M120: DISTRIBUTED SYSTEMS

Consensus & Paxos

*Slides are variant of slides provided by Indranil (Indy) Gupta

Give it a thought

Have you ever wondered why distributed server vendors always only offer solutions that promise five-9's reliability, seven-9's reliability, but never 100% reliable?

The fault does not lie with the companies themselves.

The fault lies in the impossibility of consensus

What is common to all of these?

A group of servers attempting:

- Make sure that all of them receive the same updates in the same order as each other
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously
- Elect a leader among them, and let everyone in the group know about it
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file

What is common to all of these?

A group of servers attempting:

- Make sure that all of them receive the same updates in the same order as each other [Reliable Multicast]
- To keep their own local lists where they know about each other, and when anyone leaves or fails, everyone is updated simultaneously [Membership/Failure Detection]
- Elect a leader among them, and let everyone in the group know about it [Leader Election]
- To ensure mutually exclusive (one process at a time only) access to a critical resource like a file [Mutual Exclusion]

So what is common?

- Let's call each server a "process" (think of the daemon at each server)
- All of these were groups of processes attempting to *coordinate* with each other and reach *agreement* on the value of something
  - The ordering of messages
  - The up/down status of a suspected failed process
  - Who the leader is
  - Who has access to the critical resource
- All of these are related to the *Consensus* problem

What is Consensus?

Formal problem statement

- N processes
- Each process p has

  input variable xp : initially either 0 or 1

  output variable yp : initially b (can be changed only once)
- Consensus problem: design a protocol so that at the end, either:
  1. All processes set their output variables to 0 (all-0's)
  2. Or All processes set their output variables to 1 (all-1's)

What is Consensus? (2)

- Every process contributes a value
- *Goal is to have all processes decide same (some) value*
  - Decision once made can't be changed
- There might be other constraints
  - Validity = if everyone proposes same value, then that's what's decided
  - Integrity = decided value must have been proposed by some process
  - Non-triviality = there is at least one initial system state that leads to each of the all-0's or all-1's outcomes

Why is it Important?

- Many problems in distributed systems are equivalent to *(or harder than)* consensus!
  - Perfect Failure Detection
  - Leader election (select exactly one leader, and every alive process knows about it)
  - Agreement (harder than consensus)

- So consensus is a very important problem, and solving it would be really useful!
- So, is there a solution to Consensus?

Two Different Models of Distributed Systems

- Synchronous System Model and Asynchronous System Model
- Synchronous Distributed System
  - Each message is received within bounded time
  - Drift of each process' local clock has a known bound
  - Each step in a process takes lb < time < ub

  E.g., A collection of processors connected by a communication bus, e.g., a Cray supercomputer or a multicore machine

Asynchronous System Model

- Asynchronous Distributed System
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays

  E.g., The Internet is an asynchronous distributed system, so are ad-hoc and sensor networks
- *This is a more general (and thus challenging) model than the synchronous system model. A protocol for an asynchronous system will also work for a synchronous system (but not vice-versa)*

Possible or Not

- In the synchronous system model
  - Consensus is solvable
- In the asynchronous system model
  - Consensus is impossible to solve
  - Whatever protocol/algorithm you suggest, there is always a worst-case possible execution (with failures and message delays) that prevents the system from reaching consensus
  - Powerful result (see the FLP proof)
  - Subsequently, safe or probabilistic solutions have become quite popular to consensus or related problems.

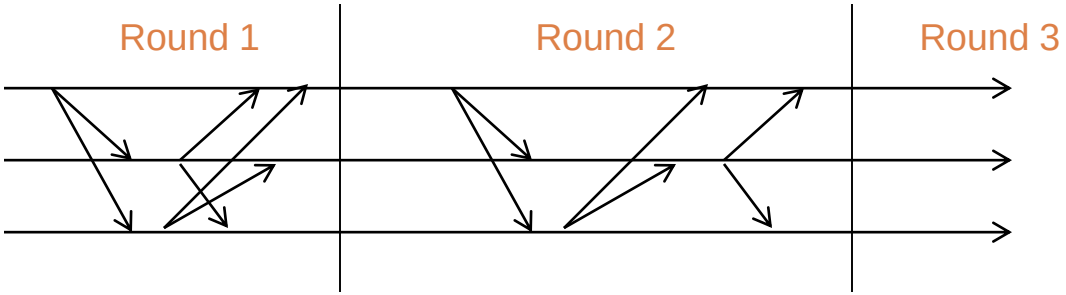
Let's Try to Solve Consensus!

- First, what's the **system model**? (assumptions!)
- Synchronous system: bounds on
  - Message delays
  - Upper bound on clock drift rates
  - Max time for each process step

  e.g., multiprocessor (common clock across processors)
- Processes can fail by stopping (crash-stop or crash failures)

Consensus in Synchronous Systems

- For a system with at most *f* processes crashing
  - All processes are synchronized and operate in "rounds" of time
  - the algorithm proceeds in *f+1* rounds (with timeout), using reliable communication to all members
  - $Values^r_i$: the set of proposed values known to p_i at the beginning of round *r*.



Consensus in Synchronous System

Possible to achieve!

- For a system with at most *f* processes crashing
  - All processes are synchronized and operate in "rounds" of time
  - the algorithm proceeds in *f +1* rounds (with timeout), using reliable communication to all members
  - $Values^r_i$: the set of proposed values known to p_i at the beginning of round *r*.

- Initially $Values^0_i = \{\}$; $Values^1_i = \{v_i\}$

  for round = 1 to *f+1* do

    multicast ($Values^r_i - Values^{r-1}_i$) // iterate through processes, send each a message

    $Values^{r+1}_i \leftarrow Values^r_i$

    for each V_j received

      $Values^{r+1}_i = Values^{r+1}_i \cup V_j$

    end

  end

  d_i = minimum($Values^{f+2}_i$)

Why does the Algorithm work?

- After *f+1* rounds, all non-faulty processes would have received the same set of Values Proof by contradiction.
- Assume that two non-faulty processes, say p_i and p_j , differ in their final set of values (i.e., after *f+1* rounds)
- Assume that p_i possesses a value *v* that p_j does not possess.
  - p_i must have received *v* in the very last round
    - Else, p_i would have sent *v* to p_j in that last round
  - So, in the last round: a third process, p_k, must have sent *v* to p_i, but then crashed before sending *v* to p_j.
  - Similarly, a fourth process sending *v* in the second-to-last round must have crashed; otherwise, both p_k and p_j should have received *v*.
  - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
  - This means a total of *f+1* crashes, while we have assumed at most *f* crashes can occur => contradiction.

Next

- Let’s be braver and solve Consensus in the Asynchronous System Model

Consensus Problem

- Consensus impossible to solve in asynchronous systems (FLP Proof)
  - Key to the Proof: It is impossible to distinguish a failed process from one that is just very very (very) slow. Hence the rest of the alive processes may stay ambivalent (forever) when it comes to deciding.
- But Consensus important since it maps to many important distributed computing problems
- So, can't we just solve consensus?

Yes we Can!

- Paxos algorithm
  - Most popular “consensus-solving” algorithm
  - Does not solve consensus problem (which would be impossible, because FLP already proved that)
  - But provides safety and eventual liveness
  - A lot of systems use it
    - Zookeeper (Yahoo!), Google Chubby, and many other companies

Yes we Can!

- Paxos invented by Leslie Lamport

- Paxos provides safety and eventual liveness
  - 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.

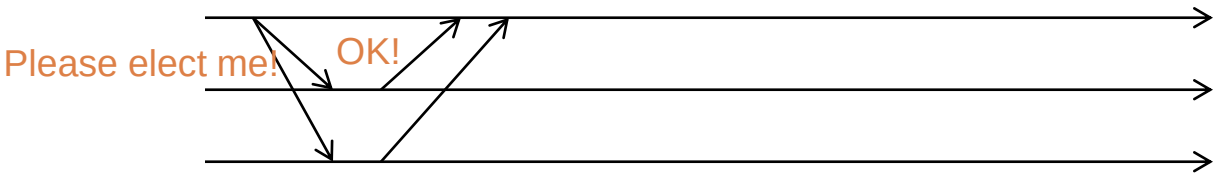
- FLP result still applies: Paxos is not *guaranteed* to reach Consensus (ever, or within any bounded time)

Political Science 101 (Paxos Grokked)

- Paxos has rounds; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - If you're in round *j* and hear a message from round *j+1*, abort everything and move over to round *j+1*
  - Use timeouts; may be pessimistic
- Each round itself broken into phases (which are 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)

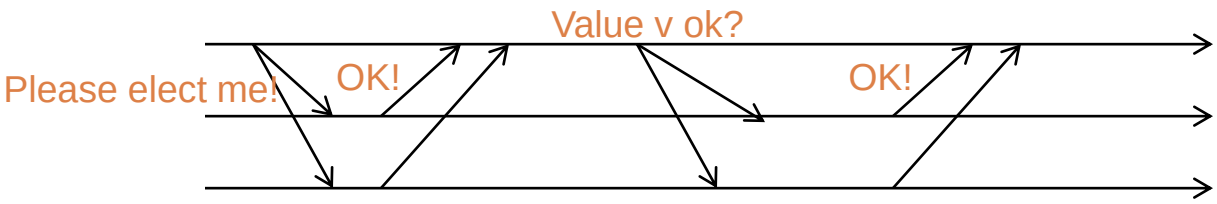
Phase 1 – Election

- Potential leader chooses a unique ballot id, higher than anything seen 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 only discuss 1 leader case
  - 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 (i.e., quorum) respond OK then you are the leader
  - If no one has majority, start new round
- (If things go right) A round cannot have two leaders (why?)



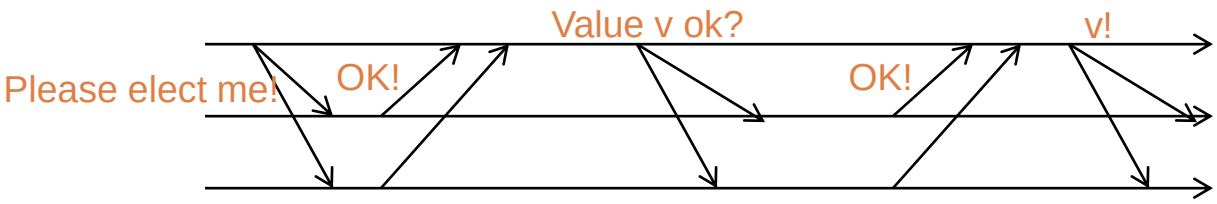
Phase 2 – Proposal (Bill)

- Leader sends proposed value v to all
  - use v=v’ if some process already decided in a previous round and sent you its decided value v’
- Recipient logs on disk; responds OK



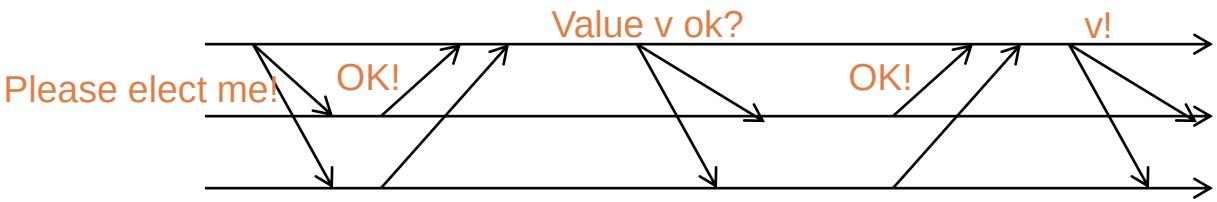
Phase 3 – Decision (Law)

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



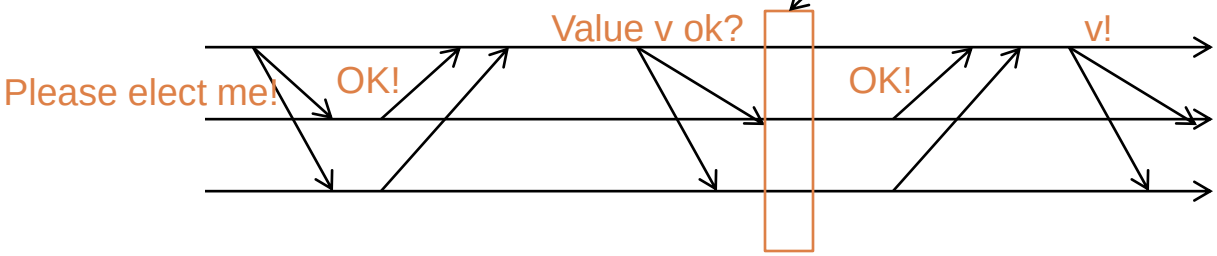
Which is the point of No-Return?

- That is, when is consensus reached in the system



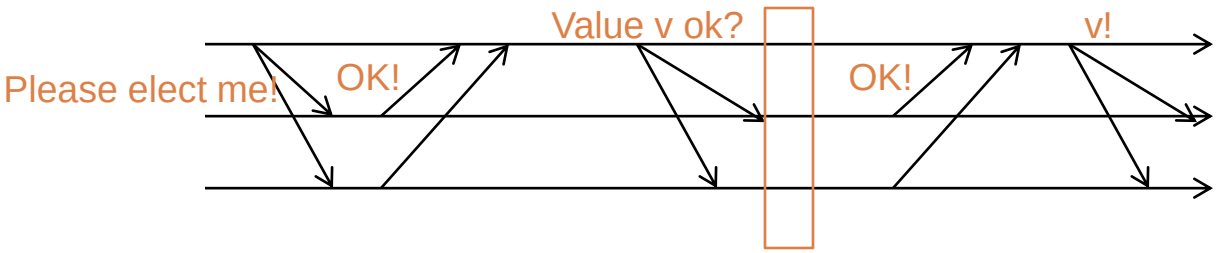
Which is the point of No-Return?

- If/when a majority of processes hear proposed value and accept it (i.e., are about to/have respond(ed) 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



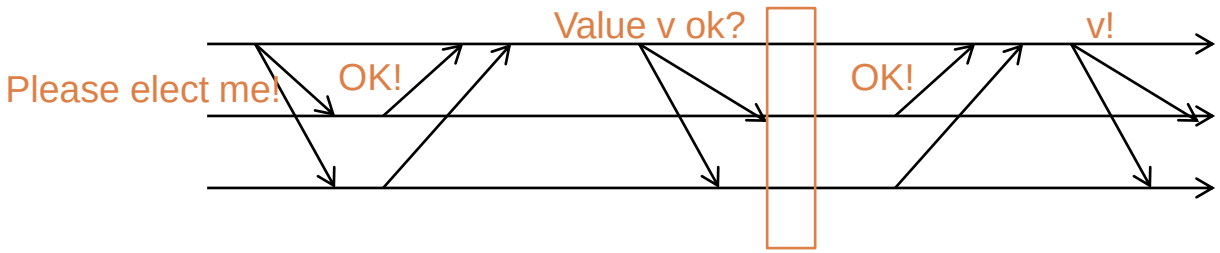
Safety

- Paxos guarantees that two different values are not decided by two different processes
- If some round has a majority (i.e., quorum) hearing proposed value v’ and accepting it (middle of Phase 2), then subsequently at each round either: 1) the round chooses v’ as decision or 2) the round fails



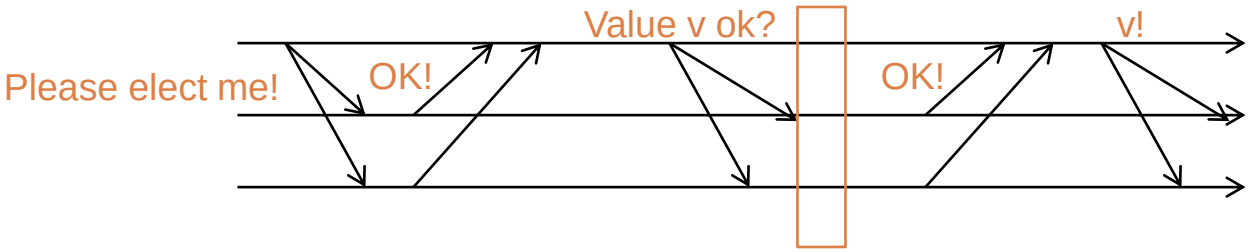
Safety

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



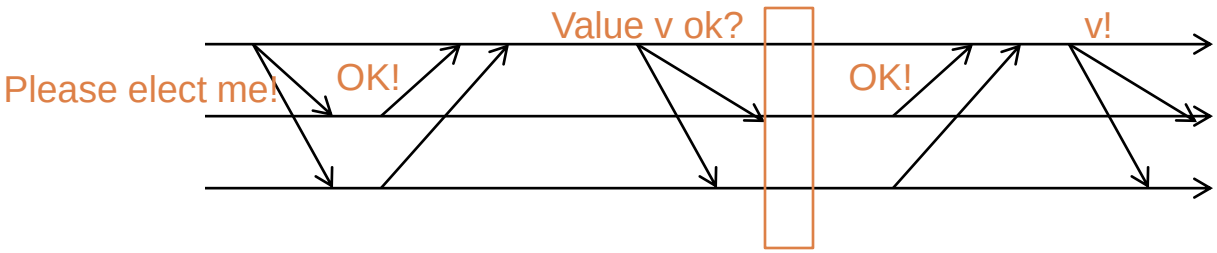
What could go Wrong?

- Process fails
  - Majority does not include it
  - When process restarts, it uses log 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!
  - Impossibility result not violated
  - 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: "A Part-time Parliament"
- The algorithm is for a single instance of consensus



Paxos optimizations

- Using a separate leader-election scheme we can reduce the risk of having two competing leaders that interfere with each other (if that happens, they can repeatedly abort)
- We can batch requests and do several a time
- We can combine several proposals and run them all at the same time, for distinct slots
- Lamport extended Paxos to support changing membership
- The trick is that we build this as incremental steps so the “correctness” of the core protocol is unchanged

Paxos summary

- An important and widely studied/used protocol (perhaps the most important agreement protocol)
- Developed by Lamport but the protocol per-se wasn't really the innovation
  - Similar protocols were widely used prior to Paxos
- The key advance was the proof methodology
  - We touched on one corner of it
  - Lamport addresses the full set of features in his papers

Leslie Lamport's Reflections

- **"Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island.**

- **"To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist.**

- **"My attempt at inserting some humor into the subject was a dismal failure.**

The History of the Paper by Lamport

- **"I submitted the paper to *TOCS* in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. I was quite annoyed at how humorless everyone working in the field seemed to be, so I did nothing with the paper."**
- **"A number of years later, a couple of people at SRC needed algorithms for distributed systems they were building, and Paxos provided just what they needed. I gave them the paper to read and they had no problem with it. So, I thought that maybe the time had come to try publishing it again."**

- *Along the way, Leslie kept extending Paxos and proving the extensions correct. And this is what made Paxos important: the process of getting there while preserving correctness!*

Summary

- Consensus is a very important problem
  - Equivalent to many important distributed computing problems that have to do with *reliability*
- Consensus is possible to solve in a synchronous system where message delays and processing delays are bounded
- Consensus is impossible to solve in an asynchronous system where these delays are unbounded
- Paxos protocol: widely used implementation of a safe, eventually-live consensus protocol for asynchronous systems
  - Paxos (or variants) used in Apache Zookeeper, Google's Chubby system, Active Disk Paxos, and many other cloud computing systems

Consensus in an Asynchronous System

- Impossible to achieve!
- Proved in a now-famous result by Fischer, Lynch and Patterson, 1983 (FLP)
  - Stopped many distributed system designers dead in their tracks
  - A lot of claims of "reliability" vanished overnight

Recall

Asynchronous system: All message delays and processing delays can be arbitrarily long or short.

Consensus:

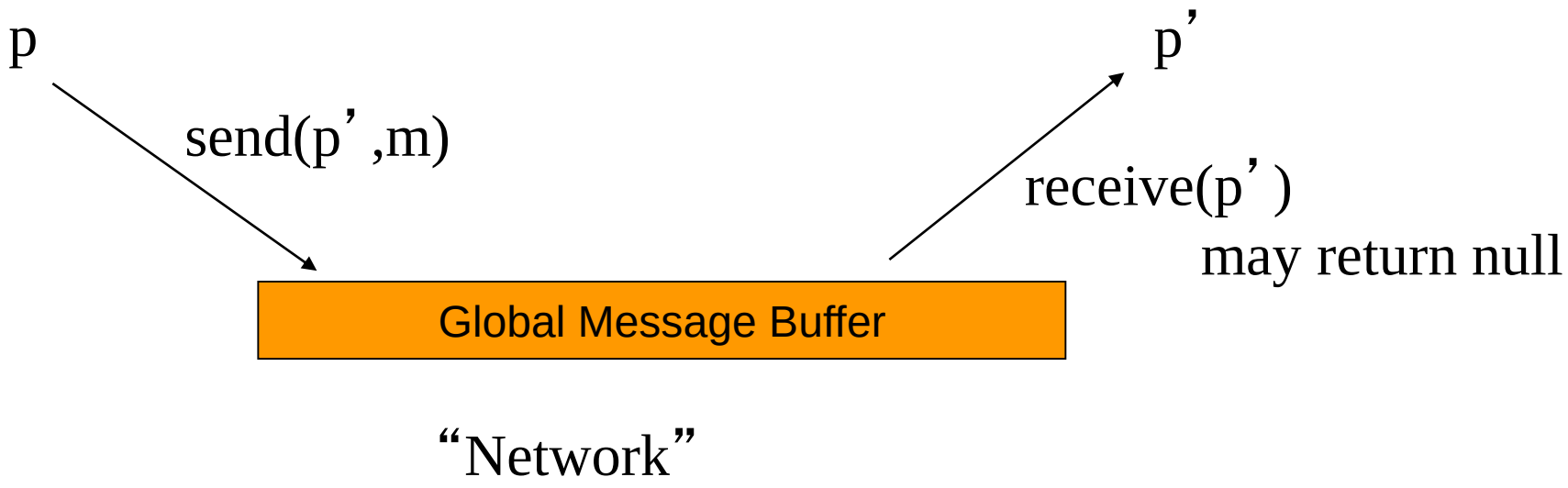
- Each process p has a state
  - program counter, registers, stack, local variables
  - input register xp : initially either 0 or 1
  - output register yp : initially b (undecided)
- Consensus Problem: design a protocol so that either
  - all processes set their output variables to 0 (all-0's)
  - Or all processes set their output variables to 1 (all-1's)
  - Non-triviality: at least one initial system state leads to each of the above two outcomes

Proof Setup

- For impossibility proof, OK to consider

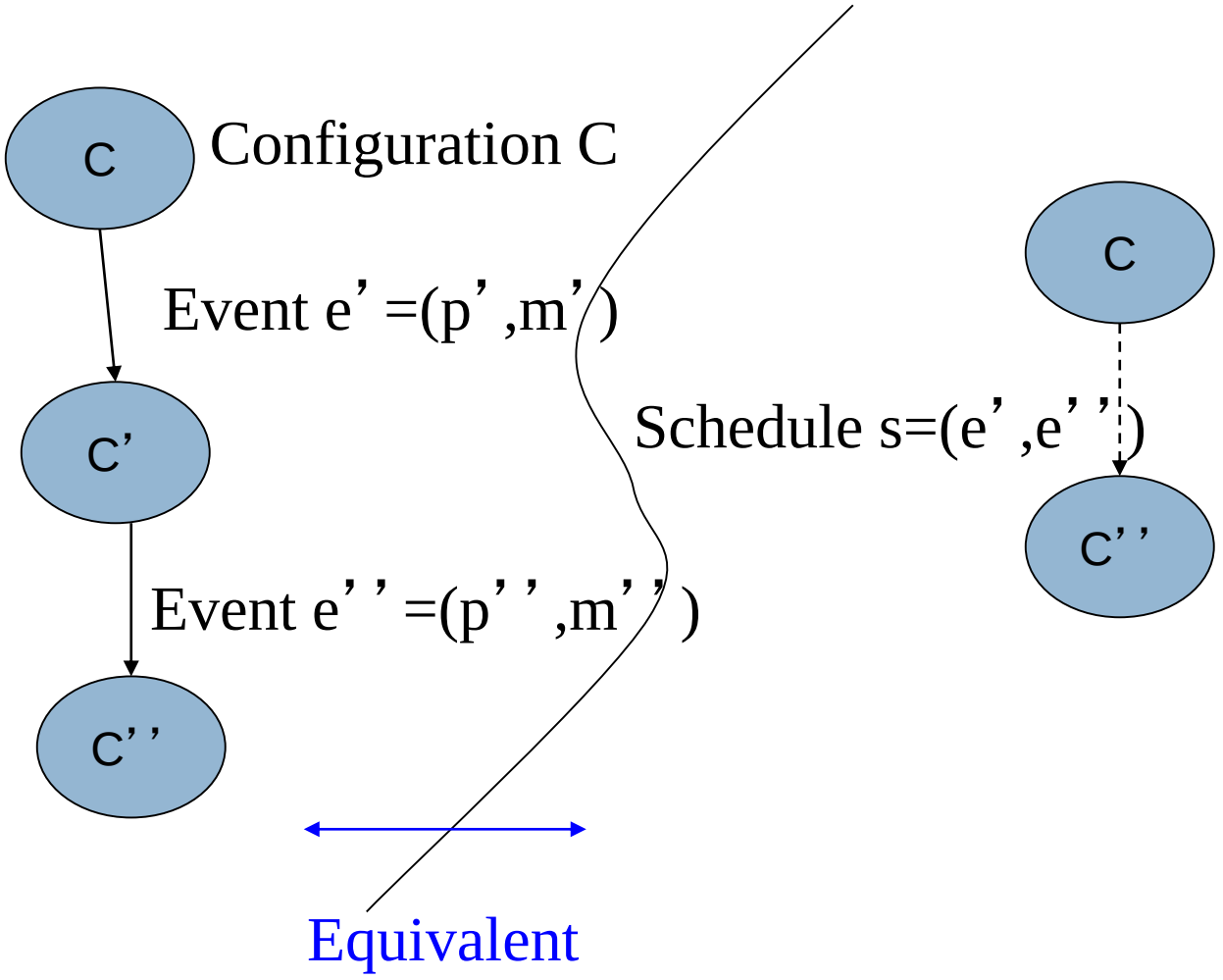
1. more restrictive system model, and
2. easier problem
   - Why is this is ok?

Network


p
p'
send(p',m)
receive(p')
may return null
Global Message Buffer
"Network"

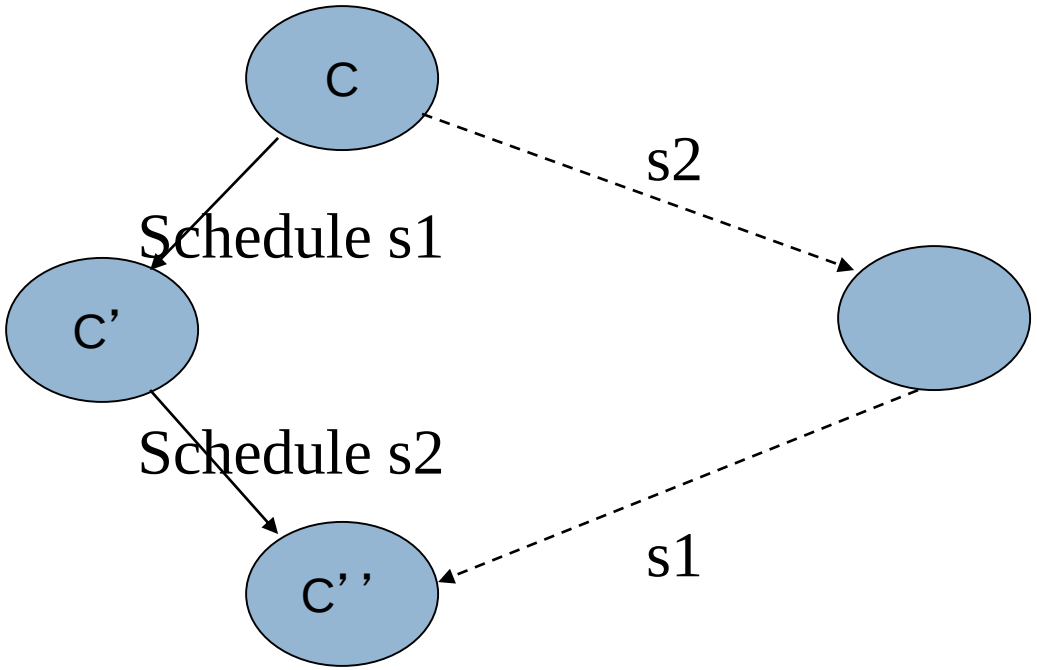
States

- State of a process
- **Configuration**=global state. Collection of states, one for each process; alongside state of the global buffer.
- Each Event (consists of 3 steps, executed atomically)
  - receipt of a message by a process (say p)
  - processing of message (may change recipient's state)
  - sending out of all necessary messages by p
- Schedule: sequence of events


C
Configuration C
Event e’ =(p’ ,m’)
C’
Event e’’ =(p’’ ,m’’)
C’’
C
Schedule s=(e’ ,e’’)
C’’
Equivalent

Lemma 1

Disjoint schedules are commutative



s1 and s2 involve disjoint sets of receiving processes, and are each applicable on C

Easier Consensus Problem

- Easier Consensus Problem:
  - some process eventually sets yp to be 0 or 1
- Only one process crashes
  - we're free to choose which one

Easier Consensus Problem

- Let config. C have a set of decision values V reachable from it
  - If |V| = 2, config. C is bivalent
  - If |V| = 1, config. C is 0-valent or 1-valent, as is the case

- Bivalent means outcome is unpredictable

What the FLP proof shows

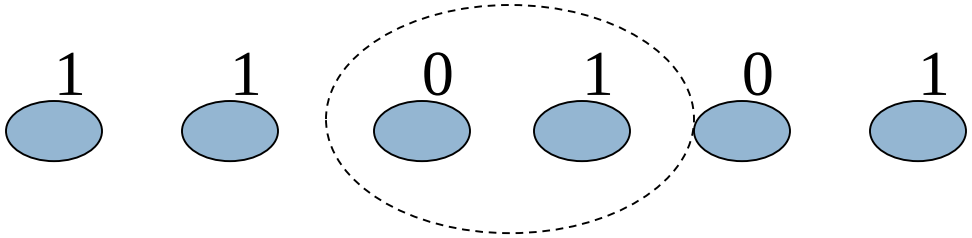
1. There exists an initial configuration that is bivalent

2. Starting from a bivalent config., there is always another bivalent config. that is reachable

Lemma 2

Some initial configuration is bivalent

- Proof by contradiction
- Suppose all initial configurations were either 0-valent or 1-valent.
- If there are N processes, there are 2^N possible initial configurations
- Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial xp value for exactly one process.



- There has to be some adjacent pair of 1-valent and 0-valent configs.

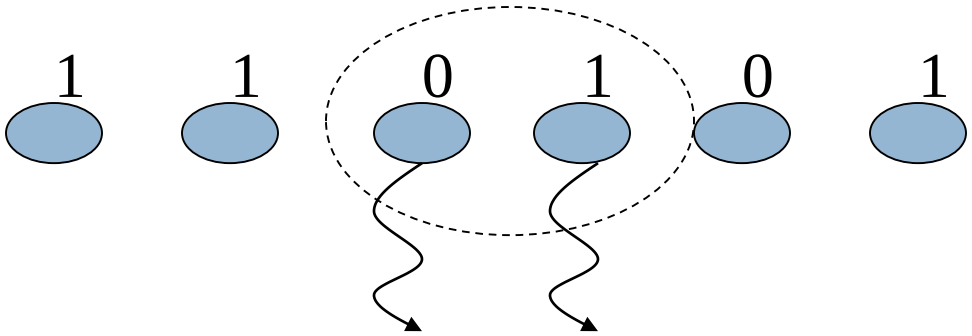
Example: Lattice for 2 processes P1 and P2 will be a square:

00--01
| |
10--11

Lemma 2

Some initial configuration is bivalent

- There has to be some adjacent pair of 1-valent and 0-valent configs.
- Let the process p, that has a different state across these two configs., be the process that has crashed (i.e., is silent throughout)



Both initial configs. will lead to the same config. for the same sequence of events

Therefore, both these initial configs. are bivalent when there is such a failure

What we'll show

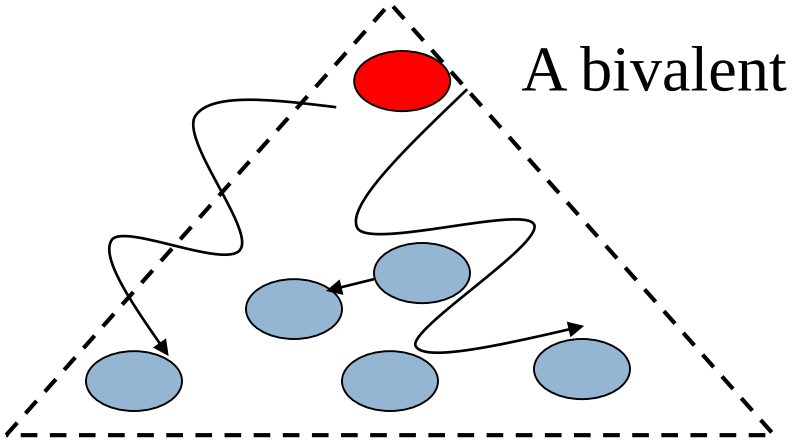
1. There exists an initial configuration that is bivalent

2. Starting from a bivalent config., there is always another bivalent config. that is reachable

Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable

Lemma 3

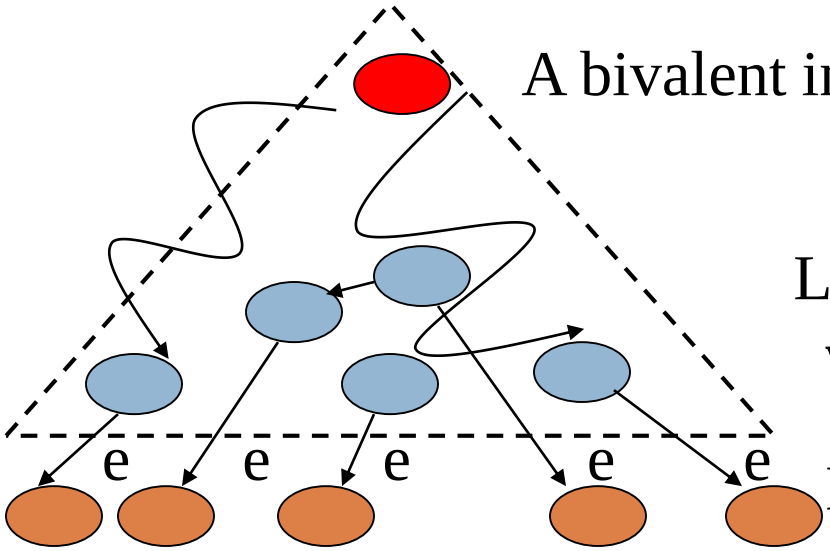


A bivalent initial config.

let e=(p,m) be some event applicable to the initial config.

Let ***C*** be the set of configs. reachable **without** applying e

Lemma 3



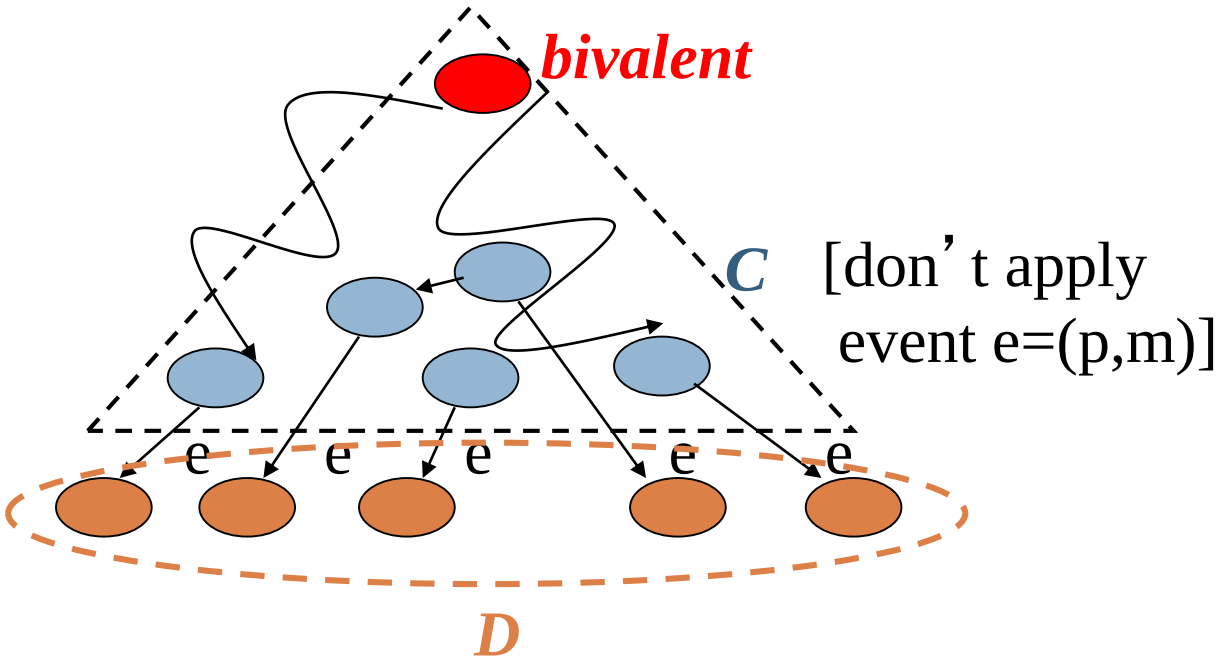
A bivalent initial config.

let e=(p,m) be some event applicable to the initial config.

Let ***C*** be the set of configs. reachable **without** applying e

Let ***D*** be the set of configs. obtained by **applying e** to some config. in ***C***

Lemma 3

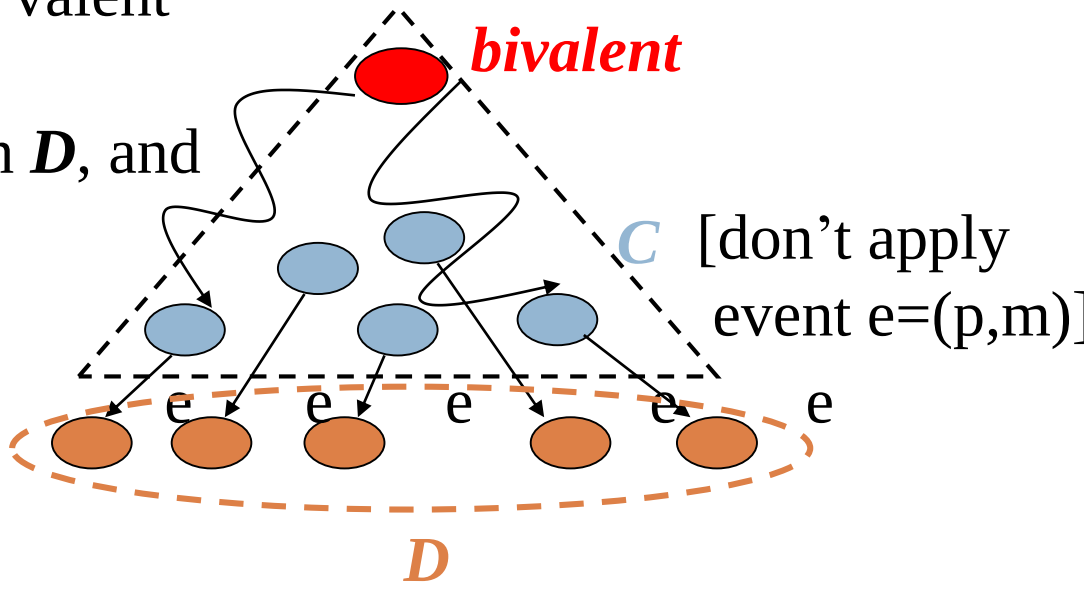


Claim. Set D contains a bivalent config.

Proof. By contradiction. That is, suppose ***D*** has only 0- and 1- valent states (and no bivalent ones)

- There are states D0 and D1 in ***D***, and C0 and C1 in ***C*** such that
  - D0 is 0-valent, D1 is 1-valent
  - D0=C0 foll. by e=(p,m)
  - D1=C1 foll. by e=(p,m)
  - And C1 = C0 followed by some event e’ =(p’ ,m’)

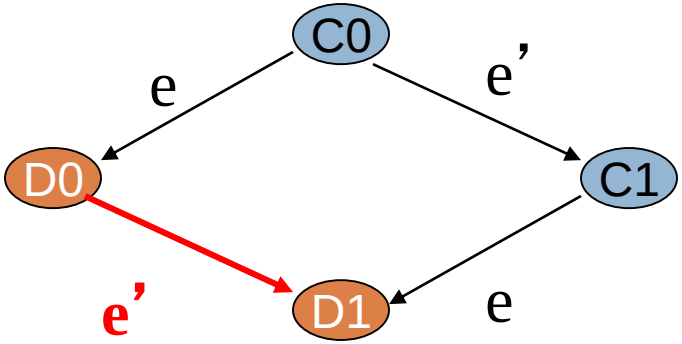
(why?)



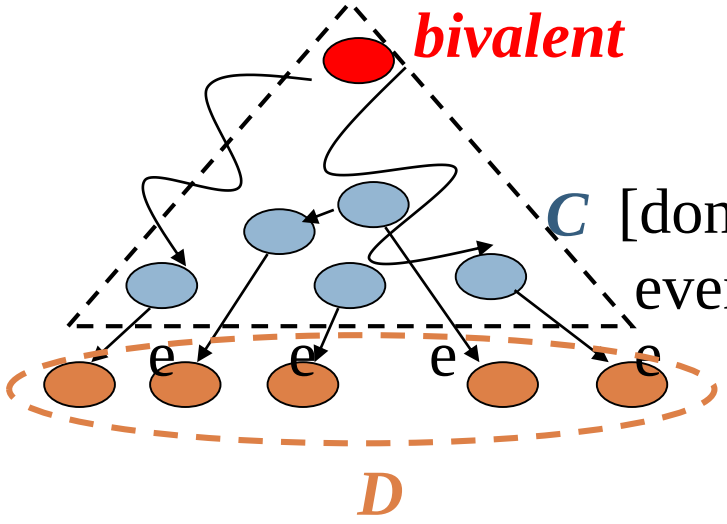
e’ =(p’ ,m’)

Proof. (contd.)

- Case I: p’ is not p
- Case II: p’ same as p



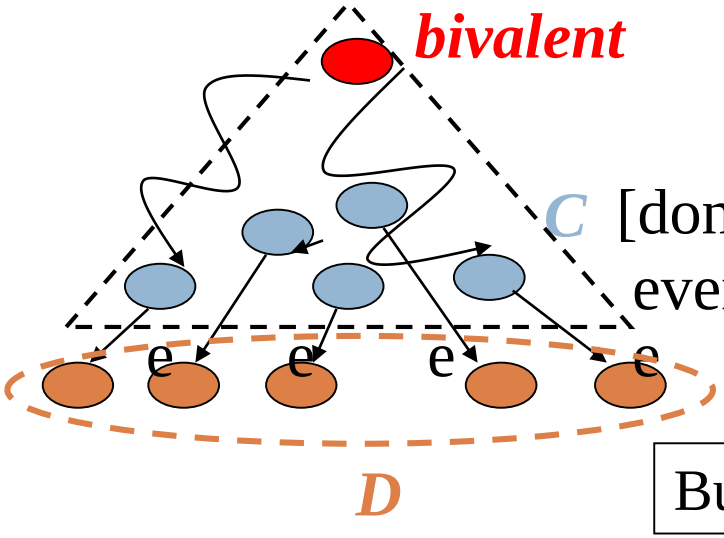
Why? (Lemma 1)
But D0 is then bivalent!



[don’t apply event e=(p,m)]
e

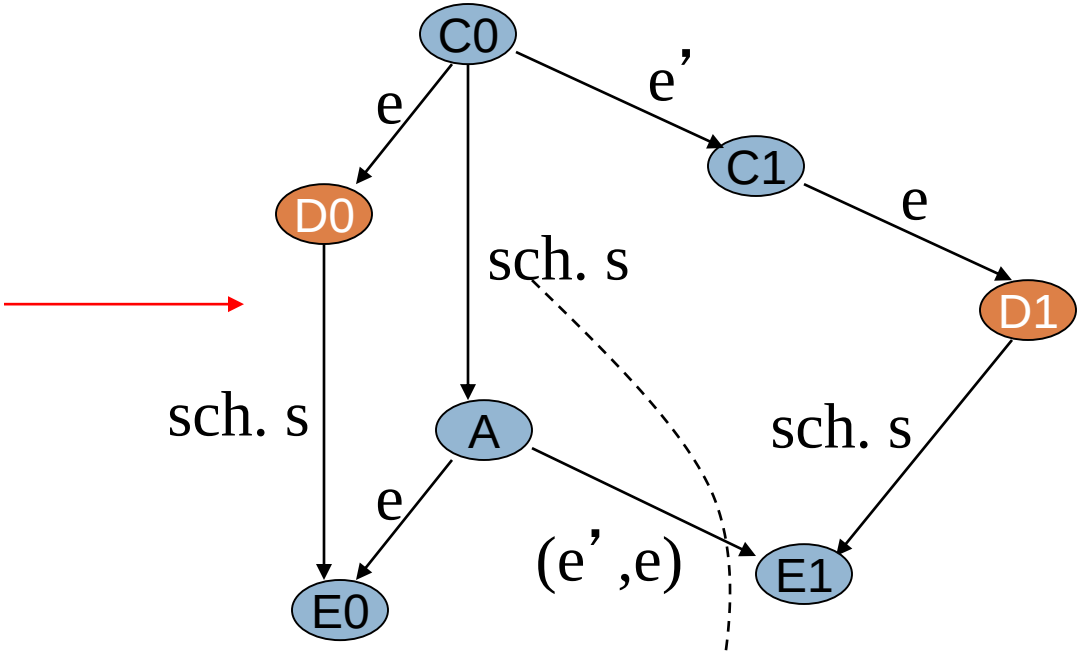
Proof. (contd.)

- Case I: p’ is not p
- Case II: p’ same as p



[don’t apply event e=(p,m)]
e

But A is then bivalent!



- finite
- **deciding run** from C0
- *p takes no steps*

Lemma 3

Starting from a bivalent config., there is always another bivalent config. that is reachable

Putting it all Together

- Lemma 2: There exists an initial configuration that is bivalent
- Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable

- Theorem (Impossibility of Consensus): There is always a run of events in an asynchronous distributed system such that the group of processes never reach consensus (i.e., stays bivalent all the time)

More on what "impossibility" means

- 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 (terminating) 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")

FLP Proof Methodology

- A very clever adversarial attack
  - They assume they have perfect control over which messages the system delivers, and when
  - They can pick the exact state in which a message arrives in the protocol
- They use this ultra-precise control to force the protocol to loop in the manner we've described
- In practice, no adversary ever has this much control

In the real world?

- The FLP scenario "could happen"
  - After all, it is a valid scenario.
  - … And any valid scenario can happen
- But step by step they take actions that are incredibly unlikely. For many to happen in a row is just impossible in practice
  - A "probability zero" sequence of events
  - Yet in a temporal logic sense, FLP shows that if we can prove correctness for a consensus protocol, we'll be unable to prove it live in a realistic network setting, like a cloud system

So…

- Fault-tolerant consensus is…
  - Definitely possible (not even all that hard). Just vote!
  - And we can prove protocols of this kind correct.

- But we can't prove that they will terminate
  - If our goal is just a probability-based guarantee, we actually *can* offer a proof of progress
  - But in temporal logic settings we want perfect guarantees and we can't achieve that goal

Thus far…

- We have an asynchronous model with crash failures
  - A bit like the real world!
- In this model we know how to do some things
  - Tracking "happens before", implementing total ordered multicast
  - Implementing replicated data, solving consensus
- But now we also know that there will always be scenarios in which our solutions can't make progress
  - Often can engineer system to make them extremely unlikely
  - Impossibility doesn't mean these solutions are wrong – only that they live within this limit

Lecture Summary

- Consensus Problem
  - Agreement in distributed systems
  - Solution exists in synchronous system model (e.g., supercomputer)
  - Impossible to solve in an asynchronous system (e.g., Internet, Web)
    - Key idea: with even one (adversarial) crash-stop process failure, there are always sequences of events for the system to decide any which way
    - Holds true regardless of whatever algorithm you choose!
  - FLP impossibility proof applies to all consensus protocols
- One of the most fundamental results in distributed systems