Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Phan Binh Nguyen

CSU33031 Computer Networks Assignment #2: Flow Forwarding

Contents

1 Introduction

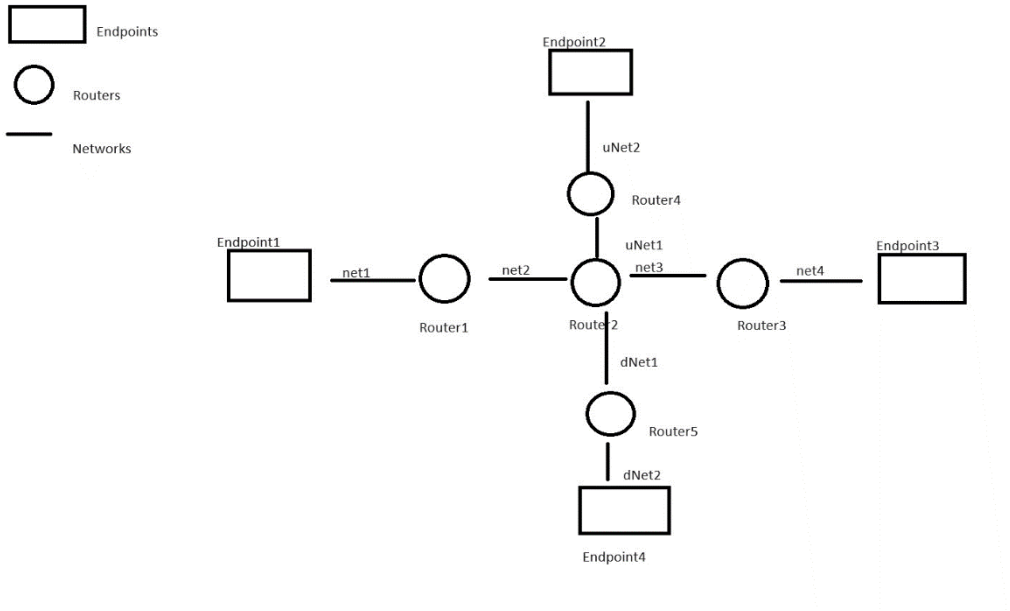
In this report, I will review the preparation, implementation and thought process for this assignment.

I used Python 3, socket, zlib, psutil, and time built-in libraries. In my implementation, there are multiple nodes which are connected by networks. A node is either a router or an endpoint, but it doesn't know whether other nodes around it does exist. So, the main problem is searching and finding routes within networks and creating a route to send data directly.

2 Theory of Topic

In this section, I will talk about my implementation's topology and flow control and the overall thought process behind this project.

2.1 Topology

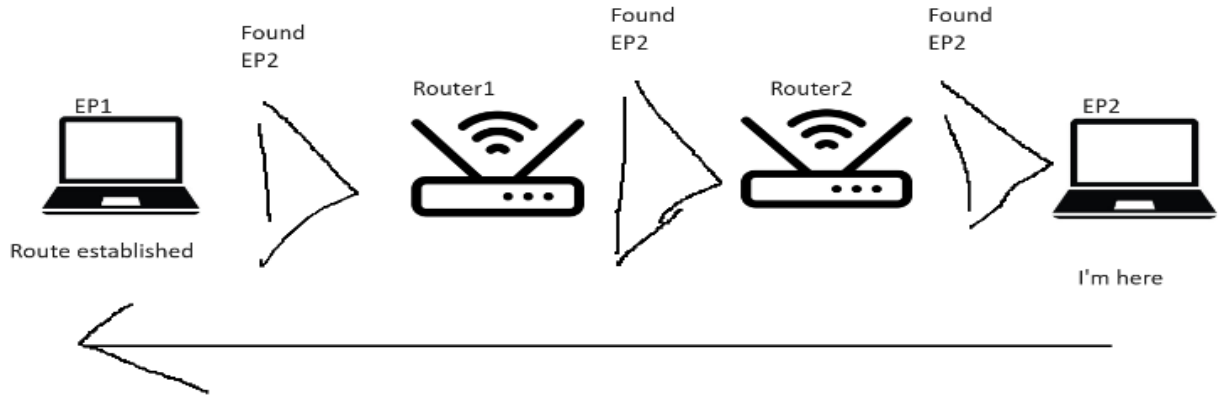


I am using four endpoints (rectangles), five routers (circles) and eight networks (lines). It can be expanded to use more endpoints, routers, or networks. Endpoints do not know whether other endpoints exist, nor do they know about the connected routers. So, the only way to send data between endpoints is to search for a route through the networks.

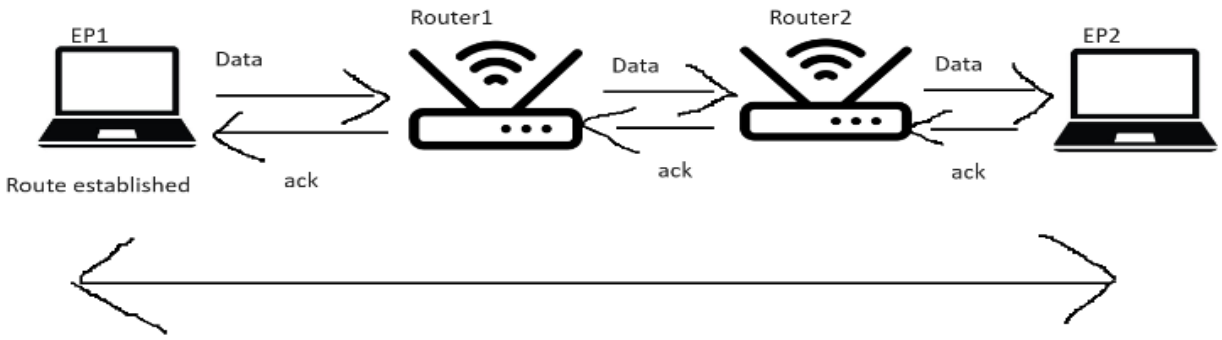
2.2 Creating Routes/ Flow control

In order to create a route, and endpoint should broadcast a search message, before sending the actual message.

In this shortened example, there are two endpoints and two routers. EP1 would want to send a message to EP2, but it has yet to learn if EP2 even exist. Since EP2 is not on the same network as EP1, EP1 will broadcast the search message, which router1 should be able to get. Router1 would also post the same message, which should reach Router2, and at last, Router2 broadcast the message, and this time, the search message reached its destination, namely endpoint2.

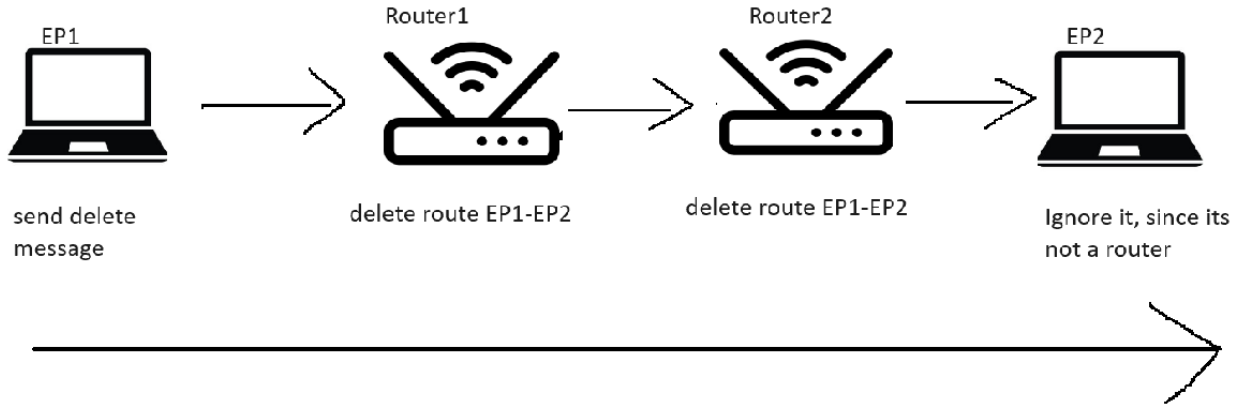


After finding EP2, EP2 will try to find the route back to the sender, which would be EP1. It finds EP1 the same way as EP1 found EP2. At this point, all the routers have saved the route, so they do not have to broadcast the message; they can send it directly.



If the route has been established, EP1 can send data directly to EP2 through the routers. EP2 verifies the data by comparing the checksum in the header to the calculated one. If it matches, it will return an acknowledgement that the data has safely arrived. Otherwise, it could request resending the data once more. If the data fails to arrive to EP2 twice, EP1 will stop resending the data and will move on, avoiding a potential infinite loop.

As for the last step, after sending the last data packet and receiving the acknowledgement, EP1 would send a last direct message towards EP2, which should signal all the routers in the direction of EP2 that they should delete this route.

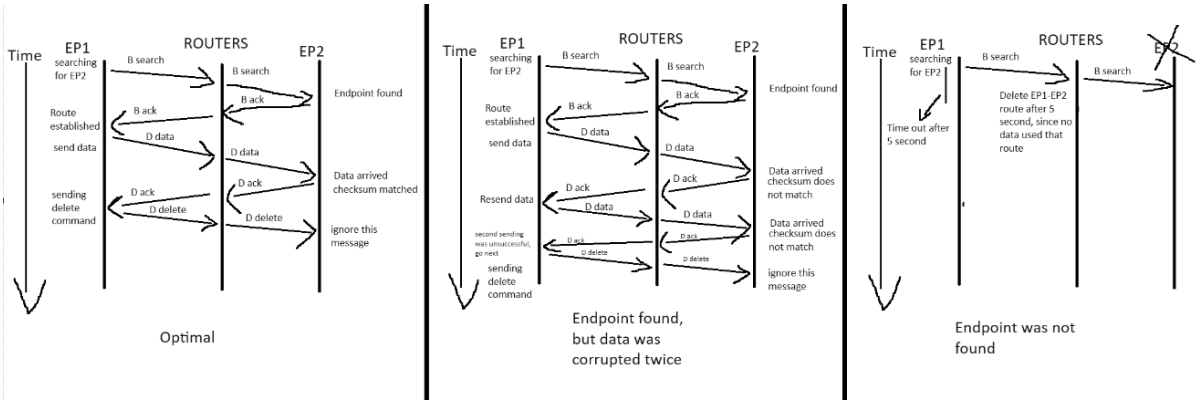


If everything worked as intended, then there should be no route saved in the routers between EP1 and EP2.

If the searched endpoint does not exist, EP1 will time out and move on. The route in the routers would be deleted after 5 seconds if no data went through it. (For instance, if EP1 and EP2 already established a path, and EP1 was sending data to EP2, and EP2 wanted to send data to an endpoint which doesn't exist, EP3, then after 5 seconds, the router would only delete the route between EP2 and EP3, since no data packet went through that channel.)

This is a simplified version of the model I created, but everything should work the same way I just described.

2.3 ARQ



In these diagrams, routers are shown with one line since it does not matter whether the data has to go through 1 router till it reaches the wanted destination or it has to go through 3 routers.

The first diagram shows the optimal situation, where the searched endpoint exists, and data was not corrupted during the transfer.

The second diagram illustrates a situation where the data was corrupted during transfer. Hence, the sender endpoint had to resend it, but the data got corrupted for the second time, so the sender endpoint just moved on, avoiding a potential infinite loop.

The third part of the picture shows what happens if the endpoint does not exist or the acknowledgement does not arrive in time. In both cases, there is a 5-second window. If the acknowledgement does not arrive after sending data, the sender endpoint will try to resend it. In the case of searching for the receiving endpoint, if the sender does not receive any signal from the receiving endpoint, it would assume that it does not exist and will not try to send any packet.

In the picture, B stands for broadcasting, and D stands for direct.

So, “B search” means that it's a search message and it was broadcasted, while “D data” implies that it was a direct data sending.

3 Implementation

In this section, I will focus on discussing the implementation of the protocol. Each router and endpoint have a unique ID, which cannot be modified. I used the ID to check if the sender was the same as the destination, to sort out messages which would have been broadcasted to itself.

3.1 Checksum

I am calculating the checksum using the crc32 function from the Zlib library. The crc32 function calculates the CRC (Cyclic Redundancy Check) checksum of the given data and returns a 32-bit integer.

```
def checksum_calculator(data):
    checksum = zlib.crc32(data)
    checksum = checksum%255
    return checksum
```

I took the mod 255 of that number so the checksum could be fit under one byte in every circumstance. Every time a data packet is sent, the receiver separates the header from the data and calculates the checksum from the extracted data. If the checksum in the header and the checksum of the extracted data match, the receiver will know that the data arrived successfully.

3.2 Routing table

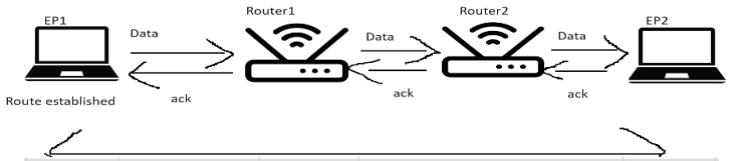
The routing table is just an empty list initially. The router can store or delete objects from the table.

3.2.1 Add a route

Whenever data goes through the router, which is not a delete-type message, the routing table stores the sender and destination endpoints' IDs, the sender's address (the router does not know other's address) and it gives a time stamp. The routing table only stores unique instances of address pairs.

```
def addToRoutingTable(originalSender,dest,addr):
    isThere=False
    for i in range(len(RoutingTable)):
        if(RoutingTable[i][0]==str(originalSender)and RoutingTable[i][1]==(str(dest))):
            isThere=True
            break
    if(not isThere):
        RoutingTable.append([str(originalSender),str(dest),addr,timer])
```

In this example, the route has already been established so that the data can flow directly between the two endpoints.



The routing table at this point for Router 1 should look like the picture to the right. The time is arbitrary since every time a packet uses a specific route, it's time limit would be refreshed.

	Destination EP	Sender EP	Next address to send	Time
1	EP2	EP1	(IP, Port)| adress of router 2	5
2	EP1	EP2	(IP, Port)| adress of EP1	5

3.2.2 Delete a route

A router deletes a route in two cases, either by command from one of the endpoints or automatically through a timer. Whenever a message goes through a router, the router checks the type of data by checking the 7th byte of the header. If that byte is equal to 3, the router would first store the following hop address in a temp, then delete the routes both ways and send the same message to the next hop. The router deletes both ways since it's more efficient than going from EP1 to EP2 and returning to EP1 with the delete message. (In my case, the message would just go from EP1 to EP2, and EP2 would ignore it, but the effect would be the same.)

The router also deletes any route not used for more than 5 seconds by comparing the timestamp in the routing table to the timer variable. Since it runs on a different thread, the routing table is updated almost in real-time.

3.3 Header

I represent the ID-s with hexadecimal characters. The header is usually 7 bytes long and 8 bytes long if the data packet has an actual message since the 8th byte holds the checksum.

Name:	Destination EP's ID	Sender EP's ID	Current Sender ID	message type	checksum
length in bytes:	2	2	2	1	1

The structure of the header. The destination and sender IDs are just for identification, the current sender ID is used to filter out messages that arrived to itself through broadcast.

On the right there is a table for each number which, could be a message type.

Only a message with message type 2 would have a checksum attached since the other carries all the essential information in the header.

0	searching for destination endpoint
1	ack, destination endpoint found
2	data/message
3	delete route
4	ack, message arrived, checksum matched
5	ack, message arrived, checksum not matched, resend

message type meaning

3.3.1 Examples

These are examples from the pcap file.
Every endpoint's ID starts with aa and every router's ID starts with bb. (EP= endpoint)

```
aa 03 aa 01 bb 01 00 73  65 61 72 63 68 69 6e 67    ······s earching
```

AA03= Destination is EP AA03 | AA01: sender EP|BB01: just sent by router BB01| 00: searching for endpoint

```
aa 01 aa 03 bb 04 01 61  63 6b                      ·····a ck
```

AA01= Destination is EP AA01 | AA01: sender EP |BB04: sent by router BB04| 01: ack, and found the endpoint

```
aa 03 aa 01 bb 01 02 68  69                         ·····h i
```

AA03= Destination EP | AA01: sender EP |BB01: sent by router BB01| 02: actual message| 68: checksum

```
aa 01 aa 03 bb 04 04 61  63 6b                      ·····a ck
```

AA01= Destination EP | AA03: sender EP |BB04: sent by router BB04| 04: ack, that message arrived to AA03, and checksum matched, no need to resend.

```
aa 03 aa 01 bb 01 03 64  65 6c 65 74 65             ·····d elete
```

AA03= Destination EP| AA01 sender EP| BB01: sent by router BB01| 03: delete→ router should delete the route AA03-AA01 or AA01-AA03

This concludes how data transfer works, including searching for and deleting routes between endpoints AA01 and AA03.

3.4 Endpoints

The only difference between endpoints is the ID since every endpoint has a unique ID.

```
address="AA 04"
```

Each endpoint has three threads.

3.4.1 Timer thread

The timer thread is to increase the timer variable, and it is used to check for timeouts.

```
def increase_counter():
    global timer
    while True:
        timer += 1
        time.sleep(1)
```

3.4.2 Sender thread

The sender thread waits for user input, which should be in the format of:
[Endpoint ID] message,
Example: AA03 hello there
I split the user input into two parts, where the first part would be the endpoint ID, while the rest would be the actual message which would be sent.

```
msgFromUser = input("Type your message: ")
msgFromUserArr = msgFromUser.split(' ',1)
```

The first step would be finding the route to the other endpoint.

```
for interface_name, addrs in interfaces.items():
    for addr in addrs:
        if addr.family == socket.AF_INET:  # We only want to broadcast on IPv4 interfaces
            ip = addr.address.rsplit('.', 1)[0] + '.255'  # Calculate the broadcast IP for
            sock.sendto(Dest+Sender+PrevSender+hexType+message.encode(), (ip, port))
```

I'm using the psutil library to check all the connected networks and then iterate through it. Then, the endpoint calculates the broadcasting IP by replacing the last part of the IP with 255 (essentially masking it).
After sending the search message, the endpoint will wait until there is a time out or an ack arrives.
Timer and block are global variables. If an ack arrives from the correct endpoint, the listener thread will modify the block to False; otherwise, after 5 seconds, the endpoint will stop waiting, and it will raise the timeout (quitByTimer) flag.
If the route does not exist, then the endpoint would just go back to the first step and would wait for a user input.

```
initialTimer=timer
quitByTimer=False
while(block):
    if(timer-initialTimer>=5):
        block=False
        quitByTimer=True
    continue
```

If the route does exist, then the address of the next hop has been stored by the listener thread, and the endpoint can send data directly to its destination.

```
hexType=bytes.fromhex("02")
checkSum=checksumCalculatorB(msgFromUser.encode())
sock.sendto(Dest+Sender+PrevSender+hexType+checkSum+msgFromUser.encode(), currentDest)
```

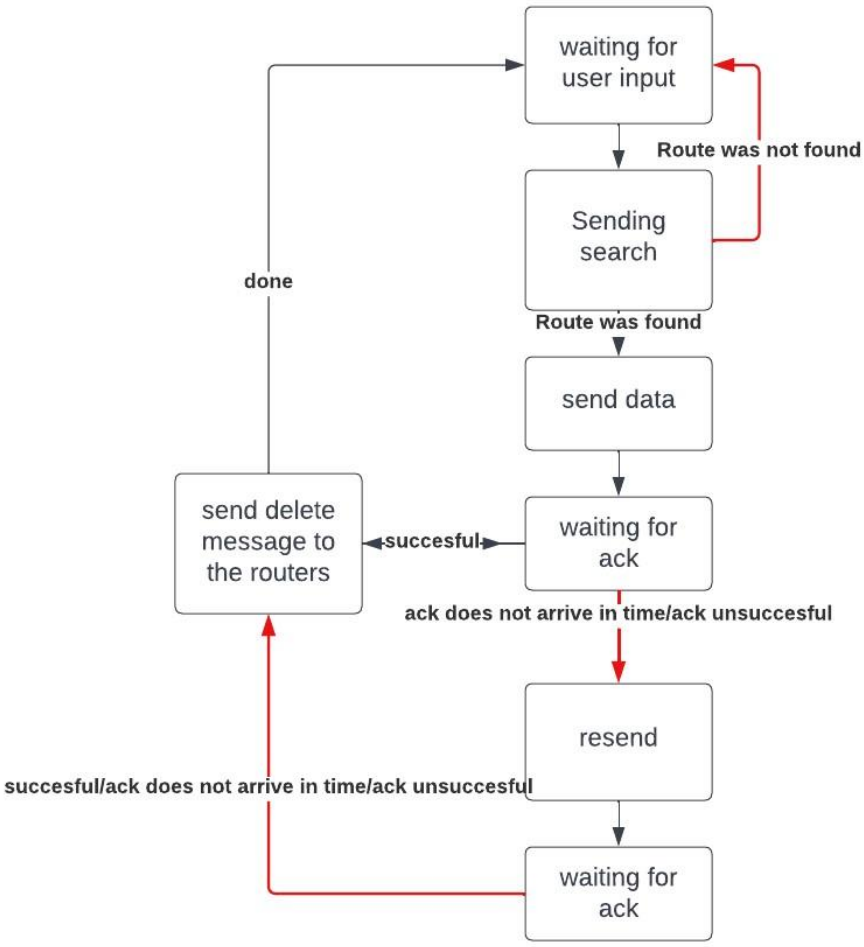
currentDest is a global variable which holds the address of the router/endpoint where the message needs to be sent to get to the destination. (It's modified by the listener thread when an ack arrives and reset by the sender thread whenever it sends a search message.)
After sending the message, it waits for acknowledgement or timeout, similarly, when it was searching for a route. If the ACK says that the data was corrupted or a timeout happens, the sender thread will try to resend the data.
Either way, the last step is always a route deleting message.

The message itself is unimportant since the routers only check for the 7th byte of the header.

```
#send deleting line
hexType=bytes.fromhex("03")
message="delete"
sock.sendto(Dest+Sender+PrevSender+hexType+message.encode(), currentDest)
```

Sender thread summary

This flowchart shows what the endpoint does when it comes to sending a message.



sender thread in a nutshell

3.4.3 Listener thread

The listener thread gets every message coming to the endpoint.

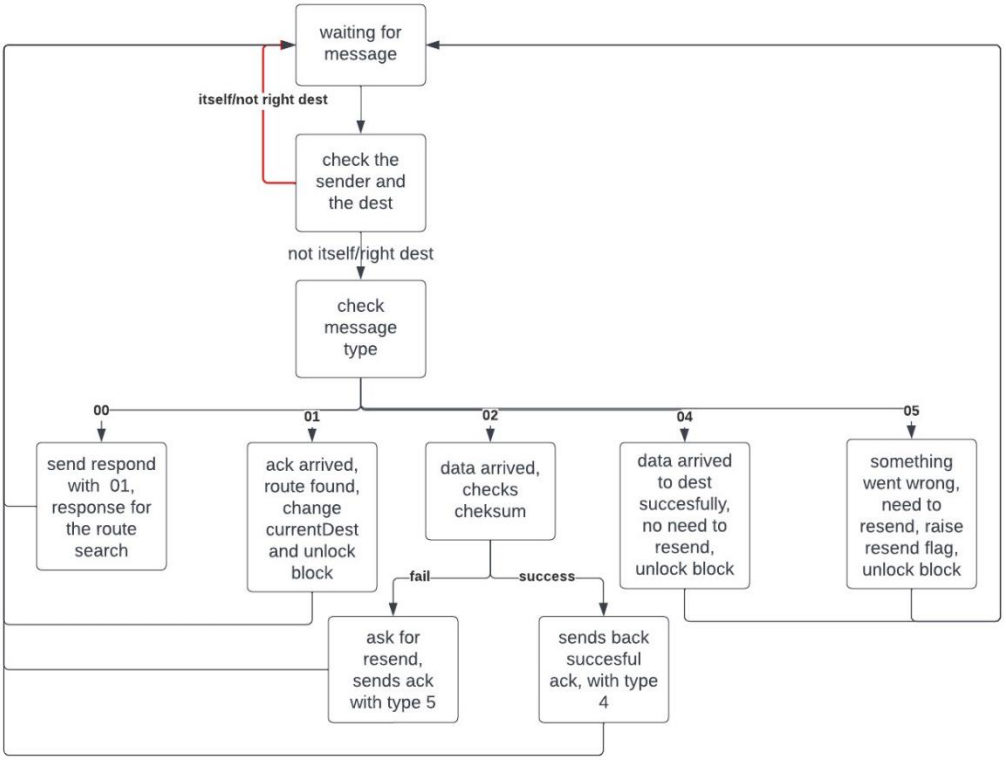
After receiving a message, it checks whether the sender was itself and if it was intended for this endpoint, if yes then the thread just ignores the data.

```
sock.bind(("0.0.0.0", port))
while True:
    data, addr = sock.recvfrom(65535)
    sender=data[2:4]
    dest=data[0:2]
    if (sender!=bytes.fromhex(address) and dest==bytes.fromhex(address)):
```

If the data was intended for this endpoint, and the sender was not itself, it checks the message type and responds according to it.
If it's 0, it returns an acknowledgement that the endpoint is here by changing the msg-type to 1.
If the msg-type is 1, the endpoint knows the route has been found, so it sends the actual data.
If the msg-type is 2it, the actual message has arrived, so the endpoint should check the checksum and respond accordingly.
If the msg type is 4, then ack arrived, and everything went well, but if the msg-type is 5, then something went wrong;
EP should try to resend the message.

```
if msgType== 0:
    sock.sendto(Dest+originalSender+PrevSender+bytes.fromhex("01")+messageA.encode(), addr)
if msgType== 1:
    if currentDest==str(Dest):
        currentDest=addr
        print("Route found")
        block=False
if msgType== 2:
    messageFromEP=data[8:]
    checkSumFromPacket=data[7]
    checksum=checksum_calculator(messageFromEP)
    if(checkSumFromPacket==checksum):
        print(data)
        sock.sendto(Dest+originalSender+PrevSender+bytes.fromhex("04")+messageA.encode(), addr)
    else:
        sock.sendto(Dest+originalSender+PrevSender+bytes.fromhex("05")+messageA.encode(), addr)
if msgType== 4:
    print("ack Arrived")
    block=False
    retry=False
if msgType== 5:
    print("ack Arrived")
    block=False
    retry=True
```

The listener thread listens to any incoming message and then raises flags according to the message. The central part of the endpoints is happening in the sender thread.



Simplified representation of the listener thread

3.5 Routers

The difference between the routers is their ID since every router has a unique ID.
Every router has three threads, albeit one thread is just for user input and used for debugging.

This thread waits for user input and then prints out the routing table. It can be used to check if the routing table works as intended. (For example, if a route has been deleted after 5 seconds of not being used.)

```
def askRoutingTable():
    global RoutingTable
    while(True):
        msgFromUser = input("Type your message: ")
        if msgFromUser=="table":
            print("table")
            print(RoutingTable)
```

3.5.1 Timer thread

The timer thread increases the global variable timer every second. It also checks whether there is a route in the routing table which has not been used for more than 5 seconds.

An abandoned route usually happens when one of the endpoints is trying to find a non-existent endpoint.

```
def increase_counter():
    global RoutingTable
    global timer
    while True:
        timer += 1
        tempTable=[]
        for i in range(len(RoutingTable)):
            if(timer-RoutingTable[i][3]>5):
                continue
            else:
                tempTable.append(RoutingTable[i])
        RoutingTable=tempTable
        time.sleep(1)
```

3.5.2 Listener thread

This is the main thread of the router.
The router is just forwarding messages according to the message type and the routing table.
If the message type is 3, it deletes the route mentioned in the header from the routing table. (Mentioned in section 3.2.2)
The router adds a route to the table for every other message type. (If that route has not been added, if the path already exists, nothing would change in the table.)
For example, if a message arrived with the header AA02|AA03|BB02|00
The router would store the sender's address (it does not matter if it's a router or an endpoint), and the original endpoint sender's ID (AA03) so that next time the router knows where to route any message which would want to get to that specific endpoint (AA03 in this case).

If the route does exist to the destination endpoint, then the routing table would refresh the timer of the route and send the message directly to the next address. Keep in mind that the router does not know whether the address stored in the table is the address of a router or the endpoint's address; it just knows that to get to the destination, it needs to send the message that way.

```
for routes in RoutingTable:
    if(routes[0]==str(dest)):
        if(msgType==1):
            hexMsgType=bytes.fromhex("01")
        if(msgType==0):
            hexMsgType=bytes.fromhex("00")
        if(msgType==2):
            hexMsgType=bytes.fromhex("02")
        if(msgType==4):
            hexMsgType=bytes.fromhex("04")
        if(msgType==5):
            hexMsgType=bytes.fromhex("05")
        print("direct")
        routes[3]=timer
        sock.sendto(dest+originalSender+bytes.fromhex(address)+hexMsgType+message,routes[2])
        isSent=True
```

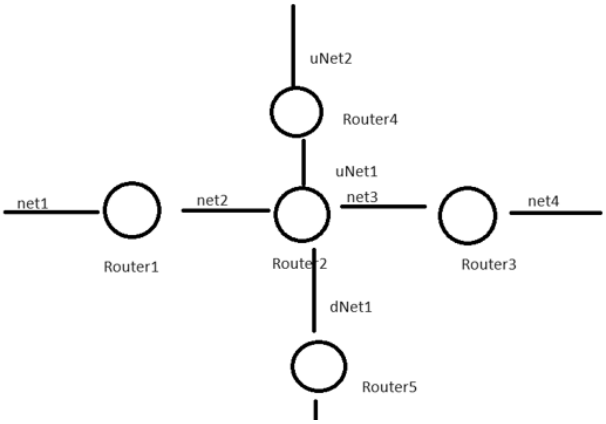
The if statements, is just there so that it would keep the message type unchanged.

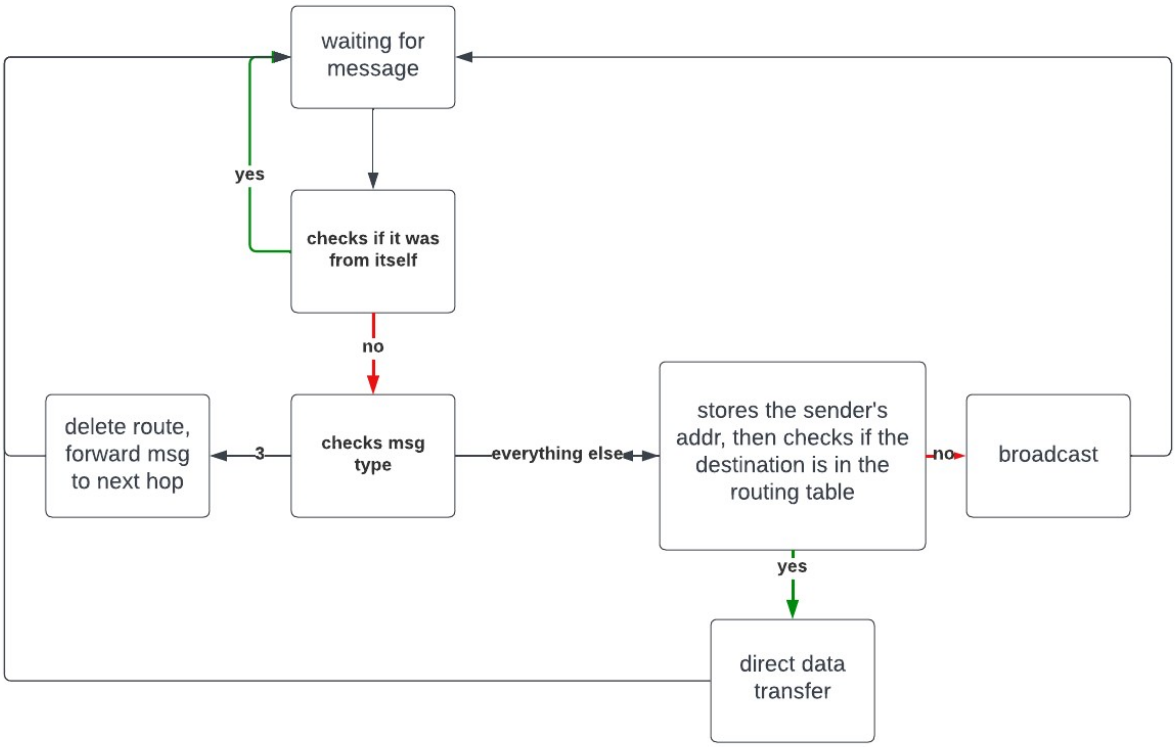
If the route does not exist, the router broadcasts the same message to all the other networks it's connected to except the one from which it got the message.

```
for interface_name, addrs in interfaces.items():
    for addr in addrs:
        if addr.family == socket.AF_INET:  # We only want to broadcast on IPv4 interfaces
            ip = addr.address.rsplit('.', 1)[0] + '.255'  # Calculate the broadcast IP for
            if senderip.startswith(addr.address.rsplit('.', 1)[0]):
                continue  # Skip this network
            else:
                sock.sendto(dest+originalSender+bytes.fromhex(address)+hexMsgType+message, (ip, port))
print("broadA")
```

For example, if there were routers in this order:
Router 2 got the data from Router 1 through Network2.
Router 2 will check for a potential route to the destination endpoint. If there is no such route, Router2 will broadcast the message on every network except Network2 since it got the message through that network.

Theoretically, this should only happen with the search message, and when the actual message is being sent, a route should have been already created. This way, other endpoints would not get important data from two connected endpoints.





Router functionality in a nutshell

4 Running the application

In order to run the application, use the included Dockerfile. (It's the same as the python example one, but with pip and psutil already installed.
Run:
docker build -t csnetimage .

After creating the images, create the containers, ideally 4 endpoints and 5 routers.
(Ideally name the containers as endpoints, and nodes/routers)

docker create -ti --name [name] --cap-add=all -v [pathToTheFolder]:/compnets csnetimage /bin/bash

For example:

docker create -ti --name producer --cap-add=all -v C:\Users\balin\Documents\GitHub\a2python:/compnets csnetimage /bin/bash

You should have 9 containers now.

Create the networks between routers and endpoints:

In my example I used 8 networks with the naming net1-net4,downnet1-2 and upnet1-2 for more clarity, but it should work with any number of endpoint/router/network, if there is no loop.

docker network create -d bridge –subnet 172.20.0.0/16 net1

docker network create -d bridge –subnet 172.21.0.0/16 net2

….

You should have 8 networks

Connect the containers to the network with.

docker network connect [network name] [container name]

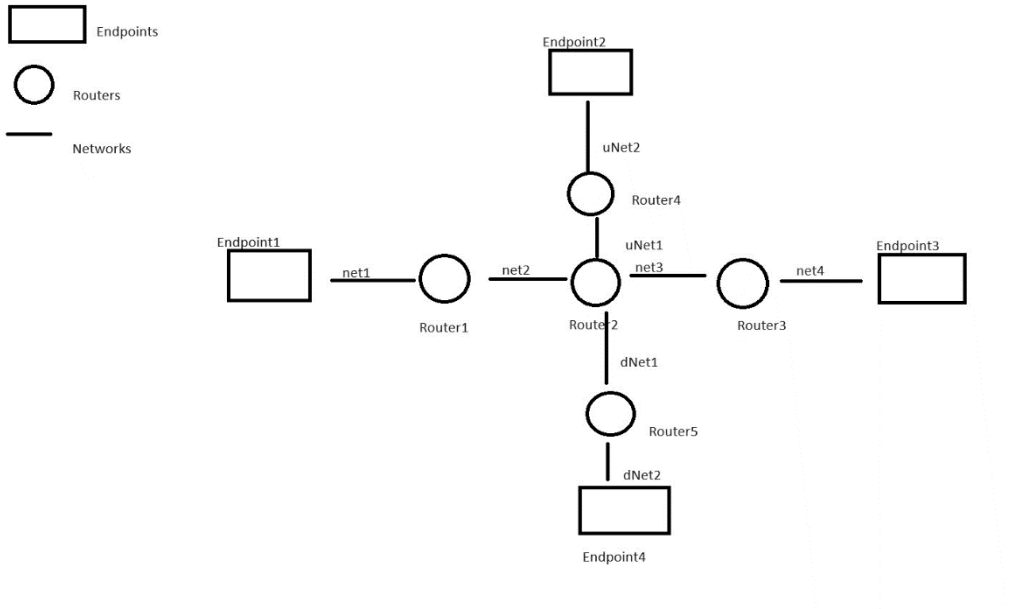
For example, assuming the containers are named as ep1 and node1:

docker network connect net1 ep1.

docker network connect net1 node1

Do this for all the nodes and and routers.

So in the end the topology should look something like this.



4.1 IMPORTANT

Do not forget to remove the bridge network that Docker assigns initially.
docker network disconnect [network name] [container name]
For example:
docker network disconnect bridge ep1

Run the containers by:

docker start -i [container name]

If everything works correctly, by running ls all the files should have been listed.

After opening the containers for atleast two endpoints and two routers, run the scripts by writing

python3 ep1.py

python3 ep2.py

for the endpoints

python3 node1.py

python3 node2.py
for the routers.

Ideally you should see something like this:

This is just the basic setup, but it works with a more complex setup, like the one shown in section 2.1 or the video as part of the submission for part 2.1.

Assuming that, ep2 has the ID of AA03 we can try to send a message from ep1.

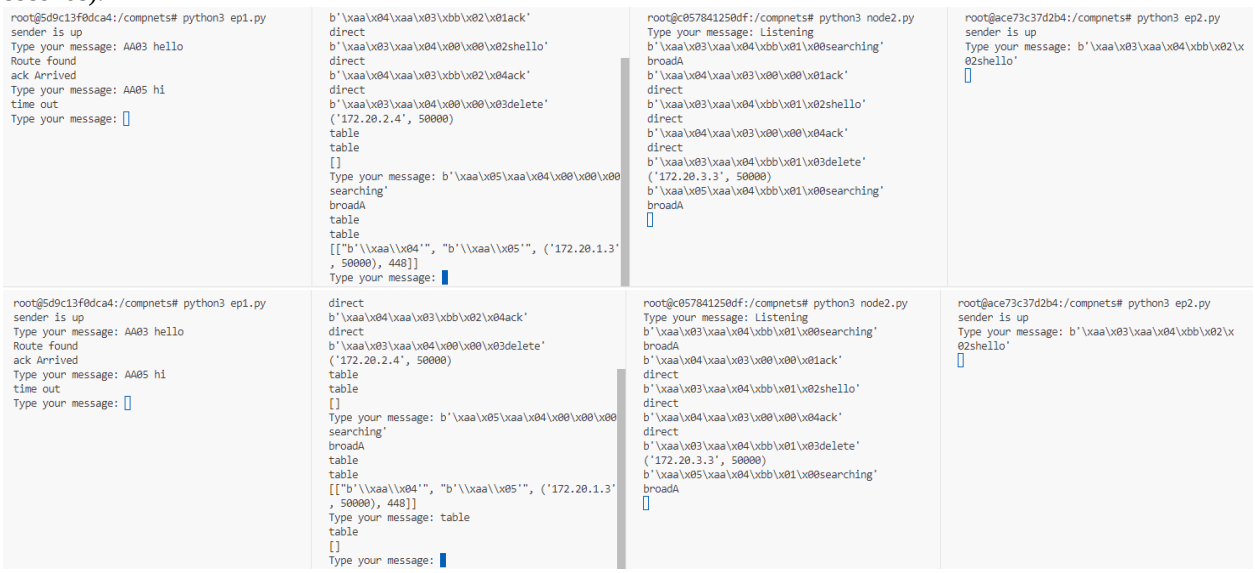
The message arrived in ep2 and went through the routers by searching for a route, sending a direct message to the endpoint, and deleting the route.

To check if the route was indeed deleted, we can review it by typing “table” into one of the routers.

We can check what happens if we are trying to send a message to a non-existent endpoint by typing a non-existent ID and then a message.

As we can see, the routing table initially stored the address, but the message never arrived at the destination. Since the destination does not exist, the sender had a timeout. (The message at ep2 is still from the previous message, which was sent to ep2.)

After 5 seconds, if we run the “table” command, we should see that the route has been indeed deleted after 5 seconds).



5 Reflection

Overall, I enjoyed doing this project, but there are still some things I would have modified if I had time. For example, to send messages to multiple endpoints, I wanted to have a function that would create threads every time there is a search message so that each endpoint could communicate with more than two endpoints simultaneously. However, it would have complicated the project more since I would have had hashmaps to track who could send what. The assignment provided me with extensive experience in Docker and flow control.