CSCI 463 Assignment 7 – IPC

10 Points

Abstract

In this assignment, you will implement *client* and *server* applications that use a TCP stream socket to communicate with each other.

1 Problem Description

Create client and server applications that use a TCP socket to execute a transaction by sending data from the client to the server for processing and return a result that the client will display.

2 Files You Must Write

You will write two C++ programs suitable for execution on `hopper.cs.niu.edu` (and/or `turing.cs.niu.edu`.)

Your source files *MUST* be named exactly as shown below or they will fail to compile and you will receive zero points for this assignment.

Create a directory named `a7` and place within it the following files:

- `client.cpp` The client application is implemented in this file.
- `server.cpp` The server application is implemented in this file.

2.1 client.cpp

Implement a transactional TCP client program:

- Usage: `client [-s server-ip] server-port`
  - `server-ip`: Specify the server's IPv4 number in dotted-quad format. (By default, use 127.0.0.1) Use `inet_pton()` to parse the IPv4 address as part to build the `sockaddr_in` of the server to which to connect.
  - `server-port`: The server port number to which the client must connect. *Don't forget to use* `htons()` *on the integer port number from the command line!*
- Read (possibly binary) input from `stdin` and send it to the server by copying it to the socket to send it to the server. *Since the input is binary you must* ***not*** *treat it like a null-terminated C string!*
- After sending the input data, use the `shutdown()` system library call to let the server know that the request phase of the transaction has completed.
- Read any response data from the server and copy it to `stdout` until the server closes the socket. Make no assumptions as to the nature of the response data from the server and simply copy the raw bytes to `stdout`.

Note: Unless there is an error, the *only* output that will be written to `stdout` by the client application will be the response message bytes from the server.

If any command-line arguments are invalid then print appropriate error and/or Usage messages and terminate the program in the traditional manner. (See https://en.wikipedia.org/wiki/Usage_message.)

~/NIU/courses/463/2022-fa/assignments/a7/handout.tex
jwinans@niu.edu 2022-11-19 09:25:12 -0600 v2.1-106-ga3341c4c

Input

During the request phase of the transaction the client application will read binary input data from `stdin` and send it to the server using a TCP socket connection until it encounters EOF on `stdin`.

The client must make no assumptions about the input data, its format, or where it will come from (keyboard, redirected from a file,...)

During the response-phase of the transaction the client application will read data bytes from the server using the TCP socket connection.

Output

The client application will write its transaction response data to `stdout`.

Any error messages must be written/printed to `stderr` by using `std::cerr` or by calling the standard library `perror()` function.

Do not use `printf()` or `fprintf()` in this application.

Your program will be tested with a combination of the command-line arguments and will be `diff`'d against the output from a reference implementation.

Here are some test-cases run with the reference implementation (assuming that the server application has already been started and is listening on port 9965) submitting the test files that you already have from prior assignments:

```
winans@hopper:~/a7$ ./client 9965 < ~/a5/sieve.bin | hexdump -C
00000000  53 75 6d 3a 20 36 32 37  32 38 20 4c 65 6e 3a 20  |Sum: 62728 Len: |
00000010  32 31 30 31 39 32 0a                              |210192.|
00000017
winans@hopper:~/a7$ ./client 9965 < ~/a5/torture5.bin | hexdump -C
00000000  53 75 6d 3a 20 32 31 34  34 32 20 4c 65 6e 3a 20  |Sum: 21442 Len: |
00000010  31 32 37 36 0a                                    |1276.|
00000015
winans@hopper:~/a7$ ./client 9965 < ~/a3/testsx.in | hexdump -C
00000000  53 75 6d 3a 20 33 32 36  34 30 20 4c 65 6e 3a 20  |Sum: 32640 Len: |
00000010  32 35 36 0a                                       |256.|
00000014
winans@hopper:~/a7$ ./client -s 127.9.9.9 200 < ~/a5/torture5.bin | hexdump -C
connecting stream socket: Connection refused
winans@hopper:~/a7$ ./client 5 < ~/a5/torture5.bin | hexdump -C
connecting stream socket: Connection refused
```

2.2 server.cpp

Implement a transactional TCP server program:

- Usage: `server [-l listener-port]`
  - `listener-port`: The port number to which the server must listen. By default, the port number should be zero (wildcard/ephemeral.) *Don't forget to use* `htons()` *on the integer port number from the command line when building the* `sockaddr_in` *used by* `bind`*!*
- Enter an infinite loop that will:

- `accept()` a connection from a client.
- Print an "Accepted connection" message displaying the IPv4 address and port of the peer client socket. Use `inet_ntop()` to format the client's IPv4 address from the `sockaddr_in` that is filled in by `accept()`.

  Don't forget to use `ntohs()` *on the port number before printing it out!*
- Read all the bytes sent from the client while adding them to a `uint16_t` variable to accumulate the checksum of each byte that is received.
- When an EOF is encountered from the client socket, write the response message formatted like this: `Sum: 123 Len: 456\n`. Note that the message from the server includes the carriage return (shown as `\n` above) that the client will ultimately write to its `stdout`.

The server application must be serially reusable such that after a connection has been closed, regardless of reason, the server will accept another TCP connection for another transaction.

Unless there is an error during initialization, the only way the server should terminate is if a user presses `^C` to kill the process from the command-line terminal from which it was started.

Input

The server application will read its input from the socket.

The individual bytes in the input byte-stream will be counted and summed. The byte count must be stored in a `uint32_t` variable and the sum into a `uint16_t`. The individual bytes that are received by the server must each be saved/treated as `uint8_t` in order for the compiler to properly zero-extend them while summing them. This is necessary so that any overflows involved with the summing and/or counting will match that of the reference implementation!

Observe that if one were to create a file containing exactly 256 bytes containing the values from `0x00` through `0xff` then the unsigned decimal sum would be `128*255 = 32640`. Consider creating such a test file (or use the `testsx.in` file from Assignment 3) as a development/debugging test case.

Output

The server application will display the IP number and port that it is listening on on `stdout`.

The response message from the server must not include a trailing null-terminating character. (See the `hexdump -C` client sample output above.)

Any error messages must be written/printed to `stderr` by using `std::cerr` or by calling the standard library `perror()` function.

Do not use `printf()` or `fprintf()` in this application.

Your program will be tested with a combination of the command-line arguments and will be `diff`'d against the output from a reference implementation.

Here are some example test-run cases:

```
winans@hopper:~$ ./server -l 9965
Socket has port #9965
Accepted connection from '127.0.0.1', port 38998
Ending connection
Accepted connection from '127.0.0.1', port 39002
Ending connection
Accepted connection from '127.0.0.1', port 39004
```

```
Ending connection
^C
winans@hopper:~$ ./server -l 6
binding stream socket: Permission denied
winans@hopper:~$
```

Note: Since the client uses an ephemeral port, the port numbers printed above that the client uses will vary.

3 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your a7 directory is/are the source files defined and discussed above. Then, in the parent of your a7 directory, use the mailprog.463 command to send the contents of the files in your a7 project directory in the same manner as we have used in the past.

4 Grading

The grade you receive on this programming assignment will be scored according to the syllabus and its ability to compile and execute on the Computer Science Department's computer.

It is your responsibility to test your program thoroughly.

When we grade your assignment, we will compile it on `hopper.cs.niu.edu` using these exact commands:

```
g++ -g -Wall -Werror -std=c++14  client.cpp -o client
g++ -g -Wall -Werror -std=c++14  server.cpp -o server
```

To receive full credit the `hexdump -C` output from your client app showing the length and checksum values must *exactly* match that of the reference implementation. The examples above were executed using some of the binary files from assignment 5.

5 Hints

- Watch the course lectures on IPC using TCP.
- You can read raw bytes into a character array buffer from `stdin` in a manner very much like you read from a socket's file-descriptor, like this:

  ```
  char buf[2048];
  ssize_t len = read(fileno(stdin), buf, sizeof(buf));
  ```

- You can write raw bytes to `stdout` from a character array buffer in a manner very much like you write to a socket's file-descriptor, like this:

  ```
  ssize_t len = write(fileno(stdout), buf, buf_len);
  ```

  Keep in mind that the number of bytes written may be less than the number requested in the `buf_len` *argument above.*

- Implement and use a `safe_write()` function, as discussed in lecture, in both your client and server applications.

- Don't forget to ignore broken pipes in your server so that a rogue client won't crash it. (See lecture for details.)
- Note that one can format a printable string using a `std::ostringstream` (as seen in Assignments 4 and 5) and then extract a C string from it like this:

```
std::ostringstream os;
os << ...
std::string str = os.str();
const char *ch = str.c_str();
```

- Note that the method used to calculate a checksum for this assignment is not typical. If you search the internet for calculating checksums, it is *very unlikely* that you will find one that you can cut & paste into this this assignment. Implement your logic precisely as described in the server input section above.
- Only one application can listen on any given port at one time. Plus, the operating system may require that a period of time to pass after an application closes a port before it can use it again!

  Therefore, you may encounter a situation where your server cannot bind to a particular address. Should this happen, try using another port.
- Port numbers that are less than 1024 are reserved. You can not expect to bind to them. However, they would make good test cases to verify that your error handing for bind failures is working properly.

  For example:

```
winans@hopper:~$ ./server -l 5
binding stream socket: Permission denied
```

- You can see what ports are currently in use with the `netstat` command. The numbers following the IPv4 numbers in the column labeled "Local Address" are the port numbers. For example, the first line shows that something is listening for connections on port 9102. The second line shows something listening on port 111... (At the time this was written, the process listening on port 9965 is the server reference implementation.)

```
winans@hopper:~$ netstat -ant4
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.158.56.24:9102       0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44881           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:9965            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:661             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:25              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:2049            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:900             0.0.0.0:*               LISTEN
tcp        0      0 192.168.8.11:53556      192.168.8.1:636         ESTABLISHED
tcp        0      0 192.168.8.11:816        192.168.8.10:2049       ESTABLISHED
tcp        0      0 192.168.8.11:2049       192.168.8.10:790        ESTABLISHED
...
```

  Keep in mind that willful and malicious interference with other applications running on University servers may be considered grounds for academic disciplinary action. Play nice.