Feb 27

Impossibility Result and Consensus

Vijay Chidambaram

Consensus

- A set of asynchronous processes have to agree on a value
- In the more general case, each process proposes a value, and one of the values is agreed upon
- Simpler case is when the values are 0 or 1
- Some processes may fail
- Desired properties: termination, agreement, validity
- In FLP proof, we only need some process to decide (not all) - weak termination

Impossibility Result

- No completely asynchronous consensus protocol can tolerate even a single unannounced process death.
- Assumptions:
  - no byzantine failures (only fail-stop)
  - reliable communication channels (exactly once reliably delivery, not FIFO, may be delayed and re-ordered)
  - processes do not have access to a synchronized clock

Modeling the system

- Each process is a finite state automaton
- In one atomic step:
  - each process receives one message
  - does local computation in responses to message
  - sends out finite number of messages (atomic broadcast)

Window of vulnerability

- Period of time in commit or consensus protocol when the processes wait indefinitely for a failed process
- Impossibility result indicates every fully async commit or consensus protocol has this window of vulnerability

Modeling the system (more detail)

- A consensus protocol P is an asynchronous system of N processes (N >= 2)
- Each process p has a one-bit input register x_p, an output register y_p with values in (b, 0, 1), and an unbounded amount of internal storage
- Decision states: when output register has 0 or 1
- Output cannot change once decided

Modeling the system (more detail)

- Message format: (destination, message-value)
- send() and receive() similar to Project 1
- Configuration: internal state of each process + message buffers
- A consensus protocol is partially correct if it satisfies two conditions:
    - (1) No accessible configuration has more than one decision value.
    - (2) For each v E (0, I), some accessible configuration has decision value v.

Correctness Condition

- A consensus protocol P is totally correct in spite of one fault if it is partially correct, and every admissible run is a deciding run.
- Our main theorem shows that every partially correct protocol for the consensus problem has some admissible run that is not a deciding run.

Overview of proof

- The basic idea is to show circumstances under which the protocol remains forever indecisive.
- First, we argue that there is some initial configuration in which the decision is not already predetermined.
- Second, we construct an admissible run that avoids ever taking a step that would commit the system to a particular decision.

Step 1: There is a bivalent initial config

- Consider all initial configs: some lead to 0, some lead to 1
- Order all initial configs such that two configs which differ only in one value are next value
- Somewhere in this initial config, there must be a pair where one config is 0, another config is 1
    - These two configs differ only by the value of one process (lets call that p)
- We allow one process to fail, so 0-config should decide 0 even if p fails
    - But if it does not depend on p, 1-config (which only differs by p) should also decide 0.
    - Thus, 1-config is actually bivalent: it can decide 0 or 1

Step 2

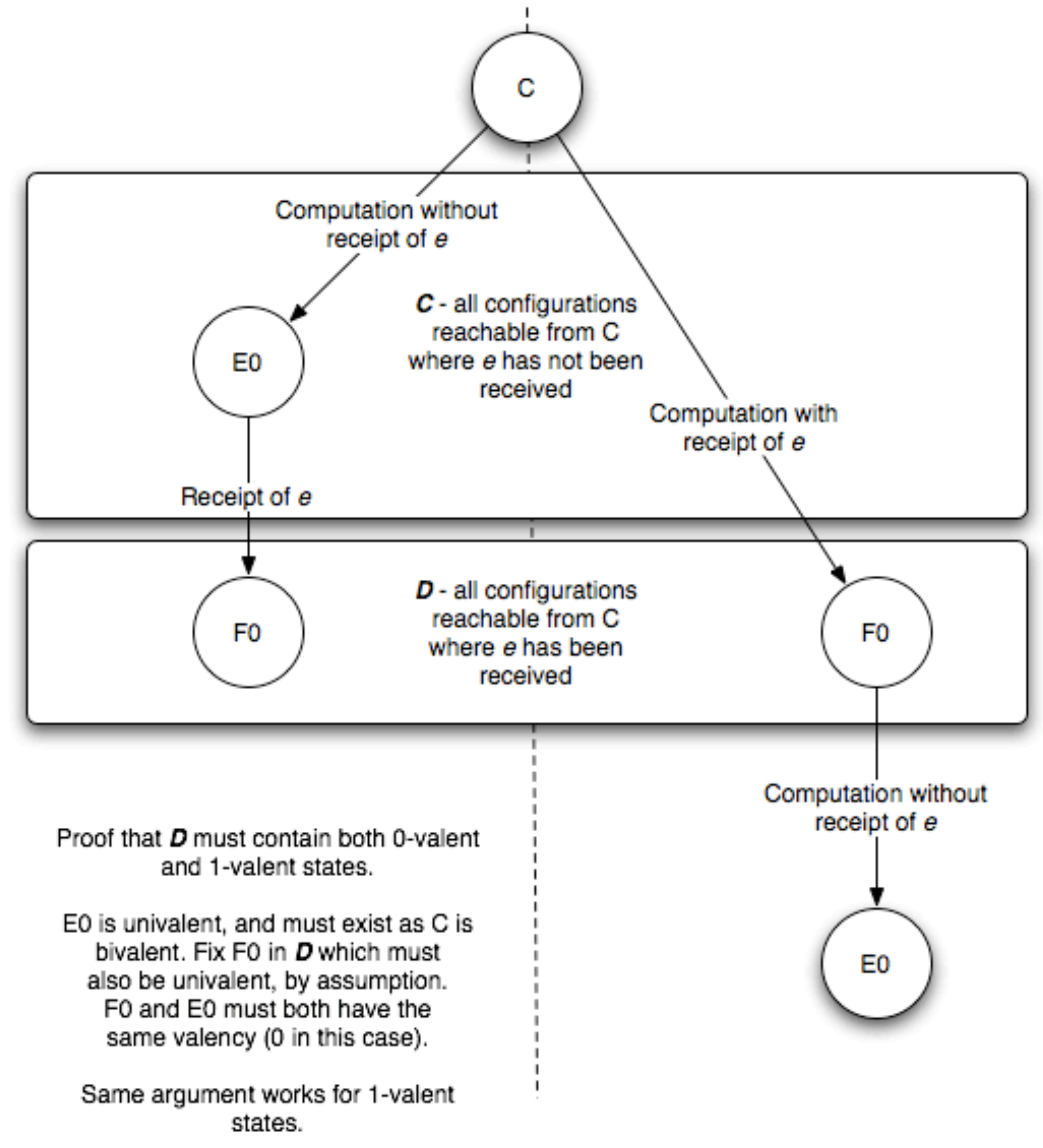
- If you start from a bivalent config and apply message e, the resulting set of configs contains a bivalent config
- In other words, if you delay a message e long enough, you can move from a bivalent config to another bivalent config

Proving Step 2

- See https://www.the-paper-trail.org/post/2008-08-13-a-brief-tour-of-flp-impossibility/
- Good explanation of the proof
- Much more easier to understand than the paper

Terms

- C: starting bivalent config
- R: configs where e has not been received
- D: configs where e has been received


C
Computation without receipt of *e*
E0
C - all configurations reachable from C where *e* has not been received
Computation with receipt of *e*
Receipt of *e*
F0
D - all configurations reachable from C where *e* has been received
F0
Computation without receipt of *e*
E0

Proof that ***D*** must contain both 0-valent and 1-valent states.

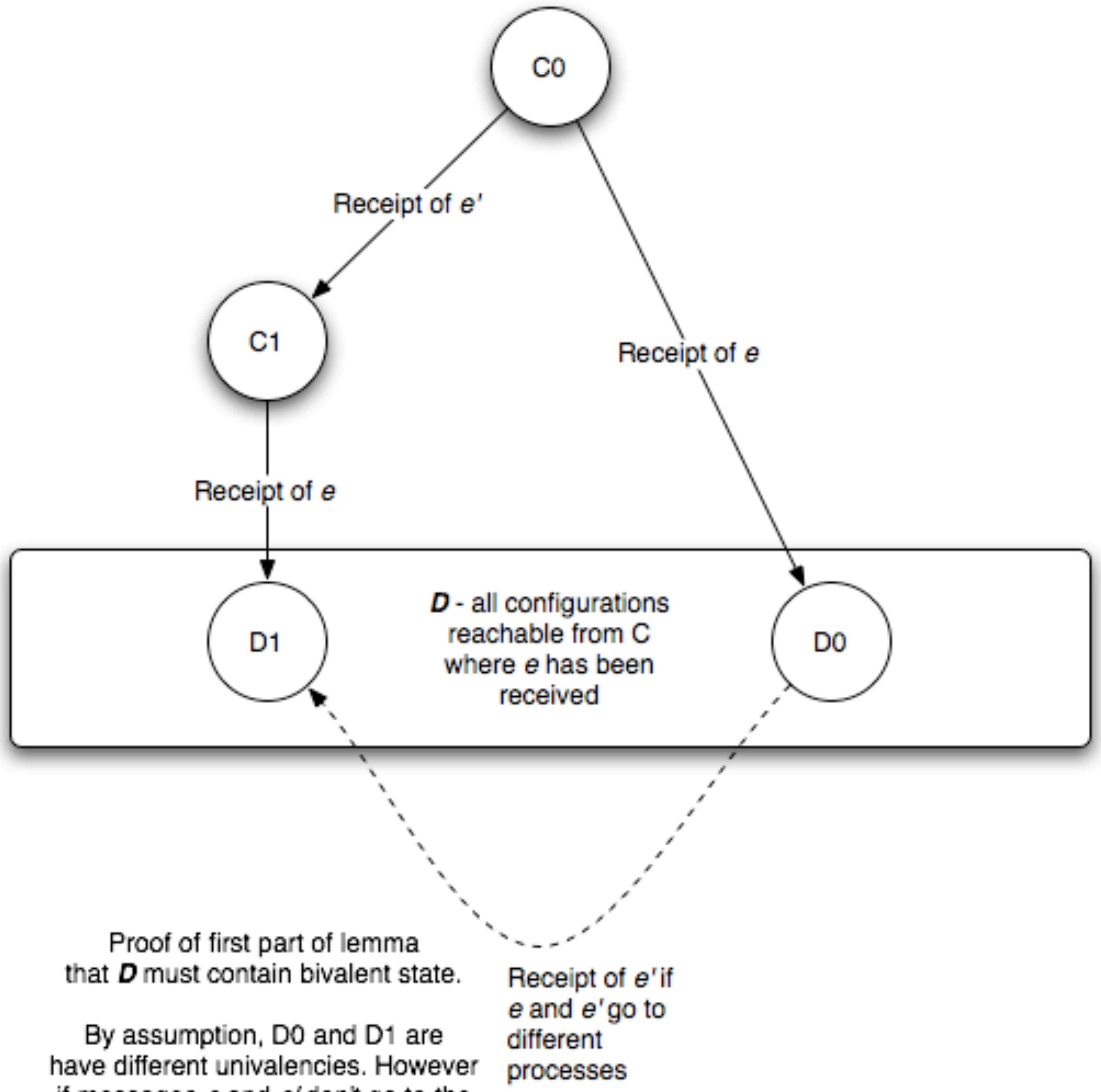
E0 is univalent, and must exist as C is bivalent. Fix F0 in ***D*** which must also be univalent, by assumption. F0 and E0 must both have the same valency (0 in this case).

Same argument works for 1-valent states.

Figure 2.1

Established so far..

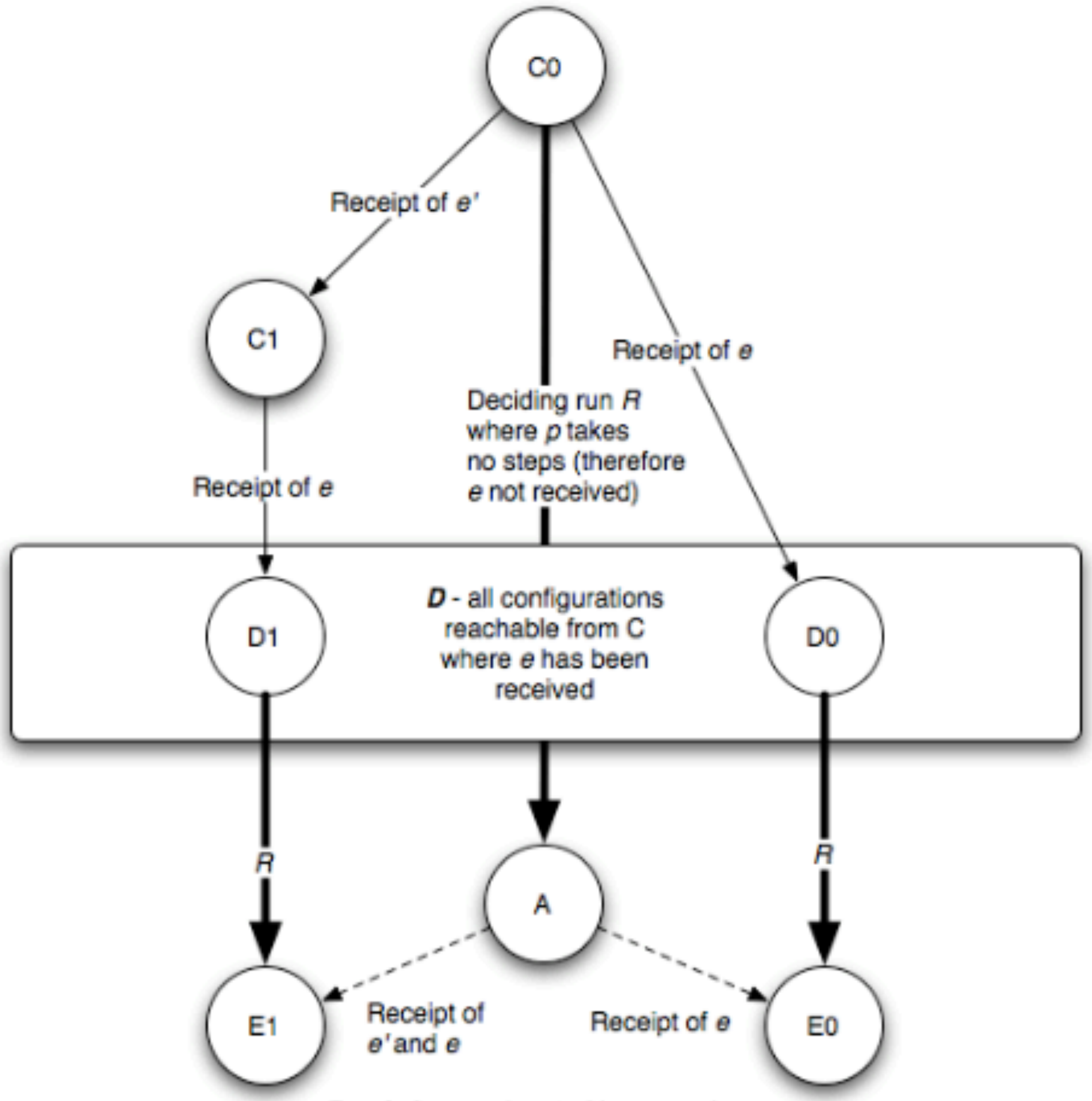
- That D must have both 0-configs and 1-configs
- Assumption: D does not have bivalent configs


C0
Receipt of *e′*
C1
Receipt of *e*
Receipt of *e*
D1
D - all configurations reachable from C where *e* has been received
D0
Receipt of *e′* if *e* and *e′* go to different processes

Proof of first part of lemma that ***D*** must contain bivalent state.

By assumption, D0 and D1 are have different univalencies. However if messages *e* and *e′* don't go to the same process they may be applied in any ordering and all processes must end in same state (as they have seen the same order of messages locally).

This is a contradiction as now D0 is bivalent.


C0
Receipt of *e'*
C1
Receipt of *e*
Deciding run *R* where *p* takes no steps (therefore *e* not received)
Receipt of *e*
D1
D - all configurations reachable from C where *e* has been received
D0
R
A
R
E1
Receipt of *e'* and *e*
Receipt of *e*
E0

Proof of second part of lemma, where *e'* and *e* are addressed to the same process *p*: E0 and E1 are 0- and 1-valent respectively.

A is a univalent state reached by a deciding run from C0 where the process *p* for which *e'* and *e* are intended takes no steps (as if it had failed).

But at A either *e'* then *e* or just *e* can be received by *p* which places it in one of two univalent states. But A is itself univalent, and so this is a contradiction.

Constructing admissible runs which never decide

- We start from a bivalent configuration
- Let e be the message to be processed by process p next
- There is a bivalent configuration C' reachable from this config where e is the last message applied
- So we move from bivalent configs to other bivalent configs, and no decision is ever reached

Paxos

- P1. An acceptor must accept the first proposal that it receives.
  - An acceptor can accept multiple values
- P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v.
- P2a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v.
- P2b. If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

Paxos

- P2c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either (a) no acceptor in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.
- P1a . An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having a number greater than n.

The Paxos Algorithm

- Phase 1. (a) A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.
- (b) If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

The Paxos Algorithm

- Phase 2. (a) If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.
- (b) If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n.