

# Distributed Systems

Instructor: Ajay Kshemkalyani

## Failure Detectors

Presented by,

Archana  
Bharath  
Lakshmi

# Failure Detector

- **Failure detector** is an application that is responsible for detection of node failures or crashes in a distributed system.
- A **failure detector** is a distributed oracle that provides hints about the operational status of other processes

# Why Failure Detectors

- The design and verification of ***fault-tolerant*** distributed system is a difficult problem.
- The ***detection of process failures*** is a crucial problem, system designers have to cope with in order to build fault tolerant distributed platforms

# Synchronous Vs Asynchronous

- A distributed system is synchronous if:
  - there is a known upper bound on the transmission delay of messages
  - there is a known upper bound on the processing time of a piece of code
- A distributed system is asynchronous if:
  - there is no bound on the transmission delay of messages
  - there is no bound on the processing time of a piece of code

# Why Failure Detectors cont...

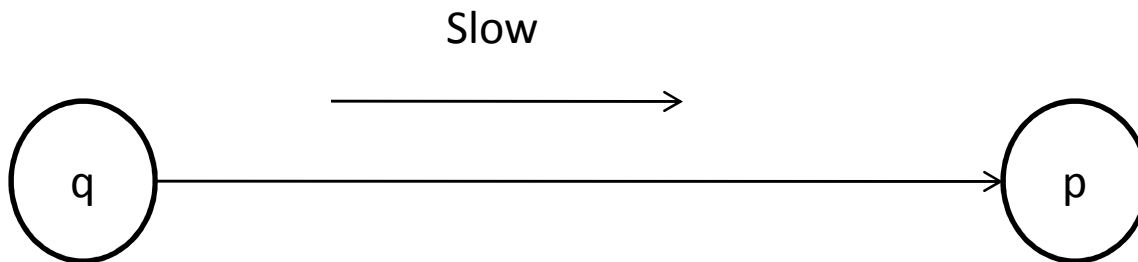
- **To stop waiting or not to stop waiting?**
- Unfortunately, it is impossible to distinguish with certainty a ***crashed process from a very slow process*** in a purely asynchronous distributed system.
- Look at two major problems
  - Consensus
  - Atomic Broadcast

## Liveness & Safety

- The problem can be defined with a **safety** and a **liveness** property.
- The safety property stipulates that ***“nothing bad ever happens”***
- The liveness property stipulates that ***“something good eventually happens”***

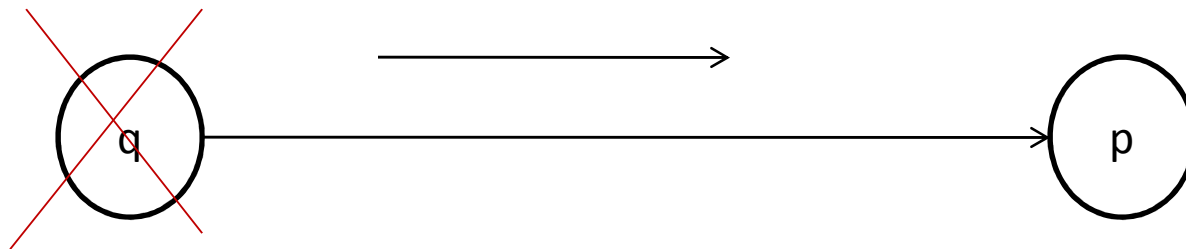
# 'q' not crashed

- The message from  $q$  to  $p$  is *only* very slow.
- Assuming that 'q' has crashed will violate the **safety** property



# *'q' has crashed*

- To prevent the bad previous scenario from occurring, p must wait until it gets q's message.
- It is easy to see that p will wait forever, and the **liveness** property of the application will never be satisfied



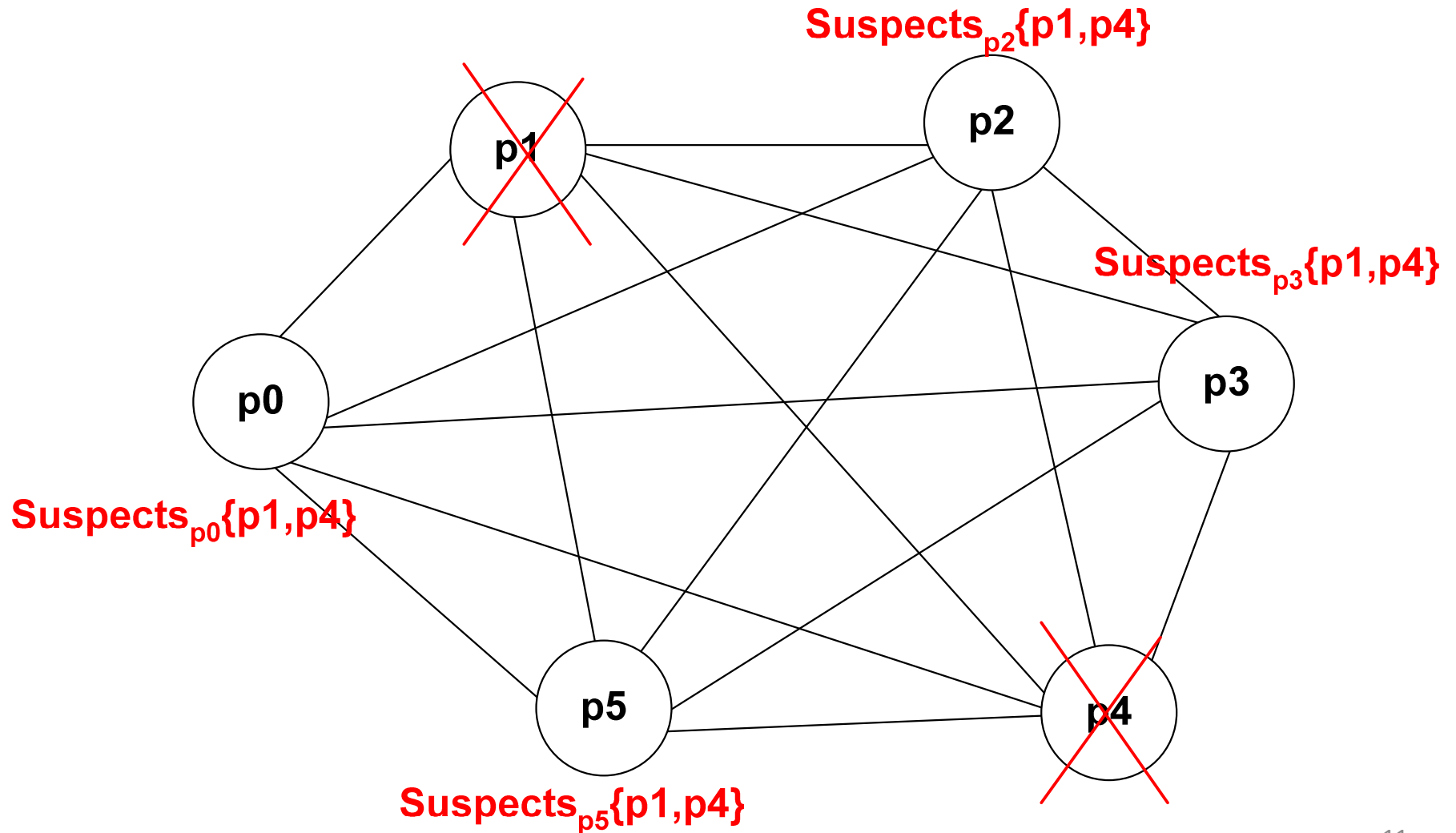
# Characterizing Failure Detectors

- Completeness
  - Suspect every process that actually crashes
- Accuracy
  - Limit the number of correct processes that are suspected

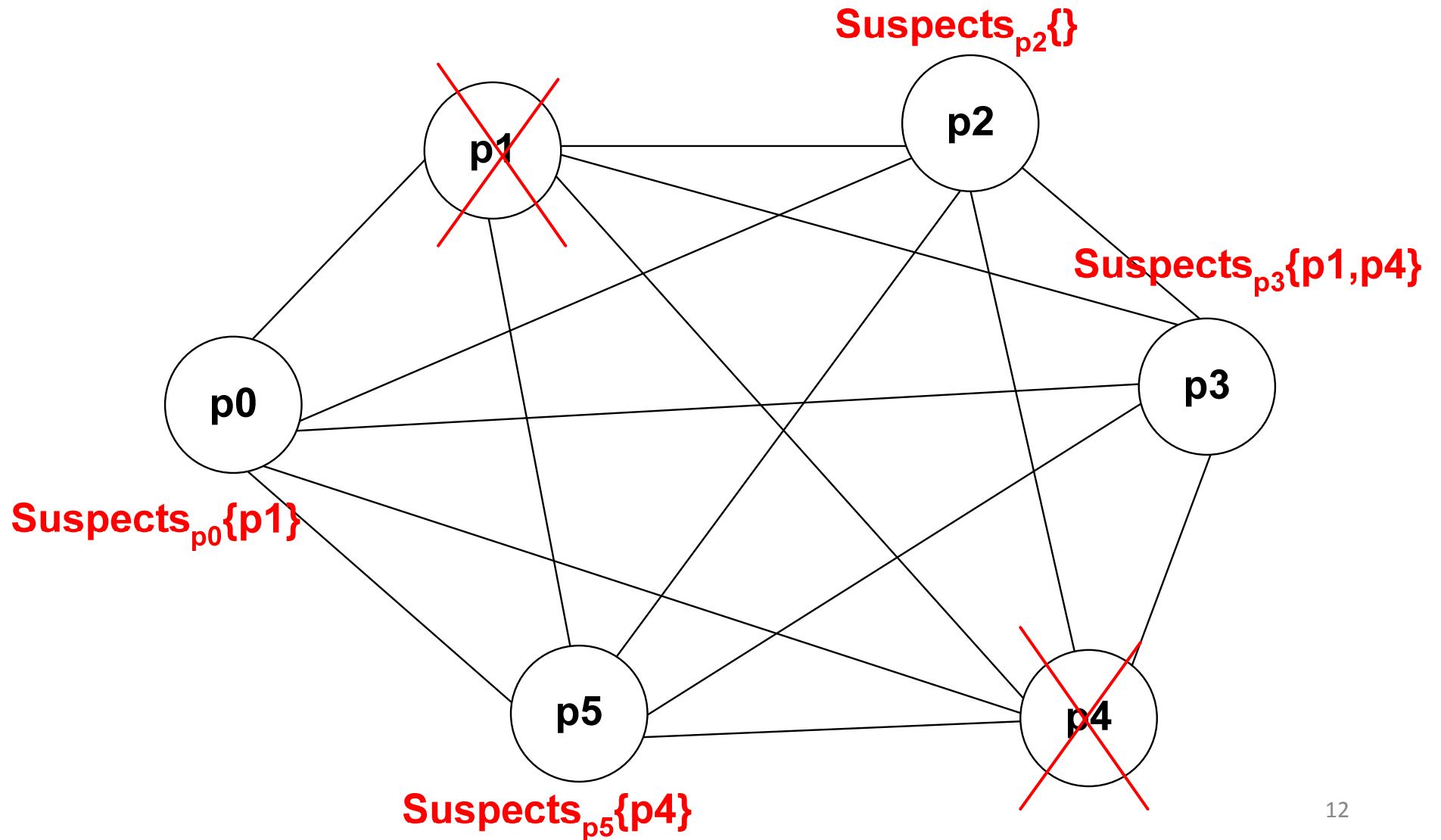
# Completeness

- Strong Completeness
  - Eventually, every crashed process is permanently suspected by *every* correct process
- Weak Completeness
  - Eventually, every crashed process is permanently suspected by *some* correct process

# Strong Completeness



# Weak Completeness



# Accuracy

- Strong Accuracy
  - A process is *never* suspected before it crashes by any correct process
- Weak Accuracy
  - Some correct process *never* suspected by any correct process

## Perpetual Accuracy!

As these properties hold all the times

# Eventual Accuracy

- Eventual Strong Accuracy
  - After a time, correct processes do not suspect correct processes
- Eventual Weak Accuracy
  - After a time, *some* correct process is not suspected by any correct process

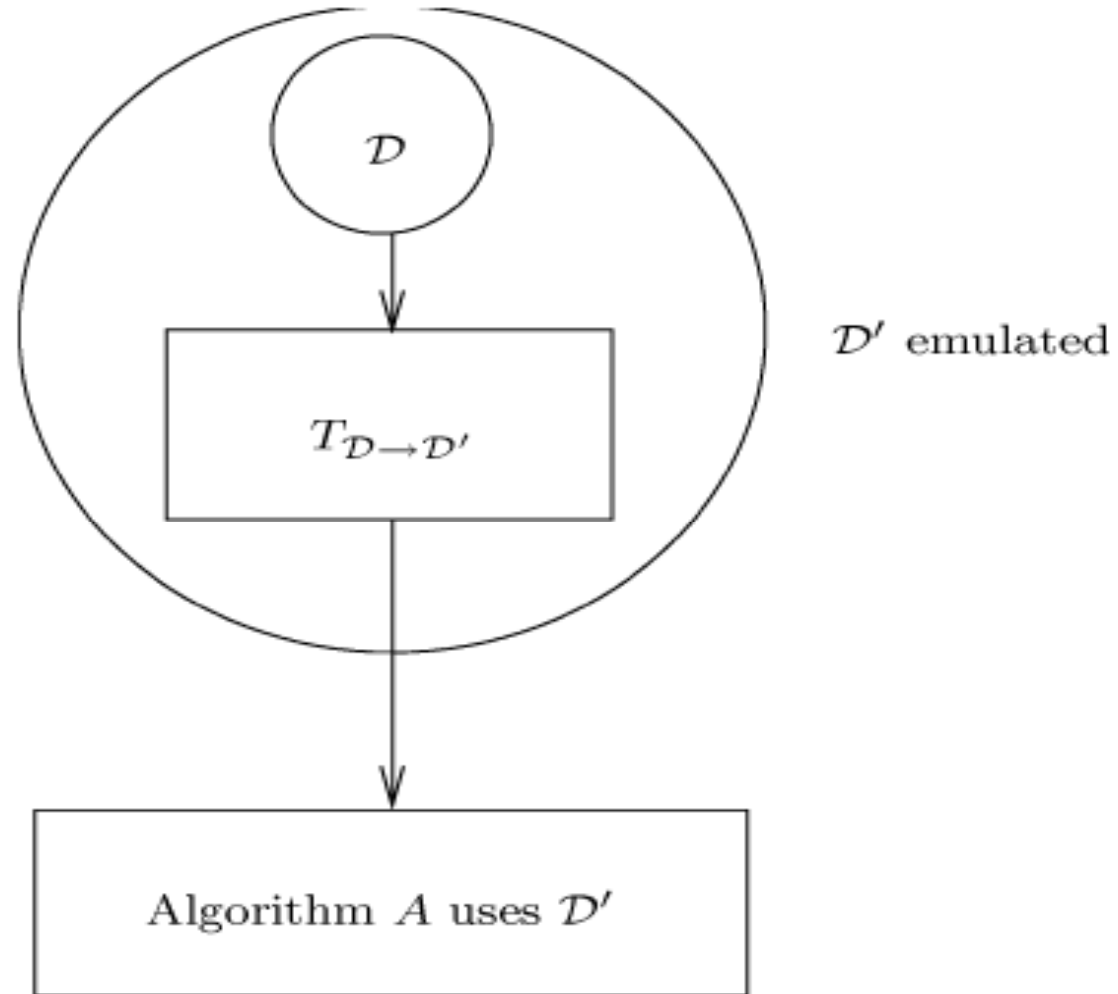
# Failure Detector Classes

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	Perfect $P$	Strong $S$	Eventually Perfect $\diamond P$	Eventually Strong $\diamond S$
Weak	$V$	Weak $W$	$\diamond V$	Eventually Weak $\diamond W$

# Reducibility

- A Failure detector  $D$  is reducible to another failure detector  $D'$  if there exist a reduction algorithm  $T_{D \rightarrow D'}$  that transforms  $D$  to  $D'$ .
- Then
  - $D'$  is Weaker than  $D$  (i.e)  $D \sqsubseteq D'$
- If  $D \sqsubseteq D'$  and  $D' \sqsubseteq D$  then  $D$  and  $D'$  are ***equivalent*** (i.e)  $D \equiv D'$
- Suppose a given algorithm 'A' requires failure detector  $D'$ , but only  $D$  is available.

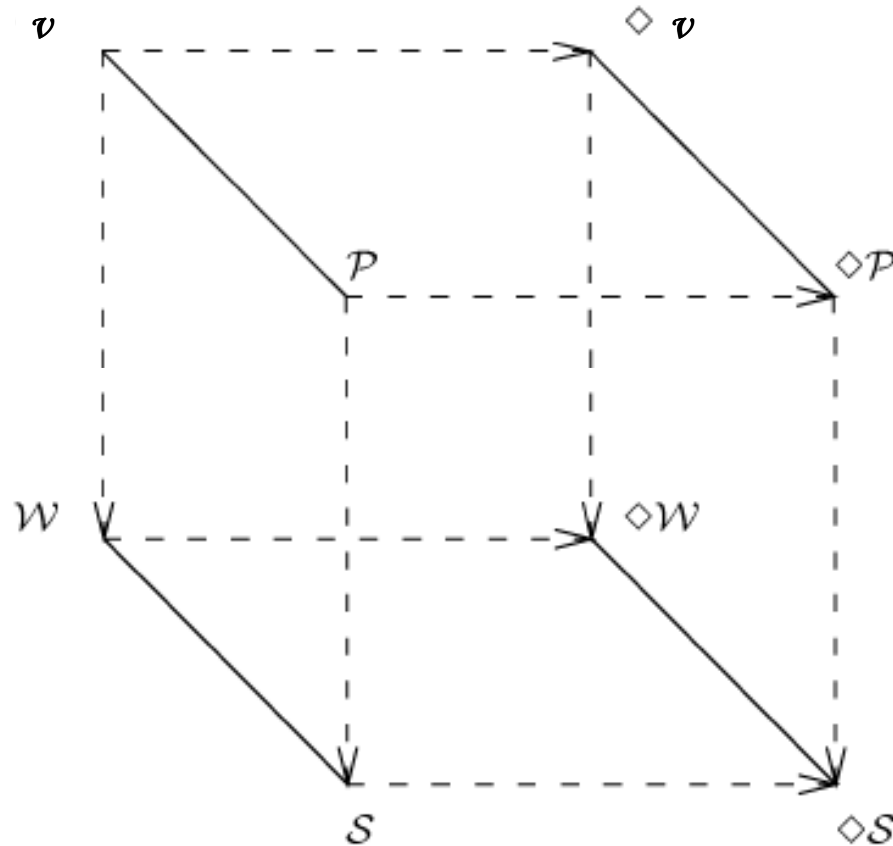
# Example



## Reducibility of FD

- $\mathcal{P} \sqsubseteq \mathcal{V} ; S \sqsubseteq \mathcal{W} ; \diamond \mathcal{P} \sqsubseteq \diamond \mathcal{V} ; \diamond S \sqsubseteq \diamond \mathcal{W}$
- $\mathcal{V} \sqsubseteq \mathcal{P} ; \mathcal{W} \sqsubseteq S ; \diamond \mathcal{V} \sqsubseteq \diamond \mathcal{P} ; \diamond \mathcal{W} \sqsubseteq \diamond S$
- $\mathcal{P} \equiv \mathcal{V} ; S \equiv \mathcal{W} ; \diamond \mathcal{P} \equiv \diamond \mathcal{V} ; \diamond S \equiv \diamond \mathcal{W}$
- Hence if we solve a problem for four failure detectors with strong completeness, the problem is automatically solved for the remaining four failure detectors.

# Comparing Failure detectors by Reducibility



$C \text{ ---} \rightarrow C'$ :  $C'$  is strictly weaker than  $C$

$C \text{ —} \rightarrow C'$ :  $C$  is equivalent to  $C'$

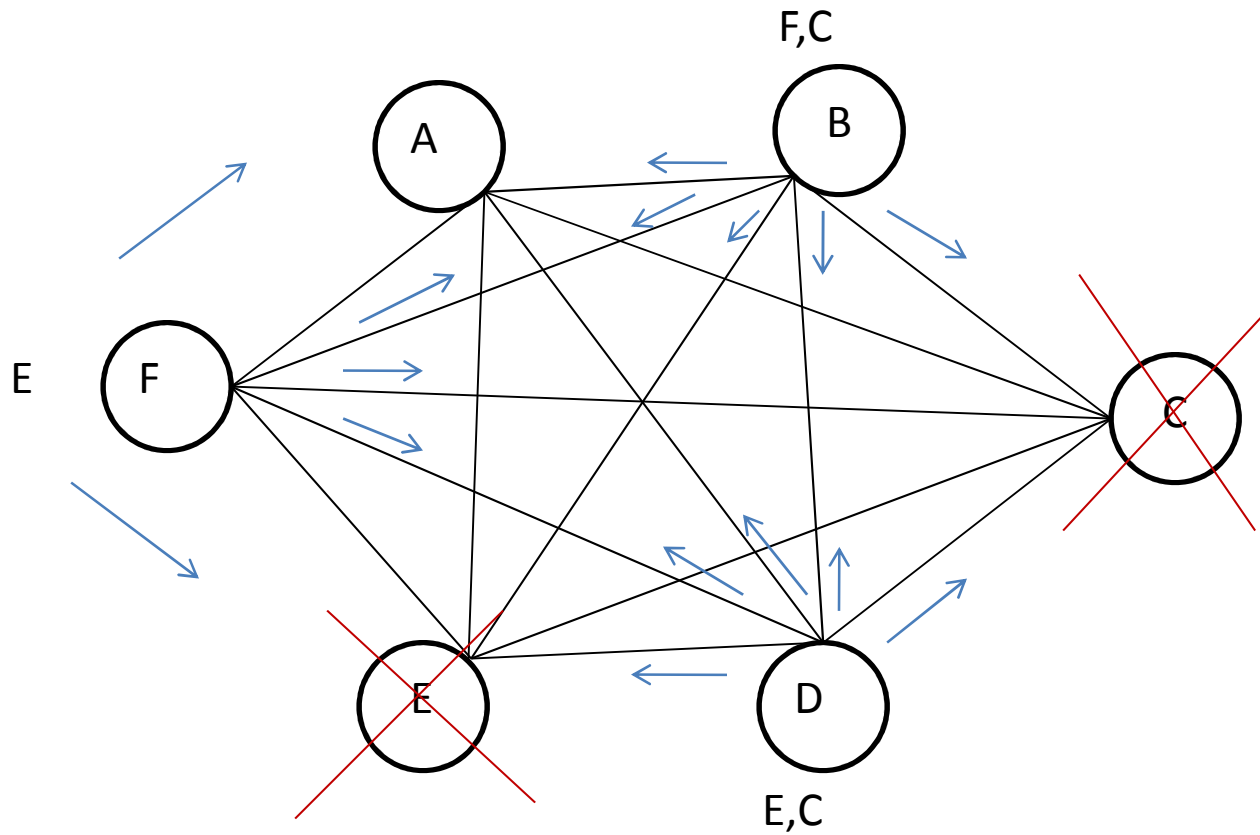
# Failure Detectors : Reducibility

- Two failure detectors are equivalent if they are reducible to each other.
- Failure detector with weak completeness is equivalent to corresponding failure detector with strong completeness.
- $\mathcal{P} \equiv \mathcal{V}; \diamond \mathcal{P} \equiv \diamond \mathcal{V}; \mathcal{S} \equiv \mathcal{W}; \diamond \mathcal{S} \equiv \diamond \mathcal{W}$
- Solving a problem for the four failure detectors with strong completeness, automatically solves for the remaining four failure detectors.

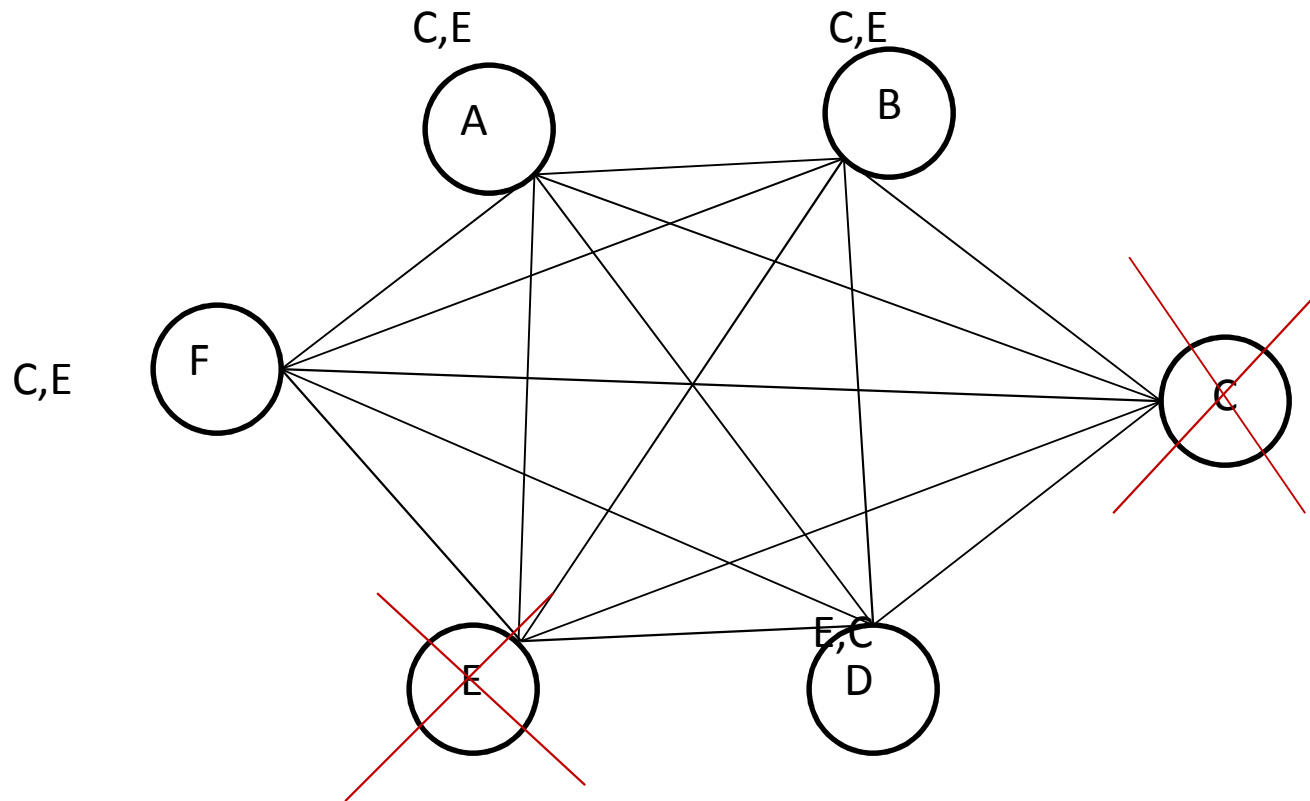
# Weak to Strong Completeness

- Every process  $p$  executes the following:
- $\text{Output}_p \leftarrow \text{Null}$
- `cobegin`
  - //Task 1: repeat forever
    - $\text{suspects}_p \leftarrow D_p$  { $p$  queries its local failure detector module  $D_p$ }
    - *send( $p, \text{suspects}_p$ ) to all other processes.*
  - //Task 2: when receive ( $q, \text{suspects}_q$ ) for a process  $q$ 
    - $\text{output}_p \leftarrow \text{output}_p \cup \text{suspects}_q - \{q\}$  {*output<sub>p</sub> emulates  $E_p$* }
- `coend`

# Weak to Strong Completeness



# Weak to Strong Completeness



# The consensus problem

- **Termination** : Every correct process eventually decides some value.
- **Uniform integrity** : Every process decides at most once.
- **Agreement** : No two correct processes decide differently.
- **Uniform validity** : If a process decides a value  $v$ , then some process proposed  $v$ .
- It is widely known that the consensus cannot be solved in *asynchronous systems in the presence of even a single crash failure*

# Solutions to the consensus problem

- $\mathcal{P} \equiv \mathcal{V}; \diamond\mathcal{P} \equiv \diamond\mathcal{V}; \mathcal{S} \equiv \mathcal{W}; \diamond\mathcal{S} \equiv \diamond\mathcal{W}$
- Solving a problem for the four failure detectors with strong completeness, automatically solves for the remaining four failure detectors
- Since  $\mathcal{P}$  is reducible to  $\mathcal{S}$  and  $\diamond\mathcal{P}$  is reducible to  $\diamond\mathcal{S}$ .
- The algorithm for solving consensus using  $\mathcal{S}$  also solve consensus using  $\mathcal{P}$ .
- The algorithm for solving consensus using  $\diamond\mathcal{S}$  also solve consensus using  $\diamond\mathcal{P}$ .

# Consensus using $\mathcal{S}$

Every process  $p$  executes the following:

**procedure** *propose*( $v_p$ )

$V_p \leftarrow \langle \perp, \perp, \dots, \perp \rangle$

{ $p$ 's estimate of the proposed values}

$V_p[p] \leftarrow v_p$

$\Delta_p \leftarrow V_p$

**Phase 1:** {asynchronous rounds  $r_p$ ,  $1 \leq r_p \leq n - 1$ }

**for**  $r_p \leftarrow 1$  to  $n - 1$

send  $(r_p, \Delta_p, p)$  to all

**wait until**  $[\forall q : \text{received } (r_p, \Delta_q, q) \text{ or } q \in \mathcal{D}_p]$  {query the failure detector}

$msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid \text{received } (r_p, \Delta_q, q)\}$

$\Delta_p \leftarrow \langle \perp, \perp, \dots, \perp \rangle$

**for**  $k \leftarrow 1$  **to**  $n$   
   **if**  $V_p[k] = \perp$  **and**  $\exists (r_p, \Delta_q, q) \in \text{msgs}_p[r_p]$  **with**  $\Delta_q[k] \neq \perp$  **then**  
      $V_p[k] \leftarrow \Delta_q[k]$   
      $\Delta_p[k] \leftarrow \Delta_q[k]$

**Phase 2:** send  $V_p$  to all

**wait until**  $[\forall q : \text{received } V_q \text{ or } q \in \mathcal{D}_p]$  *{ query the failure detector }*

$\text{lastmsgs}_p \leftarrow \{V_q \mid \text{received } V_q\}$

**for**  $k \leftarrow 1$  **to**  $n$

**if**  $\exists V_q \in \text{lastmsgs}_p$  **with**  $V_q[k] = \perp$  **then**  $V_p[k] \leftarrow \perp$

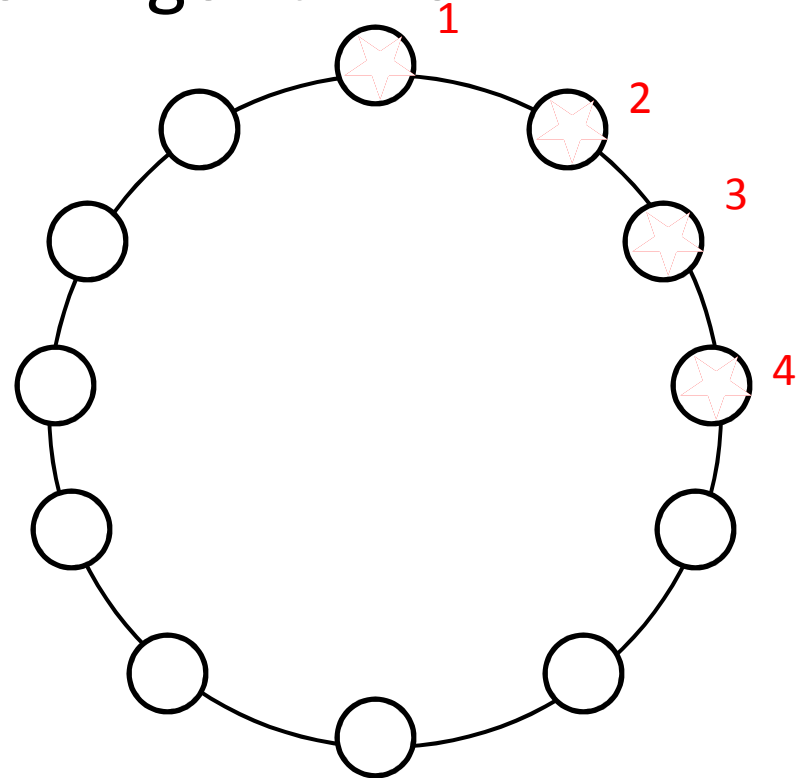
**Phase 3:** *decide*( first non- $\perp$  component of  $V_p$  )

# Solving Consensus using $\diamond S$ :

## Rotating Coordinator Algorithms

Work for up to  $f < n/2$  crashes

- Processes are numbered  $1, 2, \dots, n$
- They execute asynchronous *rounds*
- In round  $r$ , the *coordinator* is process  $(r \bmod n) + 1$
- In round  $r$ , the coordinator:
  - tries to impose its estimate as the consensus value
  - succeeds if  $\diamond S$  does not crash and it is not suspected by  $\diamond S$



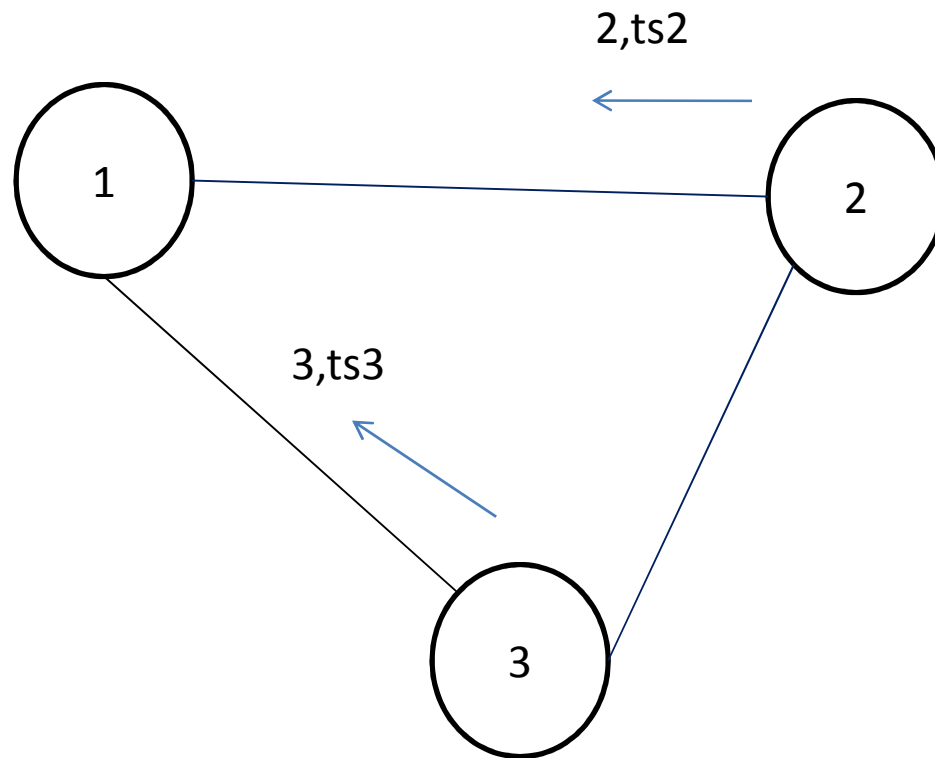
# Consensus using $\diamond S$

- The algorithm goes through
  - three Asynchronous stages
    - Each stage has several asynchronous rounds
      - Each round has 2 tasks
        - » Task 1
          - Four asynchronous phases
        - » Task 2
- In the first stage, several decision values are proposed
- In second stage, a value gets locked: no other decision value is possible
- In the third and final stage, the processes decide on the locked value and consensus is reached.

# Consensus using $\diamond S$

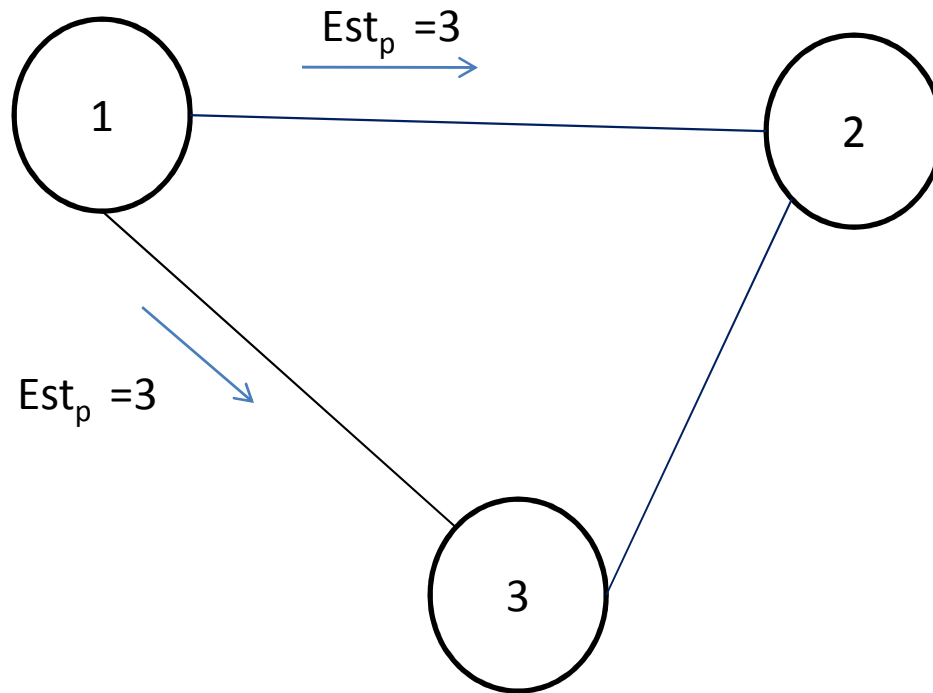
- Task 1
  - Phase 1
    - Every process 'p' sends
      - Current estimate to coordinator  $C_p$
      - Round number  $ts_p$
  - Phase 2
    - $C_p$  gathers  $\lceil (n+1)/2 \rceil$  estimates
    - Selects one with largest time stamp  $estimate_p$
    - Send the new estimate to all processes
  - Phase 3
    - Each process 'p'
      - May receive  $estimate_p$ 
        - » Send an ack to  $C_p$
      - May not receive  $estimate_p$ 
        - » Send a nack to  $C_p$  (suspecting  $C_p$  has crashed)
  - Phase 4
    - Waits for  $\lceil (n+1)/2 \rceil$  (acks or nacks)
      - If all are acks then  $estimate_p$  is locked
      - $C_p$  broadcasts the decided value  $estimate_p$
- Task 2
  - If a process 'p' receives a broadcast on decided value and has not already decided
    - Accepts the value

# Consensus using $\diamond S$

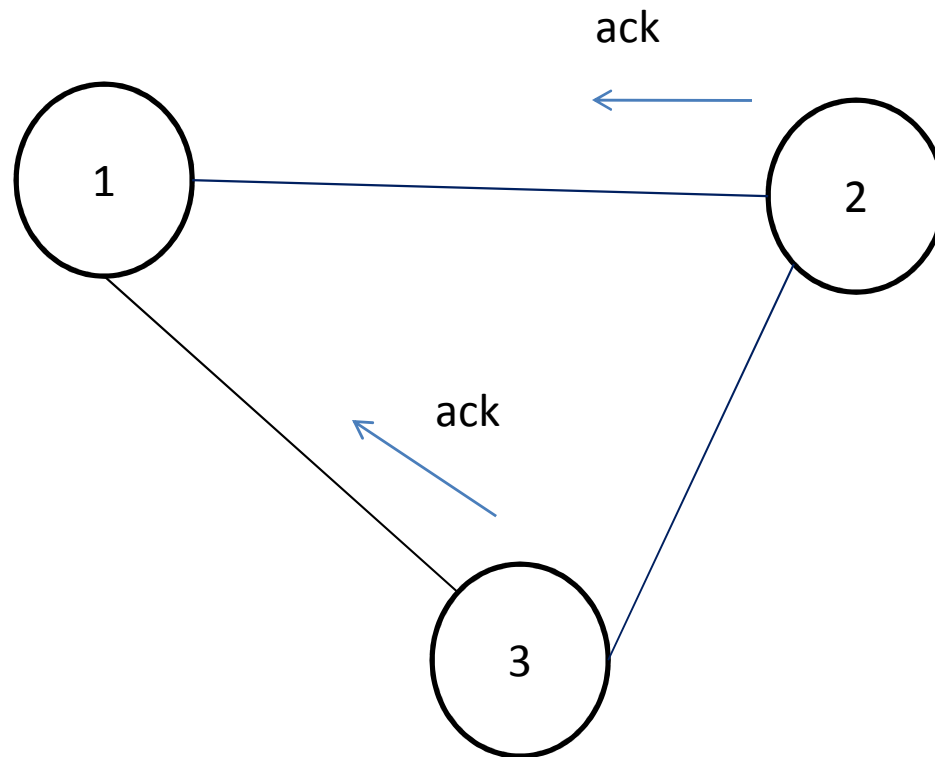


Let  $ts2 < ts1 < ts3$

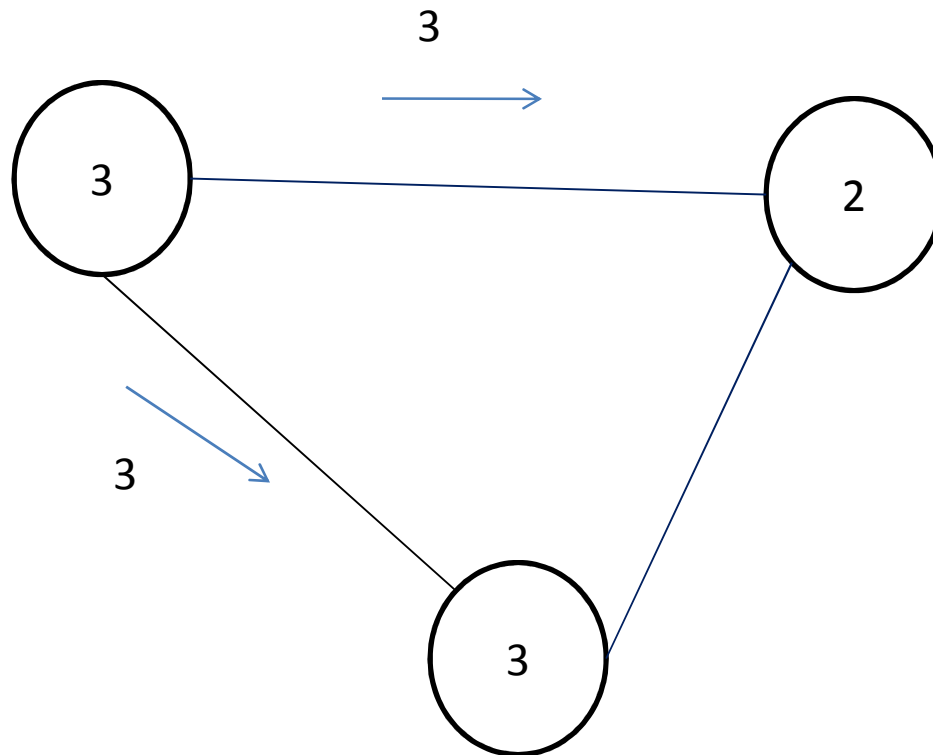
# Consensus using $\diamond S$



# Consensus using $\diamond S$

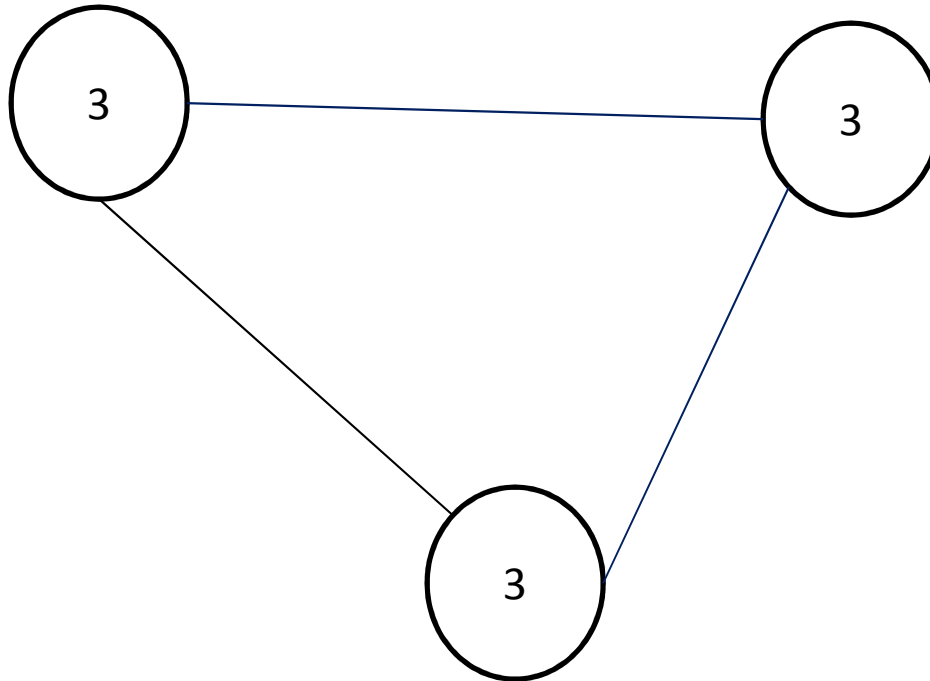


# Consensus using $\diamond S$



Locks 3 and broad casts

# Consensus using $\diamond S$



Locks 3 and broad casts

# Consensus using $\diamond S$

Every process  $p$  executes the following:

**procedure** *propose*( $v_p$ )

$estimate_p \leftarrow v_p$  { $estimate_p$  is  $p$ 's estimate of the decision value}

$state_p \leftarrow undecided$

$r_p \leftarrow 0$  { $r_p$  is  $p$ 's current round number}

$ts_p \leftarrow 0$  { $ts_p$  is the last round in which  $p$  updated  $estimate_p$ , initially 0}

{Rotate through coordinators until decision is reached}

**while**  $state_p = undecided$

$r_p \leftarrow r_p + 1$

$c_p \leftarrow (r_p \bmod n) + 1$  { $c_p$  is the current coordinator}

**Phase 1:** {All processes  $p$  send  $estimate_p$  to the current coordinator}

send  $(p, r_p, estimate_p, ts_p)$  to  $c_p$

# Consensus using $\diamond S$ cont...

**Phase 2:**  $\{$  The current coordinator gathers  $\lceil \frac{(n+1)}{2} \rceil$  estimates and proposes a new estimate  $\}$

if  $p = c_p$  then

wait until [for  $\lceil \frac{(n+1)}{2} \rceil$  processes  $q$  : received  $(q, r_p, estimate_q, ts_q)$  from  $q$ ]  
 $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p \text{ received } (q, r_p, estimate_q, ts_q) \text{ from } q\}$   
 $t \leftarrow$  largest  $ts_q$  such that  $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$   
 $estimate_p \leftarrow$  select one  $estimate_q$  such that  $(q, r_p, estimate_q, t) \in msgs_p[r_p]$   
send  $(p, r_p, estimate_p)$  to all

**Phase 3:**  $\{$  All processes wait for the new estimate proposed by the current coordinator  $\}$

wait until [received  $(c_p, r_p, estimate_{c_p})$  from  $c_p$  or  $c_p \in \mathcal{D}_p$ ]{Query the failure detector}

if [received  $(c_p, r_p, estimate_{c_p})$  from  $c_p$ ] then  $\{p \text{ received } estimate_{c_p} \text{ from } c_p\}$

$estimate_p \leftarrow estimate_{c_p}$

$ts_p \leftarrow r_p$

send  $(p, r_p, ack)$  to  $c_p$

else send  $(p, r_p, nack)$  to  $c_p$   $\{p \text{ suspects that } c_p \text{ crashed}\}$

# Consensus using $\diamond S$ cont...

Phase 4:  $\left\{ \begin{array}{l} \text{The current coordinator waits for } \lceil \frac{(n+1)}{2} \rceil \text{ replies. If they indicate that } \lceil \frac{(n+1)}{2} \rceil \\ \text{processes adopted its estimate, the coordinator R-broadcasts a decide message} \end{array} \right\}$

if  $p = c_p$  then

wait until [for  $\lceil \frac{(n+1)}{2} \rceil$  processes  $q$  : received  $(q, r_p, ack)$  or  $(q, r_p, nack)$ ]

if [for  $\lceil \frac{(n+1)}{2} \rceil$  processes  $q$  : received  $(q, r_p, ack)$ ] then

$R\text{-broadcast}(p, r_p, estimate_p, decide)$

{If  $p$  R-delivers a decide message,  $p$  decides accordingly}

when  $R\text{-deliver}(q, r_q, estimate_q, decide)$

if  $state_p = undecided$  then

$decide(estimate_q)$

$state_p \leftarrow decided$

# Atomic Broadcast

- Informally, atomic broadcast requires that all correct processes deliver the same set of messages in the same order (i.e., deliver the same sequence of messages).
- Formally atomic broadcast can be defined as a reliable broadcast with the total order property
- Chandra and Toueg showed that the result of consensus can be used to solve the problem of atomic broadcast.

- **Reliable Broadcast**

- **Validity** : If the sender of a broadcast message  $m$  is non-faulty, then all correct processes eventually deliver  $m$ .
- **Agreement** : If a correct process delivers a message  $m$ , then all correct processes deliver  $m$ .
- **Integrity** : Each correct process delivers a message at most once.

- **Total Order**

- If two correct processes  $p$  and  $q$  deliver two messages  $m$  and  $m'$ , then  $p$  delivers  $m$  before  $m'$  if and only if  $q$  delivers  $m$  before  $m'$ .

# Reliable Broadcast

*Every process  $p$  executes the following:*

*To execute R-broadcast( $m$ ):*

*send  $m$  to all (including  $p$ )*

*R-deliver( $m$ ) occurs as follows:*

**when** receive  $m$  for the first time

**if**  $sender(m) \neq p$  **then** send  $m$  to all

*R-deliver( $m$ )*

# Atomic Broadcast

- The algorithm consists of three tasks :
- **Task 1 :**
  - when a process  $p$  wants to A-broadcast a message  $m$ , it *R\_broadcasts*  $m$ .
- **Task 2 :**
  - a message  $m$  is added to set  $R\_delivered$  when process  $p$  *R\_delivers* it.
- **Task 3 :**
  - when a process  $p$  *A\_delivers* a message  $m$ , it adds  $m$  to set  $A\_delivered$ .
  - Process  $p$  periodically checks whether  $A\_undelivered$  contains messages. If it contains messages,  $p$  enters its next execution of consensus, say the  $k$ th one, and proposes  $A\_undelivered$  as the next batch of messages to be *A\_delivered*.

# Atomic Broadcast

Every process  $p$  executes the following:

Initialisation:

$R\_delivered \leftarrow \emptyset$   
 $A\_delivered \leftarrow \emptyset$   
 $k \leftarrow 0$

To execute  $A$ -broadcast( $m$ ):

{ Task 1 }

$R$ -broadcast( $m$ )

$A$ -deliver( $-$ ) occurs as follows:

**when**  $R$ -deliver( $m$ )

{ Task 2 }

$R\_delivered \leftarrow R\_delivered \cup \{m\}$

**when**  $R\_delivered - A\_delivered \neq \emptyset$

{ Task 3 }

$k \leftarrow k + 1$

$A\_undelivered \leftarrow R\_delivered - A\_delivered$

propose( $k, A\_undelivered$ )

**wait until**  $decide(k, msgSet^k)$

$A\_deliver^k \leftarrow msgSet^k - A\_delivered$

atomically deliver all messages in  $A\_deliver^k$  in some deterministic order

$A\_delivered \leftarrow A\_delivered \cup A\_deliver^k$

# Implementation of failure detector

- **Task 1** : Each process  $p$  periodically sends a “ $p$ -is-alive” message to all other processes. This is like a heart-beat message that informs other processes that process  $p$  is alive.
- **Task 2** : If a process  $p$  does not receive a “ $q$ -is-alive” message from a process  $q$  within  $p(q)$  time units on its clock, then  $p$  adds  $q$  to its set of suspects if  $q$  is not already in the suspect list of  $p$ .
- **Task 3** : When a process delivers a message from a suspected process, it corrects its error about the suspected process and increases its timeout for that process.
  - If process  $p$  receives “ $q$ -is-alive” message from a process  $q$  that it currently suspects,  $p$  knows that its previous timeout on  $q$  was premature –  $p$  removes  $q$  from its set of suspects and increases its timeout period for process  $q$ ,  $p(q)$ .

# Implementation of failure detector

Every process  $p$  executes the following:

```
 $output_p \leftarrow \emptyset$   
for all  $q \in \Pi$   $\{\Delta_p(q)$  denotes the duration of  $p$ 's time-out interval for  $q\}$   
   $\Delta_p(q) \leftarrow$  default time-out interval  
  
cobegin  
  || Task 1: repeat periodically  
    send “ $p$ -is-alive” to all  
  
  || Task 2: repeat periodically  
    for all  $q \in \Pi$   
      if  $q \notin output_p$  and  
         $p$  did not receive “ $q$ -is-alive” during the last  $\Delta_p(q)$  ticks of  $p$ 's clock  
           $output_p \leftarrow output_p \cup \{q\}$   $\{p$  times-out on  $q$ : it now suspects  $q$  has crashed $\}$   
  
  || Task 3: when receive “ $q$ -is-alive” for some  $q$   
    if  $q \in output_p$   $\{p$  knows that it prematurely timed-out on  $q\}$   
       $output_p \leftarrow output_p - \{q\}$   $\{1. p$  repents on  $q$ , and $\}$   
       $\Delta_p(q) \leftarrow \Delta_p(q) + 1$   $\{2. p$  increases its time-out period for  $q\}$   
coend
```

Fig. 10. A time-out based implementation of  $\mathcal{D} \in \diamond\mathcal{P}$  in models of partial synchrony.

# Lazy failure detection protocol

- A relatively simple protocol that allows a process to “monitor” another process, and consequently to detect its crash.
- This protocol enjoys the nice property to rely as much as possible on application messages to do this monitoring.
- The cost associated with the implementation of a failure detector incurs only when the failure detector is used (hence, it is called a lazy failure detector).
- Each process  $p_i$  has a local hardware clock  $hc_i$  that strictly monotonically increases.
- The local clocks are not required to be synchronized
- Every pair of processes is connected by a channel and they communicate by sending and receiving messages through channels.
- Channels are not required to be FIFO

# Lazy failure detection protocol

- (1) **when** SEND  $M$  to  $p_j$  **is invoked**:
- (2)  $m.content \leftarrow M; m.st \leftarrow hc_i;$
- (3)  $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] \cup \{m.st\}$
- (4) **send**  $appl(m)$  to  $p_i$
  
- (5) **when**  $type(m)$  **is received** from  $p_j$ :
- (6) **case**  $type=appl$  **then** **transmit**  $M = m.content$  to the upper layer; % RECEIVE  $M$  %
- (7) **send**  $ack(m)$  to  $p_j$  %  $m.st$  keeps its value %
- (8) **type=ack** **then**  $rt \leftarrow hc_i;$
- (9)  $max\_rtd_i[j] \leftarrow \max(max\_rtd_i[j], rt - m.st);$
- (10)  $pending\_msg\_st_i[j] \leftarrow pending\_msg\_st_i[j] - \{m.st\}$
- (11) **type=ping** **then** **send**  $ack(m)$  to  $p_j$  %  $m.st$  keeps its value %
- (12) **endcase**
  
- (13) **when** QUERY( $j$ ) **is invoked**:
- (14) **if**  $pending\_msg\_st_i[j] = \emptyset$  **then** **create** a control message  $m$ ;
- (15)  $m.content \leftarrow null; m.st \leftarrow hc_i;$
- (16) **send**  $ping(m)$  to  $p_j$ ;
- (17)  $pending\_msg\_st_i[j] \leftarrow \{m.st\};$
- (18) **return** ( $no\_suspect$ )
- (19) **else**  $rt \leftarrow hc_i;$
- (20) **if**  $rt - \min(pending\_msg\_st_i[j]) > max\_rtd_i[j]$
- (21) **then** **return** ( $suspect$ )
- (22) **else** **return** ( $no\_suspect$ )
- (23) **endif**
- (24) **endif**

# **A short introduction to failure detectors for asynchronous Distributed Systems**

# Failure Detectors-Definition

## Why use FD?

- Based on well defined set of Abstract concepts
- Not dependant on any particular implementation
- Layered approach favors design, proof and portability of protocol
- Helps to solve impossible time-free asynchronous distributed system problems like the Consensus problem.
- Eventually accurate failure detectors helps in designing indulgent algorithms.

# Asynchronous System Models

## Process model

- A process can fail by premature halting(crashing).
- A process is correct if it does not crash else it is faulty

## Computation models

- **FLP** Crash-prone processes and reliable links
- **FLL** Crash-prone processes and fair lossy links

# Asynchronous System Models

## Communication model

Processes communicate and synchronize by exchanging messages through links.

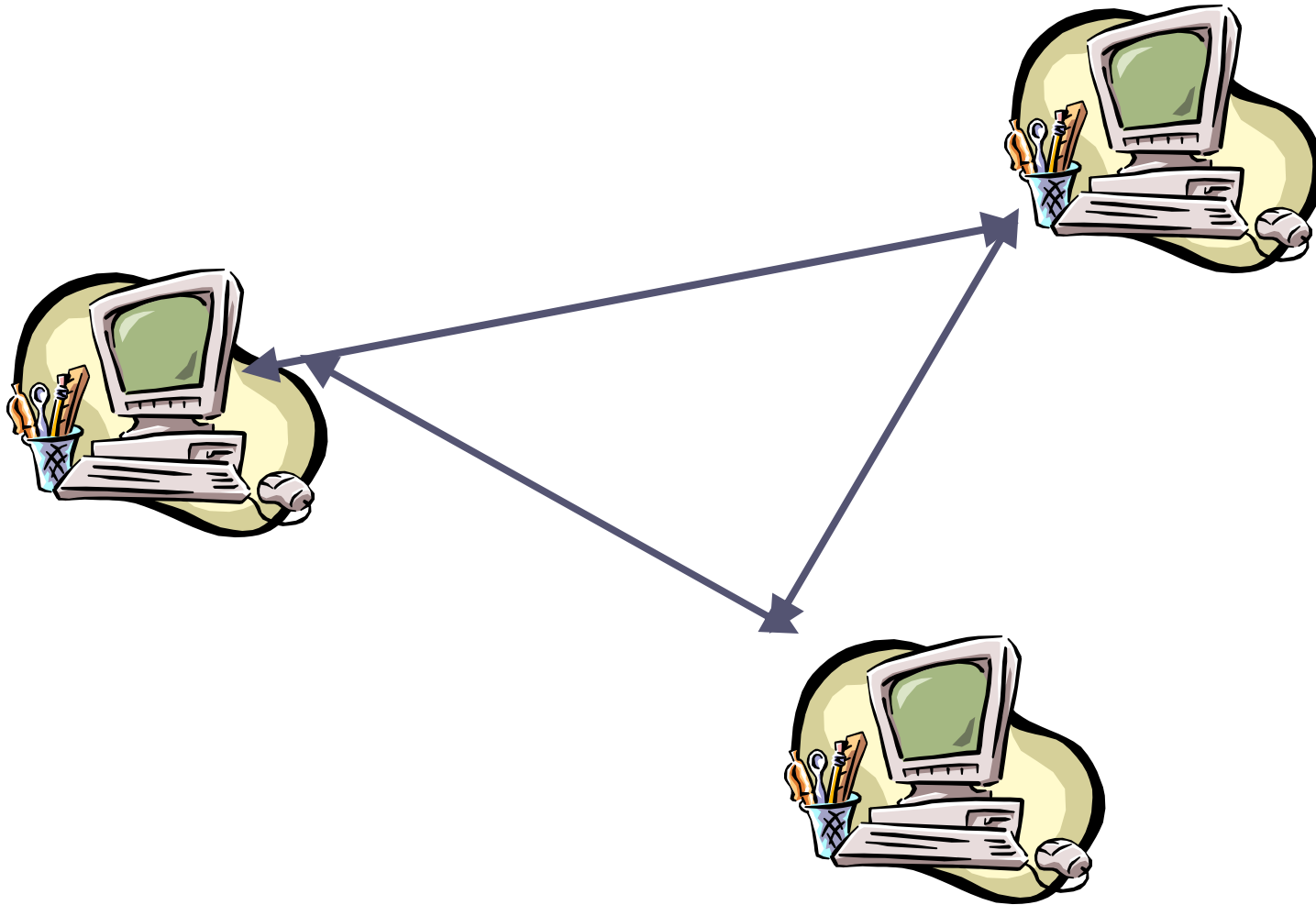
### Reliable

- Does not create or duplicate messages
- Every message sent by  $P_i$  to  $P_j$  is eventually received by  $P_j$

### Fair lossy

- Does not create or duplicate messages
- Can lose message
- Can send infinite number of messages from one process to another

# Consensus



# Consensus

- All the processes, propose a initial value and they all have to agree upon some common value proposed
- Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, terminating reliable broadcast)

# Consensus definition

***C-Validity:*** Any value decided is a value proposed

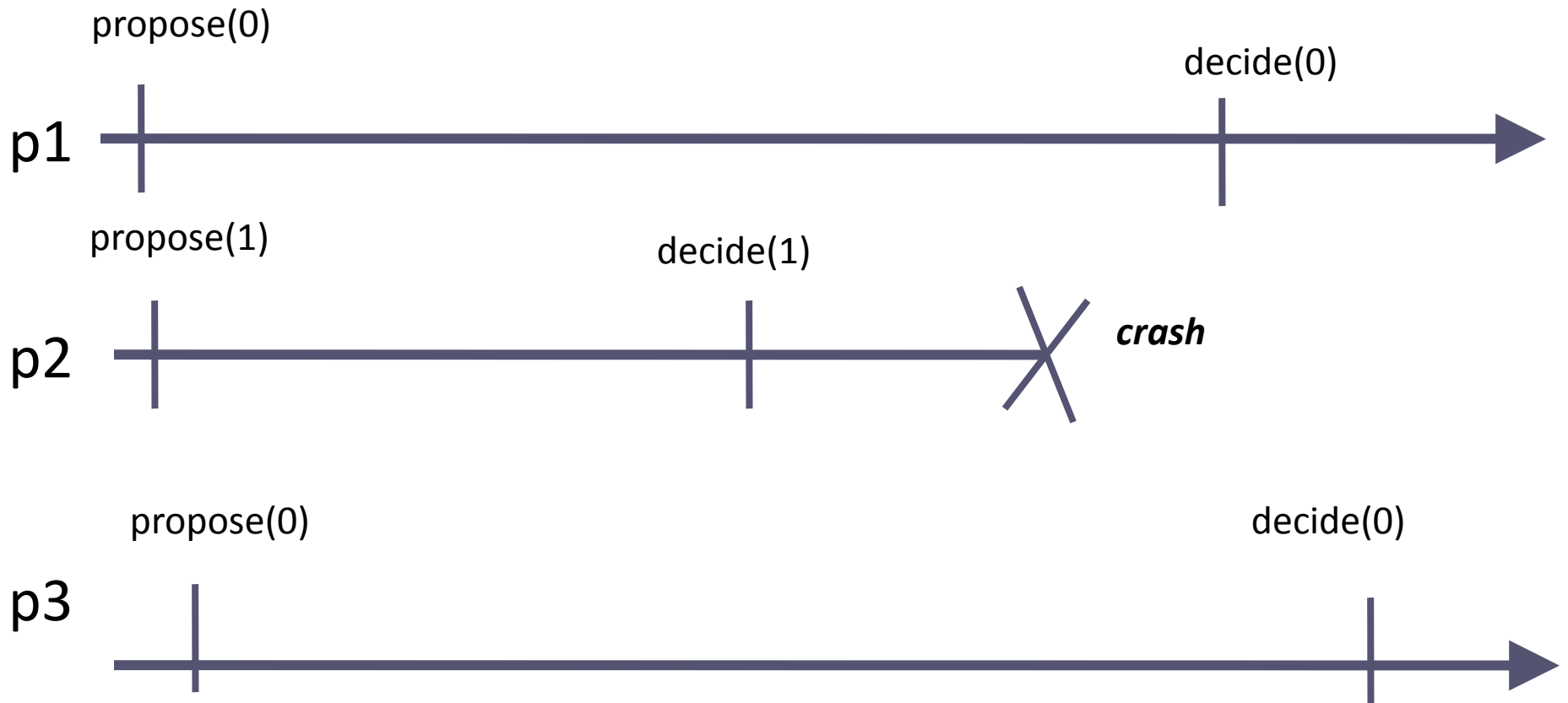
***C-Agreement:*** No two correct processes decide differently

***C-Termination:*** Every correct process eventually decides

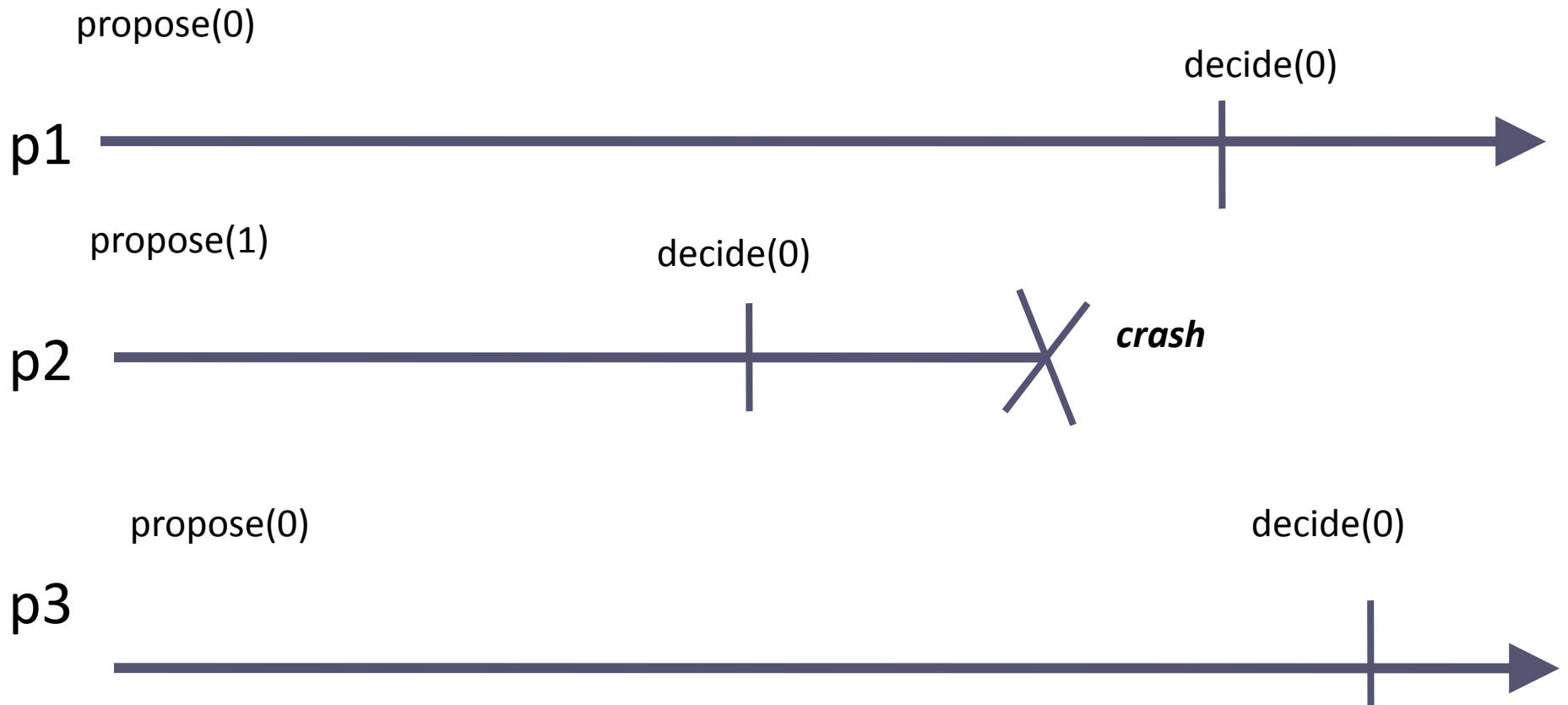
***C-Integrity:*** No process decides twice

***C-Uniform Agreement:*** No two (correct or not) processes decide differently

# Consensus



# Uniform Consensus



# Eventually accurate failure detectors

- **Strong Completeness**

Eventually, all processes that crash are suspected by every correct process

- **Eventually Weak Accuracy**

There is a time after which some correct process is never suspected by the correct processes

## ◇ *S-based Consensus Protocol*

- FLP model
- Indulgent
  - Never violates consensus safety
  - Terminates when the sets contain correct values during a long enough period
- Requires majority of correct processes ( $t < n/2$ )
- Proceeds in asynchronous consecutive rounds
- Each round  $r$  is coordinated by process  $p_c$  such that,  $c = (r \bmod n) + 1$

# Initialization

- $v_i = \text{value}$  initially *proposed* by  $p_i$ .
- $est_i = p_i$ 's *estimate* of the decision value.
- In round  $r$ , its coordinator  $p_c$  tries to impose its current estimate as the decision value.
- Algorithm runs in two phases.

# Phase 1

- $p_c$  sends  $est_c$  to all the processes
- process  $p_i$  waits until it receives  $p_c$ 's estimate or suspects it.
- Based on result of waiting, either
$$aux_i = v (= est_c)$$
or
$$aux_i = \perp$$
- Due to the completeness property of the underlying failure detector no process can block forever

## Phase 2

- All process exchange the values of their  $aux_i$  variables
- Due to the “*majority of correct processes*” assumption, no process can block forever
- Only two values can be exchanged:  $v = est_c$  or  $\perp$ .
- Therefore,

$$rec_i = \{\{v\}, \{v, \perp\}, \text{ or } \{\perp\}\}$$

- Impossible for two sets  $rec_i$  and  $rec_j$  to be such that

$$rec_i = \{v\}$$

$$rec_j = \{\perp\}$$

# Phase 2

$$\text{rec}_i = \{v\} \Rightarrow (\forall p_j : (\text{rec}_j = \{v\}) \vee (\text{rec}_j = \{v, \perp\}))$$

$$\text{rec}_i = \{\perp\} \Rightarrow (\forall p_j : (\text{rec}_j = \{\perp\}) \vee (\text{rec}_j = \{v, \perp\})).$$

$$\text{rec}_i = \{v\}$$

$$\text{est}_i = v.$$

To prevent possible deadlock situations,  $p_i$  broadcasts its decision value.

$$\text{rec}_i = \{v, \perp\}$$

$$\text{est}_i = v.$$

proceeds to the next round.

$$\text{rec}_i = \{\perp\}$$

$p_i$  proceeds to the next round without modifying  $\text{est}_i$ .

# A Simple S-Based Consensus Protocol ( $t < n/2$ )

Function Consensus( $v_i$ )

Task T1:

(1)  $r_i \leftarrow 0$ ;  $est_i \leftarrow v_i$ ;

(2) **while true do**

(3)  $c \leftarrow (r_i \bmod n) + 1$ ;  $r_i \leftarrow r_i + 1$ ; %  $1 \leq r_i < +\infty$  %

----- Phase 1 of round  $r$ : from  $p_c$  to all -----

(4) **if ( $i = c$ ) then broadcast phase1( $r_i, est_i$ ) endif;**

(5) **wait until (phase1( $r_i, v$ ) has been received from  $p_c \forall c \in suspected_i$ );**

(6) **if (phase1( $r_i, v$ ) received from  $p_c$ ) then  $aux_i \leftarrow v$  else  $aux_i \leftarrow \perp$  endif;**

----- Phase 2 of round  $r$ : from all to all -----

(7) **broadcast phase2( $r_i, aux_i$ );**

(8) **wait until (phase2 ( $r_i, aux$ ) msgs have been received from a majority of proc.);**

(9) **let  $rec_i$  be the set of values received by  $p_i$  at line 8;**

% We have  $rec_i = \{v\}$ , or  $rec_i = \{v, \perp\}$ , or  $rec_i = \{\perp\}$  where  $v = est_c$  %

(10) **case  $rec_i = \{v\}$  then  $est_i \leftarrow v$ ; broadcast decision( $est_i$ ); stop T1**

(11)  **$rec_i = \{v, \perp\}$  then  $est_i \leftarrow v$**

(12)  **$rec_i = \{\perp\}$  then skip**

(13) **endcase**

(14) **endwhile**

Task T2: **when decision( $est$ ) is received: broadcast decision( $est_i$ ); return( $est$ )**

# Findings

- The strong completeness property is used to show that the protocol never blocks.
- The eventual weak accuracy property is used to ensure termination.
- The majority of correct processes is used to prove consensus agreement.

# Interactive consistency

- Harder than consensus problem
- Process has to agree on a vector of values!

## Termination

Every correct process eventually decides on a vector

## Validity

Any decided vector  $D$  is such that  $D[i] \in \{v_i, \perp\}$ , and is  $v_i$  if  $p_i$  does not crash

## Agreement:

No two processes decide differently

# Perfect failure detectors

- Requires perfect failure detectors

## **Strong Completeness**

- Every process that crashes is eventually permanently suspected

## **Strong Accuracy**

- No process is suspected before it crashes

# Perfect failure detector

**init:**  $suspected_i \leftarrow \emptyset; seq_i \leftarrow 0$

**task T1:** *while true do*

$seq_i \leftarrow seq_i + 1; \% IC\ instance\ number\ \%$

$D_i \leftarrow IC\ Protocol(seq_i, v_i); \% v_i = \perp\ \%$

$suspected_i \leftarrow \{j \mid D_i[j] = \perp\}$

**enddo**

**task T2:** *when  $p_i$  issues QUERY:*

*return( $suspected_i$ )*

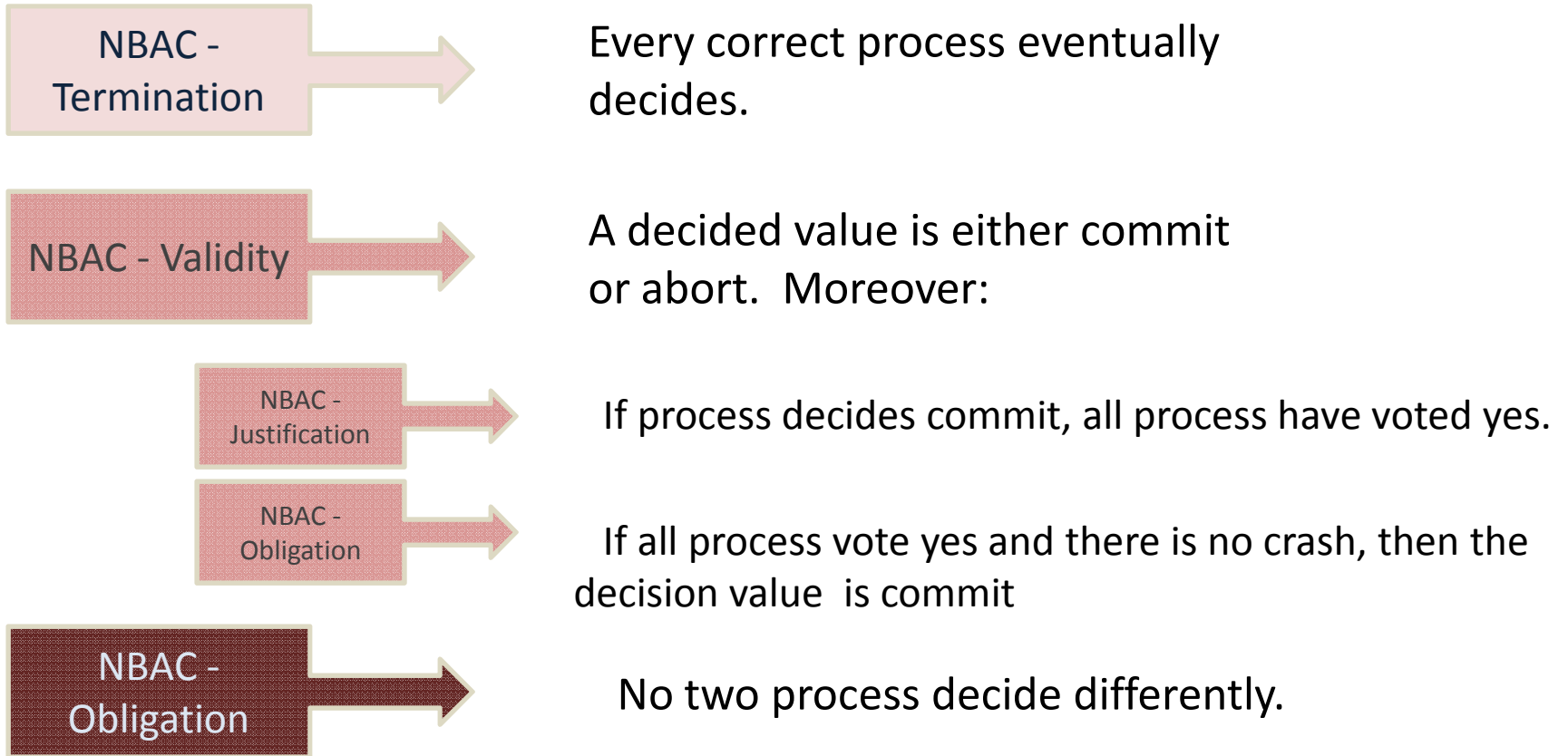
# Non-Blocking Atomic Commit Problem (NBAC)



- Yet another agreement problem in the world of distributed computing
- Each process cast their votes (yes or no).
- Non-crashed process decide on single value (*commit* or *abort*)

# Properties

The problem is defined by following properties



## Continued

- Justification property relates commit decision to yes.
- Obligation property eliminates trivial solution of all process opting abort.
  - “*good*” *run* – all process wants to commit and the environment is free of crashes.
- Process crashes are explicit in NBAC compared consensus.

# Appropriate Failure Detector

Why appropriate failure detector?

- To solve NBAC in the FLP model

*Timeless failure detectors* – No information ( sense of time ) when failure occurred.

## Anonymous Perfect Failure Detectors

$\mathcal{P}$  and  $\diamond S$  - timeless failure detectors.

To address this problem, class  $\mathcal{P}$  anonymous perfect failure detector introduced.

- **Anonymous completeness:** If a crash occurs, eventually every correct process is permanently informed that some crash occurred.
- **Anonymous accuracy:** No crash is detected unless some process crashed.

Class  $\mathcal{P} + \diamond S$  - weakest class to solve NBAC, assuming a majority of correct process.

The following protocol converts NBAC to consensus and subsequently uses subroutine consensus protocol.

## Simple $\diamond$ P + $\diamond$ S-Based NBAC protocol ( $t < n/2$ )

```
Function Nbac(  $vote_i$  )  
  broadcast MY_VOTE( $vote_i$ );  
  wait until ( MY_VOTE( $vote_i$ ) has been received from each  
process  $\forall ap\_flag_i$ );  
  if ( a vote yes has been received from each of the n  
processes)  
    then  $output_i \leftarrow$  Consensus(commit)  
    else  $output_i \leftarrow$  Consensus(abort)  
  endif;  
  return( $output_i$ )
```

# Quiescence Problem

- Consider processes  $p_i$  and  $p_j$  that do not crash connected by fair lossy link, a basic communication problem is to build a reliable link on top of fair lossy link.
- Protocol used ( including TCP ) are quiescent - no message transfer after some time. ( communication ceases)
- What if process  $p_j$  crashes?
- How to solve quiescent communication problem?
  - *Heartbeat failure detectors*

# Heartbeat Failure Detector

- Failure detector outputs an array  $HB_i [1 ..n]$  – non decreasing counter at each process which satisfies.....
  - **HB-completeness:** If  $p_j$  crashes, then  $HB_i[j]$  stops increasing.
  - **HB-accuracy:** If  $p_j$  is correct, then  $HB_i[j]$  never stops increasing.
- Easy implementation but it is not quiescent.
- Allows the non-quiescent part of communication protocol to be isolated.
- Favors design modularity and eases correctness proof.
- “service” can be extended to upper layer applications.

# Quiescent Implementation

**Sender  $p_i$ :**

when SEND( $m$ ) TO  $p_j$  is invoked:

$seq_i \leftarrow seq_i + 1;$

**fork task** *repeat\_send*( $m, seq_i$ )

**task** *repeat\_send*( $m, seq_i$ )

$prev\_hb \leftarrow 1;$

**repeat periodically**  $hb \leftarrow HB_i[j];$

**if** ( $prev\_hb < hb$ ) **then** send *msg*( $m, seq$ ) to  $p_j$ ;

$prev\_hb \leftarrow hb$

**endif**

**until** (*ack*( $m, seq$ ) is received)

**Receiver  $p_j$ :**

**when** *msg*( $m, seq$ ) is received from  $p_i$ :

**if** (first reception of *msg*( $m, seq$ )) **then**  $m$  is RECEIVED **endif**;

send *ack*( $m, seq$ ) to  $p_i$

# Failure Detectors in Synchronous Systems

## Synchronous System Model

- Synchronous systems – characterized by time bound to receive & send message.
- Local computations take no time & transfer delays bounded by D.
  - Message sent at time 't' is not received after t+D (D-timeliness)
  - Links are reliable ( no duplication, losses)
  - Process have access to common clock.

Consider  $p_i$  sends message to  $p_j$  &  $p_k$ , D-timeliness and no-loss properties gives rise to following scenarios...

- $P_i$  crashes at time t, no message sent
- $P_i$  crashes at time t,  $p_j$  receives while  $p_k$  doesn't by t + D, vice versa.
- $P_i$  doesn't crash,  $p_j$  &  $p_k$  receives message by t + D

# Fast Failure Detectors

- Fast failure detector provides processes with following properties ( $d < D$ )
  - **d – Timely completeness:** If a process  $p_j$  crashes at time  $t$ , then, by time  $t + d$ , every alive process suspects it permanently.
  - **Strong accuracy:** No process is suspected before it crashes.
- Implemented with specialized hardware, also attains time complexity lower bounds  $\ll$  pure synchronous system.
- Protocol described in the following slide illustrates early deciding property, reducing time complexity to  $D + fd$  ( $f$  – actual number of process crashes)
- Snapshot of the Synchronous Consensus with Fast Failure Detector implementation is illustrated as follows...

# Fast Failure Detector Implementation

**init**  $est_i \leftarrow v_i; max_i \leftarrow 0$

**when**  $(est, j)$  is received:

**if**  $(j > max_i)$  **then**  $est_i \leftarrow est; max_i \leftarrow j$  **endif**

**at time**  $(i-1)d$  **do**

**if**  $(\{p_1, p_2, \dots, p_{i-1}\} \subseteq suspected_i)$  **then**  $broadcast(est, i)$  **endif**

**at time**  $(j-1)d + D$  **for every**  $1 \leq j \leq n$  **do**

**if**  $((p_j \notin suspected_i) \wedge (p_i \text{ has not yet decided}))$  **then**  $return(est_i)$

**endif**

Thank You