# PROJECT AND TEAM INFORMATION

## Project Title

|  |
| --- |
| Multithreaded Client-Server File Transfer |

## Student / Team Information

|  |  |
| --- | --- |
| Team Name:  Team # | PBL - SD  SE(OS)-VI-T127 |
| **Team member 1 (Team Lead)**  Singh,Jasmeet - 22011017  singhjasmeet200526@gmail.com |  |
| **Team member 2**  Negi,Karan -  22021583  karanegi076@gmail.com |  |
| **Team member 3**  Sati,Anshul – 220211252  [satianshul9@gmail.com](mailto:satianshul9@gmail.com) |  |

# PROPOSAL DESCRIPTION (10 pts)

## Motivation (1 pt)

|  |
| --- |
| In today's world, the need for secure and reliable data transmission over networks is greater than ever. Whether it's transferring sensitive documents, backing up data, or syncing systems across multiple devices, **ensuring the integrity and completeness of files** during transmission is a critical challenge.  We chose this project to explore **real-time file transfer using multithreading over a network socket**, combined with **cryptographic integrity validation (SHA-256)**. By implementing client-server communication using TCP sockets and threading, we aimed to maximize performance and reliability. Furthermore, by incorporating **checksum-based validation**, we address an essential requirement in secure data transmission—**verifying that the file received is exactly the same as the one sent**.  This project not only deepened our understanding of **network programming, multithreading, file I/O**, and **cryptographic hashing**, but also reflects real-world applications in areas like software distribution, cloud storage services, and peer-to-peer systems. It represents a practical solution to a common problem, making it both technically enriching and highly relevant. |

## State of the Art / Current solution (1 pt)

|  |
| --- |
| Currently, there are numerous established solutions for file transfer and verification in both open-source and commercial domains. Tools like **rsync**, **scp**, **FTP**, and cloud-based services (e.g., Google Drive, Dropbox) handle file transmission over networks with varying levels of performance, encryption, and reliability.  These systems typically use **TCP/IP protocols** for reliable communication and may include integrity checks using **MD5**, **SHA-1**, or **SHA-256** hash algorithms to ensure that the file received matches the file sent. For example, **rsync** efficiently synchronizes files and supports checksum-based verification to detect corruption or mismatch during transfer.  On a lower level, file transfer protocols such as **HTTP**, **SFTP**, and **TFTP** provide mechanisms for segmented transmission, retries, and partial downloads, but are often implemented using large libraries or external dependencies.  While these tools are powerful, they are often black-box systems—offering limited customization or visibility into internal operations like threading behavior or segment handling. This project explores a more **lightweight, customizable approach**, where we implement multithreaded file transfer and SHA-256 checksum validation, giving us full control over how data is sent, received, and verified. |

## Project Goals and Milestones (2 pts)

|  |
| --- |
| **Project Goals:**   * Develop a **Client-Server architecture** using TCP sockets in C. * Implement **multithreaded communication** for concurrent file segment transfer. * Ensure **data integrity** using **SHA-256 checksum verification**. * Allow the **client to reassemble** received segments into a complete file. * Perform **checksum comparison** between client and server to confirm successful transfer. * Use **mutex locks** to safely manage concurrent file writes during multi-threaded execution.  **Project Milestones:**  * Set up the **development environment** (GCC, OpenSSL, socket headers). * Establish a **basic TCP socket connection** between client and server. * Implement **multithreading** on both client and server sides. * Perform **file I/O operations** for segment reading and writing. * Integrate **SHA-256 checksum calculation** using OpenSSL EVP APIs. * Use **mutex locking** to handle thread synchronization during file writing. * Conduct **testing and debugging** with various file sizes and thread counts. |

## Project Approach (3 pts)

|  |
| --- |
| **Client-Server Architecture**   * Built using TCP sockets to enable reliable, connection-oriented communication between client and server. * Server listens for incoming connections and handles requests for file transfers.   **Multithreading for Efficiency**   * The server uses multiple threads to divide the file into segments and send them concurrently. * The client creates corresponding threads to receive and write segments to the output file simultaneously. * Threads are managed using the pthread library for parallelism and performance.   **File Segmentation and Transmission**   * Files are read in chunks (e.g., 1024 bytes per segment) on the server side. * Each segment is transmitted over the socket to the client thread responsible for writing it.   **Data Synchronization with Mutex Locks**   * A pthread\_mutex\_t lock ensures that file writes are not corrupted when multiple threads write to the same file. * Only one thread writes at a time to prevent race conditions.   **Checksum Generation and Verification**   * Before sending the file, the server computes a **SHA-256** hash using the OpenSSL EVP interface and sends it to the client. * After receiving and assembling the file, the client calculates its own SHA-256 hash. * The two hashes are compared to validate the **integrity** of the transferred file.   **Error Handling and Debugging**   * Includes checks for socket creation, connection failures, file I/O issues, and hashing errors. * Debug messages (e.g., bytes received, hash comparison) are printed for clarity and troubleshooting.   **Modularity and Reusability**   * Functions are divided clearly: hashing, segment handling, thread management, and socket communication. * The code is written to be extendable for future features such as encryption, file compression, or GUI support. |

## System Architecture (High Level Diagram) (2 pts)

|  |
| --- |
|  |

## Project Outcome / Deliverables (1 pts)

|  |
| --- |
| The deliverables include:   * A **working Client-Server application** built in C using TCP sockets for reliable file transfer. * Implementation of **multithreaded data transmission** to handle file segments concurrently. * Successful **file reassembly** on the client side from received segments. * **SHA-256 checksum verification** to ensure end-to-end file integrity. * Clear **console-based output logs** showing file transfer status and hash comparison results. * A well-documented **C source codebase** with comments and modular functions. |

# Assumptions

|  |
| --- |
| * The client and server **run on the same local network** or localhost (127.0.0.1) for testing purposes. * The **file to be transferred exists** on the server and has appropriate read permissions. * The number of threads provided by the client is a **positive integer** and within practical system limits. * The **network delay or latency is minimal**, allowing real-time communication between threads. * Only **one client is connected to the server** at a time during file transfer. * All required libraries (e.g., OpenSSL for hashing) are **properly installed** and linked. |

## References

|  |
| --- |
| * OpenSSL Project – Secure Sockets Layer toolkit 🔗 <https://github.com/openssl/openssl> * Stack Overflow: Alternatives to deprecated OpenSSL functions for SHA-256 🔗 <https://stackoverflow.com/questions/34289094/alternative-for-calculating-sha256-to-using-deprecated-openssl-code> * Deprecation of OpenSSL APIs 🔗 <https://docs.openssl.org/master/man3/SHA256_Init/> * Beej's Guide to Network Programming – Socket programming basics 🔗 <https://beej.us/guide/bgnet/> * Multithreading 🔗<https://www.geeksforgeeks.org/multithreading-in-c/> * GeeksforGeeks – File handling and socket programming in C 🔗 <https://www.geeksforgeeks.org/socket-programming-cc/> 🔗 <https://www.geeksforgeeks.org/file-handling-c-classes/> |