Operating Systems. Homework #5.

Submission - June 21, 23:55.

In this exercise, you will implement a toy client/server architecture: a *printable characters counting server*. The exercise also involves some self-study, because you will need to learn how to convert host names (like www.google.com) to IP addresses.

You need to implement two programs:

1. `pcc_server`: The server accepts TCP connections from clients. A client that connects sends the server a stream of *N* bytes (*N* depends on the client, and is not a global constant). The server counts the number of printable characters in the stream (a *printable character* is a byte whose value is in the range [32,126]). Once the stream ends, the server sends the count back to the client over the same connection.
   In addition, the server maintains a data structure in which it counts the number of times each printable character was observed in all the connections. When the server receives a SIGINT, it prints these counts and exits.
2. `pcc_client`: The client creates a TCP connection to the server and sends it *N* bytes read from `/dev/urandom`, where *N* is a user-supplied argument. The client then reads back the count of printable characters from the server, prints it, and exits.

Client Specification

*Command line argument*s:

1. *Server host*: Assume it is either a valid IP address or (possibly invalid) host name.
2. *Server port*: Assume it is a 16-bit unsigned integer.
3. *Length*: The number of bytes to read from `/dev/urandom` and send to the server. Assume it is an unsigned integer.

High-level flow:

1. Create a TCP connection to the specified *server port* on the specified *server host*. (To do this, you may need to convert the host name to an IP address first. See details later.)
2. Open `/dev/urandom` for reading.
3. Transfer the specified *length* bytes from `/dev/urandom` to the server over the TCP connection.
4. Obtain the count of printable characters computed by the server, *C* (an unsigned int), and print it to the user in the following manner:
   ```
   printf("# of printable characters: %u\n", C);
   ```
5. Exit with code 0.

Remarks:

- You define the protocol used over the TCP connection between the client and the server. All that matters is that in the end, the client receives the number of printable characters computed on the server, and stores it in an `unsigned int` that is printed as described above.

- On error, print an error message containing the `errno` string (i.e., with `perror()` or `strerror()`) and exit with a non-zero return code.
- There's no need to clean up file descriptors or free memory when exiting.
- Do not assume a bound on the length of the *server host* argument.
- Do not assume a bound on the size specified by the *length* argument, except that it's specified by an `unsigned int` (i.e., at most 2^{32}-1).

Server Specification

Command line arguments:

1. *Server port*: Assume this is a 16-bit unsigned integer.

High-level flow:

1. Initialize a data structure `pcc_total` that will count how many times each printable character was observed in all client connections.
2. Enter a loop, in which you:
    A. Accept a TCP connection on the specified *server port*.
    B. When a connection is accepted, start a Client Processor thread. This thread reads a stream of bytes from the client, computes its printable character count and writes the result to the client over the TCP connection. This thread also updates the `pcc_total` global data structure.
    In the main thread, continue accepting new connections.
3. If the user hits Ctrl-C, perform the following actions:
    A. Stop accepting TCP connections.
    B. Wait until all active Client Processor threads (if any exist) finish computing their printable character counts and updating the `pcc_total` global data structure.
    C. Print out the number of times each printable character was observed by a Client Processor threads (including those threads waited for in step B).
    The format of the printout is the following line, for each printable character:
    `char '%c' : %u times\n`
    D. Exit with exit code 0.

Remarks:

- You can `bind()` to the address INADDR_ANY to accept connections on all network interface. See the `ip(7)` manual page for more details.
- You define the data structures required to implement the above specification.
- You decide the synchronization needed. The only requirements are:
    - A Client Processor thread must update the `pcc_total` global data structure *once*, and the time it takes to perform the global update should not depend on the number of bytes sent by the client.
    - Client Processor threads must be able to read/write their TCP connections in parallel.
- On error, print an error message containing the `errno` string (i.e., with `perror()` or `strerror()`) and exit with a non-zero return code.

- There's no need to clean up file descriptors or free memory when exiting.

Example Execution

client console	server console

`$ pcc_client server 2233 4`
`# of printable characters: 1`
	`$ pcc_server 2233`
	`^C`
	`char ' ' : 0 times`
	`char '!' : 1 times`
	`…`
	`char '~' : 0 times`

Background Info and Tips

- `/dev/urandom` is a pseudo file that returns random bytes. Therefore, it has no size; you can `read()` from it repeatedly and keep getting random bytes forever.
- Read about the `inet_aton()` function for converting a string containing IP numbers-and-dots notation to binary format.
- Read about the `getaddrinfo()` function for mapping a host name to an IP address in binary format.
- Read the manual pages of `ip(7)`, `connect(2)`, `listen(2)`, `accept(2)`, and `bind(2)`.
- The IP address 127.0.0.1 specifies the local host (it is called a *loopback*) address. If you don't know the IP address of your machine, or are working on a VM without an Internet connection, you can still connect to 127.0.0.1.
- For testing/debugging, consider using a regular file with predefined content instead of `/dev/urandom`.
- You can use the NetCat utility (**`nc`**) to simulate a server or client.
- You can use the `ngrep` utility to monitor the traffic on specific port. Pay attention that root permissions are required to run it, so use sudo.
  Example:

Console 1

```
$ nc -l 2233  # Server listens on
              # port 2233
Hello, world # Server gets the
              # string and prints
              # it.
```

Console 2

```
$ nc 127.0.0.1 2233 # Client connects
Hello, world        # to port 2233, and
                    # sends the string
```

Console 3

```
$ sudo ngrep -d any port 2233              # Start intercepting IP traffic
[sudo] password for eug:                   # on any device, port 2233
interface: any
filter: (ip or ip6) and ( port 2233 )
####                                       # Server accepted connection
```

```
T 127.0.0.1:41104 -> 127.0.0.1:2233 [AP]    # Transfer started
  Hello, world.                             # Dumping the data
####                                        # Connection closed
```

Submission

Submit a ZIP archive named ex5_012345678.zip, where 012345678 is your ID. The archive should contain at least 2 files, `pcc_client.c`, and `pcc_server.c`. *Mac users! Don't submit hidden folders!*

Your code must compile without warning with the following compilation commands:

```
gcc -std=c99 -O3 -Wall -o pcc_server pcc_server.c -pthread
gcc -std=c99 -O3 -Wall -o pcc_client pcc_client.c
```

Or:

```
gcc -std=gnu99 -O3 -Wall -o pcc_server pcc_server.c -pthread
gcc -std=gn99 -O3 -Wall -o pcc_client pcc_client.c
```