

TCP/IP Sockets

This handout describes the basics of using BSD Unix "socket" interface for making TCP/IP connections. BSD Unix and C are the historical starting point for a sockets, so although sockets work on many different platforms, they following the naming and operations started with BSD Unix.

Sockets were designed to add networking functionality with a coding style where reading and writing with sockets resembles reading and writing with files. Creating and connecting a socket is a little awkward, but once it is connected, it operates much like a regular file for reading and writing.

Sockets are very general — the same interface support connections of many different types — TCP/IP is just one of the channels which sockets can use. The generality is nice, but it complicates the API's somewhat as you will see.

The headers for the socket functions are in the following header files in /usr/include/ and each function typically has a description in the man pages.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
```

Generic Socket Address

```
// The generic socket address structure used by all socket types.
// This template must be able to hold any sort of address which
// sockets can connect to -- not just IP addresses.
// <sys/socket.h>

struct sockaddr {
    u_short  sa_family;    // AF_INET means an Internet address

    // bytes which depend on how this family does addressing
    // (some use more than 14 bytes anyway, so this struct is
    // just an overlay really)
    char     sa_data[14];
};
```

Internet Socket Address

```
// The specialized form of above for IP addresses..
//    <netinet/in.h>

// The 4-byte IP addr (a structure for historical reasons)
// Stored big-endian.
struct in_addr {
    u_long s_addr;
};

// Combines the IP addr and port #
struct sockaddr_in {
    u_short  sa_family;    // newer implementations divide this into
                          // length and family characters
    u_short  sin_port;
    struct in_addr  sin_addr;
    char     sin_zero[8];
};
```

GetHostByName()

```
struct hostent*  gethostbyname(const char*);
```

Takes a string form domain (DNS) name like "cse.stanford.edu" and attempts to find that host's IP address. This may take a little time since it will likely require contacting one or more remote name servers. The `struct hostent` contains assorted information about the host including alias names and multiple IP addresses. In particular the `h_addr` field will be the first IP address for the host stored in a big-endian format and the field `h_length` will be its size in bytes. On a little-endian machine, the address may look wrong if you print it as a single number, but it is in the correct (standard) form to pass to other socket calls.

The old (SunOS) man page for `gethostbyname()` was terrible — it failed to address the questions which bug-wary programmers always worry about: who allocates the memory and who owns it when? The Solaris version of the man page is much better. It defines that the system owns the struct, and that the next call to `gethostbyname()` will probably re-use it, so the caller should copy the information out.

GetServByName()

```
struct servent*  getservbyname(const char* service,
                              const char* protocol);
```

This function takes a name for a service like "finger" or "whois", and the protocol string "tcp" or "udp" and returns a `struct servent` which contains information about the name, port, and protocol for that pair. The one thing you tend care about is the port number in the field `s_port`. As before, the number is already big-endian. This call is probably consulting some local database to determine which port is being used for which service. On Unix, the database is in the file `/etc/services`, and it looks something like....

```

elaine10:/usr/include/sys> more /etc/services
## @(#)services 1.12 (Stanford)
# maintained by dennis@jessica.stanford.edu
# Do not edit this file directly, as it will be overwritten by bundle.
# See rfc1700 for the official list of assigned services
#
# Network services, Internet style
tcpmux      1/tcp
echo        7/udp
echo        7/tcp
discard     9/udp          sink null
discard     9/tcp          sink null
systat      11/tcp         users
systat      11/udp         users
daytime     13/udp
daytime     13/tcp
netstat     15/tcp
qotd        17/tcp          quote
qotd        17/udp          quote
chargen     19/tcp         ttytst source
chargen     19/udp         ttytst source
ftp-data    20/tcp
ftp         21/tcp
telnet      23/tcp
smtp        25/tcp          mail
...

```

This is not so important for services, like finger, which have well defined port numbers defined in their RFC. This does matter if you are programming a new service, and you want the sysadmins of the machines you are running on to be able to control the port number which your service uses. That's the theory anyway— in reality, new services just pick a port number and specify that that's what everyone use in the RFC, so the whole "looking up what port number to use" abstraction has turned out to not be very important.

There is a related function `getprotobyname(const char* protocol)` which can maps protocol names to a protocol enumeration— unlikely to ever be necessary for TCP/IP based sockets. This call and the one above are probably more interesting for non-TCP/IP uses of sockets where there is more variation in service, protocol, and port.

Socket()

```
int socket(int family, int type, int protocol);
```

- `family` use `PF_INET` (protocol family Internet) for a TCP/IP type socket. The same `socket()` function is capable of creating file sockets, etc. if you pass it different family constants.
- `type` For a TCP/IP socket there are two choices— use `SOCK_STREAM` for a connection oriented service (TCP), or `SOCK_DGRAM` for a connectionless datagram service (UDP).
- `protocol` Pass 0 to get the default which for TCP/IP is fine. For some other use of sockets, you might want to pass whatever `getprotobyname()` returned.

Allocates a new socket and returns its integer descriptor (like a file descriptor). Returns -1 on error and sets the global `errno` to an explanatory error code. A newly created socket is not yet pointing to any particular remote address.

Connect()

```
int connect(int socket, struct sockaddr*, int addrSize);
```

Connect a socket to a remote address. Returns 0 on success, -1 otherwise and sets `errno`. For TCP/IP communications, the second argument can be the address of a (`struct sockaddr_in`) filled in with the address of the remote machine, and the size should be `sizeof(struct sockaddr_in)`. The values to fill in the `struct sockaddr_in` come from calls to `gethostbyname()` and `getservbyname()`. At the time of the connection, the system will assign this end of the socket a temporary IP address and port number for the lifetime of the connection.

The core code to make a connection is...

```
struct servent *serviceInfo;
struct hostent *hostInfo;
struct sockaddr_in inetAddr;

int sock

//// Zero out the whole inetAddr
memset(&inetAddr, '\0', sizeof(inetAddr));

inetAddr.sin_family = AF_INET;           // "address family internet"

//// Lookup the port# for this server and put it in inetAddr
serviceInfo = getservbyname("finger", "tcp");
inetAddr.sin_port = serviceInfo->s_port;

//// DNS lookup to get IP addr, and put it into inetAddr
hostInfo = gethostbyname("binky.watsamatta.edu");
memcpy(&inetAddr.sin_addr, hostInfo->h_addr, hostInfo->h_length);

//// Create our socket
sock = socket(PF_INET, SOCK_STREAM, 0);    // or use SOCK_DGRAM for UDP
                                           // PF_INET = protocol family inet

//// Attempt to connect using our computed IP addr and port #
connect(sock, &inetAddr, sizeof(inetAddr));
...

```

The above code does no error checking at all which is quite unrealistic. Internet software runs into error cases all the time: bad host name, connection refused...Also, rather than hardwiring the host name and other values, this code should probably be hidden away in a general connection creating utility function which takes the hostname and port# (or service name) as parameters and returns the connected socket or an error code.

Write()

```
int write(int sock, char* buff, int buffLen);
```

Send some bytes out over a connected socket. May block in some cases, but often will return immediately having enqueued the write request to be processed asynchronously by the system. The bytes are not necessarily sent immediately. Returns the number of bytes written, or on error returns -1 and sets `errno`. In some cases, `write()` may not send all of the data, the number returned will be less than `buffLen`, and the client is supposed to `write()` again to send the rest of the data. This interface seems a little lame — it would be nice if there were a variation where the `write()` would just block until it had really written the bytes or errored out. The rule for API design is: "make easy things easy, make hard things possible." `Write()` violates the "easy things easy" part since even the simplest application of `write()` technically should have this retry-loop around every `write()`. Anyway, because the provided interface irritates me, you are free to never bother with the retry-loop. If `write()` returns less than `buffLen`, you may treat it as an error.

For a `SOCK_DGRAM` (UDP) connection, each call to `write()` sends all the data as one message.

Read()

```
int read(int sock, char* buff, int buffLen);
```

Read bytes from a socket into the given buffer. Returns the number of bytes read, or returns -1 and sets `errno` on error. The bytes in the buffer are not C-String null terminated; that's why it returns the int number of bytes. A return of 0 is an effective EOF— there are no more bytes. `Read()` will not necessarily fill the entire buffer every time it is called. Typically, many successive calls to `read()` will be required to get all the bytes, either because they will not all fit in the given buffer, or because not all the bytes have arrived the first time `read()` is called.

For a `SOCK_DGRAM` (UDP) connection, each call to `read` will return one entire message, assuming the client has provided a sufficiently large buffer.

Shutdown()

```
int shutdown(int sock, int direction);
```

A socket has both reading and writing ends. `Shutdown()` allows you to close one end while leaving the other end open. `Direction = 0` closes the reading end, `1` closes down the writing end, and `2` closes down both. Typically, you want to close the writing end when done writing, but leave the reading end open to receive the last bytes from the remote host. Closing the writing end will cause the remote host to get a 0 eventually when they call `read()`— it's like sending an EOF. If you think about it, the system cannot know you are not going to call `write()` any more until you explicitly indicate so. Some implementations (e.g. `finger`) get confused if you `shutdown` just your writing end — they think mistakenly think you are killing the whole dialog, where you really just meant that you weren't going to write any more.

For a `SOCK_DGRAM` (UDP) connection, `Shutdown()` and `Close()` just mark the local data, they do not communicate to the remote end. You will need to have your own scheme to communicate the status of the dialog to the remote end.

Close()

```
void close(int sock);
```

Terminates communication properly and removes the socket. If the socket descriptor is shared (because of a call to `fork()`), this just decrements the reference count and only really closes the socket when the count reaches 0.

Read/Write C String Code

Here is some typical reading and writing code which use C strings...

```
// Write a C string to a socket
void WriteString(int sock, const char* string) {
    int err;
    int len = strlen(string);
    err = write(sock, string, len);
    if (err < len) ErrorOut("Write error", errno);
}

// Do a single read() on a socket and put
// the result as a C string into the buffer.
// Returns the number of chars read.
int ReadString(int sock, char* buff, int buffSize) {
    int err;
    err = read(sock, buff, buffSize-1);
    if (err == -1) ErrorOut("Read error", errno);
    buff[err] = '\0';
    return(err);
}
```

Other Socket Calls

Sockets can be set to have different blocking behaviors. For example, a non-blocking `read()` will always return immediately, but a return of 0 no longer means EOF. In that case 0 just means there were no more bytes at that moment. Testing for the real EOF needs to be done separately. See the `fcntl()` function to blocking/non-blocking on a socket— we will use the default blocking sockets. To be on the "listening end" — waiting on a given port number for incoming connections, see the functions `accept()` and `listen()`.

Perl Socket Code

The Perl socket calls parallel the C versions. Here's what the socket interfaces look like in Perl. Each of the socket functions returns false for error conditions.

```
#!/usr/bin/perl
use Socket;

$port = 80; # or could call: getservbyname("http", "tcp");

$ipaddr = inet_aton($hostname); # like gethostbyname()

$sockaddr = sockaddr_in($port , $ipaddr); # form a socket addr

socket(SOCK, PF_INET, SOCK_STREAM, 0); # allocate the socket SOCK
## "SOCK" is an inherently global variable in Perl
## but it can be passed as with ILine() below

connect(SOCK, $sockaddr); # attempt to connect the socket
```

```
## this incantation sets SOCK to be unbuffered which is what we need
my $temp = select(SOCK);
$| = 1;
select($temp);
```

Writing is simple — just use `print`. Reading lines of text is actually somewhat complicated because of the interaction between the various end of line conventions and Perl's `<file>` line reading operator. So we provide the short `ILine()` function which reads a single "Internet" text line for you...

```
print SOCK "hello there\r\nI'm writing over a socket\r\n";

$line = ILine(SOCK); # Call ILine() to read an Internet line

# Given a socket, read and return a single text line
# with the EOLN conventions converted so that the line
# ends with a single '\n'. Returns false on EOF.
sub ILine {
    my ($sock) = @_;
    my($line);
    $line = <$sock>; ## read a single text line
    ## check if there's nothing (EOF case)
    if (!defined($line)) { return $line; }

    $line =~ s/[\r\n]//g; ## delete all the \r \n's
    return $line . "\n"; ## return something with exactly one '\n'
}
```