Project 1 - Networks

Muhammad Abyan Ahmed - ma8145, Alisha Atif - aa10699

31st March 2025

1 Introduction

This report documents the development of a file-sharing application project using a hybrid model that combines client-server and peer-to-peer (P2P) architectures. The project involves two main parts: sharing files via chunk distribution to peers and retrieving files through tracker-based peer lookup and reconstruction. The goal is to enable efficient and modular file sharing across a network. The programming language used for this project was Python, with different libraries such as **socket**, **os**, **json** and others used to handle files and communicate with networking hardware.

2 Project Objectives

The objectives for this project included implementing a file-sharing mechanism using file chunking, distributing and storing file chunks across multiple peers, maintain a tracker to manage peer and file chunk information, and allow other users to retrieve and reconstruct file chunks from other peers.

3 Basic System Architecture

The project uses a hybrid model combining the client-server and peer-to-peer (P2P) architectures to enable file-sharing. The tracker is the main server, with the peers, the sender and the receiver all acting as clients. The sender is titled **"Alice"** and the receiver called **"Bob"** in this project. The clients communicate with and receive data about each other from the server, the tracker. Within this client-server model, the peers store and receive the chunks of each file and then send these forward to other peers as required, following the peer-to-peer model, with the tracker storing information about the different peers connected to the network and the files they store.

4 Implementation

As mentioned, the project was implemented in Python using separate **.py** files for each of the **peer**, **tracker**, **Alice (sender)** and **Bob (receiver)**. Below, there is a summary what each of these .py files aimed to implement.

4.1 Alice

In this project, the sender, called "Alice" (**alice.py**), performs a crucial role in the file-sharing process. This file is responsible for splitting the file to be shared into smaller chunks so that it can be shared over the network. It also generates a **file_metadata.json** file containing the original filename and extension. Then, it contacts the tracker (**tracker.py**) to get information about the peers that are active. It sends the chunks of the file to the first two active peers (by design).

4.2 File Chunking

Files are divided into chunks so they can be easily sent and received over the network. The following function splits a big file into smaller chunks and saves them in the "chunks" folder. It reads the file "chunk by chunk" and creates new files for each chunk, effectively splitting the file.

```
def split_file(file_path, chunk_size= 1024 * 1024):
```

```
# Directory where we save the chunks
chunks_dir = "chunks"
# Clear existing chunks from previous runs
if os.path.exists(chunks_dir):
    for filename in os.listdir(chunks_dir):
        file_path_to_delete = os.path.join(chunks_dir, filename)
        if os.path.isfile(file_path_to_delete):
            os.remove(file_path_to_delete)
# Create the chunks folder if it doesn't exist
else:
    os.makedirs(chunks_dir)

# Verify that the file exists. If not, we display an error
if not os.path.exists(file_path):
    print(f"Error: File {file_path} not found")
    return []

# Check file size (this is an additional chekc, where if it is 0,
we return that file is empty)
file_size = os.path.getsize(file_path)
if file_size == 0:
    print(f"Warning: File {file_path} is empty")

# To make our program functionable for several data types (like txt and jpeg),
we implemented the following. We retrieved the extension of the files
# that Alice gets as input
# Get original file extension
_, file_extension = os.path.splitext(file_path)

# Create a metadata file which will store filename and extension
of original file
file_metadata = {
    "original_filename": os.path.basename(file_path),
    "extension": file_extension,
    "size": file_size,
    "chunk_count": 0
}

# Print message to infrom user we are splitting the files
print(f"Splitting file: {file_path} ({file_size} bytes) with extension:
{file_extension}")

# List to store chunk file names
chunks = []

# Open the file in read mode
with open(file_path, "rb") as f:
    # Chunk counter
    i = 0
    while True:
        # Read a chunk from the file
```

```
            chunk = f.read(chunk_size)
            # If no more data, we're done
            if not chunk:
                break

            # Create the chunk file name
            chunk_name = f"chunk_{i}.part"
            chunk_path = os.path.join(chunks_dir, chunk_name)

            # Open the chunk file in write mode
            with open(chunk_path, "wb") as chunk_file:
                # Write the chunk data to the file
                chunk_file.write(chunk)

            # Add just the chunk name to the list
            chunks.append(chunk_name)

            # Display a message for user to stay updated
            print(f"Chunk created: {chunk_name} ({len(chunk)} bytes)")
            i += 1

    # Update metadata with chunk count and save it
    file_metadata["chunk_count"] = i
    metadata_path = os.path.join(chunks_dir, "file_metadata.json")
    with open(metadata_path, "w") as metadata_file:
        json.dump(file_metadata, metadata_file)

    # Add metadata file to chunks list
    chunks.append("file_metadata.json")

    # Notify the user
    print(f"Finished splitting file { total chunks: {i}")
    print(f"Metadata saved with original filename:
    {file_metadata['original_filename']} and
    extension: {file_metadata['extension']}")

    # Return the list of chunk names
    return chunks
```

4.3 Chunk Distribution

Once the file is split into smaller chunks, Alice selects the peers from the tracker and establishes a connection with them. Before any chunk is sent, the function sends a notification (READY_TO_SEND) to the peer, ensuring that the peer is ready to receive the data. The peer must acknowledge this notification with a READY_ACK message, confirming that it is prepared for the incoming chunk. Only after this acknowledgment does Alice proceed to send the chunk. This process ensures that each peer is notified and prepared before retrieving any chunk. By iterating over the list of chunks and sending them sequentially to the selected peers, the function ensures the distribution of file chunks across at least two peers for storage, as required.

```
# This function (send_chunks_to_peer) sends all the chunks from the "chunks" folder to
the specified peer.
def send_chunks_to_peer(peer_ip, peer_port):
    # Get and sort the chunk file names
    chunks_dir = "chunks"
    if not os.path.exists(chunks_dir):
        print(f"Error: Chunks directory {chunks_dir} not found")
        return False
    chunks = sorted(os.listdir(chunks_dir))

    # Move metadata file to the end to ensure it's processed last
    if "file_metadata.json" in chunks:
        chunks.remove("file_metadata.json")
        chunks.append("file_metadata.json")

    # Check if there are any chunks to send.
    if not chunks:
        print("No chunks found to send")
        return False

    print(f"Preparing to send {len(chunks)} chunks to {peer_ip}:{peer_port}")

    # Counter for the successfully sent chunks
    success_count = 0
    for chunk_file in chunks:
     # Initiliaze the path to the chunk file.
        chunk_path = os.path.join(chunks_dir, chunk_file)

        # Skip if not a file or empty
        if not os.path.isfile(chunk_path) or os.path.getsize(chunk_path) == 0:
            print(f"Skipping {chunk_file} - not a valid file")
            continue

        # Create a socket for each chunk
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            try:
                # Connect to the peer
                s.connect((peer_ip, int(peer_port)))
                # Send a handshake message
                s.sendall(b'READY_TO_SEND')
                # Wait for acknowledgement
                ack = s.recv(1024)
                # If acknowledgement is incorrect
                if ack != b'READY_ACK':
                    print(f"{peer_ip}:{peer_port} isn't ready. Got: {ack}. Skipping...")
                    continue

                # Send the chunk file name
                s.sendall(os.path.basename(chunk_file).encode())
                # Small delay to ensure the peer is ready
                time.sleep(0.1)
```

```
                # Send the chunk data
                with open(chunk_path, "rb") as f:
                    data = f.read()
                    s.sendall(data)

                # Close the connection when we are done
                s.shutdown(socket.SHUT_WR)

                # Output message and incrementing the success counter for chunks
                print(f"Sent {chunk_file} to {peer_ip}:{peer_port}
                ({os.path.getsize(chunk_path)} bytes)")
                success_count += 1

            except Exception as e:
                print(f"Couldn't send {chunk_file} to {peer_ip}:{peer_port}
                { Error: {e}")

    # Output user for the user if all chunks sent succesfully
    print(f"Successfully sent {success_count} of {len(chunks)} chunks
    to {peer_ip}:{peer_port}")
    # Return True if at least one chunk was sent successfully.
    return success_count > 0
```

4.4 Tracker

The **tracker.py** is the file that maintains information (like IP Port numbers) about all other participants in the network and sends this data if and when it is requested. It keeps track of all active peers that are ready to receive chunks and shares the list of these peers when required (by for e.g. the sender or receiver). When a peer sends a request to retrieve the list of active peers (by the GET_PEERS request), the tracker responds by sending the current list of registered peers, which allows Alice to find and distribute file chunks to them. Additionally, when a new peer wants to register itself with the tracker (via the REGISTER_PEER request), the tracker adds the peer's information to the list of active peers, ensuring that it can be contacted by other peers for file sharing. The tracker remains active for as long as the project is working and only terminates if done so by the user. This simple mechanism ensures that the tracker can manage and track all active peers, and it serves as the main interface between the peers and Alice, helping to facilitate the chunk distribution process.

```
def start_tracker(host='0.0.0.0', port=9090):
    # Create a socket for the tracker to listen for connections
    tracker_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tracker_socket.bind((host, port))
    # Start listening for incoming connections (we can queue upto 5 connections)
    tracker_socket.listen(5)

    # Let our user know the tracker is running.
    print(f"Tracker is up and running on {host}:{port}")

    while True:
        connection, address = tracker_socket.accept()
         # Inform the user when we get a connection.
```

```
print(f"Got a connection from {address}")
#  Receive data from the peer (up to 1024 bytes) and decode it.
request = connection.recv(1024).decode()
print(f"Request received: {request}")

if request == 'GET_PEERS':
    # If the peer wants the list of peers, send it the 'peers' list.
    connection.sendall(json.dumps(peers).encode())
elif request.startswith('REGISTER_PEER'):
    # If the peer wants to register itself, extract the peer's information.
    # We split the request into two parts: 'REGISTER_PEER' and the peer's info.
    _, peer_info = request.strip().split(' ', 1)
    # If the peer isn't already in the list, add it. This will help us optimise
    and avoid duplicates
    if peer_info not in peers:
        peers.append(peer_info)
        print(f"New peer registered: {peer_info}")
# Close the connection with the peer when we're done with this request
connection.close()
```

4.5 Peer

The **peer.py** file creates a new peer and registers it with the tracker. It then listens for file chunks sent from other peers (e.g. Alice) and saves each chunk locally in a folder. It then remains active to send or receive chunks to other peers on the network. So peers act as both servers and clients, receiving and serving file parts based on requests. Each instance of the **peer.py** file creates a new peer on the network with a dynamically chosen port number.

4.6 Bob

The **bob.py** is designed to retrieve the file shared by Alice. Bob begins by contacting the tracker to get a list of available peers that are hosting the file chunks. Once Bob has a list of peers, it connects to them individually, requesting the chunks it needs. After successfully downloading all the chunks, Bob proceeds to reconstruct the original file using the downloaded data. The reconstruction process uses metadata information (such as the original filename and file extension) to ensure the file is properly assembled. Finally, Bob verifies the reconstructed file to ensure its integrity, completing the download process.

4.6.1 Connecting with peers

To establish a connection with peers and retrieve the necessary chunks, three main functions have been implemented:

- `get_peer_list()`: This function connects to the tracker and retrieves a list of active peers that can be contacted for downloading the file chunks.
- `request_chunks_from_peer()`: After obtaining the peer list, this function contacts each peer to request the list of chunks it has available for download.
- `download_chunk()`: Once Bob knows which chunks are available on each peer, this function is used to download the requested chunks from the peer.

These three functions work together to establish a connection, retrieve available chunks from the peers, and download the required chunks.

Code for `get_peer_list()`

```
def get_peer_list():
    # Creating TCP socket
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        try:
            # Connect to the tracker
            s.connect((TRACKER_IP, TRACKER_PORT))
            # Send a request to get the peer list
            s.sendall(b'GET_PEERS')
            # Receive the peer list data from the tracker
            peer_data = s.recv(4096).decode()
            # Parse the received JSON data and return the peer list
            print(f"Received peer data: {peer_data}")
            return json.loads(peer_data)
        except Exception as e:
            print(f"Error getting peer list: {e}")
            # Return an empty list if an error occurs
            return []
```

Code for `request_chunks_from_peer()`

```
def request_chunks_from_peer(peer_ip, peer_port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        try:
             # Connect to the peer
            s.connect((peer_ip, peer_port))
            # Send a request to get the chunk list
            s.sendall(b'REQUEST_CHUNKS')
            # Receive it
            response = s.recv(4096).decode()
            # Print the received chunk list
            print(f"Received chunk list from {peer_ip}:{peer_port}: {response}")
            chunk_list = json.loads(response)
            # Return the chunk list.
            return chunk_list
        except Exception as e:
            # Print an error message if an exception occurs.
            print(f"Failed to connect to {peer_ip}:{peer_port} { {e}")
            return []
```

Code for `download_chunk()`

```
def download_chunk(peer_ip, peer_port, chunk_name):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((peer_ip, peer_port))

            # Send a request to download a chunk
            s.sendall(b'REQUEST_CHUNK')
            # Receive acknowledgement from the peer
```

```
        ack = s.recv(1024)
        # Check if the acknowledgement is correct, if not we display error
        if ack != b'REQUEST_ACK':
            print(f"{peer_ip}:{peer_port} didn't acknowledge chunk request.
            Got: {ack}")
            return False

        # Send the chunk name to the peer
        s.sendall(chunk_name.encode())
        # Define the path to save the chunk
        save_path = os.path.join(DOWNLOAD_DIR, chunk_name)
        # Open the file in write mode
        with open(save_path, 'wb') as f:
            # Receive the first part of the chunk data
            first_data = s.recv(1024)
            # Check if the chunk was not found on the peer
            if first_data == b'NOT_FOUND':
                # Print error in that case
                print(f"{chunk_name} not found on {peer_ip}:{peer_port}")
                # Remove the empty file
                os.remove(save_path)
                return False


            # Otherwise, it is the first chunk of data and we write it
            f.write(first_data)

            # Continue receiving data until not data when we break
            while True:
                try:
                    data = s.recv(1024)
                    if not data:
                        break
                    f.write(data)
                except:
                    break

        # Verify the chunk is not empty and if it empty, we return false
        if os.path.getsize(save_path) == 0:
            print(f"Downloaded empty chunk: {chunk_name}")
            os.remove(save_path)
            return False

        # Return True if the chunk was downloaded successfully.
        print(f"Downloaded: {chunk_name} from {peer_ip}:{peer_port}
        ({os.path.getsize(save_path)} bytes)")
        return True

    except Exception as e:
        # Return False if the chunk was downloaded successfully.
        print(f"Error downloading {chunk_name} from {peer_ip}:{peer_port} { {e}")
```

```
        return False
```

4.6.2 File Reconstruction:

To reconstruct the original file, Bob uses the chunks he has successfully downloaded from the peers. The following two functions were used to achieve this:

- `get_file_metadata()`: This function retrieves the metadata of the original file (original filename and extension) from a metadata file, which was generated by Alice when the file was initially split into chunks.
- `reconstruct_file()`: Using the metadata and the downloaded chunks, this function reconstructs the original file by concatenating all the chunks in the correct order.

These two functions work together to ensure that Bob can accurately recreate the file from the pieces he has downloaded from multiple peers.

Code for `get_file_metadata()`

```
def get_file_metadata():
    # Reads the file metadata from 'file_metadata.json' we created
    metadata_path = os.path.join(DOWNLOAD_DIR, "file_metadata.json")
    # If it does not exist, we display error
    if not os.path.exists(metadata_path):
        print("No metadata file found. Cannot determine original file extension.")
        return None

    try:
         # Open the metadata file in read mode
         with open(metadata_path, 'r') as f:
             # Load the JSON data
            metadata = json.load(f)
            # Print it and return the metadata
            print(f"Read metadata: {metadata}")
            return metadata
    except Exception as e:
        # Print an error message if an exception occurs
        print(f"Error reading metadata: {e}")
        return None
```

Code for `reconstruct_file()`

```
def reconstruct_file():
    # Get metadata for original filename and extension
    metadata = get_file_metadata()

    # We check if metadata exists and display error in case it doesnt
    if not metadata:
        print("Missing metadata - attempting to reconstruct without file extension")
        output_path = OUTPUT_FILE
    else:
        # Use original extension from metadata
        extension = metadata.get('extension', '')
        # Get the original filename from the metadata
```

```
    original_filename = metadata.get('original_filename', 'reconstructed_file')
     # Get output file path using the original filename
    output_path = os.path.join(DOWNLOAD_DIR, original_filename)
    print(f"Using original filename: {original_filename}")

# Find all chunk files and sort them (excluding the metadata file)
chunk_files = sorted([f for f in os.listdir(DOWNLOAD_DIR)
                      if f.startswith("chunk_") and f.endswith(".part")])

if not chunk_files:
    print("No chunks found for reconstruction!")
    return False

# Print the number of chunk files found.
print(f"Found {len(chunk_files)} chunk files for reconstruction: {chunk_files}")

# Reconstructing the file
# Open the output file in write mode
with open(output_path, "wb") as output:
    # Iterate over each chunk file
    for chunk_name in chunk_files:
        # Define the path to the chunk file
        chunk_path = os.path.join(DOWNLOAD_DIR, chunk_name)
        print(f"Adding {chunk_name} to reconstructed file
        ({os.path.getsize(chunk_path)} bytes)")
        # Open the chunk file in read
        with open(chunk_path, "rb") as chunk_file:
            # Write the chunk data to the output file
            output.write(chunk_file.read())

# Check if the reconstructed file is empty and display error if it is
if os.path.getsize(output_path) == 0:
    print("Warning: Reconstructed file is empty!")
    return False

# Print a success message
print(f"\nReconstructed file saved as: {output_path}
({os.path.getsize(output_path)} bytes)")
return True
```

5 Testing and Results

To test the system, all .py files for each of the tracker, Alice, and the peers are run first so that Alice can share the files after splitting to the peers. The screenshots from the terminal illustrate this:

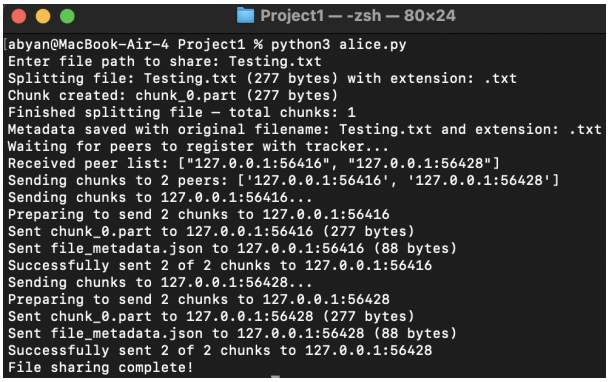

Figure 1: alice.py

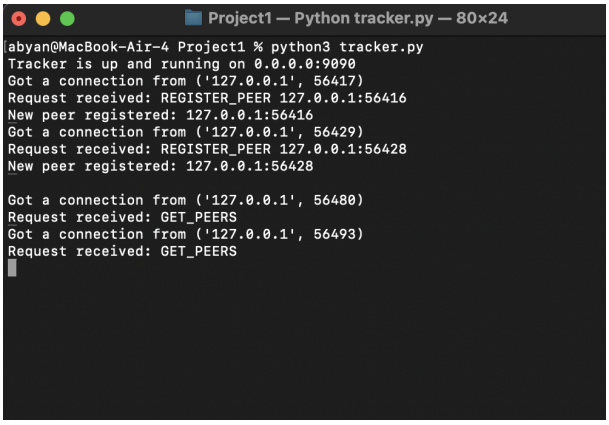

Figure 2: tracker.py

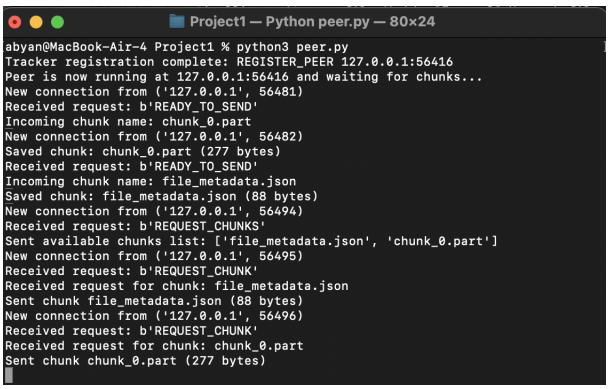

Figure 3: peer.py

For testing the functionality of the file-sharing system, three different types of files are used to ensure the system handles different scenarios effectively.

1. Small Text File (1 KB)

This file contains some small text information, allowing verification that the file is properly split into chunks, chunks are sent to peers and received correctly and the original file is accurately reconstructed. The small size (1 KB) made it easy to quickly test and compare the original and reconstructed files to check if the system was working as expected.

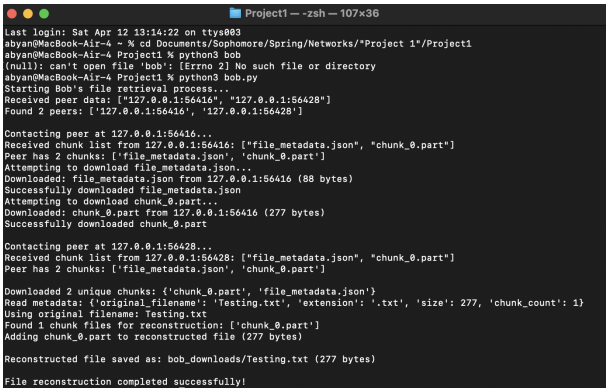

Figure 4: bob receiving the .txt file

2. PDF Document (5.4 MB)

Next, a larger PDF document is used to test how the system handles larger files. This file (5.4 MB) required splitting into multiple chunks, which tested the system's ability to split a large file into multiple chunks, transfer chunks to multiple peers and retrieve chunks from multiple peers and reconstruct the file.

Figure 5: bob receiving the .pdf file

3. Image File (16 KB)

Lastly, an image file was used to ensure the system works with not just .txt files, but files with .jpeg extensions as well that contain binary data. This test ensured that chunks of the binary data (image data) are correctly transmitted without corruption.

Figure 6: bob receiving the .jpeg file

6 Conclusion

The project began with a basic system architecture and the goal of sending a single file over the network, using file chunking and the peer-to-peer architecture. At the end, a fully functioning network is created, with a sender, receiver and a tracker allowing any number of peers to connect, register themselves on the network, and start sending and receiving files.

Throughout the development of this project, special attention was given to ensuring a smooth and user-aware experience. The system was designed to provide clear updates at each stage of the file-sharing process. The users are notified when peers are contacted, confirm chunk downloads, and indicate successful reconstruction. Robust error handling was implemented by using try-except blocks across the modules to prevent crashes and provide meaningful feedback when issues arise. To support a wide range of file types, a metadata file was introduced to preserve important information like the original file name and extension, ensuring the reconstruction process restores the file in its intended format.

The challenges included setting up the basic framework for the network and configuring the connections between the tracker and other peers on the network. The ports and gateways for communication needed to be clearly defined, as well as a way to assign ports to newly connected peers. Figuring out an effective method to split and reconstruct the files shared over the network also required some creative thinking, however iterating through the files and creating new chunks and then reversing the process during file reconstruction, as well as storing and transmitting the metadata of the file helped to reach a solution.

There could be some improvements in terms of giving peers additional functionality to create their own files etc. but that is beyond the scope of this project.