Sharif University of Technology

A Simple Peer To Peer Network Implementation

November 2018

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

Supervisor:
Dr. Mahdi Jafari

Computer Engineering Department

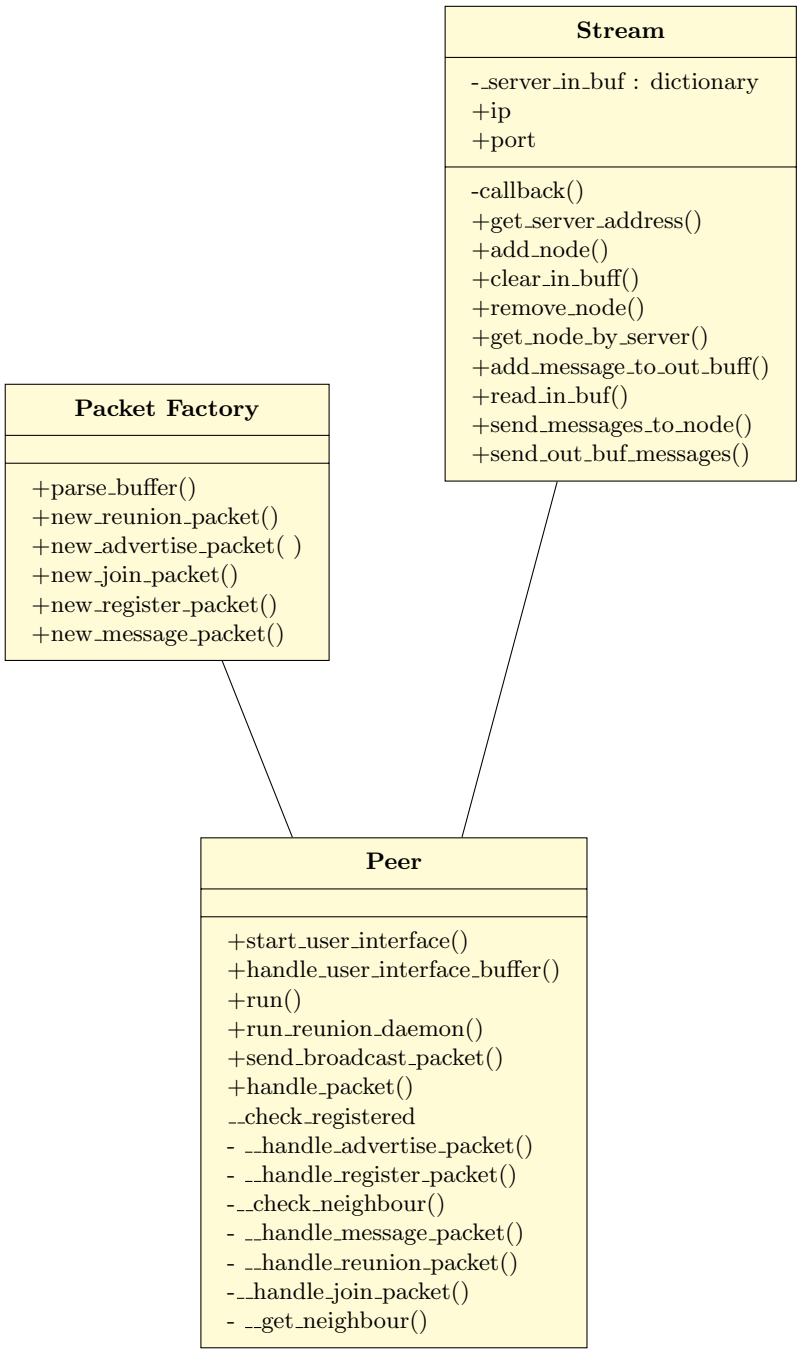
Contents

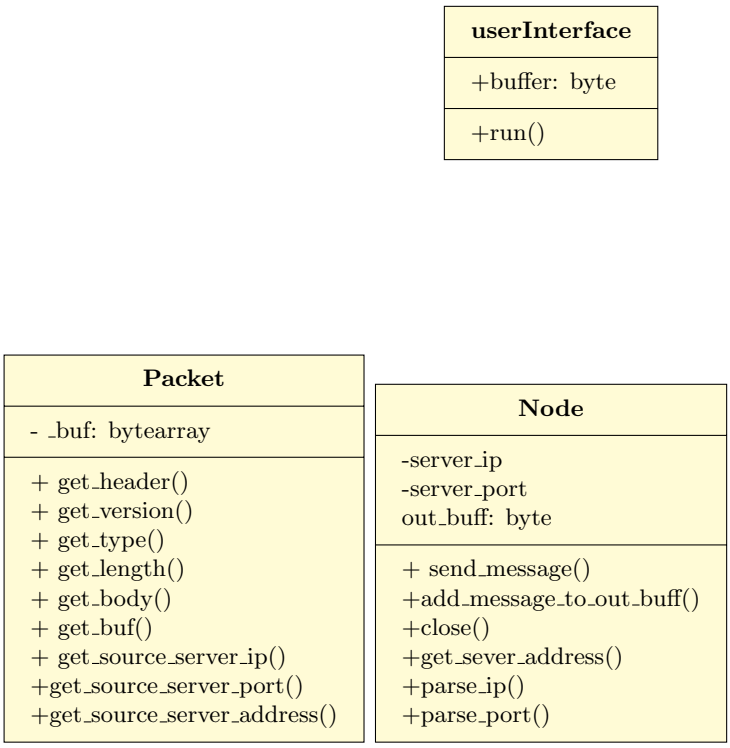
1 Introdeuction

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
+ip
+port
-callback()
+get_server_address()
+add_node()
+clear_in_buff()
+remove_node()
+get_node_by_server()
+add_message_to_out_buff()
+read_in_buf()
+send_messages_to_node()
+send_out_buf_messages()
Packet Factory
+parse_buffer()
+new_reunion_packet()
+new_advertise_packet( )
+new_join_packet()
+new_register_packet()
+new_message_packet()
Peer
+start_user_interface()
+handle_user_interface_buffer()
+run()
+run_reunion_daemon()
+send_broadcast_packet()
+handle_packet()
__check_registered
- __handle_advertise_packet()
- __handle_register_packet()
-__check_neighbour()
- __handle_message_packet()
- __handle_reunion_packet()
-__handle_join_packet()
- __get_neighbour()


userInterface
+buffer: byte
+run()
Packet
- _buf: bytearray
+ get_header()
+ get_version()
+ get_type()
+ get_length()
+ get_body()
+ get_buf()
+ get_source_server_ip()
+get_source_server_port()
+get_source_server_address()
Node
-server_ip
-server_port
out_buff: byte
+ send_message()
+add_message_to_out_buff()
+close()
+get_sever_address()
+parse_ip()
+parse_port()

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_inBuf 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()
```

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.

```
#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.

```
#packetFactory
```

3.4 Packet

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

```
#Packet
```

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

reunion(packet) checks the connection of the nodes to the root. This is one type of packets which each node has to send it to the root every 20 seconds to clarify is still live.
A node who want to send reunion packet, after creating the header, inserts it's ip and port in the body and passes it to his father. Then the father should append it's ip and port at the end of the body then passes it to the next father...
When the root recieves the reunion packet from one node, the root should reverse the sort of ip and ports in the reunion's body then sets it as the reunion answer's body and pass it to the node which gave the reunion to the root.
Each node which receives the reunion answer from it's father, should check if the first ip and port in the body are for himself or not. If yes, it should pop his port and ip from the body and then send it to the next node which it's ip and port are in the top of the body. This process continue until the destination node receives the reunion answer.

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

3.6 Node

```
#Node
```

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

3.7 Graph Node

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

3.8 Network Node

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

3.9 Semi Node

3.10