COSC 465: Computer Networking
Lab 6: netcat clone
Due: April 4, 2012, before lab

1 Overview

For this lab, you will write a program similar to a UNIX utility called `nc`, aka "netcat". `nc` is similar to the standard utility `cat`: whereas `cat` will print out everything from a file (or anything on `stdin`) to `stdout`, `nc` will take anything received on `stdin` and send it on a network connection (specifically, a TCP connection). `nc` is different from `cat` in that it also can *receive* information back from the remote host. (The standard `cat` utility just works in one direction only.)

You'll write a program similar to `nc` called `netcat` that can behave either like an internet server or an internet client, and copy everything received from the keyboard (`stdin`) to the remote host. The program will stop when the other participant closes the connection.

Your `netcat` program will use TCP as a transport protocol, which is the main challenge for this lab. Also, your program will need to implement both server and client behavior as described below. Nonetheless, your source code should be relatively short.

Basic templates for `netcat.cc` and `SConstruct` are posted on Moodle.

2 Detailed Description

The `netcat` program will take certain command-line arguments depending whether it is started as a server or as a client. If started as a server, you will need to supply a `-s` flag as well as the TCP port number on which to listen. For example:

```
prompt> ./netcat -s 12000
```

If you start `netcat` as a client, you will need to provide the IP address or host name and port of the server to connect to, e.g.:

```
prompt> ./netcat cs.colgate.edu 12000
```

The same program file (`netcat.cc`) will need to implement both the server and client functionality. You'll know from whether you receive the `-s` flag if you need to run as a server or as a client.

For implementing `netcat`, I suggest that you start with the server functionality. Once you have that implemented, you'll be able to test it by using the `telnet` program to connect to the IP address and port on which your server is listening in order to test it (as described below). Once you have the server functionality (mostly) correct, you can work on the client-side functionality.

Server functionality

For a `netcat` server, you will need to wait and listen on the given port number until you receive a TCP connection from a client. Once a connection has been made, you will need to (a) print all data received from the network to `stdout`, and (b) send all data received from `stdin` to the client.

The basic steps (API calls) for implementing a TCP server are as follows:

1. `socket`. The arguments to `socket` will be a little bit different for creating a TCP socket. Instead of `SOCK_DGRAM` for UDP, you should say `SOCK_STREAM` to get a TCP socket.

2. `bind`. Next, you need to "bind" your server to a local port number and IP address. Prepare a `struct sockaddr_in` by filling in the `sin_family` and `sin_port`, but leaving the `sin_addr` field as 0. If you leave the address as zeroes, the OS will assume that you want to accept connections on any of the IP addresses configured on the local machine. You can call `bind` with your socket and the `sockaddr_in` structure to make this binding happen.

   Note: immediately after creating the server-side socket and *before* calling `bind`, you may wish to use the `setsockopt` system call to ask the OS to allow your program to re-bind to a given port without having to wait for a "2 X MSL" timeout (2 maximum segment lifetimes). The call is like this:

   ```
   // assuming your socket variable is named sock
   int opt = 1;
   setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
   ```

3. `listen`. Now, you need to tell the OS how many outstanding clients you're willing to service. The argument to `listen` is just one number, and it specifies the size of a queue of pending requests for your server. For this program, you can just set this queue size to 1. (Note that if we were writing a web server, we'd set this queue size the maximum allowable size specified by the operating system.)

4. `accept`. Finally, you're ready to tell the OS that you're ready to accept a connection from a client. When you call `accept`, your program will "block" until a connection is received from a client. The return value from `accept` is a new socket descriptor that you should use for communicating with the newly connected client. The previous socket that you created in step 1 can be closed (you only need to accept one connection for every time the server is started). See the `accept` man page for more details.

5. Now that you've accepted a connection from a client, you can use the `send` or `recv` calls to send or receive data to/from the client. `send` and `recv` are slightly easier to use than `sendto` and `recvfrom`; see their respective man pages for details.

   Important Note! TCP sockets work a bit differently than UDP sockets in that they work on *byte streams* rather than discrete chunks of data. In particular, with UDP sockets, `recvfrom` behaves in an all-or-nothing manner (the entire message is sent, or you get failure). With TCP, you may get partial success when calling `send`: be careful to check the return value to find out exactly how much data got sent. This implies that you may need to call `send` more than once to complete sending a large chunk of data.

You should be able to handle text from `stdin` or from the network in arbitrary order. That is, your program should not *block* on one or the other. This probably means that you should use `poll` (or `select`) to test whether there is data to read from `stdin` (file descriptor 0) or from your network socket. To read from `stdin`, you can just use the `getline` method on the `std::cin` object (see `http://en.cppreference.com/w/cpp/io/basic_istream`).

Your server program should exit if (a) the other side (the client) closes the TCP connection, or if (b) EOF is received on `stdin`. There's an `eof` method you can call on `std::cin` to check for EOF. For detecting whether the client closes the connection, check out the man page for `recv`.

When your program exits, be sure that you're closing all sockets and releasing any heap-allocated memory.

Client functionality

For the client-side part of `netcat`, you will make a TCP connection to the remote host and port given on the command line. Then, you'll do basically the same as you did with the server once the connect is made: (a) print all data received from the server to `stdout`, and (b) send all data received from `stdin` to the server.

The steps for connecting to a server using TCP are as follows:

1. `socket`. Again, the arguments to `socket` will be slightly different for creating a TCP socket. Instead of `SOCK_DGRAM` for UDP, you should say `SOCK_STREAM` to get a TCP socket.

2. `connect`. Once you have a socket and create an appropriately initialized `sockaddr_in` structure (i.e., that has the server's IP address and port number), you can connect to the server. See the man page for `connect` for the specific arguments to this function call. The call to `connect` essentially blocks until the three-way TCP handshake completes.

3. Once `connect` returns successfully, you can use `send` and `recv` and send and receive data to/from the server. These are slightly easier to use than `sendto` and `recvfrom`; they basically perform the same function.

   Important Note! TCP sockets work a bit differently than UDP sockets in that they work on *byte streams* rather than discrete chunks of data. A note about this is found above in the description of the server-side implementation.

The same caveat as with the server implementation applies: don't block on either receiving data from the network or from `stdin`. For that, you'll need to use `poll` (or `select`). As noted above, the file descriptor number for `stdin` is 0.

Your client program can exit in one of three ways: (a) if the server closes the connection you should exit, (b) if you haven't received data from the server for 10 seconds you should close the connection, or (c) if you receive an EOF on `stdin`. For EOF, you can "simulate" an end of file by typing `Ctrl+D` on a line by itself. In the case of an EOF, the client should wait for a short period of time until it appears that the server-side has no more data to send (for example, wait for 500 milliseconds). Note that this timeout is a bit different than the 10 second idle timeout.

Finally, it's important to note that the only real difference between the server functionality and client functionality is in the network setup. After that, they should behave basically the same. Your code structure should reflect that and avoid duplicating functionality. My solution code is only about 250 lines long, so yours should probably not be too much longer than that (and possibly shorter).

Testing

Once your server-side functionality is implemented, you can use the `telnet` program to test it. First, start your server:

```
prompt> ./netcat -s 12001
```

Now, use `telnet` to connect to your server (`telnet` should be available on all Linux and UNIX-like systems):

```
prompt> telnet <server ip address or host name> 12001
```

If the connection is successful, you should be able to type anything in the telnet window and see it appear on your server-side, and vice-versa.

Once you have the server-side functionality working, you can then use your own server to test the `netcat` client-side functionality.

A fun way to test your client is to use it with any web server by redirecting a file containing an HTTP request to your `netcat` program. For example, if the following lines are stored in the file `request.txt` (the two blank lines after the line with 'Host:' are important in the HTTP protocol, so you'll need them to make this work right):

```
GET /cs HTTP/1.1
Host: cs.colgate.edu


```

You can then redirect the contents of this file to `netcat` and connect to a web server, as in:

```
prompt> ./netcat cs.colgate.edu 80 < request.txt
prompt> # can also use the following to pipe contents into netcat:
prompt> cat request.txt | ./netcat cs.colgate.edu 80
```

If your client program works right, you should see the HTTP response and contents of the web page. Fun stuff! We'll use this capability for testing with the next lab.

3 Submission

Submit your `netcat.cc` source code to Moodle. Be sure to note in the comments at the top of the file who the authors are (esp. if you work with a partner).