Sockets In A Nutshell

The Berkeley socket interface provides a common interface to mechanisms by which processes running on the same or on different machines can communicate. Because of its generality (the Linux implementation supports 25 combinations of protocol families and socket types), it can often be daunting to the uninitiated. This document attempts a gentle introduction to socket programming for one protocol family (the Internet protocols) and one type of socket (stream sockets). This combination, of course, is TCP/IP, and provides reliable, connection-oriented, stream-based communication.

Documentation

The socket interface is well documented by man pages on Unix systems. However, it is a good idea to have a book such as Rick Stevens's "Unix Network Programming, Volumes I and II" handy.

Sockets

A socket is an endpoint for communication—a "doorway" to a process. Communication requires two such endpoints, a socket for the local process and a socket for the remote process, and a connection between them. Sockets have a type that consists of the protocol family they support, and the specific protocol within that family. Sockets are created using the socket() call. Consider the following.

```
int sock=socket(AF_INET, SOCK_STREAM,0);
```

This creates a TCP/IP socket. "AF_INET" indicates the Internet protocol family, while SOCK_STREAM denotes a stream-oriented protocol within that family. The only one of these is TCP. The final argument is used to select between different protocols if there is more than one that matches the first two arguments.

In Unix, a socket is a file descriptor, and so one can use all the interfaces that take file descriptors. These are described in a separate document.

Giving A Socket An Address

A doorway is useless if others cannot find it. Before you use a socket, you generally need to give it an address. In some cases, as we shall see later, the address is given implicitly. The form of the address depends on the protocol family. For Internet protocols, the socket address consists of an IP address and a port. The following is an example of how to bind a socket to port 1500 on the local host.

```
struct sockaddr_in sa;
memset(&sa,0,sizeof sa);
sa.sin_port=htons(1500);
sa.sin_addr.s_addr=htonl(INADDR_ANY);
sa.sin_family=AF_INET;
bind(sock,(struct sockaddr *)&sa,sizeof sa);
```

There are several things to note here. First, the functions htons() and htonl() are used to changed their arguments from the host byte order to the network byte order. Forgetting to do this is the source of many strange socket errors. Next, note that the IP address we bind to, INADDR_ANY, is a shortcut to say "any IP address that this host supports." If we wanted to, we could bind to a specific IP address as well. Finally, note that in terms of naming and addressing, sockets operate in a quite different manner than files. To open

a file, one supplies a name and gets a file descriptor. To open a socket, one creates a socket, getting a file descriptor, and then binds it to an address.

Connections

Communication requires a connection between two endpoints. We can think of a connection as being uniquely specified by the addresses if its endpoints (sockets), and by the protocol being used. For Internet protocols, then, a connection is a five-tuple: the IP address and port of the machine that actively opens the connection, the IP address and port of the machine that passively opens the connection, and the protocol (TCP, here).

Server

In opening a connection, we generally think of a machine that passively accepts the connection (the "server") and a machine that actively requests the connection (the "client"). Our code examples so far are for a server.

After binding its socket to a local IP address and port, the server must declare that the socket is ready to accept connections. It does so using the listen call:

```
listen(sock,5);
```

This call does two things. First, it tells TCP to begin accepting connections. Second, it asserts that TCP should try to accept and queue up to five connections that have not yet been accepted by the application.

The application can now accept a connection using the accept call:

```
struct sockaddr_in sa2;
int sock2 = accept(sock, (struct sockaddr *)&sa2, sizeof sa2);
```

Accept() will block until a client attempts to connect. When a connection request comes in accept, accept() will return a new socket (sock2) that can be used to communicate with the client. Sa2 contains the address of the remote (client) socket.

At this point, the server has a number of options. For example, it could accept another connection, fork and let its child handle the connection, pass the connection to another process or thread, or simply communicate with the client. For simplicity, we'll assume it does the latter.

A connected socket can simply be used like any other Unix file descriptor. For stream sockets, such as we are using in this example, the read() and write() calls can be used to communicate over the connection. Socket communication is bi-directional. A write sends data to the remote end of the connection, while a read receives data from the remote end. For simplicity, let's assume that our server will receive one (up to) 80 character line from the client, and then send it right back. We could implement this in the following way:

```
char buf[80]
int n=read(sock2,buf,80);
write(sock2,buf,n);
```

There are several things to point out here. First, the reads and writes are blocking. The server will stall in read() until data has been received from the client and in write() until the data has been handed off to TCP. Second, read() and write() may actually read and

write fewer bytes than requested. It is important to always look at the return value when using read() and write(). This will be the number of bytes actually read or written. Third, read() and write() may fail, returning a negative number and setting the global error code errno.

After all communication has been completed, the server must close the socket:

```
close(sock2);
```

The socket on which it is listening may also be closed if the server no longer wants to accept new connections:

```
close(sock);
```

Client

The client must also begin by creating an appropriate socket:

```
int sock=socket(AF_INET,SOCK_STREAM,0);
```

Next it creates the address to which it wants to connect:

```
struct sockaddr_in sa;
memset(&sa,0,sizeof sa);
sa.sin_port=htons(1500);
sa.sin_addr.s_addr=htonl(ipaddress_of_server);
sa.sin_family=AF_INET;
```

Notice that this address is identical to the address to which the server bound except that the IP address of the server is used. It makes no sense to connect to the wildcard address (INADDR_ANY). Also, notice that the socket is not bound to the address. We do not care what local IP address or port will be used, so we allow Unix to choose an address.

The client connects by issuing a connection request to the address:

```
connect(sock,(struct sockaddr *)&sa, sizeof sa);
```

Connect will block until the server's TCP has accepted the connection. If the server has not yet run listen() or it has more than five outstanding connections, the connection will be refused. Otherwise, the connection will be established.

Once the connection is established, the client is free to send data to the server using read() and write():

```
char *bufout="Hello";
char bufin[80];
write(sock,bufout,strlen(bufout)+1);
int n=read(sock,bufin,80);
```

As before, the reads and writes may fail or transfer fewer bytes than requested.

After the client is finished with the socket, it should close it:

```
close(sock);
```

Using DNS

The running example above used IP addresses. Generally, humans prefer to use DNS names. The socket interface includes functions for mapping these human-readable names to IP addresses and back. Here are some of the relevant functions:

gethostname()	gets the name of the local host
gethostbyname()	gets the IP addresses of the given hostname
inet_addr()	converts an IP address given as a string into a 32 bit integer