ProjReport

Project

Or, the group's apology.

In addition to our "official" report (the `README.md` file which explains the code's basic functioning), we decided to include a written report including a more detailed overview of our project, tests, and implementation choices.

As it stands, not all of the project is as finished as we would've wanted it to be.

The Struct

As explained in `README.md`, we decided to represent packets mainly with the `Packet` structure:

```
// PACKET STRUCTURE
typedef struct  {
    char E; // 1 byte
    char D; // 1 byte
    char r; // 1 byte
    uint16_t data_size; // 2 byte  </
    int8_t code; // 1 byte
    char option1[32]; //32 bytes
    char option2[32]; //32 bytes
    char * data_ptr;
} Packet ;
```

The `E`, `D`, and `r` chars are just for verifying the packet string's integrity.

The newly created files

`functions.c`, present in `/usrc/` and having a corresponding `.h` file, includes the functions that are useful both to the `server` and `client` side of operations. This includes:

- safer, faster string concatenation
- integer to string conversion (for the packet's arguments, which are `char *`)
- a string slicing function (for splitting large data in several packets)
- reading/writing to files

`struct_packet.c` describes the main packet structure and some of its base functions.

- freeing its data
- printing its contents elegantly
- transforming them to a string and vice-versa
- creating empty packets, or empty packets with an error code.

`testing.c` tested some of the functions before their server implementation.

Of course, the `Makefile` was adapted to compile these functions and create their corresponding object and executable files.

Parameter Implementation

We worked in implementing all required parameters:

QuotaSize and QuotaNumber

`QUOTASIZE` and `QUOTANUMBER` are defined as integer parameters given by the server, consisting of the maximum size (in bytes) and maximum number of files the server can store, respectively. Set as `-quotasize VAL` and `-quotanumber VAL` during `./server` execution.

`SERVER_DIRECTORY` and `CLIENT_DIRECTORY` are the working directories for the client and the server. These ***must*** end in a `/`, for proper concatenation with filenames.

We use dedicated functions to set these values, as we need to check the parameters given are valid.

Other functions in `student_server.c` check the remote directory's file count and size to make sure the quotas are not exceeded (this is done **before** adding a new file).

```
void set_quota_size(int qs);
void set_quota_number(int qn);

void set_server_directory(const char *string);
void set_client_directory(const char *string);

void force_server_directory_format();
void force_client_directory_format();
```

In `add_remote_file()`:

```
C
if(files_in_folder(directory) + 1 > QUOTANUMBER){

printf("Error: QUOTA NUMBER (%d) WOULD BE EXCEEDED BY TRAN
return error_packet(QUOTA_EXCEEDED);
}

if(folder_size(directory) + in->data_size > QUOTASIZE){
printf("Error: QUOTA SIZE (%d) WOULD BE EXCEEDED BY TRANSF
return error_packet(QUOTA_EXCEEDED);
```

Testing our Functions

In a Linux device, I open three terminals.

- One to open `./server -directory "./" -quotasize 800000 -quotanumber 80`
- One to open `./client <ip.address> -directory "./"`
- One to `make` every time changes are made.

As of writing, the function `put` works as expected, getting the file from the CLIENT_DIRECTORY and sending it to the SERVER_DIRECTORY.

Working Functions

clientsmall is a file present in client side with enough data to fit on one packet. `> put clientsmall`

The client successfully sends the contents of `./clientsmall` over to the remote directory, we can verify this by calling:

```
> cat clientsmall 2
```

Since `clientsmall` is now present in the client directory, we can now print the first two lines in its content:

```
Line 1  -- of clientsmall -- somme useless chars to fill the line
Line 2  -- of clientsmall -- somme useless chars to fill the line

>
```

To rename that file, we can call:

```
> mv clientsmall clientsmall_but_in_server
```

This is the packet sent as a request:

```
Print Packet :
        -Const E : E
        -Const D : D
        -Const r : r
        -No Data Size
        -Code :  3
        -Option 1 : clientsmall
        -Option 2 : clientsmall_but_in_server
        -No Data Pointer Provided
Amount Send : 70
Data to Send : 0
Succesfully sent packet
```

Now, let's try to call `cat clientsmall 2` again...

```
        -Data Pointer Provided ? : 0 (1 <=> True)
ERROR: File ./clientsmall does not exist on directory!
No modifications will be made.
```

Indeed, the file no longer exists. Instead, we call:

```
> cat clientsmall_but_in_server 2
```

```
Line 1  -- of clientsmall -- somme useless chars to fill the line
Line 2  -- of clientsmall -- somme useless chars to fill the line
```

And if we don't want it anymore, we can **remove** it:

`> rm clientsmall_but_in_server`

Calling `cat clientsmall_but_in_server` returns a NOT_FOUND error. It removes the client_side

Then, let's try and fetch a file present in the remote server. This is done via the `get` command.

Non-Working Functions

`> ls` does not work when there is more than three files to print, which happens very often.

`> get` does not work when files are larger than one packet.

`> put` accepts files that are not text-only.

`> cat` *does* work for files requiring more than one packet, but it crashes immediately after printing. The error is `***stack smashing detected ***`

We created functions to track these cases, but they yield segmentation faults. The principle of all of them can be detailed below: