

Sockets

Sockets are the most popular way to use TCP streams. Sockets make the network connection look like a file. The socket is a two-way connection between two computers. Connecting a socket has a couple steps, since there are a few details of the address of the other computer to specify. Once the socket is connected, it's easy to use. The `write()` function sends bytes to the other end, and `read()` gets bytes sent from the other end. This handout shows the Perl versions, but the socket functions look very similar in other languages -- there are some functions to create a socket, and then the language's natural read/write operations work on the socket. A Perl program needs the statement "use Socket;" up near its top to import these functions. In C, the headers are in "sys/socket.h".

Sockets and Blocking

If a program calls `read()` on a socket that has no data, the program will block (suspend) efficiently in the call to `read()`. When data is available, the call to `read()` will return with the data. While the program is blocked, other programs can make use of the hardware. Although somewhat more rare, a program can block in `write()` as well. If the program is writing data faster than the network or the computer on the other end can process it, the writing program may be blocked at times so the reading computer can catch up.

In reality, modern CPUs are so fast compared to the network connections we have, that most programs spend most of their time blocked, waiting for data to show up over the socket.

Error Handling

When writing Internet socket code, it's important to actually put in the error handling logic, since in a network situation the errors will actually happen with some regularity (host name not found, connection refused...). You want to put in the error handling so you can distinguish the network errors from your own programming errors. Most of the Perl networking functions return false on error. In addition, in the case of an error, the system functions will put a somewhat descriptive string in the global variable `$!`.

Buffering -- Autoflush

Many I/O systems do "buffering" automatically for efficiency. When you do a `write()`, the data is not written immediately. The system buffers up the characters until there is enough data to send all at once. For example, when writing to a file, it would be inefficient to write to the disk each time a line is printed. Instead, the file writing system might buffer the lines until there is a good amount of data (say 4k), and then write all the data in a single disk operation.

Buffering can be a real problem in a networking protocol where you send, say, one line of text and then wait for a response. If a socket is marked as "unbuffered", then each `write()` operation sends the data immediately.

Alternately, there may be a "flush" operation which tells the system "ok, that's all the data, please really send it now." In Perl, the FileHandle module provides the "autoflush" function which sets a file handle so it does not buffer...

```
use FileHandle;

....

## Suppose the SOCK file handle has been opened,
## set it to be unbuffered
autoflush SOCK, 1;
```

Below are the major socket functions in the order that you typically use them. Most of complexity is in setting up the socket. The actual reading and writing is pretty simple.

1. Hostname Lookup

\$ip = inet_aton(\$hostname)

Tries to look up the IP address of a hostname such as "www.yahoo.com". This will probably do a quick transaction with the local DNS server to get the IP address. Fails if the hostname cannot be looked up either because the hostname is wrong, or the local DNS system is down. Returns false on error and sets \$!.

2. Create Socket Address

\$sockaddr = sockaddr_in(\$port , \$ip);

Given a port number and an ipaddr, combine the two into a socket address that can be passed to connect(). We'll assume this step doesn't fail.

3. Allocate Socket

socket(SOCK, PF_INET, SOCK_STREAM, 0);

Given a file handle (e.g. SOCK, or could be a string like "HANDLE" stored in a variable \$sock), allocate a TCP socket and connect it to the file handle. The other arguments are constants. This just allocates the socket resources; it does not try to connect the socket to anything. The constants shown create a TCP stream connection. Other constants can be used to create other sorts of sockets. Fails if there are not enough resources to create another socket on the local machine. Most OS's have some kernel limit on how many simultaneous open sockets (or files) can be maintained -- probably a few thousand. This function is unlikely to fail unless the program opens many many sockets. Returns false on error and sets \$!.

4. Connect

connect(SOCK, \$sockaddr)

Given a socket file handle (from step 3) and a socket address (from step 2), try to connect to the given port on that machine. On success, the given socket can now be used for reading and writing to the remote machine. Fails if the connection cannot be made to the remote machine because it is unreachable or it is not listening on that port number. Both of those error conditions are quite common. Returns false on error and sets \$!.

5. Read

In Perl, the simplest way to read text data is just with the standard <SOCK> file reading operator. Returns undef on error or the EOF.

```
...
connect(SOCK, $sockaddr);
$line = <SOCK>;    ## read a line of text
```

Most protocols involve taking turns exchanging a few lines of text back and forth. The code for such a protocol will need to know when it is supposed to read and when it is supposed to write.

Reading all the data until the EOF is like reading from a file.

```
while ($line = <SOCK>) {
    ...
}
```

This while-loop form is only appropriate if you want to continually read data until the other end does a close().

The function sysread(SOCK, str, length) is an alternative that tries to read the given number of bytes, and puts them in the given string. Do not use both <> and sysread() -- use one or the other. Sysread() can be more efficient, since it does not have to read the data one line at a time. Sysread() is also appropriate for reading binary data, such as a JPEG, where the text end-of-line nature of <> does not make sense.

```
$result = "";
while(sysread(SOCK, $buff, 1000)) { ## attempt to read 1000 bytes at a time
    $result = $result . $buff;
}
```

Irregular Timing

The data will not all show up at once. The reading code will block when there is no data. Usually, the data will show up in irregularly timed and sized bursts. In reality, the CPU is much, much faster than the network, so the code will spend most of its time blocked, waking up for the occasional burst of network activity.

Irregular Sizing

Suppose that one end of the socket makes three separate write calls of 100 bytes each. This does not mean that the data will arrive in three 100 byte chunks. They may show up as a 150 byte chunk, a 37 byte chunk, and a 113 byte chunk. The reading code is responsible for accumulating the bytes to build up a meaningful chunk..

6. Write

In Perl, the simplest way to send data on the socket is using the standard print function...

```
print SOCK "Hello there, how are you?\012";
```

The function `syswrite(SOCK, str, length)` is an alternative. Do not use both `print()` and `syswrite()` -- use one or the other. Most Internet protocols end their text lines with a linefeed ("`\012`") or a carriage-return followed by a linefeed ("`\015\012`"). "`\012`" is 10 in octal which is the linefeed character. "`\015`" is 13 which is the carriage-return. Specifying the characters in this way avoids the problem where "`\n`" is mapped to the local end-of-line character, which changes with the platform you are running on.

7 Close

`close(SOCK)` -- closes the reading and writing capabilities of this end of the socket. The close on the writing end here will show up as an EOF on the other end.

finger.pl example

Finger is a simple old protocol where the client connects on port 79, sends a username followed by an end-of-line, and then prints out all the text the server sends back.

```
#!/usr/bin/perl -w
## A simple example client-side sockets
##
## The "finger" service listens on port 79
## You send it a user name on a line by itself,
## and it sends back some lines of text
## It used to be that finger worked on most hosts.
## Now however, it's rare, since it allows a bad-guy
## to figure out user names on the host.
## finger.pl nick whois.stanford.edu works
## finger.pl nick elaine.stanford.edu rejected
## finger.pl president whitehouse.gov -- hangs ?!

use strict 'vars';    ## enforce global/local declarations
use Socket;
use FileHandle;

## The end-of-line we'll use
## (return+linefeed)
my($EOL);
$EOL = "\015\012";

## Because of 'use strict' -- we declare all vars,
## even these globals
my($hostname, $user);

(scalar(@ARGV) == 2) || die "usage: <user> <host>";
$user = shift(@ARGV) || die "need username first";
$hostname = shift(@ARGV) || die "need hostname second";

my ($sport, $err, $ipaddr, $sockaddr);

$sport = 79;## port for "finger" service
## could do the service->port num lookup dynamically

## Look up the hostname to get its IP addr
$ipaddr = inet_aton($hostname) || die "cannot find '$hostname'";

## Make a socket addr out of the IP+port
$sockaddr = sockaddr_in($sport , $ipaddr);

## Create the socket
socket(SOCK, PF_INET, SOCK_STREAM, 0) || die "cannot create socket";

## Connect the socket
connect(SOCK, $sockaddr) || die "cannot connect";

## Set the socket to be unbuffered
autoflush SOCK, 1;

## Send the username
print SOCK "$user$EOL";
```

```

## Read back and print what they send back until EOF

my($line);

while ($line = <SOCK>) {
    chomp($line);
    print $line, "\n";
}

## Alternately, we could use the perl function
## sysread(SOCK, str, desired_len) for reading

close(SOCK);

```

Running Finger

```
elaine0:~/my193i> finger.pl nick whois.stanford.edu
```

Stanford University Whois Service

Try "whois help" for more information.

For directory service on the web: <http://www.stanford.edu/services/stanford.who>

```

-----
STANFORD-ONLY view - not public
The information below may not be available to the public. People in
the Stanford directory may choose whether information about them is
"public", "Stanford-only" or "private". Since this is the Stanford view,
please do not share the information below with the public. To see the
public view, go to http://www.stanford.edu/services/stanford.who
-----

```

```

Nick Parlante                <DS186W641>    <Phone Unavail>
  nick@cs.stanford.edu
  Computer Science, Academic staff-Lecturer

```

```

Nick Petalas                 <DS450S829>    (650) 555-5555
  nick_spam@pangea.Stanford.EDU
  Petroleum Engineering (2220), Academic staff-Engr Res Assoc

```

```

elaine0:~/my193i> finger.pl nick nosuchhost.stanford.edu
cannot find 'nosuchhost.stanford.edu' at finger.pl line 37.
elaine0:~/my193i> finger.pl nick elaine27.stanford.edu
cannot connect at finger.pl line 46.

```

HTTP Socket Example

Here is a simple HTTP client implementation. It separates out the socket creation code into a separate routine, and shows how to use `sysread()`...

```

#!/usr/bin/perl -w
## usage: webclient.pl www.example.com /foo/bar.html
## An simple HTTP client
## Sends a request like

```

```

## "GET $path HTTP/1.0\r\n\r\n"
## where $path is something like "/" or "/bar.html"
## Also, demonstrates separating out the socket
## code in a separate subroutine.

use strict 'vars';
use Socket;
use FileHandle;

## The end-of-line we'll use
## (return+linefeed)
my($EOL);
$EOL = "\015\012";

## Because of 'use strict' -- we declare all vars,
## even these globals
my($host, $request);

(scalar(@ARGV) == 2) || die "usage: www.example.com /index.html";
$host = shift(@ARGV);
$request = shift(@ARGV);

## Call our socket subroutine
my($err);
$err = ConnectSocket('SOCK', $host, 80);
if ($err ne "") { die($err); }

## Send the HTTP request
print SOCK "GET $request HTTP/1.0$EOL$EOL";

## Read back and print the response
## (two different forms of reading are shown)

my($buff, $result);
$result = "";

## 1. Reading with <SOCK>
if (0) {
    ## read 1 line at a time
    while ($buff = <SOCK>) {
        $result = $result . $buff;
    }
}
else {
    ## 2. Reading with sysread()
    ## read in large chunks of 1000 or fewer bytes, so quicker.
    while(sysread(SOCK, $buff, 1000)) {
        $result = $result . $buff;
    }
}

close(SOCK);

```

```

## Clean up the line endings
$result =~ s/\r\n?/\n/g;

print $result;

exit(0);

## Attempts to open a connection using the first argument
## filehandle, to the given hostname and port number
## On success, the file handle is connected, and
## the empty string is returned. Otherwise a non-empty
## error string is returned (in this way, the caller
## can use their own error handling/reporing strategy).
## Usage: ConnectSocket('SOCK', "www.example.com", 80)
sub ConnectSocket {
    my($sock, $hostname, $port) = @_;
    my ($ipaddr, $sockaddr);

    $ipaddr = inet_aton($hostname)           || return "hostname";
    $sockaddr = sockaddr_in($port, $ipaddr)  || return "address"; ## never errs
    socket($sock, PF_INET, SOCK_STREAM, 0)   || return "socket";

    connect($sock, $sockaddr)               || return "connect";
    autoflush $sock, 1;

    return "";
}

```

Sample Run

```

> webclient.pl www.stanford.edu /
HTTP/1.1 200 OK
Date: Fri, 12 Apr 2002 18:21:44 GMT
Server: Stronghold/3.0 Apache/1.3.19 RedHat/3014c WebAuth/2.5 (Unix) mod_ssl/2.8.1
OpenSSL/0.9.6 WebAuth/2.5 mod_fastcgi/2.2.10
Connection: close
Content-Type: text/html

<html>
<head>
<title>Stanford Home Page: Welcome to Stanford University</title>
<META NAME="description" CONTENT="The Stanford University home page is a good place
to start your
    search of Stanford University's web resources and
    websites.">
<META NAME="keywords" CONTENT="index, stanford,
    Stanford, stanford websites, stanford web, stanford
    home page, university, college">
<META name="provider" content="andyk@leland.stanford.edu">

</head>

<body bgcolor="#FFFFFF" link="#003366" vlink="#990000">
  <p>
<p>&nbsp;</p>

.... more html ...

```