Problemi e Algoritmi distribuiti

Problemi nei sistemi distribuiti

Problemi di coordinamento:

- Mutua esclusione
- Leader Election
- Consenso

Problemi di comunicazione ordinata:

- Comunicazione FIFO
- Comunicazione causale
- Comunicazione totalmente ordinata

Specifica di un Problema

- Si ha un problema che deve essere risolto in un sistema distribuito attraverso un algoritmo deterministico

- La **specifica** del problema viene espressa nei termini di due proprietà:

  - *Safety* (nothing "bad" will happen). Definisce quali sono i comportamenti ammissibili. Se un algoritmo soddisfa questa proprietà non esibirà mai un comportamento non ammesso (bad).

  - *Liveness* (something "good" will eventually happen). Ha a che fare col concetto di "progresso" del sistema. Un algoritmo che soddisfa safety è anche un algoritmo che non esegue alcuna azione (in questo modo di certo non eseguirà azioni bad). In questo caso il concetto di liveness impone all'algoritmo di eseguire azioni prima o poi (eventually).

Relazione tra problemi, algoritmi e modelli

- Un problema può essere **risolvibile** (*ovvero si può definire un algoritmo deterministico che soddisfi la specifica*) in alcuni modelli di sistema distribuito (specificato nei termini di assunzioni di sincronia, tipi di guasti, topologia, determinismo dei processi) in modo semplice, in altri modelli in modo più complesso.

- *Lo stesso problema puo' essere risolvibile in un sistema ed impossibile da risolvere in un altro*.

Relazione tra problemi, algoritmi e modelli
esempio

- **Problema**: *Fissare un Appuntamento (orario e luogo) tra due persone*

- **Specifica**:
  - *Safety*: le due persone non si devono recare in luoghi diversi o nello stesso luogo ad orari diversi
  - *Liveness*: in un tempo finito le due persone fisseranno un appuntamento

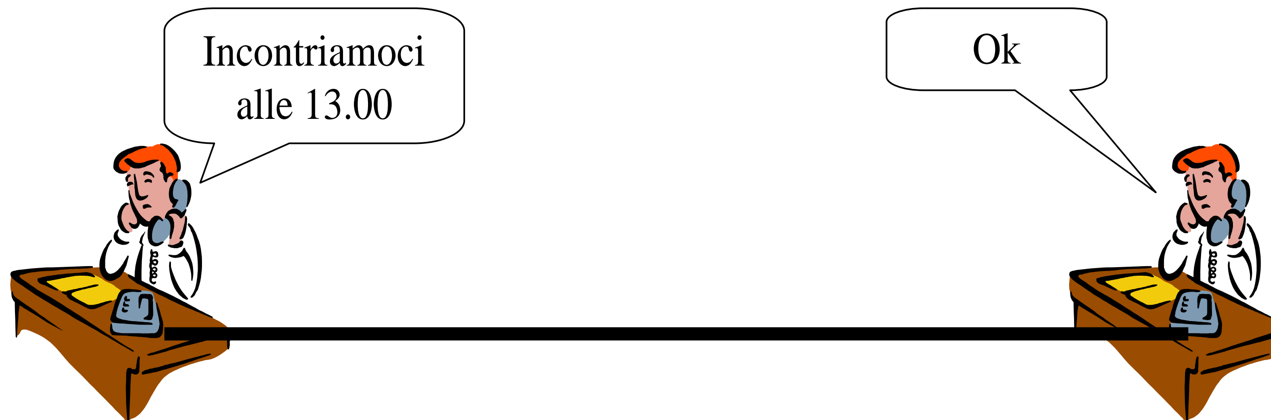
- Per quanto riguarda l'algoritmo? Ho diversi algoritmi in funzione del modelllo di sistema assunto, analizziamo tre sistemi differenti tra loro: sistema orale, telefonico e di e-mail

Sistema Orale

- Modello di sistema:
  - Sincrono: esistenza di un limite (bound) temporale
    - sulla consegna dei messaggi nel canale di comunicazione
    - sul tempo che ci mette l'interlocutore (processo) a rispondere
  - No assunzione di guasti
  - Topologia rete: punto-punto

- ***l'algoritmo di risoluzione necessita di 2 msg***

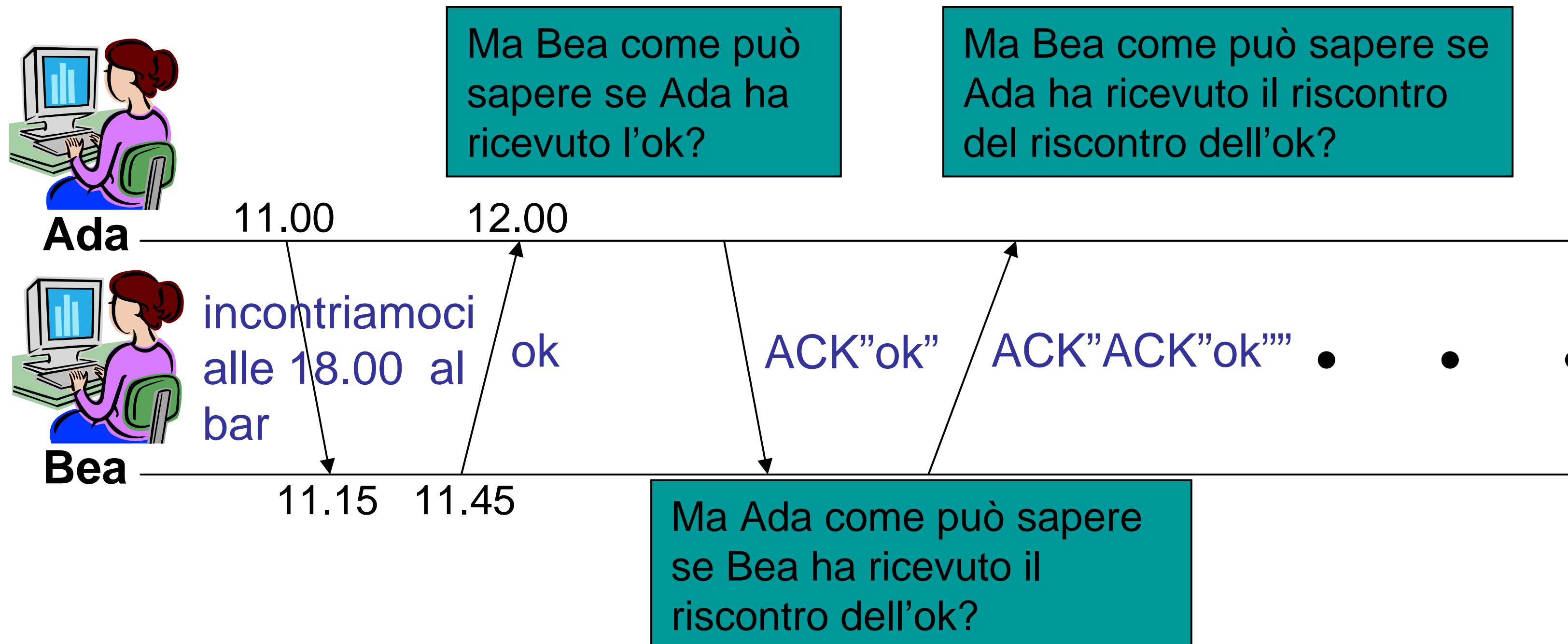
Sistema Telefonico



- Modello di sistema:
  - Sincrono: esistenza di un limite (bound) temporale
    - sulla consegna dei messaggi nel canale di comunicazione
    - sul tempo che ci mette l'interlocutore (processo) a rispondere
  - Guasti: failstop (la caduta della linea è un evento rilevabile)
  - Topologia rete: punto-punto

- ***l'algoritmo di risoluzione necessita di 2 msg se i processi sono corretti. Si noti che la specifica, in assunzione di guasti, viene in genere definita solo su processi corretti***

Sistema di e-mail



- Modello di sistema:
  - Asincrono: tempo di trasmissione di un messaggio impredicibile
  - Guasti: Omission
  - Topologia rete: punto-punto
- ***l'algoritmo di risoluzione non esiste!***

Progettazione di un algoritmo

- Quando si progetta un algoritmo distribuito e' necessario essere molto chiari sui seguenti punti:

  - **Specifica** del **problema** computazionale

  - Definire il **modello di sistema distribuito** sul quale gira l'algoritmo tenendo sempre in mente che possono esistere modelli di sistema in cui il problema dato non è risolvibile.

Problema del Consenso

- Il gruppo di processi devono mettersi d'accordo su un valore (es. commit/abort di una transazione). E' l'astrazione di una classe di problemi in cui i processi partono con le loro "opinioni" (forse divergenti) e devono **accordarsi** su un'opinione comune

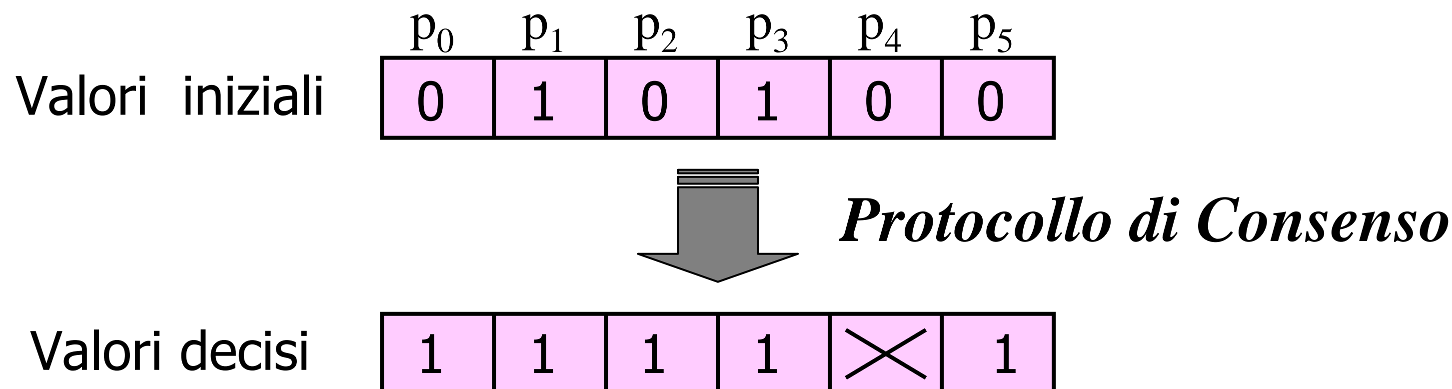
- E' un problema fondamentale: qualsiasi soluzione per mutua esclusione, leader election, comunicazione totalmente ordinata risolve il Consenso

- Con guasti? Impossibilità di risolverlo in sistemi asincroni

- Per le sue caratterisitiche studieremo algoritmi e modelli "minimali" per risolvere questo problema *in presenza di guasti*

Consenso\definizione

- Insieme di processi distribuiti con valori iniziali ∈ {0,1}.
- Tutti devono decidere lo stesso valore ∈ {0,1}, basandosi sui loro stati iniziali.



Nota 1: esistono stati iniziali (dei processi) per i quali la decisione è 0 e altri per i quali la decisione raggiunta è 1 (si evitano protocolli di consenso banali in cui la decisione è sempre la stessa).

Nota 2: l' assunzione di valori ∈ {0,1} semplifica la discussione.

Consenso\specifica

- **Agreement**: Se un processo corretto decide un valore *v*, allora tutti i processi corretti decidono *v*
- **Integrity**: Ogni processo decide al più una volta, e se decide per *v* allora qualche processo ha precedentemente proposto *v*

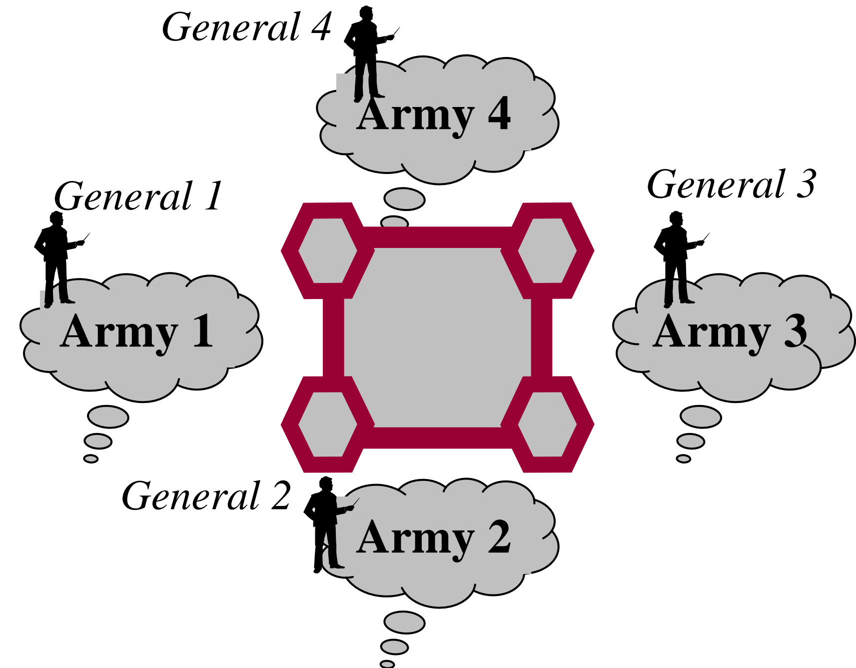
safety

- **Termination**: Ogni processo corretto alla fine decide un valore.

liveness

Consenso\esempio

- 4 eserciti alleati, ognuno dei quali guidato da un generale, assediano un castello.
- Un esercito senza un generale non combatte.
- Per impadronirsi del castello **tutti e quattro** devono attaccare.
- **Comunicazioni** (messaggeri) **affidabili**, ma impiegano un tempo impredicibile
- **I generali possono essere uccisi**



I generali devono raggiungere un consenso e decidere tra ATTACCO e RITIRATA

Se non c'è risposta da un generale? morto / pigro / comunicazione ritardata?

I generali riusciranno a decidere tra ATTACCO e RITIRATA? **NO!**

Impossibility Result

Impossibilità del Consenso
in Sistemi Asincroni

Fisher, Lynch e Patterson hanno provato nel 1985 che è **impossibile risolvere il problema del Consenso in modo deterministico in sistemi asincroni in cui anche un solo processo puo' guastarsi per crash** (**FLP result**).
Ref: *Journal of the ACM, Vol. 32, No. 2, April 1985*.

Impossibilità e modelli di sistema

Sistema Asincrono:

➕ **Modello *attraente*: no assunzioni di sincronia**

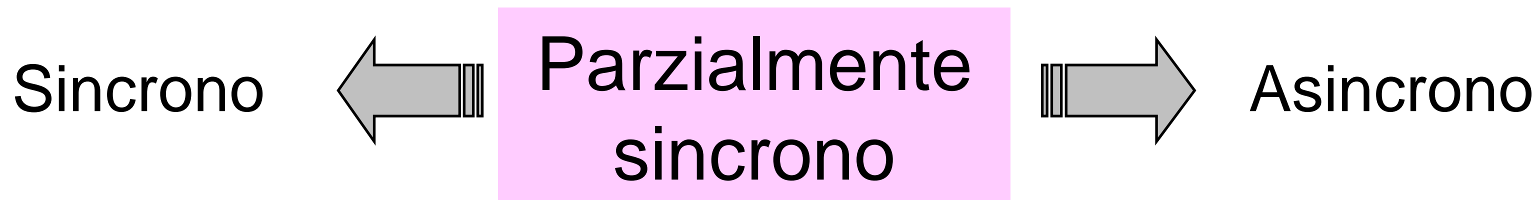
➖ **Modello *debole* per la tolleranza ai guasti:**
Il Consenso non può essere risolto in tale modello, anche supponendo che esista *un solo* processo che può guastarsi e i canali siano affidabili (FLP)!

Sistema Sincrono:

➕ **Modello *attraente* per la tolleranza ai guasti**

➖ **Modello *debole* a causa dell'assunzione di sincronia: difficile che un sistema reale rispetti *sempre* tali assunzioni**

Aggirare FLP:
il modello parzialmente sincrono



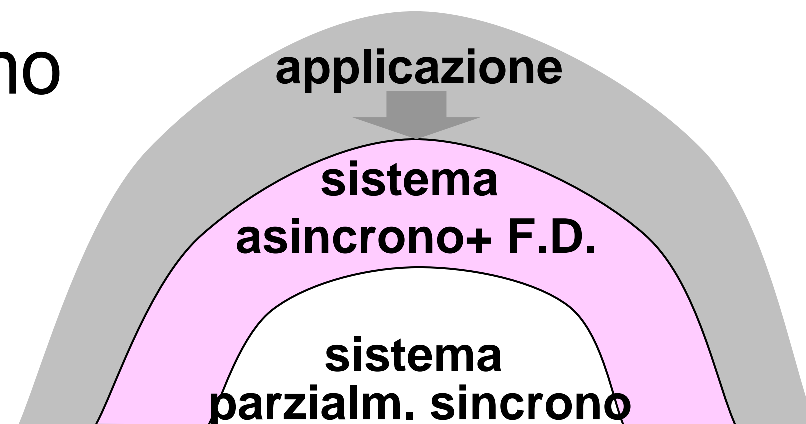
- Vengono fatte alcune assunzioni di sincronia necessarie per risolvere il Consenso. Esempi
  - processi con velocità limitata e clock sincronizzati, oppure
  - ritardo dei messaggi limitato ma sconosciuto

- Si sfrutta la *sincronia parziale* esibita dalla maggiorparte dei sistemi reali: e.g. ritardo di msg e/o tempo di transizione di stato dei processi LIMITATO nella vasta maggioranza dei casi

- In questo caso il Consenso (e problemi correlati) viene risolto usando timeouts per rilevare i processi guasti.

Failure detectors

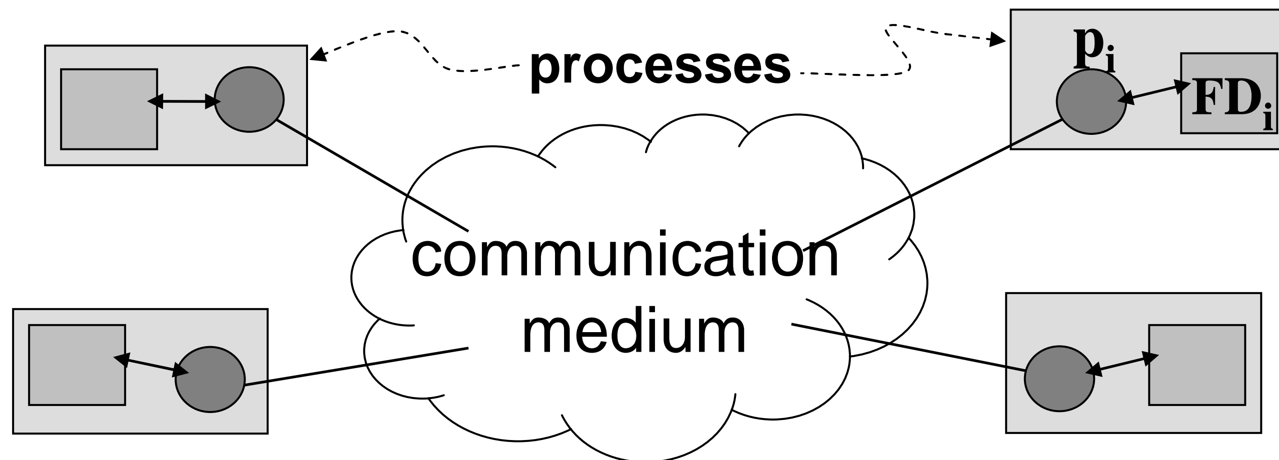
- Progettare algoritmi che integrino le assunzioni di sincronia non è agevole!
- **Failure Detectors**: servizio di sistema per incapsulare le proprietà di sincronia parziale.
- Le applicazioni vengono sviluppate come per **Sistema Asincrono + Failure Detectors**



Failure detectors

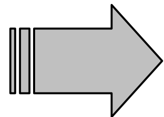
- **Failure Detector:** oracolo ***distribuito***.

- Ogni processo applicazione p_i ha un **modulo FD locale** FD_i**.** Un modulo FD mantiene una lista di processi del sistema "sospettati" di essere guasti. I processi consultano i loro FD locali per informarsi dello stato degli altri processi. I moduli FD scambiano info sui processi sospettati. Un modulo FD potrebbe erroneamente sospettare un processo; un processo potrebbe essere rimosso dalla lista dei "sospettati" - ***unreliable*** FDs.



Failure Detector\specifica

- L'algoritmo dell'applicazione si basa su **FD specs** e **non** su **implem. di FD**. In questo modo:
  - ⇒ L'applicazione indipendente dalle vere assunzioni di sincronia sottostanti
  - ⇒ Applicazione portabile in differenti (parzialm. sincr.) modelli
- Cambiando modelli di sistema, solo l'implementazione di FD (s/w o h/w) deve cambiare (non l'implementazione dell'applicazione)!



specifica di FD importante per la progettazione dell'applicazione!

2 proprietà

- **completeness:** "i processi **guasti** nel sistema sono **sospettati**"
- **accuracy:** "i processi **corretti** nel sistema **non** sono sospettati"

Failure Detector\specifica

Perfect Failure Detector:

- **Strong Completeness**: **Ogni** processo guasto è alla fine sospettato permanentemente da **ogni** processo corretto.

- **Strong Accuracy**: **Nessun** processo corretto è mai sospettato da un altro processo corretto.

Perfect FD molto ***conveniente*** per i programmatori di applicazioni, ma ***difficile*** da implementare, specialmente in sistemi parzialmente sincroni "deboli"...

indeboliamo il Perfect FD …

Failure Detector\specifica

... in termini di **Completeness** :

- **Weak Completeness**: **Ogni** processo guasto è alla fine sospettato permanentemente da **qualche** processo corretto.

... in termini di **Accuracy** :

- **Weak Accuracy**: **Qualche** processo corretto non è mai sospettato da un altro processo corretto.

- **Eventual Strong Accuracy**: *Esiste un tempo dopo il quale* **nessun** processo corretto viene più sospettato da un altro processo corretto.

- **Eventual Weak Accuracy**: *Esiste un tempo dopo il quale* **qualche** processo corretto non viene più sospettato da un altro processo corretto.

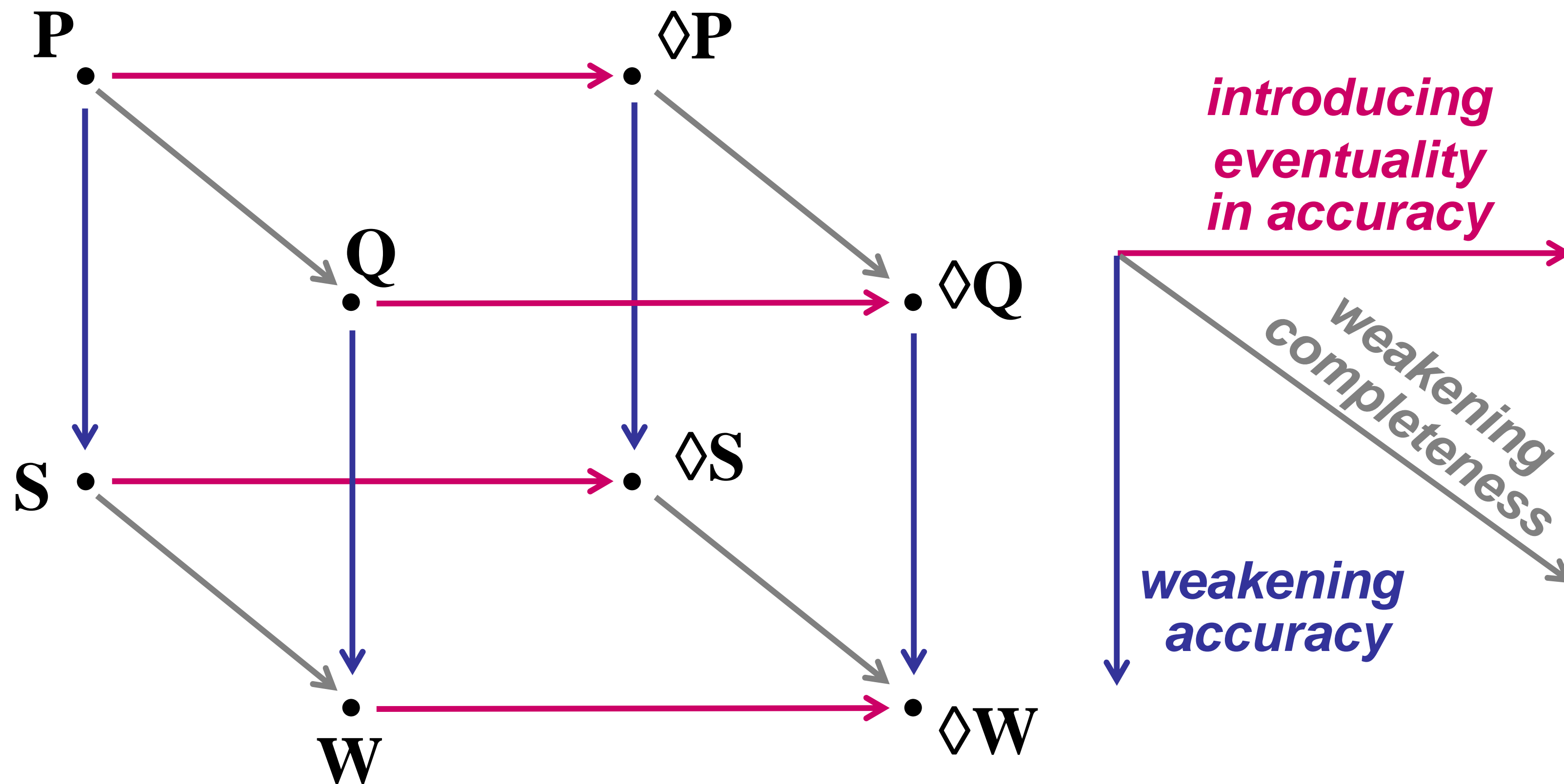
Classi di Failure Detector

2 versioni di completeness **x** 4 versioni di accuracy = 8 FDs

Completeness \ Accuracy	**S**	**W**	**ES**	**EW**
S	P	S	◊P	◊S
W	Q	W	◊Q	◊W

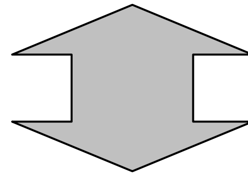
Ref: Chandra and Toueg "*Unreliable failure detectors for reliable distributed systems*", Journal of the ACM, 43(2), March 1996.

Comparazione tra Failure Detectors



Comparazione tra Failure Detectors

- **D → D′: proprietà di D implicano logicamente le proprietà di D′**



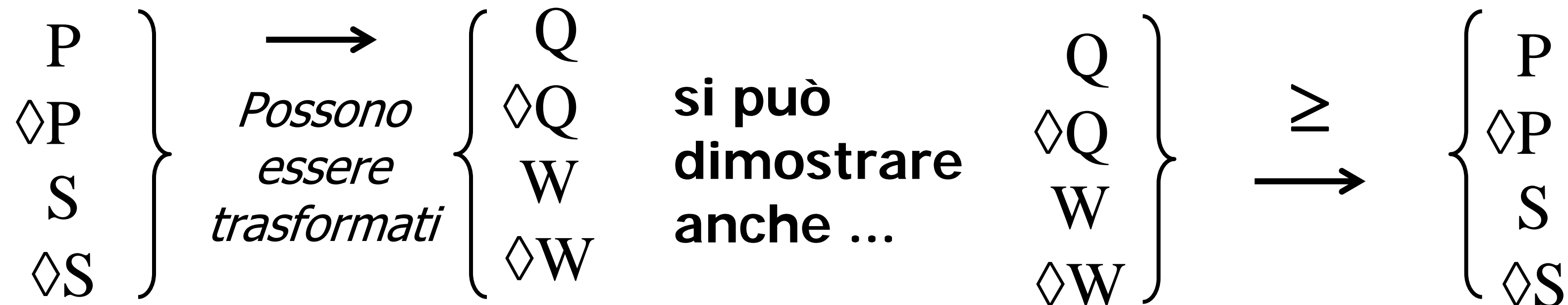
- **D ≥ D′ : D è computazionalmente "stronger" di D' (D assicura qualcosa in più rispetto a D')**

⇒ Se un problema **Pb** può essere risolto usando **D'** e **D** ≥ **D'** allora **Pb** può essere risolto usando **D.**

⇒ **D** può essere **trasformato** in **D'**

Comparazione tra Failure Detector

- Le frecce nel "cubo" **apparentemente** indicano relazioni di "forza"...



- Gli insiemi di FD **{Q,◊Q,W,◊W}** sono rispettivamente **equivalenti** a **{P,◊P,S,◊S}**, quindi

Q ≡ P

◊Q ≡ ◊P

W ≡ S

◊W ≡ ◊S

- cosa significa?

  weak e strong completeness sono equivalenti

Risolvere il consenso con FD

- Quindi **Q** ≡ **P**, ◊**Q** ≡ ◊**P**, **W** ≡ **S** e ◊**W** ≡ ◊**S** ma cosa implica questa equivalenza?

- L'equivalenza significa che se troviamo un algoritmo per risolvere il Consenso usando un FD in **{Q**,◊**Q**,**W**,◊**W}**, allora il suo FD equivalente in **{P**,◊**P**,**S**,◊**S}** può essere automaticamente trasformato per risolvere il Consenso con lo stesso algoritmo e **viceversa.**

- Le proprietà degli FD in **{P**,◊**P**,**S**,◊**S}** sembrano intuitivamente "stronger". Quindi, progettiamo algoritmi per risolvere il Consenso usando **P**, ◊**P**, **S**, **e** ◊**S** …

Algoritmi di risoluzione

- Vedremo tre algoritmi di risoluzione che impiegano tre classi di FD diversi, rispettivamente P, S, ◊S.
- Si noti che l'algoritmo che usa il FD più forte (P) sarà anche l'algoritmo più semplice, mentre l'algoritmo che usa ◊S sarà il più complesso tra i tre.
- Non esistono algoritmi che possono sfruttare eventuali FD più deboli di ◊S per risolvere il consenso.

Algoritmi di risoluzione\modello

- Per tutti e tre gli algoritmi il **modello di sistema** è il seguente:
  - il sistema è asincrono*
  - il modello di guasto è il crash
  - la rete è fully interconnected
  - i canali sono reliable

*Nota che in realtà il modello di sistema esibisce le assunzioni di sincronia necessarie per implementare il FD considerato, tuttavia il progettista della soluzione può considerare il sistema asincrono (progettare l'algoritmo ignorando le caratteristiche di sincronia del sistema).

Algoritmo usando FD di classe P

- Indichiamo con f il massimo numero di processi che possono guastarsi

- Ogni processo $\mathbf{p_i}$ mantiene un vettore $\mathbf{V_i}$[1..n], che contiene l'insieme di valori proposti dagli altri processi di cui $\mathbf{p_i}$ è a conoscenza. Quindi $\mathbf{V_i}$[j] è il valore proposto da $\mathbf{p_j}$ secondo la conoscenza del processo $\mathbf{p_i}$ (a quanto sa $\mathbf{p_i}$, $\mathbf{p_j}$ ha proposto $\mathbf{V_i}$[j])

- Inizialmente $\mathbf{p_i}$ conosce solo il valore proposto da se stesso

Algoritmo usando FD di classe P

L'algoritmo funziona basandosi su *round asincroni.* E' costituito da 2 fasi:

I. costituita da f+1 round. In ogni round:
   - ogni processo manda a tutti gli altri processi valori di V che non ha mai mandato prima. Quindi nel primo round manda solo il valore da lui proposto, mentre nei round successivi manda solo quei valori che ha visto per la prima volta nel round precedente
   - ogni processo riceve valori dagli altri processi. Aspetta un messaggio da tutti i processi che non sono nella lista dei sospettati fornita dal FD

II. Dopo f+1 round ogni processo decide il primo valore non null presente nel vettore V

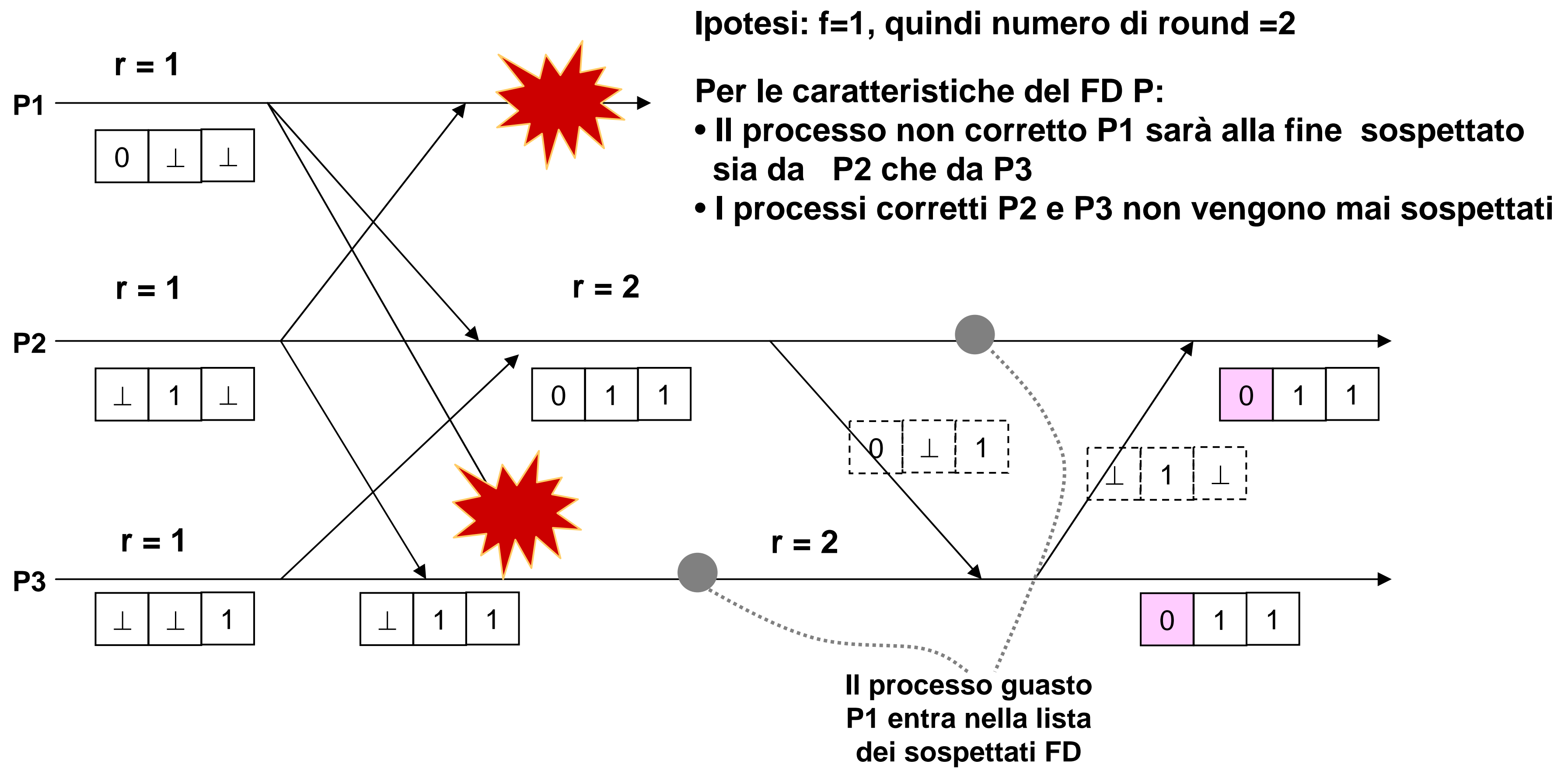
Algoritmo usando P\pseudo-codice

```
Vi:= ⟨⊥, ⊥,... ⊥⟩ ; Vi[i]:=v;   Δi:= Vi

for  ri := 1 .. f+1  do   // f =massimo #  di processi guasti
    send(⟨ri, Δi, i⟩) to all;
    wait until ( for all j : recv(⟨ri, Δj, j⟩) or j∈FDi );
    Δi  := ⟨⊥ .. ⊥⟩;
    for k := 1 .. n do
        if  (received ⟨ri, Δ, j⟩  t.c. Δ[k] ≠⊥  and  Vi[k]=⊥)
                then Vi[k] := Δi [k]:= Δ[k];


decisioni:= prima entry Vi[j] ≠⊥
```

Algoritmo usando P\esempio



Algoritmo usando P\liveness

Termination (ogni processo corretto alla fine decide un valore)

- Supponiamo *per assurdo* che un processo corretto pi non riesca a decidere. Dallo pseudo-codice ciò implica che il processo rimane bloccato alla condizione di wait. Supponiamo rimanga bloccato al round r. In questo caso esiste un messaggio proveniente da un processo pj, non presente nella lista dei sospettati, che non arriva mai. Due casi:
  - Il processo mittente è corretto: in questo caso dalle proprietà dei canali il messaggio non può perdersi e verrà alla fine ricevuto dal processo pi. *Contraddizione*
  - Il processo mittente non è corretto. In tal caso dalla proprietà di completness del failure detector il processo mittente entrerà alla fine nella lista dei sospettati di ogni processo corretto. *Contraddizione*

Algoritmo usando P\safety

Agreement (se un processo corretto decide un valore *v*, allora tutti i processi corretti decidono *v*)

- Supponiamo per assurdo che esistano due processi corretti pi e pj che decidono due valori diversi. Se entrambi hanno deciso allora sono arrivati al round f+1 con due diversi set di valori. Senza perdita di generalità supponiamo che pi abbia un valore che pj non possiede. Dallo pseudo-codice segue che l'unico modo per pi di avere alla fine della computazione un valore non posseduto da pj è che un'altro processo pk abbia inviato nell'ultimo round il valore a pi ma abbia fatto crash prima che pj potesse ricevere questo valore. Poichè pj non ha mai visto questo valore segue che stiamo supponendo che ci sia stato un crash in ogni round, crash riguardante il processo che inviava il valore non visto da pj. Poichè si è supposto che il numero max di guasti è f e il numero di round è f+1 si arriva ad una contraddizione.

Algoritmo usando P\safety

Integrity (Ogni processo decide al più una volta, e se decide per *v* allora qualche processo ha precedentemente proposto *v*)

- Supponiamo per assurdo che esista un processo pi che decide un valore che nessun processo ha proposto. Significa che decide per una entry che contiene un valore non proposto da alcun processo. Tuttavia ogni entry per i ≠ j è aggiornata solo quando viene ricevuto un valore proposto dal processo pj. Dalle proprietà dei canali nessun messaggio può essere danneggiato. Ciò significa che il valore rcievuto corrisponde al valore inviato. Inoltre per i=j il valore contenuto è il valore proposto da pi stesso dall'inizializzazione dell'algoritmo. Contraddizione
- Supponiamo per assurdo che un processo decide due volte. Dallo pseudo-codice segue immediatamente che la linea di decisione viene eseguita al più una volta. Contraddizione

Algoritmo usando S

Rivediamo dapprima le proprietà del FD di tipo S:

classe S (“Strong FD”):

- **Weak Accuracy**: **qualche** processo corretto non è **mai** sospettato.
- **Strong Completeness**: **Ogni** processo guasto è alla fine sospettato permanentemente da **ogni** processo corretto.

Algoritmo con FD S \implementazione

- Questo algoritmo tollera al più n-1 processi guasti
- Rispetto all'algoritmo che usa FD P deve eseguire una fase in più:

I. La prima fase è in sostanza identica alla prima fase dell'algoritmo precedente solo che in questo caso dura per n-1 round asincroni

II. **Questa è la fase in più rispetto al precedente algoritmo: Viene inviato il vettore dei valori proposti a tutti i processi. Quindi p_i aspetta il vettore dei valori proposti da ogni processo non contenuto nella lista dei sospettati e aggiorna il vettore. Alla fine di questa fase il vettore dei valori proposti è lo stesso in ogni processo corretto. Il k-simo elemento o contiene il valore proposto dal processo p_k oppure contiene un valore null**

III. La terza fase è la decisione del valore. Procede come nel precedente algoritmo

Algoritmo usando S\pseudo-codice(1)

V_i:= ⟨⊥, ⊥,... ⊥⟩ ; V_i[i]:=v; Δ_i:= V_i

for r_i := 1 .. **n-1** **do**

  send(⟨r_i, Δ_i, i⟩) to all;

  wait until (**for all** j : recv(⟨r_i, Δ_j, j⟩) or j∈FD_i);

  Δ_i := ⟨⊥ .. ⊥⟩;

  for k := 1 .. n **do**

    if (received ⟨r_i, Δ, j⟩ t.c. Δ[k] ≠⊥ **and** V_i[k]=⊥)

      then V_i[k] := Δ_i [k]:= Δ[k];

Algoritmo usando S\pseudo-codice(2)

send(⟨n, V_i, i⟩) to all;

wait until (**for all** j : recv(⟨r_i, V_j, j⟩) or $j \in FD_i$);

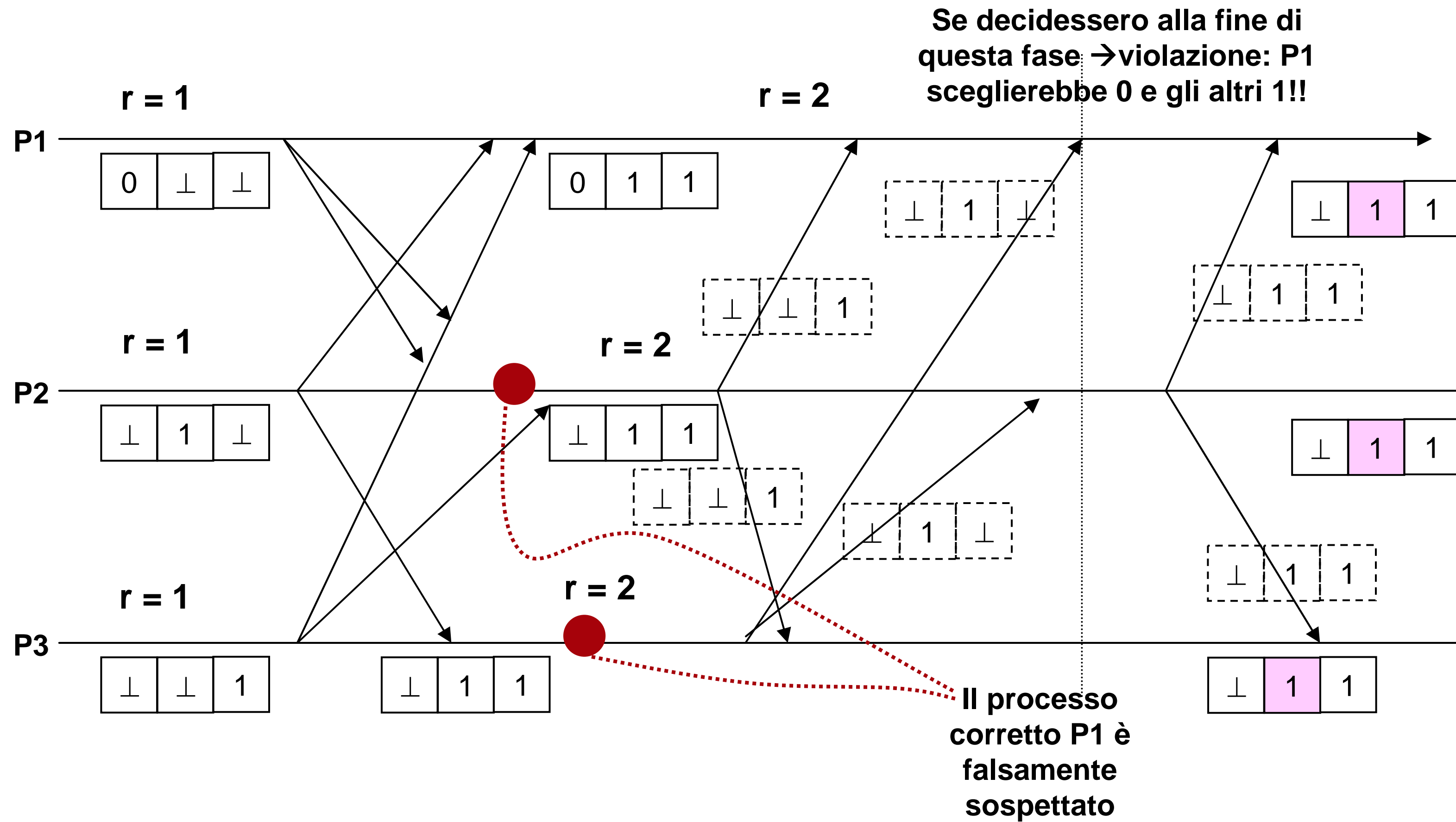
for k := 1 .. n **do**

if (received ⟨n, V, j⟩ t.c. $V[k]=\perp$)

then $V_i[k] := V[k]$;

$decision_i := $ prima entry $V_i[j] \neq \perp$

Algoritmo usando S\esempio



Per le caratteristiche del FD S:

- **Il processo corretto P1 viene falsamente sospettato da P2 e da P3**
- **Il processo corretto P2 non viene mai sospettato**

Algoritmo usando ◊S

Rivediamo dapprima le proprietà del FD di tipo ◊S:

classe ◊S:

• **Eventual Weak Accuracy**: *Esiste un tempo dopo il quale* **qualche** processo corretto non viene più sospettato da un altro processo corretto.

• **Strong Completeness**: **Ogni** processo guasto è alla fine sospettato permanentemente da **ogni** processo corretto.

Algoritmo con FD ◊S \implementazione

- Ogni processo ha accesso ad un modulo D_i di FD di classe ◊S
- Tollera al più t processi guasti, dove **n > 2t** (maggioranza di processi corretti)
- Usa il paradigma di *rotating coordinator*, ogni processo sa che durante il round *r* il coordinatore è *c=(r mod n)*+1. Il coordinatore p_c prova a decidere il valore. Se è corretto e non è sospettato avrà successo e manda in *Reliable-broadcast* il valore deciso
- Ogni round è diviso in 4 fasi

Algoritmo con FD ◊S \implementazione

Fase 1: ogni processo invia la stima corrente del valore deciso (etichettato con un timestamp pari al valore dell'ultimo round in cui il valore è stato aggiornato) al processo coordinatore *c*

Fase 2: il processo coordinatore *c* raccoglie una maggioranza di tali valori stimati, seleziona quello con il timestamp più alto, e lo invia a tutti i processi come nuova stima

Fase 3: per ogni processo corretto 2 possibilità:

- *p* riceve il valore stimato da *c* e invia un *ack* a *c* per indicare che ha adottato questo valore come nuova stima; oppure
- *p* sospetta *c*, quindi invia un *nack* a *c*

Fase 4: *c* raccoglie sia *ack* che *nack.* Se riceve una maggioranza di *ack:* il valore è *locked*. *c* lo manda in Reliable-broadcast. Il valore è deciso!

Algoritmo con FD ◊S \pseudo-code

```
upon propose(v):
  r ← 0          // current round
  τ ← 0          // last round in which v was updated
  while not decided do
    c ← (r mod n) + 1
    send message (vote, r, v, τ) to Pc
    if i = c then
      wait for messages (vote, r, v′, τ′) from ⌈(n+1)/2⌉ servers
      t ← largest τ′ received in vote messages
      v ← some v′ received in a vote message with τ′ = t
      send message (propose, r, v) to all
    wait for a message (propose, r, v′) from P_c or c ∈ D_i
    if a (propose, . . . ) message was received then
      v ← v′; τ ← r
      send message ack to P_c
    else
      send message nack to P_c
    if i = c then
      wait for ack or nack messages from ⌈(n+1)/2⌉ servers
      if all are ack messages then
        send message (decide, v) to all
    r ← r + 1
upon receiving a message (decide, v′):
  if not decided then
    send the message (decide, v′) to all
    decide(v′)
```

fase 1

fase 2: v è il valore stimato

fase 3

fase 4: si noti che i processi implementano un reliable broadcast

Run “Well-Behaved”

$(1, v_1)$

stima = v_1

(ack_1)

v_1 è locked

decide v_1

1 2 . . . n

(nuova stima= $v1$, nuovo timestamp=1)

Algoritmo con FD ◊S e liveness

- Si noti che se in ogni fase 3 una maggioranza di processi sospetta falsamente il coordinatore, l'algoritmo ricomincerebbe sempre da capo…(non si entrerebbe mai nell'epoca in cui il valore è locked)

- Perché riesce a rispettare la proprietà di liveness? Dipende dalla proprietà di eventual weak accuracy del FD: esiste un tempo t dopo il quale qualche processo corretto (alla fine un coordinatore sarà tra questi) non sarà più sospettato

- In realtà la rete alterna periodi di stabilità a periodi di instabilità. Nei periodi di stabilità il FD non genera falsi sospetti, quindi il coordinatore corretto non viene sospettato e l'algoritmo TERMINA (il valore viene deciso)