

Leader Election

- Risolve il problema di nominare un processo che è un unico rappresentante riconosciuto da tutti gli altri processi
- Spesso c'è bisogno di un processo leader perchè deve svolgere un ruolo particolare all'interno della computazione. Es:
 - Coordinatore nella mutua esclusione risolta con algoritmo centralizzato. Nota che l'elezione viene chiamata ogni volta che il coordinatore corrente vuole lasciare il suo ruolo
 - Primary nello schema di replicazione primary-backup
- Vedremo due classi di algoritmi:
 - Algoritmi che funzionano assumendo NO GUASTI (utile nel caso del coordinatore nell'algoritmo di mutua esclusione)
 - Algoritmi tolleranti ai guasti (utili nel caso che l'elezione avvenga nel caso di guasto dell'attuale leader, es. primary-backup)

Leader Election

- ➡ Qualsiasi processo può **chiamare** un'elezione (per esempio un processo chiama l'elezione quando scopre che il coordinatore si è guastato)
- ➡ Un processo può chiamare **al più un'elezione alla volta**
- ➡ Potrebbero esserci **chiamate concorrenti** per la stessa elezione
- ➡ Il risultato di un'elezione non dipende dal processo che l'ha chiamata
- ➡ Processi coinvolti in un'elezione sono chiamati i "**participants**" all'elezione
- ➡ Tra i processi non guasti deve essere eletto il **migliore**. Eleggere il migliore significa scegliere un parametro sulla base del quale confrontare i diversi processi. Non esistono due processi con un valore del parametro dello stesso valore. Ciò significa che i processi devono essere ordinati secondo un ordine totale.

Leader Election\specifica

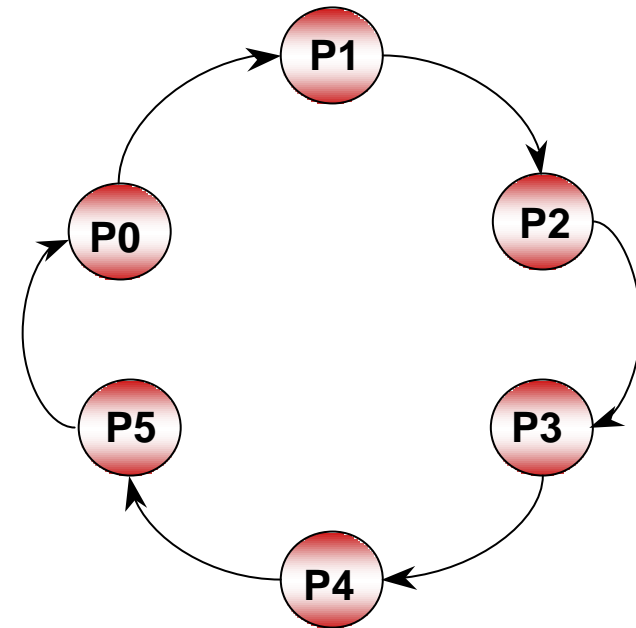
- ➡ Processi hanno una conoscenza a-priori di una funzione O , $O(p_i) = \{p_j, \dots, p_k\}$, che rappresenta un total order tra i processi (*a royal hierarchy*)
- ➡ Qualsiasi run di un algoritmo che implementa la Leader Election deve garantire
 - ➡ **Safety**: If process p_i is leader at time t , then every process in $O(p_i)$ has crashed by t
 - ➡ **Liveness** : If some process is correct, then at any time, some process is eventually leader

Algoritmo di Chang-Roberts

- Si assume che ogni processo abbia un identificatore unico
- L'algoritmo assicura che il processo con il massimo identificatore diventa leader, i.e. in questo caso la royal hierarchy è costituita dagli identificativi dei processi
- Alla fine dell'algoritmo tutti i processi concordano su un processo eletto
- Ogni processo può essere partecipante o no all'elezione
- La safety può essere riformulata in questo modo:
 - **Safety:** \forall processo p_i se p_i viene eletto leader in un certo istante di tempo allora p_i è il processo con il più alto identificativo
 - Poichè gli attributi sono ordinati secondo un ordine totale...IL PROCESSO E' UNICO!

Algoritmo di Chang-Roberts \modello

- Sistema asincrono
- No guasti: processi corretti e canali affidabili (nessun messaggio perso, cambiato, spurio);
 - no failure happens during the run of the election algorithm
- Topologia della rete: ad anello *logico* con canali unidirezionali. Il processo P_i è collegato con P_{i-1} e con $P_{i+1} \pmod n$
 - p_i has a communication channel to $p_{(i+1) \bmod N}$.
 - All messages are sent clockwise around the ring



Algoritmo di Chang-Roberts \implementazione

- Ogni processo P_i mantiene due interi $myid$ e $leader$, un booleano $candidate$ inizializzato a false
- *Chiamata di elezione:*
 - invia un msg di "election" con allegato il suo identificativo $myid$
 - $candidate=true$
- *Ricezione di "election":*
 - Se l'id ricevuto è maggiore di $myid$ allora inoltra il msg ricevuto
 - Se l'id ricevuto è uguale a $myid$ allora invia un msg "leader" con allegato il suo id $myid$ (se ciò avviene significa che è esso stesso il processo con l'identificativo più alto)
 - Se l'id ricevuto è minore di $myid$ e non è $candidate$ (non ha ancora inoltrato nessun messaggio "election"), invia un msg di "election" con allegato il suo identificativo $myid$
 - $candidate=true$
- *Ricezione di "leader":*
 - $leader=$ identificativo allegato al msg ricevuto
 - se l'id allegato al msg ricevuto è diverso da $myid$, inoltra il msg ricevuto

Chang Roberts/pseudo-code

boolean candidate:=**false**

integer leader:=**null**

integer myid

when startelection

trigger(send("election", myid)) *% il processo inizia l'elezione candidandosi*
candidate := **true**

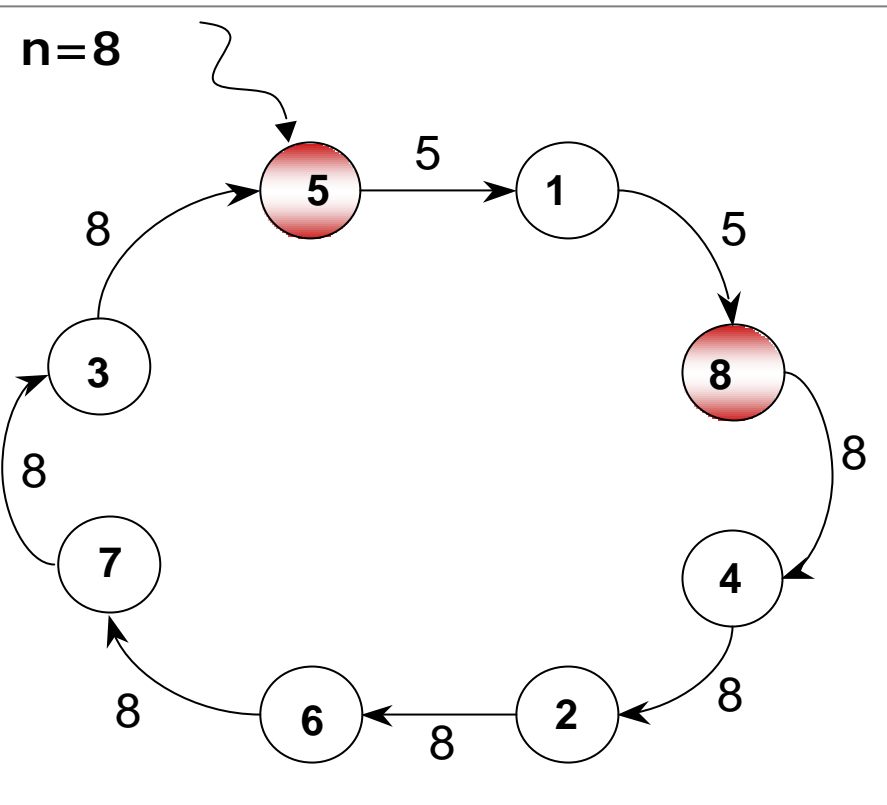
when receive("election", nid)

if nid>myid **then trigger**(send("election", nid)) **else** *% inoltra del msg*
 if nid=myid **then trigger**(send("leader", myid)) **else** *% invio id del leader*
 if (nid<myid and not(candidate) **then trigger**(send("election", myid))
candidate:=**true** *% il processo si candida solo se non ha*
 ancora mai visto un id maggiore del suo

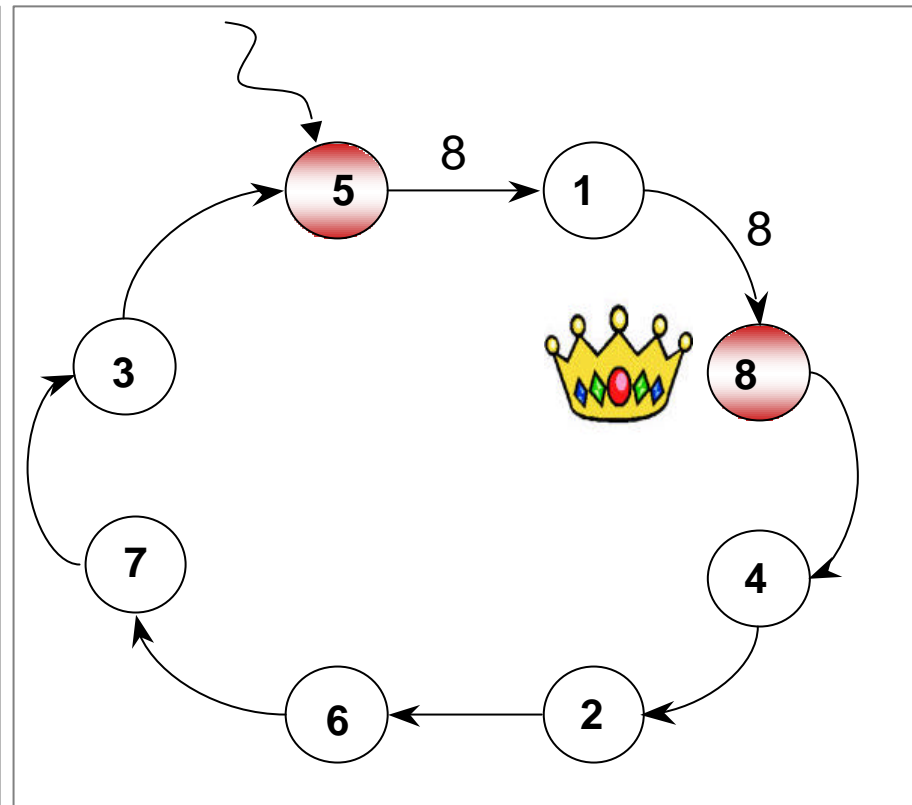
when receive("leader", nid)

trigger(leader:=nid)
 if (nid ? myid) **then trigger**(send("leader", nid)) *% inoltra del msg*

Chang-Roberts/scenario



numero di messaggi "election" = n



+ 2

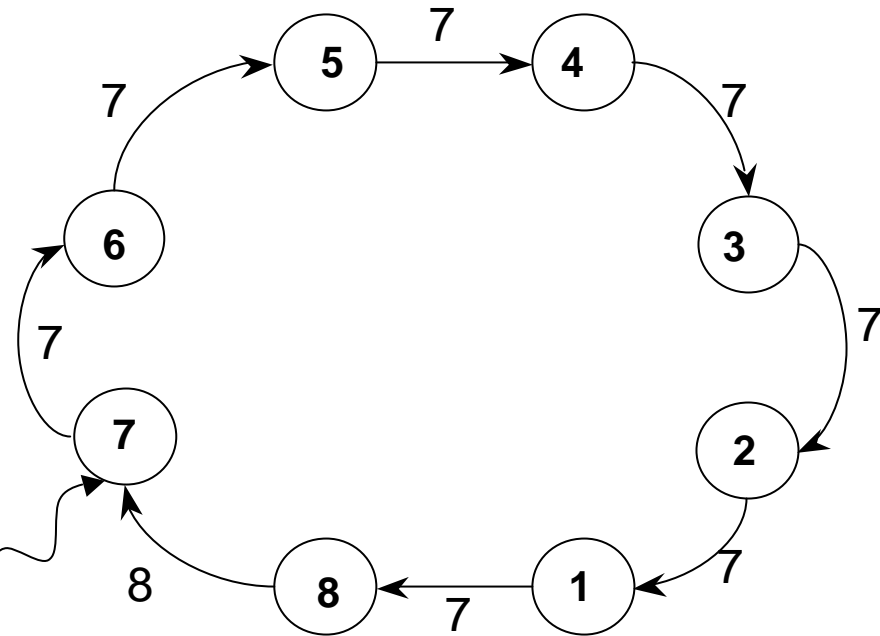
I 2 messaggi rappresentano la distanza che separa il processo che inizia l'elezione da quello che ha un identificativo maggiore

Qual'è il caso peggiore?...

E il migliore?

Chang-Roberts/complessità

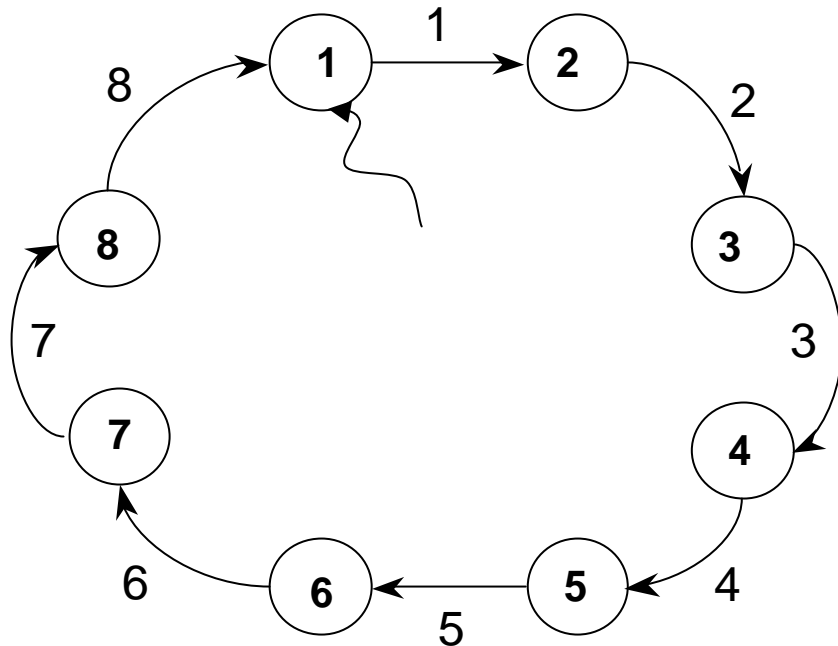
worst case: identificativi
decescenti in senso orario



Se inizia 7 → $7+n$
 Se inizia 6 → $6+n$
 Se inizia 5 → $5+n$
 ...

Se iniziano tutti
contemporaneamente
l'elezione ho $O(n^2)$

best case: identificativi
crescenti in senso orario



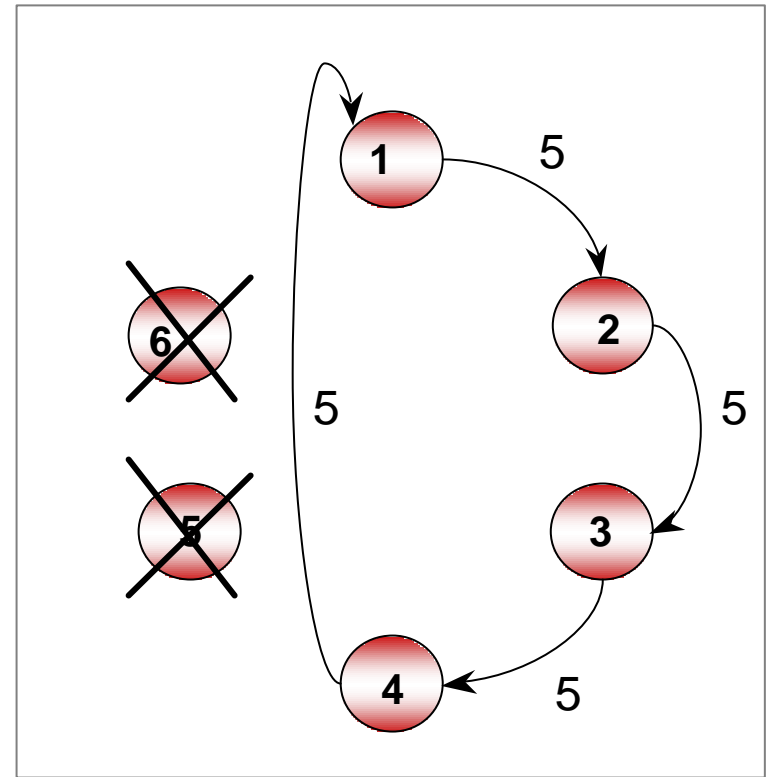
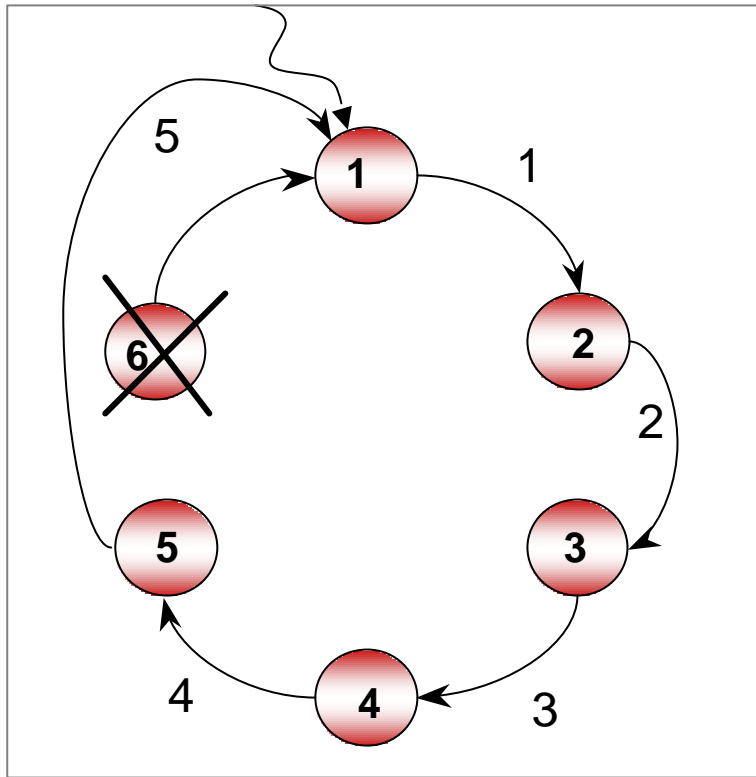
Se inizia 7 → $(n-7)+n$
 Se inizia 6 → $(n-6)+n$
 Se inizia 5 → $(n-5)+n$
 ...

Se iniziano tutti
contemporaneamente
l'elezione ho $O(n)$

Chang-Roberts/correttezza

- Safety: suppongo per contraddizione che venga eletto un leader con $myid <$ del più grande identificativo $maxid$
 - sia p il processo con identificativo $myid$. p non ha mai ricevuto $maxid$. Poiché $leader := myid$ implica che ad un certo punto p ha ricevuto $nid = myid$. nid non ha mai incontrato un processo con $id > myid$. Poiché ha contattato tutti i processi appartenenti all'anello implica che $myid \geq$ di tutti gli id presenti nell'anello. Dato l'ordine totale degli id $myid = maxid$. Contraddizione
- Nota che l'ordine totale implica che $maxid$ è unico
- Liveness: per esercizio

Chang-Roberts/caso di guasto durante l'elezione



➡ *Quale proprietà viene violata? La proprietà di Liveness*

Algoritmo "Bully"

- ➡ E' un algoritmo che tollera guasti durante l'elezione
- ➡ Assunzioni:
 - ➡ i processi hanno identificativi totalmente ordinati
 - ➡ ogni processo conosce l'identificativo di tutti gli altri (al contrario di Chang-Roberts)
- ➡ L'algoritmo elegge il processo con il massimo identificatore

Algoritmo "Bully" /modello

➡ Sistema **sincrono**

➡ Guasti di tipo crash con numero di guast f pari a $f=n-1$

➡ canali affidabili

➡ Tutti i messaggi arrivano entro un tempo T_{trans}

➡ Viene fatto il dispatch di una risposta entro un tempo $T_{process}$ dalla ricezione di un messaggio

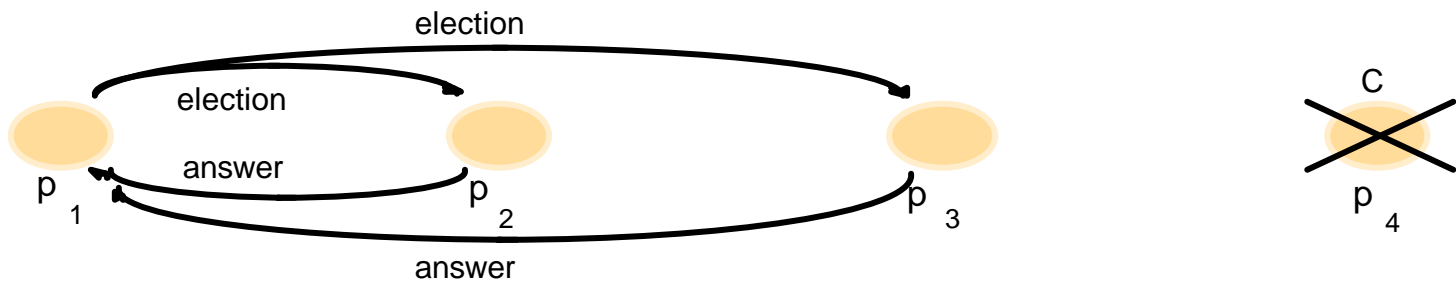
➡ Se nessuna risposta è ricevuta in $2T_{trans} + T_{process}$, il nodo è assunto guasto

Algoritmo "Bully" /implementazione

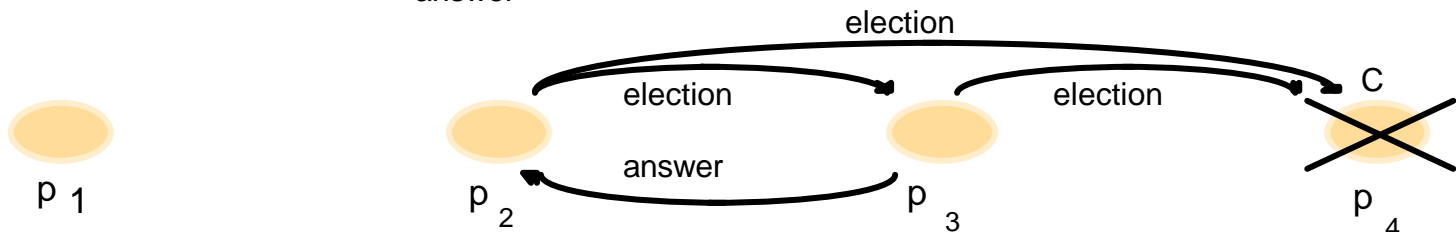
- Se un processo sa di avere l'identificatore più alto, allora elegge se stesso e manda un messaggio "coordinatore" a tutti i processi con identificativo più basso
- Un processo che fa iniziare un elezione, manda un messaggio "election" a tutti i nodi con identificativo maggiore del suo
 - se non ottiene risposte informa i processi con identificativo più basso del suo che è il nuovo coordinatore
 - se riceve qualche risposta significa che esiste un processo attivo con identificatore più alto. Quindi il processo aspetta il messaggio coordinator da un altro processo entro un tempo T' (T' dipende dal codice e dalla velocità del processore. Va accuratamente determinato per ogni sistema)
- Un processo che riceve un messaggio "election" risponde "OK" e inizia un'altra elezione (nel caso non l'avesse ancora fatto)

Algoritmo "Bully" /scenari

Stage 1
p1 rileva il guasto di p4



Stage 2



Stage 3
p3 si guasta prima di mandare "coordinatore"



Eventually.....

Stage 4



Elezione del coordinatore p₂, dopo il guasto di p₄ e p₃

Algoritmo "Bully" /pseudo-code

identifier myid

processlist P := [id1, ...idn]

higherlist L := [id1, ...idk] s.t. $idi > myid$

timeout T

timeout T'

identifier leader := null

boolean candidate := false

|| **when** startelection

startelection := false

candidate := true

trigger(send("election") to all pi s.t. $idi \in L$)

set T := $2T_{trans} + T_{process}$

| **when** receive("election") from pj

trigger(send("OK") to pj) **and if not**(candidate) **then** startelection := true

| **when** receive ("OK") from pi

received := received U idi

Algoritmo "Bully" /pseudo-code

```
|| when T expires
    for all idi not in RECEIVED then
        received:=null;
        P:=P-{idi}
        L:=L-{idi}

        if (L?null) set T'

|| when L=null then trigger(send(coordinator) ) to P

|| when receive("coordinator") from pi
    trigger(leader:=idi)

|| when (T' expires and leader=null)
    startelection:=true
```

Algoritmo "Bully" /performance

- **Best case:** il processo con il secondo identificativo più alto si accorge del guasto del coordinatore ed elegge se stesso. Vengono inviati $N-2$ **messaggi coordinator**. il nuovo coordinatore è scelto in un tempo pari al tempo di trasmissione di un messaggio
- **Worst case?**
il processo con l' identificativo più basso si accorge del guasto del coordinatore. $N-1$ processi iniziano l'elezione mandando msg ai processi con gli identificativi più alti. L'overhead dovuto ai msg è $O(N^2)$. Il nuovo coordinatore è scelto in un tempo approssimativamente pari al tempo di trasmissione pari di 5 messaggi

Algoritmo "Bully" /correttezza

- ➡ **Safety**(sketch): suppongo per contraddizione che nel sistema un processo con un certo identificatore id si elegga coordinatore e che esista un processo vivo con identificatore più alto che si elegge coordinatore. In tal caso il processo con identificatore id si accorgerà della presenza dell'altro.
- ➡ **Liveness**(sketch): segue da assunzione di canali affidabili

Algoritmo "Bully" /correttezza

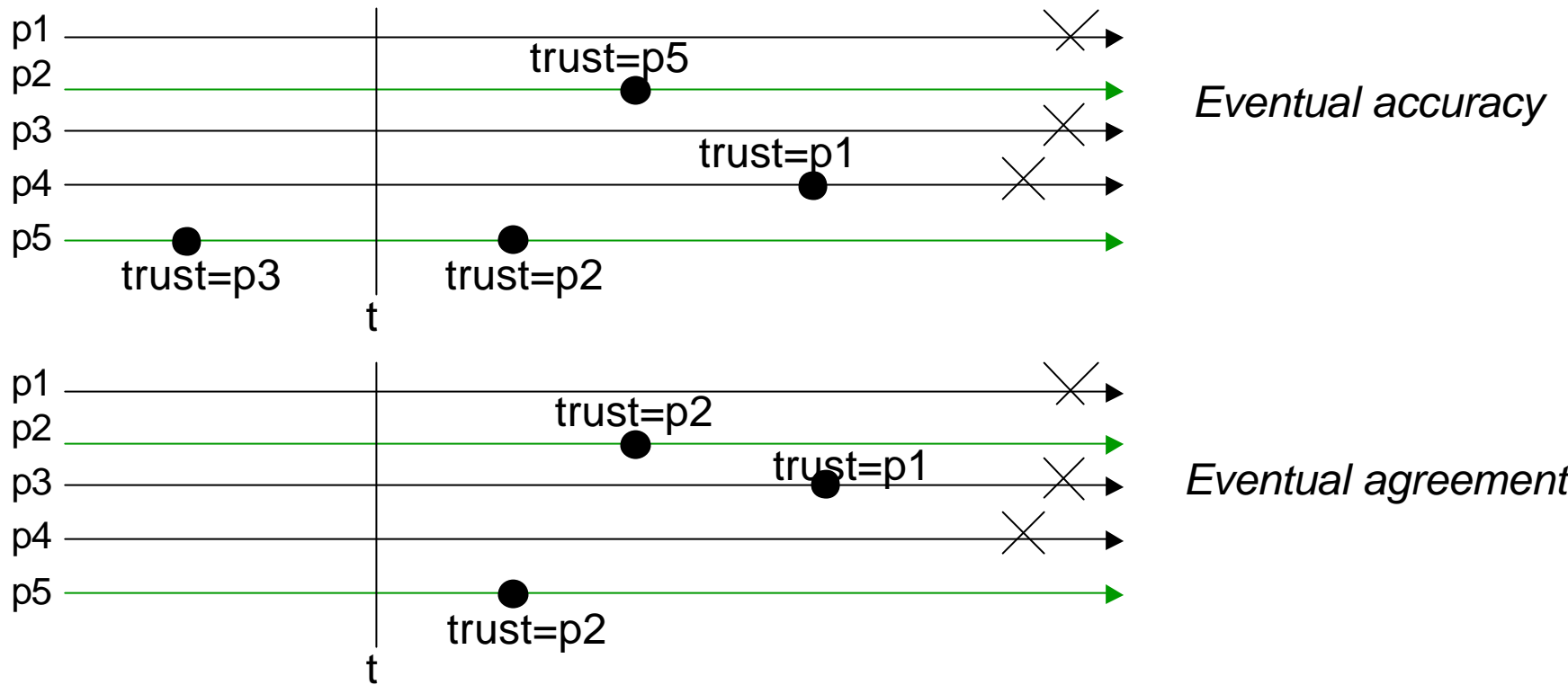
- ▶ Quale failure detector può essere usato per implementare il bully algorithm?
 - ▶ esercizio. Scrivere l'algoritmo bully non facendo uso dei timeout in modo esplicito ma affidandosi al failure detector di cui sopra
- ▶ Indebolendo quali assunzioni viene compromessa la **Safety**?
 - ▶ in un modello crash recovery
 - ▶ in un modello non sincrono

Eventually Leader Election

- La specifica di safety impone l'uso di un modello di sincronia perfetta perchè richiede che se un processo è eletto leader in un determinato istante di tempo allora tutti i processi che lo precedono nella royal hierarchy hanno già fatto crash. Tuttavia se esistono falsi sospetti il processo può dedurre erroneamente che un processo (corretto o meno) che lo precede nella gerarchia ha già fatto crash (nel caso che non sia corretto vuol dire che farà crash più avanti nel tempo)
- L'idea è quindi quella di indebolire la specifica per poter utilizzare un modello di sistema più debole: Eventually Leader Election
 - Il processo *leader* è *trusted* dagli altri processi
 - L' eventually leader election può essere incapsulata in un servizio che permette di capire quale processo è *trusted* dagli altri processi e quindi di capire chi è il leader. Il servizio ritorna l'identificativo del processo *trusted*

Eventually Leader Election/specifica

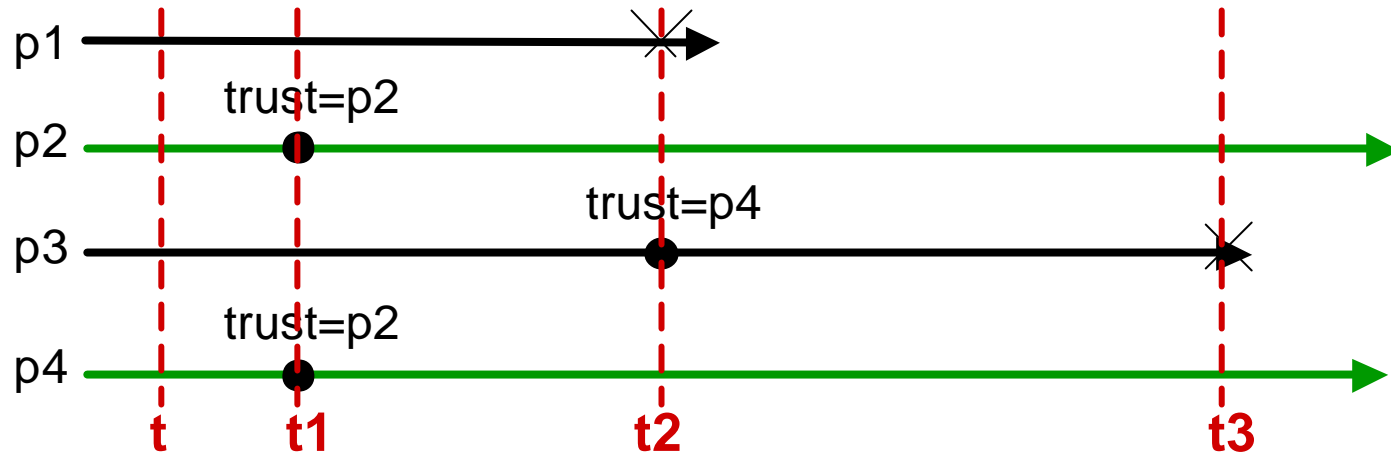
- *Eventual accuracy*: esiste un tempo t dopo il quale tutti i processi corretti hanno un *trust* su processi corretti
- *Eventual agreement*: esiste un tempo t dopo il quale se un processo p è trusted da un processo corretto, allora per ogni processo q trusted da un processo corretto vale $p=q$



Eventually Leader Election e Leader Election

Eventually Leader Election assicura qualcosa di molto meno forte rispetto alla specifica di Leader Election: assicura che prima o poi i processi corretti eleggeranno lo stesso processo corretto come leader e questo processo rimarrà leader per tutta la durata della computazione (deriva dal fatto che non si guasterà mai). *Tuttavia:*

- il processo eletto potrebbe non essere il primo processo nella royal hierarchy in un certo istante
- non tutti i processi sono sicuramente concordi sul medesimo leader in un certo istante di tempo



Dall'istante t3 in poi l'eventually leader election corrisponde alla Leader Election. Nota che prima di t due processi corretti potrebbero considerare leader due differenti processi. Dopo t (per tutti gli istanti di tempo t il sistema si stabilizza) i processi corretti raggiungono un accordo su un singolo leader ma un processo che farà crash potrebbe non condividere la scelta (t2). Inoltre il processo considerato leader da tutti i processi corretti potrebbe non essere in un certo intervallo di tempo il più alto nella gerarchia (vedi l'intervallo [t1 t2])

Eventually Leader Election\modello

- modello parzialmente sincrono:
 - Tanto sincrono da implementare quale classe di failure detector? à P
- Guasti dei processi: assunti guasti di tipo crash. Un process che fa crash può fare recovery:
 - Nota che in assenza di recovery l'eventually leader election può essere implementata direttamente facendo uso di del failure detector di cui sopra
- Assunzione: almeno un processo è corretto, i.e.
 - non fa mai crash (always up)
 - fa crash ma prima o poi fa recovery e non fa più crash (eventually up)
- Canali: FairLossPoinToPointLinks
 - FairLoss, Finite Duplication, NoCreation

Eventually Leader Election\implementazione

- Ogni processo mantiene una lista di potenziali leader in una variabile *possible*. Inizialmente la variabile contiene tutti i processi
- Ogni processo tiene traccia di quante volte ha fatto crash e recovery con una variabile *epoch*
- Ogni processo periodicamente manda a tutti un msg di *heartbeat* insieme al suo epoch number (valore corrente di *epoch*):
 - i processi che non rispondono in tempo (secondo un certo timeout) vengono esclusi dalla lista *possible*
 - Un processo che risponde in tempo ma non è in *possible* (perchè aveva fatto crash ed ora ha fatto recovery, oppure era diventato lento) viene aggiunto di nuovo in *possible*
- Inizialmente il leader per ogni processo è p1, dopo ogni timeout p1 controlla se è ancora il leader. Il controllo viene fatto utilizzando una funzione *select* che prende come argomento *possible* e seleziona un processo tra i processi presenti in *possible*. Il processo ritornato è, tra quelli che hanno il minimo numero di epoca, quello con identificativo più basso.
 - Questo permette di selezionare un processo eventually up
- Un processo incrementa il timeout ogni volta che la *select* torna un processo diverso dal leader corrente
 - se la *select* torna un leader diverso non perchè il corrente sia guasto ma solo perchè il ritardo sul canale di comunicazione è maggiore rispetto al timeout, ciò permette di correggere il timeout troppo corto e di far tornare il processo escluso ingiustamente nella lista dei *Possible*

Eventually Leader Election \pseudo-code

|| **when** startelection

leader:=p1
possible:= Π
epoch:=0
T:=timeout

|| **when** receive(heartbeat,epc) **from** pi

possible:= possibleU {pi,epc}

|| **when** recovery

retrieve(epoch)
epoch++
store(epoch)

|| **when** T expires

if leader? **select**(possible) **then**
 set T to $T+\Delta$
 leader:=**select**(possible)
 trigger(trust, leader)

possible:=null

while(true) **do**
 trigger(send(heartbeat, epoch))

} nota che questo implementa
uno stubborn link

Eventually Leader Election\correttezza

eventual accuracy: supponi per contraddizione che un processo corretto p_i ha un trust permanente su un processo non corretto p_j dopo un certo istante di tempo t . Due casi:

➔ (1) **p_j prima o poi fa crash e non fa mai più recovery:**

Poichè p_j fa crash e non fa mai recovery allora p_j invierà i suoi heartbeat solo un numero finito di volte. Da finite duplication e no creation dei canali esisterà un tempo t dopo il quale p_i non consegnerà più alcun messaggio da p_j . In tal caso p_j verrà eliminato dalla list locale *Possible* e p_i cambierà il suo trust su un altro leader. Contraddizione.

➔ (2) **p_j fa crash e recovery all'infinito.**

In questo caso l'epoch number di p_j crescerà all'infinito. Per ogni processo corretto (anche per gli eventually up) invece l'epoch number prima o poi smetterà di crescere. Quindi arriverà un momento in cui p_i non avrà più il trust su p_j . Contraddizione.

eventual agreement: considera il sottoinsieme S dei processi corretti. Considera il tempo t dopo il quale:

- ➔ il sistema diventa stabile (sincrono)
- ➔ i processi in S non fanno più crash, i.e. gli epoch number dei processi in S non vengono più incrementati
- ➔ per ogni processo corretto p_i e per ogni processo non corretto p_j , p_i smette di consegnare messaggi da p_j in modo permanente oppure l'epoch number di p_j diventa più grande di quello di p_i

Dopo questo istante di tempo ogni processo trusted da un processo corretto sarà un processo appartenente a S . La funzione select assicura che sia il processo con l'id più piccolo.

Dall'ordine totale degli id implica che il processo selezionato sia lo stesso per ogni processo