UNIVERSITY OF TEXAS AT ARLINGTON

6324-002 ADV TOPS OF SOFTWARE ENGINEERING

ON

Cloud-Based Storage Project

PROJECT GROUP-01
MAHICHARAN V-1002156386

Submitted to:
Lei ,Yu

Introduction

This report presents the implementation details of a cloud-based storage system developed in Java as part of a software engineering project. The primary objective of this endeavor was to create a robust and scalable client-server application capable of synchronizing files across multiple clients through a centralized server. The cloud storage system was designed to facilitate seamless file management, enabling users to add, modify, and delete files locally while ensuring consistency and real-time synchronization across all connected clients. The project was structured into three distinct scopes, each building upon the previous one, culminating in an advanced system that leverages UDP (User Datagram Protocol) for efficient data transfer with support for multiple pending packets. Additionally, a bonus feature for delta synchronization was implemented to optimize the synchronization process by transferring only the modified portions of files, thereby reducing network overhead, and improving overall performance. Throughout the development process, a strong emphasis was placed on adhering to software engineering principles, employing industry-standard practices, and leveraging the capabilities of the Java programming language to deliver a robust and user-friendly solution.

Core challenge of the project implementation:

- *How to detect if a binary of text file has changed?*

One of the core challenges in implementing the cloud-based storage system was efficiently detecting when files had been modified by the user on any of the connected clients. To address this, a comprehensive file monitoring mechanism was developed. The system leveraged Java's WatchService API, which provides a scalable and efficient way to monitor file system events, including file creation, modification, and deletion. A dedicated file watcher thread was implemented to continuously monitor a designated directory where the user's files were stored. Whenever a file system event was detected, the file watcher would analyze the event type and the associated file path to determine the appropriate action. If a file was created or modified, its contents were read and prepared for synchronization with the server and other clients. Conversely, if a file was deleted, a deletion notification was propagated. To ensure real-time synchronization, the file watcher employed a highly responsive event-driven architecture, allowing it to promptly respond to user actions without introducing significant delays or system lags. This real-time file change detection mechanism formed the foundation for the cloud storage system's ability to maintain consistent and up-to-date file synchronization across all connected clients.

- *TCP communication in a local IP network:*

The cloud storage system leveraged TCP (Transmission Control Protocol) for reliable and ordered data transfer between the clients and the server within a local IP network. TCP's connection-oriented nature ensured that data packets were delivered in the correct sequence, enabling seamless file synchronization without data corruption or loss. The Java implementation utilized TCP sockets, which provide a low-level abstraction for network communication. On the server-side, a ServerSocket was created to listen for incoming client connections on a designated port. When a client attempted to connect, the server would accept the connection request, establishing a dedicated Socket for that client. This allowed multiple clients to simultaneously connect to the server, facilitating seamless file synchronization across all connected devices. The clients, on the other hand, utilized Socket instances to initiate connections with

the server, specifying the server's IP address and port number. Once a connection was established, bidirectional streams were created, enabling the client and server to exchange data, control messages, and file synchronization instructions. TCP's built-in flow control, error detection, and retransmission mechanisms ensured that data integrity was maintained throughout the synchronization process, even in the presence of network congestion or packet loss within the local network.

- *UDP communication in a local IP network:*

To facilitate efficient and low-latency file transfers, the cloud storage system employed UDP for data communication within the local IP network. Unlike TCP, UDP is a connectionless protocol that does not establish a dedicated end-to-end connection between the client and server. Instead, it operates by sending independent packets of data, known as datagrams, without guaranteeing delivery or maintaining any predefined order. This approach minimizes the overhead associated with establishing and maintaining connections, making UDP well-suited for time-sensitive applications where occasional packet loss can be tolerated.

In the Java implementation, UDP sockets were utilized to enable communication between the clients and the server. On the server-side, a DatagramSocket was created to listen for incoming datagrams on a designated port. Clients, on the other hand, instantiated DatagramSocket instances and encapsulated file data or control messages within DatagramPacket objects, specifying the server's IP address and port number as the destination. These packets were then sent via the send() method of the DatagramSocket. The server continuously monitored the designated port for incoming datagrams, processing them as they arrived. While UDP does not inherently provide reliability mechanisms, the implementation incorporated custom logic to handle packet loss and reordering, ensuring data integrity during file transfers within the local network environment.

- *How to transfer big files:*

Transferring large files over a UDP connection presents unique challenges due to the protocol's lack of inherent mechanisms for reliability, flow control, and congestion avoidance. To overcome these limitations, a custom protocol was implemented that divided large files into smaller chunks or packets for transmission. On the sender side, files were read in fixed-size blocks, which were then encapsulated into individual UDP datagrams with a sequence number and other metadata. These datagrams were sent consecutively to the receiver, leveraging UDP's low-overhead and connectionless nature for efficient data transfer.

On the receiving end, a buffer was maintained to reassemble the file from the incoming packets. As datagrams arrived, they were inspected for their sequence numbers, and their payloads were placed in the appropriate positions within the buffer. To handle packet loss or reordering, the receiver employed a sliding window mechanism, where it continuously tracked the sequence numbers of received and missing packets. If a packet was not received within a specified timeout, the receiver would send a negative acknowledgment (NAK) to the sender, requesting retransmission of the missing packet(s). This process continued until all packets were successfully received and the complete file was reconstructed in the buffer. Periodic acknowledgments were also sent to the sender, allowing it to adjust the transmission rate and window size dynamically, improving throughput and minimizing network congestion for efficient large file transfers over UDP.

Potential failure modes, and issues on TCP communication:

There are a few potential failure modes or issues that can arise with TCP communication in Java for this cloud-based storage project.

- Connection reset: The search results mention a SocketException related to "Connection reset", which can occur if the connection between the client and server is unexpectedly terminated. This could happen due to network issues, server crashes, or other problems.
- Incomplete data transfer: The search results indicate that sometimes only the first message from the client is received by the server, even though the client sends multiple messages. This suggests an issue with the full data transfer not being completed.
- Blocking I/O: The use of blocking I/O operations like InputStream.read() and DataInputStream.readUTF() can cause the server thread to get stuck waiting for data that never arrives, especially in a multi-threaded server implementation.
- Synchronization issues: With multiple clients connecting to the server, there could be synchronization problems if the server's handling of client connections and data is not properly coordinated.
- Network latency and packet loss: When running the client and server on different machines across a network, issues like high latency or packet loss could impact the reliability and performance of the TCP communication.

To address these potential failure modes, the project may need to implement robust error handling, non-blocking I/O, and proper synchronization mechanisms. Additionally, techniques like retransmission, timeouts, and buffering may be required to ensure reliable and efficient data transfer over the network.

Potential failure modes, and issues on UDP communication:

The key failure modes or issues that can arise with UDP communication include:

- Packet loss: UDP is a connectionless protocol, so there is no guarantee that packets will be delivered successfully. Packets can be lost due to network congestion, errors, or other issues.
- Out-of-order delivery: UDP does not guarantee that packets will be delivered in the same order they were sent. This can cause issues with reassembling the data on the receiving end.
- Lack of flow control: Unlike TCP, UDP does not have built-in flow control mechanisms. This means the sender can overwhelm the receiver with too many packets, leading to packet loss or delays.
- Unreliable data transfer: Since UDP does not have reliability mechanisms like retransmission, the data transferred may be incomplete or corrupted.
- Synchronization issues: With multiple clients sending UDP packets to the server, there could be challenges in properly coordinating and handling the concurrent connections and data.
- Firewall/NAT traversal: UDP communication may be blocked or affected by firewalls or network address translation (NAT) devices, which can introduce additional complexities.
- Invalid network handle or PDU ID: If an invalid network handle or PDU ID is used, the UDP Network Management (UdpNm) module may not execute the requested function and instead report an error.
- Null pointer issues: If a null pointer is passed to a UdpNm service, the function may not be executed, and an error may be reported.

To address these failure modes, the project may need to implement additional reliability mechanisms, such as custom retransmission, sequence numbers, and packet reassembly logic. Proper synchronization and error handling strategies would also be crucial for a robust UDP-based cloud storage solution.

Method and resource:

I strategically employed a variety of methods and resources to surmount challenges and drive the project towards successful completion. Embracing the Java programming language as the foundation of the project, I utilized its robust features and libraries to implement the client-server architecture and file synchronization functionalities effectively. Leveraging PyCharm IDE as my development environment, I benefited from its intuitive interface and powerful debugging tools, which streamlined the coding process and facilitated efficient troubleshooting of any Java-related issues. Furthermore, I extensively referenced online resources such as GitHub documentation, Java programming guides, and networking tutorials to deepen my understanding of TCP and UDP communication protocols. This knowledge proved instrumental in designing and implementing reliable data transfer mechanisms, ensuring seamless synchronization of files across multiple clients. By integrating practical implementation with theoretical insights gleaned from these resources, I navigated the intricacies of TCP and UDP communication, effectively mitigated potential failure modes, and ultimately delivered a robust cloud storage solution that aligned with project objectives and showcased the synergy between Java programming, PyCharm IDE, and comprehensive research.

Inside the Java daemon server code: The FileServer.java

Java code defines a file server that listens for both TCP and UDP connections. Next its main components and functionalities:

- Imports: The code imports various Java standard library packages and classes necessary for file I/O, networking, concurrency, cryptographic operations, and exception handling.
- Constants: Constants are defined for server configuration parameters such as IP addresses, port numbers, buffer sizes, directory paths, and message separators.
- Main Class: The FileServer class contains the main method, which is the entry point of the application. It creates the necessary directories using Files.createDirectories() and starts two server instances using an ExecutorService with fixed thread pools.
- UDP Server: The startUdpServer method listens for UDP packets on a specified port. It receives data packets, processes them, and saves the received file chunks to the server's directory. It also handles reassembly of file chunks, calculates file hashes, and saves hash values to indicate complete file reception.
- TCP Server: The startTcpServer method listens for TCP connections on a specified port. It accepts client connections and delegates communication handling to the handleTcpCommunication method.
- Communication Handling: The handleTcpCommunication method reads messages from TCP clients, processes them, and generates appropriate responses. It parses incoming messages, checks file integrity using hash values, identifies new or changed files, and sends responses containing file status information back to clients.

The detail of each component and functionalities:

The imports:

```java
import java.io.*;
```
This line imports all classes from the `java.io` package, which provides classes for input and output operations.

```java
import java.net.*;
```
This line imports all classes from the `java.net` package, which provides classes for networking operations, such as working with URLs, sockets, and connections.

```java
import java.nio.file.*;
```
This line imports all classes from the `java.nio.file` package, which provides classes for file I/O operations, especially those that deal with file paths, directories, and file attributes.

The import stament bolk:

```java
import java.nio.charset.StandardCharsets;
```
This line imports the `StandardCharsets` class from the `java.nio.charset` package, which provides constants for various character encoding schemes. It's commonly used for specifying character encodings when working with text data.

```java
import java.security.MessageDigest;
```
This line imports the `MessageDigest` class from the `java.security` package, which provides functionality for generating cryptographic hash functions. It's commonly used for generating hash values for data integrity verification.

```java
import java.util.*;
```
This line imports all classes from the `java.util` package, which contains various utility classes and data structures like lists, sets, maps, etc.

```java
import java.util.concurrent.*;
```
This line imports all classes from the `java.util.concurrent` package, which provides utilities for concurrent programming, such as thread pools, locks, and synchronization utilities.

```java
import java.security.NoSuchAlgorithmException;
```
This line imports the `NoSuchAlgorithmException` class from the `java.security` package, which is an exception thrown when a particular cryptographic algorithm is requested but is not available in the environment.

The class declaraction:

```java
public class FileServer {
```

It defines the beginning of a Java class named FileServer. In Java, classes are the fundamental building blocks of object-oriented programming.
public: This is an access modifier that specifies that the class FileServer can be accessed from any other class.
class: This keyword is used to declare that what follows is a class definition.
FileServer: This is the name of the class. It's conventional in Java to capitalize the first letter of each word in a class name (this is known as camelCase).

The Constants:

```
    private static final boolean VERBOSE = true;
    private static final String SERVER_IP = "127.0.0.1";
    private static final int SERVER_PORT_TCP = 9091;
    private static final int SERVER_PORT_UDP = 9999;
    private static final String SERVER_DIRECTORY = "remote_folder";

    private static final int BUFFER_SIZE = 1024;
    private static final int BLOCK_SIZE = 4 * 1024 * 1024;

    private static final String MESSAGE_SEPARATOR = "-x-xx-x-";
```

- `private static final boolean VERBOSE = true;`: This line declares a private, static, and final boolean variable named `VERBOSE` with the value `true`. It's likely used to control whether the server should output verbose logging information.
- `private static final String SERVER_IP = "127.0.0.1";`: This line declares a private, static, and final String variable named `SERVER_IP` and assigns it the value `"127.0.0.1"`. This typically represents the IP address of the server. In this case, it's set to `127.0.0.1`, which is the localhost or loopback address.
- `private static final int SERVER_PORT_TCP = 9091;`: This line declares a private, static, and final integer variable named `SERVER_PORT_TCP` and assigns it the value `9091`. This variable likely represents the TCP port number on which the server listens for incoming TCP connections.
- `private static final int SERVER_PORT_UDP = 9999;`: This line declares a private, static, and final integer variable named `SERVER_PORT_UDP` and assigns it the value `9999`. This variable likely represents the UDP port number on which the server listens for incoming UDP packets.
- `private static final String SERVER_DIRECTORY = "remote_folder";`: This line declares a private, static, and final String variable named `SERVER_DIRECTORY` and assigns it the value `"remote_folder"`. This variable likely represents the directory where the server stores its files or where it expects to receive files from clients.
- `private static final int BUFFER_SIZE = 1024;`: This line declares a private, static, and final integer variable named `BUFFER_SIZE` and assigns it the value `1024`. This variable likely represents the size of the buffer used for reading data from streams or sockets.
- `private static final int BLOCK_SIZE = 4 * 1024 * 1024;`: This line declares a private, static, and

final integer variable named `BLOCK_SIZE` and assigns it the value `4 * 1024 * 1024`, which is equal to `4194304`. This variable likely represents the size of data blocks used for file transfer or processing.

- `private static final String MESSAGE_SEPARATOR = "-x-xx-x-";`: This line declares a private, static, and final String variable named `MESSAGE_SEPARATOR` and assigns it the value `"-x-xx-x-"`. This variable likely represents a separator used to delimit messages or parts of messages in communication protocols.

The main pint of entrance:

```
  public static void main(String[] args) throws IOException {
    Files.createDirectories(Paths.get(SERVER_DIRECTORY));

    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.submit(() -> startUdpServer(SERVER_IP, SERVER_PORT_UDP, BUFFER_SIZE,
SERVER_DIRECTORY, true));
    executor.submit(() -> startTcpServer(SERVER_IP, SERVER_PORT_TCP, BUFFER_SIZE,
VERBOSE));
  }
```

This code segment represents the main method of the FileServer class. Let's break it down step by step:

public static void main(String[] args) throws IOException {: This line defines the main method, which is the entry point of the program. It takes an array of strings as input arguments (although it's not currently being used in this code). The throws IOException clause indicates that the method can throw an IOException, and it's not handling it locally.

Files.createDirectories(Paths.get(SERVER_DIRECTORY));: This line creates the directory specified by the SERVER_DIRECTORY constant using the Files.createDirectories method. This method ensures that the directory is created if it doesn't exist already. It's a precautionary step to ensure that the server's designated directory exists before starting the server.

ExecutorService executor = Executors.newFixedThreadPool(2);: This line creates a thread pool with a fixed number of threads equal to 2 using the Executors.newFixedThreadPool method. This thread pool will be used to execute tasks concurrently.

executor.submit(() -> startUdpServer(SERVER_IP, SERVER_PORT_UDP, BUFFER_SIZE, SERVER_DIRECTORY, true));: This line submits a task to the executor thread pool. The task is a lambda expression that calls the startUdpServer method with parameters SERVER_IP, SERVER_PORT_UDP, BUFFER_SIZE, SERVER_DIRECTORY, and true. This initiates the UDP server.

executor.submit(() -> startTcpServer(SERVER_IP, SERVER_PORT_TCP, BUFFER_SIZE, VERBOSE));: This line submits another task to the executor thread pool. Similar to the previous line, this task is a lambda expression that calls the startTcpServer method with parameters SERVER_IP, SERVER_PORT_TCP, BUFFER_SIZE, and VERBOSE. This initiates the TCP server.

}: This curly brace marks the end of the main method.

The "ServerStart" method:

```
  private static void startUdpServer(String serverIp, int serverPort, int bufferSize, String outputDir,
```

```
    boolean verbose) {
      try (DatagramSocket udpSocket = new DatagramSocket(serverPort)) {
         if (verbose) {
            System.out.println(String.format("Server listening on %s:%d", serverIp, serverPort));
         }

         while (true) {
            byte[] buffer = new byte[bufferSize];
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            udpSocket.receive(packet);

            String data = new String(packet.getData(), StandardCharsets.UTF_8).trim();

            String[] parts = data.split(MESSAGE_SEPARATOR);
            List<String> nonEmptyParts = new ArrayList<>();

            // Iterate through the parts array
            for (int i = 0; i < parts.length; i++) {
               // Check if the length of the part is greater than 0
               if (parts[i].length() > 0) {
                  // If it is, add it to the list of non-empty parts
                  nonEmptyParts.add(parts[i]);
               }
            }

            // Convert the list of non-empty parts back to an array
            parts = nonEmptyParts.toArray(new String[0]);

            if (parts.length >= 6) {
               String file = parts[0];
               String partName = parts[1];
               String fileHash = parts[2];
               int partNum = Integer.parseInt(parts[3]);
               int totalParts = Integer.parseInt(parts[4]);
               int size = Integer.parseInt(parts[5]);

               List<byte[]> chunks = new ArrayList<>();
               for (int i = 0; i < size; i++) {
                  packet = new DatagramPacket(buffer, buffer.length);
                  udpSocket.receive(packet);
                  chunks.add(Arrays.copyOfRange(packet.getData(), 0, packet.getLength()));
               }

               if (verbose) {
                  System.out.println(String.format("File '%s %d/%d' received and saved as '%s'", file,
partNum + 1, totalParts, partName));
               }

               File fileToWrite = new File(outputDir, partName);
               try (FileOutputStream fos = new FileOutputStream(fileToWrite)) {
                  for (byte[] chunk : chunks) {
                     fos.write(chunk);
                  }
               }
```

```
            if (partNum + 1 == totalParts) {
                File hashFile = new File(outputDir, "." + file);
                try (PrintWriter pw = new PrintWriter(hashFile)) {
                    pw.println(fileHash);
                }
            }
        }
    }
  } catch (SocketException e) {
    System.err.println("Error starting UDP server: " + e.getMessage());
    e.printStackTrace(); // Print stack trace for debugging
  } catch (IOException e) {
    System.err.println("Error receiving UDP packet: " + e.getMessage());
    e.printStackTrace(); // Print stack trace for debugging
  } catch (NumberFormatException e) {
    System.err.println("Error parsing integer: " + e.getMessage());
    e.printStackTrace(); // Print stack trace for debugging
  }
}
```

This Java code defines a method `startUdpServer` that creates and runs a UDP server. Let's break down each part:

- `private static void startUdpServer(String serverIp, int serverPort, int bufferSize, String outputDir, boolean verbose) {`: This line defines the method `startUdpServer`, which takes several parameters: `serverIp` (the IP address of the server), `serverPort` (the port on which the server listens), `bufferSize` (the size of the buffer used for receiving data), `outputDir` (the directory where received files will be saved), and `verbose` (a boolean flag indicating whether to print verbose output).
- `try (DatagramSocket udpSocket = new DatagramSocket(serverPort)) {`: This line creates a new `DatagramSocket` bound to the specified `serverPort` within a try-with-resources block. The try-with-resources statement automatically closes the `udpSocket` after the try block finishes.
- `if (verbose) { System.out.println(String.format("Server listening on %s:%d", serverIp, serverPort)); }`: If the `verbose` flag is set to true, this line prints a message indicating that the server is listening on the specified IP address and port.
- `while (true) {`: This line starts an infinite loop, which continuously listens for incoming UDP packets.
- `byte[] buffer = new byte[bufferSize];`: This line creates a byte array `buffer` with the specified `bufferSize`, which will be used to store incoming data.
- `DatagramPacket packet = new DatagramPacket(buffer, buffer.length);`: This line creates a new `DatagramPacket` with the `buffer` as its data array and the specified `buffer.length`. This packet will be used to receive incoming UDP packets.
- `udpSocket.receive(packet);`: This line blocks until a UDP packet is received, and then stores the received data in the `packet`.
- `String data = new String(packet.getData(), StandardCharsets.UTF_8).trim();`: This line converts the received data from the `packet` into a String using UTF-8 encoding and trims any leading or trailing whitespace.

- `String[] parts = data.split(MESSAGE_SEPARATOR);`: This line splits the received data into an array of strings using the `MESSAGE_SEPARATOR` as the delimiter.
- `List<String> nonEmptyParts = new ArrayList<>();`: This line creates a new ArrayList to store non-empty parts of the split data.
- `for (int i = 0; i < parts.length; i++) { ... }`: This for loop iterates through the `parts` array and filters out any empty strings, adding non-empty parts to the `nonEmptyParts` list.
- `parts = nonEmptyParts.toArray(new String[0]);`: This line converts the `nonEmptyParts` list back to an array and assigns it to the `parts` variable.
- The subsequent code processes the received data, including extracting file information, receiving file chunks, saving files, and saving file hashes. If verbose mode is enabled, it prints messages indicating the progress of file reception.
- The method is enclosed in a try-catch block to handle potential exceptions. If any exceptions occur (e.g., `SocketException`, `IOException`, or `NumberFormatException`), an error message is printed, and the stack trace is printed for debugging purposes.

The calculateFileHash method:

```
private static String calculateFileHash(String filePath) throws NoSuchAlgorithmException {
    try (InputStream input = new FileInputStream(filePath)) {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] buffer = new byte[65536];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            md.update(buffer, 0, bytesRead);
        }
        byte[] digest = md.digest();
        return bytesToHex(digest);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

This Java method `calculateFileHash` is responsible for computing the SHA-256 hash of a file given its file path. Let's break down each part of the method:

- `private static String calculateFileHash(String filePath) throws NoSuchAlgorithmException {`: This line defines the method `calculateFileHash`, which takes a `filePath` as input and returns a `String` representing the computed hash. It specifies that the method may throw a `NoSuchAlgorithmException` if the specified cryptographic algorithm (SHA-256 in this case) is not available.
- `try (InputStream input = new FileInputStream(filePath)) {`: This line opens an `InputStream` for reading the file specified by the `filePath` using a try-with-resources block. The try-with-resources block ensures that the `input` stream is closed automatically after use.
- `MessageDigest md = MessageDigest.getInstance("SHA-256");`: This line creates a `MessageDigest` instance for the SHA-256 algorithm. `MessageDigest.getInstance()` throws a `NoSuchAlgorithmException` if the specified algorithm is not available, hence the method signature declares that this exception may be thrown.

- `byte[] buffer = new byte[65536];`: This line creates a buffer of size 65536 bytes (64 KB) to read data from the file.
- `int bytesRead;`: This variable will hold the number of bytes read by each `input.read()` call.
- `while ((bytesRead = input.read(buffer)) != -1) { ... }`: This line reads data from the file into the `buffer` in a loop until the end of the file is reached (`input.read()` returns -1). The number of bytes read is stored in the `bytesRead` variable.
- `md.update(buffer, 0, bytesRead);`: This line updates the `MessageDigest` with the bytes read from the file. It specifies that only `bytesRead` number of bytes from the buffer should be used.
- `byte[] digest = md.digest();`: This line computes the final hash value by calling `md.digest()`, which returns a byte array representing the hash value.
- `return bytesToHex(digest);`: This line returns the hash value converted to a hexadecimal string using the `bytesToHex` method.
- `} catch (IOException e) { throw new RuntimeException(e); }`: If an `IOException` occurs while reading the file, it's caught here. Instead of handling the exception, it's wrapped in a `RuntimeException` and rethrown. This simplifies error handling by propagating the exception to the caller.

The bytesToHex method:

```
private static String bytesToHex(byte[] bytes) {
    try {
        StringBuilder sb = new StringBuilder();
        for (byte b : bytes) {
            sb.append(String.format("%02x", b));
        }
        return sb.toString();
    } catch (Exception e) {
        System.err.println("Error converting bytes to hexadecimal: " + e.getMessage());
        return ""; // Return empty string if an error occurs
    }
}
```

This Java method `bytesToHex` converts a byte array into a hexadecimal string representation.

- `private static String bytesToHex(byte[] bytes) {`: This line defines the method `bytesToHex`, which takes a byte array `bytes` as input and returns a hexadecimal string representation.
- `try { ... }`: This line starts a try block to handle potential exceptions that may occur during the conversion process.
- `StringBuilder sb = new StringBuilder();`: This line creates a `StringBuilder` instance named `sb` to efficiently build the hexadecimal string.
- `for (byte b : bytes) { ... }`: This line initiates a loop over each byte in the `bytes` array using an enhanced for loop.
- `sb.append(String.format("%02x", b));`: Inside the loop, each byte `b` is converted to a two-digit hexadecimal string using `String.format()` with the format specifier `%02x`. This ensures that each byte is represented by exactly two characters (e.g., `0A`, `FF`, etc.), padding with leading zeros if necessary. The resulting hexadecimal string is appended to the `StringBuilder` `sb`.

- `return sb.toString();`: After the loop completes, the `StringBuilder` `sb` is converted to a string using the `toString()` method, and the resulting hexadecimal string is returned.
- `} catch (Exception e) { ... }`: If an exception occurs during the conversion process, it's caught here. Instead of specifying a particular type of exception, it simply catches any `Exception`. This is not recommended for production code, as it can catch unexpected exceptions that may not be related to the conversion process.
- `System.err.println("Error converting bytes to hexadecimal: " + e.getMessage());`: Inside the catch block, an error message is printed to the standard error stream (`System.err`) indicating that an error occurred during the conversion process. It also includes the specific error message obtained from the exception (`e.getMessage()`).
- `return "";`: If an exception occurs, the method returns an empty string. This is not an ideal error handling strategy, as it silently fails and doesn't provide any indication to the caller that an error occurred. It would be better to log the error and consider throwing an exception to indicate the failure to convert the bytes to hexadecimal.

The startTcpServer method:

```
   private static void startTcpServer(String serverIp, int serverPort, int bufferSize, boolean verbose) {
      try (ServerSocket tcpSocket = new ServerSocket(serverPort)) {
         if (verbose) {
            System.out.println(String.format("TCP Server started, waiting for connections on %s:%d",
serverIp, serverPort));
         }

         while (true) {
            Socket clientSocket = tcpSocket.accept();
            try {
               handleTcpCommunication(clientSocket, bufferSize, verbose);
            } catch (Exception e) {
               System.err.println("Error handling TCP communication: " + e.getMessage());
               e.printStackTrace(); // Print stack trace for debugging
            }
         }
      } catch (IOException e) {
         System.err.println("Error starting TCP server: " + e.getMessage());
      } catch (Exception e) {
         System.err.println("Unexpected error in TCP server: " + e.getMessage());
      }
   }
```

This Java method `startTcpServer` is responsible for initializing and running a TCP server.

- `private static void startTcpServer(String serverIp, int serverPort, int bufferSize, boolean verbose) {`: This line defines the method `startTcpServer`, which takes several parameters: `serverIp` (the IP address of the server), `serverPort` (the port on which the server listens), `bufferSize` (the size of the buffer used for communication), and `verbose` (a boolean flag indicating whether to print verbose output).

- `try (ServerSocket tcpSocket = new ServerSocket(serverPort)) {`: This line creates a new `ServerSocket` bound to the specified `serverPort` within a try-with-resources block. The try-with-resources block ensures that the `tcpSocket` is closed automatically after use.
- `if (verbose) { System.out.println(String.format("TCP Server started, waiting for connections on %s:%d", serverIp, serverPort)); }`: If the `verbose` flag is set to true, this line prints a message indicating that the TCP server has started and is waiting for connections on the specified IP address and port.
- `while (true) {`: This line starts an infinite loop, which continuously waits for incoming TCP client connections.
- `Socket clientSocket = tcpSocket.accept();`: This line blocks until a client connection is established, and then returns a new `Socket` object representing the connection.
- `try { ... }`: Inside the loop, a try block is used to handle potential exceptions that may occur during communication with the client.
- `handleTcpCommunication(clientSocket, bufferSize, verbose);`: This line calls the `handleTcpCommunication` method to handle communication with the connected client. It passes the `clientSocket`, `bufferSize`, and `verbose` parameters to the method.
- `} catch (Exception e) { ... }`: If an exception occurs during communication with the client, it's caught here. Instead of specifying a particular type of exception, it simply catches any `Exception`. This is not recommended for production code, as it can catch unexpected exceptions that may not be related to communication with the client.
- `System.err.println("Error handling TCP communication: " + e.getMessage());`: Inside the catch block, an error message is printed to the standard error stream (`System.err`) indicating that an error occurred during communication with the client. It also includes the specific error message obtained from the exception (`e.getMessage()`).
- `e.printStackTrace();`: Additionally, the stack trace of the exception is printed to aid in debugging.
- `} catch (IOException e) { ... }`: If an `IOException` occurs while setting up the `ServerSocket`, it's caught here. An error message indicating the issue is printed to the standard error stream.
- `} catch (Exception e) { ... }`: If any other unexpected exception occurs, it's caught here. An error message indicating the unexpected error is printed to the standard error stream.

The handleTcpCommunication method:

```
    private static void handleTcpCommunication(Socket clientSocket, int bufferSize, boolean verbose) {
        try (BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
             PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), true)) {

            String message;
            while ((message = reader.readLine()) != null) {
                if (verbose) {
                    System.out.println(String.format("Received command from %s:%d",
clientSocket.getInetAddress().getHostAddress(), clientSocket.getPort()));
                    System.out.println(message);
                }

                List<String[]> messageCollector = new ArrayList<>();
```

```
  for (String line : message.split("\n")) {
                String[] parts = line.split(MESSAGE_SEPARATOR);
                messageCollector.add(parts);
            }

            List<String[]> serverFiles = new ArrayList<>();
            List<String[]> newOrChangedFiles = new ArrayList<>();
            List<String[]> tndFiles = new ArrayList<>();

            for (String[] fileInfo : messageCollector) {
                if (fileInfo.length < 3) {
                    System.err.println("Invalid message format: " + Arrays.toString(fileInfo));
                    continue; // Skip this message and proceed with the next one
                }

                String file = fileInfo[1];
                String fileHash = fileInfo[2];
                File fileToCheck = new File(SERVER_DIRECTORY, file);
                try {
                    if (fileToCheck.exists()) {
                        String existingHash = calculateFileHash(fileToCheck.getAbsolutePath());
                        if (existingHash.equals(fileHash)) {
                            continue;
                        }
                    }
                } catch (NoSuchAlgorithmException e) {
                    System.err.println("Error calculating file hash: " + e.getMessage());
                    continue; // Skip this file and proceed with the next one
                }
                serverFiles.add(fileInfo);
                newOrChangedFiles.add(fileInfo);
            }

            for (File file : new File(SERVER_DIRECTORY).listFiles()) {
                if (!file.getName().startsWith(".")) {
                    boolean found = false;
                    for (String[] fileInfo : messageCollector) {
                        if (fileInfo.length < 2) {
                            System.err.println("Invalid message format: " + Arrays.toString(fileInfo));
                            continue; // Skip this message and proceed with the next one
                        }
                        if (file.getName().equals(fileInfo[1])) {
                            found = true;
                            break;
                        }
                    }
                    if (!found) {
                        try {
                            tndFiles.add(new String[]{null, file.getName(),
calculateFileHash(file.getAbsolutePath()), null});
                        } catch (NoSuchAlgorithmException e) {
                            System.err.println("Error calculating file hash: " + e.getMessage());
                            continue; // Skip this file and proceed with the next one
                         }}}}
```

```
            if (verbose) {
                System.out.println(newOrChangedFiles);
            }

            StringBuilder response = new StringBuilder();
            for (String[] fileInfo : newOrChangedFiles) {
                response.append(String.format("%s-x-xx-x-%s-x-xx-x-%s-x-xx-x-%s\n", "None",
fileInfo[1], fileInfo[2], "None"));
            }
            if (newOrChangedFiles.isEmpty()) {
                response.append("UPTODATE");
            }

            if (verbose) {
                System.out.println(response.toString());
            }

            writer.println(response.toString());

            System.out.println("MESSAGE SENT");

        }
    } catch (IOException e) {
        System.err.println("Error handling TCP communication: " + e.getMessage());
    }

    System.out.println("handleTcpCommunication: ENDING");
}
```

This Java method `handleTcpCommunication` is responsible for managing communication with a TCP client.

- `private static void handleTcpCommunication(Socket clientSocket, int bufferSize, boolean verbose) {`: This line defines the method `handleTcpCommunication`, which takes several parameters: `clientSocket` (the socket representing the client connection), `bufferSize` (the size of the buffer used for communication), and `verbose` (a boolean flag indicating whether to print verbose output).
- `try (BufferedReader reader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));`: This line creates a `BufferedReader` to read data from the client's input stream (`clientSocket.getInputStream()`). It uses a try-with-resources block to ensure that the reader is closed automatically after use.
- `PrintWriter writer = new PrintWriter(clientSocket.getOutputStream(), true)) {`: This line creates a `PrintWriter` to write data to the client's output stream (`clientSocket.getOutputStream()`). It also uses a try-with-resources block to ensure that the writer is closed automatically after use.
- `String message; while ((message = reader.readLine()) != null) { ... }`: This line initiates a loop that reads lines of text from the client until the client closes the connection or an error occurs. Each line of text is stored in the `message` variable.

- Inside the loop:
  If `verbose` is true, it prints the received command and the message from the client.
  It splits the received message into parts based on the newline character (`\n`) and stores each part in `messageCollector`.
  It iterates through the collected messages (`messageCollector`) to process file information sent by the client. It checks if the files exist on the server, compares their hashes, and categorizes them into `serverFiles`, `newOrChangedFiles`, and `tndFiles`.
  It constructs a response message based on the files that are new or have been changed on the server.
- `} catch (IOException e) { ... }`: If an `IOException` occurs during communication with the client, it's caught here. An error message indicating the issue is printed to the standard error stream.
- `System.out.println("handleTcpCommunication: ENDING");`: After the loop ends, this line prints a message indicating that the `handleTcpCommunication` method is ending.

The FileSync code explanation:

This Java class `FileSync` is responsible for synchronizing files between a client and a server using TCP and UDP protocols. Its components:

- Imports:
  - `java.io.*`: Provides classes for input and output operations.
  - `java.net.*`: Provides classes for networking operations.
  - `java.nio.file.Files` and `java.nio.file.Paths`: Provides classes for file operations.
  - `java.security.MessageDigest` and `java.security.NoSuchAlgorithmException`: Used for calculating file hashes.
  - `java.util.*`: Provides utility classes like `ArrayList`, `List`, and `Arrays`.
  - `java.util.regex.Pattern`: Used for splitting strings based on a regular expression pattern.

- Constants:
  - `VERBOSE`, `SERVER_IP`, `SERVER_PORT_TCP`, `SERVER_PORT_UDP`, `CLIENT_DIRECTORY`: Configuration constants for server IP address, TCP and UDP ports, client directory, and verbosity flag.
  - `BUFFER_SIZE`, `BLOCK_SIZE`, `CHUNK_SIZE`: Constants for buffer sizes and block sizes used for data transfer.
  - `MESSAGE_SEPARATOR`, `UDP_MESSAGE_SEPARATOR`: Delimiters used for separating parts of messages in TCP and UDP communication.

- `main` Method:
  - Creates the client directory if it doesn't exist.
  - Calls the `sync` method to synchronize files with the server.

- `sync` Method**:
  - Retrieves the list of files in the client directory.
  - Creates an initial synchronization message for each file and sends it to the server over TCP.
  - Receives a response from the server, indicating whether files are up to date or need synchronization.
  - Parses the server response and processes each file accordingly:
    - Calculates the file hash and divides the file into chunks.
    - Constructs UDP messages containing file information and chunks.
    - Sends UDP messages to the server for each file chunk.
  - Handles exceptions related to I/O operations and hashing.

- Helper Methods:
- `getFileBlocks`: Reads a file and divides it into blocks.
- `calculateFileHash`: Calculates the SHA-256 hash of a file.
- `bytesToHex`: Converts byte arrays to hexadecimal strings.
- `parseMessages`: Parses a string into a list of message parts.
- `createMessage`: Creates a synchronization message for a file.
- `createUdpMessage`: Creates a UDP message containing file information and chunks.
- `udpSend`: Sends UDP messages to the server.

This class implements a file synchronization mechanism between a client and a server using TCP for control messages and UDP for transferring file chunks, with error handling and verbosity control.

The import block:

```
import java.io.*;
import java.net.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.security.MessageDigest;
import java.util.*;
import java.security.NoSuchAlgorithmException;
import java.util.regex.Pattern;
```

- `import java.io.*;`: This imports all classes from the `java.io` package, which provides classes for input and output operations.
- `import java.net.*;`: This imports all classes from the `java.net` package, which provides classes for networking operations, like working with sockets and URLs.
- `import java.nio.file.Files;` and `import java.nio.file.Paths;`: These import specific classes from the `java.nio.file` package, which provides an alternative, more modern way of working with files and directories.
- `import java.security.MessageDigest;`: This imports the `MessageDigest` class from the `java.security` package, which is used for cryptographic hashing operations.
- `import java.util.*;`: This imports all classes from the `java.util` package, which provides utility classes like collections, date and time facilities, etc.
- `import java.security.NoSuchAlgorithmException;`: This imports the `NoSuchAlgorithmException` class from the `java.security` package, which is thrown when a particular cryptographic algorithm is requested but is not available in the environment.
- `import java.util.regex.Pattern;`: This imports the `Pattern` class from the `java.util.regex` package, which is used for defining patterns for matching strings.

These import statements bring in the necessary classes for various operations that the program might perform, such as networking, file handling, cryptographic hashing, and regular expressions.

The entry point:

```
public static void main(String[] args) throws IOException {
    File directory = new File(CLIENT_DIRECTORY);
    if (!directory.exists()) {
        directory.mkdir();
    }
    try {
```

```
            sync(SERVER_IP, SERVER_PORT_TCP, SERVER_PORT_UDP, CLIENT_DIRECTORY,
VERBOSE);
        } catch (IOException e) {
            e.printStackTrace(); // or handle the exception as per your application's requirements
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace(); // or handle the exception as per your application's requirements
        }
    }
```

This code segment is the `main` method of a Java program:

- `public static void main(String[] args) throws IOException {`: This line declares the main method, which serves as the entry point of the Java program. It takes an array of strings (`args`) as input parameters, but it's not currently using them. It throws an `IOException` because the `sync` method it calls inside may throw this exception, and the method does not handle it locally.
- `File directory = new File(CLIENT_DIRECTORY);`: This line creates a new `File` object named `directory` representing a directory in the file system. The directory's path is specified by the `CLIENT_DIRECTORY` constant.
- `if (!directory.exists()) {`: This line checks if the directory represented by the `directory` object exists in the file system.
- `directory.mkdir();`: If the directory doesn't exist, this line creates it. It's using the `mkdir()` method of the `File` class to create the directory.
- `try {`: This begins a try block, indicating that the code inside will be executed, and exceptions will be caught and handled if they occur.
- `sync(SERVER_IP, SERVER_PORT_TCP, SERVER_PORT_UDP, CLIENT_DIRECTORY, VERBOSE);`: This line calls the `sync` method, passing in several parameters:
  - `SERVER_IP`: The IP address of the server.
  - `SERVER_PORT_TCP`: The TCP port number of the server.
  - `SERVER_PORT_UDP`: The UDP port number of the server.
  - `CLIENT_DIRECTORY`: The directory path on the client where files will be synchronized.
  - `VERBOSE`: A boolean flag indicating whether to print verbose output.
- `} catch (IOException e) {`: This catches an `IOException` if it's thrown by the `sync` method or any method it calls. Inside the catch block, the exception is printed to the standard error stream using `e.printStackTrace()`.
- `} catch (NoSuchAlgorithmException e) {`: This catches a `NoSuchAlgorithmException` if it's thrown by the `sync` method or any method it calls. Inside the catch block, the exception is printed to the standard error stream using `e.printStackTrace()`.

The sync method:

```
    private static void sync(String serverIp, int serverPortTcp, int serverPortUdp, String inputDir, boolean
    verbose) throws IOException, NoSuchAlgorithmException {
        File[] files = new File(inputDir).listFiles();

        List<String> messageCollector = new ArrayList<>();
        for (File file : files) {
            String filePath = file.getAbsolutePath();
            String message = createMessage(file.getName(), "INIT", calculateFileHash(filePath), 0);
```

```
    messageCollector.add(message);
}

String msg = String.join("\n", messageCollector);

try (Socket tcpSocket = new Socket(serverIp, serverPortTcp)) {
    OutputStream output = tcpSocket.getOutputStream();
    output.write(msg.getBytes());
    InputStream input = tcpSocket.getInputStream();
    byte[] responseBytes = new byte[BUFFER_SIZE];

    // Read at least one byte
    int bytesRead = input.read(responseBytes);

    String response = "";
    if (bytesRead > 0) {
        // Process the received bytes
        response = new String(responseBytes, 0, bytesRead);
        // Your processing logic here
        if (verbose) {
            System.out.println("Received response from server: " + response);
        }
    } else {
        // Handle case where no bytes are received
        System.out.println("No response received from server.");
    }


    // byte[] responseBytes = input.readNBytes(BUFFER_SIZE);

    if (response.equals("UPTODATE")) {
        return;
    }

    List<String[]> messageCollectorList = parseMessages(response);
    if (verbose) {
        System.out.println("Message collector list:");
        for (String[] msgData : messageCollectorList) {
            System.out.println(Arrays.toString(msgData));
        }
    }

    for (String[] msgData : messageCollectorList) {
        if (msgData.length < 4) { // 6
            System.err.println("Invalid message format: " + Arrays.toString(msgData));
            continue; // Skip this message and proceed with the next one
        }

        String file = msgData[1];
        String partName = msgData[1];
        String fileHash = msgData[2];
        int partNum = 0;
        int totalParts = 1;
        int size = 1;
```

```
            List<byte[]> fileBlocks = getFileBlocks(inputDir, file);
            String calculatedFileHash = calculateFileHash(file);
            int totalChunks = fileBlocks.size() / CHUNK_SIZE + (fileBlocks.size() % CHUNK_SIZE > 0 ?
1 : 0);

            if (verbose) {
                System.out.println("Uploading file: " + file);
            }

            for (int idx = 0; idx < fileBlocks.size(); idx += CHUNK_SIZE) {
                List<byte[]> chunks = fileBlocks.subList(idx, Math.min(idx + CHUNK_SIZE,
fileBlocks.size()));
                String udpMessage = createUdpMessage(file, partName, calculatedFileHash, partNum,
totalChunks, chunks);

                udpSend(serverIp, serverPortUdp, udpMessage, chunks, verbose);
                partNum++;
            }
        }
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Array index out of bounds error: " + e.getMessage());
        e.printStackTrace();
    }}
```

This `sync` method is responsible for synchronizing files between the client and the server:

- `File[] files = new File(inputDir).listFiles();`: This line retrieves an array of `File` objects representing the files in the `inputDir` directory on the client side.
- `List<String> messageCollector = new ArrayList<>();`: This initializes an empty `ArrayList` to collect synchronization messages for each file.
- `for (File file : files) { ... }`: This loop iterates over each file in the `files` array.
- `String message = createMessage(file.getName(), "INIT", calculateFileHash(filePath), 0);`: For each file, this line creates a synchronization message using the `createMessage` method. The message contains the file name, a status ("INIT" indicating initialization), the file hash calculated using `calculateFileHash`, and a part number (0 for initialization).
- `messageCollector.add(message);`: This adds the synchronization message to the `messageCollector` list.
- `String msg = String.join("\n", messageCollector);`: This line joins all the synchronization messages into a single string, separated by newline characters.
- `try (Socket tcpSocket = new Socket(serverIp, serverPortTcp)) { ... }`: This begins a try-with-resources block to create a TCP socket connection to the server specified by `serverIp` and `serverPortTcp`. The `try` block ensures that the socket is properly closed after use.
- `OutputStream output = tcpSocket.getOutputStream();`: This obtains an output stream from the socket to send data to the server.
- `output.write(msg.getBytes());`: This line sends the synchronization messages to the server by writing the byte representation of the `msg` string to the output stream.
- `InputStream input = tcpSocket.getInputStream();`: This obtains an input stream from the socket to receive data from the server.

- `int bytesRead = input.read(responseBytes);`: This reads bytes from the input stream into the `responseBytes` buffer and returns the number of bytes read.
- `if (bytesRead > 0) { ... }`: This checks if any bytes were read from the input stream.
- `response = new String(responseBytes, 0, bytesRead);`: If bytes were read, this line converts the received bytes to a string representing the server's response.
- `if (response.equals("UPTODATE")) { return; }`: If the server's response indicates that the client's files are up to date ("UPTODATE"), the method returns early, indicating that no further synchronization is needed.
- `List<String[]> messageCollectorList = parseMessages(response);`: This line parses the server's response into a list of message arrays using the `parseMessages` method.
- The subsequent code processes each message received from the server, extracts file information, calculates file hashes, and uploads file chunks using UDP communication.
- `udpSend(serverIp, serverPortUdp, udpMessage, chunks, verbose);`: This line sends UDP messages containing file chunks to the server for upload.
- `catch (ArrayIndexOutOfBoundsException e) { ... }`: This catch block handles the `ArrayIndexOutOfBoundsException` that might occur during message processing. It prints an error message and stack trace for debugging purposes.

The getFileBlock method:

```
private static List<byte[]> getFileBlocks(String folder, String file) throws IOException {
    List<byte[]> blocks = new ArrayList<>();
    try (InputStream input = new FileInputStream(new File(folder, file))) {
        byte[] buffer = new byte[BLOCK_SIZE];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            blocks.add(Arrays.copyOfRange(buffer, 0, bytesRead));
        }
    }
    return blocks;
}
```

This `getFileBlocks` method is responsible for reading a file from the specified folder and breaking it into blocks of bytes:

- `private static List<byte[]> getFileBlocks(String folder, String file) throws IOException {`: This line declares the method signature. It specifies that the method is private, static, and returns a list of byte arrays (`List<byte[]>`). The method takes two parameters: `folder`, representing the folder where the file is located, and `file`, representing the name of the file to read.
- `List<byte[]> blocks = new ArrayList<>();`: This line initializes an empty `ArrayList` called `blocks` to store the blocks of bytes read from the file.
- `try (InputStream input = new FileInputStream(new File(folder, file))) { ... }`: This begins a try-with-resources block to open an input stream (`InputStream`) for reading the file specified by the `folder` and `file` parameters. The try-with-resources statement ensures that the input stream is properly closed after use.

- `byte[] buffer = new byte[BLOCK_SIZE];`: This line declares a byte array called `buffer` with a size of `BLOCK_SIZE`. The `BLOCK_SIZE` constant determines the size of each block of bytes to read from the file.
- `int bytesRead;`: This declares an integer variable `bytesRead` to store the number of bytes read from the input stream in each iteration of the loop.
- `while ((bytesRead = input.read(buffer)) != -1) { ... }`: This loop reads bytes from the input stream into the `buffer` array until the end of the file is reached (`input.read(buffer)` returns `-1`). In each iteration, it also assigns the number of bytes read to the `bytesRead` variable.
- `blocks.add(Arrays.copyOfRange(buffer, 0, bytesRead));`: Inside the loop, this line adds a copy of the portion of the `buffer` array that contains the bytes read (`0` to `bytesRead`) to the `blocks` list. This ensures that each block contains only the actual bytes read, avoiding any extra unused space at the end of the array.
- `return blocks;`: Finally, this line returns the list of byte arrays containing the file blocks.

The calculateFileHash method:

```
private static String calculateFileHash(String filePath) throws NoSuchAlgorithmException {
    try (InputStream input = new FileInputStream(filePath)) {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] buffer = new byte[65536];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            md.update(buffer, 0, bytesRead);
        }
        byte[] digest = md.digest();
        return bytesToHex(digest);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

This `calculateFileHash` method is responsible for calculating the SHA-256 hash of a file given its file path.

- `private static String calculateFileHash(String filePath) throws NoSuchAlgorithmException {`: This line declares the method signature. It specifies that the method is private, static, and returns a `String` representing the calculated hash. The method takes a `String` parameter `filePath`, which represents the path to the file for which the hash is to be calculated. It also declares that the method may throw a `NoSuchAlgorithmException` if the specified cryptographic algorithm (SHA-256) is not available.
- `try (InputStream input = new FileInputStream(filePath)) { ... }`: This begins a try-with-resources block to open an input stream (`InputStream`) for reading the file specified by the `filePath` parameter. The try-with-resources statement ensures that the input stream is properly closed after use.
- `MessageDigest md = MessageDigest.getInstance("SHA-256");`: This line creates a `MessageDigest` object named `md` using the SHA-256 hashing algorithm. The `getInstance` method of the `MessageDigest` class is used to obtain a `MessageDigest` instance for the specified algorithm.

- `byte[] buffer = new byte[65536];`: This line declares a byte array called `buffer` with a size of 65,536 bytes (or 64 KB). This buffer is used to read data from the input stream in chunks.
- `int bytesRead;`: This declares an integer variable `bytesRead` to store the number of bytes read from the input stream in each iteration of the loop.
- `while ((bytesRead = input.read(buffer)) != -1) { ... }`: This loop reads bytes from the input stream into the `buffer` array until the end of the file is reached (`input.read(buffer)` returns `-1`). In each iteration, it also assigns the number of bytes read to the `bytesRead` variable.
- `md.update(buffer, 0, bytesRead);`: Inside the loop, this line updates the `MessageDigest` object `md` with the bytes read from the `buffer`. It specifies the portion of the buffer to use for the update operation, starting from index `0` up to `bytesRead`.
- `byte[] digest = md.digest();`: After reading the entire file and updating the `MessageDigest`, this line computes the final hash value (digest) by calling the `digest` method on the `MessageDigest` object `md`.
- `return bytesToHex(digest);`: Finally, this line returns the hexadecimal representation of the calculated hash by invoking the `bytesToHex` method with the `digest` byte array as an argument.
- `} catch (IOException e) { throw new RuntimeException(e); }`: This catch block handles any `IOException` that may occur during the file reading process. It throws a `RuntimeException` (which is an unchecked exception) with the original `IOException` as its cause. This approach propagates any I/O errors as runtime exceptions, simplifying error handling in the calling code.

The parseMessages method:

```
private static List<String[]> parseMessages(String response) {
    List<String[]> messageList = new ArrayList<>();
    String[] lines = response.split("\n");
    for (String line : lines) {
        String[] parts = line.split(Pattern.quote(MESSAGE_SEPARATOR));
        messageList.add(parts);
    }
    return messageList;
}
```

This `parseMessages` method takes a `String` `response` as input, which presumably contains multiple messages separated by newline characters (`\n`). Each message itself is assumed to be composed of multiple parts separated by a predefined separator, `MESSAGE_SEPARATOR`. The method then splits the response into individual lines and parses each line into an array of parts using the `MESSAGE_SEPARATOR`.

- `private static List<String[]> parseMessages(String response) {`: This line defines the method `parseMessages`, indicating that it's private, static, and returns a list of string arrays. The method takes a single parameter, `response`, which is the string to be parsed.
- `List<String[]> messageList = new ArrayList<>();`: This line initializes an empty list named `messageList`. This list will contain arrays of strings, each representing the parts of a parsed message.
- `String[] lines = response.split("\n");`: Here, the `response` string is split into individual lines using the `split` method with `\n` as the delimiter. This assumes that each message is on a separate line.
- `for (String line : lines) {`: This initiates a loop that iterates over each line in the `lines` array.

- `String[] parts = line.split(Pattern.quote(MESSAGE_SEPARATOR));`: For each line, the `split` method is called to split the line into parts using the `MESSAGE_SEPARATOR` as the delimiter. `Pattern.quote()` is used to ensure that the separator is treated as a literal string rather than a regular expression pattern. The result is an array of strings representing the parts of the message.
- `messageList.add(parts);`: After splitting the line into parts, the array of parts is added to the `messageList`. Each element of `messageList` represents a single message parsed into its constituent parts.
- `return messageList;`: Finally, the method returns the `messageList`, which contains arrays of strings representing the parsed messages.

The createMessage method:

```
    private static String createMessage(String file, String status, String fileHash, int partNum) {
        return String.format("%s%s%s%s%s%s", status, MESSAGE_SEPARATOR, file,
MESSAGE_SEPARATOR, fileHash, MESSAGE_SEPARATOR, partNum);
    }
```

This `createMessage` method constructs a message string by combining several components:

1. `status`: A string representing the status of the message.
2. `file`: A string representing the name of the file.
3. `fileHash`: A string representing the hash of the file.
4. `partNum`: An integer representing the part number.

These components are combined into a single string using the `String.format` method, where `%s` is a placeholder for each component. Between each component, the `MESSAGE_SEPARATOR` is inserted to delineate the boundaries between them.

Here's the format string used in `String.format`:

- `%s`: Placeholder for the `status` component.
- `MESSAGE_SEPARATOR`: Insertion of the `MESSAGE_SEPARATOR`.
- `%s`: Placeholder for the `file` component.
- `MESSAGE_SEPARATOR`: Insertion of the `MESSAGE_SEPARATOR`.
- `%s`: Placeholder for the `fileHash` component.
- `MESSAGE_SEPARATOR`: Insertion of the `MESSAGE_SEPARATOR`.
- `%s`: Placeholder for the `partNum` component.

So, the final constructed message string will look like this:

```
status -x-xx-x- file -x-xx-x- fileHash -x-xx-x- partNum
```

where `-x-xx-x-` represents the `MESSAGE_SEPARATOR`.

The createUpdMessage method:

```
    private static String createUdpMessage(String file, String partName, String fileHash, int partNum, int
totalParts, List<byte[]> chunks) throws UnsupportedEncodingException {
```

```
        StringBuilder sb = new StringBuilder();
        sb.append(file).append(UDP_MESSAGE_SEPARATOR)
                .append(partName).append(UDP_MESSAGE_SEPARATOR)
                .append(fileHash).append(UDP_MESSAGE_SEPARATOR)
                .append(partNum).append(UDP_MESSAGE_SEPARATOR)
                .append(totalParts).append(UDP_MESSAGE_SEPARATOR)
                .append(chunks.size()).append(UDP_MESSAGE_SEPARATOR);

        for (byte[] chunk : chunks) {
            sb.append(new String(chunk, "UTF-8")).append(UDP_MESSAGE_SEPARATOR);
        }

        return sb.toString();
    }
```

This `createUdpMessage` method constructs a UDP message string with multiple components:

1. `file`: A string representing the name of the file.
2. `partName`: A string representing the part name.
3. `fileHash`: A string representing the hash of the file.
4. `partNum`: An integer representing the part number.
5. `totalParts`: An integer representing the total number of parts.
6. `chunks`: A list of byte arrays representing the data chunks of the file.

These components are concatenated into a single string using a `StringBuilder`. Each component is separated by the `UDP_MESSAGE_SEPARATOR` string.

Here's a breakdown of how the method constructs the message:

1. Append `file` followed by `UDP_MESSAGE_SEPARATOR`.
2. Append `partName` followed by `UDP_MESSAGE_SEPARATOR`.
3. Append `fileHash` followed by `UDP_MESSAGE_SEPARATOR`.
4. Append `partNum` followed by `UDP_MESSAGE_SEPARATOR`.
5. Append `totalParts` followed by `UDP_MESSAGE_SEPARATOR`.
6. Append the size of the `chunks` list followed by `UDP_MESSAGE_SEPARATOR`.
7. Iterate through each chunk in the `chunks` list:
   - Convert the byte array to a string using UTF-8 encoding.
   - Append the string representation of the chunk followed by `UDP_MESSAGE_SEPARATOR`.

The udpSend method:

```
    private static void udpSend(String serverIp, int serverPortUdp, String message, List<byte[]> chunks, boolean verbose) throws IOException {
        InetAddress serverAddress = InetAddress.getByName(serverIp);
        DatagramSocket udpSocket = new DatagramSocket();

        byte[] fileBytes = message.getBytes();
        DatagramPacket packet = new DatagramPacket(fileBytes, fileBytes.length, serverAddress, serverPortUdp);
        udpSocket.send(packet);

        if (verbose) {
```

```
            System.out.println("    ... sending UDP message");
        }

        for (int i = 0; i < chunks.size(); i++) {
            byte[] chunkBytes = chunks.get(i);
            packet = new DatagramPacket(chunkBytes, chunkBytes.length, serverAddress, serverPortUdp);
            udpSocket.send(packet);

            if (verbose) {
                System.out.println("    ... sending chunk " + (i + 1) + " of " + chunks.size());
            }
        }

        udpSocket.close();
    }
```

This `udpSend` method is responsible for sending UDP messages, including data chunks, to a specified server.

- Retrieve Server Address: The method starts by converting the `serverIp` string into an `InetAddress` object representing the server's IP address.

```java
InetAddress serverAddress = InetAddress.getByName(serverIp);
```

- Create DatagramSocket: A new `DatagramSocket` object is created, which is used to send UDP packets over the network.

```java
DatagramSocket udpSocket = new DatagramSocket();
```

- Send Message: The method constructs a UDP packet (`DatagramPacket`) containing the message (`message`) converted to bytes. This packet is then sent to the server at the specified IP address (`serverAddress`) and UDP port (`serverPortUdp`).

```java
byte[] fileBytes = message.getBytes();
DatagramPacket packet = new DatagramPacket(fileBytes, fileBytes.length, serverAddress, serverPortUdp);
udpSocket.send(packet);
```

- Verbose Output: If the `verbose` flag is set to `true`, the method prints a message indicating that a UDP message is being sent.

```java
if (verbose) {
    System.out.println("    ... sending UDP message");
}
```

- Send Chunks: The method iterates over each byte array (`chunkBytes`) in the `chunks` list. Each byte array represents a data chunk. For each chunk, a new `DatagramPacket` is created and sent to the server.

```java
for (int i = 0; i < chunks.size(); i++) {
    byte[] chunkBytes = chunks.get(i);
    packet = new DatagramPacket(chunkBytes, chunkBytes.length, serverAddress, serverPortUdp);
    udpSocket.send(packet);

    if (verbose) {
        System.out.println("    ... sending chunk " + (i + 1) + " of " + chunks.size());
    }
}
```

- Close Socket: Finally, the `DatagramSocket` is closed to release the network resources.

```java
udpSocket.close();
```

Conclusions, and Recommendation's:

To properly handle UDP communication issues in Java:

- Packet loss:
  - Implement a custom retransmission mechanism where the client retransmits packets if it doesn't receive a response within a certain timeout.
  - Use sequence numbers to detect missing packets and request retransmission.
  - Incorporate forward error correction (FEC) techniques to add redundant data that can help recover lost packets.
- Out-of-order delivery:
  - Use sequence numbers to properly reassemble the data on the receiving end.
  - Maintain a buffer on the receiver side to reorder the packets before processing.
- Lack of flow control:
  - Implement a custom flow control mechanism, such as a sliding window protocol, to prevent the sender from overwhelming the receiver.
  - Use timeouts and backoff strategies to throttle the sending rate if packets are being dropped.
- Unreliable data transfer:
  - Incorporate checksum or hash-based verification to detect corrupted data.
  - Retransmit packets if corruption is detected.
  - Consider using a more reliable protocol like TCP for critical data transfers.
- Synchronization issues:
  - Use proper synchronization primitives, such as locks or semaphores, to coordinate access to shared resources on the server.
  - Maintain connection state information on the server to handle concurrent client requests.
- Firewall/NAT traversal:
  - Implement techniques like UDP hole punching to establish a direct UDP connection between the client and server.

    - Use a relay server or a UDP-based protocol like STUN to help traverse firewalls and NAT devices.
- Invalid network handle or PDU ID:
    - Implement robust error handling to gracefully handle and report these types of errors.
    - Validate input parameters before executing any UDP-related operations.
- Null pointer issues:
    - Thoroughly check for null pointers and handle them appropriately to prevent runtime exceptions.

References:

1. https://docs.github.com/en/get-started/quickstart/creating-an-account-on-github
2. https://classroom.github.com/assets/deadline-readme-button-24ddc0f5d75046c5622901739e7c5dd533143b0c8e959d652212380cedb1ea36.svg
3. https://classroom.github.com/a/Y4WP1NMj
4. https://github.com/googleapis/nodejs-storage
5. https://firebase.google.com/docs/storage/web/create-reference
6. https://firebase.google.com/docs/storage/web/create-reference - Firebase documentation on creating a Cloud Storage reference on the web, explaining how to upload, download, delete files, and get or update metadata by creating references to files in the cloud.
7. https://www.slideshare.net/slideshow/project-report-on-cloud-storage/105098832 - A project report on cloud storage describing a student project aimed at developing a cloud-based data storage application, focusing on cloud computing, Java, and the development process to create software for secure data storage and access across devices.