CS 645 Computer Networks	**Fall 1999**
Assignment 0	
Assigned: 10 September 1999	*Due: 21 September 1999*

Overview

In this assignment you will develop simple programs, written in C, which interact with a "CS 645 Server" running on a remote machine (`gladstone.dcs.uky.edu`, IP address 204.198.76.142) using the sockets interface. The server waits for TCP connections on TCP port 7531. When a client establishes a connection to that port, the server waits for the client to initiate communication.

The information exchange between client and server in the first part of the exercise proceeds as follows:

1. The client program sends a request message (a string of ASCII characters) to the server, identifying itself and the transaction it is requesting.
2. The server then sends two lines of ASCII text to the client, the second containing a random number of characters.
3. The client sends back a number equal to the number of characters in the second server line (i.e. that number represented as a string of ASCII digits), and closes its end of the connection. So if the server sent 132 bytes followed by `\r\n`, the client would send "132".
4. The server sends back a response indicating whether the client's input was correct.

In the second part of the assignment, you will write additional code to extend the interaction to include a second connection, initiated by the server.

The primary objectives of this assignment are:

- to illustrate the client-server paradigm;
- to show you how programs actually access network services;
- to introduce you to the UNIX *sockets* interface.

Setup and Preparation

To complete this assignment, you may use any UNIX system that is connected to the campus network. This assignment requires that you have some understanding of the use of the `sockets` API for connection-oriented interprocess communication. To that end, you will be given a copy of the TCP/IP *Pocket Socket Guide*, which provides an introductory overview of how things work, some examples, and some reference information. You will also need to read carefully the manual ("man") pages for the `socket()`, `connect()`, `send()`, `recv()`, `close()`, `bind()`, `listen()`, `accept()`, and `shutdown()` routines, if you are not already familiar with them. (The *Pocket Socket Guide* includes a synopsis of each but you will need the manual pages for details.)

Note: To read the man page for the "foo" routine, type "`man foo`". This will print the first man page encountered that is titled "foo". However, if you want the man page for the "foo" *system call* and there's also a *command* foo, you have to give additional input to the `man` command to get it to give you what you want. Unfortunately this extra info varies from system to system. But you should be able to do `man man` to find out how it works on your system.

Note Also: On **SunOS** and **linux** systems, the routines making up the socket interface are *system calls*. Among other things, this means that they are included in the standard C library so you don't have to do anything special at compile time, and their manual pages are in section 2. On **Solaris** systems the routines are *not* system calls but library routines, which means the man pages are in section 3 (actually section 3N). More importantly, it means that your code must be linked with the `sockets` and `nsl` libraries. This is done as part of the compilation process:

```
gladstone% [g]cc [flags] your_file.c -lsocket -lnsl ...
```

Also, "Solaris" —for reasons known only to Sun— is the same as "SunOS 5". If you run the command "`uname -a` on a Solaris system, it will say "**SunOS ... 5.5.1**" or something of that form.

Grading Criteria

You will turn in a writeup comprising your well-commented code and its output. You will be graded on (i) the correctness of your code; and (ii) its readability and structure. Note that the correctness is easier to determine if the code is readable and well commented. Also note that each exercise requires you to get a "cookie" from the server; your writeup should this cookie.

Exercise 0: Initiating Connections

In this exercise you will write a client and use it to interact with the 645 Server according to the protocol described next.

The Protocol (Part I)

The client and server communicate by exchanging lines of printable ASCII characters via the reliable byte-stream service provided by TCP. (For this lab, the socket type is `SOCK_STREAM`, and the protocol family is `PF_INET`.)

The interaction between the client and server for this exercise proceeds as follows:

1. The server "listens" for connections on port 7531 of `gladstone.dcs.uky.edu` (IP address `204.198.76.142`).

2. The client opens a connection to the server's socket.

3. The server accepts the connection and waits to receive a request string from the client.

4. Once the client is connected to the server, it immediately sends a request string. The format of the request string is:

`CS645` ⟨WS⟩ ⟨request type⟩ ⟨WS⟩ ⟨username⟩ ⟨WS⟩ ⟨endpoint-specifier⟩ ⟨newline⟩

where:

- ⟨WS⟩ is "whitespace", one or more blank or tab characters.
- The ⟨request type⟩ is a string of the form "`typei`", where `i` is either `0` or `1`. For this exercise use `type0`.
- The ⟨username⟩ is the student's (i.e. your) name, in the form of first initial, middle initial, last name, all one word with no whitespace ("`K.L.Calvert`"). The username must contain only printable ASCII characters, and may be **no more than 64 characters** in length.
- The ⟨endpoint specifier⟩ is of the form

  ⟨dotted quad⟩-⟨port number⟩

  The endpoint specifier in this exercise refers to the *client's* socket.
- ⟨newline⟩ is the end-of-line marker, namely the two-character sequence "`\r\n`".

Thus, if the client's socket is bound to port 12981 on `sac.uky.edu`, the request string looks like:

```
CS645 type0  I.M.Funny  128.163.1.163-12981\r\n
```

Note that the maximum length of the request string, excluding whitespace and the terminating `\r\n` sequence, is 96 characters.

5. Upon receiving and parsing the request string, the server responds by sending a greeting terminated by "`\r\n`". If the request string was correctly formed, the line looks something like:

   ```
   hello 128.163.1.163-12981, welcome to the CS645 server\r\n
   ```

   If the request is not properly formed, or does not contain the proper endpoint identifier, the line will contain an error message.

6. After sending the greeting line, the server sends a random number of bytes, followed by `\r\n`. (The random bytes may include `\r` and `\n` individually, but not together in sequence.)

7. The client returns a string of ASCII characters (specifically, decimal digits) indicating the number of bytes sent by the server, *excluding the terminal* `\r\n`. This number is delimited by the end of the connection. That is, after sending it, the client closes its (sending) side of the connection by calling `shutdown()`.

8. The server then sends a single line of the form:

   ⟨outcome string⟩ ⟨WS⟩ ⟨cookie⟩ `\r\n`

   and closes the connection. The ⟨outcome string⟩ is either "OK" or a brief description of any error that occurred, and the ⟨cookie⟩ is a random number of around 9 decimal digits. The server also records the cookie in its log, so the cookie serves as proof that the client completed a transaction.

Client Operation

To implement the above protocol, the client does the following:

(i) Create a socket (of type `SOCK_STREAM` and family `PF_INET`) using `socket()`.

(ii) Fill in a `sockaddr_in` data structure with the host IP address and port number respectively) of the 645 server. To convert a "dotted quad-port" string to a socket address, i.e. to fill in a `sockaddr_in` structure, use the `inet_addr()` (3N) routine to convert a "dotted quad" string to the correct 32-bit unsigned integer representation, and `atol()` (3) to convert the port number.

(iii) Call `connect()` with the filled-in `sockaddr_in` data structure to initiate a connection to the server. If unsuccessful, print out the reason for the error and exit.

(iv) Construct a request string using the client's address information. To determine the client address information, you will need to use the `getsockname()` routine (see the man page).[1] You may want to use `sprintf()` to format the request line.

(v) Send the request to the server (using `send()`).

(vi) Read data from the socket (using `recv()`) until you encounter the `\r\n` sequence. Verify that the server accepted your request string. Then read and count bytes until you encounter the `\r\n` sequence again.

(vii) Send a string which is the decimal representation of the number of bytes received, then close the sending side of the connection using the `shutdown()` call.

(viii) Read the result and cookie line from the server.

Note that what you send should look *exactly* as specified above. In particular, the first character of any line *must* be non-whitespace, or the server will complain of a syntax error. Also, you must take care not to send extra characters after the end-of-line terminator. Remember that the TCP byte-stream service doesn't know anything about null-terminated strings. That is, if you are going to use `printf()` to display what you receive over the network, you must make sure you put in a null character to terminate the string after calling `recv()`.

Writeup

Turn in your well-commented code along with a record of the information your client received from the server, including the random number provided in step 8 of the protocol.

[1] When the socket is first created, it is not bound to any address. At connect time, however, the system implicitly assigns an address to the socket. The IP address will be the host's IP address, while the port number is chosen more-or-less arbitrarily. The point is the desired IP address information can't be obtained using `getsockname()` until after the socket is connected to the server.

Exercise 1: Listening for Connections

In this part of the exercise, the client sends a request with the same format as above but with a different request type ("`type1`"). For this type of request, the server interprets the endpoint specifier as an endpoint to which *it* (the server) should initiate a connection. After receiving the request line, the server sends the greeting string indicating a well-formed request, followed by a single line containing a cookie string. Then it tries to open a new connection to the indicated host and port. If it succeeds in connecting to the address specified in the request, the server sends the string "`hello\r\n`", then waits to receive the *same cookie* that it just sent on the *original* connection, followed by end-of-stream. Upon receiving the cookie, the server sends a *new* cookie and closes its end of the second connection. Finally, the client sends the *second* cookie to the server on the original connection, and closes it as before (i.e. this step looks like Step 7 in the first protocol, except the number has a different meaning). The server then sends the outcome message (including a third cookie) on the *original* connection, and closes it as before.

Thus, in this exercise, the roles of the client and server are swapped during part of the transaction, with the client passively waiting for a connection initiated by the server. You will need to write additional code to create a second socket, bind it to a port, and accept and use this latter connection.

The Protocol (Part II)

The first several steps of the protocol are identical to Part I. The request format is identical to that of Part I, except that the request type is 1 instead of 0. The endpoint-specifier in this request, however, *refers to a different socket address than the client side of the initial connection*. (This is because there can be only one connection between any pair of (IP,port) pairs.) In other words, the client has to have two sockets for this exercise.

The protocol diverges from Part I at Step 6: (**Note:** Throughout this description, the terms "server" and "client" refer to the **same programs** as they did above, i.e. the server is running on `gladstone`.)

6a The server sends a second line containing a random number (no more than 10 decimal digits).

7. The server then initiates a connection to the socket address specified in the request. This address must refer to a host and port on which the client (or some other program) is "listening" for a new connection. That is, after receiving the confirmation from the server, the client must be prepared to accept a connection on the specified address. Note that either the host address or the port number must be *different* from the ones used by the client's existing connection to the server.

8. After accepting the second connection, the client waits to receive a greeting line from the server.

9. The client then sends the cookie from step 7 above, and shuts down its end of the connection (i.e. to indicate "I have no more to send").

10. The client then waits to receive a second cookie (call it c_2) on the new connection. (This is to prove that it successfully communicated over the second connection.) After sending it, the server closes the second connection.

11. The client sends c_2 on the *original* connection, then shuts down its end of that connection. (The second connection is now defunct.)

12. The server sends an outcome string on the original connection and closes it (i.e. this is exactly the same as step 8 above).

"Client" Operation

The steps followed by the client for this part of the protocol are:

a. Create a socket (call it `psock`) of type `SOCK_STREAM` and family `AF_INET` using `socket()`.

b. Bind the socket `psock` to an available port, using `bind()`. Note that by calling `bind()` with the special IP address `INADDR_ANY` and port number 0, the system will bind the socket to a "random" available port on the host. Alternatively, you can choose a "permanent" port number for your client and bind to it explicitly.[2]

c. Find out what port `psock` was bound to using the `getsockname()` system call.[3] (If you specified an explicit port number in the call to `bind()`, and that call returned success, you don't need to do this.) Print out the address and port, so the user can see what's going on.

d. Call `listen()` on `psock` so it will accept new connections.

e. Construct the request message to be sent to the server. The process is similar to Part I, but the second address has a different meaning (see above).

f. Open the connection to the server (using a socket *other* than `psock`) and send the request.

g. Receive the greeting string from the server; if it indicates a correct request line, receive the next line containing the cookie. Otherwise close the connection, print the error message, and exit.

h. Call `accept()` on `psock`; if successful, this will return another socket (call it `newsock`) for the second connection, initiated by the server.

i. Receive the greeting line on the `newsock` connection.

j. `send()` the cookie on the `newsock` connection, and then call `shutdown` to close the sending part of that connection (i.e. with argument 1).

k. Receive the second cookie on `newsock`.

l. Send the second cookie on the original connection, then shut down the sending side (with `shutdown()`).

---

[2]Note that you must specify the port number to the system in **network byte order**. That means (if it's other than 0) that you must use the routines `htons()` to put the port number in the correct byte ordering before calling `bind()`. If you are using a little-endian machine (e.g. Intel architecture) this step is critical. See the *Pocket Socket Guide* and the man page fo `htons()` for details.

[3]Note that the IP part of the socket address will be specified as 0 (wildcard) if that is what was specified when it was bound. To determine the IP address of the socket, you can use the `gethostname()` and `gethostbyname()` routines.

m. Receive the result line including the third cookie on the original connection. Print out information about the result, and exit.

All you need to do is follow the steps above *in the order given*, using the routines according to the man pages. In particular, be sure to call `listen()` on `psock` *before* sending the initial request to the server. Otherwise, the server connection may arrive before `psock` is ready to accept connections, and it will be refused.

Writeup

Turn in your code and a record of your program's interactions with the server, including all three cookies receved.

Other Notes

The server handles only one client at a time. If a client request arrives while another client is being serviced, it will be queued until the first client is finished. There is a system-imposed limit (typically five) on the number of connections that can be queued in this way—see the man page for `listen()`. In the unlikely event that your client's connection request arrives and this queue is already full, it may be "refused" (your `connect()` call will fail with the error ECONNREFUSED. In this case, wait a few seconds and try again.

Because the server is single-threaded, it has only limited patience. That is, it will only wait a few seconds for the client to take any action it is supposed to take, and will try to ensure that the entire transaction lasts at most 10 seconds. This is required so that a misbehaving client cannot tie up the server and thus deny service to other clients. But your client should be prepared to wait up to 50 seconds for an initial response from the server (but that should be extremely rare).