IERG3310 Computer Networks Fall 2019
Lab 2: Socket Programming - A Remote Backup System
Due: 11:59pm, Monday, 02/12/2019,

1 Objective

The objective of this assignment is to gain experience in UDP/TCP socket programming. You will implement a small remote backup system with a simple protocol to exchange information between server and client. With this backup system, you can use the client to list whatever files that have been stored in the remote server's backup folder. You can also send a local file to be stored in that backup folder on the server. If you do not want a file any more, you have the client send command to remove that files from the server's backup folder.

2 Specification

This protocol includes five command functions, "`ls`", "`send`", "`remove`", "`quit`" and "`shutdown`", as shown in Figure 1, 2, 3 and 4. Figure 5 is an example of how this system works. After the startup of the server and the client, user sends input command from standard input (such as keyboard) to the client interactively, then the commands received at the client will be sent over the network to the server. Then, there will probably be some messages exchanges between the server and the client. After that, the server will execute the corresponding functions and send back the information requested by the client. Specifically, when the client receives the "`ls`" command from the user, it queries the server and lists all the available files in the server's backup folder. When the client receives the "`send`" command together with a specific filename from the user, this file will be sent to the server and the server will store it in the backup folder. In addition, if the client receives the "`remove`" command together with a specific filename from user, it will send the command to the server and the server will delete the file from the backup folder. The client uses "`shutdown`" command to stop the server and uses "`quit`" command to terminate.

Before discussing the protocol, we define two types of messages used for implementing the protocol. One is command message (`Cmd_Msg_T`) and the other is data message (`Data_Msg_T`). The command message is used to send the command and exchange the necessary information to assist the data transmission for the functions. For example, when you implement "`ls`" function, the "`CMD_LS`" command is sent as command message. The number of files is also sent back to client as command message to assist future filename reception. However, the exact filenames are sent as the data message.

2.1 Command and Data Message

The command message contains the following three fields. To support communication between different computer architectures, we use data types defined in the `stdint.h` library. The message data type is defined in `message.h`. The command message (`Cmd_Msg_T`) contains the following elements:

- command type: 8 bits unsigned integer (`uint8_t`)
- filename: character [`FILE_NAME_LEN`]
- size: 32 bits unsigned integer (`uint32_t`)
- port: 16 bits unsigned integer (`uint16_t`)
- error: 16 bits unsigned integer (`uint16_t`)

Please note that not all the elements will be used in a single command transmission. For most of the command message, only part of its elements are used in that transmission. The data message (`Data_Msg_T`) contains:

- data: character [`DATA_BUF_LEN`]

For this assignment, we define `FILE_NAME_LEN` to be 128 and `DATA_BUF_LEN` to be 3000. Here we notice that the `size` and `port` are multi-byte integers. Transmitting them directly over the network would encounter some problem if the sender and receiver have different architectures and use different byte orders to represent integers. This potential problem can be solved by using `htons()` and `htonl()` to convert the `size` and `port` from the host machine byte order to the network byte order on the sender side for transmission. Thus the receiver can use `ntohs()` and `ntohl()` to convert the received data back to host machine byte order. However, this problem will rarely happen in your testing environment. But note that when you obtain the port number from operating system, the operating system gives the port number in network byte order and also when you use the port number to transmit message, the operating system can only understand the port number in network byte order.

2.2 "`Waiting`" state for user input

After the startup of the server program, the server initiates a UDP socket on a port. The port number can be given from command line or assigned by operating system automatically if none is given from command line. The server will then print out port number and wait for the UDP command from the client on this port. After the startup of the client program, the client will print "$ " and wait for any command from standard input (keyboard) and send the UDP command message to the server via the server's UDP port. The following message shall be printed to the console

- after the startup and everytime when the server enters '`Waiting`" state, the server shall print "Waiting UDP command @: xxxxx". (xxxxx is port number)
- if the command from user is neither of the supported commands, the client shall print " - wrong command"
- if server receives a command, it shall print "[CMD RECEIVED]: xxx". (xxx is the command label in type `Cmd_T`). See the example at the end of this handout.

2.3 "`ls`" command

This function lists the available files in the server's backup folder. It uses the UDP socket. Figure 1 shows the detailed procedure to implement this function. After the startup of the server and the client program, if there is an "`ls`" command input from user, the client will send this command through a command message with "`cmd = CMD_LS`" to the server. After receiving this message, the server will check the backup folder and send back a message with "`cmd = CMD_LS`" and "`size = N`" to inform the client the total number (N) of files that it has. If there is no files in the folder or the folder does not exist, the server will send this message with "`size = 0`", otherwise the server will transmit each filename through a data message to the client and the client will display each filename upon the reception. Upon the completion of the "`ls`" function, both the server and the client will return to initial states where the next command can be received from user input. Following are messages that need to be printed to the console.

- if the server finds the backup folder is empty, the server shall print " - server backup folder is empty" on the server side
- if the client receive the message indicating that the server is empty, the client shall print " - server backup folder is empty" on the client side
- the server and the client shall print out each filename with prefix " - " in a single line, respectively
- if the client receive a message with a "`cmd`" field other than "`CMD_LS`", it shall print " - command response error."

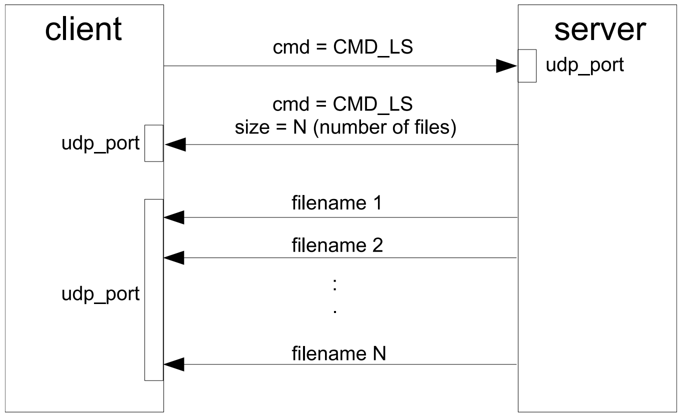


Figure 1: "ls" command.

2.4 "`send`" command

This function sends a specific file to the server and the server will store it in its backup folder. It uses both the UDP and TCP sockets to implement the funciton (Figure 2). After the startup or the completion of the previous command, both the server and the client are in the waiting status for the next user input. If the command "`send myFilename`" is received from the user, the client will send the UDP message with "`cmd=CMD_SEND`", "`filename=myFilename`", "`size=filesize`" and "`error=0`" to the server. Please note that any normal messages should have a "`error=0`". Upon the reception of the command message and file information, the server will check if a file with the same name exists in the backup folder. The protocol will be slightly different depending on the existance of the file in the backup folder.

If the file already exists, the server will send back an error message with "`error=2`" and "`cmd=CMD_SEND`", and print "file xxx exist; overwrite?". (Please note that error message has the same structure as a normal message, but a nonzero `error` code). Once the error message is received by the client, the client prints "file exists. overwrite? (y/n):" and waiting for the user input from keyboard. If anything other than "Y" or "y" is received from the user, the client sends an error message with "`error=2`" and "`cmd=CMD_SEND`" to the server. Once this error message is received by the server, the server will return back to "`Waiting`" states, and so does the client. If the user input is "Y" or "y", the client needs to send a normal message with "`error=0`" and "`cmd=CMD_SEND`" to the server.

If the file does not exist in the backup folder or a normal message "`error=0`" and "`cmd=CMD_SEND`" is sent from client in the previous step and received by the server, the server opens a file with the received filename in the backup folder. Then the server will open a TCP connection with a port number assigned by the operating system and send this TCP port number back to the client. Then, the server will wait at this port for the connection from the client for the file data reception. If the server cannot open a file for future writing, the server will send back to the client an error message with "`cmd = CMD_SEND`", "`port=0`" and "`error=1`". In this case if client receives "`error=1`", both the client and the server will return back to "`Waiting`" state.

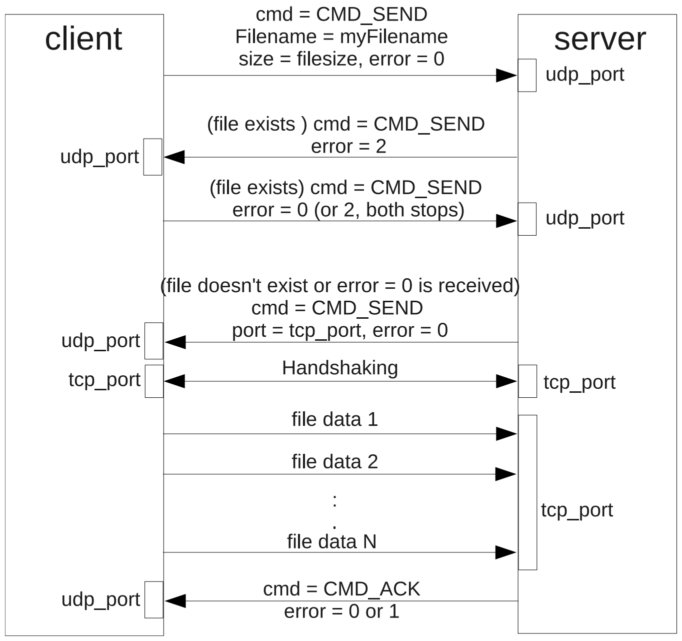


Figure 2: "send" command.

After receiving the message with a nonzero TCP port number, the client initiates the handshaking with the server at that TCP port. If the connection is established, the client will divide the original file into several segments and send each to the server until the end of file is reached. When the file data transmission completes, the client will wait for the acknowledgment message from the server. The server will write each received segment to a local file in the order of the reception. Once the server receives the whole file, the server will close the file and send back a message with "`cmd=CMD_ACK`" and "`error=0`" to the client and both the client (after receiving this acknowledgement) and the server will close the TCP connection, and then return to the "`Waiting`" state for the next user command. During the reception, the server shall print out the total number of bytes received after each reception(Figure 5). If any error happened during the server reception, the server will terminate the reception and send back the acknowledge message with "`error=1`". Following messages shall be printed to the console

- if the client receives "`error=2`", it shall print "file exists. overwrite? (y/n):" and wait for user input
- if the server receives "`error=2`", it shall return to "`Waiting`" state
- if the client cannot open the file for sending, it shall print " - cannot open file: xxx". (xxx is filename), otherwise it shall print " - filesize:xxx". (xxx is filesize in bytes)
- if the server fails to open a file for writing, it shall print " - open file xxx error.", otherwise it prints " - filename:xxx" and " - filesize:xxx"
- if TCP binding is listening on a port, the server shall print " - listen @:xxx". (xxx is TCP port number)
- if server successfully accepts the connection request, it shall print " - connected with the client."
- if any error happens during the message reception on the server, the server shall print " - message reception error."
- if the server sends the acknowledgement, it shall print " - send acknowledgement".
- if client receives "`port=0`", "`cmd`$\neq$`CMD_SEND`" or "`error=1`" from the server, the client shall prints " - error or incorrect response from server.", otherwise it prints " - TCP port:xxx". (xxx is port number)
- if client receives an acknowledgement without error (`error=0`), it shall print " - file transmission is completed.", otherwise it prints " - file transmission is failed"

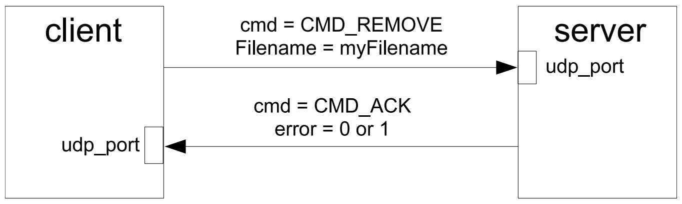


Figure 3: "remove" command.

2.5 "**`remove`**" command

This function removes a file from the server's backup folder (Figure 3). The procedure starts from the waiting status of the server and the client. During the waiting status, if the command "`remove myFilename`" is received from the user input at the client side, the client will send a command message with "`CMD_REMOVE`" and "`filename=myFilename`" to the server. Then server will try to delete this file from the backup folder. If the file exists, the server will perform the deletion and send a acknowledge message with "`cmd=CMD_ACK`" and "`error=0`" . If the file does not exist, the server will send back the acknowledge message with "`error=1`". Following messages need to be printed out to the console.

- if the corresponding file is not in the backup folder, the server shall print " - file doesn't exist"
- if the client receive with "`error=1`", it shall print " - file doesn't exist."

2.6 “quit” and “shutdown” command

The quit command is designed for client to quite from execution. Once the client receives this command, it shall exit the program. There is no need to communicate with server before quiting. However, the shutdown command is designed to be sent from the client to the server to shut the server down (Figure 4). If a shutdown command is received by the server, it sends back an acknowledgement with “error=0”, then exits the program. The client will turn to waiting status after receiving this acknowledgement. You may need to send back “error=1” if you implement the bonus part as other transmissions may be in progress during the shutdown request.

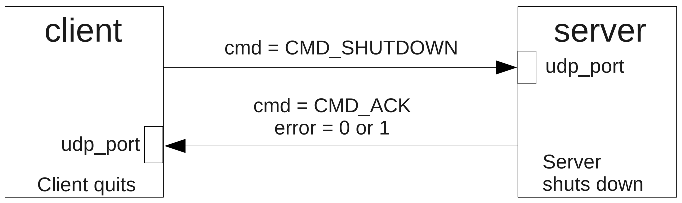


Figure 4: “shutdown” command.

2.7 Socket error handling

The socket errors should be handled by the server and client program as follows.

- for any server and the client functions, if it fails to create or bind the socket, it shall print “ - failed to create/bind TCP/UDP socket”
- if the server fails to accept the TCP request, it shall print “ - failed to accept TCP connection”.
- if the client fails to establish TCP with server, it shall print “ - failed to connect server with TCP”.
- in all the above situations, the program shall exit.

2.8 Program Requirements

1. (server side) Use “server” as the name of your server executable. The server program shall be invoked from command line and it may require an argument which indicates port number used for UDP command reception. If there is no command line input for the argument, the server shall ask the operating system to choose a port number. In either way it shall print that port out to the console as mentioned before. You shall then use that port number as an argument to start the client so that the client can use this port number to send UDP commands.

   - −p may be used to specify a port number. This can be omitted so that the operating system will choose a port number.

   Example invocation: `./server -p 35887`

2. (client side) Use “client” as the name of your client executable. It should have the following two arguments.

- `-s` is used to indicate the server address [default: `127.0.0.1`]. This argument can be omitted if the server is local. If the server is a remote computer running Debian Linux, you can run `/sbin/ifconfig` on the server to check the address of the server.
- `-p` is the UDP port command transmission. (`udp_port`). It cannot be omitted. It ranges up to 65535.

Example: `./client -s 35.9.42.35 -p 35887`
When the client starts, it will wait for the user input from the command line.

3. You need to run the server and client on different computers to test program. Their correct execution on the same computer doesn't necessarily guarantee that they can work correctly over the network.

2.9 Deliverables and Grading

You can download a package of skeleton files from the course website. You can choose to use either C or C++, but the skeleton file is given as C++. If you use C, you need to change the compiler in the `Makefile`. You do not need to follow the skeleton at all, but it may help you focus on socket programing. You can also use the provided executables to test your implementations of client/server, especially if you plan to complete part of the assignment described later. These executables are compiled and linked on `black` server. After you test your program, you need to deliver following files.

`server.cc` - server source code

`server.h` - server header

`client.cc` - client source code

`client.h` - client header

`message.h` - message formats shared between server and client

`*.cc` and `*.h` - any extra source code required for the program

`makefile` - compilation script to produce executables

You need to submit all the aforementioned deliverables to make sure your program can compile, even if you do not change some of them. However, you can come up with a completely different file structure, as long as it can compile by using makefile. If you only complete part of the assignment, the entire package shall still compile and you need to describe what exactly is implemented in `README.txt` file. If there are any extra comments for your programs, please submit a `README.txt` file to include your comments. The code should compile and link on `adriatic.cse.msu.edu`. Programs that do not compile will receive a zero score. Please test your programs and make sure they can compile before handing them in.

The executables are provided so that you can use them to test your program. For example, you can use the server's executable "`server`" to test your client program and vice versa. Following is the score assigned for each module.

- client "`ls`" function - 25pts
- client "`send`" function - 15pts
- client "`remove`" funciton - 5pts

- client "`quit`" function - 5pts
- client "`shutdown`" function - 5pts
- server "`ls`" function - 20pts
- server "`send`" function - 15pts
- server "`remove`" funciton - 5pts
- server "`shutdown`" function - 5pts

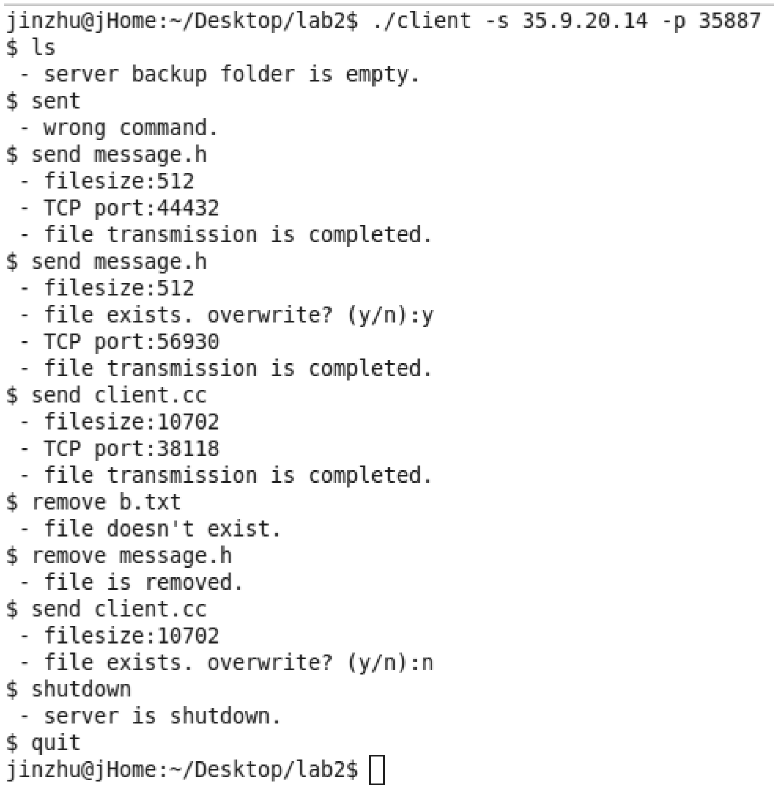
Please note that for "`send`" function, you need to make sure that the file received on the server side is correct. That is, the file received at the server must EXACTLY match the file sent by the client. You can transmit a binary file (pdf or image) to test the integrity of the file by opening it after transmission over the network. Another way is to use program 'diff' to compare the two files. A pdf file will be used for grading. If the file received is not correct (cannot open, incorrect file size, etc.), 30% will be deduced from your score of "`send`" function.

If your program can only work locally, you will receive 10% deduction in the final score.

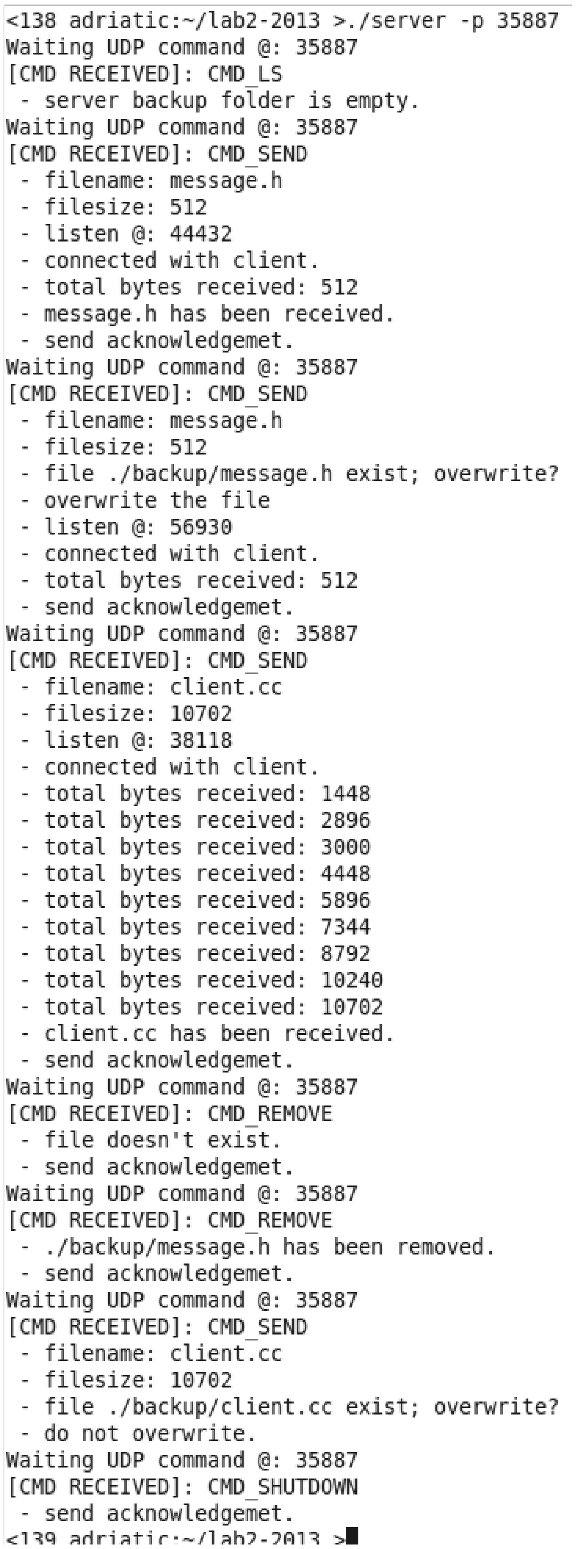
Bonus score: 15pts. The current implementation of the backup system can only handle one client at a time. For example, if client A is transmitting a large file to the server, client B cannot send any command to the server. Also the client has to wait for the completion of the current command before it can send another new command. This bonus assignment requires you to design the system so that the client/server can send/receive new command while another command is still being executed. Hint: you may consider using multi-thread/process programing (be careful with the port used for different threads/processes). You can make necessary changes to the protocol for this bonus assignment. For example, you can add/remove the messages exchanged between the client and server if neccessary. However, you need to make sure that the server uses the same port number for all the incoming commands. When you print the message to the console, add "task:xxx" (xxx is process id or a number that can be used to differentiate the concurrent command tasks) in front of all the messages. Please note that your system shall be able to handle the situation when two clients simultaneously request to overwrite the same file. To submit a solution for this bonus assignment, you need to describe the details of design for the system in a file and include the filename in your `README.txt` file.

You should submit your source code via the handin utility. All files must be submitted by *11:59 PM on Wednesday, April 17, 2013*. Late submissions will not be accepted.

3 Example Output



```
jinzhu@jHome:~/Desktop/lab2$ ./client -s 35.9.20.14 -p 35887
$ ls
 - server backup folder is empty.
$ sent
 - wrong command.
$ send message.h
 - filesize:512
 - TCP port:44432
 - file transmission is completed.
$ send message.h
 - filesize:512
 - file exists. overwrite? (y/n):y
 - TCP port:56930
 - file transmission is completed.
$ send client.cc
 - filesize:10702
 - TCP port:38118
 - file transmission is completed.
$ remove b.txt
 - file doesn't exist.
$ remove message.h
 - file is removed.
$ send client.cc
 - filesize:10702
 - file exists. overwrite? (y/n):n
$ shutdown
 - server is shutdown.
$ quit
jinzhu@jHome:~/Desktop/lab2$
```



```
<138 adriatic:~/lab2-2013 >./server -p 35887
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_LS
 - server backup folder is empty.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_SEND
 - filename: message.h
 - filesize: 512
 - listen @: 44432
 - connected with client.
 - total bytes received: 512
 - message.h has been received.
 - send acknowledgemet.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_SEND
 - filename: message.h
 - filesize: 512
 - file ./backup/message.h exist; overwrite?
 - overwrite the file
 - listen @: 56930
 - connected with client.
 - total bytes received: 512
 - send acknowledgemet.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_SEND
 - filename: client.cc
 - filesize: 10702
 - listen @: 38118
 - connected with client.
 - total bytes received: 1448
 - total bytes received: 2896
 - total bytes received: 3000
 - total bytes received: 4448
 - total bytes received: 5896
 - total bytes received: 7344
 - total bytes received: 8792
 - total bytes received: 10240
 - total bytes received: 10702
 - client.cc has been received.
 - send acknowledgemet.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_REMOVE
 - file doesn't exist.
 - send acknowledgemet.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_REMOVE
 - ./backup/message.h has been removed.
 - send acknowledgemet.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_SEND
 - filename: client.cc
 - filesize: 10702
 - file ./backup/client.cc exist; overwrite?
 - do not overwrite.
Waiting UDP command @: 35887
[CMD RECEIVED]: CMD_SHUTDOWN
 - send acknowledgemet.
<139 adriatic:~/lab2-2013 >
```

Figure 5: Example of the **client (left)** and **server (right)**.