15-440 Distributed Systems

Midterm Review on 10/16/18

Midterm on 10/18/18 10.25am - 11.55am

15-440 Midterm Topics Overview

1) Distribution Systems Intro (Y)
2) Communication - Internet in a Day (D)
3) Classical Consistency/Synchronization (Y)
4) Time Synchronization (D)
5) Remote Procedure Calls (Y)
6) Distributed Filesystems (Y)
7) Distributed Mutual Exclusion (D)
8) Distributed Concurrency Control (Y)
9) Logging and Crash Recovery (Y)
10) Distributed Replication (Y)
11) Fault Tolerance & RAID (D)
12) Distributed Databases Case Study (D)

15-440 Distributed Systems

Distribution Systems Intro

What Is A Distributed System?

"A collection of independent computers that appears to its users as a single coherent system."

- Features:
  - No shared memory – message-based communication
  - Each runs its own local OS
  - Heterogeneity
  - Expandability
- Ideal: to present a single-system image:
  - The distributed system "looks like" a single computer rather than a collection of separate computers.

Definition of a Distributed System

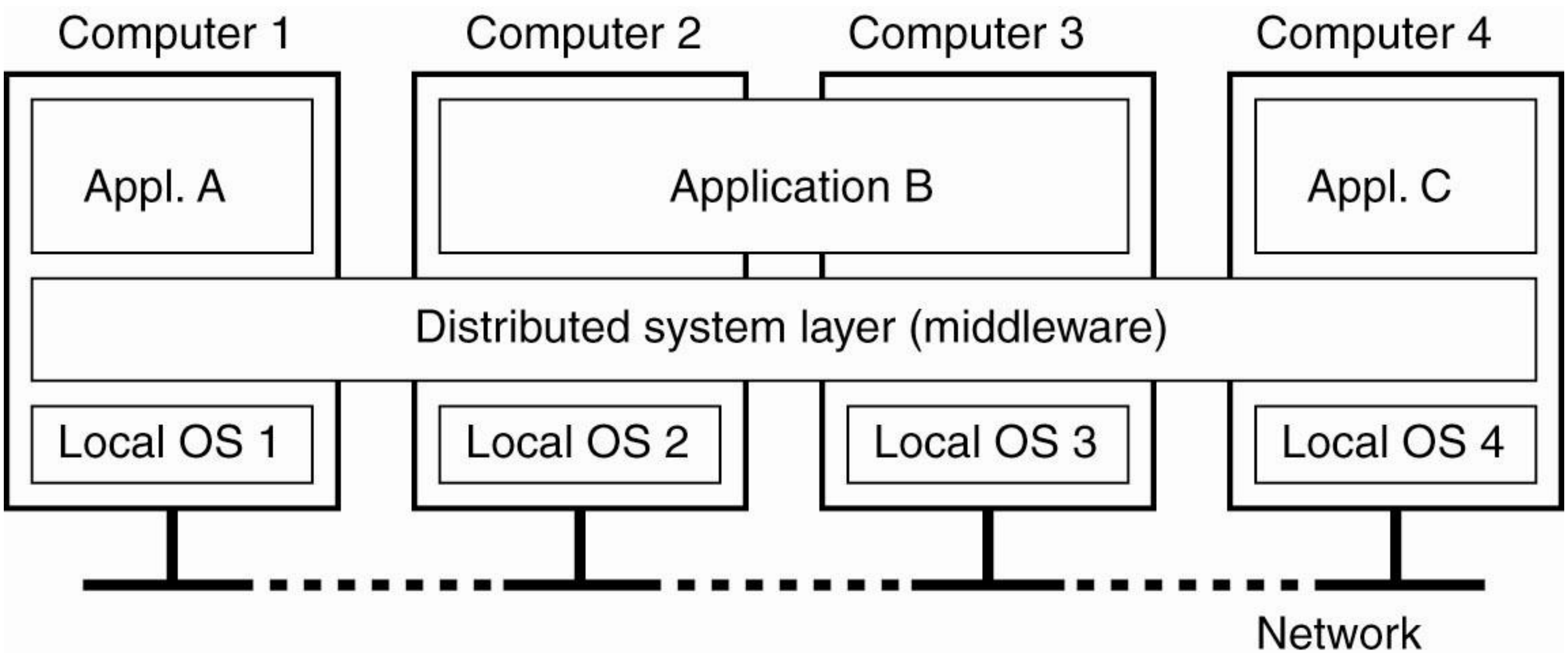


Figure 1-1. A distributed system organized as middleware. The middleware layer runs on all machines, and offers a uniform interface to the system

Distributed Systems: Goals

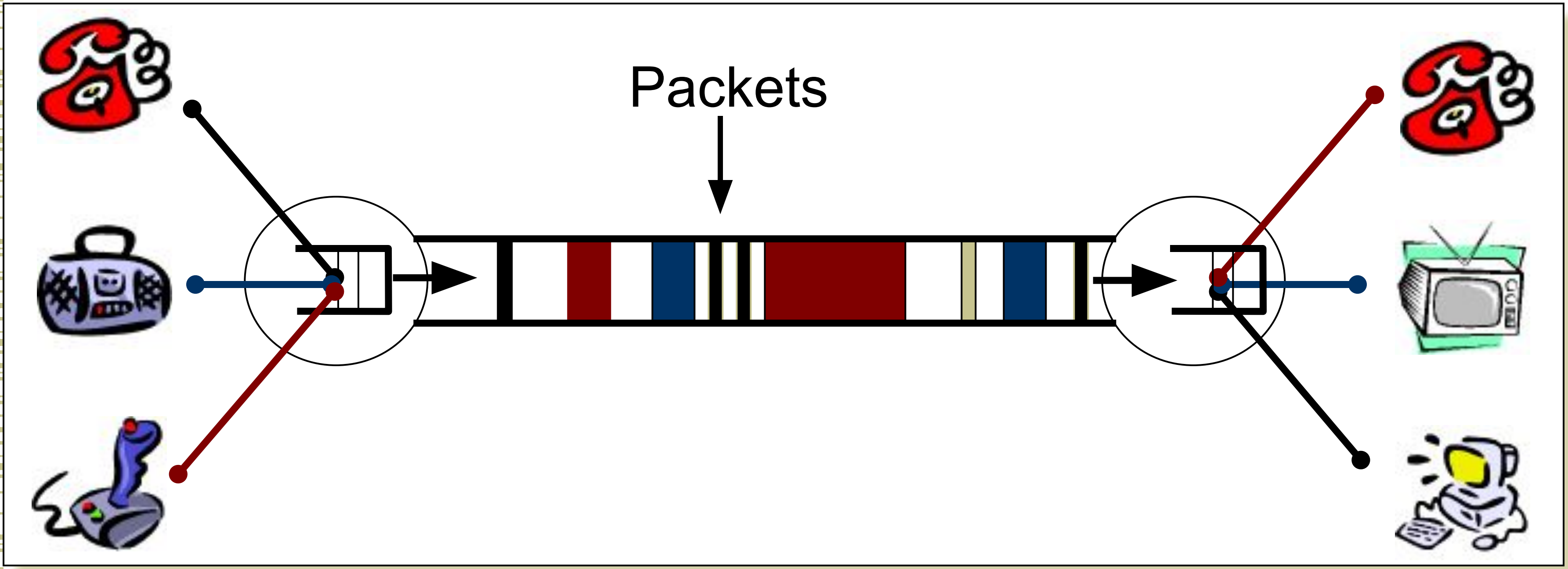
- Resource Availability: remote access to resources
- Distribution Transparency: single system image
  - Access, Location, Migration, Replication, Failure,…
- Openness: services according to standards (RPC)
- Scalability: size, geographic, admin domains, …

- Example of a Distributed System?
  - Web search on google
  - DNS: decentralized, scalable, robust to failures, ...
  - ...

15-440 Distributed Systems

Communication - Internet in a Day

Packet Switching – Statistical Multiplexing



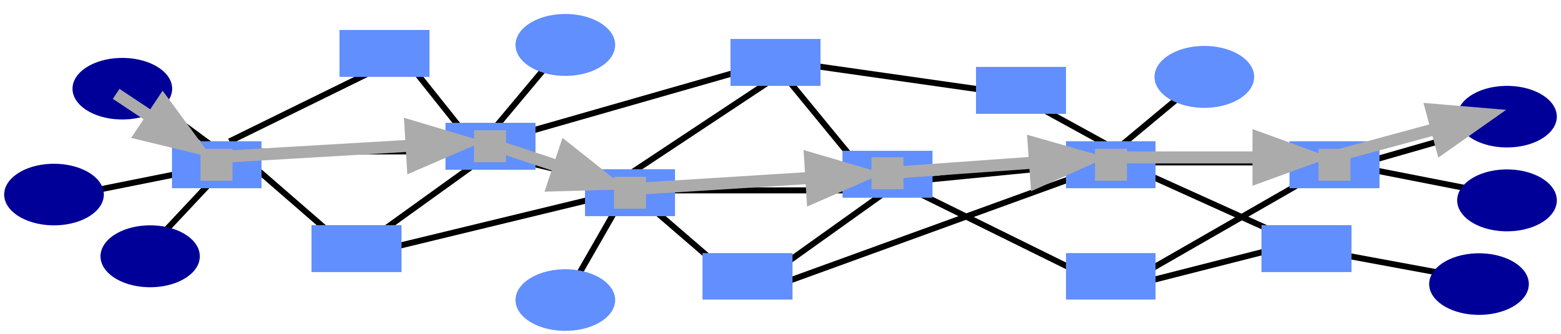
- Switches arbitrate between inputs
- Can send from *any* input that's ready
  - Links never idle when traffic to send
  - (Efficiency!)

Model of a communication channel

- Latency - how long does it take for the first bit to reach destination
- Capacity - how many bits/sec can we push through? (often termed “bandwidth”)
- Jitter - how much variation in latency?
- Loss / Reliability - can the channel drop packets?
- Reordering

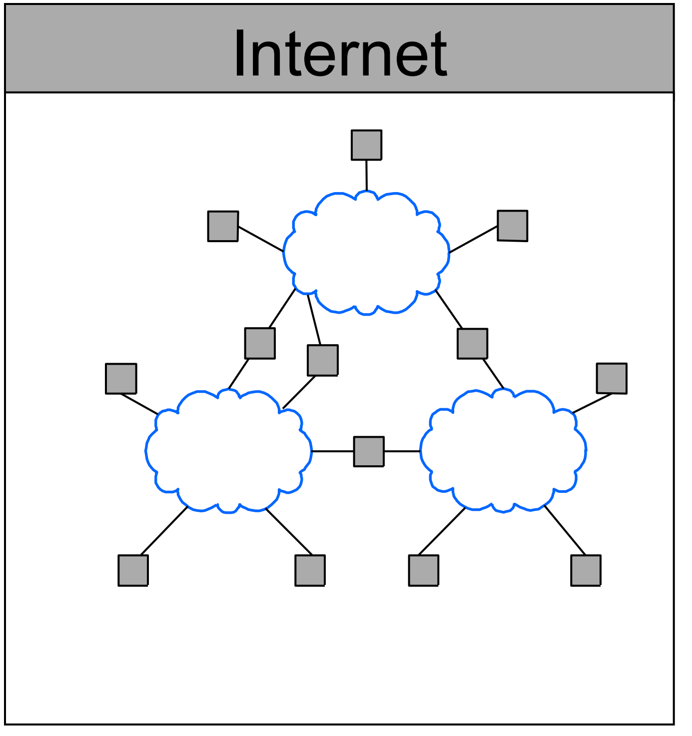
Packet Switching

- Source sends information as self-contained packets that have an address.
  - Source may have to break up single message in multiple
- Each packet travels independently to the destination host.
  - Switches use the address in the packet to determine how to forward the packets
  - Store and forward
- Analogy: a letter in surface mail.



Internet

- An inter-net: a network of networks.
  - Networks are connected using routers that support communication in a hierarchical fashion
  - Often need other special devices at the boundaries for security, accounting, ..
- The Internet: the interconnected set of networks of the Internet Service Providers (ISPs)
  - About 17,000 different networks make up the Internet



Network Service Model

- What is the *service model* for inter-network?
  - Defines what promises that the network gives for any transmission
  - Defines what type of failures to expect
- Ethernet/Internet: *best-effort* – packets can get lost, etc.

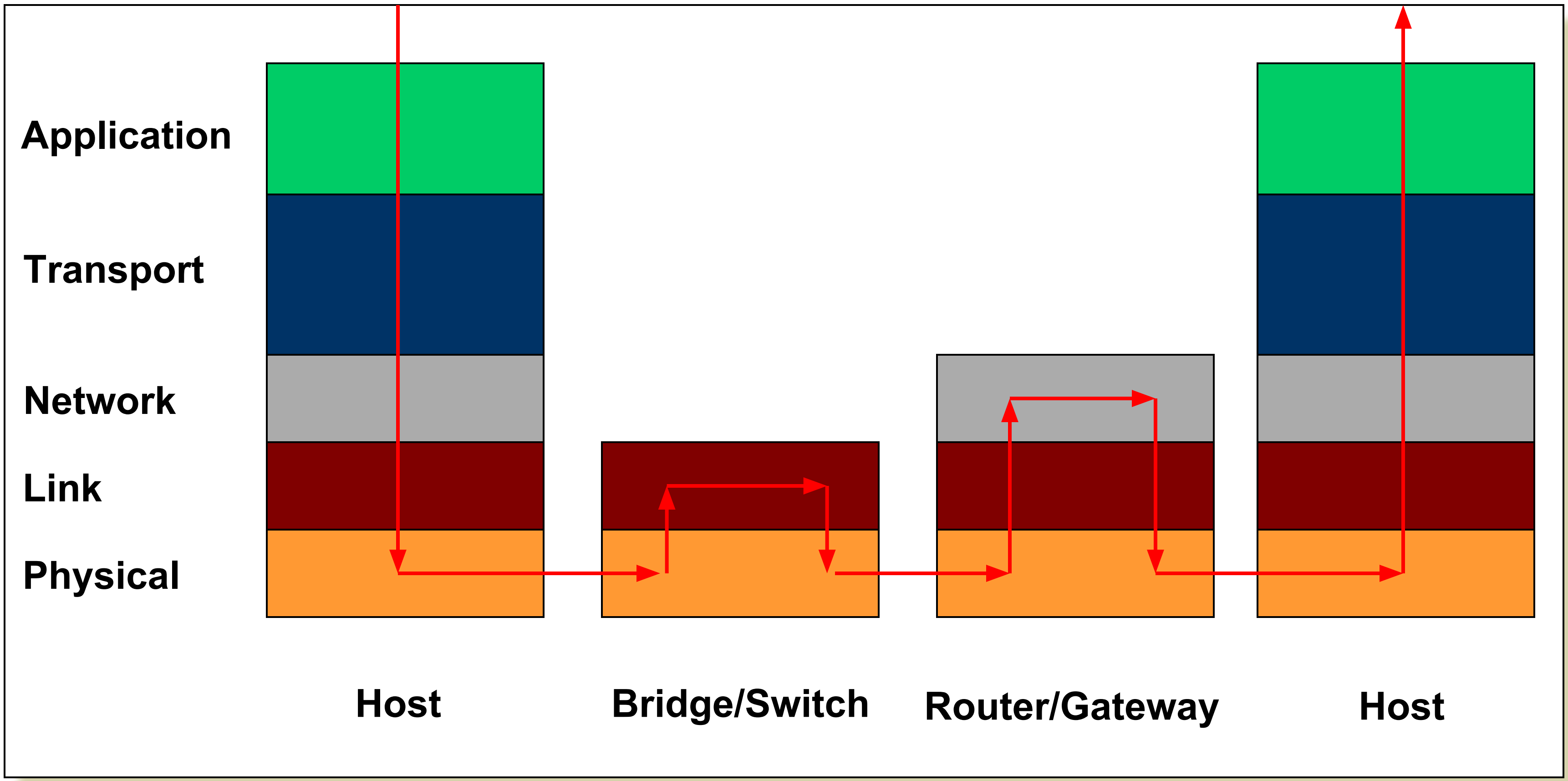
Possible Failure models

- Fail-stop:
  - When something goes wrong, the process stops / crashes / etc.
- Fail-slow or fail-stutter:
  - Performance may vary on failures as well
- Byzantine:
  - Anything that can go wrong, will.
  - Including malicious entities taking over your computers and making them do whatever they want.
- These models are useful for proving things;
- The real world typically has a bit of everything.

- Deciding which model to use is important!

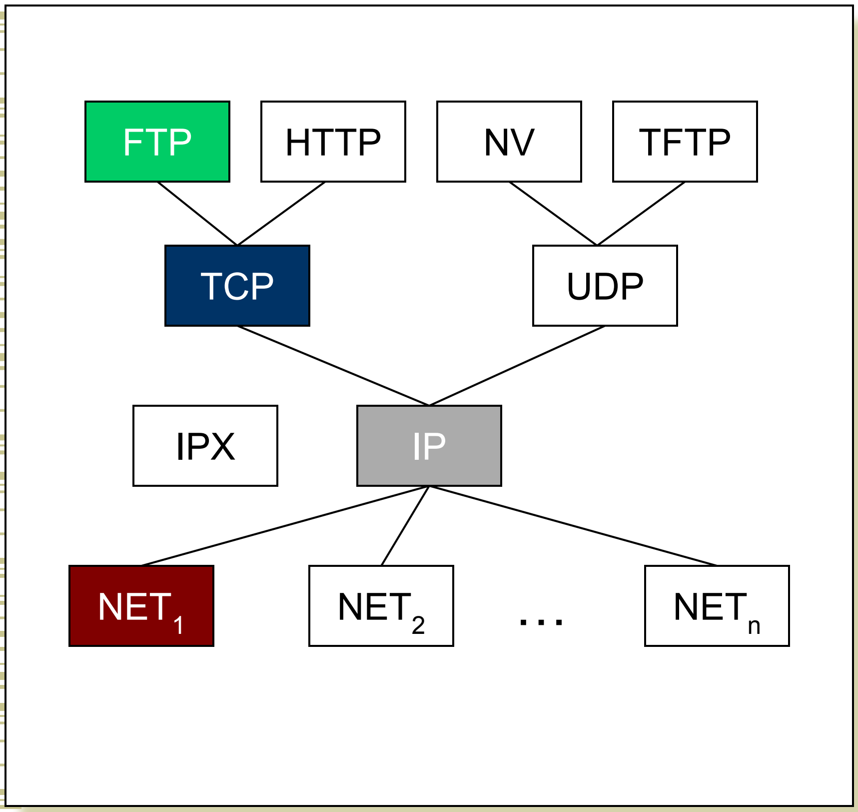
IP Layering

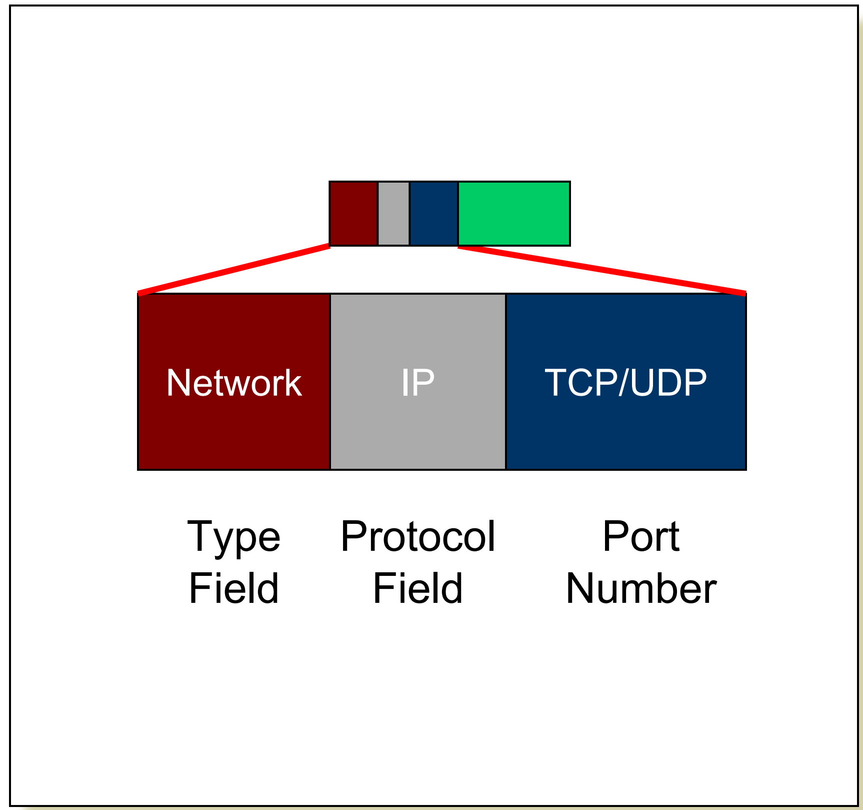
- Relatively simple



Protocol Demultiplexing

- Multiple choices at each layer





User Datagram Protocol (UDP): An Analogy

UDP	Postal Mail
• Single socket to receive messages • No guarantee of delivery • Not necessarily in-order delivery • Datagram – independent packets • Must address each packet	• Single mailbox to receive letters • Unreliable ☺ • Not necessarily in-order delivery • Letters sent independently • Must address each letter

Example UDP applications
Multimedia, voice over IP

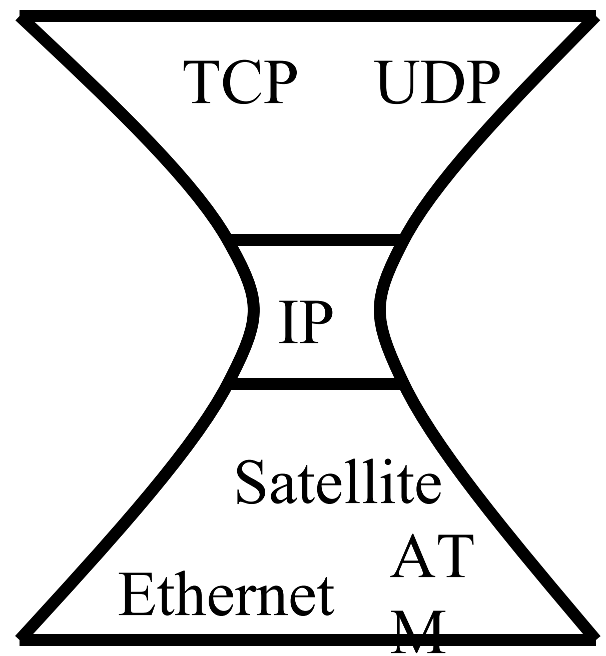
Transmission Control Protocol (TCP): An Analogy

TCP	Telephone Call
• Reliable – guarantee delivery • Byte stream – in-order delivery • Connection-oriented – single socket per connection • Setup connection followed by data transfer	• Guaranteed delivery • In-order delivery • Connection-oriented • Setup connection followed by conversation

Example TCP applications
Web, Email, Telnet

Summary: Internet Architecture

- Packet-switched datagram network
- IP is the “compatibility layer”
  - Hourglass architecture
  - All hosts and routers run IP
- Stateless architecture
  - no per flow state inside network



Summary: Minimalist Approach

- Dumb network
  - IP provide minimal functionalities to support connectivity
    - Addressing, forwarding, routing
- Smart end system
  - Transport layer or application performs more sophisticated functionalities
    - Flow control, error control, congestion control
- Advantages
  - Accommodate heterogeneous technologies (Ethernet, modem, satellite, wireless)
  - Support diverse applications (telnet, ftp, Web, X windows)
  - Decentralized network administration

15-440 Distributed Systems

Classical Consistency/Synchronization

Terminology

- **Critical Section:** piece of code accessing a shared resource, usually variables or data structures
- **Race Condition:** Multiple threads of execution enter CS at the same time, update shared resource, leading to undesirable outcome
- **Indeterminate Program:** One or more Race Conditions, output of program depending on ordering, non deterministic

Classic synchronization primitives

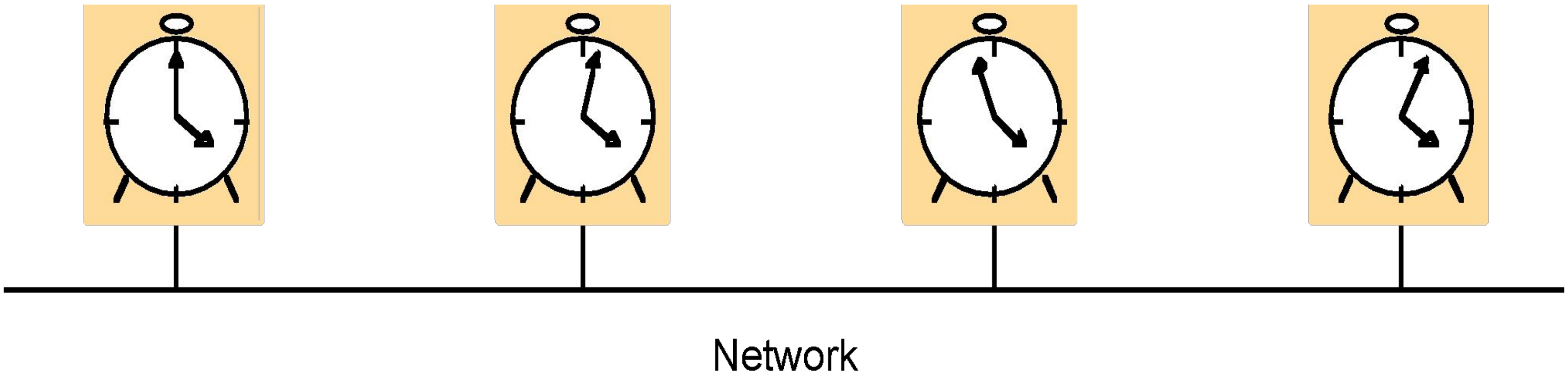
- Basics of concurrency
  - Correctness (achieves Mutex, no deadlock, no livelock)
  - Efficiency, no spinlocks or wasted resources
  - Fairness
- Synchronization mechanisms
  - Semaphores (P() and V() operations)
  - Mutex (binary semaphore)
  - Condition Variables (allows a thread to sleep)
    - Must be accompanied by a mutex
    - Wait and Signal operations

  - Work through examples again

15-440 Distributed Systems

Time Synchronization

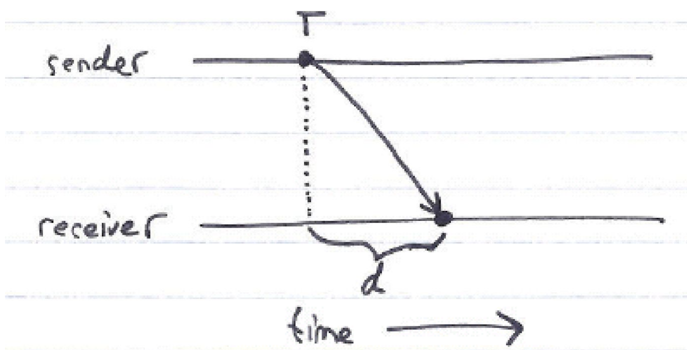
Clocks in a Distributed System



- Computer clocks are not generally in perfect agreement
  - **Skew**: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to clock drift (they count time at different rates)
  - **Clock drift rate**: the difference per unit of time from some ideal reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Perfect networks

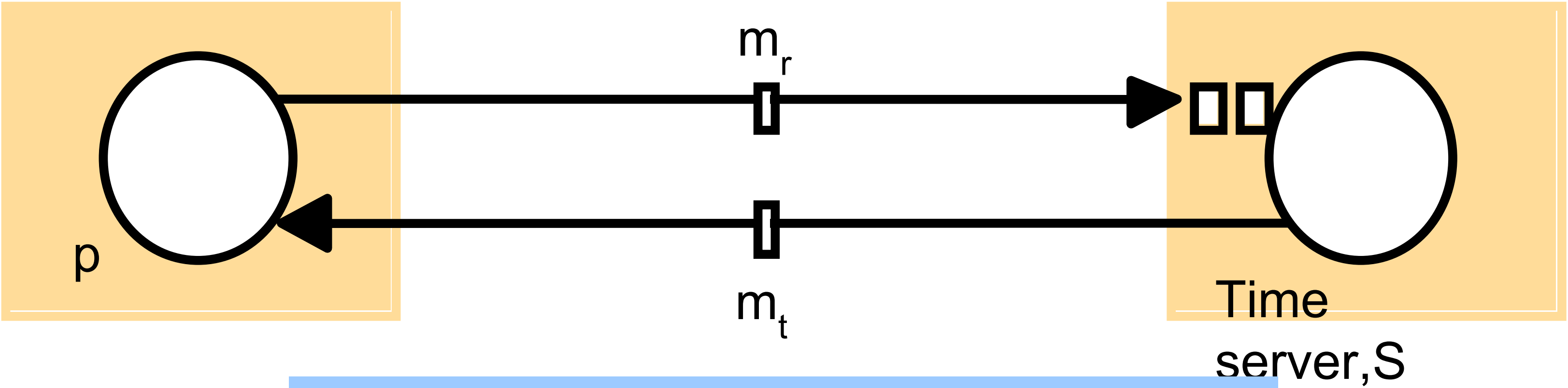
- Messages always arrive, with propagation delay exactly *d*



- Sender sends time *T* in a message
- Receiver sets clock to *T*+*d*
    - Synchronization is exact

Cristian's Time Sync

- A time server *S* receives signals from a UTC source
  - Process *p* requests time in m_r and receives *t* in m_t from *S*
  - *p* sets its clock to $t + RTT/2$
  - Accuracy ± ($RTT/2$ - *min*) :
    - because the earliest time *S* puts *t* in message m_t is *min* after *p* sent m_r.
    - the latest time was *min* before m_t arrived at *p*
    - the time by *S*'s clock when m_t arrives is in the range [t+*min*, $t + RTT$ - *min*]



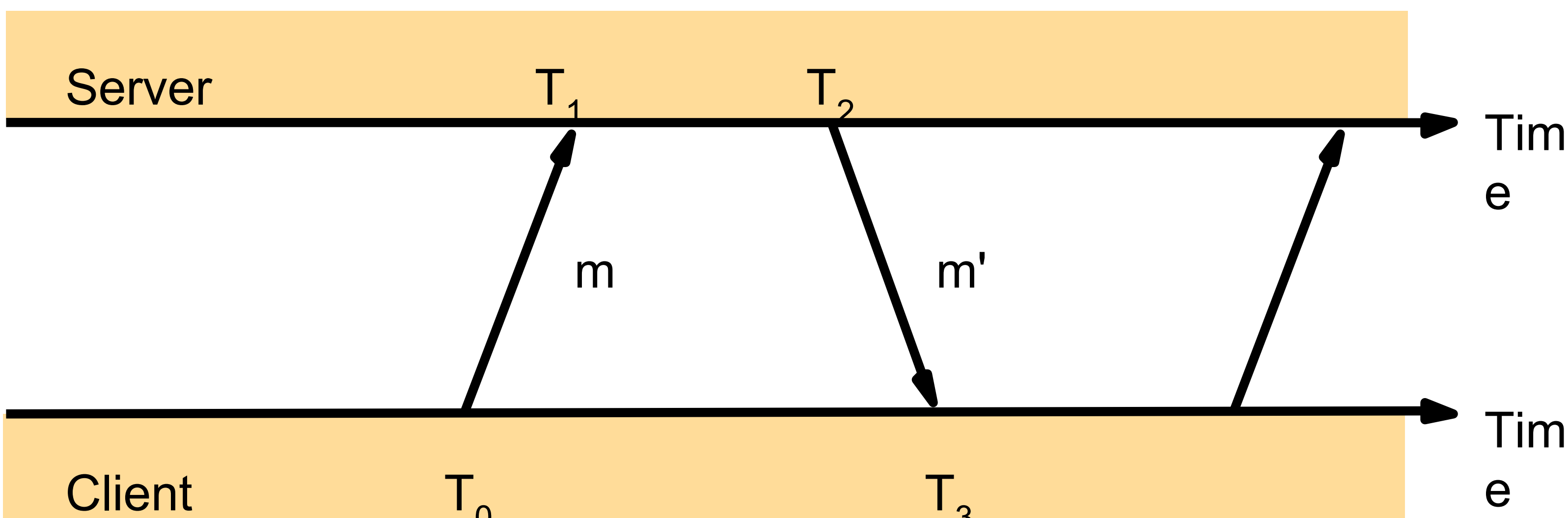
T_{round} is the round trip time recorded by *p*
min is an estimated minimum round trip time

Berkeley algorithm

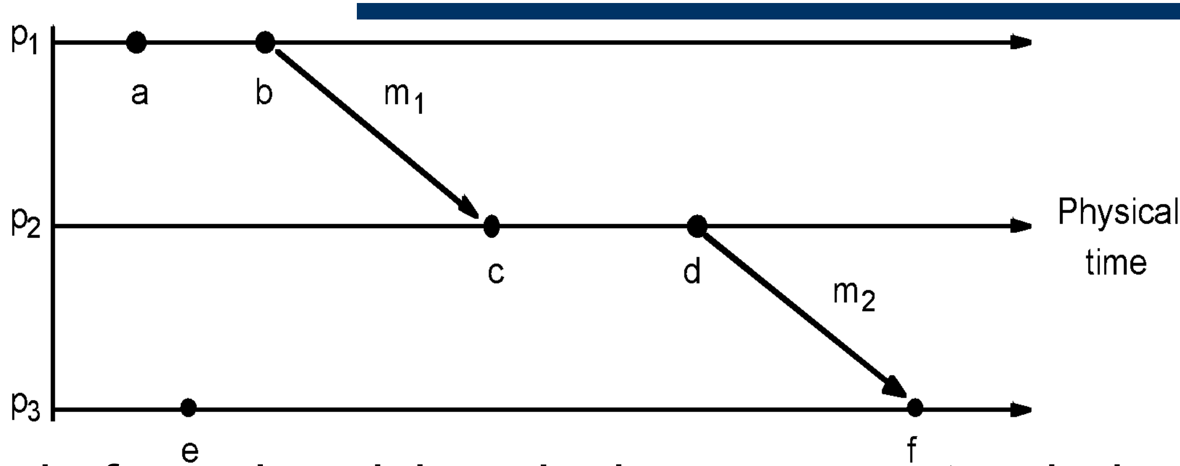
- Cristian's algorithm -
  - a single time server might fail, so they suggest the use of a group of synchronized servers
  - it does not deal with faulty servers
- Berkeley algorithm (also 1989)
  - An algorithm for internal synchronization of a group of computers
  - A *master* polls to collect clock values from the others (*slaves*)
  - The master uses round trip times to estimate the slaves' clock values
  - It takes an average (eliminating any above average round trip time or with faulty clocks)
  - It sends the required adjustment to the slaves (better than sending the time which depends on the round trip time)
  - Measurements
    - 15 computers, clock synchronization 20-25 millisecs drift rate < $2x10^{-5}$
    - If master fails, can elect a new master to take over (not in bounded time)

NTP Protocol

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of Send and Receive of previous message
  - Local times of Send of current message
- Recipient notes the time of receipt T_3 (we have T_0, T_1, T_2, T_3)



Logical time and logical clocks (Lamport 1978)



- Instead of synchronizing clocks, use event ordering

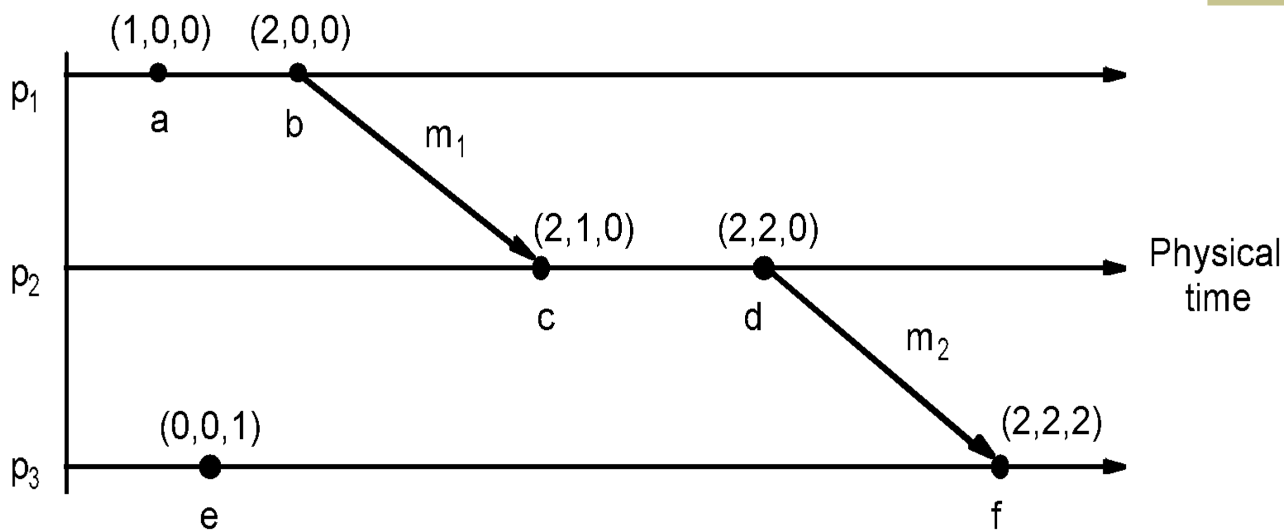
1. If two events occurred at the same process p_i ($i = 1, 2, \ldots N$) then they occurred in the order observed by p_i, that is the definition of: "$\rightarrow_i$"
2. when a message, m is sent between two processes, send(m) happens before receive(m)
3. The happened before relation is transitive

- The happened before relation is the relation of causal ordering

Total-order Lamport clocks

- Many systems require a total-ordering of events, not a partial-ordering
- Use Lamport's algorithm, but break ties using the process ID
  - $L(e) = M * L_i(e) + i$
    - M = maximum number of processes
    - i = process ID

Vector Clocks



- Note that e → e' implies V(e)<V(e'). The converse is also true
- Can you see a pair of parallel events?
  - *c* || *e* (parallel) because neither *V*(c) <= *V*(e) nor *V*(e) <= *V*(c)

Clock Sync Important Lessons

- Clocks on different systems will always behave differently
  - Skew and drift between clocks
- Time disagreement between machines can result in undesirable behavior
- Two paths to solution: synchronize clocks or ensure consistent clocks

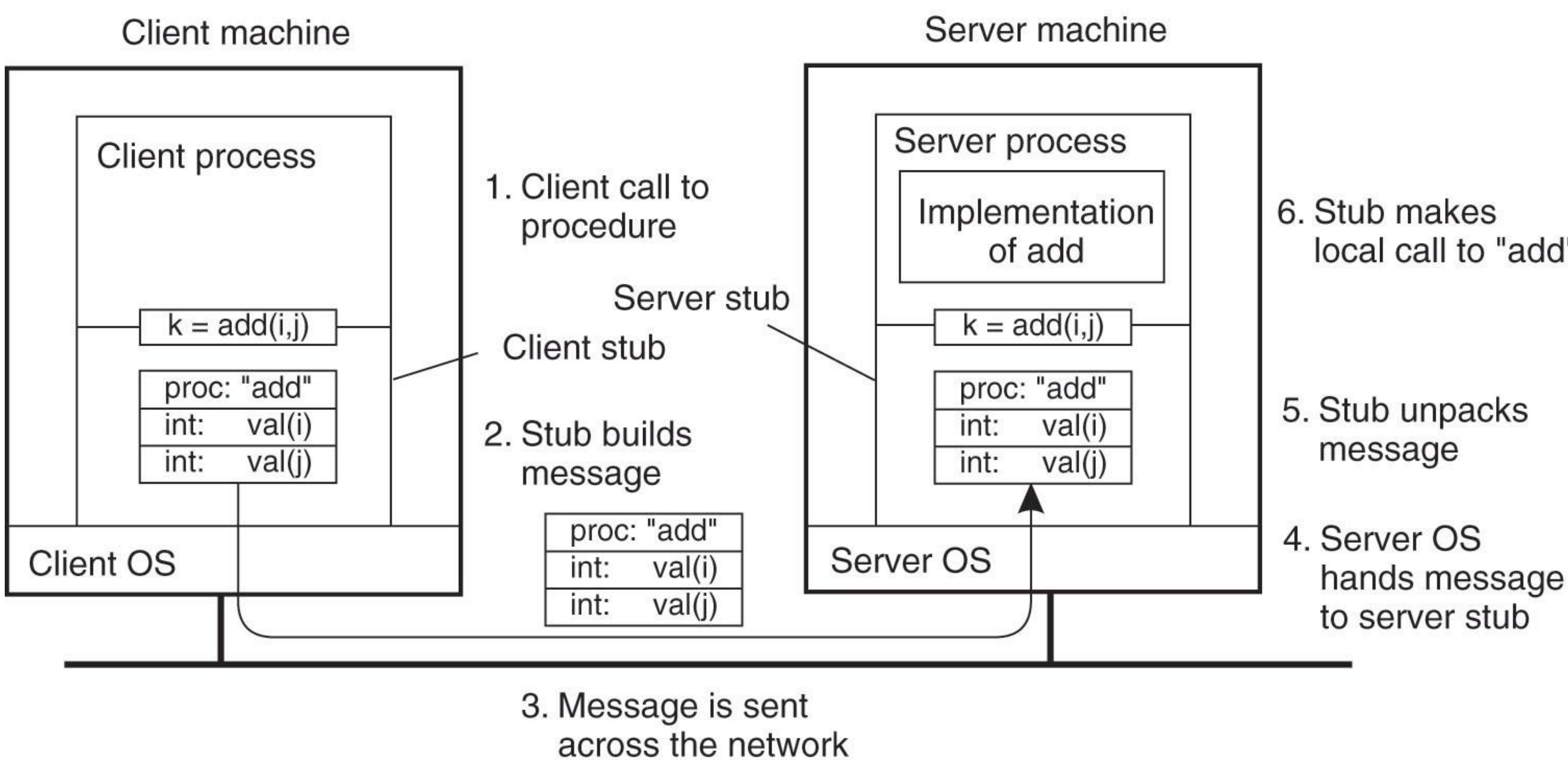
Clock Sync Important Lessons

- Clock synchronization
  - Rely on a time-stamped network messages
  - Estimate delay for message transmission
  - Can synchronize to UTC or to local source
  - Clocks never exactly synchronized
  - Often inadequate for distributed systems
  - might need totally-ordered events
  - might need millionth-of-a-second precision
- Logical Clocks
  - Encode causality relationship
  - Lamport clocks provide only one-way encoding
  - Vector clocks provide exact causality information

15-440 Distributed Systems

Remote Procedure Calls

Passing Value Parameters (1)



- The steps involved in a doing a remote computation through RPC.

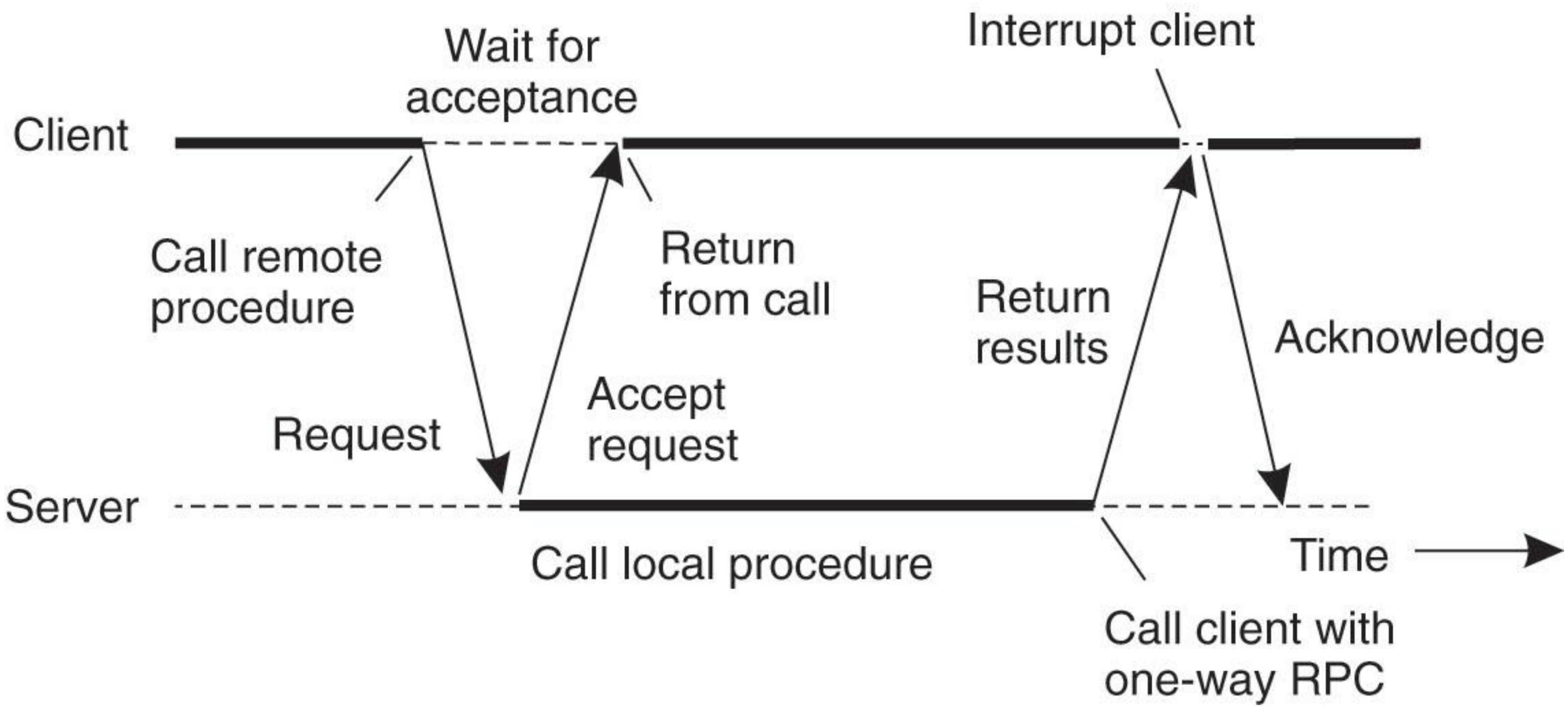
Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - **Marshals** arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - **Unmarshals** result and returns to caller
- Server stub
  - **Unmarshals** arguments and builds stack frame
  - Calls procedure
  - Server stub **marshals** results and sends reply

Real solution: break transparency

- Possible semantics for RPC:
  - Exactly-once
    - Impossible in practice
  - At least once:
    - Only for idempotent operations
  - At most once
    - Zero, don’t know, or once
  - Zero or once
    - Transactional semantics

Asynchronous RPC (3)



- A client and server interacting through two asynchronous RPCs.

Important Lessons

- Procedure calls
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Not only way…
- Hard to provide true transparency
  - Failures
  - Performance
  - Memory access
  - Etc.

15-440 Distributed Systems

Distributed File Systems

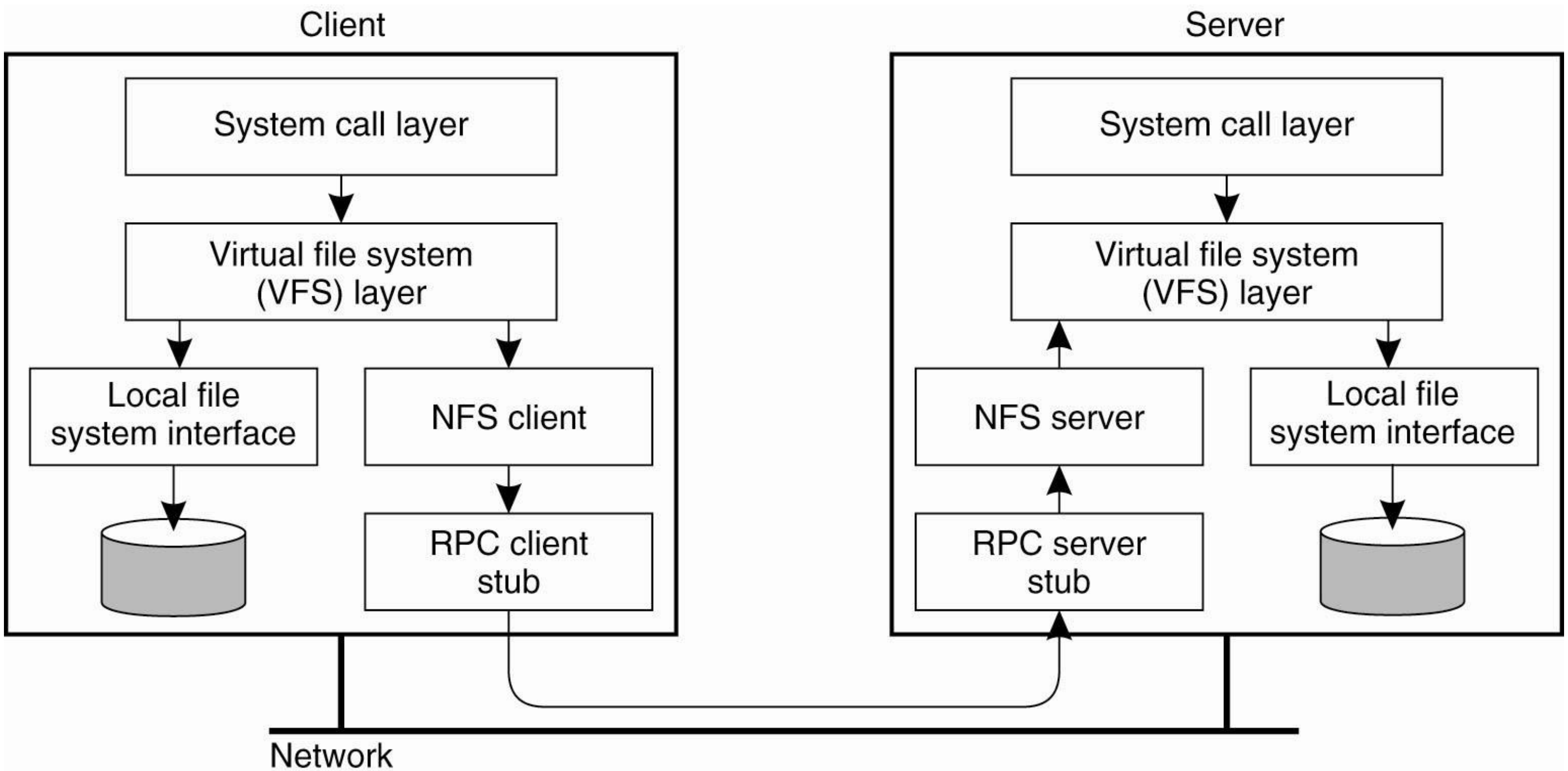
Why DFSs?

- Why Distributed File Systems:
  - Data sharing among multiple users
  - User mobility
  - Location transparency
  - Backups and centralized management
  - Examples: NFS, AFS, CODA, LBFS
- Idea: Provide file system interfaces to remote FS's
  - Challenge: heterogeneity, scale, security, concurrency,..
  - Non-Challenges: AFS meant for campus community
  - Virtual File Systems: pluggable file systems
  - Use RPC's

DFS Important bits (1)

- Distributed filesystems almost always involve a tradeoff: consistency, performance, scalability.
- We've learned a lot since NFS and AFS (and can implement faster, etc.), but the general lesson holds. Especially in the wide-area.
- We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff. Consistency, Availability, Partition-resilience.

VFS Interception



NFS's Failure Handling – Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of "who has this file open" (permission check on each operation on open file!)
  - No "pending transactions" across crash
- Crash recovery is "fast"
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocol in NFSv4
- Stateless protocol: requests specify exact state. read() → read([position]). no seek on server.

NFS's Failure Handling

- Operations are **idempotent**
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.
  - Not perfect → e.g., mkdir
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can't handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

AFS Cell/Volume Architecture

- Cells correspond to administrative groups
  - /afs/andrew.cmu.edu is a ***cell***
- Cells are broken into ***volumes*** (miniature file systems)
  - One user's files, project source tree, ...
  - Typically stored on one server
  - Unit of disk quota administration, backup
- Client machine has cell-server database
  - ***protection server*** handles authentication
  - ***volume location server*** maps volumes to servers

Client Caching in AFS

- Callbacks! Clients register with server that they have a copy of file;
  - Server tells them: “Invalidate!” if the file changes
  - This trades state for improved consistency
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone “who has which files cached?”
- What if client crashes?
  - Must revalidate any cached content it uses since it may have missed callback

AFS Write Policy

- Writeback cache
  - Opposite of NFS “every write is sacred”
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts “assumed rare”
  - Who wants to see a half-written file?

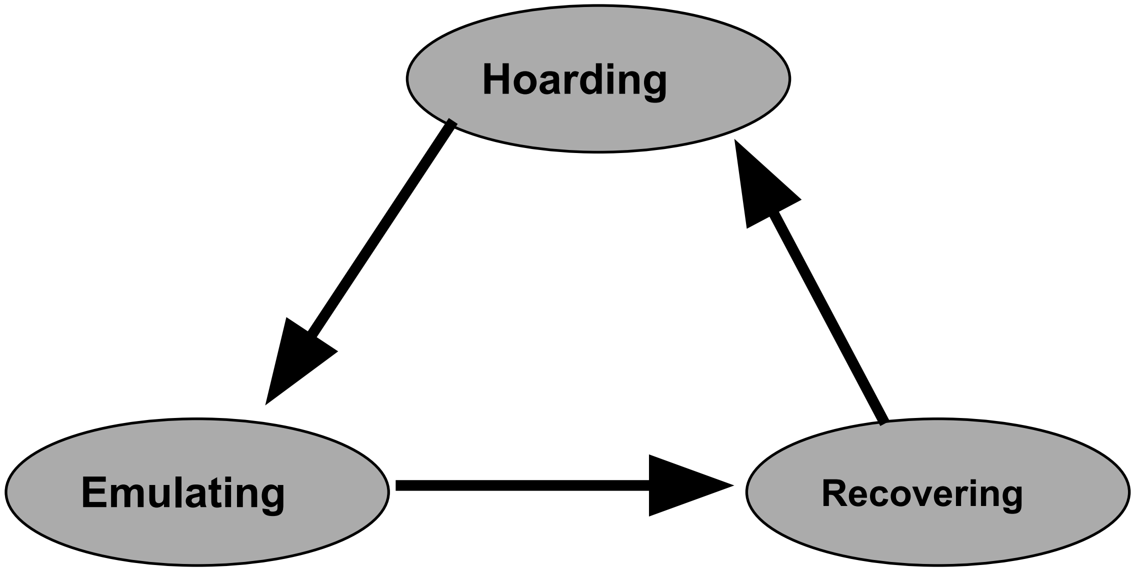
DFS: Name-Space Construction and Organization

- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1
- AFS: global name space
  - Name space is organized into Volumes
    - Global directory /afs;
    - /afs/cs.wisc.edu/vol1/…; /afs/cs.stanford.edu/vol1/…
  - Each file is identified as fid = <vol_id, vnode #, unique identifier>
  - All AFS servers keep a copy of "volume location database", which is a table of vol_id→ server_ip mappings

Coda Summary

- Distributed File System built for mobility
  - Disconnected operation key idea
- Puts scalability and availability before data consistency
  - Unlike NFS
- Assumes that inconsistent updates are very infrequent
- Introduced disconnected operation mode and file hoarding and the idea of “reintegration”

Coda States



1. ***Hoarding:***
   Normal operation mode
2. ***Emulating:***
   Disconnected operation mode
3. ***Reintegrating:***
   Propagates changes and detects inconsistencies

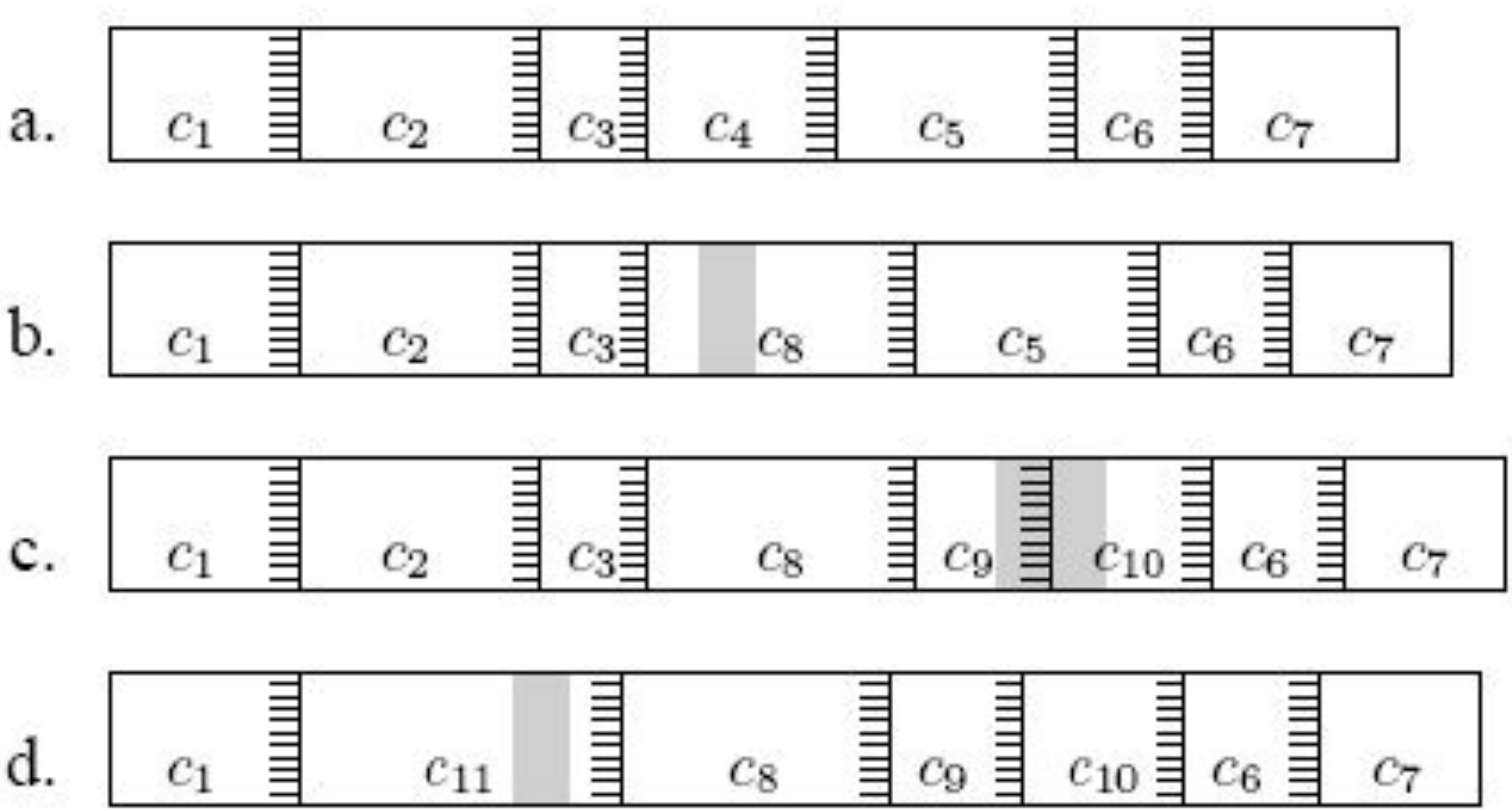
Low Bandwidth File System Key Ideas

- A network file systems for slow or wide-area networks
- Exploits similarities between files
  - Avoids sending data that can be found in the server's file system or the client's cache
  - Uses RABIN fingerprints on file content (file chunks)
    - Can deal with byte offsets when part of file change
- Also uses conventional compression and caching
- Requires 90% less bandwidth than traditional network file systems

LBFS chunking solution

- Considers only non-overlapping chunks
- Sets chunk boundaries based on file contents rather than on position within a file
- Examines every overlapping 48-byte region of file to select the boundary regions called *breakpoints* using Rabin fingerprints
  - When low-order 13 bits of region's fingerprint equals a chosen value, the region constitutes a breakpoint

Effects of edits on file chunks



- Chunks of file before/after edits
  - Grey shading show edits
- Stripes show regions with magic values that create chunk boundaries

15-440 Distributed Systems

Distributed Mutual Exclusion

What is “Scalability”?

Ability to easily and rapidly grow the system

A consequence of success   failing systems rarely grow :-)

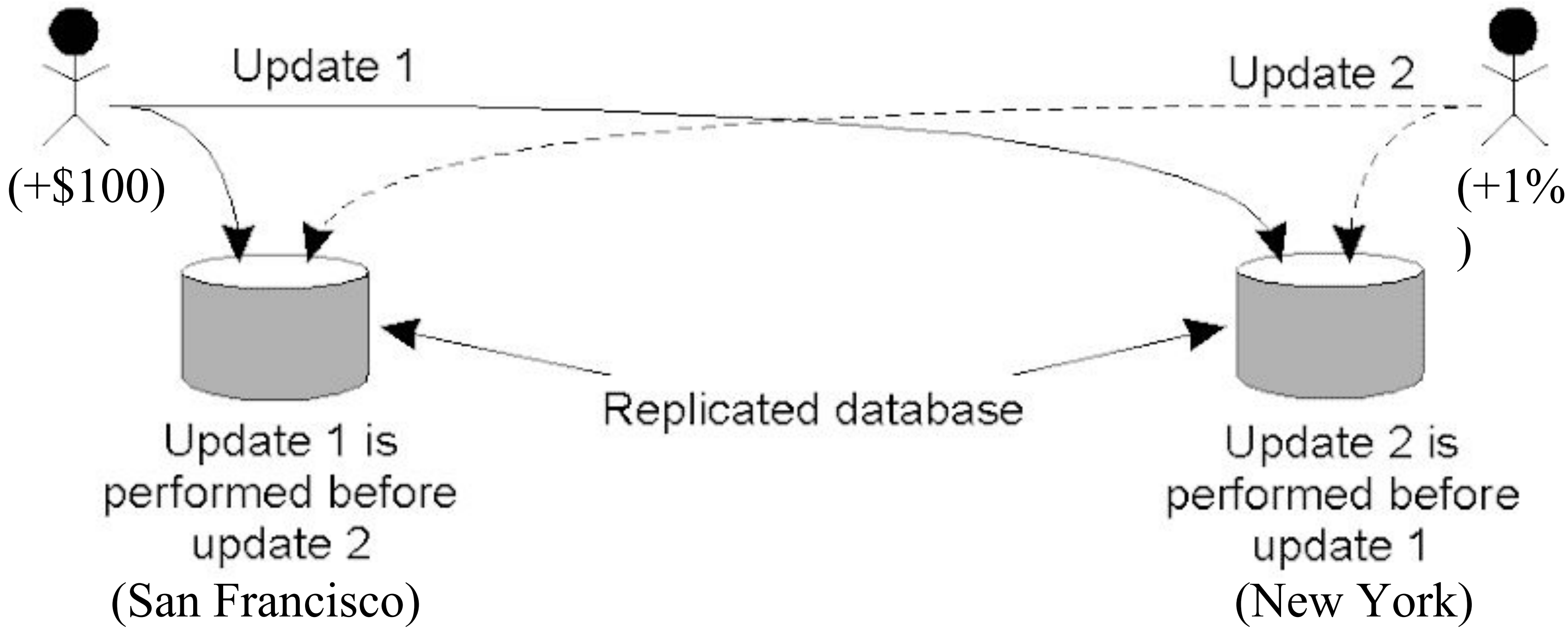
How to scale?

Two fundamental approaches:

Scale Up (aka “vertical scaling”)	Scale Out (aka “horizontal scaling”)
add resources to a single node e.g., more and faster CPUs, GPUs	**add more nodes** to the distributed system
no application changes huge win in terms of cost and time	**application has to conform to scale out design** - may involve total rewrite

Challenges when scaling out?

Motivation: Need for Distributed Mutex



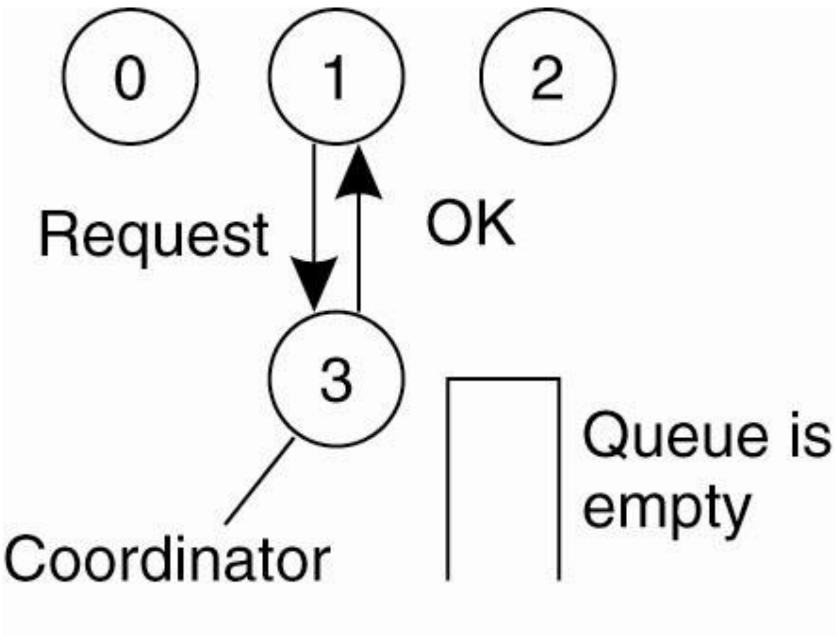
- San Fran customer adds $100, NY bank adds 1% interest
  - San Fran will have $1,111 and NY will have $1,110
- Updating a replicated database and leaving it in an inconsistent state

Comparison of 5 Mutex Algorithms

- Which one would you choose?
- What happens with crashes?

Algorithm	# Messages per cycle	Delay before entry	Problems
Centralized	3	2	Coordinator crash
Decentralized	2 m k + m, k≥1	2m	Starvation
Lamport	3 (N-1)	2 (N-1)	Crash of any process, inefficient
Ricart & Agrawala	2 (N-1)	2 (N-1)	Crash of any process
Token ring	1 to infinite	0 to (N-1)	Lost token, process crash

A Centralized Algorithm (1)



@ Server:

```
while true:
   m = Receive()
   If m == (Request, i):If Available():
         Send (Grant) to i
```

@ Client → Acquire:

```
Send (Request, i) to coordinator
Wait for reply
```

Distributed Algorithm (Strawman)

- Assume that there are n coordinators
  - Access requires a majority vote from m > n/2 coordinators.
  - A coordinator always responds immediately to a request with GRANT or DENY
- Node failures are still a problem
- Large numbers of nodes requesting access can affect availability

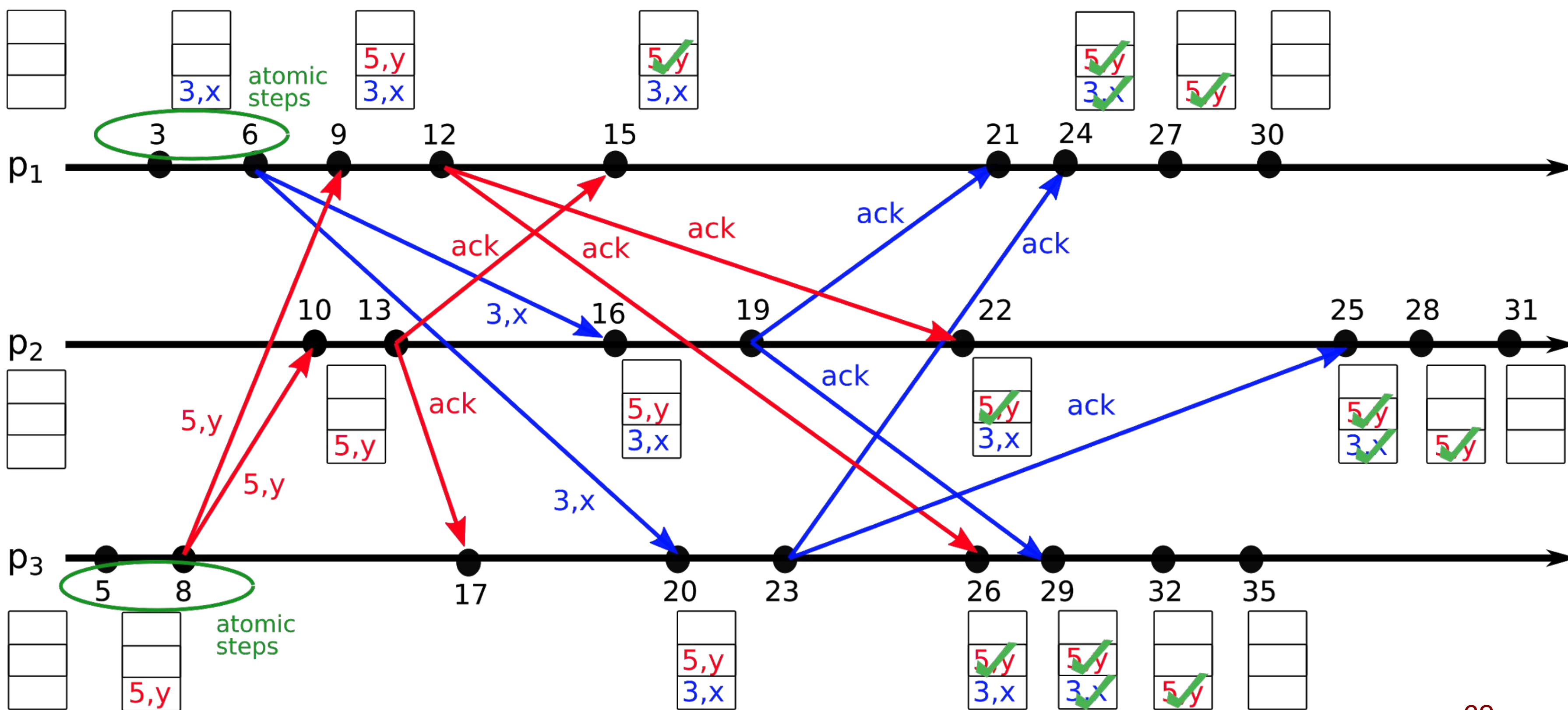
Book pp. 313

Totally-Ordered Multicast

- A multicast operation by which all messages are delivered in the same order to each receiver.
- Distributed data structure (priority queue)
- Queue messages until they're ACKed
- Uses TO-Lamport Clocks:
    - Each message is timestamped with the current logical time of its sender.
    - Multicast messages are also sent back to the sender.
    - **<u>Assume all messages sent by one sender are received in the order they were sent and that no messages are lost.</u>**

Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



Lamport Mutual Exclusion

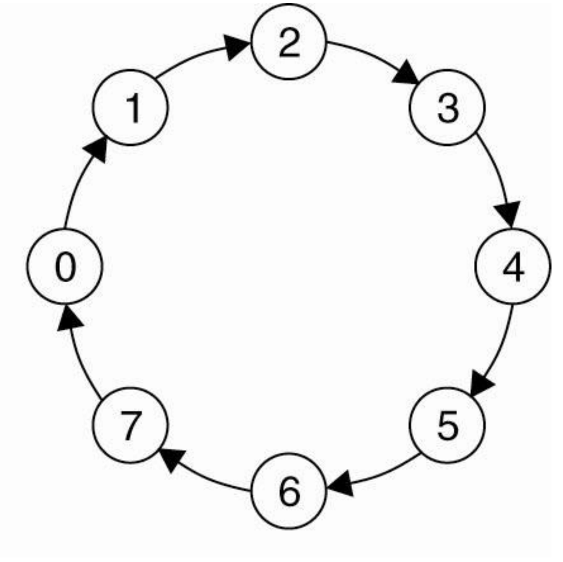
- Every process maintains a queue of pending requests for entering critical section in order. The queues are ordered by virtual time stamps derived from Lamport timestamps
  - For any events e, e' such that e --> e' (causality ordering), T(e) < T(e')
  - For any distinct events e, e', T(e) != T(e')

- When node i wants to enter C.S., it sends time-stamped request to all other nodes (including itself)
  - Wait for replies from all other nodes.
  - If own request is at the head of its queue and all replies have been received, enter C.S.
  - Upon exiting C.S., remove its request from the queue and send a release message to every process.

Ricart & Agrawala Algorithm

- Also relies on Lamport totally ordered clocks.
- When node i wants to enter C.S., it sends time-stamped request to all other nodes. These other nodes reply (eventually). When i receives n-1 replies, then can enter C.S.
- Trick: Node j having earlier request doesn't reply to i until after it has completed its C.S.

A Token Ring Algorithm



- Organize the processes involved into a logical ring
- One token at any time → passed from node to node along ring

A Token Ring Algorithm

- Correctness:
  - Clearly safe: Only one process can hold token
- Fairness:
  - Will pass around ring at most once before getting access.
- Performance:
  - Each cycle requires between 1 - ∞ messages
  - Latency of protocol between 0 & n-1
- Issues
  - Lost token

Summary

- Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue).
- Ricart & Agrawala's algorithms demonstrate utility of logical clocks.
- Centralized & ring based algorithms much lower message counts
- None of these algorithms can tolerate failed processes or dropped messages.

15-440 Distributed Systems

Distributed Concurrency Management

Distributed Concurrency Management

- Single Server: Transactions (RD/WR to Global State)
- ACID: Atomicity, Consistency, Isolation, Durability
  - E.g. banking app => ACID is violated if not careful
- Solutions: 2-phase locking (General, strict, strong strict)
  - Deadling with deadlocks => build “waits-for” graph
  - Transactions: 2 phases (prep, commit/abort)
    - Preparation: generate Lock Set “L”, Updates “U”
    - COMMIT (updated global state), ABORT (leave state as is)
    - Example using banking app

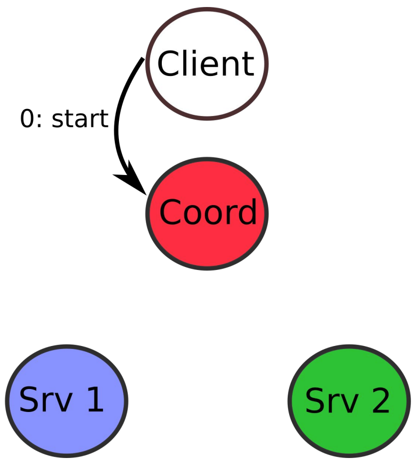
Transactions – split into 2 phases

- Phase 1: Preparation:
  - Determine what has to be done, how it will change state, without actually altering it.
  - Generate Lock set "L"
  - Generate List of Updates "U"
- Phase 2: Commit or Abort
  - Everything OK, then update global state
  - Transaction cannot be completed, leave global state as is
  - In either case, RELEASE ALL LOCKS

Distributed Transactions – 2PC

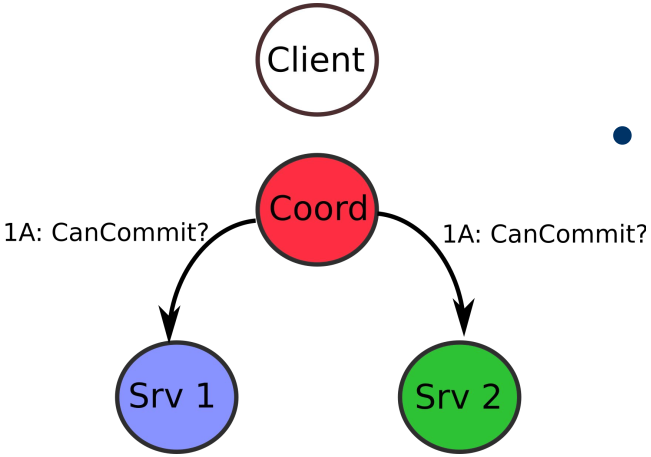
- Similar idea as before, but:
  - State spread across servers (maybe even WAN)
  - Want to enable single transactions to read and update global state while maintaining ACID properties
- Overall Idea:
  - Client initiate transaction. Makes use of “co-ordinator”
  - All other relevant servers operate as “participants”
  - Co-ordinator assigns unique transaction ID (TID)

Implementing 2-Phase Commit



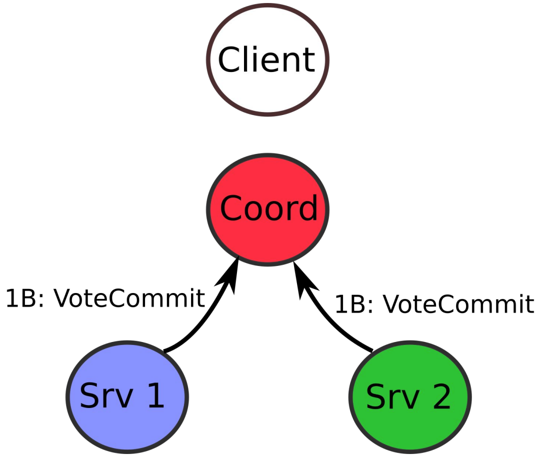
- Implemented as a set of messages

Implementing 2-Phase Commit



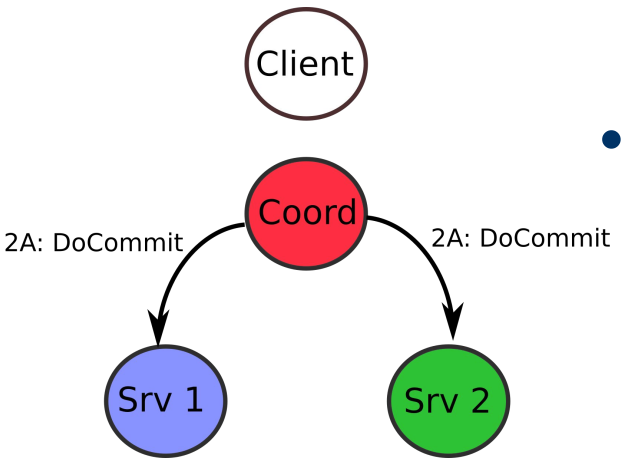
- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants

Implementing 2-Phase Commit



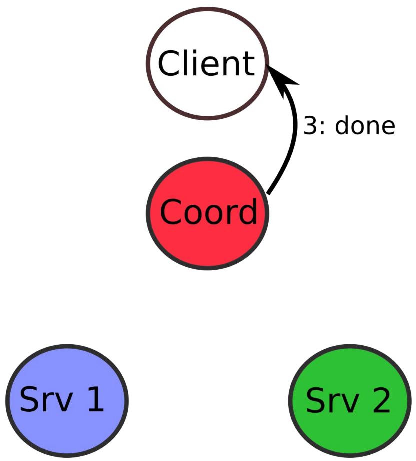
- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants
  - B: Participants respond: “VoteCommit” or “VoteAbort”

Implementing 2-Phase Commit



- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends "CanCommit?" to participants
  - B: Participants respond: "VoteCommit" or "VoteAbort"
- Messages in the second phase
  - A: All "VoteCommit": , Coord sends "DoCommit"
  - If any "VoteAbort": abort transaction. Coordinator sends "DoAbort" to everyone => release locks

Implementing 2-Phase Commit



- Implemented as a set of messages
- Messages in first phase
  - A: Coordinator sends “CanCommit?” to participants
  - B: Participants respond: “VoteCommit” or “VoteAbort”
- Messages in the second phase
  - A: All “VotedCommit”: , Coord sends “DoCommit”
  - If any “VoteAbort”: abort transaction. Coordinator sends “DoAbort” to everyone => release locks

Deadlocks and Livelocks

- Distributed deadlock
  - Cyclic dependency of locks by transactions across servers
  - In 2PC this can happen if participants unable to respond to voting request (e.g. still waiting on a lock on its local resource)
  - Handled with a timeout. Participants times out, then votes to abort. Retry transaction again.
    - Addresses the deadlock concern
    - However, danger of LIVELOCK – keep trying!

Summary: Distributed Concurrency

- Distributed consistency management
- ACID Properties desirable
- Single Server case: use locks, and in cases use 2-phase locking (strict 2PL, strong strict 2PL), transactional support for locks
- Multiple server distributed case: use 2-phase commit for distributed transactions. Need a coordinator to manage messages from partipants.

15-440 Distributed Systems

Fault Tolerance, Logging and Recovery

Summary – Fault Tolerance

- Real Systems (are often unreliable)
  - Introduced basic concepts for Fault Tolerant Systems including redundancy, process resilience, RPC
- Fault Tolerance – Backward recovery using checkpointing, both Independent and coordinated
- Fault Tolerance –Recovery using Write-Ahead-Logging

Dependability Concepts

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system runs continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen. (e.g. process control systems)
- *Maintainability* – when a system fails, it can be repaired easily and quickly (sometimes, without its users noticing the failure)

Masking Failures by Redundancy

- **Strategy**: hide the occurrence of failure from other processes using *redundancy*.

1. *Information Redundancy* – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).
2. *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
3. *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

Recovery Strategies

- When a failure occurs, we need to bring the system into an error free state (recovery). This is fundamental to Fault Tolerance.

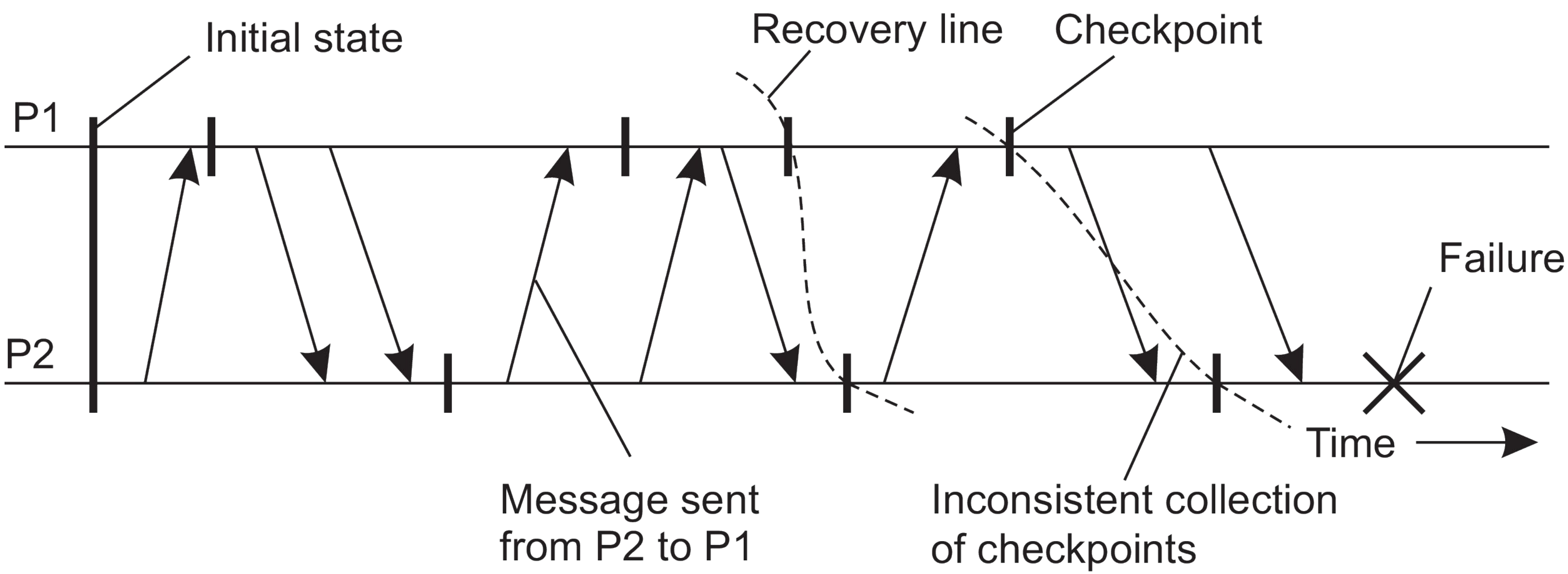
1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.

   -- Can be expensive, however still used

2. **Forward Recovery**: bring the system into a correct new state, from which it can then continue to execute.

   -- Need to know potential errors up front!

Independent Checkpointing



Recovery line: correct distributed snapshot

This becomes challenging if checkpoints are un-coordinated

Coordinated Checkpointing

- Key idea: each process takes a checkpoint after a globally coordinated action. (why is this good?)

- Simple Solution: 2-phase blocking protocol
  - Co-ordinator multicast *checkpoint_REQUEST* message
  - Participants receive message, takes a checkpoint, stops sending (application) messages, and sends back *checkpoint_ACK*
  - Once all participants ACK, coordinator sends *checkpoint_DONE* to allow blocked processes to go on

- Optimization: consider only processes that depend on the recovery of the coordinator (those it sent a message since last checkpoint)

Write-Ahead-Logging

- Provide Atomicity and Durability
- Idea: create a log recording every update to database
- Updates considered reliable when stored on disk
- Updated versions are kept in memory (page cache)
- Logs typically store both REDO and UNDO operations
- After a crash, recover by replaying log entries to reconstruct correct state
- 3 Passes: (Analysis Pass, recovery pass, Undo Pass)
- WAL is common, fewer disk operations, transactions considered committed once log written.

Recovery using WAL – 3 passes

- Analysis Pass
  - Reconstruct TT and DPT (from start or last checkpoint)
  - Get copies of all pages at the start
- Recovery Pass (redo pass)
  - Replay log forward, make updates to all dirty pages
  - Bring everything to a state at the time of the crash
- Undo Pass
  - Replay log file backward, revert any changes made by transactions that had not committed (use PrevLSN)
  - For each write Compensation Log Record (CLR)
  - Once you reach BEGIN TXN, write an END TXN entry

Optimizing WAL

- As described earlier:
  - Replay operations back to the beginning of time
  - Log file would be kept forever, (entire Database)
- In practice, we can do better with CHECKPOINT
  - Periodically save DPT, TT
  - Store any dirty pages to disk, indicate in LOG file
  - Prune initial portion of log file: All transactions upto checkpoint have been committed or aborted.

15-440 Distributed Systems

Distributed Replication

Distributed Consistency Concepts

- Requires write replication, and some degree of consistency
  - **Strict** Consistency
    - Read always returns value from latest write
  - **Sequential** Consistency
    - All nodes see operations in some sequential order
    - Operations of each process appear in-order in this sequence
  - **Causal Consistency**
    - All nodes see causally related writes in same order
    - But concurrent writes may be seen in different order on different machines
  - **Eventual Consistency**
    - All nodes will learn eventually about all writes, in the absence of updates

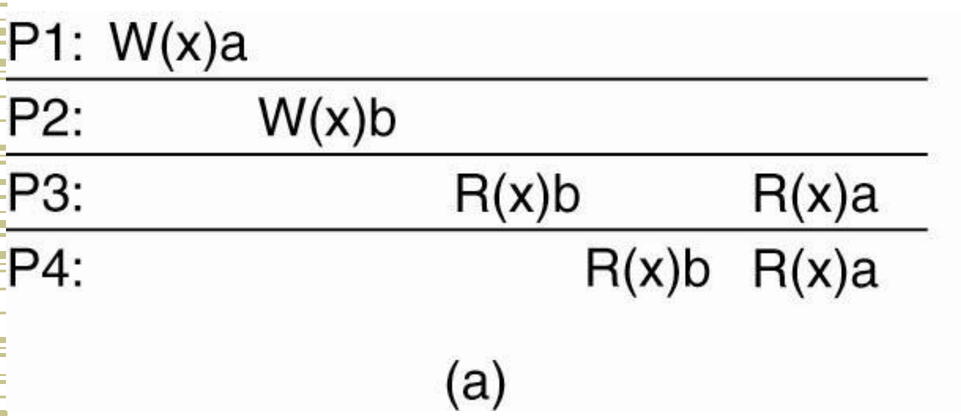
Sequential Consistency (1)

- Behavior of two processes operating on the same data item. The horizontal axis is time.
- P1: Writes “W” value a to variable “x”
- P2: Reads `NIL’ from “x” first and then `a’

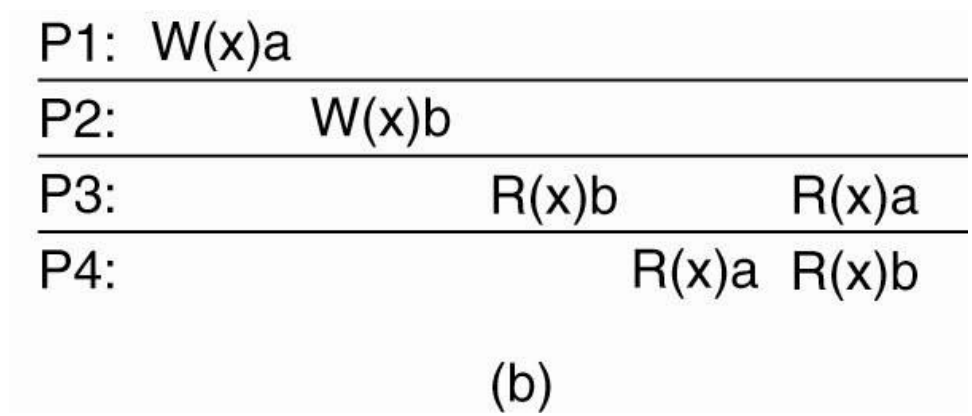
Sequential Consistency (2)

- A data store is sequentially consistent when:
- The result of any execution is the same as if the (read and write) operations by all processes on the data store …
  - Were executed in some sequential order and …
  - the operations of each individual process appear …
    - in this sequence
    - in the order specified by its program.

Sequential Consistency (3)



(a) A sequentially consistent data store.



(b) A data store that is not sequentially consistent.

Causal Consistency (1)

- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:
- Writes that are potentially causally related …
  - must be seen by all processes
  - in the same order.
- Concurrent writes …
  - may be seen in a different order
  - on different machines.

Causal Consistency (2)

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Figure 7-8. This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Replicate: State versus Operations

Possibilities for what is to be propagated:

- Propagate only a notification of an update.

- Sort of an “invalidation” protocol

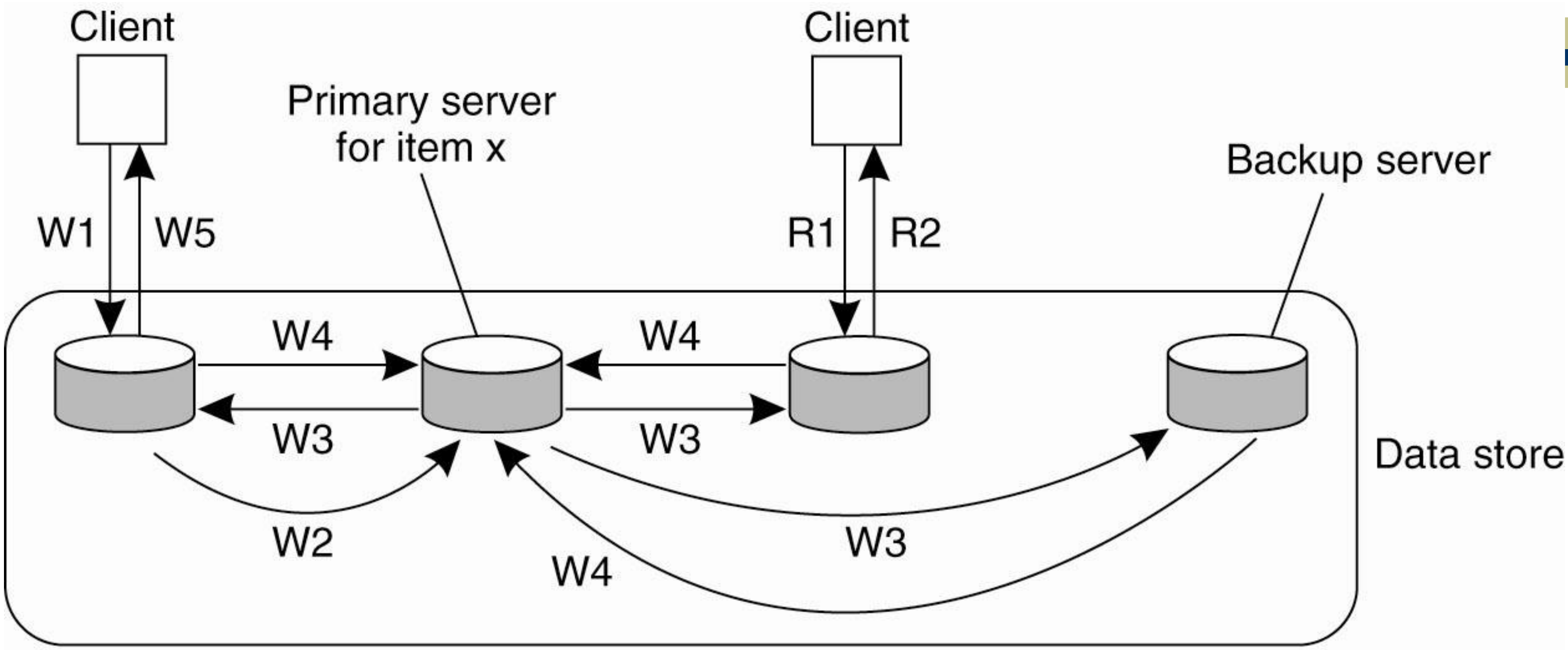
- Transfer data from one copy to another.

- Read-to-Write ratio high, can propagate logs (save bandwidth)

- Propagate the update operation to other copies

- Don’t transfer data modifications, only operations – “Active replication”

Remote-Write PB Protocol



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Updates are blocking, although non-blocking possible

Replication: Quorum based consensus

- Quorum consensus
  - Designed to have fast response time even under failures
  - Replicas are “active” - participate in protocol; there is no master, per se.
  - Good: Clients don’t even see the failures. Bad: More complex.

PAXOS: Requirement

- Correctness (safety):
  - All nodes agree on the same value
  - The agreed value X has been proposed by some node
- Fault-tolerance:
  - If less than N/2 nodes fail, the rest should reach agreement *eventually w.h.p*
  - Liveness is not *guaranteed*
- *Termination (not guaranteed)*

Fischer-Lynch-Paterson [FLP'85] impossibility result

- It is impossible for a set of processors in an asynchronous system to agree on a binary value, even if only a single processor is subject to an unannounced failure.
- Synchrony --> bounded amount of time node can take to process and respond to a request
  Asynchrony --> timeout is not perfect

Single Decree Paxos: Protocol

Proposers

1)Choose new proposal number n, value v

2)Broadcast Prepare(n) to all servers

4)When responses received from majority:

- If any acceptedValues returned
  v = acceptedValue of highest acceptedProposal

5)Broadcast Accept(n, value) to all servers

6)When Accept-OK from majority
  Value is chosen (Commit)
Else
  Restart: goto 1, with larger number n

Acceptors

3)Respond to Prepare(n):

- If n > minProposal then minProposal = n
  Prepare-OK(acceptedProposal, acceptedValue)
  else
  Prepare-REJECT()

6)Respond to Accept(n, value):

- If n ≥ minProposal
    acceptedProposal = minProposal = n
    acceptedValue = value
    Accept-OK()
  else
    Accept-REJECT()

Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)

Some Remarks

- Only proposer knows chosen value (majority accepted)
- Only a single value is chosen → MultiPaxos
- No guarantee that proposer's original value v is chosen by itself
- Number n is basically a Lamport clock → always unique n
- Key invariant:
  - If a proposal with value `v' is chosen, all higher proposals must have value `v'
- Dueling proposer
  - Resolved using number n in prepare
- There are challenging corner cases

15-440 Distributed Systems

Fault Tolerance and RAID

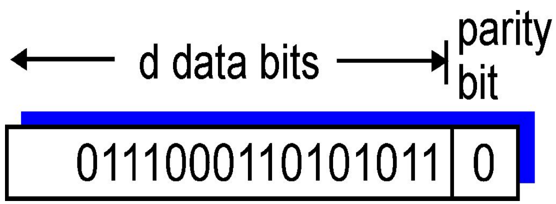
Outline

- Errors/error recovery
- Using multiple disks
  - Why have multiple disks?
  - problem and approaches
- RAID levels and performance
- Estimating availability

Parity Checking

Single Bit Parity:

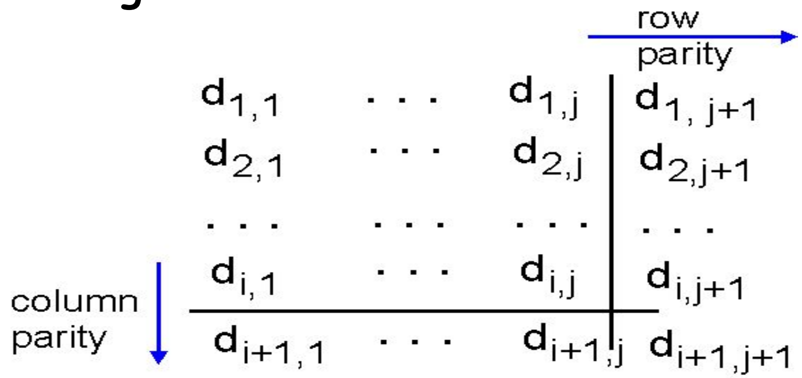
Detect single bit errors

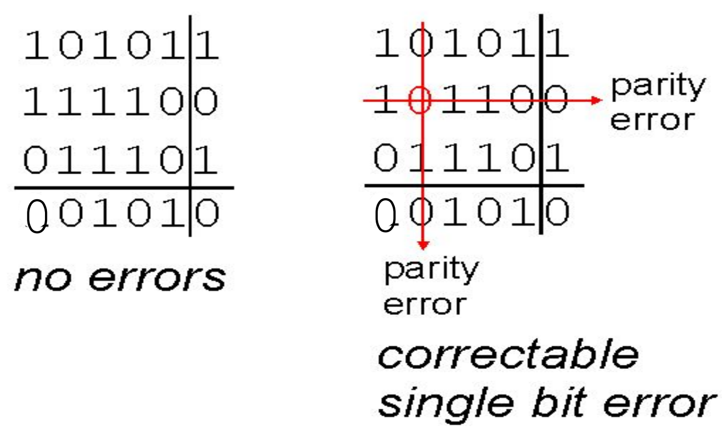


Error Recovery – Error Correcting Codes (ECC)

Two Dimensional Bit Parity:

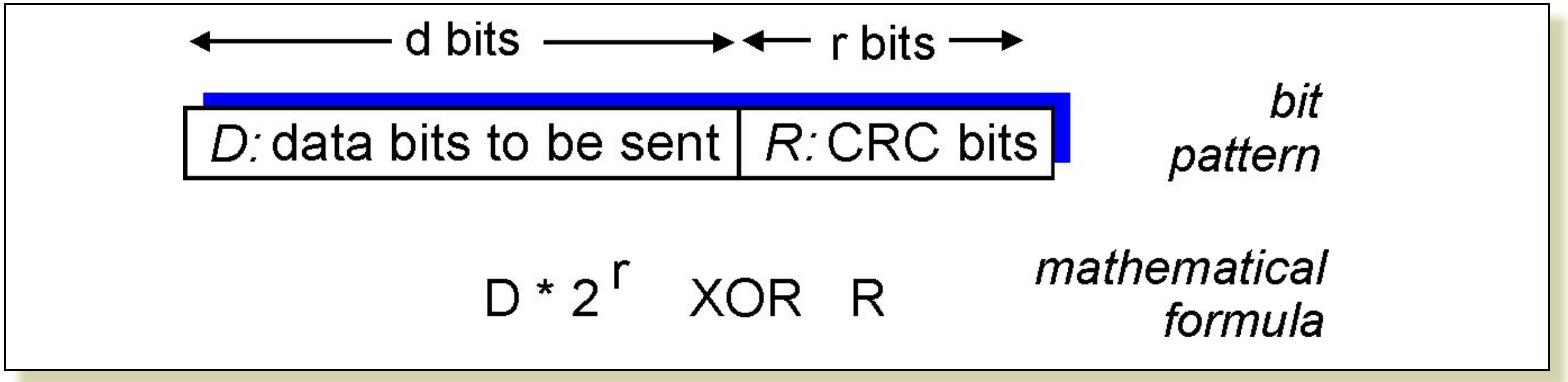
Detect *and correct* single bit errors





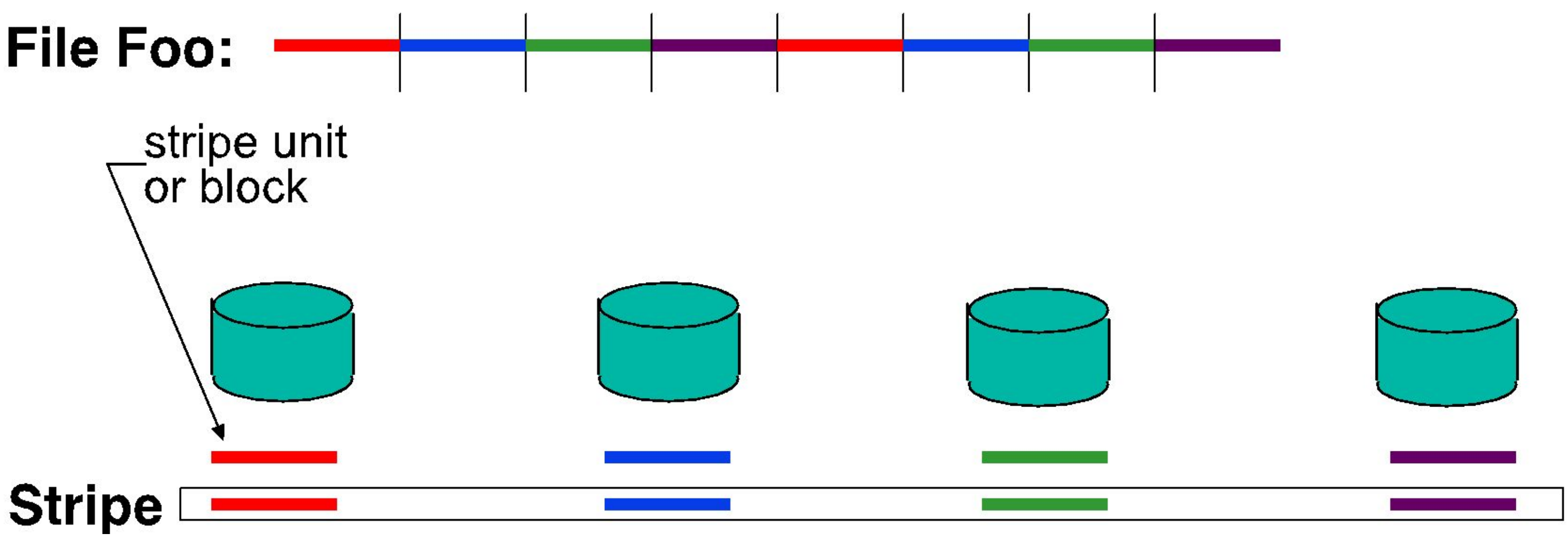
Error Detection – CRC

- View data bits, D, as a binary number
- Choose r+1 bit pattern (generator), G
- Goal: choose r CRC bits, R, such that
  - <D,R> exactly divisible by G (modulo 2)
  - Receiver knows G, divides <D,R> by G. If non-zero remainder: error detected!
  - Can detect all burst errors less than r+1 bits
- Widely used in practice: Ethernet, disks



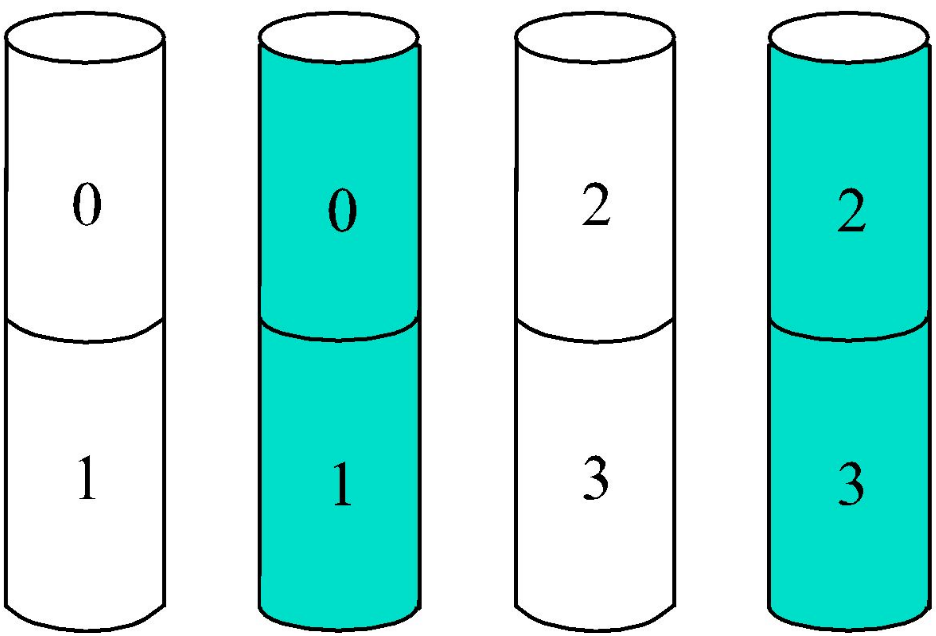
Disk Striping

- Interleave data across multiple disks
  - Large file streaming can enjoy parallel transfers
  - Small requests benefit from load balancing
    - If blocks of hot files equally likely on all disks (really?)



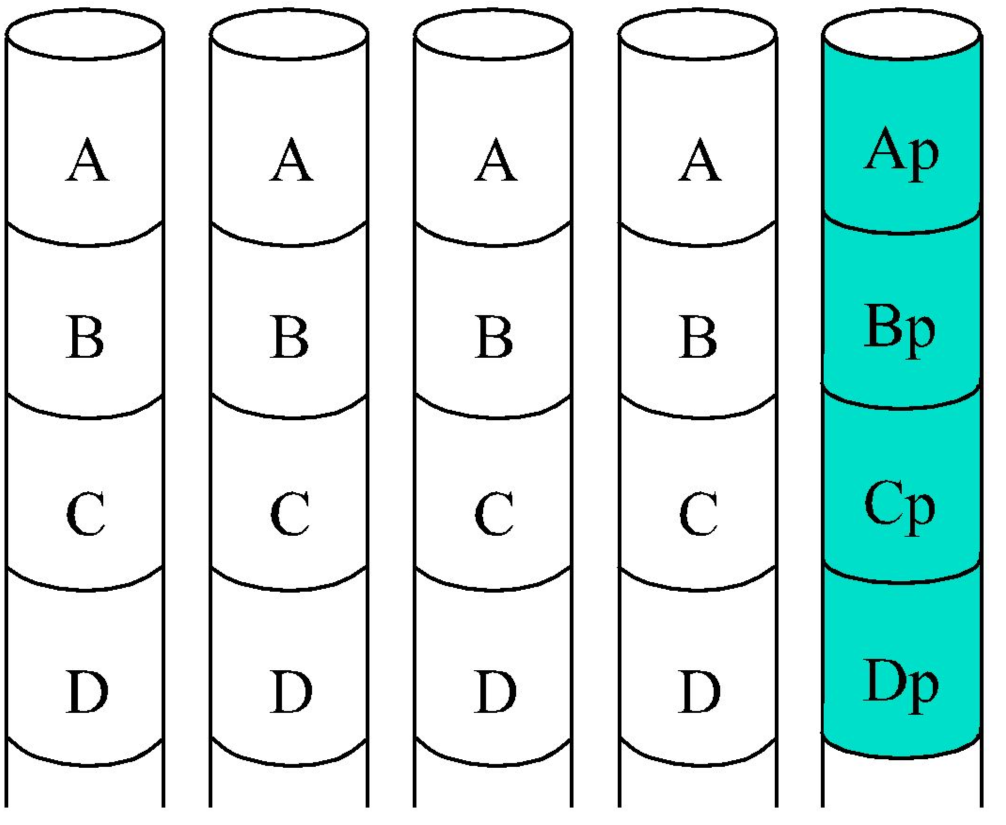
Redundancy via replicas

- Two (or more) copies
  - mirroring, shadowing, duplexing, etc.
- Write both, read either

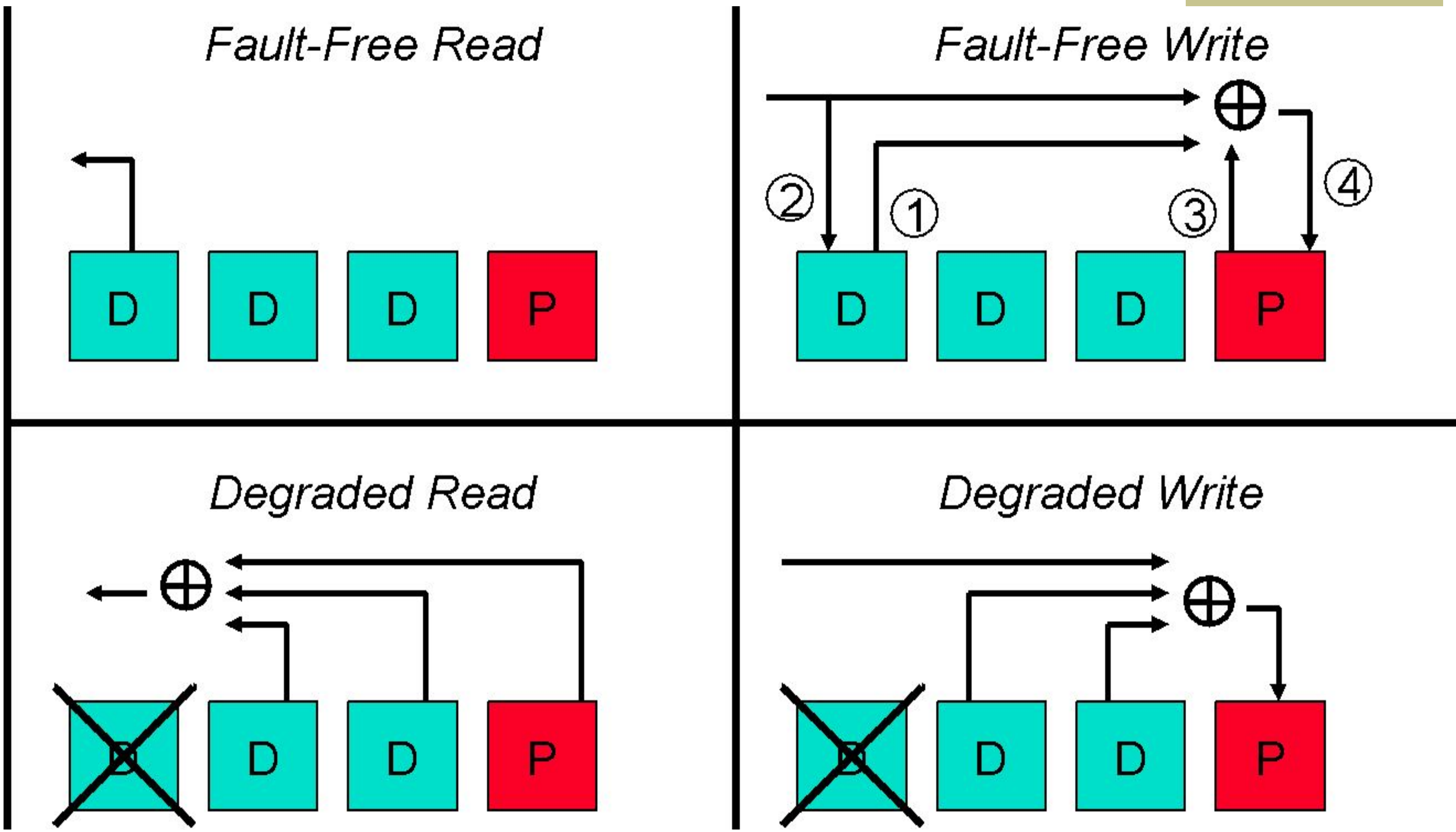


A Better Approach?: Parity Disk

- Capacity: one extra disk needed per stripe
- Disk failures are self-identifying (a.k.a. erasures)
  - Don't have to find the error

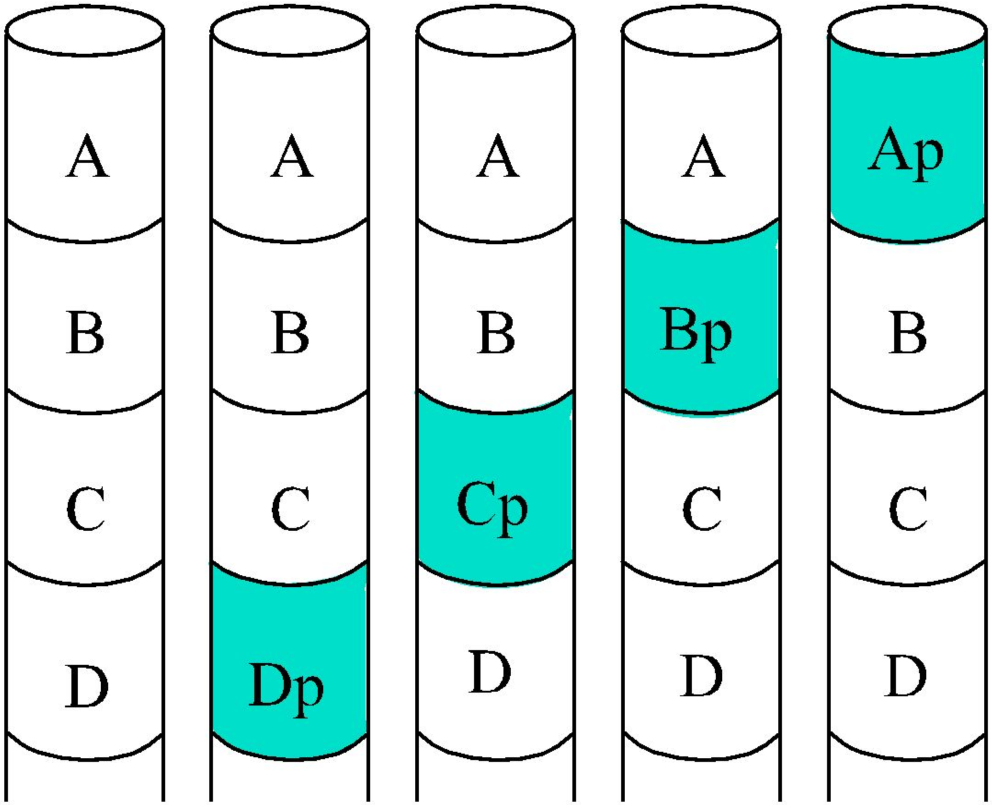


Updating and using the parity



Better: Striping the Parity

- Removes parity disk bottleneck



Performance

B: # of blocks per disk
R: R/W throughput of a disk
N: # of disks
D: time to R/W block

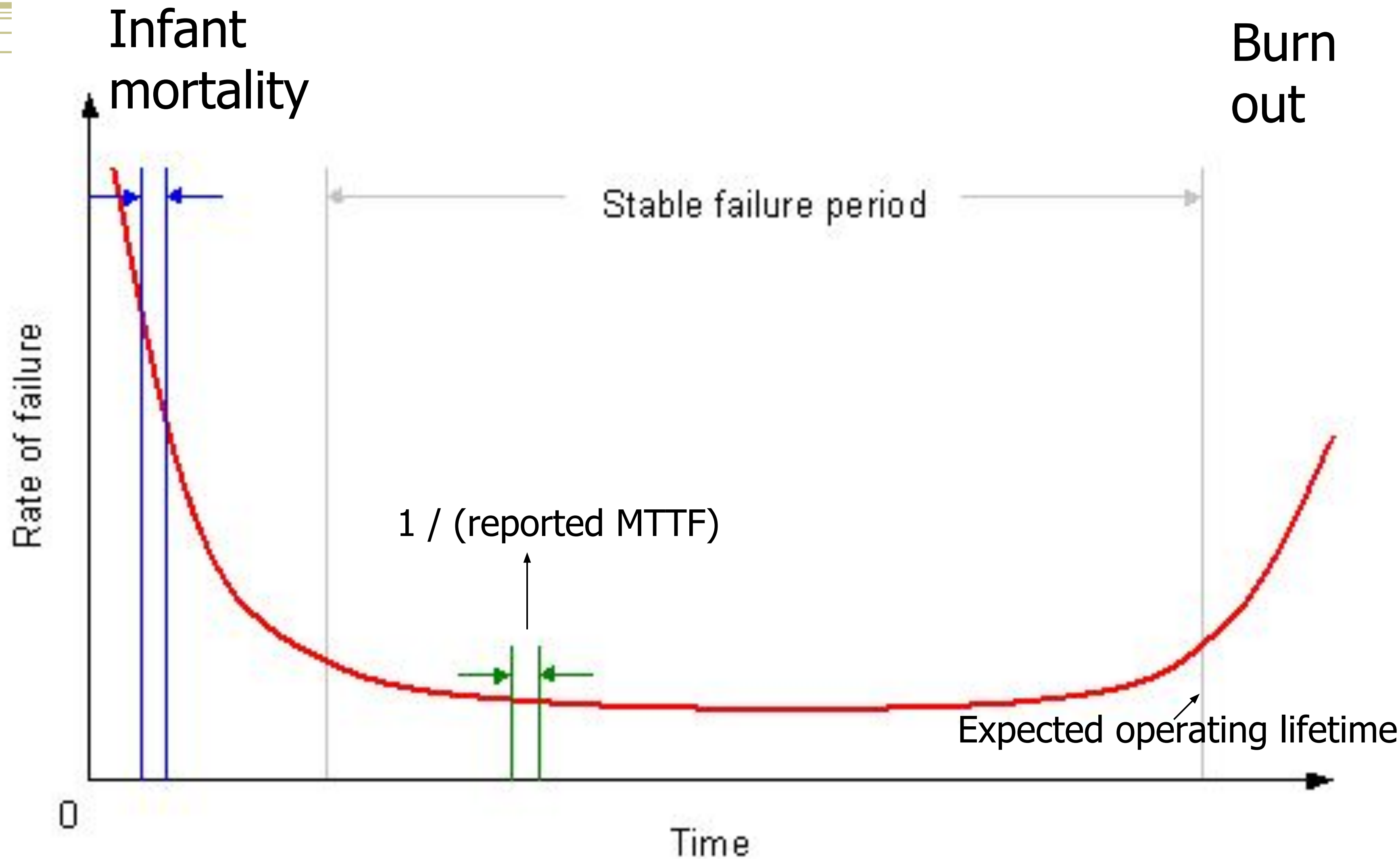
Level	Scheme	Capacity	Reliability	Read Throughput	Write Throughput
Single Disk		B	0	R	R
RAID-0	Striping	$N \cdot B$	0	$N \cdot R$	$N \cdot R$
RAID-1	Mirroring	$\frac{N}{2} \cdot B$	1 (for sure) $\frac{N}{2}$ (if lucky)	$N \cdot R$	$\frac{N}{2} \cdot R$
RAID-4	Parity Disk	$(N-1)B$	1	$(N-1)R$	$\frac{R}{2}$
RAID-5	Rotating Parity	$(N-1)B$	1	$N \cdot R$	$\frac{N}{4} \cdot R$

Measuring Availability

- Mean time to failure (MTTF) - "uptime"
- Mean time to repair (MTTR)
- Mean time between failures (MTBF)
- MTBF = MTTF + MTTR

- Availability = MTTF / (MTTF + MTTR)
  - Suppose OS crashes once per month, takes 10min to reboot.
  - MTTF = 720 hours = 43,200 minutes
    MTTR = 10 minutes
  - Availability = 43200 / 43210 = 0.997 (~"3 nines")

Disk failure conditional probability distribution - Bathtub curve



Reliability without rebuild

- 200 data drives with $MTTF_{drive}$
  - $MTTDL_{array} = MTTF_{drive} / 200$
- Add 200 drives and do mirroring
  - $MTTF_{pair} = (MTTF_{drive} / 2) + MTTF_{drive} = 1.5 * MTTF_{drive}$
  - $MTTDL_{array} = MTTF_{pair} / 200 = MTTF_{drive} / 133$
- Add 50 drives, each with parity across 4 data disks
  - $MTTF_{set} = (MTTF_{drive} / 5) + (MTTF_{drive} / 4) = 0.45 * MTTF_{drive}$
  - $MTTDL_{array} = MTTF_{set} / 50 = MTTF_{drive} / 111$
- These are **approximations**

15-440 Distributed Systems

Distributed Databases Case Study

Consistency Definitions

External Consistency

- If T1 commits before T2, then the commit order must be T1 before T2

Sequential Consistency

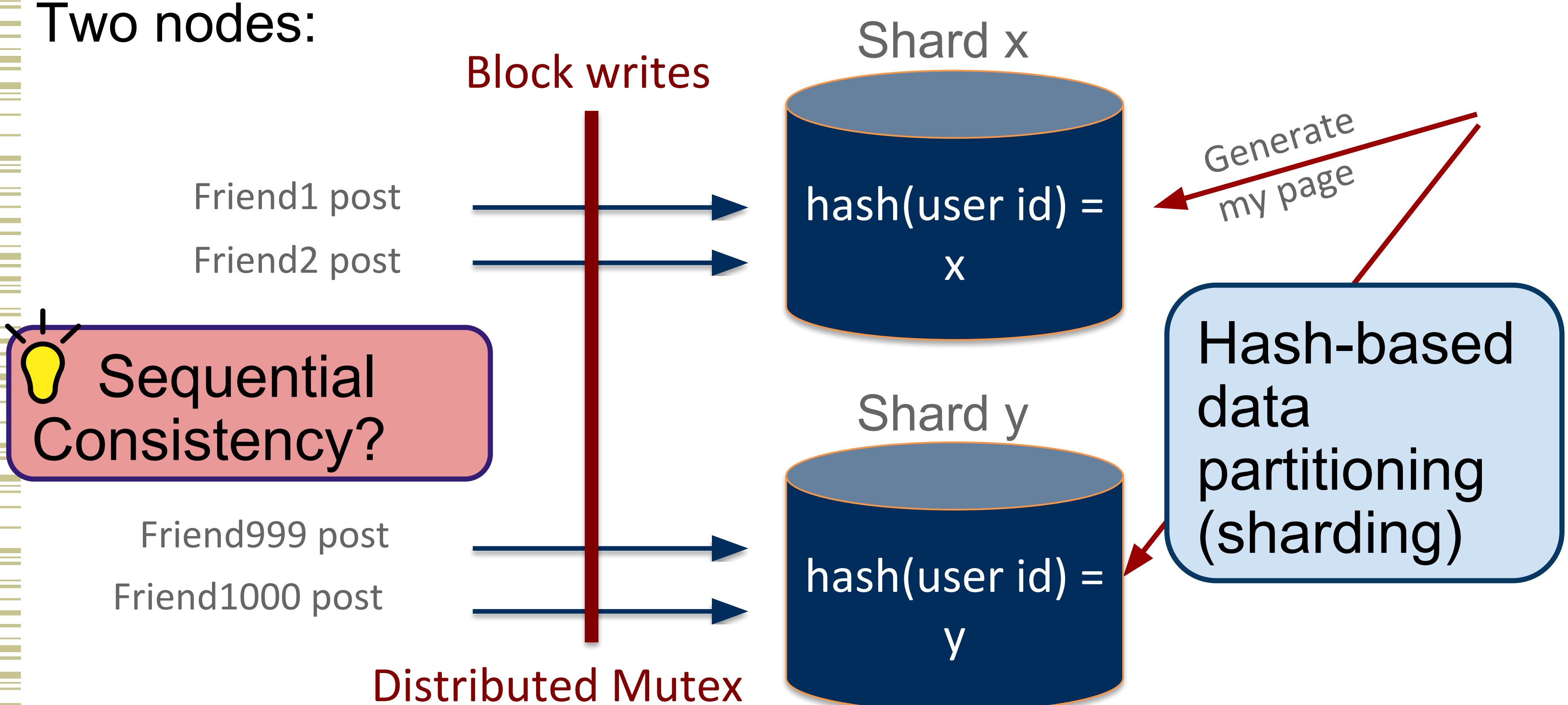
- All nodes see operations in some sequential order
- Operations of each process appear in-order in this sequence

Eventual Consistency

- All nodes will learn eventually about all writes, in the absence of updates

Consistent Distributed Database

Two nodes:



Summary So Far: When to Use What?

	Use Case	Problems
Distributed Mutex	Distributed KV without transactions	Failures + Slow
2PC	Distributed DB with transactions (e.g., Spanner)	Failures
Primary-Backup	Cost-efficient fault tolerance (e.g., FaRM, GFS, VMWare-FT)	Correlated failures
Paxos	Staying up no matter the cost (e.g., Spanner, FaunaDB)	Delay and huge cost overhead
RAID, Checksums	Every system	Node failures

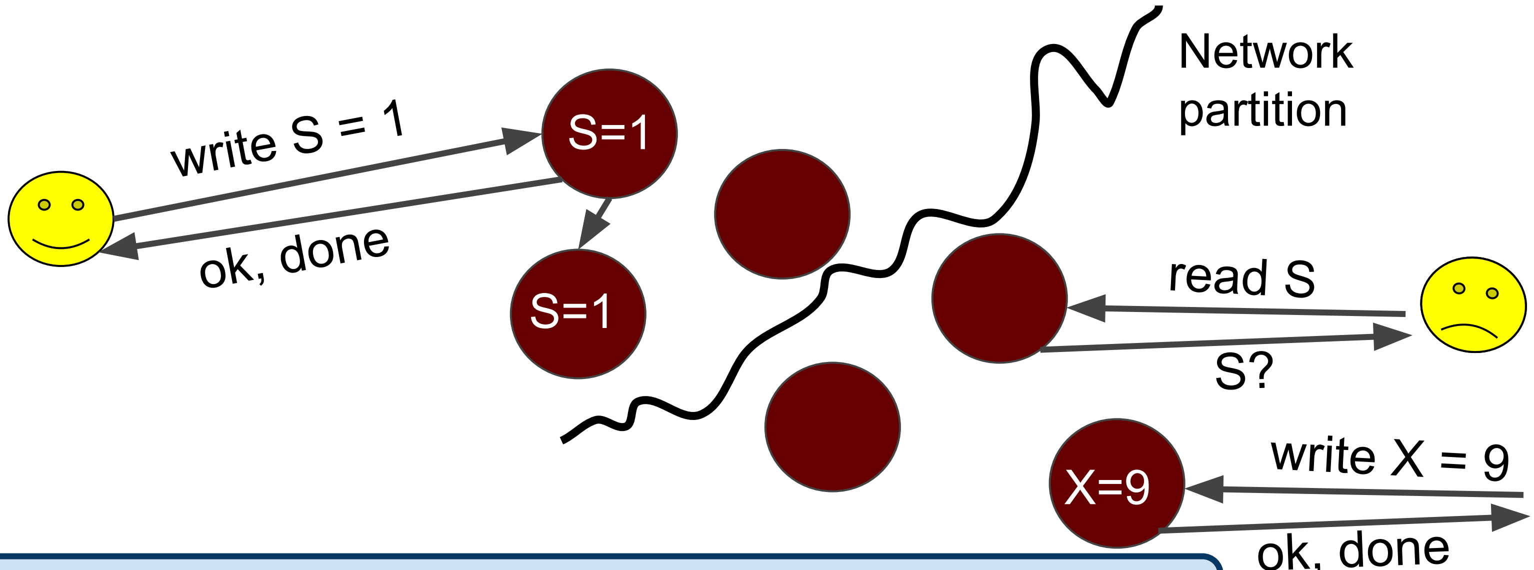
Practical Constraints: Alternative I

2005-2012: NoSQL systems

Only eventually consistent!

Amazon's Dynamo, Facebook's Cassandra,
Microsoft's Azure CosmosDB, Apache CouchDB, Basho's Riak

Design choices: AP: availability over consistency
"infinitely" scalable



Challenge: version reconcilation (parallel writes..)

Practical approach (Dynamo): Vector Clocks

Practical Constraints: Alternative II

- 2012-2018: resurgence of consistent distributed DBs
  - Google's Spanner, Microsoft's FaRM, Calvin and FaunaDB

These guarantee at least sequential consistency, unlike NoSQL.

- Three key reasons [→ Daniel Abadi, UMD]

1. application code gets too complex and buggy without consistency support in DB
2. better network availability, CP (from CAP) choice is less relevant, availability sacrifice hardly noticeable
3. CAP asymmetry: CP can guarantee consistency, AP can't guarantee availability (only question of degree)

Even stronger consistency requirements.

Most workloads are read heavy. New systems support lock-free consistent reads.