Further Java 2022 Questions

William.D

August 21, 2024

A novice Java developer designs a client-server file download service for the Internet. On start-up the server accepts a directory name on the command line and then operates in a loop, retrieving files from this directory when requested to do so by clients. Clients request the contents of a file by connecting to the server on port 1991 and sending the text string of the filename followed by a newline character. The server responds to a client request for a file by sending the contents of the file as bytes to the client before closing the connection and waiting for the request from the next client. If the client requests a file which does not exist, the server closes the connection.

Part (a)

Sketch Java code for a single-threaded version of the server with blocking I/O as specified above. You do not need to handle failure due to java.io.Exception. You may make use of the Java standard library and you might find the following classes helpful: java.io.BufferedReader, java.io.File, java.io.FileInputStream and java.io.InputStreamReader.

Java Code:

```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class FileDownloadServer {

    // Directory from which files will be served
    private File directory;

    // Constructor that sets the directory
    public FileDownloadServer(String directoryName) {
        this.directory = new File(directoryName);
        if (!this.directory.isDirectory()) {
            throw new IllegalArgumentException("The␣provided␣path␣is␣not␣a␣
                directory.");
        }
    }

    // Main server loop
    public void start() {
```

```java
   try (ServerSocket serverSocket = new ServerSocket(1991)) {
      System.out.println("Server␣started␣on␣port␣1991");

      while (true) {
         try (Socket clientSocket = serverSocket.accept();
             BufferedReader reader = new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));
             OutputStream out = clientSocket.getOutputStream()) {

            // Read the file name from the client
            String fileName = reader.readLine();
            System.out.println("Client␣requested␣file:␣" + fileName);

            // Create a File object for the requested file
            File requestedFile = new File(directory, fileName);

            // Check if the file exists and is a file (not a directory)
            if (requestedFile.exists() && requestedFile.isFile()) {
               // Send the file contents to the client
               try (FileInputStream fileInputStream = new FileInputStream
                  (requestedFile)) {
                  byte[] buffer = new byte[4096];
                  int bytesRead;
                  while ((bytesRead = fileInputStream.read(buffer)) !=
                     -1) {
                     out.write(buffer, 0, bytesRead);
                  }
               }
               System.out.println("File␣sent␣successfully.");
            } else {
               // If the file does not exist, simply close the connection
               System.out.println("Requested␣file␣not␣found:␣" + fileName
                  );
            }

         } catch (IOException e) {
            // Handle exceptions related to a specific client connection
            System.err.println("Error␣handling␣client␣request:␣" + e.
               getMessage());
         }
      }
   } catch (IOException e) {
      // Handle exceptions related to the server socket
      System.err.println("Error␣starting␣server:␣" + e.getMessage());
   }
}

public static void main(String[] args) {
   if (args.length != 1) {
      System.err.println("Usage:␣java␣FileDownloadServer␣<directory>");
      System.exit(1);
   }
```

```
        String directoryName = args[0];
        FileDownloadServer server = new FileDownloadServer(directoryName);
        server.start();
    }
}
```

Part (b)

Describe a significant disadvantage of implementing the server as required for Part (a). How might this be mitigated?

Disadvantage:
The main disadvantage of the server implementation as presented in Part (a) is that it is single-threaded and uses blocking I/O operations. This means the server can only handle one client at a time. If a client is downloading a large file or if the file system is slow, other clients must wait until the current operation completes. This can lead to poor performance, especially under high load, resulting in a poor user experience.
Mitigation Strategies:
To mitigate this disadvantage, the following approaches can be considered:

- **Multi-Threading:** Implement a multi-threaded server where each client request is handled in a separate thread. This allows the server to handle multiple clients simultaneously. However, thread management can become complex, especially under high load.

- **Non-Blocking I/O (NIO):** Use Java's NIO package to handle I/O operations asynchronously. This approach allows the server to handle multiple connections with fewer threads, improving scalability and efficiency. NIO-based servers are more complex to implement but offer better performance under high concurrency.

- **Asynchronous I/O (AIO):** Use Java's Asynchronous I/O package, where I/O operations are performed asynchronously with callbacks. This approach improves the efficiency of I/O-bound operations and is particularly useful for high-performance servers.

Each of these strategies can significantly improve the server's performance and scalability, making it more suitable for real-world applications where multiple clients may request files concurrently.

Part (c)

Another developer suggests the server is re-written to send or receive serialized Java objects in all communications between the client and the server. Briefly outline how this approach would work and provide suitable class definitions for any required serialised objects. Describe one advantage and one disadvantage of this approach compared to the method used in Part (a).

Overview: Using serialized Java objects in client-server communication involves converting Java objects into a byte stream for transmission over a network, and then reconstructing the objects at the receiving end. This approach simplifies the transmission of complex data structures.

How it works:

- **Serialization:** The client sends a request by serializing an object (e.g., a request containing the filename).
- **Deserialization:** The server deserializes the object, processes the request, and sends back a serialized response object (e.g., containing file data or an error message).
- **Example Classes:** Two classes, `FileRequest` and `FileResponse`, can be defined to encapsulate the request and response data.

Advantage: Serialized objects allow for structured data transfer, making it easier to handle complex and organized communication protocols, such as including metadata with file data.
Disadvantage: Serialized objects can introduce compatibility issues between different versions of the client and server applications, requiring careful versioning and maintenance.

Java Code:

```java
import java.io.*;

// Request object
public class FileRequest implements Serializable {
    private String fileName;

    public FileRequest(String fileName) {
        this.fileName = fileName;
    }

    public String getFileName() {
        return fileName;
    }
}

// Response object
public class FileResponse implements Serializable {
    private byte[] fileData;
    private String errorMessage;

    public FileResponse(byte[] fileData) {
        this.fileData = fileData;
    }

    public FileResponse(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public byte[] getFileData() {
        return fileData;
    }

    public String getErrorMessage() {
        return errorMessage;
    }
}
```

Part (d)

Outline in words how you might improve this service so that clients can securely download, upload, modify and delete files stored by the server. Consideration should also be given to communicating server errors to the client.

Improvement Outline:
To enhance the file service for secure downloading, uploading, modifying, and deleting files, while effectively communicating server errors, consider the following:

- **Authentication and Authorization:** Implement user authentication and authorization to control access to the file service based on user roles and permissions.
- **Secure Data Transmission:** Use Transport Layer Security (TLS) to encrypt data transmitted between the client and server, and implement integrity checks to verify file integrity during transfer.
- **File Operations:**
  - **Uploading Files:** Securely upload files with validation for size and type.
  - **Downloading Files:** Validate user permissions and handle file data in chunks for efficiency.
  - **Modifying Files:** Ensure updates are validated and permissions are enforced.
  - **Deleting Files:** Verify user permissions before allowing file deletions.
- **Error Handling and Communication:**
  - **Server Error Responses:** Use standardized error codes and messages to communicate issues to the client.
  - **Client-Side Handling:** Implement error handling on the client side for user feedback and retry mechanisms.

– **Logging and Monitoring:** Introduce server-side logging for tracking and diagnosing issues.

Example Class Definitions:

```
import java.io.*;

// Request object
public class FileRequest implements Serializable {
    private static final long serialVersionUID = 1L;
    private String fileName;
    private String operation; // "upload", "download", "modify", "delete"

    public FileRequest(String fileName, String operation) {
        this.fileName = fileName;
        this.operation = operation;
    }

    public String getFileName() {
        return fileName;
    }

    public String getOperation() {
        return operation;
    }
}

// Response object
public class FileResponse implements Serializable {
    private static final long serialVersionUID = 1L;
    private byte[] fileData;
    private String errorMessage;
    private int errorCode; // Example: 404 for "Not Found", 403 for "Forbidden"

    public FileResponse(byte[] fileData) {
        this.fileData = fileData;
        this.errorMessage = null;
        this.errorCode = 0;
    }

    public FileResponse(String errorMessage, int errorCode) {
        this.errorMessage = errorMessage;
        this.errorCode = errorCode;
        this.fileData = null;
    }

    public byte[] getFileData() {
        return fileData;
    }

    public String getErrorMessage() {
        return errorMessage;
    }
```

```java

    public int getErrorCode() {
        return errorCode;
    }
}
```