COMPUTER SYSTEMS 2 (CEN502)

PROJECT 2

A SIMPLE PEER TO PEER FILE SHARING SYSTEM

GROUP 25:

ABHISHEK JOSHI

ANIKET WANI

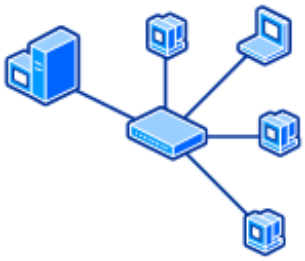
SIVAKUMAR VENKATARAMAN

INDEX

1. Introduction:

File sharing is the practice of distributing files from one computer to another over a network or the internet. File sharing can be implemented in many ways. Two main models used in the current network applications are the client-server architecture and the peer-to peer (P2P) architecture. [1]

The primary difference between peer-to-peer and client-server networks is that peer-to-peer networks do not have a central server to manage network resources. Instead, resources on a peer-to-peer network are distributed between the various clients that make up that network. As a result, peer-to-peer networks are easy to configure, but can be less secure than client-server networks. [1]



In a server-based network, the server is the central location where users share and access network resources. This dedicated computer controls the level of access that users have to the shared resources. Shared data is in one location, making it easy to back up critical business information. Each computer that connects to the network is called a client computer. In a server-based network, users have one user account and password to log on to the server and to access shared resources. Server operating systems are designed to handle the load when multiple client computers access server-based resources. [2]

In a peer-to-peer network, a group of computers is connected together so that users can share resources and information. There is no central location for authenticating users, storing files, or accessing resources. This means that users must remember which computers in the workgroup have the shared resource or information that they want to access. It also means that users must log on to each computer to access the shared resources on that computer. Often, there are multiple versions of the same file on different computers in the workgroup. [2]

In this project, we built a simple peer to peer file sharing system which consists of one centralized directory server, which maintains a directory of text files that users are willing to share. The directory server contains entries which list text file name, file size, and the IP address of the peer where the file resides. Whereas, each peer combines the functionality of a P2P client and a transient P2P server.
The two major implementations done in this project are:

1. The implementation of a communication protocol between the directory server and a P2P client – using UDP sockets.
2. The implementation of a communication protocol between a P2P client and a transient P2P server – using TCP sockets.

The overall implementation details, design decisions, protocol choices and all relevant details have been discussed further in this report.

2. Components of our P2P file sharing system:

2.1. Central Directory Server:

The Central Directory Server stores the information of what files are present in which P2P client. We have used a python list where we store three data fields for each file information. The three data fields are **file name, file size and the IP address** of the host where the file is present. The list is updated dynamically based on requests that the Central Directory Server receives. Central Directory Server was designed to handle three kinds of commands/messages from the P2P client **1. Query for Content 2. Inform and Update 3. Exit**. The message types and functions are explained later in the report in 'Message Types' section. The server functions independently **without user intervention**. However, it shows meaningful processing messages as and when it receives any command. This was shown to trace the functioning of the server and better illustrate its working. The Server was **multithreaded** and for every message it receives, it will start a new thread to process the information needed. Meanwhile, it can receive and process messages from other P2P clients.

Data Structure Used: Python List- A simple array was used to store the filename, file sizes and IP address of the peer that has it. The array is traversed linearly to look up entries.

2.2. P2P Client:

P2P client can interact with the Central Directory Server through UDP messages. It can send three types of commands to the Central Directory Server 1. Query for Content 2. Inform and Update 3. Exit. Our P2P client design provides a console based interface for the user. In the same interface that is used for sending/receiving messages, P2P client can be made to ask for a file from a transient P2P server in the network. We left it to the user discretion to choose the peer to ask the file.

2.3. P2P Server(Transient):

Each P2P client also acts as a transient server and can accept requests from other P2P clients for files. This was done by creating a separate thread for the transient server part. Therefore, it can accept connections from other P2P clients (through TCP sockets) while it is sending other messages to the Central Directory Server. The file transfer between the P2P transient server and P2P client server is done by using TCP sockets. The P2P transient server can service multiple P2P clients at the same. It opens a **separate connection** for each P2P client and services them separately. This was done by implementing **multithreading** in the transient server. It accepts a simple 'GET' request, which resembles HTTPv1.1 GET request, and sends the file to the client in the response.

3. Communication between P2P client and Central Directory Server:

3.1. Types of Messages:

1. Query for Content:

   Whenever a P2P client wants to know if a file is present in any of its peers, it sends this message to the Central Directory Server. We used (and built on) the HTTP message format. The server on receiving this message scans through its 'List of Files' and finds each of the peers that has the file. It then returns a message that includes the peer IP addresses where the file can be found. The response message serves as an acknowledgement and it also uses HTTP response format with proper status codes for 'Success' or 'Error'. The P2P client can also send a Query for Content message with no filename specified. In this case, the server responds with the entire list of files and the peer IP address where they are present.

2. Inform and Update:

   This type of message is used when the P2P client wants to let the server know it has some files and is ready to share it. The file names and sizes are attached in the message and sent to the server. The server adds the file names, files sizes and the IP address of the client from which it received this message to the list. Server also responds with a 'Success' message after adding the information to its list. This response also serves as an acknowledgement.

3. Exit:

   When the client doesn't want to share any more files with any other peers, it asks the Central Directory Server to remove its address from all the entries in the directory. The server on receiving this message removes all the entries associated to this client in its directory and sends a 'Success' acknowledgement to the P2P client.

3.2. Message Formats:

From Client to Central Directory Server:

UDP request message format:

Sequence Number	Request Number	Partial or not (i.e. fragmented or not)	Method (Type of request)	URL(Client Hostname)	Version(Client address)	Filename (if sent)	File Size (if sent)

All the fields are separated by one 'whitespace' character. This convention was followed to keep the message format simple.

From Server to P2P client:

UDP response message format:

Sequence Number	Request Number	Partial or not (i.e. fragmented or not)	Method (Type of request)	Status Code (200,400)	Status Phrase	File Name (if needed)	File Size(if needed)	Peer IP address(If needed)

Same as the request message, all the fields are separated by one 'whitespace' character.

3.3. Fragmentation and Reassembly:

Since the MTU (maximum transmission unit) is given as 128 bytes for all the communication between Central Directory Server and the P2P clients, we fragment big packets into packets of size 128 bytes or less and send it. The server and client are designed to fragment and are also capable of reassembling the packets.
When fragmenting, we kept sequence number, request number and 'fragmented or not' fields to be mandatory in every packet. This enables us to reassemble the packet without hassle. Sequence number helps us in uniquely identifying a packet, while the request number tells us if the incoming fragments belong to the same request. The 'Partial or not' field is similar to frag field in the IP datagram and it helps the server and client determine if the received message is a fragmented message.

3.4. Sequence Numbers and Acknowledgements:

We have implemented a form of rdt 3.0 stop and wait protocol. Each packet has to be acknowledged for the next packet to be sent. This enforces reliability while keeping the design simple.
Every packet sent by the client has a new sequence number which is the first field in any message sent between the P2P Client and the Central Directory Server. The server responds with the acknowledgement that has the same sequence number. If the client receives an acknowledgement with sequence number not matching with the packet that it sent, it resends the packet just like rdt 3.0. This also handles duplicate acknowledgements if any was received.

3.5. Timeout and Retransmission:

Python sockets are 'blocking' by default. We implemented timeouts such that when a packet doesn't receive any acknowledgement, it is sent again after timeout. Through this method, we enforce reliability and handle packet lost and acknowledgement lost scenarios.

- Calculation of Timeout Interval:

  We use the parameters given in textbook, namely alpha (0.125) and beta (0.25), and dynamically update the timeout interval for every acknowledgement received (otherwise for every Rtt).
  These formulas were used for implementing the 'Exponential Weighted Moving Average' timeout that also accounts for deviation. [3]

- EstimatedRtt=((1-alpha)*EstimatedRtt)+(alpha*SampleRtt)
- DevRtt=((1-beta)*DevRtt)+(beta*|SampleRtt-EstimatedRtt|)
- TimeoutInterval=EstimatedRtt+(4*DevRtt)

Although we were asked to use initial EstimatedRtt=0.1 seconds (100 ms), we found it was too low to start with since it doesn't take into account the time taken to process the message by the server. Hence we used an initial EstimatedRtt of 2 seconds. This initial assumption time seems optimal in different tries and different scenarios and is also not that larger than the given initial EstimatedRtt. Further, this value is dynamically updated based on the Sampled Rtt for every acknowledgement received.

3.6. Protocol:

As per the requirement, UDP socket was used for all the communication between P2P client and Central Directory Server. We implemented a form of rdt 3.0 stop and wait protocol for all communication between P2P client and Central Directory Server. [3]

- All new packets from the client to server have a new sequence number. They can have the same request number if they are part of the same request. Their 'Partial or Not' field is set to 1 if the packet belongs to a message that was segmented/fragmented.
- Server acknowledges each packet sent by the client. If the packet is lost or acknowledgement is lost, client resends the packet after timeout. This enforces reliability.
- Timeout was decided based on the parameters α and β from the textbook and also initial EstimatedRTT was taken to be 2 seconds.

4. Communication between P2P Client and transient P2P Server:

4.1. Types of Messages:

From P2P Client to P2P Transient Server:
Only one type of request was required- HTTP GET request message and it was implemented. [4]

From P2P Transient Server to P2P Client:
The response message from P2P transient server to the P2P client is also in the HTTP v1.1 format. It supports all the status codes what were required (200,400,404 and 505). The file is attached to the contents of the response message if the request satisfies all conditions and if the file was found. [4]

4.2. Message format between P2P client and P2P server:

HTTP v1.1 was used for the messages between P2P client and P2P server.
The request message from P2P client to transient P2P server:

Type of request(GET)	URL (File name)	HTTP Version

All the fields are separated by whitespaces.

The response message from transient P2P server to P2P client:

HTTP Version	Status Code	Status Phrase	Contents of the file(if needed)

All the fields are separated by whitespaces.

4.3. Protocol:

TCP socket was used for the file transfer between P2P client and transient P2P server.

- P2P client sends a 'GET' request to the server.[4]
- P2P transient server can respond with any of the following response messages Ok (200), Bad Request (400) and Not Found (404) and HTTP Version Not Supported (505) based on the message it received.[4]
- P2P transient server sends the file through the TCP socket if the request was a legitimate request and if the file was found. The file is sent in the response message itself.

Note: Reliability and Fragmentation is enforced by TCP.

4.4. Handling Large Files:

As we know, TCP can handle fragmentation and reassembly by itself. But it is limited by its receiving buffer size. This was handled by removing data from buffer as and when it is filled up in the receiver side. P2P server sends the file size in the response message along with the contents of the file. This file size was then used as a loop counter to receive parts of the file and combine them.

5. Design Choices:

1. The Central Directory Server's IP address must be known to the P2P client for it to join the network.
2. All transient P2P servers listen to a fixed listening port. This was done for simplicity. Because of this design choice, all P2P clients can contact any P2P transient server just by knowing its IP address. The P2P client gets the IP address from the Central Directory Server by sending a 'Query for Content' message. If we hadn't made this design choice, we would have to store the TCP port number (in which the peer listens) in the Central Directory Server and send it to P2P client along with the IP address. This increases the amount of information stored in the Central Directory Server.
3. Initial EstimatedRtt was set to 2 seconds instead of 100 ms to account for the processing time overhead in the Central Directory Server. We arrived at this value after trying different values and different scenarios.
4. Our sequence numbers are not just 0 & 1 as in rtd3.0. We use sequence numbers that keep on incrementing. This was done with the intention of implementing a pipelined protocol in the future.
5. Informing the Central Directory Server after getting a file from a P2P transient server. We decided to leave it to the user to update (or not) the Central Directory Server after it gets a file. This gives more control to the peer on whether to share the file to other peers after getting the file.

6. Pseudocode/Algorithm:

6.1. Central Directory Server:

1. Listen and receive messages on a predetermined UDP port.
2. Start a new thread for every message received and pass the received message to the thread
3. Server Process Thread:
    a) Process the received string.
    b) Check if the message is a client acknowledgement.
    c) If yes, save it in the 'ClientAck' list for future reference and exit. If no, continue to next step.
    d) Check if the message is a partial message.
    e) If yes(message is partial),
        - Combine with previous partial message from the same request. If no previous partial message was there, this is the first partial message. Acknowledge client, store it for future reference and exit.
        - Check if the combined message is a full message. If yes, proceed to process the full message. If no, the message is still partial, acknowledge client.
    f) If no(message is not partial), proceed to process the full message
    g) Processing the full message:
        - If message is a 'Query for Content' message, go through the list of files and append results to the response message based on the user's message. Proceed to send a response/acknowledgement.
        - If message is an 'Inform and Update' message, add the filenames and respective file sizes to the directory information list along with the client address. Proceed to send a response/acknowledgement.
        - If message is an 'Exit' message. Delete all file names associated with the client. Proceed to send a response/acknowledgement.
    h) Sending the response message: Determine the size of the response message. If it's more than 128 bytes, fragment it and send every fragment after receiving acknowledgement for each fragment. If the size is less than or equal to 128 bytes, send the full response.

6.2. P2P Client(also Transient Server):

1. Start the thread for acting as a P2P transient server.
2. Start the thread for user interface and communication with the central directory server.
3. User Interface Thread:
    a) Ask the user to choose from 4 choices. 1, Query for Content 2, Inform and Update 3, Exit and 4, Ask for a file from a peer.
    b) Choice 1(Query for Content):
        - If the user is looking for a specific file, ask for the file name, compose the request message. If the user wants the entire directory listing, no filename is attached to the request message.

- Break the message into fragments and send if the size of the message is more than 128 bytes. Wait for acknowledgement for each fragment and retransmit the message if timeout happens.
- Server can send partial response fragments based on response message size. Store the partial fragments and combine them to get the full response.
- Display appropriate message based on the server response.

c) Choice 2(Inform and Update):
- Ask for the filenames and sizes from the user and append them to the request message.
- Break the message into fragments and send if the size of the message is more than 128 bytes. Wait for acknowledgement for each fragment and retransmit the message if timeout happens.
- Based on the server response, display the appropriate message.

d) Choice 3(Exit):
- Compose the 'Exit' message in the appropriate format and send to the Central Directory Server.
- Wait for acknowledgement from server and retransmit the message if timeout happens.
- Display the appropriate message when the acknowledgement is received.

e) Choice 4(Ask for a file from a peer/P2P file transfer client):
- Ask the user to enter the IP address of the P2P peer from which they wish to request the file. Also, ask for the file name.
- Open a TCP connection to the P2P peer. Send a message in HTTPv1.1 GET message format.
- Display appropriate message based on server response.
- If the response was '200'-'Ok' message, strip the file contents from the response message and store the file.

f) Repeat from step a.

4. P2P server thread:

a) Listen in the TCP welcome port (predetermined port number) for incoming connection request.

b) If connection request is received, start a separate thread to service the connection.

c) File Transfer Thread:
- Receive the message in the established TCP connection.
- If the message is not in 'HTTPv1.1', send an acknowledgement with '505'-'HTTP version not supported' response and close the connection.
- If the message is not a 'GET' message, send an acknowledgement with '400'-'Bad Request' response and close the connection.
- If the message is in HTTPv1.1 format and if it is a 'GET' message, look for the file. If the file is not found, send an acknowledgement with '404'-'Not Found' response and close the connection.
- If the file was found, append the file contents to the response message, and send the response message.

7. Features Implemented:

- Reliability:
  Since UDP was used for the all the communication between Central Directory Server and the P2P client and since we know it is not reliable, we enforced reliability at the application layer. This was done with help of sequence numbers, acknowledgements, timeout and retransmission. We have implemented a form of rdt 3.0 stop and wait protocol.
- Multi-threading:
  The Central Directory Server was multi-threaded and it can receive messages from multiple clients at the same time. P2P transient server was also multi-threaded and can handle connections to multiple clients at the same time.

8. Conclusion:

We have developed a simple peer to peer file sharing system similar to Napster but for text files. Sockets were used to send messages through the network. Specifically, UDP sockets were used for all the communication between Central Directory Server and the P2P Clients. By trying to implement reliability over UDP, we realized the importance and all the benefits of TCP in current internet structure. TCP sockets were used for the file transfer between peers.

9. References:

1. http://classroom.synonym.com/primary-difference-between-peertopeer-clientserver-architectures-15494.html
2. https://technet.microsoft.com/en-us/library/cc527483(v=ws.10).aspx
3. Computer Networking- A Top Down Approach (Sixth Edition) by James F. Kurose and Keith W. Ross
4. https://www.ietf.org/rfc/rfc2616.txt
5. Python 3.4.3 Socket Documentation- https://docs.python.org/3.4/library/socket.html?highlight=socket