CS447 - Computer Networks Project
Final Report

Decentralized Peer-to-Peer File Sharing App

Prepared by:

Ege Yenen, S034413

Hüseyin Bora Baran, S034244

Beril Eda Teberci, S033888

İrem Akova, S034436

İris Akdemir, S034350

1. Abstract

This project presents a decentralized Peer-to-Peer (P2P) file-sharing system designed for robust NAT traversal and secure, reliable file transfers.The idea behind this implementation is to have the peers learn their public IP address and port mappings using the STUN protocol, thereby allowing them to set up their own peer-to-peer connections over UDP hole punching. This Python-written system has a GUI with Tkinter for human interaction and a modular backend for connection management and cryptographic security: it uses RSA for key exchange and AES for session data security, as well as a custom reliable UDP transfer protocol to implement file sharing between peers without the need for relay servers. It transmits files in chunks, using checksums, acknowledgments (ACKs/NAKs), and retransmissions to ensure file integrity and completeness.

2. Introduction

The transformation of digital communication has always been accompanied by a consumer demand for the fast file-sharing and resilient mechanism. Peer-to-Peer (P2P) architectures follow a decentralized design, giving the users an opportunity to exchange data between themselves, thereby decreasing the server load and the possibility of having a single point of failure. Let us consider a big problem for Pure P2P systems: NAT is a nearly universal technique applied on private network IP addresses. This serves to hide the internal IP addresses, making it difficult for an external connection to connect with the P2P inside.

In the past, there were various solutions for NAT traversal such as UPnP, manually opening ports in NAT boxes, or using relays. While the relay server ensures connectivity by forwarding traffic, it also ensures centralization in routing, potential bottlenecks, and costs to run that server. Many pure P2P systems attempt direct connections and are thus faced with the problem of traversing complex NAT behaviors.

This project tries to address the file-sharing problem in the decentralized P2P paradigm, solving the NAT traversal problem by using the STUN protocol and UDP hole punching. Using STUN, a peer learns public-facing IP address and port and the type of NAT it is behind. The peers then use this information to attempt a UDP hole punch whereby peers simultaneously send UDP packets to each other to open a communication path through their respective NATs.

The system is built in Python, with a Tkinter GUI for ease of use. Users can initialize their P2P node, identify their network information, connect to other peers by exchanging connection information such as node ID, external IP, external port, and thereafter send files or exchange messages directly. Security is provided with an RSA key exchange mechanism for generating a shared AES session key used to encrypt all communication and file transfer data. The files are transferred over UDP very reliably with a custom protocol that chunks files into pieces, prepares each chunk with checksums for integrity, collects acknowledgments and negative acknowledgments (ACKs/NAKs) per chunk, and keeps retrying.

This paper presents the design, the realization, and the facilities of this decentralized peer-to-peer file exchange system. It demonstrates the system's treatment of NAT traversal, the secure exchange of messages, and reliable data transfer, representing a genuine approach to direct peer-to-peer file exchange.

By embracing direct P2P connectivity through STUN and hole punching, this project strives to provide an efficient file exchange player that is truly decentralized and requires minimal infrastructure as intermediary entities during the data transfer stage.

3. Methodology

Our project revolves around building and deploying a decentralized P2P file-sharing application. A key objective here is to allow users to connect and exchange files in a direct and robust manner, especially the notably challenging NAT traversal problem. NATs can disguise the true network reachability of peers and hence hinder traditional P2P connections; these NAT devices are mostly located at home and corporate networks. The approach is to embed known NAT traversal techniques within the peer application itself so that two users can share files directly without going through data relay servers.

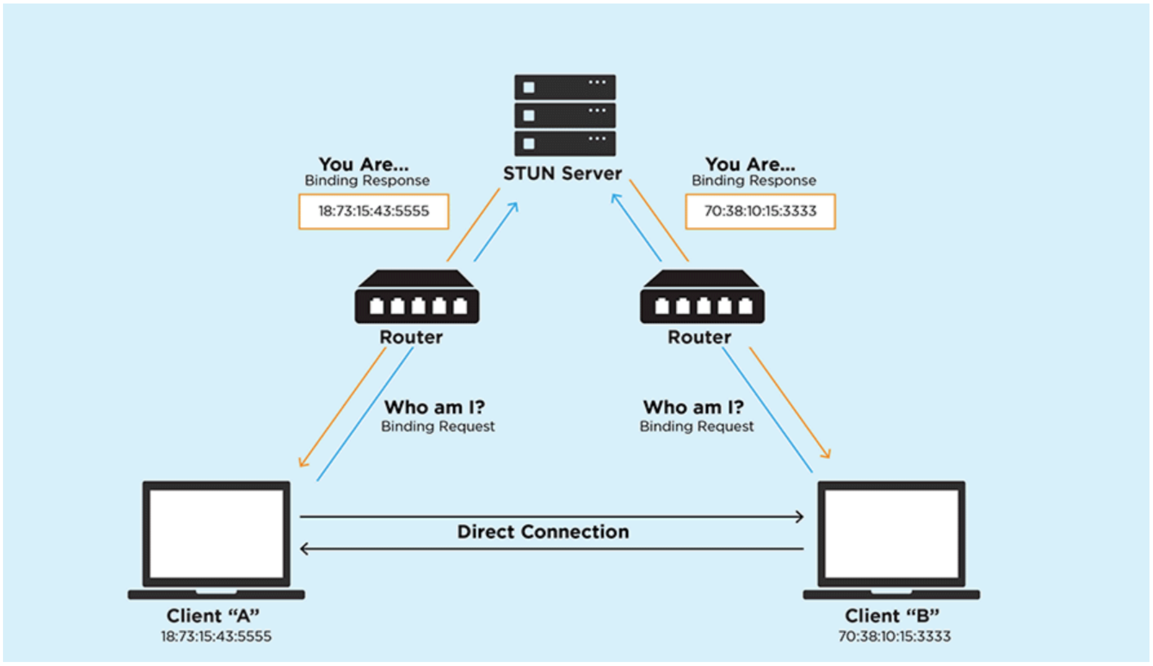


Figure.1: Process Diagram (Conceptual for STUN/Hole Punching)

a. Approach

In decentralized P2P file-sharing systems, direct connections between peers become the value technical hurdle because the peers are operating behind NATs. Our approach attacks this directly, in that we use STUN (Session Traversal Utilities for NAT) to discover the network configuration and UDP hole punching to present direct communication channels between the peers. When the user starts the application, their P2P node queries a set of public STUN servers. This interaction allows the peer to learn its public IP address and the port assigned as seen by the rest of the world, along with a superficial assessment of its NAT type. The discovered network identity encoded as a connection string consists of a unique Node ID, the public IP, and public port. Next, it must be exchanged between users by some external means such as messaging software or email. When Peer A tries to use the connection string to actually contact the Peer B, both applications simultaneously start sending UDP packets to the respective IP and ports where the other one is identified. This simultaneous outward traffic opens the NAT to allow direct communication, translating it into direct bidirectional UDP paths. Once this path opens, the interfacing file transfer and messaging are done directly between the two peers, keeping the system decentralized.

b. Tools and Platforms

This P2P application construction rests on tools and software platforms chosen carefully for sustaining its decentralized architecture and network communication requirements. Python has been chosen as the language for development, with networking, concurrency, UI development, and security libraries alongside it. A significant part of the NAT traversal method is held by external network services, namely public STUN servers.

i. Public STUN Servers

These STUN servers (e.g., stun.l.google.com, stun.ekiga.net) are public resources crucially used for NAT discovery. In NAT discovery, each peer queries the STUN server in order to learn its external address from the perspective of the public internet, i.e., an IP and port. This information is critical to even attempt UDP hole punching. It must be noted here that STUN servers are utilized for discovery only and are not used in the actual data transfer between the peers.

ii. Python

Python was regarded as the appropriate principal language for the entire application, hence encompassing the P2P node logic on the client-side, graphical user interface, file transfer, and cryptographic operations. Due to its comprehensive standard library, in particular for networking and operating system calls, and its richness in the third-party package ecosystem, it is best suited for the development of complex network applications and for accelerating development cycles.

iii. Libraries

Socket: This standard Python library is mainly the basis for low-level networking. It can create UDP sockets and manipulate them for sending STUN requests, setting up peer-to-peer connections, and transferring packets between peers.

Threading: Due to the concurrent nature of P2P applications, the majority of the code uses threading. Threading enables the application to concurrently listen to incoming messages from peers, carry out file transfers in the background without blocking the GUI, and keep the GUI from freezing.

Tkinter: Tkinter is used for a graphical interface. Tkinter is Python's standard interface to the Tk GUI toolkit. It allows making an interactive and user-friendly application window through which the node can be configured by the user, connections can be managed, and files can be shared.

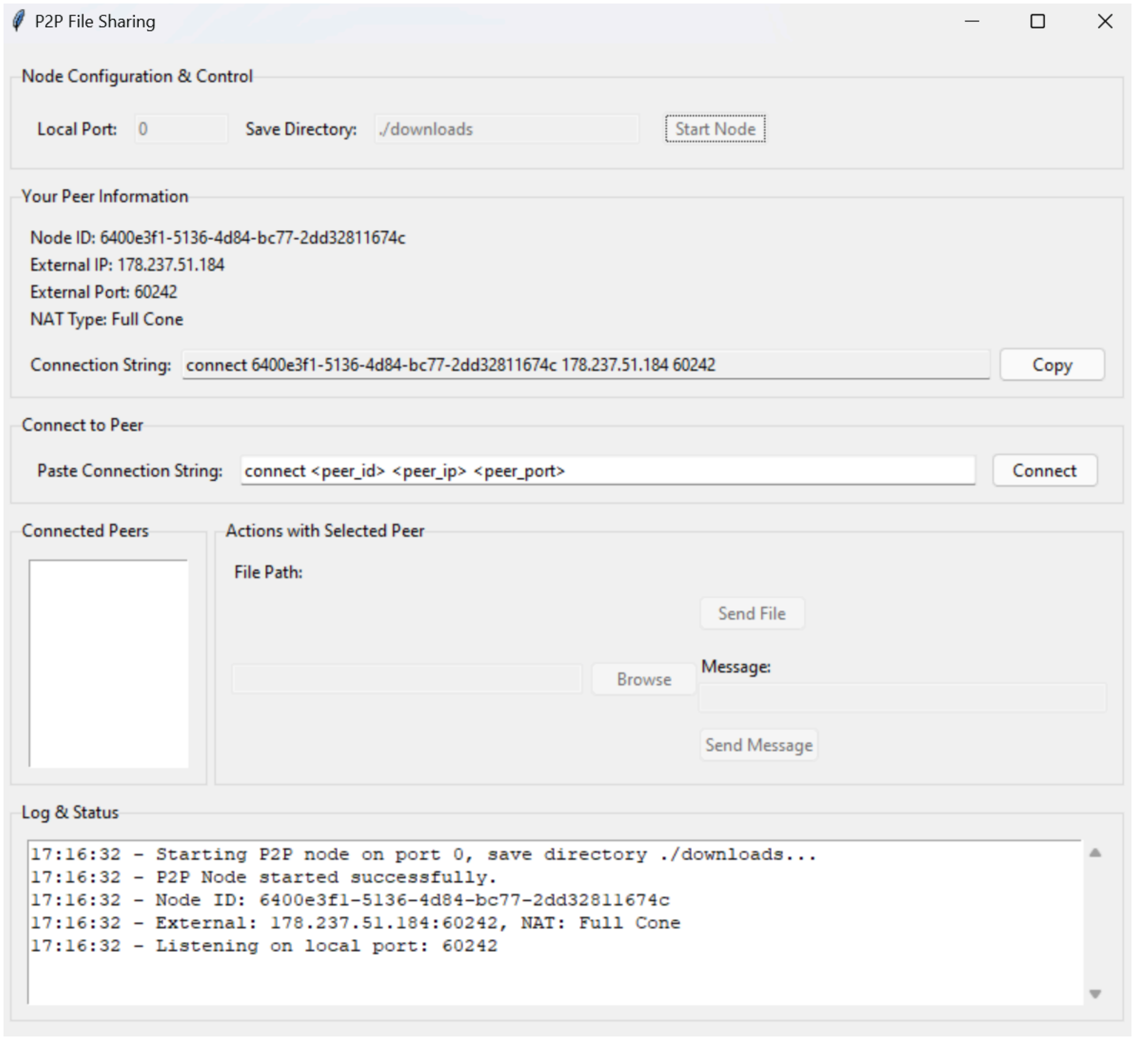
OS: This library interacts with the operating system. It uses this for manipulating file paths (e.g., building save paths for downloads), checking the existence of files to send, and creating directories when necessary (e.g., download directory).

Cryptography (cryptography.io library): For secure communication, this library is needed. It supplies cryptographic primitives necessary for making RSA key pairs (used in the initial handshake for authentication and secured ses

UUID: To ensure each peer in the network can be uniquely identified, the uuid library is used to generate universally unique identifiers (UUIDs) that serve as Node IDs.

c. Graphical User Interface

Tkinter library has been used for creating a GUI aimed at providing the user with a simple, friendly, and practical experience for handling the P2P Node and interacting with other nodes. The design maintains simplicity to enable even users without a technical background to work on the application's core functionalities.



Local Port: 0
Save Directory: ./downloads

Node ID: 6400e3f1-5136-4d84-bc77-2dd32811674c
External IP: 178.237.51.184
External Port: 60242
NAT Type: Full Cone

Connection String: connect 6400e3f1-5136-4d84-bc77-2dd32811674c 178.237.51.184 60242

Paste Connection String: connect <peer_id> <peer_ip> <peer_port>

```
17:16:32 - Starting P2P node on port 0, save directory ./downloads...
17:16:32 - P2P Node started successfully.
17:16:32 - Node ID: 6400e3f1-5136-4d84-bc77-2dd32811674c
17:16:32 - External: 178.237.51.184:60242, NAT: Full Cone
17:16:32 - Listening on local port: 60242
```

Figure.2: User Interface

i. Node Control and Information Display

The GUI provides the main section for node configuration and control. Here, users determine the local UDP port they want their node to listen to (with '0' as a user input, the system will pick a random available port) and then designate a local directory under which downloaded files should be saved. A large button labeled "Start Node" will activate the P2P node and invoke the STUN discovery process to know about its public network presence. After the node starts, the "Your Peer Information" section will be populated with the node's unique ID, its discovered external IP address and port number, and its discovered NAT type. This section also contains the essential "Connection String" (e.g., connect <node_id> <external_ip> <external_port>) that a user must communicate with others to enable connections alongside the "Copy" button.

ii. Peer Interaction and File Transfer

An input field allows forge connections to other users who paste a peer's connection string and hit an accompanying "Connect" button to instigate the UDP hole-punching attempt. Once connections are forged, a listbox displays "Connected Peers" who show their Node IDs and network addresses. A context-specific action panel is activated once a released peer is selected from the list, which can then either be used to "Send File" with file path selection for which a "Browse" button assists, or "Send Message" through a text input box. A scrolled text area completes the picture with a "Log & Status" display, giving outnode feedback on node operations, connection attempts, received messages, and the status of file transfers.

iii. Design and Usability

The layout is neatly divided into logical sections using LabelFrame widgets for clarity. Standard Tkinter widgets are used to preserve the familiar look-and-feel. Buttons and input fields would be dynamically enabled and disabled based on whether the node is in an operational state or not: for example, until the node is started, all connection-related buttons will be disabled. Error messages, warnings, and critical notifications are conveyed via message box pop-ups. The GUI is programmed to refresh periodically to update the user on incoming messages, connected peers, and file transfer statuses.

d. Backend

The backend of the P2P application includes the core logic constituting the application's working method, i.e., the processes for NAT traversal, setting up secure communication, establishing reliable file transfers, and handling concurrency. As soon as peers get connected, this works in a decentralized manner

i. NAT Traversal and Connection Management

This very important component, implemented in large part within p2p_node.py supported by stun_client.py, begins its work with STUNClient sending STUN binding requests to public servers so as to discover the node's external IP and port, and perhaps more indicative of the NAT type by parsing XOR-MAPPED-ADDRESS attributes. Upon node initialization, a Node ID is spawned, the STUNClient discovers the NAT, and a local UDP socket is bound. connect_to_peer sends the first UDP packets to the external address of the peer for hole punching purpose. Then it sends a "hello" message that carries the Node ID and public RSA key of the sender. The system stores a dictionary of known peers, their addresses, public keys, and connection states, updated from time to time when "hello_ack" messages are received, thus confirming the completion of handshaking. A separate thread constantly listens on that UDP socket, parsing incoming JSON control messages or detecting binary file chunks.

ii. Secure Communication

Such security is now implemented through functions in crypto.py and p2p_node.py. Each node at startup generates its RSA key pair. The public key is sent during the "hello" handshake. Then an AES session key (a key and an initialization vector) is created for each peer-to-peer session by either peer. This AES key package is encrypted with the public RSA key of the recipient peer and sent in a "key_exchange" message. The recipient uses its private RSA key to decrypt this package and obtain the shared AES session key and sends out a "key_exchange_ack" message confirming receipt. All further file data and potentially confidential messages are to be encrypted using the AES session key, thus preserving their confidentiality (however, the crypto.py given contains passthrough encryption/decryption for testing).

iii. Reliable File Transfer

The file_transfer.py module mixed with p2p_node.py is a sturdy file transfer protocol over UDP. The file is split into chunks of fixed sizes (say 32KB). Transfer begins with a FILE_INIT message (with file metadata: name, size, total chunks, transfer ID), which is acknowledged by the receiver with FILE_INIT_ACK. For each chunk, the sender sends a JSON FILE_CHUNK header (containing a transfer ID, chunk index, and a checksum of the encrypted chunk data), immediately followed by the binary chunk data (which may be encrypted). The receiver calculates a checksum from the binary data received, compares it to the header's checksum, and, if it matches, goes on to process the chunk (decrypt, write to file) and sends back a CHUNK_ACK. In the event of a mismatch, a CHUNK_NAK is sent back instead. The sender will await an ACK for every chunk; if a timeout occurs or a NAK is detected, the chunk is sent again, with the cycle repeating until the maximum number of retries, MAX_RETRIES, is reached. After all chunks have been ACKed, the sender sends out the FILE_END, which is confirmed by the receiver with FILE_END_ACK. Callbacks inform the node and the GUI of progress.

iv. TCP File Transfer and ngrok Integration Attempt

We considered integrating TCP-based file transfers in addition to our reliable UDP protocol to deal with scenarios in which UDP traversal might not work or be blocked. We did this by setting up a TCP server on each peer, which allowed files to be transferred directly over an assigned TCP port. However, NAT limitations presented difficulties when we tested across two PCs on different networks. Unless particular port forwarding rules are set up on the router, inbound TCP connections are usually blocked in these networks. In order to overcome this restriction, we tried using ngrok, a tunneling service that offers public TCP endpoints that are mapped to local ports. The public address was used by the other peer to establish a connection after one of the peers ran a ngrok tunnel on it. Although this made TCP-level communication possible, complete integration with our peer finding and key exchange method is still something that has to be worked out. When standard NAT traversal techniques are ineffective, this method showed how tunneling services like ngrok might provide a possible backup for direct peer-to-peer file transfers.

v. Multi Threading

Concurrent operations in the application are handled via Python's threading library. The main loop for handling messages (_handle_messages in P2PNode), which spends much time listening for data over the network, runs in its own thread. A file upload method call (_upload_worker in FileTransfer) also runs in its own thread to allow transfers to be carried out concurrently, without blocking the rest of the application or GUI updates (update_ui_status in P2PGUI). These updates are scheduled with Tkinter's root.after function, which guarantees that all UI updating is done safely from within the main Tkinter thread. Temporary threads might also be used in the connect_to_peer connection retry logic.

vi. Error Handling

There exist mechanisms for handling probable errors in the backend. Handling includes catching and logging errors in the socket or network operation, JSON errors due to malformed messages, and timeouts during communication waits. File I/O errors are things like file-not-found situations during send or permission issues during save. Invalid messages or data from unknown peers will typically be logged and discarded by the system. Failures of file transfer, e.g. upon exceeding max retries for a chunk, are logged and reported with status updates visible in logs.

4. Results and Discussion

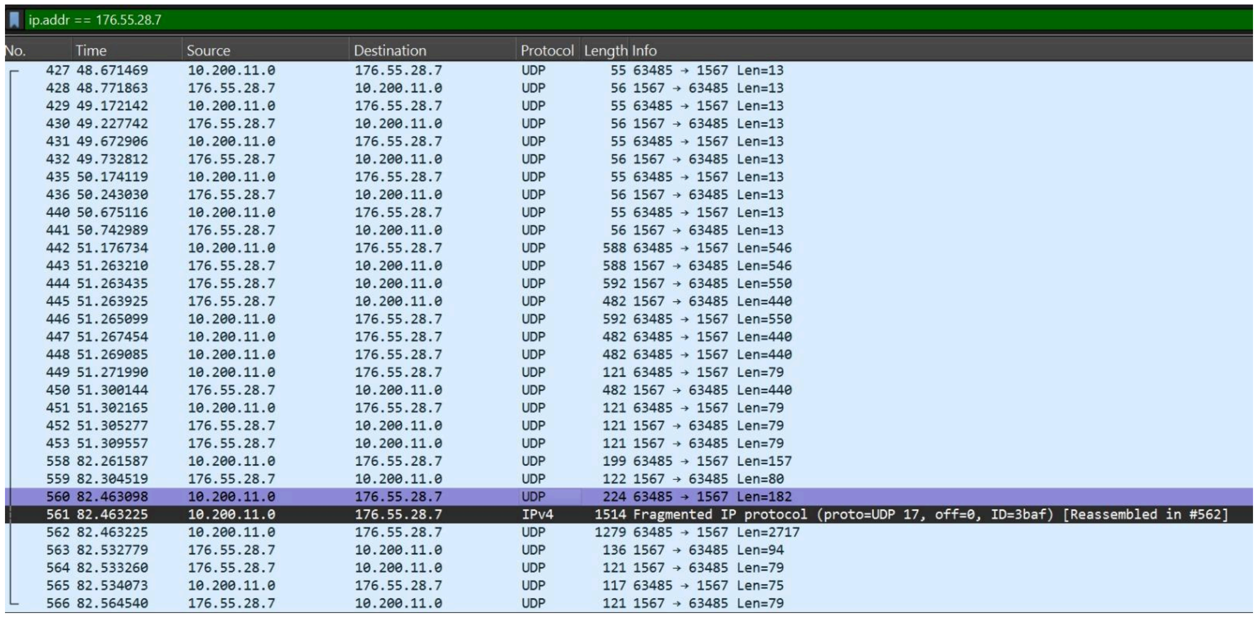
The intent of the study was to provide a functional system embodying a decentralized P2P file-sharing application allowing for the operation through NATs without the presence of a central relay server. This implementation provides the support of a STUN-based NAT discovery and UDP hole punching, which takes care of creating a direct peer-to-peer connection in many common network configurations.

The STUNClient module enables querying trusted public STUN servers so that peers relate to their external IP addresses and ports, as well as assess a simplified NAT type. This information is very important in the subsequent UDP hole-punching attempts carried on by the P2PNode. These connection handshakes using "hello" and "key_exchange" messages serve to establish not only connectivity but also a secure channel with RSA-encrypted AES session keys. The FileTransfer module provides a layer for reliable transmission of data over UDP. With each file being split into chunks, accompanied by per-chunk checksums, acknowledges (ACKs/NAKs), and retries, the chunking mechanism itself substantially decreases the risk of any consequences due to UDP packet loss or corruptions so that the files will be reliably and correctly delivered from one peer to the other.

The Tkinter-based GUI (P2PGUI) affords the user an interface for configuring nodes, connecting peers, and starting file or message transfers. Real-time log display provides feedback to the user regarding operational and transfer statuses for the node.

Development was mainly challenged by the inherent difficulty of NAT traversal and unreliable UDP graces. UDP hole punching does not always work; its success depends on the kind of NATs involved and in particular symmetric NATs cannot be hole-punched very easily. Whereas our current STUN implementation determines a very basic NAT type (and simplifies it to "Full Cone" if any success can be achieved), conducting a more elaborate NAT type determination would inform more sophisticated hole-punching strategies or inform when a TURN relay (not on our table) should be called for. Debugging distributed behavior of multiple peer instances under different network scenarios demanded a lot of logging and testing.

Still, these hurdles offered evidence to demonstrate the viability of STUN and UDP hole punching in pairs to establish direct P2P connections. The design of the system in a modular way kept separate concerns such as NAT discovery, cryptography, file transfer logic, and the GUI; this proved useful for both development and testing. The system, by way of fulfilling this wish, presents the ability to share files directly; in other words, a user can share files directly through this system, representing in spirit a distributed P2P network.



```
ip.addr == 176.55.28.7
No.   Time        Source        Destination   Protocol Length Info
  427 48.671469   10.200.11.0   176.55.28.7   UDP        55 63485 → 1567 Len=13
  428 48.771863   176.55.28.7   10.200.11.0   UDP        56 1567 → 63485 Len=13
  429 49.172142   10.200.11.0   176.55.28.7   UDP        55 63485 → 1567 Len=13
  430 49.227742   176.55.28.7   10.200.11.0   UDP        56 1567 → 63485 Len=13
  431 49.672906   10.200.11.0   176.55.28.7   UDP        55 63485 → 1567 Len=13
  432 49.732812   176.55.28.7   10.200.11.0   UDP        56 1567 → 63485 Len=13
  435 50.174119   10.200.11.0   176.55.28.7   UDP        55 63485 → 1567 Len=13
  436 50.243030   176.55.28.7   10.200.11.0   UDP        56 1567 → 63485 Len=13
  440 50.675116   10.200.11.0   176.55.28.7   UDP        55 63485 → 1567 Len=13
  441 50.742989   176.55.28.7   10.200.11.0   UDP        56 1567 → 63485 Len=13
  442 51.176734   10.200.11.0   176.55.28.7   UDP       588 63485 → 1567 Len=546
  443 51.263210   176.55.28.7   10.200.11.0   UDP       588 1567 → 63485 Len=546
  444 51.263435   176.55.28.7   10.200.11.0   UDP       592 1567 → 63485 Len=550
  445 51.263925   176.55.28.7   10.200.11.0   UDP       482 1567 → 63485 Len=440
  446 51.265099   10.200.11.0   176.55.28.7   UDP       592 63485 → 1567 Len=550
  447 51.267454   10.200.11.0   176.55.28.7   UDP       482 63485 → 1567 Len=440
  448 51.269085   10.200.11.0   176.55.28.7   UDP       482 63485 → 1567 Len=440
  449 51.271990   10.200.11.0   176.55.28.7   UDP       121 63485 → 1567 Len=79
  450 51.300144   176.55.28.7   10.200.11.0   UDP       482 1567 → 63485 Len=440
  451 51.302165   10.200.11.0   176.55.28.7   UDP       121 63485 → 1567 Len=79
  452 51.305277   176.55.28.7   10.200.11.0   UDP       121 1567 → 63485 Len=79
  453 51.309557   176.55.28.7   10.200.11.0   UDP       121 1567 → 63485 Len=79
  558 82.261587   10.200.11.0   176.55.28.7   UDP       199 63485 → 1567 Len=157
  559 82.304519   176.55.28.7   10.200.11.0   UDP       122 1567 → 63485 Len=80
  560 82.463098   10.200.11.0   176.55.28.7   UDP       224 63485 → 1567 Len=182
  561 82.463225   10.200.11.0   176.55.28.7   IPv4     1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=3baf) [Reassembled in #562]
  562 82.463225   10.200.11.0   176.55.28.7   UDP      1279 63485 → 1567 Len=2717
  563 82.532779   176.55.28.7   10.200.11.0   UDP       136 1567 → 63485 Len=94
  564 82.533260   176.55.28.7   10.200.11.0   UDP       121 1567 → 63485 Len=79
  565 82.534073   10.200.11.0   176.55.28.7   UDP       117 63485 → 1567 Len=75
  566 82.564540   176.55.28.7   10.200.11.0   UDP       121 1567 → 63485 Len=79
```

Figure.3: Packet Analysis

5. Conclusion and Future Work

Through this project, a truly decentralized P2P software was created, which uses STUN for NAT interaction and UDP hole punching for making direct connections that can certainly ensure a safe and sound transfer of files. The Python system with the Tkinter GUI very concretely demonstrates the P2P concept in practice. The reliable UDP protocol created by us, encryption for session security, and the easy-to-understand interface all combine to provide a workable file-sharing tool.

The project basically operates around the problem of achieving NAT traversal for Direct P2P communication without relying on a central data relay server. In a modular manner, its backend implements peer connection, secure key exchange, and reliable file transfer through chunking, checksums, and an ACK/NAK-based retry mechanism.

Future Enhancements:

Automated Peer Discovery: Introduce a decentralized discovery mechanism (for instance, a Distributed Hash Table like Kademlia) or a lightweight tracker server to allow peers to find one another without manually changing connection strings.

Performance and Scalability: Allow concurrent transmission of files to/from a multitude of peers. Optimize chunk sizes based upon network conditions.

In summary, this project sets up a strong basis for a decentralized P2P file-sharing application. NAT traversal techniques implemented successfully, followed by a secure communication protocol and a file-transfer mechanism, clearly give proof to the possibility of a fast, direct peer-to-peer network solution. The future works can assure the robustness, usability, and feature set of this project, making it an even more viable P2P tool.

6. References

Rosenberg, J., Weinberger, J., Huitema, C., & Mahy, R. (2008). STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 5389, IETF.

Ford, B., Srisuresh, P., & Kegel, D. (2005). Peer-to-Peer Communication Across Network Address Translators. Proceedings of the USENIX Annual Technical Conference.

Kurose, J. F., & Ross, K. W. (2021). Computer Networking: A Top-Down Approach (8th ed.). Pearson