

15-441 Spring 06
Project 2: Network Layer

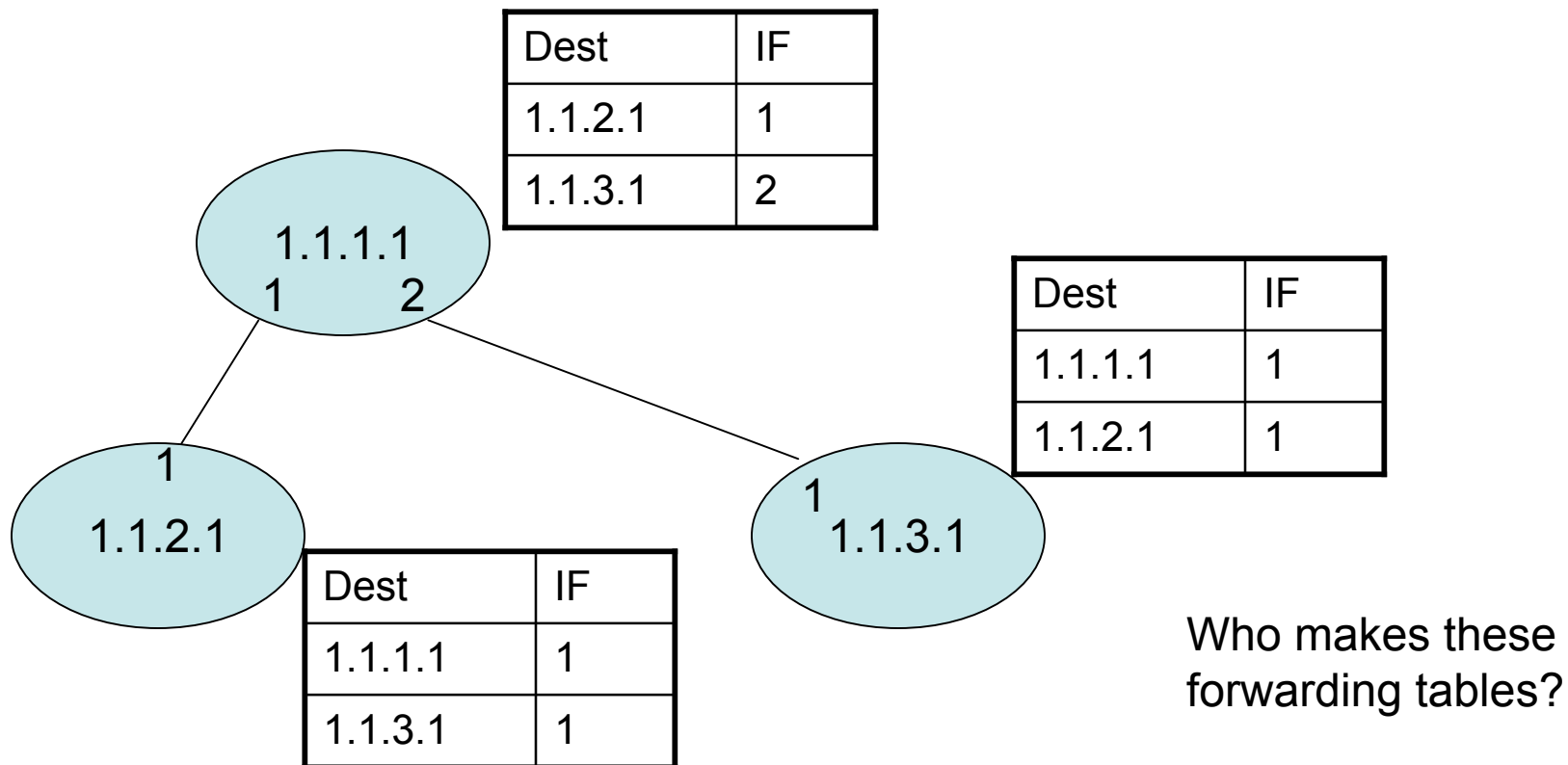
Mike Cui

Network Layer

- Gets data from host A to host B
 - Global addressing uniquely identifies A and B
 - Data finds its way to host B, directly or indirectly
 - Don't care how how they are connected, as long as they are.
- Where is host B, and how to get there?

Forwarding

- Finds host B by looking up in a forwarding table.



Routing

- Finds the paths to host B
- Fills in forwarding tables with the “best” path
- How?
 - Static
 - Manually set it (part 1)
 - Dynamic
 - Use a routing algorithm (part 2)

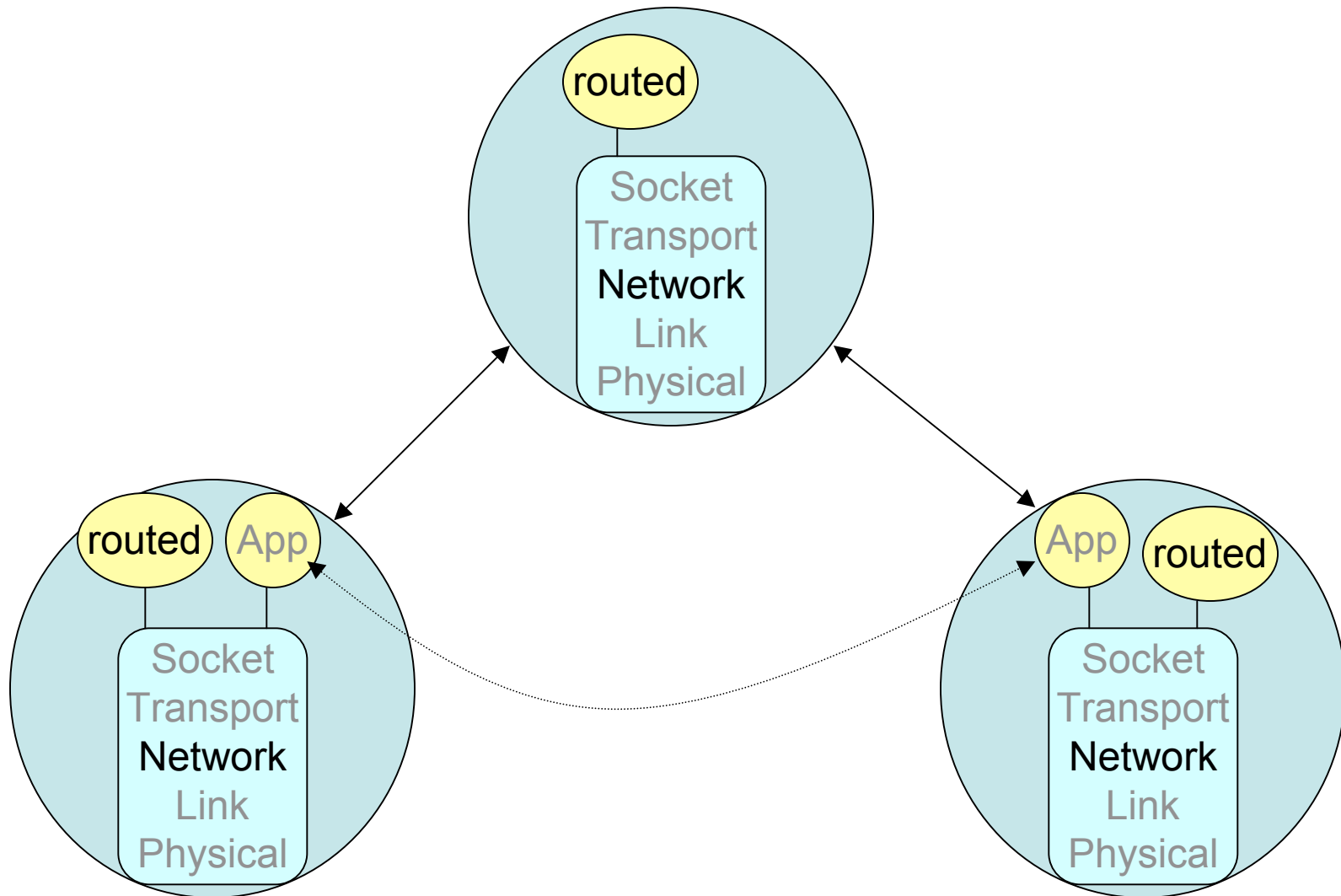
Your Mission

- Implement the network layer for a simulated operating system kernel
 - IPv4 (RFC 791)
 - Relay bytes between transport and physical layers
 - Forward packets between hosts
 - Not required: fragmentation and reassembly
- Implement a simple routing daemon
 - Using OSPF protocol

IPv4

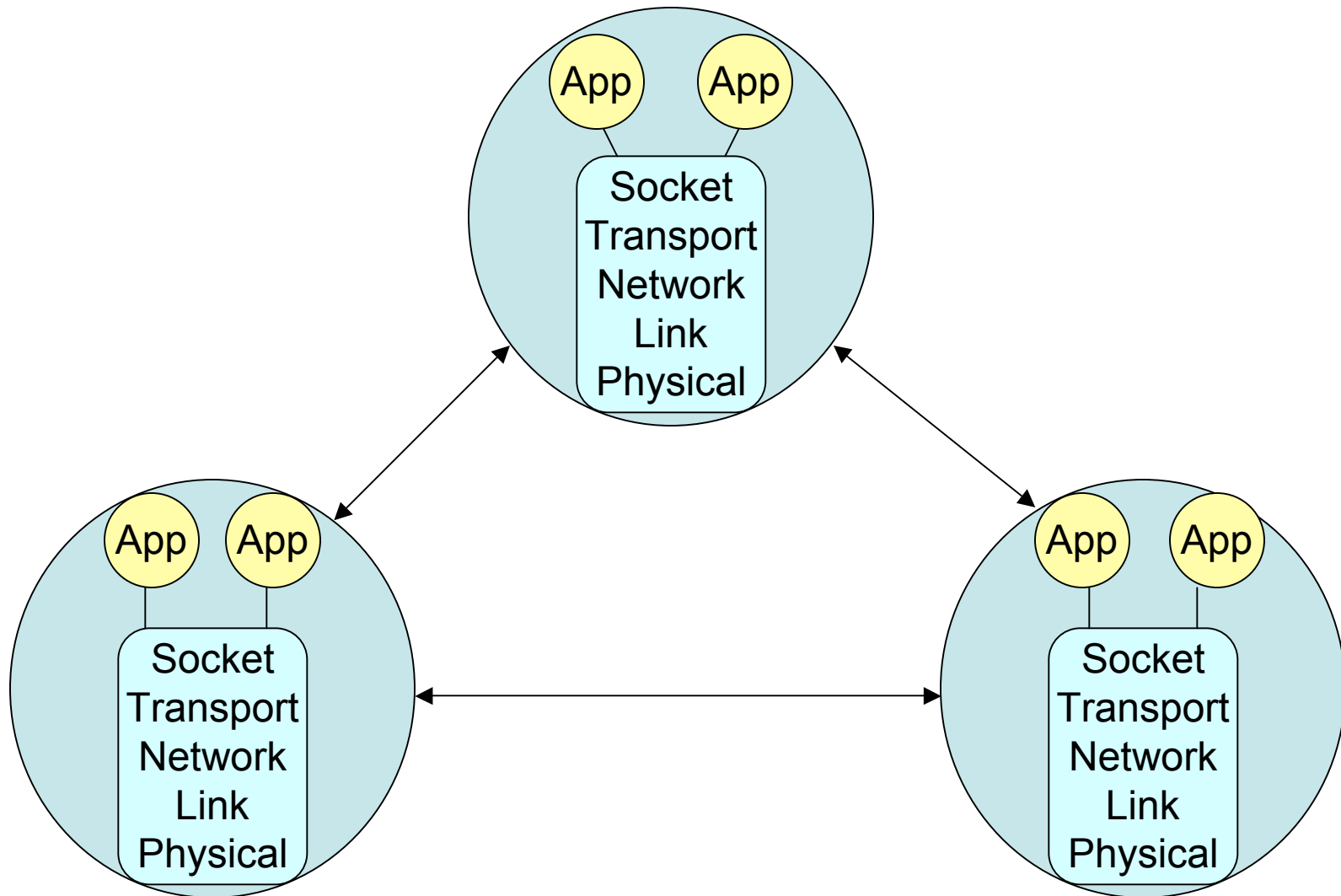
- You must handle
 - Checksum
 - Header length
 - Packet length
 - Source and destination address
 - Protocol number
 - TTL
 - IP version number
- Your implementation need not handle
 - Fragmentation
 - Options
 - Multicast/broadcast
 - ToS

When You Are All Done

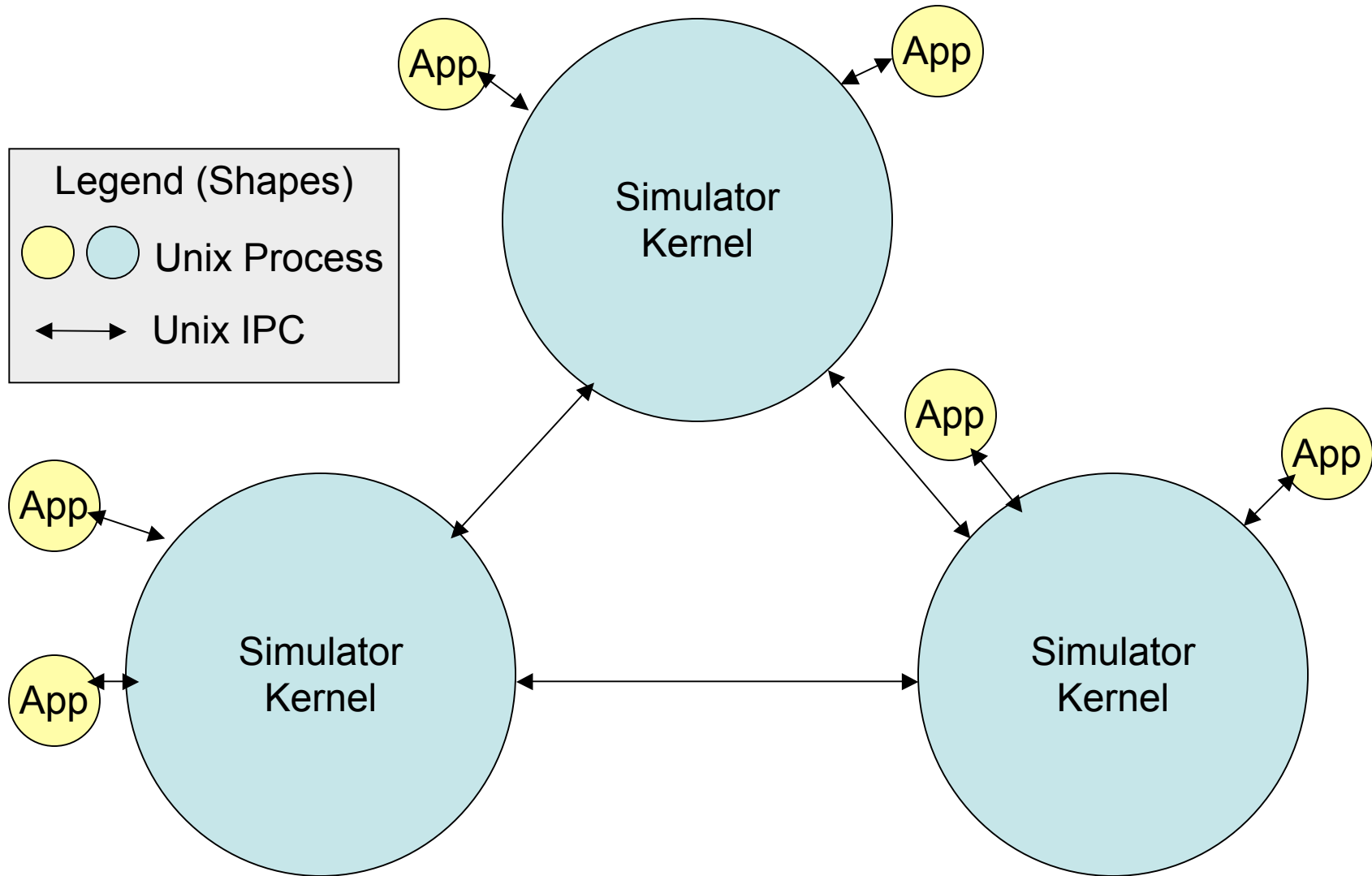




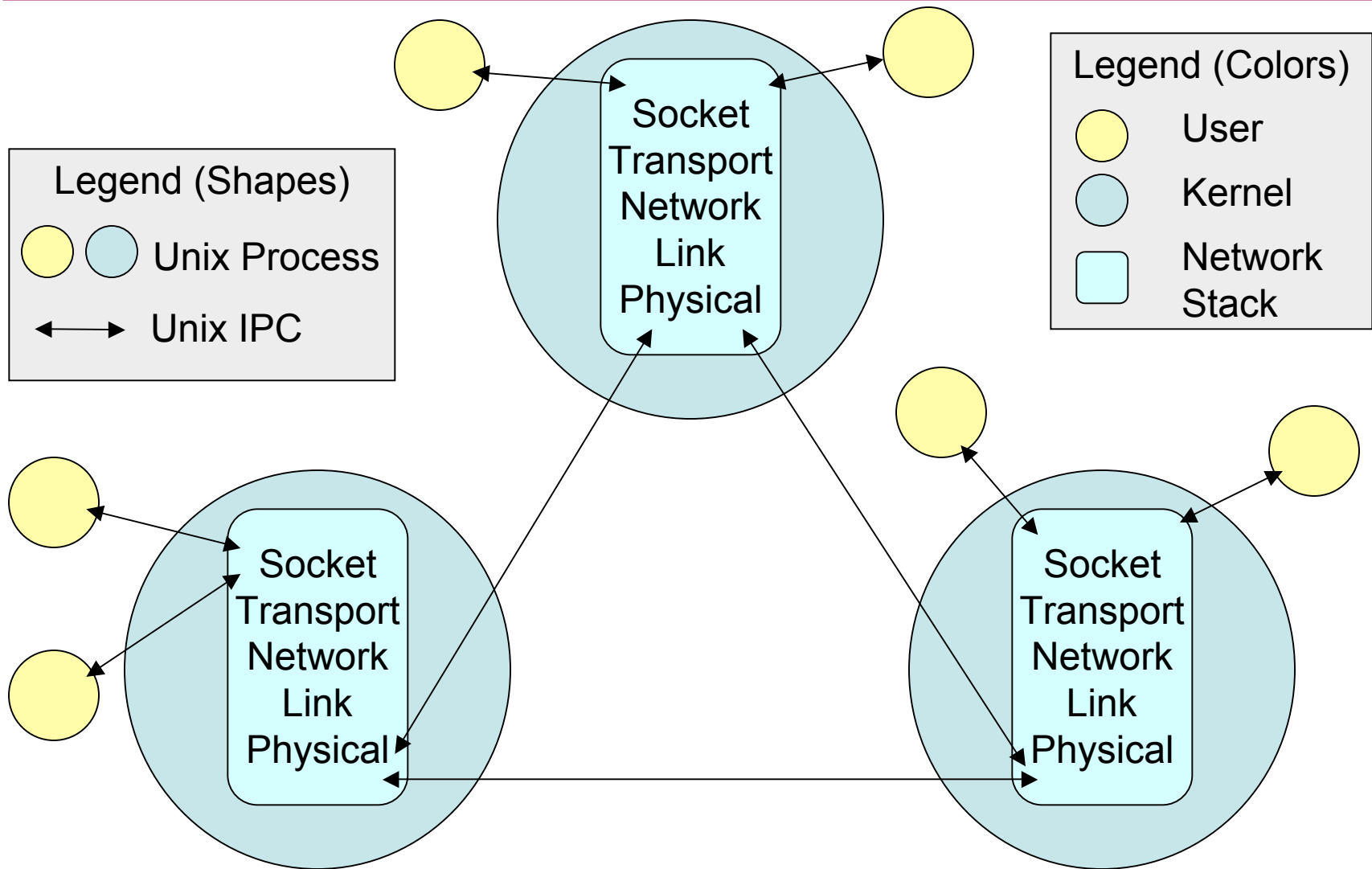
Simulator: Logical View



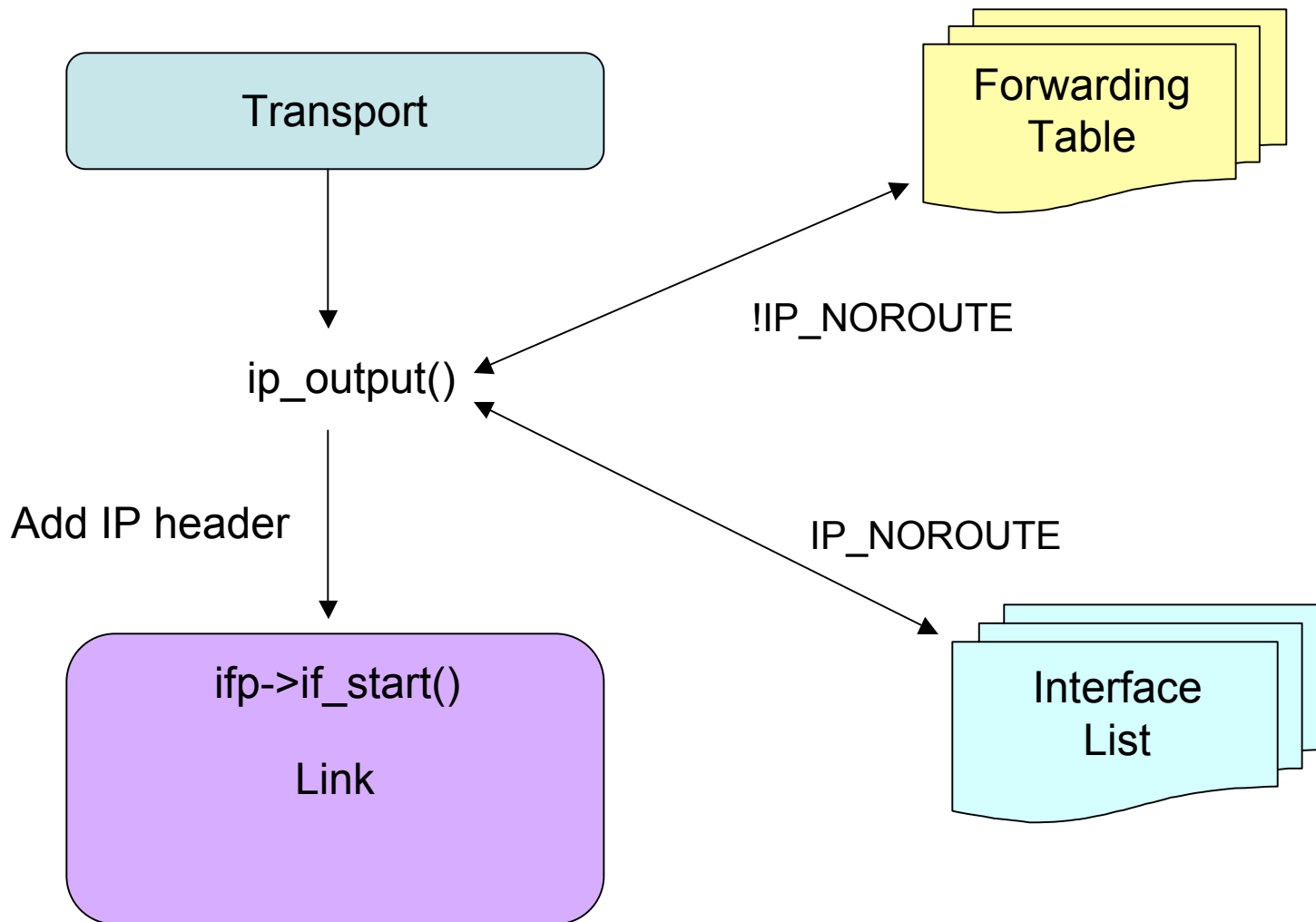
Simulator: Implementation



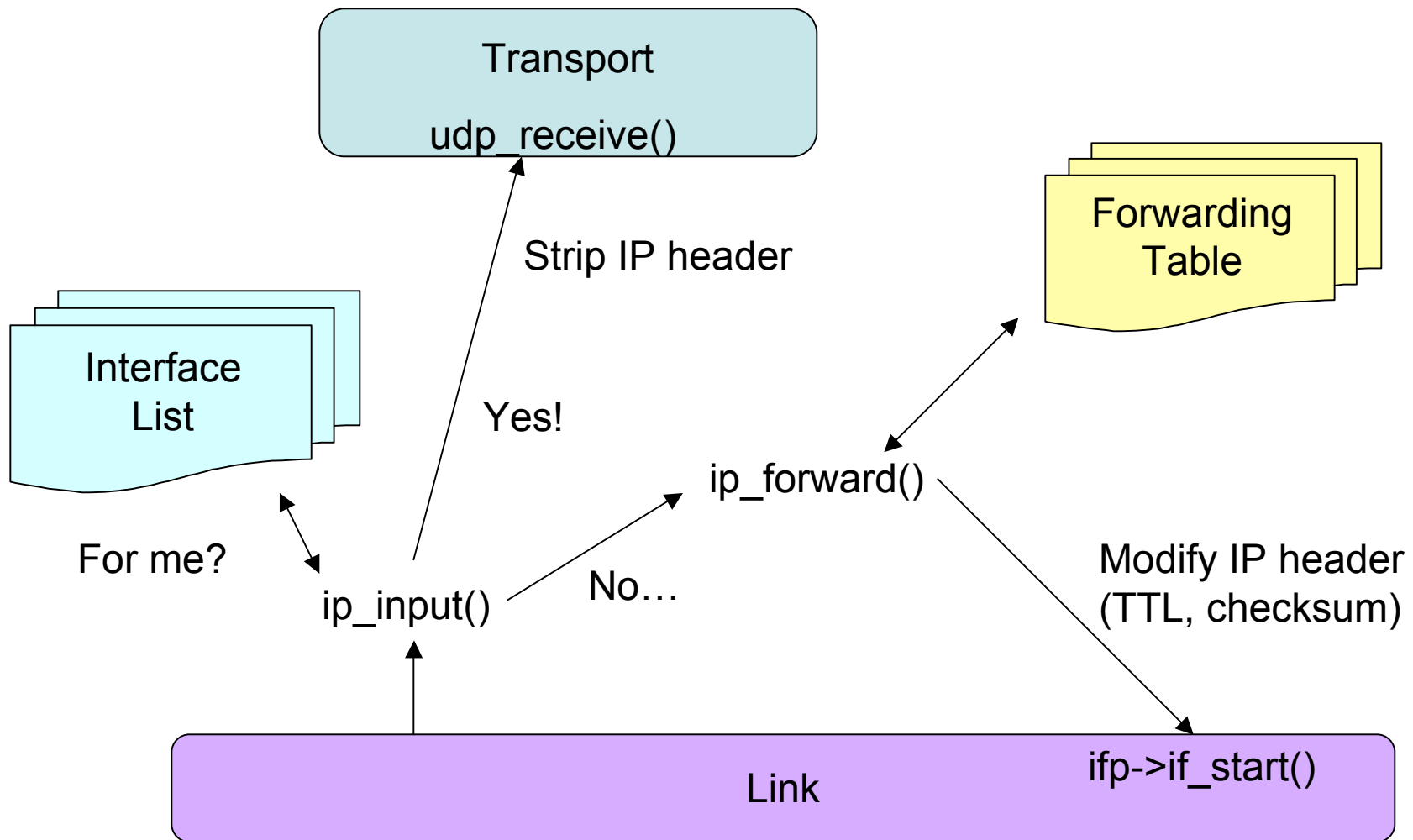
Simulator: Full Picture



Sending a Packet



Receiving a Packet



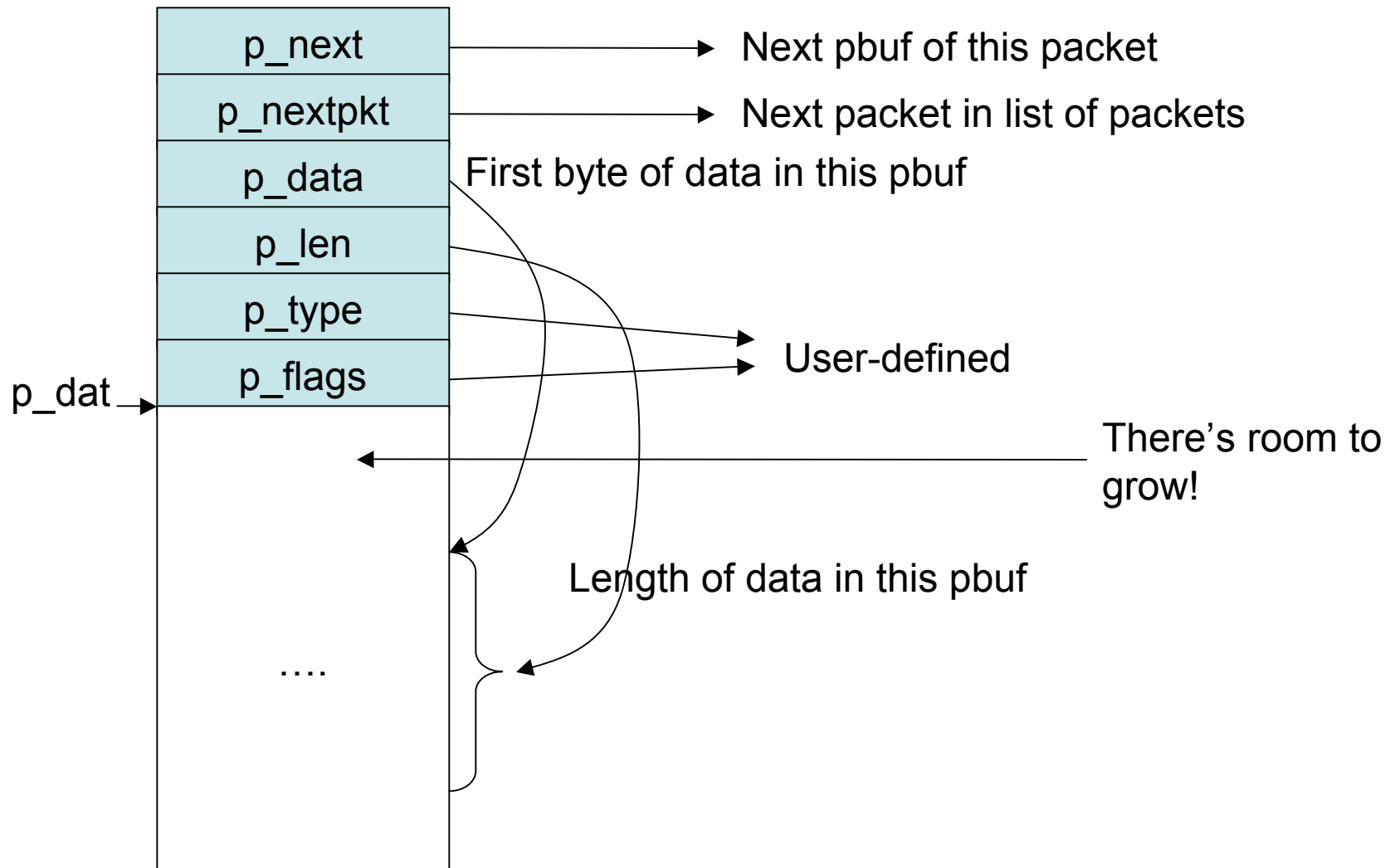
pbuf

- Linked list of fixed-size (512 byte) buffers

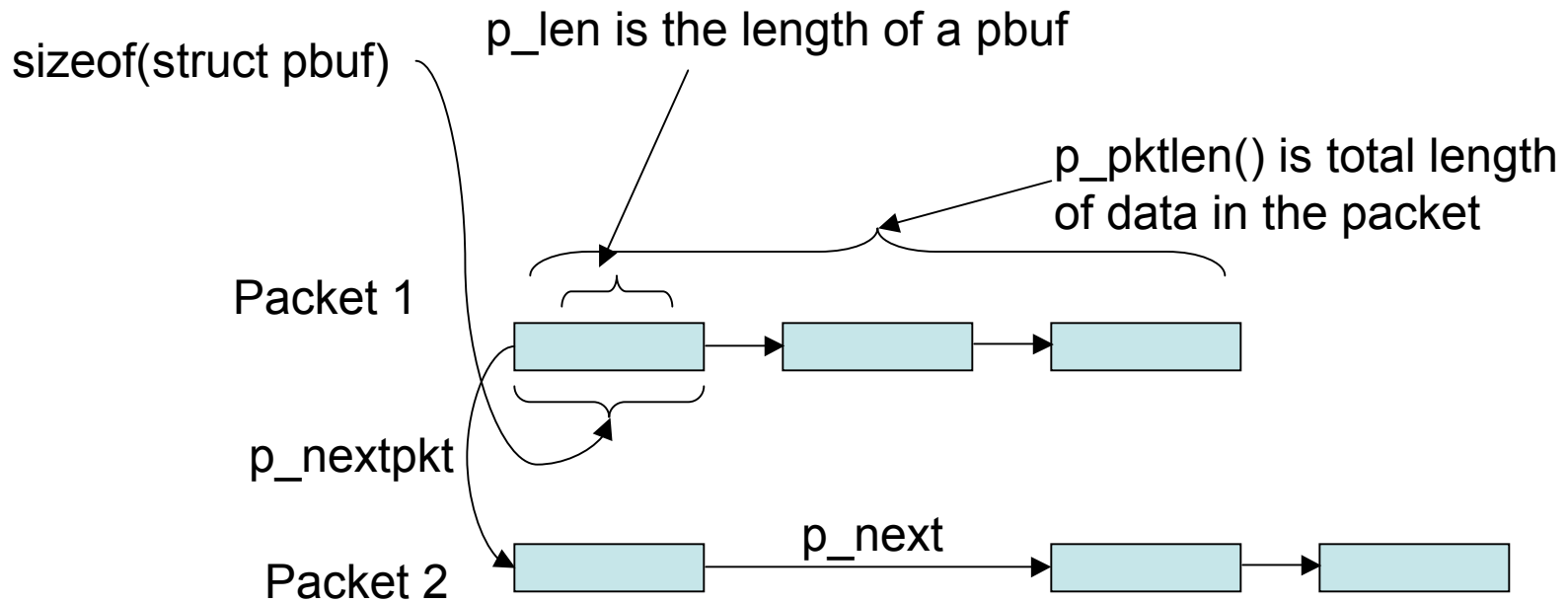


- Why?
 - Dynamic, variable-sized memory allocation is expensive
 - Takes time, function of size of heap
 - Wastes space due to fragmentation

Inside a pbuf



pbuf Chain



IP Interface

- When `ip_input()` is called...
 - `p_data` points to beginning of IP header
 - Strip off IP header before passing onto transport layer
- When `ip_output()` is called...
 - `p_data` points to beginning of IP payload
 - Prepend an IP header before handing to link layer
 - Can assume there's enough room to prepend a IP header
 - Should not need to allocate more pbufs
- Helper functions in `pbuf.h`
 - `p_prepend`, `p_strip`

Connecting to the Simulator

- `#include <project2/include/Socket.h>`
- **Use Socket-API functions with first letter capitalized**
 - `Socket()`, `Bind()`, `Connect()` ...
- **Make sure to Close()**
 - Simulator isn't an operating system, it doesn't clean up after you

Testing Your Network Layer

- Use `fdconfig` to set up static routes
- Try UDP applications
 - `unreliable-server`,
`unreliable-client`
 - Your P1 TFTP server (single client)

Simulator Internals

- Multithreaded implementation
 - System call interface
 - User processes must register with simulator
 - One thread to handle registration requests
 - One thread per process to handle system calls
 - Network devices
 - One thread per network device
 - Wakes up when packet arrives
- What does this mean for you?
 - Your code will be executed by multiple threads..
 - Your code must be re-entrant!

Concurrency Reminder

- What you think

```
ticket = next_ticket++; /* 0 ⇒ 1 */
```

- What really happens (in general)

```
ticket = temp = next_ticket; /* 0 */  
++temp; /* invisible to other threads */  
next_ticket = temp; /* 1 is visible */
```

Murphy's Law (Of Threading)

- The world may *arbitrarily interleave* execution
 - Multiprocessor
 - N threads executing instructions *at the same time*
 - Of course effects are interleaved!
 - Uniprocessor
 - Only one thread running at a time...
 - But N threads runnable, timer counting down toward zero...
- The world will choose the most painful interleaving
 - “Once chance in a million” happens every minute

Your Hope

| T0 | | T1 | |
|-------------------------------|---|-------------------------------|---|
| <code>tk = tmp = n_tk;</code> | 0 | | |
| <code>++tmp;</code> | 1 | | |
| <code>n_tk = tmp;</code> | 1 | | |
| | | <code>tk = tmp = n_tk;</code> | 1 |
| | | <code>++tmp;</code> | 2 |
| | | <code>n_tk = tmp</code> | 2 |
| Final Value | 1 | | 2 |

Your Bad Luck

| T0 | | T1 | |
|---------------------------------|---|---------------------------------|---|
| <code>tkt = tmp = n_tkt;</code> | 0 | | |
| | | <code>tkt = tmp = n_tkt;</code> | 0 |
| <code>++tmp;</code> | 1 | | |
| | | <code>++tmp;</code> | 1 |
| <code>n_tkt = tmp;</code> | 1 | | |
| | | <code>n_tkt = tmp</code> | 1 |
| Final Value | 1 | | 1 |

Two threads have the same “ticket” !

What To Do

- Use mutual exclusion

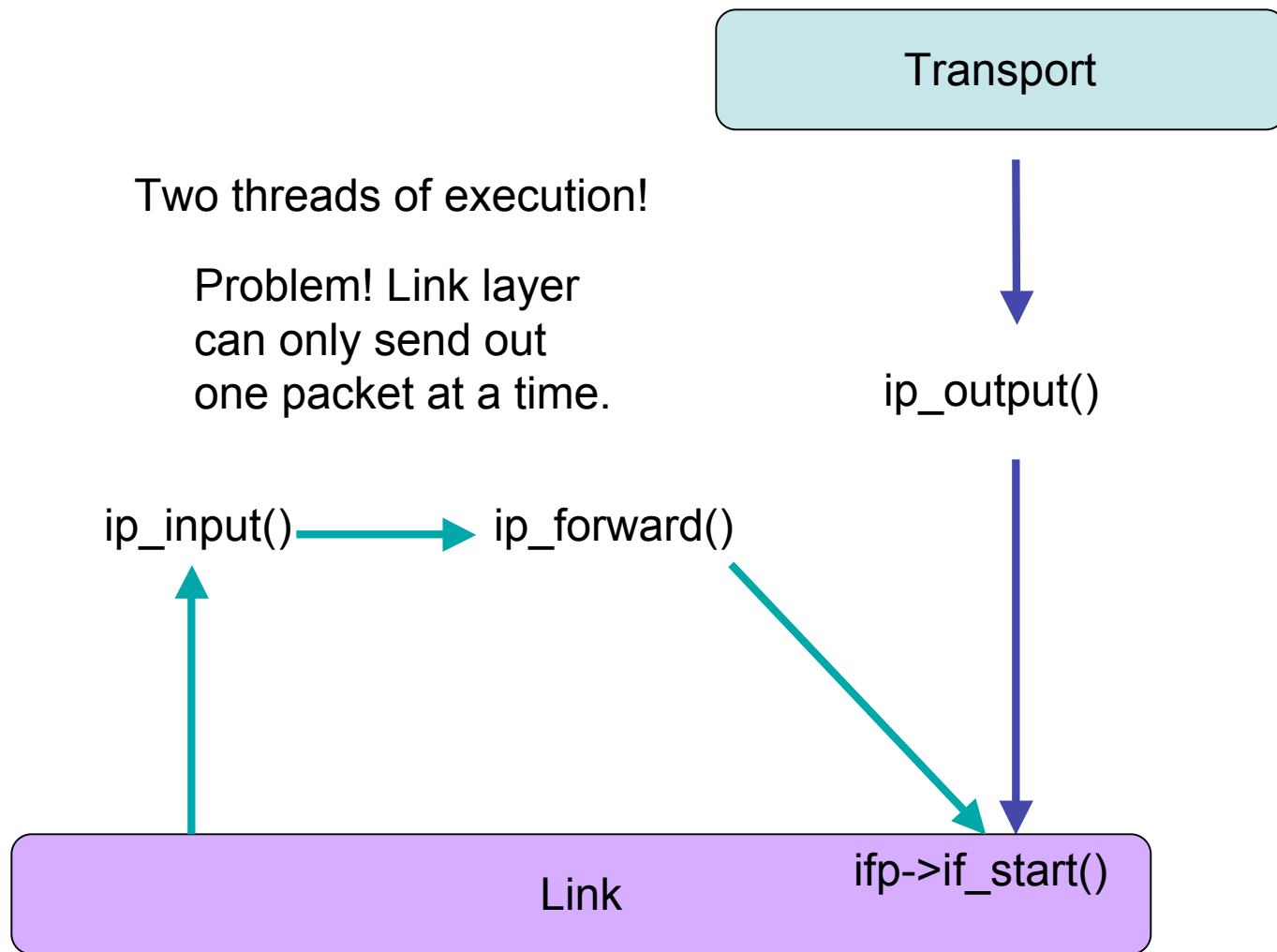
```
MUTEX_LOCK(m);
```

```
ticket = next_ticket++;
```

```
MUTEX_UNLOCK(m);
```

- Now no other thread's execution of the “critical section” can be interleaved with yours

IP Dataflow Revisited



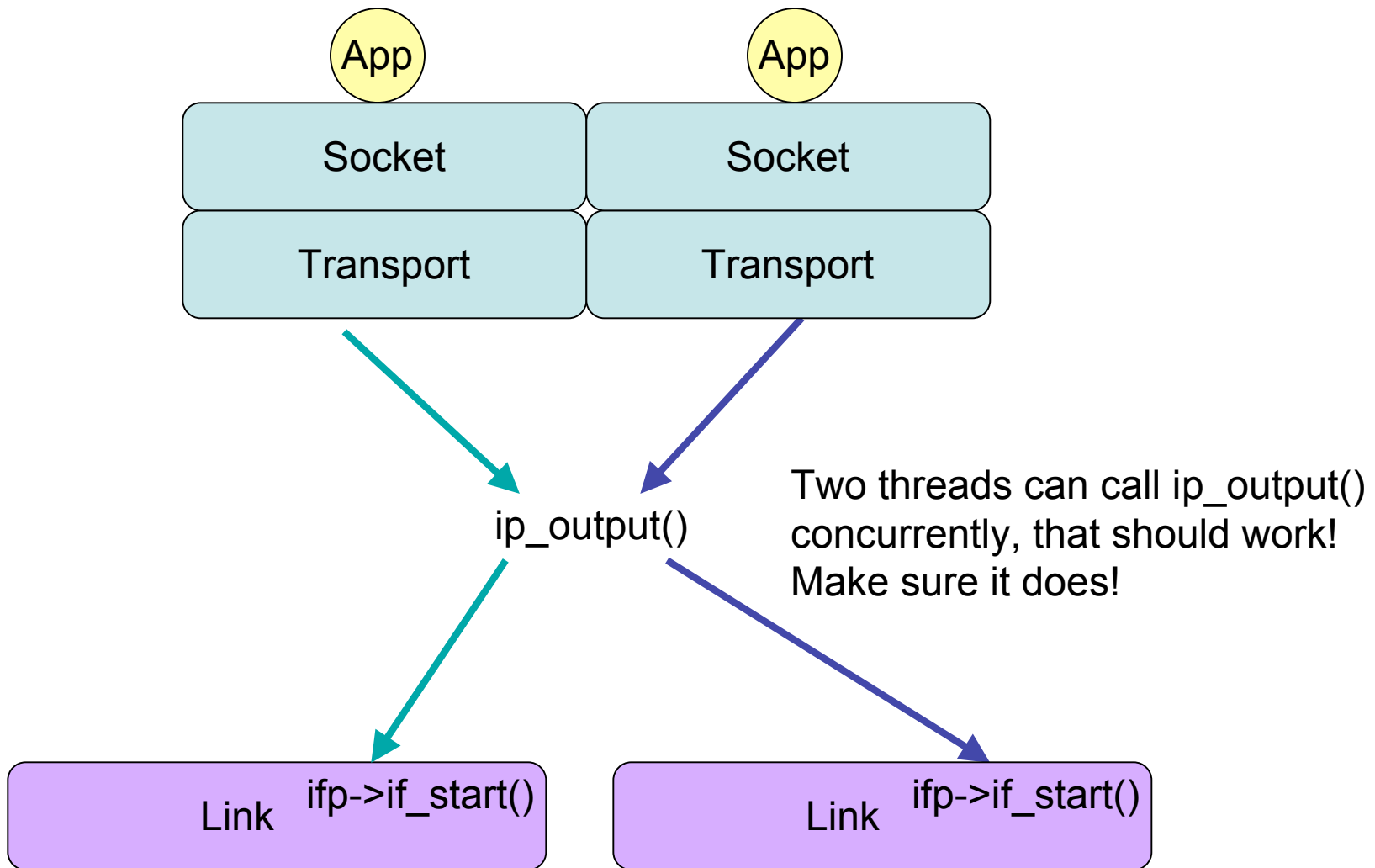
One at a Time

- Only one thread can send through a particular device at a time
 - Otherwise the device will fail and cause kernel to panic.
- Need mutual exclusion
 - Use a mutex (`pthread_mutex_t`) for each device
 - Declared in `if.h`, Mutex wrappers in `sync.h`

```
MUTEX_LOCK(&ifp->if_mutex);  
ifp->start(ippkt);  
...  
MUTEX_UNLOCK(&ifp->if_mutex);
```

Critical section! Mutex ensures that only one thread can enter at a time.

IP Dataflow Revisited (again)



Many at a Time

- More than one thread could invoke an IP layer function at the same time
 - Each invocation has to be independent of one another
 - Each invocation needs to have its own state
 - Stack variables are independent, global variables are shared
 - Shared state needs to be protected

Debugging Multiple Threads

- Using `gdb`
 - `info thread` lists all running threads
 - `thread n` switches to a specific thread
 - `bt` to get stack trace for the current thread
 - Look for the function `thread_name` in stack trace, name of thread is in the argument
 - “link n:i to k:j” device thread for interface i on node n to interface j on node k
 - “user_pid” system call handling thread for user process pid.