

CSI62
Operating Systems and
Systems Programming
Lecture 21

Filesystem Transactions (Con't),
End-to-End Argument,
Distributed Decision Making

April 16th, 2020

Prof. John Kubiawicz

<http://cs162.eecs.Berkeley.edu>

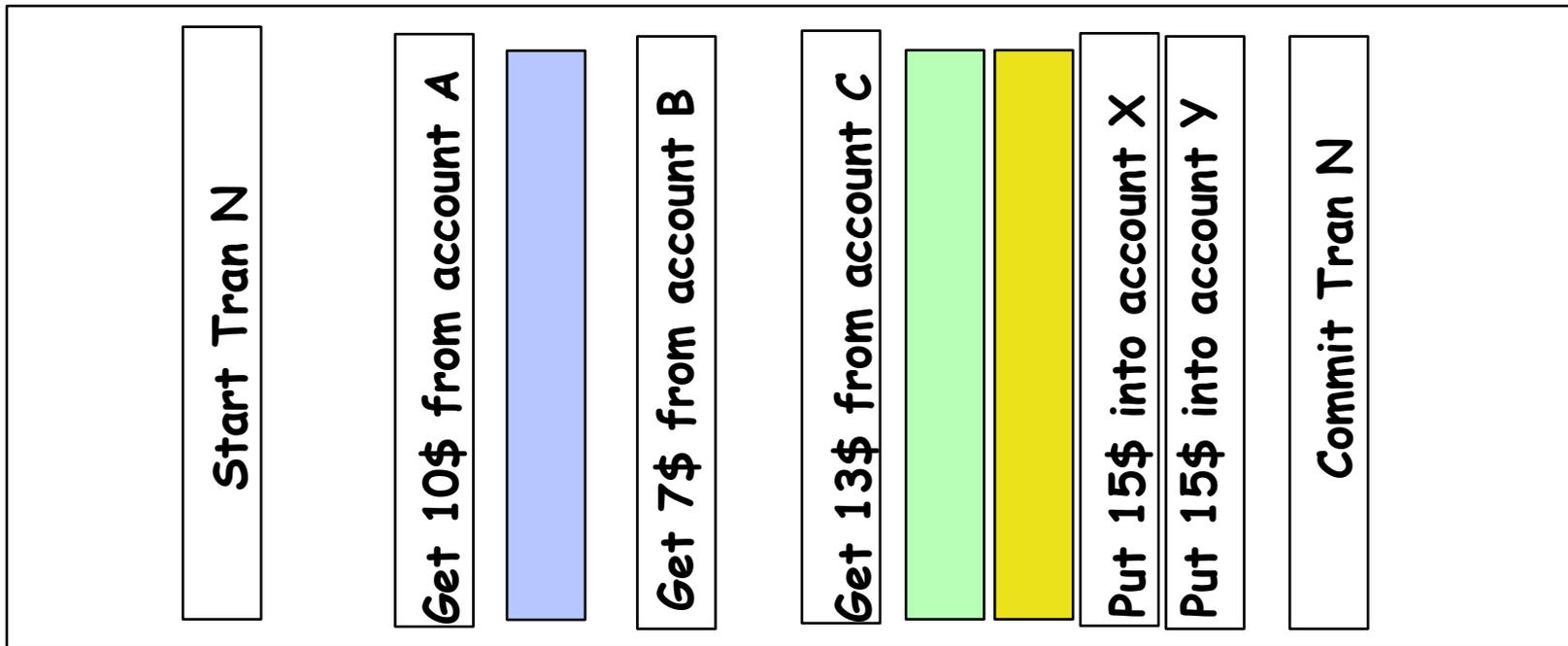
Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiawicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.

Recall: 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

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



Transactional File Systems

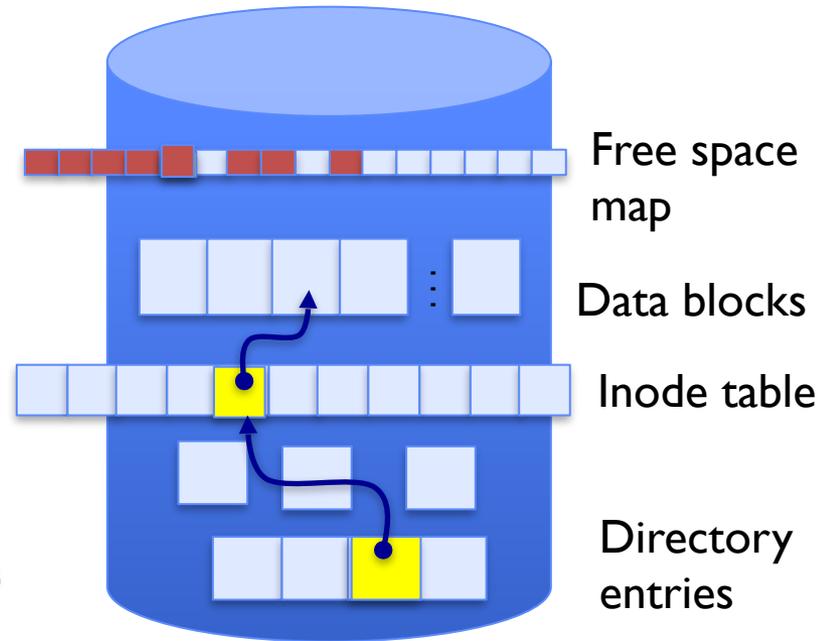
- Better reliability through use of log
 - All changes are treated as *transactions*
 - A transaction is *committed* once it is written to the log
 - » Data forced to disk for reliability
 - » Process can be accelerated with NVRAM
 - Although File system may not be updated immediately, data preserved 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 System
 - Applies updates to system metadata using transactions (using logs, etc.)
 - Updates to non-directory files (i.e., user stuff) can be done in place (without logs), full logging optional
 - Ex: NTFS, Apple HFS+, Linux XFS, JFS, ext3, ext4

Journaling File Systems

- Instead of modifying data structures on disk directly, write changes to a journal/log
 - Intention list: set of changes we intend to make
 - Log/Journal is **append-only**
 - Single commit record commits transaction
- Once changes are in the log, it is safe to apply changes to data structures on disk
 - Recovery can read log to see what changes were intended
 - Can take our time making the changes
 - » As long as new requests consult the log first
- Once changes are copied, safe to remove log
- But, ...
 - If the last atomic action is not done ... poof ... all gone
- Basic assumption:
 - Updates to sectors are atomic and ordered
 - Not necessarily true unless very careful, but key assumption

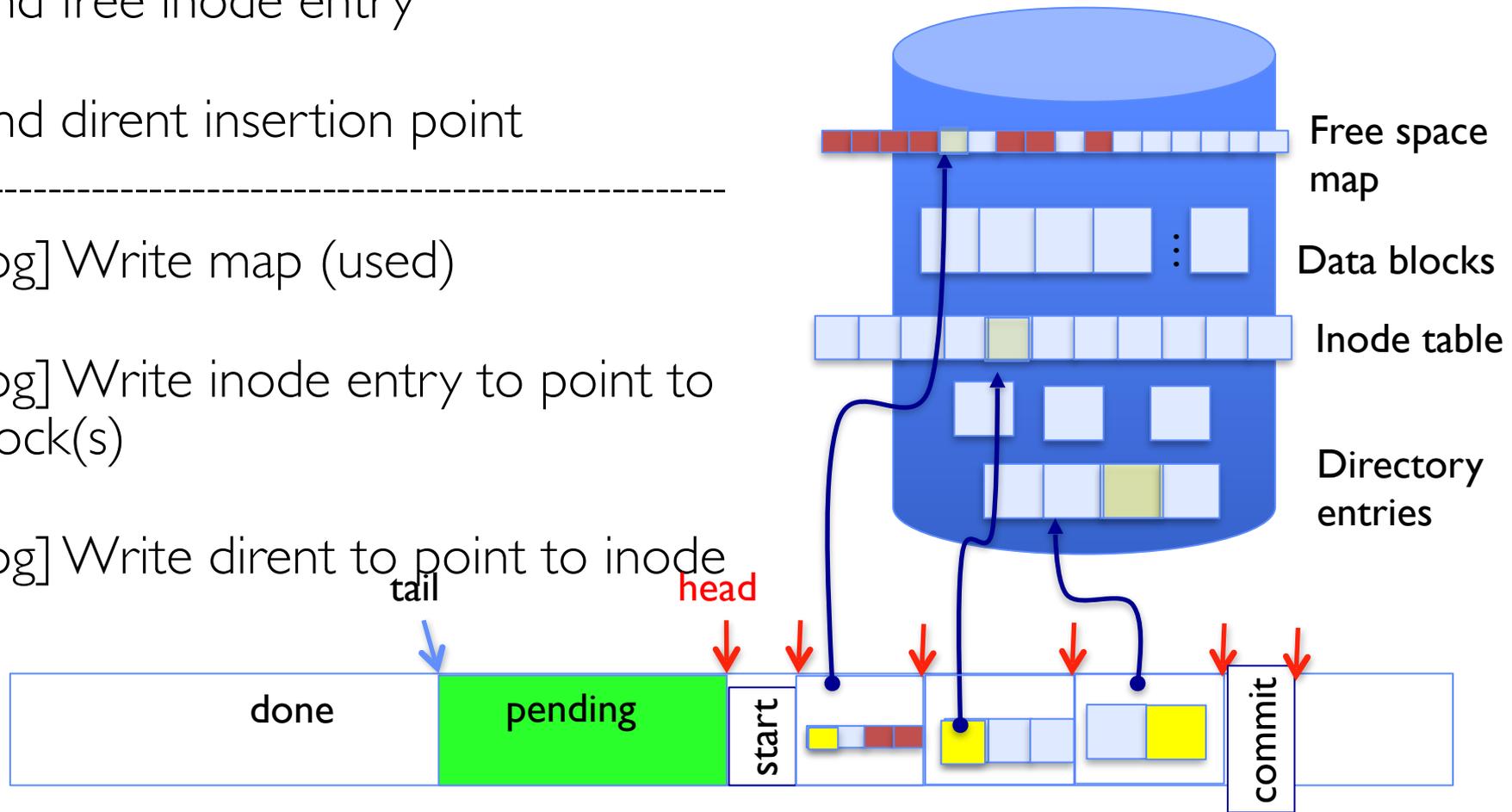
Example: Creating a File

- 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



Ex: Creating a file (as a transaction)

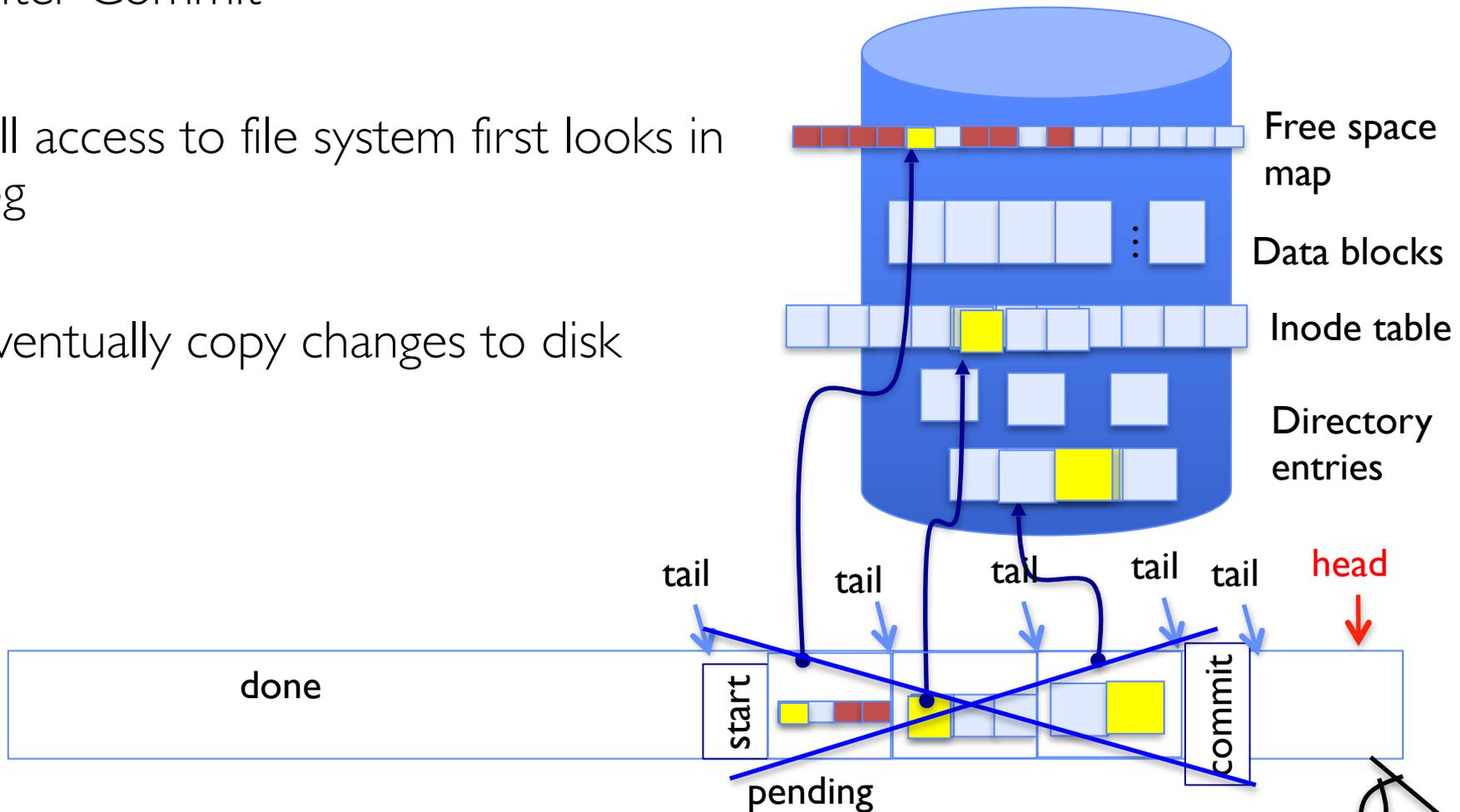
- Find free data block(s)
 - Find free inode entry
 - Find dirent insertion point
-
- [log] Write map (used)
 - [log] Write inode entry to point to block(s)
 - [log] Write dirent to point to inode



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

“Redo Log” – Replay Transactions

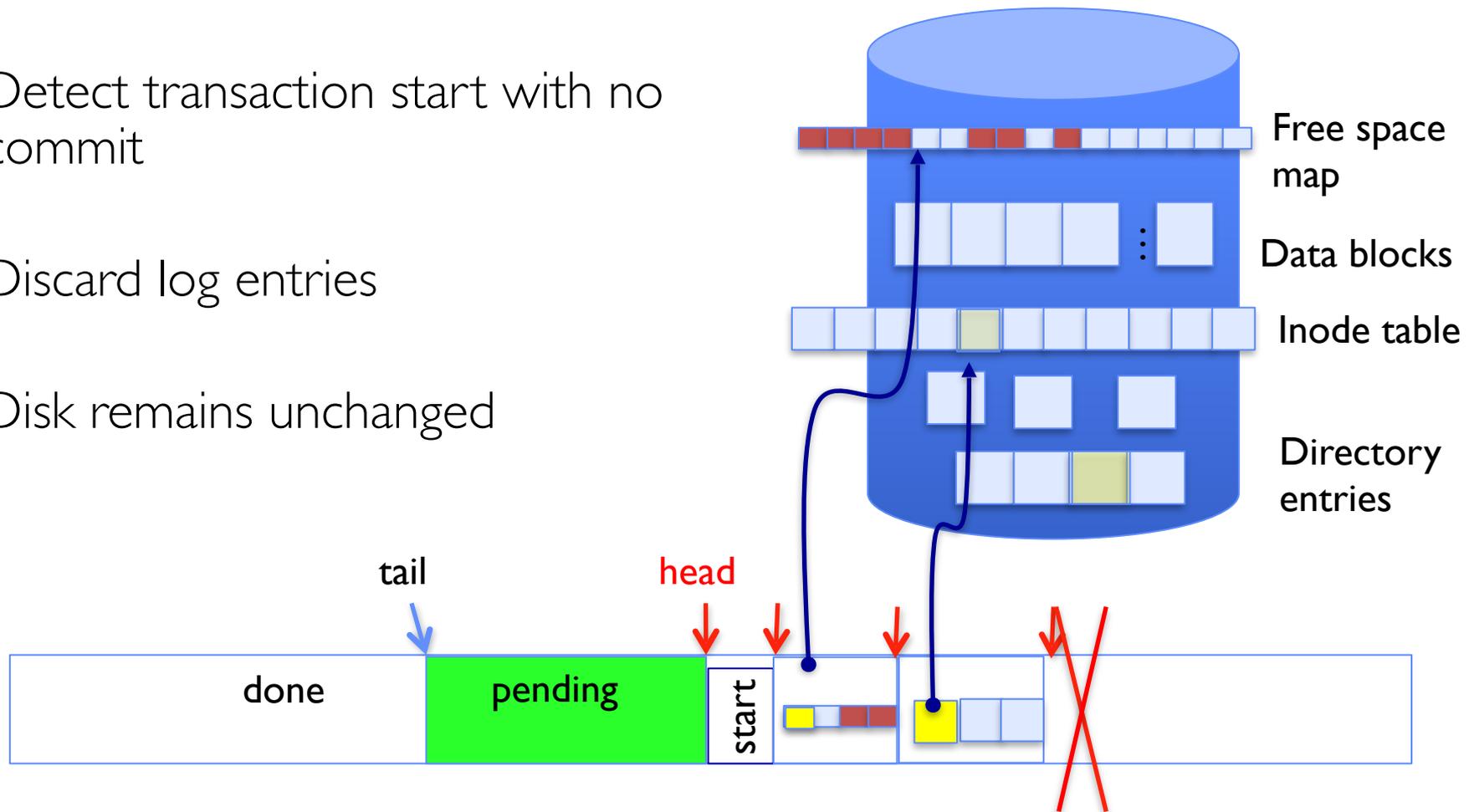
- After Commit
- All access to file system first looks in log
- Eventually copy changes to disk



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

Crash During Logging – Recover

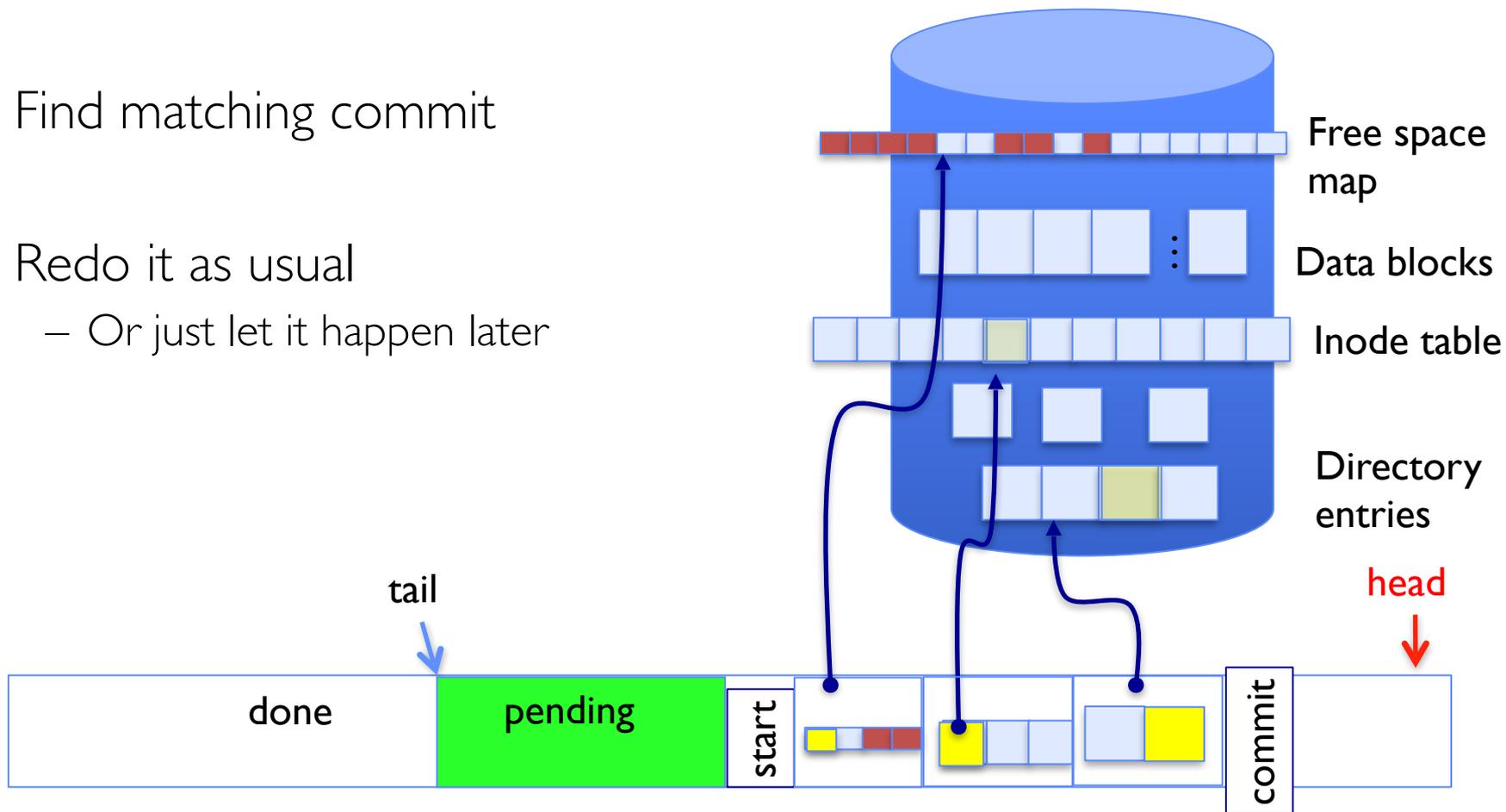
- Upon recovery scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



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

Recovery After Commit

- 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 offer an option to journal metadata updates only
 - Record modifications to file system data structures
 - But apply updates to a file's contents directly

Going Further – Log Structured File Systems

- The log IS what is recorded on disk
 - File system operations *logically* replay log to get result
 - Create data structures to make this fast
 - On recovery, replay the log
- Index (inodes) and directories are written into the log too
- Large, important portion of the log is cached in memory
- Do everything in bulk: log is collection of large segments
- Each segment contains a summary of all the operations within the segment
 - Fast to determine if segment is relevant or not
- Free space is approached as continual cleaning process of segments
 - Detect what is live or not within a segment
 - Copy live portion to new segment being formed (replay)
 - Garbage collection entire segment
 - No bit map

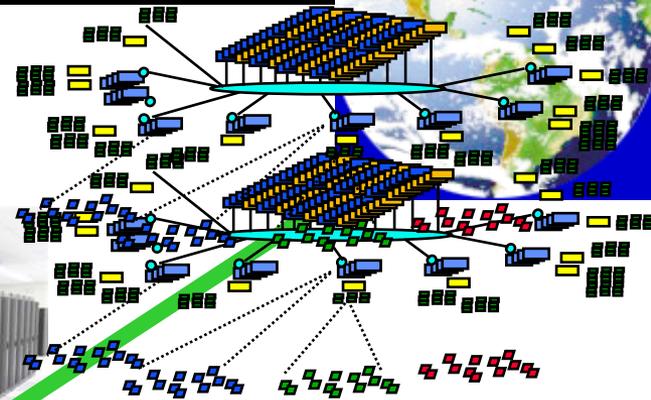
Example Use of LFS: F2FS: A Flash File System

- File system used on many mobile devices
 - Including the Pixel 3 from Google
 - Latest version supports block-encryption for security
 - Has been “mainstream” in linux for several years now
- Assumes standard SSD interface
 - With built-in Flash Translation Layer (FTL)
 - Random reads are as fast as sequential reads
 - Random writes are bad for flash storage
 - » Forces FTL to keep moving/coalescing pages and erasing blocks
 - » Sustained write performance degrades/lifetime reduced
- Minimize Writes/updates and otherwise keep writes “sequential”
 - Start with Log-structured file systems/copy-on-write file systems
 - Keep writes as sequential as possible
 - Node Translation Table (NAT) for “logical” to “physical” translation
 - » Independent of FTL
- For more details, check out paper in *Readings* section of website
 - “F2FS: A New File System for Flash Storage” (from 2015)
 - Design of file system to leverage and optimize NAND flash solutions
 - Comparison with Ext4, Btrfs, Xfs, etc

Societal Scale Information Systems



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

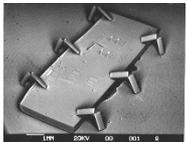


Internet
Connectivity

Scalable, Reliable,
Secure Services

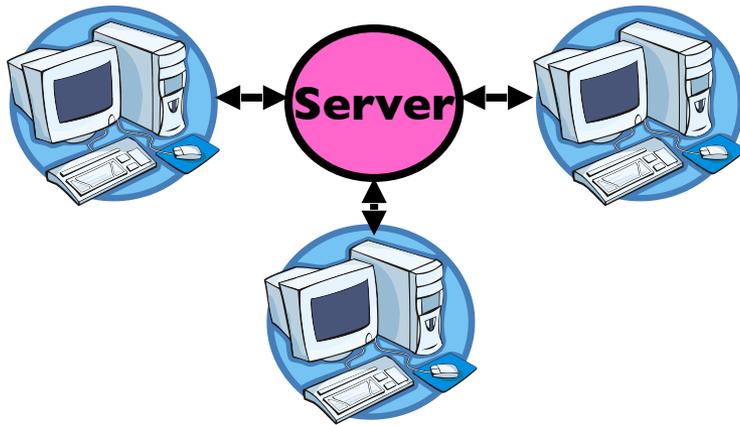


Databases
Information Collection
Remote Storage
Online Games
Commerce
...

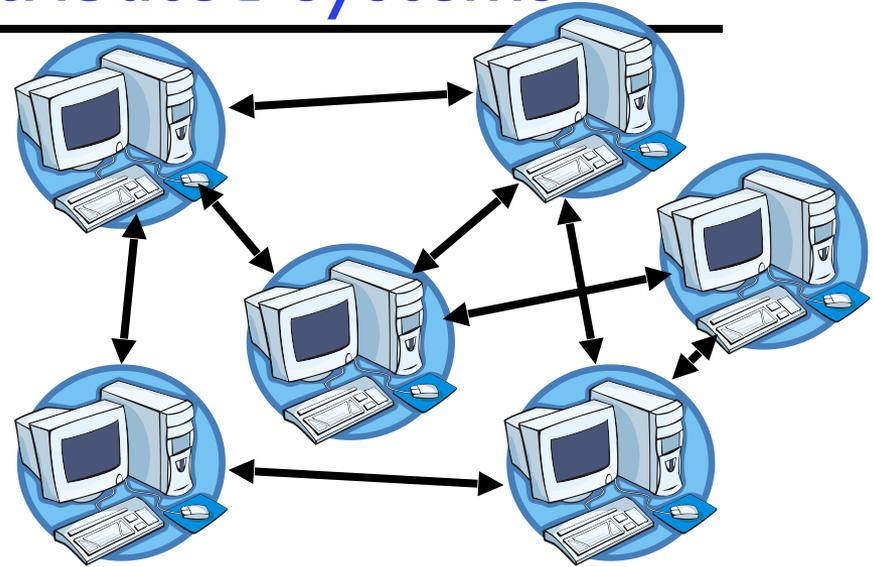


MEMS for
Sensor Nets

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

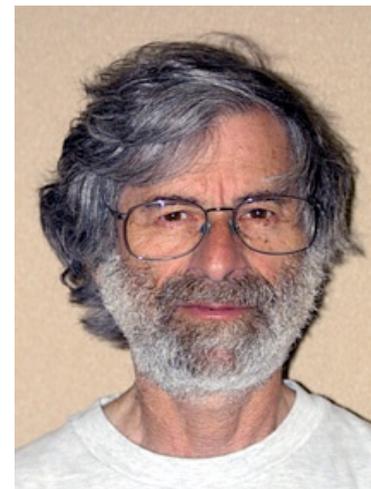
- **Centralized System:** System in which major functions are performed by a single physical computer
 - Originally, everything on single computer
 - Later: client/server model
- **Distributed System:** physically separate computers working together on some task
 - Early model: multiple servers working 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

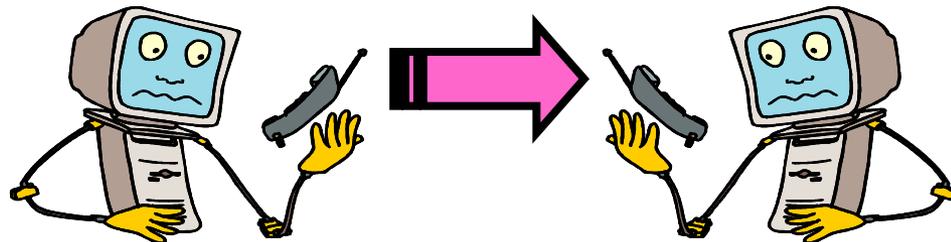
- Reality has been disappointing
 - *Worse availability*: depend on every machine being up
 - » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
 - *Worse reliability*: can lose data if any machine crashes
 - *Worse security*: anyone in world can break into system
- Coordination is more difficult
 - Must coordinate multiple copies of shared state information (using only a network)
 - 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!”



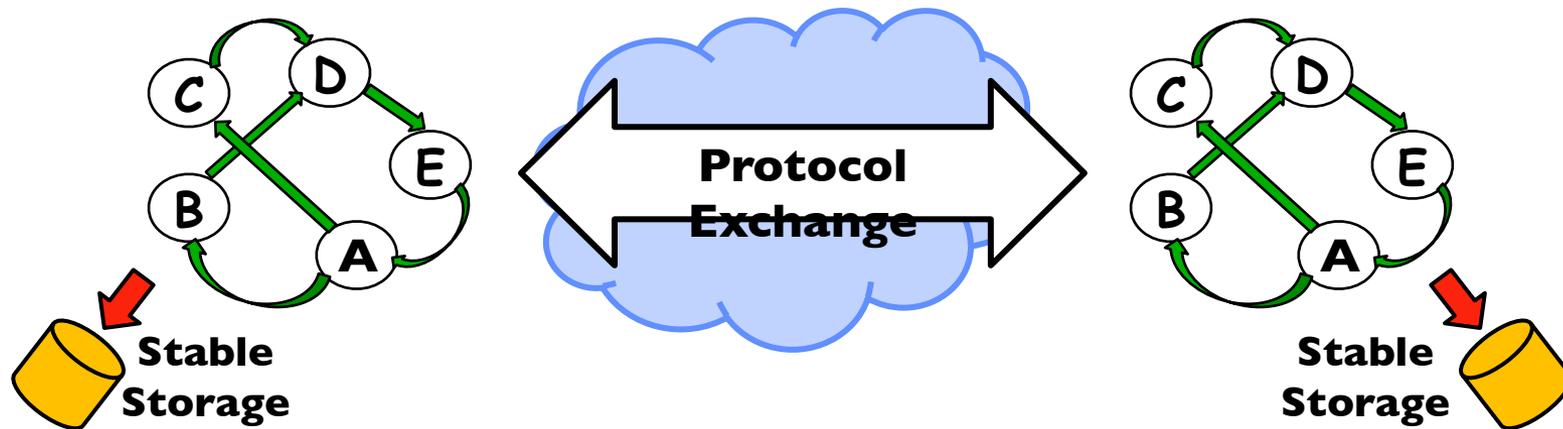
Leslie Lamport

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!



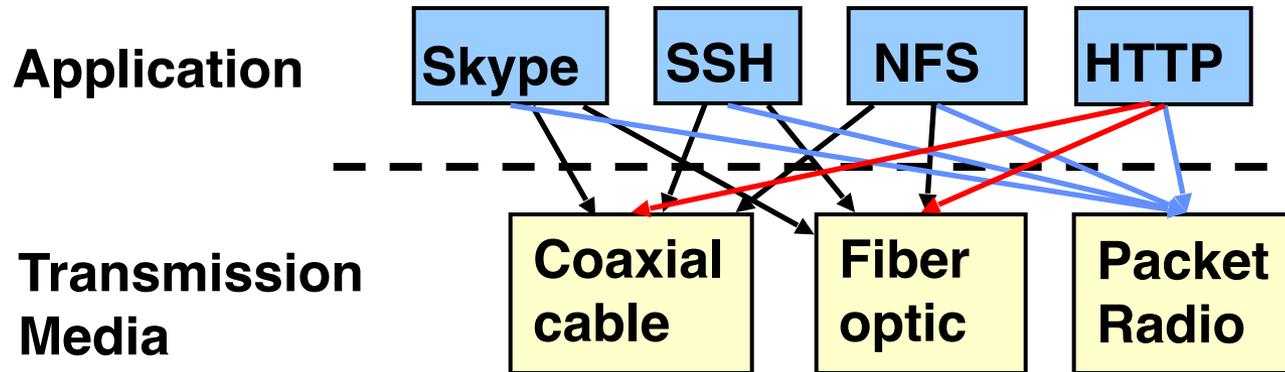
- A protocol is **an agreement on how to communicate**, including:
 - **Syntax**: how a communication is specified & structured
 - » Format, order messages are sent and received
 - **Semantics**: what a communication means
 - » Actions taken when transmitting, receiving, or when a timer expires
- Described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

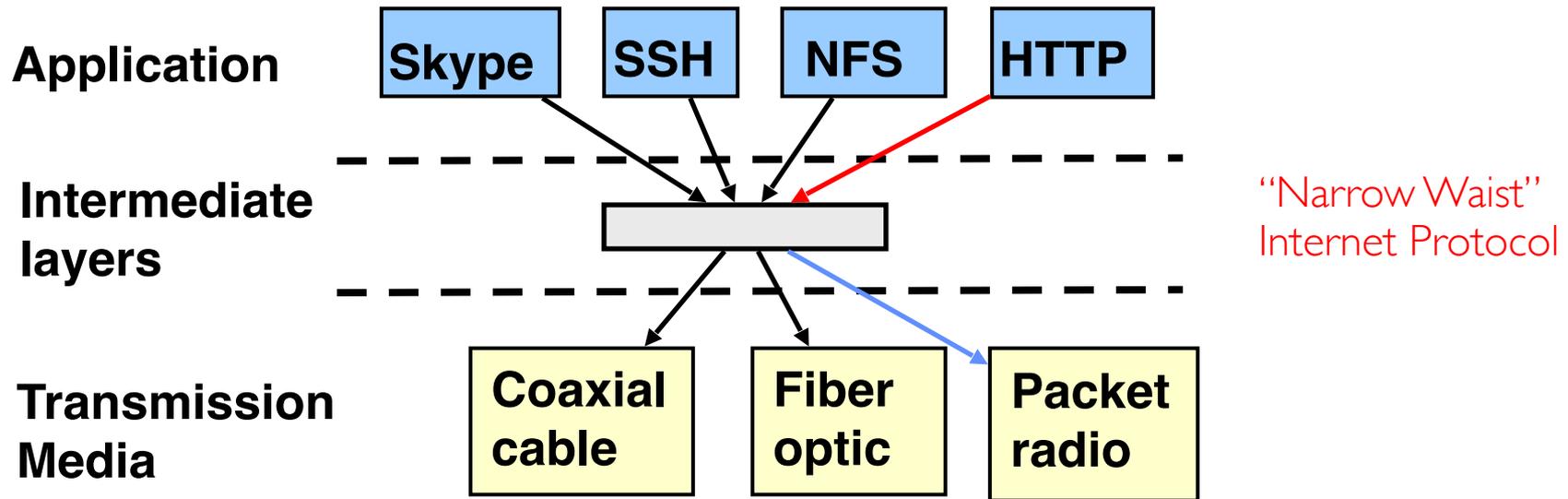
1. (Pick up / open up the phone)
2. Listen for a dial tone / see that you have service
3. Dial
4. Should hear ringing ...
5. Callee: "Hello?"
6. Caller: "Hi, it's John...."
Or: "Hi, it's me" (← what's *that* about?)
7. Caller: "Hey, do you think ... blah blah blah ..." **pause**
1. Callee: "Yeah, blah blah blah ..." **pause**
2. Caller: Bye
3. Callee: Bye
4. Hang up

Global Communication: The Problem



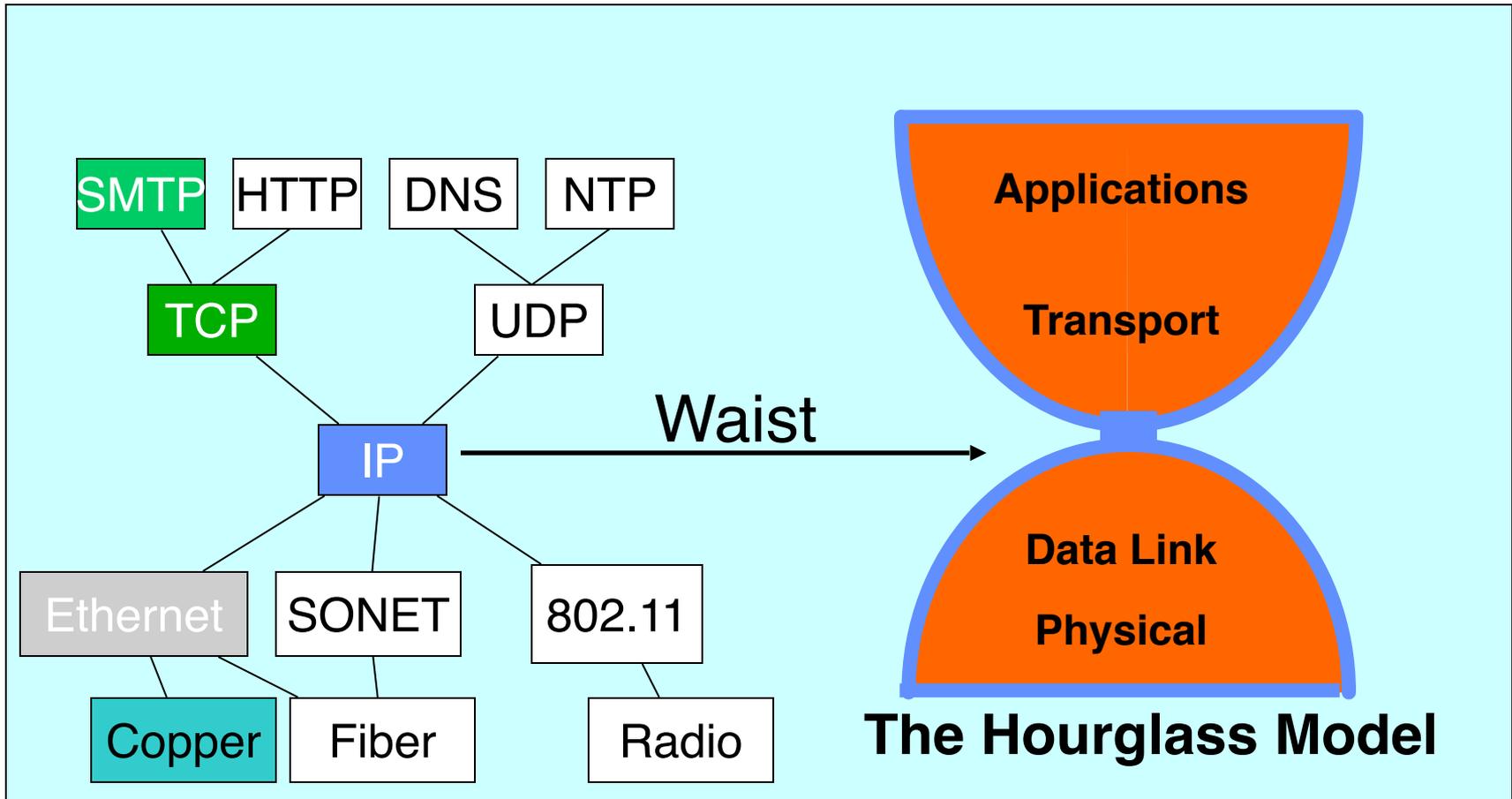
- Many different applications
 - email, web, P2P, etc.
- Many different network styles and technologies
 - Wireless vs. wired vs. optical, etc.
- How do we organize this mess?
 - Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

Solution: Intermediate Layers



- Introduce intermediate layers that provide **set of abstractions** for various network functionality & technologies
 - A new app/media implemented only once
 - Variation on “add another level of indirection”
- **Goal: Reliable communication channels on which to build distributed applications**

The Internet *Hourglass*



There is just **one** network-layer protocol, IP.

The “narrow waist” facilitates **interoperability**.

Implications of Hourglass

Single Internet-layer module (IP):

- Allows arbitrary networks to interoperate
 - Any network technology that supports IP can exchange packets
- Allows applications to function on all networks
 - Applications that can run on IP can **use any network**
- Supports simultaneous innovations above and below IP
 - But changing IP itself, i.e., IPv6, very involved

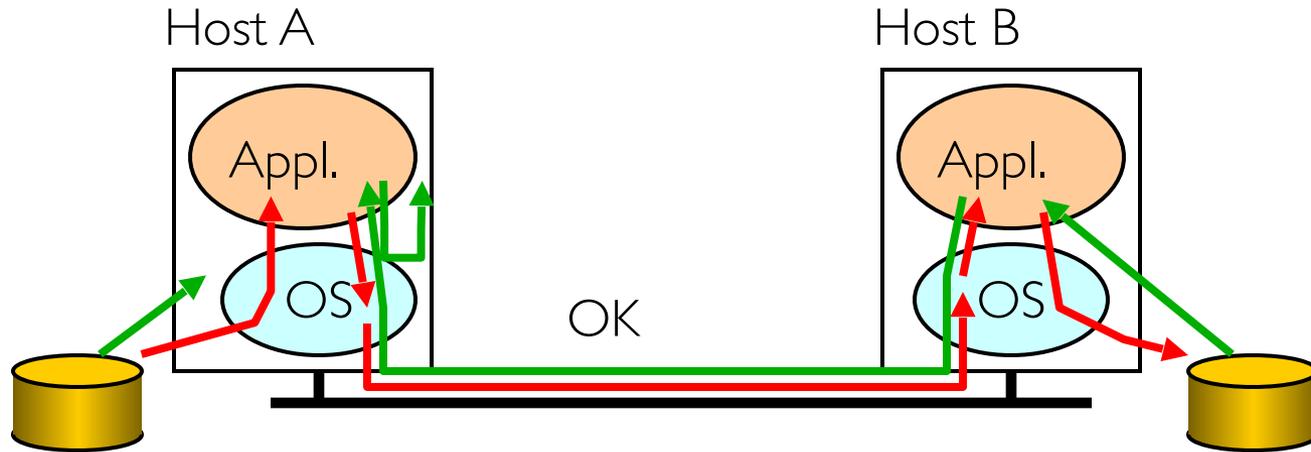
Drawbacks of Layering

- Layer N may duplicate layer N-1 functionality
 - E.g., error recovery to retransmit lost data
- Layers may need same information
 - E.g., timestamps, maximum transmission unit size
- Layering can hurt performance
 - E.g., hiding details about what is really going on
- Some layers are not always cleanly separated
 - Inter-layer dependencies for performance reasons
 - Some dependencies in standards (header checksums)
- Headers start to get really big
 - Sometimes header bytes \gg actual content

End-To-End Argument

- Hugely influential paper: “End-to-End Arguments in System Design” by Saltzer, Reed, and Clark (‘84)
- “Sacred Text” of the Internet
 - Endless disputes about what it means
 - Everyone cites it as supporting their position
- Simple Message: Some types of network functionality can only be correctly implemented **end-to-end**
 - Reliability, security, etc.
- Because of this, end hosts:
 - Can satisfy the requirement without network’s help
 - Will/**must** do so, since can’t **rely** on network’s help
- Therefore **don’t** go out of your way to implement them in the network

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then **concatenate** them
- Solution 2: end-to-end **check** and try again if necessary

Discussion

- Solution 1 is **incomplete**
 - What happens if memory is corrupted?
 - Receiver has to do the check anyway!
- Solution 2 is **complete**
 - Full functionality can be entirely implemented at application layer with **no** need for reliability from lower layers
- *Is there any need to implement reliability at lower layers?*
 - Well, it could be **more efficient**

End-to-End Principle

Implementing complex functionality in the network:

- Doesn't reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, **even if they don't need functionality**
- However, implementing in network **can** enhance performance in some cases
 - e.g., very lossy link

Conservative Interpretation of E2E

- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level
- Or: Unless you can relieve the burden from hosts, don't bother

Moderate Interpretation

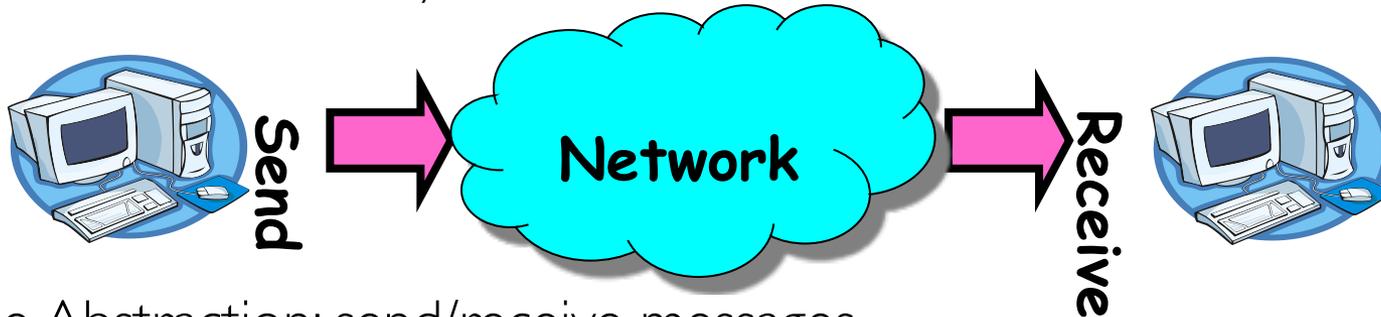
- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it in a lower layer **only** as a performance enhancement
- But do so only if it **does not impose burden** on applications that do not require that functionality
- This is the interpretation we are using

- **Is this still valid?**
 - What about Denial of Service?
 - What about Privacy against Intrusion?

 - Perhaps there are things that must be in the network???

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
 - 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 I-way communication from $T1 \rightarrow T2$
 - $T1 \rightarrow \text{buffer} \rightarrow T2$
 - Very similar to producer/consumer
 - » $\text{Send} = V, \text{Receive} = P$
 - » 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
 - Next time: will discuss fact that this is one of the roles 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 \equiv requester; Server \equiv 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!

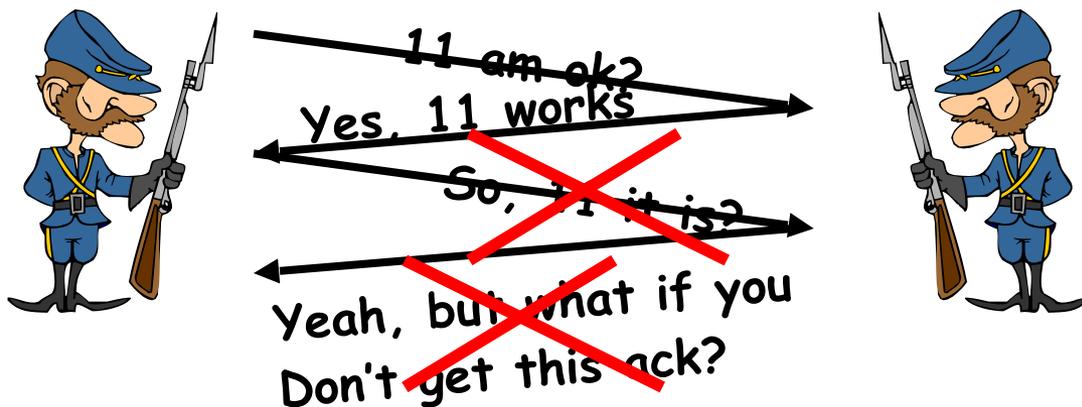
General's Paradox

- 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)

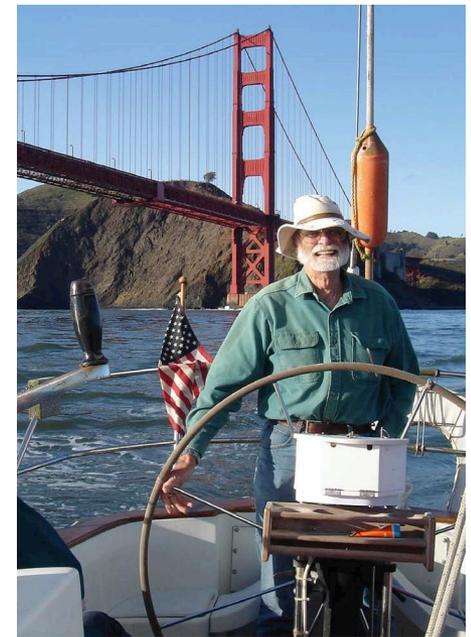
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
 - Remarkably, “no”, even if all messages get through



- 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

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

Agree 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

Agree 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

Detailed Algorithm

Coordinator Algorithm

Worker Algorithm

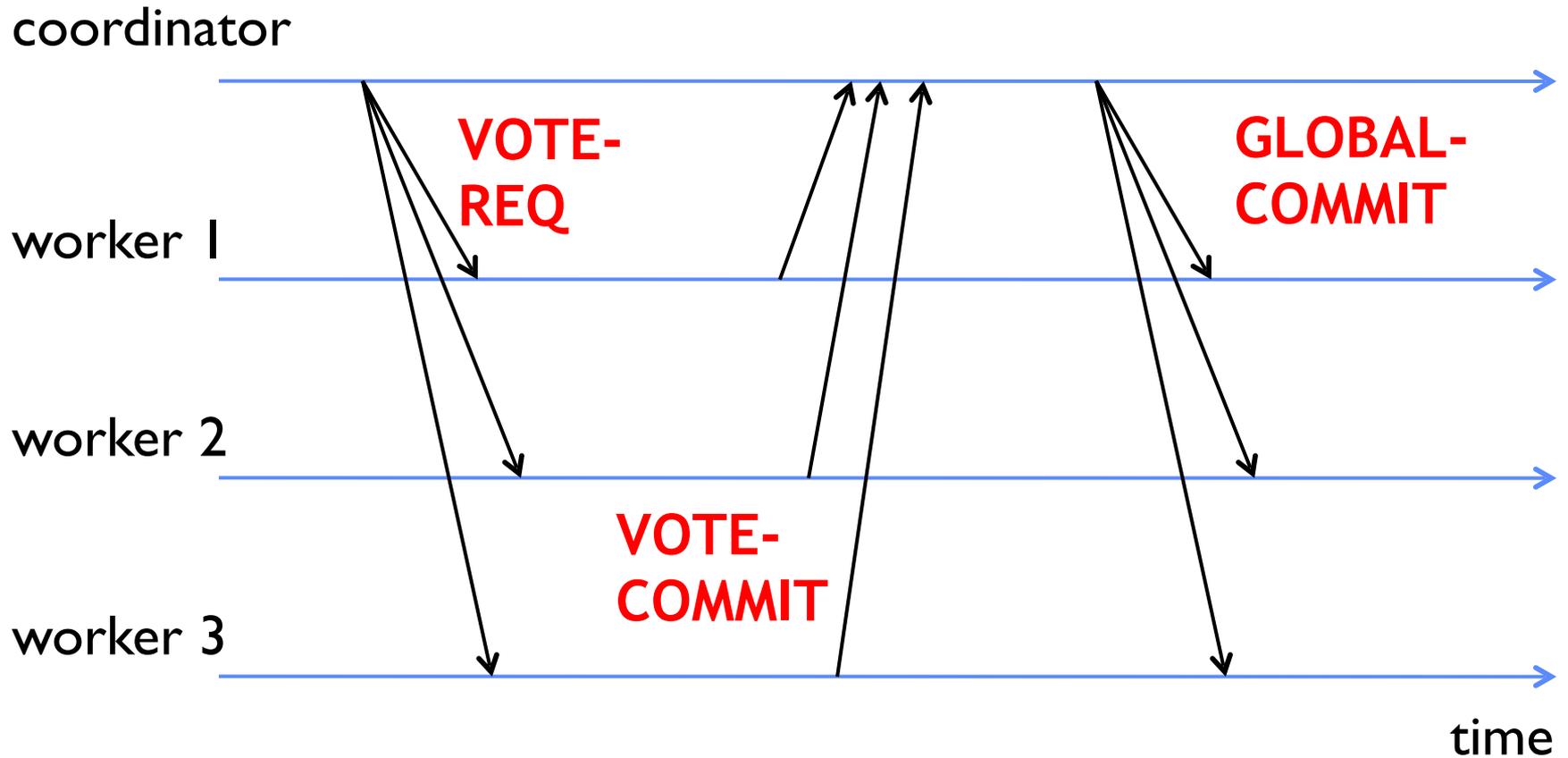
Coordinator sends **VOTE-REQ** to all workers

- 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 **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

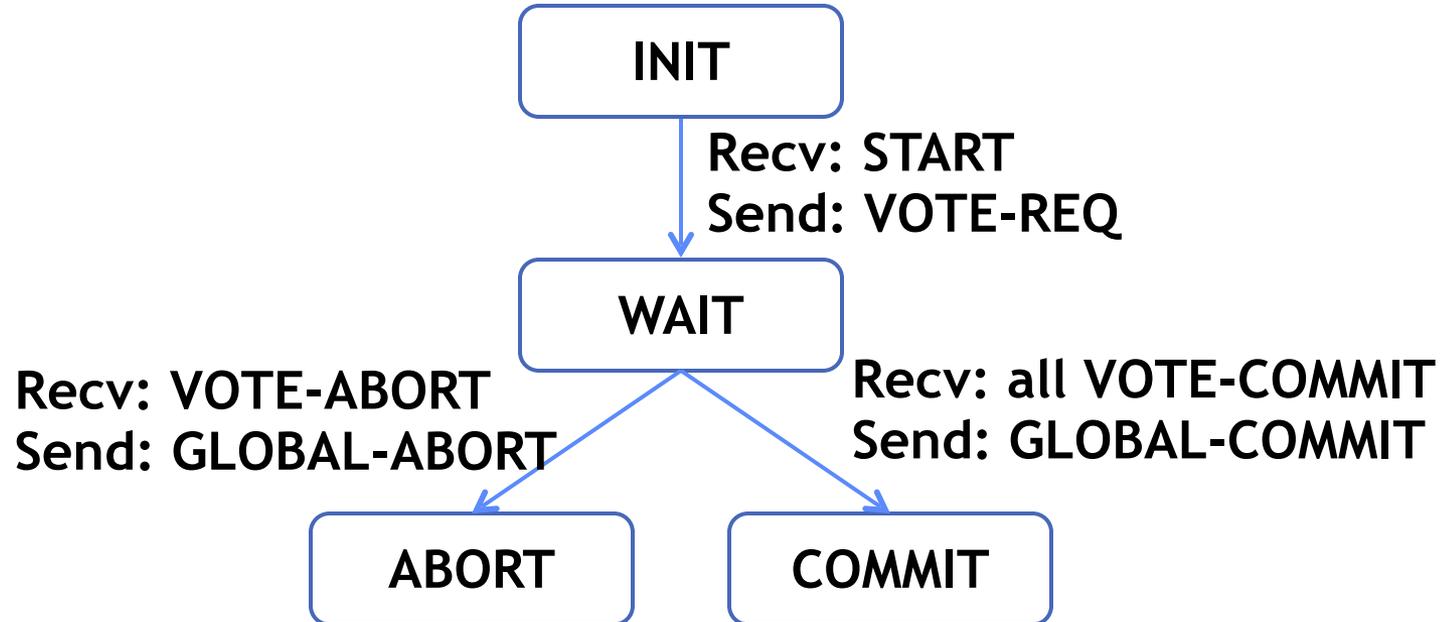
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution

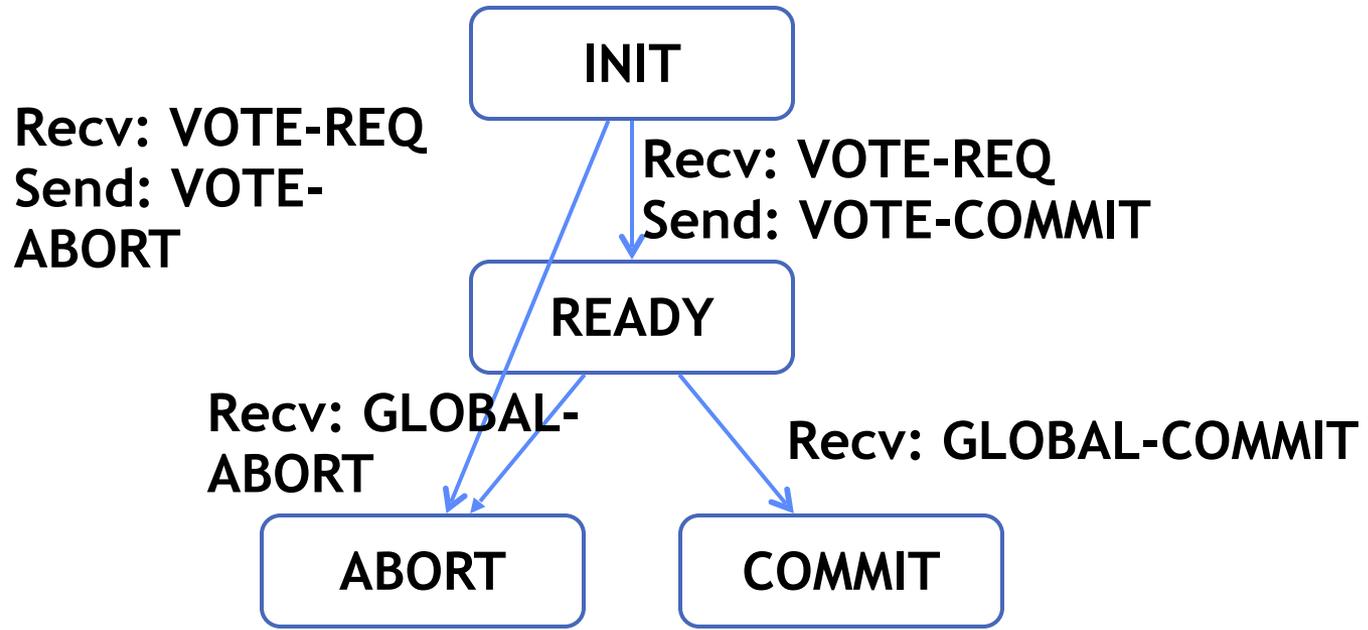


State Machine of Coordinator

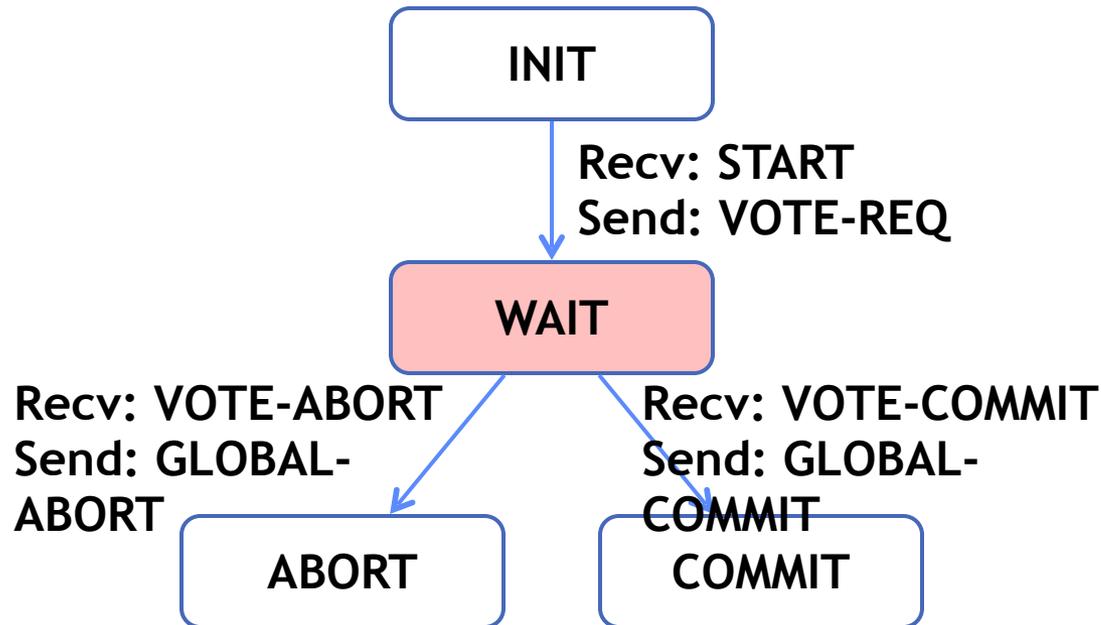
- Coordinator implements simple state machine:



State Machine of Workers

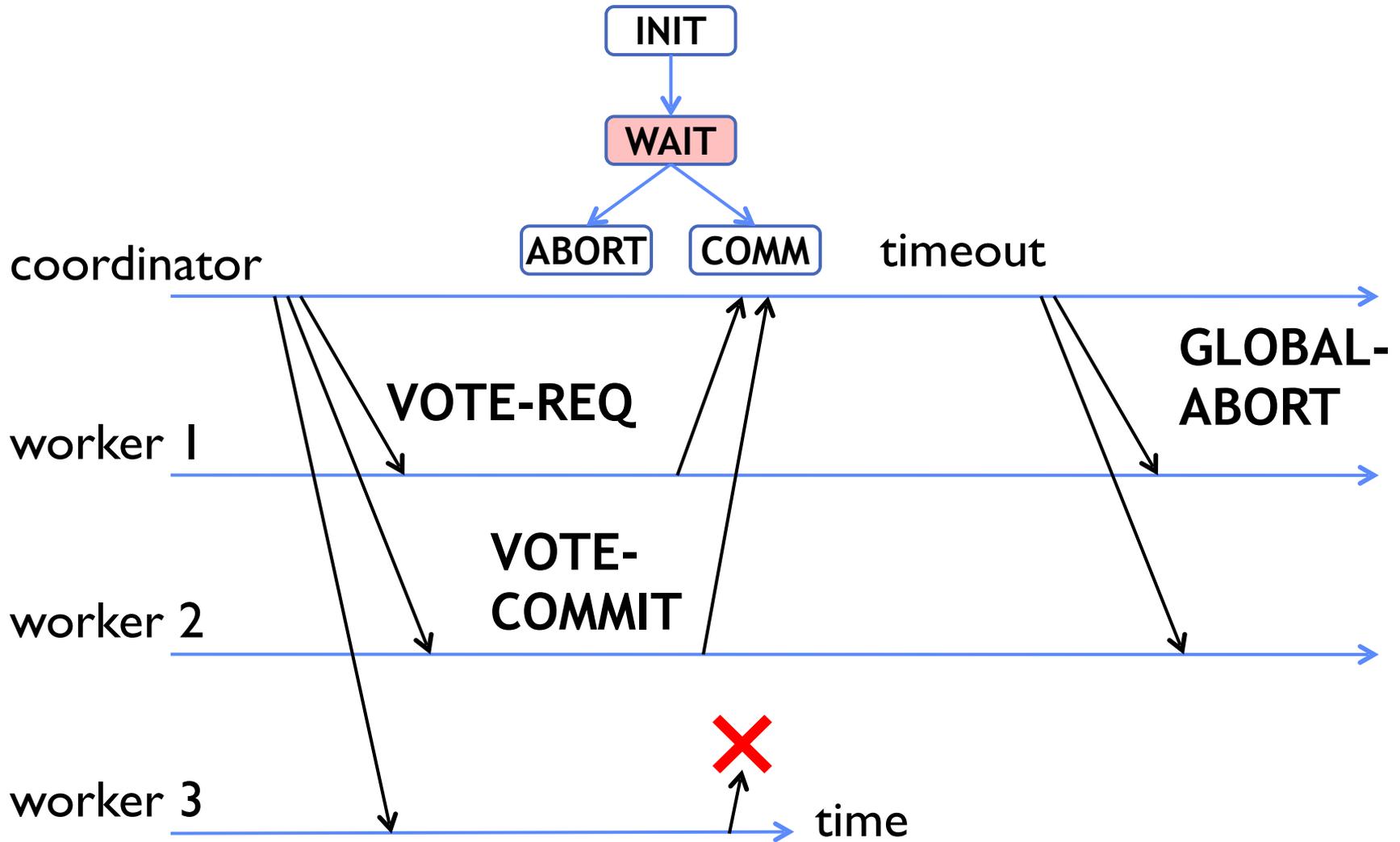


Dealing with Worker Failures

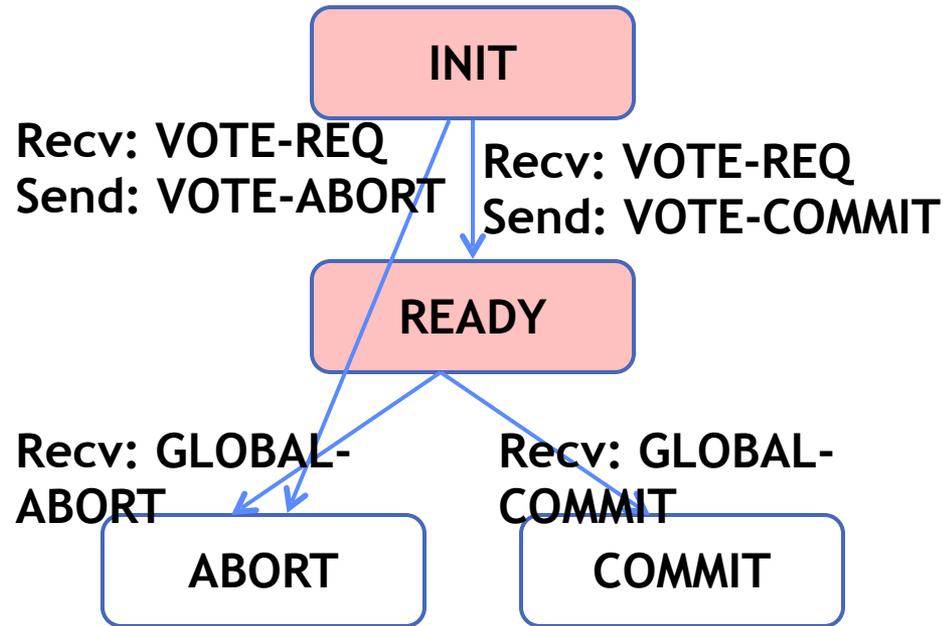


- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in “**WAIT**” state
- In **WAIT**, if doesn't receive N votes, it times out and sends **GLOBAL-ABORT**

Example of Worker Failure

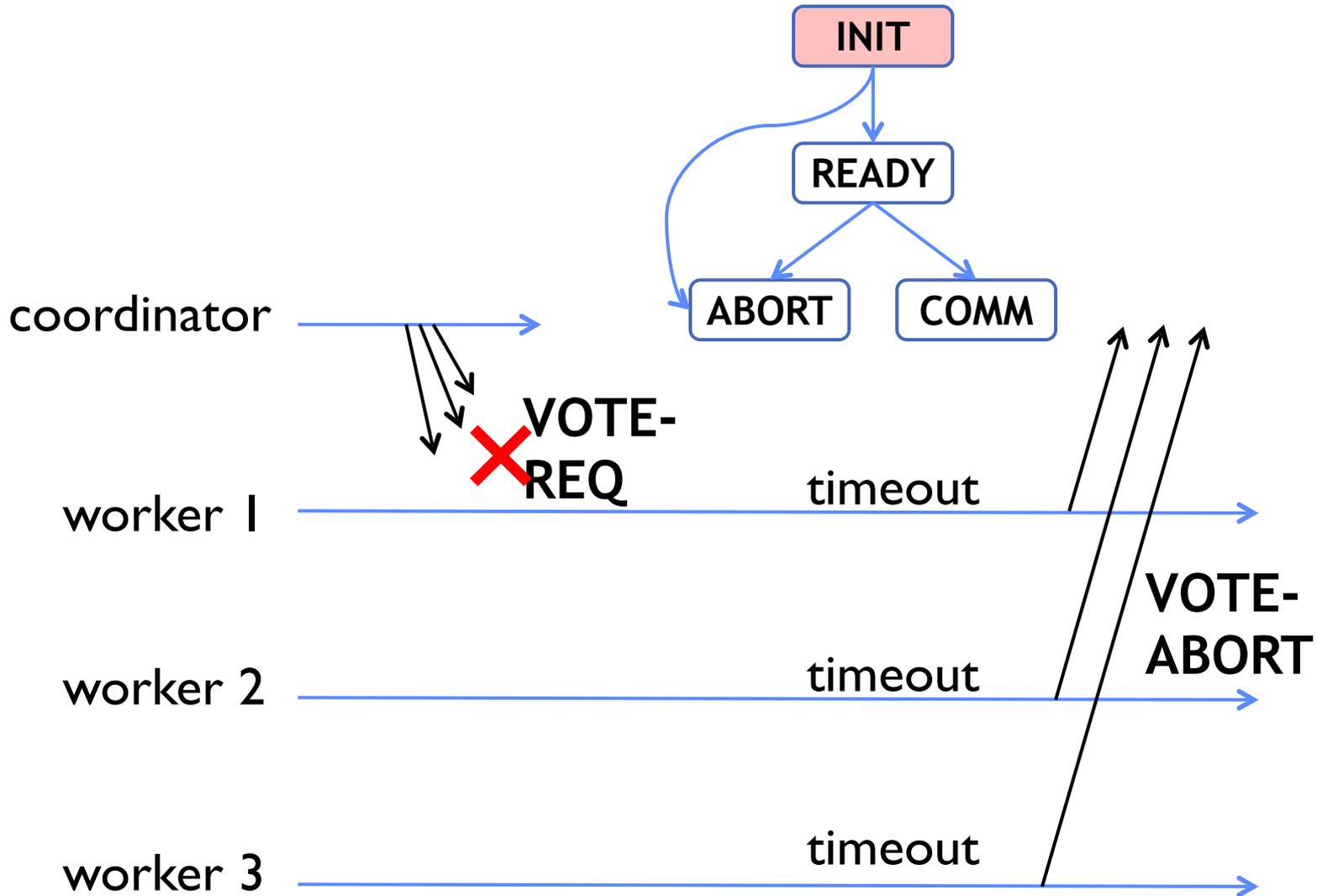


Dealing with Coordinator Failure

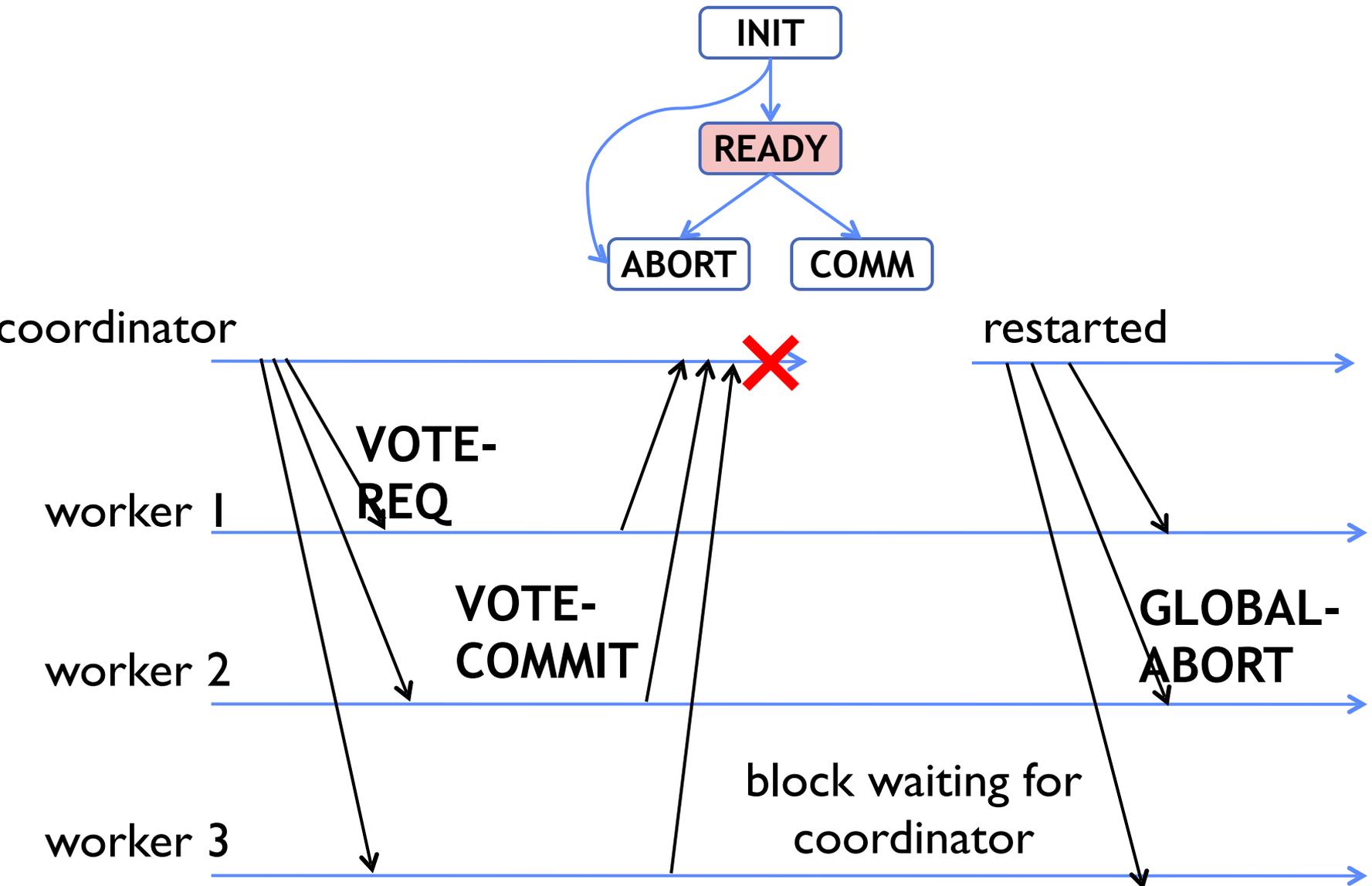


- Worker waits for **VOTE-REQ** in **INIT**
 - Worker can time out and abort (coordinator handles it)
- Worker waits for **GLOBAL-*** message in **READY**
 - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send **GLOBAL_*** message

Example of Coordinator Failure #1



Example of Coordinator Failure #2

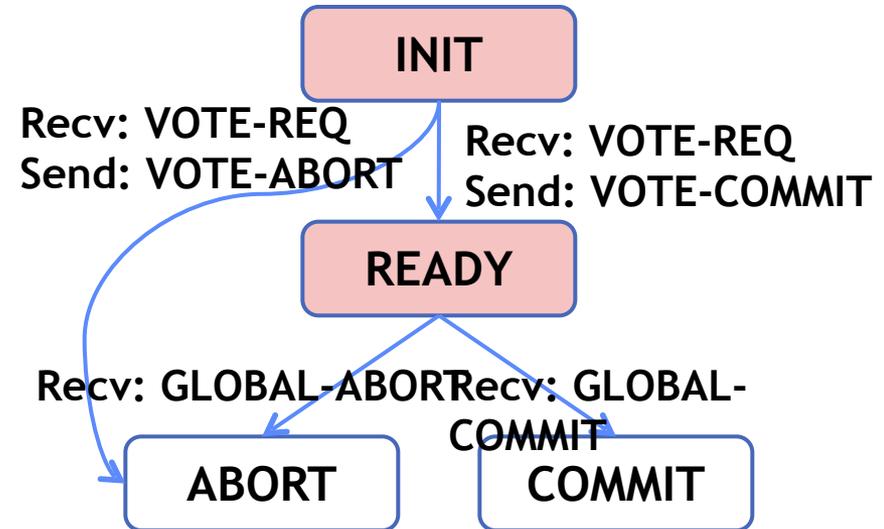


Durability

- 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, 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**

Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state
 - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
 - » Thus, worker can safely abort or commit, respectively
 - If another worker is still in INIT state then both workers can decide to abort
 - If all workers are in ready, need to **BLOCK** (don't know if coordinator wanted to abort or commit)



Distributed Decision Making Discussion (1/2)

- Why is distributed decision making desirable?
 - Fault Tolerance!
 - A group of machines can come to a decision even if one or more of them fail during the process
 - » Simple failure mode called “failstop” (different modes later)
 - After decision made, result recorded in multiple places

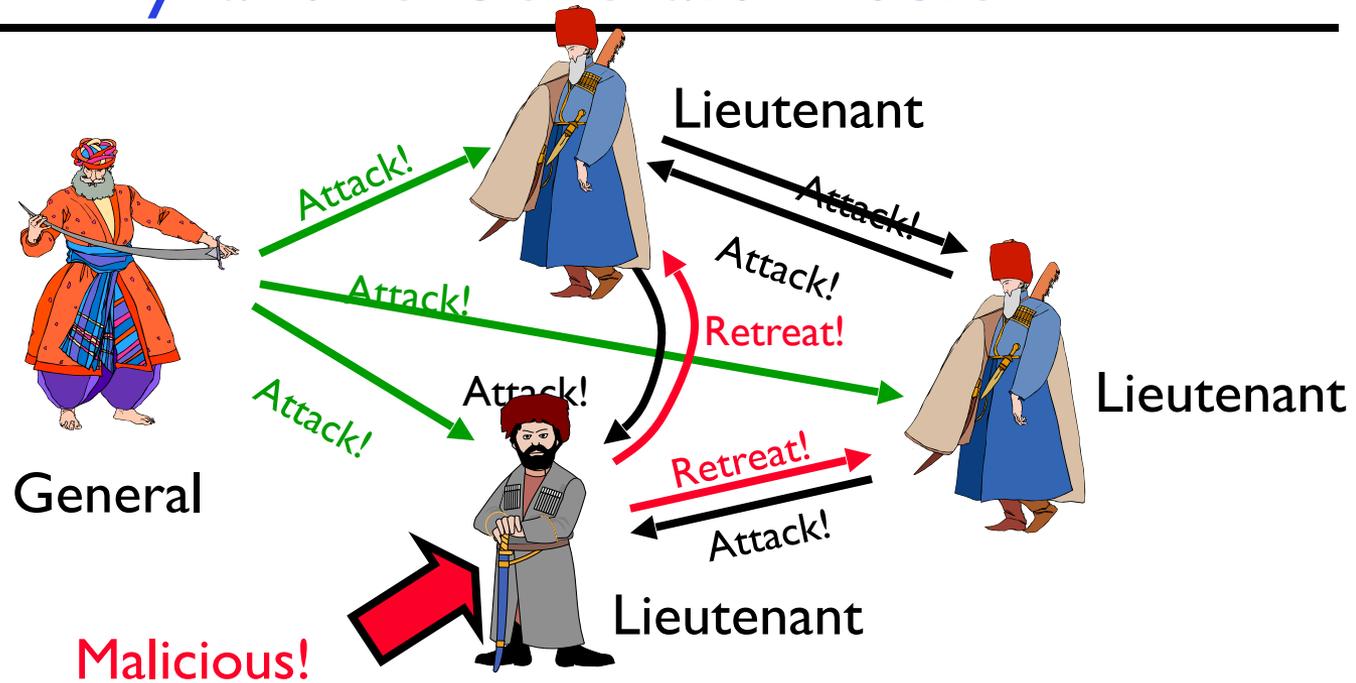
Distributed Decision Making Discussion (2/2)

- Undesirable feature of Two-Phase Commit: Blocking
 - One machine can be stalled until another site recovers:
 - » Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
 - » Site A crashes
 - » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
 - » B is blocked until A comes back
 - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update

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 Ousterhout (Stanford)
 - Simpler to describe complete protocol
- What happens if one or more of the nodes is malicious?
 - **Malicious:** attempting to compromise the decision making

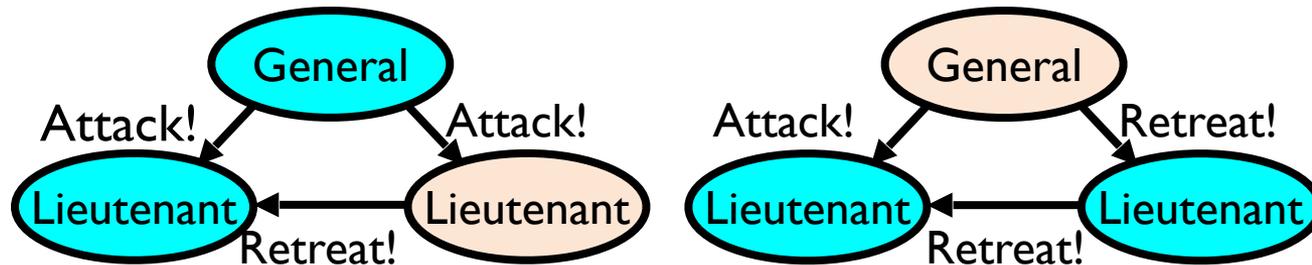
Byzantine General's Problem



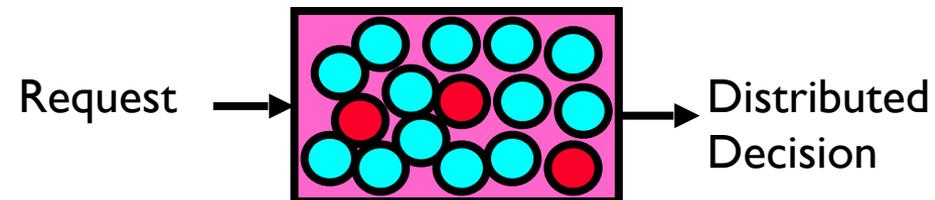
- Byzantine General's Problem (n players):
 - One General and $n-1$ Lieutenants
 - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his $n-1$ lieutenants such that the following Integrity Constraints apply:
 - IC1: All loyal lieutenants obey the same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

Byzantine General's Problem (con't)

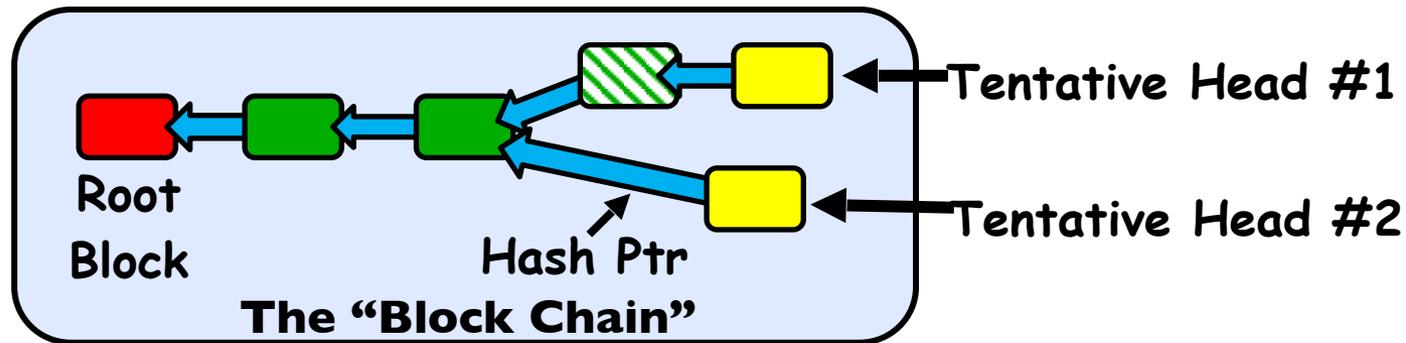
- Impossibility Results:
 - Cannot solve Byzantine General's Problem with $n=3$ because one malicious player can mess up things



- 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

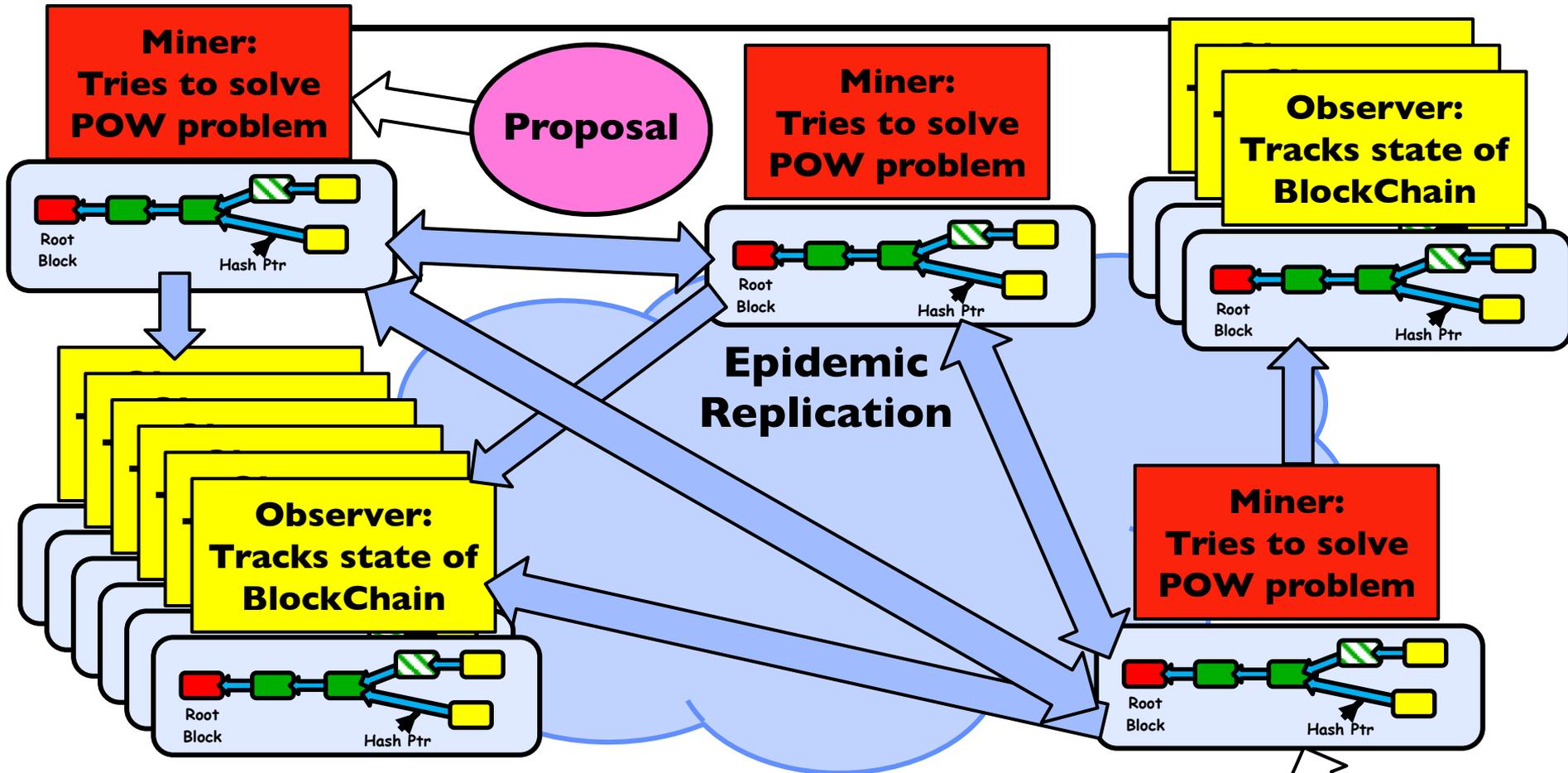


Is a Blockchain a Distributed Decision Making Algorithm?



- 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 (1/2)

- Protocol: Agreement between two parties as to how information is to be transmitted
- E2E argument encourages us to keep Internet communication simple
 - If higher layer can implement functionality correctly, implement it in a lower layer **only** if:
 - » it improves the performance significantly for application that need that functionality, and
 - » it **does not impose burden** on applications that do not require that functionality
- Two-phase commit: distributed decision making
 - First, make sure everyone guarantees that they will commit if asked (prepare)
 - Next, ask everyone to commit

Summary (2/2)

- 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
 - Could be used for distributed decision making