An Introduction to Socket Programming

(by) Reg Quinton <reggers@julian.uwo.ca>

© *$Id: socket.html,v 1.8 1997/05/02 20:17:16 reggers Exp $*

Contents:

Introduction:

These course notes are directed at Unix application *programmers* who want to develop client/server applications in the **TCP/IP** domain (with some hints for those who want to write **UDP/IP** applications). Since the Berkeley socket interface has become something of a standard these notes will apply to programmers on other platforms.

Fundamental concepts are covered including network addressing, well known services, sockets and ports. Sample applications are examined with a view to developing similar applications that serve other contexts. Our goals are

- to develop a function, **`tcpopen(server,service)`**, to connect to service.
- to develop a server that we can connect to.

This course requires an *understanding* of the C programming language and an *appreciation* of the programming environment (ie. compilers, loaders, libraries, Makefiles and the RCS revision control system). If you want to know about socket programming with perl(1) then see below but you should read everything first.

Our example is the UWO/ITS whois(1) service -- client and server sources available in:

> Network Services: http://www.uwo.ca/its/ftp/pub/unix/network

Look for the whois(1) client and the whoisd(8) server. You'll find extensive documentation on the UWO/ITS Whois/CSO server -- that's the **whoisd(8)** server. It also includes some Perl clients which access the server to provide a gateway service (for the Finding People Web page and for CSO/PH clients). The Unix whois(1) client will be pretty obvious after you've read these notes.

BEWARE:

> If C code scares you, then you'll get some concepts but you might be in the wrong course. You need to be a programmer to write programs (of course). This *isn't* an Introduction to C (or Perl)!

Existing Services:

Before starting, let's look at existing services. On a Unix machine there are usually lots of TCP/IP and UDP/IP services installed and running:

```
[1:17pm julian] netstat -a
Active Internet connections (including servers)
Proto R-Q S-Q  Local Address Foreign Address    (state)
tcp     0   0  julian.2717   vnet.ibm.com.smtp  ESTABLISHED
tcp     0   0  julian.smtp   uacsc2.alban.55049 TIME_WAIT
tcp     0  13  julian.nntp   watserv1.wat.3507  ESTABLISHED
tcp     0   0  julian.nntp   gleep.csd.uw.3413  ESTABLISHED
tcp     0   0  julian.telnet uwonet-serve.55316 ESTABLISHED
tcp     0   0  julian.login  no8sun.csd.u.1023  ESTABLISHED
tcp     0   0  julian.2634   Xstn15.gaul..6000  ESTABLISHED
          etc...
tcp     0   0  *.printer     *.*                LISTEN
tcp     0   0  *.smtp        *.*                LISTEN
tcp     0   0  *.waisj       *.*                LISTEN
tcp     0   0  *.account     *.*                LISTEN
tcp     0   0  *.whois       *.*                LISTEN
tcp     0   0  *.nntp        *.*                LISTEN
          etc...
```

```
     udp      0   0  *.ntp          *.*
     udp      0   0  *.syslog       *.*
     udp      0   0  *.xdmcp        *.*
```

Netstat Observations:

Inter Process Communication (or IPC) is between **host.port** pairs (or **host.service** if you like). A process *pair* uses the connection -- there are client and server applications on each end of the IPC connection.

Note the two protocols on IP -- **TCP** (Transmission Control Protocol) and **UDP** (User Datagram Protototocol). There's a third protocl **ICMP** (Internet Control Message Protocol) which we'll not look at -- it's what makes IP work in the first place!

We'll be looking in more detail at TCP services and will not look at UDP -- but see a sample Access Control List client/server pair which uses UDP services, you'll find that in:

Access Control Lists: http://www.uwo.ca/its/ftp/pub/unix/security/acl

TCP services are connection orientated (like a stream, a pipe or a tty like connection) while UDP services are connectionless (more like telegrams or letters).

We recognize many of the services -- SMTP (Simple Mail Transfer Protocol as used for E-mail), NNTP (Network News Transfer Protocol service as used by Usenet News), NTP (Network Time Protocol as used by xntpd(8)), and SYSLOG is the BSD service implemented by syslogd(1M).

The netstat(1M) display shows many TCP services as ESTABLISHED (there is a connection between **client.port** and **server.port**) and others in a LISTEN state (a server application is listening at a port for client connections). You'll often see connections in a CLOSE_WAITE state -- they're waiting for the socket to be torn down.

Host names and IP numbers:

Hosts have *names* (eg. julian.uwo.ca) but IP addressing is by *number* (eg. [129.100.2.12]). In the old days name/number translations were tabled in /etc/hosts.

```
     [2:38pm julian] page /etc/hosts
     # /etc/hosts: constructed out of private data and DNS. Some machines
     # need to know some things at boot time. Otherwise, rely on DNS.
     #
     127.0.0.1       localhost
     129.100.2.12    julian.uwo.ca
     129.100.2.26    backus.ccs.uwo.ca loghost.its.uwo.ca
     129.100.2.33    filehost.ccs.uwo.ca
     129.100.2.14    panther.uwo.ca
               etc...
```

These days name to number translations are implemented by the Domain Name Service (or DNS) -- see named(8). and resolv.conf(4).

```
[2:43pm julian] page /etc/resolv.conf
# $Author: reggers $
# $Date: 1997/05/02 20:17:16 $
# $Id: socket.html,v 1.8 1997/05/02 20:17:16 reggers Exp $
# $Source: /usr/src/usr.local/doc/courses/socket/RCS/socket.html,v $
# $Locker:  $
#
# The default /etc/resolv.conf for the ITS solaris systems.
#
nameserver 129.100.2.12
nameserver 129.100.2.51
nameserver 129.100.10.252
domain its.uwo.ca
search ncsm.its.uwo.ca its.uwo.ca uwo.ca
```

Programming Calls:

Programmers don't scan /etc/hosts nor do they communicate with the DNS. The C library routines gethostbyname(3) (and gethostbyaddr(3) on the same page) each return a pointer to an object with the following structure:

```
struct     hostent {
   char   *h_name;        /* official name */
   char   **h_aliases;    /* alias list */
   int    h_addrtype;     /* address type */
   int    h_length;       /* address length */
   char   **h_addr_list;  /* address list */
};
#define h_addr h_addr_list[0]
  /* backward compatibility */
```

The structure **`h_addr_list`** is a list of IP numbers (recall that a machine might have several interfaces, each will have a number).

Good programmers would try to connect to each address listed in turn (eg. some versions of ftp(1) do that). *Lazy programmers* (like me) just use **`h_addr`** -- the first address listed. But see the acl(1) and acld(8) example noted earlier -- the client will try each server until it gets an answer or runs out of servers to ask.

Client applications connect to a **host.port** (cf. netstat output) for a service provided by the application found at that address.

```
Proto R-Q S-Q  Local Address  Foreign Address    (state)
tcp     0   0  julian.2717    vnet.ibm.com.smtp  ESTABLISHED
tcp     0  13  julian.nntp    watserv1.wat.3507  ESTABLISHED
```

The connection is usually prefaced by translating a *host name* into an *IP number* (but if you knew the IP number you could carefully skip that step).

```
int     tcpopen(host,service)
char    *service, *host;
{
    struct  hostent         *hp;
          etc...
```

```
        if ((hp=gethostbyname(host)) == NULL) then error...
```

I say "carefully" because the IP address is a structure of 4 octets. Watch out for byte ordering. An unsigned long isn’t the same octet sequence on all machines. See byteorder(3N) for host to net conversions (host format to/from network format).

Services and Ports:

Services have *names* (eg. SMTP the Simple Mail Transfer Protocol). Ports have *numbers* (eg. SMTP is a service on port 25). The mapping from service names to port numbers is listed in /etc/services.

```
[1:22pm julian] page /etc/services
# $Author: reggers $
# $Date: 1997/05/02 20:17:16 $
#
# Network services, Internet style
          etc...
ftp             21/tcp
telnet          23/tcp
smtp            25/tcp          mail
whois           43/tcp          nicname
domain          53/tcp          nameserver
domain          53/udp          nameserver
tftp            69/udp
finger          79/tcp
nntp            119/tcp         readnews untp
ntp             123/udp
snmp            161/udp
xdmcp           177/udp         xdm
          etc...
```

Programming Calls:

But programmers don’t scan /etc/services, they use library routines. The C library routines getservbyname(3N) (and getservbyport(3N) on the same page) each return a pointer to an object with the following structure containing the broken-out fields of a line in /etc/services.

```
struct   servent {
   char   *s_name;      /* name of service */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port for service */
   char   *s_proto;     /* protocol to use */
};
```

Client applications connect to a service port. Usually this is prefaced by translating a *service name* (eg. SMTP) into the *port number* (but if you knew the port number you could carefully skip that step).

```
int     tcpopen(host,service)
char    *service, *host;
{
    struct  servent         *sp;
          etc...
```

```
if ((sp=getservbyname(service,"tcp")) == NULL) then error...
```

Ie. to determine the port number for a particular **tcp service**. Note that you'd do the same to determine port numbers for UDP services.

Socket Addressing:

A Socket Address is a **host.port** pair (communication is between **host.port** pairs -- one on the server, the other on the client). We know how to determine host numbers and service numbers so we're well on our way to filling out a structure were we specify those numbers. The structure is **`sockaddr_in`**, which has the address family is **`AF_INET`** as in this fragment:

```
int     tcpopen(host,service)
char    *service, *host;
{   int     unit;
    struct  sockaddr_in     sin;
    struct  servent         *sp;
    struct  hostent         *hp;
          etc...
    if ((sp=getservbyname(service,"tcp")) == NULL) then error...
    if ((hp=gethostbyname(host)) == NULL) then error...

    bzero((char *)&sin, sizeof(sin));
    sin.sin_family=AF_INET;
    bcopy(hp->h_addr,(char *)&sin.sin_addr, hp->h_length);
    sin.sin_port=sp->s_port;
          etc...
```

The code fragment is filling in the IP address type **`AF_INET`**, port number and IP address in the Socket Address structure -- the address of the remote **host.port** where we want to connect to find a service.

There's a generic Socket Address structure, a **`sockaddr`**, used for communication in *arbitrary* domains. It has an address family field and an address (or data) field:

```
/* from: /usr/include/sys/socket.h */
struct sockaddr {
    u_short sa_family;   /* address family */
    char    sa_data[14]; /* max 14 byte addr */
};
```

The **`sockaddr_in`** structure is for Internet Socket Addresses (address family **`AF_INET`**). An instance of the *generic* socket address.

```
/* from: /usr/include/netinet/in.h */
struct sockaddr_in {
    short   sin_family;       /* AF_INET      */
    u_short sin_port;         /* service port */
    struct  in_addr sin_addr; /* host number  */
    char    sin_zero[8];      /* not used     */
};
```

The family defines the interpretation of the data. In other domains addressing will be different -- services in the UNIX domain are names (eg. /dev/printer). In the **`sockaddr_in`** structure we've got fields to specify a port and a host IP number (and 8 octets that aren't used at all!). That structure specifies one end of an IPC connection. Creating that structure and filling in the right numbers has been pretty easy so far.

File Descriptors and Sockets:

File Descriptors:

File Descriptors are the fundamental I/O object. You read(2) and write(2) to file descriptors.

```
        int cc, fd, nbytes;
        char *buf;

        cc = read(fd, buf, nbytes);
        cc = write(fd, buf, nbytes)
```

The **`read`** attempts to read **`nbytes`** of data from the object referenced by the file descriptor **`fd`** into the buffer pointed to by **`buf`**. The write does a write to the file descriptor from the buffer. Unix I/O is a byte stream.

File descriptors are numbers used for I/O. Usually the result of open(2) and creat(2) calls.

All Unix applications run with **`stdin`** as file descriptor 0, **`stdout`** as file descriptor 1, and **`stderr`** as file descriptior 3. But **`stdin`** is a **`FILE`** (see stdio(3S)) not a file descriptor. If you want a stdio **`FILE`** on a file descriptor use fdopen(3S).

Sockets:

A Socket is a Unix file descriptor created by the socket(3N) call -- you don't open(2) or creat(2) a socket. By way of comparison pipe(2) creates file descriptors too -- you might be familiar with pipes which predate sockets in the development of the Unix system.

```
       int s, domain, type, protocol;
       s = socket(domain, type, protocol);
              etc...
       cc = read(s, buf, nbytes);
```

The **`domain`** parameter specifies a communications domain (or address family). For IP use **`AF_INET`** but note that **`socket.h`** lists all sorts of address families. This is to inform the system *how* an address should be *understood* -- on different networks, like **`AF_DECnet`**, addressing may be longer than the four octets of an IP number. We're only concerned with IP and the **`AF_INET`** address family.

The **`type`** parameter specifies the semantics of communication (sometimes know as a specification of *quality of services*). For TCP/IP use **`SOCK_STREAM`** (for UDP/IP use **`SOCK_DGRAM`**). Note that *any* address family might support those service types. See **`socket.h`** for a list of service types that might be supported.

A **SOCK_STREAM** is a sequenced, reliable, two-way connection based byte stream. If a data cannot be successfully transmitted within a reasonable length of time the connection is considered broken and I/O calls will indicate an error.

The **protocol** specifies a particular protocol to be used with the socket -- for TCP/IP use 0. Actually there's another programmers interface getprotobyname(3N) that provides translates protocol names to numbers. It's an interface to the data found in /etc/protocols -- compare with the translation of service *names* to port *numbers* discussed above.

Client Connect:

A client application creates a socket(3N)and then issues a connect(3N) to a service specified in a **sockaddr_in** structure:

```
int     tcpopen(host,service)
char    *service, *host;
{   int     unit;
    struct  sockaddr_in     sin;
    struct  servent         *sp;
    struct  hostent         *hp;

    if ((sp=getservbyname(service,"tcp")) == NULL) then error...
    if ((hp=gethostbyname(host)) == NULL) then Ierror...
    bzero((char *)&sin, sizeof(sin))
          etc...
    if ((unit=socket(AF_INET,SOCK_STREAM,0)) < 0) then error...
    if (connect(unit,&sin,sizeof(sin)) < 0) then error...
    return(unit);
}
```

The result returned is a file descriptor which is connected to a server process. A communications channel on which one can conduct an application specific protocol.

Client Communication:

Having connected a socket to a server to establish a file descriptor communication is with the usual Unix I/O calls. You have Inter Process Communication (or IPC) to a server.

Many programmers turn file descriptors into stdio(3S) streams so they can use fputs, fgets, fprintf, etc. -- use fdopen(3S).

```
main(argc,argv)
int     argc;
char    *argv[];
{
    int     unit,i;
    char    buf[BUFSIZ];
    FILE    *sockin,*sockout;

    if ((unit=tcpopen(WHOHOST,WHOPORT))
         < 0) then error...
```

```
        sockin=fdopen(unit,"r");
        sockout=fdopen(unit,"w");
              etc...
        fprintf(sockout,"%s\n",argv[i]);
              etc...
        while (fgets(buf,BUFSIZ,sockin)) etc...
```

Stdio Buffers:

Stdio streams have powerful manipulation tools (eg. fscanf is amazing). But beware, streams are buffered! This means a well placed fflush(3S) is often required to flush a buffer to the peer.

```
        fprintf(sockout,"%s\n",argv[i]);
        fflush(sockout);

        while (fgets(buf,BUFSIZ,sockin)) etc...
```

Many client/server protocols are client driven -- the client sends a command and expects an answer. The server won't see the command if the client doesn't flush the output. Likewise, the client won't see the answer if the server doesn't flush it's output.

Watch out for client and server blocking -- both waiting for input from the other.

Server Applications:

A system offers a service by having an application running that is *listening* at the service port and willing to *accept* a connection from a client. If there is no application listening at the service port then the machine doesn't offer that service.

The SMTP service is provided by an application listening on port 25. On Unix systems this is usually the sendmail(1M) application which is started at boot time.

```
    [2:20pm julian] ps -agx | grep sendmail
      419   ? SW    0:03 /usr/lib/sendmail -bd -q15m
    18438   ? IW    0:01 /usr/lib/sendmail -bd -q15m

    [2:28pm julian] netstat -a | grep smtp
    tcp  0 0 julian.3155 acad3.alask.smtp SYN_SENT
    tcp  0 0 *.smtp      *.*              LISTEN
```

In the example we have a process *listening* to the smtp port (for inbound mail) and another process *talking* to the smtp port on **acad3.alaska.edu** (ie. sending mail to that system).

So how do we get a process bound behind a port?

Server Bind:

A Server uses bind(3N) to establish the local **host.port** assignment -- ie. so it is the process behind that port. That's really only required for servers -- applications which accept(3N) connections to provide a service.

```
        struct servent    *sp;
        struct sockaddr_in sin;

        if ((sp=getservbyname(service,"tcp")) == NULL) then error...

        sin.sin_family=AF_INET;
        sin.sin_port=sp->s_port;
        sin.sin_addr.s_addr=htonl(INADDR_ANY);

        if ((s=socket(AF_INET,SOCK_STREAM,0)) < 0) then error...
        if (bind(s, &sin, sizeof(sin)) < 0) then error...
```

htonl(3N) converts a long to the right sequence (given different byte ordering on different machines). The IP address **`INADDR_ANY`** means *all interfaces*. You could, if you wanted, provide a service only on *some* interfaces -- eg. if you only provided the service on the loopback interface (127.0.0.1) then the service would only be available to clients on the same system.

What this code fragment does is specify a *local* interface and port (into the **`sin`** structure). The process is bound to that port -- it's now the process behind the local port.

Client applications usually aren't concerned about the local **host.port** assignment (the connect(3N) does a bind o some random but unused local port on the right interface). But rcp(1) and related programs (like **rlogin(1)** and **rsh(1)**) do connect from reserved port numbers.

I've done the same in some of my programming. See, for example, the version of tcpopen.c used in our Passwdd/Passwd -- An authentication Daemon/Client. There's an instance where a client application connects from a reserved port.

Listen and Accept:

To accept connections, a socket is created with socket(3N), it's bound to a service port with bind(3N), a queue for incoming connections is specified with listen(3N) and then the connections are accepted with accept(3N) as in this fragment:

```
        struct servent    *sp;
        struct sockaddr_in sin,from;

        if ((sp=getservbyname(service,"tcp")) == NULL) then error...
        sin.sin_family=etc...
        if ((s=socket(AF_INET,SOCK_STREAM,0)) < 0) then error...
        if (bind(s, &sin, sizeof(sin)) < 0) then error...
        if (listen(s,QUELEN) < 0) then error...
        for (;;) {
           if ((g=accept(f,&from,&len)) < 0) then error...
           if (!fork()) {
              child handles request...
                 ...and exits
               exit(0);
           }
           close(g);   /* parent releases file */
        }
```

This is the programming schema used by utilities like sendmail(1M) and others -- they create their socket and listen for connections. When connections are made, the process forks off a child to handle that service request and the parent process continues to listen for and accept further service requests.

But, you really don't want to use that programming paradigm unless you really have to. There are lots of hidden issues (like becoming a detached process and more) that you'd rather avoid.

Fortunately, there's an easier method.

Inetd Services:

Not all services are started at boot time by running a server application. Eg. you won't usually see a process running for the finger service like you do for the smtp service. Many are handled by the InterNet Daemon inetd(1M). This is a generic service configured by the file inetd.conf(4).

```
[2:35pm julian] page /etc/inetd.conf
# $Author: reggers $
# $Date: 1997/05/02 20:17:16 $
#
# Internet server configuration database
ftp     stream tcp nowait root   /usr/etc/ftpd      ftpd
telnet  stream tcp nowait root   /usr/etc/telnetd   telnetd
shell   stream tcp nowait root   /usr/etc/rshd      rshd
login   stream tcp nowait root   /usr/etc/rlogind   rlogind
exec    stream tcp nowait root   /usr/etc/rexecd    rexecd
uucpd   stream tcp nowait root   /usr/etc/uucpd     uucpd
finger  stream tcp nowait nobody /usr/etc/fingerd   fingerd
          etc...
whois   stream tcp nowait nobody /usr/lib/whois/whoisd whoisd
          etc...
```

Inetd Comments:

For each service listed in /etc/inetd.conf the inetd(1M) process, and that is a process is started at boot time, executes the socket(3N), bind(3N), listen(3N) and accept(3N) calls as discussed above. Inetd also handles many of the daemon issues (signal handling, set process group and controlling tty) which we've studiously avoided.

The inetd(1M) process spawns the appropriate server application (with fork(2) and exec(2)) when a client connects to the service port. The daemon continues to listen for further connections to the service while the spawned child process handles the request which just came in.

The server application (ie. the child spawned by inetd(1M)) is started with **`stdin`** and **`stdout`** connected to the remote **host.port** of the client process which made the connection. Any input/output by the server appliation on **`stdin/stdout`** are sent/received by the client application. You have Inter Process Communication (or IPC)!

This means, any application written to use **`stdin/stdout`** can be a server application. Writing a server application should therefore be fairly simple.

Whois Daemon:

On julian we have an entry in /etc/inetd.conf for the UWO/ITS whois service:

```
[3:25pm julian] grep whois /etc/inetd.conf
whois   stream tcp nowait nobody /usr/lib/whois/whoisd   whoisd
```

This is our local directory service -- it's implemented on a TCP/IP stream (all whois services are), at the **`whois`** port (all whois services should be at that port), it's ran as user **`nobody`** (you don't need to run servers as user **`root`**), the program to run is **`/usr/lib/whois/whoisd`**, and the command line to the program is just **`whoisd`**.

This is a standard **whois** service -- it implements the trivial protocol required of *all* **whois** servers. Any **whois** client can use the service. The program conducts an application protocol on **`stdin/stdout`** (which is usually connected by a TCP/IP socket to a client application). The protocol is trivial -- server accepts a one line query, answers back and exits.

Running the Daemon:

You can run the whois daemon (on the server) to see what it does:

```
[3:27pm julian] echo reggers | /usr/lib/whois/whoisd
There were 1 matches on your request.

           Full Name: Quinton, Reg
          Department: Info Tech Svcs
                Room: NSC 214
               Phone: 679-2111x(6026)
           Index Key: 481800
     Machine Address: reggers@julian.uwo.ca
 Directory Addresses: reg.quinton@uwo.ca
                    : r.quinton@uwo.ca
                    : reggers@uwo.ca
                    : quinton@uwo.ca

For more information try 'whois help'.
```

The program is command driven -- you give a command (or query string) on **`stdin`**, it produces results on **`stdout`**, and exits. This is a very simple protocol, compare with fingerd(1M).

Actually the example is a misrepresentation -- our server will only answer questions if it's input is a **`socket`** in the **`AF_INET`**. That's because we want to syslog(3) all transactions -- we want to know where the connection came from.

The Code:

The server program is easy enough -- read a line, switch on command, and exit.

```
fgets(string,BUFSIZ,stdin); read from socket...
```

```
/* for some reason people send the whois phrase */

again:
     strcpy(verb,""); strcpy(args,"");
     sscanf(buf,"%[^ \t\r\n]%*c%[^\r\n]",verb,args);
     if (!strcasecmp(verb,"whois")) {
       strcpy(buf,args);
       goto again;
     }
     sscanf(buf,"%[^\r\n]",buf);

/* switch on command verbs */
if (!strcasecmp(verb,"help"))
       givehelp(args);        output sent to stdout...
else   etc...

/* or just display a person */
else    listdisplay(lookbyname(buf));
                              output sent to stdout...

fflush(stdout);       push output to client ...
```

Server programs can be that simple.

Connecting to the Server:

You can make a telnet(1) connection to the **whois** service on the server.

```
[3:47pm julian] telnet julian whois
Trying 129.100.2.12 ... Connected to julian.uwo.ca.
Escape character is ’^]’.
reggers .... my command input
There were 1 matches on your request.

           Full Name: Quinton, Reg
          Department: Info Tech Svcs
                Room: NSC 214
               Phone: 679-2111x(6026)
           Index Key: 481800
     Machine Address: reggers@julian.uwo.ca
 Directory Addresses: reg.quinton@uwo.ca
                    : r.quinton@uwo.ca
                    : reggers@uwo.ca
                    : quinton@uwo.ca

For more information try ’whois help’.
Connection closed by foreign host.
```

But we wouldn’t normally use telnet as the client application (although in this case we could).

Whois Client:

The whois(1) client makes a TCP/IP connection to the server (using the tcpopen function we’ve developed here) and conducts the kind of protocol that you would type if you where to make a connection by hand:

```
[7:30am julian] whois reggers
```

```
There were 1 matches on your request.

           Full Name: Quinton, Reg
          Department: Info Tech Svcs
                Room: NSC 214
               Phone: 679-2111x(6026)
           Index Key: 481800
     Machine Address: reggers@julian.uwo.ca
 Directory Addresses: reg.quinton@uwo.ca
                    : r.quinton@uwo.ca
                    : reggers@uwo.ca
                    : quinton@uwo.ca

For more information try 'whois help'.
```

The client sends the command **"reggers"**, the server sends back the answer and the client displays the answer received to the user. When the server is finished the connection is closed.

If you understand the development of the tcpopen function then the rest of the code for that client should not be too difficult. See the entire distribution for that application -- there's only one main program to complete the kit.

Perl Socket Programming:

These days it's not unusal to see socket programming in perl(1) as well as C programs. Assuming you have been able to follow the notions presented above in the development of a tcpopen function written in C as used by our whois(1) client the following is for the Perl enthusiast:

```
sub tcpopen {
   use Socket;                       # need socket interface
   my($server, $service) = @_;       # args to this function
   my($proto, $port, $iaddr);        # local variables
   my($handle)="$server\:\:$service"; # localized obscure handle

   die("550:Cannot getprotobyname('tcp')\r\n")
      unless ($proto = getprotobyname('tcp'));

   die("550:Cannot getservbyname($service)\r\n")
      unless ($port = getservbyname($service, 'tcp'));

   die("550:Cannot gethostbyname($server)\r\n")
      unless ($iaddr = gethostbyname($server));

   die("550:Cannot create socket\r\n")
      unless socket($handle, PF_INET, SOCK_STREAM, $proto);

   die("550:Cannot connect($service://$server)\r\n")
      unless connect($handle, sockaddr_in($port, $iaddr));

   # unbuffered I/O to that service

   select($handle); $| = 1; select(STDOUT); $| = 1;

   return($handle);
}
```

See whois2ph(8), the whois2ph source, whois2html(8), and the whois2html source -- both are *production gateways* in Perl to interface with our whoisd(8) server.

Final Comments:

The whois example uses a line based protocol. The strategy is common but by no means universal. For example, the lpd protocols use octets (ie. single characters) for the commands.

Inetd servers are the simplest to implement. However, this may not be optimal. Especially if the server has to do a lot of work first (eg. loading in a big data base).

Stand alone servers have to deal with many daemon issues -they should ignore most signals, set a unique process group and get rid of the controlling terminal.

Daemons like nntp could (in theory) handle many clients from a single daemon using interrupt driven I/O. As currently implemented most have an nntp daemon for each client (but INN uses a single daemon for flooding).

You'll note that Socket programmers use alarm(2), setjmp(2), and signal(2) calls. The intent is to prevent a process (client or server) from hanging in a wait for I/O state by setting and trapping on an alarm.

Note Well:

- The best way to code a client/server program is to reuse code from an existing service. There's lots of public domain examples to work from -- nntp, lpd, sendmail, and even our whois service.
- A simple solution that works is much better than a fancy solution that doesn't -- ***keep it simple.***
- Presentation issues, ie. the display for the user, should not effect the protocol or server. Again, protocols have to be simple!
- Don't ever assume the client or server applications are well behaved!

Suggested Reading:

It shoud be clear that we have lots of real world examples you can look at and work from:

- *UWO/ITS Whois/CSO server*, by Reg Quinton, UWO/ITS, 1992-97
- *UWO/ITS Whois client*, by Reg Quinton, UWO/ITS, 1992-97
- *Passwdd/Passwd -- An authentication Daemon/Client*, by Reg Quinton, UWO/ITS, 1992-97
- *ACL -- Access Control Lists*, by Reg Quinton, UWO/ITS, 1995-97

More detailed documentation, should you need it, can be found at:

- *Introductory 4.3BSD Interprocess Communication*, by Stuart Sechrest, (in) UNIX Programmer's Supplementary Documents, Vol1, 4.3 Berkeley Software Distribution, PS1:7.
- *Advanced 4.3BSD Interprocess Communication*, by Samuel J. Leffler et al, (in) UNIX

Programmer’s Supplementary Documents, Vol1, 4.3 Berkeley Software Distribution, PS1:8.

- *Introduction to the Internet Protocols*, Computer Science Facilities Group, Rutgers. (See ftp:/ftp.uwo.ca/nic)
- *Networking with BSD-style Sockets*, by John Romkey, (in) Unix World, July-Aug. 1989.
- *How to Write Unix Daemons*, by Dave Lennert, (in) Unix World, Dec. 1988.
- *A Socket-Based Interprocess Communications Tutorial*, Chpt. 10 of SunOS Network Programming Guide.
- *An Advanced Socket-Based Interprocess Communications Tutorial,* Chpt. 11 of SunOS Network Programming Guide.

Author:

Comments, concerns, questions, etc. about these notes should be directed to the author:

Reg Quinton <reggers@julian.uwo.ca>
(for) The UWO Network Information Centre
Information Technology Services
The University of Western Ontario
London, Ontario N6A 5B7 Canada
+1 519 661-2151x6026

Copyright 1991-97: