

## **Table of Contents**

A. Introduction	2
B. Description of Client and Server Programs	3
B1. Content Registration	
B2. Content Download	
B3. Content Listing	
B4. Content De-Registration	
B5. Quit	
C. Observations and Analysis	7
C.1 Connection	
C.2 Content Registration (Type R)	
C.3 List of Registered Elements (Type O)	
C.4 Content Download (Type C and D)	
C.5 Local List of Files	
C.6 File Selection (Type S)	
C.7 De-Register Content (Type T)	
C.8 Quit	
C.9 Error and ACK (Type E and Type A)	
D. Conclusion	18
E. Appendix	19
F. References	25

## A. Introduction

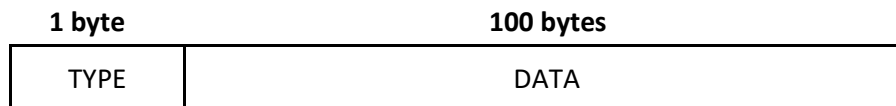
This project entails key aspects of networking including implementation of file transfer applications based on UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). UDP is a connectionless protocol which means it does not need to establish a connection before sending data. TCP is a connection-oriented protocol, defined by establishing and maintaining a network connection in which the applications (Server and Client) can exchange data.

Our server can handle file transfer sessions with a given client, where the client can request both download and upload of a given file. Furthermore, the client can request to list the files in a given directory and change to a given directory on the server end. This is so the client can see the options of files available to be downloaded, or to see if the uploaded file is in fact uploaded to the server. In this environment, a peer (the network application) can be both a client and server. More specifically, when Peer A allows its files to be downloaded it then registers the content to the index server. When the content becomes registered it becomes the server of the content. Peer B will have to contact the index server and get the address of Peer A in order to download the file; hence becoming the server of the content as well.

We use socket programming for the establishment of connection between the client and the server. Socket programming in a broader sense is the linking of the client and server. In other words, it is the process in which the client and server establish connection with each other. The project uses sockets as a means of communication through ethernet ports and is programmed. The sockets act as the interface between the application and the transport layer in the OSI layer model. The sockets are first created in the files and they contain three parameters domain, type and protocol. There is a server program and a peer program each of which communicate with each other with their associated IP address and port number and can exchange files. In the project, the peer has the ability to be both a client and server which requires it to have a numerous number of sockets opened. The role of its clients can be associated with sockets and can also be associated with its servers from which the peers can download contents.

## B. Description of Client and Server Programs

The UDP protocol was implemented with the use of Protocol Datagram Unit(s) (PDUs). PDU is a block of units that is transferred between the client-server network connection. The PDU is so significant as it is structured as a means of information that is passed for an intended function or purpose, which can be clearly seen in the implementation of this project. This project uses eight types of PDU formats to build a Peer to Peer Application program. All of these PDUs have the specific format given in Figure 1 below.



**Figure B.1.** PDU format used for the project [1].

The eight PDU formats that are used within this project to build a P2P connection are provided below:

**Type 'R'** - Content Registration

**Type 'D'** - sent by client to request a file download from server.

**Type 'S'** - Search Content Server

**Type 'T'** - Content De-Registration

**Type 'C'** - Content Data

**Type 'O'** - List of On-Line Registration Content

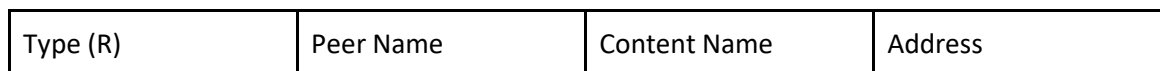
**Type 'A'** - Acknowledgement

**Type 'E'** - Error

We build this P2P application using two programs, index\_server and peer, with the help of these eight PDU types. We will first further discuss these eight different types of PDUs, with the addition of how each program (index server and peer) handle these different types of PDUs.

### 1. Content Registration

The R-type PDU is sent by using UDP which allows a peer to register its contents to the index server. As seen in Figure 1, the data portion of the PDU contains a 100 bytes and consists of the peer name, content name, and the address which is the IP address plus the port number. This is where the content can be downloaded. The figure below illustrates the format of the R-type PDU.



**Figure B.2.** Representation of R Type PDU [1].

Where type R is a data type and peer name, content name and address are data content from exchanged between the peers and server. The size of the peer name and content name fields are fixed

to 20 bytes each for simplification. When an R-type PDU is received by the index server, it will first validate whether or not another peer with the same name registered the same content name. When this occurs, an E-type PDU will be sent to the index server to prompt the peer to select a specific peer name. When the peer name no longer clashes, the message server will record the object and store the address associated with it. Afterwards, it sends back an A-type PDU, which states the success of the content registration. The figure above illustrates the registration procedure among a peer and index server when the peer needs to change its name because of same name conflict.

In implementational words, once the peer chooses to register a file, they input in the command 'R'. Once the peer program recognizes this command, it'll ask the peer to enter the filename of the file they wish to register. Once the filename is inputted by the peer, the peer program will build an R-type PDU with this filename and peer's name, along with the peer's address (IP address + Port), and send it to the index server for registration. The index server will extract the filename and peer name from this PDU once it's been received, and then it will check the data type. It will then use the extracted peer name and filename to check whether these elements already exist within the list of registered files. IF these elements are not already existing within the list, the index server will append this new data into the list and send the peer an A-type (Acknowledgement) PDU, ensuring the peer that their file has been registered successfully.

In the case that the peer name and filename matches an entry within the list, the index server will send back an E-type PDU with an error message. Informing the peer to choose to register under a different peer name. Once the peer receives this E PDU, the peer program will run through a while loop and ask the peer to choose a different name. The peer program will consistently send more R-Type PDUs to the index server under these new names. Once the peer has successfully chosen a unique name, the index server will follow through with the A-type PDU, a success scenario as mentioned above..

## 2. Content Download

The index server is contacted by the peer in order to find the address of the content server for content download. A S-type PDU is used to do this with the format shown in the following figure.

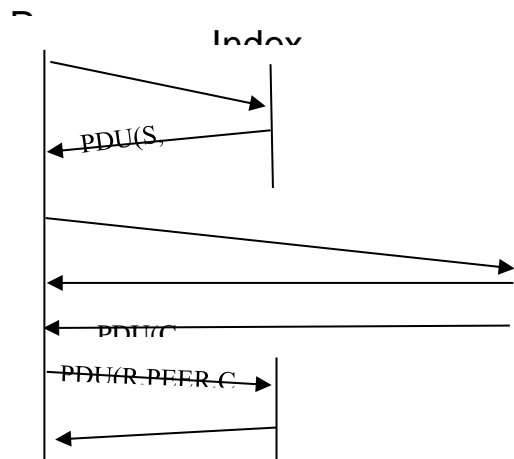
Type (S)	Peer Name	Content Name
----------	-----------	--------------

**Figure B.3.** Representation of S Type PDU [1].

The data type here is 'S' and the data consists only of the Peer Name and Content Name. The index server can either send a S-type PDU or an E-type PDU to respond, depending on whether there is an error or not. An S-type PDU means it contains the address of the content server. An E-type PDU shows that the content wanting to be retrieved by the peer is not accessible.

A TCP connection will be set up with the content server and the address will be derived from the PDU if an S-type PDU is received by the peer. A successful TCP connection results in a D-type PDU to be

sent to the content server for downloading. A C-type PDU is sent if the content is available. The peer becomes the server by downloading the content and registering the content to the index server. The figure below shows the PDU transaction for content downloading.



**Figure B.4.** PDU Transaction for Content Downloading [1].

### 3. Content Listing

If a peer wishes to view the files that have already been registered by other peers, they can use the 'O' command. An O-type PDU is sent by the peer to the index server to determine the registered on-line list of contents. The index server will also send an O-type PDU in return in response. The peer can view this list and wish to obtain any file they please, once the file the peer wishes to download is inputted, the peer will send an S-type PDU to the index server. The download steps will follow what's mentioned in section

#### 2. Content Download.

### 4. Content De-Registration

When a peer wishes to de-register any one of their files, a T-type PDU is sent by the peer to the index server to de-register their content. On the peer program's side, a T-type PDU is enabled when the peer inserts 'T' as a command.

The peer program will then ask the peer which file they wish to de-register. Once this file is inputted, the peer program will call onto the `de_register()` function, this function builds a T-type PDU with the peer's name and the target file they wish to de-register. Once the index server receives this PDU, it extracts the peer name and filename from the PDU's data and checks within the pre-registered files if the filename and peer name matches any options (if file exists). If the file exists, the index server will remove the file from the list, hence de-registering the file itself. The file will no longer be accessible seeing as it no longer exists, unless it is registered again.

### 5. Quit

Registered content must first be de-registered to keep the index server up to date, which is done by sending multiple T-type PDUs to the index server. The peer may use the Quit [Q] command to de-register all of their files from the registered list of files and exit the program entirely (log-out).

Once the peer inserts their [Q] command, the peer program will run through the list of pre-registered files (O-type) and check every file that shares the same name as the username (`peer_name == username`), and run the `de_register()` function on all of those items. The `de_register()` function is the same as the one used for de-registering single files (T-type). Essentially, for the [Q] command, the program will iterate through every single tuple which shares the same name as the user and it will send T-type PDUs (using the `de_register()` function) to the index server for each one of them. Ultimately, deleting all the content that the peer had registered under their name. And then the program uses the `exit()` function to quit the program (log-out).

## C. Observations and Analysis

### 1. Connection

Before we send or receive any form of data between our programs, we must first establish a network connection. In order to connect to the peer, the server will first create a socket object and bind it to its port, using its IP address. Once binded, the server will then listen to incoming client connections on its port.

```
Server listening...  
<socket.socket fd=412, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 60000)>  
Client is connected
```

*Figure C.1.1. Representation of listening to incoming client connections*

In the above image, we can see the output print statements of the server. The server here is listening to client connections to connect with, we can also see the address that has been binded by the server: ('192.168.107.1', 60000). This address displays firstly, the IP address of the index server and secondly, the port of the index server. These addresses are useful when wanting to connect to other peers or other servers. Once the server is connected to the peer, it prints the statement "Client is connected" and sends a message: "Thank you for connecting" to the peer.

When we run our peer, we are met with our main menu. Once the connection is established and the peer receives the server's message, our program will ask for the name of the peer who is logging in and the port the peer is logging in from. Our first peer will be named Sam, connecting from port 10000.

```
b'Thank you for connecting'  
Please enter preferred username: Sam  
Please enter listening port number: 10000  
waiting for the next event  
Please choose from the list below:  
[O] Get On-line list  
[L] List local files  
[R] Register a file  
[T] De-register a file  
[Q] Quit the program  
|
```

*Figure C.1.2. Representation of peer logging on*

From here, the peer is able to use any command they desire, whether they wish to register a new file, see the list of registered files, de-register a file, download a file, etc.

### 2. Content Registration (R-Type)

Once the program runs, the peer is asked to input in their preferred username and their port number (used later for Download Content). In this section, we will first register a file: Text.txt, under the peer name: Sam. In order to register, we need to input in the command 'R'. The program will then ask us to enter a filename that we wish to register. Once we've registered, it'll send this inputted information to the index server as an R-type PDU. If we are successful in registering our content, we will receive an acknowledgement message from the index server (A-PDU). This success scenario can be seen in the figure below.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
R
To register a file, please enter filename below:
filename: Text.txt
Done sending R-type PDU to Server
ACK Message from server: Successfully registered on list
File registration acknowledged! File registered!
```

**Figure C.2.1.** Representation of content registration

Let's consider the scenario where we receive an error message, where we are trying to register an already existing file name under an already existing peer name. Let's try registering the file 'Text.txt' again under Sam. This is shown in the figure below.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
R
To register a file, please enter filename below:
filename: Text.txt
Done sending R-type PDU to Server
Error Message from server: Username and file already exists in list, please pick another username/file combination
Please enter preferred username: |
```

**Figure C.2.2.** Example of error message

When we try to register with the same peer name and filename, we get an error message saying that the file already exists in the list. Our peer program then prompts us to re-enter a different username to register the file under. Let's say we've picked the name 'Jo' to register under, our program does the same as it did before. It sends an R-Type PDU to the index server to check whether this



combination is viable (if it isn't already pre-existing). Since we've only registered Text.txt under Sam, we can easily register Text.txt under Jo.

```
To register a file, please enter filename below:
filename: Text.txt
Done sending R-type PDU to Server
Error Message from server: Username and file already exists in list, please pick another username/file combination
Please enter preferred username: Jo
Done sending R-Type PDU to server
ACK Message from server: Successfully registered on list
```

**Figure C.2.3.** Representation of registering a file

Lastly, let's try to register another filename, Text1, under the username 'Jo'. This output is shown in the figure below. We can see that we are still able to register a different filename under the same peer.

```
To register a file, please enter filename below:
filename: Text1.txt
Done sending R-type PDU to Server
ACK Message from server: Successfully registered on list
File registration acknowledged! File registered!
```

**Figure C.2.4.** Representation of registering a different file under the same peer

Jo now has two file registrations and Sam has one. We will look at the list of these registrations in the next section.

### 3. List of Registered Elements (O-Type)

So far, we've only registered three files under two peer names. Peer Sam has registered file Text.txt and peer Jo has registered Text.txt and Text1.txt. If a peer wishes to check what files have already been registered, they can simply use the [O] command. This command will prompt the index server to send the data within the list of pre-registered files over to the peer program to display to the peer. This is shown in the figure below.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
O
[('Sam', 'Text.txt'), ('Jo', 'Text.txt'), ('Jo', 'Text1.txt')]
```

**Figure C.3.1.** Representation of on-line list of pre-registered files

Our peer program also then prompts the user to enter a file name they wish to download. We will look into this in the next section.

#### 4. Content Download (C & D Types)

When the peer wishes to download a file, they can first place in the command [O]. Command [O], as mentioned before, retrieves the list of online registered files. The peer can look through this list and choose to download any file they wish.

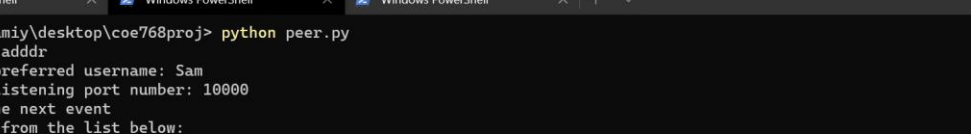
```

Please choose from the list below.
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
0
[['Sam', 'Text.txt'], ('Jo', 'Text1.txt')]
Enter file you wish to obtain from list: Text.txt
S
('192.168.107.1', 10000)
192.168.107.1
sending Type D request
waiting for the next event
3
Connection from ('192.168.107.1', 50881)
b'\x80\x04\x95H\x00\x00\x00\x00\x00\x00\x00CS\x80\x04\x95H\x00\x00\x00\x00\x00\x00\x00\x00\x8c\x08_main_\x94\x8c\x03PDU\x94\x93\x94\x8c\x01D\x94']\x94(\x8c\tpeer_name\x94\x8c\x02Jo\x94\x8c\tfile_name\x94\x8c\x08Text.txt\x94u\x86\x94\x81\x94.\x94.'
b'\x80\x04\x95H\x00\x00\x00\x00\x00\x00\x00\x00\x8c\x08_main_\x94\x8c\x03PDU\x94\x93\x94\x8c\x01D\x94']\x94(\x8c\tpeer_name\x94\x8c\x02Jo\x94\x8c\tfile_name\x94\x8c\x08Text.txt\x94u\x86\x94\x81\x94.'
Determined file to get content

```

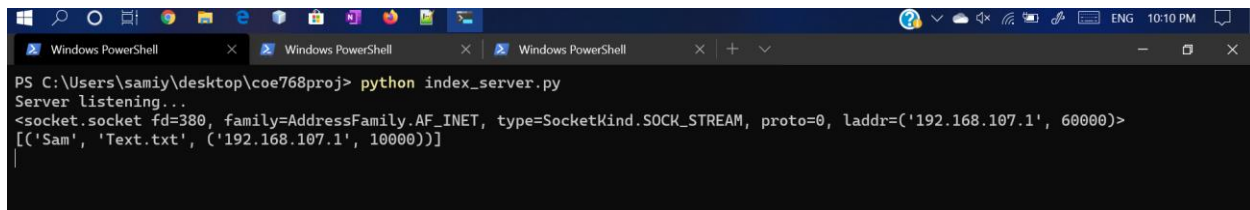
**Figure C.4.1.** Representation of D-Type PDU.

Here, we can see that the peer has placed in the command [O] which gives them the registered list. Currently, there are two registered files, Text.txt by Sam and Text1.txt by Jo. We see here that the peer wishes to obtain file Text.txt. The peer sends a D-Type PDU to the content server in order to retrieve the file contents and download the file. However, our program is unable to get a response from the content server. Hence, this results in the server not being able to respond back with a C-Type PDU carrying the file contents in 100 byte packets.



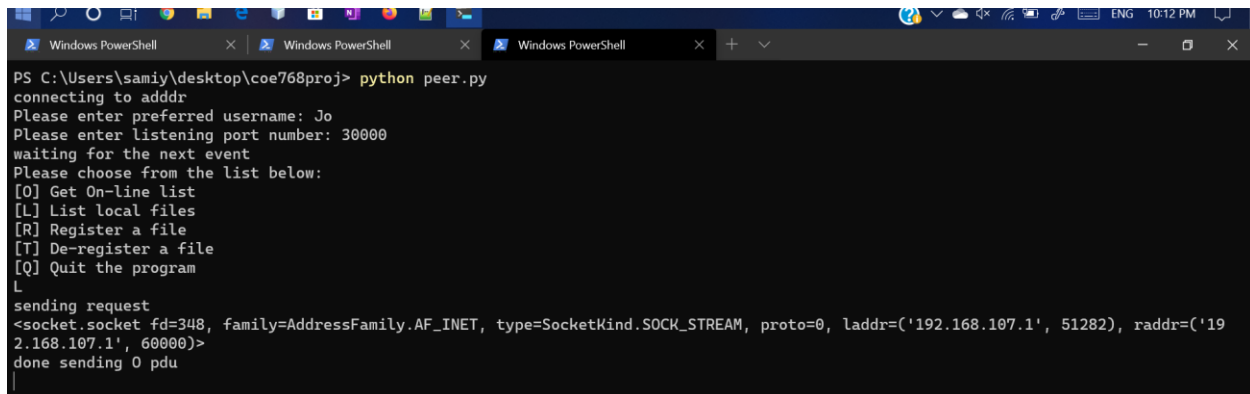
```
PS C:\Users\samiy\desktop\coe768proj> python peer.py
connecting to addrr
Please enter preferred username: Sam
Please enter listening port number: 10000
waiting for the next event
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
R
To register a file, please enter filename below:
filename: Text.txt
Done sending R-type PDU to Server
ACK Message from server: Successfully registered on list
File registration acknowledged! File registered!
waiting for the next event
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
```

**Figure C.4.2.a.** Representation of D-Type PDU.



```
PS C:\Users\samiy\desktop\coe768proj> python index_server.py
Server listening...
<socket.socket fd=380, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 60000)>
[('Sam', 'Text.txt', ('192.168.107.1', 10000))]
```

**Figure C.4.2.b.** Representation of D-Type PDU.



```
PS C:\Users\samiy\desktop\coe768proj> python peer.py
connecting to addr
Please enter preferred username: Jo
Please enter listening port number: 30000
waiting for the next event
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
L
sending request
<socket.socket fd=348, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 51282), raddr=('192.168.107.1', 60000)>
done sending 0 pdu
```

**Figure C.4.2.c.** Representation of D-Type PDU.

Looking at the figures in C.4.2, C.4.2.a is peer Sam (peer A), C.4.2.b displays the index server program and C.4.2.c displays the peer Jo (peer B) logging in. We can see that the index server only responds to Sam and not to the second peer that has logged in. This makes download content retrieval difficult, seeing as the second peer is needed to be logged in order for the connection to go through and the peer B to send the download content to peer A. Unfortunately, our program is unable to do this, it is only able to send a D-type PDU. C-type has been implemented, but since the connection isn't able to be formed, we are unable to see the downloaded file.

## 5. Local List of Files (L-Type)

If the peer wishes to view the files they've already registered, without viewing the entire list of registered files. They can use the common [L] to retain a list of only their registered files. The figure below shows this PDU type for peer Jo.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
L
sending request
<socket.socket fd=300, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 64229), raddr=('192.168.107.1', 60000)>
done sending 0 pdu
Printing list of registered elements...
('Jo', 'Text.txt')
('Jo', 'Text1.txt')
```

*Figure C.5.1. Representation peer Jo's local files.*

## 6. File Selection (S-Type)

S-type PDUs are sent from the peer if they wish to search content and retrieve a file which is handled by the index server. We can reach the S-type PDU through O-Type PDU on the peer end. Once the O-type PDU prints out the list of files within the on-line registered list, it asks the user to input in a filename that they wish to obtain.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
O
[('Sam', 'Text.txt'), ('Jo', 'Text.txt'), ('Jo', 'Text1.txt')]
Enter file you wish to obtain from list: Text1.txt
S
('Jo', 'Text1.txt', ('192.168.107.1', 10000))
```

*Figure C.6.1. S-Type PDU being sent from Index Server to acknowledge server content.*

If we wish to obtain the file Text1.txt from the server index, the index will look for the file and send back an S type PDU as acknowledgement with the properties within the list. As we can see, the index has sent over the peer's name, filename and address. We can use this data from this S-PDU to retrieve and download our file from the peer. Let's try to search for a file that does not exist.

```

Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
O
[('Sam', 'Text.txt'), ('Jo', 'Text.txt'), ('Jo', 'Text1.txt')]
Enter file you wish to obtain from list: Txt
E
Error: Could not find file to download

```

**Figure C.6.2.** Error message when Index Server can't find inputted server content.

As seen in the image above, in the event that the file entered does not exist, the server will send an error message saying the file could not be found.

## 7. De-Register Content (T-Type)

In order to de-register content, the peer can use the command [T]. This command will enable the peer to choose a file of their choice that they wish to de-register from the list of registered content. Let's say peer Jo wishes to de-register their file Text.txt. When we print the local list, we can see that Text.txt no longer exists for peer Jo.

```

Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
T
Enter filename you wish to de-register below:
filename: Text.txt
Done sending T PDU
Filename: Text.txt
successfully removed from list
waiting for the next event
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
L
sending request
<socket.socket fd=312, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 64257), raddr=('192.168.107.1', 60000)>
done sending 0 pdu
Printing list of registered elements...
('Jo', 'Text1.txt')

```

**Figure C.7.1.** De-registering files

From our server side, we can see that after client Jo connects and chooses to de-register their 'Text.txt' file, the server looks for the file and deletes it once it is found (shown below).

```
Client is connected
Found file to delete
[('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text1.txt', ('192.168.107.1', 10000))]
file deleted
[('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text1.txt', ('192.168.107.1', 10000))]
```

**Figure C.7.2.** De-registering files and deleting file once found

Let's consider the case where the peer wishes to de-register a file that doesn't exist. Since, we've already de-registered Text.txt, let's try to de-register this file once again.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
T
Enter filename you wish to de-register below:
filename: Text.txt
Done sending T PDU
Filename: Text.txt
Error: Could not remove file
```

**Figure C.7.3.** Error message when file cannot be removed

We are met with an error message from the server informing the peer that the file could not be removed. This is because the file no longer exists.

## 8. Quit

When a peer wishes to de-register all of their files, they can use the command [Q]. This command will de-register all of their files and simultaneously exit the program (log them out). Let's use the command [Q] on Jo. To set this into perspective, let's first use the command [L] to see what files Jo has registered.

```
Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
L
sending request
<socket.socket fd=312, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 64257), raddr=('192.168.107.1', 60000)>
done sending 0 pdu
Printing list of registered elements...
('Jo', 'Text1.txt')
```

**Figure C.8.1.** Implementation of command Q and L

Looks like Jo has Text1.txt still registered, we are aware of this seeing as we did not touch this file whatsoever. Now, let's call Q on Jo and see what happens.

```

Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
Q
done sending 0 pdu
printing files list
Registered files before [Q]:
[('Sam', 'Text.txt'), ('Jo', 'Text1.txt')]
Done sending T PDU
Filename: Text1.txt
successfully removed from list

```

**Figure C.8.2.** Successful De-registration of all Files.

We can see in the above image that Jo has successfully de-registered Text1.txt from the list. On the server side, we can see the behaviour as well.

```

Client is connected
Found file to delete
(('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text1.txt', ('192.168.107.1', 10000)))
file deleted
(('Sam', 'Text.txt', ('192.168.107.1', 10000)),)
Client is connected

```

**Figure C.8.3** Successful deletion of files.

This runs similar to the T-type PDU, seeing as [Q] sends multiple T-type PDUs for the server to delete. We can see from the server's end, that the file Text1.txt has been successfully found and deleted. To see it from a different perspective, let's register some files under Jo. We've gone ahead and registered three files under Jo, as shown in the figure below.

```

Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
L
sending request
<socket.socket fd=320, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('192.168.107.1', 64336), raddr=('192.168.107.1', 60000)>
done sending 0 pdu
Printing list of registered elements...
('Jo', 'Text.txt')
('Jo', 'Text2.txt')
('Jo', 'Text3.txt')

```

**Figure C.8.4.** Files registered by peer Jo

We have Text.txt, Text2.txt and Text3.txt files registered under Jo's name. Let's call the [Q] command and see what happens.

```

Please choose from the list below:
[O] Get On-line list
[L] List local files
[R] Register a file
[T] De-register a file
[Q] Quit the program
Q
done sending 0 pdu
printing files list
Registered files before [Q]:
[('Sam', 'Text.txt'), ('Jo', 'Text.txt'), ('Jo', 'Text2.txt'), ('Jo', 'Text3.txt')]
Done sending T PDU
Filename: Text.txt
successfully removed from list
Done sending T PDU
Filename: Text2.txt
successfully removed from list
Done sending T PDU
Filename: Text3.txt
successfully removed from list
Registered files after [Q]:
[('Sam', 'Text.txt'), ('Jo', 'Text.txt'), ('Jo', 'Text2.txt'), ('Jo', 'Text3.txt')]
PS C:\Users\samiy\desktop\a> |

```

**Figure C.8.5.** De-Registering using [Q].

Here, our program iterates through each of the peer's files and calls upon the `de_register()` function to individually send T-type PDUs to the server in order to delete each file individually. We can see that all three files have been removed at the same time and then the program exits. From the server's side, we can take a better look at it. This is shown in the figure below.

```

Client is connected
Found file to delete
[('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text2.txt', ('192.168.107.1', 10000)), ('Jo', 'Text3.txt', ('192.168.107.1', 10000))]
file deleted
(('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text2.txt', ('192.168.107.1', 10000)), ('Jo', 'Text3.txt', ('192.168.107.1', 10000)))
Client is connected
Found file to delete
(('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text2.txt', ('192.168.107.1', 10000)), ('Jo', 'Text3.txt', ('192.168.107.1', 10000)))
file deleted
(('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text3.txt', ('192.168.107.1', 10000)))
Client is connected
Found file to delete
(('Sam', 'Text.txt', ('192.168.107.1', 10000)), ('Jo', 'Text3.txt', ('192.168.107.1', 10000)))
file deleted
(('Sam', 'Text.txt', ('192.168.107.1', 10000)),)

```

**Figure C.8.6.** Representation of T-PDU deletion from Index Server.

We can see how the program works here better. As the peer program is sending each individual T-type PDU over to the index server to de-register, the index server de-registers one at a time, as it receives each separately. We see `Text.txt` first being found and deleted, then we see `Text2.txt` being deleted and finally we see `Text3.txt` being deleted.



## **9. Error and ACK (E-type & A-type)**

In the above cases, we have gone through error and acknowledgement type PDUs. These are used as a form of communication between servers and clients. Error messages/PDUs are sent from the server to the client when there is an error, for example when searching for a file that does not exist. Acknowledgement type messages are sent when the retrieval or function is successful. For example when registering a filename under a peer successfully, the index server will send an ACK type PDU to the peer. These forms of communications are very important, as they show what is going on both ends. When we connect clients to servers, these programs run independently for the most part. This makes it harder to understand what is going on the other side of the network. Error and acknowledgment messages help with moving forward with the program and communicating when necessary. These messages also help both server and client know when to wait and send a packet.

## D. Conclusion

To conclude, implementation of file transfer functions based on TCP and UDP was a success. Sockets were created and the server code is run, the server then listens and waits for the client to send a request to connect. Shortly after their respective sockets are then used to send data blocks called PDUs. It essentially was able to upload and download files while also being able to move around the directory. This project demonstrated various components that are essential to our digital and social society. Furthermore, this project successfully enabled us to understand the true importance of sockets, TCPs, UDPs, and the steps in transferring data.