Networked Systems (H) laboratory exercise 1: Network Programming in C

Dr Colin Perkins
School of Computing Science
University of Glasgow
https://csperkins.org/teaching/2016-2017/networked-systems/

11 January 2017

Introduction

The laboratory exercises for Networked Systems (H) will introduce you to network programming in C on Unix systems using the Berkeley Sockets API, and help you understand the operation and structure of the network. The laboratory exercises build on your knowledge of C programming from the Advanced Programming (H) course last semester, and introduce network programming in C. Other exercises illustrate some key points in the operation of the network.

There are a mixture of formative and summative exercises. The formative exercises will give you practice in programming networked systems in C; they are not assessed. The summative exercises test your C programming skills, your ability to use the network by developing a networked system, and your understanding of the network. The exercises complement the lectures, which explain how the network operates, and how the protocols work.

This exercise is an introduction to TCP client/server programming in C, using the Berkeley Sockets API. It is a formative exercise, and is not assessed.

Background

The standard API for network programming in C is Berkeley Sockets. This API was first introduced in 4.3BSD Unix, and is now available on all Unix-like platforms including Linux, macOS, iOS, Android, FreeBSD, and Solaris. An almost identical API, known as WinSock, is available in Microsoft Windows.

The recommended reference book for the Berkeley Sockets API is W. R. Stevens, B. Fenner, and A. M. Rudoff, “Unix Network Programming volume 1: The Sockets Networking API”, 3rd Edition, Addison Wesley, 2003, ISBN 978-0131411555. Numerous on-line tutorials, of varying quality, also exist.

Creating a Socket

A *socket* provides an interface between the network and the application. There are two types of socket: a *stream socket* provides reliable and in-order delivery of a byte stream, while a *datagram socket* provides unreliable and unordered delivery of packets of data. When used in the Internet environment, stream sockets correspond to TCP/IP connections and datagram sockets to UDP/IP datagrams. This exercise will only consider TCP/IP socket programming, since this is the most widely used service, and forms the basis for most Internet applications.

To use the sockets API, you must first include the appropriate header files in your source code:

```
#include <sys/types.h>
#include <sys/socket.h>
```

You create a socket by calling the `socket()` function, as follows (the `...` indicates omitted code, where you insert error handling and application logic):

```
...
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd == -1) {
  ... // an error occurred
}
...
```

The `socket()` function takes three parameters. The first, `AF_INET`, selects the Internet address family, IPv4; `AF_INET6` would create an IPv6 socket. The second parameter, `SOCK_STREAM`, creates a TCP socket; use `SOCK_DGRAM` to create a UDP datagram socket. The final parameter is not used with TCP or UDP sockets, and set to zero. On success, this function returns an integer known as a *file descriptor* that identifies the newly created socket. If an error occurs, the function returns -1 and sets the global variable `errno` to indicate the type of error (you can bring the definition of the `errno` variable include scope using `#include <errno.h>`. The Unix man page for the `socket()` function lists the possible errors, e.g., `EACCES`, each of which corresponds to a `#define` in the `<errno.h>` header file.

A newly created socket is not connected to the network. How you connect it to the network depends whether you are making a server socket that waits for and accepts incoming connections from clients, or a client socket that can make a connection to a server.

Implementing a TCP Server

To turn a newly created socket into a TCP server, you bind the socket to a *port*, on which it listens for connections. The clients connect to a server based on the combination of a network address and port number. After establishing the connection, either client or server can write data into their socket, where it becomes available for the other to read.

Each different type of server uses a different port number. Port numbers are 16-bit unsigned integers, in the range 0-65535. Some port numbers are well known, for example web servers use port 80, while others are more obscure. The IANA maintains the master list of registered port numbers for different types of server (see `http://www.iana.org/assignments/service-names-port-numbers/`). Port numbers 0-1023 are generally reserved for system services, and are not accessible to non-admin users on Unix-like systems.

The process for making a TCP server is therefore to create a socket, bind that socket to a port, listen for connections on that port, then loop accepting new connections and responding to requests.

You bind a socket to a port using the `bind()` function. This takes three parameters: a file descriptor, `fd` that was previously returned from a `socket()` call, a pointer to an address structure that contains the port number, and the size of that structure:

```
...
if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
  ... // an error occurred
}
...
```

The `bind()` function returns -1 and sets `errno` if an error occurs; it returns zero on success.

The `addr` parameter to the `bind()` function is a pointer to a variable of type `struct sockaddr`. The `<sys/socket.h>` header defines this as follows (the exact size of the `sa_data` field will depend on what types of socket the system supports):

```
struct sockaddr {
  uint8_t         sa_len;
  sa_family_t     sa_family;
  char            sa_data[22];
}
```

A variable of type `struct sockaddr` can hold any type of address. The `sa_len` and `sa_family` fields hold the size and type of the address, while the `sa_data` field contains the data, and is large enough to contain any address. Applications *do not* use `struct sockaddr` directly. Rather, if they use IPv4 addresses, they instead use a `struct sockaddr_in`:

```
struct in_addr {
  in_addr_t       s_addr;
}

struct sockaddr_in {
  uint8_t         sin_len;
  sa_family_t     sin_family;
  in_port_t       sin_port;
  struct in_addr  sin_addr;
  char            sin_pad[16];
}
```

or for IPv6 addresses, they use a `struct sockaddr_in6`:

```
struct in6_addr {
  uint8_t         s6_addr[16];
}

struct sockaddr_in6 {
  uint8_t         sin6_len;
  sa_family_t     sin6_family;
  in_port_t       sin6_port;
  uint32_T        sin6_flowinfo;
  struct in6_addr sin6_addr;
}
```

All the `sockaddr...` types are defined in the <`netinet/in.h`> header; don't define them yourself.

You'll note that a `struct sockaddr_in` is exactly the same size in bytes as `struct sockaddr`, and has the `sin_len` and `sin_family` fields in exactly the same place as the `sa_len` and `sa_family` fields. Similarly, the `sin_port`, `sin_addr` and `sin_pad` fields replace the `sa_data` field. A `struct sockaddr_in` can therefore be freely cast to a `struct sockaddr`, and *vice versa*, since they have the same size, and the common fields are in the same place in memory. The same is true for IPv6, with the `struct sockaddr_in6`. This allows for a primitive form on sub-classing, where the sockets functions take a generic structure (`struct sockaddr`) while the program uses a variant specific to IPv4 (`struct sockaddr_in`) or IPv6 (`struct socaddr_in6`) and casts to/from `struct sockaddr` as appropriate.

When calling `bind()` to create a server socket, you therefore create an appropriate `sockaddr...` structure, fill in the family and port number, indicate that any available network address is suitable, and leave the other fields unspecified. For example, to create an IPv4 server bound to port 80, you'd write:

```
struct sockaddr_in  addr;
...
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_family      = AF_INET;
addr.sin_port        = htons(80);
```

```
if (bind(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
  ...  // an error occurred
}
...
```

The use of `INADDR_ANY` lets the server use any available network interface, if it's running on a host with more than one network connection.

After binding a socket to a port, you instruct the operating system to begin listening for connections on that port by calling `listen()`:

```
...
int backlog = 10;
...
if (listen(fd, backlog) == -1) {
  ... // an error occurred
}
...
```

The value of the `backlog` parameter to the `listen()` function specifies the number of simultaneous clients that can queue up waiting for the server to accept their connections, before it starts giving a "connection refused" to new clients (note: this is *not* the maximum number of clients that can connect, but rather the maximum number of clients that can be waiting for the server to `accept()` their connection at once; a value of 10 is reasonable unless your server is slow to accept connections).

The server calls `socket()`, `bind()`, and `listen()` once to create the listening socket, then calls the `accept()` function in a loop to accept connections from clients:

```
...
while (...) {
  struct sockaddr_in cliaddr;
  socklen_t          cliaddr_len = sizeof(cliaddr);

  int connfd = accept(fd, (struct sockaddr *) &cliaddr, &cliaddr_len);
  if (connfd == -1) {
    ... // an error occurred
  }
  ...
}
```

On success, `accept()` returns a *new* file descriptor, called `connfd` in this example, representing the connection to the client. It sets `cliaddr` to the address of the client, and `cliaddr_len` to the length of that address. If there are no clients waiting, then the `accept()` function blocks until a client connects.

Implementing a TCP Client

To turn a newly created socket into a TCP client, call the `connect()` function. This takes three parameters: the file descriptor of the newly created socket, the address and port of the server it should connect to (cast to a `struct sockaddr`), and the size of the address:

```
struct sockaddr_in addr; // Or struct sockaddr_in6 if using IPv6
...
if (connect(fd, (struct sockaddr *) &addr, sizeof(addr)) == -1) {
  ... // an error occurred
}
...
```

The difficulty with `connect()` is knowing what is IP address of the server, to put into the `struct sockaddr` (you usually know the DNS name of the server, but not the IP address). The port number is assumed to be known, since it's fixed for each application.

You can look up the IP address of a server given a DNS name using the `getaddrinfo()` function. This function takes as parameters the server name, port, and hints about the type of address required, and returns a linked list of possible IP addresses for the server (a server can have multiple IP addresses if it has multiple network connections for robustness against network outages, or if it has both IPv4 and IPv6 addresses). You then need to iterate through the list of addresses, trying each in turn until you succeed in making a connection to the server.

For example, to connect to a server called "www.example.com" on port 80, you would use code like:

```
...
struct addrinfo  hints;
struct addrinfo *ai0;
int              i;
...
memset(&hints, 0, sizeof(hints));
hints.ai_family   = PF_UNSPEC;   // Can use either IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // Want a TCP socket
if ((i = getaddrinfo("www.example.com", "80", &hints, &ai0)) != 0) {
  printf("Error: unable to lookup IP address: %s", gai_strerror(i));
  ...
}
// ai0 is a pointer to the head of a linked list of struct addrinfo
// values containing the possible addresses of the server; interate
// through the list, trying to connect to each turn, stopping when
// a connection succeeds:
struct addrinfo  *ai;
int               fd;

for (ai = ai0; ai != NULL; ai = ai->ai_next) {
  fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
  if (fd == -1) {
    // Unable to create socket, try next address in list
    continue;
  }
  if (connect(fd, ai->ai_addr, ai->ai_addrlen) == -1) {
    // couldn't connect to the address, try next in list
    close(fd);
    continue;
  }
  break; // successfully connected
}
if (ai == NULL) {
  // Couldn't connect to any of the addresses, handle failure...
  ...
} else {
  // Successfully connected: fd is a file descriptor of a socket
  // connected to the server; use the connection
  ...
}
```

The Unix manual page for the `getaddrinfo()` function explains all the fields of the `struct addrinfo`, and explains how to use this function in more detail.

Sending and Receiving Data

Once the client has connected to the server, you can send and receive data over the connection using the `send()` and `recv()` functions. The `send()` function takes as arguments a file descriptor of a connected socket, a pointer to the data (`char *`), the size of the data in bytes, and a set of flags. A useful flag to set is `MSG_NOSIGNAL`, which makes `send()` to return an error if the far end closes the connection, rather than delivering a `SIGPIPE` signal (which kills the process, if not handled):

```
char   *data = ".....";
size_t  data_len = ...;
int     flags = MSG_NOSIGNAL;
...
if (send(fd, data, data_len, flags) == -1) {
  ... // error
}
...
```

On success, `send()` returns the number of bytes it sent (which might be less than `data_len`, if the network is congested). If an error occurs, it returns -1 and sets `errno`. A call to `send()` can take a long time to complete, depending on the speed of the network and the amount of data sent.

The `recv()` function take as parameters a connected socket, a pointer to a buffer in which to store the data, the size of the buffer, and a set of flags (a flag that's sometimes useful is `MSG_PEEK`, which peeks at incoming data without removing it from the connection, but it's usual not to set any flags). On success, `recv()` returns the number of bytes read. If an error occurs, it returns -1 and sets `errno`:

```
#define BUFLEN 1500
...
ssize_t  rcount;
char     buf[BUFLEN];
int      flags = 0;    // No flags set
...
rcount = recv(fd, buf, BUFLEN, flags);
if (rcount == -1) {
  ... // error
}
...
```

Note that the `recv()` function *does not* add a terminating zero byte to the data it reads, so it is unsafe to use string functions on the data unless you add the terminator yourself. Note also that `recv()` will silently overflow the buffer and corrupt memory if the buffer length passed to the function is too short. *These are significant security risks, so be careful.*

A TCP connection buffers data, so data written with a single call to `send()` might arrive split across more than one `recv()` call. Alternatively, data from more than one `send()` request might arrive in a single `recv()` call. TCP sockets deliver data reliably and in-order, but the timing and message boundaries are not necessarily preserved.

The `recv()` call will block if there is nothing available to read from the socket. The `send()` call will block until it is able to send some data (`send()` may return after sending only some of the data requested, if the network is heavily congested - you need to check the return value to see if it sent all the data).

Handling Multiple Sockets

It's common to have more than one socket open at once. An example might be a server that has multiple clients connected, perhaps a web server that is serving multiple pages at once.

On modern multi-core systems, the best way of handling multiple sockets is often to use multiple threads, one per socket, to send and receive data, since this allows each socket to proceed concurrently with the others. One thread uses `bind()` and `listen()` to setup the socket, then calls `accept()` in a loop to accept and process new connections. The file descriptor returned from each call to `accept()` is passed to a new thread that handles the connection, sending and receiving data as needed, and closes the file descriptor when done. Since there is only one listening socket, there's no benefit to calling `accept()` from multiple threads: the correct pattern is to have one thread accepting connections, and pass those new connections to members of a thread pool for processing.

When passing the file descriptor from the accepting thread to the thread that will handle the connection, it's important to avoid a race condition with the following `accept()` call. Use `malloc()` to allocate space, copy the file descriptor into that space, and pass the copy to the thread, to avoid looping round and overwriting the file descriptor with the results of the next call to `accept()`.

An alternative to multi-threading is to use the `select()` function, which monitors multiple sockets to see if they have data available to receive, or space to send. This lets a single thread handle more than one socket. An example of using `select()` to receive from two sockets, with a timeout, is show below:

```
#include <sys/select.h>

int            fd1, fd2;
fd_set         rfds;
struct timeval timeout;

timeout.tv_sec  = 1;   // 1 second timeout; pass NULL as the last
timeout.tv_usec = 0;   // argument to select() for no timeout

FD_ZERO(&rfds);        // Create the set of file descriptors to
FD_SET(fd1, &rfds);    // select upon (limited to FD_SETSIZE as
FD_SET(fd2, &rfds);    // as defined in <sys/select.h>)

// The nfds argument is the value of the largest file descriptor
// used, plus 1 (note: not the number of file descriptors used).
int nfds = max(fd1, fd2) + 1;

int rc = select(nfds, &rfds, NULL, NULL, &timeout);
if (rc == 0) {
  ... // timeout, with nothing available to recv()
} else if (rc > 0) {
    if (FD_ISSET(fd1, &rfds)) {
        ... // Data is available to recv() from fd1
    }
    if (FD_ISSET(fd2, &rfds)) {
        ... // Data is available to recv() from fd2
    }
} else if (rc < 0) {
  ... // error
}
```

Using `select()` can be more efficient when connections are very short-lived, or if there are many more connections than processor cores so it's necessary to manage multiple connections per thread. The `select()` call is portable, but is slow when given a large number of connections to monitor. If you need to monitor hundreds or thousands of file descriptors at once, there are non-portable alternative such as `epoll()` on Linux or `kqueue` on FreeBSD/macOS that scale better. Alternatively, libraries such as `libuv` that provide portable abstractions for event-based I/O that efficiently support very large numbers of simultaneous connections.

Closing Connections

Once you have finished with the connection, call the `close()` function to terminate it:

```
close(fd);
```

Remember to close the connection at both the client and the server ends. For a server socket, close the per-connection file descriptor when you have finished with that connection, and the close underlying file descriptor when you have finished accepting new connections.

When closing a connection, the client should call `close()` before the server. This is because the system that calls `close()` will enter a "time wait" state where it doesn't allow another socket to bind to the same port until some timeout has expired, to make sure any late arriving data doesn't end up in the wrong connection. A consequence is that, if the server crashes or otherwise closes the connection before the client, then you won't be able to restart it until the timeout expires (the `bind()` call will fail and return an `EADDRINUSE` error). Setting the `SO_REUSEADDR` socket option avoids the timeout, but introduces a security hole by allowing data from a previous connection to appear on the new connection, so should not be used.

Formative Exercise 1: Networked "Hello, World" Application

The first formative exercise demonstrates how to build a simple client-server application using TCP/IP. You should write two programs:

`hello_server` The server should listen on TCP port 5000 for incoming connections. It should accept the first connection made, receive all the data it can from that connection, print that data to the screen, close the connection, and exit. If TCP port 5000 is in use on your systems, pick another port number instead – the choice of port is unimportant for this exercise.

`hello_client` Your client should connect to your chosen TCP port on the host named on the command line, send the text "Hello, world!", then close the connection. The client should take the name of the machine on which the server is running as its single command line argument (i.e., if the server is running on machine `bo720-1-01u.dcs.gla.ac.uk` you should run your client using the command `hello_client bo720-1-01u.dcs.gla.ac.uk`.

Run your client and server, and demonstrate that you can send the text "Hello, world!" from one to the other. Try this with client and server running on the same machine, and with them running on two different machines. Once this is working, modify your client to send a much longer message (more than 1500 characters), and check that works too.

When you have large messages working, modify your client and server so that the server can send a message back to the client. The client should send "Hello, world!", then wait for and display the message sent back by the server. The contents of the message returned by the server are unimportant.

Finally, modify your server to handle connections from multiple clients at once.

Write a simple Makefile to compile your code, rather than running the compiler by hand. You are *strongly advised* to enable all compiler warnings (at *minimum*, use `clang -W -Wall -Werror`, noting that the three occurrences of the letter W are capitalised), and to fix your code so it compiles without warnings. Compiler warnings highlight code which is legal, but almost certainly doesn't do what you think it does (i.e., they show the location of bugs in your code). Use them to help you find problems.

This exercise is not assessed, and you do not need to submit your code. The assessed exercises later in the course build on the skills you will learn in completing this formative exercise, however, so you should complete this exercise carefully, and seek help if you have any questions about the material.