**Table of Content:**

1. ABSTRACT
2. INTRODUCTION
3. ALGORITHM
4. IMPLEMENTATION
5. CODES
6. RESULT & INFERENCES
7. APPLICATION ORIENTED LEARNING
8. CONCLUSION
9. REFERENCES

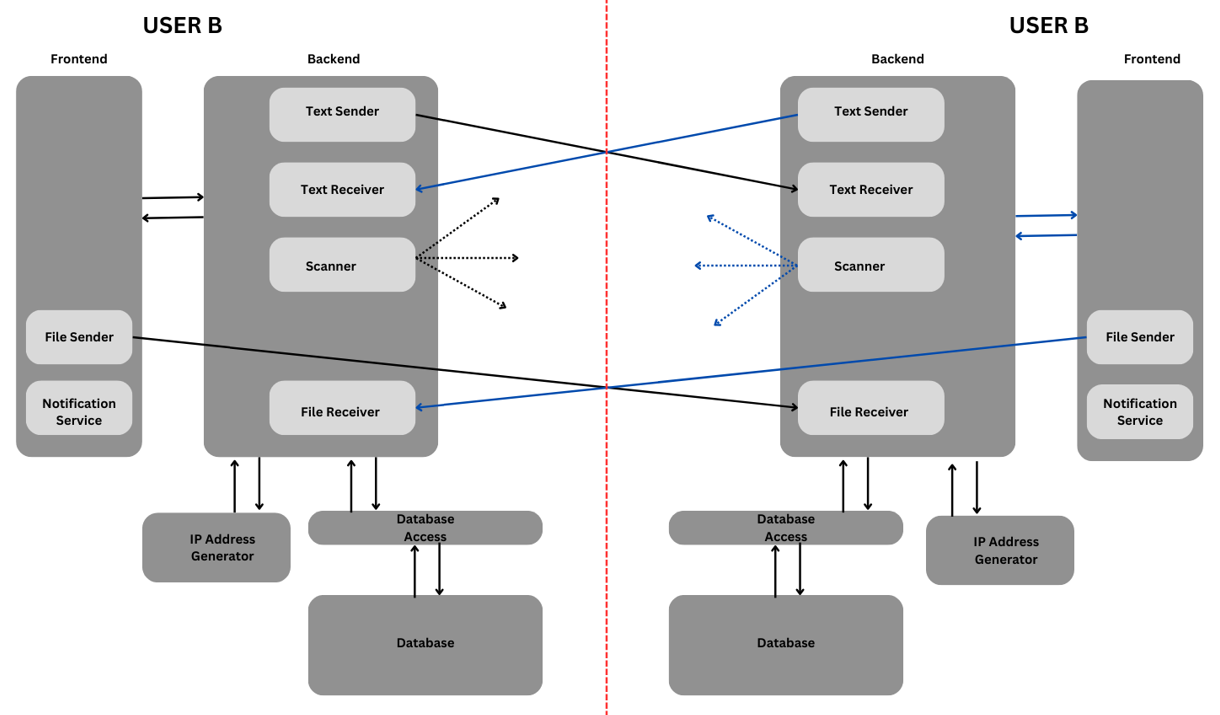
**ABSTRACT:**

This report outlines the design and implications of a peer-to-peer (P2P) communication application which is confined to a Local Area Network (LAN). In the system every user acts both as a client and a server depending on the situation thus allowing the exchange of messages and files. The application has three basic parts: a graphical user interface which is the end-user part of the system, an API which responds to clients requests, and processor programs to carry out tasks that need heavy computing such as generating IP addresses and accessing databases. Using a decentralized model, there is less delay in communication and better interaction in real time because there is no central server. This system is oriented towards providing direct archival peer-to-peer messaging that is enhanced with file transfers services in a closed network, and as such is scalable and modular in nature.

**INTRODUCTION:**

Most often, the central servers in modern communication networks can cause latency, bottlenecks, or breach of privacy. The objective of this project is to address the problems posed by a central server by introducing a P2P communication system. The application lets users send and receive messages and share files over devices connected to the local area network with each other. Here every user acts as a client and a server depending on the task. The system relies on simple HTTP GET and POST requests, JavaScript-based notification services, and multi-threaded scanning to find devices. By separating the front end, API handler, and processing units of the application, the system is both effective and flexible in design and use. This approach enables internal communication in termed networks devoid of changes while allowing other changes in the future.

**ALGORITHM:**



**Fig: Flow Chart of Application Design and Working**

The whole application is designed on the concept of a model in which each user (running application of user system) is both client as well as server depending upon the scenario. For example, during the sending of any test message the sender is a client and receiver is a server.

The whole application can be divided into three parts as follow: Frontend, API handler and Processor Program

**Frontend:**

This is a basic HTML and JavaScript based web page for user interaction. These are the function performed by it –

1. It receives input from user like text message to be sent, file to be send, receiver ID, display the messages and continuously check for any received file or messages.
2. After receiving any given input from user, it make a simple HTTP Post request to it server to upload the text input
3. **Notification Service** : - It is a function written in JavaScript which continuously make a get request to server and in response it may gets the ID and name of user which send either text or file. This function is written as asynchronous function having infinite while loop repeating after each 1 sec. The asynchronous nature of the function make it to run continuously in a separate thread parallel to the main thread helps in natural execution of rest of the program.
4. **File Sender** :- As the HTTP is no longer a protocol for only sending text, but using post method of HTTP not only longer text can be send but also various files irrespective whether it is a video file, image file, etc. File sender is also a function which send file to the receiver. When user select a file and hit the send button then its functionality starts. After checking basic details like valid file and ID, it fetch the current IP address of the given receiver ID from server, prepare a post request on received IP address with attached file, then make the post request on the receiver. Here it should be noted that, file is not send on the server first to dispatch it to receiver, it is directly send from frontend itself

**API Handler:**

It is a part of server/backend. Its sole purpose it to receive API request (in the form of either http GET or POST).

It has two category of API routes :-

(a). API for interaction with frontend. Like for receiving inputs, sending messages and other information

(b). API for handling core functionality like sending & receiving data and scanning.

There are various HTTP routes for different purpose, routes for implementing core functionality is shown in block diagram and explained below-

1. **Text Sender: -** This route receive the message and ID of the receiver (for sending message) from frontend. Interact with database using a program called Database\_Access to get the current IP address of the receiver ID. Prepare API which include text message and sender ID, then make a get request to the API. (<http://receiverIP:port/receiveText?id=senderID&message=MESSAGE>)
2. **Text Receiver:-** This route listens for any http get request on “/receiveText” route. On receiving such request it extract sender ID and message from the API. Interact with Database to update the message on sender ID. It also extract the IP address from where this API is send and update this current IP address of the sender in it database.
3. **Scanner:**- On receiving any request for scanning from frontend it does following-
   * 1. It generate a list of all possible IP address on the current local network using subnet mask and the IP address of the device with the help of Processor Program called IP\_Address\_Generator
     2. Start making get request by attaching it ID to each IP. With the help of multithreading these requests are send parallelly in each separate thread making the scanning process much faster.
     3. Each request has a timeout of 2-3 seconds. If before the timeout the response is not received for given IP address then it is assumed that that device is unavailable.
     4. For the purpose of making response, “/respond” route is defined. When any device receive request on this route then it not only respond to the sender but also capture the sender IP address and ID(as scanning device attach it ID in get request) and update in its databse.
4. **Files Receiver:-** Unlike Text Receiver, it listens for POST request having a file attached on the request. On receiving such request it extract file and ID. Save the file by naming filename as ID\_originalFileName.
5. **Enabling Notification Feature:-** A simple technique is implemented. For this feature a global variable is maintained. Whenever Receiver route receive something then it override the variable with sender ID otherwise just a empty string. When a request from frontend is received for any update then the variable value is send in response.

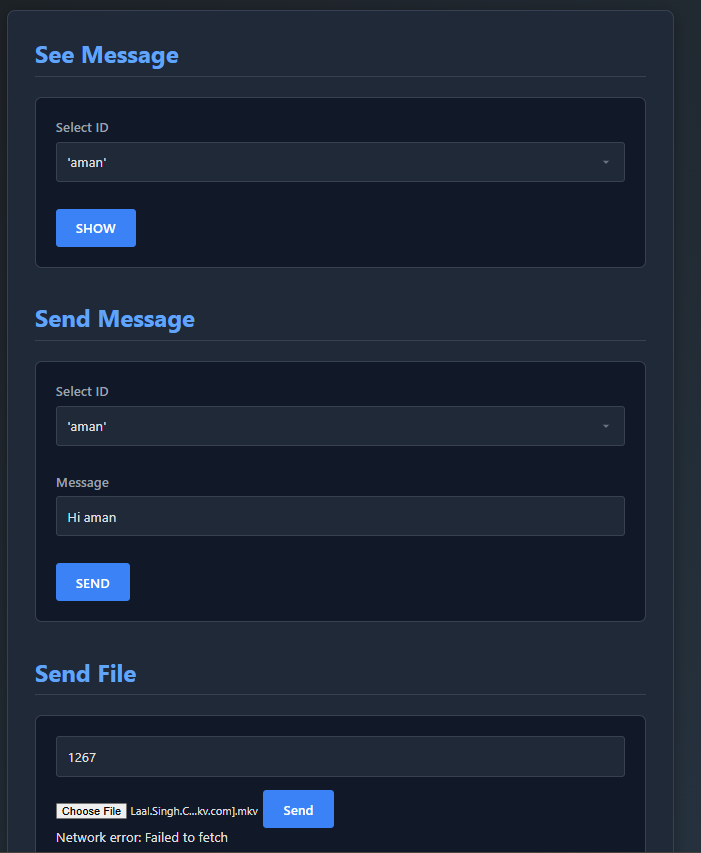
**Processor Files: -**

These are the collection of program files which helps in any additional complex processing, removing the burden of complex computations from API Handler. It is also a part of backend. It has two files:-

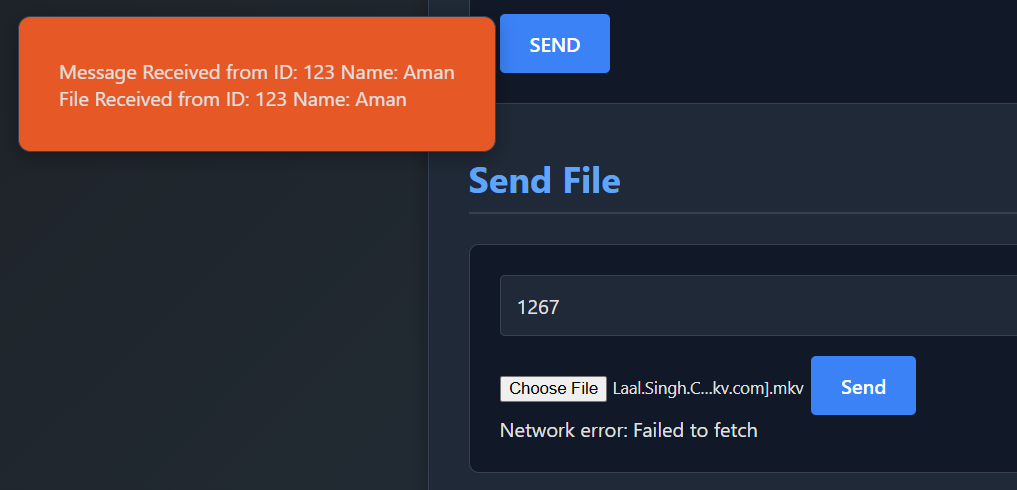
1. **IP\_Address\_Generator :-** Its function is to generate a list of IP addressed after performing computations on subnet mask and user current IP address and return this list to the API Handler
2. **Database\_Access :**- It further smoothens the process of interacting with database. It has several function defined. Like:-
   * 1. **getIP(id) –** return the IP address
     2. **addIP(id, ip, name) –** update/add the database with given details
     3. **getMessage(id) –** return list of all sent/received message on given id
     4. **getName(id) –** return name of given id
     5. **addMessage(id, message, send/receive) –** append the latest message to given id
     6. **getNameIdList():** return a list of all id and corresponding name

**IMPLEMENTATION (REAL TIME):**

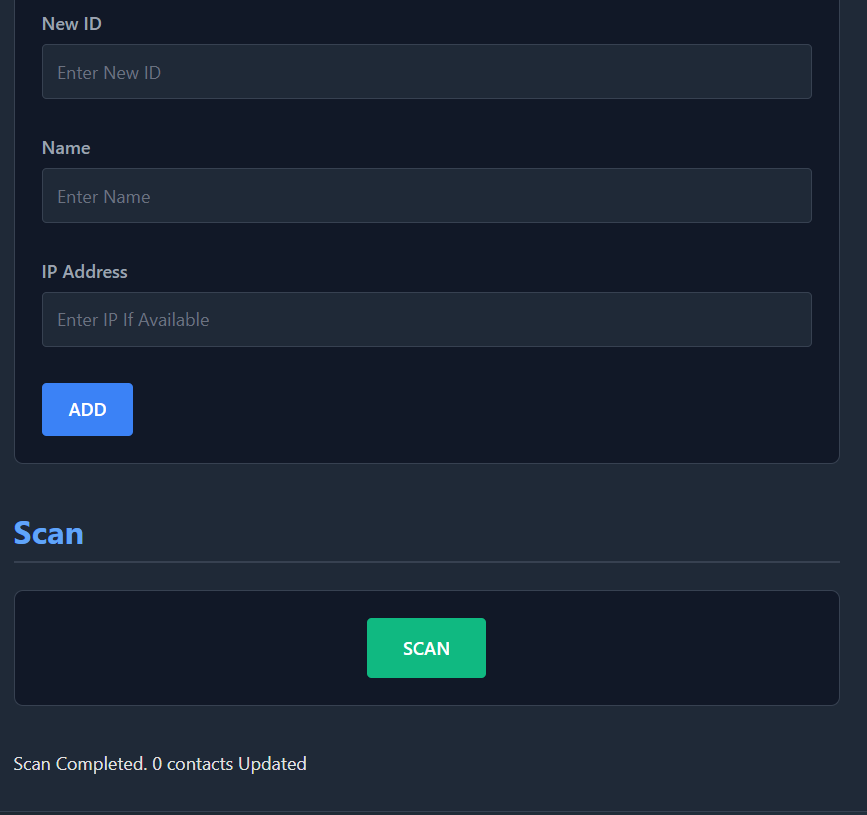
**User Interface: -**



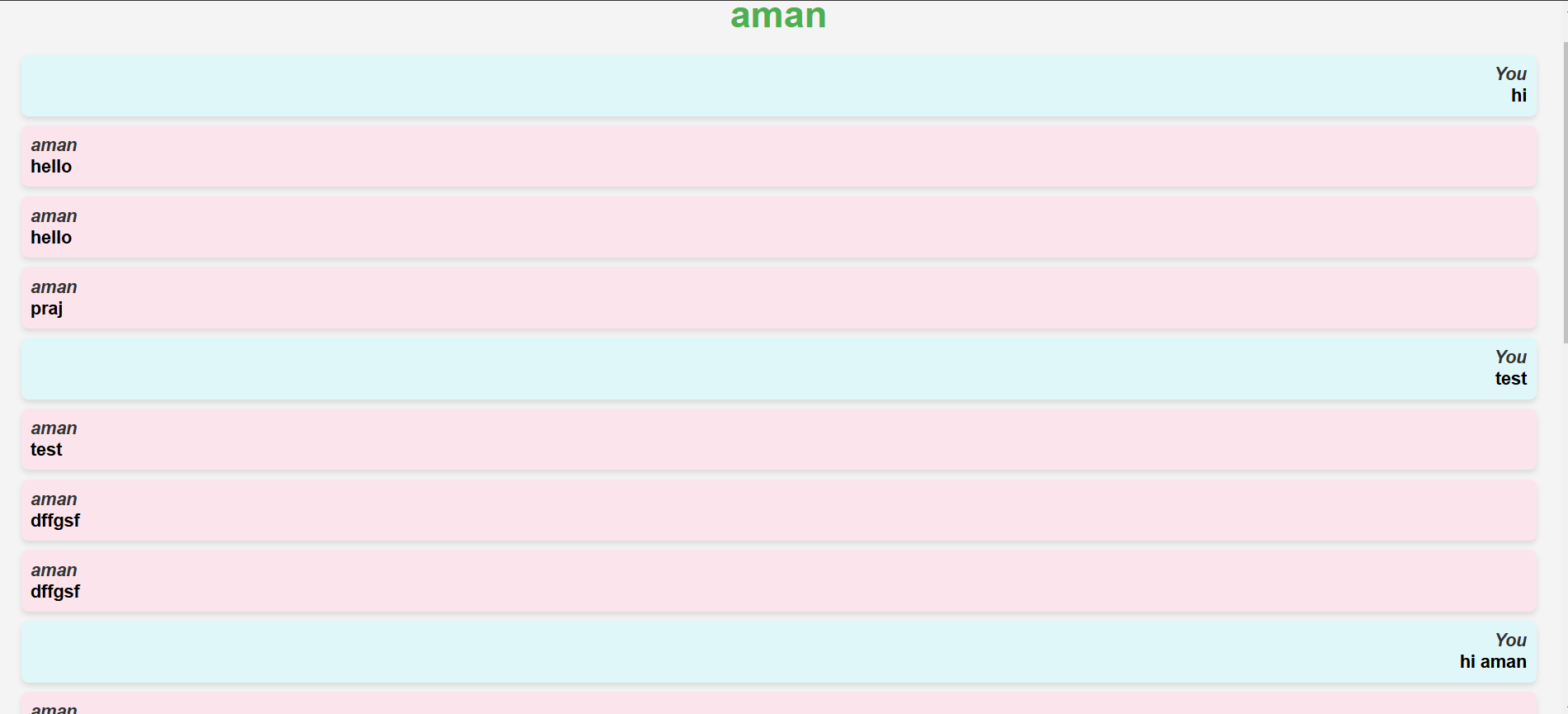
**Users can either view their previous messages, send a new message or file by selecting their saved contact from dropdown.**

****

**Get the notification and update**

****

**Add contact to the database and scan to update its current IP address**



**Interactive User Interface for messages**

**Technical Description as a Prototype Product**

The application functions as a peer-to-peer communication tool within a local network, with each user acting as both a client and server depending on the context of the interaction. This dual-role design allows users to send and receive messages or files directly from one device to another without centralized routing, reducing latency and enhancing privacy.

The application is structured into three primary components:

1. **Frontend**: The interface allows users to send messages, select and upload files, and view incoming messages.
2. **API Handler**: The backend serves as the main processing unit for the application, managing data received from the frontend and performing core functions such as sending and receiving text, managing the file-sharing process, and handling IP scanning requests.
3. **Processor Files**: Supporting the API Handler, the Processor Files manage essential tasks like IP list generation for network scanning and database interactions, freeing the main API from handling these computationally intensive processes.

This architecture ensures smooth communication within the network, where each device is dynamically detected and added, allowing for real-time data exchange without centralized dependencies.

**Standard:**

Currently it doesn’t have commercial and not a production grade standard. Having many security concerns. This is just technical demonstration from idea to implementation. However, the separation of files for dedicated function implements a modular and clean design which is very scalable and maintainable. This design strategy helps in simultaneous development and scaling of each modules without depending much on other part.

**Testing, Bugs and Errors: -**

Current implementation doesn’t pass from industry standard testing. However, a robust error handling is maintained throughout the application to prevent it from crash in case of possible error, instead display informative message. A bug is found as follow – during scanning an existing IP address in network gets skipped, on after restarting the server it works properly.

**CODES:**

**API Handler:-**

from flask import Flask, request, jsonify, render\_template

import IPListAccess, genIPList

import requests

import os

from flask\_cors import CORS

from concurrent.futures import ThreadPoolExecutor

app = Flask(\_\_name\_\_)

PORT = 3000

CORS(app)

MESSAGE = [] #this will we equal to the id of the sender when receve something

FILE = [] #this will we equal to the id of the sender when receve something

UPLOAD\_FOLDER = 'uploads'

if not os.path.exists(UPLOAD\_FOLDER):

    os.makedirs(UPLOAD\_FOLDER)

app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

# ------------------------------------------------------------------------------------------

#HOME PAGE

@app.route('/', methods=['GET'])

def start():

    return render\_template("index.html")

#API TO FETCH AND DISPLAY MESSAGES

@app.route('/showmsg', methods=['POST'])

def show1():

    try:

        id=request.form.get('id')

        return render\_template('showmsg.html', name=IPListAccess.getName(id), message=IPListAccess.getMessage(id))

    except:

        return "cannot show Message"

# API TO SEND MESSAGE(link destination IP, message). IT SEND THE MESSAGE BY MAKING THE GET REQUEST TO DESTINATION IP

@app.route('/sendText', methods=['POST'])

def send1():

    try:

        id=request.form.get('id')

        receiver\_ip = IPListAccess.getIP(id)

        message=request.form.get('message')

        url = f'http://{receiver\_ip}:{PORT}/receiveText?id={genIPList.MYID}&message={message}'

        try:

            response = requests.get(url, timeout=1)

            temp = response.text

            IPListAccess.addMessage(id, message, 'S')

            return temp

        except requests.RequestException as e:

            return f"Message cannot be delivered. The IP address {receiver\_ip} is not found or may be changed"

    except:

        return "There is some error in sending Message"

#API TO CAPTURE THE MESSAGE WHICH IS BEING SENT ON THIS IP AND MAKE AN APPROPRIATE RESPONCE

@app.route('/receiveText')

def receive():

    try:

        id = request.args.get('id')

        ip = request.remote\_addr

        message = request.args.get('message')

        IPListAccess.addIP(id, ip)

        IPListAccess.addMessage(id, message, 'R')

        MESSAGE.append(str(id))

        return "message\_received"

    except:

        return "Receiver cannot process the message"

#API FOR RECEIVING THE FILE FROM SENDER

@app.route('/receiveFile', methods=['POST'])

def receiveFile():

    try:

        file = request.files['file']

        id = str(request.form.get('id'))

        if(id=="None"  or file=="None"):

            return "File received in Invalid Format"

        file.save(os.path.join(app.config['UPLOAD\_FOLDER'], id+"\_\_"+file.filename))

        FILE.append(id)

        return "ACK from Receiver: File received succefully"

    except:

        return "ACK from Receiver: There is some error"

#ADDING IP TO LIST, API FOR ADDING ID AND IP TO ID-IP TABLE MANUALLY BY USER

@app.route('/addIP', methods=['POST'])

def addip():

    try:

        ip=request.form.get('ip')

        name=request.form.get('name')

        try:

            id=request.form.get('id')

            IPListAccess.addIP(id, ip, name)

            return "IP added. Now Scan to update to Latest IP"

        except :

            IPListAccess.addIP(id, "", name)

            return "IP added. Now Scan to update to Latest IP"

    except:

        return "Cannot Add IP"

#FOR NOTIFICATION

@app.route('/status', methods=['GET'])

def status(MESSAGE=MESSAGE, FILE=FILE):

    try:

        message=""

        file=""

        if(len(MESSAGE) == 0):

            message = " : "

        else:

            message = f"{MESSAGE[0]}:{IPListAccess.getName(MESSAGE[0])}"

        if(len(FILE) == 0):

            file = " : "

        else:

            file = f"{FILE[0]}:{IPListAccess.getName(FILE[0])}"

        MESSAGE.clear()

        FILE.clear()

        return message+":"+file

    except:

        MESSAGE = ""

        FILE = ""

        return "ERROR"

#RETURN THE TABLE WITH ID-IP LIST TO SHOW TO THE USER IN FRONTEND

@app.route('/getNameIdList', methods=['GET'])

def getList():

    try:

        name, id = IPListAccess.getNameIdList()

        dictio = dict()

        for i in range(0, len(name)):

            dictio.update({str(id[i]):str(name[i])})

        return str(dictio)

    except:

        return "Cannot fetch ID-IP list"

#SCANNING

def send\_request(i):

    url = f'http://{i}:{PORT}/respond?id={genIPList.MYID}'

    try:

        response = requests.get(url, timeout=2)

        if (response.text).isdigit() == True:

            IPListAccess.addIP(response.text, i)

            return True # Indicating a successful request and response

        return False  # Indicating a successful request but unsuccessful response

    except requests.RequestException:

        pass

    return False  # Indicating a failed request

@app.route('/scan', methods=['GET'])

def scan():

    try:

        ipList = genIPList.generate\_IP\_List()

        count = 0

        # Use ThreadPoolExecutor to run requests in parallel

        with ThreadPoolExecutor(max\_workers=200) as executor:

            futures = [executor.submit(send\_request, i) for i in ipList]

            for future in futures:

                if future.result():  # If the request was successful

                    count += 1

        return render\_template("index.html", MESSAGE=str(f"Scan Completed. {count} contacts Updated")), False

        # return f"Scan Completed\n{count} contacts Updated", False

    except:

        return "There is some Error during scanning!!!"

#API FOR RESPONDING DURING SCANNING PROCESS BY A SENDER DEVICE

@app.route('/respond', methods=['GET'])

def respond():

    senderID = request.args.get('id')

    senderIP = request.remote\_addr

    IPListAccess.addIP(senderID, senderIP)

    return str(genIPList.MYID)

@app.route('/getIP', methods=['GET'])

def getip():

    try:

        id = request.args.get('id')

        IP = IPListAccess.getIP(id)

        return IP

    except:

        return "there is some error"

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True, host='0.0.0.0', port=PORT)

**Generate\_IP\_List:-**

import psutil

import socket

import ipaddress

#Current User ID

MYID=123

def generate\_ip\_list(local\_ip, subnet\_mask):

    network = ipaddress.IPv4Network(f'{local\_ip}/{subnet\_mask}', strict=False)

    ip\_list = [str(ip) for ip in network.hosts()]

    ip\_list.remove(local\_ip)

    return ip\_list

def get\_subnet\_mask():

    hostname = socket.gethostname()

    local\_ip = socket.gethostbyname(hostname)

    for interface, addrs in psutil.net\_if\_addrs().items():

        for addr in addrs:

            if addr.address == local\_ip and addr.family == socket.AF\_INET:

                return addr.netmask

    return "Subnet mask not found."

def generate\_IP\_List():

    local\_ip = socket.gethostbyname(socket.gethostname())

    subnet\_mask = get\_subnet\_mask()

    return generate\_ip\_list(local\_ip, subnet\_mask)

**Database\_Access**

import pandas as pd

import random as rd

def getIP(id):

    df = pd.read\_csv("ipList.csv")

    if len(df)==0:

        return "NULL"

    return (df.loc[df['id'] == int(id), 'ip']).iloc[0]

def addIP(id, ip, name=str('contact'+str(rd.randint(1000,9999)))):

    id=int(id)

    df = pd.read\_csv("ipList.csv")

    if id in df['id'].values:

        df.loc[df['id'] == id, 'ip'] = ip

        df.to\_csv('ipList.csv', index=False)

        return

    new\_row = {'id': str(id), 'name' : name, 'ip': ip, 'message':"[]"}

    df = df.\_append(new\_row, ignore\_index=True)

    df.to\_csv('ipList.csv', index=False)

def getMessage(id):

    df = pd.read\_csv("ipList.csv")

    if len(df)==0:

        return "NULL"

    return (df.loc[df['id'] == int(id), 'message']).iloc[0]

def getName(id):

    df = pd.read\_csv("ipList.csv")

    if len(df)==0:

        return "NULL"

    return (df.loc[df['id'] == int(id), 'name']).iloc[0]

def addMessage(id, message, person): #person is sender or receiver

    id=int(id)

    df = pd.read\_csv("ipList.csv")

    if id in df['id'].values:

        lst=eval((df.loc[df['id'] == id, 'message'].iloc[0]))

        if person == "S":

            lst.append("@S:"+str(message))

        else:

            lst.append("@R:"+str(message))

        df.loc[df['id'] == id, 'message'] = str(lst)

        df.to\_csv('ipList.csv', index=False)

def getNameIdList():

    df = pd.read\_csv("ipList.csv")

    return (df['name']), df['id']

**Frontend (JS)**

//Function for introducting delay

        function delay(ms=500) {

            return new Promise(resolve => setTimeout(resolve, ms));

        }

        // Function to populate dropdown options

        async function populateDropdowns() {

            let response = await fetch("http://localhost:3000/getNameIdList");

            let data = await response.text()

            data=data.replace("{", " "); data=data.replace("}", "");

            console.log(data);

            data = data.split(",")

            let seemsg = document.querySelector("#see-message-id")

            let sendmsg = document.querySelector("#send-message-id")

            seemsg.innerHTML = "";

            sendmsg.innerHTML = "";

            for(let i=0; i<data.length; i++){

                let info = (data[i]).split(":");

                let e1 = document.createElement("option");

                let e2 = document.createElement("option");

                e1.setAttribute("value", info[0].slice(2, -1));

                e2.setAttribute("value", info[0].slice(2, -1));

                e1.innerHTML = info[1];

                e2.innerHTML = info[1];

                seemsg.appendChild(e1);

                sendmsg.appendChild(e2);

            }

        }

        // Function to show alert

        function showAlert(type, duration = 3000) {

            const successAlert = document.getElementById('successAlert');

            const errorAlert = document.getElementById('errorAlert');

            successAlert.style.display = 'none';

            errorAlert.style.display = 'none';

            if (type === 'success') {

                successAlert.style.display = 'block';

                setTimeout(() => {

                    successAlert.style.display = 'none';

                }, duration);

            } else if (type === 'error') {

                errorAlert.style.display = 'block';

                setTimeout(() => {

                    errorAlert.style.display = 'none';

                }, duration);

            }

        }

        //Function for sending file

        async function sendFile(){

            const fileInput = document.getElementById('file-input');

            const file = fileInput.files[0];

            if (!file) {

                document.getElementById('response-message').innerText = "Please select a file to upload.";

                return;

            }

            const formData = new FormData();

            formData.append('file', file);

            let id = document.querySelector("#sendFileID").value

            formData.append('id', id);

            try {

                document.getElementById('response-message').innerHTML = "Sending...";

                let IP = await fetch(`http://localhost:3000/getIP?id=${id}`)

                // console.log(IP.text())

                let response = await fetch(`http://${await IP.text()}:3000/receiveFile`, {

                    method: 'POST',

                    body: formData

                });

                response = await response.text()

                document.getElementById('response-message').innerHTML = response;

            }

            catch (error) {

                document.getElementById('response-message').innerText = `Network error: ${error.message} `;

            }

        }

        //function for Notification

        async function getStatus(){

            while(true){

                await delay(1000)

                try{

                    let response = await fetch(`http://localhost:3000/status`)

                    response = await response.text()

                    if(response == "ERROR") continue;

                    response = await response.split(":")

                    if(response[0]==" " && response[2]==" ") continue

                    let message = ""

                    let file = ""

                    if(response[0] != " "){

                        message = `Message Received From ID: ${response[0]}    Name: ${response[1]}`

                        document.querySelector("#message").innerHTML = message

                    }

                    if(response[2] != " "){

                        file = `file Received From ID: ${response[2]}    Name: ${response[3]}`

                        document.querySelector("#file").innerHTML = file

                    }

                    let elmntNoti = document.querySelector("#notification")

                    elmntNoti.style.display = "block"

                    await delay(5000)

                    elmntNoti.style.display = "none"

                }

                catch{

                    elmntNoti.style.display = "none"

                }

            }

        }

        getStatus()

        // Initialize the page

        document.addEventListener('DOMContentLoaded', async function () {

            await populateDropdowns();

        });

        document.querySelector('#scan-btn').addEventListener('click', function () {

            const spinnerContainer = document.getElementById('spinner-container');

            spinnerContainer.style.display = 'flex';  // Show spinner and blur background

        });

**RESULTS & INFERENCES:**

1. **Seamless Peer-to-Peer Communication**: The application successfully facilitates direct peer-to-peer communication between devices on the same local network. Users can send and receive text messages and files without routing through a centralized server, minimizing delays and enhancing data privacy.
2. **Reliable Device Detection and Network Scanning**: The IP scanning functionality consistently identified active devices on the local network. Through parallelized HTTP requests, the scanning process proved to be both efficient and responsive, accurately updating the database with the current IP addresses of detected devices.
3. **Real-Time Notification Service**: The notification feature was able to keep users updated on incoming messages or file transfers in near real-time. The asynchronous, non-blocking implementation in JavaScript allowed for uninterrupted user interactions.
4. **File Transfer**: The file sender and receiver functions successfully managed file transfers of various formats and sizes.
5. **Efficiency with Processor Programs**: By separating IP generation and database access to specialized processor programs, the application achieved a high level of efficiency. This modular approach separate computationally intensive tasks from the API handler, reducing latency and improving overall performance.
6. **Scalability and Modularity**: The modular design of the application, with separate roles for frontend, API handler, and processor programs, makes it scalable. Each component can be improved or extended independently, allowing new features or optimizations without affecting the overall system.
7. **Reduced Latency Through Decentralization**: By implementing a decentralized model where each user acts as both client and server, the application minimized data transfer times and network latency. This structure is particularly effective in closed network environments like local networks where real-time, peer-to-peer interactions are crucial

**APPLICATION ORIENTED LEARNING:**

**Real-Time Applications:**

* Local Network Communication: This application is essential in places such as universities, offices, or residential places where people are required to communicate with each other through their devices without having access to the internet and relying on other forms of communication.
* File Sharing in Closed Networks: It can be used to transport files between computers present in an enclosed space without the need for cloud services and other file repositories.
* User-Friendly Device Discovery: The scanning option allow users to find connected peripherals efficiently in IoT systems or Intranet server standing in certain locations.
* Client-Server Architecture: Sensor fusion, continuous client and server roles and shifting of tasks between devices based on the context.
* HTTP Protocols: Application of the HTTP GET and POST methods in sending messages and files.
* Asynchronous Programming: In Javascript multidimensional array is used in defining and creating separate chunks using asynchronous functions to provide call back function within the main executing thread.
* Multithreading: To achieve faster device discovery, a network scanning operation was parallelized to utilize multithreading or simultaneous operations to enhance the process.
* Peer-to-Peer Networking: Familiarity with non-centralized networks to minimize time wastage and protect potential sensitive information.

**CONCLUSION:**

The project achieved its objective of designing a local area network with a server-less communication network whereby the users get to connect in real-time and exchange information. The project is divided into three components, Frontend, API handler, and processor, which allows for both development and application of the project in a fast, flexible and scalable way. Despite these advantages, a number of issues were experienced in due course of developing the project. For instance, sometimes in the course of the project, active devices erroneously failed to be detected in the course of IP scanning, which could only be rectified by restarting the server. Moreover, issues like unencrypted message and file communication raise challenges that need to be addressed in the next iterations of the project. Future work could imply encryption mechanism enforcement, improved device verification mechanism or extending the system to offer services to a wider area than just local areas. The present edition is still in the pilot stage but in its structure, there are promising aspects that can be harnessed for practical purposes after some modifications.

**REFERENCES:**

* HTTP Methods: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>
* Asynchronous JavaScript: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous>
* Multithreading in Network Programming: <https://www.geeksforgeeks.org/multithreading-in-java/>
* IP Addressing and Subnetting: <https://www.cisco.com/c/en/us/support/docs/ip/routing-information-protocol-rip/13788-3.html>