CompSys A3 - Computer Networking

Computersystemer 2025
Department of Computer Science
University of Copenhagen

Jakob Legaard (dmx289), Tommas Cerro(sjm478), Melanie Yu(tjc846)

November 2025

1 Introduction

This report documents our implementation process of the peer-to-peer (P2P) file sharing system where identical peer processes communicate via a custom protocol to register on a network, discover other peers, and transfer files. The system employs a fully distributed architecture where each peer operates concurrently as both a client and a server, enabling bidirectional communication and file exchange.

The protocol implements three core message types: REGISTER (network join), INFORM (peer discovery propagation), and RETRIEVE (file request). Each messsage includes authentication through SHA256 hashed signatures with random salts. The implementation handles concurrent connexions through POSIX threads and protects the shared state with mutex locks.

Implementation

1.1 Peer Registration

the registration process begins when a peer connects to an existing network member. The client thread constructs a 60-byte registration message containing the peer's IP address (16 bytes), port (4 bytes), signature (32 bytes), command code (4 bytes), and body length (4 bytes). The signature is generated by hashing the peer's password, which provides basic authentication.

On the server side the handle_connection function processes incoming registration requests. It validates the message structure, generates a random 16-byte salt, and creates a double-hashed signature by applying SHA256 to the concatenation of the provided signature and salt. this salted signature is stored alongside the peer's network information. The server responds with status code 1 (success) or 2 (re-registration) and includes the complete network list in the response body, formatted as consecutive 68-byte peer entries.

1.2 Network Propagation

Upon a successful registration, the accepting peer broadcasts INFORM messages to all existing network members, except for the newly registered peer. Each INFORM message contains the 68-byte peer entry, including the IP address, port, salted signature, and salt.

The handle_connection function processes INFORM messages by parsing the fixed-size body and checking for duplicate entries before insertions. The implementation iterates through the existing network list, comparing IP addresses and ports. If no match is found, realloc expands the network array and appends the new peer. The mutex is held during this entire operation to maintain consistensy.

1.3 File Retrieval

File requests are initiated by the client thread, which prompts the user for a filename and selects a random peer from the network list. The RETRIEVE message includes the filename in the body, with the length field specifying the string size.

the server validates that the requesting peer exists in the network before attempting file access. The implementation employs compsys_helper_readn and compsys_helper_writen for robust I/O operations that handle partial reads/writes and EINTR signals. SHA256 hashes are computed for both individual blocks and the entire file. The total hash is calculated by hashing the concatenation of all block hashes, providing hierarchical integrity verification. Retrieved files are written to the peer's directory using the same filename.

1.4 Message Fragmentation

Files exceeding MAX_MESSAGE_SIZE (minus header overhead) are automatically fragmented. The implementation calculates the number of required blocks by dividing the file size by the maximum body size per message. Each fragment includes a reply header specifying this_block (0-indexed block number), block_count (total fragments), block_hash (SHA256 of the current block), and total_hash (SHA256 of all block hashes concatenated).

The client must buffer received blocks and potentially reorder them if they arrive out of sequence, although the current implementation assumes in-order delivery. Each block's hash is verified upon receipt before reassembly. Once all blocks are received, the total hash is recalculated and compared against the received value to ensure complete file integrity.

1.5 Concurrency and Thread Safety

The system employs a dual-thread architecture: one client thread for user interaction and outbound requests, and a server thread that spawns detached handler threads for each incoming connexion. The global network_mutex protects all access to the network array and peer_count.

1.5.1 shared Data structures

these data structures are accessed by multiple threads:

- `network[]` array: dynamicaly allocated array storing all the peer information
- `peer_count`: an integer tracking the number of registered peers in the network
- `my_address`: Contains the local peer's IP and port

1.5.2 Race condition prevention

Modifications to shared data structures that occur within mutex-protected critical sections:

- Registration requests lock `network_mutex` before checking for duplicate peers and before inserting new entries.
- The client thread locks `network_mutex` when selecting a random peer for file requests
- File I/O operations executes outside critical sections to minimize lock hold time and prevent blocking other threads during slow disk operations

1.5.3 Deadlock Prevention

The system uses a single global mutex (network_mutex), eliminating lock ordering issues. However, some error paths in handle_connection may return without releasing the mutex, potentially causing deadlock (see the limitations section).

Testing

Our testings are conducted manually via local host on MacOS machine. Beyond the provided .txt files, we also tested with custom edge cases like empty files and files with special characters only. In general, the testings included invalid requests, non-existent files and network configuration with 2-4 concurrent peers, where no automation was used. The different kind of tests, the description of what was tested, data/input and resulted behavior can be found on "Tabular 1: Testing".

Tests	Description	Data/Input	Machine	Resulted Behaviour
Large file	Multi-block file transfer	hamlet.txt	Local	Automatic retry: Retrieve failed with status 5 -> status 1: length 196746
Small file	Single block file transfer	Tiny.txt	Local	Immediate response: status 1: length: 30
Empty file	0 byte file handling	Empty.txt	Local	Immediate response: status 1: length 0
Special file	Only special character file handling	Special_char.txt	Local	Immediate response: Status 1: length 21
Invalid file	Non existent file handling	Nonexistent.txt	Local	Retrieve failed with status 5
Multiple peers	Basic connection by letting `cpeer2` connect with `cpeer1` and request through `cpeer2`	3 peers - 1 python, 2 C Used: tiny.txt	Local	Immediate response: Status 1, length 30. However followed by retrieve failed status 4 and 5
Stress testing with multiple peers	Connection 2 cpeers with first python and then connect with second	4 peers - 2 python, 2 C Used: tiny.txt and hamlet.txt	Local	Some transfers successes while other had status 5,6
Duplicate peer	Connecting same C peer twice to the same python peer	3 peers - 1 python, 2 C	Local	Showed status 2 error
Wrong IP/Port	Connect C peer to invalid port using port 99999	192.0.1.2.99999	Local	Recognized invalid port and failed connection

Table 1: Testing

Limitations and potential problems

The primary limitation is incomplete error handling in critical sections. Several code paths within handle_connection return early without releasing network_mutex, potentially causing permanent deadlock. For example, memory allocation failures when expanding the network list may return without unlocking. This should be addressed by adding pthread_mutex_unlock calls before all return statements, or by using a cleanup handler with pthread_cleanup_push.

The fixed MAX_MESSAGE_SIZE of 8196 bytes could create a possible performance bottleneck for larger files, as they would require excessive round-trips. A possible better approach for us to implement/improve could be adding adaptive block sizing based on file size and network conditions, which could include compression support and transfer resumption capabilities

Conclusion

The implemented peer-to-peer file sharing system successfully demonstrates core implemented networking concepts, including concurrent client-server operation, message fragmentation, and authentication. The concurrency model using POSIX threads and mutexes provides basic thread safety for the shared network list; however, though improvements are needed in error handling and resource bounds. The protocol design achieves enabling distributed file sharing.

The testing phase revealed both strengths and weaknesses in the implementation. While the system successfully handled edge cases like empty files and special characters, it exhibited reliability issues under stress testing with multiple concurrent peers, with some transfers failing with status codes 4, 5, and 6. The automatic retry mechanism for large files (as seen with hamlet.txt) demonstrates robustness, though the underlying cause of initial failures warrants further investigation.

Theoretical questions

Reserving connections

On the networking layer, the internet uses the connection less internet protocol. It makes uses of data being divided into packets which each are sent individually through the network between the sender and receiver. This approach is quiet efficient as it avoids circuit switching where a path must be established to maintain a channel. Instead, most IP rely on TCP because it establish a virtual connection between the devices using the 3-way-handshake and ensures all packets are delivered correctly in sequence. Therefore, the internet remains the efficiency through the connection less protocols while using connection-oriented TCP, making it both connectionless and connection-dependent.

Reliable communication

The protocol built on UDP must implement the same cores that TCP provides to ensure reliability. This would include guarantee for the received data being exactly the same that was sent through ACK and retransmission for lost packets. Since UDP does not take into account of packet orders, the protocol must also include sequential numbers to ensure the packets are delivered in the correct order, as TCP already does. It shall also have flow control to prevent buffer overflow and data loss, as well as cognition control to prevent overwhelming the network.

DNS

It shall have the top-level domain .sm for Scotmark, as according to the ISO3166-1 standard. They shall first determine the root DNS servers which acts like a masterbase over all second level domains. Next, the root serves will direct queries to the appropritae TLD nameserves for organizing specific sectors. This needs TLD servers which manages the websites endings to know where to find the domains, so for instance the goverments website would be .gov.sm and universities would be .edu.sm. Lastly, they would have an authoritative DNS server which holds the IP addresse for all hosts in the domain and additionally holds their record types.

Hashing

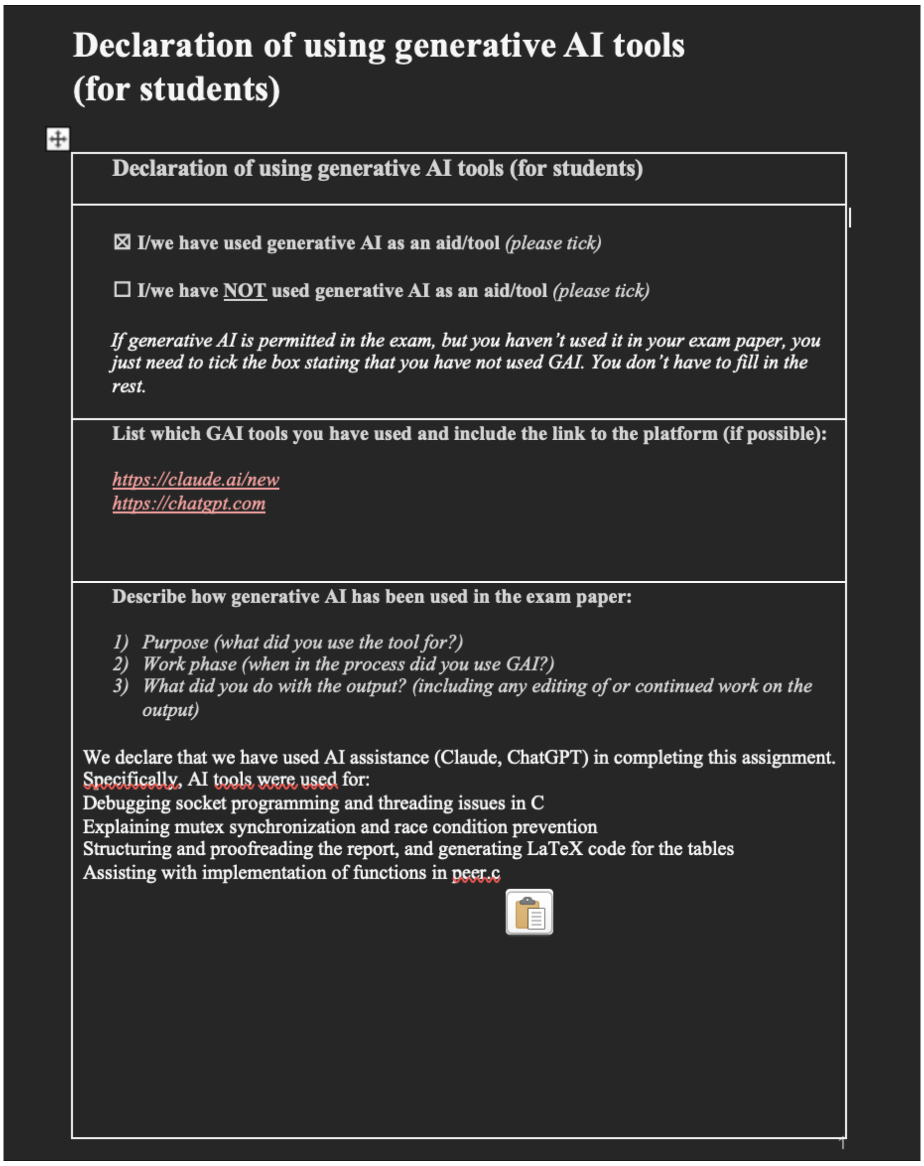
It is not advertised to use only hash function for each application because it would be complicated to make them fully secured. Hash functions are determinitis, which means the same input always give the same output. They are collision resistant, making it hard to find two different input with the same hash. Meanwhile rainbow tables are defated best by not only using hash functions but also adding unique salt to each input. In this way, it will ensure unique hashes and prevent attacks. They would no matter what be vulnerable to attacks such as by rainbow tables, which would become easy for the attacker to hack into shared passwords. Therefore, using properly designed hash functions that are combined with salting for the unique user is the most effective and secured approach.

Deadlock

The Santa Claus Problem involves Santa, 9 reindeer, 12 elves, and a stable coordinating through three mutexes: reindeer_mutex, busy_mutex, and elf_mutex. To determine if deadlock can occur, we analyze the dependencies between threads. A deadlock requires circular wait—where thread A waits for B, B waits for C, and C waits for A. Santa acquires busy_mutex with either reindeer_mutex or elf_mutex (for elves). Stable uses only reindeer_mutex. either Santa nor Stable ever wait for other threads—they only process messages and release locks. Reindeer never acquire locks, while Elves briefly hold elf_mutex to send messages but release it before waiting for responses.

Santa processes three elves while holding elf_mutex for 20 seconds. If Elf4 attempts to send a message during this time, it waits for Santa. However, Santa does not wait for Elf4, so the wait chain terminates without cycling back. When multiple reindeer register simultaneously, they wait for Stable
deadlock requires: mutual exclusion, hold-and-wait, no preemption, and circular wait. While the first three hold, circular wait is violated. Therefore, the system is deadlock-free.

2 AI-Declaration



Declaration of using generative AI tools (for students)

Declaration of using generative AI tools (for students)
☒ **I/we have used generative AI as an aid/tool** *(please tick)* ☐ **I/we have <u>NOT</u> used generative AI as an aid/tool** *(please tick)* *If generative AI is permitted in the exam, but you haven't used it in your exam paper, you just need to tick the box stating that you have not used GAI. You don't have to fill in the rest.*
List which GAI tools you have used and include the link to the platform (if possible): *https://claude.ai/new* *https://chatgpt.com*
Describe how generative AI has been used in the exam paper: *1) Purpose (what did you use the tool for?)* *2) Work phase (when in the process did you use GAI?)* *3) What did you do with the output? (including any editing of or continued work on the output)* We declare that we have used AI assistance (Claude, ChatGPT) in completing this assignment. Specifically, AI tools were used for: Debugging socket programming and threading issues in C Explaining mutex synchronization and race condition prevention Structuring and proofreading the report, and generating LaTeX code for the tables Assisting with implementation of functions in peer.c

Figure 1: Enter Caption