Hacettepe University - Computer Engineering

BBM203 Software Practicum I - Fall 2023

PROGRAMMING ASSIGNMENT 3

PROGRAMMING ASSIGNMENT 3

Subject: Stack, Queue, Dynamic Memory Allocation, File I/O
Instructors: Assoc. Prof. Dr. Adnan ÖZSOY, Asst. Prof. Dr. Engin DEMİR, Assoc. Prof. Dr. Hacer YALIM KELEŞ
TAs: Alperen ÇAKIN*, Ardan YILMAZ, **Dr. Selma DİLEK***
Programming Language: C++11 - **You MUST USE this starter code**
Due Date: **Friday, 08/12/2023 (23:59:59)**

HUBBM Computer Networking

Introduction

The evil teaching assistants *Elmas Delik*, *Alenper Kıçan*, and *Radan Zamyıl* from the Department of Computer Engineering at Hacettepe University grew weary of students sourcing code from the Internet rather than crafting it themselves for their programming assignments. Determined to change this, they devised a plan to generate an electromagnetic pulse (EMP) that would disable the network interface cards (NICs) in the students' computers, cutting off their Internet access. Naturally, they searched "*building your own EMP generator for dummies*" online to learn the process in their clandestine Crazy Projects Lab (ironically utilizing the Internet for this purpose). However, their lack of expertise in electrical and electronics design led to a disastrous EMP generator, built with incorrect capacitor values, power supplies, wire diameters, and whatnot, resulting in a perilously potent weapon of mass-electronics-destruction.

When they unleashed their monstrous EMP generator on the department building on the assignment's announcement day, it led to a horrific calamity. The blast incapacitated most electronics across the entire Beytepe campus, severely disrupting the university's operations.

Yet, the remarkable students from the Hacettepe Computer Engineering Department quickly rallied and formulated a rescue strategy. They unanimously decided to immediately construct their own network HUBBMNET, developing necessary network protocols. This would first enable them to reestablish computer communication within the department, complete and submit their assignment on time, and subsequently restore functionality across the entire campus.

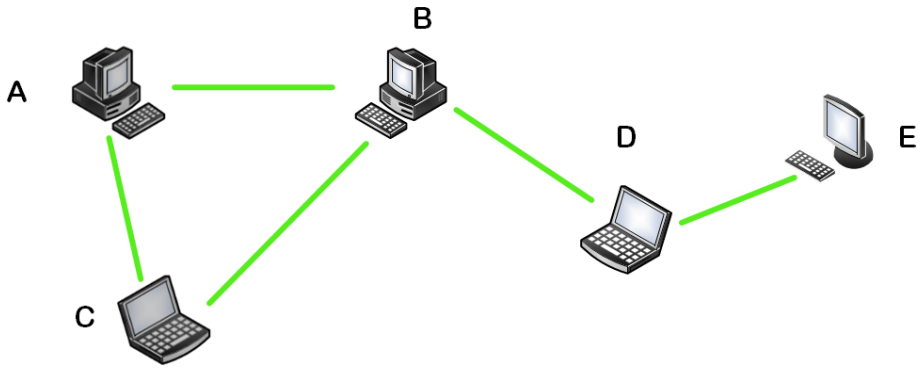
Mission Objectives

Your task is to implement a basic version of network communication between peers within a computer network; that is, to implement a highly simplified computer networking protocol family similar to the **Internet protocol suite**.

The network protocol stack will consist of four layers, each of which will have a different purpose, as shown in the Table below:

Layer ID	Layer	Protocol Services
0	**Application**	Communication between applications that use the network
1	**Transport**	End-to-end data delivery
2	**Network**	Defines packages and provides routing
3	**Data Link / Physical**	Routines for accessing physical media

The network will have a peer-to-peer model. A sample network topology is shown in the figure below.



The objective is to enable the peers (**from now on, we will refer to them as clients**) to communicate by exchanging messages while abiding by a set of protocols.

Basic Layer Services

- A client application will place a request to send a message of an arbitrary length to another client in the network. The Application Layer will be responsible for dividing the message into message chunks of allowable size because the links in the communication network will have a limited message size that can be delivered in a single frame over the physical media. At this point, the sender's and the receiver's IDs will be relevant to identify the communicating parties. Each message chunk will be sent in a separate frame.
- Transport Layer is responsible for appending the correct port numbers to the outgoing frames, based on which application is trying to communicate and whether it is an outgoing or an incoming socket.
- Network Layer needs to append the information regarding the sender and receiver IP addresses to the frame.
- Data Link / Physical Layer will finally append the physical MAC addresses of the clients on both ends of the link over which frames will make the next hop.

A Summary of Network Services

A summary of network services that you need to implement is as follows:

- Enable communication among the clients within a network based on the set of protocols.
- Provide a simple routing of data packets using the clients' routing tables.
- Enable logging of all network activity.
- Print log reports as necessary.

Network Structure and Protocols

In this section, we cover the basics of the network protocols you are expected to implement. Close adherence to dynamic memory allocation requirements is crucial for full credit.

Network Packets

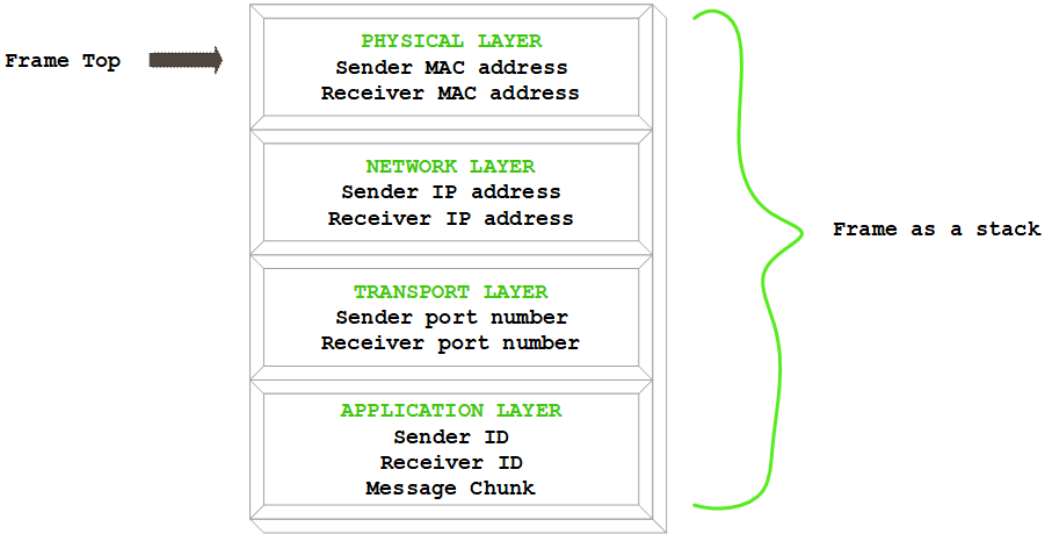
In networking, a network packet is a basic unit of data (a small segment of a larger message) that is grouped together and transferred over a computer network. In the HUBBMNET that you are going to develop, a **`Packet`** will be the fundamental unit that stores necessary data corresponding to each network layer respectively. Since each network layer focuses on different planes of the networking, a **`Packet`** is expected to contain the following data fields based on the layer it represents:

Type of the Packet	Data Fields in the Packet Load
Application Layer Packet	- Layer ID - Sender ID - Receiver ID - Message Chunk
Transport Layer Packet	- Layer ID - Sender port number - Receiver port number
Network Layer Packet	- Layer ID - Sender IP address - Receiver IP address
Physical Layer Packet	- Layer ID - Sender MAC address - Receiver MAC address

Note that an **`Application Layer Packet`** can only carry a message (or a message fragment) that does not exceed the maximum allowable data length (expressed as an upper limit on the number of characters that can fit into a single Message Chunk data field). Therefore, if the size of the message exceeds the given limit, it will have to be broken into message chunks and sent in multiple network frames. Frames are explained in the next section.

Frames as Stacks of Network Packets (Last In First Out)

Before data packets that carry message fragments can be sent over the network, they will have to be encapsulated into Data Link/Physical Layer **frames** that **must be implemented as stacks of packets** as defined by the network protocol stack. A frame structure is shown in the figure below.



When **Client X** wishes to transmit a message to **Client Y**, the process begins in the messaging application of Client X, where an `Application Layer Packet` is constructed. This packet contains the essential details: sender and receiver IDs, and the message content. Should the message size surpass the maximum length permitted, it will be segmented into multiple packets each of which will carry one message segment.

Subsequently, this `Application Layer Packet` is conveyed to the Transport Layer. Here, a corresponding `Transport Layer Packet` is created with the sender and receiver port numbers, and then placed on top of the `Application Layer Packet`.

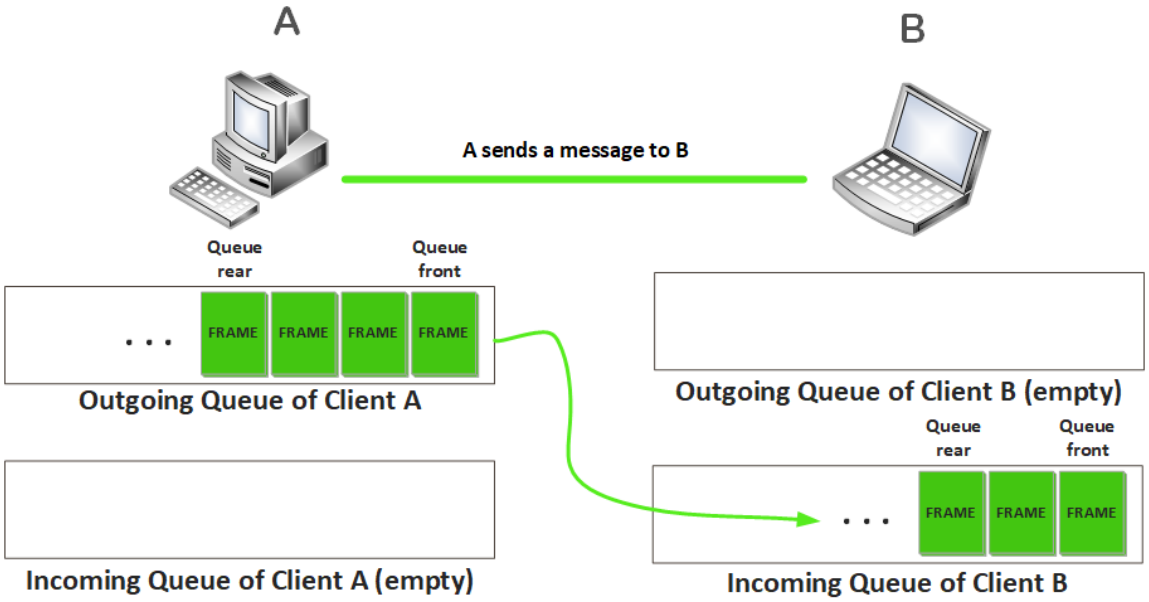
The process continues as these packets reach the Network Layer. At this stage, a corresponding `Network Layer Packet` is generated with the sender and receiver IP addresses, crucial for navigating the interconnected network paths.

The final encapsulation occurs at the Physical Layer. The data is prepared for transmission across the physical network medium, with each `Physical Layer Packet` containing the sender and receiver MAC addresses essential for the local network routing. Unlike other packets that specify the final receiver's data, the `Physical Layer Packet` should contain the MAC address of the next hop client that will forward the frames, and not the MAC address of the final intended destination client. If the next hop isn't the final receiver, the receiver MAC address will differ from the receiver info in the lower packets in the stack.

As these packets are generated, they are placed on top of each other on the stack to create a `frame`. Since a frame is a stack of packets, the frame top will point to the `Physical Layer Packet` and the `Application Layer Packet` will always be the last one popped from the frame stack when a frame reaches its destination.

Outgoing and Incoming Queues (First In First Out)

Each client will have the send and receive buffers named as **Outgoing and Incoming Queues** (**that must be implemented as queues of frames**) for outgoing and incoming messages, respectively (see the illustration below). You may assume that the queue size is infinite; i.e., packets should never be dropped because there is not enough space in a queue. Instead, the queue will expand as much as necessary to accommodate any frames that need to be placed.



The figure illustrates an example of communication between **Client A** and **Client B** in which **A** is messaging **B**. The message is first encapsulated in frames, each carrying a message fragment, and placed onto the sender's outgoing queue. As a frame leaves the sender's outgoing queue and gets transferred over the physical link, it will be placed into the receiver's incoming queue. Frames must follow the FIFO method (the first message chunk will be sent in the first frame and received in the first frame on the receiver's side).

Routing

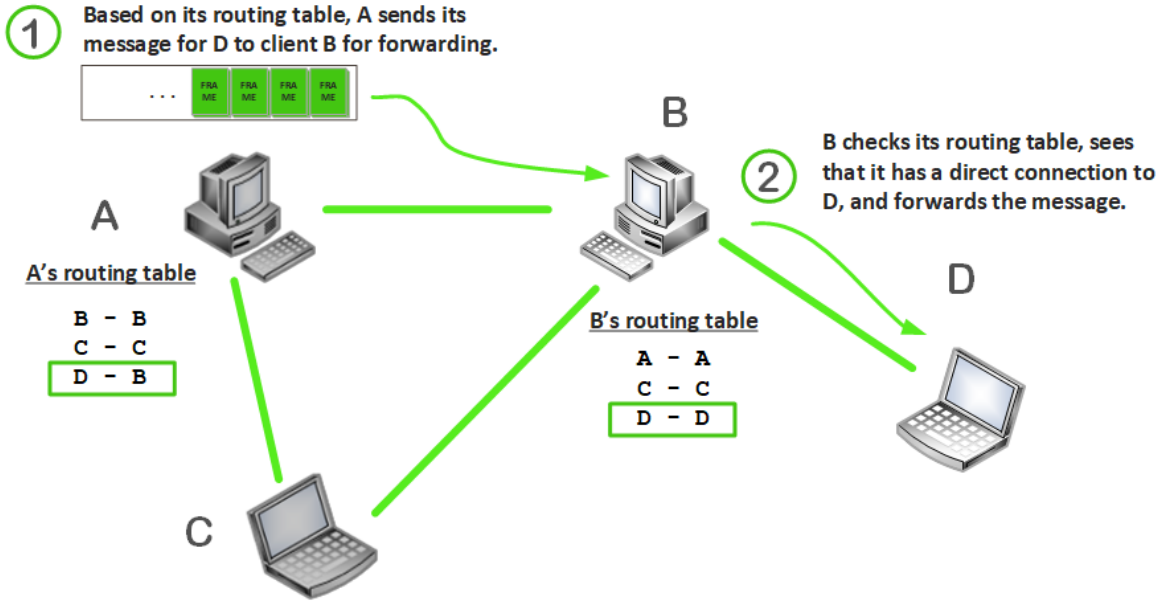
When a client receives message frames in its incoming queue, it needs to determine the intended receiver. If the frame is meant for the client itself, it should start unpacking the message chunks once the final fragment arrives. Then, based on the port number of its listening socket, it should pass these chunks to the application layer for message reassembly. On the other hand, if the message is meant for a different client (and the current client is merely a relay in the network), the client should check its routing table to identify the next hop for the frames en route to their final destination. This routing procedure should also be followed by the initial sender.

Routing tables of each client will have the following row format:

```
<Intended Destination Client ID><space><Neighbor Client ID to which the frame should be forwarded>
```

Routing tables may have multiple rows, one for each client in the network. For instance, the entry | A B | means that if a frame needs to reach **Client A**, it should be sent (or forwarded) to **Client B**.

A routing example is illustrated in the figure below. In this scenario, **Client A** wishes to communicate with **Client D**. However, the message surpasses the maximum permissible length, leading to its fragmentation into four distinct frames. Consulting its routing table for **D**, **A** finds no direct link. Instead, the table indicates an intermediary client, **Client B**, that can eventually relay (forward) the message to **D**. Thus, **A** dispatches its message to **B** for forwarding. Upon receiving a message aimed at **D**, **B** checks its routing table to determine the next hop. As **D** is directly connected to **B** (as per **B**'s routing table entry), **B** promptly forwards the message to **D**. If data regarding the subsequent hop is absent or compromised, the message will be dropped (discarded).



Frames may have to make multiple hops over the network before they reach their final destination. The information about the total number of hops taken should also be saved in the logs.

Logs

All network activities must be meticulously logged. As such, each client will maintain a `Log` that will document details of messages sent, received, forwarded, or dropped. Each entry in the `Log` should encapsulate the following data:

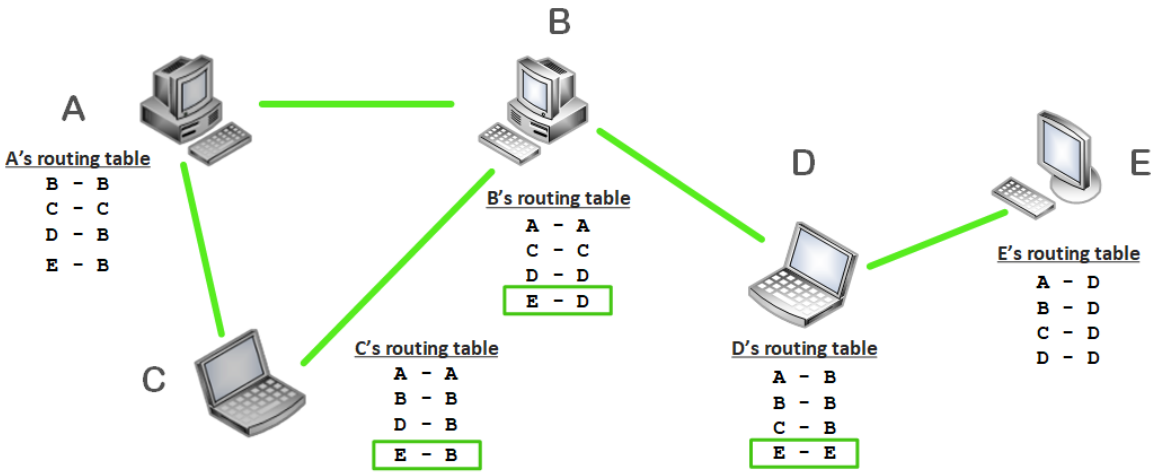
- `timestamp`: The date and time of the activity,
- `number_of_frames`: The cumulative number of frames comprising the entire message.
- `number_of_hops`: The total hops the frames have traversed within the network.
- `sender_id`: Application Layer Sender ID.
- `receiver_id`: Application Layer Receiver ID.
- `activity_type`: Designating whether the message was sent, received, forwarded, or dropped.
- `success_status`: Indicating the success (successful or failed) of the activity.

For `Log` entries corresponding to messages that were either sent or received, the following additional data should be recorded:

- `message_content`: The complete message that was sent/received.

An Example Use Case

Assume we have a network, illustrated in the figure below, in which the maximum message length supported by the links is **20 characters**. The routing tables of each client would be as shown below.



Also assume that the client IDs, addresses, and the socket port numbers are as follows.

Client ID	Client IP Address	Client MAC Address
A	1.2.3.4	AAAAAAAAAA
B	4.3.2.1	BBBBBBBBBB
C	8.8.8.8	CCCCCCCCCC
D	9.9.9.9	DDDDDDDDDD
E	0.0.1.1	EEEEEEEEEE

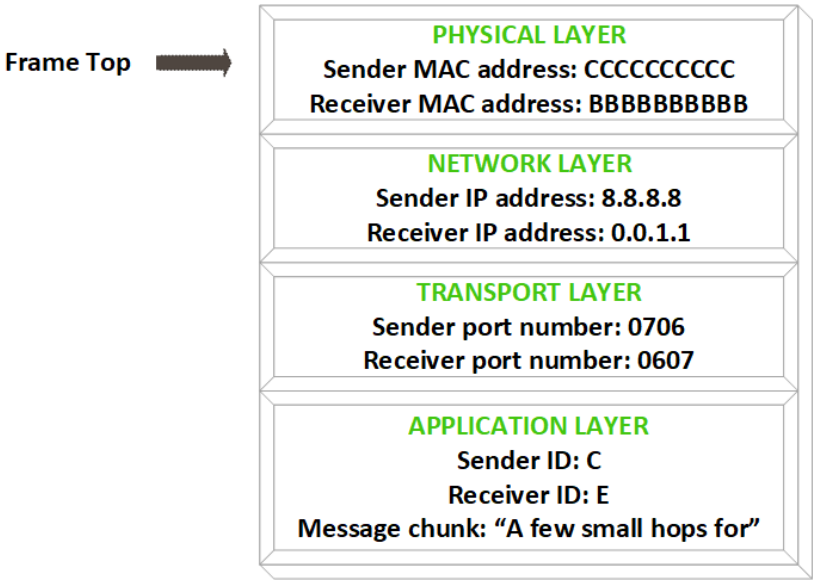
Service	Port Number
Sending socket	0706
Listening socket	0607

Let's suppose that **Client C** wishes to communicate with **Client E** and intends to send a message with the content: "**A few small hops for frames, but a giant leap for this message.**" **A message will always end in one of the following punctuation marks: period (.), question mark (?), or exclamation mark (!), and these punctuation marks can only appear at the end of a message**. Given that the message length is 63 characters and the maximum frame message length supported is 20 characters, the message will need to be transmitted over four distinct frames. Note that spaces are also considered as characters in the message chunks:

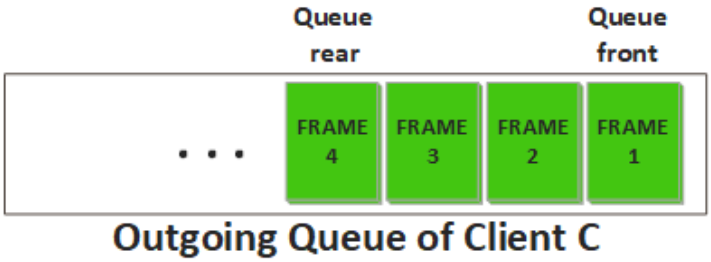
1. "`A few small hops for`"
2. "` frames, but a giant`"
3. "` leap for this messa`"
4. "`ge.`"

The following steps are required to occur within the network to facilitate this communication:

1. **C** will fragment the message into four parts, encapsulate each chunk within a frame, and add these frames to its outgoing queue. For example, the format of the first frame would be as follows:



   The subsequent three frames will follow a similar format, with variations only in the message chunk they contain. After this process, client **C**'s outgoing queue will appear as illustrated below.

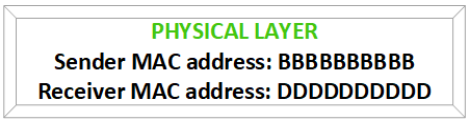


2. **C** checks its routing table to determine the next receiver. According to **C**'s routing table, the next hop for packets destined for **E** is **Client B**, as **E** is not a direct neighbor of **C**. Therefore, **C** sends the frames to **B**, setting the Physical Layer information (MAC addresses) accordingly, to reflect the fact that the frames will make the next hop over the **C**-**B** link.

3. **B** receives the frames in its incoming queue in the same sequence they were dequeued from **C**'s outgoing queue. This transfer should adhere to queue operations; that is, when a frame is dequeued from the sender's outgoing queue, it's enqueued onto the receiver's incoming queue, until all frames have been transferred.

4. Upon receipt, **B** examines the receiver information within the frames to identify the intended recipient. This is achieved by popping the packets from the received frame until the necessary data (Receiver ID) is accessed. **B** determines that the message is intended for another client in the network, specifically **E**.

5. **B** then checks its routing table to determine the next hop for the frames. It determines that the frames should be relayed to **D**, on their way to **E**. Consequently, the frames are reassembled by pushing the packets onto the frame stack in order, while the MAC addresses of the `Application Layer Packets` are updated to indicate that the next transmission will occur over the **B**-**D** link. The `Application Layer Packet` of each frame is then set as:



6. Then, **B** dequeues the frames from its incoming queue and enqueues them onto its outgoing queue for forwarding.

7. Similarly to the process performed in the previous message receiving step, **D** receives the frames in its incoming queue in the same order they are dequeued from **B**'s outgoing queue.

8. **D** inspects the receiver information within the frames to identify the intended recipient using the same steps described above. Recognizing the destination as **E**, **D** reconstructs and transfers the frames from its incoming queue to its outgoing queue for further forwarding. For reconstruction of the frames at this step, **D** checks its routing table to determine the next hop the frames should take, and determines that **E** is its direct neighbor. Hence, **D** forwards the frames to **E**, adjusting the MAC addresses as required.

9. Finally, when **E** receives the frames in its incoming queue, it identifies that the message has arrived at its intended destination. **E** then unpacks the message and passes it to the respective application.

Note that each activity must be logged by the corresponding client in the Log format described above. The message frames should travel through the network until they reach the intended client, or are dropped due to some unforeseen circumstances (e.g., unreachable user).

In summary, when receiving frames, clients must first identify the intended recipient. If the frames have reached their destination, they can be unpacked to reassemble the message; otherwise, they should be forwarded by doing the necessary Physical Layer packet adjustments. When sending or forwarding a message, clients should consult their routing tables to determine the next hop for the frames. If the next hop cannot be determined (e.g., corrupted routing table), the frames will be dropped.

Input Files and Parameters

There will be three input files (two for initializing the network and one with the commands), and *three additional command-line arguments specifying the **maximum supported message size**, **outgoing** and **incoming port numbers** for applications that will communicate, respectively*. Clients and the network topology will be initialized through the first two input files.

Client Info File Format

An input file containing details about the network **clients** will be supplied in `DAT` format via the *first command line argument* with the following format:

```
number_of_clients
Client_ID<space>Client_IP_Address<space>Client_MAC_Address
```

For example, to initialize a network with five clients from our example scenario, the input file will have content shown on the right.

```
5
A 1.2.3.4 AAAAAAAAAA
B 4.3.2.1 BBBBBBBBBB
C 8.8.8.8 CCCCCCCCCC
D 9.9.9.9 DDDDDDDDDD
E 0.0.1.1 EEEEEEEEEE
```

Routing Tables File Format

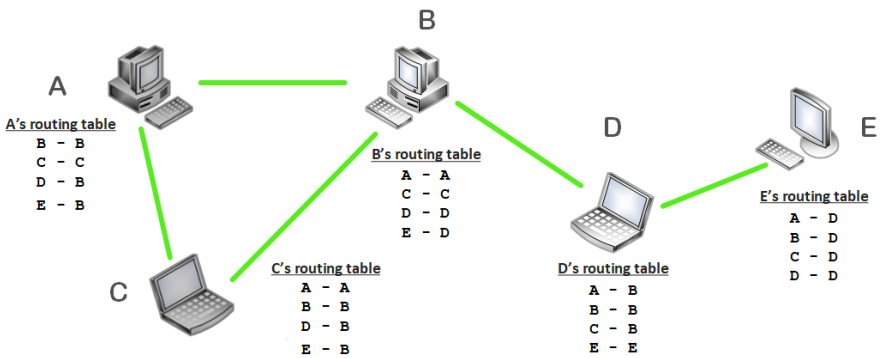
The second input file containing the details about the network topology (**routing tables**) will also be supplied in `DAT` format via the *second command line argument* in the format shown on the right. Where each `client_routing_table` entry has the following format:

```
Client_1_routing_table
-
Client_2_routing_table
-
...
-
Client_n_routing_table
```

```
Destination_Client_ID<space>Nearest_neighbor_Client_ID_on_the_path
```

For example, our sample use case scenario, illustrated in the figure to the right, would have the following routing info file contents:

```
B B
C C
D B
E B
-
A A
C C
D D
E D
-
A A
B B
D B
E B
-
A B
B B
C B
E E
-
A D
B D
C D
D D
```



Commands File Format

Students' implementations will be tested via network commands that will be given in `DAT` format via the *third command line argument*. The commands will specify networking activity your program must simulate (e.g., which client wants to communicate with whom and what message needs to be sent, what output is expected regarding network status, etc.). The file format is given below.

```
number_of_commands
COMMAND<space>command_parameter_1<space>command_parameter_2...
```

If we were to request a network status check in the slightly modified example use case discussed above, here is a sample command input file `commands.dat` that would accomplish that (in this modified version, **Client C** is sending two messages to **Client E**):

```
17
MESSAGE C E #A few small hops for frames, but a giant leap for this message.#
MESSAGE C E #TAs' PAs = no sleep!#
SHOW_FRAME_INFO C out 3
SHOW_Q_INFO C out
SHOW_Q_INFO C in
SHOW_FRAME_INFO C in 5
SEND
RECEIVE
SEND
RECEIVE
SEND
RECEIVE
PRINT_LOG A
PRINT_LOG B
PRINT_LOG C
PRINT_LOG D
PRINT_LOG E
```

Instructions about the commands that should be executed are given in the next section.

Program Execution

When your program is executed, all of these parameters will be given as command-line arguments in the following order:

```
$ ./HUBBMNET clients.dat routing.dat commands.dat max_msg_size outgoing_port
↪  incoming_port
```

Note that file names may change, but the order will always stay the same. Your program must be able to accommodate this.

Networking Commands

HUBBMNET features **six** networking commands that will be listed in the corresponding input file, each on a new line. Commands must be interpreted dynamically and immediately as they are encountered, not stored in memory, mimicking real-time networking conditions. The commands are as follows:

- **MESSAGE**: Initiates communication between the clients.
- **SHOW_FRAME_INFO**: Displays the status of a message frame.
- **SHOW_Q_INFO**: Displays the status of a client's incoming or outgoing queue.
- **SEND**: Initiates a single transmission step within the entire network.
- **RECEIVE**: Initiates a single reception step within the entire network.
- **PRINT_LOG**: Prints the log of a given client.

Client Communication

Suppose we want to simulate a message exchange between two clients. **MESSAGE** command will accomplish the preparation of the message for transmission, that is, its fragmentation and packing into frames. The format of the **MESSAGE** command is:

```
MESSAGE<space>sender_ID<space>receiver_ID<space>#message#
```

This command will trigger the preparation of the message (given between two hashtags that should not be included in the message) for transmission by fragmenting it if necessary, packing it into frames, and queueing the frames onto the sender's outgoing queue. **The message should not be actually sent until the command SEND is given.**

The output of our sample **MESSAGE** command "`MESSAGE C E #A few small hops for frames, but a giant leap for this message.#`" should print the information about the message and all frames that were prepared for its transmission in the format given below. Additionally, the sender client should log this action appropriately (see **Client Logs** section for details).

```
--------------------------------------------------------------------------------
Command: MESSAGE C E #A few small hops for frames, but a giant leap for this message.#
--------------------------------------------------------------------------------
Message to be sent: "A few small hops for frames, but a giant leap for this message."

Frame #1
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: "A few small hops for"
Number of hops so far: 0
--------
Frame #2
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: " frames, but a giant"
Number of hops so far: 0
--------
Frame #3
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: " leap for this messa"
Number of hops so far: 0
--------
Frame #4
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: "ge."
Number of hops so far: 0
--------
```

Note, again, that the receiver information in the Data Link/Physical layer (MAC address) does not match the receiver IP address and ID. As we discussed on Page 5, the `PhysicalLayerPacket` should contain the MAC address of the **next hop client that will forward the frames**, and not the MAC address of the final intended destination client, unlike other packets that specify the final receiver's data. Finally, a similar output would be produced for the second sample command "`MESSAGE C E #TAs' PAs = no sleep!#`", with a difference of having a whole message fit into a single frame. Please check the given sample IO for details.

Incoming/Outgoing Queue Status Check

The command **SHOW_FRAME_INFO** is used to inspect the contents of a frame on a client's queue (incoming or outgoing). Its format is:

```
SHOW_FRAME_INFO<space>client_ID<space>queue_selection<space>frame_number
```

The parameter `queue_selection` can be either "`in`" or "`out`" specifying the incoming and outgoing queue, respectively. Suppose we requested to inspect the contents of the **Frame #3** on the **Client C**'s **outgoing queue** with the third sample command, just as the communicating application placed the request for sending the message. The required printed output should be formatted as follows:

```
------------------------------
Command: SHOW_FRAME_INFO C out 3
------------------------------
Current Frame #3 on the outgoing queue of client C
Carried Message: " leap for this messa"
Layer 0 info: Sender ID: C, Receiver ID: E
Layer 1 info: Sender port number: 0706, Receiver port number: 0607
Layer 2 info: Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Layer 3 info: Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Number of hops so far: 0
```

SHOW_Q_INFO command is used to inspect a client's queue in general. Its format is:

```
SHOW_Q_INFO<space>client_ID<space>queue_selection
```

In our example, we want to inspect the **Client C**'s **outgoing queue** status with the fourth sample command. The required printed output should be formatted as follows:

```
-------------------------
Command: SHOW_Q_INFO C out
-------------------------
Client C Outgoing Queue Status
Current total number of frames: 5
```

Similarly, with the fifth command, the status of **C**'s **incoming queue** would be printed (note that C has not received anything yet):

```
------------------------
Command: SHOW_Q_INFO C in
------------------------
Client C Incoming Queue Status
Current total number of frames: 0
```

With the sixth command, we requested to inspect the contents of the Frame #5 on the **C**'s **incoming queue**, but there is no such frame. The output in such cases should be:

```
------------------------------
Command: SHOW_FRAME_INFO C in 5
------------------------------
No such frame.
```

Sending and Receiving Messages

SEND command triggers the transmission of message frames from all clients' outgoing queues to their respective next hop in the network, determined by the receiver MAC address for each message frame. The format of this command is:

```
SEND
```

When this command is given, all frames in any client's outgoing queue will be forwarded from the sender to the next hop recipient's incoming queue, with hop counts updated accordingly. A network trace will be printed to `STDOUT` in the following format:

```
-------------
Command: SEND
-------------
Client C sending frame #1 to client B
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: "A few small hops for"
Number of hops so far: 1
--------
Client C sending frame #2 to client B
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: " frames, but a giant"
Number of hops so far: 1
--------
Client C sending frame #3 to client B
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: " leap for this messa"
Number of hops so far: 1
--------
Client C sending frame #4 to client B
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: "ge."
Number of hops so far: 1
--------
Client C sending frame #1 to client B
Sender MAC address: CCCCCCCCCC, Receiver MAC address: BBBBBBBBBB
Sender IP address: 8.8.8.8, Receiver IP address: 0.0.1.1
Sender port number: 0706, Receiver port number: 0607
Sender ID: C, Receiver ID: E
Message chunk carried: "TAs' PAs = no sleep!"
Number of hops so far: 1
--------
```

RECEIVE command initiates the reception and processing of frames within the network. The format of this command is:

```
RECEIVE
```

When this command is given, each client's incoming queue should be checked for any potential incoming frames. If the frames have arrived to their final destination, **the message should be assembled**, otherwise, **the frames should be modified to reflect the next hop in the network they need to take** and **placed onto the outgoing queue**. The most important thing in forwarding is setting the proper MAC addresses within the frames as they hop over each link. Additionally, actions will be logged at each client (see **Client Logs** section for details), and a network trace will be printed to `STDOUT` in the following format:

```
----------------
Command: RECEIVE
----------------
Client B receiving a message from client C, but intended for client E. Forwarding...
Frame #1 MAC address change: New sender MAC BBBBBBBBBB, new receiver MAC DDDDDDDDDD
Frame #2 MAC address change: New sender MAC BBBBBBBBBB, new receiver MAC DDDDDDDDDD
Frame #3 MAC address change: New sender MAC BBBBBBBBBB, new receiver MAC DDDDDDDDDD
Frame #4 MAC address change: New sender MAC BBBBBBBBBB, new receiver MAC DDDDDDDDDD
--------
Client B receiving a message from client C, but intended for client E. Forwarding...
Frame #1 MAC address change: New sender MAC BBBBBBBBBB, new receiver MAC DDDDDDDDDD
--------
```

Forwarding message to its intended destination may not be always possible. Suppose that in our example use case the routing table of **B** was as shown on the right:

A	B
C	C
D	D
E	X

That is, when the message intended for **Client E** reaches **client B**, **B** would not be able to find what the next hop should be. In that case, frames should be dropped, and an error message stating that the user is unreachable should also be displayed in the following format:

```
----------------
Command: RECEIVE
----------------
Client B receiving frame #1 from client C, but intended for client E. Forwarding...
Error: Unreachable destination. Packets are dropped after 1 hops!
Client B receiving frame #2 from client C, but intended for client E. Forwarding...
Error: Unreachable destination. Packets are dropped after 1 hops!
Client B receiving frame #3 from client C, but intended for client E. Forwarding...
Error: Unreachable destination. Packets are dropped after 1 hops!
Client B receiving frame #4 from client C, but intended for client E. Forwarding...
Error: Unreachable destination. Packets are dropped after 1 hops!
--------
Client B receiving frame #1 from client C, but intended for client E. Forwarding...
Error: Unreachable destination. Packets are dropped after 1 hops!
--------
```

Client Logs

With the command **PRINT_LOG**, we request to print all logs of a client in the network. The format of this command is:

```
PRINT_LOG<space>client_ID
```

In our sample use case, when the commands `PRINT_LOG D` and `PRINT_LOG E` are given, **D** has received and forwarded two messages, and **E** has received two messages, so the `STDOUT` outputs should be as shown below:

```
--------------------
Command: PRINT_LOG D
--------------------
Client D Logs:
--------------
Log Entry #1:
Activity: Message Forwarded
Timestamp: 2023-11-22 20:30:03
Number of frames: 4
Number of hops: 2
Sender ID: C
Receiver ID: E
Success: Yes
--------------
Log Entry #2:
Activity: Message Forwarded
Timestamp: 2023-11-22 20:30:03
Number of frames: 1
Number of hops: 2
Sender ID: C
Receiver ID: E
Success: Yes
```

```
--------------------
Command: PRINT_LOG E
--------------------
Client E Logs:
--------------
Log Entry #1:
Activity: Message Received
Timestamp: 2023-11-22 20:30:03
Number of frames: 4
Number of hops: 3
Sender ID: C
Receiver ID: E
Success: Yes
Message: "A few small hops for frames, but a giant leap for this message."
--------------
Log Entry #2:
Activity: Message Received
Timestamp: 2023-11-22 20:30:03
Number of frames: 1
Number of hops: 3
Sender ID: C
Receiver ID: E
Success: Yes
Message: "TAs' PAs = no sleep!"
```

In our sample use case, when the commands `PRINT_LOG C` is given, **C** has sent two messages. The output of this command is given below on the left. Finally, remember the case in which the frames were dropped by **B** because the next hop could not be determined due to the lack of routing information? In such case, the output of `PRINT_LOG B` of would be formatted as shown below on the right.

```
--------------------
Command: PRINT_LOG C
--------------------
Client C Logs:
--------------
Log Entry #1:
Activity: Message Sent
Timestamp: 2023-11-22 20:30:03
Number of frames: 4
Number of hops: 0
Sender ID: C
Receiver ID: E
Success: Yes
Message: "A few small hops for frames, but a giant leap for this message."
--------------
Log Entry #2:
Activity: Message Sent
Timestamp: 2023-11-22 20:30:03
Number of frames: 1
Number of hops: 0
Sender ID: C
Receiver ID: E
Success: Yes
Message: "TAs' PAs = no sleep!"
```

```
--------------------
Command: PRINT_LOG B
--------------------
Client B Logs:
--------------
Log Entry #1:
Activity: Message Dropped
Timestamp: 2023-11-22 20:30:03
Number of frames: 4
Number of hops: 1
Sender ID: C
Receiver ID: E
Success: No
--------------
Log Entry #2:
Activity: Message Dropped
Timestamp: 2023-11-22 20:30:03
Number of frames: 1
Number of hops: 1
Sender ID: C
Receiver ID: E
Success: No
```

In case a client does not have any log entries yet, nothing should be printed.

Invalid Commands

In case an invalid command is given, your program should print an error message in the following format:

```
----------------------
Command: DDOS_ATTACK A
----------------------
Invalid command.
```

Assignment Implementation Tasks and Requirements

In this section, we outline the classes and functions you are required to implement. As aspiring engineers, it's crucial to base your code on the provided starter code that offers a predetermined class structure. This ensures code clarity, proper encapsulation, and facilitates unit testing. It's imperative that you do not alter the names or signatures of functions provided in the template files. Likewise, refrain from renaming or changing the types of the specified member variables. Other than that, you're free to introduce any additional functions or variables as needed.

`Client` Class

- **Constructor**:
  `Client(string const& id, string const& ip, string const& mac)`
  - Initializes a network client (already given).
- **Operator** $<<$: Overloads output stream operator $<<$ (already given).
- **Destructor**:
  `~Client()`
  - Frees any dynamically allocated memory if necessary.

`Network` Class

- **Functions**:
  `vector<Client> read_clients(string const &filename)`
  - Reads clients from the given input file and returns a vector of `Client` instances.

  ```
  void read_routing_tables(vector<Client> & clients, string const
  ↪  &filename)
  ```
  - Reads the routing tables from the given input file and populates the clients' `routing_table` member variable.

  `vector<string> read_commands(const string &filename)`
  - Reads commands from the given input file and returns them as a vector of strings.

  ```
  void process_commands(vector<Client> &clients, vector<string> &commands,
  ↪  int message_limit, const string &sender_port, const string
  ↪  &receiver_port)
  ```
  - Executes the commands given as a vector of strings while utilizing the remaining arguments.

`Packet` Class

- **Constructor**:
  `Packet(int _layer_ID)`
  - Initializes a network packet (already given).
- **Function**:
  `virtual void print()`
  - A virtual `print` function that needs to be overriden in subclasses.
- **Operator** $<<$: Overloads output stream operator $<<$ (already given).

- **Destructor**:
  `~Packet()`
  - Frees any dynamically allocated memory if necessary.

`ApplicationLayerPacket` Class

- **Constructor**:
  `ApplicationLayerPacket(int _layer_ID, const string& _sender_ID, const string& _receiver_ID, const string& _message_data)`
  - Initializes an Application Layer packet that extends Packet class (already given).
- **Function**:
  `void print()`
  - Overrides the virtual `print` function from `Packet` class to print layer-specific properties.
- **Destructor**:
  `~ApplicationLayerPacket()`
  - Overrides `Packet` class destructor and frees any dynamically allocated memory if necessary.

> The instructions given for the `ApplicationLayerPacket` class above, are also applicable for `TransportLayerPacket`, `NetworkLayerPacket`, and `PhysicalLayerPacket` classes. The expected output format of the overriden `print` function for each subclass is given below.

`ApplicationLayerPacket` class `print()` function format:

```
Sender ID: A, Receiver ID: B
```

`TransportLayerPacket` class `print()` function format:

```
Sender port number: 0706, Receiver port number: 0607
```

`NetworkLayerPacket` class `print()` function format:

```
Sender IP address: 1.2.3.4, Receiver IP address: 4.3.2.1
```

`PhysicalLayerPacket` class `print()` function format:

```
Sender MAC address: AAAAAAAAAA, Receiver MAC address: BBBBBBBBBB
```

`Log` Class

- **Constructor**:
  `Log(const string &_timestamp, const string &_message, int _number_of_frames, int _number_of_hops, const string &_sender_id, const string &_receiver_id, bool _success, ActivityType _type)`
  - Initializes an instance of a client log (already given).
- **Destructor**:
  `~Log()`
  - Frees any dynamically allocated memory if necessary.

Must-Use Starter Code

You MUST use **this starter code**. All classes should be placed directly inside your **zip** archive.

Build and Run Configuration

Here is an example of how your code will be compiled (note that instead of `main.cpp` we will use our test files):

```
$ g++ -std=c++11  *.cpp, *.h -o HUBBMNET
```

Or, you can use the provided `Makefile` or `CMakeLists.txt` within the sample input to compile your code:

```
$ make
```

or

```
$ mkdir HUBBMNET_build
$ cmake -S . -B HUBBMNET_build/
$ make -C HUBBMNET_build/
```

After compilation, you can run the program as follows:

```
$ make
$ ./HUBBMNET clients.dat routing.dat commands.dat max_msg_size outgoing_port
↪  incoming_port
```

Grading Policy

- No memory leaks and errors: 10%
  - No memory leaks: 5%
  - No memory errors: 5%
- Correct implementation of the networking protocols: 80%
  - Proper network initialization from the input files 5%
  - Implementation of message fragmentation and stack frame initialization 10%
  - Implementation of frame queue initialization 10%
  - Implementation of stack reassembly and dequeuing/enqueuing operations 15%
  - Implementation of message sending 10%
  - Implementation of message forwarding 10%
  - Implementation of message assembly if message received 10%
  - Implementation of client logs 10%
- Output tests: 10%

Note that you need to get a NON-ZERO grade from the assignment in order to get the submission accepted. Submissions with grade 0 will be counted as NO SUBMISSION!

Important Notes

- Do not miss the deadline: **Friday, 08/12/2023 (23:59:59)**.
- Save all your work until the assignment is graded.
- The assignment solution you submit must be your original, individual work. Duplicate or similar assignments are both going to be considered as cheating.
- You can ask your questions via Piazza (`https://piazza.com/hacettepe.edu.tr/fall2023/bbm203`), and you are supposed to be aware of everything discussed on Piazza.
- You must test your code via **Tur^{3}Bo Grader** `https://test-grader.cs.hacettepe.edu.tr/` (**does not count as submission!**).
- You must submit your work via `https://submit.cs.hacettepe.edu.tr/` with the file hierarchy given below:
  - **b<studentID>.zip**
    - ApplicationLayerPacket.cpp <FILE>
    - ApplicationLayerPacket.h <FILE>
    - Client.cpp <FILE>
    - Client.h <FILE>
    - Log.cpp <FILE>
    - Log.h <FILE>
    - Network.cpp <FILE>
    - Network.h <FILE>
    - NetworkLayerPacket.cpp <FILE>
    - NetworkLayerPacket.h <FILE>
    - Packet.cpp <FILE>
    - Packet.h <FILE>
    - PhysicalLayerPacket.cpp <FILE>
    - PhysicalLayerPacket.h <FILE>
    - TransportLayerPacket.cpp <FILE>
    - TransportLayerPacket.h <FILE>
- **You MUST use this starter code**. All classes should be placed directly in your **zip** archive.
- This file hierarchy must be zipped before submitted (not .rar, only .zip files are supported).
- Do not submit any object or executable files. Only header and C++ files are allowed.

Academic Integrity Policy

All work on assignments **must be done individually**. You are encouraged to discuss the given assignments with your classmates, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in pseudocode) **will not be tolerated**. In short, turning in someone else's work (including work available on the internet), in whole or in part, as your own will be considered as **a violation of academic integrity**. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

The submissions will be subjected to a similarity check. Any submissions that fail the similarity check will not be graded and will be reported to the ethics committee as a case of academic integrity violation, which may result in the suspension of the involved students.