Using open() for IPC

Perl's basic open() statement can also be used for unidirectional interprocess communication by either appending or prepending a pipe symbol to the second argument to open(). Here's how to start something up in a child process you intend to write to:

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

And here's how to start up a child process you intend to read from:

```
open(STATUS, "netstat -an 2>&1 |")
                || die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS || die "bad netstat: $! $?";
```

If one can be sure that a particular program is a Perl script that is expecting filenames in @ARGV, the clever programmer can write something like this:

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

and irrespective of which shell it's called from, the Perl program will read from the file *f1*, the process *cmd1*, standard input (*tmpfile* in this case), the *f2* file, the *cmd2* command, and finally the *f3* file. Pretty nifty, eh?

You might notice that you could use backticks for much the same effect as opening a pipe for reading:

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

While this is true on the surface, it's much more efficient to process the file one line or record at a time because then you don't have to read the whole thing into memory at once. It also gives you finer control of the whole process, letting you to kill off the child process early if you'd like.

Be careful to check both the open() and the close() return values. If you're *writing* to a pipe, you should also trap SIGPIPE. Otherwise, think of what happens when you start up a pipe to a command that doesn't exist: the open() will in all likelihood succeed (it only reflects the fork()'s success), but then your output will fail--spectacularly. Perl can't know whether the command worked because your command is actually running in a separate process whose exec() might have failed. Therefore, while readers of bogus commands return just a quick end of file, writers to bogus command will trigger a signal they'd better be prepared to handle. Consider:

```
open(FH, "|bogus");
print FH "bang\n";
close FH;
```

Filehandles

Both the main process and the child process share the same STDIN, STDOUT and STDERR filehandles. If both processes try to access them at once, strange things can happen. You may want to close or reopen the filehandles for the child. You can get around this by opening your pipe with open(),

but on some systems this means that the child process cannot outlive the parent.

Background Processes

You can run a command in the background with:

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

The command's STDOUT and STDERR (and possibly STDIN, depending on your shell) will be the same as the parent's. You won't need to catch SIGCHLD because of the double-fork taking place (see below for more details).

Complete Dissociation of Child from Parent

In some cases (starting server processes, for instance) you'll want to complete dissociate the child process from the parent. The following process is reported to work on most Unixish systems. Non-Unix users should check their Your_OS::Process module for other solutions.

- Open /dev/tty and use the TIOCNOTTY ioctl on it. See *tty(4)* for details.
- Change directory to /
- Reopen STDIN, STDOUT, and STDERR so they're not connected to the old tty.
- Background yourself like this:

```
fork && exit;
```

Safe Pipe Opens

Another interesting approach to IPC is making your single program go multiprocess and communicate between (or even amongst) yourselves. The open() function will accept a file argument of either "-|" or "|-" to do a very interesting thing: it forks a child connected to the filehandle you've opened. The child is running the same program as the parent. This is useful for safely opening a file when running under an assumed UID or GID, for example. If you open a pipe *to* minus, you can write to the filehandle you opened and your kid will find it in his STDIN. If you open a pipe *from* minus, you can read from the filehandle you opened whatever your kid writes to his 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 only
```

```
        open (FILE, "> /safe/file")
            || die "can't open /safe/file: $!";
        while (<STDIN>) {
            print FILE; # child's STDIN is parent's KID
        }
        exit;  # don't forget this
    }
```

Another common use for this construct is when you need to execute something without the shell's interference. With `system()`, it's straightforward, but you can't use a pipe open or backticks safely. That's because there's no way to stop the shell from getting its hands on your arguments. Instead, use lower-level control to call `exec()` directly.

Here's a safe backtick or pipe open for read:

```
    # add error processing as above
    $pid = open(KID_TO_READ, "-|");

    if ($pid) {   # parent
        while (<KID_TO_READ>) {
            # do something interesting
        }
        close(KID_TO_READ) || warn "kid exited $?";

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

And here's a safe pipe open for writing:

```
    # add error processing as above
    $pid = open(KID_TO_WRITE, "|-");
    $SIG{ALRM} = sub { die "whoops, $program pipe broke" };

    if ($pid) {  # parent
        for (@data) {
            print KID_TO_WRITE;
        }
        close(KID_TO_WRITE) || warn "kid exited $?";

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

Note that these operations are full Unix forks, which means they may not be correctly implemented on alien systems. Additionally, these are not true multithreading. If you'd like to learn more about threading, see the *modules* file mentioned below in the SEE ALSO section.

Bidirectional Communication with Another Process

While this works reasonably well for unidirectional communication, what about bidirectional communication? The obvious thing you'd like to do doesn't actually work:

```
    open(PROG_FOR_READING_AND_WRITING, "| some program |")
```

and if you forget to use the **-w** flag, then you'll miss out entirely on the diagnostic message:

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

If you really want to, you can use the standard open2() library function to catch both ends. There's also an open3() for tridirectional I/O so you can also catch your child's STDERR, but doing so would then require an awkward select() loop and wouldn't allow you to use normal Perl input operations.

If you look at its source, you'll see that open2() uses low-level primitives like Unix pipe() and exec() to create all the connections. While it might have been slightly more efficient by using socketpair(), it would have then been even less portable than it already is. The open2() and open3() functions are unlikely to work anywhere except on a Unix system or some other one purporting to be POSIX compliant.

Here's an example of using open2():

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

The problem with this is that Unix buffering is really going to ruin your day. Even though your Writer filehandle is auto-flushed, and the process on the other end will get your data in a timely manner, you can't usually do anything to force it to give it back to you in a similarly quick fashion. In this case, we could, because we gave *cat* a **-u** flag to make it unbuffered. But very few Unix commands are designed to operate over pipes, so this seldom works unless you yourself wrote the program on the other end of the double-ended pipe.

A solution to this is the nonstandard *Comm.pl* library. It uses pseudo-ttys to make your program behave more reasonably:

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

This way you don't have to have control over the source code of the program you're using. The *Comm* library also has expect() and interact() functions. Find the library (and we hope its successor *IPC::Chat*) at your nearest CPAN archive as detailed in the SEE ALSO section below.

Sockets: Client/Server Communication

While not limited to Unix-derived operating systems (e.g., WinSock on PCs provides socket support, as do some VMS libraries), you may not have sockets on your system, in which case this section probably isn't going to do you much good. With sockets, you can do both virtual circuits (i.e., TCP streams) and datagrams (i.e., UDP packets). You may be able to do even more depending on your system.

The Perl function calls for dealing with sockets have the same names as the corresponding system calls in C, but their arguments tend to differ for two reasons: first, Perl filehandles work differently than C file descriptors. Second, Perl already knows the length of its strings, so you don't need to pass that information.

One of the major problems with old socket code in Perl was that it used hard-coded values for some of the constants, which severely hurt portability. If you ever see code that does anything like explicitly setting $AF_INET = 2, you know you're in for big trouble: An immeasurably superior

approach is to use the Socket module, which more reliably grants access to various constants and functions you'll need.

If you're not writing a server/client for an existing protocol like NNTP or SMTP, you should give some thought to how your server will know when the client has finished talking, and vice-versa. Most protocols are based on one-line messages and responses (so one party knows the other has finished when a ``\n'' is received) or multi-line messages and responses that end with a period on an empty line (``\n.\n'' terminates a message/response).

Internet TCP Clients and Servers

Use Internet-domain sockets when you want to do client-server communication that might extend to machines outside of your own system.

Here's a sample TCP client using Internet-domain sockets:

```
#!/usr/bin/perl -w
require 5.002;
use strict;
use Socket;
my ($remote,$port, $iaddr, $paddr, $proto, $line);

$remote  = shift || 'localhost';
$port    = shift || 2345;  # random port
if ($port =~ /\D/) { $port = getservbyname($port, 'tcp') }
die "No port" unless $port;
$iaddr   = inet_aton($remote)               || die "no host: $remote";
$paddr   = sockaddr_in($port, $iaddr);

$proto   = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die "socket: $!";
connect(SOCK, $paddr)    || die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}

close (SOCK)            || die "close: $!";
exit;
```

And here's a corresponding server to go along with it. We'll leave the address as INADDR_ANY so that the kernel can choose the appropriate interface on multihomed hosts. If you want sit on a particular interface (like the external side of a gateway or firewall machine), you should fill this in with your real address instead.

```
#!/usr/bin/perl -Tw
require 5.002;
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
use Socket;
use Carp;

sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

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

socket(Server, PF_INET, SOCK_STREAM, $proto)        || die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR,
                                pack("l", 1))   || die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY))        || die "bind: $!";
listen(Server,SOMAXCONN)                            || die "listen: $!";
```

```
    logmsg "server started on port $port";

    my $paddr;

    $SIG{CHLD} = \&REAPER;

    for ( ; $paddr = accept(Client,Server); close Client) {
        my($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, "\n";
    }
```

And here's a multithreaded version. It's multithreaded in that like most typical servers, it spawns (forks) a slave server to handle the client request so that the master server can quickly go back to service a new client.

```
    #!/usr/bin/perl -Tw
    require 5.002;
    use strict;
    BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }
    use Socket;
    use Carp;

    sub spawn;  # forward declaration
    sub logmsg { print "$0 $$: @_ at ", scalar localtime, "\n" }

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

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

    logmsg "server started on port $port";

    my $waitedpid = 0;
    my $paddr;

    sub REAPER {
        $waitedpid = wait;
        $SIG{CHLD} = \&REAPER;  # loathe sysV
        logmsg "reaped $waitedpid" . ($? ? " with exit $?" : '');
    }

    $SIG{CHLD} = \&REAPER;

    for ( $waitedpid = 0;
          ($paddr = accept(Client,Server)) || $waitedpid;
          $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 ", scalar localtime, "\n";
            exec '/usr/games/fortune'
                or confess "can't exec fortune: $!";
        };

    }

    sub spawn {
        my $coderef = shift;

        unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
            confess "usage: spawn CODEREF";
        }

        my $pid;
        if (!defined($pid = fork)) {
            logmsg "cannot fork: $!";
            return;
        } elsif ($pid) {
            logmsg "begat $pid";
            return; # I'm the parent
        }
        # else I'm the child -- go spawn

        open(STDIN,  "<&Client")   || die "can't dup client to stdin";
        open(STDOUT, ">&Client")   || die "can't dup client to stdout";
        ## open(STDERR, ">&STDOUT") || die "can't dup stdout to stderr";
        exit &$coderef();
    }
```

This server takes the trouble to clone off a child version via fork() for each incoming request. That way it can handle many requests at once, which you might not always want. Even if you don't fork(), the listen() will allow that many pending connections. Forking servers have to be particularly careful about cleaning up their dead children (called ``zombies'' in Unix parlance), because otherwise you'll quickly fill up your process table.

We suggest that you use the **-T** flag to use taint checking (see the perlsec manpage) even if we aren't running setuid or setgid. This is always a good idea for servers and other programs run on behalf of someone else (like CGI scripts), because it lessens the chances that people from the outside will be able to compromise your system.

Let's look at another TCP client. This one connects to the TCP ``time'' service on a number of different machines and shows how far their clocks differ from the system on which it's being run:

```
    #!/usr/bin/perl  -w
    require 5.002;
    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(time());

    foreach $host (@ARGV) {
        printf "%-24s ", $host;
        my $hisiaddr = inet_aton($host)     || die "unknown host";
```

```
    my $hispaddr = sockaddr_in($port, $hisiaddr);
    socket(SOCKET, PF_INET, SOCK_STREAM, $proto)   || die "socket: $!";
    connect(SOCKET, $hispaddr)          || die "bind: $!";
    my $rtime = '    ';
    read(SOCKET, $rtime, 4);
    close(SOCKET);
    my $histime = unpack("N", $rtime) - $SECS_of_70_YEARS ;
    printf "%8d %s\n", $histime - time, ctime($histime);
}
```

Unix-Domain TCP Clients and Servers

That's fine for Internet-domain clients and servers, but what about local communications? While you can use the same setup, sometimes you don't want to. Unix-domain sockets are local to the current host, and are often used internally to implement pipes. Unlike Internet domain sockets, Unix domain sockets can show up in the file system with an `ls(1)` listing.

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

You can test for these with Perl's **-S** file test:

```
unless ( -S '/dev/log' ) {
    die "something's wicked with the print system";
}
```

Here's a sample Unix-domain client:

```
#!/usr/bin/perl -w
require 5.002;
use Socket;
use strict;
my ($rendezvous, $line);

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

And here's a corresponding server.

```
#!/usr/bin/perl -Tw
require 5.002;
use strict;
use Socket;
use Carp;

BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }

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

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

logmsg "server started on $NAME";
```

```
    $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 localtime, "\n";
            exec '/usr/games/fortune' or die "can't exec fortune: $!";
        };
    }
```

As you see, it's remarkably similar to the Internet domain TCP server, so much so, in fact, that we've omitted several duplicate functions--spawn(), logmsg(), ctime(), and REAPER()--which are exactly the same as in the other server.

So why would you ever want to use a Unix domain socket instead of a simpler named pipe? Because a named pipe doesn't give you sessions. You can't tell one process's data from another's. With socket programming, you get a separate session for each client: that's why accept() takes two arguments.

For example, let's say that you have a long running database server daemon that you want folks from the World Wide Web to be able to access, but only if they go through a CGI interface. You'd have a small, simple CGI program that does whatever checks and logging you feel like, and then acts as a Unix-domain client and connects to your private server.