CS435: Distributed Systems (242)

Assignment 2: A Distributed File-server using Java RMI

Estimated Time to complete: 2-3 hrs **Weight (3 points)**

Image Created with DALL-E 3

Imagine you and your friends launching your own **Indie Game Studio**, where you're not just coding but building an entire ecosystem for game development. As the CEO and lead developer, you're not just making games—you're also solving real-world software engineering challenges. One of the biggest technical hurdles? Managing game assets efficiently.

A game project isn't just about code; it involves massive files—textures, sound effects, animations, and 3D models. Your team of artists, programmers, and designers needs a secure, private, and high-performance file-sharing system to collaborate seamlessly. To tackle this, you decide to build a distributed file system, designed exclusively for your studio. This system would:

- Enable real-time collaboration – Artists can upload assets, programmers can fetch textures, and designers can preview content without downloading large files.
- Ensure secure access – Only authorized team members can modify or view assets.
- Support direct media streaming – Team members can preview images, play sounds, and test assets directly from the server, without copying files locally.

This is not just game development—it's systems programming, distributed computing, and cybersecurity all rolled into one. As a computer science student, building such an infrastructure could be your ultimate capstone project—one that blends software architecture, networking, and game development into something truly exciting.

Are you ready to build the backbone of your own game studio?

The objective of this assignment is to design and implement a distributed file sharing system using Java RMI, focusing on client-server interaction, file handling, and basic network programming. Students will create a system where clients can connect to a server, browse a list of shared files, and upload/download those files.

Requirements:

1. **Server Component and Interface:**
    - Use RMI to make the `FileServerInterface` available remotely.
    - Handle concurrent client requests. Use threads or a thread pool to avoid blocking.
    - Implement appropriate error handling (e.g., file not found, access denied).
    - Maintains a directory of shared files (on the server)
    - Provides a remote interface (`FileServerInterface`) with the following methods:
        - `getFileList()`: Returns a list of filenames (or file metadata like name, size, last modified date) available for sharing.
        - `downloadFile(String filename)`: Returns the file content (or a stream for larger files) for the specified filename. Consider how to handle large files efficiently (e.g., streaming).
        - `uploadFile(String filename, String fileContent)`: Allows clients to upload files to the server.
2. **Client Component:**
    - Use RMI to connect to the server and invoke remote methods.
    - LIST retrieves the list of available files from the server. Displays the list to the user.
    - GET allows the user to select a file to download. Downloads the selected file from the server. Saves the downloaded file to the client's local file system.
    - PUT allows the user to upload a local file system file to the server.
    - Handle potential exceptions during file transfer.
3. **Implementation Details:**
    - **File Transfer:** For large files, consider using `java.io.InputStream` and `java.io.OutputStream` to stream the file content rather than loading the entire file into memory. This is crucial for performance.
    - **Concurrency:** The server may handle multiple client requests concurrently. Thread safety is essential when accessing shared resources.

When a client connects to the server, the server first reads a one-line command from the client. The command can be:

- The list command "list". In this case, the server responds by sending a list of names of all the files that are available on the server.
- The “get” command, i.e. "get <filename>", where <filename> is a file name.
    - The server checks whether the requested file exists.
    - It sends the contents of the file and closes the connection.
    - Otherwise, it sends a line beginning with the word "error" to the client.
- The “put” command, i.e. “put <filename>”,

    - The server receives the name of the file <filename> and creates a file locally
    - It copies the incoming buffer to the file <filename> and save the file.
    - For any errors, it sends the "error" to the client.
- The server can also respond with the message "unknown command" if the command it reads is not one of the above possible legal commands.

The server should not stop after handling one request; it should continue to accept new requests.

The server program runs from the console as follows:

`java -jar JARFILE <Directory Path> <IPAddress> <Port>`

Sample Run:

`java -jar JARFILE ./indie 192.168.15.10 4567`

For this run, the server program connects to port 4567 and listens. If a list command is received, it shows all the files and directories in **`./indie`** folder. For a get command, it sends the appropriate file to the Client as requested. For a put command, it receives the content of a file and saves it locally.

Export your jar file(s) to multiple clients nodes in the network. Test your file server for the following cases:

- Single client
- 2 to 4 clients
- 2 clients downloading the same file.

A barebone of your project files are available in this zip file. Modify it to complete your assignment.

Assignment Deliverables

The deliverables for the project are the following. These need to be uploaded to LMS.

- A word document (Report) that includes:
    - Java code for all 4 files
    - Conclusion: write about the effect of:
        - Multiple clients requesting list at the same time
        - Multiple clients reading the same file
        - Multiple clients writing the same file
    - Use the following to enhance your report:
        - Screenshots of server responding to client list request
        - Screenshots of server responding to client get request and file content
        - Screenshots of server responding to client put request and file content
        - Screenshots of server responding to erroneous request

Submission and Grading

All submissions are through LMS. Upload the Five deliverables to the LMS.

Grading:

- Concurrent connections (4 clients): 25%
- Client processing of user inputs (ls): 10%
- Client download/creation of a file (get): 10%
- Server download/creation of a file (put): 10%
- Correct usage of classes and related methods: 20%
- Quality of conclusions in the report: 25%

Additional Note:

Students should copy-paste code in the doc file. Screenshots of code are not acceptable.

Any Student would be requested to present their work. The instructor reserves the right to "**interview**" any student on their submission to see the understanding of the submission. The instructor may also ask the student to modify the code to satisfy any test-case(s) there in.

Code Inspection and plagiarism:

The code would be inspected by the instructor. The instructor would use the MOSS tool (https://theory.stanford.edu/~aiken/moss/) to determine the originality of submission.

If the similarity is above 50%, the students would earn ZERO score on the assignment.

Submission Dead-Line:

The submission deadline is final. Late Submissions will be awarded ZERO points.

Important Notes:

- It is the student's responsibility to check/test/verify/debug the code before submission.
- It is the student's responsibility to check/test/verify all submitted work (including jar files)
- It is the student's responsibility to verify that all files have been uploaded to the LMS.
- Incomplete or wrong file types that do not execute will NOT be graded.
- After an assignment/project has been graded, re-submission with an intention to improve an assignments score will not be allowed.
- The instructor has the right to share project execution reports that may have been auto-generated on the course website.