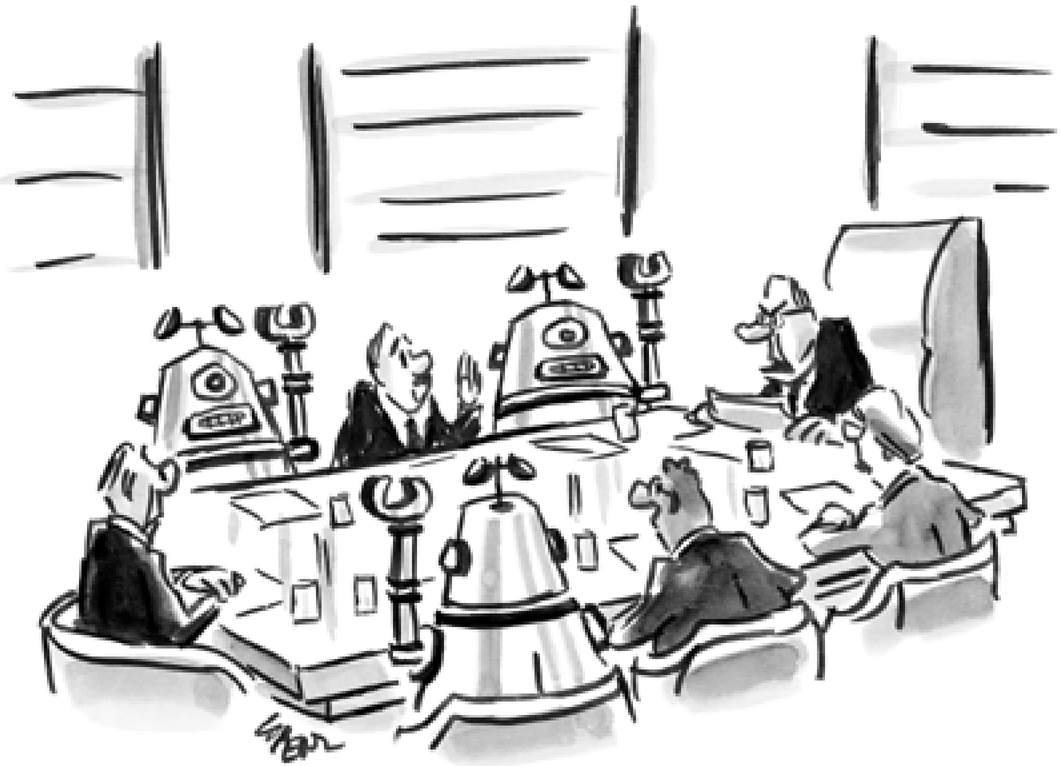
Consensus, impossibility results and Paxos

Ken Birman

Consensus… a classic problem

- Consensus abstraction underlies many distributed systems and protocols
  - N processes
  - They start execution with inputs $\in \{0,1\}$
  - Asynchronous, reliable network
  - At most 1 process fails by halting (crash)
  - Goal: protocol whereby all “decide” same value *v*, and *v* was an input

Distributed Consensus



Jenkins, if I want another yes-man, I'll build one!

Lee Lorenz, Brent Sheppard

Asynchronous networks

- No common clocks or shared notion of time (local ideas of time are fine, but different processes may have very different “clocks”)
- No way to know how long a message will take to get from A to B
- Messages are never lost in the network

Quick comparison...

Asynchronous model	Real world
Reliable message passing, unbounded delays	Just resend until acknowledged; often have a delay model
No partitioning faults (“wait until over”)	May have to operate “during” partitioning
No clocks of any kinds	Clocks but limited sync
Crash failures, can’t detect reliably	Usually detect failures with timeout

Fault-tolerant protocol

- Collect votes from all N processes
  - At most one is faulty, so if one doesn't respond, count that vote as 0
- Compute majority
- Tell everyone the outcome
- They "decide" (they accept outcome)
- ... *but this has a problem! Why?*

What makes consensus hard?

- Fundamentally, the issue revolves around membership
  - In an asynchronous environment, we can't detect failures reliably
  - A faulty process stops sending messages but a "slow" message might confuse us
- Yet when the vote is nearly a tie, this confusing situation really matters

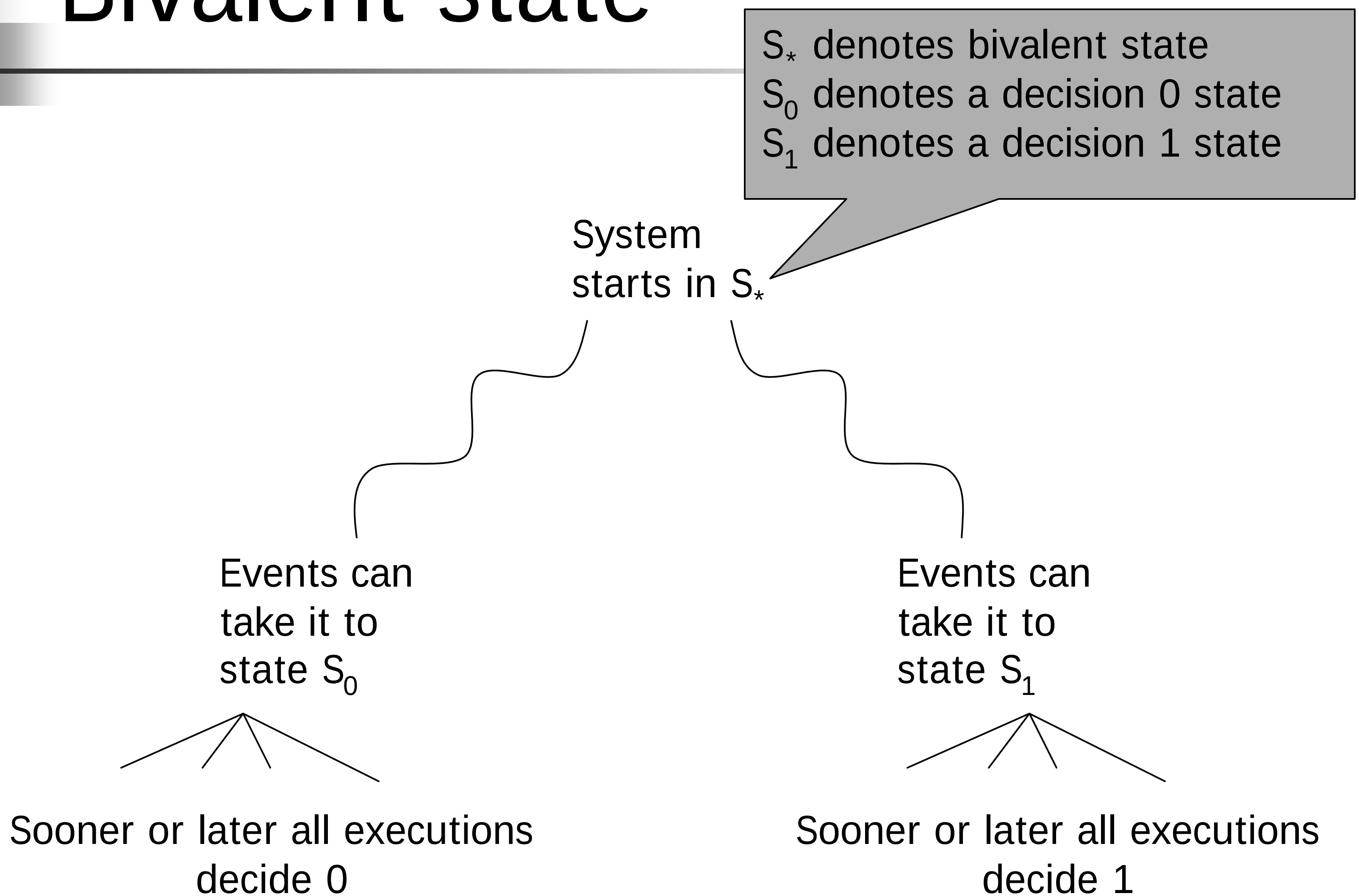
Fischer, Lynch and Patterson

- A surprising result
  - *Impossibility of Asynchronous Distributed Consensus with a Single Faulty Process*
- They prove that no asynchronous algorithm for agreeing on a one-bit value can guarantee that it will terminate in the presence of crash faults
  - And this is true even if no crash actually occurs!
  - Proof constructs infinite non-terminating runs

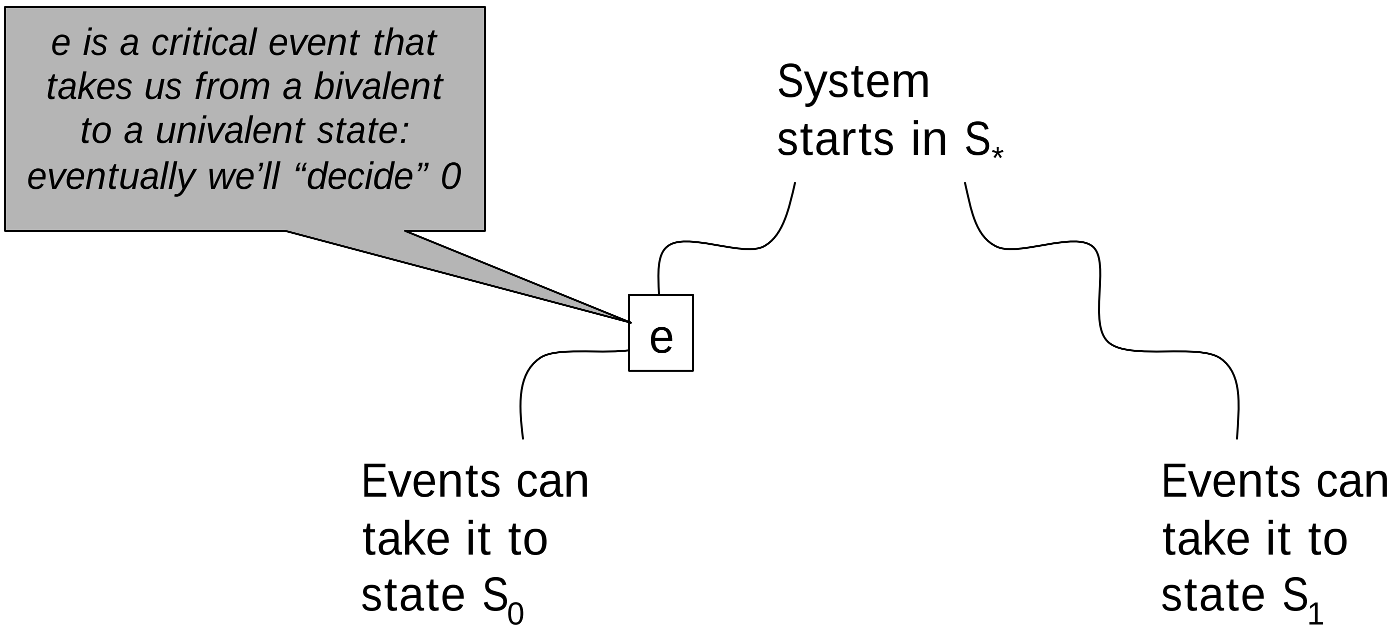
Core of FLP result

- They start by looking at a system with inputs that are all the same
  - All 0’s must decide 0, all 1’s decides 1
- Now they explore mixtures of inputs and find some initial set of inputs with an uncertain (“bivalent”) outcome
- They focus on this bivalent state

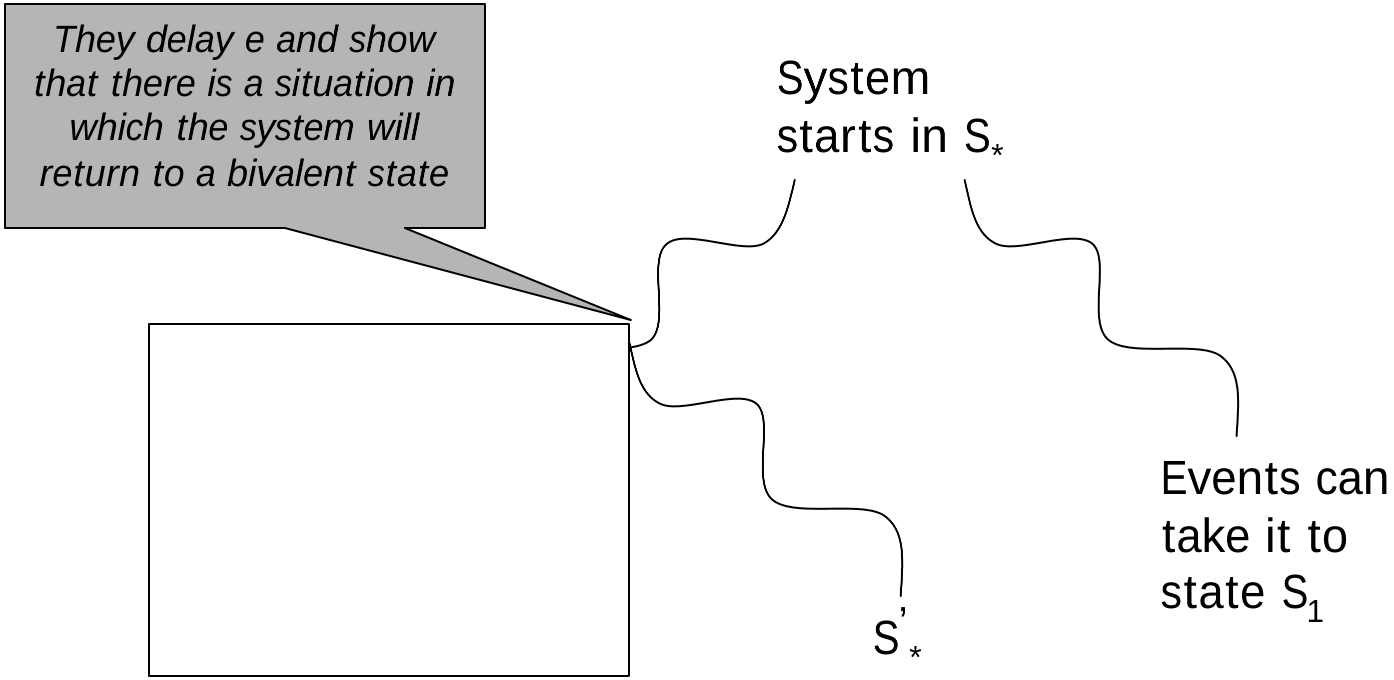
Bivalent state



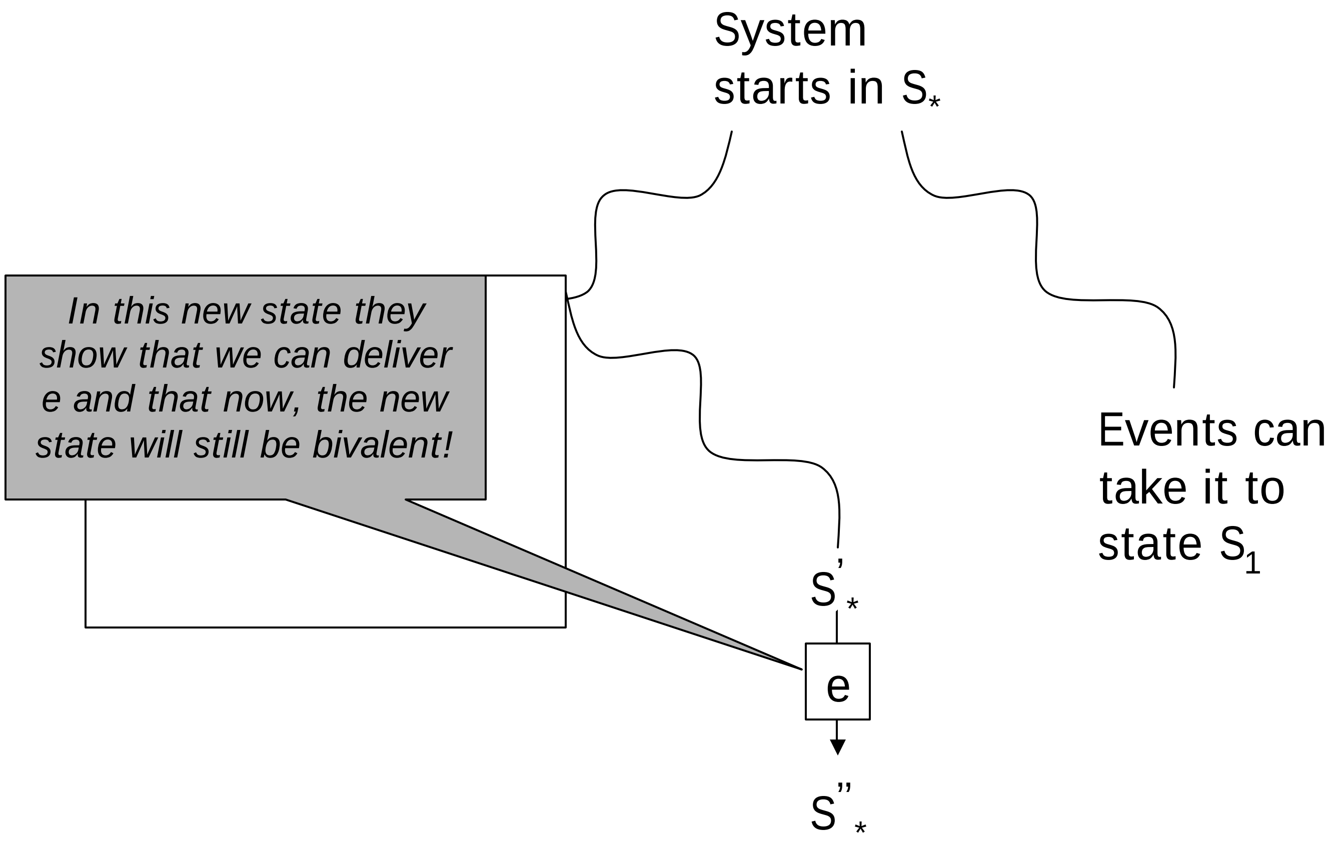
Bivalent state



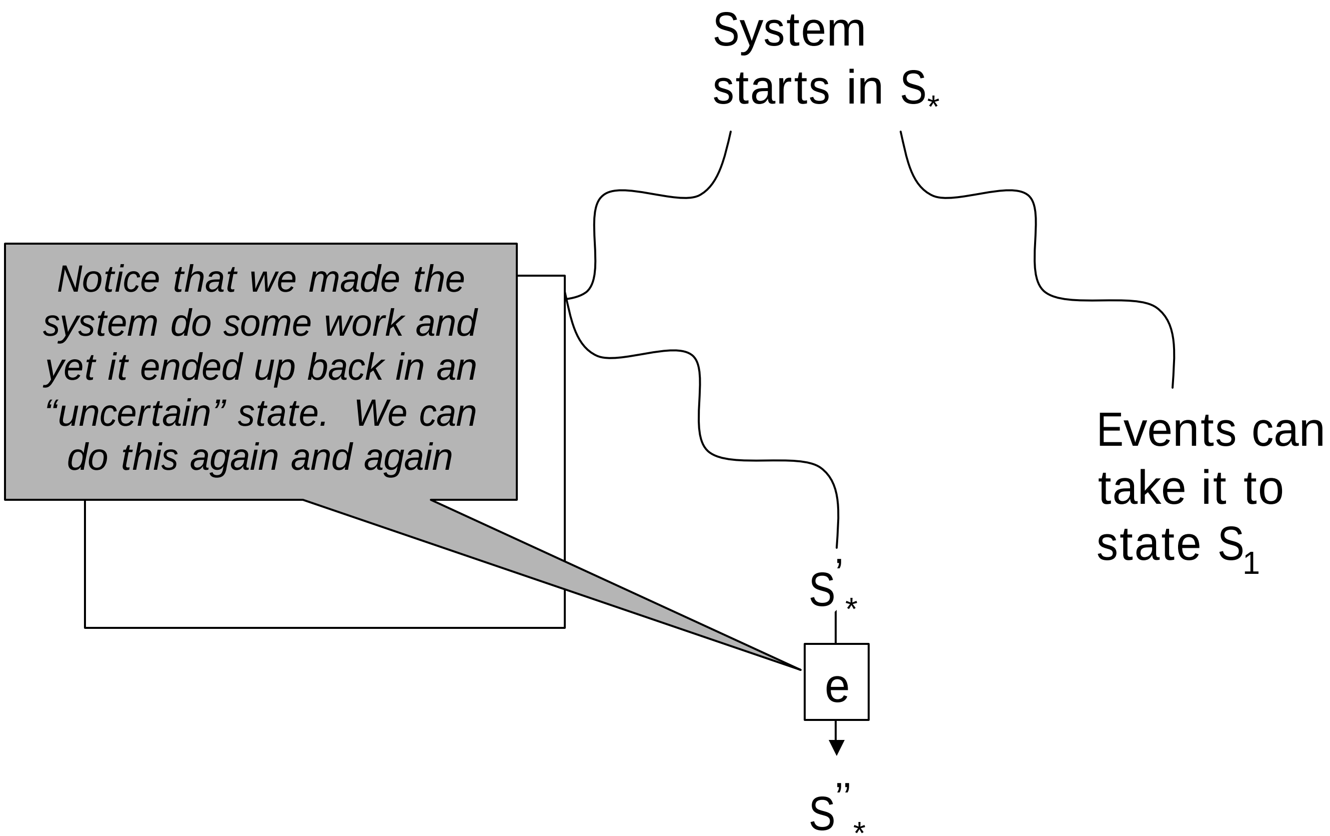
Bivalent state



Bivalent state



Bivalent state



Core of FLP result in words

- In an initially bivalent state, they look at some execution that would lead to a decision state, say "0"
  - At some step this run switches from bivalent to univalent, when some process receives some message *m*
  - They now explore executions in which *m* is delayed

Core of FLP result

- So:
  - Initially in a bivalent state
  - Delivery of *m* would make us univalent but we delay *m*
  - They show that if the protocol is fault-tolerant there must be a run that leads to the <u>other</u> univalent state
  - And they show that you can deliver *m* in this run without a decision being made
- This proves the result: they show that a bivalent system can be forced to do some work and yet remain in a bivalent state.
  - If this is true once, it is true as often as we like
  - In effect: we can delay decisions indefinitely

Intuition behind this result?

- Think of a real system trying to agree on something in which process p plays a key role
- But the system is fault-tolerant: if p crashes it adapts and moves on
- Their proof “tricks” the system into treating p as if it had failed, but then lets p resume execution and “rejoin”
- This takes time... and no real progress occurs

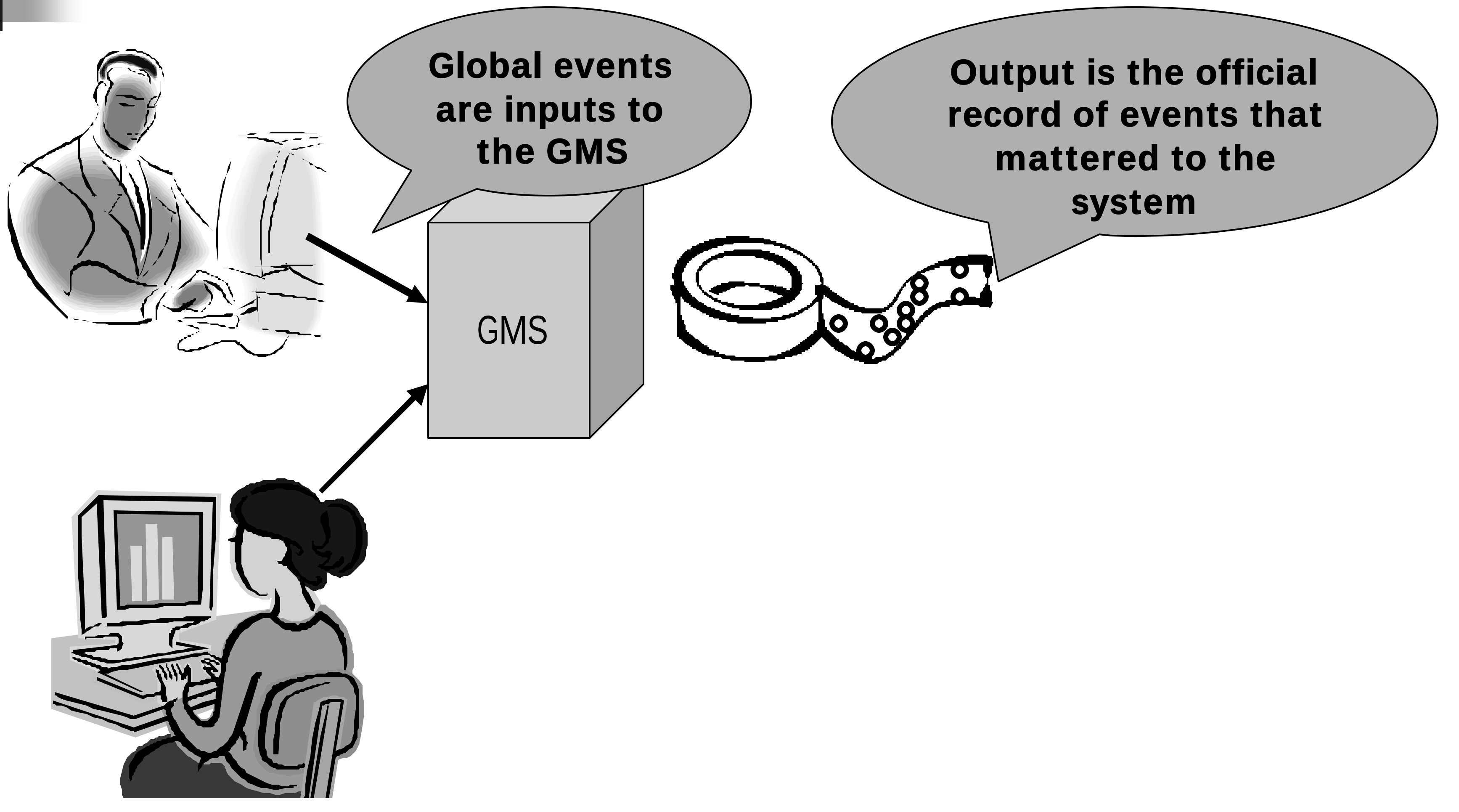
But what did “impossibility” mean?

- In formal proofs, an algorithm is totally correct if
  - It computes the right thing
  - And it *always* terminates
- When we say something is possible, we mean “there is a totally correct algorithm” solving the problem
- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are extremely unlikely (“probability zero”)
  - Yet they imply that we can’t find a totally correct solution
  - And so “consensus is impossible” (“not always possible”)

Solving consensus

- Systems that “solve” consensus often use a membership service
  - This GMS functions as an oracle, a trusted status reporting function
- Then consensus protocol involves a kind of 2-phase protocol that runs over the output of the GMS
- It is known precisely when such a solution will be able to make progress

GMS in a large system



Paxos Algorithm

- Distributed consensus algorithm
  - Doesn’t use a GMS… at least in basic version… but isn’t very efficient either
- Guarantees safety, but not liveness.
- Key Assumptions:
  - Set of processes that run Paxos is known a-priori
  - Processes suffer crash failures
  - All processes have Greek names (but translate as “Fred”, “Cynthia”, “Nancy”…)

Paxos “proposal”

- Node proposes to append some information to a replicated history
- Proposal could be a decision value, hence can solve consensus
- Or could be some other information, such as “Frank’s new salary” or “Position of Air France flight 21”

Paxos Algorithm

- Proposals are associated with a *version number*.
- Processors vote on each proposal. A proposal approved by *a majority* will get passed.
  - Size of majority is “well known” because potential membership of system was known a-priori
  - A process considering two proposals approves the one with the larger version number.

Paxos Algorithm

- 3 roles
  - proposer
  - acceptor
  - Learner

- 2 phases
  - Phase 1: prepare request ←→ Response
  - Phase 2: Accept request ←→ Response

Phase 1: (prepare request)

(1) A proposer chooses a new proposal version number n , and sends a prepare request ("prepare",n) to a majority of acceptors:

- (a) Can I make a proposal with number n ?
- (b) if yes, do you suggest some value for my proposal?

Phase 1: (prepare request)

(2) If an acceptor receives a prepare request ("prepare", n) with n greater than that of any prepare request it has already responded, sends out ("ack", n, n', v') or ("ack", n, $\perp$, $\perp$)

  (a) responds with a promises not to accept any more proposals numbered less than n.

  (b) suggest the value v of the highest-number proposal that it has accepted if any, else $\perp$

Phase 2: (accept request)

(3) If the proposer receives responses from a majority of the acceptors, then it can issue a accept request (“accept”, n , v) with number n and value v:

(a) n is the number that appears in the prepare request.

(b) v is the value of the highest-numbered proposal among the responses

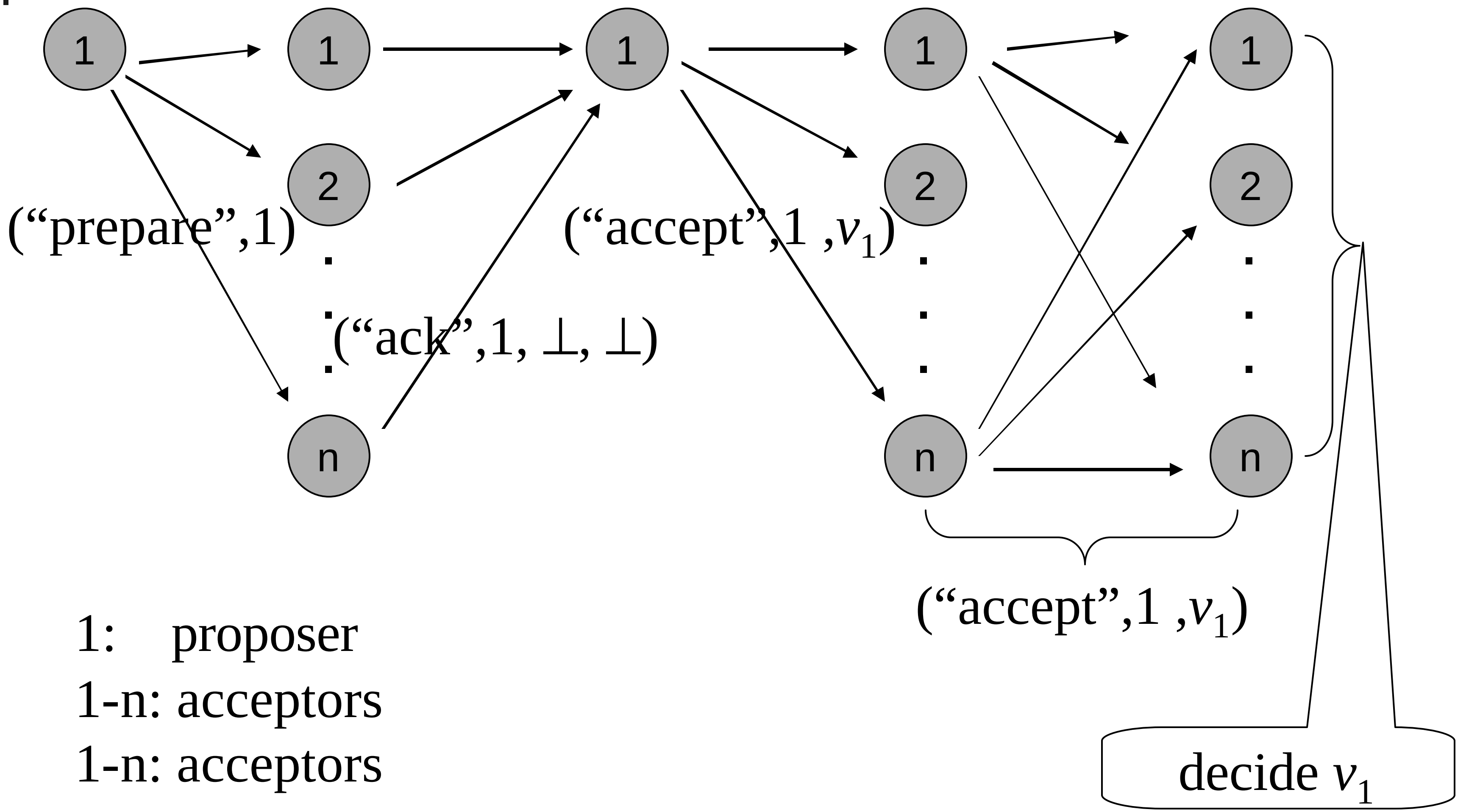
Phase 2: (accept request)

(4) If the acceptor receives an accept request (“accept”, n , v) , it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Learning the decision

- Whenever acceptor accepts a proposal, respond to all learners (“accept”, n, v).
- Learner receives (“accept”, n, v) from a majority of acceptors, decides v, and sends (“decide”, v) to all other learners.
- Learners receive (“decide”, v), decide v

In Well-Behaved Runs



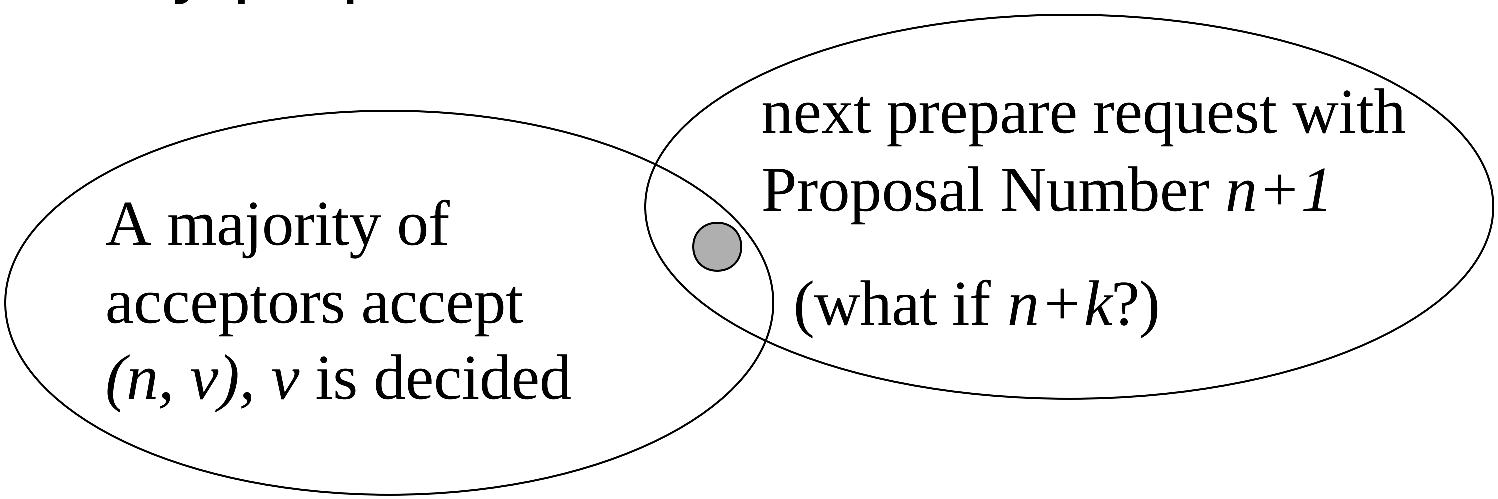
1: proposer

1-n: acceptors

1-n: acceptors

Paxos is safe...

- Intuition:
  - If a proposal with value v is decided, then every higher-numbered proposal issued by any proposer has value v.



Safety (proof)

- Suppose ***(n, v)*** is the earliest proposal that passed. If none, safety holds.
- Let ***(n', v')*** be the earliest issued proposal after ***(n, v)*** with a different value ***v'!=v***
- As ***(n', v')*** passed, it requires a major of acceptors. Thus, some process approve both ***(n, v)*** and ***(n', v')***, though it will suggest value ***v*** with version number ***k>= n***.
- As ***(n', v')*** passed, it must receive a response ***("ack", n', j, v')*** to its prepare request, with ***n<j<n'***. Consider ***(j, v')*** we get the contradiction.

Liveness

- Per FLP, cannot guarantee liveness
- Paper gives us a scenario with 2 proposers, and during the scenario no decision can be made.

Liveness(cont.)

- Omissions cause the Liveness problem.
  - Partitioning failures would look like omissions in Paxos
  - Repeated omissions can delay decisions indefinitely (a scenario like the FLP one)
- But Paxos doesn't block in case of a lost message
  - Phase I can start with new rank even if previous attempts never ended

Liveness(cont.)

- As the paper points out, selecting a distinguished proposer will solve the problem.
  - “Leader election”
  - This is how the view management protocol of virtual synchrony systems works... GMS view management “implements” Paxos with leader election.
  - Protocol becomes a 2-phase commit with a 3-phase commit when leader fails

A small puzzle

- How does Paxos scale?
  - Assume that as we add nodes, each node behaves iid to the other nodes
  - ... hence likelihood of concurrent proposals will rise as $O(n)$
- Core Paxos: 3 linear phases... but expected number of rounds will rise too... get $O(n^2)$... $O(n^3)$ with failures...

Summary

- Consensus is “impossible”
  - But this doesn’t turn out to be a big obstacle
  - We can achieve consensus with probability one in many situations
- Paxos is an example of a consensus protocol, very simple
- We’ll look at other examples Thursday