SPL - 171
Assignment 3

Published on: 3.1.2017
Due date: 20.1.2017
Responsible TA's: **Tom Mahler**, **Matan Drory**

1 General Description

In this assignment you will implement an extended TFTP (Trivial File Transfer Protocol) server and client. The communication between the server and the client(s) will be performed using a binary communication protocol, which will support the upload, download and lookup of files. Please read the entire document before starting.

The implementation of the server will be based on the **Reactor** and **Thread-Per-Client (TPC)** servers taught in class. The servers, as seen in class, do not support bi-directional message passing. Any time the server receives a message from a client it can replay back to the client itself. But what if we want to send messages between clients, or braodcast an announcment to all the clients. The first part of the assignment will be to replace some of the current interfaces with new interfaces that will allow such a case. Note that this part changes the servers pattern and **must not know** the specific protocol it is running. The current server pattern also works that way (Generics and interfaces).

Once the server implementation has been extended you will have to implement an example protocol. The extended TFTP Specification The TFTP (Trivial File Transfer Protocol) allows users to upload and download files from a given server. Our extended version will require a user to perform a passwordless server login as well as enable the server to communicate broadcast messages to all users and support for directory listings. It is a binary protocol (non-text-base). Binary protocols are very efficient when it comes to reducing the bandwidth used. The commands are defined by an **opcode** that describes the incoming command. For each command, a different length of data needs to be read according to it's specifications. In the following sections we will define the specifications of the commands supported by the extended TFTP.

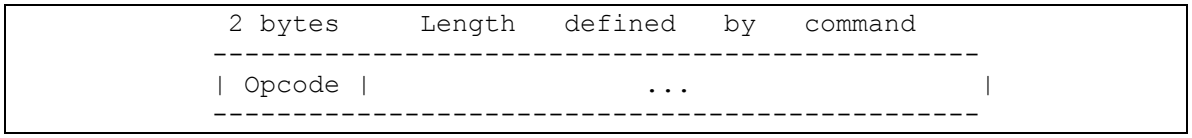
The original TFTP protocol is based on UDP. Since our servers are TCP based, we altered the original protocol.

1.1 Establishing a client/server connection

Upon connecting, a client must specify their username using a Login command. The nickname must be unique and cannot be changed after it is set. Once the command is sent, the server will reply on the validity of the username. Once a user is logged in successfully, he can submit other commands. The login command is stated in the following section.

1.2 Supported Commands

The extended TFTP supports various commands needed in order receive and upload files. There are two types of commands, Server-to-Client and Client-to-Server. The commands begin with 2 bytes (short) to describe the **opcode**. The rest of the message will be defined specifically for each command as such:



We begin with description of Client-to-Server packets. The extended TFTP supports 10 types of packets:

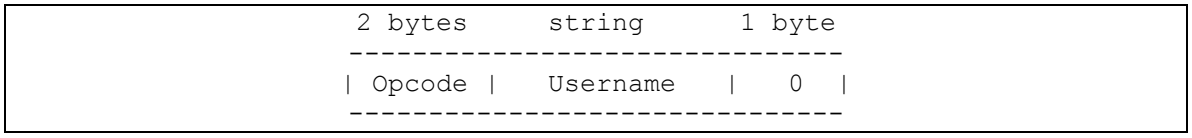
opcode	operation
1	Read request (RRQ)
2	Write request (WRQ)
3	Data (DATA)
4	Acknowledgment (ACK)
5	Error (ERROR)
6	Directory listing request (DIRQ)
7	Login request (LOGRQ)
8	Delete request (DELRQ)
9	Broadcast file added/deleted (BCAST)
10	Disconnect (DISC)

Encoding and Decoding binary information:

In this assignment, you must decode and encode binary data. When encoding anything that takes more than a byte (int, short, float ect…) you must consider how the number is held in memory. For that reason, when passing numbers via a network protocol we use Big-endian, You can read more about Endianness. In this assignment we provide 4 functions that handle short to byte[2] and byte[2] to short conversions on both Java and C++ (Note that in C++ byte[] is char[]). Their code can be found in the assignment page under the **FAQ** section. **You MUST use and understand them**. Working with little-endian will resolve in tests failing.

LOGRQ Packets:

Packets have the following format:



A LOGRQ packet is used to login a user into the server. This packet must be the first packet to be sent by the client to the server, or an ERROR packet is returned. If successful an ACK packet will be sent in return.

Parameters:

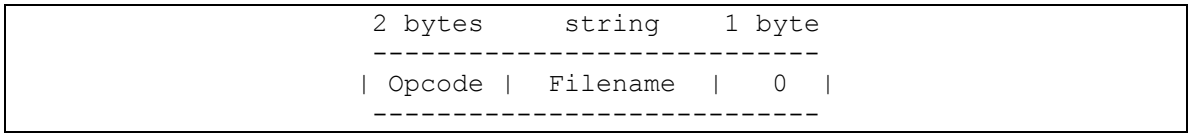
- Opcode: 7.
- Username**:** The username to register in the server. If the user already exists, sends an ERROR packet.

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **LOGRQ <Username>**

DELRQ Packets:

Packets have the following format:



A DELRQ packet is used to request the deletion of a file in the server.

Parameters:

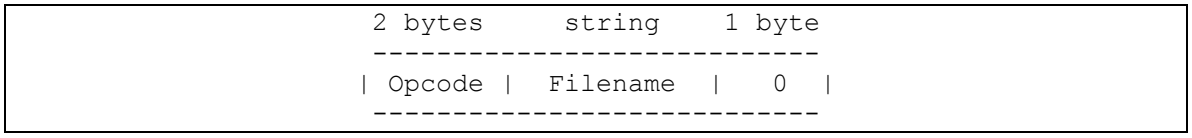
- Opcode: 8.
- Filename**:** a sequence of bytes in UTF-8 terminated by a zero byte.

Command initiation:

- This command is initiated by entering the following text in the client command line interface: **DELRQ <Filename>**

RRQ/WRQ Packets:

Packets have the following format:



Packets that appear only in a Client-to-Server communication. The 1 byte "0" is used to specify end of filename string. Note that for this reason it is important to make sure that you **don't** have 0 byte (‘\0’) in the filename string. Once the command has been acknowledges the file is transferred using DATA packets. Files should be saved in the ***Files*** folder on the server side and in the **current working directory** in the client side.

Parameters:

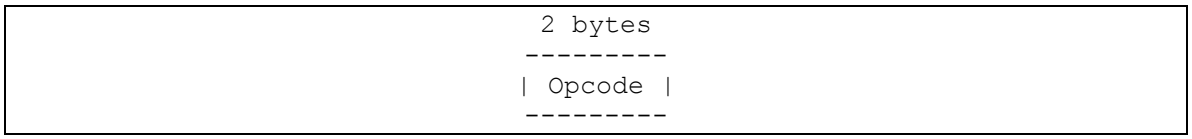
- Opcode: either 1 for RRQ (read request) or 2 for WRQ (write request).
- Filename**:** a sequence of bytes in UTF-8 terminated by a zero byte.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **<RRQ/WRQ> <file name>**

DIRQ Packets:

Packets have the following format:



A DIRQ packet requests a directory listing from the server. The directory listing will be returned as DATA packets. The DATA packet should return as a string of file names divided by ‘\0’. Note: directory listing should not include files currently uploading but should include any file in the ***Files*** directory.

Parameters:

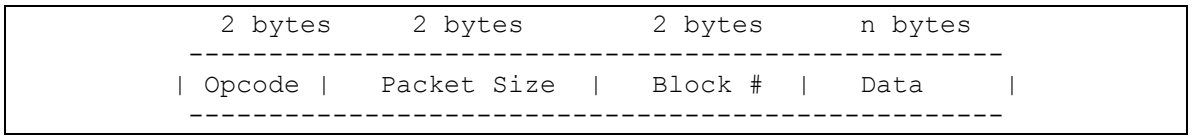
- Opcode: 6.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **DIRQ**

DATA Packets:

Packets have the following format:



These packets can appear either in a Client-to-Server communication (uploading a file) or in a Server-to-Client communication (downloading a file or dir listing). If a file or directory listing are longer than 512 bytes, the data will be divided to blocks of 512 bytes. The next block will be sent only after the former is acknowlaged.

Parameters:

- Opcode: 3.
- Packet Size**:** the size of the data (in bytes) in this packet. Maximal data is 512 bytes. (The size of n)
- Block # (number)**:** begin with 1 and increase by one for each new block of data. This restriction allows for an implementation that does not care for order.
- Data: field from zero to 512 bytes long. If it is 512 bytes long, the block is not the last block of data; if it is from zero to 511 bytes long, it signals the end of the transfer. (See the section on Normal Termination for details).

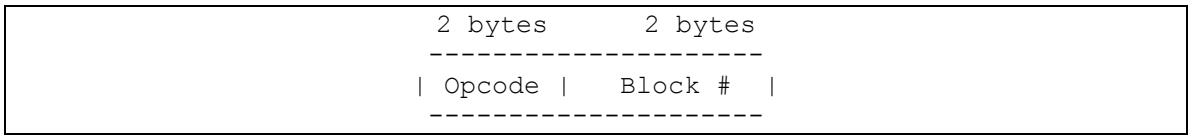
Normal Termination:

A normal termination of file is marked by a data packet with packet size smaller then 512. The original TFTP protocol used that specific number to conform to the size of a UDP datagram packet. We will use it to acknowledge specific blocks (Next command) and so we will be able to receive braodcast messages between file blocks. If a large file was sent in one message, we would not be able to receive messages in the client while that single message is being sent.

- Upon successfull completion of Download / Upload The **Client** should write to screen: **<RRQ/WRQ> <file name> complete**
- Upon successfull completion of Directory listing, The **Client** shoud write to screen:
  <file name 1> \n
  <file name 2> \n
  …
  <file name n> \n

ACK Packets:

Packets have the following format:



ACK packets are used to acknowledge different packets. The block number is used when acknowledging a DATA packet. Once a DATA packet is acknowledged, The next block can be sent. Other packets: LOGRQ, WRQ, DELRQ and DISC should be acknowledged with block = 0 if successful.

Parameters:

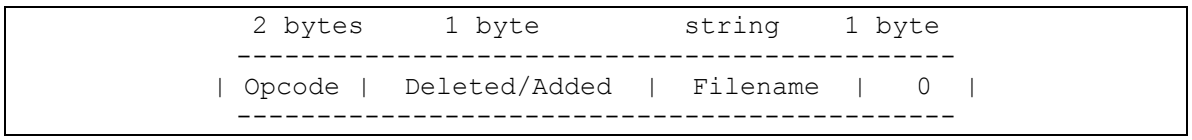
- Opcode: 4.
- Block # (number)**:** an ACK echoes the block number of the DATA packet being acknowledged. A WRQ for example is acknowledged with an ACK packet having a block number of zero.

Acknowledgment Notification:

- Any ACK message received in **client** should be written to screen:
  ACK <block no>

BCAST Packets:

Packets have the following format:



A BCAST packet is used to notify **all** logged-in clients that a file was deleted/added. This means users that are connected but have not completed a successfull login should not receive this message. This is a Server to client message noly.

Parameters:

- Opcode: 9.
- Deleted/added: indicates if the file was deleted (0) or added (1).
- Filename**:** a sequence of bytes in UTF-8 terminated by a zero byte.

BCAST Behaviour:

- A file has completed **uploading** to the server.
- A file was **deleted** from the server.
- When a client receives a BCAST message it should print:
  BCAST <del/add> <file name>

ERROR Packets:

Packets have the following format:

```
                2 bytes     2 bytes      string    1 byte
                ----------------------------------------
               | Opcode |  ErrorCode |   ErrMsg   |   0  |
                ----------------------------------------
```

An ERROR packet may be the acknowledgment of any other type of packet. In case of error, an error packet should be sent. A list or errors shown below.

Parameters:

- Opcode: 5.
- Error code**:** a short indicating the nature of the error. The table of values and meanings is given below:

Value	Meaning
0	**Not defined, see error message (if any).**
1	**File not found – RRQ \ DELRQ of non-existing file**
2	**Access violation – File cannot be written, read or deleted.**
3	**Disk full or allocation exceeded – No room in disk.**
4	**Illegal TFTP operation – Unknown Opcode.**
5	**File already exists – File name exists on WRQ.**
6	**User not logged in – Any opcode received before Login completes.**
7	**User already logged in – Login username already connected.**

- Error message: intended for human consumption, and should be in UTF-8. Like all other strings, it is terminated with a zero byte.

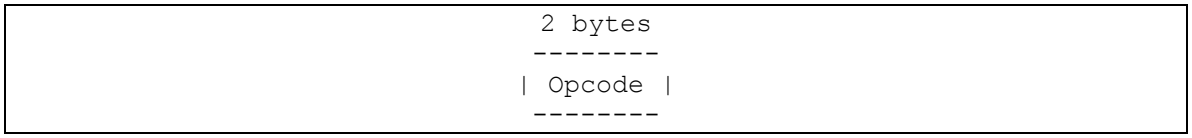
Error Notification:

- Any error message received in **client** should be written to screen:
  Error <Error No>

Note that is more then one error applys, select the lower error code. Example: Client sends undefined opcode before loggins in, the correct error code is 4 (illegal TFTP operation) even though the error 6 (User not logged in) also applys.

DISC Packets:

Packets have the following format:



Packets that appear only in a Client-to-Server communication. Informs the server on client disconnection. Client may terminate only after reciving ACK packet in replay.

Parameters:

- Opcode: 10.

Command initiation:

- This command is initiated by entering the following texts in the client command line interface: **DISC**

2 Implementation Details

2.1 General Guidelines

- The server should be written in Java. The client should be written in C++ with BOOST. Both should be tested on Linux installed at CS computer labs.

- You must use maven as your build tool for the server and MakeFile for the c++ client.

- The same coding standards expected in the course and previous assignments are expected here.

2.2 Server

You will have to implement a single protocol, supporting both the **Thread-Per-Client** and ***Reactor*** server patterns presented in class. Code seen in class for both servers is included in the assignment wiki page. You are also provided with 3 new or changed interfaces:

- **Connections** – This interface should map a unique ID for each active client connected to the server. The implementation of Connections is part of the server pattern and not part of the protocol. It has 3 functions that you must implement (You may add more if needed):
    - **boolean send(int connId, T msg)** – sends a message T to client represented by the given connId
    - **void broadcast(T msg)** – sends a message T to **all** active clients. This includes clients that has not yet completed log-in by the extended TFTP protocol. Remember, Connections<T> belongs to the server pattern implemenration, not the protocol!.
    - **void disconnect(int connId)** – removes active client connId from map.

- **ConnectionHandler<T>** - A function was added to the existing interface.
    - **Void send(T msg)** – sends msg T to the client. Should be used by **send** and **broadcast** in the **Connections** implementation.

- **BidiMessagingProtocol** – This interface replaces the MessagingProtocol interface. It exists to support peer 2 peer messaging via the Connections interface. It contains 2 functions:
    - **void start(int connectionId, Connections<T> connections)** – initiate the protocol with the active connections structure of the server and saves the owner client's connection id.
    - **void process(T message)** – As in MessagingProtocol, processes a given message. Unlike MessagingProtocol, responses are sent via the *connections* object **send** function.

Left to you, are the following tasks:

1. Implement ***Connections<T>*** to hold a list of the new ConnectionHandler interface for each active client. Use it to implement the interface functions. Notice that given a connections implementation, any protocol should run. This means that you keep your implementation of *Connections* on T.
   public class ConnectionsImpl<T> implements Connections<T> {…}.

2. Refactor the **Thread-Per-Client** server to support the new interfaces. The *ConnectionHandler* should implement the new interface. Add calls for the new *Connections<T>* interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol<T>.

3. Refactor the **Reactor** server to support the new interfaces. The *ConnectionHandler* should implement the new interface. Add calls for the new *Connections<T>* interface. Notice that the ConnectionHandler<T> should now work with the BidiMessagingProtocol<T> interface instead of MessagingProtocol<T>.

4. **Tasks 1 to 3 MUST not be specific for the protocol implementation**. Implement the new *BidiMessagingProtocol* and *MessageEncoderDecoder* to support the extended TFTP protocol as described in section 1.2. You will also need to define messages(<T> in the interfaces). You may add more classes as neccesery to implement the protocol (shared protocol data ect…).

Leading questions:

- Which classes and interfaces are part of the Server pattern and which are part of the Protocol implementation?
- When and how do I register a new connection handler to the **Connections** interface implementation?
- When do I call **start** to initiate the connections list? **Start** must end before any call to **Process** occurs. What are the implications on the reactor?
- How do you collect a message? Are all packet types collected the same way?
- How do I implement **BCAST**? as it should send a message to all **logged-in** clients.

Testing run commands:

- Reactor server**:**
  mvn exec:java -Dexec.mainClass="bgu.spl171.net.impl.TFTPreactor.ReactorMain" -Dexec.args="<port>"
- Thread per client server**:**
  mvn exec:java -Dexec.mainClass="bgu.spl171.net.impl.TFTPtpc.TPCMain" -Dexec.args="<port>"

The **server** directory should contain a **pom.xml** file, a **Files** directory (for the uploaded files) and the **src** directory. Compilation will be done from the server folder using:
mvn compile

2.3 Client

An echo client is provided, but its a single threaded client. While it is blocking on stdin (read from keyboard) it does not read messages from the socket. You should improve the client so that it will run 2 threads. One should read from keyboard while the other should read from socket. Both threads may write to the socket. The client should receive the server's IP and PORT as arguments. You may assume a network disconnection does not happen (like disconnecting the network cable). You may also assume that a new commands will **not** be entered to the terminal while waiting for file download, directory listing or acknowlagment of any command sent.

The client should recive commands using the standard input. Commands are defined in section 1.2. Notice that the client should not close until he recives an **ACK** packet for the **DISC** call.

The **Client** directory should contain a src, include and bin subdirectories and a Makefile as shown in class. The output executable for the client is named **TFTPclient** and should reside in the bin folder after calling **make**.

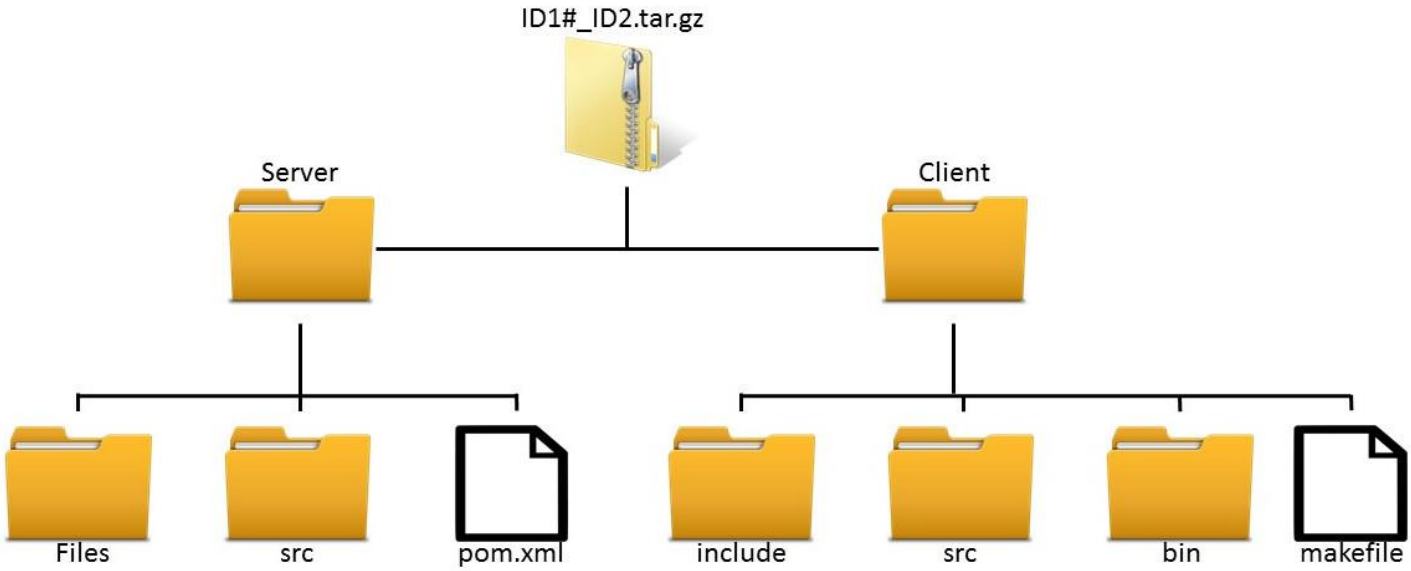
Testing run commands: **TFTPclient <ip> <port>**

3 Submission instruction

- Submission is done only in pairs. If you do not have a pair, find one. You need explicit authorization from the course staff to submit without a pair. You cannot submit in a group larger than two.
- You must submit one .tar.gz file with all your code. The file should be named "ID#1_ID#2.tar.gz". Note: We require you to use a .tar.gz file. Files such as .rar, .zip, .bz, or anything else which is not a .tar.gz file will not be accepted and your grade will suffer.
- Extension requests are to be sent to majeek. Your request email must include the following information:
    - Your name and your partners name.
    - Your id and your partners id.
    - Explanation regarding the reason of the extension request.
    - Offcial certification for your illness or army drafting.

  Requests without a compelling reason will not be accepted

- The submitted file should contain a **Client** directory and a **Server** directory (Their content was explained in the implementation section).



4 Examples

The following section contains examples of commands running on client. It assumes that the software opened a socket properly and a connection has been initiated.
We use "<" for keyboard input and ">" for screen output at the client side only. Server and client actions are explained in between.

4.1 Login and file download

Server assumptions for example:

- User named Timmy is currently **not** logged-in
- The server has a file named Living_A_Lie.mp3 the is 2.4KB long

```
< LOGRQ Timmy
(Server checks if a user named Timmy is logged-in)
> ACK 0
< RRQ Living_A_Lie.mp3
(server sends DATA packet with opcode = 3, packet size =512, block = 1 and
512 bytes of file data)
(Client sends ACK 1)
(server sends DATA packet with opcode = 3, packet size =512, block = 2 and
512 bytes of file data)
(Client sends ACK 2)
…
(server sends DATA packet with opcode = 3, packet size =352, block = 5 and
352 bytes of file data)
(Client sends ACK 5)
> RRQ Living_A_Lie.mp3 complete
< DISC
(Server removes Timmy from logged-in list)
> ACK 0
```

4.2 Login and file upload

Server assumptions for example:

- User named Timmy is currently **not** logged-in
- The server does not have a file named Living_A_Lie.mp3
- The client has a file named Living_A_Lie.mp2 the is 2.4KB long

```
< LOGRQ Timmy
(Server checks if a user named Timmy is logged-in)
> ACK 0
< WRQ Living_A_Lie.mp3
(Server checks if such a file exists, sees that it does not and sends an ACK
packet)
> ACK 0
(client sends DATA packet with opcode = 3, packet size =512, block = 1 and
512 bytes of file data)
(Server sends ACK 1)
> ACK 1
(client sends DATA packet with opcode = 3, packet size =512, block = 2 and
512 bytes of file data)
(Server sends ACK 2)
> ACK 2
…
(client sends DATA packet with opcode = 3, packet size =352, block = 5 and
352 bytes of file data)
(Server sends ACK 5)
(Server sends BCAST to all logged in clients opcode = 9, added = 1, file name =
Living_a_Lie.mp3)
> ACK 5
> WRQ Living_A_Lie.mp3 complete
> BCAST add Living_A_Lie.mp3
< DISC
(Server removes Timmy from logged-in list)
> ACK 0
```

4.3 Errors, DIRQ and DELRQ

Server assumptions for example:

- User named Timmy is currently logged in
- User named Jimmy is currently **not** logged in
- The server contains 2 file:
    - Living_A_lie.mp3 (2.4KB)
    - Jimmy_Best_Jokes.txt (500B)

< DIRQ
(Server creates an error message since user is not logged in yet. Opcode = 5, error code = 6 ,error message= whatever you want and a byte with 0)
> Error 6
< LOGRQ Timmy
(Server checks if a user named Timmy is logged-in, since it is an error is sent)
> Error 7
< LOGRQ Jimmy
(Server checks if a user named Jimmy is logged-in)
> ACK 0
< WRQ Living_A_Lie.mp3
(Server checks if such a file exists, since it exists, an error is sent)
> Error 5
< DIRQ
(Server sends DATA packet with opcode = 3, packet size =38, block = 1 and 38 bytes that are “Living_A_Live.mp3” + ‘\0’ + “Jimmy_Best_Jokes.txt” + ‘\0’)
(Client sends ACK 1)
> Living_A_Lie.mp3
> Jimmy_Best_Jokes.txt
< DELRQ Living_A_Lie.mp3
(Server checks if a file by that name exists and then deletes it and sends an ack packet)
(Server sends BCAST to all logged in clients opcode = 9, added = 0, file name = Living_a_Lie.mp3)
> ACK 0
> BCAST del Living_A_Lie.mp3
< DISC
(Server removes Jimmy from logged-in list)
> ACK 0