Computer Networks Coursework - 231006

Source Code Design and Protocol Description

To implement the protocol correctly, I begun by creating an encoder and a decoder for the TFTP packets, this way I was able to reliably encode and decode information into TFTP packets. These files could be used by both my TFTP client and my TFTP server and helped reduce repetition. The two classes responsible for this are `TFTPRequestDecoder` and `TFTPRequestBuilder`. This is where the packet types as described in RFC 1350 are actually implemented.

Building TFTP Requests

The `TFTPRequestBuilder` class is responsible for encoding request data into a buffer, which can then be sent via either UDP or TCP.

Creating RRQ and WRQ Packets

```
// RRQ/WRQ packet
//
//  2 bytes     string    1 byte    string    1 byte
// ------------------------------------------------
// | Opcode |  Filename  |   0  |    Mode    |   0  |
// ------------------------------------------------
```

The `RRQ` and `WRQ` packets have relatively similar implementation and so we use a helper method which can perform both:

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

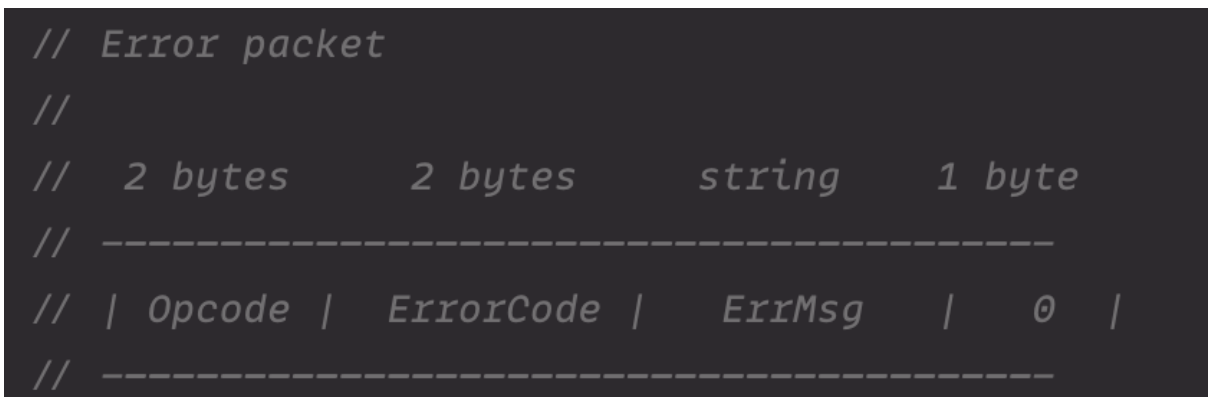
Creating DATA Packets


```
public static int packData(byte[] buf, int block, byte[] data)
```

Creating ACK Packets


```
public static int packAck(byte[] buf, int block)
```

Creating Error Packets



```
public static int packError(byte[] buf, int errorCode, String errorMessage)
```

Decoding TFTP Requests

In order for our server and client to interact, we also must be able to decode the requests our server/client sends. This is the responsibility of the `TFTPRequestDecoder` class.

Our request decoder is able to unpack data packets that are received as `byte[]` and returns the data needed for that packet in a more understandable form. For example, the `unpackWRQorRRQ` method returns a `WrqOrRrqPacket`. Whilst a data packet is decoded into an instance of the `DataPacket` class.

Handling TFTP Requests

Using the decoding and encoding techniques described above for TFTP requests, we now can easily unpack or pack and data or request we may need to send or receive from either the client or the server.

> Note: Each `RequestHander` for TFTP Server is assigned to a client based on its IP Address and stores information about the current client and the `DataPacketsBuilder` which is used to store data packets received. On the TFTP Client, the `Client` class stores the `DataPacketsBuilder`

Handling WRQ Packets

When an `WRQ` is sent to the server or the client, the recipient begins by unpacking the filename of the request and assigning it to the filename of the `DataPacketsBuilder` so the server knows where to save the data packets once the `DATA` blocks have been received. The recipient then sends back an `ACK` block with a block number 0 to

acknowledge the request and tell the sender that it is ready for `DATA` packets to be sent.

Handling RRQ Packets

When an `RRQ` request is sent to the server or the client, the responder first attempts to find that file in its directory, if that file is not found, an `ERROR` packet is sent to the sender of the request.

If the file does exist, the recipient of the request will immediately begin sending the data packets each of maximum size of 512 bytes (excluding headers). After each `DATA` block is sent, the request recipient will be expected to send an `ACK` packet acknowledging the receipt of the DATA packet.

Handling DATA Packets

When a `DATA` packet is received, the data is unpacked from the packet and is added to the `DataPacketsBuilder` instance on the request handler's class using the `addDataPacket` method. If the size of the unpacked data in the packet is less than 512 bytes, then this tells us we have reached the end of the transfer and should save the file to the current directory.

Server Implementation

UDP Server

The UDP version of the server uses the `DatagramSocket` class to receive and send data over a socket. It runs concurrently by storing request handlers in a Hash-map with a key to denote the IP address and port of the client. When a request is received, the server checks if the client has an existing `TFTPRequestHandler` in the Hash-map and if not, they create one. Using the request handler, we then run the `handle` method on it, passing it the packet.

TCP Server

The TCP version of the TFTP server uses a `ServerSocket` to accept requests from a client socket. The clients request is then handled by a `TFTPRequestHandler` thread. This server listens for connections on port 8888.

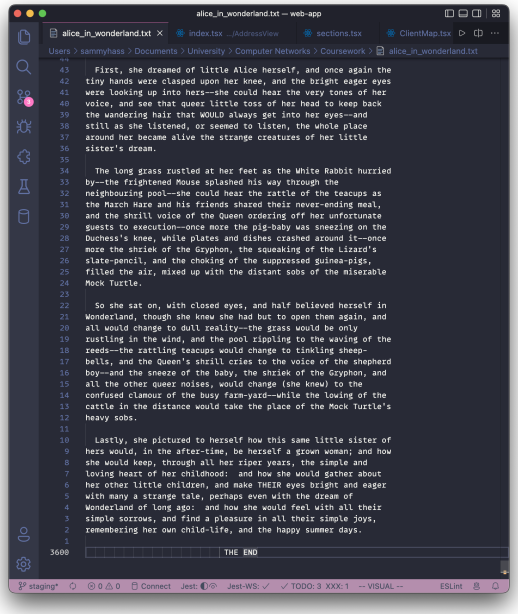
Client Implementation

Both clients I have implemented have a simple command line interface with four commands: `upload`, `download`, `help` and `exit`. The command line interface is implemented in the `Cmd` class, whilst the actual requests are sent from the `TFTPClient` class.

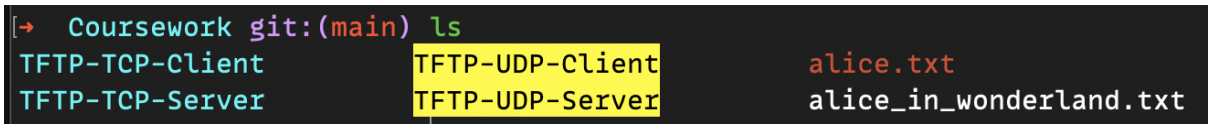
Interoperability

To show that the UDP version of the TFTP Server implements correctly the protocol as described in RFC 1350, I opted to use the built in unix TFTP client - `tftp` . To demonstrate that the server works correctly with a third-party client, I chose to transfer back and forth between the server and client the entire text of the book *Alice in Wonderland* as a text file.

The start of the file used for testing the TFTP server.



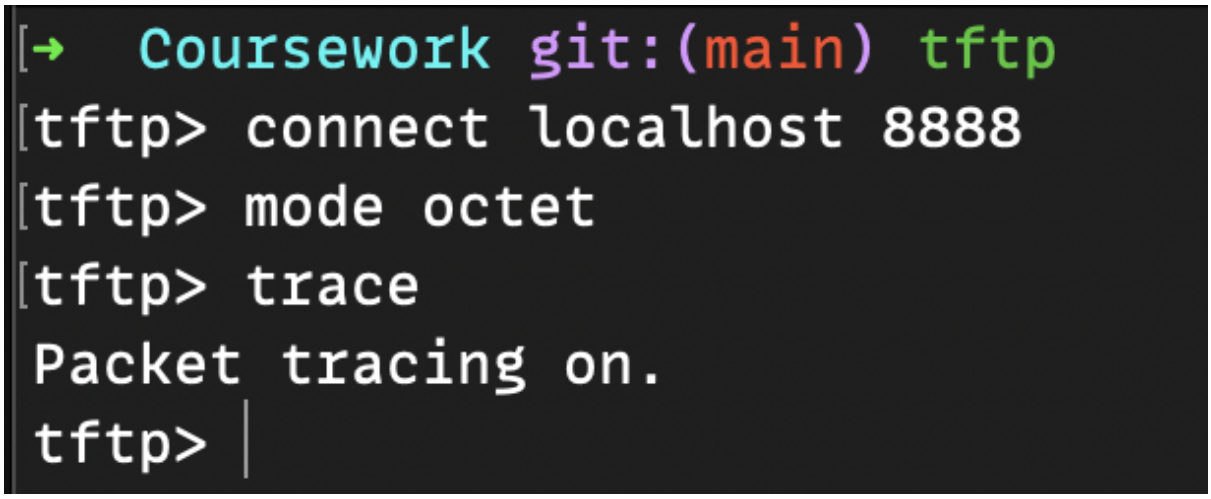
End of the file used for testing the TFTP server, note, the file has 3600 lines exactly.



Directory listing for the coursework root, note, the file `alice_in_wonderland.txt` which contains the entire of the book Alice in Wonderland, as described above.

Connecting to the Server via TFTP Unix Client

The UDP version of the TFTP Server runs locally on port 8888. To connect to the server using the command line client, we must first launch the `tftp` command line tool and use the connect command to connect to `localhost:8888`. In the screenshots below, I also enable trace mode so that we can see the packets received and sent by the client, in addition we must enable `octet` mode on the client side.



Starting up the `tftp` client, and connecting to a server on `localhost:8888`, enabling octet mode and turning on packet tracing.

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.

```
tftp> put alice_in_wonderland.txt
sent WRQ <file=alice_in_wonderland.txt, mode=octet>
received ACK <block=0>
sent DATA <block=1, 512 bytes>
received ACK <block=1>
sent DATA <block=2, 512 bytes>
received ACK <block=2>
sent DATA <block=3, 512 bytes>
received ACK <block=3>
sent DATA <block=4, 512 bytes>
received ACK <block=4>
```

Output from `tftp` unix client when sending a write request. As we can see, the WRQ request is sent, and an ACK is received with block no. 0. Then the client starts sending data, waiting for an ACK packet after each block is sent.

```
sent DATA <block=291, 94 bytes>
received ACK <block=291>
Sent 148574 bytes in 0.1 seconds
```

Final line of output from the `tftp` unix client when uploading a file to the server.

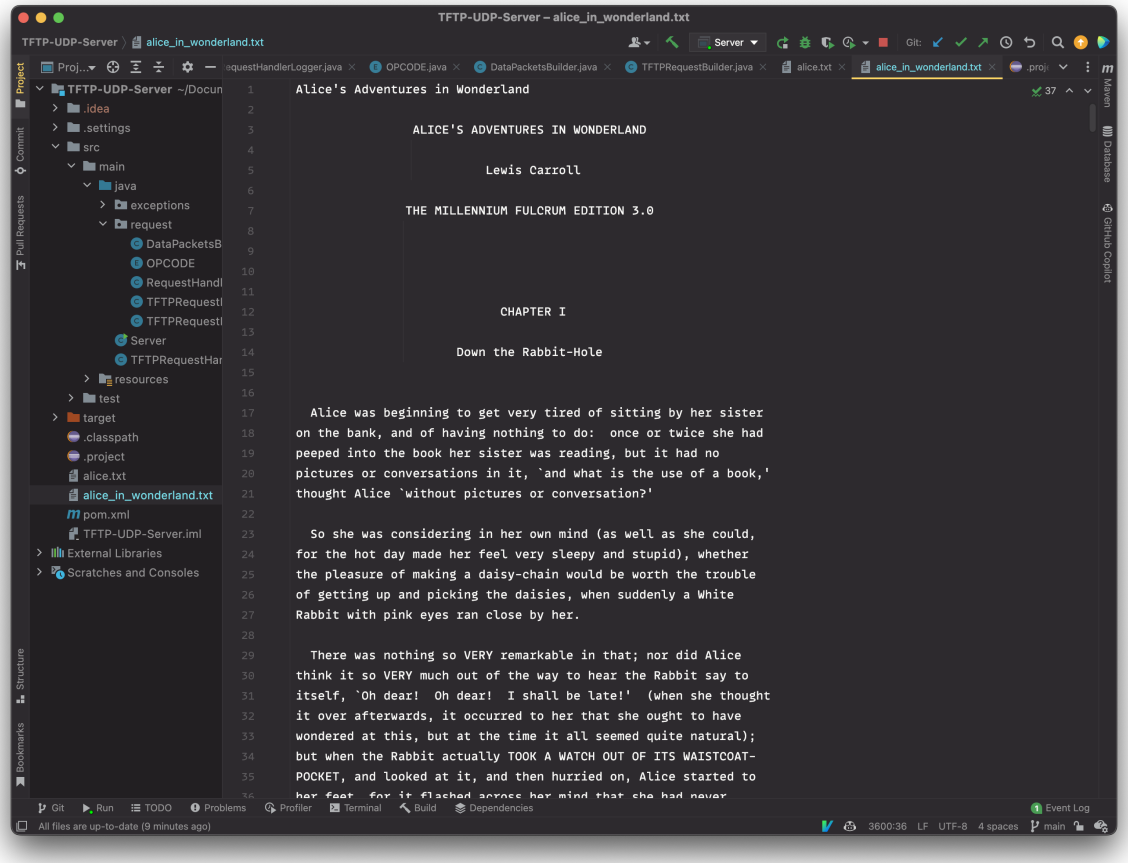
Upon examining the output of the server, we can see that the server received the request replied to the client as expected.

```
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received WRQ - Filename: alice_in_wonderland.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 0
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 1
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 1
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 2
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 2
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 3
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 3
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 4
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 4
```

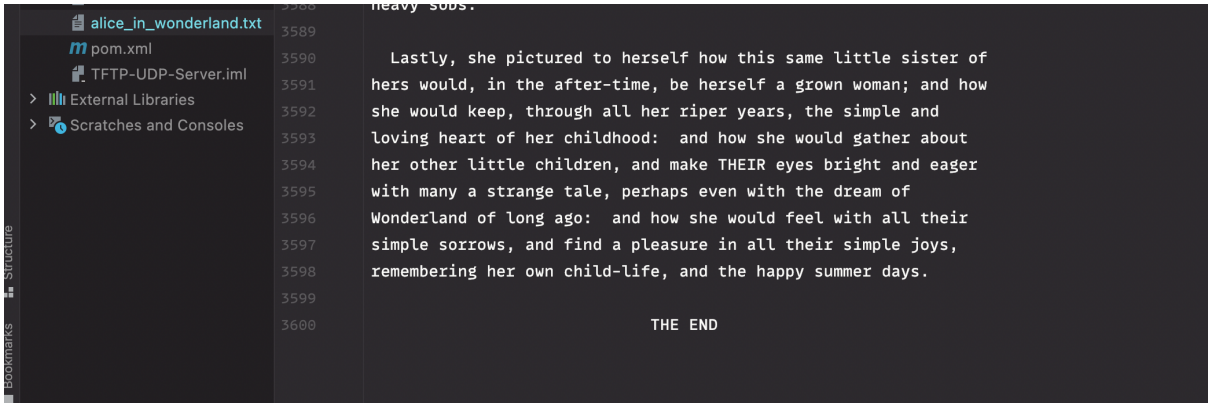
```
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 290 of size 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 290
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received DATA for alice_in_wonderland.txt - Block no 291 of size 94 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent ACK Block 291
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received all data from alice_in_wonderland.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Saved file to alice_in_wonderland.txt
```

End of the output from the server. Showing that the number of blocks was identical as was the size of the final packet (meaning that no data was lost)

The server has now saved the file to its own directory, `/TFTP-UDP-Server`. We can inspect this file and see that it has been transferred correctly with the same number of lines as the original (3600)



Newly created `alice_in_wonderland.txt` file



The end of the file, displaying the same number of lines as the original.

Interoperability - Receiving a File from the TFTP Server

In order to demonstrate retrieving a file from the server, we will first delete the original `alice_in_wonderland.txt` from the root of the `Coursework` folder. This way we can guarantee that a new file is saved when the request is sent.

```
→  Coursework git:(main) ✗ rm alice_in_wonderland.txt
→  Coursework git:(main) ✗ ls
TFTP-TCP-Client TFTP-TCP-Server TFTP-UDP-Client TFTP-UDP-Server alice.txt
→  Coursework git:(main) ✗
```

Deleting the original `alice_in_wonderland.txt` file from the coursework folder.

Now from within the same session as we sent the file, we can use the `get` command to retrieve the file from the server.

```
tftp> get alice_in_wonderland.txt
sent RRQ <file=alice_in_wonderland.txt, mode=octet>
received DATA <block=1, 512 bytes>
sent ACK <block=1>
received DATA <block=2, 512 bytes>
sent ACK <block=2>
received DATA <block=3, 512 bytes>
sent ACK <block=3>
received DATA <block=4, 512 bytes>
sent ACK <block=4>
received DATA <block=5, 512 bytes>
sent ACK <block=5>
received DATA <block=6, 512 bytes>
```

```
received DATA <block=288, 512 bytes>
sent ACK <block=288>
received DATA <block=289, 512 bytes>
sent ACK <block=289>
received DATA <block=290, 512 bytes>
sent ACK <block=290>
received DATA <block=291, 94 bytes>
Received 148574 bytes in 0.1 seconds
tftp>
```

Output from the unix `tftp` client when sending a get request for `alice_in_wonderland.txt`. Note end of output with final data block number 291 of size 94 bytes, (the same size as the file was when we sent it).

Upon examining the output from our server, we can see the request was fulfilled correctly, the server begins sending DATA blocks immediately and waited for the client to sent an ACK block after each DATA block.

```
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received RRQ - Filename: alice_in_wonderland.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 1/291 for alice_in_wonderland.txt - Block size: 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 1
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 2/291 for alice_in_wonderland.txt - Block size: 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 2
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 3/291 for alice_in_wonderland.txt - Block size: 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 3
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 4/291 for alice_in_wonderland.txt - Block size: 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 4
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 5/291 for alice_in_wonderland.txt - Block size: 512 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 5
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 6/291 for alice_in_wonderland.txt - Block size: 512 bytes
```

```
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent DATA block 291/291 for alice_in_wonderland.txt - Block size: 94 bytes
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Received ACK Block 291
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:49983: Sent all data from alice_in_wonderland.txt
```

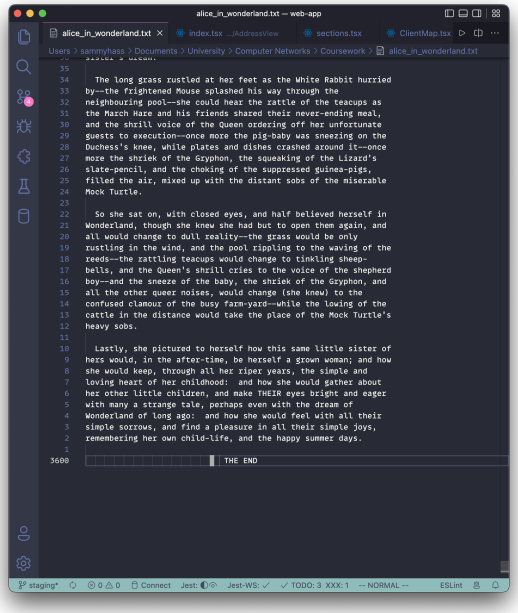
As can be seen above, our server sent the file correctly to the client.

We can now look at the root directory for the coursework again, the current working directory for the `tftp` utility and find the `alice_in_wonderland.txt` file as it was originally.

```
➜  Coursework git:(main) ✗ ls
TFTP-TCP-Client         TFTP-UDP-Client         alice.txt
TFTP-TCP-Server         TFTP-UDP-Server         alice_in_wonderland.txt
```

When listing the directory, we can see the `alice_in_wonderland.txt` file is back.

Start of the new file that's just been retrieved from the server.



End of the new file sent from the server (still with 3600 lines)

Interoperability - Handling Errors

Our TFTP Server should be able to handle errors where the file is not found by the server upon a read request. We can test this works between the `tftp` utility and the server I have made by sending a get request for a file that doesn't exist on the server.

```
tftp> get this_isnt_here.txt
sent RRQ <file=this_isnt_here.txt, mode=octet>
received ERROR <code=1, msg=File Not Found>
```

Error packet sent when a file isn't found. (`tftp` unix client)

```
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:64818: Received RRQ - Filename: this_isnt_here.txt
TFTP_REQUEST_HANDLER - 0:0:0:0:0:0:0:1:64818: File Not Found
```

Server output when file not found.

As can be seen the, correct error code (`1`) is sent back to the client. This indicates that the error that occurred was a file not found error.