Byzantine Generals Problem:

Solution using signed messages

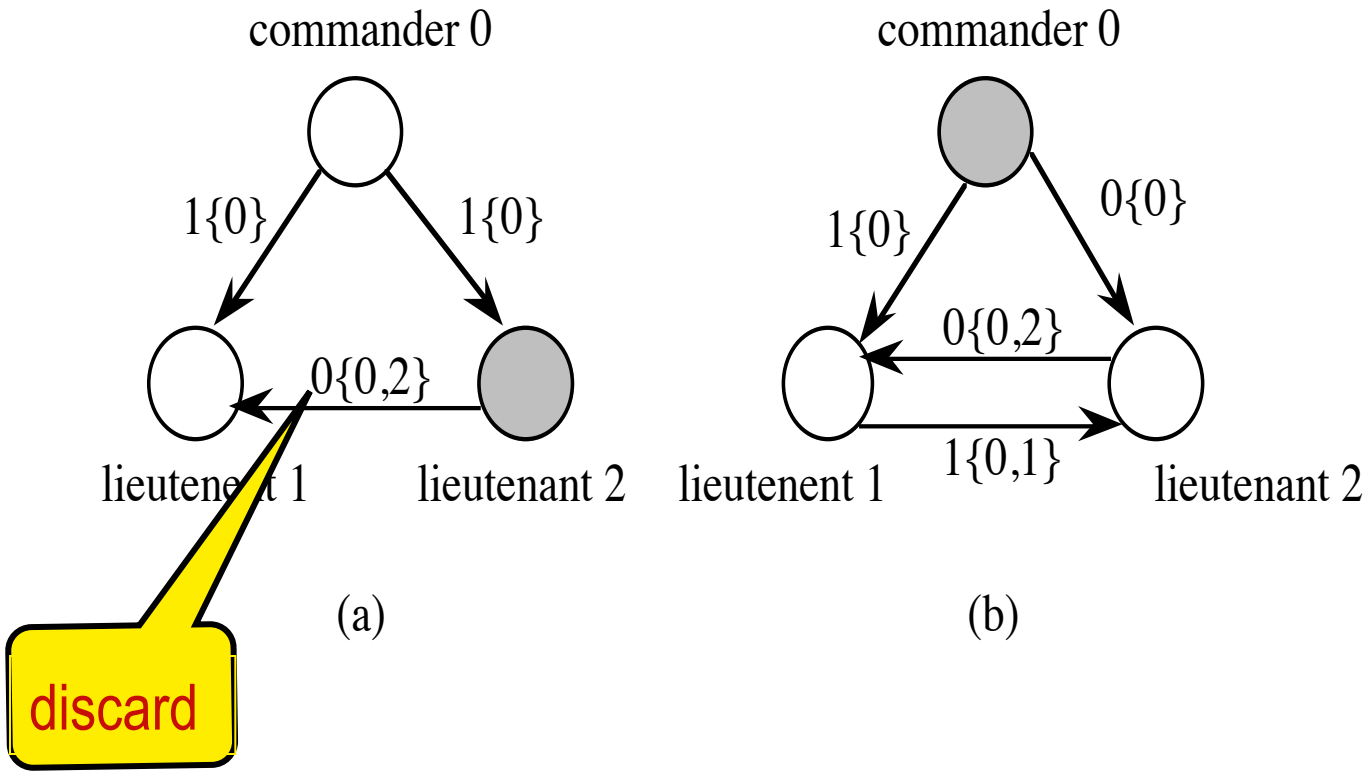
The signed message model

A **signed message** satisfies all the conditions of oral message, plus **two extra conditions**

- Signature cannot be forged. Forged message are detected and discarded by loyal generals.
- Anyone can verify its authenticity of a signature.

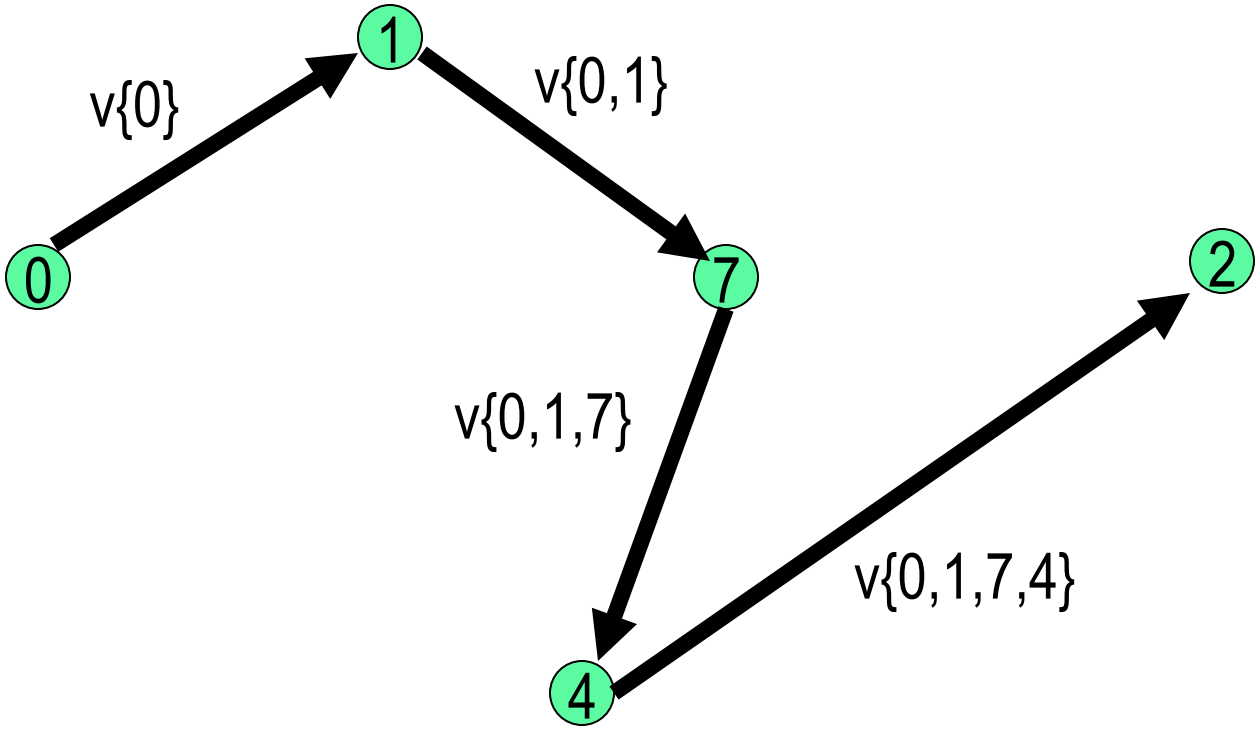
Signed messages improve resilience.

Example



Using signed messages, byzantine consensus ***is feasible*** *with 3 generals and 1 traitor. In (b) the the loyal lieutenants compute the consensus value by applying some choice function on the set of values*

Signature list



Byzantine consensus:
The signed message algorithms SM(m)

Commander **i** sends out a signed message **v{i}** to each lieutenant j ≠ i

Lieutenant **j**, after receiving a message **v{S}**, appends it to a set **V.j,** only if
(i) it is not forged, and (ii) it has not been received before.

If the *length* of **S** is less than **m+1**, then lieutenant **j**
(i) appends his own signature to **S**, and
(ii) sends out the signed message to every other lieutenant
whose signature does not appear in **S**.

Lieutenant **j** applies a choice function on **V.j** to make the final decision.

Theorem of signed messages

If **n ≥ m + 2**, where **m** is the maximum number of traitors, then **SM(m)** satisfies both **IC1** and **IC2**.

Proof.

Case 1. Commander is loyal. The bag of each process will contain exactly one message, that was sent by the commander.

(Try to visualize this)

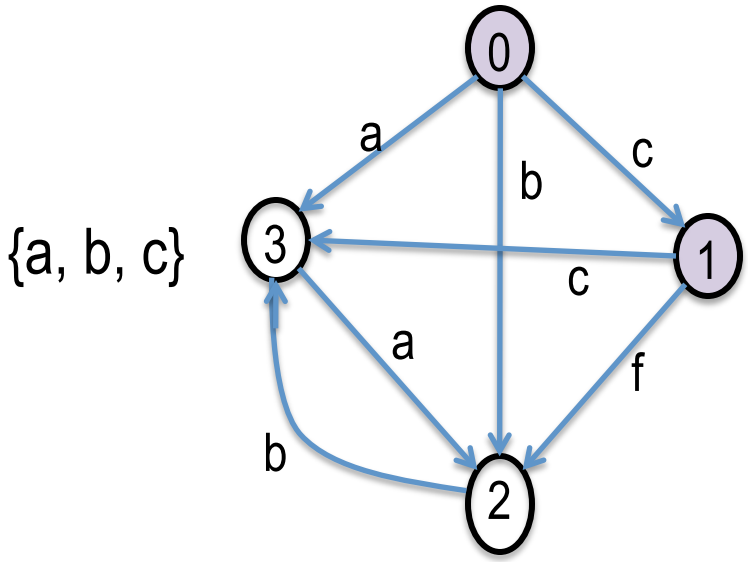
Proof of signed message theorem

Case 2. Commander is traitor.

- The signature list has a size (m+1), and there are m traitors, so at least one lieutenant signing the message must be loyal.

- Every loyal lieutenant i will receive every other loyal lieutenant's message. So, every message accepted by **j** is also accepted by **i** and vice versa. So **V.i = V.j**.

Example



3 accepts c,
but 2 rejects f

With m=2 and a signature list of length 2, the loyal generals may not receive the same order from the commander who is a traitor.
When the length of the signature list grows to 3, the problem is resolved

Concluding remarks

- The signed message version tolerates a larger number (n-2) of faults.

- Message complexity however is the same in both cases.

  Message complexity = (n-1)(n-2) … (n-m+1)

Failure detectors

Failure detector for crash failures

- The design of fault-tolerant algorithms will be simple if processes can detect (crash) failures.

- In synchronous systems with bounded delay channels, crash failures can definitely be detected using timeouts.

Failure detectors for asynchronous systems

In asynchronous distributed systems, the detection of crash failures is imperfect. There will be **false positives** and **false negatives**. Two properties are relevant:

Completeness. Every crashed process is eventually suspected.

Accuracy. No correct process is ever suspected.

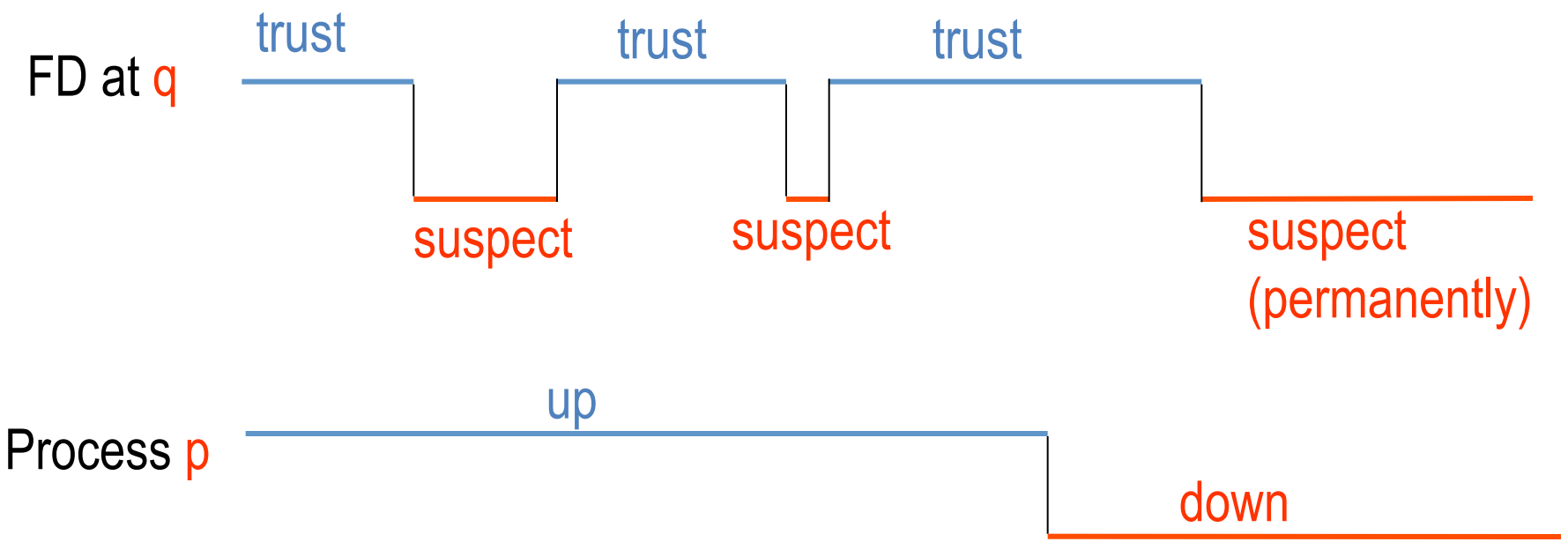
Failure Detectors

An FD is a distributed oracle that provides hints about the operational status of processes.

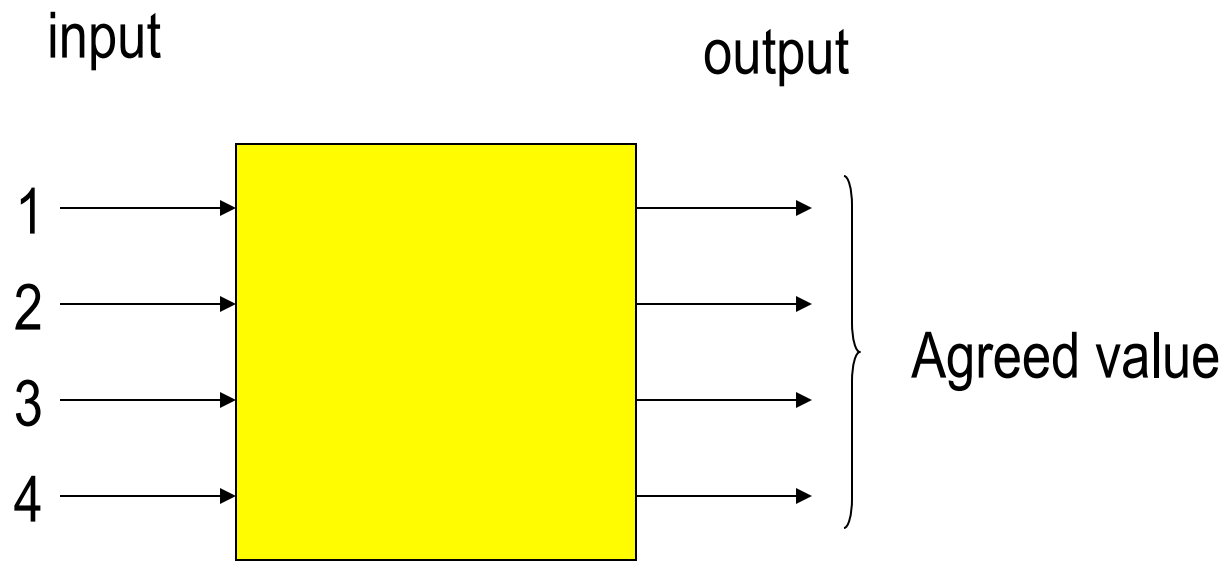
However:

- Hints may be incorrect
- FD may give different hints to different processes
- FD may change its mind (over & over) about the operational status of a process

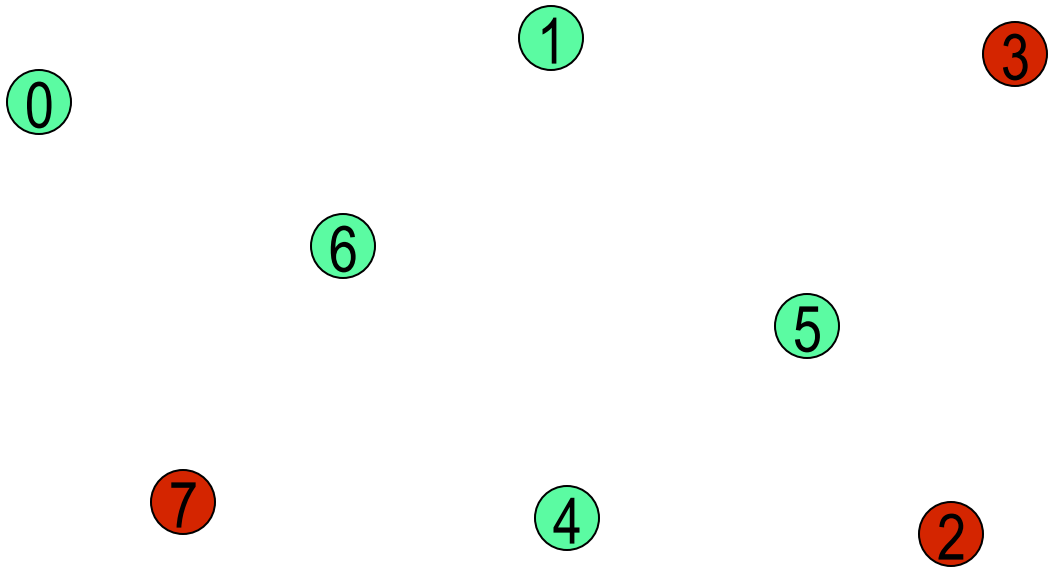
Typical FD Behavior



Revisit the Consensus problem



Example



0 suspects {1,2,3,7} to have failed. Does this satisfy completeness?
Does this satisfy accuracy?

Classification of completeness

- **Strong completeness.** Every crashed process is eventually suspected by *every* correct process, and remains a suspect thereafter.

- **Weak completeness.** Every crashed process is eventually suspected by *at least one* correct process, and remains a suspect thereafter.

Note that we don't care what mechanism is used for suspecting a process.

Classification of accuracy

- **Strong accuracy.** No correct process is ever suspected.

- **Weak accuracy.** There is at least one correct process that is never suspected.

Transforming completeness

Weak completeness can be transformed into strong completeness

Program *strong completeness (program for process i};*
define D: set of process ids (representing the suspects);
initially D is generated by the weakly complete failure detector of i;

{program for process i}
do true →
    send D(i) to every process **j ≠ i;**
    receive D(j) from every process **j ≠ i;**
    D(i) := D(i) ∪ D(j);
    if j ∈ D(i) → D(i) := D(i) \ j **fi**
od

Eventual accuracy

A failure detector is *eventually strongly accurate*, if there exists a time **T** after which no correct process is suspected.

(*Before that time, a correct process be added to and removed from the list of suspects any number of times*)

A failure detector is *eventually weakly accurate*, if there exists a time **T** after which at least one process is no more suspected.

Classifying failure detectors

Perfect P. (Strongly) Complete and strongly accurate

Strong S. (Strongly) Complete and weakly accurate

Eventually perfect ◊P.

(Strongly) Complete and eventually strongly accurate

Eventually strong ◊S

(Strongly) Complete and eventually weakly accurate

Other classes are feasible: W (weak completeness) and weak accuracy) and ◊**W**

Motivation

The study of failure detectors was motivated by those who studied the consensus problem. Given a failure detector of a certain type, how can we solve the consensus problem?

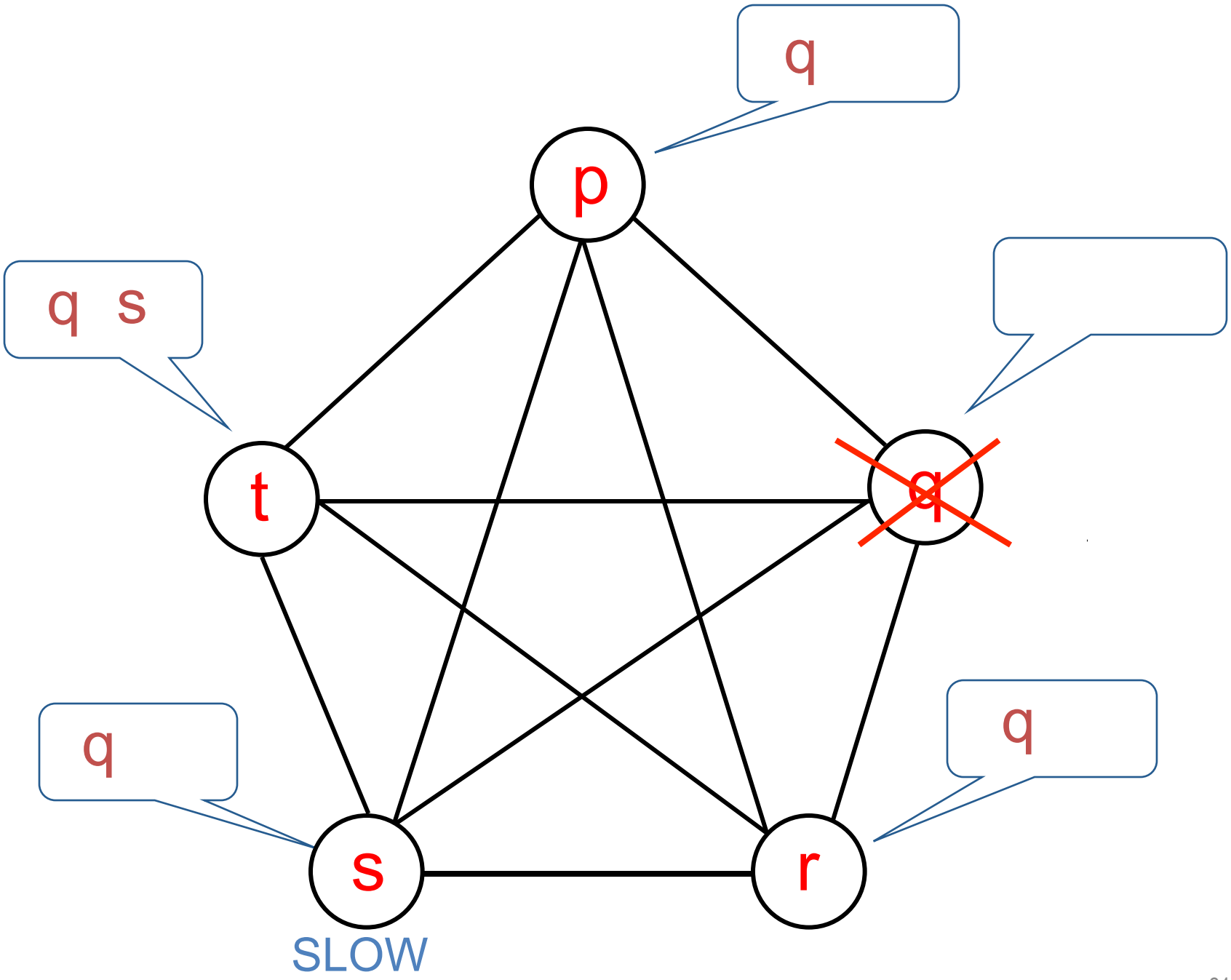
Question 1. How can we implement these classes of failure detectors in asynchronous distributed systems?

Question 2. What is the weakest class of failure detectors that can solve the consensus problem? (*Weakest class of failure detectors is closer to reality)*

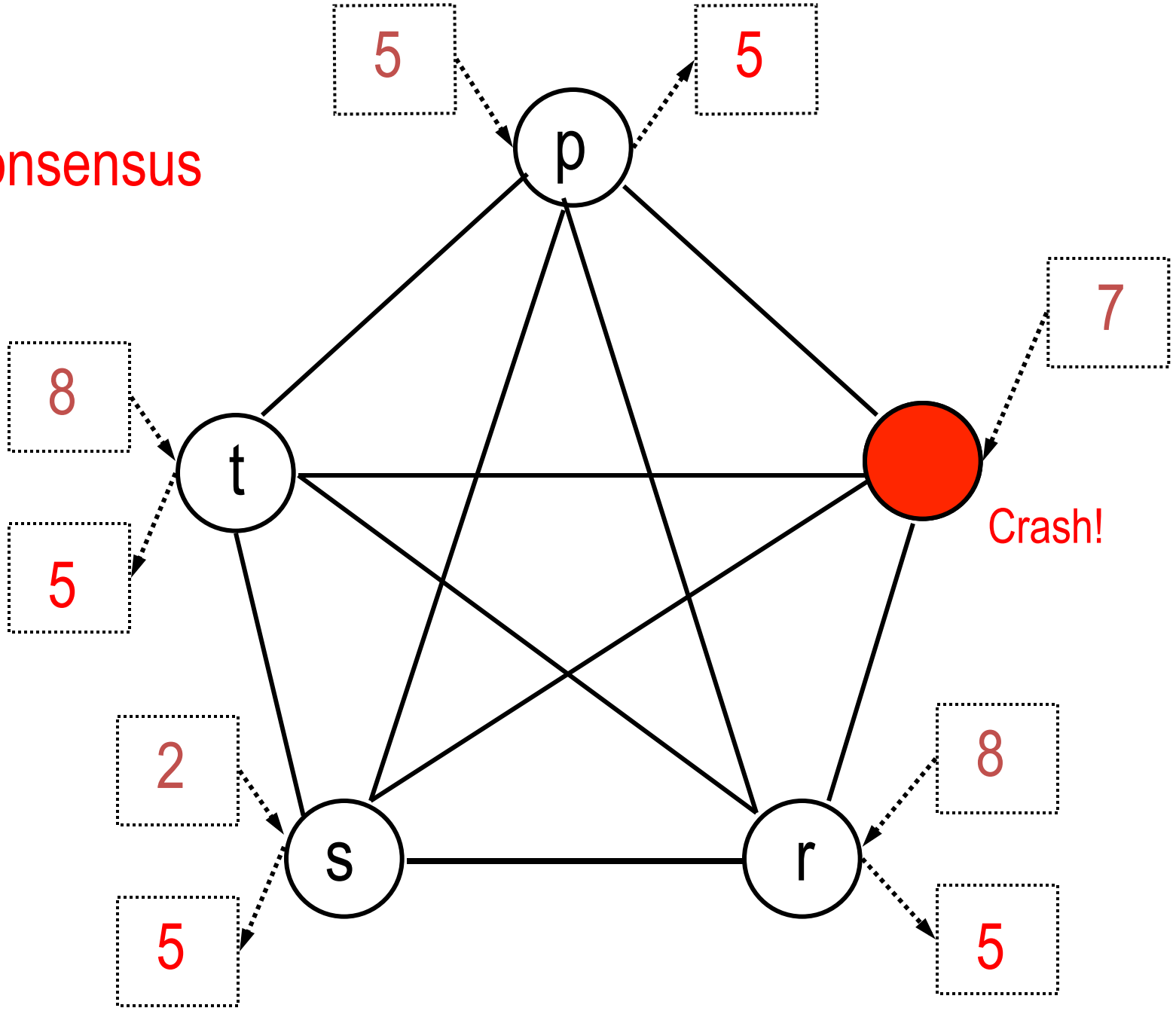
Application of Failure Detectors

Applications often need to determine which processes are up (operational) and which are down (crashed). This service is provided by Failure Detector. FDs are at the core of many fault-tolerant algorithms and applications, like

- Group Membership
- Group Communication
- Atomic Broadcast
- Primary/Backup systems
- Atomic Commitment
- Consensus
- Leader Election
- …..


q
p
q s
t
q
q
s
r
q
SLOW

Consensus



Solving Consensus

- In synchronous systems: Possible

- In asynchronous systems: Impossible [FLP83]

  even if:
  - at most one process may crash, and
  - all links are reliable

A more complete classification of failure detectors

	strong accuracy	weak accuracy	◊ strong accuracy	◊ weak accuracy
strong completeness	**Perfect P**	**Strong S**	**◊P**	**◊S**
weak completeness		Weak W		**◊W**

Motivation

Question 1. Given a failure detector of a certain type, how can we solve the consensus problem?

Question 2. How can we implement these classes of failure detectors in asynchronous distributed systems?

Question 3. What is the weakest class of failure detectors that can solve the consensus problem?

(Weakest class of failure detectors is closest to reality)

Consensus using P

{program for process p, t = max number of faulty processes}

initially V_p := (⊥ , ⊥ , ⊥ , …, ⊥); {array of size n}

$V_p[p]$ = input of p; $D_p := V_p$; $r_p := 1$

{$V_p[q]$ = ⊥ means, process p thinks q **is** a suspect. Initially everyone is a suspect}

{Phase 1} for round r_p**= 1 to t +1**

send (r_p, D_p, p) to all;

wait to receive (r_p, D_q, q) from all q, {or else q becomes a suspect};

for k = 1 **to** n $V_p[k] = \perp \wedge \exists (r_p, D_q, q): D_q[k] \neq \perp \rightarrow V_p[k] := D_q[k]$

end for

end for

{at the end of Phase 1, V_p for each correct process is identical}

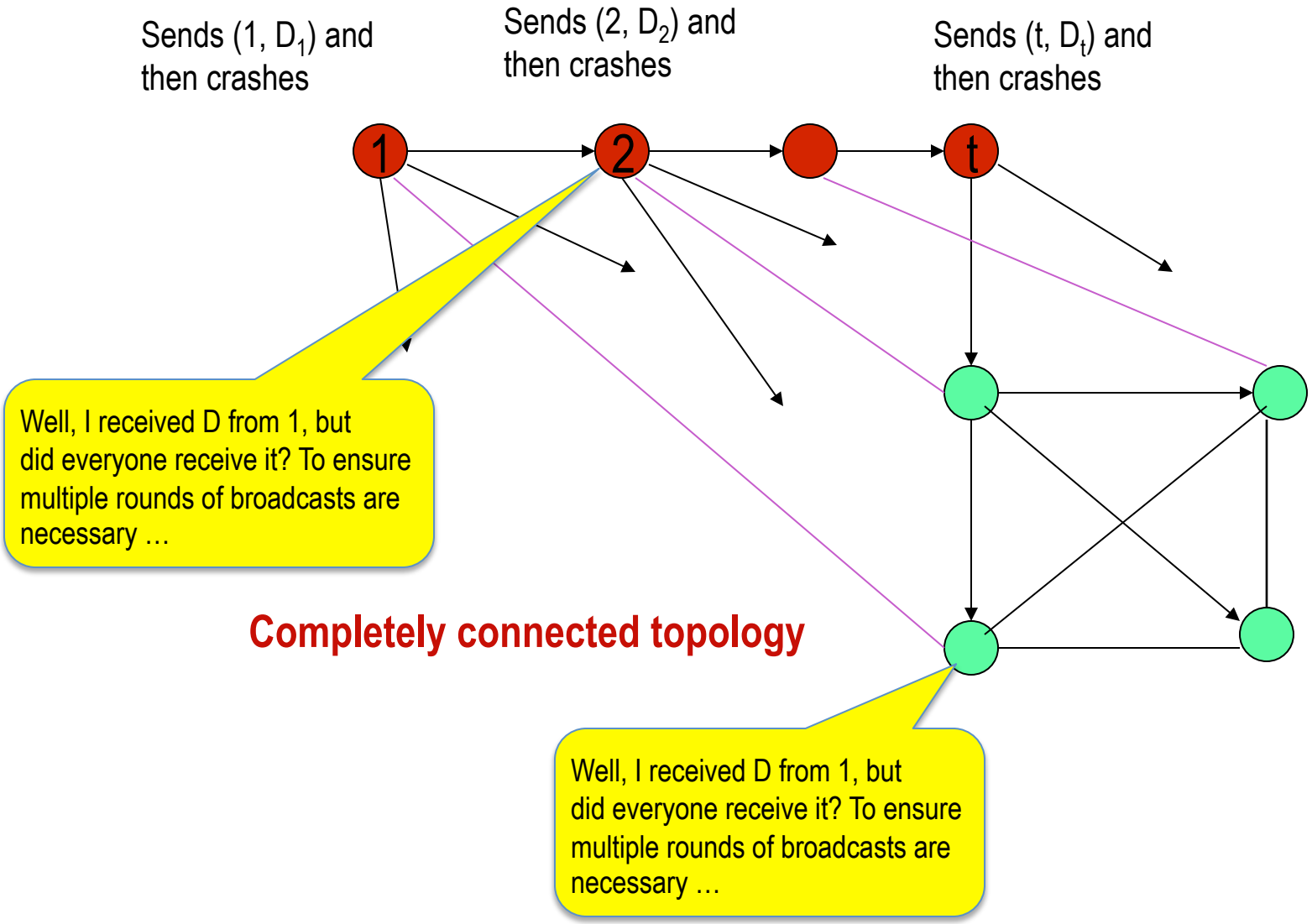
{Phase 2} Final decision value is the input from the *first* element $V_p[j]$: $V_p[j] \neq \perp$

Understanding consensus using P

Why continue (t+1) rounds?

It is possible that a process **p** sends out the first message to **q** and then crashes. If there are **n** processes and **t** of them crashed, then after at most **(t +1)** asynchronous rounds, $\mathbf{V_p}$ for each correct process **p** becomes identical, and contains all inputs from processes that may have transmitted at least once.

Understanding consensus using P



Consensus using other type of failure detectors

Algorithms for reaching consensus with several other forms of failure detectors exist. In general, the weaker is the failure detector, the closer it is to reality (a truly asynchronous system), but the harder is the algorithm for implementing consensus.

Consensus using S

$V_p := (\perp, \perp, \perp, \ldots, \perp)$; $V_p[p] :=$ input of p; $D_p := V_p$

(Phase 1) Same as phase 1 of *consensus with* P – it runs for (t+1) asynchronous rounds

(Phase 2) send (V_p, p) to all;

receive (D_q, q) from all q;

for k = 1 **to** n $\exists V_q[k]: V_p[p] \neq \perp \wedge V_q[k] = \perp \rightarrow V_p[k] := D_p[k] := \perp$ **end for**

(Phase 3) Decide on the *first* element $V_p[j]: V_p[j] \neq \perp$

Consensus using S: example

Assume that there are six processes: 0,1,2,3,4,5. Of these 4, 5 crashed. And 3 is the process that will never be suspected. Assuming that k is the input from process k, at the end of phase 1, the following is possible:

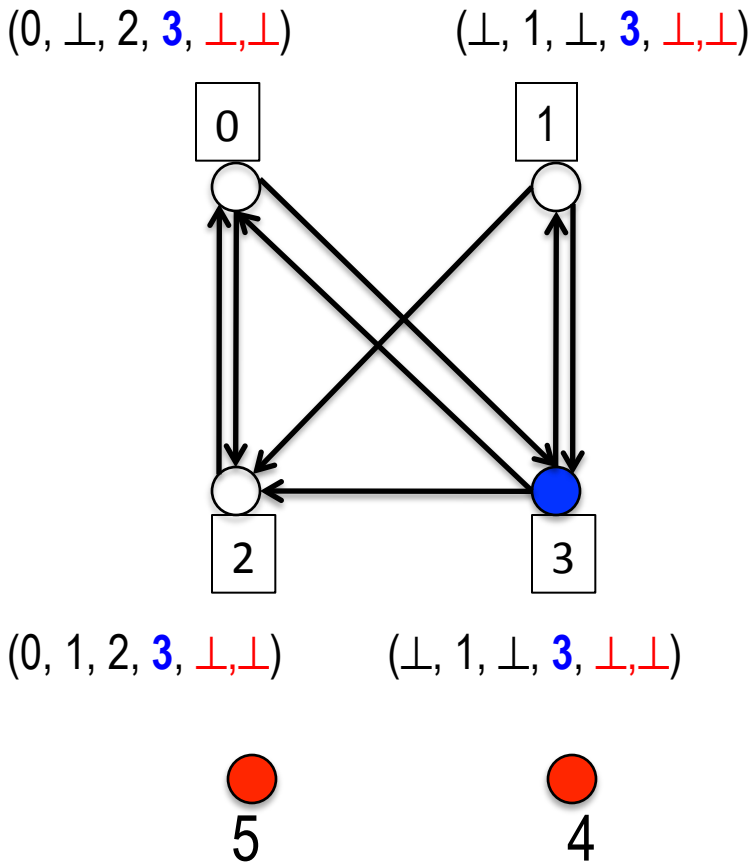
V0 = (0, ⊥, 2, **3**, ⊥,⊥)

V1 = (⊥, 1, ⊥, **3**, ⊥,⊥)

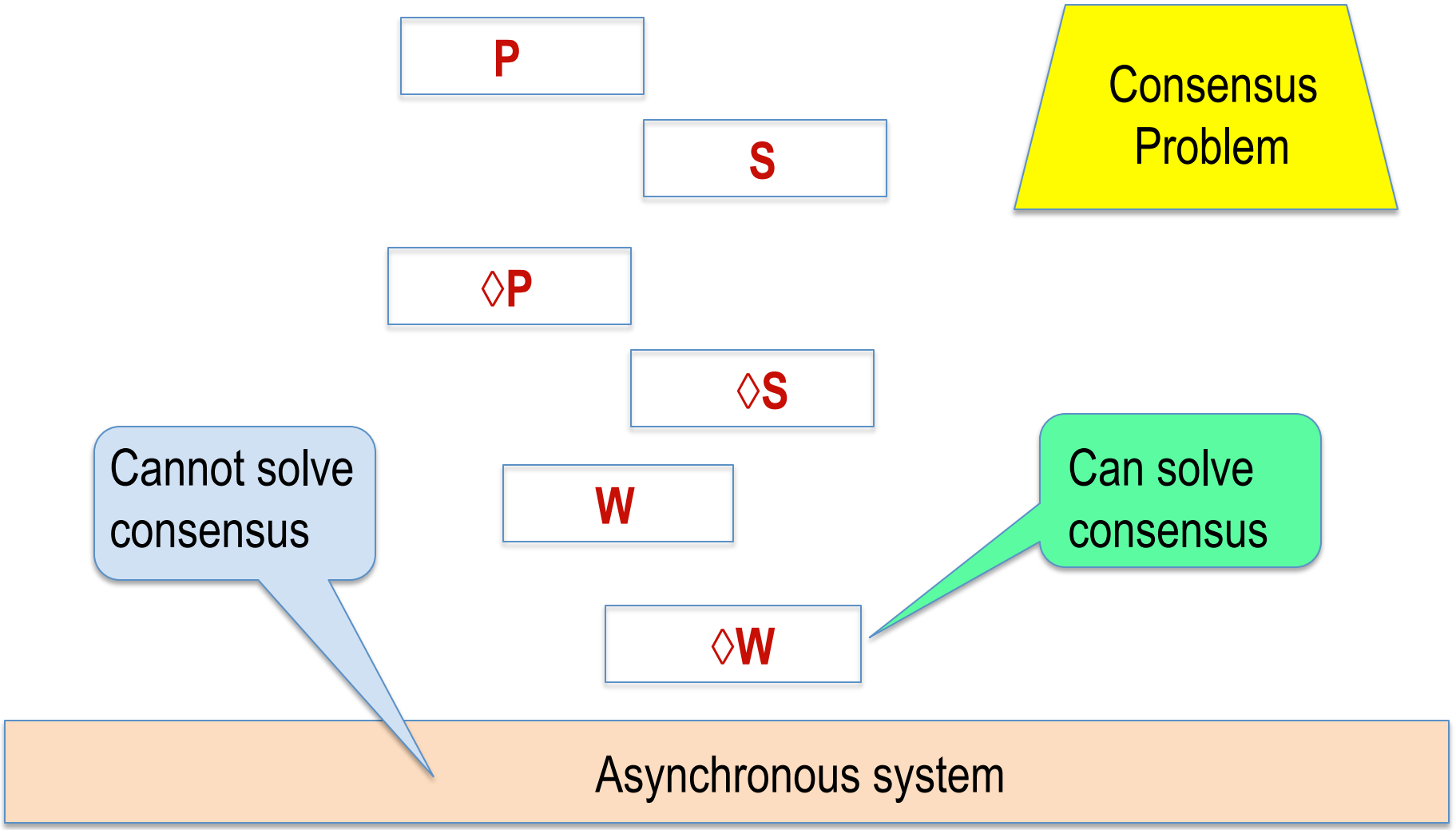
V2 = (0, 1, 2, **3**, ⊥,⊥)

V3 = (⊥, 1, ⊥, **3**, ⊥,⊥)

At the end of phase 3, the processes agree upon the input from process 3



Conclusion


P
S
◊P
◊S
W
◊W
Consensus Problem
Cannot solve consensus
Can solve consensus
Asynchronous system