CS162
Operating Systems and Systems Programming
Lecture 24

Distributed 1: Reliability, Transactions, Distributed Decision Making, 2PC

April 18th, 2023

Prof. John Kubiatowicz

http://cs162.eecs.Berkeley.edu

Review: Important “ilities”

- Availability: the probability that the system can accept and process requests
  - Measured in “nines” of probability: e.g. 99.9% probability is “3-nines of availability”
  - Key idea here is independence of failures

- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn’t necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone

- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only “up”, but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

Review: How to Make File Systems more Durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects

- Make sure writes survive in short term
  - Either abandon delayed writes or
  - Use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache

- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…

Review: RAID 6 and other Erasure Codes

- In general: RAIDX is an “erasure code”
  - Must have ability to know which disks are bad
  - Treat missing disk as an “Erasure”
- Today, disks so big that: RAID 5 not sufficient!
  - Time to repair disk sooooo long, another disk might fail in process!
  - “RAID 6” – allow 2 disks in replication stripe to fail
  - Requires more complex erasure code, such as EVENODD code (see readings)
- More general option for general erasure code: Reed-Solomon codes
  - Based on polynomials in GF(2^k) (I.e. k-bit symbols)
  - m data points define a degree m polynomial; encoding is n points on the polynomial
  - Any m points can be used to recover the polynomial; $n - m$ failures tolerated
- Erasure codes not just for disk arrays. For example, geographic replication
  - E.g., split data into $m = 4$ chunks, generate $n = 16$ fragments and distribute across the Internet
  - Any 4 fragments can be used to recover the original data --- very durable!

File System Reliability: (Difference from Block-level reliability)

- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete

- Having RAID doesn't necessarily protect against all such failures
  - No protection against writing bad state
  - What if one disk of RAID group not written?
- File system needs durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

- But durability is not quite enough…!

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - – inode, indirect block, data block, bitmap, …
  - – With sector remapping, single update to physical disk block can require multiple (even lower level) updates to sectors
- At a physical level, operations complete one at a time
  - – Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an inconsistent state
  - Example: transfer funds from one bank account to another
  - What if transfer is interrupted after withdrawal and before deposit?

- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Reliability Approach #1: Careful Ordering

- Sequence operations in a specific order
  - Careful design to allow sequence to be interrupted safely
  - Data block ⇐ inode ⇐ free ⇐ directory

- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed

- Approach taken by
  - FAT and FFS (fsck) to protect filesystem structure/metadata
  - Many app-level recovery schemes (e.g., Word, emacs autosaves)

Berkeley FFS: Create a File

Normal operation:

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery:

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found dir
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times

Time proportional to disk size

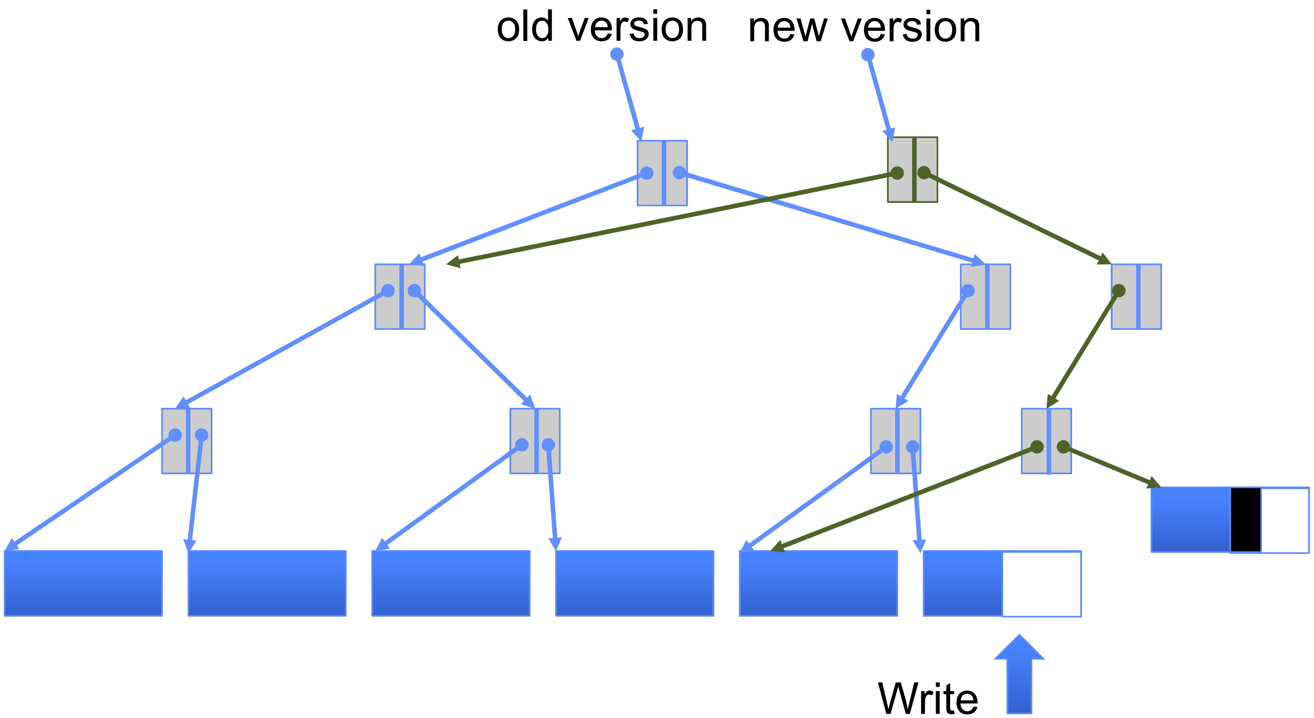
Reliability Approach #2: Copy on Write File Layout

- Recall: multi-level index structure lets us find the data blocks of a file
- Instead of over-writing existing data blocks and updating the index structure:
  - Create a new version of the file with the updated data
  - Reuse blocks that don't change much of what is already in place
  - This is called: Copy On Write (COW)

- Seems expensive! But
  - Updates can be batched
  - Almost all disk writes can occur in parallel

- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

COW with Smaller-Radix Blocks



- If file represented as a tree of blocks, just need to update the leading fringe

Example: ZFS and OpenZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

Administrivia

- Midterm 3: Next Thursday!
  - No class on day of midterm
  - Three double-sided pages of notes
  - Watch for Ed post about where you should go: we have multiple exam rooms
  - Confict request form due Thursday!
- All material up to next Tuesday's lecture is fair game
- Final deadlines during RRR week:
  - Yes, there will be some office hours – watch for specifics
- Extra "fun" lecture on Tuesday of RRR week!

https://tinyurl.com/mby6f47t

More General Reliability Solutions

- Use Transactions for atomic updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either all or none of the updates
  - Most modern file systems use transactions internally to update filesystem structures and metadata
  - Many applications implement their own transactions

- Provide Redundancy for media failures
  - Redundant representation on media (Error Correcting Codes)
  - Replication across media (e.g., RAID disk array)

Transactions

- Closely related to critical sections for manipulating shared data structures

- They extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures

- Many ad-hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error (fsck)
  - Applications use temporary files and rename

Key Concept: Transaction

- A *transaction* is an atomic sequence of reads and writes that takes the system from consistent state to another.

consistent state 1

transaction

consistent state 2

- Recall: Code in a critical section appears atomic to other threads
- Transactions extend the concept of atomic updates from *memory* to *persistent storage*

Typical Structure

- Begin a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

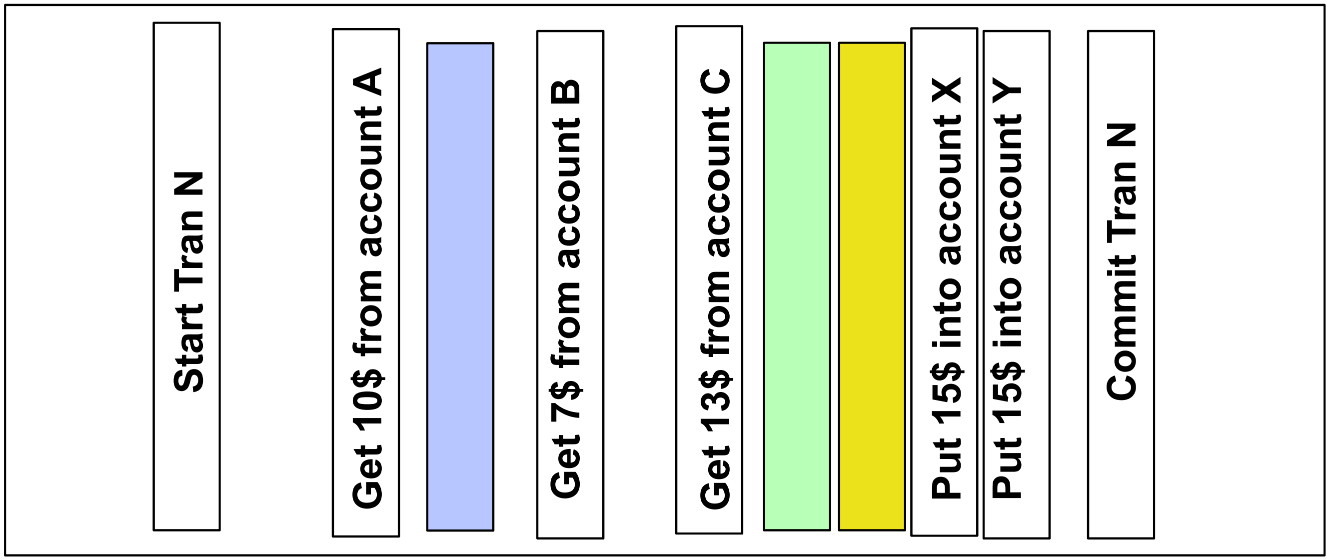
“Classic” Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00 WHERE
   name = 'Alice';

 UPDATE branches SET balance = balance - 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Alice');

 UPDATE accounts SET balance = balance + 100.00 WHERE
   name = 'Bob';

 UPDATE branches SET balance = balance + 100.00 WHERE
   name = (SELECT branch_name FROM accounts WHERE name
   = 'Bob');
COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice’s account to Bob’s account

Concept of a log

- One simple action is atomic – write/append a basic item
- Use that to seal the commitment to a whole series of actions



Start Tran N

Get 10$ from account A

Get 7$ from account B

Get 13$ from account C

Put 15$ into account X

Put 15$ into account Y

Commit Tran N

Transactional File Systems

- Better reliability through use of log
  - Changes are treated as transactions
  - A transaction is committed once it is written to the log
    - Data forced to disk for reliability
    - Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log

- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery

Journaling File Systems

- Don’t modify data structures on disk directly
- Write each update as transaction recorded in a log
  - Commonly called a journal or intention list
  - Also maintained on disk (allocate blocks for it when formatting)
- Once changes are in the log, they can be safely applied to file system
  - e.g. modify inode pointers and directory mapping
- Garbage collection: once a change is applied, remove its entry from the log

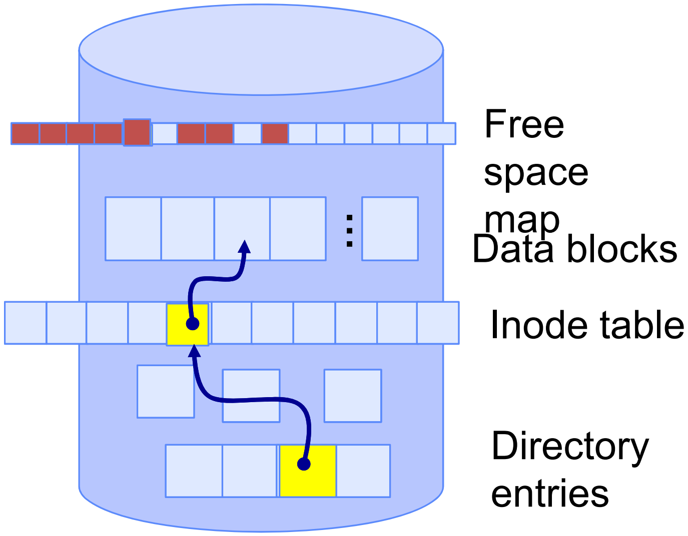
- Linux took original FFS-like file system (ext2) and added a journal to get ext3!
  - Some options: whether or not to write all data to journal or just metadata

- Other examples: NTFS, Apple HFS+/apfs, Linux XFS, JFS, ext4

Creating a File (No Journaling Yet)

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

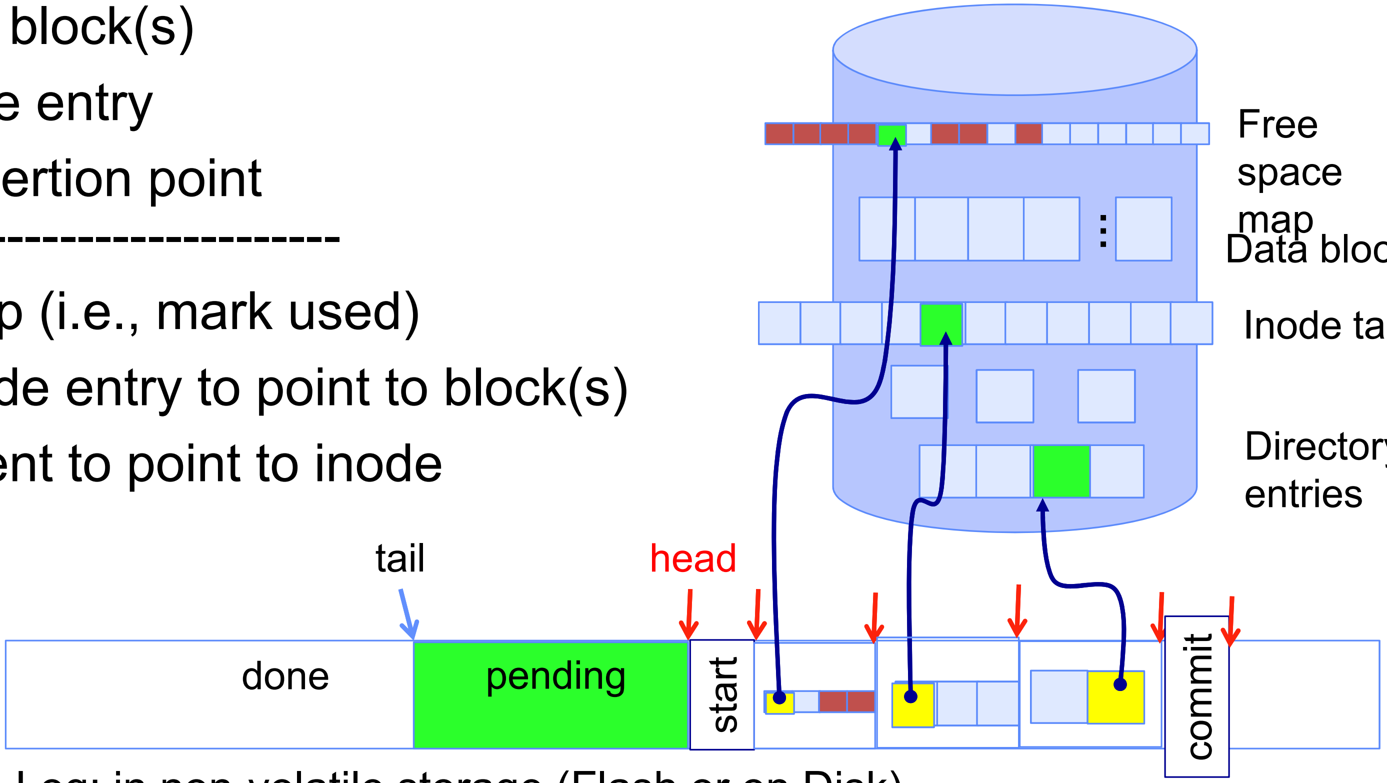
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



Creating a File (With Journaling)

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

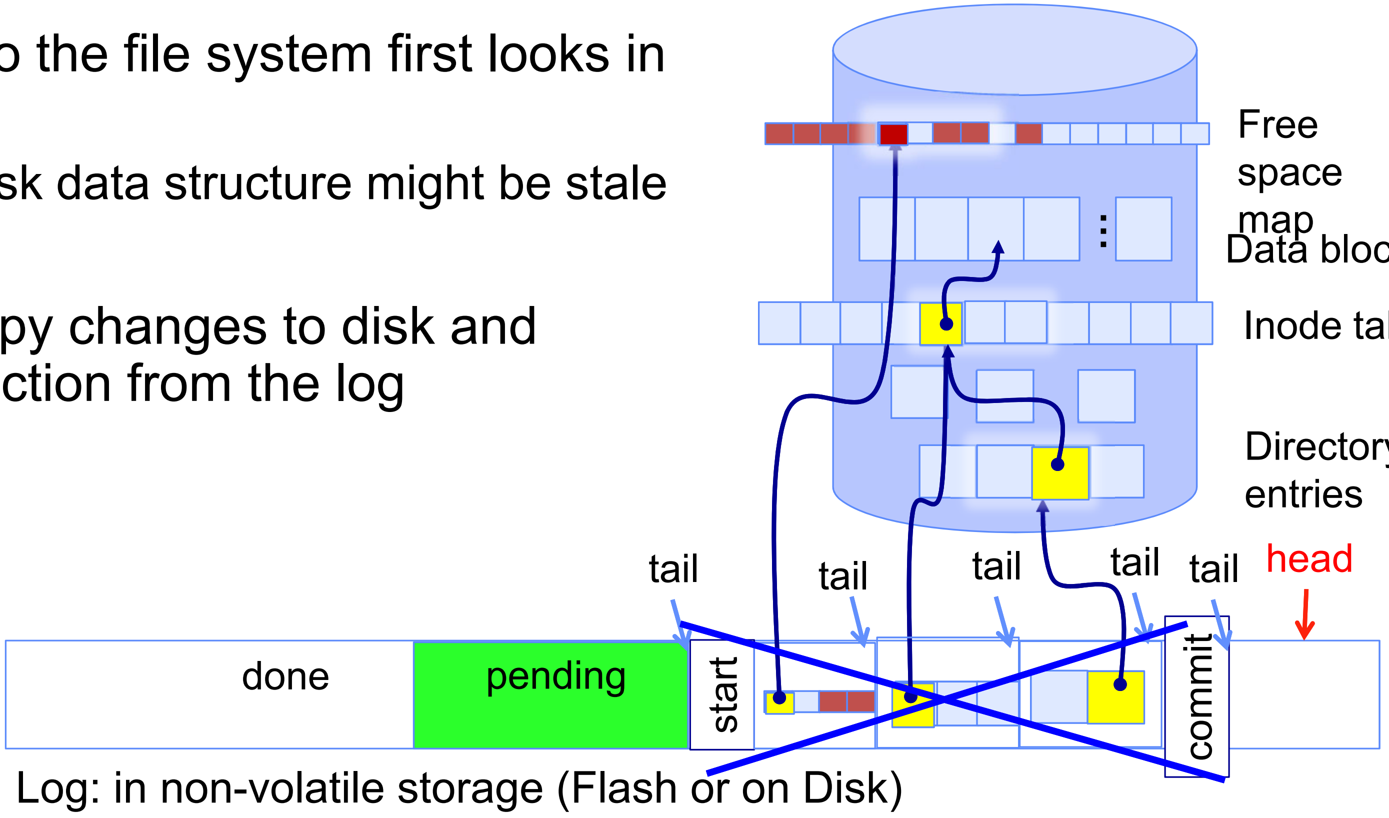
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



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

After Commit, Eventually Replay Transaction

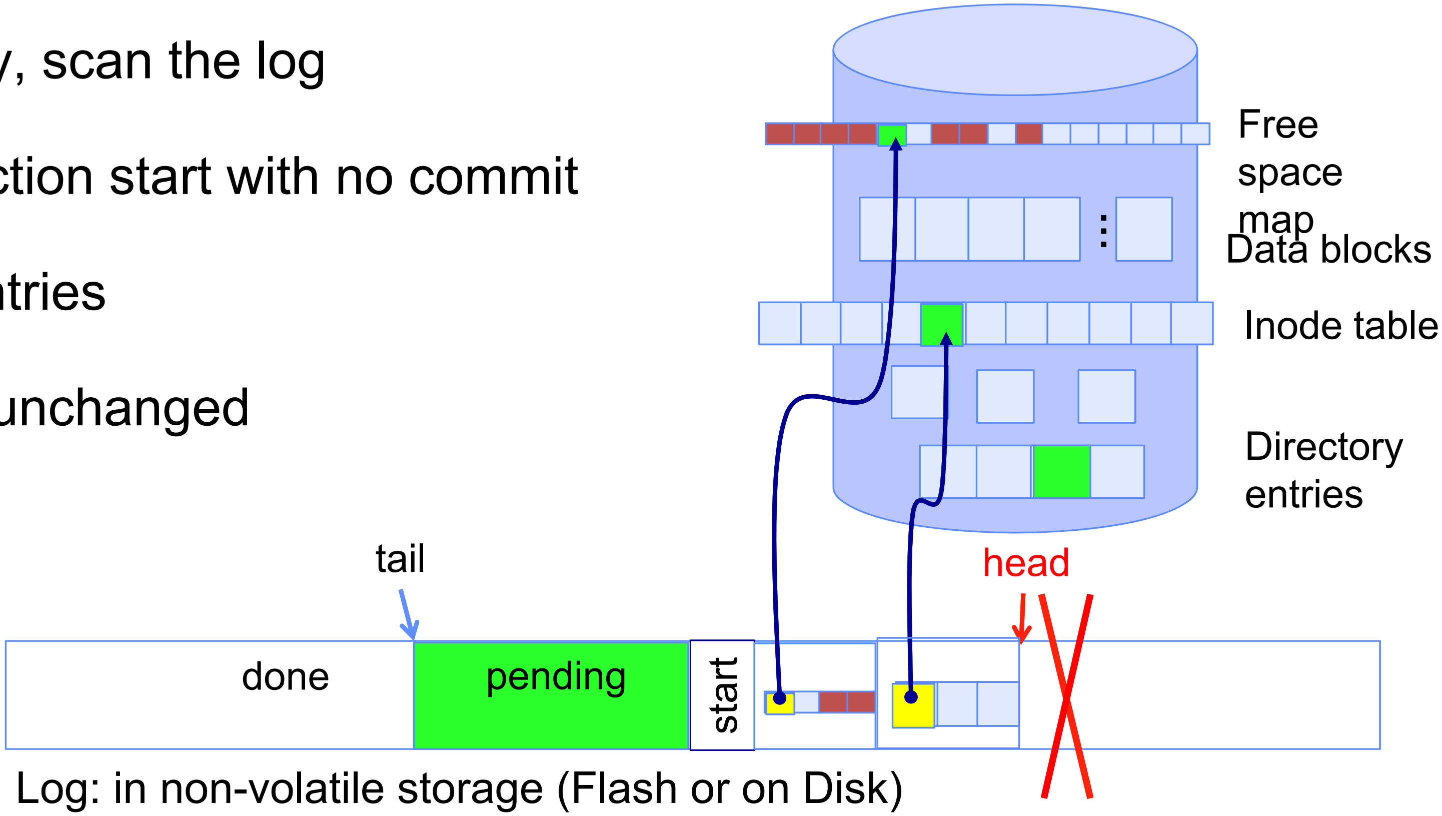
- All accesses to the file system first looks in the log
  - Actual on-disk data structure might be stale
- Eventually, copy changes to disk and discard transaction from the log



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

Crash Recovery: Discard Partial Transactions

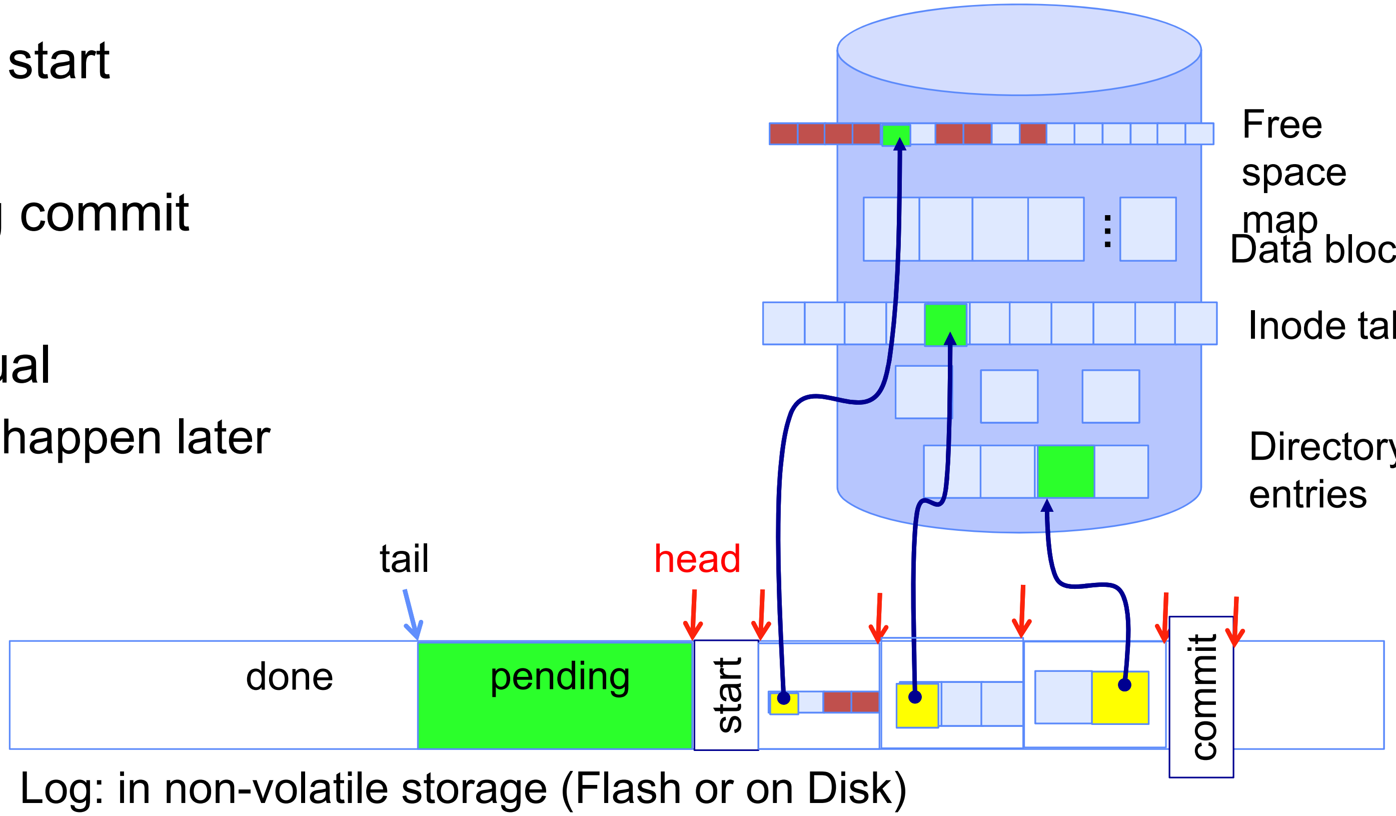
- Upon recovery, scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



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

Crash Recovery: Keep Complete Transactions

- Scan log, find start
- Find matching commit
- Redo it as usual
  - Or just let it happen later



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

Journaling Summary

Why go through all this trouble?

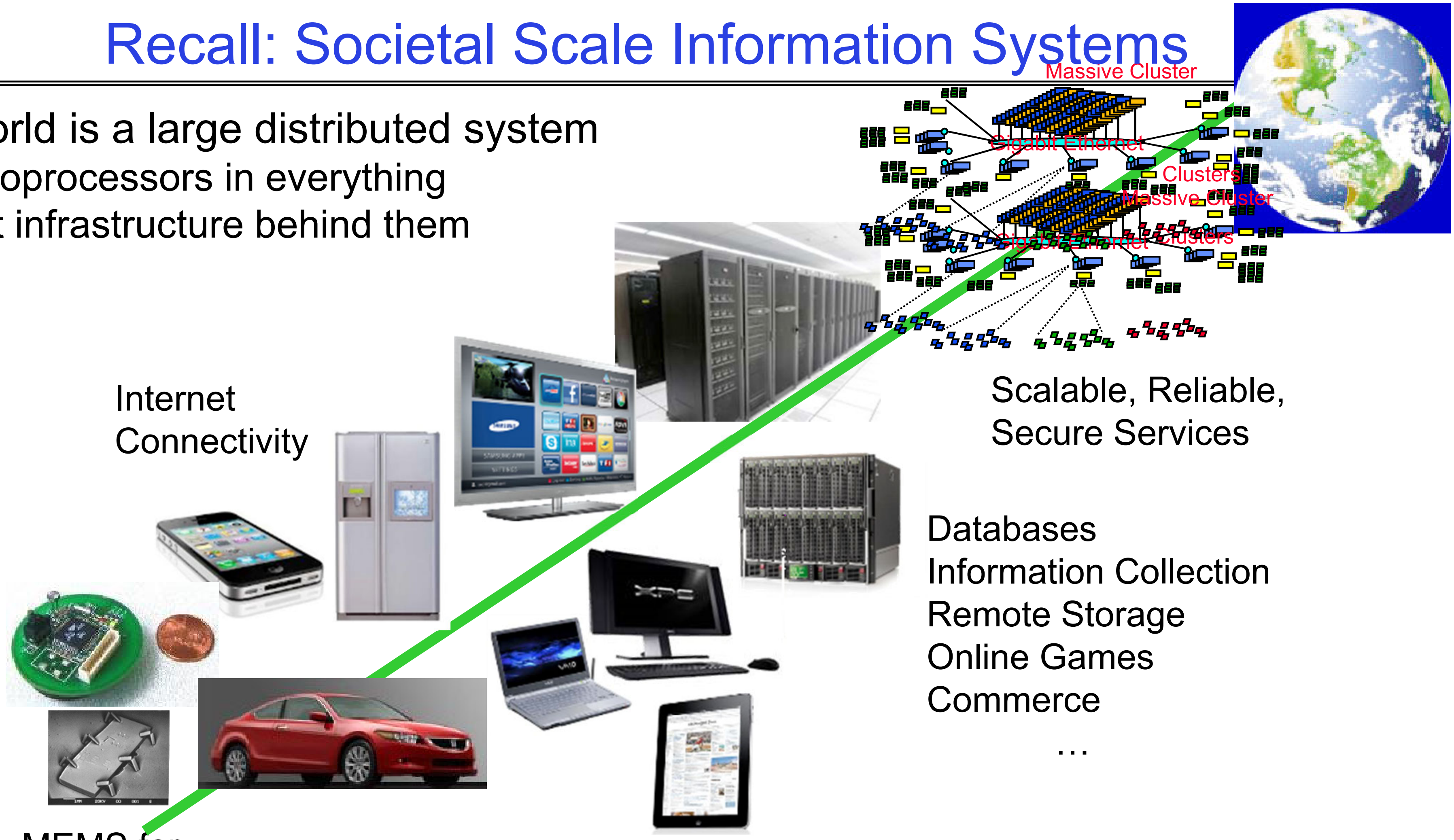
- Updates atomic, even if we crash:
  - Update either gets fully applied or discarded
  - All physical operations *treated as a logical unit*

Isn’t this expensive?

- Yes! We're now writing all data twice (once to log, once to actual data blocks in target file)
- Modern filesystems journal metadata updates only
  - Record modifications to file system data structures
  - But apply updates to a file’s contents directly

Recall: Societal Scale Information Systems

- The world is a large distributed system
  - Microprocessors in everything
  - Vast infrastructure behind them



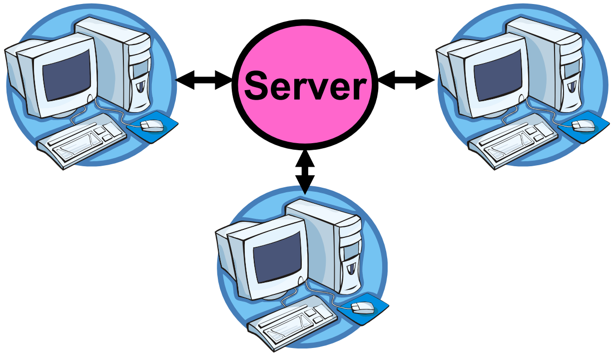
Scalable, Reliable,
Secure Services

Databases
Information Collection
Remote Storage
Online Games
Commerce
…

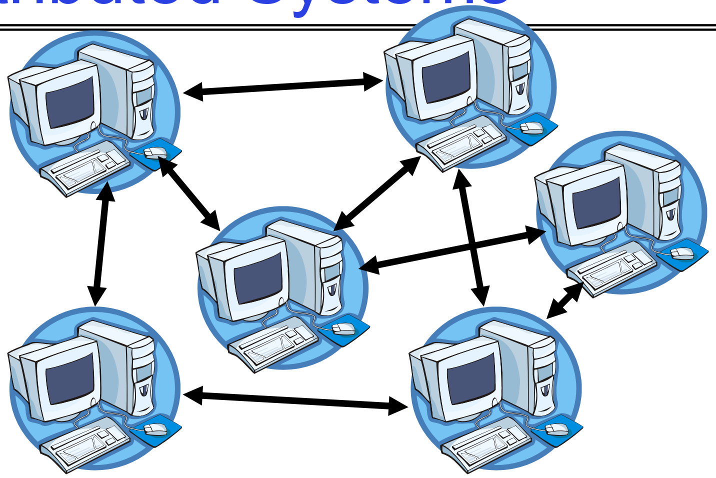
Internet
Connectivity

MEMS for
Sensor Nets

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

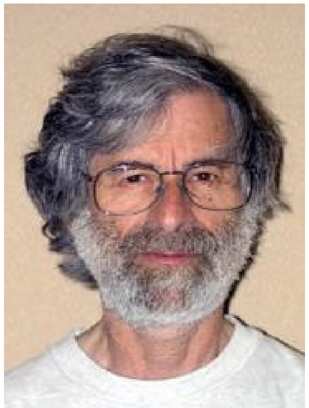
- Centralized System: major functions performed by a single physical computer
  - – Originally, everything on single computer
  - – Later: client/server model
- Distributed System: physically separate computers working together on task
  - – Early model: multiple servers working together
    - » Probably in the same room or building
    - » Often called a “cluster”
  - – Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues/Promise

- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: much easier for users to collaborate through network resources (such as network file systems)

- The *promise* of distributed systems:
  - *Higher availability*: one machine goes down, use another
  - *Better durability*: store data in multiple locations
  - *More security*: each piece easier to make secure

Distributed Systems: Reality

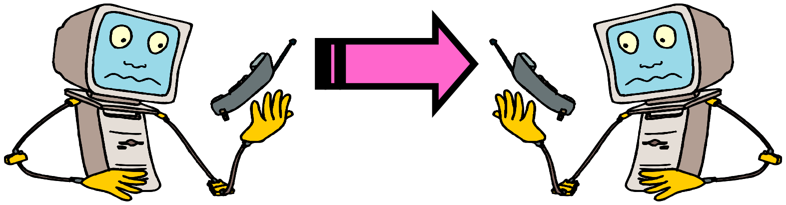


Leslie Lamport

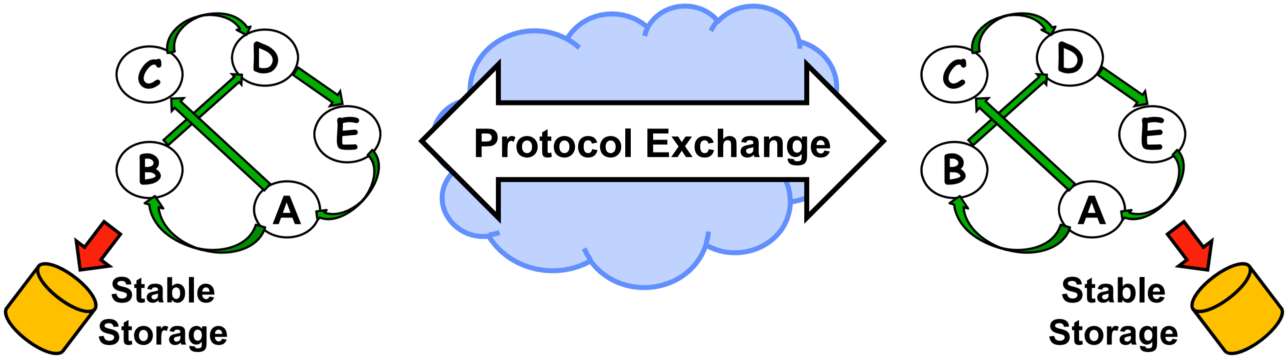
- Reality has been disappointing
  - *Worse availability*: depend on every machine being up
    - Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
  - *Worse reliability*: can lose data if any machine crashes
  - *Worse security*: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information
  - What would be easy in a centralized system becomes a lot more difficult
- Trust/Security/Privacy/Denial of Service
  - Many new variants of problems arise as a result of distribution
  - Can you trust the other members of a distributed application enough to even perform a protocol correctly?
  - Corollary of Lamport's quote: "A distributed system is one where you can't do work because some computer you didn't even know existed is successfully coordinating an attack on my system!"

Distributed Systems: Goals/Requirements

- Transparency: the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - Location: Can't tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can't tell how many copies of resource exist
  - Concurrency: Can't tell how many users there are
  - Parallelism: System may speed up large jobs by splitting them into smaller pieces
  - Fault Tolerance: System may hide various things that go wrong
- Transparency and collaboration require some way for different processors to communicate with one another



How do entities communicate? A Protocol!



- A protocol is an agreement on how to communicate, including:
  - – 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
- Described formally by a state machine
  - – Often represented as a message transaction diagram
  - – Can be a partitioned state machine: two parties synchronizing duplicate sub-state machines between them
  - – Stability in the face of failures!

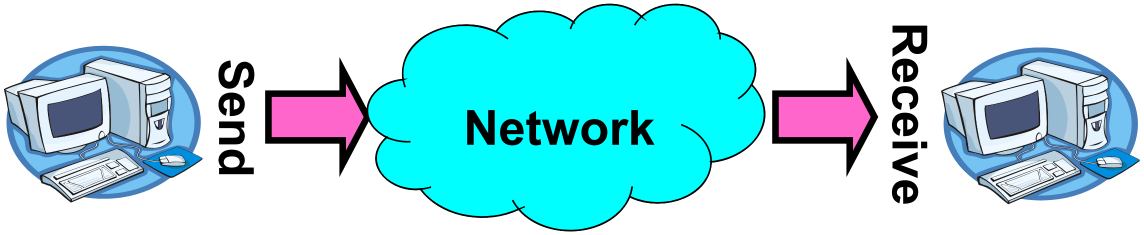
Examples of Protocols in Human Interactions

- Telephone
  1. (Pick up / open up the phone)
  2. Listen for a dial tone / see that you have service
  3. Dial
  4. Should hear ringing …
  5. Callee: “Hello?”
  6. Caller: “Hi, it’s Anthony….”
     Or: “Hi, it’s me” (← what’s *that* about?)
  7. Caller: “Hey, do you think … blah blah blah …” **pause**

  1. Callee: “Yeah, blah blah blah …” **pause**
  2. Caller: Bye
  3. Callee: Bye
  4. Hang up

Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
    - » Over Internet, destination specified by IP address and Port (Recall Web server example!)
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - – When receiver gets message? (i.e. ack received)
  - – When message is safely buffered on destination?
  - – Right away, if message is buffered on source node?
- Actually two questions here:
  - – When can the sender be sure that receiver actually received the message?
  - – When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - – T1→buffer→T2
  - – Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
    int msg1[1000];
    while(1) {
        prepare message;
        send(msg1,mbox);
    }
```

Send Message

```
Consumer:
    int buffer[1000];
    while(1) {
        receive(buffer,mbox);
        process message;
    }
```

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - This is one of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - Read a file stored on a remote machine
    - Request a web page from a remote web server
  - Also called: client-server
    - Client ≡ requester, Server ≡ responder
    - Server provides “service” (file storage) to the client
- Example: File service

```
Client: (requesting the file)
  char response[1000];

  send(“read rutabaga”, server_mbox);
  receive(response, client_mbox);

Server: (responding with the file)
  char command[1000], answer[1000];

  receive(command, server_mbox);
  decode command;
  read file into answer;
  send(answer, client_mbox);
```

Request File

Get Response

Receive Request

Send Response

Distributed Consensus Making

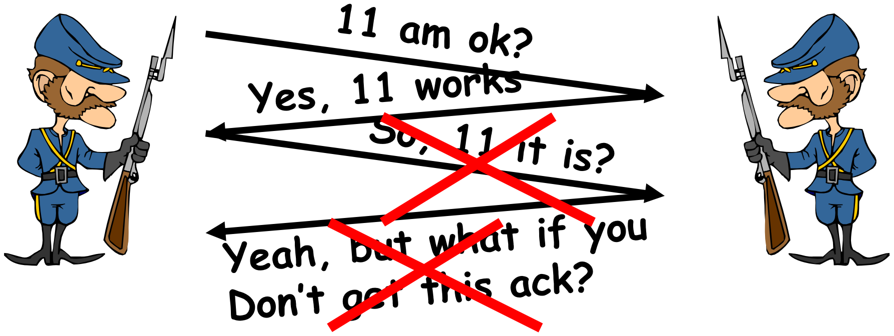
- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values
- Distributed Decision Making
  - Choose between “true” and “false”
  - Or Choose between “commit” and “abort”
- Equally important (but often forgotten!): make it durable!
  - How do we make sure that decisions cannot be forgotten?
    - This is the “D” of “ACID” in a regular database
  - In a global-scale system?
    - What about erasure coding or massive replication?
    - Like BlockChain applications!

General's Paradox

- General's paradox:
  - – Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - – Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - – Named after Custer, who died at Little Big Horn because he arrived a couple of days too early

General’s Paradox (con’t)

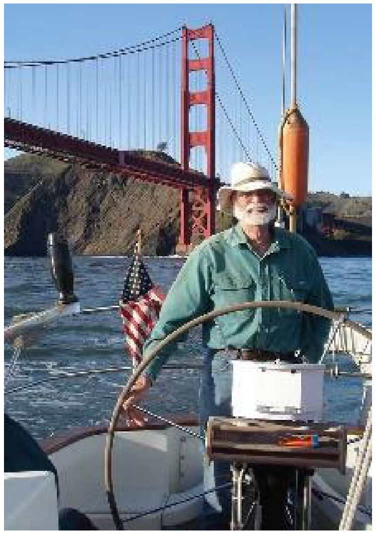
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, “no”, even if all messages get through



  - No way to be sure last message gets through!
  - In real life, use radio for simultaneous (out of band) communication
- So, clearly, we need something other than simultaneity!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
- Distributed transaction: Two or more machines agree to do something, or not do it, atomically
  - No constraints on time, just that it will eventually happen!
- Two-Phase Commit protocol: Developed by Turing award winner Jim Gray
  - (first Berkeley CS PhD, 1969)
  - Many important DataBase breakthroughs also from Jim Gray



Jim Gray

Two-Phase Commit Protocol

- Persistent stable log on each machine: keep track of whether commit has happened
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
- Prepare Phase:
  - The global coordinator requests that all participants will promise to commit or rollback the transaction
  - Participants record promise in log, then acknowledge
  - If anyone votes to abort, coordinator writes `"Abort"` in its log and tells everyone to abort; each records `"Abort"` in log
- Commit Phase:
  - After all participants respond that they are prepared, then the coordinator writes `"Commit"` to its log
  - Then asks all nodes to commit; they respond with ACK
  - After receive ACKs, coordinator writes `"Got Commit"` to log
- Log used to guarantee that all machines either commit or don’t

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
- Use a persistent, stable log on each machine to keep track of what you are doing
  - If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash

Two-Phase Commit: Setup

- One machine *(coordinator)* initiates the protocol
- It asks *every* machine to **vote** on transaction

- Two possible votes:
  - **Commit**
  - **Abort**

- Commit transaction only if unanimous approval

Two-Phase Commit: Preparing

Worker Agrees to Commit

- Machine has **guaranteed** that it will accept transaction
- Must be **recorded in log** so machine will remember this decision if it fails and restarts

Worker Agrees to Abort

- Machine has **guaranteed** that it will **never accept** this transaction
- Must be **recorded in log** so machine will remember this decision if it fails and restarts

Two-Phase Commit: Finishing

Commit Transaction

- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least one machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

Two-Phase Commit: Finishing

Commit Transaction

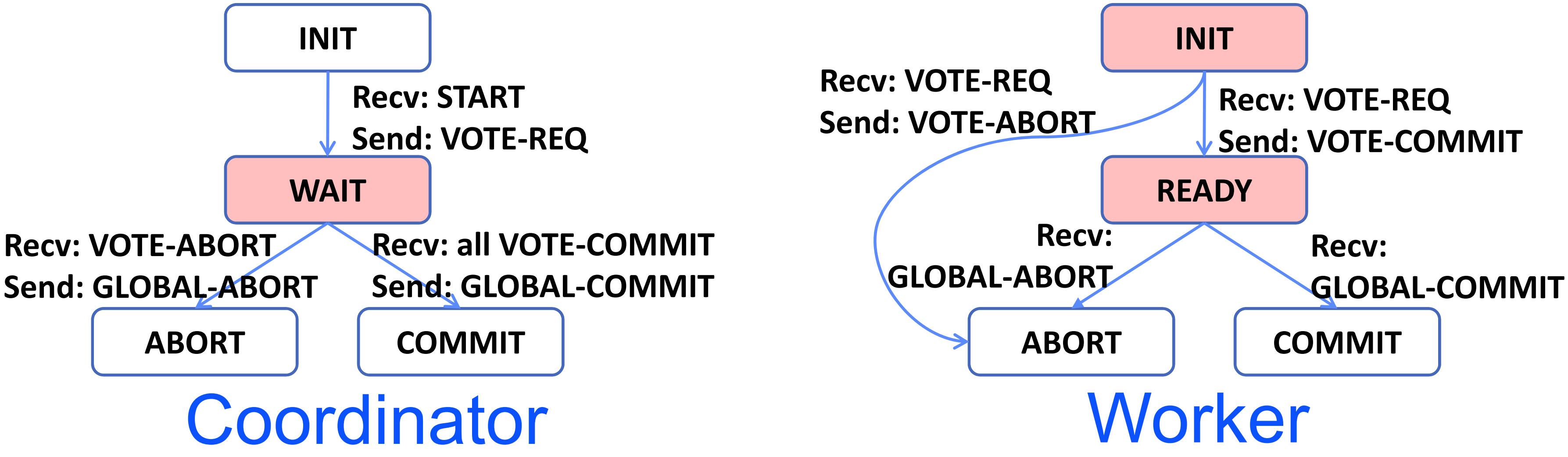
- Coordinator learns *all machines have agreed to commit*
- Record decision to commit in local log
- Apply transaction, inform voters

Abort Transaction

- Coordinator learns *at least one machine has voted to abort*
- Record decision to abort in local log
- Do not apply transaction, inform voters

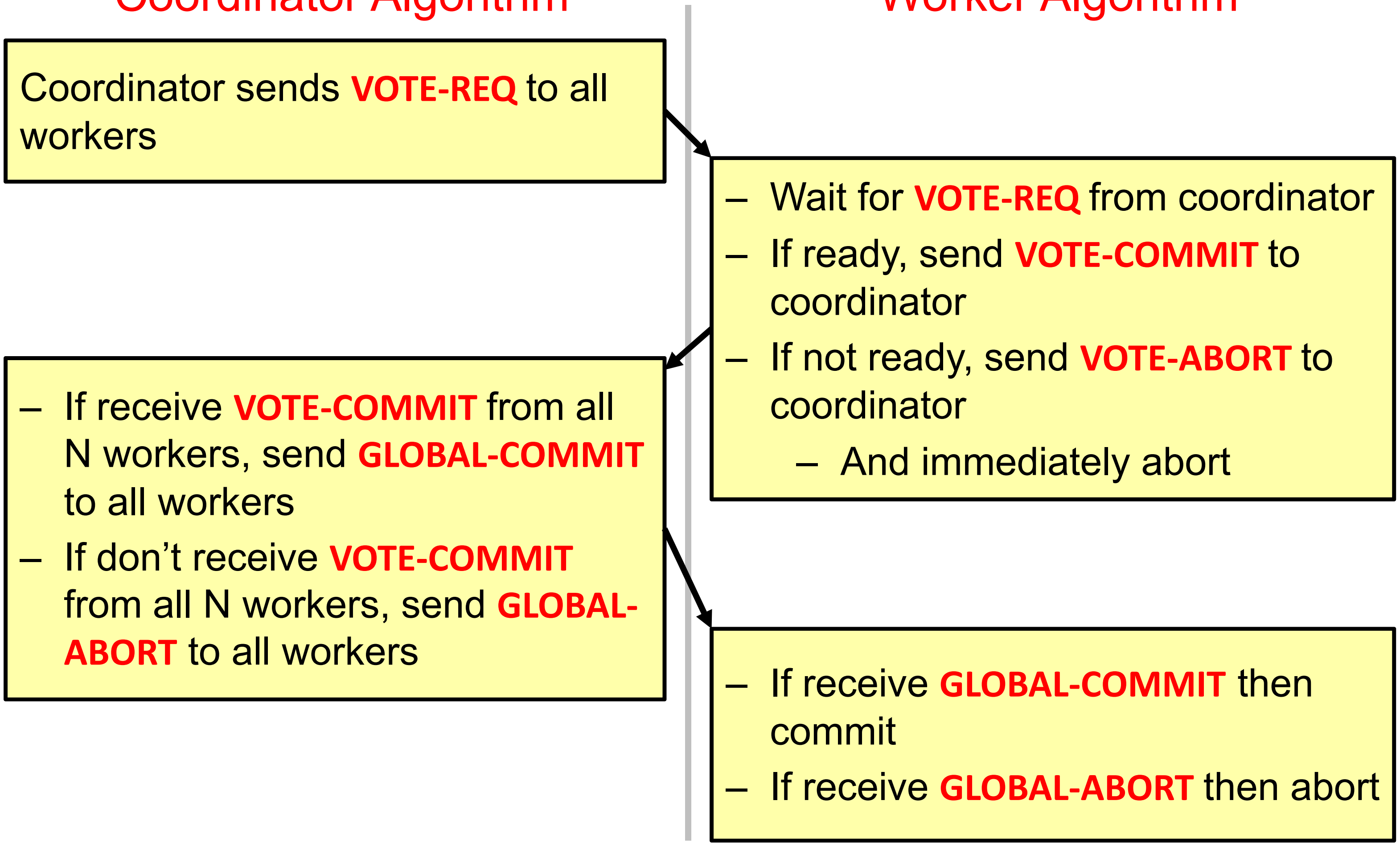
Because no machine can take back its decision, exactly one of these will happen

State Machine Description of 2PC



- Two Phase Commit (2PC) can be described with interacting state machines
- Coordinator only waits for votes in “WAIT” state
  - In WAIT, if doesn’t receive N votes, it times out and sends GLOBAL-ABORT
- Worker waits for VOTE-REQ in INIT
  - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
  - Coordinator fails ⇒ workers BLOCK waiting for coordinator to recover and send GLOBAL_* message

Detailed Algorithm



Coordinator Algorithm

Coordinator sends **VOTE-REQ** to all workers

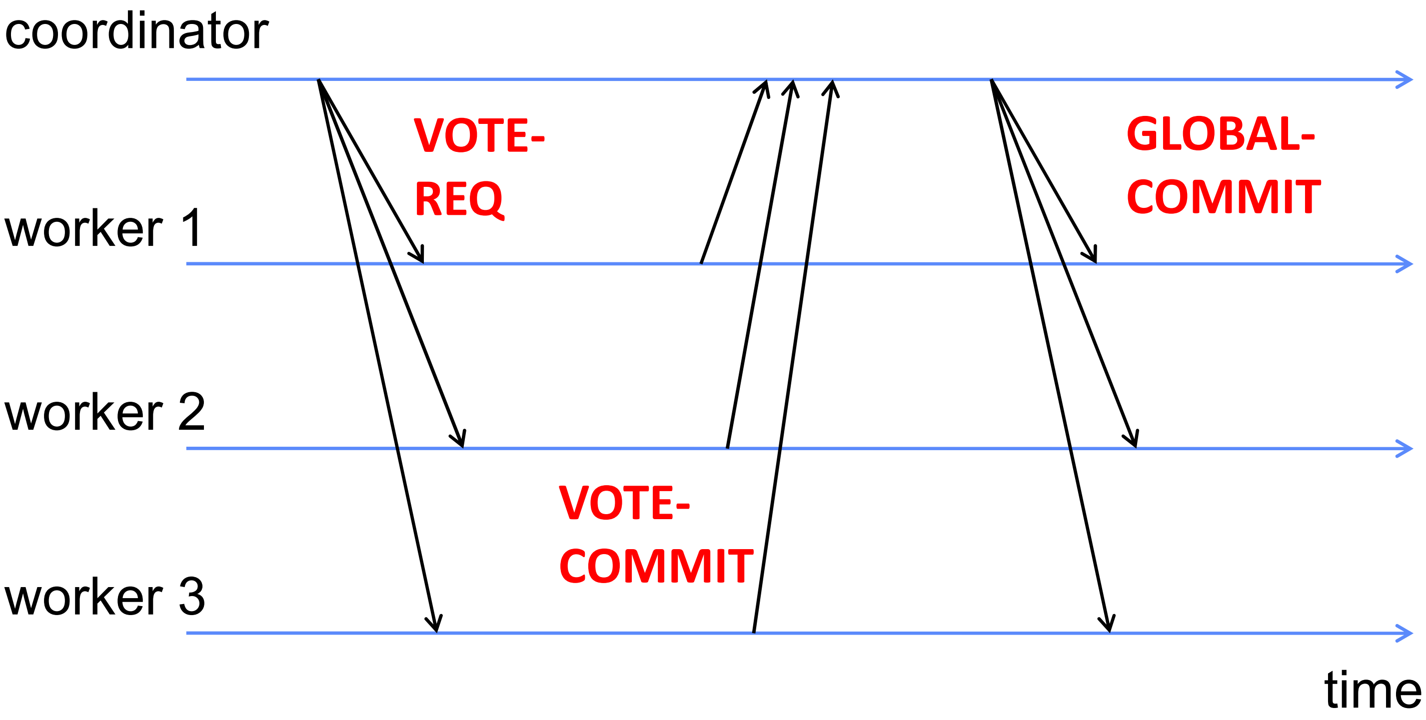
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If don't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

Worker Algorithm

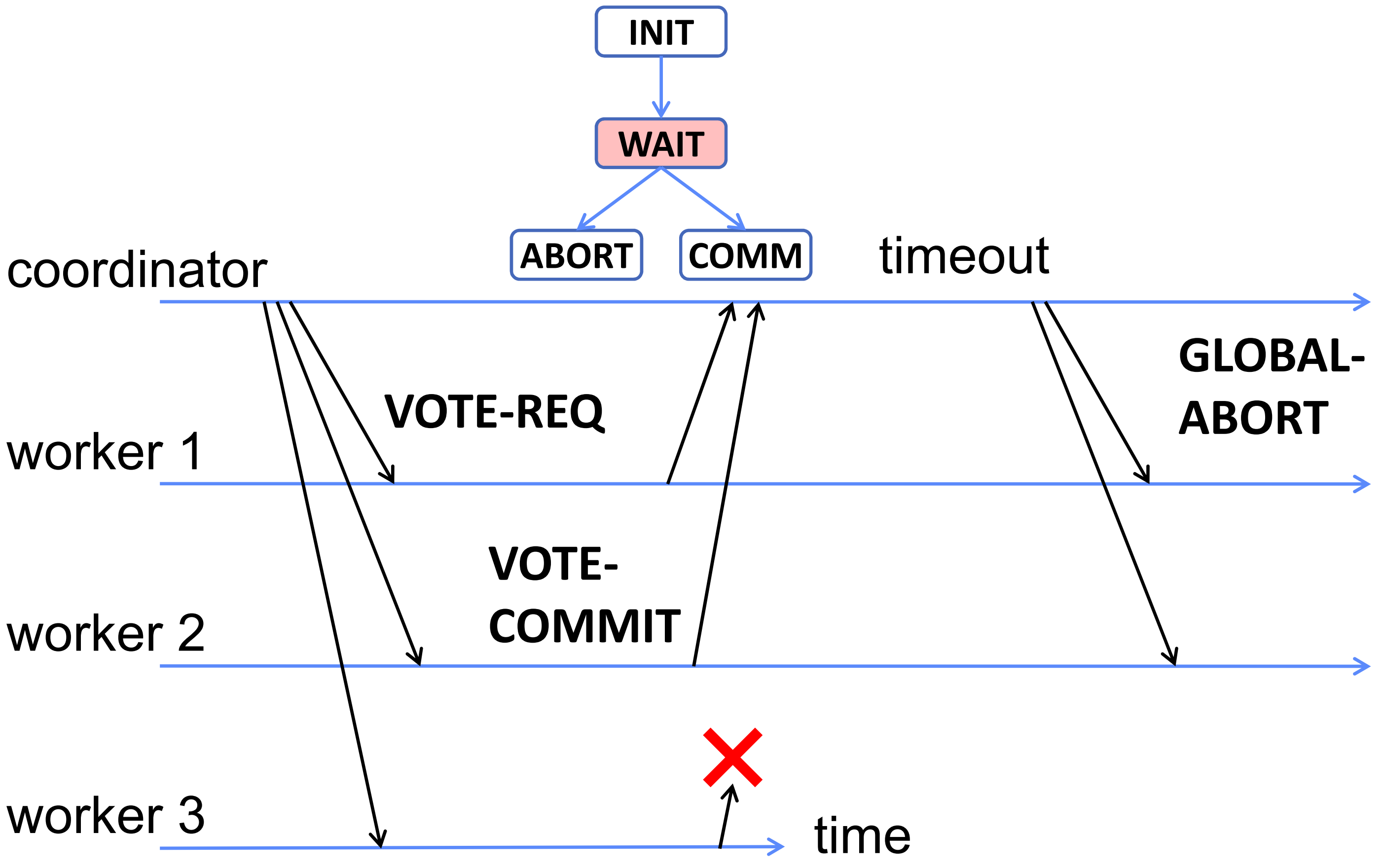
- Wait for **VOTE-REQ** from coordinator
- If ready, send **VOTE-COMMIT** to coordinator
- If not ready, send **VOTE-ABORT** to coordinator
  - And immediately abort

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

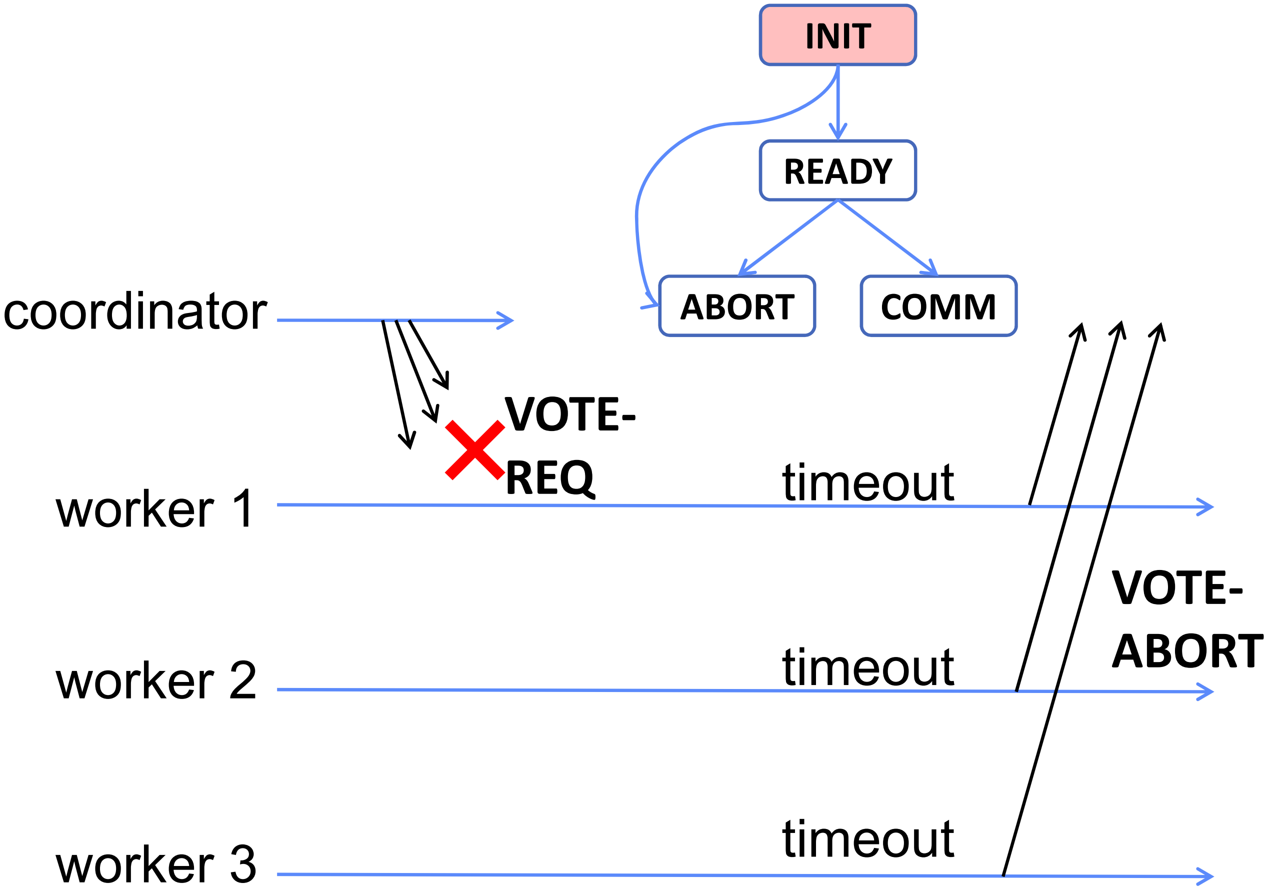
Failure Free Example Execution



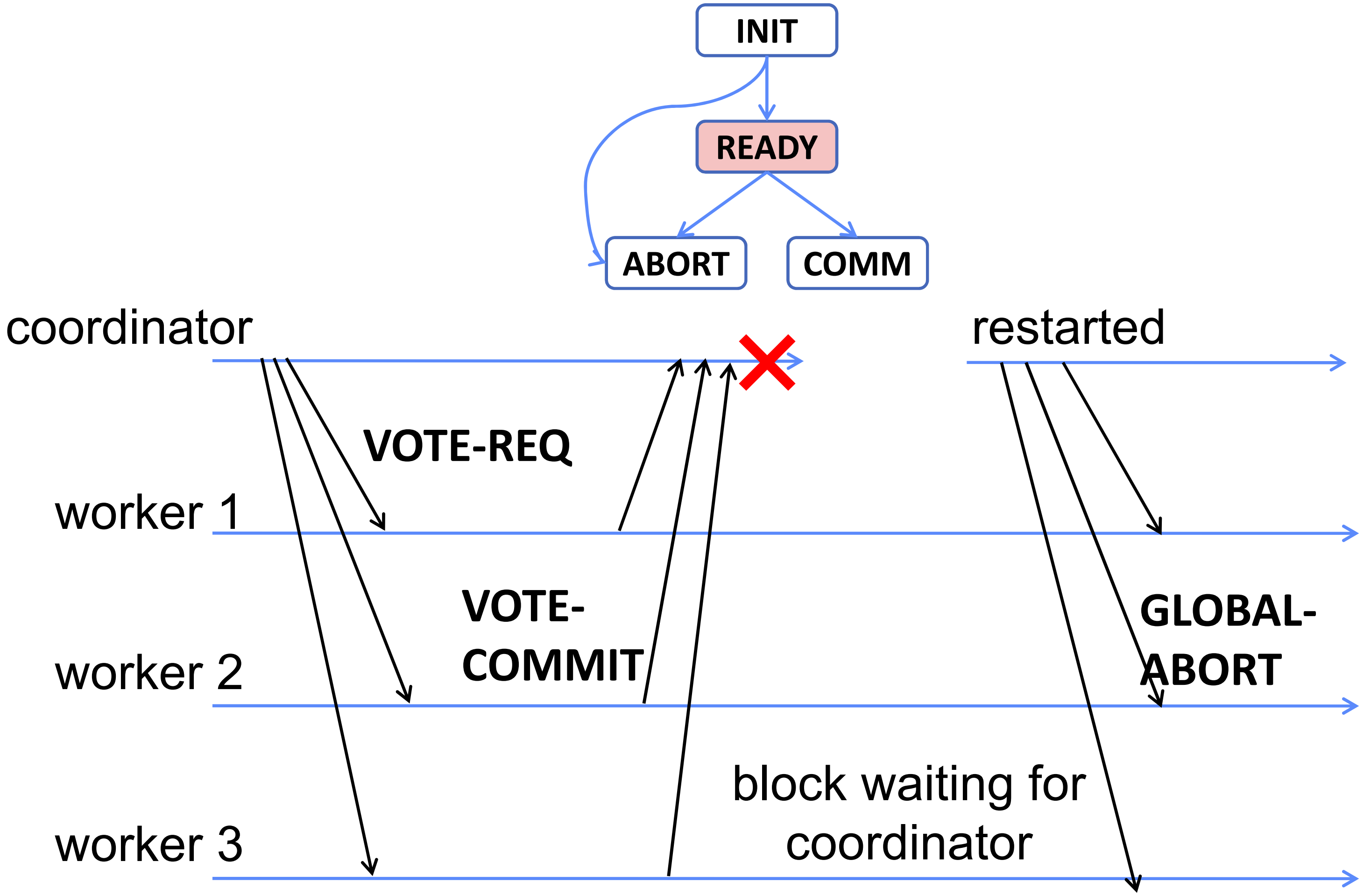
Example of Worker Failure



Example of Coordinator Failure #1



Example of Coordinator Failure #2



Durability

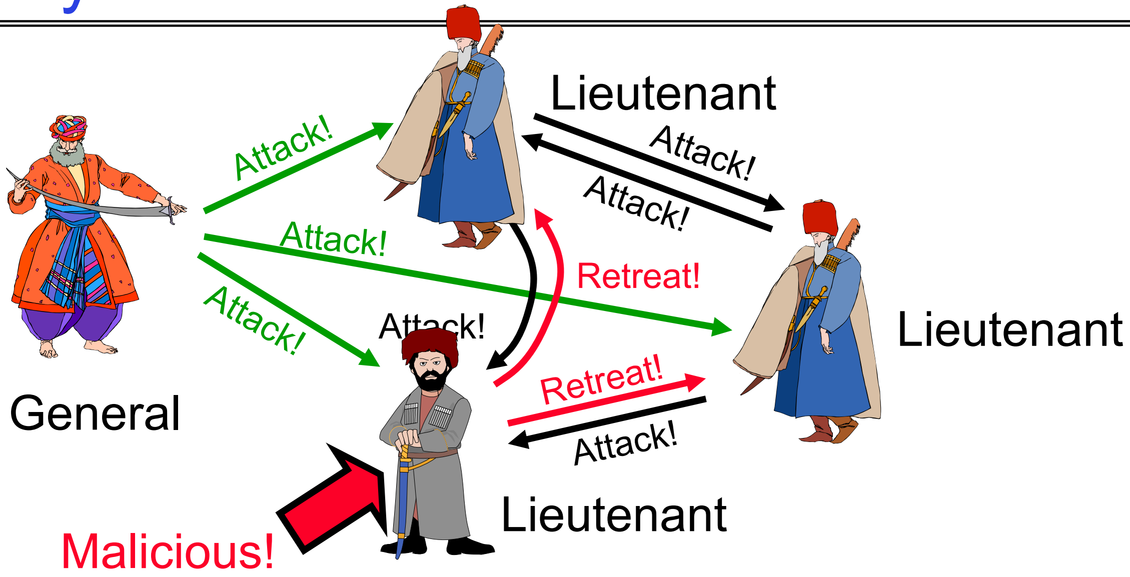
- All nodes use stable storage to store current state
  - stable storage is non-volatile storage (e.g. backed by disk) that guarantees atomic writes.
  - E.g.: SSD, NVRAM
- Upon recovery, nodes can restore state and resume:
  - Coordinator aborts in INIT, WAIT, or ABORT
  - Coordinator commits in COMMIT
  - Worker aborts in INIT, ABORT
  - Worker commits in COMMIT
  - Worker "asks" Coordinator in READY

Alternatives to 2PC

- Three-Phase Commit: One more phase, allows nodes to fail or block and still make progress.
- PAXOS: An alternative used by Google and others that does not have 2PC blocking problem
  - Develop by Leslie Lamport (Turing Award Winner)
  - No fixed leader, can choose new leader on fly, deal with failure
  - Some think this is extremely complex!
- RAFT: PAXOS alternative from John Osterhout (Stanford)
  - Simpler to describe complete protocol

- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making
  - Use a more hardened decision making process: Byzantine Agreement and Block Chains

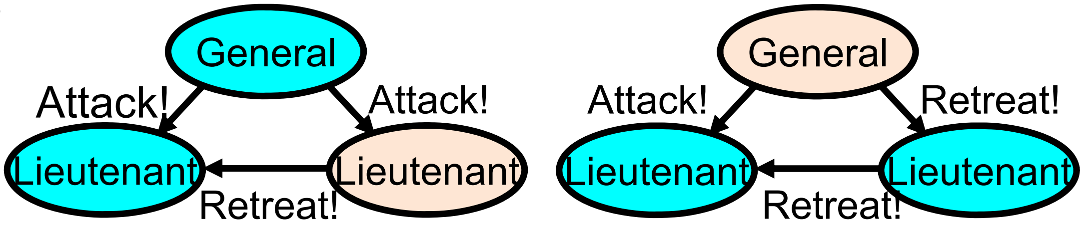
Byzantine General's Problem



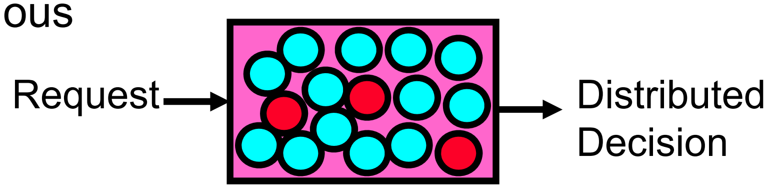
- Byazantine General's Problem (n players):
  - One General and n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that the following Integrity Constraints apply:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

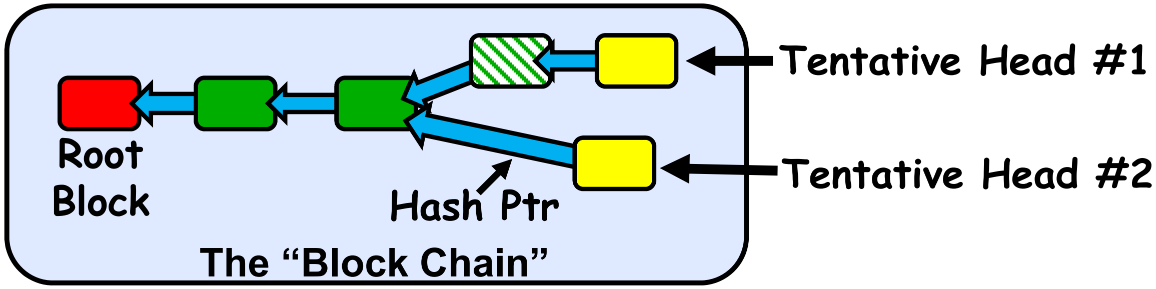
- Impossibility Results:
  - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things



  - With f faults, need n > 3f to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity O(n^2)
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious

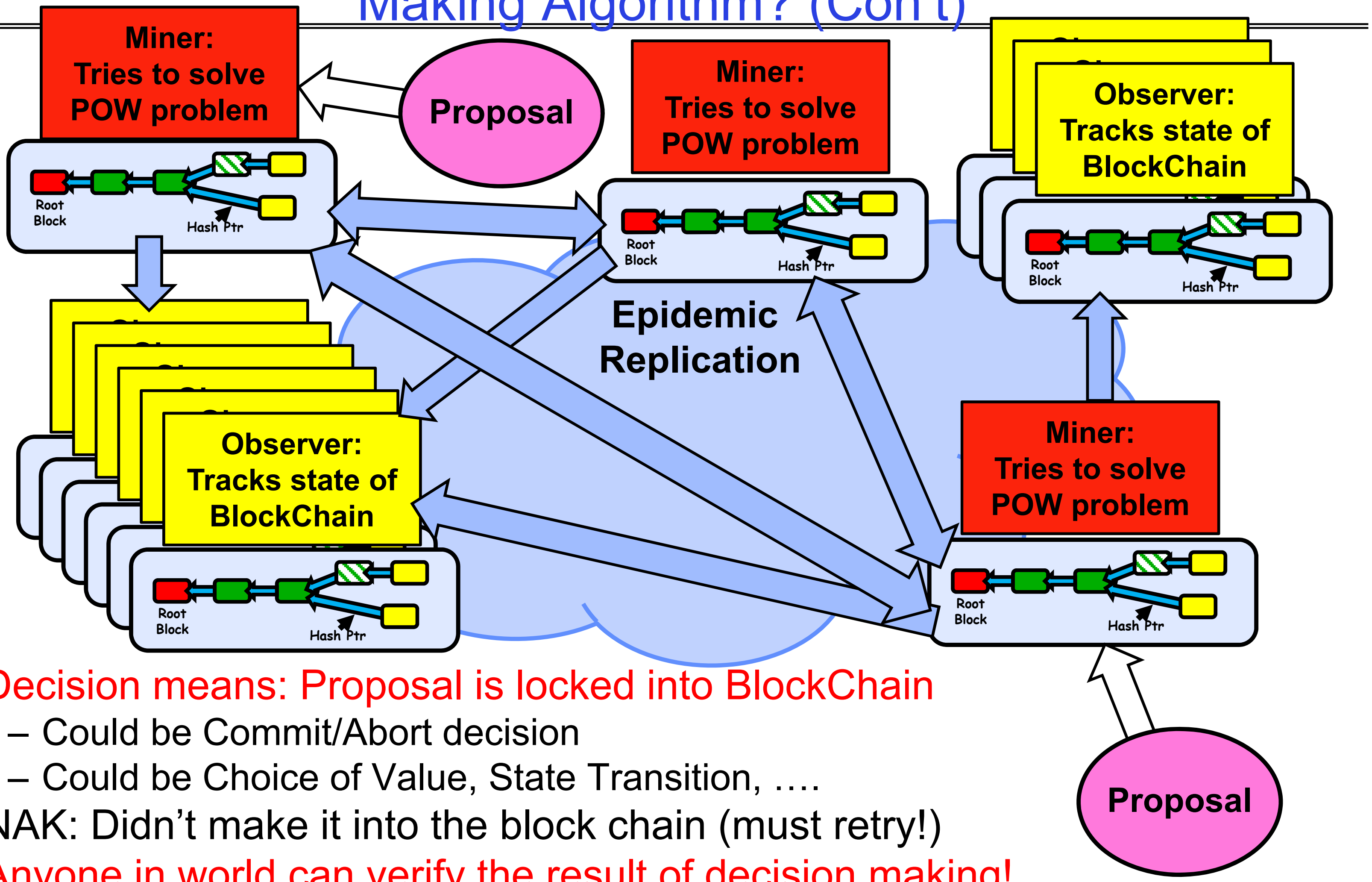


Is a BlockChain a Distributed Decision Making Algorithm?



- BlockChain: a chain of blocks connected by hashes to root block
  - – The Hash Pointers are unforgeable (assumption)
  - – The Chain has no branches except perhaps for heads
  - – Blocks are considered "authentic" part of chain when they have authenticity info in them
- How is the head chosen?
  - – Some consensus algorithm
  - – In many BlockChain algorithms (e.g. BitCoin, Ethereum), the head is chosen by solving hard problem
    - » This is the job of "miners" who try to find "nonce" info that makes hash over block have specified number of zero bits in it
    - » The result is a "Proof of Work" (POW)
    - » Selected blocks above (green) have POW in them and can be included in chains
  - – Longest chain wins

Is a Blockchain a Distributed Decision Making Algorithm? (Con't)



- Decision means: Proposal is locked into BlockChain
  - Could be Commit/Abort decision
  - Could be Choice of Value, State Transition, ….
- NAK: Didn't make it into the block chain (must retry!)
- Anyone in world can verify the result of decision making!

Summary (1/3)

- Copy-on-write provides richer function (versions) with much simpler recovery
  - Little performance impact since sequential write to storage device is nearly free
- Transactions over a log provide a general solution
  - Journaled file systems such as ext3, NTFS
  - Commit sequence to durable log, then update the disk
  - Log takes precedence over disk
  - Replay committed transactions, discard partials

Summary (2/3)

- A protocol is an agreement on how to communicate, including:
  - 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
- Consensus problem
  - All nodes propose a value
  - Some nodes might crash and stop responding
  - Eventually, all remaining nodes decide on the same value from set of proposed values
- Two-phase commit: a form of distributed decision making
  - First, make sure everyone guarantees they will commit if asked (prepare)
  - Next, ask everyone to commit

Summary (3/3)

- Byzantine General's Problem: distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$
- BlockChain protocols:
  - Cryptographically-driven ordering protocol
  - Could be used for distributed decision making