Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

October 23, 2012

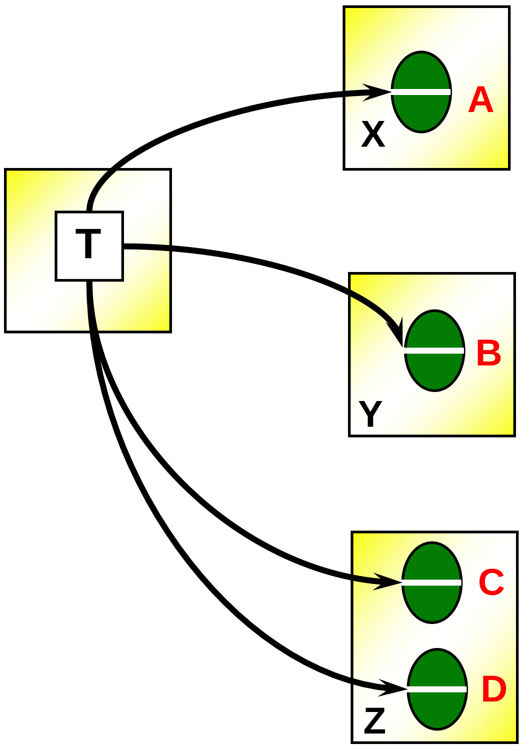
Lecture 17

Two Phase Commit and Paxos

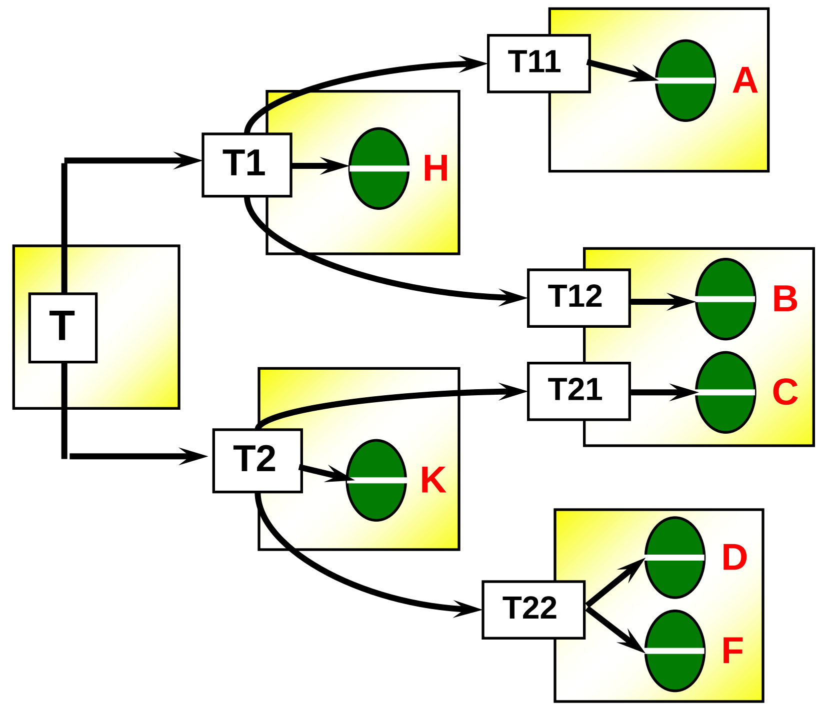
Reading: 21.5.2 (Paxos Sections)

Distributed Transactions

- A transaction that invokes operations at several servers.

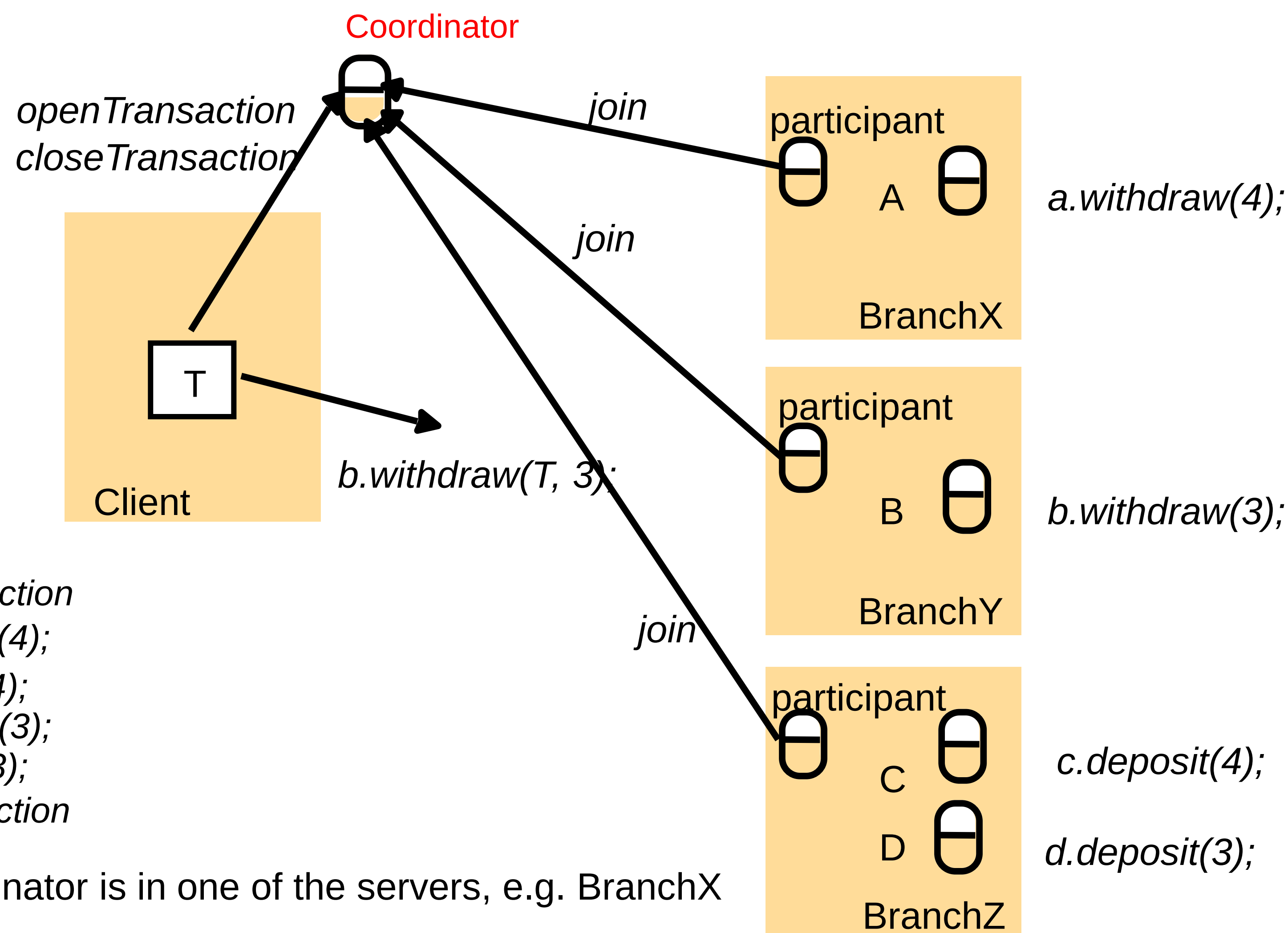


Flat Distributed Transaction



Nested Distributed Transaction

Distributed banking transaction



T = openTransaction
A.withdraw(4);
C.deposit(4);
B.withdraw(3);
D.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

Atomic Commit Problem

- **Atomicity principle requires that either all the distributed operations of a transaction complete, or all abort.**
- **At some stage, client executes closeTransaction(). Now, atomicity requires that either *all* participants (remember these are on the server side) and the coordinator commit or *all* abort.**
- **What problem statement is this?**

Atomic Commit Protocols

- Consensus, but it's impossible in asynchronous networks!
- So, need to ensure *safety property* in real-life implementation. Never have some agreeing to commit, and others agreeing to abort. Err on the side of safety.
- First cut: *one-phase commit* protocol. The coordinator unilaterally communicates either commit or abort, to all participants (servers) until all acknowledge.
  - Doesn't work when a participant crashes before receiving this message (partial transaction results are lost).
  - Does not allow participant to abort the transaction, e.g., under error conditions.

Atomic Commit Protocols

- Consensus, but it′s impossible in asynchronous networks!
- So, need to ensure *safety property* in real-life implementation. Never have some agreeing to commit, and others agreeing to abort. Err on the side of safety.
- Alternative: *Two-phase commit* protocol
  - First phase involves coordinator collecting a vote (commit or abort) from each participant
    - Participant stores partial results in permanent storage before voting
  - Now coordinator makes a decision
  - If all participants want to commit and no one has crashed, coordinator multicasts "commit" message
    - Everyone commits
  - If any participant has crashed or aborted, coordinator multicasts "abort" message to all participants
    - Everyone aborts

RPCs for Two-Phase Commit Protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote. Phase 1.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction. Phase 2.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction. Phase 2.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still has received no reply within timeout. Used to recover from server crash or delayed messages.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction. (May not be required if getDecision() is used)

The two-phase commit protocol

Phase 1 (voting phase):

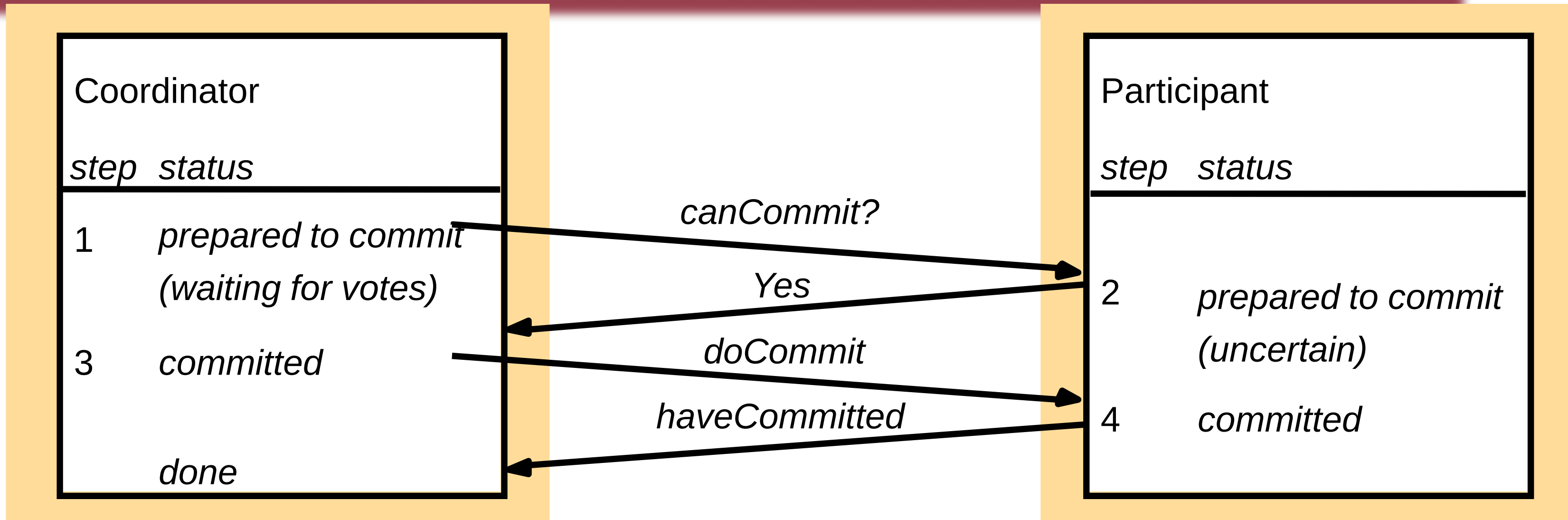
1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request, it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If its vote is *No,* the participant aborts immediately.

Recall that a server may crash

Phase 2 (completion according to outcome of vote):

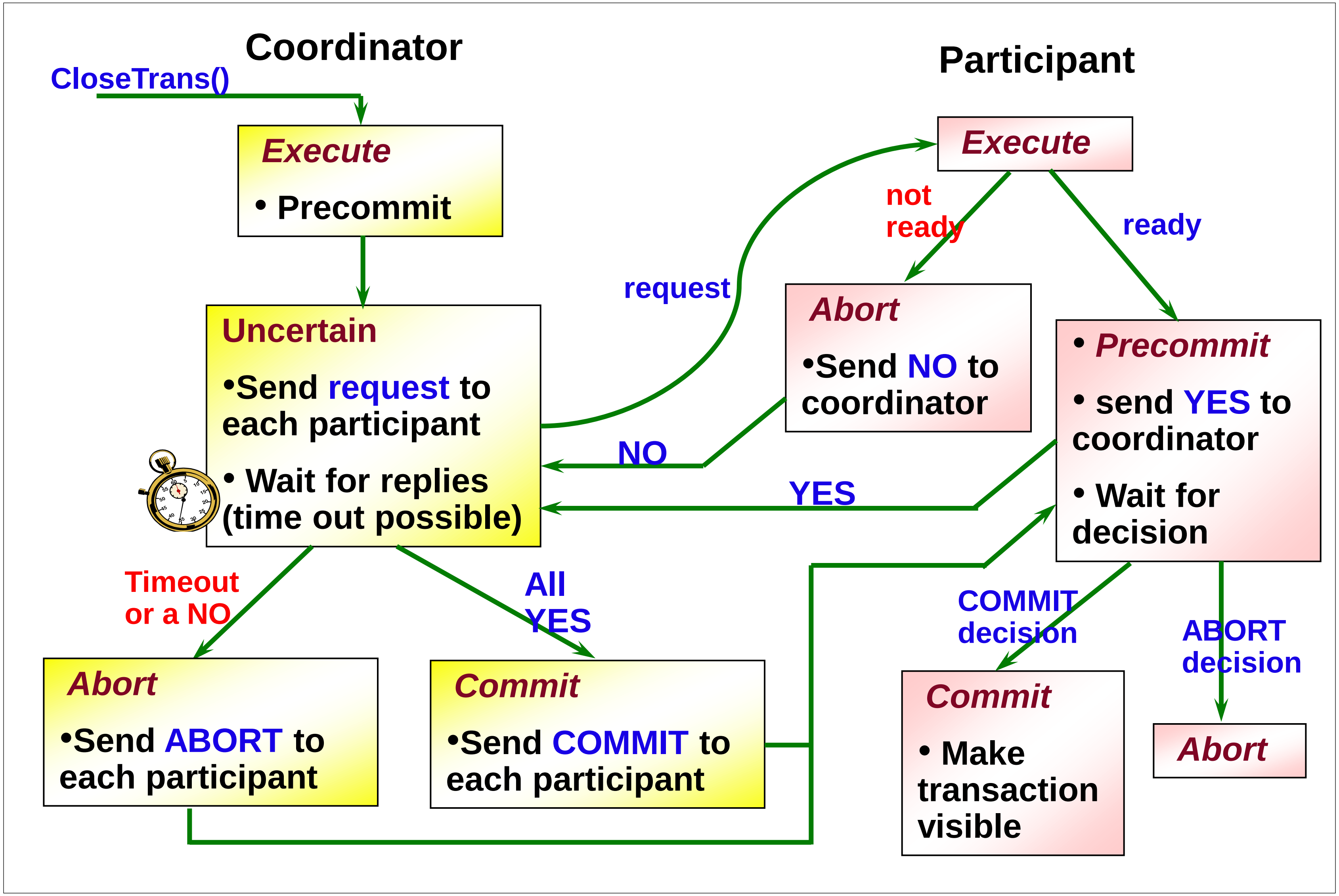
3. The coordinator collects the votes (including its own), makes a decision, and logs this on disk.
   - (a) If there are no failures and all the votes are *Yes,* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   - (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*. This is the step erring on the side of safety.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages, it acts accordingly – when committed, it makes a *haveCommitted* call.
   - If it times out waiting for a doCommit/doAbort, participant keeps sending a getDecision to coordinator, until it knows of the decision

Communication in Two-Phase Commit



- **To deal with participant crashes**
  - **Each participant saves tentative updates into permanent storage, right before replying yes/no in first phase. Retrievable after crash recovery.**
  - **Coordinator logs votes and decisions too**
- **To deal with canCommit? loss**
  - **The participant may decide to abort unilaterally after a timeout for first phase (participant eventually votes No, and so coordinator will also abort)**
- **To deal with Yes/No loss, the coordinator aborts the transaction after a timeout (pessimistic!). It must annouce doAbort to those who sent in their votes.**
- **To deal with doCommit loss**
  - **The participant may wait for a timeout, send a getDecision request (retries until reply received). Cannot abort/commit after having voted Yes but before receiving doCommit/doAbort!**

Two Phase Commit (2PC) Protocol



Issues with 2PC

- If something goes wrong, need to keep retrying the 2PC
- Leader failure and election
- Bad participants may cause frequent aborts

- Um, can’t we just solve consensus?

Yes we can!

- **But really?**
- **Paxos algorithm**
  - **Most popular “consensus-solving” algorithm**
  - **Does not solve consensus problem (which would be impossible, because we already proved that)**
  - **But provides <u>safety</u> and <u>eventual liveness</u>**
  - **A lot of systems use it**
    - **Zookeeper (Yahoo!), Google Chubby, and many other companies**
- **Paxos invented by? (take a guess)**

Yes we can!

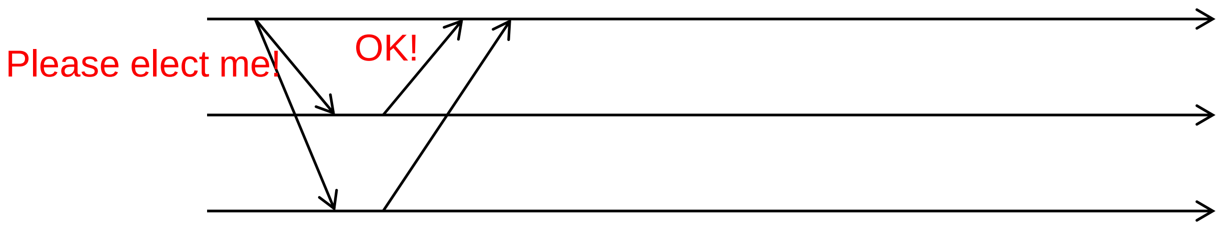
- **Paxos invented by Leslie Lamport**
- **Consensus, in brief**
  - **Processes have different values + need everyone to <u>decide</u> same value + cannot have trivial solutions**
  - **Also, if everyone votes V (Yes or No), then the decision is V**
- **Paxos provides <u>safety</u> and <u>eventual liveness</u>**
  - **Safety: Consensus is not violated**
  - **Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.**

Political Science 101, i.e., Paxos Groked

- Paxos has rounds; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - Use timeouts; may be pessimistic
- Each round broken into phases (also asynchronous)
  - Phase 1: A leader is elected (Election)
  - Phase 2: Leader proposes a value, processes ack (Bill)
  - Phase 3: Leader multicasts final value (Law)

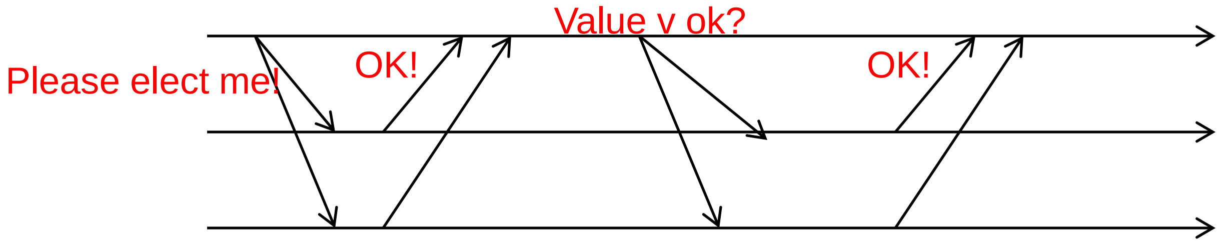
Phase 1 – Election

- Potential leader chooses a unique ballot id, higher than anything so far
- Sends to all processes
- Processes wait, respond once to highest ballot id
  - If potential leader sees a higher ballot id, it can't be a leader
  - Paxos tolerant to multiple leaders, but we'll discuss 1 leader
  - Processes also log received ballot ID on disk
- If a process has in a previous round decided on a value v', it includes value v′ in its response
- If majority respond OK then you are the leader
  - If no one has majority, start new round
- A round cannot have two leaders (why?)



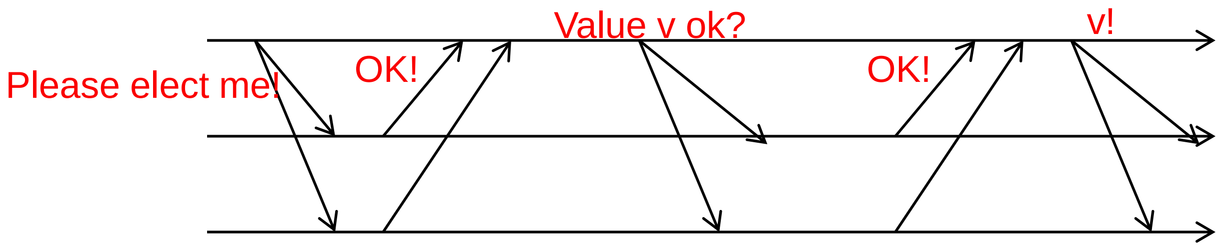
Phase 2 – Proposal (Bill)

- **Leader sends proposed value v to all**
  - **use v′ if some process already decided in a previous round**
- **Recipient logs on disk; responds OK**

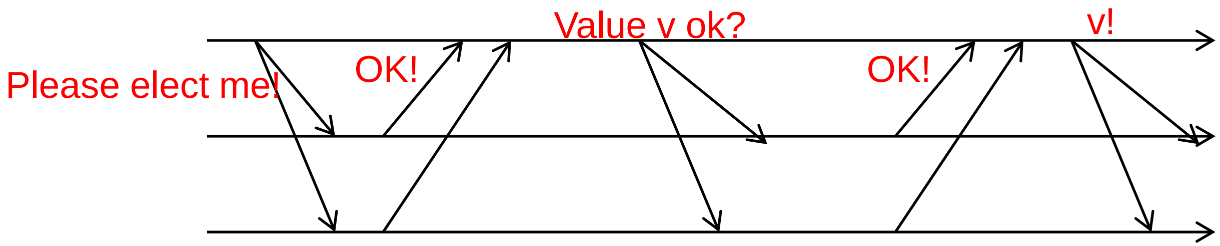


Phase 3 – Decision (Law)

- **If leader hears a majority of OKs, it lets everyone know of the decision**
- **Recipients receive decision, log it on disk**

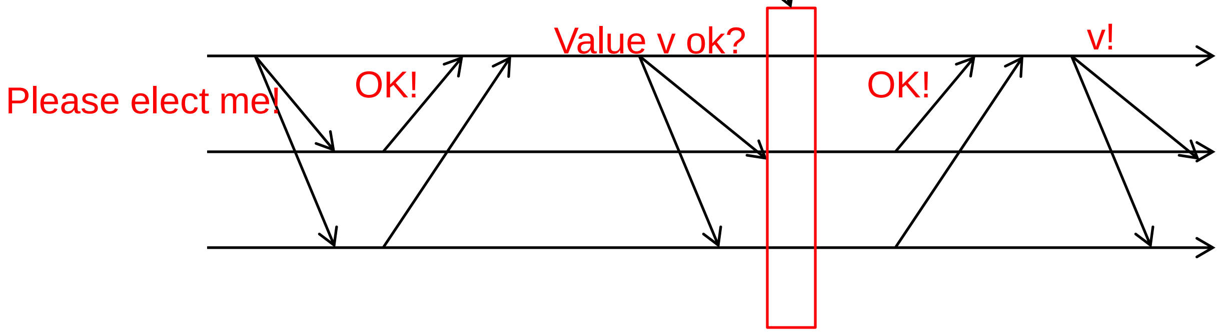


Which is the point of no-return?



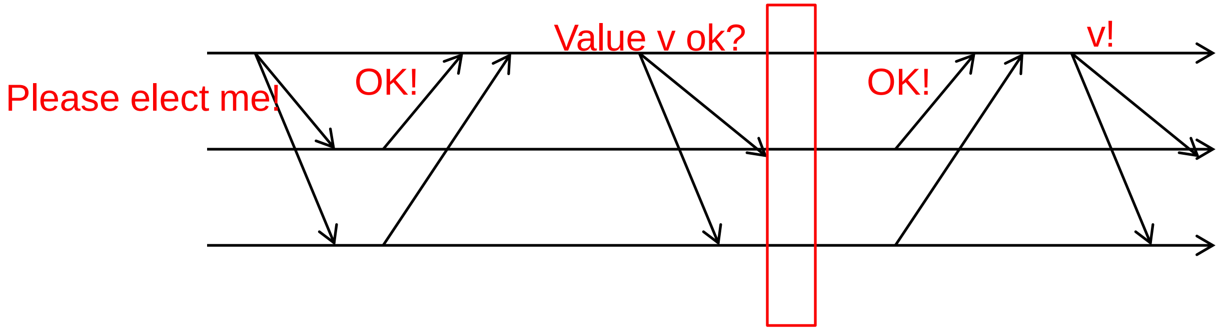
Which is the point of no-return?

- **If a majority of processes hear proposed value and accept it (i.e., are about to/have responded with an OK!)**
- **Processes *may not know it yet*, but a decision has been made for the group**
  - **Even leader does not know it yet**
- **What if leader fails after that?**
  - **Keep having rounds until some round completes**



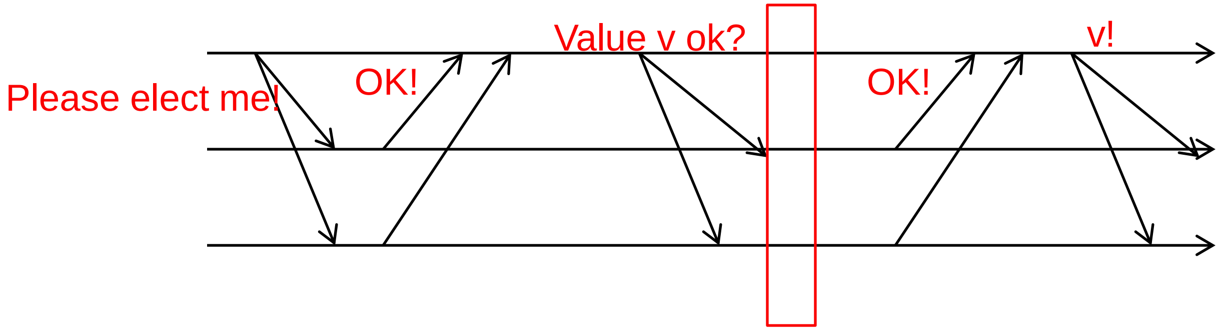
Safety

- If some round has a majority hearing proposed value v′ and accepting it (middle of Phase 2), then each subsequent round either: 1) chooses v′ as decision or 2) round fails
- Proof:
  - Potential leader waits for majority of OKs in Phase 1
  - At least one will contain v’
  - It will choose to send out v’ in Phase 2
- Success requires a majority, and any two majority sets intersect



What could go wrong?

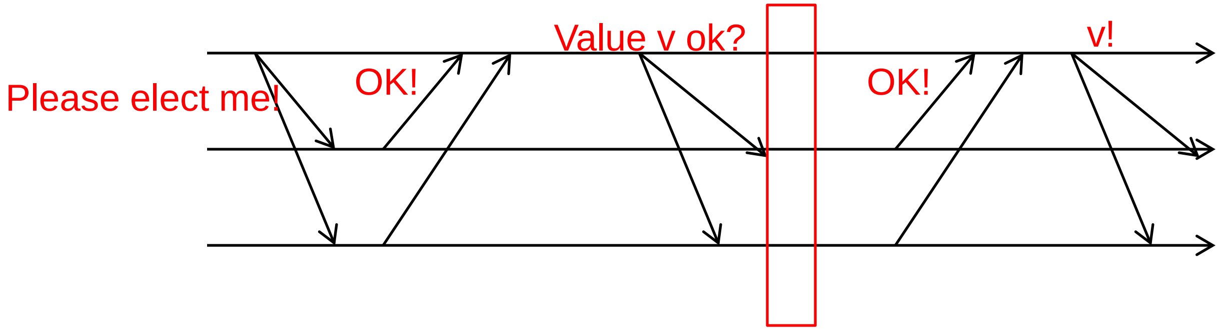
- **Process fails**
  - Majority does not include it
  - When process restarts, it uses disk to retrieve a past decision (if any) and past-seen ballot ids. Tries to know of past decisions.
- **Leader fails**
  - Start another round
- **Messages dropped**
  - If too flaky, just start another round
- **Note that anyone can start a round any time**
- **Protocol may never end – tough luck, buddy!**
  - If things go well sometime in the future, consensus reached



What could go wrong?

- **A lot more!**

- **This is a highly simplified view of Paxos.**
- **See Lamport's original paper: http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf**



Etc.

- **MP3 has been released last week**
  - **You're building a distributed file system, similar to HDFS**
  - **Start NOW**

- **HW3 will be out today**