- Main
- Course info: Arch&SPlab, Splab, Arch only.
- Announcements
- Assignments
- Class material
- Practical sessions
- Lab sessions
- Lab Completions
- Labs Schedule
- FAQ
- Exams
- Previous exams
- Grades
- Useful links
- Online Quiz
- Forum
- Labs Forum
- Submission system

- recent changes
- logout

zoharith logged in
student mode

• Lab9 » Tasks printable version

Lab 9 - TCP client/server

There are no completion tasks in this lab. All tasks must be completed during the lab session

Changes to the lab
In order to ease the burden of the last week of the semester we made the following changes of the lab.

1. The lab can be done in pairs. However, both partners are required to work on the entire lab and expected to be familiar with all its aspects.
2. Task 0 will be worth 50 points (out of 100), and it will be possible to finish it at lab time. Our suggestion is to complete task0 at home so you'll have enough time to complete the rest of the lab during the lab session. Do remember, there is no completion lab.
3. Task 2 is now a bonus task (0-20 points).

Introduction

In this lab you will implement an simple client and server for a toy FTP-like protocol. Your system will support connecting, listing directories and downloading files.

General requirements

The following are mandatory requirements for all the tasks in this lab.

1. The client will be implemented in a source file called client.c and the server in server.c
2. The client and the server share some functionality (e.g. debug mode) and structures (e.g. client_state). You should implement these and any other shared parts in common.c (and the header common.h).
3. You mustn't use magic numbers. Numbers with special meaning (like the 2018 port number or false and true) should be named using macros.
4. The client must check the return value of all socket APIs for errors (errors are very common in networking code). On error the client will print the error (using perror) and **exit with code -1**.
5. The server must check the return value of all socket APIs for errors (errors are very common in networking code). On error the server will print the error (using perror), **free any resources allocated for the client, and return to listening for new connections**.

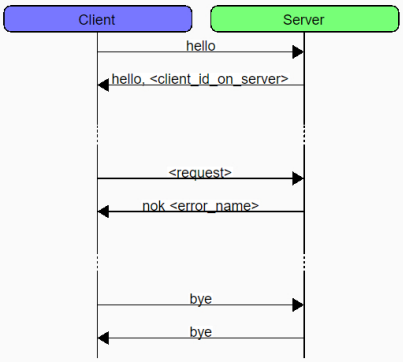
Your client and server are decoupled. Your client is expected to work with every server that implements our protocol, and same goes for your server. Your work might be tested against different software.

Task 0 - Base client and server

Our base client and server will only support connecting and disconnecting using the messages "hello" and "bye".

Protocol

This is the (very basic) protocol you'll be implementing in this task:



Task 0a - client

In this task you will write the base code for the client. The client runs an infinite prompt-loop (like your shell from labs 5/6):

1. Display a prompt - "server:%s>" where %s is the address of your server (see below) or the string "nil" if not connected to any server
2. Read a line from stdin (assume it's no longer than 2kB). It is advisable to use fgets.
3. Parse the input. You can use the line_parser from labs 5/6 (it's an overkill, but you're familiar with it. Don't forget free_cmd_lines).
4. Write a function `int exec(char* cmd, char** args, int args_len)` that takes a command and possible arguments. This function executes the various commands the client will support (initially just "conn" and "bye"). You may change the signature of this function as you wish (e.g. if you're using the line parser).
5. End the infinite loop of the shell if the command "quit" is entered in the shell, and exit with `return 0`.

Also, the client will maintain a global state using the structure:

```
typedef enum {
        IDLE,
        CONNECTING,
        CONNECTED,
        DOWNLOADING,
} c_state;

typedef struct {
        char* server_addr;      // Address of the server as given in the [connect] command. "nil" if not connected to any server
        c_state conn_state;     // Current state of the client. Initially set to IDLE
        char* client_id;        // Client identification given by the server. NULL if not connected to a server.
        int sock_fd;            // The file descriptor used in send/recv
```

```
} client_state;
```

Initially, the client's state is `IDLE`, the server address is "nil", the client's ID is NULL and the socket file descriptor is -1.

conn

At the moment, the client supports only two commands `conn <srv_addr>` and `bye` (disconnect). `connect` will cause the client to connect to a server at <serv_addr> over TCP port 2018 and send the string "hello". The client will then set its `conn_state` to `CONNECTING` and wait for a response. The server should respond with "hello <client_id>" where <client_id> is some string the server uses to identify the client (see below). These steps are called a handshake.
Once the server responds, if the response matches the expected response, the client will update the global state:

1. Set `client_id` to <client_id>.
2. Set conn_state to CONNECTED.
3. Set sock_fd to the connection's socket.
4. Set server_addr to <serv_addr>.

To test your work, you can run this script (based on the NetCat Linux tool) that simulates a server (without any of the logic).

bye

The `bye` command, simply disconnects from a server by sending the message "bye" to it (to let it know the client is disconnecting), and then closing the connection. After sending the "bye" message, the global state is updated:

1. Set `client_id` to NULL.
2. Set conn_state to IDLE.
3. Set sock_fd to -1.
4. Set server_addr to "nil".

This command does **not** quit the client (the `quit` special command does that), the client may now connect to a new server.

Requirements

1. `connect` may only run in `IDLE` state. If it's called in any other state, the function should return immediately with value -2;
2. `bye` may only run in `CONNECTED` state. If it's called in any other state, the function should return immediately with value -2;
3. On any other error, `connect` and `bye` should return -1. On success they should return 0.
4. If the server, for some reason, can't handle the request, it'll respond with "nok <error_name>". At which point the client will print "Server Error: %s" where %s is <error_name> to stderr and reset its state to its initial values.
5. The client should accept an optional command line argument "-d" (debug). In debug mode (use a global flag), the client prints to stderr any message received from the server in the format: "%s|Log: %s", where the first string is the server's address and the second is the message. You should create a global debug flag, and a function that takes a string, and prints it only if that flag is set to TRUE. Do not litter your code with checks of the debug flag. Use a function or a macro that handles that.
6. If the server returned the wrong response (something other than "hello "), the client should print a relevant error to the screen and return to the main promopt-loop.
7. The client does not need to wait for the server's response to the "bye" message.

Task 0b - server

Now you will write the server. The server will listen to incoming connections on TCP port 2018. Once a connection arrives, The server will start handling this client until the client disconnects[1].

Initially the server sets up a `client_state` object for the client:

1. Set `client_id` to NULL.
2. Set conn_state to IDLE.
3. Set sock_fd to -1.
4. Set server_addr to the name of the machine on which the server is running (use gethostname.

Once the client state is set up, the server will enter a client-loop that receives client messages and executes them.

[1]Our server only handles one client at a time (other potential clients are blocked in `connect` until the server gets to them). This is never the case with real world servers, but we use this to simplify the task.

hello

At the moment, the server supports only two messages, "hello" and "bye" (as defined by the protocol). "hello" will initiate a handshake. The server will change the client's state to `CONNECTED` and respond with "hello %s" where %s is the `client_id` assigned to the client. The server will print to stdout "Client %s connected" where %s is `client_id`. The client's ID and connection socket will be stored in the state struct.

bye

When the server receives a "bye" message, it means the client is disconnecting (or has already disconnected). At this point the server will free any resources allocated for this client (including the client state object, if allocated on the heap). Next the server will respond with "bye" and print to stdout "Client %s disconnected" where %s is `client_id` and then the server will will close the client's file descriptor and the client-loop will end.

Requirements

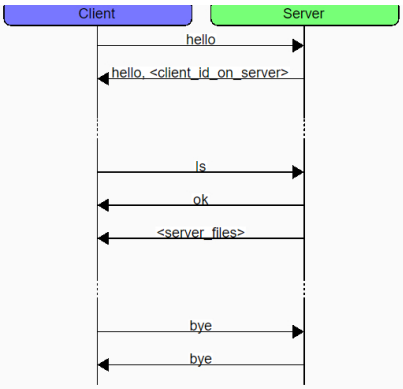
1. "hello" may be received only when the client is in `IDLE` state and "bye" in `CONNECTED` state. If a message is received in an incorrect state, the server will respond with "nok state";
2. Like the client, the server should accept an optional command line argument "-d" (debug). Also like the client, in debug mode server prints to stdout any message received from the client in the format: "%s|Log: %s", where the first string is the server's address and the second is the message. Implement this debug functionality similarly as you did in the client.
3. Any message received from the client that isn't a defined by the protocol is printed to stderr in the form "%s|ERROR: Unknown message %s" where the first %s is the client's ID and the second is the received message.
4. "nok <error>" response triggers the server to disconnect the client (reset the state and release/close the client's resources).

Task 1 - ls

In this task, you'll add the functionality of listing the files on the server's current directory.

protocol

This is the (still basic) protocol you'll be implementing in this task:



Task 1a - Client

You will now add support in the client for a new command, `ls`. `ls` will cause the client to send the message "ls" to the server and wait for a response.
If the server understood the message and can respond to it, it will reply with the message "ok" and then the server will send another message containing the actual list of files. The client will print this list to stdout.

> ⚠ Because the server can, potentially, send two responses in quick succession ("ok" and then the listing itself), the client might receive them in a single `recv` (as if they were sent with a single call to `send`). This can make things difficult for you. A solution to this problem would be to only `recv` 3 bytes initially, compare these to "nok" or "ok " and only if everything is OK `recv` the directory listing (or if everything is not OK, `recv` the error name and act accordingly)

Requirements

1. `ls` may only run in `CONNECTED` state. If it's called in any other state, the function should return immediately with value -2;
2. A "nok <error_name>" response message should be printed to stderr as "Server Error: %s" where %s is <error_name>.
3. If a "nok <error_name>" message is given, the client will disconnect from the server (in the same way as in the `bye` command).
4. Assume the maximal server response is `LS_RESP_SIZE` = 2kB.

Task 1b - Server

Add support in the server's client-loop for the new message, "ls". When receiving the "ls" message" the server will read all the regular files in the current working directory to create string of all the files. Use `list_dir` (from common.c) to create the list of files. If `list_dir` fails, you should stop and return "nok filesystem" to the client.

Once the list of files is ready, the server will send "ok" to the client and then send the list.

If no errors occurred, at the end of the operation the server will print to stdout "Listed files at %s" where %s is the current working directory (you might want to use `getcwd`).

Requirements

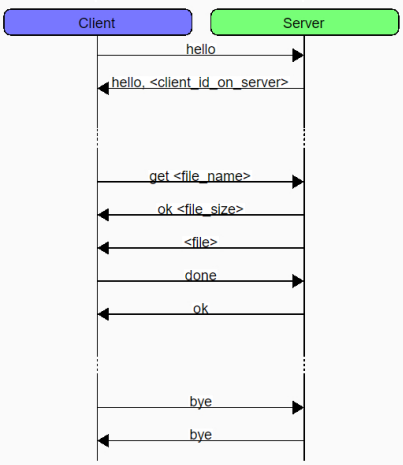
- "ls" may be received only when the client is in `CONNECTED` state. If "ls" is received in another state, the server will respond with "nok state";
- Unknown messages (i.e. not "ls" or "bye") are replied to with "nok message".
- Assume the maximal server response is `LS_RESP_SIZE` = 2kB.
- "nok <error>" response triggers the server to disconnect the client (in the same way as in the "bye" message").

Task 2 - get

In this task, you'll add the functionality of downloading a file from the server.

protocol

This is the (still basic) protocol you'll be implementing in this task:



Task 2a - Client

You will now add support in the client for a new command, `get <file_name>`. `get` will cause the client to send the message "get %s" to the server, where %s is a file name If the server understood the message and can execute the request, it will reply with the message "ok <file_size>".
Once the server responds with "ok <file_size>", the client expects to get another message, this time the file itself. The client will `fopen` a file called <file_name>.tmp for writing and set its state to `DOWNLOADING`. However, the client can't `recv` the entire file in a single call (due to potentially large file sizes). Your client will create a modest buffer of size `BUFF_SIZE`, and create a loop that keeps `recv`ing `BUFF_SIZE` chunks for the file and writing them to the newly opened file.

Once <file_size> bytes are `recv`ed, the client will send "done" to the server and wait for the server to reply with "ok". However, if the server is not yet ready to receive a "done" message, it will not respond (or respond with "nok …" message). The client should wait at most 10 seconds for the "ok" message. Once an "ok"

message is received, the client will move (i.e. rename) <file_name>.tmp to <file_name> and set its state to `CONNECTED`. If the client receives a "nok ..." message or no message at all after 10 seconds, the client will delete <file_name>.tmp and print the error message "Error while downloading file %s" where %s is <file_name>.

Requirements

1. `get` may only run in `CONNECTED` state. If it's called in any other state, the function should return immediately with value -2;
2. BUFF_SIZE = 1kB.
3. A "nok <error_name>" response message should be printed to stderr as "Server Error: %s" where %s is <error_name>.
4. If a "nok <error_name>" message is given, the client will disconnect from the server (in the same way as in the `bye` command).

Task 2b - Server

Now you'll add support in the server's client-loop for the new message, "get <file_name>". When the server receives "get <file_name>" message, it will send a file to the client. First, the server will calculate the size of <file_name> (use `file_size` in common.c), and respond with "ok %d" where %d is the file size. If the server can't calculate the file size, it'll respond with "nok file". Next, if everything went well, The server will set the client state to `DOWNLOADING` and the server will start sending the file itself. To send the file use sendfile.

Next, the server will expect a "done" message. If this message does not arrive, the server will respond with "nok done". If a "done" message indeed arrives, the server will set the client state back to `CONNECTED` and respond with "ok". If no errors occurred, after the server responds with "ok" it will print to the screen "Sent file %s" where %s is the filename requested by the client.

Requirements

- "get" may be received only when the client is in `CONNECTED` state. If "get" is received in another state, the server will respond with "nok state";
- A "nok <error_name>" response message should be printed to stderr as "Server Error: %s" where %s is <error_name>.
- Unknown messages (i.e. not "get", "ls" or "bye") are replied to with "nok message".
- "nok <error>" response triggers the server to disconnect the client (in the same way as in the "bye" message").

Deliverables

There are no completion tasks in this lab. All tasks must be completed during the lab session

You must submit source files for task 1 and task 2 and a makefile that compiles them. This should be your zipped directory structure:

+ task1
  - makefile
  - client.c
  - server.c
  - common.h
  - common.c
+ task2
  - makefile
  - client.c
  - server.c
  - common.h
  - common.c