DSCI 551 – Spring 2023

Project Final Report

EDFS - Emulating HDFS

Group No. 43
Team Members:

Name	Email	USC ID
Snigdha Chenjeri	chenjeri@usc.edu	2159139179
Harsh Parikh	harshjin@usc.edu	2348613152
Aman Kumar	kumarama@usc.edu	8562322021

Project Link:

https://drive.google.com/drive/folders/1u-1O11OmZj7EXhnh2OhCTli9uMtosHCM?usp=sharing

Video Link:
https://youtu.be/HxFM4vY_vL4

Contents:

Project Description

HDFS (Hadoop Distributed File System) is a distributed storage system designed to store large datasets on a cluster of commodity hardware. HDFS stores data as blocks, typically 128 MB in size, and distributes replicas of each block across multiple nodes in the cluster for reliability. It has two main components: the Namenode, which manages the metadata for the file system, and the Datanode, which stores the actual data blocks.

This project aims to build an Emulated Distributed File System (EDFS) that emulates the working of HDFS. It will contain one metadata server (Namenode) and multiple data servers (Datanodes). A DFS client will accept and execute shell commands including ls, rm, put, get, mkdir, rmdir, cat and many more. A web browser-based UI will allow users to upload and download files from EDFS and explore the directory structure and the content of each file on EDFS.

Objectives

1. Create an EDFS client.
2. Create the NameNode server and DataNode servers.
3. Establish a connection between the client and servers.
4. Implement the functionalities of writing a file according to the replica placement policy, reading the contents of a file and deleting a file/directory from EDFS.
5. Maintain and regularly update metadata on the NameNode.
6. Accept and execute shell commands on the EDFS client.
7. Create a web user interface that lets users upload and download files from EDFS, view the directory structure and the contents of a block of a file.

Planned Implementation

- Constructing a Metadata Server - This will be used for storing metadata about the file system, which includes mapping of files to blocks and their locations on the data nodes.
- Constructing Data Servers - These will be used to store file's system data blocks.
- Client - Server Architecture needs to be established.
- File Splitting - The EDFS system should split files into blocks if they exceed a predefined size and store the blocks on different data servers.
- A DFS client to be implemented - Accept shell commands, similar to HDFS, including: ls, rm, put, get, mkdir, rmdir, cat. The client should communicate with the metadata server to retrieve information about the file system and perform operations on the data servers.
- Ensure Reliability and Data Handling through Replication.
- Maintain Data Consistency through Failure Handling.

- Creation of a Web Interface [UI] - To explore EDFS file structure and directory structure, along with the features to upload/ download the file and view its content.

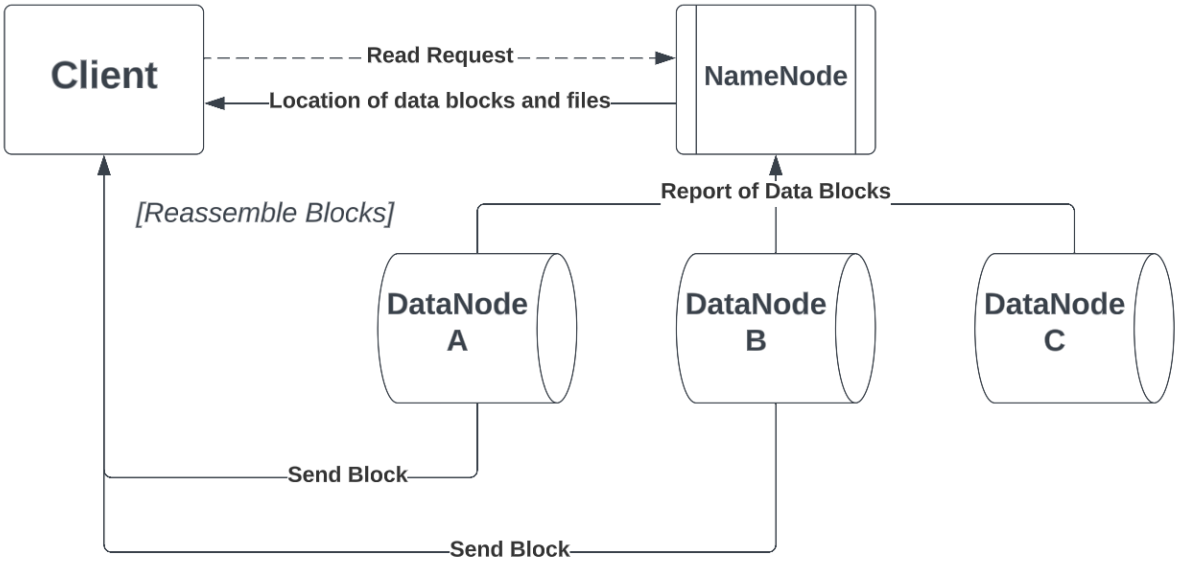


Figure 1: Reading a file in EDFS

To read a file in EDFS, the client first sends a request to the NameNode with the name of the file to be read. The NameNode fetches the locations where blocks of the file are stored and sends them to the client. The client then requests the DataNodes for the file blocks. Upon receiving the required data, the client reassembles all the blocks of the file. To read a directory, the same process is repeated for each file in the directory.

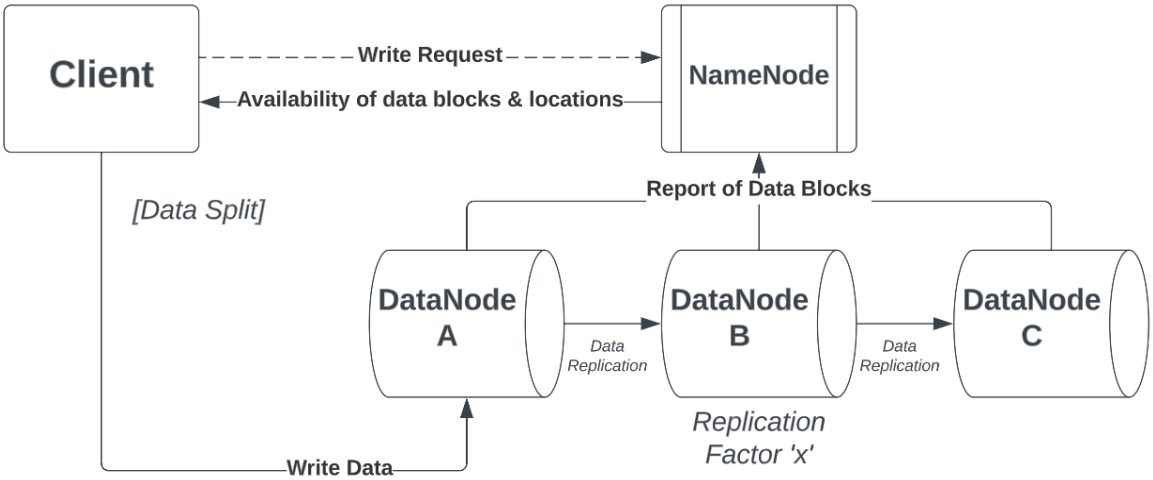


Figure 2: Writing a file in EDFS

To write a file in EDFS, the client sends a request to the NameNode. The NameNode checks the file size, and if it is greater than the block size, it fetches the number of blocks, replicas and

DataNodes where the replicas are to be stored. The client then splits the data into chunks and writes the chunks as separate blocks to the designated DataNodes. The client then informs the NameNode of the location of the block replicas. The Namenode then updates its metadata to reflect the new block locations and adds the block to the file's list of blocks.

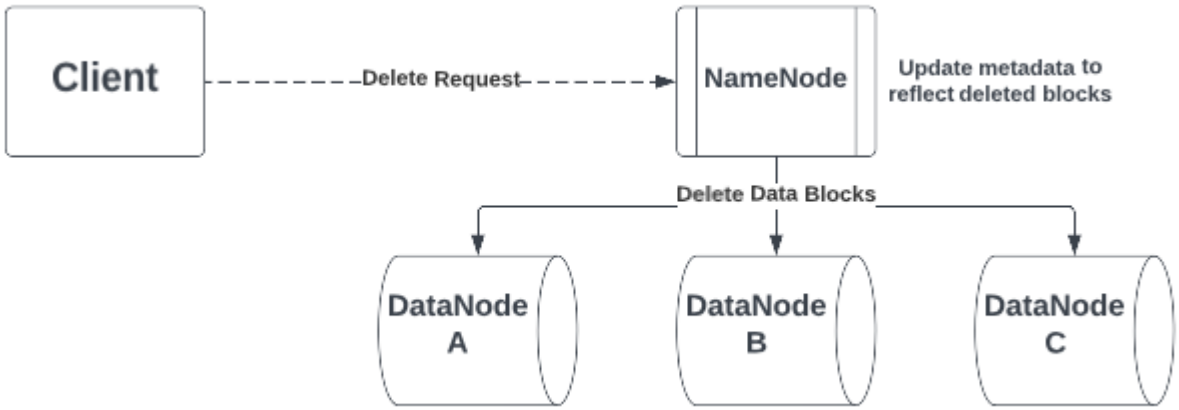


Figure 3: Deleting a file in EDFS

To delete a file, the client sends a delete request to the NameNode. The NameNode updates its metadata to mark the file and its blocks as deleted. The NameNode sends a command to each DataNode that contains a replica of a block belonging to the deleted file, instructing it to delete the block. The Datanodes delete the blocks from their local storage.

Client.py:

client.py is the client-side script that provides the interface for users to interact with the EDFS. The Client class contains methods to handle all of the EDFS commands, such as ls, rm, put, get, mkdir, rmdir, cat, etc. Here's a breakdown of the code:

1. **__init__(self, ip, port, namenode_port)**: The constructor initializes the Client object with the IP address, port, and the port on which the Namenode is running.
2. **run(self)**: This method runs an infinite loop, taking user input and executing the corresponding commands.
3. **execute_command(self, command)**: Parses the user input, calls the appropriate method based on the command, and validates the command's arguments.
4. **send_to_namenode(self, command)**: Sends a command to the Namenode via a socket connection.
5. **send_to_datanode(self, command, dn_port)**: Sends a command to a specific Datanode via a socket connection.
6. **split_file(self, src_file_path, src_file_name, partition_size, splits_dir)**: Splits a file into partitions based on the specified partition size and saves the partitions to a directory.
7. **put(self, src, dst)**: Uploads a file from the local machine to the EDFS.
8. **ls(self, path)**: Lists the contents of a directory in the EDFS.

9. **rm(self, path)**: Deletes a file in the EDFS.
10. **mkdir(self, path)**: Creates a new directory in the EDFS.
11. **rmdir(self, path)**: Deletes a directory in the EDFS.
12. **get(self, file_path, local_path)**: Downloads a file from the EDFS to the local machine.
13. **cat(self, file_path)**: Displays the content of a file in the EDFS on the terminal.
14. **get_blocks_metadata(self, file_path)**: Retrieves metadata for the blocks of a file, used for the client UI.
15. **get_block_content(self, block_id, datanode_port)**: Retrieves the content of a specific block, used for the client UI.

At the end of the script, the Client object is created and its **run** method is called to start the command prompt, allowing the user to interact with the EDFS.

```python
# send commands to namenode over socket
def send_to_namenode(self, command):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect(("localhost",self.namenode))
            s.sendall(json.dumps(command).encode())
            response = s.recv(4096)
            return json.loads(response)
    except Exception as e:
        print(f"Error sending command to NameNode: {e}")
        return {"status": "error", "message": str(e)}

# send commands to datanode over socket
def send_to_datanode(self, command, dn_port):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect(("localhost",dn_port))
            s.sendall(json.dumps(command).encode())
            response = s.recv(4096)
            return json.loads(response)
    except Exception as e:
        print(f"Error sending command to DataNode: {e}")
        return {"status": "error", "message": str(e)}

# split a file into partitions by size
def split_file(self, src_file_path, src_file_name, partition_size, splits_dir):
    with open(src_file_path, 'r') as f:
        file_contents = f.read()
    num_partitions = (len(file_contents) + partition_size - 1) // partition_size

    for i in range(num_partitions):
        # slice the file contents based on the indices
        start_index = i * partition_size
        end_index = (i + 1) * partition_size
        partition_contents = file_contents[start_index:end_index]
        # write
        partition_path = splits_dir+"/"+src_file_name[:src_file_name.index(".")]+"_"+str(i+1)+"."+src_file_name[src_file_name.index(".")+1:]
        with open(partition_path, 'w') as partition_file:
            partition_file.write(partition_contents)
```

```python
# command ls - list contents of a directory
def ls(self, path):
    namenode_response = self.send_to_namenode({"command":"ls", "path":path})
    if namenode_response["status"]=="error":
        print(namenode_response["message"])
        return 0
    elif namenode_response["status"]=="success":
        print()
        for item in namenode_response["list"]:
            print(item)
        print()
        return 1

# command rm - delete a file
def rm(self, path):
    namenode_response = self.send_to_namenode({"command":"rm", "path":path})
    print(namenode_response["message"])
    return 0 if namenode_response["status"]=="error" else 1

# command mkdir - create a directory
def mkdir(self, path):
    namenode_response = self.send_to_namenode({"command":"mkdir", "path":path})
    print(namenode_response["message"])
    return 0 if namenode_response["status"]=="error" else 1

# command rmdir - delete a directory
def rmdir(self, path):
    namenode_response = self.send_to_namenode({"command":"rmdir", "path":path})
    print(namenode_response["message"])
    return 0 if namenode_response["status"]=="error" else 1

# command get - download file from edfs to local machine
def get(self, file_path, local_path):
    namenode_response = self.send_to_namenode({"command":"get", "file_path":file_path})
    if namenode_response["status"]=="error":
        print(namenode_response["message"])
        return 0
```

The motive of the **client.py** file is to provide an interface for users to interact with the EDFS (Emulated Distributed File System) by executing various commands like listing directory contents, uploading and downloading files, creating and deleting directories, and more. The EDFS is a distributed file system designed to store and manage data across multiple DataNodes, and the **client.py** file serves as the entry point for users to interact with the system.
The client-side script plays a crucial role in the client-server architecture of the EDFS. Here are some key responsibilities of the client:

1. User Interface: It provides a command-line interface for users to input commands and interact with the EDFS.
2. Command Parsing: The client parses user input and calls the appropriate method based on the command and its arguments.
3. Communication with the NameNode: The client communicates with the NameNode to obtain metadata information, like file locations and DataNode addresses. The NameNode is responsible for managing the EDFS namespace, and the client relies on it for all metadata operations.
4. Communication with DataNodes: The client directly interacts with DataNodes to read and write file data. It establishes socket connections with DataNodes to send and receive file data, block information, and other necessary commands.
5. File Splitting and Merging: The client handles file splitting when uploading large files to the EDFS, dividing them into smaller blocks to be stored across multiple DataNodes. Similarly, it takes care of merging file blocks when downloading or displaying the content of a file.

In summary, the client in **client.py** serves as a bridge between the user and the EDFS, facilitating the execution of various file system commands and managing the communication between the user, the NameNode, and the DataNodes.

<u>Talking about the ‘put’ and get’ commands:</u>

The **put** and **get** commands in the **client.py** script are used to upload and download files to and from the Emulated Distributed File System (EDFS), respectively.

1. **put**: The **put** command is used to upload a file from the local machine to the EDFS. The process consists of the following steps:

a. Obtain the file size and send a write request to the NameNode with the file path and file size.
b. The NameNode returns a list of locations (DataNodes and block IDs) where the file or its partitions should be stored, along with the default EDFS block size to help split the file if necessary.
c. If the file size is less than or equal to the block size, no file splitting is required. Read the file, encode it in base64, and send it to each replica location specified by the NameNode.

d. If the file size is larger than the block size, split the file into partitions based on the block size. Read each partition, encode it in base64, and send it to each replica location specified by the NameNode.
e. After uploading the file or its partitions to the DataNodes, inform the NameNode to update its metadata by sending the locations and block sizes of the newly saved file.

2. **get**: The **get** command is used to download a file from the EDFS to the local machine. The process consists of the following steps:

a. Send a read request to the NameNode with the file path.
b. The NameNode returns a list of blocks associated with the file, including the partition number, block ID, and DataNode address for each block.
c. For each partition, iterate through the replicas and try to download the block from the corresponding DataNode. Break the loop once a successful download occurs for each partition.
d. Decode the base64-encoded data received from the DataNodes and concatenate the partitions in the correct order.
e. Write the concatenated file content to the specified local path.
The **put** and **get** commands enable users to interact with the EDFS by uploading and downloading files, while ensuring the data is distributed across multiple DataNodes to enhance fault tolerance and improve load balancing.

Namenode.py:

This **namenode.py** script implements the NameNode class, which is the central component of the Emulated Distributed File System (EDFS) that stores the file system's metadata and manages file operations. The NameNode listens to client requests and communicates with DataNodes to perform tasks such as file creation, deletion, and retrieval.

The **NameNode** class has several key methods that handle client requests:

1. **start()**: Starts the NameNode server and listens for incoming client connections. It accepts connections and processes them by calling the **handle_client()** method.
2. **handle_client()**: Receives client requests and processes the commands by calling the **process_command()** method.
3. **process_command()**: Parses the command received from the client and delegates the task to the appropriate method based on the command type (e.g., put, get, ls, rm, mkdir, rmdir).
4. **send_to_datanode()**: Sends a command to a specified DataNode and returns the response.

The NameNode class includes several methods for file operations:

1. **write_new_file()**: Handles file creation by calculating the number of partitions, generating block IDs, and selecting DataNodes to store each partition.
2. **write_new_file_update_metadata()**: Updates the metadata of a newly created file, including file size, locations, and block sizes.

3. **ls()**: Lists the contents of a directory.
4. **remove_file()**: Deletes a file from the file system and updates the metadata.
5. **make_directory()**: Creates a new directory.
6. **remove_directory()**: Removes an existing directory.

The script also contains a few methods for retrieving file and block information:

1. **get_block_locations()**: Returns the block locations for a file.
2. **blocks_metadata()**: Returns the block metadata for a file.

When the script is executed, it reads the command-line arguments for the NameNode IP, NameNode port, and DataNode ports. It then creates an instance of the NameNode class and starts the server by calling the **start()** method.

```
# start listening on namenode port
def start(self):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((self.ip, self.port))
        s.listen()
        print(f"NameNode started at {self.ip}:{self.port}")
        try:
            while True:
                conn, addr = s.accept()
                print(f"Connection from {addr}")
                self.handle_client(conn)

        except KeyboardInterrupt:
            s.close()
            print("\nProgram terminated by user.")

# receive command from client
def handle_client(self, conn):
    with conn:
        data = conn.recv(4096)
        command = json.loads(data)
        response = self.process_command(command)
        conn.sendall(json.dumps(response).encode())
```

The motive of the **namenode.py** file is to implement the NameNode server, which is an essential component of a distributed file system like Emulated Distributed File System (EDFS). The NameNode server is responsible for managing the file system's metadata and coordinating file operations.

The NameNode server plays several important roles in the distributed file system:

1. Metadata management: The NameNode stores and maintains metadata information about files and directories, such as file size, block locations, and directory structure. This information is used to coordinate file operations and ensure data consistency across the system.
2. File operations coordination: The NameNode processes client requests for file operations, such as creating, deleting, or retrieving files. It interacts with the DataNodes to perform these operations and ensure that the data is stored or retrieved correctly.
3. Load balancing and replication: The NameNode is responsible for distributing data across the DataNodes in the system. It selects which DataNodes to store each file partition and ensures that the data is replicated according to the configured replication factor.

4. Fault tolerance and recovery: The NameNode helps in maintaining system reliability by detecting DataNode failures and managing data replication to ensure that the file system can recover from failures and continue functioning.

Overall, the NameNode server is the central point of control and coordination for the distributed file system. It manages the file system's metadata, oversees file operations, and maintains system reliability and performance.

```python
# updates metadata
def write_new_file_update_metadata(self, file_path, file_size, locations, block_sizes):
    self.file_metadata[file_path] = {
        "rf": 2,
        "size": file_size,
        "blocks": []
    }
    p = 0
    for partition in locations:
        block_size = block_sizes[str(p)]
        p+=1
        for replica in partition:
            self.file_metadata[file_path]["blocks"].append({
                "id":replica[1],
                "partition": p,
                "datanode":replica[0],
                "num_bytes": block_size
            })

    parent_path = file_path[:file_path.rfind('/')]
    if parent_path=="":
        parent_path="/"
    file_name = file_path[file_path.rfind('/')+1:]
    self.directory_metadata[parent_path]["children"].append(file_name)

    self.save_metadata()    # saving metadata.json
    return {"command":"put_update", "status":"success", "message":"File uploaded successfully"}

def save_metadata(self):
    data = {"file":self.file_metadata, "dir":self.directory_metadata}
    with open("metadata.json","w") as f:
        json.dump(data,f)
```

Updating metadata is namenode server's job, and this is how we have executed it.

The **write_new_file_update_metadata** function in **namenode.py** is responsible for updating the metadata after a new file is written to the EDFS. The function takes the following parameters:

1. **file_path**: The path of the new file in the file system.
2. **file_size**: The size of the new file.
3. **locations**: The list of DataNode locations where the file blocks are stored.
4. **block_sizes**: A dictionary containing the sizes of each file block.

The function performs the following steps:

1. It adds a new entry for the file in **self.file_metadata** with the following information:
    - Replication factor (rf): The number of replicas for each block of the file.
    - Size: The size of the file.
    - Blocks: An initially empty list that will be filled with block information.
2. It iterates through the **locations** list, which contains the DataNode locations and unique block IDs for each file block. For each partition (block), it updates the **blocks** list in the file metadata with the following information:
    - ID: The unique block ID.
    - Partition: The partition number.
    - DataNode: The DataNode port where the block is stored.
    - Num_bytes: The size of the block.

3. It updates the **self.directory_metadata** to include the new file in the parent directory's list of children.
4. The **save_metadata** function is called to save the updated metadata to the **metadata.json** file.
5. Finally, the function returns a success message indicating that the file has been uploaded successfully and the metadata has been updated.

By updating the metadata, the NameNode ensures that it has accurate information about the file system, enabling it to manage file operations, replication, and recovery effectively.

By saving the metadata to a JSON file, the NameNode ensures that the file system state is persisted and can be recovered even after the NameNode process is stopped or restarted. This is important for maintaining a consistent view of the EDFS and ensuring the availability and reliability of the distributed file system.

Datanode.py:

This script implements a DataNode class in a simple distributed file system, similar to HDFS. DataNodes are responsible for storing the actual data blocks and handling read and write operations.

The **DataNode** class has the following attributes:

- **ip**: IP address of the DataNode
- **port**: Port on which the DataNode listens for incoming connections
- **blocks_available**: Number of blocks available for storing data
- **storage_path**: Directory where the data blocks are stored

The main methods in the **DataNode** class are:

1. **start()**: This method starts the DataNode, setting up a socket to listen for incoming connections. It runs in an infinite loop and accepts connections from clients. When a connection is established, it calls the **handle_command()** method to process the request.
2. **handle_command(conn)**: This method receives a command from the client, decodes it, and calls the **process_command()** method to execute the corresponding operation. It then sends the response back to the client.
3. **process_command(command)**: This method processes the command received from the client. Depending on the command type, it calls the appropriate method to handle the request, such as **write_new_file()**, **get_file_content()**, or **remove_file()**.
4. **write_new_file(block_id, data)**: This method writes a new data block to the DataNode. It takes the block ID and data, decodes the data, and writes it to a file with the specified block ID in the storage directory.
5. **get_file_content(block_id)**: This method reads the content of a data block. It takes the block ID as input, reads the corresponding file from the storage directory, and returns the content as a base64-encoded string.
6. **remove_file(block_id)**: This method removes a data block from the DataNode. It takes the block ID as input and deletes the corresponding file from the storage directory.

At the end of the script, the main function checks if the correct number of command-line arguments are provided (IP and port). If so, it creates an instance of the **DataNode** class and starts it.

```
def write_new_file(self, block_id, data):
    data = base64.b64decode(data)
    local_file_path = self.storage_path+"/"+str(block_id)
    with open(local_file_path, "wb") as f:
        f.write(data)
    return {"command":"put", "status":"success","message":"Successfully written on DataNode "+str(self.port)}

def get_file_content(self, block_id):
    local_file_path = self.storage_path+"/"+str(block_id)
    try:
        with open(local_file_path, "rb") as f:
            block = f.read()
        # base64 encoding required to send file over socket
        block_b64 = base64.b64encode(block).decode()
        return {"command":"get", "status":"success","block":block_b64}
    except:
        return {"command":"get", "status":"error","message":"Block not found on datanode"+str(self.port)}

def remove_file(self, block_id):
    local_file_path = self.storage_path+"/"+str(block_id)
    try:
        os.remove(local_file_path)
        print(f"File '{local_file_path}' deleted successfully.")
        return {"command":"rm", "status":"success","message":"Deleted on DataNode "+str(self.port)}
    except OSError as e:
        {"command":"rm", "status":"error", "message":f"Failed to delete file '{local_file_path}': {e}"}
```

1. **write_new_file(self, block_id, data)**: This method is responsible for writing a new data block to the DataNode's storage.
    - **block_id**: The ID of the data block to be written.
    - **data**: The data to be written, encoded as a base64 string.

The method first decodes the base64-encoded data using **base64.b64decode(data)**. Then, it constructs the path to the local file where the data block will be stored, using the DataNode's **storage_path** and the **block_id**.
The method opens the file in binary write mode (**"wb"**), and writes the decoded data to the file. After writing the data, the method returns a dictionary with the command status and a success message.

2. **get_file_content(self, block_id)**: This method reads the content of a data block stored on the DataNode.
    - **block_id**: The ID of the data block to be read.

The method first constructs the path to the local file containing the data block, using the DataNode's **storage_path** and the **block_id**. Then, it tries to open the file in binary read mode (**"rb"**).
If the file is successfully opened, the method reads the entire content of the file into a variable named **block**. The content is then base64-encoded using **base64.b64encode(block).decode()** to prepare it for transmission over a socket.
The method returns a dictionary with the command status, success message, and the base64-encoded content of the block. If an error occurs (e.g., the file is not found), the method returns a dictionary with an error status and an error message.

3. **remove_file(self, block_id)**: This method deletes a data block from the DataNode's storage.
    - **block_id**: The ID of the data block to be deleted.

The method first constructs the path to the local file containing the data block, using the DataNode's **storage_path** and the **block_id**. Then, it tries to delete the file using the **os.remove(local_file_path)** function.
If the file is successfully deleted, the method prints a success message and returns a dictionary with the command status and a success message. If an error occurs (e.g., the file is not found or cannot be deleted), the method returns a dictionary with an error status and an error message, including the details of the **OSError.**

Metadata.json:

The initial structure of **metadata.json** contains only the root directory ("/") with no parent (denoted by **0**) and no children (an empty list). As new files and directories are added to the distributed file system, the **metadata.json** file will be updated accordingly.

In this project, the metadata plays a crucial role as it helps the NameNode manage the distributed file system by keeping track of:

- The hierarchical structure of directories and files in the file system.
- The relationships between files, directories, and their parents/children.
- The locations of the data blocks on the DataNodes for each file, which enables the NameNode to locate and manage the data blocks as needed (e.g., reading, writing, replicating, or deleting blocks).

In summary, the metadata in the **metadata.json** file is essential for the efficient management and operation of the distributed file system.

App.py:

The **app.py** file is a Flask web application for managing the distributed file system. It provides a web interface to interact with the NameNode and DataNodes, allowing users to perform operations like navigating directories, reading and writing files, and viewing file blocks.

Here's an overview of the key components in **app.py**:

1. **ui_client**, **ui_namenode**, and **ui_datanodes**: These objects are instances of the **Client**, **NameNode**, and **DataNode** classes, respectively. They facilitate communication between the Flask web application and the backend distributed file system components.

2. **app**: This is an instance of the Flask web application.
3. **is_folder**, **get_separate_files_and_folders**: These utility functions help to process and organize file and folder information in the metadata.
4. Routes and their corresponding functions:
    - **@app.route('/', methods=['GET', 'POST'])**: The home route that displays the root directory's contents.
    - **@app.route('/folder/<parent>/<folder_name>', methods=['POST', 'GET'])**: Displays the contents of a specified folder.
    - **@app.route('/file/<parent>/<file_name>', methods=['POST', 'GET'])**: Displays the contents of a specified file.
    - **@app.route('/upload', methods=['POST'])**: Uploads a file to the distributed file system.
    - **@app.route('/download', methods=['POST'])**: Downloads a file from the distributed file system.
    - **@app.route('/blocks', methods=['POST'])**: Displays the block information and content for a specified file.

In summary, **app.py** is a Flask web application that provides a user interface for managing the distributed file system. It communicates with the NameNode and DataNodes to perform operations like creating, reading, and deleting files and directories.

```python
# Upload a file (command: put)
@app.route('/upload', methods=['POST'])
def upload():
    parent = request.form.get('path')
    parent = parent.replace('-','/')
    if parent=="root":
        parent = "/"
    else:
        parent = "/"+parent.replace("root/","")+"/"

    files = request.files.getlist('file[]')
    for file in files:
        edfs_path = parent+file.filename
        file_content = file.read()

        file_name = file.filename[:file.filename.rfind(".")]
        file_ext = file.filename[file.filename.rfind(".")+1:]
        temp_file = file_name+"_temp."+file_ext

        with open(temp_file, 'wb') as f:
            f.write(file_content)
        ui_client.execute_command("put "+temp_file+" "+edfs_path)
        os.remove(temp_file)

    if parent=="/":
        return redirect(url_for("home"))
    parent = ("root/"+parent[1:-1]).replace("/","-")
    return redirect(url_for("folder", parent=parent[:parent.rfind("-")], folder_name=parent[parent.rfind("-")+1:]))


# Download a file (command: get)
@app.route('/download',methods=['POST'])
def download():
    edfs_path = request.form.get('parent').replace('-','/')
    file_name = request.form.get('file_name')
    local_path = str(os.getcwd())+"/"+file_name
    ui_client.execute_command("get "+edfs_path+" "+local_path)
    parent = "root" if edfs_path[:edfs_path.rfind("/")]=="" else ("root"+edfs_path[:edfs_path.rfind("/")]).replace("/","-")

    return redirect(url_for("file", parent=parent, file_name=file_name,dic=local_path))

# View file blocks
@app.route('/blocks',methods=['POST'])
def blocks():
    edfs_path = request.form.get('parent').replace('-','/')
    file_name = request.form.get('file_name')
    blocks_metadata = ui_client.get_blocks_metadata(edfs_path)

    content = "<hr>\n\n"
    for b in blocks_metadata:
        content = content + "Partition: " + str(b["partition"]) + "\nSize: " + str(b["num_bytes"]) + "\nDatanode: " + str(b["datanode"]) + "\nBlock ID: " + b["id"]
        content = content +ui_client.get_block_content(b["id"],b["datanode"])
        content = content + "\n\n<hr>\n\n"
    content = content.replace('\n', '<br>')

    return render_template('display_blocks.html',path=edfs_path,file_name=file_name, content=content)
```

These three functions are the route handlers for uploading, downloading, and viewing file blocks in the Flask web application:

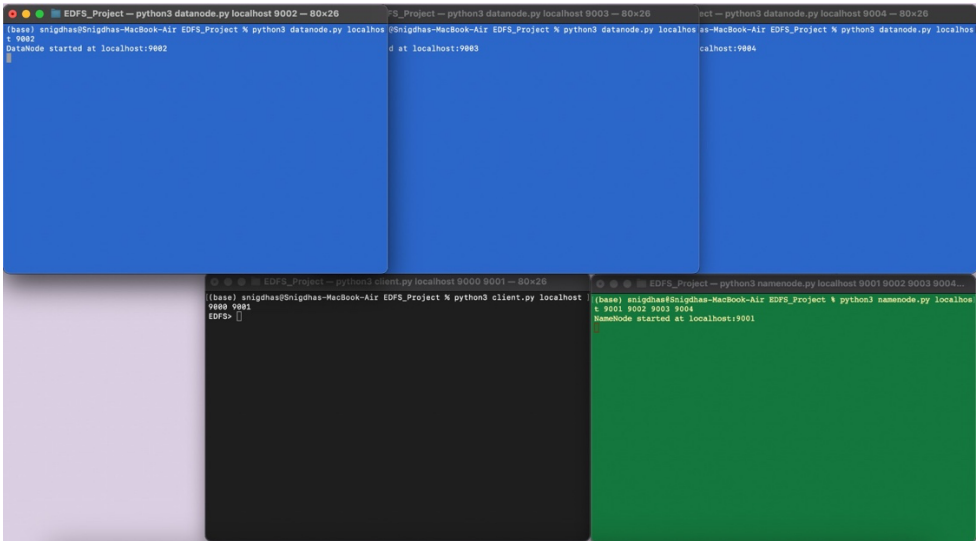
1. **upload()**: This function handles the file upload (PUT) operation. It first retrieves the parent path from the form data and normalizes it to construct the correct EDFS path. Then, it iterates over the uploaded files, reads their content, and creates a temporary file on the local system. It then calls the **ui_client.execute_command()** method to upload the temporary file to the distributed file system with the specified EDFS path. After the upload is complete, the temporary file is removed from the local system. Finally, the function redirects the user to the appropriate folder view based on the parent path.

2. **download()**: This function handles the file download (GET) operation. It first retrieves the EDFS path and file name from the form data and constructs the local path for saving the downloaded file. Then, it calls the **ui_client.execute_command()** method to download the file from the distributed file system and save it to the local path. Finally, it redirects the user to the file view page with the downloaded file's content.

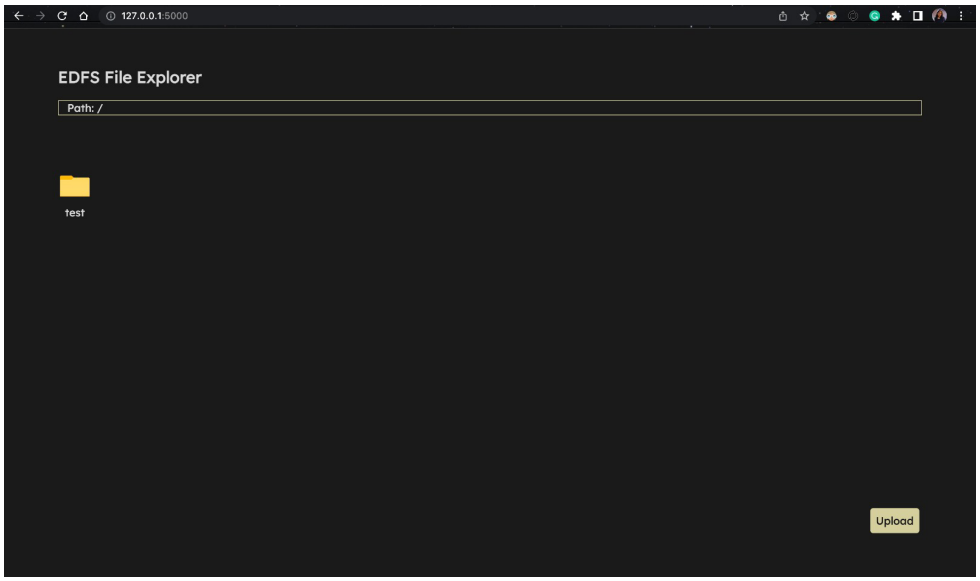
3. **blocks()**: This function handles viewing the file blocks and their metadata. It first retrieves the EDFS path and file name from the form data and then calls the **ui_client.get_blocks_metadata()** method to get the metadata of the file's blocks. It then iterates over the blocks' metadata and constructs an HTML string containing block information such as partition number, size, DataNode, block ID, and the block content. The block content is fetched using the **ui_client.get_block_content()** method. Finally, the function returns the **display_blocks.html** template with the constructed HTML content.

In summary, these three functions provide the functionality for uploading, downloading, and viewing file blocks in the distributed file system through the Flask web application. They communicate with the Client, NameNode, and DataNodes to perform the required operations and present the results to the user.

Terminal Window Layout:



Web Application User Interface:



File Structure Explanation:
Confirm 3 empty folders with names **9002**, **9003** and **9004** are present in the current working directory.

Port Numbers
Client: 9000
Namenode: 9001
Datanode 1: 9002
Datanode 2: 9003
Datanode 3: 9004
IP: localhost (127.0.0.1:5000)

To Run
Terminal 1 (Client): python3 client.py localhost 9000 9001
Terminal 2 (NameNode): python3 namenode.py localhost 9001 9002 9003 9004
Terminal 3 (DataNode 1): python3 datanode.py localhost 9002 9001
Terminal 4 (DataNode 2): python3 datanode.py localhost 9003 9001
Terminal 5 (DataNode 3): python3 datanode.py localhost 9004 9001
Terminal 6 (Flask UI): python3 app.py

Shell commands

- List (ls)

ls <path>
Example: ls /test

- Make directory (mkdir)

mkdir <path>
Example: mkdir /test

- Remove directory (rmdir)

rmdir <path>
Example: rmdir /test

- PUT

put <local_src> <edfs_dst>
Example: put hello.txt /test/hello.txt

- GET

get <edfs_dst> <local_src>
Example: get /test/hello.txt hello1.txt

- Remove (rm)

rm <edfs_dst>
Example: rm /test/hello.txt

- Concatenate (cat)

cat <path>
Example: cat /test/hello.txt

Tech Stack:

Language: *Python*
Storing Metadata: *JSON*
Web Socket Communication: *Socket Library*
Web Interface: *Flask*

Timeline and Task Distribution

Tasks	**Members**	**Timeline**
Fixing connectivity issues, file splitting, handling edge cases	Snigdha Chenjeri, Harsh Parikh, Aman Kumar	March 28 - April 11, 2023
Implementing web browser based UI	Snigdha Chenjeri, Harsh Parikh, Aman Kumar	April 11 to April 18, 2023
Final Compilation and Documentation	Snigdha Chenjeri, Harsh Parikh, Aman Kumar	April 19 to April 25, 2023
Video along with final report	Snigdha Chenjeri, Harsh Parikh, Aman Kumar	