# NAME/NOM

perlipc - Communication inter-processus en Perl (signaux, files d'attente, tubes, sous-processus sûrs, sockets et sémaphores)

# DESCRIPTION

Les équipements CIP (IPC en anglais, NDT) de base de Perl sont construits sur les signaux, tubes nommés, tubes ouverts, routines de socket de Berkeley et appels CIP du System V, du bon vieil Unix. Chacun est utilisé dans des situations légèrement différentes.

# Signaux

Perl utilise un modèle simple de manipulation des signaux : le hachage %SIG contient les noms des handlers (gestionnaires) de signaux installés par l'utilisateur, ou des références vers eux. Ces handlers seront appelés avec un argument qui est le nom du signal qui l'a déclenché. Un signal peut être généré intentionnellement par une séquence particulière au clavier comme control-C ou control-Z, ou vous être envoyé par un autre processus, ou être déclenché automatiquement par le noyau lorsque des événements spéciaux se produisent, comme la fin d'un processus fils, ou l'épuisement de l'espace disponible sur la pile de votre processus, ou la rencontre d'une limite sur la taille d'un fichier.

Par exemple, pour capturer un signal d'interruption, installez un handler comme ceci. Faites aussi peu de choses que possible dans votre handler ; notez comment nous nous débrouillons pour ne faire que placer une variable globale et lever une exception. C'est parce que sur la plupart des systèmes, les bibliothèques ne sont pas réentrantes, en particulier l'allocation de mémoire et les routines d'entrée/sortie. Cela signifie que pratiquement *quoi que ce soit* dans votre handler pourrait en théorie provoquer une faute de segmentation et par conséquent un coredump.

```
sub catch_zap {
    my $signame = shift;
    $shucks++;
    die "Somebody sent me a SIG$signame";
}
$SIG{INT} = 'catch_zap';  # pourrait echouer da
$SIG{INT} = \&catch_zap;  # meilleur strategie
```

Les noms des signaux sont ceux qui sont listés par `kill -l` sur votre système, ou que vous pouvez obtenir via le module Config. Installez une liste @signame indexée par numéro pour avoir le nom et une table %signo indexée par le nom pour avoir le numéro :

```
use Config;
defined $Config{sig_name} || die "No sigs?";
foreach $name (split(' ', $Config{sig_name})) {
    $signo{$name} = $i;
    $signame[$i] = $name;
    $i++;
}
```

Donc pour vérifier si le signal 17 et SIGALRM sont les mêmes, faites juste ceci :

```
print "signal #17 = $signame[17]\n";
if ($signo{ALRM}) {
    print "SIGALRM is $signo{ALRM}\n";
}
```

Vous pouvez aussi choisir d'affecter les chaînes 'IGNORE' ou 'DEFAULT' comme handler, dans ce cas Perl essaiera d'ignorer le signal ou de réaliser l'action par défaut.

Sur la plupart des plateformes Unix, le signal CHLD (parfois aussi connu sous le nom CLD) a un comportement spécial en fonction de la valeur 'IGNORE'. Mettre $SIG{CHLD} à 'IGNORE' sur une telle plateforme a pour effet de ne pas créer de processus zombie lorsque le processus père échoue dans l'attente (wait()) de son fils (i.e. les processus fils sont automatiquement détruits). Appeler wait() avec $SIG{CHLD} placé à 'IGNORE' retourne habituellement -1 sur de telles plateformes.

Certains signaux ne peuvent être ni piégés ni ignorés, comme les signaux KILL et STOP (mais pas TSTP). Une stratégie pour ignorer temporairement des signaux est d'utiliser une déclaration local(), qui sera automatiquement restaurée à la sortie de votre bloc d'instructions (Souvenez-vous que les valeurs local() sont ``héritées'' par les fonctions appelées depuis ce bloc).

```
sub precious {
    local $SIG{INT} = 'IGNORE';
    &more_functions;
}
sub more_functions {
    # interrupts still ignored, for now...
}
```

Envoyer un signal à un identifiant de processus négatif signifie que vous envoyez ce signal à tout le groupe de processus Unix. Ce code envoie un signal hang-up à tous les processus du groupe courant (et place $SIG {HUP} sur IGNORE pour qu'il ne se tue pas lui-même) :

```
{
```

```
        local $SIG{HUP} = 'IGNORE';
        kill HUP => -$$;
        # snazzy writing of: kill('HUP', -$$)
    }
```

Un autre signal intéressant à envoyer est le signal numéro zéro. Il n'affecte en vérité aucun autre processus, mais vérifie s'il est encore en vie ou a changé son UID.

```
    unless (kill 0 => $kid_pid) {
        warn "something wicked happened to $kid_pid
    }
```

Vous pourriez aussi vouloir employer des fonctions anonymes comme handlers de signaux simples :

```
    $SIG{INT} = sub { die "\nOutta here!\n" };
```

Mais cela sera problématique pour les handlers plus compliqués qui ont besoin de se réinstaller eux-mêmes. Puisque le mécanisme de signalisation de Perl est basé sur la fonction `signal(3)` de la bibliothèque C, vous pourriez parfois avoir la malchance d'utiliser des systèmes où cette fonction est ``cassée'', c'est-à-dire qu'elle se comporte selon l'ancienne façon peu fiable du SysV plutôt que la plus récente et plus raisonnable méthode BSD et POSIX. Vous verrez donc les gens sur la défensive écrire des handlers de signaux ainsi :

```
    sub REAPER {
        $waitedpid = wait;
        # maudissez sysV : il nous force non seulem
        # reinstaller le  handler, mais aussi a le
        # apres le wait
        $SIG{CHLD} = \&REAPER;
    }
    $SIG{CHLD} = \&REAPER;
    # maintenant, on fait quelque chose qui forke..
```

Ou même de cette façon plus élaborée :

```
use POSIX ":sys_wait_h";
sub REAPER {
    my $child;
    while (($child = waitpid(-1,WNOHANG)) {
        $Kid_Status{$child} = $?;
    }
    $SIG{CHLD} = \&REAPER;  # maudissez encore
}
$SIG{CHLD} = \&REAPER;
# on fait quelque chose qui forke...
```

La manipulation de signaux est aussi utilisée pour les timeouts sous Unix. Pendant que vous êtes soigneusement protégé par un bloc eval{}, vous installez un handler de signal pour piéger les signaux d'alarme puis programmez de vous en faire délivrer un dans un certain nombre de secondes. Puis vous essayez votre opération bloquante, annulant l'alarme lorsque c'est fait mais pas avant d'être sorti de votre bloc eval{}. Si cela ne marche pas, vous utiliserez die() pour sauter hors du bloc, tout comme vous utiliseriez longjmp() ou throw() dans d'autres langages.

Voici un exemple :

```
eval {
    local $SIG{ALRM} = sub { die "alarm clock r
    alarm 10;
    flock(FH, 2);   # lock d'ecriture blocante
    alarm 0;
};
if ($@ and $@ !~ /alarm clock restart/) { die }
```

Si l'opération chronométrée est system() ou qX(), cette technique peut générer des zombies. Si cela vous préoccupe, vous devrez faire vos propres fork() et exec(), et tuer les processus fils errants.

Pour une manipulation plus complexe des signaux, vous pourriez voir le module du standard POSIX. Il est lamentable que Ceci soit presque entièrement non

documenté, mais le fichier *t/lib/posix.t* de la distribution source de Perl contient quelques exemples.

## Tubes nommés

Un tube nommé (souvent appelé une file d'attente FIFO) est un vieux mécanisme de CIP Unix pour les processus communicant sur la même machine. Il fonctionne tout comme un tube anonyme, normal et connecté, sauf que les processus se donnent rendez-vous en utilisant un nom de fichier et n'ont pas besoin d'être apparentés.

Pour créer un tube nommé, utilisez la commande Unix `mknod(1)` ou sur certains systèmes, mkfifo(1). Elles peuvent ne pas se trouver dans votre chemin d'accès normal.

```
# le systeme renvoit les vals a l'envers, donc
#
$ENV{PATH} .= ":/etc:/usr/etc";
if  (      system('mknod',  $path, 'p')
        && system('mkfifo', $path) )
{
    die "mk{nod,fifo} $path failed";
}
```

Une FIFO est pratique quand vous voulez connecter un processus à un autre sans qu'ils soient apparentés. Lorsque vous ouvrez une FIFO, le programme se bloque jusqu'à ce que quelque chose arrive à l'autre bout.

Par exemple, disons que vous aimeriez que votre fichier *.signature* soit un tube nommé connecté à un programme en Perl. Désormais, chaque fois qu'un programme (comme un gestionnaire de courrier électronique, un lecteur de news, un finger, etc.) essaye de lire le contenu de ce fichier, il se bloque et

votre programme lui fournit une nouvelle signature. Nous utiliserons le test de fichier **-p** qui teste un tube pour déterminer si quelqu'un (ou quelque chose) a accidentellement supprimé notre FIFO.

```
chdir; # a la maison
$FIFO = '.signature';
$ENV{PATH} .= ":/etc:/usr/games";
while (1) {
    unless (-p $FIFO) {
        unlink $FIFO;
        system('mknod', $FIFO, 'p')
            && die "can't mknod $FIFO: $!";
    }
    # la ligne suivante bloque jusqu'a ce qu'il
    open (FIFO, "> $FIFO") || die "can't write
    print FIFO "John Smith (smith\@host.org)\n"
    close FIFO;
    sleep 2;    # pour eviter les signaux dupli
}
```

## AVERTISSEMENT

En installant du code Perl qui traite des signaux, vous vous exposer à deux types de dangers. D'abord, peu de fonctions de la bibliothèque système sont réentrantes. Si le signal provoque une interruption pendant que Perl exécute une fonction (comme `malloc(3)` ou printf(3)), et que votre handler de signal appelle alors la même fonction, vous pourriez obtenir un comportement non prévisible - souvent, un core dump. Ensuite, Perl n'est pas lui-même réentrant aux niveaux les plus bas. Si le signal interrompt Perl pendant qu'il change ses propres structures de données internes, un comportement non prévisible similaire peut apparaître.

Vous pouvez faire deux choses, suivant que vous êtes paranoïaque ou pragmatique. L'approche paranoïaque

est d'en faire aussi peu que possible dans votre handler. Fixez une variable entière existant et ayant déjà une valeur et revenez. Ceci ne vous aide pas si vous êtes au milieu d'un appel système lent, qui se contentera de recommencer. Cela veut dire que vous devez mourir avec die pour sauter (longjump(3)) hors du handler. Même ceci est quelque peu cavalier pour le véritable paranoïaque, qui évite die dans un handler car le système *est* là pour vous attraper. L'approche pragmatique consiste à dire ``je connais les risques, mais je préfère la facilité'', et ensuite faire tout ce qu'on veut dans le handler de signal, en étant prêt à nettoyer les coredumps de temps en temps.

Interdire totalement les handlers interdirait aussi beaucoup de programmes intéressants, y compris virtuellement tout ce qui se trouve dans cette page de manuel, puisque vous ne pourriez plus écrire ne serait-ce qu'un handler SIGCHLD. Ce problème épineux devrait être réglé dans la version 5.005.

## Utilisation de open() pour la CIP

l'instruction open() de base en Perl peut aussi être utilisée pour la communication inter-processus unidirectionnelle en ajoutant un symbole de tube juste avant ou après le second argument de open(). Voici comment démarrer un processus fils dans lequel vous avez l'intention d'écrire :

```
open(SPOOLER, "| cat -v | lpr -h 2>/dev/null")
                 || die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe brok
print SPOOLER "stuff\n";
close SPOOLER || die "bad spool: $! $?";
```

Et voici comment démarrer un processus fils depuis lequel vous désirez lire :

```
open(STATUS, "netstat -an 2>&1 |")
```

```
                    || die "can't fork: $!";
    while (<STATUS>) {
        next if /^(tcp|udp)/;
        print;
    }
    close STATUS || die "bad netstat: $! $?";
```

Si l'on peut etre certain qu'un programme specifique est un script Perl que attend des noms de fichiers dans @ARGV, le programmeur futé peut écrire quelque chose comme ceci :

```
% program f1 "cmd1|" - f2 "cmd2|" f3 < tmpfile
```

et, indépendemment du shell d'où il est appelé, le programme Perl lira dans le fichier *f1*, le processus *cmd1*, l'entrée standard (*tmpfile* dans ce cas), le fichier *f2*, la commande *cmd2*, et finalement dans le fichier *f3*. Plutôt débrouillard, hein ?

Vous pourriez remarquer que l'on pourrait utiliser des accents graves pour obtenir à peu près le même effet que l'ouverture d'un tube en lecture :

```
print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`
die "bad netstat" if $?;
```

Même si ceci est vrai en surface, il est bien plus efficace de traiter le fichier une ligne ou un enregistrement à la fois car alors vous n'avez pas à charger toute la chose en mémoire d'un seul coup. Cela vous donne aussi un contrôle plus fin sur tout le processus, vous laissant tuer le processus fils plus tôt si vous le voulez.

Prenez soin de vérifier à la fois les valeurs de retour de open() et de close(). Si vous *écrivez* dans un tube, vous pouvez aussi piéger SIGPIPE. Sinon, pensez à ce qui se passe lorsque vous ouvrez un tube vers une commande qui n'existe pas : le open() réussira très probablement (il ne fait que refléter le succès du fork ()), mais ensuite votre sortie échouera -

spectaculairement. Perl ne peut pas savoir si la commande a fonctionné parce que votre commande tourne en fait dans un processus séparé dont l'exec() peut avoir échoué. Par conséquent, ceux qui lisent depuis une commande bidon retournent juste une rapide fin de fichier, ceux qui écrivent vers une commande bidon provoqueront un signal qu'ils feraient mieux d'être préparés à gérer. Considérons :

```
open(FH, "|bogus")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: $!";
```

Ceci n'explosera pas avant le close, et ce sera sur un SIGPIPE. Pour l'intercepter, vous pourriez utiliser ceci :

```
$SIG{PIPE} = 'IGNORE';
open(FH, "|bogus")  or die "can't fork: $!";
print FH "bang\n"   or die "can't write: $!";
close FH            or die "can't close: status
```

## Handles de Fichiers

Le processus principal et tous les processus fils qu'il peut générer partagent les mêmes handles de fichiers STDIN, STDOUT et STDERR. Si deux processus essayent d'y accéder en même temps, des choses étranges peuvent se produire. Vous pourriez aussi fermer ou réouvrir les handles de fichiers pour le fils. Vous pouvez contourner cela en ouvrant votre tube avec open(), mais sur certains systèmes cela signifie que le processus fils ne peut pas survivre à son père.

## Processus en Arrière-Plan.

Vous pouvez exécuter une commande en arrière-plan avec :

```
    system("cmd &");
```

Les STDOUT et STDERR de la commande (et peut-être STDIN, selon votre shell) seront les mêmes que ceux du père. Vous n'aurez pas besoin de piéger SIGCHLD à cause du double fork qui se produit (voir plus bas pour plus de détails).

## Dissociation Complète du Fils et de son Père

Dans certains cas (démarrage de processus serveurs, par exemple) vous voudrez complètement dissocier le processus fils de son père. Ceci est souvent appelé une démonisation. Un démon bien élevé fera aussi un chdir() vers le répertoire root (pour qu'il n'empêche pas le démontage du système de fichiers contenant le répertoire à partir duquel il a été lancé) et redirige ses descripteurs de fichier standard vers */dev/null* (pour que des sorties aléatoires ne finissent pas sur le terminal de l'utilisateur).

```
    use POSIX 'setsid';
    sub daemonize {
        chdir '/'                 or die "Can't chdir
        open STDIN, '/dev/null' or die "Can't read
        open STDOUT, '>/dev/null'
                                  or die "Can't write
        defined(my $pid = fork) or die "Can't fork:
        exit if $pid;
        setsid                    or die "Can't start
        open STDERR, '>&STDOUT' or die "Can't dup s
    }
```

Le fork() doit venir avant le setsid() pour s'assurer que vous n'êtes pas un leader de groupe de processus (le setsid() échouera si vous l'êtes). Si votre système ne possède pas la fonction setsid(), ouvrez */dev/tty* et utilisez dessus l'ioctl() TIOCNOTTY à

la place. Voir *tty(4)* pour plus de détails.

Les utilisateurs non unixiens devraient jeter un oeil sur leur module Votre_OS::Process pour avoir d'autres solutions.

## Ouvertures Sûres d'un Tube

Une autre approche intéressante de la CIP est de rendre votre simple programme multiprocessus et de communiquer entre (ou même parmi) eux. La fonction open() acceptera un fichier en argument pour "-|" ou "|-" afin de faire une chose très intéressante : elle forkera un fils connecté au descripteur de fichier que vous venez d'ouvrir. Le fils exécute le même programme que le parent. C'est utile par exemple pour ouvrir de façon sécurisée un fichier lorsque l'on s'exécute sous un UID ou un GID présumé. Si vous ouvrez un tube *vers* minus, vous pouvez écrire dans le descripteur de fichier que vous avez ouvert et votre fils le trouvera dans son STDIN. Si vous ouvrez un tube *depuis* minus, vous pouvez lire depuis le descripteur de fichier tout ce que votre fils écrit dans son STDOUT.

```
use English;
my $sleep_count = 0;
```

```
do {
    $pid = open(KID_TO_WRITE, "|-");
    unless (defined $pid) {
        warn "cannot fork: $!";
        die "bailing out" if $sleep_count++ > 6
        sleep 10;
    }
} until defined $pid;
```

```
if ($pid) {  # parent
    print KID_TO_WRITE @some_data;
    close(KID_TO_WRITE) || warn "kid exited $?"
} else {     # child
```

```
        ($EUID, $EGID) = ($UID, $GID); # suid progs
        open (FILE, "> /safe/file")
            || die "can't open /safe/file: $!";
        while (<STDIN>) {
            print FILE; # child's STDIN is parent's
        }
        exit;  # n'oubliez pas ceci
    }
```

Un autre usage courant de cette construction apparaît lorsque vous avez besoin d'exécuter quelque chose sans interférences de la part du shell. Avec system(), c'est direct, mais vous ne pouvez pas utiliser un tube ouvert ou des accents graves de façon sûre. C'est parce qu'il n'y a pas de moyen d'empêcher le shell de mettre son nez dans vos arguments. À la place, utilisez le contrôle de bas niveau pour appeler exec() directement.

Voici un backtick ou un tube ouvert en lecture sûrs :

```
    # ajoutez un traitement d'erreur comme ci-dessu
    $pid = open(KID_TO_READ, "-|");

    if ($pid) {   # parent
        while (<KID_TO_READ>) {
            # faites quelque chose d'intéressant
        }
        close(KID_TO_READ) || warn "kid exited $?";

    } else {      # fils
        ($EUID, $EGID) = ($UID, $GID); # suid uniqu
        exec($program, @options, @args)
            || die "can't exec program: $!";
        # PASATTEINT
    }
```

Et voici un tube sûr ouvert en écriture :

```
    # ajoutez un traitement d'erreur comme ci-dessu
    $pid = open(KID_TO_WRITE, "|-");
    $SIG{ALRM} = sub { die "whoops, $program pipe b

    if ($pid) {  # parent
        for (@data) {
```

```
            print KID_TO_WRITE;
        }
        close(KID_TO_WRITE) || warn "kid exited $?"
    } else {     # fils
        ($EUID, $EGID) = ($UID, $GID);
        exec($program, @options, @args)
            || die "can't exec program: $!";
        # PASATTEINT
    }
```

Notez que ces opérations sont des forks Unix complets, ce qui signifie qu'ils peuvent ne pas être correctement implémentés sur des systèmes étrangers. De plus, il ne s'agit pas d'un vrai multithreading. Si vous désirez en apprendre plus sur le threading, voyez le fichier *modules* mentionnée plus bas dans la section VOIR AUSSI.

## Communication Bidirectionnelle avec un autre Processus

Même si ceci fonctionne raisonnablement bien pour la communication unidirectionnelle, qu'en est-il pour la communication bidirectionnelle ? La chose la plus évidente que vous aimeriez faire ne marche en fait pas :

```
open(PROG_FOR_READING_AND_WRITING, "| un progra
```

et si vous oubliez d'utiliser le pragma `use warnings` ou l'option **-w**, alors vous raterez complètement le message de diagnostic :

```
Can't do bidirectional pipe at -e line 1.
```

Si vous le voulez vraiment, vous pouvez utiliser la fonction de la bibliothèque standard `open2()` pour attraper les deux bouts. Il existe aussi une `open3()` pour les E/S tridirectionnelles de façon que vous

puissiez aussi récupérer le STDERR de votre fils, mais faire ceci nécessiterait une boucle `select()` maladroite et ne vous permettrait pas d'utiliser les opérations d'entrée normales de Perl.

Si vous jetez un oeil sur son source, vous verrez que `open2()` utilise des primitives de bas niveau, comme les appels `pipe()` et `exec()` d'Unix, pour créer toutes les connexions. Il aurait peut-être été un peu plus efficace d'utiliser socketpair(), mais cela serait devenu encore moins portable que ce ne l'est déjà. Les fonctions `open2()` et `open3()` ont peu de chances de fonctionner ailleurs que sur un système Unix ou quelques autres prétendant être conformes à la norme POSIX.

Voici un exemple d'utilisation de open2():

```
use FileHandle;
use IPC::Open2;
$pid = open2(*Reader, *Writer, "cat -u -n" );
print Writer "stuff\n";
$got = <Reader>;
```

Le problème avec ceci est que le buffering d'Unix va vraiment rendre cette opération pénible. Même si votre descripteur de fichier `Writer` est vidé automatiquement, et même si le processus à l'autre bout reçoit vos données de façon régulière, vous ne pouvez habituellement rien faire pour le forcer à vous les renvoyer rapidement de la même façon. Dans ce cas-ci, nous le pouvons, car nous avons donné à *cat* une option **-u** pour qu'il n'ait pas de tampon. Mais très peu de commandes Unix sont conçues pour fonctionner à travers des tubes, ceci marche donc rarement à moins que vous ayez écrit vous-mêmes le programme se trouvant à l'autre bout du tube à deux sens.

Une solution à ceci est la bibliothèque non standard *Comm.pl*. Elle utilise des pseudo-ttys pour rendre le

comportement de votre programme plus raisonnable :

```
require 'Comm.pl';
$ph = open_proc('cat -n');
for (1..10) {
    print $ph "a line\n";
    print "got back ", scalar <$ph>;
}
```

De cette façon vous n'avez pas besoin de contrôler le code source du programme que vous utilisez. La bibliothèque *Comm* contient aussi les fonctions `expect()` et interact(). Vous trouverez la bibliothèque (et, nous l'espérons, son successeur *IPC::Chat*) dans l'archive CPAN la plus proche de vous, comme il est détaillé dans la section VOIR AUSSI ci-dessous.

Le module plus récent Expect.pm sur le CPAN s'occupe aussi de ce genre de choses. Ce module nécessite deux autres modules du CPAN : IO::Pty et IO::Stty. Il installe un pseudo-terminal pour interagir avec les programmes qui insistent pour discuter avec le pilote de périphérique du terminal. Si votre système fait partie de ceux supportés, cela pourrait être la meilleure solution pour vous.

## Communication Bidirectionnelle avec Vous-même

Si vous voulez, vous pouvez faire des `pipe()` et des `fork()` de bas niveau pour coudre tout cela ensemble à la main. Cet exemple ne se parle qu'à lui-même, mais vous pourriez réouvrir les handles appropriés vers STDIN et STDOUT et appeler d'autres processus.

```
#!/usr/bin/perl -w
# pipe1 - communication bidirectionnelle utilis
# de tubes concus pour les systèmes n'ayant pas
use IO::Handle;  # milliers de lignes juste pou
pipe(PARENT_RDR, CHILD_WTR);                # X
```

```
pipe(CHILD_RDR,  PARENT_WTR);               # X
CHILD_WTR->autoflush(1);
PARENT_WTR->autoflush(1);

if ($pid = fork) {
    close PARENT_RDR; close PARENT_WTR;
    print CHILD_WTR "Parent Pid $$ is sending t
    chomp($line = <CHILD_RDR>);
    print "Parent Pid $$ just read this: `$line
    close CHILD_RDR; close CHILD_WTR;
    waitpid($pid,0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close CHILD_RDR; close CHILD_WTR;
    chomp($line = <PARENT_RDR>);
    print "Child Pid $$ just read this: `$line'
    print PARENT_WTR "Child Pid $$ is sending t
    close PARENT_RDR; close PARENT_WTR;
    exit;
}
```

Mais vous n'avez en fait pas besoin de faire deux appels à des tubes. Si vous disposez de l'appel système socketpair(), il fera tout cela pour vous.

```
#!/usr/bin/perl -w
# pipe2 - communication bidirectionnelle utilis
#  "les choses partagées dans les deux sens son

use Socket;
use IO::Handle;  # milliers de lignes juste pou
# Nous utilisons AF_UNIX car bien que *_LOCAL e
# POSIX 1003.1g de la constante, beaucoup de ma
# l'ont pas encore.
socketpair(CHILD, PARENT, AF_UNIX, SOCK_STREAM,
                           or  die "socketpair

CHILD->autoflush(1);
PARENT->autoflush(1);

if ($pid = fork) {
    close PARENT;
    print CHILD "Parent Pid $$ is sending this\
    chomp($line = <CHILD>);
    print "Parent Pid $$ just read this: `$line
    close CHILD;
```

```
        waitpid($pid,0);
    } else {
        die "cannot fork: $!" unless defined $pid;
        close CHILD;
        chomp($line = <PARENT>);
        print "Child Pid $$ just read this: `$line'
        print PARENT "Child Pid $$ is sending this\
        close PARENT;
        exit;
    }
```

## Sockets : Communication Client/Serveur

Même si elles ne sont pas limitées aux systèmes d'exploitation dérivés d'Unix (e.g., WinSock sur les PC fournit le support des sockets, tout comme le font les bibliothèques VMS), vous ne disposez peut-être pas des sockets sur votre système, auquel cas cette section ne vous fera probablement pas beaucoup de bien. Avec les sockets, vous pouvez à la fois créer des circuits virtuels (i.e., des streams TCP) et des datagrammes (i.e., des paquets UDP). Vous pouvez même peut-être faire plus, en fonction de votre système.

Les appels Perl pour traiter les sockets ont les mêmes noms que les appels système correspondants en C, mais leurs arguments tendent à être différents pour deux raisons : tout d'abord, les handles de fichiers de Perl fonctionnent différemment des descripteurs de fichiers en C. Ensuite, Perl connaît déjà la longueur de ses chaînes, vous n'avez donc pas besoin de passer cette information.

L'un des problèmes majeurs avec l'ancien code des sockets de Perl était qu'il utilisait des valeurs codées en dur pour certaines des constantes, ce qui endommageait sévèrement la portabilité. Si vous avez déjà vu du code qui fait des choses comme fixer

explicitement `$AF_INET = 2`, vous savez que vous avez un gros problème : une approche incommensurablement supérieure est d'utiliser le module `Socket`, qui fournit un accès plus fiable aux différentes constantes et fonctions dont vous aurez besoin.

Si vous n'êtes pas en train d'écrire un couple serveur/client pour un protocole existant comme NNTP or SMTP, vous devriez réfléchir à la façon dont votre serveur saura quand le client a fini de parler, et vice-versa. La plupart des protocoles sont basés sur des messages et des réponses d'une ligne (de façon que l'une des parties sache que l'autre a fini quand un ``\n'' est reçu) ou sur des messages et des réponses de plusieurs lignes qui se finissent par un point ou une ligne vide (``\n.\n'' termine un message ou une réponse).

## Terminateur de Ligne Internet

Le terminateur de ligne de l'Internet est ``\015\012''. Sous certaines variantes ASCII d'Unix, cela peut habituellement être écrit ``\r\n'', mais sous certains autres systèmes, ``\r\n'' peut certaines fois être ``\015\015\012'', ``\012\012\015'', ou quelque chose de complètement différent. Les standards spécifient d'écrire ``\015\012'' pour être conforme (soyez strict sur ce que vous fournissez), mais ils recommandent aussi d'accepter un ``\012'' solitaire en entrée (soyez indulgent sur ce que vous attendez). Nous n'avons pas toujours été très bon à ce sujet dans le code de cette page de manuel, mais à moins que vous ne soyez sur un Mac, cela devrait aller.

## Clients et Serveurs TCP Internet

Utilisez des sockets Internet lorsque vous voulez

effectuer une communication client-serveur qui pourrait s'étendre à des machines hors de votre propre système.

Voici un exemple de client TCP utilisant des sockets Internet :

```
#!/usr/bin/perl -w
use strict;
use Socket;
my ($remote,$port, $iaddr, $paddr, $proto, $lin
$remote  = shift || 'localhost';
$port    = shift || 2345;  # port pris au hasar
if ($port =~ /\D/) { $port = getservbyname($por
die "No port" unless $port;
$iaddr   = inet_aton($remote)               ||
$paddr   = sockaddr_in($port, $iaddr);
$proto   = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)  ||
connect(SOCK, $paddr)    || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}
close (SOCK)            || die "close: $!";
exit;
```

Et voici le serveur correspondant pour l'accompagner. Nous laisserons l'adresse sous la forme INADDR_ANY pour que le noyau puisse choisir l'interface appropriée sur les hôtes à plusieurs pattes. Si vous voulez rester sur une interface particulière (comme le côté externe d'une passerelle ou d'un firewall), vous devriez la remplacer par votre véritable adresse.

```
#!/usr/bin/perl -Tw
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;
$EOL = "\015\012";
sub logmsg { print "$0 $$: @_ at ", scalar loca
```

```
    my $port = shift || 2345;
    my $proto = getprotobyname('tcp');
    $port = $1 if $port =~ /(\d+)/; # nettoie le nu

    socket(Server, PF_INET, SOCK_STREAM, $proto)
    setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
                                        pack("l", 1
    bind(Server, sockaddr_in($port, INADDR_ANY))
    listen(Server,SOMAXCONN)

    logmsg "server started on port $port";

    my $paddr;

    $SIG{CHLD} = \&REAPER;

    for ( ; $paddr = accept(Client,Server); close C
        my($port,$iaddr) = sockaddr_in($paddr);
        my $name = gethostbyaddr($iaddr,AF_INET);

        logmsg "connection from $name [",
                inet_ntoa($iaddr), "]
                at port $port";

        print Client "Hello there, $name, it's now
                        scalar localtime, $EOL;
    }
```

Et voici une version multithread. Elle est multithread en ce sens que comme la plupart des serveurs, elle génère (fork) un serveur esclave pour manipuler la requête du client de façon que le serveur maître puisse rapidement revenir servir un nouveau client.

```
    #!/usr/bin/perl -Tw
    use strict;
    BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
    use Socket;
    use Carp;
    $EOL = "\015\012";

    sub spawn;  # déclaration préalable
    sub logmsg { print "$0 $$: @_ at ", scalar loca

    my $port = shift || 2345;
    my $proto = getprotobyname('tcp');
    $port = $1 if $port =~ /(\d+)/; # nettoie le nu

    socket(Server, PF_INET, SOCK_STREAM, $proto)
```

```
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
                                       pack("l", 1
bind(Server, sockaddr_in($port, INADDR_ANY))
listen(Server,SOMAXCONN)

logmsg "server started on port $port";

my $waitedpid = 0;
my $paddr;

sub REAPER {
    $waitedpid = wait;
    $SIG{CHLD} = \&REAPER;  # abhorrons sysV
    logmsg "reaped $waitedpid" . ($? ? " with e
}

$SIG{CHLD} = \&REAPER;

for ( $waitedpid = 0;
      ($paddr = accept(Client,Server)) || $wait
      $waitedpid = 0, close Client)
{
    next if $waitedpid and not $paddr;
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);

    logmsg "connection from $name [",
            inet_ntoa($iaddr), "]
            at port $port";

    spawn sub {
        print "Hello there, $name, it's now ",
        exec '/usr/games/fortune'           # X
            or confess "can't exec fortune: $!"
    };
}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef
        confess "usage: spawn CODEREF";
    }

    my $pid;
    if (!defined($pid = fork)) {
        logmsg "cannot fork: $!";
        return;
    } elsif ($pid) {
```

```
            logmsg "begat $pid";
            return; # Je suis le pere
        }
        # ou bien je suis le fils - on se multiplie
        open(STDIN,  "<&Client")   || die "can't du
        open(STDOUT, ">&Client")   || die "can't du
        ## open(STDERR, ">&STDOUT") || die "can't d
        exit &$coderef();
    }
```

Ce serveur prend la peine de cloner un processus fils via fork() pour chaque requête entrante. De cette façon, il peut répondre à de nombreuses requêtes en même temps, ce que vous pourriez ne pas toujours désirer. Même si vous n'utilisez pas fork(), le listen() permettra de nombreuses connexions en cours. Les serveurs qui se dupliquent doivent prendre tout particulièrement soin de nettoyer leurs enfants morts (appelés ``zombies'' en jargon unixien), car autrement vous remplirez rapidement votre table des processus.

Nous vous suggérons d'utiliser l'option **-T** pour profiter du taint checking (voir la page de manuel perlsec) même si vous ne tourner pas en setuid ou en setgid. C'est toujours une bonne idée pour les serveurs et les programmes exécutés au nom de quelqu'un d'autre (comme les scripts CGI), car cela diminue le risque que des personnes extérieures compromettent votre système.

Étudions un autre client TCP. Celui-ci se connecte au service TCP ``time'' sur de nombreuses machines différentes et montre à quel point leurs horloges diffèrent du système sur lequel il est exécuté :

```
#!/usr/bin/perl  -w
use strict;
use Socket;

my $SECS_of_70_YEARS = 2208988800;
sub ctime { scalar localtime(shift) }
```

```
    my $iaddr = gethostbyname('localhost');
    my $proto = getprotobyname('tcp');
    my $port = getservbyname('time', 'tcp');
    my $paddr = sockaddr_in(0, $iaddr);
    my($host);
```

```
    $| = 1;
    printf "%-24s %8s %s\n",  "localhost", 0, ctime
```

```
    foreach $host (@ARGV) {
        printf "%-24s ", $host;
        my $hisiaddr = inet_aton($host)     || die
        my $hispaddr = sockaddr_in($port, $hisiaddr
        socket(SOCKET, PF_INET, SOCK_STREAM, $proto
        connect(SOCKET, $hispaddr)          || die
        my $rtime = '    ';
        read(SOCKET, $rtime, 4);
        close(SOCKET);
        my $histime = unpack("N", $rtime) - $SECS_o
        printf "%8d %s\n", $histime - time, ctime($
    }
```

## Clients et Serveurs TCP du Domaine Unix

C'est bien pour les clients et les serveurs Internet, mais pour les communications locales ? Même si vous pouvez utiliser la même configuration, vous ne le voulez pas toujours. Les sockets du domaine Unix sont locales à l'hôte courant, et sont souvent utilisées en interne pour implémenter des tubes. Contrairement aux sockets du domaine Internet, les sockets Unix peuvent se voir dans le système de fichier par ls(1).

```
    % ls -l /dev/log
    srw-rw-rw-  1 root            0 Oct 31 07:23 /d
```

Vous pouvez les tester avec le test de fichier **-S** de Perl :

```
    unless ( -S '/dev/log' ) {
```

```
        die "something's wicked with the print syst
    }
```

Voici un exemple de client du domaine Unix :

```
    #!/usr/bin/perl -w
    use Socket;
    use strict;
    my ($rendezvous, $line);

    $rendezvous = shift || '/tmp/catsock';
    socket(SOCK, PF_UNIX, SOCK_STREAM, 0)      ||
    connect(SOCK, sockaddr_un($rendezvous))    ||
    while (defined($line = <SOCK>)) {
        print $line;
    }
    exit;
```

Et voici un serveur correspondant. Vous n'avez pas besoin de vous soucier ici des stupides terminateurs de réseau car les sockets Unix sont de façon certaine sur l'hôte local (localhost, NDT), et ainsi tout fonctionne bien.

```
    #!/usr/bin/perl -Tw
    use strict;
    use Socket;
    use Carp;

    BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
    sub logmsg { print "$0 $$: @_ at ", scalar loca

    my $NAME = '/tmp/catsock';
    my $uaddr = sockaddr_un($NAME);
    my $proto = getprotobyname('tcp');

    socket(Server,PF_UNIX,SOCK_STREAM,0)       ||
    unlink($NAME);
    bind  (Server, $uaddr)                     ||
    listen(Server,SOMAXCONN)                   ||

    logmsg "server started on $NAME";

    my $waitedpid;

    sub REAPER {
        $waitedpid = wait;
```

```
        $SIG{CHLD} = \&REAPER;  # maudissez sysV
        logmsg "reaped $waitedpid" . ($? ? " with e
    }
    $SIG{CHLD} = \&REAPER;
    for ( $waitedpid = 0;
          accept(Client,Server) || $waitedpid;
          $waitedpid = 0, close Client)
    {
        next if $waitedpid;
        logmsg "connection on $NAME";
        spawn sub {
            print "Hello there, it's now ", scalar
            exec '/usr/games/fortune' or die "can't
        };
    }
```

Comme vous le voyez, il est remarquablement similaire au serveur TCP Internet, à tel point que, en fait, nous avons omis plusieurs fonctions identiques - spawn(), logmsg(), ctime(), et `REAPER()` - qui sont exactement les mêmes que dans l'autre serveur.

Alors pourquoi vouloir utiliser une socket du domaine Unix au lieu d'un tube nommé plus simple ? Parce qu'un tube nommé ne vous procure pas de sessions. Vous ne pouvez pas différencier les données d'un processus de celle d'un autre. Avec la programmation de sockets, vous obtenez une session distincte pour chaque client : c'est pourquoi `accept()` prend deux arguments.

Par exemple, supposons que vous voulez donner accès à un démon serveur de base de données tournant depuis longtemps à des copains sur le web, mais seulement s'ils passent par une interface CGI. Vous aurez un programme CGI petit et simple qui fera toutes les vérifications et les connexions que vous voudrez, puis agira comme un client du domaine Unix et se connectera à votre serveur privé.

# Clients TCP avec IO::Socket

Pour ceux qui préfèrent une interface de plus haut niveau pour la programmation des sockets, le module IO::Socket fournit une approche orientée objet. IO::Socket fait partie de la distribution standard de Perl à partir de la version 5.004. Si vous exploitez une version précédente de Perl, récupérez juste IO::Socket sur le CPAN, où vous trouverez aussi des modules fournissant des interfaces simples pour les systèmes suivants : DNS, FTP, Ident (RFC 931), NIS et NISPlus, NNTP, Ping, POP3, SMTP, SNMP, SSLeay, Telnet, et Time - pour n'en citer que quelques-uns.

## Un Client Simple

Voici un client qui crée une connexion TCP avec le service ``daytime'' sur le port 13 de l'hôte appelé ``localhost'' et affiche tout ce que le serveur veut bien fournir.

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
                    Proto    => "tcp",
                    PeerAddr => "localhost",
                    PeerPort => "daytime(13)",
                )
               or die "cannot connect to daytime
while ( <$remote> ) { print }
```

Lorsque vous lancez ce programme, vous devriez obtenir quelque chose qui ressemble à ceci :

```
Wed May 14 08:40:46 MDT 1997
```

Voici ce que signifient les paramètres du constructeur `new` :

**Proto**

C'est le protocole à utiliser. Dans ce cas, le handle de socket retourné sera connecté à une socket TCP, car nous voulons une connexion orienté flux (stream, NDT), c'est-à-dire une connexion qui se comporte assez comme un bon vieux fichier. Toutes les sockets ne sont pas de ce type. Par exemple, le protocole UDP peut être utilisé pour créer une socket datagramme, pour transmettre des messages.

**PeerAddr**

C'est le nom ou l'adresse Internet de l'hôte distant sur lequel le serveur tourne. Nous aurions pu spécifier un nom plus long comme `"www.perl.com"`, ou une adresse comme `"204.148.40.9"`. Pour les besoins de la démonstration, nous avons utilisé le nom d'hôte spécial `"localhost"`, qui doit toujours désigner la machine sur laquelle le programme tourne. L'adresse Internet correspondant à localhost est `"127.1"`, si vous préférez l'utiliser.

**PeerPort**

C'est le nom du service ou le numéro de port auquel nous aimerions nous connecter. Nous aurions pu nous contenter d'utiliser juste `"daytime"` sur les systèmes ayant un fichier de services système bien configuré, [FOOTNOTE: Le fichier de services système est dans */etc/services* sous Unix] mais juste au cas où, nous avons spécifié le numéro de port (13) entre parenthèses. La simple utilisation du numéro aurait aussi fonctionné, mais les numéros constants rendent nerveux les programmeurs soigneux.

Vous avez remarqué comment la valeur de retour du constructeur `new` est utilisée comme descripteur de fichier dans la boucle `while` ? C'est ce qu'on appelle un descripteur de fichier indirect, une variable scalaire contenant un descripteur de fichier. Vous pouvez l'utiliser de la même façon qu'un descripteur normal.

Par exemple, vous pouvez y lire une ligne de la manière suivante :

```
$line = <$handle>;
```

toutes les lignes restantes de cette façon :

```
@lines = <$handle>;
```

et y écrire une ligne ainsi :

```
print $handle "some data\n";
```

## Un Client Webget

Voici un client simple qui prend en paramètre un hôte distant, puis une liste de documents à récupérer sur cet hôte. C'est un client plus intéressant que le précédent car il commence par envoyer quelque chose avant de récupérer la réponse du serveur,

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host docume
$host = shift(@ARGV);
$EOL = "\015\012";
$BLANK = $EOL x 2;
foreach $document ( @ARGV ) {
    $remote = IO::Socket::INET->new( Proto
                                     PeerAddr
                                     PeerPort
                                    );
    unless ($remote) { die "cannot connect to h
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0" . $B
    while ( <$remote> ) { print }
    close $remote;
}
```

le serveur web gérant le service ``http'', qui est supposé être à son numéro de port standard, le 80. Si

le serveur auquel vous essayez de vous connecter est à un numéro de port différent (comme 1080 ou 8080), vous devez le spécifier par le paramètre nommé `PeerPort => 8080`. La méthode `autoflush` est utilisée sur la socket car autrement le système placerait toutes les sorties que nous y envoyons dans un buffer (si vous êtes sur un Mac, vous devrez aussi remplacer tous les `"\n"` dans votre code qui envoie des données sur le réseau par des `"\015\012"`.)

Se connecter au serveur est seulement la première partie du travail : une fois que vous avez la connexion, vous devez utiliser le langage du serveur. Chaque serveur sur le réseau à son propre petit langage de commande, qu'il attend en entrée. La chaîne que nous envoyons au serveur et qui commence par ``GET'' est dans la syntaxe HTTP. Dans ce cas, nous demandons simplement chaque document spécifié. Oui, nous créons vraiment une nouvelle connexion pour chaque document, même s'ils se trouvent tous sur le même hôte. Vous avez toujours parlé HTTP de cette façon. Les versions les plus récentes des clients web peuvent demander au serveur distant de laisser la connexion ouverte un petit moment, mais le serveur n'est pas tenu d'honorer une telle requête.

Voici un exemple d'exécution de ce programme, que nous appellerons *webget* :

```
% webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html

<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found</H1>
The requested URL /guanaco.html was not found o
</BODY>
```

Ok, ce n'est donc pas très intéressant, car il n'a pas trouvé ce document en particulier. Mais une longue réponse n'aurait pas tenu sur cette page.

Pour avoir une version un peu plus développée de ce programme, vous devriez jeter un oeil sur le programme *lwp-request* inclus dans les modules LWP du CPAN.

## Client Interactif avec IO::Socket

Bien, ceci est parfait si vous voulez envoyer une commande et obtenir une réponse, mais pourquoi ne pas mettre en place quelque chose de pleinement interactif, un peu à la façon dont *telnet* fonctionne ? Ainsi vous pouvez taper une ligne, obtenir la réponse, taper une autre ligne, avoir la réponse, etc.

Ce client est plus compliqué que les deux que nous avons écrits jusqu'à maintenant, mais si vous êtes sur un système qui supporte le puissant appel `fork`, la solution n'est pas si rude que ça. Une fois que vous avez obtenu la connexion au service avec lequel vous désirez discuter, appelez `fork` pour cloner votre processus. Chacun de ces deux processus identiques à un travail très simple à réaliser : le parent copie tout ce qui provient de la socket sur la sortie standard, pendant que le fils copie simultanément sur la socket tout ce qui vient de l'entrée standard. Accomplir la même chose en utilisant un seul processus serait *bien plus* difficile, car il est plus facile de coder deux processus qui font une seule chose, qu'un seul processus qui en fait deux (Ce principe de simplicité est une pierre angulaire de la philosophie Unix, ainsi que du bon génie logiciel, c'est probablement pour cela qu'il s'est étendu à d'autres systèmes).

Voici le code:

```
    #!/usr/bin/perl -w
```

```
use strict;
use IO::Socket;
my ($host, $port, $kidpid, $handle, $line);

unless (@ARGV == 2) { die "usage: $0 host port"
($host, $port) = @ARGV;

# cree une connexion tcp a l'hote et sur le por
$handle = IO::Socket::INET->new(Proto     => "t
                                PeerAddr  => $h
                                PeerPort  => $p
       or die "can't connect to port $port on $

$handle->autoflush(1);      # pour que la sorti
print STDERR "[Connected to $host:$port]\n";

# coupe le programme en deux processus, jumeaux
die "can't fork: $!" unless defined($kidpid = f

# le bloc if{} n'est traverse que dans le proce
if ($kidpid) {
    # copie la socket sur la sortie standard
    while (defined ($line = <$handle>)) {
        print STDOUT $line;
    }
    kill("TERM", $kidpid);          # envoie SI
}
# le bloc else{} n'est traverse que dans le fil
else {
    # copie l'entree standard sur la socket
    while (defined ($line = <STDIN>)) {
        print $handle $line;
    }
}
```

La fonction kill dans le bloc if du père est là pour envoyer un signal à notre processus fils (qui est dans le bloc else) dès que le serveur distant a fermé la connexion de son côté.

Si le serveur distant envoie les données un octet à la fois, et que vous avez besoin de ces données immédiatement sans devoir attendre une fin de ligne (qui peut très bien ne jamais arriver), vous pourriez remplacer la boucle while dans le père par ce qui suit :

```
my $byte;
while (sysread($handle, $byte, 1) == 1) {
    print STDOUT $byte;
}
```

Faire un appel système pour chaque octet que vous voulez lire n'est pas très efficace (c'est le moins qu'on puisse dire) mais c'est le plus simple à expliquer et cela fonctionne raisonnablement bien.

## Serveurs TCP avec IO::Socket

Comme toujours, mettre en place un serveur est un peu plus exigeant que de faire tourner un client. Le modèle est que le serveur crée un type de socket spécial qui ne fait rien à part écouter un port particulier pour attendre l'arrivée d'une connexion. Il le fait en appelant la méthode `IO::Socket::INET->new()` avec des arguments légèrement différents de ceux du client.

### Proto

C'est le protocole à utiliser. Comme pour nos clients, nous spécifierons toujours `"tcp"` ici.

### LocalPort

Nous spécifions un port local dans l'argument `LocalPort`, ce que nous n'avons pas fait pour le client. C'est le nom du service ou le numéro du port dont vous voulez être le serveur (sous Unix, les ports en dessous de 1024 sont réservés au superutilisateur). Dans notre exemple, nous utiliserons le port 9000, mais vous pouvez utiliser n'importe quel port non utilisé couramment sur votre système. Si vous essayez d'en utiliser un qui soit déjà exploité, vous obtiendrez le message ``Address already in use''. Sous Unix, la commande `netstat -a` vous montrera quels services ont des serveurs en cours.

### Listen

le paramètre `Listen` détermine le nombre maximal de connexion en cours que nous pouvons accepter avant de commencer à rejeter les clients arrivants. Pensez-y comme à une file d'attente pour vos appels téléphoniques. Le module Socket de bas niveau a un symbole spécial pour le maximum du système, qui est SOMAXCONN.

### Reuse

Le paramètre `Reuse` est nécessaire pour que nous puissions redémarrer notre serveur manuellement sans devoir attendre quelques minutes que les tampons du système soient vidés.

Une fois que la socket générique du serveur a été créée en utilisant les paramètres listés ci-dessus, le serveur attend qu'un nouveau client s'y connecte. Le serveur se bloque dans la méthode `accept`, qui devient finalement une connexion bidirectionnelle avec le client distant (faites un autoflush sur ce handle pour éviter toute mise en tampon).

Pour être plus gentil avec les utilisateurs, notre serveur leur offre un prompt pour qu'ils entrent leurs commandes. La plupart des serveurs ne font pas cela. À cause du prompt sans caractère de nouvelle ligne, vous devrez utiliser la variante `sysread` du client interactif ci-dessus.

Ce serveur accepte cinq commandes différentes, renvoyant la sortie au client. Notez que contrairement à la plupart des serveurs en réseau, celui-ci ne sait gérer qu'un seul client à la fois. Les serveurs multithreads sont traités dans le chapitre 6 du Chameau (``Programmation en Perl'', NDT, avec un dromadaire sur la couverture).

Voici le code.

```
#!/usr/bin/perl -w
```

```
use IO::Socket;
use Net::hostent;               # pour la version O

$PORT = 9000;                   # choisir quelque c

$server = IO::Socket::INET->new( Proto     => 'tcp
                                 LocalPort => $POR
                                 Listen    => SOMA
                                 Reuse     => 1);

die "can't setup server" unless $server;
print "[Server $0 accepting clients]\n";

while ($client = $server->accept()) {
  $client->autoflush(1);
  print $client "Welcome to $0; type help for comm
  $hostinfo = gethostbyaddr($client->peeraddr);
  printf "[Connect from %s]\n", $hostinfo->name ||
  print $client "Command? ";
  while ( <$client>) {
    next unless /\S/;       # ligne vide
    if    (/quit|exit/i)    { last;
    elsif (/date|time/i)    { printf $client "%s\n
    elsif (/who/i )         { print  $client `who
    elsif (/cookie/i )      { print  $client `/usr
    elsif (/motd/i )        { print  $client `cat
    else {
      print $client "Commands: quit date who cooki
    }
  } continue {
     print $client "Command? ";
  }
  close $client;
}
```

## UDP : Transfert de Message

Un autre type de configuration client-server n'utilise pas de connexions, mais des messages. Les communications UDP nécessitent moins de ressources mais sont aussi moins fiables, car elles ne promettent pas du tout que les messages vont arriver, sans parler de leur ordre ou de l'état dans lequel ils seront. Toutefois, l'UDP offre certains avantages sur TCP, à

commencer par la capacité à faire des ``broadcasts'' ou des ``multicasts'' à destination de tout un tas d'hôtes d'un seul coup (habituellement sur votre sous-réseau local). Si vous êtes très inquiet sur la fiabilité et commencez à mettre en place des vérifications dans votre système de messages, alors vous devriez probablement juste utiliser TCP pour commencer.

Voici un programme UDP similaire au client TCP Internet donné précédemment. Toutefois, au lieu de contacter un hôte à la fois, la version UDP en contactera de nombreux de façon asynchrone en simulant un multicast puis en utilisant `select()` pour attendre une activité sur les E/S pendant dix secondes. Pour faire quelque chose de similaire en TCP, vous seriez obligé d'utiliser un handle de socket différent pour chaque hôte.

```
#!/usr/bin/perl -w
use strict;
use Socket;
use Sys::Hostname;

my ( $count, $hisiaddr, $hispaddr, $histime,
     $host, $iaddr, $paddr, $port, $proto,
     $rin, $rout, $rtime, $SECS_of_70_YEARS);

$SECS_of_70_YEARS      = 2208988800;

$iaddr = gethostbyname(hostname());
$proto = getprotobyname('udp');
$port = getservbyname('time', 'udp');
$paddr = sockaddr_in(0, $iaddr); # 0 means let

socket(SOCKET, PF_INET, SOCK_DGRAM, $proto)   |
bind(SOCKET, $paddr)                          |

$| = 1;
printf "%-12s %8s %s\n",  "localhost", 0, scala
$count = 0;
for $host (@ARGV) {
    $count++;
    $hisiaddr = inet_aton($host)    || die "unk
    $hispaddr = sockaddr_in($port, $hisiaddr);
    defined(send(SOCKET, 0, 0, $hispaddr))    |
```

```
}
$rin = '';
vec($rin, fileno(SOCKET), 1) = 1;
# timeout after 10.0 seconds
while ($count && select($rout = $rin, undef, un
    $rtime = '';
    ($hispaddr = recv(SOCKET, $rtime, 4, 0))
    ($port, $hisiaddr) = sockaddr_in($hispaddr)
    $host = gethostbyaddr($hisiaddr, AF_INET);
    $histime = unpack("N", $rtime) - $SECS_of_7
    printf "%-12s ", $host;
    printf "%8d %s\n", $histime - time, scalar
    $count--;
}
```

## CIP du SysV

Même si la CIP du System V n'est pas aussi largement utilisée que les sockets, elle a des usages intéressants. Vous ne pouvez pas, toutefois, utiliser efficacement la CIP du Système V ou le `mmap()` de Berkeley pour obtenir de la mémoire partagée afin de partager une variable entre plusieurs processus. C'est parce que Perl réallouerait votre chaîne alors que vous ne le voulez pas.

Voici un petit exemple montrant l'utilisation de la mémoire partagée.

```
use IPC::SysV qw(IPC_PRIVATE IPC_RMID S_IRWXU);
$size = 2000;
$id = shmget(IPC_PRIVATE, $size, S_IRWXU) || di
print "shm key $id\n";
$message = "Message #1";
shmwrite($id, $message, 0, 60) || die "$!";
print "wrote: '$message'\n";
shmread($id, $buff, 0, 60) || die "$!";
print "read : '$buff'\n";
# le tampon de shmread est cadre a droite par d
```

```
substr($buff, index($buff, "\0")) = '';
print "un" unless $buff eq $message;
print "swell\n";

print "deleting shm $key\n";
shmctl($id, IPC_RMID, 0) || die "$!";
```

Voici un exemple de sémaphore :

```
use IPC::SysV qw(IPC_CREAT);

$IPC_KEY = 1234;
$id = semget($IPC_KEY, 10, 0666 | IPC_CREAT ) |
print "shm key $id\n";
```

Mettez ce code dans un fichier séparé pour être exécuté dans plus d'un processus. Appelez le fichier *take* :

```
# creation d'un semaphore

$IPC_KEY = 1234;
$id = semget($IPC_KEY,  0 , 0 );
die if !defined($id);

$semnum = 0;
$semflag = 0;

# semaphore 'take'
# on attend que le semaphore soit à zero
$semop = 0;
$opstring1 = pack("s!s!s!", $semnum, $semop, $s

# on incremente le compteur du semaphore
$semop = 1;
$opstring2 = pack("s!s!s!", $semnum, $semop,  $
$opstring = $opstring1 . $opstring2;

semop($id,$opstring) || die "$!";
```

Mettez ce code dans un fichier séparé pour être exécuté dans plus d'un processus. Appelez ce fichier *give* :

```
# 'give' le semaphore
# faites tourner ceci dans le processus origine
# que le second processus continue
```

```
    $IPC_KEY = 1234;
    $id = semget($IPC_KEY, 0, 0);
    die if !defined($id);

    $semnum = 0;
    $semflag = 0;

    # Decremente le compteur du semaphore
    $semop = -1;
    $opstring = pack("s!s!s!", $semnum, $semop, $se

    semop($id,$opstring) || die "$!";
```

Le code de CIP SysV ci-dessus fut écrit il y a longtemps, et il est définitivement déglingué. Pour un look plus moderne, voir le module IPC::SysV qui est inclus dans Perl à partir de la version 5.005.

Un petit exemple démontrant les files de messages SysV :

```
    use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT

    my $id = msgget(IPC_PRIVATE, IPC_CREAT | S_IRWX

    my $sent = "message";
    my $type = 1234;
    my $rcvd;
    my $type_rcvd;

    if (defined $id) {
        if (msgsnd($id, pack("l! a*", $type_sent, $
            if (msgrcv($id, $rcvd, 60, 0, 0)) {
                ($type_rcvd, $rcvd) = unpack("l! a*
                if ($rcvd eq $sent) {
                    print "okay\n";
                } else {
                    print "not okay\n";
                }
            } else {
                die "# msgrcv failed\n";
            }
        } else {
            die "# msgsnd failed\n";
        }
        msgctl($id, IPC_RMID, 0) || die "# msgctl f
    } else {
```

```
        die "# msgget failed\n";
    }
```

# NOTES

La plupart de ces routines retournent silencieusement mais poliment `undef` lorsqu'elles échouent, au lieu de mourir sur-le-champ à cause d'une exception non piégée (en vérité, certaines des nouvelles fonctions de conversion *Socket* font un `croak()` sur les arguments mal définis). Il est par conséquent essentiel de vérifier les valeurs de retour de ces fonctions. Débutez toujours vos programmes utilisant des sockets de cette façon pour obtenir un succès optimal, et n'oubliez pas d'ajouter dans les serveurs l'option **-T** de vérification de pollution dans la ligne #! :

```
#!/usr/bin/perl -Tw
use strict;
use sigtrap;
use Socket;
```

# BUGS

Toutes ces routines créent des problèmes de portabilité spécifiques à chaque système. Comme il est noté par ailleurs, Perl est à la merci de vos bibliothèques C pour la plus grande partie de comportement au niveau système. Il est probablement plus sûr de considérer comme déficiente la sémantique des signaux de SysV et de s'en tenir à de simples opérations TCP et UDP sur les sockets ; e.g., n'essayez pas de passer des descripteurs de fichiers ouverts par une socket datagramme UDP locale si vous voulez que votre code ait une chance d'être portable.

Comme il l'a été mentionné dans la section sur les signaux, puisque peu de fournisseurs offrent des bibliothèques C réentrantes de façon sûre, le

programmeur prudent fera peut de choses dans un handler à part modifier la valeur d'une variable numérique préexistante ; ou, s'il est bloqué dans un appel système lent (redémarrage), il utilisera `die()` pour lever une exception et sortir par un longjmp(3). En fait, même ceci peut dans certains cas provoquer un coredump. Il est probablement meilleur d'éviter les signaux sauf lorsqu'ils sont absolument inévitables. Ce problème sera réglé dans une future version de Perl.

# VOIR AUSSI

La programmation d'applications en réseau est bien plus vaste que ceci, mais cela devrait vous permettre de débuter.

Pour les programmeurs intrépides, le livre indispensable est *Unix Network Programming* par W. Richard Stevens (publié chez Addison-Wesley). Notez que la plupart des livres sur le réseau s'adressent au programmeur C ; la traduction en Perl est laissée en exercice pour le lecteur.

La page de manuel IO::Socket(3) décrit la bibliothèque objet, et la page `Socket(3)` l'interface de bas niveau vers les sockets. Au-delà des fonctions évidentes dans la page de manuel perlfunc, vous devriez aussi jeter un oeil sur le fichier *modules* de votre miroir CPAN le plus proche (Voir la page de manuel perlmodlib ou mieux encore, la *Perl FAQ*, pour une description de ce qu'est le CPAN et pour savoir où le trouver).

La section 5 du fichier *modules* est consacrée au ``Networking, Device Control (modems), and Interprocess Communication'', et contient de nombreux modules en vrac, de nombreux modules concernant le réseau, les opérations de Chat et d'Expect, la programmation CGI, le DCE, le FTP, l'IPC, NNTP, les Proxy, Ptty, les RPC, le SNMP, le SMTP,

Telnet, les Threads, et ToolTalk - pour n'en citer que quelques-uns.

# AUTEUR

Tom Christiansen, avec des vestiges occasionnels de la version originale de Larry Wall et des suggestions des porteurs de Perl.

# TRADUCTION

## Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.6.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

## Traducteur

Roland Trique <*roland.trique@free.fr*>

Conseils : Guy Decoux <*decoux@moulon.inra.fr*>, Guillaume van der Rest <*vanderrest@magnet.fsu.edu*>

## Relecture

Guy Decoux <*decoux@moulon.inra.fr*>, Guillaume van der Rest <*vanderrest@magnet.fsu.edu*>