

Operating Systems (Honor Track)

File System 4: Transactions & Distributed Decision Making

Xin Jin

Spring 2022

Recap: Two Reliability Approaches

Careful Ordering and Recovery

- FAT & FFS + fsck
- Each step builds structure
- Data block \leftarrow inode \leftarrow free \leftarrow directory
- Last step links it in to rest of FS
- Recover scans structure looking for incomplete actions

Versioning and Copy-on-Write

- ZFS, ...
- Version files at some granularity
- Create new structure linking back to unchanged parts of old
- Last step is to declare that the new version is ready

Recap: 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)

Recap: Key Concept: Transaction

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



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

Recap: 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

Recap: Transactional File Systems

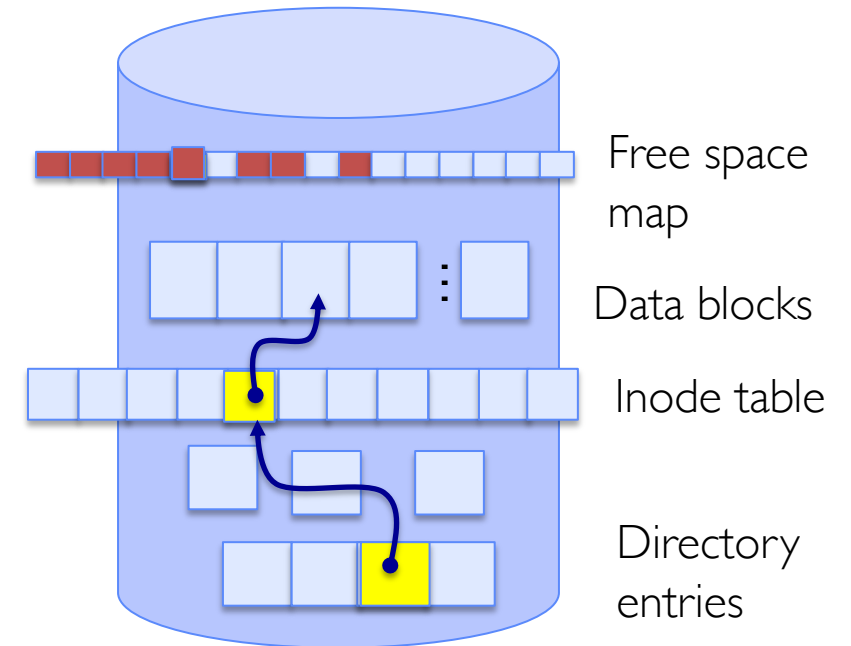
- Better reliability through use of log
 - Changes are treated as transactions
 - A transaction is committed once it is written to the log
 - » Data forced to disk for reliability
 - » Process can be accelerated with NVRAM
 - Although File system may not be updated immediately, data preserved in the 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+, Linux XFS, JFS, ext4

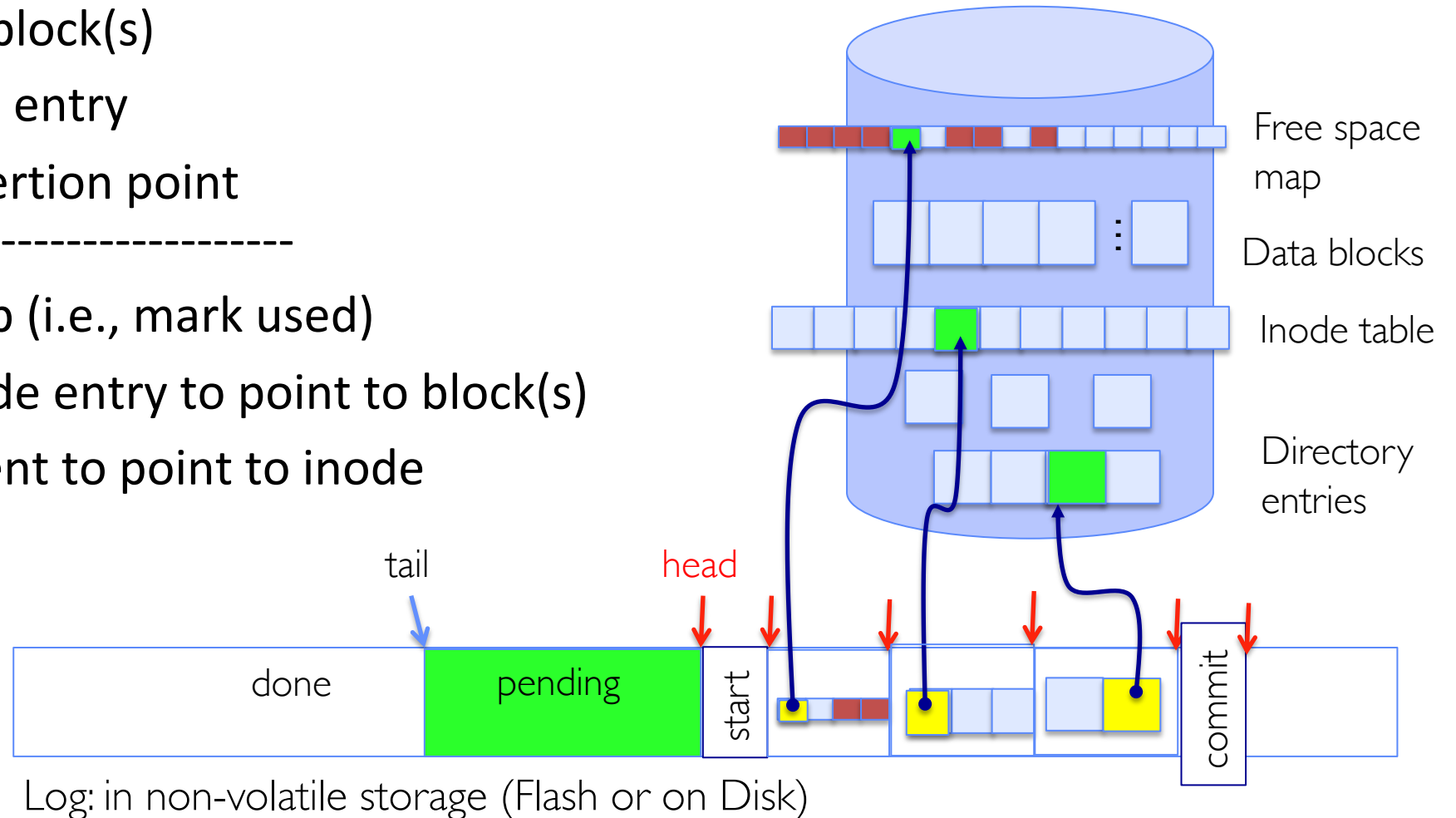
Creating a File (No Journaling Yet)

- 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



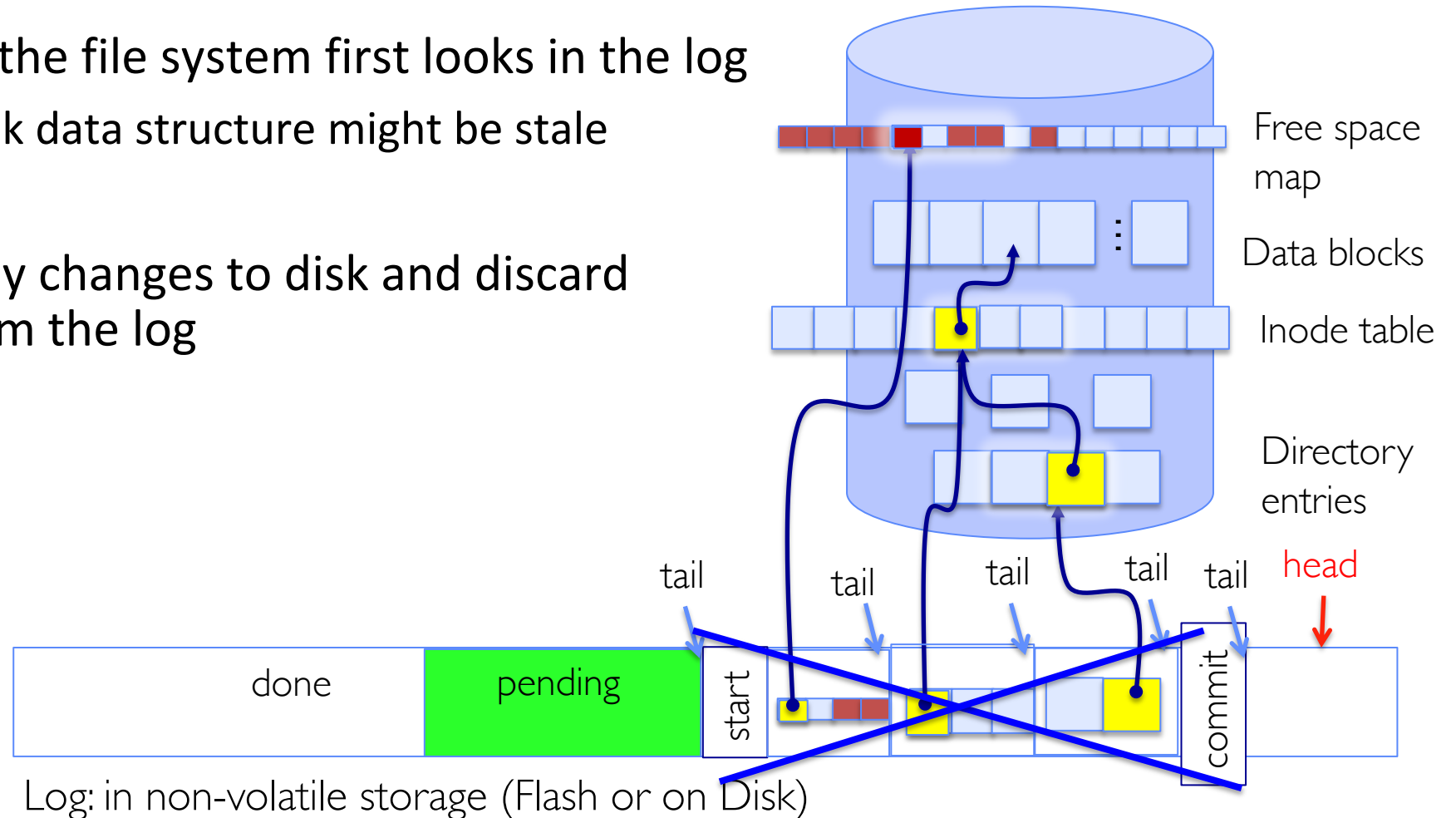
Creating a File (With Journaling)

- 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



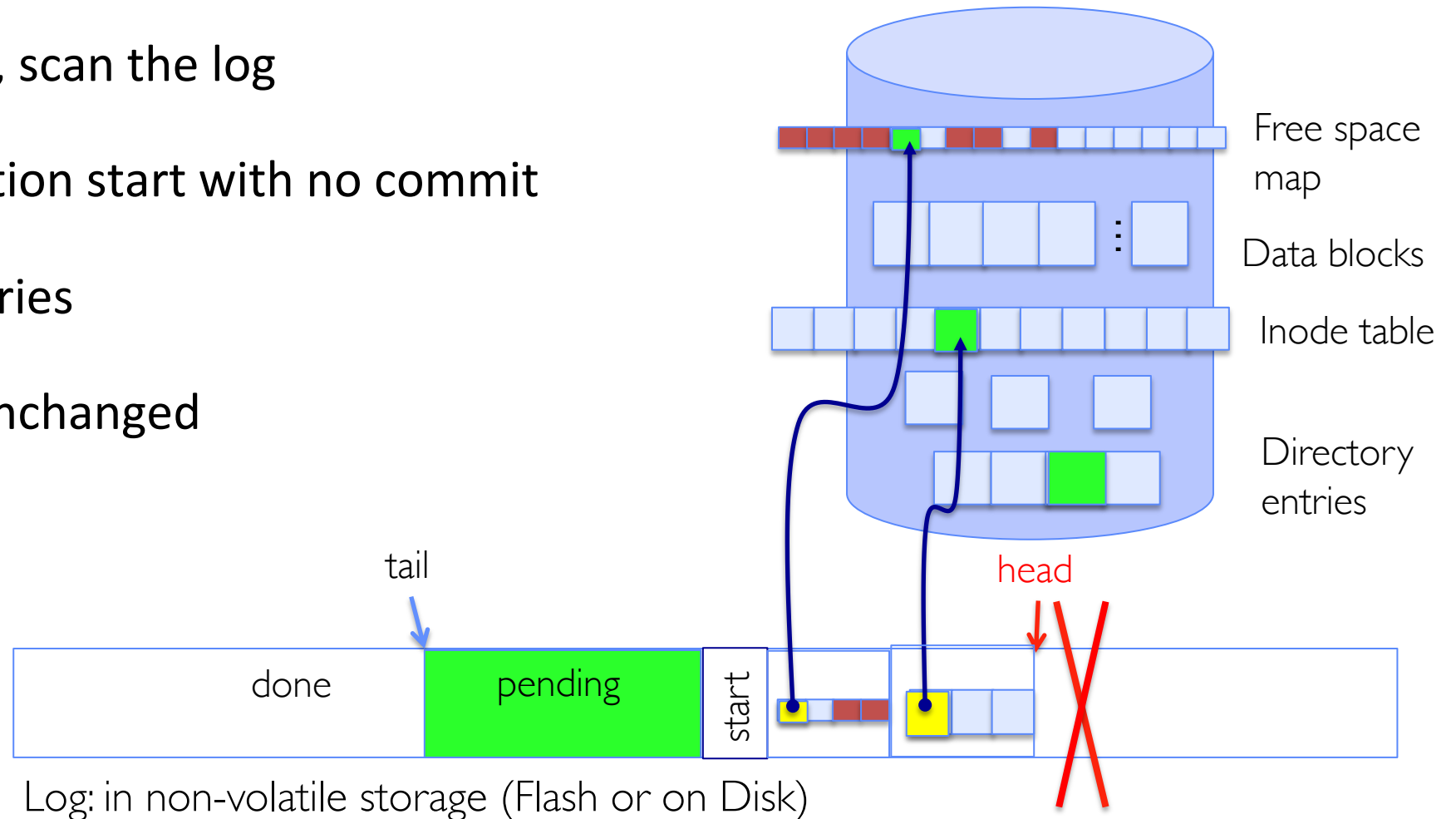
After Commit, Eventually Replay Transaction

- 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



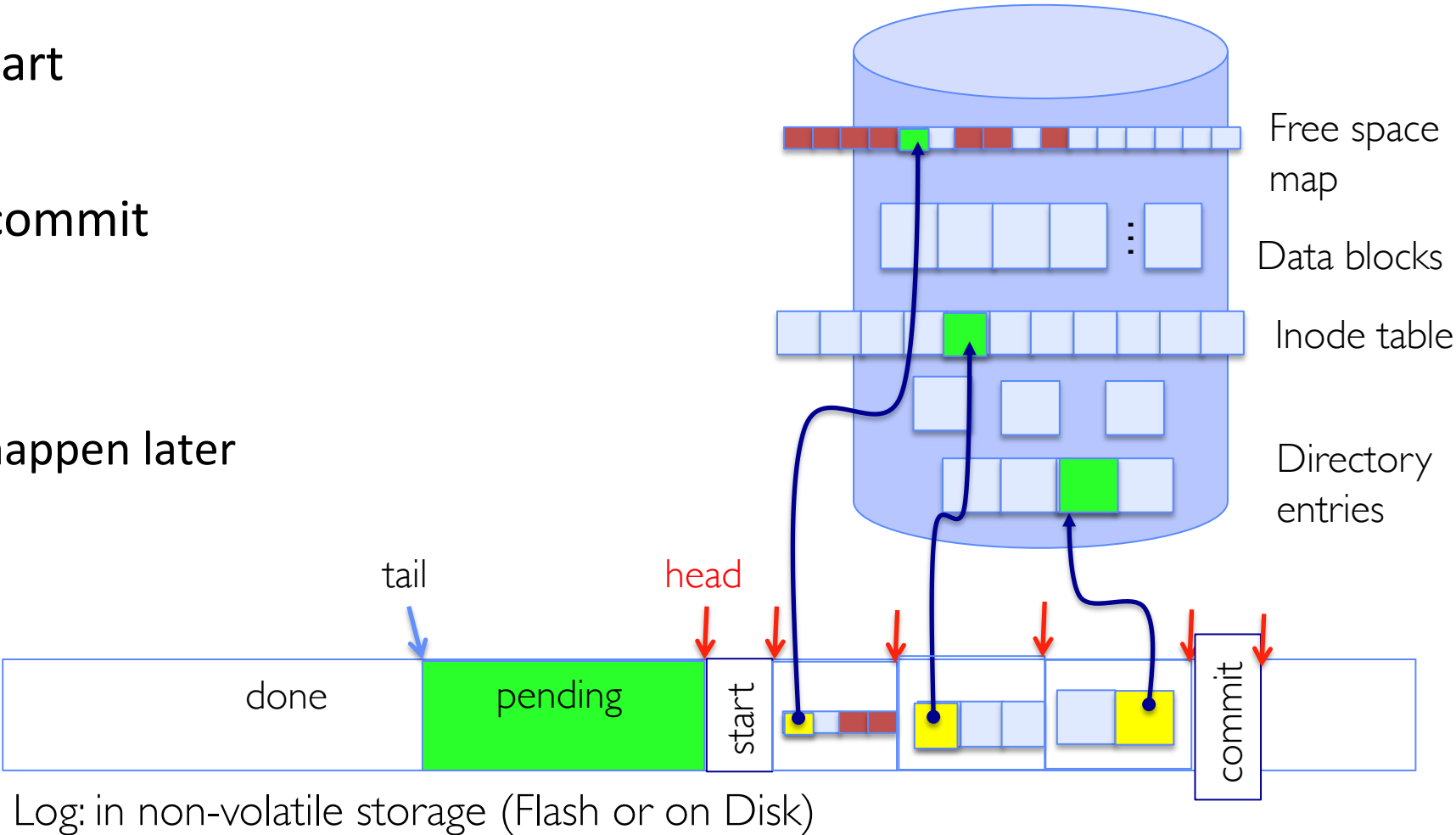
Crash Recovery: Discard Partial Transactions

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



Crash Recovery: Keep Complete Transactions

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



Journaling Summary

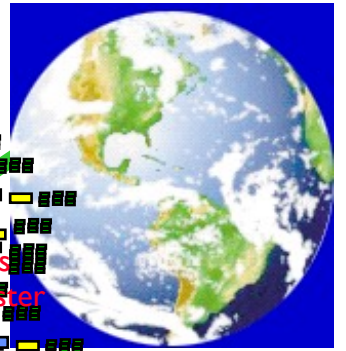
Why go through all this trouble?

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

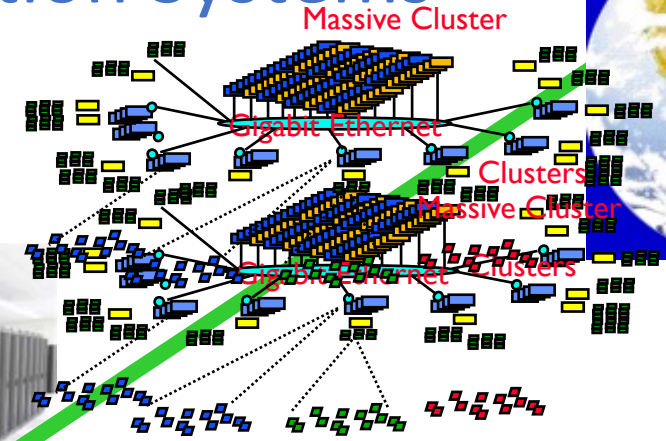
Isn't this expensive?

- 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

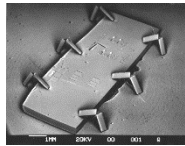


- The world is a large distributed system
 - Microprocessors in everything
 - Vast infrastructure behind them



Internet
Connectivity

Scalable, Reliable,
Secure Services

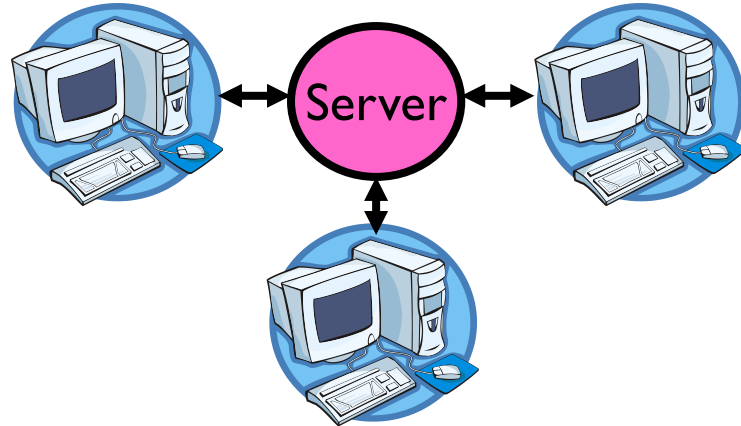


MEMS for
Sensor Nets

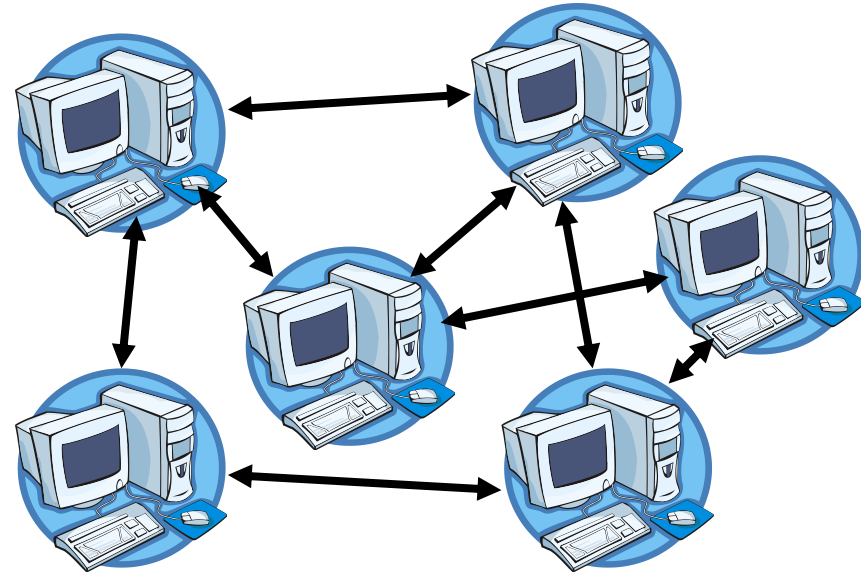


Databases
Information Collection
Remote Storage
Online Games
Commerce
...

Centralized vs Distributed Systems



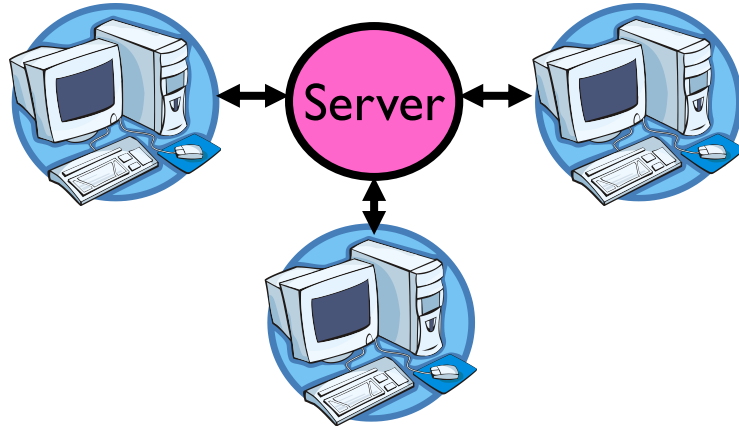
Client/Server Model



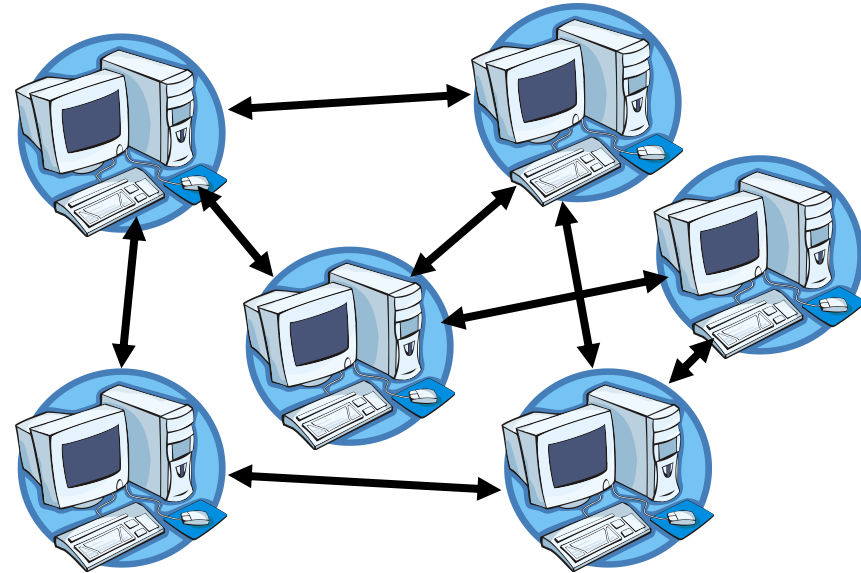
Peer-to-Peer Model

- **Centralized System:** System in which major functions are performed by a single physical computer
 - Originally, everything on single computer
 - Later: client/server model

Centralized vs Distributed Systems



Client/Server Model



Peer-to-Peer Model

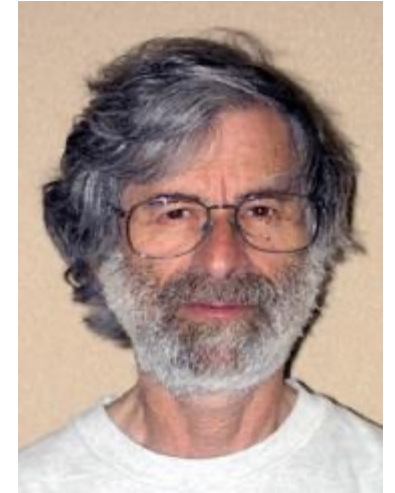
- **Distributed System:** physically separate computers working together on some task
 - Early model: multiple servers working together
 - » Probably in the same room or building
 - » Often called a “cluster”
 - Later models: peer-to-peer/wide-spread collaboration

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

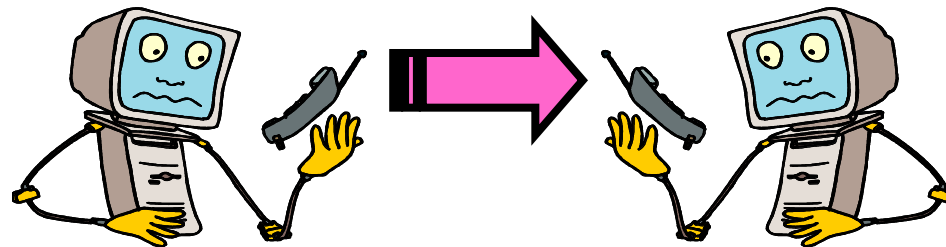
- Reality has been disappointing
 - *Worse availability*: depend on every machine being up
 - » Lamport: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.”
 - *Worse reliability*: can lose data if any machine crashes
 -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



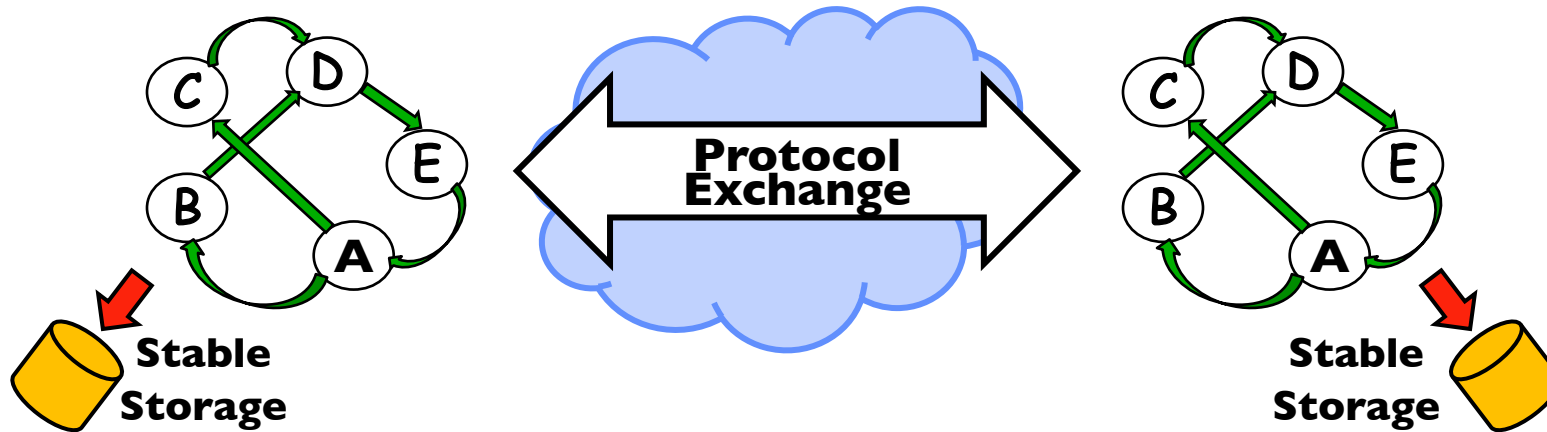
Leslie Lamport

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!

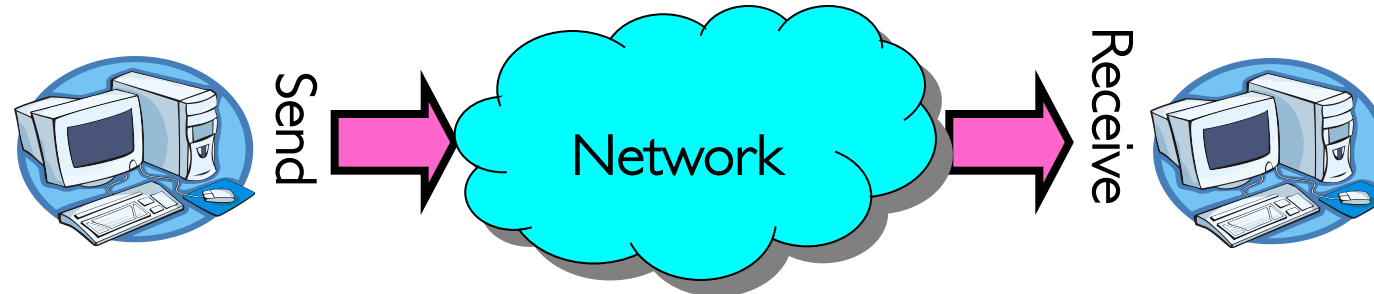
Examples of Protocols in Human Interactions

- 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

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

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?

Two General's Paradox

- Two 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

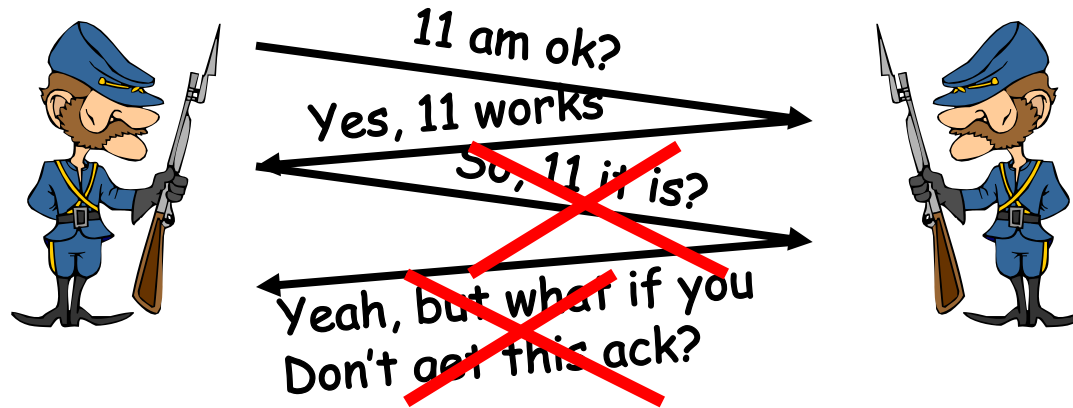
- » If they attack at different times, they all die
- » If they attack at same time, they win

- Named after Custer, who died at Little Big Horn because he arrived a couple of days too early



Two 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 Two 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

Two-Phase Commit 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 on 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

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

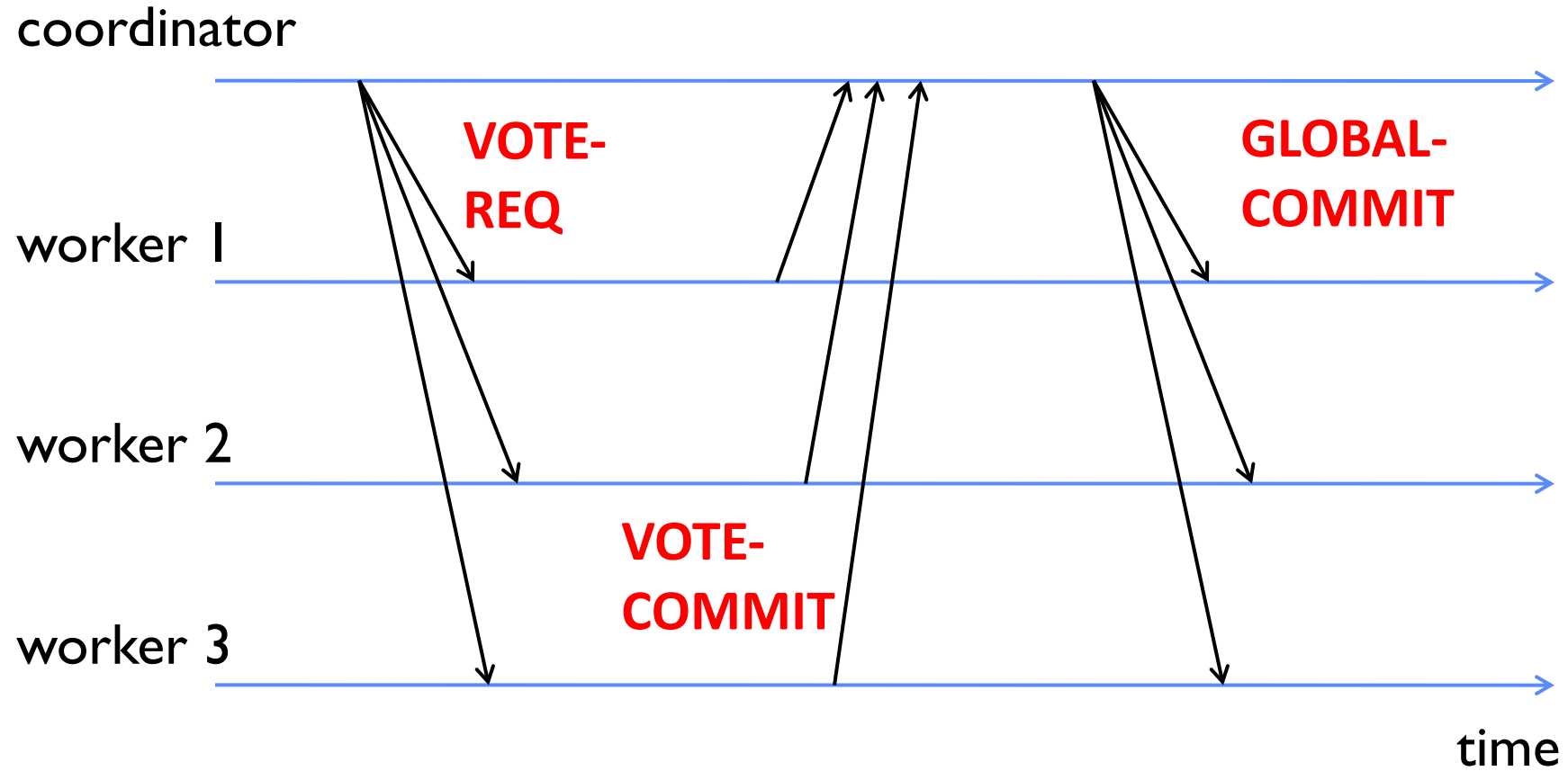
- If receive **VOTE-COMMIT** from all N workers, send **GLOBAL-COMMIT** to all workers
- If don't receive **VOTE-COMMIT** from all N workers, send **GLOBAL-ABORT** to all workers

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

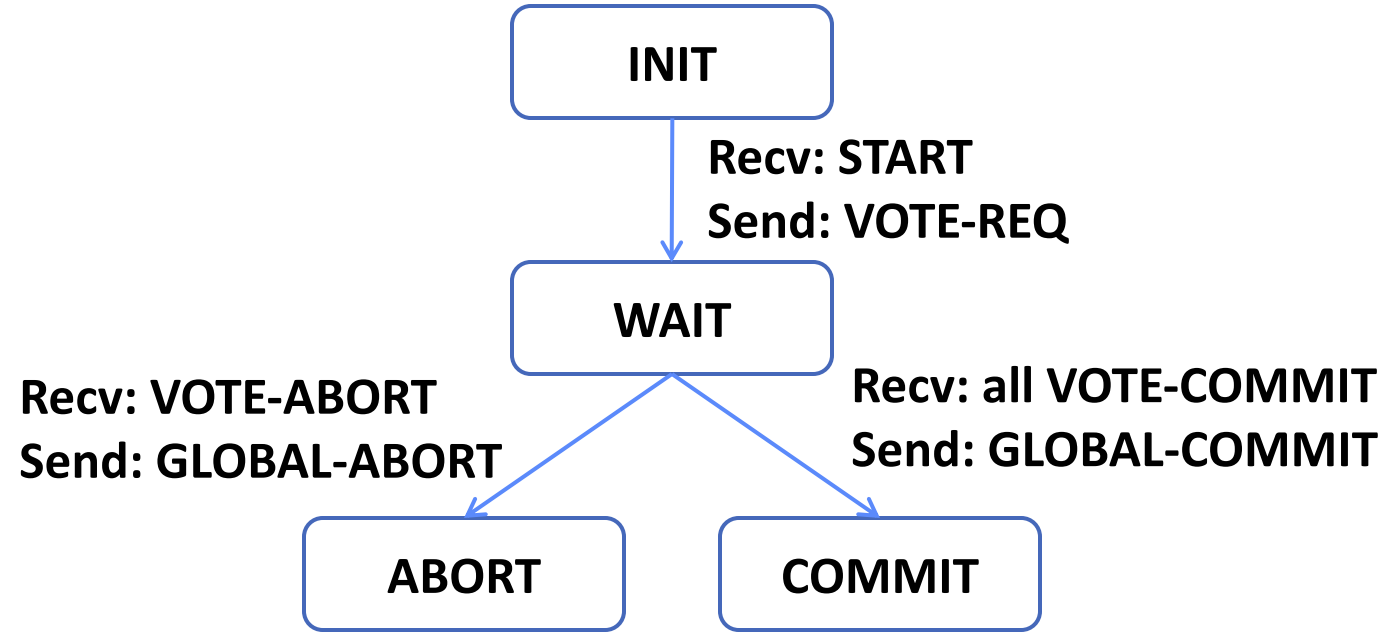
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution

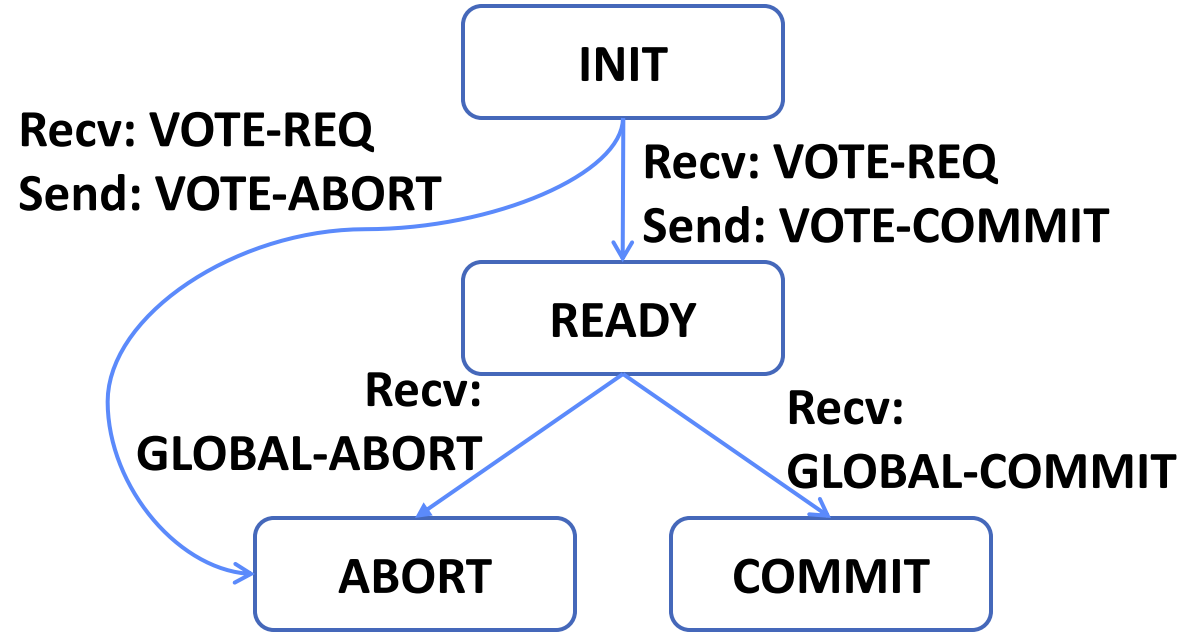


State Machine of Coordinator

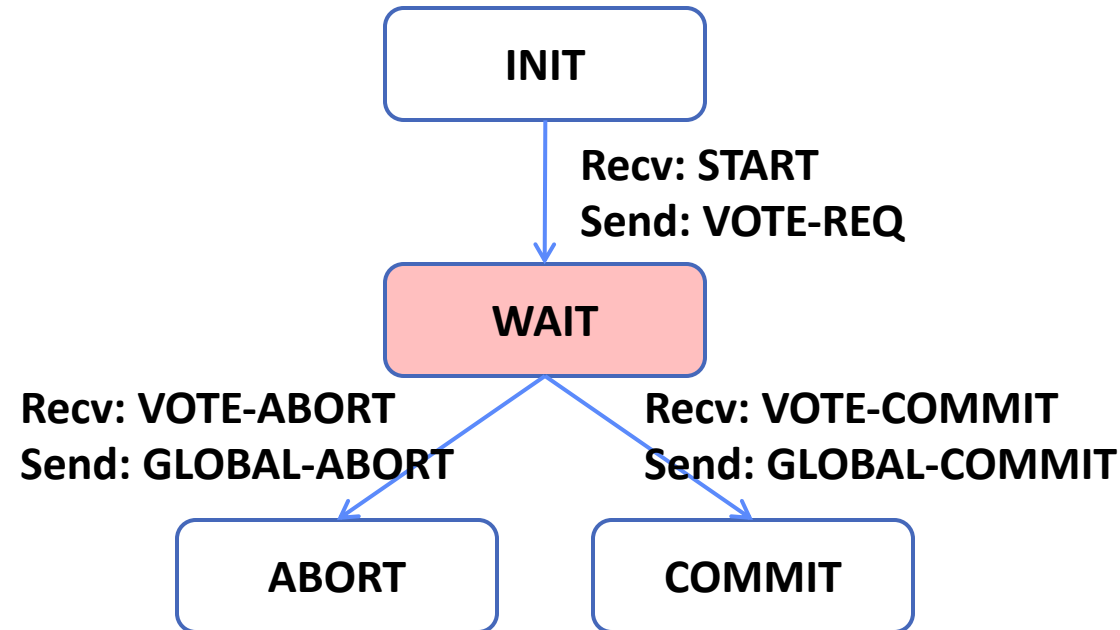
- Coordinator implements simple state machine:



State Machine of Workers

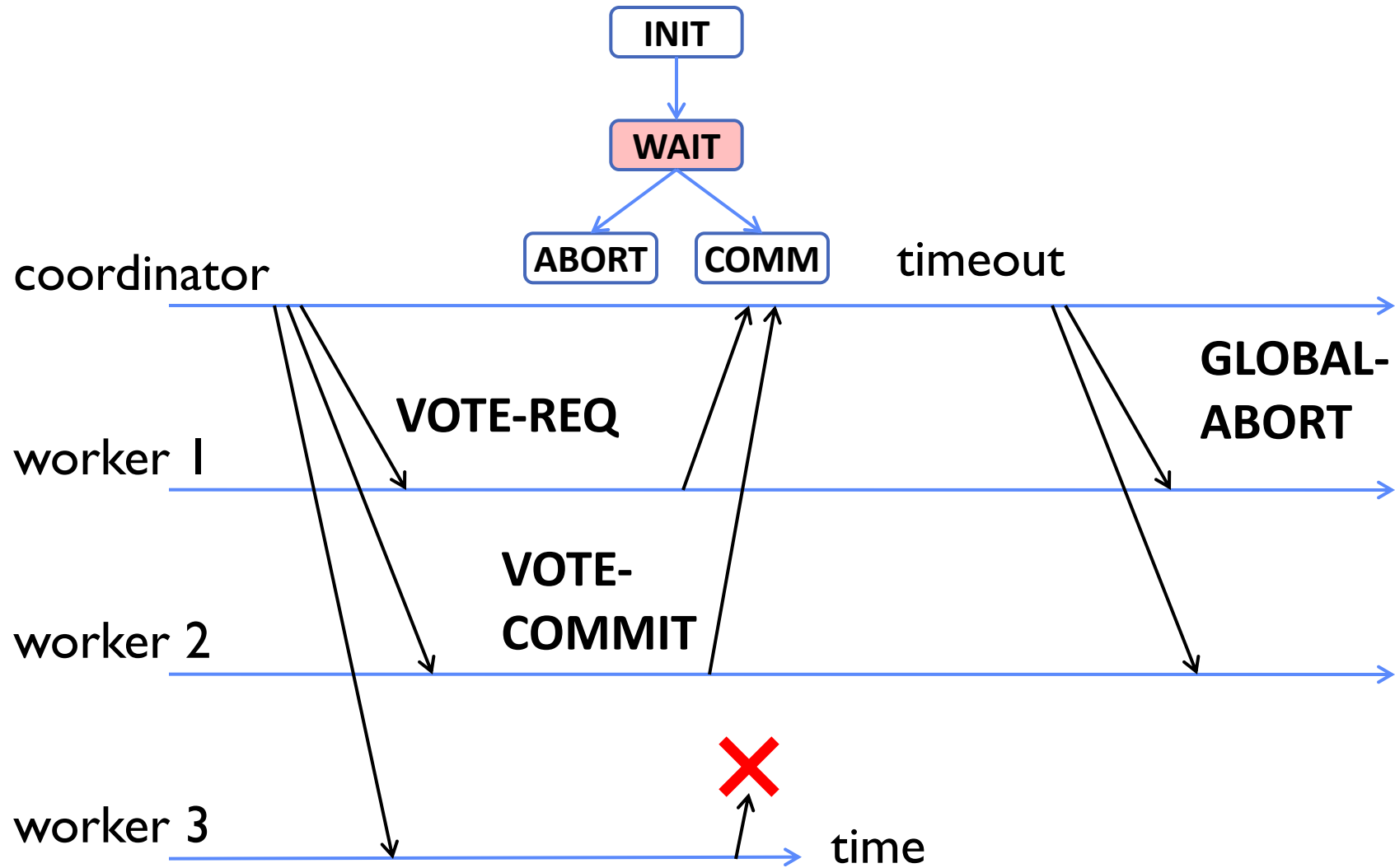


Dealing with Worker Failures

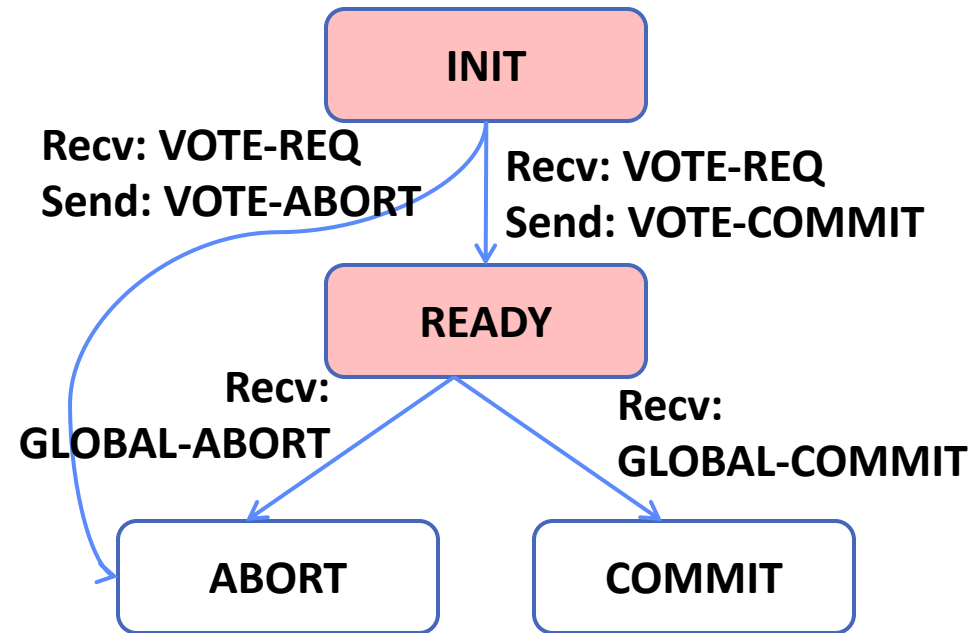


- Failure only affects states in which the coordinator is waiting for messages
- Coordinator only waits for votes in “WAIT” state
- In **WAIT**, if doesn't receive N votes, it times out and sends **GLOBAL-ABORT**

Example of Worker Failure

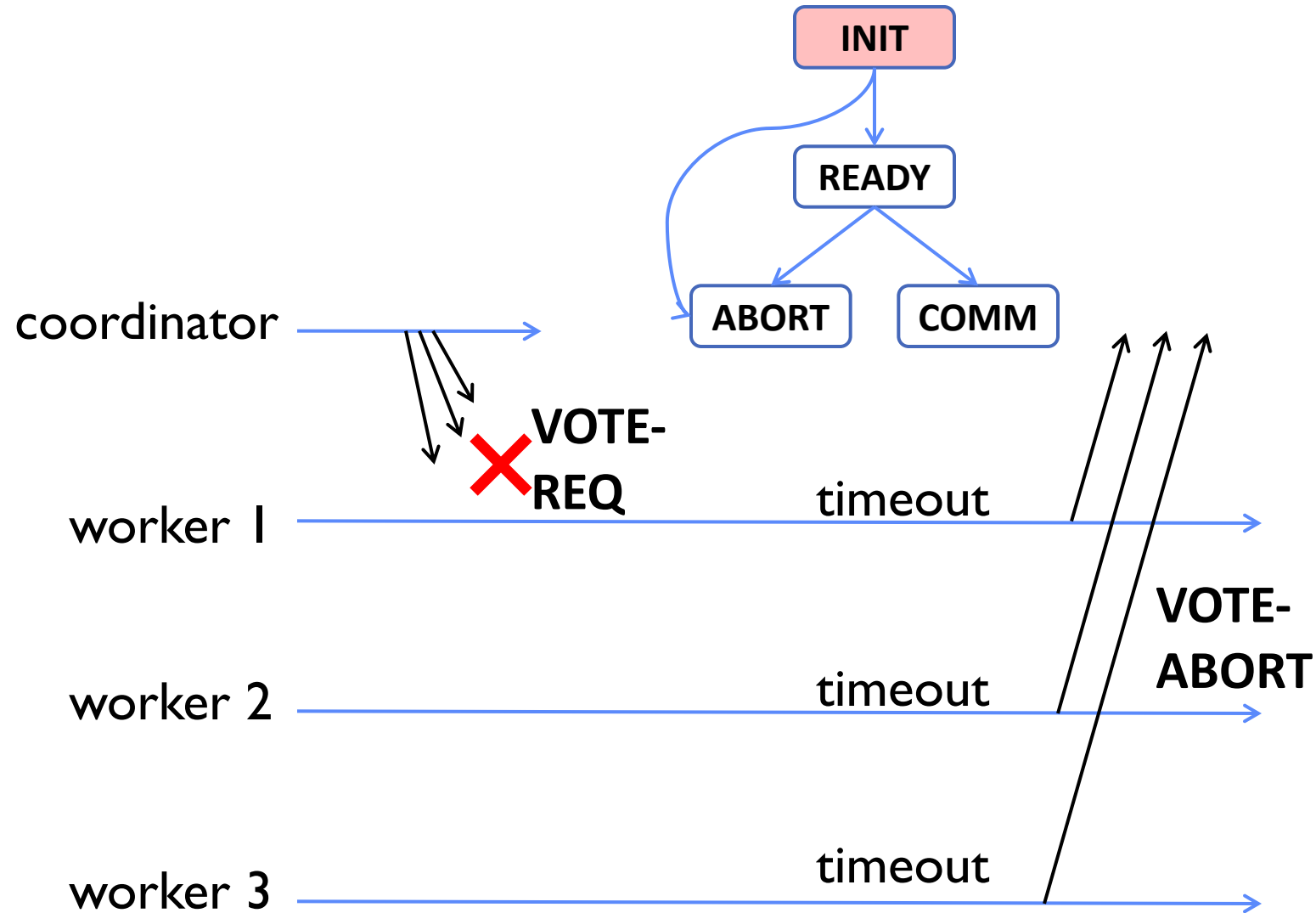


Dealing with Coordinator Failure

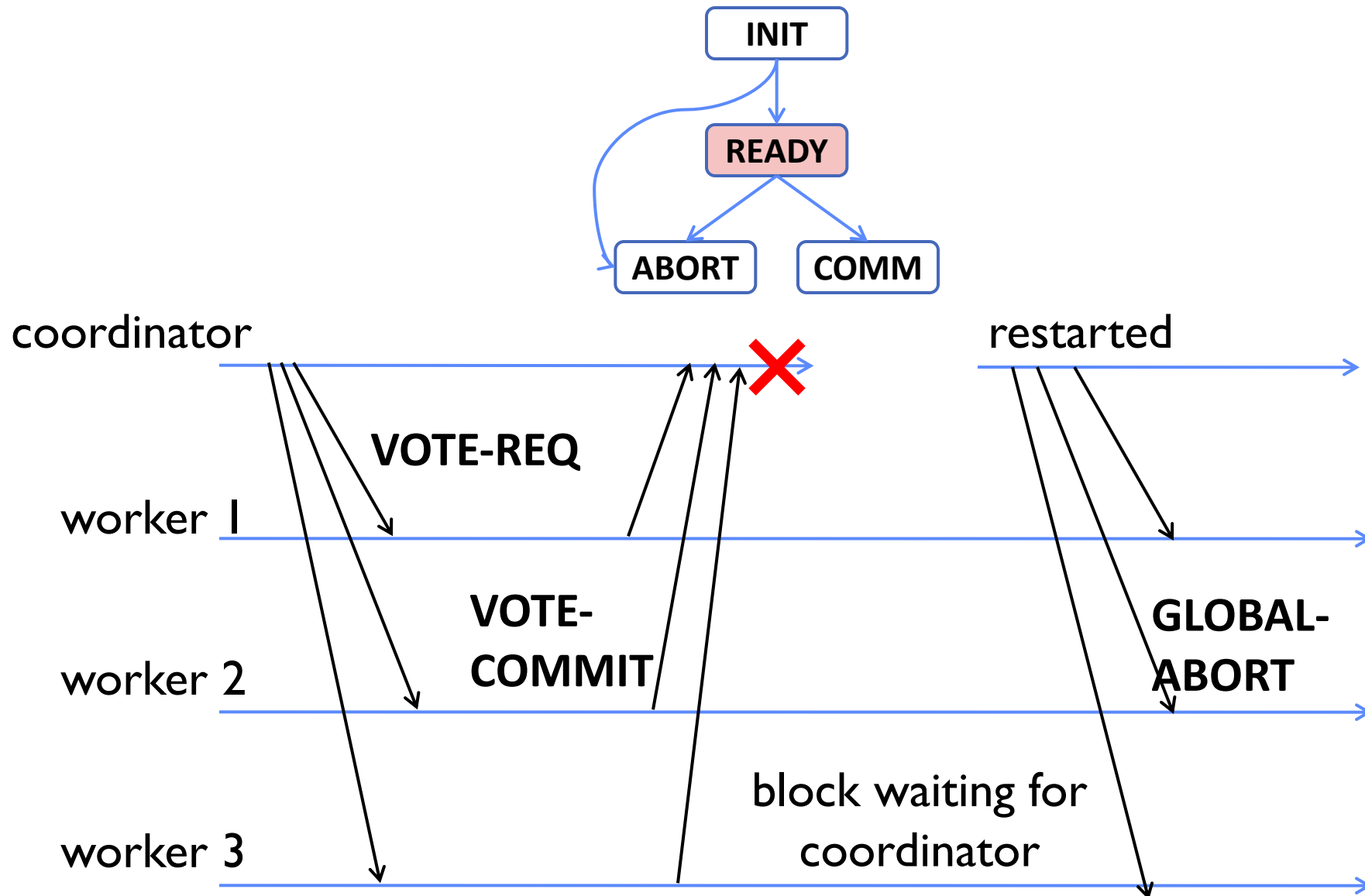


- Worker waits for VOTE-REQ in INIT
 - Worker can time out and abort (coordinator handles it)
- Worker waits for GLOBAL-* message in READY
 - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message

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, 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

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”
 - After decision made, result recorded in multiple places
- Why is 2PC not subject to the Two General’s paradox?
 - Because 2PC is about *all nodes eventually coming to the same decision – not necessarily at the same time!*
 - Allowing us to reboot and continue allows time for collecting and collating decisions

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

Summary (1/2)

- Important system properties
 - **Availability**: how often is the resource available?
 - **Durability**: how well is data preserved against faults?
 - **Reliability**: how often is resource performing correctly?
- **RAID**: Redundant Arrays of Inexpensive Disks
 - RAID1: mirroring, RAID5: Parity block
- Copy-on-write provides richer function (versions) with much simpler recovery
 - Little performance impact since sequential write to storage device is nearly free
- Use of Log to improve Reliability
 - Journalled file systems such as ext3, NTFS

Summary (2/2)

- Transactions over a log provide a general solution
 - Commit sequence to durable log, then update the disk
 - Log takes precedence over disk
 - Replay committed transactions, discard partials
- Protocol: Agreement between two parties as to how information is to be transmitted
- Two-phase commit: distributed decision making
 - First, make sure everyone guarantees they will commit if asked (prepare)
 - Next, ask everyone to commit