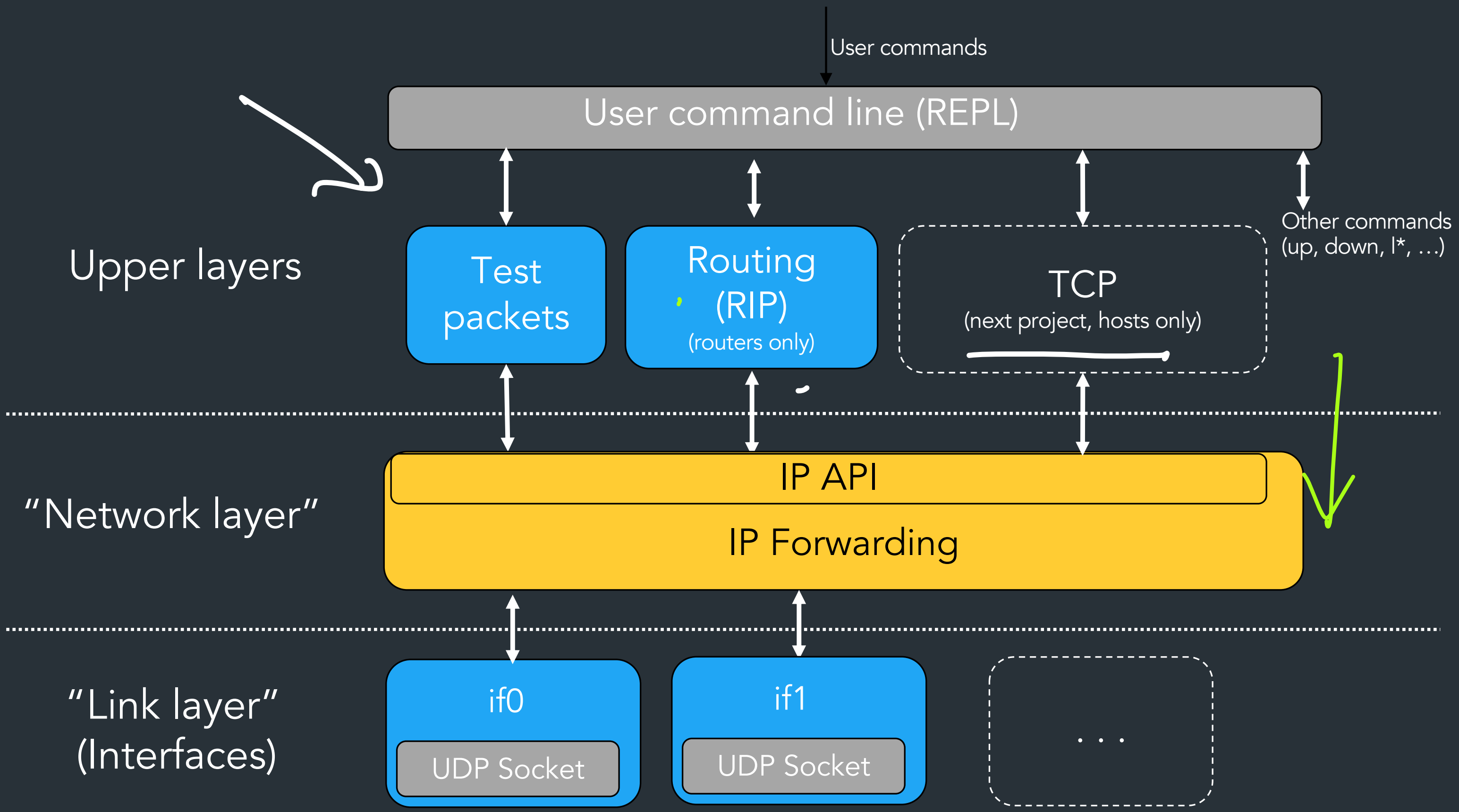
F24

# IP Project Gearup II

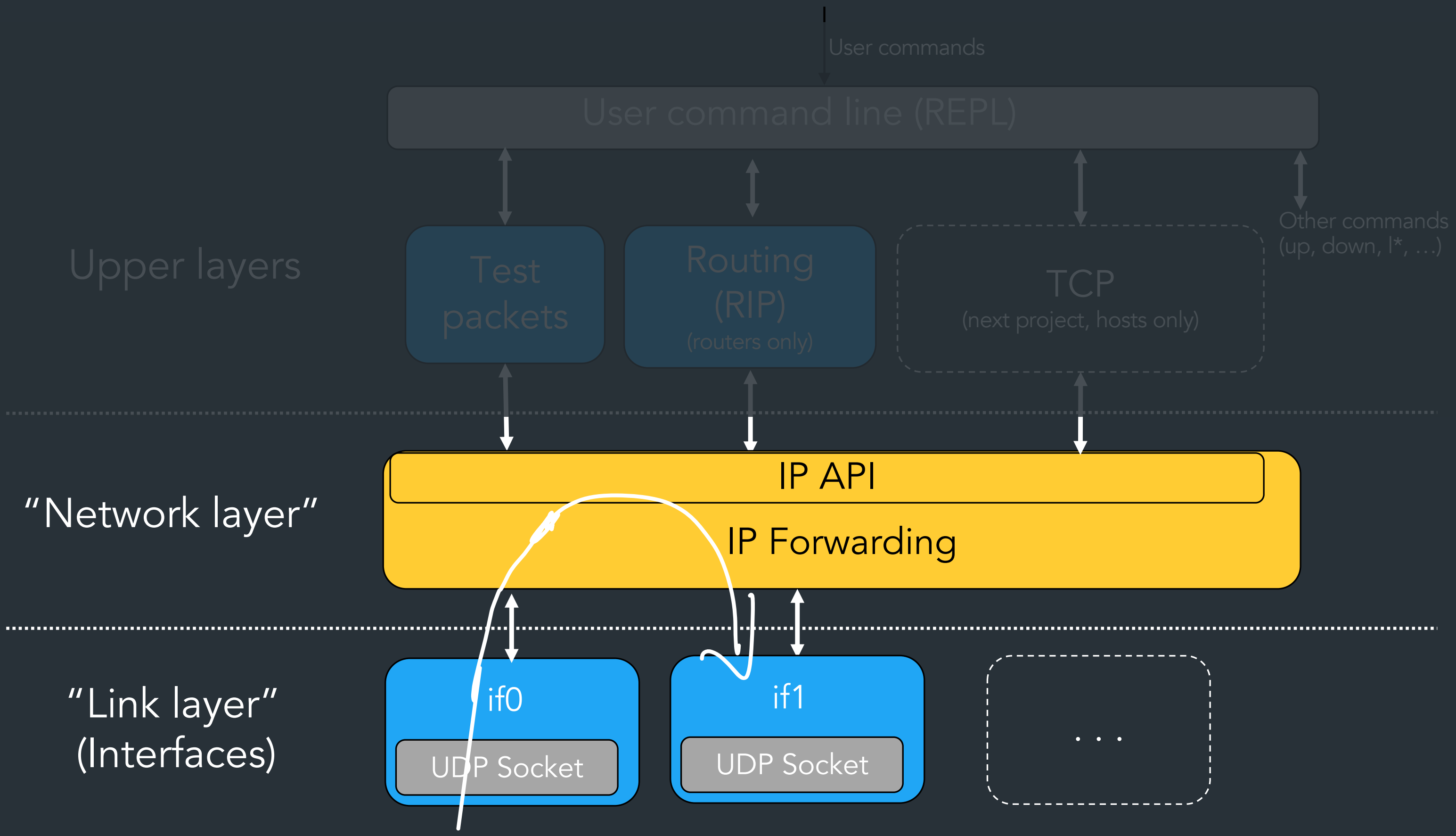
# Overview

- How to think about the link-layer/forwarding
- How to send packets / test with Wireshark
- Implementation notes
- Any questions you have

# The Big Picture



# What you should be focusing on first

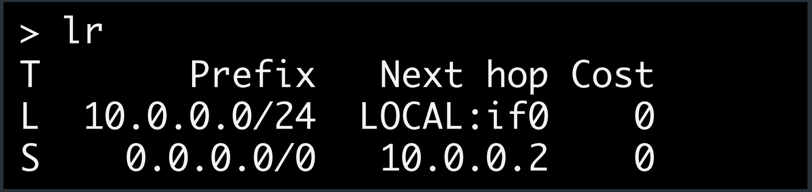


How to receive packets on interfaces, send them back out

## How does the link-layer work?

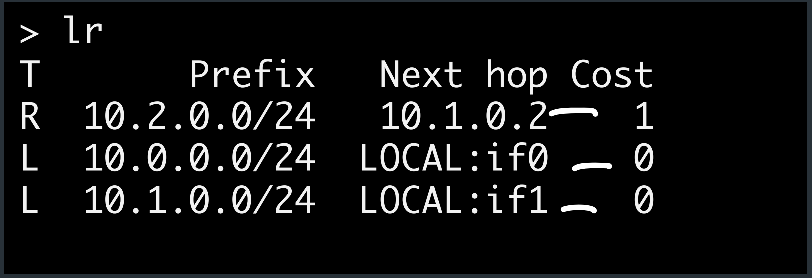
## What does it mean to forward vs. send on an interface?

HOST



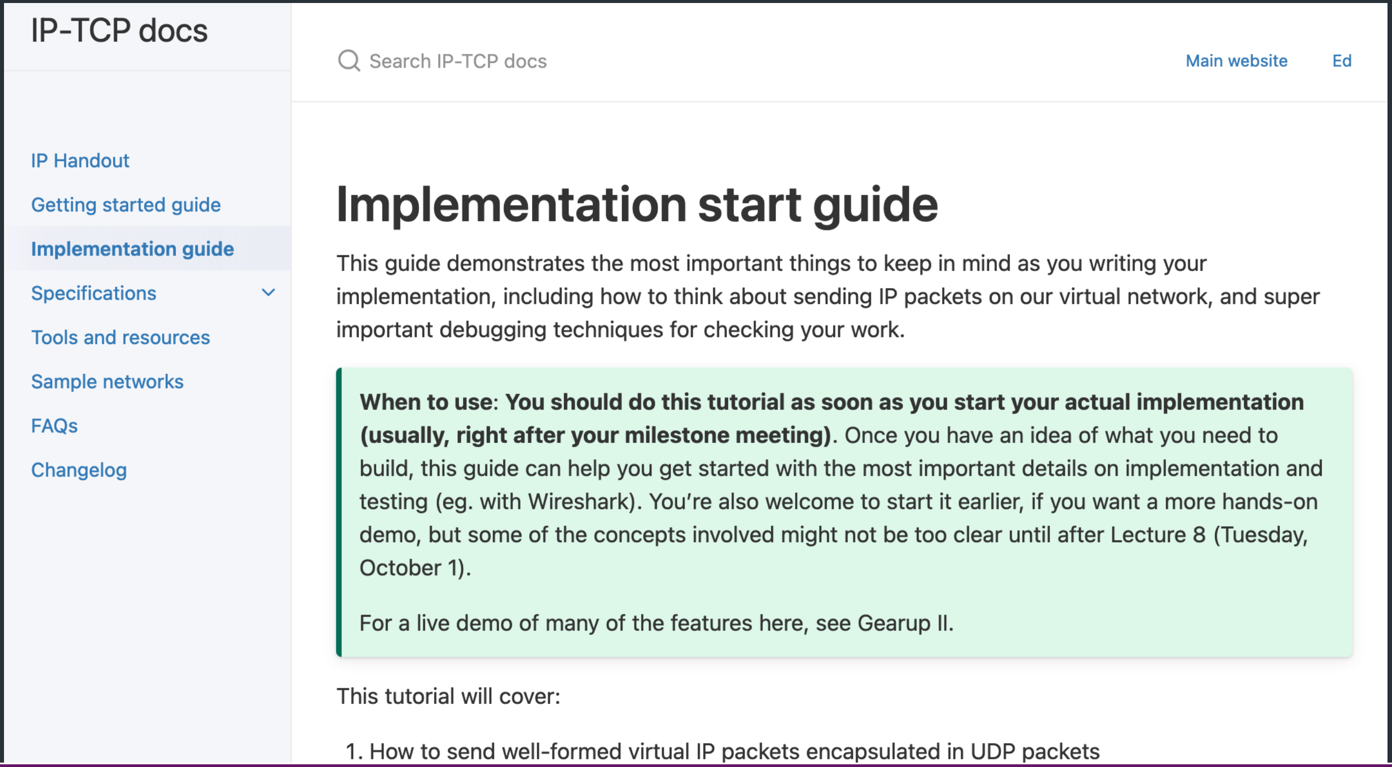
```
> lr
T       Prefix   Next hop Cost
L  10.0.0.0/24  LOCAL:if0    0
S    0.0.0.0/0   10.0.0.2    0
```

ROUTER



```
> lr
T       Prefix   Next hop Cost
R  10.2.0.0/24   10.1.0.2    1
L  10.0.0.0/24  LOCAL:if0    0
L  10.1.0.0/24  LOCAL:if1    0
```

# Key resource: Implementation Start Guide



## Implementation start guide

This guide demonstrates the most important things to keep in mind as you writing your implementation, including how to think about sending IP packets on our virtual network, and super important debugging techniques for checking your work.

**When to use: You should do this tutorial as soon as you start your actual implementation (usually, right after your milestone meeting)**. Once you have an idea of what you need to build, this guide can help you get started with the most important details on implementation and testing (eg. with Wireshark). You're also welcome to start it earlier, if you want a more hands-on demo, but some of the concepts involved might not be too clear until after Lecture 8 (Tuesday, October 1).

For a live demo of many of the features here, see Gearup II.

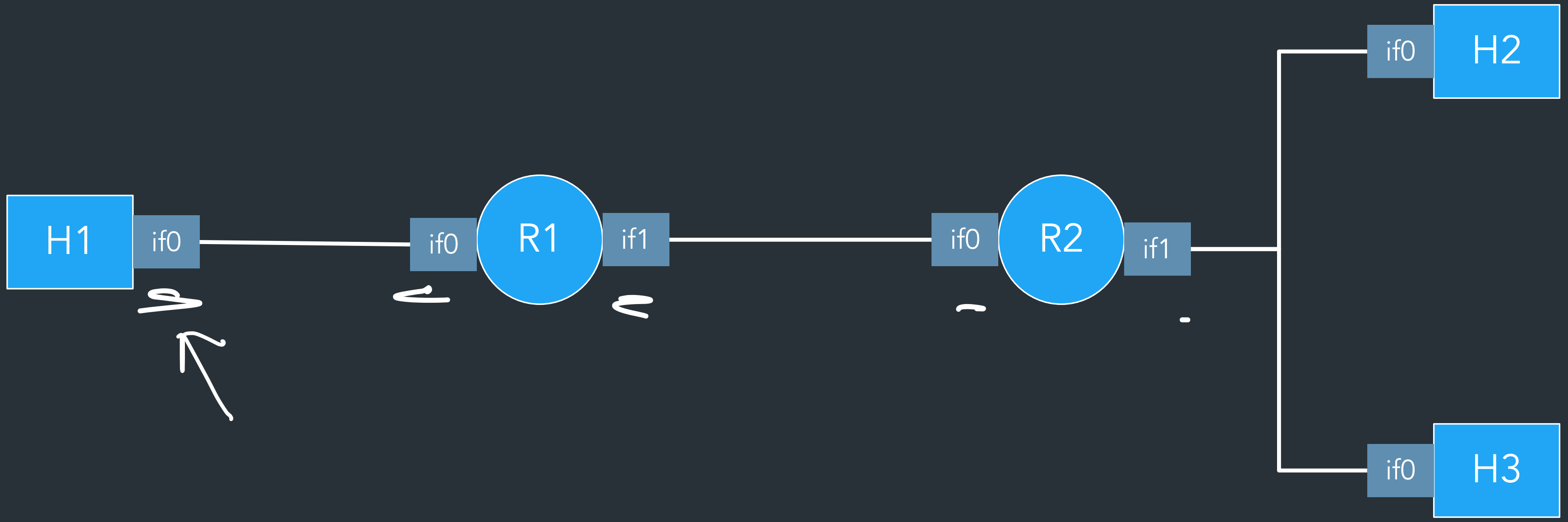
This tutorial will cover:

1. How to send well-formed virtual IP packets encapsulated in UDP packets

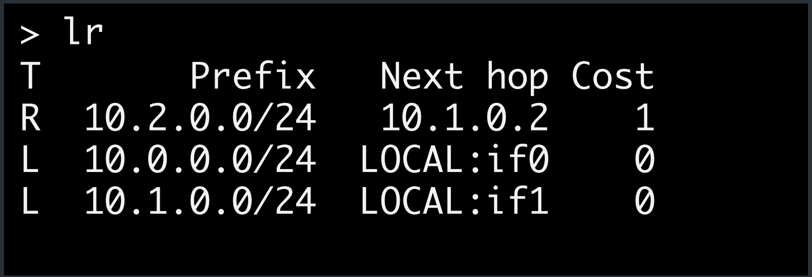
USE THIS!!!

=> Tutorial on how to set up sockets, what link-layer should look like
Do this when you're ready to start implementing. Find it here.

# doc-example

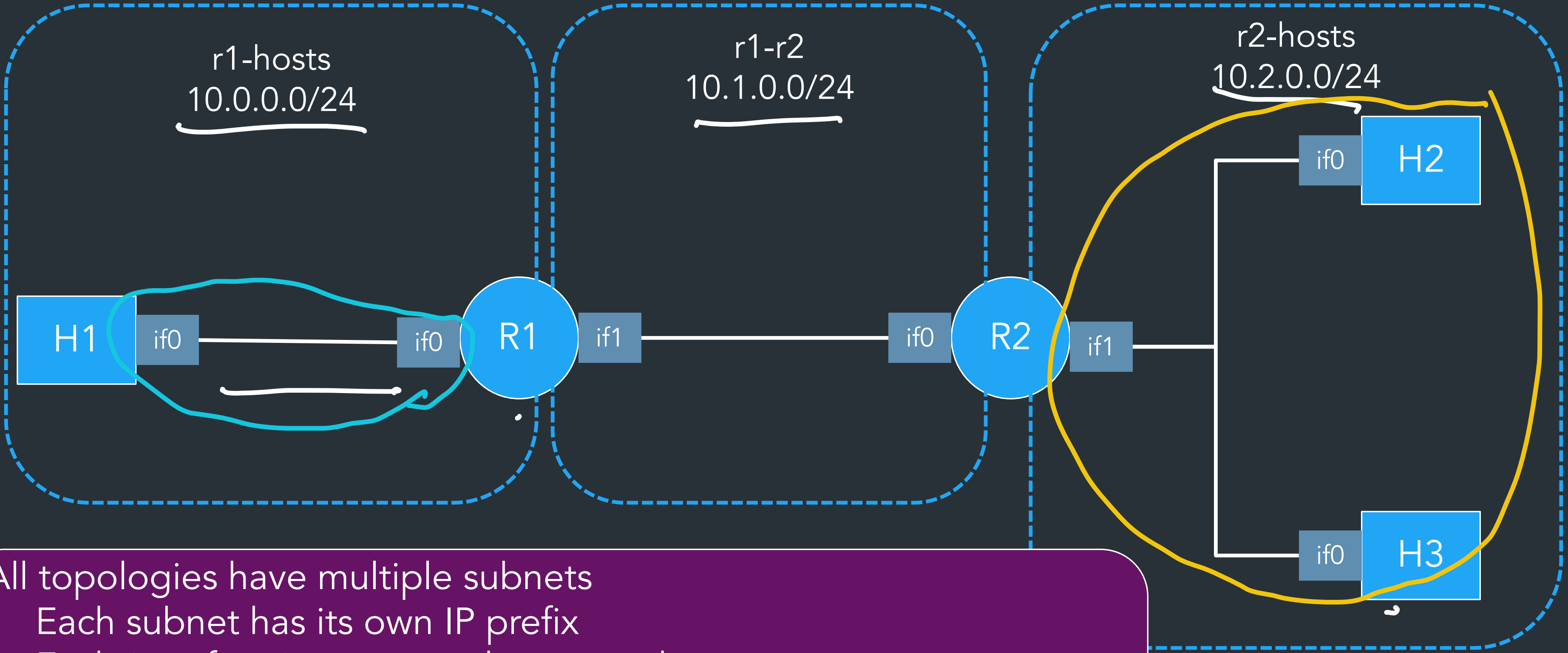


Node ::= “host” or “router”
All nodes connect via interfaces
⇒ Hosts have exactly one interface
⇒ Routers have multiple interfaces



```
> lr
T        Prefix     Next hop  Cost
R   10.2.0.0/24     10.1.0.2     1
L   10.0.0.0/24   LOCAL:if0     0
L   10.1.0.0/24   LOCAL:if1     0
```

# doc-example



All topologies have multiple subnets

- Each subnet has its own IP prefix
- Each *interface* is connected to one subnet
- Nodes on the same subnet are *neighbors*

*=> Nodes always know how to send packets to their neighbors*

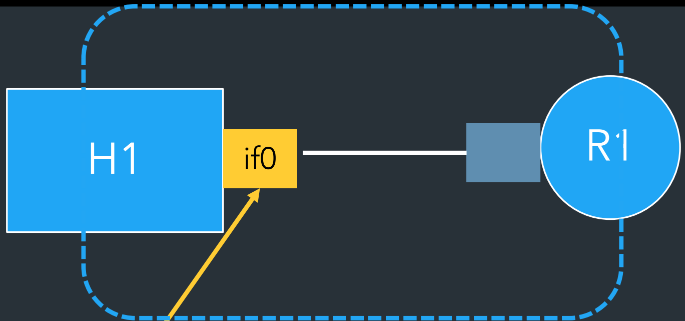
*Example:*

*- H2, H3, and R2's if1 are neighbors*

*- H1 and R1's if0 are neighbors*

# Interface:  has a virtual IP, network, “link-layer” UDP port

```
h1.lnx
interface if0 10.0.0.1/24 127.0.0.1:5000 # to network r1-hosts
neighbor 10.0.0.2 at 127.0.0.1:5001 via if0 # r1
route 0.0.0.0/0 via 10.0.0.2
```



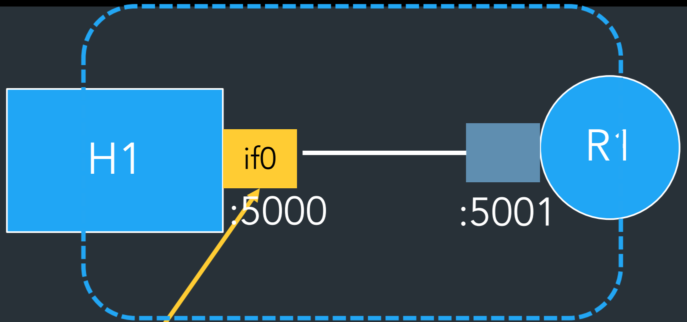
Config for if0

Virtual IP:  10.0.0.1
Network:  10.0.0.0/24
UDP:  bind on 127.0.0.1:5000

"LINK-LAYER": ALL INTERFACES LISTEN ON ONE UDP PORT.
(HERE, PORT 5000)

```
h1.lnx
interface if0 10.0.0.1/24 127.0.0.1:5000 # to network r1-hosts
neighbor 10.0.0.2 at 127.0.0.1:5001 via if0 # r1
route 0.0.0.0/0 via 10.0.0.2
```

One "neighbor" directive for each node on this subnet



Each interface has a set of neighbors

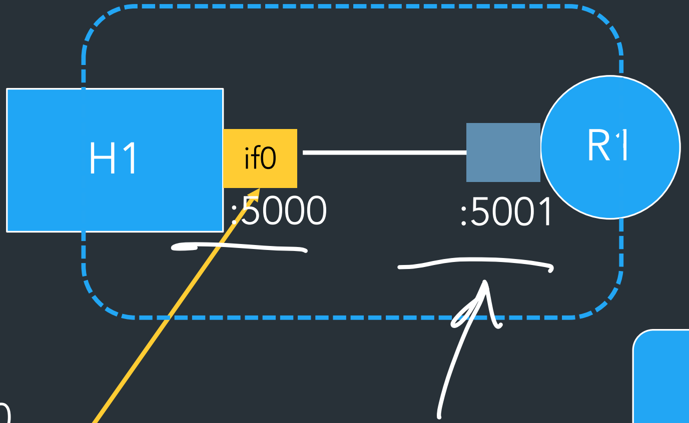
Can always send directly to your neighbors (ie, always know the UDP ports for your neighbors)

Config for if0

Virtual IP: 10.0.0.1
Network: 10.0.0.0/24
UDP: bind on 127.0.0.1:5000
neighbors: { 10.0.0.2 => 127.0.0.1:5001 }

Each interface has a list of neighbors: mapping of IPs to UDP ports
=> Like an ARP table, but always known ahead of time

```
h1.lnx
interface if0 10.0.0.1/24 127.0.0.1:5000 # to network r1-hosts
neighbor 10.0.0.2 at 127.0.0.1:5001 via if0 # r1
route 0.0.0.0/0 via 10.0.0.2
```



Config for if0

Virtual IP:  10.0.0.1
Network:  10.0.0.0/24
UDP:  bind on 127.0.0.1:5000
neighbors:  { 10.0.0.2 => 127.0.0.1:5001 }

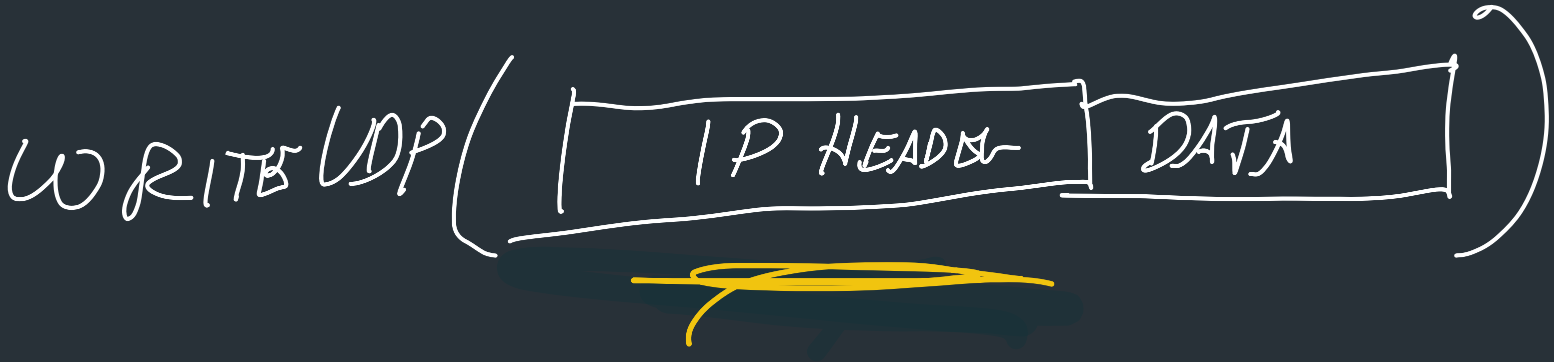
=> H1 can reach 10.0.0.2
by sending to UDP port 5001

So if we want to send from H1 to R1,
we need to send something to UDP port 5001 => but what?

# How to think about encapsulation

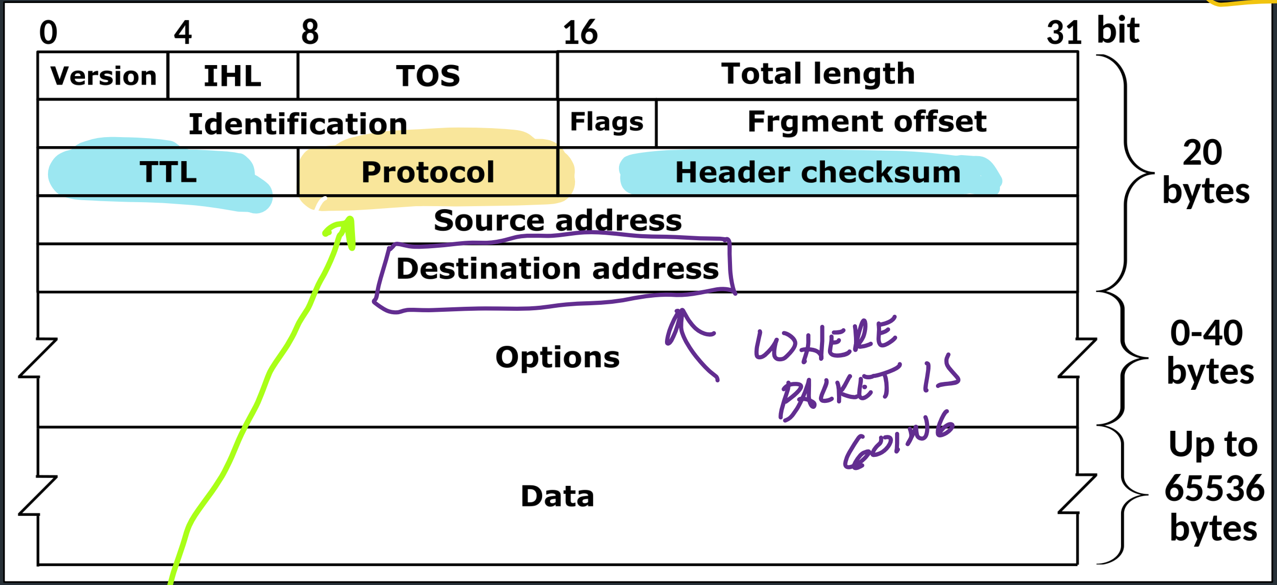
- Each interface:  thread/goroutine/etc listening on a UDP port
- Each packet contains an IP header + whatever message content

WHAT IS SENT ON SOCKET:



FOR MORE INFO,
SEE LECTURE 7

# IP Header



0 4 8 16 31 bit

Version | IHL | TOS | Total length

Identification | Flags | Frgment offset

TTL | Protocol | Header checksum

Source address

Destination address

Options

Data

20 bytes

0-40 bytes

Up to 65536 bytes

WHERE PACKET IS GOING

HOW TO INTERPRET THE DATA IN THE PACKET

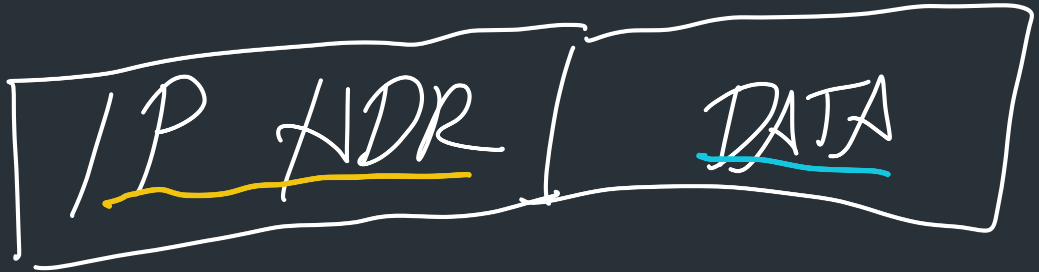
# UDP-in-IP example

- Complete code example for building an IP header, adding it to a packet, and sending it via UDP
  - Also computes/validates checksum!

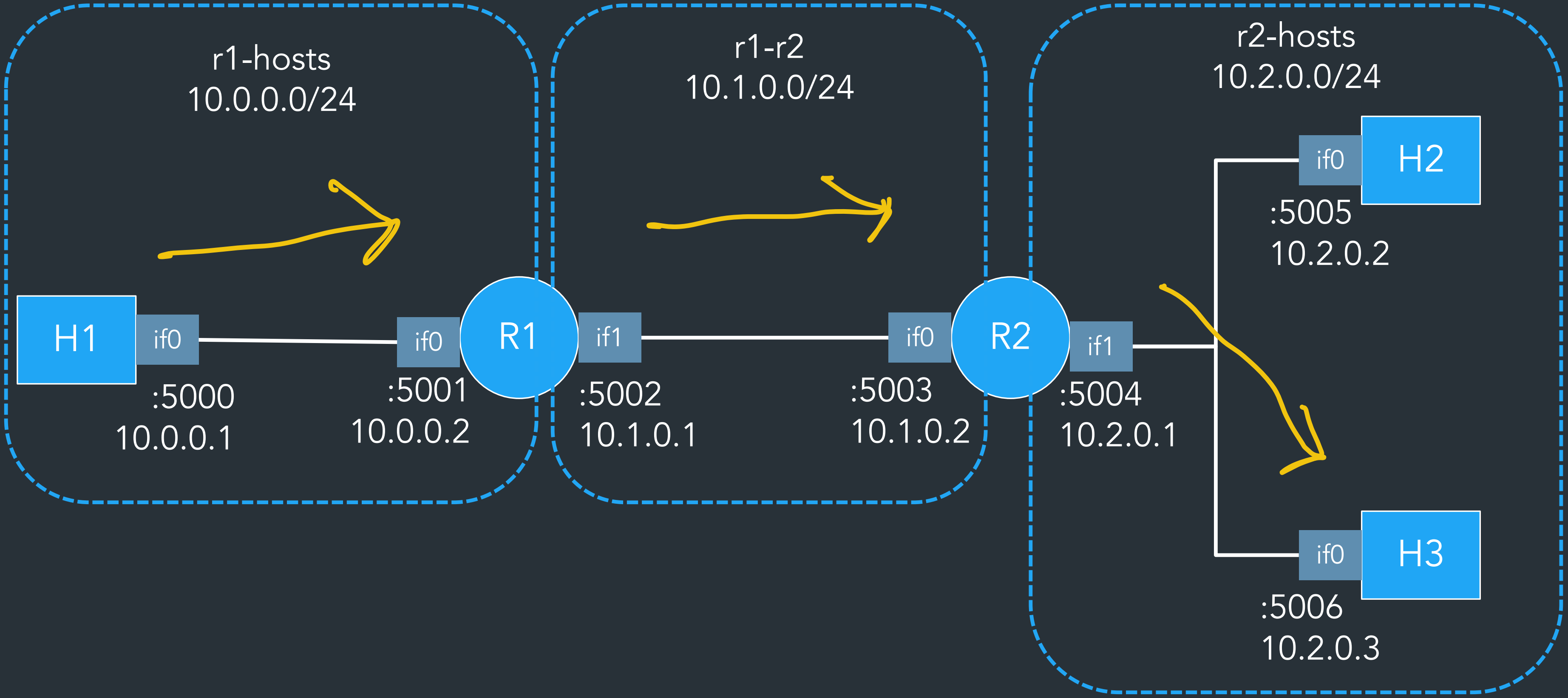
- Let's break down how this works…

# To send some data

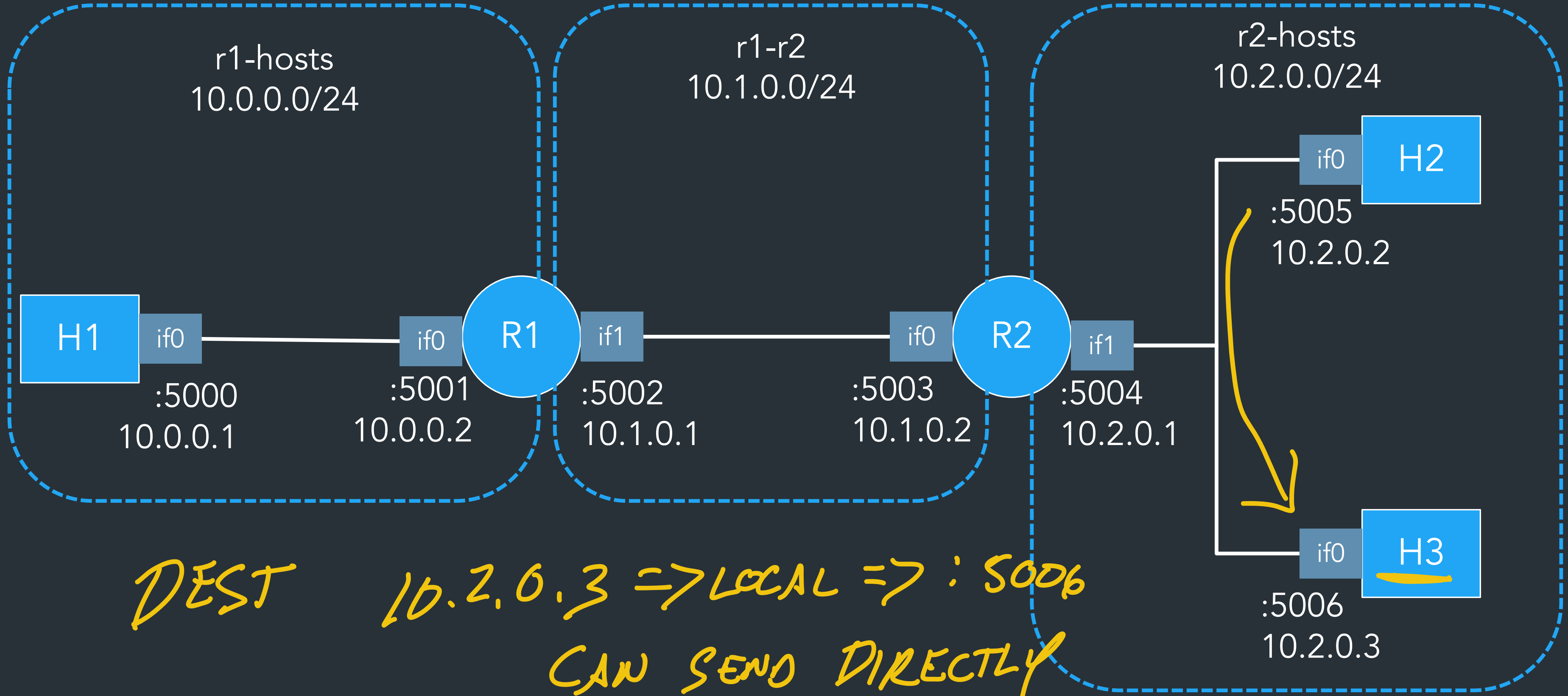
- Build an IP header
  - Fill in all header fields as appropriate (source, dest IP, etc.)
  - Compute the checksum
- UDP Packet: IP header + data you want to send
- Send packet via socket for that interface



# What would it look like to send from h1 -> h3?



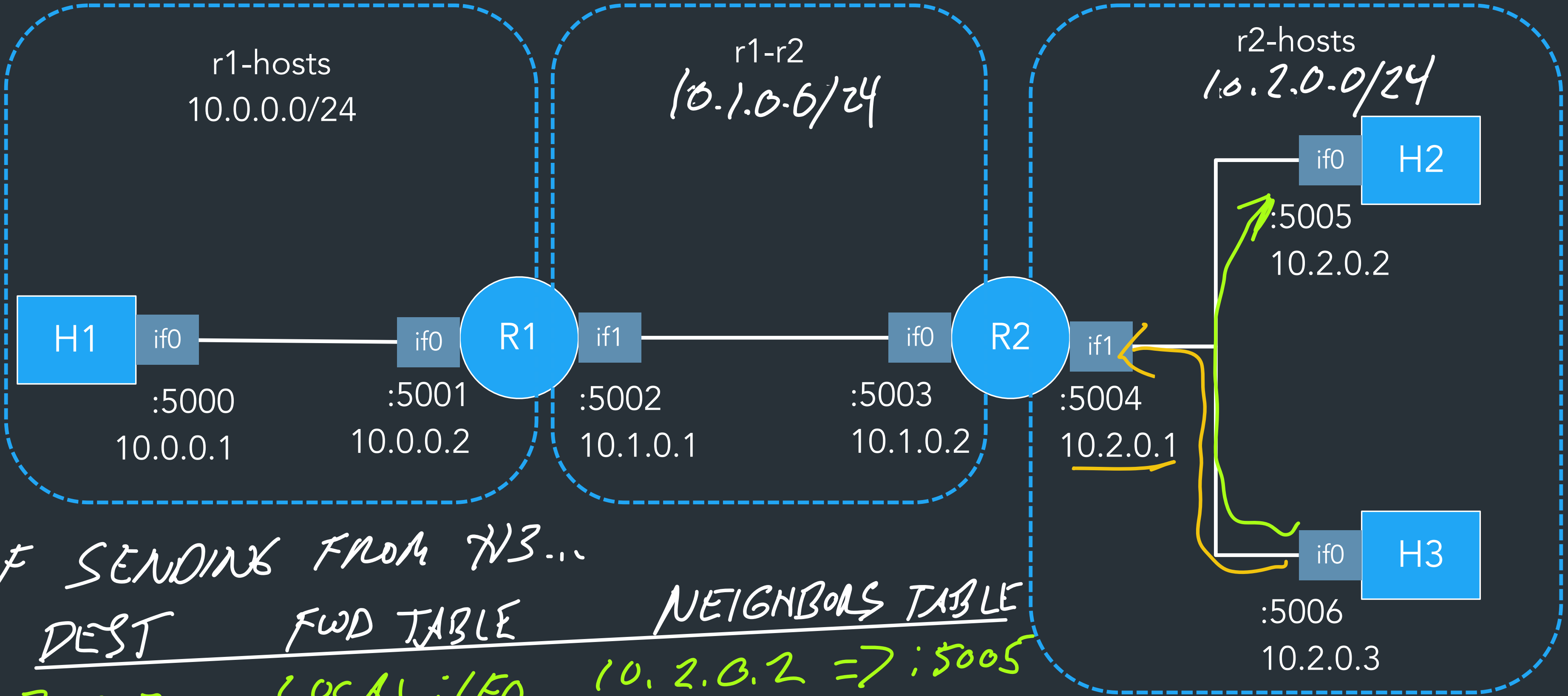
# What happens if h2 sends to h3?



DEST 10.2.0.3 => LOCAL => : 5006

CAN SEND DIRECTLY

TO N3!



IF SENDING FROM H3...

| DEST | FWD TABLE | NEIGHBORS TABLE |
|---|---|---|
| 10.2.0.2 | LOCAL:IF0 | 10.2.0.2 => :5005 |
| 10.0.0.1 | 10.2.0.1 (DEFAULT) | 10.2.0.1 => :5004 |

H2 IS NEIGHBOR, SO CAN SEND DIRECTLY TO HOST!

# FORWARDING STEPS:

## CONSIDER PACKET WITH DESTINATION IP D

If destination IP D matches one of this node's assigned IPs
  => Packet is for this node  => Send "up" (more on this later)

Otherwise, check forwarding table to look for a match
  (If multiple matches, take the most specific prefix (lecture 7, 9)

If the result is a local route (ie, maps to some ifX)

  => Look up UDP port for D in neighbors table for ifX
    Send packet to this port

(SEND DIRECTLY
E.G. H2 → $H_3$)

If the result is not a local route (ie, has next hop IP G)
  => Need to send packet to G instead:
  Look up G in forwarding table
    => maps to some local route on some interface ifY
  Look up UDP port for G in ifY's neighbor's table
    Send packet to this port

(SEND VIA GATEWAY
E.G. H2 → R2 → ...)

CHOOSING NEXT HOP DESTINATION!

# HOW TO SEND "UP"?

OUR NODES DO DIFFERENT THINGS WITH PACKETS:

PROTOCOL NUM

HOSTS:

- TEST PACKETS (0)
- TCP (6)

ROUTERS

- TEST PACKETS (0)
- RIP PACKETS (200)

⇒ LOOK UP A "HANDLER" (CALLBACK FUNC) FOR PACKET BASED ON PROTOCOL NUMBER

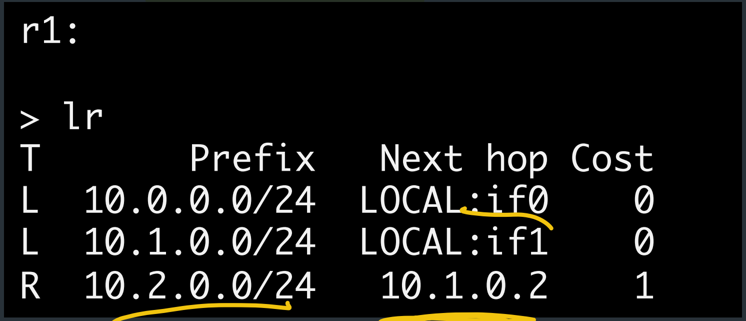
RegisterHandler(num, someFunc)

DO THIS AT STARTUP — TELL IP STACK TO CALL SOMEFUNC WHEN RECEIVING A PACKET W/ THIS PROTOCOL.

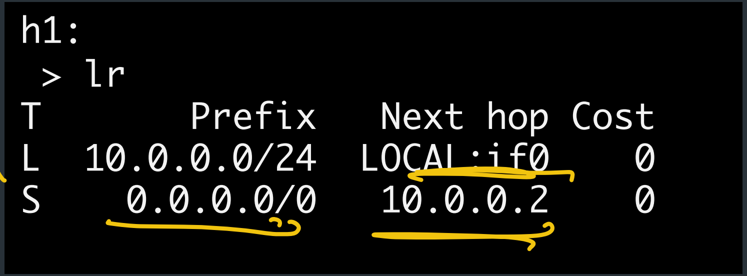
# How to table lookup?

Dst => 10.0.5 => LOCAL

Dest IP == 10.0.0.5, where to send packet?



```
r1:

> lr
T        Prefix    Next hop  Cost
L  10.0.0.0/24  LOCAL:if0     0
L  10.1.0.0/24  LOCAL:if1     0
R  10.2.0.0/24   10.1.0.2     1
```

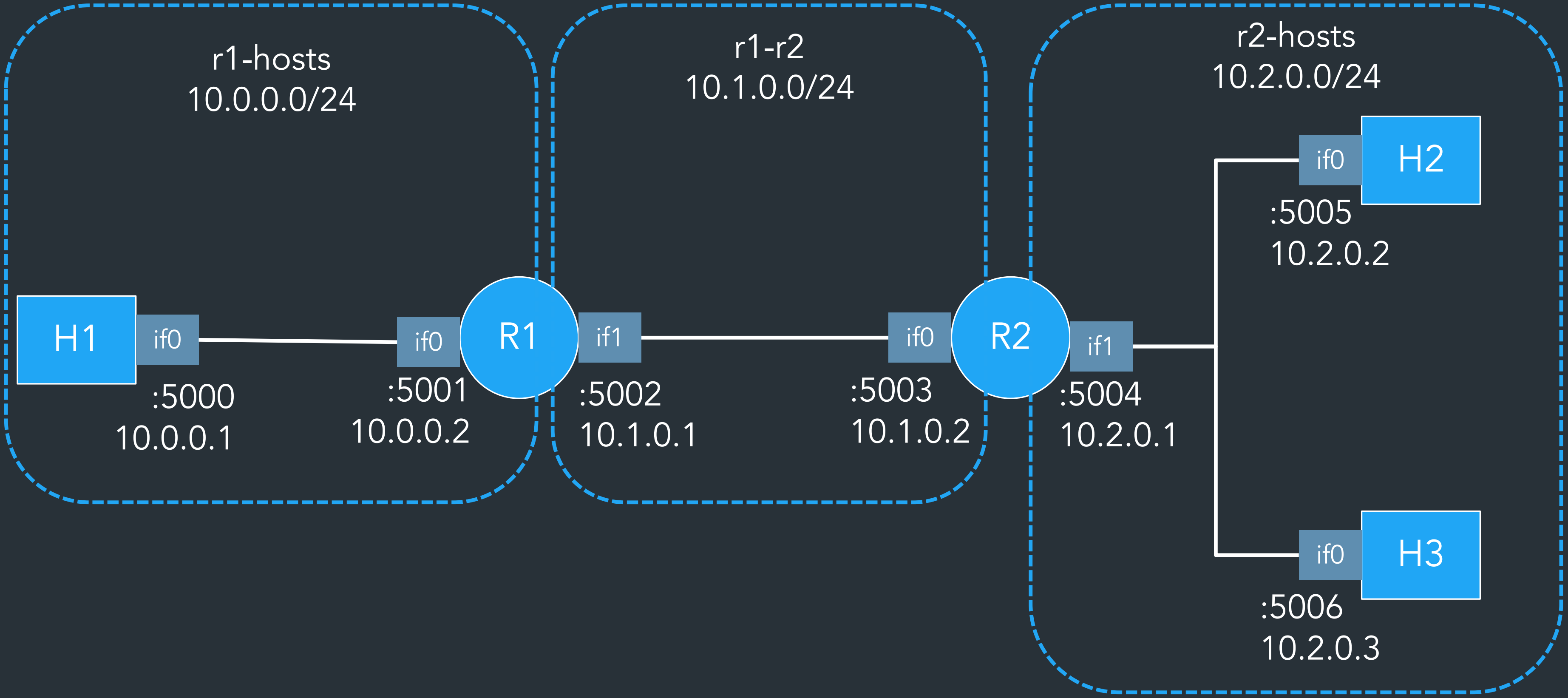


```
h1:
 > lr
T        Prefix    Next hop  Cost
L  10.0.0.0/24  LOCAL:if0     0
S    0.0.0.0/0   10.0.0.2     0
```

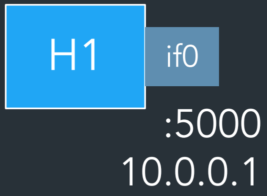
- You can decide how to store the table
- Need to find the most specific matching prefix
- Use built-in datatypes to help you!
  Go:  prefix.Contains() (netip.Prefix)

You do NOT need to be particularly efficient about this step!

# How to think about routing?



# ON A HOST:

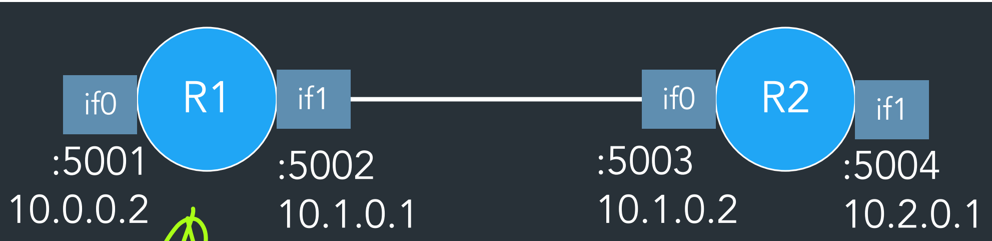


```
h1.lnx
interface if0 10.0.0.1/24 127.0.0.1:5000 # to network r1-hosts
neighbor 10.0.0.2 at 127.0.0.1:5001 via if0 # r1


routing static

# Default route
route 0.0.0.0/0 via 10.0.0.2
```

← No RIP.

Hosts don't use RIP
=> forwarding table is constant (just send to router)

ON A ROUTER:



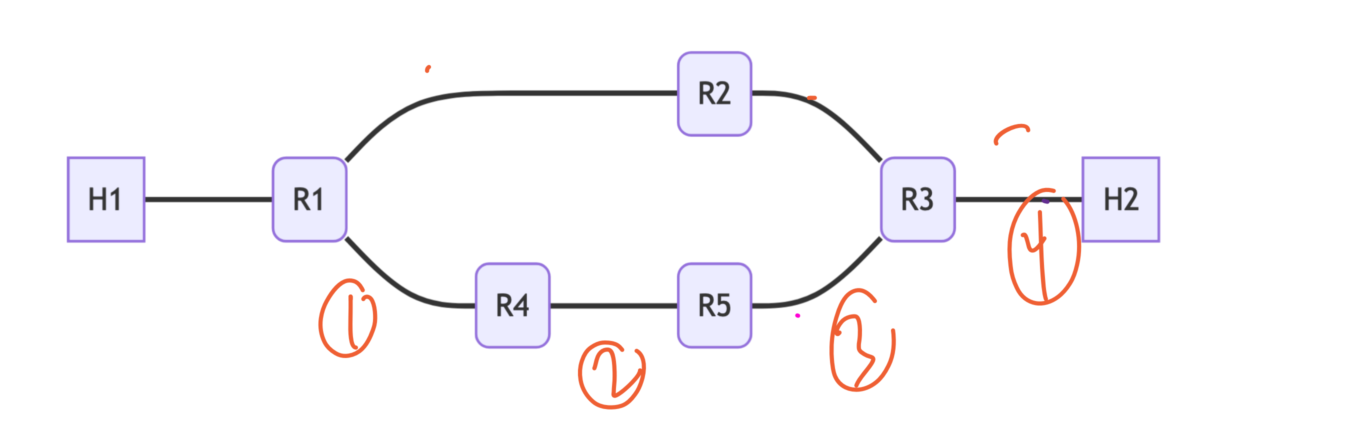
```
r1.lnx
interface if0 10.0.0.2/24 127.0.0.1:5001 # to network r1-hosts
neighbor 10.0.0.1 at 127.0.0.1:5000 via if0 # h1


interface if1 10.1.0.1/24 127.0.0.1:5002 # to network r1-r2
neighbor 10.1.0.2 at 127.0.0.1:5003 via if1 # r2


routing rip

# Neighbor routers that should be sent RIP messages
rip advertise-to 10.1.0.2

# Timing parameters for RIP
rip periodic-update-rate 5000 # in milliseconds
rip route-timeout-threshold 12000 # in milliseconds
```

Routers use RIP
=> Send updates to neighboring routers
=> Set of neighbor routers always known

SET OF "RIP NEIGHBORS"

← CONSTANTS FOR TIMING

# The “loop” network



Multiple paths from H1 <-> H2 of different costs
=> Use this to test RIP (after you've tested on a smaller network with only two routers)

=> See video for a demo of how things should look when you test

# Implementation: key resources

- Use an external library for parsing IP header (don't do this yourself)
  - For Go/C, see UDP-in-IP examples
  - Rust: etherparse library

- We provide parsers for the lnx files—don't make your own

- You're welcome to use third-party libraries, so long as they don't trivialize the assignment (ask if you're concerned)
  - Data structures, argument parsing, are fine

# IP types and go

Go has two IP types, ~~net.IP~~ and (newer) netip.Addr

- netip.Addr and netip.Prefix the one you want

⇒ These libraries have useful helper functions, use them!

# Testing your IP

vnet_run: Run all nodes in a network automatically

- Can run on your node, or the reference
- Uses tmux: see getting started guide for details

Lots of ways to test => See Tools and Resources!

- Wireshark: your best resource (see implementation guide)
- Can run some nodes as reference, some nodes as yours
- Can run nodes with debugging

# Viewing packets in wireshark

SEE VIDEO + IMPL GUIDE!

# Sample Topologies

Some example networks you can test with…

See “Sample networks” page for more info, including what kinds of things you can test with each network

# Roadmap

Start with forwarding first:
**Think about:** Listening on interfaces, parsing/sending IP packets, consulting forwarding table, printing test packets

IMPL GUIDE!

1. Send across one link: H1->R1
2. Forward across one router :
   - linear-r1h2: H1->R1->H2
   - linear-r1h4 (same thing, multiple hosts)

   …

# Roadmap

Once you can send across one router, start thinking about RIP

3. Make sure you can share routes and update the forwarding table
   - Eg. linear-r2h2: H1 -> R1 -> R2 -> H2

4. Try disabling/enabling links, make routes expire

5. Loop network: finding best path, updating routes as topology changes