DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

Architetture distribuite per il Cloud

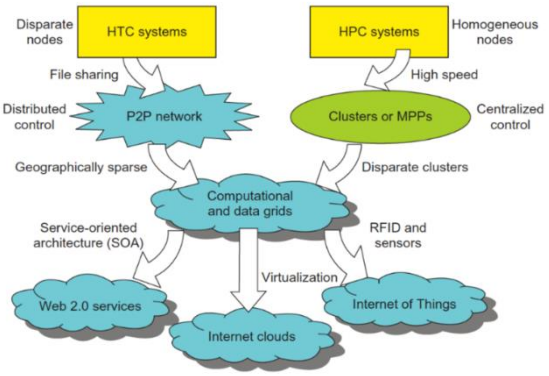
Indice:

Lezione 1, Introduzione e concetti base:

Il Cloud si basa sulla capacità di compiere **high performance computing**, ovvero calcolo con una **grande capacità di computazione**, tramite l'utilizzo della **virtualizzazione**. Il corso verte su problemi di **high performance computing**, di **comunicazione P2P** e di **virtualizzazione**.

Progettare un algoritmo parallelo è qualcosa di profondamente diverso, dal punto di vista concettuale, da progettare un algoritmo sequenziale. Concettualmente, infatti, non esiste un modello simile alla **macchina di**

Turing in grado di rappresentare la computazione parallela inglobando quelli che sono i problemi dei problemi non parallelizzabili.



Computazione parallela

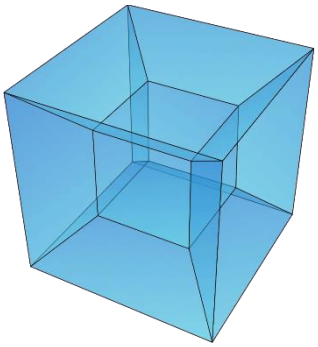
Quando si parla di **speed up**, parliamo di quanto sia più efficiente l'elaborazione fatta da una rete di **n nodi** rispetto a un **solo nodo**. Quello che non si pensa è che un **algoritmo parallelo** è completamente diverso da un **algoritmo sequenziale**, in sostanza si vanno a confrontare cose molto diverse fra di loro.

La complessità di un algoritmo sequenziale si basa sulla **macchina di Turing**, di cui non esiste una controparte per la computazione parallela. In sostanza sono cose semplicemente incomparabili perché fondamentalmente diverse.

Un altro aspetto rilevante è come i nodi **comunicano** fra di loro, esistono infatti delle limitazioni legate alla **topologia del grafo** utilizzato, che dipendono da tanti fattori (per esempio se mando informazioni a due nodi molto lontani), infatti la rete impone dei limiti inferiori molto rigidi, per questo motivo si devono prendere in considerazione diversi parametri:

- **Bisezione minima:** quanti archi devo tagliare per bisecare l'architettura (se la bisezione è 1, mi basta tagliare un solo arco, questo vuol dire che esiste un **collo di bottiglia**);
- **Diametro**: della rete di comunicazione (**log(n)** nel **ipercubo**);
- **Grado**: numero di connessioni per nodo (**log(n)** nel **ipercubo**);

Detto questo, il miglior grafo per fare computazione parallela è un **Ipercubo (**dove un nodo è connesso a **log(n)** nodi, se n sono i nodi del grafo), che offre le condizioni migliori per progettare un algoritmo che elabori in un tempo proporzionale a **log(n)**.



Quando si parla di ipercubi il fatto che il grado del grafo non è costante ma dipende dal numero di nodi crea problemi, perché se voglio scalare il numero di porte per ogni singolo nodo deve crescere in funzione di **n** e questo da problemi, per questo sono nati tutta una serie di emulatori di **ipercubi**, i più celebri sono:

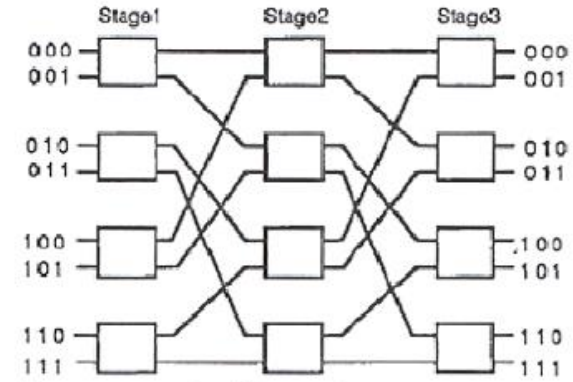
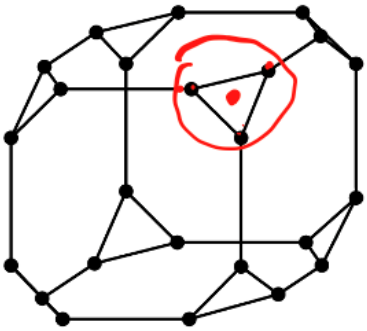
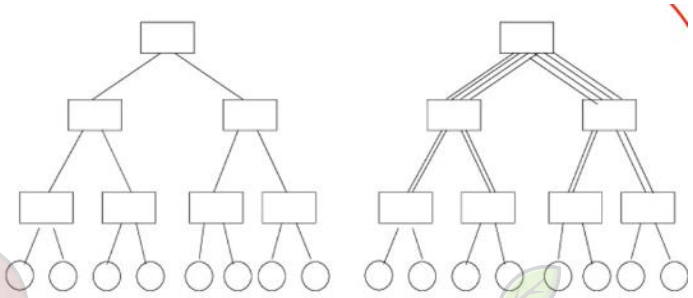


Fig. 1. 8x8 Shuffle exchange network


Cube Connected Cycles

Che sono utilizzati al posto degli ipercubi ed offrono prestazioni ottimali (Scarano regna anche su questo).

Di fatto oggi si usano quelli che sono chiamati **Fat Tree** ovvero alberi binari, dove il canale di comunicazione diventa sempre più "grosso" (raddoppia) mentre si va verso la radice e questo assicura una comunicazione **tutti a tutti** con tempi incomparabili con qualsiasi altra architettura, in sostanza c'è una rete di permutazione che permette di permutare i dati in modo molto veloce (proporzionale a **log(n)**, di fatti è al momento la rete di **routing** più utilizzata.


(a) Binary tree (b) Binary fat–tree

Sistemi Distribuiti: Message Passing vs Shared Memory

Un sistema distribuito è una **collezione di dispositivi**, con **autonome capacità di calcolo** e **capaci di comunicare tra loro**, esso fa uso di **algoritmi distribuiti** per svolgere operazioni di calcolo. I sistemi distribuiti sono incredibilmente **flessibili** ed **efficienti** ed incredibilmente **difficili da mettere a punto**, sono infatti:

- Composti da **hardware eterogeneo**;
- Sono **asincroni**;
- Hanno una **conoscenza locale limitata**: un nodo conosce i vicini ma non la topologia della rete;
- Sono composti da un **numero di nodi molto elevato**, e quindi non si può evitare di incappare in **fallimenti**, che vanno gestiti. Un primo approccio è la **ridondanza**, oppure inventarsi tecniche per avere risultati attendibili anche se i nodi non lo sono.

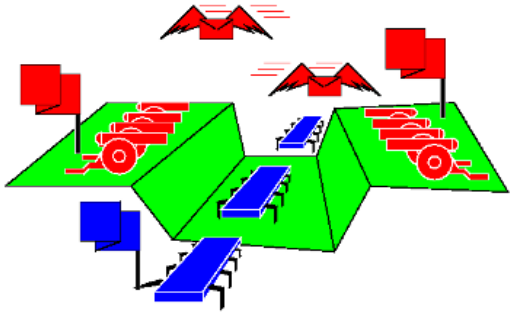
Progettare algoritmi distribuiti è veramente complicato, non esiste infatti l'equivalente della **macchina di Touring**, ne linguaggi di **programmazione** ad hoc, bisogna inoltre decidere il tipo di comunicazione (**Message Parsing** o **Shared Memory**), se è **sincrono** o **asincrono**, se ci sono **fallimenti** o meno, ma sopratutto è veramente oneroso e complesso **dimostrarne la correttezza**, in quanto non è semplice ottenere uno **stato** chiaro del sistema.

Un'altra difficoltà sta nel misurare gli **algoritmi distribuiti** (non c'è un **clock** univoco), di solito si usa il **numero totale di messaggi** spediti come **misura di complessità**. Inoltre, siccome i **tempi di computazione** sono estremamente contenuti rispetto ai **tempi di comunicazione**, quello che si va a misurare sono proprio quest'ultimi in quanto più **rilevanti** per il calcolo del **tempo complessivo**. Per esempio, se ho un anello di nodi, e per comunicare ogni messaggio fa il giro dell'anello, saranno mandati **N messaggi** (complessità di messaggio). Per il tempo invece si fa riferimento **all'arco più lento** che si assume essere **l'unità di tempo**.

Incontreremo diverse dimostrazioni di **impossibilità**, ovvero problemi che non si possono risolvere in sistemi asincroni, due esempi:

- **Non esiste un algoritmo distribuito per il problema dei due generali bizantini.**

- **Non esiste un algoritmo deterministico per l'elezione di un leader in un anello se i processori non sono dotati di nomi distinti.** (trattato in seguito)



Problema dei due generali Bizantini:

In maniera astratta il problema è quello di vincere una battaglia con dei generali che non seguono esattamente gli ordini, in altri termini:

- I **rossi** vincono se le due armate attaccano simultaneamente.
- I **blu** vincono in caso contrario.
- I due generali **rossi** devono coordinare l'attacco.
- I **messaggeri** sono l'unica comunicazione possibile.
- I **messaggeri** devono passare attraverso la valle e non sempre ci riescono.

Il problema è quello di progettare un algoritmo distribuito che assicuri l'attacco contemporaneo da parte dei due generali rossi.

Dimostrazione:

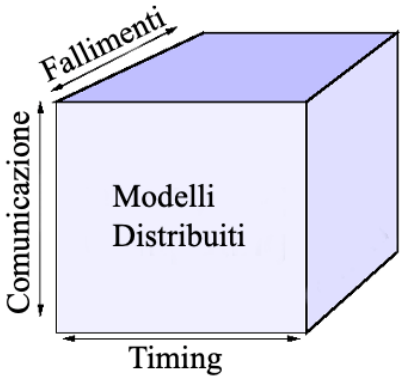
Claim: Non esiste alcun algoritmo distribuito che possa garantire che i due generali attacchino contemporaneamente.

Prova: Si assuma che tra tutti gli algoritmi per la soluzione del problema **A** sia quello che impieghi il minimo numero di messaggi, **A** deve assicurare la correttezza anche se un messaggio non viene mai recapitato (condizione del problema). Supponendo che l'ultimo messaggio non venga mai recapitato, questo implica che questo messaggio è **superfluo** e può quindi essere **eliminato**, questo genera una contraddizione in quanto **A** non è l'algoritmo che impiega il minimo numero di messaggi.

Tutto questo per dire che, laddove calcolare la correttezza di algoritmo di distribuito è complicato, dimostrarne che non esiste una soluzione è stranamente **semplice**.

Modello formale

Fondamentale avere un **modello formale** quando si parla di computazione in un sistema distribuito, infatti il modello utilizzato è spesso **ideale** e nasconde tutta una serie di complicati dettagli delle applicazioni reali. Ma è utile per cogliere aspetti fondazionali.
Il modello si basa su tre assi:



Comunicazione:

- **Shared Memory:** in cui abbiamo una memoria condivisa che viene usata dai vari nodi per comunicare.

- **Message Passing:** abbiamo **n** processori o nodi collegati tramite dei link di comunicazione **bidirezionali**. Il tutto si rappresenta tramite un automa dove ogni nodo effettua delle comunicazioni ovvero, invia e riceve messaggi.

Timing:

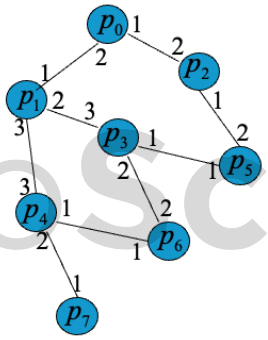
Non abbiamo un **clock condiviso**, quindi distinguiamo tra sistemi di tre tipi:
- **Sincrono:** poco realistico, ma ottimo per lo sviluppo di algoritmi, per le prove di impossibilità e limiti inferiori.
- **Semi-Sincrono:** la frequenza del clock tra dei nodi che non comunicano tra loro è ormai indicativamente la stessa. E ci si può basare su questo. Detto anche modello asincrono **temporizzato**.
- **Asincrono:** in cui è impossibile qualsiasi operazione di timing, ritardi di trasmissione non predicibili, buon modello per sistemi concorrenti..

Fallimenti:

- **Fallimento Crash**: fallimento dei singoli nodi, senza creare interferenza (solo nel momento in cui muore al massimo, ma poi si spegne).
- **Fallimento Bizantino**: il processore che fallisce si mette a lavorare contro, creando dei problemi.

Un sistema distribuito è inoltre composto da:
- **n** processori o nodi **p(0), p(1), … p(n-1);**
- **Link** di comunicazione bidirezionali;
- **∀p(i)** i link sono numerati con **1, 2, … l : l = deg(p(i));**

Pertanto l'elaborazione in un sistema distribuito può essere vista come un **automa a stati**, che parte da uno stato iniziale che cambia nel tempo in seguito a operazioni di comunicazione seguite da operazioni di computazione.



All'interno di questo automa ogni **p(i)** è una macchina a stati e:
- **Q(i)** è l'insieme degli stati (può essere un insieme infinito);
- **I(i) ⊂ Q(i)** è l'insieme degli stati iniziali;
- **δ (i)** è la funzione di transizione (elaborazione locale);

Ogni processore **p(i)** mantiene:
- Un insieme di **variabili locali;**
- Un vettore **outbuf(i)(l)**, **l = 1, …, deg(p(i))** che contiene l'insieme dei messaggi inviati, ma non ancora trasmessi.
- Un vettore **inbuf(i)(l), l = 1, …, deg(p(i))** che contiene l'insieme dei messaggi ricevuti, ma non ancora elaborati. Inizialmente vuoto.

Lo stato **p(i)** è dato dal valore delle **variabili locali** e **inbuf(i)**.

La funzione di transizione **δ**, non dipende da **outbuf(i)**, ma consuma tutti i messaggi in **inbuf(i)** produce al più **un messaggio per link** (vincolo che ci semplifica la vita).

Configurazione

Una configurazione **C** è una sequenza **q(0), q(1), …, q(n-1)** dove **q(i)** è uno stato di **p(i)**. Se **q(i) ∈ I(i), ∀(i)**, **C** è detta iniziale.

Esecuzione

L'esecuzione in un sistema **MP** (Message Passing) è modellata con eventi, più nel dettaglio un esecuzione è un sequenza infinita **C(0)Φ(1)C(1)Φ(2)C(2)Φ(3)C(3)Φ(4)…C(k)Φ(k)…** dove:

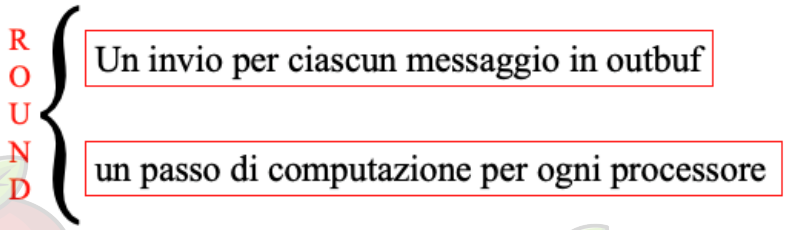
- **C(0**) è una configurazione iniziale;
- **C(k)** sono configurazioni;
- **Φ(k)** sono eventi;

Quindi in altre parole, un esecuzione è data da una **sequenza infinita di configurazione ed eventi**. Inoltre se **∃j: ∀i > j C(i) = C(i-1)**, l'esecuzione si dice terminata, ovvero non cambiano più gli stati (la configurazione non cambia).

Evento

Si indica con **ø** e dipende dal **timing**:

- **Asincrono:** ho **n** nodi, all'interno di una **configurazione** (non finale) esisterà un sottoinsieme di processori che potrà fare qualcosa (computare) (altri sono in dipendenza funzionale quindi devono attendere o semplicemente non devono computare), all'interno di una sistema asincrono io lascio che solo un processore per volta possa computare (scelto casualmente tra quelli che possono elaborare), questo per permettermi di dimostrare la correttezza (non funziona cosi durante la computazione standard) quindi l'evento è dato da: **ø = ø(i)**
- **Sincrono:** c'è un momento in cui **tutti i processori elaborano** e poi c'è un momento di **comunicazione/spedizione** del risultato elaborato, quindi tutti elaborano e tutti trasmettono. Infatti un sistema sincrono presenta tutti i vincoli dell'asincrono e l'esecuzione si basa su **round**, composti da un momento di elaborazione e uno di comunicazione. In questo caso un evento è definito come: **ø = ø(0), ø(1), … ø(n-1)**



Tipologie di eventi

Un evento può essere sia un passo di computazione **comp(i)**, oppure un invio **del(i, j, m)** (sposta m da i a j)

Per esempio data un'esecuzione **C(0)Φ(1)C(1)Φ(2)…** e un evento **ø(k) = del(i, j, m)** si hanno i seguenti effetti:

- Viene rimosso **m** da **outbuf(i)(l)** dove **l** è l'indice del link **(p(i), p(j))** per **p(i);**
- Viene aggiunto **m** ad **inbuf(j)(h)** dove **h** è l'indice del link **(p(i), p(j))** per **p(j);**
- In sostanza rimuove m da outbuf di i e lo inserisce in inbuf di j

Mentre se viene data un'esecuzione **C(0)Φ(1)C(1)Φ(2)…** e un evento **ø(k) = comp(i)** si hanno i seguenti effetti:

- Le variabili locali di **p(i)** vengono aggiornate;
- Tutti i messaggi in **inbuf(i)** vengono processati;
- Viene aggiunto al più **un messaggio** per ogni elemento di **outbuf(i)**;

Safety e Liveness

L'esecuzione di un **MPA** (Asincrono) / **MPS** (Sincrono) gode di due proprietà:

- **Safety:** ogni processore esegue un numero infinito di passi di computazione (non assumiamo che qualche processore si rompa).
- **Liveness:** ogni messaggio inviato verrà recapitato (non assumiamo rotture dei sistemi di comunicazione).

Ogni esecuzione che di queste due proprietà è detta **esecuzione ammissibile**.

Schedule

Per ogni esecuzione **C(0)Φ(1)C(1)Φ(2)C(2)Φ(3)C(3)Φ(4)…C(k)Φ(k)…** la sequenza degli eventi **Φ(1)Φ(2)Φ(3)Φ(4)…Φ(k)…** viene chiamata **schedule.**

Uno **schedule** è detto **aperto** se c'è almeno un arco sul quale non viene spedito alcun messaggio.

Complessità

Esistono due tipologie di complessità:

- **Di messaggio** (asintotico): numero totale di **messaggi spediti** (vale sia per il sincrono che per l'asincrono).
- **Di tempo:** legata a doppio filo con il concetto di **terminazione** (∃j: ∀i > j C(i) = C(i-1)), dove si definisce **passo** (istante) di terminazione il **min j: ∀i > j C(i) = C(i-1)**;
  - Per il **sincrono** è il numero **totale di round** fino alla **terminazione;**
  - Per **l'asincrono** si fa riferimento al link più lento è si assume quella come unità, pertanto la complessità è data dal **totale delle unità**;

Complessità di tempo per un **MPA** più nel dettaglio, tramite alcuni concetti:
- **Esecuzione temporizzata:** data un esecuzione **C(0)Φ(1)C(1)Φ(2)…** dove ad ogni evento è associato il tempo in cui l'evento accade. Il **tempo** è una **funzione non decrescente**, ma è strettamente **crescente** per gli **eventi** di un **singolo processore**;
- **Ritardo:** la differenza tra il tempo **dell'evento che processa il messaggio** ed il tempo **dell'evento che lo aveva spedito**.

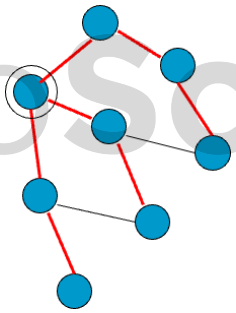
Pertanto la complessità di tempo per un **MPA** è data dal **massimo tempo di esecuzione** fino alla terminazione tra tutte le esecuzioni ammissibili dove il **ritardo** massimo è **normalizzato** ad 1 (i ritardi di messaggio sono al più 1).

Lezione 2, Algoritmi di Base:

La parte divertente del corso comincia ora :-) (a fess ra mamm).

Broadcast

Un nodo ha un messaggio e ha necessità di mandarlo a tutti, il nodo non sa nulla della rete, o meglio, sa solo di avere alcuni vicini. L'algoritmo suppone di avere uno **spanning tree radicato** e suppone di trovarsi in un sistema di tipo **MPA.**



Un nodo **p(i)** contiene le seguenti variabili locali:
- **parent(i)**: indice del nodo padre;
- **children(i)**: insieme dei figli;
- **terminated(i)**: indica se il nodo è in uno stato finale;

Inizialmente il messaggio **M** è in transito nella radice p(r) dello ST e l'algoritmo si comporta nel seguente modo:

Codice per p_r

1. Se non ricevi messaggi
2. Termina

Codice per $p_i, 0 \le i \le n-1, i \ne r$

3. Se ricevi M da $parent_i$
4. Invia M a $children_i$
5. Termina

Importante notare che si assume che l'algoritmo si avvii e basta (cosa che nella realtà non si verifica). Di norma sono necessarie degli scambi di messaggi o dei metodi per iniziare la computazione.

Tempo di elaborazione su sistema sincrono

L'elaborazione su un sistema sincrono avanza in **round**, in questo caso la fase di elaborazione non fa nulla (**delta** semplicemente rinvia il messaggio ai figli) quindi l'elaborazione finirà in un tempo proporzionale a **d** ovvero la profondità dello **ST** (mando ai vicini della radice, poi ai vicini dei vicini e cosi via).

Claim: L'algoritmo ha complessità di tempo **d** per **MPS**, dove **d** è la profondità dello **ST**.

Prova: Per induzione sulla profondità **t** dei nodi: **t = 1**: i vicini della radice ricevono **M** al passo **1**, supponiamo che i nodi a profondità **t-1** ricevano **M** al tempo **t-1**. Al tempo **t** al passo **4** dell'algoritmo tutti e soli i processori a profondità **t-1** inviano **M** a processori a profondità **t**.

Tempo di elaborazione su sistema asincrono

Abbiamo un tempo 1 (**normalizzato, che è il massimo tempo di comunicazione**) che è il tempo di comunicazione attraverso i link.

Claim: l'algoritmo ha complessità di tempo **d** per sistemi **MPA**, dove **d** è la profondità dello **ST**.

Prova: Per induzione sulla profondità **t**:
Se **t = 1**: i nodi a **profondità 1** sono figli della radice dello **ST**. Per definizione di complessità di tempo per **MPA** essi riceveranno **M** al tempo **1**.

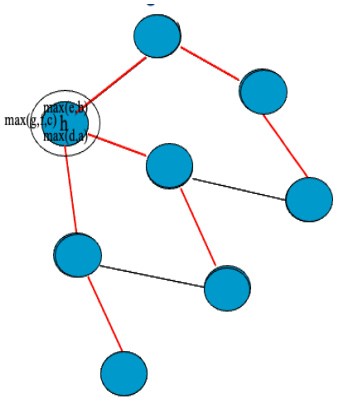
Supponendo che questo sia vero fino a profondità **t-1**.

- Sia **P(i)** un nodo a profondità **t**. Il padre di **p(i)** si trova a profondità **t-1** ed ha ricevuto **M** all'instante **t-1**.
- Ancora dalla definizione di complessità di tempo per **MPA** e per passo **4** dell'algoritmo **p(1)** riceverà **M** all'istante **t**.

Poiché l'algoritmo spedisce un messaggio per ogni arco dello **ST** si può affermare che esiste un algoritmo **MPS**/**MPA** per il **broadcast** con complessità di messaggio **n-1** e complessità di tempo **d**, assumendo sia noto uno **ST** radicato di profondità **d**.

Convergecast:

Le foglie contengono delle informazioni che devono essere raccolte nella radice, anche in questo caso andiamo a supporre l'esistenza di uno **spanning tree radicato.**
Teorema (non dimostrato):



Esiste un algoritmo **MPA** per il **convergecast** con:

- Complessità di messaggio **n-1**;
- Complessità di tempo **d**;

Assumendo che sia noto uno **ST** di profondità **d**.

Broadcast senza un ST

Invece di supporre di avere un **ST** si suppone di avere un nodo fissato **p(r)** per la radice. In questo modo possiamo usare un algoritmo di **flooding** (che parte da **p(r)**) in questo caso l'algoritmo funziona in questo modo:

1. **p(r)** invia **M** a tutti i suoi vicini e termina;
2. Quando un processore **p(i)** riceve **M** per la prima volta, tra tutti i vicini che lo hanno inviato ne sceglie uno (ad es. **p(j)**), invia **M** a tutti i vicini escluso **p(j)** e termina.

Complessità di messaggio: 2m-(n-1) dove **m** è il numero di **link** del sistema.

La prova è basata su due osservazioni:
- Un nodo non manda mai più di una volta **M** su uno stesso link
- L'arco da cui un nodo riceve **M** per la prima volta viene attraversato da **M** una volta sola.

In sostanza io nodo, ricevo il messaggio e lo riamando a tutti i miei **link** tranne quello da cui l'ho ricevuto, quindi ogni **link** viene attraversato tendenzialmente **2 volte**, quello che manca sono gli **n -1 archi** che componevano lo **ST** (ovvero i **link** di arrivo a cui non ho rimandato il messaggio).

Modifica al flooding per generare uno ST

E' possibile leggermente modifica il **flooding** per costruire un **ST**
1. **p(r)** invia **M** a tutti i vicini.
2. Quando un processore **p(i)** riceve **M** per la prima volta, tra tutti i vicini che lo hanno inviato ne sceglie uno (ad ed. **P(j)**), ed invia a quest'ultimo il messaggio **<parent>**, inoltra **M** a tutti gli altri.
3. Assegna alla variabile **parent(i)** il valore **j** e risponde **<reject>** a tutti i successivi messaggi **M**.
4. Se riceve un messaggio **<parent>** da **p(l)** aggiunge **l** all'insieme **children(i)**.

Complessità di messaggio: sono stati aggiunti due nuovi messaggi rispetto al flooding, ogni messaggio **M** su un arco produce un messaggio **<reject>** oppure un messaggio **<parent>** la complessità di messaggio è quindi O**(m)**

Algoritmo:

```
// Inizialmente parent=null, children e other sono vuoti

// Nodo padre
Se (non ricevi messaggi):
    Se (i=r && parent=nil):
        Invia M a tutti i vicini
        parent=i

// Tutti gli altri
Se (ricevi M dal vicino):
    Se (parent=nil):
        // Setto che ho trovato un padre
        parent=j
        // Dico al mio padre che sono suo figlio
        Invia <parent> a p(j)
        Invia M a tutti i vicini eccetto p(j)
else
    // Ho già un padre
    invia <reject> a p(j)

Se (ricevi <parent> dal vicino p(j)):
    Aggiungi j a children
    Se (children U other contiene tutti i vicini tranne parent):
        // Vuol dire che ho finito
        termina
Se (ricevi <reject> dal vicino):
    Aggiungi j ad other
    Se (children U other contiene tutti i vicini tranne parent):
        // Vuol dire che ho finito
        termina
```


Proprietà interessanti:
- Una volta stabilità la relazione padre figlio essa non può essere modificata;

- L’insieme dei figli non decresce;
- Ad un certo punto il grafo indotto dalla relazione padre/figlio non cambia più;
- Il valore assurdo dalle variabili è consistente;

Il grafo indotto da questo algoritmo dalle relazioni padre/figlio è **connesso** ed **aciclico**.

Lui a lezione ha zombato ma le dimostrazioni stanno: Lezione 2, Slide 25-29.

Algoritmo per la costruzione di un albero DFS

L’algoritmo assume di avere un nodo distinto che farà da radice. Ogni nodo avrà una variabile **unexplored** dove vengono mantenuti i vicini ancora da visitare. I vicini **unexplored** vengono visitati uno per volta. Il tutto è molto simile all’algoritmo per il **flooding** la differenza sostanziale è la complessità di tempo, che passa da **O(n)** nel caso peggiore a **O(m)** dove **n** è il numero di nodi e **m** il numero di archi.

Questo è uno dei classici problemi **non parallelizzabili**.

Algoritmo per la costruzione di un albero DFS senza nodo distinto

Supponiamo di voler costruire uno spanning tree DFS **senza partire da un nodo distinto**. Per fare questo è strettamente necessario che la rete sia **identificata**, ovvero che ogni nodo abbia un identificatore unico della rete.

Non avendo un nodo distinto la computazione può essere iniziata da qualsiasi nodo (eventualmente tutti), all’interno del messaggio è contenuto l’identificatore. Il risultato è lo **ST DFS** la cui radice ha l’**id** più alto **tra i partecipanti.**

Algoritmo: inizialmente per ogni nodo:
- **parent** è **nil**;
- **leader =** 0: id del nodo radice, in pratica l’identificativo dell’albero.
- **children** è vuoto;
- **unexplored** contiene tutti i vicini **p(i)** id contiene l’identificatore del nodo;

```
// Mi sono svegliato spontaneamente non alla ricezione
// di un messaggio
if (non ricevi messaggi):
    if (parent=nil):
        // Mi propongo come radice di un frammento
        leader=id;
        // Parenti punta a me stesso perchè sono la radice
        parent=i
        // Per ogni vicino (credo)
        Sia p(j) un processore in unexplored
            rimuovi p(j) da unexplored
            inva <leader> a p(j)

if (ricevi <new_id> da p(j)):
    if (leader <new_id>):
        leader=new_id;
        parent=j
        // Butto tutto quello che ho fatto
        unexplored="tutti i vicini tranne p(j)"
        if (unexplored≠vuoto):
        // Continuo o inizio la visita DFS
        Sia p(j) un processore in unexplored
            rimuovi p(j) da unexplored
            inva <leader> a p(j)
        elif
            invia <parent> a parent
elif if (leader = new_id):
    invia <already> a p(j)

if (ricevi <parent> o <already> da p(j)):
    if (ricevi <parent>):
        aggiungi j a children
    if (unexplored vuoto):
        if (parent≠i):
            invia <parent> a parent
        else
            termina con radice
    else
        // Continua visita DFS
        Sia p(j) un processore in unexplored
        rimuovi p(j) da unexplored
        inva <leader> a p(j)
```

In questo algoritmo bisogna distinguere due situazioni: il caso in cui il nodo ha cominciato ad elaborare perché ha ricevuto un messaggio oppure perché si è svegliato spontaneamente (vuol dire che in inbuf non c'è niente, quindi avrò parent a nil, aggiungo la mia id come leader cosi da iniziare a costruire un albero con il mio id, setto parent a me stesso cosi da distinguere la radice, in quanto la radice si distingue dal fatto che parent punta a se stesso). Una volta fatto ciò, invio a pj il messaggio contenente la mia id, cosi da dirgli l'albero nel quale deve entrare.
Successivamente se ricevo qualcosa da pj ci sono due possibilità:

1. Se la variabile locale leader è minore di quella ricevuta allora devo cambiare, perché io appartengo ad un albero che non sarà mai completo in quanto l'albero completo sarà quello che avrà id massima, cambiare significa che il mio nuovo leader è new id e il nuovo parent è j e metto tutto in unexplored. Se unexplored non è vuoto ricomincia la fase precedente mostrata nella foto precedente, se è vuoto mando parent a parent
2. altrimenti mi chiedo se è uguale gli comunico che stanno nello stesso albero altrimenti non faccio nulla, significa che è partita una richiesta che è arrivata ad un nodo che ha un id più grande (quindi sono nell'albero che potrebbe diventare l'albero finale, chi mi ha mandato il messaggio non lo sarà mai quindi blocca quella computazione perché non riceve risposta e ad un certo punto il nodo verrà contattato e sarà assorbito)

La particolarità di questo algoritmo sta nel fatto che ci sono dei casi in cui le elaborazioni parziali fatte vanno buttate e si deve ricominciare (se contatto un nodo che già fa parte dello ST) , per questo la visita **DFS** non è **parallelizzabile**, perché se parto in parallelo a fare la visita non posso "**mischiare**" le soluzioni e questo non mi permette di parallelizzare.

Complessità di messaggio: risulta **O(nm)** in quanto sono **O(n)** costruzioni distinte.

Complessità di tempo: risulta **O(m)** in quanto attraversiamo tutti gli archi.

Spanning Tree

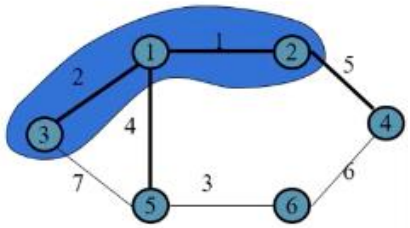
Algoritmo distribuito per la determinazione del **Minimum-weight Spanning Tree** ovvero quello con la somma dei pesi degli archi minore. Abbiamo n nodi, il grafo non lo conosco e gli archi sono pesati

Assunzioni:

- **Grafo bidirezionale**, **connesso** e **pesato** (pesi distinti e finiti);
- **PID** univoci (il peso identifica l'arco);
- Ogni nodo conosce il **peso** degli archi incidenti;
- I messaggi possono essere trasmessi in entrambe le direzioni, arrivano dopo un ritardo finito ma imprevedibile, **senza errori** e in **ordine di spedizione**;

Minimum-weight Spanning Tree:

L'algoritmo sequenziale si basa su una strategia **greedy** per produrre l'ottimo, in sostanza una volta che l'algoritmo ha prodotto un frammento dello **MST** va a prendere l'arco con peso minimo che esce da questo frammento (**Minimum-weight outgoing edge**).



Proprietà:

1. Dato un frammento di un **MST**, sia **e** il **minimum-weight outgoing edge** del frammento. L'unione di **e**, del nodo{ adiacente non appartenente al frammento **e** del frammento è ancora un frammento del **MST**.
2. Se tutti gli archi del grado hanno **pesi distinti**, allora il **MST** è unico.

Prova proprietà 2: Supponiamo esistano due **ST T** e **T'** di peso minimo. Sia **e** l'arco di peso minimo non appartenente ad entrambi. Senza perdita di generalità supponiamo che **e** appartenga a **T**.

Il grado **T'∪{e}** avrà un ciclo che conterrà **e** ed almeno un arco **e'** non appartenente a **T** poiché **T** non ha cicli. Per come è stato scelto **e**, **W(e)<W(e').** Il grafo **T'∪{e}-{e'}** sarà un nuovo **ST** con peso minore di quello di **T'**.

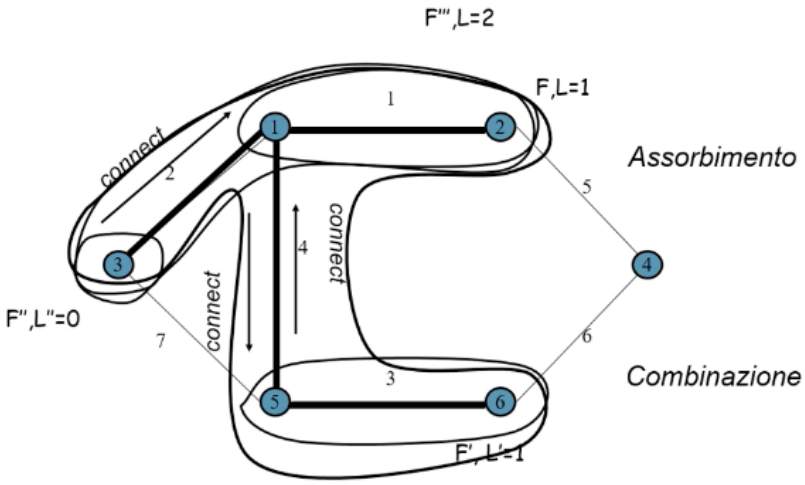
Contraddizione !!!

Terminologia dell'algoritmo distribuito:

- **Frammento:** sotto grafo connesso del MST;
- **Outgoing Edge** (OE): arco esistente tra un nodo appartenente al frammento e uno non appartenente.
- **Minimum Outgoing Edge** (MOE): OE di peso minimo.
- **Livello di un frammento:** usati nella costruzione:
  - **0:** se il frammento è costituito da un solo nodo;
  - **L+1:** se si combinano due frammenti di livello **L**;
  - **L1:** se si uniscono due frammenti di livello **L1** e **L2** con **L1>L2** (il frammento di livello **L2** viene **assorbito** dal frammento di livello **L1**);
- **Core:** arco radice di un frammento, sa cui viene ricavato il peso dello ST;

Schema dell'algoritmo:

Ogni nodo inizia l'algoritmo come frammento individuale, ogni frammento cerca il suo MOE, il frammento cerca di unirsi con il frammento che si trova all'altro estremo del MOE inviando richieste di connessione.



Ci sono due modi per unire due frammenti:

- **Assorbimento:** per esempio se un frammento di livello 0 prova a connettersi a un frammento di livello 1 viene assorbito.
- **Combinazione:** per esempio quando due frammenti dello stesso livello hanno lo stesso arco uscente di peso minimo ed interagiscono, avviene la combinazione e si ottiene un frammento di livello+1, il core di questo frammento diventa l'arco su cui passano le due connect.

Stati del nodo:

- **Sleeping:** stato iniziale;
- **Find:** sta partecipando alla ricerca del **MOE** del frammento a cui appartiene; inoltra la richiesta di ricerca di ST e fa una ricerca locale sceglie il minimo (tutti i suoi figli hanno risposto ed ha trovato l'arco di peso minimo) e quando il nodo passa nello stato found vuol dire che ha fatto report al padre.
- **Found:** ha trovato il **MOE**;

Stati dell'arco:

- **Branch:** l'arco appartiene al **MST** del frammento corrente;
- **Reject:** l'arco non è un ramo del **MST** e connette due nodi dello stesso frammento;
- **Basic**: altrimenti (arco non esplorato);

Informazioni memorizzate da ogni nodo	Messaggi scambiati
■ ***FN***, peso del *core* (id del frammento) ■ ***LN***, livello del frammento a cui appartiene il nodo ■ ***SN***, stato del nodo (Sleeping,Find,Found) ■ ***SE(j)*** per ogni arco j incidente al nodo, stato dell'arco (Basic,Branch,Rejected) ■ ***find_count***, numero degli archi su cui è in atto la ricerca del MOE ■ ***in_branch***, arco dal quale il nodo ha ricevuto il messaggio di inizio ricerca del MOE (messaggio ***initiate***) ■ ***best_edge*** e ***best_wt***, informazioni relative al nodo che ha il MOE ■ ***coda di messaggi***	■ ***Initiate(L,w,s)***; spedito per iniziare la ricerca del MOE ■ ***Test(w,L)***; spedito dai nodi nello stato Find su archi Basic per verificare se sono OE ■ ***Reject()***; spedito in risposta ad un messaggio di Test da un nodo nello stesso frammento ■ ***Accept();*** spedito in risposta ad un messaggio di Test da un nodo in un frammento diverso ■ ***Report(w)***; spedito da un nodo al padre nello ST del frammento durante la ricerca del MOE ■ ***Change-core()***; spedito dai nodi adiacenti al core ai nodi adiacenti al MOE calcolato ■ ***Connect(L);*** richiesta di unione (combinazione o assorbimento)

find_count indica a quanti figli ho inoltrato il messaggio di ricerca, quando arriva a '0 vuol dire che non devo ricevere più nulla dai figli.

best_edge indica l'arco che mi porta verso l'arco uscente di peso minimo mentre **best_wt** indica il peso migliore visto tra i report dei figli e quello calcolato localmente
Messaggio di **Test**: tra gli archi che non sono stati processati (basic) sceglie quello di peso minimo per vedere se è un arco uscente (usando Test). Come può fare un nodo a capire se un arco è un arco uscente? Manda un test con le informazioni, il nodo che riceve il test lo processerà.
Devo vedere se il frammento w da cui proviene il test è diverso dal frammento di cui faccio parte. Quindi se il nome del frammento e il livello sono gli stessi non è un arco uscente (reject).
Se non è cosi:

- il mio livello è più grande del livello che c'è nel test, significa che c'è un frammento che sta cercando l'arco uscente di peso minimo, mentre io potrei star facendo altro (stato di chi riceve non prevedibile). Quindi se chi riceve il test è a un livello superiore rispetto a chi lo manda sicuramente i frammenti sono diversi, perché il contenuto delle variabili locali di chi riceve il test potrebbero non essere aggiornati, in questa situazione il livello potrà solo crescere
- Se il livello di chi riceve il test è uguale al livello da cui arriva il test, controllo anche il nome del frammento e se è lo stesso mando un reject.
- Se il livello di chi riceve il test è più piccolo del livello da cui arriva il test, non completa e lo rimette in coda per processarlo dopo perché non può dir nulla. Devo aspettare che la situazione cambi.Questo caso verrà escluso, quindi quando invieremo la connect, lo faremo trovandoci in uno dei due casi precedenti.

Descrizione dell'algoritmo:

1. I nodi adiacenti al core inviano il messaggio **Initiate(w, L, find)** sugli archi **branch**;

2. Un nodo **p(i)** che riceve un messaggio **Initiate** aggiorna tutte le informazioni locali che riguardano il livello, la **id** del frammento e l'arco che porta il **core**. Bisogna tener presente che se cambia il core, ci sono dei nodi che non lo sanno perché questa cosa è avvenuta attraversando una path che non li coinvolgeva. Quindi lo sapranno nel momento in cui riceveranno **Initiate** perché porterà con se il livello, il nome dell'arco radice e lo stato, cosi aggiorneranno le loro informazioni locali con questi nuovi dati.
   a. Se ha archi **branch** invia **Initiate(w, L, find)** su ciascuno.
   b. Se ha archi inesplorati cerca il **MOE** locale.

3. Un nodo foglia del frammento (ha solo un arco branch) invia verso il padre **report(w)** dove **w** è il peso del suo **MOE**.

4. Un nodo interno **p(i):**
   a. Aspetta di ricevere un messaggio **report** su tutti i suoi archi **branch**, sceglie il minimo peso **w** includendo il proprio minimo locale ed invia il messaggio **report(w)** verso il padre.
   b. Aggiorna la variabile **best_edge**, se necessario: se il nodo deriva da report dei figli deve aggiornarlo
   c. Entra nello stato **Found.**

5. In nodi adiacenti al **core**, sulla base dei **report** ricevuti, trovano il **MOE** del frammento. Tra i due il nodo che, tra i report, ha ricevuto il peso minimo **w** invia un messaggio **change_core** attraverso la path dei **best_edge** fino al nodo in cui **best_edge** non è nello stato **branch**. Il nodo raggiunto invia una **connect(w, L)** sul suo **MOE**.

Come faccio a capire che sono arrivato all'arco uscente di peso minimo partendo dal core? MI fermo quando il nodo è direttamente connesso all'arco uscente di peso minimo e lo capisco perché tutti i best_edge sono già branch perché già fanno parte dello ST, quando trovo un arco che non è branch mi accorgo che devo mandare una connect. L'algoritmo in sostanza usa best_edge best_edge.. fino a quando l'arco è branch, quando best_edge non è più branch manda una connect.

Regole per la combinazione dei frammenti:

Supponendo che:

- Il nodo **p(i)** appartiene al frammento **F1(V1,E1)** con id **w1** e livello **L1**;
- Il nodo **p(j)** appartiene al frammento **F2(V2,E2)** con id **w2** e livello **L2**;

Supponendo che il nodo **p(i)** spedisce **Connect(L1)** a **p(j)** sull'arco **e** valgono le seguenti regole di combinazione:

Se $(L_2=L_1)$ ed (**e** è il MOE di F_1 ed F_2),
I nodi adiacenti al nuovo core, p_i e p_j, iniziano una nuova ricerca inviando il messaggio ***Initiate(W(e), L_1+1, Find)*** su tutti gli archi ***branch***.
-- F_1 ed F_2 si combinano.
--si crea il frammento $F(V_1 \cup V_2, E_1 \cup E_2 \cup \{e\})$ con id ***W(e)***, livello L_1+1, core **e**.

Se $(L_2>L_1)$,
p_i inizia l'assorbimento inviando un messaggio ***Initiate***(w_2, L_2, s_i)
dove s_i è lo stato (*Find* o *Found*) di p_i.
--F_1 viene assorbito da F_2.
--Si crea il frammento $F(V_1 \cup V_2, E_1 \cup E_2 \cup \{e\})$ con id w_2, livello L_2, core quello di F_2.

Se $(L_2=L_1)$ ed (il MOE di F_1 e F_2 sono diversi),
p_i aspetta che una delle due condizioni cambi.

Identificazione di un arco uscente dal frammento:

Frammento di livello 0: il MOE è uguale all'arco uscente di peso minimo.

Frammento di livello L: p(i) spedisce il messaggio **Test(w1, L1)** sull'arco **Basic** di peso minimo ad esso incidente.

p(j) alla ricezione del messaggio **Test(w1, L1):**

Se $(w_1,L_1) = (w_2,L_2)$ spedisce ***Reject()***
Se $(w_1,L_1) \neq (w_2,L_2)$ e $L_1 \leq L_2$ spedisce ***Accept()***
Se $(w_1,L_1) \neq (w_2,L_2)$ e $L_1 > L_2$ ritarda la risposta

Algoritmo:

1-2) L'algoritmo parte spontaneamente ed esegue la procedura di **wakeup:**

```
begin
   sia m l'arco uscente di peso minimo
   SE(m) = Branch;
   LN = 0;
   SN = Found;
   Find_count = 0;
   send Connect(0) sull'arco m
end
```

3) La gestione del messaggio **Connect(L)** sull'arco **j**, viene effettuata nel seguente modo:

```
begin
   if SN=Sleeping then esegui procedura wakeup;
   if L<LN then
      begin
         SE(j)=Branch;
         send Initiate(LN,FN,SN) sull'arco j;
         if SN=Find then
           find_count=find_count+1;
      end
   else if SE(j)=Basic then
           accoda il messaggio
      else
           send Initiate(LN+1,w(j),Find) sull'arco j;
end
```

Se il livello che arriva è minore del livello locale, assorbo e quindi l'arco dal quale ricevo la connect diventa branch, mando initiate sull'arco j, se lo stato è find incremento find_count di 1, cosi so a quanti nodi hon inviato initiate e da quanti ricevere report. Altrimenti se per me j è basic, devo aspettare perché i livelli sono uguali, l'arco è basic e non l'ho ancora processato. Se è branch e non è basic significa che l'ho trovato, l'ho fatto diventare branch ed ho inviato una connect., quindi invio una initiate sull'arco j, con il livello attuale+1, il core diventa il peso sul quale hanno viaggiato le due connect e siamo nello stato find.

4) La gestione del messaggio **Initiate(L, F, S)** sull'arco **j**, viene effettuate nel seguente modo:

```
begin
   LN=L; FN=F; SN=S; in_branch=j; best_edge=nil; best_wt=∞

   for all i≠j tale che SE(i)=Branch
      do  begin
          send Initate(L,F,S) sull'arco i;
          if S=Find then
                find_count=find_count+1;
          end
   if S=Find then
       esegui procedura test
end;
```

Aggiornamento delle variabili locali, ricomincio con un nuovo frammento.Per tutti gli i diverso da j, tale che j fa parte dello ST invio Initiate, se lo stato presente nel messaggio Initiate è find,inizio la procedura di test

5) Procedura **test:**

```
if archi nello stato Basic then
  begin
      test-edge = arco basic di minimo peso;
      send Test(LN,FN) su test_edge;
   end
else
  begin
     test-edge=nil;
     esegui procedura report
  end
```

Se ci sono archi nello stato basic, seleziono quello di peso minimo e diventa quello su cui sto facendo il test, se non ho archi nello stato basic allora metto test_edge a nil eseguo la procedura report per indicare che
ho terminato l'elaborazione

6) Gestione del messaggio **Test(L, F)** sull'arco **j:**

```
begin
  if SN=Sleeping then esegui procedura wakeup;
  if L>LN then accoda il messaggio;
  else
       if F ≠ FN then send Accept sull'arco j;
       else
         begin
              if SE(j)=Basic then SE(j)=Rejected;
              if test_edge ≠ j then send Reject sull'arco j;
              else  esegui procedura test
          end
  end
```

Se il livello presente in Test è maggiore del mio accodo il messaggio altrimenti se il nome del frammento è diverso dal mio allora invio una accept sull'arco j, altrimenti se sono uguali, e lo stato dell'arco j è basic lo metto a rejected ed se test_edge è diverso da j, significa che io non sto facendo test allora dico che è reject, nel caso in cui stavo facendo test su quell'arco ed ho fallito, devo ricominciare con il test.

7) Gestione del messaggio **Accept** sull'arco **j:**

```
begin
   test_edge=nil;
   if w(j) < best_wt then
      begin
         best_edge = j;
         best_wt =w(j);
      end
   esegui procedura report
end
```

Se ricevo un accept, ho terminato l'elaborazione, ed eventualmente aggiorno best_edge e best_wt ed eseguo report

8) Ricezione del messaggio **Reject** sull'arco **j:**

```
begin
   if SE(j) = Basic then SE(j) = Rejected;
   esegui procedura test;
end
```

Metto lo stato di j a rejected e rieseguo la procedura test

9) Procedura **report:**

```
if find_count = 0 and test_edge=nil then
   begin
        SN = Found;
        send Report(best_wt) su in_branch;
   end
```

10) Ricezione del messaggio **Report(w)** sull'arco **j:**

```
if j≠in_branch then
  begin
       find_count = find_count-1;
       if w < best_wt then
         begin
            best_wt = w;
            best_edge = j;
         end
       esegui procedura report
  end
else
  if SN = Find then accoda il messaggio
  else
     if w > best_wt then esegui procedura change_core;
     else if w = best_wt = infinito then halt
```

11) Procedura **change_core:**

```
if SE(best_edge) = Branch then
                send Change_core su best_edge;
else
  begin
     send Connect(LN) su best_edge;
     SE(best_edge) = Branch;
  end
```

12) risposta al messaggio **change_core:** esegui procedura **change_core;**

Terminazione:

Lemma: Da una configurazione con almeno due frammenti si perviene sempre ad una combinazione o ad un assorbimento.

Prova: Sia **L** il minimo livello di un frammento della configurazione. Sia **F** il frammento di livello **L** con **MOE** di peso minimo tra tutti i frammenti di livello **L.**

Un messaggio di **Test()** raggiunge un frammento **F'** con livello **L'≥L** oppure si crea un nuovo frammento di livello 0.

- Nel primo caso la risposta viene inviata immediatamente.
- Nel secondo caso **L** non è più il livello minimo, c'è un nuovo frammento di livello 0**.**

A partire da questa nuova configurazione possiamo iterare la strategia scegliendo il frammento di livello minimo e con il **MOE** di peso minimo. Infatti visto che il numero di nodi è finito arriveremo ad una configurazione in cui non ci sono nodi in stato di **Sleeping** e quindi tutti i messaggi **Test()** da **F** ricevono una risposta immediata. Questo significa che **F** troverà il suo **Moe** ed invierà una **connect** ad un altro frammento **F'** di livello **L'≤L.** Se **L'<L** avverrà un assorbimento, altrimenti avverrà una combinazione.

Correttezza:

Lemma: se **e** è **il core** di qualche frammento **F**, al livello **L**, non sarà core di alcun altro frammento in altri livelli.

Prova: Se **F** cambia il livello (per assorbimento o combinazione) necessariamente cambia il **core.** Il vecchio **core** di **F** dopo l'assorbimento o la combinazione diventa un arco interno (stato Branch) del **MST**. Un arco **Branch** non **core** non diventerà mai **core**.

Lemma: Un nodo in cui la **id** ed il livello del frammento sono rispettivamente **w** e **L**, appartiene ad un frammento di livello **L'≥L.**

Prova: il livello del frammento non è corretto solo durante **l'assorbimento** o la **combinazione**. In entrambi i casi il livello corretto è maggiore di quello attuale.

Lemma: se **p(j)** (frammento **F2**, core **w2,** livello **L2**) invia un messaggio **Accept** a **p(i)** (frammento **F1**, core **w1**, livello **L1**) allora **p(i)** e **p(j)** non sono nello stesso frammento.

Prova: un messaggio **Accept** è inviato sa **p(j)** a **p(i)** solo se **(w1, L1)≠(w2, L2)** e **L1≤L2.**

Se **L1<L2** allora il livello reale di **F2** può solo essere maggiore o uguale di **L2** e quindi di **L1**. I livelli sono differenti e quindi anche i core lo saranno per il lemma precedente.

Le **L1=L2** e **(w1, L1)≠(w2, L2)**, allora il reale livello di **F2** può solo essere maggiore o uguale di **L2**. Se è uguale allora le informazioni di **p(j)** sono corrette e quindi **w2≠w1.** se è maggiore di **L2**, sarà anche maggiore di **L1** e quindi i core saranno differenti.

Considerazioni sull'assorbimento: L'assorbimento avviene nel caso in cui un nodo **p(j)** del frammento **F1** invia un messaggio di **connect** sull'arco **w** ad un nodo **p(j)** del frammento **F2** ed **L1<L2.** Il nodo **p(j)** invia immediatamente il messaggio **Initiate(w2, L2, s(j))** a **p(i)** dove **s(j)** è lo stato attuale di **p(j).**

Se **p(j)** è nello stato **Find**, non ha ancora trovato il **MOE** e ed aspetterà che anche il frammento assorbito dia il suo contributo.

Se **p(j)** è nello stato **Found**, ha inviato **report(w')** e non ha inviato il messaggio **test()** sull'arco **w** coinvolto nell'assorbimento.

Dato che il **MOE** di **F2** avrà peso **w'<w** e in **F1** il messaggio **connect** è stato inviato sul **MOE w** di **F1**, è inutile chiedere a **p(i)** di cercare il **MOE.**

Complessità:

Un frammento di livello **L** contiene almeno **2^L** nodi. Il massimo livello per un frammento è **log2(N)**.

Complessità di messaggio O(E+N log N): ogni arco è posto allo stato **Rejected** solo una volta (1 messaggio di **Test** e 1 messaggio di **Reject**) quindi al più **2E(n archi)** messaggi.
Nello ST, anche se il numero di archi dovesse scendere una complessità inferiore ad **NlogN** non riusciamo a ottenere perché:
Ad ogni livello per ogni nodo:

- **Messaggi ricevuti:** al più 1 messaggio per **Initiate** e un messaggio per **Accept**;
- **Messaggi inviati:** 1 messaggio di **Test**, 1 messaggio di **Report** e 1 messaggio di **Change-core** o **Connect**;

Quindi al più **5N messaggi**.

Su tutti i livelli i messaggi saranno appunto: **5N log2 (N)** per la parte legata alla ricerca dello ST più la parte degli E archi che non fanno parte dello ST.

Lezione 3, Sincronizzazione:

Tutti i sistemi operativi attuali, che sono progettati per essere nodi di una rete, hanno al loro interno un **algoritmo di elezione**, che risponde al problema dell'impossibilità di avere una gerarchia all'interno di un sistema distribuito estremamente variabile e complesso.

In sostanza l'unico modo che si ha di **sincronizzare** è quello di eleggere un **leader (temporaneamente)** che dice a tutti quello che devono fare per poi tornare a essere un nodo normale e si riparte.

Problema dell'elezione di un leader:

Quello che si fa è, dati degli **ID** univoci per i nodi, selezionare secondo dei criteri uno di questi **ID** (minimo/massimo) che diventerà il leader, dall'altro canto se non si hanno degli **ID** distinguibili vale il seguente teorema:

Teorema: Non esiste un algoritmo deterministico per eleggere un leader in un anello in cui i nodi non sono distinguibili.

Prova (Da lui saltata): Per il modello sincrono (caso particolare di quello asincrono). Tutti i nodi hanno la stessa funzione di transizione. Se i nodi non sono distinguibili, le transizioni dipendono solo dai messaggi ricevuti.

Poiché tutti i nodi partono dallo stato iniziale (con nodi che non hanno un id unica), tutti eseguiranno la stessa transizione e tutti invieranno gli stessi messaggi durante il primo round. Poiché le transizioni dipendono solo

dai messaggi ricevuti, tutti eseguiranno la stessa transizione ed invieranno gli stessi messaggi al round due… Pertanto se esistesse un algoritmo, tutti i nodi raggiungerebbero lo stato di eletto.

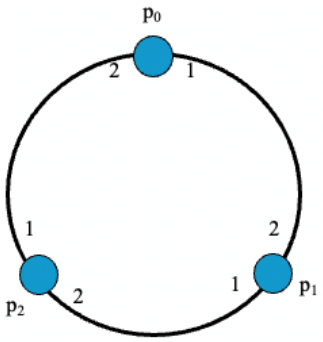
In sostanza il problema è **un eccesso di simmetria…** Ci serve **randomizzare…** e questo processo viene fatto tante volte fino a quando non c'è un minimo unico (algoritmo probabilistico)

Definizione del problema:

Ogni nodo ha un sottoinsieme di stati **eletto** ed un sottoinsieme di stati **non eletto**. Se un nodo entra in uno dei due sottoinsiemi, vi rimane stabilmente. In particolare in un esecuzione ammissibile:

- Esattamente un processore entra nello stato di **eletto;**
- Ogni processore entra nello stato **eletto** oppure **non eletto**;

Anelli orientati: per semplificare il problema assumeremo di utilizzare degli **anelli orientati** in cui ogni processore ha una nozione consistente di destra e sinistra.



Inoltre esistono due classi di algoritmi per questa classe di problemi:

- **Algoritmi uniformi:** non usano (non conoscono) la dimensione dell'anello. In cui ogni nodo usa la stessa macchina a stati **A** qualunque sia la dimensione. In questo caso la macchina a stati di ogni singolo nodo non dipende da n ma dipende dall'id del nodo,
- **Algoritmi non uniformi:** usano (conoscono) la dimensione dell'anello. In cui la macchina a stati è una collezione di macchina **A(n)** una per ogni dimensione dell'anello. In questo caso dipende da n.

Algoritmo:

Ogni nodo estrae un intero tra **1** e **n** usando una distribuzione uniforme e lo assume com **ID**. Il nodo eletto sarà quello con **ID** minimo. Ma l'id minima **potrebbe non essere unica**.

Questa tipologia di soluzione rientra nella classe di algoritmi **Trial-and-Error** che constano di tre passi:

1. Generazione casuale di identificatori
2. Verifica delle condizioni di elezione
3. Procamazione

Dove i passi **1** e **2** sono ripetuti finché la verifica non da esito positivo (**fase**), il passo **3** invece viene eseguito per informare tutti della avvenuta elezione (**terminazione**).

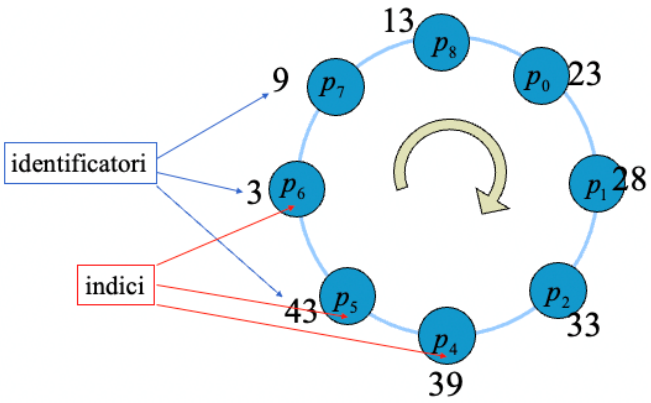
Discorso sulla probabilità di generare un minimo univoco (secondo me trascurabile):

Il numero medio di fasi per la generazione di un minimo univoco è di 3.

Complessità di messaggio: data dal **numero di fasi** (in media 3) * **complessità di una fase**, ovvero dell'algoritmo di elezione su **rete identificata** (ovvero di andare a trovare il minimo effettivo).

Algoritmi di proclamazione:

Mi trovo in uno stato ideale, ovvero con tutti i nodi identificati con un **ID** univoco, e devo pertanto solo calcolare il minimo per eleggerlo leader.



3,9,13,23,28,33,39,43

Primo algoritmo:

Un idea è quella che tutti i nodi inviino un messaggio a **destra** e ogni nodo vada confrontare questo messaggi con un minimo locale salvato, se il messaggio è minore, il minimo viene aggiornato e il messaggio viene lasciato passare altrimenti il messaggio viene **bloccato**.

```
Invia ID a sinistra
Quando ricevi un ID' da destra:
        se ID'>ID lascia proseguire ID' e passa nello
                stato non eletto
        se ID'=ID passa nello stato eletto
```

La **correttezza** è banale, in quando l'unico messaggio che viene lasciato passare è quello che contiene l'**ID minima**. La **complessità di messaggio** invece non è delle migliori in quanto nel caso pessimo (Anello di ID ordinato al contrario) il numero di messaggi inviati sarà dato da:

$$\sum_{i=1}^{n} i = \Theta(n^2)$$

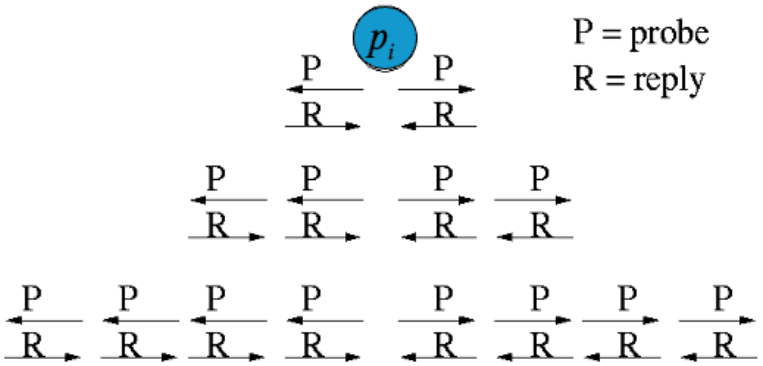
Secondo algoritmo (Descritto meglio sul libro e da scrivere meglio con le cose che dice):

Algoritmo è organizzato in **fasi**, all'inizio di ogni fase il nodo invia lo stesso messaggio **PROBE**, contente la proprio **ID**, contemporaneamente a **destra** e a **sinistra**. I destinatari sono due processori la cui distanza da chi spedisce cresce ad ogni fase. In particolare la distanza parte da 1 e raddoppia ad ogni fase.

```
Se ricevi un PROBE contenente ID':
   se ID' > ID allora
      se raggiunta la destinazione allora
         invia indietro un messaggio REPLY
      altrimenti
         lascia proseguire il messaggio
   se ID'=ID allora
         passa nello stato eletto
```

```
Quando ricevi un REPLY non tuo
        lascia passare il messaggio
Quando ricevi entrambi i tuoi REPLY
        passa alla fase successiva
```

Nel messaggio è contenuto anche la fase, in modo che il node che riceve il probe sa se è l'ultimo. Una volta arrivati all'ultimo nodo viene generato un messaggio di ritorno chiamato **REPLY**, in modo che se al nodo che ha mandato il **PROBE** arrivano entrambi i **REPLY** quest'ultimo avanza di fase.



Correttezza: L'**ID** massimo non viene mai fermato.

Complessità di messaggio: Un **PROBE** in fase **i** produce al più **2^i** messaggi. Per ogni **PROBE** esiste un corrispondente **REPLY** questo implica che un nodo contribuisce per al più **4*2^i** messaggi nella fase **i**. I nodi che partono nella fase **i** sono i vincitori della fase precedente che sono al più:

$$\frac{n}{2^{i-1}+1}$$

Si può pertanto calcolare che tale algoritmo ha complessità pari a: **5n+8n log(n)** (non si può fare di meglio! sotto la dimostrazione).

Dimostrazione che il secondo algoritmo è asintoticamente ottimo:

Consideriamo un algoritmo di elezione **A** per un anello con le seguenti caratteristiche:
- Asincrono
- Uniforme
- Elegge il nodo con **ID** massimo
- Garantisce che tutti conoscano l'**ID** del leader

Dimostriamo che la complessità di messaggio di **A** è limitata inferiormente da: **Ω(n log n)**

Teorema: Per ogni insieme **S** con **n=2^d** identificatori ed ogni anello che usa gli elementi di **S** come **ID**, l'algoritmo **A** ammette uno schedule aperto in cui vengono spediti almeno **M(n)** messaggi con:

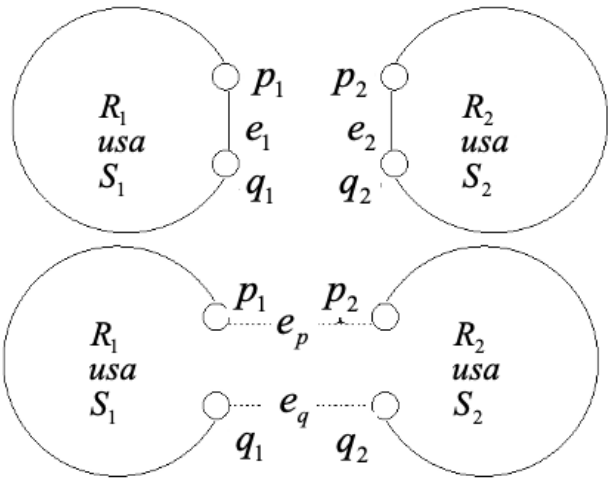
$$M(2)=1$$

$$M(n)=2\cdot M(\tfrac{n}{2})+\tfrac{1}{2}\cdot(\tfrac{n}{2}-1), n>2$$

Prova per induzione: per **n=2** supponiamo che **x>y**. Ad un certo punto **p(0)** deve mandare un messaggio a **p(1)** in modo da comunicargli il proprio valore. Se fermiamo l'esecuzione dopo aver inviato il primo messaggio lo schedule è quello desiderato.
Per **n≥4**, dividiamo **S** in due metà **S'** e **S''**. Per ipotesi induttiva esistono due anelli **R'** e **R''**, il primo usa **S'** e il secondo **S''**.

Inoltre:
- **R'** ha uno schedule aperto **α'** in cui vengono spediti **M(n/2)** messaggi e sia **e' = (p', q')** l'arco aperto.
- **R''** ha uno schedule aperto **α''** in cui vengono spediti **M(n/2)** messaggi e sia **e'' = (p'', q'')** l'arco aperto.

Usiamo **R'** e **R''** in corrispondenza dei due archi aperti in modo da ottenere **R**:



Iniziamo a costruire lo schedule su **R** nel seguente modo:

- Eseguiamo **α'**
- Eseguiamo **α''**

Poiché l'algoritmo è uniforme le due esecuzioni sono le medesime di quelle effettuate in **R'** e **R''**. Vengono quindi inviate **M(n/2)** messaggi eseguendo **α'** in **R** ed altrettanti eseguendo **α''.**

Sono quindi possibili due casi:

1. Esiste un'estensione di **α'α''** in **R** in cui vengono spediti altri **1/2(n/2 - 1)** messaggi senza utilizzare gli archi aperti. Abbiamo lo schedule.
2. Tutte le estensione di **α'α''** che non utilizzano gli archi aperti portano ad uno stato di quiescenza, questo implica che gli unici messaggi in transito sono sugli archi aperti. Sia **α'''** una di tali estensioni.

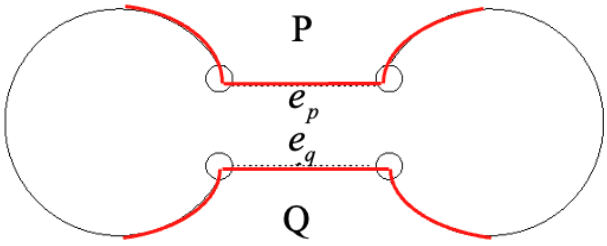
Sia quindi **α'α''α'''α''''** uno schedule ammissibile, che raggiunga quindi la terminazione, vale la seguente affermazione:

Claim: almeno **n/2** messaggi sono inviati in **α''''**.

Prova: poiché il leader deve inviare a tutti la sua **ID**, dovrà inviare un messaggio alla metà dell'anello cui non appartiene. Prima di **α''''** le due metà non si sono scambiate alcun messaggio.

Sia **α''''*** il più piccolo prefisso di **α''''** in cui sono stati inviati **n/2 - 1** messaggi. I messaggi inviati lungo i due archi aperti non possono incontrarsi duranti **α''''*** poiché complessivamente sono stati inviati meno di **n/2** messaggi.

Chiamiamo **P** (**risp. Q**) l'insieme dei processori che durante **α''''*** è stata raggiunto da messaggi inviati sull'arco **(p', p'')** risp. **(q', q'')**).



Da quanto detto, **P** e **Q** sono disgiunti. Senza perdita di generalità supponiamo che **P** sia, tra i due, l'insieme dei processori coinvolto nella spedizione/ricezione del maggior numero di messaggi. Sia **α''''** la sequenza di eventi ottenuti da **α''''** prendendo solo gli eventi di processori in **P** ne deduciamo che **α'α''α'''α''''** è lo schedule cercato, infatti è aperto e il numero di messaggi inviati è almeno:

$$2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2}\left(\frac{n}{2} - 1\right)$$

Algoritmo Sincrono per la proclamazione del leader:

Nel caso **sincrono** esiste un orologio comune, quindi tutti contano con la stessa frequenza. Se la dimensione dell'anello è nota, si può, ad esempio, inviare un messaggio a destra e sapere a che istante esso dovrà tornare a sinistra. **Questo ci permette di acquisire informazioni senza ricevere messaggi**.

Algoritmo non uniforme:

Ogni fase dura **n** round all'interno dei quali viene eletto il processore con **ID** minima. Alla fase **i**, il processore con **ID** uguale ad **i**, se esiste, invia un messaggio, aspetta che il messaggio torni e si elegge leader.

Correttezza: il processore con **ID** minimo **min** invia il messaggio **msg(min)** all'istante **min*n.** Il processore con **ID** subito successivo **min+1** invierebbe il suo messaggio **n** istanti dopo. Ma esattamente all'istante **(min+1)n**, **msg(min)** torna al processore **min** avendo posto in uno stato non eletto tutti gli altri processori.

Complessità:
- **Messaggi: O(n)**
- **Tempo: O(n*min)** non limitata da n.

L'algoritmo precedente usa la funzione di attesa **wait(i) = n * i**. Cosi facendo il processore **i** attende **wait(i)** prima di inviare messaggi. La funzione di attesa è tale che**:**

$$wait(i)+n \le wait(i+1)$$

L'algoritmo funziona a patto che i processori partano insieme. Se cosi non è, la funzione di attesa deve tener conto del fatto che il processore con **ID = i** può partire **n - 1** istanti dopo il processore con **ID = i + 1**, pertanto la funzione deve godere della seguente proprietà:

$$wait(i)+2n-1 \le wait(i+1)$$
$$wait(i+1)-wait(i) \ge 2n-1$$

Che con la condizione iniziale **wait(0)=0** ha soluzione del tipo:

$$wait(i)=(2n-1)i$$

L'algoritmo pertanto diventa:

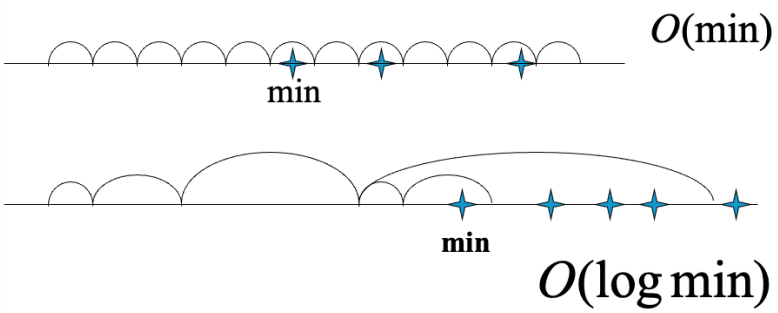
se non hai ricevuto messaggi allora invia <inizio elezione> a destra conta **wait(i)** se ricevi <inizio elezione> allora se non in attesa allora invia <inizio elezione> a destra conta **wait(i)**	se ricevi <eletto> allora se sei in attesa allora passa nello stato *non eletto* se termina conteggio e stato diverso da *non eletto* allora invia < eletto> a destra aspetta n passa nello stato *eletto*

Correttezza: simile al precedente.

Complessità: i messaggi <inizio elezione> sono **n** i messaggi <eletto> sono **n**, il tempo totale è **(2n-1)min**.

Algoritmo di ricerca in una tavola non limitata:

Il problema dell'elezione è stato convertito nel problema di cercare un estremo, gli algoritmi visti fino ad ora sono ottimali sul numero di messaggi, ma per entrambi la complessità di tempo è funzione del valore minimo di **ID**.



E' quindi necessario trovare un algoritmo efficiente per la ricerca di un elemento in una tavola non limitata.

L'algoritmo proposto è organizzato in fasi, dove ogni processore che accede alla fase **j** comincia la ricerca del valore **k(j)**. Ogni processore ha inoltre una procedura che gli permette di calcolare la stessa sequenza di valore **b1b2b3…** Inoltre essendo sincrono, al round **i** tutti i processori producono **b(i).**

Al round i della fase j: se $ID \leq k_j + b_i$ allora invia un messaggio e aspetta n istanti. sovrastima se ricevi un messaggio prima di n allora aspetta fino ad n e passa alla fase successiva se ricevi un messaggio all'istante n allora passa nello stato eletto	sottostima altrimenti {se ID > $k_j + b_i$} aspetta n istanti se ricevi un messaggio allora passa nello stato non eletto altrimenti passa al round successivo

Complessità:
- **Tempo: O(n log^2 min)**
- **Messaggi: O(n sv(A))** dove **sv(A)** è il numero di sovrastime che l'algoritmo effettua.

Algoritmo di ricerca in una tavola non limitata uniforme:

I processori non necessariamente partono allo stesso round, un processore infatti, o si sveglia spontaneamente oppure alla ricezione di un messaggio. Inoltre il numero di processo **n** non è noto.

Funzionamento dell'algoritmo:
Un processore che si sveglia spontaneamente (**attivo**) invia la sua **ID** in un messaggio **veloce** (1 arco/round). Mentre un processore che si sveglia alla ricezione di un messaggio (**relay**) non è in gara.

Un messaggio **veloce** inoltre diventa **lento**, 1 arco/2^m round (**m** è la **id** del messaggio), se raggiunge un processore attivo.

I processori (**attivi** o **relay**) inoltrano solo messaggi la cui **ID** è minore di una qualunque altra **ID** osservata in precedenza (ignorando le **ID** dei processori relay). Se un processore riceve la propria **ID**, si proclama leader.

Algoritmo:
Inizialmente la coda di **waiting** è vuota e **status = asleep**

```
1   // Sia R l'insieme di messaggi ricevuti in questo evento
2   S := insieme vuoto // messaggi da spedire
3   if (status = asleep):
4       if  (R è vuoto):
5           status = participating
6           min := id
7           aggiungi <id,1> ad S //messaggi di prima fase
8       else
9           status = relay
10          min := infinito
11  for (ciascun <m,h> in R):
12      if (m < min):
13          diventa non eletto
14          min := m
15          if ((status = relay) & (h = 1)):
16              aggiungi (m,h) to S
17          else
18              aggiungi (m,2) a waiting salvando il roudn corrente
19      else if (m = id):
20          diventa eletto
21  for (ciascun <m,2> in waiting):
22      if (<m,2> è stato ricevuto 2^m - 1 round prima):
23          rimuovi <m,2> da waiting e aggiungilo a S
24  Invia S
```

Correttezza: il processore con **ID** minima **min**, tra i processori attivi, viene eletto.

Prova: Supponiamo che anche **<j>,** oltre a **<min>**, torna al processore che lo ha inviato. E' passato quindi anche per **p(min)**. Ma poiché **min < j**, **p(min)** non lo inoltra. Contraddizione.

Complessità di messaggio: il messaggio del leader è il più veloce. In un unico giro dell'anello riesce a superare tutti gli altri messaggi e a fermarli prima che molti messaggi vengano inviati. Inoltre dividiamo i messaggi in tre categorie:

- Messaggi veloci
- Messaggi lenti inviati mentre il messaggio del leader è veloce.
- Messaggi lenti inviati mentre il messaggio del leader è lento.

Sincronizzazione con Memoria condivisa:

I processori comunicano attraverso un insieme di **variabili condivise.** Ogni variabile condivisa ha un **tipo** che definisce le operazioni che possono essere eseguite sulla variabile in maniera **atomica**. Più nel dettaglio:

- Non ci sono **inbuf** ed **outbuf;**

- La configurazione include il valore delle **variabili condivise**;
- Il solo tipo di evento è un **passo di computazione**;
- Una esecuzione ammissibile è quella in cui ogni processore esegue un **numero infinito di passi**;

$$n \text{ processori } (p_0, p_1, \ldots, p_{n-1})$$
$$m \text{ registri condivisi } (R_0, R_1, \ldots, R_{m-1})$$

Quando **p(i)** esegue un passo:
- Lo stato attuale di **p(i)** definisce a quale variabili condivise bisogna accedere e con quali operazioni.
- Il valore delle variabili viene aggiornato in accordo alla semantica delle operazioni specificate dalla funzione di transizione.
- Lo stato di p(i) si aggiorna sulla base dello stato precedente e del risultato delle operazioni.

Complessità di spazio:

Per quanto riguarda la memoria condivisa, siamo interessati alla complessità di spazio, ovvero:
- Quante variabili condivise sono necessarie;
- Quanti bit per ogni variabile condivisa;

Configurazione per il modello a memoria condivisa:

Una configurazione per il modello a memoria condivisa è il vettore, **(q(0),q(1), … , q(n-1), r(0),r(1), … , r(m-1))** dove **q(i)** è lo stato del processore **p(i)** e **r(j)** è il valore della variabile condivisa **R(j).**

Un'esecuzione di un algoritmo è una sequenza (finita o infinita) del tipo, **C(0), Φ(0), C(1), Φ(1), C(2), Φ(2), …**

Mutua esclusione, Deadlock, No lockout:

I processi devono condividere una risorsa, uno per volta, ovvero in **Mutua Esclusione (ME)**, evitando di incorrere in **Dead Lock (ND)** ovvero situazioni in cui solo qualche processore, tra quelli interessati, accede alla risorsa.

E' inoltre fondamentale che alla fine tutti accedano alla risorsa **No Lockout (NL)** e che ci sia un limite superiore di attesa **Bounded Waiting (BW).**

Quantità di registri (non molto chiaro, forse intende le operazioni sui registri ?):

Se utilizziamo solo due tipologie di registri (lettura e scrittura) per assicurare la **ME** saranno necessari **n** registri (sia come **Upper Bound** che come **Lower Bound**).

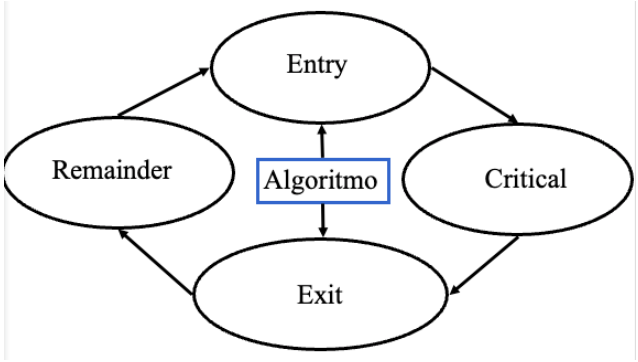
Se invece utilizziamo tre tipologie di registri (lettura/modifica/scrittura) avremo i seguenti bound in base a ciò che vogliamo ottenere:

	Upper	Lower
ND	O(1)	O(1)
NL	O(log n)	O(log n)
BW	O(log n)	O(log n)

Schema di esecuzione:

Distinguiamo tra quattro stati di esecuzione:
- **Remainder:** Stato in cui il processore non è interessato a entrare nella sezione critica;
- **Entry:** Stato di ingresso alla sezione critica;
- **Exit:** Stato di uscita dalla sezione critica;
- **Critical:** Sezione critica;

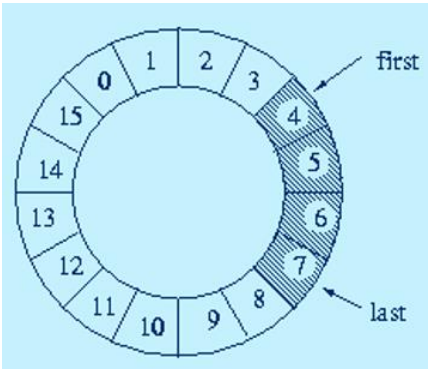
L'**algoritmo** si pone tra la sezione di **ingresso** e di **uscita** alla **CS**



Discorso molto fast sul Set&Test (Lezione 3 Slide 133/134)

Un primo algoritmo:

Basato sull'uso di una coda circolare con **n** ingressi, in sostanza se voglio entrare nella sezione critica me metto in coda e attendo.



Questo si traduce nel mantenere due registri, **l'indice della coda** e **l'indice della testa**. In tutto sono necessari **log(n) bit** che è il meglio che si può fare, dimostrato dal seguente teorema:

Teorema: Un algoritmo per la mutua esclusione tra **n** processori che assicura il **ND** e il **k-BW (**se sono in coda con un altro nodo, posso rimanere in attesa mentre l'altro accedere al massimo **k** volte poi tocca a me, **BW** normale invece vale solo per 1 accesso, poi tocca a me**)** ha necessità di variabili condivise con almeno **n** stati distinti (nelle variabili posso scrivere **n** valori distinti, quindi ho bisogno di n bit).

Prova (scriverla in base a quello che dice lui): La configurazione iniziale **C** è certamente una configurazione quiescente (nessuno è interessato alla sezione critica). Estraiamo dallo schedule (infinito) dell'algoritmo gli eventi (infiniti) che riguardalo il solo processore **p(0)**.

Uno schedule di questo tipo lo chiamiamo **p(0)**

Registri R/W:

Esistono diversi algoritmo per l'accesso alla mutua esclusione:

Algoritmo del Fornaio (**ME**, **NL**): algoritmo noto che ha il problema che bisogna usare degli artefici per limitare il contatore superiormente (tutto quel bordello di Scarano).

Algoritmo per due processori con variabili di dimensione limitata (**ME**, **NL**): abbiamo tre variabili condivise:

- **want(0)** e **want(1)**;
- **priority**: vale 0 (processo 1) oppure 1 (processo 2);

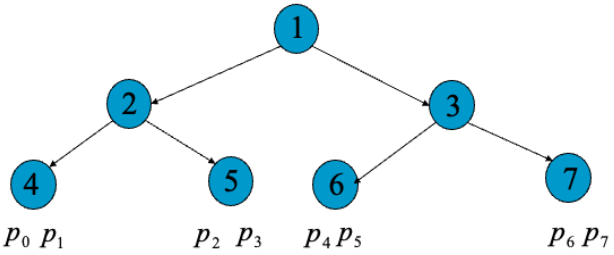
Metto la mia variabile condivisa a 0, per dire che non sono interessato (in realtà lo sono), successivamente aspetto fino a quando la priorità diventa mia o l'altro non è più interessato, dopo di che setto la mia variabile condivisa a 1 e mi chiedo se la priorità è dell'altro e l'altro è interessato ala CE ricomincio, altrimenti aspetto che l'altro finisce e setta il suo want a 0 ed entro in CE.

<entry>
want$_0$=0
wait until (want$_1$=0)
or (priority=0)
want$_0$=1
if (priority=1) then
if (want$_1$=1) then restart
else wait until (want$_1$=0)
 	
<exit>
priority=1
want$_0$=0
	<entry>
	want$_1$=0
	wait until (want$_0$=0)
	or (priority=1)
	want$_1$=1
	if (priority=0) then
	if (want$_0$=1) then restart
	else wait until (want$_0$=0)
	
	<exit>
	priority=0
	want$_1$=0

p_0	p_1

La variabile wait messa a 0 serve a sbloccare e a evitare i deadlock. Il tutto funziona usando tre bit.

Algoritmo per n processori con variabili di dimensione limitata (**ME**, **NL**): L'idea si basa su un torneo, in cui una copia dell'algoritmo per due processori è associata ad ogni nodo dell'albero, se per esempio ho 8 processori si ha questa situazione:



Quello che succede è che fra **p(0)** e **p(1)** uno solo passa, poi viene lanciata un altra istanza dell'algoritmo **(nodo 2)** e cosi via finché non si arriva a un unico processore che entrerà in **CE**.

Il vantaggio sta nel fatto che il numero totale di istanze è lineare ed ogni processore ha bisogno di una complessità di spazio pari a **3n** variabili booleane **R/W**.

Algoritmo:

```
1   Node(v:integer, side: 0..1){
2       Wantv [side] := 0
3       wait until (Wantv [1-side] = 0 or Priorityv = side)
4       Wantv [side] := 1
5       if(Priorityv = 1-side):
6           if(Wantv [1-side] = 1):
7               goto 1
8       else
9           wait until (Wantv [1-side] = 0)
10      if(v=1):
11          <Critical>
12      else
13          Node(v/2, v mod 2)
14          Priorityv := 1-side
15          Wantv [side] := 0
16  }
```

Lezione 4, Consenso con crash:

Ogni processore **p(i)** (0≤i≤n-1) in un Sistema Distribuito ha un input **x(i)** ed un output **y(i)**. Un sottoinsieme dei processori può essere in errore (**faulty**).

Una soluzione a questo problema deve garantire:

- **Terminazione:** In ogni esecuzione ammissibile e per tutti i processori non faulty **p(i)**, ad **y(i)** (output) viene assegnato un valore;
- **Accordo:** In ogni esecuzione se per una coppia di processori non faulty **p(i)** e **p(j)** ad **y(i)** sono stati assegnati valori, allora **y(i)=y(j)**;

- **Validità (debole):** In ogni esecuzione, se per ogni **x(i)=v** e se, per qualche processore non faulty **p(i)**, ad **y(i)** è stato assegnato un valore, allora **y(i)=v.** In altre parole, se tutti gli input sono uguali, l'output deve essere uguale. Serve ad evitare di accettare algoritmi che di fatto non sono algoritmi di consenso.

Modello Message Passing utilizzato:

Si effettuano le seguenti assunzioni:
- Si assume che il grafo sia **completo;**
- Si assume che il sistema si **sincrono;**
- Si assume che siano possibili errori di tipo **crash;**

Errori di tipo crash:

Un **processore** ha un errore (fault) di tipo **crash** se a partire da un round smette ogni attività di comunicazione. Nel round in cui avviene il crash può inviare un sottoinsieme dei messaggi che dovrebbe inviare in quel round.

Un **sistema** è f-resilient se assicura un corretta esecuzione quando al più **f** processori falliscono. In particolare un esecuzione in un sistema di questo tipo ha le seguenti proprietà:
- Esiste un sottoinsieme **F** di processori con al più **f** processori in fault.
- L'insieme **F** cambia ad ogni esecuzione.
- Ciascun round è composta da esattamente un passo di computazione per i processori non in **F** ed al più un passo di computazione per i processori in **F**.
- Se un processore in **F** non ha un passo di computazione in un certo round, non avrà passi di computazione in tutti i round che seguono.
- Nell'ultimo round in cuci un processore in **F** ha un passo di computazione invia un sottoinsieme dei messaggi previsti in quel round.

Algoritmo:

Ogni processore ha un insieme **v** locale inizialmente ci mette solo il suo input e poi itera **f+1** il codice sotto, ovvero manda **v** a tutti gli altri processori se non è stato già inviato, riceve valori dagli altri e aggiorna **v,** quindi **v** conterrà l'input di tutti gli altri, alla fine delle iterazioni, calcolo il minimo di **v** e lo scrivo in **y(i).**

Inizialmente $V=\{x_i\}$

round k, $1 \leq k \leq f+1$

1: send $\{v \in V: v \text{ non è stato ancora inviato da } p_i\}$
2: receive S_j da p_j $0 \leq j \leq n-1$, $j \neq i$
3: $V := V \cup S_0 \cup S_1 \cup \ldots \cup S_{i-1} \cup S_{i+1} \cup \ldots \cup S_{n-1}$
4: if $k=f+1$ then $y_i := \min(V)$

Se ho un fault al primo round, un nodo non manda tutti i messaggi, quindi gli insiemi v, non sono uguali e quindi potrebbe non esserlo anche il minimo, al secondo passo però rimangono **n-1** processori che rimandano tutto e gli insiemi sono di nuovo uguali. Se i fault sono **f** invece, c'è almeno un round dove almeno un processore non va in fault e quindi i valori vengono spediti lo stesso.

Le proprietà di **terminazione** e **validità** sono immediatamente verificate, mentre per la proprietà di **accordo** occorre essere più formali:

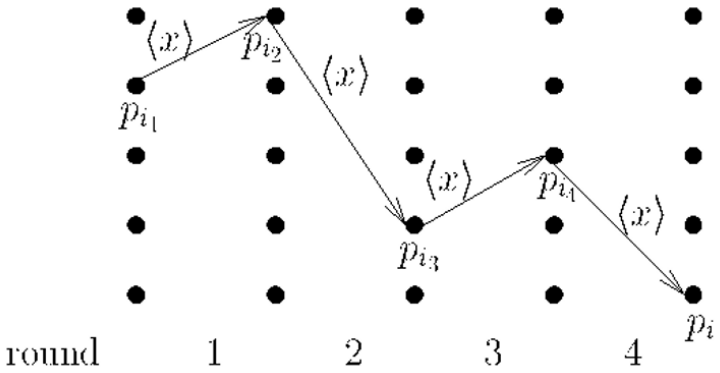
Limite superiore al numero di round:

Lemma: In ogni esecuzione, alla fine del round **f+1**, **V(1)=V(j)**, per ogni coppia di processori non faulty **p(i)** e **p(j)**

Proveremo che se **x∈V(i)** alla fine del round **f+1** allora **x∈V(j)** per ogni coppia di processori non faulty **p(i)** e **p(j)**.

Sia **p(i)** un processore non faulty. Sia **r** il primo round in cui **x** è aggiunto a **v(i).** Si assume che **r = 0** se **x** era in **V(i)** al primo round. Se **r≤f** allora al round **r+1≤f+1 p(i)**, che non è faulty, invia **x** a tutti gli altri. I processori non faulty riceveranno **x** e lo aggiungeranno al proprio **V**.

Supponiamo che **r=f+1**, questo significa che **p(i)** ha ricevuto **x** per la prima volta all'ultimo round. Ma questo implica che esiste una catena di **f+1** processori in fault distinti tra loro e distinti da **p(i)** che ha trasmesso **x** a **p(i)**.



Siccome il sistema è **f-resilient**, **f+1** processori faulty non possono esserci, il processore **p(i)** non può ricevere **x** per la prima volta al round **f+1**. Possiamo quindi enunciare il seguente teorema:

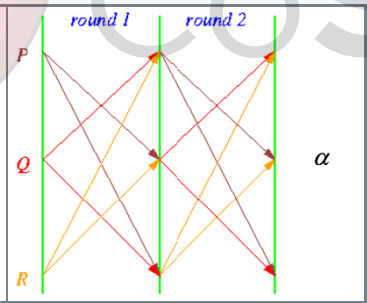
Teorema: f+1 round sono sufficienti per risolvere il problema del consenso in presenza di **f** fault di tipo crash.

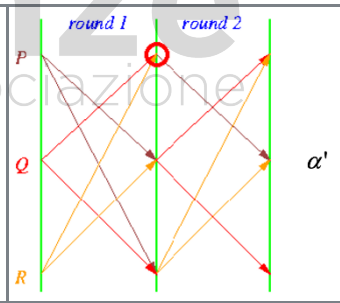
Limite inferiore al numero di round:

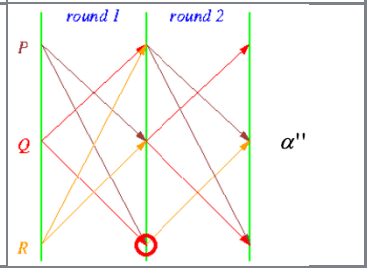
Proveremo che **f+1** è anche un limite inferiore al numero di round in presenza di errori di tipo crash.

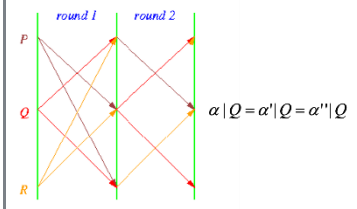
Caso semplice: mostriamo che in presenza di **un** fault, **due** round sono necessari.

Definizione: Sia α una esecuzione e sia p_i un processore. La vista che p_i ha di α, denotata con $\alpha\|p_i$, è data dalla sottosequenza di eventi di computazione e delivery che avvengono in p_i durante α e dallo stato di p_i nella configurazione iniziale di α.	Definizione: Siano α_1 e α_2 due esecuzioni e sia p_i un processore non faulty in α_1 e in α_2. L'esecuzione α_1 è simile alla esecuzione α_2 rispetto a p_i se $\alpha_1\|p_i = \alpha_2\|p_i$. Indicheremo tale similarità con il simbolo









Lemma: se **α(1)≈(p(i))α(2)** con **α(1)** e **α(2)** esecuzione ammissibili e **p(i)** processore non faulty, allora il valore di decisione in **α(1) dec(α(1))** è uguale al valore di decisione in **α(2) dec(α(2)).**

La chiusura transitiva di **≈(p(i))** è indicata con **≈** ed è definita come segue:

Definizione: α(1)≈α(2) se esistono esecuzione **β(1), β(2), …, β(k+1)** ed una sequenza di processori non faulty **p(i)(1), p(i)(2), …, p(i)(k)** tali che:

$$\alpha_1 = \beta_1 \approx_{p_{i_1}} \beta_2 \approx_{p_{i_2}} \ldots \approx_{p_{i_k}} \beta_{k+1} = \alpha_2$$

Lemma: siano **α(1)** e **α(2)** due esecuzioni ammissibili. Se **α(1)≈α(2)** allora **dec(α(1))=dec(α(2)).**

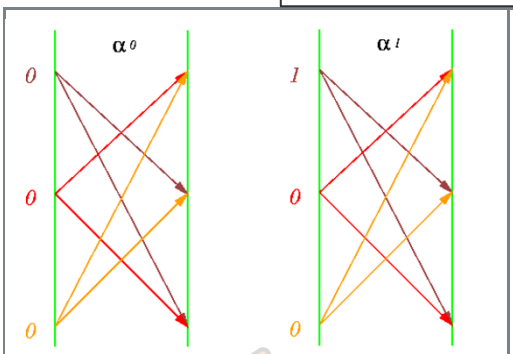
Nelle prove che seguono si assume che tutti i processori inviano un messaggio (eventualmente vuoto) in ogni round.

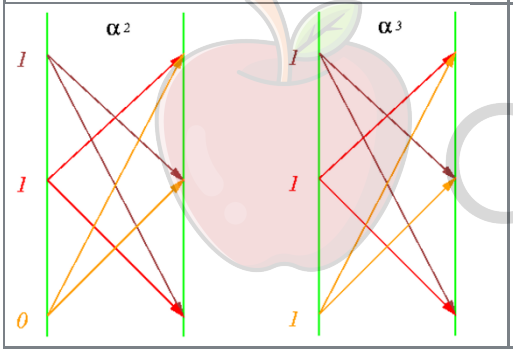
Teorema: Se **n≥3**, non esiste un algoritmo che risolve il problema del consenso:
- Con meno di 2 round.
- In presenza di 1 fault di tipo crash.

Prova per contraddizione: supponiamo esista un algoritmo **C** che in 1 round risolva il problema del consenso in presenza di 1 fault di tipo **crash** ed **n≥3**.

Costruiamo quindi la sequenza di esecuzioni ammissibili **α(i) i=0,1, …, n** nel modo seguente:

α^i i=0,…,n è l'esecuzione ammissibile dell'algoritmo C in cui
- i primi i processori (p_0,…,p_{i-1}) ricevono come input 1
- gli altri (p_i,…,p_{n-1}) ricevono come input 0
- non ci sono fault





Mostreremo che:

$$\alpha^0 \approx \alpha^1 \approx \ldots \approx \alpha^n$$

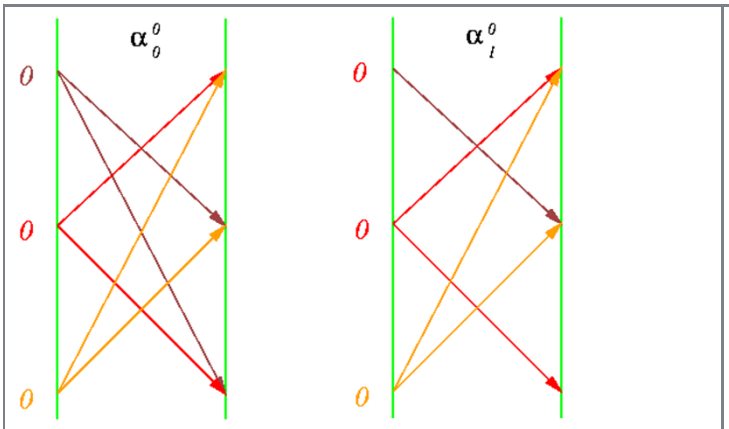
Supponiamo che sia vero. A causa della proprietà di validità **dec(α^0)=0** e **dec(α^n)=1.** Dalle proprietà della relazione ≈ possiamo d'altra parte affermare che **dec(α^0)=dec(α^n). Contraddizione !!**

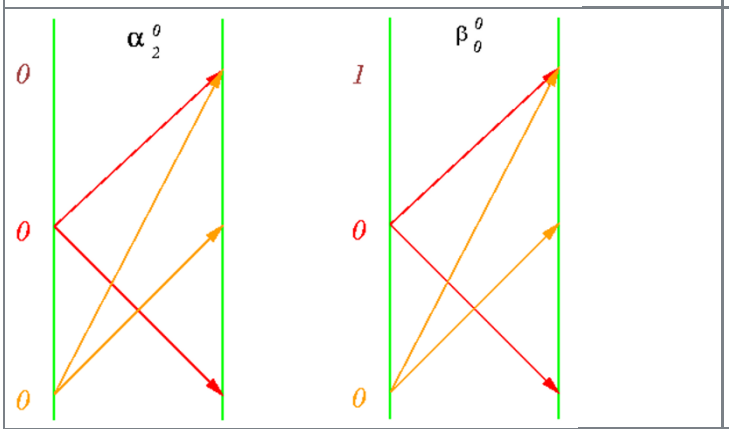
Occorre mostrare che per **i=0,1, … n-1:**

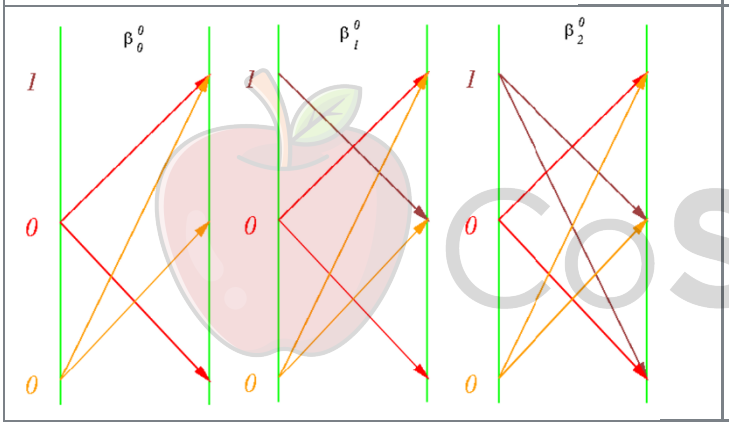
$$\alpha^i \approx \alpha^{i+1}$$

Illustriamo prima la strategia per **n=3**. Mostriamo che **α^0≈α^1** costruendo una nuova sequenza di esecuzione:

$$\alpha_0^0 \alpha_1^0 \alpha_2^0 \beta_0^0 \beta_1^0 \beta_2^0$$



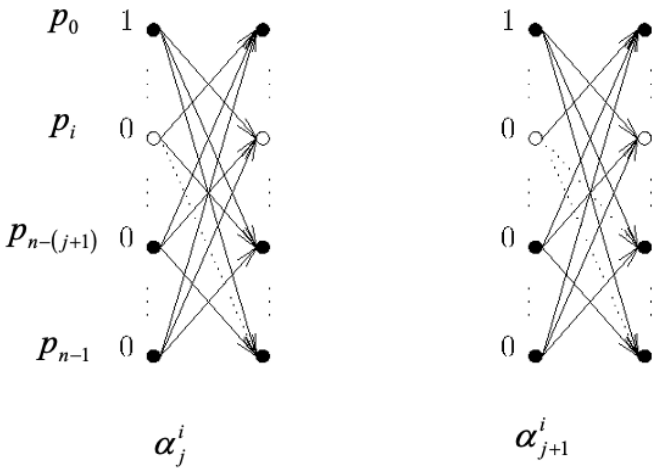




Costruiamo una sequenza di esecuzioni simili che a partire da **α^i** raggiunge **α^(i+1)**. Le prime **n**, **α^i(j) j=0,1, …,n-1**, sono costruite imponendo il fallimento del processore **p(i)** nel modo seguente:

- **α^(i)0** Coincide **α^i**;
- **α^(i)1 p(i)** non invia il messaggio al processore **p(n-1);**
- …
- **p(i)** non invia il messaggi ai processori **p(n-1), p(n-2), …, p(n-j);**

Escludendo il processore **p(i)**.



Le altre **n**, **β^i(j) j=0,1, …, n-1**, sono costruite a partire da **α^i(n-1),** sostituendo l'input **0** a **p(i)** con un 1 e ricomponendo l'insieme dei messaggi di **p(i)** uno per volta. E' abbastanza immediato mostrare che la **α^i(j)** e le **β^i(j)** sono simili tra lo e che **α^i(n-1)≈β^i(0).**

Il caso generale **(f>1)** è basato sulla stessa tecnica. Dobbiamo mostrare che: Se **n≥3** non esiste un algoritmo che risolve i problema del consenso:

- Con meno di **f+1** round.
- In presenza di **f** fault di tipo **crash.**

Occorre mostrare che di nuovo che **αi≈ αi+1**

Definiamo **crash(α, p i ,r)** come una esecuzione del tutto identifica ad **α**, ma in cui al round **r** il processore **p i** va in crash non inviando alcun messaggio. Mostreremo che:

$$\alpha^i \approx crash(\alpha^i, p_i, 1) \approx crash(\alpha^{i+1}, p_i, 1) \approx \alpha^{i+1}$$

La parte centrale è immediata perché le esecuzioni differiscono per l'input ad un processore **p i** che non invia alcun messaggio in entrambe. Per mostrare le altre similarità occorre il seguente lemma:

Lezione 5, Consenso con comportamento Bizantino:

Ogni processore **p(i)** (0≤i≤n-1) in un Sistema Distribuito ha un input **x(i)** ed un output **y(i)**. Un sottoinsieme dei processori può essere in errore (**faulty**)**.**

Una soluzione a questo problema deve garantire:
- **Terminazione:** In ogni esecuzione ammissibile e per tutti i processori non faulty **p(i)**, ad **y(i)** (output) viene assegnato un valore;
- **Accordo:** In ogni esecuzione se per una coppia di processori non faulty **p(i)** e **p(j)** ad **y(i)** sono stati assegnati valori, allora **y(i)=y(j)**;
- **Validità (debole):** In ogni esecuzione, se per ogni **x(i)=v** e se, per qualche processore non faulty **p(i)**, ad **y(i)** è stato assegnato un valore, allora **y(i)=v.** In altre parole, se tutti gli input sono uguali, l'output deve essere esattamente quell'input. Serve ad evitare di accettare algoritmi che di fatto non sono algoritmi di consenso.

Modello Message Passing utilizzato:

Si effettuano le seguenti assunzioni:
- Si assume che il grafo sia **completo;**
- Si assume che il sistema sia **sincrono;**
- Si assume che siano possibili errori di tipo **bizantino;**

Errori di tipo bizantino:

Un **processore** ha un errore (fault) di tipo **bizantino** se il nuovo stato del processore in un passo di computazione ed il contenuto dei messaggi da spedire sono assolutamente senza restrizioni (il processore in sostanza fa quello che vuole, ed ha anche sufficiente potenza di calcolo per farlo).

Un **sistema** è f-resilient se assicura un corretta esecuzione quando al più **f** processori falliscono. In particolare un esecuzione in un sistema di questo tipo ha le seguenti proprietà:
- Esiste un sottoinsieme **F** di processori con al più **f** processori in fault.
- L'insieme **F** cambia ad ogni esecuzione.
- Ciascun round è composta da esattamente un passo di computazione per tutti i processori (siano essi in **F** oppure no). I messaggi spediti sono trasmessi nello stesso round.

Osservazioni:

Un processore **faulty** può mandare **messaggi diversi** a processori diversi quando dovrebbe inviare lo stesso messaggio a tutti.

Un processore **faulty** può simulare il comportamento di un **crash** (normale) e il messaggio può essere **alterato** in modo plausibile. Pertanto un processore non faulty può **non accorgersi**, all'atto della ricezione, che il messaggio è falso.

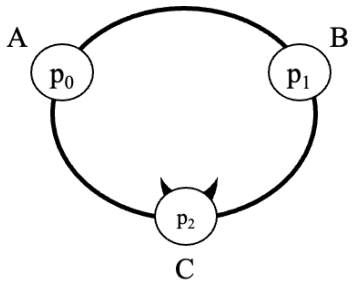
Primo lower bound:

S=2 ed **F=1:** Nel caso crash i nodi scompaiono, nel caso bizantino invece non c'è una descrizione del tipo di fault quindi se un nodo lavora correttamente e l'altro può fare quello che vuole, un accordo non si raggiungerà mai.

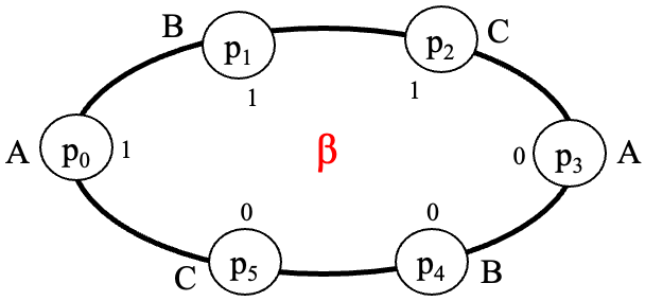
S=3 ed **F=1:** Ho un processore che può fare tutto, ed altri due nodi che seguono un algoritmo, tre nodi formano un anello i cui input possono essere alterati dal bizantino, quindi ancora non funziona.

Teorema: In un sistema distribuito, con **tre processori**, con al più **un fault** di tipo **bizantino** e di tipo **message passing sincrono** il consenso è impossibile.

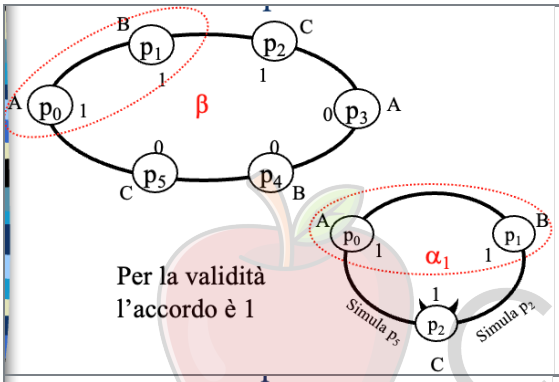
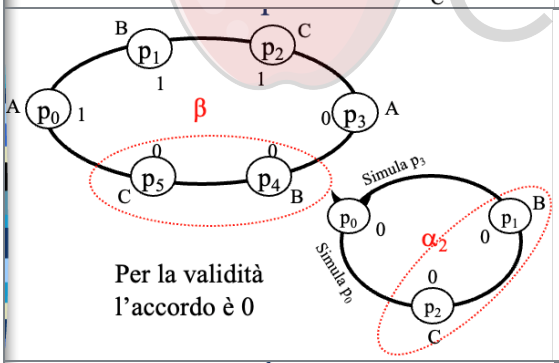
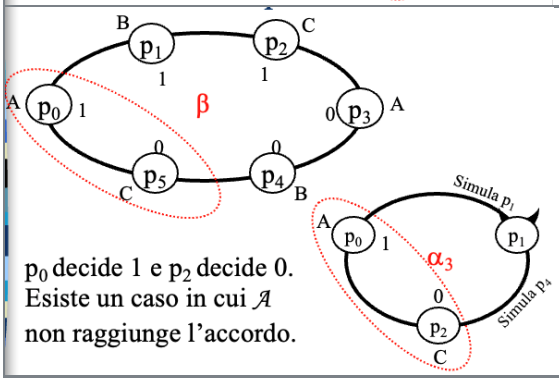
Prova: supponiamo che esista un algoritmo per il consenso **A=(A,B,C) (**composto da tre parti, dove ogni processore ne esegue una**)** per **tre processori** e **un fault** di tipo bizantino.



Il nodo bizantino può fare quello che vuole, anche inviare messaggi diversi a nodi diversi. Per mostrane l'esecuzione consideriamo un anello con **6 processori** in cui le componenti di **A** sono elaborate nel seguente modo e con i seguenti input:



Distinguiamo i seguenti casi:

	Simuliamo l'esecuzione **β**, su **p(0)**, **p(1)**, **p(2)** su input binari 1 e gli altri su 0. Nell'esecuzione **α(1)** il processore **p(2)** simula il comportamento di **p(5)** e **p(2)**. Se esistesse un algoritmo, per la validità l'accordo deve essere su **1** perché gli input sono tutti uguali.
	Simuliamo l'esecuzione **β**, su **p(0)**, **p(1)**, **p(2)** su input binari 1 e gli altri su 0. Nell'esecuzione **α(2)** il processore **p(0)** simula il comportamento di **p(3)** e **p(0)**. Se esistesse un algoritmo, per la validità l'accordo deve essere su **0** perché gli input sono tutti uguali.
	Simuliamo l'esecuzione **β**, su **p(0)**, **p(1)**, **p(2)** su input binari 1 e gli altri su 0. Nell'esecuzione **α(3)** il processore **p(1)** simula **p(1)** verso **p(0)** e simula **p(4)** verso **p(2)**, **p(0)** riceveva **1** come input, mentre **p(2)** riceveva **0**, ora, per le simulazioni fatte da **p(1)**, **p(0)** crede che ci sia una maggioranza di **1** e **p(2)** che ci sia una maggioranza di **0**, quindi è impossibile raggiungere l'accordo.

Vale quindi il seguente teorema**:**

Teorema: In un sistema distribuito con **n** processori e al più **f** fault di tipo bizantino, non esiste un algoritmo che risolve il problema del consenso con **n ≤ 3f**.

Prova: Supponiamo esista un algoritmo che raggiunga il consenso in un sistema distribuito con **n** processore ed **f** fault con **n ≤ 3f.** Partizioniamo i processori in **3** insiemi **P(0)**, **P(1)** e **P(2)** ognuno dei quali contiene al più **n/3** processori.

Consideriamo il sistema composto da 3 processori **p(0)**, **p(1)** e **p(2)** in cui il processore **p(i)(i=0, 1, 2)** simula tutti i processori nell'insieme **P(i).**

Se uno dei tre processori del sistema che simula è faulty, al più **f** processori sono **faulty** nel sistema simulato. Poichè per ipotesi l'algoritmo nel sistema simulato assicura l'accordo se al più **f** processori sono **faulty**, lo stesso deve accadere anche al sistema che simula.

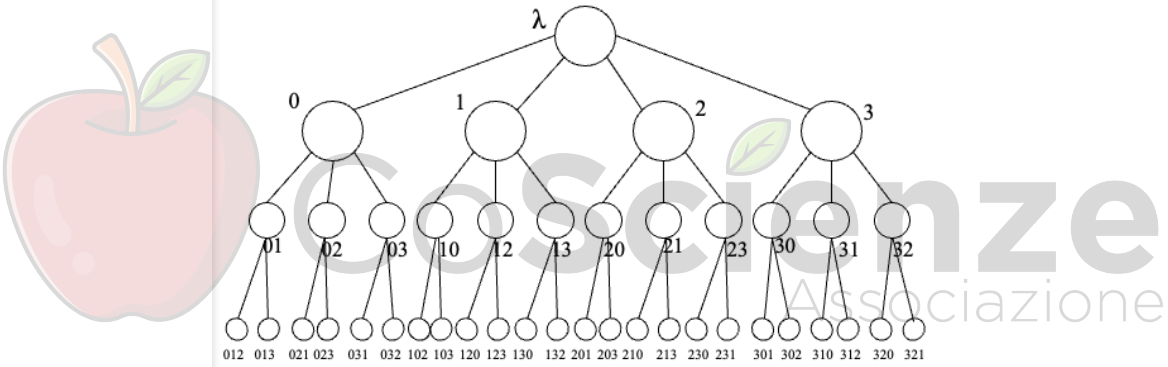
Abbiamo quindi derivato un algoritmo per il problema dell'accordo in un sistema con 3 processori e con al più 1 **fault** di tipo bizantino, ma abbiamo appena dimostrato che tale algoritmo non esiste.

Poichè un processore con fault di tipo bizantino può simulare un fault di tipo crash possiamo affermare che: **f+1** round sono necessari per raggiungere l'accordo in un sistema distribuito message passing con **n** processori ed al più **f** fault di tipo bizantino.

Primo algoritmo:

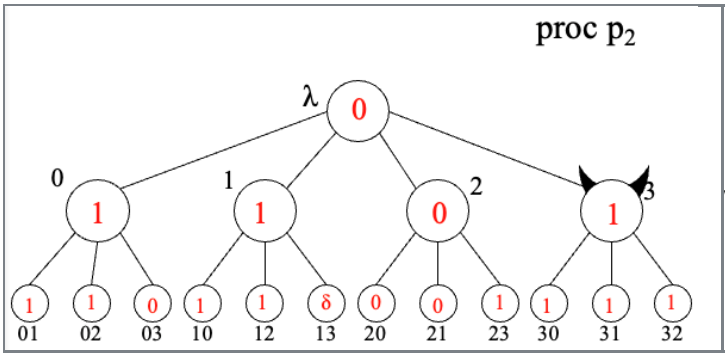
Ciascun processore mantiene localmente una struttura dati ad albero. I nodi dell'albero hanno come etichette sequenze di indici di processori senza ripetizione.

- La radice ha etichetta vuota **λ.**
- Un nodo a livello **d** con etichetta **v** ha **n-d** figli con etichette da **v0** a **v(n-1)**, saltando tutte quelle con ripetizione.
- Le foglie sono al livello **f+1**.



Tutti inizialmente scrivono l'input nella radice:

- **Round 1:** tutti i nodi inviano il contenuto della radice. Il nodo **p(i)** memorizza il valore ricevuto da **p(j)** nel nodo **j**. Scrive δ se non riceve nulla.
- **Round t(t ≤ f+1):** tutti i nodi inviano al livello **t-1** dell'albero (grande scambio di messaggi). Il nodo **p(i)** memorizza il valore (ciò che **p(j)** comunica riguardo il nodo **π**) ricevuto da **p(j)** riguardante il nodo **π** nel nodo di etichetta **πj**. Scrive δ in **πj** se non riceve nulla.

	**p(2)** ha input **0** e carica il valori ricevuti dagli altri processori nel primo livello. Al round successivo, gli **n** valori di ogni nodo sono spediti a tutti gli altri, quindi **p(2)** riceve da **p(1)** il suo primo livello.

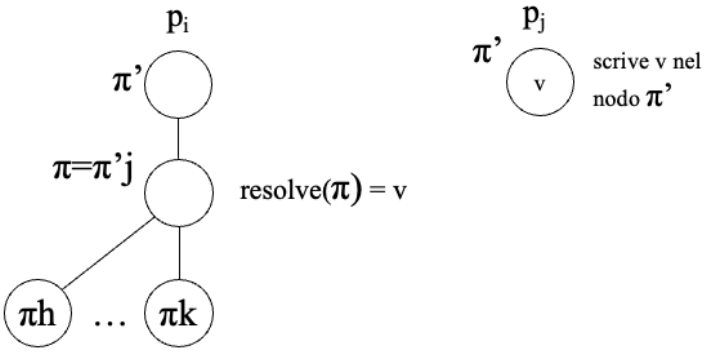
Al termine ogni processore usa l'albero per calcolare il valore di decisione **resolve(λ):**

$$\textbf{resolve}(\pi)=\begin{cases}\text{valore contenuto in } \pi \text{ se } \pi \text{ è una foglia}\\ \text{majority}\{\text{resolve}(\pi_*): \pi_* \text{ sono i figli di } \pi\};\\ \delta \text{ se la majority non esiste}\end{cases}$$

Per dimostrare che questo algoritmo è corretto dimostreremo il seguente lemma:

Lemma: il valore calcolato da **p(i)**, non faulty, per il nodo **π = π'j (**i.e. **resolve(π))** è uguale al valore **p(j),** non faulty, ha memorizzato nel nodo **π'.**

Ovvero questa situazione:



PROVA PER INDUZIONE DA SCRIVERE

Validità: Se applichiamo il precedente lemma i nodi del livello 1 dell'albero possiamo affermare che: Per ciascun processore non faulty **p(j) resolve(j)** è il valore memorizzato alla radice dell'albero **p(j)**, che è l'input a **p(j)**, **resolve(λ)** è il valore di maggioranza tra tutti i **resolve(j)** Se gli input ai processori sono tutti identici, **resolve(λ)** non può che esser il valore input.

Accordo: Occore provare che tutti i processori non faulty decidono lo stesso valore.

Definizione: un nodo dell'albero si dice **common** se tutti i processori non faulty calcolano lo stesso valore per esso.

Prova: DA SCRIVERE

Secondo algoritmo (Algoritmo del King):

Sono sufficienti messaggi di dimensione polinomiale, ma ne il numero di processori non **faulty (n > 4f),** ne il numero di **round (2(f+1))** risulta ottimale.

Lezione 6, Impossibilità di consenso asincrono ed algoritmo Paxos:

E' possibile ereditare risultati in un modello partendo da risultati in un altro modello, in particolare tra il modello **message passing** e il modello **shared memory**, in particolare:

- L'operazione di **scrivere nella variabile x(j)** è equivalente a una **send(m, p(j))**;
- L'operazione di **leggere il contenuto della variabile x(j)** è equivalente a una **receive(m,p(j))**;

Un esecuzione ammissibile per il modello **shared memory** con fault richiede che tutti tranne al più **f** processori eseguano un numero infinito di passi.

Un esecuzione ammissibile per il modello **message passing** con fault richiede che tutti i messaggi spediti debbano essere prima o poi trasmessi.

Prova di impossibilità:

E' stato dimostrata l'impossibilità del consenso in un sistema distribuito anche con un un solo fault.

La dimostrazione di questo passa per la dimostrazione del seguente teorema:

Teorema: Non esiste un algoritmo per il consenso asincrono per il modello **shared memory** se sono ammessi **n-1** fault.

Questo risultato per **n=2** afferma che non esiste un algoritmo per il consenso con due processori e un fault.

Invece per dimostrare che non esiste un algoritmo per il consenso asincrono con **n** processori per il modello **shared memory** se è ammesso **1 fault**, si utilizza l'ipotetico algoritmo per il consenso con **n** processori ed **1 fault** per progettare un algoritmo per **2** processori ed **1 fault.**

Una volta ricavata questo risultato per la shared memory, lo trasformiamo in message passing. Visto che sono equivalenti.

Prova di impossibilità: Wait-Free

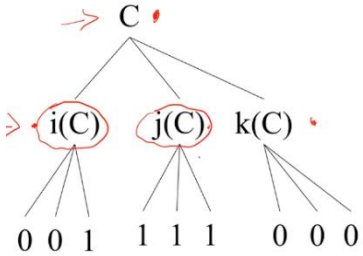
Dovrebbe essere una parte dello schema di dimostrazione del teorema sopra dovrebbe…

Un algoritmo si dice **wait-free** se riesce a tollerare **n-1** fallimenti, ovvero che non si ferma.

Teorema: Non esiste alcun algoritmo **wait-free** per il consenso in un sistema **SM** con **n** processori che utilizzano registri di tipo **R/W**.

Prova: assumiamo che il consenso sia su valori {0,1}. Definiamo un configurazione **bivalente** se entrambe le decisioni sono ancora raggiungibili (non sono ancora state prese decisioni), mentre è **univalente** se una sola delle decisioni è raggiungibile (se il valore di accordo è stato deciso ma l'algoritmo ancora non è finito).

Parleremo di configurazione **0-valenti** (già scelto 0) e **1-valenti** (già scelto 1).



Lemma: se **C(1)** e **C(2)** sono **univalenti** e **C(1) ≈ C(2)** allora **C(1)** e **C(2)** hanno stessa valenza.

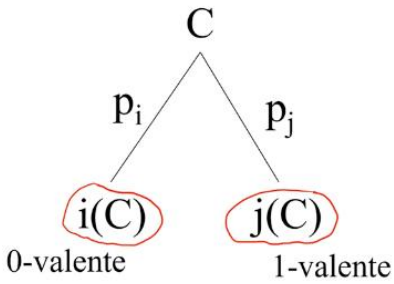
Lemma: Esiste una configurazione iniziale bivalente.

Prova: Se voglio una soluzione al problema del consenso parto da una configurazione dove il consenso non c'è è quindi è bivalente.
Definiamo un processore **p(i) critico** per la configurazione **C**, se un passo **p(i)** in **C** conduce ad una configurazione **univalente**.

Lemma: Se **C** è **bivalente**, allora esiste almeno un processore non critico per **C.**

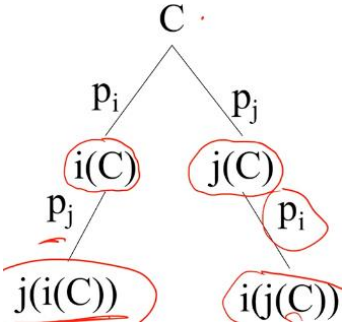
Dimostrando questo si prova che c'è sempre un processore la cui esecuzione non cambia la valenza della configurazione. Questo dimostra che esistono esecuzioni che non terminano e quindi di trovare il valore di accordo.

Prova: supponiamo che tutti i processori siano critici per C. Allora esisteranno **p(i)** e **p(j)** tali che **p(i)** conduce ad una configurazione, **i(C)**, **0-valente** e **p(j)** conduce ad una configurazione, **j(C)**, **1-valente:**



Supponiamo che **p(i)** e **p(j)** accedano a registri condividi diversi, oppure accedano allo stesso registro solo in lettura.

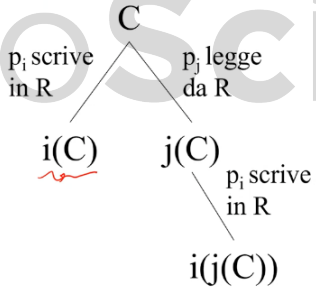
Consideriamo le seguenti esecuzioni:
Per entrambi i processori **j(i(C))** e i**(j(C))** sono indistinguibili, quindi non possono avere valenza diversa.



Arriviamo a delle configurazioni, dove se la modifica avverrebbe in momenti di versi, alla fine le due configurazioni non possono essere diverse e quindi non possono avere valenza diversa.

Supponiamo che **p(i)** e **p(j)** accedano allo stesso registri, oppure e che **p(i)** scriva in un registro **R** e **p(j)** legga dallo stesso registro.

Consideriamo le seguenti esecuzioni:
Per **p(i) i(j(C))** e i**(C)** sono indistinguibili, quindi non possono avere valenza diversa.



Se i due processori sono critici le due configurazione dovrebbero avere valenza diversa ma non accade.

Aver mostrato questa cosa significa che in linea di principio nel caso distribuito asincrono, possono avere un esecuzione che non termina mai. E quindi possiamo dire che l'impossibilità e un realtà un **non sempre possibile**.

Algoritmo Paxos:

Siccome il caso asincrono è il più diffuso, non possiamo pensare che non sia possibile aggiungere un accordo, ci accontentiamo quindi di soluzione non sempre corrette. In sostanza progetto l'algoritmo e utilizzo un timeout che se entro il quale non è stato raggiunto l'accordo rifaccio ripartire l'algoritmo e dopo un tot di volte beccherò un esecuzione ch emi permette di raggiungere l'accordo. E questo nella maggior parte dei casi funziona.

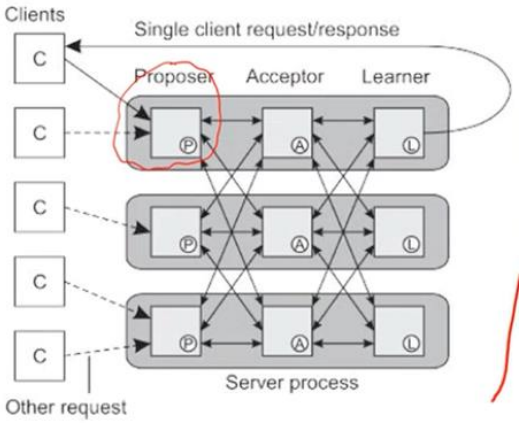
E' un algoritmo basato sulla maggioranza, garantiamo la correttezza quando l'algoritmo si ferma, ma non garantiamo che si ferma.

Funzionamento:

I nodi comunicano inviando messaggi. I messaggi sono **asincroni**, ovvero hanno un tempo di spedizione finito ma non prevedibile, possono essere duplicati e possono essere perduti. Inoltre i nodi possono andare in **crash**.

Un singolo nodo può assumere tre ruoli distinti:

- **Proposer:** propongono di iniziare la ricerca di un valore di accordo **proposal** sulla base di una richiesta che viene dall'esterno.
- **Acceptor:** decidono quale valore scegliere.
- **Learner:** apprendono quale valore è stato scelto e riportano la decisione finale ai richiedenti consenso.



Quando arrivano gli input arrivano ai **proposer** che fanno qualcosa e devono trovare il valore di accordo, per fare questo usano gli **acceptor** che hanno il compito di trovare un valore che sia scelto dalla metà + 1. A questo punto il valore viene trasferito al **learner** e il risultato al **client**.

Quando i **proposer** sono più di uno possono esserci delle complicazioni in quanto gli **acceptor** hanno ache fare con più **proposer**, l'idea è che una maggioranza deve avere per forza **n/2 + 1** nodi quindi **2** maggioranze condividono per forza qualche nodo, la soluzione è che se un **acceptor** ha in precedenza già deciso un valore, praticamente se decide di cambiare **proposer** deve imporre che il successivo valore di accordo sia sempre quello, questo assicura che non mando in crisi quelli che avevano già scelto un valore.

Inoltre quando il **proposer** comincia potrebbe fallire, è necessario quindi riprovare e in questo punto potrebbe non convergere, perché i proposer fanno **ping pong** tra di loro.

Valore scelto:

Quando il learner riceve il valore dalla maggioranza degli **acceptor** quello il momento in cui si sceglie e il client riceve il valore scelto.

Extra:

L'accordo è una proprietà in cui i processo non faulty scelgono il valore questo valore è uguale per tutti, Paxos lo fa tramite la maggioranza.

La terminazione non possiamo assicurarla perché non è un algoritmo corretto.

Paxos con leader:

Ci sono delle versioni di Paxos dove si elegge un leader per migliorare le performance.

Algoritmo Raft:

Modello lo stesso di Paxos, il problema è invece decomposto in problemi indipendenti,