SPL241: Assignment 3 instructions

Hedi Zisling and Or Kadosh

March 3, 2024

Contents

1 General Description

In this assignment, you will implement an extended TFTP (Trivial File Transfer Protocol) server and client. the TFTP server is a file transfer protocol allowing multiple users to upload and download files from the server and announce when files are added or deleted to the server. 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 **Thread-Per-Client** (TPC) servers taught in class, 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 broadcast an announcement 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 (as described in Section 4.3). Note that this part changes the server pattern and **must not know** the specific protocol it is running. The current server pattern also works that way (Generics and interfaces).

To get you started, we supply a few examples for different protocols and clients, the most complete ones being **newsfeed** and **echo**, we recommend you go over them. Specifically.

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

NOTE: The examples protocol doesn't use a bi-directional message-passing interface. When you start modifying the server to use bi-directional message-passing they will not compile anymore and you can delete them.

2 Client Behavior

In this section, we define the client's Behavior and command. The client uses two threads one thread is reading input from the user keyboard and the other one reads input from the socket (will be named listening thread). **Keyboard thread** reads commands from the keyboard and sends packets to the server defined by the command. **Listening thread** reads packets from the socket and displays messages or sends packets in return.

2.1 Keyboard thread Commands

In this section, we describe the Commands that we can type to our terminal and the resulting behavior. We enter the command into the terminal and send it by pressing enter.

2.1.1 LOGRQ

- Login User to the server
- Format: **LOGRQ <Username>**
- Result: Sending a **LOGRQ packet** to the server with the <Username> and **waiting** for **ACK** with block number 0 or **ERROR** packet to be received in the **Listening thread**.
- Example: LOGRQ KELVE_YAM
- **NOTE**: KELVE_YAM is not equal to kelve_yam they have two different UTF-8 encodings.

2.1.2 DELRQ

- Delete File from the server.
- Format: **DELRQ <Filename>**
- Result: Sending a **DELRQ packet** to the server with the <Filename> and **waiting** for **ACK** with block number 0 or **ERROR** packet to be received in the **Listening thread**.
- Example: DELRQ lehem hvita
- **NOTE**: "lehem hvita" is one filename with space char in it (this is ok).

2.1.3 RRQ

- Download file from the server Files folder to current working directory.
- Format: **RRQ <Filename>**
- Result: Creating a file in current working directories(if not exist) and then send a **RRQ packet** to the server with the <Filename> and **waiting** for **file to complete the transfer** (Server sending DATA Packets) or **ERROR** packet to be received in the **Listening thread**.
- Error handling:
  - **File already exists in the client side**: print to terminal "file already exists" and **don't** send **RRQ packet** to the server.
  - **Listening thread received an Error**: deleted created file.
- On complete transfers: print to terminal "RRQ <Filename> complete".
- Example of Command: RRQ kelve_yam.mp3

2.1.4 WRQ

- Upload File from current working directory to the server.
- Format: **WRQ <Filename>**
- Result: Check if file exist then send a **WRQ packet** and wait for **ACK** or **ERROR** packet to be received in the **Listening thread**. If received **ACK** start transferring the file.
- Error handling:
  - **File does not exist in the client side**: print to terminal "file does not exists" and **don't** send **WRQ packet** to the server.
  - **Listening thread received an Error**: stop transfer.

- On complete transfers: print to terminal "WRQ <Filename> complete".
- Example of Command: WRQ Operation_Grandma.mp4

2.1.5 DIRQ

- List all the file names that are in Files folder in the server.
- Format: **DIRQ**
- Result: Sending a **DIRQ packet** to the server with **waiting** for the filenames to **complete the transfer**(server sending DATA pakets) or an **ERROR** packet to be received in the **Listening thread**.
- Error handling:
  - **Listening thread received an Error**: Nothing.
- On complete transfers of file names: print to terminal:

  <file name 1>\n
  <file name 2>\n
  ...
  <file name n>\n

2.1.6 DISC

- Disconnect (Server remove user from Logged-in list) from the server and close the program.
- Format: **DISC**
- Result: Check if User is logged in.
  - User is logged in sending **DISC packet** and waits for **ACK** with block number 0 or **ERROR** packet to be received in the **Listening thread** then closes the socket and exit the client program.
  - User is not logged in close socket and exit the client program.
- Error handling:
  - **Listening thread received an Error**: Don't exit the program.

2.2 Listening thread

In this section, we describe the behavior of the listing thread resulting from received packets.

2.2.1 DATA Packet

When received a **DATA packet** save the data to a file or a buffer depending if we are in **RRQ Command** or **DIRQ Command** and send an **ACK packet** in return with the corresponding block number written in the DATA packet.

2.2.2 ACK Packet

Print to the terminal the following:

ACK <block number>

2.2.3 BCAST Packet

Print to the terminal the following:

BCAST <del/add> <file name>

2.2.4 Error Packet

Print to the terminal the following:

Error <Error number> <Error Message if exist>

3 TFTP Protocol

The extended TFTP supports various commands for receiving and uploading files.

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 commands which deal with file transferring.

3.1 Packets

In this section, we describe the various massages of the protocol. Each of these messages should be implemented in the server.

The packet format of each message type is described with an example in HEX code of every byte in the packet, strings are written in the bytes encoded in UTF-8.

3.2 Supported Packets

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:

2 bytes	Length defined by command
Opcode	...

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

opcode	Value	bytes	HEX Value	operation	length in bytes
RRQ	1	[0, 1]	0x0001	Read request	2 + bytes-size(filename in UTF8) + 1
WRQ	2	[0, 2]	0x0002	Write request	2 + bytes-size(filename in UTF8) + 1
DATA	3	[0, 3]	0x0003	Data packet	2 + 2 + 2 + packet size
ACK	4	[0, 4]	0x0004	Acknowledgment	2 + 2 (block no.)
ERROR	5	[0, 5]	0x0005	Error	2 + 2 (ERROR code) + bytes-size(ERRMsg in UTF8) + 1
DIRQ	6	[0, 6]	0x0006	Directory listing request	2
LOGRQ	7	[0, 7]	0x0007	Login request	2 + bytes-size(username in UTF8) + 1
DELRQ	8	[0, 8]	0x0008	Delete file request	2 + bytes-size(filename in UTF8) + 1
BCAST	9	[0, 9]	0x0009	Brodcast file added/deleted	2 + 1 + bytes-size(filename in UTF8) + 1
DISC	10	[0, 10]	0x000A	Disconnect	2

3.2.1 LOGRQ

The packets of this message type have the following format:

type/size	2 bytes	UTF-8 string	1 byte
definition	Opcode	Username	0
example	0, 7,	6b, 65, 6c, 65, 76, 5f, 79, 61, 6d,	0

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 is already logged in, the server should return an Error packet. The Username is a sequence of bytes encoded in UTF-8 and terminated by a zero byte.

3.2.2 DELRQ

The packets of this message type have the following format:

type/size	2 bytes	UTF-8 string	1 byte
definition	Opcode	Filename	0
example	0, 8,	d7, 91, d7, 95, d7, a8, d7, a7, d7, a1, 20, d7, 94, d7, a2, d7, 92, d7, 9c, d7, 94,	0

Parameters

- Opcode: 8.
- Filename: The Filename of the file to delete from the server Files folder. The Filename is a sequence of bytes encoded in UTF-8 and terminated by a zero byte.

3.2.3 RRQ/WRQ

The packets of this message type have the following format:

type/size	2 bytes	UTF-8 string	1 byte
definition	Opcode	Filename	0
example RRQ	0, 1,	6c, 65, 68, 65, 6d, 5f, 68, 61, 76, 69, 74, 61,	0
example WRQ	0, 2,	6c, 65, 68, 65, 6d, 5f, 68, 61, 76, 69, 74, 61,	0

Packets that appear only in a Client-to-Server communication. The 1 byte "0" is used to specify the end of the filename string. Note that for this reason, it is important to make sure that you don't have 0 byte in the filename string. Once the command has been acknowledged 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 on the client side.

Parameters

- Opcode: 1 for RRQ and 2 for WRQ.
- Filename: The Filename of the file to write/read from the server Files folder. The Filename is a sequence of bytes encoded in UTF-8 and terminated by a zero byte.

3.2.4 DIRQ

The packets of this message type have the following format:

type/size	2 bytes
definition	Opcode
example	0, 6

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 byte.

Example of the Data that returned(this just the data part and not the full packet): [6c, 65, 68, 65, 6d, 5f, 68, 61, 76, 69, 74, 61, **0**, d7, 91, d7, 97, d7, 95, d7, a8, 20, d7, 98, d7, 95, d7, 91, **0**, d7, 90, d7, 99, d7, a6, d7, 99, d7, a7, 20, d7, 96, d7, 99, d7, a0, d7, 95, **0**, d7, a6, d7, 99, d7, 95, d7, 9f, 20, d7, 9e, d7, 9c, d7, 9b, d7, 94] you can see that after every filename is separated by **0** byte.

Note: directory listing should not include files currently uploading but should include any file in the **Files** directory.

Parameters

- Opcode: 6.

3.2.5 DATA

The packets of this message type have the following format:

type/size	2 bytes	2 bytes	2 bytes	n byte
definition	Opcode	Packet Size	Block Number (n)	Data
example	0, 3,	0, 1a,	0, 1	d7, a6, d7, 95, d7, 95, d7, aa, 20, d7, a9, d7, 90, d7, a0, d7, 99, 20, d7, 91, d7, 95, d7, a0, d7, 94

Parameters

- Opcode: 3.
- Packet Size: the size of the data (in bytes) in this packet. The maximal data section size is 512 bytes and the minimal size is 0.
- Block Number: begin with 1 and increase by one for each new block of data.

- Data: field from zero to 512 bytes long. if it is 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 a file is marked by a data packet with a data section size smaller than 512. We will use it to acknowledge specific blocks we sent before we send the next block of data. If we want to send a file larger than 512 bytes size we will send it in several blocks when each block number increase by one and start from 1.

3.2.6 ACK

The packets of this message type have the following format:

type/size	2 bytes	2 bytes
definition	Opcode	Block Number
example	0, 4,	0, 1

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: ab ACK echoes the block number of the DATA packet being acknowledged. A WRQ for example is acknowledged with and ACK packet having a block number of zero.

3.2.7 BCAST

The packets of this message type have the following format:

type/size	2 bytes	1 byte	UTF-8 string	1 byte
definition	Opcode	Deleted/Added	Filename	0
example	0, 9	1,	63, 6f, 6d, 65, 20, 74, 6f, 20, 4e, 69, 73, 6f,	0

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

Parameters

- Opcode: 9.
- Deleted/Added: indicates if the file was deleted (0) or added (1).
- Filename: The Filename of the file to delete from the server Files folder. The Filename is a sequence of bytes encoded in UTF-8 and terminated by a zero byte.

3.2.8 ERROR

The packets of this message type have the following format:

type/size	2 bytes	2 bytes	UTF-8 string	1 byte
definition	Opcode	ErrorCode	ErrMsg	0
example	0, 5,	0, 7	55, 73, 65, 72, 20, 61, 6c, 72, 65, 61, 64, 79, 20, 6c, 6f, 67, 67, 65, 64, 20, 69, 6e, 20, e2, 80, 93, 20, 4c, 6f, 67, 69, 6e, 20, 75, 73, 65, 72, 6e, 61, 6d, 65, 20, 61, 6c, 72, 65, 61, 64, 79, 20, 63, 6f, 6e, 6e, 65, 63, 74, 65, 64, 2e,	0

Parameters

- Opcode: 5.
- Error code: a short indication of the nature of the error. the table of values and meanings is given below.
- Error message: intended for human consumption, and should be in UTF-8. like all other strings, it is terminated with a zero byte.

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.

Note: if more than one error applies, select the lower error code.

Example: The client sends an undefined opcode before logging in, the correct error code is 4 (illegal TFTP operation) even though the error 6 (User not logged in) also applies.

Note: We will not test any error that cannot be created using a correctly implemented client and we assume that IO operations will not fail because of an internet connection or computer condition.

For example, an Unknown Opcode cannot be sent by a correctly implemented client so error 4 will not be tested but is highly recommended to implement it for debugging.

But for errors **1, 5, 6**, and **7** can be tested and we expect you to handle them and give them an error message that describes the error.

3.2.9 Disc

Packets have the following format:

type/size	2 bytes
definition	Opcode
example	0, a

Packets that appear only in a Client-to-Server communication. Informs the server of client disconnection. The client may terminate(close the program) only after receiving the ACK packet.

Parameters

- Opcode: 10.

4 Implementation Details

4.1 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, etc...) 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. **Working with little-endian will resolve in tests failing**.

We provide here the code of how to convert a short to a 2-byte and a 2-byte to a short in Java:

```
// converting short to byte array
short a = 10;
byte[] a_bytes = new byte[]{(byte) = (a >> 8), (byte) (a & 0xff)};
// converting 2 byte array to a short
byte[] b = new byte[]{0, 10}
short b_short = (short) (((short) bytes[0]) << 8 | (short) (bytes[1]))
```

4.2 General Guidelines

- The server and the client will be written in Java.
- You must use maven as your build tool for the server and client.
- The same coding standards expected in the course and previous assignments are expected here.

4.3 Server

You will have to implement a single protocol, supporting the **Thread-Per-Client** server pattern presented in class. Code seen in class for the server is included in the template. You are also provided with 3 new or changed interfaces:

- `Connections<T>`

  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):

  - `void connect(int connectionId, ConnectionHandler<T> handler);`

    add an client `connectionId` to active client map.

    Note: you can change the return value to `boolean` if you want.

  - `boolean send(int connectionId, T msg);`

    sends a message T to the client represented by the given `connectionId`.

  - `void disconnect(int connectionId);`

    Removes an active client `connectionId` from the 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 the send commands in the Connections implementation.

- `BidiMessagingProtocol<T>`

  This interface replaces the `MessagingProtocol` interface. It exists to support p2p (peer-to-peer) messaging via the Connections interface. It contains 3 functions:

  - `void start(int connectionId, Connections<String> connections);`

    Initiate the protocol with the active connections structure of the server and saves the owner client's connection id.

  - `void process(String message);`

    As in `MessagingProtocol`, processes a given message. Unlike `MessagingProtocol`, responses are sent via the connections object send functions (if needed).

- `boolean shouldTerminate();`

  true if the connection should be terminated

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. create a new class `TftpEncoderDecoder` that implement `MessageEncoderDecoder<byte[]>`.
3. Refactor the **TPC** server to support the new interfaces. The `ConnectionHandler` should implement the new interface. Add calls for the new `Connections<T>` interface.
4. Create an implementation of the `BidiMessagingProtocol` interface according to the specification in the previous sections.

You may add classes as you wish. Note that the server implementation is agnostic to the TFTP protocol implementation, and can work with different TFTP implementations, as long as they follow the rules defined by the protocol.

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.
- How do you collect a message? Are all message types collected the same way?

Tips

- It's recommended to start by implementing `TftpEncoderDecoder`.
- Use `java.io.FileInputStream` to read file to a buffer of bytes and don't read files as string.
- You can check your Server code without writing client code by just sending hard-coded packets in different java code and then waiting for returned packets and just printing them.
- You can use TFTP example client to test your server implementation when you complete the server. For instruction click here.
- You can test tasks 1–2 by fixing one of the examples in the impl folder in the supplied spl-net.zip to work with the new interfaces (easiest is the echo example)

4.4 Client

You receive a folder with echo protocol implementation and a tftp folder with the file TftpClient.java where you should implement your client.

Note: the client is a completely separate Java project and you should not import files directly from the server folder. You can copy files between them.

Tips

- starts by creating 2 threaded echo clients Where you have a keyboard thread where you send messages.
- Copy `TftpEncoderDecoder` from the server.
- Don't copy the protocol `TftpProtocol`.

4.5 Testing run commands

- Server: from server folder
  - Build using: `mvn compile`
  - Thread per client server:
    `mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.tftp.TftpServer" -Dexec.args="<port>"`
- Client: from client folder
  - Build using: `mvn compile`
  - Thread per client server:
    `mvn exec:java -Dexec.mainClass="bgu.spl.net.impl.stomp.tftp.TftpClient" -Dexec.args="<ip> <port>"`

The server directory and client directory should contain a **pom.xml** file and the src directory. Compilation will be done from the server folder using: **mvn compile** The server should be implemented in the file **"server/src/main/java/bgu/spl/net/impl/tftp/TftpServer.java"** and for the client should be implemented in the file **"client/src/main/java/bgu/spl/net/impl/tftp/TftpClient.java"**.

5 Running Examples

The following section contains examples of commands running on the 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 on the client side only. Server and client actions are explained in between.

5.1 Login and file download

Server assumptions for example:

- User named Dekel is currently not logged-in
- The server has a file named kelev_yam.mp3 the is 2.4KB long

```
< LOGRQ Dekel
(Server checks if a user named Dekel is logged-in)
> ACK 0
< RRQ kelev_yam.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 kelev_yam.mp3 complete
< DISC
(Server removes Dekel from logged-in list)
> ACK 0
```

5.2 Login and file upload

Server assumptions for example:

- User named Dekel is currently not logged-in
- The server does not have a file named kelev_yam.mp3
- The client has a file named kelev_yam.mp3 the is 2.4KB long

```
< LOGRQ Dekel
(Server checks if a user named Dekel is logged-in)
> ACK 0
< WRQ kelev_yam.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 =
kelev_yam.mp3)
> ACK 5
> WRQ kelev_yam.mp3 complete
> BCAST add kelev_yam.mp3
(Server removes Dekel from logged-in list)
> ACK 0
```

5.3 Errors, DIRQ and DELRQ

Server assumptions for example:

- User named Itzik is currently logged in
- User named Zino is currently not logged in
- The server contains 2 files:
  - The client has a file named kelev_yam.mp3 the is 2.4KB long
  - The client has a file named lemon_community_memes.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 Itzik
(Server checks if a user named Itzik is logged-in, since it is an error is sent)
> Error 7
< LOGRQ Zino
(Server checks if a user named Zino is logged-in)
> ACK 0
< WRQ kelev_yam.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 = 40, block = 1 and 40 bytes
that are "kelev_yam.mp3" + '0' + "lemon_cmmunity_memes.txt" + '\0')
(Client sends ACK 1)
> kelev_yam.mp3
> lemon_community_memes.txt
< DELRQ kelev_yam.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 =
kelev_yam.mp3)
> ACK 0
> BCAST del kelev_yam.mp3
< DISC
(Server removes Zino from logged-in list)
> ACK 0
```

6 Submission instruction

Your submission should be in a single zip file called "student1ID_student2ID.zip". The files in the zip should be set in the following structure:

- client/
  - src/
  - The pom.xml file used to build the client project.
- server/
  - src/
  - Files/ (The folder where the server store the files)
  - The pom.xml file used to build the server project.

we will use the test command from 4.5 so make sure they **work** before submission with 127.0.0.1 as ip and 7777 as port.

Note: other types of archive files, such as .rar .bz .tar.gz or anything else which is not a .zip file will not be accepted

NOTE: If your submission is not the structure your grade will be 0!!!

6.1 Grading

The grading of the assignment is as follows:

- 85% for the server.
- 15% for the client.

This means you should pay the main effort of the server implementation, and in case time is left implement the client as well.

To let you work on the server without/before implementing the client, we provide a client-ready-to-use here: https://github.com/b
rust-client