Understanding Network Components and Functionalities in a Software-Based Network Emulator

Maitry Kamlesh Chauhan
mc22k@fsu.edu
Florida State University
Tallahassee, Florida, USA

ABSTRACT

Network emulation is a powerful tool that enables efficient testing and debugging, saves resources, simplifies the debugging process, offers scalability, and helps in the development of network prototypes. As networks continue to evolve and play a central role in modern technology, the importance of effective network can be highly simplified with network emulators. With this software based emulator, we are replicating a simple IP network which consists of stations, bridges, routers inferencing their functionalities. Our project uses TCP socket connections to create a network emulator that simulates real world network components like stations, bridges, and routers in software. Bridges work as servers, and stations/routers act as clients. Bridges distribute Ethernet data to all stations, learning their locations automatically eventually through self learning tables. Routers manage data flow between interfaces based on IP addresses and routing tables. The emulator also incorporates ARP, which connects IP addresses with MAC addresses, allowing smooth data flow. This setup is a practical way to apply and study network concepts, offering a controlled environment for experiments and learning about networking. Our research project not only demonstrates the feasibility of implementing networking concepts and functionalities in software but also provides a valuable tool for network research and education, making it easier for researchers and educators to conduct experiments and simulations in an ideal environment, enabling a deeper understanding of networking principles and protocols.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In terms of computer networking, this project introduces the Network Emulator—a sophisticated and adaptive tool designed to address the detailed challenges of analyzing, experimenting, and testing complex network architectures. It comes with no surprise that the field of computer networks continually evolves, resulting in hundreds and thousands of new protocols and topologies emerging everyday. Responding to this dynamic nature, Network Emulation emerges as a responsive and holistic solution, providing a virtualised platform that indirectly mirrors real-world network behaviors. Thus it is very compelling to create a network emulator.

Network emulation is essential for cost-effective experimentation, as it avoids the high expenses associated with physical network setups, including hardware, maintenance, and scalability costs. This approach enables researchers and developers to test various network scenarios economically, without needing extensive physical infrastructure. Emulation also provides a controlled environment, allowing for the consistent and reproducible testing of network protocols and algorithms. This consistency enhances the reliability of research outcomes.

The network emulator is composed of three main components, each necessary for simulating the complexity of real-world networks. These components work

in unison to replicate the complex functions and interactions typical in actual network environments.

Stations: Stations in the network emulator function as equivalents to end-user devices, packaging messages into IP packets and engaging in communications with bridges and routers. These stations play a crucial role in the emulation environment, as they mirror the detailed interactions that take place in various network scenarios. Stations are like clients in socket programming, They connect with servers to pass the data from one client to other client or one client to the server.

Bridges:Bridges in the network emulator act as key coordinators in Local Area Networks (LANs), managing the flow of data between connected stations. They use a self-learning approach to adapt and improve data forwarding efficiently. Functioning as servers, bridges set up TCP socket connections with stations, mimicking the physical connections in a network. They constantly listen for new connections while also transferring data across interfaces. Bridges can have multiple interfaces to handle various connections.

Routers: Routers in the network emulator act as smart messengers, helping data move between different LANs. They use routing tables to direct the data flow. These tables keep track of IP addresses and the next router to send data to. Routers also remember the MAC addresses of these next routers, which speeds up data forwarding because they don't have to send an ARP request every time. This use of ARP (Address Resolution Protocol) is crucial as it makes routers more efficient at sending packets. Overall, this setup allows routers to choose the best paths for sending IP packets across the emulated network.

The network emulator we're discussing uses advanced software to mimic physical network links, making it flexible and scalable. This software-based method is more adaptable and less expensive than using real hardware for network setups. It provides a great platform for experimenting with different network types and keeps up with changing network technologies.

This emulator is very useful for both research and education. In research, it's a reliable tool for studying network protocols, algorithms, and new behaviors. For education, it gives students practical experience in setting up, managing, and fixing networks in a safe, controlled way. This helps bridge the gap between theoretical learning and real-world practice, deepening their understanding of networking.

As we explore the emulator further, we'll look at its design, how it's made, and what it can do. This project is at the forefront of network experimentation, combining sophisticated software with detailed emulation. It's a key tool for advancing network research and education, offering new possibilities and insights into the complex world of networking. It meets a wide range of needs, showing its importance in the future of network studies at the same time give a of opportunity to beginners to learn the basic functionality of a fully developed network.

2 OVERVIEW

In our reference network model, there are three Local Area Networks (LANs), each linked to a bridge. Routers join these LANs, allowing stations on different LANs to communicate with each other. It's important to mention that a single bridge can connect to multiple stations and routers. Similarly, in a different setup we'll explore later, a station can be connected to more than one interface, each on different bridges.

The network components are:

- Bridges: Bridges connect multiple LAN segments together. They forward frames between LAN segments and learn the MAC addresses and port numbers of stations on the LANs to forward frames directly to the destination station. It creates the self learning tables and keepsit updated.
- Stations: Stations are network devices connected to a LAN. They send and receive data packets over the LAN.
- Routers: Routers connect two or more networks together. They forward packets between networks and learn the routes to different networks to forward packets to the correct destination. Routers mostly never looks at the message itself, It just looks at the source and destination and send to the next hop acordingly.

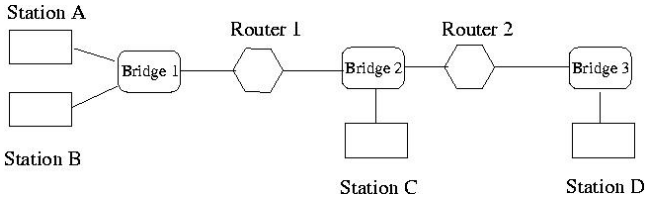

Figure 1: Simple Network Topology

The image shows a network diagram with three LANs - Bridge 1,2 and 3, which are interconnected by two routers- Routers r1 and r2. It also has end devices also called stations as Station A,B,C and D. Each LAN is connected to a bridge, which learns the location of the stations on the LAN and forwards frames to the necessary segment of the LAN. Stations and routers are assigned globally unique IP addresses. IP routers are responsible for delivering IP packets to their next-hops towards the destination based on the packet destination IP address.

The figure depicts as below:

- The three LANs are labeled LAN 1, LAN 2, and LAN 3.
- Each LAN is connected to a bridge, which is labeled Bridge 1, Bridge 2, and Bridge 3, respectively.
- The two routers are labeled as Router 1 and Router 2, respectively.
- Router 1 interconnects LAN 1 and LAN 2, while Router 2 interconnects LAN 2 and LAN 3.
- Each station and router is assigned a unique IP address. The IP addresses are labeled Station A, Station B, Station C, and Station D, respectively.

To understand the Topology better, Lets take a simple example of how a data packet would be transmitted from Station A to Station D:

- Station A encapsulates the message in an IP packet and adds an IP packet header.
- Station A consults its routing table to determine the next-hop IP address for Station D. The routing table will show that the next-hop IP address for Station D is Router 1.
- Station A sends the IP packet to Router 1.
- Router 1 consults its routing table to determine the next-hop IP address for Station D. The routing table will show that the next-hop IP address for Station D is Router 2.
- Router 1 forwards the IP packet to Router 2.
- Router 2 forwards the IP packet to Station D.
- Station D receives the IP packet and decapsulates the message.

Note: The above description is the most simplified version. Actual implementation requires consulting ARP tables and pending queues before sending the messages.

Data Packet Transmission When a station sends a data packet, it encapsulates the message in an IP packet and adds an IP packet header. The IP packet header contains the source IP address (the IP address of the sending station) and the destination IP address (the IP address of the receiving station). The station then consults its routing table to determine the next-hop IP address. The routing table is a table that contains the destination network prefix, the next-hop IP address, the network mask, and the network interface through which the next hop can be reached. In order to determine the next hop, It does binary 'AND' operation and accordingly decides the next hop which is the most optimum jump to reach to the destination. There are various other complex algorithms to determine the shortest and most optimum path, the routing tables inherently use these algorithms. The station then sends the IP packet to the next-hop IP address. If the next-hop IP address is on the same LAN as the sending station, the station sends the packet directly to the next-hop station. If the next-hop IP address is on a different LAN, the station sends the packet to the bridge. But, IN order to transmit the Ethernet frame to the next hop, it must know the MAC address of next hop, It can determine the mac address by sending an arp request to the station and the next hop staion will send its mac address which will be stored in arp cache for future use. In between the time of ARP request and reply, The packets stay in pending queue, After arp reply, its removed from pending queue and sent to next hop.The process of forwarding packets is repeated until the packet reaches the destination station.

3 DESIGN

The network emulator is designed in this project by using functionalities of bridge, station and router and discussed each of them as below:

3.1 Bridge Functionality

We have implemented the bridge code such that it executes using two command line arguments lan-name and num-ports to configure its operation.

(1) **lan-name**: It specifies the name of the LAN to which the bridge belongs, enabling proper communication within that LAN.
(2) **num-ports** : It indicates the maximum number of stations (and routers) that can connect to the bridge simultaneously, ensuring efficient resource utilization

- **TCP Socket Creation**: Upon starting the bridge it creates a TCP socket to serve as the primary channel for communication with connected stations and routers. This socket handles the exchange of data and control messages, enabling the bridge to manage connections and handle network traffic.
- **Symbolic Link File Generation**: To make its IP address and port number readily accessible to connected devices, the bridge generates two symbolic link files. These files store crucial information like IP address and port number that enables stations and routers to establish connections with the bridge:
  a. IP Address File: This file contains the real IP address of the machine running the bridge. Stations and routers use this information to determine the network location of the bridge and initiate connection requests. The IP address serves as the unique identifier of the bridge on the network, allowing devices to direct data packets towards it.
  b. Port Number File: This file stores the port number associated with the bridge's TCP socket. Stations and routers use this port number to specify the specific port on which the bridge is listening for incoming connections. The port number acts as a virtual endpoint for the bridge, enabling devices to connect to the correct communication channel.
- **Broadcasting Ethernet Frames** : When the destination MAC address is not found in the MAC table, the emulator broadcasts the Ethernet frame to all other connected sockets. This ensures that the frame reaches all devices in the LAN, which is like a broadcast behavior.
- **Data Frame Reception**: The bridge continuously monitors connected ports for incoming data frames. These frames are the data packets sent by stations or routers, encapsulating the actual data being transmitted and additional information headers such as source , destination MAC addresses and message type .
- **Destination MAC Address**: Upon receiving a data frame, the bridge extracts the destination MAC address from the frame header. This address identifies the intended recipient of the data packet, allowing the bridge to determine the appropriate destination port for forwarding.
- **Self-Learning:** The bridge performs self-learning to maintain an accurate mapping between incoming ports and their corresponding source MAC addresses. When a data frame arrives from a port not yet associated with a MAC address, the bridge stores the mapping between the port and the source MAC address extracted from the frame. This process enables the bridge to learn the network topology and forward data frames effectively.
- **Data Frame Forwarding**: Based on the extracted destination MAC address, the bridge forwards the data frame to the appropriate port. If the destination MAC address is unknown, the bridge broadcasts the frame to all connected ports, allowing the intended recipient to receive it. This ensures that data packets reach their intended destinations, regardless of their current location on the network. If the station is not intended receiver of the message, it ignores it.
- **Performing clean up**: whenever a bridge quits, Its corresponding symbolic links and socket information should also be cleared.

3.2 Station Functionality

The station program in this project utilizes three command-line arguments: interface, routing table and hostname. These arguments specify the files that contain the necessary information for the station to operate effectively.

- **Interface**: This file contains the interface configuration details for the station, including its IP address, network mask, MAC address, and the name of the LAN it is connected to.
- **Routing table:** This file stores the routing table that guides the station in forwarding packets to their intended destinations. It includes the destination network prefix, next-hop IP address, network mask, and the interface through which the next hop can be reached.
- **Hostname**: This file maintains the mapping between host names and their corresponding IP addresses, enabling the station to resolve host names to IP addresses for message transmission.
- **TCP Socket Connection Establishment**: For each LAN specified in the interface file it reads

the corresponding symbolic link file to retrieve the bridge's IP address and port number. It further creates a TCP socket with the socket information, it connects to the bridge using its IP address and port number. If the message received from bridge is "accept", we can be rest assured that the connection is successful. If the message is "reject", the ports are exhausted or we have some issue connecting to the bridge.

- **ARP Resolution**: ARP resolution is done majorly through a three step process. First it will check if the entry for mac address exists in ARP table, IT it is present, it will directly use it. If it is not present, It will broadcast ARP request, the intendent recipient will send back arp reply. We have implemneted the following methods that caters to these functionalities. The fetch_arp_reply method waits for an ARP reply for a given IP address within a specified timeout. Broadcast_arp_request sends an ARP request to discover the MAC address for a given IP. send_arp_reply sends an ARP reply in response to an ARP request.
- **Routing Table Handling**: The routing decisions are based on IP addresses and subnet masks, and these tables are necessary for the emulator to accurately represent how data moves across a network. The method get_next_hop_and_iface extracts the next hop and interface details based on the routing table. Moreover, forward_ip_packet forwards the ip packet to the next hop.
- **ARP Packet Handling**:If the received frame is an ARP packet, the system processes it according to its type, whether it's an ARP request or an ARP reply. This procedure manages the interactions within the ARP protocol, which are crucial for maintaining an accurate ARP cache. The ARP cache is updated based on the information contained in the ARP packet, further ensuring that the cache stays current for the efficient resolution of MAC addresses.
- **ARP Reply**:After receiving an ARP reply it removes the packet from the pending queue for IP packets destined for the MAC address provided in the ARP reply. With the recieved ARP reply packet, we can update the ARP cache for future use. Now that station has mac address, it will send the packet to next hop. This process is repeated until it reaches the destination.
- **Pending queue**:When a station wants to send an IP packet but does not have the corresponding MAC address of the destination IP, it places the packet in this queue. This situation occurs when the ARP cache does not contain a mapping for the required IP address. The pending queue temporarily stores these packets while the station issues an ARP request to resolve the IP to MAC address mapping. Once the ARP reply is received, the station removes the packet from the pending queue and proceeds with the transmission. Pending queues are used to send the packets to an inactive station when it becomes active. In this case, all the pending packets are flooded with the latest message.
- **Retry Mechanism** The retry mechanism in the project involves repeatedly attempting to connect to bridge when the connection fails. If a connection is unsuccessful, possibly due to network issues or unresponsive bridges, the station waits for a two seconds and try again. It tries for 7 iterations before quitting. This mechanism is very important when we want to scale our network by adding new stations and bridges.

3.3 Development

In this section, I will discuss the components I have contributed to for this project, which encompass the self-learning table for bridge, ARP protocol implementation, and packet routing mechanisms.

3.3.1 ***Self Learning.*** : The self-learning feature of a bridge allows it to learn and remember where devices are located on the network. When the bridge gets data from a device with a new MAC address through a port, it keeps track of which port received data from that MAC address. This way, the bridge gets smarter about where to send data in the future, reducing unnecessary data sending to the entire network. This not only makes the network run more smoothly but also helps the bridge to quickly adjust to any changes in how devices are connected.

The self-learning capability of a bridge works by maintaining a MAC address table. When a frame arrives at a bridge, the bridge examines the source MAC

```python
def show_self_learning_table(self):
    current_time = time.time()
    latest_entries = {}

    # Iterate through the MAC table and update the latest entries
    for mac, entry in self.mac_table.items():
        if isinstance(mac, bytes):
            mac_str = ':'.join(format(x, '02x') for x in mac).upper()  # Format the MAC address
        else:
            mac_str = mac.upper()

        # Check if this MAC has a more recent entry
        if mac_str not in latest_entries or entry.timestamp > latest_entries[mac_str].timestamp:
            latest_entries[mac_str] = entry

    # Display the latest entries
    print("Self-Learning Table:\n")
    print("-" * 50)  # Separator line
    print("{:<17}| {:<3}| {:<8}| {}".format("MAC", "SD", "Port", "TTL (sec)"))
    print("-" * 50)  # Separator line

    for mac_str, entry in latest_entries.items():
        ttl = max(60 - (current_time - entry.timestamp), 0)  # Calculate TTL, ensuring it's not negative
        print("{:<17}| {:<3}| {:<8}| {:.2f}".format(mac_str, entry.socket.fileno(), entry.socket.getpeername()[1], ttl))

    print("-" * 50)  # Separator line
```

Figure 2: Self Learning Implementation

address and records it along with the port number the frame was received on. This MAC-to-port mapping is then used for future decisions on where to forward frames. If a frame arrives with a destination MAC address that is not in the table, the bridge broadcasts the frame to all ports except the originating one. Entries in the MAC table are time-stamped and regularly checked; entries that exceed a certain age without being updated are removed. This process allows the bridge to efficiently route frames to their destinations, reducing unnecessary traffic and improving network performance. In Figure 2 ‘show_self_learning_table‘ displays a self-learning table of MAC addresses and associated information. It iterates through the class’s ‘mac_table‘, updates a dictionary with the latest entries for each MAC address, and prints a formatted table. The output includes MAC addresses, source/destination indicators, port information, and Time-To-Live (TTL) values. TTL is calculated based on the timestamp of each entry.

3.3.2 ***ARP Implementation***. : Address Resolution Protocol (ARP) is a fundamental networking protocol that facilitates communication between devices within a local network. Its primary purpose is to map Internet Protocol (IP) addresses to corresponding Media Access Control (MAC) addresses at the data link layer. When a device needs to send data to another device on the same local network and lacks the target’s MAC address, it initiates an ARP request. This request is broadcast to all devices on the network, seeking the MAC address associated with a specific IP address. The device with the destined IP address responds with an ARP reply, providing its MAC address. Devices maintain ARP tables or caches locally, storing these mappings to optimize future communication. ARP is crucial for efficient data transmission within a network, ensuring that devices can communicate seamlessly by resolving the relationship between IP and MAC addresses.

```python
def send_message(self, message, dest_name):
    try:
        # Get the destination IP from the hostnames
        dest_ip = self.hostname_info.get(dest_name)
        if dest_ip is None:
            print("Hostname ",dest_name," not found.")
            return
        sockid = None

        # Determining the next hop based on the routing table
        next_hop_ip, iface_name = self.get_next_hop_details(dest_ip)

        if iface_name not in self.iface_info:
            print("No interface found for destination :",dest_name)
            return

        # determining interface for the ARP request
        iface_structure = self.iface_info[iface_name]

        # Checking the ARP cache for the next hop's MAC address
        if self.arp_cache.get(socket.inet_aton(next_hop_ip)):
            next_hop_mac, timestamp,sock = self.arp_cache.get(socket.inet_aton(next_hop_ip),(None, None,None))
        else:
            next_hop_mac, timestamp,sock = self.arp_cache.get(next_hop_ip,(None, None,None))

        if next_hop_mac is None:
            # Sent ARP request to find out the MAC address of the next hop
            self.broadcast_arp_request(iface_structure, next_hop_ip)
            # Waiting for the ARP reply
            next_hop_mac,sockid = self.fetch_arp_reply(next_hop_ip)

        #the ipaddress can be in string or bytes
        if next_hop_mac is not None:
            if timestamp and (time.time() - timestamp) < 60:
                if isinstance(next_hop_ip,str) and self.arp_cache.get(socket.inet_aton(next_hop_ip)):
                    (a,b,c) = self.arp_cache[socket.inet_aton(next_hop_ip)]
                    self.arp_cache[socket.inet_aton(next_hop_ip)] = (a, time.time(),c)
                elif isinstance(next_hop_ip,bytes) and self.arp_cache.get(next_hop_ip):
                    (a,b,c) = self.arp_cache[next_hop_ip]
                    self.arp_cache[next_hop_ip] = (a, time.time(),c)
                elif isinstance(next_hop_ip,bytes) and self.arp_cache.get(socket.inet_ntoa(next_hop_ip)):
                    (a,b,c) = self.arp_cache[socket.inet_ntoa(next_hop_ip)]
                    self.arp_cache[socket.inet_ntoa(next_hop_ip)] = (a, time.time(),c)
                elif isinstance(next_hop_ip,str) and self.arp_cache.get(next_hop_ip):
                    (a,b,c) = self.arp_cache[next_hop_ip]
                    self.arp_cache[next_hop_ip] = (a, time.time(),c)

            # Constructing and sending the Ethernet frame
            if sockid:
                if next_hop_ip in self.pending_queue:
                    for src_ip, dest_ip, message,sockk in self.pending_queue[next_hop_ip]:

                        success = self.send_frame_to_next_hop(iface_structure, message, dest_ip, next_hop_mac, sockid)
                        if not success:
                            print("[ERROR] Failed to send queued packet to ", next_hop_ip)

                    del self.pending_queue[next_hop_ip]  # Clearing the queue for this IP as we sent it
                success = self.send_frame_to_next_hop(iface_structure, message, dest_ip, next_hop_mac,sockid)

            elif c:
                print("[DEBUG] Fetched entry from ARP cache")

                success = self.send_frame_to_next_hop(iface_structure, message, dest_ip, next_hop_mac,c)
                if next_hop_ip in self.pending_queue:
                    for src_ip, dest_ip, message,sockk in self.pending_queue[next_hop_ip]:
                        success = self.send_frame_to_next_hop(iface_structure, message, dest_ip, next_hop_mac, c)
                        if not success:
                            print("[ERROR] Failed to send queued packet to ", next_hop_ip)

                    del self.pending_queue[next_hop_ip]
                if success:
                    print("[DEBUG] Packet sent to {}. Removing from pending queue.".format(next_hop_ip))
                    self.remove_from_pending_queue(next_hop_ip)
                else:
                    print("[ERROR] Failed to send packet to {}. Packet remains in queue.".format(next_hop_ip))
        else:
            print("Could not find MAC address for ",next_hop_ip)
    except Exception as e:
        print("Error in sending message: , ", e)
```

Figure 3: Sending Message

In the code snippet in FIgure 5, the ARP cache is implemented as a dictionary-like data structure named self_arp_cache. This cache serves as a storage for maintaining mappings between IP addresses and their corresponding MAC addresses, along with additional information such as a timestamp and socket id. The cache is referred to determine the MAC address associated with a given IP address before sending a message. When attempting to obtain the MAC address for a destination IP, the code first checks the ARP cache using ‘self.arp-cache.get()‘. If the cache contains an entry for the target IP, the associated MAC address, timestamp, and socket id are fetched. In case the entry is not found, an ARP request is broadcasted to the network, and the function waits for an ARP reply. Upon successful receipt of the reply, the MAC address is obtained, and the cache is updated with the new mapping, including a timestamp to track the timout of the entry. The ARP cache is a

critical component for optimizing network communication by reducing the need for repeated ARP requests and enhancing the efficiency of message forwarding within the local network.

3.3.3 ***ARP timeout***. I have implemented a timeout functionality for ARP, where entries are removed from the cache once their timer reaches zero. This is essential to prevent the accumulation of outdated entries in the ARP cache, which could otherwise exhaust the router's resources. Additionally, whenever an entry is utilized again, its timer is reset to 60 seconds to ensure its retention in the cache as long as it remains relevant.

```
def broadcast_arp_request(self,iface_details, dest_ip):
    if dest_ip not in self.pending_queue:
        # self.pending_queue[dest_ip] = []
        pass
    # Construct the ARP request packet
    arp_request = ARPPacket(
        sender_mac=iface_details['mac'],  # Your station's MAC address
        sender_ip=iface_details['ip'],    # Your station's IP address
        target_mac='00:00:00:00:00:00',    # Target MAC address is unknown for ARP request
        target_ip=dest_ip,                 # The IP address you want to resolve
        opcode=1                           # Opcode for ARP request
    )
    ethernet_frame = EthernetFrame(

        dst_mac=bytes.fromhex('ff:ff:ff:ff:ff:ff'.replace(':', '')),  # Destination MAC in bytes
        src_mac=bytes.fromhex(iface_details['mac'].replace(':', '')),
        payload=arp_request      # The ARP request is the payload of the Ethernet frame
    )
    ethernet_frame_bytes = ethernet_frame.serialize()
    print("[DEBUG] Broadcasting ARP request for IP: ",dest_ip)
    # Send the Ethernet frame to the network (i.e., broadcast it)
    self.send_raw_bytes_to_network(ethernet_frame_bytes)
```

Figure 4: ARP Broadcast

3.4 ARP Reply

When communication between two stations start at place, and checks the sl table and if doesn't find the mac address of the destination then the station broadcasts an arp response and it waits for an arp reply.

```
def send_arp_reply(self, arp_request, sock, iface_details):
    # Create an ARP reply packet
    arp_reply = ARPPacket(
        sender_mac=iface_details['mac'],        # Your station's MAC address
        sender_ip=iface_details['ip'],          # Your station's IP address
        target_mac=arp_request.sender_mac,      # The MAC address of the requester
        target_ip=socket.inet_ntoa(arp_request.sender_ip),  # The IP address of the requester
        opcode=2                                # Opcode for ARP reply
    )

    dst_mac_bytes = arp_request.sender_mac if isinstance(arp_request.sender_mac, bytes) else ARPPacket.mac_to_bytes(arp_request.sender_mac)
    src_mac_bytes = iface_details['mac'] if isinstance(iface_details['mac'], bytes) else ARPPacket.mac_to_bytes(iface_details['mac'])
    ethernet_frame = EthernetFrame(
        dst_mac=dst_mac_bytes,          # The requester's MAC address
        src_mac=src_mac_bytes,          # Your station's MAC address
        payload=arp_reply               # The ARP reply bytes
    )

    ethernet_frame_bytes = ethernet_frame.serialize()
    sock.send(ethernet_frame_bytes)
```

Figure 5: ARP Reply

The 'send-arp-reply' function generates and transmits an ARP (Address Resolution Protocol) reply in response to an ARP request. It constructs an ARP reply packet with essential information, including the MAC and IP addresses of both the responding station and the requester. The ARP reply is then encapsulated in an Ethernet frame, specifying the destination MAC as the requester's address and the source MAC as the station's own address. This Ethernet frame is serialized into bytes and sent over the provided socket, completing the process of responding to an ARP request. The function plays a pivotal role in dynamic network communication by facilitating the resolution and updating of MAC address mappings in ARP caches across connected devices within the local network.

3.4.1 ***Packet Forwarding***. : The forward_ip_packet method serves as a crucial component in the project for routers. It is responsible for forwarding IP packets within a network. It begins by determining the next hop and corresponding interface based on the destination IP address. The function manages the ARP cache, checking and updating entries to obtain the MAC addresses of both the sender and the next hop. If necessary, ARP requests are broadcasted, and the function waits for ARP replies to complete MAC address resolution. Once the MAC address of the next hop is obtained, the method sends an Ethernet frame with the IP packet payload to facilitate forwarding. The code also handles pending messages in the queue, ensuring that any messages waiting for ARP resolution are processed and sent after successful resolution. In case of errors during the forwarding process, the IP packet is added to the pending queue for subsequent processing, contributing to robust and efficient network communication.

```
def forward_ip_packet(self, ip_packet,sock):
    try:
        if not isinstance(ip_packet.dest_ip, bytes):
            next_hop_ip, iface_name = self.get_next_hop_and_iface(ip_packet.dest_ip)
        else:
            next_hop_ip, iface_name = self.get_next_hop_and_iface(socket.inet_ntoa(ip_packet.dest_ip))
        if next_hop_ip:
            if self.arp_cache.get(ip_packet.src_ip):
                mac_sender, timestamp,sock = self.arp_cache.get(ip_packet.src_ip,(None, None,None))
                self.arp_cache[ip_packet.src_ip] = (mac_sender, time.time(), sock)
            elif self.arp_cache.get(socket.inet_aton(ip_packet.src_ip)):
                mac_sender, timestamp,sock = self.arp_cache.get(socket.inet_aton(ip_packet.src_ip),(None, None,None))
                self.arp_cache[socket.inet_aton(ip_packet.src_ip)] = (mac_sender, time.time(), sock)
            if self.arp_cache.get(socket.inet_aton(next_hop_ip)):
                next_hop_mac, timestamp,sock = self.arp_cache.get(socket.inet_aton(next_hop_ip),(None, None,None))
                self.arp_cache[socket.inet_aton(next_hop_ip)] = (next_hop_mac, time.time(), sock)
            else:
                next_hop_mac, timestamp,sock = self.arp_cache.get(next_hop_ip,(None, None,None))
                self.arp_cache[next_hop_ip] = (next_hop_mac, time.time(), sock)
            if not next_hop_mac:
                self.broadcast_arp_request(self.iface_info[iface_name], next_hop_ip)
                next_hop_mac,sock = self.wait_for_arp_reply(next_hop_ip)
            if next_hop_mac:
                self.send_ethernet_frame_to_next_hop(self.iface_info[iface_name], ip_packet.payload, socket.inet_ntoa(ip_packet.dest_ip)
                                                     if isinstance(ip_packet.dest_ip,bytes) else ip_packet.dest_ip, next_hop_mac,sock,newsock=None,src_ip = ip_packet.src_ip)

                if next_hop_ip in self.pending_queue:
                    print("IN PENDING QUEUE in handling messages")
                    for src_ip, dest_ip, message,sock in self.pending_queue[next_hop_ip]:
                        print("PQ: ", self.pending_queue[next_hop_ip])
                        success = self.send_ethernet_frame_to_next_hop(self.iface_info[iface_name], message, dest_ip, next_hop_mac, sock)
                        if not success:
                            print("[ERROR] Failed to send queued packet to ", next_hop_ip)
                    del self.pending_queue[next_hop_ip]  # Clear the queue for this IP
            else:
                print("No ARP entry for next hop IP: ",next_hop_ip)
        else:
            print("No route to host for IP: ",socket.inet_ntoa(ip_packet.dest_ip) if isinstance(ip_packet.dest_ip,bytes) else ip_packet.dest_ip)
    except Exception as e:
        print("Error in forwarding ip packet: , ", e)
        if next_hop_ip not in self.pending_queue:
            self.pending_queue[next_hop_ip] = []
        self.pending_queue[next_hop_ip].append((self.iface_info[iface_name]['ip'], ip_packet.dest_ip, ip_packet.payload,sock))
```

Figure 6: IP Packet Forwarding

3.4.2 ***Determining next-hop***. : The get_next_hop_and_iface method is designed to determine the next hop IP address and the associated network interface for forwarding an IP packet based on the destination IP address.

This method iterates through the routing table, which contains information about network destinations, netmasks, gateways, and associated interfaces. It converts the destination IP address, netmask, and network prefix into integers for bitwise operations. The method checks if the destination IP address matches any network prefix in the routing table by performing a bitwise AND operation. If a match is found, it evaluates whether a gateway is specified. If a gateway is present, it returns the gateway and the associated interface. If the gateway is '0.0.0.0', indicating a direct route, it returns the destination IP address and the corresponding interface. If no match is found in the routing table, the method defaults to the last entry in the routing table, representing a default route, and returns the default gateway and interface.

```
def get_next_hop_and_iface(self, dest_ip):
    for route in self.routing_table:
        dest_addr = int.from_bytes(socket.inet_aton(dest_ip), 'big')
        netmask = int.from_bytes(socket.inet_aton(route['netmask']), 'big')
        network_prefix = int.from_bytes(socket.inet_aton(route['destination']), 'big')
        if dest_addr & netmask == network_prefix:
            if route['gateway'] == '0.0.0.0':
                return dest_ip, route['iface']
            return route['gateway'], route['iface']
    default_route = self.routing_table[-1]
    return default_route['gateway'], default_route['iface']
```

Figure 7: Determine next-hop

4 EVALUATION

This project has successfully implemented all the requirements and functionalities of bridge, router and station and below are the screenshots for the same.

```
show sl
Self-Learning Table:

-------------------------------------------------
MAC               | SD | Port    | TTL (sec)
-------------------------------------------------
00:00:0C:04:52:67| 5  | 49090   | 39.76
08:00:20:02:97:66| 4  | 46840   | 39.76
-------------------------------------------------
```

Figure 8: show sl

5 CONCLUSION

In summary, the network emulator we discussed is a great tool for learning, researching and debugging on networks. It uses python to imitate complex networks,

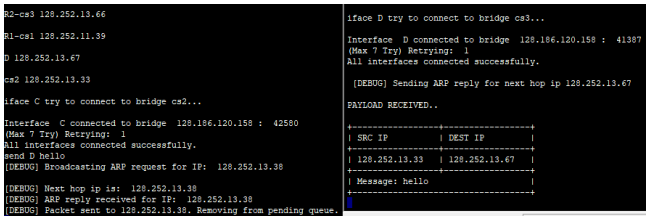

Figure 9: send message to station

```
show arp
+----------------+--------------------+-------------+
|      IP        |         MAC        |  Time Left  |
+----------------+--------------------+-------------+
| 128.252.13.33 | 00:00:0c:04:52:33 | 54.20 sec    |
| 128.252.13.69 | 00:00:0c:04:52:69 | 54.20 sec    |
+----------------+--------------------+-------------+
```

Figure 10: show arp

```
show pq
+-------------------------------------------------+
+-------------------------------------------------+
| next hop              | # PKTS Waiting          |
+-------------------------------------------------+
+-------------------------------------------------+
| 128.252.13.67         | 1                       |
+-------------------------------------------------+
|                    IP PACKET                    |
+-------------------------------------------------+
| MESSAGE      | SRC IP          | DEST IP        |
| Hii          | 128.252.13.66   | 128.252.13.67  |
+----------------------+--------------------------+
```

Figure 11: show pq

which is cheaper and more flexible than using real network. This emulator helps people understand how networks work by letting them see and work with different network setups, like how LANs, bridges, and routers connect. The provided software adheres to all protocols and is efficient to be used as a substitute of actual network for demonstration purposes.

```
show iface
-------------------------------------------------------------------------
Interface  IP Address      Netmask         MAC Address       LAN
-------------------------------------------------------------------------
R1-cs2     128.252.13.35   255.255.255.224 08:00:20:02:97:AB cs2
R1-cs1     128.252.11.39   255.255.255.0   08:00:20:75:41:85 cs1
-------------------------------------------------------------------------
```

Figure 14: show iface

```
show host
--------------------------------
Hostname         IP Address
--------------------------------
Acs2             128.252.13.40
A                128.252.11.23
D                128.252.13.67
E                128.252.13.69
R2-cs2           128.252.13.38
C                128.252.13.33
R1-cs1           128.252.11.39
R2-cs3           128.252.13.66
B                128.252.11.38
R1-cs2           128.252.13.35
Acs1             128.252.11.23
--------------------------------
```

Figure 15: show host

```
PAYLOAD RECEIVED..

+------------------+------------------+
| SRC IP           | DEST IP          |
+------------------+------------------+
| 128.252.13.40    | 128.252.13.69    |
+------------------+------------------+
| Message: Hi test from a             |
+-------------------------------------+

PAYLOAD RECEIVED..

+------------------+------------------+
| SRC IP           | DEST IP          |
+==================+==================+
| 128.252.13.66    | 128.252.13.69    |
+------------------+------------------+
| Message: Hiii                       |
+-------------------------------------+

PAYLOAD RECEIVED..

+------------------+------------------+
| SRC IP           | DEST IP          |
+------------------+------------------+
| 128.252.13.66    | 128.252.13.69    |
+------------------+------------------+
| Message: Hey man                    |
+-------------------------------------+

PAYLOAD RECEIVED..

+------------------+------------------+
| SRC IP           | DEST IP          |
+------------------+------------------+
| 128.252.13.66    | 128.252.13.69    |
+------------------+------------------+
| Message: Hi all                     |
+-------------------------------------+
```

Figure 12: show pq after reconnect

```
show rtable
-----------------------------------------------------------
Destination     Gateway         Netmask         Interface
-----------------------------------------------------------
128.252.11.0    0.0.0.0         255.255.255.0   B
128.252.13.0    128.252.11.39   255.255.255.0   B
0.0.0.0         128.252.11.39   0.0.0.0         B
-----------------------------------------------------------
```

Figure 13: show rtable