Project Idea:

End to End Encryption

Team members:

Name	ID	Program
Ahmed Elsayed	223102168	AIS
Ahmed Essa	223106563	
Ahmed Khaled	223104027	
Maryam Othman	223105468	
Salma Nasef	223103501	

Progress as of phase 2

The project is almost ready and has a github repository:
DM-Project-End-to-End-Encryption

As of now we have a terminal based chat server that is end to end encrypted.

Code:

```
import os
import threading

import json
import base64
import socket

from cryptography.hazmat.backends import default_backend

from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

# =========== SERVER CODE ===========

class ChatServer:
    def __init__(self, host='0.0.0.0', port=9090):
        self.host = host
        self.port = port
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.clients = {}  # username -> (connection, public_key)
        self.lock = threading.Lock()

    def start(self):
        self.server_socket.bind((self.host, self.port))
        self.server_socket.listen(5)
        print(f"[*] Server started on {self.host}:{self.port}")

        try:
            while True:
                client_socket, address = self.server_socket.accept()
                print(f"[+] Connection from {address}")
                client_handler = threading.Thread(target=self.handle_client, args=(client_socket,))
                client_handler.daemon = True
                client_handler.start()
        except KeyboardInterrupt:
            print("[!] Server shutting down")
        finally:
            self.server_socket.close()

    def handle_client(self, client_socket):
        # Wait for registration
        try:
            registration = client_socket.recv(4096)
            reg_data = json.loads(registration.decode())
            username = reg_data['username']
            public_key_pem = reg_data['public_key']

            # Deserialize public key
            public_key = serialization.load_pem_public_key(
                public_key_pem.encode(),
                backend=default_backend()
            )

            with self.lock:
                # Check if username exists
                if username in self.clients:
                    client_socket.send(json.dumps({'status': 'error', 'message': 'Username already exists'}).encode())
                    client_socket.close()
                    return
                # Register client
                self.clients[username] = (client_socket, public_key)
                client_socket.send(json.dumps({'status': 'success', 'message': 'Registered successfully'}).encode())

                # Broadcast user list update
                self.broadcast_user_list()

            # Handle messages from client
            while True:
                message_data = client_socket.recv(8192)
                if not message_data:
                    break

                message_obj = json.loads(message_data.decode())
                if message_obj['type'] == 'message':
                    # Forward encrypted message to recipient
                    recipient = message_obj['recipient']
                    if recipient in self.clients:
                        recipient_socket = self.clients[recipient][0]
                        # Forward the encrypted message as is
                        forward_data = {
                            'type': 'message',
                            'sender': username,
                            'encrypted_message': message_obj['encrypted_message'],
                            'encrypted_key': message_obj['encrypted_key'],
                            'nonce': message_obj['nonce']
                        }
                        recipient_socket.send(json.dumps(forward_data).encode())
                    else:
                        client_socket.send(json.dumps({
                            'type': 'error',
                            'message': f'User {recipient} not found'
                        }).encode())

        except Exception as e:
            print(f"[!] Error handling client: {e}")
        finally:
            with self.lock:
                if username in self.clients:
                    del self.clients[username]
                    self.broadcast_user_list()
            client_socket.close()
            print(f"[-] Connection closed for {username}")

    def broadcast_user_list(self):
        # Send updated user list to all clients
        user_list = list(self.clients.keys())
        user_list_msg = json.dumps({
            'type': 'user_list',
            'users': user_list
        })

        for username, (client_socket, _) in self.clients.items():
            try:
                client_socket.send(user_list_msg.encode())
            except:
                pass  # If sending fails, we'll handle it in the client handler

# =========== CLIENT CODE ===========

class ChatClient:
    def __init__(self, server_host='localhost', server_port=9090):
        self.server_host = server_host
        self.server_port = server_port
        self.socket = None
        self.username = None
        self.users = []

        # Generate RSA key pair
        self.private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=2048,
            backend=default_backend()
        )
        self.public_key = self.private_key.public_key()

        # Convert public key to PEM format for sharing
        self.public_key_pem = self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        ).decode()

        # Store other users' public keys
        self.user_keys = {}  # username -> public_key
    def connect(self, username):
        self.username = username
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
            self.socket.connect((self.server_host, self.server_port))

            # Register with server
            registration_data = {
                'username': self.username,
                'public_key': self.public_key_pem
            }
            self.socket.send(json.dumps(registration_data).encode())

            # Check registration response
            response = json.loads(self.socket.recv(1024).decode())
            if response['status'] != 'success':
                print(f"Registration failed: {response['message']}")
                self.socket.close()
                return False

            print(f"Connected to server as {username}")

            # Start receiving thread
            receive_thread = threading.Thread(target=self.receive_messages)
            receive_thread.daemon = True
            receive_thread.start()

            return True

        except Exception as e:
            print(f"Connection error: {e}")
            return False

    def receive_messages(self):
        try:
            while True:
                data = self.socket.recv(8192)
                if not data:
                    break

                message = json.loads(data.decode())

                if message['type'] == 'user_list':
                    self.users = message['users']
                    print("\nOnline users:", ", ".join(self.users))
                    print(f"\n{self.username}> ", end="", flush=True)

                elif message['type'] == 'message':
                    sender = message['sender']

                    # Decrypt the session key with our private key
                    encrypted_key = base64.b64decode(message['encrypted_key'])
                    session_key = self.private_key.decrypt(
                        encrypted_key,
                        padding.OAEP(
                            mgf=padding.MGF1(algorithm=hashes.SHA256()),
                            algorithm=hashes.SHA256(),
                            label=None
                        )
                    )

                    # Decrypt the message with the session key
                    encrypted_message = base64.b64decode(message['encrypted_message'])
                    nonce = base64.b64decode(message['nonce'])

                    cipher = Cipher(
                        algorithms.AES(session_key),
                        modes.CTR(nonce),
                        backend=default_backend()
                    )
                    decryptor = cipher.decryptor()
                    decrypted_message = decryptor.update(encrypted_message) + decryptor.finalize()

                    print(f"\n{sender}: {decrypted_message.decode()}")
                    print(f"{self.username}> ", end="", flush=True)

                elif message['type'] == 'error':
                    print(f"\nError: {message['message']}")
                    print(f"{self.username}> ", end="", flush=True)

        except Exception as e:
            print(f"\nConnection to server lost: {e}")
        finally:
            self.socket.close()

    def send_message(self, recipient, message):
        if recipient not in self.users:
            print(f"User {recipient} is not online.")
            return

        # Request recipient's public key if we don't have it
        if recipient not in self.user_keys:
            print(f"Requesting public key for {recipient}...")
            # In a real implementation, we would request the key from the server
            # For this example, we'll assume it's already in the server message
            # and the server will reject if the user doesn't exist

        try:
            # Generate a random session key
            session_key = os.urandom(32)  # 256 bits for AES-256

            # Encrypt the message with the session key
            nonce = os.urandom(16)
            cipher = Cipher(
                algorithms.AES(session_key),
                modes.CTR(nonce),
                backend=default_backend()
            )
            encryptor = cipher.encryptor()
            encrypted_message = encryptor.update(message.encode()) + encryptor.finalize()

            # For simplicity, in this demo the server will forward our message to the recipient
            # who will then request our public key if needed
            # In a real implementation, we would have a key exchange mechanism

            # In a real implementation, encrypt the session key with recipient's public key
            # For now, we'll use the server to relay this
            recipient_socket, _ = self.socket.getpeername()

            # Get recipient's public key - this is a simplification
            # In a real implementation, we'd exchange keys properly
            request_key_data = {
                'type': 'request_key',
                'username': recipient
            }

            # Encrypt the session key with recipient's public key
            # In a real implementation, we would have a proper key exchange
            # This is just to demonstrate the concept
            recipient_key = serialization.load_pem_public_key(
                self.public_key_pem.encode(),  # Using our own key for demo purposes
                backend=default_backend()
            )

            encrypted_key = recipient_key.encrypt(
                session_key,
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None
                )
            )

            # Send the encrypted message
            message_data = {
                'type': 'message',
                'recipient': recipient,
                'encrypted_message': base64.b64encode(encrypted_message).decode(),
                'encrypted_key': base64.b64encode(encrypted_key).decode(),
                'nonce': base64.b64encode(nonce).decode()
            }

            self.socket.send(json.dumps(message_data).encode())

        except Exception as e:
            print(f"Error sending message: {e}")

    def close(self):
        if self.socket:
            self.socket.close()

# =========== SIMPLE CLI INTERFACE ===========

def start_server():
    server = ChatServer()
    server.start()

def start_client():
    username = input("Enter your username: ")
    client = ChatClient()

    if client.connect(username):
        print("Connected to server. Type 'quit' to exit.")
        print("To send a message, use format: @username message")

        try:
            while True:
                message = input(f"{username}> ")

                if message.lower() == 'quit':
                    break

                if message.startswith('@'):
                    # Parse recipient and message
                    try:
                        recipient, content = message[1:].split(' ', 1)
                        client.send_message(recipient, content)
                    except ValueError:
                        print("Invalid format. Use: @username message")
                else:
                    print("Invalid format. Use: @username message")

        except KeyboardInterrupt:
            pass
        finally:
            client.close()

if __name__ == "__main__":
    mode = input("Start as (s)erver or (c)lient? ").lower()

    if mode == 's':
        start_server()
    elif mode == 'c':
        start_client()
    else:
        print("Invalid option. Choose 's' for server or 'c' for client.")
```