

UNIVERSITY OF SUSSEX



Computer Networks

coursework

TITLE:

**<<REPORT DESCRIBING PROTOCOLS
TCP/UDP, SOURCE CODE AN DESIGN>>**

Subject: Computer Networks

Candidate number: 252719

Table of Contents

Introduction:	4
Description for Source code, Design, and protocol	4
Building TFTP Requests	4
Brief design and structure on handling RRQ/WRQ Packet	4
Creating ACK Packets	6
Creating Error Packets	7
Creating Data Packets	8
Create Packet Unit16	8
Understanding TFTP Requests	9
Managing TFTP Requests	9
Handling WRQ Packets	9
Handling RRQ Packets	10
Handling DATA Packets	10
Server Implementation:	10
UDP Server	10
SEND:	10
File Sent:	10
ACKNOWLEDGEMENTS	10
ERROR PACKET SENDING	11
RECEIVE	11
PROCESSING INCOMING REQUESTS	11
RECEIVING FILE DATA	11
WRITING FILE	11
ACKNOWLEDGEMENTS	12
TCP Server	12
Client Implementation	12
Similarities with UDP:	12
Client Differences:	12
Server Differences:	12
Interoperability	12
Connecting to the Server via TFTP Unix Client	15
Interoperability - Receiving a File from the TFTP Server	17
Interoperability - Handling Errors	17

Conclusion	19
EXTRA WORK illustration (running script server and client).....	19
<i>I have developed a bash file script to run both server and client on both protocols TCP UDP in order to illustrate .log files when executing all the protocols and you could see clearly the execution of the whole project, also you could always run it through the script yourself but you just need to change the paths on both side JAVAHOME and class path.</i>	
	19

Introduction:

In the field of network programming, when it comes to handling data packets efficiently, it is utterly crucial for the perfect operation of communication protocols. It is quite well-known when developing applications that involve file transfer capabilities, particularly using the Trivial File Transfer Protocol (TFTP).

Description for Source code, Design, and protocol

Project Set-Up

The project features both UDP and TCP clients and servers, allowing users to input a server IP address and select a port number above 1024 to comply with TFTP standards and avoid potential issues. An auto-configure feature is included to quickly set up both the client and server, ensuring seamless connections. On the client side, files are stored in a folder named ("TFTP-UDP-Client"), while the server uses a folder called ("TFTP-UDP-Server"). Once everything is configured, users are presented with a menu offering four commands: transfer, receive, help, exit.

Implementation

To implement the protocols correctly, mainly decompose the problem to an easier and apprehensible level, starting by creating two different classes encoder and decoder for TFTP packets, and that was the way to encode and decode all data into TFTP packets. Furthermore, these two classes could be used for both folders (TFTP client and TFTP server), so the classes are TFTPRequestDecoder and TFTPRequestBuilder. On the other hand, RFC 1350 provides you with packet types of descriptions and how they are maintained and implemented.

Building TFTP Requests

The structure and the role of this class (TFTPRequestBuilder) is to encode request data into a buffer, which could be transmit via either UDP or TCP.

Brief design and structure on handling RRQ/WRQ Packet

2 bytes	string	1 byte	string	1 byte
Opcode	Filename	0	Mode	0

Responsible Method:

createPackRRQorWRQ():

that's designed to build a data packet for either a Read Request (RRQ) or Write Request (WRQ) in a network operation. It takes a storage buffer, an operation code (opcode), and a file name as inputs. The method works by sequentially adding the opcode, file name, and the word "octet" to the buffer, each followed by a null terminator to signal the end of that segment. These elements are added using helper methods like createPackString16 and createPackString16 the opcode and the word "octet" are parts of the protocol that specify the type of packet.

```
/**
 * Internal helper method to create either an RRQ or WRQ packet.
 * @param buf The buffer to store the packet data.
 * @param op The opcode indicating whether this is an RRQ or WRQ.
 * @param filename The filename involved in the request.
 * @return The total length of the created packet.
 */
private static int createPackRRQorWRQ(byte[] buf, OPCODE op, String filename) { 2 usages
    int length = 0;
    length += createPackUInt16(buf, length, op.getValue()); // Pack the opcode
    length += createPackString(buf, length, filename); // Pack the filename
    buf[length++] = 0; // Null terminator for the string
    length += createPackString(buf, length, str: "octet"); // Pack the mode
    buf[length++] = 0; // Null terminator for the mode
    return length;
}

private static int packRRQorWRQ(byte[] buf, OPCODE op, String filename)
```

```

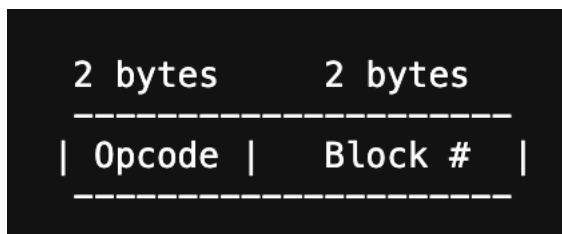
/**
 * Creates a Read Request (RRQ) packet.
 * @param buf The buffer to store the packet data.
 * @param filename The filename to request from the server.
 * @return The total length of the created RRQ packet.
 */
> public static int createPackRRQ(byte[] buf, String filename) { return createPackRRQorWRQ(buf, OPCODE.RRQ, filename); }

/**
 * Creates a Write Request (WRQ) packet.
 * @param buf The buffer to store the packet data.
 * @param filename The filename to write to the server.
 * @return The total length of the created WRQ packet.
 */
> public static int createPackWRQ(byte[] buf, String filename) { return createPackRRQorWRQ(buf, OPCODE.WRQ, filename); }

```

Creating ACK Packets

The 'createPackACK()' method makes a special kind of packet called an acknowledgment (ACK) packet, which is used to let the server know that a data packet was received correctly. It builds this ACK packet using an array of bytes, which includes the opcode (a code that tells what the packet is for), the block number (which is like a label to keep track of each packet), and a null terminator (a marker to show the end of the packet).



Responsible method:

```

/**
 * Creates an ACK packet.
 * @param buf The buffer to store the packet data.
 * @param block The block number being acknowledged.
 * @return The total length of the created ACK packet.
 */
public static int createPackAck(byte[] buf, int block) { no usages
    int length = 0;
    length += createPackUInt16(buf, length, OPCODE.ACK.getValue()); // Pack the ACK opcode
    length += createPackUInt16(buf, length, block); // Pack the block number
    length += createPackUInt16(buf, length, value: 0); // Zero padding for alignment
    return length;
}

```

Creating Error Packets

2 bytes	2 bytes	string	1 byte
Opcode	ErrorCode	ErrMsg	0

Responsible method:

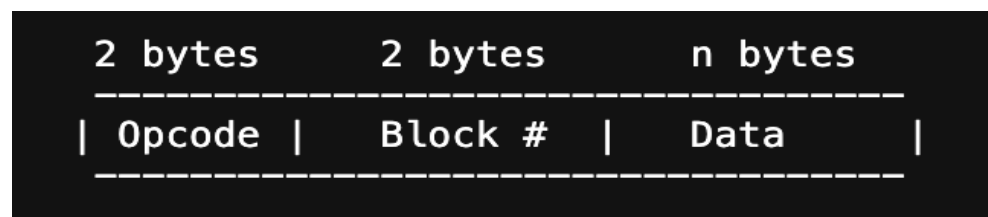
```

/**
 * Creates an ERROR packet.
 * @param buf The buffer to store the packet data.
 * @param errorCode The error code.
 * @param errorMessage The error message.
 * @return The total length of the created ERROR packet.
 */
public static int createPackError(byte[] buf, int errorCode, String errorMessage) { no usages
    int length = 0;
    length += createPackUInt16(buf, length, OPCODE.ERROR.getValue()); // Pack the ERROR opcode
    length += createPackUInt16(buf, length, errorCode); // Pack the error code
    length += createPackString(buf, length, errorMessage); // Pack the error message
    buf[length++] = 0; // Null terminator for the string
    return length;
}

```

Creating Data Packets

The 'createPackData()' method is a piece of code that makes a data packet to send information to a server. It puts together a bunch of bytes that stand for different parts of the data packet, like the opcode (which is a kind of code that says what the packet is supposed to do), the block number (which keeps track of the packets), and the actual data itself. It also chops up the data from a file into smaller.



Responsible method:

```
/**
 * Creates a DATA packet.
 * @param buf The buffer to store the packet data.
 * @param block The block number of the data.
 * @param data The actual data bytes.
 * @return The total length of the created DATA packet.
 */
public static int createPackData(byte[] buf, int block, byte[] data) { no usages
    assert data.length <= MAX_BYTES; // Ensure data does not exceed max bytes
    int length = 0;
    length += createPackUInt16(buf, length, OPCode.DATA.getValue()); // Pack the DATA opcode
    length += createPackUInt16(buf, length, block); // Pack the block number
    System.arraycopy(data, srcPos: 0, buf, length, data.length); // Copy the actual data
    length += data.length;
    return length;
}
```

Create Packet Unit16

Without Forgetting the other method helper (createPackUnit):

This function has the functionality of build a 16-bit integer into a buffer at the specified offset.

Responsible Method:

```
/**
 * Packs a 16-bit integer into the buffer at the specified offset.
 * @param buf The buffer where the integer is to be packed.
 * @param offset The offset within the buffer where packing starts.
 * @param value The integer value to pack.
 * @return The number of bytes used in the buffer.
 */
public static int createPackUInt16(byte[] buf, int offset, int value) { 8 usages
    buf[offset] = (byte) (value >> 8); // High byte
    buf[offset + 1] = (byte) (value); // Low byte
    return 2;
}
```

Understanding TFTP Requests

To make the communication between our server and client, it's quite critical that we can interpret the requests sent between them. This task is handled by the TFTPRequestDecoder class.

The decoder efficiently processes data packets, which comes in some form of byte arrays, and translates them into a format that's easier to understand. For instance, the method `decodeWRQorRRQ` extracts information from the packets and presents it as a `WrqOrRrqPacket`. Similarly, when decoding a data packet, the output is an instance of the `DataPacket` class.

Managing TFTP Requests

By utilising the decoding and encoding methods outlined earlier for TFTP requests, we can now smoothly manage the packaging and unpacking of any data or requests that need to be exchanged between the client and the server.

Furthermore, in a TFTP Server, each `TFTPRequestHandler` is linked to a client by IP Address, keeping track of client details and using a `DataPacketsBuilder` to store received data packets. Similarly, the TFTP Client's class also maintains a `DataPacketsBuilder`.

Handling WRQ Packets

First off, when a Write Request (WRQ) is received by the server or client, the first step is to unpack the filename from the request. This filename is then assigned to the `DataPacketsBuilder` to inform the server where the incoming data packets should be stored after they are received. In response, an acknowledgment (ACK) packet with block number 0 is sent to confirm the request and to indicate readiness to receive data packets.

Handling RRQ Packets

For Read Requests (RRQ), the server or client first checks its directory for the requested file. If the file is not there, an error packet gets triggered. If the file is found, the server or client starts sending data packets, each up to 512 bytes in size (excluding headers). After each data packet is sent, an ACK packet is expected in return to acknowledge the successful receipt of the data.

Handling DATA Packets

Upon receiving a DATA packet, the contents are extracted and added to the `DataPacketsBuilder` using the appropriate method. If the data contained in a packet is less than 512 bytes, this would tell us the end of the file transfer, and then prompting the system to save the file to the current directory.

Server Implementation:

UDP Server

The UDP server operates using the `DatagramSocket` class for sending and receiving data over a socket. It manages multiple requests by maintaining a `HashMap` data structure where each request handler is mapped with a client's IP address and port. Plus, when a new request is received from an unknown client, a new request handler is instantiated and used to process the packet.

SEND:

File Sent:

When you want to send a file to the client, you use a method called `'sendFileToClient'`. It kicks off when the client asks for a specific file. This method figures out which file the client wants by looking at the name in the request packet, and then it finds where the file is stored to get it ready for sending. The file doesn't go all at once; instead, it's broken up into smaller parts, each no bigger than 512 bytes.

ACKNOWLEDGEMENTS

There is method send the initial ACK packet to the client with a block number of 0. It uses the 'Create' class to make the ACK packet which, is sent, to the client using a 'DatagramPacket'. The method works in the same way but sends the block number sequentially greater than 0. This means that it would start from 1 and then increment by 1 every time it sends an ACK.

ERROR PACKET SENDING

If something goes wrong, there's a method that takes charge. It sends a message to the client that's basically saying, "Oops, we hit a snag," with details about what went wrong. This message, called an error packet, includes error codes and messages to let the client know exactly what the issue is. Just like with the other messages.

RECEIVE

There's a class in the system called 'Receive' and its job is to deal with data or requests that come in from clients.

PROCESSING INCOMING REQUESTS

it uses a Byte Buffer and sets it to be big enough for the data plus a little extra for the header part, which is just 4 bytes more. When a packet arrives, the method checks its opcode either read (RRQ) or write (WRQ) a file. If it's WRQ, it starts up receive file from client to get the file from the client. If it's RRQ, it sends the file that the client wants.

RECEIVING FILE DATA

when a client wants to write a file to the server, it first checks the packet to get the file name out of it. It knows the file name is done when it hits a null byte, which is like a stop sign. After that, it sends back an acknowledgment to the client. All this data gets written down in a new file on the server.

WRITING FILE

For writing the data into a file. This one use 'Datagram Packets' to grab chunks of the file data as they come in. A Byte Buffer helps it keep track of the order by looking at the block numbers. Then it takes the data and writes it into the server's storage area using 'File Output Stream'. It sends an acknowledgment after getting each chunk to confirm that it's been received.

ACKNOWLEDGEMENTS

Lastly, 'acknowledgeAck' is the method that waits for acknowledgments (ACKs) from the client. It uses 'Datagram Packet' to catch these ACKs. Inside, it peeks at the block number with the help of a Byte Buffer to make sure it's getting the right responses in the right order.

TCP Server

The TCP server utilises a client socket to handle incoming requests, operating on port 8888. This server continuously listens for connections and processes requests through a thread.

Client Implementation

The clients feature a command-line interface implemented in the Command class, supplying commands like 'transmit', 'receive', 'help', and 'exit'. Actual network requests are handled by the Command class, making the interface user-friendly and straight forward for typical file transfer tasks.

Similarities with UDP:

The TCP client creates WRQ, RRQ, and DATA packets the same way as UDP and sends data in numbered chunks.

Client Differences:

TCP clients send data as a continuous stream through 'Data Output Stream', which automatically handles packet ordering.

Server Differences:

TCP servers use 'Server Socket' for a steady connection, managing sessions directly with the client through data streams, ensuring ordered and error-checked data transmission without the need for manual packet management like UDP.

Interoperability

To verify that the UDP version of our TFTP Server follows the protocol outlined in RFC 1350 correctly, using the built-in Unix TFTP client was ideal for demonstration, tftp command. To further explaining that the server functions properly with a third-party client, I conducted tests by transferring the entire text of the book "a0.txt" as a text file back and forth between the server and the client.

The screenshot shows an IDE with a Java project named "TFTP-UDP-Server". The project structure includes a "src" directory with "main" and "exceptions" subdirectories. The "main" directory contains a "request" subdirectory. The "exceptions" directory contains a "request" subdirectory. The "request" subdirectory contains a "DataPacketBuilder" class. The "DataPacketBuilder" class is the source of the log output.

The log output shows a sequence of requests and responses between a client and a server. The log is displayed in a console window at the bottom of the IDE. The log entries are as follows:

```

TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received WQ - Filename: a3.txt
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 0
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 1 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 1
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 2 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 2
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 3 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 3
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 4 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 4
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 5 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 5
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 6 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 6
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 7 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 7
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 8 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 8
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 9 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 9
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 10 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 10
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 11 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 11
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 12 of size 508 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 12
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received DATA for a3.txt - Block no 13 of size 64 bytes
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent ACK Block 13
TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received all data from a3.txt
  
```

The log output is displayed in a console window at the bottom of the IDE. The console window shows the log output of the "TFTP-UDP-Server" application. The log output is as follows:

```

Last login: Fri Apr 26 18:48:13 on console
tftp (tftp)
tftp> put a3.txt
Sent 46904 bytes during 0.1 seconds in 92 blocks
tftp>
  
```

Figure 1 shows file transition Using Unix and gets logged in file.log


```
Project
├── TFTP-UDP-Server - [Documents/NetCoursework/TFTP-UDP-Server]
│   ├── src
│   │   ├── main
│   │   │   └── java
│   │   │       ├── exceptions
│   │   │       └── request
│   │           ├── DataPacketBuilder
│   │           ├── OPCode
│   │           ├── RequestHandlerLogger
│   │           ├── TFTPRequestBuilder
│   │           ├── TFTPRequestDecoder
│   │           └── TFTPRequestHandler
│   ├── logs
│   └── server.log
└── runServer.sh
```

```
Server
TFTPRequestHandler.java
DataPacketBuilder.java
TFTPRequestDecoder.java
runServer.sh
server.log
```

```
794 TFTPRequestHandler.java
795 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 2/13 for a3.txt - Block size: 508 bytes
796 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 2
797 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 3/13 for a3.txt - Block size: 508 bytes
798 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 3
799 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 4/13 for a3.txt - Block size: 508 bytes
800 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 4
801 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 5/13 for a3.txt - Block size: 508 bytes
802 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 5
803 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 6/13 for a3.txt - Block size: 508 bytes
804 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 6
805 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 7/13 for a3.txt - Block size: 508 bytes
806 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 7
807 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 8/13 for a3.txt - Block size: 508 bytes
808 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 8
809 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 9/13 for a3.txt - Block size: 508 bytes
810 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 9
811 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 10/13 for a3.txt - Block size: 508 bytes
812 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 10
813 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 11/13 for a3.txt - Block size: 508 bytes
814 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 11
815 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 12/13 for a3.txt - Block size: 508 bytes
816 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 12
817 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent DATA block 13/13 for a3.txt - Block size: 64 bytes
818 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Received ACK Block 13
819 TFTP_REQUEST_HANDLER - 127.0.0.1:54371: Sent all data from a3.txt
```

```
resources
a0.txt 26/04/2024, 16:58, 46.9 KB
a00.txt 25/04/2024, 18:27, 46.9 KB
a3.txt 26/04/2024, 15:01, 6.16 KB
a33.txt 25/04/2024, 18:34, 6.16 KB
runServer.sh 26/04/2024, 16:10, 1.28 KB 33 minutes ago
```

```
Server
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Received DATA for a0.txt - Block no 91 of size 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Sent ACK Block 91
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Received DATA for a0.txt - Block no 92 of size 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Sent ACK Block 92
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Received all data from a0.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Saved file to a0.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Received RRR - Filename: a0.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Sent DATA block 1/93 for a0.txt - Block size: 508 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Received ACK Block 1
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:65407: Sent DATA block 2/93 for a0.txt - Block size: 508 bytes
```

```
381:1 TFTP-UDP-Server Material Deep Ocean AWS: No credentials selected LF UTF-8 4 spaces INSERT 1568 of 4800
```

```
ftp (ftp)
Last login: Fri Apr 26 18:48:13 on console
~ - Documents
~ - Documents cd NetCoursework
~/Doc/NetCoursework
TFTP-UDP-Client TFTP-UDP-Client a0.txt
TFTP-UDP-Server TFTP-UDP-Server a3.txt
~/Doc/NetCoursework/TFTP-UDP-Client - US
client.log pom.xml target
a3.txt
a0.txt
logs
~/Doc/NetCoursework/TFTP-UDP-Client - ftp
ftp> connect localhost 8888
ftp> mode octet
ftp> put a0.txt
Sent 46904 bytes during 0.1 seconds in 92 blocks
ftp> get a0.txt
Received 508 bytes during 0.0 seconds in 1 blocks
ftp>
```

Figure 2 shows getting file from server in octet mode Using Unix as third party.

The screenshot shows a terminal window titled 'tftp (tftp)' with a dark background. The terminal displays the following commands and output:

```
Last login: Fri Apr 26 10:48:13 on console
~ ➔ cd Documents
~/Documents ➔ cd NetCoursework
~/Doc/NetCoursework ➔ ls
TFTP-TCP-Client TFTP-UDP-Client a0.txt
TFTP-TCP-Server TFTP-UDP-Server a3.txt
~/Doc/NetCoursework ➔ cd TFTP-UDP-Client
~/Doc/NetCoursework/TFTP-UDP-Client ➔ LS
a0.txt      client.log  pom.xml    target
a3.txt      logs       src
~/Doc/NetCoursework/TFTP-UDP-Client ➔ tftp

tftp> connect localhost 8888
tftp> mode octet
tftp> put a0.txt
Sent 46904 bytes during 0.1 seconds in 92 blocks
tftp> get a0.txt
Received 508 bytes during 0.0 seconds in 1 blocks
tftp>
```

On the right side of the terminal, there is a vertical list of timestamps with green checkmarks and a clock icon, indicating packet capture events:

- ✓ 04:42:05 pm
- ✓ 04:42:13 pm
- ✓ 04:42:17 pm
- ✓ 04:42:19 pm
- ✓ 04:42:27 pm
- ✓ 04:42:28 pm

At the bottom of the terminal window, there is a status bar showing a battery icon at 5%, a network icon, and the date/time '26/4, 4:51 PM'.

Figure 3 shows starting TFTP client, and connecting to a server on localhost 8888, set on octet mode, and turning on packet tracing, the number of the blocks match which is 92 on both windows (92 ACK / 92 blocks).

Connecting to the Server via TFTP Unix Client

Our TFTP Server's UDP version operates on the local system using port 8888. To connect, you first need to open the tftp command line tool. Once it's open, use the command 'connect localhost:8888' to establish a connection. In the screenshots included here, I've turned on binary mode to display the packets being sent and received by the client smoothly and respectively. Additionally, it's important to set the client to binary mode to handle the data correctly when you the size of the data is a whole book.

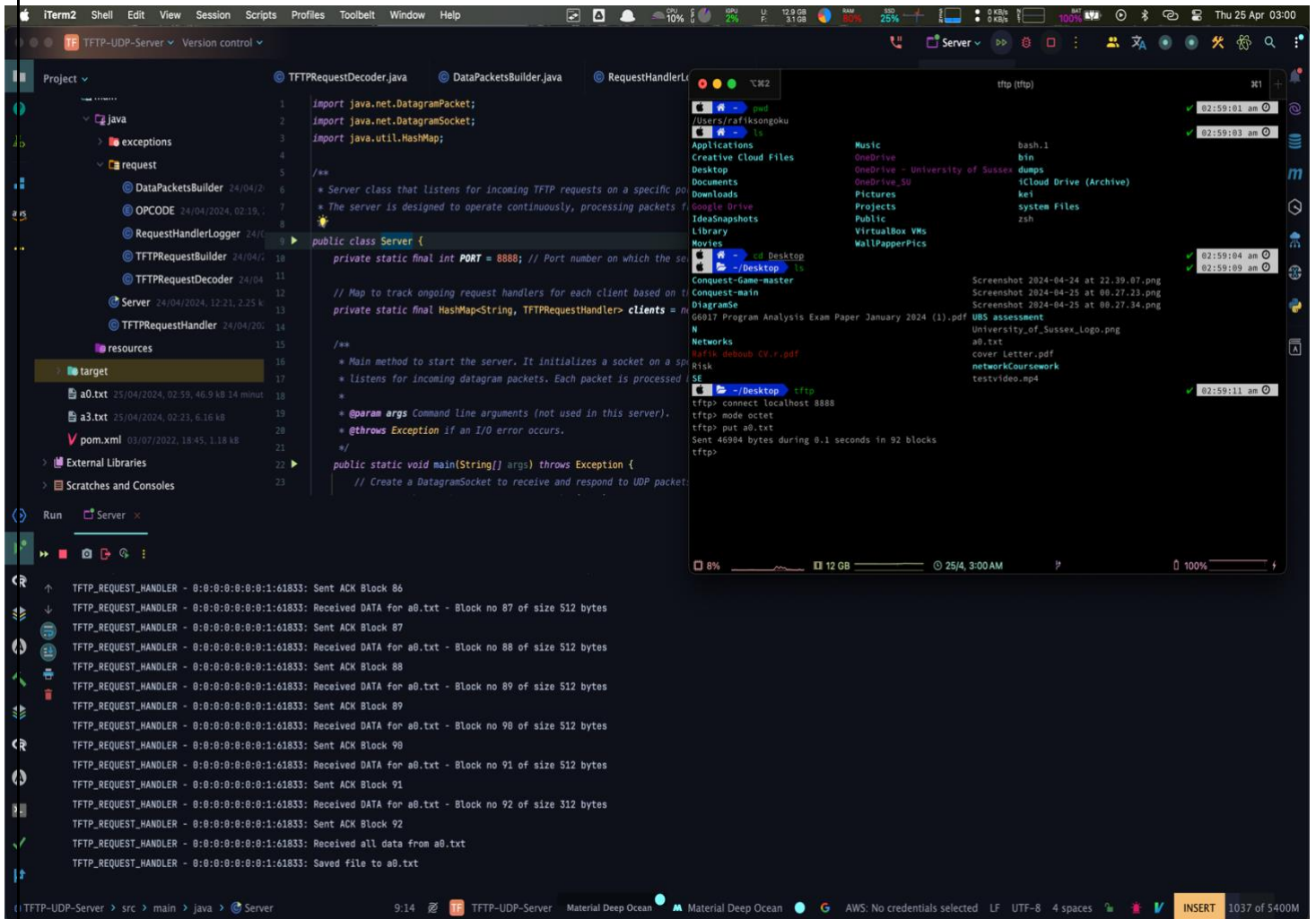


Figure 4 Shows the transmitting of file with 92 blocks.

Interoperability - Sending a File to the Server over TFTP with the Unix Client

To send a file from the client to the server, we use the `<put>` command from within the TFTP utility.

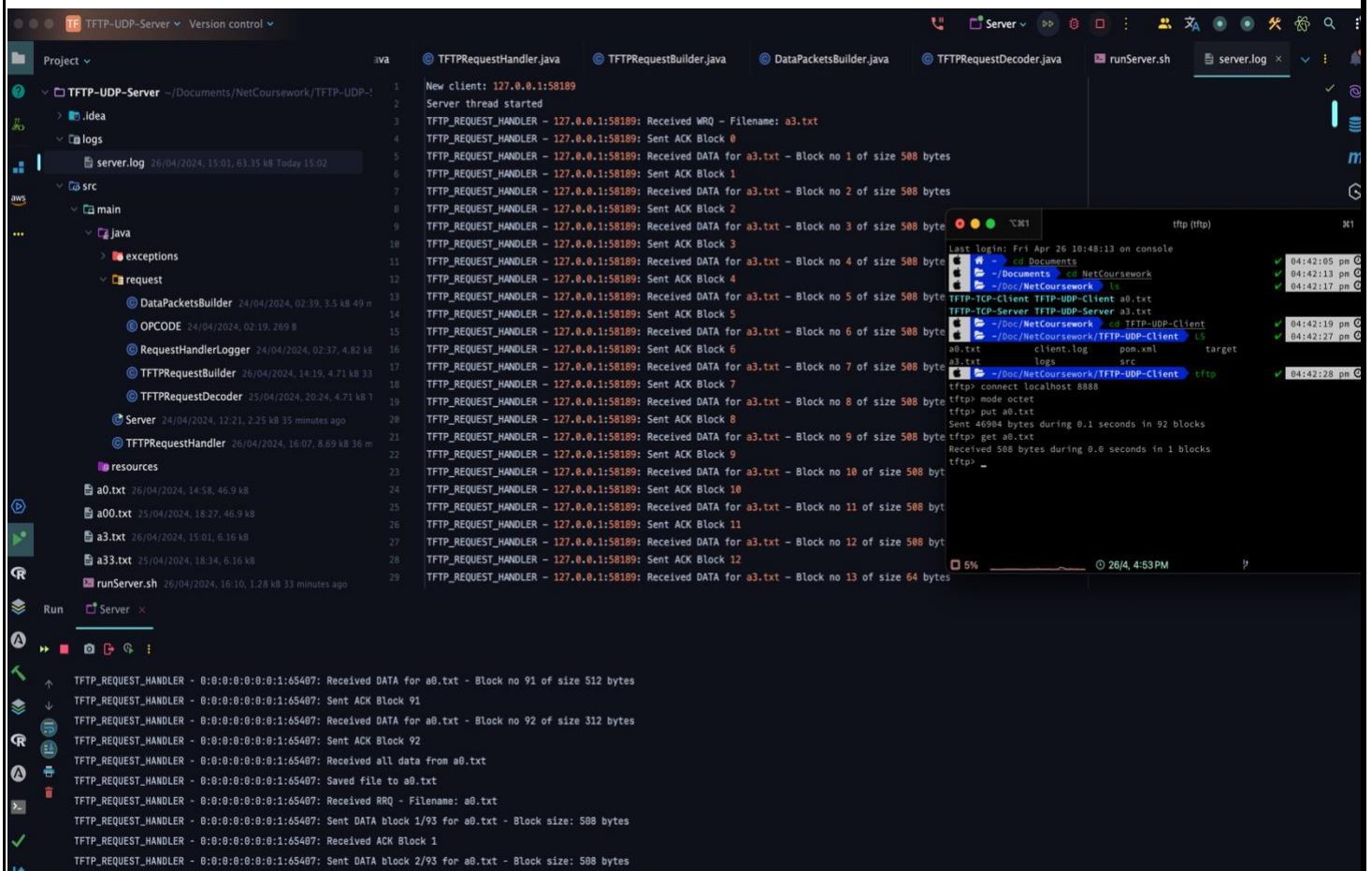


Figure 5 Newly created file a0.txt.

Interoperability - Receiving a File from the TFTP Server

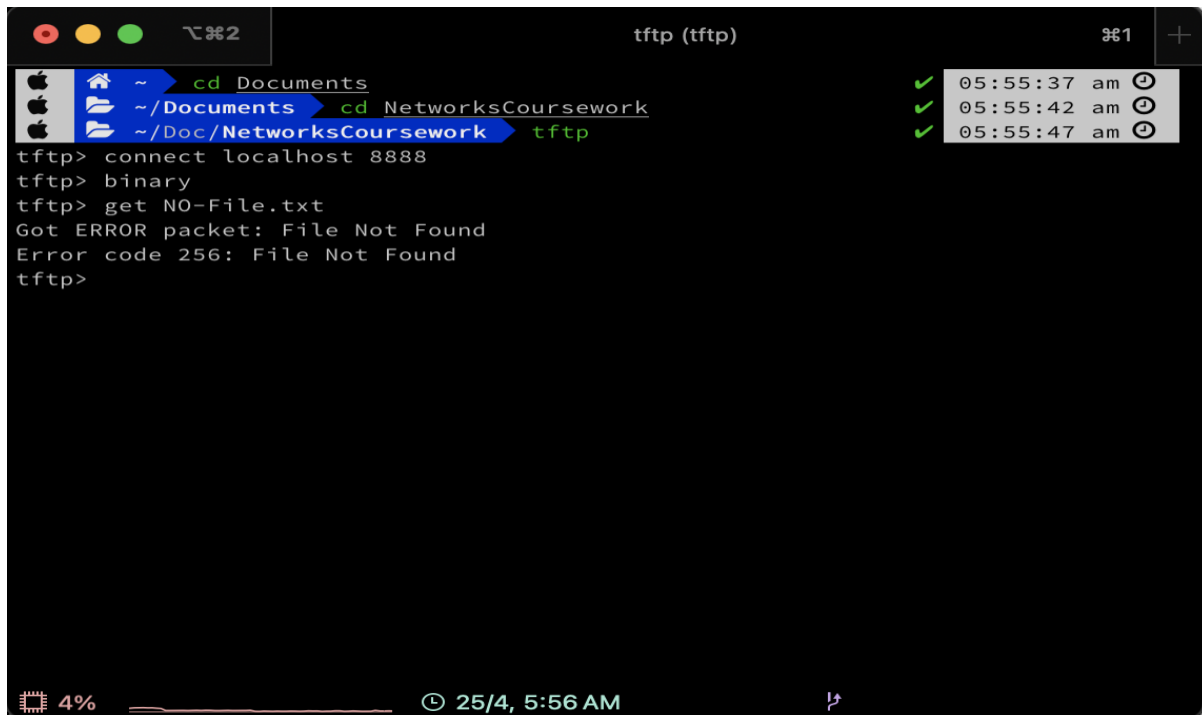
To show how to retrieve a file from the server, we will first remove the original **a0.txt** from the root of the Coursework folder.

Now, within the same session where we sent the file, we can use the `<get>` command to download the file back from the server.

Figure 6 Shows the files have been transferred back.

Interoperability - Handling Errors

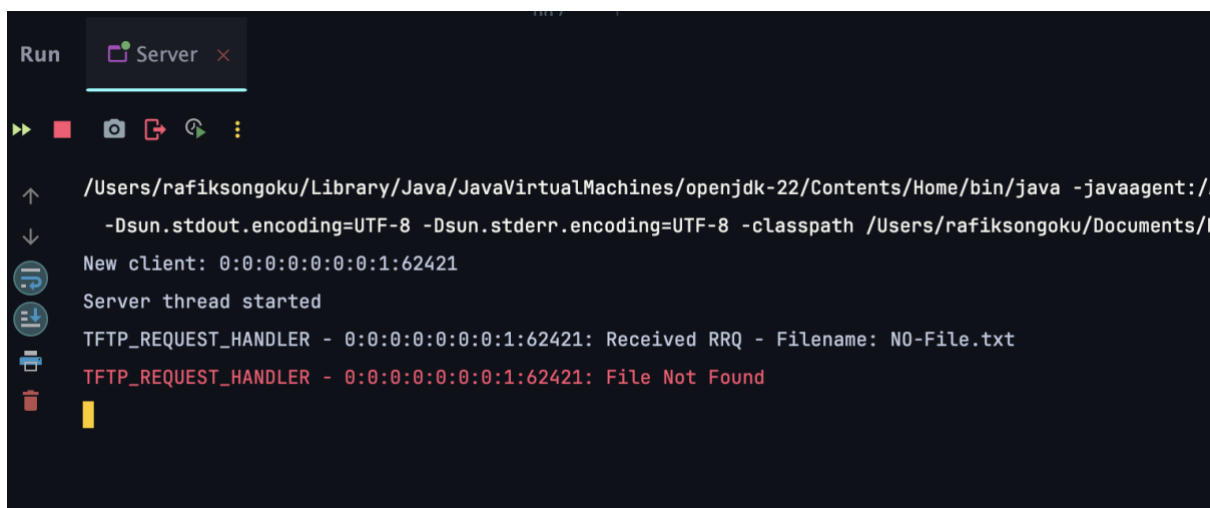
Our TFTP Server is designed to manage errors effectively, especially in cases where a file requested for reading is not found. To verify this functionality:



The screenshot shows a terminal window titled "tftp (tftp)". The user has navigated through the file system: `cd Documents`, `cd NetworksCoursework`, and then `tftp`. The user connects to `localhost 8888`, sets the mode to `binary`, and attempts to get `NO-File.txt`. The terminal displays the error: `Got ERROR packet: File Not Found` and `Error code 256: File Not Found`. A sidebar on the right shows a list of files with green checkmarks and timestamps: `05:55:37 am`, `05:55:42 am`, and `05:55:47 am`. The bottom status bar shows `4%` battery and the time `25/4, 5:56 AM`.

```
tftp (tftp)
~
cd Documents
~/Documents
cd NetworksCoursework
~/Doc/NetworksCoursework
tftp
tftp> connect localhost 8888
tftp> binary
tftp> get NO-File.txt
Got ERROR packet: File Not Found
Error code 256: File Not Found
tftp>
```

Figure 7 Error packet shows when a file does not exist. (TFTP Client)



The screenshot shows a Java server output window titled "Server". The output displays the following messages: `New client: 0:0:0:0:0:0:1:62421`, `Server thread started`, `TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:1:62421: Received RRQ - Filename: NO-File.txt`, and `TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:1:62421: File Not Found`. The window also shows a "Run" button and a "Server" tab.

```
Run
Server
New client: 0:0:0:0:0:0:1:62421
Server thread started
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:1:62421: Received RRQ - Filename: NO-File.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:1:62421: File Not Found
```

Figure 8 Server output when a file no found.

Conclusion

Each of these methods has its strengths and weaknesses. UDP is faster but less reliable, while TCP is slower but makes sure everything gets to where it needs to go safely. This shows how TFTP can be adjusted to work well in different types of network situations, giving users the choice to pick what works best for their needs.

EXTRA WORK illustration (running script server and client)

I have developed a bash file script to run both server and client on both protocols TCP / UDP in order to illustrate .log files when executing all the protocols and you could see clearly the execution of the whole project, also you could always run it through the script yourself, but you just need to change the paths on both side JAVAHOME and class path.