An Introduction to Socket Programming

Last Edit
March 23, 1992

Reg Quinton <reggers@julian.uwo.ca>
Computing and Communications Services
The University of Western Ontario
London, Ontario N6A 5B7
Canada

1. Socket Programming

This course is directed at Unix application programmers who want to develop client/server applications in the TCP/IP domain.

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.

This course requires an understanding of C programming and an appreciation of the programming environment (ie. compilers, loaders, libraries, Makefiles and the RCS revision control system).

> **BEWARE:** If C code scares you, then you'll get some concepts but you might be in the wrong course.

Our example is the UWO `whois(1)` service — client and server sources available in:

```
julian:~ftp/pub/unix/networking/rwhois
```

2. Existing Services

On a Unix machine there are usually lots of TCP/IP services installed and running (tons on julian!).

```
[1:17pm julian] netstat -a
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address Foreign Address        (state)
tcp        0      0  julian.2717   vnet.ibm.com.smtp      ESTABLISHED
tcp        0      0  julian.smtp   uacsc2.albany.ed.55049 TIME_WAIT
tcp        0     13  julian.nntp   watserv1.waterlo.3507  ESTABLISHED
tcp        0      0  julian.nntp   gleep.csd.uwo.ca.3413  ESTABLISHED
tcp        0      0  julian.telnet uwonet-server2.c.55316 ESTABLISHED
tcp        0      0  julian.login  no8sun.csd.uwo.c.1023  ESTABLISHED
tcp        0      0  julian.2634   Xstn15.gaul.csd..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       *.*
```


2.1. Netstat Observations

Inter Process Communication, IPC, is between host.port pairs (or host.service). A process pair uses the connection — client and server applications.

Two protocols on IP -- TCP (Transmission Control Protocol) and UDP (User Datagram Protototocol). We'll be looking in more detail at TCP services and will not look at UDP at all.

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

We recognize many of the services — SMTP (Simple Mail Transfer Protocol, E-mail), NNTP (Network News Transfer Protocol service, Usenet News), NTP (Network Time Protocol), and SYSLOG is the BSD service implemented by `/etc/syslogd`.

The `netstat` 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).

3. 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`.

```
[10:25am suncon] page /etc/hosts
# Sun Host Database
#
127.0.0.1       localhost
129.100.1.75    suncon.ccs.uwo.ca
```

These days name to number translations are implemented by the Domain Name Service (or DNS) — see `named(8)`.

```
[10:26am suncon] page /etc/resolv.conf
nameserver 129.100.2.12
nameserver 129.100.7.100
nameserver 129.100.2.13
domain ccs.uwo.ca
[10:26am suncon] nslookup whohost
Server:  julian.uwo.ca
Address:  129.100.2.12

Name:    julian.uwo.ca
Address:  129.100.2.12
Aliases:  whohost.uwo.ca
```


3.1. Programming Calls

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

```
struct     hostent {
   char   *h_name;        /* official name of host */
   char   **h_aliases;    /* alias list */
   int    h_addrtype;     /* host address type */
   int    h_length;       /* length of address */
   char   **h_addr_list;  /* list of addresses */
};
#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`). Lazy programmers (like me) just use `h_addr` — the first address listed.

Client applications connect to a host.port (cf. `netstat` output) for a service.

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

The connection is usually prefaced by translating a hostname 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)
        error...
```

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 `htonl(3)` and `ntohl(3)` for host to net conversions.

4. 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: 1992/02/13 15:58:44 $
#
# 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...
```

4.1. Programming Calls

The C library routines `getservbyname(3)`, and `getservbyport(3)` 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;      /* official name of service */
   char   **s_aliases;  /* alias list */
   int    s_port;       /* port service resides at */
   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)
       error...
```


5. Socket Addressing

A Socket Address is a host.port pair (communication is between host.port pairs). The structure is `sockaddr_in`, the address family is `AF_INET`:

```
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)
       error...
    if ((hp=gethostbyname(host)) == NULL)
       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 is filling in the port and host address in the Socket Address structure — the address of the remote host.port where we want to connect.

There's a generic Socket Address structure, a `sockaddr`, used for communication in arbitrary domains.

```
/* from: /usr/include/sys/socket.h */

struct sockaddr {
    u_short sa_family;   /* address family */
    char    sa_data[14]; /* up to 14 bytes of direct add
};
```

The `sockaddr_in` structure is for Internet Socket Addresses. 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];
};
```

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`).

6. File Descriptors and Sockets

6.1. 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 1, and `stderr` as 3. But `stdin` is a `FILE` (see `stdio(3)`) not a file descriptor. If you want a stdio stream on a file descriptor use `fdopen(3)`.

6.2. Sockets

A Socket is a Unix file descriptor created by the `socket(2)` call — you don't `open(2)` or `creat(2)` a socket. Cf. `pipe(2)` creates file descriptors.

```
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.

The `type` parameter specifies the semantics of communication. For TCP/IP use SOCK_STREAM (for UDP/IP use SOCK_DGRAM).

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. But see `/etc/protocols` to get really confused.

7. Client Connect

A client application creates a `socket(2)` and `connect(2)` to a service.

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

The result returned is a file descriptor.

7.1. Client Communication

Having connected a socket to a server to establish a file descriptor communication is with the usual Unix I/O calls.

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

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

    if ((unit=tcpopen(WHOHOST,WHOPORT)) < 0)
        error...
    sockin=fdopen(unit,"r");
    sockout=fdopen(unit,"w");
    etc...
    fprintf(sockout,"WHOIS %s\n",argv[i]);
    etc...
    while (fgets(buf,BUFSIZ,sockin)) etc...
```


7.2. Stdio Buffers

Stdio streams have powerful manipulation tools (eg. `fscanf` is amazing). But beware, streams are *buffered*!

This means a well placed `fflush(3)` is often required to flush a buffer to the peer.

```
    fprintf(sockout,"WHOIS %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.

8. Server Applications

A system offers a service by having an application running that is listening at the service port for a connection. 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` 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.alaska.edu.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).

8.1. Server Bind

A Server uses `bind(2)` to establish the local host.port assignment. Only required for servers — applications which `accept(2)` connections from a host.port.

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

if ((sp=getservbyname(service,"tcp")) == NULL)
   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)
   error...
if (bind(s, &sin, sizeof(sin)) < 0)
   error...
```

`htonl` converts a long to the right sequence (given different byte ordering on different machines). The IP address `INADDR_ANY` means all interfaces.

Client applications usually aren't concerned about the local host.port assignment (the `connect(2)` does a bind for the local address). But `rcp`, `rlogin`, etc. do connect from reserved port numbers.

8.2. Listen and Accept

To accept connections, a socket is first created with `socket(2)`, a queue for incoming connections is specified with `listen(2)` and then the connections are accepted with `accept(2)`.

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

if ((sp=getservbyname(service,"tcp")) == NULL) error...
sin.sin_family=etc...
if ((s=socket(AF_INET,SOCK_STREAM,0)) < 0) error...
if (bind(s, &sin, sizeof(sin)) < 0) error...

if (listen(s,QUELEN) < 0) error...
for (;;) {
   if ((g=accept(f,&from,&len)) < 0) error...
   if (!fork()) {
       child handles request...
       exit(0);
   }
   close(g);
}
```

This is the programming schema used by utilities like `sendmail` and `lpd` — they create their socket and listen for connections.

9. 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(8)`. This is a generic service configured by the file `/etc/inetd.conf`.

```
[2:35pm julian] page /etc/inetd.conf
# $Author: reggers $
# $Date: 1992/02/13 15:58:44 $
#
# 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...
nntp    stream tcp nowait root   /usr/lib/newsbin/nntpd          nntpd
whois   stream tcp nowait nobody /usr/ccs/lib/directory/rwhoisd  rwhoi
tn3270  stream tcp nowait nobody /usr/ccs/bin/tn3270             tn327
account stream tcp nowait nobody /usr/ccs/bin/accountd           accou
```

9.1. Inetd Comments

For each service listed in `/etc/inetd.conf` the `inetd` process (this process is started at boot time) executes the socket, bind, listen and accept calls as discussed above. Inetd also handles many of the daemon issues (signal handling, set process group and controlling tty).

`Inetd` spawns the appropriate application (with `fork(2)` and `exec(2)`) when a client connects.

The application is started with `stdin` and `stdout` connected to the remote port. Any input/output on stdin/stdout are sent/received by the client.

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

9.2. Whois Daemon

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

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

This is the UWO `whois` service (as listed in `/etc/services`), on a TCP/IP stream, ran as user `nobody`, the program to run is listed, and the command line to the program.

Note that this is **not** the standard `/usr/ucb/whois` service that talks to `nic.ddn.mil`. The UWO whois service talks to a different server and implements a different protocol.

The program conducts a protocol on stdin/stdout (which is usually connected by a TCP/IP socket to a client application).

10. Running the Daemon

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

```
[3:27pm julian] /usr/ccs/lib/directory/rwhoisd
220 Directory Service $Revision: 1.1 $ ready at julian.u
help .... my command
350 I don't know much but I can understand
        HELP, QUIT, WHOIS
.
whois quinton .... my command
350 Matches on quinton follow:
Reg.Quinton:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930

reggers:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930

.
quit .... my command
220 Quit accepted, terminating session
[3:30pm julian]
```

The program is command driven — you give commands on stdin, it produces results on stdout.

10.1. The Code

The program is easy enough: read a line, switch on command and do command.

```
printf("220 Directory Service %s ready at %s\n", VERSION
fflush(stdout);

while (fgets(string,BUFSIZ,stdin))
{   if (isprefix(string,"HELP")) printf(HELPMSG);
    else if (isprefix(string,"QUIT"))
    {       printf("220 Quit accepted, terminating sessi
            fflush(stdout); sleep(3); exit(0);
    }
    else if (isprefix(string,"WHOIS "))
    {       sscanf(string,"%*s%*[ ]%[^ \r\n]",name);
            printf("350 Matches on %s follow:\n",name);
            fflush(stdout);
            sprintf(string,"%s '%s'",GREP,name);
            system(string);    printf(".\n");
    }
    else printf("550 command makes no sense\n");
    fflush(stdout);
}
printf("550 Oops... you've stopped talking\n");
```

The protocol is line based. This works well with stdio streams. Also easy to test from a terminal. Compare with line based protocols for NNTP and SMTP.

10.2. Connecting to the Server

You can make a `telnet(1)` connection to the server:

```
[3:47pm suncon] grep whois /etc/services
whois           43/tcp          nicname
[3:47pm suncon] telnet julian 43
Trying 129.100.2.12 ... Connected to julian.uwo.ca.
Escape character is '^]'.
220 Directory Service $Revision: 1.1 $ ready at julian.u
help .... my command
350 I don't know much but I can understand
        HELP, QUIT, WHOIS
.
whois quinton .... my command
350 Matches on quinton follow:
Reg.Quinton:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930

reggers:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930


.
quit .... my command
220 Quit accepted, terminating session
Connection closed by foreign host.
[3:48pm suncon]
```


10.3. Whois Client

The whois client makes a TCP/IP connection to the server and conducts the kind of protocol you would type if you where to make a connection by hand:

```
[7:30am zebra] whois quinton
Reg.Quinton:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930

reggers:
 Reg Quinton  <reggers@julian.uwo.ca>
 CCS,programmer,NSC,214,661-2151,6026,#11930

[7:30am zebra]
```

The client sends the command "`WHOIS quinton`", the server sends back the answer and the client displays the answer to the user. When finished the client sends "`QUIT`".

The server response codes assist in the parsing of the results.

The client code is complicated (a bit) by the piping through a pager.

11. 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 we have an `nntp` daemon for each client.

You'll note that Socket programmers use `alarm(3)`, `setjmp(3)`, and `signal(3)` calls. The intent is to prevent a process (client or server) from hanging in a wait for I/O state.

11.1. 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 — KISS. Protocols have to be 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!

12. Suggested Reading

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. (on ˜ftp/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.