NS3 Lab 1 – TCP/IP Network Programming in C

Dr Colin Perkins
School of Computing Science
University of Glasgow
http://csperkins.org/teaching/ns3/

21/22 January 2014

Introduction

The laboratory exercises for Networked Systems 3 (NS3) will introduce you to network programming in C on Unix/Linux systems, and help you understand the operation of the network. There are weekly laboratory sessions for this course, during which you will complete several exercises. These exercises will build on your knowledge of C programming from the Advanced Programming 3 (AP3) course last semester, and on the material in the NS3 lectures. There are a mixture of formative and summative exercises. The formative exercises are intended to give you practice in programming networked systems in C; they are not assessed. The two summative exercises test your ability to program networked systems.

This is NS3 lab 1, an introduction to TCP client/server programming in C. It comprises one formative C programming exercise that is expected to take 1-2 hours and should be completed in the first two laboratory sessions. This laboratory exercise is not assessed, but is important to help you understand network programming, and also the principles of networked systems.

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 X, FreeBSD, and Solaris. A very similar network API is available on Windows. The standard 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.

Creating a Socket

Sockets provide a standard interface between the network and application. There are two types of socket: a *stream socket* provides a reliable byte stream transport service, while a *datagram socket* provides unreliable delivery of individual data packets. 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 stream sockets, as they are by far the most widely used.

To use the sockets API, you must first include the appropriate header files:

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

Once the headers have been included, you can create a socket by calling the `socket` function. This function returns an integer known as a *file descriptor* that identifies the newly created socket.

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

(the ... shows that some application code has been omitted). The `AF_INET` parameter to the `socket()` call indicates that the Internet address family (IPv4) is to be used; `AF_INET6` would create an IPv6 socket. The `SOCK_STREAM` parameter indicates that a TCP stream socket is desired; use `SOCK_DGRAM` to create a UDP datagram socket. The final parameter is not used with TCP or UDP sockets, and is set to zero. The socket that is created is unbound, and not yet connected to the network. If an error occurs, -1 is returned and the global variable `errno` is set to indicate the type of error. The Unix man page for the `socket` function lists the possible errors.

A TCP/IP connection provides a reliable byte-stream connection between two computers. TCP/IP is most commonly used in a client-server fashion: the server creates a socket, binds it to a well-known *port*, and listens for connections; meanwhile, the client creates a socket, and connects to the specified port on the server. Ports are identified by 16-bit unsigned integers, and the IANA maintains a list of well-known ports for different applications. Once the connection is established, either client or server can write data into the connection, where it is available for the other party to read.

TCP Server Sockets

To make a newly created socket into a TCP server, it is first necessary to bind that socket to the appropriate well-known port. This is done using the `bind()` function, specifying the file descriptor, address, and port on which you wish to listen for connections:

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

As with the `socket()` function, -1 is returned and `errno` is set if an error occurs. The `addr` parameter to the `bind()` function is specified to be a pointer to a variable of type `struct sockaddr`. This structure is defined in the `<sys/socket.h>` header as follows:

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

The `struct sockaddr` is intended to be generic. The `sa_len` and `sa_family` fields hold the size and type of the address, while the `sa_data` field is large enough to hold any type of address. It treats the address as an opaque piece of binary data.

Applications do not use a `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;
}
```

You will note that the `struct sockaddr_in` is exactly the same size in bytes as the `struct sockaddr`, and has the `sin_len` and `sin_family` fields in exactly the same places as the `sa_len` and `sa_family` fields. The `sa_data` field of the `struct sockaddr` is replaced by the `sin_port`, `sin_addr` and `sin_pad` fields, which have the same total size. 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 laid out in memory in the same way. The same is true for IPv6, with the `struct sockaddr_in6`. These definitions allow for a primitive form on sub-classing, where the Sockets API takes a generic superclass (`struct sockaddr`) while the program uses a subclass specific to IPv4 (`struct sockaddr_in`) or IPv6 (`struct socaddr_in6`) and casts to/from `struct sockaddr` as appropriate.

When creating a server socket, you generally do not care about the address used by the server host, and so only specify the address family (IPv4 or IPv6) and the port. For example, an IPv4 server listening on port 80 would use (the equivalent code for IPv6 would use `struct sockaddr_in6` instead):

```
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
  ..
}
...
```

Once a socket has been bound to a port, you can instruct the operating system to begin listening for connections on that port using the `listen()` function:

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

The value of the `backlog` parameter to the `listen()` function specifies how many simultaneous clients can be queued up waiting for their connections to be accepted by the server before the server returns a "connection refused" error to new clients (note: this is *not* the maximum number of clients that can be connected at once, 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 and also slow to accept connections).

Finally, you can accept a new connection by calling the `accept()` function:

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

Note that if there are no clients with connections queued up, then the `accept()` function will block until a client tries to connect. The `cliaddr` parameter will be filled in with the address of the client, and the `cliaddrlen` parameter will be set to the length of that address. The `accept()` function returns a *new* file descriptor, called `connfd` in this example, that represents the connection to this particular client (i.e., it's a new socket, connected to the client at the client at `cliaddr`). The original listening socket, and the corresponding file descriptor, remain untouched, and you can call `accept()` again on it to accept the next connection. A server application will generally operate in a loop, repeatedly calling `accept()` to accept connections from new clients and passing them to a thread for processing, then looping back to accept the next client.

TCP Client Sockets

A TCP client can connect to a server by calling the `connect()` function. This function takes three parameters: the file descriptor of a 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()` comes with knowing the IP address of the server, to put into the `struct sockaddr`, since usually only the DNS name of the server is known and not the IP address.

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 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 {
  // at this point, 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 client and server are connected, you can send an receive data over the connection using the `write()` and `read()` functions. The `write()` function takes as arguments a file descriptor of a connected socket, a pointer to the data (`char *`), and the size of the data in bytes. It returns the number of bytes sent, or -1 if an error occurs:

```
char *data;
int   datalen;
...
if (write(fd, data, datalen) == -1) {
  // An error has occurred...
  ...
}
...
```

The `read()` function take as parameters a connected socket, a pointer to a buffer in which to store the data, and the size of the buffer; it returns the number of bytes read, or -1 if an error occurred:

```
#define BUFLEN 1500
...
ssize_t  rcount;
char     buf[BUFLEN];
...
rcount = read(fd, buf, BUFLEN);
if (rcount == -1) {
  // An error has occurred...
  ...
}
...
```

Note that the `read()` 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 `read()` will silently overflow the buffer and corrupt memory if the buffer length passed to the function is too short.

A TCP connection buffers data, so that data written with a single call to `write()` might be returned split across multiple `read()` calls. Alternatively, data from several `write()` requests might be combined and returned by a single `read()` call. Data will be delivered reliably and in-order when using TCP sockets, but the timing and record boundaries are not necessarily preserved.

The `read()` call will block if there is nothing available to read from the socket. The `write()` call will block until it is able to send some data (`write()` may return after sending only some of the data requested, if the network is heavily congested).

Closing the Connection

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

```
...
close(fd);
...
```

Remember to close both the client and server sockets. For the server socket, close the per-connection fd when you have finished with that connection, and the underlying fd when you have finished accepting new connections.

Formative Exercise 1: Networked "Hello, World" Application

The first formative exercise demonstrates how to build the most 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, read all the data it can from that connection, print that data to the screen, close the connection, and exit.

`hello_client` Your client should connect to TCP port 5000 of a 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 `bo720-1-01.dcs.gla.ac.uk` you should run your client using the command `hello_client bo720-1-01.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 (several thousand 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.

Check that your server can handle connections from multiple clients without crashing, and without needing to be restarted.

You are *required* to write a simple Makefile to compile your code, rather than running the compiler by hand. You are also *strongly advised* to enable all compiler warnings (at *minimum*, use `gcc -W -Wall -Werror`), 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. Use them to help you find problems.

This exercise is not assessed, and you do not need to submit your code. The summative assessed web server exercise later in the course builds on the skills you will learn in completing this formative exercise, however, so you are advised to complete this exercise carefully, and seek help if you have any questions or problems in doing so.