iNetSim : *simulating internet communication*

Author: Mahidhar Reddy Narala, Graduate Student, Computer Science Department, Florida State University

***Abstract*— The internet constitutes of various complex systems and individual components interconnected. In this paper we simulate the building blocks of the Internet i.e. the basic components and the protocols that are used by devices and networks to send each other messages and achieve reliable communication. We intend to do this programmatically but to only a certain simple extent for demonstration purpose. In this paper, we will demonstrate a simple way to simulate the internet communication.**

***Keywords*— network bridge/switch, routers, ip, arp, ethernet, client server paradigms, tcp socket communication**

I. Introduction

The internet, according to Wikipedia, is a global system of interconnected computer networks that uses the Internet protocol suite (TCP/IP) to communicate between networks and devices. Put simply it is a network of networks. Each network contains multiple devices connected to a network bridge or network switch that facilitates communication within the network. Each network will be connected to at least one other network by a special kind of device/station i.e. a router. A router is responsible to connect multiple networks thereby achieving communication across different networks. In this manner, individual networks are interconnected and the devices within them communicate with each other using certain protocols like Ethernet, IP, ARP which we will also simulate in this paper. Now with this information in mind, let us think of what else do we need to achieve communication between devices to understand how these protocols came into place.

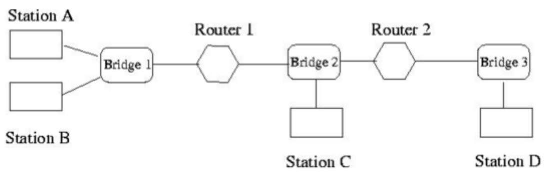


Fig. 1. A simple interconnected networks topology.

Each device will need a unique address so it can be identified in a network. We call this address MAC address. It is the physical address assigned to the network card of a device. No two devices will have the same mac address. This comes under the Ethernet protocol. Hence, within an individual network, if we know the mac address of the destination device that we want to communicate with, a device can simply send messages to bridge with the destination mac address and the bridge can forward those messages to the destination device as it is connected to it. This message must have a universal format so that all devices can understand and parse the messages. This format is called the Ethernet message format. Essentially, it contains the source mac address of the device the message originated from, the destination mac address of the device the message is intended for, some other fields for meta information, and the actual message/payload.

But how does a device know the mac address of other devices in its network? This is where ARP protocol comes in. Basically, whenever a device wants to know the mac address of any other device, it needs to send out an ARP request which will be broadcasted to all devices in the network. When the destination device gets the request, it needs to send the ARP response message back with its mac address.

But a mac address is only confined to one network. Devices from different networks will have no visibility of other network device's mac addresses. So, how do devices know how to send messages to destination devices in a different network? We'll need another way to identify the devices across networks. We call this is the IP address. Each device will have a unique IP address. So, if the IP address of the destination device can be known, any device can send messages; provided there is a system in place to forward this message from source to destination. Also, all devices need to follow a universal message format so all devices can parse the messages correctly. An IP packet essentially contains the source IP address, destination IP address, some more meta fields, and the actual message.

There needs to be some way that the IP address of the destination device can be retrieved. This is provided by a service called DNS. If the domain name of the destination is known, a DNS server will provide the IP address of it. There also needs to be a defined way for each network to pass the message and forward it to next network and so on until it reaches the destination network, the device is present in. This is the routers job to decide the next hop for each message at each network.

Hence, if a station needs to send messages, it will need to get the IP address of the destination device using DNS and form an IP packet. After forming this IP message packet, it needs to wrap this inside the ethernet packet with the destination mac of the next hop router/station within the network and send it to bridge so that the bridge can send it to that next hop device within the network. If the destination device isn't within the network, the message would have ended up with a router in the network. The router now needs to decide the next hop router/station in a different connected network and forward it. In this manner, the message gets passed around through various networks until the destination is reached. This is how messages are formed and sent across networks to reach its destination.

II. Overview

The basic network components so far discussed are bridge, station, and router. In this paper, all connections between components are implemented using actual tcp socket connections.

A. Bridge

A network bridge also called switch, acts as a server and it connects to multiple stations. This essentially acts as an individual local area network. A bridge's job is to support multiple connections to stations forming a local network and send/broadcast messages from one device to other devices based on mac address. We can simulate this by making this a server with a listening tcp socket where stations can connect to. This means the bridge needs to let stations know how to connect to it i.e., actual IP address and port where the server is listening to. This can be achieved by storing this information in a file whenever the server is started with the network name so that the stations can read it dynamically.

A bridge will always receive ethernet packets with some kind of payload inside it (IP/ARP). How does the bridge know whom to send the messages it receives? It is based on the destination mac address of the ethernet message. For this to happen, a bridge must know the mac addresses of all its connected stations. One way to achieve this is to let the bridge self-learn all the mac addresses whenever it receives messages and maintain a database. When a bridge receives its first message from its connected station, it knows which port the message came from, and it can read the source mac address implying the mac address of the station connected at that port. It can then store this port to mac mapping in its database. If the bridge doesn't know the destination mac address, it will just broadcast the message to all its stations. The stations can discard messages not intended for them. This way, the message will reach its destination and the bridge slowly learns the mac addresses of all the stations in its network.

This self-learning database must also have a timeout for each mapping so that the bridge will not try to send messages to stations that are inactive for a certain period. The bridge should also have a limit to the number of stations it can handle. It needs to handle all connections and functionalities concurrently for efficiency.

B. Station

A station is the end device which initiates message communication or receives a communicated message. It needs to connect to the bridge it wants by reading the corresponding network file with the IP address and port information that the bridge must have created when it started. The station must first be assigned a name, IP address, subnet mask and mac address. We can store this information in an interface file for each station along with the network(bridge) name it must be connected to. A station can connect to multiple bridges i.e. have multiple interfaces defined in the interface file.

A station needs to know the following to send a message – the IP address of the destination station, and the mac address of the next hop station. The IP address can be retrieved using a DNS service. This can be mimicked by maintaining a simple hostnames file with the names and corresponding IP addresses of the stations.

The station can know the mac address of the next hop station/router by asking for it in an ARP request. While the station waits for an ARP response back, it mustn't stay idle, but keep on processing for more user input. This means, that the station must maintain a pending queue for the messages it couldn't send yet because its waiting for an ARP response back. As soon as it gets the response, it must go back and send the message and remove it from the queue. All of this must happen concurrently as well for efficiency.

But how does the station know which station is the next hop station? We can achieve this by defining a routing table for each station/router. Given a destination IP address, the station will consult the routing table to figure out the next hop. The routing table must have the destination IP network prefix, the subnet mask for the destination network and the IP address of the next hop station and the interface(station) name through which this can be forwarded in case of station having multiple interfaces i.e., stations that are connected to multiple networks/bridges like routers. Note that this routing information is static and stored in a file for each station. This is a simple routing protocol used in this paper. There are many protocols and algorithms that can used to find the best possible route efficiently in all kinds of network topologies. We are not going into that detail in this paper.

After retrieving the necessary information, the station will create the IP packet with the actual message, wrap it in an ethernet packet and send it to the bridge. The bridge will forward it to the destination mac and the packet will be on its way. Any station when it receives a message, will process it only if the destination mac address is its own. And further, it will process the ethernet payload only if the destination IP address is its own. If not, the station will simply ignore the message.

C. Router

A router is nothing but a station with only one difference. The router will accept and process an ethernet packet only if the destination mac address is its own, but if the destination IP packet isn't its own, the router must again forward it on to the next hop station/router based on its routing table and the destination IP address.

III. DESIGN & DEVELOPMENT

We implemented all the components in python 3.9.6 programming language. We have implemented the program in a layered design i.e. separated the socket functionalities from the network component functionalities for ease of coding. From the overview, we understood that we have 4 basic components we had to implement. They are socket server, socket client, network bridge, network station. We understood that a network bridge is just an extension of a socket server and likewise, network station is an extension of a socket client. Note that a router is same as station with only one small difference. We have designed the components using classes and the extensions using the Inheritance concept of OOP (object-oriented programming) concepts. In addition, we have various defined structures for the messages and other data structures that we grouped together in a separate program *dstruct.py*. Furthermore, some more common functionalities and constant variables that may be used by any program are grouped together in a separate *util* (util.py) program.

A. *Socket Server*

A socket server as the name suggests is a program which opens a TCP socket as a server socket and listens for new connections and accommodates those clients. There are many ways a server can be implemented. In this paper, the server is programmed as a multiplexed and multithreaded server for efficiency. Ideally, a multi-processed server would be better, but for the purpose of demonstration, multi-threaded would work just fine. The program for the server is written in *server.py*. The server program has the implementation of the *Server* class.

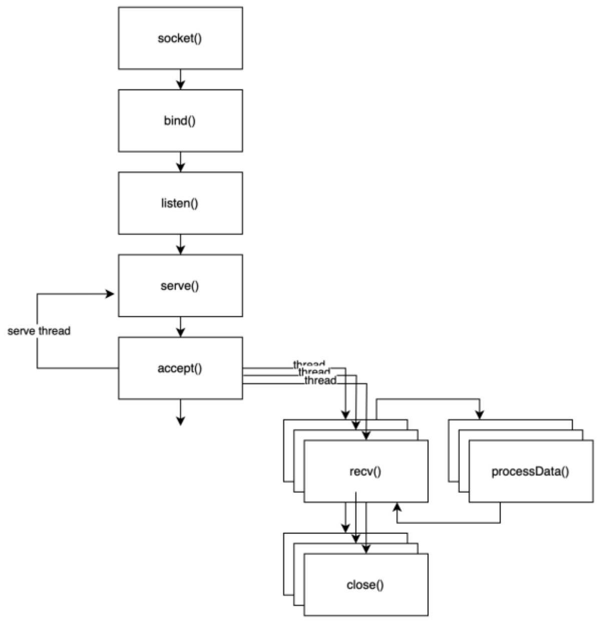


Fig. 2. A multithreaded server architecture.

The server program has the implementation of the *Server* class. The class Server contains the following field components:

- *servSock* – the tcp server socket object.
- *host* – the actual ip address of the server.
- *port* – the actual port of the server.
- *numPorts* – limit of number of client ports.
- *numClients* – number of clients connected.
- *clientSocks* – list of socket objects of the connected clients.
- *exitFlag* – a flag indicating if the threads should stop processing and quit.

The functionalities of the server are as follows:

1) *start*

Starts the server by opening an AF_INET, sock_stream type tcp socket in python. Binds the socket to the host and port set in the server object. Starts listening on the socket. Prints the ip and port information to console.

2) *serve*

This function is meant to be run in a separate thread which basically acts as a main server thread listening on the server socket. As listening on socket is a blocking function call, this functionality is multiplexed using *select* and whenever there is activity on the socket, it means there is a new connection on the socket. So, *acceptConnections* function is called when that happens. All this is kept in an infinite loop and the loop will quit when *exitFlag* is set thereby terminating the thread as well. As the select is a blocking itself, a timeout (SELECT_TIMEOUT defined in util.py) is set after which the loop is repeated if no activity happens on the socket.

3) *acceptConnections*

This function is called when there a new client attempts to connect to the server. If the server is within the numPorts limit, an *accept* message is sent to the client acknowledging the connection completion. If not, a *reject* message is sent. When the connection is accepted, the new socket object is stored with the port information in *clientSocks* list and the *numClients* field is incremented by 1. After this, a new thread is created and started for the newly accepted client socket whose sole purpose is to read any data sent to the server on that client socket and process that data. This functionality is written in *readConnection* function. So essentially, each connected client is handled by a separate thread.

4) *readConnection*

This function is meant to be run in a separate thread called by *acceptConnections* function. The function is again an infinite loop which exits if exitFlag is set thereby terminating the thread as well. The thread will keep listening on the client socket it was assigned i.e. *recv* function and if there is data on the client socket, it reads the data and calls the *processData* function. As recv is a blocking call, it is multiplexed using a *select*. As the select is a blocking itself, a timeout (SELECT_TIMEOUT defined in *util.py*) is set after which the loop is repeated if no activity happens on the socket. In case there is any error on while reading the data from socket or the client socket terminates the connection, the thread catches the error, and quits the loop. When the loop is quit, it closes the socket and removes the socket information from the stored clientSocks and decrements the *numClients* field by 1.

5) processData

This function is called by the client thread when there is data on the client socket. Here, it just prints the data to the console including the port on which the data was read.

6) sendData

This function is called when any data needs to be sent to a particular client. So, it takes the client socket and data as arguments and sends the data to the client socket.

7) broadcastData

This function is called when any data needs to be broadcasted to all clients. So, it takes the data as argument and sends the data to all clients. Optionally any client socket is also given as argument in which case it refrains from sending the data only on that client socket.

8) close

This function is called when we want to close and shutdown the server. It sets the exitFlag which makes sure that all the threads running will quit. It closes any connected client sockets and then closes the server socket.

B. Network Bridge

A network bridge is an extension of a socket server. So, using the *Bridge* class inherits the *Server* class and it is written in *bridge.py*. This means that a bridge object can call all the functionalities of the server and assumes all the server fields as the server object within itself. The bridge program is meant to be the main program to call so it can run as an independent network bridge component process.

The Bridge class contains the following fields.

- *exitFlag* – flag indicating when bridge is ready to quit.
- lanName – the network name assigned for this bridge to form.
- *numPorts* - limit of number of client ports.
- *addrFilename* – '.' + {lanName} + '.addr'
- *portFilename* - '.' + {lanName} + '.port'
- *slDb* – the self-learning database of the mac addresses of the stations and the ports they are connected at. This is basically a map data structure with key as mac address and value being a data structure for the client (*ClientDB* defined in dstruct.py)

The Bridge class has the following functionalities.

1) start

This function starts up the bridge network process. It calls the super start functionality of the server function that starts up the socket server. After the socket server is setup, the actual ip address and port information are stored in the files with filenames *addrFileName* and *portFileName*. This is done by *saveBridgeAddr* function. Then, a separate server thread is created and ran for the server's main *serve* function. In addition, a separate self-learning Db cleanup thread is created and ran to clean up the stale records. That is written in *cleanUpSlDb* function, and the thread is set as a daemon thread as we want this to run as long as the main threads are running. After the bridge's main thread basically waits for the server thread by calling *join*.

2) saveBridgeAddr

This function stores the ip address and port information of the server with the *lanName* of the bridge as part of the file names in the same directory as the *bridge.py*. These are the actual address and ports so that the stations can lookup these files when they want to connect to any bridge using the networks *lanName*.

3) processData

This function overrides the *processData* function of the *Server* class. This means when any of the server's client thread reads data on its client socket, this function is called. Here, the data passed is unpacked first in to an ethernet packet object. Then, the *slDb* is updated with the source mac address of the ethernet packet along with the timestamp. Then, based on the payload type, the payload is processed.

If the payload is ARP packet, the ethernet packet is broadcasted to all clients by calling the server's *broadcastData* function if the payload is ARP request. If the payload is ARP response or an IP packet, the destination mac address of the ethernet packet is checked in the *slDb* and sent to the corresponding client by calling the *sendData* function of the server. If *slDb* does not contain the destination mac address, the ethernet packet is simply broadcasted.

4) cleanUpSlDb

This function is meant to be run in a separate thread. It basically is an infinite loop with a refresh period of *SL_REFRESH_PERIOD* (defined in *util.py* usually set to 1 sec so that the thread runs every second) sleep for each loop. In each loop, it checks the timestamps of each record in the *slDb* and if more time has elapsed till current time from the stored timestamp than the SL_TIMEOUT (defined in *util.py*), it removes that record. This is to remove stale connection information if the station has been idle for more than SL_TIMEOUT period. The thread quits the loop when *exitFlag* is set thereby terminating itself.

5) *serveUser*

This function is meant to be run in a separate thread. It basically waits for user input on the console (sys.stdin) and calls *processUserCommand* function. The thread exits when *exitFlag* is set.

6) *processUserCommand*

This function is to handle two commands supported by the bridge.

- *quit* - shuts down the bridge by calling bridge's shutdown function.
- *show sl* - prints the bridge's self-learning database of the mac address of the connected stations along with the TTL of each record in a neat human readable format.

It also checks if the command passed matches the right ones and prints usage information to console if incorrect/unknown command is entered.

7) *shutdown*

This function makes sure that the bridge process exits in a controlled manner. It sets the exitFlag so that all the threads terminate gracefully. It then removes the files stored at the start of bridge about the ip address and port. After that, it calls the server's *close* function so that the server closes all client connections and the server socket.

The bridge program contains the following functions in addition to the *Bridge* class.

a) *main*

The is the entry point to the bridge program. It takes in and checks the right arguments are passed and then instantiates the bridge object. It then checks if a bridge with the given *lanName* already exists or not before starting up the bridge. It also registers a signal handler for keyboard interrupts (ctrl + c) and calls bridge's *shutdown* if interrupted. Finally, a separate user thread is created and ran which checks for user input on the console by calling *processUserCommand* function of the bridge. This thread is a daemon thread. After this, bridge is started by calling bridge's *start* function.

b) *bridgeLanExists*

This function checks if any bridge with the same *lanName* given as the argument already exists by checking for the ip and port files.

The bridge process can be started by executing the command as follows in the terminal:

```
$ python3 bridge.py <lanName> <numPorts>
```

C. *Socket Client*

A socket client is a program which opens a tcp socket and connects to a server specified by the bridge's ip address and port. The client is implemented in a rather simple way in the *client.py* program in *Client* class.

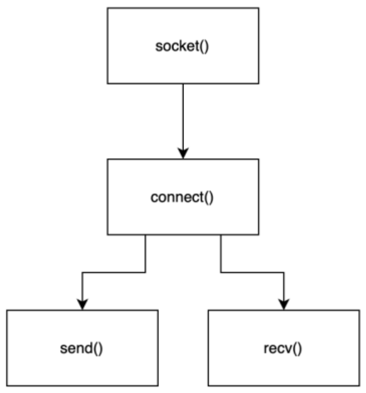


Fig. 3. A simple client architecture.

The following are the fields that the Client class contains.

- *server_addr* – set of host's ip address and port.
- *cliSock* – the client socket object.
- *exitFlag* – flag indicating if client must exit.

The following are the functionalities of Client class.

1) *connect*

This function opens a tcp AF_INET, sock_stream type socket and connects to the server at *server_addr*.

2) *run*

This function is meant to be run in a separate thread. It is an infinite loop where it just waits to read any data on the client socket (*recv*) that the server might have sent and calls *processData*. As *recv* is a blocking call, it is multiplexed using a *select*. As *select* is a blocking itself, a timeout (*SELECT_TIMEOUT* defined in *util.py*) is set after which the loop is repeated if no activity happens on the socket. In case there is any error while reading the data from socket or the server socket terminates the connection, the thread catches the error, and quits the loop. The loop is also quit when *exitFlag* is set. When the loop is quit, it closes the client socket before exiting.

3) *processData*

This function is called by the client thread when there is data on the client socket. Here, it just prints the data to the console including the port on which the data was read.

4) *send*

This function sends data to the server socket.

5) *close*

This function sets the *exitFlag* and closes the client socket.

D. Network Station

A Network Station is an extension of a Socket Client. So, the *Station* class inherits the *Client* class, and it is written in *station.py*. This means that a *Station* object can call all the functionalities of *Client* and assumes all the *Client* fields as client object within itself. The station program is meant to be the main program to call so it can run as an independent network station component process.

A station can connect to multiple networks. Note that a router is a station which connects to at least two networks. So, a station in essence is a collection of multiple interfaces. So, to accommodate this, we have defined one more class called *MultiStation* in the same program to encapsulate the list of stations within it.

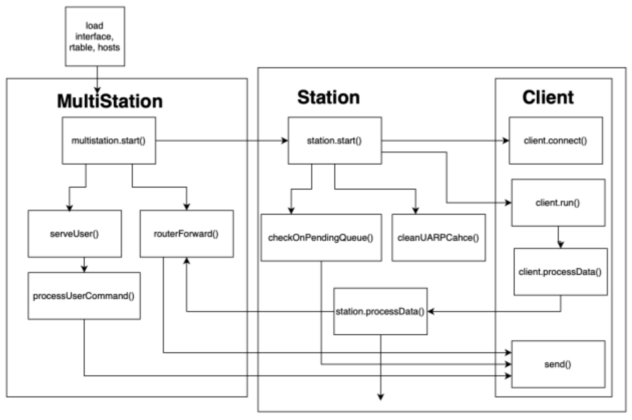


Fig. 4. Station architecture.

The MultiStation class contains the following fields.

- *exitFlag* – flag indicating if threads should quit.
- *hosts* – the list of hostnames and their ip addresses.
- *rTable* – the data structure to hold the routing table information for the station.
- *forwardQueue* – a queue to hold the packets received that are meant to be forwarded to next networks.
- *numStations* – number of stations (interfaces) in this multistation.
- *stationType* – the station type i.e., station/router.
- *stations* – the list of station class objects.
- *stationThreads* – list of station threads.

The MultiStation class has the following functionalities.

1) *loadInterface*

This function loads the interfaces of this multistation by reading the interface file given and initializes the station objects for each interface.

2) *loadHosts*

This function loads the hosts information that is meant to serve as DNS service by reading the hostnames file given.

3) *loadRoutingTable*

This function loads the routing table information assigned to this station by reading the routing table file given.

4) *start*

This is the start of the multistation. Here, each station is started and a separate thread is created and ran for each station. If the station is a router, a separate routing thread is also created and ran to forward packets concurrently. Then the main thread here waits for all threads to quit by calling *join* on each.

5) *routerforward*

This function is meant to be ran in a thread of its own. Here for each packet in the *forwardQueue*, it unpacks the ethernet packet and reads the IP payload in it. Based on the destination ip address, the next hop station is figured out along with the station interface with which the packet must be sent (*getNextRoute* in *util.py*). Then it is sent to Mac layer (*sendMac* in *util.py*) to be sent to the next station.

6) *serveUser*

This function is meant to be ran in a thread of its own. It basically waits for user input on the console (sys.stdin) and calls *processUserCommand* function. The thread exits when *exitFlag* is set.

7) *processUserCommand*

This function is to handle the following commands supported by the station.

a) *quit*

Quits the station program by calling multistation's shutdown function.

b) *send <destination> <message>*

Calls the send function of multistation to send the message to its destination.

c) *show iface*

Prints the interfaces of this station to console.

d) *show rtable*

Prints the routing table of this station to console.

e) *show hosts*

Prints the hostnames and ip addresses to console.

f) *show arp*

Prints arp cache of each station to console along with TTL to console in a neat readable format.

8) *send*

This function is called when the station needs to send a message. First the destination argument is read. If it's a hostname, it checks the *hosts* data structure to get its ip address. After this, the IP packet is formed with message, source & destination ip addresses. Then, the next hop station is figured out (*getNextRoute* in *util.py*) and sent to the mac layer (*sendMac* in *util.py*) to be sent to the next station.

9) *shutdown*

This function makes sure that the station process exits in a controlled manner. It sets the exitFlag so that all the threads terminate gracefully. Then each station's shutdown function is called so they quit processing sockets.

The Station class contains the following fields.

- *exitFlag* - flag indicating if threads should quit.
- *stationType* - the station type i.e., station/router.
- *hosts* - multistation's *hosts.*
- *rtable* - multistation's *rtable.*
- *interface* - the interface data structure which contains ip, mac, mask, and interface names of the stations.
- *forwardQueue* – multistation's *forwardQueue.*
- *bridgeHost* – actual ip address of bridge.
- *bridgePort* – actual port of bridge.
- *arpCache* – the data structure to store the arp cache for this station i.e. the ip and mac mappings.
- *pendQ* – this is the queue to store packets that are waiting to be sent.

The Station class has the following functionalities.

10) *getBridgeAddr*

This function reads the files of the bridge it wants to connect to using the network name (*lanName*) in the interface file to get the ip address and port information.

11) *start*

This function starts the station by calling the *connect* function *Client* class. Then, it creates a separate thread for the client and runs it by calling client's *run* function. Also, separate threads are created and ran for the arp clean up and the pending queue processing.

12) *validateBridgeAccept*

This function validates whether the bridge accepted the connection or rejected. If rejected, the station quits.

13) *processData*

This function is called when bridge sends any packet to this station. Here, the ethernet packet is unpacked and if the destination mac address is not same as this station's mac address or if it's not a broadcast mac address (ff.ff.ff.ff.ff.ff), the packet is dropped. If it's the same, the payload is processed. If the payload is an ARP packet and it's an ARP Request packet, then *sendARPResponse* function is called.

If the payload is IP packet, the destination ip address is read. If the station is not a router and the destination ip address is not the same as the stations ip address, the packet is dropped. If it's the same, the message within the packet is printed to console with the host name of the sender using the source ip address and hosts information.

If the station is a router and the destination ip address is not the same as the stations ip address, then next hop station is figured out (*getNextRoute* in *util.py*) and the packet kept in the *forwardQueue* to be forwarded to the next network.

If the received packet is an ARP Response, *arpCache* is updated with the along with the timestamp.

14) *checkOnPendingQueue*

This function is meant to be ran in a thread of its own. It is an infinite loop where it checks for packets in the *pendQ* and tries to send each packet. If the next hop station's mac address is known for the next hop ip address in the *arpCache,* the packet is sent by calling the client's *send* function. If the mac address is not known, an ARP request is sent for it. This loop runs continuously with a refresh period (*STATION_PQ_REFRESH_PERIOD* defined in *util.py*). This loop quits if *exitFlag* is set thereby terminating the thread as well.

15) *sendARPResponse*

This function is called when the station needs to send the ARP response back with the mac address of this station in response to an ARP Response packet it received.

16) *cleanUpARPCache*

This function is meant to be ran in a thread of its own. It is an infinite loop with a refresh period of *ARP_REFRESH_PERIOD* (defined in *util.py* usually set to 1 sec so that the thread runs every second) sleep for each loop. In each loop, it checks the timestamps of each record in the *arpCache* and if more time has elapsed till current time from the stored timestamp than the ARP_TIMEOUT (defined in *util.py*), it removes that record. This is to remove stale connection information if the station has been idle for more than ARP_TIMEOUT period. The thread quits the loop when *exitFlag* is set thereby terminating itself.

17) *shutdown*

The station program contains the following functions in addition to the *Station* and *MultiStation* class.

a) *main*

This is the entry point to the station program. Here, the arguments passed are validated and then the multi station object is initiated. It also registers a signal handler for keyboard interrupts (ctrl + c) and calls multistation's *shutdown* if interrupted. Finally, a separate user thread is created and ran which checks for user input on the console by calling *processUserCommand* function of the multistation. This thread is a daemon thread. After this, multistation is started by calling its *start* function.

The *util.py* contains the following functionalities in addition to the various constant variables that control the threading behavior.

18) sendARPReq

This function is called when the mac address of the next hop station is not known. It creates and ARP Request packet with the ip address of the station whose mac address is needed along with the mac address of the station that is sending the request. This is wrapped in an ethernet packet and sent to the bridge by calling Client's *send* function.

19) getNextRoute

This function is called when the station or router needs to send a packet to the next hop station. This is done by referring to the routing table. The destination ip address of the packet is matched with the network prefix of each of the records in the routing table and the closest match is selected. That record will contain the ip address of the next hop station along with the interface name of the station from where the packet needs to be sent.

20) sendMac

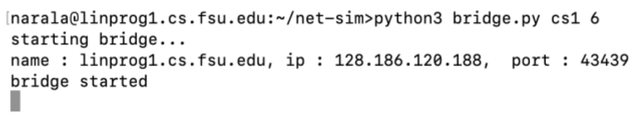
Here, the ethernet packet is formed and it wraps the IP packet sent with the mac address of the next hop station and the source mac address as the current station. It then puts the ethernet packet in the station's *pendQ* to be sent.

The station process can be started by executing the command as follows in the terminal:

```
$ python3 station.py -no/-route <iface_filepath> <rtable_filepath> <hosts_filepath>
```

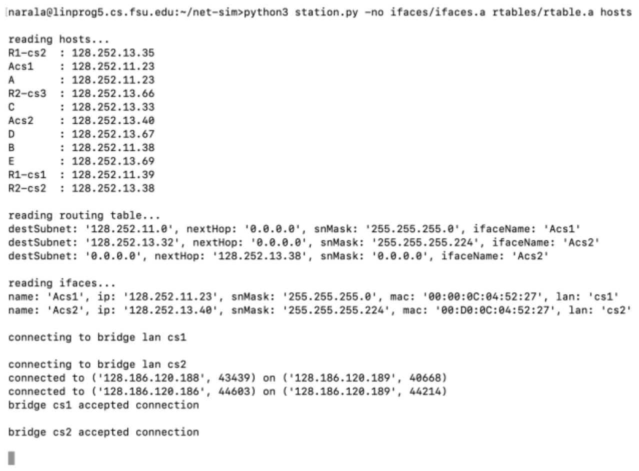
IV. ILLUSTRATIONS

The following are illustrations of the bridge and station programs sending messages and showing their information.



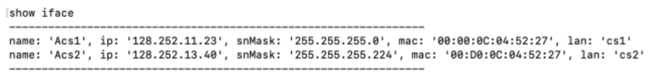
```
narala@linprog1.cs.fsu.edu:~/net-sim>python3 bridge.py cs1 6
starting bridge...
name : linprog1.cs.fsu.edu, ip : 128.186.120.188,  port : 43439
bridge started
```

Fig. 5. Bridge startup.



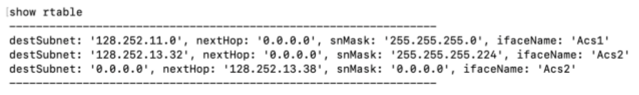
```
narala@linprog5.cs.fsu.edu:~/net-sim>python3 station.py -no ifaces/ifaces.a rtables/rtable.a hosts

reading hosts...
R1-cs2  : 128.252.13.35
Acs1    : 128.252.11.23
A       : 128.252.11.23
R2-cs3  : 128.252.13.66
C       : 128.252.13.33
Acs2    : 128.252.13.40
D       : 128.252.13.67
B       : 128.252.11.38
E       : 128.252.13.69
R1-cs1  : 128.252.11.39
R2-cs2  : 128.252.13.38

reading routing table...
destSubnet: '128.252.11.0', nextHop: '0.0.0.0', snMask: '255.255.255.0', ifaceName: 'Acs1'
destSubnet: '128.252.13.32', nextHop: '0.0.0.0', snMask: '255.255.255.224', ifaceName: 'Acs2'
destSubnet: '0.0.0.0', nextHop: '128.252.13.38', snMask: '0.0.0.0', ifaceName: 'Acs2'

reading ifaces...
name: 'Acs1', ip: '128.252.11.23', snMask: '255.255.255.0', mac: '00:00:0C:04:52:27', lan: 'cs1'
name: 'Acs2', ip: '128.252.13.40', snMask: '255.255.255.224', mac: '00:D0:0C:04:52:27', lan: 'cs2'

connecting to bridge lan cs1

connecting to bridge lan cs2
connected to ('128.186.120.188', 43439) on ('128.186.120.189', 40668)
connected to ('128.186.120.186', 44603) on ('128.186.120.189', 44214)
bridge cs1 accepted connection

bridge cs2 accepted connection
```

Fig. 6. Station startup.



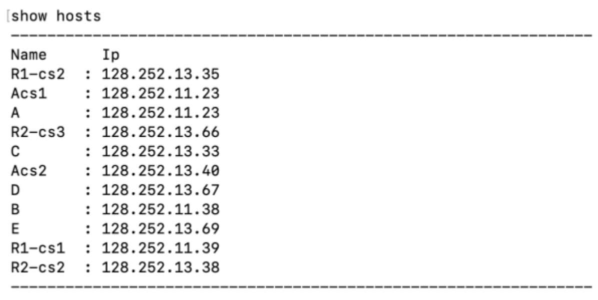
```
show iface
------------------------------------------------------------------
name: 'Acs1', ip: '128.252.11.23', snMask: '255.255.255.0', mac: '00:00:0C:04:52:27', lan: 'cs1'
name: 'Acs2', ip: '128.252.13.40', snMask: '255.255.255.224', mac: '00:D0:0C:04:52:27', lan: 'cs2'
------------------------------------------------------------------
```

Fig. 7. Station showing interface.



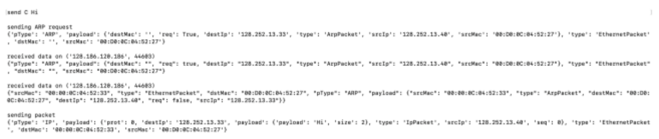
```
show rtable
------------------------------------------------------------------
destSubnet: '128.252.11.0', nextHop: '0.0.0.0', snMask: '255.255.255.0', ifaceName: 'Acs1'
destSubnet: '128.252.13.32', nextHop: '0.0.0.0', snMask: '255.255.255.224', ifaceName: 'Acs2'
destSubnet: '0.0.0.0', nextHop: '128.252.13.38', snMask: '0.0.0.0', ifaceName: 'Acs2'
------------------------------------------------------------------
```

Fig. 8. Station showing routing table.



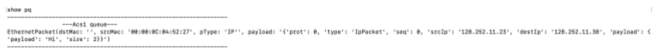
```
show hosts
------------------------------------------------------------------
Name      Ip
R1-cs2  : 128.252.13.35
Acs1    : 128.252.11.23
A       : 128.252.11.23
R2-cs3  : 128.252.13.66
C       : 128.252.13.33
Acs2    : 128.252.13.40
D       : 128.252.13.67
B       : 128.252.11.38
E       : 128.252.13.69
R1-cs1  : 128.252.11.39
R2-cs2  : 128.252.13.38
------------------------------------------------------------------
```

Fig. 9. Station showing hosts.



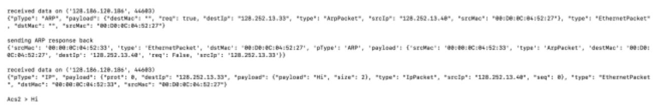
```
send C Hi

sending ARP request
{'pType': 'ARP', 'payload': {'destMac': '', 'req': True, 'destIp': '128.252.13.33', 'type': 'ArpPacket', 'srcIp': '128.252.13.40', 'srcMac': '00:D0:0C:04:52:27'}, 'type': 'EthernetPacket'
, 'dstMac': '', 'srcMac': '00:D0:0C:04:52:27'}

received data on ('128.186.120.186', 44603)
{"pType": "ARP", "payload": {"destMac": "", "req": true, "destIp": "128.252.13.33", "type": "ArpPacket", "srcIp": "128.252.13.40", "srcMac": "00:D0:0C:04:52:27"}, "type": "EthernetPacket"
, "dstMac": "", "srcMac": "00:D0:0C:04:52:27"}

received data on ('128.186.120.186', 44603)
{"srcMac": "00:00:0C:04:52:33", "type": "EthernetPacket", "dstMac": "00:D0:0C:04:52:27", "pType": "ARP", "payload": {"srcMac": "00:00:0C:04:52:33", "type": "ArpPacket", "destMac": "00:D0:
0C:04:52:27", "destIp": "128.252.13.40", "req": false, "srcIp": "128.252.13.33"}}

sending packet
{'pType': 'IP', 'payload': {'prot': 0, 'destIp': '128.252.13.33', 'payload': {'payload': 'Hi', 'size': 2}, 'type': 'IpPacket', 'srcIp': '128.252.13.40', 'seq': 0}, 'type': 'EthernetPacket
', 'dstMac': '00:00:0C:04:52:33', 'srcMac': '00:D0:0C:04:52:27'}
```

Fig. 10. Station sending a message.



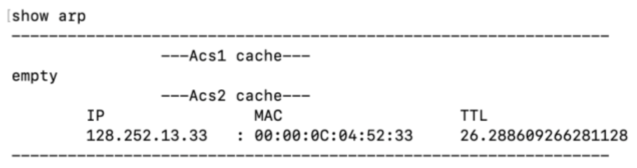
```
show pq
------------------------------------------------------------------
                ---Acs1 queue---
EthernetPacket(dstMac: '', srcMac: '00:00:0C:04:52:27', pType: 'IP'', payload: '{'prot': 0, 'type': 'IpPacket', 'seq': 0, 'srcIp': '128.252.11.23', 'destIp': '128.252.11.38', 'payload': {
'payload': 'Hi', 'size': 2}}')
------------------------------------------------------------------
```

Fig. 11. Station showing packets in its pending queue.



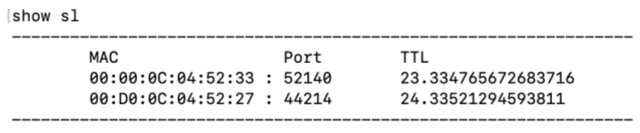
```
received data on ('128.186.120.186', 44603)
{"pType": "ARP", "payload": {"destMac": "", "req": true, "destIp": "128.252.13.33", "type": "ArpPacket", "srcIp": "128.252.13.40", "srcMac": "00:D0:0C:04:52:27"}, "type": "EthernetPacket"
, "dstMac": "", "srcMac": "00:D0:0C:04:52:27"}

sending ARP response back
{'srcMac': '00:00:0C:04:52:33', 'type': 'EthernetPacket', 'dstMac': '00:D0:0C:04:52:27', 'pType': 'ARP', 'payload': {'srcMac': '00:00:0C:04:52:33', 'type': 'ArpPacket', 'destMac': '00:D0:
0C:04:52:27', 'destIp': '128.252.13.40', 'req': False, 'srcIp': '128.252.13.33'}}

received data on ('128.186.120.186', 44603)
{"pType": "IP", "payload": {"prot": 0, "destIp": "128.252.13.33", "payload": {"payload": "Hi", "size": 2}, "type": "IpPacket", "srcIp": "128.252.13.40", "seq": 0}, "type": "EthernetPacket
", "dstMac": "00:00:0C:04:52:33", "srcMac": "00:D0:0C:04:52:27"}

Acs2 > Hi
```

Fig. 12. Station receiving packets.



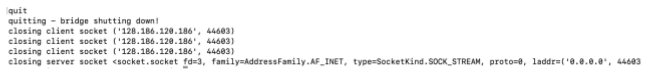
```
show arp
-----------------------------------------------------------------
                ---Acs1 cache---
empty
                ---Acs2 cache---
        IP               MAC                    TTL
        128.252.13.33   : 00:00:0C:04:52:33     26.288609266281128
-----------------------------------------------------------------
```

Fig. 13. Station showing its ARP Cache with TTL.



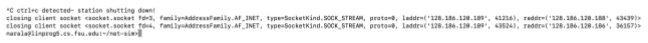
```
show sl
------------------------------------------------------------------
        MAC                    Port        TTL
        00:00:0C:04:52:33 : 52140       23.334765672683716
        00:D0:0C:04:52:27 : 44214       24.33521294593811
------------------------------------------------------------------
```

Fig. 14. Bridge showing its self-learning database with TTL.



```
quit
quitting - bridge shutting down!
closing client socket ('128.186.120.186', 44603)
closing client socket ('128.186.120.186', 44603)
closing client socket ('128.186.120.186', 44603)
closing server socket <socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 44603
```

Fig. 15. A bridge quitting with quit command.



```
^C ctrl+c detected- station shutting down!
closing client socket <socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('128.186.120.189', 41216), raddr=('128.186.120.188', 43439)>
closing client socket <socket.socket fd=4, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('128.186.120.189', 43524), raddr=('128.186.120.186', 36157)>
narala@linprog5.cs.fsu.edu:~/net-sim>
```

Fig. 16. Station quitting on Keyboard Interrupt (Ctrl + C).

V. CONCLUSION

In this paper, we have successfully simulated the building blocks of the Internet i.e., network bridges/switches and Stations/Devices/Routers. We have also successfully implemented the IP/ARP/Ethernet protocols through which the communication is made possible. We have learnt various self-learning and clearing functionalities of the components. Granted, we didn't go too deep into the routing algorithms etc., but this simulator can serve as a base to anyone that wants to test and evaluate any novel protocol or component in conjunction with the existing internet communication.