Introduction to Network Programming in C

Networked Systems (H) 2021-2022 – Laboratory Exercise 1

Dr Colin Perkins, School of Computing Science, University of Glasgow

1 Introduction

The laboratory exercises for Networked Systems (H) will introduce you to network programming in C using the Berkeley Sockets API, and help you understand the operation and structure of the network. The exercises will help you practice C programming, building on the Systems Programming (H) course, and introduce network programming in C. Other exercises will illustrate key points in the operation of the network. The laboratory exercises are intended to complement the material covered in the lectures. Some expand on the lectures to give you broader experience in a particular subject. Others exercises cover material, such as network programming in C, that's better taught by doing than by lecturing.

There are a mixture of formative and summative exercises. The formative exercises will give you practice in programming networked systems in C, and allow you to gain further experience in C programming. They are not assessed. The two summative exercises will focus on security and protocol ossification, and on understanding the network topology. They are assessed and together are worth a total of 20% of the marks for this course.

This exercise is an introduction to network programming in C using the Berkeley Sockets API. **This is a formative exercise, and is not assessed.**

2 Formative Exercise 1: Networked "Hello, World" Application

The first formative exercise demonstrates how to build a client-server application that uses a TCP/IP connection for communication. It builds on material from the Networks and Operating Systems Essentials course, and on your C programming experience from Systems Programming (H). You should write two C programs:

`hello_server` The server should listen on a 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 the server on the port you have chosen, send the text "Hello, world!", and then close the connection. The client should take the name of the host on which the server is running as its single command line argument (i.e., if the server is running on `stlinux03.dcs.gla.ac.uk` you should run the client using the command `hello_client stlinux03.dcs.gla.ac.uk`.

The client and server should ideally be run on two different machines. You can do this by running two instances of `ssh`, to connect to two of the student Linux servers (`stlinux01.dcs.gla.ac.uk` to `stlinux08.dcs.gla.ac.uk`) provided by the School, or you can use any other two machines that you have access to and between which communication is permitted (note that the University network has a firewall that blocks incoming TCP connections, so you cannot run the client on your home machine and the server on one of the student Linux servers).

If you only have access to one machine, you can run client and server on the same machine. In this case, when running the client, use `localhost` as the name of the machine running the server (the `localhost` is an alias for "this machine").

The material in the later sections of this handout provides guidance on how to write these programs, and presentation slides summarising the material are available on Moodle. **You are strongly advised to read this entire handout before starting the exercise.**

Write a simple Makefile to compile your code, rather than running the compiler by hand (if you don't know how to do this, ask the lecturer or one of the lab demonstrators). 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 – that is, they show the location of bugs in your code. Use them to help you find problems.

Run your client and server, and demonstrate that you can send the text "Hello, world!" from one to the other. If possible, 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. If it does not, discuss with the lecturer or lab demonstrators why not.

When you have large messages working, modify your client and server so that the server can send a message back to the client through the open connection. 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.

This exercise is not assessed, and you do not need to submit your code. Other 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.

3 Background: An Introduction to Network Programming in C

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 definitive 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 free on-line tutorials, of highly variable quality, exist. Lecture 2 also covers some of this material.

You can freely reuse any of the code fragments below, both in these exercises and in future programs you may write.

3.1 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 in a C program, you must first include the appropriate header files at the start of your code:

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

You create a socket by calling the `socket()` function, as shown in the following code fragment (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
}
...
```

This code fragment goes inside your `main()` function, or in some other function you have written. The `socket()` function call takes three parameters:

- The first parameter specifies the network layer protocol used by the socket. Specify `AF_INET` to create an IPv4 socket or `AF_INET6` to create an IPv6 socket.
- The second parameter specifies the transport layer protocol to be used. Use `SOCK_STREAM` to create a TCP stream socket or `SOCK_DGRAM` to create a UDP datagram socket.
- The final parameter is not used with TCP or UDP sockets, and is set to zero.

On success, the `socket()` function returns an integer known as a *file descriptor* that identifies the newly created socket. When an error occurs, the function returns -1 and sets the value of a special global variable `errno` to indicate the type of error (the global variable `errno` is defined by the C runtime; you don't need to define it yourself). The documentation for the `socket()` function lists the possible error codes, which are defined in the <`errno.h`> header file. The standard library function `perror()` can be used to print an appropriate error message, based on the value of `errno`.

The `socket()` function creates a socket, but does not connect it to the network. How you use the socket depends whether you are making a server that waits for and accepts incoming connections from clients, or a client that can make a connection to a server.

3.2 Implementing a TCP Server

To turn a newly created socket into a TCP server, it is necessary to bind the socket to a *port* on a network interface, then tell it to listen for connections. Clients then connect to the server using the combination of the IP address and port number of the server. After establishing the connection, the client and server can exchange data.

Port numbers range from 0-65535 (i.e., they're 16-bit unsigned integer values). Some port numbers are reserved for well known applications, for example `http` servers use port 80 and `ssh` servers used port 22, while other ports are unassigned. The Internet Assigned Numbers Authority (IANA) maintains the master list of what port numbers are assigned to different applications (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.

To make TCP server application, is first necessary to create a TCP socket as shown in Section 3.1. That socket is then bound to a port and told to listen for connections. Finally, the server loops, accepting new connections on the port and responding to requests. To bind the socket to a port, use the `bind()` function, as shown in the following code fragment:

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

The `bind()` function takes three parameters:

- a file descriptor, `fd`, representing a socket, as returned by a call to the `socket()` function;

- a pointer to a socket address (cast to type `struct sockaddr *`) that specifies the network interface and port number to which the socket should be bound; and
- the size of the socket address.

On success, `bind()` returns zero. If an error occurs, `bind()` returns -1 and sets `errno`.

The second parameter given to the `bind()` function is the socket address. This determines the network interface and port number to which the server should be bound. It is given as a pointer to a variable of type `struct sockaddr`. The `<sys/socket.h>` and `<netinet/in.h>` headers define this as follows (you don't need to copy this definition, just `#include` both of those headers):

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

A network interface can have either an IPv4 or an IPv6 address, and so a variable of type `struct sockaddr` needs to be able to hold either type of address. Unfortunately, the C programming language is not object oriented and does not have the ability to support sub-types. Instead, the `sa_len` and `sa_family` fields hold the size and type of the IP address, and the `sa_data` field contains the data and is large enough to hold either type of address. Applications *do not* use `struct sockaddr` directly. Rather, if they use IPv4, they instead use a `struct sockaddr_in`, defined as:

```
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];
}

struct in_addr {
  in_addr_t       s_addr;
}
```

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

```
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;
}

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

The `sockaddr_in`, `in_addr`, `sockaddr_in6`, and `in6_addr` types, are defined in the standard `<sys/socket.h>` and `<netinet/in.h>` headers. Don't define them yourself, just `#include` those headers.

You'll note that a `struct sockaddr_in` is exactly the same size in bytes as `struct sockaddr`. The `sin_len` and `sin_family` fields are in exactly the same order and have the same size as the `sa_len` and `sa_family` fields in the `struct sockaddr`. Similarly, the `sin_port`, `sin_addr` and `sin_pad` fields take up exactly the space occupied by the `sa_data` field of a `struct sockaddr`.

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 bind() function takes a generic structure (struct sockaddr) that can hold either sort of IP address, but the application uses a variant specific to IPv4 (struct sockaddr_in) or IPv6 (struct socaddr_in6) and casts it to struct sockaddr when calling bind().

If the above description is not clear, then ask the lecturer or one of the lab demonstrators for an explanation – this behaviour illustrates several important, but non-obvious, behaviours of the C programming language.

When calling bind() to create a server socket, you therefore create either a struct sockaddr_in or struct sockaddr_in6, fill in the address family, network interface address, and port number, then cast it to struct sockaddr before use. For example, to create an IPv4 server bound to port 80, you'd write:

```
struct sockaddr_in  addr;

addr.sin_family      = AF_INET;    // IPv4
addr.sin_addr.s_addr = INADDR_ANY; // Any available network interface
addr.sin_port        = htons(80);  // Port 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. For IPv6 server, use AF_INET6 instead of AF_INET and IN6ADDR_ANY instead of INADDR_ANY.

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 very busy or slow to accept connections).

The server calls socket(), bind(), and listen() once to create the listening socket. At this point, the socket has been created, bound to a port, and is listening for incoming connection. The server can then call the accept() function in a loop to accept new 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. The listening file descriptor, `fd`, remains available and can be used to accept connections from new client in future iterations of the loop. A common mistake is to confuse these two file descriptors. The variable `cliaddr` is set to the address of the client that has just connected, and `cliaddr_len` to the length of that address. If there are no clients waiting to connect, then the `accept()` function will block until a client connects.

Once a connection to a client has been accepted, the server can send and receive data as described in Section 3.4.

3.3 Implementing a TCP Client

To turn a newly created socket into a TCP client, rather than a TCP server, call the `connect()` function. The `connect()` function takes three parameters: the file descriptor of a socket created as shown in Section 3.1, the socket address of the server to which it should connect, and the size of the socket address. Similarly to the `bind()` function, the socket address is stored in a variable of type `struct sockaddr_in` if IPv4 is used, or of type `struct sockaddr_in6` if IPv6 is used, and cast to a `struct sockaddr *` on use:

```
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 the IP address to put into the `struct sockaddr`, since you usually know the DNS name of the server, e.g., `www.example.com`, but not its IP address. The port number is 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:

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

struct addrinfo  hints;
struct addrinfo *ai0;
int              i;

// Set the hints:
memset(&hints, 0, sizeof(hints));
hints.ai_family   = PF_UNSPEC;   // Unspecified protocol (IPv4 or IPv6 okay)
hints.ai_socktype = SOCK_STREAM; // Want a TCP socket


// Perform a DNS lookup. The variable ai0 is set to point to the head
// of a linked list of struct addrinfo values containing the possible
// addresses of the server.
if ((i = getaddrinfo("www.example.com", "80", &hints, &ai0)) != 0) {
  printf("Error: unable to lookup IP address: %s", gai_strerror(i));
  ...
}
```

```
struct addrinfo  *ai;
int               fd;

// Iterate over the linked list of results of the DNS lookup, trying to
// connect to each in turn, breaking out when successfully connected.
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 documentation for the `getaddrinfo()` function explains all the fields of the `struct addrinfo`, and explains how to use this function in more detail. If you don't understand this code, or how to integrate it into your client, ask the lecturer or one of the lab demonstrators.

3.4 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 (on a server, make sure you use the `connfd` returned by `accept()` and not the listening file descriptor), 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()` return an error if the far end closes the connection, rather than delivering a `SIGPIPE` signal that 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). Again, if implementing a server, be sure to use the file description for the connection, rather than the listening file descriptor. 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.

Incorrectly handling the data returned by `recv()` can lead to significant security vulnerabilities, so be careful and ask the lecturer or one of the demonstrators if you're not sure why this is a concern, or how to safely use the data returned..

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, and potentially then call `send()` again to send any outstanding data).

3.5 Handling Multiple Sockets

It's sometimes necessary for a server to have more than one socket open and listening for connections 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 that particular 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 the sockets representing 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 previously created sockets, `fd1` and `fd2`, 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 (tens of thousands) of simultaneous sockets to monitor. If you need to monitor many thousands of file descriptors at once, there are non-portable alternative such as `epoll()` on Linux or `kqueue` on FreeBSD/macOS that scale better, and libraries such as `libuv` provide portable abstractions for event-based I/O that efficiently support very large numbers of simultaneous connections.

Note, however, that modern systems run efficiently with many thousands of threads. The MacBook Air laptop on which these notes are being written is currently running 2,849 active threads, on a four-year-old Intel Core i5 with 16GB of memory, in normal use. Code that just creates one thread for each connection will scale to many thousands of simultaneous connections without difficulty, on a relatively low-end server.

3.6 Closing Connections

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

```
#include <unistd.h>
...
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 (i.e., `connfd` in the example code) when you have finished with that connection, and the close underlying file descriptor (`fd` in the example code) 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). It is possible to set the `SO_REUSEADDR` socket option to avoid the timeout, but this introduces a security hole by allowing data from a previous connection to appear on the new connection, so should not be used.

- + -