Consensus and Related Problems

莊 裕 澤
Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

DIS Distributed Agreement

- A set of distributed processes decide on a common value out of the values they propose.
  - One of the most fundamental problems in distributed systems

- Why is this problem meaningful?
  - Atomic transaction
  - Database distributed commitment
  - Blockchains
- Why is this problem difficult?
  - Processes may fail, and thus not behave as they are supposed to

DIS Synchronous vs. Asynchronous

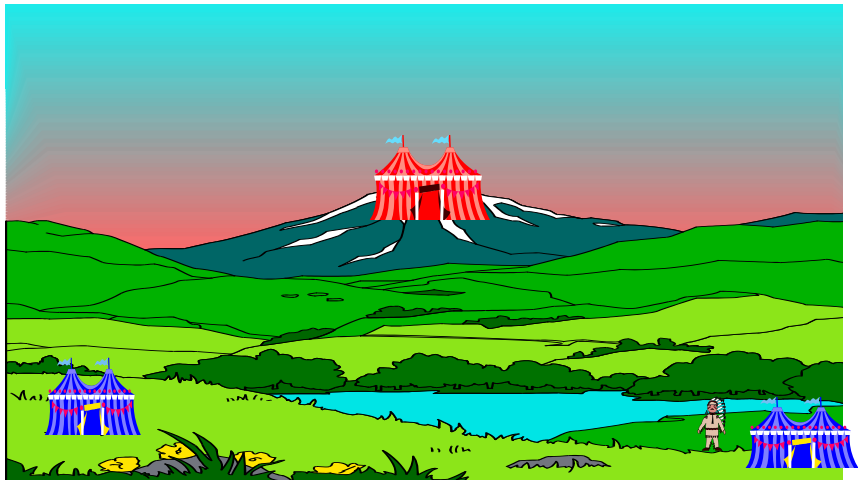
- **Asynchronous Systems**: make no assumptions about process execution speeds and/or message delivery delays.
- **Synchronous Systems**: Timeouts and other time-based protocol techniques are possible only when a system is synchronous.
  - time to execute each step of a process has known lower and upper bound.
  - message transmission time is bounded.
  - local clock whose drift rate from real time is bounded.

Types of Failures

- Message loss (but can be detected)
- Message corrupt (but forged messages can be detected (e.g., by a checksum)
- Timing failures
- Server crashes --- amnesia-crash, pause crash, halting crash (fail-stop)
- Byzantine failure (arbitrary failure)

DIS Two Generals Problem (Chinese Generals Problem)

- Can the two blue squads reach an agreement in spite of their scouts may fail to deliver messages to each other?
- Requirement: the agreement must reflect the initial proposals by both parties if the proposals are the same.
- Platform:
  - Asynchronous
  - Synchronous

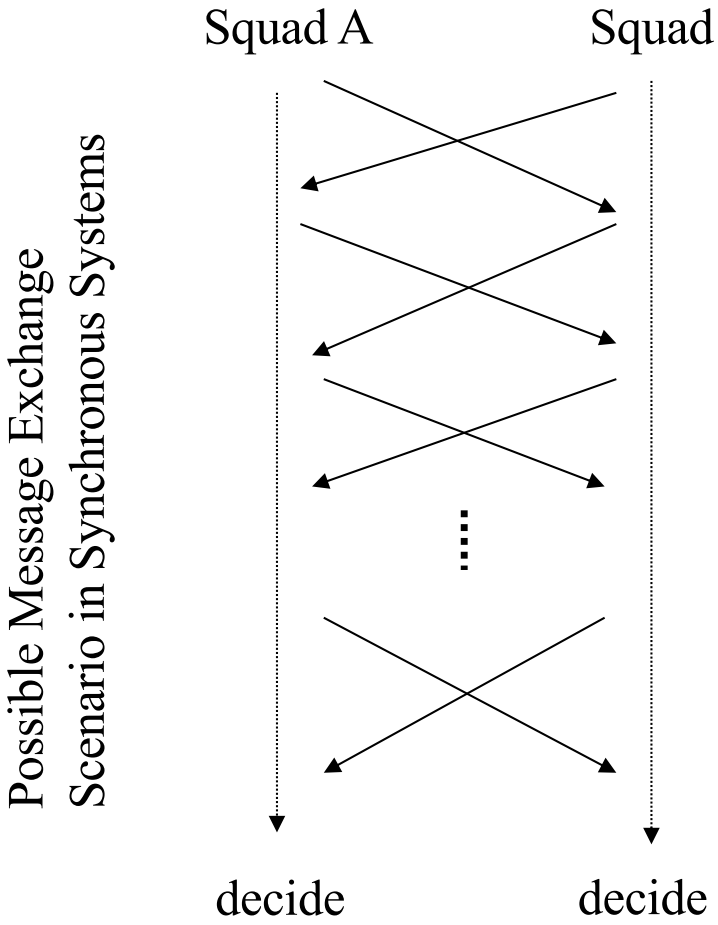


Attack at 4am

Attack at 6am

DIS Impossibility Result

- It is impossible to reach an agreement in a **synchronous** system if messages can be lost, even if there are only two processes!



Exchange of messages to reach an agreement

An algorithm must terminate in some rounds despite of failures

DIS Implication

- This implies that the problem cannot be solved in an **asynchronous** system if messages can be lost.
- Implication: the commitment problem in distributed databases.
  - This is in theory (formal proof)
  - In real life, although some scenario may cause an algorithm not to terminate, the probability of the occurrence of the scenario is extremely unlikely.

DIS Fischer, Lynch & Paterson's Impossibility Result

Fischer, M. J.; Lynch, N. A.; Paterson, M. S. (1985). "Impossibility of distributed consensus with one faulty process". Journal of the ACM. 32 (2): 374–382

2001 Dijkstra prize for the most influential paper in distributed computing

(Google citation)

(Screen captured on 2020.04.25)

DIS FLP's Impossibility Result

- In an asynchronous model where only one node may crash, there is no distributed algorithm that it will terminate and solves the consensus problem.
- Setting:
  - Number of processes (nodes) *N* is known a priori
  - Communication by message-passing.
  - Message delivery is reliable.
  - Messages can take arbitrarily long to be delivered
  - Processes operate at arbitrary speed but may fail by stopping (crash failure).

DIS Correctness Requirements

- **Validity**:
  - Any decided value must be one of the proposed values.
    - If all processes propose the same value, then they must decide on the value.
- **Safety** requirement:
  - No two non-faulty processes decide on a different value.
- **Liveness** (Termination) requirement:
  - All non-faulty processes eventually decide on a value.

Without loss of generality, for simplicity, we assume that each process proposes either 0 or 1.

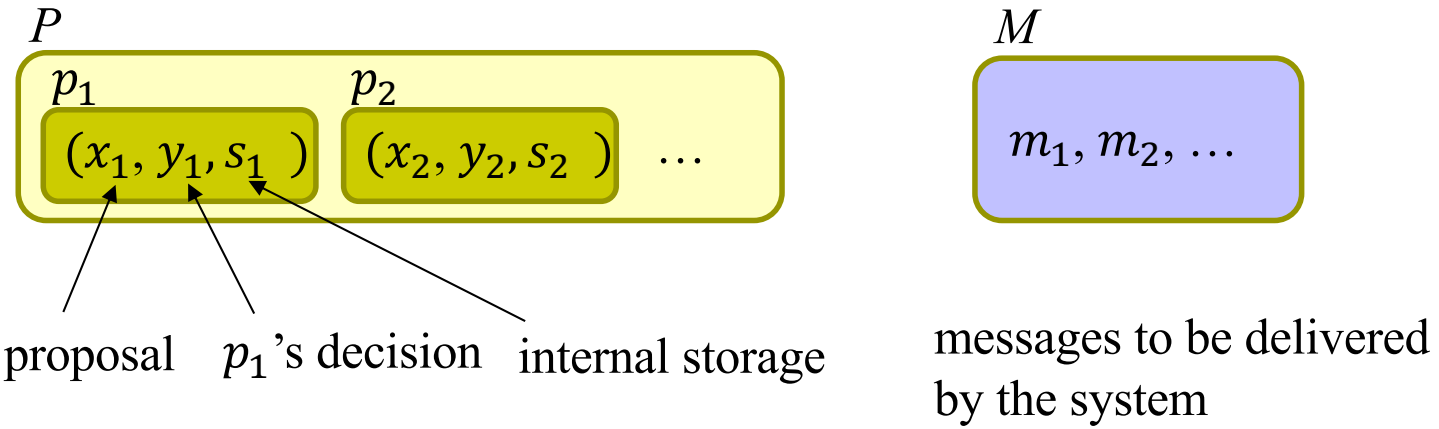
DIS An Obvious Algorithm …

- A process tells all other processes its proposal and collects proposals from all *N* nodes
- Decides when a majority of proposals are collected

What wrong with the above algorithm?

DIS Model (1)

- Configuration *C=(P, M):* the internal state of each process $p \in P$ + the message buffer *M* (sent but yet to be delivered).
- Initial configuration: each process starts at an initial state and the message buffer *M* is empty.



A step: $C_1 = (P_1, M_1) \xrightarrow{e} C_2 = (P_2, M_2)$ $\qquad C_2 = e(C_1)$

e is either a message sent from p_i to p_j (but the message is to buffer *M*, waiting to be delivered to p_j)

Or a delivery of a message from *M* to its destination

Or just an internal state transition of a process

DIS Model (2)

A step: $C_1 = (P_1, M_1) \xrightarrow{e} C_2 = (P_2, M_2)$ $\quad C_2 = e(C_1)$

e is either a message sent from p_i to p_j (but the message is to buffer M, waiting to be delivered to p_j)
Or a delivery of a message from M to its destination
Or just an internal state transition of a process

We can also combine a process p's send and receive steps into a single step e by viewing a step as occurring in two phases:

1. receive(p) is performed to receive a message m from buffer M_1 (or an empty message $\emptyset$ meaning that p just performs an internal action without receiving any message)
2. depending on p's internal state in C_1 and on m, p enters a new internal state and sends a finite set of messages to other processes (the messages go to M_2).

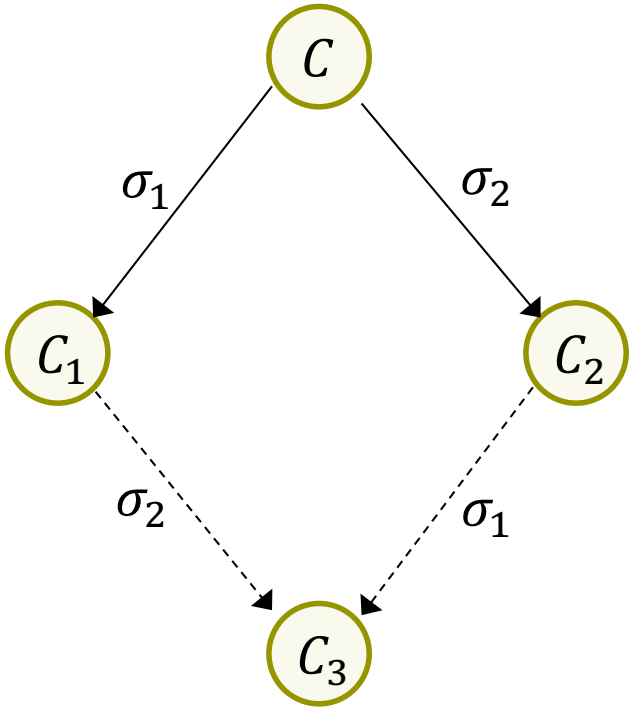
The step is completely determined by p and m, so we denote an event $e = (p, m)$.

DIS Model (3)

- A **schedule** from configuration C is a finite or infinite sequence σ of events that can be applied, in turn, starting from C.
- The associated sequence of steps is called a **run**. If σ is finite, we let $\sigma(C)$ denote the resulting configuration.
- A configuration reachable from some initial configuration is said to be **accessible**.

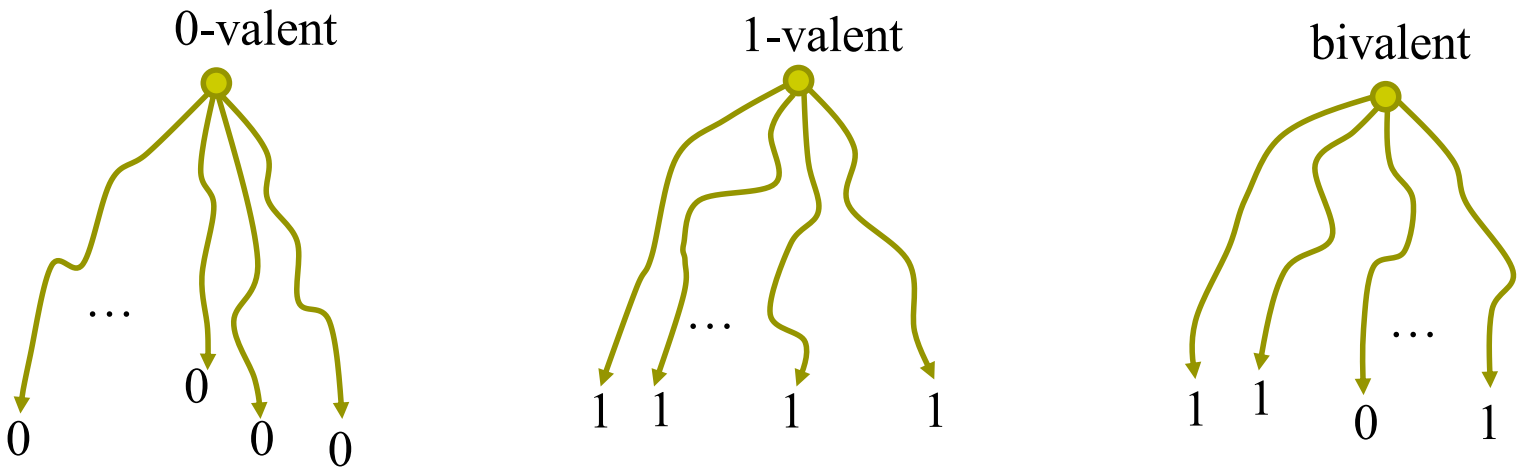
DIS Lemma 1: Commutativity of Schedules

- Suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2, respectively. If the sets of processes taking steps in σ_1 and σ_2, respectively, are *disjoint*, then σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C_3



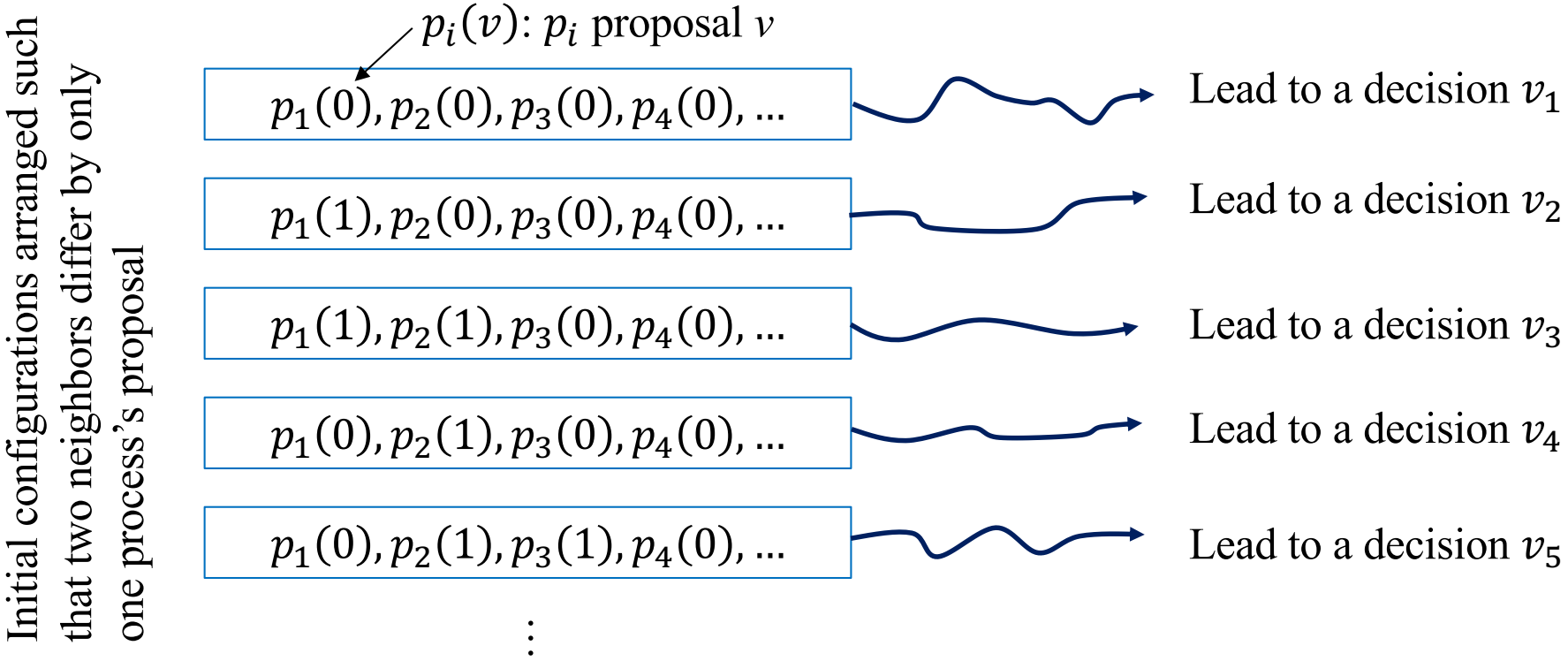
DIS Bivalent & Univalent

- A configuration *C* is **0-valent** (1-valent , respectively) if all configurations reachable from *C* lead to a decision of **0** (1, respectively).
- A configuration is **univalent** if it is either 0-valent or 1-valent.
- Otherwise, it is **bivalent**.



Lemma 2: bivalent initial configuration

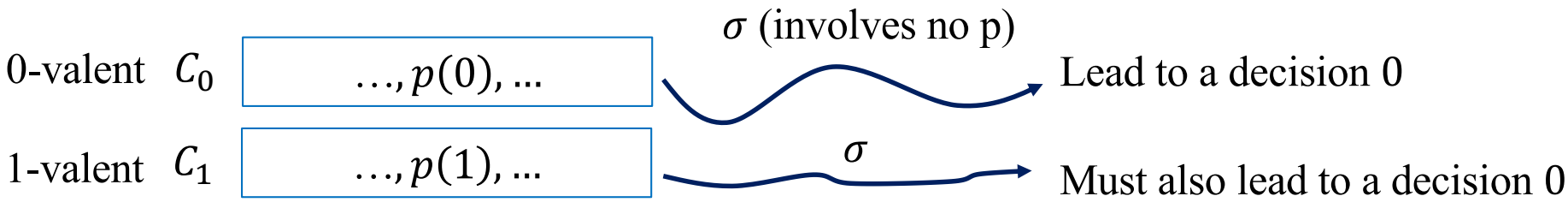
- A totally corrected algorithm **P** must have has a bivalent initial configuration.
  - Proof by contradiction.



By validity, some v_i are 0, and some v_j are 1. So there are two neighbors with different outcome: say, configuration C_0 (lead to 0) and C_1 (lead to 1).

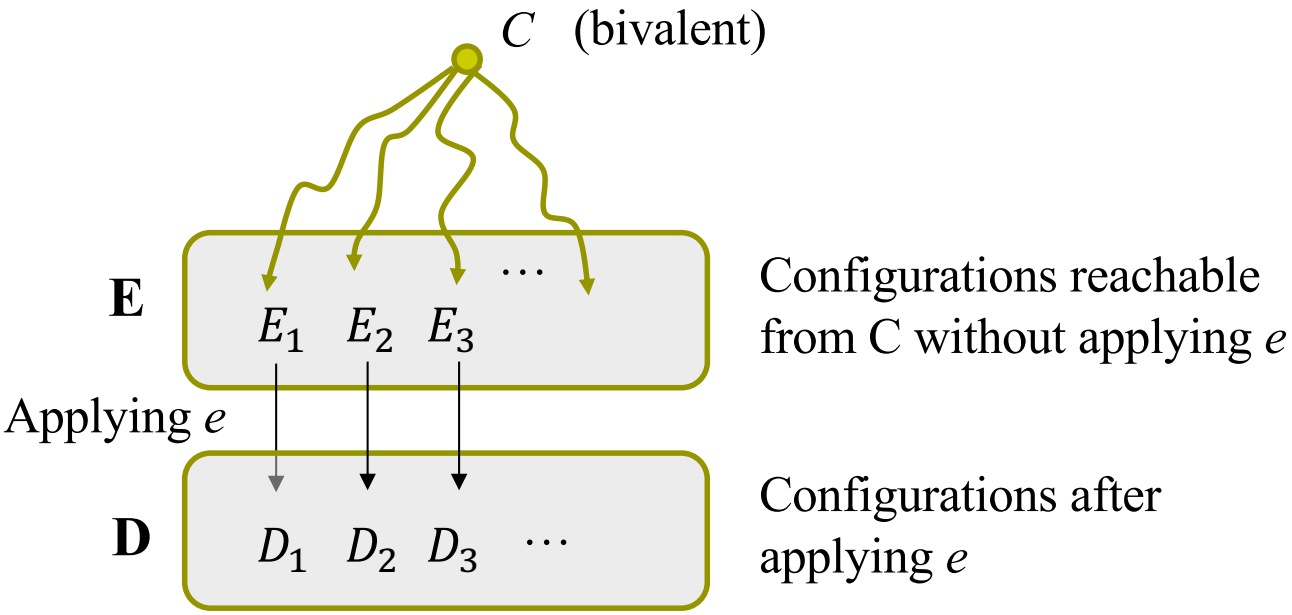
DIS Lemma 2 Proof

- Since C_0 and C_1 differ only in the initial value x_p of a single process p, and the algorithm **P** is fault-tolerant, there is some schedule σ that is applicable to C_0 and that does not involve any action of p, but still leads to a decision of 0.
- Then σ can also be applied to C_1 as well. But C_1 is a 1-valent configuration, and since C_0 and C_1 differ only in the initial value x_p of p, and σ involves no p, by the commutativity property, the result must be same as when σ is applied to C_0, contradiction.



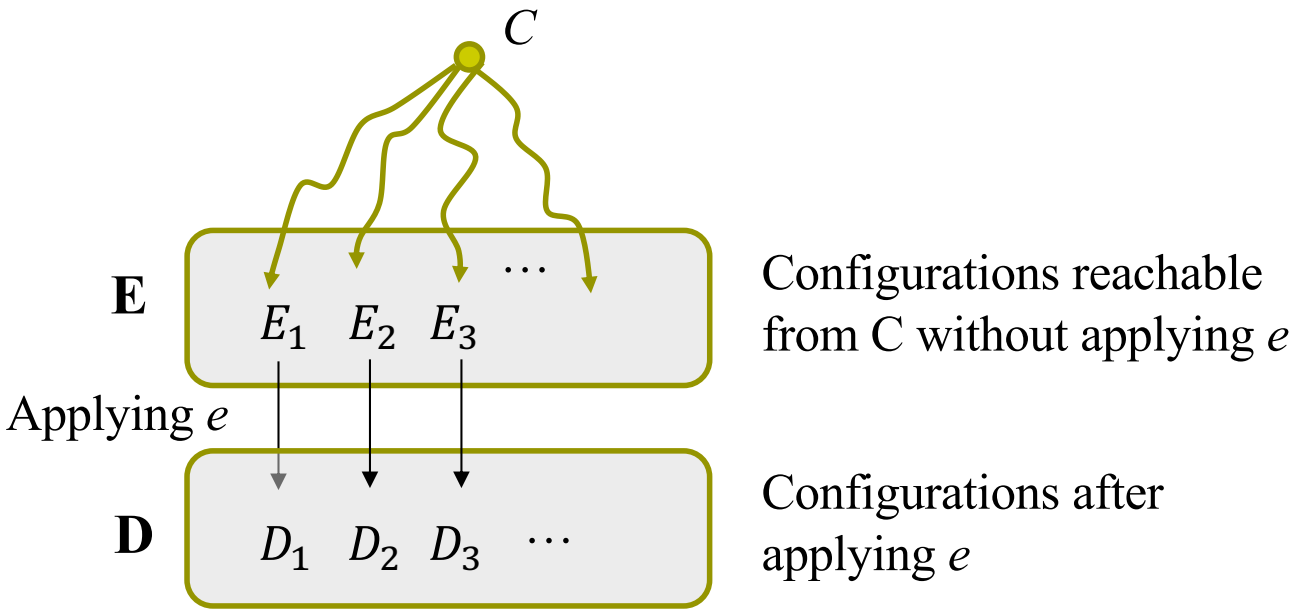
DIS Lemma 3:

- Let C be a bivalent configuration of **P**, and $e = (p, m)$ be an event that is applicable to C. Let **E** be the set of configurations reachable from C without applying e, and let $\mathbf{D} = e(\mathbf{E}) = \{e(E) | E \in \mathbf{E}$ and e is applicable to $E\}$ be the set of configurations after applying e to all those in **E**. Then, **D** contains a bivalent configuration.



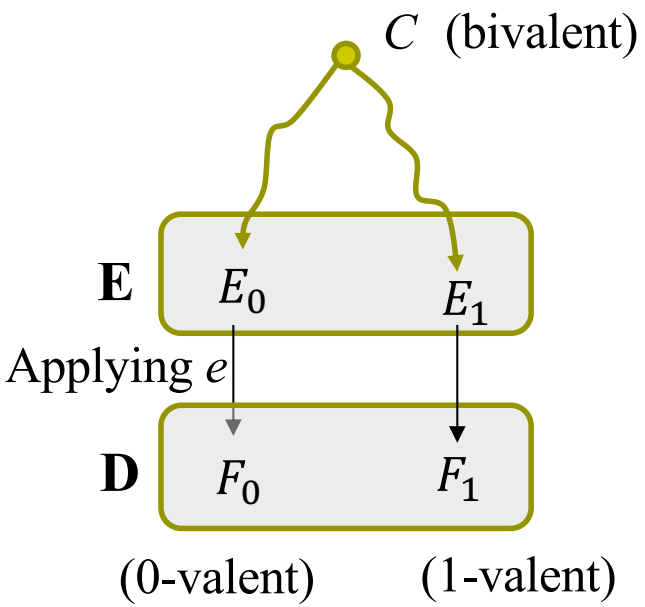
DIS Proof of Lemma 3 (1/4)

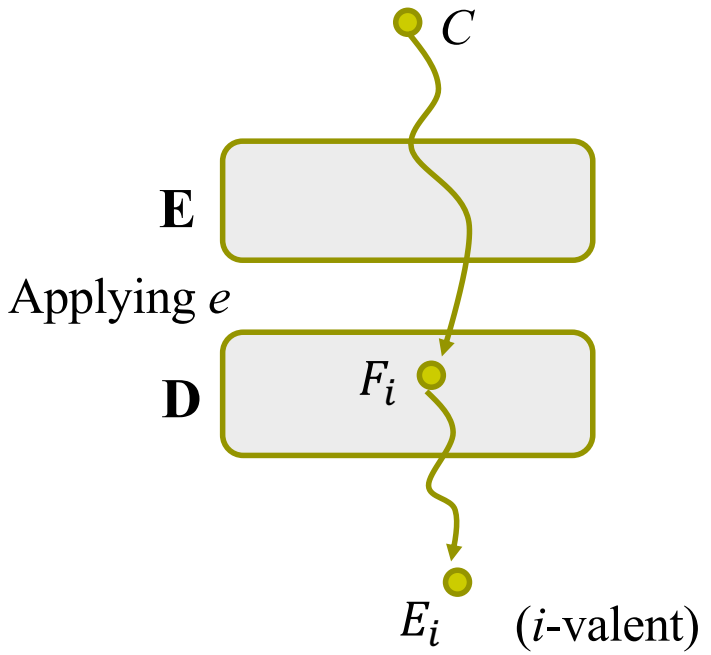
- Proof by contradiction. Assume that **D** contains no bivalent configuration.
- First, argue that **D** contains both 0-valent and 1-valent.



DIS Proof of Lemma 3 (2/4)

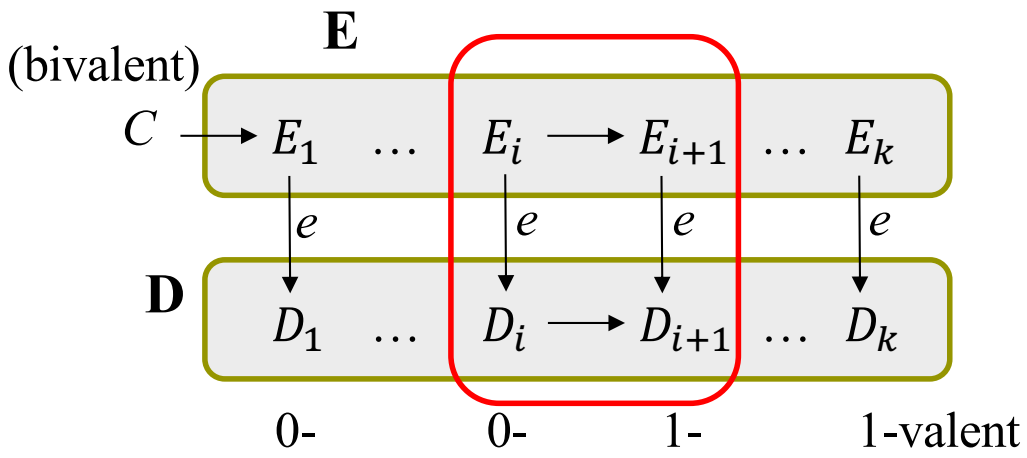
- C is bivalent, there is a 0-valent configuration E_0 and 1-valent configuration E_1 both reachable from C.
  - If both E_0 and E_1 are in **E**, then $F_0 = e(E_0)$ and $F_1 = e(E_1)$ are in **D**, and F_0 is a 0-valent while F_1 is a 1-valent.
  - If some E_i is not in **E**, then e was applied to reach E_i . So there exists some $F_i \in$ **D** from which E_i is reachable. Then F_i is i-valent (because **D** contains no bivalent configuration, and every configuration reachable from i-valent is i-valent)

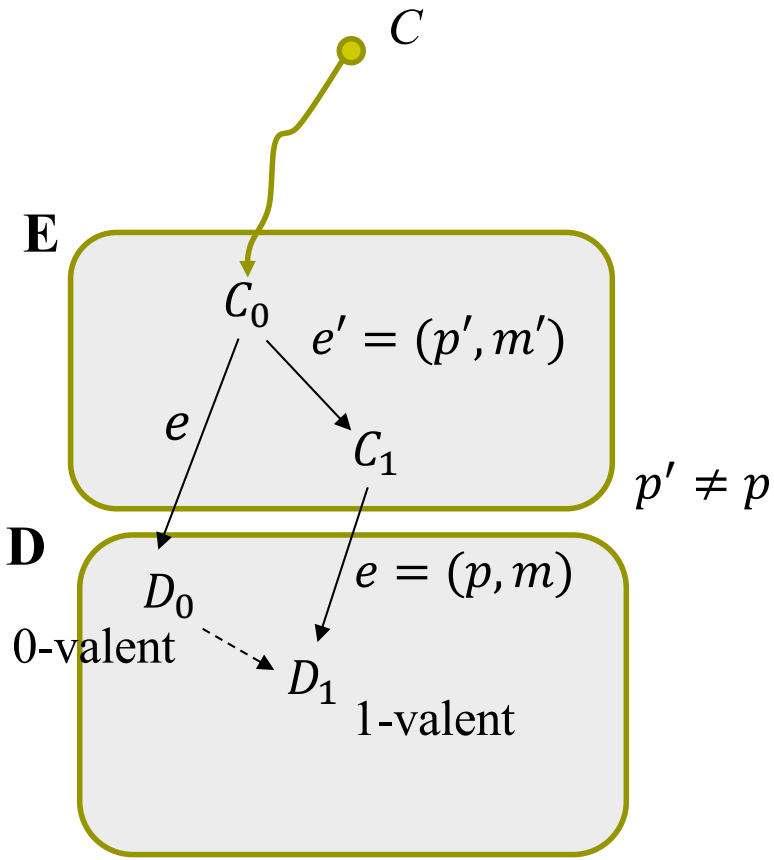




DIS Proof of Lemma 3 (3/4)

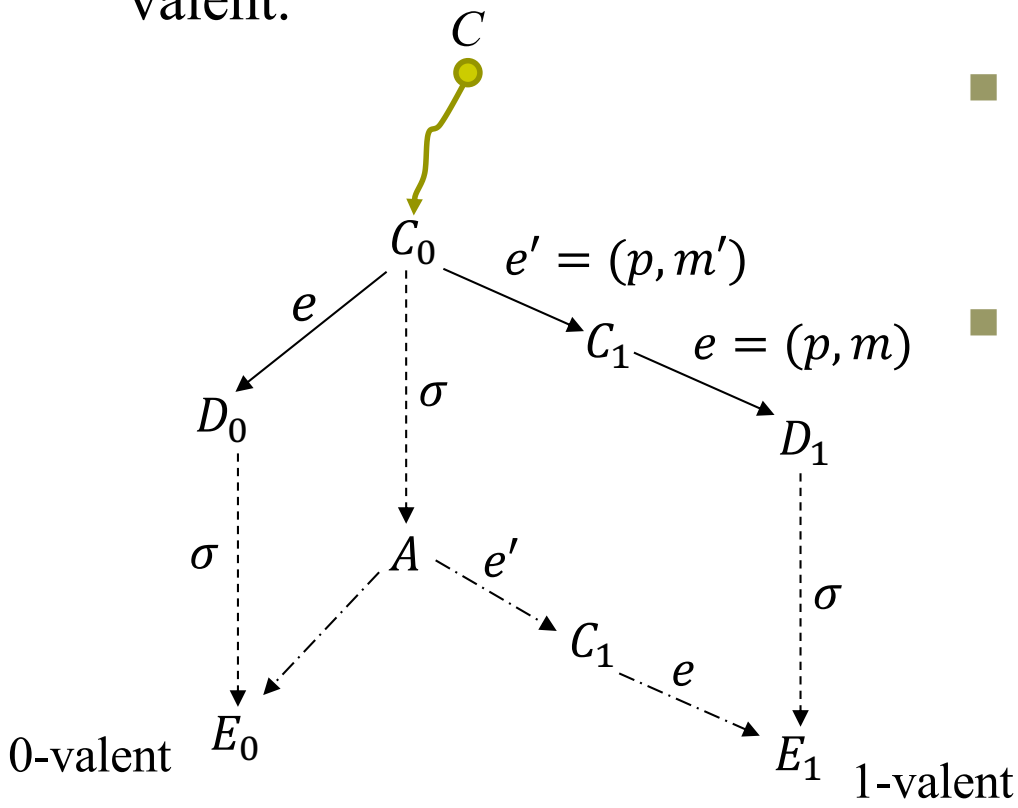
- There exists neighbors $C_0, C_1 \in \mathbf{E}$ such that $D_i = e(C_i)$ is i-valent, $i = 0, 1$ (see left Fig.)
  - If $p' \neq p$, then by Lemma 1, $D_1 = e'(D_0)$, but D_0 is a 0-valent, it cannot have a successor that is 1-valent. Contradiction. (see right Fig.)





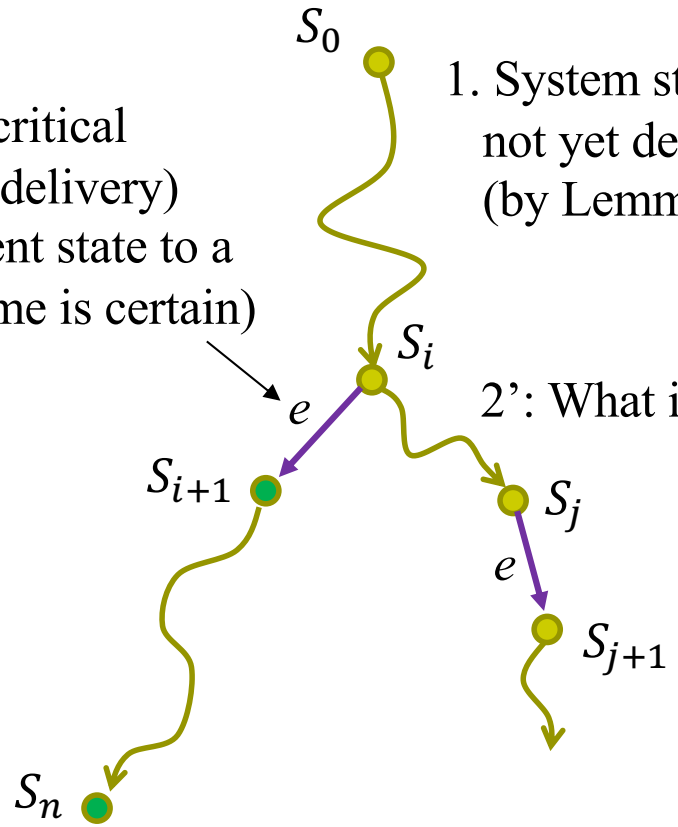
DIS Proof of Lemma 3 (4/4)

- If $p' = p$, let σ be a schedule of a finite *deciding* run (some process reaches a decision state) from C_0 in which p takes no steps (such run exists as **P** is fault-tolerant), and let $A = \sigma(C_0)$.
- By Lemma 1, σ is applicable to D_0 and D_1, and $E_0 = \sigma(D_0)$ is a 0-valent (because D_0 is a 0-valent), and similarly, $E_1 = \sigma(D_1)$ is a 1-valent.



- Also by Lemma 1, σ is applicable to D_0 and D_1, and $E_0 = e(A)$ and $E_1 = e(e'(A))$.
- So A is a bivalent, but this contradicting to the fact that the run to A is deciding.

DIS Proof of Impossibility



1. System starts in a **bivalent** state (outcome not yet determined, could be either 0 or 1) (by Lemma 2 such a state exists)

2. There must be a critical event e (message delivery) that leads a bivalent state to a **univalent** (outcome is certain)

2': What if this e does not happen?

3': The algorithm must still allow the system to proceed as it is fault tolerant, and by Lemma 3 there must be some bivalent state S_{j+1} reachable from S_i

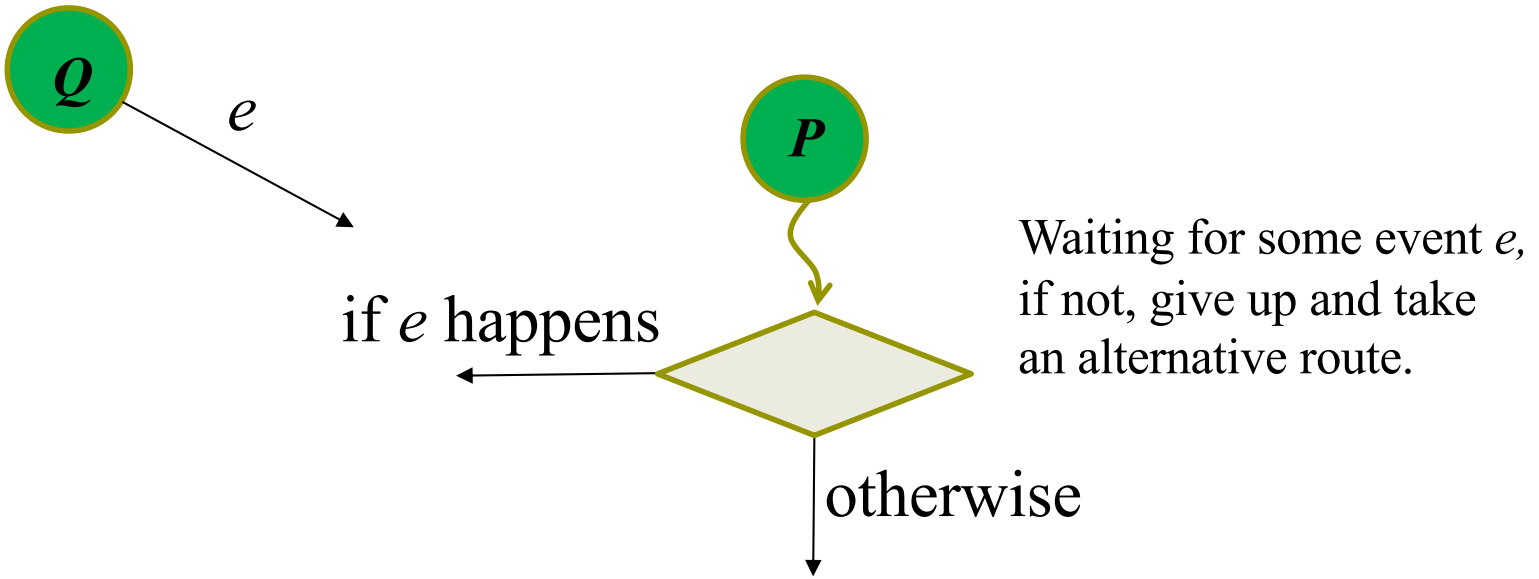
3. Eventually the algorithm terminates and decide a value (say, 0)

As we can see, by just delaying critical event e, we can "pump" the computation further without entering a univalent state. By continually this arguing, we can construct a scenario causing the claiming algorithm not to terminate.

DIS Intuition to the Impossibility

Note that the impossibility proof doesn't even need any process to fail!

It is the unbounded delay that strike the system: an algorithm/protocol cannot distinguish between crash failure and slow speed (in processing or message delivery)!



DIS The Meaning of "Impossibility"

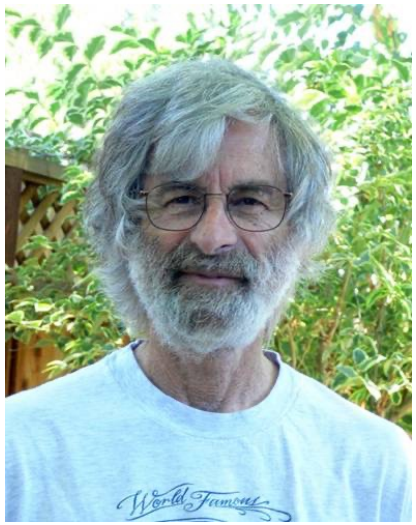
- In formal proofs, an algorithm is correct (solve a problem) if
  - It is safe (does not have output violate the problem requirement)
  - It always terminate (will eventually have the result) (liveness)

- The impossibility result shows that in some extremely rare situation, an algorithm attempting to solve consensus may not terminate.
- But life goes on, we still need to solve this consensus problem (as it is so fundamental in distributed system). It is just in what degree we can tolerate the extreme situation.

DIS The Paxos Algorithm (Lamport, 1998)

- Readings:
  - The Part-Time Parliament, L. Lamport, *ACM TOCS 16(2)*, pp. 133-169, 1998.
    - The paper was submitted in 1990, not favor for publication because of nonstandard style of presentation
    - Eventually published "as is", because people began recognizing its importance, both in theory and in practice.
  - Paxos Made Simple, L. Lamport, *ACM SIGACT News 32(4)*, pp. 51-58, 2001. (Google citation)

Turing Award (2013)
Dijkstra Prize (2000, 2005, and 2014)



Leslie Lamport (Source: wiki)

DIS Paxos: Assumption

- Asynchronous processes, stopping failures, also recovery
  - A process may crash, but will recover eventually
- Messages may be lost, duplicated, but never garbled or forged, delivery speed is not guaranteed
  - No Byzantine failures
- Number of processes *N* is known a priori

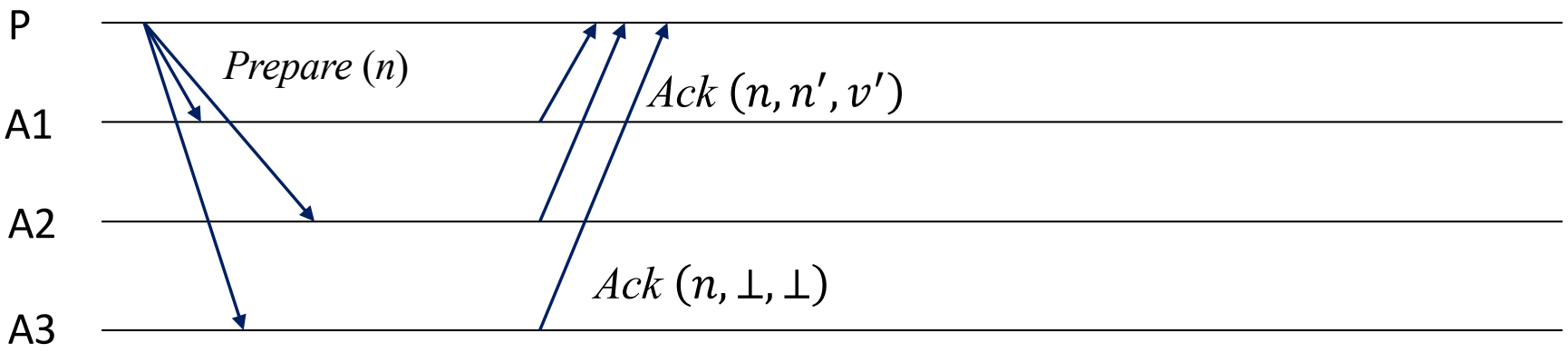
DIS An overview of Paxos Algorithm

- 3 roles of a process (although some may not play all 3 roles)
  - proposer
  - acceptor
  - learner
- 2 phases
  - Phase 1: prepare
  - Phase 2: propose

DIS Phase 1 (Prepare):

a) A proposer selects a proposal number n and sends a *prepare*(n) request with number n to a majority of acceptors.

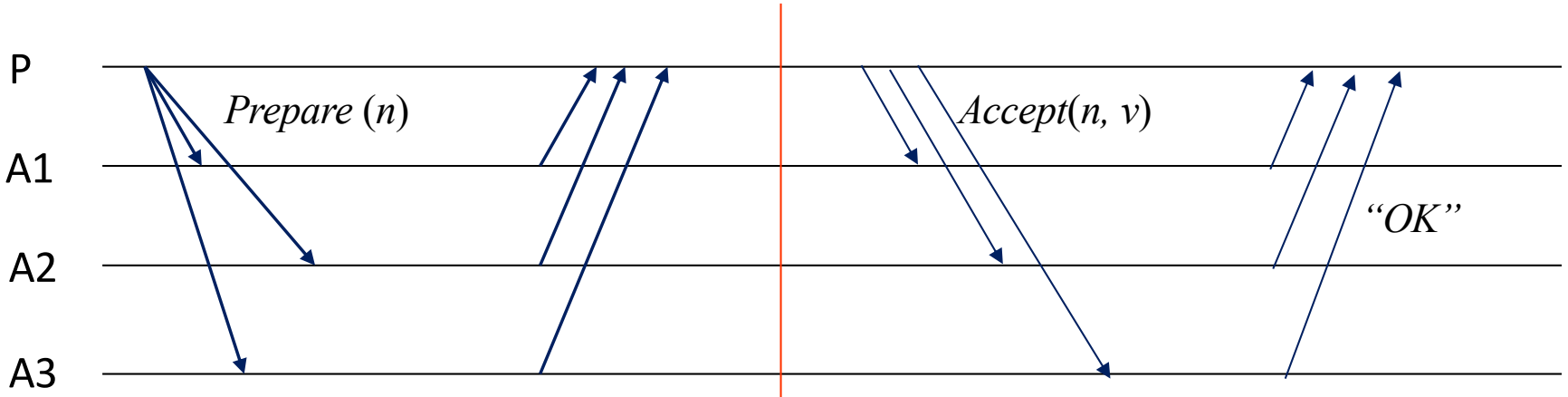
b) If an acceptor receives a *prepare*(n) request with n greater than that of any *prepare* request to which it has already responded, then it responds to the request $Ack(n, n', v')$ with a promise not to accept any more proposals numbered less than n, where v' is the value of the highest-numbered proposal that it has accepted and n' is its number if there is such a proposal, otherwise $n' = v' = \perp$.



DIS Phase 2 (Propose):

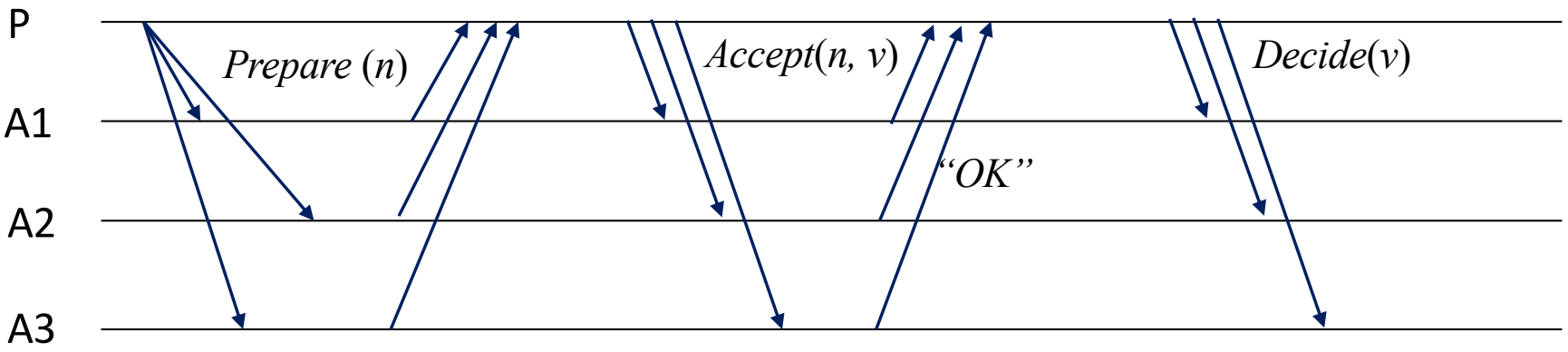
a) If the proposer receives a response to its *prepare*(n) requests from a majority of acceptors, then it sends an *accept*(n,v) request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

b) If an acceptor receives an *accept*(n,v), it accepts the proposal unless it has already responded to a *prepare*(n') request having $n' > n$.



DIS (Phase 3) Decision:

- Whenever an acceptor accepts a proposal *accept*(*n*,*v*), it informs all learners *Accept*(*v*).
- When a Learner receives *Accept*(*v*) from a majority of acceptors, it decides *v*, and sends *Decide*(*v*) to all other learners.



DIS Safety Property

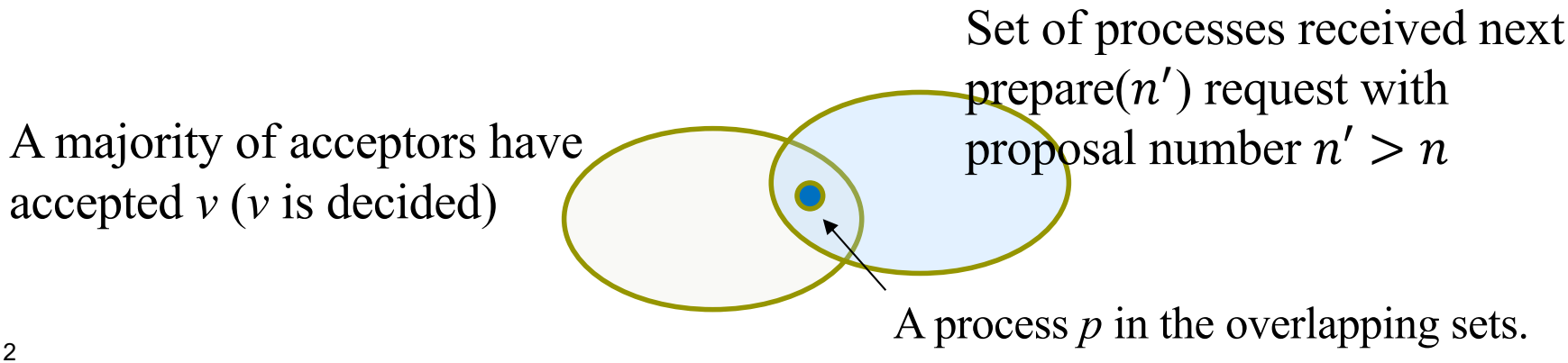
- If a proposal with value *v* is decided, then every higher-numbered proposal issued by any proposer has value *v*.

This implies:

No two correct acceptors accept/decide on different values.

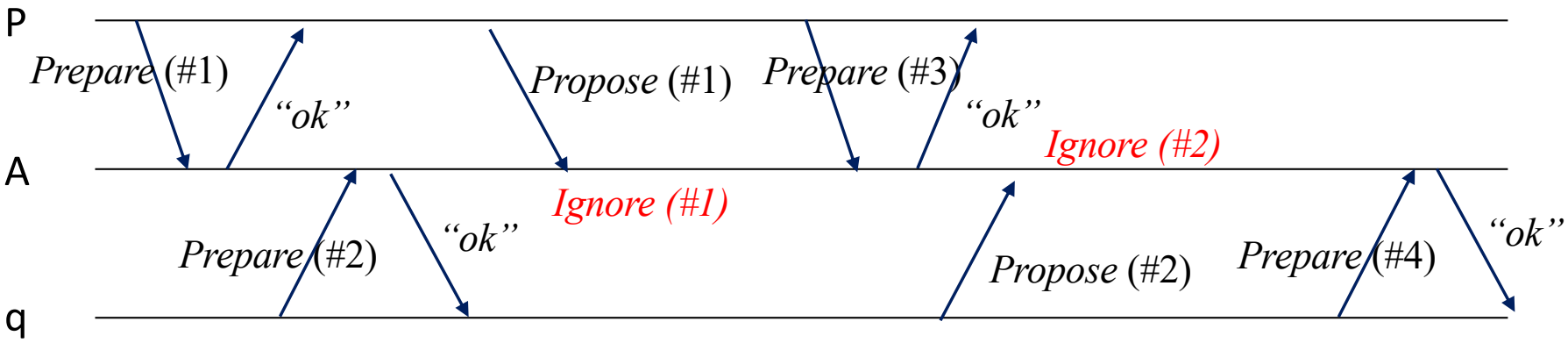
DIS Safety Proof

- Suppose (n, v) is the earliest proposal that is decided.
  - If none, safety holds.
- Let (n', v') be the earliest issued proposal after (n, v) with a different value $v' \neq v$, and assume it is proposed by process q.
- For q to propose (n', v'), its prepare(n') message must be acknowledged by a majority of acceptors.
- Thus, some process p approves both prepare(n) and prepare(n').
- But p received prepare(n') after prepare (n), so by the algorithm p must reply q's prepare(n') with $Ack(n', j, v)$, with $n \leq j < n'$. This would then let q propose v instead of v'. Contradiction.



DIS Paxos Cannot Guarantee Liveness

- Can construct a scenario in which 2 proposers each keeps issuing a sequence of proposals with increasing numbers, none of which are ever chosen.



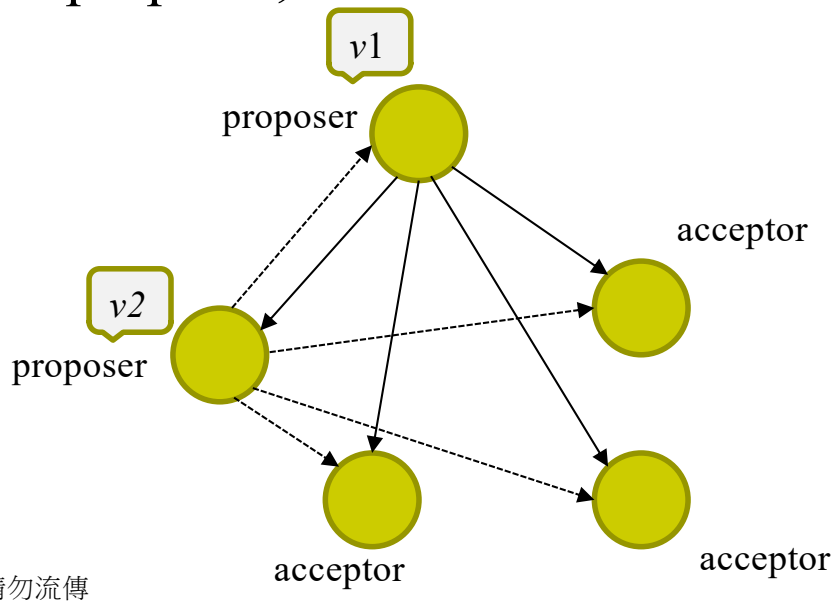
FLP's impossibility result already implies Paxos algorithm may not terminate!

But in real world, there are ways to deal with this

And the non-terminating scenarios occur extremely unlikely

DIS Wrap up the idea

- A process that wishes to propose a value, obtains a unique ballot no. for its proposal request, and solicit permissions to propose from others (act as acceptors)
- The process announces the proposed value if its proposal request is granted by a majority of the processes, or tries again if it fails.
  - ballot no. are used to number proposals and yield priority
- When a process received a proposed value, it accepts it, unless it saw a potential new proposal.
- When a majority of processes accept a proposal, the value is decided.



DIS Paxos in Real Life

- Widely used for distributed applications that need to agree on something given that nodes may crash (but majority of them are alive), messages may be arbitrarily delayed or lost, and network may be partitioned.
  - E.g.,
    - nodes agree that client X obtains a lock
    - nodes agree that Y is the master/leader
    - nodes agree that Z is the next operation to executed
    - …
  - Systems that embed some forms of Paxos
    - Google BigTable (Chubby lock service)
    - Apache ZooKeeper (Yahoo)
    - SQL server clusters

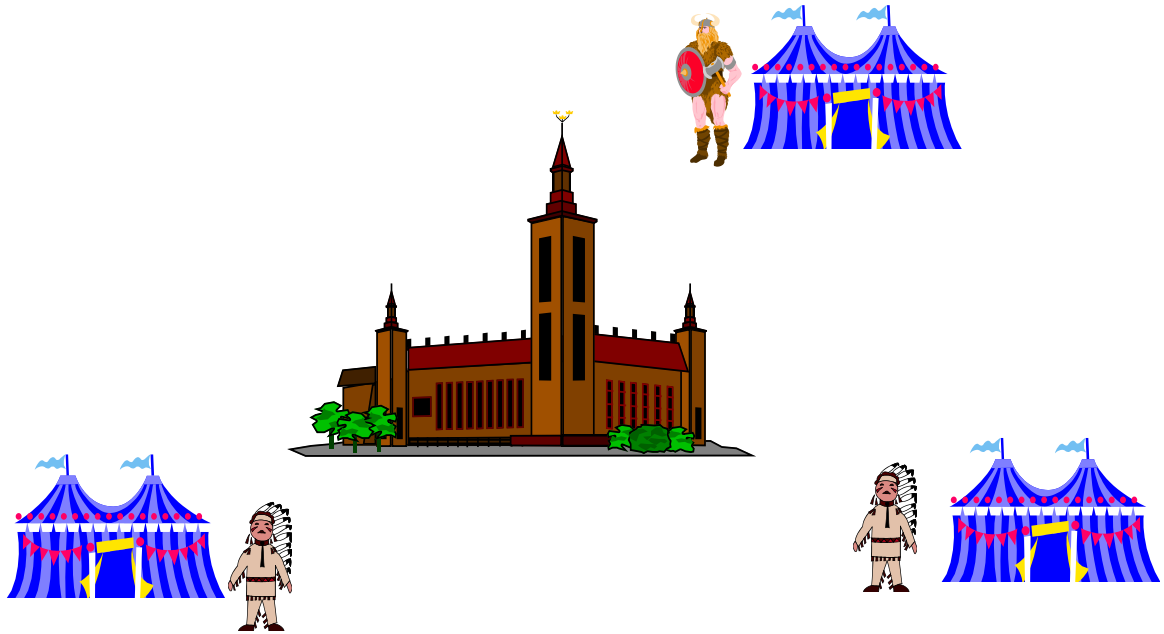
  Paxos algorithm gets complex if we need to reconfigure it to change the set of nodes running the protocol!

DIS Wrap up of the unit

Fischer, M. J.; Lynch, N. A.; Paterson, M. S. (1985). "Impossibility of distributed consensus with one faulty process". Journal of the ACM. 32 (2): 374–382

Paxos Made Simple, L. Lamport, *ACM SIGACT News 32(4),* pp. 51-58, 2001.

The Byzantine Generals Problem



DIS Reading

- Lamport, L.; Shostak, R.; Pease, M. (1982). "The Byzantine Generals Problem." *ACM Transactions on Programming Languages and Systems*. 4 (3): 382–401.


dl.acm.org › doi - 翻譯這個網頁

The Byzantine Generals Problem | ACM Transactions on ...

由 L Lamport 著作 - 1982 - 被引用 6397 次 - 相關文章

The Byzantine Generals Problem. Share on. Authors: Leslie Lamport profile ... 2.DOLEV, D. The Byzantine generals strike again. J. Algorithms 3, I (Jan. 1982).

(Screen captured on 2020.04.27)

DIS Types of Agreement Problems

- Byzantine Generals
- Distributed Consensus
- Interactive Consistency

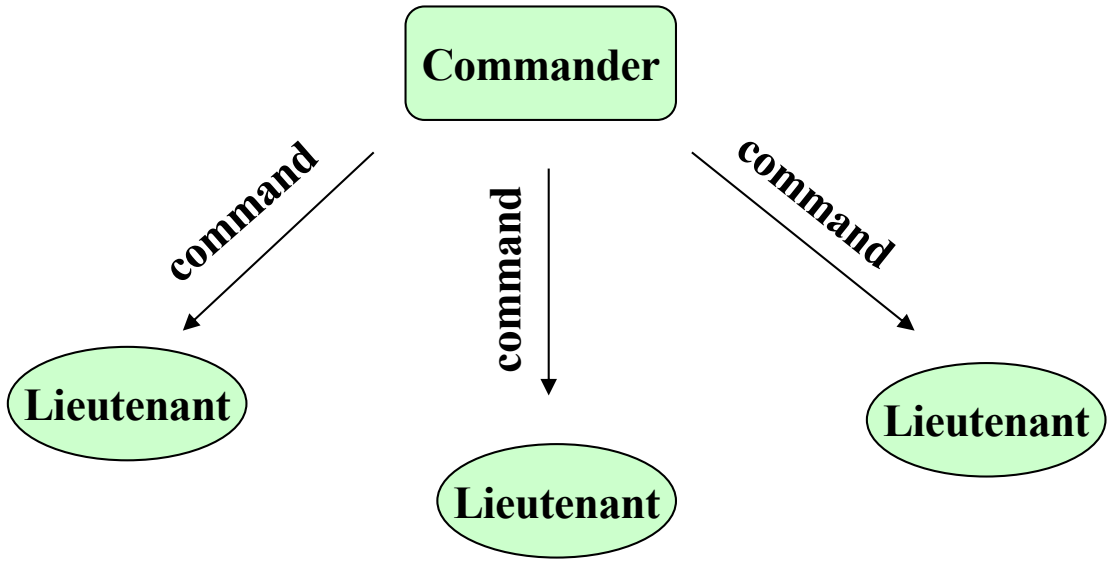
Byzantine Generals

Problem: the commander proposes a value, and then each process decides on a value.

Requirements:

Agreement: all correct processes eventually choose the same value.

Validity: if the commander is correct, then all correct processes decide on the value the commander proposed.



DIS (Distributed) Consensus

Problem: every process proposes a value, and then each process decides on a value.

Requirements:

Agreement: all correct processes eventually choose the same value.

Validity: if all correct processes propose the same value, then they decide on this value.

Strong Validity: the decision value is the same as that proposed by some correct process.

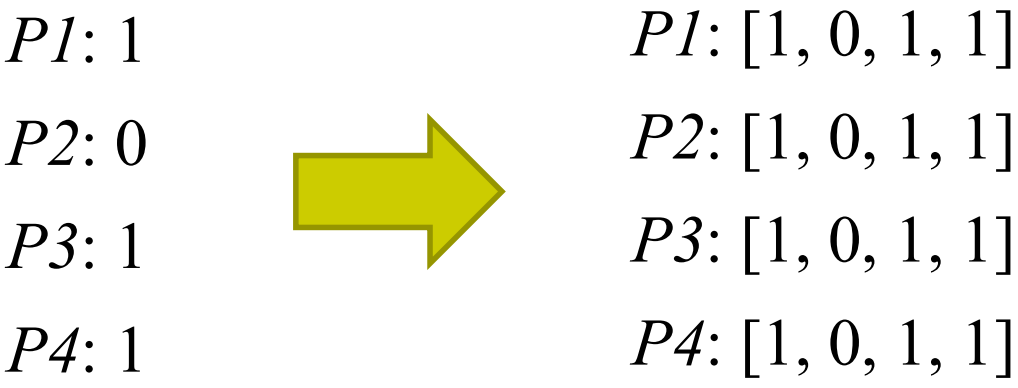
DIS Interactive Consistency

Problem: every process proposes a value, and then each process decides on a vector of values, one for each process.

Requirements:

Agreement: all correct processes eventually choose the same vector.

Validity: if process p_i proposes v_i, then all correct processes decide on v_i as the i th component of their vectors.



DIS Notations

BG_i (j, v): returns a decision value made by p_i in the Byzantine Generals problem, where p_j is the commander and it proposes v.

DC_i (v_1,…, v_n): returns a decision value made by p_i in the Distributed Consensus problem, where p_j proposes v_j, $i=1…n$.

IC_i (v_1,…, v_n)[j]: returns the j'th value in the decision vector made by p_i in the Interactive Consistency problem, where p_j proposes v_j, $i=1…n$.

DIS Relationship between these problems

- **Solve IC from BG:**

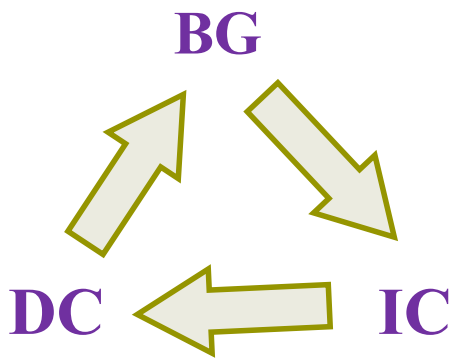
  $IC_i(v_1,\ldots,v_n)[j] = BG_i(j,v_j)$

- **Solve DC from IC:**

  $DC_i(v_1,\ldots,v_n) = majority(IC_i(v_1,\ldots,v_n)[1], \ldots, IC_i(v_1,\ldots,v_n)[n])$

  (If no majority exists, the function *majority* returns some fixed value.)

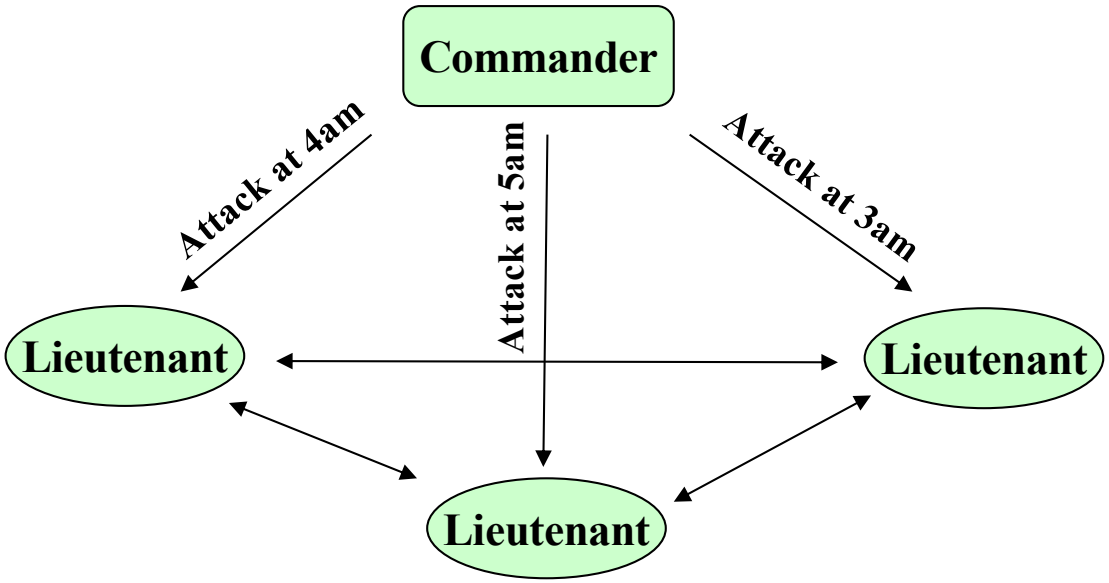
- **Solve BG from DC:**
  - The commander p_i sends its proposed value to itself and every other process.
  - All processes run DC with the values $v_1,\ldots,v_n$ they receive.
  - $BG_i(j,v) = DC_i(v_1,\ldots,v_n)$



So the three problems are equivalent!

DIS Assumptions for the Byzantine Generals Problem

- There is a direct, private channel between each pair of processes.
- Channels deliver messages correctly.
- A process knows the identity of every received message's sender.
- A process can detect the absence of a message.
- Messages are not signed.



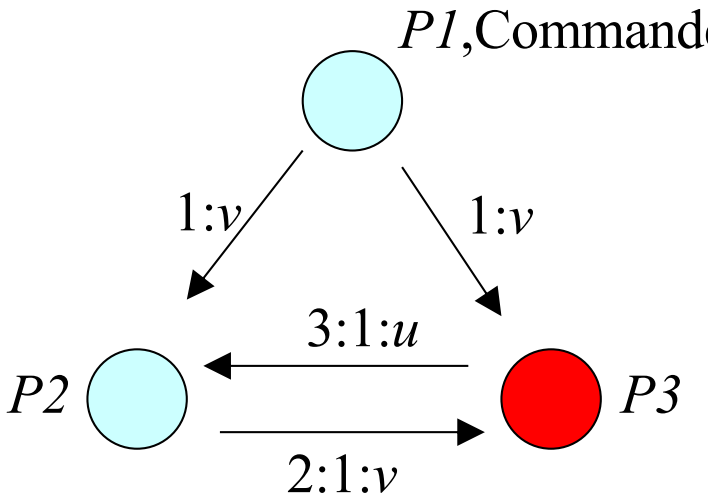
DIS Possibility and Impossibility Results

Let n be the total number of processes, and m be the number of faulty processes. Then there is no solution for the Byzantine Generals in synchronous systems if, and only if, $n \leq 3m$.

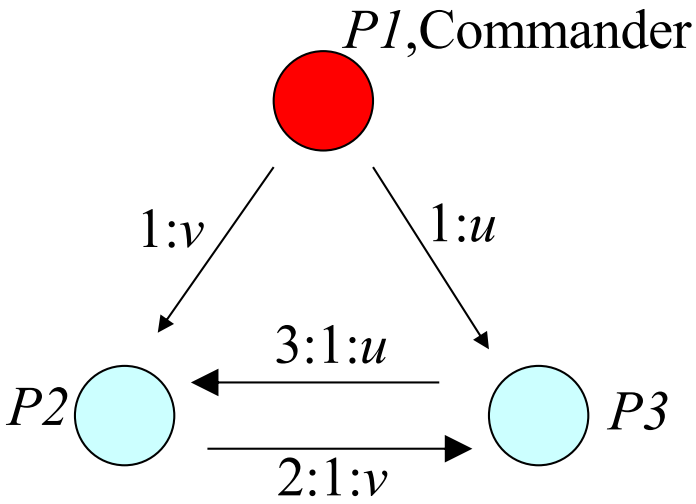
This then implies the problem cannot be solved in asynchronous system either.

DIS Impossibility Result: 3-General

Two possible scenarios:



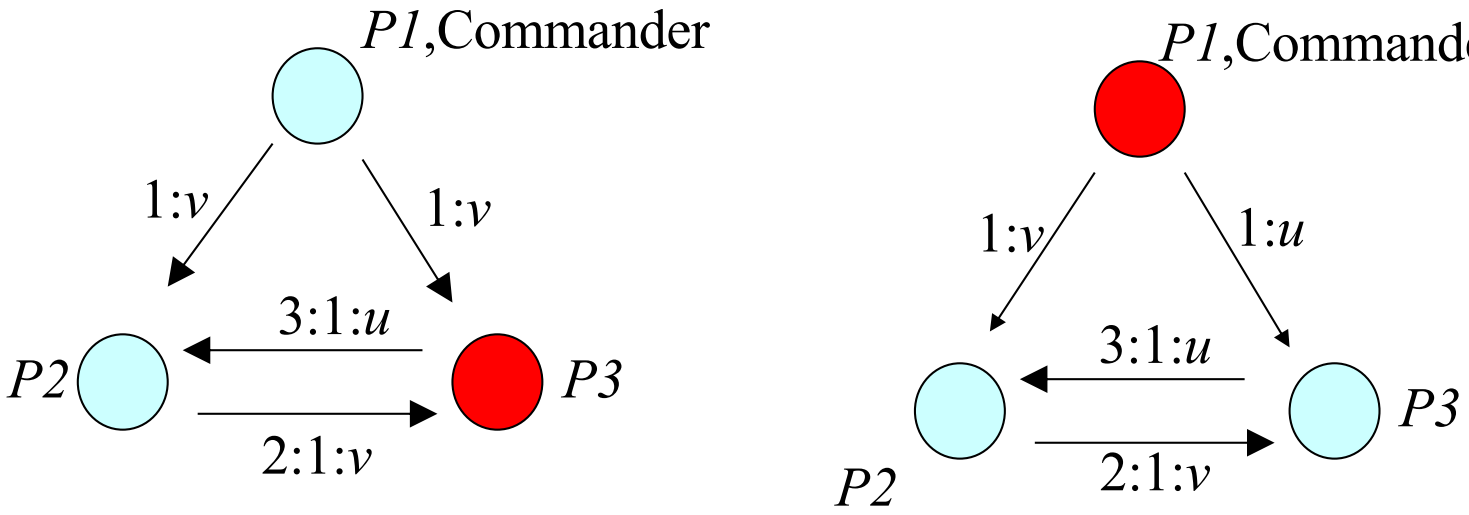
P3 is faulty



P1 is faulty

Three processes with lieutenant faulty (left)
or commander faulty (right).

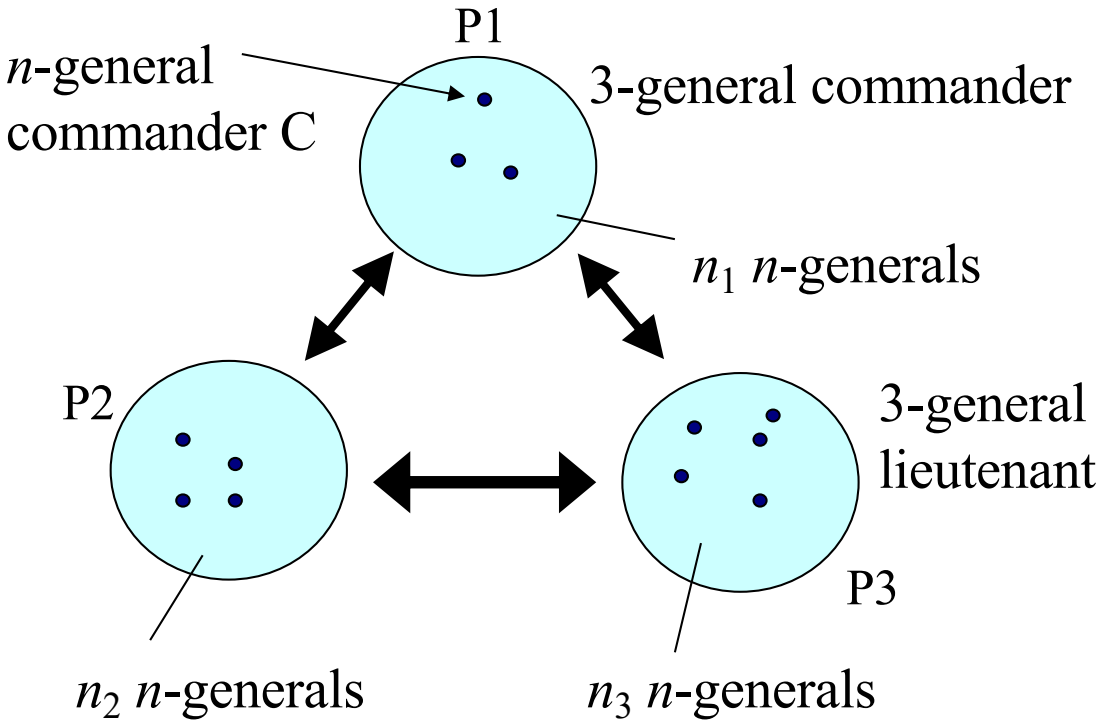
DIS Proof Outline



- P2 is unable to distinguish the two scenarios, and thus must make the same decision in them.
- P2's final decision must be v, o/w if P1 is non-faulty, it cannot reach agreement with P1.
- By a symmetric reasoning, P3 must also decide on the value proposed by P1.
- Therefore, P2 and P3 must decide on whatever P1 proposed. If P1 is faulty, it can tell P2 and P3 different values it proposed, thus causing them unable to reach agreement.

DIS Impossibility Result: *n*-General

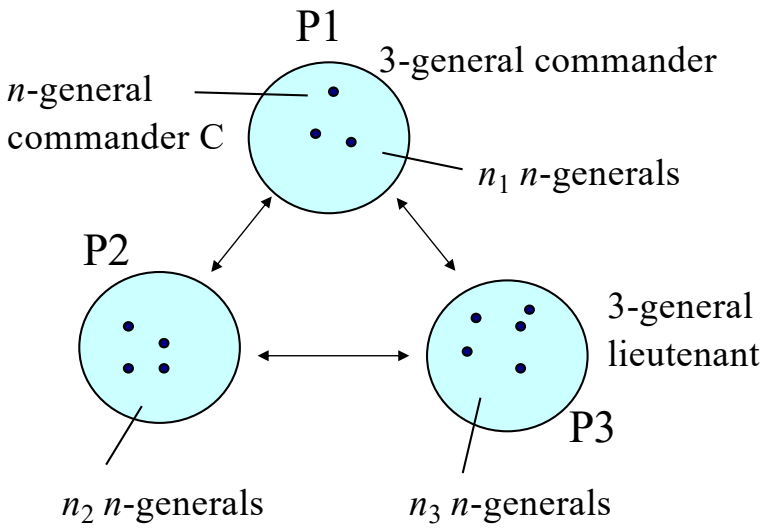
- Proof by contradiction: if *n*-general is solvable, then 3-general is solvable as well.
- Idea: simulating the n-general algorithm to solve 3-general



Assume:
- The system can tolerate m faulty processes
- $n \leq 3m$
- $n_1, n_2, n_3 \leq n/3$
- $n_1+n_2+n_3 = n$

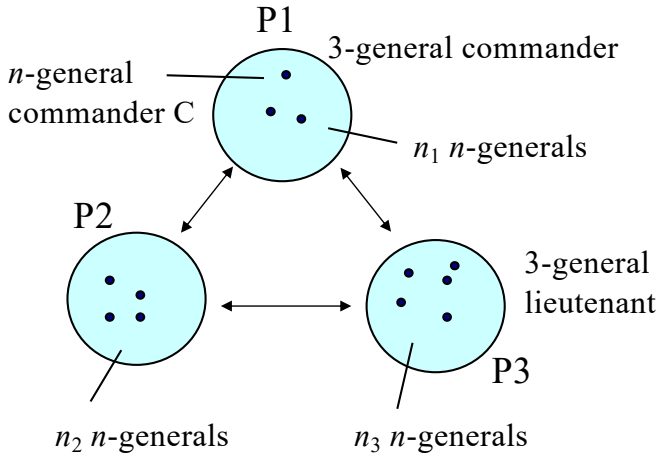
DIS Simulation Outline

- Assume an algorithm BG(n, m) to solve BG with n processes among which at most m can be faulty.
- Equally divide the n processes into 3 groups of size n_1, n_2, and n_3, each of which simulated by one of the processes in the 3-general case. The one (P1 in the fig) that simulates the n-general commander is the 3-general commander C.
- The simulation starts by letting the n-general commander C propose a value v (corresponding to that P1 proposes v), and then let each of P1, P2, and P3 simulate actions of the generals they cover according to BG(n, m).
  - Comm. between two n-general processes simulated by different 3-general processes represents an actual comm. between the two 3-general processes.
  - Comm. between two n-general processes simulated by the same 3-general processes represents an internal action of the 3-general process.



DIS Contradiction Argument

- Let each of the 3-general processes use the majority value of the *n*-general processes it simulates as its final value in 3-general problem.
- If a 3-general process is non-faulty, then all *n*-general processes it simulates are non-faulty.
- Since at most one of the 3-general processes can be faulty, the simulated BG(*n, m*) algorithm can result in at most *n*/3 *n*-general faulty processes.
- According to BG(*n, m*), all correct *n*-general processes decide on the same value.
- Hence, all correct 3-general processes decide on the same value. → × ←



DIS Solution with $n \geq 3m+1$

BG(m) : an algorithm solving Byzantine Generals for $3m+1$or more generals.

Algorithm BG(0):

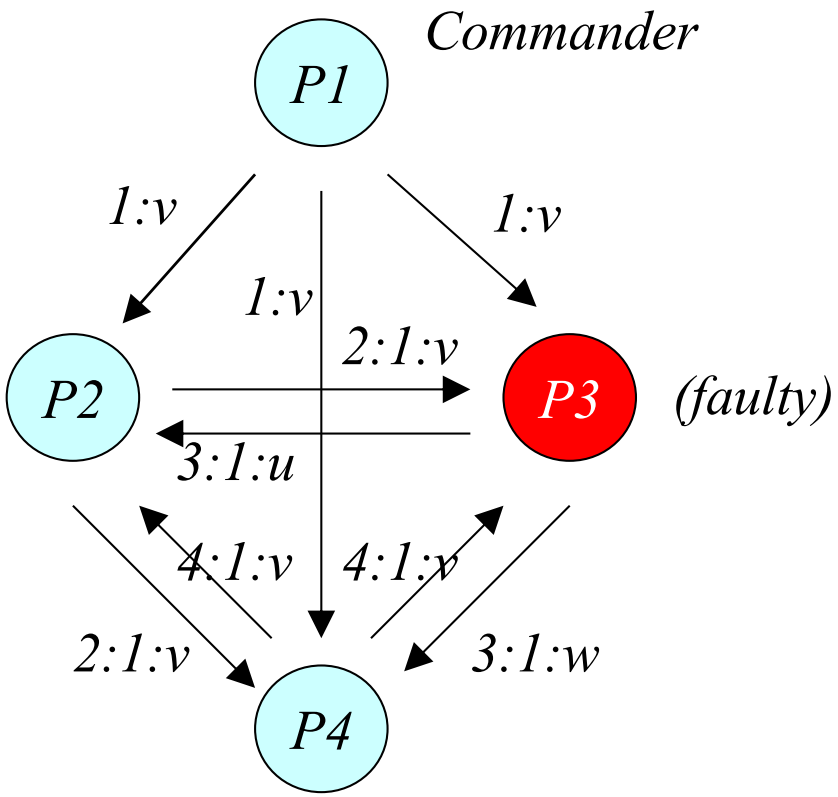
- The commander sends its value to every lieutenant.
- Each lieutenant uses the value he receives from the commander, or uses some fixed (say $\perp$) value if he receives no value.

DIS Solution with $n \geq 3m+1$ (cont.)

Algorithm BG(m):

- The commander sends its value to every lieutenant.
- For each i, let v_i be the value Lieutenant i receives from the commander, or else uses $\perp$ if he receives no value. Lieutenant i acts as the commander in algorithm BG($m-1$) to send the value v_i to each of the $n-2$ other lieutenants.
- For each i, and each $j \neq i$, let v_i be the value Lieutenant i receives from Lieutenant j in the above step (*using Algorithm BG($m-1$)*), or else uses $\perp$ if he receives no value. Lieutenant i uses the value $majority(v_1, ..., v_{n-1})$.

Solution to BG With Four Processes, at Most One Faulty (1)



P1: *v*

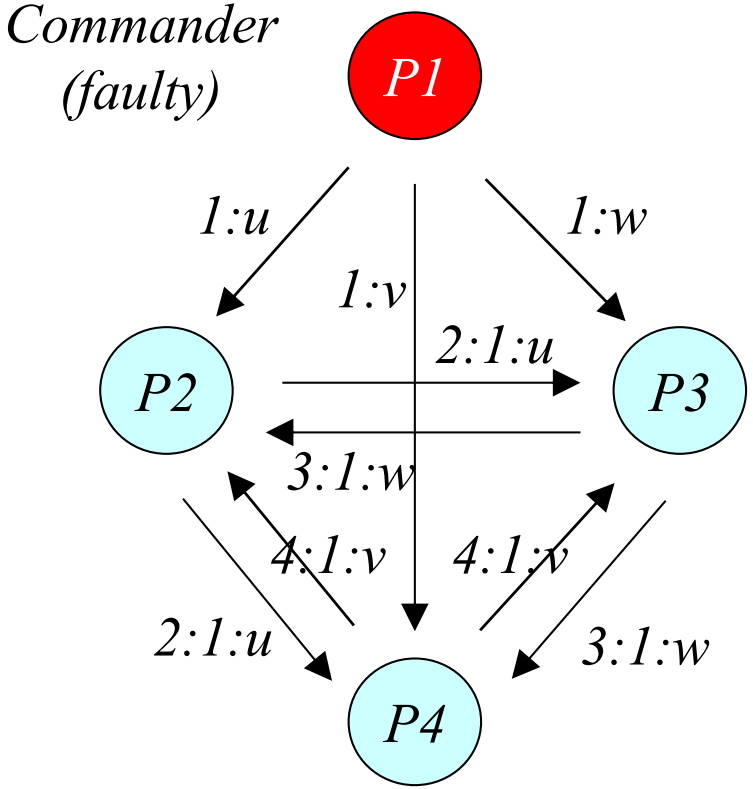
P2: *maj.* (*v*, _ , *u*, *v*) = *v*

P3: X

P4: *maj.* (*v*, *v*, *w*, _) = *v*

All correct processes decide on the same value, and is the same as *P1* proposes.

Solution to BG With Four Processes, at Most One Faulty (2)



P1: X

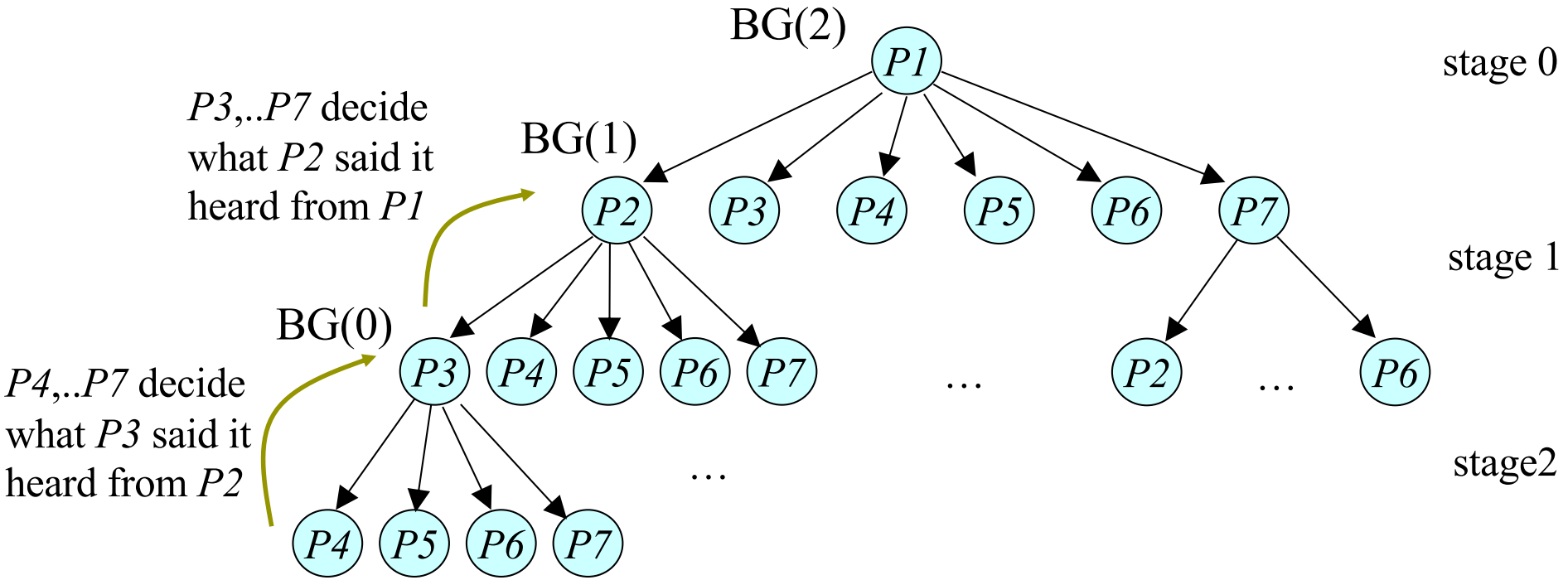
P2: *maj.* (*u*, _ , *w*, *v*)

P3: *maj.* (*w*, *u*, _ , *v*)

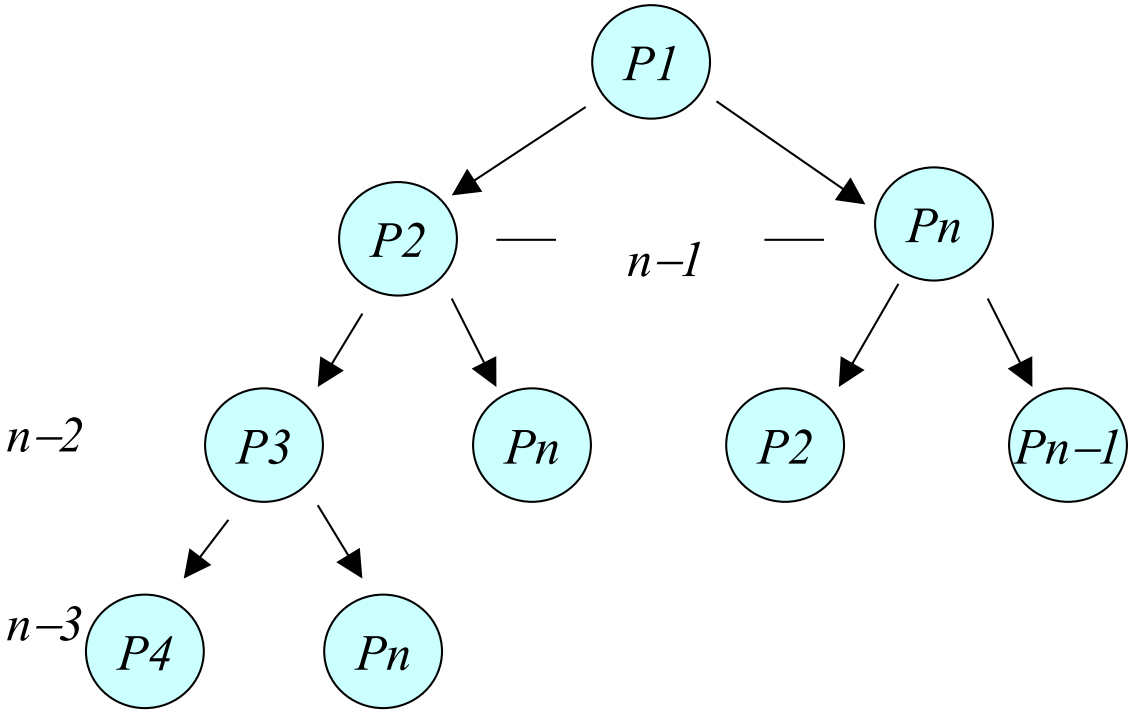
P4: *maj.* (*v*, *u*, *w*, _)

All correct processes decide on the same value.

DIS $N = 7, m = 2$


BG(2)
P1
stage 0
P3,..P7 decide
what P2 said it
heard from P1
BG(1)
P2
P3
P4
P5
P6
P7
stage 1
BG(0)
P4,..P7 decide
what P3 said it
heard from P2
P3
P4
P5
P6
P7
…
P2
…
P6
stage2
…
P4
P5
P6
P7

The Stages in the Lamport, Shostak and Pease Algorithm



stage 0:(n–1) msgs

stage 1:(n–1)(n–2) msgs

stage2:(n–1)(n–2)(n–3) msgs

⇩

stage *m*

DIS Complexity

Message Complexity :

$$(n-1) + (n-1)(n-2) + (n-1)(n-2)(n-3) + \ldots + (n-1)(n-2)\ldots(n-m-1)$$

$$= O(n^{m+1})$$

Time Complexity :

$m+1$ rounds

Dealing with Byzantine failures is expensive!