

CS162
Operating Systems and
Systems Programming
Lecture 22

End-to-End Arguments, Distributed Decision Making

Centralised vs Distributed Systems



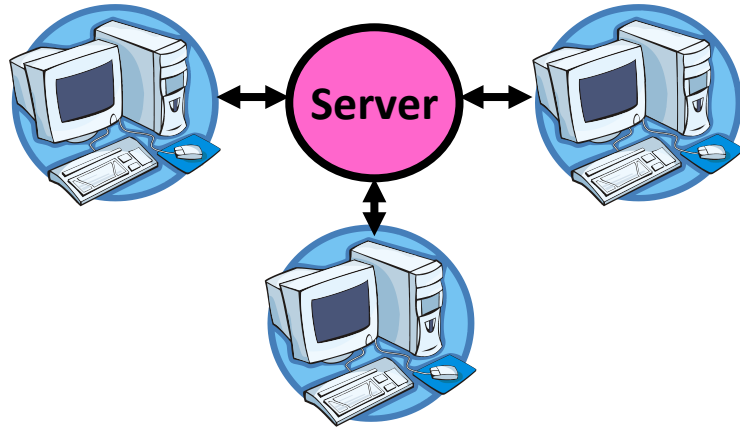
Considered a single computer! All computation was done on the local computer in isolation



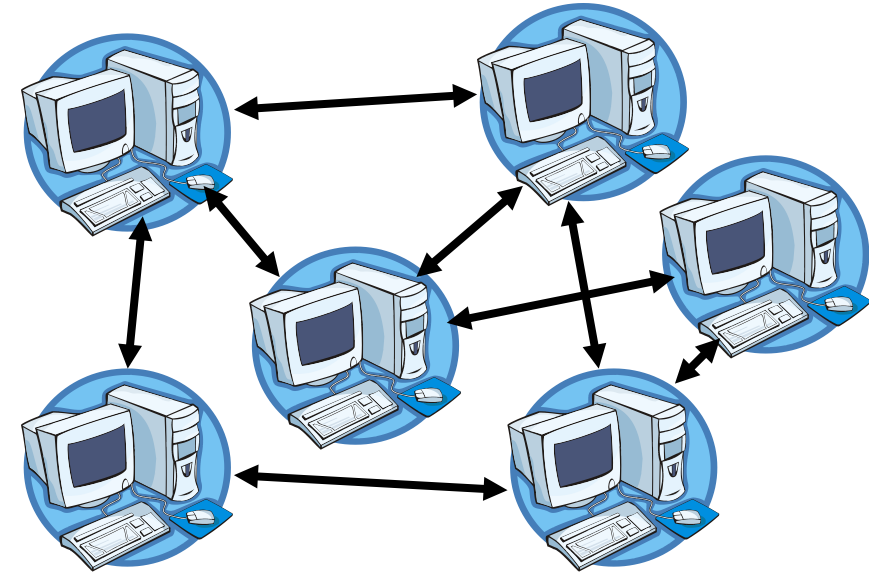
The world is a large distributed system

- Microprocessors in everything
- Vast infrastructure behind them

Two types of distributed systems



Client/Server Model



Peer-to-Peer Model

One or more server provides services to clients

Clients makes *remote procedure calls* to server
Server serves *requests* from clients

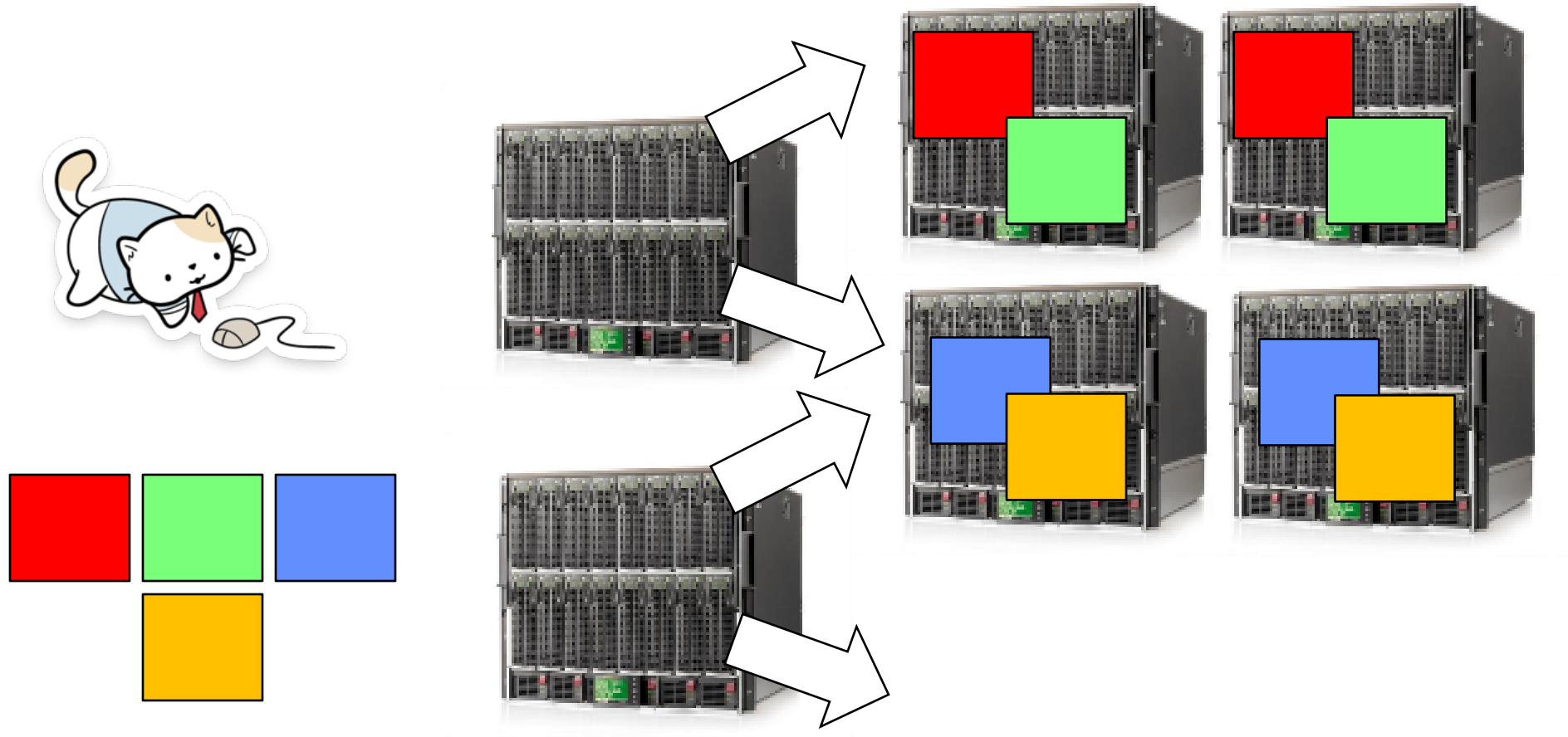
Hierarchical relationship between client and server

Each computer acts as a peer

No hierarchy or central point of coordination

All-way communication between peers through *gossiping*

Example: How do I store all my data?



The promise of distributed systems

Availability

Proportion of time system is in functioning condition

=> One machine goes down, use another

Fault-tolerance

System has well-defined behaviour when fault occurs

=> Store data in multiple locations

Scalability

Ability to add resources to system to support more work

=> Just add machines when need more storage/processing power

The requirements of distributed systems

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

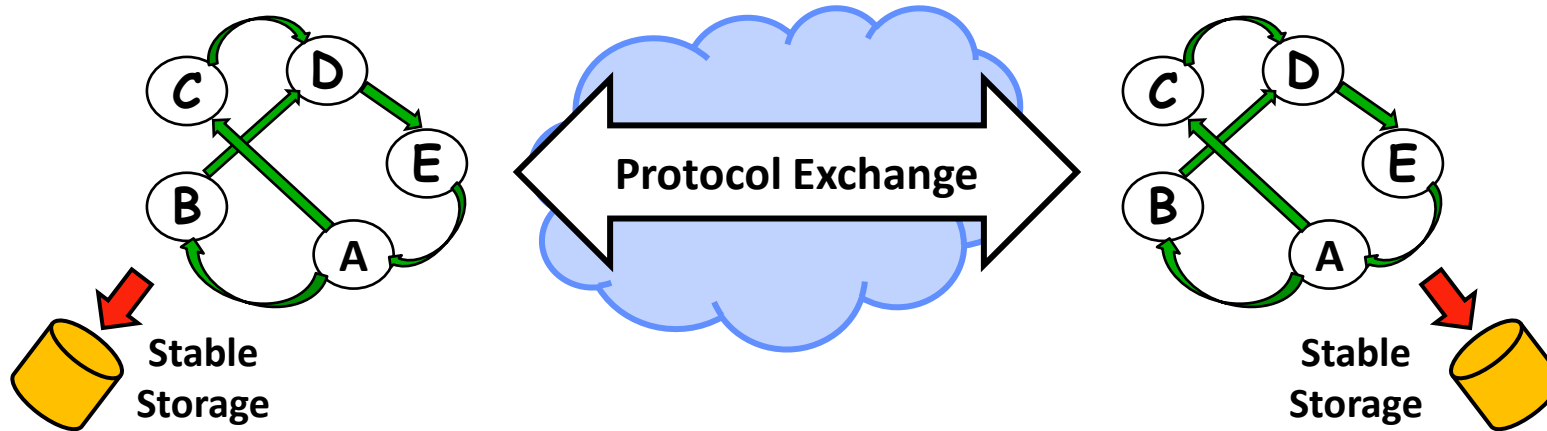
The challenges of distributed systems

- How do you get machines to **communicate**?
- How do you get machines to **coordinate**?
- How do you deal with **failures**?
- How do you deal with **security** (corrupted machines)?

Lecture roadmap







- How do machines communicate?
 - *Through protocols*
- Case study: The Internet
 - *Layering*
 - *The End-To-End Principle*
- How do machines coordinate?
 - *Hint: it's hard!*
 - *2 Phase Commit*
 - *Hint: it's even harder when machines aren't honest*
 - *PBFT/Blockchains*

How do entities communicate? A Protocol!



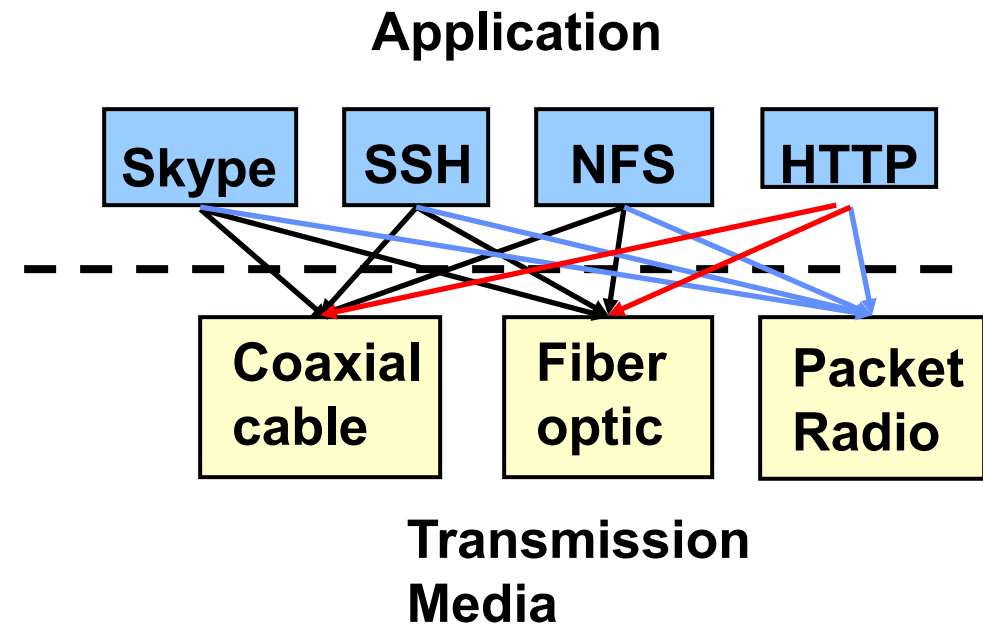
- A protocol is **an agreement on how to communicate**, including:
 - **Syntax**: how a communication is specified & structured
 - » Format, order messages are sent and received
 - **Semantics**: what a communication means
 - » Actions taken when transmitting, receiving, or when a timer expires
- Described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

Examples of Protocols in Human Interactions

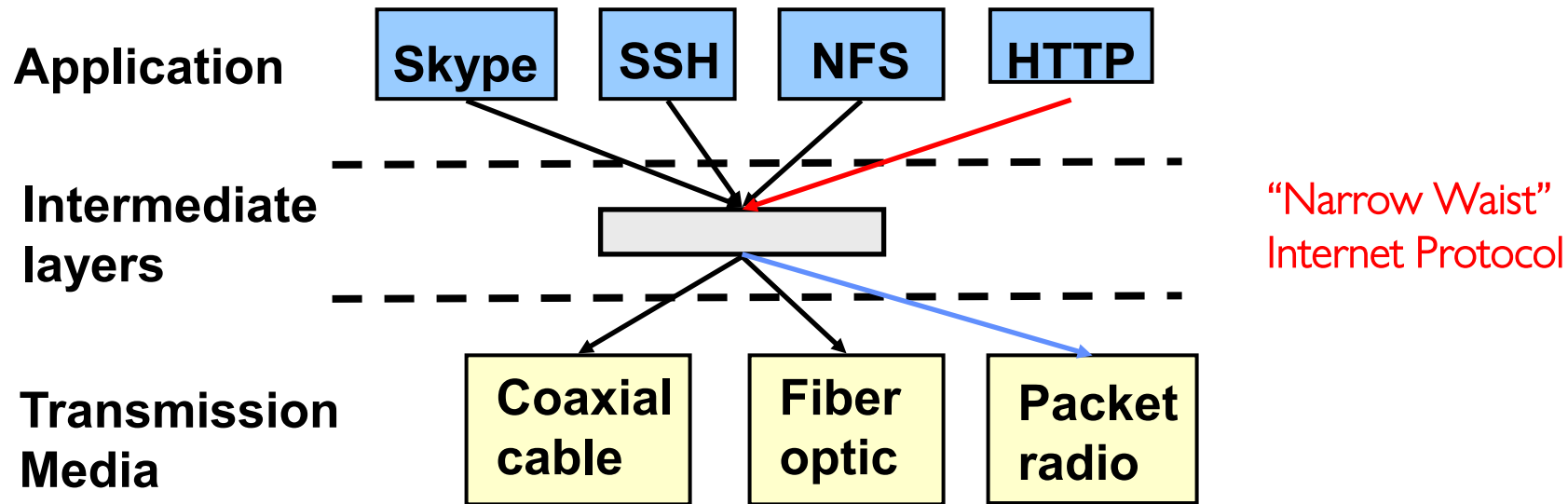
- Telephone
 1. (Pick up / open up the phone)
 2. Listen for a dial tone / see that you have service
 3. Dial
 4. Should hear ringing ...
 5.  Callee: "Hello?"
 6. Caller: "Hi, it's Natacha..."
Or: "Hi, it's me" (← what's *that* about?)
 7. Caller: "Hey, do you think ... blah blah blah ..." **pause**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

Case study: The Internet

- The Internet is the largest distributed system that exists!
- Many different applications
 - Email, web, P2P, etc.
- Many different operating systems and devices
- Many different network styles and technologies
 - Wireless, wired, optical
- How do we organize this mess
 - Layering & end-to-end principle

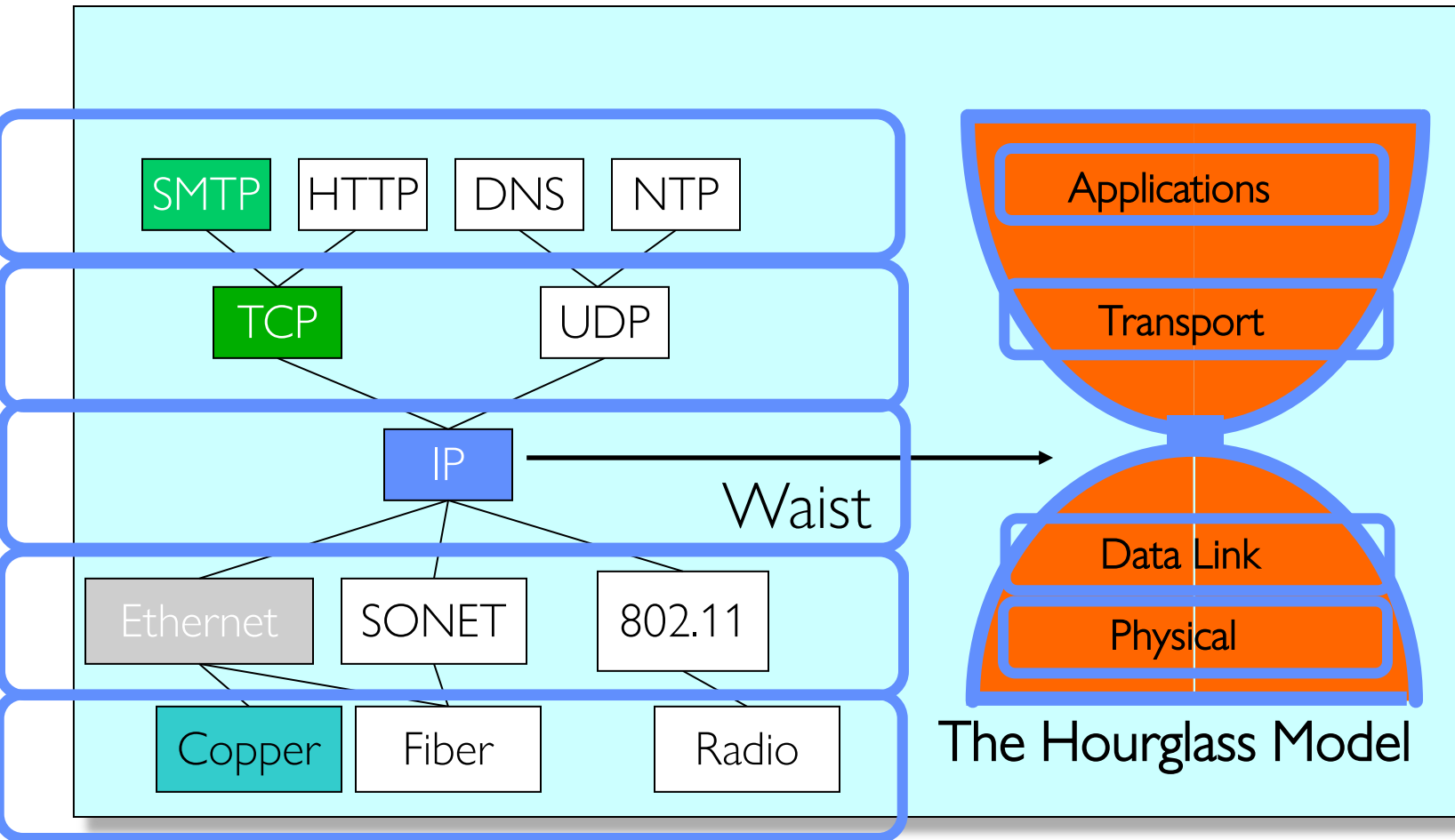


The Internet: Layers, Layers, Layers



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

The Internet: The *hourglass*



“Narrow waist” facilitates *interoperability*

Layers “abstract” away hardware so that upper layers are agnostic to lower layers

=> Sound familiar?

The Internet: 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

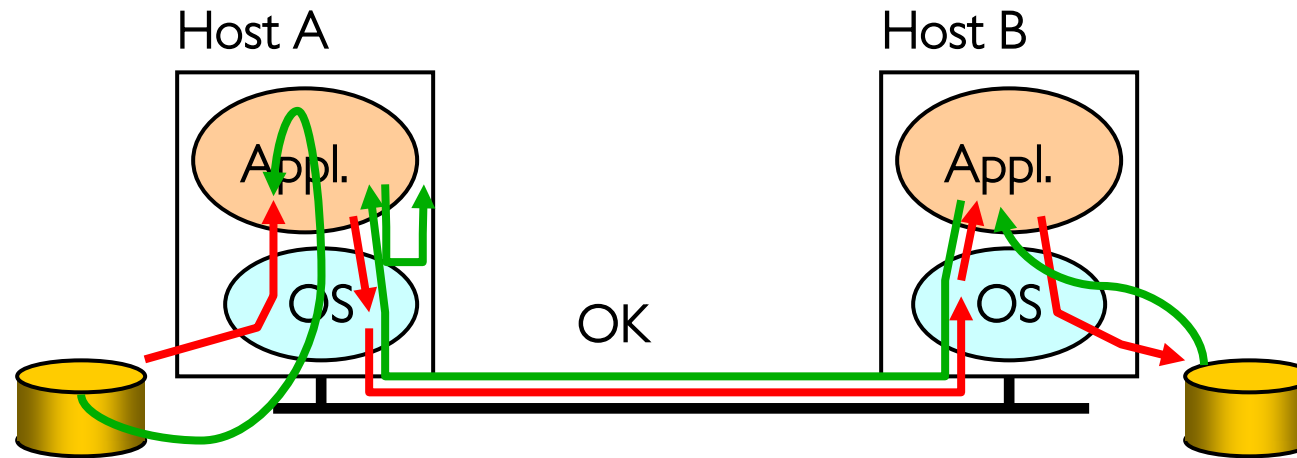
The Internet: Drawbacks of Layering

- 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)

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.
- Hosts cannot rely on the network help to meet requirement, so must implement it themselves

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 always 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

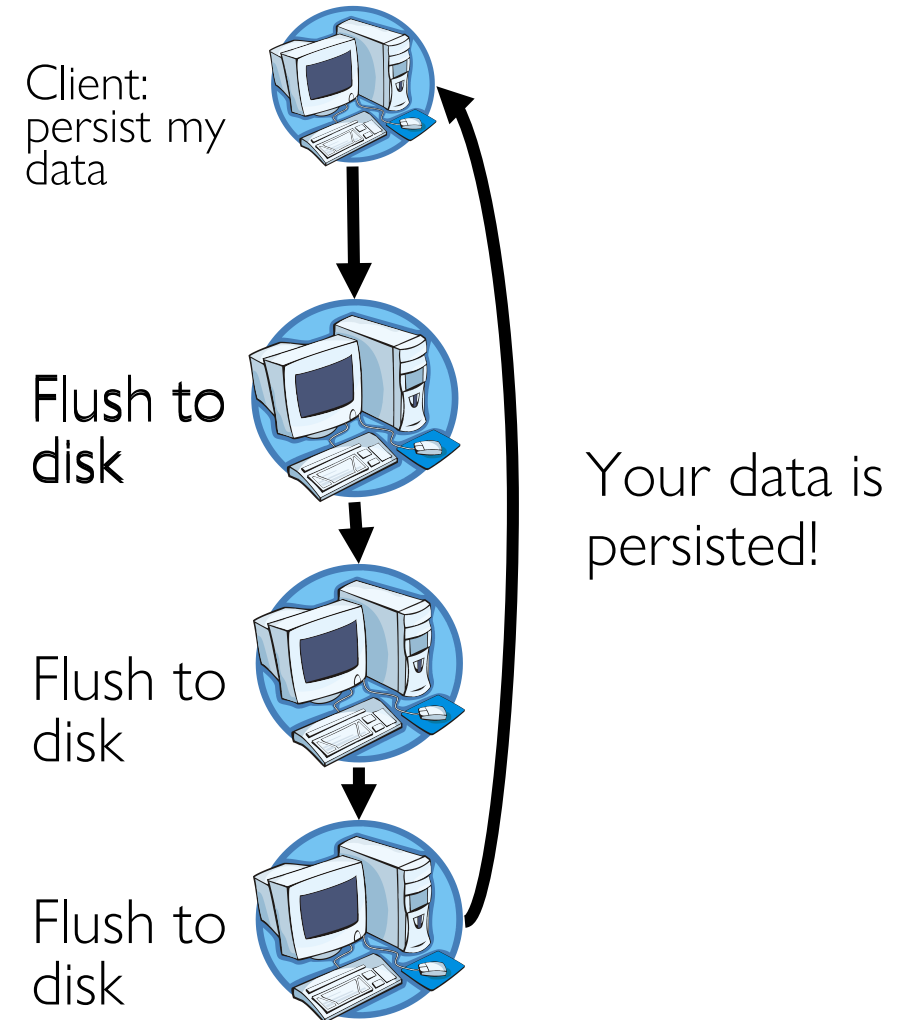
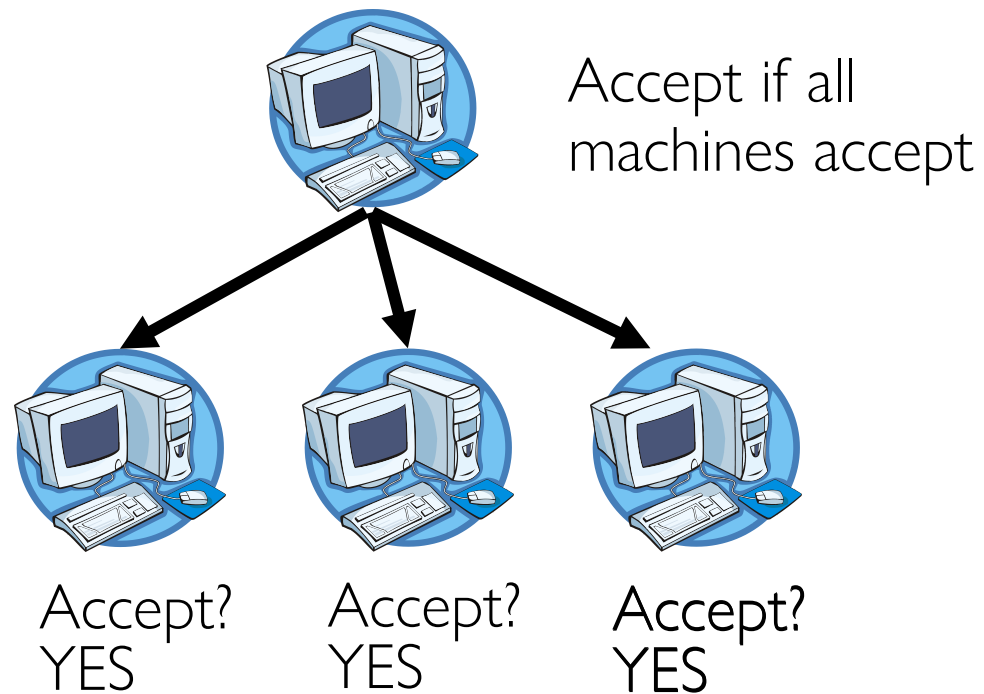
- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level
- Or: Unless you can relieve the burden from hosts, don't bother

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

Coordination: making distributed decisions

- Functionality is spread across machines. Requires coordination to reach distributed decision



Coordination is hard!

- When machines can fail!
- When networks are slow and/or unreliable
- When machines may receive conflicting proposals on what to do

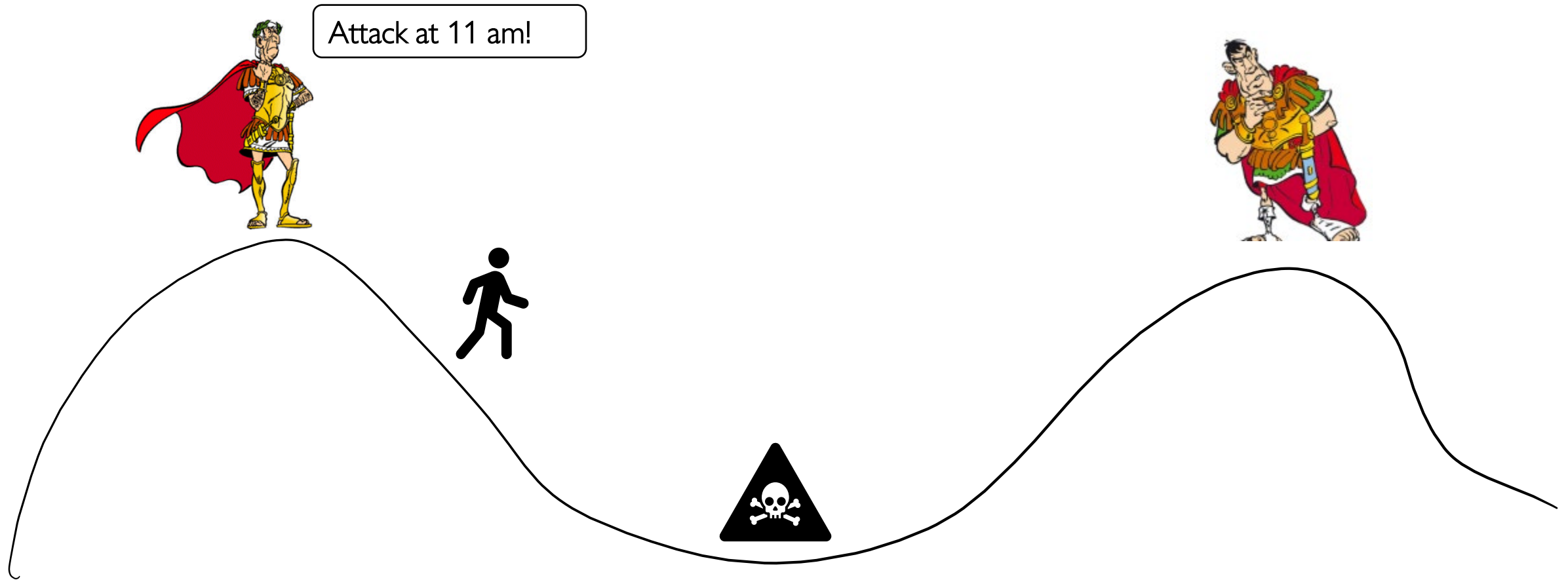
Agreeing simultaneously: General's Paradox

- 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

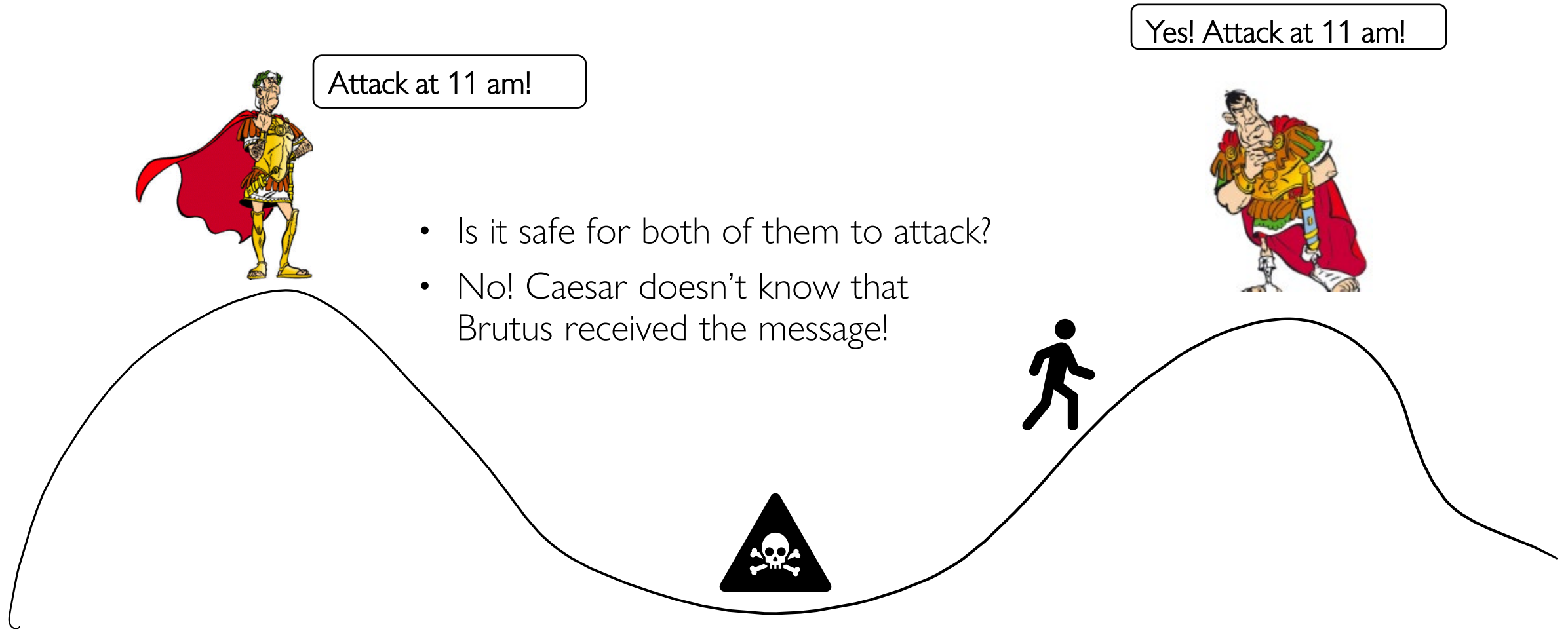


Can messages over an unreliable network be used to guarantee two entities do something simultaneously?

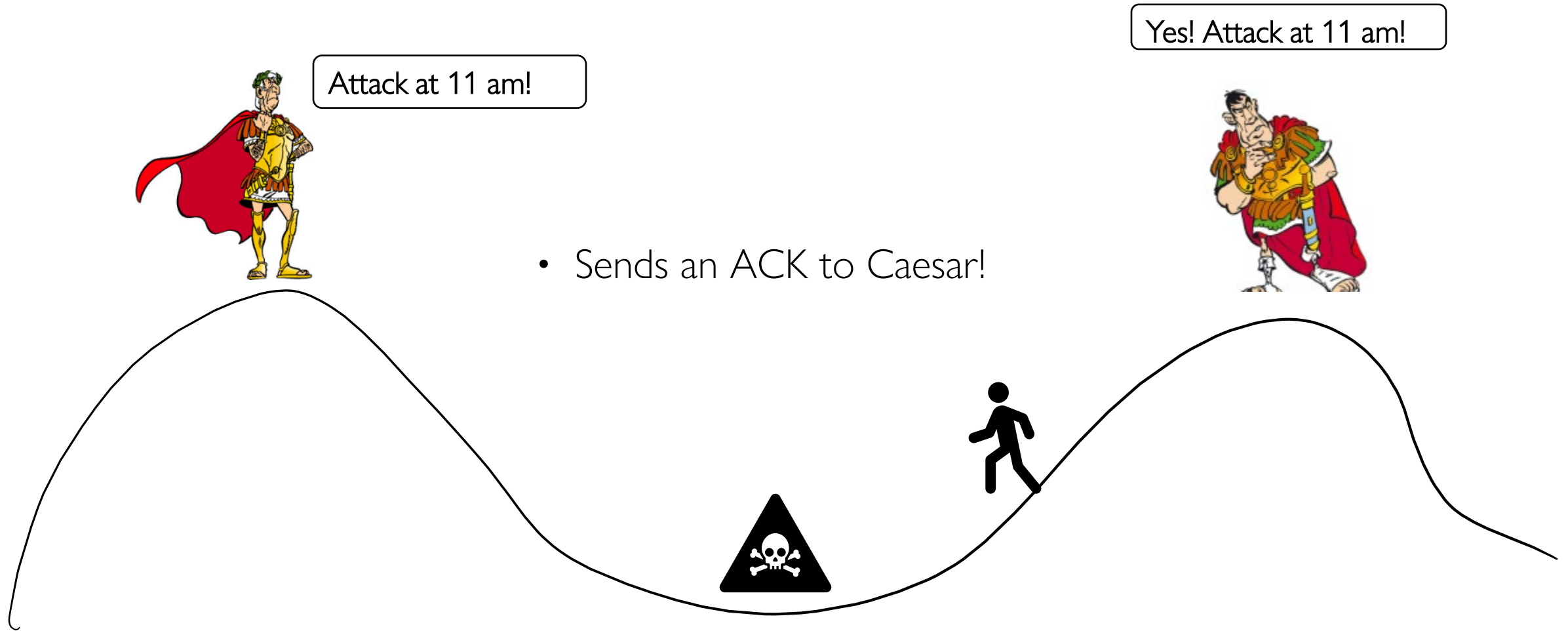
General's Paradox: Scenario 1



General's Paradox: Scenario 1



General's Paradox: Scenario 1



General's Paradox: Scenario 1

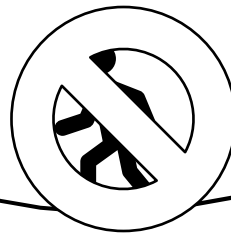


Attack at 11 am!

- Now is it safe?
- No! Messenger could have been attacked



Yes! Attack at 11 am!



General's Paradox: Scenario 1



Caesar needs to know that
Brutus knows that
Caesar knows that
Brutus knows that
They are attacking at 11 am



Impossible to achieve simultaneous
actions with unreliable channels
because never know whether
messenger or ACK got lost

Eventual Agreement: 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
- Used in most modern distributed systems! Representative of other coordination protocols



Jim Gray

Eventual Agreement: Two-Phase Commit

Goal: determine whether should commit or abort a transaction

- All processes that reach a decision reach the same one (*Agreement*)
- A process cannot reverse its decision after it has reached one (*Finality*)
- If there are no failures and every process votes yes, the decision will be commit (*Consistency*)
- If all failures are repaired and there are no more failures, then all processes will eventually decide commit/abort (*Termination*)

2PC Terminology

- Setup:
 - One *coordinator*
 - A set of *participants*
- Each process has access to a *persistent log*:
 - recorded information on the log will persist after crashes
- Coordinator asks all processes to vote
- Each participant (including coordinator) can vote either YES or NO
 - If all vote YES, coordinator must vote COMMIT
 - If one of them votes NO, coordinator must vote ABORT
- Example use in databases:
 - If a database transaction executes on multiple machines, used to determine whether all machines agree that transaction should commit

2PC: The easy case (No failures)

Coordinator Algorithm

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

3. Collect votes

- 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

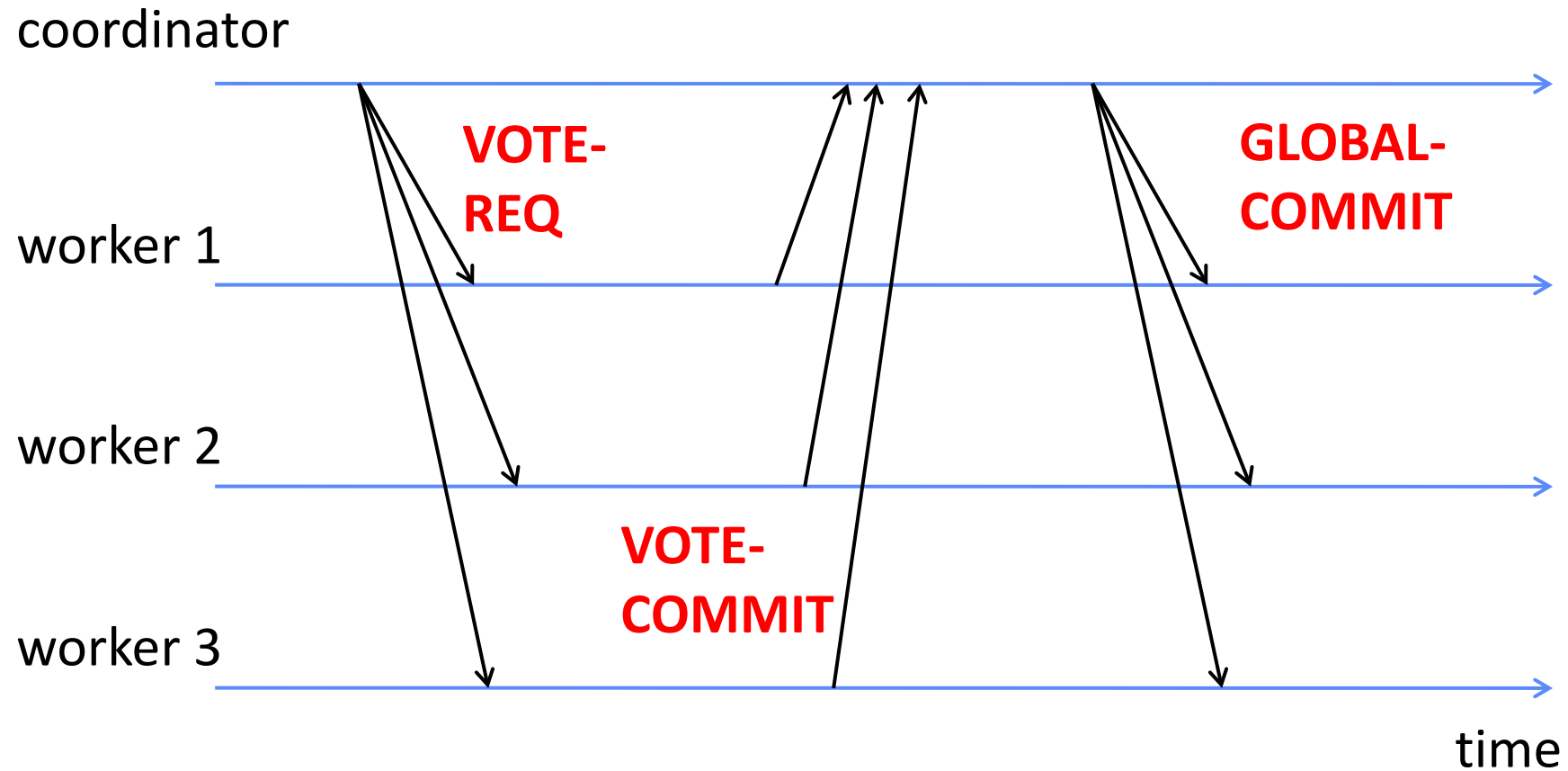
2.

- Send **VOTE-COMMIT** or **VOTE-ABORT** to coordinator
- If sent **VOTE-ABORT** immediately abort

4.

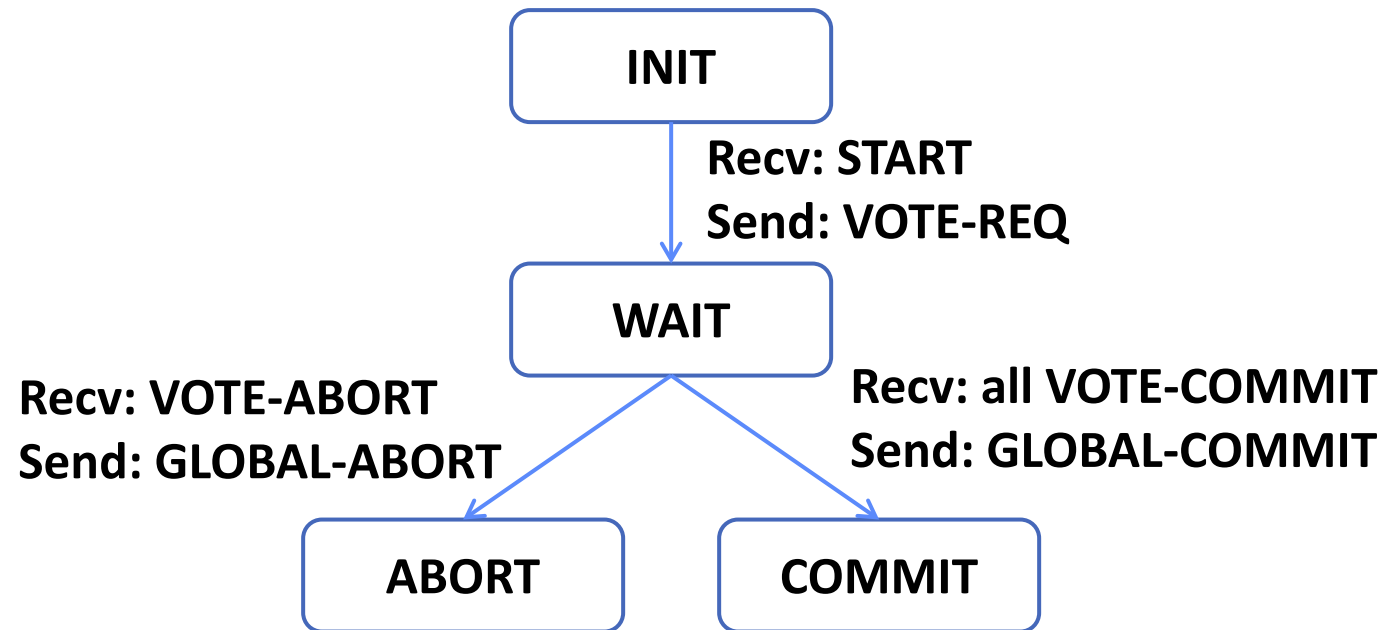
- If receive **GLOBAL-COMMIT** then commit
- If receive **GLOBAL-ABORT** then abort

Failure Free Example Execution



State Machine of Coordinator

- Coordinator implements simple state machine:



What about failures?

Coordinator Algorithm

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

Worker Algorithm

- 2.
- Send **VOTE-COMMIT** or **VOTE-ABORT** to coordinator
 - If sent **VOTE-ABORT** immediately abort

3. Collect votes
- 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

- 4.
- If receive **GLOBAL-COMMIT** then commit
 - If receive **GLOBAL-ABORT** then abort

1) What happens when waiting for a message that never comes?

2) What happens during when participant recovers from a failure?

What happens when a message never comes?

Failure only affects states in which waiting for messages

Coordinator
Algorithm

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

Worker
Algorithm

2.
– Send **VOTE-COMMIT** or **VOTE-ABORT** to coordinator
– If sent **VOTE-ABORT** immediately abort

3. Collect votes
– 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

4.
– If receive **GLOBAL-COMMIT** then commit
– If receive **GLOBAL-ABORT** then abort

- Step 2: worker waiting from VOTE-REQ from coordinator
- Step 3: Coordinator is waiting for vote from participants
- Step 4: Worker who voted YES is waiting for decision

What happens when a message never comes?

Coordinator Algorithm

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

3. Collect votes

- 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

2.

- Send **VOTE-COMMIT** or **VOTE-ABORT** to coordinator
- If sent **VOTE-ABORT** immediately abort

4.

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

- Step 2: worker waiting from VOTE-REQ from coordinator
Since it has not cast its vote yet, worker can decide abort and halt
- Step 3: Coordinator is waiting for vote from participants
Coordinator can always vote abort herself, so votes abort and sends GLOBAL-ABORT to all participants
- Step 4: Worker who voted COMMIT is waiting for decision
Worker cannot decide: it must run a termination protocol

Termination Protocol

- Option 1: Simply wait for coordinator to recover.

If all failures are repaired and there are no more failures, then all processes will eventually decide commit/abort (Termination)

=> No need to recover until coordinator has recovered

- (Better) Option 2: Ask a friendly participant p

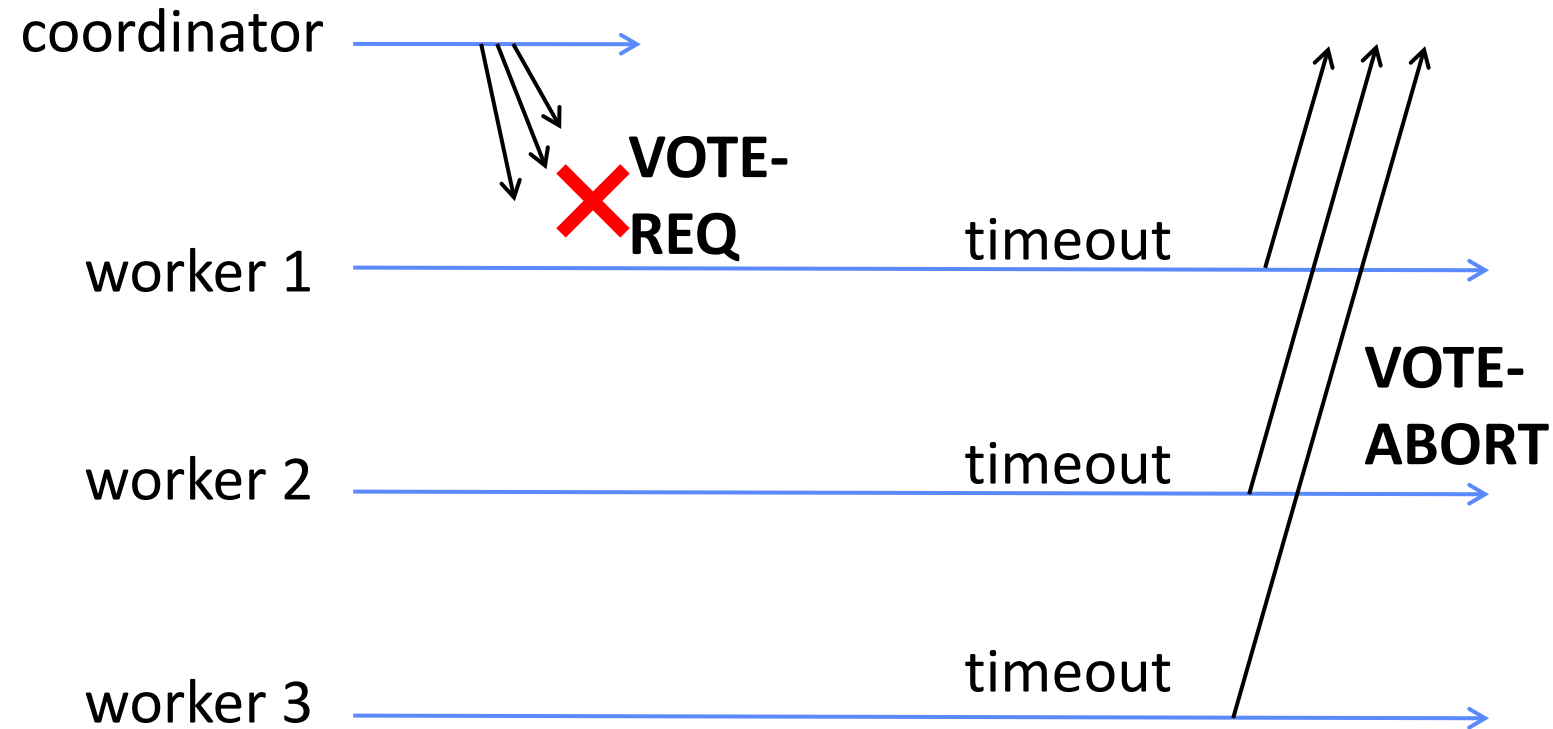
If every participant voted COMMIT and coordinator crashes before sending decision, must wait for coordinator to recover to decide!

Case 1: *If p has decided COMMIT/ABORT, forwards decision to initiator*

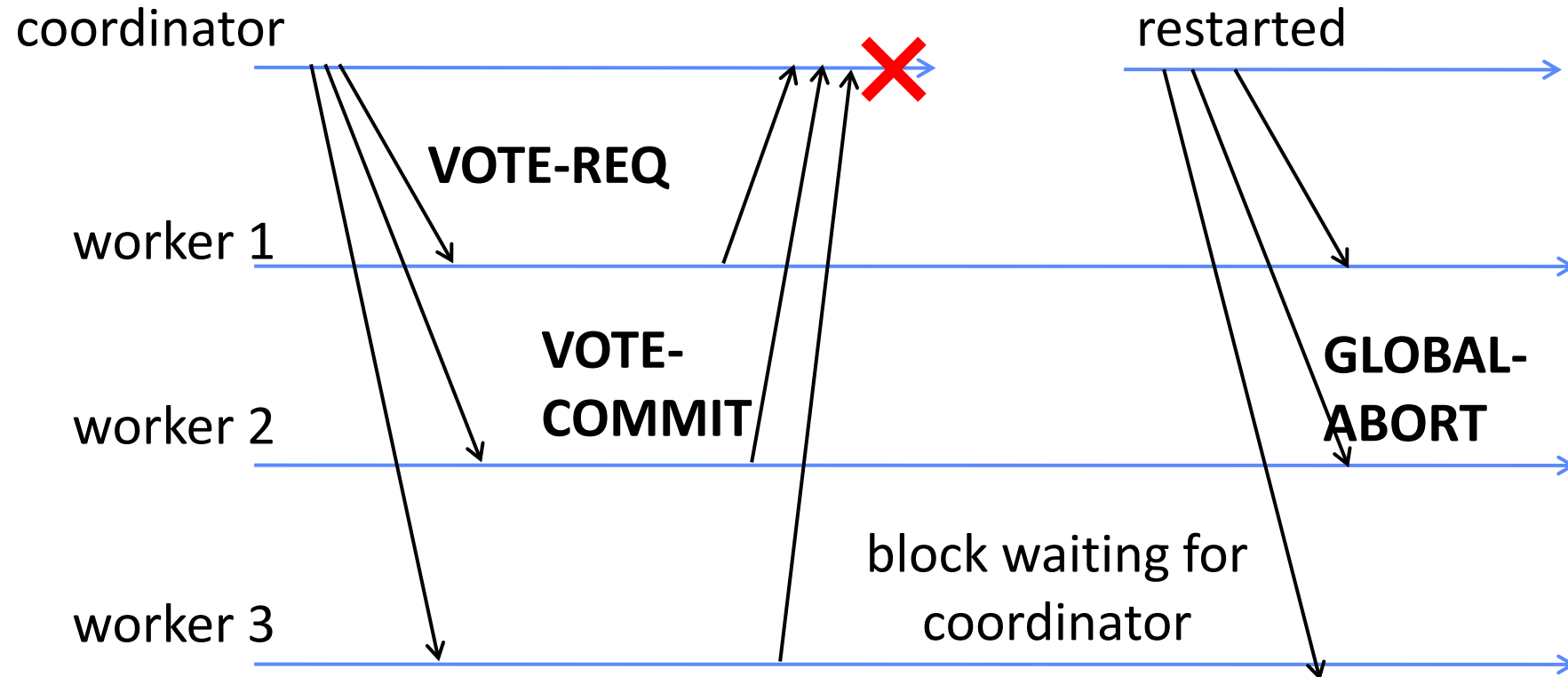
Case 2: *If P has not decided, votes ABORT, sends abort to initiator.
Initiator knows decision will be ABORT. So can decide*

Case 3: *If P has voted COMMIT, P is also stuck and can't help initiator*

Example of Coordinator Failure #1



Example of Coordinator Failure #2



Machine recovery

All nodes use **stable storage** to store current state (e.g. backed by disk/SSD)

Upon recovery, nodes can restore state and resume

When coordinator sends VOTE-REQ, writes START-2PC to log

=> *Coordinator reads log, if sees VOTE-REQ but no decision, decides ABORT unilaterally*

Before voting, participant writes VOTE-* to stable log, then sends vote

=> *Participant reads log, if doesn't see record, sends VOTE-ABORT. If VOTE-COMMIT, contacts friend*

Before sending decision, coordinator writes GLOBAL-* to stable log, then sends decision

=> *Coordinator reads log, if sees GLOBAL-*, resends decision*

After receiving GLOBAL-*, participant writes commit/abort to stable log

=> *Participants read log, 2PC instance has already been terminated*

2PC Summary

- Why is 2PC not subject to the 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
- Biggest downside of 2PC: blocking
 - A failed node can prevent the system from making progress
 - Still one of the most popular coordination algorithms today

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
 - Developed by Leslie Lamport (Turing Award Winner)
 - No fixed leader, can choose new leader on fly, deal with failure
- 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 **Blockchains**

Summary

- 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