NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES
(KARACHI CAMPUS)
FAST School of Computing
Spring 2025

TEAM MEMBERS:

Aisha Asif – 23K-0915
Nabira Khan– 23K-0914
Rameen Zehra - 23K-0501

SECTION:
BCS-4C

INSTRUCTOR:

Sir Abdul Rahman

ABSTRACT

ByteFlow-FTP is a secure, multithreaded File Transfer Protocol (FTP) system developed in C with SSL encryption and SQLite-based user authentication. Designed as a practical application for mastering systems programming, the project features a thread pool, synchronization primitives (mutexes, semaphores, read-write locks), and client-server interaction over secure sockets. ByteFlow-FTP supports file operations such as uploading, downloading, listing directories, and user management with role-based privileges (admin/user). This report details the objectives, architecture, implementation strategy, challenges encountered, and solutions adopted to ensure a secure and efficient file transfer experience. It demonstrates how systems-level programming can be harnessed to create robust network applications.

INTRODUCTION

ByteFlow-FTP addresses the challenge of building a real-world, secure, and concurrent file transfer system. Developed in C, it utilizes low-level constructs like POSIX threads and inter-process communication along with high-level components such as SQLite and OpenSSL. The goal was to enable users to connect over SSL, authenticate via a secure database, and perform various file operations within their home directories. The server uses a thread pool to manage simultaneous clients efficiently, ensuring scalability and responsiveness. The client application provides an intuitive interface for users and administrators, offering commands such as ls, get, put, cd, pwd, mkdir, and rmdir, while administrators can manage users securely. The project showcases core concepts in operating systems, networking, and concurrency.

BACKGROUND (RESEARCH AND PROJECT SELECTION)

The ByteFlow-FTP project was conceptualized to integrate and apply a wide range of operating systems and systems programming concepts into a secure, real-world network application. Rather than relying on conventional FTP solutions that often lack encryption and concurrency safety, ByteFlow-FTP was built from the ground up to provide robust, scalable, and secure file transfer capabilities tailored for educational and enterprise use.

This project brings together a multitude of advanced techniques:

- OpenSSL provides TLS/SSL encryption, ensuring all file transfers and user credentials are protected over the network.
- SQLite is used as a lightweight embedded database to manage users, passwords, and home directories, enabling persistent and secure authentication.
- A custom command-response protocol was designed to handle structured communication between client and server.
- A POSIX thread pool enables the server to efficiently manage multiple clients concurrently, avoiding thread explosion.
- Fork-based process isolation is used for authentication, mitigating potential security breaches by sandboxing sensitive database operations.
- Reader-writer locks are employed to support simultaneous downloads without conflict, ensuring thread-safe file access.
- Binary semaphores prevent race conditions during file uploads, ensuring only one active upload per file.
- IPC and signal handling are integrated to support clean shutdowns, zombie process prevention, and inter-thread communication.
- Concepts from synchronization challenges like Dining Philosophers, Reader-Writer Problems, and Bounded Buffer are embedded in the solution's concurrency handling strategies.

The project was chosen to unify theory and practice, reinforcing fundamental OS concepts such as memory management, IPC, multithreading, file systems, synchronization, and network sockets. The rich combination of features, technologies, and real-world problem-solving made ByteFlow-FTP an ideal capstone to demonstrate applied systems programming.

PROJECT SPECIFICATIONS

FEATURE	DESCRIPTION
SSL Encrypted Connection	Ensures all data is securely transmitted between client and server.
Thread Pool	Handles multiple client connections concurrently using worker threads.
Admin/User Privileges	Admins can add/delete/list users, while normal users access only their directories.
File Upload/Download	Users can upload to and download from their respective home directories.
Directory Management	Supports mkdir, rmdir, cd, pwd commands for file system navigation.
SQLite User Database	Stores credentials and home directories, preventing unauthorized access.
Synchronization Primitives	Uses mutexes, semaphores, and read-write locks to avoid race conditions.

PROCEDURE / FUNCTION NAME	PURPOSE
init_database	Initializes SQLite and creates user table. Adds default admin user.
authenticate_in_process	Authenticates users in a forked child process using shared memory and semaphores.
handle_client	Main request handler for each client. Parses and processes commands.
enqueue_client / dequeue_client	Adds/removes clients from the work queue using mutex-protected circular buffer.
thread_function	Each worker thread executes this to process client tasks from the queue.
create_ssl_context / load_certificates	Initializes SSL context and loads TLS certificates.
add_user / delete_user / list_users	Admin-only database operations for managing user access.
upload / download handlers	Perform file transfers with proper locks to prevent race conditions.

PROBLEM ANALYSIS

PROBLEM	IMPACT	ANALYSIS	SOLUTION
Concurrent Upload/Download Conflicts	Race conditions and file corruption	Without synchronization, multiple clients could write or read simultaneously	Used semaphores for exclusive uploads and RWLocks to allow multiple downloads
Authentication Vulnerability	Potential SQL injection and insecure access	Direct queries with user input posed security risks	Used prepared SQLite statements and isolated authentication in a forked process
File Path Traversal	Security breach allowing access outside home directories	Clients could send paths like ../ to escape sandbox	Validated paths using realpath and ensured directory was under home path
Client Overload	Server could crash under many connections	Unbounded client handling without threading limits	Implemented bounded thread pool and condition variables for queuing

SOLUTION DESIGN

1. Modular Architecture & Procedure Design

- The ByteFlow-FTP system was structured using a modular, layered approach to separate concerns and improve maintainability. Core components include:
- Server Core: Manages client connections, SSL handshakes, and command execution.
- Client Interface: Provides a simple command-line interface for file and directory operations.
- Authentication Module: Isolated in a forked child process for secure, sandboxed database access.
- Command Handler: Processes operations like get, put, cd, mkdir, etc., using a unified protocol structure.

All operations are encapsulated in clearly defined procedures (e.g., handle_client(), enqueue_client(), upload_file()) to promote reuse, ease debugging, and ensure scalability.

---

2. Threading and Concurrency Control

Concurrency was a major design pillar of ByteFlow-FTP:

- A fixed-size thread pool (POSIX threads) processes multiple client requests without spawning excessive threads.
- A circular queue buffers incoming client connections and is guarded by a mutex and condition variable to synchronize producer-consumer behavior.
- File access concurrency is managed through:
  - Reader-writer locks allowing multiple concurrent downloads (pthread_rwlock_rdlock) while blocking them during uploads (pthread_rwlock_wrlock).
  - A binary semaphore (sem_t upload_sem) ensures only one upload occurs at a time, preventing file corruption.

This mirrors classic synchronization challenges like Dining Philosophers and Reader-Writer Problems, applying real solutions like priority-based resource access and deadlock prevention.

---

3. Secure Communication via SSL

The project uses OpenSSL to establish a TLS-encrypted connection between clients and the server. Both certificate (cert.pem) and private key (key.pem) are loaded on the server side:

- The SSL context is initialized using TLS_server_method() and SSL_new().
- All communication, including commands and file transfers, happens via SSL_read() and SSL_write() to ensure end-to-end encryption.

This prevents packet sniffing and man-in-the-middle attacks, making ByteFlow-FTP viable for use on insecure networks.

---

4. Custom Protocol and Data Structures

A custom command-response protocol defines a consistent structure for communication:

- Commands are defined in a shared command_t struct (e.g., type, path, username).
- Files are transferred using file_transfer_t, which encapsulates metadata like filename and filesize.
- Responses and status messages follow predictable tags (e.g., END_LIST, UPLOAD_COMPLETE) for easy client-side parsing.

This approach maintains clarity while allowing future extensibility (e.g., resume support, directory sync).

---

5. User Authentication and Privilege Control

User credentials are securely stored in an SQLite3 database, including home directory paths and hashed (or plaintext for now) passwords.

- Authentication is handled by a forked child process with IPC via shared memory and a semaphore, ensuring isolation of database queries.
- Admin privileges are recognized by username, allowing restricted access to sensitive commands (e.g., adduser, deluser, listusers).

This mirrors sandboxing techniques and simulates multi-user permission models seen in real UNIX systems.

---

6. File System Safety and Path Validation

To prevent directory traversal attacks and unauthorized access:

- Every cd, mkdir, and rmdir operation resolves paths using realpath() to normalize them.
- A security check confirms that the resolved path lies within the user's home directory, rejecting access otherwise.
- File operations are always relative to the home directory, sandboxing each user session.

---

7. Signal and Interrupt Handling

To ensure graceful termination and cleanup:

- The server traps SIGINT and SIGTERM to destroy synchronization primitives, terminate threads, and close the database.
- Zombie processes are cleaned up using waitpid() in a loop after fork-based authentication.
- Threads exit cleanly when shutdown_server is set, ensuring no dangling threads or leaks.

---

8. Performance and Resource Optimization

- Thread pool prevents the overhead of frequent thread creation.
- Minimal disk I/O is ensured by buffered file transfers and chunked reads/writes.
- Unnecessary memory allocation is avoided by using statically allocated command and transfer structs.
- The use of lightweight SQLite instead of a full-scale DBMS makes the system fast and portable.

---

9. Error Handling and Debugging Aids

- All SSL, socket, and file operations include error checks with fallback or cleanup behavior.
- Invalid user input or file paths trigger informative responses (e.g., CD_FAILED, UPLOAD_FAIL).
- Server-side errors are logged using perror() or fprintf(stderr, ...) to aid debugging.

IMPLEMENTATION AND TESTING

ByteFlow-FTP was implemented in C using POSIX threads, OpenSSL, and SQLite, emphasizing secure, efficient, and modular system design. The server employs a multithreaded model with a thread pool, while the client provides a command-line interface for executing FTP operations. SSL certificates ensure encrypted communication, and SQLite handles user authentication securely. Thread synchronization primitives, such as mutexes, semaphores, and read-write locks, ensure consistent behavior under concurrent access.

Testing focused on functionality, concurrency handling, and security:

- **Unit Testing** validated key functions including file upload/download handlers, user database operations, and command parsing.
- **Integration Testing** verified consistent interaction between modules—SSL communication, user authentication, file system operations, and thread management.
- **Concurrency Testing** simulated multiple simultaneous clients uploading/downloading files to ensure stability, lock correctness, and fairness.
- **Security Testing** confirmed that unauthorized access (e.g., to other users' directories or admin functions) was properly restricted through path validation and privilege checks.
- **Real-World Simulation** was conducted using the client to perform all supported commands in varied sequences to ensure reliability under normal and edge-case usage.

PROJECT BREAKDOWN STRUCTURE

- **Design and Planning:** 2 days
- **Code Implementation:** 2 weeks
- **Feature Integration and Debugging:** 2 days
- **Testing and Optimization**: 2 days
- **Documentation and Final Review:** 1 day

For efficient project management and task tracking, a dynamic spreadsheet was utilized to monitor progress, identify bottlenecks, and ensure adherence to timelines. Regular checkpoints and reviews were conducted to address challenges promptly, ensuring the project's success within the given timeframe.

CONCLUSION

The ByteFlow-FTP project successfully demonstrates the integration of core systems programming concepts into a secure, robust, and scalable file transfer application. By combining POSIX threading, secure socket programming with OpenSSL, and SQLite-based user authentication, the project delivers a feature-rich FTP system tailored for real-world use.

From multithreaded client handling using a thread pool to sandboxed authentication via forked processes, the system effectively manages concurrency, security, and user access. Admin-specific functionalities, such as user management and access control, add an additional layer of privilege separation and operational integrity.

Throughout development, challenges such as race conditions, file access conflicts, and input validation were addressed through appropriate synchronization mechanisms and rigorous testing. The final implementation reflects not only technical proficiency in C and operating systems but also a thoughtful design geared toward safety, maintainability, and extensibility.

ByteFlow-FTP stands as a practical demonstration of how foundational OS principles — threads, semaphores, sockets, IPC, and signals — can be applied cohesively to solve complex real-world problems in systems and network software development.