CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 4 (document version 1.0)
Network Programming using C

Overview

- This homework is due by 11:59:59 PM on Thursday, April 26, 2018.
- This homework will count as 8% of your final course grade.
- This homework is to be completed **individually**. Do not share your code with anyone else.
- You **must** use C for this homework assignment, and your code **must** successfully compile via `gcc` with absolutely no warning messages when the `-Wall` (i.e., warn all) compiler option is used. We will also use `-Werror`, which will treat all warnings as critical errors.
- Your code **must** successfully compile and run on Submitty, which uses Ubuntu v16.04.3 LTS. Note that the `gcc` compiler is version 5.4.0 (`Ubuntu 5.4.0-6ubuntu1~16.04.5`).
- If you decide to use a multi-threaded approach, to compile your code, use `-pthread` to include the Pthread library.

Homework Specifications

In this fourth and final homework assignment, you will use C to write server code to implement a chat server using sockets. Clients will be able to send and receive short text messages from one another. Further, clients will also be able to send and receive files, including both text and binary files (e.g., images). Note that all communication between clients must be sent via your server.

Clients communicate with your server via TCP or UDP. For TCP, clients connect via a specific TCP port number (i.e., the listener port); this TCP port number is the first command-line argument to your server. For UDP, clients send datagram(s) to a specific UDP port number; this UDP port number is the second command-line argument to your server (and could be the same port number as the TCP listener).

Your server must **not** be a single-threaded iterative server. Instead, your server must use either multiple threads or multiple child processes to handle TCP connections. Your choice! Further, to support both TCP and UDP at the same time, you must use the `select()` system call to poll for incoming TCP connections and UDP datagrams.

As with previous assignments, your server must be parallelized to the extent possible. As such, be sure you handle all potential synchronization issues.

Note that your server must support clients implemented in any language (e.g., Java, C, Python, Fortran, etc.); therefore, only handle streams of bytes as opposed to language-specific structures.

And though you will only submit your server code for this assignment, plan to create one or more test clients. Test clients will **not** be provided, but feel free to share test clients with others via Piazza. Also note that you should use `netcat` to test your server; do not use `telnet`.

Supporting TCP and UDP

To provide flexibility to clients, clients can either establish a connection via TCP or simply send/receive datagrams via UDP. Your server must support at least 32 concurrently connected clients (i.e., TCP connections, with each connection corresponding to a child thread or a child process). Overall, your server must support at least 64 active users at any given time.

For TCP, use a dedicated child process or child thread to handle each TCP connection. In other words, after the `accept()` call, immediately create a child process or child thread to handle that connection, thereby enabling the parent process or thread to loop back around and call `select()` again.

Since UDP is connectionless, for UDP, use an iterative approach (i.e., handle incoming UDP datagrams in the parent process or main thread, then loop back around to the `select()` system call).

Application-Layer Protocol

The application-layer protocol between client and server is a line-based protocol. Streams of bytes (i.e., characters) are transmitted between clients and your server. Note that all commands are specified in upper-case and end with a newline (`'\n'`) character. In general, when the server receives a request, it responds with either a three-byte "`OK\n`" response or an error. When an error occurs, the server must respond with:

```
ERROR <error-message>\n
```

For any error message not specified below, use a short human-readable description matching the simple format shown above. Expect clients to display these error messages directly to users.

LOGIN

Regardless of whether a client uses TCP or UDP, a user must first log in (though authentication is not required). A user identifies itself as follows:

```
LOGIN <userid>
```

Note that a valid `<userid>` is a string of alphanumeric characters with a length in the range `[3,20]`. Upon receiving a `LOGIN` request, if successful, the server responds by sending the three-byte "`OK\n`" string. If instead the given `<userid>` is already connected via TCP, the server responds with an `<error-message>` of "`Already connected`" (only for TCP). Otherwise, if `<userid>` is invalid, the server responds with an `<error-message>` of "`Invalid userid`" (for TCP and UDP).

WHO

A user may send a `WHO` request to obtain a list of all users currently active within the chat server. When your server receives this request, in addition to sending "`OK\n`" to the client, the response should consist of an ASCII-based sorted list of all users, with users delimited by newline (`'\n'`) characters.

As an example, the server may respond with the following:

```
OK\nMorty\nRick\nShirley\nemacs\vi\n
```

LOGOUT

A user may send a `LOGOUT` request to ensure the server marks the user as being completely logged out and inactive. Note that this is recommended but not required for TCP, since the client can simply close its connection to indicate it is logging out.

For UDP, if a `LOGOUT` is not sent, the given user is assumed to still be logged in until a new `LOGIN` request is received for that given user.

When a `LOGOUT` command is sent, the server is required to send an "`OK\n`" response.

SEND

A user may attempt to send a private message to another user via the `SEND` command. The required format of the `SEND` command is as follows:

```
SEND <recipient-userid> <msglen> <message>
```

To be a valid `SEND` request, the `<recipient-userid>` must be a currently active user and the `<msglen>` (i.e., length of the `<message>` portion) must be an integer in the range `[1,994]`.

Note that the `<message>` can contain any bytes whatsoever. You may assume that the number of bytes will always match the given `<msglen>` value.

If the request is valid, the server responds by sending an "`OK\n`" response. Further, the server attempts to send the message to the `<recipient-userid>` by sending either a datagram (UDP) or packet (TCP) using the following format:

```
FROM <sender-userid> <msglen> <message>
```

If the request is invalid, send the appropriate error message from among the following:

- "`Unknown userid`"
- "`Invalid msglen`"
- "`Invalid SEND format`"

BROADCAST

If a user wishes to send a message to *all* active users, the `BROADCAST` command can be used. The format of this command is as follows:

```
BROADCAST <msglen> <message>
```

The `<msglen>` and `<message>` parameters match that of the `SEND` command above.

SHARE

If a user wishes to share a file with another user, the `SHARE` command is sent by the client by first sending the command request as follows:

```
SHARE <recipient-userid> <filelen>
```

After receiving the "`OK\n`" response, the client sends the file (of length `<filelen>` byte) in 1024-byte chunks (i.e., `send()` or `sendto()` calls using a 1024-byte buffer). Each chunk must be acknowledged by the server with a three-byte "`OK\n`" response. And only the last chunk sent can be less than 1024 bytes.

The server subsequently sends the `SHARE` command to the `<recipient-userid>` in the same manner, though the recipient client does not send acknowledgement "`OK\n`" messages back to the server. Further, as with the `SEND` command, the format of the `SHARE` command sent to the `<recipient-userid>` is as follows:

```
SHARE <sender-userid> <filelen>
```

Note that the `SHARE` command is only available if both users (i.e., sender and recipient) are connected via TCP. If this is not the case, send the appropriate error message from among the following:

- "`SHARE not supported over UDP`"
- "`SHARE not supported because recipient is using UDP`"

Text versus Binary Files

All regular files must be supported, meaning that both text and binary (e.g., image) files must be supported. To achieve this, be sure you do **not** assume that files consist of strings; in other words, do **not** use string functions that rely on the `'\0'` character. Instead, rely on specific byte counts.

As noted above, you can assume that the correct number of bytes will be sent and received by client and server. In practice, this is not a safe assumption, but it should greatly simplify your implementation.

Required Output

Your server is required to output one or more lines describing each request that it receives. Required output is illustrated in the example below.

Since you are required to use either child processes or child threads, the child IDs shown in the examples are either thread IDs or process IDs. And as per usual, output lines may be interleaved as multiple clients interact with the server simultaneously.

```
bash$ ./a.out 9876 9889
MAIN: Started server
MAIN: Listening for TCP connections on port: 9876
MAIN: Listening for UDP datagrams on port: 9889
...
MAIN: Rcvd incoming UDP datagram from: <client-IP-address>
MAIN: Rcvd LOGIN request for userid Rick
...
MAIN: Rcvd incoming TCP connection from: <client-IP-address>
CHILD 13455: Rcvd LOGIN request for userid Morty
CHILD 13455: Rcvd WHO request
CHILD 13455: Rcvd SEND request to userid Rick
CHILD 13455: Rcvd SEND request to userid Summer
CHILD 13455: Sent ERROR (Unknown userid)
CHILD 13455: Rcvd WHO request
CHILD 13455: Rcvd SHARE request
CHILD 13455: Rcvd LOGOUT request
CHILD 13455: Client disconnected
...
MAIN: Rcvd incoming TCP connection from: <client-IP-address>
CHILD 19232: Rcvd LOGIN request for userid Rick
CHILD 19232: Rcvd WHO request
CHILD 19232: Rcvd SEND request to userid Rick
CHILD 19232: Rcvd SEND request to userid Rick
CHILD 19232: Rcvd SEND request to userid Morty
MAIN: Rcvd incoming UDP datagram from: <client-IP-address>
MAIN: Rcvd LOGIN request for userid Rick
MAIN: Sent ERROR (Already connected)
CHILD 19232: Rcvd SEND request to userid Morty
CHILD 19232: Rcvd BROADCAST request
CHILD 19232: Client disconnected
...
```

Note that the required output above certainly differs from the specific data sent and received via the application-layer protocol. On Submitty, test clients will connect to your server and test whether you have correctly implemented all aspects of the application-layer protocol.

Handling System Call Errors

In general, if a system call fails, use `perror()` to display the appropriate error message on `stderr`, then exit the program and return `EXIT_FAILURE`. If a system or library call does not set the global `errno`, use `fprintf()` instead of `perror()` to write an error message to `stderr`. See the various examples on the course website and corresponding `man` pages.

Error messages must be one line only and use one of the appropriate formats shown below:

```
MAIN: ERROR <error-text-here>
```

Or:

```
CHILD 17552: ERROR <error-text-here>
```

Submission Instructions

To submit your assignment (and also perform final testing of your code), please use Submitty, the homework submission server. The specific URL is on the course website. As described above, please only submit server code. Do not submit any client code.

Note that this assignment will be available on Submitty a few days before the due date. Please do not ask on Piazza when Submitty will be available, as you should perform adequate testing on your own Ubuntu platform.

That said, to make sure that your program does execute properly everywhere, including Submitty, use the techniques below.

First, as discussed in class (on 1/18), output to standard output (`stdout`) is buffered. To ensure buffered output is properly flushed to a file for grading on Submitty, use `fflush()` after every set of `printf()` statements, as follows:

```
printf( ... );     /* print something out to stdout */
fflush( stdout );  /* make sure that the output is sent to a */
                   /*  redirected output file, if specified  */
```

Second, also discussed in class (on 1/18), use the `DEBUG_MODE` technique to make sure you do not submit any debugging code. Here is an example:

```
#ifdef DEBUG_MODE
    printf( "the value of x is %d\n", x );
    printf( "the value of q is %d\n", q );
    printf( "why is my program crashing here?!" );
    fflush( stdout );
#endif
```

And to compile this code in "debug" mode, use the `-D` flag as follows:

```
bash$ gcc -Wall -Werror -D DEBUG_MODE homework4.c -pthread
```