Recap – Home Stretch

David E. Culler

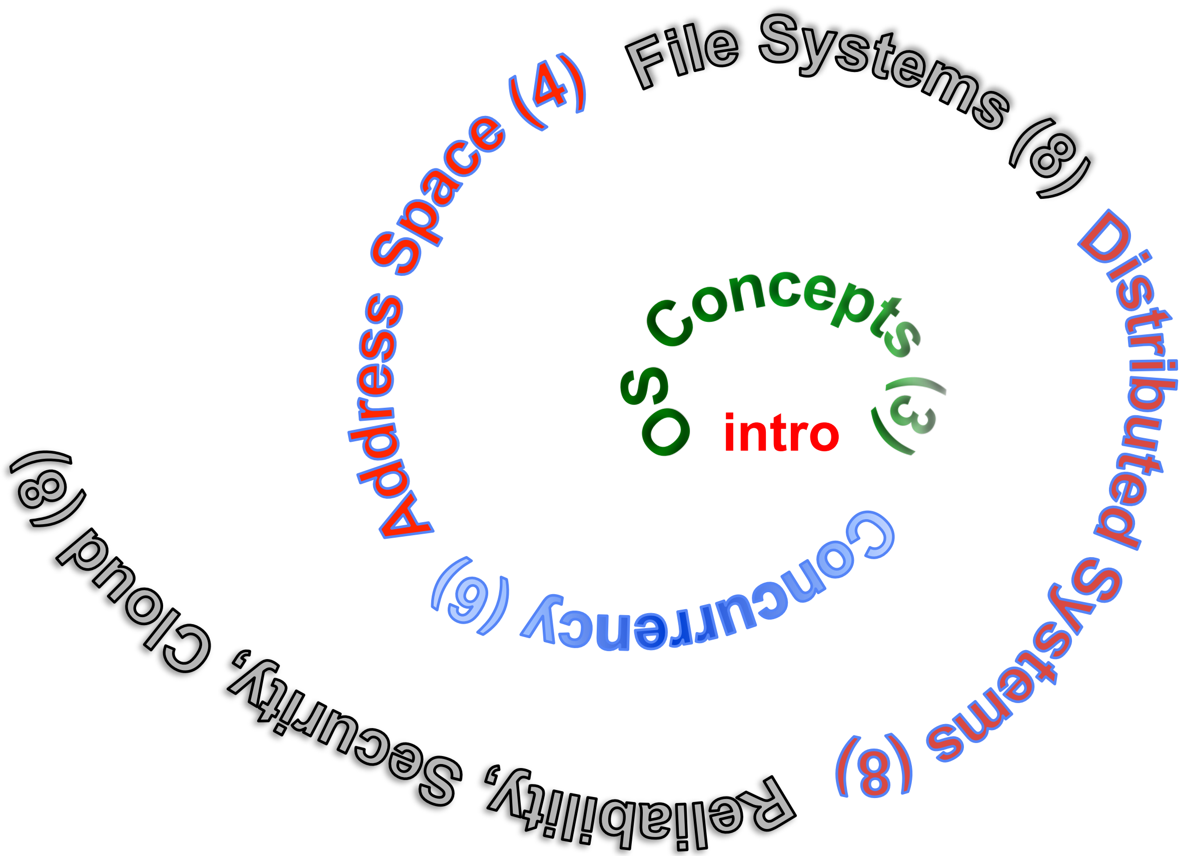
CS162 – Operating Systems and Systems Programming

http://cs162.eecs.berkeley.edu/

Lecture 43

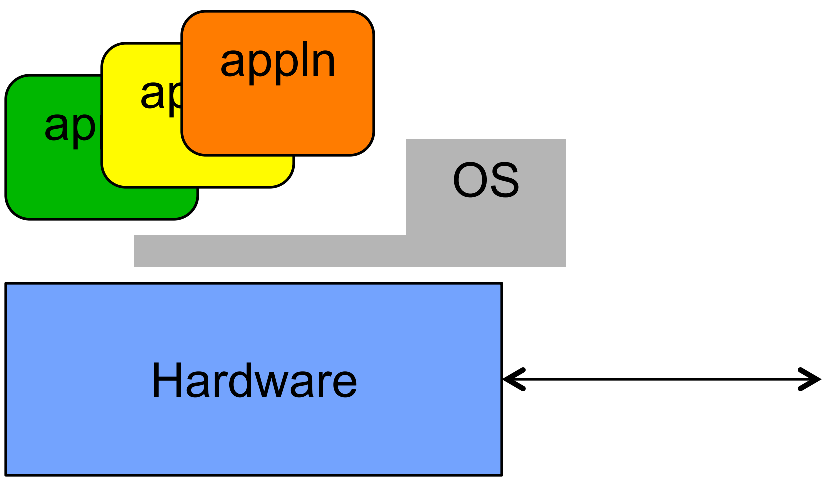
December 10, 2014

Course Structure: Spiral



What is an operating system?

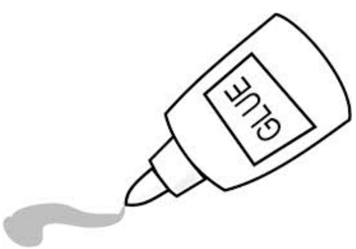
- **Special layer of software that provides application software access to hardware resources**
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities



What is an Operating System?

- **Referee**
  - Manage sharing of resources, Protection, Isolation
    - » Resource allocation, isolation, communication

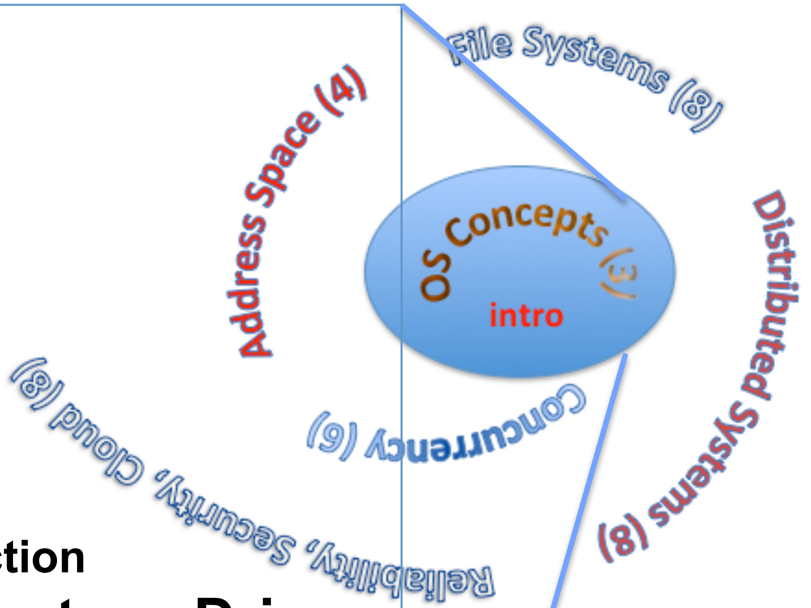
- **Illusionist**
  - Provide clean, easy to use abstractions of physical resources
    - » Infinite memory, dedicated machine
    - » Higher level objects: files, users, messages
    - » Masking limitations, virtualization



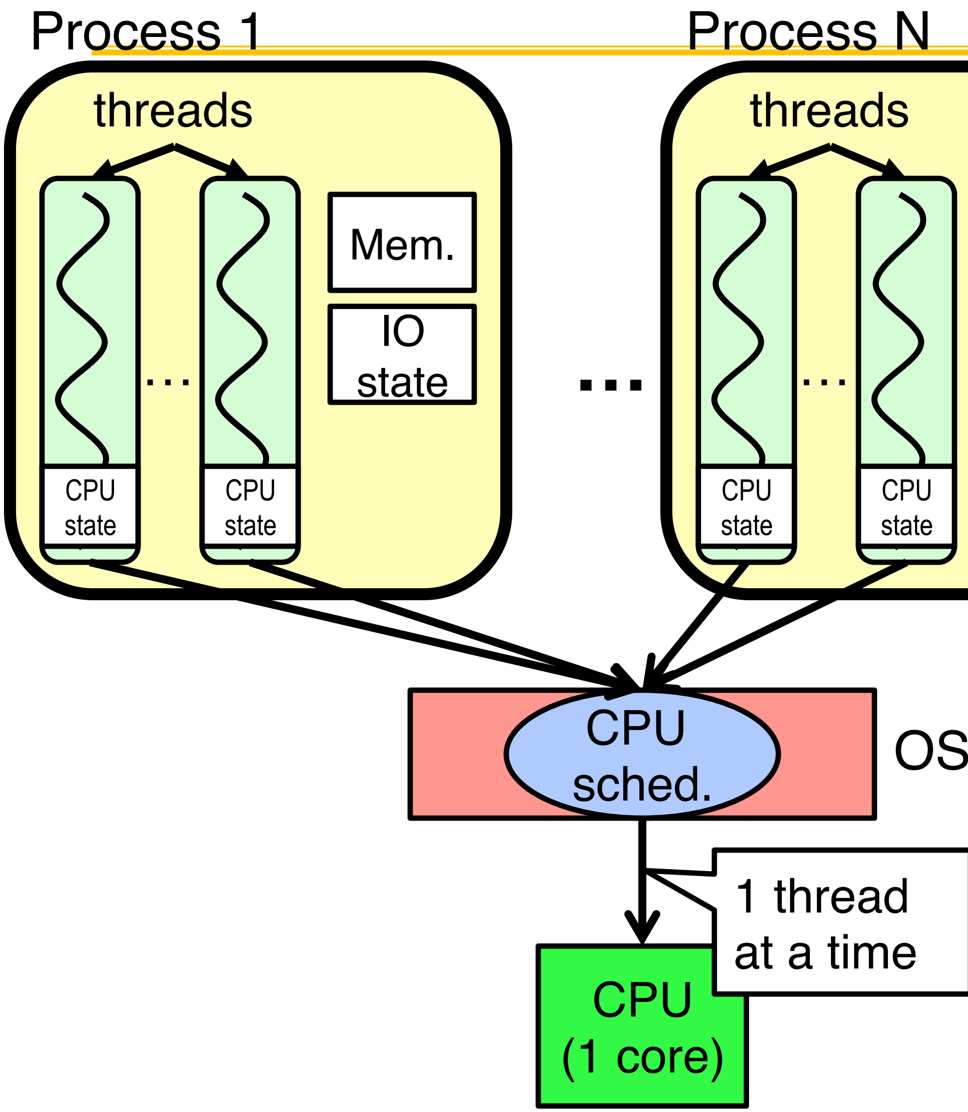
- **Glue**
  - Common services
    - » Storage, Window system, Networking
    - » Sharing, Authorization
    - » Look and feel

Core Concepts

- **Processes**
  - Thread(s) + address space
- **Address Space**
- **Protection**
- **Dual Mode**
- **Interrupt handlers**
  - Interrupts, exceptions, syscall
- **File System**
  - Integrates processes, users, cwd, protection
- **Key Layers: OS Lib, Syscall, Subsystem, Driver**
  - User handler on OS descriptors
- **Process control**
  - fork, wait, signal, exec
- **Communication through sockets**
  - Integrates processes, protection, file ops, concurrency
- **Client-Server Protocol**
- **Concurrent Execution: Threads**
- **Scheduling**

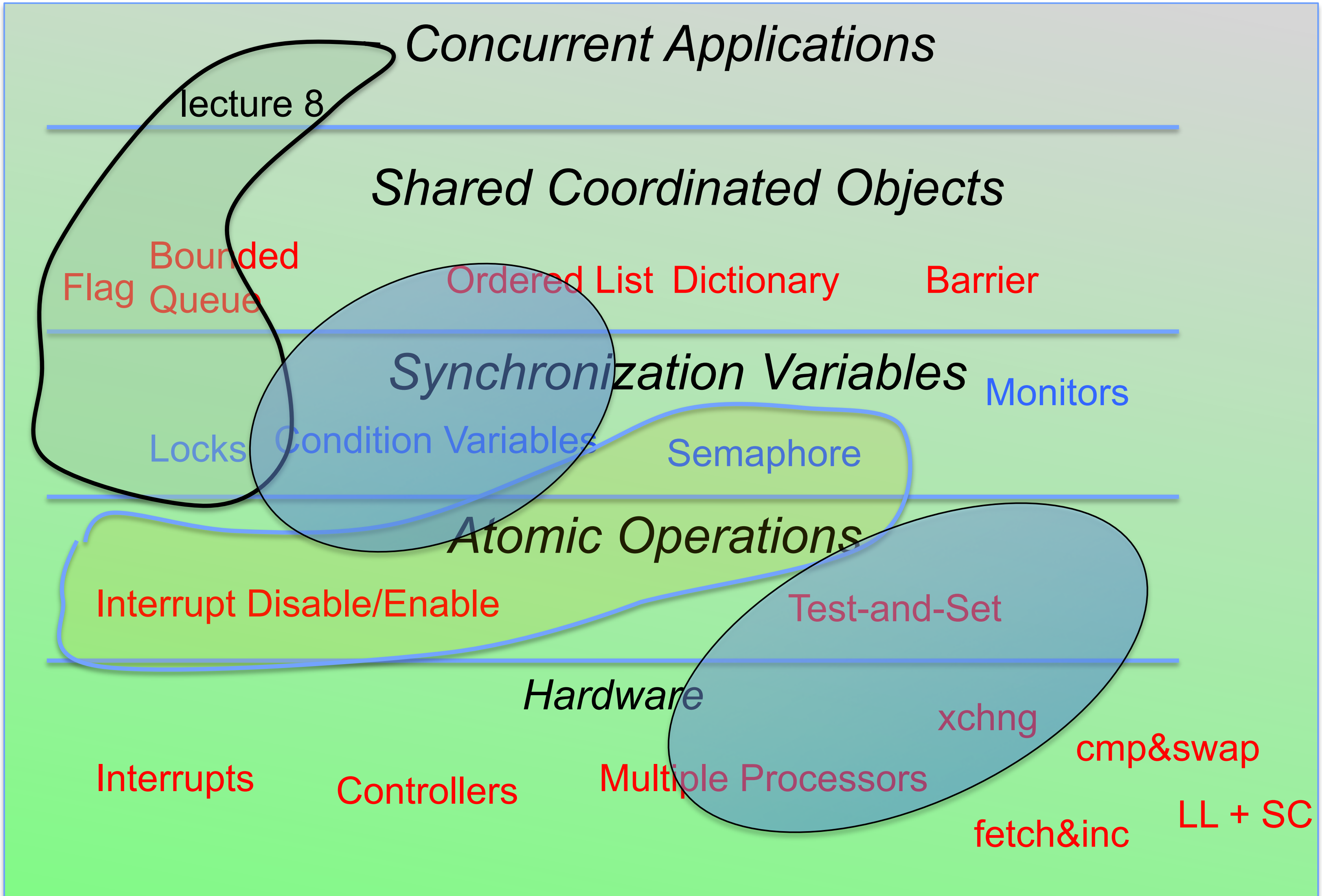


Threads



- **Independently schedulable entity**
- **Sequential thread of execution that runs concurrently with other threads**
  - **It can block waiting for something while others progress**
  - **It can work in parallel with others (ala cs61c)**
- **Has local state (its stack) and shared (static data and heap)**

Concurrency Coordination Landscape



Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We'll show that is hard to build anything useful with only reads and writes
- **Critical Section**: piece of code that only one thread can execute at once
- **Mutual Exclusion**: ensuring that only one thread executes critical section
  - One thread *excludes* the other while doing its task
  - Critical section and mutual exclusion are two ways of describing the same thing

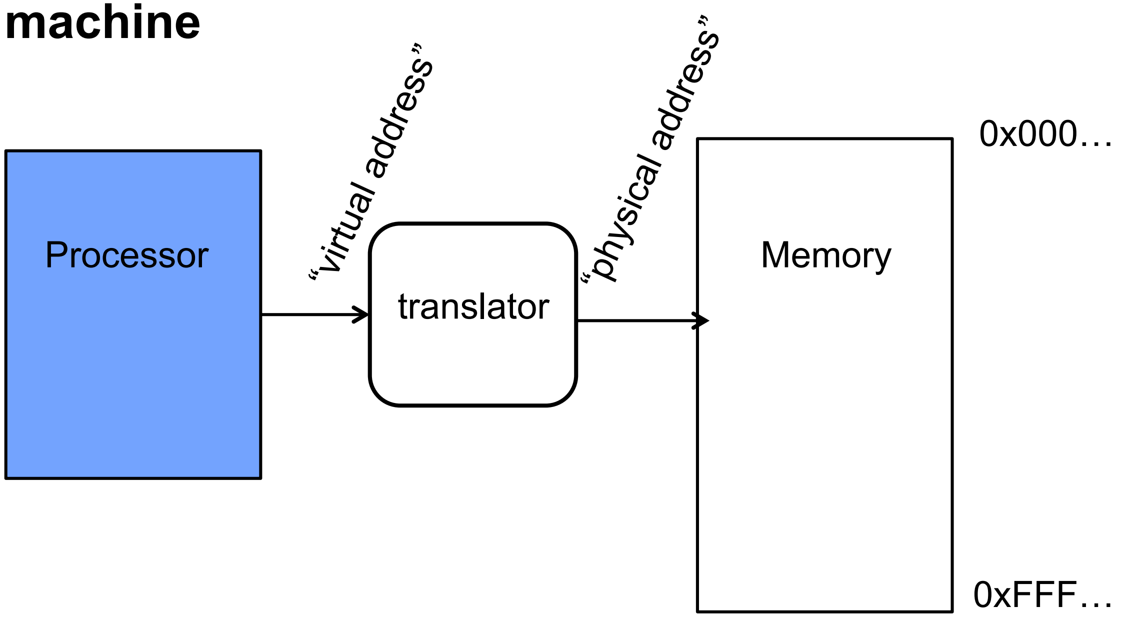
Scheduling Summary

- Scheduling: selecting a process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)
- **Shortest Remaining Time First (SRTF):**
  - **Run whatever job has the least remaining amount of computation to do**
  - **Pros: Optimal (average response time)**
  - **Cons: Hard to predict future, Unfair**

Address Translation

Key OS Concept: Address Space

- **Program operates in an address space that is distinct from the physical memory space of the machine**

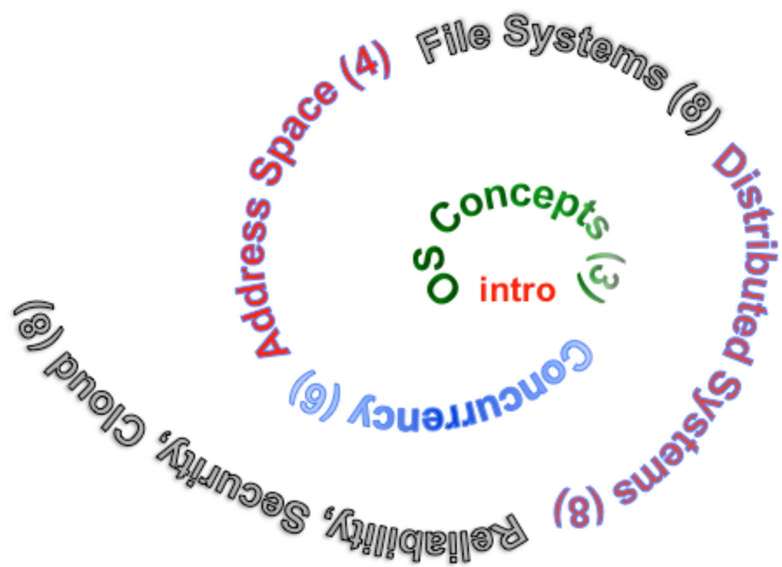


Summary of Translation

- **Memory is a resource that must be multiplexed**
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- **Simple Protection through segmentation**
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well
- **Page Tables**
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address
- **Multi-Level Tables**
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- **Inverted page table: size of page table related to physical memory size**

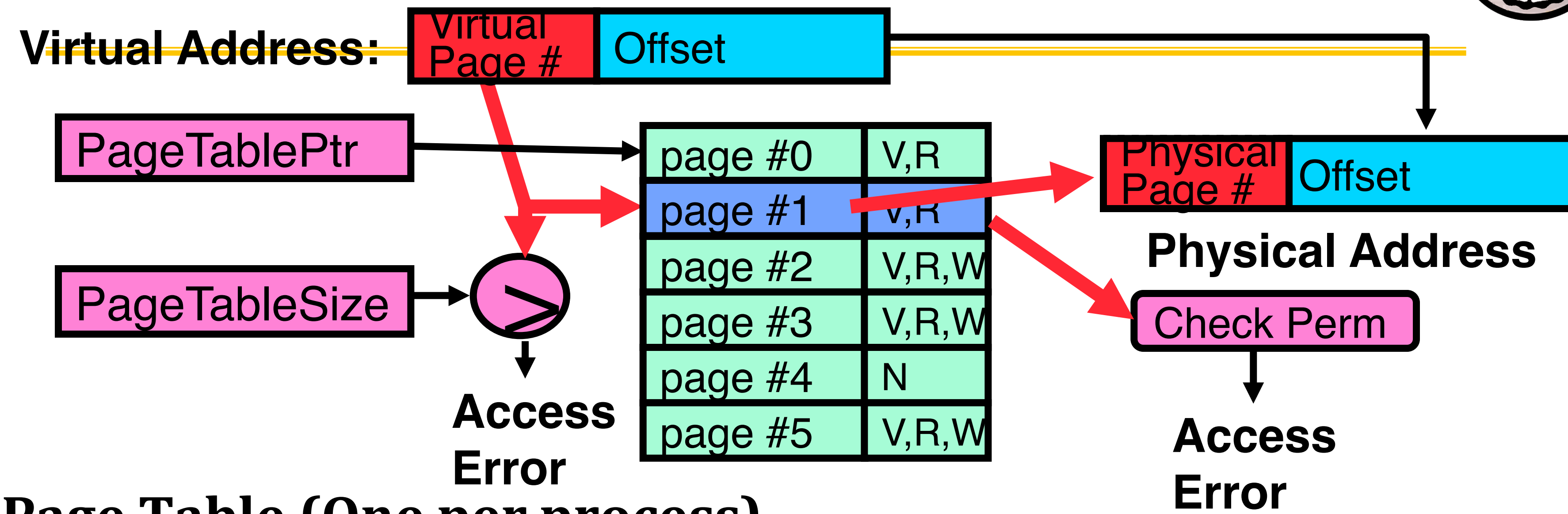
Objective

- **Dive deeper into the concepts and mechanisms of address translation**
- **Enabler of many key aspects of operating systems**
  - Protection
  - Multi-programming
  - Isolation
  - Memory resource management
  - I/O efficiency
  - Sharing
  - Inter-process communication
  - Debugging
  - Demand paging



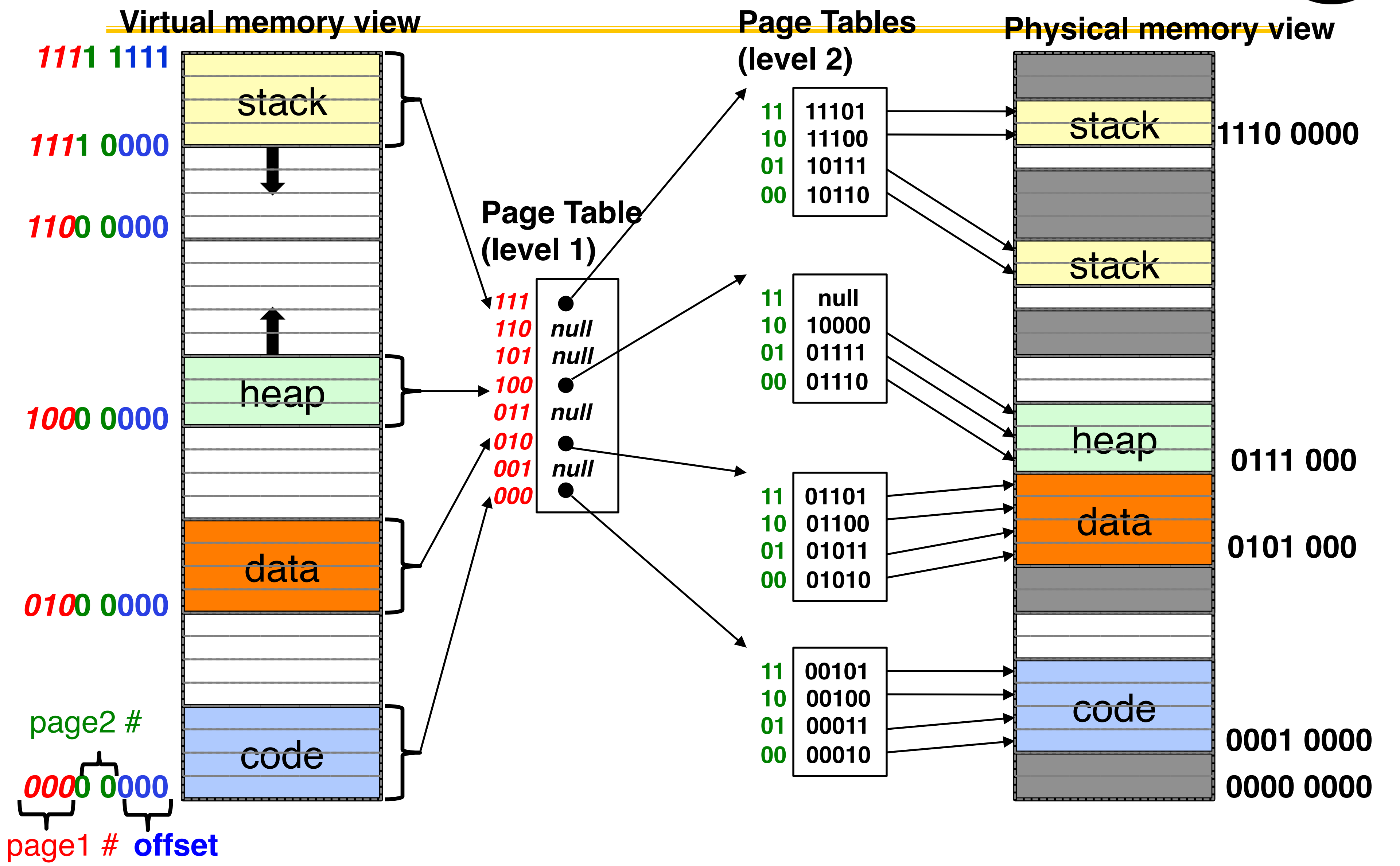
- **Today: Linking, Segmentation, Paged Virtual Address**

How to Implement Paging?

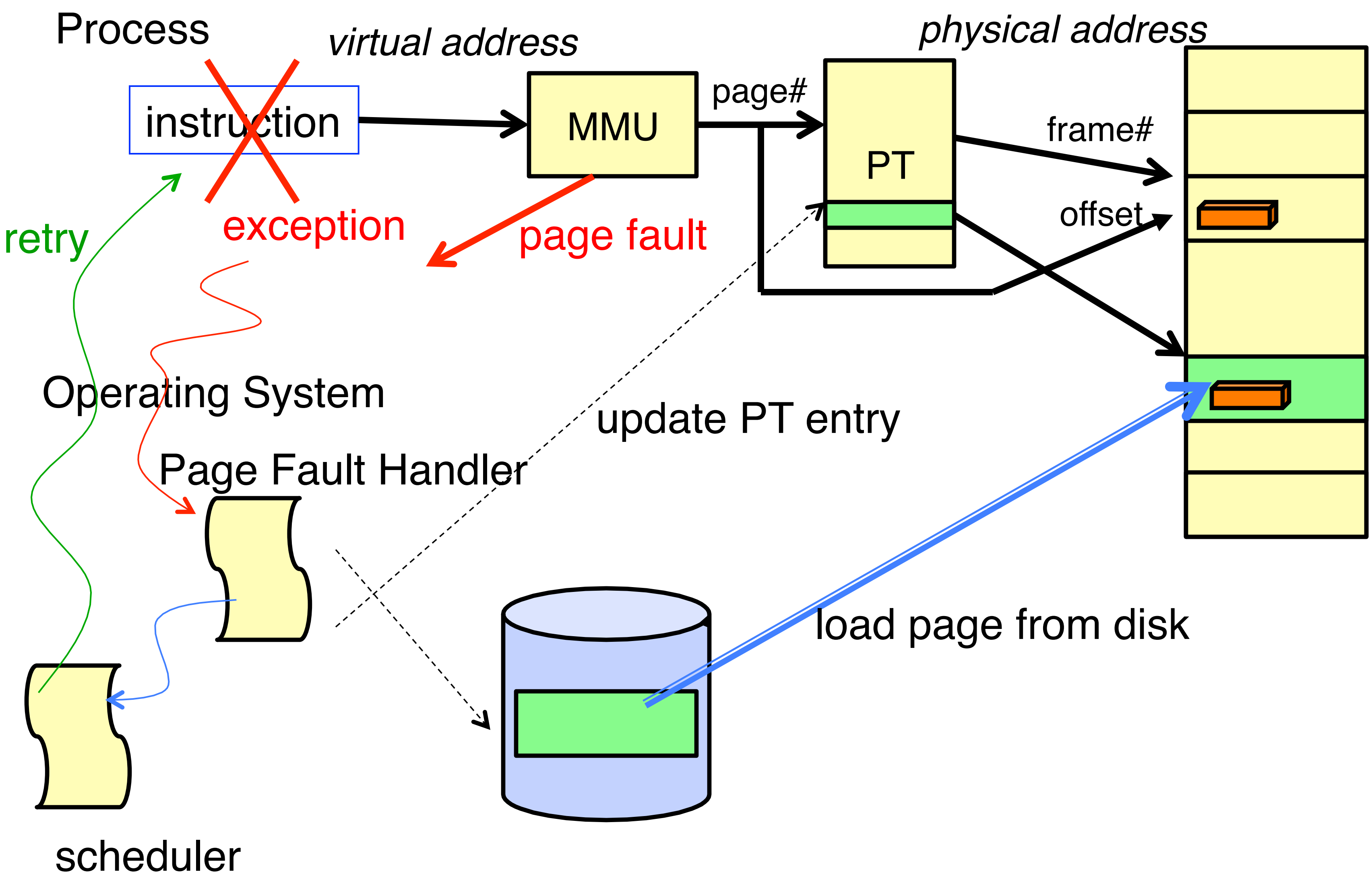


- **Page Table (One per process)**
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- **Virtual address mapping**
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

Summary: Two-Level Paging



What happens when …



I/O & Storage Layers – Today

Application / Service

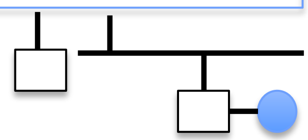
High Level I/O

Low Level I/O

Syscall

File System

I/O Driver



Operations and Interface

streams

fopen, fread, fgets, …, fwrite, fclose on FILE *

handles

open, read, write, close on int & char *

registers

EAX, EBX, … ESP

descriptors

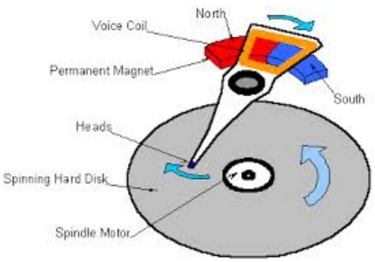
Commands and Data Transfers

ld, st PIO ctrl regs, dm

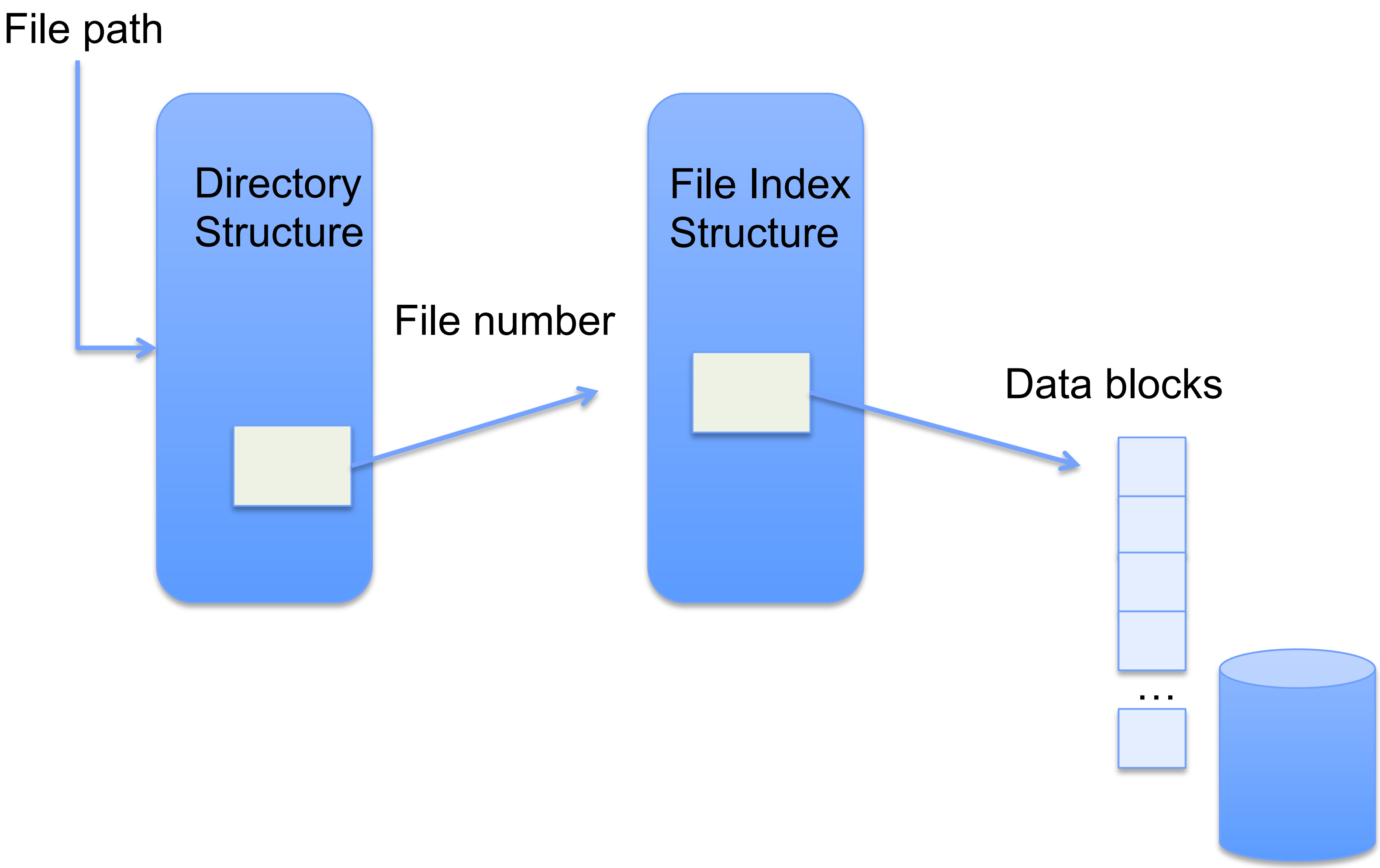
Disks, Flash, Controllers, DMA



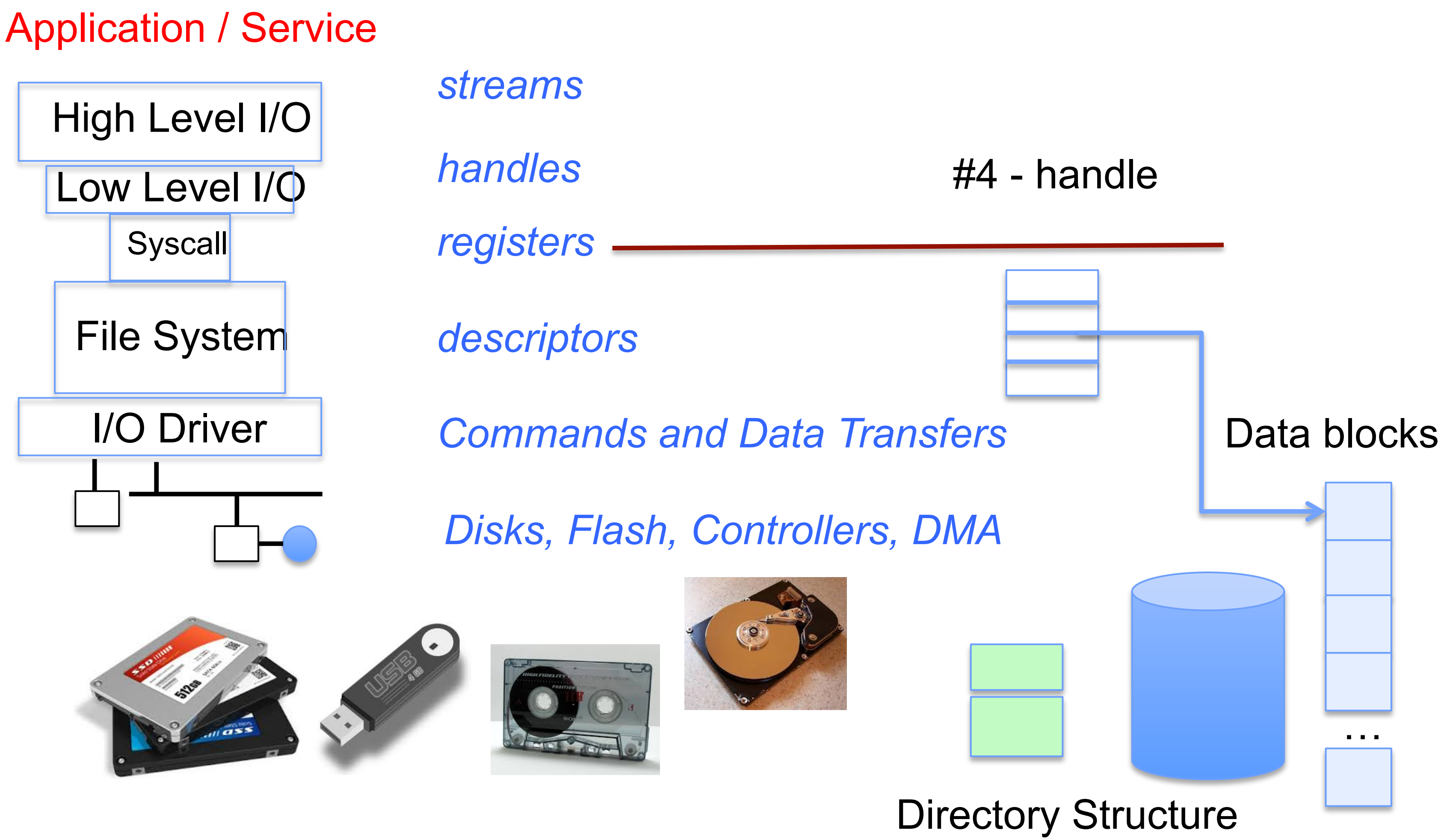




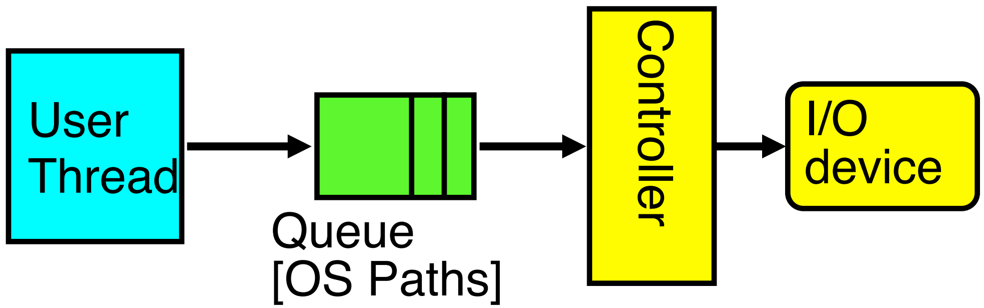
Recall: Components of a File System



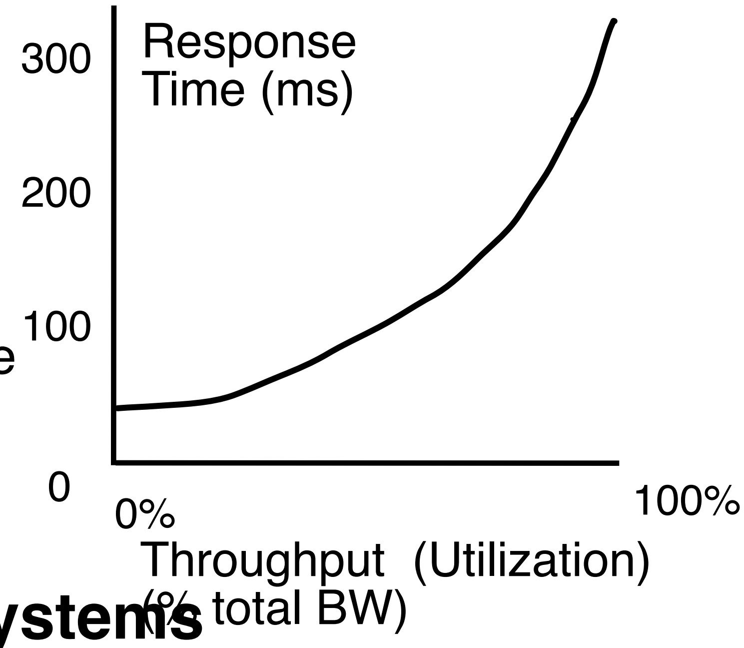
I/O & Storage Layers



I/O Performance

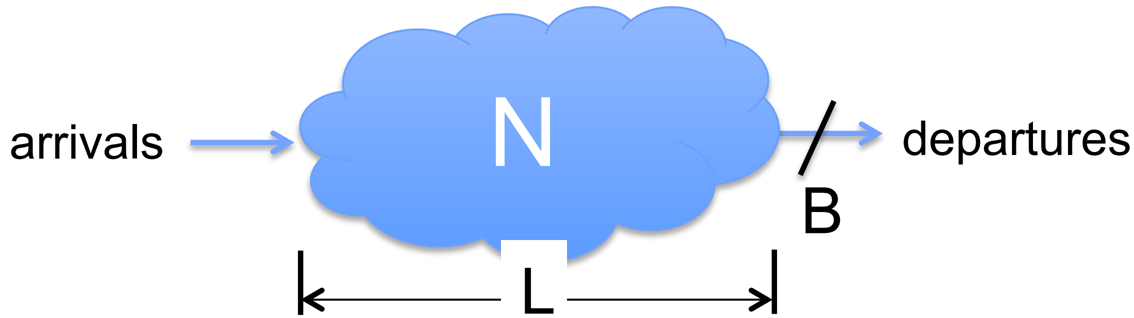


Response Time = Queue + I/O device service time



- **Solutions?**
  - **Make everything faster ☺**
  - **More Decoupled (Parallelism) systems**
    - » **multiple independent buses or controllers**
  - **Optimize the bottleneck to increase service rate**
    - » **Use the queue to optimize the service**
  - **Do other useful work while waiting**
- **Queues absorb bursts and smooth the flow**
- **Admissions control (finite queues)**
  - **Limits delays, but may introduce unfairness and livelock**

Little's Law



- In any *stable* system
  - Average arrival rate = Average departure rate
- the average number of tasks in the system (N) is equal to the throughput (B) times the response time (L)
- N (ops) = B (ops/s) x L (s)
- Regardless of structure, bursts of requests, variation in service
  - instantaneous variations, but it washes out in the average
  - Overall requests match departures

File System Summary (1/2)

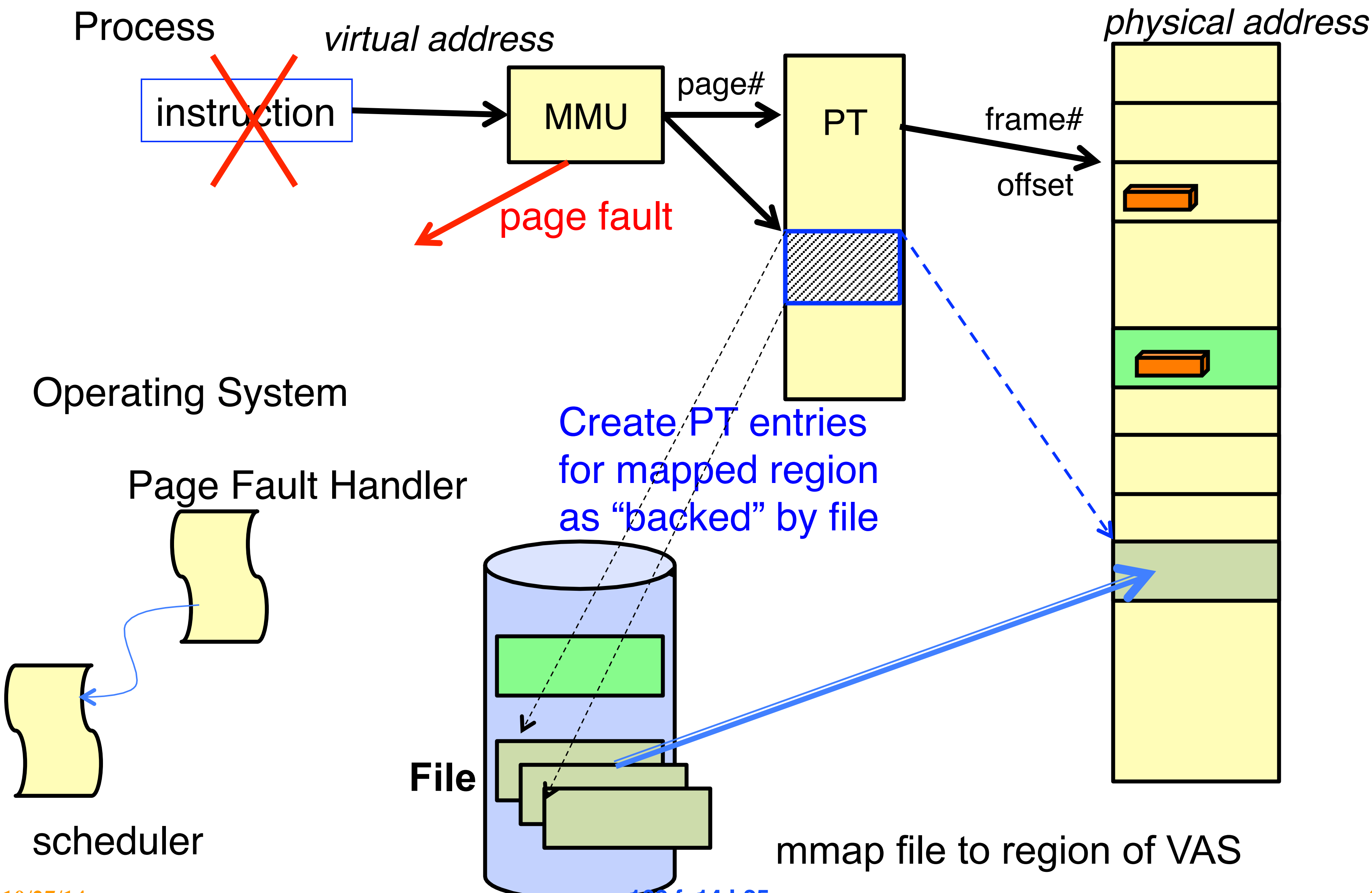
- **File System:**
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- **File defined by header, called "inode"**
- **Multilevel Indexed Scheme**
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS uses variable extents, rather than fixed blocks, and tiny files data is in the header
- **4.2 BSD Multilevel index files**
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization

File System Summary (2/2)

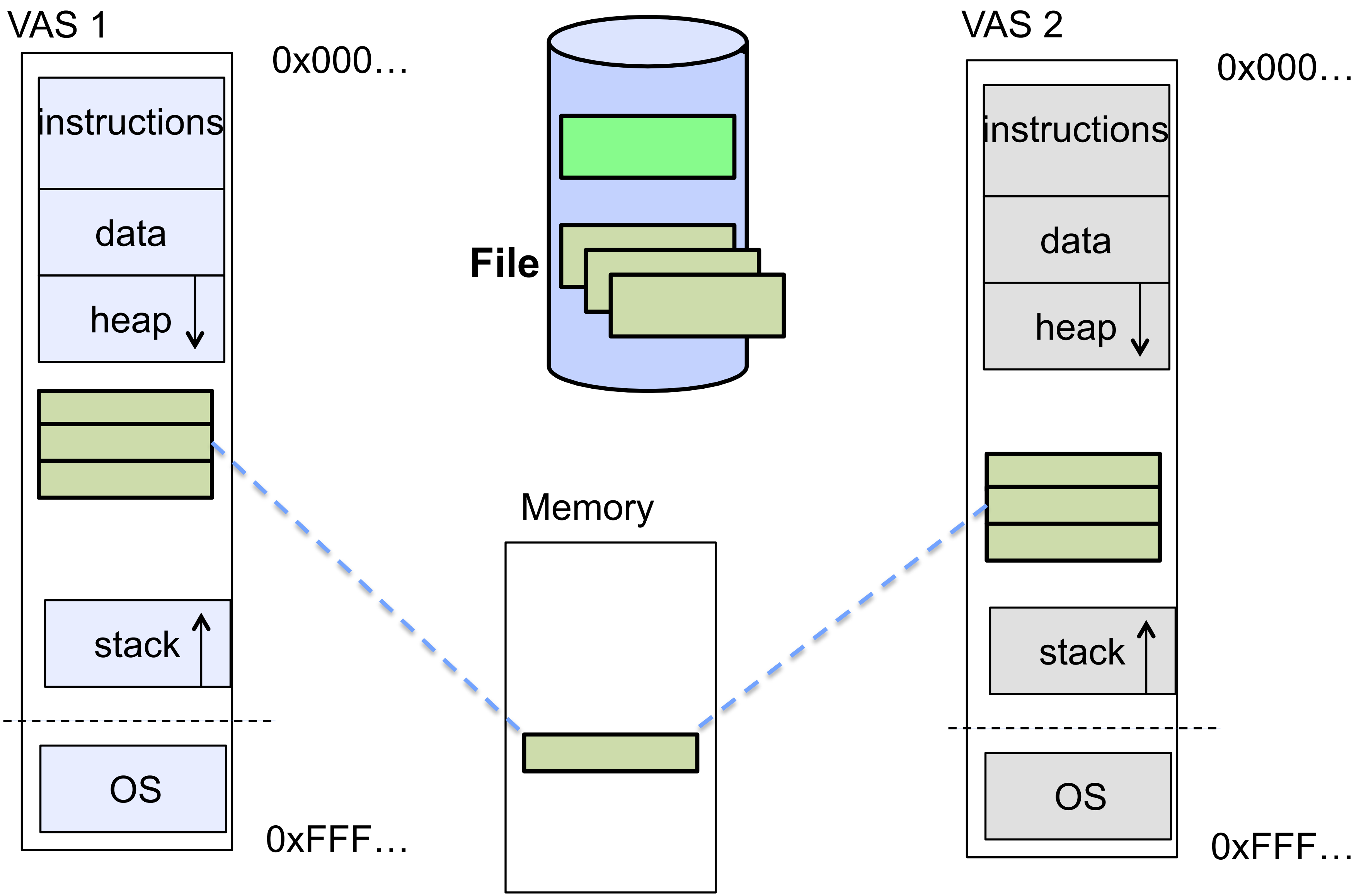
- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- File layout driven by freespace management
  - Integrate freespace, inode table, file blocks and directories into block group
- Copy-on-write creates new (better positioned) version of file upon burst of writes
- Deep interactions between memory management, file system, and sharing

Mid Term III

Using Paging to mmap files



Sharing through Mapped Files



Reliability and Availability

- A system is *reliable* if it performs its intended function.
- A system is *available* if it currently can respond to a request.

- A storage system's *reliability* is the probability that it will continue to be reliable for some specified period of time.
- Its *availability* is the probability that it will be available at any given time.

Definitions

- A system is *reliable* if it performs its intended function.
- A system is *available* if it currently can respond to a request.

- A storage system's *reliability* is the probability that it will continue to be reliable for some specified period of time.
- Its *availability* is the probability that it will be available at any given time.

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**

Achieving File System Reliability

- Problem posed by machine/disk failures
- Transaction concept
- Approaches to reliability
  - Careful sequencing of file system operations
  - Copy-on-write (WAFL, ZFS)
  - Journalling (NTFS, linux ext4) – Transactions within file system
  - Log structure (flash storage) – Transactions for user data too
- Approaches to availability
  - RAID

Reliability Approach #2: Copy on Write File Layout

- To update file system, write a new version of the file system containing the update
  - Never update in place
  - Reuse existing unchanged disk blocks
- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel
- Approach taken in network file server appliances (WAFL, ZFS)

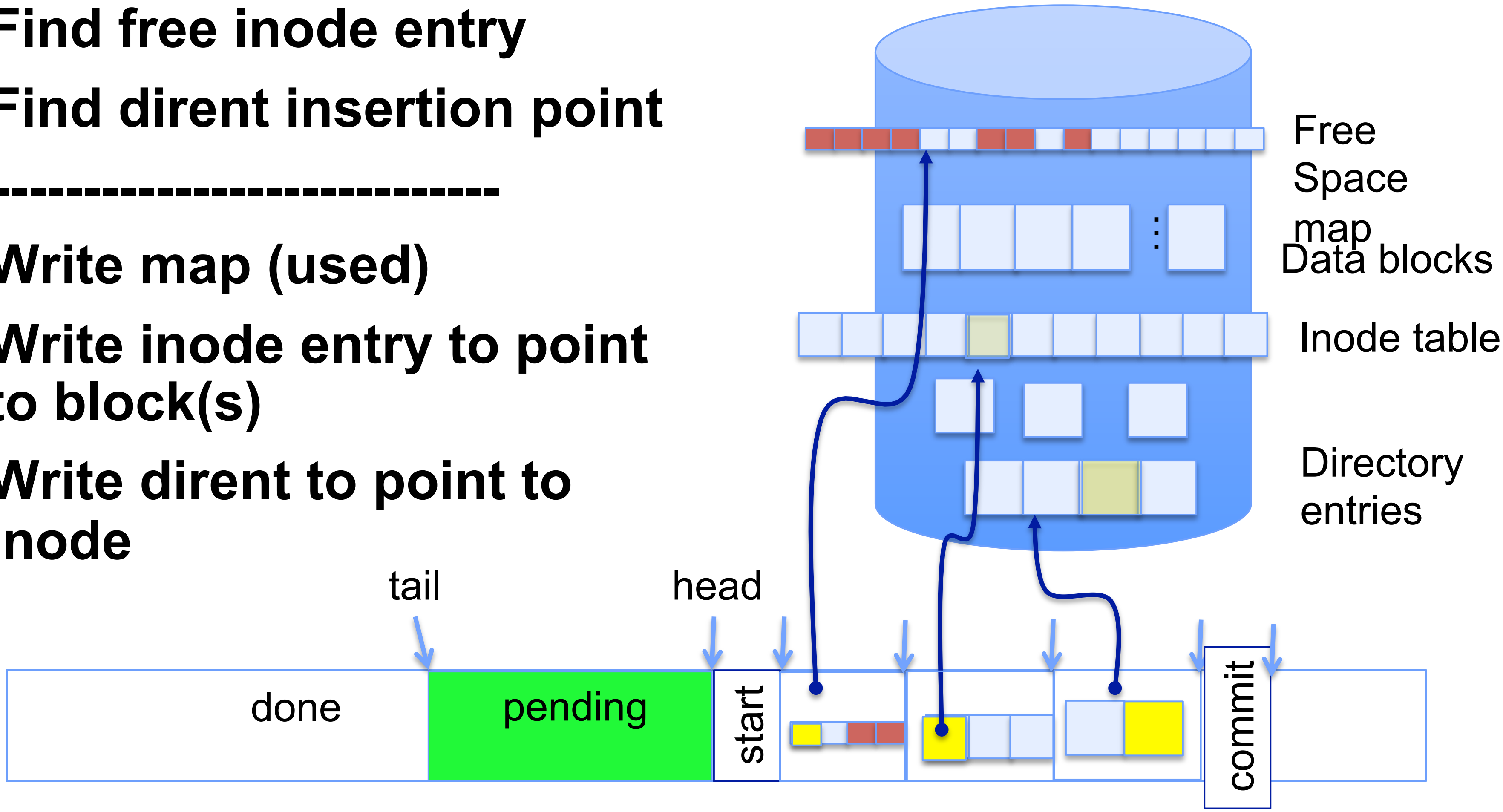
Redo Logging

- Prepare
  - Write all changes (in transaction) to log
- Commit
  - Single disk write to make transaction durable
- Redo
  - Copy changes to disk
- Garbage collection
  - Reclaim space in log
- Recovery
  - Read log
  - Redo any operations for committed transactions
  - Garbage collect log

Ex: Creating a file (as a transaction)

- **Find free data block(s)**
- **Find free inode entry**
- **Find dirent insertion point**

- **Write map (used)**
- **Write inode entry to point to block(s)**
- **Write dirent to point to inode**



Log in non-volatile storage (Flash or on Disk)

Performance

- Log written sequentially
  - Often kept in flash storage
- Asynchronous write back
  - Any order as long as all changes are logged before commit, and all write backs occur after commit
- Can process multiple transactions
  - Transaction ID in each log entry
  - Transaction completed iff its commit record is in log

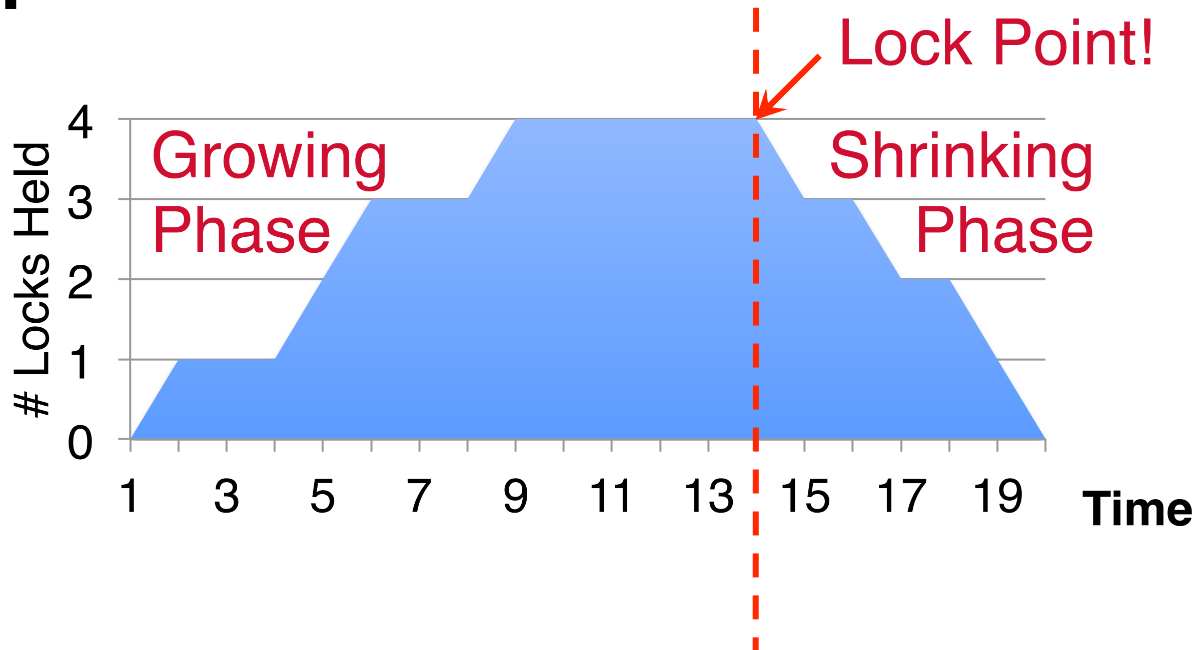
Two-Phase Locking (2PL)

1) Each transaction must obtain:
   - S (*shared*) or X (*exclusive*) lock on data before reading,
   - X (*exclusive*) lock on data before writing

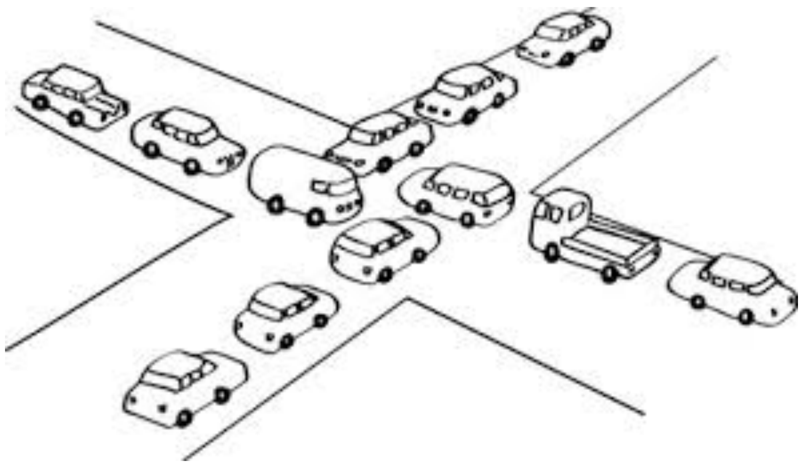
2) A transaction can not request additional locks once it releases any locks

Thus, each transaction has a "growing phase" followed by a "shrinking phase"

Avoid deadlock by acquiring locks in some lexicographic order



What's a Deadlock?



- Situation where all entities (e.g., threads, clients, …)
  - have acquired certain resources and
  - need to acquire additional resources,
  - but those additional resources are held some other entity that won't release them

Summary: Deadlock

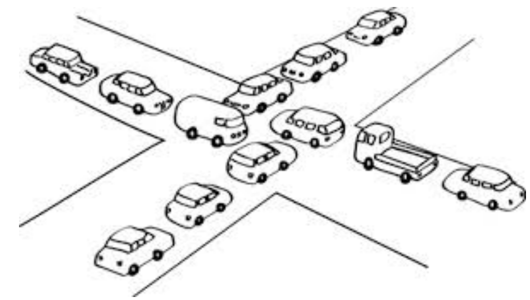
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Deadlock detection and preemption
- Deadlock prevention
  - Loop Detection, Banker's algorithm

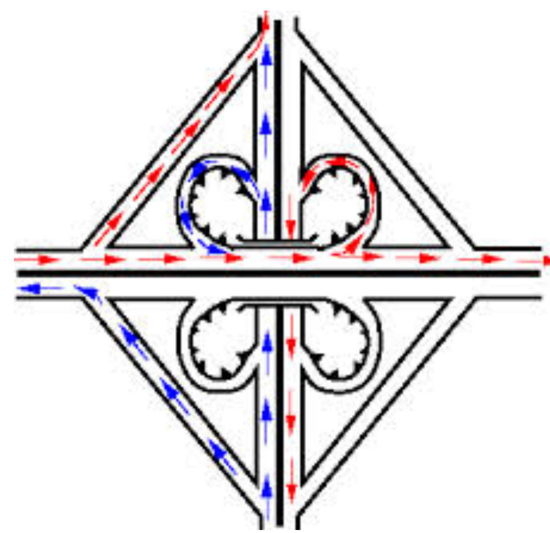
Methods for Handling Deadlocks

- Deadlock **prevention**: design system to ensure that it will *never* enter a deadlock
  - E.g., monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Allow system to enter deadlock and then recover
  - Requires deadlock **detection** algorithm
    - » E.g., Java JMX findDeadlockedThreads()
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ignore the problem and hope that deadlocks never occur in the system
  - Used by most operating systems, including UNIX
  - Resort to manual version of recovery

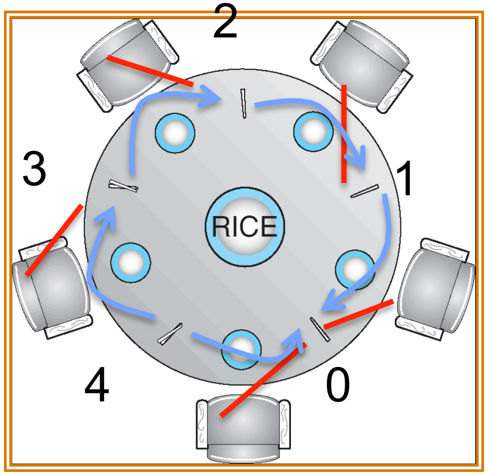
Techniques for Deadlock Prevention

- **Eliminate the Shared Resources**
- **Eliminate the Mutual Exclusion**
- **Eliminate Hold-and-Wait**
- **Permit pre-emption**
- **Eliminate the creation of circular wait**
  - Dedicated resources to break cycles
  - Ordering on the acquisition of resources





Ordered Acquisition to prevent cycle from forming



- **Suppose everyone grabs lowest first**
- **Dependence graph is acyclic**
- **Someone will fail to grab chopstick 0 !**
- **How do you modify the rule to retain fairness ?**
- **OS: define ordered set of resource classes**
  - **Acquire locks on resources in order**
  - **Page Table => Memory Blocks => …**

Two-Phase Locking (2PL)

- **2PL guarantees that the dependency graph of a schedule is acyclic.**
- **For every pair of transactions with a conflicting lock, one acquires it first → ordering of those two → total ordering.**

- **Therefore 2PL-compatible schedules are conflict serializable.**

- **Note: 2PL can still lead to deadlocks since locks are acquired incrementally.**

- **An important variant of 2PL is strict 2PL, where all locks are released at the end of the transaction.**

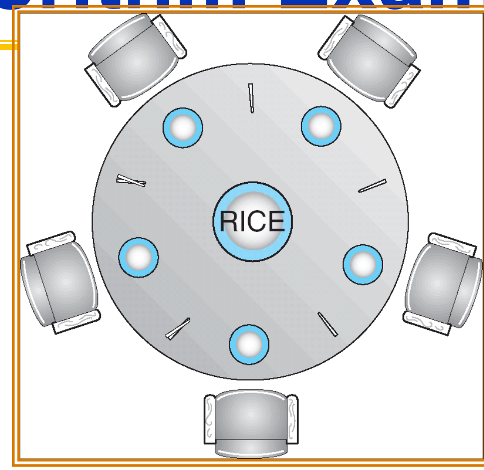
Transaction Isolation

Process A:	Process B:
LOCK x, y	LOCK x, y and `log`
move foo from dir x to dir y	grep across x and y
`mv x/foo y/`	`grep 162 x/* y/* > log`
	Commit and Release x, y, log

Commit and Release x, y

- **grep appears either before or after move**
- **Need log/recover AND 2PL to get ACID**

Banker's Algorithm Example



- **Banker's algorithm with dining philosophers**
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed philosophers? Don't allow
    - » It's the last one, no one would have k
    - » It's 2^{nd} to last, and no one would have k-1
    - » It's 3^{rd} to last, and no one would have k-2
    - » …

What Is A Protocol?

- A protocol is an agreement on how to communicate
- Includes
  - Syntax: how a communication is specified & structured
    - » Format, order messages are sent and received
  - Semantics: what a communication means
    - » Actions taken when transmitting, receiving, or when a timer expires

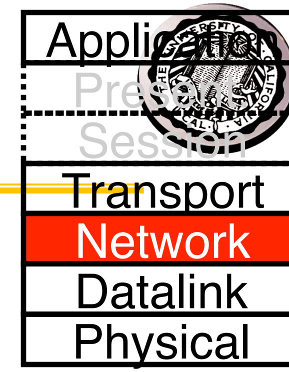
Network System Modularity

Like software modularity, but:

- **Implementation distributed across many machines (routers and hosts)**
- **Must decide:**
  - How to break system into modules:
    - » Layering
  - What functionality does each module implement:
    - » End-to-End Principle: don't put it in the network if you can do it in the endpoints.
- **Partition the system**
  - Each layer solely relies on services from layer below
  - Each layer solely exports services to layer above
- **Interface between layers defines interaction**
  - Hides implementation details
  - Layers can change without disturbing other layers

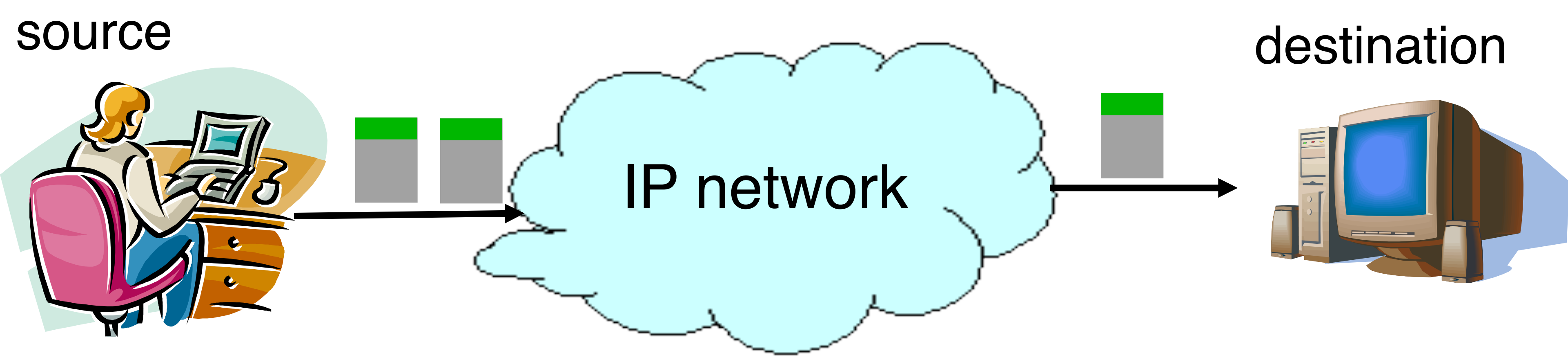
The E2E Concept

- Traditional Engineering Goal: design the infrastructure to meet application requirements
  - Optimizing for Cost, Reliability, Performance, …
- Challenge: infrastructure is most costly & difficult to create and evolves most slowly
  - Applications evolve rapidly, as does technology
- End-to-end Design Concept
  - Utilize intelligence at the point of application
  - Infrastructure need not meet all application requirements directly
  - Only what the end-points cannot reasonably do themselves
    - » Avoid redundancy, semantic mismatch, …
  - Enable applications and incorporate technological advance
- Design for Change! - and specialization
  - Layers & protocols

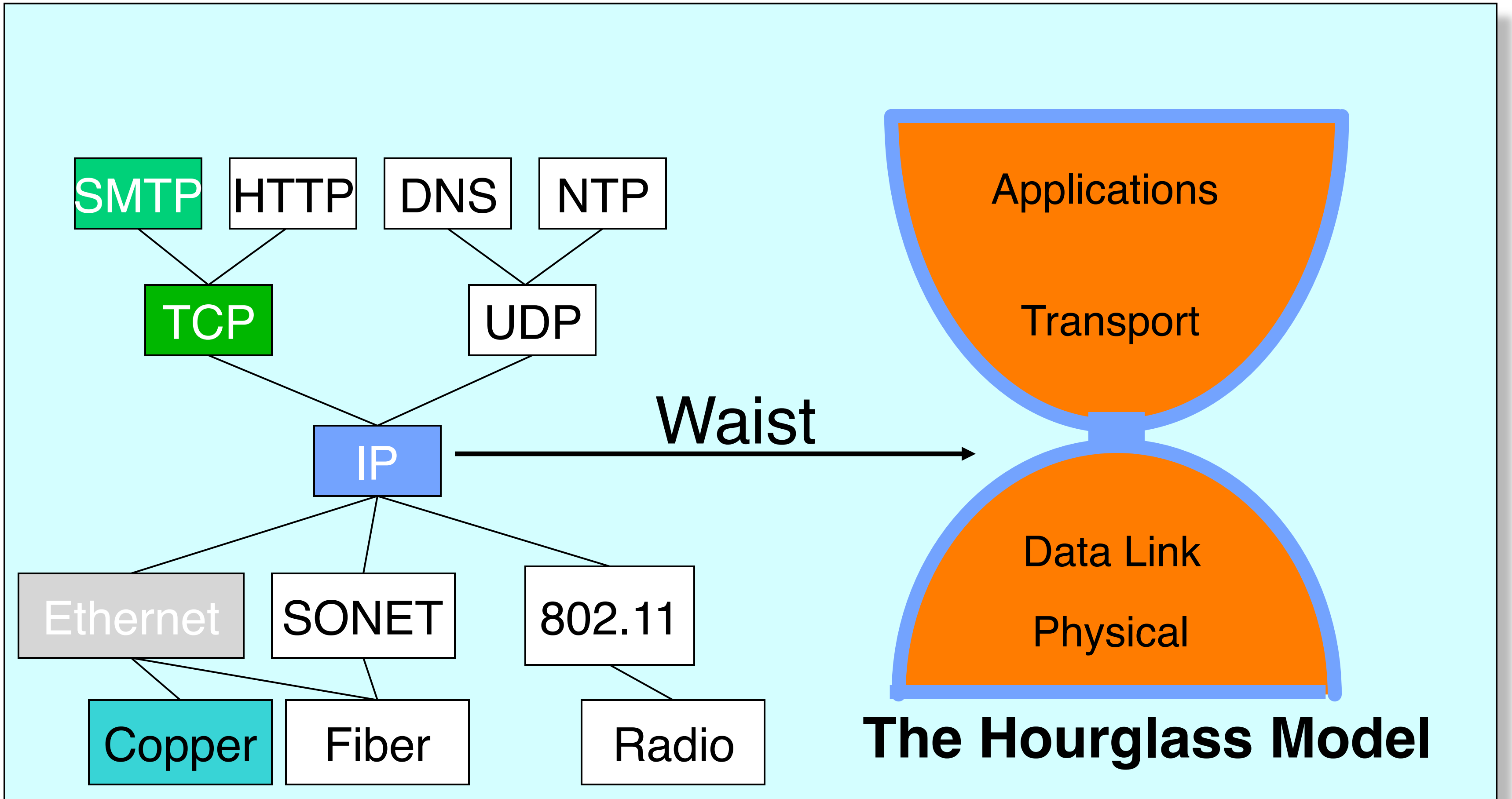


Internet Protocol (IP)

- **Internet Protocol: Internet's network layer**
- **Service it provides: "Best-Effort" Packet Delivery**
  - **Tries it's "best" to deliver packet to its destination**
  - **Packets may be lost**
  - **Packets may be corrupted**
  - **Packets may be delivered out of order**

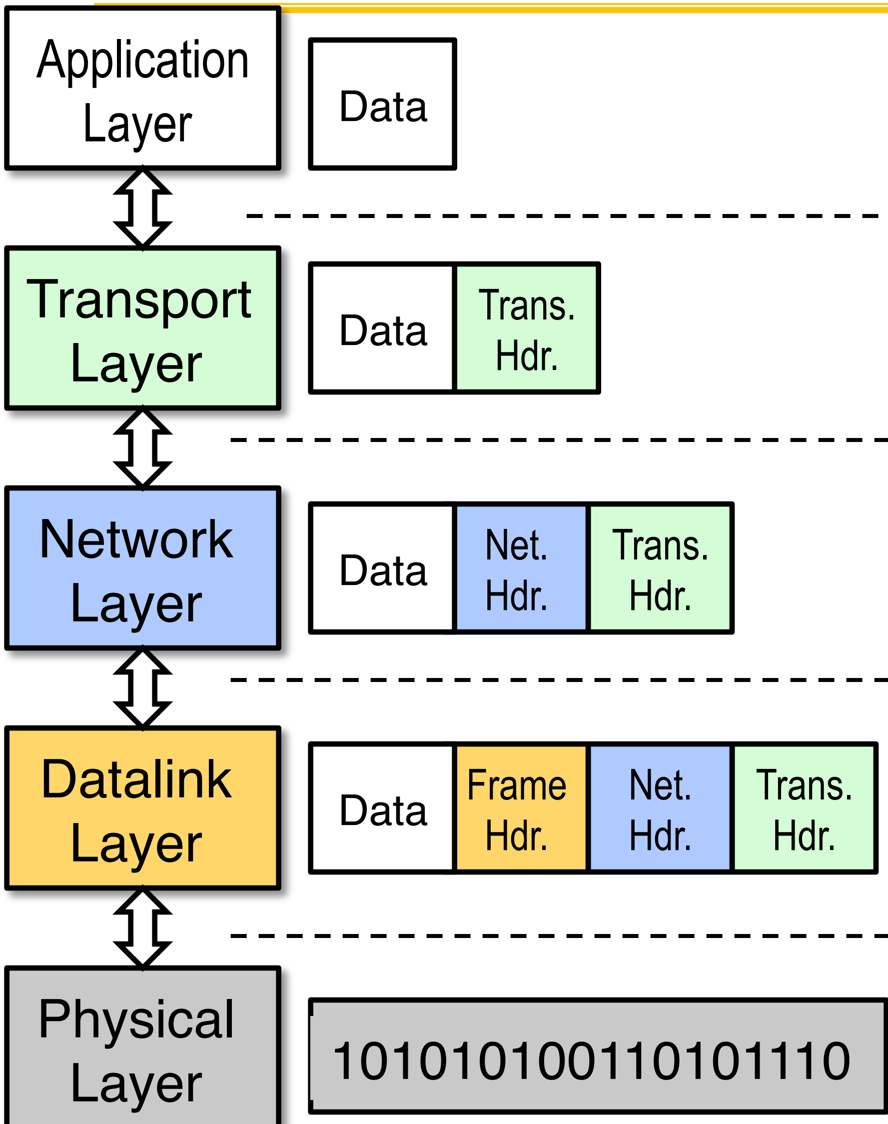


The Internet *Hourglass*



There is just one network-layer protocol, **IP**
The “narrow waist” facilitates interoperability

Internet Layering – engineering for intelligence and change



Any distributed protocol (e.g., HTTP, Skype, p2p, KV protocol in your project)

Send *segments* to another *process* running on same or different node

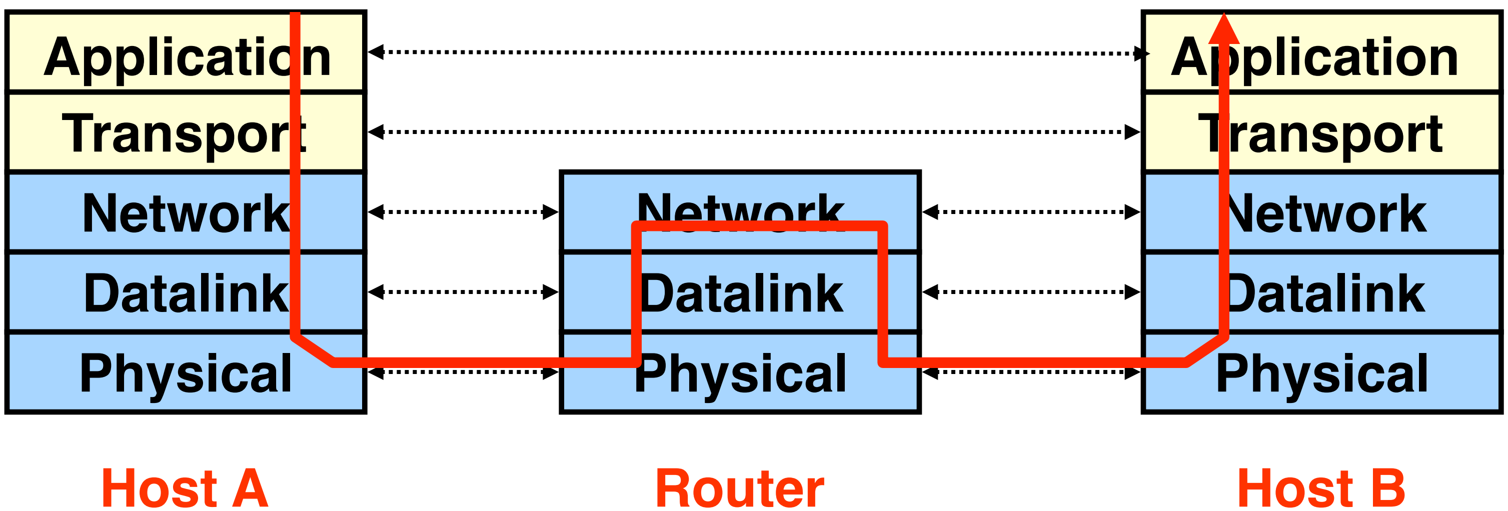
Send *packets* to another node possibly *located* in a different network

Send *frames* to other node directly connected to same physical network

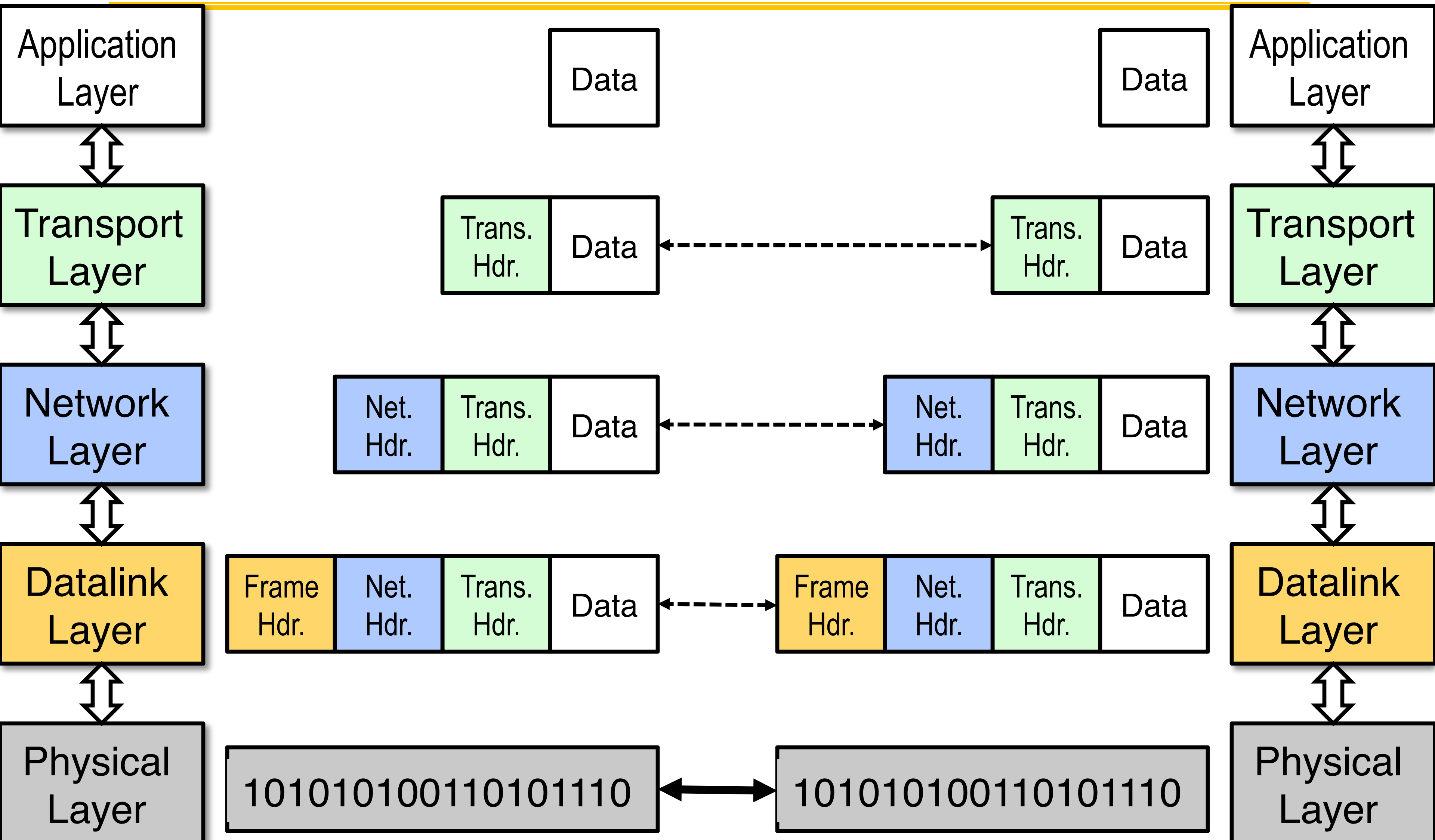
Send *bits* to other node directly connected to same physical network

Internet Architecture: The Five Layers

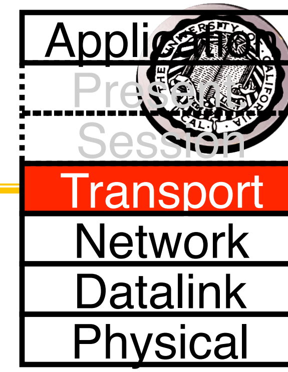
- Lower three layers implemented everywhere
- Top two layers implemented only at hosts
- Logically, layers interacts with peer's corresponding layer



Layering: Packets in Envelopes



Internet Transport Protocols



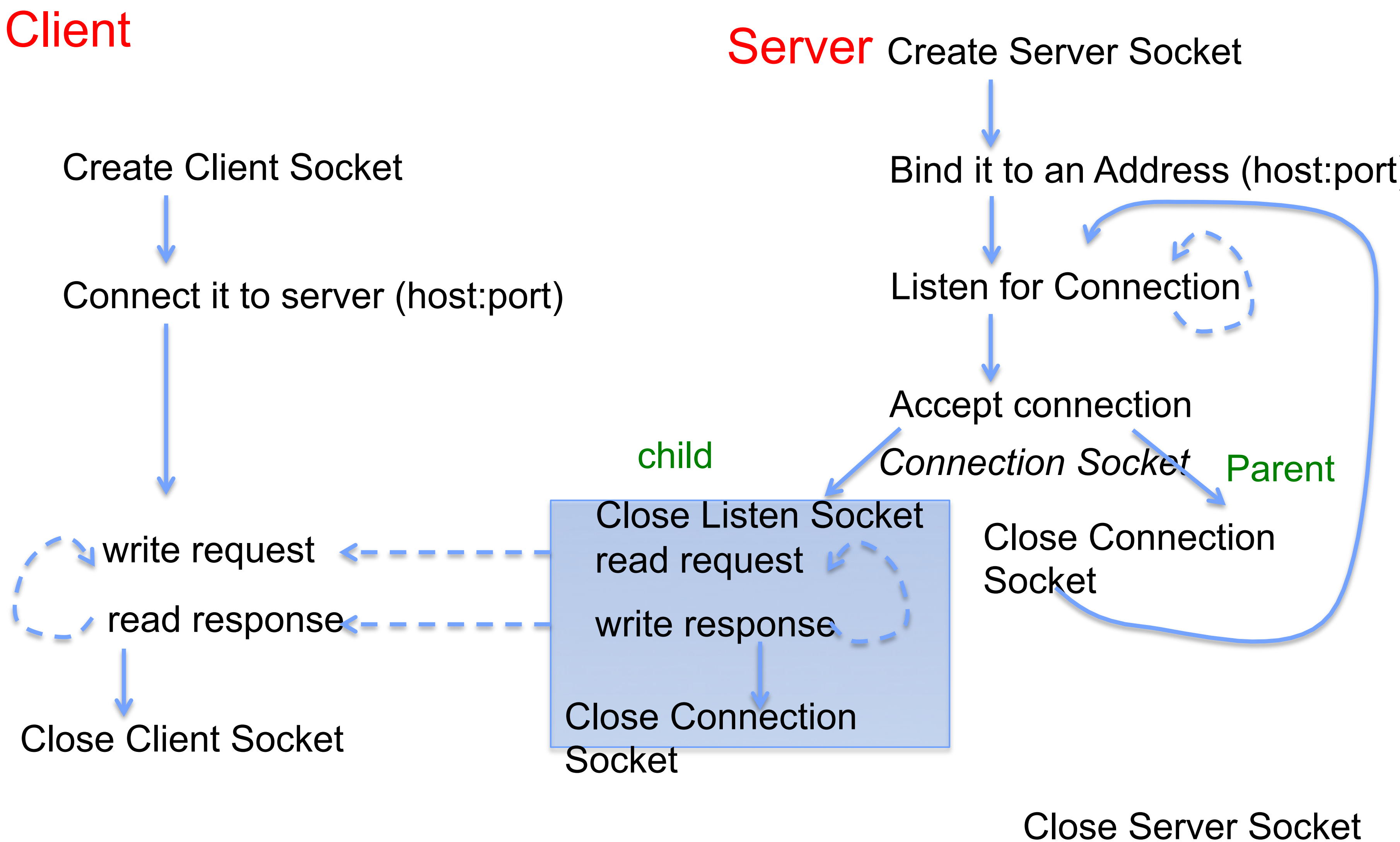
- **Datagram service (UDP)**
  - **No-frills extension of "best-effort" IP**
  - **Multiplexing/Demultiplexing among processes**
- **Reliable, in-order delivery (TCP)**
  - **Connection set-up & tear-down**
  - **Discarding corrupted packets (segments)**
  - **Retransmission of lost packets (segments)**
  - **Flow control**
  - **Congestion control**
- **Services not available**
  - **Delay and/or bandwidth guarantees**
  - **Sessions that survive change-of-IP-address**

Transport Layer (4)

Application
Presentation
Session
Transport
Network
Datalink
Physical

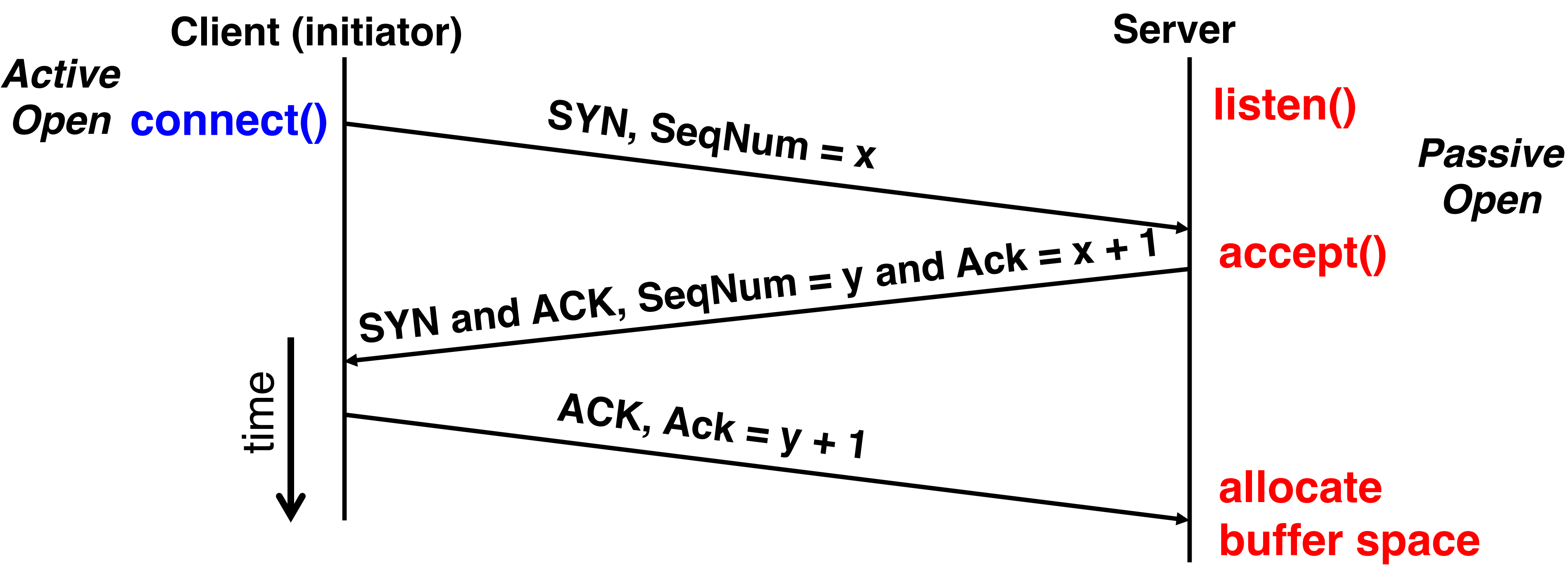
- **Service:**
  - Provide end-to-end communication between **processes**
  - **Demultiplexing** of communication between hosts
  - Possible other services:
    - » **Reliability** in the presence of errors
    - » **Timing** properties
    - » **Rate adaption** (flow-control, congestion control)
- **Interface: send message to "specific process" at given destination; local process receives messages sent to it**
  - How are they named?
- **Protocol: port numbers, perhaps implement reliability, flow control, packetization of large messages, framing**
- **Prime Examples: TCP and UDP**

Sockets in concept

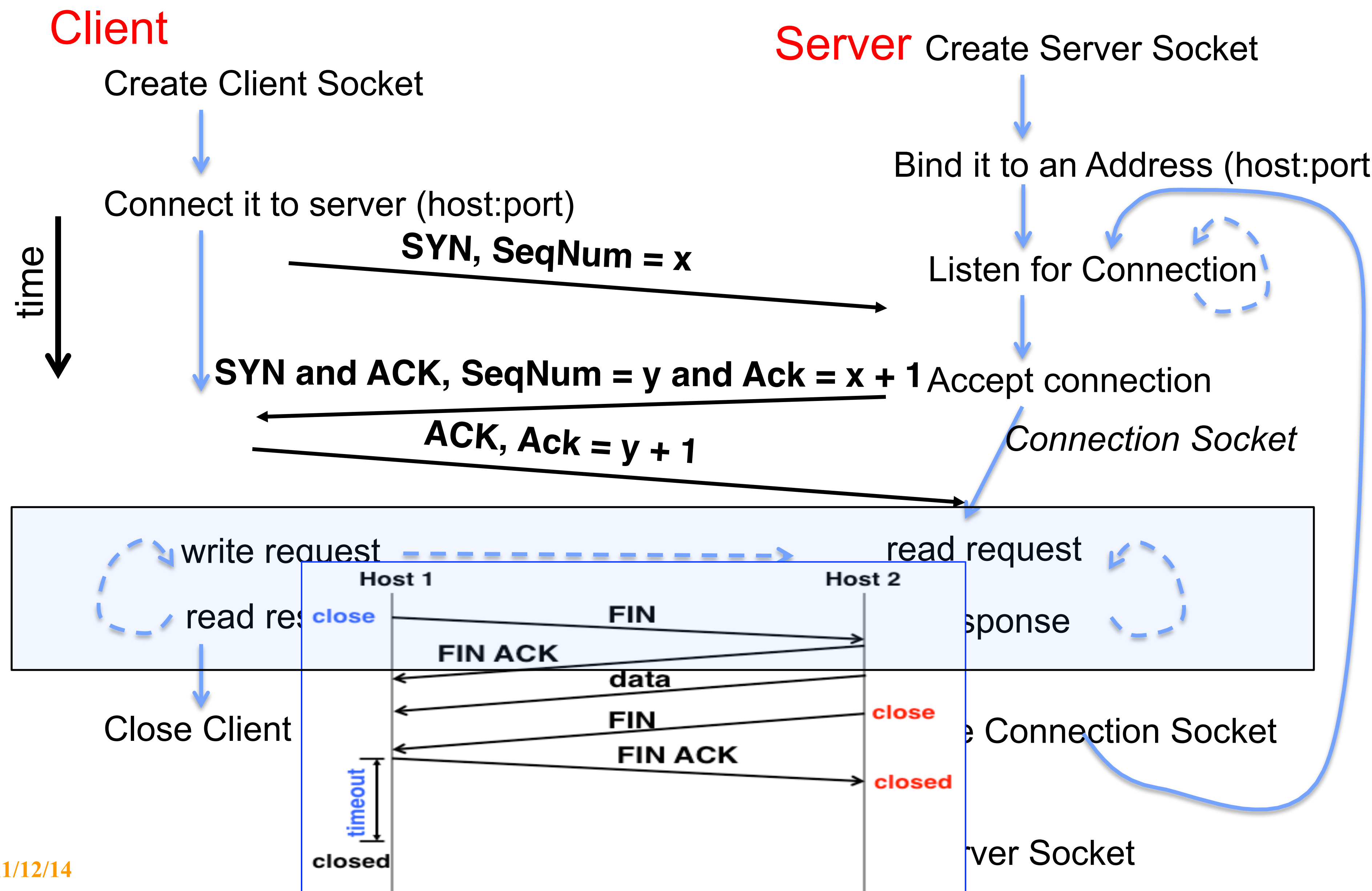


Open Connection: 3-Way Handshaking

- **If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing**
  - **Client's sequence number incremented by one, (x + 1)**
    - » **Why is this needed?**
  - **A sequence number proposal, y, for first byte server will send**

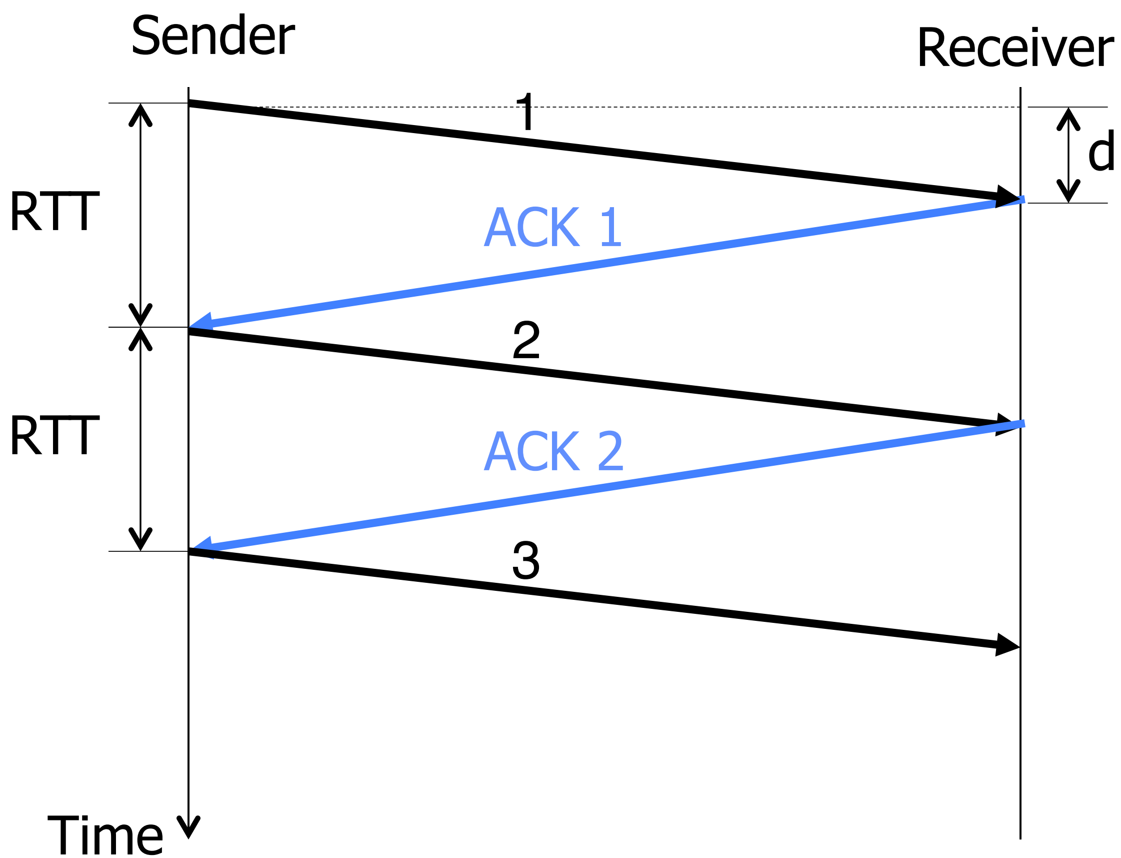


Recall: Connecting API to Protocol



Stop & Wait w/o Errors

- **Send; wait for ack; repeat**
- **RTT: Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back**
  - **One-way latency (d): one way delay from sender and receiver**



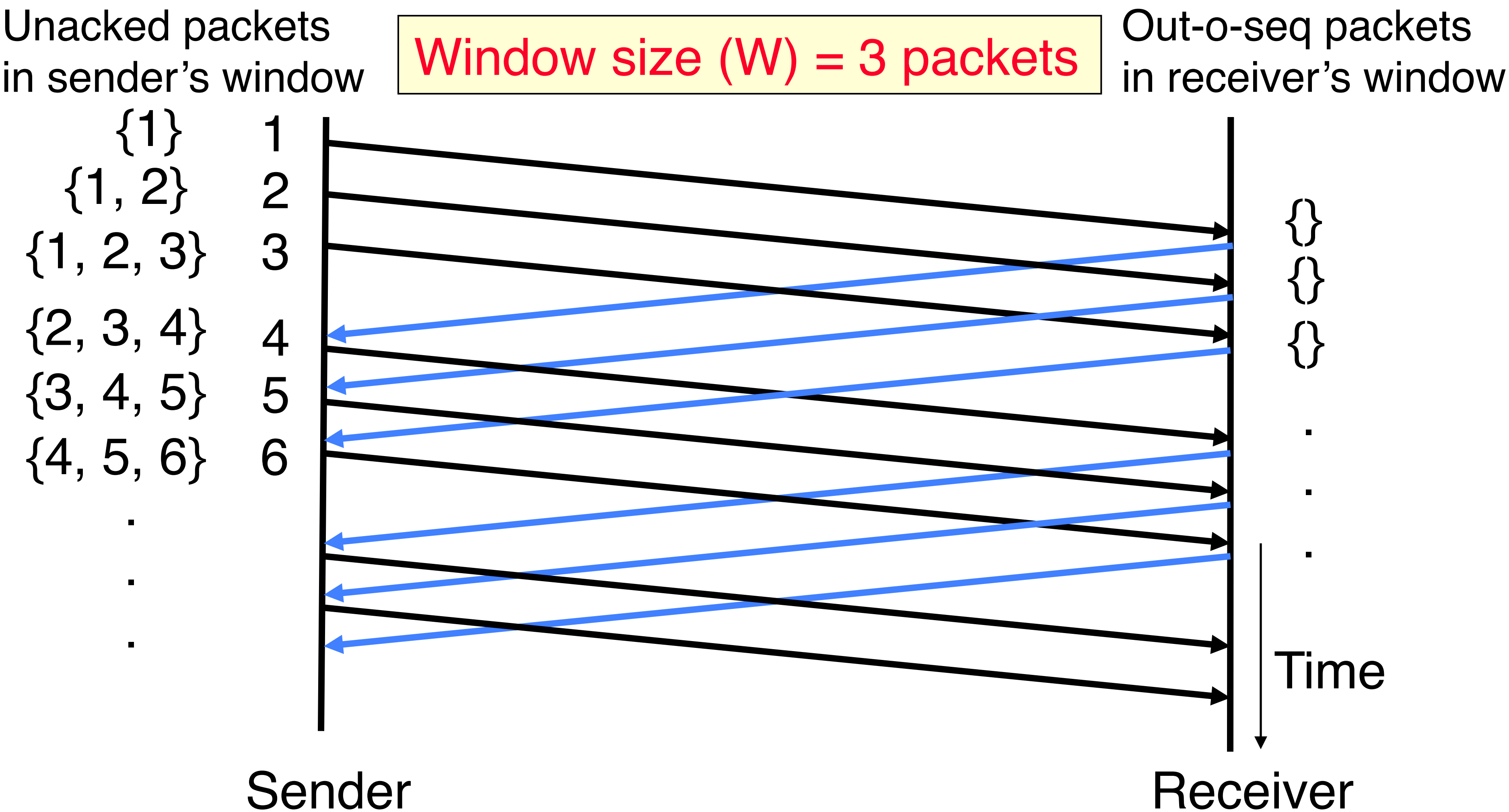
RTT = 2*d
(if latency is symmetric)

Sliding Window

- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*
- Assume window size is n
- Let A be the last ACK'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
- Sender can send packets in its window
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
- Receiver can accept out of sequence, if in window

Sliding Window w/o Errors

- **Throughput = W*packet_size/RTT**



Example: Sliding Window w/o Errors

- **Assume**
  - Link capacity, C = 1Gbps
  - Latency between end-hosts, RTT = 80ms
  - packet_length = 1000 bytes
- **What is the window size W to match link's capacity, C?**

- **Solution**

  We want Throughput = C

  Throughput = W*packet_size/RTT

  C = W*packet_size/RTT

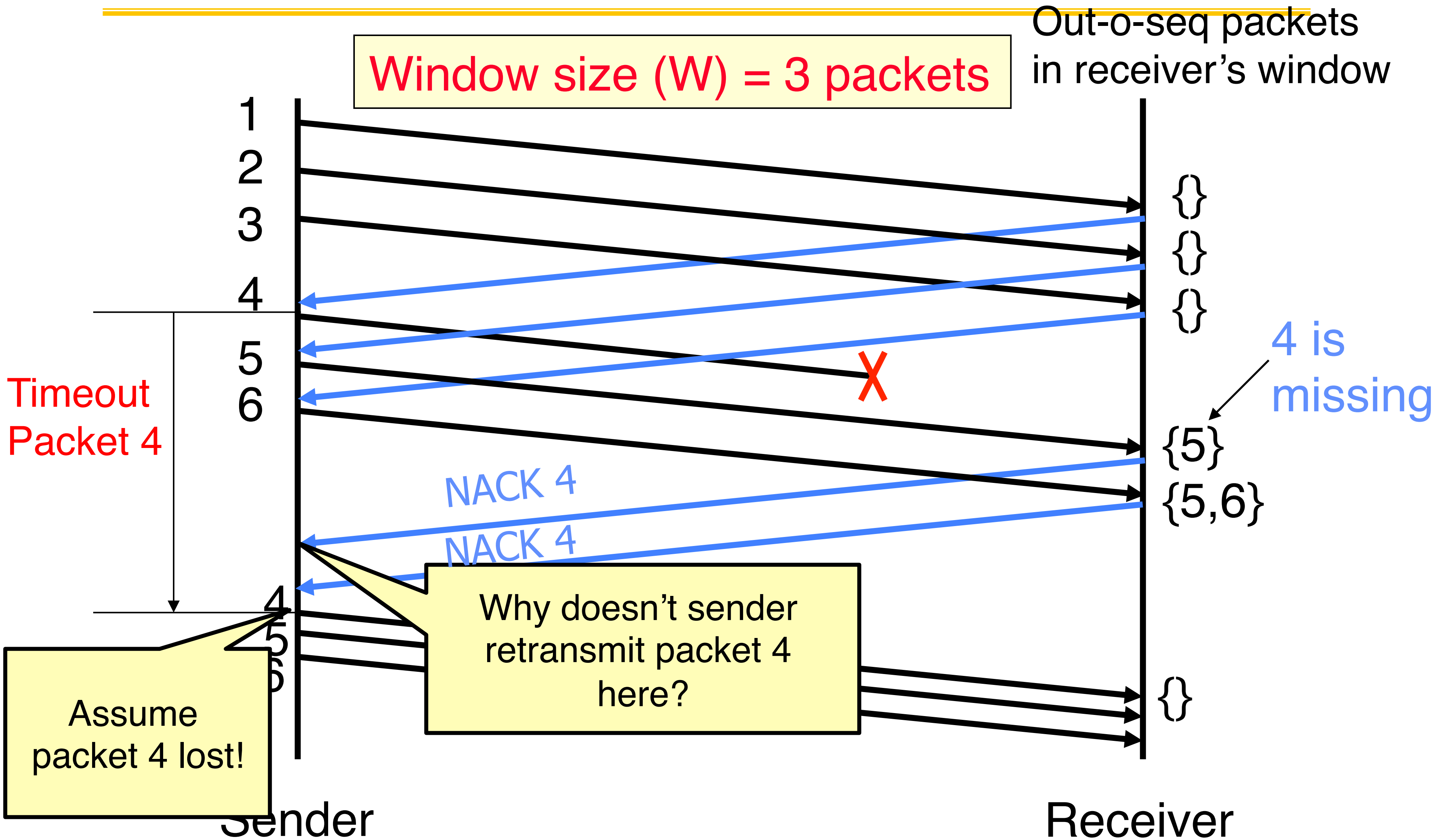
  W = C*RTT/packet_size = 10^9bps*80*10^{-3}s/(8000b) = 10^4 packets

Bandwidth-Delay Product

Window size ~ Bandwidth (Capacity) x delay (RTT/2)

Remember Little's Law !

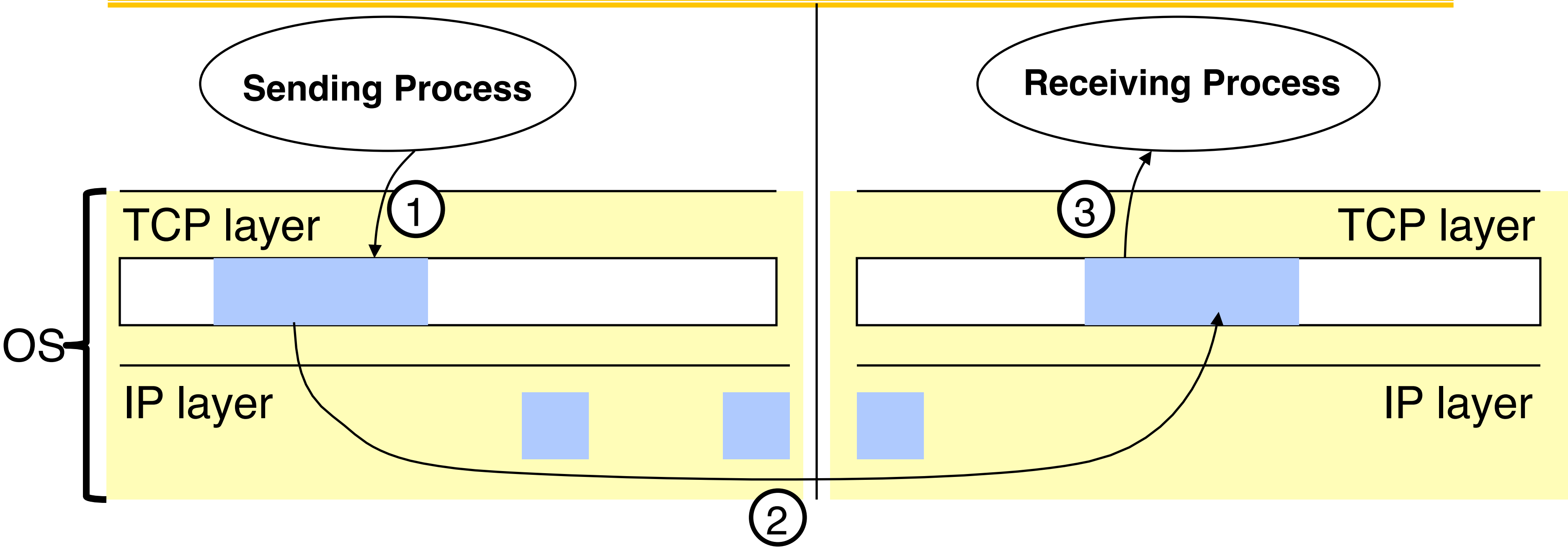
GBN Example with Errors



TCP Flow Control

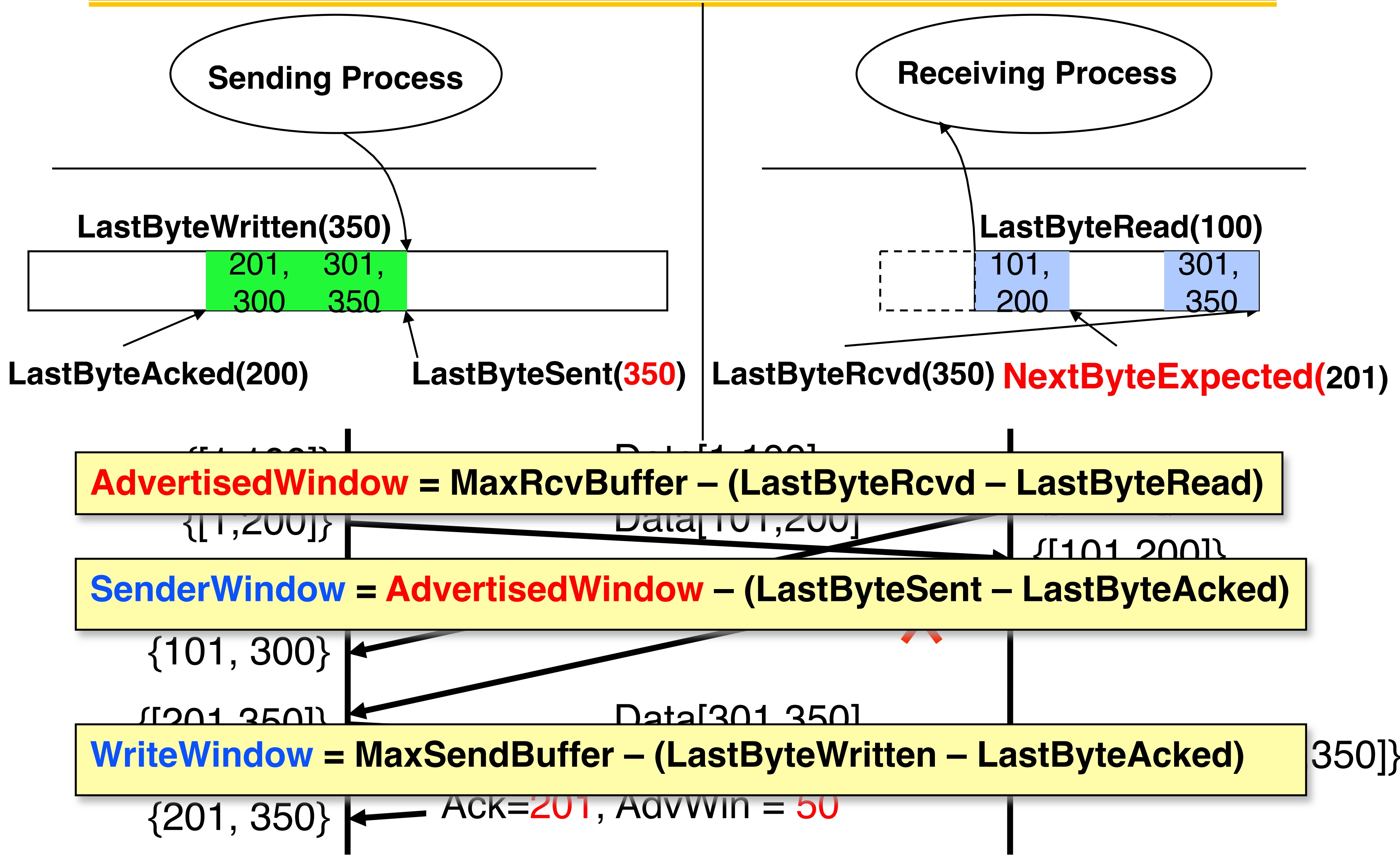
- **TCP: sliding window protocol at byte (not packet) level**
  - Go-back-N: TCP Tahoe, Reno, New Reno
  - Selective Repeat (SR): TCP Sack
- **Receiver tells sender how many more bytes it can receive without overflowing its buffer**
  - the AdvertisedWindow
- **The ACK contains sequence number N of next byte the receiver expects,**
  - receiver has received all bytes in sequence up to and including N-1

TCP Flow Control



- **Three pairs of producer-consumer's**
  - ① sending process → sending TCP
  - ② Sending TCP → receiving TCP
  - ③ receiving TCP → receiving process

Recap: TCP Flow Control



AdvertisedWindow = MaxRcvBuffer – (LastByteRcvd – LastByteRead)

SenderWindow = AdvertisedWindow – (LastByteSent – LastByteAcked)

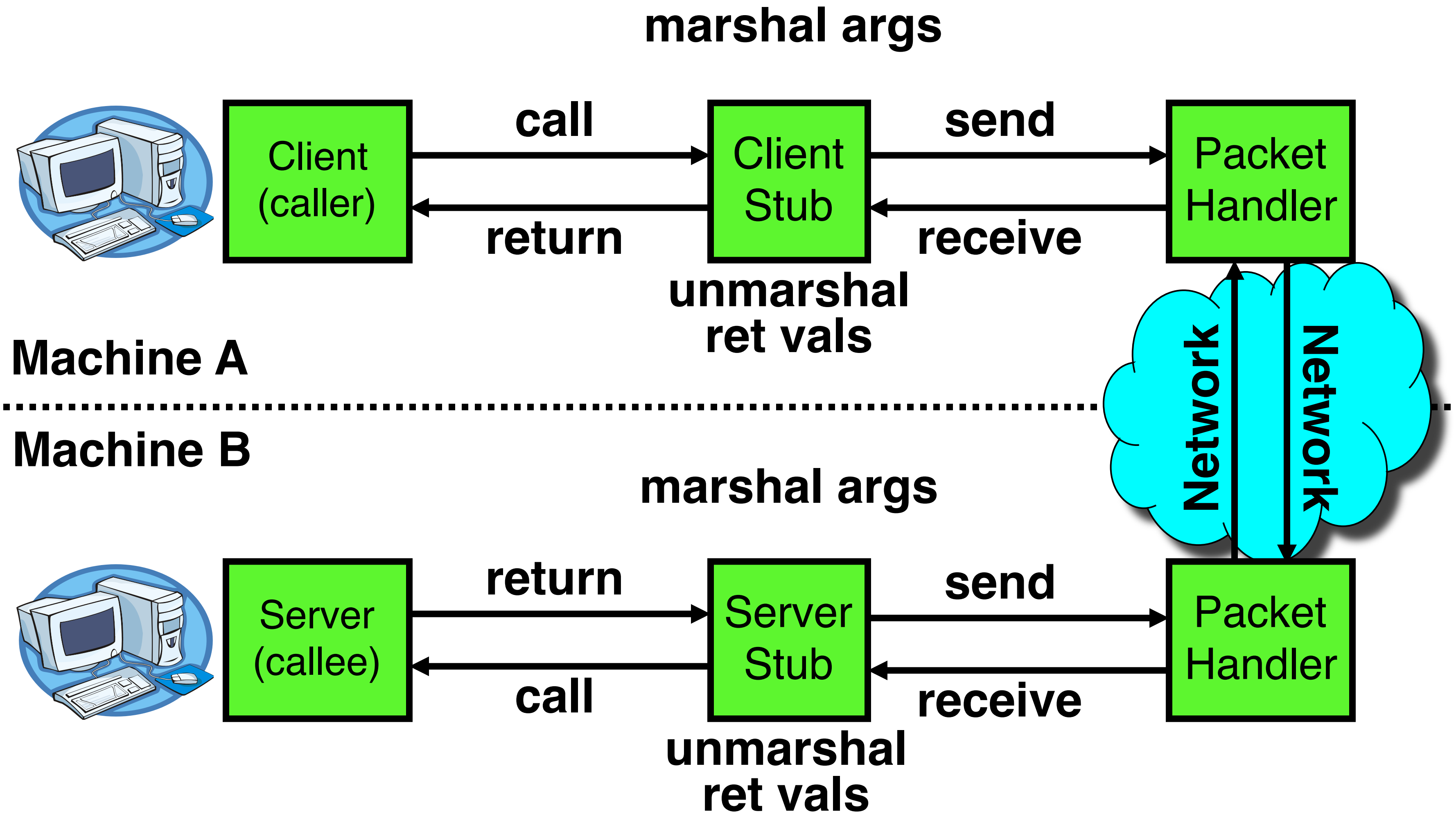
WriteWindow = MaxSendBuffer – (LastByteWritten – LastByteAcked)

Summary: Reliability & Flow Control

- **Flow control: three pairs of producer consumers**
  - Sending process → sending TCP
  - Sending TCP → receiving TCP
  - Receiving TCP → receiving process

- **AdvertisedWindow: tells sender how much new data the receiver can buffer**
- **SenderWindow: specifies how more the sender can transmit.**
  - Depends on AdvertisedWindow and on data sent since sender received AdvertisedWindow
- **WriteWindow: How much more the sending application can send to the sending OS**

Review: Remote Procedure Call



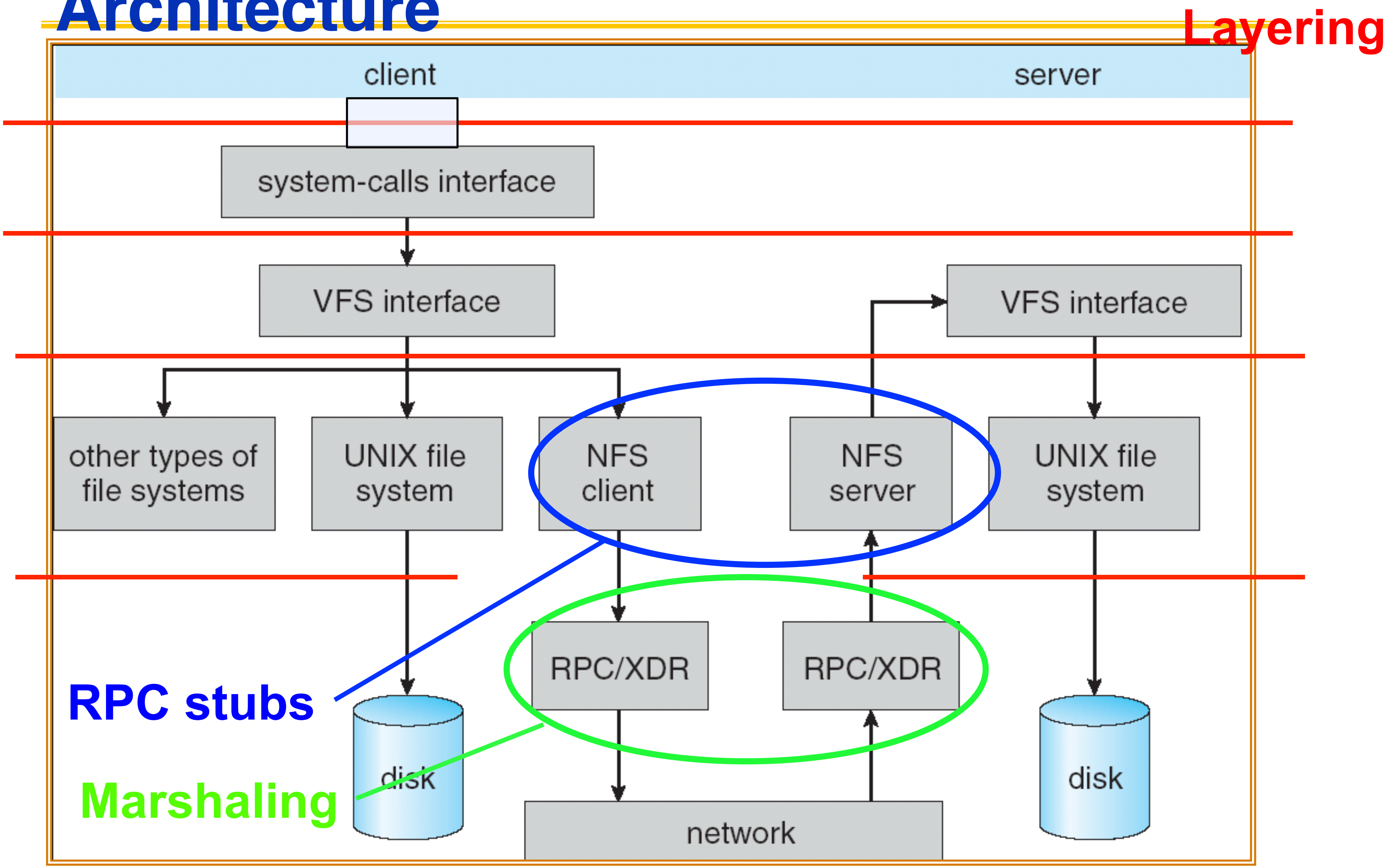
Six steps

1. The client calls the client stub. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called marshalling.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The server stub unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.
6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction

Motivation for RPC

- **RPC's can be used to communicate between processes on different machines or the same machine**
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote

Review: Schematic View of NFS Architecture



Goals of NFS

- **Transparent File Access**
  - Programs access remote files in the same way as local files
  - Programs cannot tell which file system is being used
- **Simple Crash Recovery**
  - When file server crashes
  - When client crashes
  - When network is down
- **Adequate Performance**
  - Not slower than other network utilities, e.g., rcp
  - Original NFS paper sets the goal 80% as fast as local disk

Transparent File Access

- **Don't need to use different APIs for different file systems**
  - Provide UNIX file system interface
    - » `open()`, `read()`, `write()`, `close()`, `mkdir()`, etc.
- **Don't need to know which file system is being used**
  - Virtual File System and vnode
    - » Abstraction layer for multiple file systems, including NFS
- **Don't need to provide file-system-specific parameters during file operation**
  - The idea of "early binding" doing mount
  - For NFS
    - » Client only specifies server hostname when mounting the NFS
    - » No need to know hostname while working on files

NFS Design Principles

- **Stateless protocol: A protocol in which all information required to process a request is passed with request**
  - Server keeps no state about client
  - Thus, if server crashed and restarted, requests can continue where left off (in many cases)
- **Idempotency: Performing requests multiple times has same effect as performing it exactly once, e.g., writing value to memory.**
  - If server's response doesn't come back to client (e.g., network failure, server crashes and restarts)
  - Client simply **retries** the same request, which will have the same effect.
  - Even if the server already did the job, it can re-do it because of idempotency.

The Shared Storage Abstraction

- Information (and therefore control) is communicated from one point of computation to another by
  - The former storing/writing/sending to a location in a shared address space
  - And the second later loading/reading/receiving the contents of that location

- Memory (address) space of a process
- File systems
- Dropbox, …
- Google Docs, …
- Facebook, …

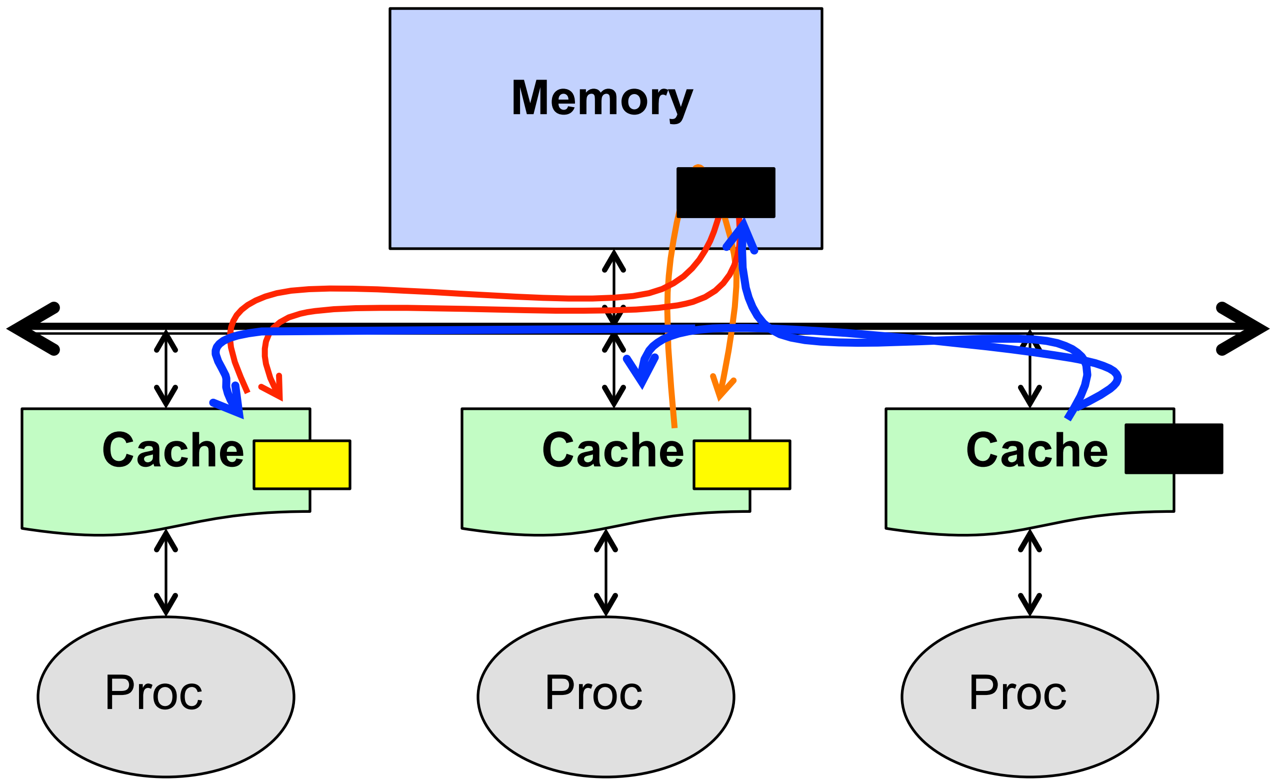
What are you assuming?

- Writes happen
  - Eventually a write will become visible to readers
  - Until another write happens to that location
- Within a sequential thread, a read following a write returns the value written by that write
  - Dependences are respected
  - Here a control dependence
  - Each read returns the most recent value written to the location
- A sequence of writes will be visible in order
  - Control dependences
  - Data dependences
  - May not see every write, but the ones seen are consistent with order written
- A readers see a consistent order
  - It is as if the total order was visible to all and they took samples

Basic solution to multiple client replicas

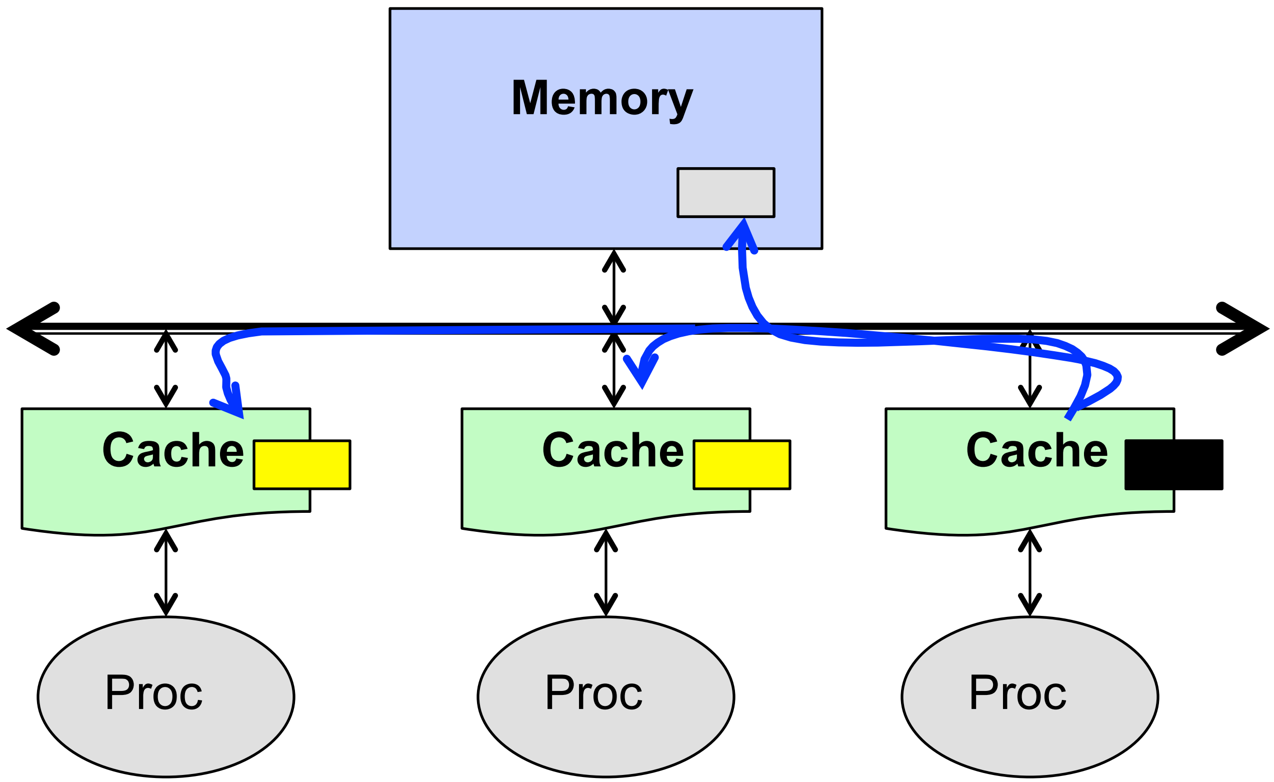
- Enforce single-writer multiple reader discipline
- Allow readers to cache copies
- Before an update is performed, writer must gain exclusive access
- Simple Approach: invalidate all the copies then update
- Who keeps track of what?

The Multi-processor/Core case



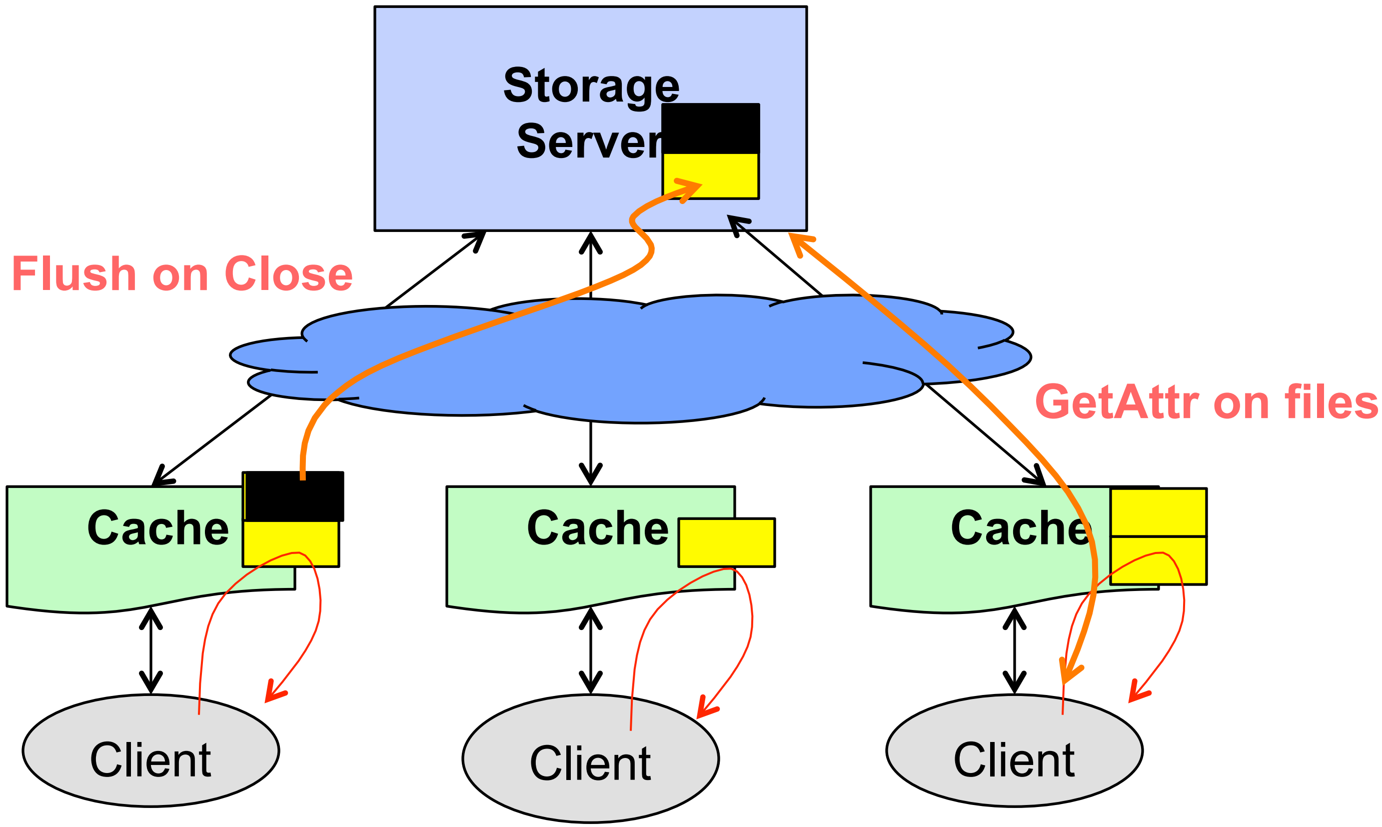
- **Interconnect is a broadcast medium**
- **All clients can observe all writes and invalidate local replicas (write-thru invalidate protocol)**

The Multi-processor/Core case



- **Write-Back via read-exclusive**
- **Atomic Read-modify-write**

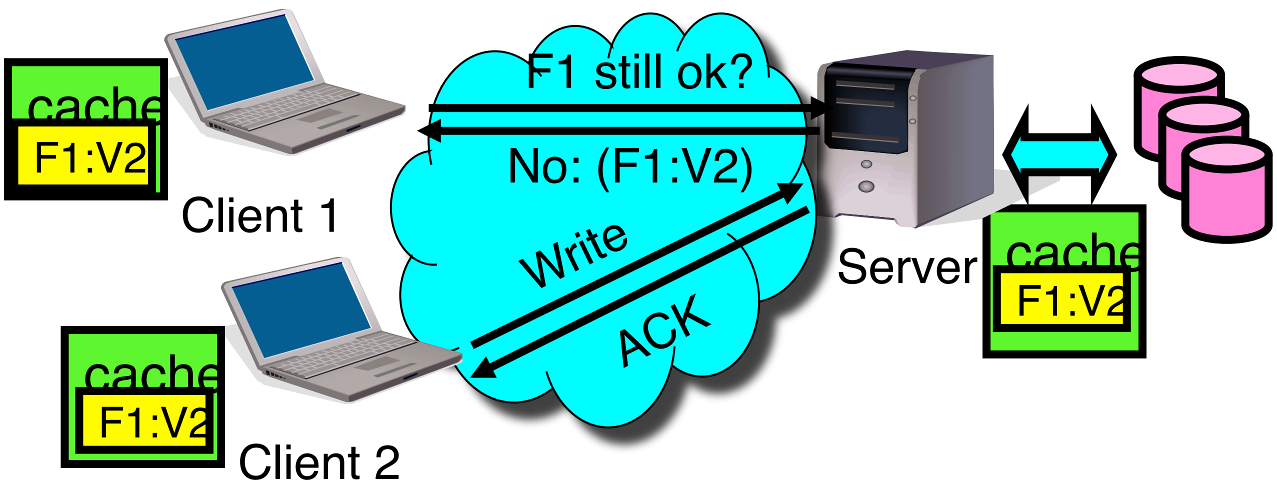
NFS "Eventual" Consistency



- **Stateless server allows multiple cached copies**
  - **Files written locally (at own risk)**
- **Update Visibility by "flush on close"**
- **GetAttributes on file ops to check modify since cache**

NFS Caching Consistency

- **NFS protocol: weak consistency**
  - **Client polls server periodically to check for changes**
    - » **Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).**
    - » **Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.**

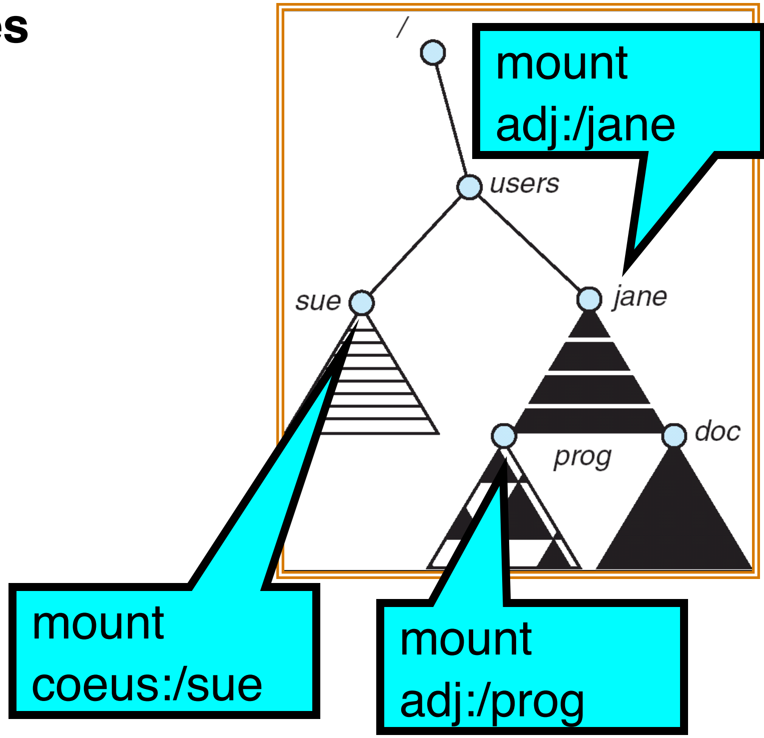


What if multiple clients write to same file?

- » **In NFS, can get either version (or parts of both)**
- » **Completely arbitrary!**

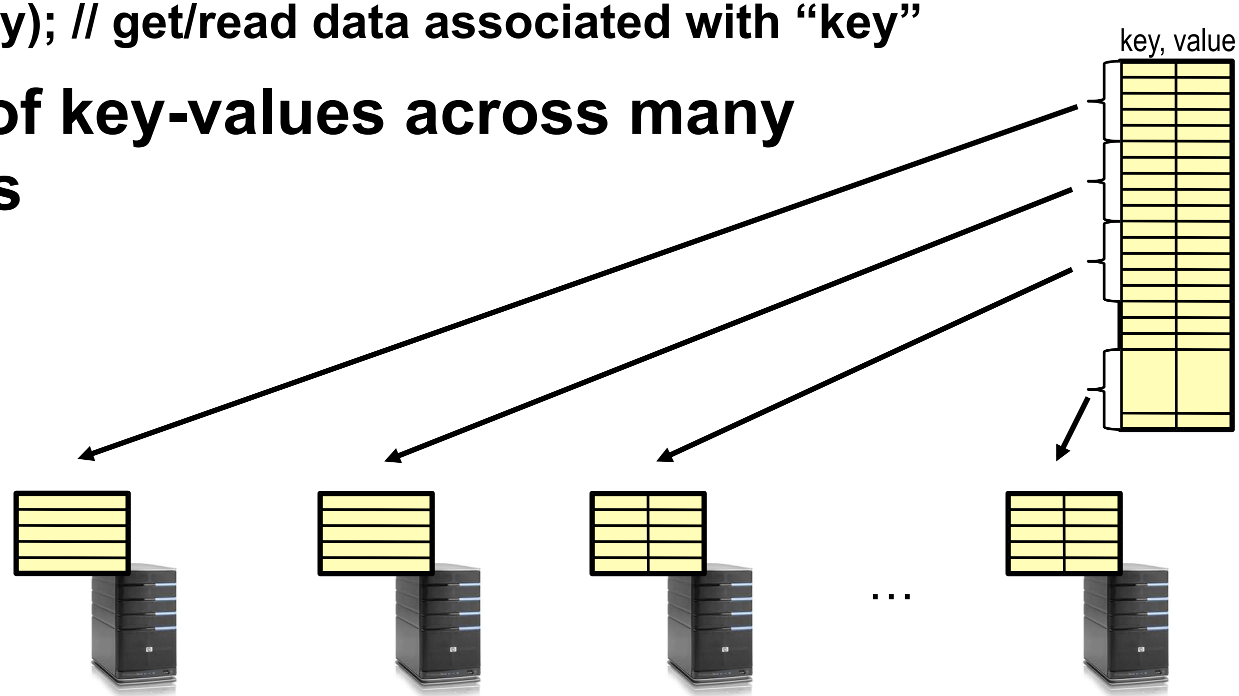
Naming

- **Naming choices:**
  - ***Hostname:*****localname: Name files explicitly**
    - » **No location or migration transparency**
  - ***Mounting* of remote file systems**
    - » **System manager mounts remote file system by giving name and local mount point**
    - » **Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server**
  - ***A single, global name space:* every file in the world has unique name**
    - » **Location Transparency: servers can change and files can move without involving user**

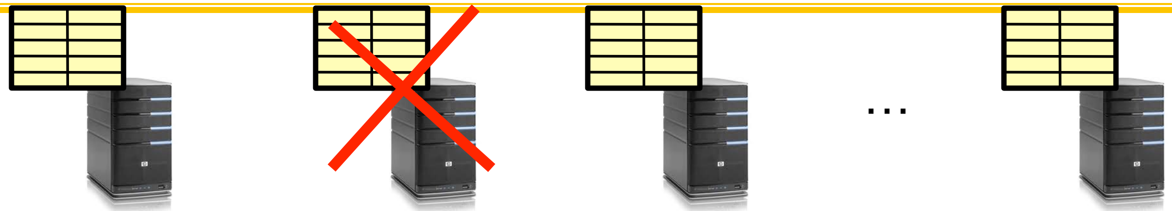


Key Value Store

- Handle huge volumes of data, e.g., PBs
  - Store (key, value) tuples
  - Used sometimes as a simpler but more scalable "database"
  - Also called Distributed Hash Tables (DHT)
- Simple interface
  - put(key, value); // insert/write "value" associated with "key"
  - value = get(key); // get/read data associated with "key"
- partition set of key-values across many machines

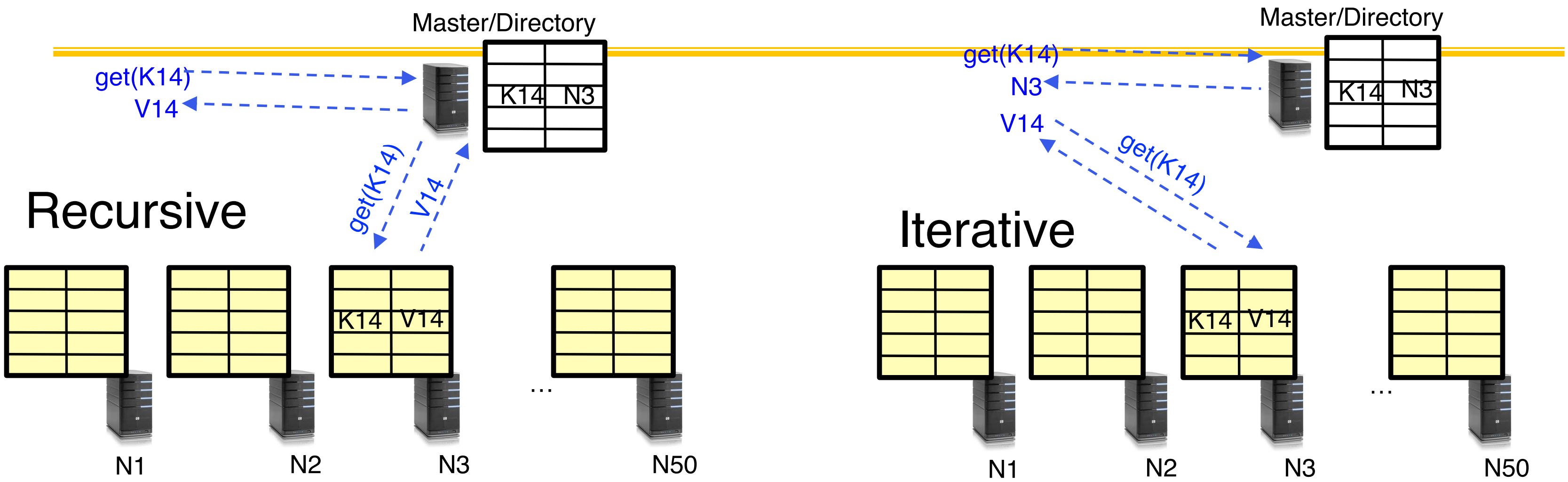


Challenges



- **Fault Tolerance: handle machine failures without losing data and without degradation in performance**
- **Scalability:**
  - **Need to scale to thousands of machines**
  - **Need to allow easy addition of new machines**
- **Consistency: maintain data consistency in face of node failures and message losses**
- **Heterogeneity (if deployed as peer-to-peer systems):**
  - **Latency: 1ms to 1000ms**
  - **Bandwidth: 32Kb/s to 100Mb/s**

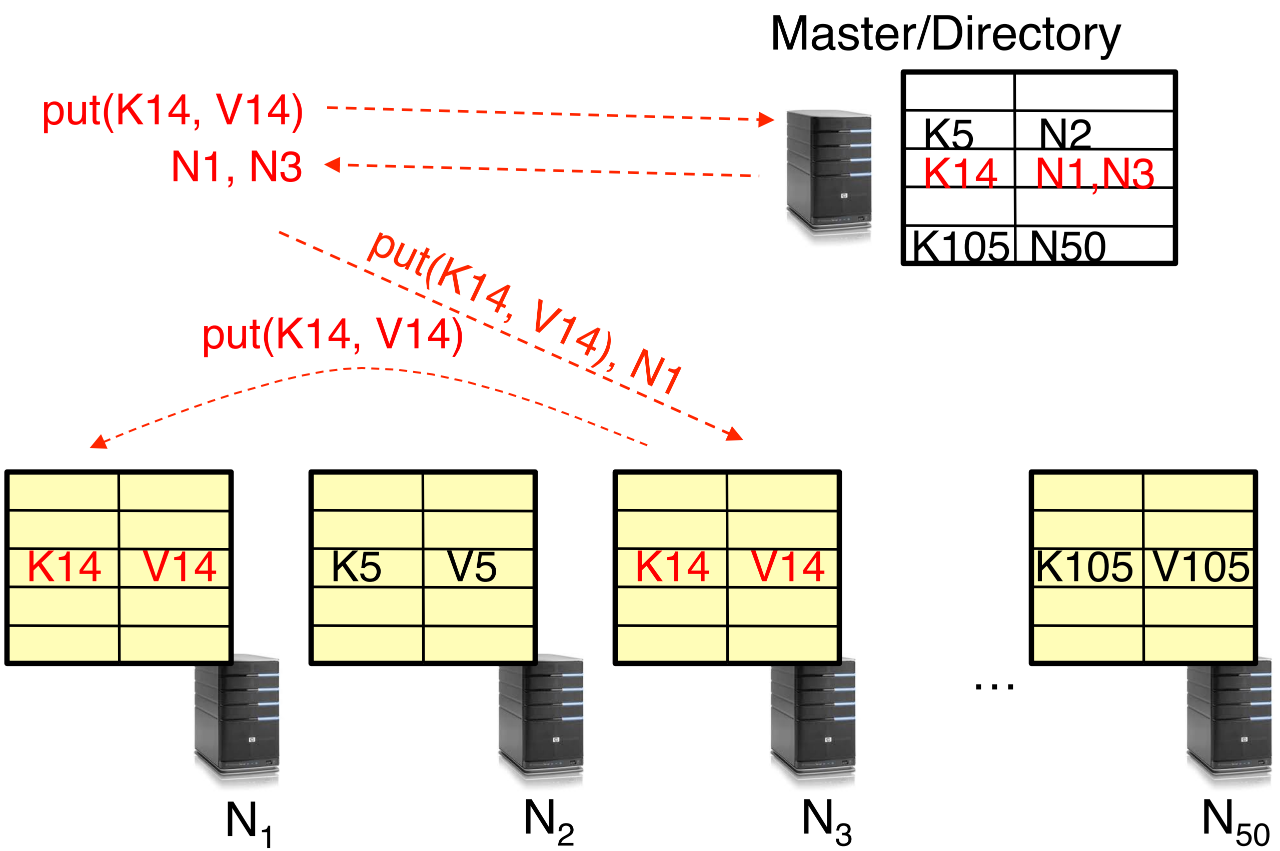
Discussion: Iterative vs. Recursive Query



- **Recursive Query:**
  - **Advantages:**
    - **Faster, as typically master/directory closer to nodes**
    - **Easier to maintain consistency, as master/directory can serialize puts()/gets()**
  - **Disadvantages: scalability bottleneck, as all "Values" go through master/directory**
- **Iterative Query**
  - **Advantages: more scalable**
  - **Disadvantages: slower, harder to enforce data consistency**

Fault Tolerance

- **Replicate value on several nodes**
- **Usually, place replicas on different racks in a datacenter to guard against rack failures**



Two Phase (2PC) Commit

- **2PC is a distributed protocol**
- **High-level problem statement**
  - If no node fails and all nodes are ready to commit, then all nodes COMMIT
  - Otherwise ABORT at all nodes
- **Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)**

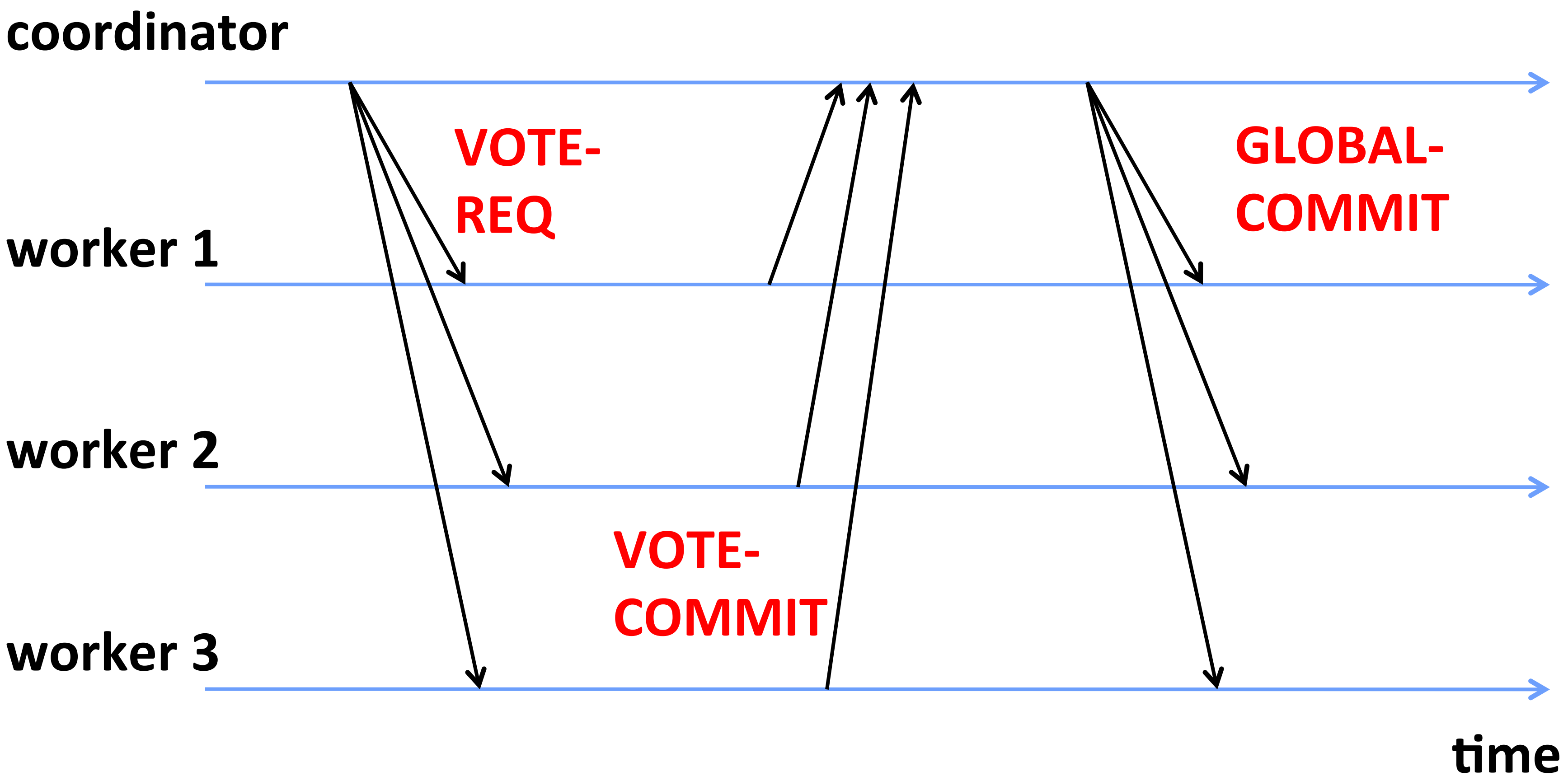
2PC Algorithm

- One coordinator
- N workers (replicas)

- High level algorithm description
  - Coordinator asks all workers if they can commit
  - If all workers reply "VOTE-COMMIT", then coordinator broadcasts "GLOBAL-COMMIT",

    Otherwise coordinator broadcasts "GLOBAL-ABORT"
  - Workers obey the GLOBAL messages

Failure Free Example Execution



Detailed Algorithm

Coordinator Algorithm

Coordinator sends VOTE-REQ to all workers

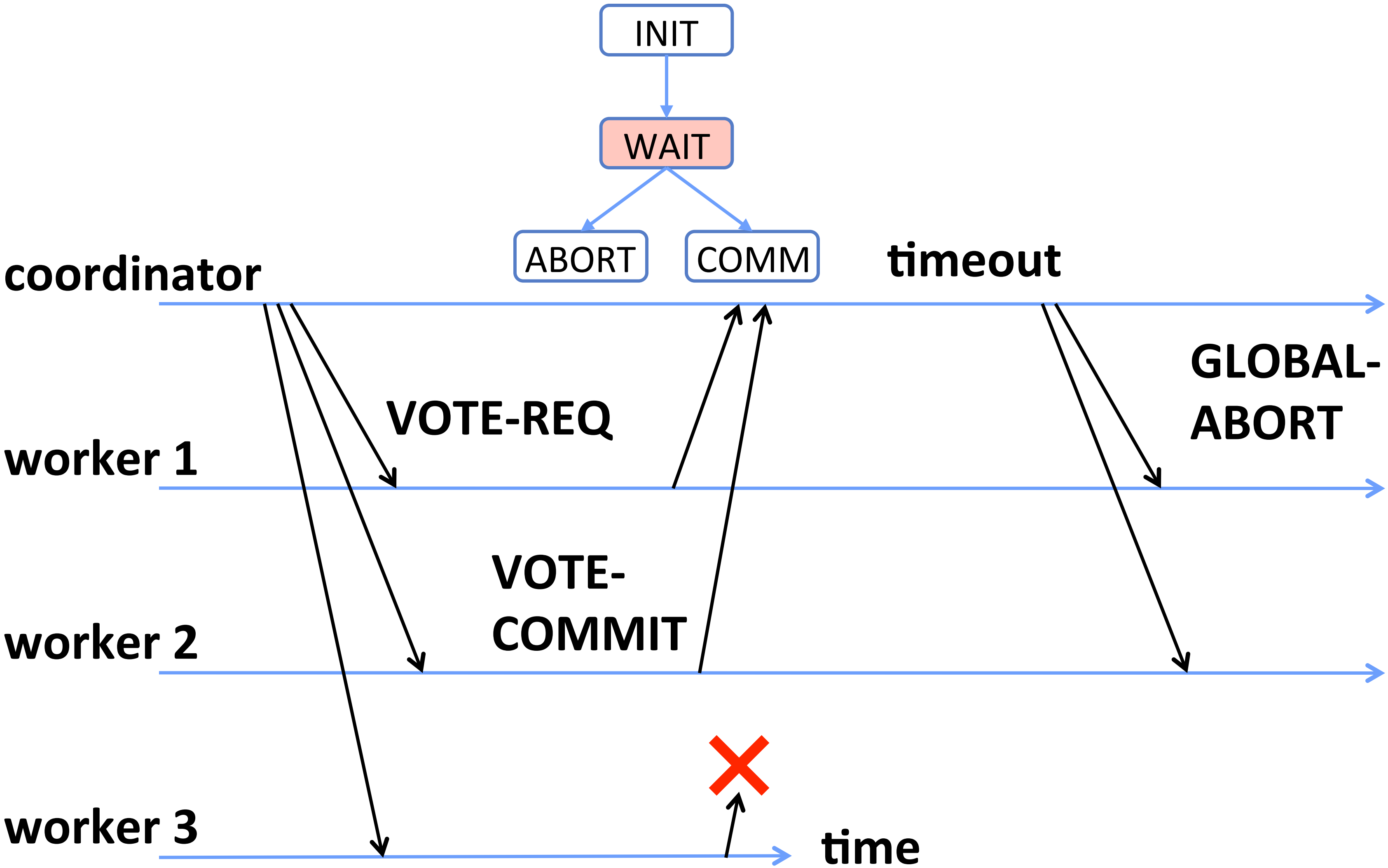
- If receive VOTE-COMMIT from all N workers, send GLOBAL-COMMIT to all workers
- If doesn't receive VOTE-COMMIT from all N workers, send GLOBAL-ABORT to all workers

Worker Algorithm

- Wait for VOTE-REQ from coordinator
- If ready, send VOTE-COMMIT to coordinator
- If not ready, send VOTE-ABORT to coordinator
  - And immediately abort

- If receive GLOBAL-COMMIT then commit
- If receive GLOBAL-ABORT then abort

Example of Worker Failure



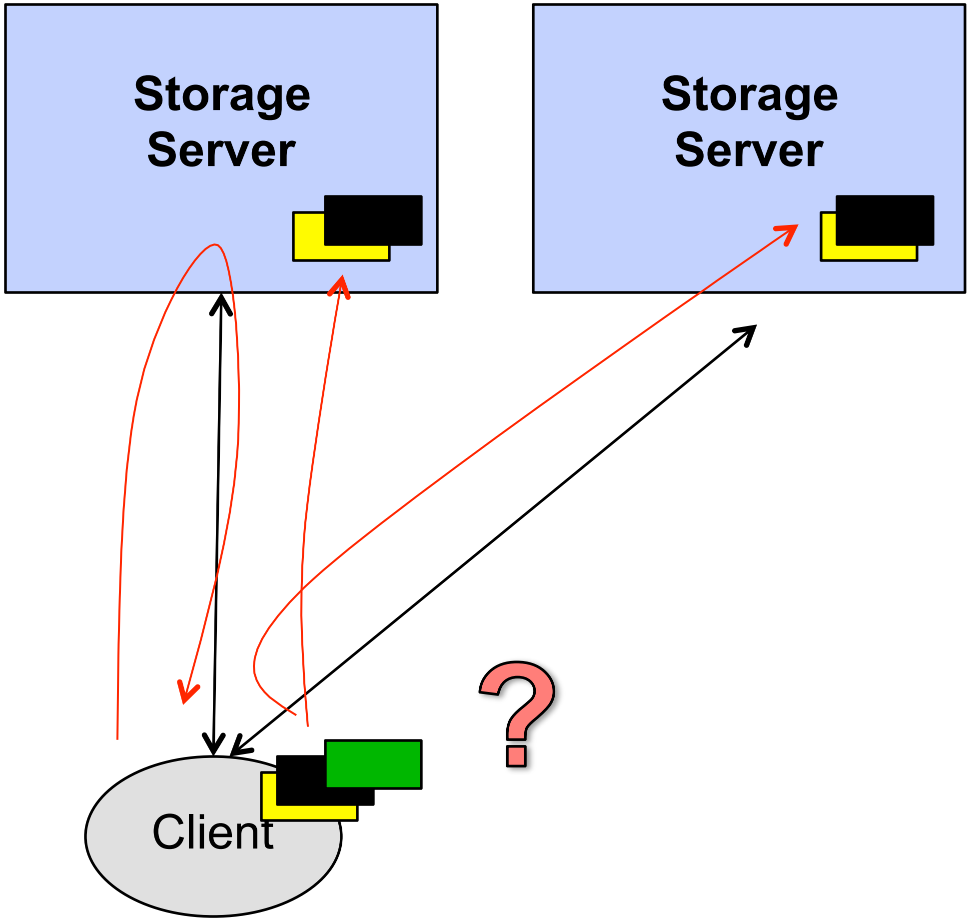
Durability

- All nodes use stable storage* to store which state they are in

- Upon recovery, it can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker asks Coordinator in READY

* - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.

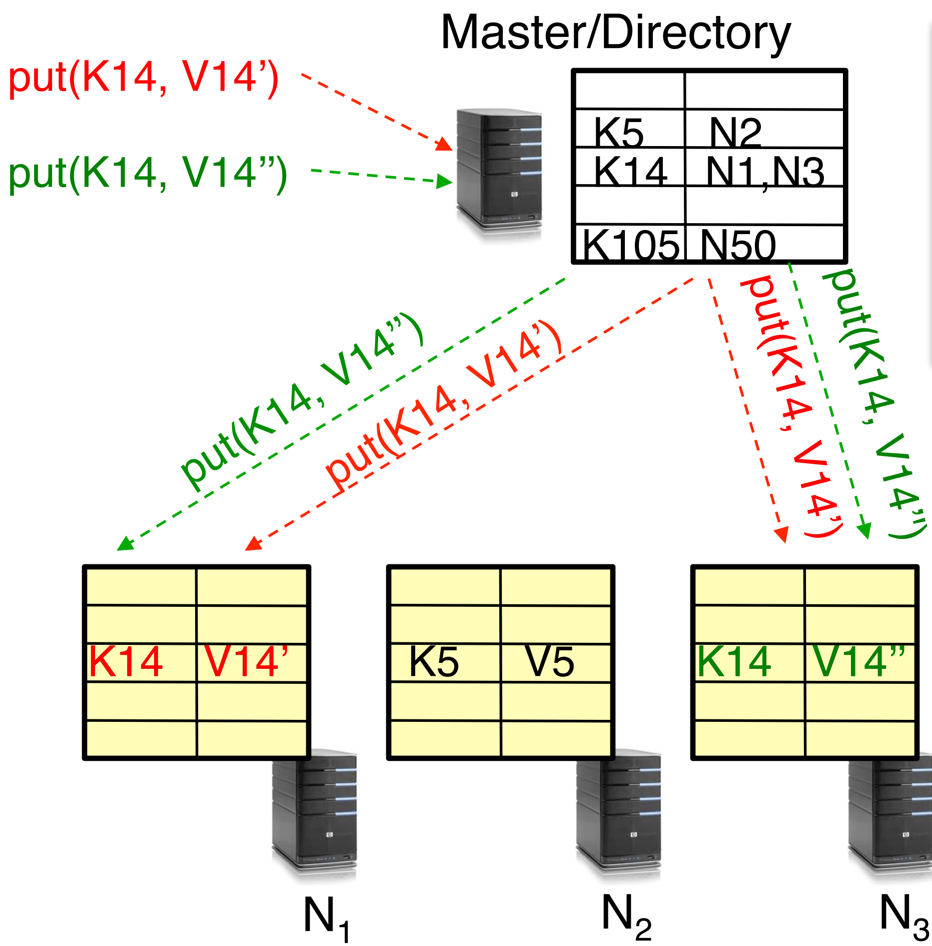
Multiple Servers



- What happens if cannot update all the replicas?
- Availability => Inconsistency

Consistency (cont'd)

- **If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order**



- put(K14, V14') and put(K14, V14'') reach N1 and N3 in reverse order
- What does get(K14) return?
  - Undefined!

Consistency

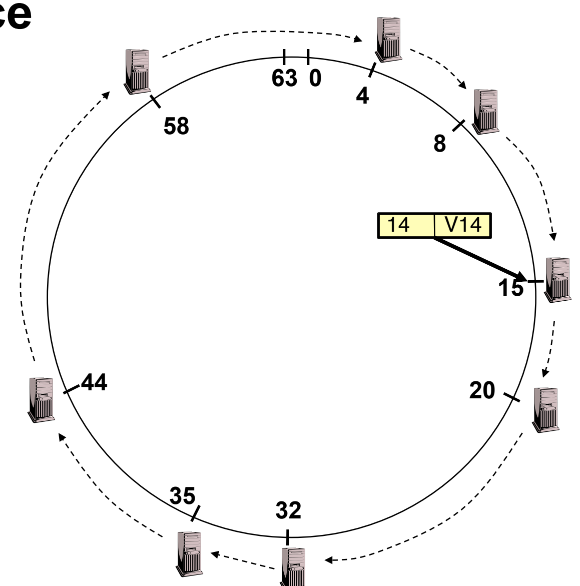
- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  – Wait for acknowledgements from every node
- What happens if a node fails during replication?
  – Pick another node and try again
- What happens if a node is slow?
  – Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  – Slow puts and fast gets

Consistency (cont'd)

- **Large variety of consistency models:**
  - Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    - » Think "one updated at a time"
    - » Transactions
  - Eventual consistency: given enough time all updates will propagate through the system
    - » One of the weakest form of consistency; used by many systems in practice
  - And many others: causal consistency, sequential consistency, strong consistency, …

Scaling Up Directory

- **Challenge:**
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!
- **Solution: consistent hashing**
- **Associate to each node a unique *id* in an *uni-*dimensional space $0..2^m-1$**
  - Partition this space across *m* machines
  - Assume keys are in same uni-dimensional space
  - Each (Key, Value) is stored at the node with the smallest ID larger than Key



The Data Center as a System

- **Clusters became THE architecture for large scale internet services**
  - Distribute disks, files, I/O, net, and compute over everything
  - Massive AND Incremental scalability
- **Search Engines the initial “Killer App”**
- **Multiple components as Cluster Apps**
  - Web crawl, Index, Search & Rank, Network, …
- **Global Layer as a Master/Worker pattern**
  - GFS, HDFS
- **Map Reduce framework address core of search on massive scale – and much more**
  - Indexing, log analysis, data querying
  - Collating, inverted indexes : map(k,v) => f(k,v),(k,v)
  - Filtering, Parsing, Validation
  - Sorting

Giant-Scale Services

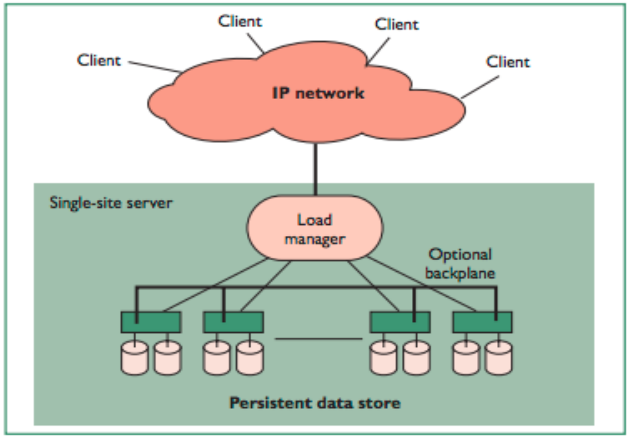
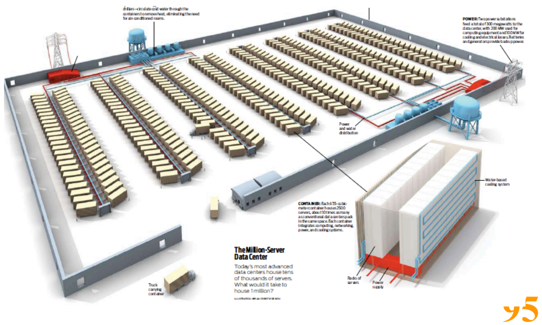


Figure 1. The basic model for giant-scale services. Clients connect via the Internet and then go through a load manager that hides down nodes and balances traffic.



Lessons from Giant-Scale Services, Eric Brewer, IEEE Computer, Jul 2001

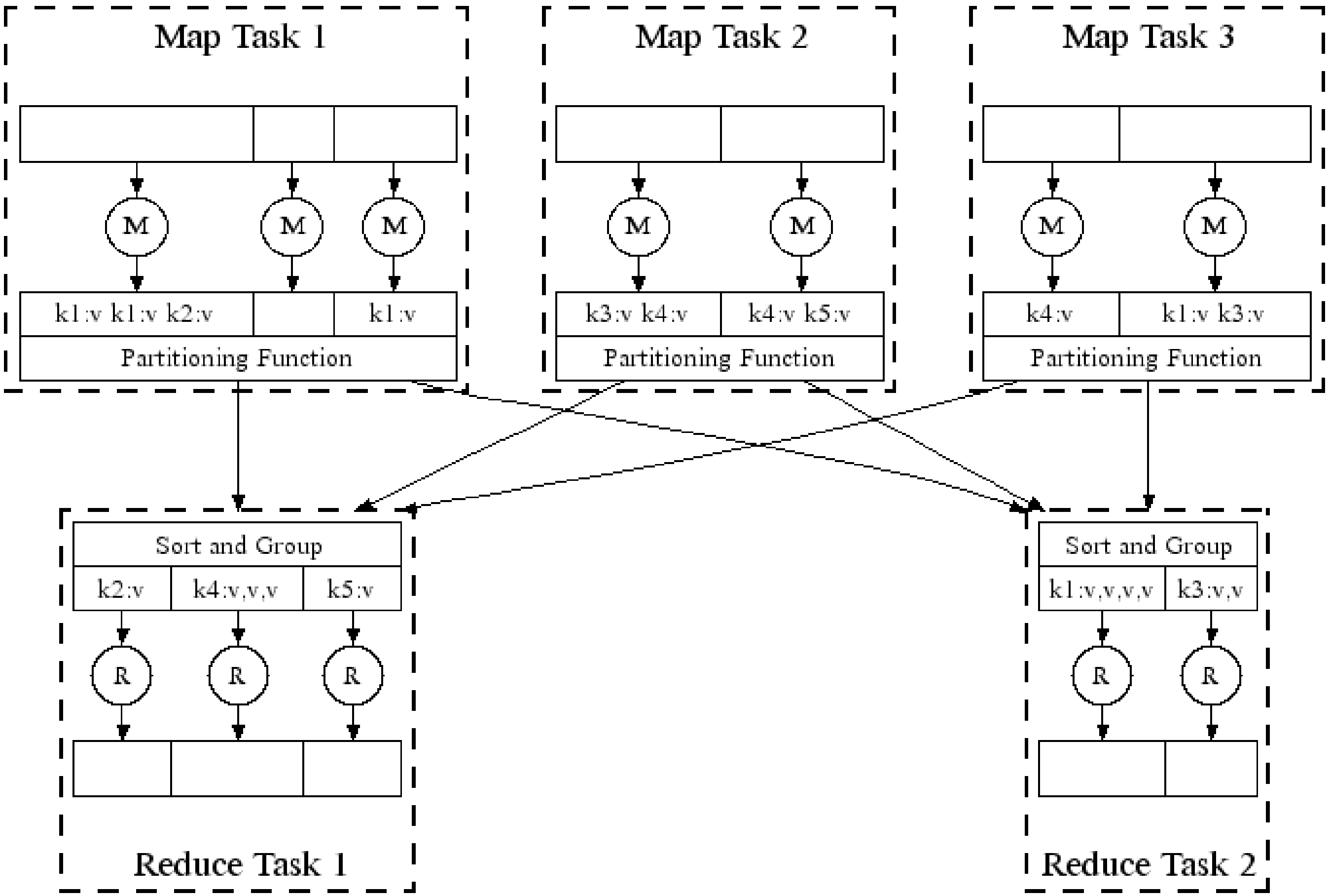
GFS/HDFS Insights

- *Petabyte* storage
  - Files split into large blocks (128 MB) and replicated across many nodes
  - Big blocks allow high throughput sequential reads/writes
- Data *striped* on hundreds/thousands of servers
  - Scan 100 TB on 1 node @ 50 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes
- *Failures* will be the norm
  - Mean time between failures for 1 node = 3 years
  - Mean time between failures for 1000 nodes = 1 day
- Use *commodity* hardware
  - Failures are the norm anyway, buy cheaper hardware
- No complicated consistency models
  - Single writer, append-only data

MapReduce Insights

- **Restricted key-value model**
  - Same fine-grained operation (Map & Reduce) repeated on huge, distributed (within DC) data
  - Operations must be deterministic
  - Operations must be idempotent/no side effects
  - Only communication is through the shuffle
  - Operation (Map & Reduce) output saved (on disk)

MapReduce Parallel Execution



Shamelessly stolen from Jeff Dean's OSDI '04 presentation
http://labs.google.com/papers/mapreduce-osdi04-slides/index.html

MapReduce Pros

- **Distribution is completely transparent**
  - Not a single line of distributed programming (ease, correctness)
- **Automatic fault-tolerance**
  - Determinism enables running failed tasks somewhere else again
  - Saved intermediate data enables just re-running failed reducers
- **Automatic scaling**
  - As operations as side-effect free, they can be distributed to any number of machines dynamically
- **Automatic load-balancing**
  - Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers*)

MapReduce Cons

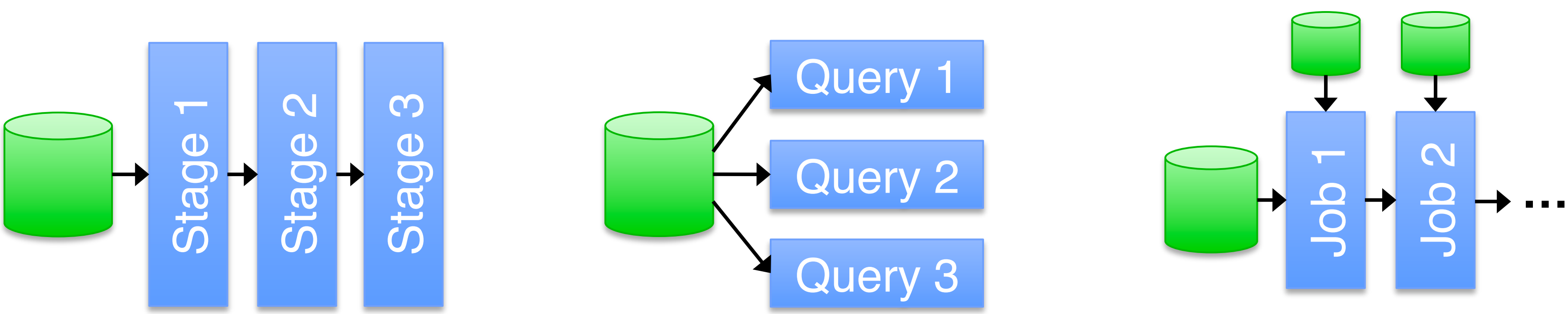
- **Restricted programming model**
  - Not always natural to express problems in this model
  - Low-level coding necessary
  - Little support for iterative jobs (lots of disk access)
  - High-latency (batch processing)

- **Addressed by follow-up research and Apache projects**
  - Pig and Hive for high-level coding
  - Spark for iterative and low-latency jobs

UCB / Apache Spark Motivation

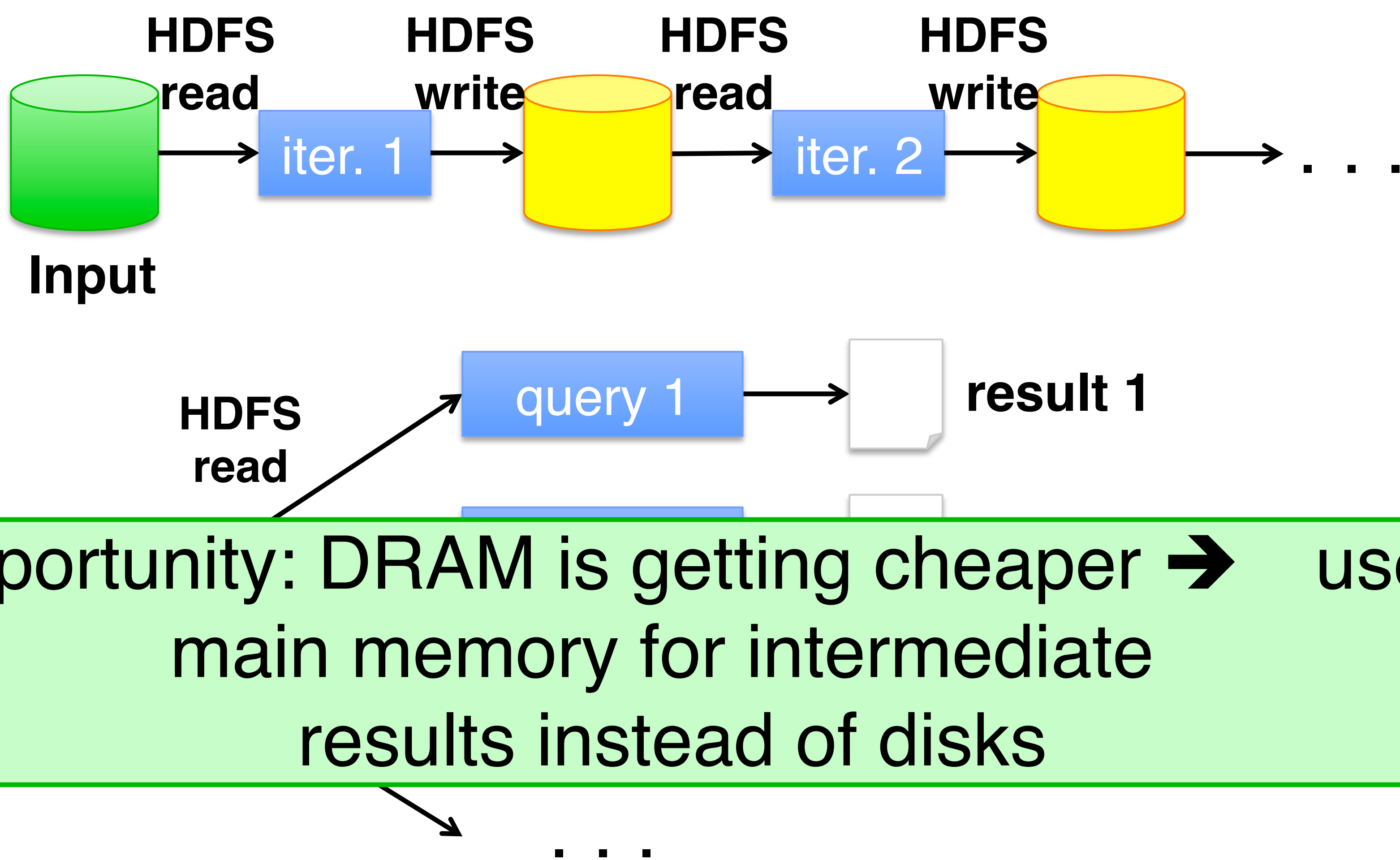
Complex jobs, interactive queries and online processing all need one thing that MR lacks:

Efficient primitives for data sharing



Iterative job **Interactive mining** **Stream processing**

Examples



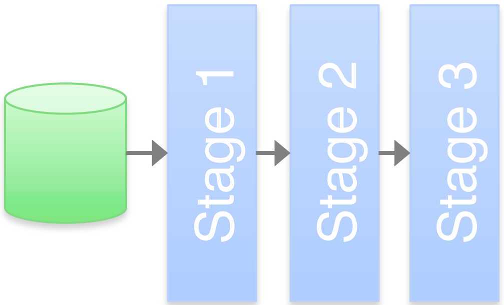
Opportunity: DRAM is getting cheaper → use main memory for intermediate results instead of disks

Spark Motivation

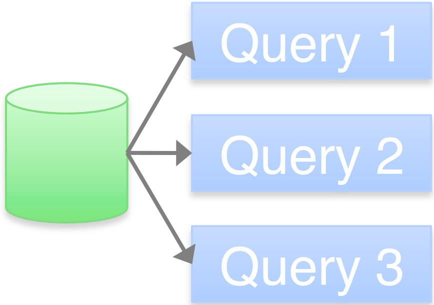
Complex jobs, interactive queries and online processing all need one thing that MR lacks:

Efficient primitives for data sharing

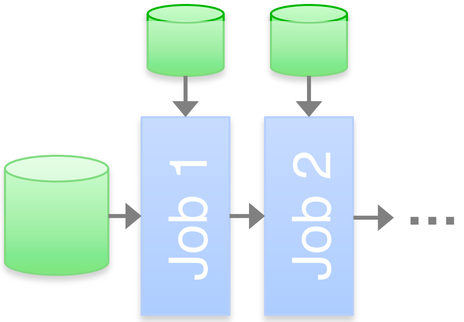
Problem: in MR, the only way to share data across jobs is using stable storage (e.g. file system) ➔ slow!



Iterative job



Interactive mining



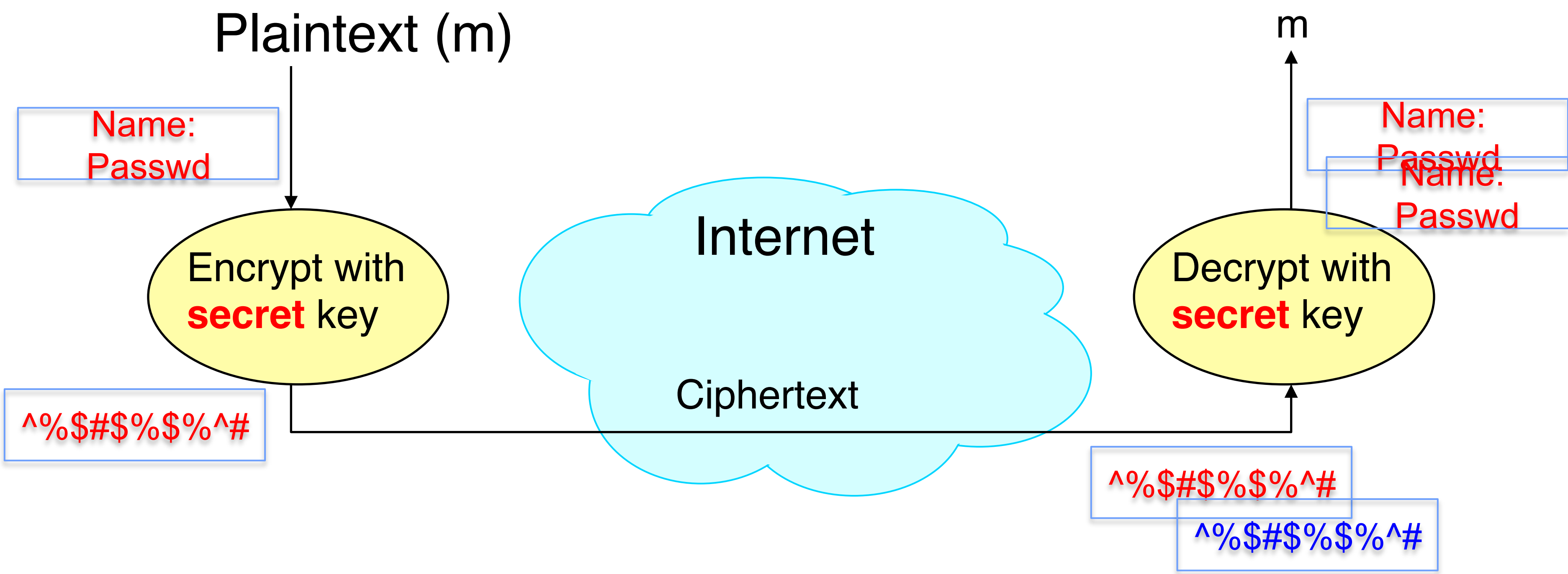
Stream processing

Security Requirements

- **Authentication**
  - Ensures that a user is who is claiming to be
- **Data integrity**
  - Ensure that data is not changed from source to destination or after being written on a storage device
- **Confidentiality**
  - Ensures that data is read only by authorized users
- **Non-repudiation**
  - Sender/client can’t later claim didn’t send/write data
  - Receiver/server can’t claim didn’t receive/write data

Using Symmetric Keys

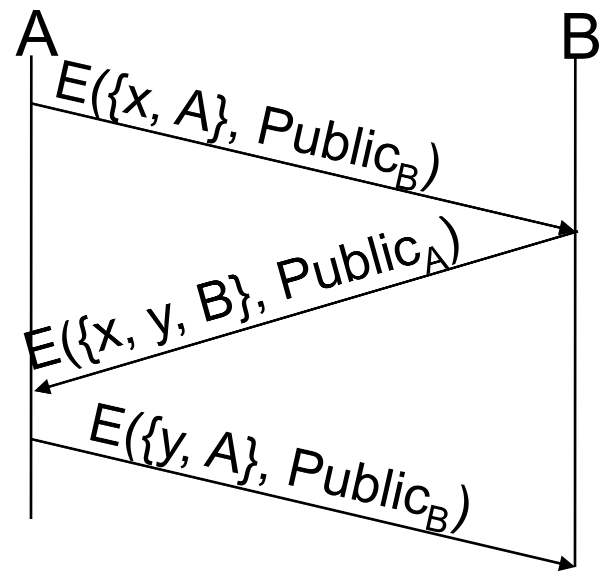
- **Same key for encryption and decryption**
- **Achieves confidentiality**
- ***Vulnerable to tampering and replay attacks***



Need integrity check and unique sequence number

Simple Public Key Authentication

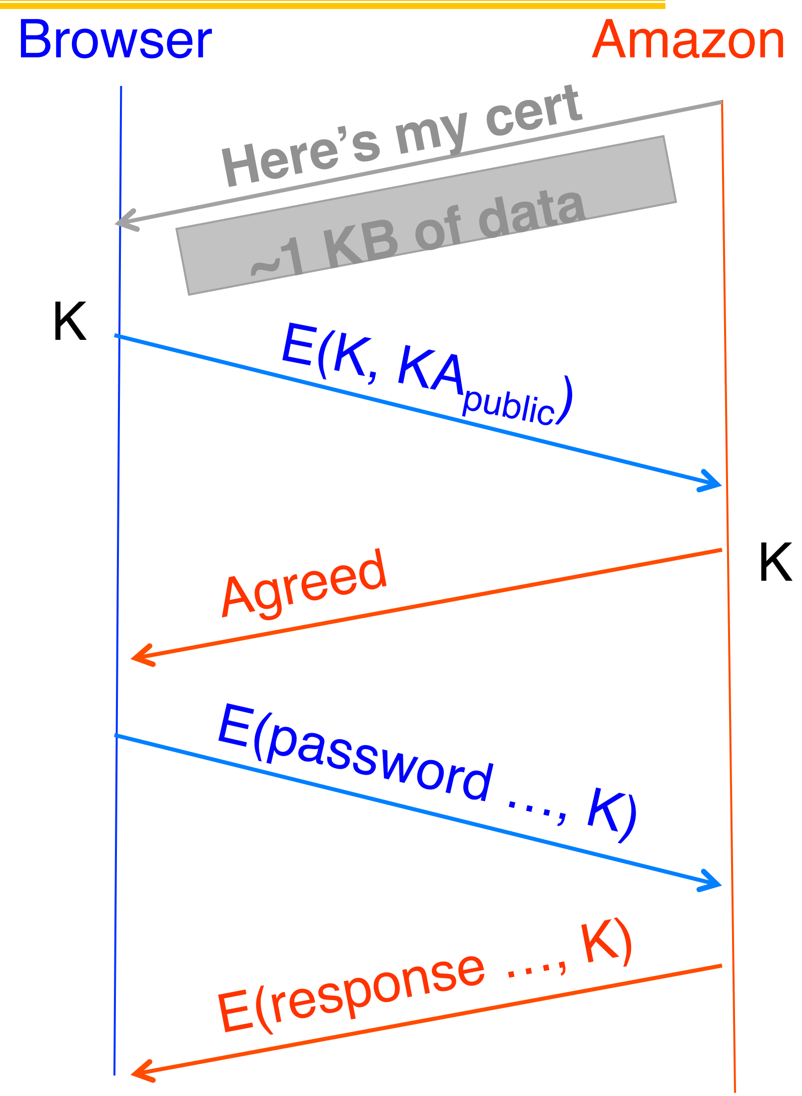
- **Each side need only know the other side's public key**
  - **No secret key need be shared**
- **A encrypts a nonce (random num.) *x***
  - **Avoid replay attacks, e.g., attacker impersonating client or server**
- **B proves it can recover *x***
- **A can authenticate itself to B in the same way with nonce, *y***
- ***Many more details to make this work securely in practice!***



Notation: E(m,k) – encrypt message m with key k

HTTPS Connection (SSL/TLS) cont'd

- **Browser constructs a random *session key* K used for data communication**
  - **Private key for bulk crypto**
- **Browser encrypts K using Amazon's public key**
- **Browser sends E(K, KA_{public}) to server**
- **Browser displays** 
- **All subsequent comm. encrypted w/ symmetric cipher (e.g., AES128) using key K**
  - **E.g., client can authenticate using a password**



Security & Privacy in a Pervasive Web