```
# 2024 Â© Idan Hazay

from modules import client_requests, networking, protocol
from modules.config import *
from modules.errors import Errors
from modules.logger import Logger

import socket, traceback, time, threading, sys
from requests import get

class Application:
    """
    Main server application handling client connections, requests, and general server functionality.
    """
    def __init__(self, addr):
        self.clients = {}
        self.bytes_recieved = {}
        self.bytes_sent = {}
        self.files_uploading = {}
        self.all_to_die = False
        self.network = networking.Network(self.clients, self.bytes_recieved, self.bytes_sent)
        self.cr = client_requests.ClientRequests()
        self.protocol = protocol.Protocol(self.network, self.clients, self.cr, self.files_uploading)
        self.addr = addr
        self.start()

    def start(self):
        """
        Start the server, listen for client connections, and manage threads for each client.
        """
        threads = []
        self.srv_sock = socket.socket()  # Server socket initialization
        self.srv_sock.bind(self.addr)  # Bind the server to the provided address
        self.srv_sock.listen(20)

        print(f"Server listening on {self.addr}")

        try:
            self.public_ip = get('https://api.ipify.org').content.decode('utf8')  # Fetch public IP
        except Exception:
            self.public_ip = "No IP found"

        try:
            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
                s.connect(("8.8.8.8", 80))  # Google's DNS server for local IP discovery
                self.local_ip = s.getsockname()[0]
        except:
            self.local_ip = "127.0.0.1"

        print(f"Public server ip: {self.public_ip}, local server ip: {self.local_ip}")

        self.srv_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # Enable address reuse
        i = 1

        try:
            self.network.encryption.create_keys()  # Encryption key generation
            self.network.encryption.load_keys()
            scheduler = threading.Thread(target=self.cleaner)  # Start cleanup process
            scheduler.start()
        except:
            print(traceback.format_exc())
            self.srv_sock.close()
            return

        dhcp_listener = threading.Thread(target=self.network.dhcp_listen, args=(self.local_ip, self.addr[1]))  # DHCP
listening thread
        dhcp_listener.start()

        print('Main thread: before accepting ...\n')
        while True:
            cli_sock, addr = self.srv_sock.accept()  # Accept incoming client connection
            t = threading.Thread(target=self.handle_client, args=(cli_sock, str(i), addr))
            t.start()  # Start client thread
            i += 1
            threads.append(t)
            if i > 100000000:
                print('\nMain thread: going down for maintenance')
                break

        self.all_to_die = True  # Stop all client threads
        print('Main thread: waiting to all clients to die')
        for t in threads:
            t.join()  # Ensure all threads finish

        self.srv_sock.close()
        print('Bye ..')
    def handle_client(self, sock, tid, addr):
        """
        Handle an individual client connection, initialize secure communication, and process client requests.
        """
        try:
            finish = False
            print(f'New Client number {tid} from {addr}')
            self.bytes_sent[tid] = 0
            self.bytes_recieved[tid] = 0
            start = self.network.recv_data(sock, tid)  # Receive initial client data
            code = start.split(b"|")[0]

            self.clients[tid] = Client(tid, "guest", "guest", 0, 0, None, False)  # Initialize client with guest role

            if code == b"RSAR":
                shared_secret = self.network.encryption.rsa_exchange(sock, tid)  # RSA key exchange
            if shared_secret == "":
                return

            self.clients[tid].shared_secret = shared_secret
            self.clients[tid].encryption = True  # Mark client as encrypted

        except Exception:
            print(traceback.format_exc())
            print(f'Client {tid} connection error')
            if tid in self.clients:
                self.clients[tid] = None  # Remove problematic client
            sock.close()
            return

        while not finish and self.clients[tid] is not None:
            if self.all_to_die:
                print('Will close due to main server issue')
                break
            try:
                entire_data = self.network.recv_data(sock, tid)  # Read client data
                t = threading.Thread(target=self.handle_request, args=(entire_data, tid, sock))
                t.start()

            except socket.error as err:
                print(f'Socket Error exit client loop: err: {err}')
                break
            except Exception as err:
                print(f'General Error: {err}')
                print(traceback.format_exc())
                break

        print(f'Client {tid} Exit')
        self.clients[tid] = None  # Mark client as disconnected
        sock.close()

    def handle_request(self, request, tid, sock):
        """
        Parse and handle a client request, sending appropriate responses.
        """
        try:
            to_send = self.protocol.protocol_build_reply(request, tid, sock)  # Build a response for the client
            if to_send is None:
                self.clients[tid] = None  # Mark client as disconnected
                print(f"Client {tid} disconnected")
                return
            to_send = to_send.encode()
            self.network.send_data(sock, tid, to_send)  # Send data back to client

            if to_send == b"EXTR":
                self.clients[tid] = None  # Disconnect client explicitly
                print(f"Client {tid} disconnected")

        except Exception:
            print(traceback.format_exc())
            to_send = Errors.GENERAL.value  # Fallback error response
            self.network.send_data(sock, tid, to_send.encode())

    def cleaner(self):
        """
        Periodically clean up database entries for ongoing file uploads.
        """
        while True:
            self.cr.clean_db(self.files_uploading)  # Remove old or invalid uploads
            time.sleep(100)  # Wait between cleanup operations

class Client:
    """
    Client class for managing individual client states.
    """
    def __init__(self, id, user, email, subscription_level, admin_level, shared_secret, encryption):
        self.id = id
        self.user = user
        self.email = email
        self.subscription_level = subscription_level
        self.admin_level = admin_level
        self.shared_secret = shared_secret
        self.encryption = encryption
        self.cwd = f"{CLOUD_PATH}\\{self.user}"

def main(addr):
    """
    Main entry point to initialize and run the server application.
    """
    app = Application(addr)

if __name__ == '__main__':
    sys.stdout = Logger()
    port = 3102
    if len(sys.argv) == 2:
        port = sys.argv[1]
    main(("0.0.0.0", int(port)))
```