

Mutua Esclusione (1)

- Nei sistemi distribuiti è essenziale che i vari processi cooperino per portare a termine un obiettivo comune.
- Tale **cooperazione** solitamente richiede un **accesso concorrente** ad una **risorsa condivisa** o ad una **sezione critica** (CS).
- Per **evitare di corrompere** la risorsa condivisa, è necessario garantire ai processi un **accesso mutuamente esclusivo**.
- In un sistema distribuito le variabili condivise (semafori) oppure un kernel locale non possono essere usati per implementare la mutua esclusione.
- Lo **scambio di messaggi** è il solo modo possibile per implementare la mutua esclusione distribuita.
- Ci sono due categorie di algoritmi di mutua esclusione distribuita:
 - soluzioni basate su **token**,
 - soluzioni basate su un sistema di **permessi**.



Mutua Esclusione (2)

- Le soluzioni basate su token godono delle seguenti proprietà:
 - la mutua esclusione è garantita dall'unicità del token,
 - permettono di evitare fenomeni di starvation,
 - i deadlock possono essere facilmente evitati,
 - gli unici problemi possono insorgere nel caso della perdita del token.
- Le soluzioni basate sui sistemi di permessi prevedono che un processo necessiti del permesso da parte di altri processi prima di accedere alla risorsa:
 - gli algoritmi differiscono nella modalità in cui garantiscono tale permesso.



Mutua Esclusione (3)

- **Requisiti degli Algoritmi di Mutua Esclusione**
- **Proprietà di Sicurezza:** ad ogni istante, soltanto un processo può eseguire la sezione critica.
- **Proprietà di Progresso:** questa proprietà stabilisce l'assenza di deadlock e starvation. Due o più entità non devono attendere in modo indefinito dei messaggi che non arriveranno mai.
- **Equità:** ogni processo ha un'equa possibilità di eseguire la sezione critica. Questa proprietà generalmente garantisce che le richieste di esecuzione delle sezioni critiche saranno soddisfatte nell'ordine di arrivo nel sistema (il tempo è determinato da un orologio logico).



Mutua Esclusione

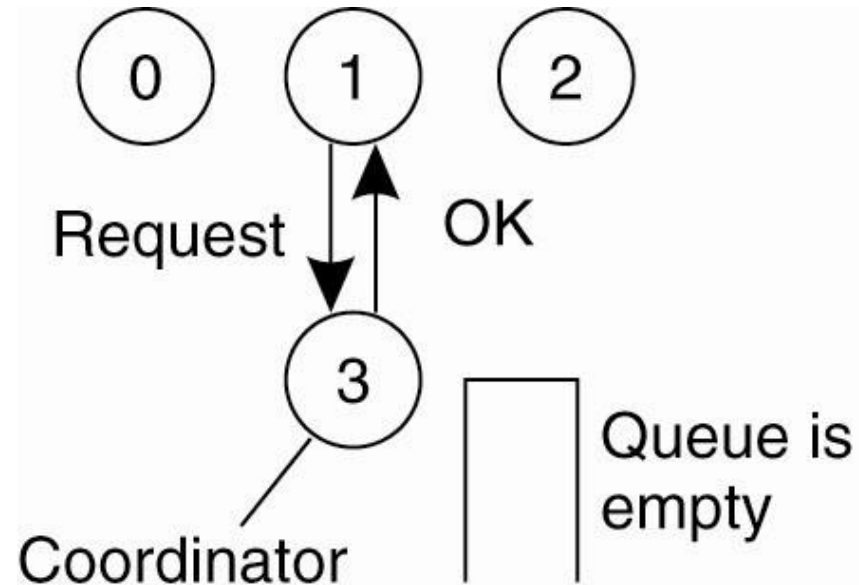
Un algoritmo centralizzato (1)

- Il modo più semplice per garantire la mutua esclusione in un sistema distribuito è imitare come questo obiettivo viene ottenuto in un sistema a processore singolo:
 - un processo viene eletto come **coordinatore**;
 - quando un processo vuole accedere ad una risorsa condivisa, invia una richiesta al coordinatore;
 - se nessun altro processo sta accedendo alla risorsa, il coordinatore assegna il permesso ed il processo richiedente può continuare;
 - altrimenti (se la risorsa è assegnata ad un altro processo), il coordinatore non può concedere l'autorizzazione;
 - il metodo esatto utilizzato per negare l'autorizzazione dipende dal sistema:
 - astensione dal rispondere,
 - rispondere "permesso negato";
 - in ogni caso, la richiesta rifiutata viene accodata: quando la risorsa tornerà libera, la prima richiesta in coda verrà autorizzata.



Mutua Esclusione

Un algoritmo centralizzato (2)



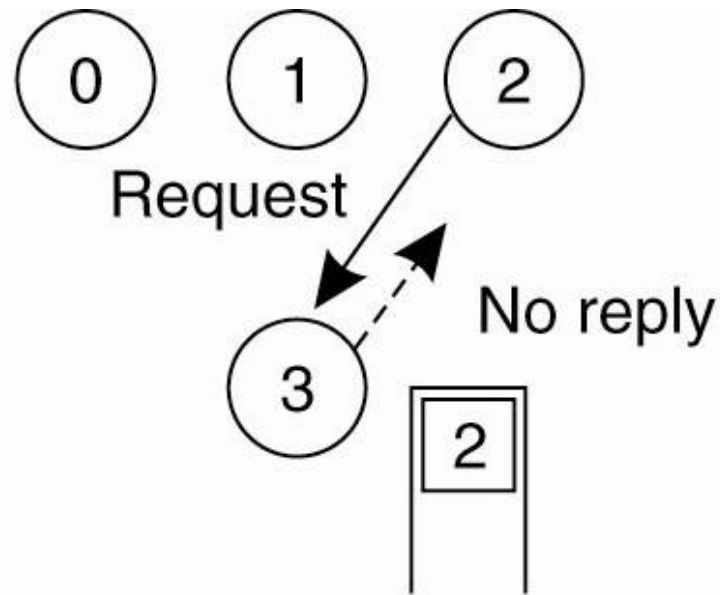
(a)

(a) Il processo 1 chiede al coordinatore il permesso di accedere alla risorsa condivisa. L'autorizzazione viene concessa.



Mutua Esclusione

Un algoritmo centralizzato (3)

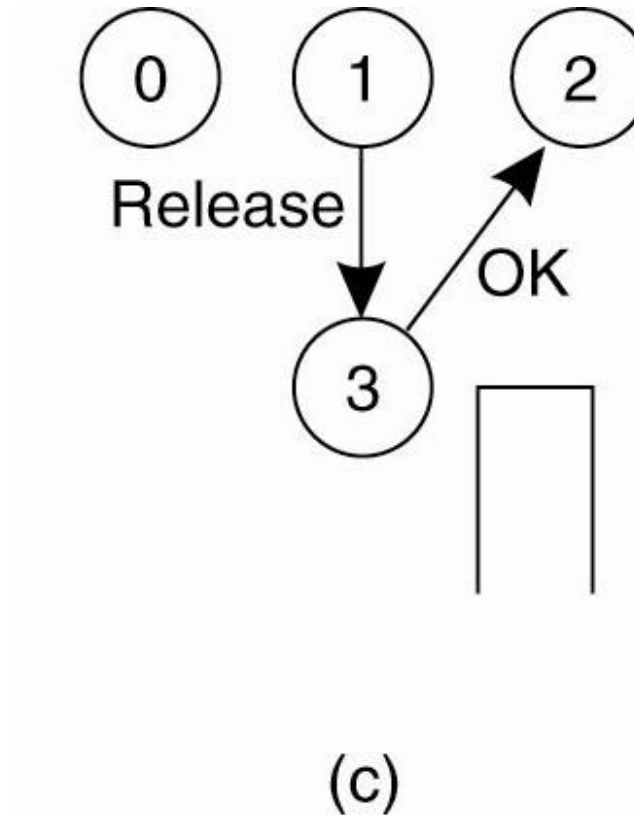


(b)

(b) Il processo 2 chiede a sua volta il permesso di accedere alla stessa risorsa. Il coordinatore non risponde.

Mutua Esclusione

Un algoritmo centralizzato (4)



(c) Quando il processo 1 rilascia la risorsa, lo comunica al coordinatore, che quindi risponde al processo 2.

Mutua Esclusione

Un algoritmo centralizzato (5)

- **Aspetti positivi:**
- L'algoritmo garantisce la mutua esclusione:
 - il coordinatore permette ad un solo processo per volta di accedere alla risorsa.
- L'algoritmo è anche **equo**: le richieste vengono autorizzate nell'ordine in cui vengono ricevute.
- L'algoritmo è facile da implementare.
- **Problemi:**
- Il coordinatore rappresenta un "single point of failure" ed un collo di bottiglia (nei sistemi distribuiti di grandi dimensioni).
- Se i processi si bloccano in seguito ad una richiesta, non sono in grado di distinguere un coordinatore non più in esecuzione (ad esempio, per un crash) da una "autorizzazione negata".



Un algoritmo decentralizzato (1)

- Lin, Lian, Chen e Zhang (2004) hanno ideato un **meccanismo di votazione** utilizzando un sistema DHT (Distributed Hash Table), per risolvere il problema della presenza di un unico coordinatore.
- Ogni risorsa viene **replicata** n volte.
- **Ogni replica** ha il proprio **coordinatore**.
- Un processo, per ottenere l'accesso alla risorsa, necessita di una **maggioranza** di voti pari a $m > n/2$ coordinatori.
- I richiedenti si vedono comunicare dai coordinatori quando la risorsa viene negata.
- **I coordinatori che subiscono un crash** tornano in esecuzione velocemente, ma perdono la memoria di ogni richiesta avvenuta (reset).
- **Rischio:** un coordinatore in crash, dopo la fase di recupero, potrebbe assegnare nuovamente in modo scorretto un permesso.



Un algoritmo decentralizzato (2)

- Sia $p=\Delta t/T$ la probabilità che un coordinatore si resettì nell'intervallo di tempo Δt durante il suo ciclo di vita T .
- Allora, la probabilità $P[k]$ che k su m coordinatori si resettino nello stesso intervallo Δt è la seguente:
$$P[k] = \binom{m}{k} p^k (1-p)^{m-k}$$
- Almeno **$2m-n$** coordinatori devono resettarsi per violare la correttezza del meccanismo di votazione.
- Quindi, la probabilità di tale evento è $\sum_{k=2m-n}^n P[k]$.
- Ad esempio, in un sistema in cui ogni nodo ha un ciclo di vita di 3 ore e Δt è pari a 10 secondi, assumendo $n=32$ e $m=0.75n$, la probabilità di violare la correttezza è meno di 10^{-40} .
- **Inconveniente:** se molti nodi vogliono accedere alla stessa risorsa, nessuno sarà in grado di ottenere abbastanza voti, lasciando la risorsa inutilizzata.



Un algoritmo distribuito (1)

- L'algoritmo di Ricart-Agrawala assume che i **canali** di comunicazione siano gestiti con politica **FIFO**. L'algoritmo utilizza due tipi di messaggi: REQUEST e REPLY.
- Un processo invia un messaggio REQUEST a tutti gli altri processi (incluso se stesso dal punto di vista logico) per richiedere il loro permesso ad entrare nella sezione critica. Un processo invia un messaggio REPLY ad un altro processo per concedergli il suo permesso.
- I processi usano degli **orologi logici in Lamport-style** per assegnare un **timestamp** alle richieste di accesso alle sezioni critiche: tali timestamp vengono utilizzati per stabilire la **priorità** delle richieste.
- Ogni processo p_i mantiene il **vettore Request-Deferred**, RD_i , la cui dimensione è la stessa del numero dei processi nel sistema.
- Inizialmente, $\forall i \forall j: RD_i[j]=0$. Ogni volta che p_i ritarda la richiesta di p_j , imposta $RD_i[j]=1$ e, dopo aver inviato un messaggio REPLY a p_j , imposta $RD_i[j]=0$.



Un algoritmo distribuito (2)

- **Richiesta di accesso alla sezione critica:**
 - (a) Quando un nodo S_i vuole accedere alla sezione critica (CS), invia in **broadcast** un messaggio REQUEST con relativo timestamp a tutti gli altri nodi.
 - (b) Quando un nodo S_j riceve un messaggio REQUEST dal nodo S_i , invia un messaggio REPLY al nodo S_i , se S_j non sta né richiedendo né eseguendo la sua CS, oppure se S_j sta effettuando una richiesta ed il timestamp della richiesta di S_i è minore di quello della richiesta di S_j . Altrimenti, la risposta viene ritardata e S_j imposta $RD_j[i]=1$.
- **Esecuzione della sezione critica:**
 - (c) Il nodo S_i entra nella CS dopo aver ricevuto un messaggio REPLY da ogni altro nodo a cui ha inviato un messaggio REQUEST.



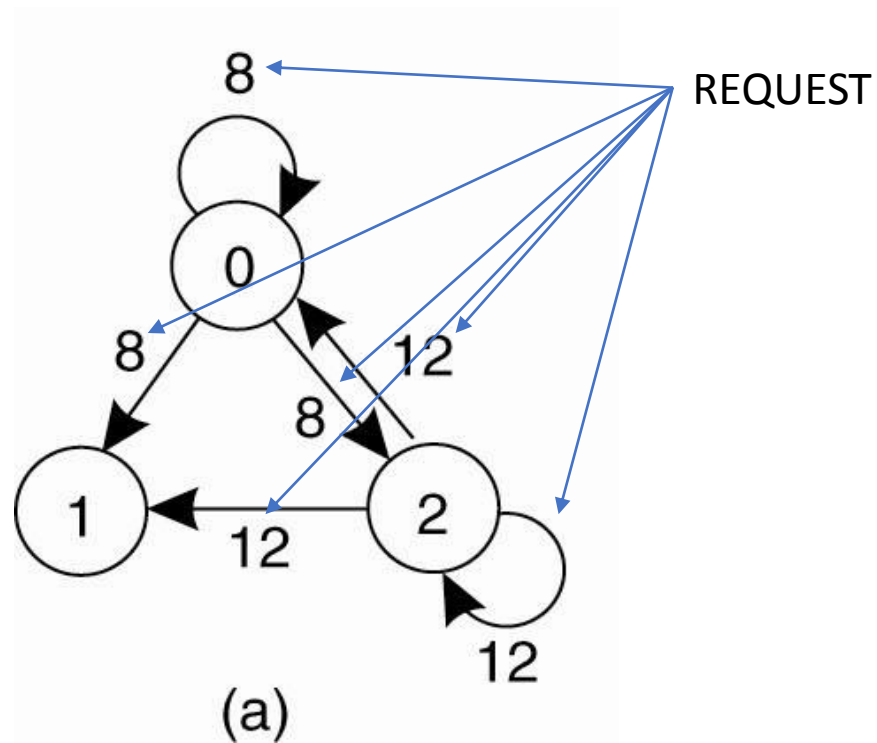
Un algoritmo distribuito (3)

- **Rilascio della sezione critica:**
- (d) Quando un nodo S_i esce dalla CS, invia tutti i messaggi REPLY che aveva lasciato in sospeso: $\forall j$ se $RD_i[j]=1$, allora invia un messaggio REPLY a S_j ed imposta $RD_i[j]=0$.
- **Note:**
- Quando un nodo riceve un messaggio, **aggiorna** il suo orologio, utilizzando il timestamp del messaggio.
- Quando un nodo prende in carico una richiesta di accesso alla CS, **aggiorna** il suo orologio locale ed assegna un timestamp alla richiesta.



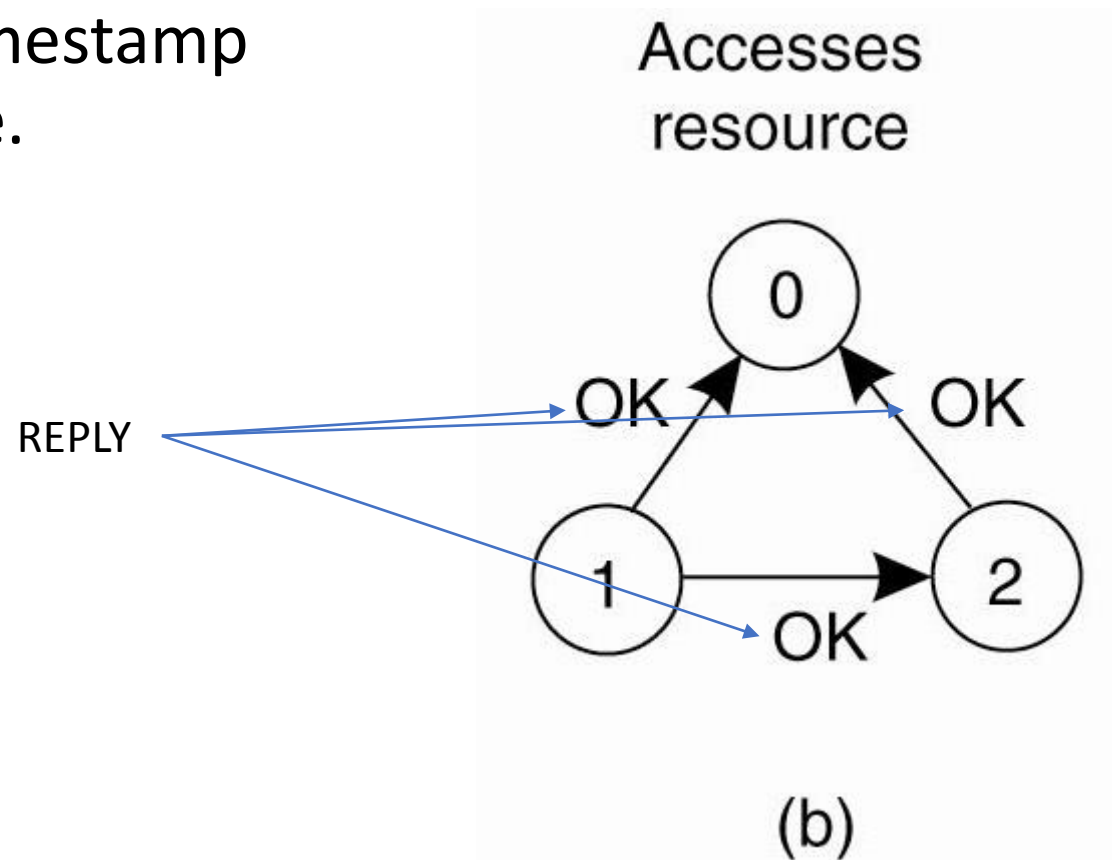
Un algoritmo distribuito (4)

(a) Due processi vogliono accedere una risorsa condivisa nello stesso istante.



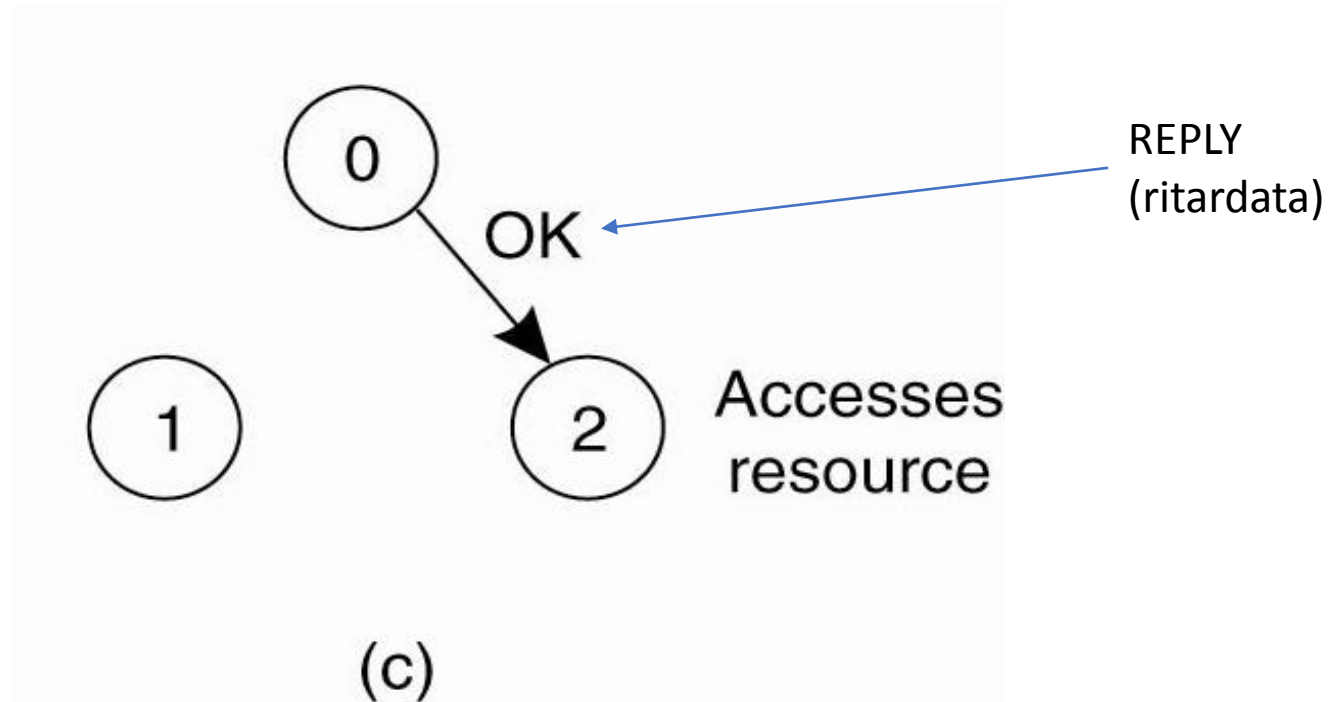
Un algoritmo distribuito (5)

(b) Il processo 0 ha il timestamp più basso e quindi vince.



Un algoritmo distribuito (6)

(c) Quando il processo 0 ha terminato la sua CS, invia un REPLY al processo 2, che ora può procedere.



Un algoritmo distribuito (7)

- **Correttezza**
- **Teorema:** l'algoritmo di Ricart-Agrawala garantisce la mutua esclusione:
- La dimostrazione procede per contraddizione. Siano dati due nodi S_i e S_j che stiano eseguendo la loro CS in modo concorrente e la richiesta di S_i abbia una priorità maggiore di quella di S_j . Quindi S_i ha ricevuto la richiesta di S_j dopo aver effettuato la propria.
- Quindi, S_j può eseguire la CS in modo concorrente con S_i solo se S_i invia un REPLY a S_j (in risposta alla richiesta di S_j) prima che S_i esca dalla CS.
- Tuttavia, ciò è impossibile dato che la richiesta di S_j ha una priorità minore. Quindi l'algoritmo di Ricart-Agrawala garantisce la mutua esclusione.

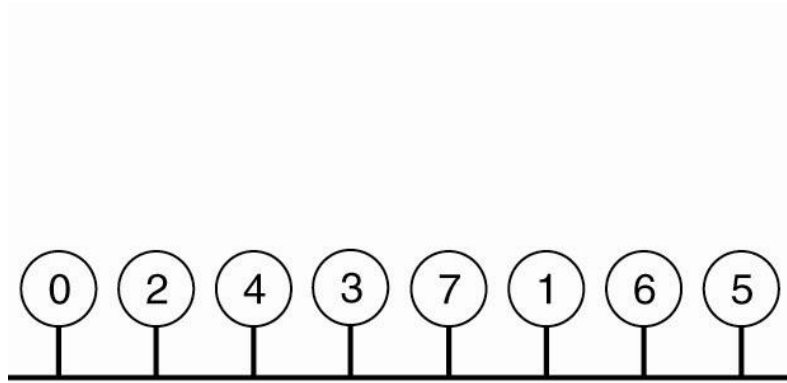


Un algoritmo distribuito (8)

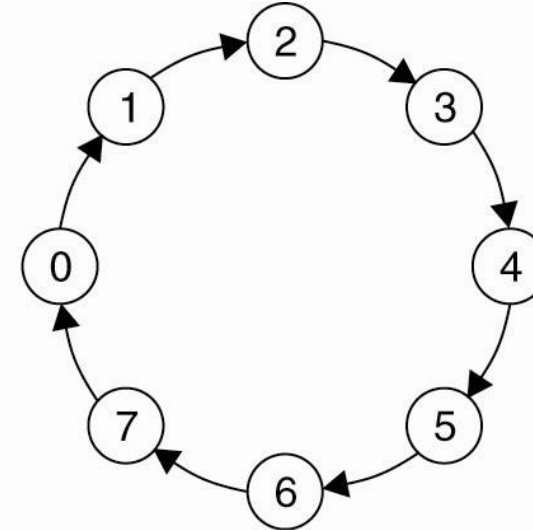
- **Prestazioni**
- Per ogni esecuzione di CS, l'algoritmo di Ricart-Agrawala richiede $(N - 1)$ messaggi di tipo REQUEST e $(N - 1)$ messaggi di tipo REPLY.
- Quindi, necessita di $2(N - 1)$ messaggi per ogni esecuzione di CS.
- Il ritardo per la sincronizzazione nell'algoritmo (i.e., il tempo richiesto prima che il nodo entri nella CS, dopo che un altro nodo ne è uscito) è il ritardo medio dei messaggi.



Un algoritmo di tipo Token Ring (1)



(a)



(b)

(a) Un gruppo non ordinato di processi in rete.

(b) Un anello logico implementato via software.

Un algoritmo di tipo Token Ring (2)

- I processi vengono **ordinati** in un anello logico.
- In seguito all'inizializzazione, viene assegnato un **token** al processo 0; tale token viaggia lungo l'anello (dal processo k al processo $k+1$ modulo la dimensione dell'anello).
- Quando un processo acquisisce il token, controlla se necessita di accedere alla risorsa condivisa:
 - in caso positivo, il processo continua ad eseguire il proprio task, accedendo alla risorsa, rilasciandola e passando il token lungo l'anello;
 - altrimenti, passa il token lungo l'anello.
- In tal modo, soltanto un processo per volta può accedere alla risorsa.
- Non si possono verificare fenomeni di starvation.
- Aspetti negativi:
 - recuperare un **token perso** è molto difficile,
 - tutti i processi devono mantenere l'informazione sulla **configurazione corrente dell'anello**, per evitare problemi nel caso del crash di uno o più processi.



Un confronto fra i quattro algoritmi

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	$3mk, k = 1, 2, \dots$	$2m$	Starvation, low efficiency
Distributed	$2(n - 1)$	$2(n - 1)$	Crash of any process
Token ring	1 to ∞	0 to $n - 1$	Lost token, process crash



Algoritmi di Elezione

- Molti algoritmi distribuiti richiedono che un processo assuma un **ruolo speciale** (e.g., il coordinatore).
- I processi devono essere distinguibili per mezzo di una caratteristica come un **numero univoco** (e.g., l'indirizzo di rete, assumendo un processo per macchina) affinché gli algoritmi funzionino.
- In generale, gli algoritmi di elezione tentano di **individuare** il processo dotato del **numero più alto**.
- Gli algoritmi differiscono per il modo in cui individuano il processo.
- Ogni processo è a conoscenza del numero di processo di ogni altro processo del sistema.
- L'obiettivo di un algoritmo di elezione è far sì che, alla fine, **tutti** i processi **concordino** su chi sia il coordinatore.



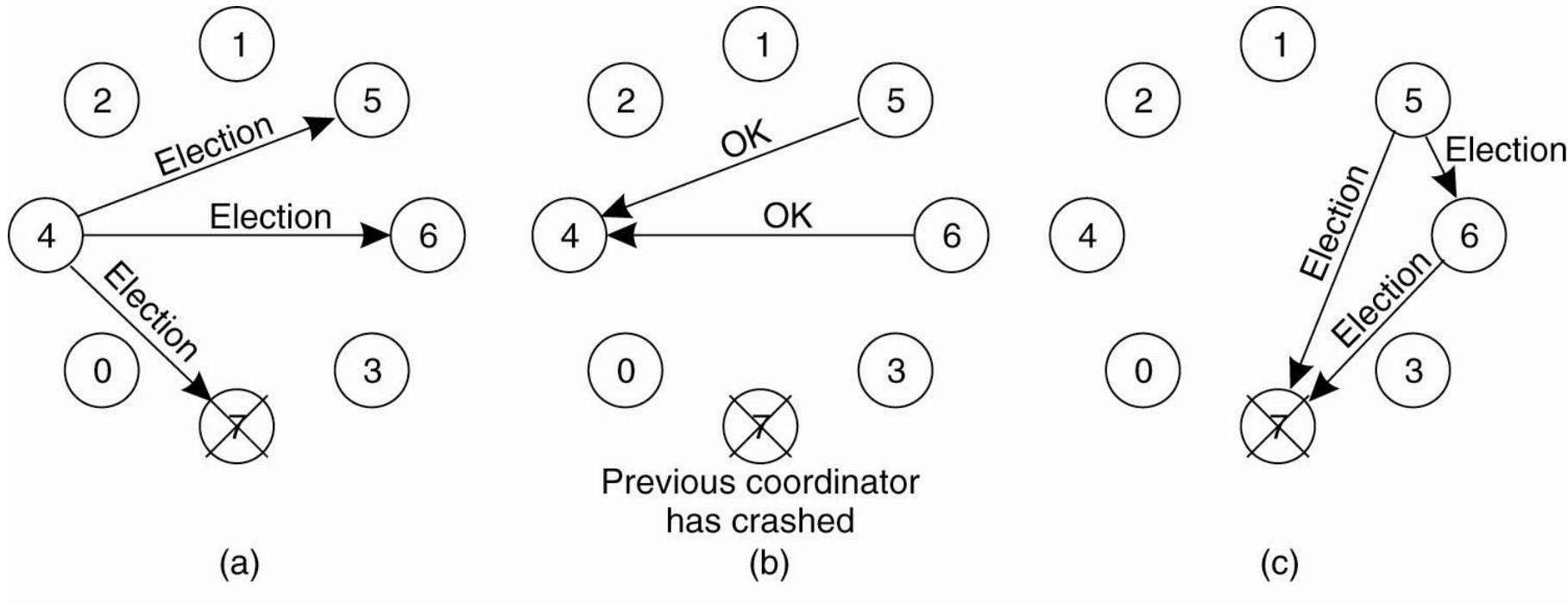
L'algoritmo del prepotente (1)

(Bully Algorithm)

- Algoritmo del "prepotente" (Garcia-Molina, 1982): quando un qualunque processo P si accorge che il coordinatore corrente non risponde più, può indire un'elezione:
 1. P invia un messaggio di tipo *ELECTION* a tutti i processi con i numeri più alti del suo.
 2. Se nessuno risponde, P vince l'elezione e diventa il nuovo coordinatore.
 3. Se uno dei processi con identità maggiore risponde, prende il ruolo di P , terminando il lavoro di quest'ultimo.



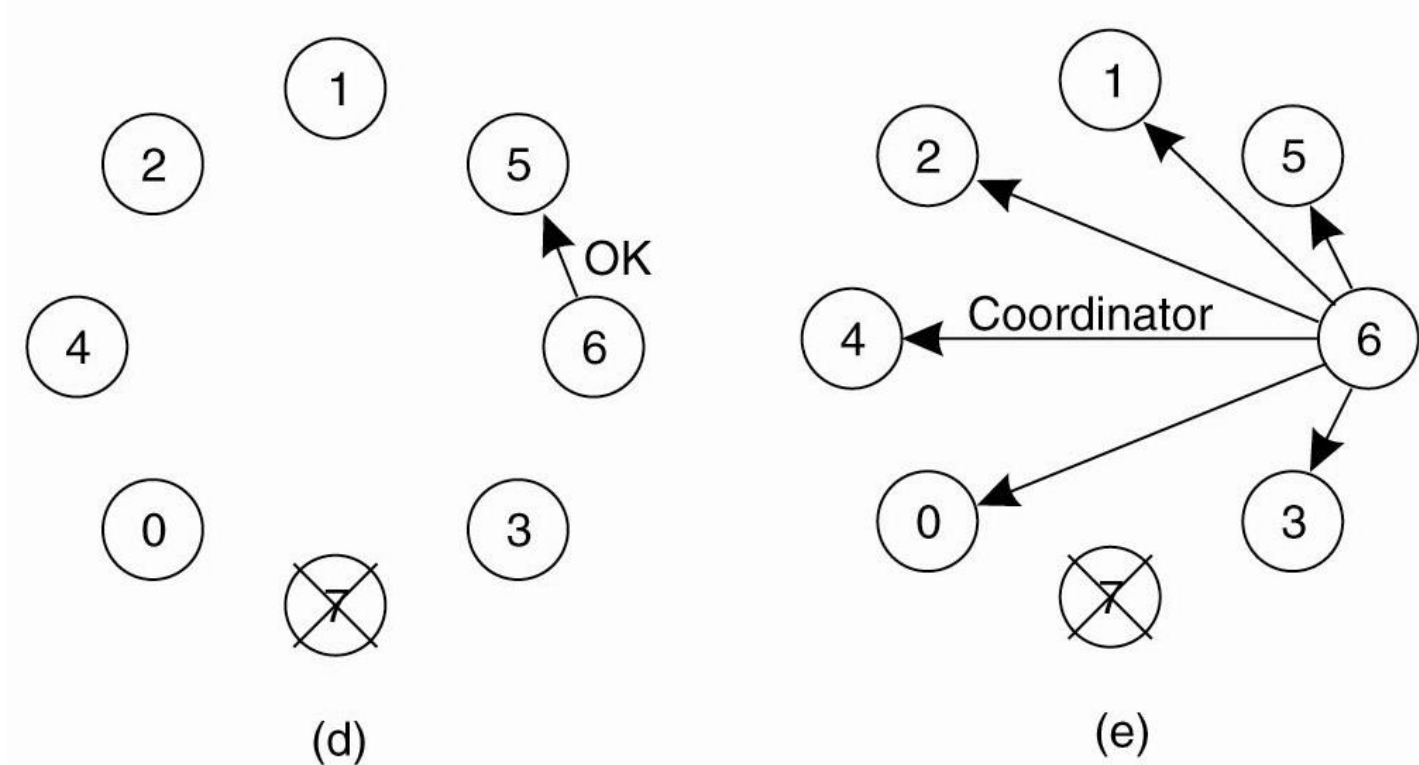
L'algoritmo del prepotente (2) (Bully Algorithm)



- L'algoritmo del prepotente. (a) Il processo 4 indice un'elezione. (b) I processi 5 e 6 rispondono, intimando a 4 di fermarsi.
- A questo punto sia 5 che 6 indicano una propria elezione.

L'algoritmo del prepotente (3) (Bully Algorithm)

- (d) L'algoritmo del prepotente. (d) Il processo 6 intima a 5 di fermarsi.
(e) Il processo 6 vince le elezioni e lo comunica a tutti.



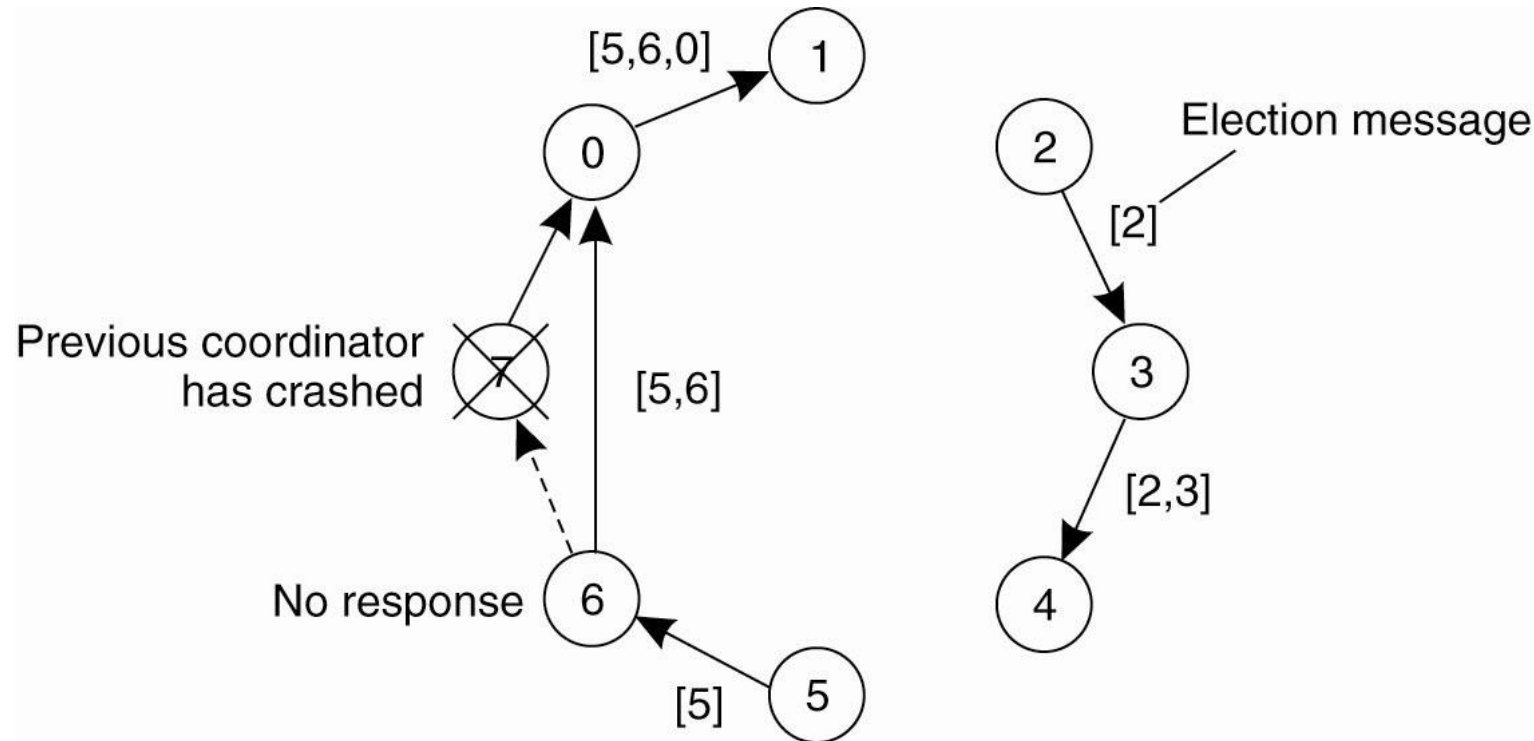
Un Algoritmo ad anello (1)

- Assumiamo che tutti i processi siano (fisicamente o logicamente) ordinati.
- Nel momento in cui un qualunque processo si accorga che il coordinatore non è più funzionante:
 - invia un messaggio di tipo ELECTION con il proprio numero di processo in allegato;
 - il messaggio viene inviato al suo successore nell'anello;
 - se il successore non è online, viene saltato ed il messaggio viene inviato al nodo successivo (finché non viene trovato un processo funzionante);
 - ogni processo attivo inoltra il messaggio, aggiungendo il proprio numero identificativo;
 - alla fine il messaggio ritornerà al mittente originale;
 - a tal punto il tipo del messaggio diventa COORDINATOR e viene fatto circolare nuovamente;
 - tale messaggio informerà tutti i processi su chi sia il nuovo coordinatore (il membro della lista con il numero più alto) e su quali siano i membri della nuova configurazione dell'anello.



Un Algoritmo ad anello (2)

Algoritmo di elezione usando un anello.



Il processo di Elezione in Ambienti Wireless (1)

- Gli algoritmi di elezione tradizionali non funzionano bene negli ambienti wireless, dato che assumono quanto segue:
 - lo scambio di messaggi è affidabile,
 - La topologia della rete non cambia.
- Nelle MANET (Mobile Ad-hoc NETwork) tali assunzioni sono generalmente false.
- Vasudevan et al. (2004) propongono una soluzione che superi i problemi relativi al crash dei nodi ed al partizionamento della rete.
- Un'utile proprietà del loro algoritmo è che alla fine viene eletto il miglior nodo invece di un elemento a caso.

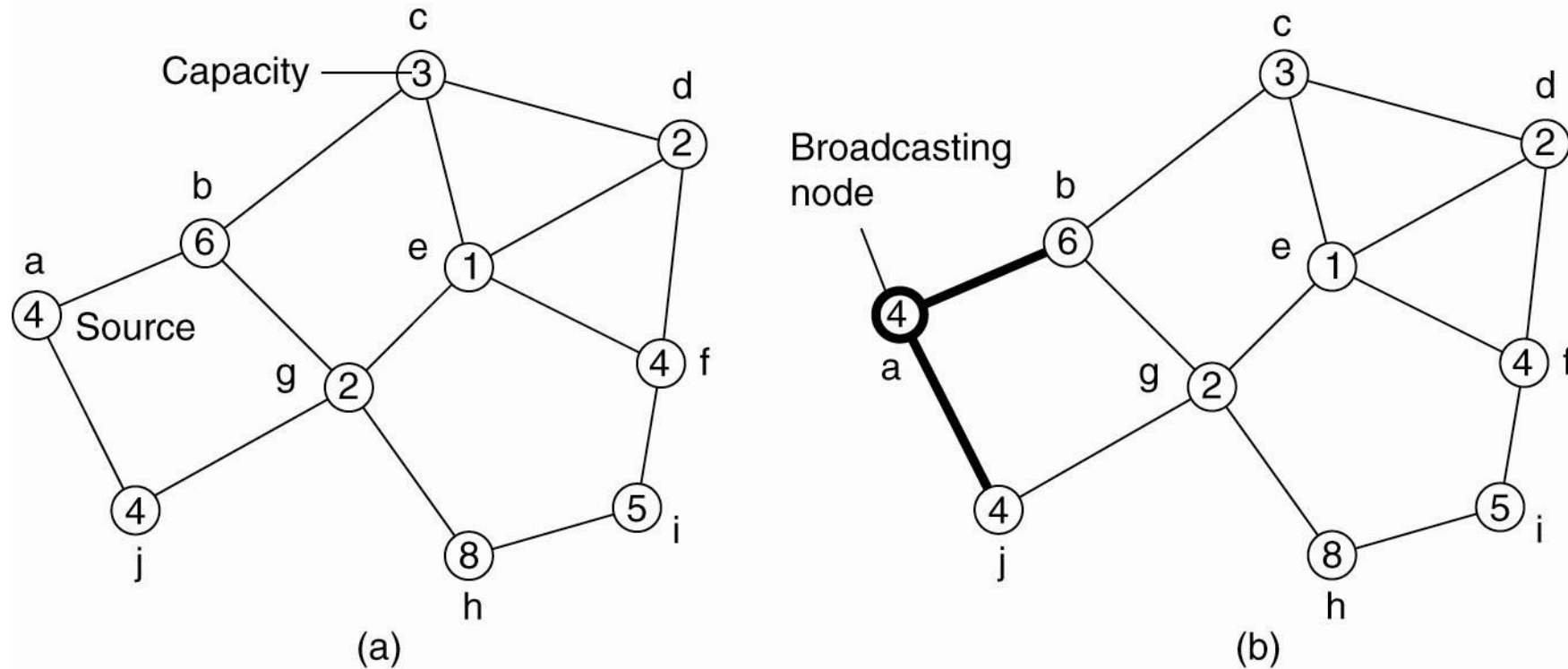


Il processo di Elezione in Ambienti Wireless (2)

- Ogni nodo della rete (denominato sorgente) può iniziare un processo di elezione inviando un messaggio di tipo ELECTION ai suoi vicini.
- Un nodo che riceve un messaggio di tipo ELECTION per la prima volta memorizza il mittente come proprio padre ed inoltra il messaggio ai propri vicini.
- Quando viene ricevuto un messaggio di tipo ELECTION da un nodo diverso dal padre, il processo comunica il riscontro senza fare altro.
- Quando R (con genitore Q) inoltra un messaggio di tipo ELECTION, esso attende un messaggio di riscontro, prima di riconoscere a sua volta a Q la ricezione.
- Nel messaggio di riscontro al genitore, vengono inviate informazioni come la capacità residua della batteria ed altri dati sulle risorse possedute.
- Tali informazioni permetteranno in seguito al nodo sorgente di selezionare il **miglior** nodo.



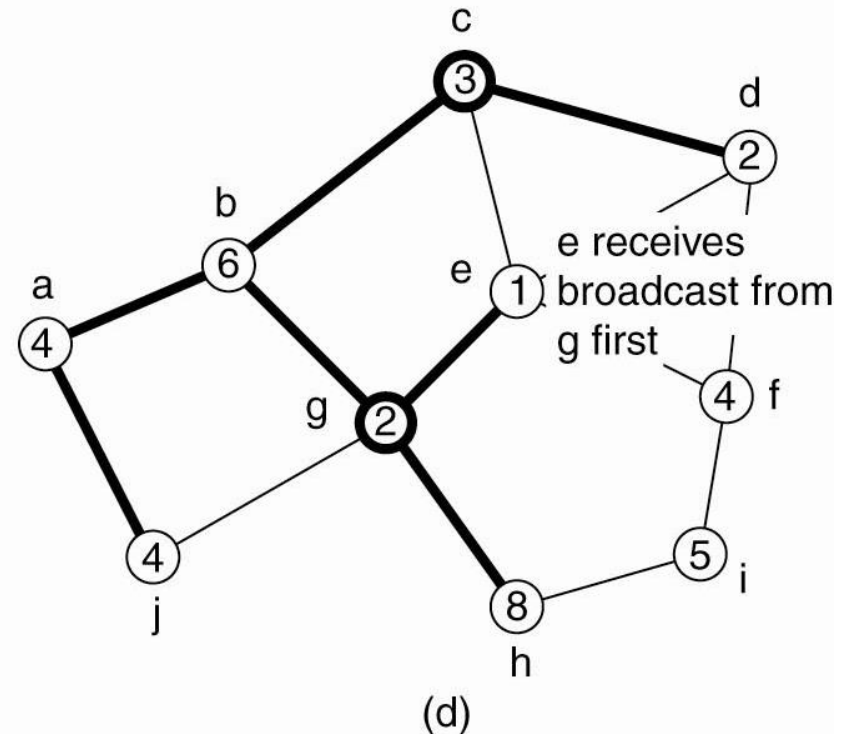
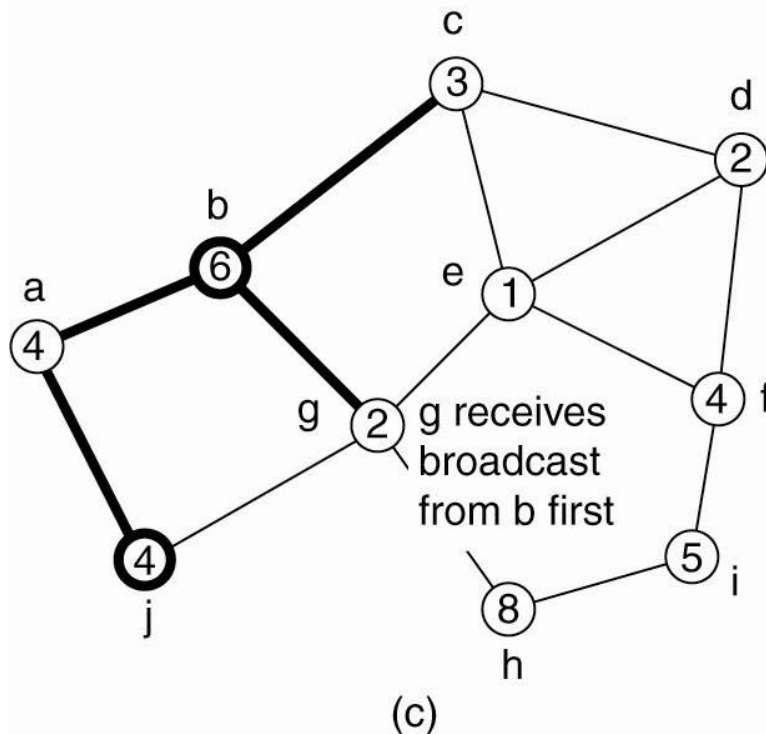
Il processo di Elezione in Ambienti Wireless (3)



Algoritmo di elezione in una rete wireless, con il nodo a come sorgente. (a) Rete iniziale. (b)–(e) Le fasi di costruzione dell'albero.

Il processo di Elezione in Ambienti Wireless (4)

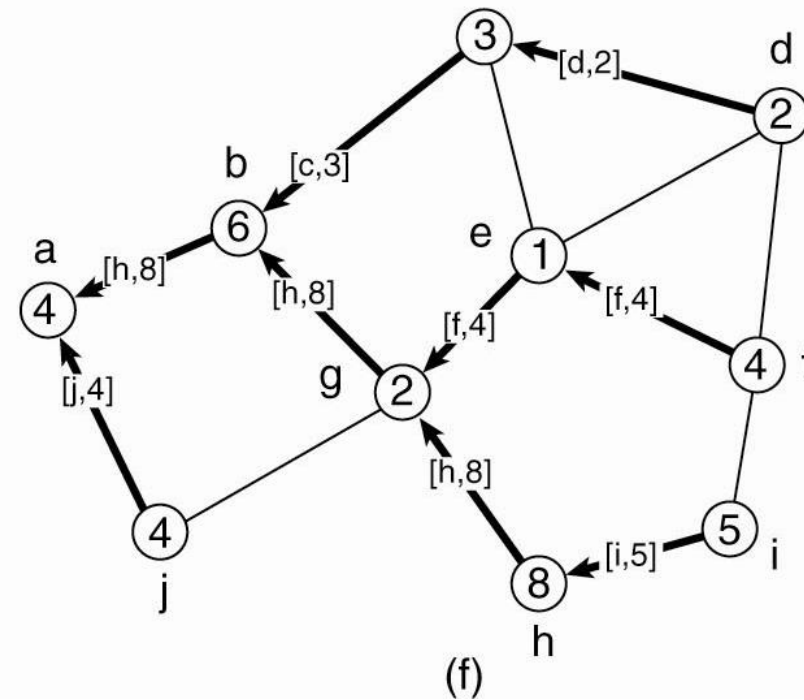
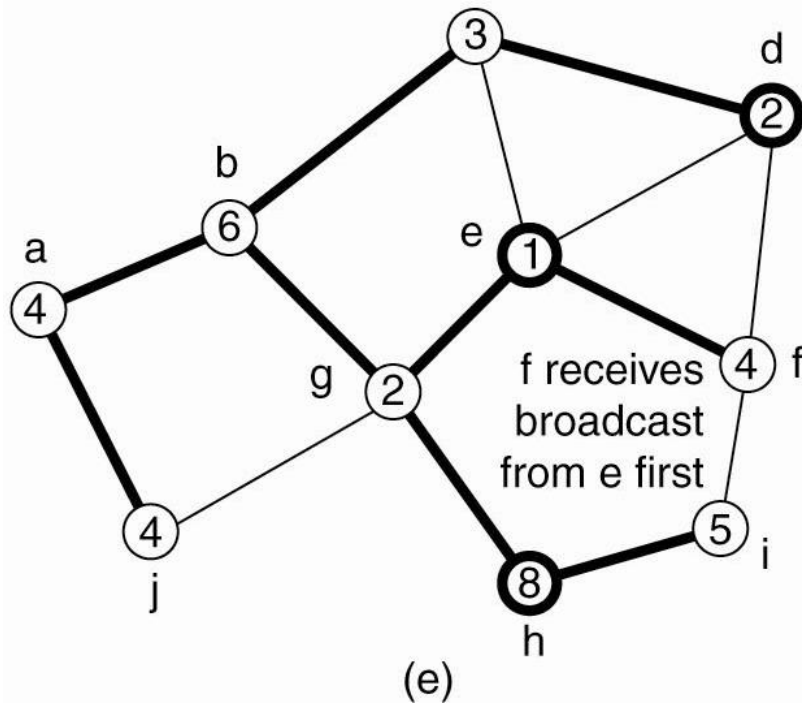
Algoritmo di elezione in una rete wireless, con il nodo a come sorgente.
(a) Rete iniziale. (b)–(e) Le fasi di costruzione dell'albero.



Il processo di Elezione in Ambienti Wireless (5)

(e) Fase di costruzione dell'albero.

(f) Comunicazione del nodo migliore al nodo sorgente.

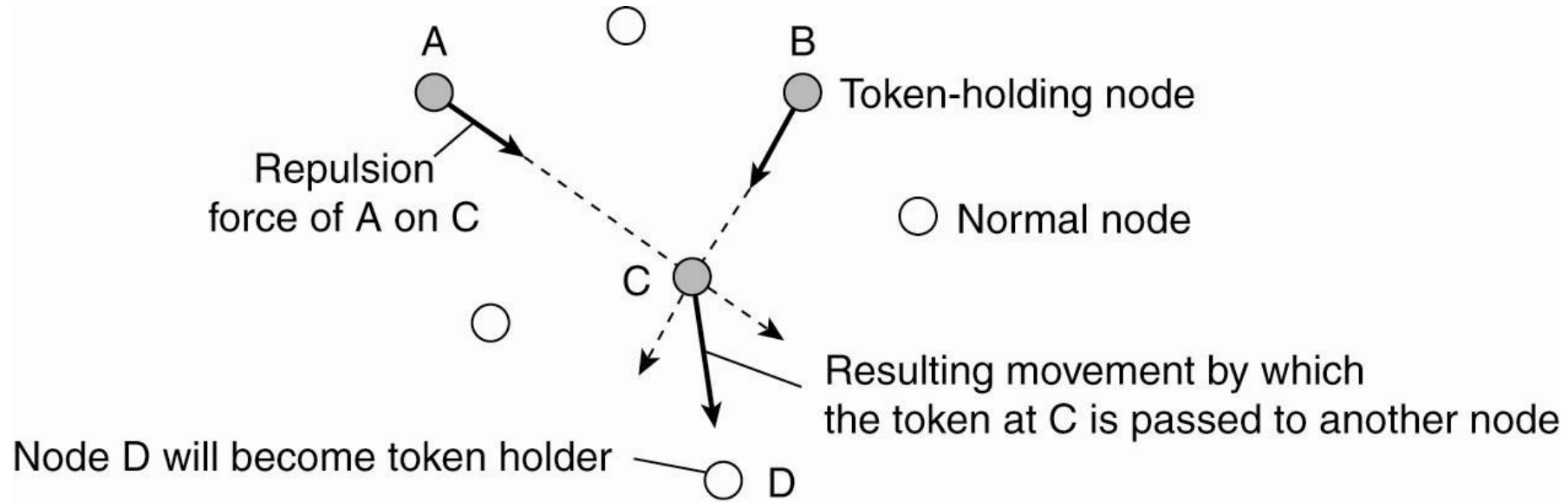


Il processo di Elezione in Sistemi a Larga Scala (1)

- Esempio: reti P2P (Peer-to-Peer), dove bisogna eleggere dei nodi **superpeer**.
- Vincoli per la selezione dei nodi superpeer:
 1. I nodi normali devono avere accesso a **bassa latenza** ai nodi superpeer.
 2. I nodi superpeer dovrebbero essere **distribuiti uniformemente** nella rete.
 3. La quantità di nodi superpeer non dovrebbe superare una **soglia predefinita** rispetto alla totalità dei nodi della rete.
 4. Ogni nodo superpeer non dovrebbe servire più di un **numero prefissato** di nodi normali.



Il processo di Elezione in Sistemi a Larga Scala (2)



Spostamento dei token in uno spazio bidimensionale, usando la metafora delle forze di repulsione.

Esempio in Python

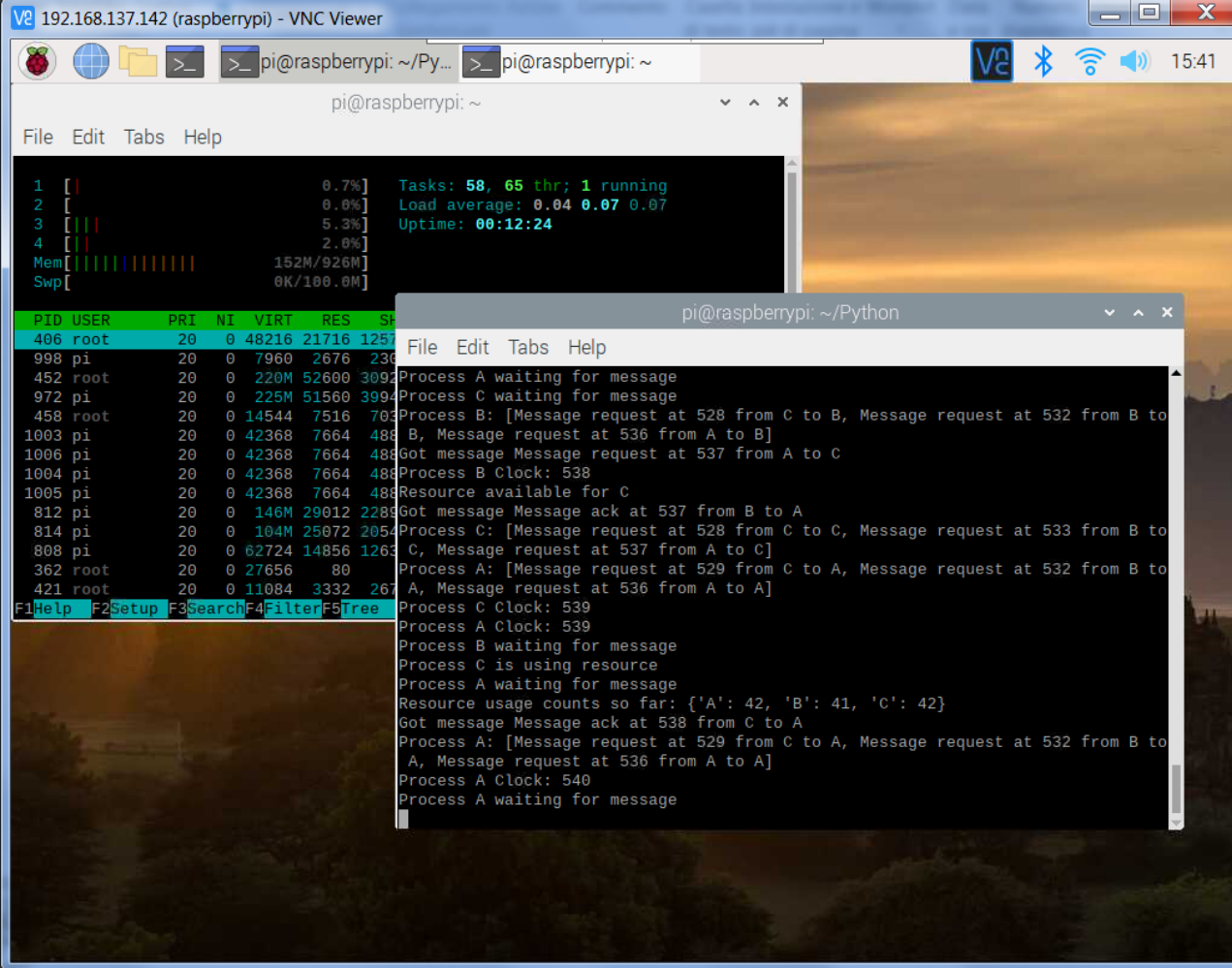
(<https://connorwstein.github.io/Lamport-Clocks/>)

- Esempio di implementazione in Python della **mutua esclusione** con gli **orologi logici di Lamport**.
- Il codice riportato fa sì che ogni processo sia a conoscenza dell'**ordinamento totale** degli eventi (i.e., le richieste di accesso/rilascio della risorsa condivisa).
- Protocollo:
 - per richiedere la risorsa, bisogna inviare un messaggio di richiesta a tutti con il timestamp impostato sul valore dell'orologio logico;
 - In seguito alla ricezione di un messaggio di richiesta, esso viene memorizzato nella coda delle richieste e viene spedito un ack al mittente;
 - al termine dell'utilizzo della risorsa, la relativa richiesta viene rimossa dalla coda delle richieste e viene trasmesso un messaggio di rilascio a tutti gli altri processi;
 - in seguito alla ricezione di un messaggio di rilascio, la relativa richiesta viene rimossa dalla coda delle richieste;
 - si accede alla risorsa, se la propria richiesta è in testa alla coda e sono visibili le richieste successive di tutti gli altri processi (qui interviene l'ordinamento totale).



Esempio in Python

(<https://connorwstein.github.io/Lamport-Clocks/>)



The screenshot shows a VNC viewer window titled "192.168.137.142 (raspberrypi) - VNC Viewer". The terminal window displays system statistics and a process list. The Python application window shows the execution of a Lamport clock algorithm with three processes (A, B, and C) and their message exchanges.

```
pi@raspberrypi: ~  
File Edit Tabs Help  
1 [ | 0.7% Tasks: 58, 65 thr; 1 running  
2 [ | 0.0% Load average: 0.04 0.07 0.07  
3 [ | 5.3% Uptime: 00:12:24  
4 [ | 2.0%  
Mem [ | 152M/926M  
Swp [ | 0K/100.0M  
PID USER PRI NI VIRT RES SH  
406 root 20 0 48216 21716 1257  
998 pi 20 0 7960 2676 236  
452 root 20 0 220M 52600 3092  
972 pi 20 0 225M 51560 3094  
458 root 20 0 14544 7516 703  
1003 pi 20 0 42368 7664 488  
1006 pi 20 0 42368 7664 488  
1004 pi 20 0 42368 7664 488  
1005 pi 20 0 42368 7664 488  
812 pi 20 0 146M 29012 2289  
814 pi 20 0 104M 25072 2054  
808 pi 20 0 62724 14856 1263  
362 root 20 0 27656 80  
421 root 20 0 11084 3332 267  
F1Help F2Setup F3Search F4Filter F5Tree
```

```
pi@raspberrypi: ~/Python  
File Edit Tabs Help  
Process A waiting for message  
Process C waiting for message  
Process B: [Message request at 528 from C to B, Message request at 532 from B to  
B, Message request at 536 from A to B]  
Got message Message request at 537 from A to C  
Process B Clock: 538  
Resource available for C  
Got message Message ack at 537 from B to A  
Process C: [Message request at 528 from C to C, Message request at 533 from B to  
C, Message request at 537 from A to C]  
Process A: [Message request at 529 from C to A, Message request at 532 from B to  
A, Message request at 536 from A to A]  
Process C Clock: 539  
Process A Clock: 539  
Process B waiting for message  
Process C is using resource  
Process A waiting for message  
Resource usage counts so far: {'A': 42, 'B': 41, 'C': 42}  
Got message Message ack at 538 from C to A  
Process A: [Message request at 529 from C to A, Message request at 532 from B to  
A, Message request at 536 from A to A]  
Process A Clock: 540  
Process A waiting for message
```

