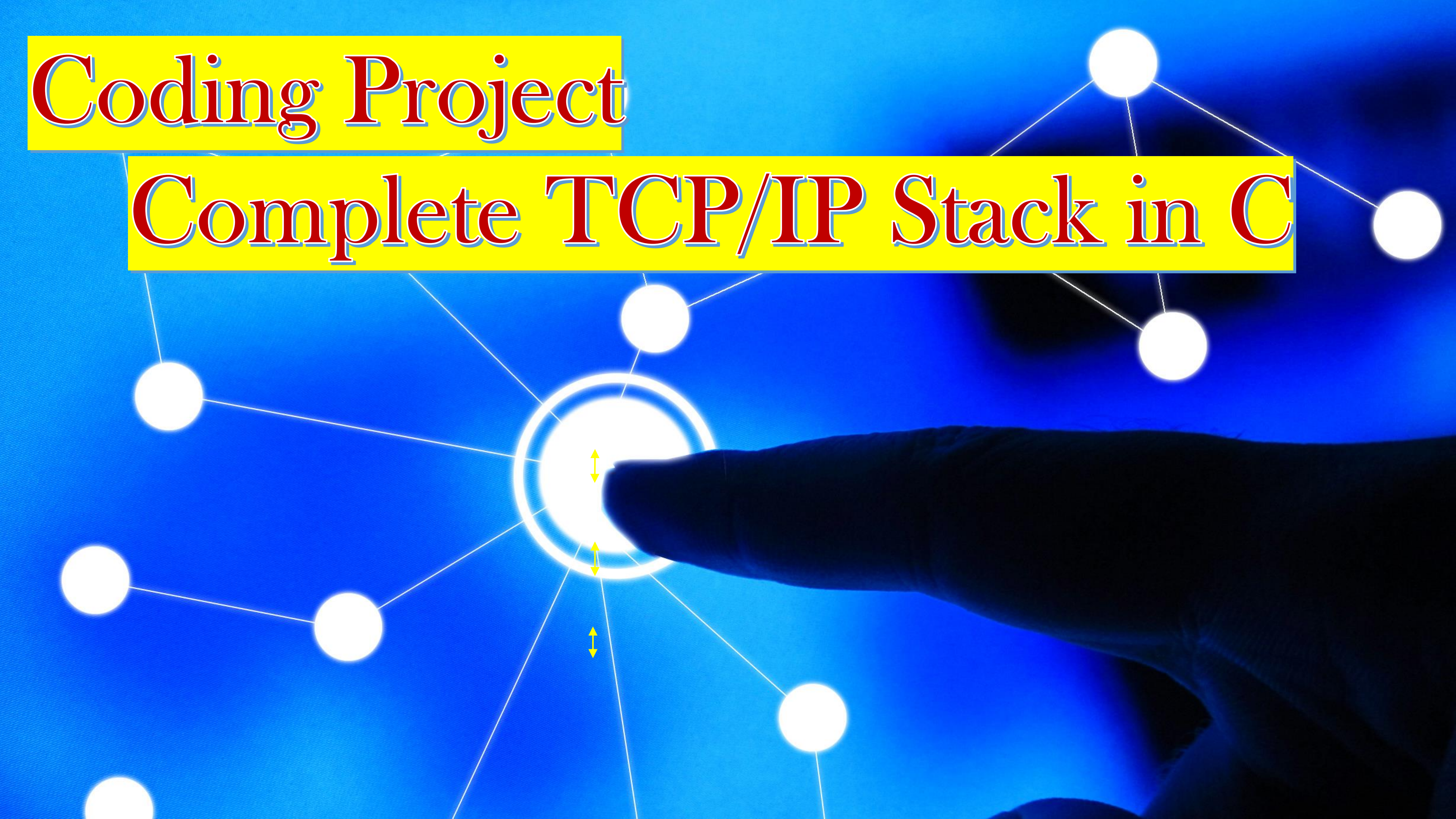
Coding Project

Complete TCP/IP Stack in C

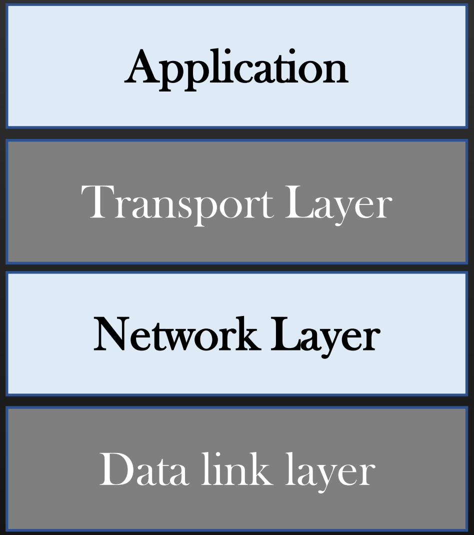


Developing TCP/IP Stack

Agenda

Implement in C :

ARP
Layer 2 Routing Complete
Layer 3 routing
ICMP, Ping , Trace-Route
Vlan Based Routing - All In one Project !!



Thanksgiving ☺

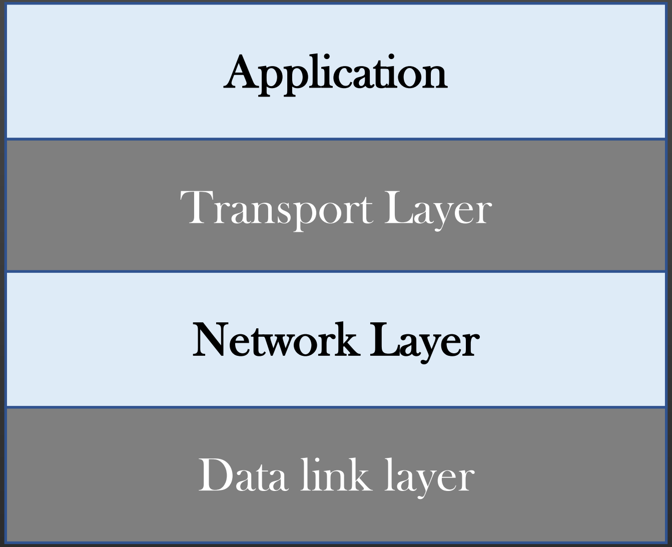
I would like to thanks some prominent personalities who encouraged me to develop this project.
The Motive is to develop a framework which could be used to try And test Networking Solutions

Miss. Twisha Nigam
- Sr. Staff Engineer – Cisco Systems

Mr. Manu Kumar
- Staff Software Engineer – Alcatel Lucent

Agenda

Developing TCP/IP Stack



PART A

1. Project Goals
2. Setting up Generic Graph
3. Setting up Network Topology
4. Integrating Command Line Interface
5. Packet Exchange Simulation Infra Setup

PART B

6. Layer 2 – ARP, MAC Forwarding
7. Layer 3 – Route Installation, L3 Routing
8. Application – Ping, Traceroute

Pre-requisite

Developing TCP/IP Stack

1. Thorough with C or C++
2. Linux Development environment
3. Basic Networking knowledge is essential
   Layer 2 , Layer 3
4. Basic UDP Socket Programming,
   Minimal Multithreading
5. Working with Git – Very important
6. Compilation, Makefile

Complexity Level : Intermediate to Advanced
(Not for absolute beginners)

Warning : Don't skip assignments in this course, else you
Won't be able to progress further

Codes written in assignment shall be used in the project

Project Extensions

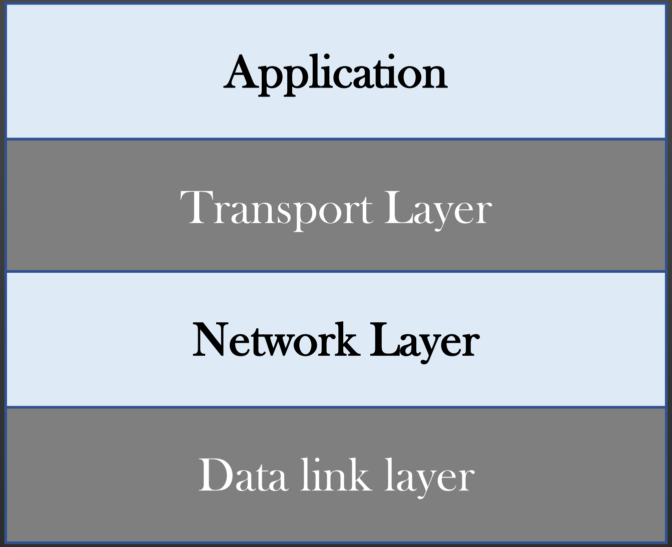
- Nobody is Stopping you to implement VLAN functionality
- Nobody us stopping you to implement IP Fragmentation
- Nobody is stopping you to implement various other protocols :
  IP-in-IP encapsulation , Tunnels
- Unlimited Scope !
- This Course shall transform you into a Networking Developer !

Project Goals

- This Course is the Practical Version of Actual OSI Model
- You shall be implementing Layer 2 and Layer 3 functionality from Scratch
- You shall be implementing all logic to parse the packet content, and take decision what to do with the packet
- You shall be implementing Traffic forwarding pipeline
- We shall be building up the topology where nodes would represent Layer3 routers and/or L2 switch or Hub. In other words, we shall be writing simplified code for L2 switching and L3 routing
- We shall be using CLI to configure our nodes (routers and switches)
- You don't need multiple machines, all shall be done on one machine, within our project !
- Take Away :
  - You shall have low level thorough knowledge of TCP/IP Stack functioning
  - Learn how to parse the packet, evaluate packet hdr content, and take action accordingly
  - A Strong candidature and portfolio to join Networking development roles Or otherwise
  - Open ended Project – You can grow old working on this project, but this project wont end

Agenda

Developing TCP/IP Stack



PART A

1. Project Goals
2. Setting up Generic Graph
3. Setting up Network Topology
4. Integrating Command Line Interface
5. Packet Exchange Simulation Infra Setup

PART B

6. Layer 2 – ARP, MAC Forwarding
7. Layer 3 – Route Installation, L3 Routing
8. Application – Ping, Traceroute

Generic Graph Construction

- First, We shall develop a library using which we can create static graph

- A static graph, as we know, consists of nodes, edges, cost of edges

- This Graph can be used to implement for other purposes :
    - Network Topology (this course)
    - Routing protocols development
    - Practice your Graph algorithms (Dijkstra etc)

- As of now, graph nodes are simple nodes, they shall not represent routing devices

- In next section, we shall extend our graph to represent Network Topology

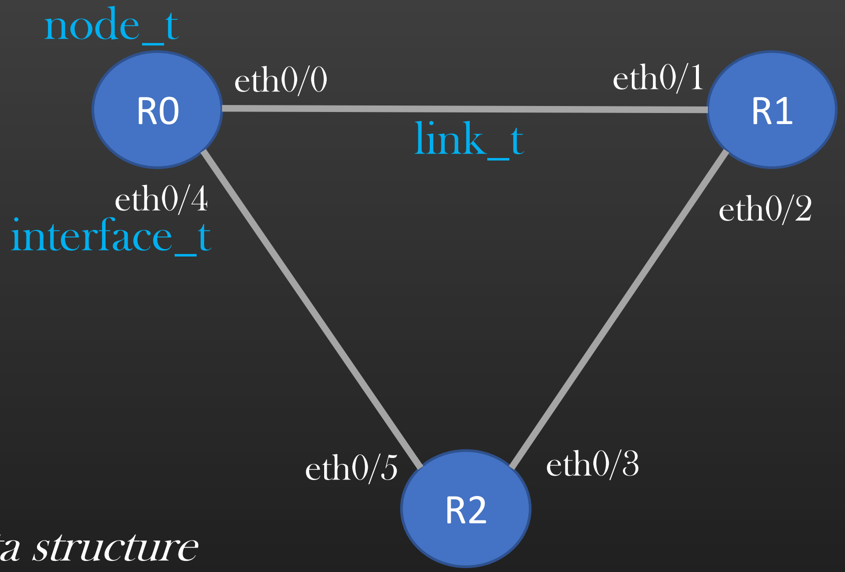
- All Source Codes :
  http://github.com/sachinites/tcpip_stack (Pre-Completed)

Generic Graph Construction

- Files to be created :

  graph.h , graph.c, testapp.c

- A graph is a collection of nodes
- An interface has a :
  - Name
  - Owning node
  - A wire (or link)
- A link is defined as pair of interfaces
- A node has a :
  - Name
  - Set of empty interface slots



Tip : Try to model Data structure Such that it depicts the Organization of information In real physical world

Generic Graph Construction

- Files to be created :

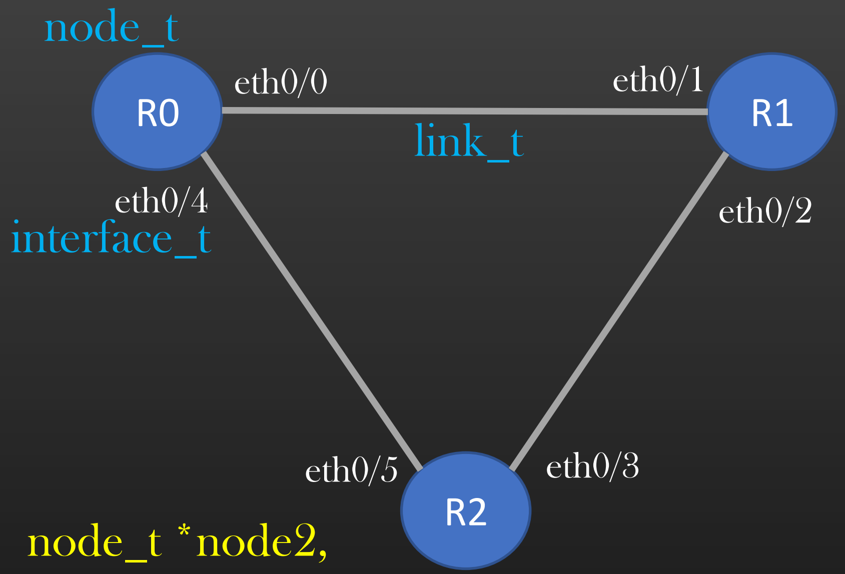
  graph.h , graph.c, testapp.c

- Public APIs

  graph_t *create_new_graph(char *topology_name);

  node_t *create_graph_node(graph_t *graph, char *node_name);

  void insert_link_between_two_nodes (node_t *node1,
  node_t *node2,
  char *from_if_name,
  char *to_if_name,
  unsigned int cost);



Display Routines :
void dump_graph (graph_t *graph);

Our First Graph : topologies.c

Writing a Project Makefile

- We have the following src files so far :
  - gluethread/glthread.c
  - graph.c
  - topologies.c
  - testapp.c

- Let us quickly setup a Makefile of our project

Agenda

Developing TCP/IP Stack

Application
Transport Layer
Network Layer
Data link layer

PART A

1. Project Goals
2. Setting up Generic Graph
3. Setting up Network Topology
4. Integrating Command Line Interface
5. Packet Exchange Simulation Infra Setup

PART B

6. Layer 2 – ARP, MAC Forwarding
7. Layer 3 – Route Installation, L3 Routing
8. Application – Ping, Traceroute

PART C – Sequel Course

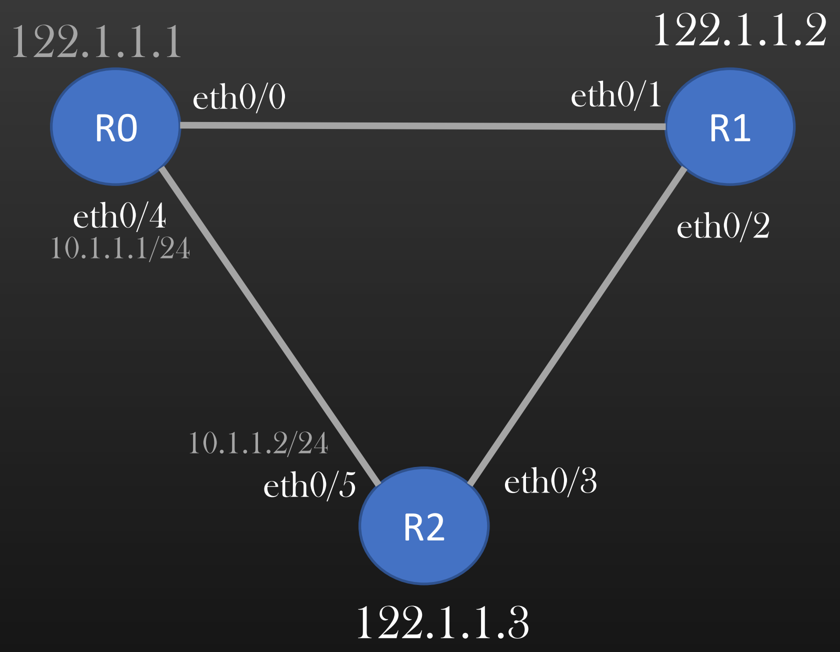
Dynamic Construction of Layer 3 Routing Table

Setting up Network Topology

- We shall extend our generic graph to represent the network topology
- We shall be adding Network Parameters to node_t, interface_t structures

```
struct node_ {
. . .
node_nw_prop_t  node_nw_prop;
. . .
};
```

```
struct interface_ {
. . .
intf_nw_props_t  intf_nw_props;
. . .
};
```

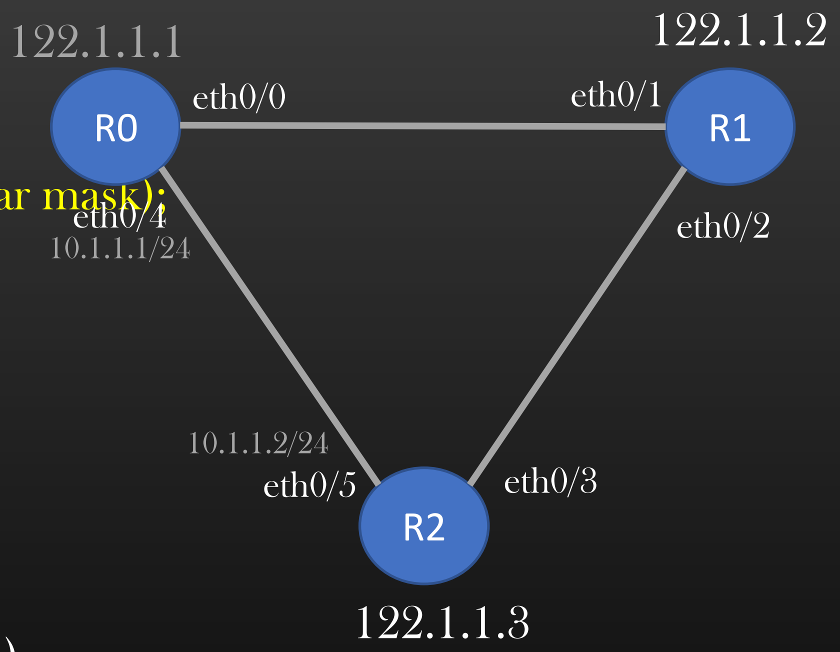


- Data structures/APIs related to network config shall be defined in net.h/net.c
- Every node has its own IP Address , called as loopback address
- Every interface MUST have mac address, and MAY have ip-address/mask

Setting up Network Topology

- Public APIs
  Declare in net.h, define in net.c , use in testapp.c

```
bool_t node_set_loopback_address(node_t *node, char *ip_addr);

bool_t node_set_intf_ip_address(node_t *node, char *local_if,
                                  char *ip_addr, char mask);

bool_t node_unset_intf_ip_address(node_t *node, char *local_if);
```



- As soon, as you add a link to the topology connecting two nodes, the end
  Interfaces must be assigned some auto generated mac addresses

  Mac_address generated = fn (node_name, interface_name, some heuristics)

```
void interface_assign_mac_address(interface_t *interface);
```

Display function – Display the entire network topology with networking properties also

```
void dump_nw_graph(graph_t *graph);
```

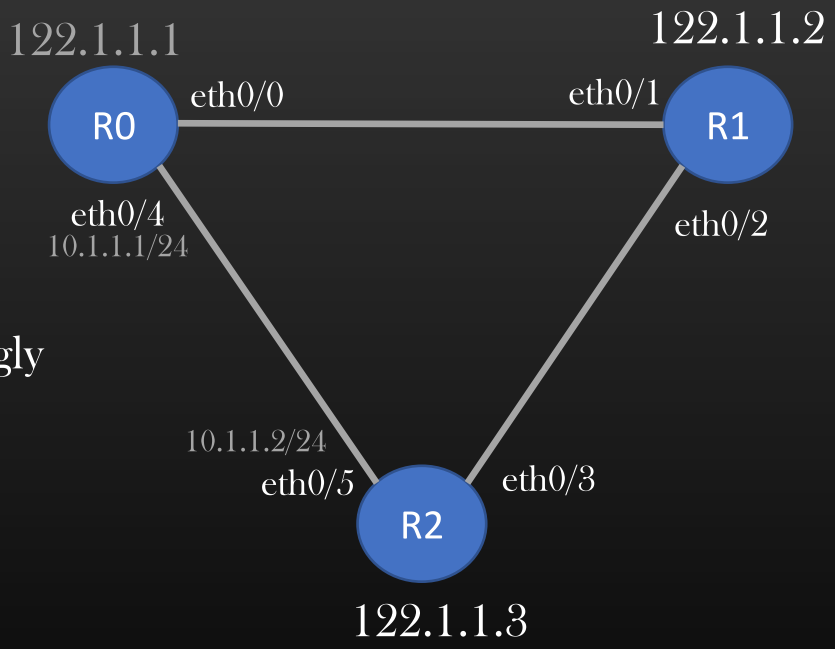
Setting up Network Topology

- Enhance build_first_topo() in topolgoes.c to add networking parameters to the graph

- Display function – Display the entire network topology with networking properties also
  net.h/net.c

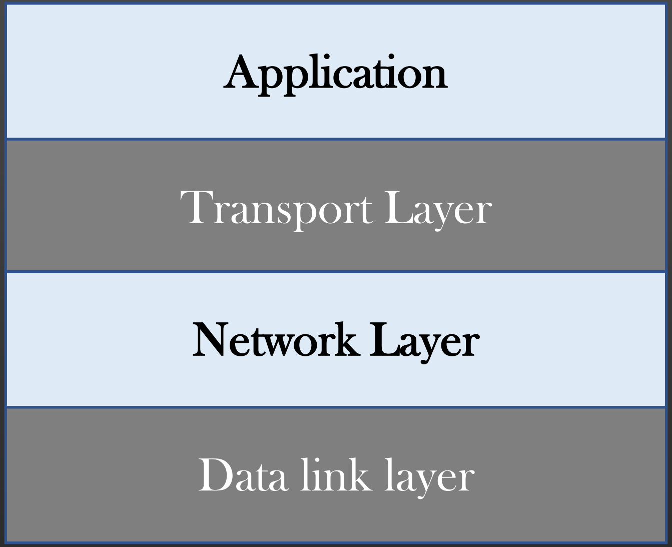
  void dump_nw_graph(graph_t *graph);

- As we progress into the course, we shall need to add more networking properties to the nodes and interfaces. We shall be defining more new members to *node_nw_prop_t* and *intf_nw_props_t* structures accordingly



Agenda

Developing TCP/IP Stack



PART A

1. Project Goals
2. Setting up Generic Graph
3. Setting up Network Topology
4. Integrating Command Line Interface
5. Packet Exchange Simulation Infra Setup

PART B

6. Layer 2 – ARP, MAC Forwarding
7. Layer 3 – Route Installation, L3 Routing
8. Application – Ping, Traceroute

PART C – Sequel Course

Dynamic Construction of Layer 3 Routing Table

CLI Integration

- User Configures/Interact with routing devices through CLI interfaces

- Let me show you Juniper Actual Router

- We shall be needing an external CLI library using which we can implement our own customize show, config, clear commands

- We shall be using the CLI to reconfigure our network topology, display information etc

- Pre-requisite :
  - You need to do the 80-minute course (Link to the course in Resource Section)
    to understand how to use CLI library, then comeback !
    Pls do assignments in the course to get a hands-on the libcli library

  - You can use this CLI library in future for your other C/C++ projects
  - Once you come back, we shall be implementing Demo commands to our project using CLI library

CLI Integration

- Inside your project directory, download libcli library code

  git clone http://github.com/sachinites/CommandParser

- In CommandParser dir, delete the hidden dir .git

- Update Project Makefile to integrate libcli library

- Verify Compilation

- Run the cmd from inside tcpip_stack/

  git add CommandParser

- Commit

CLI Integration

- Implement Commands :
  - show topology

Files to be modified :

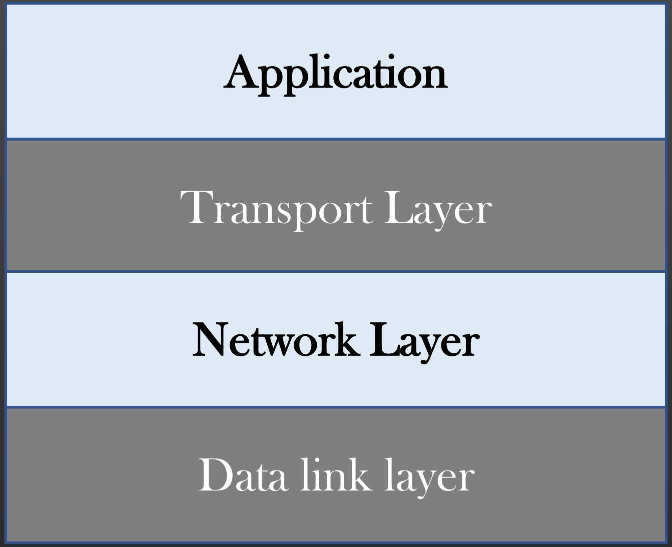
nwcli.c

cmdcodes.h

testapp.c

Agenda

Developing TCP/IP Stack



PART A

1. Project Goals
2. Setting up Generic Graph
3. Setting up Network Topology
4. Integrating Command Line Interface
5. Packet Exchange Simulation Infra Setup

PART B

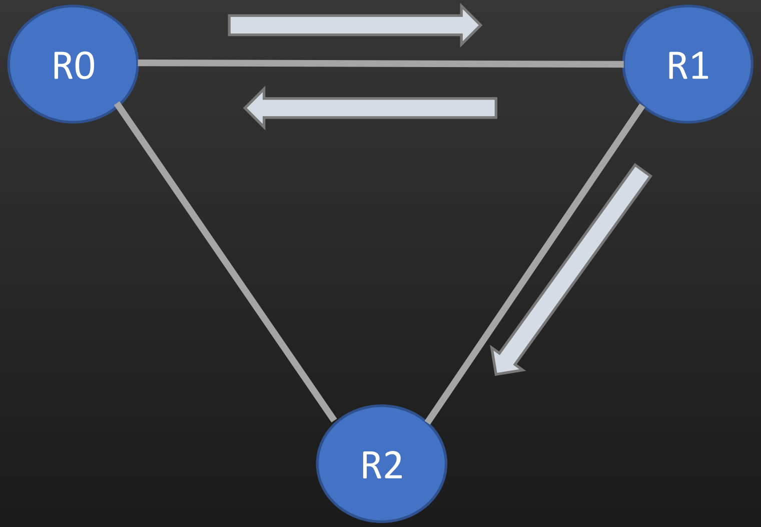
6. Layer 2 – ARP, MAC Forwarding
7. Layer 3 – Route Installation, L3 Routing
8. Application – Ping, Traceroute

PART C – Sequel Course

Dynamic Construction of Layer 3 Routing Table

Packet Exchange Simulation Infra Setup

- Now that our network graph are fully setup, and we can also interact with our Routing Devices using CLI . . .
- It's a time to setup the framework using which Nodes can exchange data/packets with direct peers



- Goal : Implement the below public APIs in comm.h/comm.c

```
int send_pkt_out (char *pkt, unsigned int pkt_size, interface_t *oif );
int pkt_receive ( node_t *node, interface_t *iintf, char *pkt, unsigned int pkt_size);
int send_pkt_flood (node_t *node, char *pkt, unsigned int pkt_size);
```

Pre-requisite :

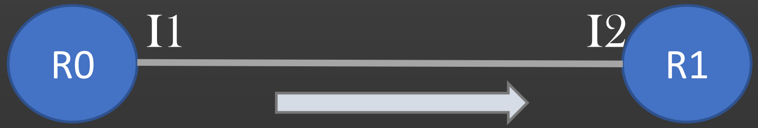
1. UDP client server program
2. select()
3. Start a thread

Now we need to do some simulation to achieve our goal, as our nodes are virtual nodes !

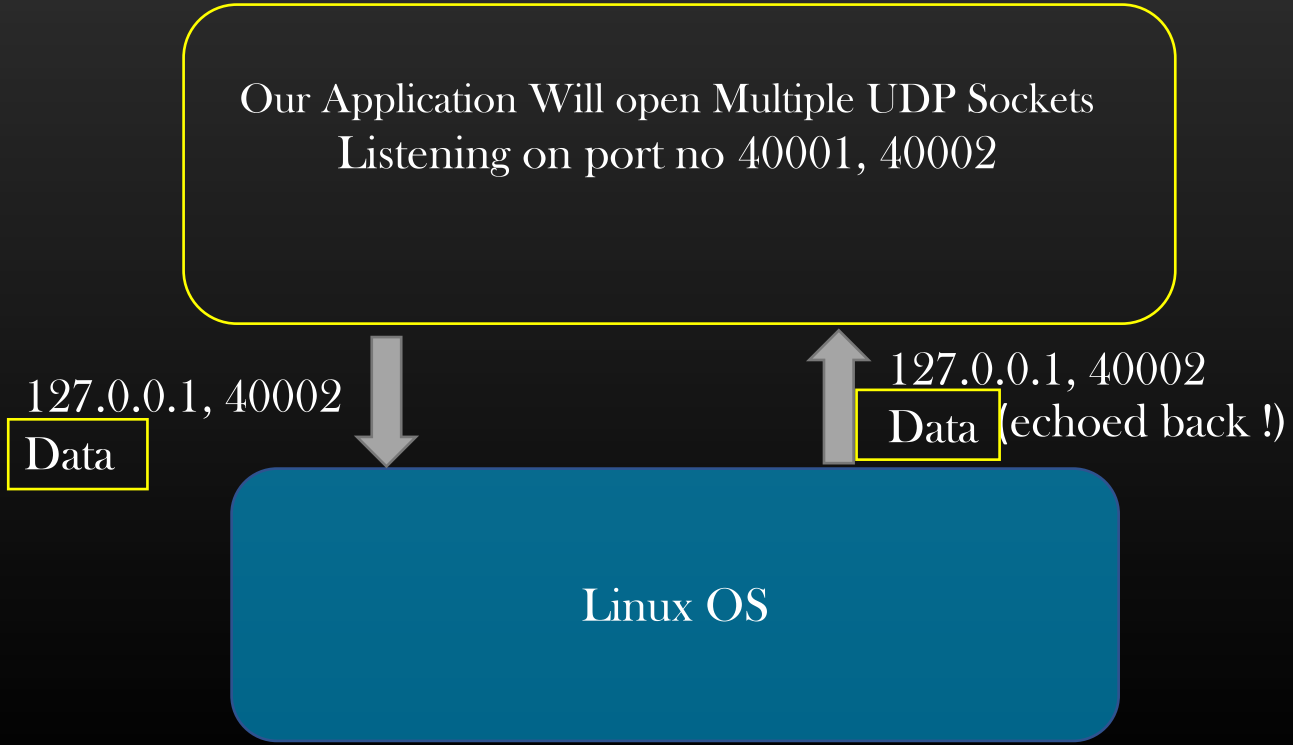
Packet Exchange Simulation Infra Setup

- Design Discussion to implement Communication between nodes over links

Logical View :



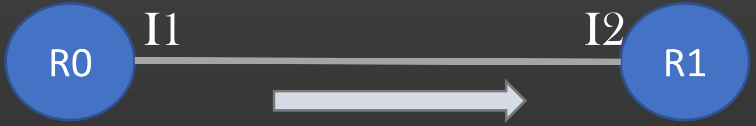
Actual View :


Our Application Will open Multiple UDP Sockets
Listening on port no 40001, 40002
127.0.0.1, 40002
Data
127.0.0.1, 40002
Data (echoed back !)
Linux OS

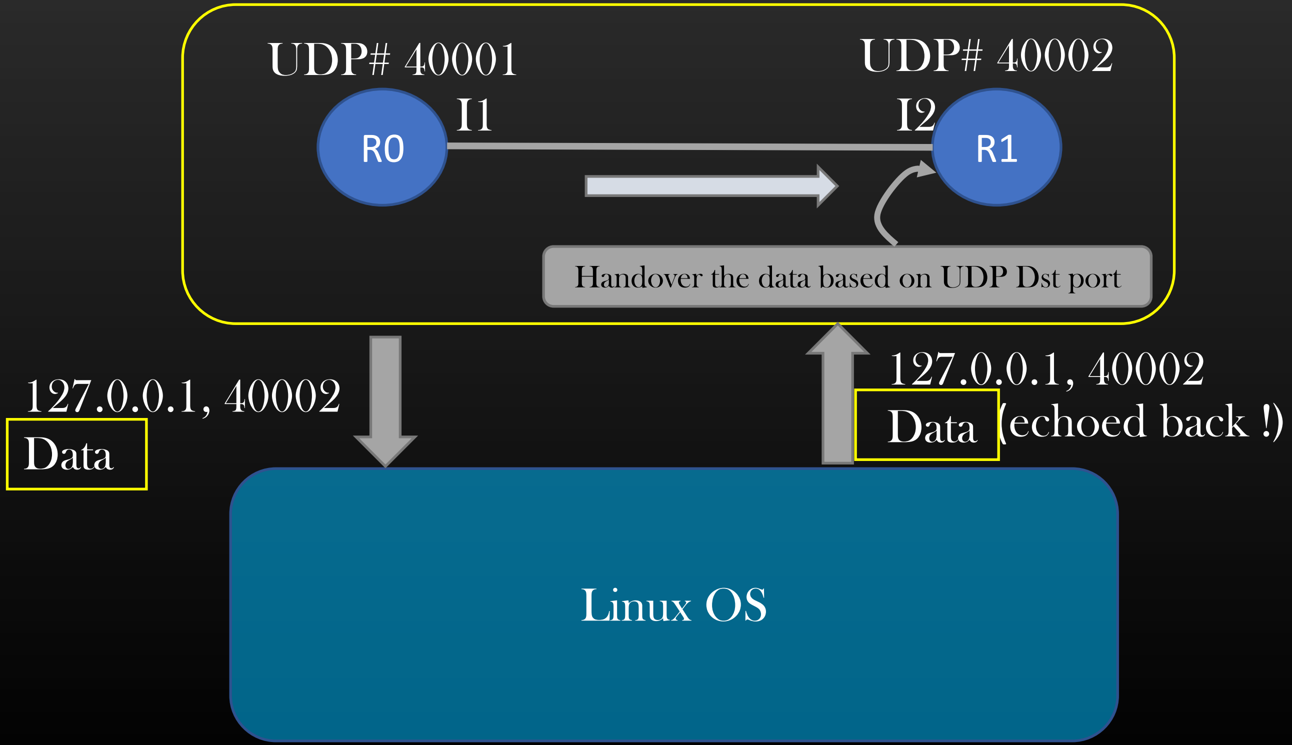
Packet Exchange Simulation Infra Setup

- Design Discussion to implement Communication between nodes over links

Logical View :



Actual View :

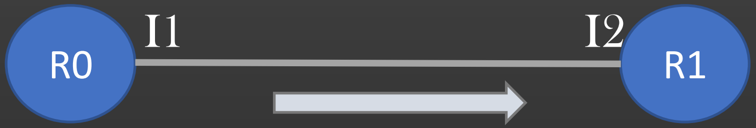


But, how R1 knows it has Received Data on Interface I2 !

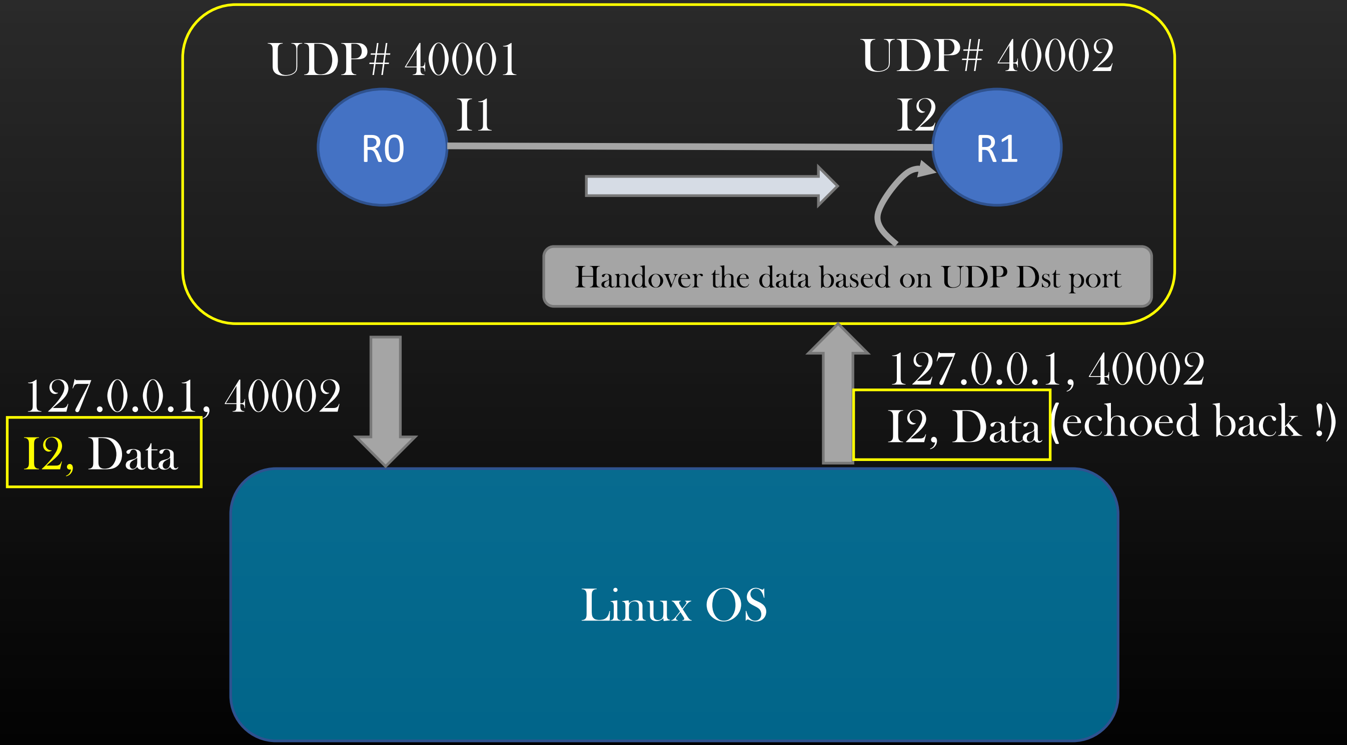
Packet Exchange Simulation Infra Setup

➢ Design Discussion to implement Communication between nodes over links

Logical View :



Actual View :



But how R1 Would know that pkt is recvd on local Interface I2 !!

Ans : R1 Cannot know

The sending node (R0) must Insert this additional info (called Auxiliary info) in pkt itself.

While receiving the data from OS, we shall segregate the aux info from actual pkt content

Time to see the code !!

Packet Exchange Simulation Infra Setup

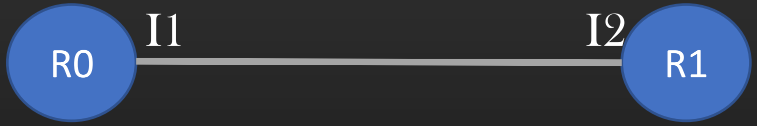
- Design Discussion to implement Communication between nodes over links

Steps :

1. Opening Sockets : Our application assigns unique UDP port number to each node of the topology
2. Opening Sockets : Our application opens a UDP socket for all port numbers,
   init_udp_socket (node_t *node)

3. Listening Sockets : Our application listen on all of UDP Sockets (select)
   network_start_pkt_receiver_thread (graph_t *topo)

4. Packet Transmission : Nodes Communicate by sending data to destination Node's port number with ip = 127.0.0.1
   send_pkt_out(char *pkt, unsigned int pkt_size, interface_t *oif)



5. Underlying OS echoes back all data back to application

6. Packet Reception : Based on Dst port number in the echoed data received by our application, our application handover the data to the destination node
   _network_start_pkt_receiver_thread(void *arg)

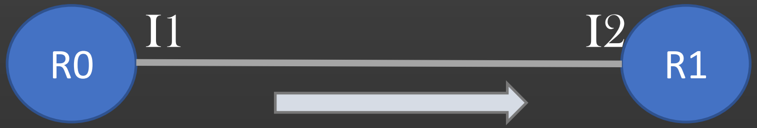
7. Auxiliary Information : Using Auxiliary information, Recipient interface name can be known
   _pkt_receive(node_t *receving_node, char *pkt_with_aux_data, unsigned int pkt_size)

8. Final Packet Reception : Actual packet (without auxiliary data) is received by the recipient node on an IIF.
   pkt_receive ((node_t *node, interface_t *interface, char *pkt, unsigned int pkt_size)

Time to See Code !

Packet Exchange Simulation Infra Setup

- Design Discussion to implement Communication between nodes over links



Let us Test the Virtual Communication !!

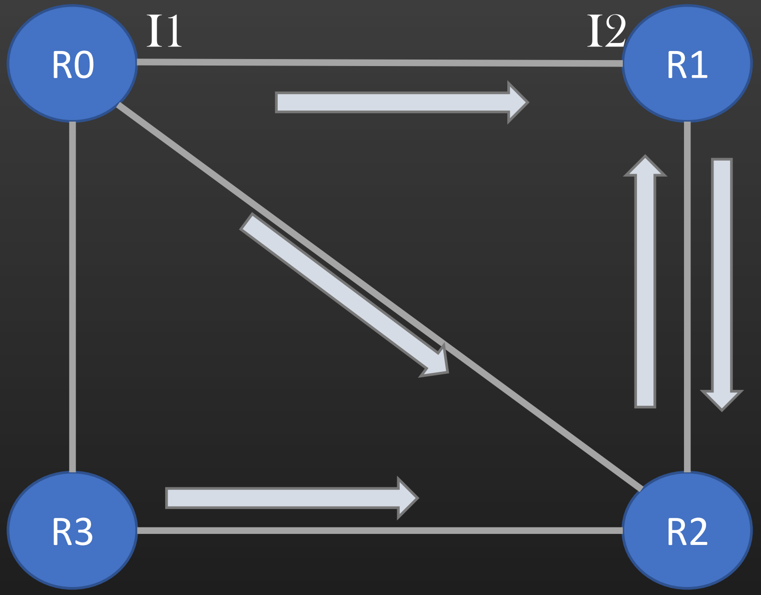
Packet Exchange Simulation Infra Setup

- Design Discussion to implement Communication between nodes over links

What we have achieved :

We are be able to
Successfully emulate the network
Topology Running on your machine

- Quickly build any topology
- Configure nodes and links with
  network properties
- Implement Routing Protocols (Or any networking
  Concepts , Research papers, Patents)
- Implement TCP/IP Stack, Ping , Traceroute and etc . . .
- Implement POC – "proof of concepts", Patent Demonstration
- Can be Used for other graph based problems – Dijkstra etc . .



PART B Agenda

6. Layer 2
   - ARP
   - L2 Mac Based Forwarding

7. Layer 3
   - Route Installation
   - L3 Routing

8. Application
   - Ping
   - Traceroute

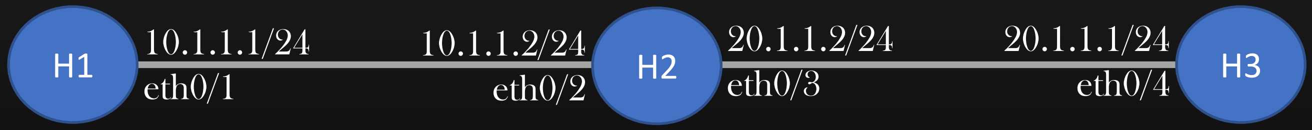
9. Supporting VLANs
   - Access Ports
   - Trunk Ports

- Be aware with standard Header formats
- Be good with pointers, how data is laid out in memory
- We shall be cooking up the packets :
  - Attaching and removing headers
  - Modifying headers
  - Copying packet contents
  - Any error -> Memory corruption !!
- Do all assignments

Implementing ARP

Goal

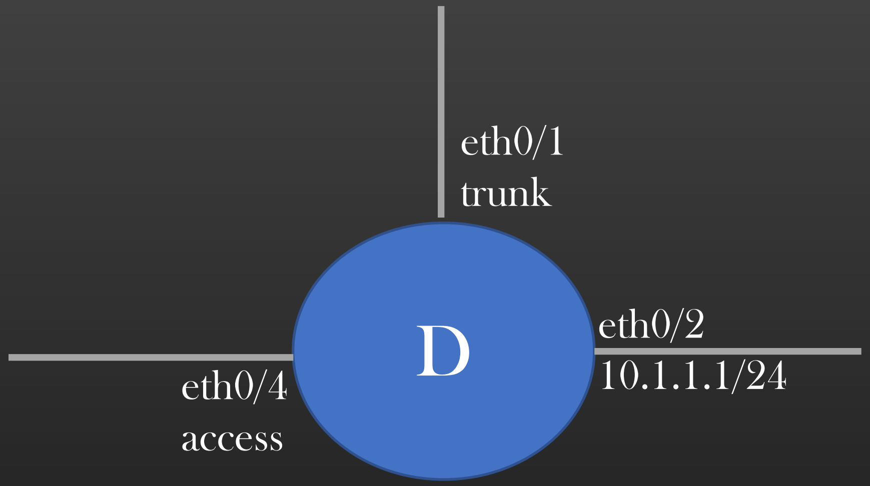
- Implement Layer 2 Address Resolution Protocol (ARP)
- First we shall implement ARP, and test it
- Then We shall implement L2 switching and see if our L2 switching switches ARP request and replies as expected,
- Let us take a simple, 3 Node topology for ARP



Before Going Forward, Let us understand preliminaries . . .

Interface Modes

- Any interface of a Routing Device (L3 Router Or L2 Switch) can operate in either of two modes at any given time :
  - L3 Mode
  - L2 Mode



Interface Operating modes

- IP address Configured
- ARP Resolution
- Process incoming packet only if
  Dst MAC in ethernet hdr = MAC(Intf)

- Operate either in ACCESS mode Or
  TRUNK Mode
- Vlan member ships
- Vlan tagging OR Un-tagging
- Accept or reject the incoming pkt based
  on Vlan Tags

- A Routing device by itself is not L2 switch or L3 Router, it is called L3 Router or L2 switch in respect of its interface configuration

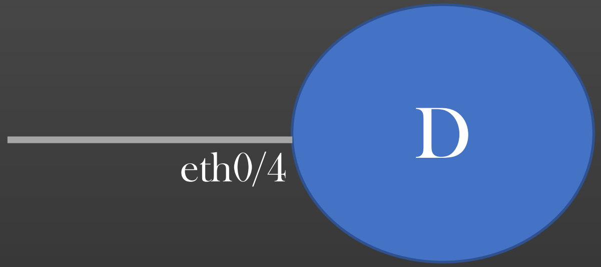
Defining Ethernet header

File : Layer2/layer2.h

- A packet must have ethernet hdr to be handled by Data link layer
- Time to define ethernet hdr structure as per the standards
- Assume you know the format of ethernet hdr , revise . . .

Assignment on Ethernet hdr and Interface Modes !

Packet Processing Criteria



- Whenever a Routing Device receives a packet on its local interface , the first thing it has to decide is whether it should process the incoming packet or reject it right away before packet could even enter into TCP/IP Stack.

- Acceptance or Rejection of the packet depends on many factors including but not limited to :

  Interface Operating Modes
  Interface Configuration
  Packet Contents

- If the packet is Accepted, Routing device handover the packet to TCP/IP stack , and the ingress journey of the packet commences

Packet Processing Criteria

API : Layer2/layer2.h

Now, we are in a position to write an API which decides whether routing device should accept Or reject the incoming packet arrived on an interface operating in L3 mode:

```
static inline bool_t
l2_frame_recv_qualify_on_interface(interface_t *interface,
                                   ethernet_hdr_t *ethernet_hdr);
```

Returns TRUE, if packet should be accepted for further processing
Returns FALSE, if packet should be rejected

Pseudocode :

- IF interface is not working in L3 mode -> Return FALSE
- IF interface is operating in L3 mode and dst mac in ethernet hdr == IF_MAC(interface) -> Return TRUE
- IF interface is operating in L3 mode and dst mac in ethernet hdr is BROADCAST MAC -> Return TRUE
- Return FALSE in any other case

Packet Processing Criteria - Enhanced

Pseudocode :

```
static inline bool_t
l2_frame_recv_qualify_on_interface(interface_t *interface,
                                   ethernet_hdr_t *ethernet_hdr);
```

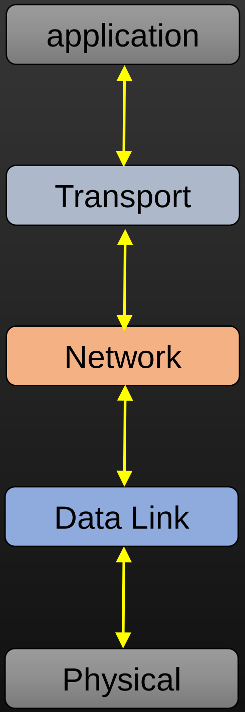
- IF interface is neither working in L3 mode nor in L2 mode -> Return FALSE
- IF interface is operating in ACCESS (or TRUNK) mode -> Return TRUE (later)
- IF interface is operating in L3 mode and dst mac in ethernet hdr == IF_MAC(interface) -> Return TRUE
- IF interface is operating in L3 mode and dst mac in ethernet hdr is BROADCAST MAC -> Return TRUE
- Return FALSE in any other case

Packet Buffer Management

- Before We write out first line of code to send and receive packets/frames between our virtual routing devices, we need to get familiar with the *packet buffers*

  – a Memory used to store the pkt/frame generated by the source layer of TCP/IP Stack

Packet Buffer Management -> Source Layers of TCP/IP Stack



The Layers of the TCP/IP Stack which generates the data to be processed by the TCP/IP Stack are called as Source Layers

"Source" – source of data

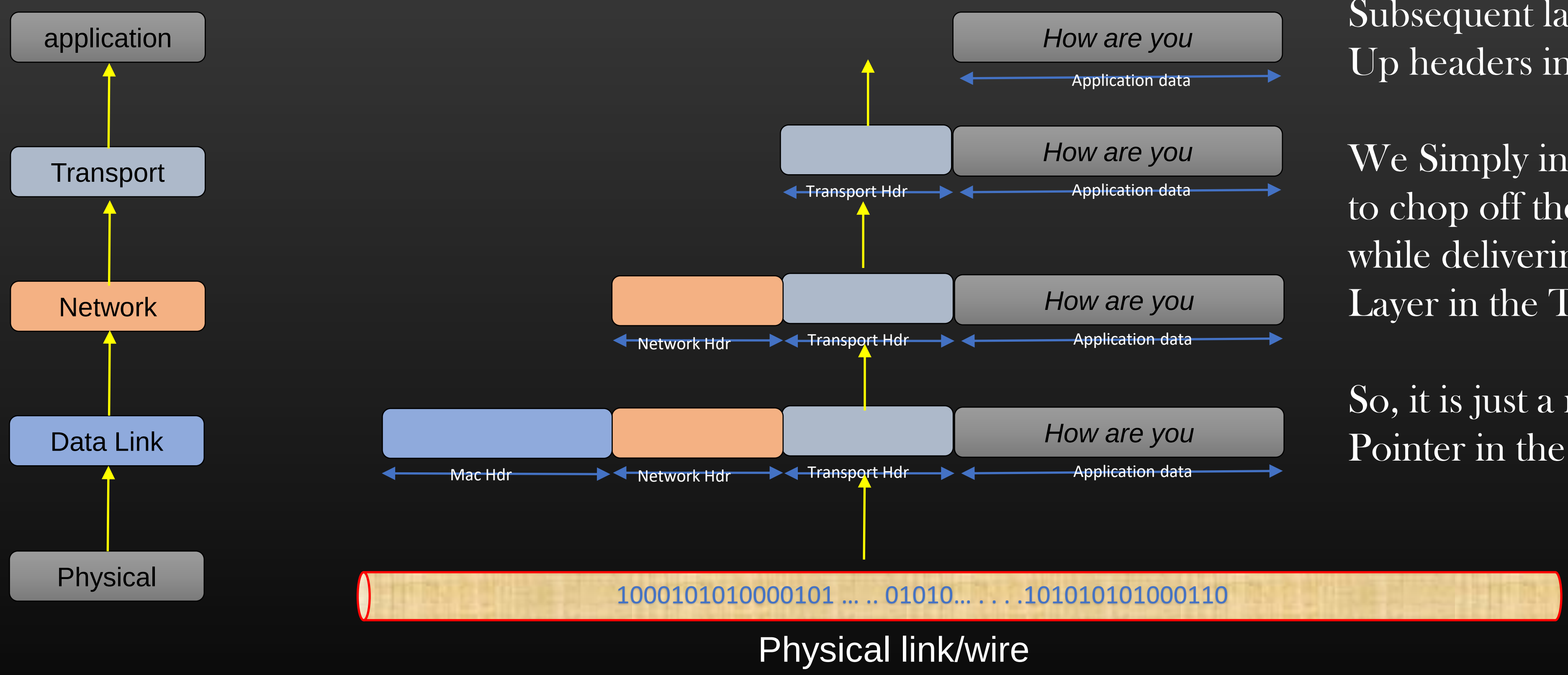
Application and Physical Layer are Source Layers of TCP/IP Stack !

Physical Layer : Converts the electrical signals on wire into Data, and feed into TCP/IP Stack From BOTTOM

Application Layer : Software Program which generates the data and feeds it into TCP/IP Stack From TOP

Packet Buffer Management -> Ingress Journey

Ingress Journey of the Packet in the TCP/IP Stack



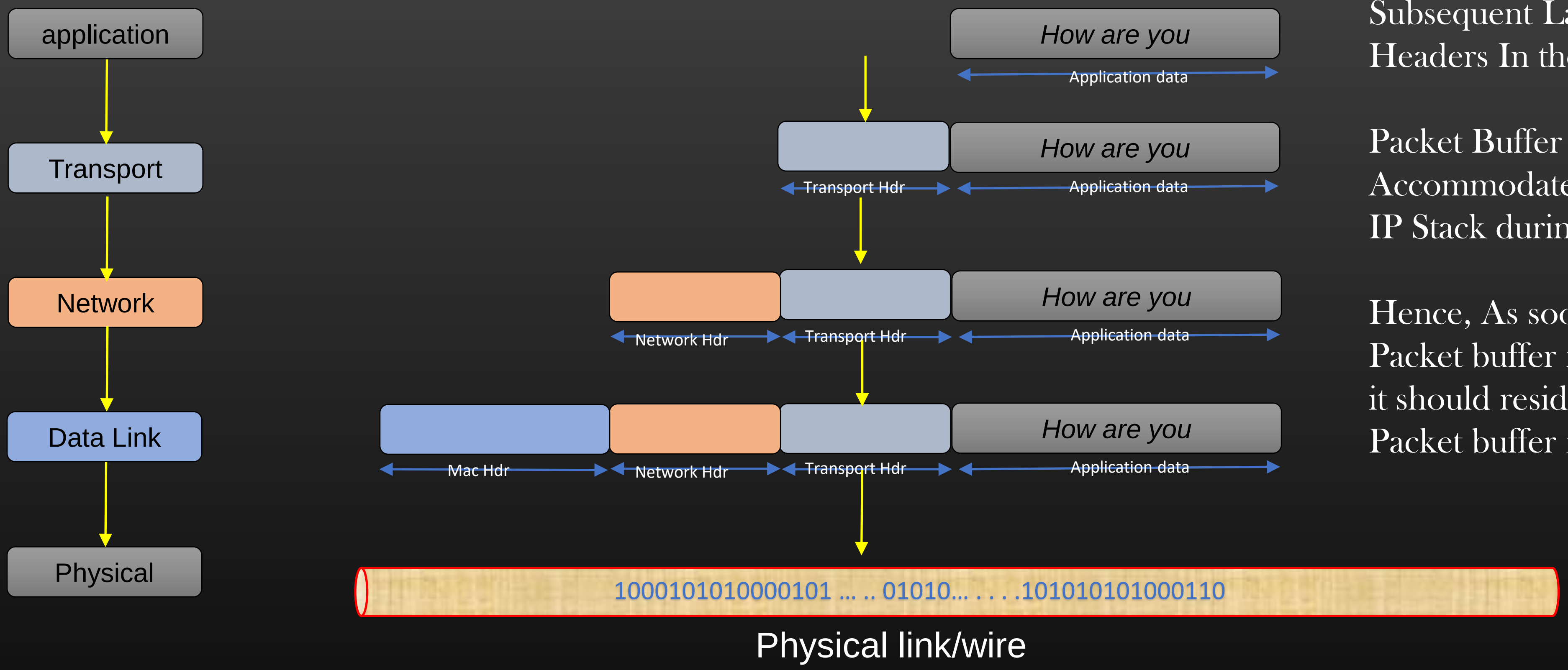
As Packet enters into TCP/IP Stack, Subsequent layers sees only the follow Up headers in the packet

We Simply increment the "pkt" pointer to chop off the header from the packet while delivering it to the next higher Layer in the TCP/IP Stack

So, it is just a matter of incrementing a pkt Pointer in the packet buffer

Packet Buffer Management -> Egress Journey

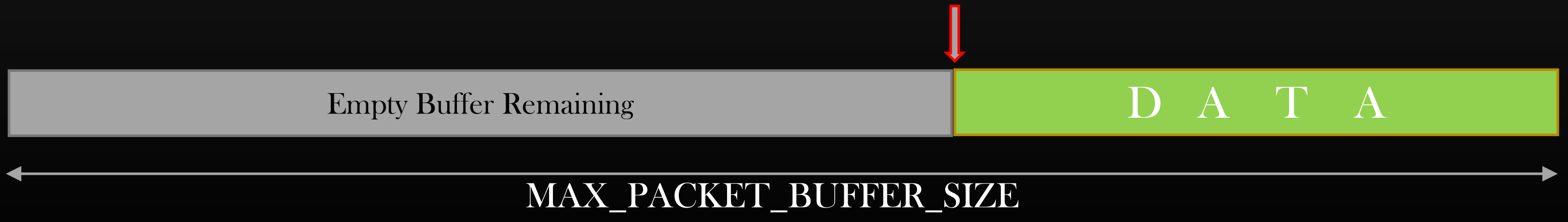
Egress Journey of the Packet in the TCP/IP Stack



But Egress Journey of the packet Requires Subsequent Layers to attach their own Headers In the front of the packet

Packet Buffer must have enough room to Accommodate headers of all layers of TCP/IP Stack during its course from top to bottom

Hence, As soon as data is created and stored in Packet buffer memory by the Source Layer, it should reside on the Right boundary of the Packet buffer memory



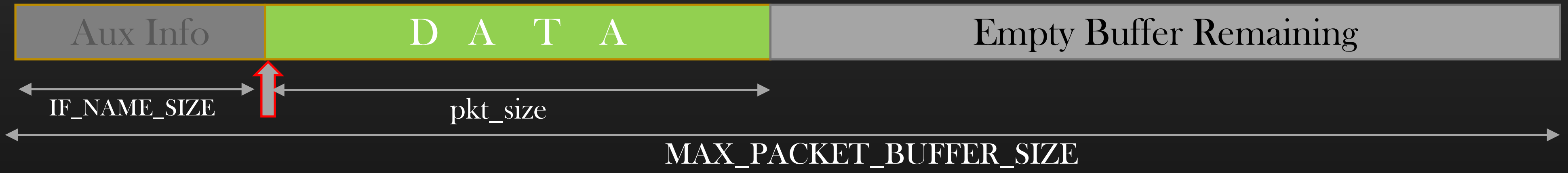
Packet Buffer Management -> Right shift the pkt buffer

- Two Packet buffers :

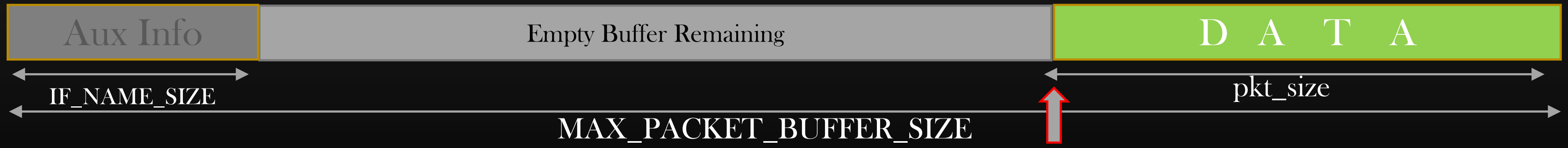
  static char recv_buffer [MAX_PACKET_BUFFER_SIZE];
  static char send_buffer [MAX_PACKET_BUFFER_SIZE];

- Whenever a Node receive a frame on its local interface from nbr node, The API *pkt_receive (. . .)* is invoked.

- Following is the snapshot of the recv_buffer when *pkt_receieve(. . .)* API is invoked

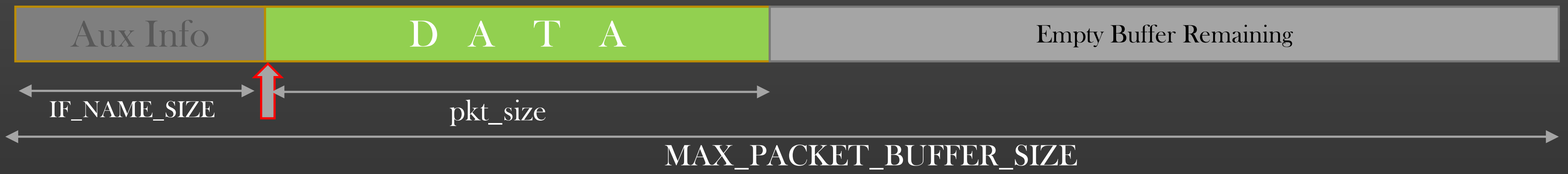


☜ This is what we receive

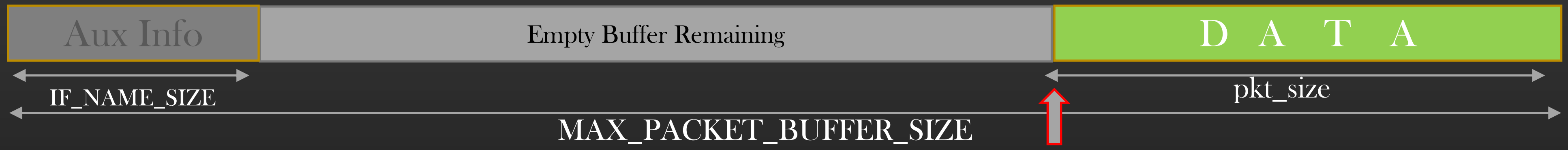


☜ "Right Shift" packet buffer, this is what should be given to TCP/IP Stack

Packet Buffer Management -> API to Right shift the pkt buffer



☜ This is what we receive



☜ "Right Shift" packet buffer

The API to perform "right shift" of data on a packet buffer shall be :

net.c/.h

```
char *
pkt_buffer_shift_right(char *pkt, unsigned int pkt_size,
                       unsigned int total_buffer_size);
```

Returns a pointer to start of data in the "right shifted" packet buffer

total_buffer_size = MAX_PACKET_BUFFER_SIZE - IF_NAME_SIZE

Warning !

- You have to be extremely careful while dealing with packet buffers, manipulating packet contents, modifying the packet headers etc

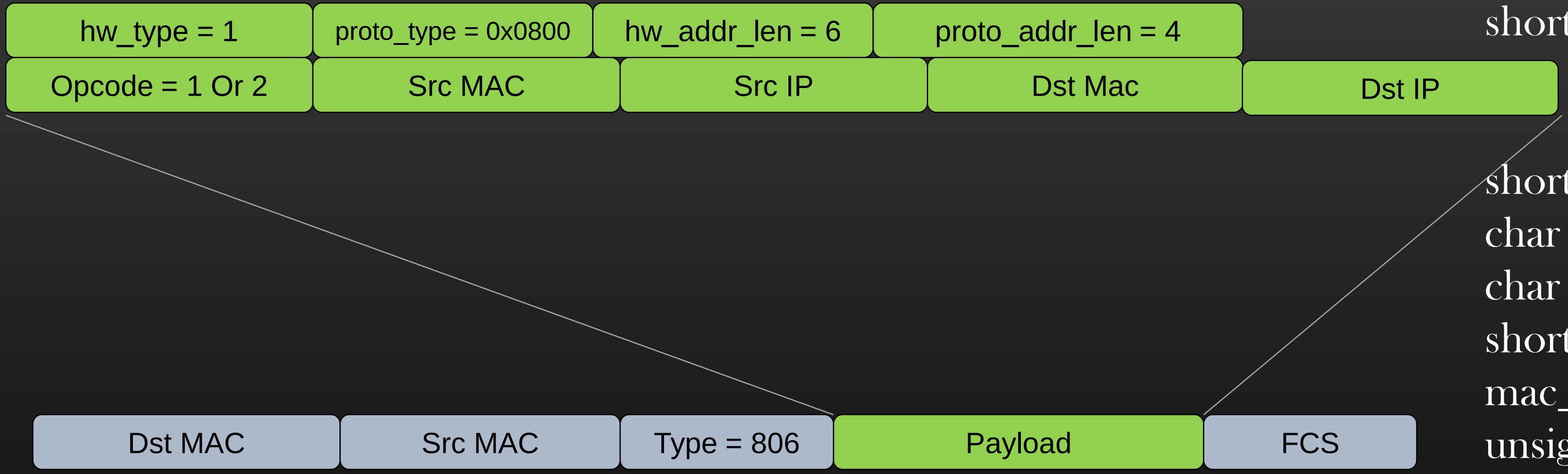
- One mistake -> Memory corruption -> difficult to debug

- This project is full of packet manipulation

- Preventive Measures :
  - Use Debuggers such as gdb
  - Use as many printfs as you want
  - Be sure what you are doing !

Implementing ARP -> Data Structures

- ARP standard Headers to be defined in Layer2/layer2.h
  - ARP Broadcast Request
  - ARP Reply



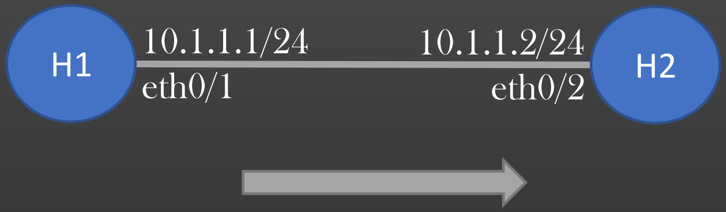
```
typedef struct arp_hdr_{

    short hw_type;            /*1 for ethernet cable*/
    short proto_type;         /*0x0800 for IPV4*/
    char hw_addr_len;         /*6 for MAC*/
    char proto_addr_len;      /*4 for IPV4*/
    short op_code;            /*req or reply*/
    mac_add_t src_mac;        /*MAC of OIF interface*/
    unsigned int src_ip;      /*IP of OIF*/
    mac_add_t dst_mac;        /*?*/
    unsigned int dst_ip;      /*IP for which ARP is being resolved*/
} arp_hdr_t;
```

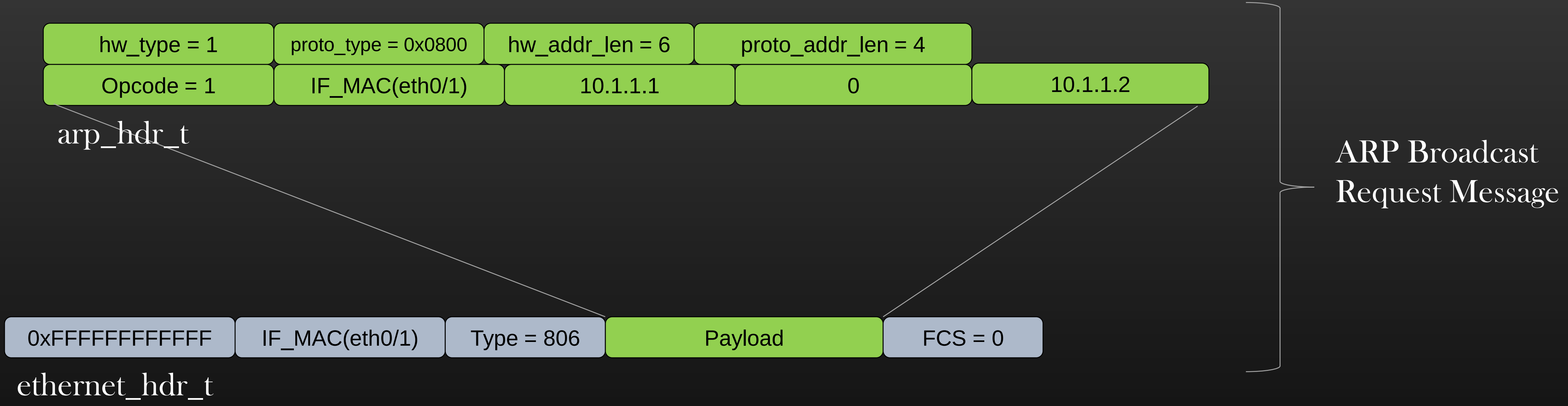
- All standard Msg types to be defined in tcpconst.h file

Implementing ARP -> Data Structures

- Example :

  H1 resolving ARP for 10.1.1.2
  On eth0/1

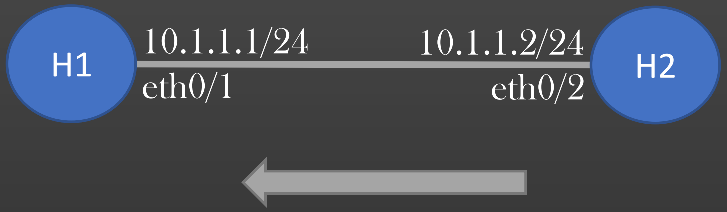


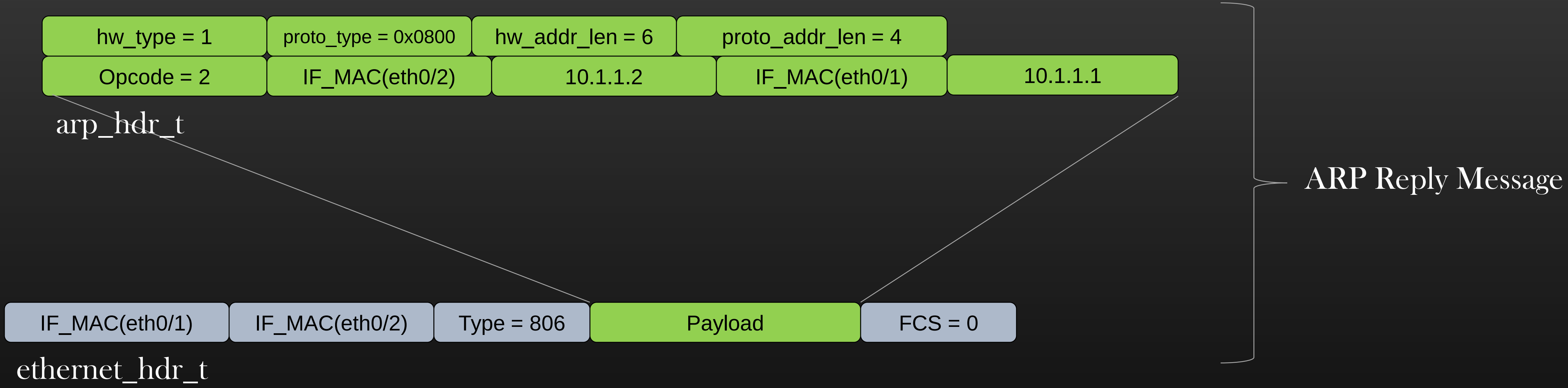


Implementing ARP -> Data Structures

- Example :

  H2 replying with ARP reply

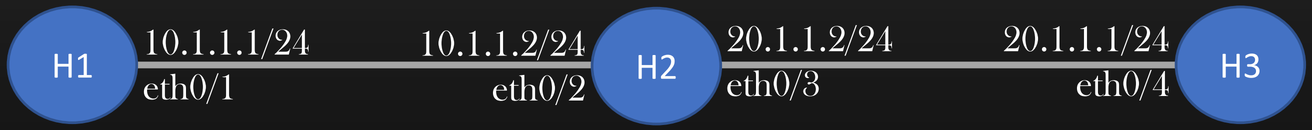




Implementing ARP -> ARP Table -> Data Structures

- ARP is used by Host Or L3 Routers to resolve MAC for known IP address
- Host/L3 Routers maintain a table called ARP table which contain ARP entries

IP address (key)	MAC Address	OIF



H1 :	10.1.1.2	IF_MAC(eth0/2)	eth0/1
H2 :	10.1.1.1	IF_MAC(eth0/1)	eth0/2
	20.1.1.1	IF_MAC(eth0/4)	eth0/3
H3 :	20.1.1.2	IF_MAC(eth0/3)	eth0/4

Layer2/layer2.h

```
/* ARP Table Data Structures */

typedef struct arp_table_{

    glthread_t arp_entries;
} arp_table_t;


typedef struct arp_entry_ arp_entry_t;
struct arp_entry_{

    ip_add_t ip_addr;   /*key*/
    mac_add_t mac_addr;
    char oif_name[IF_NAME_SIZE];
    glthread_t arp_glue;
};
```

Implementing ARP -> ARP Table -> APIs

ARP Table APIs : Layer2/layer2.h/.c

Initialize the ARP table, should be called when a node is created during topology creation i.e. from *init_node_nw_prop(. . .)*

```
void init_arp_table (arp_table_t **arp_table);
```

CRUD Operations on ARP table :

C	bool_t arp_table_entry_add (arp_table_t *arp_table, arp_entry_t *arp_entry);
R	arp_entry_t * arp_table_lookup (arp_table_t *arp_table, char *ip_addr);
U	void arp_table_update_from_arp_reply (arp_table_t *arp_table, arp_hdr_t *arp_hdr, interface_t *iif
D	void delete_arp_table_entry (arp_table_t *arp_table, char *ip_addr);

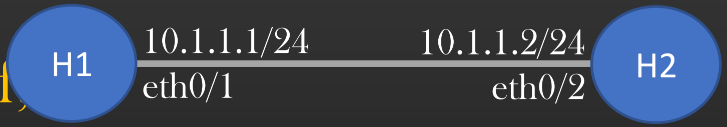
Dump API :

```
void dump_arp_table (arp_table_t *arp_table);
```

Implementing ARP -> API to trigger ARP Resolution

An API which triggers ARP resolution is :

```
void
send_arp_broadcast_request (node_t *node,
                                          interface_t *oif
                                      char *ip_addr);
```



All ARP related APIs shall go in Layer2/layer2.h/.c

Implementing ARP -> CLIs

CLI to manually trigger ARP for testing :

```
run node <node-name> resolve-arp <ip-address>
```

- nwcli.c
- Backend handler :

```
static int
arp_handler (param_t *param, ser_buff_t *tlv_buf,
                 op_mode enable_or_disable);
```

CLI to dump ARP table of a node

```
show node <node-name> arp
```

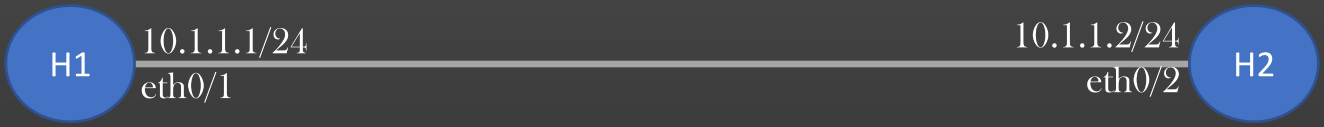
- nwcli.c
- Backend handler :

```
static int
show_arp_handler (param_t *param, ser_buff_t *tlv_buf,
               op_mode enable_or_disable);
```

Next, we shall discuss the ARP Cycle …

Implementing ARP -> ARP Cycle

layer2.h/layer2.c



```
void
send_arp_broadcast_request (node_t *node,
                            interface_t *oif,
                            char *ip_addr);
```

ARP Broadcast

```
static void
process_arp_broadcast_request (node_t *node,
                                                              in
                                                ethernet_hdr_t *ether
```

```
static void
send_arp_reply_msg (ethernet_hdr_t *ethernet_hdr_in,
                                   interface_t *oif)
```

ARP Reply

```
static void
process_arp_reply_msg(node_t *node, interface_t *iif,
                      ethernet_hdr_t *ethernet_hdr);
```

<< Made an entry into ARP table !!

Implementing ARP -> ARP Msg Processing by the Routing Device

```
layer2_frame_recv (. . .)   /*Pre-Entry point of frame into the TCP/IP Stack,  but not yet entered !*/
                                        (Standing outside the door, knocking ... !)

                                   → l2_frame_recv_qualify_on_interface(. . .)  /*Check if Frame qualifies to be processed by TCP/I
                                        (House owner asking you who are you !!)

                                  (Now, Finally you are permitted inside the house)
                                   if pkt arrived on L3 interface
                                        process the pkt as per ethernet_hdr->type value
                                                 if 806 , process_arp_broadcast_request(. . .) Or process_arp_reply_msg(. . .)
                                                 if 0x0800, promote the pkt to Layer3 (Later . . . )

                                  (Now, Finally you are permitted inside the house)
                                  if pkt arrived on L2 interface
                                     → l2_switch_recv_frame (. . .)               /*Feed it to L2 switch forwarding Algorithm, Later.
```

Implementing ARP -> Summary

- We shall come back to ARP again when we shall be implementing L3 routing

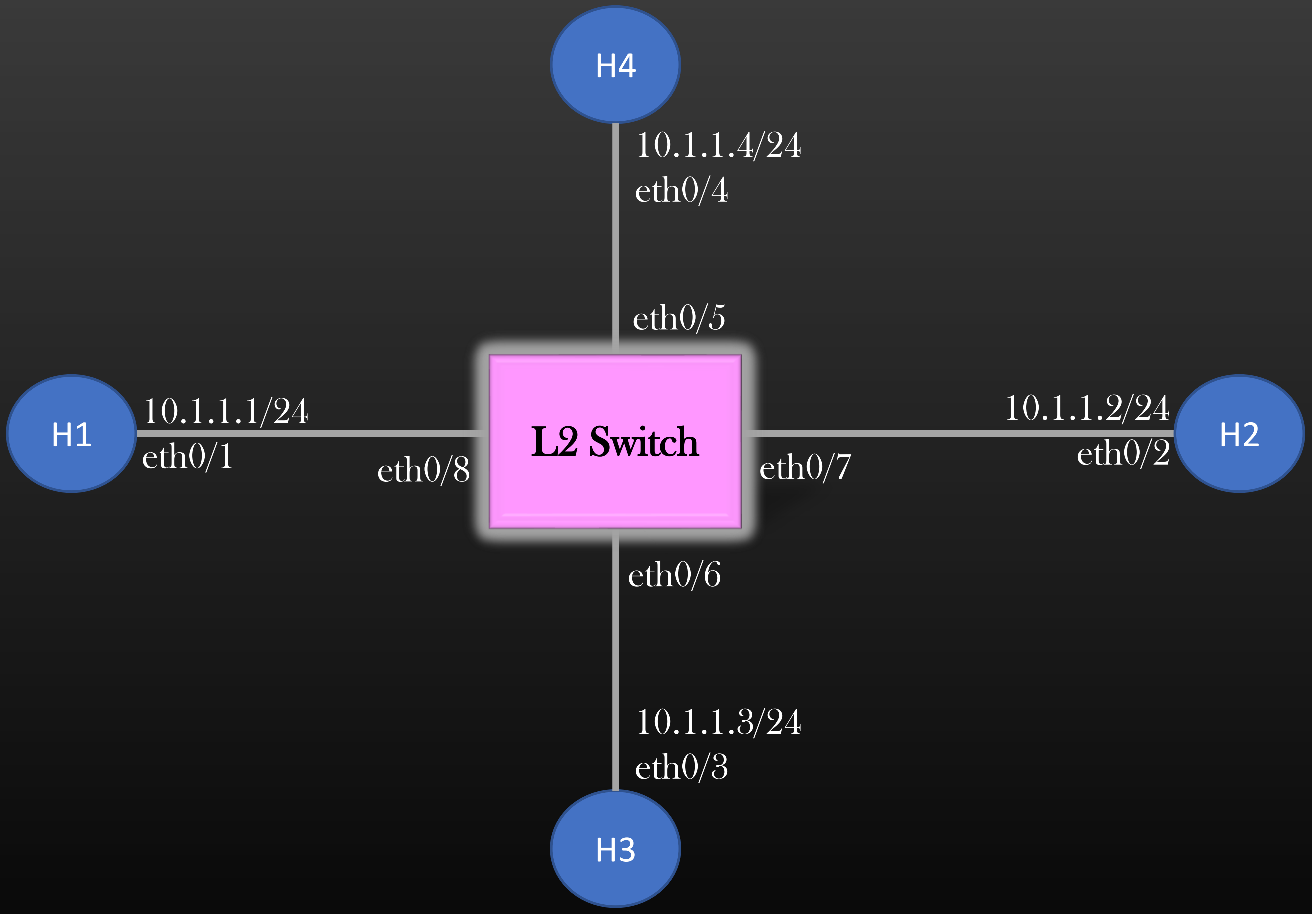
- ARP is resolved whenever the L3 router tries to :
  - Forward the packet to next router
  - Deliver the packet to host machine present in a local subnet of the router

- Next :
  - Let us implement L2 switching and MAC based forwarding

L2 Switching Implementation

Goal

- Implement Layer 2 Switching functionality i.e. Mac Based Forwarding



- L2 Switches do not have IP-address configured On its interfaces
- All hosts are in same subnet
- L2 Switches inspect on ethernet hdr of any frame passing through it
- L2 switches maintains mac table
- All ports of L2 switches operate in *access* mode to begin with
- Later We shall Implement Trunk mode also

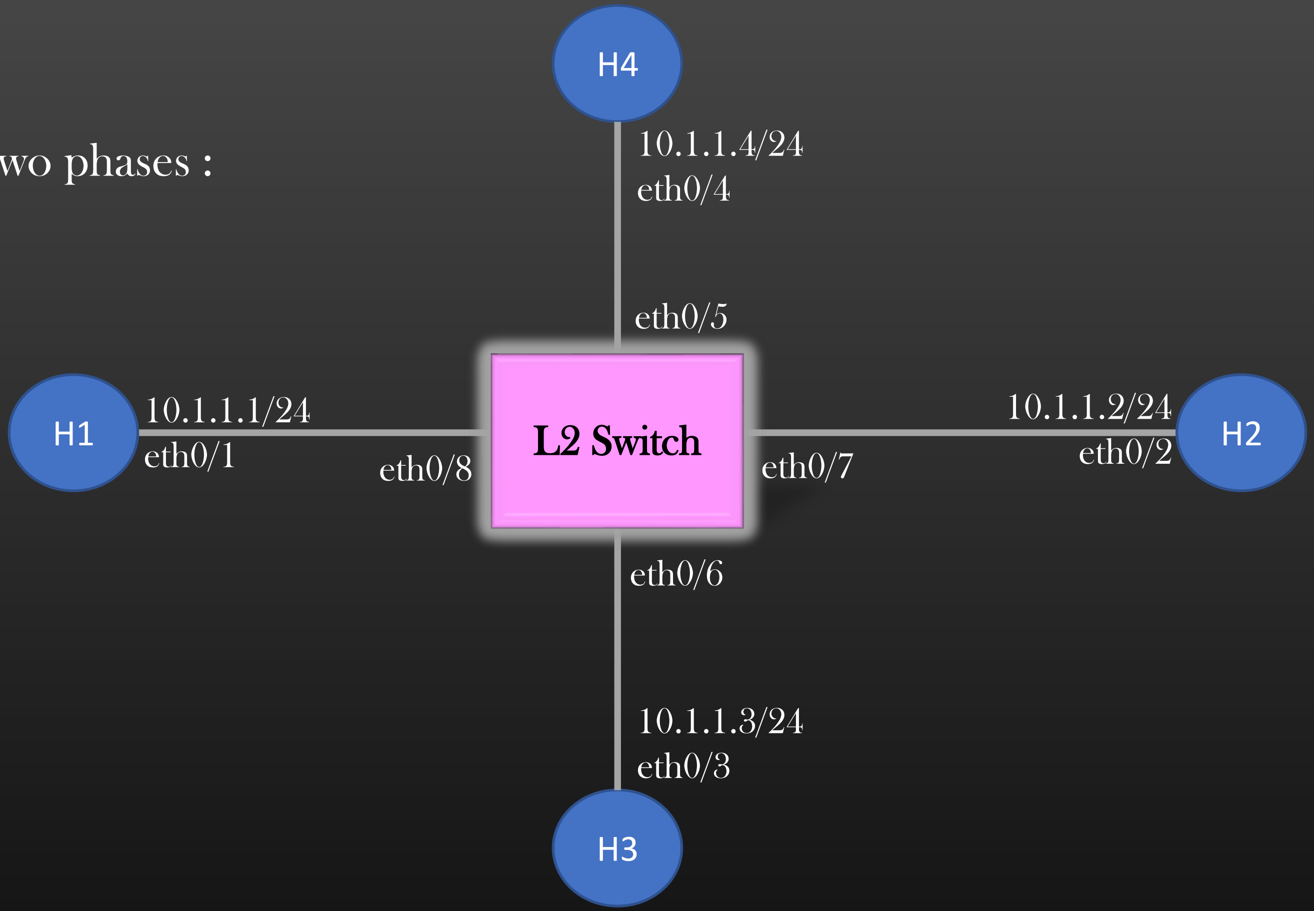
L2 Switching Implementation

L2 switching Topology Setup

We shall implement L2 Switch based functionality in Two phases :

Phase 1 : Setting up the Data structures

Phase 2 : L2 Switch Mac Learning and Forwarding



L2 Switching Implementation

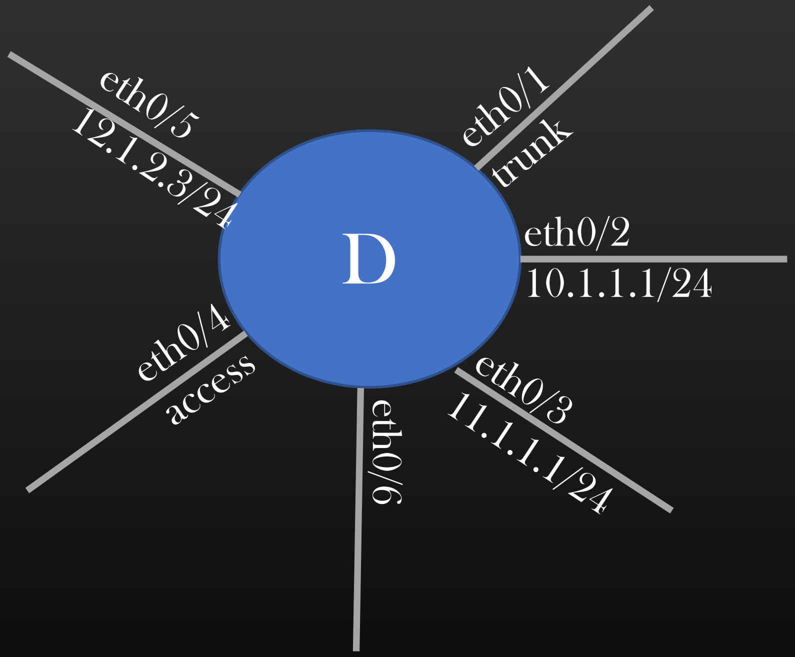
Phase 1: Setting up the Data structures

Pre-requisite : Refresh L2 Switching knowledge, understand MAC Learning and Forwarding

L2 Switching Implementation -> Switch Vs Router !

- If some interfaces of routing device are operating in L3 mode (ip-address configured), and some are in ACCESS/TRUNK mode, then node is behaving as Router as well as L2switch

- A Node by itself is not L2 switch or Router, it is called Router or L2 switch in respect of its interface configuration



This Device is L3 router wrt to interfaces eth0/2, eth0/3, eth0/5

This Device is L2 Switch wrt to interfaces eth0/1, eth0/4

Interface eth0/6 is not operational

Devices with dual functionality are called rbridges

L2 Switching Implementation -> Phase 1 -> Interface Modes

Interface Operating modes

L3
IP Address

L2
ACESS Or TRUNK

net.h

```
typedef enum{

    ACCESS,
    TRUNK,
    L2_MODE_UNKNOWN
} intf_l2_mode_t;


typedef struct intf_nw_props_ {
  . . .
  intf_l2_mode_t  intf_l2_mode;
  . . .
} intf_nw_props_t;
```

New APIs : Layer2/layer2.h, layer2.c

```
void
node_set_intf_l2_mode (node_t *node, char *intf_name,
                                        intf_l2_mode_t
intf_l2_mode);
```

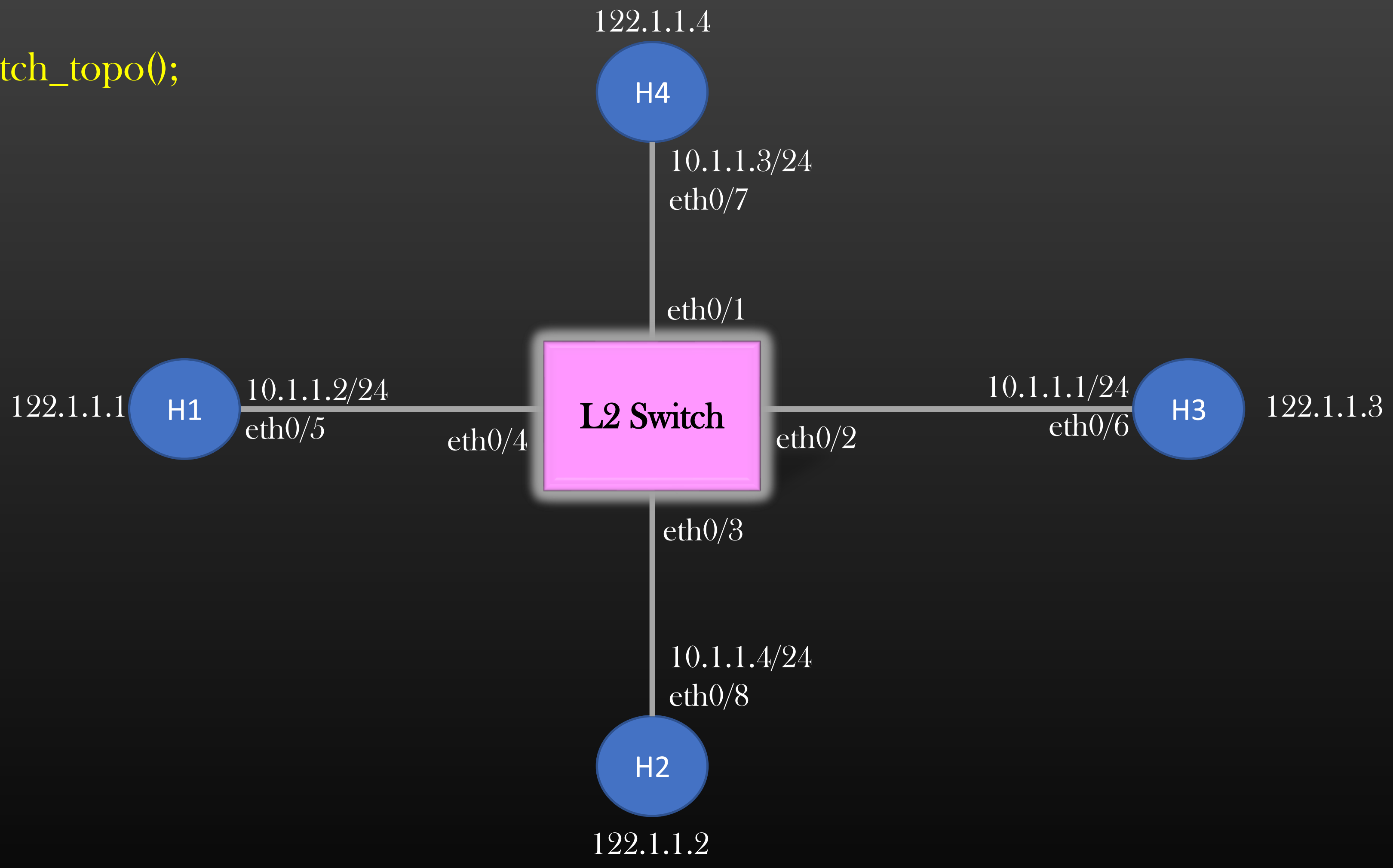
- Interface cannot operate in L3 and L2 mode at the same time
- Interface cannot operate in ACCESS and TRUNK mode at the same time
- Write Robust API to handle all scenarios
- Enhance existing *show topology* command to show interface mode of operation

- Config CLI : config node <node-name> interface < intf-name > l2mode < access|trunk >

L2 Switching Implementation

L2 switching Topology Setup

```
graph_t *
build_simple_l2_switch_topo();
```



L2 Switching Implementation -> Phase 1 -> Setting Up MAC tables

- L2 switch Devices have mac tables
- Just like we added ARP table to each Router/Host, we need to add mac table to each L2 Switch

```
typedef struct node_nw_prop_{
   . . .
   arp_table_t  *arp_table;
   mac_table_t *mac_table;
   . . .
} node_nw_prop_t;
```

Layer2/l2switch.c

```
typedef struct mac_table_entry_{

    mac_add_t mac;                          /*key*/
    char oif_name [IF_NAME_SIZE];
    glthread_t mac_entry_glue;   /*for linked-list insertion*/
} mac_table_entry_t;


typedef struct mac_table_{

    glthread_t mac_entries;
} mac_table_t;
```

L2 Switching Implementation -> Phase 1 -> Setting Up MAC tables -> Mac Table APIs

CRUD APIs on MAC Table :
Layer2/l2switch.c

CU - bool_t
```
mac_table_entry_add (mac_table_t *mac_table,
                     mac_table_entry_t *mac_table_entry);
```

R - mac_table_entry_t *
```
mac_table_lookup (mac_table_t *mac_table, char *mac);
```

D - void
```
delete_mac_table_entry (mac_table_t *mac_table, char *mac);
```

Initialize :

```
void
init_mac_table(mac_table_t **mac_table);
```

CLIs :

show node <node-name> mac
File : nwcli.c, cmdcodes.h

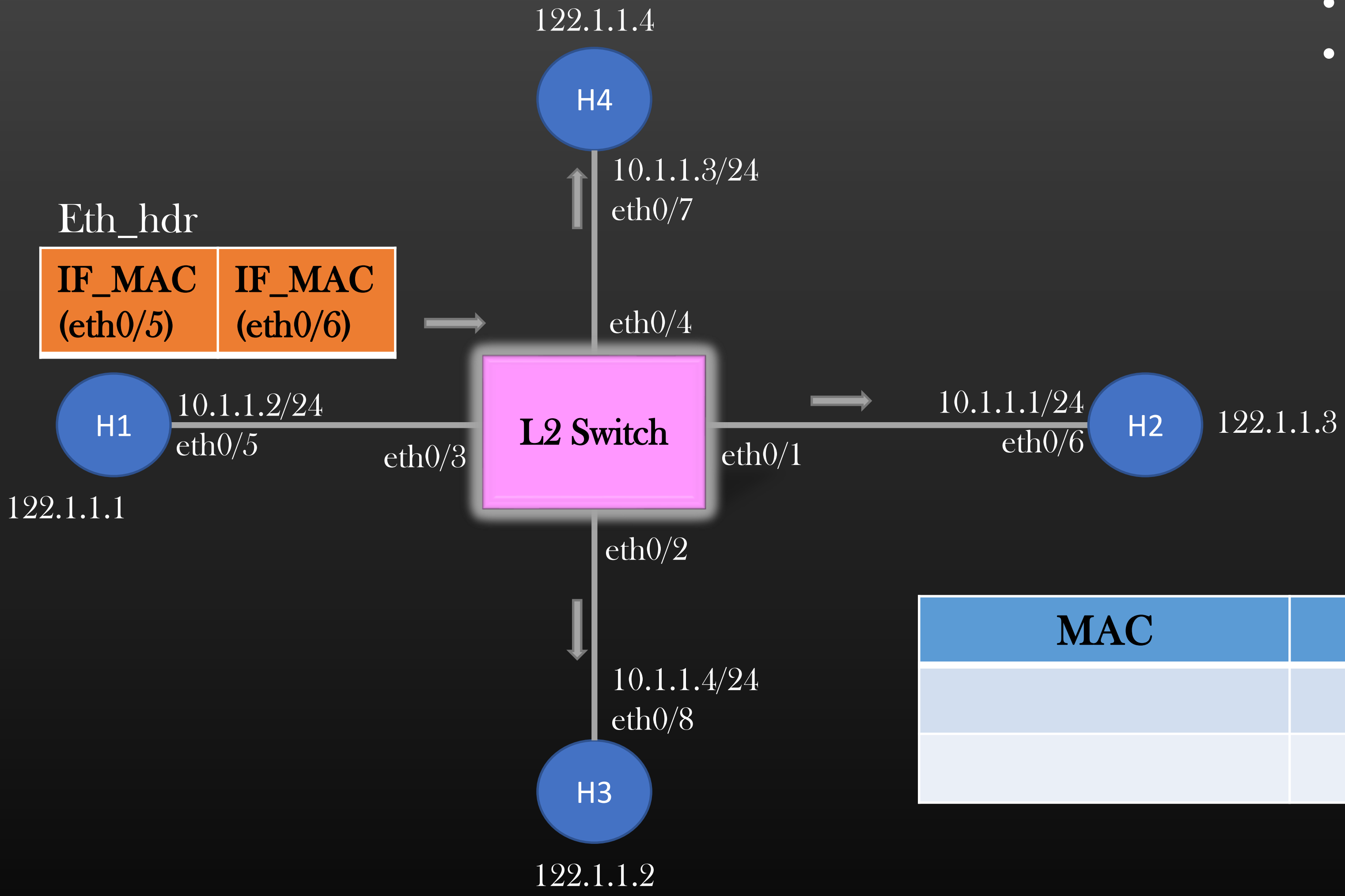
Backend Handler :
```
static int
show_mac_handler(param_t *param, ser_buff_t *tlv_buf,
                 op_mode enable_or_disable);
```

Dumping API:
Layer2/l2switch.c

```
void
dump_mac_table(mac_table_t *mac_table);
```

Phase 2 : L2 Switch Mac Learning and Forwarding

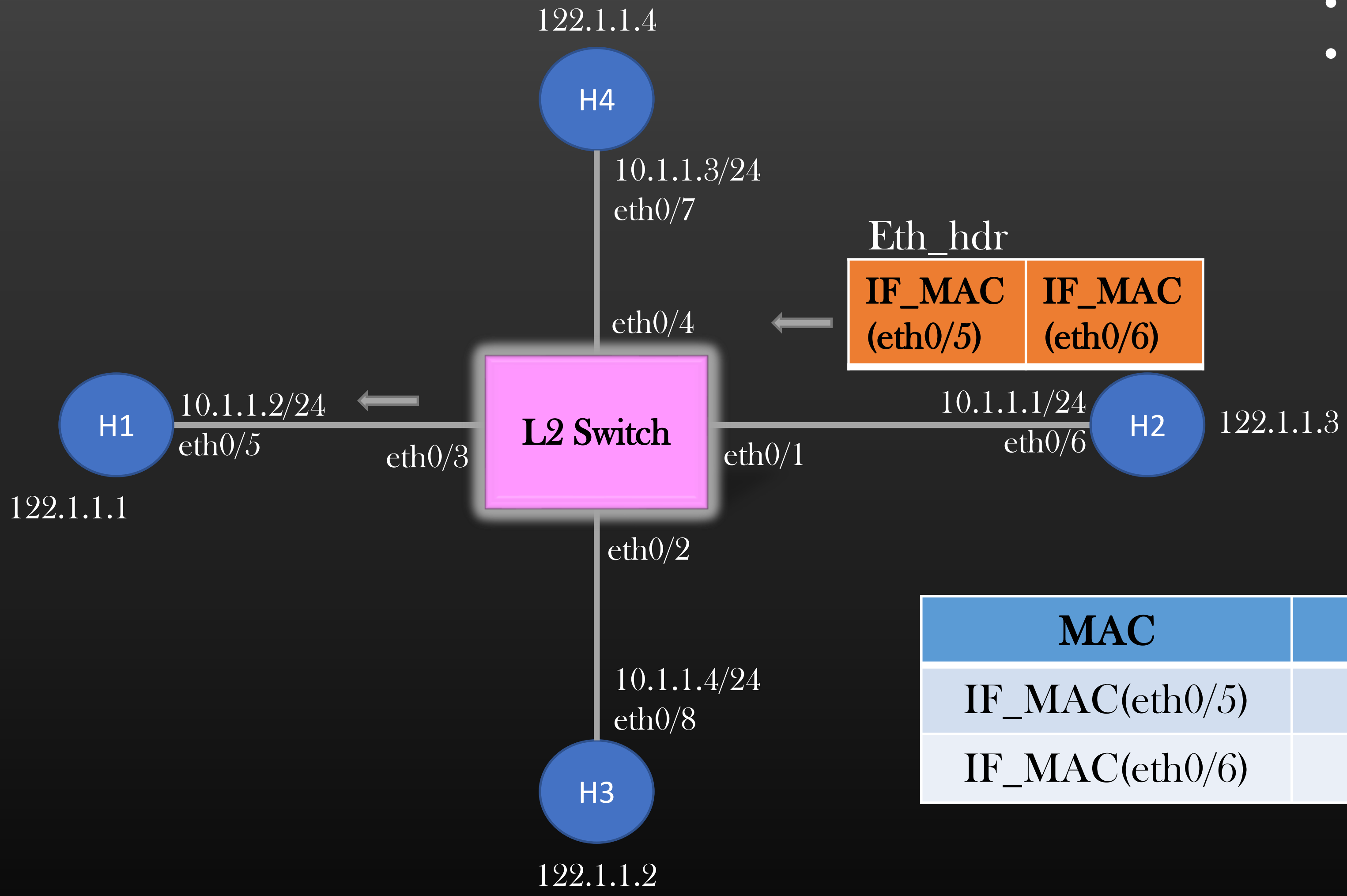
- Learn using Src MAC
- Forwards Using Dst MAC



MAC	OIF

Phase 2 : L2 Switch Mac Learning and Forwarding

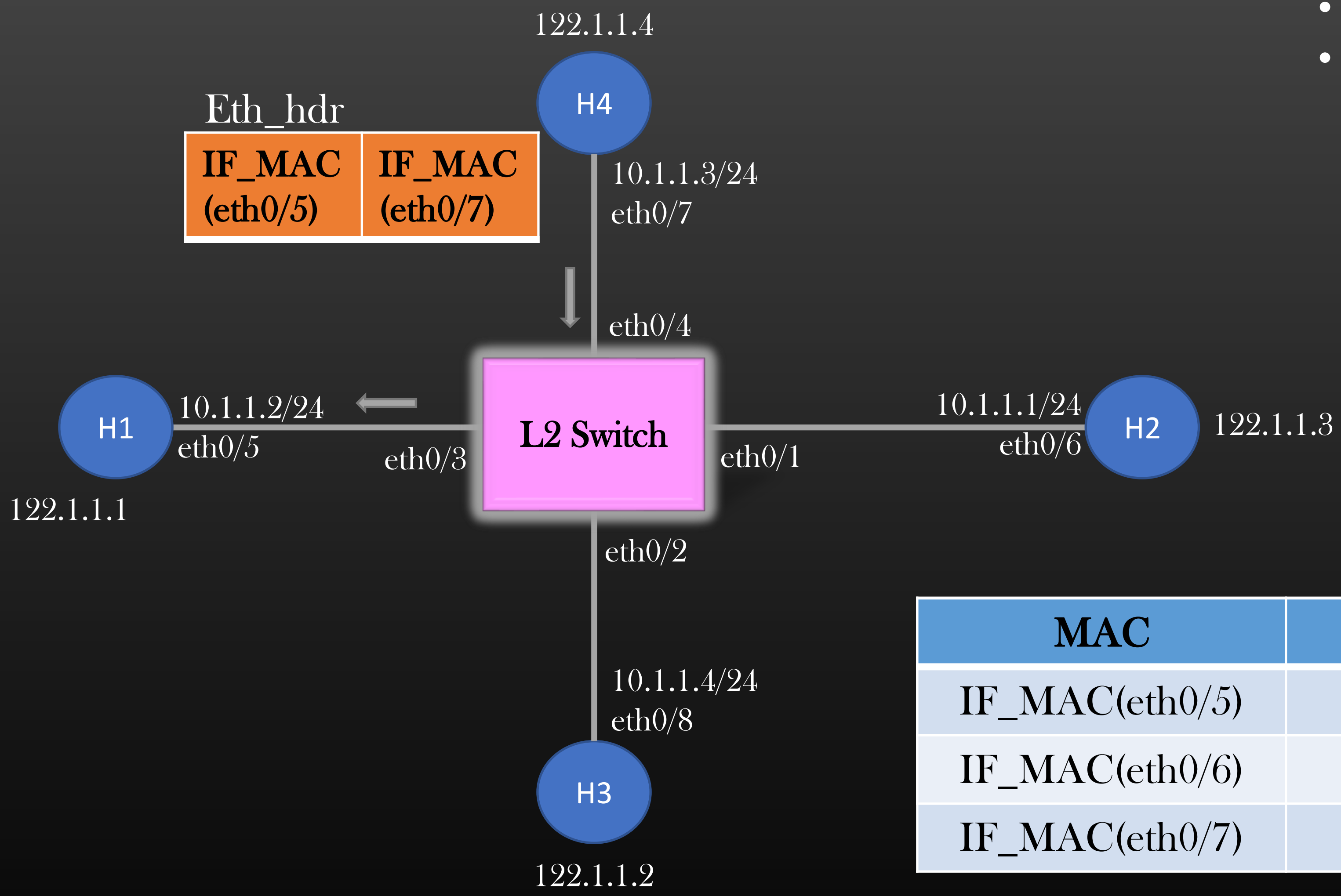
- Learn using Src MAC
- Forwards Using Dst MAC



MAC	OIF
IF_MAC(eth0/5)	eth0/3
IF_MAC(eth0/6)	eth0/1

Phase 2 : L2 Switch Mac Learning and Forwarding

- Learn using Src MAC
- Forwards Using Dst MAC



MAC	OIF
IF_MAC(eth0/5)	eth0/3
IF_MAC(eth0/6)	eth0/1
IF_MAC(eth0/7)	eth0/4

If Dst MAC is oxFFFFFFFF, L2 switch broadcast
Eg : ARP Broadcast Request Msgs

L2 Switching Implementation -> Phase 2 -> L2 Switch Mac Learning and Forwarding

```
layer2_frame_recv (. . .)   /*Pre-Entry point of frame into the TCP/IP Stack,  but not yet entered !*/
                                        (Standing outside the door, knocking ... !)


                                   → l2_frame_recv_qualify_on_interface(. . .)  /*Check if Frame qualifies to be processed by TCP/I
                                        (House owner asking you who are you !!)


                                  (Now, Finally you are permitted inside the house)
                                   if pkt arrived on L3 interface
                                        process the pkt as per ethernet_hdr->type value
                                                 if 806 , process_arp_broadcast_request(. . .) Or process_arp_reply_msg(. . .)
                                                 if 0x0800, promote the pkt to Layer3 (Later . . . )


                                  (Now, Finally you are permitted inside the house)
                                  if pkt arrived on L2 interface
                                     → l2_switch_recv_frame (. . .)                /*Feed it to L2 switch forwarding Algorithm. . .*/
```

Phase 2: L2 Switch Mac Learning and Forwarding

Layer2/l2switch.c

```
void
l2_switch_recv_frame ( interface_t *interface,
                                              char *pkt, unsigned int pkt_size) {


          l2_switch_perform_mac_learning(..)
          l2_switch_forward_frame(..)
}
```

This completes our L2 switch basic functionality !

L2 Switching Implementation -> VLAN Support

Goal : Vlan Support

- Implementing Vlan Support and Vlan based Mac forwarding
- Making our L2 Switches Vlan Aware
- Pre-requisite :
  - Understand concepts of VLANs
  - Understand 802.1q Header format
  - Vlan Tagging and Un-tagging of ethernet Hdr frames
  - I will discuss theory in fast pace as required to discuss our implementation of codes

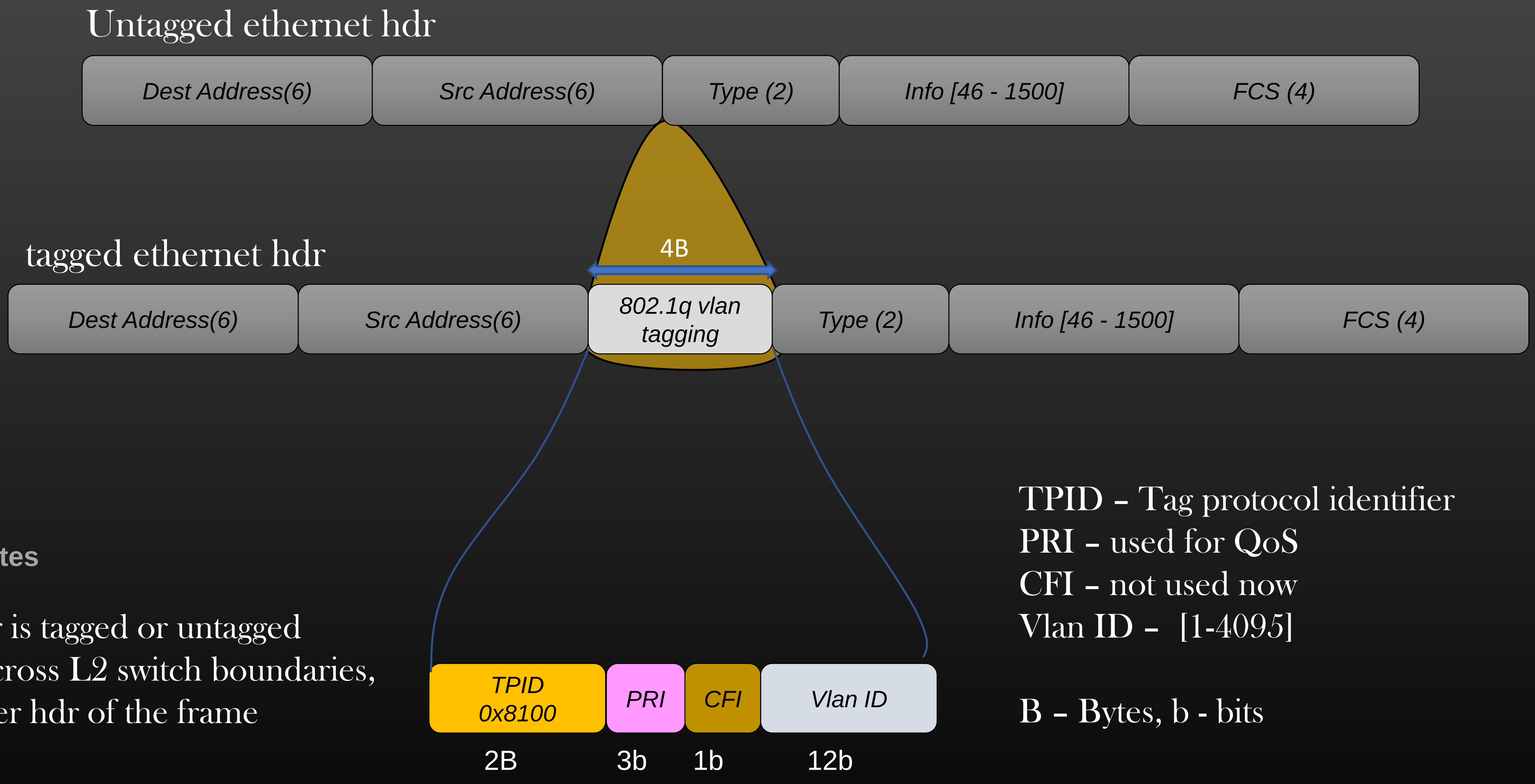
L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging

- Until now, out L2 switches were not Vlan aware
  - Our L2 ports were operating in ACCESS mode, and not in any vlan

- Vlan Aware L2 Switches performs Vlan tagging or un-tagging on frames which they receive and process

- To start with, we shall discuss the implementation of APIs responsible to tag or un-tag the frames with a given VLAN ID.

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging



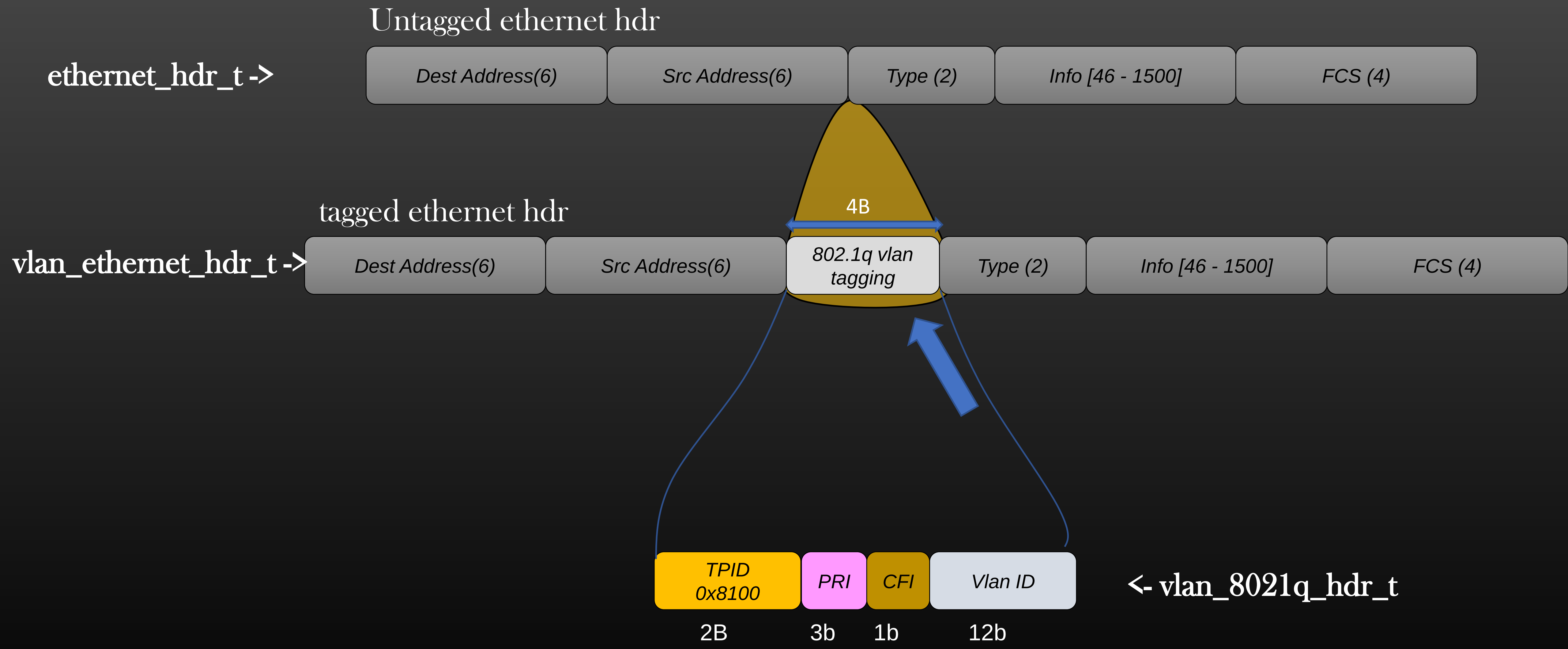
802.1Q Vlan Tag – 4 Bytes

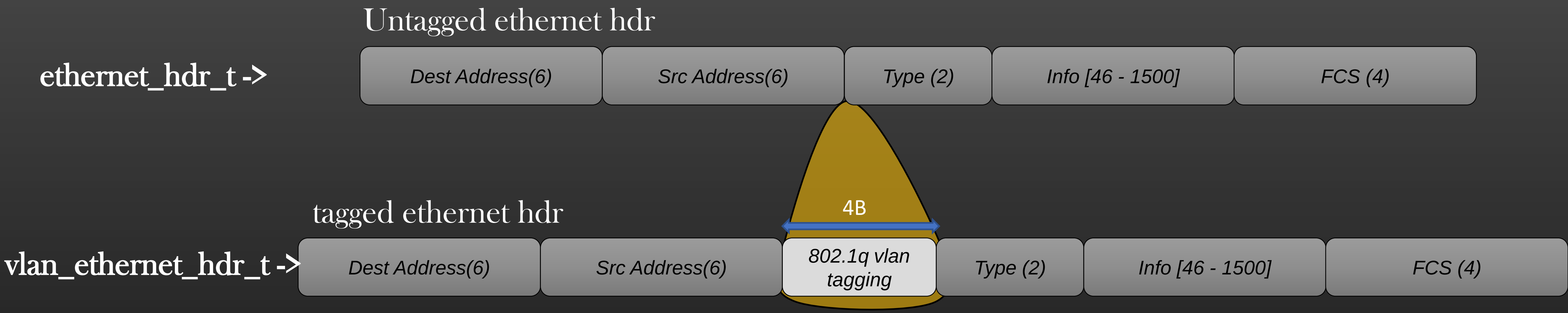
Only ethernet header is tagged or untagged when frame moves across L2 switch boundaries, no change in any other hdr of the frame

TPID – Tag protocol identifier
PRI – used for QoS
CFI – not used now
Vlan ID – [1-4095]

B – Bytes, b - bits

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> Tagged Ethernet Header



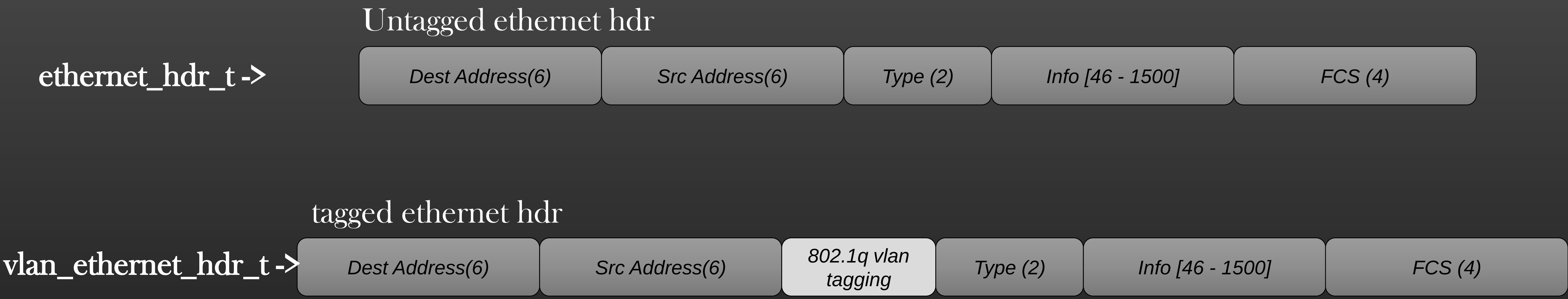


```
ethernet_hdr_t *ethernet_hdr = (ethernet_hdr_t *)pkt;
```

- Throughout our code, we shall represent the ethernet hdr of the frame with default structure ethernet_hdt_t, though it could be tagged or not

- Once it is confirmed that the frame is tagged, we shall typecast ethernet_hdr_t into vlan_ethernet_hdr_t

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> Tagged Ethernet Header

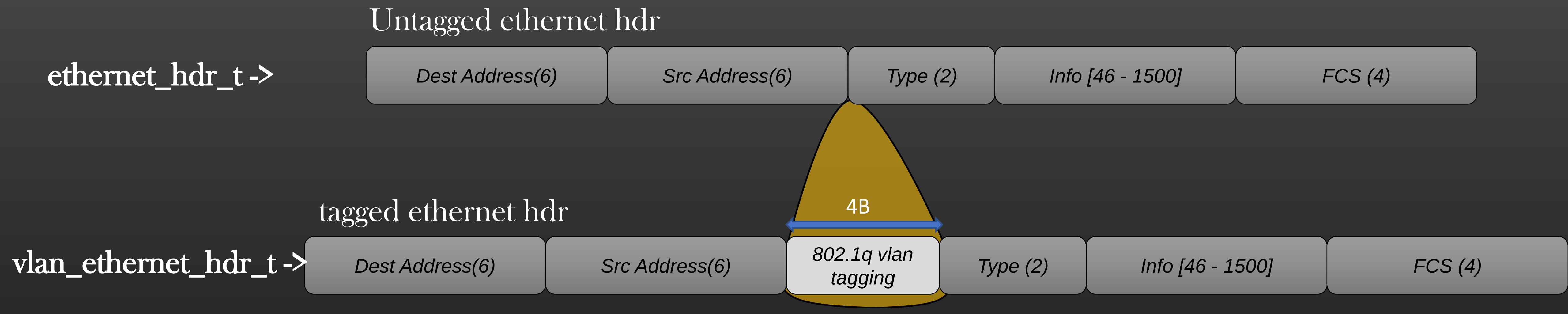


For Example :

ethernet_hdr_t *frame;   /*Given*/

How would you determine whether this frame is tagged or not ?

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> Tagged Ethernet Header

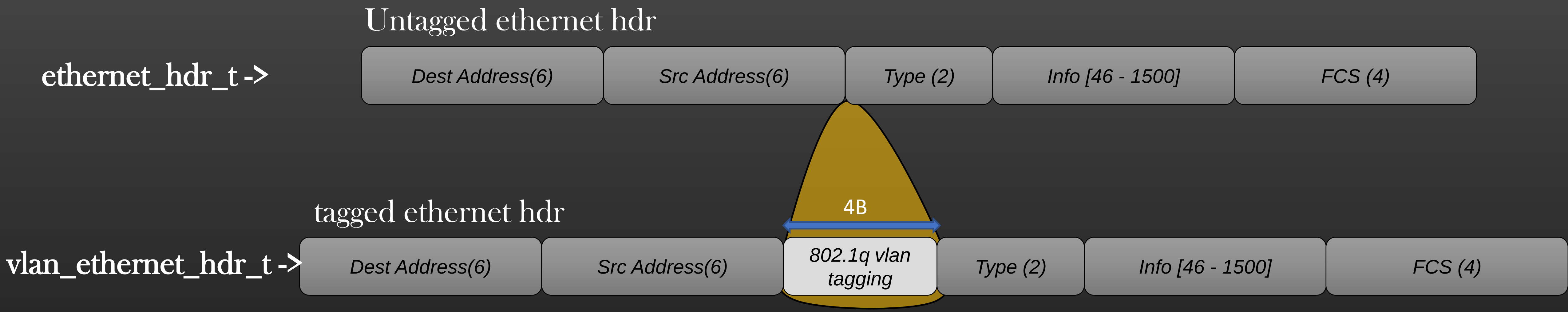


Tip : 13th and 14th bytes of both Type of hdrs contains protocol Identifier !

```
static inline vlan_8021q_hdr_t *
is_pkt_vlan_tagged (ethernet_hdr_t *ethernet_hdr){

        if(ethernet_hdr->type == 0x8100)
                return (vlan_8021q_hdr_t *)&(ethernet_hdr->type );
        else
                return NULL;
}
```

Tip : ethernet_hdr->type reads 13th and 14th bytes from the beginning of the tagged or untagged frame.

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> Tagged Ethernet Header



API to find whether the frame is tagged or not :

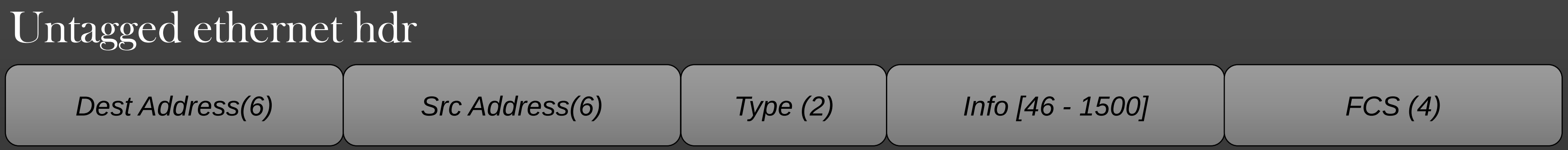
```
static inline vlan_8021q_hdr_t *
is_pkt_vlan_tagged (ethernet_hdr_t *ethernet_hdr);
```

Return ptr to *802.1Q* embedded hdr if frame is really tagged
Returns NULL if frame is not tagged with 802.1q hdr

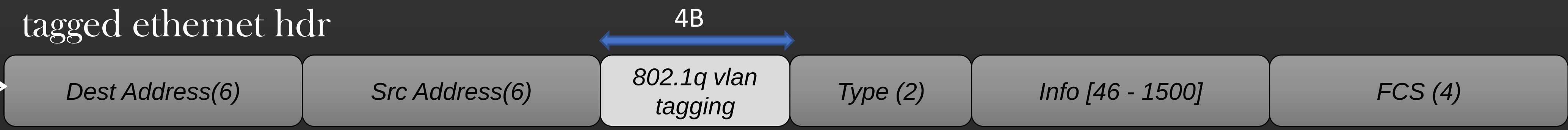
Interview Question !

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> Macros and APIs

ethernet_hdr_t ->



vlan_ethernet_hdr_t ->



Do the Assignment !!

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> APIs

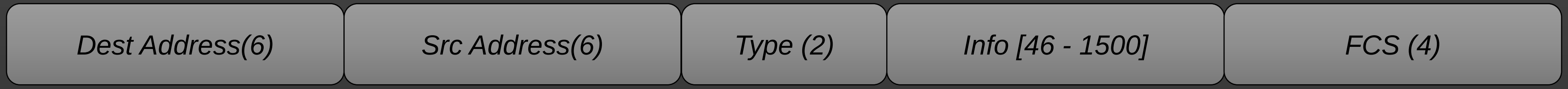
- In VLAN Based L2 Switching, L2 switches would need to convert tagged frames into untagged ones and vice versa as frame are L2 switched

- Therefore, it is essential for us to write APIs to carry out these tasks

- Assume that the frame is already “right shifted” and packet buffer has ample of space to the left of frame bits

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> APIs

Untagged ethernet hdr

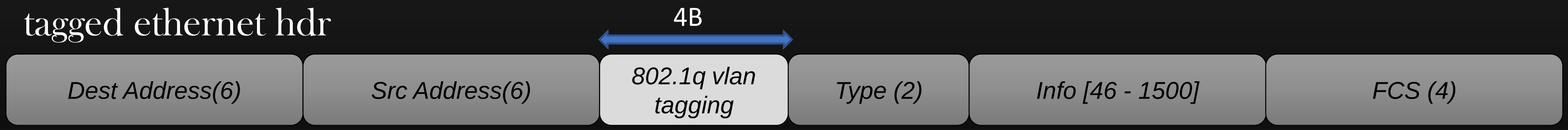


ethernet_hdr_t ->

Layer2/layer2.c/layer2.h

```
ethernet_hdr_t *
untag_pkt_with_vlan_id(ethernet_hdr_t *ethernet_hdr,
                                        unsigned int total_pkt_size,
                                        unsigned int *new_pkt_size);
```

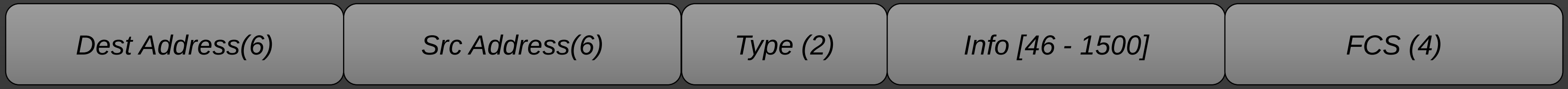
tagged ethernet hdr



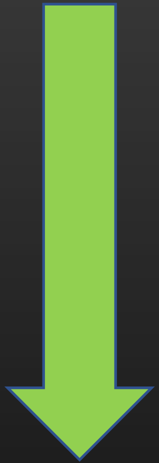
vlan_ethernet_hdr_t ->

L2 Switching Implementation -> VLAN Support -> Vlan Tagging and Untagging -> APIs

Untagged ethernet hdr



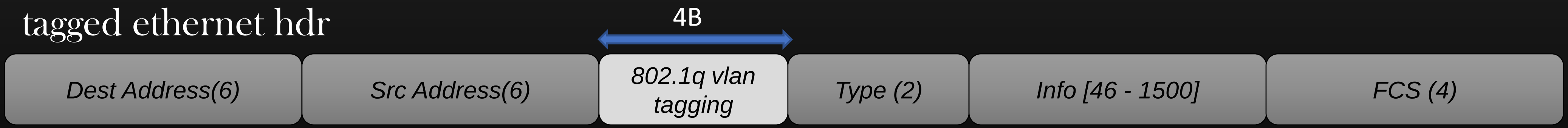
ethernet_hdr_t ->



Layer2/layer2.c/layer2.h

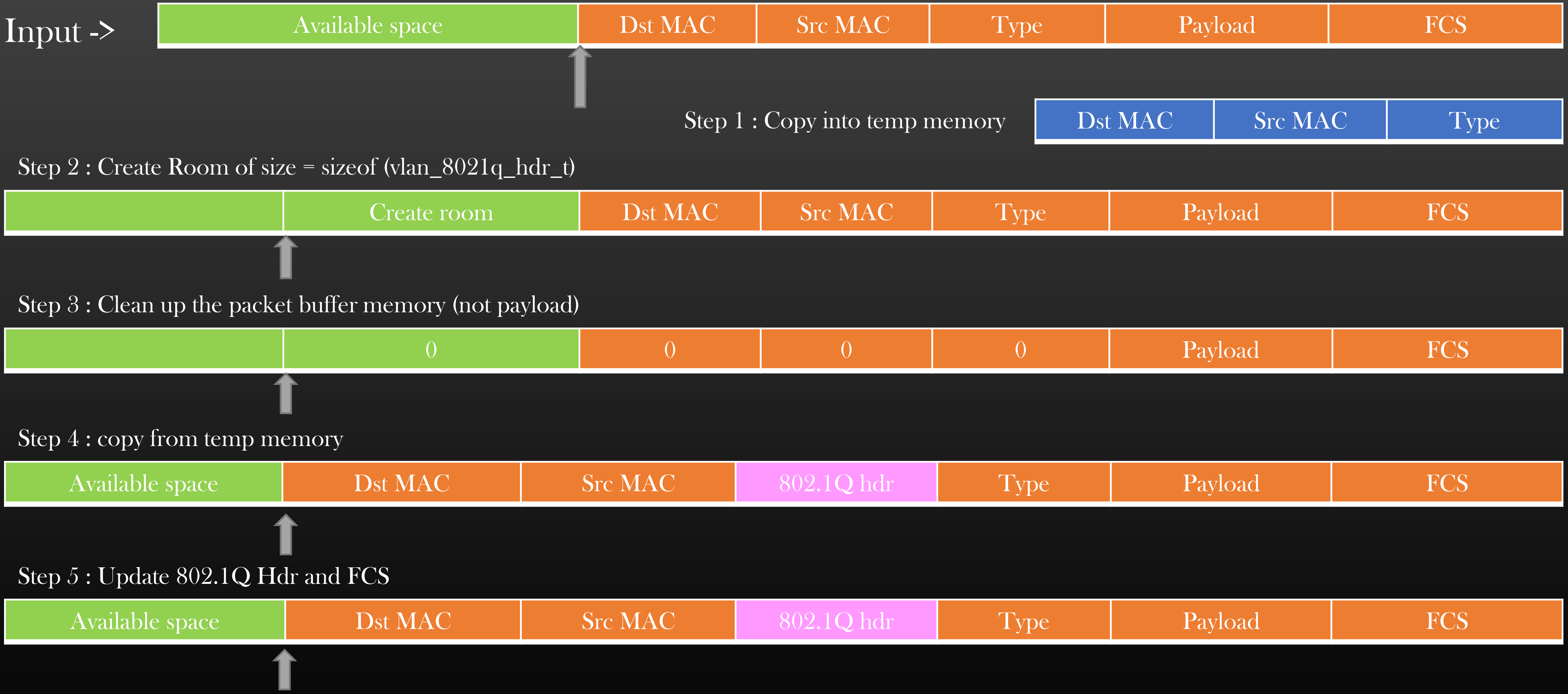
```
ethernet_hdr_t *
tag_pkt_with_vlan_id(ethernet_hdr_t *ethernet_hdr,
                     unsigned int total_pkt_size,
                     int vlan_id,
                     unsigned int *new_pkt_size);
```

tagged ethernet hdr



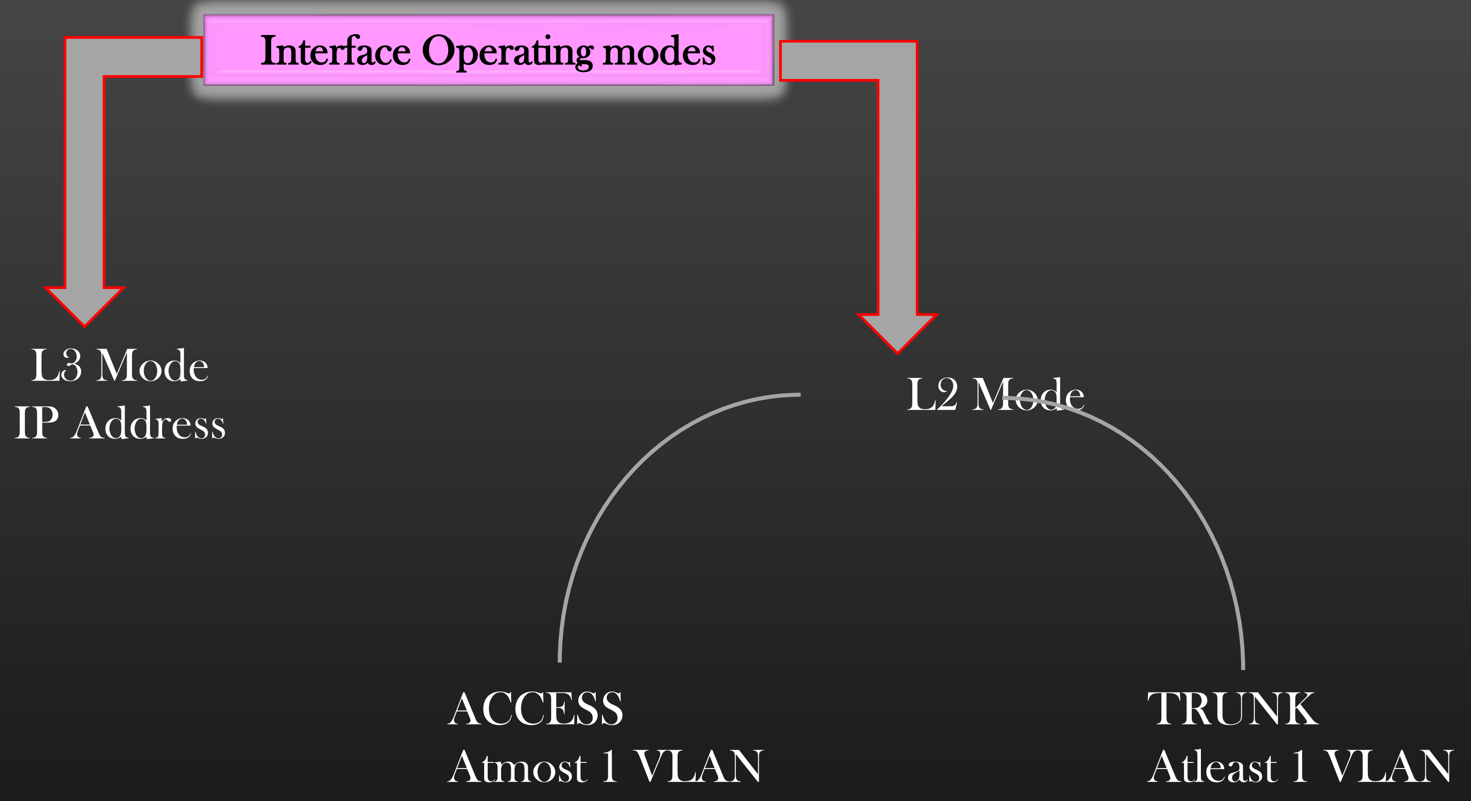
vlan_ethernet_hdr_t ->

Tagging the Frame : Steps



Step 6 : Free the temp memory if allocated on Heap and Return updated new frame header pointer

L2 Switching Implementation -> Interface Vlan member ship



- TRUNK interfaces are used to connect two L2 Switches
- ACCESS interfaces connects L2 Switches with Hosts

L2 Switching Implementation -> Interface Vlan member ship -> Data Structures

```
typedef struct intf_nw_props_ {
        . . .
        unsigned int vlans[MAX_VLAN_MEMBERSHIP];
        . . .
} intf_nw_props_t;
```

- If interface is operating in ACCESS mode, then only vlan[0] MAY be set to vlan no
- If interface is operating in VLAN mode, then vlan[ ] can contain upto MAX_VLAN_MEMBERSHIP vlan IDs

APIs : Layer2/layer2.c

```
void
node_set_intf_l2_mode (node_t *node, char *intf_name,
                                        intf_l2_mode_t intf_l2_mode);

void
node_set_intf_vlan_membsership( node_t *node,
                                                char *intf_name,
                                                unsigned int vlan_id);
```

L2 Switching Implementation -> Interface Vlan member ship -> Data Structures

Configuring using CLIs (Optional)

CLI :

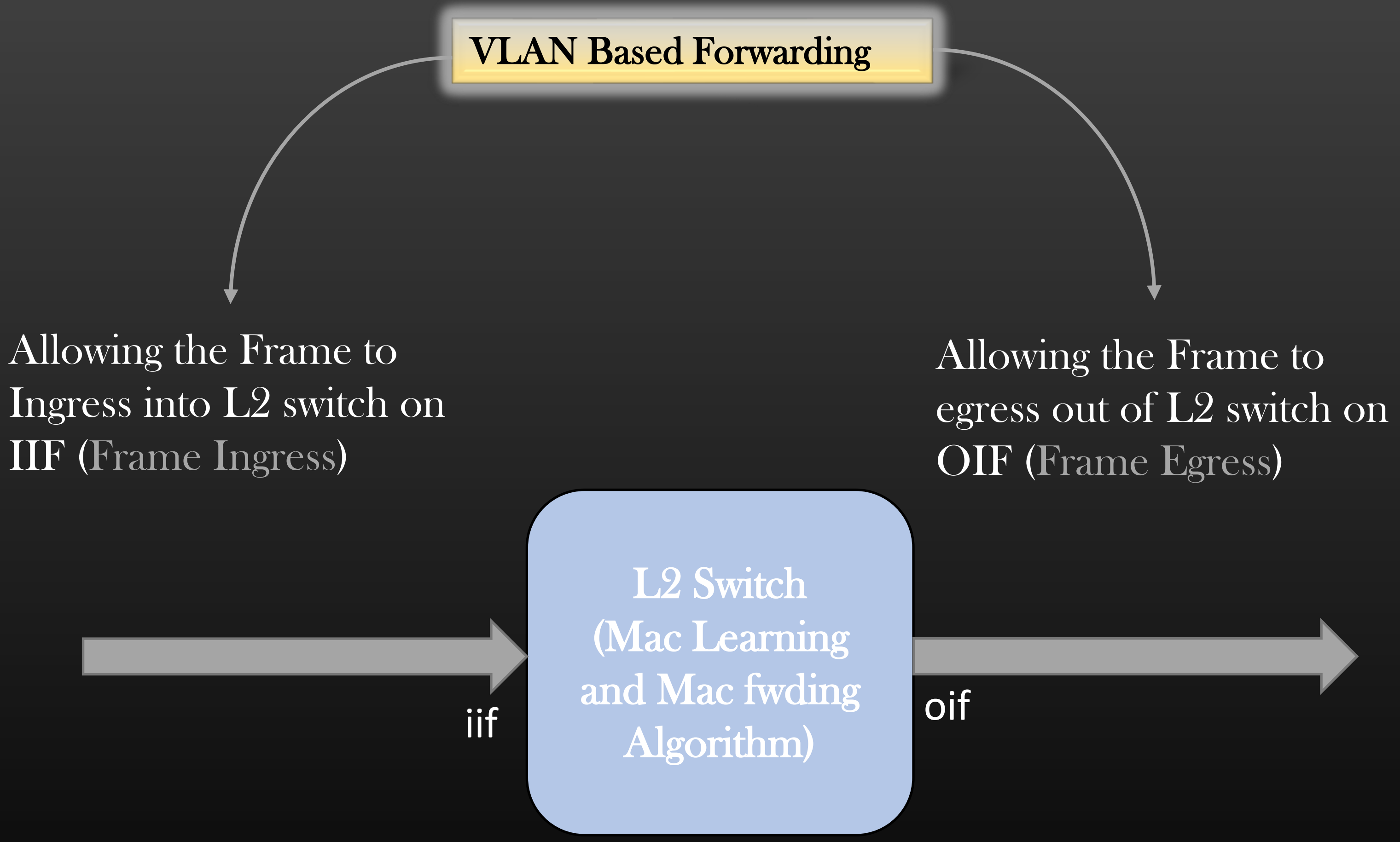
To set interface mode :

```
config node <node-name> interface <if-name> l2mode <l2-mode-val>
```

To configure Vlan membership:

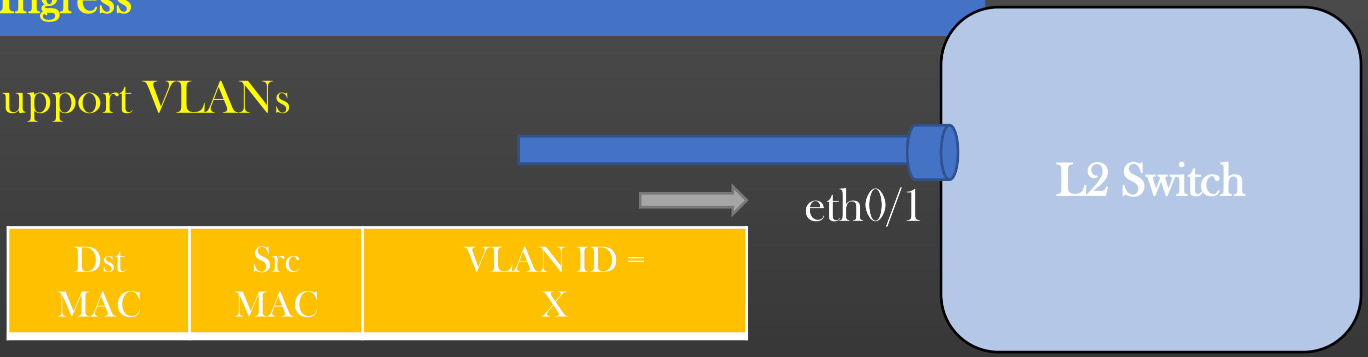
```
config node <node-name> interface <if-name> vlan <vlan-id (1-4095)>
```

L2 Switching Implementation -> VLAN Support -> VLAN based Forwarding


VLAN Based Forwarding
Allowing the Frame to Ingress into L2 switch on IIF (Frame Ingress)
Allowing the Frame to egress out of L2 switch on OIF (Frame Egress)
L2 Switch (Mac Learning and Mac fwding Algorithm)
iif
oif

L2 Switching Implementation -> VLAN Support -> Frame Ingress

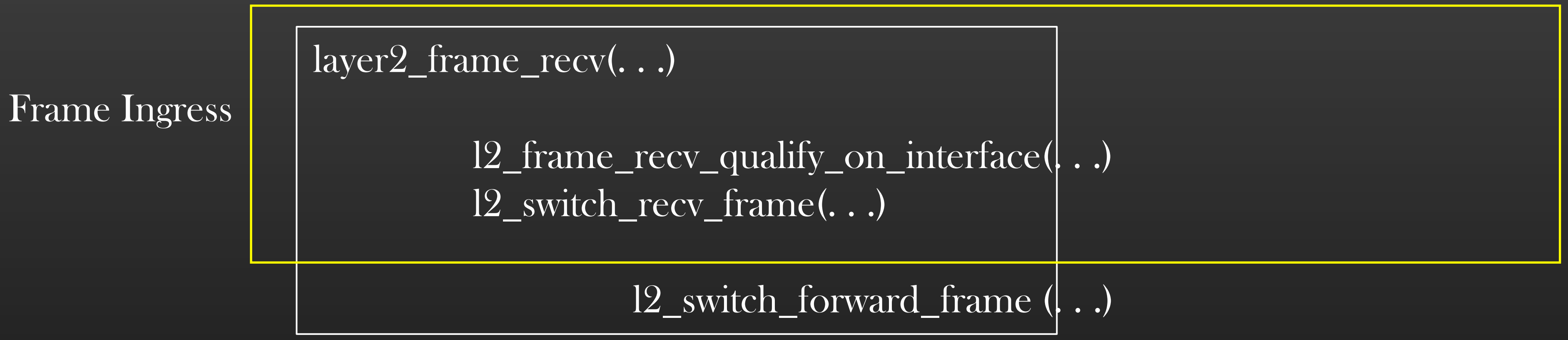
Enhancing : l2_frame_recv_qualify_on_interface(. . .) to support VLANs



Case #	Interface Operation mode	L2 mode Type (Access or Trunk)	Pkt Vlan tag	Interface vlan id	Action
1	L3 mode	-	No	-	If intf mac == Dst mac Or Dst mac is Broadcast mac then Accept, else reject
2	L3 mode	-	Yes	-	Drop the pkt
3	L2 mode	Access	No	Not enabled	Correction Drop the pkt
4	L2 mode	Access	Yes	Not enabled	Drop the pkt
5	L2 mode	Access	Yes (= X)	Yes (=Y)	L2 switch the frame if X = Y Drop the pkt with X!= Y
6	L2 mode	Access	No	Yes (= Y)	Tag the pkt with vlan Y, and L2 switch the frame
7	L2 mode	Trunk	No	Not enabled	Drop the pkt
8	L2 mode	Trunk	No	Enabled	Drop the pkt
9	L2 mode	Trunk	Yes (= X)	Yes { = Y}	Drop the pkt if X ≠ Y Else, L2 switch the frame
10	None	-	Don't matter	-	Drop the pkt

L2 Switching Implementation -> VLAN Support -> Frame Egress

➢ Once the L2 Switch Accepts the frame, it needs to forward the frame out of other L2 interfaces as by dictated by matching mac table entry


Frame Ingress
layer2_frame_recv(. . .)
l2_frame_recv_qualify_on_interface(. . .)
l2_switch_recv_frame(. . .)
l2_switch_forward_frame (. . .)

L2 Switching Implementation -> VLAN Support -> Frame Egress

➢ Once the L2 Switch Accepts the frame, it needs to forward the frame out of other L2 interfaces as by dictated by matching mac table entry

Frame Ingress

```
layer2_frame_recv(. . .)

              l2_frame_recv_qualify_on_interface(. . .)
              l2_switch_recv_frame(. . .)

                            l2_switch_forward_frame (. . .)
```

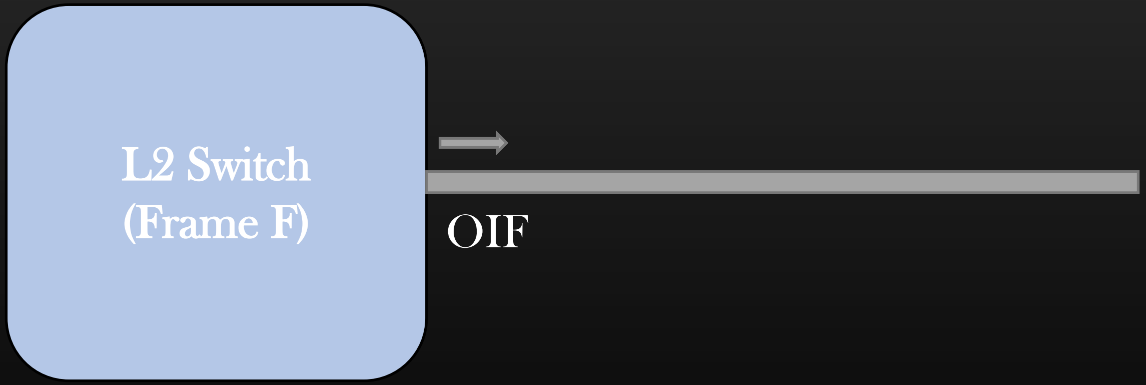
Frame Egress

```
                                          /*MAC table look up*/

                                                        l2_switch_send_pkt_out (. . .)
                                                                /*Check conditions to forward the frame*/

                                                                send_pkt_out(. . .)
                                                                        /*Finally send the frame out of a device*/
```

L2 Switching Implementation -> VLAN Support -> Frame Egress

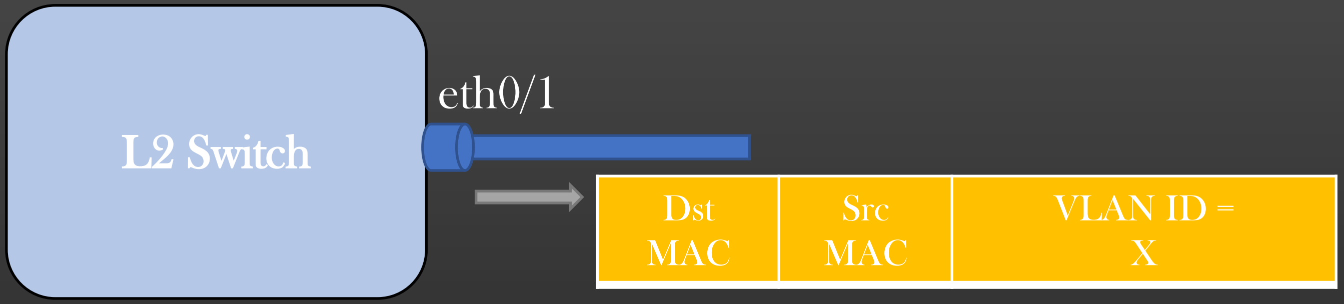
- We shall write an API specific to L2 switch, which checks all conditions before it decide to finally send the frame out of a L2 interface of a switch

```
Layer2/l2switch.c

static bool_t
l2_switch_send_pkt_out  (char *pkt,
                                        unsigned int pkt_size,
                                        interface_t *oif)
```



L2 Switching Implementation -> VLAN Support -> Frame Egress

API : l2_switch_send_pkt_out (. . .)
Layer2/l2switch.c



Case #	Interface Operation mode	L2 mode Type (Access or Trunk)	Pkt Vlan tag	Interface vlan id	Action
0	L3 Mode	-	Don't matter	-	Assert
1	L2 Mode	ACCESS	No	Not enabled	Correction Assert
2	L2 Mode	ACCESS	No	= X	Do Not Forward
3	L2 Mode	ACCESS	= X	= Y	send_pkt_out if X = Y & Untag the frame , else Do Not Forward
4	L2 Mode	ACCESS	= X	Not enabled	Do Not Forward
5	L2 Mode	TRUNK	= X	= {Y}	Forward if X belongs to Y, else Do not Forward
6					Do Not Forward

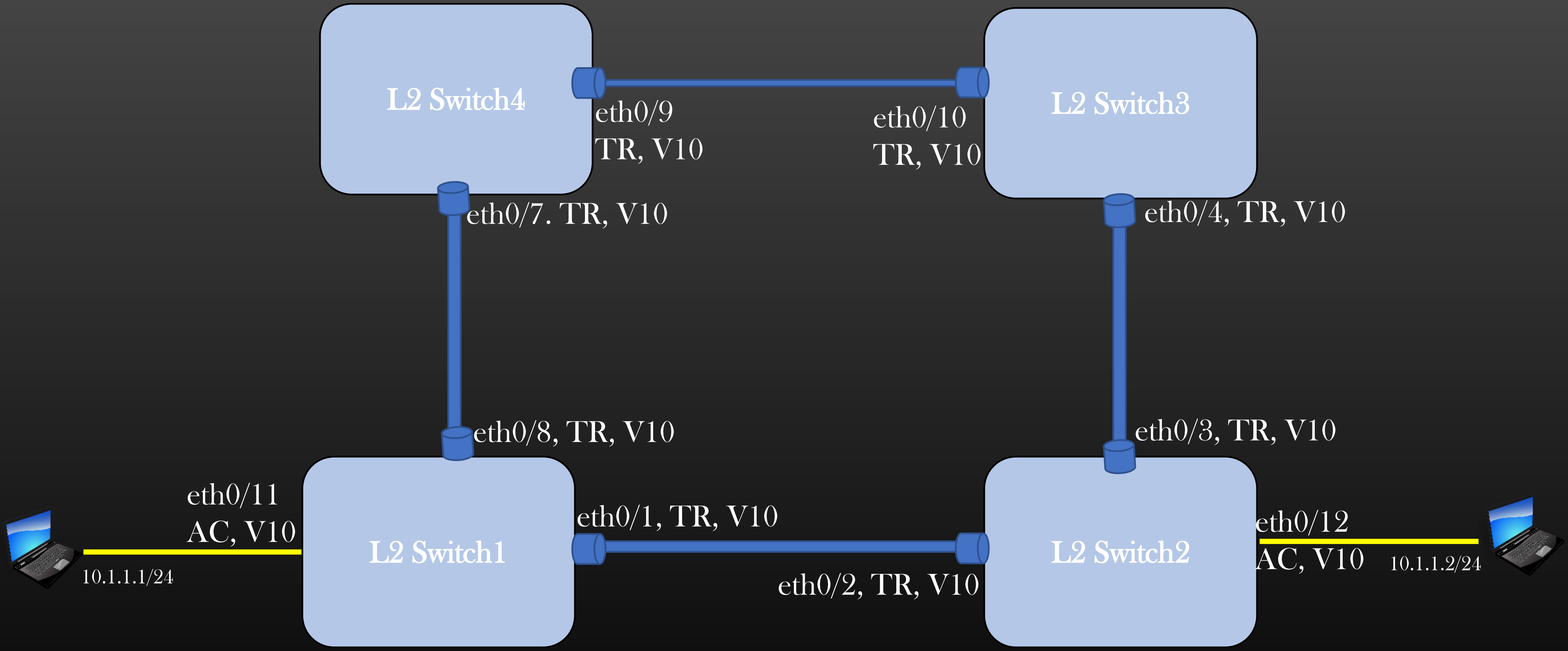
L2 Switch Forward Algorithm Revisited

```
void
l2_switch_forward_frame ( node_t *node, interface_t *recv_intf,
                          ethernet_hdr_t *ethernet_hdr,
                          unsigned int pkt_size);
```

Must Use :

l2_switch_flood_pkt_out(. . .) &
l2_switch_send_pkt_out(. . .)

L2 Switching Implementation -> VLAN Support -> L2 Loops



Chaos !! Infinite loops all over the LAN segment !
Solution : Spanning Tree Protocol

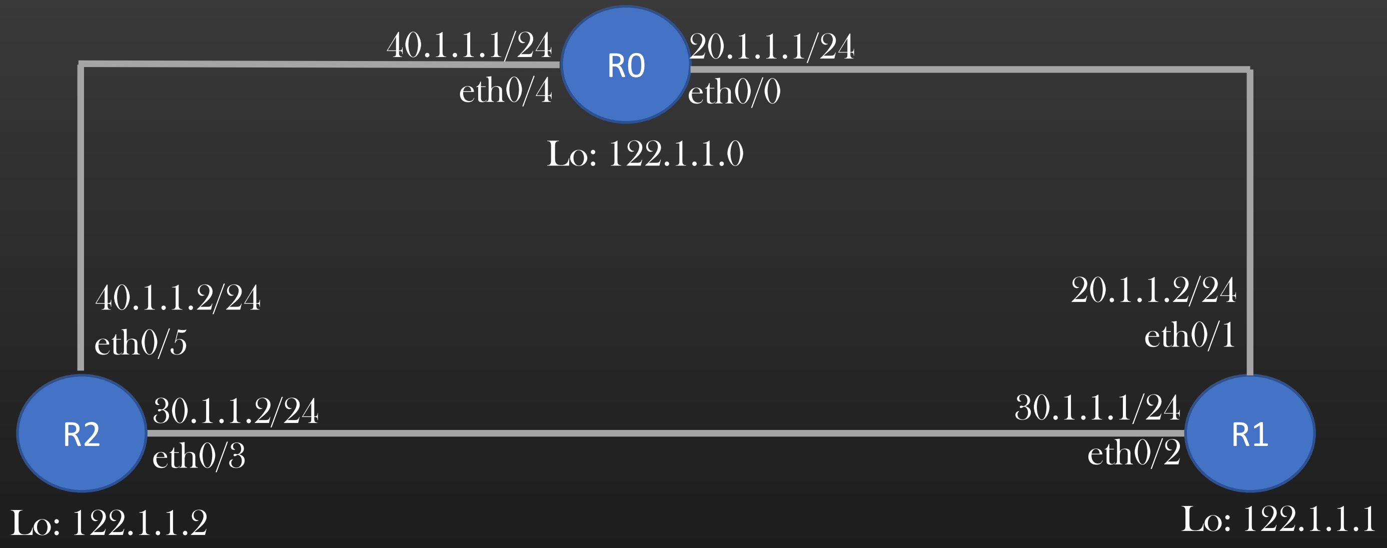
END

L3 Routing !

L3 Routing Implementation

Goal

- ➢ Implement Layer 3 Routing - IP Based Forwarding

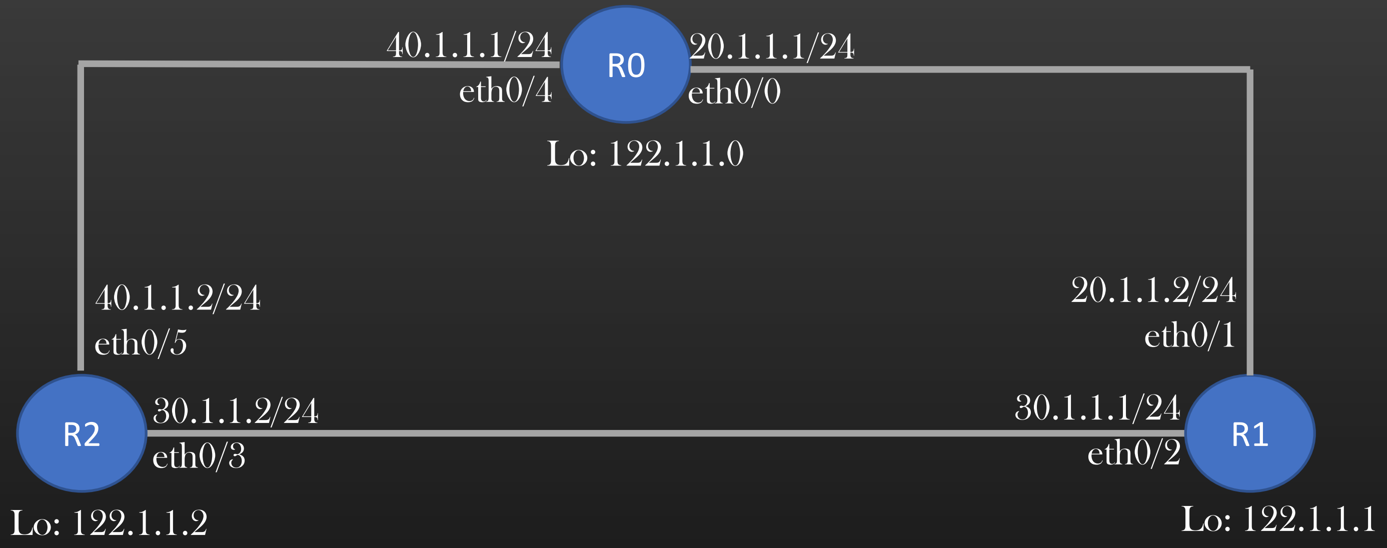


- ➢ Pre-requisites :
  - ➢ Understand L3 routing
  - ➢ Understanding L3 Routing Table
  - ➢ Longest Prefix Match
  - ➢ Local and Remote subnets
  - ➢ Loopback Addresses

L3 Routing Implementation

Goal

➢ Implement Layer 3 Routing - IP Based Forwarding



➢ All nodes in the topology are L3 routers

➢ Router's must have L3 routes to reach IP-addresses falling in remote subnets

➢ All Routers and Hosts must have L3 routing Table

➢ We shall implement L3 routing logic

For router R0, remote IP addresses are :
122.1.1.1, 122.1.1.2, 30.1.1.1, 30.1.1.2

R0 needs L3 routing support to reach remote ip addresses !

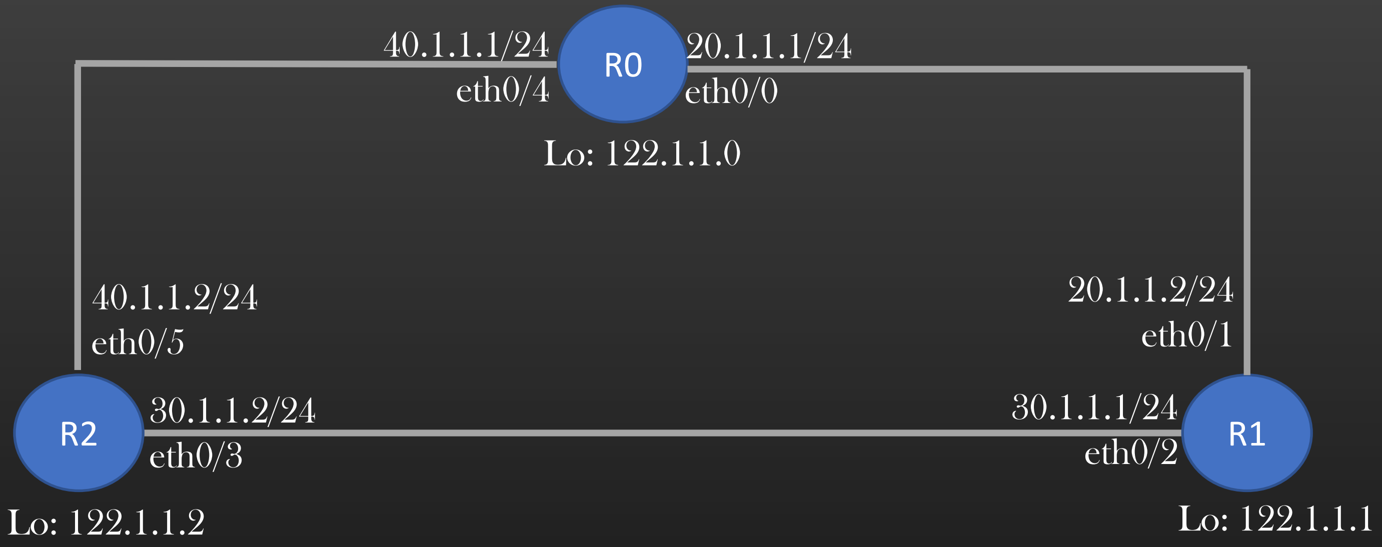
L3 Routing Implementation

L3 Routing Infrastructure Setup

- We first need to develop L3 routing Infrastructure – 8 phases :
  - Phase 1 : L3 Routing Table Data structure Setup
  - Phase 2 : L3 Route Installation/Configuration
  - Phase 3 : Defining IP Hdr
  - Phase 4 : Topology Used and ARP assumption
  - Phase 5 : Implementing Ping as an application to test our L3 code
  - Phase 6 : TCP/IP Stack Layers interaction
  - Phase 7 : L3 Routing Concepts - Revisited
  - Phase 8 : Final Flowcharts to implement L3 routing

L3 Routing Implementation -> Phase 1 Data structure Setup

> Setting Up the Routing Table



RT for R0

Dest	Mask	Is_direct	Gw_ip	oif
122.1.1.0	32	TRUE	-	-
20.1.1.0	24	TRUE	-	-
30.1.1.0	24	FALSE	20.1.1.2	eth0/0
122.1.1.1	32	FALSE	20.1.1.2	eth0/0
122.1.1.2	32	FALSE	40.1.1.2	eth0/4
40.1.1.0	24	TRUE	-	-

Layer3/layer3.h

```
typedef struct l3_route_{

    char dest[16];  /*key*/
    char mask;      /*key*/
    bool_t is_direct;  /*if set to True, then
                                    gw_ip and oif has n
    char gw_ip[16];     /*Next hop IP*/
    char oif [IF_NAME_SIZE]; /*OIF*/
    glthread_t rt_glue;
} l3_route_t;


typedef struct rt_table_{

    glthread_t route_list;
} rt_table_t;
```

L3 Routing Implementation -> Phase 1 Data structure Setup

CRUD APIs For Routing Table (Layer3.h/layer3.c):

```
void
init_rt_table (rt_table_t **rt_table);

void
rt_table_add_direct_route (rt_table_t *rt_table,
                           char *dst, char mask);

void
dump_rt_table (rt_table_t *rt_table);
```

```
void
rt_table_add_route (rt_table_t *rt_table,
                    char *dst, char mask,
                    char *gw, char *oif);

l3_route_t *
l3rib_lookup_lpm(rt_table_t *rt_table,
                 uint32_t dest_ip);
```

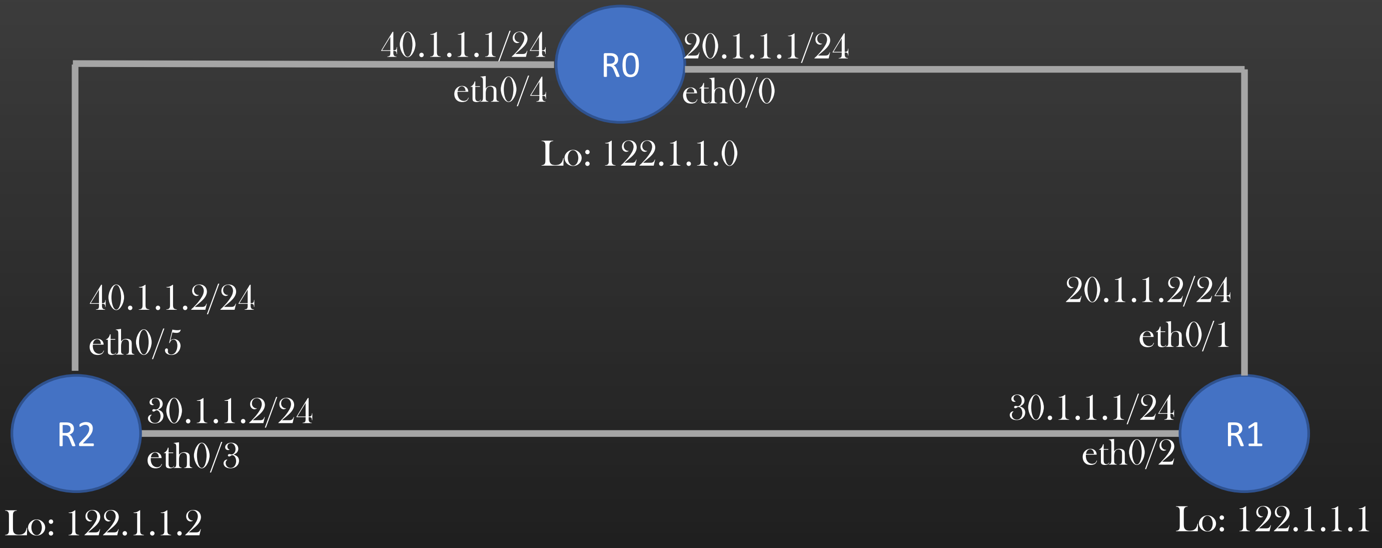
L3 Routing Implementation -> Phase 1 Data structure Setup

```
typedef struct node_nw_prop_{
. . .
arp_table_t *arp_table;
mac_table_t *mac_table;
rt_table_t *rt_table;
. . .
} node_nw_prop_t;
```

```
static inline void
init_node_nw_prop (node_nw_prop_t *node_nw_prop) {

. . .
init_rt_table(&(node_nw_prop->rt_table));
. . .
}
```

L3 Routing Implementation -> Phase 2 L3 Route Installation

Installation of L3 Local Routes



RT for R0

Dest	Mask	Is_direct	Gw_ip	oif
122.1.1.0	32	TRUE	-	-
20.1.1.0	24	TRUE	-	-
30.1.1.0	24	FALSE	20.1.1.2	eth0/0
122.1.1.1	32	FALSE	20.1.1.2	eth0/0
122.1.1.2	32	FALSE	40.1.1.2	eth0/4
40.1.1.0	24	TRUE	-	-

➢ Router must install the direct routes in its RT Automatically at the time of topology creation itself

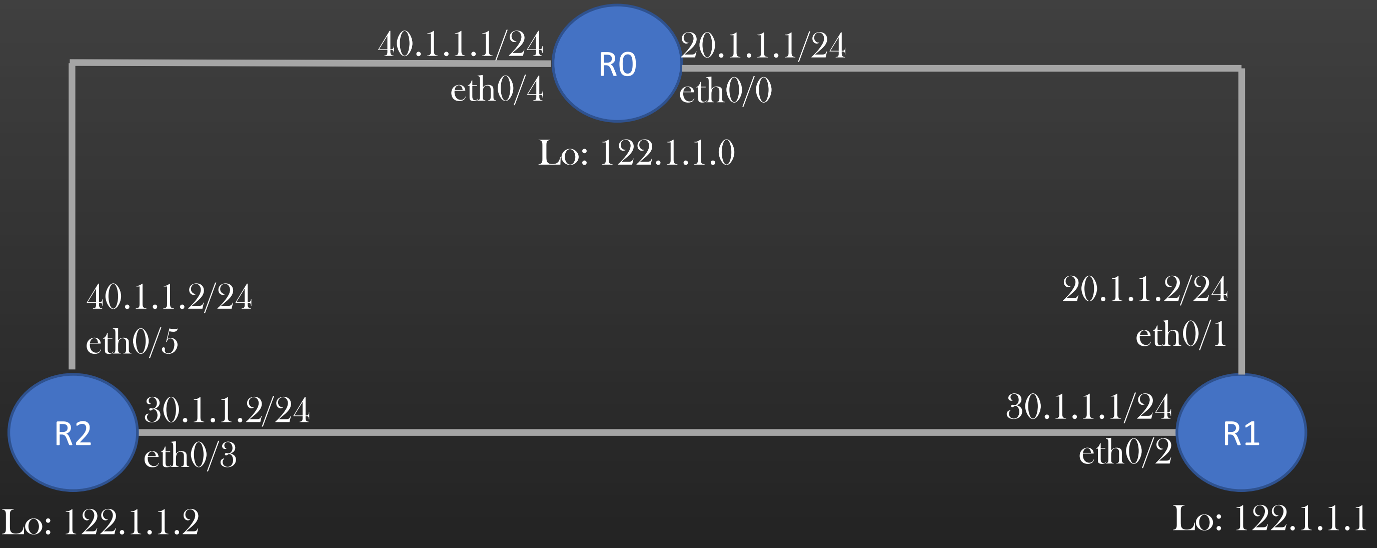
APIs to enhance :
node_set_loopback_address(. . .)
node_set_intf_ip_address(. . .)

CLI :
show node <node-name> rt

Backend handler :
```
static int
show_rt_handler(param_t *param, ser_buff_t *tlv_buf,
            op_mode enable_or_disable);
```

L3 Routing Implementation -> Phase 2 L3 Route Installation

Installation of Static L3 remote Routes



RT for R0

Dest	Mask	Is_direct	Gw_ip	oif
122.1.1.0	32	TRUE	-	-
20.1.1.0	24	TRUE	-	-
30.1.1.0	24	FALSE	20.1.1.2	eth0/0
122.1.1.1	32	FALSE	20.1.1.2	eth0/0
122.1.1.2	32	FALSE	40.1.1.2	eth0/4
40.1.1.0	24	TRUE	-	-

We need to Install L3 routes in L3 routers/Hosts
So that they can forward the traffic

CLI :

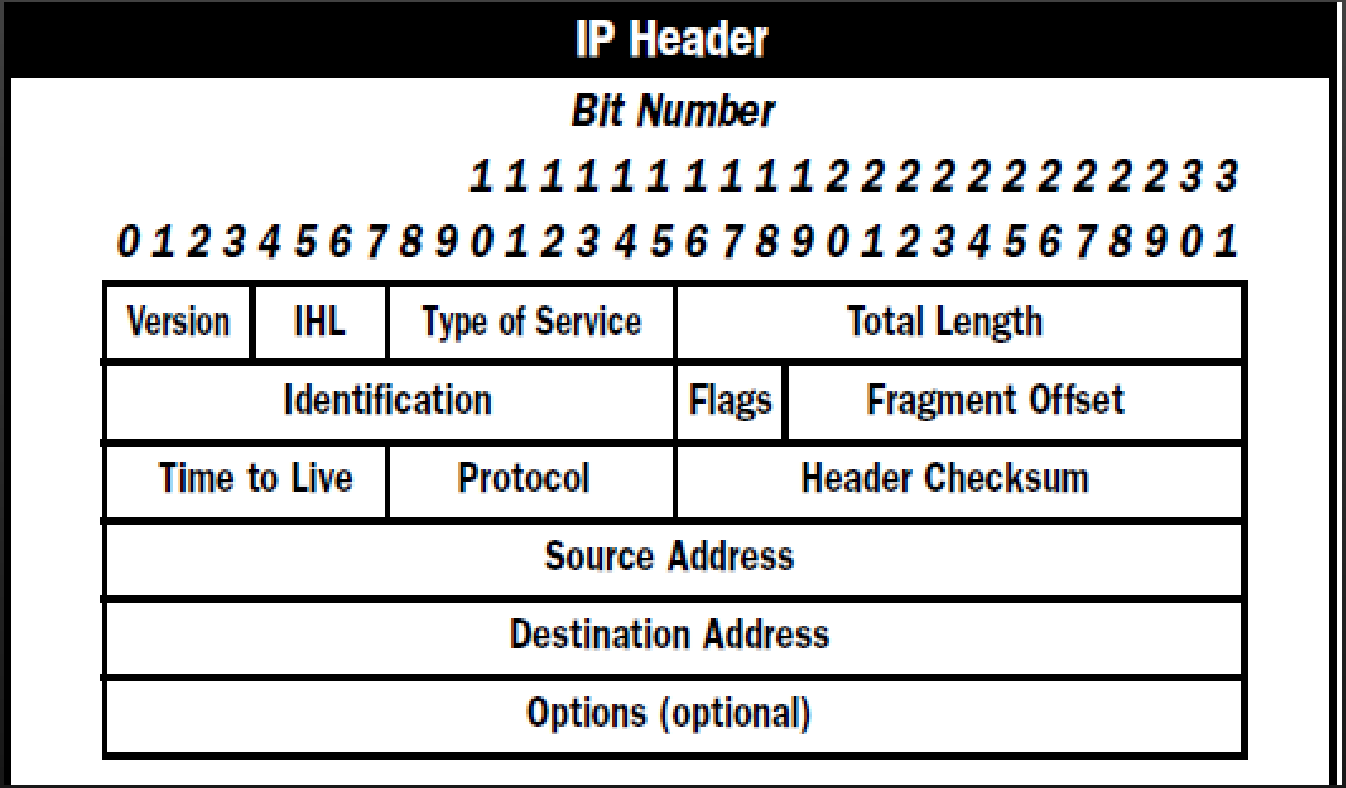
```
config node <node-name> route <dest> <mask>
        <gw-ip> <oif-name>
```

Back-end handler :

```
static int
l3_config_handler(param_t *param,
    ser_buff_t *tlv_buf,
    op_mode enable_or_disable);
```

L3 Routing Implementation -> Phase 3 Defining IP Hdr Structure

- Defining IP Hdr (Layer3/layer3.h)

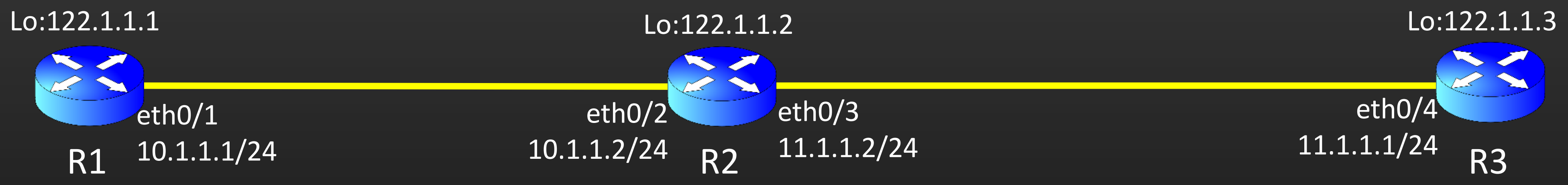


- Writing Macros

- Assignment on IP Hdr

L3 Routing Implementation -> Phase 4 Topology used and ARP assumption

Topology Used : linear_3_node_topo()

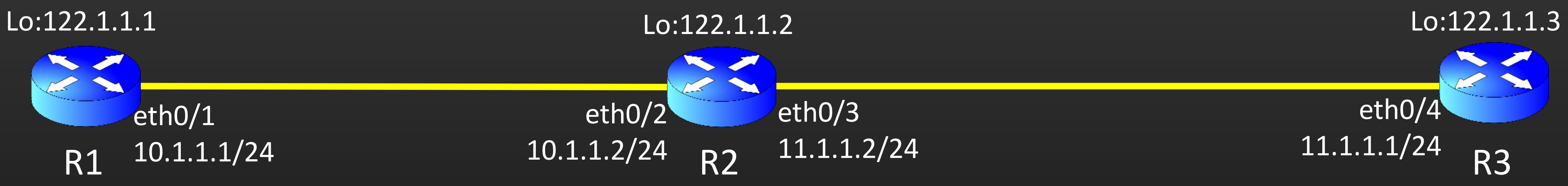


L3 Routing Implementation -> Phase 4 Topology used and ARP assumption

ARP Assumption :

> We shall assume that All routers have required ARP resolved already

> This is done to not to implement all complexity in one go, We shall deal how to resolve ARP on demand after we are done with L3 Implementation



CLI to resolve arp : run node <node-name> resolve-arp <ip-address>

> R1 should have ARP resolution for IP : 10.1.1.2

> R2 should have ARP resolution for IP : 11.1.1.1

L3 Routing Implementation -> Phase 5 Implementing Ping

- Since, we are about to implement the L3 Routing, there should be infrastructure in place using which we can incrementally test our L3 code

- We shall write a simplified ping application, which shall represent an application running in app layer of TCP IP Stack

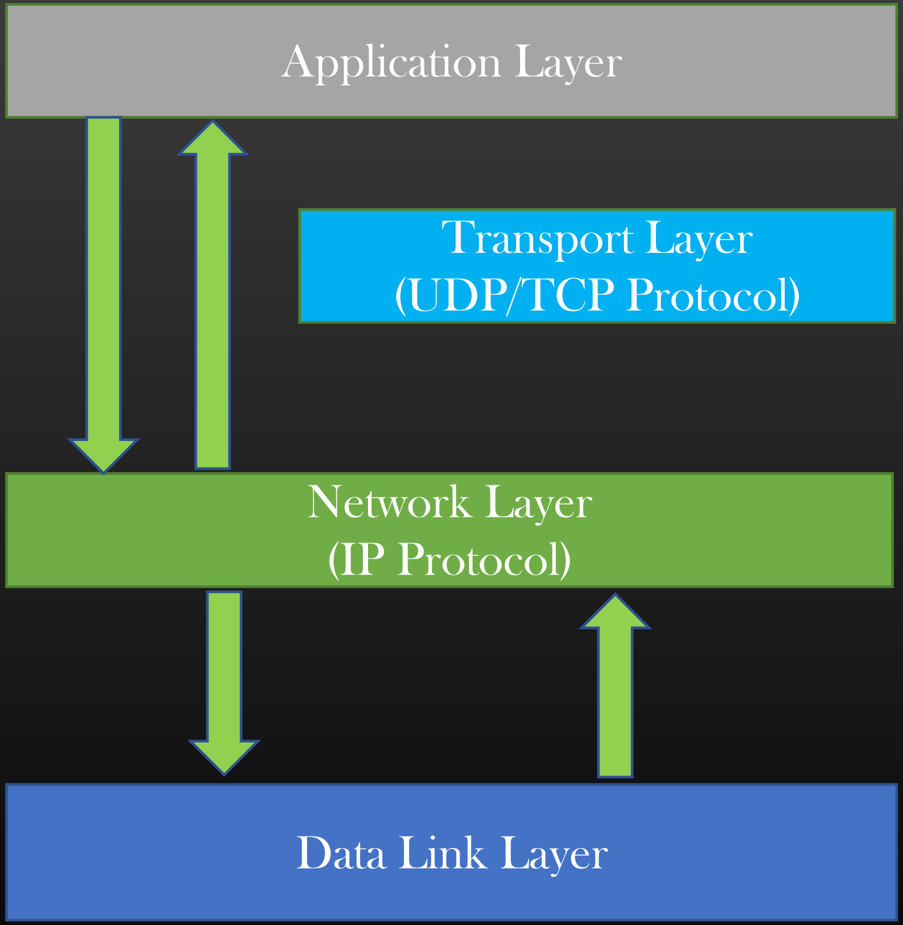
- The ping application will feed the data to network layer to be routed to remote destination node present in the network

- This Way Network Layer shall be stimulated, and we shall be able to test our L3 code

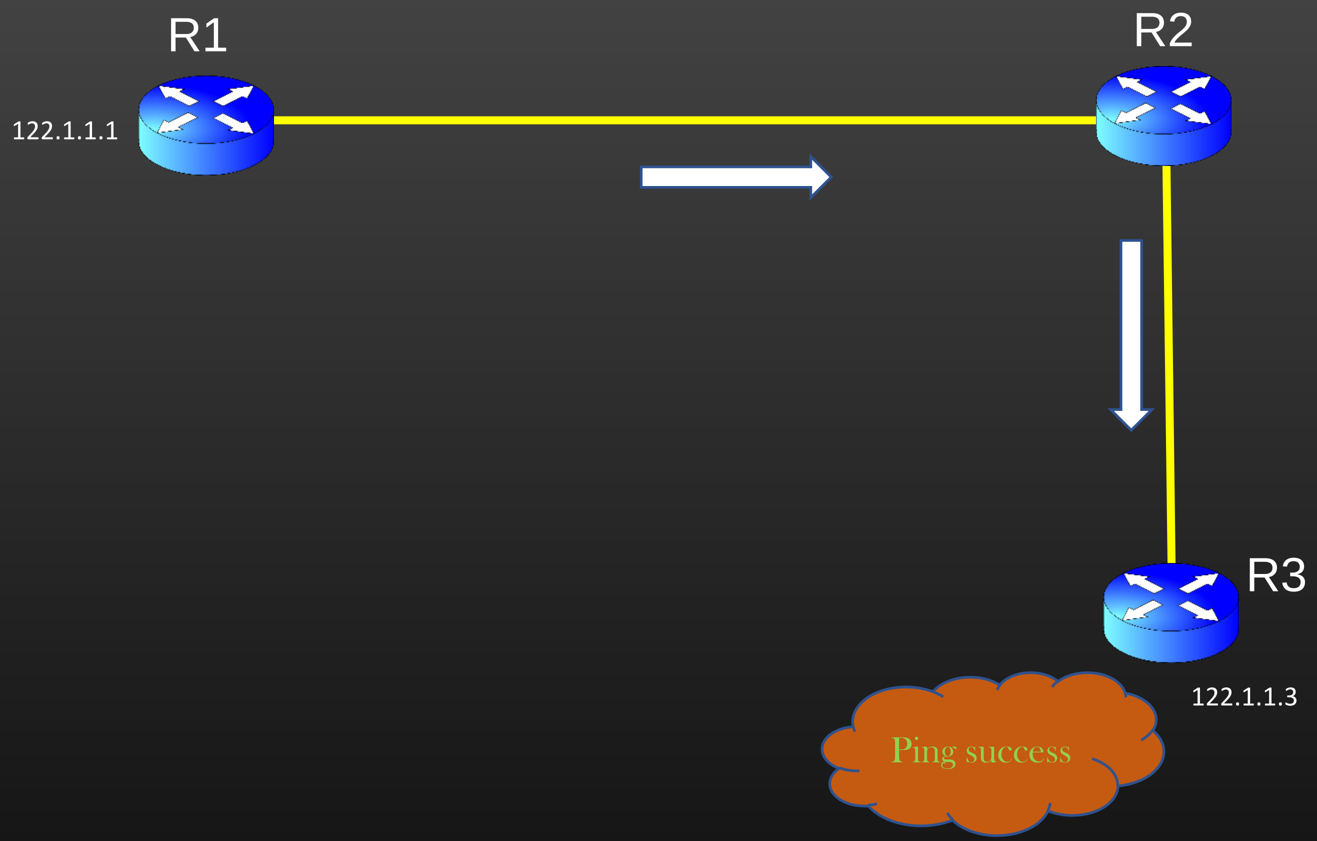
- Transport Layer is bypassed here. Application can run directly on top of Network Layer (Just like ARP application run directly on top of Data link layer)

- Not every packet being routed by Network Layer is TCP/UDP packet

- It is just a matter of writing one CLI to represent ping as an application


Application Layer
Transport Layer (UDP/TCP Protocol)
Network Layer (IP Protocol)
Data Link Layer

L3 Routing Implementation -> Phase 5 Implementing Ping



How to test :

CLI : run node <node-name> ping <ip-address>
Eg : run node R1 ping 122.1.1.3

File : nwcli.c
Backend handler :

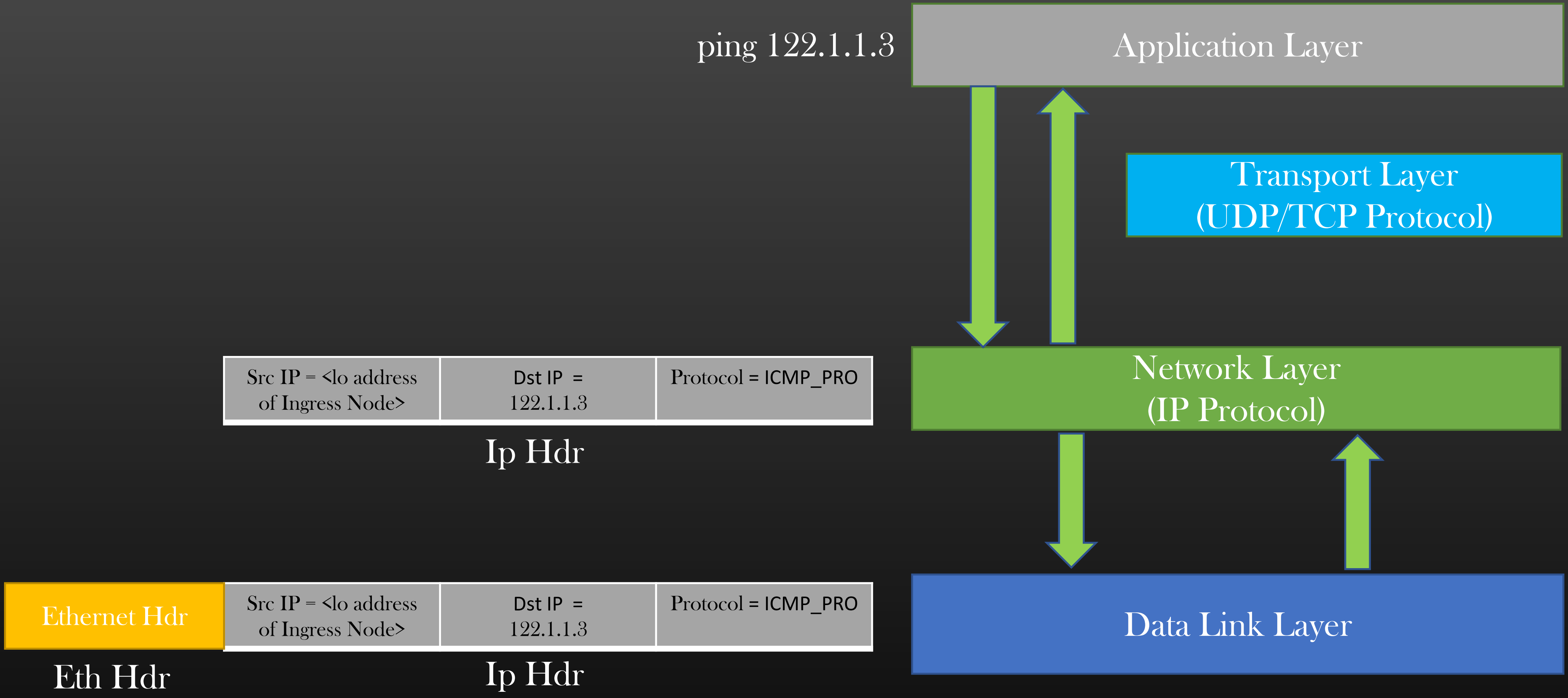
```
static int
ping_handler(param_t *param, ser_buff_t *tlv_buf,
                        op_mode enable_or_disable);
```

Which invokes :

Layer5/ping.c

```
extern void
layer5_ping_fn ( node_t *node, char *dst_ip_addr);
```

L3 Routing Implementation -> Phase 5 Implementing Ping



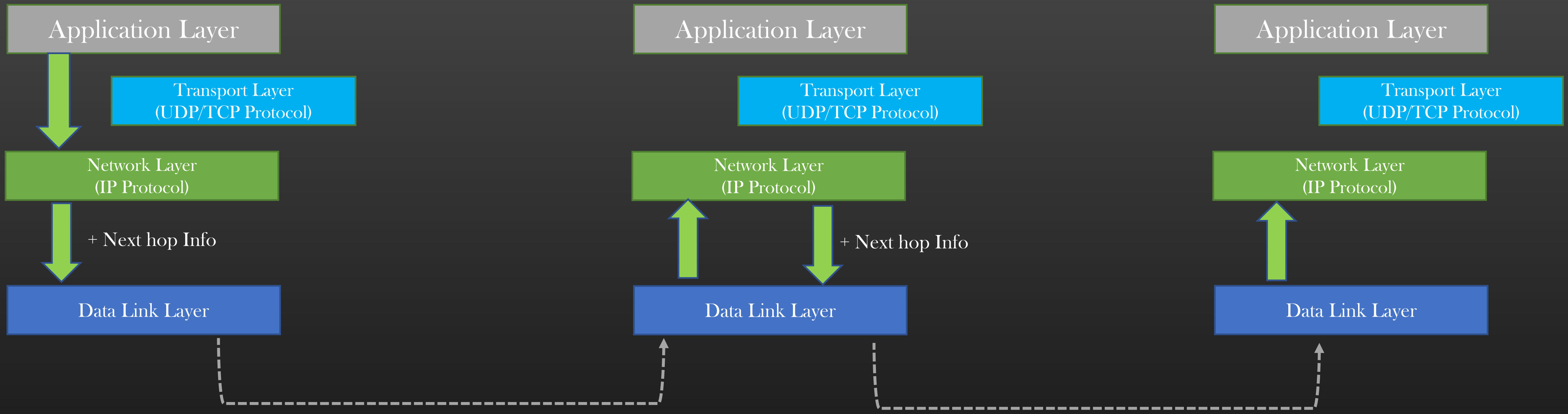
Src IP = <lo address of Ingress Node>	Dst IP = 122.1.1.3	Protocol = ICMP_PRO

Ethernet Hdr	Src IP = <lo address of Ingress Node>	Dst IP = 122.1.1.3	Protocol = ICMP_PRO

- There is no explicit application data in the pkt (Only eth hdr and IP hdr)
- For ping : Source side is triggered by the application layer, On Destination side Network layer responds

L3 Routing Implementation -> Phase 5 Implementing Ping

ping 122.1.1.3

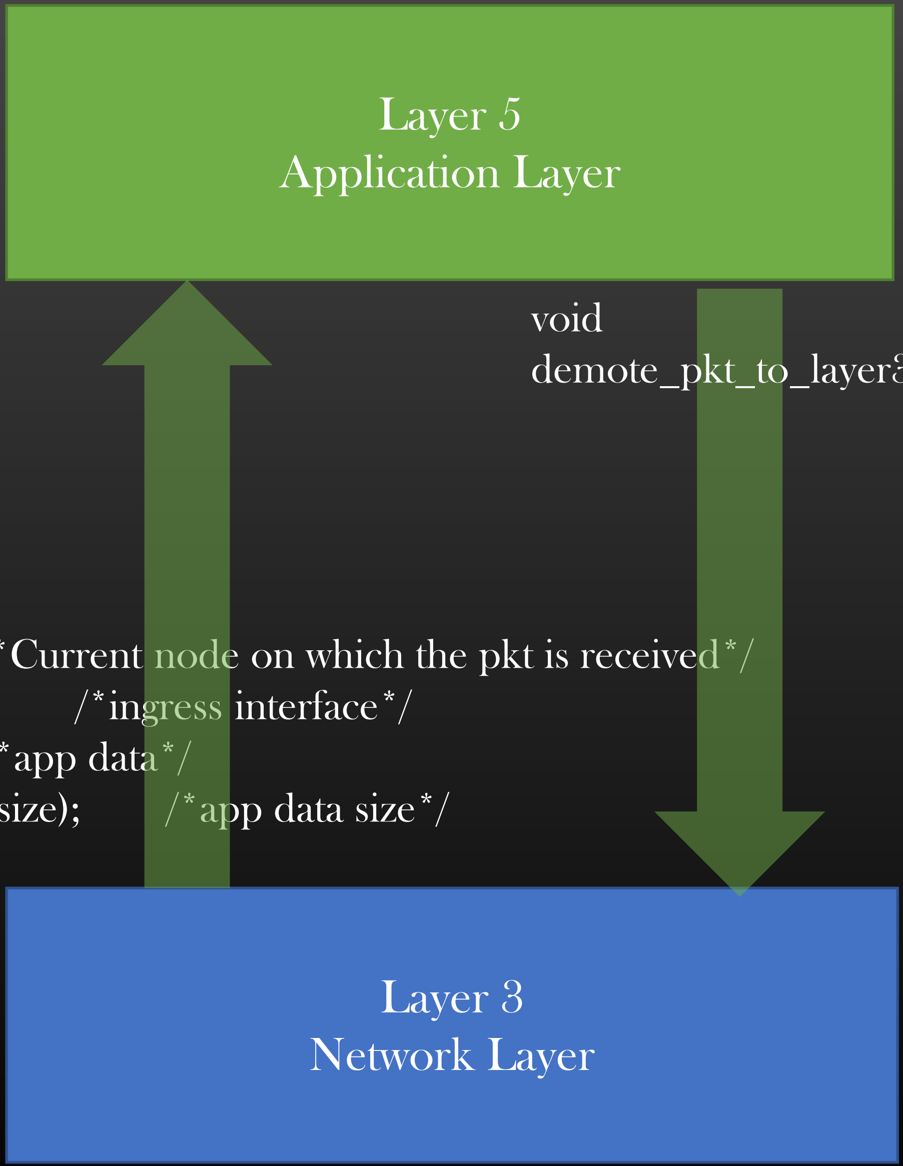

Application Layer
Transport Layer (UDP/TCP Protocol)
Network Layer (IP Protocol)
+ Next hop Info
Data Link Layer
Application Layer
Transport Layer (UDP/TCP Protocol)
Network Layer (IP Protocol)
+ Next hop Info
Data Link Layer
Application Layer
Transport Layer (UDP/TCP Protocol)
Network Layer (IP Protocol)
Data Link Layer

R1
Sets the ping Journey On

R2
L3 forwards the packet

R3
Reports "ping success"

L3 Routing Implementation -> Phase 6 Application and Network Layer Interaction

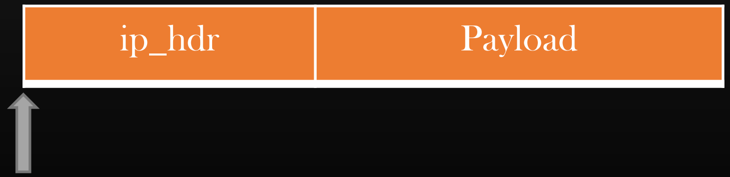


Layer3/layer3.c/.h

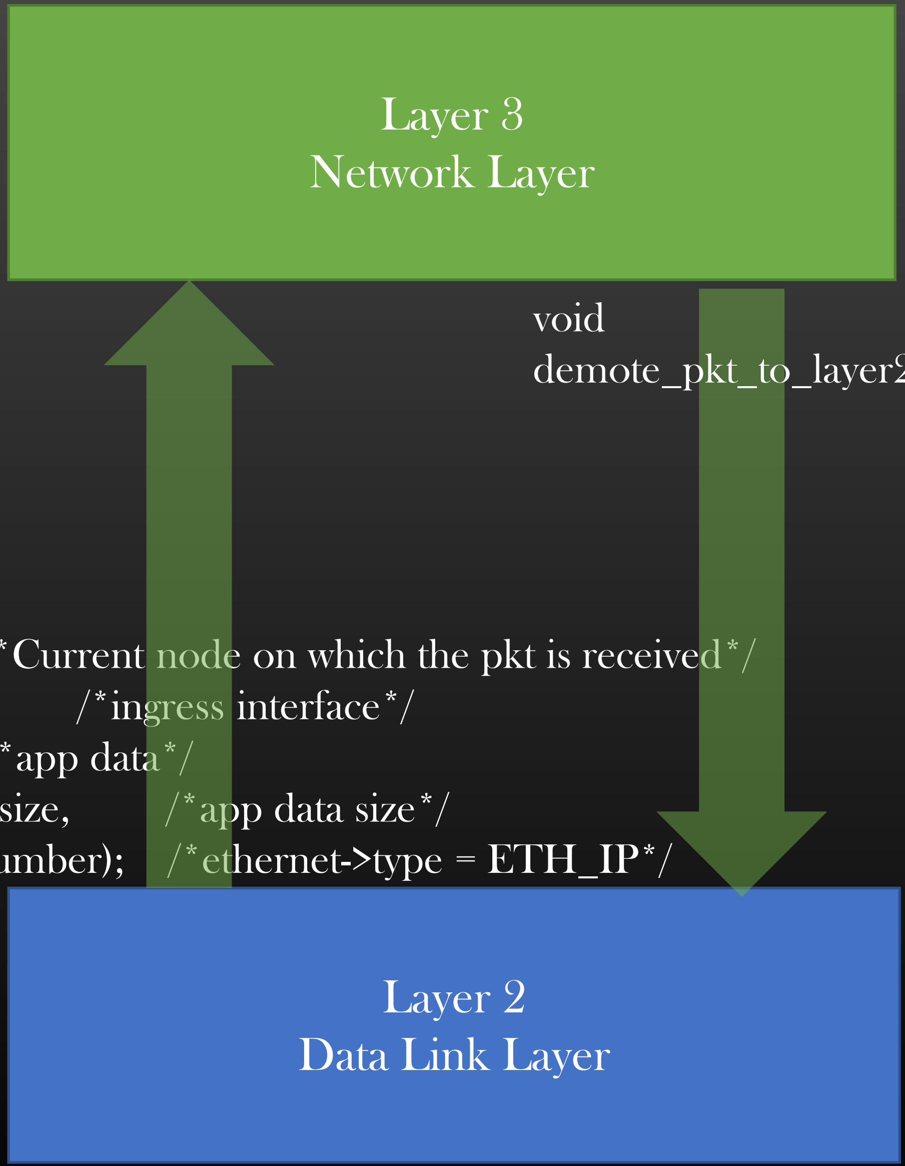
```
void
demote_pkt_to_layer3(node_t *node,         /*Current node
                     char * pkt,                                  /
                     unsigned int pkt_size,                  /*app data s
                     int protocol_number,                    /*L5 protoc
                     unsigned int dest_ip_address); /*dest ip address
```

Layer5/layer5.c/.h

```
void
promote_pkt_to_layer5 (node_t *node,      /*Current node on which the pkt is received*/
                interface_t *recv_intf,              /*ingress interface*/
                char *payload,                  /*app data*/
                        uint32_t app_data_size);      /*app data size*/
```



L3 Routing Implementation -> Phase 6 Network Layer and Data Link Layer Interaction



IP_hdr | app payload

Layer2/layer2.c/.h

```
void
demote_pkt_to_layer2(node_t *node,         /*Current node
                    uint32_t nexthop_ip,                  /*Gateway I
                    char * pkt,
                    unsigned int pkt_size,                 /*app data s
                    int protocol_number);   /*L3 protocol No = ET
```

Layer3/layer3.c/.h

```
void
promote_pkt_to_layer3 (node_t *node,      /*Current node on which the pkt is received*/
                interface_t *recv_intf,              /*ingress interface*/
                char *payload,                  /*app data*/
                        uint32_t app_data_size,        /*app data size*/
                        int L3_protocol_Number);   /*ethernet->type = ETH_IP*/
```

Eth_hdr
payload

L3 Routing Implementation -> Phase 6 Network Layer APIs

Layer3/layer3.c/.h

```
void
promote_pkt_to_layer3 (node_t *node,                    /*Current node on which the pkt is r
                       interface_t *recv_intf,             /*ingress interface*/
                       char *payload,                   /*app data*/
                                  uint32_t app_data_size,       /*app data size*/
                                  int L3_protocol_Number);  /*ethernet->type = ET


void
demote_packet_to_layer3 (node_t *node,
                                char *pkt, unsigned int size,
                                int protocol_number, /*L4 or L5 protocol type*/
                                unsigned int dest_ip_address);
```

L3 Routing Implementation -> Phase 6 Data Link Layer APIs

Layer2/layer2.c/.h

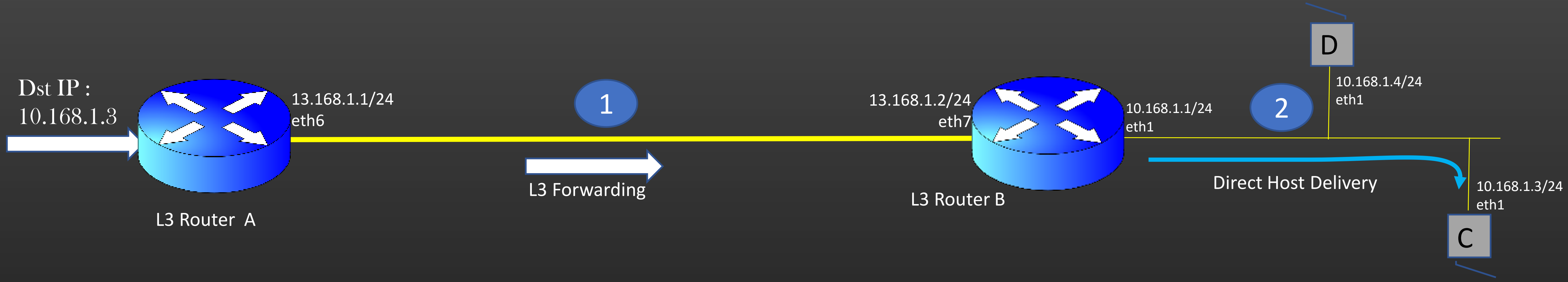
```
void
promote_pkt_to_layer2 (node_t *node,                    /*Current node on which the pkt is r
                       interface_t *recv_intf,          /*ingress interface*/
                       ethernet_hdr_t *ethernet_hdr,
                                uint32_t pkt_size);



void
demote_packet_to_layer2 (node_t *node,
                              unsigned int next_hop_ip,
                              char *outgoing_intf,
                              char *pkt, unsigned int pkt_size,
                                   int protocol_number);
```

L3 Routing Implementation -> Phase 6 Application Layer APIs

Layer5/layer5.c

```
void
promote_pkt_to_layer5 (node_t *node, interface_t *recv_intf,
                                               char *l5_hdr, uint32_t pkt_size,
                                               uint32_t L5_protocol);
```

L3 Routing Implementation -> Phase 7 L3 Routing Revisited



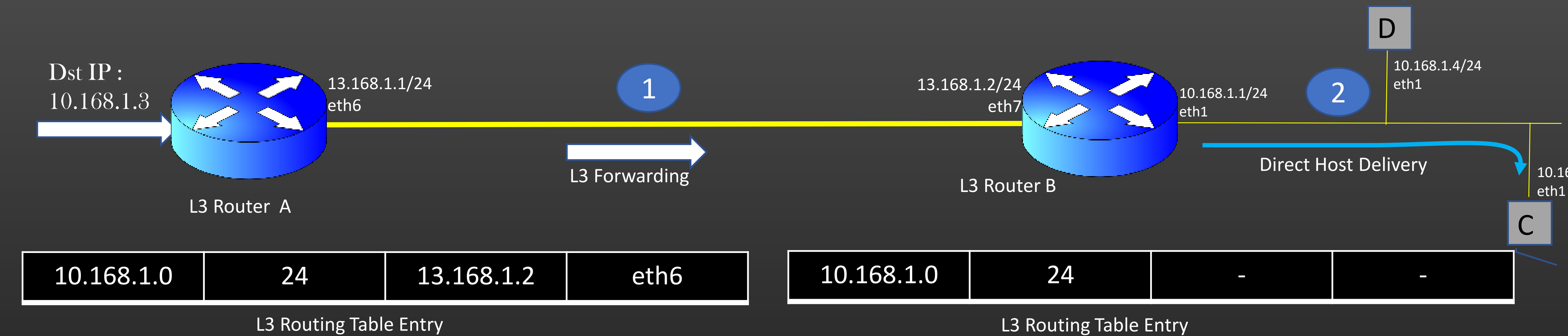
1 Forwarding Case : Router (= A) forwards the frame destined to remote subnet (dest = 10.168.1.3)

2. Direct Host Delivery Case : Router (= B) forwards the frame destined to host present in locally connected subnet (dest = 10.168.1.3)

3 Local Delivery Case : Router (= B) Or Host (= C) receives the pkt destined to itself)

4 Self ping Case : Any L3 device self originate the data destined to itself (dst ip = self loopback or exact match of local interface address)

L3 Routing Implementation -> Phase 7 L3 Routing Revisited



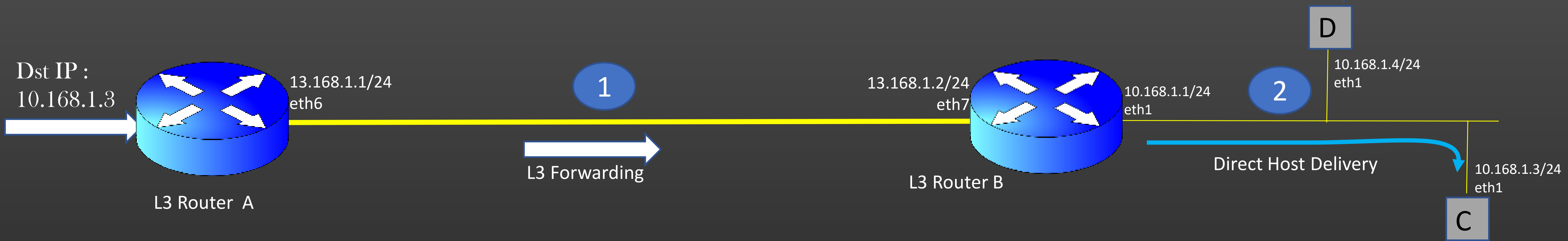
10.168.1.0	24	13.168.1.2	eth6

10.168.1.0	24	-	-

1 Forwarding Case : Steps on A (L3 Forwarding):

1. Frame arrives on L3 interface, Data link layer receives the frame if Dst Mac = IF_MAC
2. Data link layer handover the IP hdr of the frame to Network layer because ethernet_hdr->type = ETH_IP
3. Network Layer inspects the dest ip in ip hdr of the pkt, look up the route in routing table. It comes to know pkt needs to be forwarded out of interface eth6 towards gateway 13.168.1.2
4. Network Layer push the ip hdr down to Data link layer, telling data link layer to resolve ARP for 13.168.1.2 and send out the frame out of interface eth6
5. Data link layer receives the payload (= ip hdr), resolve ARP for 13.168.1.2 if not already, re-attaches ethernet hdr to ip hdr and send out the frame out of interface eth6

L3 Routing Implementation -> Phase 7 L3 Routing Revisited



10.168.1.0	24	13.168.1.2	eth6

L3 Routing Table Entry

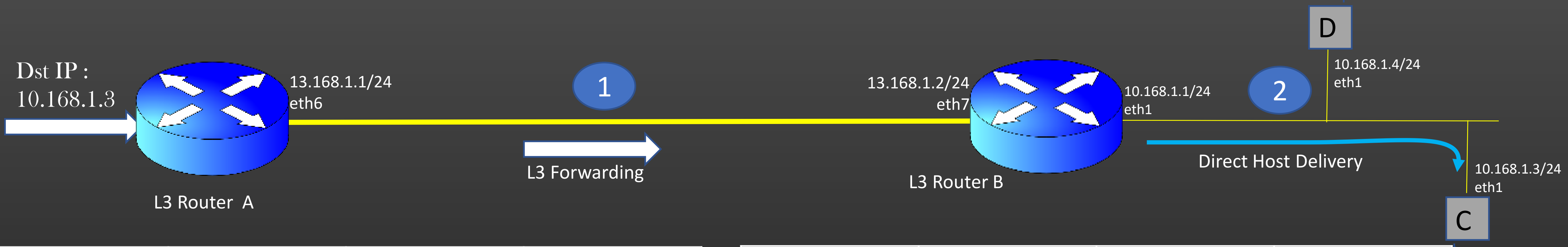
10.168.1.0	24	-	-

L3 Routing Table Entry

2 Direct Host Delivery Case : Steps on B (L2 Routing) :

1. Frame arrives on L3 interface, Data link layer receives the frame if Dst Mac = IF_MAC
2. Data link layer handover the IP hdr of the frame to Network layer because ethernet_hdr->type = ETH_IP
3. Network Layer inspects the dest ip in ip hdr of the pkt, look up the route in routing table. It comes to know that pkt needs to be forwarded to host machine present in its directly connected subnet
4. Network Layer push the ip hdr down to Data link layer, telling data link layer to resolve ARP for Dst = 10.168.1.3
5. Data link layer receives the payload (= ip hdr), resolve ARP for 10.168.1.3 on matching subnet interface if not already, re-attaches ethernet hdr to ip hdr and send out the frame on which ARP reply was received (i.e eth1)

L3 Routing Implementation -> Phase 7 L3 Routing Revisited



10.168.1.0	24	13.168.1.2	eth6

L3 Routing Table Entry

10.168.1.0	24	-	-

L3 Routing Table Entry

```
void
demote_pkt_to_layer2(A,
                     13.168.1.2,
                     eth6,
          char *payload,
                     uint32_t payload_size,
                     ETH_IP);
```

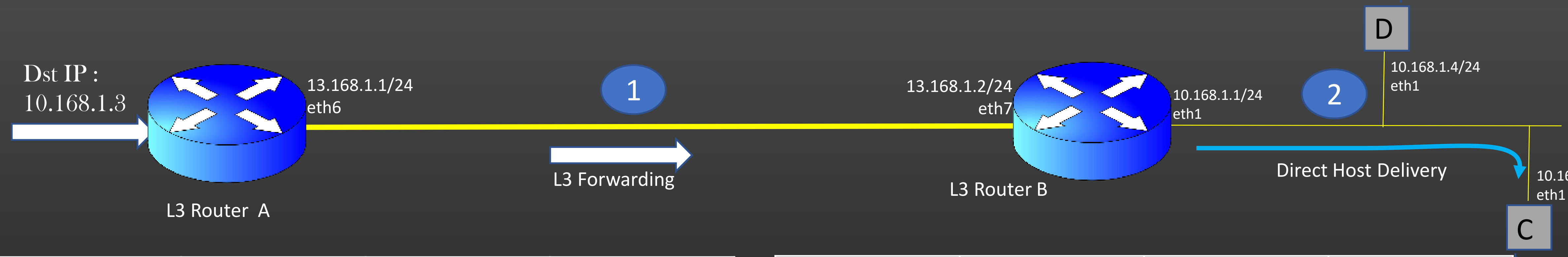
Router A (Forwarding Case)	Router B (Direct Host Delivery Case)
Router finds the L3 routing table entry pointing to remote subnet	Router finds the L3 routing table entry pointing to local subnet
L3 tells L2 to resolve ARP for Gateway IP	L3 tells L2 to resolve ARP for Destination
L3 tells L2 the OIF to resolve ARP	L3 do not tell L2 the OIF, L2 figures it based on matching subnet interface

```
void
demote_pkt_to_layer2(A,
                     10.168.1.3,
                     NULL,
          char *payload,
                     uint32_t pay
                     ETH_IP);
```

net.c/.h

```
interface_t *
node_get_matching_subnet_interface(node_t *node, char *ip_addr);
```

L3 Routing Implementation -> Phase 7 L3 Routing Revisited



10.168.1.0	24	13.168.1.2	eth6

L3 Routing Table Entry

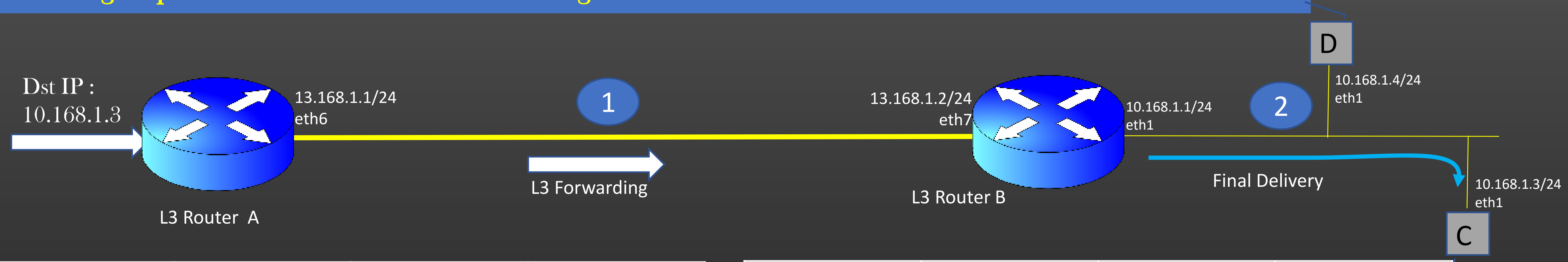
10.168.1.0	24	-	-

L3 Routing Table Entry

3 Local Delivery Case : Steps on C

1. Data link layer handover the IP hdr of received frame to Network Layer
2. Network Layer checks if dst ip = IP of any local interface OR self-loopback then, deliver the IP payload to higher layers based on *protocol field value*

L3 Routing Implementation -> Phase 7 L3 Routing Revisited

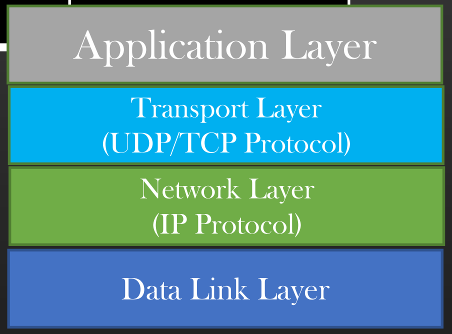


10.168.1.0	24	13.168.1.2	eth6

L3 Routing Table Entry

10.168.1.0	24	-	

L3 Routing Table Entry

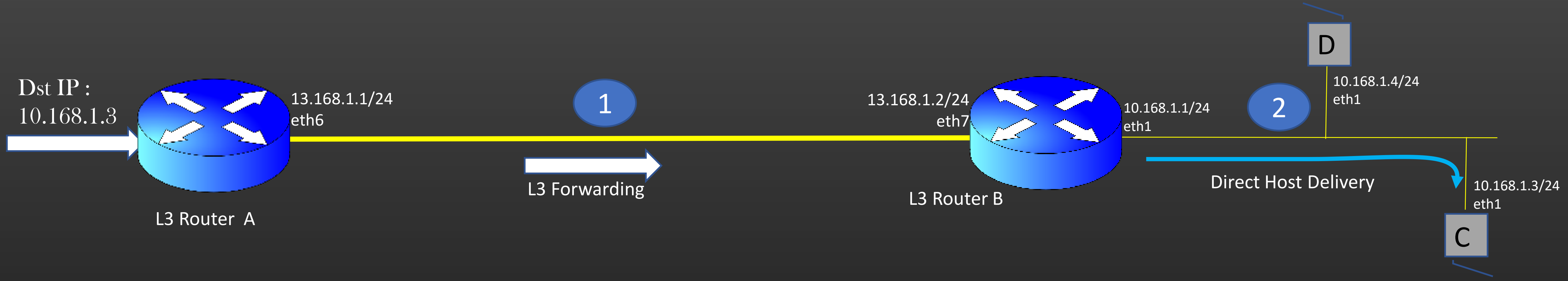


4 Self ping Case : Steps on A (self ping)

1. Network Layer receives the ping request from Higher Layer. Here Dest ip = Local interface IP Or Self-Looback address
2. Network Layer looks-up in routing table for destination ip, it finds a local route
3. Network Layer handover the request to Layer 2 as per *Direct host Deliver case*
4. Data link layer checks, if dest-ip = exact match of any local interface Or loopback address, bounce the pkt back to network layer
5. Network Layer exercise *local delivery case*

The intent is when routing device pings it-self, exercise the functionality of all the layers of TCP-IP stack – Ingress and Egress

L3 Routing Implementation -> Phase 8 L3 Routing State Machine



1 Forwarding Case : Router (= A) forwards the frame destined to remote subnet (dest = 10.168.1.3)

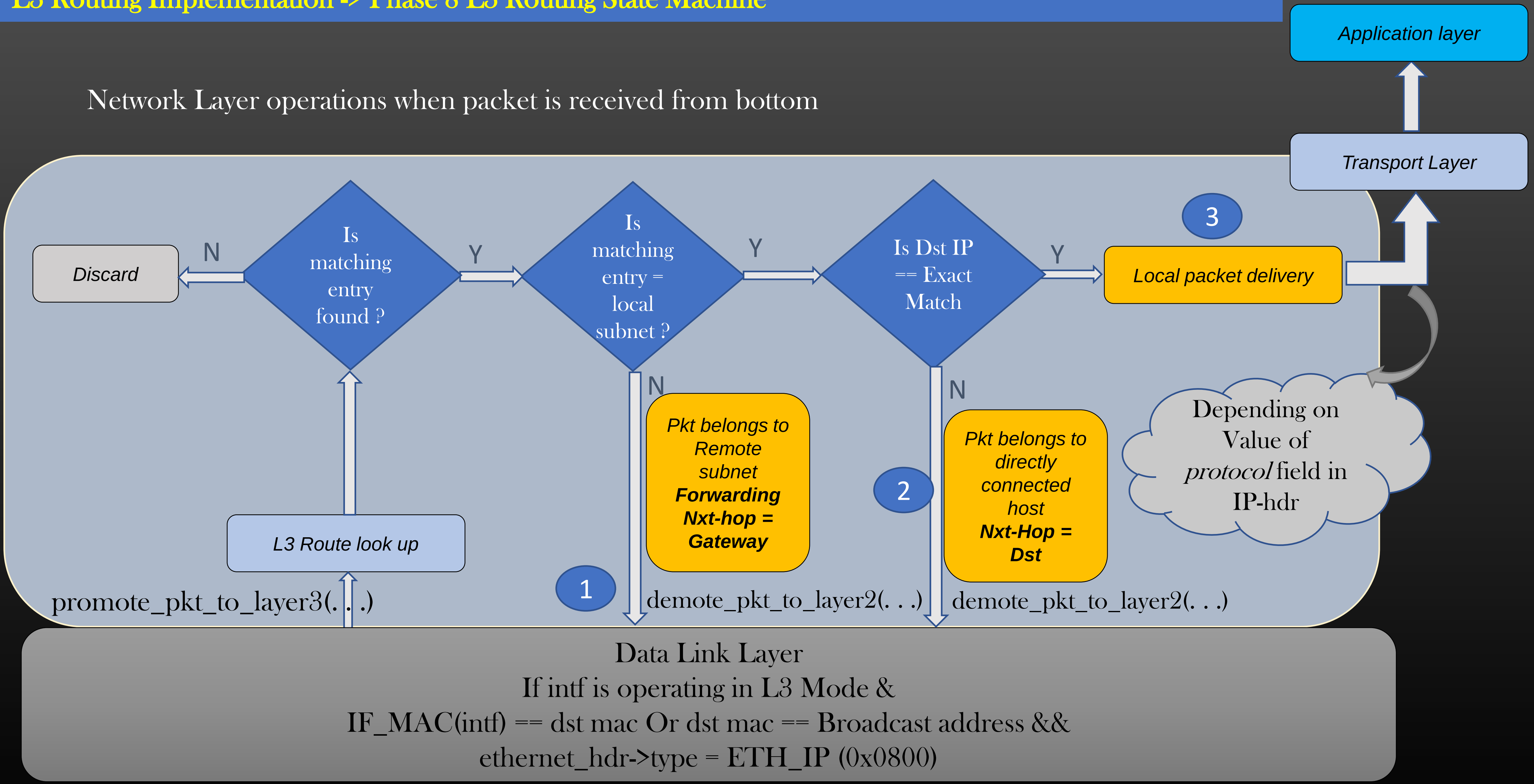
2. Direct Host Delivery Case : Router (= B) forwards the frame destined to host present in locally connected subnet (dest = 10.168.1.3)

3 Local Delivery Case : Router (= B) Or Host (= C) receives the pkt destined to itself)

4 Self ping Case : Any L3 device self originate the data destined to itself (dst ip = self loopback or exact match of local interface address)

L3 Routing Implementation -> Phase 8 L3 Routing State Machine

Network Layer operations when packet is received from bottom

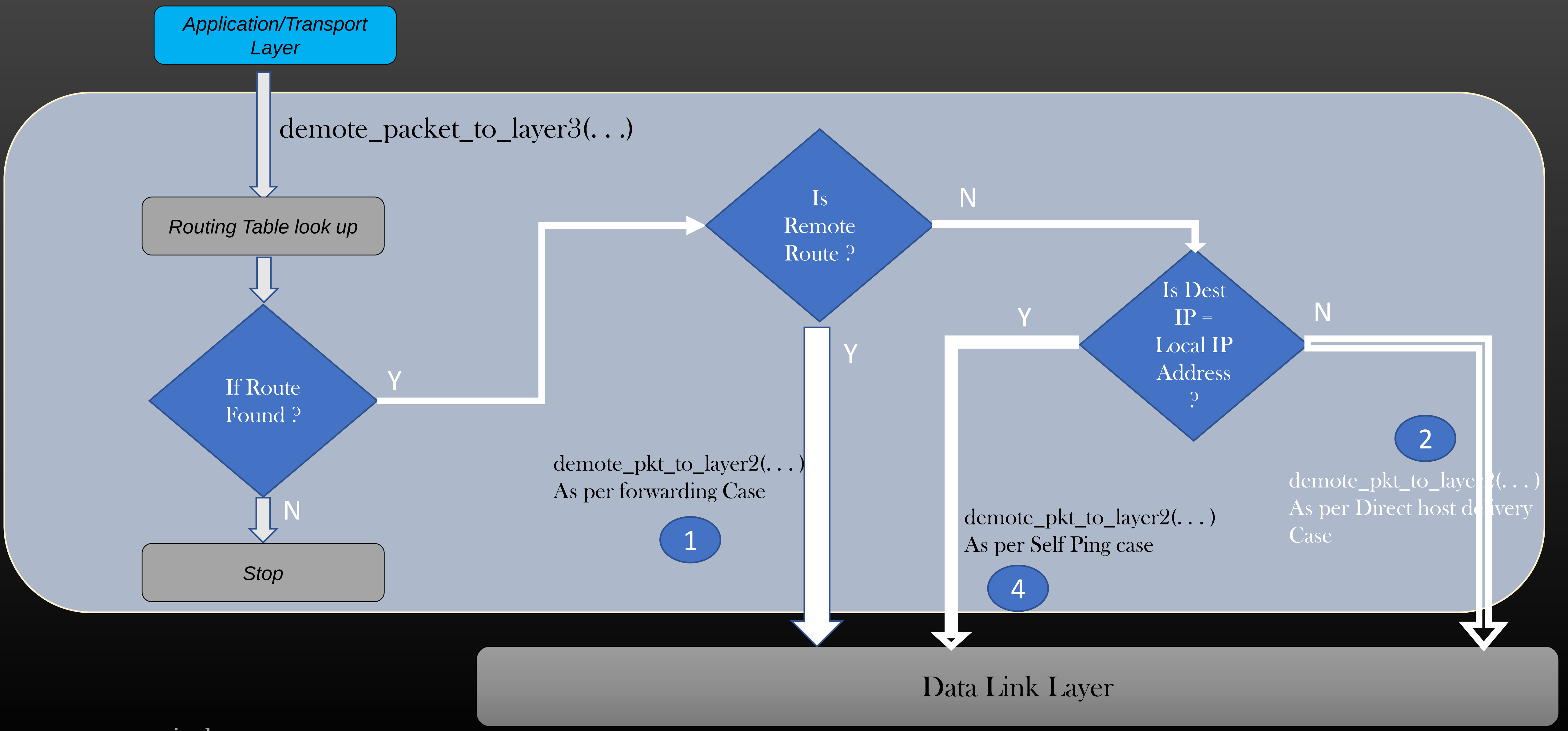


Application layer
Transport Layer
Discard
N
Is matching entry found ?
Y
Is matching entry = local subnet ?
Y
Is Dst IP == Exact Match
Y
3
Local packet delivery
N
Pkt belongs to Remote subnet Forwarding Nxt-hop = Gateway
N
2
Pkt belongs to directly connected host Nxt-Hop = Dst
Depending on Value of protocol field in IP-hdr
L3 Route look up
1
promote_pkt_to_layer3(. . .)
demote_pkt_to_layer2(. . .)
demote_pkt_to_layer2(. . .)
Data Link Layer
If intf is operating in L3 Mode &
IF_MAC(intf) == dst mac Or dst mac == Broadcast address &&
ethernet_hdr->type = ETH_IP (0x0800)

www.csepracticals.com

L3 Routing Implementation -> Phase 8 L3 Routing State Machine

Network Layer operations when packet is received from top


Application/Transport Layer
demote_packet_to_layer3(. . .)
Routing Table look up
If Route Found ?
Y
N
Stop
Is Remote Route ?
N
Y
Is Dest IP = Local IP Address ?
Y
N
demote_pkt_to_layer2(. . .)
As per forwarding Case
1
demote_pkt_to_layer2(. . .)
As per Self Ping case
4
2
demote_pkt_to_layer2(. . .)
As per Direct host delivery Case
Data Link Layer

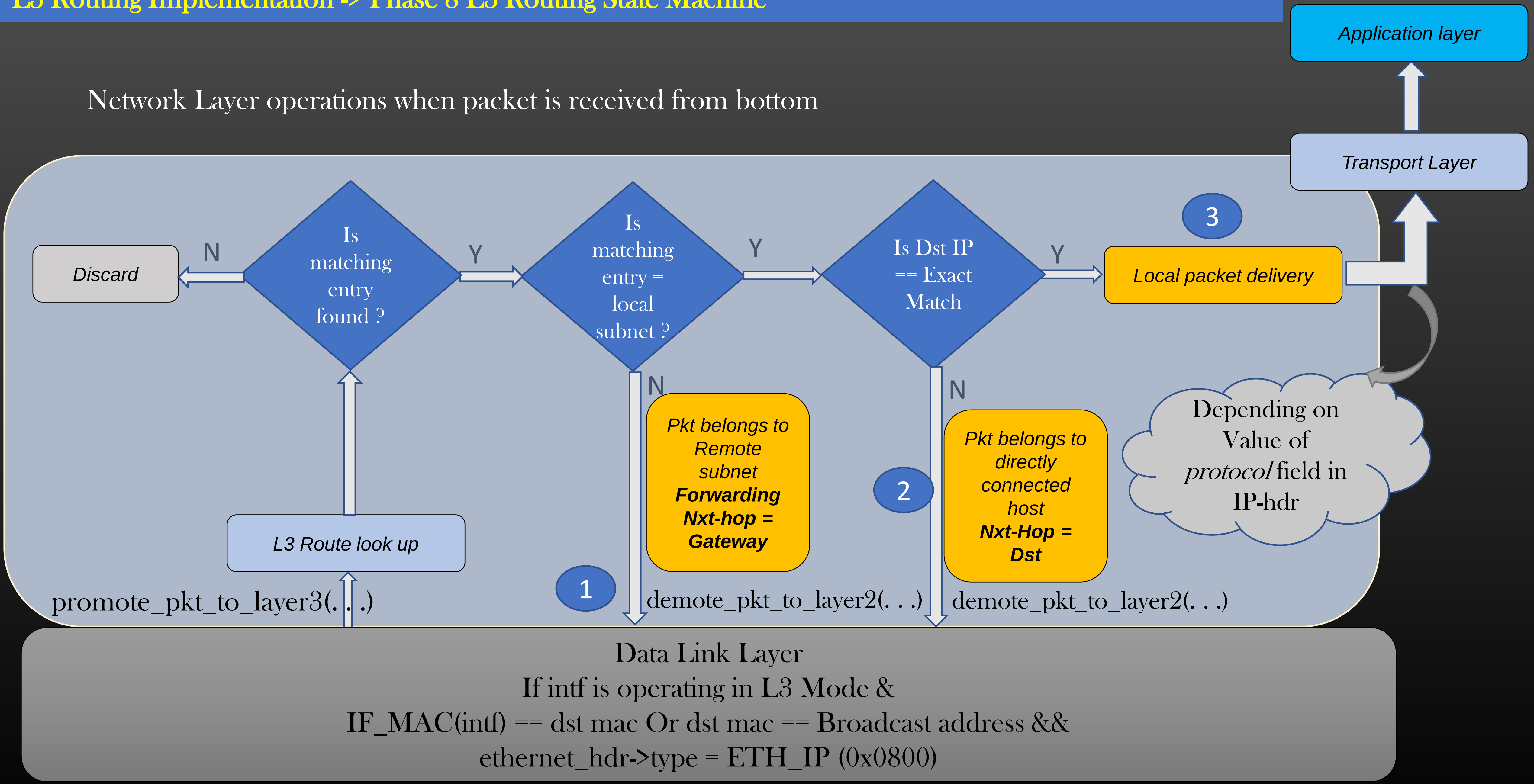
L3 Routing Implementation -> Phase 8 L3 Routing State Machine

Time to Code Network Layer Operations !!

- Pls feel free to insert as many debugging printfs as you want to triage the issues . . .
- Just follow the flow-charts and everything shall fall in place ☺

L3 Routing Implementation -> Phase 8 L3 Routing State Machine

Network Layer operations when packet is received from bottom

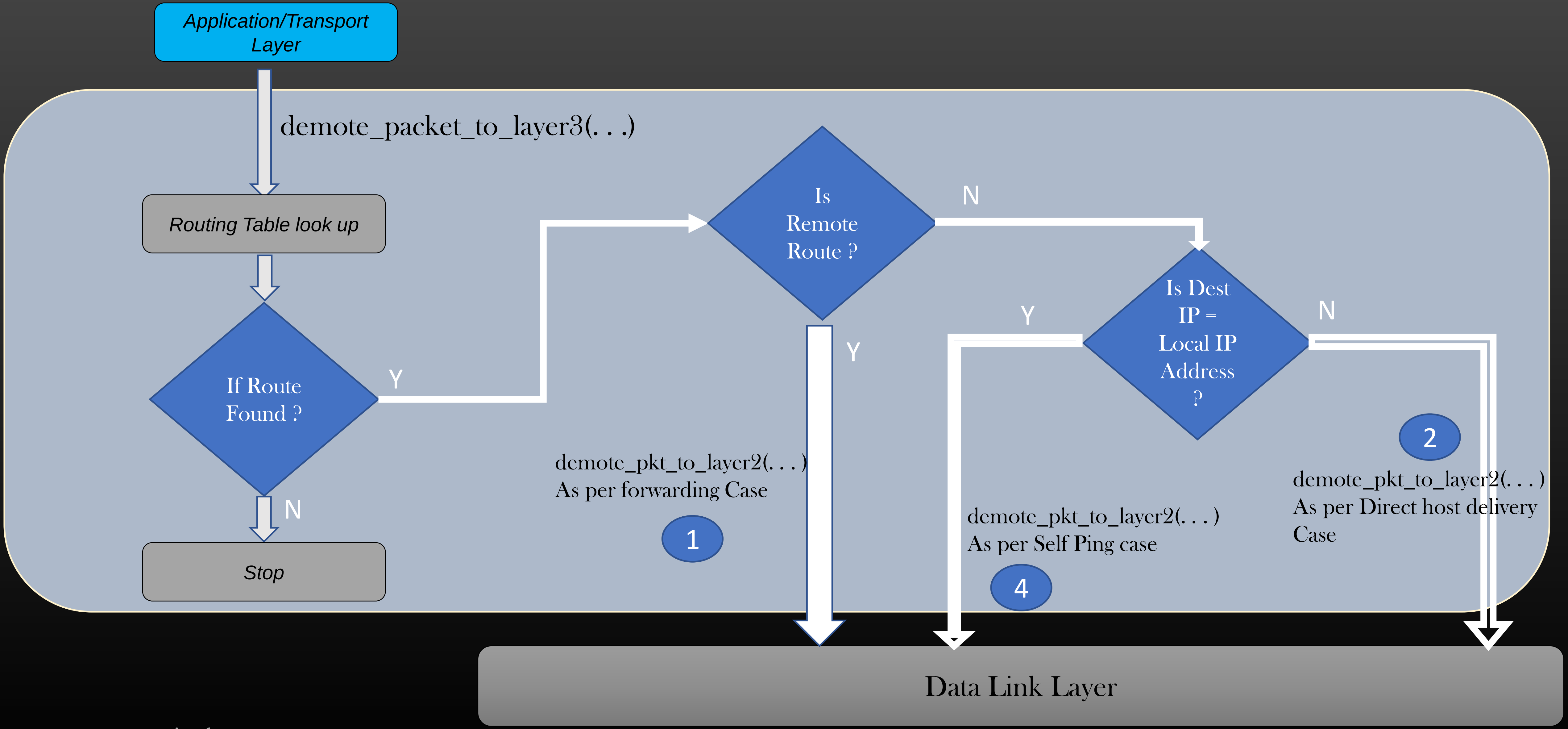


Application layer
Transport Layer
Discard
N
Is matching entry found ?
Y
Is matching entry = local subnet ?
Y
Is Dst IP == Exact Match
Y
3
Local packet delivery
N
N
Pkt belongs to Remote subnet Forwarding Nxt-hop = Gateway
2
Pkt belongs to directly connected host Nxt-Hop = Dst
Depending on Value of *protocol* field in IP-hdr
L3 Route look up
1
promote_pkt_to_layer3(. . .)
demote_pkt_to_layer2(. . .)
demote_pkt_to_layer2(. . .)

Data Link Layer
If intf is operating in L3 Mode &
IF_MAC(intf) == dst mac Or dst mac == Broadcast address &&
ethernet_hdr->type = ETH_IP (0x0800)

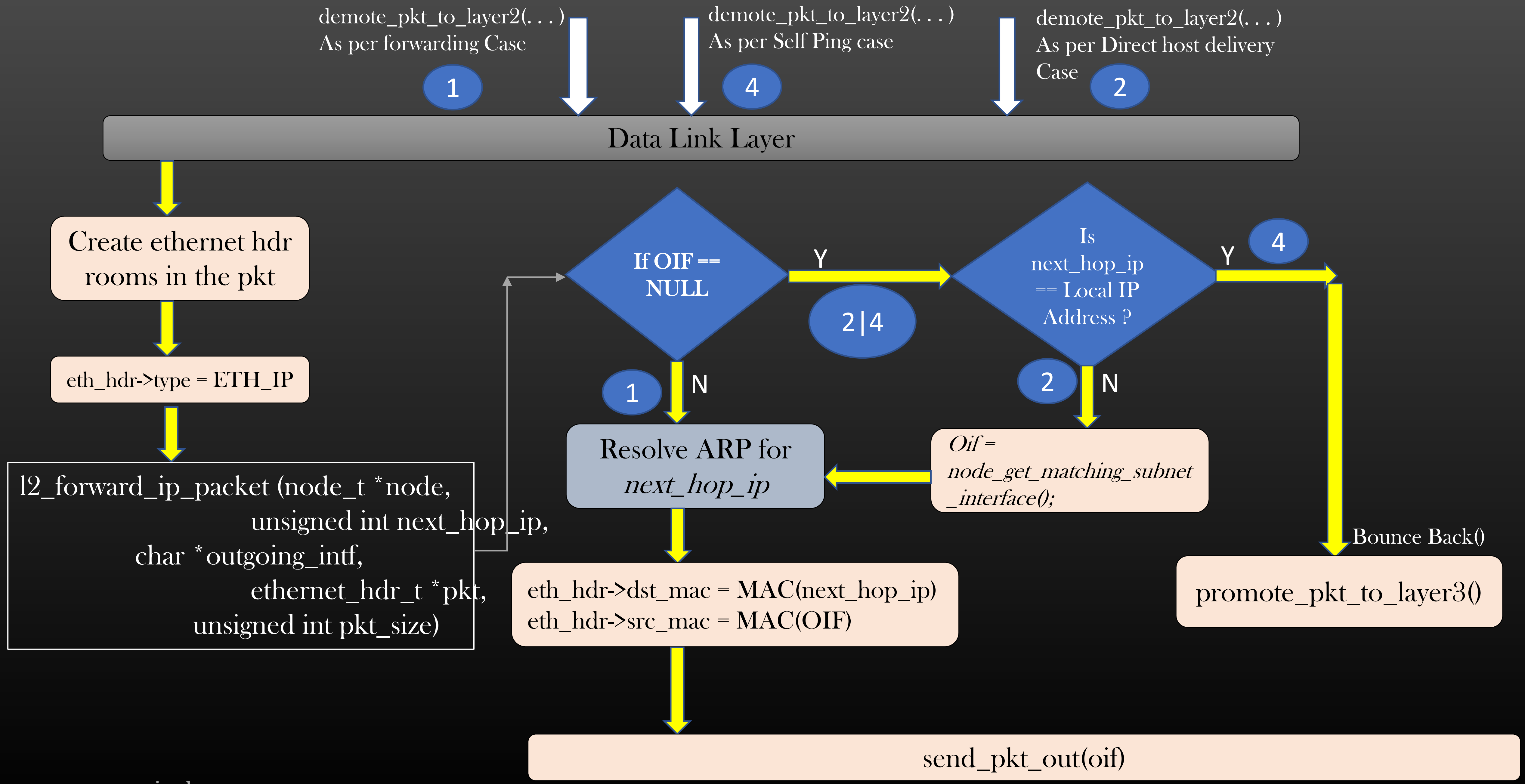
L3 Routing Implementation -> Phase 8 L3 Routing State Machine

Network Layer operations when packet is received from top



Application/Transport Layer
demote_packet_to_layer3(. . .)
Routing Table look up
If Route Found ?
Y
N
Stop
Is Remote Route ?
N
Y
Is Dest IP = Local IP Address ?
Y
N
demote_pkt_to_layer2(. . .)
As per forwarding Case
1
demote_pkt_to_layer2(. . .)
As per Self Ping case
4
2
demote_pkt_to_layer2(. . .)
As per Direct host delivery Case
Data Link Layer

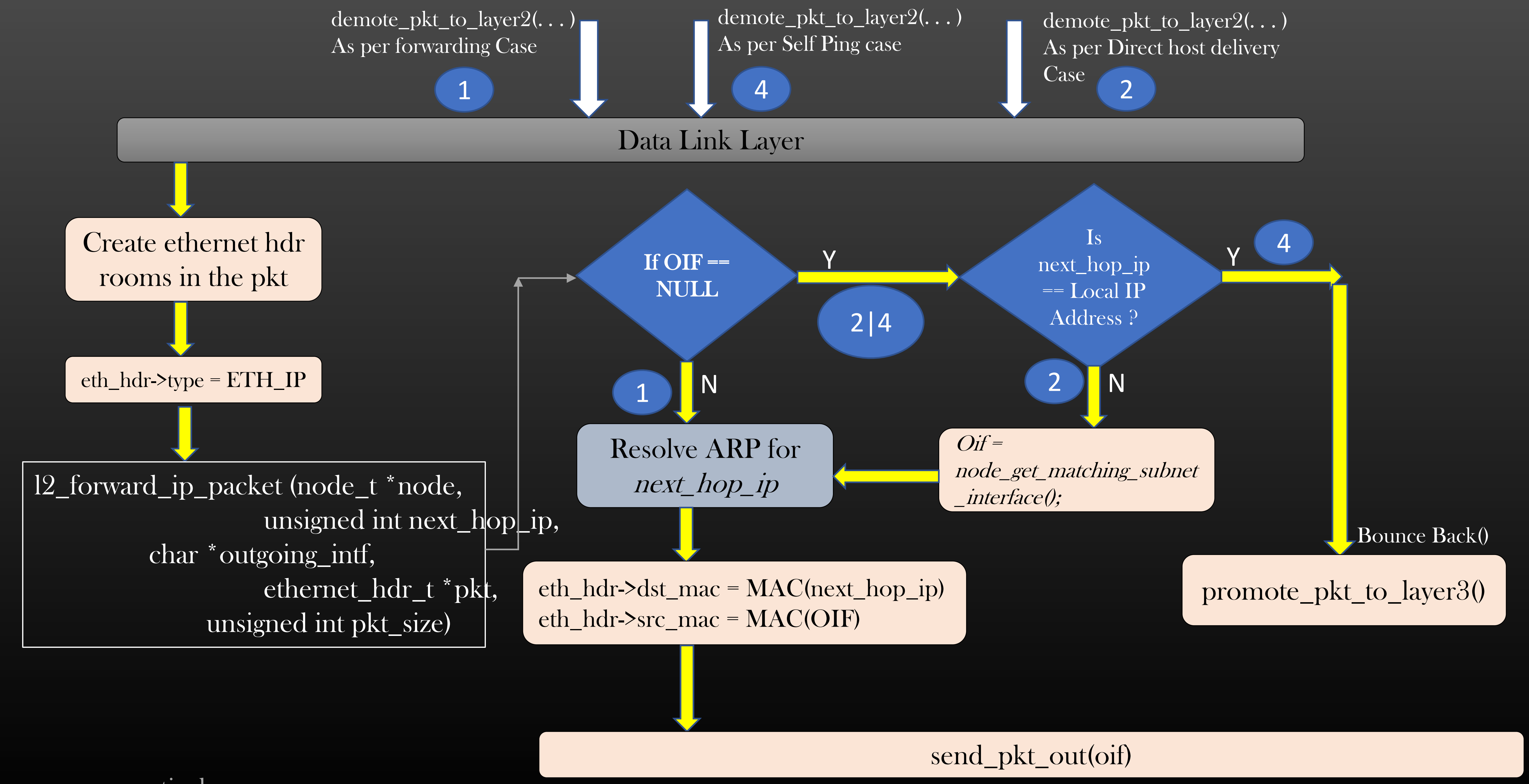
L2 Routing Implementation



demote_pkt_to_layer2(. . .)
As per forwarding Case
1
demote_pkt_to_layer2(. . .)
As per Self Ping case
4
demote_pkt_to_layer2(. . .)
As per Direct host delivery Case
2
Data Link Layer
Create ethernet hdr rooms in the pkt
eth_hdr->type = ETH_IP
If OIF == NULL
Y
2|4
Is next_hop_ip == Local IP Address ?
Y
4
N
1
N
2
Resolve ARP for *next_hop_ip*
Oif = node_get_matching_subnet_interface();
Bounce Back()
promote_pkt_to_layer3()
eth_hdr->dst_mac = MAC(next_hop_ip)
eth_hdr->src_mac = MAC(OIF)
send_pkt_out(oif)

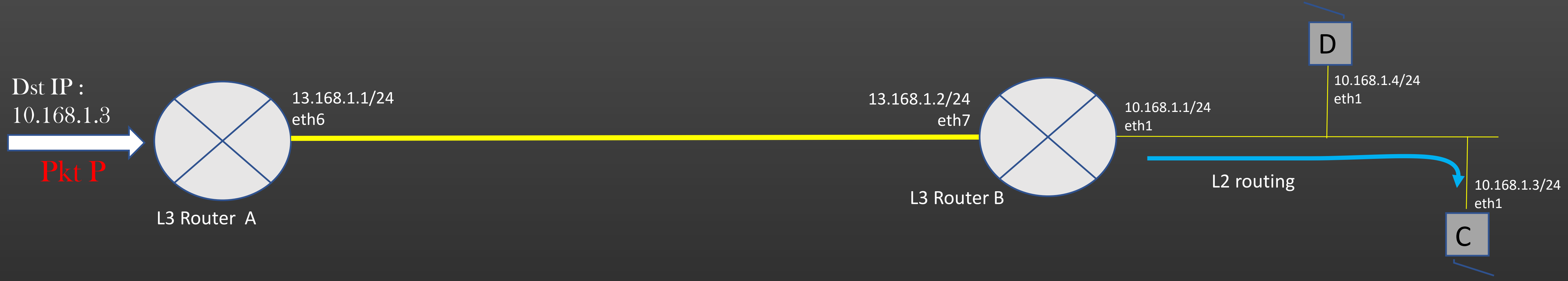
```
l2_forward_ip_packet (node_t *node,
                      unsigned int next_hop_ip,
               char *outgoing_intf,
                      ethernet_hdr_t *pkt,
                   unsigned int pkt_size)
```

L2 Routing Implementation


demote_pkt_to_layer2(. . .)
As per forwarding Case
1
demote_pkt_to_layer2(. . .)
As per Self Ping case
4
demote_pkt_to_layer2(. . .)
As per Direct host delivery Case
2
Data Link Layer
Create ethernet hdr rooms in the pkt
eth_hdr->type = ETH_IP
If OIF == NULL
Y
2|4
Is next_hop_ip == Local IP Address ?
Y
4
N
1
N
2
Resolve ARP for next_hop_ip
Oif = node_get_matching_subnet_interface();
Bounce Back()
promote_pkt_to_layer3()
eth_hdr->dst_mac = MAC(next_hop_ip)
eth_hdr->src_mac = MAC(OIF)
send_pkt_out(oif)

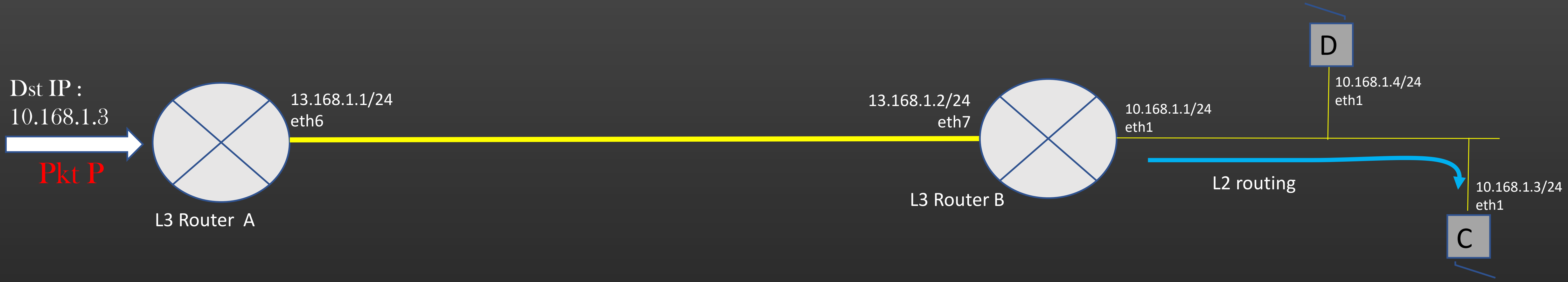
```
l2_forward_ip_packet (node_t *node,
                      unsigned int next_hop_ip,
          char *outgoing_intf,
                      ethernet_hdr_t *pkt,
                  unsigned int pkt_size)
```

ARP On Demand



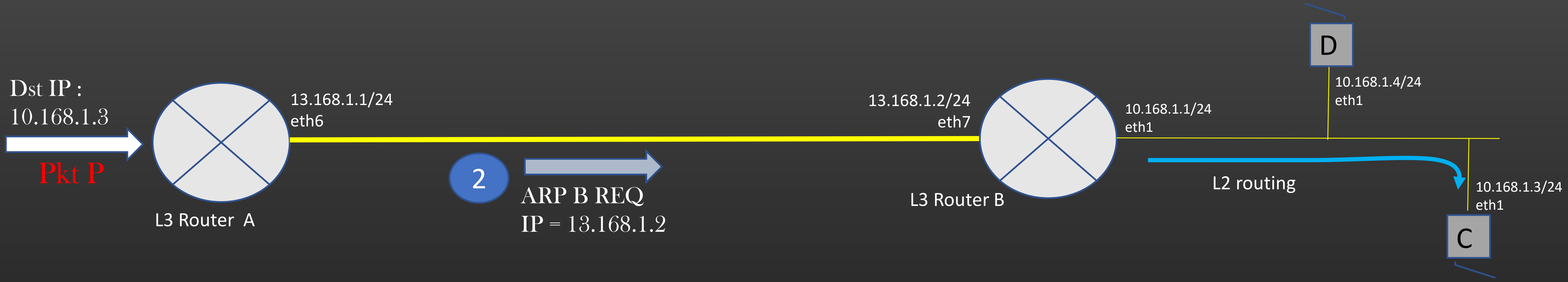
- As Soon as the Router A decides to L3 forward the Pkt P to Nexthop 13.168.1.2 out of interface eth6, it need to update the Ethernet hdr of the frame
- Suppose Router A do not have IP ← → MAC mapping in its ARP table for IP = 13.168.1.2
- Router A launch ARP Broadcast request on interface eth6 and wait for ARP reply
  - Question : What will Router A do until ARP is resolved ?
  - There could be other packets in Queue waiting to be treated by Router A
  - Router A cannot wait !
- Solution :
  - Router A temporarily stores the Pkt P, and get busy processing other incoming packets
  - As soon as ARP resolution is done, Pkt P 's ethernet hdr is updated and forwarded

ARP On Demand



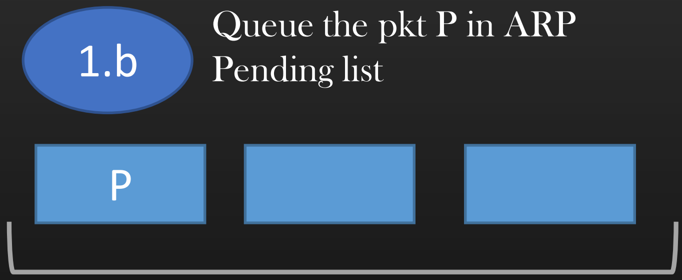
IP	MAC	Is_sane	OIF

ARP On Demand



1.a

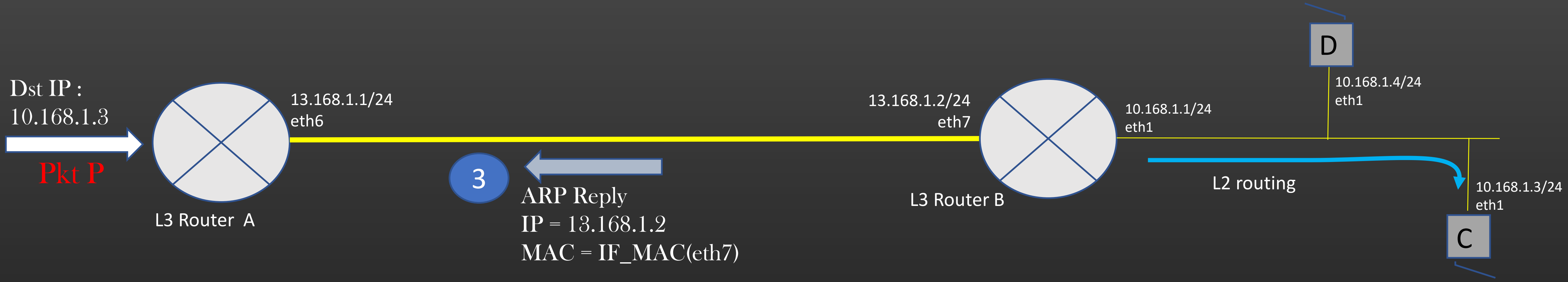
IP	MAC	Is_sane	OIF
13.168.1.2	?	Yes	?



ARP Pending list
Pks with empty Ethernet hdrs

- Each entry in ARP table maintains a list of packets whose ethernet hdr is incomplete
- These packets are awaiting ARP resolution

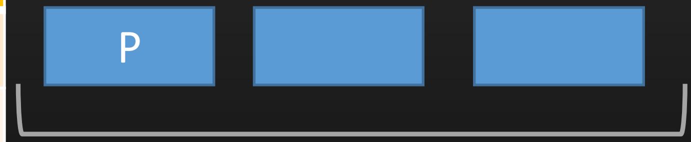
ARP On Demand



1.a

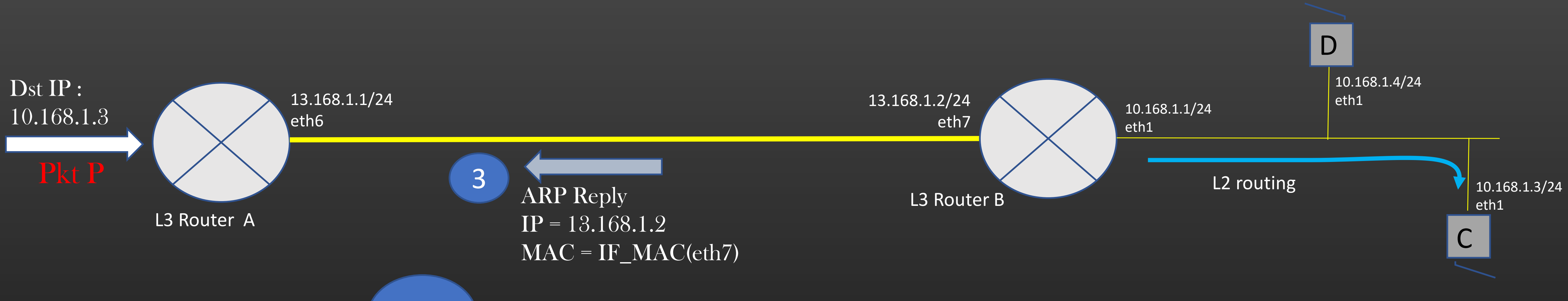
IP	MAC	Is_sane	OIF
13.168.1.2	?	Yes	?

1.b Queue the pkt P in ARP Pending list



ARP Pending list
Pks with empty Ethernet hdrs

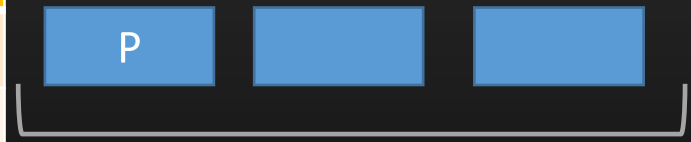
ARP On Demand



4.a Update ARP sane entry

IP	MAC	Is_sane	OIF
13.168.1.2	IF_MAC(eth7)	No	eth6

4.b Process ARP pending list



ARP Pending list
Pks with empty Ethernet hdrs

- Process ARP pending list : Update ethernet hdr of each pkt in ARP pending list and dispatch

ARP On Demand

Data Structure Changes :
Layer2/layer2.h/.c

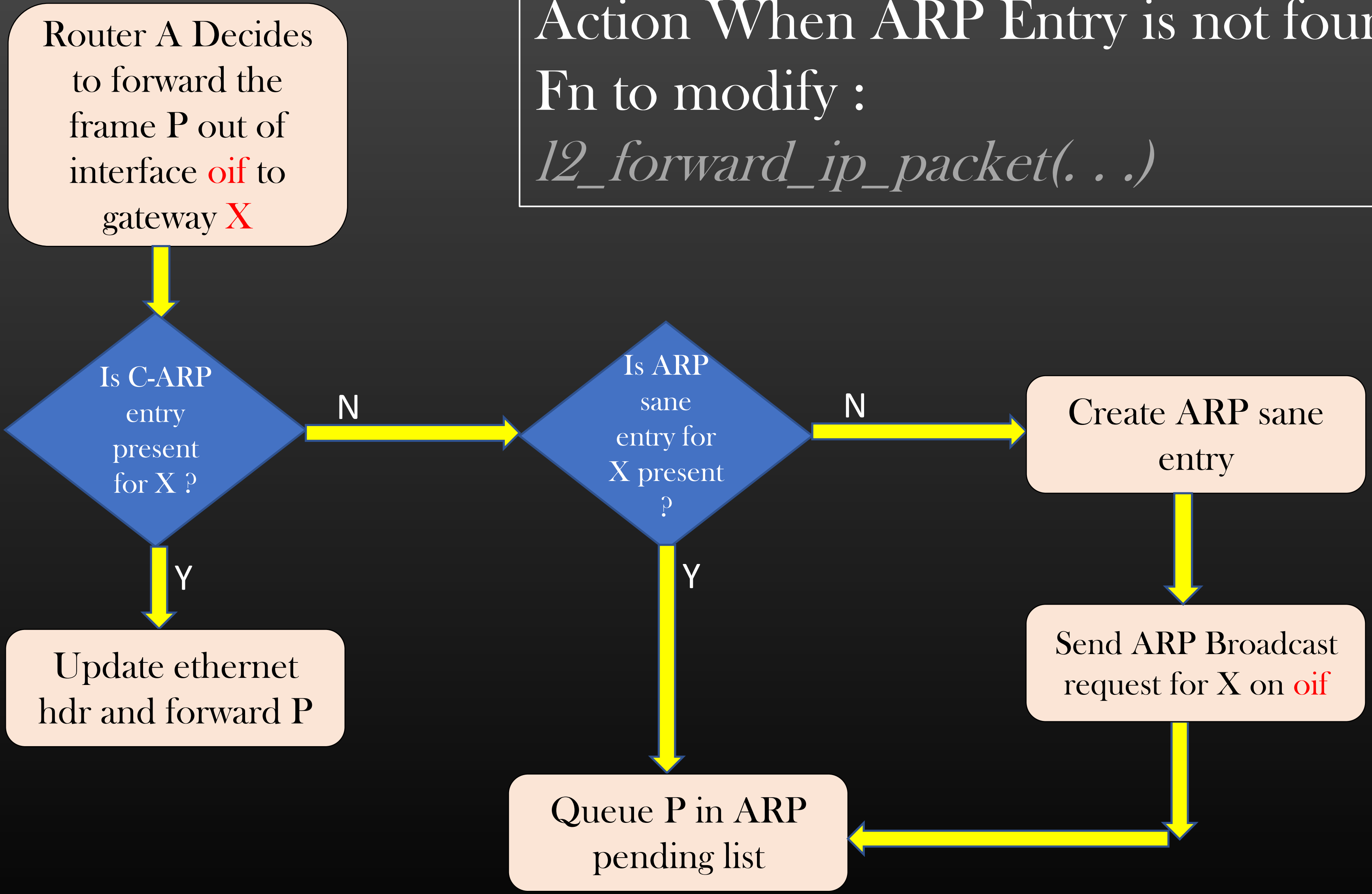
```
struct arp_entry_{

    ip_add_t ip_addr;              /*key*/
    mac_add_t mac_addr;
    char oif_name[IF_NAME_SIZE];
    glthread_t arp_glue;

    bool_t is_sane;

    /* List of packets which are pending for
     * this ARP resolution*/
    glthread_t arp_pending_list;   /*Linked List head*/
};
```

```
struct arp_pending_entry_{

    glthread_t arp_pending_entry_glue;
    arp_processing_fn cb;
    uint32_t pkt_size;    /*Including ethernet hdr*/
    char pkt[0];
};
```

```
/*ARP pending list processing fn Signature*/

typedef void (*arp_processing_fn)(node_t *,
                                  interface_t *oif,
                                  arp_entry_t *,
                                  arp_pending_entry_t *);
```

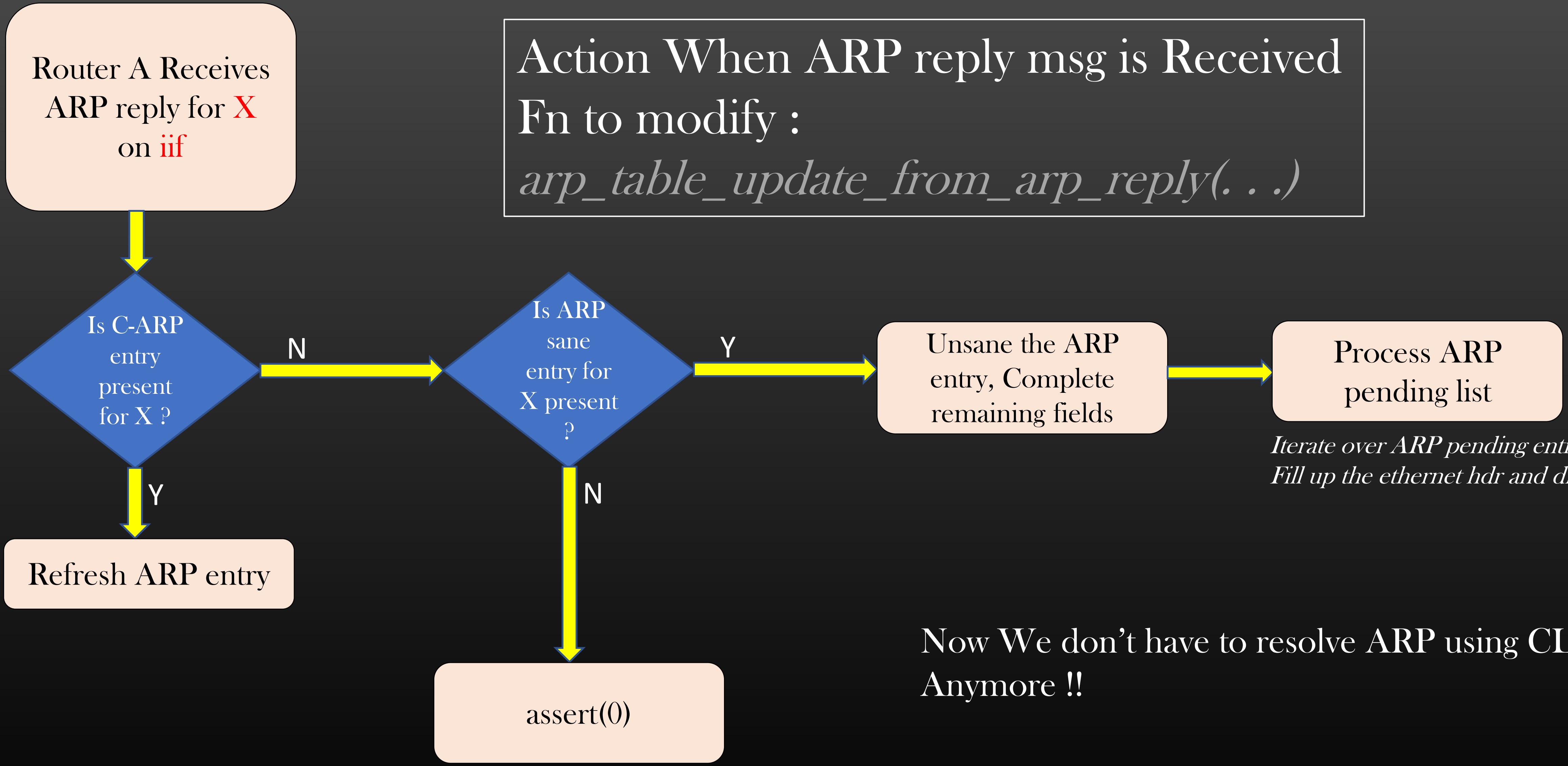
ARP On Demand

Action When ARP Entry is not found
Fn to modify :
l2_forward_ip_packet(. . .)


Router A Decides to forward the frame P out of interface oif to gateway X
Is C-ARP entry present for X ?
N
Y
Update ethernet hdr and forward P
Is ARP sane entry for X present ?
N
Y
Create ARP sane entry
Send ARP Broadcast request for X on oif
Queue P in ARP pending list

ARP On Demand

Action When ARP reply msg is Received
Fn to modify :
arp_table_update_from_arp_reply(. . .)



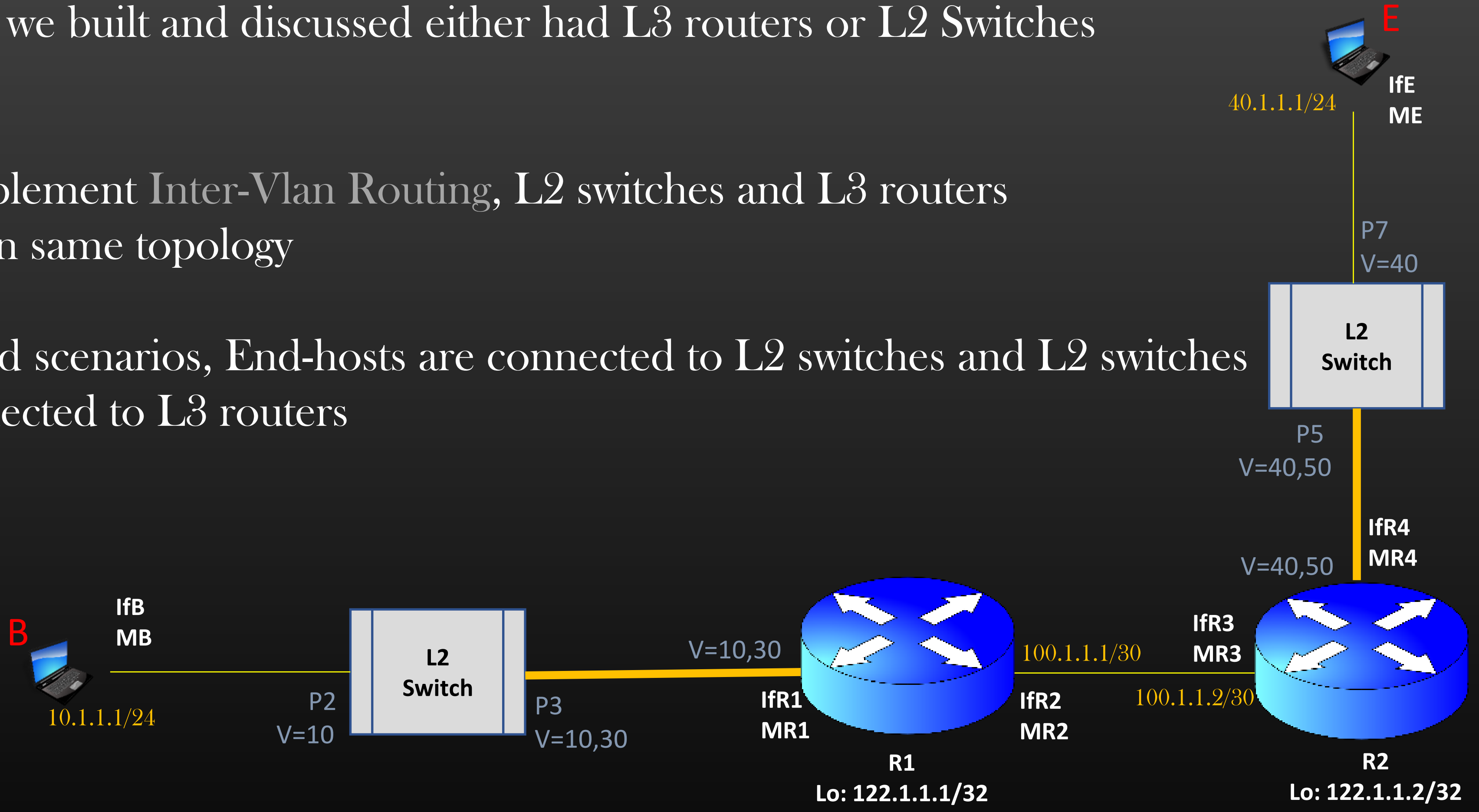
Router A Receives ARP reply for X on iif
Is C-ARP entry present for X ?
N
Y
Refresh ARP entry
Is ARP sane entry for X present ?
Y
N
assert(0)
Unsane the ARP entry, Complete remaining fields
Process ARP pending list

Iterate over ARP pending entries,
Fill up the ethernet hdr and dispatch

Now We don't have to resolve ARP using CLI Anymore !!

Your Home Work !!

- Do your homework !!
- All topologies we built and discussed either had L3 routers or L2 Switches but not both
- Unless we implement Inter-Vlan Routing, L2 switches and L3 routers cannot co-exist in same topology
- In Real-World scenarios, End-hosts are connected to L2 switches and L2 switches In-turn are connected to L3 routers



IP-In-IP Encapsulation (Tunneling)

Ip-In-IP Encapsulation

- Now that we have implemented our beloved TCP/IP Stack, let us implement some networking mini projects or concepts on top of it.

- We will implement IP-In-IP Encapsulation

- You would not have to write more than 100 LOC

- Theory reference :
  - Appendix G

- In the remaining part of this section, I assume you have understood the IP-in-IP encapsulation concept, and ready to implement it

- Advice : Try implementing it yourself and refer to this section for help/reference. Learn to consume the remaining portion of this course as a reference material

- Code access for this section : Refer to description of the lecture

Ip-In-IP Encapsulation

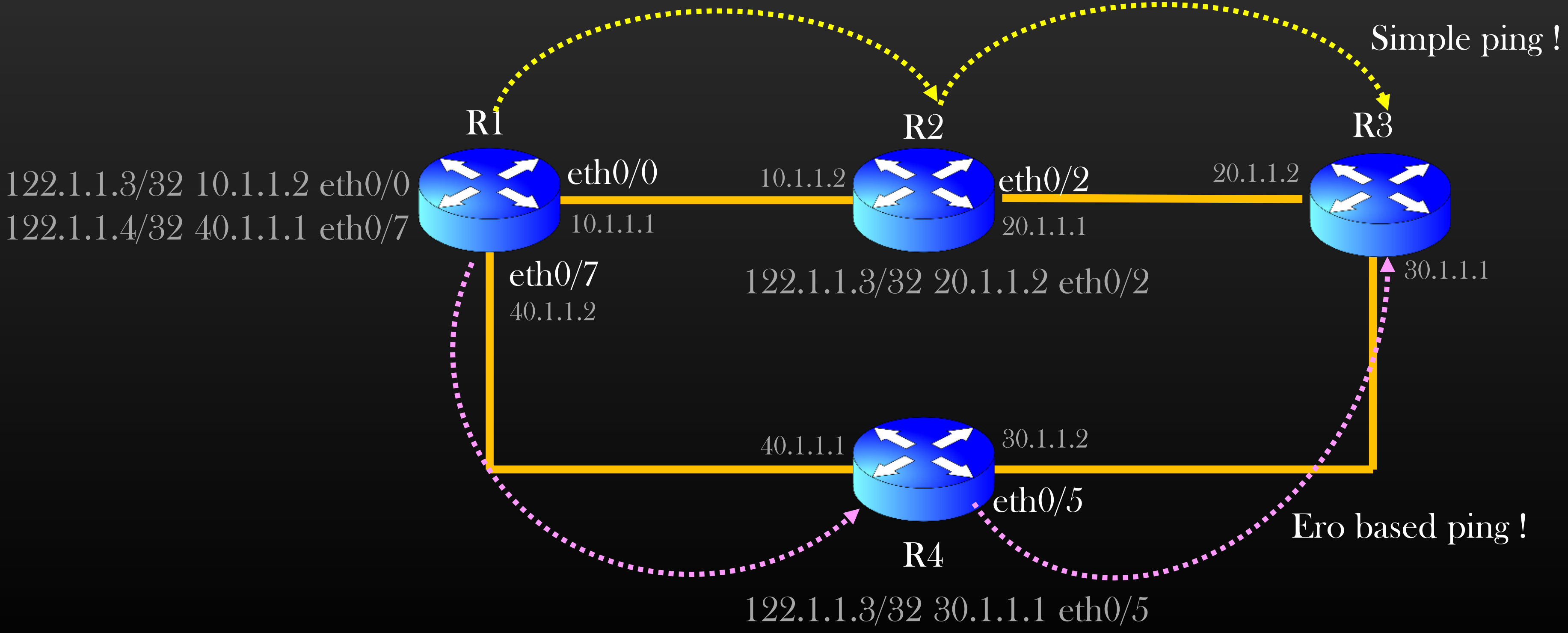
- Implementation Steps :

Step 1/3 : CLI

run node <node-name> ping <ip-address> ero <ero-ip-address>

For example : run node R1 ping 122.1.1.3 ero 122.1.1.4

This CLI mean : R1 is sending ping packet to router R3 whose ip address is 122.1.1.3 but ping packet must go through router whose ip address 122.1.1.4



Ip-In-IP Encapsulation

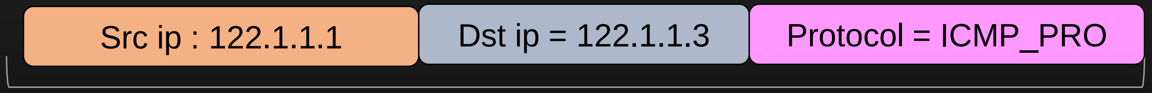
- Implementation Steps :

Step 2/3 : Backend Handler

In Layer5/ping.c

```
void
layer3_ero_ping_fn (node_t *node, char *dst_ip_addr,
                                             char *ero_ip_address);
```

- This fn must do two tasks :
  - prepare the IP Hdr without Application payload. This IP Hdr shall be inner IP hdr of ip-in-ip packet



Generated by CLI

- Call :

```
demote_packet_to_layer3 (node, (char *) inner_ip_hdr,
                                        inner_ip_hdr->total_length * 4,
                                               IP_IN_IP, ero_ip_addr_int );
```

(Notice : Inner IP hdr just acts as a payload to Network Layer)

Ip-In-IP Encapsulation

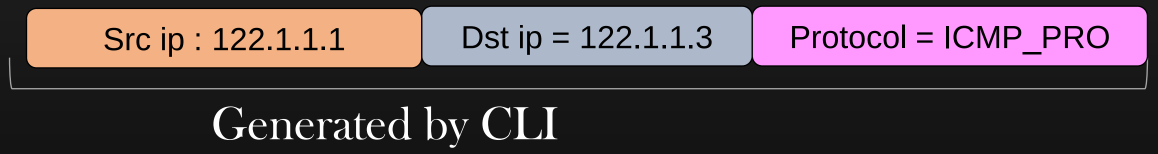
- Implementation Steps :

Step 2/3 : Backend Handler

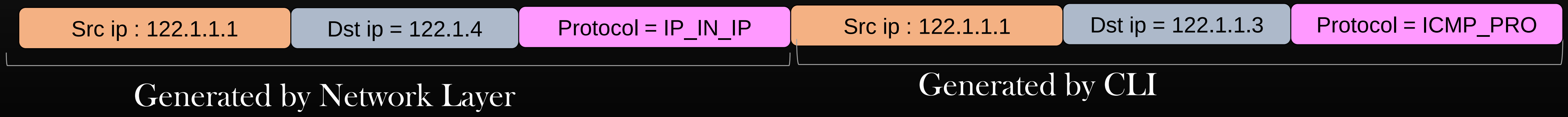
In Layer5/ping.c

```
void
layer3_ero_ping_fn (node_t *node, char *dst_ip_addr,
                                      char *ero_ip_address);
```

- This fn must do two tasks :
  - prepare the IP Hdr without Application payload. This IP Hdr shall be inner IP hdr of ip-in-in packet



Final Pkt :



Ip-In-IP Encapsulation

- Implementation Steps :

Step 3/3 : TCP/IP Stack Changes

- No Router ever sees the content of inner ip hdr as long as outer hdr is attached during the course of journey of the packet
- The TCP IP Stack will forward the pkt as usual until the packet reaches the ERO router (the dest for the outer ip hdr)
  - No changes required in the forwarding logic of Network Layer

- When the packet Reaches ERO router, ERO router must set the pkt onto its new journey to ultimate destination
  - Minor change is required in local host delivery case of Network Layer state diagram. Add the below case

```
In fn layer3_ip_pkt_recv_from_layer2(. . .) {

        case IP_IN_IP:
            layer3_ip_pkt_recv_from_layer2 (node, interface,
            (ip_hdr_t *)INCREMENT_IPHDR(ip_hdr),
            IP_HDR_PAYLOAD_SIZE(ip_hdr));
            return;
}
```

(This will set the pkt on its final course to ultimate destination !!)

Done ☺

Sequel Course Part 2

1. Interface Management & statistics
2. Dynamic L3 Route Calculation
   No More Manual installation of L3 routes
3. Making TCP/IP stack dynamic
   Dynamic ARP table Entries
4. Develop Logging Infra
   Packet Captures
5. Sample L2 Layer Application & Working with Timers
6. Programmable TCP/IP Stack

Developing TCP/IP Stack Part B

LIVE**

Pre-Requisite :

Must have completed **Part A**