

Sockets 2

Client Socket Example 2

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, $ipaddr)  || return "address"; ## this one never
happens
    socket($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 ...
```

Server Side Sockets

The client side can open a socket to "make a call" to a server -- how does the server wait for incoming calls? The code below will "listen" for incoming connections on a particular port number -- effectively waiting for an incoming connection on that port number.

On Most operating systems, port numbers below 1024 are "privileged" and only the root/administrator user has permission to listen on them. This provides a primitive sort of security. If a regular user were allowed to listen on, say, port number 110 (which is used for email POP connections), then that user could pretend to be the POP server and fake people into giving over their passwords. Therefore important services tend to run on ports below 1024, and only special users are allowed to listen on those ports. This security scheme has proved to have disadvantages as well -- the server software must run as a privileged user (i.e. root), and so if the server software is taken over, it's more damaging. (We'll look at these topics more in our security lectures.)

Setup

First, there are some basic setup operations that allocate the socket. The functions `socket()` and `sockaddr_in()` are as before. The `setsockopt()` function (optional) can be used to tell the server socket to re-use port numbers for separate connections -- important if the server is going to run for many thousand client connections over time. The `bind()` function associates the server socket with the port number, but does not yet begin listening. The functions all return false on error.

Here's what a nicely decomposed `CreateServerSocket()` function might look like...

```
## Given a socket name to use and a port #,
## create a server socket so it's ready for the listen call
## Returns an error string on error, or "" on success.
## usage: $err = CreateServerSocket('SOCK', 80);
sub CreateServerSocket {
    my($sock, $port) = @_;

    socket($sock, PF_INET, SOCK_STREAM, 0) || return("socket $!");

    ## note: (a) this setsockopt() is not required, and
    ## (b) that's an l not a 1 in the pack() call

    setsockopt($sock, SOL_SOCKET, SO_REUSEADDR, pack("l",1)) ||
        return("sockopt $!");

    bind($sock, sockaddr_in($port, INADDR_ANY)) || return("bind $!");

    autoflush $sock, 1;

    return("");
}
```

Listen

The `listen()` function gets the operating system to start listening for incoming connections on the bound port number. The first argument should be a server socket file handle that has been bound to the desired address. The call to `listen()` does not block. The OS will queue incoming socket connections up to some limit -- maybe 5 or so. The constant, `SOMAXCONN` uses the system default for the number to queue. Returns false on error.

```
listen(SERVER, SOMAXCONN) || die "listen: $!";
```

Accept

The `accept()` function blocks, waiting for an incoming connection. The first argument should be an un-initiated file-handle name which will be set to the incoming client connection. The second argument should be the server socket from the `listen()` call. Returns an address struct identifying the incoming client, or false on error.

```
my $client_addr = accept(CLIENT, SERVER); ## blocks
autoflush CLIENT, 1;
```

Miscellaneous

The function `unpack_sockaddr_in()` returns an array with the numeric IP address and port number from the client address (example below). The function `inet_ntoa()` formats the IP address as a string. The `gethostbyaddr()` function does a reverse-dns lookup to get the dns name ("elaine33.stanford.edu") from the IP address (potentially slow).

Server Example

This is a simple server example. This is a "single threaded" example -- it processes the client connections one at a time. A more sophisticated server would fork off a separate process (or thread) to deal with each incoming client connection, so they could be handled in parallel (We will stop short of that level of socket complexity in CS193i.)

```
## Setup our SERVER socket
CreateServerSocket(SERVER, $port);
listen(SERVER, SOMAXCONN);

## Wait for incoming client connections
my $client_addr;
while ($client_addr = accept(CLIENT, SERVER)) {           ## accept() blocks
    autoflush CLIENT, 1;

    my($port, $iaddr) = unpack_sockaddr_in($client_addr); ## extract ip and port

    my ($ip) = inet_ntoa($iaddr); ## ip as a string "171.64.64.250"

    my ($name) = gethostbyaddr($iaddr, AF_INET); ## lookup dns name (slow)

    print "Connection from $name [$ip] $port\n";

    ## print a line from the client (not dealing with EOLN issues)
    my($line);
    $line = <CLIENT>;
    print $line;
    print CLIENT "You said:$line";

    close(CLIENT);
}
```