Distributed Systems

CS425/ECE428

03/04/2020

Logistics

- HW3
  - Released on Monday.
  - You should be able to solve it completely after today's class.
- MP1
  - Due date extended to Monday, March 9th, 11:59pm.
- MP2
  - Will be released on Monday, March 9th (and not this Friday).

Recap: Leader Election

- In a group of processes, elect a *Leader* to undertake special tasks
  - *Let everyone know* in the group about this Leader.
- Safety condition:
  - During the run of an election, a correct process has either not yet elected a leader, or has elected process with best attributes.
- Liveness condition:
  - Election run terminates and each process eventually elects someone.
- Two classical algorithms:
  - Ring-based algorithm
  - Bully algorithm
- Difficulty of ensure both safety and liveness in an asynchronous system under failures.
  - Related to **consensus**!

Agenda for the next 2-3 weeks

- **Consensus**
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *Impossibility of Distributed Consensus with One Faulty Process, Fischer-Lynch-Paterson (FLP), 1985*
  - A good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus
    - Blockchains
    - Raft (log-based consensus)

Agenda for this week

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *Impossibility of Distributed Consensus with One Faulty Process, Fischer-Lynch-Paterson (FLP), 1985*
  - A good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus
    - Blockchains
    - Raft (log-based consensus)

Today's agenda

- **Consensus**
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *Impossibility of Distributed Consensus with One Faulty Process, Fischer-Lynch-Paterson (FLP), 1985*
  - A good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus
    - Blockchains
    - Raft (log-based consensus)

Consensus

- **Each process proposes a value.**
- **All processes must agree on one of the proposed values.**
- Examples:
  - The generals must agree on the time of attack.
  - An object replicated across multiple servers in a distributed data store.
    - All servers must agree on the current version of the object.
  - Transaction processing on replicated servers
    - Must agree on the order in which updates are applied to an object.
  - …..

Consensus

- **Each process proposes a value.**
- **All processes must agree on one of the proposed values.**
- The final value can be decided based on any criteria:
  - Pick minimum of all proposed values.
  - Pick maximum of all proposed values.
  - Pick the majority (with some deterministic tie-breaking rule).
  - Pick the value proposed by the *leader.*
    - *All processes must agree on who the leader is.*
  - If reliable total-order can be achieved, pick the proposed value that gets delivered first.
    - *All process must agree on the total order.*
  - ……

Consensus Problem

- System of N processes (P_1, P_2, …..., P_n)
- Each process P_i:
  - begins in an *undecided* state.
  - proposes value $\mathbf{v_i}$.
  - at some point during the run of a consensus algorithm, sets a decision variable $\mathbf{d_i}$ and enters the *decided* state.

Required Properties

- **Termination:** Eventually each process sets its decision variable.

- **Agreement:** The decision value of all correct processes is the same.
  - If P_i and P_j are correct and have entered the *decided* state, then $\mathbf{d_i = d_j}$.

- **Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

Definition of integrity differs across sources (lack of consensus!)

Required Properties

- **Termination:** Eventually each process sets its decision variable.
- **Agreement:** The decision value of all correct processes is the same.
  - If P_i and P_j are correct and have entered the *decided* state, then $\mathbf{d_i = d_j}$.
- **Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

Which of these properties is liveness and which is safety?

Required Properties

- **Termination:** Eventually each process sets its decision variable.
  - *Liveness*
- **Agreement:** The decision value of all correct processes is the same.
  - If P_i and P_j are correct and have entered the *decided* state, then $\mathbf{d_i = d_j}$.
  - *Safety*
- **Integrity:** If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value.

How do we agree on a value?

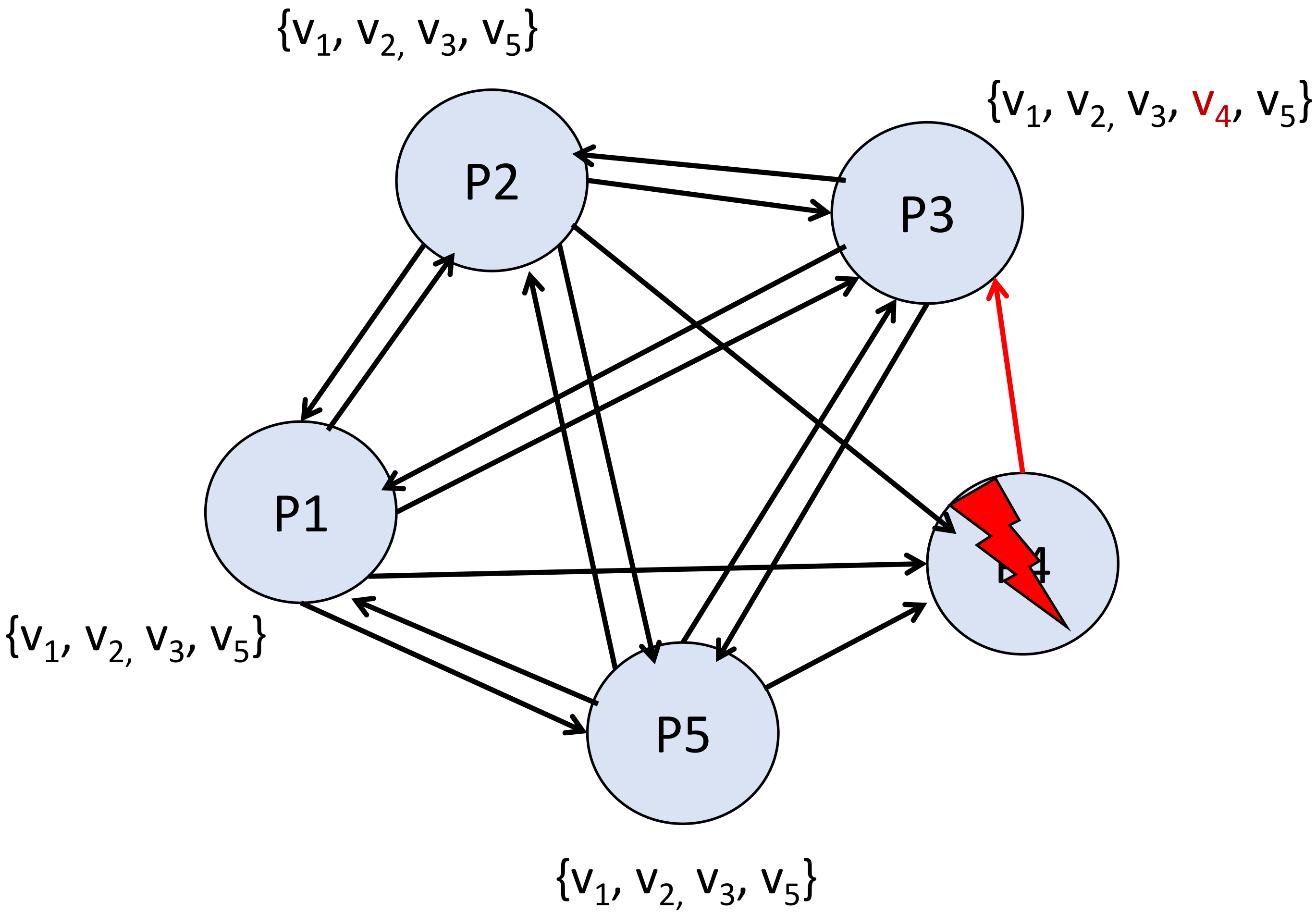
- Ring-based leader election
  - Send proposed value along with *elected* message.
  - Turnaround time: 3NT worst case and 2NT best case (without failures).
    - T is the time taken to transmit a message on a channel.
  - O(Nft) if up to f processes fail during the election run.
  - Can we do better?
- Bully algorithm
  - Send proposed value along with the *coordinator* message.
  - Turnaround time: 4T in the worst case without failures.
  - More than 2fT if up to f processes fail during the election run.

What's the best we can do?

Consider the simplest algorithm

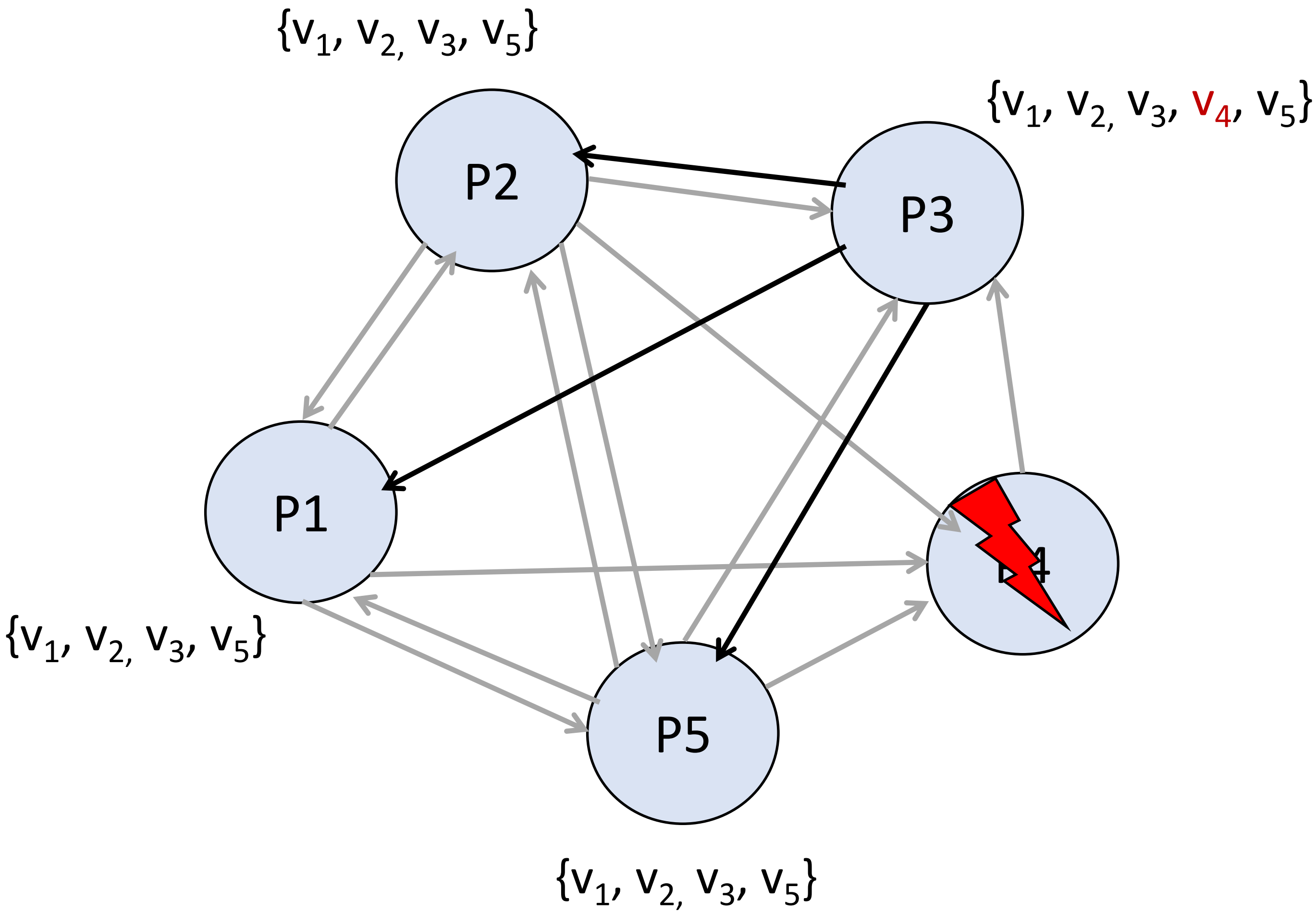
- Let's assume the system is synchronous.
- Use a simple B-multicast:
    - All processes B-multicast their proposed value to all other processes.
    - Upon receiving all proposed values, pick the minimum.
- Time taken under no failures?
    - One message transmission time (T)
- What can go wrong?
    - If we consider process failures, is a simple B-multicast enough?

B-multicast is not enough for this



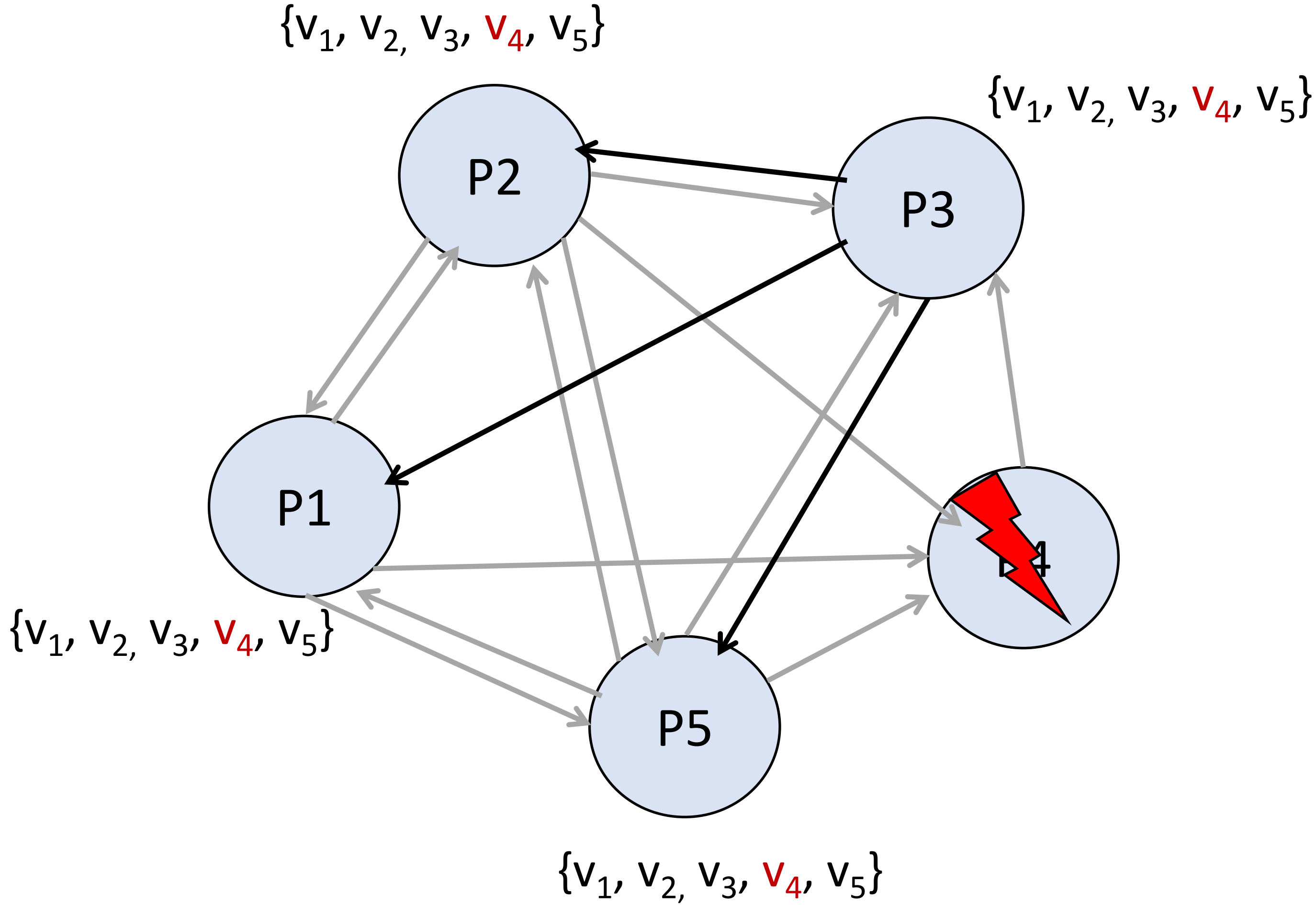
Need R-multicast

B-multicast is not enough for this



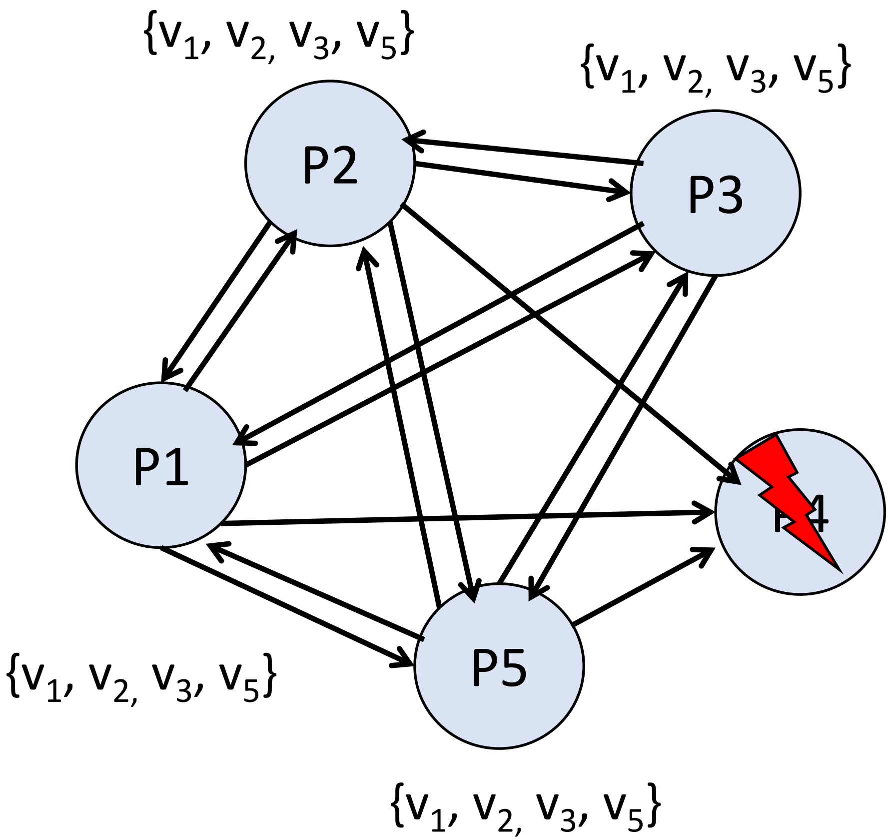
Need R-multicast

B-multicast is not enough for this



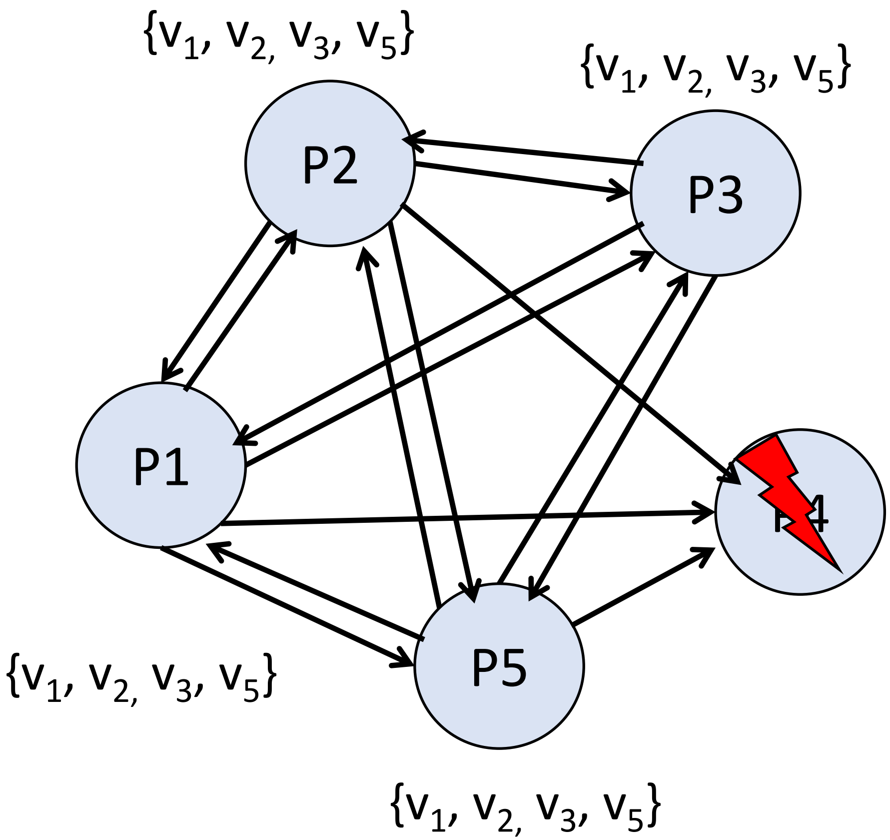
Need R-multicast

Handling failures



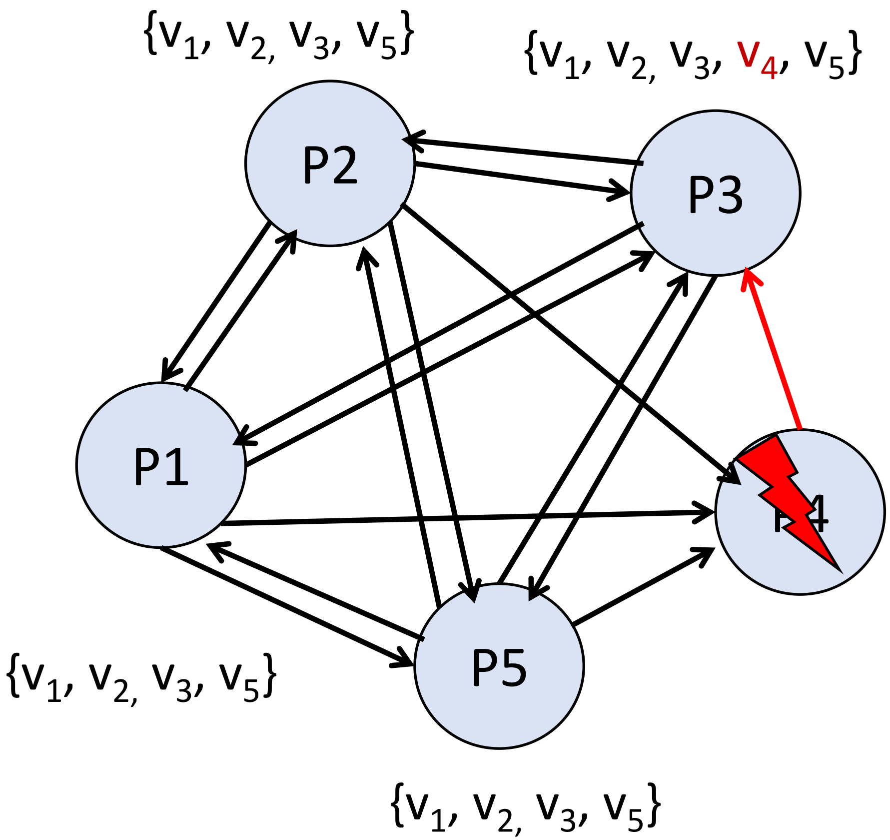
- P4 fails before sending v_4 to anyone.
- What should other processes do?
- Detect failure. *Timeout!*
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?

Handling failures



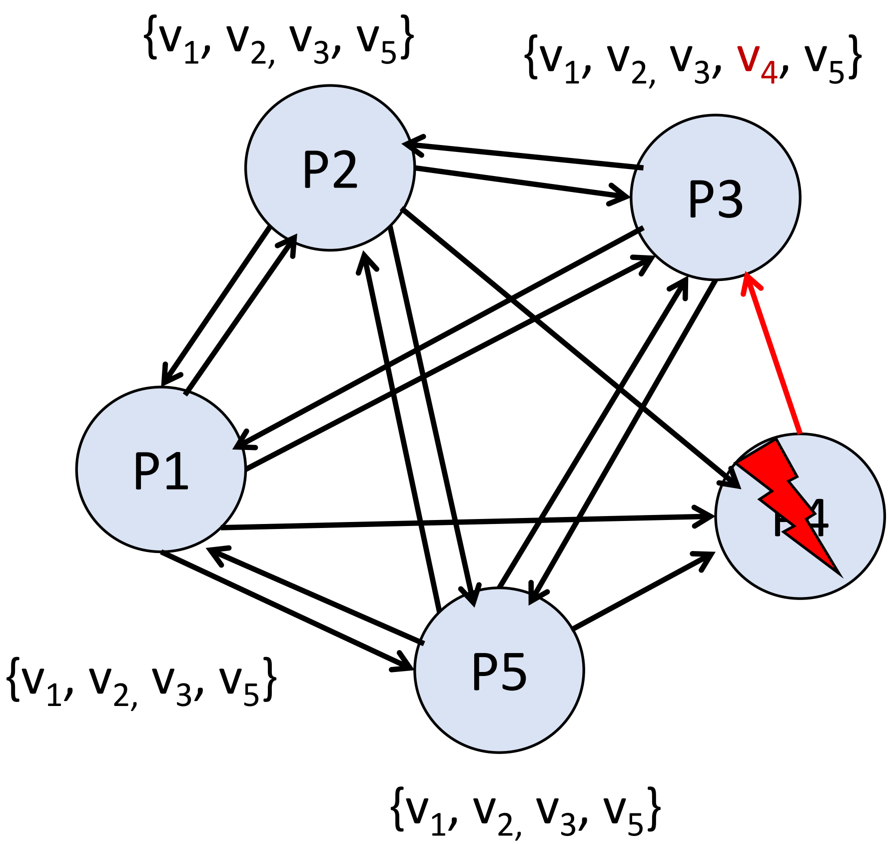
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- Option 1: $\epsilon + T$
  - Pi waits for ($\epsilon + T$) time units after sending its proposal at time 's'.
  - Any other process must have sent proposed value before s + ϵ.
  - The proposed value should have reached Pi by (s + ϵ + T).
  - *Will this work?*

Handling failures



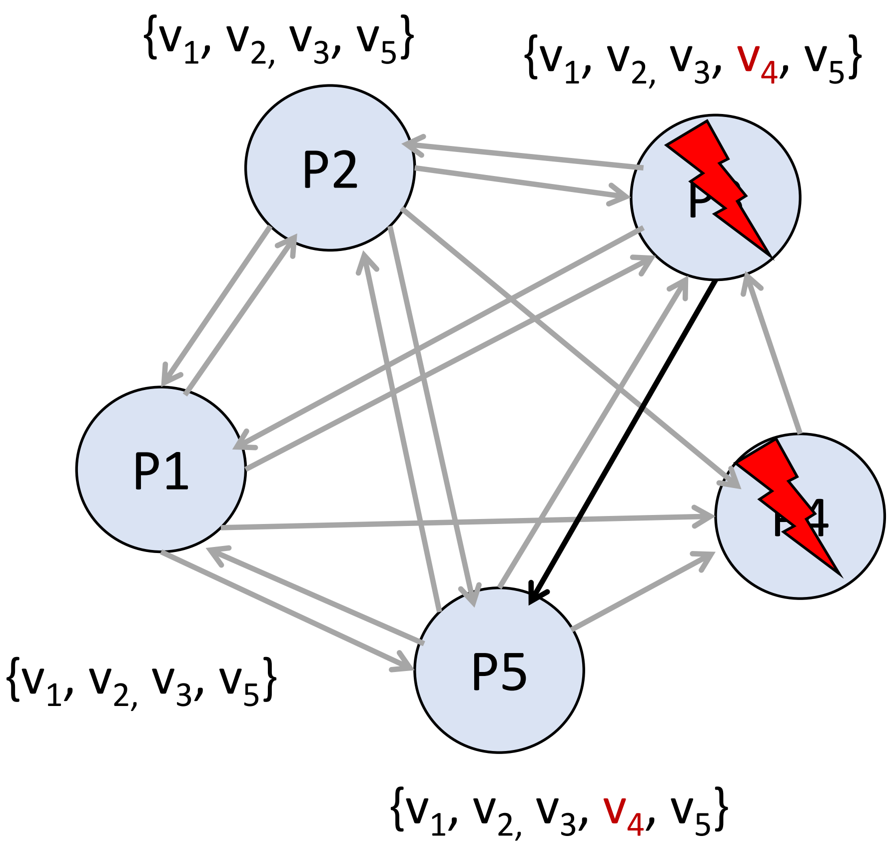
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about $\epsilon + T$?
  - Local time at a process Pi.
  - Pj must have sent proposed value before time s + ϵ.
  - The proposed value should have reached Pi by (s + $\epsilon + T$).
  - *Will this work?*

Handling failures



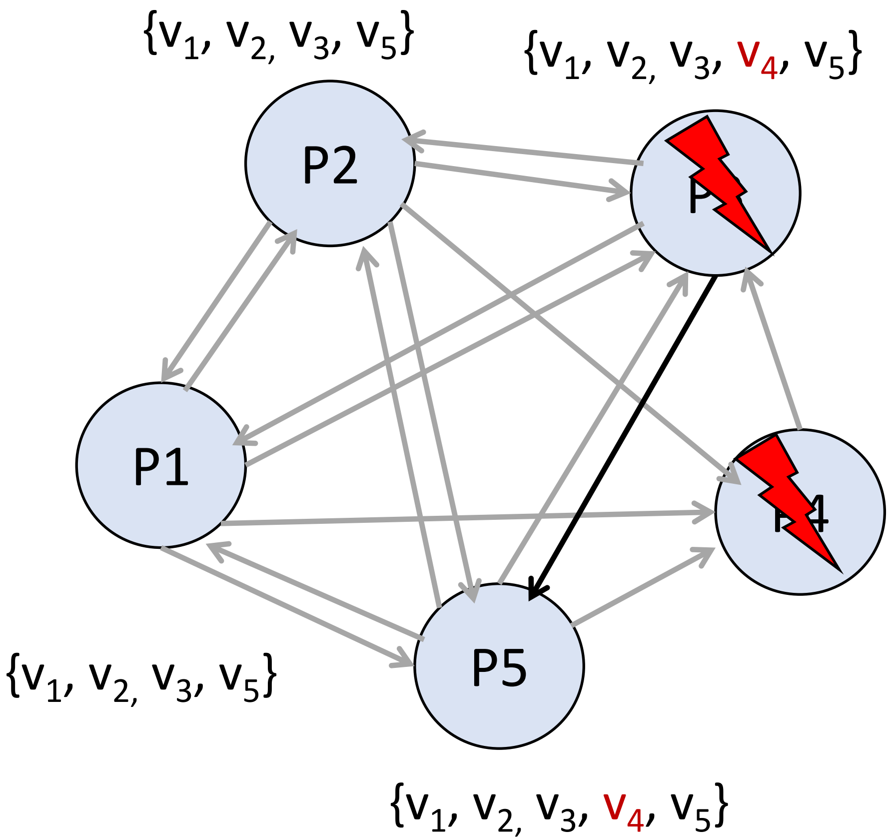
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about $\epsilon + 2*T$?
  - *Will this work?*

Handling failures



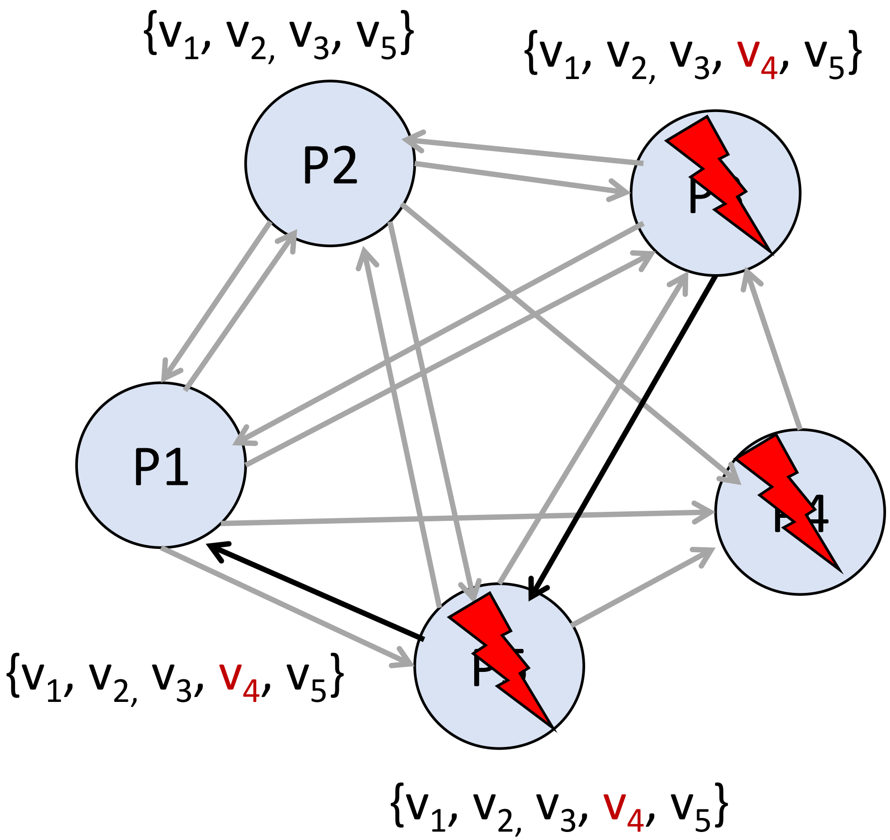
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about $\epsilon +\ 2*T$?
  - *Will this work?*

Handling failures



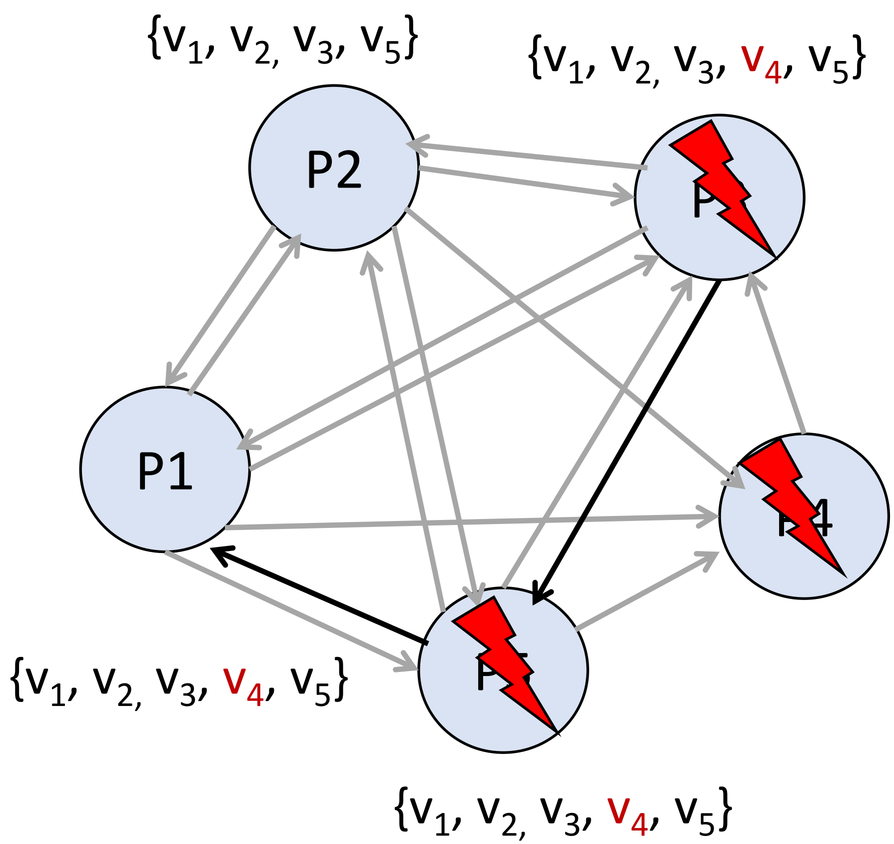
- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about $\epsilon +\ 3*T$?
    - *Will this work?*

Handling failures



- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- How about $\epsilon +$ 3*T?
  - *Will this work?*

Handling failures



- Assume proposals are sent at time 's'.
- Worst-case skew is ϵ.
- Maximum message transfer time (including local processing) is T.
- What should the timeout value be?
- Timeout = $\epsilon + (f+1)*T$ for up to f failed process.

Also holds for R-multicast from a single sender.

Round-based algorithm

- For a system with at most f processes crashing
  - All processes are *synchronized* and operate in "rounds" of time.
    - One round of time is equivalent to $\epsilon + T$ units.
    - At each process, the i^{th} round
      - starts at local time $s + (i - 1)*(\epsilon + T)$
      - ends at local time $s + i*(\epsilon + T)$
    - The start or end time of a round in two different processes differs by at most ϵ.
  - The algorithm proceeds in f+1 rounds.
  - Assume communication channels are reliable.

Round-based algorithm

$Values^r_i$: the set of proposed values known to P_i at the beginning of round r.

Initially $Values^1_i = \{v_i\}$

for round = 1 to f+1 do

B-multicast ($Values^r_i - Values^{r-1}_i$)

// iterate through processes, send each a message

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

wait until one round of time expires.

for each v_j received in this round

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

end

end

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

Why does this 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 last-but-one round must have crashed; otherwise, both P_k and P_j should have received v.
  - → Implies at least one (unique) crash in each of the preceding rounds.
  - →This means a total of f+1 crashes, contradicts our assumption of up to f crashes.

Consensus in synchronous systems

Dolev and Strong proved that for a system with up to *f* failures (or faulty processes), at least $f+1$ rounds of information exchange is required to reach an agreement.

What about asynchronous systems?

- Using time-based "rounds" or timeouts may not work.
- Cannot guarantee both completeness and accuracy.
- Cannot differentiate between an extremely slow process and a failed process.
- Key intuition behind the famous FLP result on the impossibility of consensus in asynchronous systems.
  - Stopped many distributed system designers dead in their tracks.
  - A lot of claims of "reliability" vanished overnight.
  - We will discuss the detailed proof in Friday's class.

What about asynchronous systems?

- We cannot "solve" consensus in asynchronous systems.
    - We cannot meet both safety and liveness requirements.
    - Maybe it is ok to guarantee just one requirement.
- Option 1:
    - Let's set super conservative timeout for a terminating algorithm.
    - Safety violated if a process (or the network) is very, very slow.
- Option 2:
    - Let's focus on guaranteeing *safety* under all possible scenarios.
    - If the real situation is not too dire, hopefully the algorithm will terminate.

Paxos Consensus Algorithm

- Paxos algorithm for consensus in asynchronous systems.
  - Most popular consensus-algorithm.
  - A lot of systems use it
    - Zookeeper (Yahoo!), Google Chubby, and many other companies.
  - Not guaranteed to terminate, but never violates safety.

Paxos Consensus Algorithm

- *Guess who invented it?*
  - Leslie Lamport!
- Original paper: The Part-time Parliament.
  - Used analogy of a "part-time parliament" on an ancient Greek island of Paxos.
  - No one understood it.
  - The paper was rejected.
- Published "*Paxos made simple*" 10 years later.

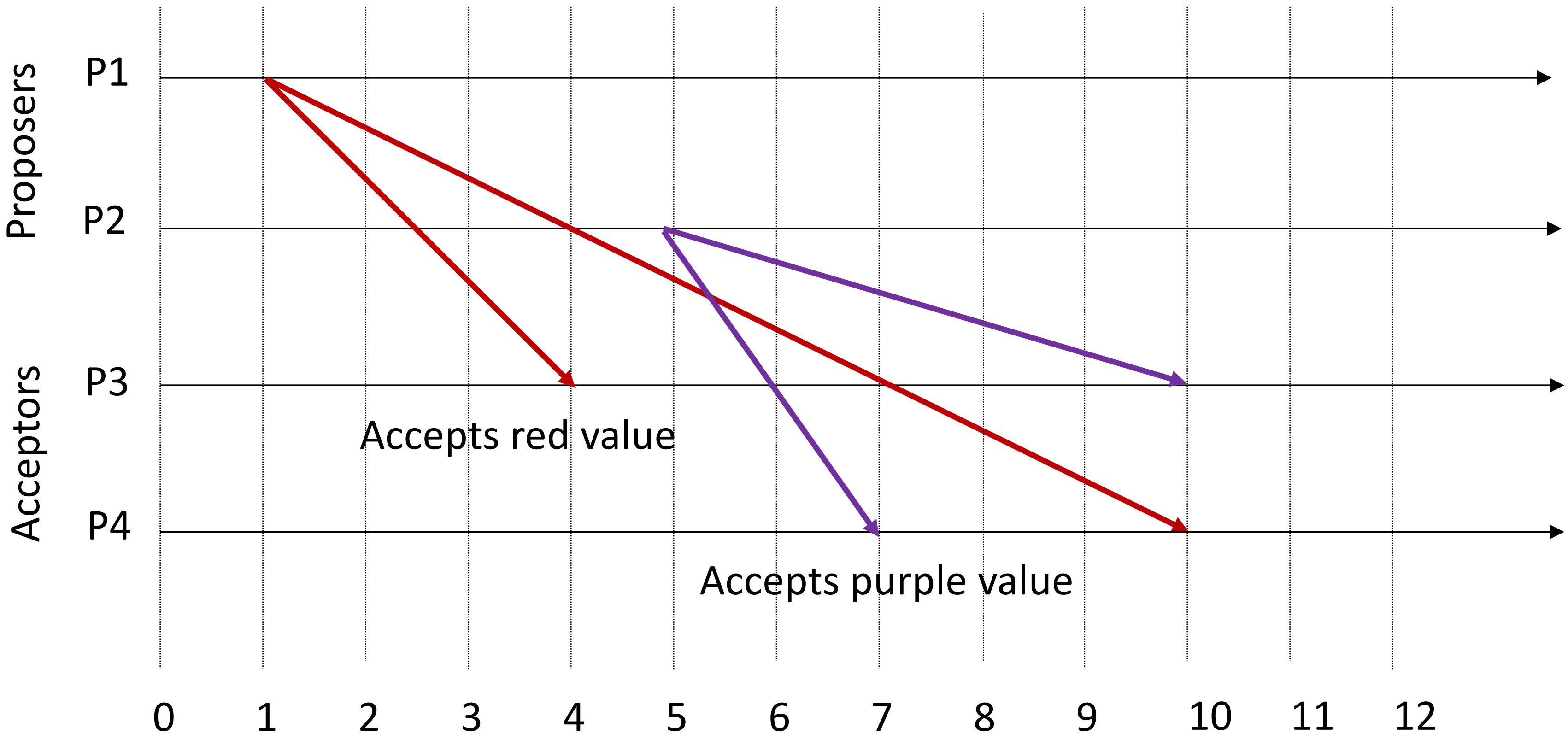
Paxos Algorithm

- Three types of roles:
  - **Proposers:** propose values to *acceptors*.
    - All or subset of processes.
    - Having a *single proposer* (leader) may allow faster termination.
  - **Acceptors:** accept proposed values (under certain conditions).
    - All or subset of processes.
  - **Learners:** learns the value that has been accepted by *majority* of acceptors.
    - All processes.

Paxos Algorithm: Try 1: Single Phase

- A proposer multicasts its proposed value to a large enough set (larger than majority) of acceptors.
- An acceptor accepts the first proposed value it receives.
- If majority of acceptors have accepted the same value v, then v is the decided value.
- *What can go wrong here?*

Paxos Algorithm: Try 1: Single phase

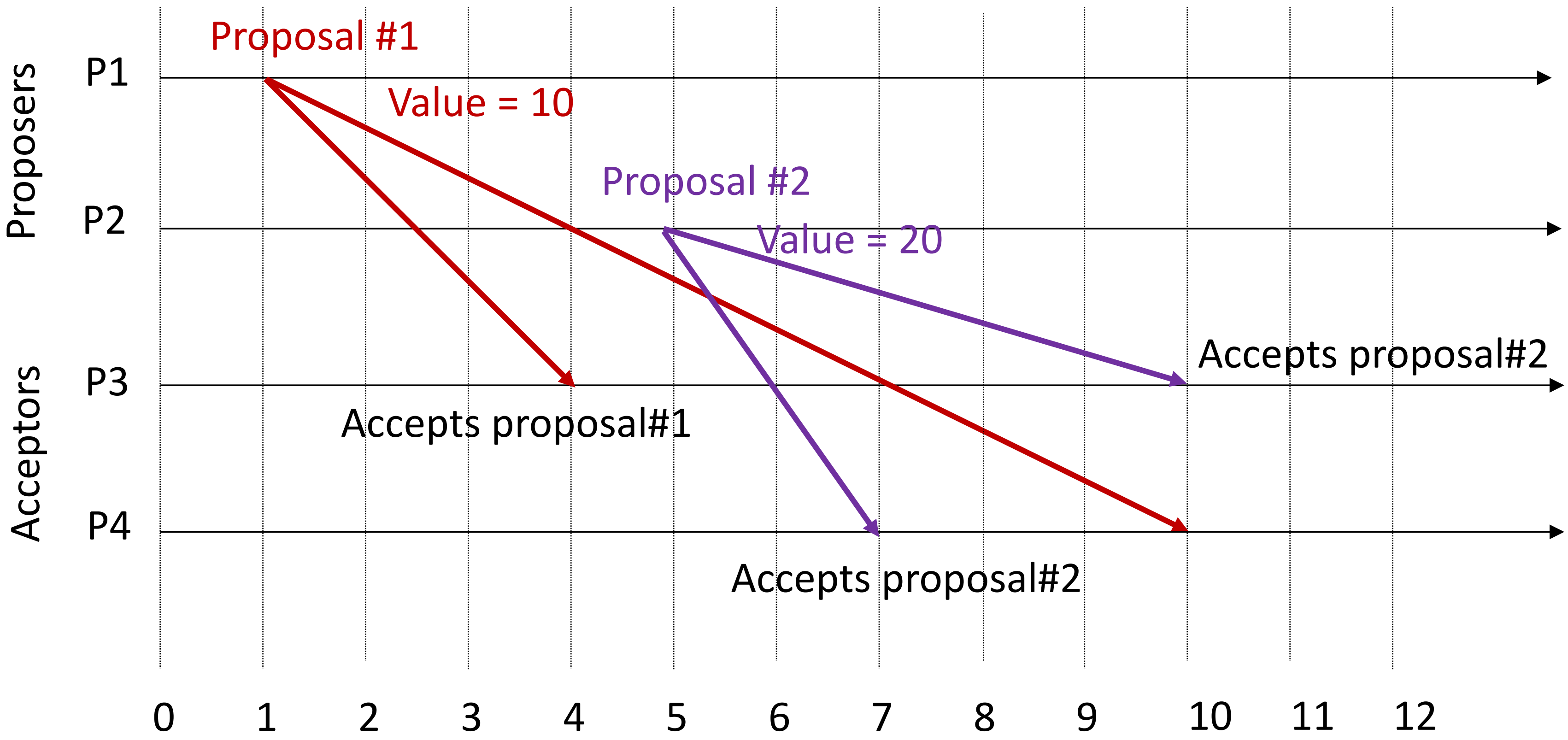


No decision reached!

Paxos Algorithm: Proposal numbers

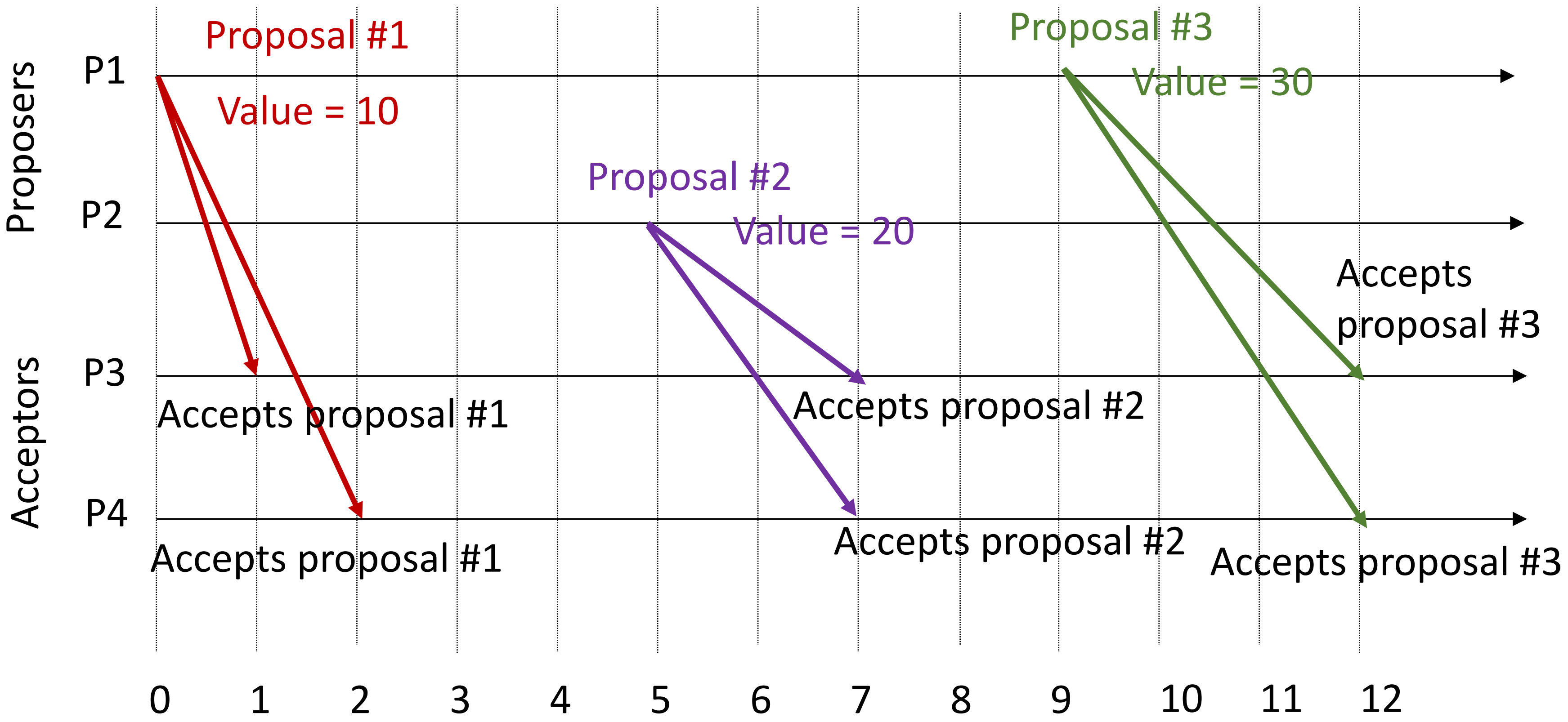
- Allow an acceptor to accept multiple proposals.
  - Accepting is different from *deciding*.
- Distinguish proposals by assigning unique ids (a proposal number) to each proposal.
  - Configure a disjoint set of possible proposal numbers for different processes.
  - Proposal number is different from proposed value!
- A higher number proposal overwrites and pre-empts a lower number proposal.

Paxos Algorithm: Try 2: Proposal #s



What can go wrong here?

Paxos Algorithm: Try 2: Proposal #s

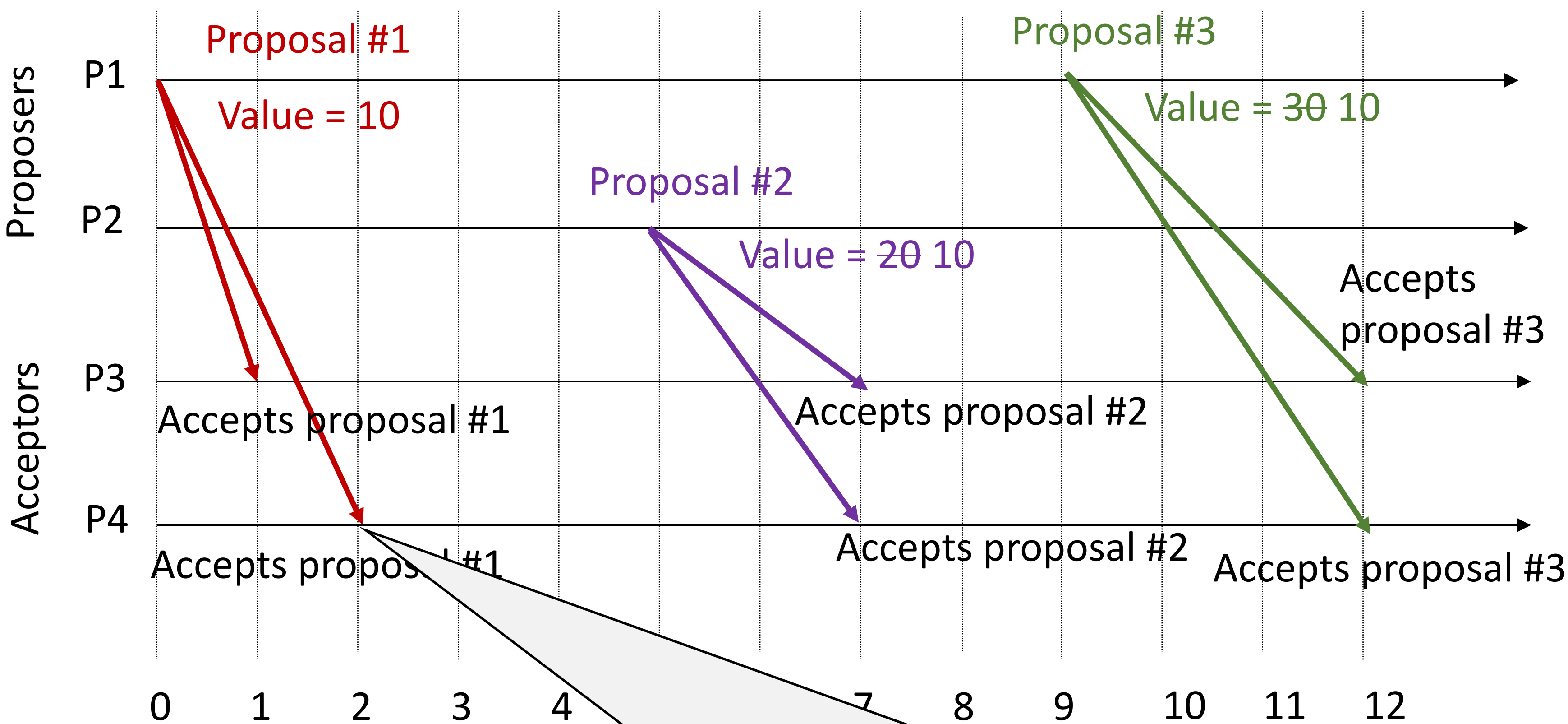


When do we stop and decide on a value?

Paxos Algorithm

- Key condition:
  - When majority of acceptors accept a single proposal with a value v, then that value v becomes the decided value.
    - This is an implicit decision. Learners may not know about it right-away.
  - Any higher-numbered proposal that gets accepted by majority of acceptors after the implicit decision must propose the same decided value.

Paxos Algorithm



Point of no return!

Any proposal accepted by majority of acceptors after this must propose the same value as proposal #1 (i.e. 10).

Paxos Algorithm: Two phases

- **Phase 1:**
  - A proposer selects a proposal number (*n*) and sends a prepare request with *n* to majority of acceptors, requesting:
    - Promise me you will not reply to any other proposal with a lower number.
    - Promise me you not accept any other proposal with a lower number.
  - If an acceptor receives a prepare request for proposal #*n*, and it has not responded to a prepare request with a higher number, it replies back saying:
    - OK! I will make that promise for any request I receive in the future.
    - (If applicable) I have already accepted a value *v* from a proposal with lower number m < n. The proposal has the highest number among the ones I accepted so far.

Paxos Algorithm: Two phases

- **Phase 2:**
  - If a proposer receives an OK response for its prepare request #n from a *majority* of acceptors, then it sends an accept request with a proposed value. What is the proposed value?
    - The value v of the *highest numbered proposal* among the received responses.
    - Any value if no previously accepted value in the received responses.
  - If an acceptor receives an accept request for proposal #*n*, and it has not responded a prepare request with a higher number, it accepts the proposal.

- *What if the proposer does not hear from majority of acceptors?*
  - Wait for some time, and then issue a new request with higher number.

Paxos Algorithm

- When majority of acceptors accept a single proposal with a value *v*, then that value v becomes the *decided* value.
  - Suppose this proposal has a number *m*.
  - By design of the algorithm: *any subsequent proposal with a number n higher than m will propose a value v.*
  - Proof by induction:
    - Induction hypothesis: every proposal with number in [m,…..n-1] proposes value *v*.
    - Consider a set C with majority of acceptors that have accepted m's proposal (and value *v*).
    - Every acceptor in C has accepted a proposal with number in [m,…..n-1] .
      - Every acceptor in C has accepted a proposal with value *v*.
    - Any set consisting of a majority of acceptors has at least one member in C.
      - Proposal #n's prepare request will receive an OK reply with value v.

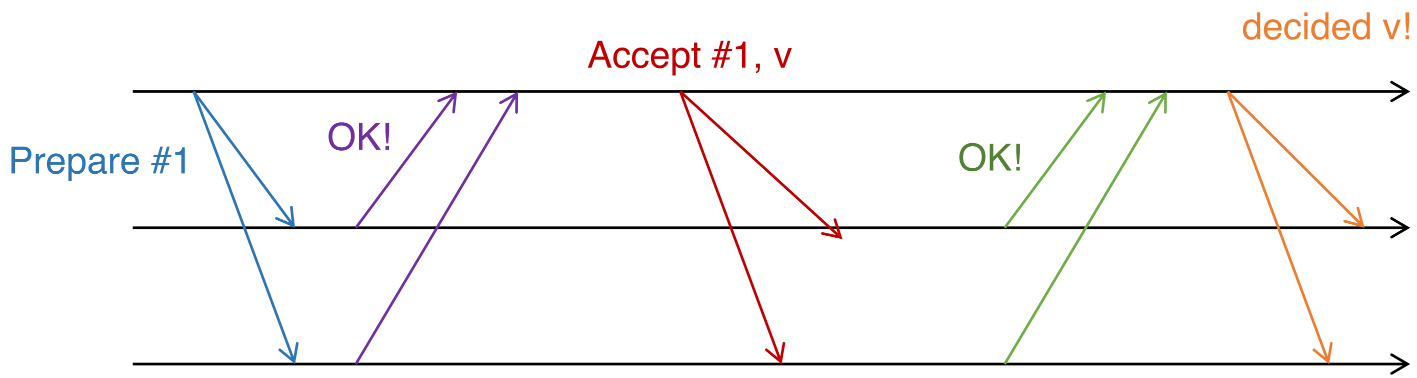
Paxos Algorithm

- When majority of acceptors accept a single proposal with a value *v*, then that value v becomes the *decided* value.

- How do learners learn about it?
  - Every time an acceptor accepts a value, send the value and proposal # to a *distinguished learner*.
  - This *distinguished learner* will check if a decision has been reached and will inform other learners.
  - Use a set of distinguished learners to better handle failures.
  - What happens if a message is lost or all distinguished learners fail?
    - May not know that a decision has been reached.
    - A proposer will issue a new request (and will propose the same value). Acceptors will accept the same value and will notify the learner again.

Paxos Algorithm

- Best strategy: elect a single leader who proposes values.
- Assume this leader is also the distinguished learner.



- What if we have multiple proposers? (leader election is not perfect is asynchronous systems)
  - May have a livelock! Two proposers may keep pre-empting each-other's requests by constantly sending new proposals with higher numbers.
  - Safety is still guaranteed!

Paxos Algorithm

- What if majority of acceptors fail before a value is decided?
  - Algorithm does not terminate.
  - Safety is still guaranteed!

- What if a process fails and recover again?
  - If it is an acceptor, it must remember highest number proposal it has accepted.
    - Acceptors log accepted proposal on the disk.
  - As long as this state can be retrieved after failure and recovery, algorithm works fine and safety is still guaranteed.

- *Exercise: think about what else can go wrong and how would Paxos handle that situation?*

Summary

- Consensus is a fundamental problem in distributed systems.
- Possible to solve consensus in synchronous systems.
  - Algorithm based on time-synchronized rounds.
  - Need at least (f+1) rounds to handle up to f failures.
- Impossible to solve consensus is asynchronous systems.
  - Details in next class!
  - Paxos algorithm:
    - Guarantees safety but not liveness.
    - Hopes to terminate if under good enough conditions.