CS162:
Operating Systems and Systems Programming

Lecture 18:
Transctions for Consistency (finish)
Networking: Sockets / IP (intro)

21 July 2015
Charles Reiss
https://cs162.eecs.berkeley.edu/

Recall:
Key to Node Mapping Example

m = 6 / ID space: 0..63

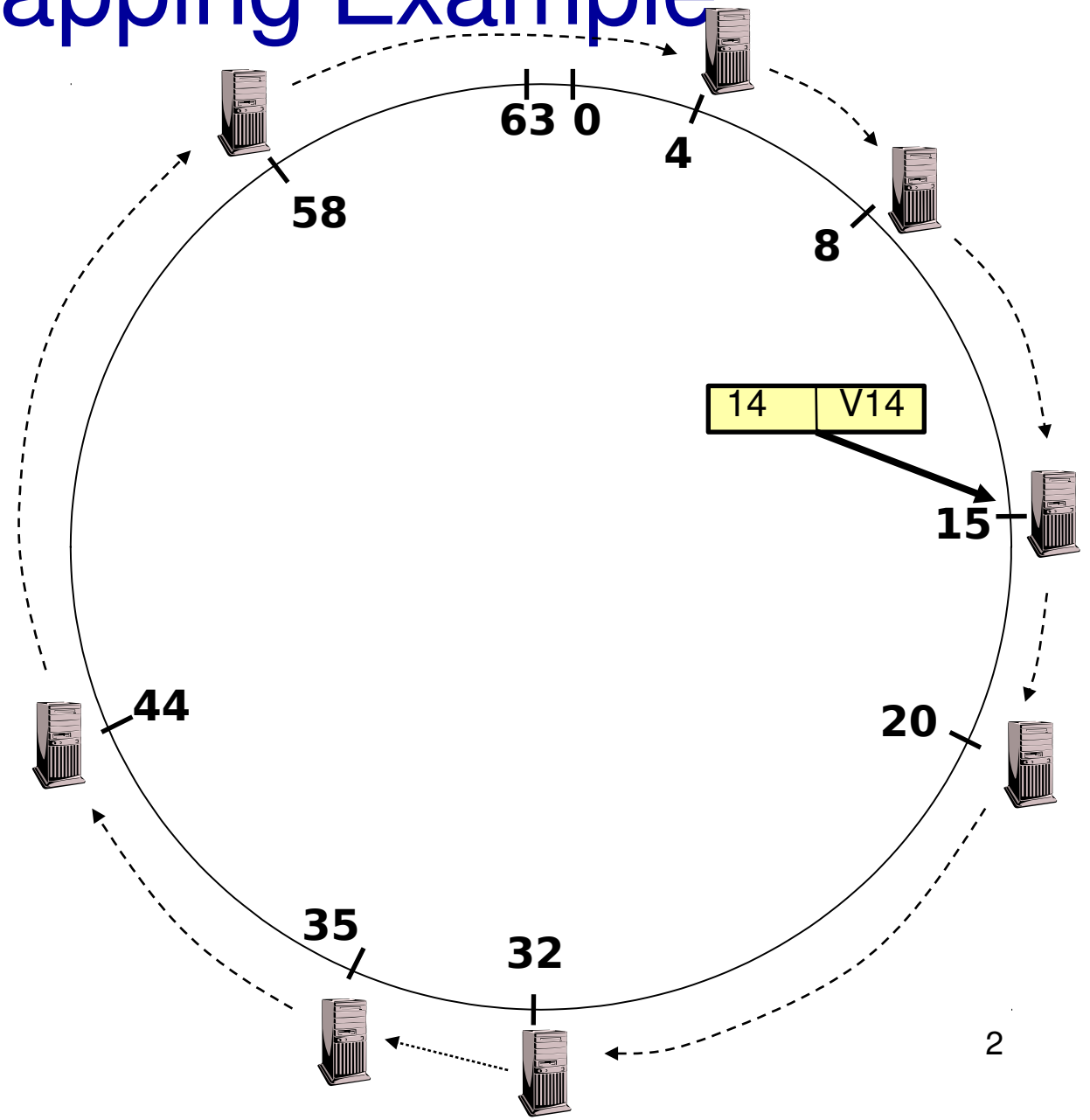
Node 8 maps keys [5,8]

Node 15 maps keys [9,15]

Node 20 maps keys [16, 20]

…

Node 4 maps keys [59, 4]



Recall: (Distributed) Consistency

Everyone agrees on the state of the system:

- Won't depend on who you ask
- Won't depend on if nodes go down

Transcation idea:

- **Atomic** change of state everywhere
- Once it happens, never taken back

CAP theorem: perfect consistency and perfect availability impossible

- Two-phase commit: **stall** system instead of letting disconnected node get out of sync

Recall: Two-Phase Commit

Voting protocol – requires unanimity

Transaction logically committed if and only if

- All workers and coordinator decide to vote to commit

Key idea: nodes **never** take back their vote

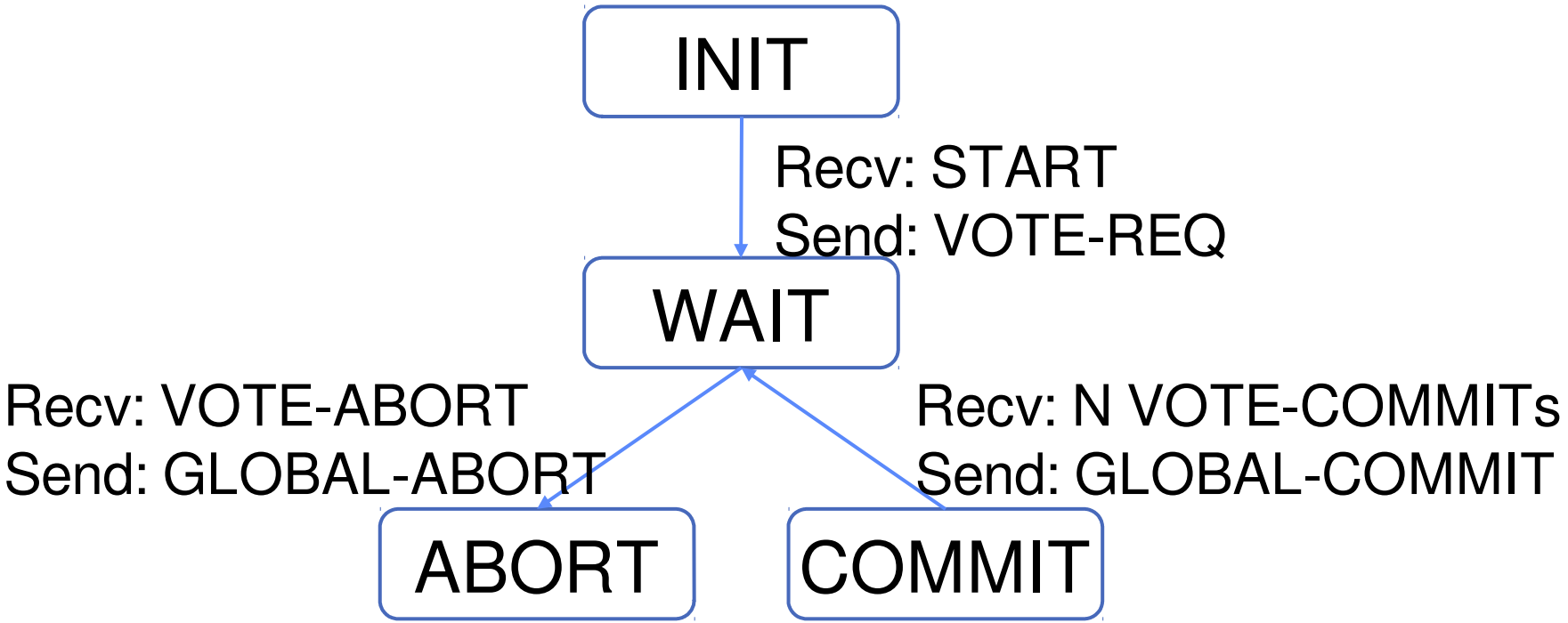
- On failure: need to recover same votes

Nodes work in lock step (for an item):

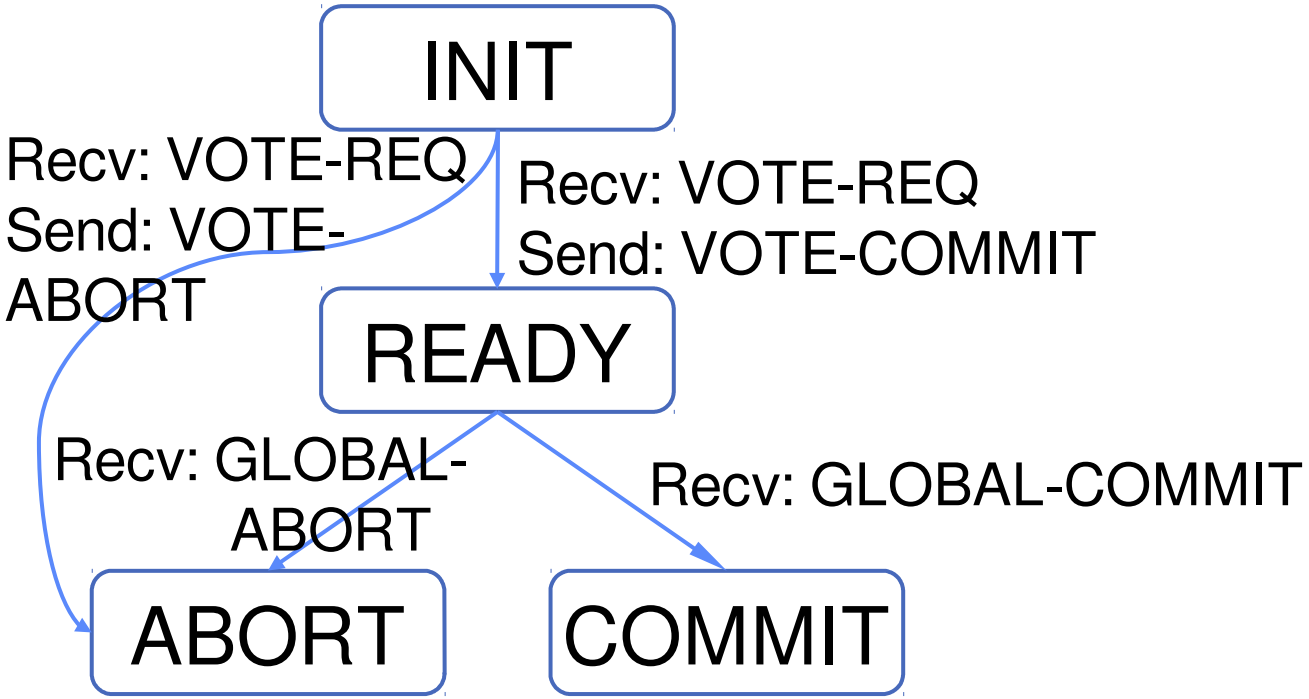
- Don't perform new transactions until old one resolved
- **Stall** until transaction is resolved

Recall: State Machine of Coordinator

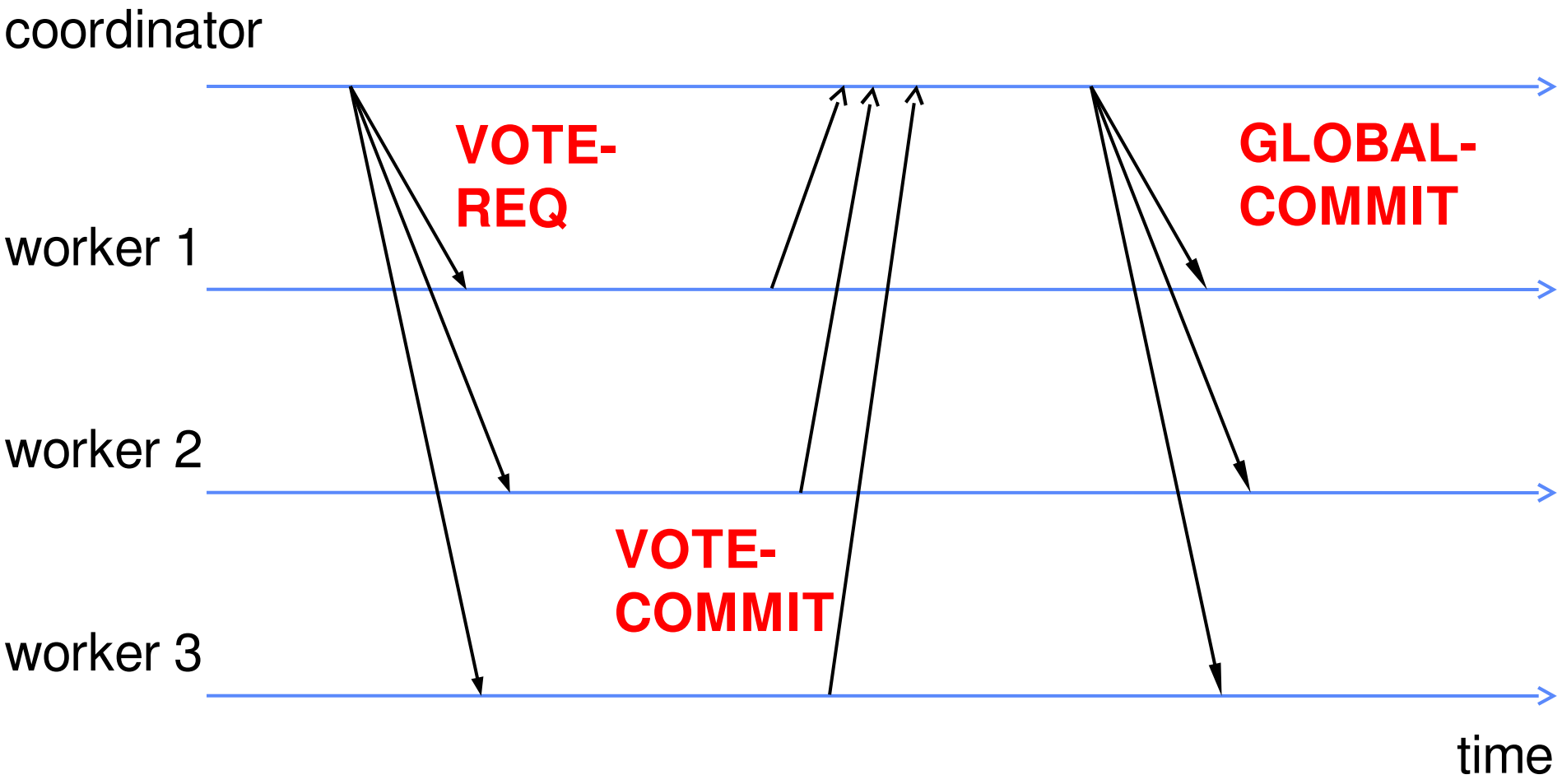
Coordinator implements simple state machine:



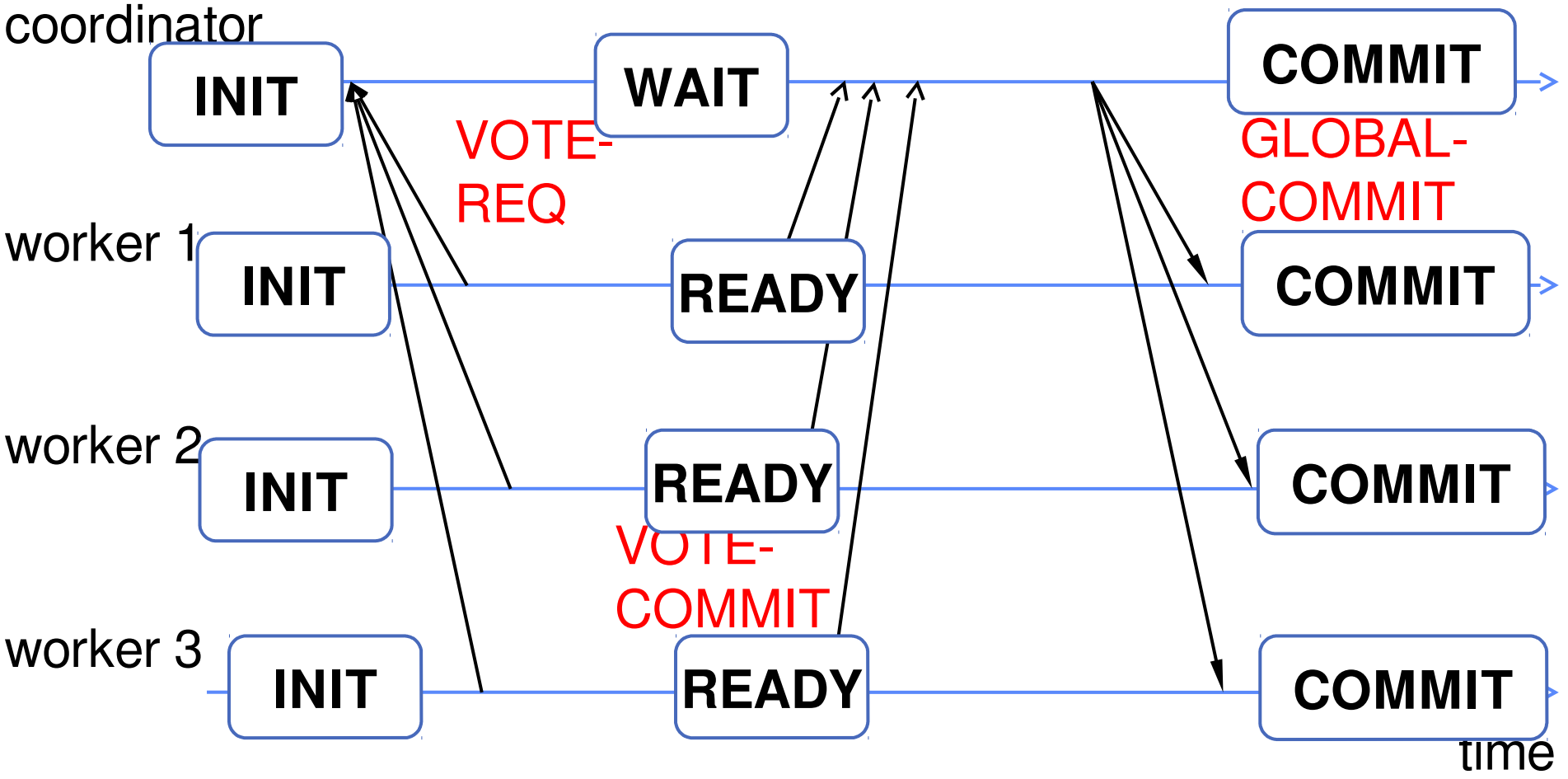
Recall: State Machine of Workers



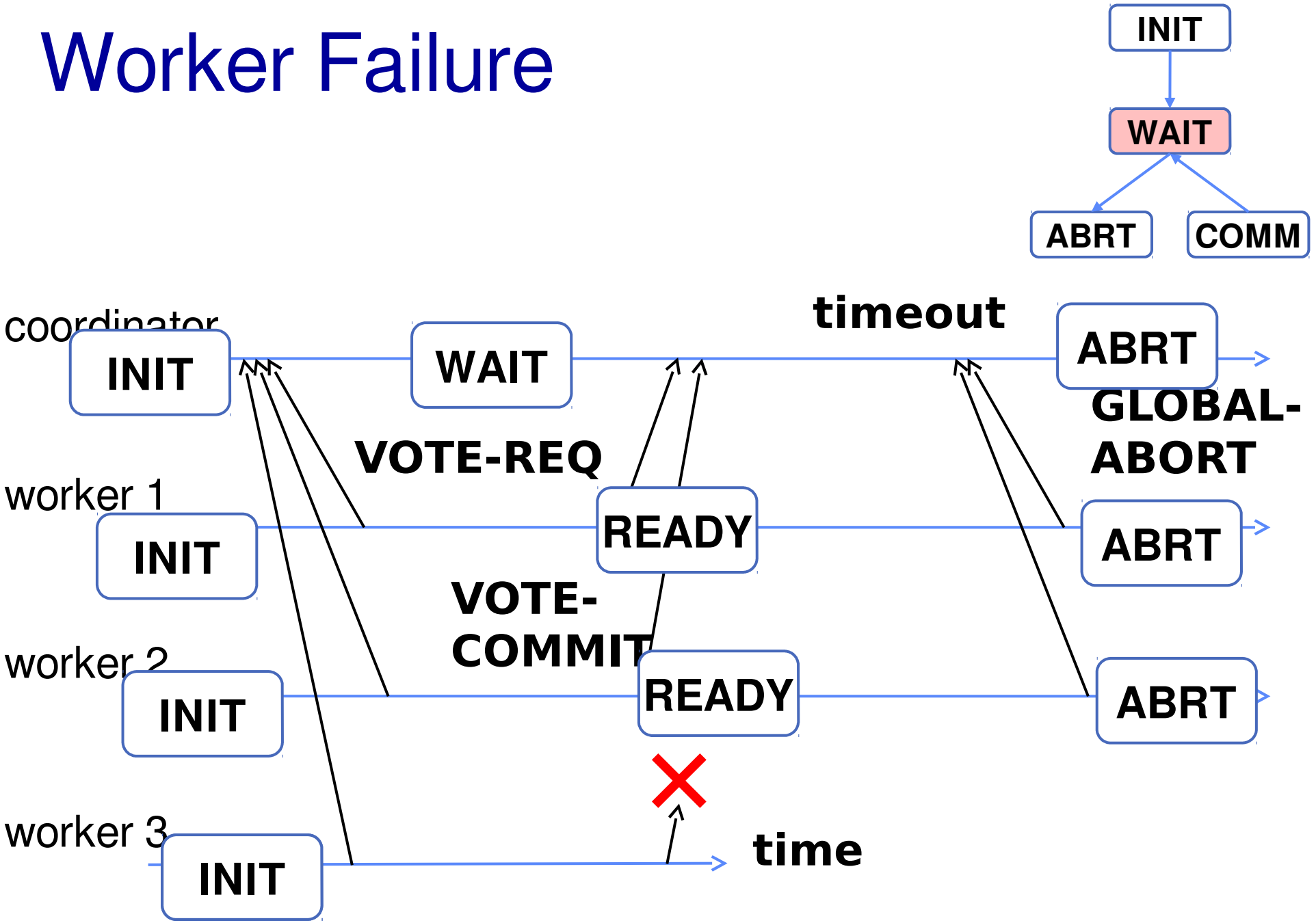
Failure Free Example Execution



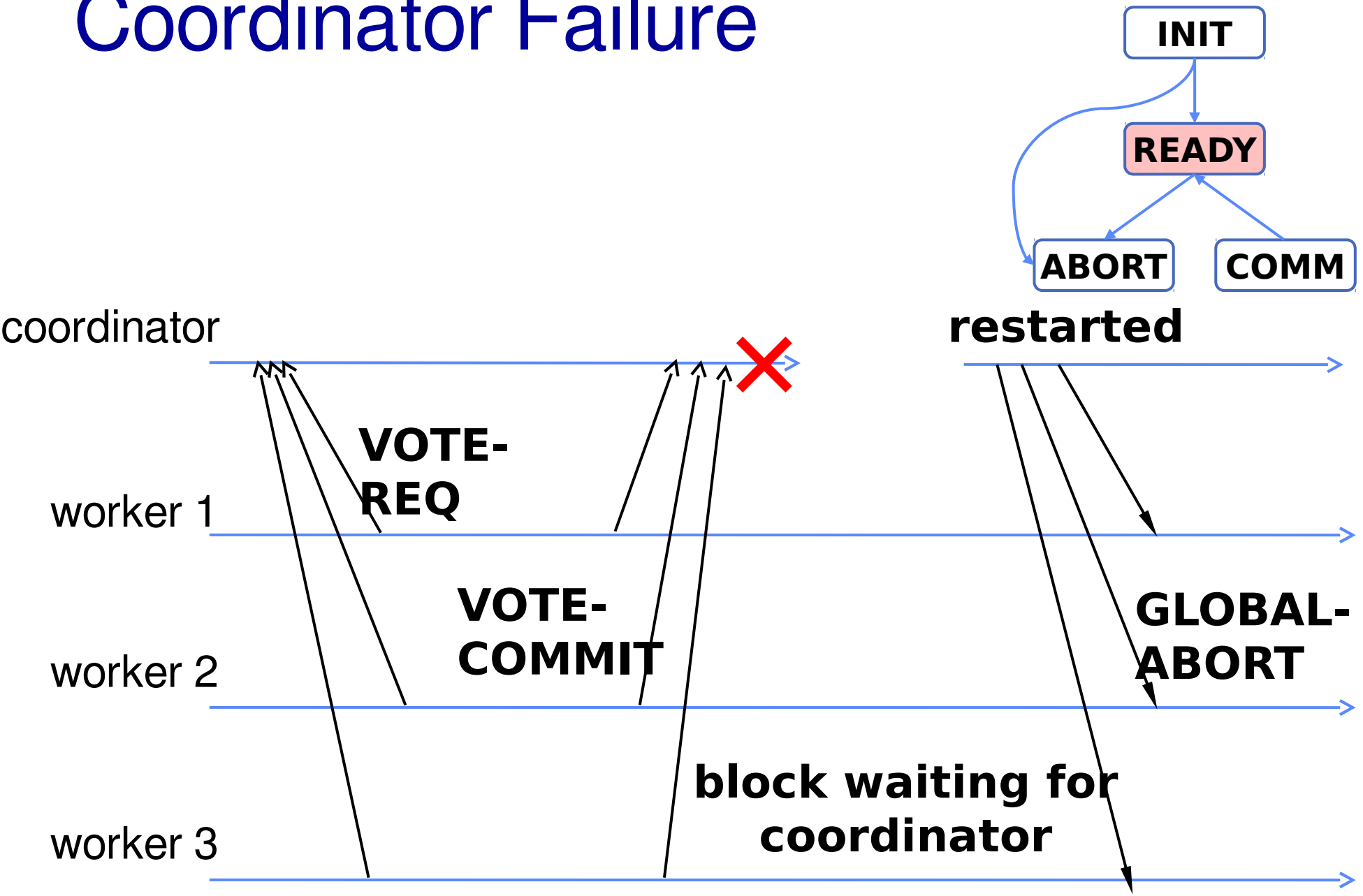
Failure Free Example Execution



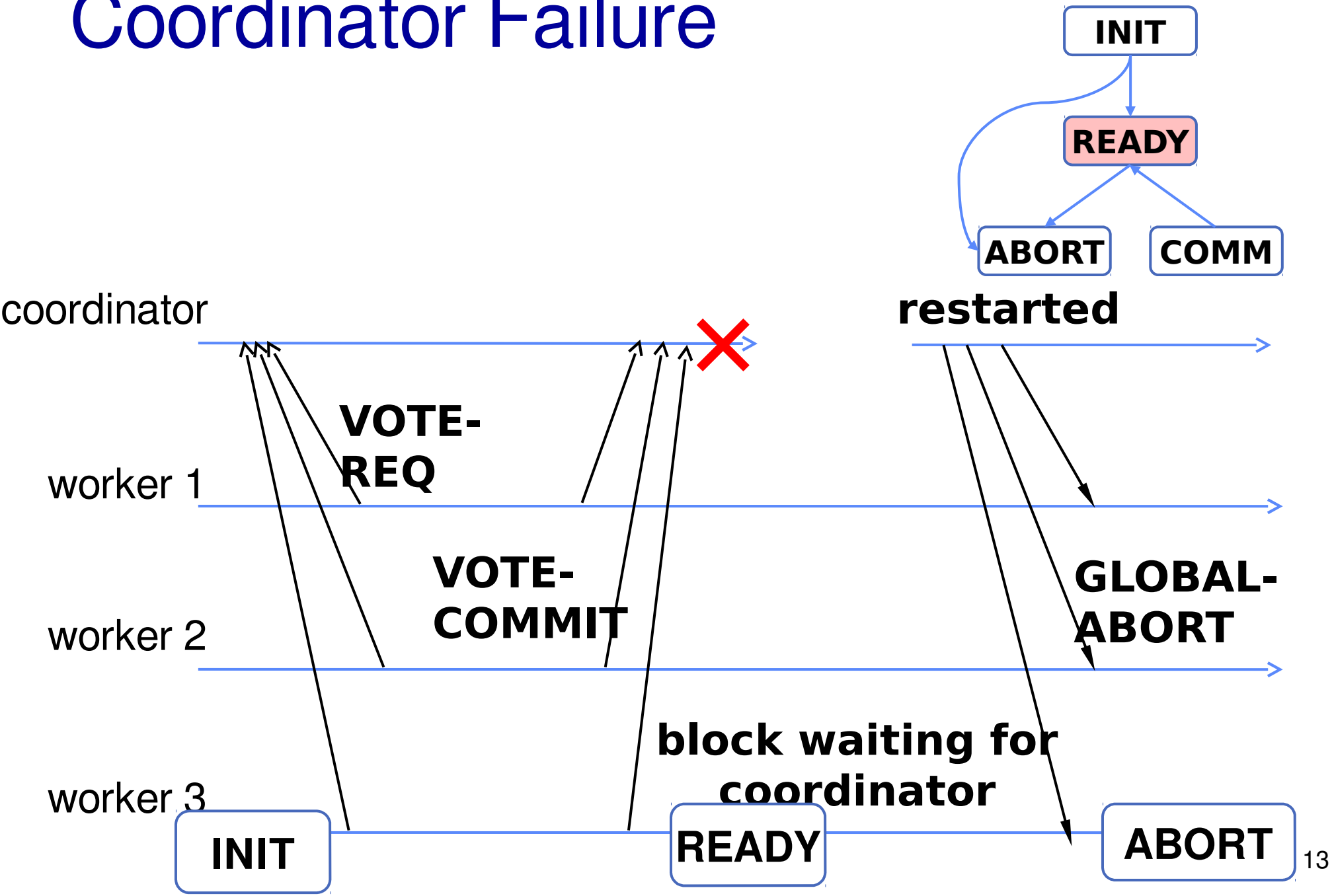
Worker Failure



Coordinator Failure



Coordinator Failure



Failure Recovery (1)

Nodes need to **know what state they are in** when they come back from a failure

How? One way: Logs on hard disk. More when we talk about filesystems.

Then recovery rules:

- Coordinator: If was in INIT, WAIT, or ABORT, goto ABORT (resend GLOBAL-ABORT)
- Coordinator: If in COMMIT, goto COMMIT

Failure Recovery (2)

Recovery rules when coming back from failure:

Coordinator:

- Was in INIT, WAIT, or ABORT?
  Goto ABORT, resend GLOBAL-ABORT
- Was in COMMIT?
  Goto COMMIT, resend GLOBAL-COMMIT

Worker:

- Was in INIT, ABORT?
  Goto ABORT, resend VOTE-ABORT
- Was in COMMIT?
  Goto COMMIT, resend VOTE-COMMIT
- Was in READY?
  Goto READY, ask coordinator for another copy of the VOTE-REQ message

Consistency with TPC

Simplify: One key, initially A

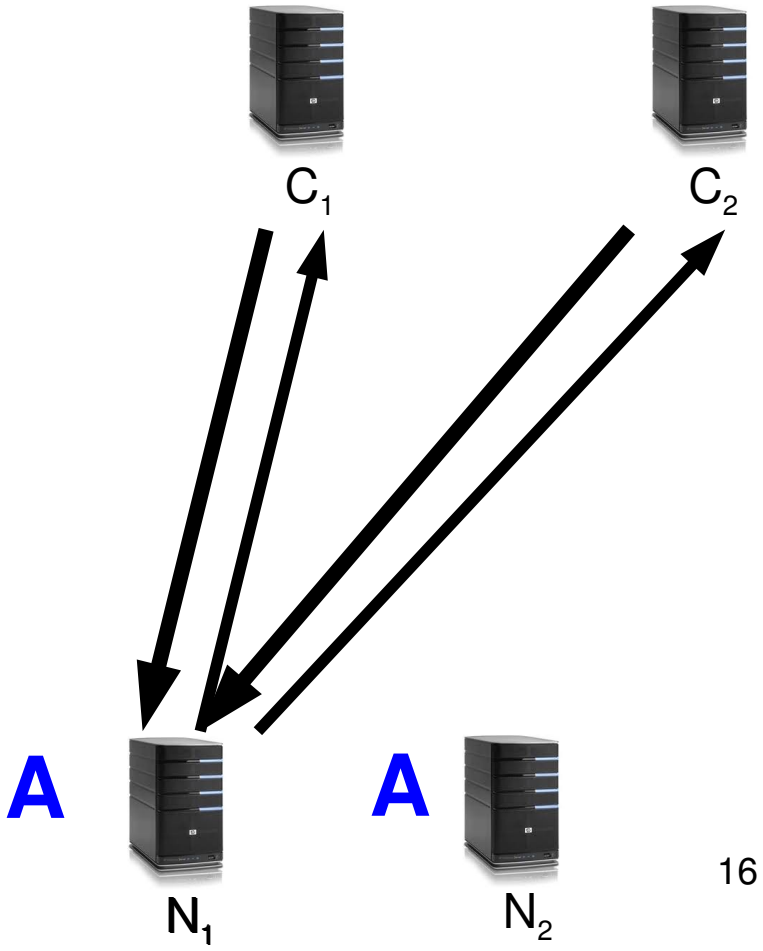
N***=2,*** W=2, R=2

Two clients

Timeline:

- C1 → N1: PREPARE C1:put(X)
- N1 → C1: VOTE-COMMIT C1:put(X)
- C2 → N1: PREPARE C2:put(Y)
- N1 → C2: VOTE-ABORT C2:put(Y)

So **C2 GLOBAL-ABORTs**



Consistency with TPC

Simplify: One ke

N***=2,*** W=2, R=2

Two clients

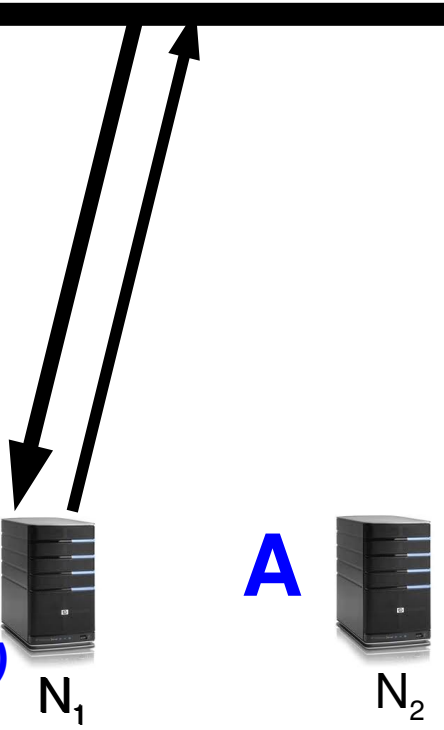
Timeline:

- C1 → N1: PREPARE C1:put(X)
- N1 → C1: VOTE-COMMIT C1:put(X)
- C2 → N1: PREPARE C2:put(Y)
- N1 → C2: VOTE-ABORT C2:put(Y)

So **C2 GLOBAL-ABORTs**

Why does N1 vote to abort?

Needs to track what it's agreed to do. (*Locking*.)

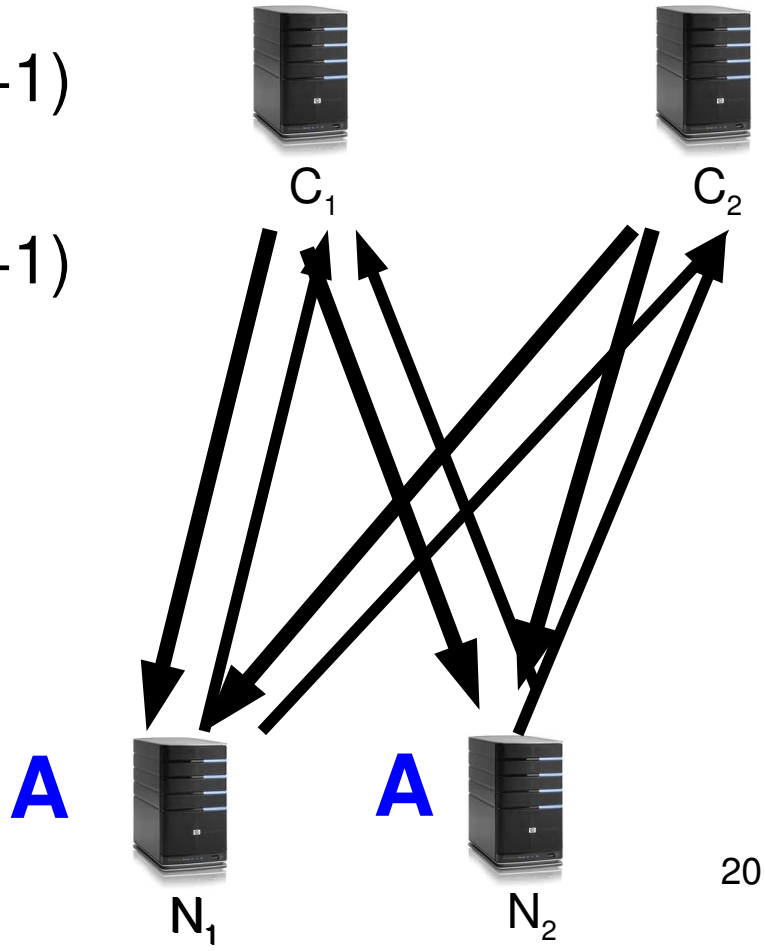


Consistency with TPC Take Two

Timeline:

- C1 → N1: get() → A
- C2 → N2: get() → A
- C1 → N1, N2: PREPARE C1:put(A+1)
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:put(A+1)
- N1, N2 → C2: VOTE-COMMIT

C1 and C2 didn't see each other's updates?



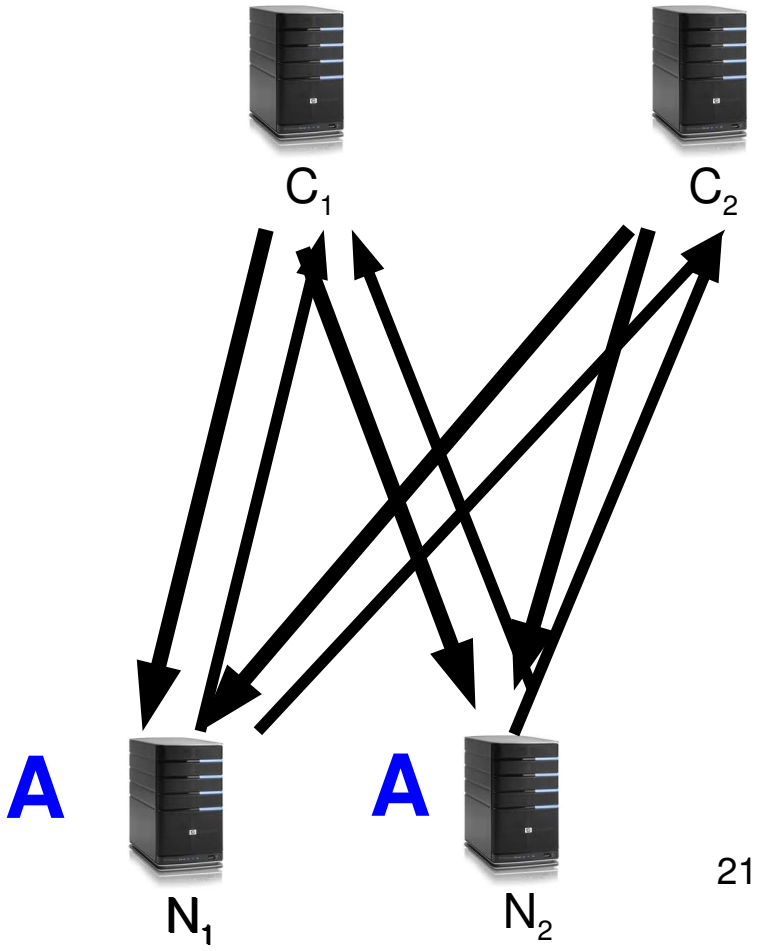
Consistency with TPC Take Three

Timeline:

- C1 → N1, N2: get() → PREPARE: C1:get()=A
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: get() → PREPARE: C2:get()=A
- N1, N2 → C2: VOTE-COMMIT
- C1 → N1, N2: PREPARE C1:put(A+1)
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:put(A+1)
- N1, N2 → C2: VOTE-COMMIT

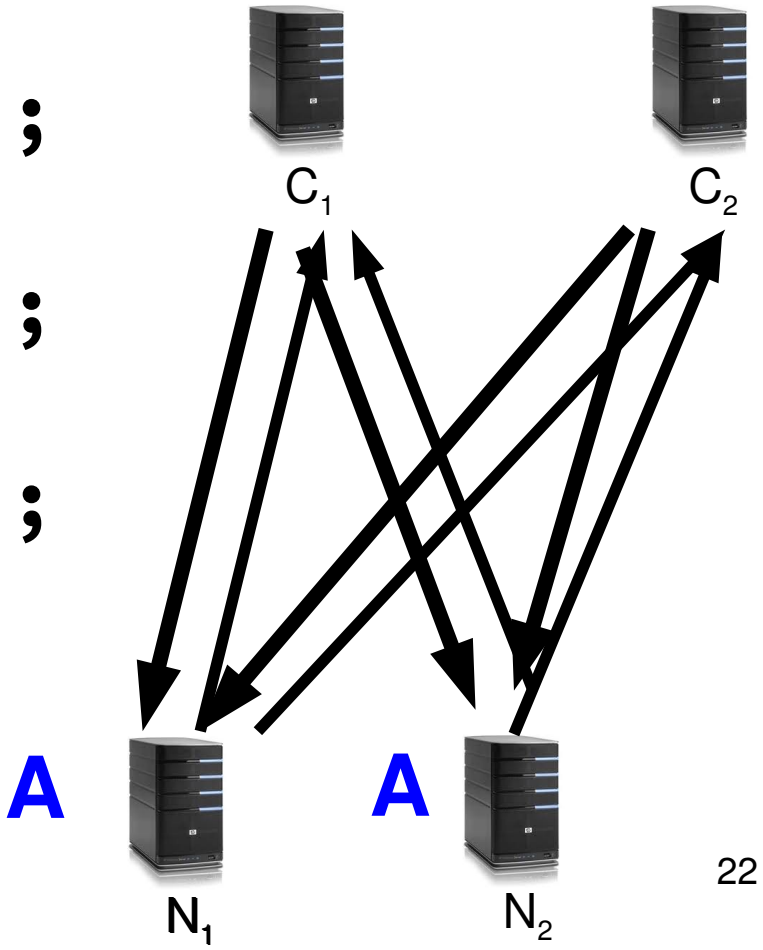
We did the read as a transaction – consistent and atomic!

C1 and C2 *still* didn't see each other's updates?



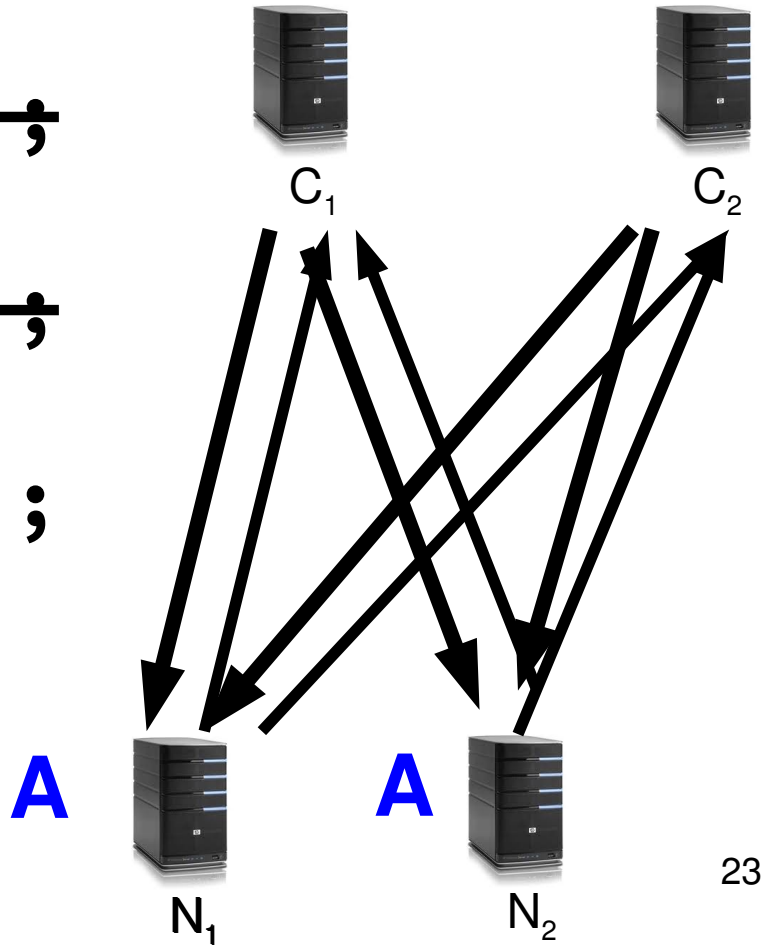
Recall: Atomic Load/Stores

```
lock_acquire(&value_lock);
temp = value;
lock_release(&value_lock);

lock_acquire(&value_lock);
value = temp + 1;
lock_release(&value_lock);
```



Recall: Atomic Load/Stores

```
lock_acquire(&value_lock);
temp = value;
lock_release(&value_lock);

lock_acquire(&value_lock);
value = temp + 1;
lock_release(&value_lock);
```

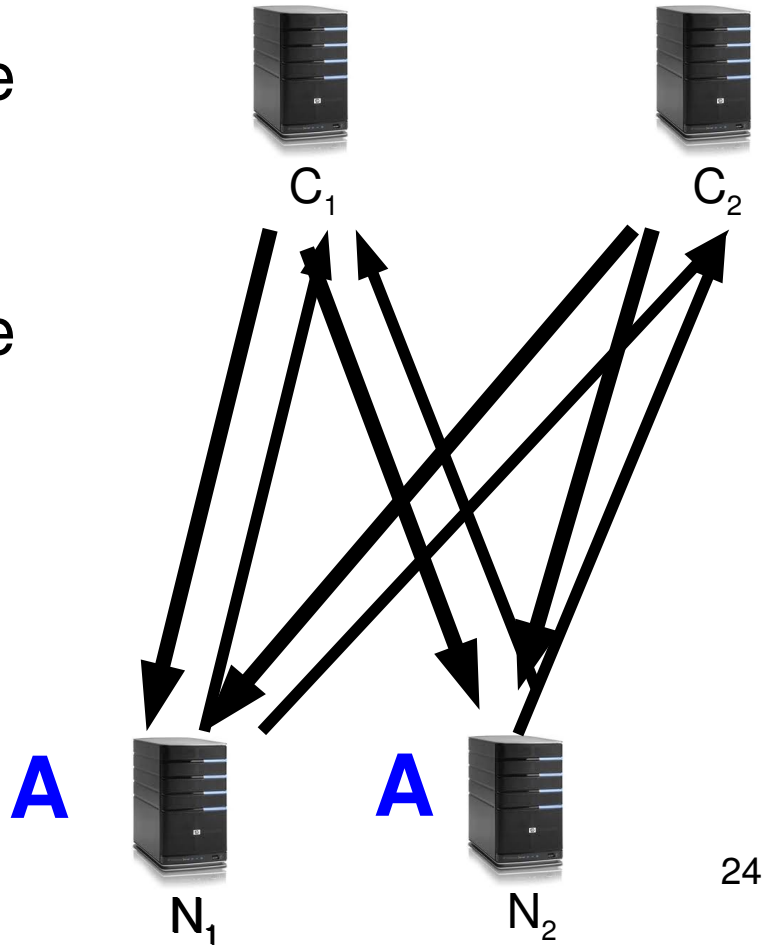


Consistency with TPC Take Four

Timeline:

- C1 → N1: get() → A
- C2 → N2: get() → A
- C1 → N1, N2: PREPARE C1:change from A to A+1
- N1, N2 → C1: VOTE-COMMIT
- C2 → N1, N2: PREPARE C2:change from A to A+1
- N1, N2 → C2: **VOTE-ABORT**

Transacation *includes entire update*



The ACID properties of Transactions

Atomicity: all actions in the transaction happen, or none happen

Consistency: transactions maintain data integrity, e.g.,

- Balance cannot be negative
- Cannot reschedule meeting on February 30

Isolation: execution of one transaction is isolated from that of all others; no problems from concurrency

Durability: if a transaction commits, its effects persist despite crashes

Isolation and Locking

Full answer in a databases class

Simplest solution: Always lock everything

- One transaction at a time

Generally: lock on on prepare, unlock on global-commit/abort

The CAP Theorem

Consistency – as if single serial order

Availability – system will accept writes/reads

Partition Tolerance – system will handle nodes failing

Choose (at most) **two**

CAP Theorem Choices

Two-phase commit with a **partition** (some nodes separated from rest of system or down):

- **Consistent** (reads never return wrong values)
- **Not available** (can't perform new transactions)

Distributed Agreement

Two-phase commit makes a **decentralized** decision

Example decision: Change value of a key.

Problem: What happens if a machine fails?

- Two-phase commit: ***blocking***

Two-Phase Commit: Blocking

A->B: Prepare to set value to 42

B: writes "transaction: set value to 42; agree-to-commit" to its log

B->A: Commit

A ***crashes***, loses message

B ***must wait until A comes back*** to use its value

Agreement in Face of Failure

Idea: if a majority of nodes agree, do it

If a minority don't participate, ignore them.

Fail-stop $\rightarrow$ non-agreeing nodes don't participate

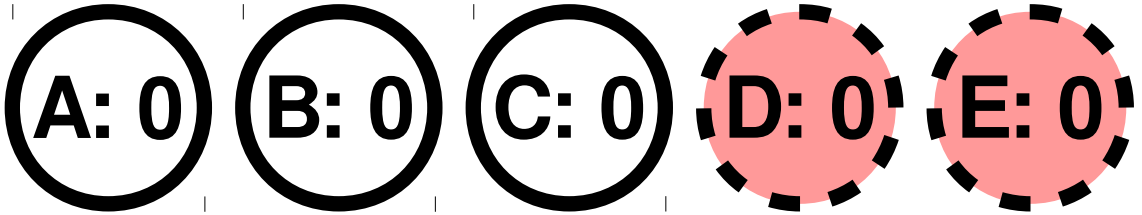
Algorithms that do this: Paxos, Raft

- ***very very very tricky***
- similar idea to two-phase commit

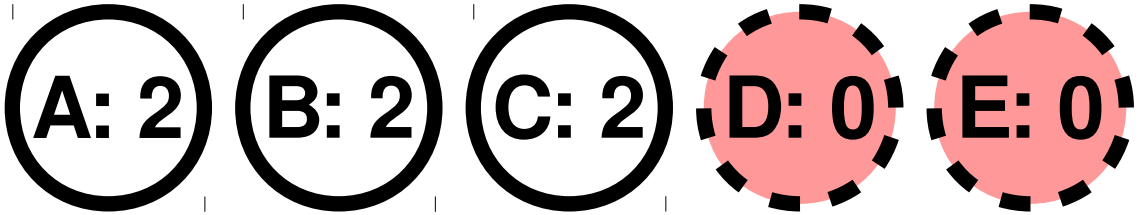
Why a majority? (1)

Key property: **Overlap** (like quorum consensus)

Suppose we use transactions to track a value, initially 0

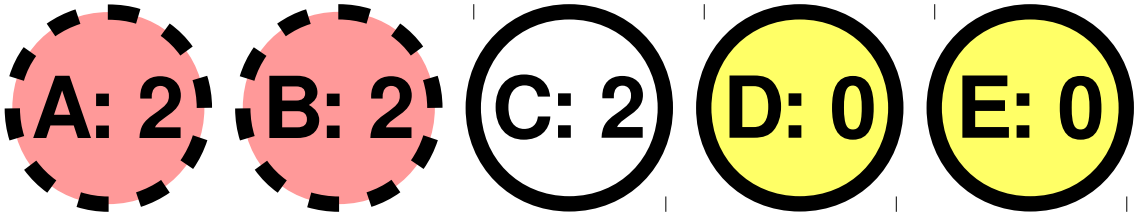


We run the transaction "+ 2" while D, E are down:



Why a majority? (2)

Now, D, E come back up and A, B go down:



Need **overlap** (C in in this case)

- Guaranteed by choice of majority

Overlap prevents us from ***losing*** transactions

- Means *every node* is responsible for resending missed updates

Beyond Fail-Stop

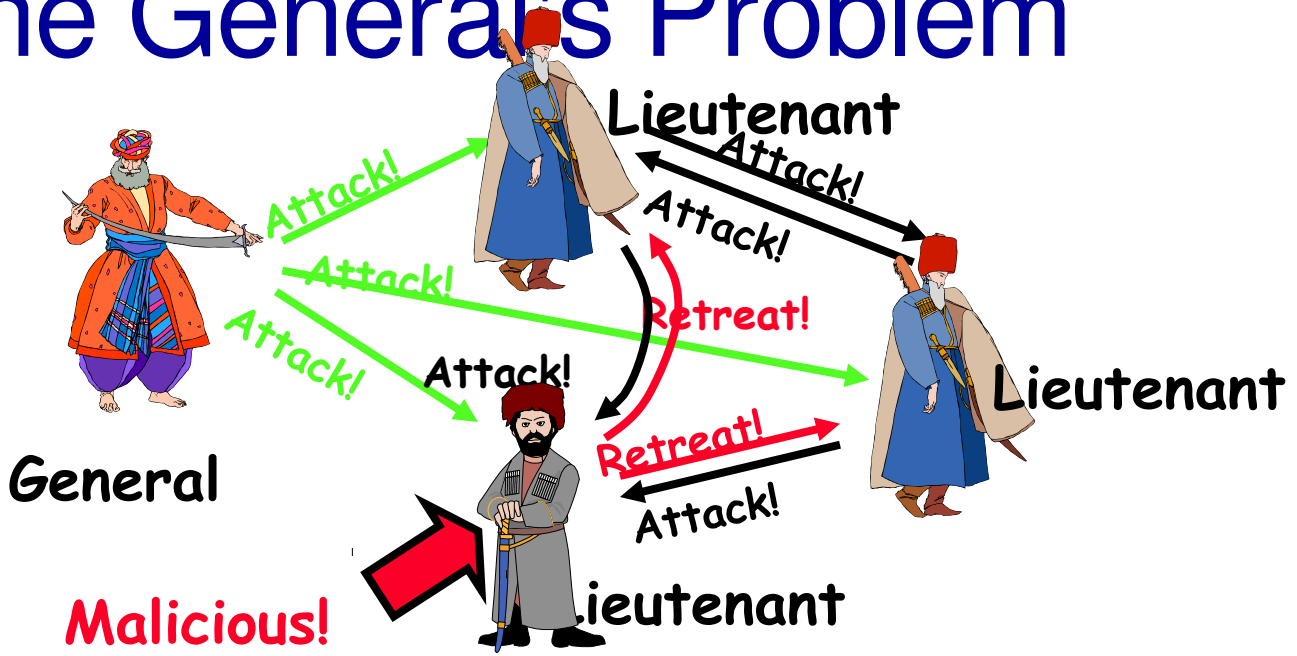
What if a minority of nodes send bad data? Selectively? Selectively drop messages?

What if a minority of nodes is malicious?

Can Paxos/Raft still work? **No.**

This is called ***Byzantine failures***

Byzantine General's Problem



One general, N-1 Lieutenants

Some number (F) want chaos

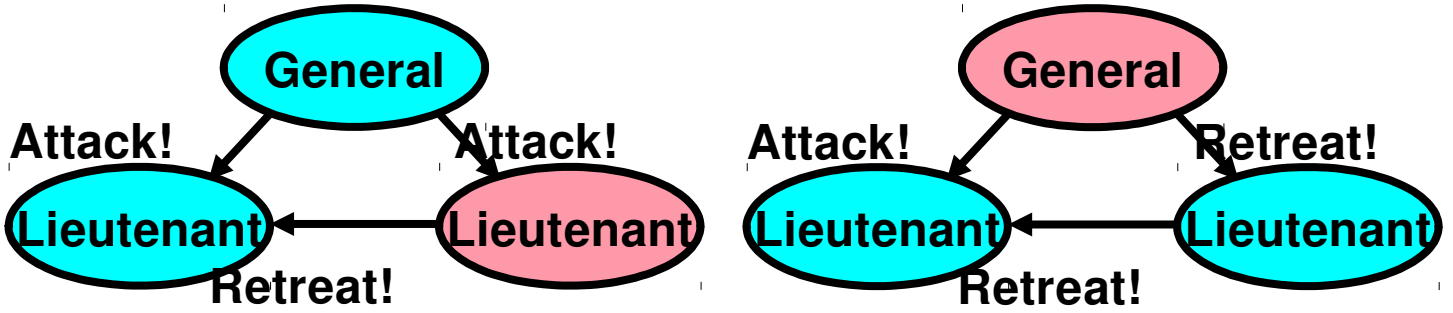
- Can say contradictory things!

Goal: General sends order and

- 1: All non-insane lieutenants obey the *same order*
- 2: If the general is not insane, they obey the General's order

Byzantine Generals: Impossibility

N=3: → **NO SOLUTION**



General theorem: **need** N >= 3F + 1

Byzantine Generals: Solutions

There are protocols that solve byzantine generals for N >= 3F + 1 (the lower bound).

Original algorithm: O(2^N) messages!

Castro and Liskov, "Practical Byzantine Fault Tolerance", O(**N^2**) messages

- Note: A lot worse than Paxos/Raft (failstop)
- Also a lot more complicated

Logistics

Break

Sockets

Talked about distributed systems sending *messages* between machines

What does the interface for that look like?

Recall: POSIX Low-level IO

open, read, write, close

Files are opaque **sequences of bytes**

Communication between processes

Files as communication channels

```
write(wfd, wbuf, wlen);
```

```
n = read(rfd,rbuf,rmax);
```

What if data written once and consumed once?

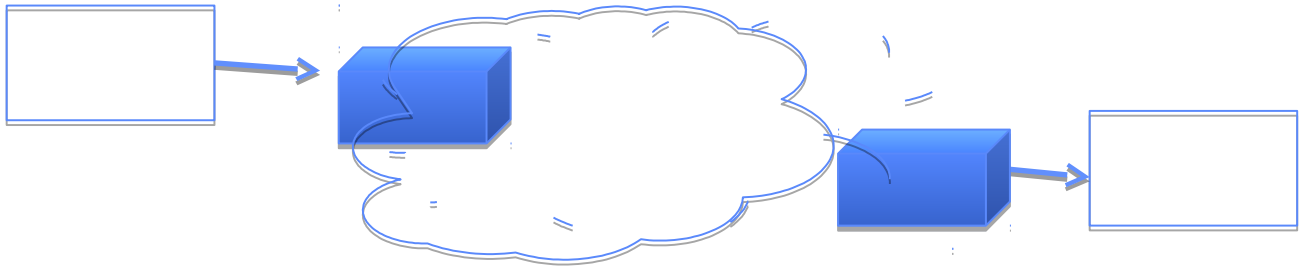
Can still ***look*** like regular file I/O!

Implemented with queue – throw away after read

Example: Unix pipe()

Communication Across the world looks like file IO

```
write(wfd, wbuf, wlen);
```



```
n = read(rfd,rbuf,rmax);
```

Sockets: Connected queues over the Internet

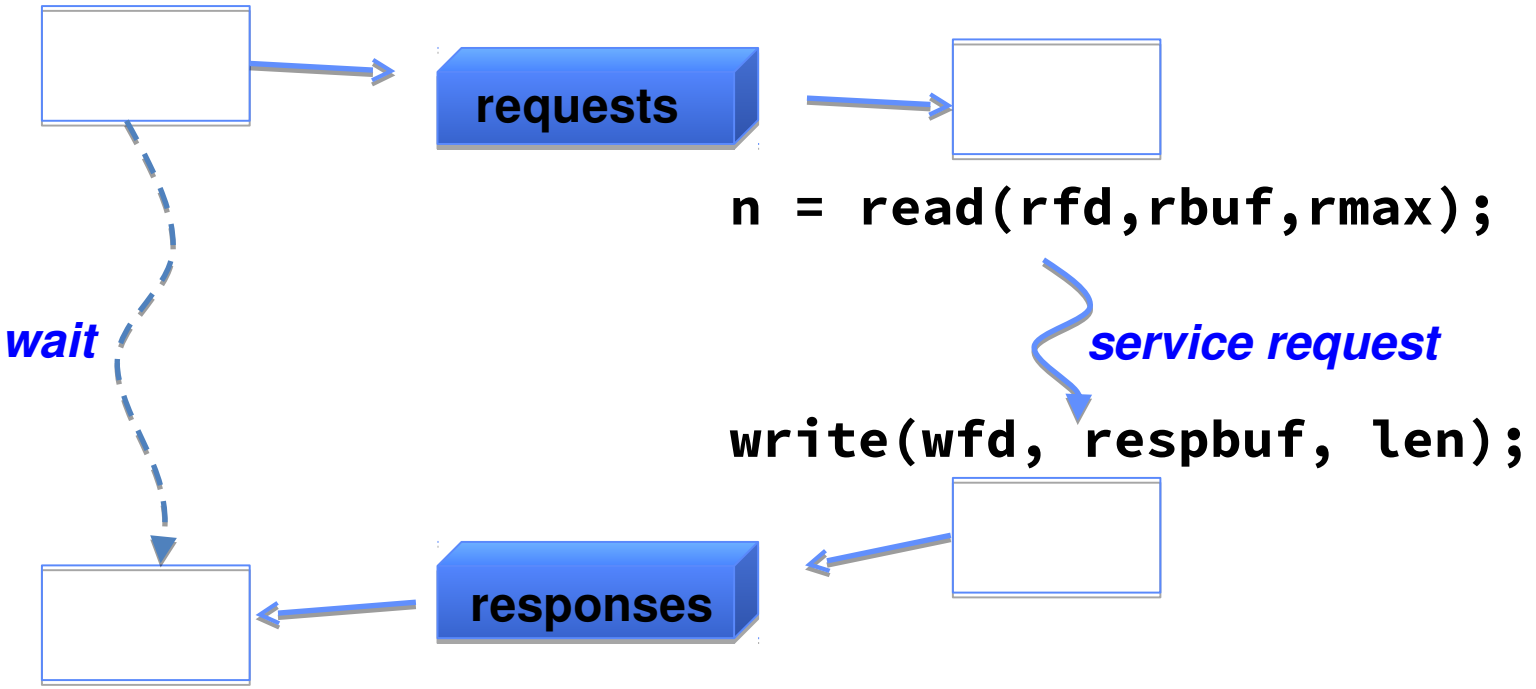
- How to open()? Filenames?
- How are they connected in time?

Request/Response Protocol

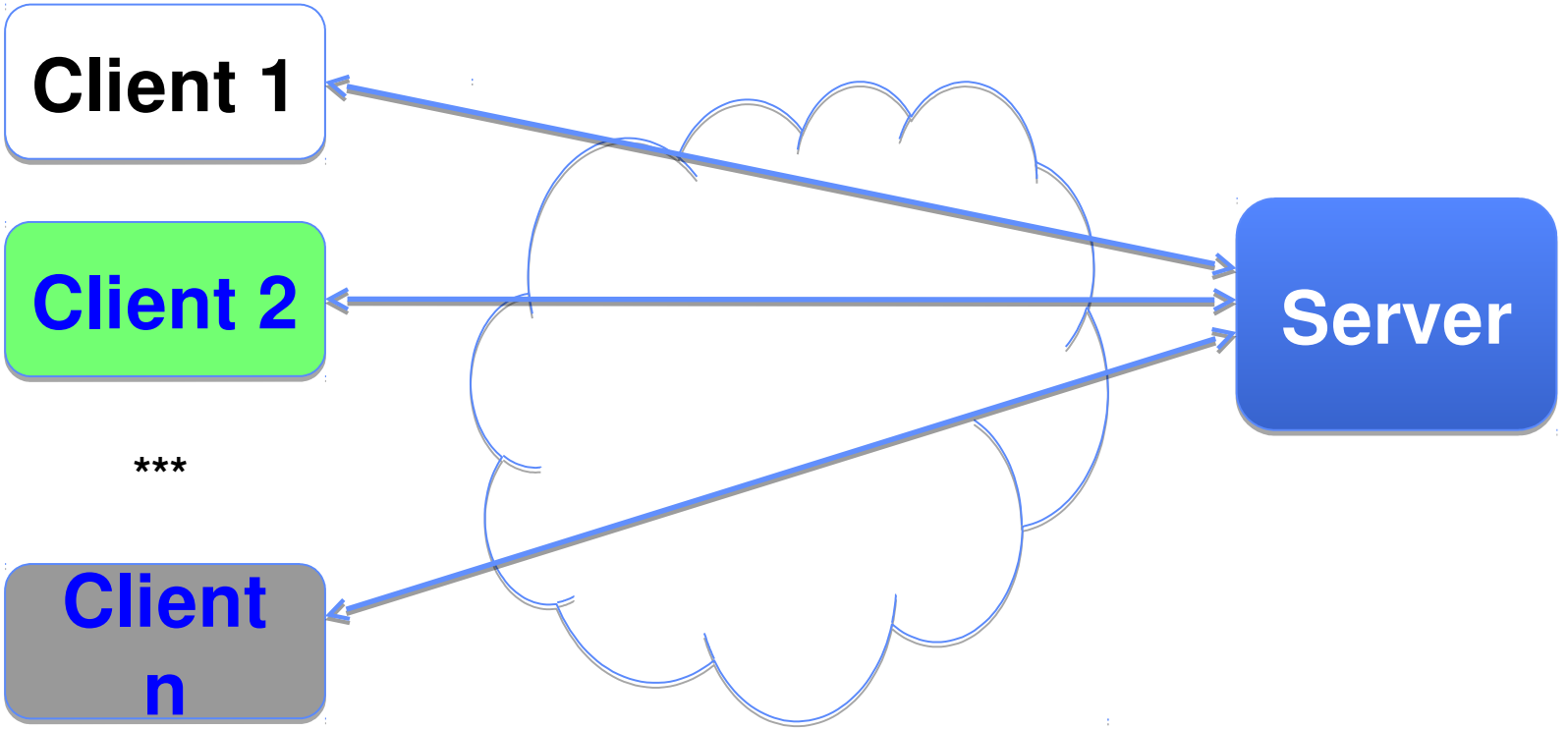
Example: Web browser and website

Client (issues requests) Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```



Client-Server Models



Many clients accessing a common server

All intelligence in the server

Distributed system? Server typically gateway

Sockets (1)

Socket: an abstraction of a network I/O queue

- Mechanism for **inter-process communication**
- Embodies one side of a communication channel
  - Same interface regardless if local or remote
- First introduced in 4.2 BSD UNIX
  - Most OSs (Linux, MacOS X, Windows, ...) provide this
  - ... even if they don't copy the rest of Unix-style low-level IO
  - Standardized by POSIX

Sockets (2)

Looks like a file with a **file descriptor**

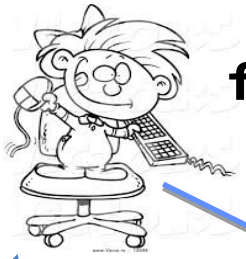
- read()/write() work – add/remove from queue
- Bidirectional: **one queue in each direction**
- Some operations **do not** work: e.g. lseek()

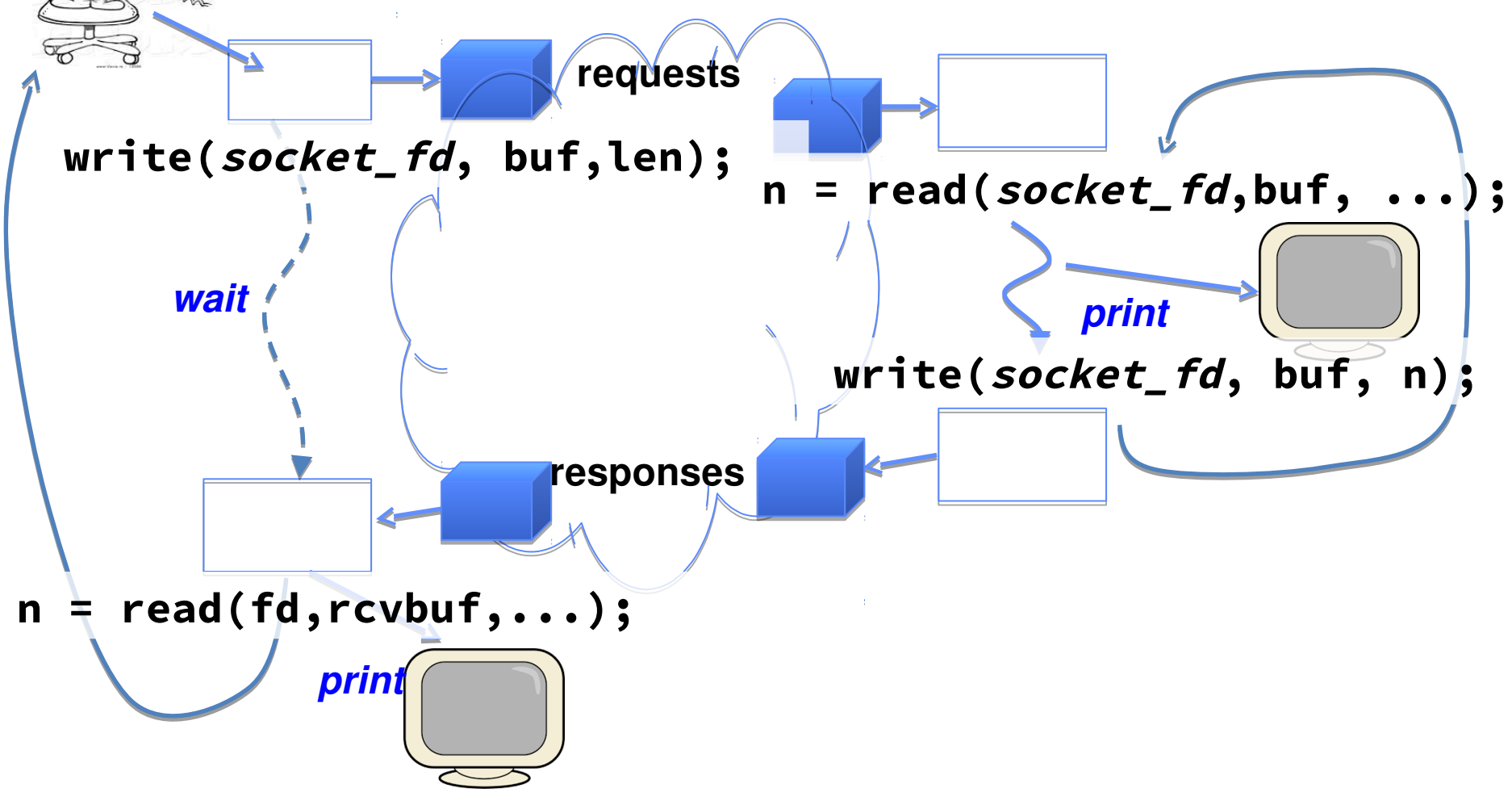
Any kind of network:

- Local
- the Internet (TCP/IP, UDP/IP)
- things "no one" uses anymore (OSI, Appletalk, SNA, IPX, SIP, ...)

Silly Echo Server – running example

Client (issues requests) Server (performs operations)





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 connsockfd) {
  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 */
  }
}
```

Prompt for input

```
char *getreq(char *inbuf, int len) {
  /* Get request char stream */
  printf("REQ: "); /* prompt */

  /* read up to a EOL */
  /* fgets returns NULL on EOF or error */
  return fgets(inbuf,len,stdin);
}
```

Socket creation and connection

File systems: permanent objects

- Exist independent of the processes

Sockets transient, tied to particular processes

Creation and connection is more complex

Form 2-way pipes between processes

- Possibly worlds away

IP namespaces

Hostnames

- www.berkeley.edu (DNS name, string)

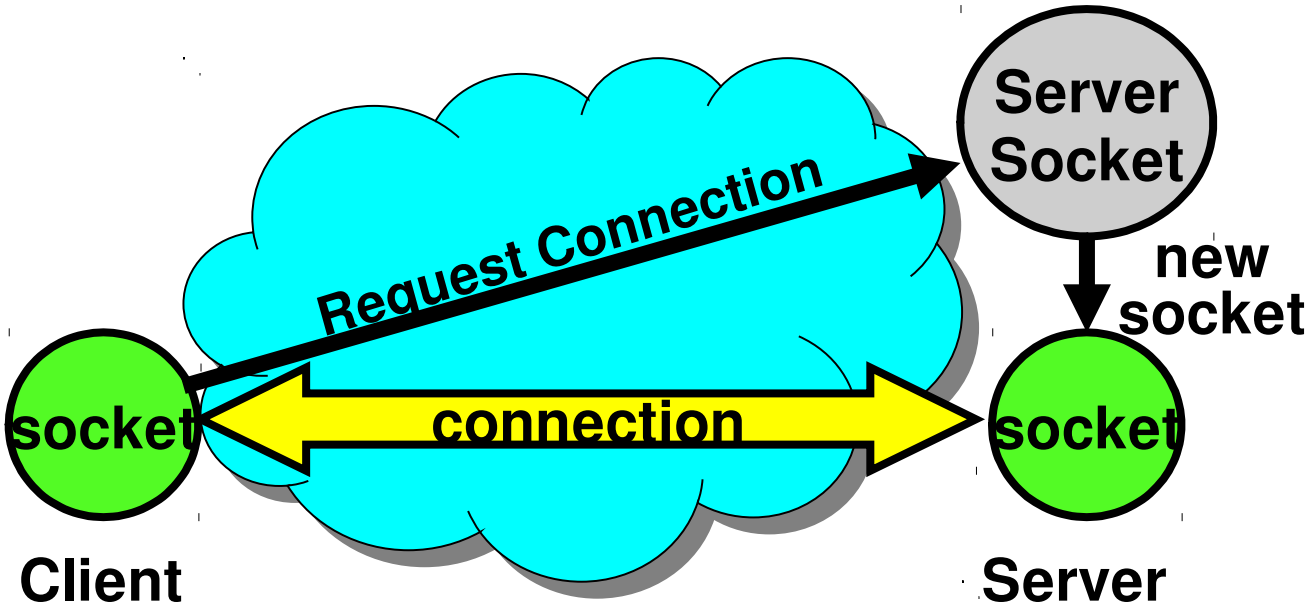
IP address

- 169.229.216.200 (IPv4, 32-bit integer)
- 2607:f140:0:81::f (IPv6, 128-bit integer)

Port Number

- 0-1023 typically reserved for administrator (superuser/sudo)
- 0– 49151 are for "well-known" numbers for specific services
  - 443 for HTTPS
- 49152–65535 (2^15+2^14 to 2^16–1) are "dynamic"/"private"
  - Automatically allocated by OS

Socket Setup over TCP/IP



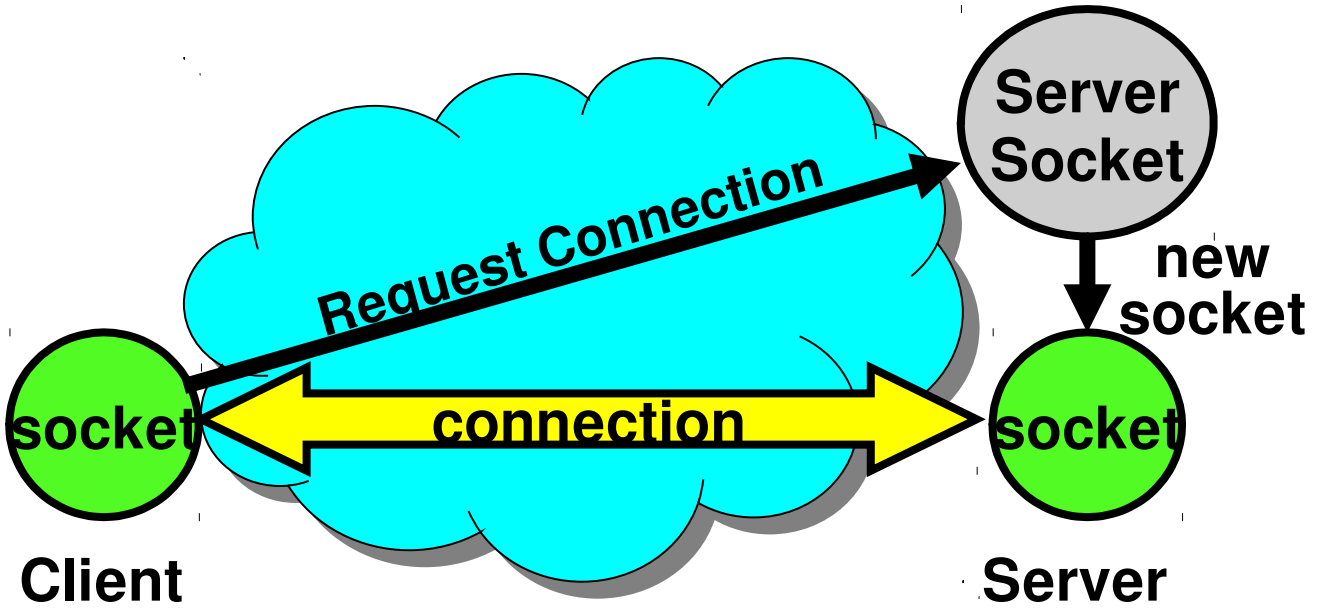
Special kind of socket: ***server socket***

- 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

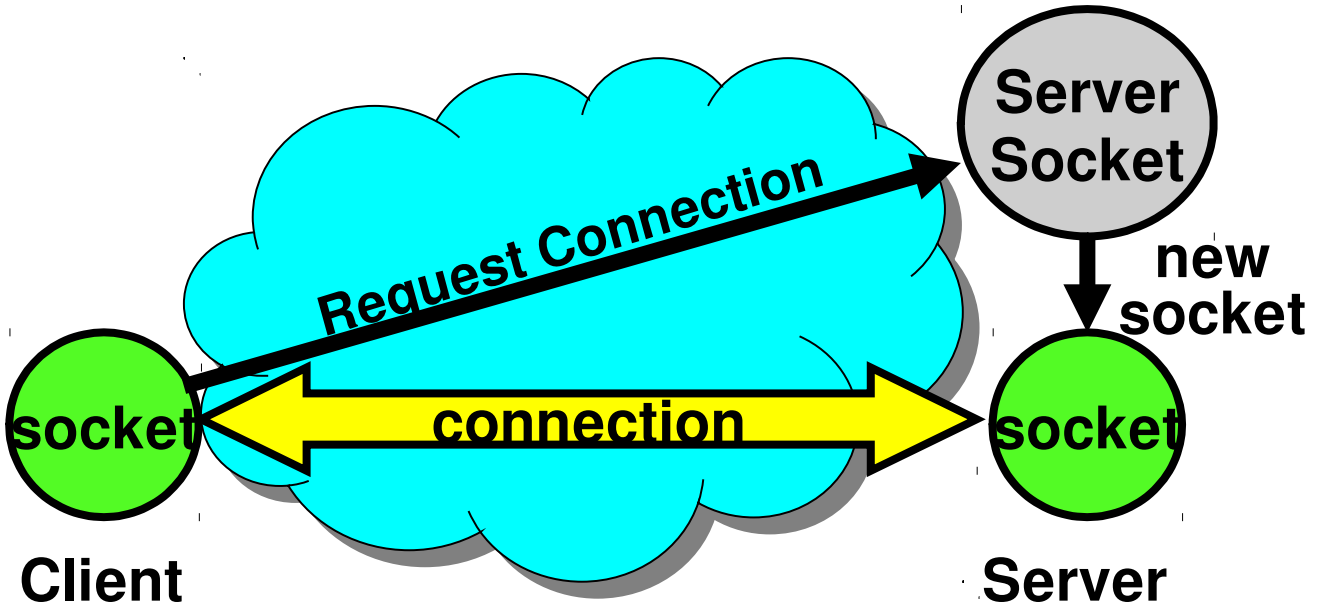
Socket Setup over TCP/IP



5-tuple identifies each connection/request:

- source IP address
- destination IP address
- source port number
- destination port number
- protocol (always TCP here)

Socket Setup over TCP/IP



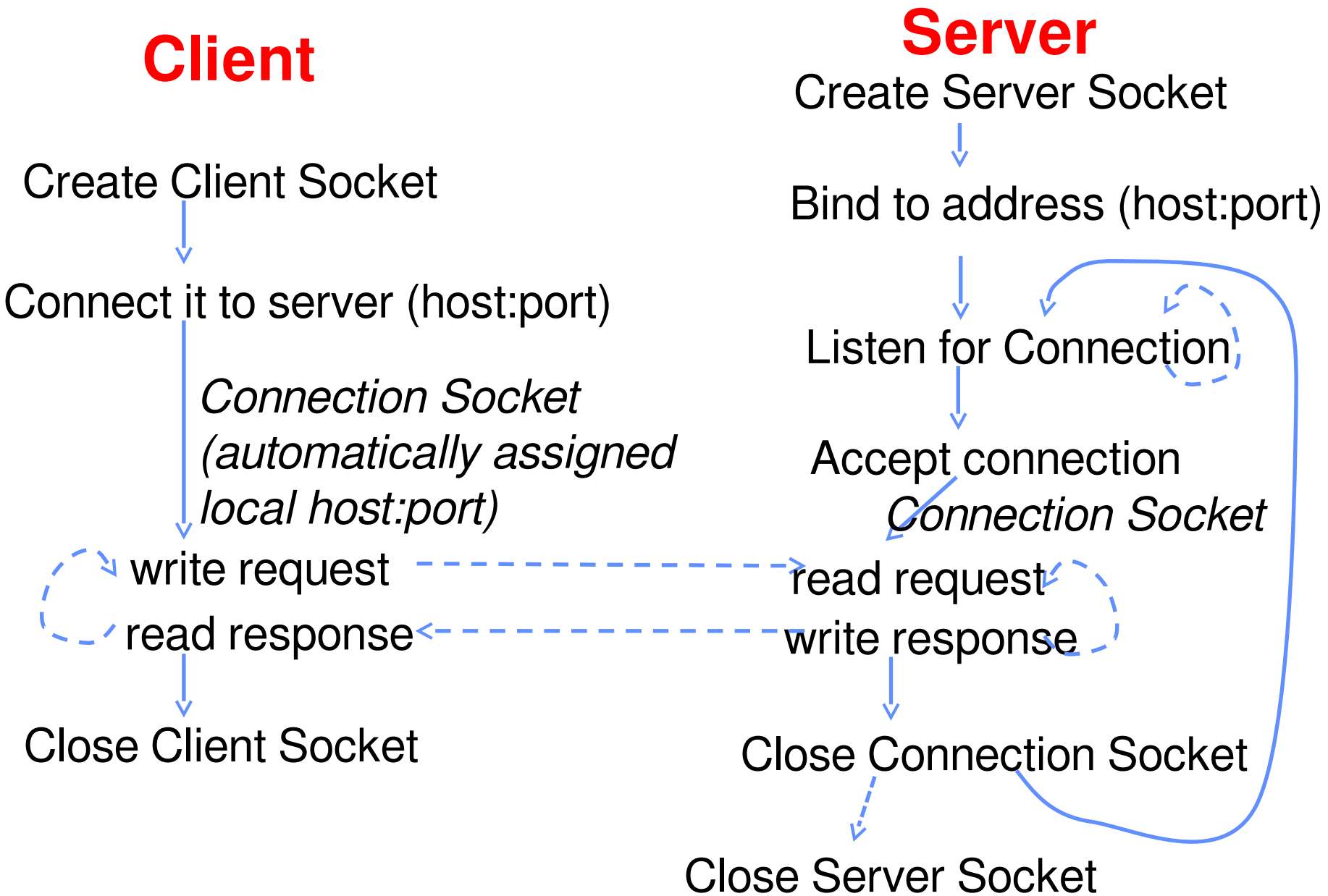
5-tuple identifies each connection/request:

- source IP address
- destination IP address
- ***source port number***
- destination port number
- protocol (always TCP here)

Where does client get its port number from?

Recall: dynamic/private port range

Sockets in concept



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);
```

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

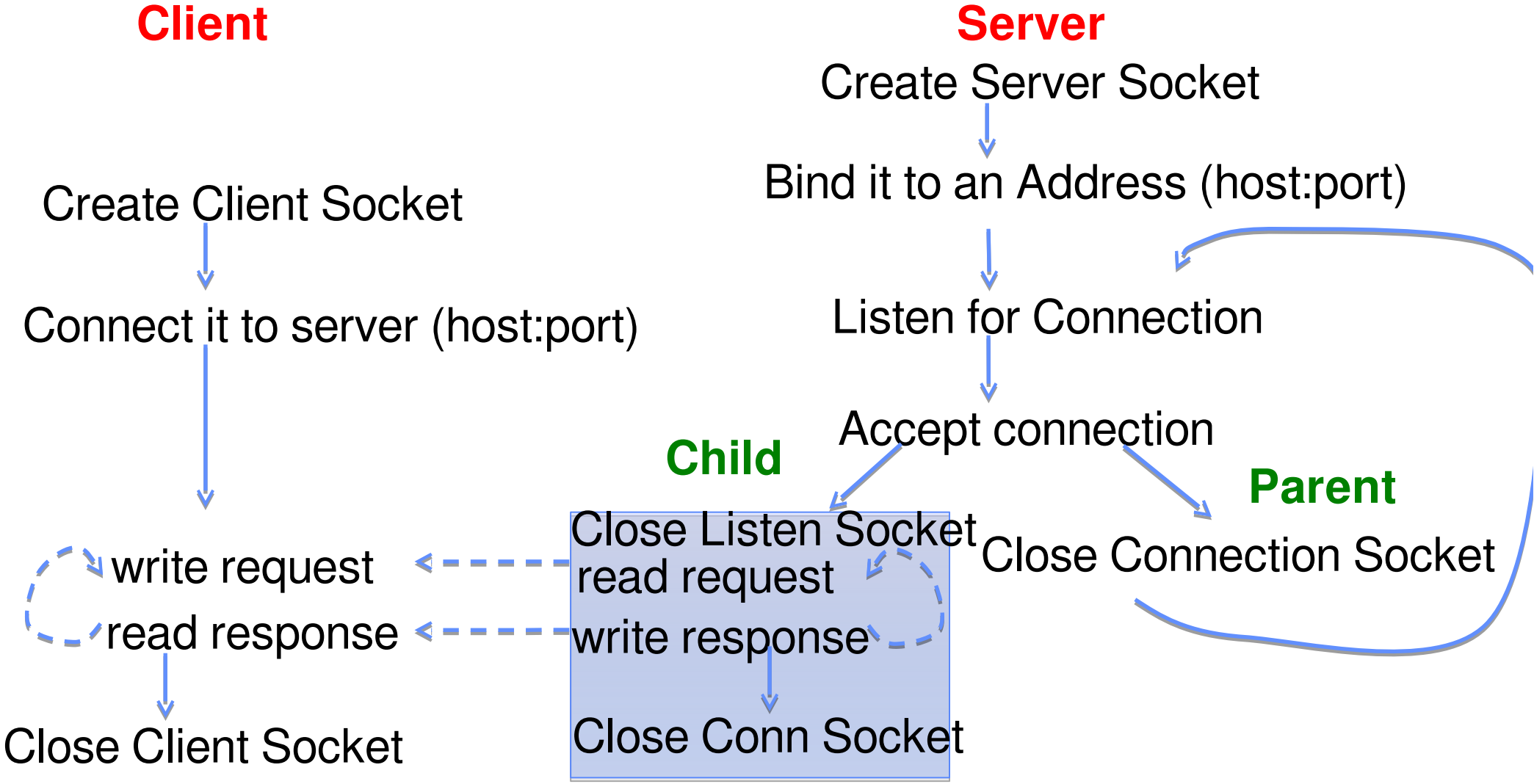
One option – fork a process for each connection

- 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***

Summary: (Distributed) Consistency

Everyone agrees on the state of the system:

- Won't depend on who you ask
- Won't depend on if nodes go down

Transcation idea:

- **Atomic** change of state everywhere
- Once it happens, never taken back

CAP theorem: perfect consistency and perfect availability impossible

- Two-phase commit: **stall** system instead of letting disconnected node get out of sync

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()

Under the hood

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

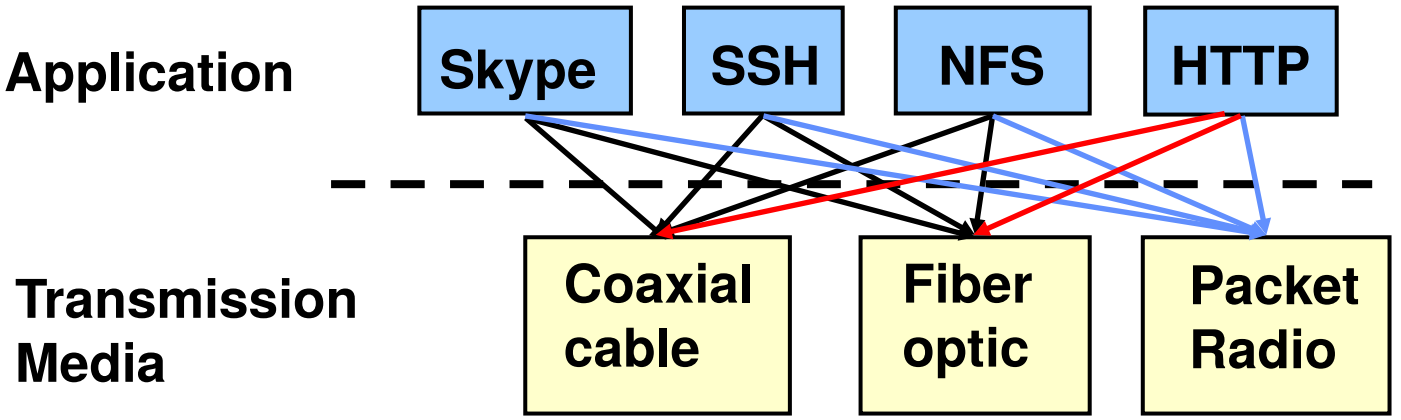
- email, web, P2P, etc.

Many different network styles and technologies

- Wireless vs. wired vs. optical, etc.

How do we organize this mess?

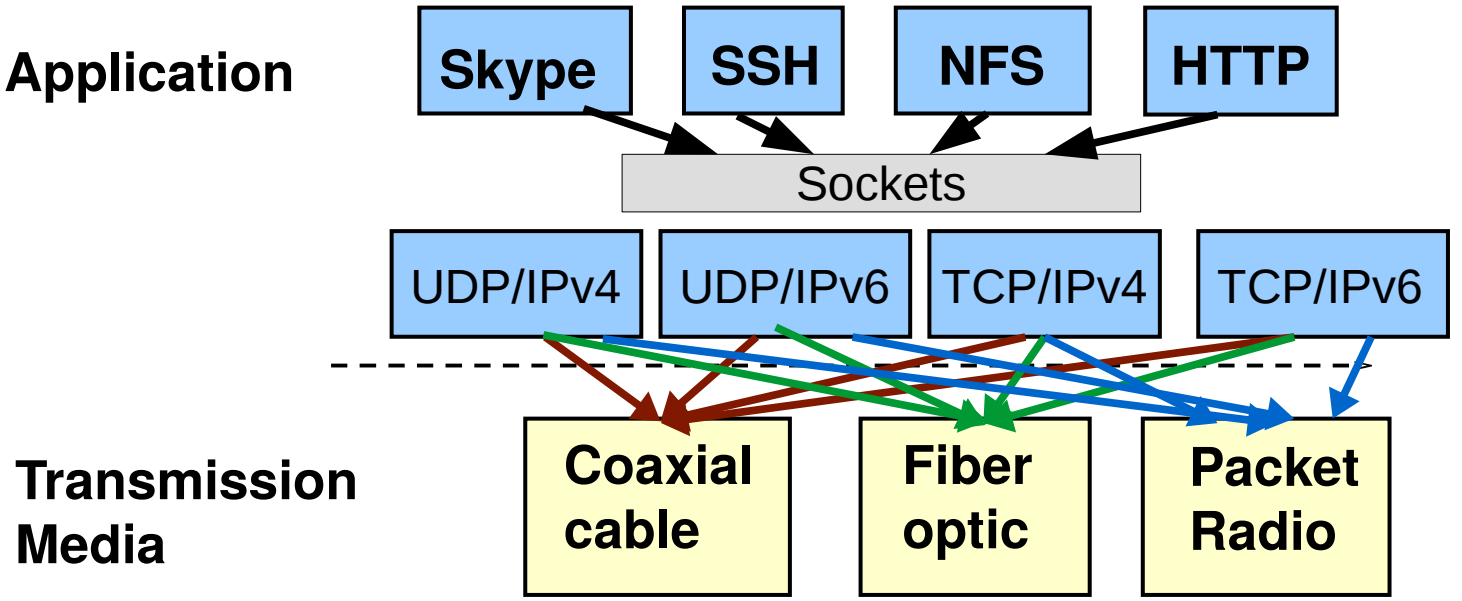
The Problem (2)



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

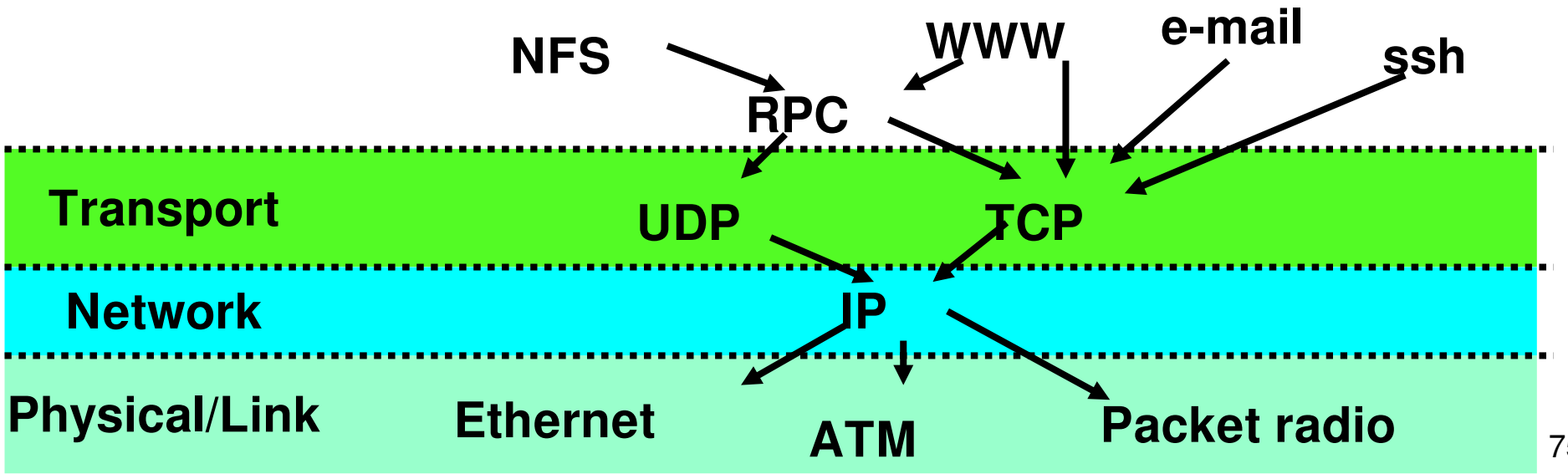
TCP/IP networking layers:

- 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

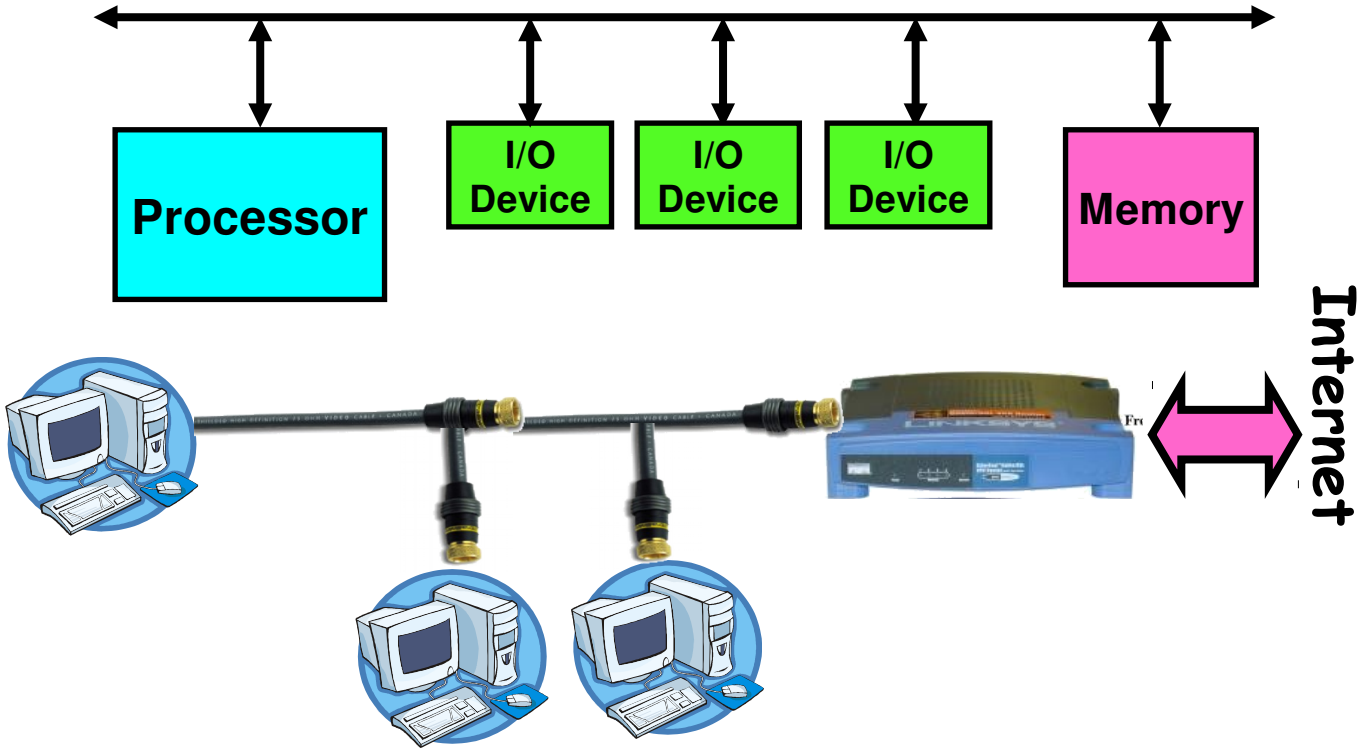
Complex services from simpler ones

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

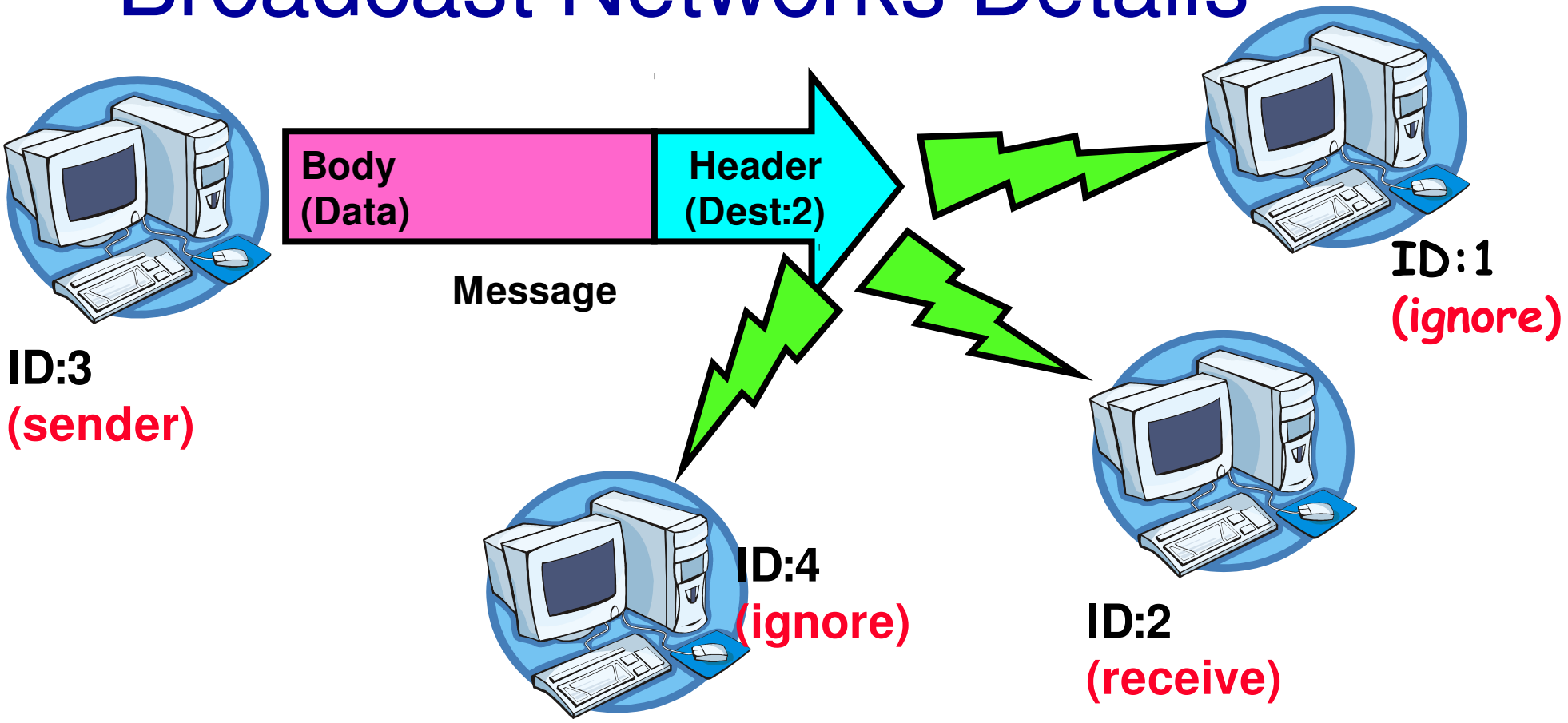
Broadcast Networks

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?

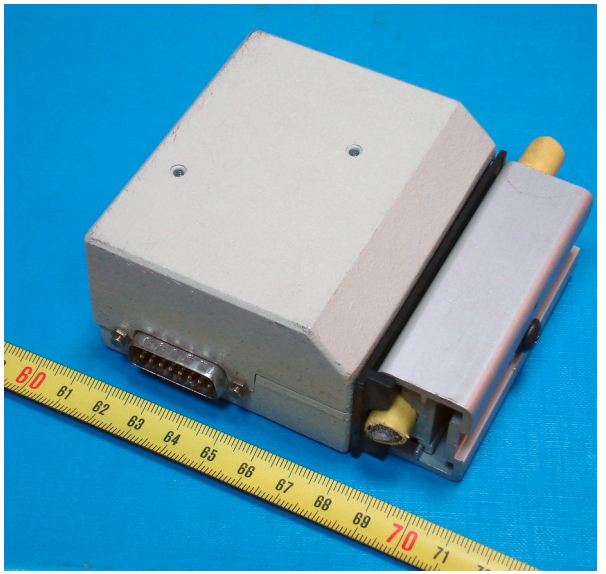
First example: Aloha network (70's): packet radio within Hawaii

- 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)

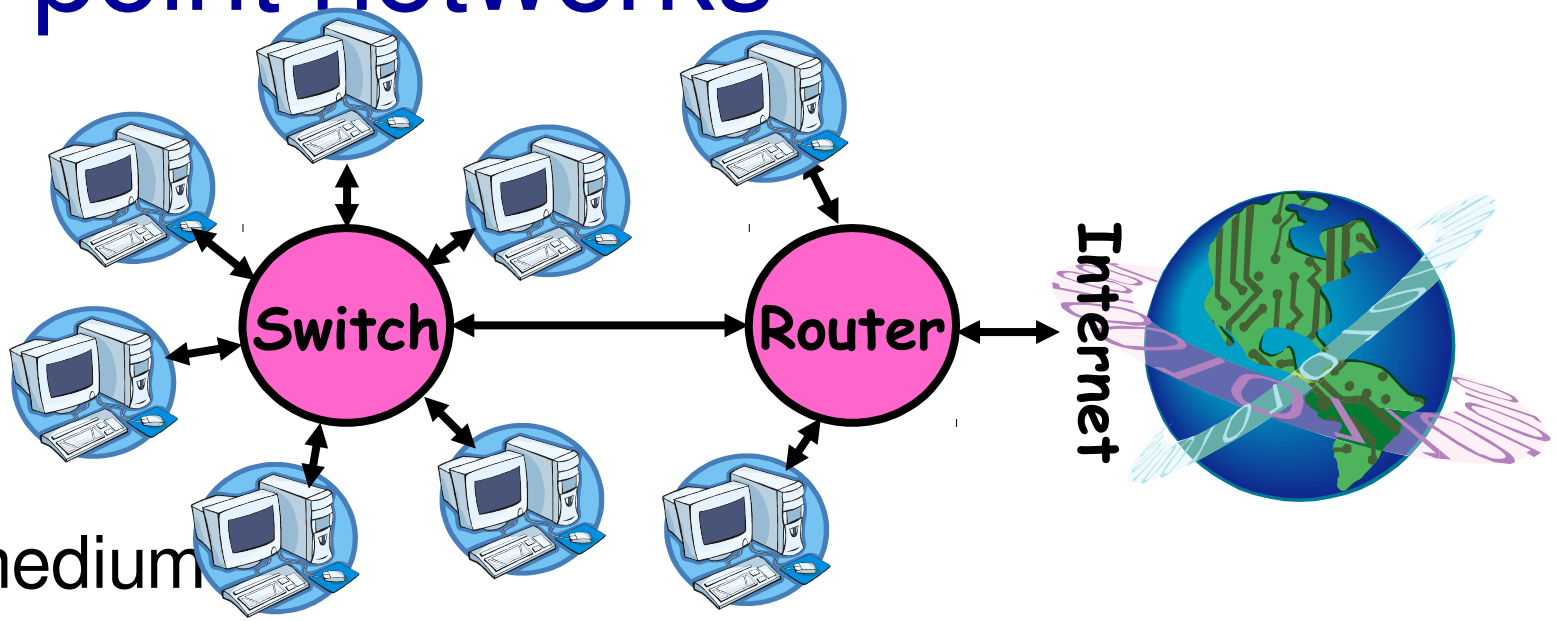
Collision Detect: sender checks if packet trampled.

- 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

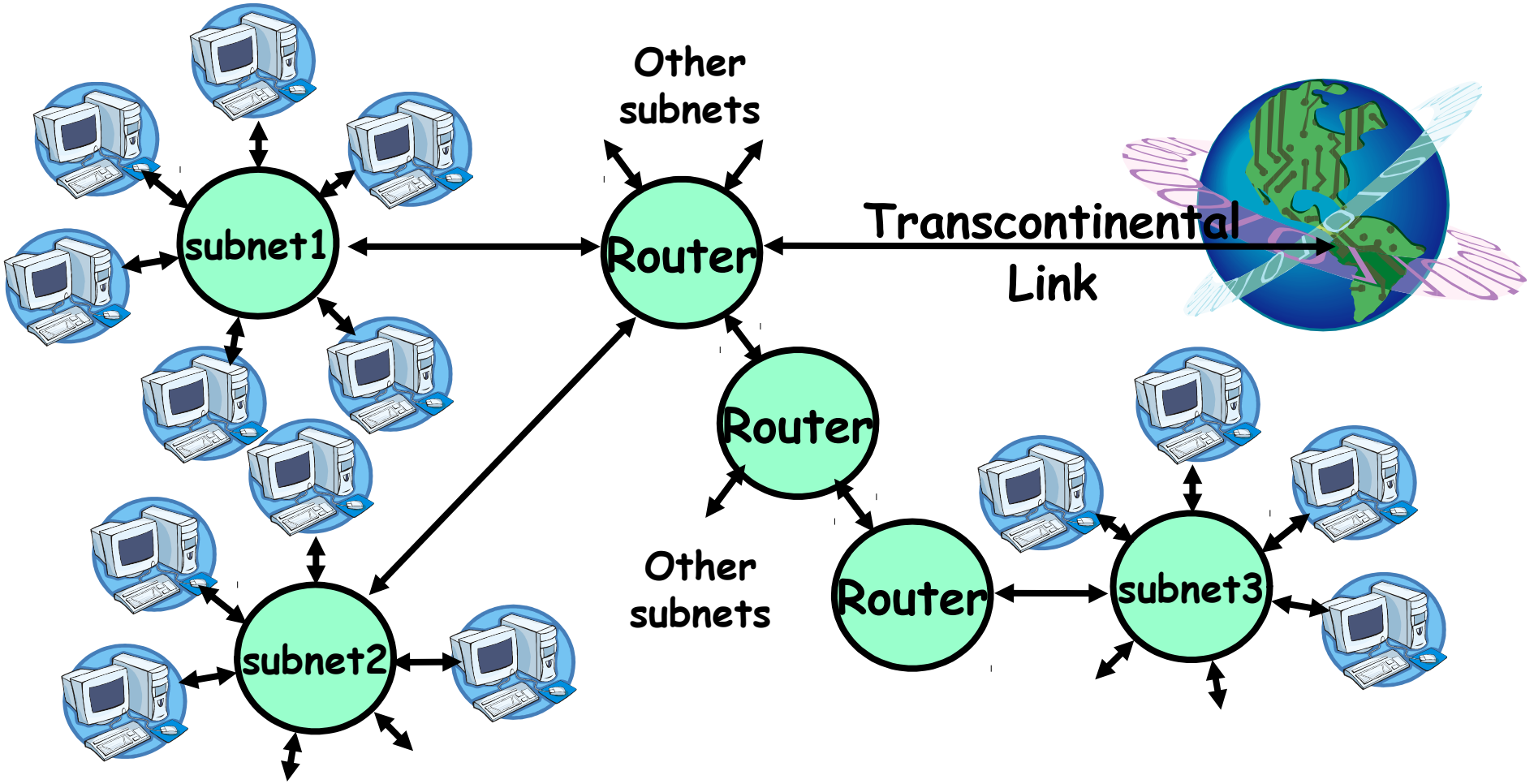
- One or more IP address – some private
- Network Address Translation – Not every computer has a unique IP

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

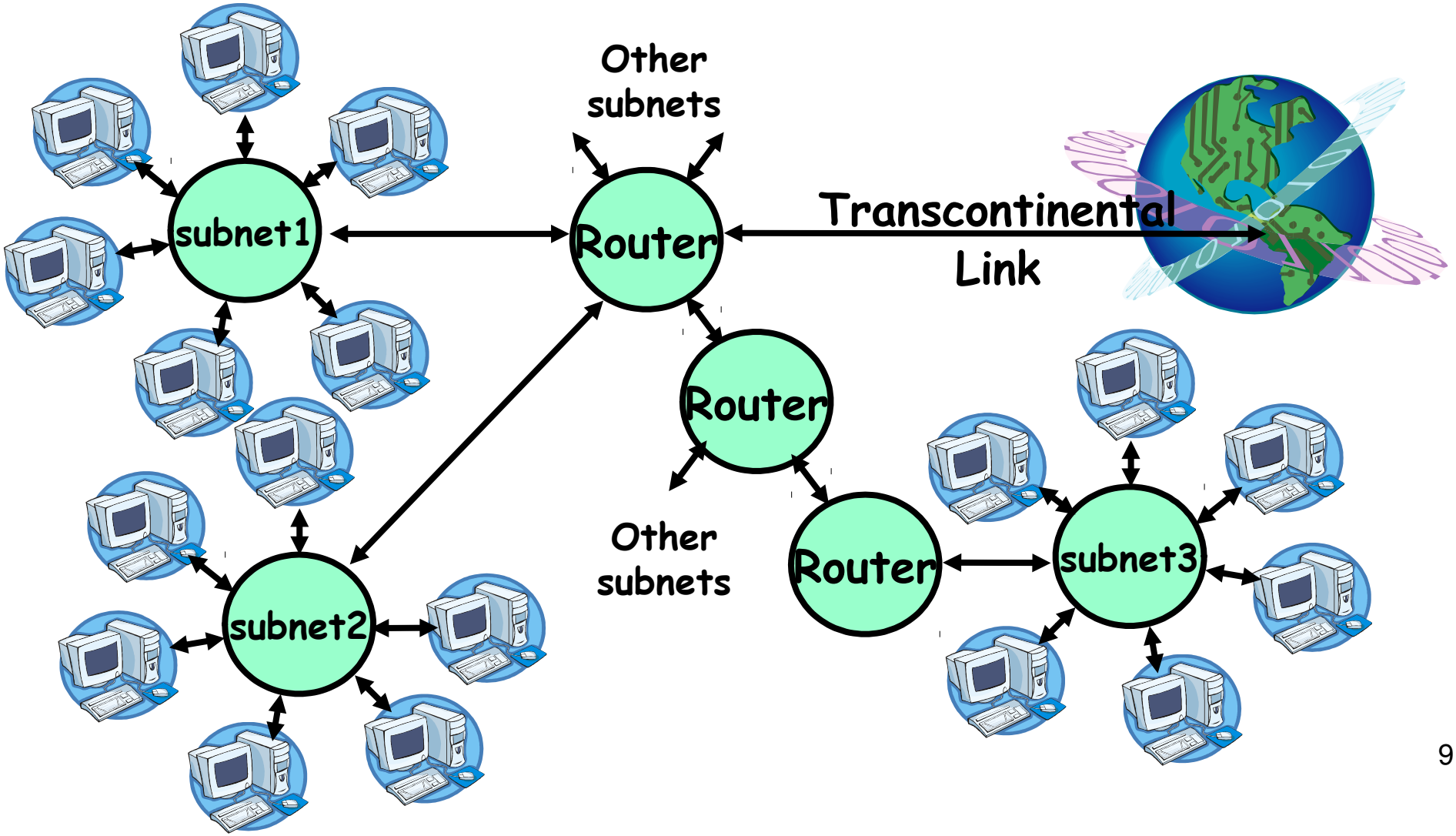
- localhost: 127.0.0.01

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



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

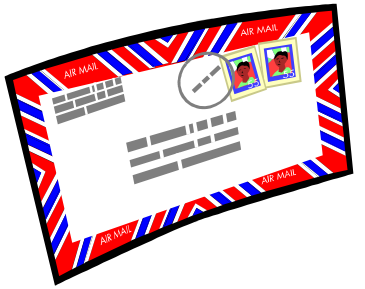
- explain how to access a resource
- change if the resource moves

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***

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*

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()

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