A Simple Peer To Peer Network Implementation

Hoora Abootalebi
Nariman Aryan
Amin Isaai
Amirhossein Khajepour
Mahdis Tajdari
Ali Zeynali

November 2018

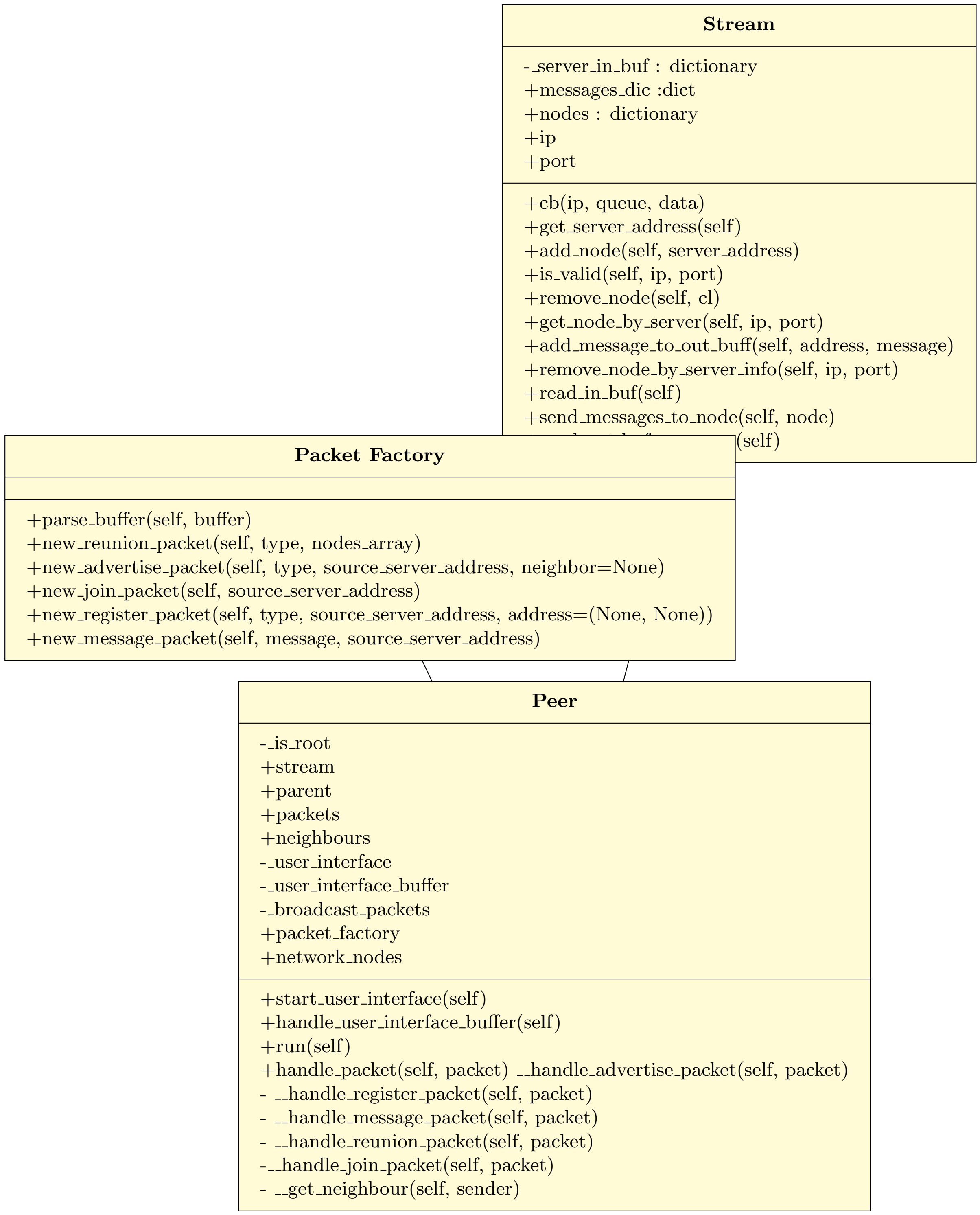
Contents

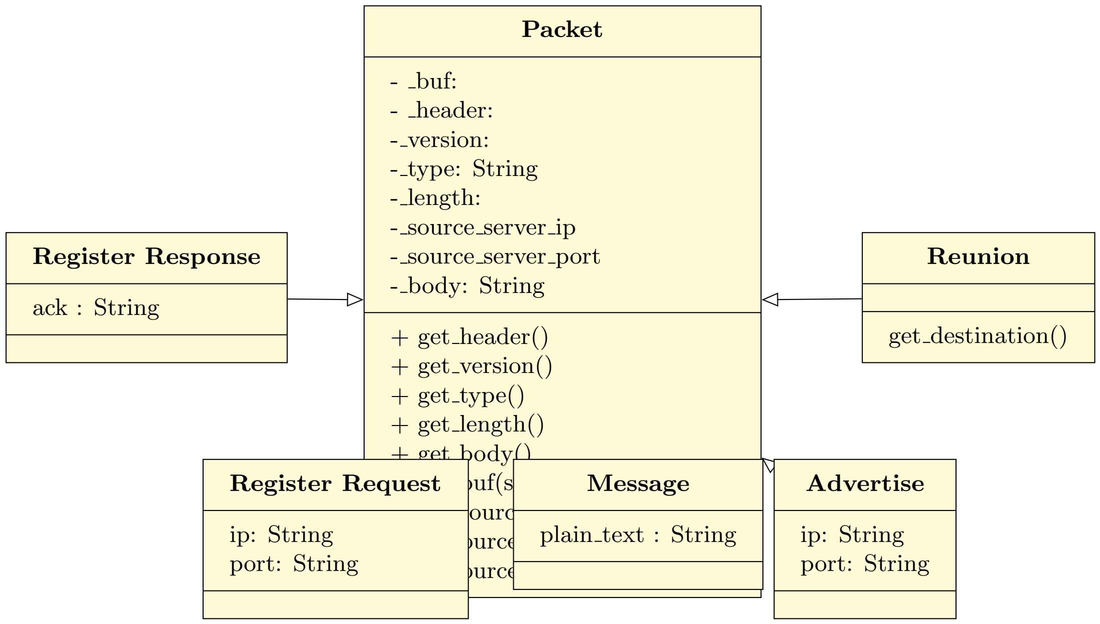
1 Introdeuction

This project aims to implement a peer to peer network. In the first step we design UML model and then we are going to explain each objects' attributes and methods.

2 UML model

We design the UML model in order to make the project more understandable, clearer and professional.


Stream
-_server_in_buf : dictionary
+messages_dic :dict
+nodes : dictionary
+ip
+port
+cb(ip, queue, data)
+get_server_address(self)
+add_node(self, server_address)
+is_valid(self, ip, port)
+remove_node(self, cl)
+get_node_by_server(self, ip, port)
+add_message_to_out_buff(self, address, message)
+remove_node_by_server_info(self, ip, port)
+read_in_buf(self)
+send_messages_to_node(self, node)
Packet Factory
+parse_buffer(self, buffer)
+new_reunion_packet(self, type, nodes_array)
+new_advertise_packet(self, type, source_server_address, neighbor=None)
+new_join_packet(self, source_server_address)
+new_register_packet(self, type, source_server_address, address=(None, None))
+new_message_packet(self, message, source_server_address)
Peer
-_is_root
+stream
+parent
+packets
+neighbours
-_user_interface
-_user_interface_buffer
-_broadcast_packets
+packet_factory
+network_nodes
+start_user_interface(self)
+handle_user_interface_buffer(self)
+run(self)
+handle_packet(self, packet) __handle_advertise_packet(self, packet)
- __handle_register_packet(self, packet)
- __handle_message_packet(self, packet)
- __handle_reunion_packet(self, packet)
-__handle_join_packet(self, packet)
- __get_neighbour(self, sender)



3 Objects

Now it's time to explain every obejct's duty.

3.1 Stream

This objects has one server and n clients. Servers are always open for reading and clients will be open whenever we want to write on a socket.
There is a clientMsg dictionary in this object to handle messages. This means that there is an array assigned to a specific client for all clients. If we need to add a new client to this object, we use add_-client() method. Consequently, an array will be assigned to this new client in clientMsg.
There is also a remove_client() method for the times when we want to remove a client from this object. This method is mostly used when reunion fails.
The read_in_buf returns the buffer of the server.
The send_msg() method is used when peer wants to send a message to a specific client.
Byte_ack() is used to reply to the receives messages. We must reply

all of the receives messages by sending Ack (which is a string).

```
#stream()
    def __init__(self, ip, port):
        """
        :param ip: 15 characters
        :param port: 5 characters
        """
        if not self.is_valid(ip, port):
            raise Exception("Invalid format of ip or port for TCPServer.")
            #   TODO    Error handling

        self.messages_dic = {}
        self._server_in_buf = []
        # self.parent = None
        #   TODO    Parent should be in Peer object not here

        def cb(ip, queue, data):
            queue.put(bytes('ACK', 'utf8'))
            print("In callback: ", data)
            # self.messages_dic.update({ip:
                 self.messages_dic.get(ip).append(data)})
            self._server_in_buf.append(data)

        print("Binding server: ", ip, ": ", port)
        self._server = TCPServer(ip, port, cb)
        self._server.run()
        self.nodes = []
        self.ip = ip
        self.port = port

    def get_server_address(self):
        return Node.parse_ip(self._server.ip),
             Node.parse_port(self._server.port)

    def add_node(self, server_address):
        self.nodes.append(Node(server_address))

    def is_valid(self, ip, port):
        if len(str(ip)) != 15 or len(str(port)) > 5:
            return False
        return True

    def remove_node(self, cl):
        self.nodes.remove(cl)
        cl.close()

    def get_node_by_server(self, ip, port):
```

```
    """
    :param ip:
    :param port:
    :return:
    :rtype: Node
    """
    port = Node.parse_port(port)
    ip = Node.parse_ip(ip)
    for nd in self.nodes:
        if nd.get_server_address()[0] == ip and
            nd.get_server_address()[1] == port:
            return nd

def add_message_to_out_buff(self, address, message):
    print("add message to out buff: ", address, " ", message)
    n = self.get_node_by_server(address[0], address[1])
    # if n is None:
    #     n = self.get_node_by_client(address[0], address[1])
    if n is None:
        raise Exception("Unexpected address to add message to out
            buffer.")

    n.add_message_to_out_buff(message)

def remove_node_by_server_info(self, ip, port):
    rem_client = None
    for nd in self.nodes:
        if nd.get_server_address[0] == ip and
            nd.get_server_address[1] == port:
            rem_client = nd
            break
    if rem_client is not None:
        self.remove_node(rem_client)

def read_in_buf(self):
    return self._server_in_buf

def send_messages_to_node(self, node):
    """
    Send buffered messages to the 'node'
    :param node:
    :type node Node
    :return:
    """

    response = node.send_message()

def send_out_buf_messages(self):
    """
    In this function we will send hole out buffers to their own
```

```
            clients.
        :return:
        """

        for n in self.nodes:
            self.send_messages_to_node(n) def __init__(self, ip, port):
        """
        :param ip: 15 characters
        :param port: 5 characters
        """
        if not self.is_valid(ip, port):
            raise Exception("Invalid format of ip or port for TCPServer.")
            #   TODO   Error handling

        self.messages_dic = {}
        self._server_in_buf = []
        # self.parent = None
        #   TODO   Parent should be in Peer object not here

        def cb(ip, queue, data):
            queue.put(bytes('ACK', 'utf8'))
            print("In callback: ", data)
            # self.messages_dic.update({ip:
                self.messages_dic.get(ip).append(data)})
            self._server_in_buf.append(data)

        print("Binding server: ", ip, ": ", port)
        self._server = TCPServer(ip, port, cb)
        self._server.run()
        self.nodes = []
        self.ip = ip
        self.port = port

    def get_server_address(self):
        return Node.parse_ip(self._server.ip),
            Node.parse_port(self._server.port)

    def add_node(self, server_address):
        self.nodes.append(Node(server_address))

    def is_valid(self, ip, port):
        if len(str(ip)) != 15 or len(str(port)) > 5:
            return False
        return True

    def remove_node(self, cl):
        self.nodes.remove(cl)
        cl.close()

    def get_node_by_server(self, ip, port):
        """
        :param ip:
        :param port:
        :return:
        :rtype: Node
        """
        port = Node.parse_port(port)
        ip = Node.parse_ip(ip)
        for nd in self.nodes:
            if nd.get_server_address()[0] == ip and
                nd.get_server_address()[1] == port:
                return nd

    def add_message_to_out_buff(self, address, message):
        print("add message to out buff: ", address, " ", message)
        n = self.get_node_by_server(address[0], address[1])
        # if n is None:
        #     n = self.get_node_by_client(address[0], address[1])
        if n is None:
            raise Exception("Unexpected address to add message to out
                buffer.")

        n.add_message_to_out_buff(message)

    def remove_node_by_server_info(self, ip, port):
        rem_client = None
        for nd in self.nodes:
            if nd.get_server_address[0] == ip and
                nd.get_server_address[1] == port:
                rem_client = nd
                break
        if rem_client is not None:
            self.remove_node(rem_client)

    def read_in_buf(self):
        return self._server_in_buf

    def send_messages_to_node(self, node):
        """
        Send buffered messages to the 'node'
        :param node:
        :type node Node
        :return:
        """

        response = node.send_message()

    def send_out_buf_messages(self):
        """
        In this function we will send hole out buffers to their own
            clients.
        :return:
        """

        for n in self.nodes:
            self.send_messages_to_node(n)
```

We also need to add a dictionary to specify every client's message(s)

3.2 Peer

This object is the main object that we are working with. It must have a Stream object which provides the connection to the socket; This means that reading and writing are done using the Stream object. Peer must also have a userInterface object in order to facilitate commanding by users.(.e.g. for connecting or sending message to a specific node)
The run method handles all of the events included in stream in an infinite loop and it also handles the received messages; This means that it does a certain action based on the type of the received packet. The handle_packet function is a wrapper for doing each packets action; this function uses several internal functions that implemented for each type of packets as listed below:

1. handle_advertise_packet
2. handle_reunion_packet
3. handle_register_packet
4. handle_message_packet

```
#Peer()
stream()
user_interface() #Which the user or client sees and works with.
run()            #This method runs every time to see
                #whether there is new messages or not.
packetFactory()
handle_packets()
```

3.3 Packet Factory

packetFactory() The main functionality of this object is to create different types of packet and return them. To be more specific, we read data from buffer and we pass it through pars_buff() method in order to get a packet.
In addition this object has some methods to create our specified packet type like: advertise, reunion, register, message; for each packet type we have a separate function.

```
#packetFactory
 def parse_buffer(self, buffer):

        """
        :param buffer: The buffer that should be parse to a validate
            packet format
        :return new packet
        :rtype Packet
        """

        return Packet(buf=buffer)

    def new_reunion_packet(self, type, nodes_array):
        """
        :param destination: (ip, port) of destination want to send
            reunion packet.
        :param nodes_array: [(ip0, port0), (ip1, port1), ...] It is the
            path to the 'destination'.
        :return New reunion packet.
        :rtype Packet
        """
        version = '1'
        packet_type = '05'
        if type == 'REQ':
            body = 'REQ'
        elif type == 'RES':
            body = 'RES'
        else:
            return None
        number_of_entity = str(len(nodes_array))
        if len(number_of_entity) < 2:
            number_of_entity = '0' + number_of_entity
        body = body + number_of_entity
        for (ip, port) in nodes_array:
            body = body + ip
            body = body + port
        length = str(len(body))
        while len(length) < 5:
            length = '0' + length
        return Packet(version + packet_type + length + body)

    def new_advertise_packet(self, type, source_server_address,
         neighbor=None):
        """
        :param type: Type of Advertise packet
        :param source_server_address Server address of the packet sender.
        :param neighbor: The neighbor for advertise response packet; The
            format is like ('192.168.001.001', '05335').
        :type type: str
        :type source_server_address: tuple
        :type neighbor: tuple
        :return New advertise packet.
        :rtype Packet
        """
        print("Creating advertisement packet")
        version = '1'
        packet_type = '02'

        if type == 'REQ':
            body = 'REQ'
            length = '00003'
            print("Request adv packtet created")
            return Packet(version + packet_type + length +
                source_server_address[0] + source_server_address[1] +
                body)

        elif type == 'RES':
            try:
                body = 'RES'
                body += neighbor[0]
                body += neighbor[1]
                length = '00023'
                print("Response adv packtet created")
                return Packet(
                    version + packet_type + length +
                        source_server_address[0] +
                        source_server_address[1] + body)
            except Exception as e:
                print(str(e))
        else:
            raise Exception("Type is incorrect")

    def new_join_packet(self, source_server_address):
        """
        :param source_server_address: Server address of the packet
            sender.
        :type source_server_address: tuple
        :return New join packet.
        :rtype Packet
        """
        print("Creating join packet")
        version = '1'
        packet_type = '03'
        length = '00004'
        body = 'JOIN'

        return Packet(version + packet_type + length +
            source_server_address[0] + source_server_address[1] + body)

    def new_register_packet(self, type, source_server_address,
         address=(None, None)):
        """
        :param type: Type of Register packet
        :param source_server_address: Server address of the packet
            sender.
        :param address: If type is request we need address; The format
            is like ('192.168.001.001', '05335').
        :type type: str
        :type source_server_address: tuple
        :type address: tuple
        :return New Register packet.
        :rtype Packet
        """
        print("Creating register packet")
        version = "1"
        packet_type = "01"

        if type == "REQ":
            length = "00023"
            body = "REQ" + '.'.join(str(int(part)).zfill(3) for part in
                address[0].split('.')) + \
                  str(address[1]).zfill(5)
            print("Request register packet created")
        elif type == "RES":
            length = "00006"
            body = "RES"
            print("Response register packet created")
        else:
            raise Exception("Irregular register type.")

        return Packet(version + packet_type + length +
            source_server_address[0] + source_server_address[1] + body)

        pass

    def new_message_packet(self, message, source_server_address):
        """
```

```
        Packet for sending a broadcast message to hole network.
        :param message: Our message
        :param source_server_address: Server address of the packet
            sender.
        :type message: str
        :type source_server_address: tuple
        :return: New Message packet.
        :rtype: Packet
        """
        version = '1'
        packet_type = '04'
        body = message
        length = len(message)
        for i in range(length, 5):
            length = '0' + length
        print("Message packet created")
        return Packet(version + packet_type + length +
            source_server_address[0] + source_server_address[1] + body)
```

3.4 Packet

Every packet consists seven differntes parts: **plain_text** which is the raw text message in the packet.

```
#Packet
def __init__(self, buf):
        self._buf = buf
        self._header = buf[0:28]
        self._version = int(buf[0], 10)
        self._type = int(buf[1:3], 10)
        self._length = int(buf[3:8], 10)
        self._source_server_ip = buf[8:23]
        self._source_server_port = buf[23:28]
        self._body = buf[8:]

    def get_header(self):
        """
        :return: Packet header
        :rtype: str
        """

        return self._header

    def get_version(self):
        """
        :return: Packet Version
        :rtype: int
        """
        return self._version

    def get_type(self):
        """
        :return: Packet type
        :rtype: int
        """
        return self._type

    def get_length(self):
        """
        :return: Packet length
        :rtype: int
        """
        return self._length

    def get_body(self):
        """
        :return: Packet body
        :rtype: str
        """
        return self._body

    def get_buf(self):
        """
        :return Packet buffer
        :return: str
        """
        return self._buf

    def get_source_server_ip(self):
        """
        :return: Server IP address for sender of the packet.
        :rtype: str
        """
        return self._source_server_ip

    def get_source_server_port(self):
        """
        :return: Server Port address for sender of the packet.
        :rtype: str
        """
        return self._source_server_port

    def get_source_server_address(self):
        """
        :return: Server address; The format is like ('192.168.001.001',
            '05335').
        :rtype: tuple
        """

        return self.get_source_server_ip(), self.get_source_server_port()
```

Node: Specifies to which node the packet sent to. **Sender** specifies who sent the packet **Validator** which makes the packet valid.
Header where the information such as type of the packet and etc. are going to be there.
Body body of our packet .

3.5 Reunion

reunion(packet) checks the connection of the nodes to the root.

```
#reunion(packet)
 get_destination()
```

3.6 Node

```
#Node
def send_message(self):
        """
        Final function to send buffer to the clients socket.
        :return:
        """
        print("in sending message: ", self.out_buff)
        for b in self.out_buff:
            print(b)
            response = self.client.send(b)

            if response.decode("UTF-8") != bytes('ACK'):
                print("The ", self.get_server_address()[0], ": ",
                     self.get_server_address()[1],
                      " did not response with b'ACK'.")

    def add_message_to_out_buff(self, message):
        """
        Here we will add new message to the server out_buff, then in
             'send_message' will send them.
        :param message: The message we want to add to out_buff
        :return:
        """
        self.out_buff.append(message)
    def close(self):
        """
        Closing client object.
        :return:
        """
        self.client.close()

    def get_server_address(self):
        """
        :return: Server address in a pretty format.
        :rtype: tuple
        """
        return self.server_ip, self.server_port

    @staticmethod
    def parse_ip(ip):
        """
        Automatically change the input IP format like '192.168.001.001'.
        :param ip: Input IP
        :type ip: str
        :return: Formatted IP
        :rtype: str
        """
        return '.'.join(str(int(part)).zfill(3) for part in
            ip.split('.'))

    @staticmethod
    def parse_port(port):
        """
        Automatically change the input IP format like '05335'.
        :param port: Input IP
        :type port: str
        :return: Formatted IP
        :rtype: str
        """
        return str(int(port)).zfill(5)
```

Every node has two parameters: **IP** and **Port**.

3.7 Resgister Request

reg_req() sends IP/Port of a node to the root to ask if it can register it.

3.8 Register Response

reg_res() should just send an *Ack* from the root to inform a node that it has been registered in the root if the reg_req() was successful.

3.9 Advertise

adv(packet)

3.10 Mesasge

msg(packet)