|  |  |
| --- | --- |
| **UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI Major : Cyber Security Course: Network Programming** |  |

**Peer to Peer Files Sharing System**

|  |  |
| --- | --- |
| ***Team Leader***: | Trương Đức Hiệp – 22BA13130 |
| ***Contributing Members*:** | Dương Minh Tiến - 22BA13297  Nguyễn Trường Giang – 22BA13111  Đặng Trung Nguyên - 22BA13239  Đặng Thu Huyền - 22BA13165  Bùi Mạnh Duy – 22BA13097  Đoàn Đình Khải - 22BA13167 |

**I. Project Overview**

**1. Introduction**

In today's digital age, sharing data between devices has become a necessity. Traditional file sharing systems based on the client-server model often suffer from scalability, performance, and single point of failure issues. This project aims to overcome these limitations by developing a fully decentralized peer-to-peer (P2P) file sharing system that allows users to both provide and consume files without the need for a central server.

This P2P system is inspired by successful file sharing protocols such as BitTorrent, Gnutella, and Napster, but is redesigned from the ground up to be easy to deploy while optimizing performance in a local network environment. Each node in the system (peer) operates independently and contributes its resources to the network, increasing scalability and reliability as the network grows.

The strengths of the system include the ability to split files into multiple parts and download them in parallel from multiple sources, optimize transfer speeds, and balance network load. Furthermore, the use of cryptographic hashing algorithms such as SHA-256 ensures the integrity of transmitted data.

**2. Project Objectives**

This project is developed with the following main objectives in mind:

* Completely Decentralized: Design a system without a centralized point, eliminating a single point of failure.
* High Efficiency: Optimize network bandwidth usage through parallel download and sharding mechanisms.
* Scalability: The system can grow with the number of users and data without performance issues.
* Reliability: Ensure data integrity with file verification mechanisms.
* Ease of Use: Provide a simple interface for end users to interact with the system.
* Self-Organizing: Partners can automatically discover and connect to each other without manual configuration.

**3. Key Features**

Our P2P system provides the following key features:

* Peer Auto-Discovery Mechanism: Peers automatically discover each other via UDP broadcasts, allowing for dynamic network formation without manual configuration. The system uses a simple communication protocol to announce and update the presence of peers in the network.
* Distributed File Indexing: Each peer maintains information about locally available files and can query this information from other peers in the network. Instead of a central index server, the system distributes this function to all peers, increasing fault tolerance and reducing the load on each node.
* File Segmentation: Files are divided into smaller pieces (default 256KB) for efficient transmission. This method allows partial file downloads, increases recovery from transmission interruptions, and supports parallel downloads.
* Parallel download from multiple peers: The system optimizes the download speed by downloading different parts of the file from multiple peers at the same time. This strategy not only increases the download speed but also distributes the load across the entire network, avoiding overloading a single peer.
* Bandwidth management: Limits the download speed to avoid network overload and ensures reasonable use of network resources. Each connection is limited to 100KB/s to avoid occupying all the user's bandwidth.
* Advanced search function: Users can search for files across the entire network by providing keywords. The search results include detailed information about the file such as name, size, number of available sources, and hash value.
* File integrity verification: Uses SHA-256 hashing to ensure the integrity of the downloaded file, avoiding fraud or data corruption during transmission. Each file is verified after the download is complete to ensure that the content has not been altered.

**4. Overall Architecture**

The system is designed in a pure peer-to-peer model without a central server, where each partner is both a client and a server. Each component in the system has a specific role in the network:

**4.1. Main Components**

* Peer Node: The basic unit of the system, each node runs the same software and performs all functions.
* UDP Manager Thread: Responsible for broadcasting and listening for peer discovery messages.
* TCP Server Thread: Handles incoming connections from other peers to search for and download files.
* Local File Indexing System: Manages information about shared files from the "shared" folder.
* Search Result Cache: Temporarily stores information about remote files from other peers.
* Download Manager: Coordinates parallel downloads from multiple sources and manages file integrity.

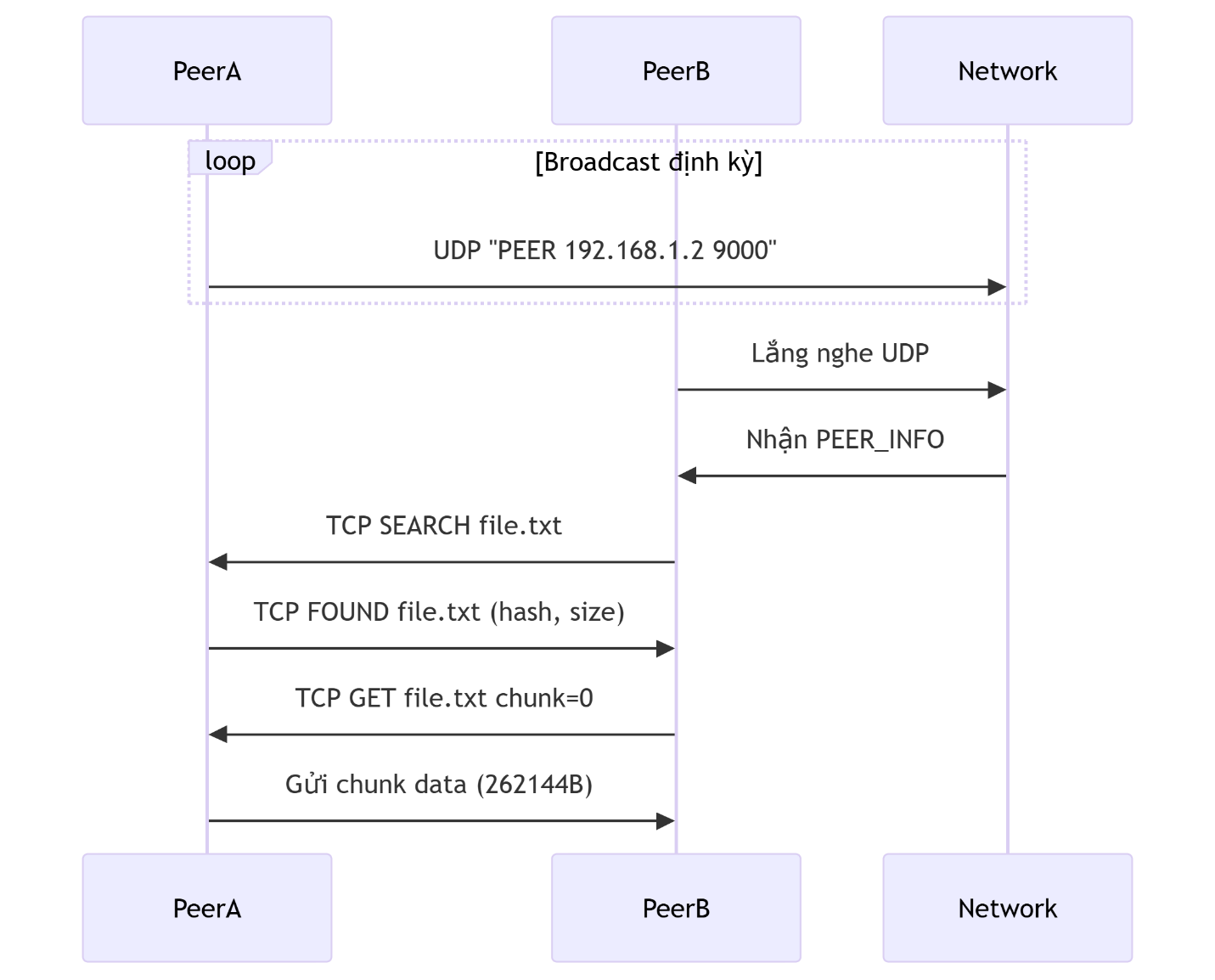
**4.2. Roles in the network**

Each peer in the system can take on one or more of the following roles, depending on the situation:

* File Provider: Shares local files with the network and responds to download requests.
* File Consumer: Searches for and downloads files from other peers in the network.
* Index Node: Maintains and provides information about files available in the system.
* Router Node: Supports peer discovery and connectivity, especially important in large networks.

**4.3. Main Data Flows**

* Peer Discovery: Uses UDP broadcast to announce presence and collect information about other partners.
* File Search: Sends queries directly to known partners to find a specific file, then aggregates the results.
* File Download: Splits a file into multiple parts, selects the appropriate source for each part, and downloads them in parallel.
* Verify and Complete: Checks the integrity of the downloaded file and notifies the user of the results.



**II. Protocol Definition and Communication Pattern**

**1. Peer Discovery Protocol**

**1.1. Protocol Definition**

Peer discovery is a fundamental process in any P2P system, allowing new nodes to join the network and maintain connections with existing networks. Our system uses a simple but effective UDP broadcast method to discover peers in the local network:

* Default Port: 9000 (UDP)
* Message Format: `PEER <IP Address> <TCP Port>`
* Broadcast Frequency: every 3 seconds
* Refresh Mechanism: When a broadcast message from a known peer is detected, the peer's "last seen" time is updated
* Expiration Time: Peers that have been inactive for a certain period of time are removed from the list

This method was chosen for its simplicity and effectiveness in a local network environment. However, this method also has some limitations, especially when extending beyond the LAN, which will be discussed in the "Limitations and Improvements" section.

**1.2. Detailed Message Flows:**

a. Initialization: When a peer starts, it creates two separate threads to handle UDP broadcasting and listening.

b, Broadcast: The broadcast thread periodically (every 3 seconds) sends a UDP message to the broadcast address 255.255.255.255 on port 9000 with the content: `PEER <current\_peer\_IP> <TCP\_port\_listening>`.

c. Listening: The listening thread continuously waits for UDP messages on port 9000:

* When a message is received, it will parse it to extract the IP address and port of the peer sending the message.
* If the message is from an existing peer, the message will be ignored.
* If the peer does not already exist in the list, a new peer will be added with the "last seen" time being the current time.
* If the partner already exists in the list, only update the "last seen" time.

d. Maintenance: A separate thread performs periodic cleanup of the partner list:

* Remove partners that have been inactive for more than a threshold amount of time (although this functionality is not fully implemented in the current code).
* Ensure the partner list does not exceed the maximum size limit (MAX\_PEERS = 32).

e. Data structure: Peer information is stored in an array of PeerInfo structures, each entry containing:

typedef struct {

char ip[INET\_ADDRSTRLEN];

int port;

time\_t last\_seen;

} PeerInfo;

f. Synchronization: Access to the peer list is protected by a mutex (peer\_mutex) to ensure consistency when multiple threads access it at the same time.

**1.3. Detailed Algorithm:**

Here is a detailed description of how the main flows involved in the peer discovery process work:

**UDP Broadcast Flow:**

void\* udp\_broadcast\_thread(void\* arg) {

int sock = socket(AF\_INET, SOCK\_DGRAM, 0);

int broadcast = 1;

setsockopt(sock, SOL\_SOCKET, SO\_BROADCAST, &broadcast, sizeof(broadcast));

struct sockaddr\_in baddr = {0};

baddr.sin\_family = AF\_INET;

baddr.sin\_port = htons(UDP\_PORT);

baddr.sin\_addr.s\_addr = inet\_addr("255.255.255.255");

char msg[64];

while (1) {

snprintf(msg, sizeof(msg), "PEER %s %d", my\_ip, listen\_port);

sendto(sock, msg, strlen(msg), 0, (struct sockaddr\*)&baddr, sizeof(baddr));

sleep(BROADCAST\_INTERVAL);

}

return NULL;

}

**UDP listening thread:**

void\* udp\_listen\_thread(void\* arg) {

int sock = socket(AF\_INET, SOCK\_DGRAM, 0);

int opt = 1;

setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

#ifdef SO\_REUSEPORT

setsockopt(sock, SOL\_SOCKET, SO\_REUSEPORT, &opt, sizeof(opt));

#endif

struct sockaddr\_in addr = {0};

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(UDP\_PORT);

addr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(sock, (struct sockaddr\*)&addr, sizeof(addr)) < 0) {

perror("bind UDP failed");

exit(1);

}

char buf[128];

while (1) {

struct sockaddr\_in src;

socklen\_t slen = sizeof(src);

int n = recvfrom(sock, buf, sizeof(buf)-1, 0, (struct sockaddr\*)&src, &slen);

if (n > 0) {

buf[n] = 0;

char ip[INET\_ADDRSTRLEN];

int port;

if (sscanf(buf, "PEER %15s %d", ip, &port) == 2) {

if (strcmp(ip, my\_ip) == 0 && port == listen\_port) continue;

pthread\_mutex\_lock(&peer\_mutex);

int found = 0;

for (int i = 0; i < peer\_count; i++) {

if (strcmp(peers[i].ip, ip) == 0 && peers[i].port == port) {

peers[i].last\_seen = time(NULL);

found = 1;

break;

}

}

if (!found && peer\_count < MAX\_PEERS) {

strcpy(peers[peer\_count].ip, ip);

peers[peer\_count].port = port;

peers[peer\_count].last\_seen = time(NULL);

peer\_count++;

printf("[DISCOVERY] New peer: %s:%d\n", ip, port);

}

pthread\_mutex\_unlock(&peer\_mutex);

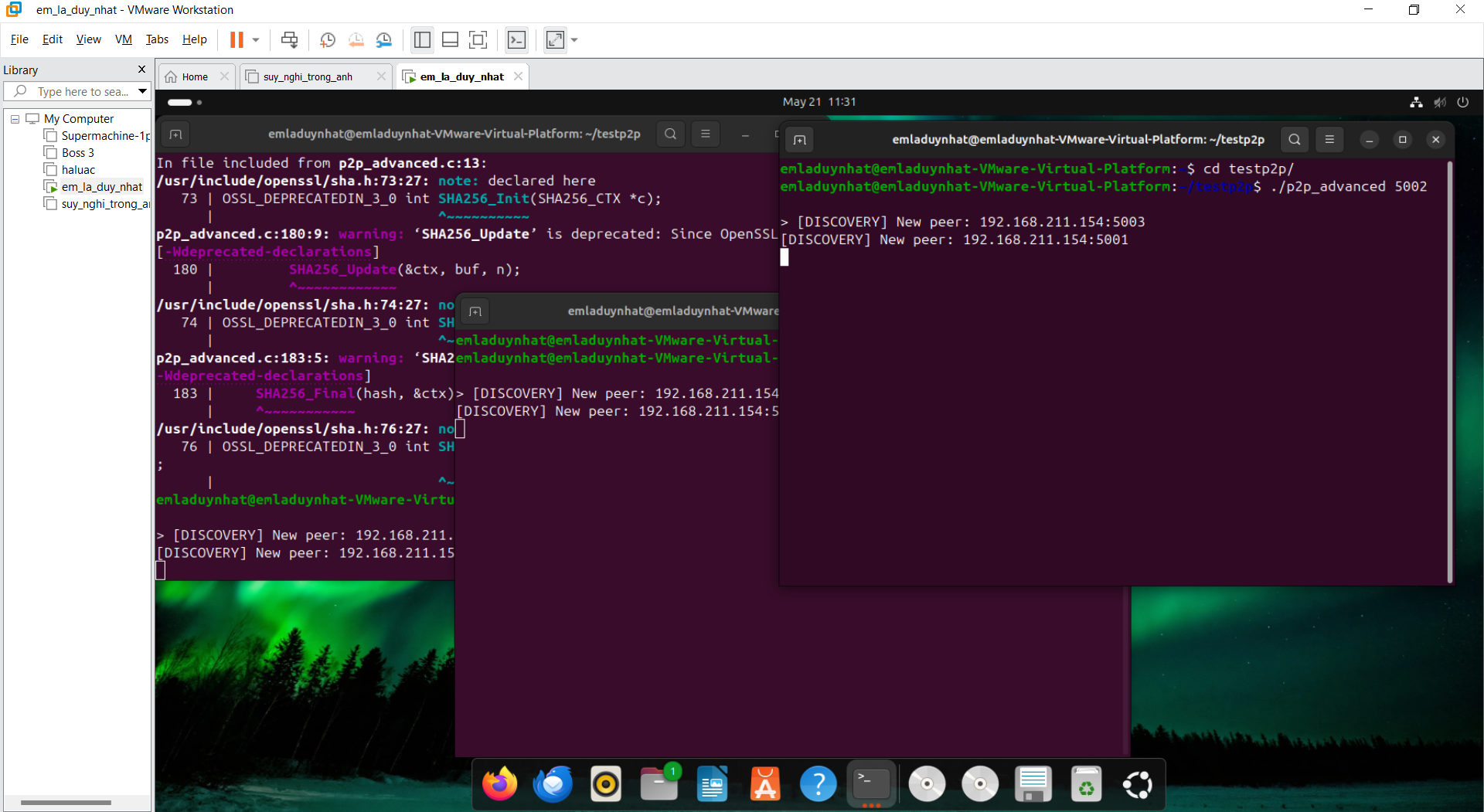
}

}

}

return NULL;

}



**2. File Query Protocol**

**2.1. Protocol Definition**

To query information about files available in the network, the system uses a TCP-based protocol to ensure the reliability of the data exchanged:

* TCP port: Defined by each partner in the UDP broadcast message
* Search query format: SEARCH [keyword]
* Search response format: FOUND <file\_name> <size> <block\_number> <hash\_sha256> <tcp\_port>

This query protocol is designed to optimize both bandwidth and accuracy. Each response contains detailed information about the file including the hash value to verify integrity, the exact size, and the number of blocks to plan the download.

**2.2. Detailed message flow:**

a. Start searching:

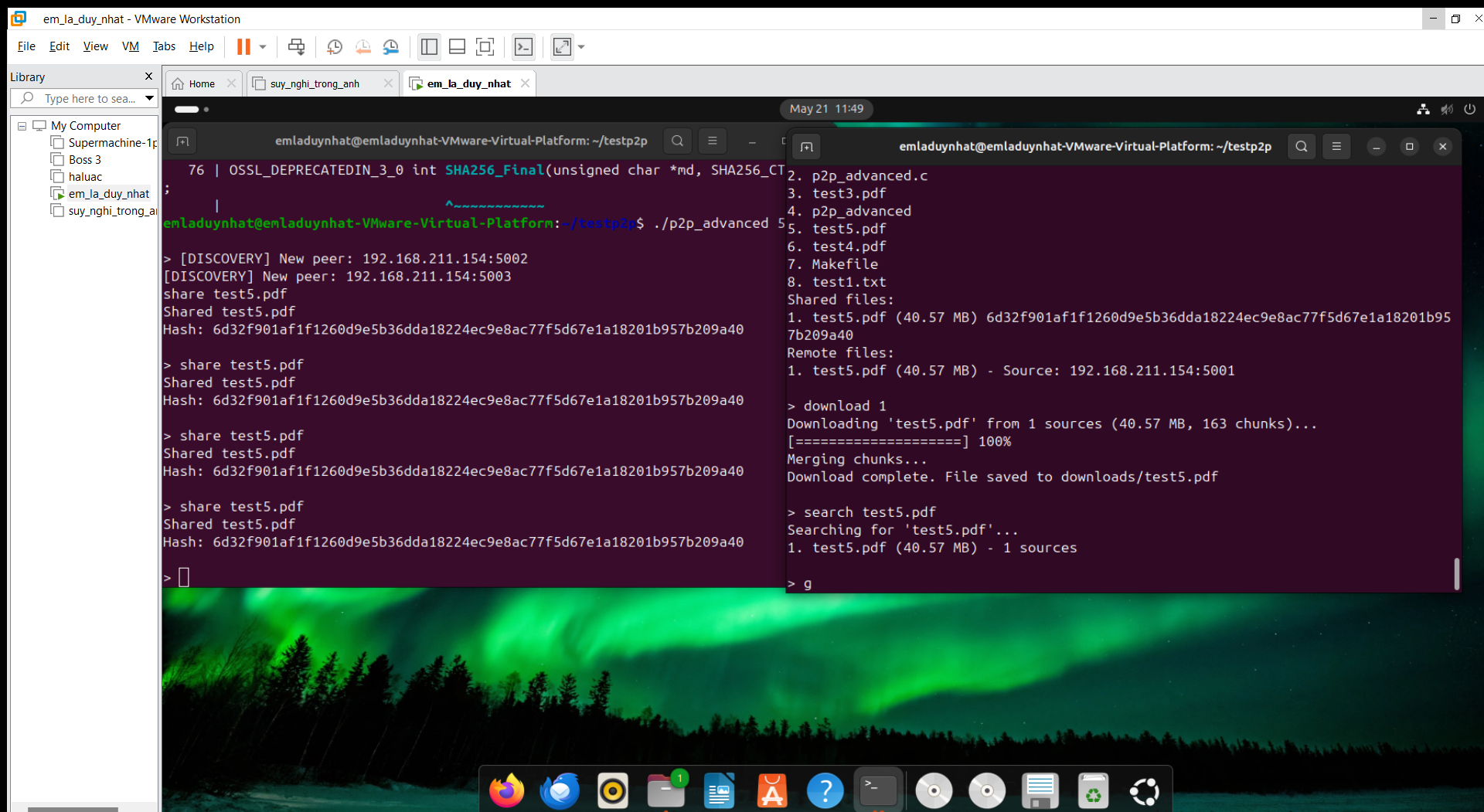
* The user enters the search command <keyword> or uses the list command to view all files.
* The system retrieves a list of known partners and starts the search process.

b. Connect to each partner:

* For each partner in the list, the system establishes a TCP connection to the specified port.
* Set a short connection timeout (2 seconds) to avoid waiting for unresponsive partners.

c. Send a query:

* Send a SEARCH <keyword> message over the TCP connection.



* If the keyword is empty (in the case of the list command), all files are returned.

d. Process the response:

* The partner receives the search query in the handle\_client() function.
* The partner locks the file mutex to ensure consistency when reading the file list.
* The partner scans the local file list and compares it with the search keyword (or returns all if the keyword is empty).
* The partner sends a FOUND message with detailed information about the file.

e. Aggregate results:

* The partner sends the query, receives the response, and processes each response individually.
* The system extracts the file information from the response and checks the hash value to avoid duplicates.
* If the file already exists in the result list (based on the hash), the new source information is added.
* If the file does not exist, a new record is created with the first source and details.

f. Display results:

* After querying all partners, the results are displayed to the user.
* Each result includes the file ID, name, size, and number of available sources.

**2.3. Managing search results**

The system stores search results in an array of RemoteFile structures, each containing:

typedef struct {

char fname[256];

unsigned char hash[SHA256\_DIGEST\_LENGTH];

size\_t size;

int chunk\_idx;

char ip[INET\_ADDRSTRLEN];

int port;

char \*chunkbuf;

int \*chunk\_lens;

int \*done;

int result;

} ChunkTask;

**2.4. Pseudocode of the search process:**

void fetch\_remote\_files(const char\* keyword, int silent) {

remote\_file\_count = 0;

pthread\_mutex\_lock(&peer\_mutex);

int n\_peers = peer\_count;

PeerInfo peerlist[MAX\_PEERS];

memcpy(peerlist, peers, sizeof(peerlist));

pthread\_mutex\_unlock(&peer\_mutex);

int found = 0;

for (int i = 0; i < n\_peers; i++) {

int sock = socket(AF\_INET, SOCK\_STREAM, 0);

struct timeval tv;

tv.tv\_sec = 2;

tv.tv\_usec = 0;

setsockopt(sock, SOL\_SOCKET, SO\_RCVTIMEO, (const char\*)&tv, sizeof(tv));

setsockopt(sock, SOL\_SOCKET, SO\_SNDTIMEO, (const char\*)&tv, sizeof(tv));

struct sockaddr\_in addr = {0};

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(peerlist[i].port);

inet\_pton(AF\_INET, peerlist[i].ip, &addr.sin\_addr);

if (connect(sock, (struct sockaddr\*)&addr, sizeof(addr)) == 0) {

char req[256];

snprintf(req, sizeof(req), "SEARCH %s", keyword);

send(sock, req, strlen(req), 0);

char buf[1024];

int n;

while ((n = recv(sock, buf, sizeof(buf)-1, 0)) > 0) {

buf[n] = 0;

char fname[256], hashstr[65];

size\_t size; int chunks, port;

if (sscanf(buf, "FOUND %255s %zu %d %64s %d", fname, &size, &chunks, hashstr, &port) == 5) {

unsigned char hash[SHA256\_DIGEST\_LENGTH];

for (int j = 0; j < SHA256\_DIGEST\_LENGTH; j++)

sscanf(hashstr + 2\*j, "%2hhx", &hash[j]);

int idx = -1;

for (int k = 0; k < remote\_file\_count; k++) {

if (memcmp(remote\_files[k].hash, hash, SHA256\_DIGEST\_LENGTH) == 0) {

idx = k;

break;

}

}

if (idx == -1 && remote\_file\_count < MAX\_REMOTE\_FILES) {

idx = remote\_file\_count++;

strncpy(remote\_files[idx].name, fname, 255);

remote\_files[idx].size = size;

remote\_files[idx].num\_chunks = chunks;

memcpy(remote\_files[idx].hash, hash, SHA256\_DIGEST\_LENGTH);

remote\_files[idx].num\_sources = 0;

strcpy(remote\_files[idx].sources[0].ip, peerlist[i].ip);

remote\_files[idx].sources[0].port = port;

remote\_files[idx].sources[0].last\_seen = time(NULL);

remote\_files[idx].num\_sources = 1;

if (!silent) {

printf("%d. %s (%.2f MB) - 1 sources\n", idx+1, fname, size/1024.0/1024.0);

}

} else if (idx != -1) {

int already = 0;

for (int s = 0; s < remote\_files[idx].num\_sources; s++) {

if (strcmp(remote\_files[idx].sources[s].ip, peerlist[i].ip) == 0 &&

remote\_files[idx].sources[s].port == port) {

already = 1;

break;

}

}

if (!already && remote\_files[idx].num\_sources < MAX\_SOURCES) {

strcpy(remote\_files[idx].sources[remote\_files[idx].num\_sources].ip,

peerlist[i].ip);

remote\_files[idx].sources[remote\_files[idx].num\_sources].port = port;

remote\_files[idx].sources[remote\_files[idx].num\_sources].last\_seen = time(NULL);

remote\_files[idx].num\_sources++;

}

}

found++;

}

}

}

close(sock);

}

if (!silent && found == 0) printf("No results found.\n");

}

**3. File Transfer Protocol**

**3.1. Protocol Definition**

To download a file, a partner uses a TCP connection to request specific blocks:

* TCP port: Same as the port used for the search query
* Block request format: GET <file\_name> <block\_index>
* Response: Binary data of the requested block
* Default block size: 256 KB (262,144 bytes)

**3.2. Message flow:**

* Partner A connects to Partner B's TCP port.
* Partner A sends a "GET" request for a specific file and block.
* Partner B reads the block from the local file and sends the binary data directly.
* Partner A stores the block in the appropriate location in the destination file.

**4. File indexing mechanism**

**4.1. Local File Index**

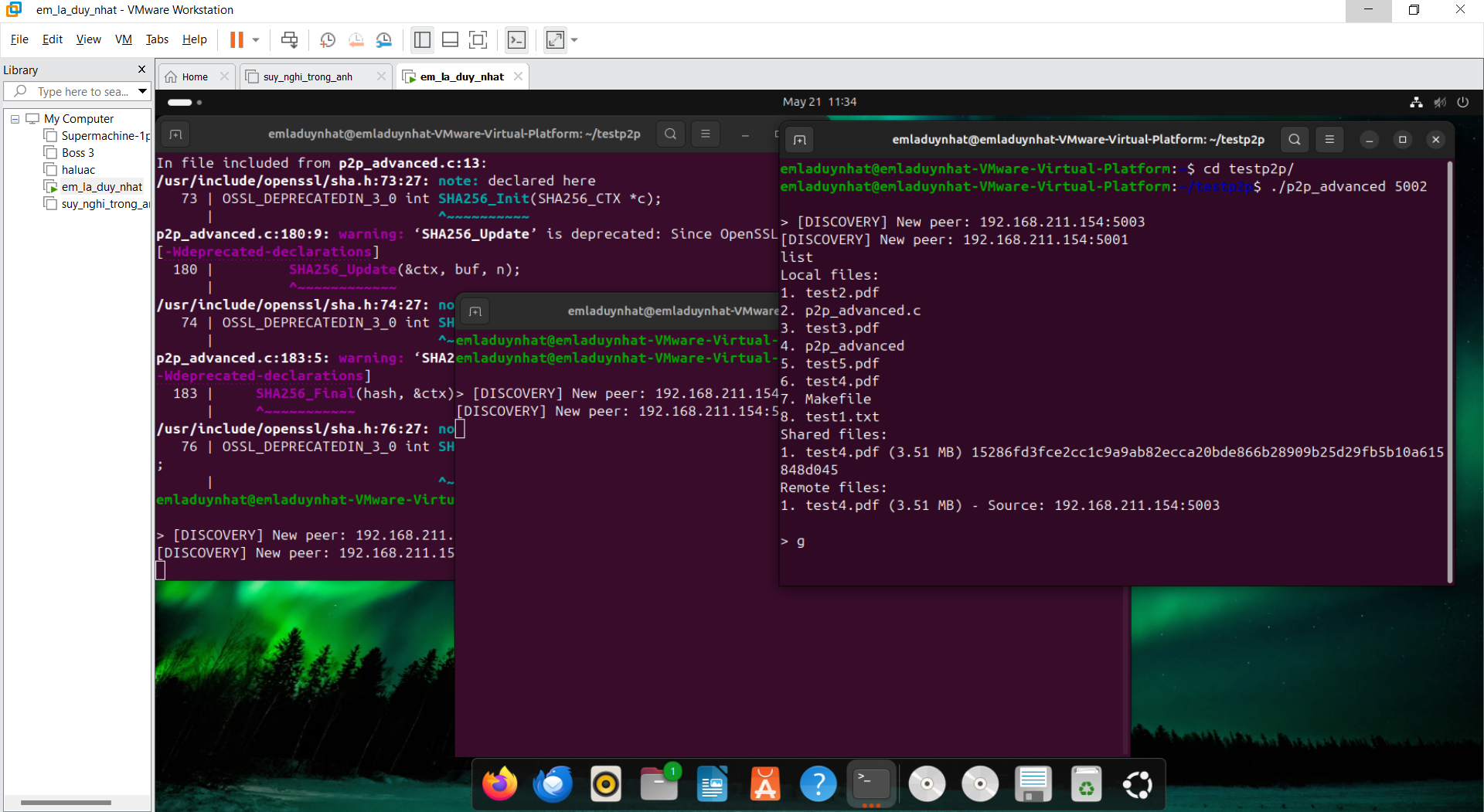
Each partner maintains an index of locally available files in a "shared" directory. The information stored for each file includes:

* File name
* File size (bytes)
* Block count
* SHA-256 hash
* Source information (IP address and port)

**4.2. Remote file index**

When searching, a partner collects information about the file from other partners and creates a temporary remote file index, which includes:

* File name
* File size
* Block count
* SHA-256 hash
* A list of sources (partners) with the file available, each with an IP address, port, and timestamp



**III. File Download and Processing Mechanism**

**1. Splitting and Parallel Download Mechanism**

Files are split into fixed-size blocks (default 256 KB) to support segmented downloads. The download process works as follows:

* File Initialization: Create an empty file with the exact size of the file to be downloaded.
* Source Allocation: For each block, randomly select a partner from the available source list.
* Parallel Download: Create multiple threads to download blocks from different partners at the same time.
* Retry on Failed Download: If a block fails to download, retry from another partner, up to 3 times.
* Block Writing: Each block is written directly to the exact location in the destination file.

**2. File Integrity Verification**

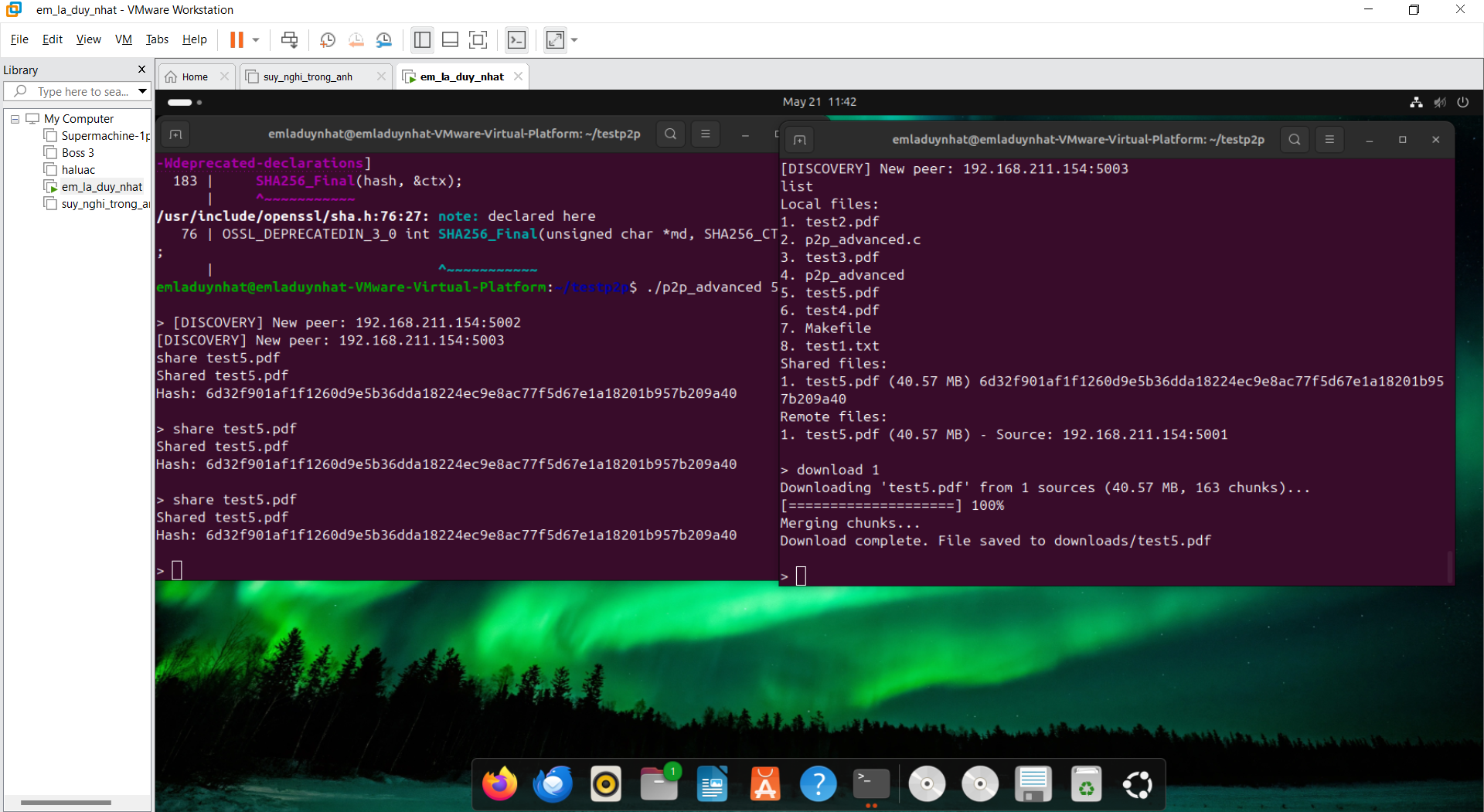
To ensure the integrity of downloaded files, the system uses the SHA-256 hash function:

* Before sharing, a SHA-256 hash is calculated for the entire file.
* This hash is transmitted when searching for results.
* After downloading all blocks, a SHA-256 hash is calculated for the downloaded file.
* If the hash matches the original hash, the file is confirmed to be intact.

**3. Bandwidth Management**

The system implements basic rate limiting to avoid network overload:

* Default rate limit: 100 KB/s per connection
* No complex congestion prevention mechanism, but may be added in future versions



**IV. User Interface**

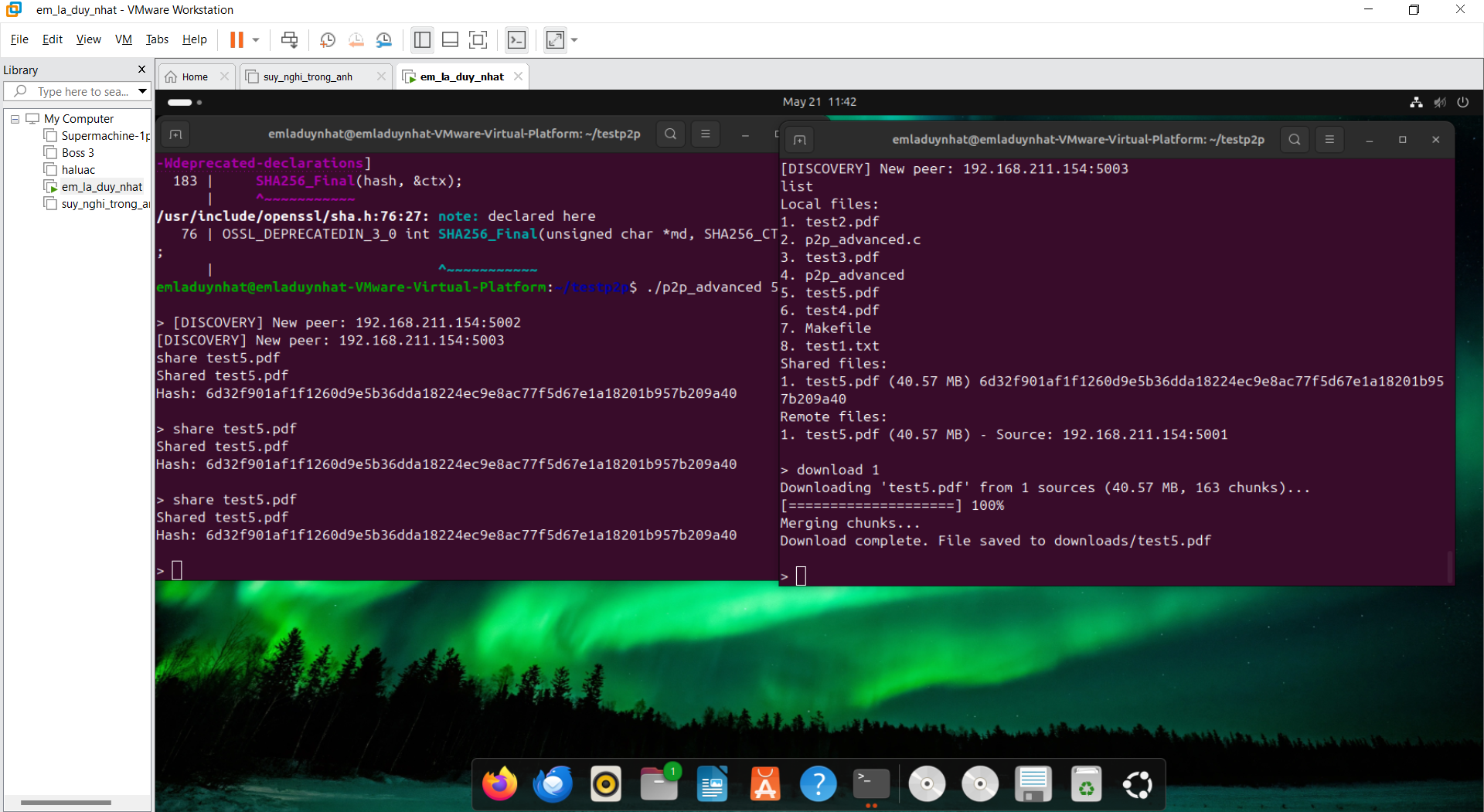
**1. Command Line Interface**

The system provides a command line interface (CLI) with the following commands:

* share <file\_path>: Share a local file with the network.
* search <keyword>: Search for a file on the network using a specific keyword.
* download <id\_file>: Download a file from the network based on the file ID.
* list: List all available local and remote files.

**2. Download Progress Report**

The system displays the download progress using an ASCII progress bar:



**V. Known Limitations and Future Improvements**

**1. Current Limitations**

* Limited peer discovery: UDP broadcasting only works within the local network and does not cross routers.
* No DHT mechanism: Lacks distributed hash tables to track files across large networks.
* Weak security: No encryption or authentication in the protocol, making it vulnerable to attacks.
* No block integrity checking: Only checks the integrity of the entire file, not individual blocks.
* Simple bandwidth management: Lacks complex congestion mitigation algorithms.
* No peer ranking: No mechanism to prioritize peers based on performance or reliability.
* No session persistence: Peer information and remote indexes are not stored between runs.
* Limited NAT support: No mechanism to handle connections through NAT or firewalls.

**2. Future improvements**

* DHT implementation: Add distributed hash table to support larger networks and better peer discovery.
* Security enhancements:
* Peer-to-peer encryption
* File authentication with digital signatures
* Block-level integrity verification
* Network scaling:
* NAT penetration support with UDP hole punching
* Bootstrap partners support initial connections
* Advanced scheduling algorithms:
* Select the least dense block first
* Priority-based download model
* Peer ranking mechanism:
* Track partner performance and reliability
* Prioritize high-performing partners
* Advanced user interface:
* Graphical user interface (GUI)
* Real-time bandwidth statistics graph
* Management features:
* Upload/download bandwidth limits
* Shared quotas
* Download scheduling
* Streaming support: Allows video/audio streaming while downloading

**VI. Performance Analysis**

**1. Comparison of File Download Methods**

The P2P analysis system offers several advantages over the server-server communication structure:

* Better scalability: Each new addition to the network also adds new resources.
* Active redundancy: Multiple sources for the same file increase reliability and download speed.
* Reliability: There is no single point of failure as in a centralized system.
* Bandwidth efficiency: Distribute the load to multiple partners instead of concentrating on one server.

**2. Partitioned Download Performance**

Initial testing found:

* Downloading songs from multiple partners can improve the overall download speed by up to 2-3 times compared to downloading from a single source.
* Download time decreases as the number of sources increases, but performance gradually decreases as more sources are added.
* 256 KB block size provides a good balance between transfer performance and system performance.

**VII. Conclusion**

The Peer-to-Peer File Sharing System developed in this project demonstrates a robust and decentralized approach to file distribution, where all nodes in the network function equally as both providers and consumers of files. By removing the need for a central server, the system enhances scalability, fault tolerance, and network autonomy—making it a practical solution for environments where centralized infrastructure is not ideal or available.

Throughout the development process, key functionalities were implemented to fulfill the core requirements of a modern P2P file sharing network:

* A peer discovery mechanism based on UDP broadcasting allows nodes to detect and connect with each other automatically within a local network.
* File indexing and search capabilities enable users to look up files across the network using simple keyword queries, improving usability and accessibility.
* The system supports segmented file transfers, breaking files into chunks that can be downloaded in parallel from multiple sources, significantly reducing total transfer time and optimizing bandwidth utilization.
* Integrity checks using SHA-256 ensure that downloaded files are not corrupted or tampered with, adding a layer of reliability and trust to the sharing process.
* Users interact with the system through a command-line interface with intuitive commands like share, search, download, and list, which provide clear feedback and progress updates during file transfers.

The application was tested successfully with at least three active peers, confirming the effectiveness of the core architecture. Each peer runs the same application binary, which simplifies deployment and maintenance, and reinforces the truly distributed nature of the system.

Overall, this project not only achieves the essential features of a decentralized file sharing system, but also lays a solid foundation for future enhancements such as global peer discovery, encrypted communication, persistent file state, and smarter search algorithms. The implementation reflects a well-balanced design between performance, reliability, and extensibility—making it a valuable learning experience in both network programming and distributed systems.

**References**

1. BitTorrent Protocol Specification. [Online]. Available: https://www.bittorrent.org/beps/bep\_0003.html
2. Maymounkov, P., & Mazières, D. (2002). Kademlia: A peer-to-peer information system based on the XOR metric. In Peer-to-Peer Systems (pp. 53-65).
3. Cohen, B. (2003). Incentives build robustness in BitTorrent. In Workshop on Economics of Peer-to-Peer systems.
4. Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.

---end---