UDPftcs

Simone Zama
simone.zama@studio.unibo.it
0000989981

June 7, 2022

Contents

1 Analysis

1.1 Project Description

The aim of this project is to develop a client-server architecture that provides a file transfer functionality based on the UDP protocol.

1.1.1 Software functionalities

The software will provide the following functionalities:

- The connection between the server and the client without identification
- Listing all files available for download from the server
- The exchange of file from the server to the client and from the client to the server

1.1.2 Server Functionalities

The server will provide the following functionalities:

- A response message to the **ls** command containing a list of all files available for download
- A response message to the **get** command containing the requested file or an error message if the requested file isn't available
- The reception of a file after the **put** command with a response message containing the outcome of the command

1.1.3 Client Functionalities

The client will provide the following functionalities:

- Sending the **ls** command to receive the list of available files
- Sending the **get** command to request a file
- Receiving a file requested with the **get** command or an error message, in which case the client will handle the error
- Sending the **put** command followed by the uploaded file and the reception of the message describing the outcome of the **put** command

The client allows the user to interact with the server, using the server's commands, and gives feedback based on the outcome of the commands.

2 Design

As stated in the Project Description section this software uses a Client-Server structure. Throughout the communication between the client and the server the two sides will exchange different kinds of messages:

- **Commands** from the client to the server
- **Acknowledgements** In both directions, these are used to communicate the outcome of a command
- **Data** the data that composes the files that are being exchanged in both directions

2.1 Client limit

The server will handle one request at a time without any identification, but will handle each request individually, meaning that more than one client can interact with the server as long as they do so one at a time and their requests don't overlap. This allows for a simpler structure and implementation without the use of multiple threads.

2.2 File transfer and data integrity

The exchange of files from the server to the client and vice versa is implemented in similar ways:

- The sender calculates the amount of segments the file will be split into and sends it to the recipient
- The recipient receives the number of segments and waits for the file data
- The sender sends each file data segment with its index which indicates its order in the original file
- The recipient receives the different segments and saves them into an ordered list based on their indexes
- After the sender has sent all segments, it sends the hash of the file that was sent
- The recipient receives the hash and compares it with the hash of the data it has previously received

File data is exchanged with the use of a dictionary:

```
{
    "index": 1,
    "bytes": b"11101110110..."
}
```

Where the index field **index** indicates the position of the packet in the list of packets that form the complete file, and the **bytes** field contains a part of the raw byte data of the file.

During the transmission of file data, the side that is uploading a file will halt execution for a brief moment to make sure that the buffer of the recipient doesn't overflow. The amount of time the sender waits for can be configured, see Configuration section.

Using a hash at the end to check for file integrity allows us to avoid the overhead that would be caused by a checksum, but forces us to transfer the entire file again in case of data corruption or loss.

If the client is downloading a file and the hashes aren't the same the client will ask to download the file again by repeating the **get** command. If the server is downloading a file from the client and the hashes aren't the same it sends a negative acknowledgment to the client, who then handles it by repeating the **put** command. The advantage of this design is that the server doesn't have to handle an error during the transmission of a file, in fact the requests that are repeated after a failed transfer look like normal requests from the server's point of view.

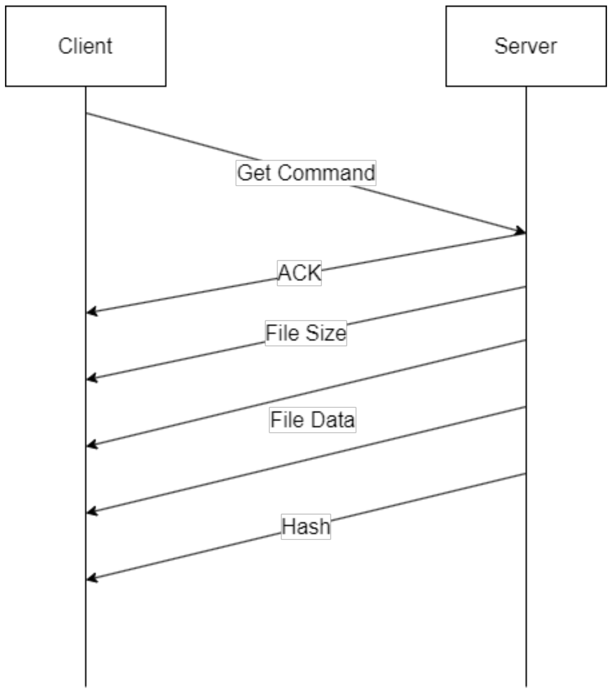


Figure 1: A successful **get** command.

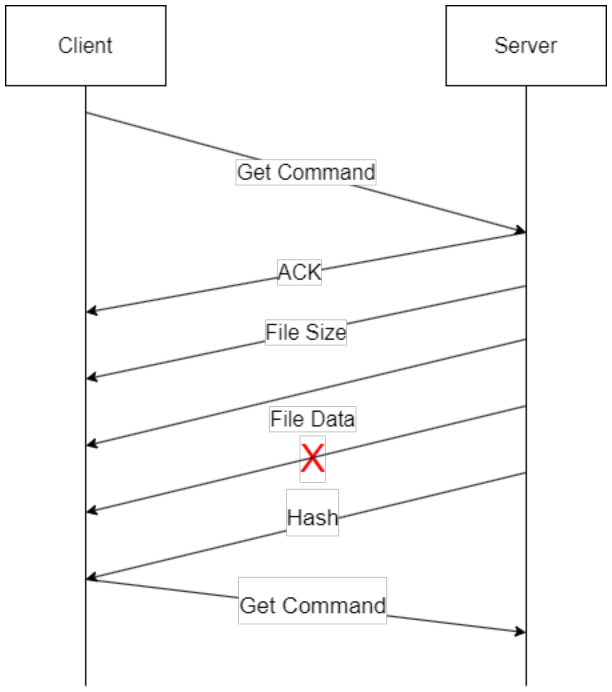


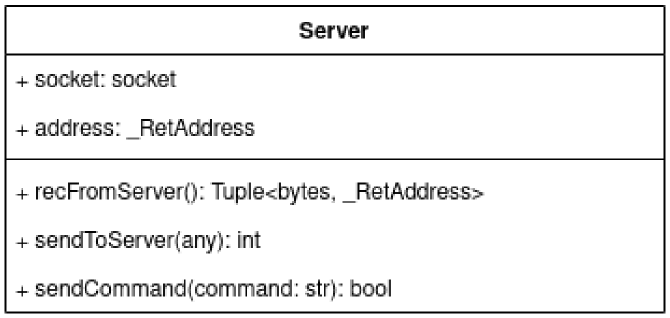
Figure 2: A failed **get** command.

2.3 Client-side Design

The client must allow the user to select which command to execute, which means that the command will be taken from user input, to minimize the amount of command parsing that will be done by the server on the command, the client will send a message to the server at the start of it's execution and the server will respond with the list of all available commands. After this exchange the client parses the command list and when the user inputs a command it will assert that the command is in said list. This allows for an easier implementation of additional functionalities and diminishes the possible errors encountered because of a faulty command sent to the server.

After a command is sent to the server the client waits for a positive acknowledgement from the server and if the command is invalid it communicates it to the user and waits for another input.

After the client receives the initial response from the server it creates a **Server** object, that contains the server's data and is used to communicate with the server from that point onward.



Server
+ socket: socket
+ address: _RetAddress
+ recFromServer(): Tuple<bytes, _RetAddress>
+ sendToServer(any): int
+ sendCommand(command: str): bool

Figure 3: The **Server** object

2.4 Server-side Design

The general handling of a request from the server is as follows:

- At the start the server waits for a request from a client
- After a request is received the server parses the command and checks it's validity
- Sends an acknowledgement based on the previous check
- Exchanges data with the client based on given command
- If the client has requested to upload a file, the server sends a message describing the outcome of the **put** command

When the server receives a request it creates a **ClientConnection** object and uses it to respond to the client's request. The objects contains information about the client that has made the request and is used to make sure no other clients are communicating with the server while it responds to a request.

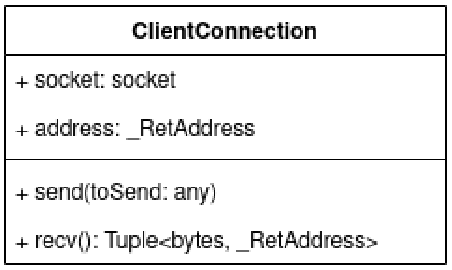


Figure 4: The **ClientConnection** object

2.5 Handling timeouts

Some messages that are being exchanged between the server and the client could be lost resulting in a timeout on the side that is waiting to receive data. The client and the server handle timeouts in different ways:

2.5.1 Client timeouts

When the client encounters a timeout it handles it by sending the request the timeout occurred on to the server again and repeats the same instructions until the request is completed.

2.5.2 Server timeouts

When the server encounters a timeout it handles it by returning to its initial state where it waits to receive a request from a client.

3 User Guide

3.1 Requirements

This software requires python 3.8 or newer to run.

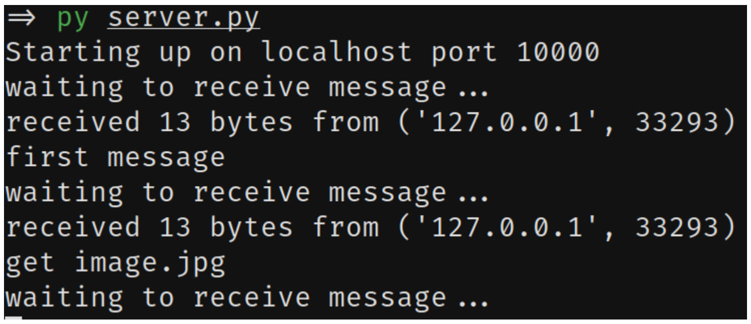
3.2 Running the Server

In order for the software to work the server must be listening for requests, to do so run the following command in the **/server** directory:

```
python3 server.py
```

While the server script is running it will output some information regarding its execution, such as:

- A message that displays on which port the server is running
- A message while the server is waiting to receive a request
- The content of the request that was received and the address of the client that as made the request
- The progress during a file download or upload
- An error message if a file download from the client has failed
- A notification that the server is reverting to its initial state if it receives more than one request at a time



```
⇒ py server.py
Starting up on localhost port 10000
waiting to receive message ...
received 13 bytes from ('127.0.0.1', 33293)
first message
waiting to receive message ...
received 13 bytes from ('127.0.0.1', 33293)
get image.jpg
waiting to receive message ...
```

Figure 5: Output from the server

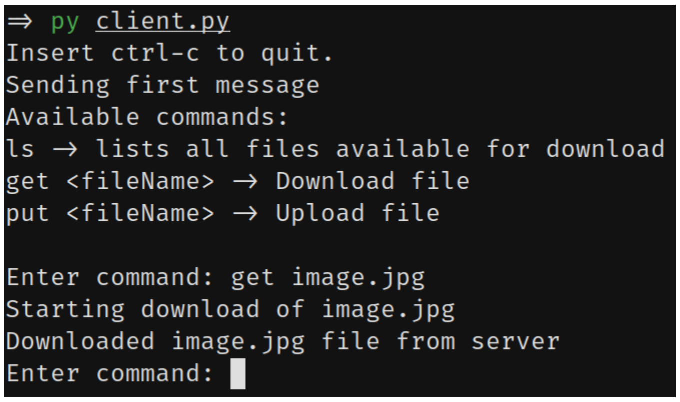
3.3 Running the Client

Once the server is running the client can be started with the following command in the **/client** directory:

```
python3 client.py
```

As explained in the Client-side Design section the client receives the list of valid commands from the server and displays it to the user. If the client doesn't receive a response from the server it updates the user and tries again until the server responds.

While the client is running it displays a command prompt and waits for the user to input a command. After the client script receives a command from the user it displays an error message if the command is invalid otherwise it forwards the request to the server. The client will display the progress during file upload and download, and warns the user of any errors during the handling of requests.



```
⇒ py client.py
Insert ctrl-c to quit.
Sending first message
Available commands:
ls → lists all files available for download
get <fileName> → Download file
put <fileName> → Upload file

Enter command: get image.jpg
Starting download of image.jpg
Downloaded image.jpg file from server
Enter command:
```

Figure 6: Output from the client

3.4 Configuration

The **config.py** modules in the **/client** and **/server** directories contain the settings used in the scripts.

3.4.1 Client configuration

- **PACKSIZE** the length of file data that is transferred in each packet
- **BUFF** the length of the buffer from which the client receives data
- **FILE_DIR** the directory where the files are downloaded to and uploaded from
- **TIMEOUT_TIMER** the amount of time, in seconds, the client waits for a response from the server
- **SERVER_ADDRESS** a Tuple that contains the server's IP address and port

3.4.2 Server configuration

- **PACKSIZE** the lenght of file data that is transferred in each packet
- **BUFF** the length of the buffer from which the client receives data
- **FILE_DIR** the directory where the files are downloaded to and uploaded from
- **COMMANDS** the string that contains all available commands and their description, this shouldn't be changed unless a new command is implemented
- **TIMEOUT_TIMER** the amount of time, in seconds, the server waits for data from a client during the **put** command
- **ADDRESS** a Tuple that contains the address and port the server will use