CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Homework 4 (document version 1.1)
Network Programming and Multi-Threaded Programming using C

Overview

- This homework is due by 11:59:59 PM on Friday, April 26, 2019.
- 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 v18.04.1 LTS. Note that the `gcc` compiler is version 7.3.0 (`Ubuntu 7.3.0-27ubuntu1~18.04`).
- Further, you must use the Pthread library. Remember, to compile your code, use `-pthread` to link the Pthread library.

Homework Specifications

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

Clients communicate with your server via TCP or UDP via the port number specified as the first command-line argument. For TCP, clients connect to this port number (i.e., the TCP listener port). For UDP, clients send datagram(s) to this port number.

To support both TCP and UDP at the same time, you must use the `select()` system call in the main thread to poll for incoming TCP connection requests and UDP datagrams.

Your server must **not** be a single-threaded iterative server. Instead, your server must allocate a child thread for each TCP connection. Further, 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, MIPS, etc.); therefore, only handle streams of bytes as opposed to any 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`.

Simultaneously supporting TCP and UDP

To provide flexibility to clients, clients can either establish a connection via TCP or simply send/receive datagrams via UDP. As noted above, to support both TCP and UDP at the same time, you must use the `select()` system call in the main thread to poll for incoming TCP connection requests and UDP datagrams.

Your server must support at least 32 concurrently connected clients (i.e., TCP connections, with each connection corresponding to a child thread). ~~And in general, your server must support at least 64 active users at any given time.~~

For TCP, use a dedicated child thread to handle each TCP connection. In other words, after the `accept()` call, immediately create a child thread to handle that connection, thereby enabling the main thread to loop back around and call `select()` again. **(v1.1)** For TCP, you are only guaranteed to receive data up to and including the first newline character in the first packet received; therefore, expect to implement multiple `read()` calls.

Since UDP is connectionless, for UDP, use an iterative approach (i.e., handle incoming UDP datagrams in the main thread, then immediately loop back 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, as shown below.

In general, when the server receives a request, it responds with either a four-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 one-line format shown above. Expect clients to display these error messages to users.

The following subsections define the application-layer protocol commands.

LOGIN

If a client uses TCP, the user must first log in (though authentication is not required). A user identifies itself as follows:

```
LOGIN <userid>\n
```

Note that a valid `<userid>` is a string of alphanumeric characters with a length in the range `[4,16]`. Upon receiving a `LOGIN` request, if successful, the server sends the four-byte "`OK!\n`" string.

If the given `<userid>` is already connected via TCP, the server responds with an `<error-message>` of "`Already connected`"; if `<userid>` is invalid, the server responds with an `<error-message>` of "`Invalid userid`" (and in both cases keeps the TCP connection open).

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\nSummer\nmeme\nqwerty\n
```

Note that the `WHO` command must be supported for both TCP and UDP.

LOGOUT

A user connected via TCP 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.

When a `LOGOUT` command is sent, the server is required to send an "`OK!\n`" response. And the connection should stay open (until the remote side closes the connection).

SEND

A user connected via TCP 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>\n<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 `[1,990]`.

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 a TCP packet using the following format:

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

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

- "`Unknown userid`"
- "`Invalid msglen`"
- "`Invalid SEND format`"
- **(v1.1)** "`SEND not supported over UDP`"

BROADCAST

If a user wishes to send a message to *all* active users, the `BROADCAST` command can be used. This command can also be used via UDP, i.e., without the need to first log in. The format of this command is as follows:

```
BROADCAST <msglen>\n<message>
```

The `<msglen>` and `<message>` parameters match that of the `SEND` command above. **(v1.1)** For UDP, use a hard-coded "`UDP-client`" string for the `<sender-userid>` that all UDP clients use.

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>\n
```

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 four-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>\n
```

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 a "`SHARE not supported over UDP`" error message.

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. As per usual, output lines may be interleaved as multiple clients interact with the server simultaneously.

```
bash$ ./a.out 9876
MAIN: Started server
MAIN: Listening for TCP connections on port: 9876
MAIN: Listening for UDP datagrams on port: 9876
...
MAIN: Rcvd incoming TCP connection from: <client-IP-address1>
CHILD <tid1>: Rcvd LOGIN request for userid Rick
...
MAIN: Rcvd incoming TCP connection from: <client-IP-address2>
CHILD <tid2>: Rcvd LOGIN request for userid Morty
CHILD <tid2>: Rcvd WHO request
CHILD <tid2>: Rcvd SEND request to userid Rick
CHILD <tid2>: Rcvd SEND request to userid Summer
CHILD <tid2>: Sent ERROR (Unknown userid)
CHILD <tid2>: Rcvd WHO request
MAIN: Rcvd incoming UDP datagram from: <client-IP-address>
MAIN: Rcvd BROADCAST request
CHILD <tid2>: Rcvd SHARE request
CHILD <tid1>: Client disconnected
CHILD <tid2>: Rcvd LOGOUT request
CHILD <tid2>: Client disconnected
...
MAIN: Rcvd incoming TCP connection from: <client-IP-address3>
CHILD <tid3>: Rcvd LOGIN request for userid Rick
CHILD <tid3>: Rcvd WHO request
CHILD <tid3>: Rcvd SEND request to userid Rick
CHILD <tid3>: Rcvd SEND request to userid Rick
CHILD <tid3>: Rcvd SEND request to userid Morty
MAIN: Rcvd incoming UDP datagram from: <client-IP-address>
MAIN: Rcvd WHO request
CHILD <tid3>: Rcvd SEND request to userid Morty
CHILD <tid3>: Rcvd BROADCAST request
CHILD <tid3>: Client disconnected
...
```

Note that the required output above is certainly different than 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`.

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

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

Or:

```
CHILD <tid>: 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.

Note that this assignment will be available on Submitty a minimum of three 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 1/10), 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 hw4.c -pthread
```

Second, as discussed in class (on 1/14), output to standard output (`stdout`) is buffered. To disable buffered output for grading on Submitty, use `setvbuf()` as follows:

```
setvbuf( stdout, NULL, _IONBF, 0 );
```

You would not generally do this in practice, as this can substantially slow down your program, but to ensure correctness on Submitty, this is a good technique to use.