IP Project Gearup I

Overview

- Motivation and overview
- “The IP stack”
- “The virtual network”
- How to get started for the milestone
- Any questions you have

IP+TCP Projects: the goal

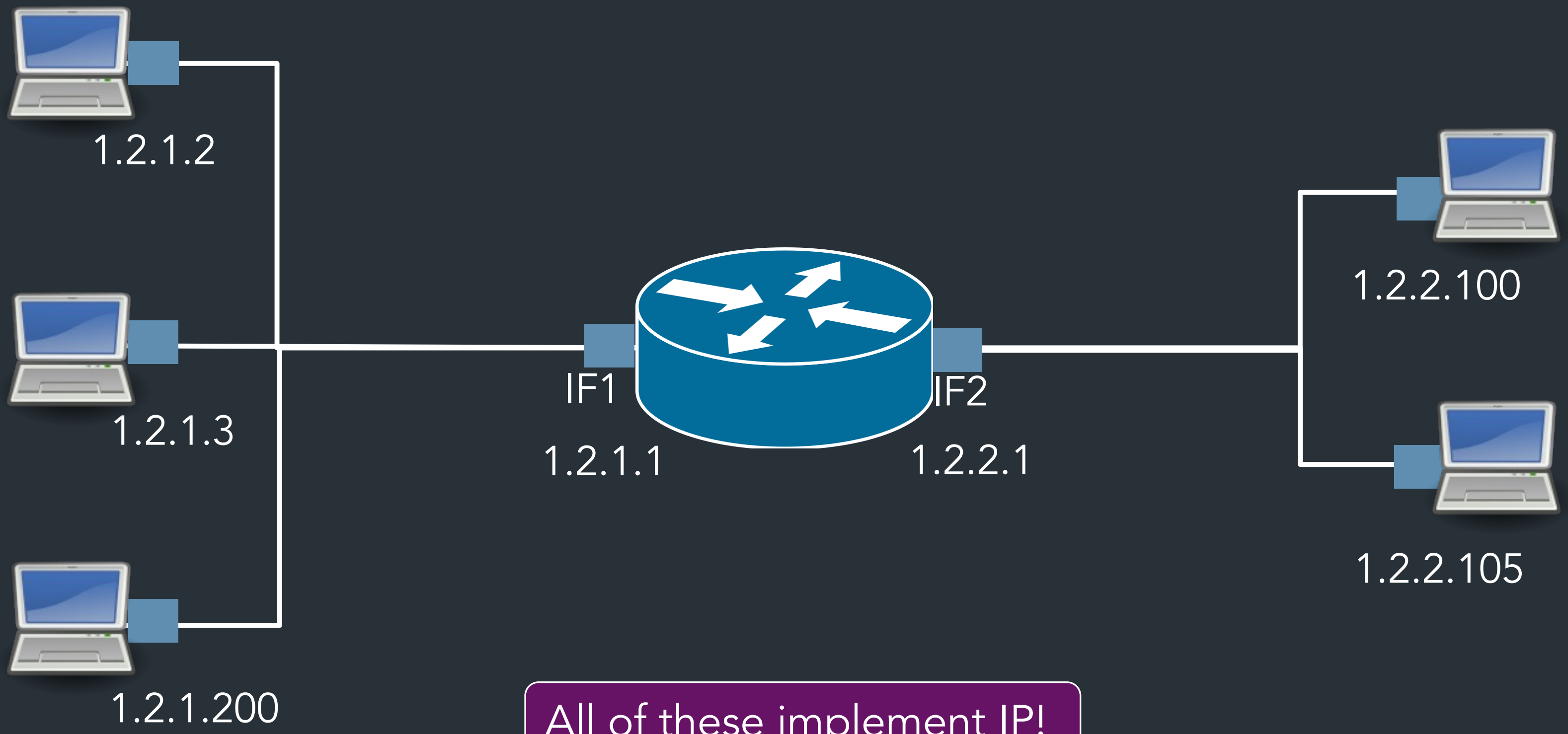
Goal: implement core parts of an OS networking stack

- Learn how core Internet protocols work
- Learn how OS implements networking
- Learn how to work and debug at *multiple layers of abstraction*

Networking and software design!

A “network stack”

Example: a network



All of these implement IP!

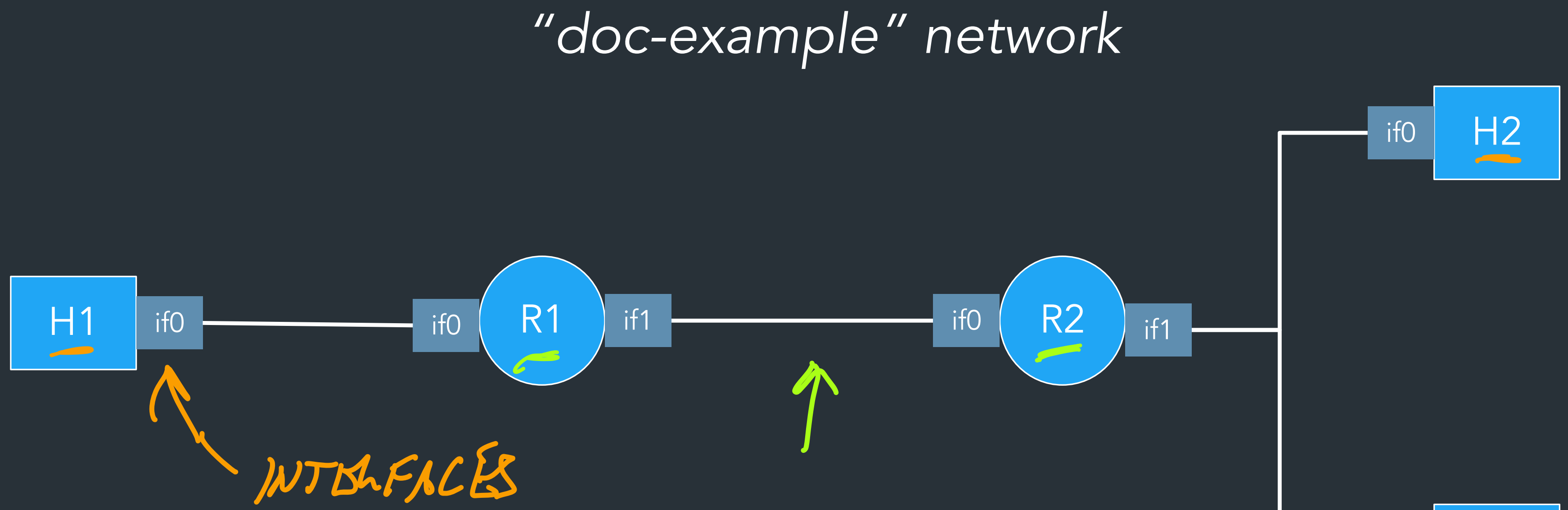
Our "virtual" network

We'll "simulate" a network, in userspace

- Build two programs: a "vhost" and "vrouter" that use your IP stack

- Networking: your programs communicate via UDP sockets (more on this later)

An example virtual network

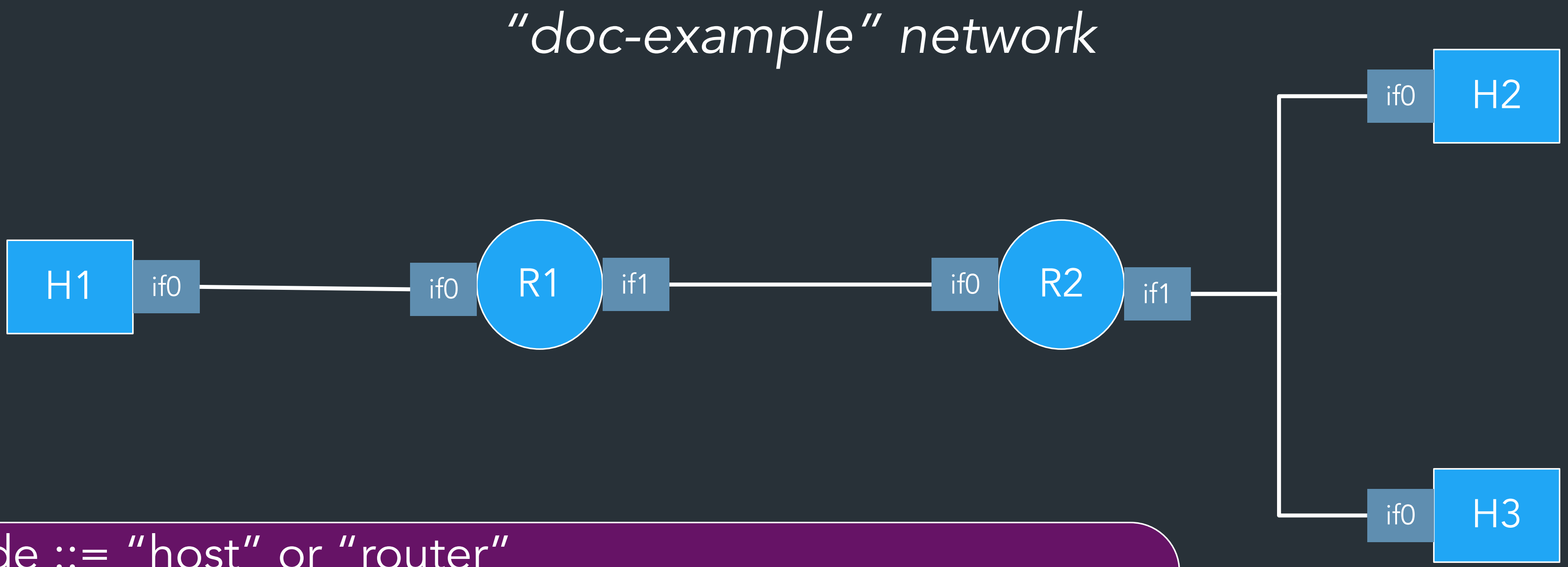


Network is made of nodes: hosts or routers

Hosts have one interface

Routers have >1 interface

An example virtual network


“doc-example” network

Node ::= “host” or “router”

All nodes connect via *interfaces*

⇒ Hosts have exactly one interface

⇒ Routers have multiple interfaces

⇒ Each interface is a UDP socket (more on this later)

Configuring the nodes

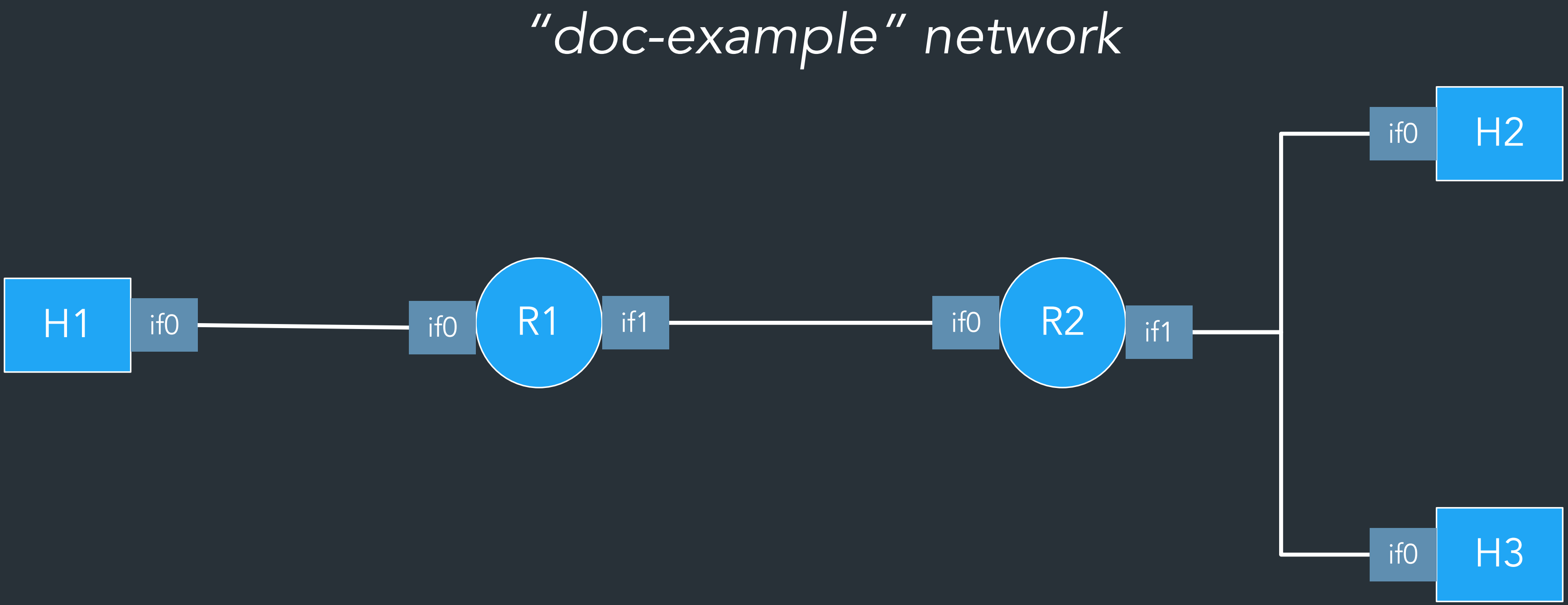
All nodes (vhost, vrouter) take in a configuration file (a ".lnx file") to tell it about the network:

- How many interfaces, their "virtual" IP addresses, etc.

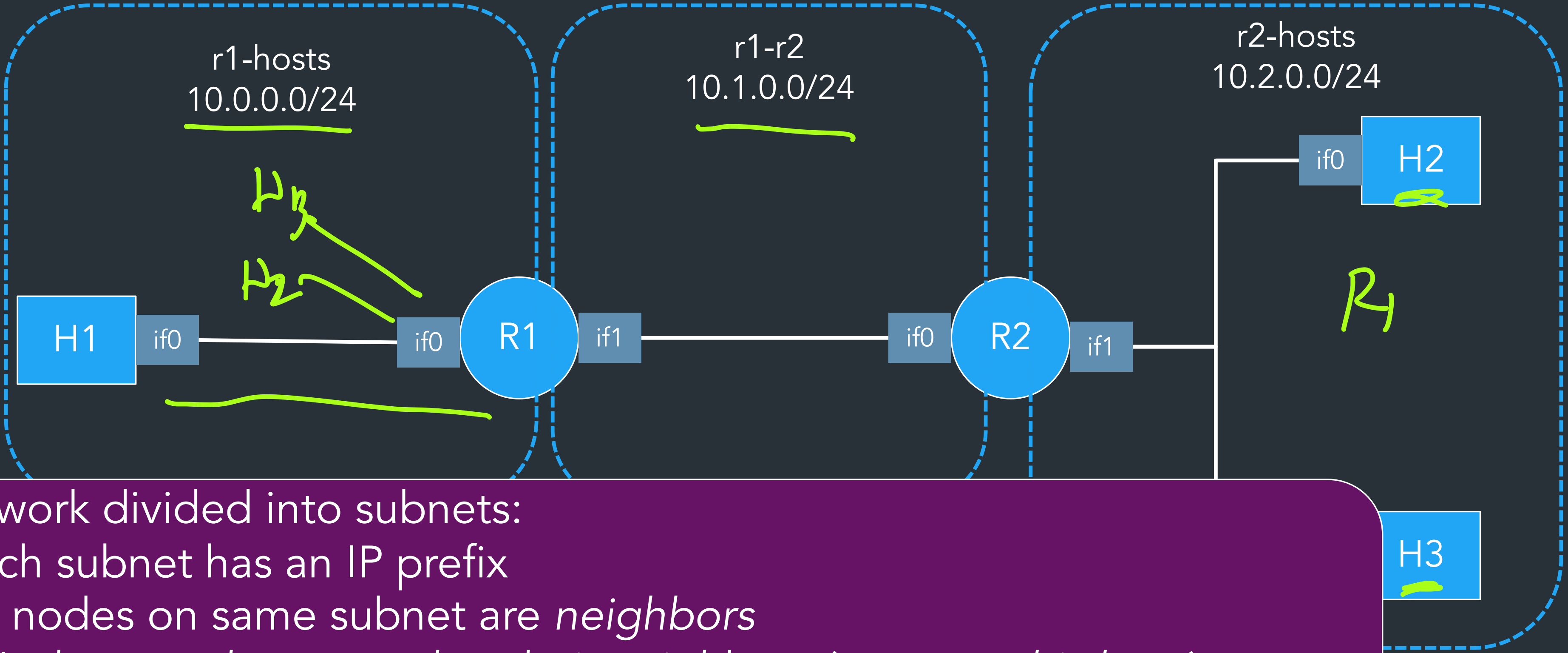
Can run your nodes in different *topologies*, depending on the config files

See "Sample networks" in docs page

An example virtual network



Example: subnets

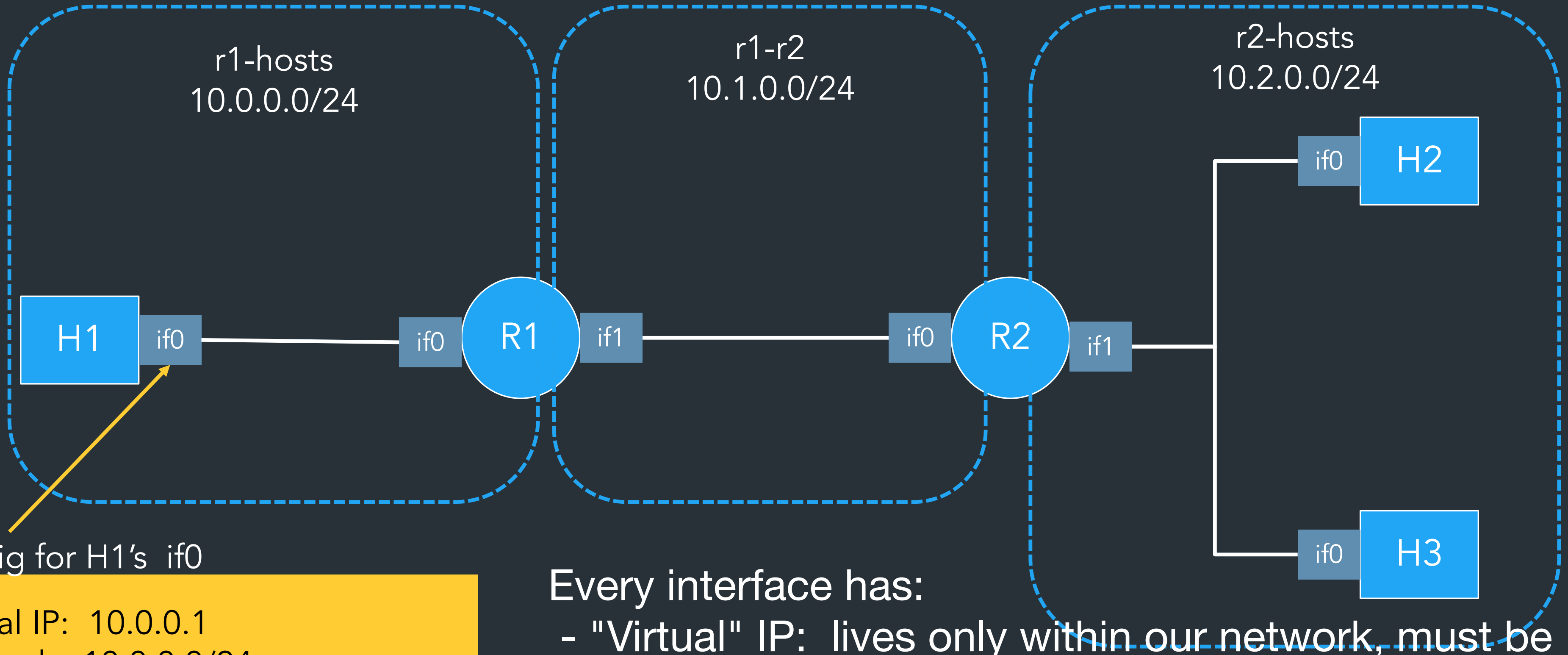


Network divided into subnets:

- Each subnet has an IP prefix
- All nodes on same subnet are *neighbors*
- *Nodes can always send to their neighbors (more on this later)*

=> Goal: routers need to forward packets between networks

Example: interfaces



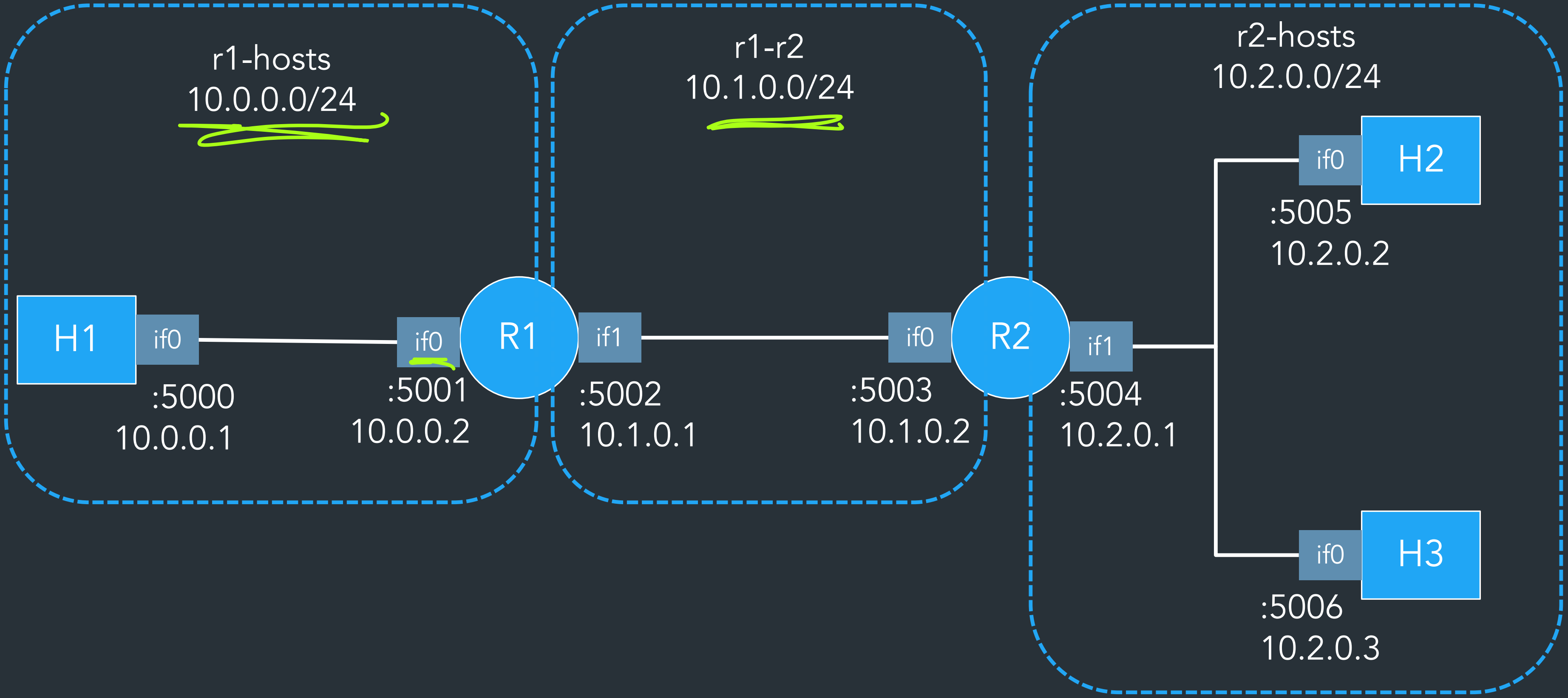
Config for H1's if0

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

Every interface has:

- "Virtual" IP: lives only within our network, must be within this subnet
- UDP port to send/recv packets on that interface
- Node knows the UDP ports of its neighbors

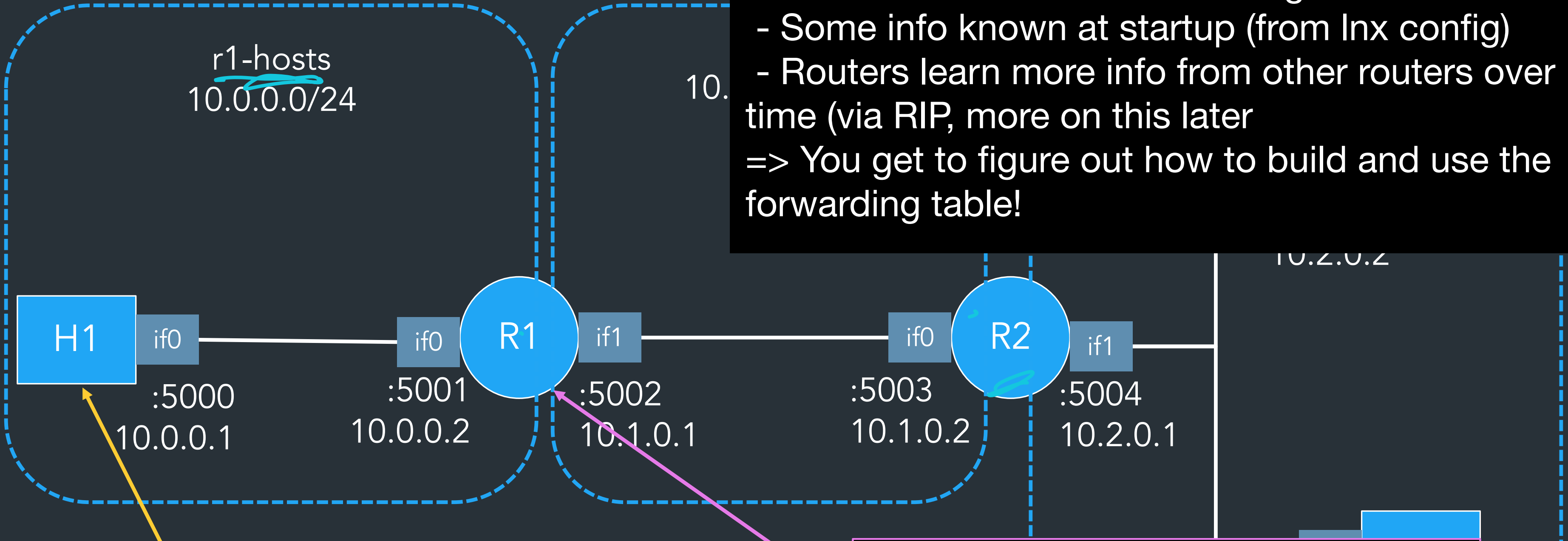
doc-example


r1-hosts
10.0.0.0/24
r1-r2
10.1.0.0/24
r2-hosts
10.2.0.0/24
H1
if0
:5000
10.0.0.1
if0
:5001
10.0.0.2
R1
if1
:5002
10.1.0.1
if0
:5003
10.1.0.2
R2
if1
:5004
10.2.0.1
if0
H2
:5005
10.2.0.2
if0
H3
:5006
10.2.0.3

Example: forwarding tables

Each node has its own forwarding table
- Some info known at startup (from lnx config)
- Routers learn more info from other routers over time (via RIP, more on this later

=> You get to figure out how to build and use the forwarding table!



```
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
```

```
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
```

IP project: Goals

- Forwarding: send packets between nodes
- Routing: Routers implement a routing protocol (RIP) to tell other routers about their networks

At startup, routers only know about their local networks
 => Routing algorithm: tell other routers about your routes => build global picture of whole network

Goal: All nodes can communicate with all other nodes!

IP project: Goals

- Forwarding: send packets between nodes

- Routing: Routers implement a routing protocol (RIP) to tell other routers about their networks

- Start of your "IP stack": API you can extend for other "applications" later

Goal: All nodes can communicate with all other nodes!

How configuration works

Two config files:

Network definition file (doc-example.json):

- Defines the subnets, how they're connected
- One per network
- You won't interact with it much

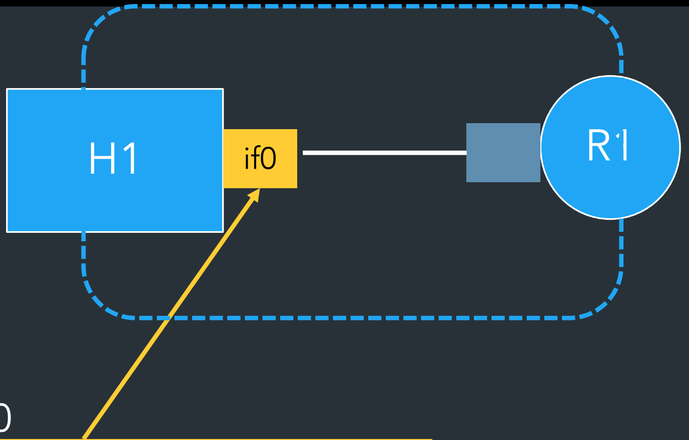
ONE FILE PER NODE.

lnx file (per-device config):

- Tells a specific vhost, vrouter what it knows about the network
- We give you the parser

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
```

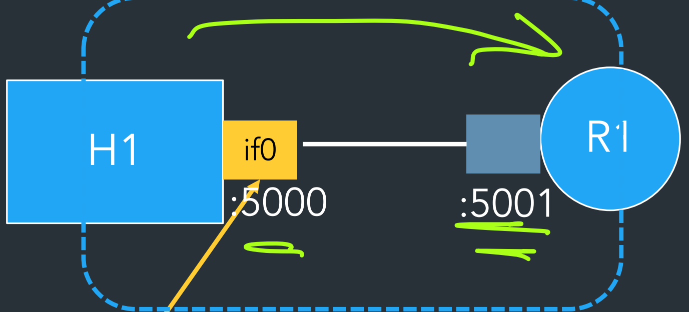


Config for if0

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

```
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
```

INPUT ON STARTUP.

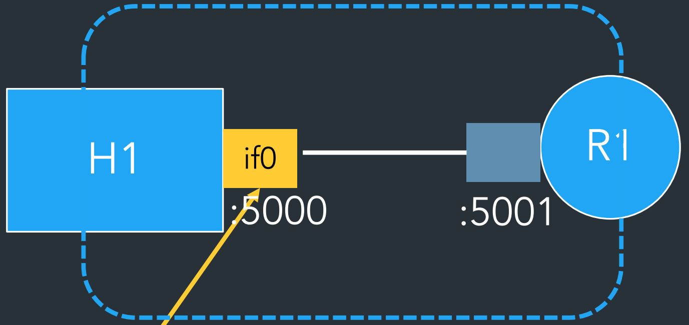


Config for if0

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

```
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
```

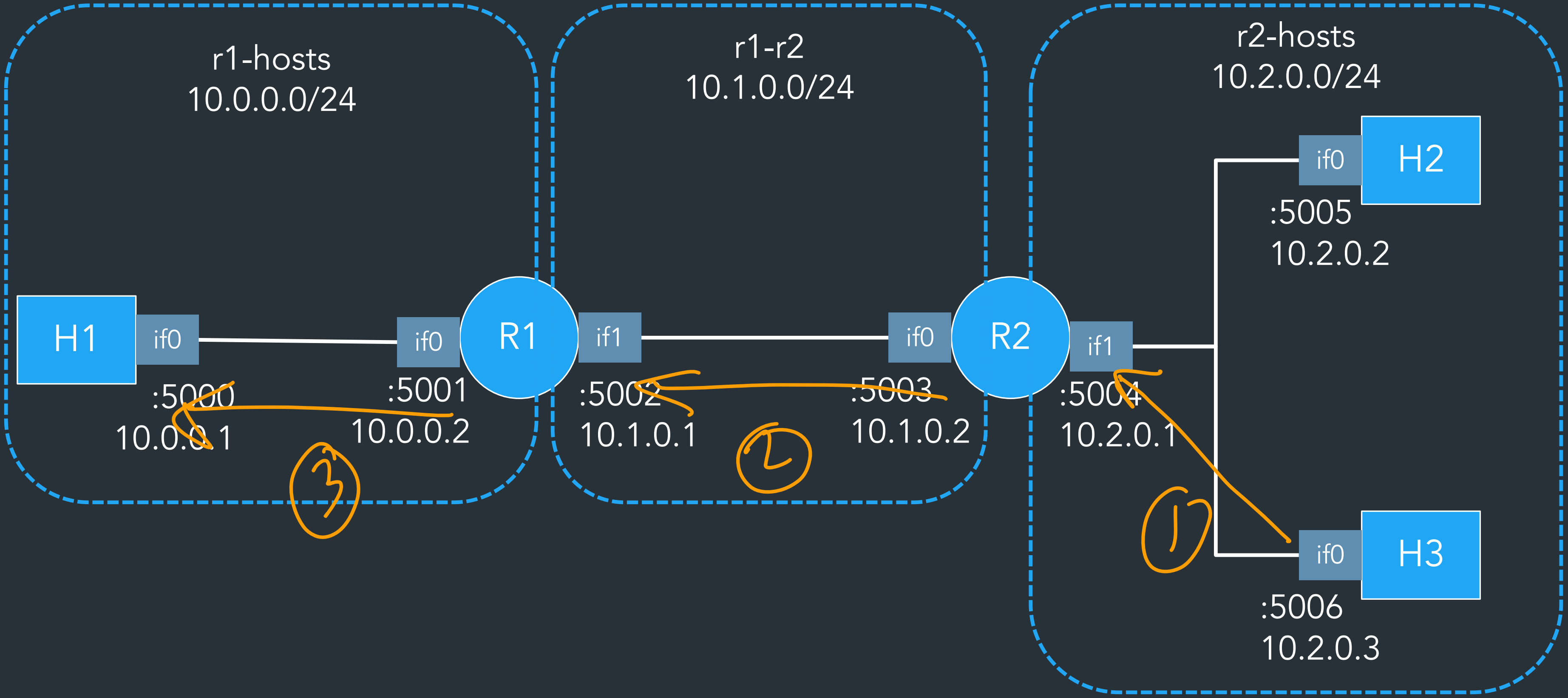


=> H1 can reach 10.0.0.2
by sending to UDP port 5001

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 }

doc-example


r1-hosts
10.0.0.0/24
r1-r2
10.1.0.0/24
r2-hosts
10.2.0.0/24
H1
if0
:5000
10.0.0.1
if0
:5001
10.0.0.2
R1
if1
:5002
10.1.0.1
if0
:5003
10.1.0.2
R2
if1
:5004
10.2.0.1
H2
if0
:5005
10.2.0.2
H3
if0
:5006
10.2.0.3
1
2
3

The Milestone

- Start by running the reference to get a feel for it
  => Setup guide online by Friday (when teams are sent out)

- For Friday (10/4): focus on sketching your high-level design for your IP stack
  - No need to have working code yet, just some serious plans/sketches
  - We'll ask you a few design questions, each graded on completion

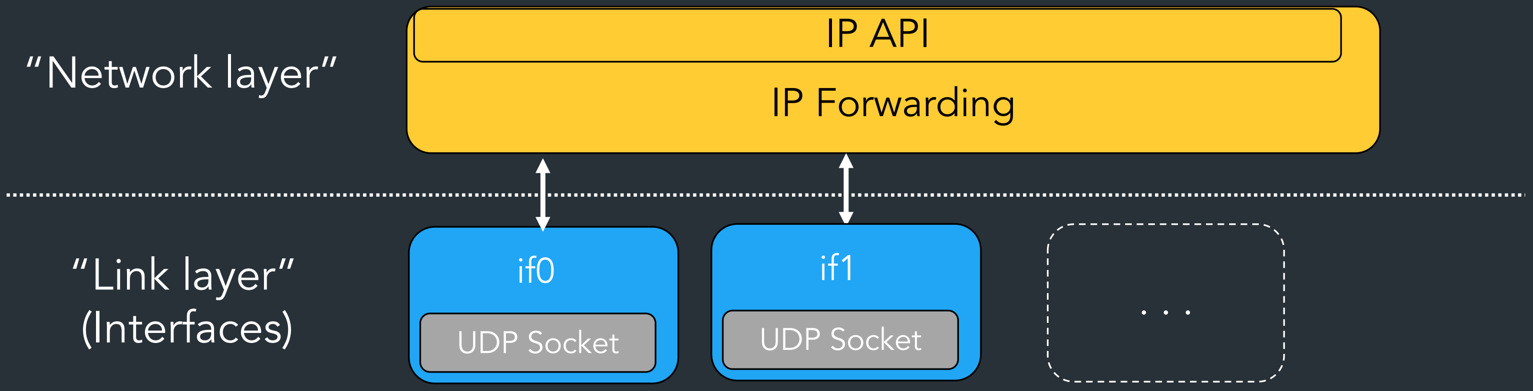
What you should be focusing on first

Focus on thinking about how you'll set up the components of your IP and link layers (what data structures, threads, etc.)

=> Link layer: one UDP socket per interface

=> IP layer: what will your forwarding table look like, how will forwarding logic use it?

=> What will your API look like for higher layers? (next page)



Your high-level API

Some key functions you want to expose for higher layers:

=> You get to decide how this works!

We suggest something like the following three components (here in pseudocode)

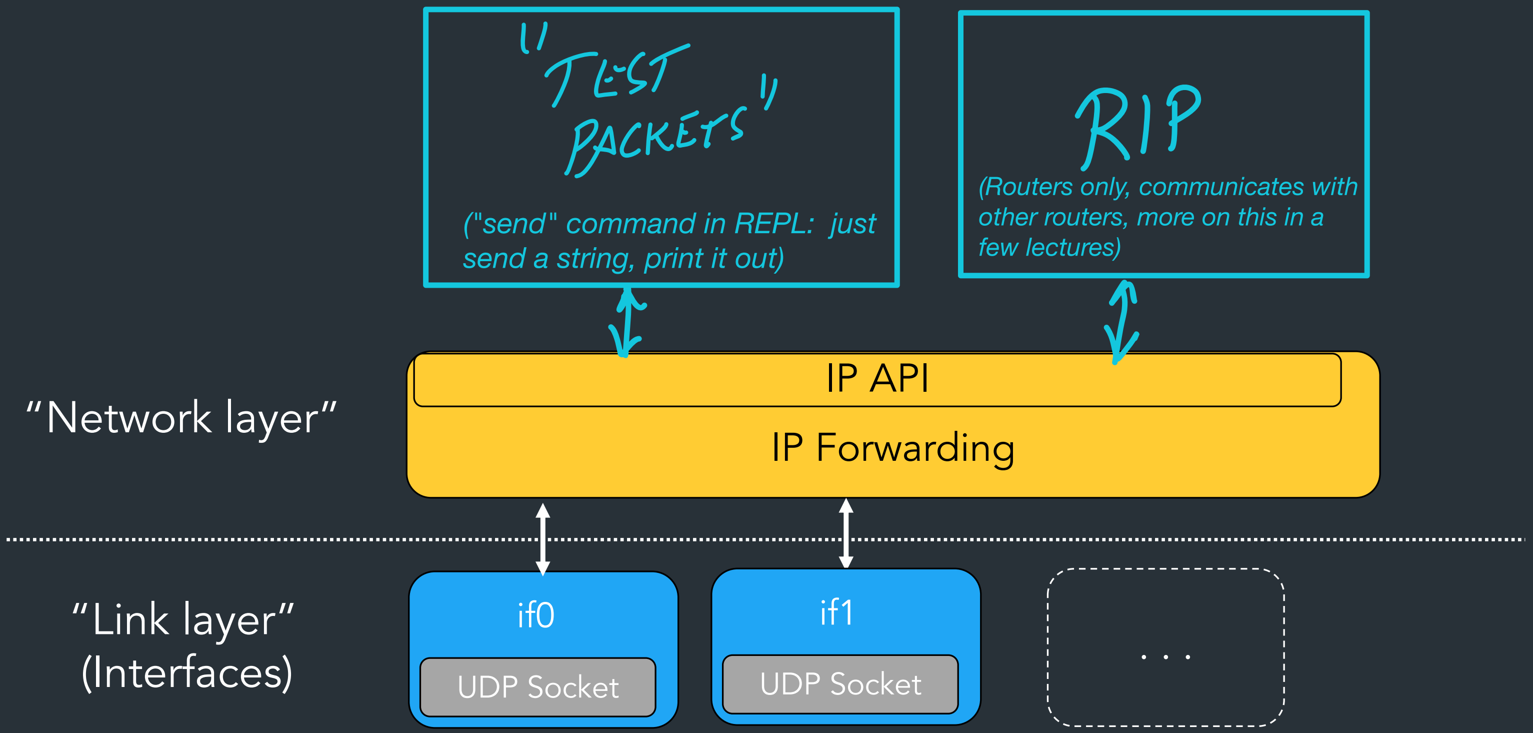
```
# Start up your IP stack
Initialize(<config struct from lnx file>)

# Send a packet to some host
SendIP(dest ip, protocolNum, []byte)

# ”Call this function when you receiving a packet”
RegisterRecvHandler(protocolNum, callbackFunc)
```

What comes next

These components will use your API in order to send/recv packets!


"TEST PACKETS"
("send" command in REPL: just send a string, print it out)
RIP
(Routers only, communicates with other routers, more on this in a few lectures)
"Network layer"
IP API
IP Forwarding
"Link layer"
(Interfaces)
if0
UDP Socket
if1
UDP Socket
. . .

Essential resources

All resources on [IP/TCP docs site]

- The handout: high level spec, grading
- Getting started guide (online soon)
- Specifications (skim now, mostly for post-milestone)
  - Lnx file structure
  - RIP specification
  - vhost/vrouter REPL commands

- Many more testing resources for later!

Implementation notes for now

- Most languages have types for IP addresses with methods you can use
  - In Go, you should use netip.Addr

- Okay to use libraries for things like data structures, parsing

Consider your software organization--you're going to be working with this code for a while, and collaborating with another person.

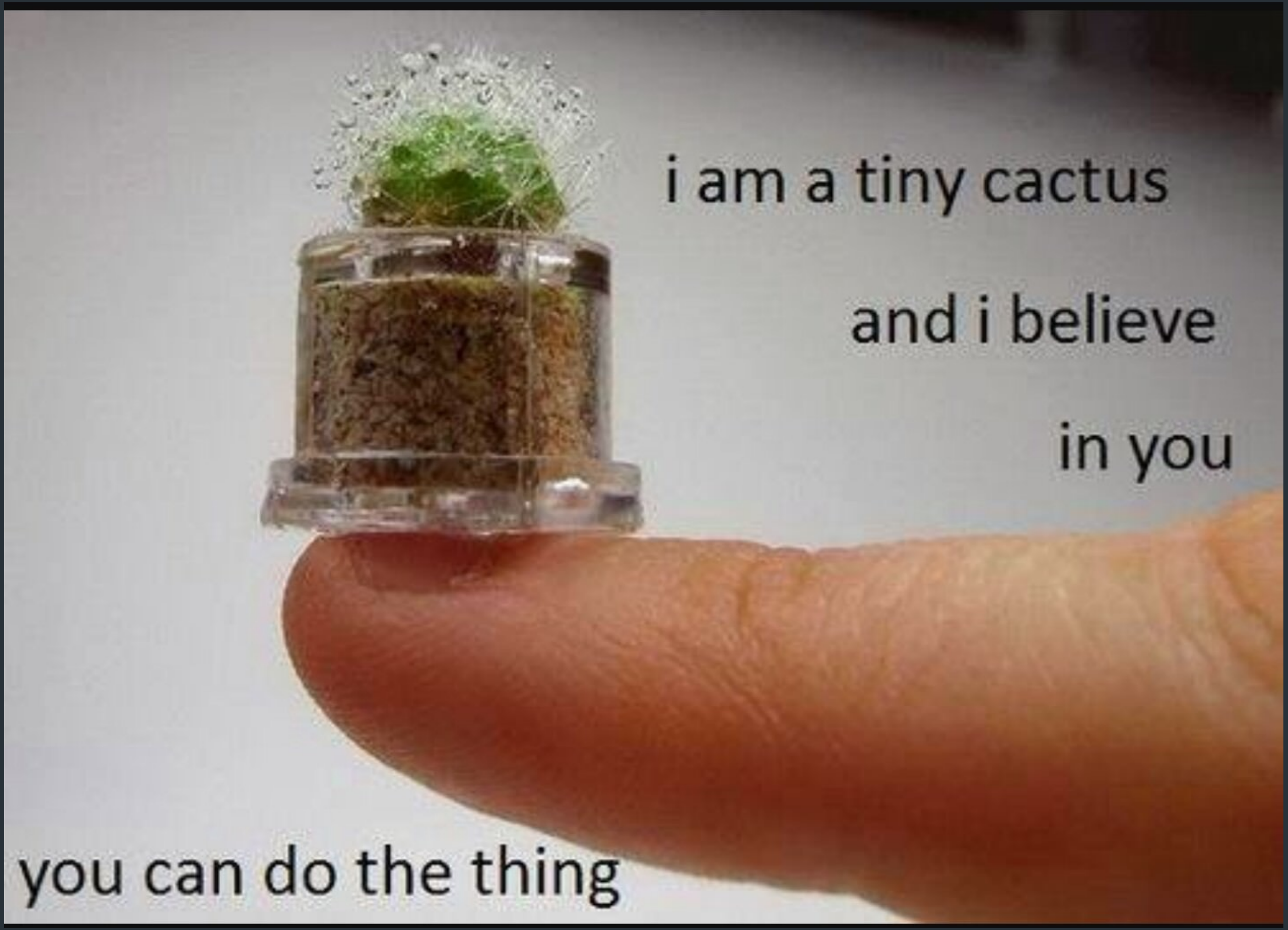
Consider what you learned from Snowcast--good software design is going to help you!
(Perhaps don't put everything in a single file, avoid magic numbers, ...)

Advice

- A lot of this project is about design. If you try to rush it, you will have problems.
  - Start early!

- Use pair programming, especially at the beginning

- You got this!


i am a tiny cactus
and i believe
in you
you can do the thing