

The Art and Science of Emulated Networks

Shivam Agnihotri

sa22bb@fsu.edu

Florida State University

Tallahassee, Florida, USA

ABSTRACT

The Network Emulator presented in this project offers a versatile platform for simulating and exploring network functionalities, encompassing bridges, routers, and stations. Key features include a well-structured design, symbolic link handling, and efficient time-out management for asynchronous operations. The emulator facilitates the initialization of network entities, supports dynamic connection establishment and retries, and enables users to inspect and modify the network state through user-friendly commands. Through a carefully designed architecture, the Network Emulator serves as a valuable tool for both educational and research purposes, providing users with a hands-on experience to enhance their understanding of networking concepts.

1 INTRODUCTION

In the ever-evolving landscape of computer networking, hands-on experience is paramount for a comprehensive understanding of network protocols, packet forwarding, and the intricate interplay between various networking devices. This project introduces a Network Emulator, a versatile tool designed to simulate and explore the functionalities of bridges, routers, and stations within a network environment.

As the demand for practical networking knowledge continues to rise, the Network Emulator bridges the gap between theoretical concepts and real-world application. By creating a simulated network environment, users can experiment with different configurations, observe packet routing, and gain insights into the intricacies of network communication.

This project delves into the creation and functionality of the Network Emulator, outlining its design principles, key features, and the benefits it offers to users. With a focus on ease of use and educational value, the emulator provides an interactive platform for users to engage with networking concepts and experiment with various scenarios. The subsequent sections will detail the core functionalities of the emulator, highlighting its architecture, design considerations, and practical applications.

2 OVERVIEW

The network model adopted in this project encompasses broadcast Ethernet LANs, where stations and routers within a LAN share a common broadcast physical medium. In this LAN setup, when a station sends a packet, it is broadcasted to and received by all other stations in the same LAN.

The LAN topology considered is a star-wired configuration where each station connects directly to a central node known as a transparent bridge or switch. A bridge with 'n' ports can accommodate 'n' stations. Upon connection, a bridge broadcasts an Ethernet data frame received from one station to all other connected stations, re-transmitting the frame through all ports except the receiving

one. This enables every station to receive transmissions from any other station connected to the bridge.

Each frame includes a header with source and destination (next-hop) MAC addresses. The bridge, through self-learning, gradually identifies the location of stations on the LAN and forwards frames only to the necessary LAN segments. Stations accept data frames addressed to them and discard frames not intended for their address.

IP routers interconnect LANs to create an IP internetwork, allowing communication between any pair of stations using the Internet Protocol. Stations are assigned globally unique IP addresses, and IP packets are transmitted by adding an IP packet header containing source and destination IP addresses. The hop-by-hop datagram delivery model is employed, where routers and stations are responsible for forwarding IP packets to their next hops based on forwarding tables.

2.1 Network Model Overview

The project's network model integrates broadcast Ethernet LANs and IP networks, with a particular focus on the distinctive star-wired LAN topology. Within this model, stations and routers in a LAN are connected to a common broadcast physical medium, allowing the broadcast of packets from one station to all others within the same LAN. The star-wired LAN topology employs a central node, termed a transparent bridge, to which each station directly connects through dedicated ports.

This network configuration supports efficient transmission of data frames, and the transparent bridges engage in a self-learning process. As data frames are transmitted, the bridges gradually map the location of stations within the LAN. This dynamic mapping enhances the forwarding efficiency of the bridges, ensuring that frames are delivered only to the necessary segments of the LAN. The star-wired LAN topology, with its emphasis on transparent bridges and self-learning mechanisms, serves as a foundational element in the emulation of broadcast Ethernet LANs and IP networks within the network emulator project.

2.1.1 Broadcast Ethernet LANs. Shared Broadcast Medium: Utilizes a shared physical medium where stations can transmit and receive data frames. This shared medium employs a common communication channel for all devices within the LAN. Broadcasting Mechanism: The broadcasting mechanism ensures that when a station transmits a data frame, it is disseminated to all other stations in the LAN. This is achieved through the shared broadcast medium, creating a one-to-many communication paradigm.

2.1.2 Star-Wired LAN Topology. Central Node - Transparent Bridge: The transparent bridge, acting as the central node, is implemented as a software entity. It manages the data frame forwarding process and maintains the self-learning database to optimize future transmissions. Individual Ports: Each station connects to the bridge through

individual ports, simulated as distinct channels in the software. The number of ports is configurable, determining the maximum number of stations that can be connected to a single bridge.

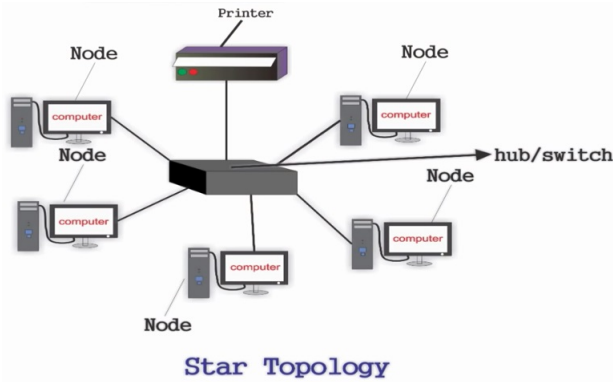


Figure 1: Star Topology

2.1.3 Self-Learning Process. Bridge Functionality: The transparent bridge, upon receiving a data frame, analyzes the source MAC address and the incoming port. It dynamically updates its self-learning database, associating the MAC address with the corresponding port to optimize future frame forwarding. Dynamic Mapping: The dynamic mapping process ensures that the bridge evolves its understanding of the network over time. If a station becomes inactive for a predefined period, its entry is removed from the database to maintain accuracy.

2.1.4 IP Networks and Routers. Interconnecting LANs: IP routers act as software entities connecting different LANs within the emulation. They use routing tables to determine the next-hop for IP packets, facilitating inter-LAN communication. Global Unique IP Addresses: Stations and router interfaces are assigned globally unique IP addresses, ensuring the uniqueness of each device in the simulated network. This enables precise identification during IP packet transmission. Hop-by-Hop Datagram Delivery: The hop-by-hop datagram delivery model involves routers and stations making forwarding decisions based on routing tables. Each device is responsible for delivering IP packets to their respective next-hop destinations.

2.1.5 Example Topology. Three LANs and Two Routers: The example topology showcases three LANs interconnected by software-based bridges. Two routers facilitate communication between different LANs in the emulation. Packet Forwarding Example: Describes how IP packets are forwarded from source to destination within a LAN and across routers, emphasizing the role of the IP addressing scheme and routing tables in the process.

2.2 Network Emulation

The network emulator pursues the emulation of physical network components within a software environment, mimicking their functionalities. Leveraging TCP socket connections to emulate physical links, the emulator embodies bridges, stations, and routers as software entities. This section provides a deeper dive into the

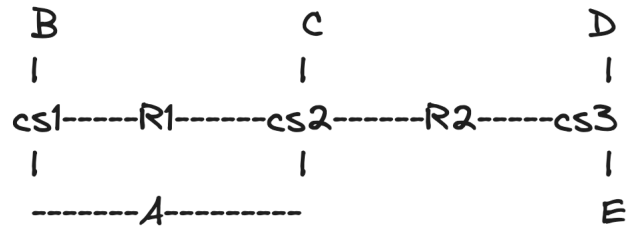


Figure 2: A simple network topology

implementation details, highlighting key components such as the forwarding table, command interface, concurrent task handling.

2.2.1 Implementation Using Socket Programming.

- **Client-Server Paradigm:** The network emulator is designed using the client-server socket programming paradigm, where bridges function as servers, and stations (including routers) act as clients. This paradigm facilitates communication between network components through TCP socket connections.
- **LAN Identification:** Bridges and stations are identified by a LAN name (lan-name) passed as a command-line argument. Each LAN has a unique name, ensuring proper identification within the network model.

2.2.2 IP and ARP Functionality.

- **IP Layer Support:** Stations support two layers: IP layer and MAC layer. The IP layer encapsulates messages into IP packets, consulting the forwarding table for the next-hop IP address. The MAC layer, through the ARP protocol, identifies MAC addresses associated with IP addresses.
- **Address Resolution Protocol (ARP):** Stations employ ARP to map IP addresses to MAC addresses. ARP requests are sent when a mapping is not present, and ARP replies update the ARP cache. The pending queue manages IP packets awaiting ARP resolution.

2.2.3 Forwarding Table Implementation.

- **Routing Table Configuration:** Stations and routers maintain a forwarding table, which dictates how IP packets should be forwarded. The forwarding table is loaded from a configuration file and contains entries specifying destination network prefixes, next-hop IP addresses, network masks, and the outgoing network interface.
- **Destination Network Prefix:** Specifies the IP address range of the destination network.
- **Next-Hop IP Address:** Indicates the next router or destination IP address.
- **Network Mask:** Defines the range of IP addresses covered by the entry.
- **Outgoing Network Interface:** Specifies the network interface through which the next hop can be reached.

2.2.4 Command Interface and Management.

- **User Commands:** The network emulator incorporates commands to display routing tables, ARP cache, name/address

mappings, interface tables on stations, and MAC address/port mappings on bridges. These commands provide insights into the internal state of the emulator.

- **Dynamic Configuration:** The emulation allows connecting and disconnecting stations and routers at any time, providing a dynamic environment for testing and simulation.
- **Clean-Up Procedures:** When a station or bridge terminates, clean-up procedures are initiated, including the closure of TCP socket connections and the release of allocated resources.

2.2.5 Concurrent Handling of Tasks.

- **Multiplexing with select():** To handle multiple simultaneous events such as connection setup requests, data frame arrivals, and user input, the select() function is employed. This enables I/O multiplexing, ensuring efficient monitoring of various events.
- **Concurrent Data and Control Handling:** Stations manage concurrent tasks, including sending/receiving messages, accepting messages, and handling control messages such as ARP. The concurrency allows the emulation to simulate real-time network behaviors.

2.2.6 Transparent Bridge (Server) Implementation.

- **Command-Line Arguments:** Bridges accept two command-line arguments: lan-name and num-ports. The lan-name uniquely identifies the LAN, while num-ports specifies the maximum number of stations that can be connected to the bridge.
- **Socket Creation and Configuration:** Upon initialization, a bridge creates a TCP socket and symbolic link files (.lan-name.addr and .lan-name.port) to store its IP address and port number. This information enables stations to connect to the bridge.
- **Connection Handling:** Bridges wait for connection setup requests from stations. Upon receiving a request, the bridge establishes a separate TCP socket connection for each station. If the connection is successful, the bridge returns "accept"; otherwise, it returns "reject."

2.2.7 Station (Client) Implementation.

- **Command-Line Arguments:** Stations require three command-line arguments: interface, routingtable, and hostname. These files contain interface information, routing tables, and IP address mappings, respectively.
- **Initialization:** Stations load configuration files, including hostname, routingtable, and interface, to initialize data structures. They connect to LANs specified in the interface file, utilizing TCP socket connections to bridges.
- **Non-Blocking Connection:** To prevent stations from waiting indefinitely for a reply from bridges, non-blocking reading on the socket connection is employed. Stations attempt to read for a preset number of times, declaring rejection if unsuccessful after multiple tries.

2.2.8 Router Implementation.

- **Router as a Station:** Routers share similarities with stations but connect to more than one bridge, facilitating packet

forwarding between LANs. They are distinguished by the -route command-line argument.

- **Multi-Bridge Connection:** Routers connect to multiple bridges based on the information provided in the interface file, allowing them to participate in the emulation across different LANs.

3 DETAILED DESIGN AND DEVELOPMENT

The detailed design and implementation of the network emulator constitute the bedrock of its functionality, requiring meticulous attention to various technical aspects. In this section, we explore the intricate details of the network emulator's inner workings, shedding light on the underlying technologies, protocols, and methodologies that drive its performance.

3.1 Packet Forwarding and Broadcasting in Bridge

The packet forwarding and broadcasting functionality in the bridge form the backbone of communication within the network. This involves the examination of incoming packets, determining the destination MAC address, and making informed decisions on whether to forward the packet to a specific interface or broadcast it to all connected stations.

3.1.1 Forwarding Table. The bridge maintains a forwarding table that correlates MAC addresses with corresponding interfaces. This table is dynamically updated as the bridge learns about the association between MAC addresses and interfaces.

3.1.2 Broadcasting. When the destination MAC address is not found in the forwarding table, the packet is broadcasted to all connected interfaces. This ensures that the packet reaches its intended destination, even if the specific interface is unknown.

```
if (!found)
{
    for(int j = 0; j < max_ports; ++j)
    {
        if (clients[j].port && j != i && j != servfd)
        {
            send(j, &ether_pkt, sizeof(EtherPkt), 0);
            if (ether_pkt.type == TYPE_IP_PKT)
            {
                cout << "Broadcasting_" << sizeof(IP_PKT)
                    << "_byte_ip_pkt\n";
                send(j, &IP_pkt, sizeof(IP_PKT), 0);
            }
            else if (ether_pkt.type == TYPE_ARP_PKT)
            {
                cout << "Broadcasting_" << sizeof(ARP_PKT)
                    << "_byte_arp_pkt\n";
                send(j, &ARP_pkt, sizeof(ARP_PKT), 0);
            }
        }
    }
}
```

3.2 Station Connection and Retries

The process of station connection involves the establishment of a connection between a station and the bridge over a LAN interface.

Given the dynamic nature of network environments, mechanisms for handling connection retries are crucial for robust and reliable network emulation.

3.2.1 Socket Establishment. Stations initialize a socket for connecting to the bridge, with LAN-specific parameters. The connection process involves attempts to establish a link with the bridge.

```
// Set up the client socket
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd == -1) {
    perror("Error creating socket");
    exit(EXIT_FAILURE);
}

struct sockaddr_in ma;
// Set up the sockaddr_in structure
ma.sin_family = AF_INET;
ma.sin_port = htons(atoi(lanport_num));
struct hostent *host = gethostbyname(lan_address);
memcpy(&ma.sin_addr, host->h_addr, host->h_length);
```

3.2.2 Non-Blocking Mode. To enhance efficiency and responsiveness, the station sets its socket to non-blocking mode. This allows the station to continue other operations while awaiting a connection, preventing potential bottlenecks. Technical considerations include the use of the `fcntl` system call to set and restore the non-blocking mode.

3.2.3 Connection Retry. A defined number of connection attempts are made, and if unsuccessful, the station implements a retry mechanism. This approach provides resilience against transient network issues.

```
// Attempt to connect for a preset number of times
const int max_attempts = 5;
const int wait_time_seconds = 2;

for (int attempt = 0; attempt <
    max_attempts; ++attempt) {
    // If connection fails...
    if (connect(sockfd, (struct sockaddr*)&ma,
        sizeof(ma)) == -1) {
        cerr << "Could not connect to server. Retrying ... \n";
        std::this_thread::sleep_for
            (std::chrono::seconds(wait_time_seconds));
    } else {
        // Connection successful
        connection_accepted = true;
        break;
    }
}
```

3.3 Set and Restore Blocking Mode

The ability to set and restore the blocking mode of sockets is essential for managing network operations effectively. This functionality ensures that the network emulator can adapt its behavior based on specific operational requirements.

3.3.1 set NonBlocking Function. A function is designed to set the socket to non-blocking mode using the `fcntl` system call. This functionality enables asynchronous socket operations, preventing the system from blocking during certain operations.

3.3.2 restore Blocking Function. Another function is implemented to restore the socket to its original blocking mode. This is particularly useful when reverting to synchronous operations is necessary, ensuring compatibility with different stages of the network emulator's execution.

```
// Function to set socket to non-blocking mode
void setNonBlocking(int sockfd) {
    int socket_flag = fcntl(sockfd, F_GETFL, 0);
    fcntl(sockfd, F_SETFL, socket_flag | O_NONBLOCK);
}
```

```
// Set socket to non-blocking
setNonBlocking(sockfd);
```

```
// Function to restore the blocking mode of the socket
void restoreBlocking(int sockfd, int socket_flag) {
    fcntl(sockfd, F_SETFL, socket_flag);
}
```

3.4 Handling ARP Requests and Responses in Station

The handling of ARP (Address Resolution Protocol) requests and responses is a fundamental aspect of the network emulator. Stations need to respond to ARP requests and update their ARP cache accordingly.

3.4.1 ARP Request Handling. Upon receiving an ARP request, a station responds with its MAC address. This response is critical for building and maintaining the ARP cache.

3.4.2 ARP Response Handling. Stations process ARP responses to update the ARP cache with mappings of IP addresses to MAC addresses. This dynamic updating ensures that the emulator has an accurate and up-to-date ARP cache.

```
else if (ether_packet.type == TYPE_IP_PKT)
{
    // cout << "IP packet\n"; // <<
    IPtostring(IP_pkt.dstip) << endl;
    recv(sockfd, &IP_pkt, sizeof(IP_PKT), 0);
    messageReceived = 0;
    string sourceStationName;
    for (int j = 0; j < ifaces.size(); j++)
    {
        if (IP_pkt.dstip == ifaces[j].ipaddr)
        {
            cout << "Message of size " << IP_pkt.length
                << " received from station " << sourceStationName;
            cout << ">>>" << IP_pkt.data << endl;
            displayIPPacket(IP_pkt.dstip, IP_pkt.srcip, IP_pkt.data);
            messageReceived = 1;
        }
    }
}
```

3.5 Handling Pending Queue

The introduction of a pending queue addresses scenarios where the destination MAC address for an IP packet is not immediately available. This queue ensures that IP packets are appropriately handled once the necessary ARP resolution occurs.

3.5.1 Queue Management. IP packets awaiting resolution of the destination MAC address are placed in a pending queue. As ARP

responses are received, the pending queue is checked, and corresponding packets are sent to their intended destinations.

```
// look up in ARP cache
if (GetMAC(IP_pkt.nexthop, ARP_cache, ether_pkt))
{
    ether_pkt.type = TYPE_IP_PKT;
}
// if in ARP cache, send IP
// otherwise, send ARP, add to pending queue
else
{
    cout << "MAC not in ARP cache, put IP packet in queue";
    pending_queue.push_back(IP_pkt);

    ether_pkt.type = TYPE_ARP_PKT;

    ARP_pkt.op = ARP_REQUEST;
    ARP_pkt.srcip = temp_i.ipaddr;
    ARP_pkt.dstip = IP_pkt.nexthop;
    strncpy(ARP_pkt.srcmac, ether_pkt.src, 18);
}
send(temp_i.sockfd, &ether_pkt, sizeof(EtherPkt), 0);
```

3.6 ARP Cache Entry Implementation (Get MAC Address)

The ARP cache plays a crucial role in efficiently mapping IP addresses to MAC addresses. The implementation of ARP cache entries, particularly the mechanism to retrieve MAC addresses, is essential for seamless communication within the network emulator.

3.6.1 GetMAC Function. A dedicated function is designed to iterate through the ARP cache and retrieve the MAC address corresponding to a given IP address. If the entry is not found, the MAC address is set to 'x' (empty).

3.7 Figuring Out NextHop in Station

Determining the next hop for packet forwarding is a key functionality in stations, especially in routing scenarios. This involves examining the destination IP address and consulting the routing table for the appropriate next hop.

3.7.1 getNextHop Function. This function iterates through the routing table to find the entry matching the destination IP address. It returns the next hop IP address, facilitating effective packet forwarding within the network.

3.8 Forwarding IP Packet in Routers

Routers play a pivotal role in forwarding IP packets between different network segments. Understanding how routers determine the next hop and effectively forward IP packets is critical for the overall functionality of the network emulator.

3.8.1 IP Packet Forwarding. The router determines the next hop for an IP packet by consulting the routing table. The IP packet is then forwarded to the next hop by updating the Ethernet packet and sending it to the appropriate interface.

```
if (!strcmp(argv[1], "-route") && !messageReceived)
{
    // Router functionality
    cout << "Forwarding IP packet.\n";
```

```
// Get next hop IP
temp_route = getNextHop(rtable, IP_pkt.dstip);
if (temp_route.nexthop != 0)
    IP_pkt.nexthop = temp_route.nexthop;
else
    IP_pkt.nexthop = IP_pkt.dstip;

cout << "Destination IP = " <<
    IPtostring(IP_pkt.dstip) << endl
    << "Sending to " << IPtostring(IP_pkt.nexthop)
    << endl;
for (int i = 0; i < ifaces.size(); i++)
    if (!strcmp(temp_route.ifacename,
        ifaces[i].ifacename, 32))
        temp_interface = ifaces[i];

strcpy(ether_packet.src, temp_interface.macaddr, 18);
```

4 EVALUATION AND ILLUSTRATION

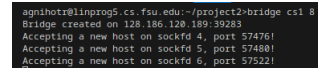


Figure 3: Bridge Connection

The illustration Figure 3 reflects real-time interactions between the bridge and stations, providing insights into the emulator's ability to facilitate seamless communication establishment.

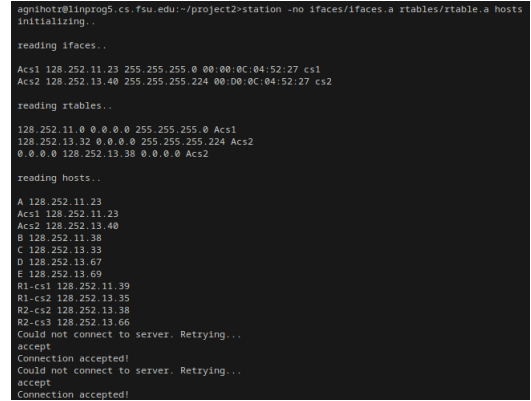


Figure 4: Station Initialization

The figure 4 showcases the loading of configuration files by stations, emphasizing the flexibility of the emulator in adapting to diverse network setups.

Each station initializes its interfaces, setting the stage for subsequent communication. This step is crucial for stations to be active participants in the network emulation.

The figure 5 demonstrates the bridge's ability to dynamically adapt to changing network conditions. As stations communicate, the bridge learns and updates its internal table for efficient packet forwarding.

The self-learning process enhances the efficiency of the emulator by minimizing unnecessary broadcasts. The bridge intelligently

```
Received ethernet header
Received 72 byte ARP frame
6 -> 08:00:20:02:97:AB
Broadcasting 72 byte arp_pkt
Broadcasting 72 byte arp_pkt
Broadcasting 72 byte arp_pkt
Received ethernet header
Received 72 byte ARP frame
7 -> 08:00:20:75:41:88
Received ethernet header
Received 32000 byte IP frame
show s1
*****
Bridge self-learning table:
MAC address      Port Socket TTL
*****
08:00:20:02:97:AB 44044 6 91
08:00:20:75:41:88 44046 7 91
*****
```

Figure 5: Self Learning Table

```
Connection accepted!
send B hi i am b
Destination IP = 128.252.11.38
Next hop interface: Ac11
Next hop IP: 128.252.11.38
IP packet destination IP: 128.252.11.38
IP packet source IP: 128.252.11.23
IP packet data: hi i am b
Got ARP response, updating pending queue
NextHop = 128.252.11.38 128.252.11.38
[]

R2-cs3 128.252.13.66
Could not connect to server. Retrying...
accept
Connection accepted!
Got ARP request
128.252.11.23->128.252.11.38
Message of size 10 received from station
>>> hi i am b
*****
IP Packet
*****
dstip: 128.252.11.38      srcip: 128.252.11.23
data: hi i am b
*****
```

Figure 6: Send and Receive Messages

forwards data frames based on its evolving understanding of station locations.

The figure 6 visually represents the process of sending and receiving messages within the emulator. It highlights the communication channels between stations and the seamless data transfer mechanism.

The illustration captures the flow of messages between stations, emphasizing the emulator’s ability to simulate data exchange in a networked environment. The depiction of message exchange underscores the real-time nature of communication, showcasing the responsiveness and accuracy of the emulator.

```
*****
show arp
*****
ARP Cache:
*****
128.252.13.48  08:00:0C:04:52:27  24
*****
```

Figure 7: Show ARP

The Figure 7 provides a snapshot of the Address Resolution Protocol (ARP) functionality within the emulator. It displays the mapping of IP addresses to MAC addresses. Address Resolution: The figure illustrates the emulator’s capacity to resolve IP addresses to corresponding MAC addresses, a critical aspect of network communication. Resource Visualization: Users can visually inspect the ARP table, gaining insights into the network’s resource utilization and the efficiency of address resolution.

Figure 8 visually captures the process of IP forwarding within the router, showcasing the router’s role in directing data packets between different networks.

Inter-Network Routing: The illustration highlights the emulator’s ability to simulate router functionality, facilitating communication between stations in distinct network segments. Routing

```
Got ARP request
128.252.13.35->128.252.13.38
Forwarding IP packet.
Destination IP = 128.252.13.69
Sending to 128.252.13.69
MAC address not in ARP cache, put IP packet in queue
Sent ARP request!
Got ARP response, updating pending queue
NextHop = 128.252.13.69 128.252.13.69
One entry in ARP-cache timed out
One entry in ARP-cache timed out
```

Figure 8: Router IP Forwarding

Optimization: Users can assess the efficiency of IP forwarding, gauging the emulator’s effectiveness in optimizing data packet routing across networks.

```
*****
pending queue
*****
Destination IP: 128.252.13.69
Next Hop IP: 128.252.13.38
Data hi
*****
```

Figure 9: Show Pending Queue

Figure 9 offers a glimpse into the emulator’s queue management, displaying pending data packets awaiting processing.

Queue Visualization: Users can observe the status of the pending queue, gauging the emulator’s ability to manage data packet processing efficiently. Resource Utilization: The figure provides insights into the emulator’s workload and processing capacity, aiding users in optimizing network configurations.

```
*****
show host
*****
Hosts:
*****
A 128.252.11.23
Ac11 128.252.11.23
Ac12 128.252.13.48
B 128.252.11.38
C 128.252.13.23
D 128.252.13.67
E 128.252.13.69
H1-cs1 128.252.11.39
H1-cs2 128.252.13.35
H2-cs2 128.252.13.98
H2-cs3 128.252.13.66
*****
```

Figure 10: Display Hosts

Figure 10 showcases the ability to display information about hosts within the network, including their statuses and configurations.

Host Information: Users can gather comprehensive details about hosts, facilitating network monitoring and troubleshooting. Configurational Transparency: The illustration emphasizes the emulator’s transparency in presenting host information, contributing to a user-friendly interface.

```
*****
show iface
*****
iface_list
*****
Ac11 128.252.11.23 255.255.255.0 08:00:0C:04:52:27
Ac12 128.252.13.48 255.255.255.224 08:00:0C:04:52:27
*****
```

Figure 11: Display Interfaces

Figure 11 provides a visual representation of the interfaces configured within the emulator.

Interface Configuration: Users can assess the status and configurations of interfaces, aiding in the identification of potential issues and network optimizations. Topology Understanding: The figure contributes to a better understanding of the network topology, showcasing the interconnected interfaces and their states.

```
show table
table
-----
128.252.13.0      0.0.0.0      255.255.255.0   Ac11
128.252.13.32    0.0.0.0      255.255.255.224  Ac12
0.0.0.0          128.252.13.30 0.0.0.0         Ac12
-----
```

Figure 12: Display Routing Table

Figure 12 offers a snapshot of the routing table, displaying the routes used by the emulator for data packet forwarding.

Routing Efficiency: Users can evaluate the effectiveness of the routing table, ensuring optimal paths are taken for data packet transmission. **Dynamic Adaptation:** The illustration showcases the emulator’s capacity to dynamically update routing information based on network changes.

```
quit
Quitting the bridge.
Shutting down.
agnhot@linprog5.cs.fsu.edu:~/project2
```

Figure 13: Quit

The figure 13 represents the option for users to exit the emulator, providing a seamless way to conclude emulation sessions.

User Convenience: The availability of a Quit option enhances user experience, allowing for a straightforward and intuitive interaction with the emulator. **Session Management:** Users can efficiently conclude emulation sessions, contributing to the overall usability and practicality of the emulator.

The figures and evaluations offer a comprehensive overview of the diverse functionalities embedded in the network emulator, showcasing its versatility in simulating various aspects of network communication and management.

5 CONCLUSION

The network emulator proves to be a robust tool for network simulation and testing. The illustrated functionalities, including Bridge Connection, Station Initialization, and Self Learning Table, showcase its ability to replicate real-world network scenarios effectively.

The visual representations of message exchanges, ARP information, router IP forwarding, pending queues, hosts, interfaces, and routing tables provide valuable insights for network analysis and troubleshooting.

The emulator’s practical features, such as displaying information and a user-friendly Quit option, contribute to its accessibility. As a versatile solution, it meets the demands of network professionals as well as new users.