CS162: Operating Systems and Systems Programming

Lecture 19: Networking: Sockets/TCP/IP

22 July 2015

Charles Reiss

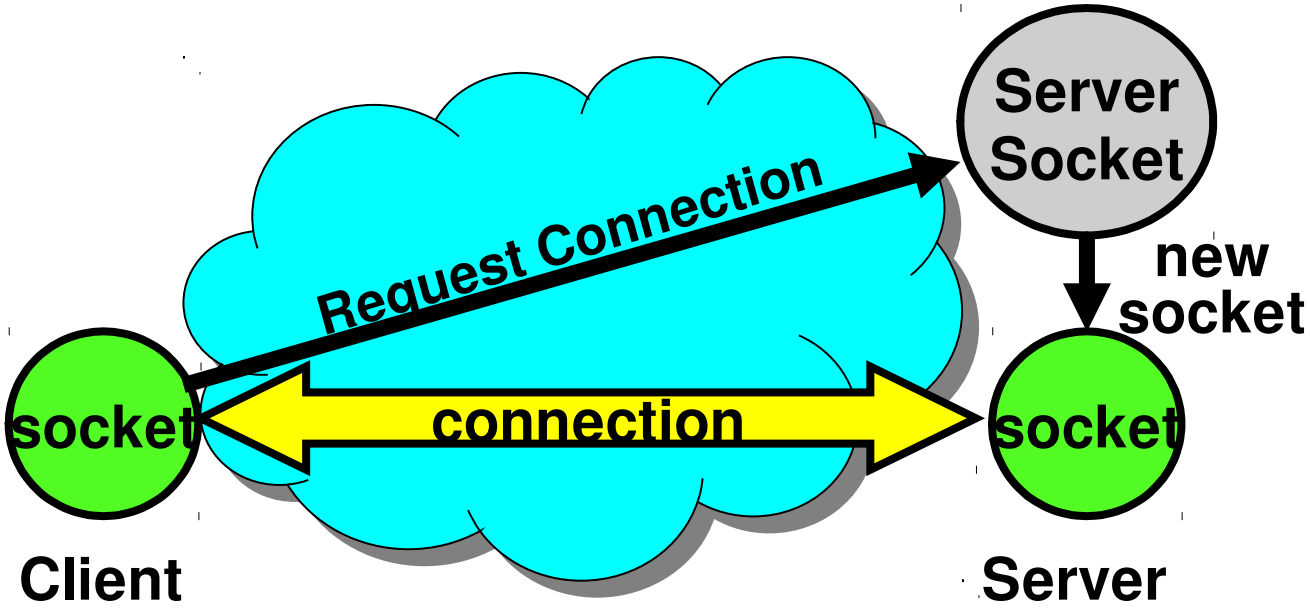
https://cs162.eecs.berkeley.edu/

Recall: Echo client-server code

```
void client(int sockfd) {
  int n; char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  while (getreq(sndbuf, MAXIN)) {    /* prompt + read */
    write(sockfd, sndbuf, strlen(sndbuf));    /* send */
    n=read(sockfd, rcvbuf, MAXOUT-1);      /* receive */
    if (n <= 0) return;       /* handle error  or EOF */
    write(STDOUT_FILENO, rcvbuf, n);          /* echo */
  }
}
```

```
void server(int consockfd) {
  int n; char reqbuf[MAXREQ];
  while (1) {
    n = read(consockfd,reqbuf,MAXREQ-1);           /* Recv */
    if (n <= 0) return;              /* handle error or EOF */
    n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
    n = write(consockfd, reqbuf, strlen(reqbuf));  /* echo */
  }
}
```

Recall: Socket Setup over TCP/IP



Special kind of socket: ***server socket***

- Has file descriptor
- Can't read or write

Two operations:

- listen(): start allowing clients to connect
- accept(): create a *new socket* for a particular client

Recall: Client Protocol

```
char *hostname; char *portname;
int sockfd;
struct addrinfo *server;
server = buildServerAddr(hostname, portname);

/* Create a TCP socket */
/* server->ai_family: AF_INET (IPv4) or AF_INET6 (IPv6) */
/* server->ai_socktype: SOCK_STREAM (byte-oriented) */
/* server->ai_protocol: IPPROTO_TCP */
sockfd = socket(server->ai_family, server->ai_socktype,
                server->ai_protocol)
/* Connect to server on port */
connect(sockfd, server->ai_addr, server->ai_addrlen);

/* Carry out Client-Server protocol */
client(sockfd);

/* Clean up on termination */
close(sockfd);
freeaddrinfo(server);
```

Recall: Server Protocol (v1)

```
/* Create Socket to receive requests*/
lstnsockfd = socket(server->ai_family, server->ai_socktype,
                    server->ai_protocol);

/* Bind socket to port */
bind(lstnsockfd, server->ai_addr, server->ai_addrlen);
while (1) {
/* Listen for incoming connections */
   listen(lstnsockfd, MAXQUEUE);

/* Accept incoming connection, obtaining a new socket for it */
   consockfd = accept(lstnsockfd, NULL, NULL);

   server(consockfd);

   close(consockfd);
  }
close(lstnsockfd);
```

Handling multiple connectoins

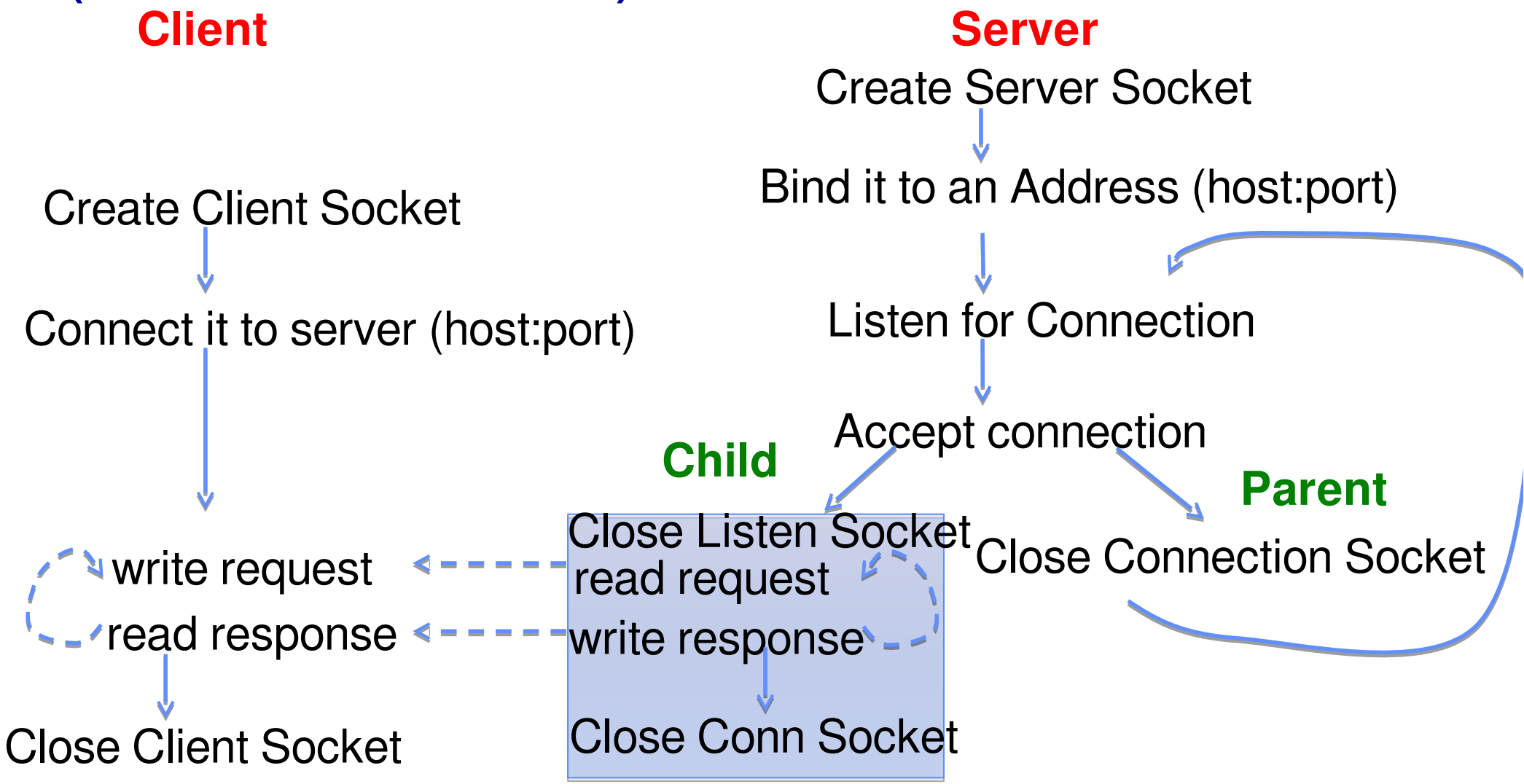
One option – fork a process for each connection

- Strong isolation between each connection
- Can accept new connections while other connections are active

Second option – spawn a thread for each connection

Third option – event-style (later)

Process Per Connection (One at a Time)



Server Protocol (v3)

```
while (1) {
    listen(lstnsockfd, MAXQUEUE);
    consockfd = accept(lstnsockfd, NULL, NULL);
    cpid = fork();              /* new process for connection */
    if (cpid > 0) {               /* parent process */
      close(consockfd);
      //tcpid = wait(&cstatus); /* Ignore SIGCHLD instead? */
    } else if (cpid == 0) {       /* child process */
      close(lstnsockfd);          /* let go of listen socket */

      server(consockfd);

      close(consockfd);
      exit(EXIT_SUCCESS);           /* exit child normally */
    }
}
```

Client: getting the server address

```
struct addrinfo *buildServerAddr(char *hostname, char *portname)
{
  struct addrinfo *result;
  struct addrinfo hints;
  int rv;
  memset(&hints, 0, sizeof(hints)); /* Clean unused hints */
  hints.ai_family = AF_UNSPEC;      /* IPv4 or IPv6 */
  hints.ai_socktype = SOCK_STREAM;  /* Stream socket -
                                       TCP / byte-oriented */

  rv = getaddrinfo(hostname, portname, &hints, &result);

  if (rv != 0) {
    /* handle error */
  }
  return result;
}

/* Later freeaddrinfo(result) */
```

Server address: with getaddrinfo

```
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints)); /* Clean unused hints */
hints.ai_family = AF_UNSPEC;      /* IPv4 or IPv6 */
hints.ai_socktype = SOCK_STREAM;  /* Stream socket -
                                     TCP / byte-oriented */
hints.ai_flags = AI_PASSIVE;      /* for listening */

rv = getaddrinfo(NULL /* hostname */, portname, &hints, &result);

if (rv != 0) {
  /* handle error */
}


/* Later freeaddrinfo(result) */
```

Server address: manually (IPv4)

```
int port_number = …;
struct addrinfo server;
struct sockaddr_in server_ip_port;
server_ip_port.sin_family = AF_INET; /* IPv4 */
server_ip_port.sin_addr.s_addr = INADDR_ANY;
   /* 0.0.0.0 - means all addresses available on the
machine */
server_ip_port.sin_port = htons(port_number);
   /* htons → host to "network" (Big Endian) order short */

server.ai_addr = (struct sockaddr*) &server_ip_port;
server.ai_addrlen = sizeof(struct sockaddr_in);
server.ai_family = AF_INET;
server.ai_socktype = SOCK_STREAM; /* byte-oriented */
server.ai_protocol = IPPROTO_TCP; /* or 0 - "choose any" */
```

Recall: Peer-to-Peer Communication

No always-on server at the center of it all

- Hosts can come and go, and change addresses
- Hosts may have a different address each time

Example: peer-to-peer file sharing (e.g., BitTorrent)

- Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
- Scalability by harnessing millions of peers
- Each peer acting as both a ***client and server***

Networking Definitions

Network: physical connection that allows two computers to communicate

Frame/Packet/Segment: unit of transfer, sequence of bits carried over the network

- Network carries packets from one CPU to another
- Destination gets interrupt when frame arrives
- Name depends on what layer (later)

Protocol: agreement between two parties as to how information is to be transmitted

The Problem (1)

Many different applications

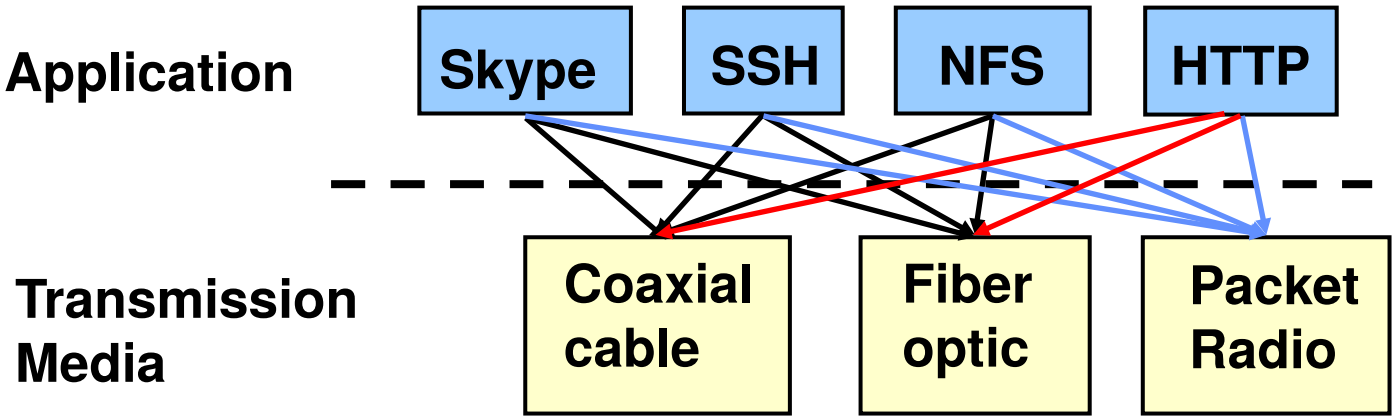
- email, web, P2P, etc.

Many different network styles and technologies

- Wireless vs. wired vs. optical, etc.

How do we organize this mess?

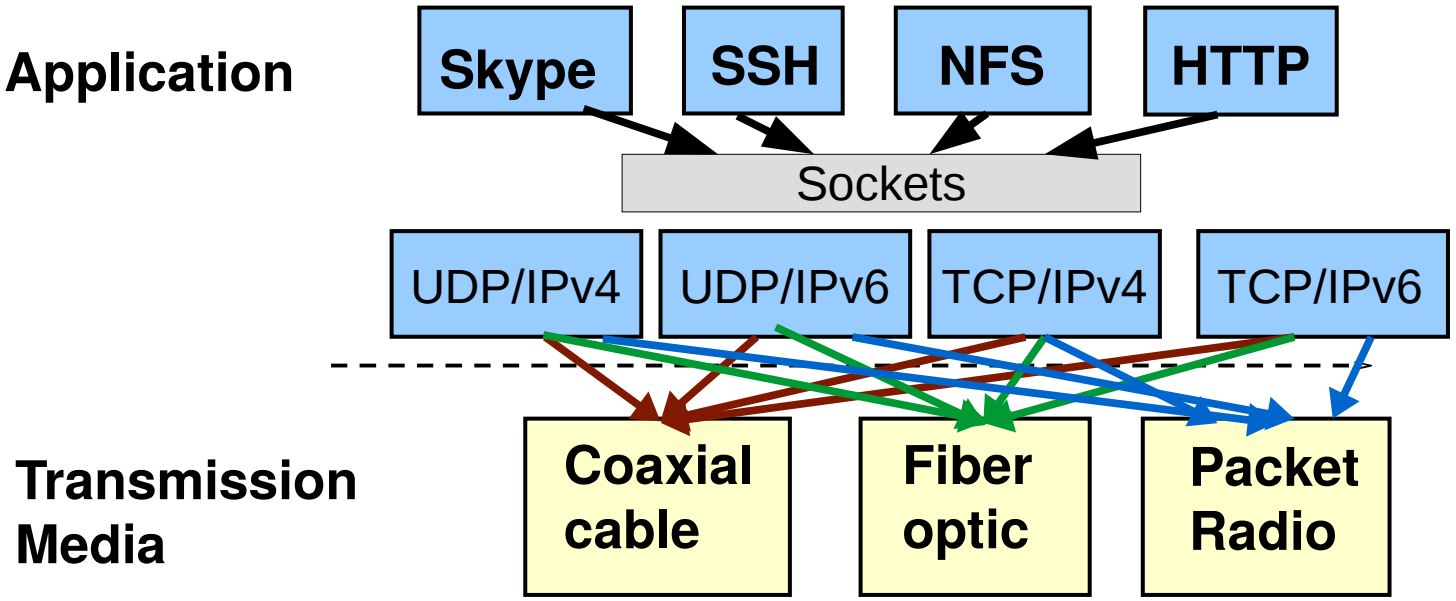
The Problem (2)



Re-implement every application for every technology?

No!

The Problem (3)



Re-implement every type of sockets for every technology?

No.

Layering

Complex services from simpler ones

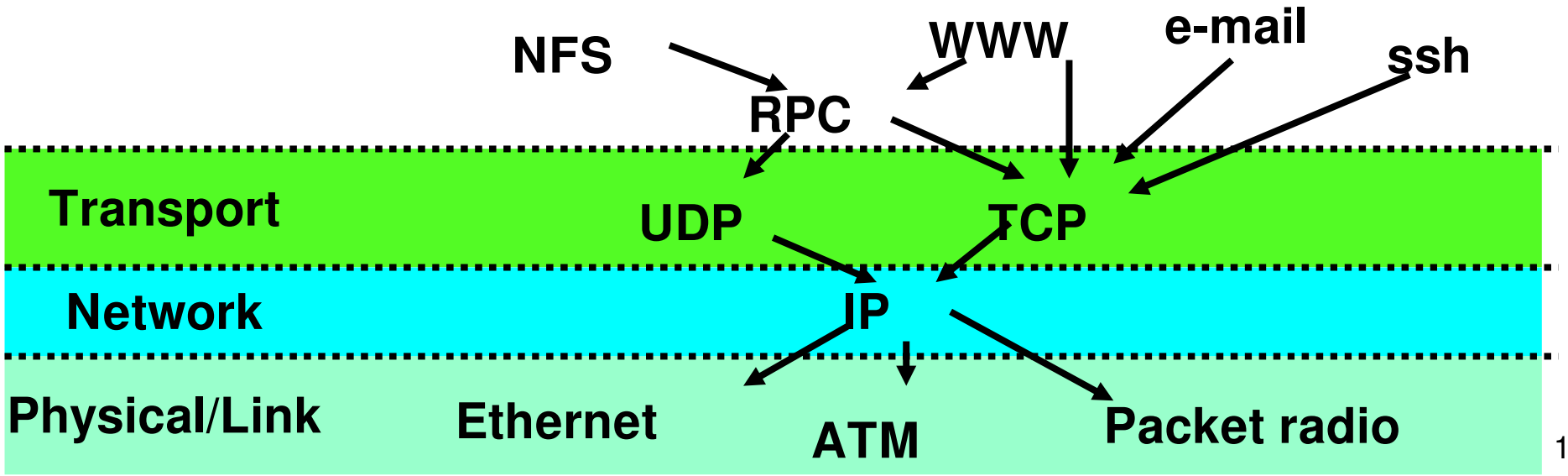
TCP/IP networking layers:

- Physical + Link (Wireless, Ethernet, ...)
  - Unreliable, local exchange of limited-sized **frames**
- Network (IP) – routing between networks
  - Unreliable, global exchange of limited-sized **packets**
- Transport (TCP, UDP) – routing
  - Reliablity, streams of bytes instead of packets, …
- Application – everything on top of sockets

Network Protocols

Protocol: Agreement between two parties as to how information is to be transmitted

Protocols on today's Internet:



Layering

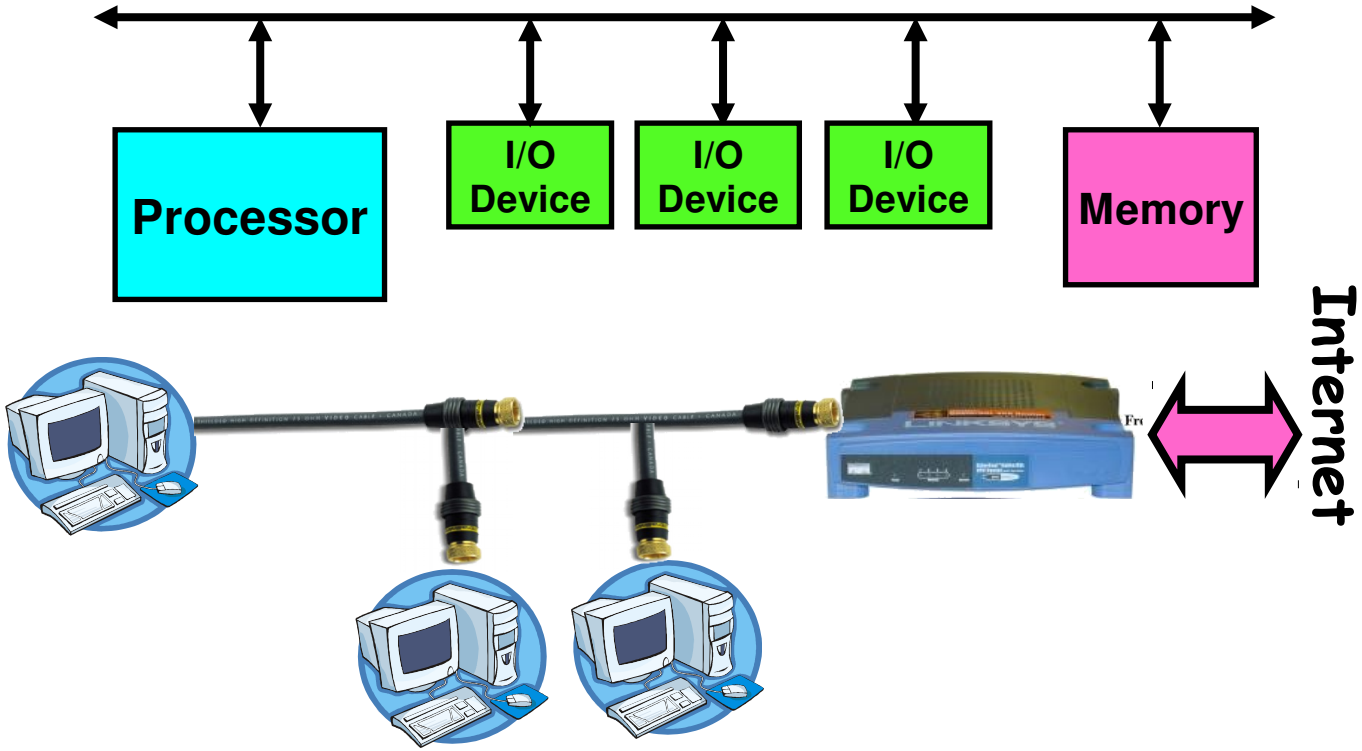
Complex services from simpler ones

TCP/IP networking layers:

- **Physical + Link (Wireless, Ethernet, ...)**
  - Unreliable, local exchange of limited-sized **frames**
- Network (IP) – routing between networks
  - Unreliable, global exchange of limited-sized **packets**
- Transport (TCP, UDP) – glue
  - Reliability (retry), ordering, streams of bytes, ...
- Application – everything on top of sockets

Broadcast Networks

Shared Communication Medium



Shared Medium can be a set of wires
Inside a computer, this is called a bus
All devices simultaneously connected to devices

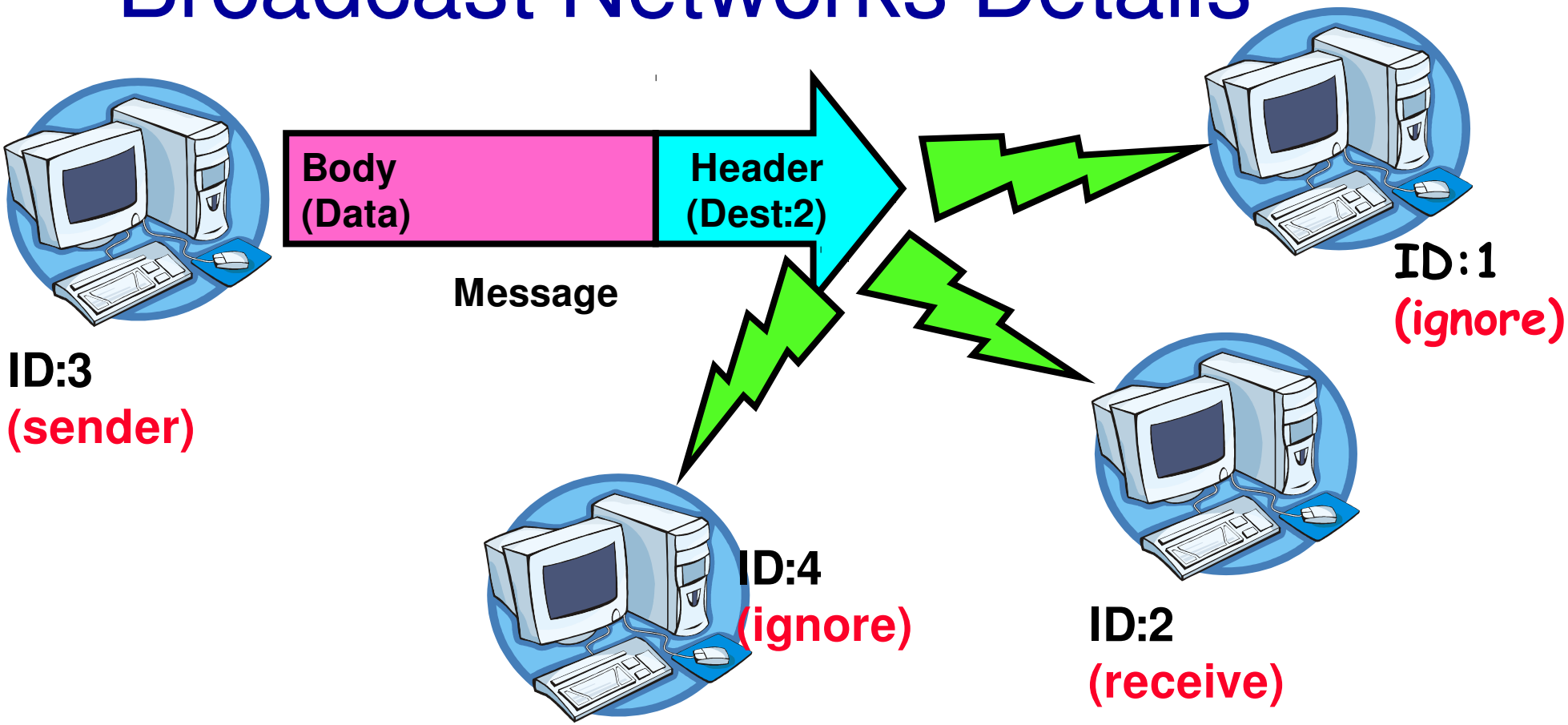
Broadcast Networks

Shared Communication Mechanism

Examples:

- Original Ethernet
- All types of wireless (WiFi, cellular, ...)
- Coaxial cable (e.g. cable internet)

Broadcast Networks Details



Building unicast (message to one) from broadcast (message to all):

- Put header on front of packet: [Destination | Packet]
- Discards if not the target (often in hardware)

Broadcast Network Arbitration

Arbitration: Who can use shared medium when?

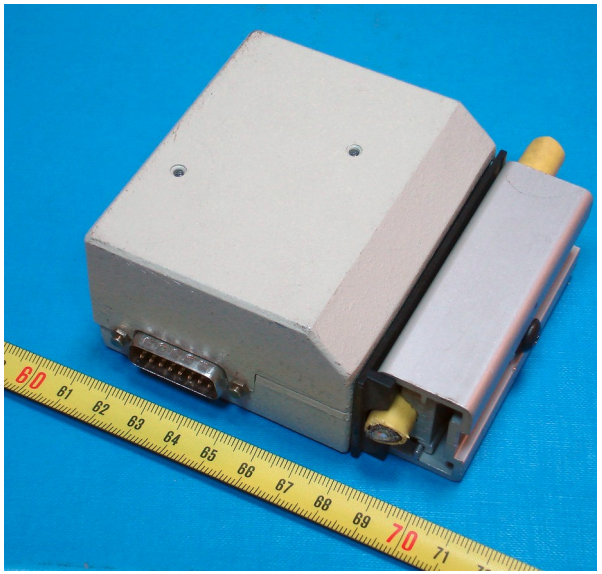
First example: Aloha network (70's): packet radio within Hawaii

- Blind broadcast, with **checksum** at end of frame.
- If two senders try to send at same time, both get garbled, both simply re-send later.

Problems: How many frames are lost?

- If network is too busy, no one gets through
- Need to not re-send in sync

Carrier Sense, Multiple Access/Collision Detection



Ethernet (early 80's):

- Originally – shared wire, clip on to it (literally)

Carrier Sense: don't send unless medium idle

- Less good for wireless (propagation delays, colliding station might be out of range)

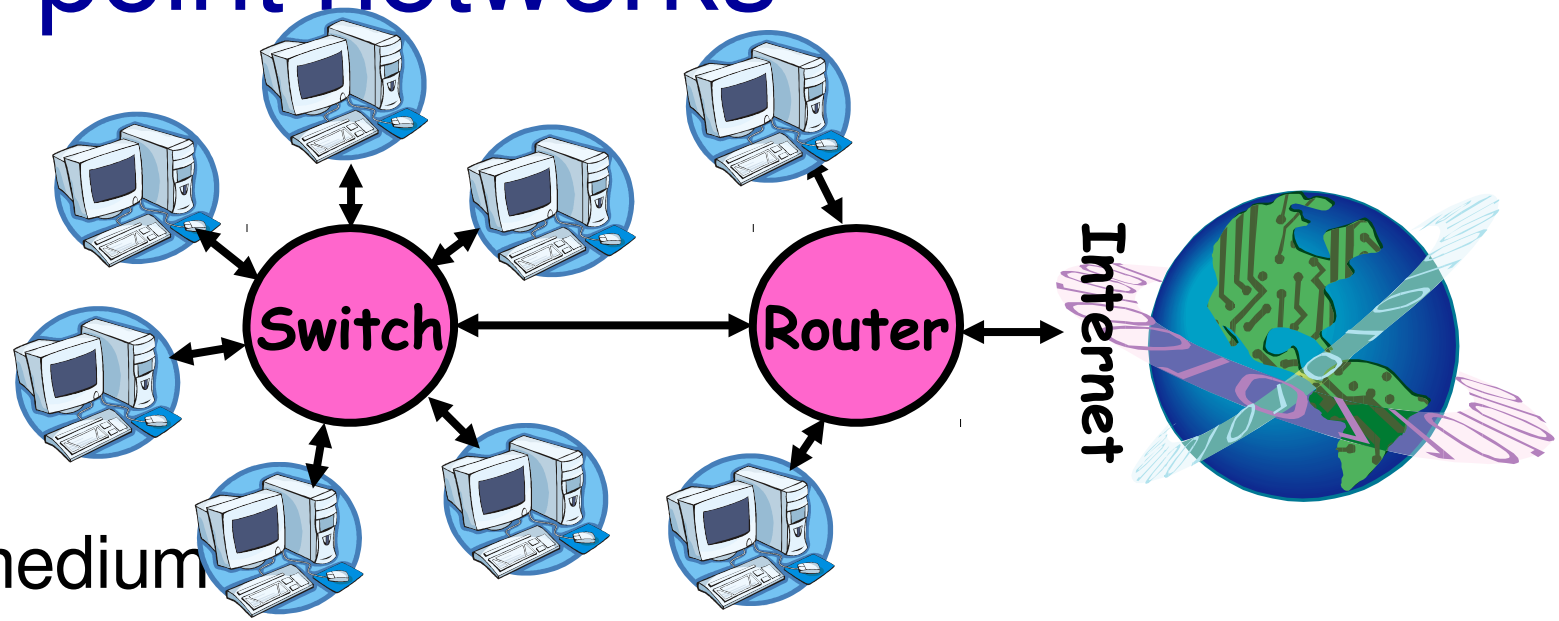
Collision Detect: sender checks if packet trampled.

- If so, abort, wait and retry

Adaptive randomized backoff

- Wait a random amount of time before retransmitting
- Avoids retransmitting at the same time as colliding machine
- **Increasing wait times** to adjust to how busy the medium is

Point-to-point networks



No shared medium

- Strictly simpler – no filtering

Switches: provide shared-bus view with point-to-point links

- Examines "destination" header, resends on appropriate link

Routers: connects two networks

- Distinction between switches/routers is actually fuzzy...

Layering

Complex services from simpler ones

TCP/IP networking layers:

- Physical + Link (Wireless, Ethernet, ...)
  - Unreliable, local exchange of limited-sized **frames**
- **Network (IP) – routing between networks**
  - Unreliable, global exchange of limited-sized **packets**
- Transport (TCP, UDP) – routing
  - Reliablity, streams of bytes instead of packets, …
- Application – everything on top of sockets

Glue: Adding Functionality

Physical Reality: Frames	Abstraction: Stream
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous

Glue: Adding Functionality

Physical Reality: Frames	Abstraction: Stream
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	**Routed anywhere**
Asynchronous	Synchronous

The Internet Protocol: “IP”

The Internet is a large network of computers spread across the globe

IP Packet: a network packet on the internet

IP Address: (in IPv4) a 32-bit integer used as the destination of an IP packet

- Often written as four dot-separated integers, with each integer from 0—255 (thus representing 8x4=32 bits)
- Example CS file server is: 169.229.60.83 = 0xA9E53C53

IPv6 extends to 128 bits (mostly the same otherwise)

Internet Host: a computer connected to the Internet

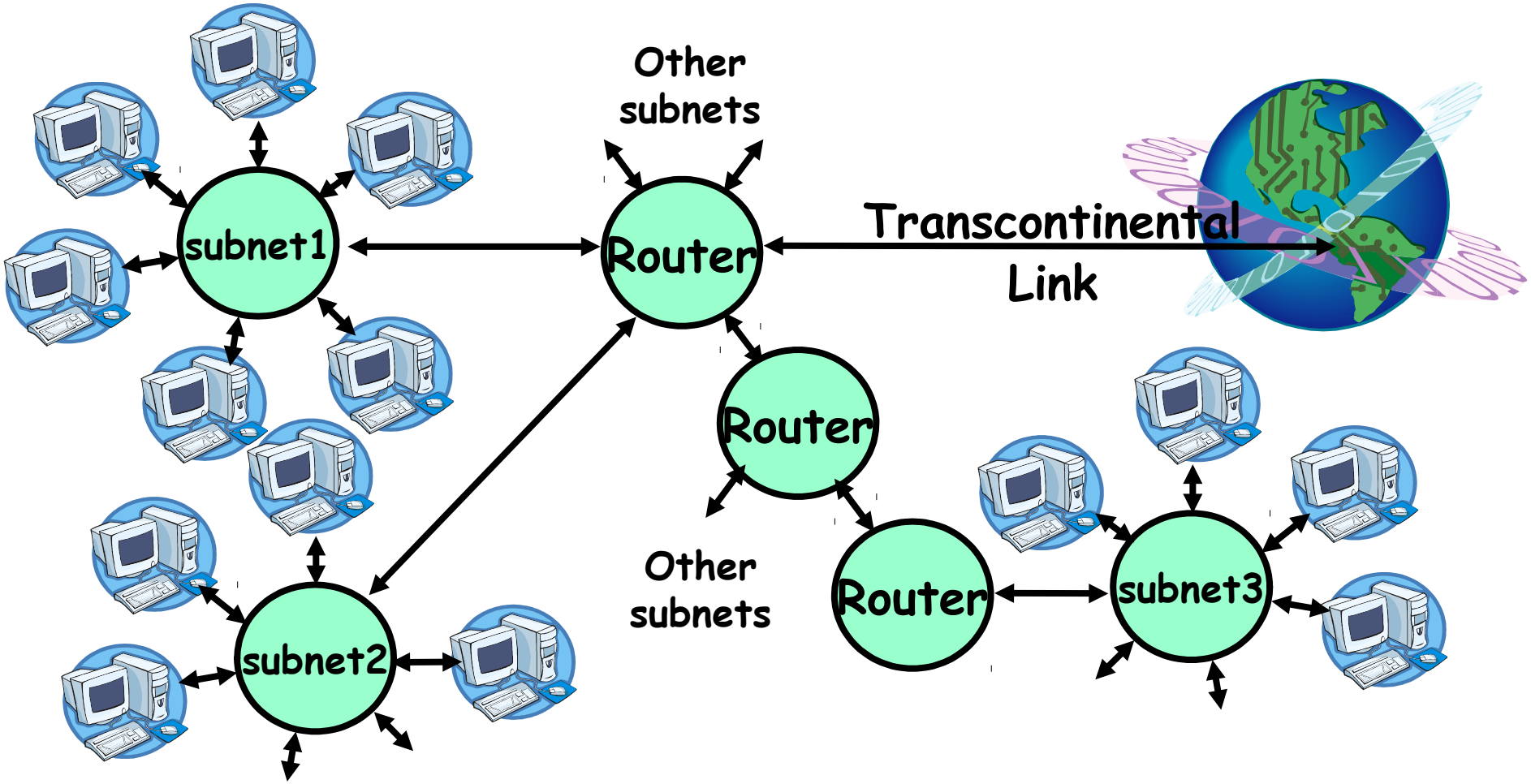
- One or more IP address – some private
- Network Address Translation – Not every computer has a unique IP

The Internet Protocol: “IP”

Internet Host: a computer connected to the Internet

- Host has one or more IP addresses used for routing
  - Some of these may be private and unavailable for routing
- Not every computer has a unique (global) IP address
  - Groups of machines may share a single IP address
  - In this case, machines have private addresses behind a “Network Address Translation” (NAT) gateway

Internet: Network of Networks



Address Subnets

With IP, all the addresses in subnet are related by a prefix of bits

- Mask: The number of matching prefix bits
  - Expressed as a single value (e.g., 24) or a set of ones in a 32-bit value (e.g., 255.255.255.0)

A IPv4 subnet is identified by 32-bit value, with the bits which differ set to zero, followed by a slash and a mask

- Example: 128.32.131.0/24 designates a subnet in which all the addresses look like 128.32.131.XX
- Same subnet: 128.32.131.0/255.255.255.0

Historically: subnet → single shared medium

Special Subnets in IPv4

127.0.0.0/8 – loopback – same machine

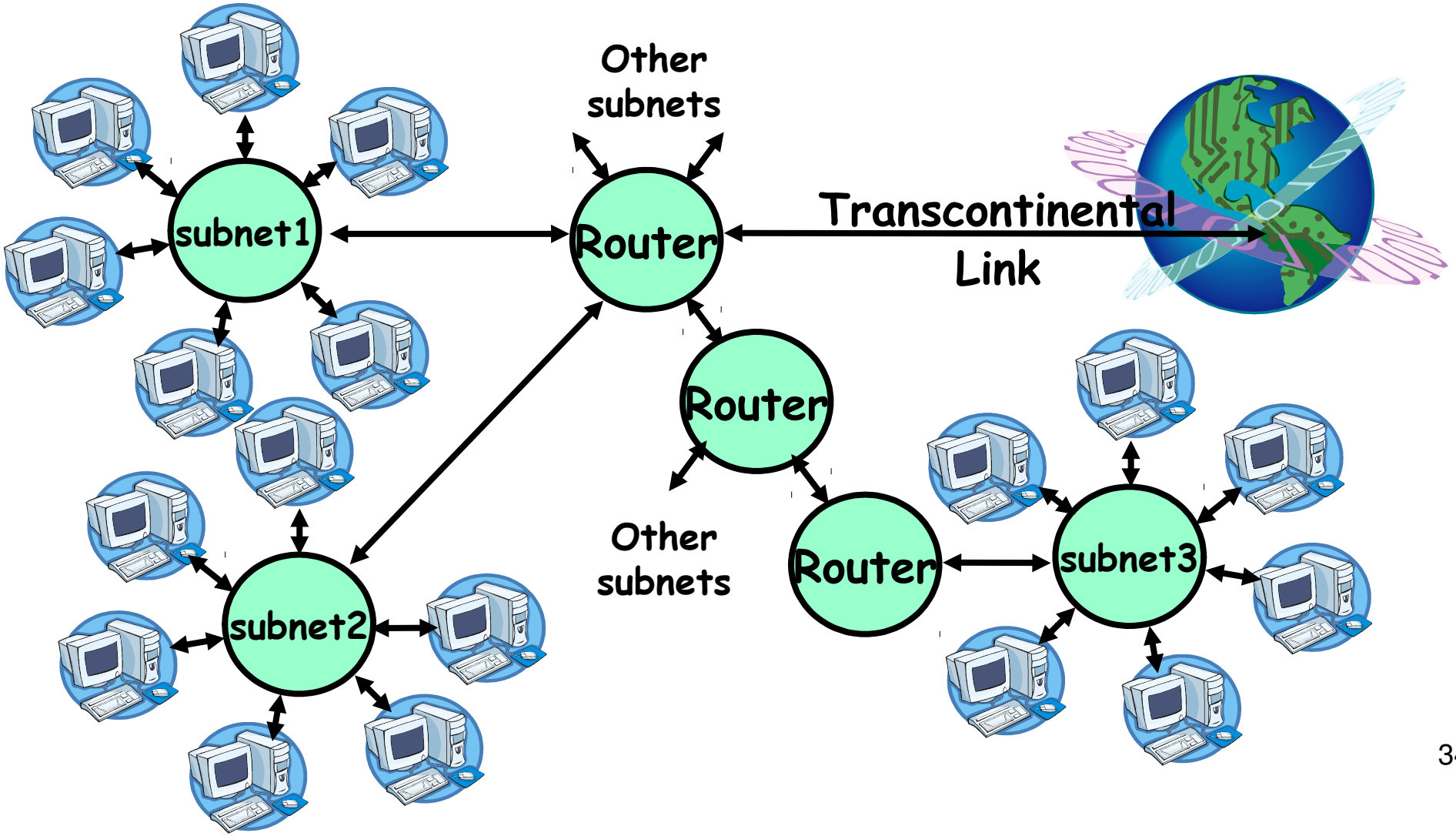
- localhost: 127.0.0.01

10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 – private

- not available globally

Hierarchical Networking: The Internet

Hierarchy of networks – scales to millions of host



Simple Network Terminology

Local-Area Network (LAN) – designed to cover small geographical area

- Multi-access bus, ring, or star network
- Speed: 10 – 1000 Megabits/second (even 40-100GB/s)
- Broadcast fast + cheap

Wide-Area Network (WAN) – links geographically separated sites

- Point-to-point connections over long-haul lines (often leased from a phone company)
- Speed: 1.544 – 150 Megabits/second
- Broadcast usually requires multiple messages

Routing (1)

Routing: the process of forwarding packets hop-by-hop through routers to reach their destination

Need more than just a destination address!

- Need a path

Post Office Analogy:

- Destination address on each letter is not sufficient to get it to the destination
- To get a letter from here to Florida, must route to local post office, sorted and sent on plane to somewhere in Florida, be routed to post office, sorted and sent with carrier who knows where street and house is…

Routing (2)

Internet routing mechanism: routing tables

- Each router does table lookup to decide which link to use to get packet closer to destination
- Don't need 4 billion entries (or 2^128 for IPv6) in table: routing is by subnet

Routing table contains:

- Destination address range: output link closer to destination
- Default entry

Setting up Routing Tables (1)

How do you set up routing tables?

Internet has no centralized state!

- No single machine knows entire topology
- Topology constantly changing (faults, reconfiguration, etc)

Dynamic algorithm that acquires routing tables

- Routers talking to each other
- Default route for "edges"

Setting up Routing Tables (2)

Possible algorithm for acquiring routing table

- Routing table has “cost” for each entry
  - Includes number of hops to destination, congestion, etc.
- Neighbors periodically exchange routing tables
  - If neighbor knows cheaper route to a subnet, replace your entry with neighbors entry (+1 for hop to neighbor)

In reality:

- Internet has networks of many different scales
- Different algorithms run at different scales

Names versus Addresses

Names are

- meaningful
- memorable
- don't change if the resource moves

Addresses

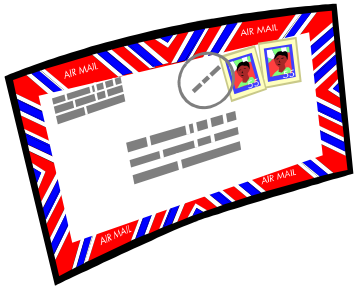
- explain how to access a resource
- change if the resource moves

Example:

- int foo; ← variable in C
- 'foo' is the name, address is a pointer to some place on the stack...

Naming in the Internet

Name ⇒ Address



You probably want to use human-readable names:

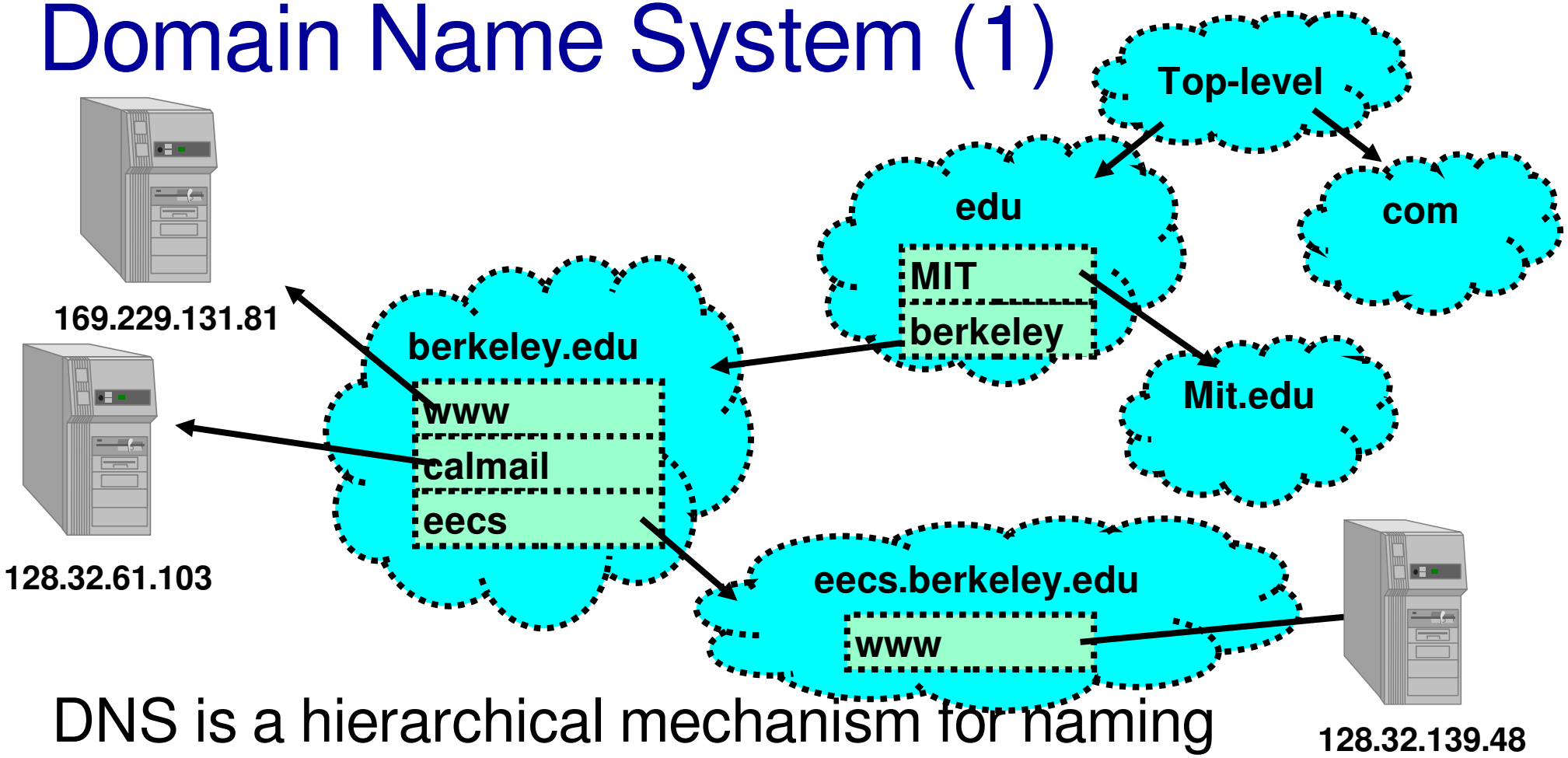
- www.google.com
- www.berkeley.edu

Network wants an IP address:

- that's what's in routing tables
- allows routing tables to take advantage of hierarchy

Mapping is done by the ***Domain Name System***

Domain Name System (1)



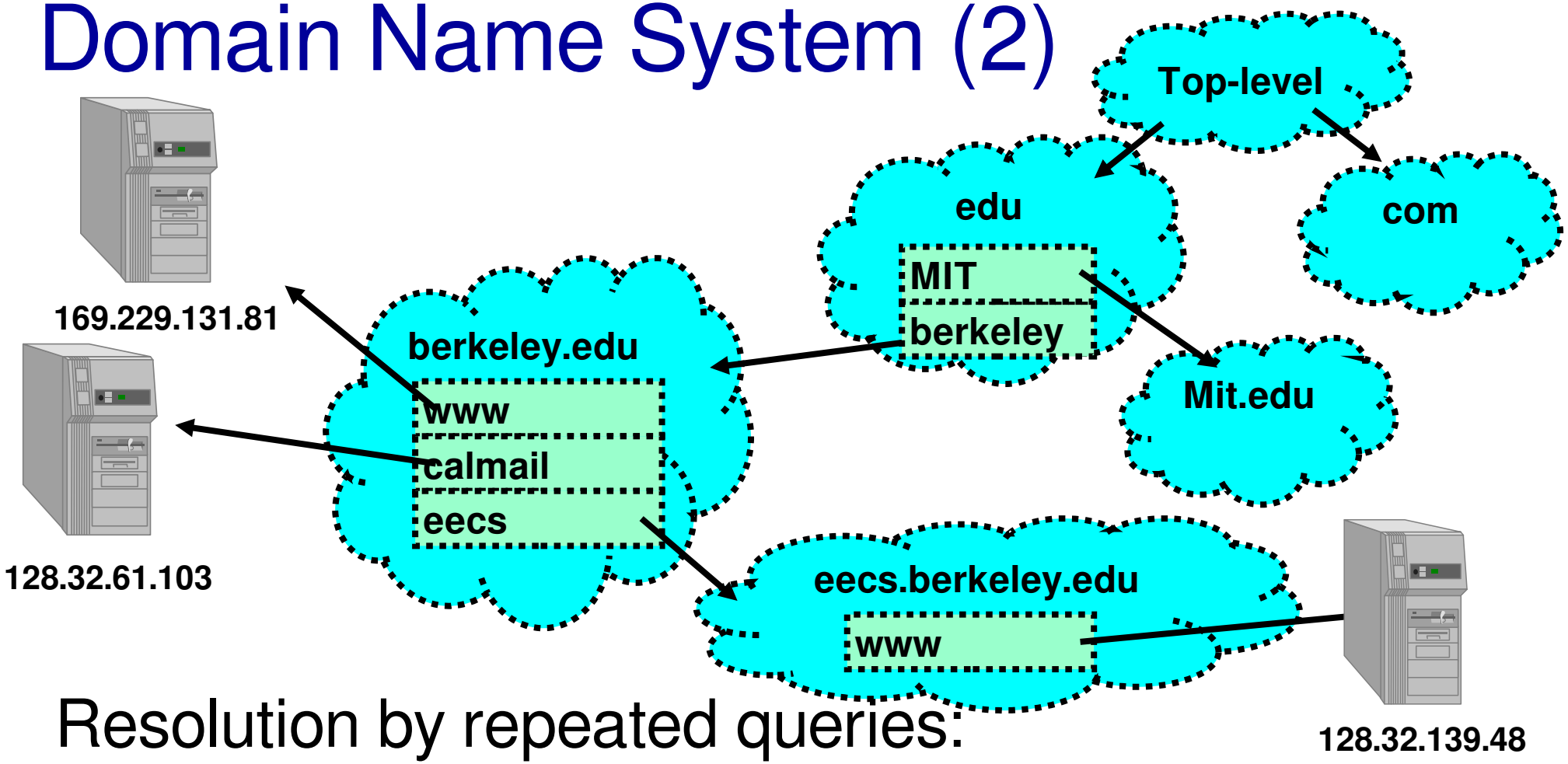
DNS is a hierarchical mechanism for naming

- Name divided into "labels", right to left: www.eecs.berkeley.edu

Each domain owned by a particular organization

- Top level handled by ICANN
- Subsequent levels owned by organizations

Domain Name System (2)



Resolution by repeated queries:

- One server for each "domain" (<root>, edu, berkeley.edu)
- Plus backups: redundancy (available, not gaurenteed consistent)

Domain Name System (3)

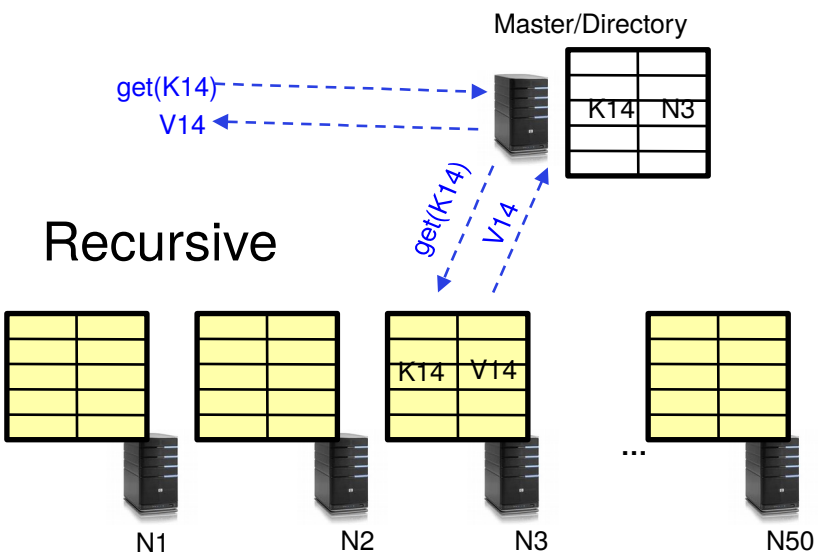
How do you find the root server?

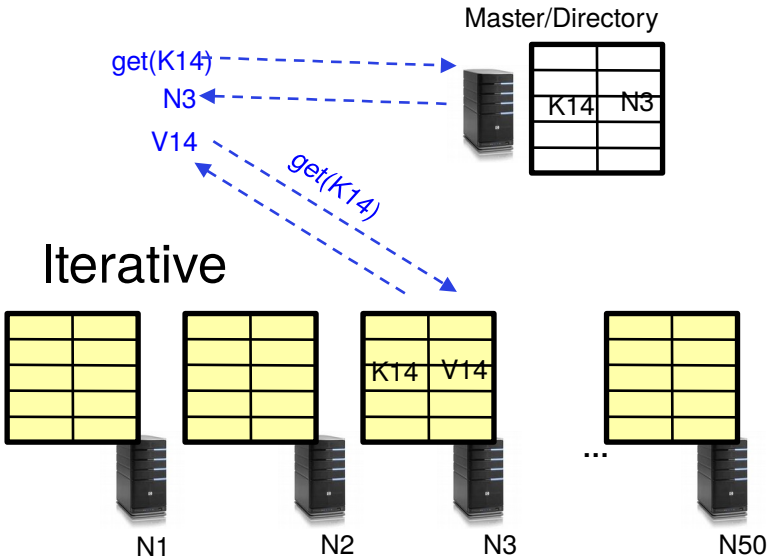
Hardcoded list of root servers and backups (updated rarely)

... or use your ISP's server (makes repeated queries on your behalf)

- *called* ***a recursive resolver***

Recall: Iterative vs. Recursive Query

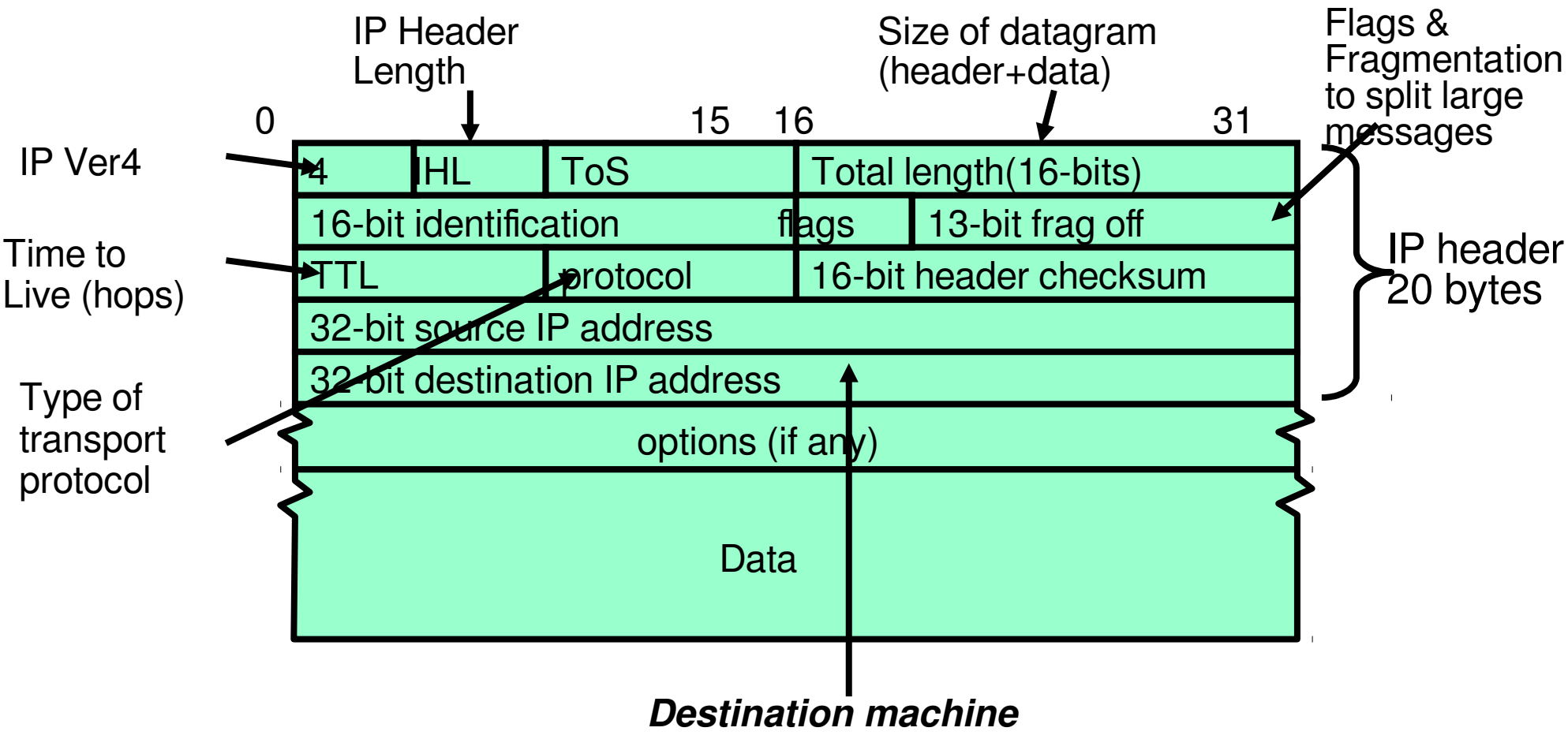




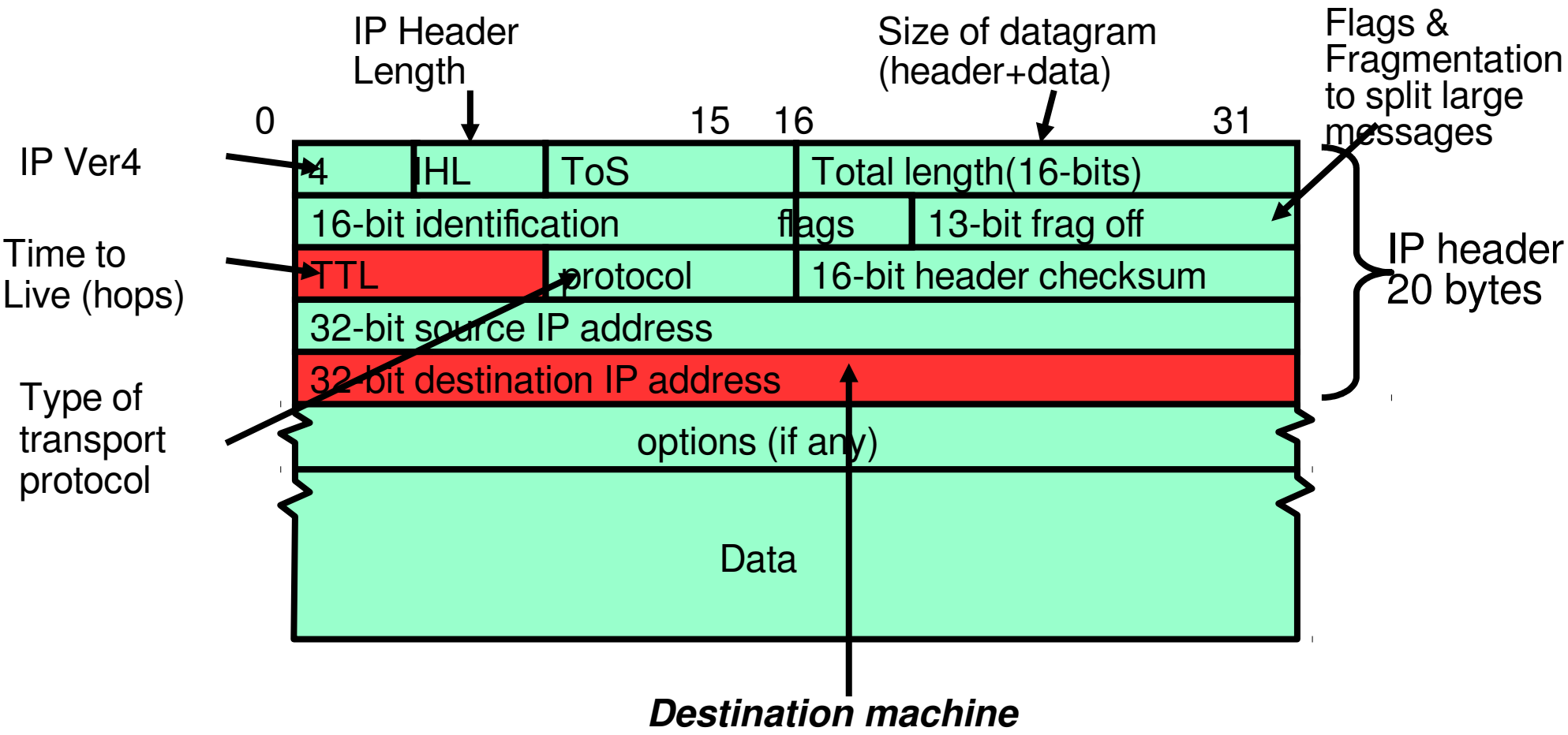
Recursive Query: Directory delegates

Iterative Query: Client delegates

IPv4 Packet Format



IPv4 Packet Format: What the router cares about



Internet Protocol Features

Routing – IP packet goes anywhere

- Just need the IP address

Fragmentation – split big messages into smaller

- Still message size limit (64K)
- Reassemble at destination
- Hides differences in physical layers

Multiple protocols on top:

- 1 byte to specify protocol:
  - ICMP (1), ***TCP*** (6), UDP (17), IPSec (50, 51), …
- Registry of protocol numbers

Internet Protocol Non-Features

Unreliable delivery

- IP packets are not guaranteed
- May be lost by underlying physical layer (radio noise?)
- May be dropped if, e.g., router out of resources

Out-of-order/duplicate delivery

- Tolerance to physical layer retrying packets
- Tolerance to multiple paths

Layering

Complex services from simpler ones

TCP/IP networking layers:

- Physical + Link (Wireless, Ethernet, ...)
  - Unreliable, local exchange of limited-sized **frames**
- Network (IP) – routing between networks
  - Unreliable, global exchange of limited-sized **packets**
- **Transport (TCP, UDP) – streams**
  - **Reliablity, streams of bytes instead of packets, ...**
- Application – everything on top of sockets

Glue: Adding Functionality

Physical Reality: Frames	Abstraction: Stream
Limited Size	**Arbitrary Size**
Unordered (sometimes)	**Ordered**
Unreliable	**Reliable**
Machine-to-machine	**Process-to-process**
Only on local area net	Routed anywhere
Asynchronous	Synchronous

Ordered Messages: Problem

Want to divide message into packets, e.g.

- "GET /static/hw/hw1.pdf HTTP/1.0" into
- "GET /static" and "/hw/hw1.pdf HTTP/1.0"

Why? Not can't always fit request in one packet (IP: 64K limit – think of uploading a file)

IP might reorder these packets:

- "/hw/hw1.pdf HTTP/1.0", then
- "GET /static"

Ordered Message: Solution

Ordered messages on top of unordered ones:

- Assign sequence numbers to packets: 0,1,2,3,4....
- If packets arrive out of order, hold and put back in order before delivering to user (through socket interface)
- For instance, hold onto #3 until #2 arrives, etc.

Tricky case: packets from "old" connections

Reliable Message Delivery: Problem

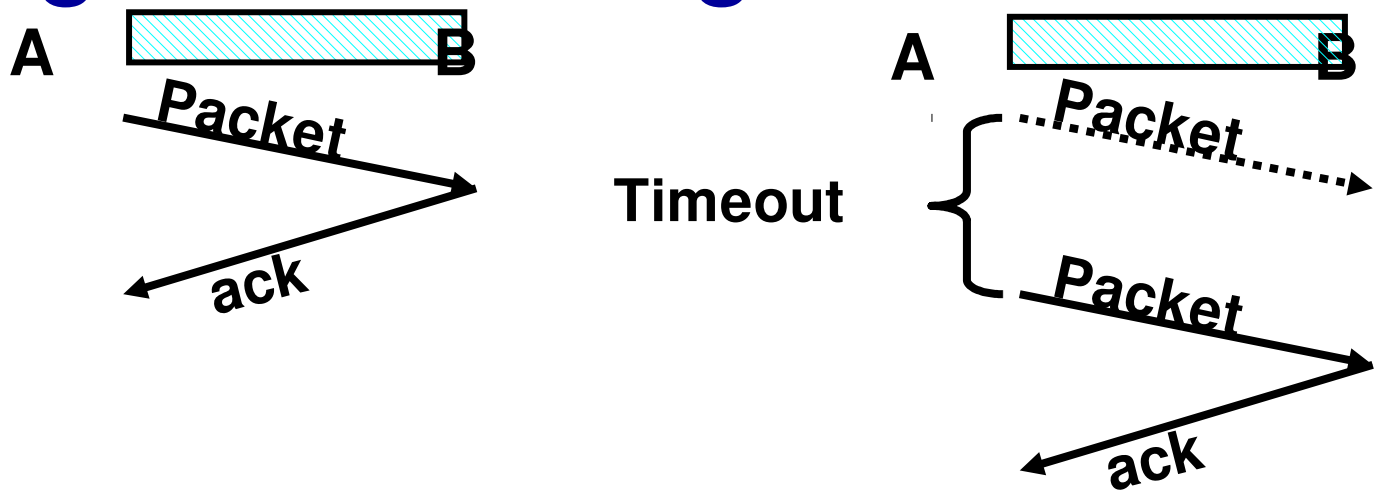
All physical networks can garble and/or drop packets

- Physical hardware problems (bad wire, bad signal)
- Therefore, IP can garble/drop packets (doesn't fix this)

Building reliable message delivery

- Confirm that packets arrive **exactly once**
- Confirm that packets aren't garbled

Using Acknowledgements



Checksum: detect garbled packets

Receiver send packet to acknowledge when packet received and ungarbled:

- No acknowledgement? **Resend** after timeout

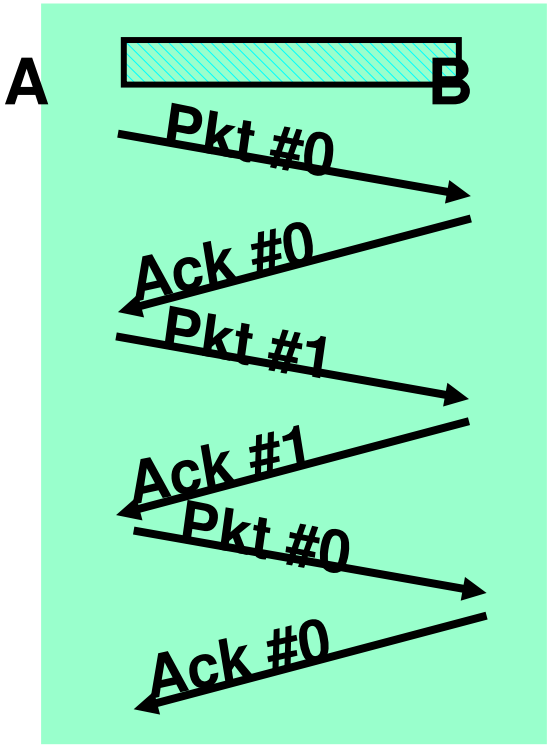
What if ack dropped?

- Packet is resent (wasteful), second chance to acknowledge it.

What about duplicates?

Recall: Sequence number

Simplest version: 1 bit (0/1):



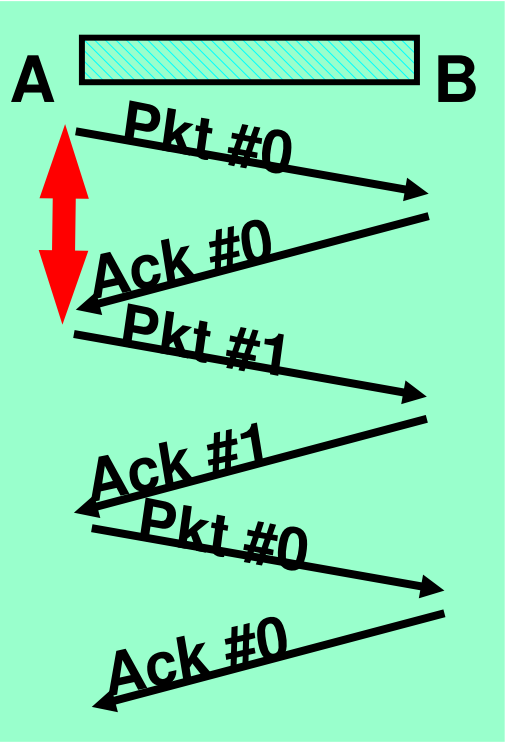
Problem: What if packet delayed too much?

Waiting for Acks: Performance

Time from Soda to google.com and back: **8 ms**

Maximum packet size: ~1500 bytes

1500 bytes / 8 ms = ***188 KByte/s***



Window-based acknowledgements (1)

Windowing protocol (not quite TCP)

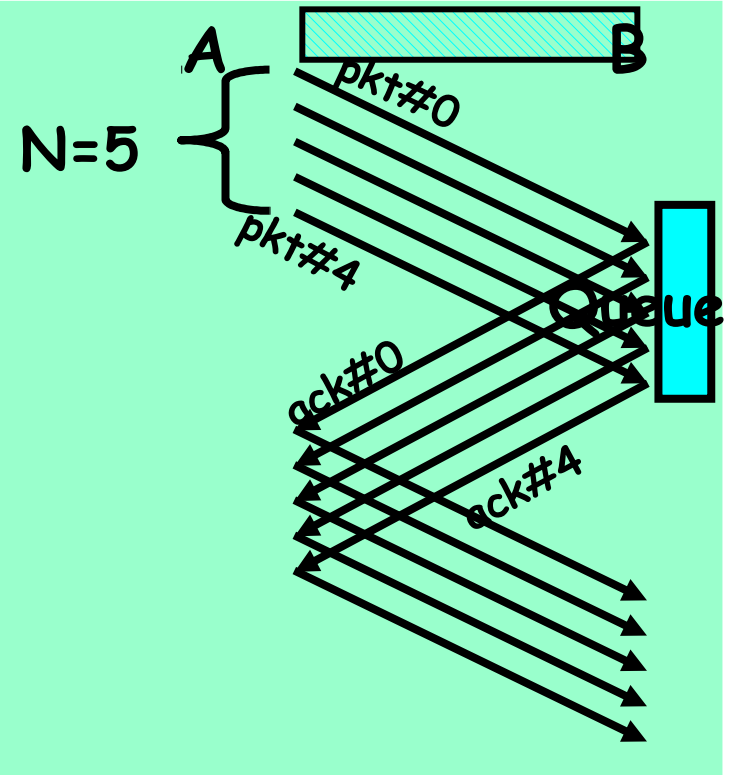
Send up to N packets without ack

- Allows pipelining of packets
- N limits **queue size**

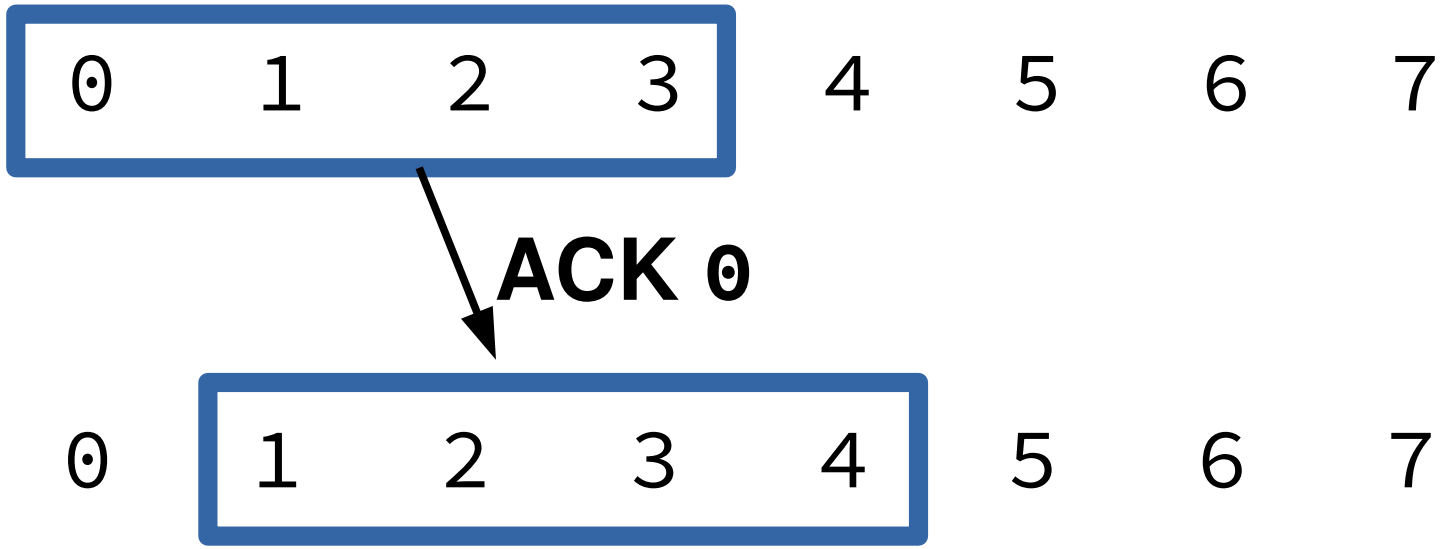
Both source and destination need to store N packets (why?)

Each packet has sequence number

- Ack says "**received all packets up to sequence number X**"
- Advances window



Sliding Window



Window represents packets:

- That might need to be retransmitted (dropped/garbled)
- That receiver needs space to buffer (ordering)

Window-based acknowledgements (2)

Packet lost?

- Resent by timeout, again

Acknowledgement lost?

- Packet resent, causing ack to be resent (again)
- If we're lucky acknowledged together with later packet instead

Discard out of order packets?

- If no, need some way to indicate "holes"

Summary: Network Layering

Link layer (local network):

- **Broadcast** or **Point-to-Point**
- Send *frames* addressed to neighboring machines
- Ethernet, Wi-Fi

Network layer (connecting network):

- ***Forwarding*** between local networks
- Send *packets* addressed to machines anywhere
- IP

Transport layer (Making streams):

- Turn sequence of packets into **reliable stream**
- TCP

Summary: Names and Addresses

DNS name: human readable

- Mapped to IP *address*
- Distributed database

IP address:

- 32-bit (IPv4) or 128-bit (IPv6) number
- Looked up in routing tables to decide where to send packets

Port number:

- 16-bit number
- Used to identify service on particular machine

Summary: Sockets

Abstraction of network I/O interface

- **Bidirectional** communication channel
- Uses **file** interface once established
  - read, write, close

Server setup:

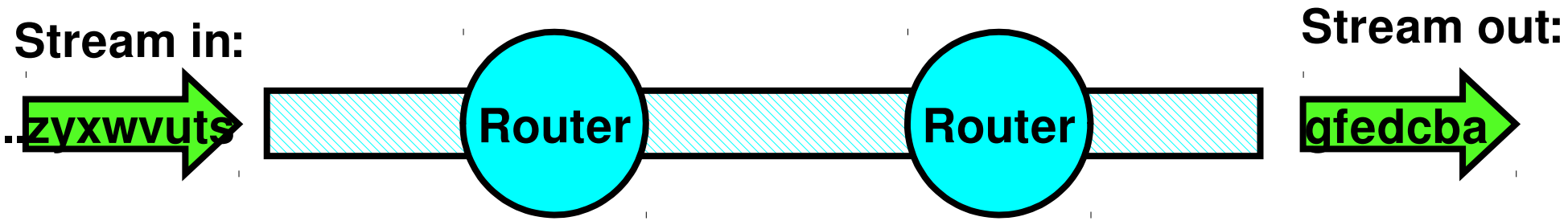
- socket(), bind(), listen(), accept()
- read, write, close from socket returned by accept

Client setup:

- socket(), connect()
- then read, write, close

getaddrinfo() to resolve names, addresses for bind() and connect()

Transmission Control Protocol (TCP)



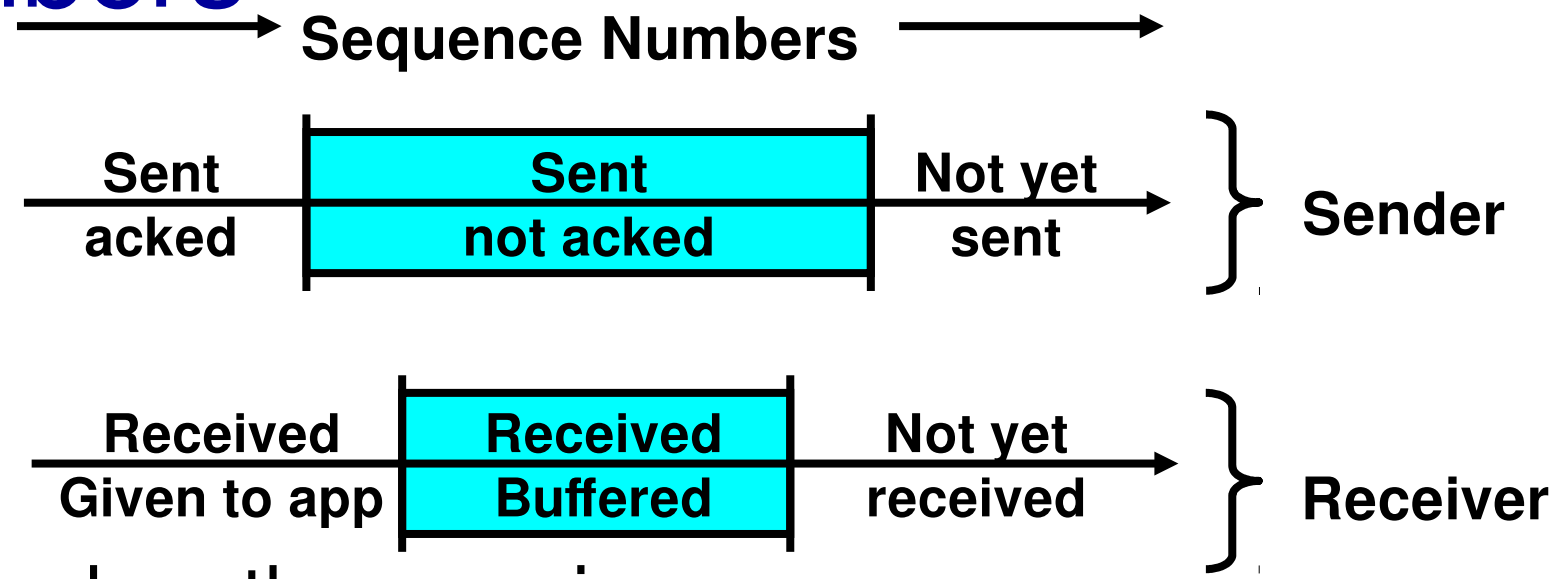
Transmission Control Protocol (TCP)

- **Reliable byte stream** between two processes on different machines over Internet (read, write)
- Bi-directional (two streams for every connection)

TCP Details

- Fragments byte stream into packets, hands packets to IP
- Window-based acknowledgement protocol
- Automatically retransmits lost packets
- Adjusts rate of transmission to avoid *congestion*
  - Mechanism: ***Window size***

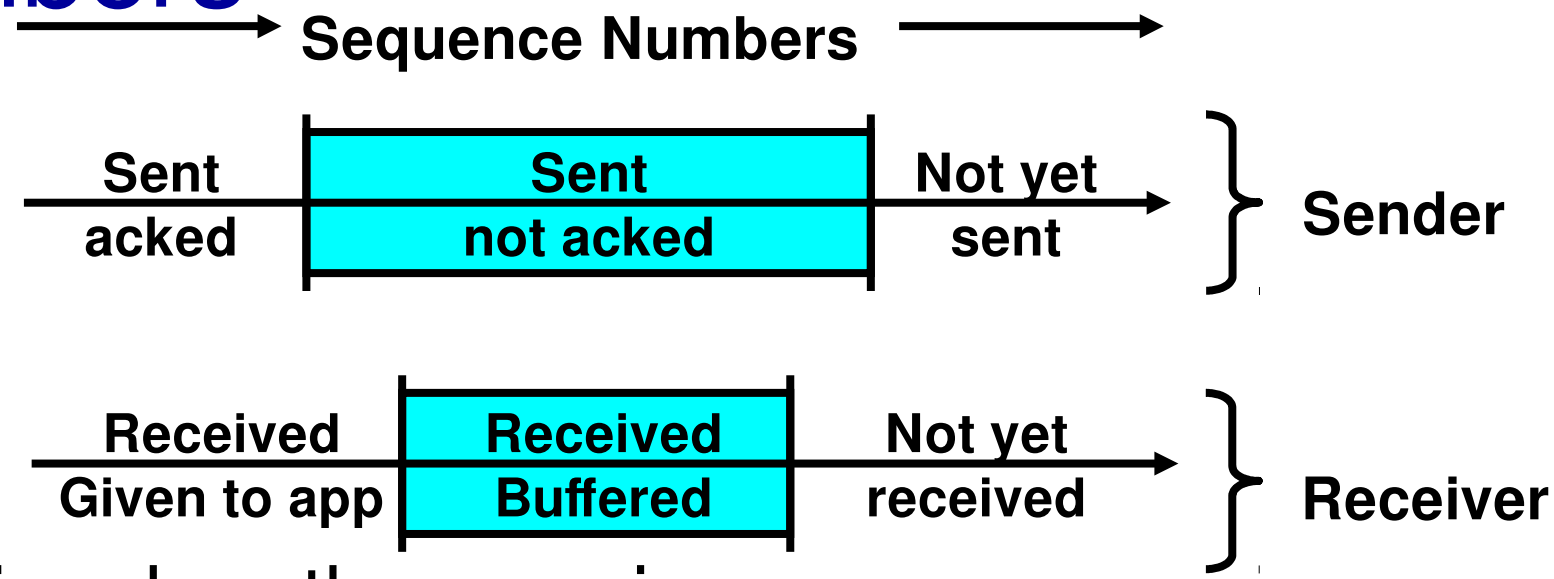
TCP Windows and Sequence Numbers



Sender has three regions:

- sent and ack'ed
- sent and not ack'ed
- not yet sent

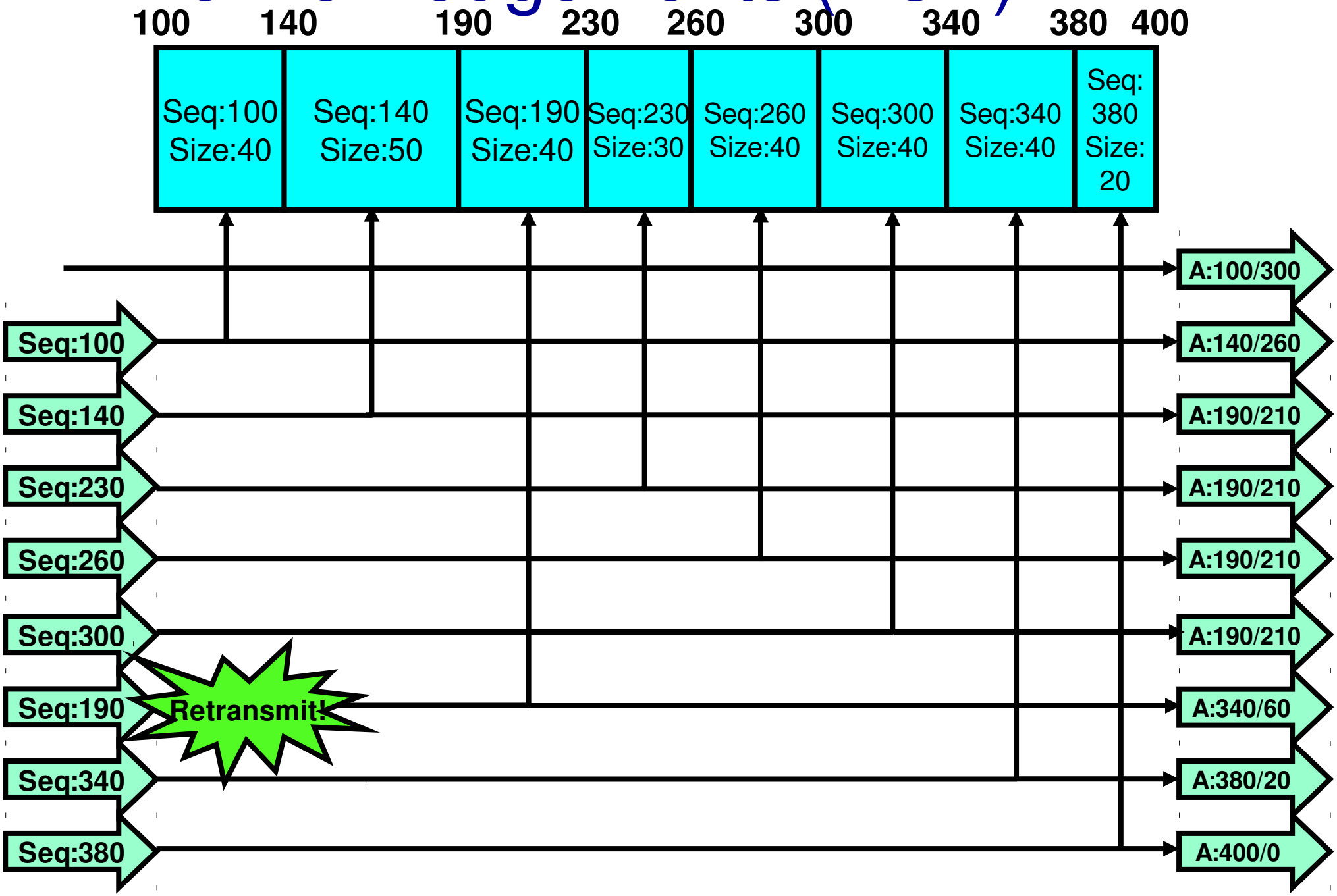
TCP Windows and Sequence Numbers



Receiver has three regions:

- received and ack'ed (given to application)
- received and buffered
- not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)



Selective Acknowledgement Option (SACK)

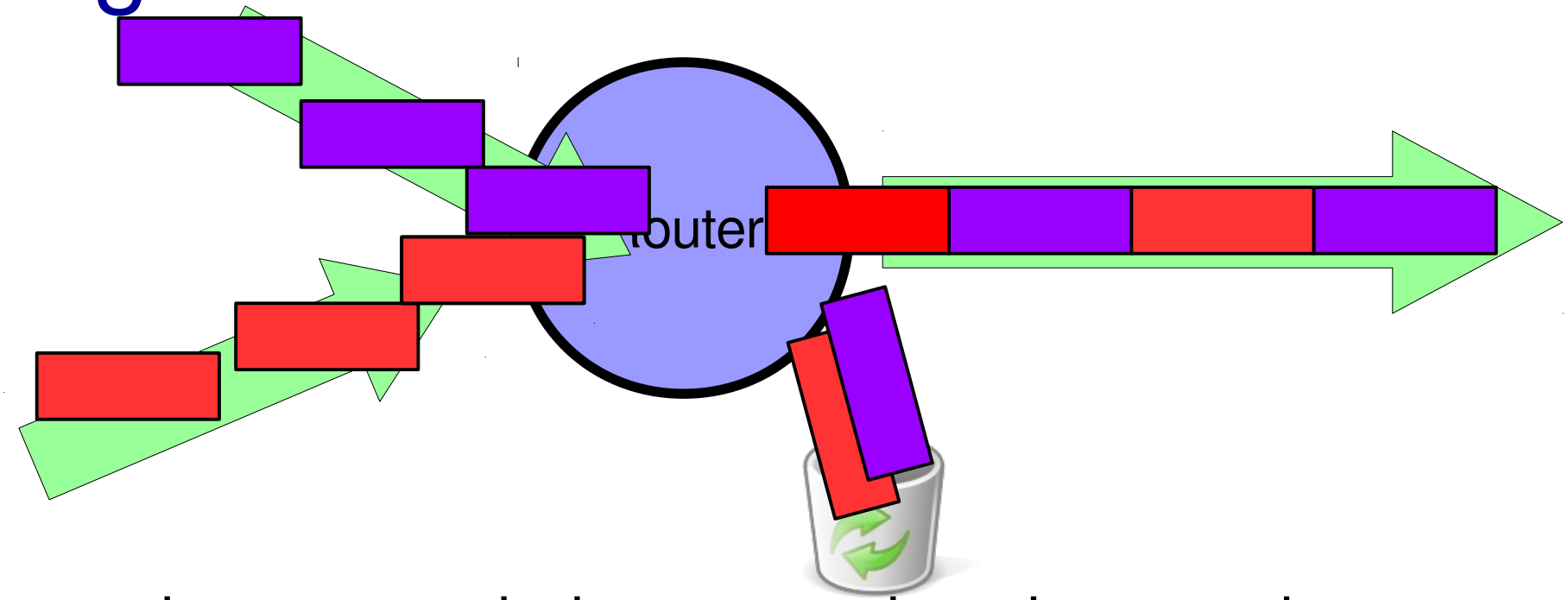
Vanilla TCP Acknowledgement

- Every message encodes Sequence number and Ack
- Can include data for forward stream and/or ack for reverse stream

Selective Acknowledgement

- Acknowledgement information includes not just one number, but rather ranges of received packets
- TCP **Option** (extension) – often not used
- Turns out extending TCP is hard (compatibility problems)

Congestion



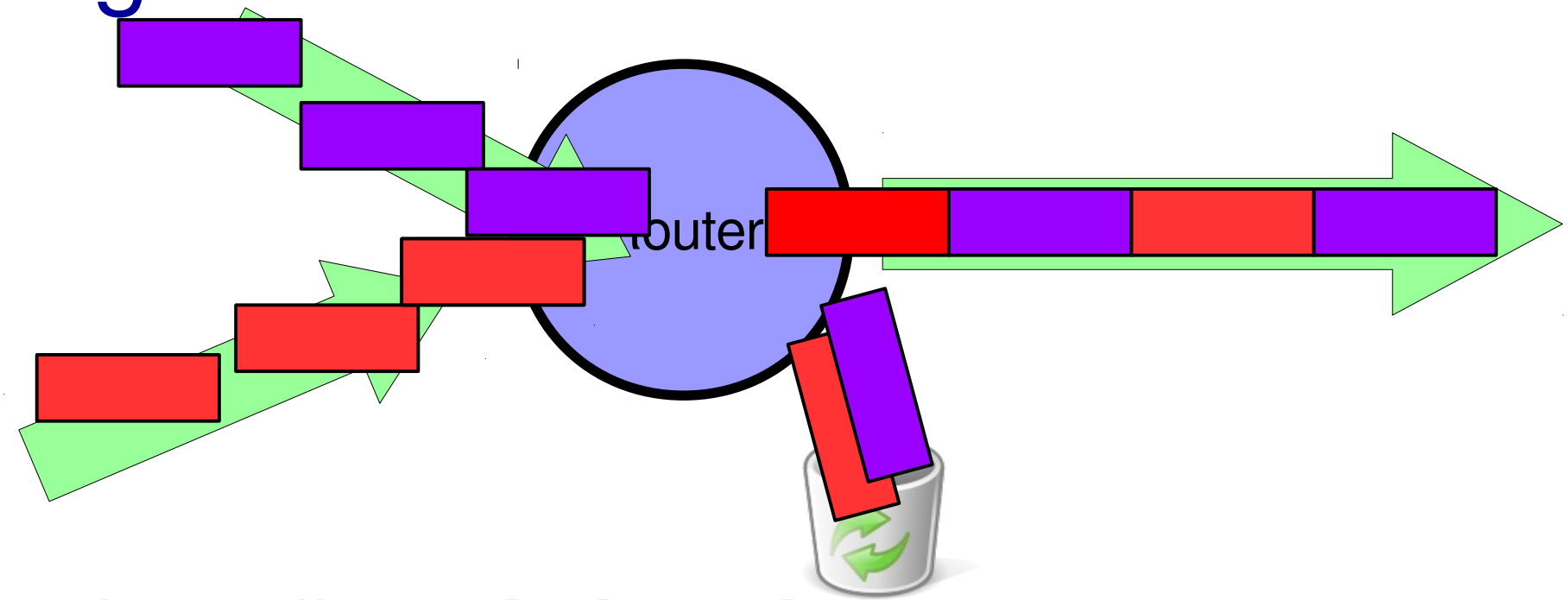
Congestion: too much data somewhere in network

IP's solution: *drop* packets

Bad for TCP

- Lots of retransmission – wasted work
- Lots of waiting for timeouts – underutilized connection

Congestion Avoidance

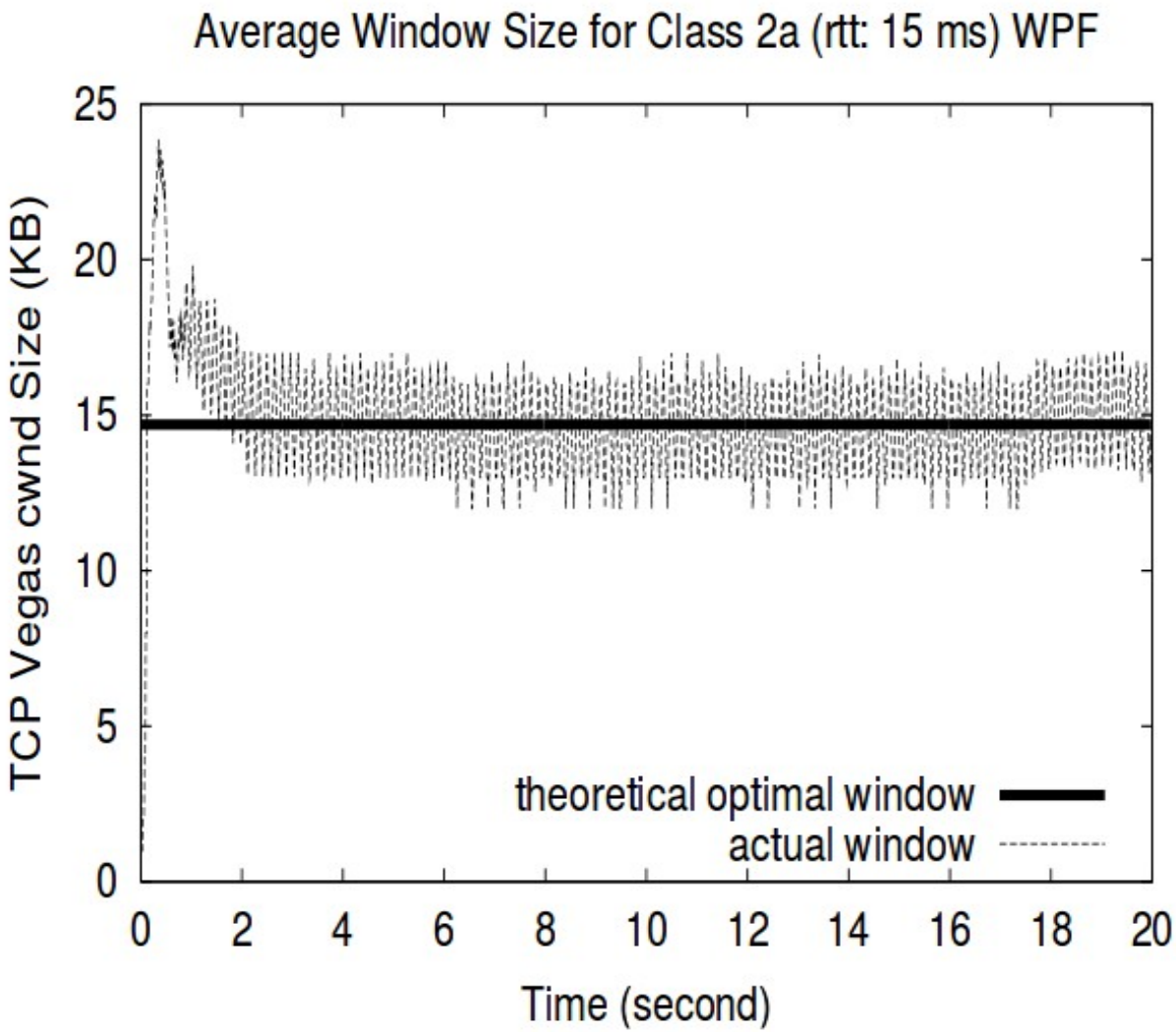


Solution: adjust **window size**

AIMD – Additive Increase, Multiplicative Decrease

- When packet dropped (missed ack) → decrease rapidly
- When packet successfully ack'd, increase slowly

Congestion Avoidance: Changing Window



From Low, Peterson, and Wang, "Understanding vegas: Duality Model", J. ACM, March 2002.

Implementing Congestion Avoidance

Need to track window size

Need to keep buffer of unacknowledged packets

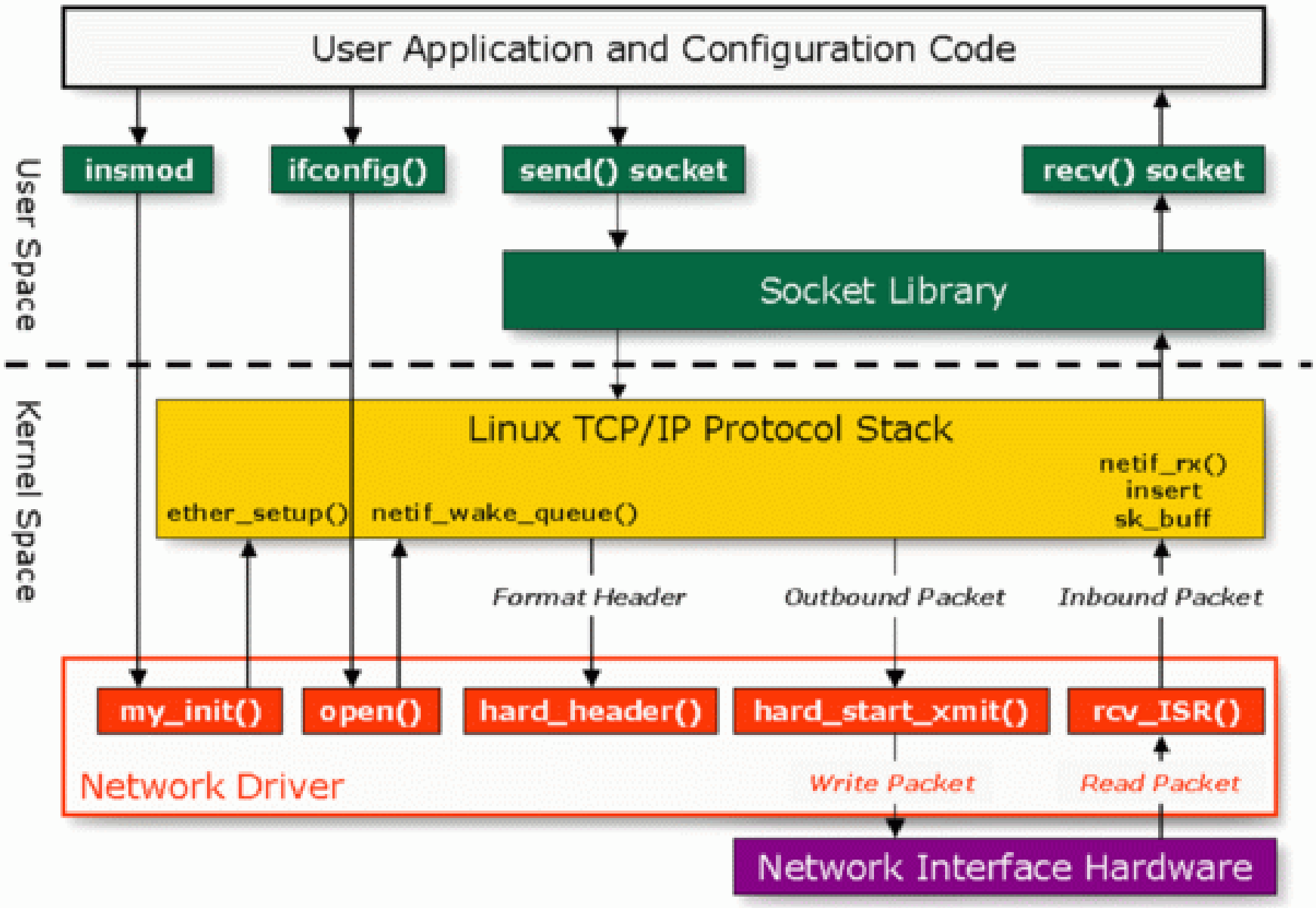
Need to keep buffer of received but unread packets

Need to resend packets when timer expires (one for each connection?)

Need to acknowledge packets and adjust window size

Who does this? On Linux – the kernel

Linux Network Architecture



Implementing Congestion Avoidance

Need to track window size

Need to keep buffer of unacknowledged packets

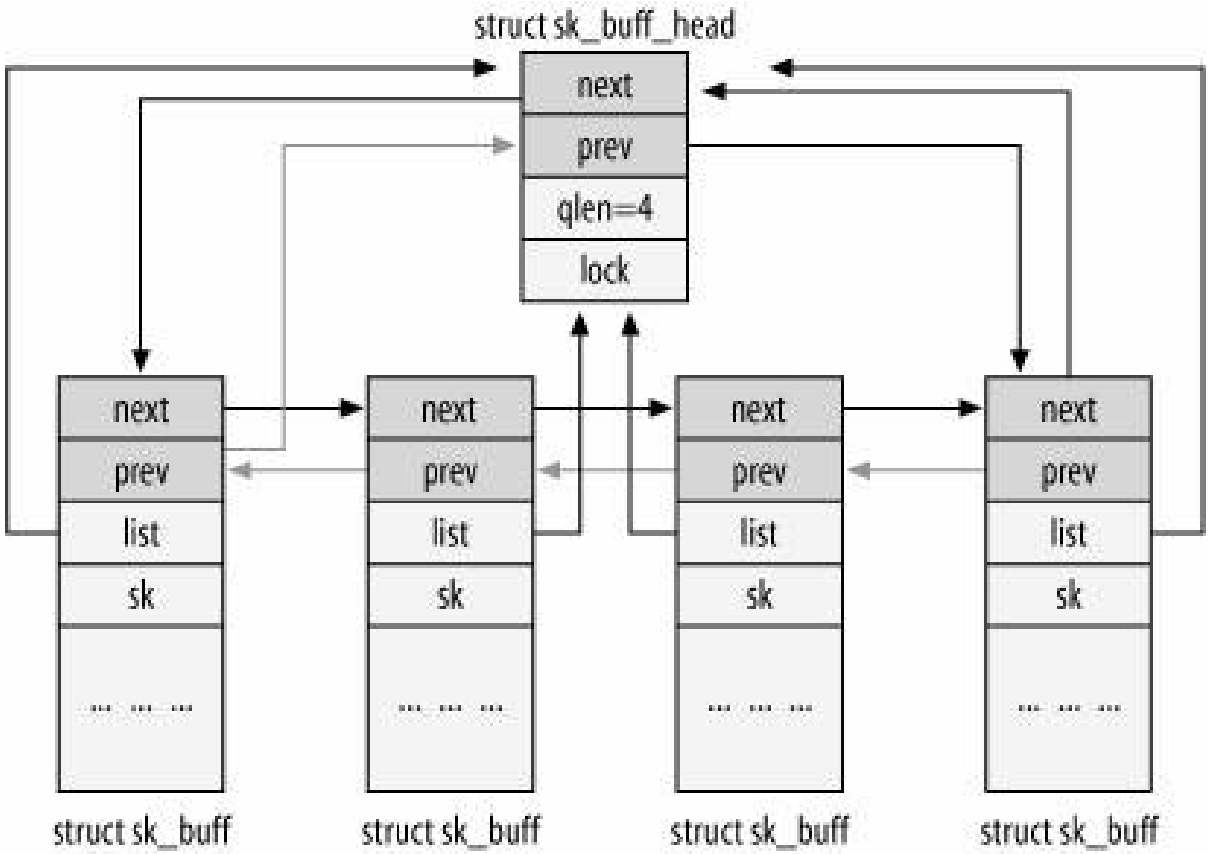
Need to keep buffer of received but unread packets

Need to resend packets when timer expires (one for each connection?)

Need to acknowledge packets and adjust window size

Who does this? On Linux – the kernel

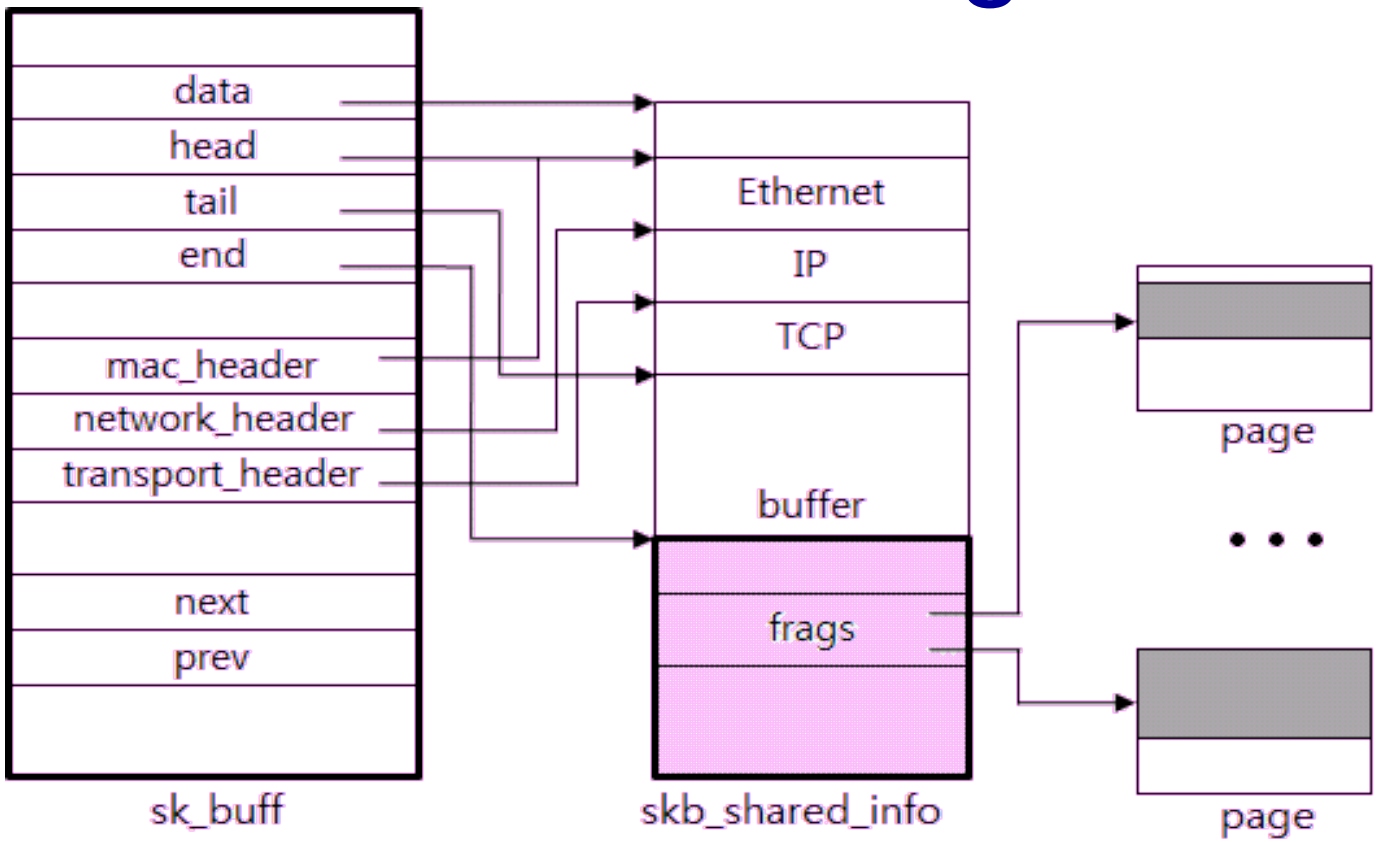
Linux Details: sk_buff structure



Socket **Buffers**: sk_buff structure

- The I/O buffers of sockets are lists of sk_buff
- Packet is linked list of sk_buff structures

sk_buff: Headers and Fragments

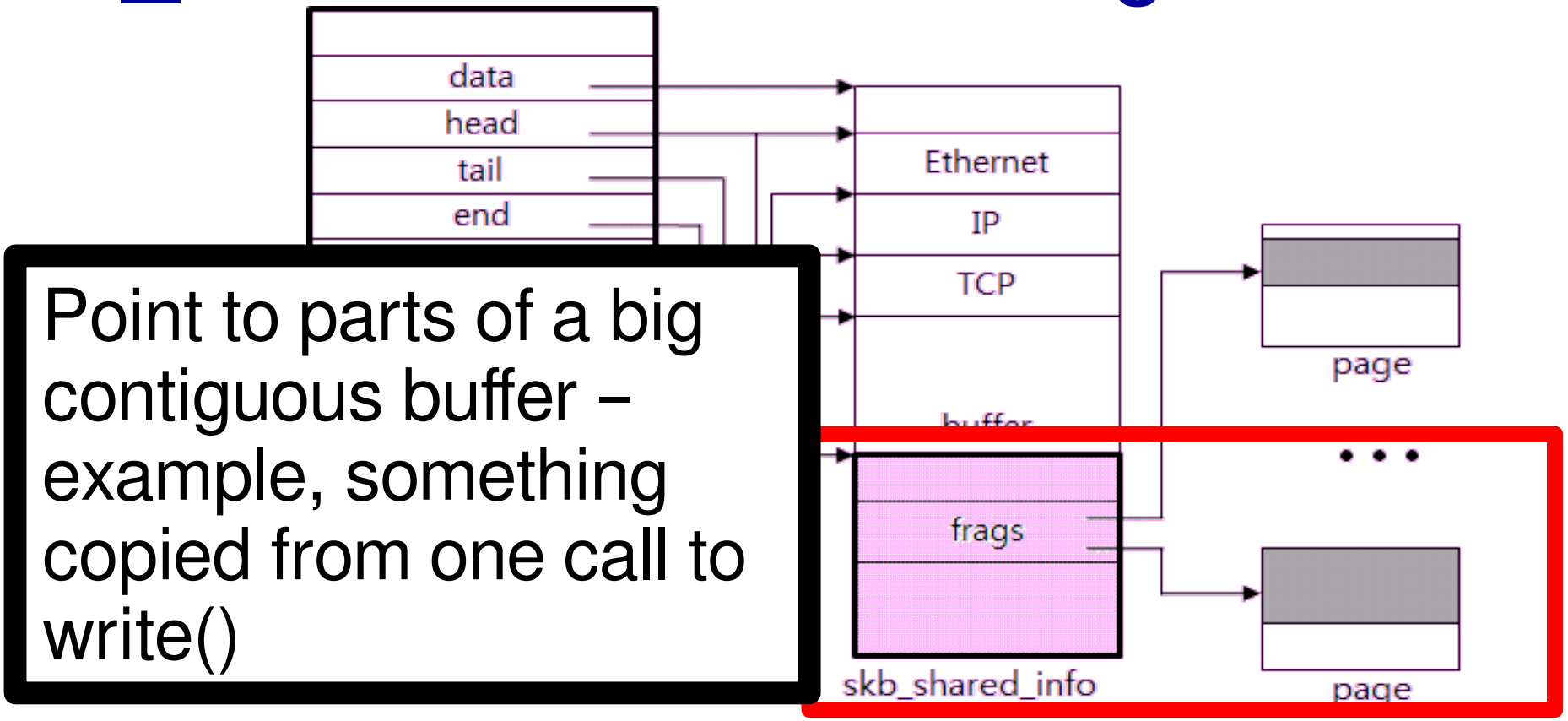


The “linear region”: Space from skb->data to skb->end

- Convenience: header pointers point to parts of packet

And the fragments (in skb_shared_info) – pointers to seperate pages

sk_buff: Headers and Fragments



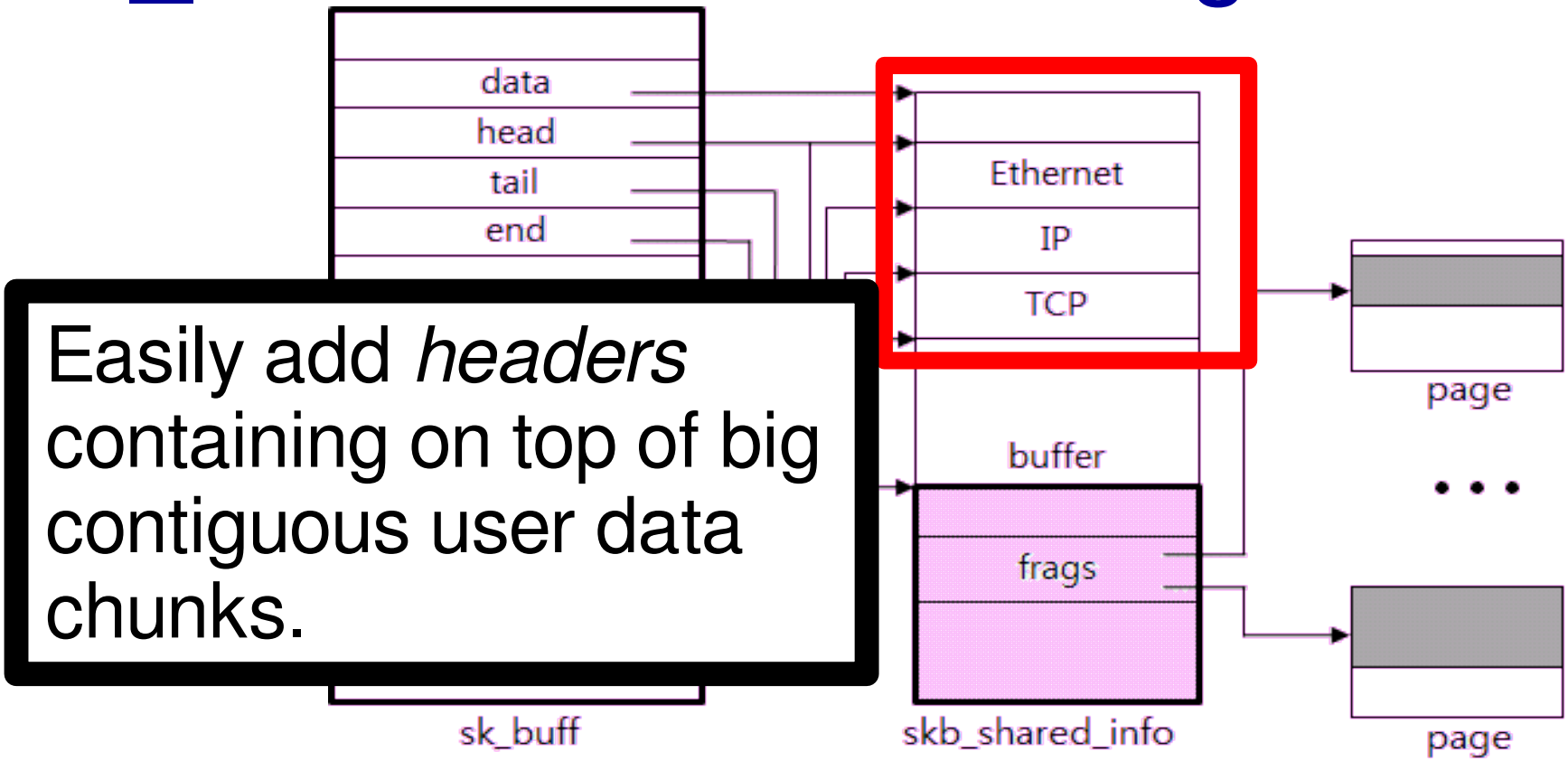
Point to parts of a big contiguous buffer – example, something copied from one call to write()

The "linear region": Space from skb->data to skb->end

- **Convenience: header pointers point to parts of packet**

And the fragments (in skb_shared_info) – pointers to seperate pages

sk_buff: Headers and Fragments



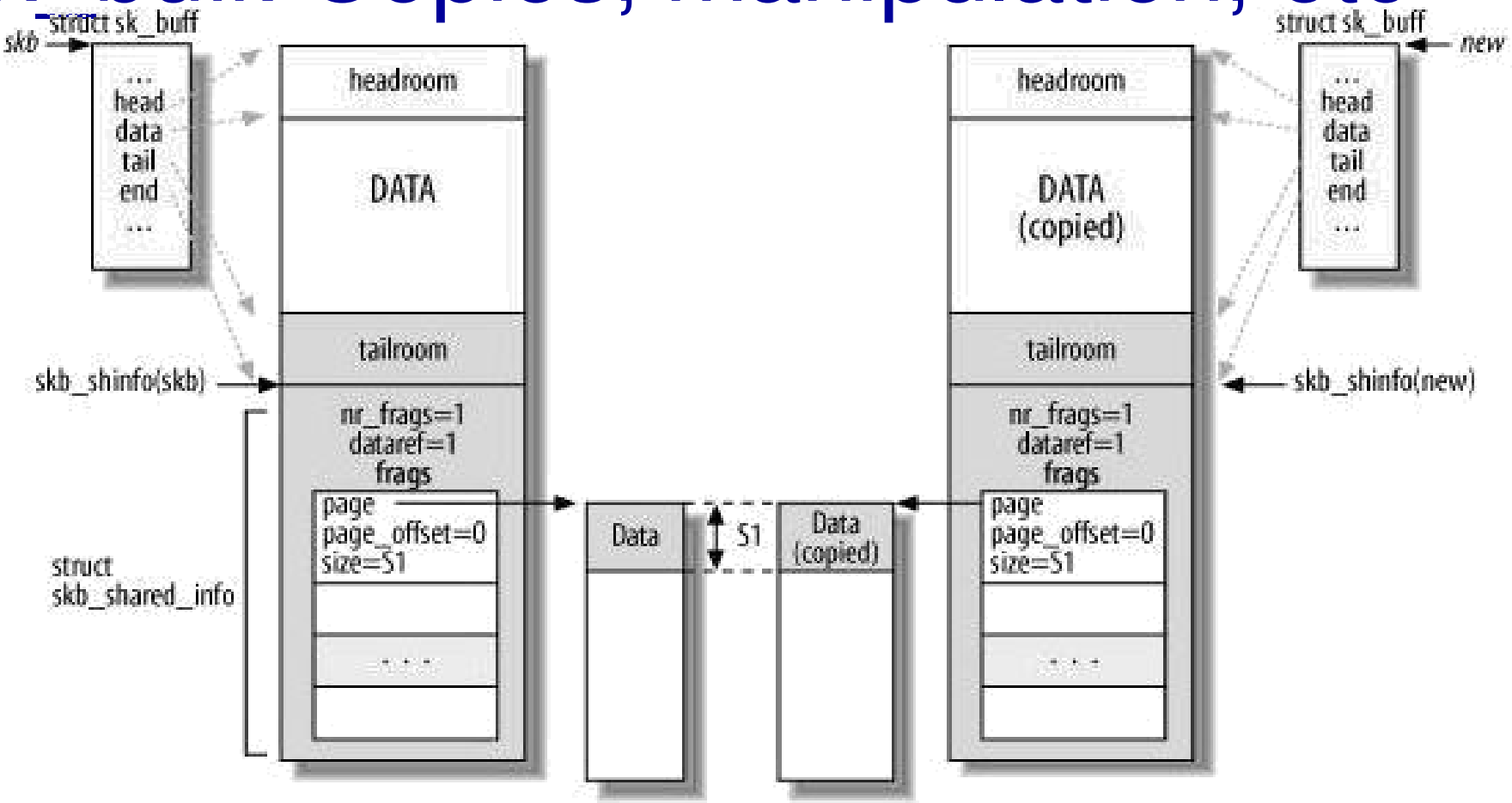
Easily add *headers* containing on top of big contiguous user data chunks.

The "linear region": Space from skb->data to skb->end

- **Convenience: header pointers point to parts of packet**

And the fragments (in skb_shared_info) – pointers to seperate pages

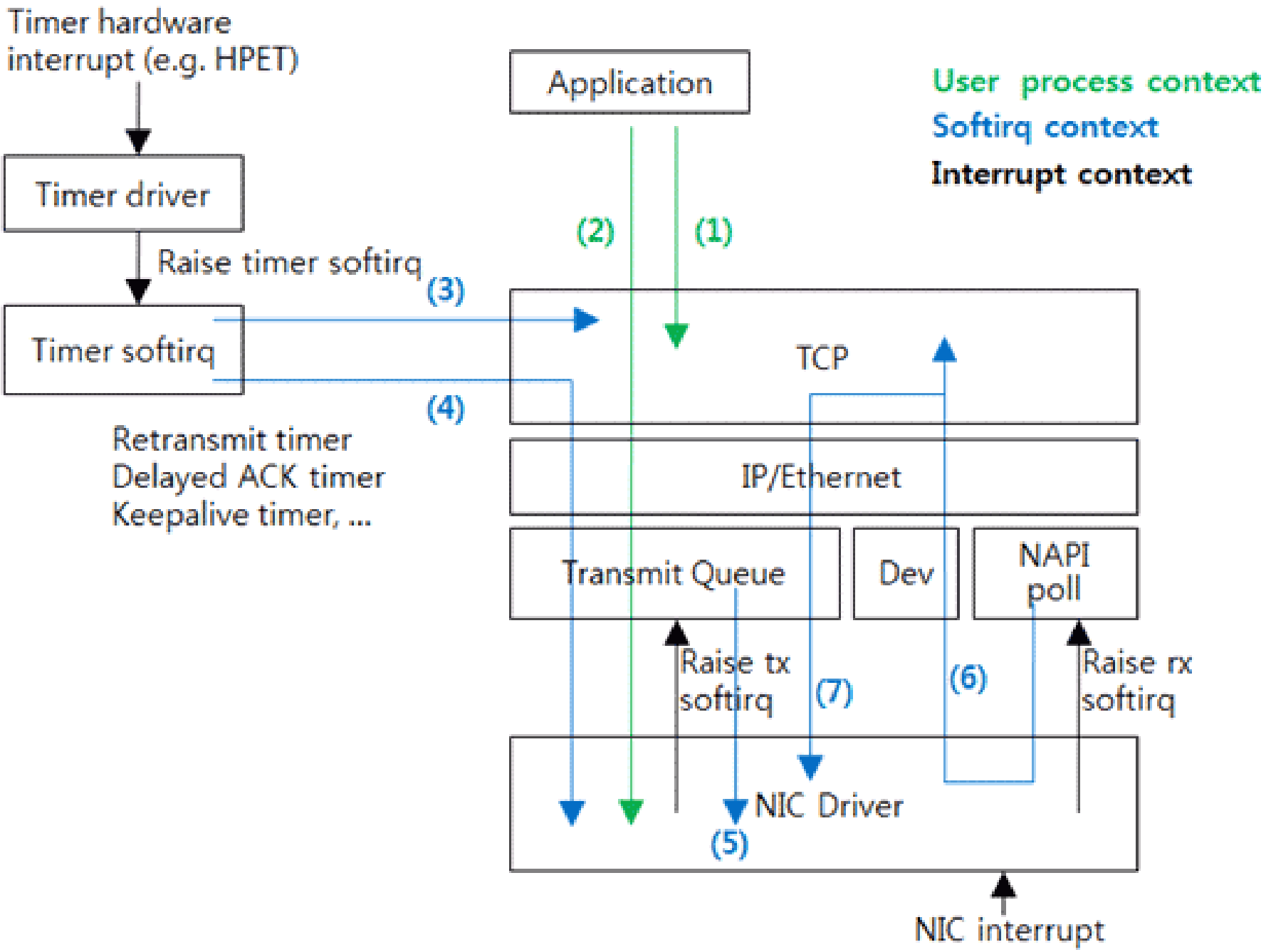
sk_buff: Copies, manipulation, etc



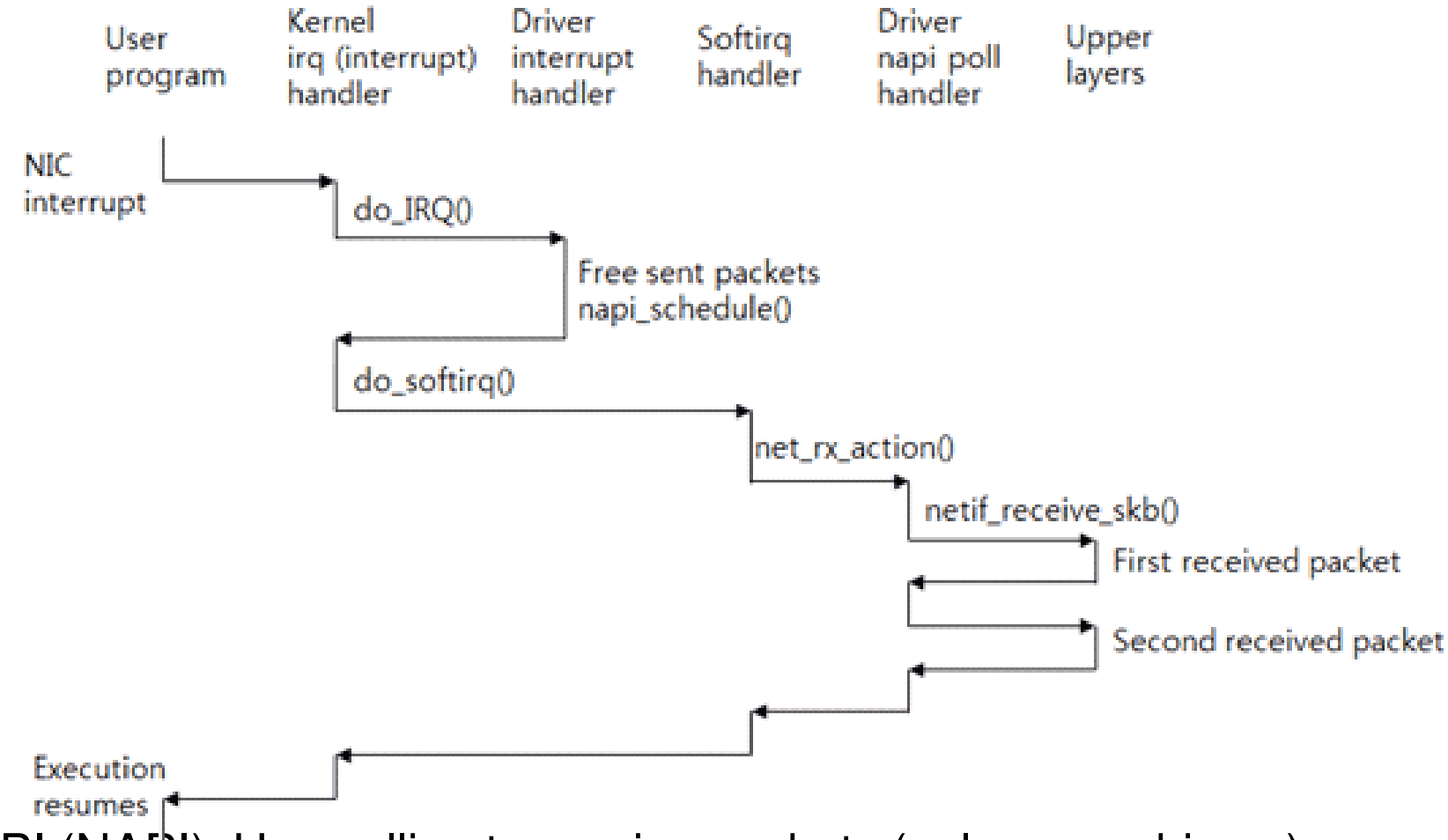
Lots of sk_buff manipulation functions for:

- removing and adding headers, merging data, pulling it up into linear region
- Copying/cloning sk_buff structures

Network Processing Contexts



Avoiding Interrupts: NAPI



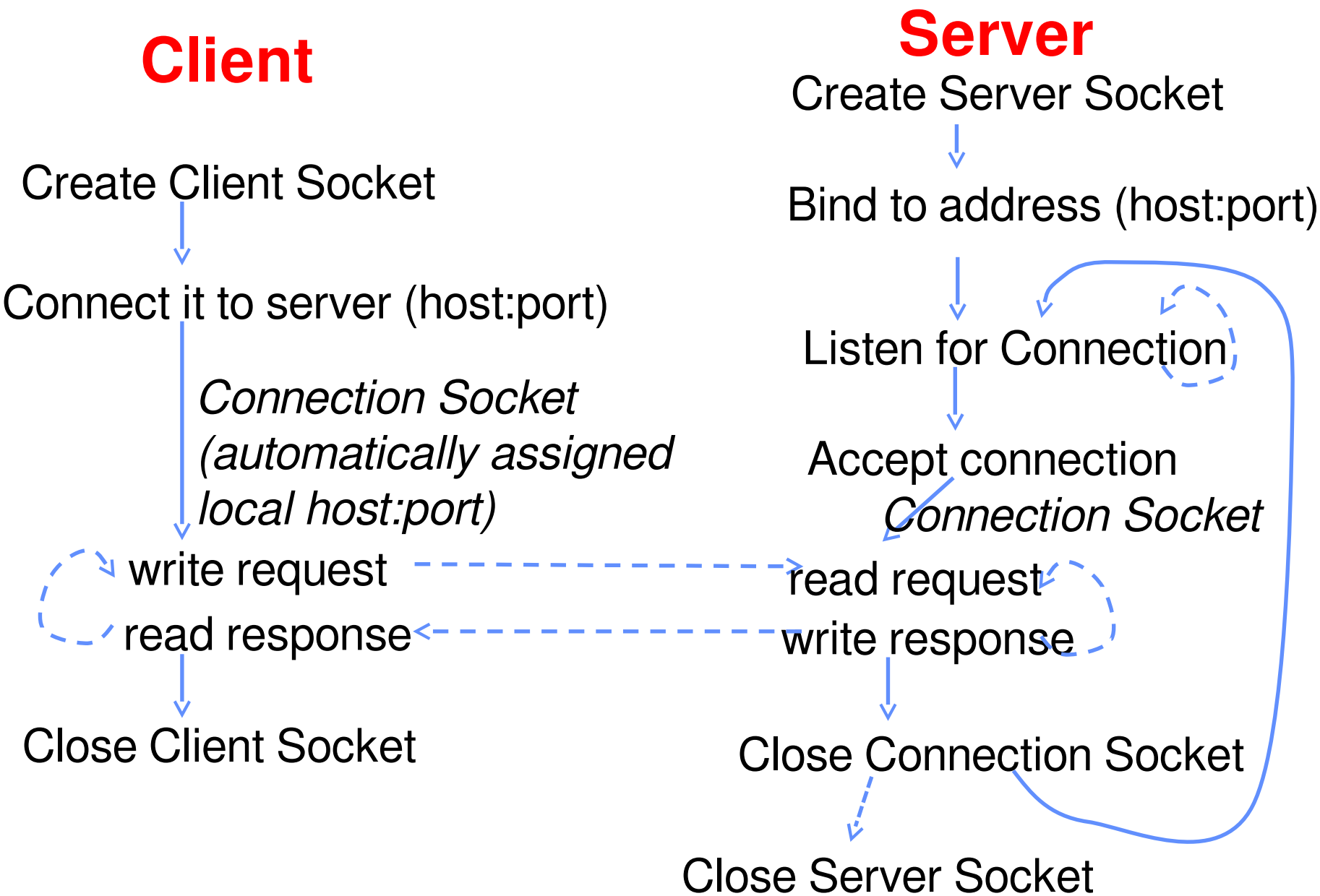
New API (NAPI): Use polling to receive packets (only some drivers)

Exit hard interrupt context as quickly as possible

- Do housekeeping and free up sent packets
- Schedule soft interrupt for further actions

Soft Interrupts: Handles receiption and delivery

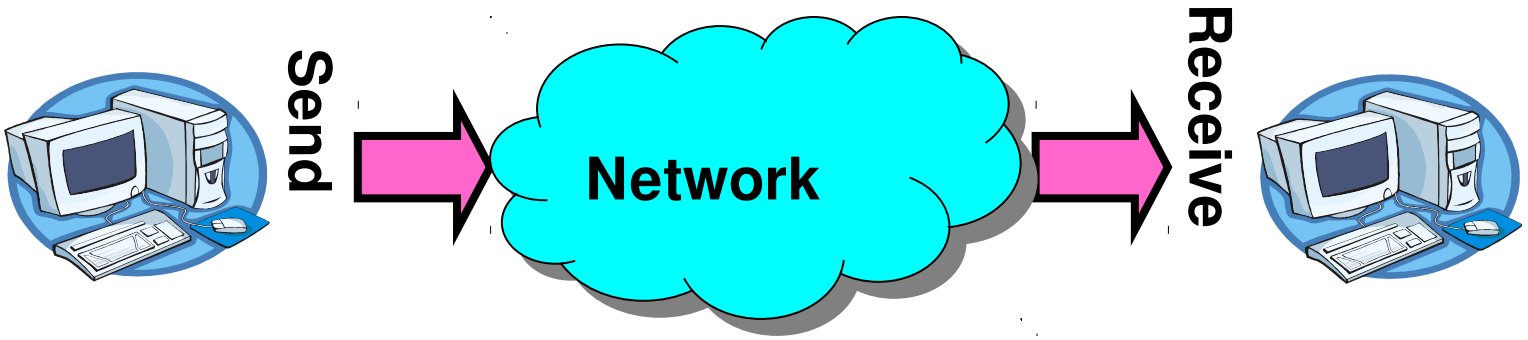
Recall: Sockets in concept



Recall: The Mailbox Abstraction (2)

Interface:

- Mailbox (mbox): temporary holding area for messages
  - Includes both destination location and queue
- Send(message,mbox)
  - Send message to remote mailbox identified by mbox
- Receive(buffer,mbox)
  - Wait until mbox has message, copy into buffer, and return
  - If threads sleeping on this mbox, wake up one of them



Mailboxes on TCP (1)

Want ***messages***, have **stream**

Need to separate messages

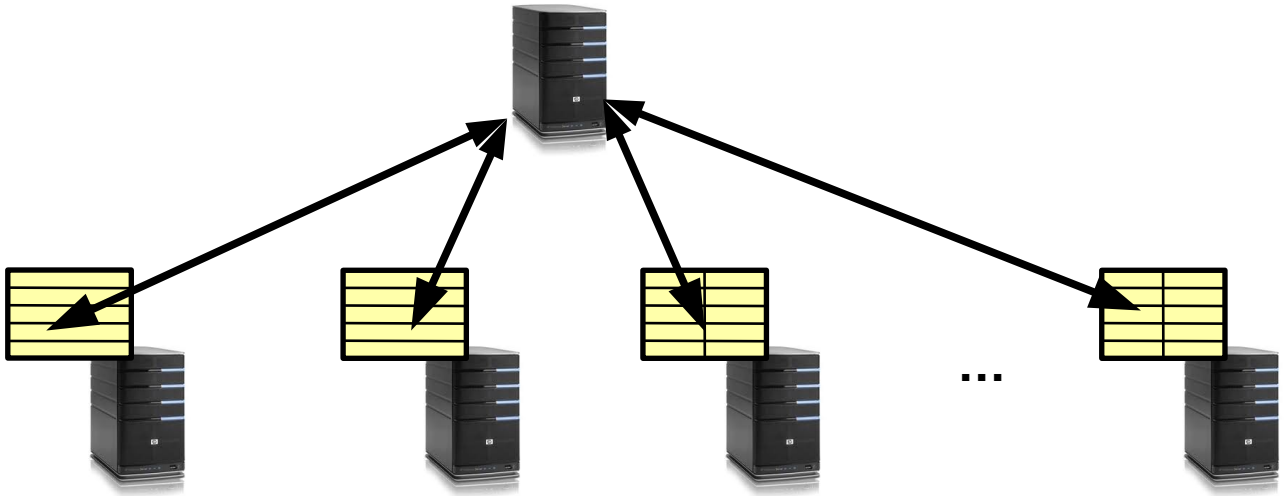
Same idea as storing multiple things in a file

Example: newline after each message

Example: size before each message

Mailboxes on TCP (2)

Want to send to many mailboxes



Solution: one connection for each destination

Reading from multiple sockets at once?

- Multiple threads + local synchronization
- select()/poll() – POSIX interface to wait for multiple files to have data

Remote Procedure Calls

Idea: make communication look like function calls

Wrapper library like for system calls

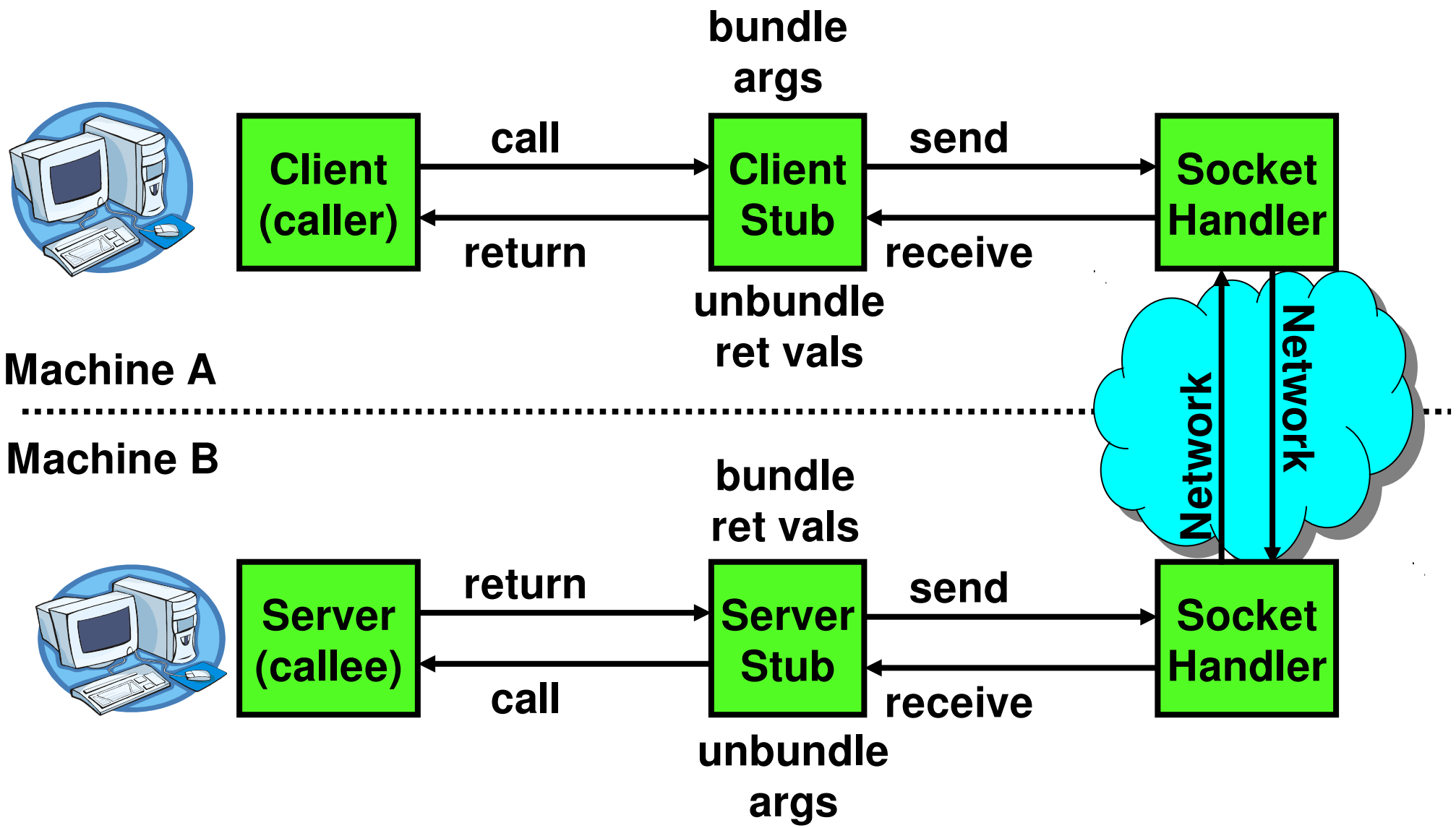
- Called ***stubs***

Also wrappers at the receiving end

- Read messages from socket, make function calls

Look "like" local function calls

RPC Information Flow



RPC (Pseudo)code

Client

```
#include <myprotocol.stubs.h> /* generated by tool */
RPCContext ctx = myprotocol_ConnectToServer(hostname, port);
…
result = myprotocol_mkdir(ctx, "/directory/name");
…
```

Server

```
#include <myprotocol.stubs.h>
main() {
  myprotocol_SetupRPCServer(port);
}
int real_myprotocol_mkdir(RPCContext ctx, char *name)
{
   …
}
```

RPC Details

Setup: Need to specify remote machine somehow

- Example: host:port

Need to ***marshall*** arguments over the network:

- Sometimes also called ***serialization***
- Done by stub

Typically code generated from **file specifying protocol**

- Called **Interface Definition Language (IDL)**
- Generates stubs
- ... including marshalling/demarshalling code

Interface Definition Language

Pseudocode:

```
protocol myprotocol {
  1: int32 mkdir(string name);
  2: int32 rmdir(string name);
}
```

Marshalling example: `mkdir("/directory/name")` `returning 0`

Client sends: `\001/directory/name\0`

Server sends: `\0\0\0\0`

Marshalling Details

Marshalling with different architectures

- Remember endianness?
- Need to choose a consistent format
- Option: Native format, mark

Marshalling with pointers

- Need to chase pointers
- Something like a graph? Doubly linked list? Gets complicated.

RPC Naming

One option: Well known ports

Another: "Dynamic binding"

- well-known service gives readable names
- NFS (Network File Server) uses this
- just specify the server name and RPC service
- allows multiple services on the same port

RPC: Really Like a Function Call? (1)

What if something fails?

```
result = myprotocol_mkdir(ctx, "/directory/name");
```

What should result be?

Did the server make the directory?

RPC: Really Like a Function Call (2)

RPC for high-level POSIX IO?

```
RemoteFile* rfh = remoteio_open(ctx, "foo.txt");
remoteio_puts(ctx, rfh, "Text.\n");
remoteio_close(ctx, rfh);
```

What happens ***if client fails***? Will the file be left open forever?

Note: not a problem for normal applications – the process dies, its files are closed.

RPC: Really Like a Function Call? (3)

```
for (big list of directories) {
  myprotocol_mkdir(ctx, current-name);
}
```

Local procedure call: ~1 ns

Local system call: ~100 ns

Network part ***only*** of remote procedure call:

- with typical ***local*** network: >400 000 ns
- with ***exceptionally fast*** local network: 2 600 ns

RPC Locally

Doesn't need to be used between different machines – maybe just different address spaces

Gives **location transparency**

- Move service whereever convenient
- Easier to treat one machine as a big machine

Much faster implementations available locally

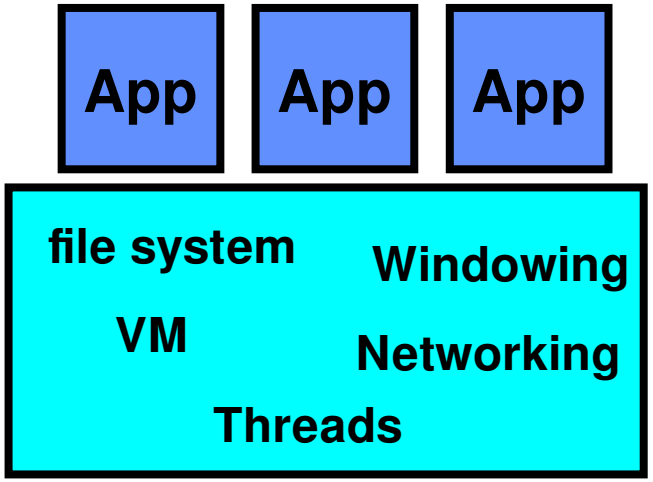
- **Shared memory**

Interlude: Microkernels (1)

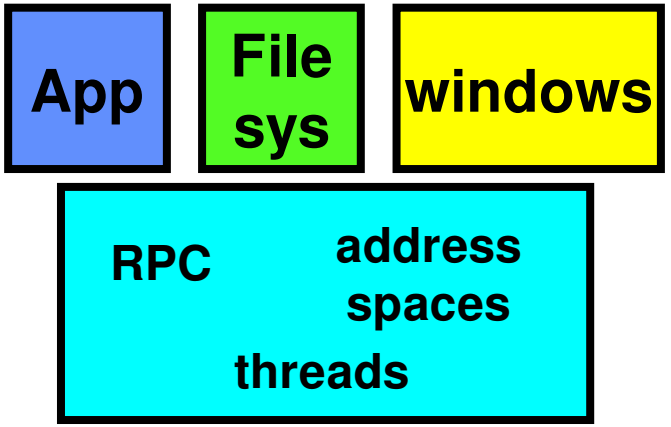
Split OS into **separate processes**

- Example: Filesystem, Network driver is external process

Use RPC to these components instead of system calls



Monolithic Structure

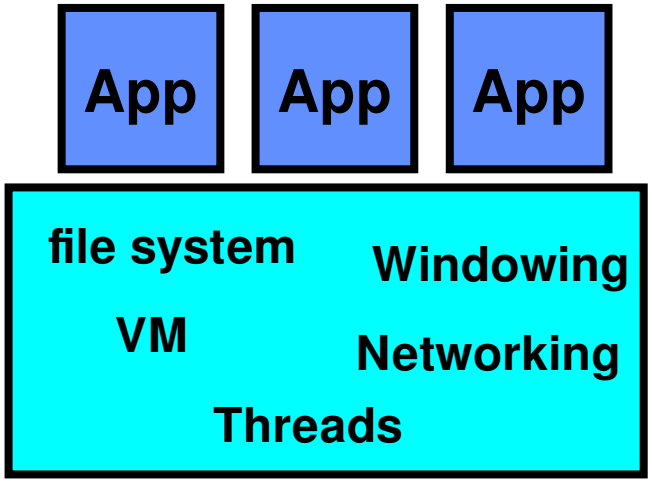


Microkernel Structure

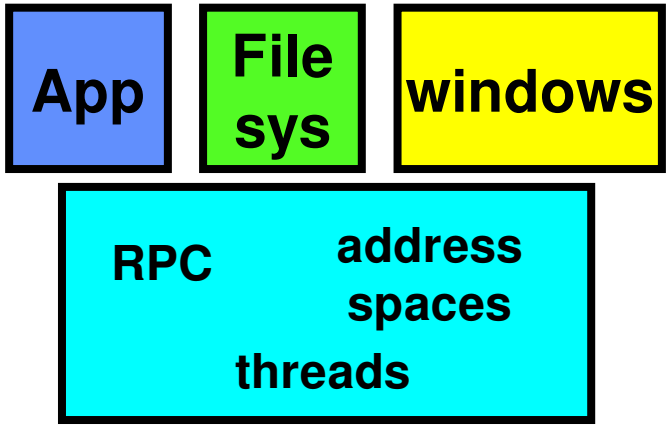
Interlude: Microkernels (2)

Microkernel provides **only services it "needs" to**:

- communication
- address space management
- thread scheduling
- almost-direct access to devices for service processes



Monolithic Structure



Microkernel Structure

Interlude: Why Microkernels?

Pro

- **Failure isolation**
- Easier to update/replace parts
- Easier to distribute – way to build OS that runs "across many machines"

Con

- More communication overhead/context switches
- Maybe harder to program?

Recall: What is an operating system?

Always:

- ***Memory management***
- I/O management
- ***CPU scheduling***
- ***Communication?***

Core OS service – not provided by kernel in "true" microkernel design

Sometimes:

- Filesystems
- ? Multimedia support
- ? User interface
- ? Internet browser

Microkernels in the Wild

Does anyone use a microkernel?

- Yes. Some embedded-targetted systems (Symbian, QNX, (se)L4)
- Some less "pure" microkernel than others (maybe some device drivers not separate services)

Many OSs provide *some services* like a microkernel

- OS X, Linux: Windowing

Some OSs started with microkernels

- Windows NT/XP/2000+/7/8: originally microkernel design (changed for performance?)
- OS X: hybrid of Mach microkernel and FreeBSD monolithic kernel

Distributed Operating Systems

Microkernel design lets OS be placed across a set of machines – just implement RPC, applications don't need to know

Questions:

- What about machine failures?
- What about the slowness of intermachine calls?
- What about sharing memory between machines?
- What about moving processes to/starting processes on remote machines?

Distributed Operating Systems

Microkernel design lets OS be placed across a set of machines – just implement RPC, applications don't need to know

Questions:

- Wh
- What about the slowness of intermachine calls?
- ***What about sharing memory between machines?***
- ***What about moving processes to/starting processes on remote machines?***

Good solutions to these problems – out of scope for this course

Distributed Operating Systems

Microkernel design lets OS be placed across a set of machines – just implement RPC, applications don't need to know

Questions:

- ***What about machine failures?***
- ***What about the slowness of intermachine calls?***
- Wha
- Wha s on remo

Can live with these problems, but ... not great.

More recently: fault-tolerant designs, application awareness of locality

Summary:

Network: physical connection that allows two computers to communicate

- Packet: sequence of bits carried over the network

Broadcast Network: Shared Communication Medium

- Transmitted packets sent to all receivers
- Arbitration: act of negotiating use of shared medium
  - Ethernet: Carrier Sense, Multiple Access, Collision Detect

Point-to-point network: a network in which every physical wire is connected to only two computers

- Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.

Summary:

Protocol: Agreement between two parties as to how information is to be transmitted

Internet Protocol (IP)

- Used to route messages through routes across globe
- 32-bit addresses, 16-bit ports

DNS: System for mapping from names to IP addresses

- Hierarchical mapping from authoritative domains

Remote Procedure Call (RPC): Call procedure on remote machine

- Provides same interface as procedure
- Automatic packing and unpacking of arguments without user programming (in stub)

Summary (1/2)

- Network: physical connection that allows two computers to communicate
  - Packet: sequence of bits carried over the network
- Broadcast Network: Shared Communication Medium
  - Transmitted packets sent to all receivers
  - Arbitration: act of negotiating use of shared medium
    - » Ethernet: Carrier Sense, Multiple Access, Collision Detect
- Point-to-point network: a network in which every physical wire is connected to only two computers
  - Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- Protocol: Agreement between two parties as to how information is to be transmitted
- Internet Protocol (IP)
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports
- DNS: System for mapping from names⇒IP addresses
  - Hierarchical mapping from authoritative domains
  - Recent flaws discovered

Summary (2/2)

- **TCP:** Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- **Two-phase commit:** distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- **Byzantine General's Problem:** distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$
- **Remote Procedure Call (RPC):** Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)

Recall: Use of Sockets in TCP

- **Socket:** an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server