une réponse et la renvoie au client.

Cette démarche, que nous pourrions qualifier de naïve, ne peut convenir qu'à des applications très simples. Considérons l'exemple d'un serveur de fichiers fonctionnant sur ce mode. Un client réseau qui s'y connecte et télécharge pour 10 Go de données accapare le serveur pendant un temps significativement long, même au regard des bandes passantes modernes. Un deuxième client réseau qui attendrait la disponibilité du même serveur pour transférer 1Ko aurait des raisons de s'impatienter!

1.1 Serveurs itératif et concourant

Un serveur itératif (" iterative server ") désigne une implémentation qui traite une seule requête à la fois.

Un serveur concourant (" concurrent server ") désigne une implémentation capable de gérer plusieurs taches en apparence simul-

tanées. Attention, cette fonctionnalité n'implique pas nécessairement que ces taches concourantes doivent toutes s'exécuter en parallèle...

Dans cette première approche purement algorithmique nous n'abordons pas la mise en œuvre technique, le paragraphe 2 s'y consacrera !

D'un point de vue conceptuel, les serveurs itératifs sont plus faciles à concevoir et à programmer que les serveurs concourants, mais le résultat n'est pas toujours satisfaisant pour les clients. Au contraire, les serveurs concourants, s'ils sont d'une conception plus savante, sont d'un usage plus agréable pour les utilisateurs parceque naturellement plus disponibles.

1.2 Le choix d'un protocole

La pile ARPA nous donne le choix entre TCP et UDP. L'alternative n'est pas triviale. Le protocole d'application peut être complètement bouleversé par le choix de l'un ou de l'autre. Avant toute chose il faut se souvenir des caractéristiques les plus marquantes de l'un et de l'autre.

Mode connecté

Le mode connecté avec TCP est le plus facile à programmer, de plus il assure que les données sont transmises, sans perte.

Par contre, Il établit un circuit virtuel bi-directionnel dédié à chaque client ce qui monopolise une socket, donc un descripteur, et interdit par construction toute possibilité de " broadcast ".

L'établissement d'une connexion et sa terminaison entraîne l'échange de 7 paquets. S'il n'y a que quelques octets à échanger entre le client et le serveur, cet échange est un gaspillage des ressources du réseau.

Il y a plus préoccupant. Si la connexion est au repos , c'est à dire qu'il n'y a plus d'échange entre le client et le serveur, rien n'indique à celui-ci que le client est toujours là ! TCP est silencieux si les deux parties n'ont rien à s'échanger[1].

Si l'application cliente a été interrompue accidentellement[2], rien n'indique au serveur que cette connexion est terminée et il maintient la socket et les buffers associés. Que cette opération se répète un grand nombre de fois et le serveur ne répondra plus, faute de descripteur disponible, voire de mémoire libre au niveau de la couche de transport (allocation au niveau du noyau, en fonction de la mémoire totale et au démarrage de la machine) !

Mode datagramme

Le mode datagramme ou " non connecté " avec UDP hérite de tous les désagréments de IP, à savoir perte, duplication et désordre introduit dans l'ordre des datagrammes.

[1] Nous verrons au chapitre suivant comment on peut modifier ce comportement par défaut

[2] crash du système, retrait du réseau,...

Pourtant malgré ces inconvénients UDP reste un protocole qui offre des avantages par rapport à TCP. Avec un seul descripteur de socket un serveur peut traiter un nombre quelconque de clients sans perte de ressources due à de mauvaises déconnexions. Le " broadcast " et le " multicast " sont possibles.

Par contre les problèmes de fiabilité du transport doivent être gérés au niveau de l'application. Généralement c'est la partie cliente qui est en charge de la reémission de la requête si aucune réponse du serveur ne lui parvient. La valeur du temps au delà duquel l'application considère qu'il doit y avoir reémission est évidement délicate à établir. Elle ne doit pas être figée aux caractéristiques d'un réseau local particulier et doit être capable de s'adapter aux conditions changeantes d'un internet.

1.3 Quatre modèles de serveurs

Deux comportements de serveurs et deux protocoles de transport combinés génèrent quatre modèles de serveurs :

Itératif Data−gramme	Itératif Connecté
Concourant Data−gramme	Concourant Connecté

figure XIII.01

La terminologie **" tache esclave "** employée dans les algorithmes qui suivent se veut neutre quant au choix technologique retenu pour les implémenter. Ce qui importe c'est leur nature concourante avec la **" tache maître "** qui les engendre.

Algorithme itératif - Mode data-gramme :

1. Créer une socket, lui attribuer un port connu des clients.
2. Répéter :
   - Lire une requête d'un client,
   - Formuler la réponse,
   - Envoyer la réponse, conformément au protocole d'application.

Critique :

Cette forme de serveur est la plus simple, elle n'est pas pour autant inutile. Elle est adaptée quand il y a un tout petit volume d'information à échanger et en tout cas sans temps de calcul pour l'élaboration de la réponse. Le serveur de date " daytime " ou le serveur de temps " time " en sont d'excellents exemples.

Algorithme Itératif - Mode connecté :

1. Créer une socket, lui attribuer un port connu des clients.
2. Mettre la socket à l'écoute du réseau, en mode passif.
3. Accepter la connexion entrante, obtenir une socket pour la traiter.
4. Entamer le dialogue avec le client, conformément au protocole de l'application.
5. Quand le dialogue est terminé, fermer la connexion et aller en 3).

Critique :

Ce type de serveur est peu utilisé. Son usage pourrait être dédié à des relations clients/serveurs mettant en jeu de petits volumes d'informations avec la nécessité d'en assurer à coup sûr le transport. Le temps d'élaboration de la réponse doit rester court.

Le temps d'établissement de la connexion n'est pas négligeable par rapport au temps de réponse du serveur, ce qui le rend peu attractif.

Algorithme concourant - Mode datagramme :

Maître :

1. Créer une socket, lui attribuer un port connu des clients.
2. Répéter :
   - Lire une requête d'un client
   - Créer une tache esclave pour élaborer la réponse.

Esclave :

1. Recevoir la demande du client,
2. Élaborer la réponse,
3. Envoyer la réponse au client, conformément au protocole de l'application,
4. Terminer la tache.

Critique :

Si le temps d'élaboration de la réponse est rendu indifférent pour cause de création de processus esclave, par contre le coût de création de ce processus fils est prohibitif par rapport à son usage : formuler une seule réponse et l'envoyer. Cet inconvénient l'emporte généralement sur l'avantage apporté par le " parallélisme ".

Néanmoins, dans le cas d'un temps d'élaboration de la réponse long par rapport au temps de création du processus esclave, cette solution se justifie.

Algorithme concourant - Mode connecté :

Maître :

1. Créer une socket, lui attribuer un port connu des clients.
2. Mettre la socket à l'écoute du réseau, en mode passif.
3. Répéter :
   - Accepter la connexion entrante, obtenir une `socket` pour la traiter,
   - Créer une tache esclave pour traiter la réponse.

Esclave :

1. Recevoir la demande du client,
2. Amorcer le dialogue avec le client, conformément au protocole de l'application,
3. Terminer la connexion et la tache.

Critique :

C'est le type le plus général de serveur parce-qu'il offre les meilleurs caractéristiques de transport et de souplesse d'utilisation pour le client. Il est sur-dimensionné pour les " petits " services et sa programmation soignée n'est pas toujours à la portée du programmeur débutant.

2 Technologie élémentaire

De la partie algorithmique découlent des questions techniques sur le " comment le faire ". Ce paragraphe donne quelques grandes indications très élémentaires que le lecteur soucieux d'acquérir une vraie compétence devra compléter par les lectures indiquées au dernier paragraphe ; la Bibliographie du chapitre (page 260). Notamment il est nécessaire de consulter les ouvrages de **W. R. Stevens** pour la partie système et **David R. Butenhof** pour la programmation des threads.
La suite du texte va se consacrer à éclairer les points suivants :

1. Gestion des " taches esclaves " (paragraphes 2.1, 2.2, 2.3, 2.4)
2. Gestion de descripteurs multiples (paragraphes 2.5, 2.6)
3. Fonctionnement des processus en arrière plan ou " daemon " (paragraphe 3)

2.1 Gestion des " taches esclaves "

La gestion des " taches esclaves " signalées dans le paragraphe 1 induit que le programme " serveur " est capable de gérer plusieurs actions concourantes, c'est à dire qui ont un comportement qui donne l'illusion à l'utilisateur que sa requête est traitée dans un délai raisonnable, sans devoir patienter jusqu'à l'achèvement de la requête précédente.

C'est typiquement le comportement d'un système d'exploitation qui ordonnance des processus entre-eux pour donner à chacun d'eux un peu de la puissance de calcul disponible (" time-sharing ").

La démarche qui parait la plus naturelle pour implémenter ces " taches esclaves " est donc de tirer partie des propriétés mêmes de la gestion des processus du système d'exploitation.

Sur un système Unix l'usage de processus est une bonne solution dans un premier choix car ce système dispose de primitives (APIs) bien rodées pour les gérer, en particulier `fork()`, `vfork()` et `rfork()`.

Néanmoins, comme le paragraphe suivant le rappelle, l'usage de processus fils n'est pas la panacée car cette solution comporte des désagréments. Deux autres voies existent, non toujours valables partout et dans tous les cas de figure. La première passe par l'usage de processus légers ou " threads ", la deuxième par l'usage du signal `SIGIO` qui autorise ce que l'on nomme la programmation asynchrone (cf 2.4).

Pour conclure il faut préciser que des taches esclaves ou concourantes peuvent s'exécuter dans un ordre aléatoire mais pas nécessairement en même temps. Cette dernière caractéristique est celle des taches parallèles. Autrement dit, les taches parallèles sont toutes concourantes mais l'inverse n'est pas vrai. Concrètement il faut disposer d'une machine avec plusieurs processeurs pour avoir, par exemple, des processus (ou des " threads kernel ", si elles sont supportées) qui s'exécutent vraiment de manière simultanée donc sur

des processeurs différents. Sur une architecture mono-processeur, les taches ne peuvent être que concourantes !

2.2 fork, vfork et rfork

Il ne s'agit pas ici de faire un rappel sur la primitive `fork()` examinée dans le cadre du cours Unix, mais d'examiner l'incidence de ses propriétés sur l'architecture des serveurs.

Le résultat du `fork()` est la création d'un processus fils qui ne diffère de son père que par les points suivants :

1. Le code de retour de `fork` : 0 pour le fils, le pid du fils pour le père
2. Le numéro de processus (pid) ainsi que le numéro de processus du processus père (ppid)
3. Les compteurs de temps (`utime, stime`, ...) qui sont remis à zéro
4. Les verrous (`flock`) qui ne sont pas transmis
5. Les signaux en attente non transmis également

Tout le reste est doublonné, notamment la " stack " et surtout la " heap " qui peuvent être très volumineuses et donc rendre cette opération pénalisante voire quasi redhibitoire sur un serveur très chargé (des milliers de processus et de connexions réseaux).

Si le but du `fork` dans le processus fils est d'effectuer un `exec` immédiatement, alors il très intéressant d'utiliser plutôt le `vfork`. Celui-ci ne fait que créer un processus fils sans copier les données. En conséquence, durant le temps de son exécution avant le `exec` le fils partage strictement les mêmes données que le père (à utiliser avec précaution). Jusqu'à ce que le processus rencontre un `exit` ou un `exec`, le processus père reste bloqué (le `vfork` ne retourne pas).

En allant plus loin dans la direction prise par `vfork`, le `rfork`[3] autorise la continuation du processus père après le `fork`, la conséquence est que deux processus partagent le même espace d'adressage simultanément. L'argument d'appel du `rfork` permet de paramètrer ce qui est effectivement partagé ou non. `RFMEM`, le principal d'entre eux, indique au noyau que les deux processus partagent tout l'espace d'adressage.

Si cette dernière primitive est très riche de potentialités[4], elle est également délicate à manipuler : deux (ou plus) entités logicielles exécutant le même code et accédant aux mêmes données sans précaution particulière vont très certainement converger vers de sérieux ennuis de fonctionnement si le déroulement de leurs opérations n'est pas rigoureusement balisé.

En effet, le soucis principal de ce type de programme multi-entités est de veiller à ce qu'aucune de ses composantes ne puisse changer les états de

[3] `clone()` sous Linux

[4] d'ailleurs l'implémentation actuelle des **threads** sous Linux emploie cette primitive avec des avantages et beaucoup d'inconvénients par rapport à ce que prévoit la norme Posix et à la gestion des processus

sa mémoire simultanément. Autrement dit, il faut introduire presque obligatoirement un mécanisme de sémaphore qui permette à l'une des entités logicielles de vérouiller l'accès à telle ou telle ressource mémoire pendant le temps nécessaire à son usage.

Cette opération de " vérouillage " elle-même pose problème, parceque les entités logicielles pouvent s'exécuter en parallèle (architecture multi-processeurs) et donc il est indispensable que l'acquisition du sémaphore qui protège une ressource commune soit une opération atomique, c'est à dire qui s'exécute en une fois, sans qu'il y ait possibilité que deux (ou plus) entités logicielles tentent avec succès de l'acquérir. C'est toute la problèmatique des **mutex**[5].

2.3 Processus légers, les " threads "

Les processus légers ou "**threads**" sont une idée du milieu des années 80. La norme Posix a posé les bases de leur développement durable en 1995 (Posix 1.c), on parle généralement de **pthreads**.

L'idée fondatrice des threads est de ne pas faire de `fork` mais plutôt de permettre le partage de l'espace d'adressage à autant de contextes d'exécution du même code[6] que l'on souhaite.

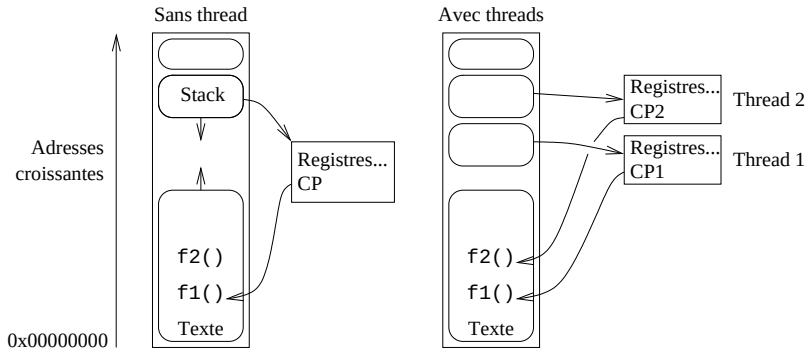


figure XIII.02

Au lieu de créer un nouveau processus on crée une nouvelle thread, ce qui revient (en gros) à ajouter un nouveau contexte d'exécution sur la pile système dans le processus. L'usage de **mutex** (cf paragraphe 2.2) est fortement recommandé pour sérialiser les accès aux " sections critiques " du code.

Sur une machine ayant une architecture mono-processeur, le premier type de threads est suffisant, mais dès que la machine est construite avec une architecture smp[7] (ce qui est de plus en plus le cas avec la banalisation des configurations bi-processeurs et plus) l'usage de threads gérables par le noyau devient beaucoup plus intéressant car il utilise au mieux les ressources de la

[5] " mutual exclusion "

[6] ordre de grandeur de quelques dizaines

[7] " Symmetric Multi Processor "

machine : un même processus pourrait avoir deux threads, une s'exécutant sur chacun des deux processeurs (ou plus bien entendu, s'il y a plus de processeurs).

Le principe étant posé, on distingue plusieurs familles d'implémentation.

D'un coté il y a les threads “ **user land** ” c'est à dire qui sont complètement gérées par le processus utilisateur et de l'autre les threads “ **kernel** ”, qui sont gérées par le noyau. Ces dernières threads sont supportées par les constructeurs de machines à architectures parallèles, traditionnellement Sun (Solaris) et Compaq (ex Digital, avec True64) et plus recement Hewlett-Packard avec la version 11.xx d'HP-UX. Le problème est très complexe et chaque constructeur développe ses propres stratégies.

Du coté des OS libres le problème n'avance pas vite car il monopolise beaucoup de programmeurs de haut niveau, non toujours disponibles pour des taches au long court...

Les threads Linux utilisent `rfork` qui est simple et très efficace. Cette approche n'est pas satisfaisante car chaque thread est exécutée dans un processus différent (pid différent donc) ce qui est contraire aux recommandations POSIX, d'une part, et d'autre par ne permet pas d'utiliser les règles de priorité définies également par POSIX. Une application avec un grand nombre de threads prend l'avantage sur les autres applications par le fait qu'elle consomme en temps cumulé bien plus que les autres processus mono-thread.

Actuellement les threads de FreeBSD sont encore du type **user land** mais un projet très ambitieux et très prometteur (dont l'issue a été repoussée plusieurs fois) est en cours de développement[8].

Conclusion :

Les threads **user land** ne s'exécutent que sur un seul processeur quelle que soit l'architecture de la machine qui les supporte. Sur une machine de type smp il faut que le système d'exploitation supporte les threads **kernel** pour qu'un même processus puisse avoir des sous-taches sur tous les processeurs existants.

[8] `http ://www.freebsd.org/smp/`

2.4 Programmation asynchrone

Les paragraphes qui précèdent utilisent un processus ou une thread pour pouvoir effectuer au moins deux taches simultanément : écouter le réseau et traiter une (ou plusieurs) requête(s). Dans le cas d'un serveur peu sollicité il tout à fait envisageable de mettre en œuvre une autre technique appellée " programmation asynchrone ".

La programmation asynchrone s'appuie sur l'usage du signal, `SIGIO` (`SIGPOLL` sur système V), ignoré par défaut, qui prévient le processus d'une activité sur un descripteur.

La gestion des entrées/sorties sur le descripteur en question est alors traitée comme une exception, par un " handler " de signaux.

Le signal `SIGIO` est ignoré par défaut, il faut demander explicitement au noyau de le recevoir, à l'aide d'un appel à la primitive `fcntl`. Une fois activé, il n'est pas reçu pour les mêmes raisons selon le protocole employé :

UDP :
- Arrivée d'un paquet pour la socket
- Une erreur

TCP :
- Une demande de connexion (attente sur un `accept`) qui arrive
- Une déconnexion
- Une demi-déconnexion (`shutdown`)
- Arrivée de données sur une socket
- Fin de l'émission de données (buffer d'émission vide) sur une socket
- Une erreur

Où l'on voit que cette technique, du moins en TCP, ne peut être envisagée pour que pour des serveurs peu sollicités. Un trop grand nombre d'interruptions possibles nuit à l'efficacité du système (changements de contexte). De plus la distinction entre les causes du signal est difficile à faire, donc ce signal en TCP est quasi inexploitable.

Conclusion :

La dénomination " programmation asynchrone " basée seulement sur l'usae du signal `SIGIO` (versus `SIGPOLL`) est abusive. Pour être vraiment asynchrones, ces opérations de lecture et d'écriture ne devraient pas être assujetties au retour des primitives `read` ou `write`[9]. Cette technique permet l'écriture du code de petits serveurs basé sur le protocole UDP (En TCP les causes de réception d'un tel signal sont trop nombreuses) sans `fork` ni `thread`.

[9]La norme POSIX permet un tel comportement avec les primitives `aio_read` et `aio_write`

2.5 La primitive `select`

Un serveur qui a la charge de gérer simultanément plusieurs sockets (serveur multi-protocoles par exemple, comme `inetd`...) se trouve par construction dans une situation où il doit examiner en même temps plusieurs descripteurs (il pourrait s'agir aussi de tubes de communication).

Il est absolument déconseillé dans cette situation de faire du `polling`. Cette activité consisterait à examiner chaque descripteur l'un après l'autre dans une boucle infinie qui devrait être la plus rapide possible pour être la plus réactive possible face aux requêtes entrantes. Sous Unix cette opération entraîne une consommation exagérée des ressources cpu, au détriment des autres usagers et services.

La primitive `select` (4.3 BSD) surveille un ensemble de descripteurs, si aucun n'est actif le processus est endormi et ne consomme aucune ressource cpu. Dès que l'un des descripteurs devient actif (il peut y en avoir plusieurs à la fois) le noyau réveille le processus et l'appel de `select` rend la main à la procédure appelante avec suffisemment d'information pour que celle-ci puisse identifier quel(s) descripteur(s) justifie(nt) son réveil !

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

int select (int maxfd, fd_set *readfs,
                       fd_set *writefs,
                       fd_set *exceptfs,
                       struct timeval *timeout) ;

FD_ZERO(fd_set *fdset) ;          /* Tous les bits a zero.        */
FD_SET(int fd, fd_set *fdset) ;   /* Positionne 'fd' dans 'fdset' */
FD_CLR(int fd, fd_set *fdset) ;   /* Retire 'fd' de 'fdset'       */
FD_ISSET(int fd, fd_set *fdset) ; /* Teste la presence de 'fd'    */

struct timeval                    /* Cf "time.h"                  */
{
    long tv_sec ;                 /* Nombre de secondes.          */
    long tv_usec ;                /* Nombre de micro-secondes.    */
} ;
```

Le type `fd_set` est décrit dans `<sys/types.h>`, ainsi que les macros `FD_XXX`.

Le prototype de `select` est dans `<sys/time.h>`.

La primitive `select` examine les masques `readfs`, `writefs` et `exceptfs` et se comporte en fonction de `timeout` :

- Si `timeout` est une structure existante (pointeur non nul), la primitive retourne immédiatement après avoir testé les descripteurs. Tous les champs de `timeout` doivent être à 0 (“ polling ” dans ce cas).
- Si `timeout` est une structure existante (pointeur non nul), et si ses champs sont non nuls, `select` retourne quand un des descripteurs est

prêt, et en tout cas jamais au delà de la valeur précisée par `timeout` (cf MAXALARM dans `<sys/param.h>`).
- Si `timeout` est un pointeur NULL, la primitive est bloquante jusqu'à ce qu'un descripteur soit prêt (ou qu'un signal intervienne).

Remarque : `select` travaille au niveau de la micro-seconde, ce que ne fait pas `sleep` (seconde), d'où un usage possible de timer de `précision`.

readfs descripteurs à surveiller en lecture.

writefs descripteurs à surveiller en écriture.

exceptfs Ce champ permet de traiter des evênements exceptionnels sur les descripteurs désignés. Par exemple :
- Données `out-of-band` sur une `socket`.
- Contrôle du statut sur un pseudo-tty maître.

maxfd prend à l'appel la valeur du plus grand descripteur à tester, plus un. Potentiellement un système BSD (4.3 et versions suivantes) permet d'examiner jusqu'à 256 descripteurs.

A l'appel, le programme précise quels sont les descripteurs à surveiller dans `readfs`, `writefs` et `exceptfs`.

Au retour, la primitive précise quels sont les descripteurs qui sont actifs dans les champs `readfs`, `writefs` et `exceptfs`. Il convient donc de conserver une copie des valeurs avant l'appel si on veut pouvoir les réutiliser ultérieurement. La primitive renvoie -1 en cas d'erreur (à tester systématiquement) ; une cause d'erreur classique est la réception d'un signal (`errno==EINTR`).

La macro `FD_ISSET` est utile au retour pour tester quel descripteur est actif et dans quel ensemble.

Le serveur de serveurs `inetd` (page 4) est un excellent exemple d'utilisation de la primitive.

2.6 La primitive poll

La primitive poll (System V) permet la même chose que la primitive select, mais avec une approche différente.

```
#include <poll.h>
int
poll(struct pollfd *fds, unsigned int nfds, int timeout);

struct pollfd {
    int fd ;        /* Descripteur de fichier */
    short events ;  /* Evenements attendus    */
    short revents ; /* Evenements observes    */
} ;
```

La primitive retourne le nombre de descripteurs rendus disponibles pour effectuer des opérations d'entrée/sortie. -1 indique une condition d'erreur. 0 indique l'expiration d'un délai (“ time-out ”).

fds est un pointeur sur la base d'un tableau de `nfds` structures du type `struct pollfd`.

Les champs `events` et `revents` sont des masques de bits qui paramètrent respectivement les souhaits du programmeur et ce que le noyau retourne.

On utilise principalement :

POLLIN

POLLOUT

POLLERR

POLLHUP

nfds Taille du vecteur.

timeout Est un compteur de millisecondes qui précise le comportement de `poll` :

- Le nombre de millisecondes est positif strictement. Quand le temps prévu est écoulé, la primitive retourne dans le code de l'utilisateur même si aucun évênement n'est intervenu.
- Le nombre de millisecondes est `INFTIM` (-1), la primitive est bloquante.
- 0. La primitive retourne immédiatement.

On s'apperçoit immédiatement que la valeur du paramètre de timeout n'est pas compatible ni en forme ni en comportement entre `select` et `poll`.

3 Fonctionnement des daemons

Sous Unix les serveurs sont implémentés le plus souvent sous forme de *daemons*[10]. La raison principale est que ce type de processus est le plus adapté à cette forme de service, comme nous allons l'examiner.

3.1 Programmation d'un *daemon*

Les *daemons* sont des processus ordinaires, mais :
- ils ne sont pas rattachés à un terminal particulier (ils sont en " arrière plan ") ;
- ils s'exécutent le plus souvent avec les droits du " super-utilisateur ", voire, mieux, sous ceux d'un pseudo-utilisateur sans mot de passe ni shell défini.
- ils sont le plus souvent lancés au démarrage du système, lors de l'exécution des shell-scripts de configuration (par exemple à partir de `/etc/rc`) ;
- ils ne s'arrêtent en principe jamais (sauf bien sûr avec le système !).

La conception d'un *daemon* suit les règles suivantes :

1. Exécuter un `fork`, terminer l'exécution du père et continuer celle du fils qui est alors adopté par `init` (traditionnellement c'est le processus N° 1). Le processus fils est alors détaché du terminal, ce que l'on peut visualiser avec un `ps -auxw` (versus `ps -edalf` sur un système V) en examinant la colonne `TT` : elle contient `??` ;
2. Appeler la primtive `setsid` pour que le processus courant devienne " leader " de groupe (il peut y avoir un seul processus dans un groupe) ;
3. Changer de répertoire courant, généralement la racine (/) ou tout autre répertoire à la convenance de l'application ;
4. Modifier le masque de création des fichiers `umask = 0` pour que le troisième argument de `open` ne soit pas biaisé par la valeur du `umask` lorsque cette primitive sert aussi à créer des fichiers ;
5. Fermer tous les descripteurs devenus inutiles, et en particulier 0, 1 et 2 (entrée et sorties standards n'ont plus de sens pour un processus détaché d'un terminal).

le source ci-après est un exemple de programmation de daemon, les appels à la fonction `syslog` font référence à un autre daemon nommé `syslogd` que nous examinons au paragraphe suivant.

[10] Si l'on en croit la première édition de " UNIX System Administration Handbook ", Nemeth, Synder & Seebass, pp 403-404 :"**Many people equate the word 'daemon' with the word 'demon' implying some kind of Satanic connection between UNIX and the underworld. This is an egregious misunderstanding. 'Daemon' is actually a much older form of 'demon' ; daemons have no particular bias towards good or evil, but rather serve to help define a person's character or personality.**"

```
1  /* $Id: diable.c,v 1.2 2002/01/08 21:12:50 fla Exp $
2   *
3   *  Diablotin : exemple de démon miniature...
4   */
5
6  #include        <stdio.h>
7  #include        <stdlib.h>
8  #include        <unistd.h>
9  #include        <sys/types.h>
10 #include        <sys/stat.h>
11 #include        <syslog.h>
12 #include        <errno.h>
13
14 int
15 main()
16 {
17     switch (fork()) {
18         case -1 :              /* erreur du "fork".            */
19             perror("fork") ;
20             exit (1) ;
21
22         case 0  :              /* Le futur "demon".            */
23             (void)printf("Je suis infernal, je me transforme en demon !\n") ;
24             (void)setsid() ;   /* Devenir chef de groupe.      */
25             (void)chdir("/") ; /* Repertoire de travail.       */
26             (void)umask(0) ;
27             (void)close(0) ;
28             (void)close(1) ;
29             (void)close(2) ;
30             openlog("diablotin",LOG_PID|LOG_NDELAY,LOG_USER) ;
31             syslog(LOG_INFO,"Attention, je suis un vrai 'daemon'...\n") ;
32             (void)sleep(1) ;
33             (void)syslog(LOG_INFO,"Je me tue !\n") ;
34             closelog() ;
35             exit(EX_OK) ;
36
37         default :
38             exit(EX_OK) ;
39     }
40 }
```

diable.c

3.2 Daemon `syslogd`

Du fait de leur fonctionnement détaché d'un terminal, les daemons ne peuvent plus délivrer directement de message par les canaux habituels (`perror`...). Pour pallier à cette déficience un daemon est spécialisé dans l'écoute des autres daemons (écoute passive :), il s'agit de `syslogd`[11].

Pour dialoguer avec ce daemon un programme doit utiliser les fonctionnalités que le lecteur trouvera très bien décrites dans "`man syslog`", sinon le paragraphe 3.4 en donne un aperçu rapide.

La *figure 3* suivante schématise le circuit de l'information dans le cas d'une utilisation de `syslogd`.

Le fichier `/etc/syslog.conf` est le fichier standard de configuration du daemon `syslogd`. Il est constitué de lignes de deux champs : un déclencheur

[11] Ce rôle stratégique lui vaut d'être lancé le premier et d'être stoppé le dernier

(*selector*) et une action. Entre ces deux champs un nombre quelconque de tabulations.

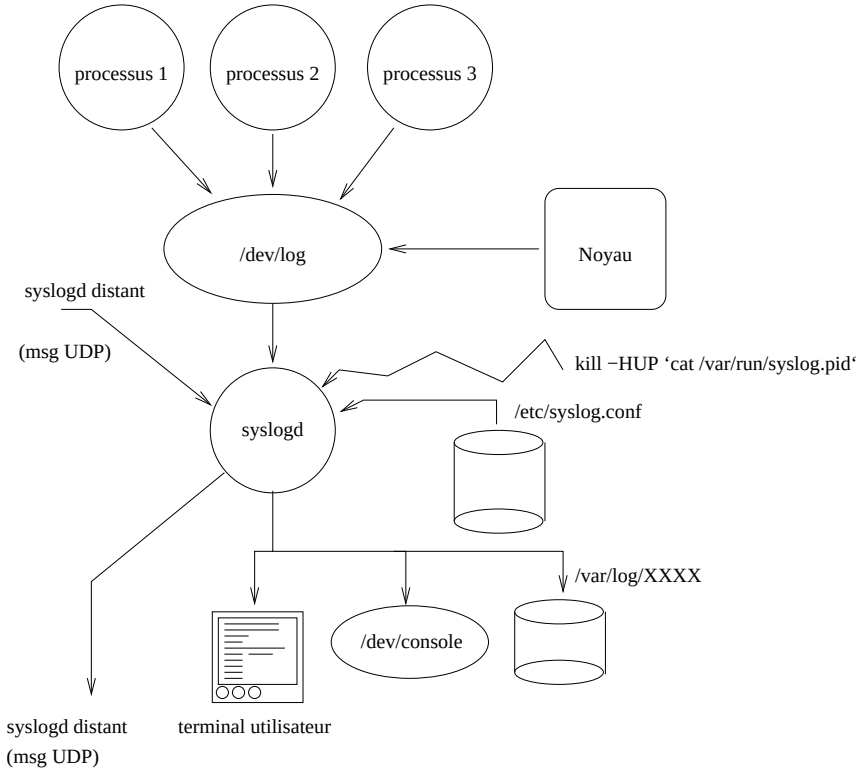


figure XIII.03

Si les conditions du déclencheur sont remplies l'action est exécutée, plus précisement :

Le déclencheur est un filtre qui associe un type de daemon avec un niveau de message. Par exemple `mail.debug` signifie les messages de niveau `DEBUG` pour le système de routage du courrier.

Les mots clefs possibles pour le type de daemon sont `auth, authpriv, cron, daemon, kern, lpr, mail, news, syslog, user, uucp,` et `local0` à `local7`. Une étoile ($\star$) à la place, signifie n'importe quel mot clef.

Le niveau de message est l'un des mots clefs suivants : `emerg, alert, crit, err, warning, notice,` et `debug`. Une étoile ($\star$) signifie n'importe lequel. Un point ($\cdot$) sépare les deux parties du filtre, comme dans `mail.debug`.

Dans les syslog plus évolués l'administrateur a la possibilité de dérouter tous les messages contenant un nom de programme (`!nom_du_prog`) ou un nom de machine (`+nom_de_machine`)

L'action est soit :

- Un fichier désigné par un chemin absolu, comme `/var/log/syslog`.
- Une liste de logins d'utilisateurs, comme `root,fla`...
- Un nom de machine distante (@machine.domaine.fr)
- Tous les utilisateurs connectés avec une étoile $\star$.

3.3 Fichier `syslog.conf`

Exemple de fichier `/etc/syslog.conf` :

```
*.err;kern.debug;auth.notice;mail.crit          /dev/console
*.notice;kern.debug;lpr,auth.info;mail.crit     /var/log/messages
mail.info                                       /var/log/maillog
lpr.info                                        /var/log/lpd-errs
cron.*                                          /var/cron/log
*.err                                           root
*.notice;auth.debug                             root
*.alert                                         root
*.emerg                                         *
*.info                                          |/usr/local/bin/traitinfo
!diablotin
*.*                                             /var/log/diablotin.log
```

Résultat de l'exécution de `diablotin` sur la machine `glups`, et dans le fichier `/var/log/diablotin.log` :

```
...
Jan 27 18:52:02 glups diablotin[20254]: Attention, je suis un vrai 'daemon'...
Jan 27 18:52:03 glups diablotin[20254]: Je me tue !
...
```

3.4 Fonctions `syslog`

Les prototypes et arguments des fonctions :

```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility) ;
void syslog(int priority, const char *message, ...) ;
void closelog(void) ;
```

Comme dans l'exemple de " diablotin ", un programme commence par déclarer son intention d'utiliser le système de `log` en faisant appel à la fonction `openlog` :

`logopt` Donne la possibilité de préciser où le message est envoyés et dans quelle condition.

`facility` Est l'étiquette par défaut des futurs messages envoyés par `syslog`.

`logopt`	description
`LOG_CONS`	Ecriture sur `/dev/console`.
`LOG_NDELAY`	Ouverture immédiate de la connexion avec `syslogd`.
`LOG_PERROR`	Ecriture d'un double du message sur `stderr`.
`LOG_PID`	Identifier chaque message avec le `pid`.

`facility`	description
`LOG_AUTH`	Services d'authentification.
`LOG_AUTHPRIV`	Idem ci-dessus.
`LOG_CRON`	Le daemon qui gère les procédures `batch`.
`LOG_DAEMON`	Tous les daemons du système, comme `gated`.
`LOG_KERN`	Messages du noyau.
`LOG_LPR`	Messages du gestionnaire d'imprimante.
`LOG_MAIL`	Messages du gestionnaire de courrier.
`LOG_NEWS`	Messages du gestionnaire de " news ".
`LOG_SYSLOG`	Messages du daemon `syslogd` lui-même.
`LOG_USER`	Messages des processus utilisateur (defaut).
`LOG_UUCP`	Messages du système de transfert de fichiers.
`LOG_LOCAL0`	Réservé pour un usage local.

Puis chaque appel à la fonction `syslog` est composé d'un message (généré par l'application) et d'un code de priorité, composé d'un niveau d'urgence précisé par le tableau ci-dessous (niveaux décroissants) et d'une étiquette optionnelle, prise dans le tableau ci-dessus ; elle prime alors sur celle précisée lors du `openlog`.

`priority`	description
`LOG_EMERG`	Une condition de " panic system ".
`LOG_ALERT`	Intervention immédiate requise.
`LOG_CRIT`	Problèmes de matériels
`LOG_ERR`	Erreurs.
`LOG_WARNING`	Messages d'avertissement.
`LOG_NOTICE`	Messages qui ne sont pas des erreurs.
`LOG_INFO`	Informations sans conséquence.
`LOG_DEBUG`	Messages pour le debug.

Enfin le `closelog` matérialise la fin d'utilisation de ce système dans le code.

4 Exemple de “ daemon ” inetd

Dans cette partie nous allons étudier un serveur de serveurs nommé `inetd` qui est un très bel exemple pour conclure ce chapitre.

Ce chapitre pourra se prolonger par la lecture du code source C d'`inetd`.

4.1 Présentation de inetd

Sous Unix on peut imaginer facilement que chacun des services réseaux offerts soient programmés comme un daemon, avec une ou plusieurs sockets, chacun surveillant son ou ses ports de communication.

Un tel fonctionnement existe, généralement repéré par le vocabulaire “ stand alone ”. Avec cette stratégie, chaque service comme “ ftp ”, “ rlogin ”, ou encore “ telnet ” fait l'objet d'un processus daemon (“ daemon ”).

Avant la version 4.3 de BSD, c'est comme cela que tous les services fonctionnaient. Le problème est que pour faire fonctionner les services de base du réseau on devait maintenir en mémoire (primaire en “ ram ” ou secondaire sur la zone de “ swap ”) un grand nombre de processus souvent complètement inutiles à un instant donné, simplement au cas ou...

L'inconvénient de cette stratégie est la consommation importante de ressources surtout avec le nombre croissant des services réseaux “ de base ”. De plus, on peut remarquer que lancés au démarrage de la machine, tous ces processus effectuent des opérations similaires (cf 3), seuls diffèrent les traitements propres aux serveurs eux-mêmes c'est à dire ceux qui relèvent du protocole de l'application.

La version 4.3 de BSD a apporté une simplification en introduisant une nouvelle notion, celle de serveur de serveurs : “ The Internet superserver — inetd ”. C'est un daemon que peuvent utiliser tous les serveurs TCP/UDP.

`Inetd` fournit essentiellement deux services principaux :

1. Il permet à un seul processus (celui d'`inetd`) d'attendre de multiples demandes de connexions au lieu d'avoir 1 processus par type de connexion. Cette stratégie réduit d'autant le nombre de processus.
2. Il simplifie l'écriture des serveurs eux-mêmes, puisqu'il gère toute la prise en charge de la connexion. Les serveurs lisent les requêtes sur leur entrée standard et écrivent la réponse sur leur sortie standard.

`Inetd` est un serveur parallèle en mode connecté ou data-gramme. De plus il combine des caractéristiques particulières, puisqu'il est également multi-protocoles et multi-services. Un même service peut y être enregistré et accessible en udp comme en tcp. Bien sûr cela sous entend que le programmeur de ce service ait prévu ce fonctionnement.

Le prix à payer pour une telle souplesse est élevé, `inetd` invoque `fork` puis `exec` pour pratiquement tous les services qu'il offre (cf lecture de code).

Sur les Unix à architecture Berkeley, `inetd` est invoqué au démarrage de la machine, dans les scripts de lancement, `/etc/rc` par exemple. Dès le début de son exécution il se transforme en daemon (cf paragraphe IV.5.3) et lit un

fichier de configuration généralement nommé `/etc/inetd.conf`. Ce fichier est en ASCII, il est lisible normalement par tous, cependant, sur certains sites et pour des raisons de sécurité, il peut ne pas l'être.

La *figure XIII.04* montre l'architecture générale (très simplifiée) de fonctionnement.

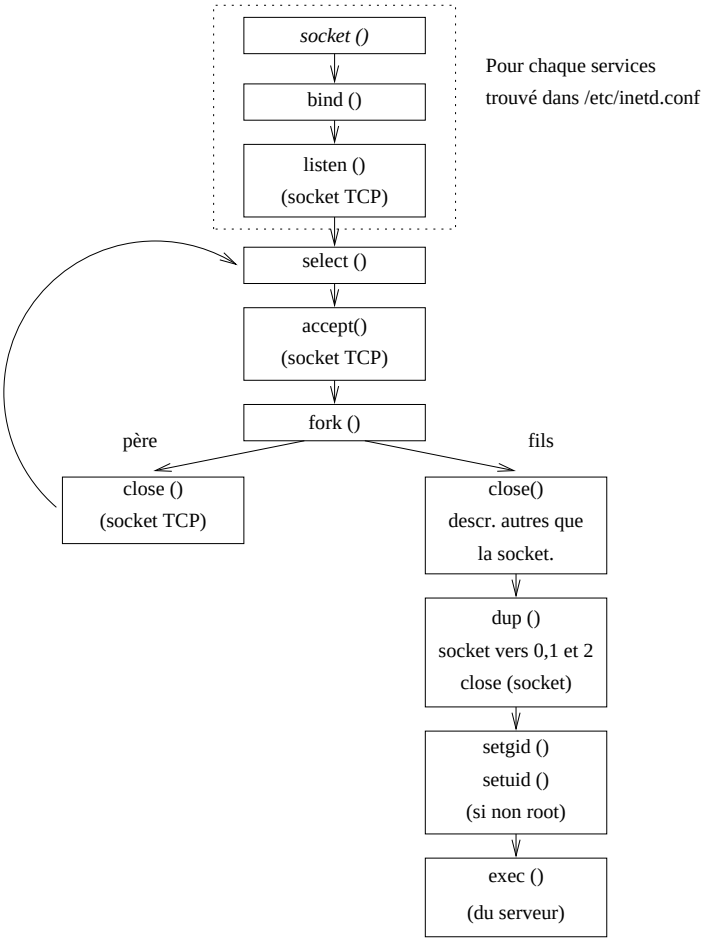


figure XIII.04

Le fichier `/etc/inetd.conf` est organisé de la manière suivante :

- Un # en début de ligne indique un commentaire, comme pour un shell-script.
- Les lignes vides ne sont pas prises en compte.
- Les lignes bien formées sont constituées de 7 champs. Chaque ligne bien formée décrit un serveur.
  Description des champs :

  1. Le nom du service, qui doit également se trouver dans le fichier `/etc/services`. C'est grâce à lui que `inetd` connait le numéro de port à employer
  2. Le type de `socket`, connectée (`stream`) ou non (`dgram`).

3. Le protocole qui doit être `tcp` ou `udp` et doit en tout cas se trouver dans le fichier `/etc/protocols`. Ce dernier fichier donne une correspondance numérique aux différents protocoles.
4. `wait` ou `nowait` suivant que le serveur est itératif ou parallèle.
5. Le nom du propriétaire (pour les droits à l'exécution). Le plus souvent c'est `root`, mais ce n'est pas une règle générale.
6. Le chemin absolu pour désigner l'exécutable du serveur.
7. Les arguments transmis à cet exécutable lors du `exec`, il y en a 20 au maximum dans les implémentations Berkeley de `inetd` (certaines re-écritures, comme celle d'HP, limitent ce nombre).

5 Bibliographie

Pour la partie architecture/configuration des serveurs :

- W. Richard Stevens — “ Unix Network Programming ” — Prentice All — 1990
- W. Richard Stevens — “ Unix Network Programming ” — Volume 1 & 2 — Second edition — Prentice All — 1998
- W. Richard Stevens — “ Advanced Programming in the UNIX Environment ” — Addison–Wesley — 1992
- Douglas E. Comer – David L. Stevens — “ Internetworking with TCP/IP – Volume III ” (BSD Socket version) — Prentice All — 1993
- Stephen A. Rago — “ Unix System V Network Programming ” — Addison–Wesley — 1993
- Man `Unix` de `inetd`, `syslog`, `syslogd` et `syslog.conf`.

Pour la programmation des threads :

- David R. Butenhof — “ Programming with POSIX Threads ” — Addison–Wesley — 1997
- Bradford Nichols, Dirsk Buttlar & Jacqueline Proulx Farell — “ Pthreads programming ” – O’Reilly & Associates, Inc. — 1996

Et pour aller plus loin dans la compréhension des mécanismes internes :

- McKusick, Bostik, Karels, Quaterman — “ The Design and implementation of the 4.4 BSD Operating System ” — Addison Wesley — 1996
- Jim Mauro, Richard McDougall — “ Solaris Internals ” — Sun Microsystems Press — 2001
- Uresh Vahalia — “ Unix Internals, the new frontiers ” — Prentice Hall — 1996

Index