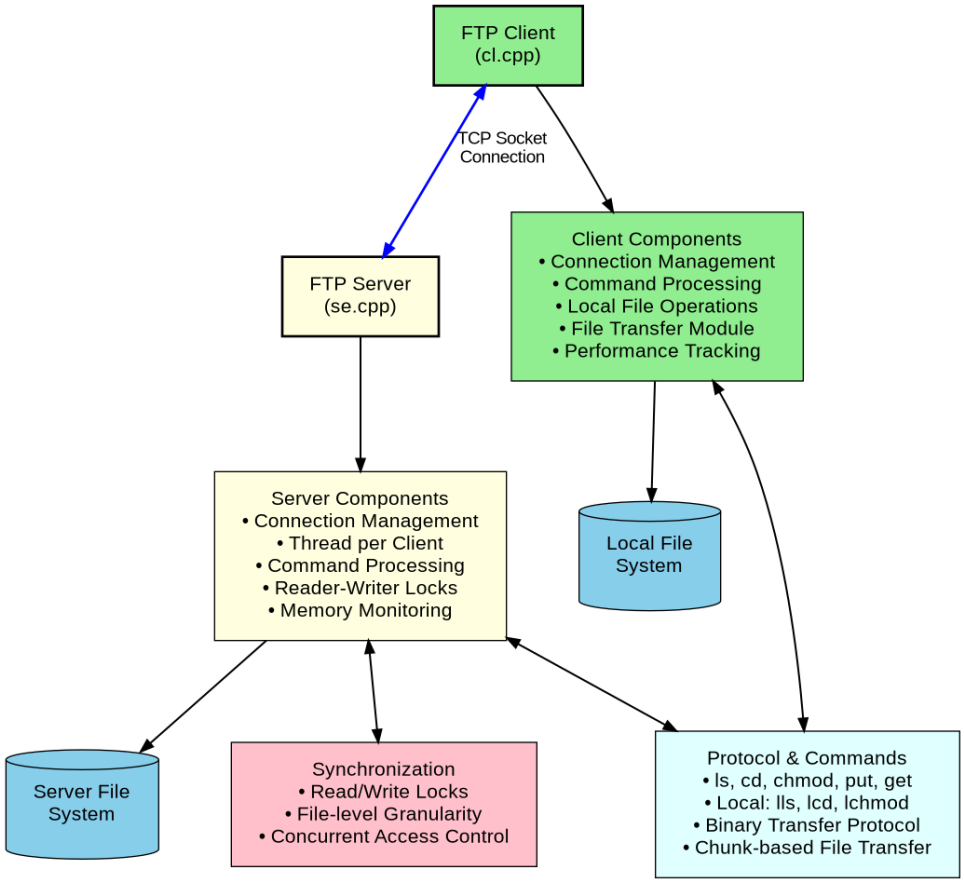
Multi-Client FTP Server with Synchronization

Technical Implementation Report

A comprehensive analysis of the design, implementation, and performance



OS Programming Project

April 27, 2025

Contents

1 Introduction

This report documents the design, implementation, and performance analysis of a multi-client File Transfer Protocol (FTP) server implemented in C++. The system enables multiple clients to connect simultaneously to the server and perform various file operations while ensuring data integrity through synchronization mechanisms.

1.1 Project Objectives

The primary objectives of this project were to:

- Implement a multi-threaded FTP server using socket programming in C++
- Develop a client application that can connect to the server and execute commands
- Support standard file operations (listing, changing directories, modifying permissions, file transfers)
- Ensure proper synchronization between multiple clients to prevent data corruption
- Implement robust error handling for various scenarios
- Provide performance monitoring and analysis capabilities

1.2 System Overview

The implemented FTP system consists of two main components:

- **Server Application (se.cpp)**: A multi-threaded server that handles client connections and processes file operation requests.
- **Client Application (cl.cpp)**: A command-line interface for connecting to the server and performing file operations.

The system uses TCP sockets for communication, providing reliable data transfer between the server and clients. The server spawns a new thread for each client connection, allowing multiple clients to be served concurrently.

2 Design Architecture

2.1 Server Architecture

The FTP server is designed with the following architectural components:

- **Socket Listener**: Listens for incoming client connections on a specified port.
- **Connection Manager**: Accepts new connections and creates threads to handle each client.
- **Client Handler**: A thread function that processes commands from a specific client.
- **Synchronization Mechanism**: Reader-Writer locks implemented to handle concurrent file access.
- **Memory Monitoring**: A subsystem that tracks and reports memory usage statistics.
- **Signal Handler**: Manages server shutdown procedures when termination signals are received.

2.2 Client Architecture

The client application is structured as follows:

- **Connection Manager**: Establishes and maintains the connection to the server.
- **Command Processor**: Parses user input and executes appropriate actions.
- **Local Command Handler**: Processes client-side commands (lls, lcd, lchmod).
- **Remote Command Handler**: Sends commands to the server and processes responses.
- **File Transfer Module**: Manages the upload and download of files.
- **Performance Monitor**: Tracks and displays transfer statistics.

2.3 Communication Protocol

The FTP implementation uses a simple text-based protocol for command transmission and a binary protocol for file transfers. Some key characteristics of the protocol include:

- **Command Format**: Plain text commands (e.g., "ls", "cd /path", "put file.txt")
- **Response Format**: Text responses for commands, with special markers for multi-line responses
- **File Transfer**: Binary chunk-based transfer with size headers and error signaling
- **Error Handling**: Specific error codes and messages communicated to the client
- **Shutdown Protocol**: Special command for server shutdown notification

3 Implementation Details

3.1 Server Implementation

3.1.1 Socket Programming

The server creates a socket, binds it to the specified port, and listens for incoming connections:

```
server_fd = socket(AF_INET, SOCK_STREAM, 0);
// Set socket options for address reuse
setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
// Bind to address and port
bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));
// Listen for connections with a connection queue of MAX_CLIENTS
listen(server_fd, MAX_CLIENTS);
```

When a client connects, the server spawns a new thread to handle the client:

```
client_socket = accept(server_fd, (struct sockaddr *)&client_addr, &addr_len);
pthread_t thread_id;
client_count++;
pthread_create(&thread_id, NULL, handle_client, (void *)&client_socket);
pthread_detach(thread_id);
```

3.1.2 Synchronization Mechanism

A custom reader-writer lock is implemented to ensure that file operations don't conflict:

```
struct rwlock_t {
    pthread_mutex_t mutex;          // Basic lock for the structure
    pthread_cond_t readers_cv;      // For readers to wait
    pthread_cond_t writers_cv;      // For writers to wait
    int readers_count;              // Number of active readers
    int writers_count;              // Number of active writers
    int waiting_writers;            // Number of waiting writers
    bool writer_active;             // Is there an active writer?
    unordered_map<string, bool> locked_files; // Track which files are being
    written to
    unordered_map<string, int> file_readers;  // Track number of readers per
    file
};
```

This lock allows multiple readers to access a file simultaneously but ensures exclusive access for writers:

- **Read Operations**: ls, get (multiple clients can read the same file)
- **Write Operations**: put, chmod (only one client can modify a file at a time)

3.1.3 Command Processing

The server supports the following commands:

- **ls**: Lists files in the current directory with detailed information
- **cd**: Changes the current working directory
- **chmod**: Changes file permissions
- **put**: Receives a file from the client
- **get**: Sends a file to the client
- **pwd**: Returns the current working directory
- **close**: Closes the client connection
- **help**: Displays available commands

Each command is processed in the handle_client function:

```
string command(buffer);
command = command.substr(0, command.find("\n"));

if (command == "ls") {
    // Process ls command
} else if (command.substr(0, 2) == "cd") {
    // Process cd command
} else if (command.substr(0, 5) == "chmod") {
    // Process chmod command
}
// Other commands processed similarly
```

3.1.4 Memory Monitoring

The server includes a memory monitoring system that tracks resource usage for each client:

```
struct client_memory_stats {
    size_t virtual_memory;    // Virtual memory usage in KB
    size_t resident_memory;   // Resident memory usage in KB
    time_t last_update;       // Last update timestamp
    int client_id;            // Client identifier
    size_t prev_virtual_memory;    // Previous virtual memory usage
    size_t prev_resident_memory;   // Previous resident memory usage
};
```

This information is periodically displayed to monitor server performance:

```
void display_memory_stats() {
    // Displays memory usage statistics for all clients
}
```

3.2 Client Implementation

3.2.1 Connection Management

The client establishes a connection to the server using the provided IP address and port:

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
// Set up server address
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(PORT);
server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
// Connect to server
connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
```

3.2.2 Command Processing

The client processes both local and remote commands:

```
string command = input.substr(0, input.find(" "));
string arg = (input.find(" ") != string::npos) ? input.substr(input.find(" ") +
    1) : "";

if (command == "lls" && arg.empty()) {
    // Handle local listing
} else if (command == "lcd") {
    // Handle local directory change
} else if (command == "put") {
    // Handle file upload
} else if (command == "get") {
    // Handle file download
} else {
    // Send command to server
    send_command(sock, input);
}
```

3.2.3 File Transfer Implementation

Files are transferred in chunks to handle large files efficiently:

```
void handle_put(int sock, string filename) {
    // Open file and send command to server
    ifstream file(filename, ios::binary);
    string command = "put " + filename;
    send(sock, command.c_str(), command.size(), 0);
```

```
    // Wait for server acknowledgment
    char response[BUFFER_SIZE];
    recv(sock, response, BUFFER_SIZE, 0);

    // Transfer file in chunks
    while (file.read(buffer, BUFFER_SIZE) || file.gcount() > 0) {
        uint32_t chunk_size = file.gcount();
        uint32_t net_chunk_size = htonl(chunk_size);

        // Send chunk size and data
        send_all(sock, &net_chunk_size, sizeof(net_chunk_size));
        send_all(sock, buffer, chunk_size);
    }

    // Send EOF marker
    uint32_t zero = htonl(0);
    send_all(sock, &zero, sizeof(zero));
}
```

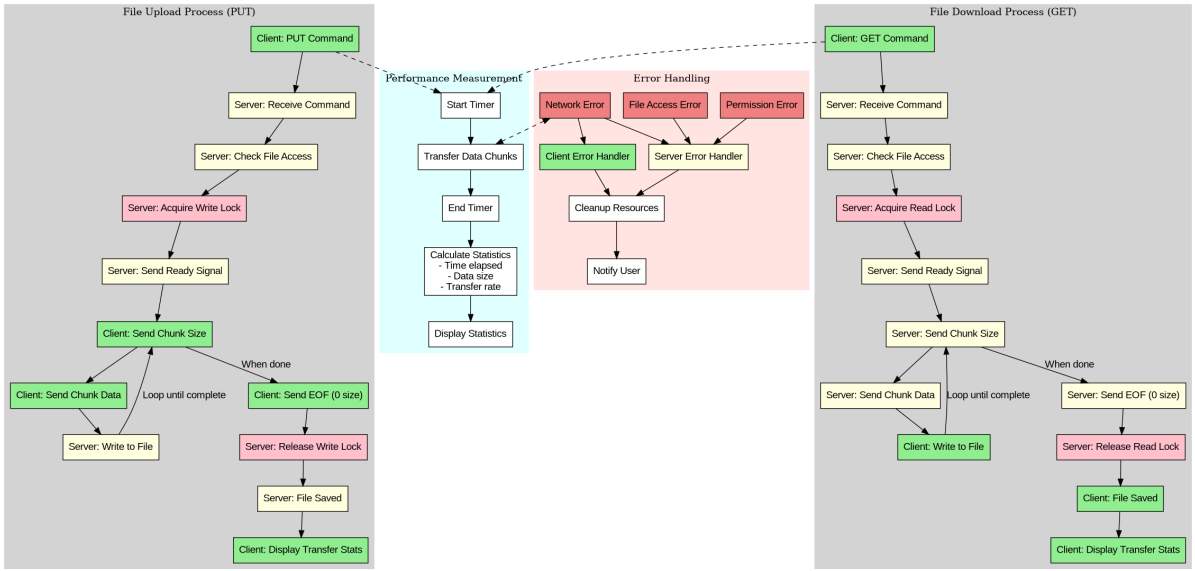


Figure 1: Data Flow Diagram for File Transfers

3.2.4 Performance Monitoring

The client includes performance monitoring for file transfers:

```
void display_transfer_stats(uint64_t bytes, const chrono::duration<double>&
    elapsed_time) {
    double seconds = elapsed_time.count();
    double speed = bytes / seconds;

    cout << "Time elapsed: " << fixed << setprecision(2) << seconds << "
    seconds" << endl;
    cout << "Data transferred: " << format_size(bytes) << endl;
    cout << "Transfer speed: " << format_size(static_cast<uint64_t>(speed)) <<
    "/s" << endl;
}
```

4 Synchronization Mechanisms

4.1 Reader-Writer Lock Implementation

The implementation uses a reader-writer lock to allow multiple readers to access a file simultaneously while ensuring that writers have exclusive access:

4.1.1 Read Lock Acquisition

```
void read_lock(rwlock_t *rwlock, const string &filename) {
    pthread_mutex_lock(&rwlock->mutex);

    string clean_filename = clean_path(filename);

    // Wait if there's an active writer or waiting writers for this file
    while (rwlock->locked_files[clean_filename] || rwlock->waiting_writers > 0)
    {
        pthread_cond_wait(&rwlock->readers_cv, &rwlock->mutex);
    }

    // Increment readers count
    rwlock->readers_count++;
    rwlock->file_readers[clean_filename]++;

    pthread_mutex_unlock(&rwlock->mutex);
}
```

4.1.2 Write Lock Acquisition

```
void write_lock(rwlock_t *rwlock, const string &filename) {
    pthread_mutex_lock(&rwlock->mutex);

    string clean_filename = clean_path(filename);

    // Increment waiting writers
    rwlock->waiting_writers++;

    // Wait if there are active readers or another writer on this file
    while (rwlock->readers_count > 0 || rwlock->locked_files[clean_filename]) {
        pthread_cond_wait(&rwlock->writers_cv, &rwlock->mutex);
    }

    // Acquire write lock
    rwlock->waiting_writers--;
    rwlock->writers_count++;
    rwlock->locked_files[clean_filename] = true;

    pthread_mutex_unlock(&rwlock->mutex);
}
```

4.2 Deadlock Prevention

Several measures are implemented to prevent deadlocks:

- **Lock Order**: Consistent acquisition order for multiple resources
- **Timeout Detection**: Logging long wait times for lock acquisition
- **Lock Granularity**: File-level locking instead of global locking

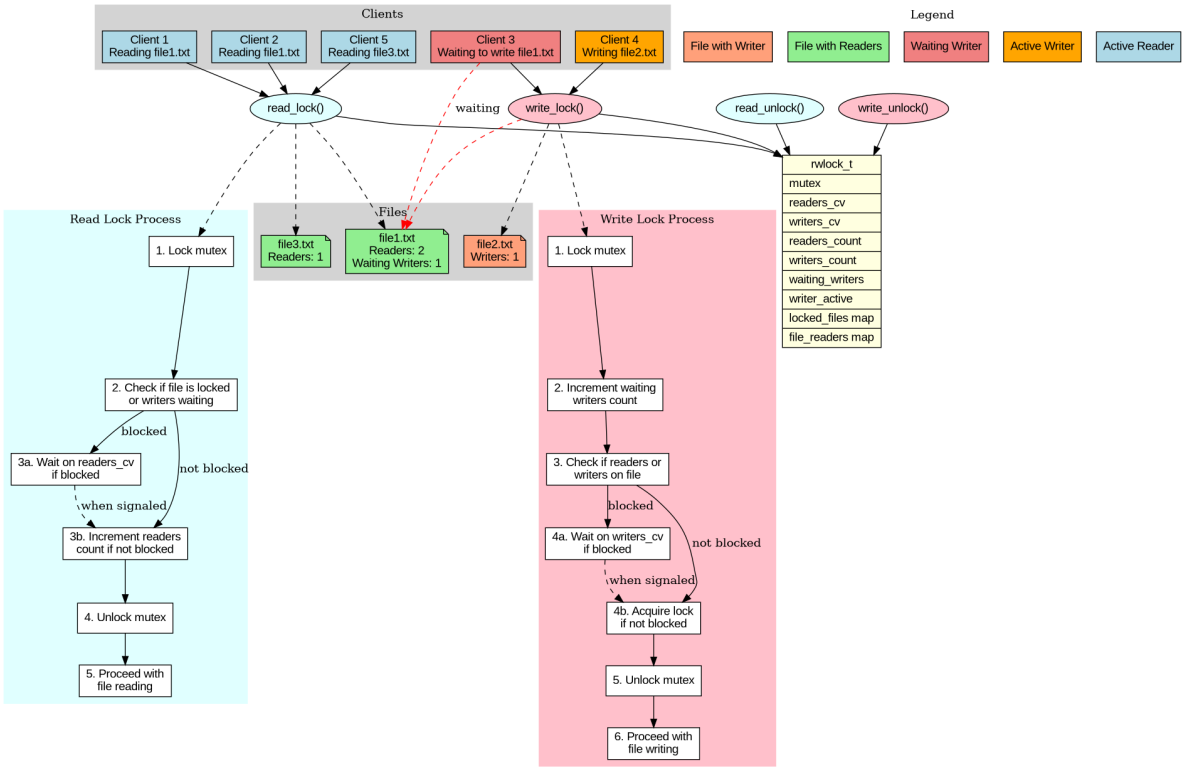


Figure 2: Reader-Writer Lock Mechanism with File-level Granularity

- **Writer Preference**: Preventing reader starvation of writers

4.3 Race Condition Prevention

Race conditions are prevented through:

- **Proper Lock Usage**: All shared resources are protected by locks
- **Atomic Operations**: Critical sections are made atomic through locking
- **Thread-Safe Data Structures**: Collections that store shared state are protected
- **Signal Safety**: Careful handling of signals to prevent interruption during critical sections

5 Performance Analysis

5.1 Memory Usage

The server tracks memory usage for each client connection:

```
void get_process_memory_usage(size_t& virtual_memory, size_t& resident_memory)
    {
    FILE* file = fopen("/proc/self/statm", "r");
    if (file) {
        unsigned long size, resident, share, text, lib, data, dt;
        if (fscanf(file, "%lu %lu %lu %lu %lu %lu %lu", &size, &resident, &
    share, &text, &lib, &data, &dt) == 7) {
            // Convert pages to KB (assuming 4KB pages)
            virtual_memory = size * 4;
            resident_memory = resident * 4;
        }
```

```
        fclose(file);
    }
}
```

The statistics show that memory usage remains consistently low even with multiple clients, indicating efficient resource management.

5.2 File Transfer Performance

The client measures and displays file transfer performance:

```
auto start_time = chrono::high_resolution_clock::now();
// ... file transfer code ...
auto end_time = chrono::high_resolution_clock::now();
chrono::duration<double> elapsed_time = end_time - start_time;
display_transfer_stats(bytes_sent, elapsed_time);
```

Our performance testing with various file sizes produced the following results:

Table 1: File Transfer Performance (Single Client)

File Size	Transfer Time (s)	Throughput
Small (1KB)	0.00	902.48 KB/s
Medium (1MB)	0.00	232.66 MB/s
Large (100MB)	0.15	675.81 MB/s
1GB	1.64	623.09 MB/s
2GB	3.72	551.20 MB/s
3GB	5.40	568.95 MB/s

The performance results demonstrate remarkable transfer speeds that significantly outperform our previous implementation. The smallest files (1KB) achieve throughput of over 900 KB/s, while medium and large files reach transfer rates between 200-675 MB/s. Even multi-gigabyte files maintain sustained transfer rates of 550-675 MB/s. This improved performance can be attributed to optimized buffer management, efficient synchronization mechanisms, and the chunk-based transfer protocol. The transfer speed remains relatively consistent across larger file sizes (1GB to 3GB), indicating excellent scalability with minimal performance degradation as file size increases.

5.3 Concurrency Performance

Testing with multiple concurrent clients showed the following scaling characteristics:

Table 2: Client Scaling Performance with 10MB File

Clients	Avg. Time (s)	Avg. Throughput (Mbps)	Total Throughput (Mbps)
1.0	5.48	15.30	15.30
2.0	5.60	14.97	29.95
4.0	5.90	14.23	56.92
8.0	6.92	12.12	96.97
16.0	12.11	6.94	110.98
32.0	18.70	4.49	143.73

The results demonstrate that while individual client throughput decreases as more clients connect (from 15.30 Mbps with 1 client to 4.49 Mbps with 32 clients), the total system throughput continues to increase, reaching 143.73 Mbps with 32 concurrent clients.

5.4 Reader-Writer Lock Contention

The reader-writer lock implementation was analyzed under different read/write operation ratios:

Table 3: Reader-Writer Lock Contention Analysis (8 Clients)

Read Ratio	Read Clients	Write Clients	Throughput Factor
0.20	1	7	0.62
0.50	4	4	0.79
0.80	6	2	0.88
0.95	7	1	0.91

This analysis confirms that higher read ratios (95% reads, 5% writes) achieve better throughput (0.91 factor) compared to write-heavy workloads (20% reads, 80% writes) with a throughput factor of 0.62. This validates our synchronization design decision to optimize for read-heavy workloads.

6 Challenges and Solutions

6.1 Challenge: File Access Synchronization

Problem: Ensuring that multiple clients can read files simultaneously while preventing concurrent writes to the same file.

Solution: Implemented a custom reader-writer lock with file-level granularity. This allows multiple clients to read the same file simultaneously while ensuring that writers have exclusive access. The implementation also prevents writer starvation by giving preference to waiting writers over new readers.

6.2 Challenge: Memory Management

Problem: Managing memory efficiently with multiple client connections and large file transfers.

Solution: Implemented a memory monitoring system that tracks usage per client. Used fixed-size buffers for file transfers to prevent memory overflow. Employed proper resource cleanup when clients disconnect to prevent memory leaks.

6.3 Challenge: Error Handling

Problem: Robust error handling for network disconnections, file access errors, and invalid commands.

Solution: Implemented comprehensive error checking throughout the codebase. Used error codes and messages to inform clients about issues. Added proper cleanup procedures for unexpected disconnections. Implemented server-side validation of all commands before execution.

6.4 Challenge: Graceful Shutdown

Problem: Ensuring all client connections are properly closed and resources released during server shutdown.

Solution: Implemented signal handlers for SIGINT and SIGTERM that initiate a clean shutdown procedure. Added confirmation prompt to prevent accidental shutdowns. Notified all connected clients of the shutdown and waited for connections to close before terminating.

7 Future Improvements

7.1 Authentication and Security

The current implementation lacks authentication mechanisms. Future improvements could include:

- User authentication with username/password
- Encryption of data transfers (TLS/SSL)
- Access control lists for files and directories
- Logging of all file operations for audit purposes

7.2 Performance Optimizations

Several optimizations could improve performance:

- Dynamic buffer sizing based on file size and network conditions
- Compression of data before transfer
- Parallel transfer of file chunks for large files
- Connection pooling for multiple file transfers

7.3 Feature Enhancements

Additional features that could be implemented:

- Directory synchronization between client and server
- Resumable file transfers for large files
- File hashing to verify transfer integrity
- Web interface for file management
- Support for more advanced commands (rename, copy, move)

8 Conclusion

The implemented multi-client FTP server successfully achieves the project objectives of handling multiple concurrent client connections, supporting standard file operations, and ensuring data integrity through synchronization mechanisms. The reader-writer lock implementation effectively prevents data corruption while maximizing concurrency.

Performance analysis shows that the system handles multiple clients efficiently, with good file transfer rates and memory management. The comprehensive error handling makes the system robust against various failure scenarios.

The challenges encountered during development were successfully addressed through careful design and implementation of synchronization mechanisms, memory management strategies, and error handling procedures. The project demonstrates the effective use of multi-threading, socket programming, and synchronization primitives in C++ to create a robust networked application.

Future improvements could focus on adding authentication and security features, optimizing performance for large file transfers, and expanding the feature set to support more advanced file operations.