```python
# 2024 Â© Idan Hazay networking.py
# Import libraries

from modules import encrypting
from modules.config import *
import struct, socket, psutil, time


class Network:
    """Handles network communication, including encryption, data transfer, and server discovery."""
    def __init__(self, log=False):
        self.log = log
        self.encryption = encrypting.Encryption(self)

    def set_sock(self, socket):
        """Assigns a socket for communication."""
        self.sock = socket

    def set_secret(self, secret):
        """Stores the shared encryption key."""
        self.shared_secret = secret

    def reset_network(self):
        """Resets encryption and clears socket and encryption key."""
        self.encryption = encrypting.Encryption(self)
        self.sock = None
        self.shared_secret = None

    def logtcp(self, dir, byte_data):
        """Logs network data transmission and reception."""
        if self.log:
            try:
                if str(byte_data[0]) == "0":
                    print("")
            except AttributeError:
                return
            if dir == 'sent':
                print(f'C LOG:Sent     >>>{byte_data}')
            else:
                print(f'C LOG:Recieved <<< {byte_data}')

    def send_data_wrap(self, bdata, encryption):
        """Sends data to the server with optional encryption."""
        if encryption:
            encrypted_data = self.encryption.encrypt(bdata, self.shared_secret)
            data_len = struct.pack('!l', len(encrypted_data))
            to_send = data_len + encrypted_data
            to_send_decrypted = str(len(bdata)).encode() + bdata  # Log decrypted data
            self.logtcp('sent', to_send)
            self.logtcp('sent', to_send_decrypted)
        else:
            data_len = struct.pack('!l', len(bdata))
            to_send = data_len + bdata
            self.logtcp('sent', to_send)

        self.sock.send(to_send)

    def recv_data(self, encryption=True):
        """Receives data from the server, decrypting if necessary."""
        try:
            b_len = b''
            while len(b_len) < LEN_FIELD:  # Ensure full length field is received
                b_len += self.sock.recv(LEN_FIELD - len(b_len))

            msg_len = struct.unpack("!l", b_len)[0]
            if msg_len == b'':
                print('Seems client disconnected')
            msg = b''
            while len(msg) < msg_len:  # Ensure full message is received
                chunk = self.sock.recv(msg_len - len(msg))
                if not chunk:
                    print('Server disconnected abnormally.')
                    break
                msg += chunk

            if encryption:
                self.logtcp('recv', b_len + msg)  # Log encrypted data
                msg = self.encryption.decrypt(msg, self.shared_secret)
                self.logtcp('recv', str(msg_len).encode() + msg)

            return msg
        except ConnectionResetError:
            return None
        except OSError:
            pass
        except AttributeError:
            pass
```

```
    except:
        print(traceback.format_exc())

@staticmethod
def get_broadcast_address(ip, netmask):
    """Calculates the broadcast address for the given IP and netmask."""
    ip_binary = struct.unpack('>I', socket.inet_aton(ip))[0] # making ip binary
    netmask_binary = struct.unpack('>I', socket.inet_aton(netmask))[0] # making netmask binary
    broadcast_binary = ip_binary | ~netmask_binary & 0xFFFFFFFF # performing an or between ip and inverted netmask
    return socket.inet_ntoa(struct.pack('>I', broadcast_binary)) # making the result into an ip again

@staticmethod
def get_subnet_mask():
    """Finds the subnet mask and local IP address of the active network interface."""
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.connect(("8.8.8.8", 80))  # Connect to Google's DNS to determine active interface
        current_ip = s.getsockname()[0]

    addrs = psutil.net_if_addrs()
    stats = psutil.net_if_stats()

    for interface, addrs_list in addrs.items():
        if stats[interface].isup:  # Ensure the interface is active
            for addr in addrs_list:
                if addr.family == socket.AF_INET and addr.address == current_ip:
                    return addr.netmask, addr.address

    return None  # No active interface found

def search_server(self):
    """Broadcasts a search request to locate an available server on the network."""
    try:
        search_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
        search_socket.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
        search_socket.settimeout(SOCK_TIMEOUT)

        netmask, ip = self.get_subnet_mask()  # Get subnet information
        broadcast_address = self.get_broadcast_address(ip, netmask)
        search_socket.sendto(b"SEAR", (broadcast_address, 31026))  # Broadcast search request

        response, addr = search_socket.recvfrom(1024)
        response = response.decode().split("|")
        if response[0] == "SERR":
            ip, port = response[1], response[2]
            return ip, int(port)

    except TimeoutError:
        print("No server found")
        return SAVED_IP, SAVED_PORT
    except:
        print(traceback.format_exc())
        return SAVED_IP, SAVED_PORT
```