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

CSU33031 Computer Networks
Assignment #2: Flow Forwarding

Ilia Romanov, Std# 23356952

December 3, 2023

Contents

1 Introduction **2**

2 Technical Background **2**
2.1 Relevant Course Concepts 2
2.1.1 Distance Vector Routing 2
2.1.2 User Datagram Protocol (UDP) 2
2.2 Technologies Used 2
2.2.1 Docker and Docker Compose 2
2.2.2 Python 3

3 Design **3**
3.1 Header Design 3
3.1.1 Overall Anatomy 3
3.1.2 Field Details 4
3.2 Forwarding Table Design 5
3.2.1 Anatomy 5
3.2.2 Field Details 6

4 Implementation **6**
4.1 Sample Topology 6
4.2 Routing and General Flow in my System 7
4.3 Actors 9
4.3.1 Client 9
4.3.2 Router 9

5 Discussion **9**

6 Reflection **10**

1 Introduction

This report describes my solution to the task of creating a network protocol for dynamically forwarding content between ”endpoint” nodes on separate networks through a series of ”router” nodes on connecting networks. A key requirement of this task is that routers and endpoints initially must not know direct IP address information of any other nodes in the system and must communicate and figure out paths between endpoint destinations based on broadcasting to nodes on their local networks.
In this report I will begin by providing background on relevant course concepts and technologies I used for my solution. I will then give an overview of my solution design for packet headers and forwarding table structure, followed by a discussion of my implementation of the solution including the network topology I used, how the routing in my system is performed, and an overview of the actors within my system. Lastly, I will conclude the report with a discussion of the strengths and weaknesses of my solution and a reflection on my experience completing this project.

2 Technical Background

In this section I will discuss course concepts relevant to the design of my solution as well as the technologies I used for implementing my solution.

2.1 Relevant Course Concepts

2.1.1 Distance Vector Routing

Distance Vector Routing is an approach used for determining paths between elements in distributed networks. The key concepts of this approach are that it relies on communication with neighbors and exchanging forwarding table information for the purpose of building routes between specific nodes in distributed network systems. Each node stores its own forwarding table and using communications with neighbors, populates its table with information such as which neighbor - also known as a ”Next Hop” - to go to next in order to reach another specific node in the system, and the number of ”Hops” it would take to reach this specific node.

While this approach to routing is the most relevant approach from the lectures to my solution, I only draw some general concepts from this method and still deviate quite a bit from it in my design and implementation. For specifics on how I draw from this approach in my implementation, you may read the ”Routing and General Flow in my System” subsection of the ”Implementation” part of my report below.

2.1.2 User Datagram Protocol (UDP)

As discussed in lecture and Assignment 1, UDP is a protocol that allows for sending and receiving packets over IP. I believe it is worth mentioning in the technical background section for this assignment since like for Assignment 1, I will be building my custom protocol for this assignment on top of UDP. UDP already contains a header which is 8 bytes long. Hence, for my custom protocol, I will be adding my own header in the body of UDP datagrams/packets and treating it as the actual header in my protocol.

2.2 Technologies Used

2.2.1 Docker and Docker Compose

Running my solution is made possible using Docker - a software that enables the creation and execution of isolated containers as well as networks to which the containers may be attached to and communicate over. Each actor that will be seen in my overall topology in the implementation section below runs in its own isolated Docker container. Additionally, for this assignment, I used docker compose for configuring and running my topology of networks and containers all from a single configuration YAML file (`compose.yaml` in my submitted code). This allowed me to run all the containers in my topology using a single terminal command, and have them execute my code concurrently, such as sending flows of packets between endpoints in my system. Docker compose also allowed me to easily pass environment variables specific to each container, for example, to allow each router actor in my system to get its own IP addresses from these environment variables and then be use it for determining the connected network's broadcasting addresses.

2.2.2 Python

The logic for the actors in my system and the communication actions between them are implemented using Python3. Python3 has a built-in module - `socket` - for creating UDP sockets and transporting packets over them. I wrote a module/class on top of the built-in `socket` module implementing the base functionality required for my custom protocol such as broadcasting, receiving, packing, and unpacking packets and corresponding headers as well as file data to be forwarded in my custom protocol. Both of the actor classes in my system - Client and Router - inherit from this base class and use it for basic protocol actions such as broadcasting and receiving messages. This class can be found in `ProtocolSocketBase.py` in my attached code.

Python3 also has built-in functionality for parsing text files and converting string type data into bytes. I used these Python features to allow for text file transfer in the form of bytes within my system.

3 Design

The two key components of my solution are the protocol header which is present in every packet sent over my protocol and used for enabling the communication and processing of different packet types within my system, and the forwarding tables which are located on each router node in my system and used for determining routing actions for finding paths between clients in my solution. This section is aimed at describing my design for these two key components. For the header, I will describe its overall anatomy, and then use screenshots of network traffic capture files being viewed through Wireshark as examples for use cases of each of the header fields.

3.1 Header Design

3.1.1 Overall Anatomy

My header consists of 6 bytes in total, broken down as shown in the example header in Figure 1 below:

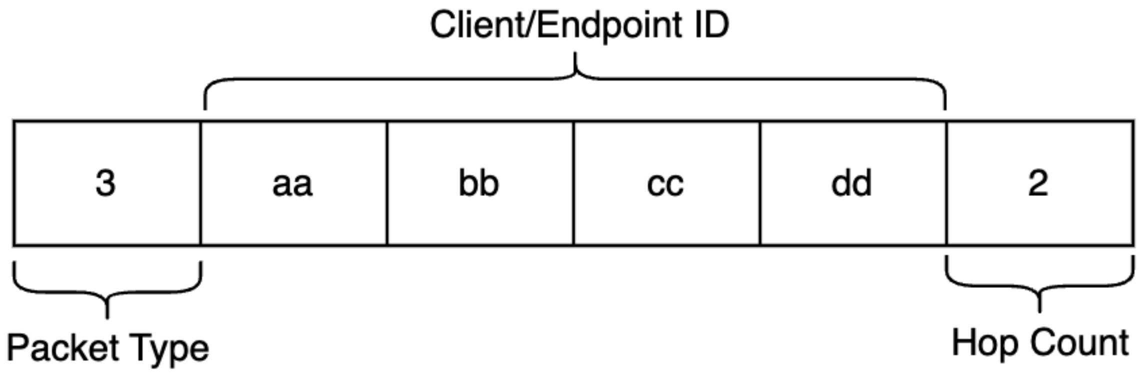


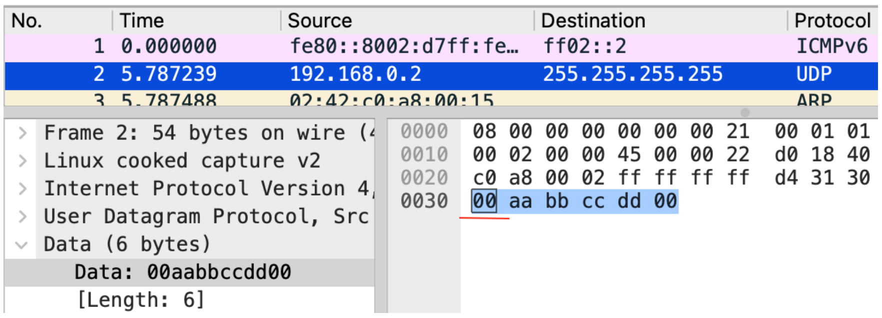
Figure 1: This figure shows an example header in my protocol. Each rectangle in this figure is to be interpreted as a single byte. The sample header in this figure shows a header with packet type of 3, client id of AA:BB:CC:DD (converted to hex bytes in the header), and a hop count of 2.

As can be seen in Figure 1, the first byte of the header stores the Packet Type, the next four bytes are an endpoint address / ID, and the last byte is for hop count.

3.1.2 Field Details

Packet Type: Needed in the code for each actor in my solution to determine how a received packet should be handled. The packet types I defined for my system are:

- `ANNOUNCE_ENDPOINT = 0` - used for packet sent by newly created client to edge router on startup. Per the assignment specification - an edge router is the single router that shares a network with an endpoint/client.
- `ANNOUNCE_ENDPOINT_ACK = 1` - the edge router replies with this packet type to let the client know its IP and confirm that connection to the system has been established.
- `FWD_REQUEST = 2` - used when any actor sends a request to forward to a specific destination. Used for both broadcast and known next hop (direct send) packets.
- `FWD_REPLY = 3` - used in reply packets to broadcasted forward requests.
- `CLEAR_REQUEST = 4` - used for packets that are sent as a request to clear a specific client/endpoint address from all router's forwarding tables in the system.

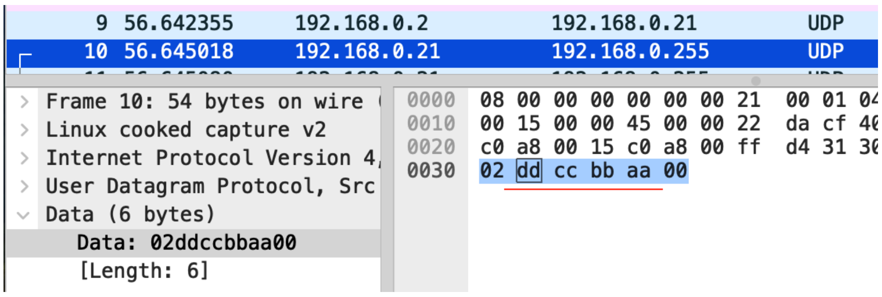


```
No.   Time        Source                 Destination       Protocol
   1  0.000000    fe80::8002:d7ff:fe…    ff02::2           ICMPv6
   2  5.787239    192.168.0.2            255.255.255.255   UDP
   3  5.787488    02:42:c0:a8:00:15                        ARP

> Frame 2: 54 bytes on wire (    0000  08 00 00 00 00 00 00 21  00 01 01
> Linux cooked capture v2        0010  00 02 00 00 45 00 00 22  d0 18 40
> Internet Protocol Version 4,   0020  c0 a8 00 02 ff ff ff ff  d4 31 30
> User Datagram Protocol, Src    0030  00 aa bb cc dd 00
v Data (6 bytes)
    Data: 00aabbccdd00
    [Length: 6]
```

Figure 2: Wireshark network traffic capture showing a packet broadcasted by a client with address AA:BB:CC:DD to find its edge router. Note the Packet Type value of zero corresponding to an `ANNOUNCE_ENDPOINT` request.

Client ID: Needed to identify clients/endpoints. For example, when a router broadcasts a packet with a forward request to some specific client (shown in Figure 3), or a client/router sends a packet replying that it has been found as a result of a forward request/broadcast.

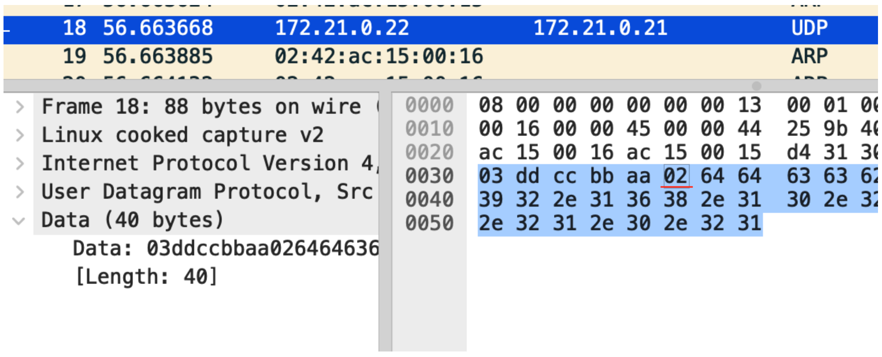


```
   9  56.642355    192.168.0.2      192.168.0.21     UDP
  10  56.645018    192.168.0.21     192.168.0.255    UDP

> Frame 10: 54 bytes on wire (    0000  08 00 00 00 00 00 00 21  00 01 04
> Linux cooked capture v2         0010  00 15 00 00 45 00 00 22  da cf 40
> Internet Protocol Version 4,    0020  c0 a8 00 15 c0 a8 00 ff  d4 31 30
> User Datagram Protocol, Src     0030  02 dd cc bb aa 00
v Data (6 bytes)
    Data: 02ddccbbaa00
    [Length: 6]
```

Figure 3: Wireshark network capture showing a packet being broadcasted by a router with IP address 192.168.0.21 requesting the forwarding of the packet to a client endpoint with id DD:CC:BB:AA. Also note that the packet type is 2 indicating the type `FWD_REQUEST`.

Hop Count: Present in reply packets, providing the number of nodes to reach a requested client (a.k.a hop count) to be placed into forwarding tables at each router which helps with finding shortest paths in my solution, error handling, and avoiding infinite execution loops. I use a hop count value of zero to indicate this field being ”null” when its not needed. The only case when this field would be used is for reply packets being sent back from a destination client and propagated through requester routers back to the client that initially sent the request for reaching the destination client (shown in Figure 4).



```
   18  56.663668      172.21.0.22             172.21.0.21             UDP
   19  56.663885      02:42:ac:15:00:16                               ARP

> Frame 18: 88 bytes on wire       0000  08 00 00 00 00 00 00 13  00 01 0
> Linux cooked capture v2          0010  00 16 00 00 45 00 00 44  25 9b 4
> Internet Protocol Version 4,     0020  ac 15 00 16 ac 15 00 15  d4 31 3
> User Datagram Protocol, Src      0030  03 dd cc bb aa 02 64 64  63 63 6
v Data (40 bytes)                  0040  39 32 2e 31 36 38 2e 31  30 2e 3
     Data: 03ddccbbaa026464636     0050  2e 32 31 2e 30 2e 32 31
     [Length: 40]
```

Figure 4: Wireshark capture showing a reply packet being sent from a router node with IP 172.21.0.22 to another router node with IP 172.21.0.21. Note that the hop count (represented by the sixth byte in the header) is 2 for this packet, meaning that the number of hops from router with IP 172.21.0.21 to client with id DD:CC:BB:AA is two; one from router 172.21.0.21 to router 172.21.0.22, and then one more from router 172.21.0.22 directly to the endpoint DD:CC:BB:AA. There is also a reply body that appears in this capture after the hop count header field which can be ignored for the purposes of this figure.

3.2 Forwarding Table Design

As mentioned briefly earlier in the report, routing in my solution is performed with the help of forwarding tables, stored at each Router node in my topology. I will only discuss the general design of the forwarding tables in this section, for a more in-depth explanation of how the forwarding tables are used to make routing decisions in my system, view Section 4.2 - ”Routing and General Flow in my System” below.

3.2.1 Anatomy

Each Router node in my system stores its forwarding table as a Python dictionary. The keys in the dictionary are client IDs, and the values are further dictionaries with three key-value pairs for Next Hop, Hop Count, and Requesters. Thus, the data type of the forward table is defined in my code as
`self._fwd_table: Dict[str, Dict[util.FwdTableKey, str | int | set]]`.
Observe the below figure for an example entry in a forwarding table in my system, noting that a forwarding table in my system is a collection of theses types of entries.

AA:BB:CC:DD	Next Hop	Hop Count	Requesters
	172.21.0.21	2	{172.21.0.22, 172.21.0.23}

Figure 5: An example entry in a forwarding table on some router in my system. This entry is indexed by a client ID of AA:BB:CC:DD. Using the data in this forward table entry, the router holding this forwarding table can deduce that to reach the client AA:BB:CC:DD, it needs to forward the packet to the router with IP 172.21.0.21, and it will take two hops to reach the client endpoint with ID AA:BB:CC:DD from the current router. Also, from this table entry the owner router knows that it received broadcasts from two other routers with IPs 172.21.0.22, and 172.21.9.23 requesting a forward to AA:BB:CC:DD, and once this owner router gets a reply propagated back from client AA:BB:CC:DD, it will forward this reply to these two requesters so they can similarly continue propagating it until it reaches the original requesting client

3.2.2 Field Details

Next Hop - Next hop is used for caching known routes towards a certain destination. For example, at the router above, since it has previously been requested to forward a packet to client AA:BB:CC:DD and received a reply, it knows that when asked to forward a packet to AA:BB:CC:DD again, the IP of the next router it should send the packet to is 172.21.0.21, instead of broadcasting to all router on its local network which would take longer.

Hop Count - As briefly mentioned in the explanation of the Hop Count field in my protocol header, the main purposes of storing hop count in my forwarding tables are for assuring the shortest path is used when forwarding information between clients, for handling errors, and for ensuring finite execution within my system. Firstly, knowing the Hop Count helps with finding the shortest paths between endpoints because my Router logic checks if for any reply packet it receives for a specific client ID, if the Hop Count in the header of the packet is smaller than the one it currently has stored in the forwarding table entry for this Client ID (assuming it has an entry already); if the received packet's hop count is smaller, the router will then update the entry to use the Next Hop corresponding to a smaller hop count towards that client ID. Second, the hop count helps with error handling in my system since for example if some error were to occur during execution leading to infinite looping/broadcasting, for example if a packet to a non-existent client ID was sent, my solution allows for the setting of a maximum Hop Count, and then any packet in which this maximum hop count value has been reached/exceeded, will be ignored by the actors in my solution eventually leading to a halt since they stop broadcasting. Similarly, this is how the hop count field helps to ensure finite execution within my system.

Requesters This field stores a set of IP addresses of routers which requested a packet forward to the entry's client ID. It is used for propagating replies through my system of routers, eventually resulting in the reply reaching the client which initially sent the forward request.

4 Implementation

In this section, I will discuss the implementation specifics of my solution. I will begin with presenting a sample topology that my solution enables, followed by an explanation of how routing decisions are made in my solution, and then finish with a brief overview of how each actor in my system operates.

4.1 Sample Topology

In my solution there are two types of actors: client and router. Each actor runs in its own separate Docker container, all orchestrated through a single `compose.yaml` file with the help of docker compose. Observe the topology below for an example of how they may be connected over a series of networks through my solution.

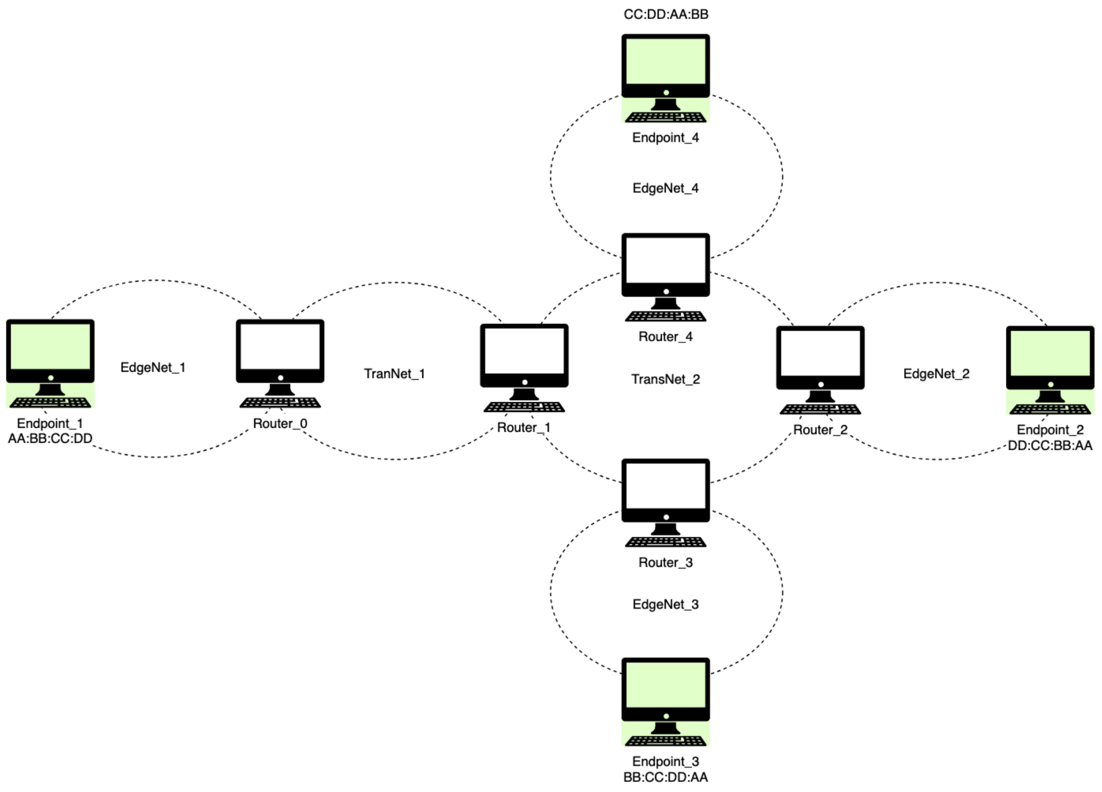


Figure 6: Diagram of a sample topology I can use to run my solution. A key takeaway from this diagram is that my solution allows for clients/endpoints to be identified by four byte IDs rather than IP addresses, and enables communication between them with the help of intermediary Router nodes which communicate using broadcasts and forwarding table information over a series of transition networks connecting them.

Note that I call the above diagram a "sample" topology since a key aspect of my implementation is that it allows for any topology consisting of any amount of Router and Endpoint nodes as long as the condition that each client/endpoint node belongs to exactly one network, and shares this network with exactly one router is met (this is assumed from the assignment specification).

4.2 Routing and General Flow in my System

Routing in my solution is facilitated through the use of broadcasts and forwarding tables performed/computed by Router nodes which form paths between different Client nodes within the system. Similarly to Direct Vector Routing, each router in my system holds and builds up its own forwarding table for determining where to forward any received packet based on the client ID it is directed to.
Observe the below flow chart in Figure 7 for an understanding of the actions a Router node in my system takes upon receiving a forward request packet.

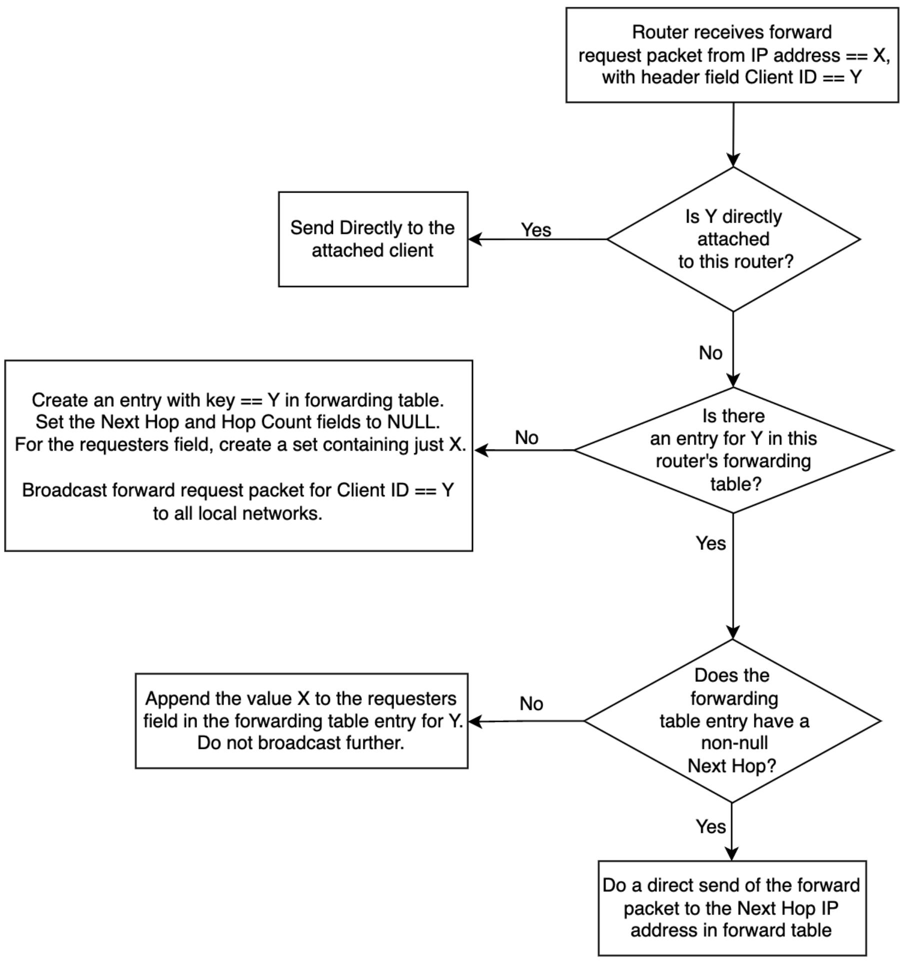

Router receives forward request packet from IP address == X, with header field Client ID == Y
Is Y directly attached to this router?
Yes
Send Directly to the attached client
No
Is there an entry for Y in this router's forwarding table?
No
Create an entry with key == Y in forwarding table. Set the Next Hop and Hop Count fields to NULL. For the requesters field, create a set containing just X.
Broadcast forward request packet for Client ID == Y to all local networks.
Yes
Does the forwarding table entry have a non-null Next Hop?
No
Append the value X to the requesters field in the forwarding table entry for Y. Do not broadcast further.
Yes
Do a direct send of the forward packet to the Next Hop IP address in forward table

Figure 7: This diagram depicts the routing flow once a router in my system receives a forward request packet. To clarify the first step, a Client being directly attached to a router means that the router is an edge router and shares a network with the requested client, in this case client Y. Note how broadcasting is only done if there is no entry for the requested client ID in the router's forwarding table. Otherwise we only add an extra requester to reply to, or we already know the Next Hop and can forward the packet.

Using the routing decisions in Figure 7 above, the forward request will eventually reach its requested destination Client, at which point the client node will send a reply packet back to the edge router to initiate a propagation of replies through the system back to the client which requested the original forward.

Once a router receives a reply to a broadcast it performed for a specific client ID, it will check the Hop count value in the forwarding table entry corresponding to the client ID. If the forwarding table Hop Count value is not Null and is less than the Hop Count value in the header of the reply packet, the router will ignore this reply packet (since a shorter route is known). If the forwarding table Next Hop value is NULL, or the Hop Count in the forwarding table entry is greater or equal to the Hop Count in the reply packet header, the router will update the forwarding table entry Next Hop value with the IP address of the sender of the reply packet, and the Hop Count value with the Hop Count in the received packet header. This router will then iterate over the IP addresses and in the "requesters" field in its forwarding table and send each of them the reply packet, now with an incremented hop count.

4.3 Actors

4.3.1 Client

In my solution Clients (a.k.a endpoints) are capable of: attaching to an edge router, sending data forwarding requests of both files or simple string messages directed to other clients, and sending requests to clear all systems Router's forwarding table entries for a specific client ID in the system.

Attaching to an edge router is done by the client performing an initial broadcast to its local network with a ANNOUNCE_ENDPOINT request. This broadcast is guaranteed to reach exactly one router (the corresponding edge router) since that should be the only other node attached to the same network as the Client as discussed in Section 4.1 - "Sample Topology" above.

Sending requests to forward data is done by simply sending a message to the attached edge router. Each client should have an attached edge router after initialization. My solution allows for command line control over what to send in theses data requests with options to specify a file path or simply send a ping to another endpoint, both achieved with the user inputting a target endpoint ID.

Lastly, sending requests to clear forwarding table entries for a specified endpoint ID is very similar to sending request to forward data. My solution makes this available through the command line, and allows users to specify the client ID to be cleared. The client then sends this clear requests to its attached edge router.

My implementation for the client logic and command line interface for it can be found in `Client.py` and `client_main.py` in my attached code.

4.3.2 Router

In my solution Routers are responsible for making routing decisions in forwarding packets between client nodes as well as facilitating the replies. They are also capable of accepting and forwarding clear requests from clients for clearing specific client ID entries from each Router's forwarding table.

For the overall logic of how Router's routing and reply decisions are made, please view Section 4.2 - "Routing and General Flow in my System" above. The way processing multiple forwarding/reply packets is enabled through my Router code is using a separate threads for listening and replying to packets. On initialization of each Router in my system, I start a secondary thread to listen for incoming packets and process them accordingly.

For the processing of clearing requests, once a Router receives the request, it checks if the requested client ID is in its forwarding table. If it is, then it clears the entry from its forwarding table and send out a broadcast requesting to clear this client ID to its local networks. If the client ID is not in the Router's forwarding table, it just ignores the request since it has already cleared it and sent the broadcast to its neighbors.

My implementation for the router logic and command line interface for it can be found in `Router.py` and `router_main.py` in my attached code.

5 Discussion

I believe the main advantage of my solution is its ability to accommodate changes and scale up in terms of both the number of actors/networks being added to the system, as well as in the number of packets being sent between the client nodes in my solution. The docker compose setup allows for the addition of any number of new actors to my system; since each Client in my solution would be running the same Client module code, and each router would be running the same Router module code, no changes in the code need to be made, and the only changes would be the environment variables passed through the single docker compose configuration file. The multi-threading in my solution would also allow for large flows of packets to

be forwarded concurrently within my system. Lastly, another advantage of my system is the error handling and ability to ensure finite execution which I discussed in several sections within this report.
An area where I believe my solution can be improved is its usability. More specifically, the ability for users to control which information is sent at which times. For example, the only way to manually send information at any desired time from a specific client within my system is to change the container running it in the docker compose yaml file, telling it to run /bin/bash instead of the Client module code, and then attach to it using docker attach. Similarly, to make Clients send specific forward requests automatically on startup of the system, users would need to change the compose.yaml to trigger these requests.

6 Reflection

Overall, I enjoyed working on this assignment and I feel that I now have a better grasp on packet flow forwarding systems and creating protocols for the purpose of such tasks. I would say I spent about 40 to 50 hours of active work on this project and most of it went to trying to make my code as readable and extendable as possible. Thus, if I were to do something different for this assignment, I would focus more on the specifics of the task at hand rather than trying to solve larger scale ideas out of the scope of the requirements of the assignment.
A mistake I think I made in approaching this assignment is starting to think too broadly of the task at hand and considering additional features of the task before ensuring that I had the basics working, especially with figuring out broadcasting. I was not able to get the broadcasting to work as I intended for the part 1 deadline because I was too focused on implementing the additional features in my solution and making my code extendable and readable before ensuring that the key aspects of broadcasting worked properly.