



# Network Programming

<http://kickme.to/tiger/>

A large, faint watermark of the Perl logo is centered on the page. The logo is circular and contains the text "PROGRAMMING REPUBLIC OF PERL" around the perimeter. Inside the circle, there is a globe with a palm tree on the left and a star on the right. The text "PERL" is at the bottom, with a small "TM" trademark symbol to its right.

# Network Programming with Perl

Graham Barr  
<[gbarr@pobox.com](mailto:gbarr@pobox.com)>

# Agenda

---

---

- ➡ Introduction
- ➡ Properties of a socket
- ➡ The socket model
- ➡ TCP server/client examples
- ➡ Using UDP
- ➡ UDP server/client examples
- ➡ IO::Socket, with examples
- ➡ Find information about a socket
- ➡ Types of server
- ➡ Common problems
- ➡ Commonly used network protocols
- ➡ Case studies

# Introduction

---

---

- ➡ Perl provides direct access to the C library routines for socket communication. Often, arguments and return values are constants defined in the C header files, or are data structures which Perl will pass in a packed binary format.
- ➡ The Socket module provides these constants and also many functions for packing and unpacking these data structures
- ➡ The IO::Socket module provides a higher level access to creating a socket
- ➡ CPAN contains many modules that provide a very high level access to specific application protocols. e.g. Net::FTP, Net::SMTP, Net::DNS, etc.

# Socket properties

---

---

- ☞ A generic socket has three properties
  - ☞ A type
  - ☞ An address family
  - ☞ A communication protocol

# Socket types

---

---

- ☞ There are many types of socket, these include
  - ☞ Stream - Connection oriented transport
  - ☞ Datagram - Connection-less transport
  - ☞ Raw - Often used to talk directly to the IP layer. For example, ping uses a raw socket to send ICMP packets
- ☞ The system socket functions use numbers to represent these. The Socket module exports constants for these

```
use Socket qw(SOCK_STREAM SOCK_DGRAM SOCK_RAW);
```

# Address families

---

---

- ☞ Available address families include
  - ☞ AF\_UNIX - Communication is limited to a single machine. Sometimes called AF\_LOCAL or AF\_FILE. The address is a filesystem path on the local machine.
  - ☞ AF\_INET - This address family uses the IP protocol to communicate with other machines over a network. The address is 193.168.1.200/21
  - ☞ Others include AF\_APPLETALK, AF\_IPX, AF\_DECnet ...
- ☞ These are represented as numbers and the Socket module exports constants for these

```
use Socket qw(AF_UNIX AF_INET AF_APPLETALK);
```

# Communication protocols

---

---

- ☞ There are two protocols that are mainly used
  - ☞ TCP is used with a stream socket to provide a reliable, sequenced, flow-controlled channel of communication.
  - ☞ UDP is used with a datagram socket and delivers datagrams to other endpoints. Message boundaries are preserved, but sequence is not and delivery is not guaranteed.
- ☞ Protocols are represented as numbers, but are not available as constants. Perl provides some functions for translating protocol names to numbers and visa-versa.

```
$number = getprotobyname( 'tcp' );  
$name   = getprotobyname( 6 );
```



# The socket model

---

---

## ☞ The Server

- ☞ Creates a generic socket with `socket`
- ☞ Binds to a known address with `bind`
- ☞ Tell system to watch for incoming connections with `listen`
- ☞ Waits for a connection with `accept` or `select`

# The socket model (*cont.*)

---

---

- ☞ The client
  - ☞ Creates generic socket with `socket`
  - ☞ Binds to an address with `bind`
  - ☞ Connects to server with `connect`, using the known address. This establishes the connection.

# The socket model (*cont.*)

---

---

- ➡ The server is notified of the new connection.
  - ↪ Either `accept` returns or `select` will report the socket as readable.
- ➡ Server and Client communicate.
- ➡ Server and Client `close` the socket to break the connection.

# Creating a socket

---

---

☞ To create a socket you need to know all three properties about the socket.

☞ import required constants from the Socket module

```
use Socket qw(AF_INET SOCK_STREAM);
```

☞ Obtain the value for the protocol

```
$proto = getprotobyname('tcp');
```

☞ Create the socket

```
socket(SOCK, AF_INET, SOCK_STREAM, $proto)  
    || die "socket: $!";
```

# Binding the socket

---

---

- ➡ `bind` takes two arguments, the first is the socket and the second is a packed address.
- ➡ The `Socket` module provides functions for packing and unpacking addresses.
- ➡ `sockaddr_in` allows you to either pack or unpack an `AF_INET` socket address. In a scalar context it packs and in a list context it will unpack.

```
$paddr = sockaddr_in($port, $inaddr);  
($port, $inaddr) = sockaddr_in($paddr);
```

- ➡ If the use of context here disturbs you then you can explicitly call `pack_sockaddr_in` and `unpack_sockaddr_in`.

# Binding the socket (*cont.*)

---

---

- ➡ Many protocols, for example FTP and Telnet, use well known port numbers. But, like communication protocols, these are not provided by constants but by lookup routines

```
$port      = getservbyname('ftp','tcp');  
$service  = getservbyport(21, 'tcp');
```

```
($name, $aliases, $port, $proto)  
  = getservbyname('ftp', 'tcp');
```

```
($name, $aliases, $port, $proto)  
  = getservbyport(21, 'tcp');
```

- ➡ If you do not care which port the socket is bound to, you can use 0 and the kernel will select a free port number.

# Binding the socket (*cont.*)

---

---

- ➡ Besides the port, `sockaddr_in` also needs an IP address.
- ➡ If you do not want to bind the socket to a particular interface the you can use `INADDR_ANY`.
- ➡ If you want to bind the socket to a particular interface then you must pass a packed IP address.
- ➡ The Socket module provides `inet_aton` and `inet_ntoa` to pack and unpack IP addresses.

```
$ipaddr = inet_aton("localhost");  
$quad  = inet_ntoa($ipaddr);
```

- ➡ Not calling `bind` is treated the same as calling `bind` with a port of 0 and `INADDR_ANY`. This is not normally useful for a server.

# Binding the socket (*cont.*)

---

---

- ☞ If the socket is of type `AF_UNIX` the the socket addresses can be manipulated with `sockaddr_un`, `pack_sockaddr_un` and `unpack_sockaddr_un`.

```
$paddr = sockaddr_un( "/tmp/sock" );  
($path) = sockaddr_un( $paddr );
```



# Listen for connections

---

---

- ☞ On the server side you must tell the system that you want to wait for incoming connections. This is done with the `listen` function

```
listen(SOCK, 10);
```

- ☞ The second argument is the queue size.
- ☞ `SOMAXCONN`, which is exported by `Socket`, is the maximum value your system will accept.
- ☞ On most systems, passing a value of 0 will cause the value `SOMAXCONN` to be used.
- ☞ On most systems, passing a value greater than `SOMAXCONN` will silently be ignored and the value of `SOMAXCONN` will be used.

# The client side

---

---

- ☞ Creating a socket on the client side is similar.

```
$proto = getprotobyname('tcp');  
socket(SOCK, AF_INET, SOCK_STREAM, $proto)  
    or die "socket: $!";
```

- ☞ Some servers may require a client to bind to a particular port. Some require use of a port number less than 1024, which on UNIX can only be performed by root.

```
$sin = sockaddr_in($port, INADDR_ANY);  
bind(SOCK, $sin) or die "bind: $!";
```

- ☞ As with the server side, if `bind` is not called, the kernel will select a port number when `connect` is called. The address will be the address of the interface used to route to the server.

# Connecting to the server

---

---

- ➡ Once a socket has been created on the client it must connect to the server at the known address.
- ➡ `connect` takes two arguments, the socket and a packed socket address for the port on the remote host to connect to

```
$port = getservbyname('daytime','tcp');  
$inaddr = inet_aton('localhost');  
$paddr = sockaddr_in($port, $inaddr);
```

```
connect(SOCK, $paddr) or die "connect: $!";
```

# Connecting to the server (*cont.*)

---

---

- ☞ `connect` has a built-in timeout value before it will return a failure.
- ☞ On many systems this timeout can be very long.
- ☞ One approach to shorten this time is to use an alarm.

```
eval {  
    local $SIG{ALRM} = sub { die "Timeout" };  
    alarm 20; # a 20 second timeout  
    my $val = connect(SOCK, $paddr);  
    alarm 0;  
    $val;  
} or die "connect: $!";
```

- ☞ Another approach is to use non-blocking IO.

# Accepting a client connection

---

---

- ➡ When a client calls `connect`, the server will be notified and can then accept the connection.

```
$peer = accept(CLIENT, SOCK);
```

- ➡ This will create a perl filehandle `CLIENT` which can be used to communicate with the client.

- ➡ `$peer` will be a packed address of the client's port, and can be unpacked with

```
($port,$inaddr) = sockaddr_in($peer);  
$dotted_quad = inet_ntoa($inaddr);
```

# example protocols

---

---

- ☞ The daytime protocol is used to keep the time on two machines in sync.
  - ☞ When the server gets a request from a client, it responds with a string which represents the date on the server.
- ☞ The echo protocol can be used to indicate that a machine is up and running. It can also be used to check the quality of the network.
  - ☞ When the server receives anything, it responds by sending it back where it came from.

# TCP daytime client

---

---

```
#!/bin/perl -w
# Example of a TCP daytime client using perl calls directly

use Socket qw(AF_INET SOCK_STREAM inet_aton sockaddr_in);

# get protocol number
$proto = getprotobyname('tcp');

# create the generic socket
socket(SOCK, AF_INET, SOCK_STREAM, $proto) or die "socket: $!";

# no need for bind here

# get packed address for host
$addr = inet_aton('localhost');

# get port number for the daytime protocol
$port = getservbyname('daytime', 'tcp');

# pack the address structure for connect
$paddr = sockaddr_in($port, $addr);
```

# TCP daytime client (*cont.*)

---

---

```
# connect to host
connect(SOCK, $paddr) or die "connect: $!";

# get and print the date
print <SOCK>;

# close the socket
close(SOCK) || die "close: $!";
```



# TCP daytime server

---

---

```
#!/bin/perl -w
# Example of a daytime TCP server using perl functions

use Socket qw(INADDR_ANY AF_INET SOMAXCONN SOCK_STREAM sockaddr_in);

# Get protocol number
my $proto = getprotobyname('tcp');

# Create generic socket
socket(SOCK, AF_INET, SOCK_STREAM, $proto) or die "socket: $!";

# Bind to the daytime port on any interface
my $port = getservbyname('daytime','tcp');
my $paddr = sockaddr_in($port, INADDR_ANY);

bind(SOCK, $paddr) or die "bind: $!";

# Notify the kernel we want to accept connections
listen(SOCK, SOMAXCONN) or die "listen: $!";

while(1) {
    if(accept(CLIENT, SOCK)) {
        print CLIENT scalar localtime, "\n";
        close CLIENT;
    }
}
```

# Using UDP

---

---

- ☞ With UDP, it is not normally required that the client connect to the server.
- ☞ Sending data is performed with `send` instead of `syswrite`.
  - ☞ `send`, unlike `syswrite`, always sends the whole buffer passed.
  - ☞ `send` takes two extra arguments, flags and the destination address. On a connected UDP socket the destination address is optional.

```
send(SOCK, $buffer, 0, $paddr);
```

# Using UDP (*cont.*)

---

---

- ☞ Reading data is performed with `recv` instead of `sysread`.

```
recv(SOCK, $buffer, $length, $flags);
```

- ☞ `recv` will read the next datagram. If the length of the datagram is longer than `$length`, then the rest of the datagram will be discarded.
- ☞ The return value from `recv` is the packed address of the sender.

# Using UDP (*cont.*)

---

---

- ☞ The flags argument can be set to MSG\_PEEK to read data from the next datagram without removing it from the input queue. This is useful if you do not know the size of the incoming datagrams.

```
recv(SOCK, $buffer, 4, MSG_PEEK);  
$length = unpack("N", $buffer);  
recv(SOCK, $buffer, $length, 0);
```

# UDP daytime client

---

---

```
#!/bin/perl -w
# Example of a daytime UDP client using perl calls directly

use Socket qw(AF_INET SOCK_DGRAM inet_aton sockaddr_in);

# get protocol number
$proto = getprotobyname('udp');

# create the generic socket
socket(SOCK, AF_INET, SOCK_DGRAM, $proto) or die "socket: $!";

# no need for bind here

# get packed address for host
$addr = inet_aton('localhost');

# get port number for the daytime protocol
$port = getservbyname('daytime', 'udp');

# pack the address structure for send
$paddr = sockaddr_in($port, $addr);
```

# UDP daytime client (*cont.*)

---

---

```
# send empty packet to server
send(SOCK,"", 0, $paddr) or die "send: $!";

$SIG{ALRM} = sub { die "Timeout" };

eval {
    recv(SOCK, $date, 1024, 0) or die "recv: $!\n";
    print $date,"\n";
} or warn $@;

close(SOCK);
```

# UDP daytime server

---

---

```
#!/bin/perl -w
# Example of a daytime UDP server using perl functions

use Socket qw(INADDR_ANY AF_INET SOMAXCONN SOCK_DGRAM sockaddr_in);

# Get protocol number
my $proto = getprotobyname('udp');

# Create generic socket
socket(SOCK, AF_INET, SOCK_DGRAM, $proto) or die "socket: $!";

# Bind to the daytime port on any interface
my $port = getservbyname('daytime', 'udp');
my $paddr = sockaddr_in($port, INADDR_ANY);

bind(SOCK, $paddr) or die "bind: $!";

# no listen() as that is a SOCK_STREAM call()

$rin = "";
vec($rin, fileno(SOCK), 1) = 1;

while (select($rout=$rin, undef, undef, undef)) {
    $from = recv(SOCK, $buffer, 1, 0) or next;
    send(SOCK, scalar localtime, 0, $from) || die "send: $!";
}
```

# IO::Socket

---

---

- ☞ IO::Socket is designed to make the creation of sockets easier.
- ☞ Although IO::Socket defines methods for most socket operations, it is **not** recommended that you use those which directly map onto perl functions.
- ☞ The IO::Socket object can be used anywhere you would normally use a filehandle.



# Create a socket with IO::Socket

---

---

- ➡ The constructor for IO::Socket takes a list of name => value pairs.
- ➡ IO::Socket->new only knows about one, which tells it the domain of the socket. Each domain is implemented in a different class and support their own name => value pairs.
- ➡ There are two ways in which a socket can be created. Both of the following do the same

```
$sock1 = IO::Socket->new(  
    Domain => 'INET', @args);  
$sock2 = IO::Socket::INET->new(@args);
```

# IO::Socket::INET

---

---

- ☞ An INET domain socket supports the following named arguments
  - ☞ PeerAddr - Remote host to connect to.
  - ☞ PeerPort - The port number at PeerAddr to connect
  - ☞ LocalAddr - Bind the socket to the this address
  - ☞ LocalPort - Bind the socket to this port
  - ☞ Proto - The protocol to use
  - ☞ Type - The type of socket
  - ☞ Listen - Length of queue for a server socket
  - ☞ Reuse - Allow reuse of address
  - ☞ Timeout - Timeout value to use during connecting

# IO::Socket::INET (*cont.*)

---

---

- ☞ IO::Socket::INET also provides a simple way to create the most commonly used sock. That is, a TCP connection to another host and port

```
use IO::Socket;  
$s = IO::Socket::INET->new('localhost:80')  
    || die "IO::Socket: $@";
```

is the same as

```
$s = IO::Socket::INET->new(  
    PeerAddr => 'localhost',  
    PeerPort => 80,  
    Proto    => 'tcp'  
);
```

# IO::Socket TCP daytime client

---

---

```
#!/bin/perl -w
# Example of tcp daytime client using IO::Socket

use IO::Socket;

my $sock = IO::Socket::INET->new("localhost:daytime")
    or die "IO::Socket: $@";

# Print the date
print <$sock>;

# close the socket
close($sock) || die "close: $!";
```

# Finding information about a socket

---

---

- ➔ `getsockname` will return a packed socket address for the socket.

```
$paddr = getsockname(SOCK);  
($port, $ipaddr) = sockaddr_in($paddr);  
$quad = inet_ntoa($ipaddr);
```

- ➔ `getpeername` will return a packed socket address for the socket at the other end of the connection.

```
$paddr = getpeername(SOCK);  
($path) = sockaddr_un($paddr);
```

# Finding information about a socket

---

---

- ☞ `getsockopt` can be used to get various options.
  - ☞ `SO_TYPE` allows you to determine the type of socket. (ie `SOCK_STREAM`, `SOCK_DGRAM` etc.)

```
$type = getsockopt(SOCK, SOL_SOCKET, SO_TYPE);
```

- ☞ This can be useful for servers that inherit a socket from their parent process, so they do not know what they are getting.

# Finding information about a socket

---

---

- ☞ If you do not know what address the socket is using, how do you know which functions to call ?
- ☞ The first element in the socket address structure is the address family. We can use perl's `unpack` function to extract this.

```
$type = unpack("S", getsockname(SOCK) );

if ($type == AF_INET) {
    ($port, $ipaddr) = sockaddr_in($paddr);
    $quad = inet_ntoa($ipaddr);
}
elseif ($type == AF_UNIX) {
    $path = sockaddr_un($paddr);
}
else {
    die "Unknown address family";
}
```

# Types of server

---

---

- ➡ Forking server
- ➡ Concurrent server
- ➡ Threaded server
- ➡ The inetd server



# Forking server

---

---

- 👉 A new process is forked for each client connection.

```
for (; $addr = accept(CLIENT, SERVER); close(CLIENT)) {
    if ( !defined($pid = fork())) {
        warn "Cannot fork: $!";
        next;
    }
    elsif ($pid == 0) {
        process_client(\*CLIENT);
        exit;
    }
}
die "accept: $!";
```

- 👉 Whenever you fork processes you need to reap them when they finish.

```
$$SIG{CHLD} = sub { wait };
```

# Concurrent server

---

---

- ☞ All client connections are handled within one process.
- ☞ `select` is used to determine when a client is ready.

```
use Symbol qw(gensym);
```

```
vec($rin = "",fileno(SERVER),1) = 1;
while (select($rout=$rin,undef,undef)) {
    if(vec($rout,fileno(SERVER),1)) {
        $client = gensym();
        $addr = accept($client, SERVER) or next;
        $client[ fileno($client) ] = $client;
        vec($rin, fileno($client), 1) = 1;
    }
    else {
        for( $loop = 0 ; $loop < @client ; $loop++) {
            process_client($client[$loop])
                if (vec($rout, $loop, 1));
        }
    }
}
```

# Threaded server

---

---

- ☞ All client connections are handled within one process.
- ☞ Each client has its own thread within the server process.

```
use Thread::Pool;  
use Symbol qw(gensym);
```

```
$pool = Thread::Pool->new;
```

```
while (accept($client = gensym(), SERVER)) {  
    $pool->enqueue(&process_client, $client);  
}
```

```
die "accept: $!";
```

 **Threads within perl are still considered severely experimental**

# The inetd server

---

---

- ➡ A forking server that listens to many sockets.
- ➡ Each socket is described in a file `/etc/inetd.conf`.

```
ftp      stream  tcp      nowait  root    /usr/sbin/tcpd  in.ftpd  -l -a
```

- ➡ Allows almost any filter program to be run as a server.

```
echo     stream  tcp      nowait  nobody  /bin/cat  -u
```

# Common problems

---

---

- ➡ Output buffer
- ➡ Comparing packed addresses
- ➡ Closing handles
- ➡ Address in use error message

# Output buffer

---

---

## Problem

 I print to the socket handle, but the server never sees my data.

## Example



```
print SOCK "command\n";  
$response = <SOCK>; # client hangs here
```

# Output buffer (*cont.*)

---

---

## Explanation

-  print is a stdio operation which uses buffering.
-  The contents of the buffer are not sent until the buffer is flushed, which by default is not until the buffer is full.

# Output buffer (*cont.*)

---

---

## Solution

 Turn on auto-flush

```
$ofh = select(SOCK)
$| = 1;
select($ofh);
```

# this is often written as

```
select((select(SOCK), $|=1)[0]);
```

 Or use `syswrite`.

 The stdio functions in perl are

 `<>`, `eof`, `getc`, `print`, `printf`, `readline`



# Comparing packed addresses

---

---

## Problem

 I receive two packets from the same host and port, but the addresses returned by `recv` are not the same.

## Example

```
$addr1 = recv(SOCK, $buffer1, 1024);  
$addr2 = recv(SOCK, $buffer2, 1024);
```



```
print "From same host\n" if $addr1 eq $addr2;
```

# Comparing packed addresses (*cont.*)

---

---

## Explanation

-  The structure used to hold an address is a union of several structures and an internet address does not use all of this structure.
-  The extra space not used by the internet address is probably filled with random data, so the addresses will not compare as equal.

# Comparing packed addresses (*cont.*)

---

---

 Solution

 Zero fill the structures.

```
$addr1 = sockaddr_in(sockaddr_in($addr1));  
$addr2 = sockaddr_in(sockaddr_in($addr2));  
  
print "From same host\n" if $addr1 eq $addr2;
```

# Closing handles

---

---

## 👉 Problem

👉 My server dies with the error "Too many open files".

or

👉 My client does not see when the server closes the connection.

## 👉 Example



```
$client = $sock->accept or die "accept: $!";  
die "fork: $!" unless defined($pid = fork());  
unless($pid) {  
    process_client($client);  
    close($client);  
    exit;  
}
```

# Closing handles (*cont.*)

---

---

## Explanation


-  When the server does a `fork` the parent still has an open file descriptor to `$client`.
-  Calling `close` in the child process does not affect the handle in the parent process.

# Closing handles (*cont.*)

---

---

## Solution

 Close `$client` in the parent process after the call to `fork`.

## Example

```
die "fork: $!" unless defined($pid = fork);
```


```
if($pid) {  
    close($client)  
} else {  
    process_client($client);  
    close($client);  
    exit(0);  
}
```

# Address in use

---

---

## Problem

 My server occasionally crashes, but when I restart it I often get "bind: Address already in use"

## Example



```
$addr = inet_aton($host);  
$paddr = sockaddr_in($port, $addr);  
  
bind(SOCK, $paddr) or die "bind: $!";
```

# Address in use (*cont.*)

---

---

## Explanation

-  When a socket is closed, the system keeps the port allocated for a short time to acknowledge the close and catch any stray packets. This period is referred to as `TIME_WAIT`.
-  Until the system releases the port, it cannot be reused.

## Solution

-  This can be avoided by telling the system that you want to allow the socket to be reused.

```
use Socket qw(SOL_SOCKET SO_REUSEADDR);
```

```
setsockopt(SERVER, SOL_SOCKET, SO_REUSEADDR, 1);  
bind(SERVER, $paddr) or die "bind: $!";
```



# Case studies

---

---

- ➡ Send Email with SMTP
- ➡ Download Email from a POP3 server
- ➡ Retrieve files from an FTP server
- ➡ Transfer files between two remote FTP servers
- ➡ Reading only selected news articles using NNTP

# POP3

---

---

 Problem

 Your ISP keeps your mail on their server and only provides access via the POP3 protocol.

 Solution

 The Net::POP3 module will give you access to the server and all the POP3 commands.

# POP3

---

---

```
#!/bin/perl -w

use GetOpt::Long;
use Net::POP3;

$user = $ENV{USER} || $ENV{LOGNAME};
$out = "/var/spool/mail/" . $user;
$passwd = "";
$host = "mailhost";

GetOptions(
    'h:s' => \$host,
    'u:s' => \$user,
    'p:s' => \$passwd,
    'o:s' => \$out
);

open(OUT, ">>$out") or die "open: $!";

$pop3 = Net::POP3->new($host) or die "$@";
defined( $pop3->login($user,$passwd) ) or die $pop3->message;
$count = $pop3->stat;
```

# POP3

---

---

```
foreach $n (1..$count) {
  if ($mesg = $pop3->get($n)) {

    # Add the From line for the mbox file format
    print OUT "From pop3get ", scalar localtime, "\n";
    print OUT map { s/^From/>From/; $_ } @$mesg;
    print OUT "\n";

    $pop3->delete($n) or warn $pop3->message;
  }
  else {
    warn $pop3->message;
  }
}

$pop3->quit;


close(OUT);
```

# FTP

---


---

## Problem

 You have a process which creates log files on a remote machine that is only accessible via FTP.

or

 You have an FTP server on a machine where customers can place files.

 You need to periodically download those files and remove them from the server.

# FTP

---

---

 Solution

 Use Net::FTP to scan the directories and download the files.

 Use cron to invoke the script periodically.

or

 Modify the script to become a daemon process.

# FTP

---

---

```
#!/bin/perl -w

use Getopt::Long;
use Net::FTP;

GetOptions(
    'h:s' => \$host,
    'u:s' => \$user,
    'p:s' => \$passwd,
    'd:s' => \$dir,
    'f:s' => \$file,
    'r'   => \$remove
);

sub fileglob_to_re {
    local($_) = @_;

    s#([\./^\$()])#\$\1#g;
    s#\?#.#g;
    s#\*#.*#g;
    s#\{([\^}]+)\}#'( . join("|", split(/,/, $1)) . ' )'#ge;
    "^$_\$";
}
}
```

# FTP

---

---

```
$ftp = Net::FTP->new($host) or die "$@";
$ftp->login($user, $passwd) or die $ftp->message;
$ftp->cwd($dir) or die $ftp->message;

$pattern = fileglob_to_re($file);
$done     = $remove ? "Deleted.\n" : "Done.\n";

foreach $file (grep { /$pattern/o } $ftp->ls ) {
    print STDERR "Get: ", $file, " ...";

    $ftp->get($file) or do { print "Failed.\n"; next };

    if ($remove) {
        $ftp->delete($file) or print STDERR "Not ";
    }

    print STDERR $done;
}

$ftp->quit;
```





# FTP - 2

---

---

## Problem

-  You have some data on one FTP server which you want to transfer to another.
-  The files are large and you do not have space for them locally.

Or

-  It would take too long to transfer each file twice.

## Solution

-  Get the source FTP server to send the file directly to the destination server.

# FTP - 2

---

---

```
#!/bin/perl -w

use Getopt::Long;
use Net::FTP;

$s_user = $d_user = 'anonymous';

GetOptions(
    'src:s' => \$src,
    'dest:s' => \$dst,
    'du:s' => \$d_user,
    'dp:s' => \$d_passwd,
    'su:s' => \$s_user,
    'sp:s' => \$s_passwd,
);

# src and dest in format ftp.host.name:/path/to/file
($s_host, $s_dir, $s_file) = $src =~ m#^([\^:]+):((?:.*\/)?)([\^/]+)$#;
($d_host, $d_dir, $d_file) = $dst =~ m#^([\^:]+):((?:.*\/)?)([\^/]+)$#;

$d_file = $s_file unless length $d_file;

$s_ftp = Net::FTP->new($s_host) or die "$@";
$d_ftp = Net::FTP->new($d_host) or die "$@";
```

# FTP - 2

---

---

```
$s_ftp->login($s_user, $s_passwd) or die $s_ftp->message;
$d_ftp->login($d_user, $d_passwd) or die $d_ftp->message;

$s_ftp->cwd($s_dir) if length $s_dir;
$d_ftp->cwd($d_dir) if length $d_dir;

# Could be ->binary
$s_ftp->ascii or die $s_ftp->message;
$d_ftp->ascii or die $s_ftp->message;

$s_ftp->pasv_xfer($s_file, $d_ftp, $d_file)
    or warn $s_ftp->ok ? $d_ftp->message : $s_ftp->message;



$s_ftp->quit;
$d_ftp->quit;
```

# Security



---

---

## Problem

-  You have written a server, but you want to restrict whom the server responds to.
-  You need to restrict based on the user running the process on the client machine and the IP address of the client machine.

## Solution

-  Determine the remote user with `Net::Ident`.
-  Check the IP address network with `Net::Netmask`.

# Security

---

---

```
#!/bin/perl -w

use Net::Ident;
use Net::Netmask qw(fetchNetblock);
use IO::Socket;
use IO::Select;
use Proc::Daemon;

my %allow = (
    '127.0.0.0/24'    => { '*' => 1 },
    '214.123.1.0/24' => { 'tchrist' => 0, '*' => 1 },
    '192.168.1.0/24' => { 'gbarr' => 1 },
);

foreach $mask (keys %allow) {
    Net::Netmask->new($mask)->storeNetblock;
}

$session_id = Proc::Daemon::init;

$sock = IO::Socket::INET->new(
    LocalPort => 'daytime',
    Listen    => SOMAXCONN,
    Proto     => 'tcp',
    Reuse     => 1,
) or die "$@";
```

# Security

---

---

```
$sel = IO::Select->new($sock);

while($sel->can_read) {
    $client = $sock->accept;
    print $client scalar localtime, "\n"
        if check_user($client);
    close($client);
}

sub check_user {
    my $client = shift;

    $peer = $client->peerhost;
    $netblock = fetchNetblock($peer);

    return 0 unless ref $netblock;

    $allow = $allow{ $netblock->desc };
    $user = Net::Ident::lookup($client);

    return $allow->{$user} if exists $allow->{$user};
    return $allow->{'*'} if exists $allow->{'*'};
    return 0;
}
```

# Security

---

---

## WARNING



**There is no secure way to determine the user at the other end of any connection. Net::Ident provides a means, but to do so it queries a server on the client's machine. For this reason it CANNOT be trusted.**

# NNTP



---

---

## Problem

-  You do not have enough time to read news.
-  You are only interested in articles about a particular subject.

## Solution

-  Periodically run a script which finds the articles and downloads them to a mail folder.
-  This can be done in a number of ways. This example uses the `NEWNEWS` command to determine which articles have been posted in a given time period.



# NNTP

---

---

```
#!/bin/perl -w

use Net::NNTP;
use Getopt::Long;

$since = '1d';
$pattern = '*';
$outfile = "out";

Net::NNTP->debug(1);

GetOptions(
    'h:s' => \$host,
    'g:s' => \$groups,
    'p:s' => \$pattern,
    'o:s' => \$outfile,
    's:s' => \$since
);

%map = ( 'm' => 60, 'h' => 60*60, 'd' => 60*60*24, 'w' => 60*60*24*7);

die "Bad since: $since" unless $since =~ /^(\d+)([mhdw])$/;

$since = time - ($1 * $map{$2});
```

# NNTP

---

---

```
$nntp = Net::NNTP->new($host) or die "$@";

open(OUT,">>$outfile") or die "open: $!";

GROUP:
foreach $group ( split(/,/ , $groups) ) {

    $nntp->group($group)
    or do { warn $group,": ",$nntp->message; next GROUP };

    $articles = $nntp->newnews($since, $group)
    or do { warn $group,": ",$nntp->message; next GROUP };

    foreach $article (@$articles) {
        $match = $nntp->xpat('Subject', $pattern, $article);

        if ($match && %$match) {
            $art = $nntp->article($article);
            print OUT 'From nntp ',scalar localtime,"\n",@$art,"\n" if $art;
        }
    }
}


$nntp->quit;
```

# SMTP

---

---

## Problem

 You have a script which needs to send Email, but an external mailer program is not available.

## Solution

 Use `Net::SMTP` to send Email directly to your mail server.

# SMTP

---

---

```
#!/bin/perl -w

use Getopt::Long;
use Net::SMTP;

$host = 'mailhost';
$from = $ENV{USER} || $ENV{LOGNAME};
$subject = "No subject!";

GetOptions(
    'h:s' => \$host,
    'f:s' => \$from,
    's:s' => \$subject
);

die "No addresses\n" unless @ARGV;

$smtp = Net::SMTP->new($host) or die "$@";

$smtp->mail($from) or die $smtp->message;
$smtp->recipient(@ARGV) or die $smtp->message;
```

# SMTP

---

---

```
$to = join(",", map { "<$_>" } @ARGV);

$header = <<"EDQ";
To: $to
Subject: $subject

EDQ

$smtp->data($header, <STDIN>) or die $smtp->message;

# This could be done as :-
# $smtp->data;
# $smtp->datasend($header);
# $smtp->datasend($_) while <STDIN>;
# $smtp->dataend;

$smtp->quit;
```

# CPAN Modules used

---

- ➡ Net::FTP, Net::SMTP, Net::NNTP, Net::POP3
  - ➡ authors/id/GBARR/libnet-1.0606.tar.gz
- ➡ Proc::Daemon
  - ➡ authors/id/EHOOD/Proc-Daemon-0.01.tar.gz
- ➡ Net::Netmask
  - ➡ authors/id/MUIR/modules/Net-Netmask-1.4.tar.gz
- ➡ Net::Ident
  - ➡ authors/id/JPC/Net-Ident-1.10.tar.gz
- ➡ Thread::Pool
  - ➡ authors/id/MICB/ThreadPool-0.1.tar.gz

# Books

---

---

- ☞ Perl Cookbook  
**Author:** Tom Christiansen & Nathan Torkington  
**Publisher:** O'Reilly & Associates  
**ISBN:** 1-56592-243-3
- ☞ Unix Network Programming, Second Edition  
**Author:** W. Richard Stevens  
**Publisher:** Prentice Hall  
**ISBN:** 0-13-490012-X