Consensus Problem & Paxos Protocol

Prof. Christof Fetzer, Ph.D.
TU Dresden

Distributed System

Set of processes that communicate with each other via message

- e.g., sending messages via https. No bound on communication ; no
- processes can fail, e.g., can crash / be slow

Consensus Problem

Example: Consensus

Agreeing on a primary process:

- Set of processes p, q, r, …
- Each process can propose a value, (in this case, the next **primary**)
- All processes need to agree on a common value (e.g., primary)

Consensus Problem

Requirements:

- **Termination**:
  - All **correct** processes will **decide**
- **Agreement**:
  - **All** processes that decide, agree on the same value
- **Validity**:
  - Decision value was **proposed** by one process

Specification

- What if we drop validity requirement?
- What if we drop termination?
- What if we change agreement to:
  - „*All correct processes agree on the same value*“?

More Examples

Why do we need consensus?

Consensus

Agreement on

- **key/value stores**
- ordering of messages
- the members of a set
- state updates (to keep replicas in sync)
- ...

How can one implement consensus?

System Model

Set of N (unique) processes

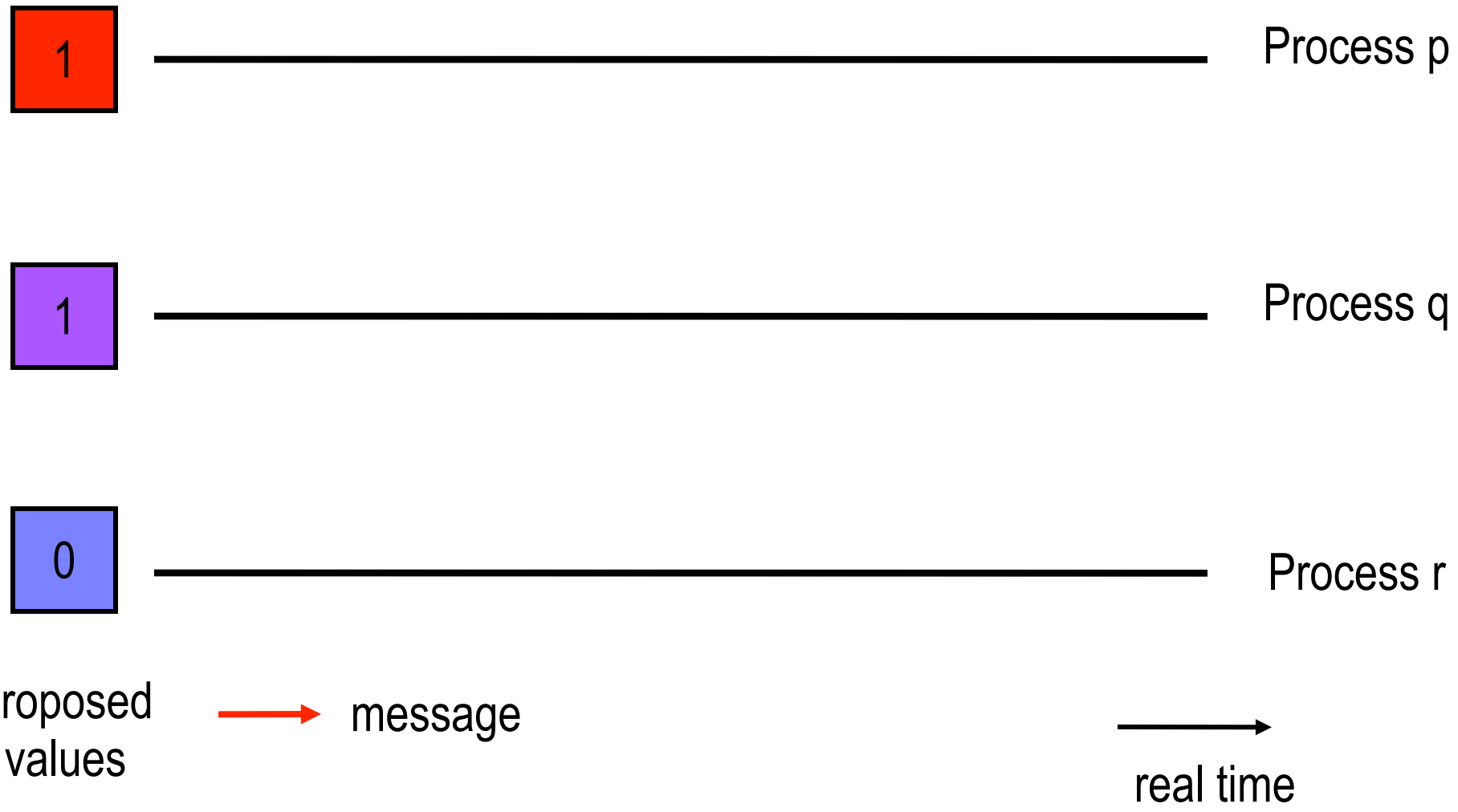
- each executed by some computer
- we know a priori all processes

Processes can exchange messages

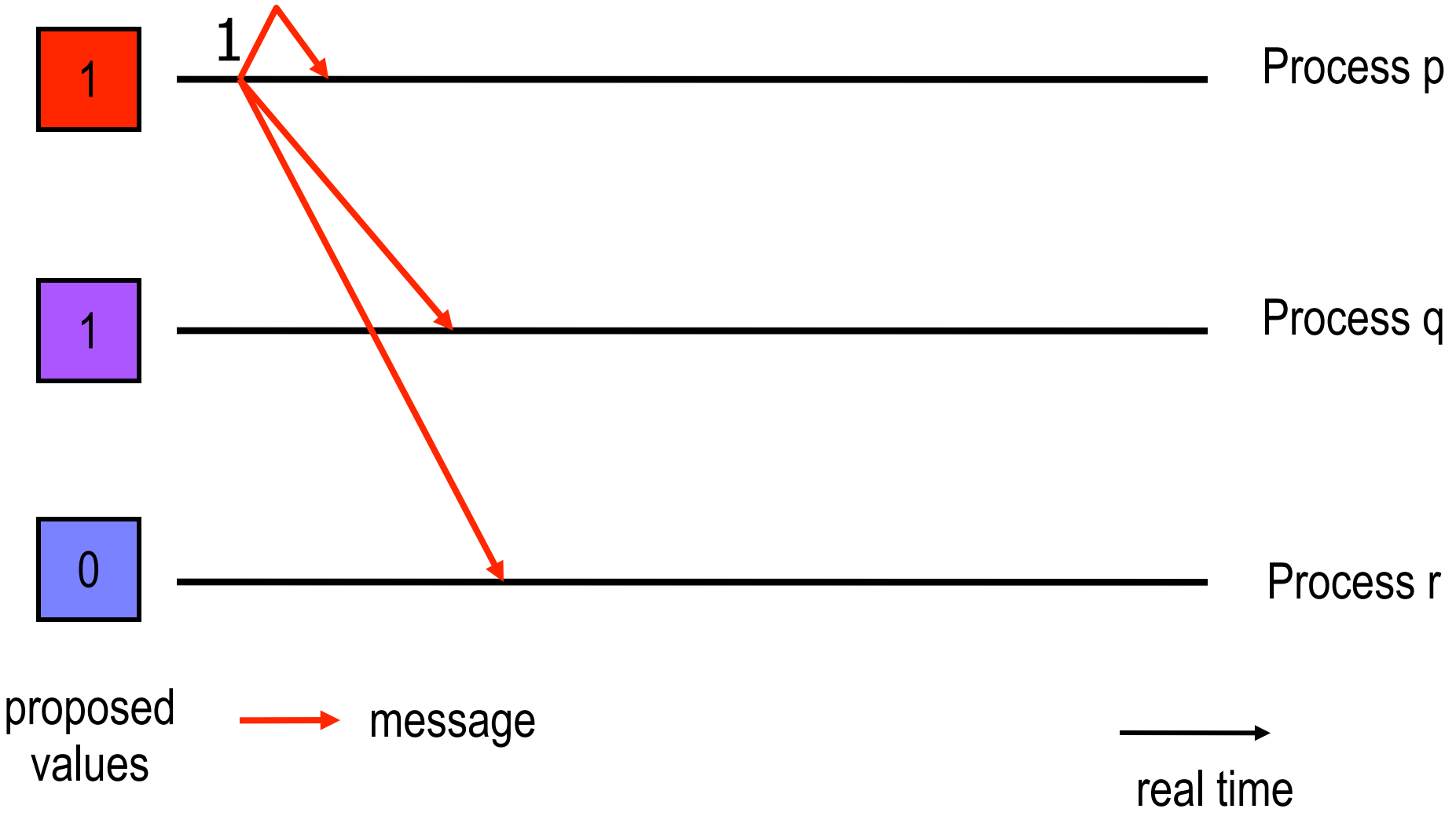
- e.g., can use TCP to exchange info

An execution of a program is called a **run**

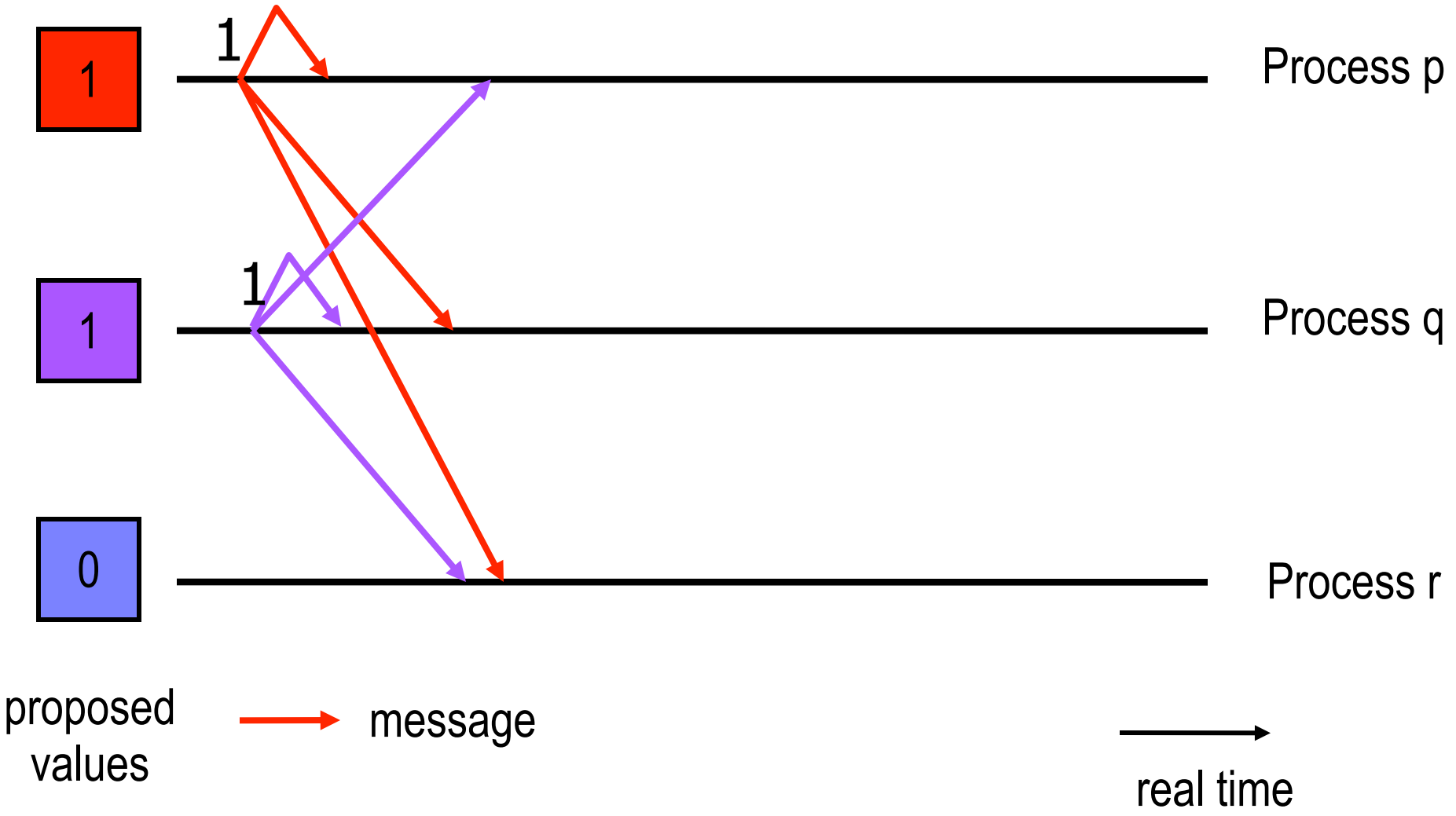
Example run 1: decision = min {values}



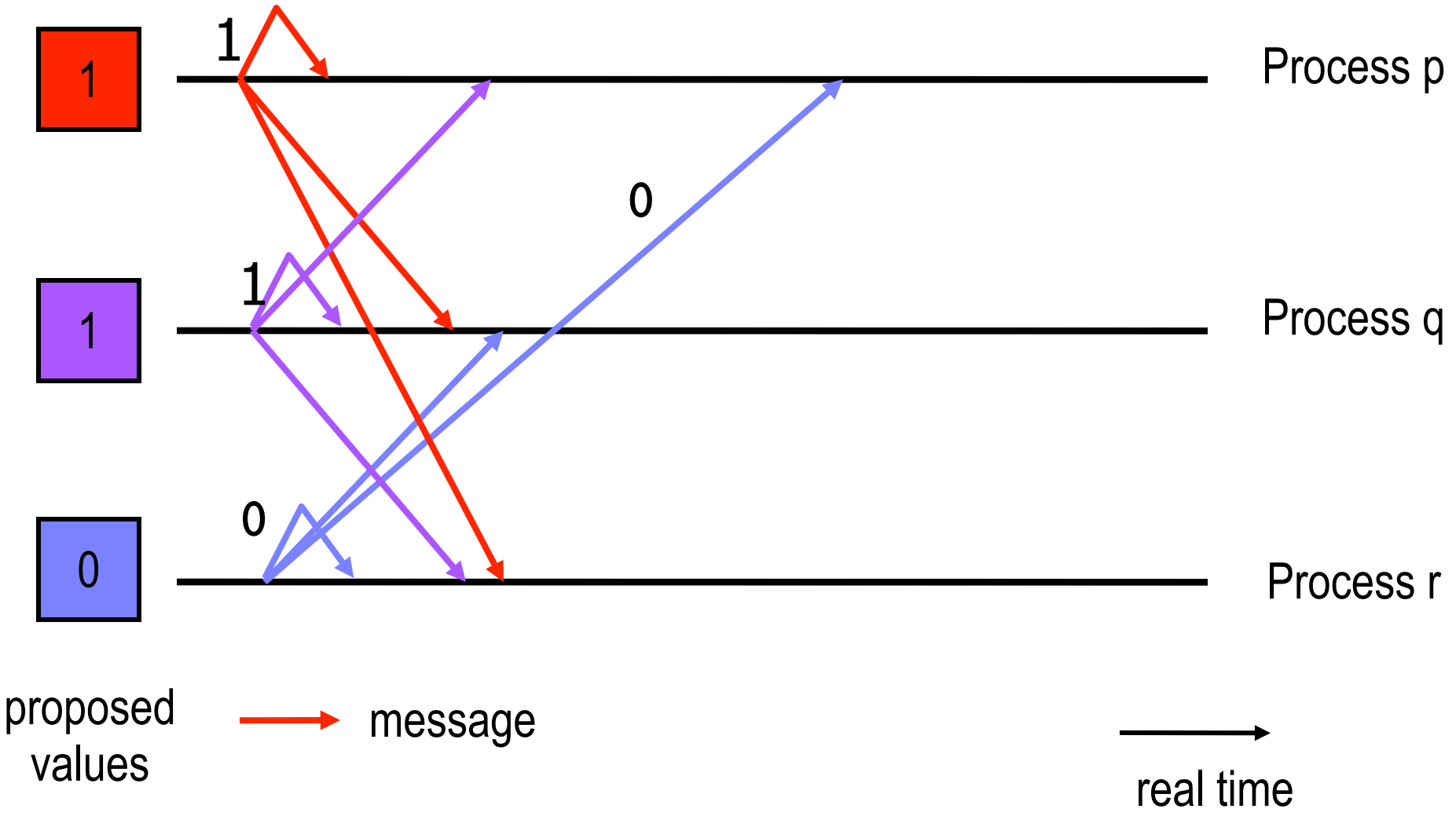
Example run 1: decision = min {values}



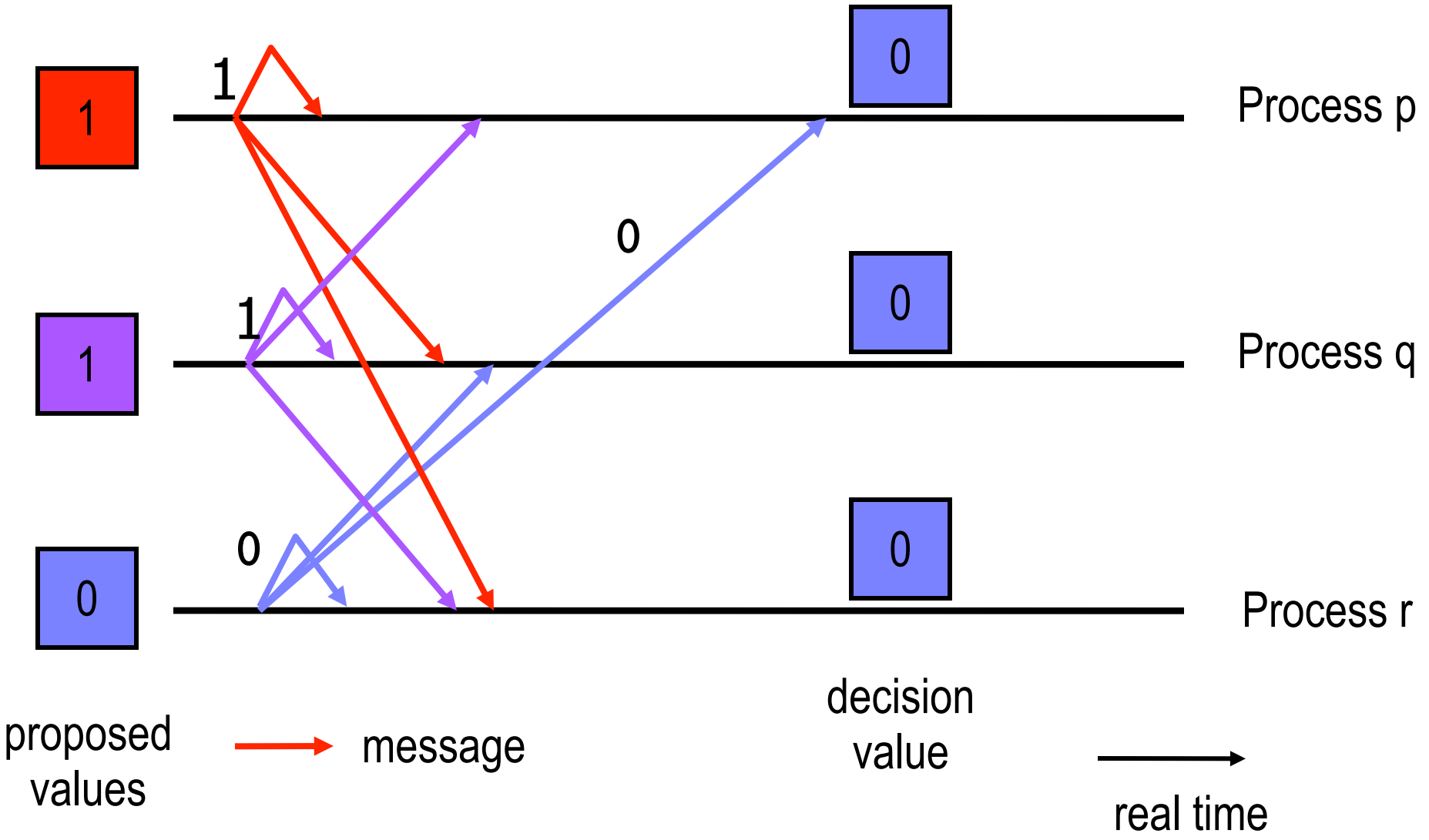
Example run 1: decision = min {values}



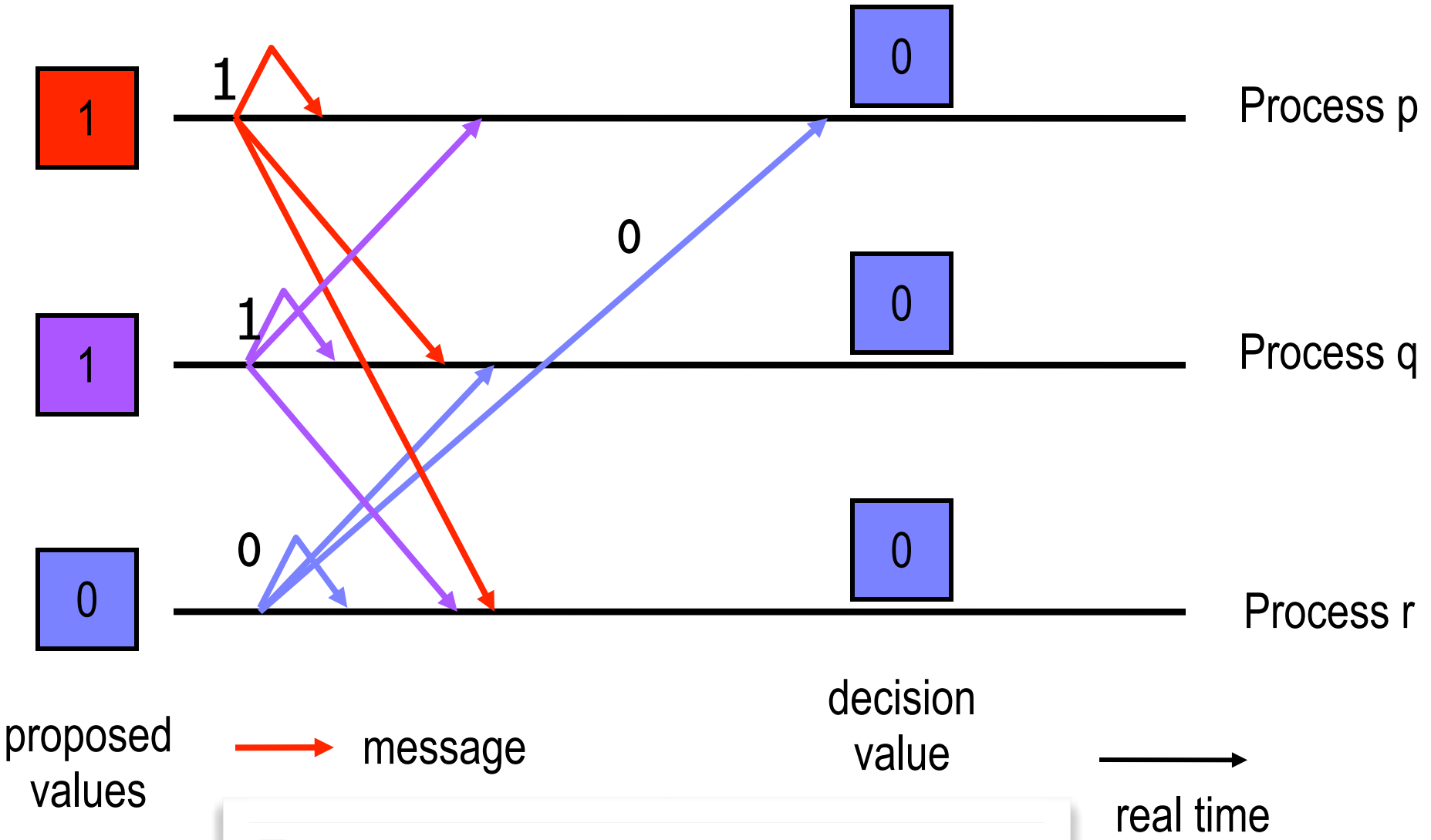
Example run 1: decision = min {values}



Example run 1: decision = min {values}



Example run 1: decision = min {values}



This is not a correct consensus protocol

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(int proposed_value) {
    broadcast(proposed_value);
}

on receive(int value) from p {
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(int prop
    broadcast(proposed
}

on receive(int value) from p {
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

set of all proposed values gotten so far

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(int proposed_value) {
    broadcast(proposed
}

on receive(int value) f
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

number of values received so far

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(int proposed_value) {
    broadcast(proposed_value);
}

on receive(int value) {
    Values.insert(value);
    if (++num_msgs ==
        decide(min(Values));
    }
}
```

called to initiate consensus - broadcasts value to all processes

Protocol

```
Set(int) Values = {}; // proposed values received
int num_msgs = 0;

void propose(int proposed_value) {
    broadcast(proposed_value);
}

on receive(int value) from p {
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

receive a broadcast msg:
add value to set of values

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(in
    broadcast(pro
}

on receive(int value) from p {
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

if we receive messages from all N processes:
decide on minimum value

Protocol

```
Set(int) Values = {};  // proposed values received
int num_msgs = 0;   // number of messages received

void propose(int proposed_value) {
    broadcast(proposed_value);
}

on rece
    Values.insert(value);
    if (++num_msgs == N) {
        decide(min(Values));
    }
}
```

upcall to say that we reached a decision

Broadcast

```
(1..N) Π;  // set of process ids

void broadcast(value) {
    foreach p in Π {
        send(value) to p;
    }
}
```

Broadcast

```
(1..N) Π;  // set of process ids

void broadcast(value) {
    foreach p in Π {
        send(value) to p;
    }
}
```

set of all process ids

Broadcast

```
(1..N) Π;  // set of process ids

void broadcast(value) {
    foreach p in Π {
        send(value) to p;
    }
}
```

broadcast value to all processes in set

Broadcast

```
(1..N) Π;  // set of process ids

void broadcast(value) {
    foreach p in Π {
        send(value) to p;
    }
}
```

iterate over all processes and send point-to-point message

Note

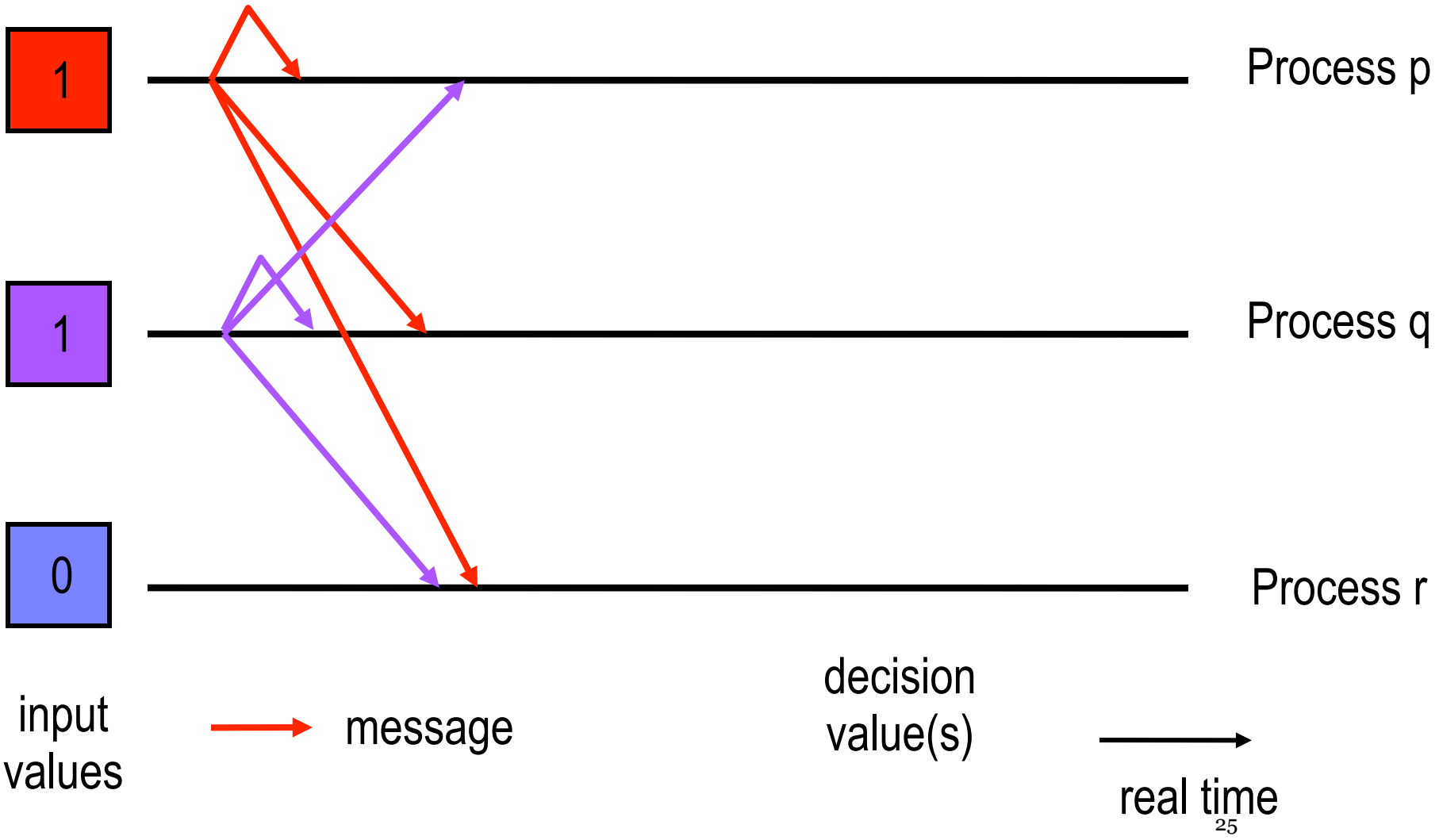
```
(1..N) Π;  // set of process ids

void broadcast(value) {
    foreach p in Π {
        send(value) to p;
    }
```

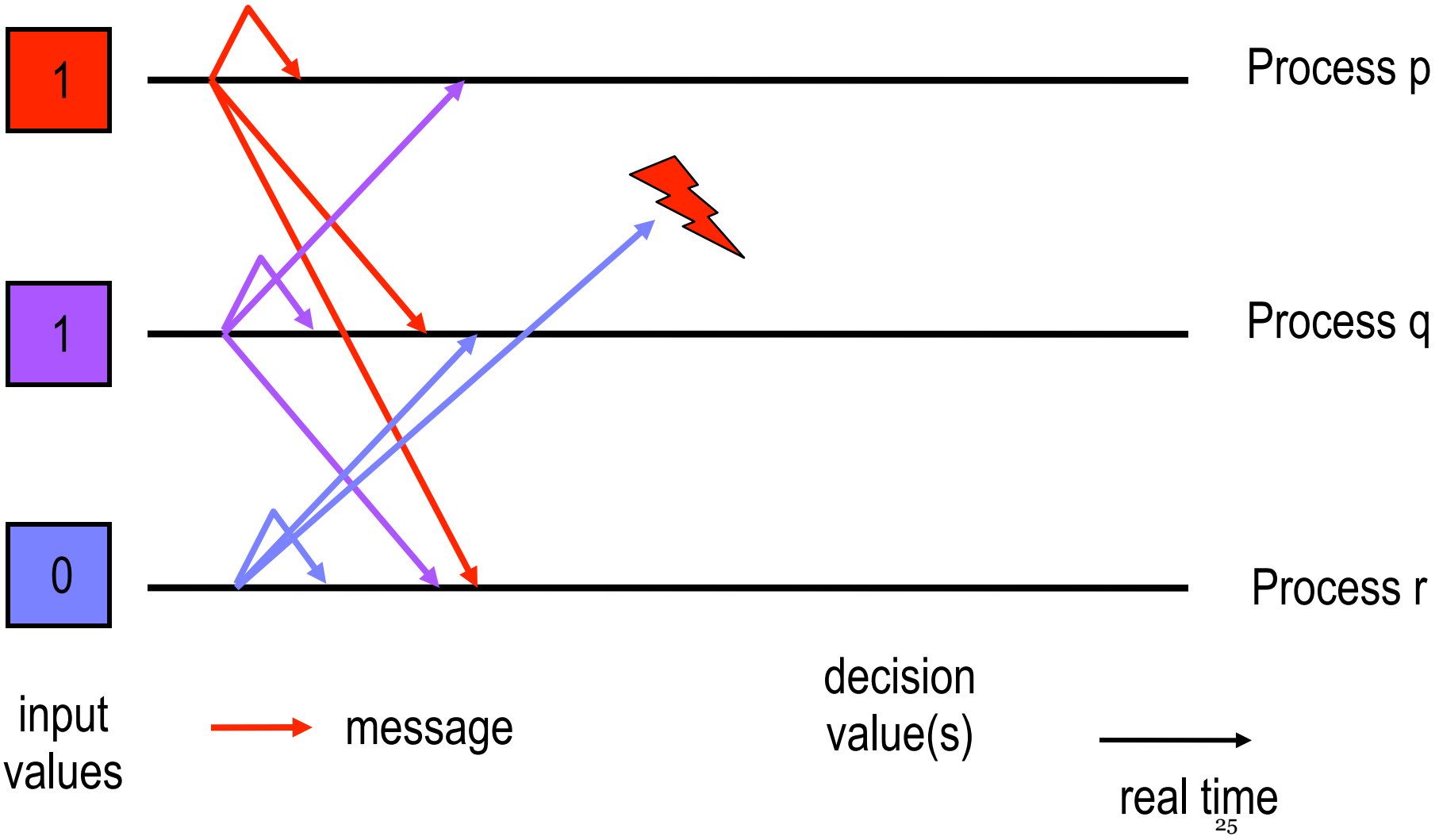
Broadcast is not **atomic**:
- crash of process could result in only a subset of processes receiving the value

What about failures?

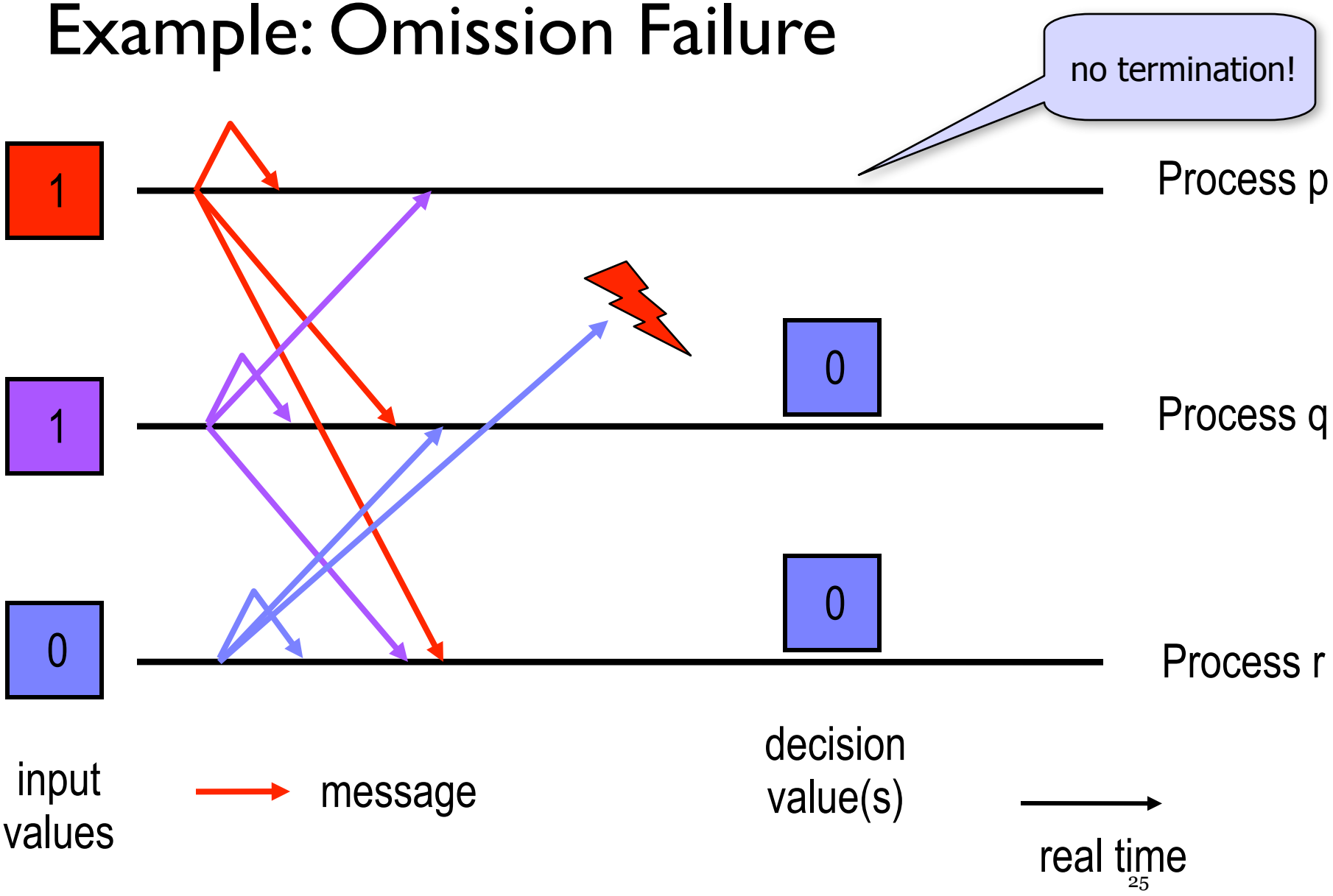
Example: Omission Failure



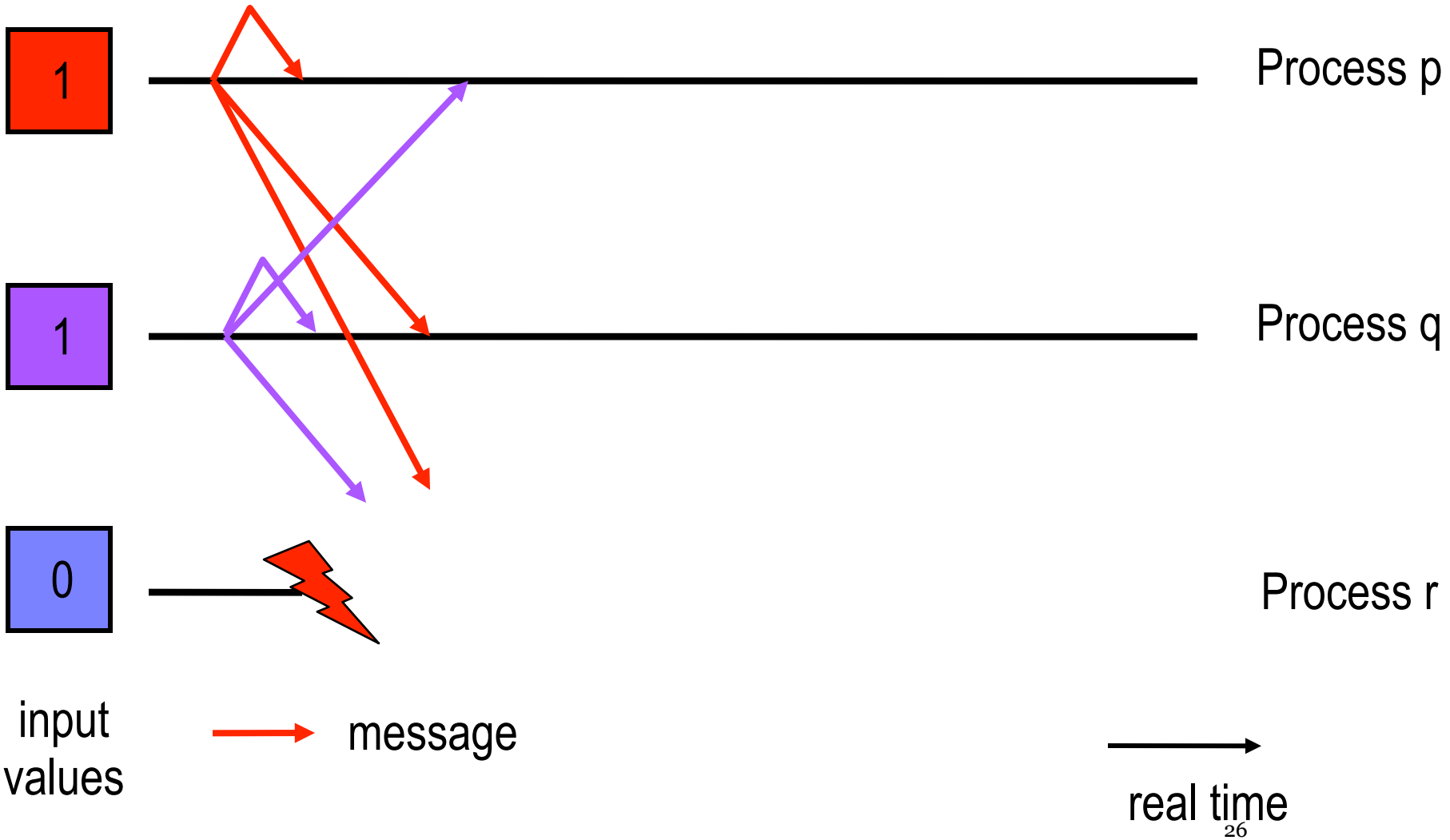
Example: Omission Failure



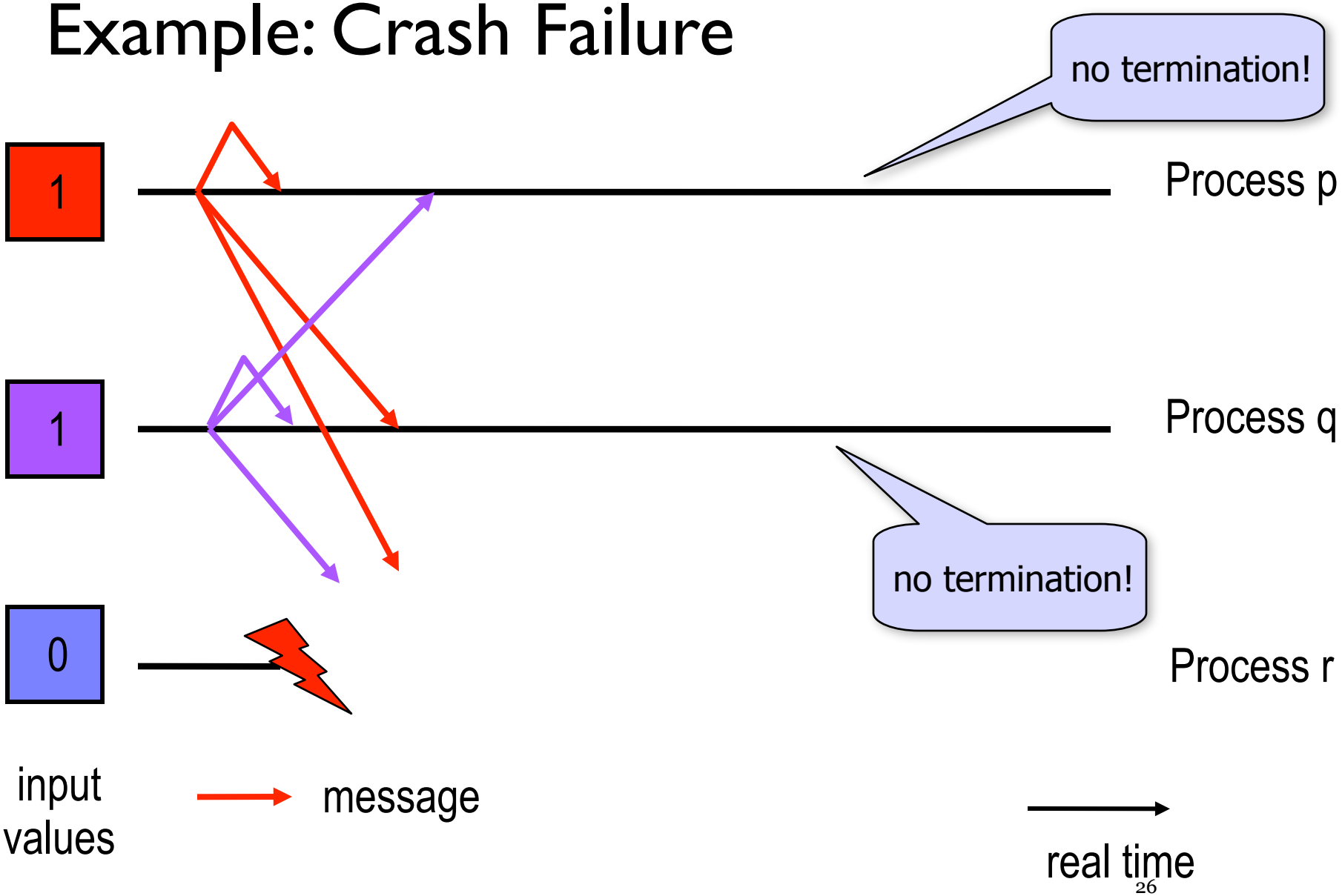
Example: Omission Failure



Example: Crash Failure

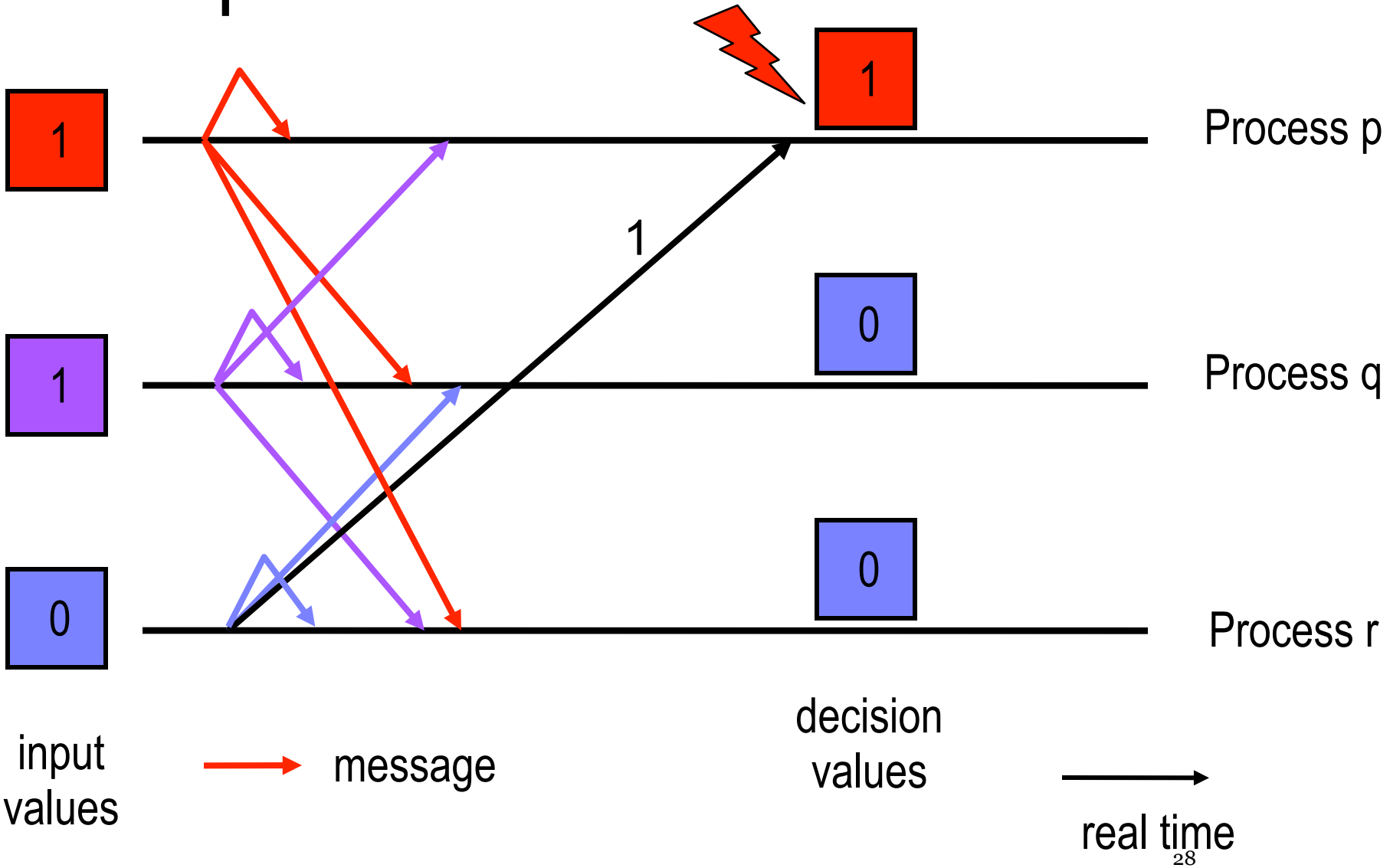


Example: Crash Failure

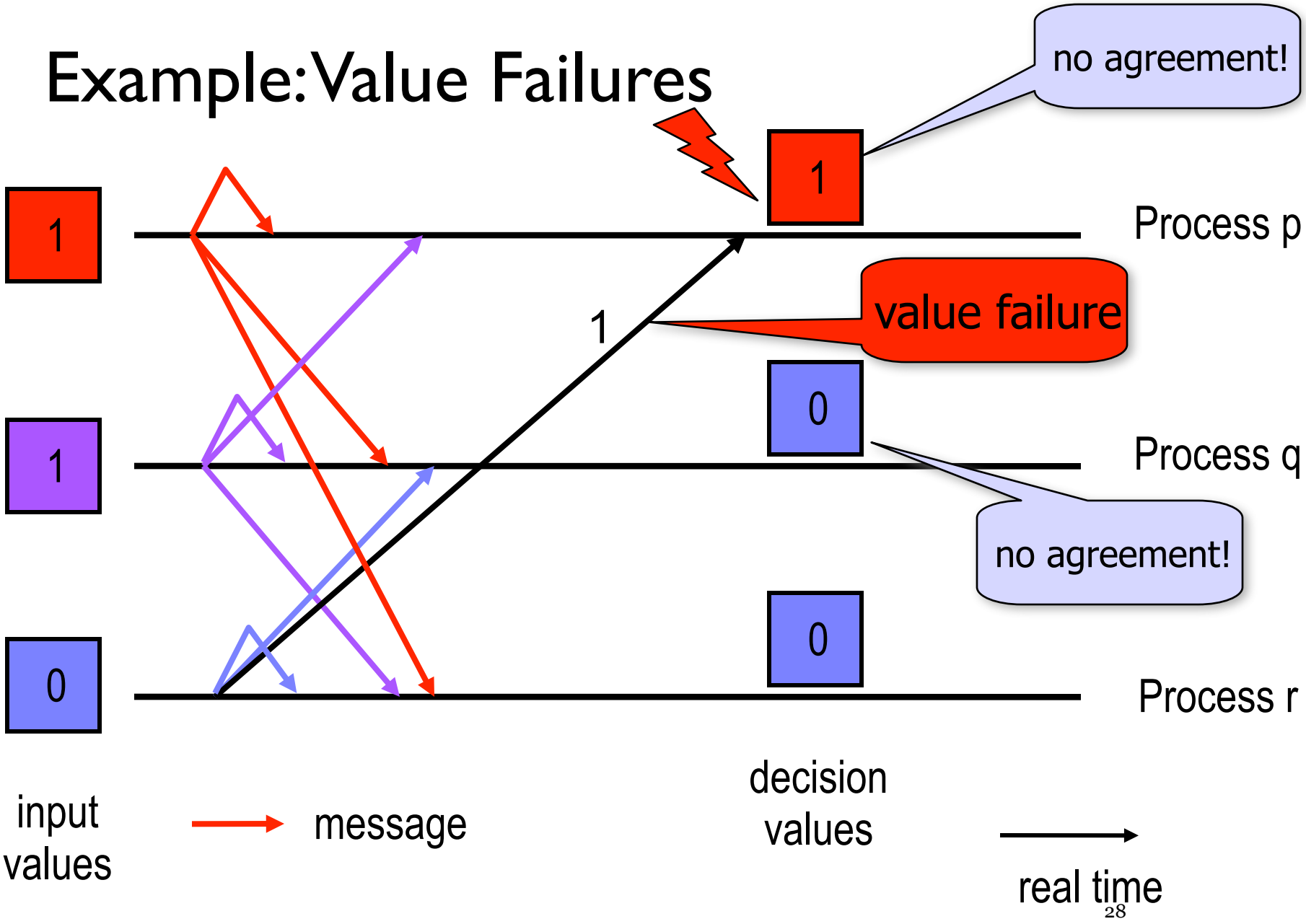


What about other failures?

Example: Value Failures



Example: Value Failures



Simplify The Problem!

Failure Assumption

Processes can only fail by crashing!

Crash failure:

- process stops executing the protocol

For now, simplify:

- we assume crashes failures only
- i.e., **no other failures:** no value failures, no message omission, …!

Is the problem
now easier to solve?

Bivalent Configuration

Bivalent Configuration

- Assume proposed values are in {0,1}

Bivalent Configuration

- Assume proposed values are in {0,1}
- Definition: a **configuration** is a
  - N-tuple of proposed values

Bivalent Configuration

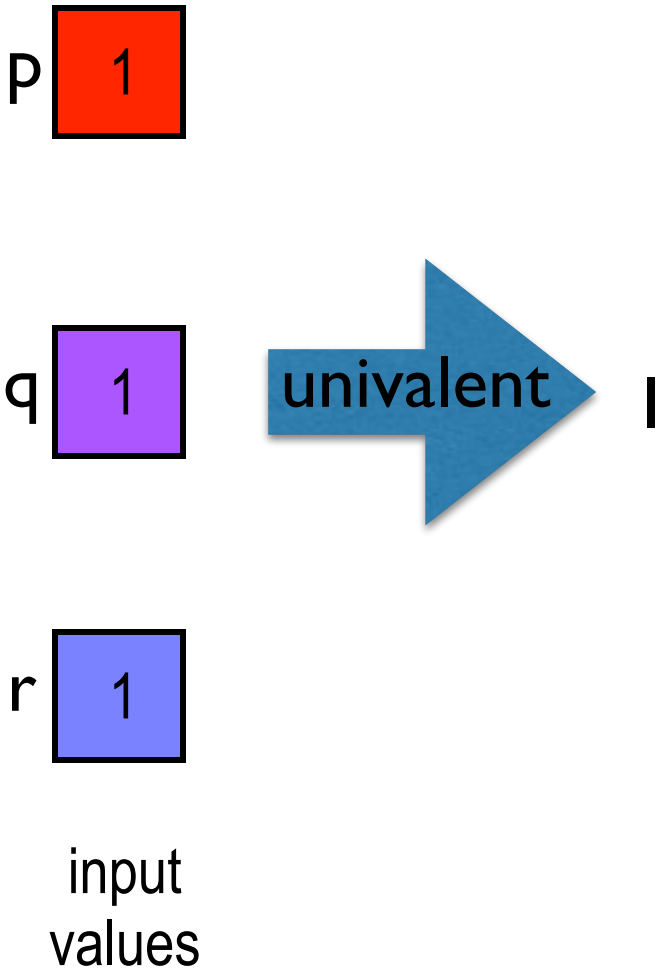
- Assume proposed values are in {0,1}
- Definition: a **configuration** is a
  - N-tuple of proposed values
- Some configurations will always result in the same decision, e.g.,
  - configuration 1,1,1 result in 1 (1-deciding)
  - configuration 0,0,0 result in 0 (0-deciding)

Bivalent Configuration

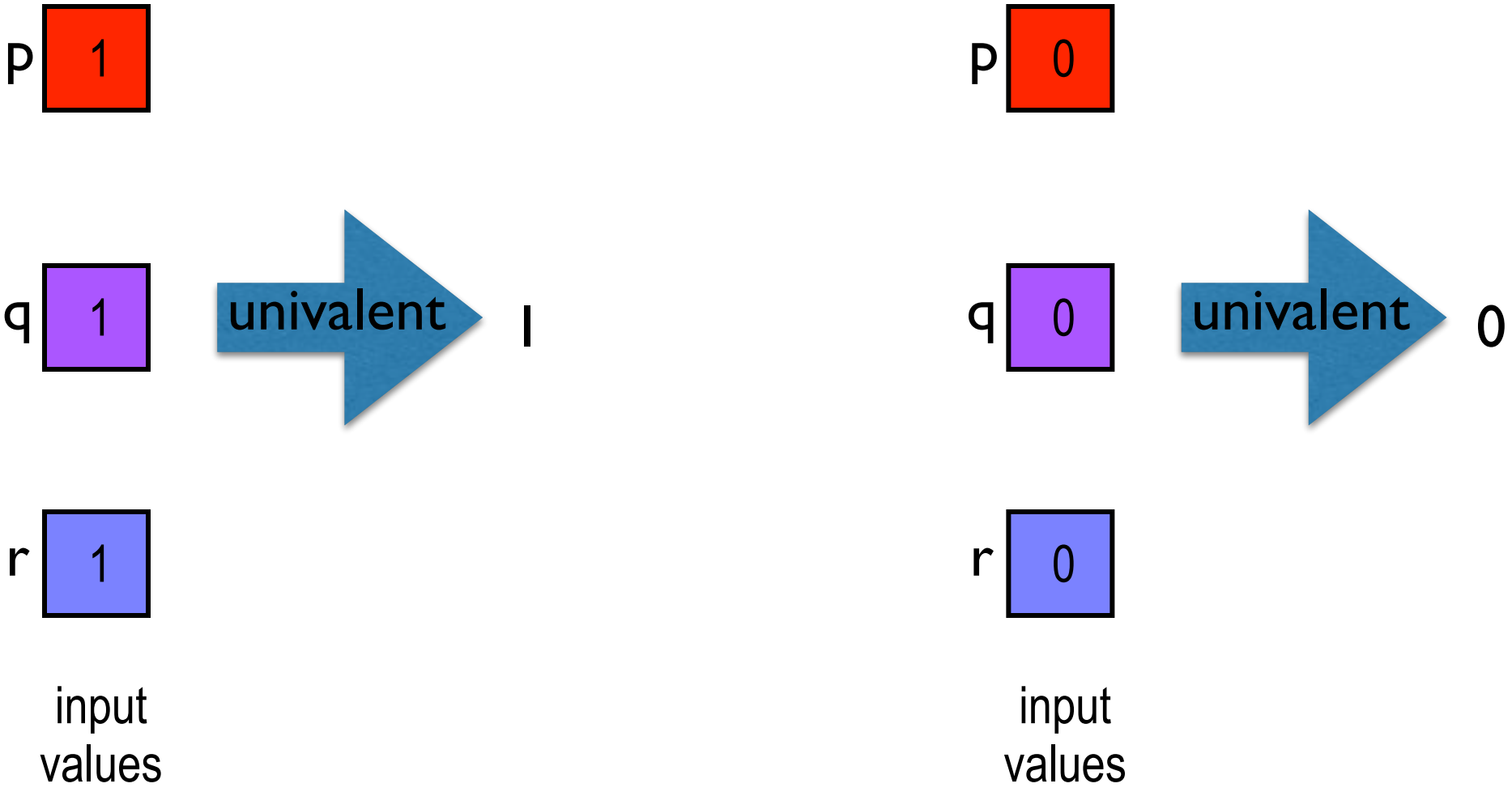
- Assume proposed values are in {0,1}
- Definition: a **configuration** is a
  - N-tuple of proposed values
- Some configurations will always result in the same decision, e.g.,
  - configuration 1,1,1 result in 1 (1-deciding)
  - configuration 0,0,0 result in 0 (0-deciding)
- **Bivalent configurations**
  - permit decision values of either 0 or 1.

Do bivalent configurations exist?

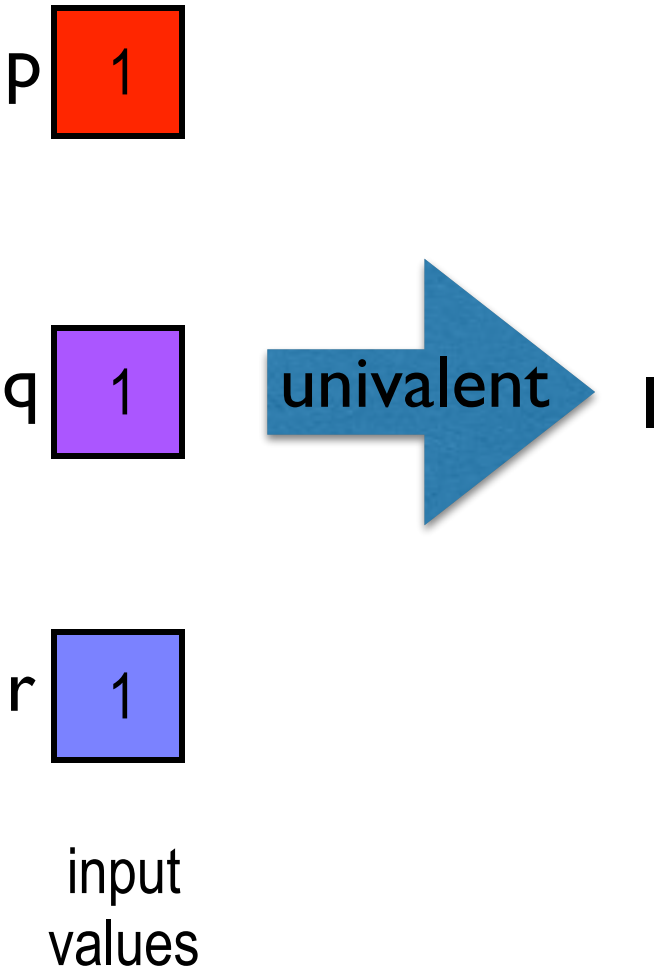
Do bivalent configurations exist?



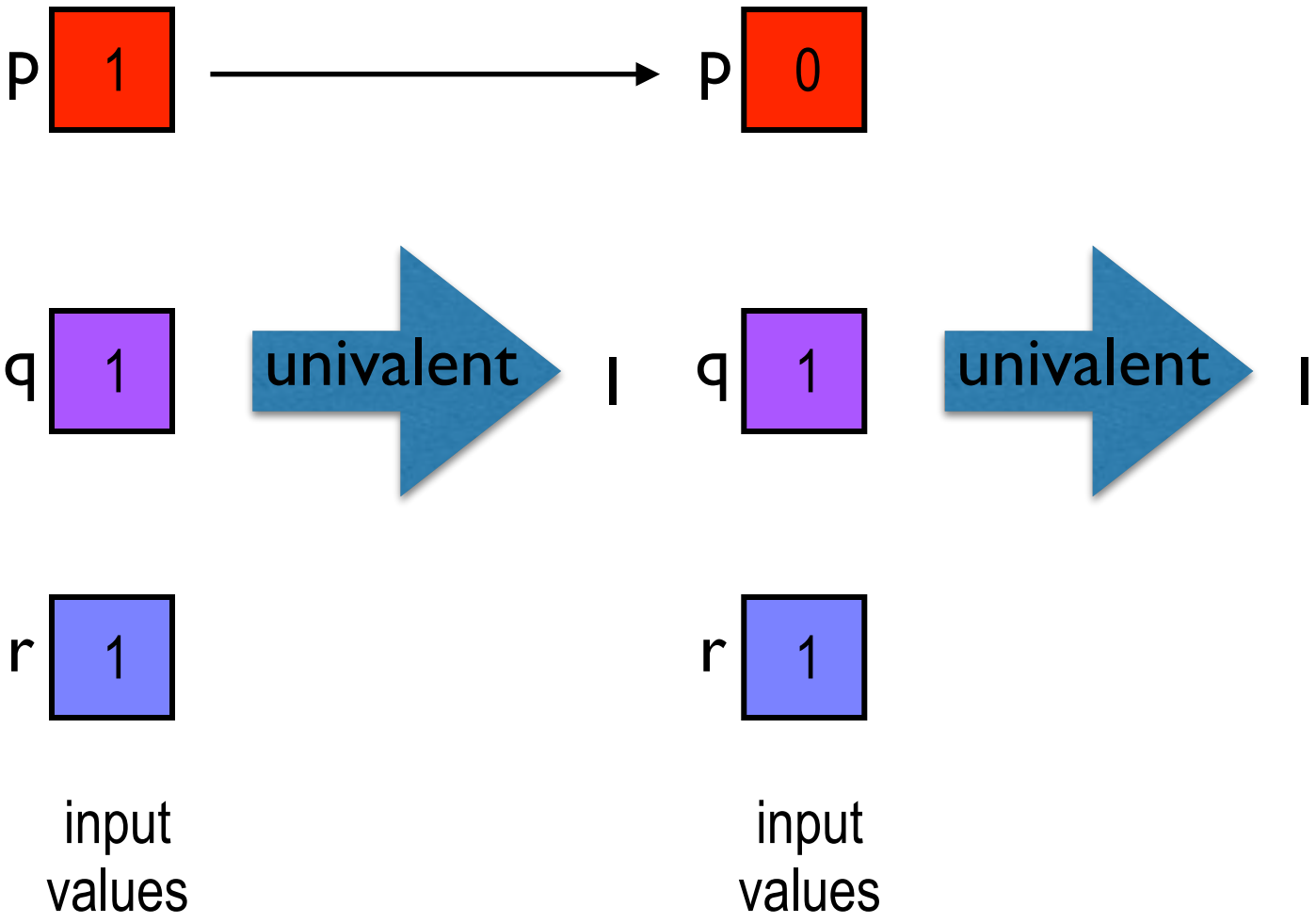
Do bivalent configurations exist?



Assume only univalent configurations exist..

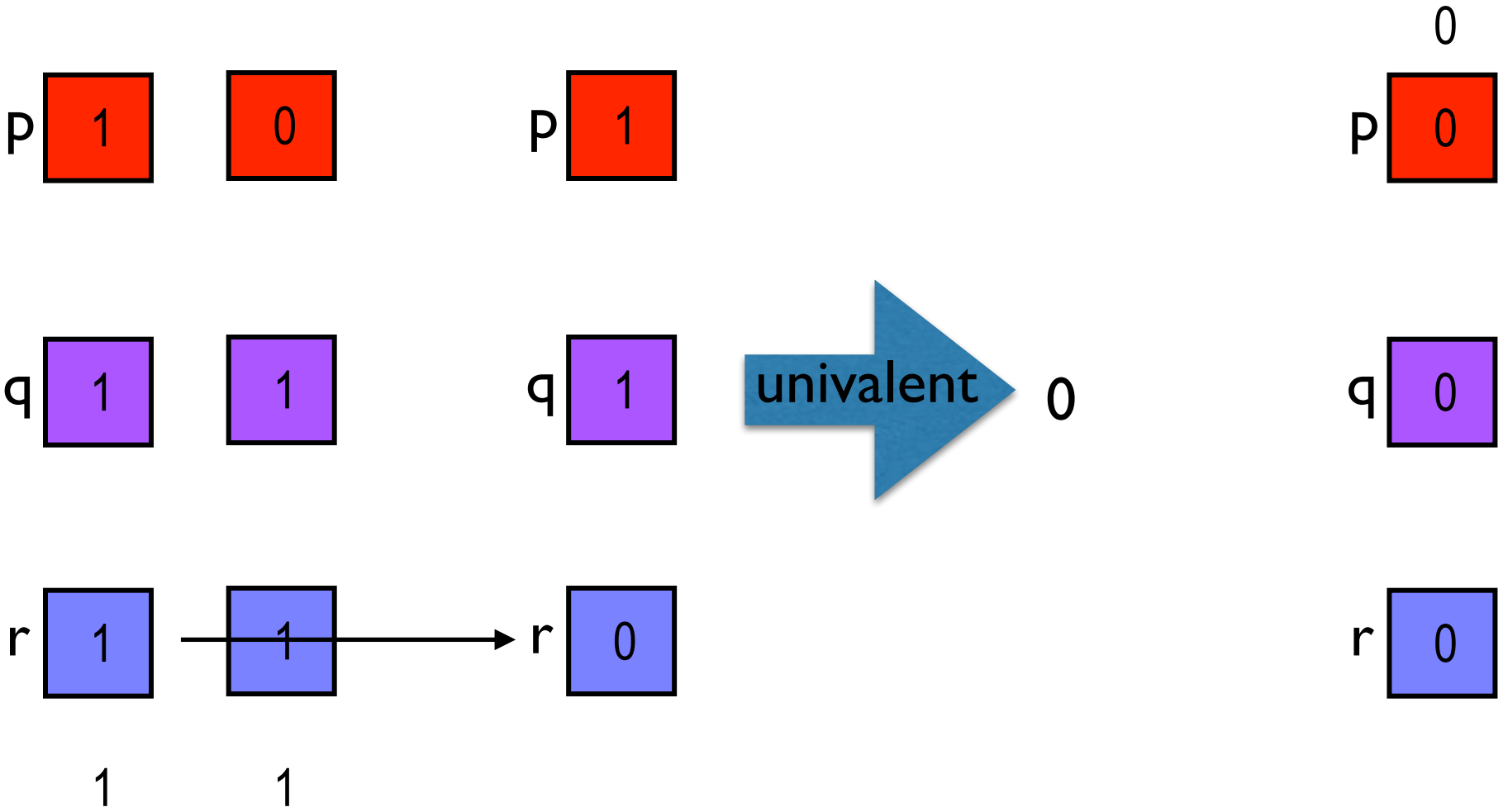


Assume only univalent configurations exist..

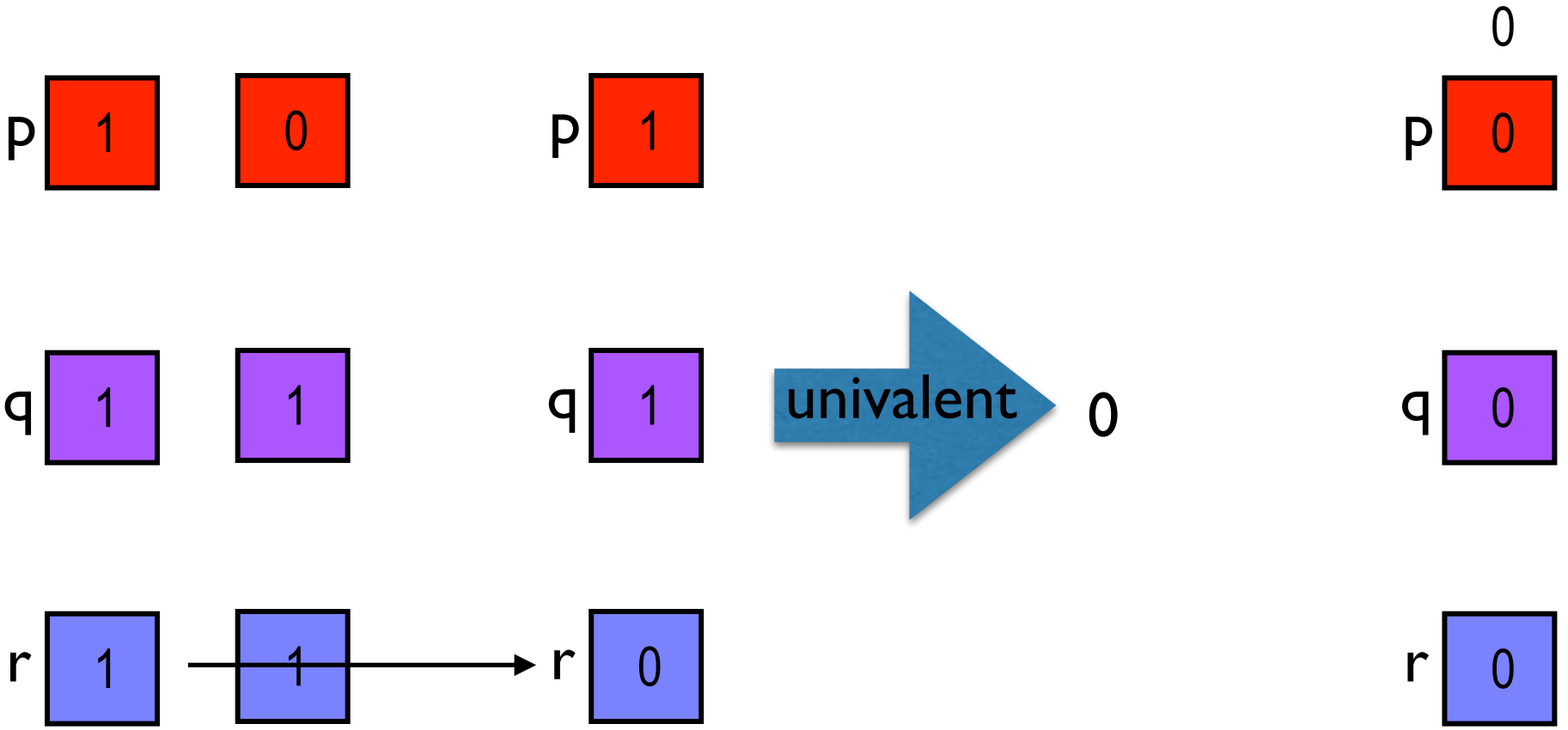


Assume only univalent configurations exist..

Assume only univalent configurations exist..

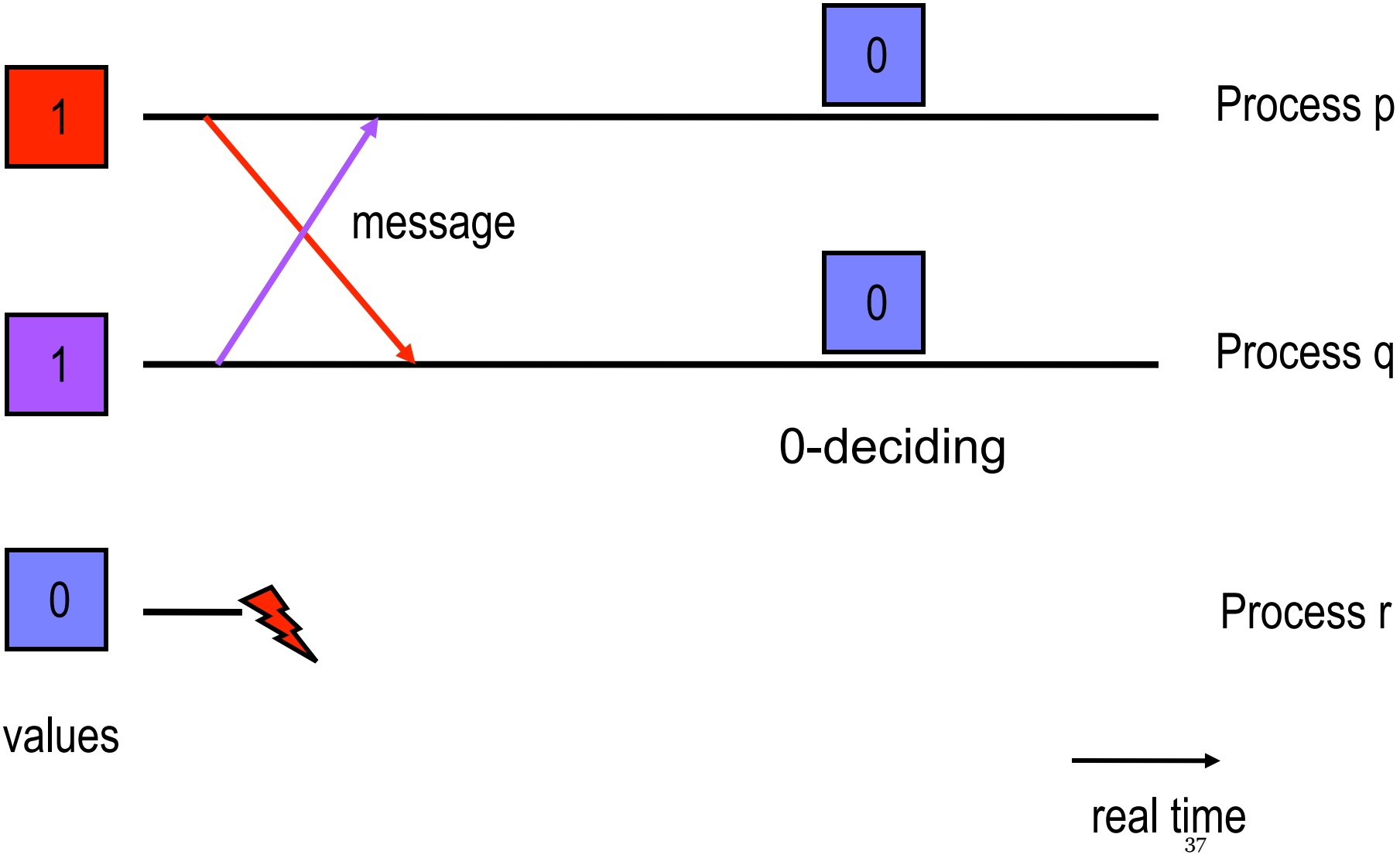


Assume only univalent configurations exist..

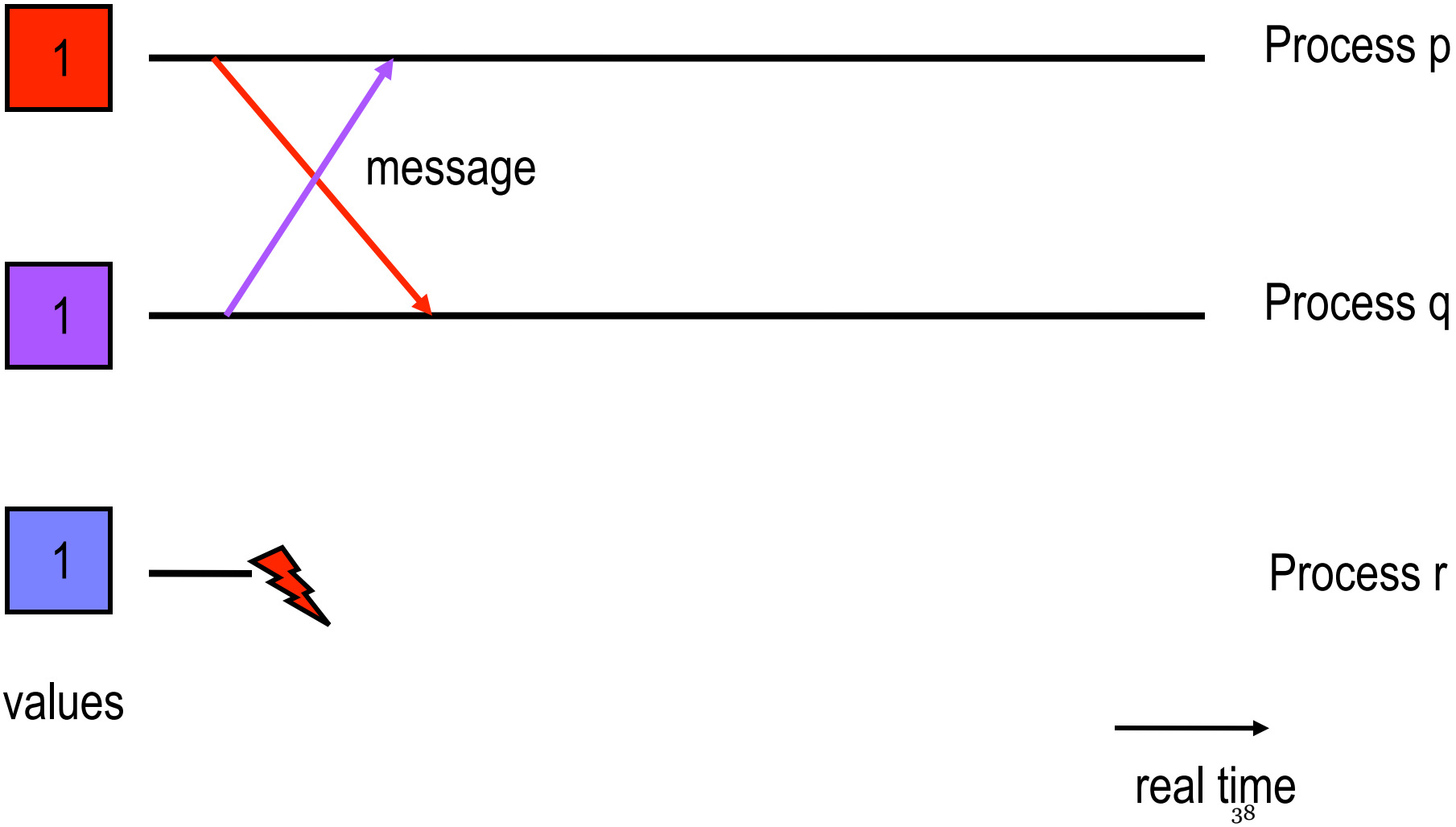


We will find a single input change for which the decision switches from 1 to 0

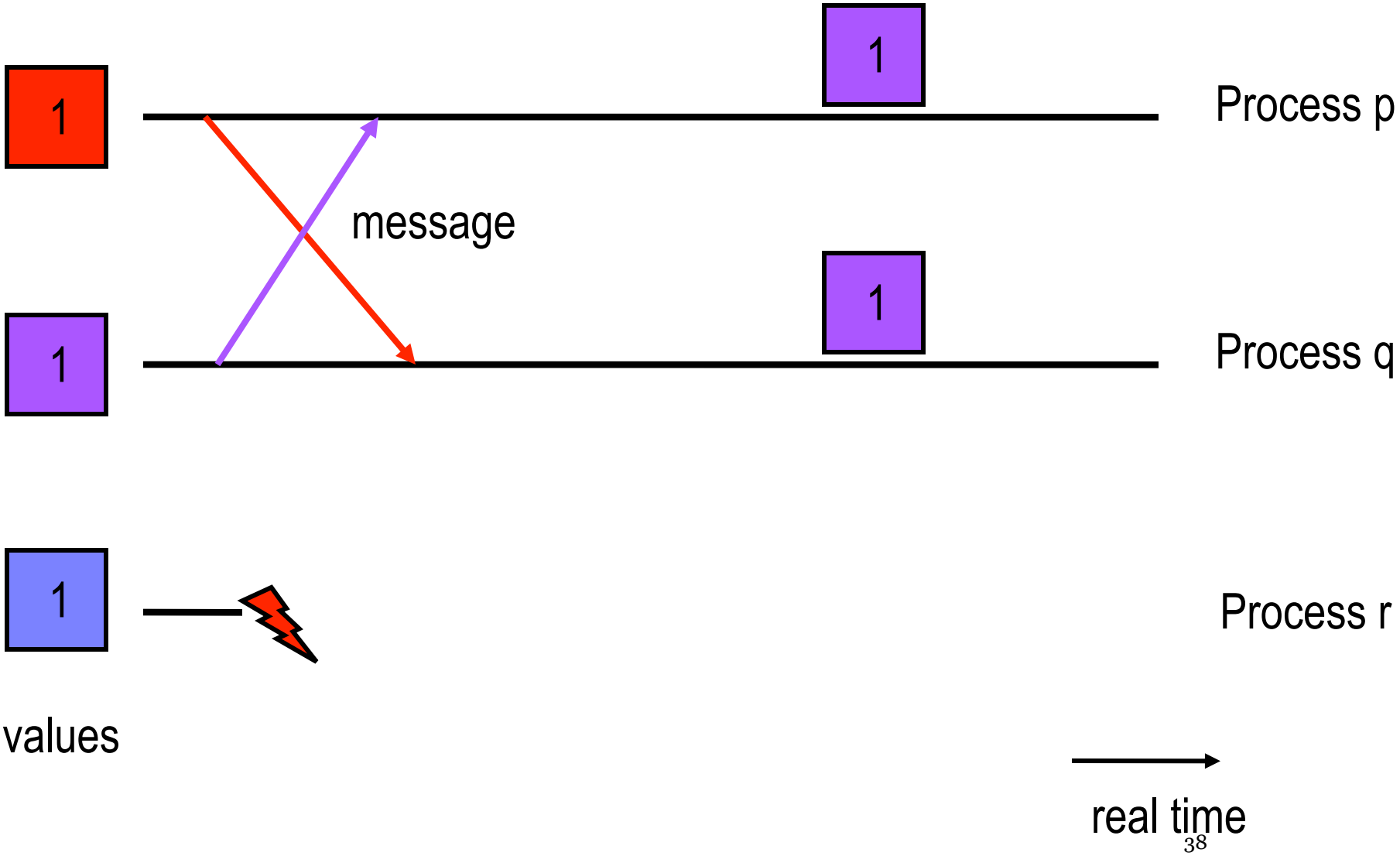
Run 2: Permitted by specification



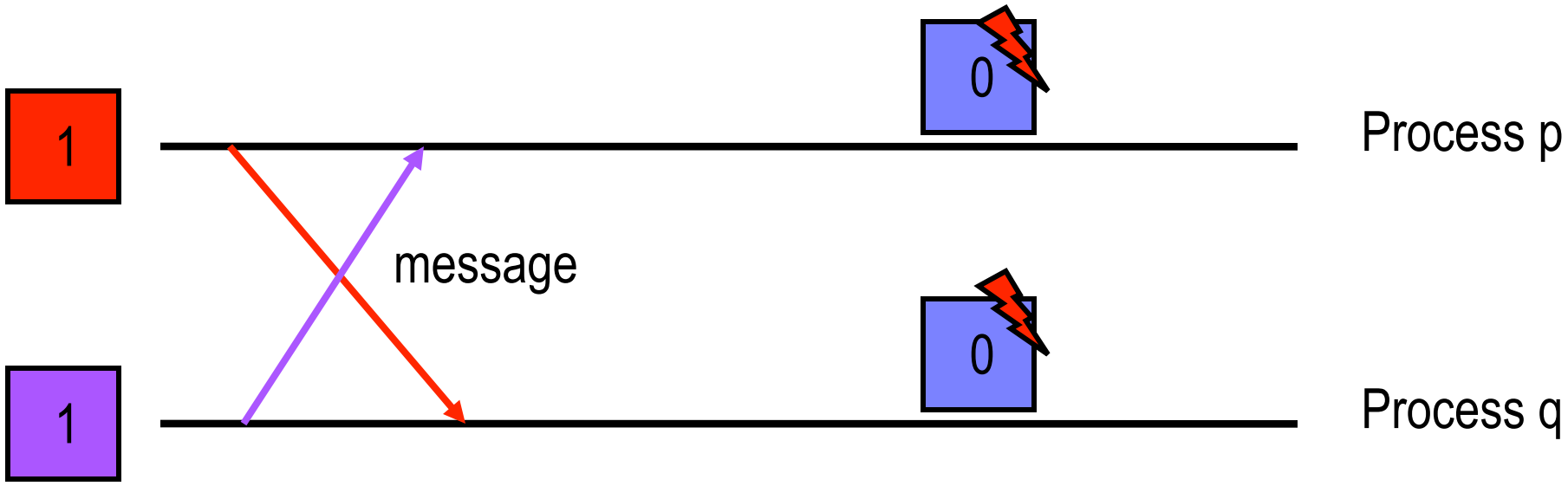
Run 3: Must be 1-deciding



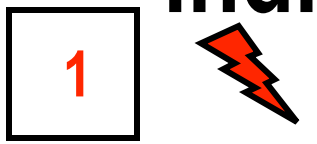
Run 3: Must be 1-deciding



Run 2 is incorrect!



indistinguishable by p and q from run 3



Process r

$\Rightarrow$ 1,1,0 is bivalent

values

real time

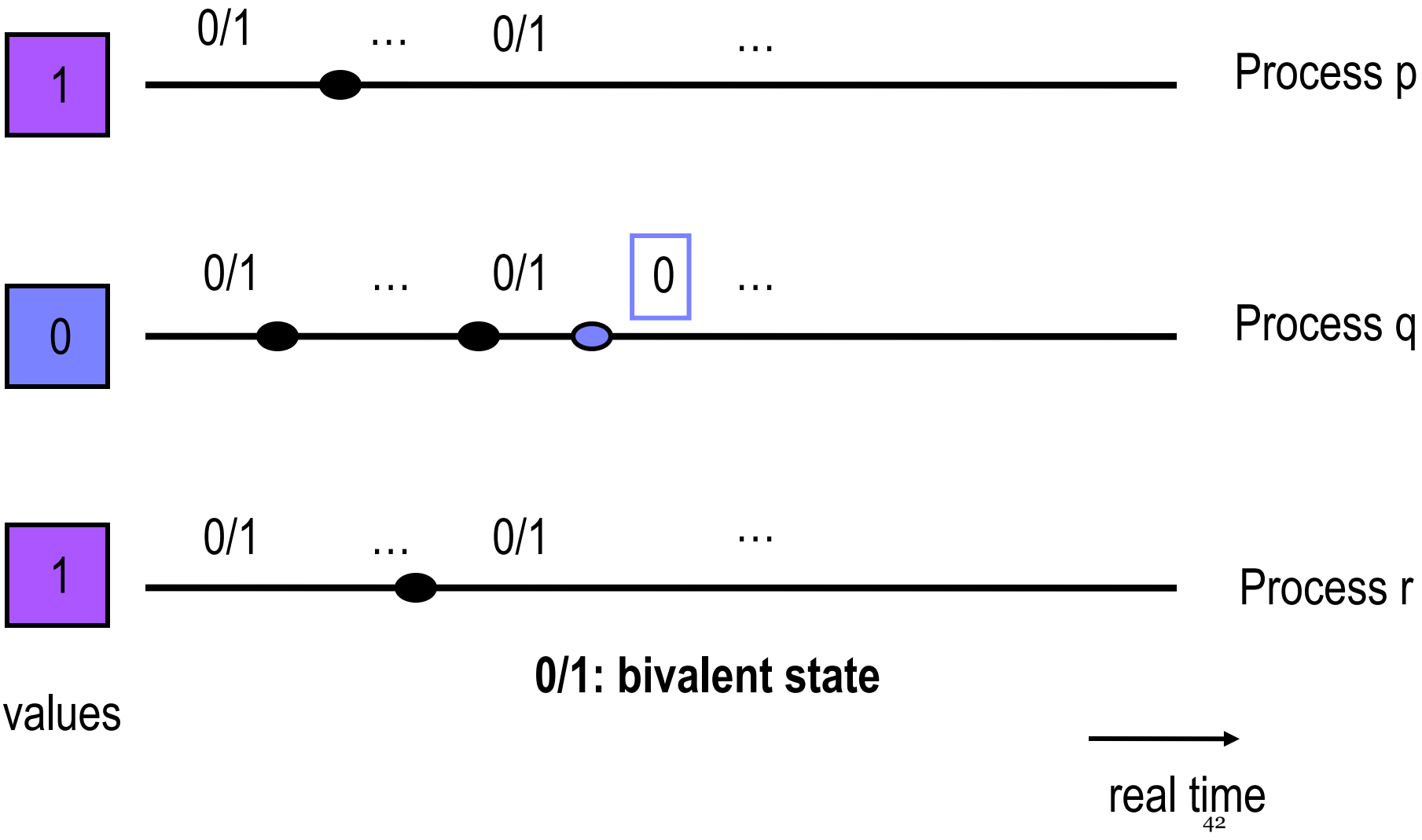
Many seemingly easy problems are impossible to solve!

Impossibility Result (FLP1985)

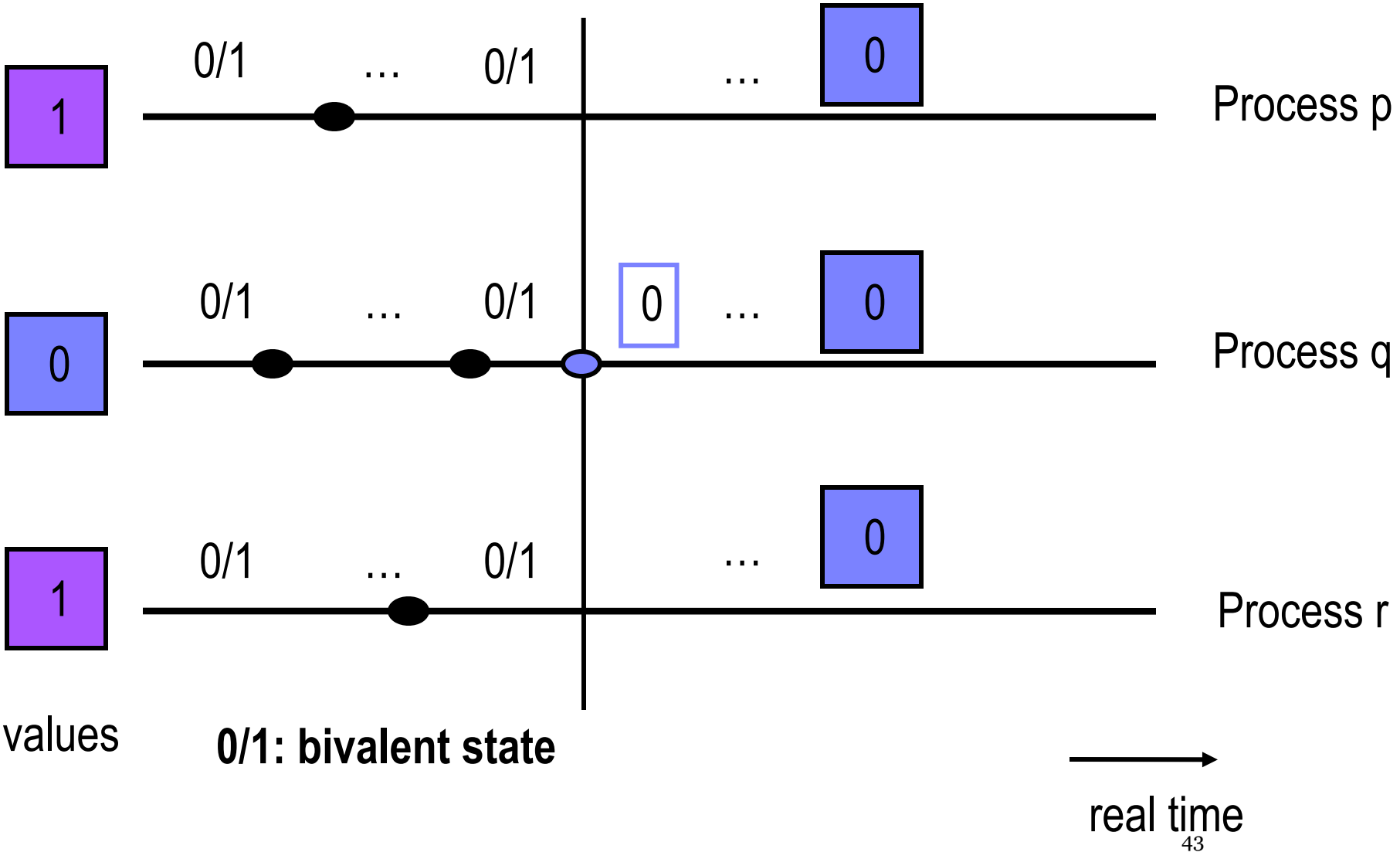
Consensus is impossible to solve even in systems with

- ***Reliable channels***
  - Messages are delivered unless receiver crashes
- ***At most one process crashes***
- ***BUT no bounds on relative speed of processes, no clocks, no failure detectors***

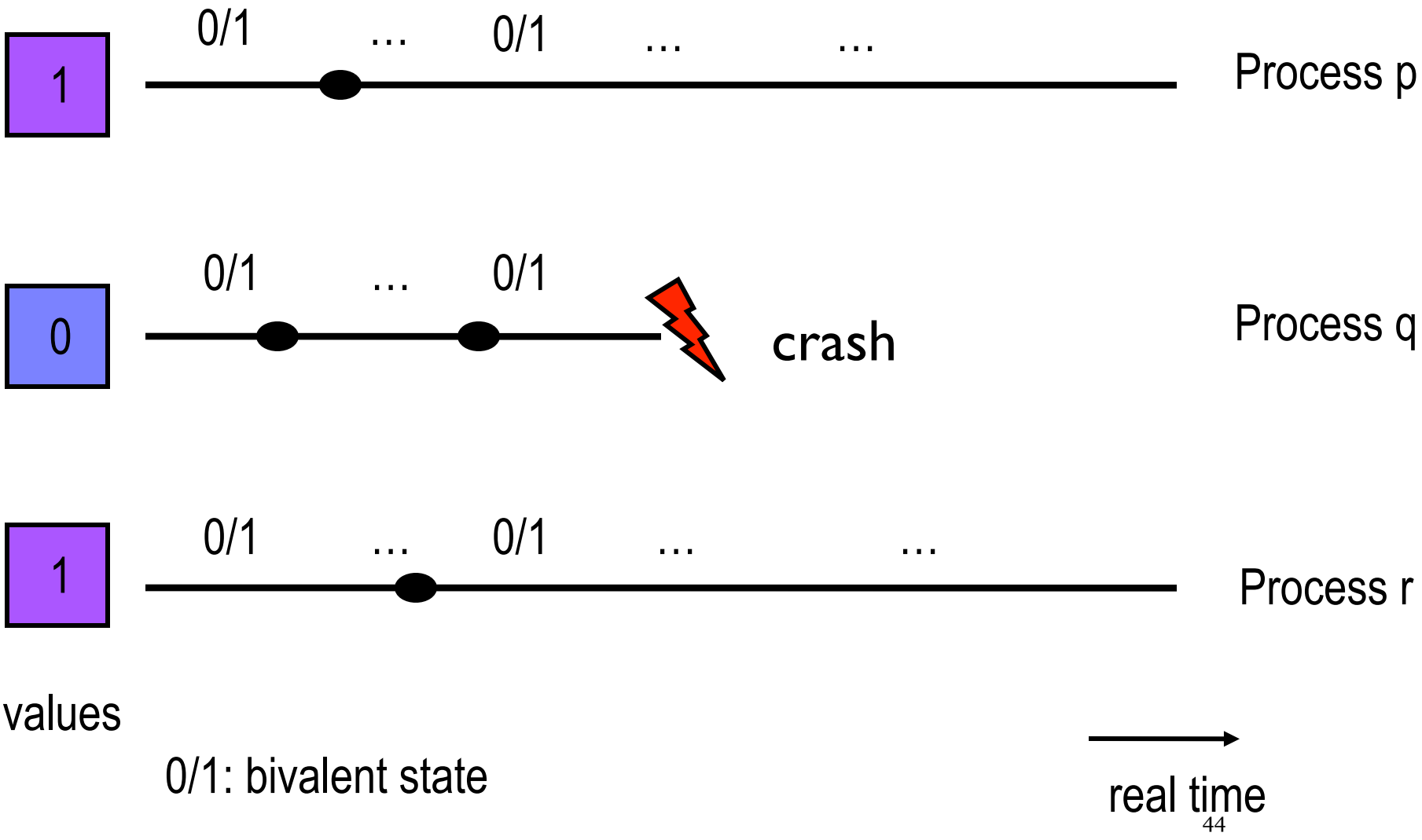
Intuition


1
0/1
…
0/1
…
Process p
0
0/1
…
0/1
0
…
Process q
1
0/1
…
0/1
…
Process r
0/1: bivalent state
values
real time

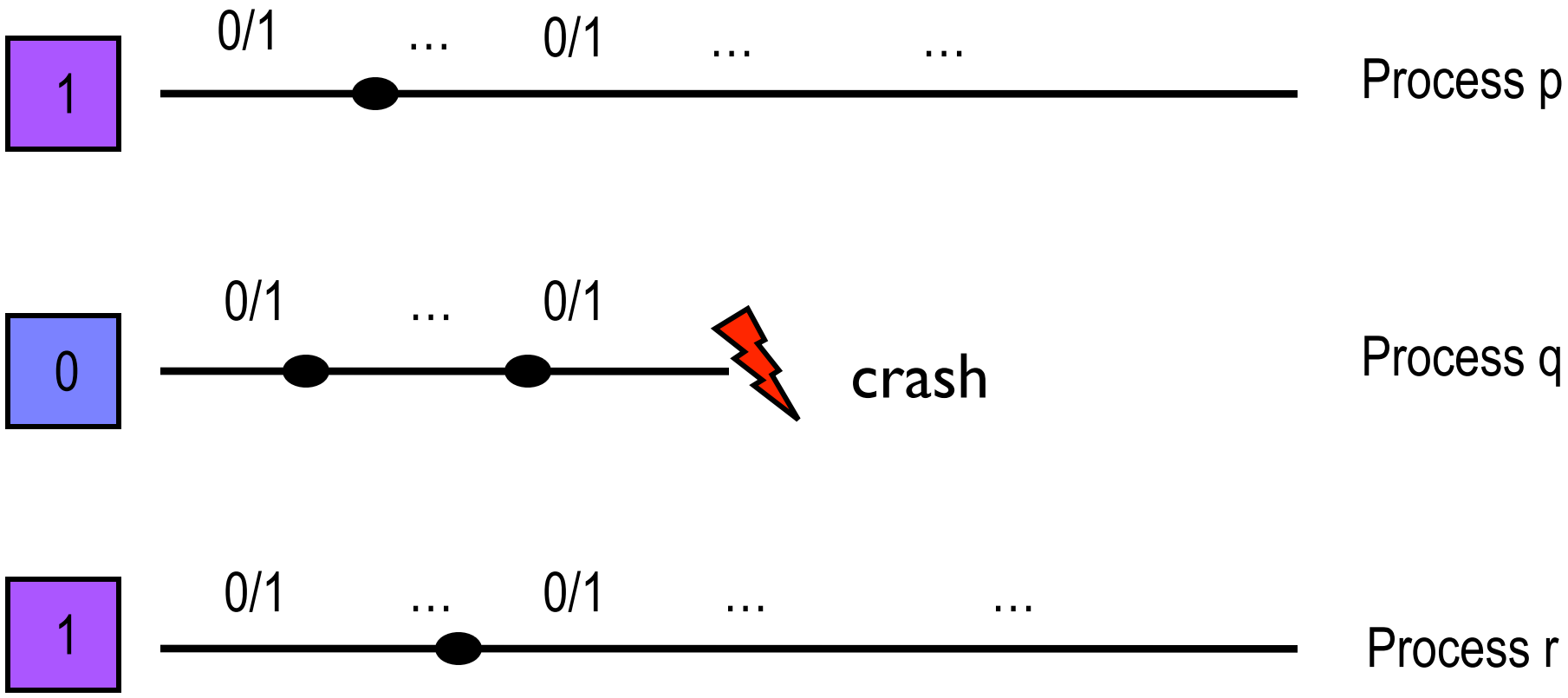
Why Impossible?



Why Impossible?

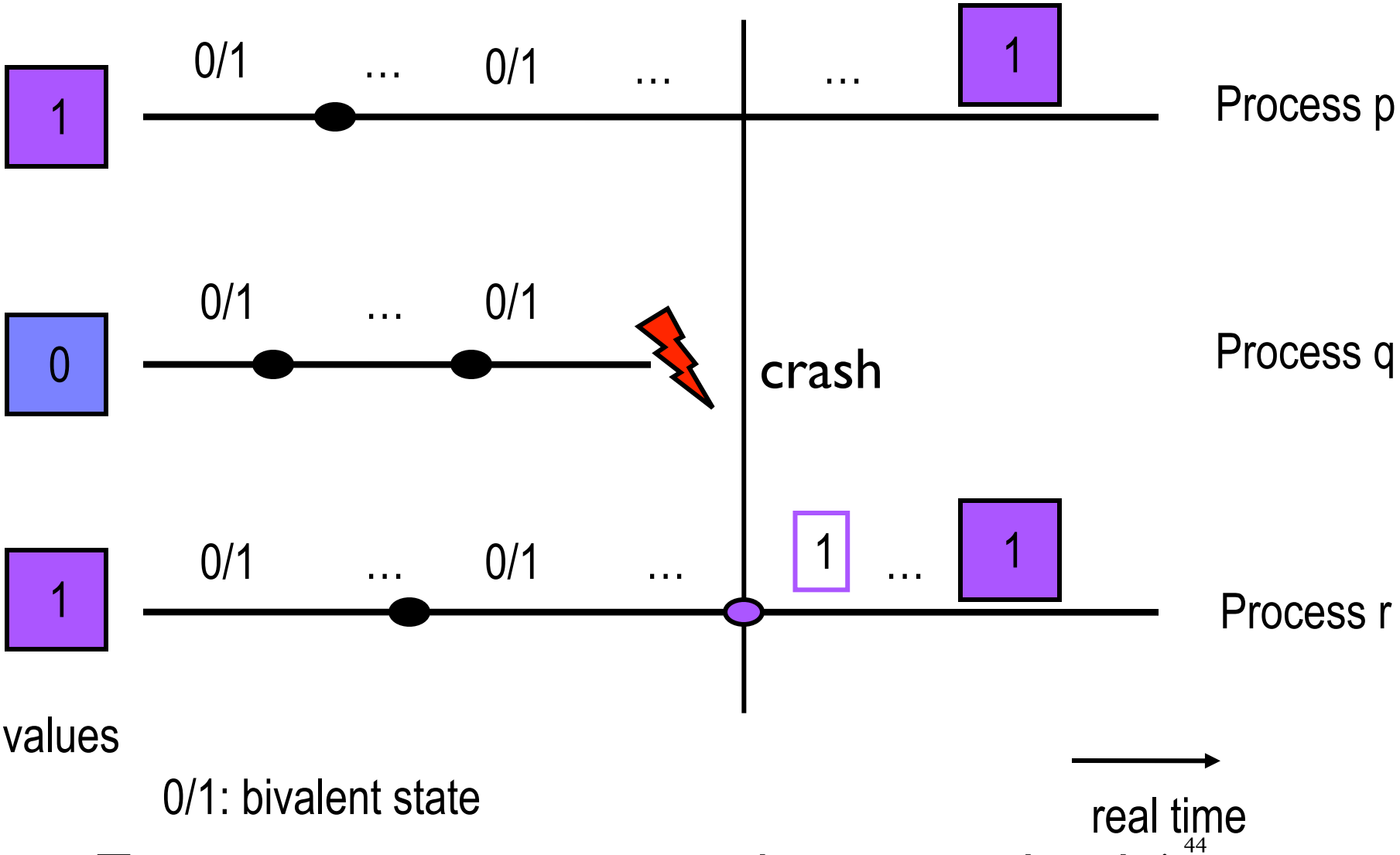


Why Impossible?



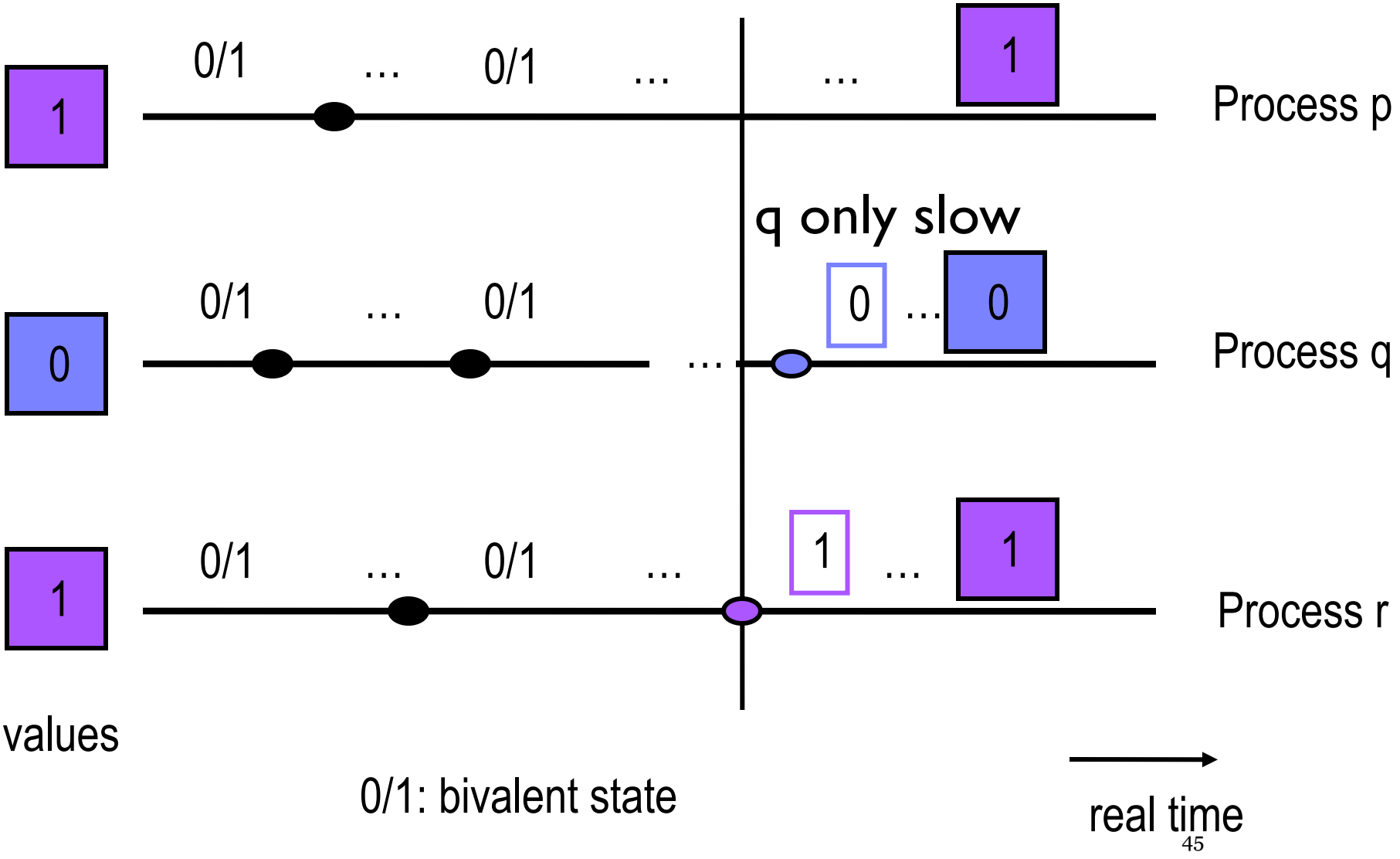
Termination requirement: others must decide!

Why Impossible?



Termination requirement: others must decide!

Why Impossible?



Note

Note:

- This is only the intuition

Question:

- how can we circumvent this impossibility result?

Potential approaches:

- use a **failure detector**, or
- do not always guarantee termination, or …

Failure Detector

- Minimum Consensus Protocol:
  - Wait to get message from all processes that have not crashed!
- Problems:
  - some processes might have received messages from crashed processes! Processes might decide on different values!
- Possible approach:
  - process p needs to ensure that all non-crashed processes learned the same values before p decides !

Paxos

Consensus Protocol

Paxos

Implementation of consensus

- used in several commercial apps (e.g., Google Chubby)

Very efficient algorithm for achieving

- consensus in a message-passing system

First described by

- Lesli Lamport in 1990 in a tech report

Guarantees safety, i.e.,

- agreement and validity

Termination

- if there is a long enough interval in which the system behaves "well".

System Model

Processes communicate

- through messages

Messages are asynchronous

- no bounds on the transmission delay, but
- eventually delivered between **correct** processes

Processes can

- **restart and remember**

Roles

There are three basic roles of a process:

- **Proposers** that propose a value for consensus;
- **Acceptors** that choose the consensus value;
- **Learners** that learn the consensus value.

A single process

- may take on multiple roles.

Paxos - Idea

A proposer attempts to gain acceptance

- by the majority of acceptors

Agreement is enforced

- by allowing only one value to be accepted by majority

Validity is enforced

- by allowing only input values to be proposed

Non-Termination

- possible with more than two proposals, or if processes fail.
- **Try to minimize by**
  - by proposers can restart the protocol (i.e., propose again)
  - acceptors can be released from old acceptance
  - unique proposal numbers and a primary proposer

Paxos - Idea

Proposal number

- is attached to each proposal

Safety properties are dependent

- only on distinct, **increasing proposal numbers**
- Can be done with timestamp or polling

Messages are

- **Prepare(n)** – n is the proposal number
- **Accept(n,v)** – v is the value

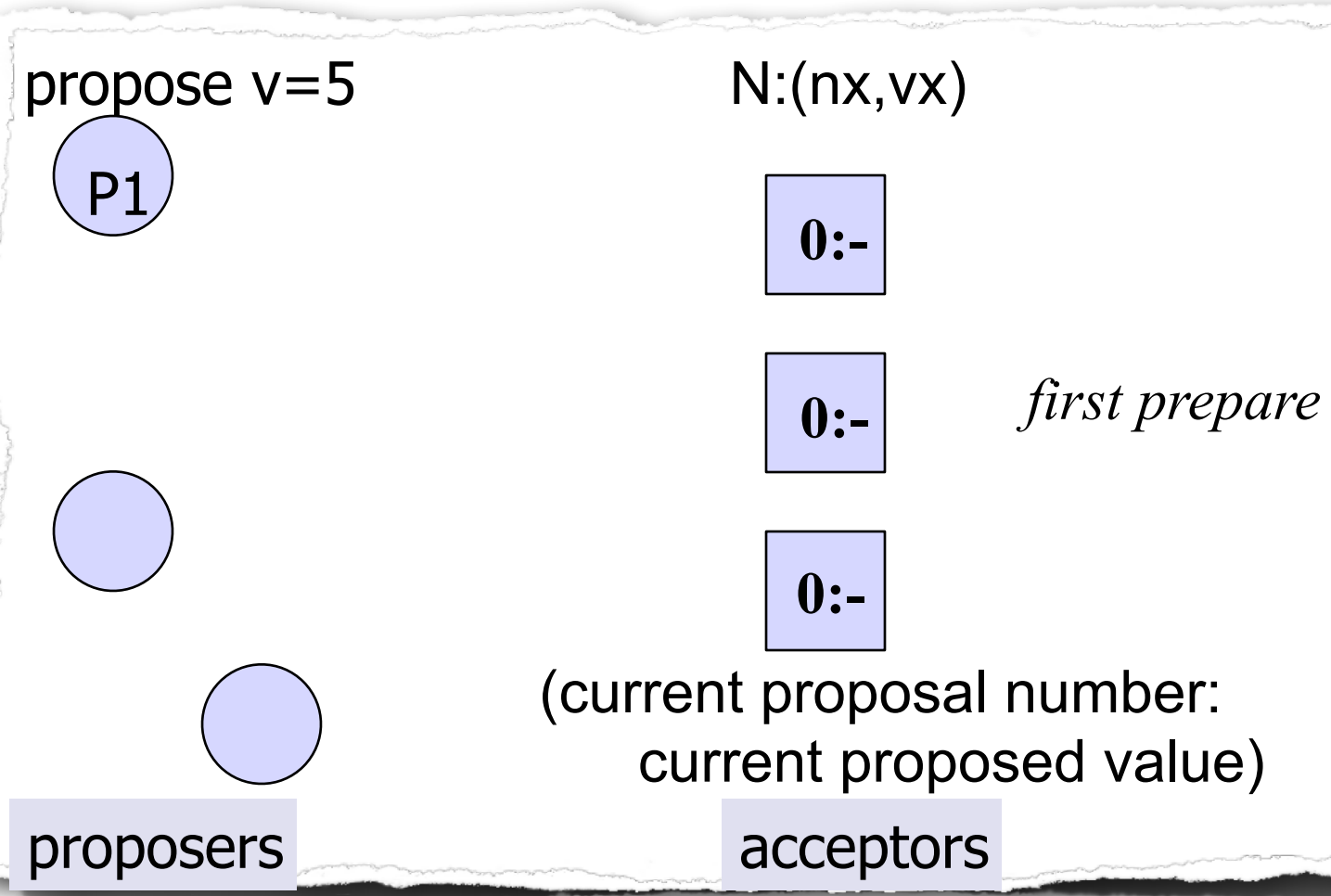
Paxos - Algorithm

- **Phase 1: Preparation**
  - ensure that we do not revise any previous decision

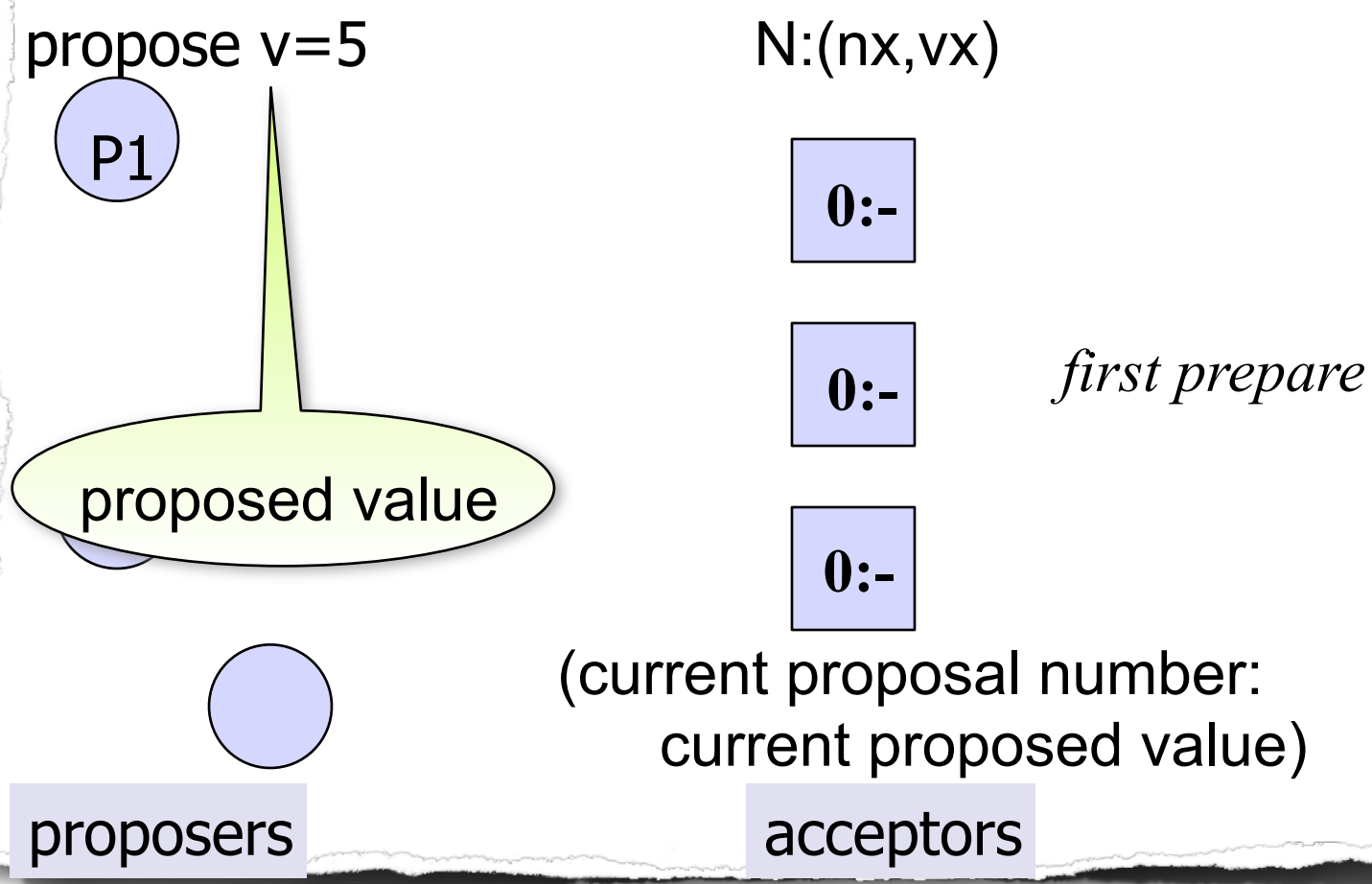
- **Proposer sends**
    - **prepare(n)** to all or a majority of acceptors ;
    - e.g., n = (time(), unique process id);

- **Each Acceptor compares n to the**
  - highest numbered proposal N which it has responded to.
  - If n > N then
    - responds with **ack(n,(nx,vx))**
    - N := n
  - **vx** is the value with the highest proposal number **nx** accepted so far (might be empty), and
  - **ack** is a promise to never **accept** proposal number less than N

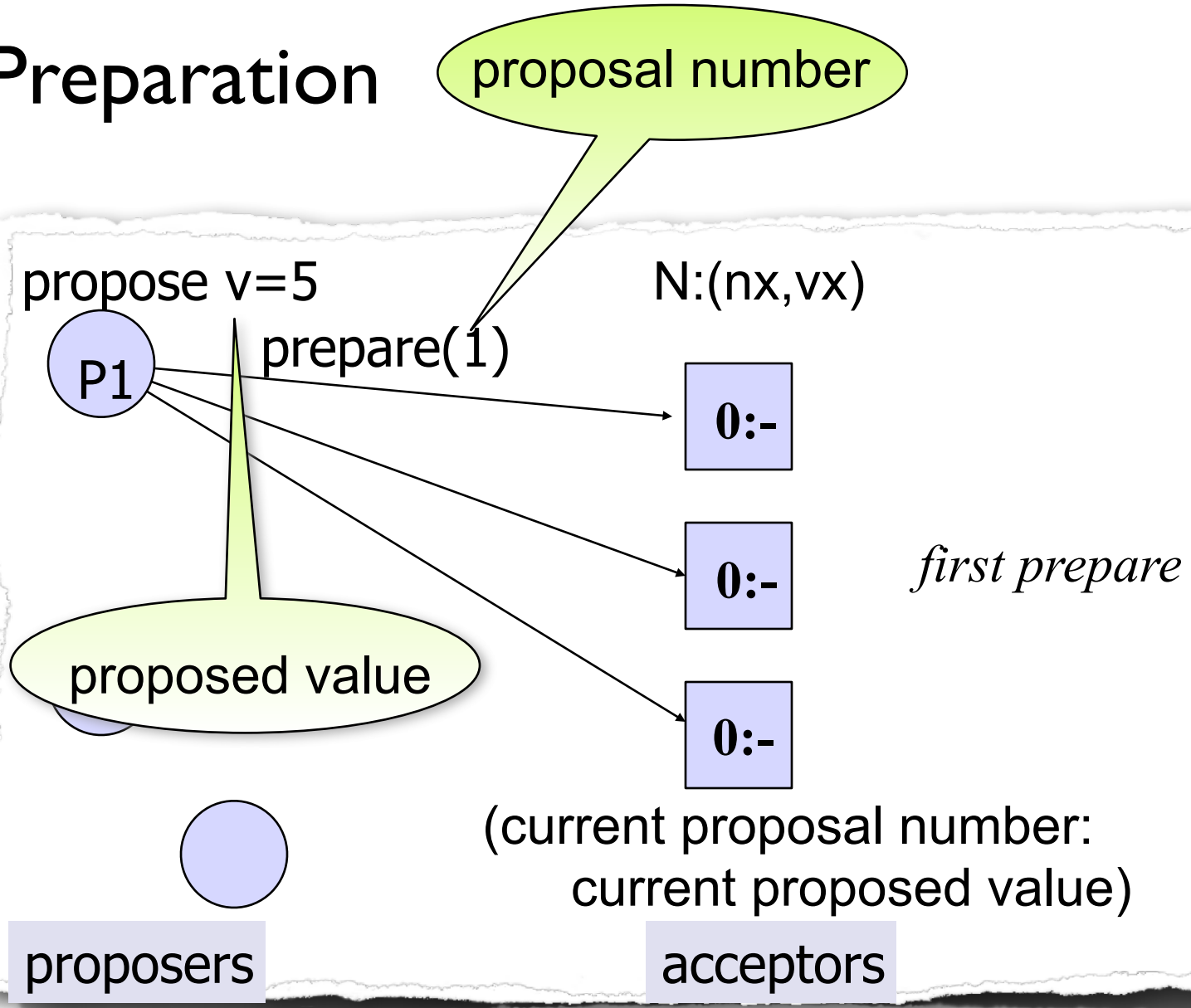
Preparation



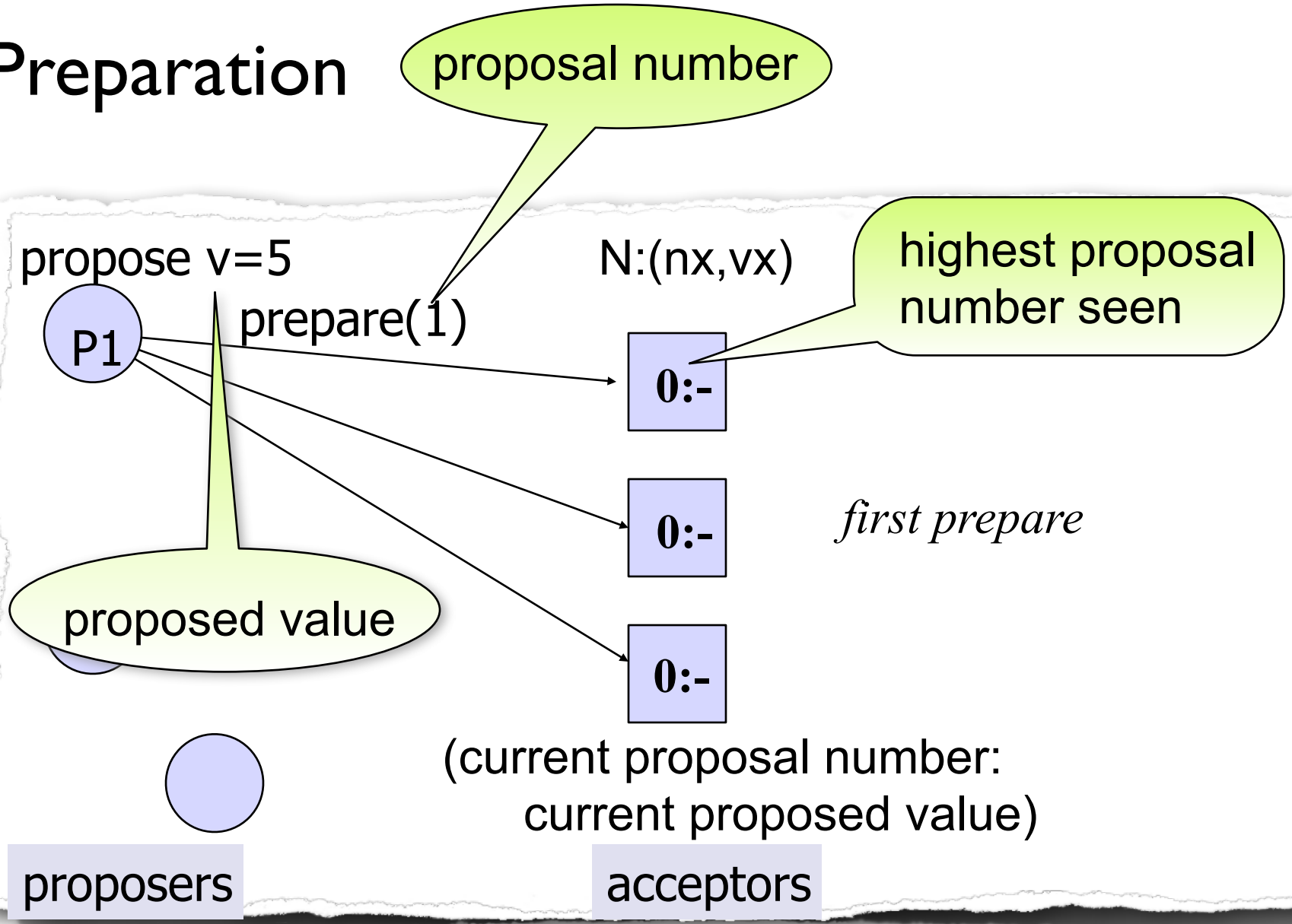
Preparation



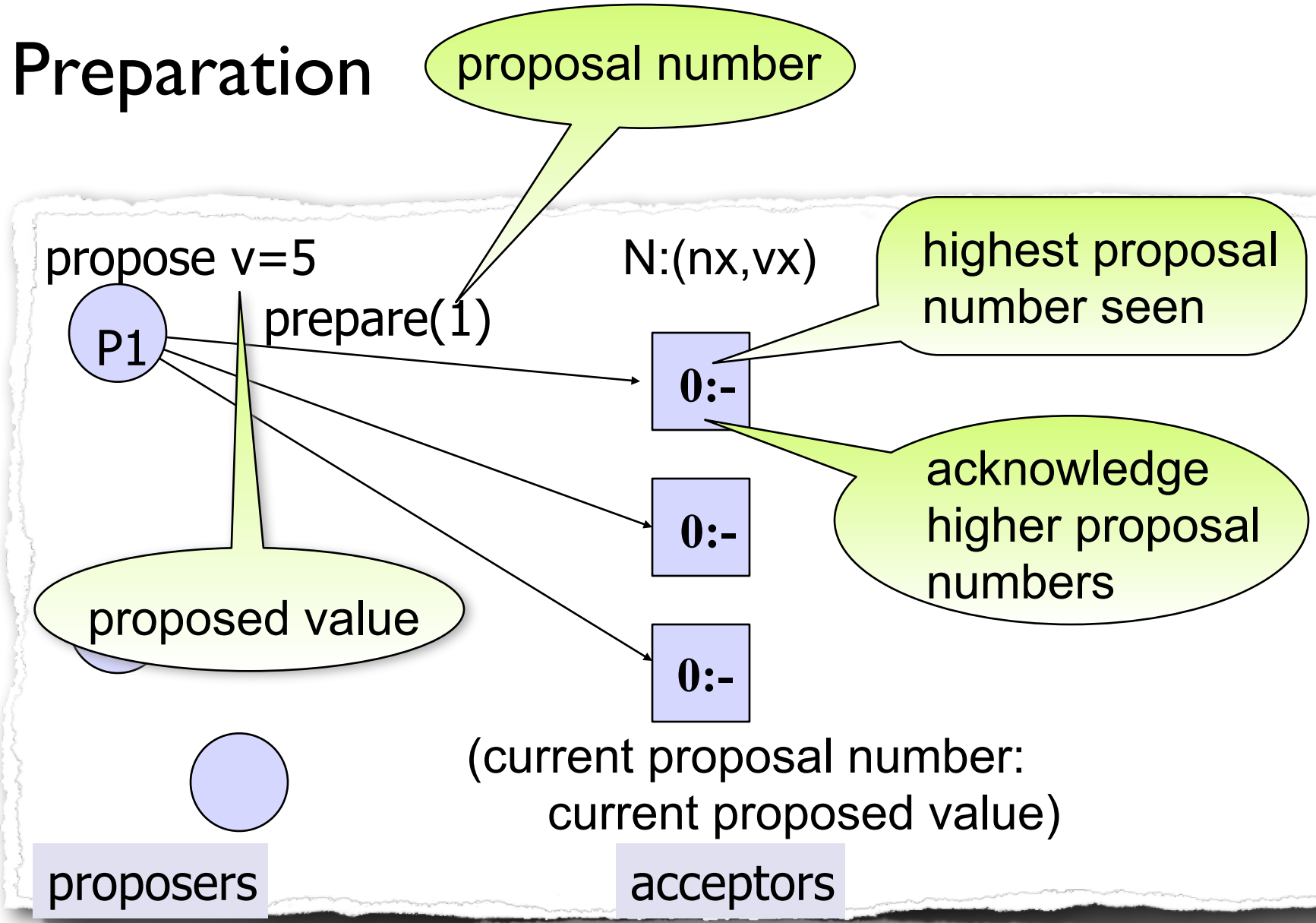
Preparation



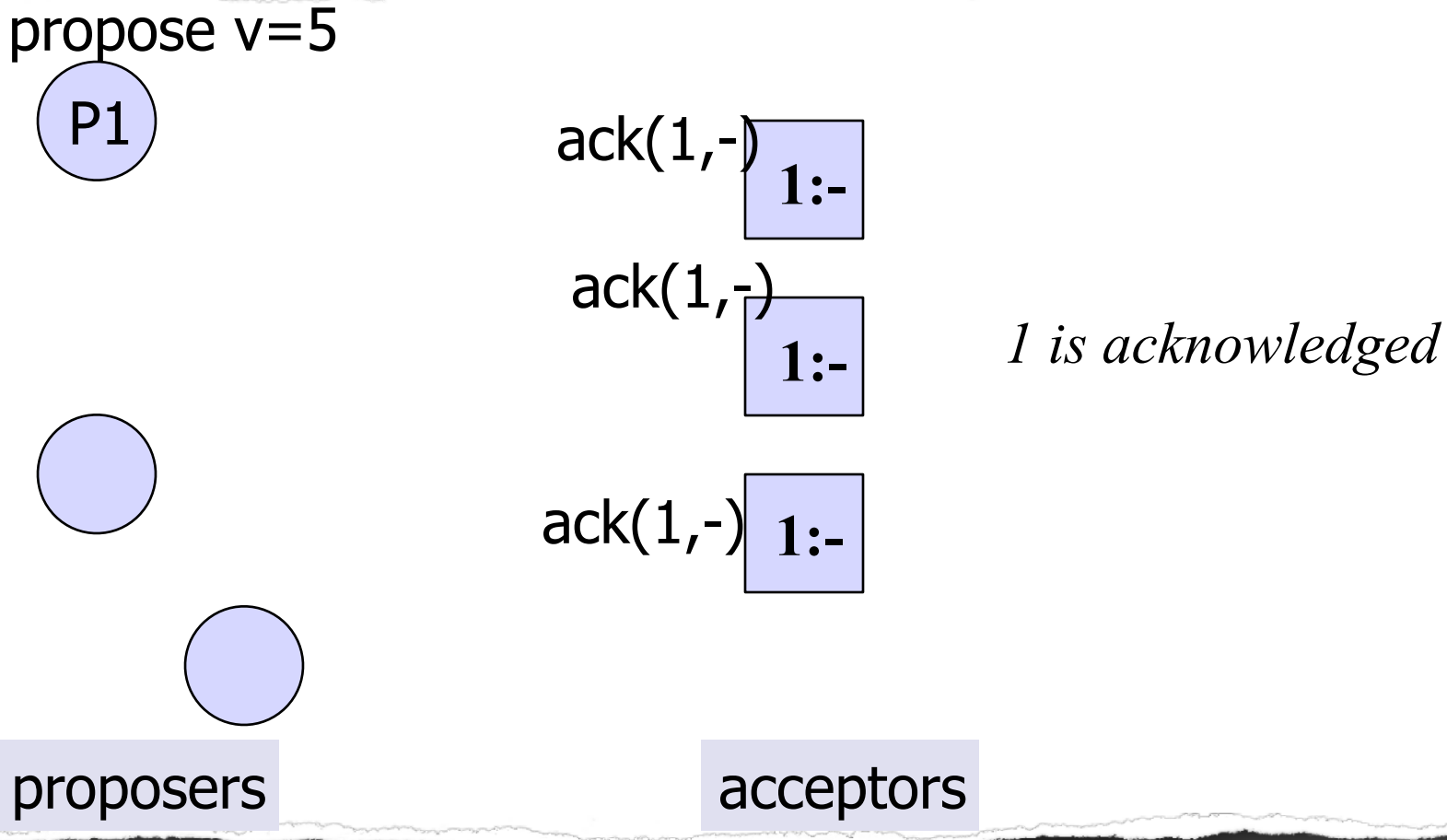
Preparation



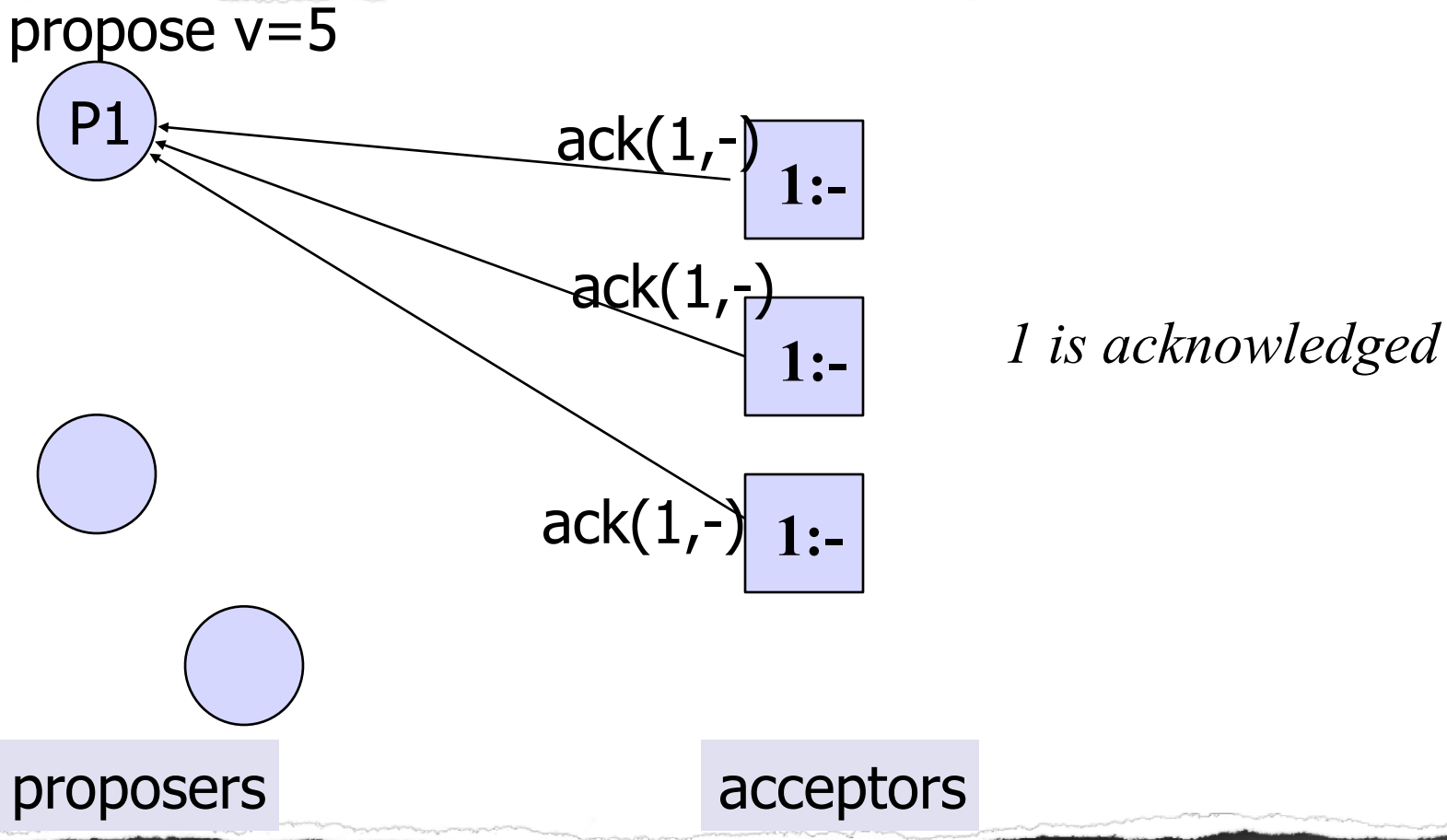
Preparation



Preparation



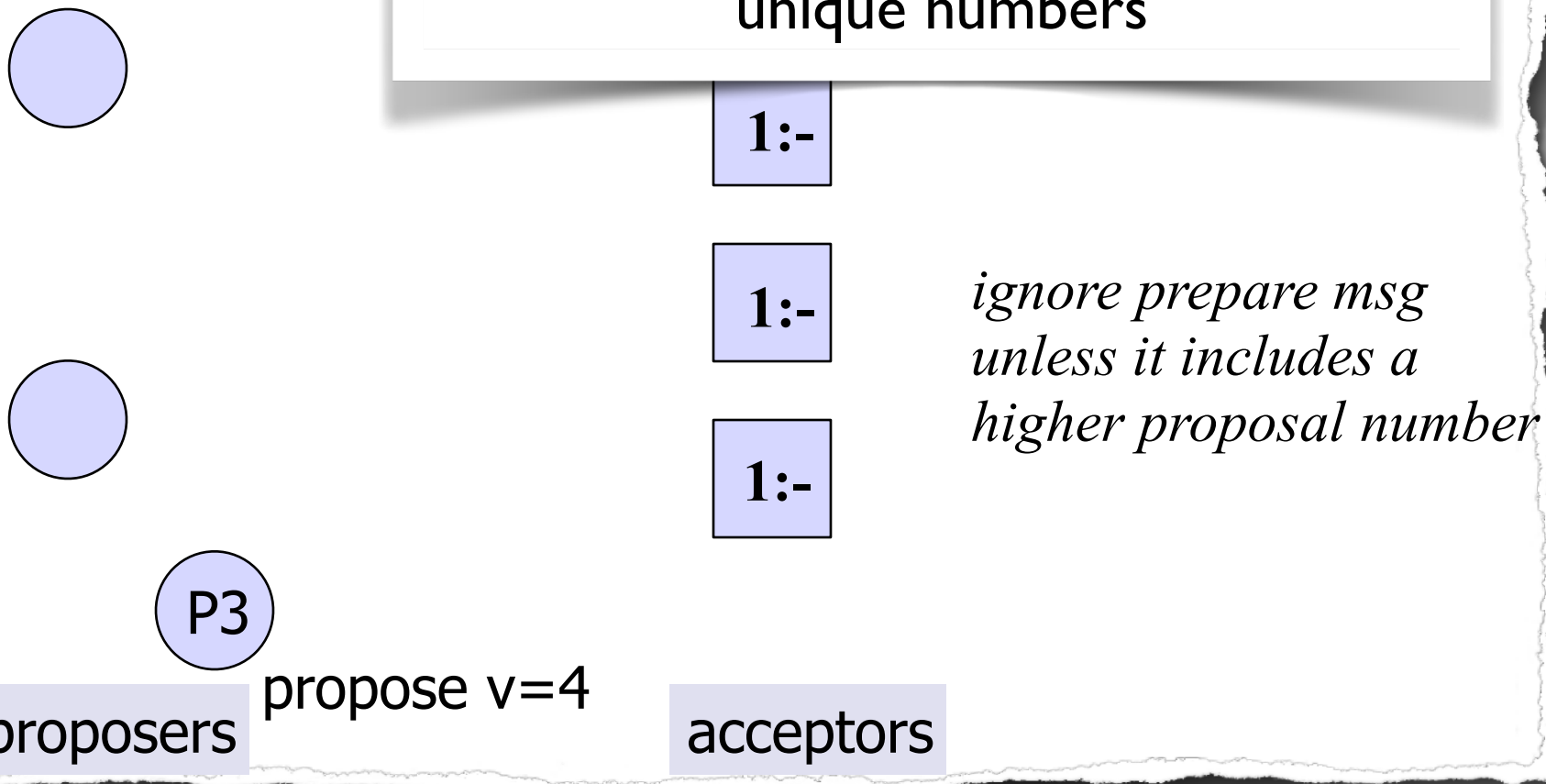
Preparation



1 is acknowledged

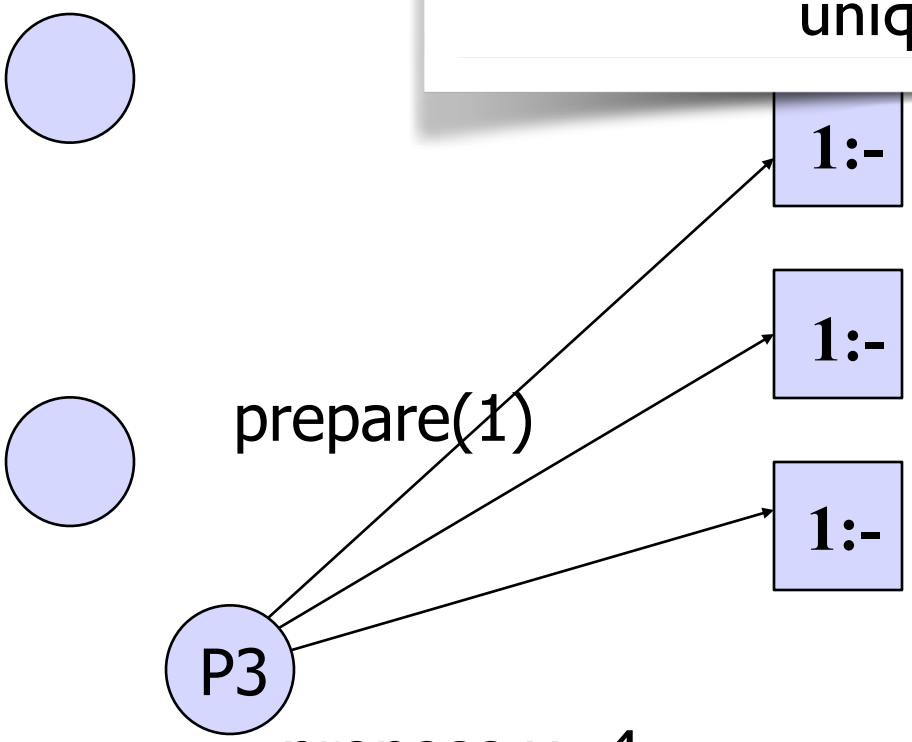
Preparation

proposal numbers may not be unique - but implementations should aim for unique numbers



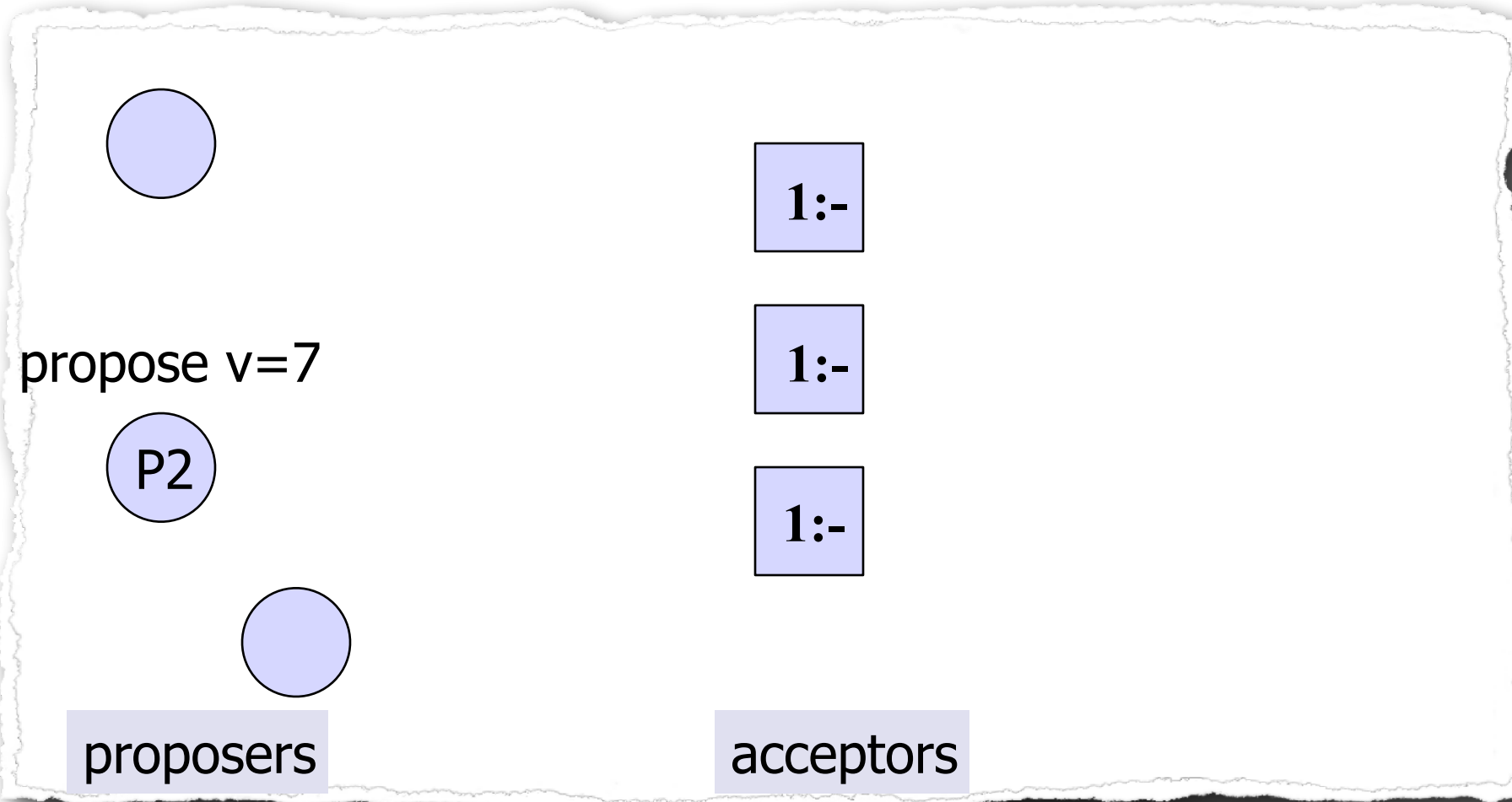
Preparation

proposal numbers may not be unique - but implementations should aim for unique numbers

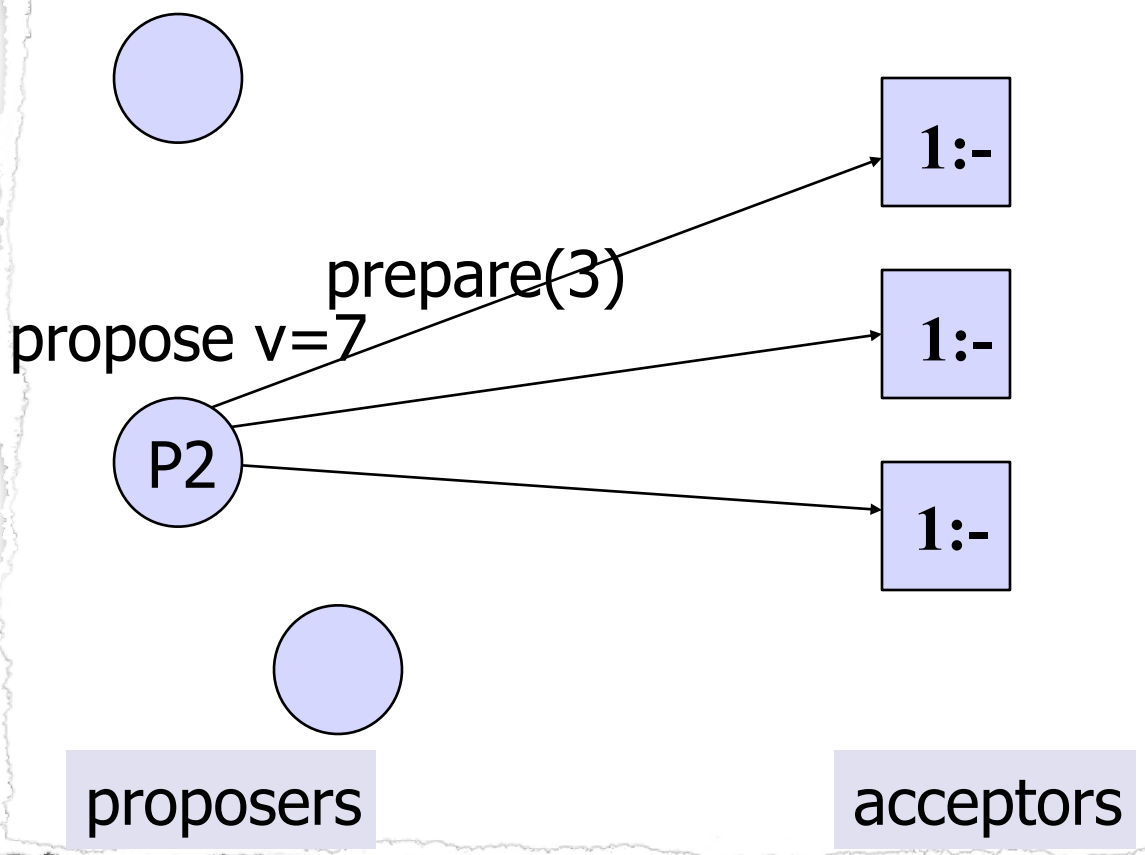


ignore prepare msg unless it includes a higher proposal number

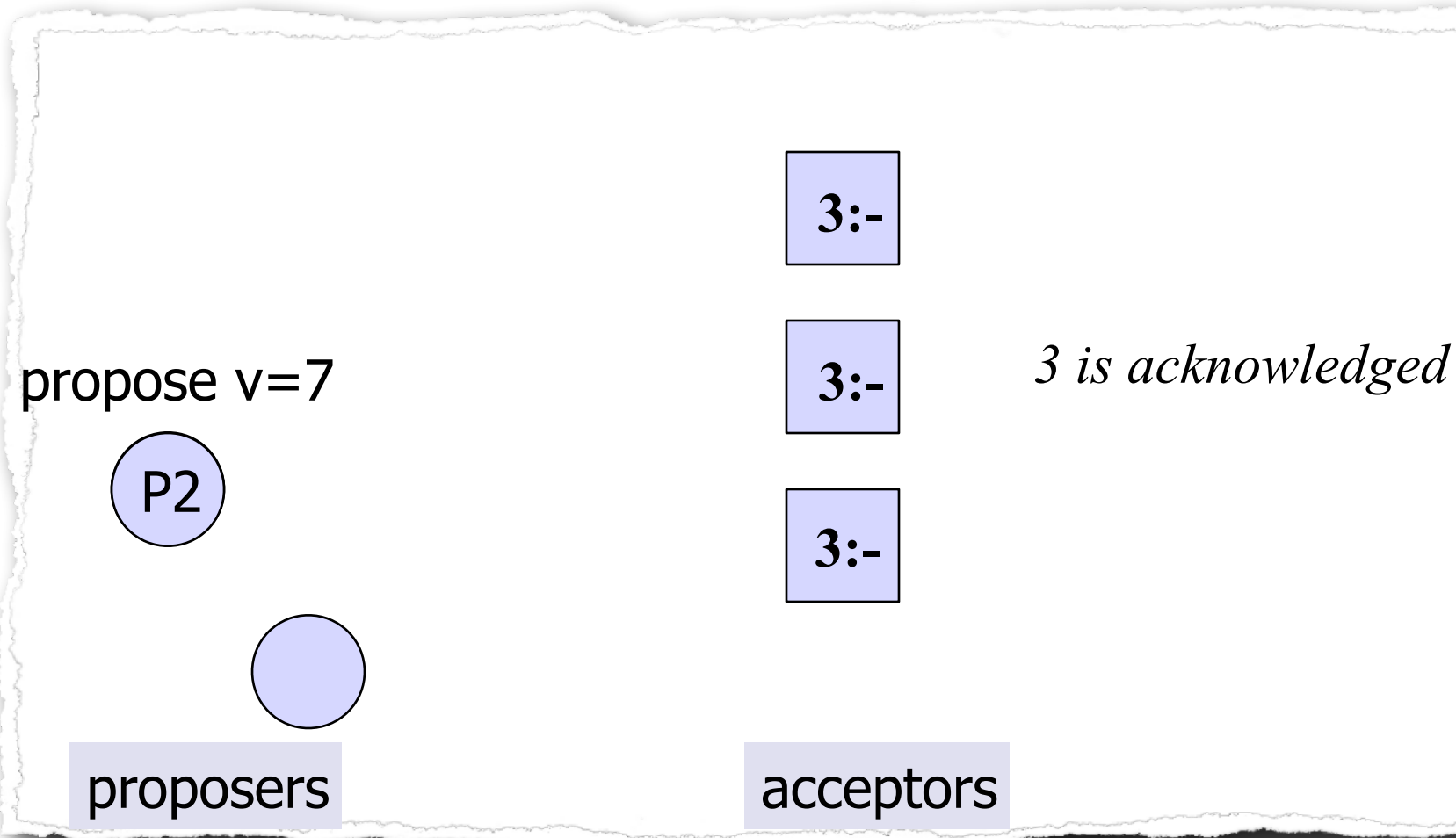
Preparation



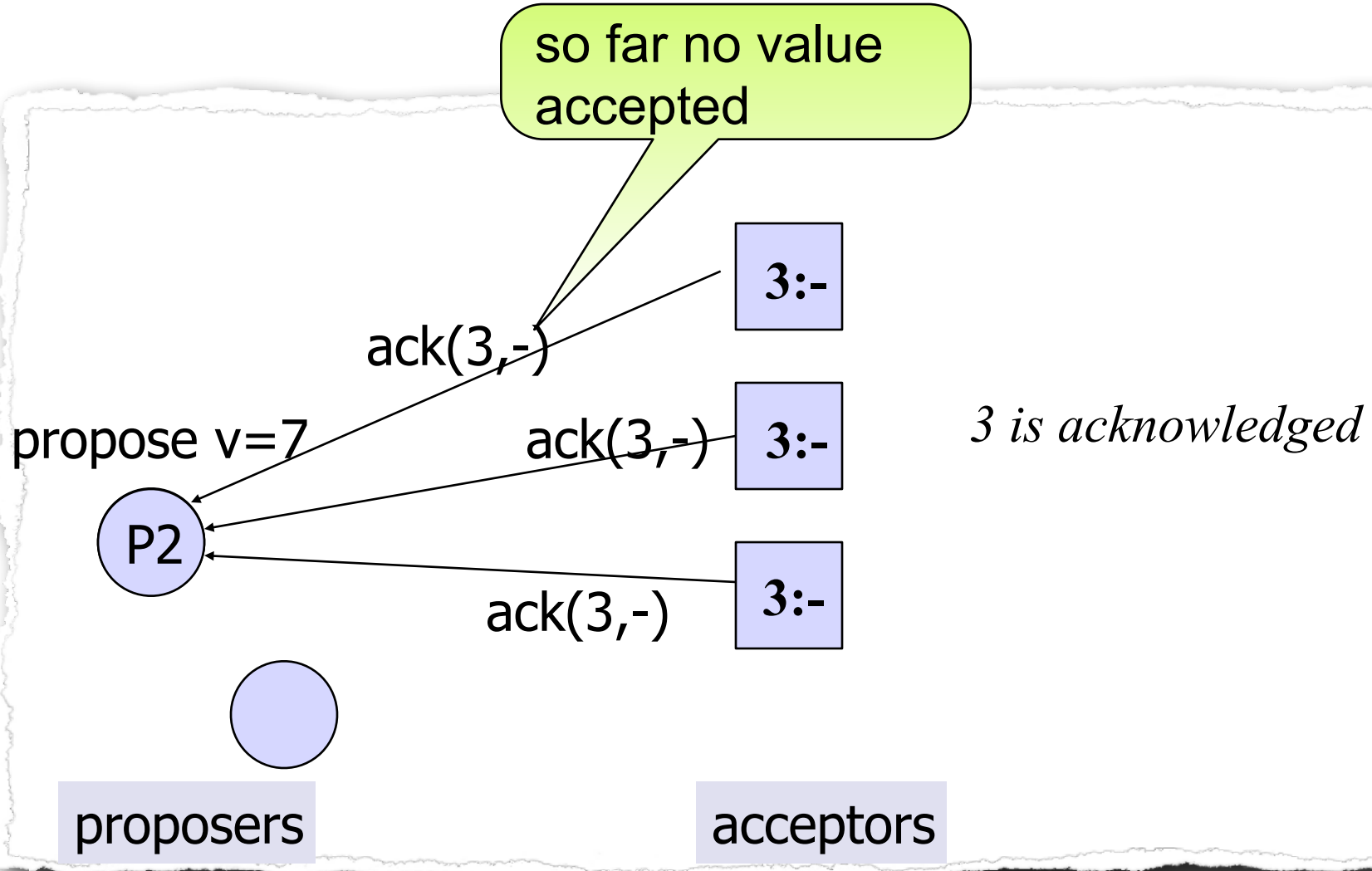
Preparation



Preparation



Preparation



3 is acknowledged

Phase 2:

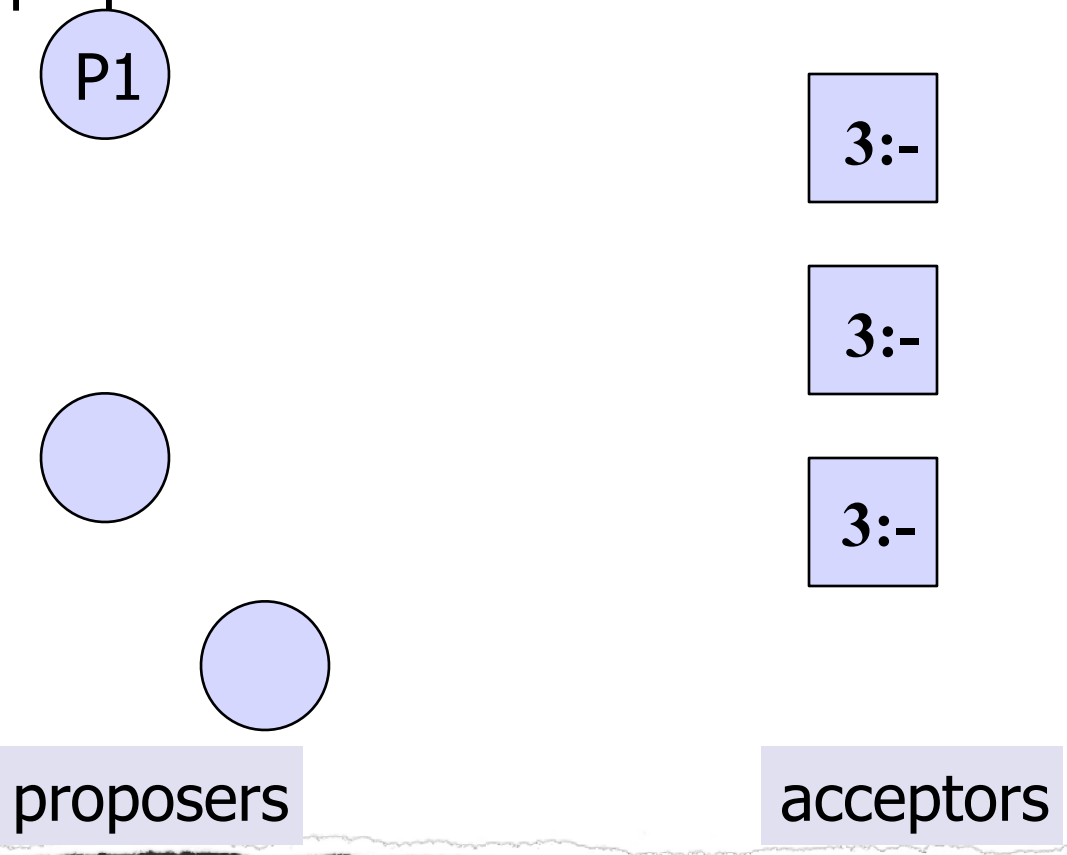
- Proposer waits for acks from majority of acceptors.
  - If exactly one **ack(n,(nx, vx))** contained a value,
    - sends **accept(n,vx)** to all acceptors
  - **If multiple acks contained value,**
    - send value with highest prev. proposal number nx
  - **Otherwise**,
    - sends **accept(n,own-value)** to all acceptors

Phase 2:

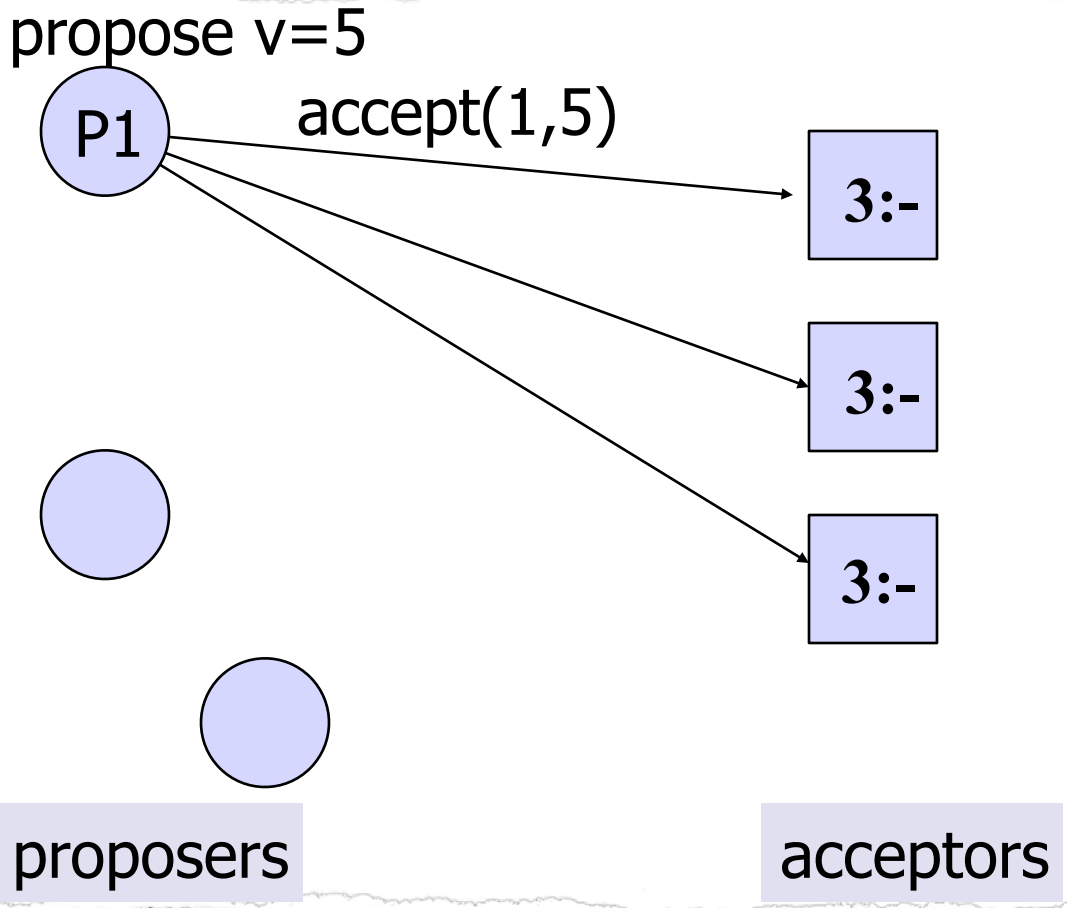
- Proposer waits for acks from majority of acceptors.
  - If exactly one **ack(n,(nx, vx))** contained a value,
    - sends **accept(n,vx)** to all acceptors
  - **If multiple acks contained value,**
    - send value with highest prev. proposal number nx
  - **Otherwise**,
    - sends **accept(n,own-value)** to all acceptors

- Upon receiving **accept(n,v)**
  - an acceptor accepts v **unless**
    - it has already received prepare(n') for some n' > n (i.e., N > n) or an accept(n',v') for some n' >= n (i.e., N >= n)
  - on acceptance, it sets:
    - N := n
    - nx := n, vx := v

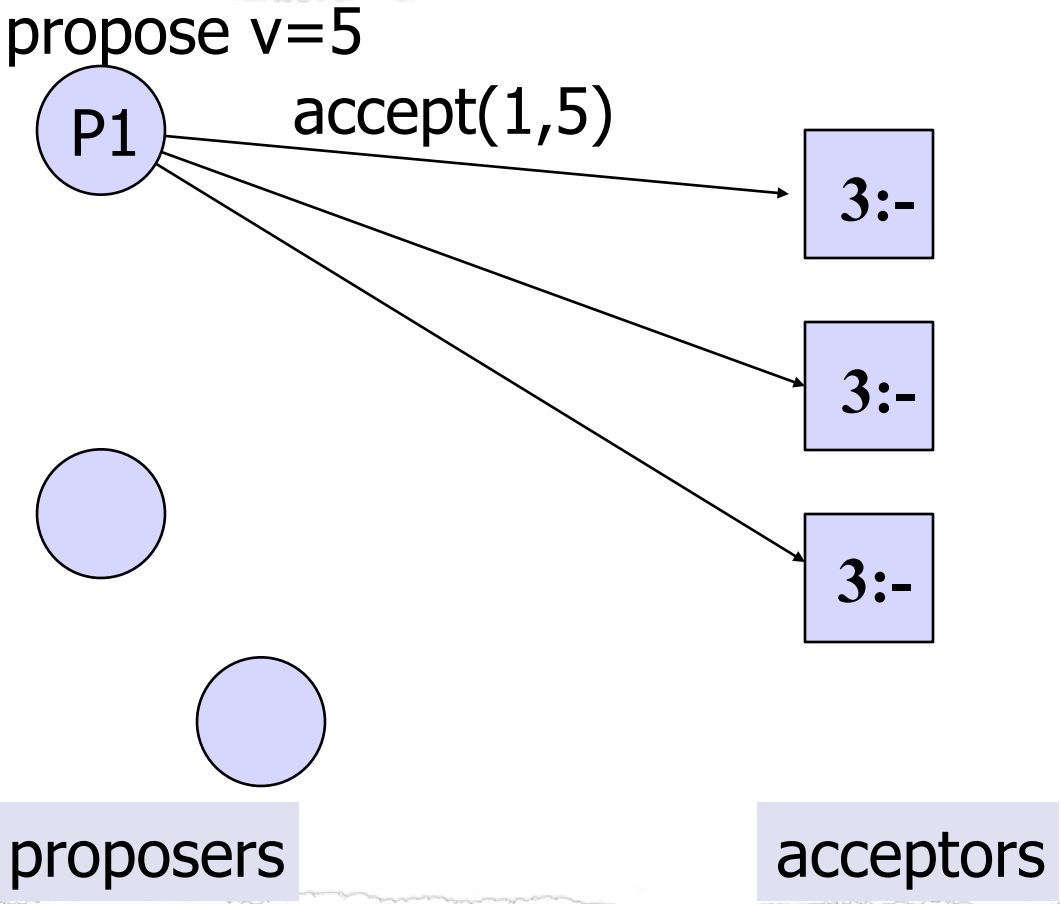
Phase 2:



Phase 2:

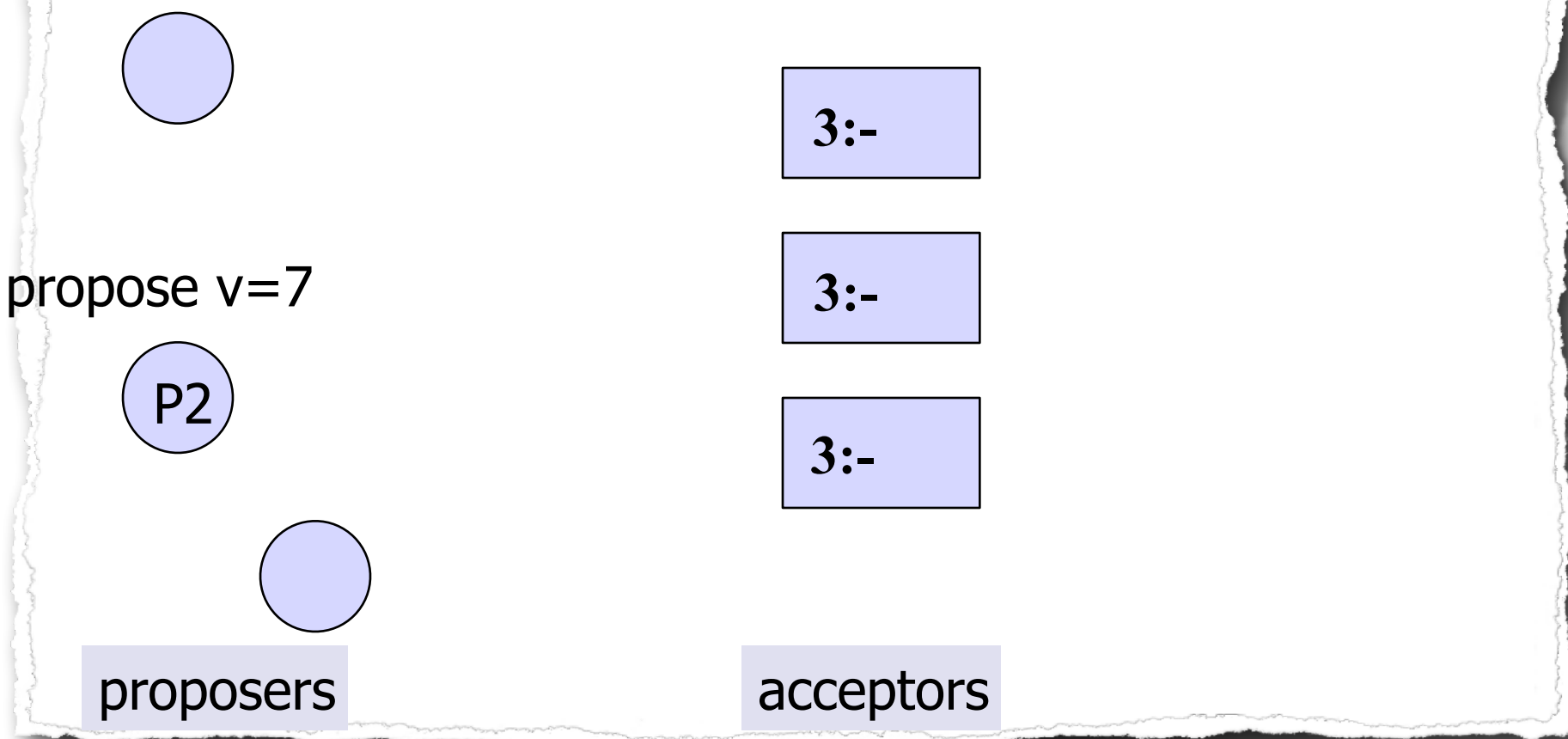


Phase 2:

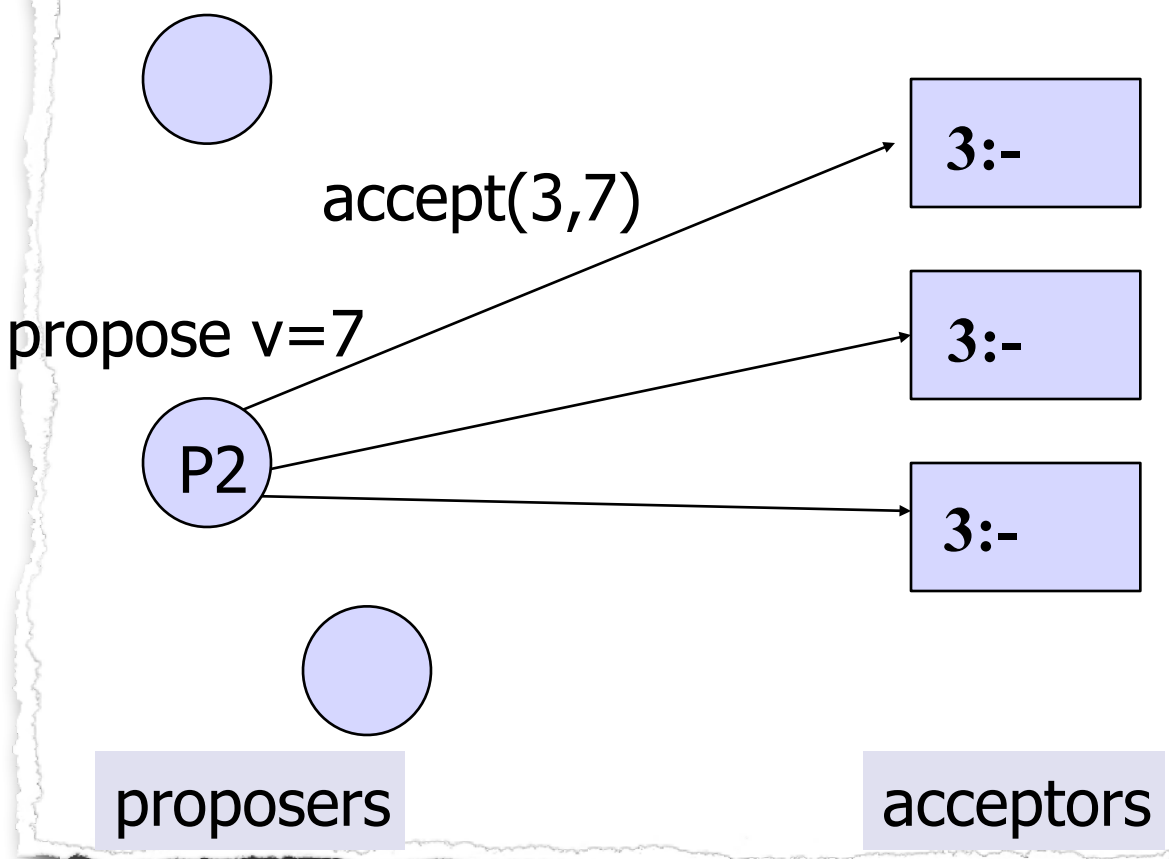


accept is ignored since proposal number is too small!

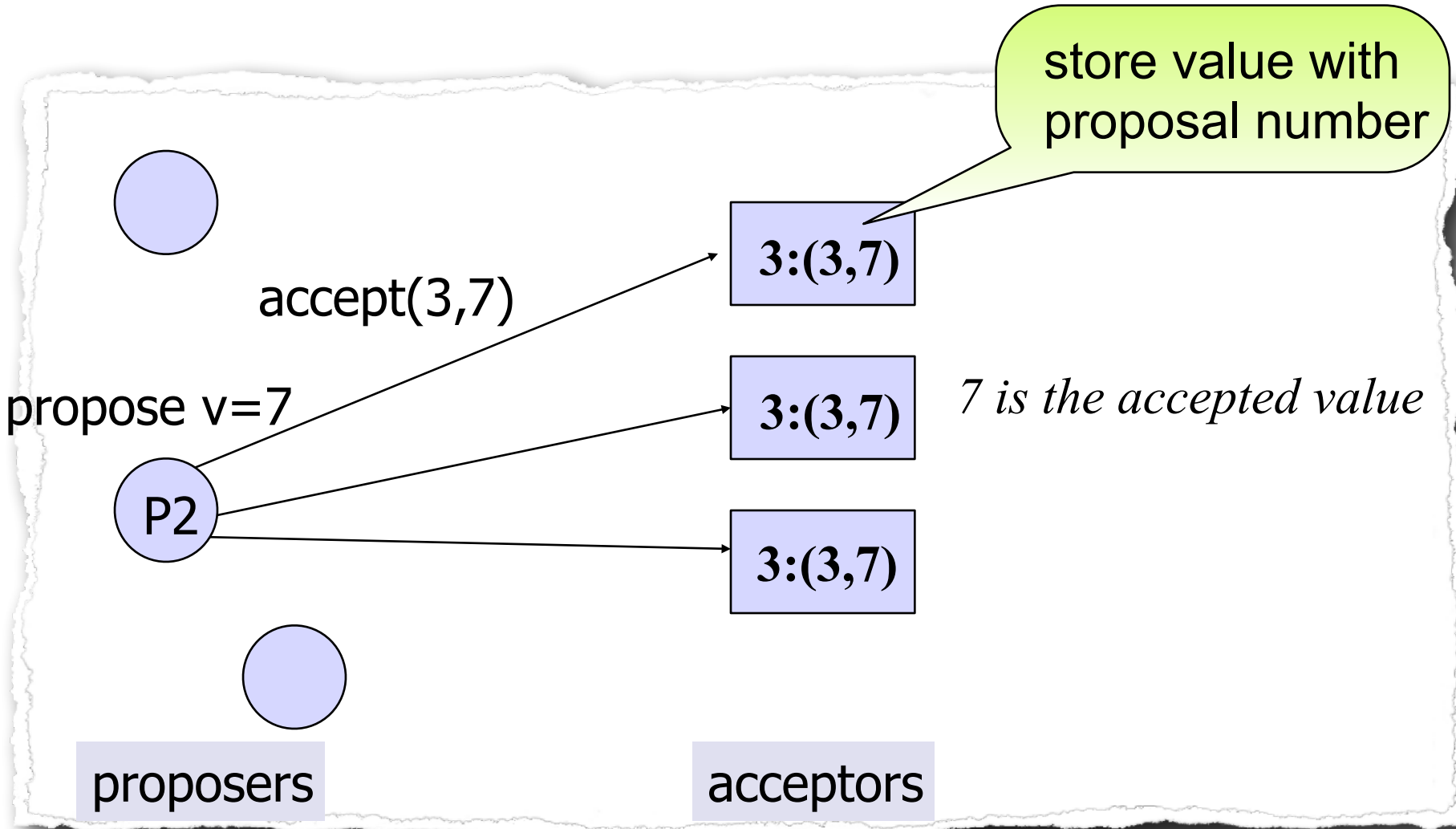
Phase 2:



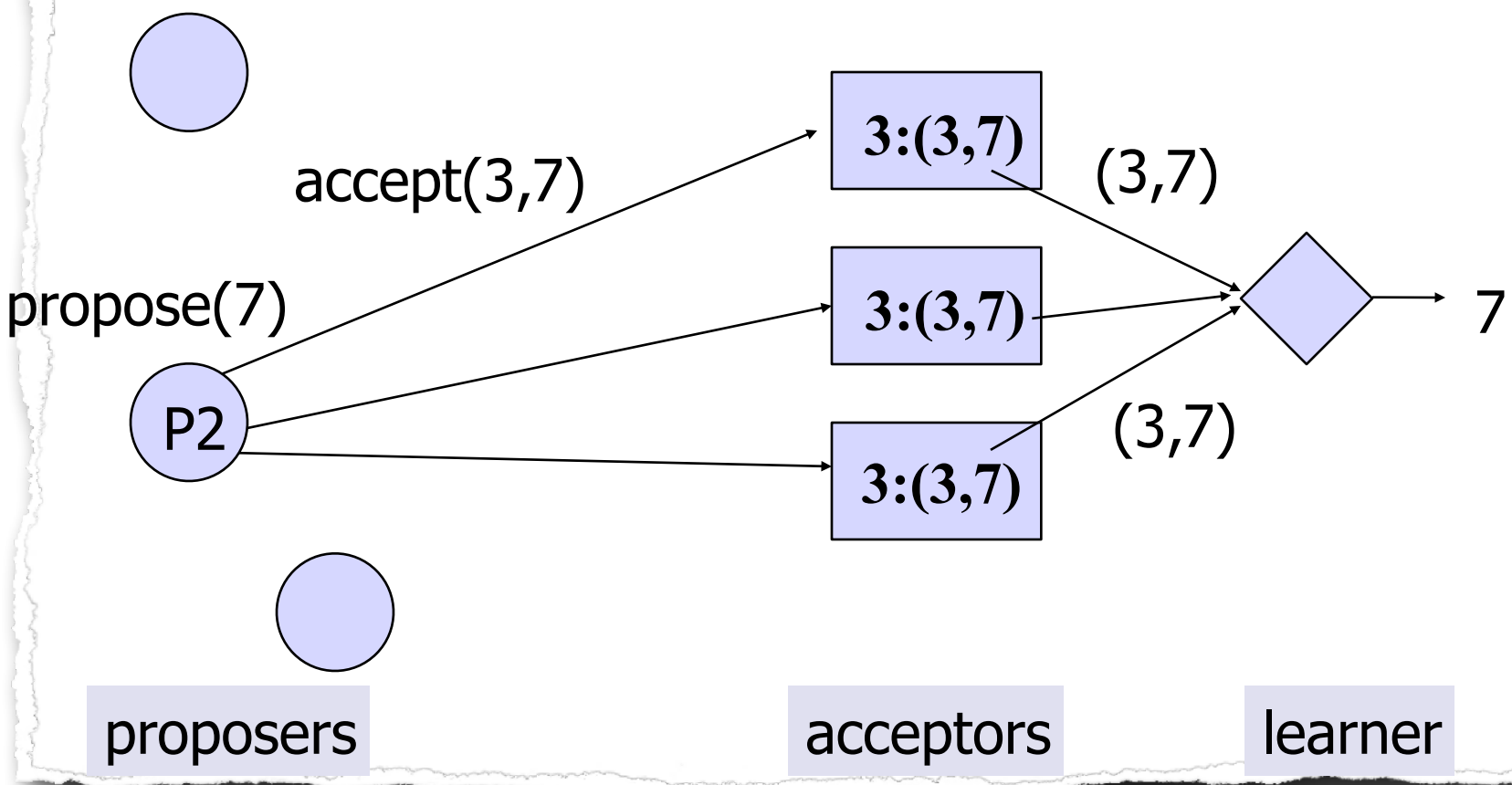
Phase 2:



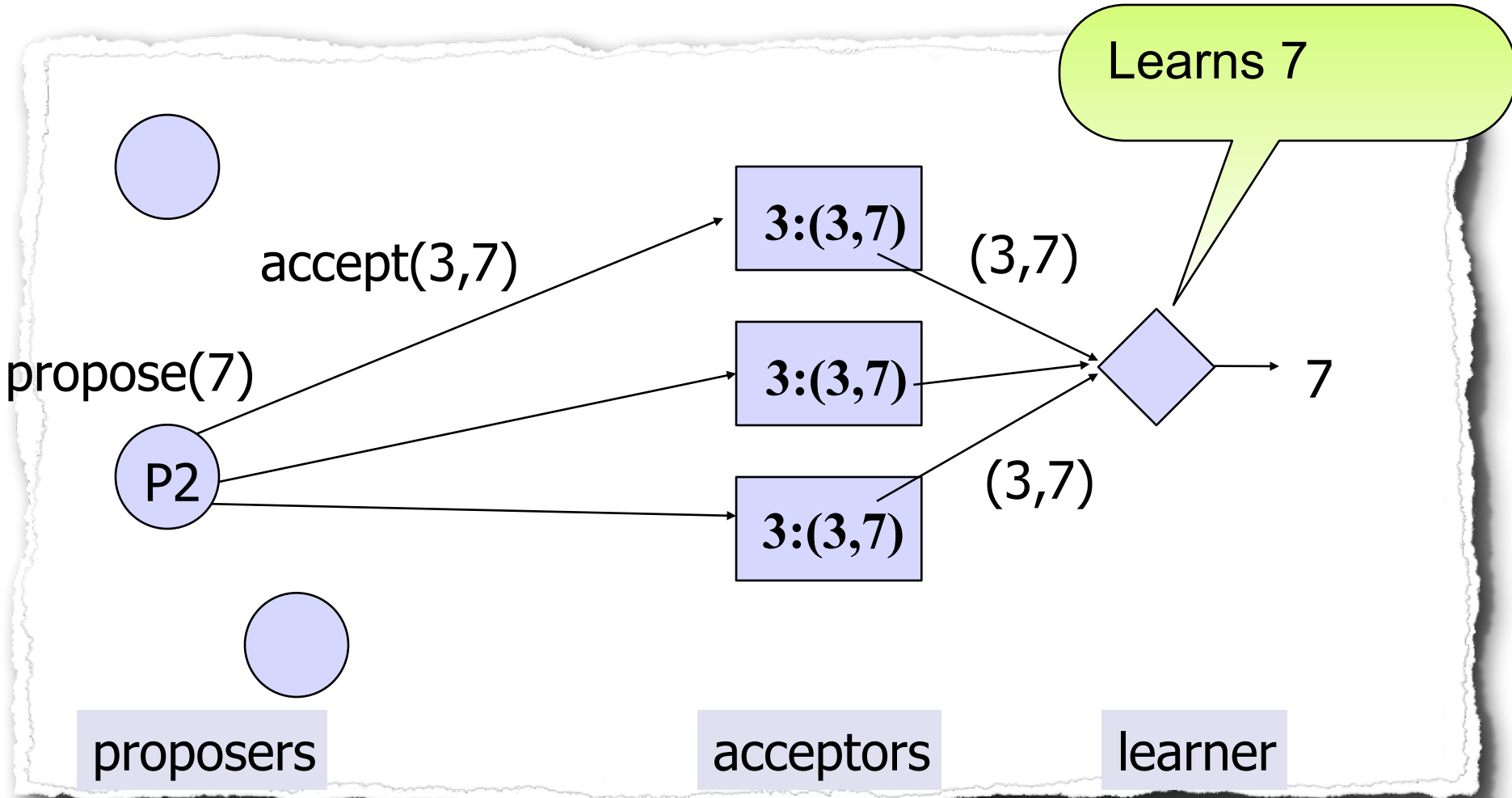
Phase 2:


store value with proposal number
accept(3,7)
propose v=7
P2
3:(3,7)
3:(3,7)
3:(3,7)
7 is the accepted value
proposers
acceptors

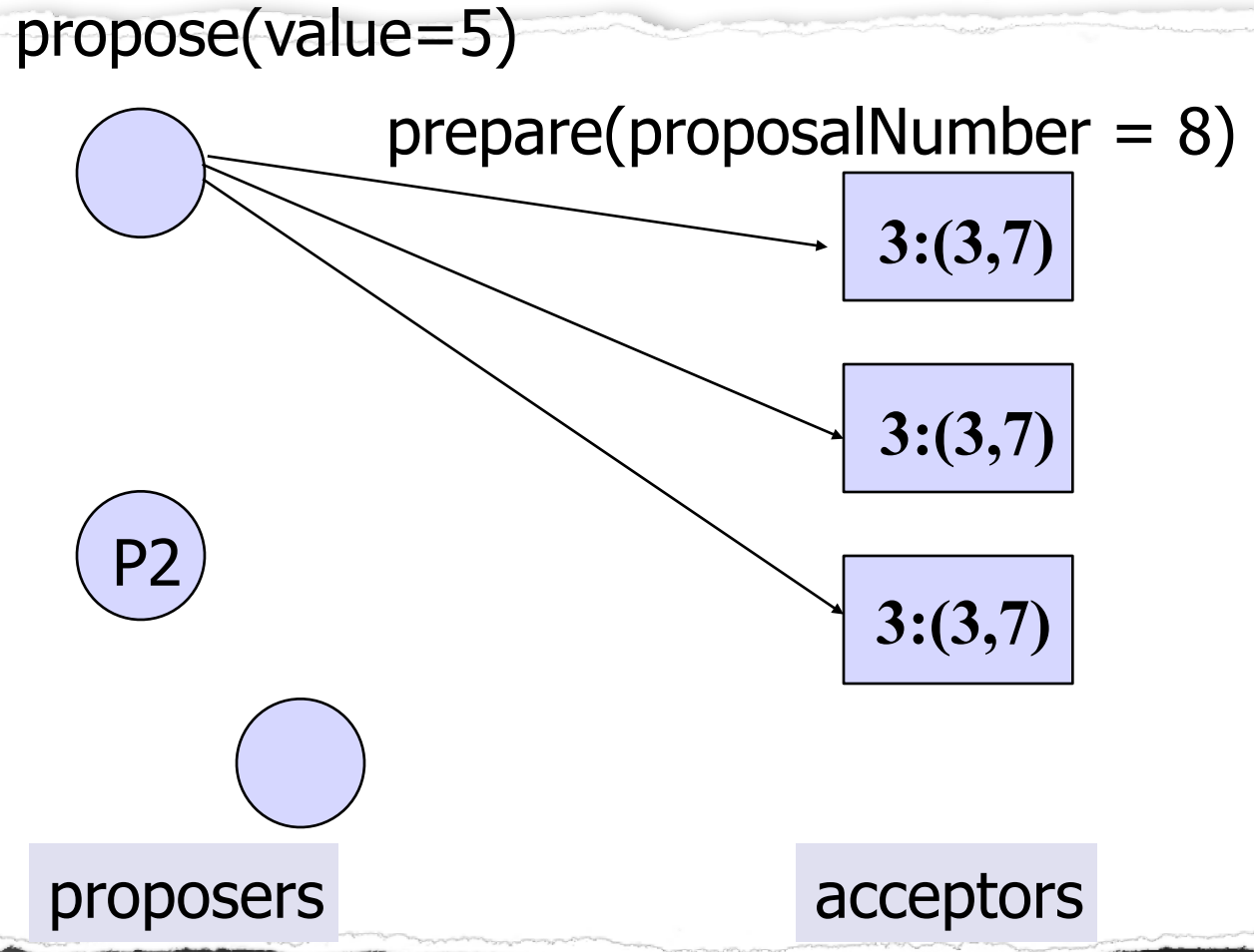
Phase 2:


accept(3,7)
propose(7)
P2
3:(3,7)
3:(3,7)
3:(3,7)
(3,7)
(3,7)
7
proposers
acceptors
learner

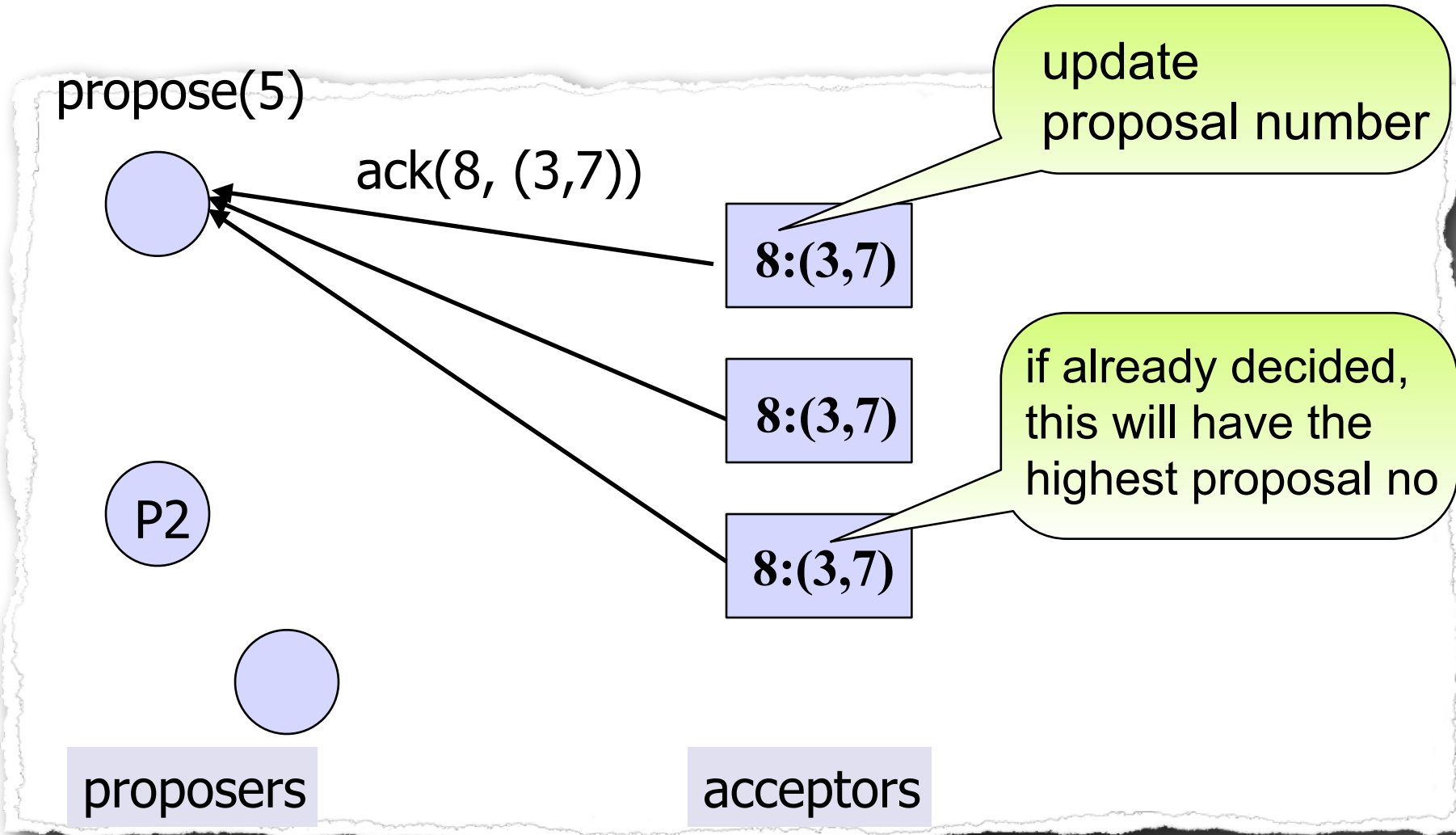
Phase 2:


Learns 7
accept(3,7)
propose(7)
P2
3:(3,7)
3:(3,7)
3:(3,7)
(3,7)
(3,7)
7
proposers
acceptors
learner

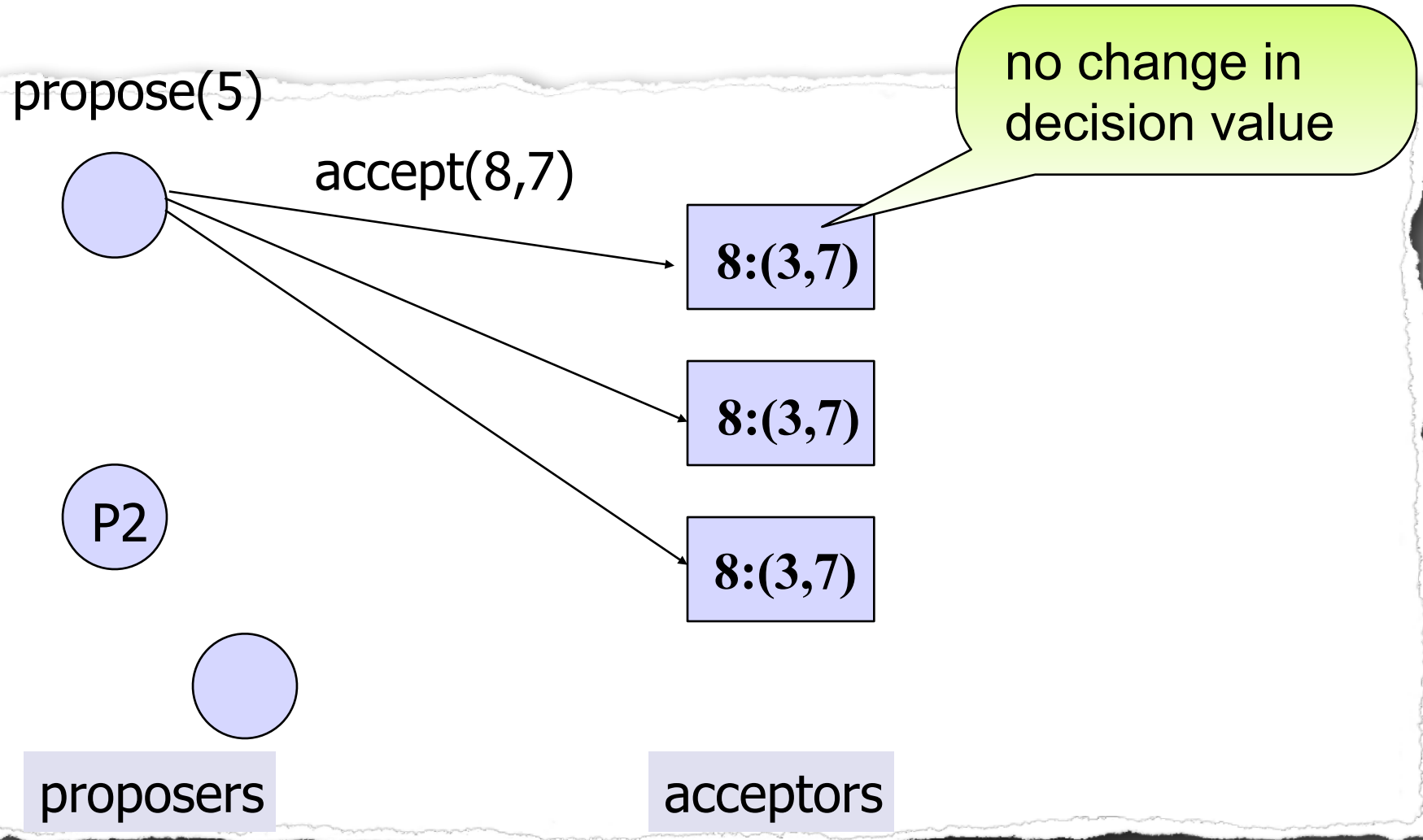
Phase 1:


propose(value=5)
prepare(proposalNumber = 8)
3:(3,7)
3:(3,7)
3:(3,7)
P2
proposers
acceptors

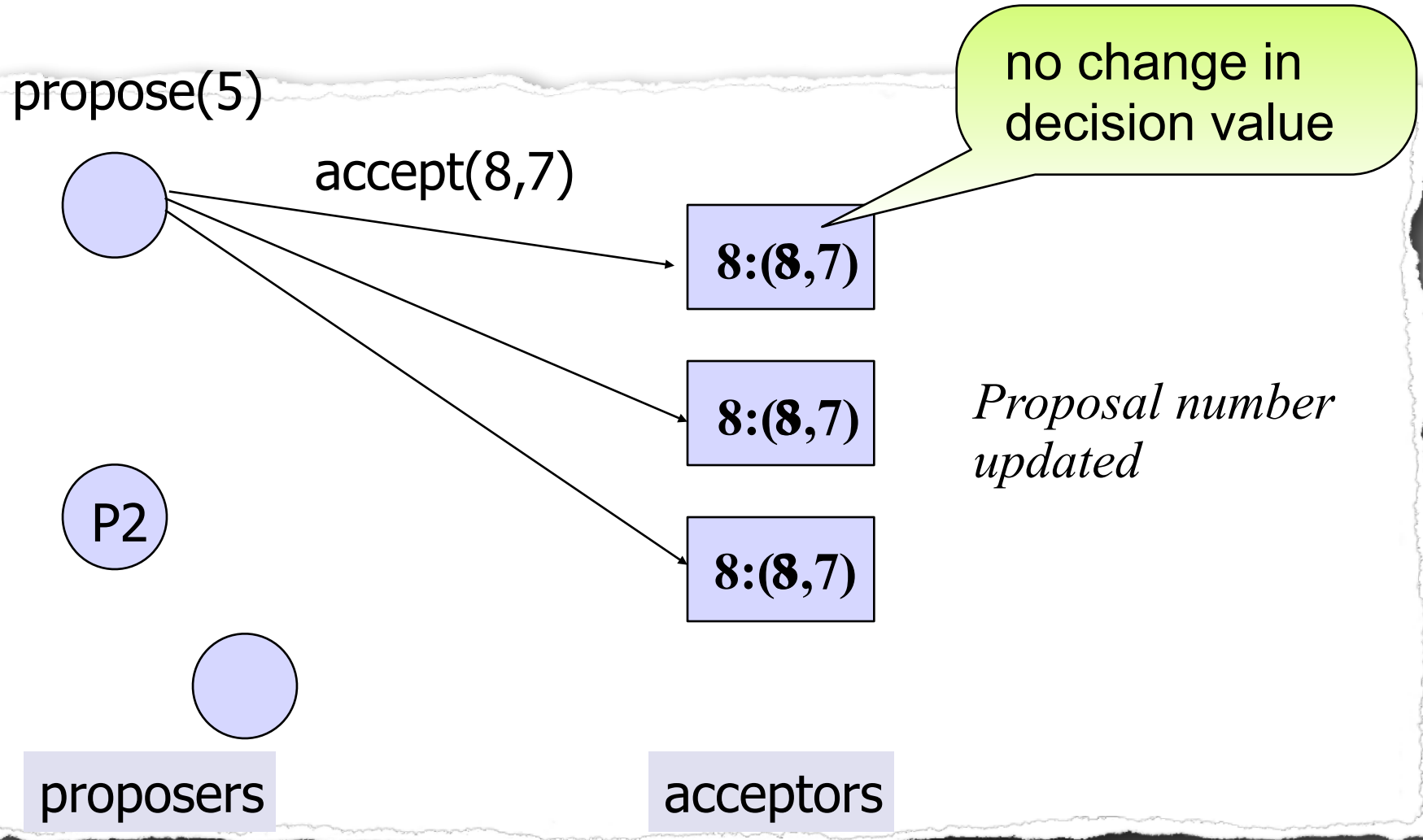
Phase 1:


propose(5)
ack(8, (3,7))
update
proposal number
8:(3,7)
8:(3,7)
8:(3,7)
if already decided,
this will have the
highest proposal no
P2
proposers
acceptors

Phase 1:


propose(5)
accept(8,7)
no change in decision value
8:(3,7)
8:(3,7)
8:(3,7)
P2
proposers
acceptors

Phase 1:

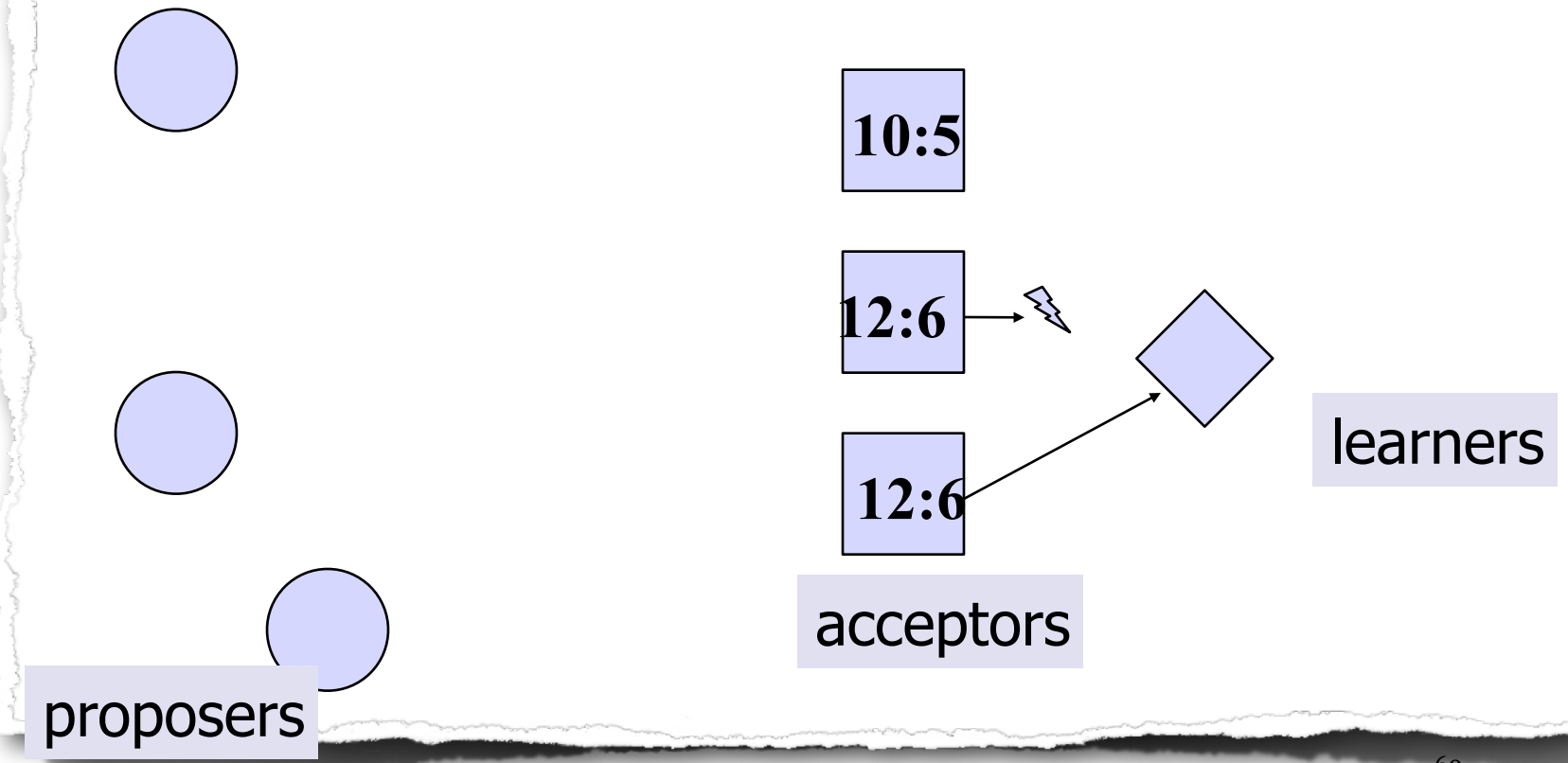

propose(5)
accept(8,7)
no change in decision value
8:(8,7)
8:(8,7)
8:(8,7)
P2
Proposal number updated
proposers
acceptors

Learning a Chosen Value (I)

- A learner must find out that a proposal has been accepted by a majority of acceptors.
  - Each acceptor sends a message to each learner whenever it accepts a proposal.
  - When a learner receives the same message from a majority of acceptors, then it knows that the value in these messages was chosen.
  - Can have a *distinguished learner* (or set of such learners) that take on this role, and can inform other learners when a value has been chosen.

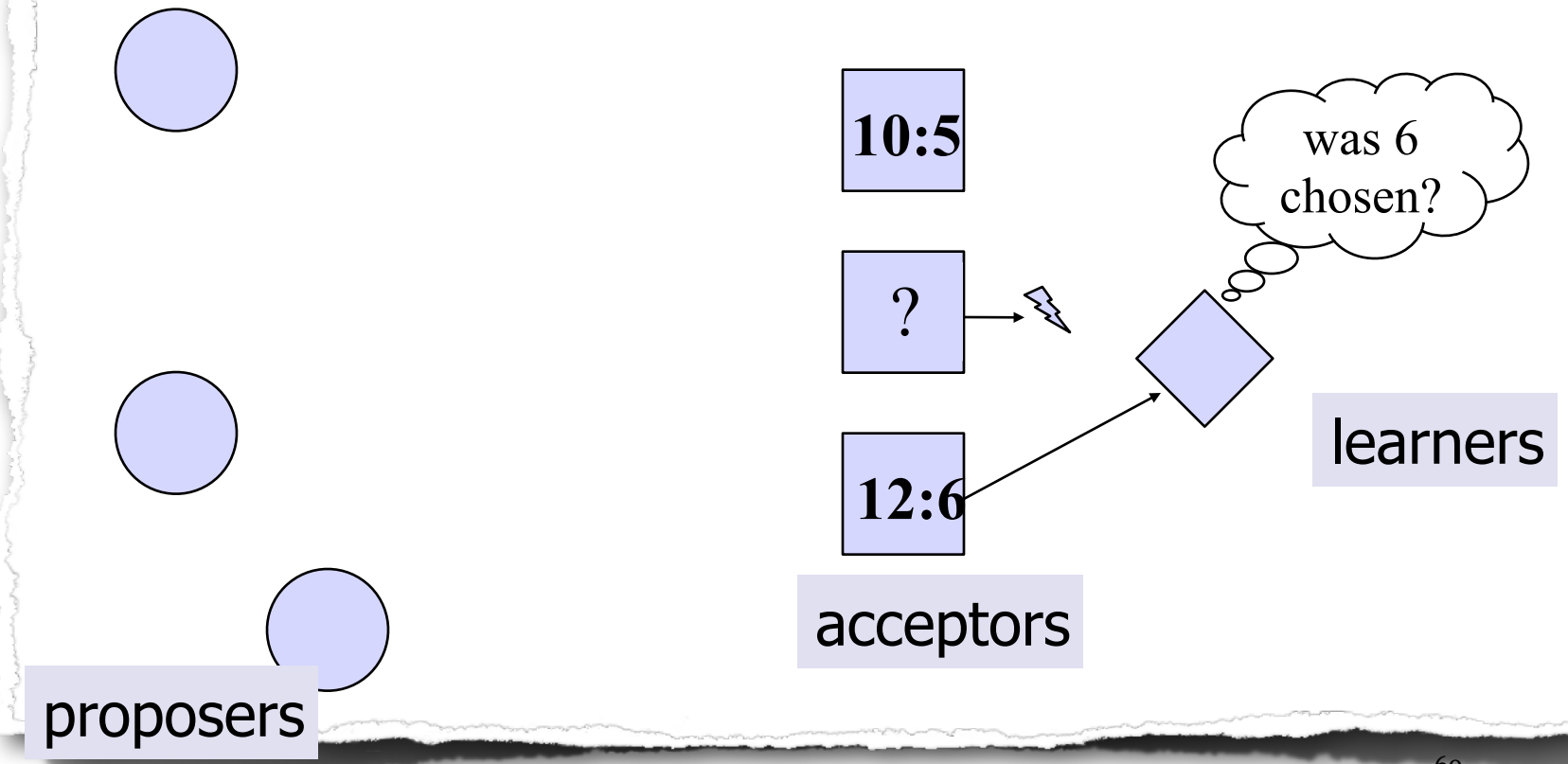
Learning a Chosen Value

- Due to message loss, a learner may not know that a value has been chosen.



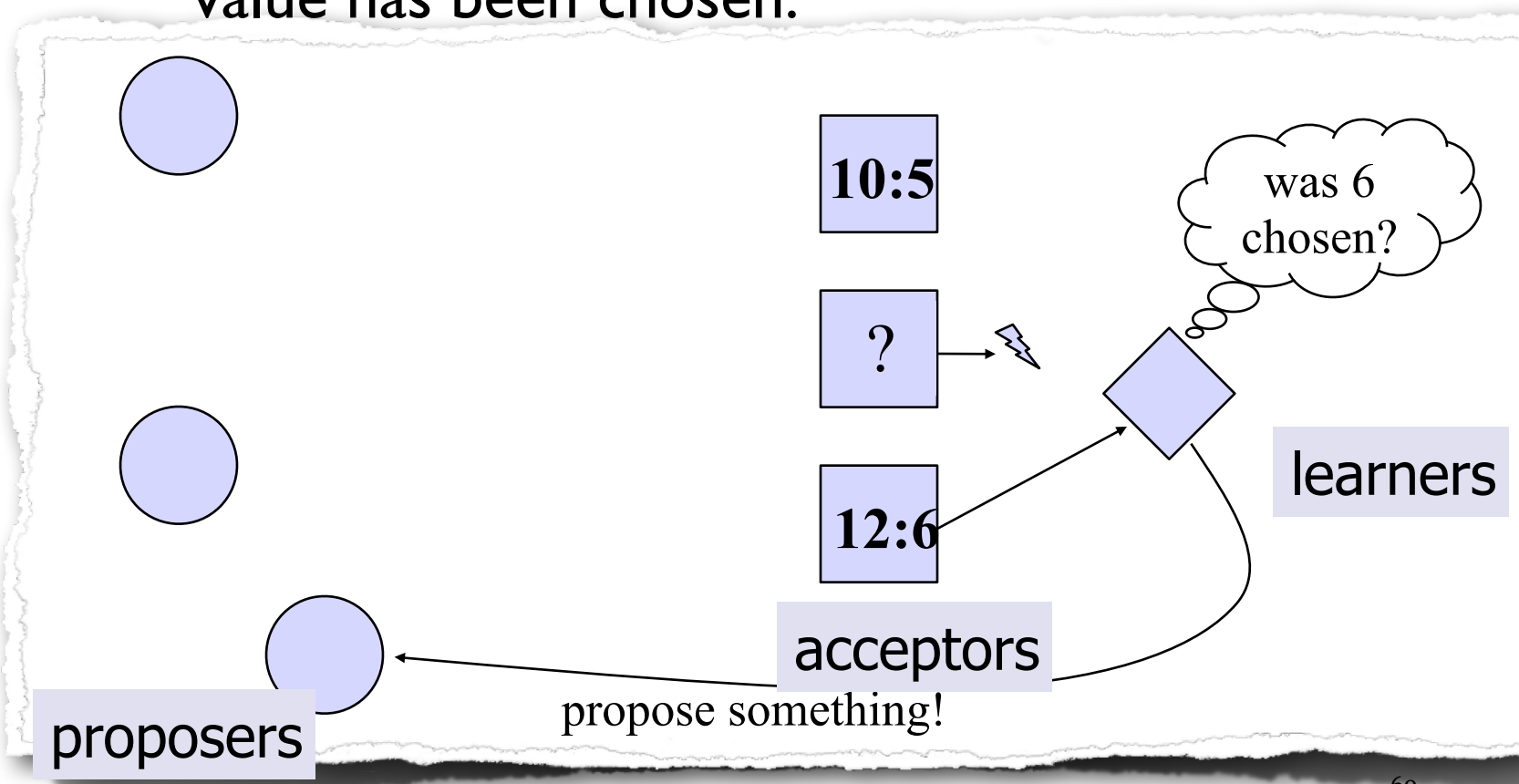
Learning a Chosen Value

- Due to message loss, a learner may not know that a value has been chosen.



Learning a Chosen Value

- Due to message loss, a learner may not know that a value has been chosen.



Why does it work?

- safety -

Validity

- *accept* message either
  - contains local value of proposer, or
  - value returned by acceptor (via ack)
- Value returned by acceptor via ack
  - is a value received via accept
- Hence,
  - chosen value was proposed by a proposer

Agreement

- To show:
  - all learners learn the same value!
- Intuition:
  - say, learners L1, L2 learn values v1, v2
  - this means, that a majority of acceptors
    - accepted messages *accept*(n1, v1) & *accept*(n2,v2)
  - we know
    - some proposer P1, sent *accept*(n1,v1)
    - some proposer P2, sent *accept*(n2,v2)
  - if (n1 == n2) then
    - majority of acceptors acked n1(=n2) to P1 **and** P2
    - this means that P1 = P2 because each proposal number is acked at most once and majorities overlap!

- An acceptor accepts an accept message with proposal numbered n
  if and only if it has not responded to a prepare message with a number n' > n
  and has not accepted an accept message with a number n‘ >= n.

Agreement…

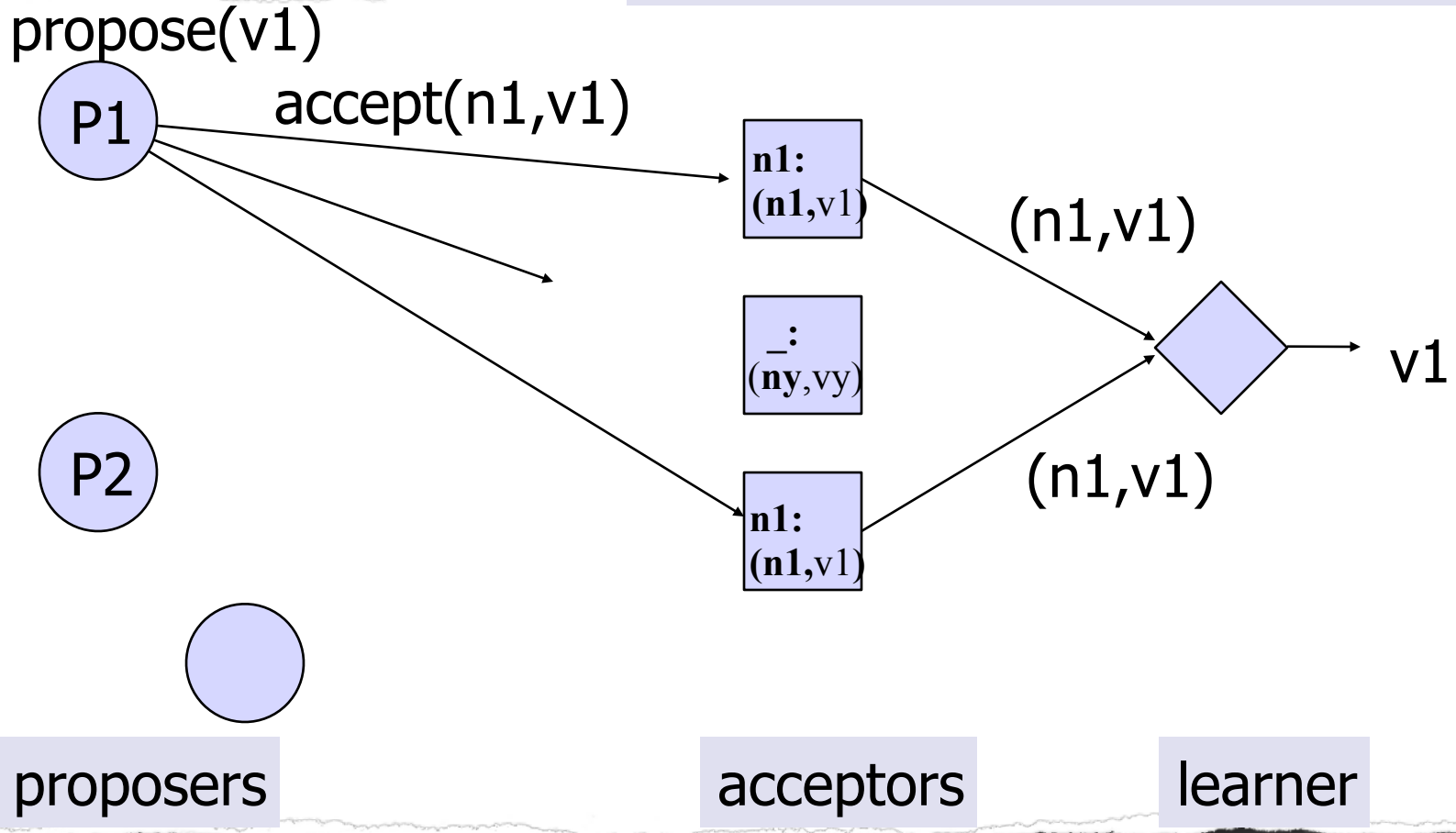
- Ok, assume n1 < n2:
  - proposer P1 got ack(n1, v1') from majority
  - proposer P2 got ack(n2, v2') from majority
- And
  - proposer P1 succeeded to send accept to at least one acceptor A1 such that P2 got
    - ack from A1 after A1 received accept(n1,v1),
    - A1 will reply with ack(n2,(nx,v1)), and
    - nx is highest proposal number in acks
  - Why??

Why…

- **P1's prepare(n1) was ack'ed by majority**
  - this means that no other proposer was able to get an ack(n,_) with n > n1 from maj beforehand
- Hence,
  - the next proposer Px that gets an ack
    - from a majority, will get (n1, v1)
  - will hence send an accept with
    - value v1
  - and all accepts will contain v1 from now on

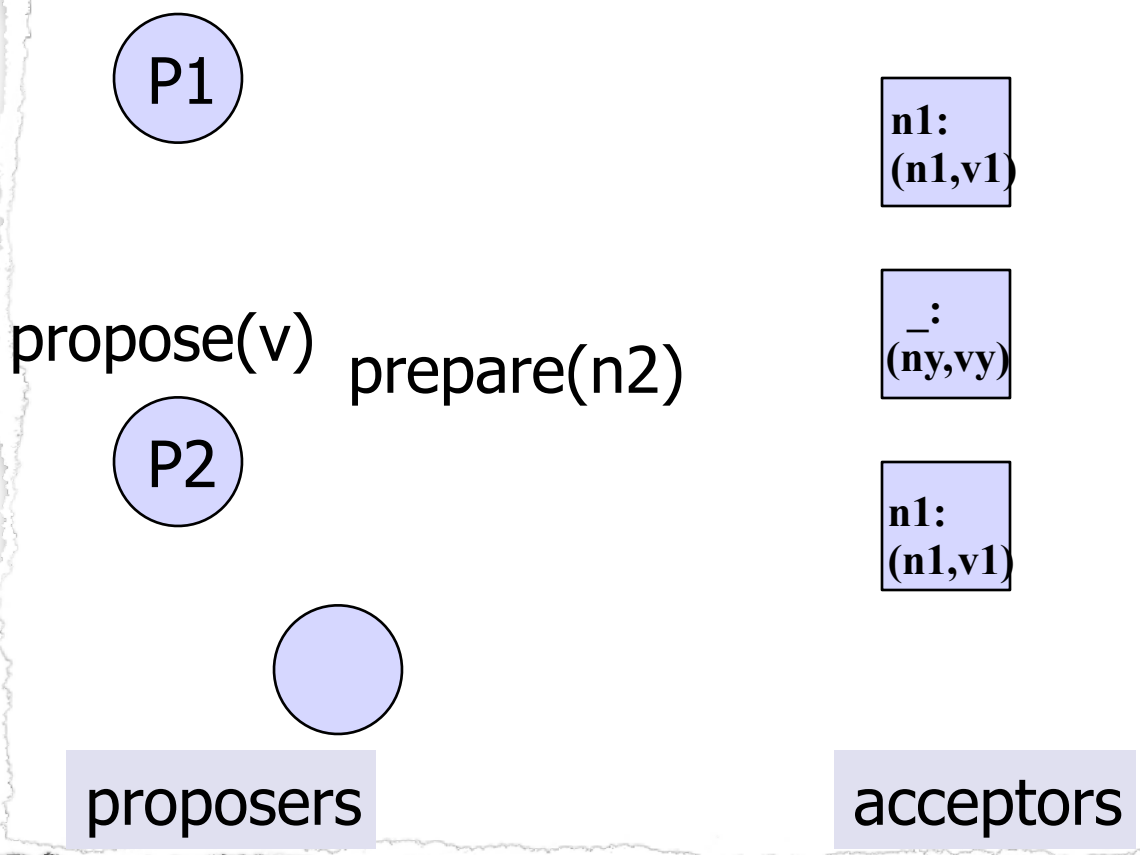
In pictures

accept(n1,v1) accepted by majority
=>
ny *< n1 because otherwise for majority of acceptors would N > n1*



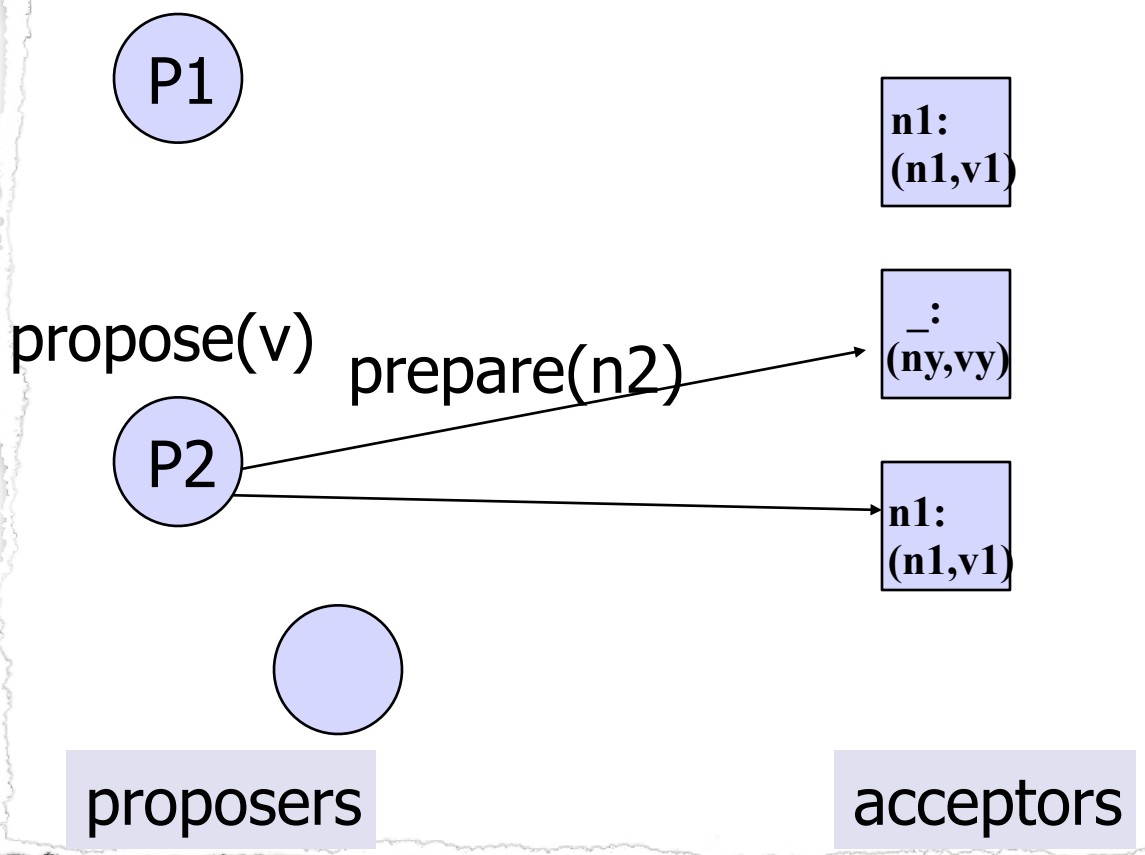
In pictures

accept(n1,v1) accepted by majority
=>
ny < n1 because otherwise for majority of acceptors would N > n1



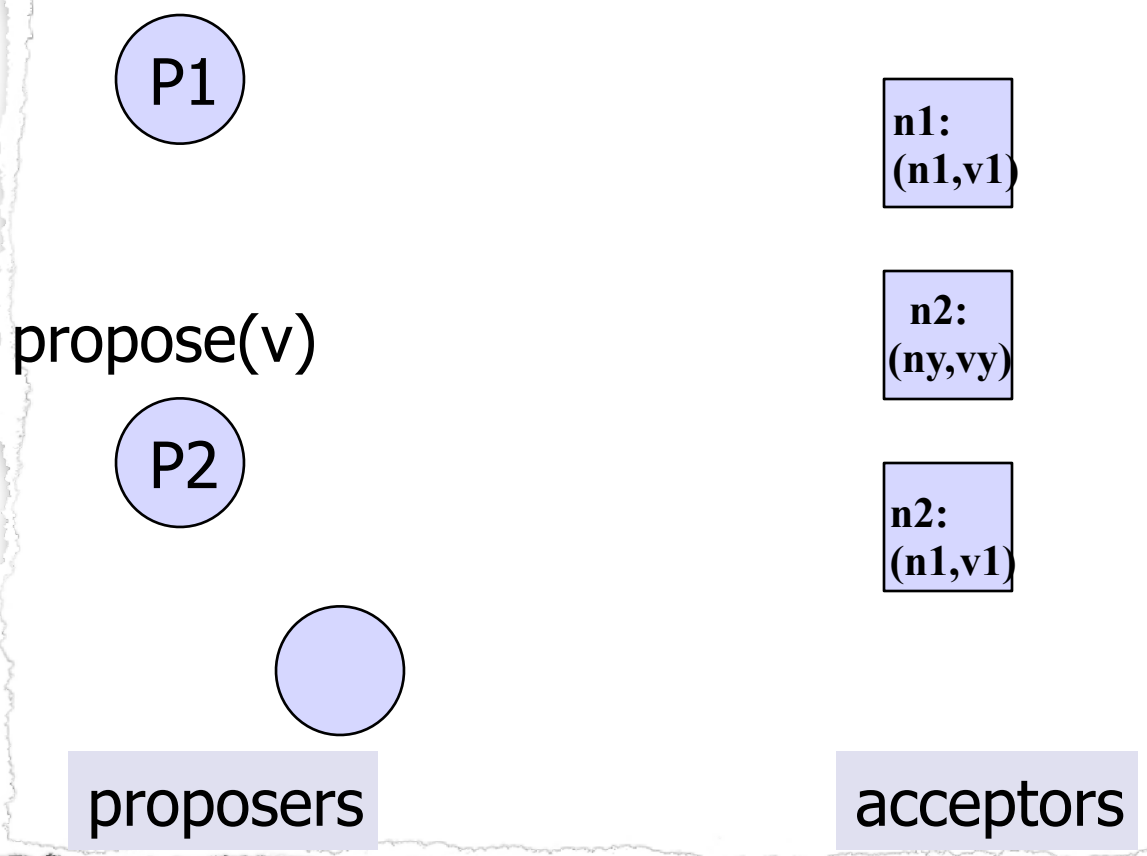
In pictures

accept(n1,v1) accepted by majority
=>
ny < n1 because otherwise for majority of acceptors would N > n1



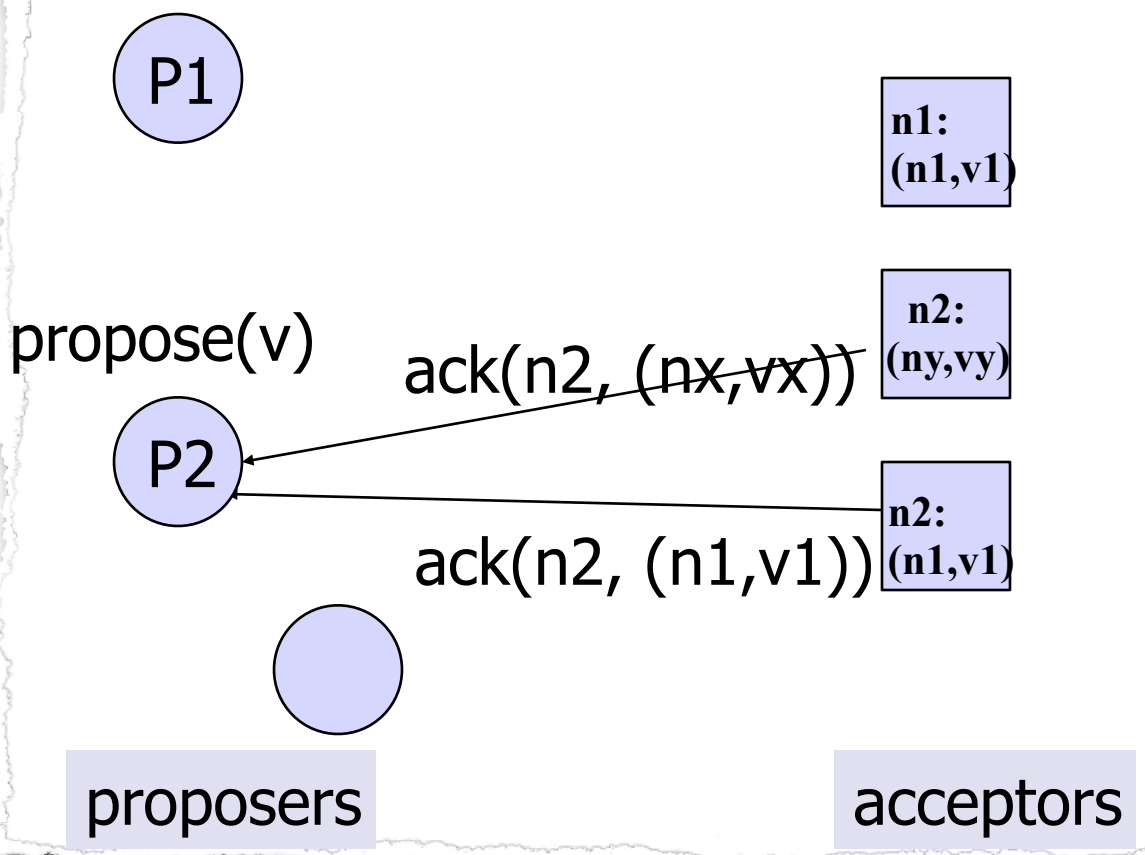
In pictures

accept(n1,v1) accepted by majority
*=> **ny < n1***



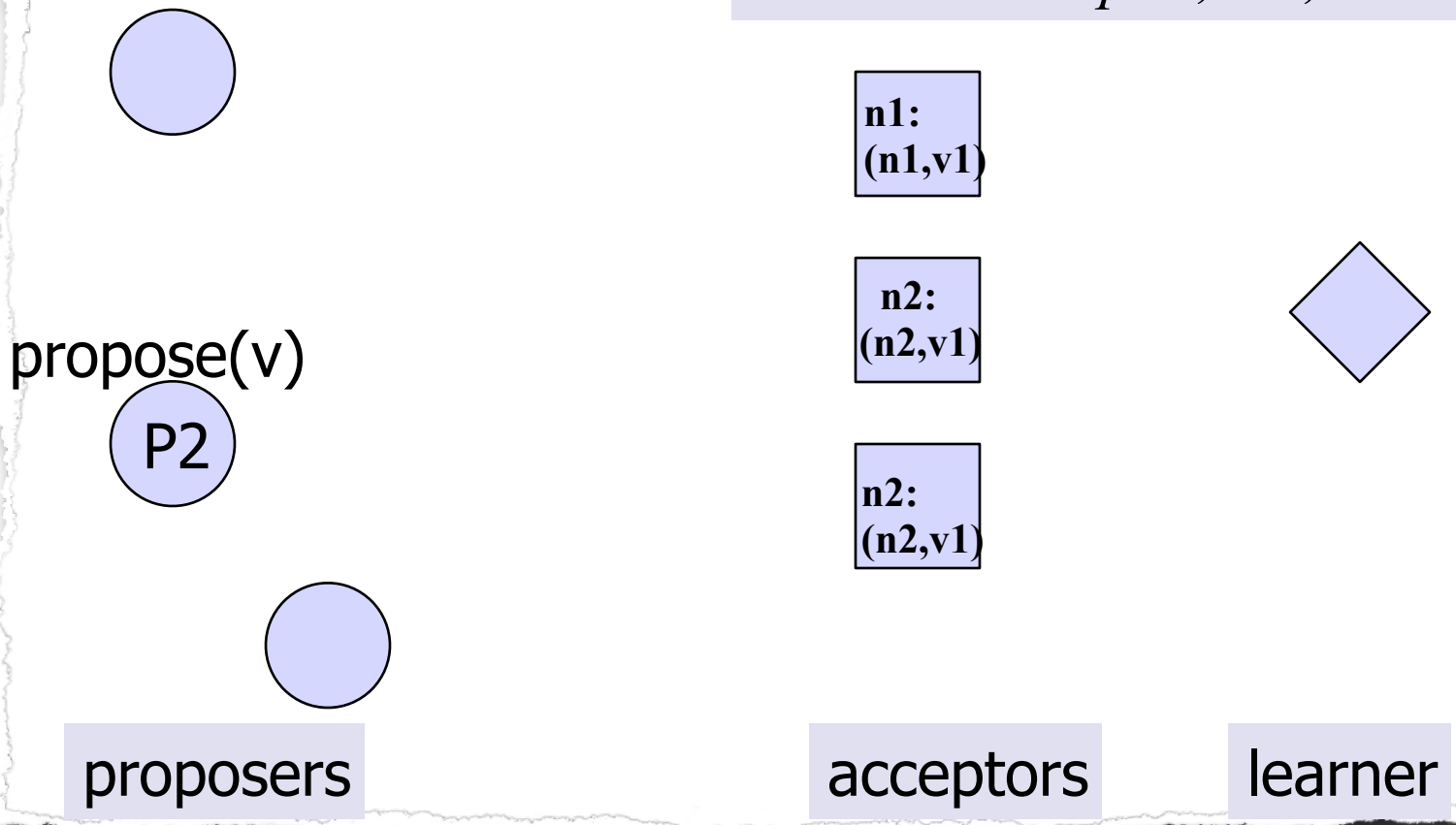
In pictures

accept(n1,v1) accepted by majority
=> ***ny < n1***



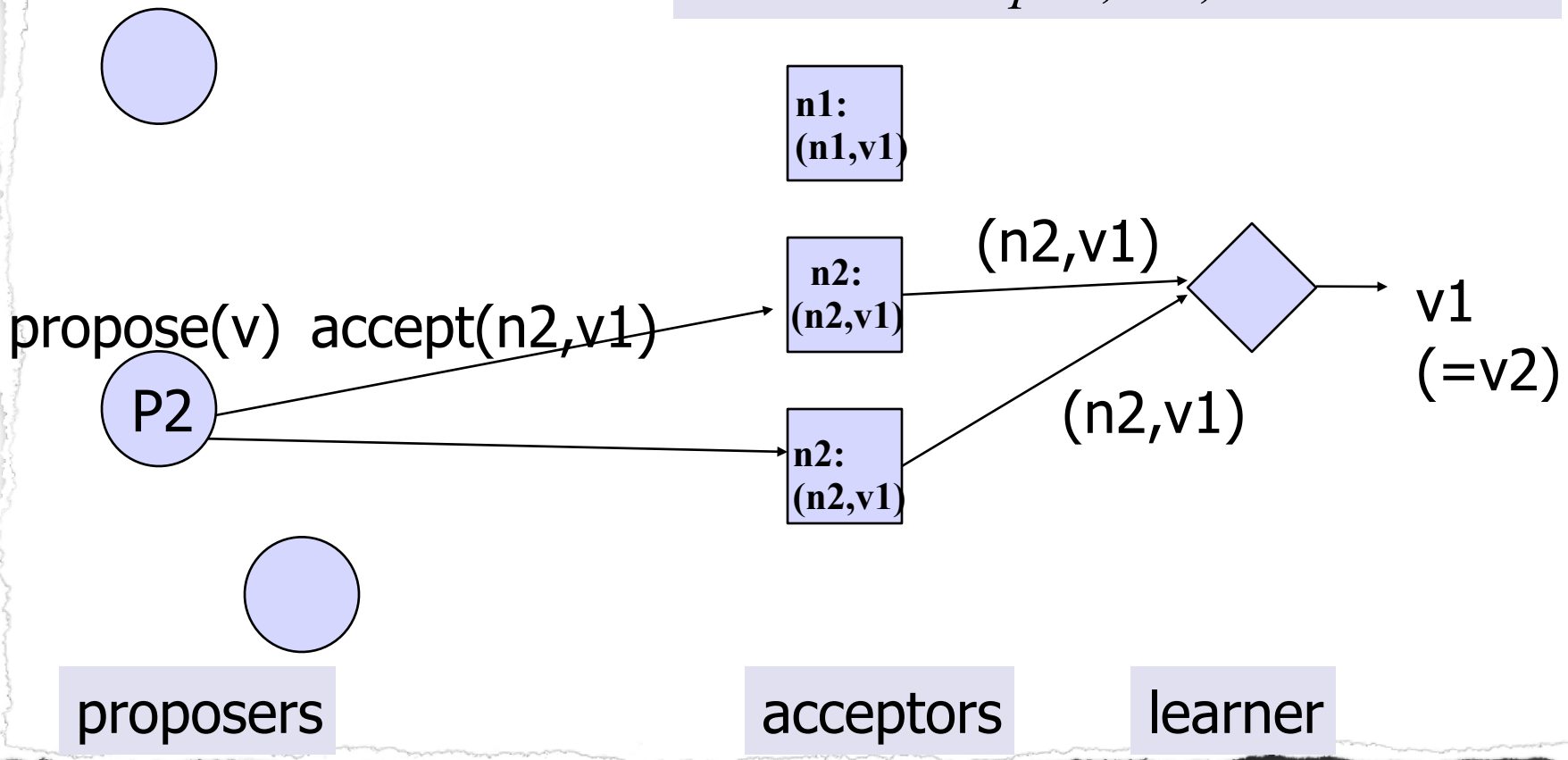
In pictures

accept(n2,v2) accepted by majority && n2 > n1
=>
P2 received ack(n2, (n1,v1)) from at least one acceptor, i.e., v2 = v1.



In pictures

accept(n2,v2) accepted by majority && n2 > n1
=>
P2 received ack(n2, (n1,v1)) from at least one acceptor, i.e., v2 = v1.



Termination?

Races

Races between proposers:

- processes race each other by increasing proposal numbers

Approach:

- assign a primary proposer
- elect a new one if old primary is slow / crashed

Conclusion

Consensus is a fundamental problem

- in distributed systems

Impossible to solve in asynchronous distributed systems

BUT

- practical solutions like PAXOS exist