13. Sockets

Contents:
Built-in Socket Functions
The IO::Socket Module

Why build networking functionality into your Perl scripts? You might want to access your email remotely, or write a simple script that updates files on a FTP site. You might want to check up on your employees with a program that searches for Usenet postings that came from your site. You might want to check a web site for any recent changes, or even write your own home-grown web server. The network is the computer these days, and Perl makes network applications easy.

Perl programmers have their choice of modules for doing common tasks with network protocols; Chapter 14, *Email Connectivity*, through Chapter 17, *The LWP Library*, cover the modules for writing email, news, FTP, and web applications in Perl. If you can do what you want with the available modules, you're encouraged to jump to those chapters and skip this one. However, there will be times that you'll have to wrestle with sockets directly, and that's where this chapter comes in.

Sockets are the underlying mechanism for networking on the Internet. With sockets, one application (a *server*) sits on a port waiting for connections. Another application (the *client*) connects to that port and says hello; then the client and server have a chat. Their actual conversation is done with whatever protocol they choose - for example, a web client and server would use HTTP, an email server would use POP3 and SMTP, etc. But at the most basic level, you might say that all network programming comes down to opening a socket, reading and writing data, and closing the socket again.

You can work with sockets in Perl at various levels. At the lowest level, Perl's built-in functions include socket routines similar to the system calls in C of the same name. To make these routines easier to use, the Socket module in the standard library imports common definitions and constants specific to your system's networking capabilities. Finally, the IO::Socket module provides an object interface to the socket functions through a standard set of methods and options for constructing both client and server communications programs.

Sockets provide a connection between systems or applications. They can be set up to handle streaming data or discrete data packets. Streaming data continually comes and goes over a connection. A transport protocol like TCP (Transmission Control Protocol) is used to process streaming data so that all of the data is properly received and ordered. Packet-oriented communication sends data across the network in discrete chunks. The message-oriented protocol UDP (User Datagram Protocol) works on this type of connection. Although streaming sockets using TCP are widely used for applications, UDP sockets also have their uses.

Sockets exist in one of two address domains: the Internet domain and the Unix domain. Sockets that are used for Internet connections require the careful binding and assignment of the proper type of address dictated by the Internet Protocol (IP). These sockets are referred to as Internet-domain sockets.

Sockets in the Unix domain create connections between applications either on the same machine or within a LAN. The addressing scheme is less complicated, often just providing the name of the target process.

In Perl, sockets are attached to a filehandle after they have been created. Communication over the connection is then handled by standard Perl I/O functions.

13.1 Built-in Socket Functions

Perl provides built-in support for sockets. The following functions are defined specifically for socket programming. For full descriptions and syntax, see Chapter 5, *Function Reference*.

`socket`

: Initializes a socket and assigns a filehandle to it.

`bind`

: For servers, associates a socket with a port and address. For clients, associates a socket with a specific source address.

`listen`

: (Server only.) Waits for incoming connection with a client.

`accept`

: (Server only.) Accepts incoming connection with a client.

`connect`

: (Client only.) Establishes a network connection on a socket.

`recv`

: Reads data from a socket filehandle.

`send`

: Writes data to a filehandle.

`shutdown` (or `close`)

: Terminates a network connection.

Regular functions that read and write filehandles can also be used for sockets, i.e., `write`, `print`, `printf`, and the diamond input operator, <>.

The socket functions tend to use hard-coded values for some parameters, which severely hurt portability. Perl solves this problem with a module called Socket, included in the standard library. Use this module for any socket applications that you build with the built-in functions (i.e., `use Socket`). The module loads the *socket.h* header file, which enables the built-in functions to use the constants and names specific to your system's network programming, as well as additional functions for dealing with address and protocol names.

The next few sections describe Perl socket programming using a combination of the built-in functions together with the Socket module. After that, we describe the use of the IO::Socket module.

13.1.1 Initializing a Socket

Both client and server use the `socket` call to create a socket and associate it with a filehandle. The `socket` function takes several arguments: the name of the filehandle, the network domain, an indication of whether the socket is stream-oriented or record-oriented, and the network protocol to be used. For example, HTTP (web) transactions require stream-oriented connections running TCP.

The following line creates a socket for this case and associates it with the filehandle SH:

```
use Socket;
socket(SH, PF_INET, SOCK_STREAM, getprotobyname('tcp')) || die $!;
```

The PF_INET argument indicates that the socket will connect to addresses in the Internet domain (i.e., IP addresses). Sockets with a Unix domain address use PF_UNIX.

Because this is a streaming connection using TCP, we specify SOCK_STREAM for the second argument. The alternative would be to specify SOCK_DGRAM for a packet-based UDP connection.

The third argument indicates the protocol used for the connection. Each protocol has a number assigned to it by the system; that number is passed to socket as the third argument. In the scalar context, getprotobyname returns the protocol number.

Finally, if the socket call fails, the program will die, printing the error message found in $!.

13.1.2 Client Connections

On the client side, the next step is to make a connection with a server at a particular port and host. To do this, the client uses the connect call. connect requires the socket filehandle as its first argument. The second argument is a data structure containing the port and hostname that together specify the address. The Socket package provides the sockaddr_in function to create this structure for Internet addresses and the sockaddr_un function for Unix domain addresses.

The sockaddr_in function takes a port number for its first argument and a 32-bit IP address for the second argument. The 32-bit address is formed from the inet_aton function found in the Socket package. This function takes either a hostname (e.g., www.oreilly.com) or a dotted-decimal string (e.g., 207.54.2.25), and it returns the corresponding 32-bit structure.

Continuing with the previous example, a call to connect could look like this:

```
my $dest = sockaddr_in (80, inet_aton('www.oreilly.com'));
connect (SH, $dest) || die $!;
```

This call attempts to establish a network connection to the specified server and port. If successful, it returns true. Otherwise, it returns false and dies with the error in $!.

Assuming that the connect call has completed successfully and a connection has been established, there are a number of functions we can use to write to and read from the file handle. For example, the send function sends data to a socket:

```
$data = "Hello";
send (FH, $data);
```

The print function allows a wider variety of expressions for sending data to a filehandle.

```
select (FH);
print "$data";
```

To read incoming data from a socket, use either the recv function or the "diamond" input operator regularly used on filehandles. For example:

```
recv (FH, $buffer);
$input = <FH>;
```

After the conversation with the server is finished, use close or shutdown to close the connection and destroy the socket.

13.1.3 Server Connections

After creating a socket with the `socket` function as above, a server application must go through the following steps to receive network connections:

1. Bind a port number and machine address to the socket.
2. Listen for incoming connections from clients on the port.
3. Accept a client request and assign the connection to a specific filehandle.

We start out by creating a socket for the server:

```
my $proto = getprotobyname('tcp');
socket(FH, PF_INET, SOCK_STREAM, $proto) || die $!;
```

The filehandle `$FH` is the generic filehandle for the socket. This filehandle only receives requests from clients; each specific connection is passed to a different filehandle by `accept`, where the rest of the communication occurs.

A server-side socket must be bound to a port on the local machine by passing a port and an address data structure to the `bind` function via `sockaddr_in`. The Socket module provides identifiers for common local addresses, such as localhost and the broadcast address. Here we use `INADDR_ANY`, which allows the system to pick the appropriate address for the machine:

```
my $sin = sockaddr_in (80, INADDR_ANY);
bind (FH, $sin) || die $!;
```

The `listen` function tells the operating system that the server is ready to accept incoming network connections on the port. The first argument is the socket filehandle. The second argument gives a queue length, in case multiple clients are connecting to the port at the same time. This number indicates how many clients can wait for an `accept` at one time.

```
listen (FH, $length);
```

The `accept` function completes a connection after a client requests and assigns a new filehandle specific to that connection. The new filehandle is given as the first argument to `accept`, and the generic socket filehandle is given as the second:

```
accept (NEW, FH) || die $!;
```

Now the server can read and write to the filehandle `NEW` for its communication with the client.

13.1.4 Socket Module Functions

The following functions are imported from the Socket module for use in socket applications:

- inet_aton
- inet_ntoa
- sockaddr_in
- sockaddr_un
- unpack_sockaddr_in
- unpack_sockaddr_un

The following constants are defined in the Socket module:

`INADDR_ANY`

> The four-byte packed string for the wildcard IP address that specifies any of the host's addresses (if the host has multiple addresses). This is equivalent to

`inet_aton('0.0.0.0')`.

`INADDR_BROADCAST`

The four-byte packed string for the broadcast address. This is equivalent to `inet_aton('255.255.255.255')`.

`INADDR_LOOPBACK`

The four-byte packed string for the loopback address. This is equivalent to `inet_aton('localhost')`.

`INADDR_NONE`

The four-byte packed string for the "invalid" IP address (bitmask). Equivalent to `inet_aton('255.255.255.255')`.