# Project 1 Readme

## Overview

This README serves as a comprehensive guide to my implementation of a multithreaded file transfer system using the GetFile protocol. The project consists of four progressive components: **Echo** (basic socket communication), **Transfer** (file transfer fundamentals), **GFLib** (GetFile protocol implementation), and **MTGF** (multithreaded server). Each component builds upon the previous, demonstrating mastery of socket programming, protocol design, and concurrent programming with proper synchronization.

---

## Warmup 1: Echo Client-Server Implementation

### Design Approach and Rationale

**What I Implemented**: A basic client-server pair where the server echoes back any message received from the client (up to 15 bytes).

**Key Design Decisions**:

- **IPv4/IPv6 Dual-Stack Support**: I chose to implement the server using `AF_INET6` with dual-stack capability rather than separate IPv4/IPv6 implementations.
    - **Why**: This approach handles both protocol families with a single socket, reducing complexity
    - **Alternative Considered**: Separate sockets for IPv4 and IPv6, but rejected due to increased resource usage and complexity
- **Client Address Resolution**: Used `AF_UNSPEC` in the client to connect to any available address family.
    - **Why**: Allows the client to connect regardless of server's address family configuration
    - **Implementation**: Iterates through all resolved addresses until successful connection
- Enabling port reuse: `setsockopt(SO_REUSEADDR)`

### Flow of Control - Echo Implementation

```
ECHO SERVER:
1. getaddrinfo(NULL, port, AF_INET6, AI_PASSIVE) → Get server address
2. socket() → Create listening socket with dual-stack support
3. setsockopt(SO_REUSEADDR) → Enable port reuse for development
4. bind() → Bind to address
5. listen() → Enter passive listening mode
6. while (1):
   ├─ accept() → Accept incoming connection
   ├─ recv(fd, buffer, 16) → Read client message
   ├─ send(fd, buffer, received_bytes) → Echo message back
   └─ close(fd) → Close connection and continue
```

```
ECHO CLIENT:
1. getaddrinfo(hostname, port, AF_UNSPEC) → Resolve server address
2. socket() → Create client socket
3. connect() → Attempt connection to server
4. send(socket, message, length) → Send message to server
5. recv(socket, buffer, 16) → Receive echoed response
6. fwrite(buffer, received_bytes, stdout) → Output response
7. close() → Clean up and exit
```

### Testing Strategy - Echo

**Tests I Developed**:

- **Message Integrity Test**: Verified exact byte-for-byte echo with various message lengths (1-15 bytes)
- **IPv4/IPv6 Compatibility**: Tested server with both IPv4 and IPv6 clients
- **Edge Case Testing**: Empty messages, maximum 15-byte messages, and connection drops

**Debugging Process**: Used CLion with Docker toolchain for development environment consistency

---

## Warmup 2: Transfer Client-Server Implementation

### Design Approach and Rationale

**What I Implemented**: A file transfer system where the server sends a complete file to any connecting client, and the client saves the received data to disk.

**Critical Design Decision - Partial I/O Handling**:

- **Problem**: Network operations don't guarantee complete data transfer in single calls
- **My Solution**: Implemented comprehensive loops for both send and receive operations

```
// Example from my transfer client - handling partial writes
while (written < received) {
    ssize_t currWritten = write(fd, buffer + written, received - written);
    if (currWritten == -1) handle_error();
    written += currWritten;
}
```

**Why This Approach**: TCP provides a stream abstraction, not message boundaries. The OS may only accept/deliver partial data based on buffer availability.

**Memory Management Strategy**: Used fixed-size buffers (512 bytes) with streaming I/O rather than loading entire files into memory.

- **Why**: Supports large file transfers without memory constraints

- **Alternative Rejected**: Full file buffering - would fail with large files or limited memory

### Flow of Control - Transfer Implementation

```
TRANSFER SERVER:
1. Initialize socket and bind to port (Same flow as echo)
2. open(filename, O_RDONLY) → Open file for reading
3. while (1):
   ├─ accept() → Accept client connection
   ├─ lseek(fd, 0, SEEK_SET) → Reset file pointer to beginning
   ├─ while (not EOF):
   │   ├─ read(fd, buffer, BUFSIZE) → Read file chunk
   │   ├─ while (sent < bytes_read):
   │   │   └─ send(socket, buffer + sent, remaining) → Send with partial handling
   │   └─ Continue until file complete
   └─ close(connection) → Signal EOF to client

TRANSFER CLIENT:
1. Connect to server
2. open(output_file, O_WRONLY|O_CREAT|O_TRUNC) → Create output file
3. while ((received = recv(socket, buffer, BUFSIZE)) > 0):
   ├─ while (written < received):
   │   └─ write(fd, buffer + written, remaining) → Handle partial writes
4. close(socket) → Server closes connection to signal completion
5. close(output_file) → File transfer complete
```

### Testing Strategy - Transfer

**Tests I Developed**:

- **File Integrity Verification**: Used Docker containers to transfer files and verify MD5 checksums match
- **Large File Testing**: Tested with files up to 100MB to verify partial I/O handling
- **Connection Robustness**: Simulated network interruptions during transfer

**Development Environment**: Used CLion with Docker toolchain for development environment consistency

---

## Part 1: GetFile Protocol Library (GFLib) Implementation

### Design Approach and Rationale

**What I Implemented**: A complete HTTP-like protocol implementation with client library (gfclient) and server library (gfserver) supporting the GetFile protocol.

**Core Design Philosophy - Opaque Pointer Pattern**:

```
struct gfcrequest_t {
    char *server;
    unsigned short portno;
    char *path;
    int sfd;
    void (*writefunc)(void *data, size_t data_len, void *arg);
    void *writearg;
    gfstatus_t status;
    size_t fileLength;
    size_t bytesReceived;
};
```

**Why Opaque Pointers**: Provides clean separation between interface and implementation, preventing client code from accessing internal structures while maintaining flexibility for future modifications.

**Protocol State Management Strategy**:

- **Challenge**: Parse variable-length headers followed by binary content without excessive buffering
- **My Solution**: Two-phase parsing approach
    1. First phase: Keep receiving header until `\r\n\r\n` delimiter to extract status and content-length
    2. Second phase: Stream body content through registered callbacks

**Callback-Based Architecture**:

- **Why**: Allows users to handle data streaming without library making assumptions about storage
- **Benefit**: Memory-efficient for large files - no need to buffer complete file in memory

## Flow of Control - GetFile Protocol

```
GFCLIENT LIBRARY EXECUTION:
APPLICATION CODE:
├─ gfc_create() → Returns opaque gfcrequest_t*
├─ gfc_set_server/port/path() → Configure request parameters
├─ gfc_set_writefunc/writearg() → Register data callback
├─ gfc_perform(request) → Execute request:
│  ├─ socket() and connect() → Establish connection
│  ├─ send("GETFILE GET /path\r\n\r\n") → Send protocol request
│  ├─ Parse response header:
│  │  ├─ recv() until "\r\n\r\n" found → Extract header
│  │  ├─ Parse status (OK/FILE_NOT_FOUND/ERROR/INVALID)
│  │  └─ Extract content-length if status is OK
│  ├─ while (bytes_received < content_length):
│  │  ├─ recv(socket, buffer, chunk_size) → Receive data chunk
│  │  └─ writefunc(buffer, received, writearg) → Callback to application
```

```
│   └─ Return final status to application
└─ gfc_cleanup() → Free all allocated resources

GFSERVER LIBRARY EXECUTION:
MAIN THREAD:
├─ gfs_create() → Initialize server structure
├─ gfs_set_handler() → Register request handler callback
├─ gfs_serve() → Start server:
│   ├─ socket(), bind(), listen() → Setup listening socket
│   ├─ while (1):
│   │   ├─ accept() → Accept incoming connection
│   │   ├─ Parse request header → Extract method and path
│   │   ├─ Reject invalid and error headers
│   │   ├─ handler(&ctx, path, arg) → Call application handler
│   │   └─ Context cleanup and connection close

HANDLER CALLBACK (Provided):
├─ Validate request path and check file existence
├─ if (file not found): gfs_sendheader(ctx, GF_FILE_NOT_FOUND, 0)
├─ else:
│   ├─ gfs_sendheader(ctx, GF_OK, file_size) → Send success header
│   ├─ while (file data remaining):
│   │   ├─ read(fd, buffer, chunk_size) → Read file chunk
│   │   └─ gfs_send(ctx, buffer, bytes_read) → Send to client
└─ return gfh_success
```

### Testing Strategy - GetFile Protocol

**Tests I Developed**:

- **Protocol Compliance Validator**: Custom test scripts sending malformed requests and verifying proper error responses
- **Edge Case Testing**: Invalid schemes, missing headers, incomplete requests
- **Status Code Verification**: Tested all status codes (OK, FILE_NOT_FOUND, ERROR, INVALID) with appropriate scenarios

**Development Tools**: CLion with Docker toolchain provided consistent development environment for protocol testing across different network configurations.

---

## Part 2: Multithreaded GetFile (MTGF) Implementation

### Design Approach and Rationale

**What I Implemented**: A complete multithreaded file server and client using the boss-worker pattern with proper synchronization.

**Critical Design Decision - Boss-Worker Pattern**

**Synchronization Strategy**:

```
typedef struct {
    int active_workers;
    int shutdown;
    steque_t *queue;
    pthread_mutex_t* mutex;
    pthread_cond_t* worker_cond;
    pthread_cond_t* finish_cond;
} worker_fn_args_t;
```

**Why This Design**:

- **Mutex**: Protects shared queue state from race conditions
- **Condition Variables**: Prevent CPU waste when workers are idle - more efficient than busy-waiting
- **Separate Condition Variables**: Different signals for worker wake-up vs. completion notification

**Context Ownership Transfer Challenge**:

- **Problem**: Safely passing `gfcontext_t` ownership from main thread to worker threads
- **My Solution**: Ownership transfer pattern with explicit pointer nullification

```
// In handler - transfer ownership to worker
task->ctx = *ctx;
*ctx = NULL;  // Prevent double-free by nullifying main thread's pointer
```

## Flow of Control - Multithreaded Server

```
MAIN THREAD (Boss):
┌─ pthread_create() × N → Create worker thread pool
├─ steque_init() → Initialize shared work queue
├─ pthread_mutex/cond_init() → Initialize synchronization primitives
├─ while (server running):
│  ├─ accept() → Wait for client connection (blocking)
│  ├─ Create task_item_t:
│  │  ├─ task->ctx = gfcontext (transfer ownership)
│  │  ├─ task->path = requested_path
│  │  └─ task->arg = application_argument
│  ├─ pthread_mutex_lock(mutex)
│  ├─ steque_push(queue, task) → Add work to queue
│  ├─ pthread_cond_signal(worker_cond) → Wake up waiting worker
│  ├─ pthread_mutex_unlock(mutex)
│  └─ Continue to next accept() (connection now owned by worker)
```

```
WORKER THREADS (N workers running worker_fn):
┌─ while (1):  // Infinite worker loop
│   ├─ pthread_mutex_lock(mutex)
│   ├─ while (steque_isempty(queue)):
│   │   └─ pthread_cond_wait(worker_cond, mutex) → Sleep until work available
│   ├─ task = steque_pop(queue) → Get work item
│   ├─ pthread_mutex_unlock(mutex)
│   ├─ content_get(task→path) → Open requested file (fd or -1)
│   ├─ if (fd == -1):
│   │   └─ gfs_sendheader(task→ctx, GF_FILE_NOT_FOUND, 0)
│   ├─ else:
│   │   ├─ fstat(fd, &file_stat) → Get file size
│   │   ├─ gfs_sendheader(task→ctx, GF_OK, file_size)
│   │   ├─ while (offset < file_size):
│   │   │   ├─ pread(fd, buffer, chunk_size, offset) → Read file chunk
│   │   │   ├─ gfs_send(task→ctx, buffer, bytes_read) → Send to client
│   │   │   └─ offset += bytes_read
│   │   └─ close(fd)
│   ├─ gfs_cleanup(task→ctx) → Clean up connection resources
│   └─ free(task) → Free work item and continue loop

CRITICAL SYNCHRONIZATION POINTS:
- Queue access: All steque operations protected by mutex
- Worker coordination: Condition variables prevent busy-waiting
- Context handoff: Ownership transfer prevents double-free errors
- Resource cleanup: Each worker responsible for complete cleanup
```

## Multithreaded Client Implementation

```
CLIENT MAIN THREAD (Boss):
├─ Parse workload file → Generate list of download requests
│─ steque_push(queue, request) → Push all download requests to queue
├─ pthread_create() × N → Create worker thread pool
│─ pthread_mutex_lock(mutex)
│─ pthread_cond_broadcast(worker_cond) → Wake all workers
├─ while (steque_isempty(queue) || active_workers > 0):
│   └─ pthread_cond_wait(finish_cond, mutex) → Wait for completion
├─ pthread_join() → Clean up worker threads
└─ clean up mutex/cond/queue/tids
```

```
CLIENT WORKER THREADS:
├─ while (1):
│   ├─ pthread_mutex_lock(mutex)
│   ├─ while (steque_isempty(queue) && !shutdown):
│   │   └─ pthread_cond_wait(worker_cond, mutex)
│   ├─ if (steque_isempty(queue) && shutdown):
│   │   └─ break → Exit worker loop
│   ├─ request = steque_pop(queue)
│   ├─ active_workers++ → Increment active count
│   ├─ pthread_mutex_unlock(mutex)
│   ├─ gfc_perform(request) → Execute download using gfclient library
│   ├─ pthread_mutex_lock(mutex)
│   ├─ active_workers-- → Decrement active count
│   ├─ if (steque_isempty(queue) && active_workers == 0):
│   │   └─ pthread_cond_signal(finish_cond) → Signal potential completion
│   └─ pthread_mutex_unlock(mutex)
```

### Testing Strategy - Multithreaded Implementation

**Tests I Developed**:

- **Concurrency Stress Testing**: Created custom workloads with 100+ simultaneous requests to identify race conditions
- **Thread Pool Scaling**: Tested with 1, 4, 8, 16, 32 worker threads to find optimal configuration
- **Resource Leak Detection**: Extended operation tests to verify no memory leaks or file descriptor leaks
- **Deadlock Prevention**: Systematic testing of different request orderings and timing scenarios

---

## References - Code and Materials Used

### Code I Used But Did Not Write

- **Steque Library (steque.c/steque.h)**: Used the provided thread-safe queue implementation for work distribution between boss and worker threads
- **Content Library (content.c/content.h)**: Used provided file access abstraction for server-side file operations
- **Workload Library (workload.c/workload.h)**: Used provided workload parsing for client request generation

### Code I Copied Within My Own Project

- **Socket Setup Pattern**: Replicated the getaddrinfo()/socket()/bind() sequence from echo server in transfer server and gfserver implementations

- **Error Handling Patterns**: Consistent error checking and cleanup patterns copied across all components
- **Partial I/O Loops**: Similar send/receive loop patterns used in transfer, gfclient, and gfserver implementations

### Technical References I Consulted

- **POSIX Threads Programming Tutorial** (LLNL): For understanding condition variable usage patterns and avoiding common deadlock scenarios
- **Beej's Guide to Network Programming**: Referenced for socket programming best practices and IPv4/IPv6 dual-stack implementation
- **CS6200 Lecture Slides on Multithreading**: Boss-worker pattern design and synchronization primitive selection
- **Stack Overflow Thread #23207312**: "pthread condition variable example" - helped understand proper condition variable signaling patterns
- **GNU C Library Manual**: File I/O operations documentation, specifically for pread() usage and file descriptor management
- **Course Piazza Posts**: Specific clarifications about GetFile protocol format and threading requirements

### Development Tools I Used

- **CLion IDE with Docker Toolchain**: Primary development environment with integrated debugging support
- **Docker**: Containerized development environment for consistent testing across platforms

**Note**: All external materials were used for concept understanding and best practice guidance. No code was directly copied from external sources - implementations represent my own understanding and application of these concepts.

---

## Project Improvement Suggestions

### 1. Enhanced Local Testing Framework

**Problem I Observed**: Students lack comprehensive testing tools and rely heavily on Gradescope for validation, limiting learning opportunities.

**Actionable Solution**: Create a local testing framework that provides immediate feedback

### 2. Improved Project Instructions

**Problem I Encountered**: The GFLib implementation instructions are unclear about responsibility boundaries between student code and provided callbacks.

**Current Instructions Say**: "The handler callback should not contain any of the socket-level code. Instead, it should use the gfs_sendheader, gfs_send, and potentially the gfs_abort functions provided by the gfserver library."

**What's Confusing**: I don't know whether invalid and wrong headers should be handled by my logic or the provided handler callback.

Actionable Solution: Include a decision flowchart showing exactly when student protocol code vs. application handler should handle different error conditions.