6
Algoritmi Distribuiti

In questo capitolo ci occuperemo di algoritmi distribuiti in cui abbiamo varie entità che cooperano per la risoluzione di un problema. Come per gli altri capitoli, la trattazione è da intendersi solo introduttiva in quanto anche in questo caso l'argomento può essere l'oggetto di interi corsi. Pertanto focalizzeremo l'attenzione su un singolo problema distribuito che per vari motivi è quello più rappresentativo: il problema del *consenso*.

6.1 Introduzione

Gli algoritmi che abbiamo studiato finora sono algoritmi sequenziali, cioè algoritmi che assumono che ci sia una solo unità logica centrale (CPU) che esegue tutte le operazioni. Questa è la situazione tipica che si ha quando il programma viene implementato per un computer con una sola CPU. Il progresso della tecnologia ha, da un lato, portato alla possibilità di usare più di una CPU su un singolo computer, dall'altro, alla possibilità di interconnettere fra di loro tramite una rete molti computer in modo tale da farli comunicare. Queste due soluzioni tecnologiche portano a un approccio algoritmico sostanzialmente diverso da quello usato per gli algoritmi sequenziali. Ovviamente l'obiettivo non cambia: sviluppare algoritmi efficienti per risolvere i problemi. Tuttavia la presenza di più unità logiche (siano esse le CPU di un singolo computer, che comunicano tramite memoria condivisa, siano esse vari nodi di una rete che comunicano tramite scambio di messaggi), il dover gestire la comunicazione fra di esse e la possibilità che si verifichino dei guasti, rendono i problemi più difficili. Il vantaggio è quello di sfruttare la *parallelizzazione* della computazione: in ogni singolo istante ci sono più unità che con la propria capacità computazionale contribuiscono a risolvere il problema. In alcuni casi vogliamo sfruttare la parallelizzazione semplicemente per avere degli algoritmi più veloci. In altri casi la parallelizzazione o più precisamente la *distribuzione* del problema su più nodi, è un fatto intrinseco del problema. Si pensi ad esempio al problema della ricerca dei percorsi migliori per far comunicare i computer connessi ad Internet: in quel caso le informazioni di input, e anche l'output, sono distribuite fra i router della rete che devono comunicare fra di loro, scambiarsi tali informazioni e ognuno di essi deve calcolare un output locale.

Storicamente si è usato l'aggettivo *parallelo* per indicare un algoritmo progettato per un computer con più processori e l'aggettivo *distribuito* per indicare un algoritmo

progettato per nodi di una rete. La differenza sostanziale è nella tipologia di comunicazione. Gli algoritmi paralleli sfruttano la memoria condivisa fra i processori per far comunicare i vari nodi. Gli algoritmi distribuiti non hanno una tale possibilità e devono necessariamente usare la spedizione di messaggi per far comunicare i nodi del sistema. Sebbene le due tipologie di comunicazione abbiano caratteristiche diverse (ad es., il passaggio di informazioni tramite memoria condivisa è immediato, mentre quello tramite messaggi comporta una latenza che dipende dalla rete), di cui si deve tenere conto nella progettazione degli algoritmi, concettualmente svolgono la stessa funzione: permettono la comunicazione. In questa parte del corso parleremo di algoritmi ditribuiti e li classificheremo in base a varie caratteristiche, fra queste la tipologia di comunicazione che include la memoria condivisa, quindi di fatto la nostra definizione di algoritmo ditribuito includerà gli algoritmi paralleli. È d'obbligo notare che la nostra trattazione sarà necessariamente sommaria: sia gli algoritmi distribuiti in senso stretto che quelli paralleli sono l'argomento di interi corsi di studio, anche di carattere avanzato. Qui ci limiteremo a una introduzione a tali argomenti.

6.1.1 Definizione di sistema/algoritmo distribuito

Che gli algoritmi distribuiti siano più ostici di quelli sequenziali lo si capisce immediatamente dalla definizione. Di fatto non esiste una singola definizione universalmente utilizzata, come per gli algoritmi sequenziali per i quali, a meno di differenze classificabili come sottigliezze linguistiche, la definizione è unica. Anche per le "variazioni sul tema" studiate nei capitoli precedenti, la definizione è chiara e precisa. Non vale lo stesso per il caso degli algoritmi distribuiti. A titolo di esempio riportiamo di seguito le definizioni utilizzate in alcuni libri di testo:

- (Lynch [26]) "Gli algoritmi distribuiti sono algoritmi progettati per funzionare su molti processori interconessi fra di loro. Parti di un algoritmo distribuito operano simultaneamente e indipendentemente e ognuna ha a disposizione informazioni limitate. Gli algoritmi devono funzionare correttamente anche se i singoli processori e i canali di comunicazione operano a velocità diverse e sono soggetti a guasti."
- (Attiya-Welch [3]) "Un sistema distribuito è un insieme di unità di computazione che comunicano fra di loro. Questa definizione generale include un ampio insieme di moderni sistemi di computazione, dai chip VLSI a Internet, passando per multiprocessori con memoria condivisa e cluster di workstations."
- (Guerraoui-Rodrigues [17]) "Computare in modo distribuito ha a che fare on la progettazione di algoritmi per un insieme di processi con l'obiettivo di cooperare. Oltre all'esecuzione simultanea, alcuni dei processi di un sistema distribuito possono fermarsi, ad esempio perché si rompono oppure perché vengono disconnessi, mentre altri rimangono attivi e continuano ad operare."
- (Tel [31]) "Per sistema distribuito intendiamo tutte le applicazioni dove vari computer o processori cooperano in qualche modo. Questa definizione include sia reti locali che reti estese, ma anche computer multiprocessore in cui ogni processore ha la sua unità di controllo ed un suo sistema di processi che cooperano."

- (Birman [5]) "Un sistema di computazione distribuita è un insieme di programmi, eseguiti su uno o più computer che coordinano le proprie azioni scambiando messaggi. Una rete di computer è una collezione di computer interconnessi da hardware che supporta lo scambio di messaggi."
- (Tanenbaum [30]) "Un sistema distributo è una collezione di computer indipendenti che appare all'utente come un sistema singolo."

Senza la pretesa, ovviamente, di avere elencato tutte le definizioni utilizzate (esistono molti altri libri su sistemi e algoritmi distribuiti e paralleli), quelle citate fanno rendere conto della difficoltà: ci sono molti aspetti da tenere in considerazione. Pertanto al posto di fornire una nuova definizione, ci limitiano ad elencare gli aspetti importanti che si evincono dalla definizioni citate:

- molte unità di computazione, che chiameremo in modo generico *nodi*, *processi* o *processori*;
- esecuzione simultanea e indipendente delle istruzioni (programma; ogni singola unità può avere il suo programma, anche se nella maggior parte dei casi il programma è lo stesso per ogni unità);
- informazioni limitate disponibili ad ogni unità di computazione;
- diversi modi di comunicare (memoria condivisa, messaggi);
- diverse velocità, sia per l'esecuzione dei programmi sia per la comunicazione;
- possibilità che ci siano dei guasti, sia per le unità di computazione sia per i canali di comunicazione.

Chiudendo il discorso sulla "definizione" di sistema distribuito, prima di proseguire, citiamo una definizione scherzosa ma molto veritiera dovuta a Leslie Lamport che l'ha utilizzata in un email nel 1987 [21]:

> *Un sistema distribuito è un sistema nel quale il guasto di un computer, di cui non sapevi nemmeno dell'esistenza, rende il tuo computer inutilizzabile.*

I sistemi/algoritmi distribuiti vengono solitamente classificati in base alle seguenti caratteristiche:

- tipo di comunicazione: scambio di messaggi e memoria condivisa;
- sincronia dei processori: sistemi sincroni e sistemi asincroni (e sistemi parzialmente sincroni)
- vari tipi di guasti: crash o comportamento arbitrario dei processori, perdita, duplicazione e riordino dei messaggi spediti.

Focalizzeremo l'attenzione sulla comunicazione con scambio di messaggi. Di fatto tutto ciò che diremo può essere applicato anche al caso della memoria condivisa in quanto avendo a disposizione una memoria condivisa è facile "simulare" la spedizione

e la ricezione di un messaggio (senza nemmeno doversi preoccupare di eventuali errori di comunicazione). Dunque la memoria condivisa è più potente dello scambio dei messaggi. Utilizzando la terminologia introdotta poc'anzi possiamo dire che un algoritmo distribuito può essere implementato anche su un sistema parallelo, mentre un algoritmo parallelo potrebbe non essere implementabile in un sistema distribuito. Il Capitolo 17 di L96 [26] discute della relazione fra questi due modelli. Senza scendere ulteriormente in dettagli, nel prosieguo considereremo algoritmi che comunicano tramite scambio di messaggi.

Inizieremo con il descrivere algoritmi per sistemi sincroni e poi tratteremo i sistemi asincroni. La nostra trattazione sarà basata su un singolo problema, il problema del *consenso*, e descriveremo sia algortimi che risultati di impossibilità. Informalmente, il problema del consenso consiste nel far prendere a tutti i processori la stessa decisione (definiremo formalmente il problema nel prosieguo) ed è di fondamentale importanza in quanto può essere considerato come la base per la coordinazione necessaria in un sistema distribuito.

6.1.2 *Formalismo e misure per l'analisi*

Analizzare un algoritmo distribuito può essere molto subdolo. All'apparenza le asserzioni che si fanno sui sistemi distributi sono semplici, tuttavia i problemi possono nascondere delle difficoltà che li rendono complicati. Per poter analizzare gli algoritmi e provare dei risultati è necessario fornire un formalismo che permetta un ragionametno rigoroso. Più preciso è il formalismo più rigoroso sarà il ragionamento. Tuttavia un formalismo troppo rigoroso porta a ragionamenti molto più dettagliati che possono essere fonte di errori se non li si affronta con adeguata attenzione. Quindi, come per la definizione di sistema distribuito, esistono varie scuole di pensiero: c'è chi opta per un formalismo molto rigoroso, come ad esempio l'IOA (automi di Input/Output) di Lynch [26], e chi per formalismi meno pesanti, come ad esempio uno pseducodice. Entrambi gli approcci hanno vantaggi e svantaggi e probabilmente la scelta migliore la si può fare in funzione del problema da affrontare. Per i nostri scopi, e anche per il tempo a disposizione, opteremo per una descrizione informale dei problemi e degli algoritmi tramite pseudocodice, in linea con quanto visto finora nel corso.

La principale differenza nella descrizione di un algoritmo distribuito rispetto a un algoritmo sequenziale è dovuta al fatto che un algoritmo distributo, per un insieme di n processori, è di fatto un insieme di n algoritmi, uno per ogni processore. Nella maggior parte dei casi gli n algoritmi sono copie dello stesso algoritmo. L'algoritmo può avere comportamenti diversi da processore a processore sfruttando delle informazioni locali, come ad esempio un identificativo del processore. Ogni processore del sistema $P = \{1, 2, \ldots, n\}$ sarà specificato tramite un indice $i \in P$, e quando faremo riferimento a un qualunque aspetto relativo al processore i, lo specificheremo usando i come pedice. Ad esempio se vogliamo specificare che una variabile, ad esempio *status*, appartiene al processore i, scriveremo $status_i$.

Quando ragioneremo su un algoritmo distribuito, vorremo tipicamente provare delle proprietà dette di

- *Safety:* non succedono mai cose cattive. Questo corrisponde alla correttezza di un algoritmo sequenziale. Ad esempio, nel problema del consenso una proprietà di safety è che due processori non facciano scelte diverse.
- *Liveness:* che prima o poi qualcosa (di buono) accade. Questo corrisponde alla proprietà di terminazione degli algoritmi sequenziali. Ad esempio, nel problema del consenso, si richiede che prima o poi i processori facciano una scelta.

Per valutare la bontà di un algoritmo saremo interessati fondamentalmente a due misure: la complessità di tempo e la complessità di comunicazione. La seconda è più facile da definire in quanto basterà contare il numero di messaggi, o se necessario il numero di bit, spediti. Per quanto riguarda il tempo può essere più difficile definire una misura in quanto essa dipende dal tipo di sincronia presente nel sistema. Per sistemi sincroni sarà il numero di *round* utilizzati dall'algoritmo (descriveremo fra poco i sistemi sincroni). Per i sistemi asincroni è molto più complicato in quanto ogni componente del sistema misura il tempo in modo diverso, per cui è necessario utilizzare un'osservazione esterna del tempo.

6.1.3 Guasti

Un altro aspetto importante è la resilienza ai guasti. Questo è un aspetto che è totalmente assente per un algoritmo sequenziale: un algoritmo sequenziale non è resiliente ai guasti in quanto l'esecuzione su un computer di un programma non prevede "guasti". Certamente, anche per un singolo computer ci possono essere dei guasti, ed è per questo, ad esempio, che facciamo delle copie di backup dei file, ma non sono contemplati guasti che riguardano l'esecuzione di un programma. In un sistema distribuito, invece, un guasto è un naturale evento che può verificarsi durante l'esecuzione di un programma distribuito. Se si assume che non ci siano guasti, progettare algoritmi distribuiti è relativamente semplice. La possibilità di avere dei guasti rende molto più complesse sia la progettazione che l'analisi degli algoritmi.

6.1.4 Sincronia

Sistemi sincroni. In un sistema sincrono i processori eseguono i programmi in perfetta sincronia e anche i canali di comunicazione operano in sincronia con i processori. Pertanto l'esecuzione di un algoritmo distribuito in un sistema sincrono procede per iterazioni, in gergo chiamate *round*. In ogni round ogni singolo processore esegue un numero fissato di istruzioni, può spedire un messaggio e tutti i messaggi spediti vengono ricevuti dai destinatari che potranno utilizzarli nel round successivo. L'esecuzione di un algoritmo in un sistema sincrono può essere dunque descritta da una sequenza

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots$$

dove C_i è una *configurazione*, cioè l'insieme degli stati dei singoli processori dopo i round, M_i è l'insieme di di messaggi spediti durante il round i, mentre N_i è l'insieme di messaggi ricevuti nel round i. I due insiemi potrebbero essere diversi a causa di guasti del sistema.

Considereremo principalmente sistemi sincroni. Verso la fine del capitolo tratteremo brevemente sistemi asincroni e parzialmente sincroni.

Sistemi asincroni. In un sistema asincrono non esiste il concetto di tempo: ogni processore opera alla sua velocità e non c'è relazione fra le velocità dei processori. Analogamente non esiste un limite di tempo per la consegna di un messaggio: ogni messaggio rimane nel canale di comunicazione per un tempo indeterminato prima di esssere consegnato.

Sistemi parzialmente sincroni. La parziale sincronia impone dei vincoli sul tempo necessario ad ogni singolo processore per eseguire una istruzione e al tempo necessario a consegnare un messaggio. Pertanto in un sistema parzialmente sincrono potremo fare delle assunzioni sulle velocità dei singolo processori e sul tempo di consegna di un messaggio.

Relazione. Un sistema sincrono è un caso particolare di un sistema asincrono, quindi da questo punto di vista progettare un un algoritmo per un sistema sincrono è più facile. Infatti in un sistema sincrono tutti i processori eseguono un'istruzione nello stesso istante e tutti i messaggi vengono immediatamente consegnati. Un algoritmo progettato per un sistema sincrono potrebbe non funzionare (anzi, nella maggior parte dei casi non funziona) in un sistema asincrono. Mentre un algoritmo progettato per un sistema asincrono funzionerà anche in un sistema sincrono. Simmetricamente un risultato di impossibilità per i sistemi sincroni è valido anche per i sistemi asincroni: se non è possibile risolvere il problema in un sistema più facile da gestire, a maggior ragione non è possibile risolvere in un sistema più difficile da gestire.

6.2 Problema del consenso

Il *problema del consenso* è l'astrazione di molti problemi di coordinazione. Dovrebbe essere abbastanza evidente che coordinare le azioni in un sistema distribuito è alla base di qualunque computazione distribuita. Coordinarsi significa mettersi d'accordo, prendere una stessa decisione. Il problema del consenso formalizza queste necessità. La definzione formale del problema ammette delle varianti. Assumeremo che i processori abbiano un input, un valore iniziale, e che debbano produrre un output. Gli output devono essere tutti uguali, questo rappresenta il mettersi d'accordo e devono anche in qualche modo essere funzione dell'input. Quest'ultimo requisito è abbastanza naturale e dal punto di vista formale serve ad evitare soluzioni precostruite che chiaramente non avrebbero senso nella realtà (ad esempio, dare in output sempre 0).

Problemi di consenso possono essere individuati in una infinità di situazioni reali. Ad esempio due processori potrebbero dover mettersi d'accordo sul se annullare o rendere definitiva una transazione in un database distribuito. Oppure due processori potrebbero doversi mettere d'accordo su delle letture indipendenti fatte a dei sensori, ad esempio in un aereo per misurare l'altitudine.

Il problema CONSENSO viene definito come segue. La seguente definizione è generica: in funzione delle particolari caratteristiche del sistema considerato (es. tipo di guasti) la definizione potrà subire delle piccole variazioni; ad esempio non avrebbe senso richiedere che un processore che si guasta in modo bizantino debba soddisfare le proprietà richieste in quanto per definizione di guasto bizantino un processore guasto può comportarsi in maniera arbitraria. Pertanto diamo qui una definizione generale del problema per poi istanziarla in modo più preciso caso per caso.

Problema CONSENSO**:** dato un sistema distribuito con n processori $p_1, \ldots, p_n$, che iniziano la computazione con un valore di input $v_i \in V$, $i = 1, \ldots, n$, dove V è l'insieme di tutti i possibili valori di input, i processori devono stabilire un valore di output, una *decisione*, in modo tale che 3 proprietà vengano soddisfatte:

- Accordo: Tutte le decisioni sono uguali.
- Validità: Se tutti i valori iniziali sono uguali, cioè se $v_i = v$ per tutti i processori, allora la decisione deve essere v.
- Terminazione: Tutti i processori decidono.

Osserviamo che se il sistema è *affidabile*, cioè non ci sono guasti né dei processori né dei canali di comunicazione, i problemi di consenso sono facili da risolvere: basta scambiarsi gli input ed usare una regola comune per decidere l'output in funzione dell'input. Quindi nel prosieguo considererermo il problema del consenso in vari scenari, in ognuno dei quali instanzieremo una specifica *inaffidabilità* del sistema.

6.3 Consenso sincrono con perdita di messaggi

In questa sezione considereremo la seguente definizione del problema del consenso, detta anche problema dell'attacco coordinato. Uno dei primi risultati relativi al problema del consenso, introduce il problema in un ipotetico scenario di guerra, chiamandolo il *problema dei due generali*, conosciuto anche come il problema dell'attacco coordinato. Immaginiamo una guerra antica in cui si usavano ancora i messaggeri per comunicare, e consideriamo una situazione in cui i generali di due armate di un esercito, devono prendere una decisione sul se attaccare o meno il nemico. I generali possono comunicare solo con dei messaggeri che però possono essere catturati dal nemico. Quindi la comunicazione non è affidabile. Nel caso del problema dei due generali ci sono solo 2 processori, più in generale il numero di processori è arbitrario.

Formalmente abbiamo il seguente problema del consenso nella versione "Attacco Coordinato".

Problema CONSENSOAC**:** In un sistema distribuito di n processori, che comunicano tramite scambio di messaggi su un canale che può perdere i messaggi, ogni processore ha in input un bit[1], cioè $V = \{0, 1\}$, e deve decidere il valore di un bit di output. Le 3 proprietà da soddsifare sono

[1] Il valore del bit di input rappresenta la propria opinione sul se attaccare, 1, o non attaccare, 0.

- Accordo: Tutte le decisioni sono uguali (quindi o tutti danno in output 0 oppure tutti danno in output 1).
- Validità: Se tutti i processori iniziano con 0 allora l'output deve essere 0. Se tutti i processori iniziano con 1 e tutti i messaggi vengono consegnati, allora l'output deve essere 1.
- Terminazione: Tutti i processori decidono.

6.3.1 Sistema deterministico

In un sistema sincrono, deterministico, e comunicazione non affidabile il problema ConsensoAC non può essere risolto.

Teorema 6.3.1 *Consideriamo un sistema distribuito con due processori* 1 *e* 2*, connessi da un canale di comunicazione. Non esiste un algoritmo che possa risolvere il problema* ConsensoAC*.*

Dimostrazione. Per contraddizione, assumiamo che esista un algoritmo A che risolve il problema. Una qualunque esecuzione dell'algoritmo dipende esclusivamente dall'input dei due processori e dallo specifico pattern di consegna dei messaggi dell'esecuzione. Senza perdere in generalità assumiamo che l'algoritmo A faccia spedire ad entrambi i processori un messaggio in ogni round, in quanto se così non fosse potremmo usare dei messaggi fittizi.

Sia α l'esecuzione in cui i due processori hanno entrambi come input 1 e in cui tutti i messaggi vengono consegnati, cioè non ci sono guasti sui canali di comunicazione. Dalla proprietà di terminazione si ha che i processori devono decidere e dalla proprietà di validità si ha che tale decisione deve essere 1. Sia r il round entro il quale entrambi decidono. Sia α_0 l'esecuzione che differisce da α solo per il fatto che tutti i messaggi dopo i primi r round vengono persi. Quindi α_0 è identica ad α per i primi r, ma diversa dal round $r+1$ in poi in quanto i messaggi spediti nel round r non verranno consegnati. Poichè le due esecuzioni sono indistiguibili fino al round $r+1$ (la perdita dei messaggi spediti al round r potrà essere rilevata dai processori solo al round $r+1$), sia p_1 che p_2 decideranno 1 anche in α_1, visto che in α decidono 1 prima del round $r+1$.

A questo punto consideriamo una sequenza $\alpha_1, \alpha_2, \ldots$ di esecuzioni costruite a partire da α_0 in cui ogni esecuzione successiva è uguale alla precendete a parte che l'ultimo messaggio non viene consegnato. Più precisamente, sia α_1 l'esecuzione indentica ad α_0 tranne che l'ultimo messaggio da p_1 a p_2 viene perso, e sia α_2 l'esecuzione indentica ad α_1 tranne che l'ultimo messaggio da p_2 a p_1 viene perso. La Figura 6.1 mostra graficamente le due esecuzioni.

Consideriamo α_1. Essa, per il processore p_1 è indistinguibile da α_0. Pertanto p_1, che decide 1 in α_0, deciderà 1 anche in α_1. Per la proprietà di accordo, anche il processore p_2 deciderà 1 in α_1.

Consideriamo ora α_2. Essa, per il processore p_2 è indistinguibile da α_1. Pertanto p_2, che decide 1 in α_1, deciderà 1 anche in α_2. Per la proprietà di accordo, anche il processore p_1 deciderà 1 in α_2.

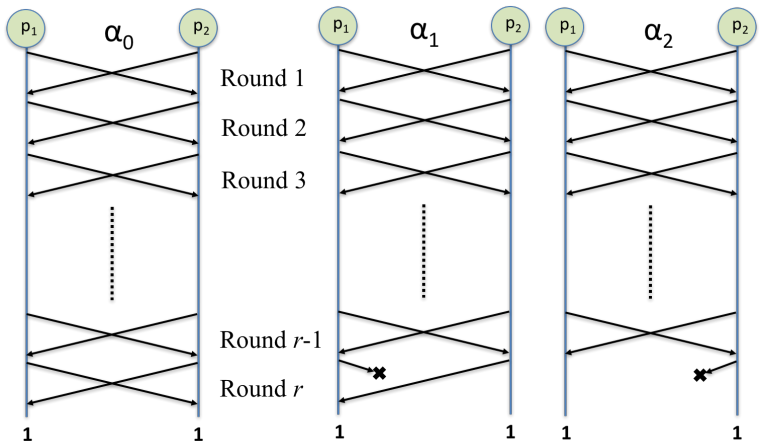

Figura 6.1: Esecuzioni α_0, α_1 e α_2

Le esecuzioni α_3 e α_4 sono definite in modo analogo, e per un ragionamento simile si ha che p_1 e p_2 decidono 1 anche in α_3 e α_4. Procedento in questo modo si arriverà a una esecuzione α' in cui nessun messaggio viene consegnato ma i processori comunque decidono 1 (ricordiamo che l'input dei processori è sempre 1 in tutte le esecuzioni finora considerate).

A questo punto consideriamo l'esecuzione α'', identica ad α', quindi senza messaggi consegnati, ma in cui p_2 ha come input 0. Per p_1 non c'è differenza fra α' e α'', quindi p_1 decide comunque 1 in α''. Di conseguenza anche p_2 decide 1 in α''. Sia α''' l'esecuzione identica ad α'', con la differenza del valore di input di p_1 che è 0. Per il processore p_2 non c'è differenza fra α'' e α''', quindi p_1 decide 1 anche in α'''. Di conseguenza, per la proprietà di accordo, anche p_1 decide 1 in α'''. Questo però è una violazione della proprietà di validità in quanto in α''' entrambi i processori hanno come input 0 ma decidono 1. Pertanto aver assunto che esiste un algoritmo A che risolve il problema ha portato a un contraddizione. □

Il Teorema 6.3.1 dimostra una chiara limitazione: c'è poco da fare se la comunicazione è inaffidabile. Ovviamente in pratica è molto difficile perdere tutti i messaggi. Inoltre possiamo anche o rilassare i requisiti del problema oppure rendere più potente il modello di computazione. Ad esempio possiamo utilizzare la randomizzazione.

6.3.2 *Algoritmo randomizzato*

Consideriamo un sistema di n processori che possono utilizzare delle scelte casuali. Ogni processore inizia con un valore di input in $\{0,1\}$. Assumeremo anche che l'algoritmo deve terminare l'esecuzione in un numero fissato $r \geq 1$ di round; in altre parole ogni processore deve decidere al più tardi durante il round r. Assumeremo che ogni processore ha una variabile *write-once* con valori in $\{0,1\}$, la cui scrittura rappresenta l'azione della decisione. Ad ogni round verrà spedito un messaggio da ogni processore (possiamo usare dei messaggi fittizi se non c'è niente da spedire), ed un qualsiasi numero di messaggi può non arrivare a destinazione. Per poter gestire le scelte casuali la proprietà di accordo verrà leggermente rilassata:

- Accordo: Tutte le decisioni sono uguali (quindi o tutti danno in output 0 oppure tutti danno in output 1) con una probabilità pari ad almeno $1 - \epsilon$, dove ϵ è un numero positivo piccolo a piacere.

Per poter formalizzare l'algoritmo e la sua analisi definiamo un *pattern di comunicazione* come un sottoinsieme dell'insieme

$$\{(i, j, k) : (i, j) \text{ è un canale di comunicazione, e } 1 \leq k\}.$$

Ogni elemento (i, j, k) di un pattern di comunicazione rappresenta la spedizione di un messaggio da i a j durante il round k. Un pattern di comunicazione γ è detto *ammissibile* se per ogni $(i, j, k) \in \gamma$ si ha che $k \leq r$.

Per analizzare l'algoritmo assumeremo che esiste un avversario che sceglie quali messaggi far arrivare a destinazione e quali no. Pertanto l'avversario sceglierà uno specifico pattern di comunicazione ammissibile; ovviamente l'avversario potrà scegliere anche l'input dei processori. Non ha, invece, controllo sulle scelte casuali dei processori.

Per una fissata strategia (pattern di comunicazione) dell'avversario, le scelte casuali effettuate dai processori determinano in modo univoco una particolare esecuzione. Quindi le scelte probabilistiche determinano una distribuzione di probabilità sull'insieme delle possibili esecuzioni.

Per semplicità di esposizione assumeremo che il grafo delle comunicazioni è completo: ogni processore può spedire messaggi ad ogni altro processore. L'algoritmo si può facilmente estendere al caso di grafi di comunicazione non completi. Durante l'esecuzione dell'algoritmo, ogni processore tiene traccia di ciò che conosce riguardo ai valori iniziali degli altri processi.

Definiamo un ordine parziale $\leq_\gamma$ relativo al pattern di comunicazione γ, sulle coppie (i, k), dove i è un processore e $k \geq 0$ un intero che rappresenta il numero di un round. Quando diremo "al tempo k", intenderemo l'istante di tempo esattamente dopo l'esecuzione di k round, e quindi prima dell'inizio del $(k + 1)$-esimo round. L'ordine parziale $\leq_\gamma$ rappresenta il flusso di informazione convogliato dai messaggi ed è definito come segue:

- $(i, k) \leq_\gamma (i, k')$ per ogni i e tutti i k, k' tali che $0 \leq k \leq k'$.
- Se $(i, j, k) \in \gamma$ allora $(i, k - 1) \leq_\gamma (j, k)$.
- Se $(i, k) \leq_\gamma (i', k')$ e $(i', k') \leq_\gamma (i'', k'')$ allora $(i, k) \leq_\gamma (i'', k'')$.

Il primo caso descrive il flusso di informazione di un singolo processore: la conoscenza di un processore può solo aumentare con il procedere dell'esecuzione. Il secondo caso descrive il flusso di informazione convogliato da un messaggio: se j riceve un messaggio da i al round k allora ciò che conosce i al round $k - 1$ sarà conosciuto da j. Questo ovviamente significa che quando un processore spedisce un messaggio include tutta la sua conoscenza. Infine il terzo caso è semplicemente la proprietà transitiva.

Per un pattern di comunicazione ammissible γ, definiamo il *livello di informazione*, $livello_\gamma(i, k)$ per un qualsiasi processore i e un tempo $k, 0 \leq k \leq r$, in modo ricorsivo come segue:

- $k = 0$. Allora $livello_\gamma(i,k) = 0$.
- $k > 0$ e c'è un qualche $j \neq i$ tale che $(j,0) \not\leq_\gamma (i,k)$. Allora $livello_\gamma(i,k) = 0$.
- $k > 0$ e $(j,0) \leq_\gamma (i,k)$ per ogni $j \neq i$. Sia per ogni $j \neq i$, $l_j = \max\{livello_\gamma(j,k') | (j,k') \leq_\gamma (i,k)\}$. Si noti che l_j è il più alto livello che i sa che j ha raggiunto. Si noti inoltre che $0 \leq l_j \leq k-1, \forall j$. Allora $livello_\gamma(i,k) = 1 + \min\{l_j | j \neq i\}$.

Informalmente, ogni processore inizia con un livello di informazione pari a 0. Quando riceve messaggi da tutti gli altri processori passa al livello di informazione 1; quando sa che tutti gli altri processori hanno raggiunto il livello 1 passa al livello 2, e così via.

La Figura 6.2 mostra un esempio per il pattern di comunicazione

$$\gamma = \{(1,2,1),(1,2,2),(2,1,2),(1,2,3),(2,1,4),(1,2,5),(2,1,5),(1,2,6)\}.$$

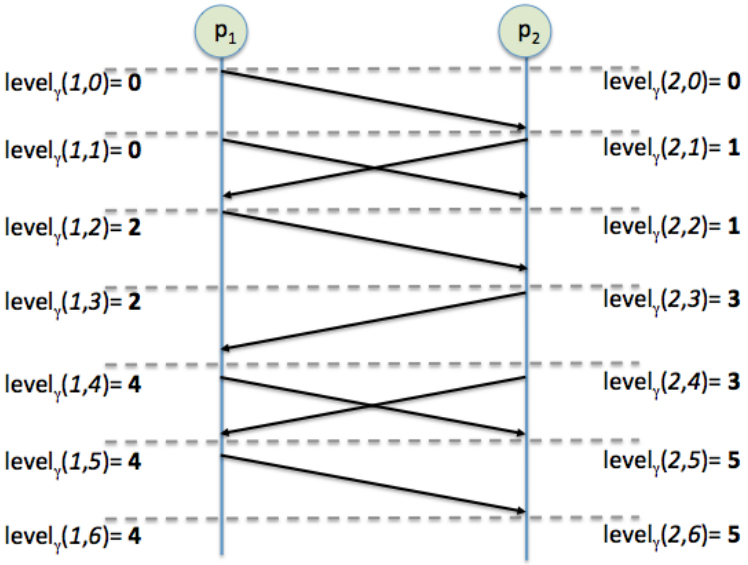

Figura 6.2: Livello di informazione

Lemma 6.3.2 *Per un qualsiasi pattern di comunicazione ammissibile γ, un qualsiasi k, $0 \leq k \leq r$, e qualsiasi i e j, si ha che $|livello_\gamma(i,k) - livello_\gamma(j,k)| \leq 1$.*

Dimostrazione. Il livello di comunicazione di un processore i può essere incrementato a un valore $s+1$ solo quando il processore i viene a sapere che il livello di comunicazione di j è s. E viceversa. Quindi non è possibile che i livelli di comunicazione di due processori differiscano per più di 1. □

Lemma 6.3.3 *Se γ è il pattern di comunicazione "completo", cioè che contiene tutte le possibili triple (i,j,k), $1 \leq k \leq r$, allora $livello_\gamma(i,k) = k$, per tutti i valori di i e k.*

Dimostrazione. Se nessun messaggio si perde, il livello di comunicazione viene incrementato di 1 ad ogni round. □

A questo punto siamo pronti a descrivere l'algoritmo che chiameremo RANDOMATTACK. Ogni processore i mantiene traccia del suo livello di informazione in una variabile *level*. Inoltre, il processore 1 sceglie un valore casuale *key*, un intero nell'intervallo $[1, r]$; questo valore sarà aggiunto a tutti i messaggi spediti. Anche i valori iniziali di tutti i processori verranno aggiunti a tutti i messaggi spediti. Dopo r round, ogni processore deciderà usando la seguente regola: se il valore del livello di informazione *level* è maggiore o uguale a *key*, allora la decisione è 1, altrimenti la decisione è 0.

Algorithm 21: RANDOMATTACK$_i$

Stato:
$rounds \in \mathbb{N}$, inizialmente 0
$decision \in \{unknown, 0, 1\}$, inizialmente *unknown*
$key \in [1, r] \cup undef$, inizialmente *undef*
for every $j, 1 \leq j \leq n$
  $val(j) \in \{undef, 0, 1\}$, inizialmente *undef* per $j \neq i$ e input di i per $j = i$
  $level(j) \in [-1, r]$, inizialmente -1 per $j \neq i$ e 0 per $j = i$

Randomizzazione:
Se $i = 1$ e $rounds = 0$ allora $key := random([1, r])$

Generazione messaggi:
Spedisci (L, V, key) dove L è il vettore dei livelli $level(j)$ e V è il vettore dei valori $val(j)$

Computazione:
$rounds := rounds + 1$
Sia (L_j, V_j, k_j) il messaggio ricevuto da j, per ogni j da cui si riceve un messaggio
if *per un qualche* j, $k_j \neq undef$ **then**
  $key := k_j$
for *tutti i* $j \neq i$ **do**
  Se per qualche i', $V_{i'}(j) \neq undef$ allora $val(j) := V_{i'}(j)$
  Se per qualche i', $L_{i'}(j) > level(j)$ allora $level(j) := \max_{i'}\{L_{i'}(j)\}$
$level(i) := 1 + \min\{level(j) | j \neq i\}$
if $rounds = r$ **then**
  if $key \neq undef$ *and* $level(i) \geq key$ *and* $\forall j, val(j) = 1$ **then**
    $decision := 1$
  else
    $decision := 0$

Teorema 6.3.4 RANDOMATTACK *risolve il problema* CONSENSOAC *con una probabilità di errore di* $\epsilon = 1/r$.

DIMOSTRAZIONE. Osserviamo prima che l'algoritmo calcola correttamente i livelli di

informazione. Cioè in ogni esecuzione con un pattern di comunicazione ammissibile γ, per ogni k, $0 \leq k \leq r$, e per ogni i, dopo k round si ha che il valore di $level(i)$ è effettivamente uguale a $livello_\gamma(i,k)$. Inoltre, osserviamo che dopo k round, se $level(i)_i \geq 1$, allora sono definiti sia key_i che $val(j)_i$ per tutti gli i e tali valori sono uguali, rispettivamente, al valore casuale scelto del processore 1 nel primo round e ai valori iniziali dei processori.

L'algoritmo soddisfa la proprietà di terminazione: tutti i processori decidono nel round r. Proviamo che l'algoritmo soddisfa la proprietà di validità. Se tutti i processori iniziano con 0, la decisione sarà 0 in quanto per decidere 1 è necessario che tutti i $val(j)$ siano 1. Se tutti i processori iniziano con 1 e tutti i messaggi vengono consegnati, dal Lemma 6.3.3 e dal fatto che l'algoritmo calcola correttamente i livelli di informazione si ha che per ogni i, $level(i)_i = r$ nel momento in cui viene presa la decisione; poiché $r \geq 1$, si ha anche che sono definiti sia key_i che $val(j)_i$ per tutti gli i ed essendo $key \leq r$, la decisione è 1.

Rimane da provare che l'algoritmo soddisfa la proprietà di accordo con una certa probabilità. Per ogni i, sia ℓ_i il valore di $level(i)_i$ nel momento in cui il processore i prende la sua decisione (nel round r). Il Lemma 6.3.2 ci dice che gli ℓ_i distano al massimo 1. Cominciamo con il distinguere due possibili casi: (1) almeno un processore inizia con 0 e (2) tutti i processori iniziano con 1. Nel primo di questi due casi tutti i processori decidono 0 quindi non c'è errore. L'errore è possibile solo quando tutti i processori hanno come input 1, perché questo è l'unico caso in cui l'istruzione **if** per la decisione può far eseguire ad alcuni processori la parte **then** e ad altri la parte **else**, in funzione dei valori di *key* e degli ℓ_i.

Se il valore scelto per *key* è maggiore di $\max\{\ell_i\}$, allora tutti i processori decidono 0 e non c'è nessun errore. Se invece $key \leq \min\{\ell_i\}$, allora tutti decidono 1 e non c'è nessun errore. Pertanto l'unico caso in cui c'è disaccordo sulla decisione, cioè alcuni processori decidono 1 e altri 0 si ha quando $key = \max\{\ell_i\}$. La probabilità di un tale evento è $1/r$, visto che $\max\{\ell_i\}$ è determinato dall'avversario ed è un valore fra 0 e r e *key* è scelto casualmente in modo uniforme fra 0 e r. □

6.4 *Consenso sincrono, deterministico*

In questa sezione considereremo il problema in sistemi sincroni deterministici (cioè senza l'uso della randomizzazione) e considereremo sia guasti di tipo stop, sia guasti di tipo bizantino.

6.4.1 *Con guasti stop, algoritmo* EIGStop

Consideriamo ora il problema del consenso in un sistema distribuito con la possibilità che i processori possano guastarsi. Il grafo di comunicazione è completo e la comunicazione è affidabile. Se un processore si guasta durante la spedizione di un messaggio in broadcast, solo un sottoinsieme (che potrebbe anche essere vuoto) dei destinatari riceverà il messaggio. I possibili valori di input e quelli di output sono presi da un insieme V.

La definizione formale del problema, che chiameremo ConsensoGS, consenso per guasti stop, è la seguente.

Problema ConsensoGS**:** Consideriamo un sistema distribuito di n processori, che comunicano tramite scambio di messaggi su un canale affidabile. I processori possono rompersi con dei guasti stop. Ogni processore ha un valore di input $v_i \in V$ e i processori devono decidere un valore di output in modo tale da soddisfare le seguenti 3 proprietà.

- (Accordo) Tutte le decisioni sono uguali.

- (Validità) Se tutti i processori iniziano con $v \in V$ allora l'output deve essere v.

- (Terminazione) Tutti i processori che non si guastano, decidono.

Assumeremo che ci sia un limite f, $0 \leq f \leq n$, al numero di guasti che si possono verificare in ogni singola esecuzione. I casi $f = 0$ e $f = n$, vengono solitamente indicati con *failure-free* e *wait-free*:

- *failure-free:* $f = 0$, non ci sono guasti quindi tutti i processori decidono.

- *wait-free:* $f = n$, tutti i processori si possono rompere. Tuttavia, ogni processore non guasto deciderà, indipendentemente dai guasti degli altri processori.

Per questo problema, poichè i processori si guastano semplicemente fermandosi, è possibile dare un algoritmo molto semplice, che chiameremo FloodSet. L'algoritmo "inonda" la rete con l'insieme dei valori di input e poi utilizza una regola prestabilita per decidere.

Sia V l'insieme dei possibili valori inziali e sia v_0 un fissato elemento di V. Sia f il numero massimo di guasti. Ogni processore manterrà nella variabile W l'insieme dei valori di input di cui conosce l'esistenza. Inizialmente per ogni processore i si avrà $W_i = \{v_i\}$. In ogni round tutti processori spediranno a tutti gli altri processori l'insieme W e il processore i ricevendo W_j aggiornerà il proprio insieme ponendo $W_i = W_i \cup W_j$. Al round $f + 1$, ogni processore deciderà usando la seguente regola: se $|W_i| = 1$ allora il processore i decide v, dove v è l'unico elemento di W_i; se invece $|W_i| > 1$ allora il processore i deciderà v_0.

Algorithm 22: FLOODSET$_i$

Stato:
rounds $\in \mathbb{N}$, inizialmente 0
decision $\in V \cup \{unknown\}$, inizialmente *unknown*
$W \subseteq V$, inizialmente $\{v_i\}$

Generazione messaggi:
if *rounds* $\leq f$ **then**
  spedisci W agli altri processori

Computazione:
rounds := *rounds*+1
Sia X_j il messaggio in arrivo da j, per ogni j da cui un messaggo arriva
$W = W \cup \bigcup_j X_j$
if *rounds* $= f + 1$ **then**
  if $|W| = 1$ **then**
    decision $:= v$, dove $W = \{v\}$ **else**
      decision $:= v_0$

La Figura 22 mostra l'algoritmo FLOODSET. Non forniamo una prova formale della correttezza dell'algoritmo, ma dovrebbe essere abbastanza facile convincersi che dopo $f + 1$ rounds tutti i processori hanno il medesimo valore per W e quindi fanno tutti la stessa scelta raggiungendo il consenso. Infatti diversi valori di W sono possibili solo quando ci sono dei guasti ma poichè l'algoritmo viene eseguito per $f + 1$ round, sicuramente ci sarà un round senza guasti che garantirà l'uniformità della conoscenza di W.

Per comprendere meglio consideriamo il seguente esempio che è un caso limite in cui in ogni round un singolo processore si rompe facendo sì che la conoscenza non sia uniforme. Abbiamo un sistema di 5 processori e $f = 3$. La conoscenza iniziale dei processori è

$$\{v_1\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}$$

Nel primo round p_1 spedisce il proprio valore solo a p_2 e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel secondo round p_2 spedisce la propria conoscenza solo a p_3 e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel terzo round p_3 spedisce la propria conoscenza solo a p_4 e poi si guasta. Quindi la conoscenza dei processori diventa:

$$guasto, guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_2, v_3, v_4, v_5\},$$

Nel quarto round non ci possono essere più guasti per cui p_4 invierà il valore v_1 anche a p_5 e pertanto alla fine del round la conoscenza sarà:

$$guasto, guasto, guasto, \{v_1, v_2, v_3, v_4, v_5\}, \{v_1, v_2, v_3, v_4, v_5\}.$$

In pratica, quello che succede è che al primo round senza guasti la conoscenza diventa uniforme. Dopo $f+1$ coi deve necessariamente essere un tale round. L'esempio fatto prima è un caso limite in cui il round senza guasti è il round $f+1$.

È immediato vedere che l'algoritmo termina dopo $f+1$ round, e che il numero di messaggi spediti è $O((f+1)n^2))$. Occorre osservare che ogni messaggio deve specificare un insieme che può contenere fino a n elementi, quindi la reale complessità dei messaggi, supponendo di poter usare un numero fissato b di bit per rappresentare ogni valore, diventa $O((f+1)n^3b)$, in quanto gli insiemi contenuti nei messaggi possono avere fino a n valori e per ognuno dei valori abbiamo bisogno di b bit.

Vediamo adesso un approccio al problema ConsensoGS un po' più complicato. Chiameremo EIGStop l'algoritmo risultante. Questo approccio è molto oneroso per il caso dei guasti stop, per il quale l'algoritmo FloodSet risolve il problema in modo molto più efficiente. Tuttavia EIGStop sarà un utilissimo punto di partenza per risolvere il problema nel caso dei guasti bizantini. L'idea di base è che ogni processore comunichi a ogni altro processore, in ogni round, non solo tutto quello che sa, ma anche come lo ha saputo. All'inizio della computazione ogni processore conosce solo il proprio valore iniziale e quindi nel primo round può comunicare questo valore a tutti gli altri processori. Nel secondo round ogni processore, oltre al proprio valore iniziale, conosce anche i valori iniziali degli altri processori (almeno di quelli dai quali è arrivato un messaggio, visto che i processori possono rompersi). Tale informazione è arrivata direttamente dai processori che l'hanno prodotta: è il processore i a comiunicare v_i, il proprio valore iniziale, a tutti gli altri processori. Nei round successivi le informazioni possono essere propagate indirettamente nel senso che un processore potrà dire di aver saputo da un altro processore che altri processori avevano quei particolari valori iniziali. Ad esempio nel terzo round il processore k potrà sapere dal processore j che il valore iniziale del processore i è v_i. Nei round successivi la catena di "questo processore mi ha detto che" si allunga. Questo approccio è molto robusto in quanto l'informazione si diffonde nel modo più ampio e completo possibile; ovviamente è estremamente oneroso in termini di numero e grandezza dei messaggi.

Per formalizzare l'algoritmo faremo ricorso a una struttura dati che chiameremo albero EIG (Exponential Information Gathering), riportato nella Figura 6.3. Ogni processore manterrà una sua copia di questa struttura dati che verrà riempita con le informazioni che arriveranno dagli altri processori. Per rendere uniforme la descrizione assumeremo che un processore invii un messaggio anche a sé stesso; in pratica questo non avviene ma è ovviamente molto semplice simulare la spedizione di un tale messaggio.

La radice dell'albero del processore i, etichettata con λ, conterrà il valore iniziale v_i del processore i. I nodi del primo livello conterranno i valori iniziali degli altri processori così come ricevuti direttamente dagli altri processori, cioè il nodo con etichetta j conterrà il valore v_j che i ha ricevuto da j. I nodi del secondo livello, invece, conterranno tutte le

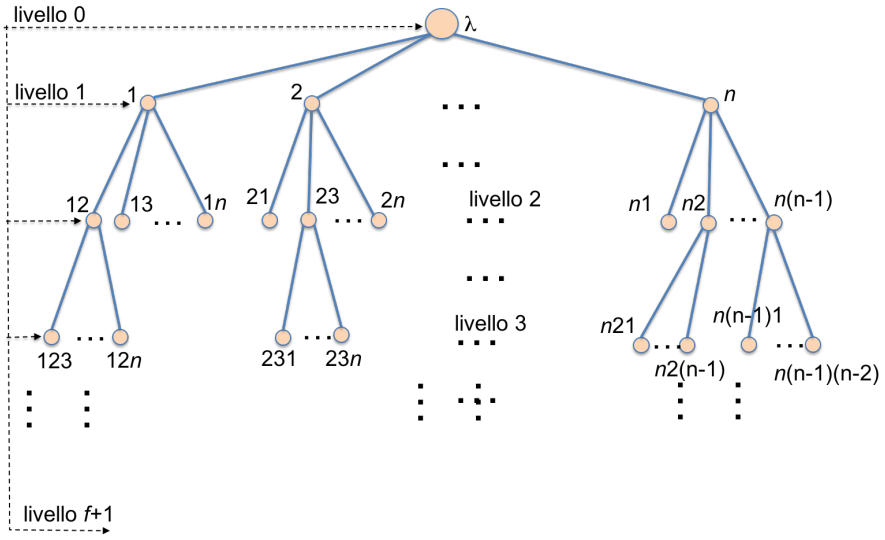

Figura 6.3: Albero EIG per n processori e f guasti.

informazioni ricevute indirettamente tramite un altro processore; l'etichetta del nodo specifica il "cammino" dell'informazione. Ad esempio il nodo con etichetta 12 conterrà il valore iniziale del processore 1 così come questo è stato comunicato prima da 1 a 2 e poi da 2 al processore i. In generale un nodo con etichetta $j_1 j_2 \dots j_k$ nell'albero del processore i al livello k, conterrà il valore iniziale del processore j_1 che il processore i ha ricevuto dal processore j_k che a sua volta lo ha ricevuto dal processore j_{k-1} che a sua volta lo ha ricevuto dal processore j_{k-2} e così via fino al processore j_1 che lo ha inviato a j_2 nel primo round. La Figura 6.4 mostra l'albero EIG per $n = 4$ e $f = 2$.

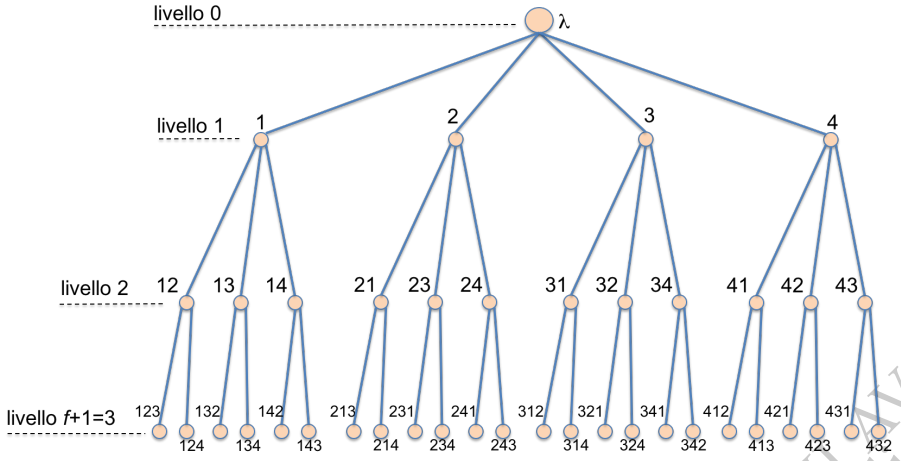

Figura 6.4: Albero EIG per $n = 4$ processori e $f = 2$ guasti.

L'algoritmo EIGSTOP, Algoritmo 23, funziona nel seguente modo. Per ogni etichetta x nell'albero EIG, ogni processore memorizza il valore corrispondente a tale etichetta nella variabile $val(x)$. All'inizio della computazione il processore i assegna il valore v_i alla variabile $val(\lambda)$.

Nel primo round il processore i invia $val(\lambda)$ a tutti gli altri processori e alla ricezione

di un valore v da j esegue l'istruzione $val(j) := v$.

Nei round successivi, cioè in ogni round k, con $2 \leq k \leq f+1$, il processore i spedisce a tutti gli altri processori il messaggio (x, v), dove x è un'etichetta del livello $k-1$ dell'albero EIG tale che x non contiene i e $v = val(x)$. Quindi il processore i, alla ricezione dei messaggi inviati dagli altri processori mnemorizza le informazioni in essi contenute. Cioè, se dal processore j arriva il messaggio (x, v), con x sequenza di indici che non contiene j, allora i esegue l'istruzione $val(xj) = v$.

Alla fine del round $f+1$, il processore i applica la regola di decisione: sia W l'insieme dei valori che sono memorizzati in tutti i nodi dell'albero EIG; se W ha cardinalità 1 allora il processore decide sull'unico valore presente in W, altrimenti decide un valore prestabilito v_0.

Algorithm 23: EIGSTOP$_i$

Stato:

Per ogni etichetta x dell'albero

val(x) $\in V \cup \{\perp\}$, inizialmente v_i per $x = \lambda$ e $\perp$ altrimenti

Generazione messaggi:

Round 1:

Invia $val(\lambda)$ a tutti (incluso i stesso)

Round k, $2 \leq k \leq f+1$:

Invia $(x, val(x))$ a tutti, per tutte le etichette x del livello $k-1$ con $i \notin x$.

Computazione:

Round 1:

if $v \in V$ arriva da j **then** $val(j) := v$ **else** $val(j) := \perp$

Round k, $2 \leq k \leq f+1$:

if (x, v) arriva da j, con $j \notin v$ **then** $val(xj) := v$ **else** $val(xj) := \perp$

if *siamo al round* $f+1$ **then**

Sia W l'insieme dei valori $val(j)$, eslcuso $\perp$

if $W = \{v\}$ **then** decidi v **else** decidi v_0

Non dovrebbe essere difficile vedere che i valori memorizzati nell'albero EIG sono quelli spiegati prima, cioè se il valore v viene memorizzato nel nodo con etichetta $i_1 \dots i_k$, con $1 \leq k \leq f+1$, allora significa che il processore i_k ha comunicato nel round k al processore i che il processore i_{k-1} ha comunicato nel round $k-1$ al processore i_k che il processore i_{k-2} ha comunicato nel round $k-2$ al processore i_{k-1} che ... il processore i_1 ha comunicato nel round 1 al processore i_2 che il valore iniziale di i_1 è v. Inoltre se il valore assegnato a un nodo con etichetta $i_1 \dots i_k$ è `null`, allora vuol dire che nella catena di messaggi da i_1 a i a un certo punto nessun messaggio è stato spedito da i_j a i_{j+1}, $j = 1, \dots, k-1$ nel round j o da i_k ad i nel round k.

Facciamo un esempio per rendere chiaro l'algoritmo. Consideriamo un sistema con $n = 3$ processori ed $f = 1$. La Figura 6.5 mostra gli alberi EIG dei 3 processori quando non si verificano guasti. I valori iniziali v_1, v_2 e v_3 dei 3 processori vengono correttamente propagati su tutti i messaggi e alla fine del secondo round tutti i processori

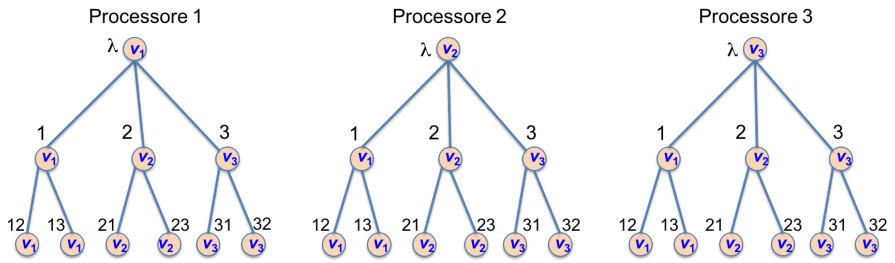

Figura 6.5: Alberi EIG con valori per un'esecuzione con $n = 3$ e $f = 1$.

avranno $W = \{v_1, v_2, v_3\}$.

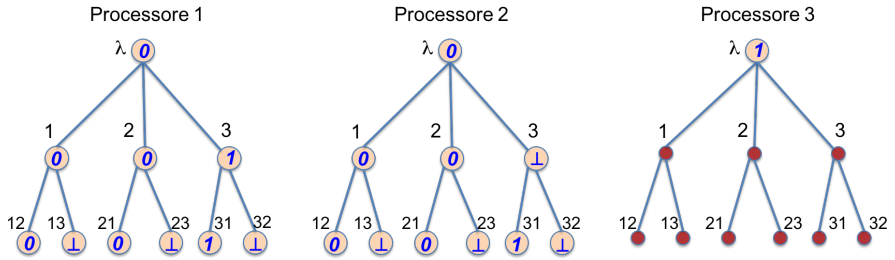

Figura 6.6: Alberi EIG con valori per $n = 3$ e $f = 1$, con un'esecuzione in cui il processore 3 si guasta subito.

La Figura 6.6 mostra gli alberi EIG dei 3 processori in una esecuzione in cui il processore 3 si guasta subito dopo aver spedito il suo valore iniziale. I valori iniziali dei processori sono 0, 0 e 1. Alla fine del secondo round i processori 1 e 2 avranno $W = \{0, 1\}$.

Proviamo adesso che l'algoritmo è corretto. Iniziamo con il provare che se due processori non si guastano, allora nel momento della decisione avranno lo stesso valore per W.

Lemma 6.4.1 *Se i processori i e j sono entrambi non guasti, allora dopo aver ricevuto i messaggi del round $f + 1$, si ha $W_i = W_j$.*

Dimostrazione. Forniamo una prova informale; il lettore interessato può trovare una prova più dettagliata in L96 [26]. Sserviamo innanzitutto che l'insieme W è l'insieme dei valori iniziali di cui il processore è venuto a conoscenza durante l'algoritmo. Inoltre, poichè i processori i e j sono entrambi non guasti ogni messaggio che inviano arriva all'altro. Ovviamente sia i che j conosceranno il valore iniziale dell'altro in quanto entrambi lo inviano nel primo round. Che possiamo dire della loro conoscenza del valore iniziale v_k di un altro processore k? Se k non è guasto, allora sicuramente v_k comparirà nell'albero di i e j già dal primo round. Se k invece si guasta potrebbe non comparire subito negli alberi di i e j. Infatti k potrebbe essere riuscito a inviare un messaggio a un altro processore k' che successivamente si è guastato, che a sua volta, nel round successivo è riuscito a inviare la conoscenza di v_k a un solo altro processore k'', e così via. Tuttavia, poichè ci sono f guasti questa sequenza di scenari con informazioni discordanti (solo un processore, o più in generale alcuni processori, conoscono il valore v_k) non può durare più di f round. Per cui, o il valore di v_k viene "perso" nel senso che

nessuno dei processori non guasti lo riceve mai, oppure, mal che vada, nel round $f+1$ verrà comunicato a tutti i processori non guasti, quindi inclusi i e j. □

In realtà quanto appena detto è una reiterazione dell'argomentazione usata per FloodSet. Infatti EIGStop, relativamente ai valori iniziali degli altri processori, mantiene esattamente le stesse informazioni di FloodSet. Proviamo ora che l'algoritmo è corretto.

Teorema 6.4.2 *L'algoritmo* EIGStop *permette di raggiungere il consenso in un sistema sincrono con un massimo di f guasti stop.*

Dimostrazione. La proprietà di terminazione è ovviamente verificata: tutti i processori producono un output al round $f+1$.

Vediamo la propietà di validità. Supponiamo che tutti i valori iniziali siano pari a v. Allora v è l'unico valore che può essere inserito in W. Poichè ogni processore non guasto inserisce v nel proprio insieme W, si ha che nel round $f+1$, tutti i processori hanno $W = \{v\}$. Quindi la decisione sarà v.

Infine la proprietà di accordo deriva direttamente dal Lemma 6.4.1 che garantisce che $W_i = W_j$ per tutti i processori non guasti. □

L'algoritmo EIGStop termina in $f+1$ round, quindi la complessità di tempo è $O(f+1)$ ed utilizza $O((f+1)n^2)$ messaggi; tuttavia i messaggi hanno lunghezza variabile in quanto ogni processore deve comunicare ad ogni altro processore molte coppie di etichette e valori; le etichette sono costituite da tutti i possibili cammini tramite i quali si è ricevuto il valore e hanno lunghezza crescente con il numero di round. Pertanto, il numero di *bit* necessari per spedire i messaggi è esponenziale nel numero di guasti: $O(n^{f+1}b)$, dove b è il numero di bit necessari a rappresentare un indice. Un miglioramento si può ottenere osservando che la conoscenza esatta di W è necessaria solo quando $|W| = 1$, nel qual caso occorre sapere quale è il singolo valore presente in W per poter decidere. Pertanto si può usare una variante dell'algoritmo in cui ogni processore effettua solo 2 spedizioni: una al round 1 con il proprio valore iniziale e la seconda al round r, con $2 \leq r \leq f+1$, quando il processore viene a conoscenza di un'altro valore iniziale diverso dal suo valore iniziale. I dettagli vengono lasciati come esercizio.

Commento. L'algoritmo EIGStop funziona per guasti stop. Lo abbiamo introdotto in quanto ci sarà utile per fonire un nuovo algoritmo, apportando delle modifiche a EIGStop, capace di funzionare anche con guasti bizantini. Tuttavia, EIGStop funziona anche con una interessante restrizione dei guasti bizantini: se la comunicazione può essere autenticata, usando ad esempio la firma digitale, allora l'algoritmo EIGStop può tollerare anche guasti bizantini.

6.4.2 *Con guasti bizantini, algoritmo* EIGByz

Consideriamo adesso il caso di guasti bizantini. Un processore guasto può esibire un comportamento arbitrario. Per guasti reali questo significa tipicamente un comportamento casuale. I guasti bizantini modellano però un ben più problematico scenario: un processore potrebbe essere violato da hackers che controllano il suo comportamento in

modo arbitrario. In questa ipotesi il comportamento può disturbare la computazione in maniera intelligente per scopi fraudolenti. In altre parole, da un guasto bizantino ci si deve aspettare il peggio.

Prima di procedere dobbiamo modificare leggermente le proprietà di accordo e validità richieste dal problema del consenso.

Problema ConsensoGB**:** Consideriamo un sistema distribuito di n processori, che comunicano tramite scambio di messaggi su un canale affidabile. I processori possono rompersi con dei guasti bizantini. Ogni processore ha un valore di input $v_i \in V$ e i processori devono decidere un valore di output in modo tale da soddisfare le seguenti 3 proprietà:

- (Accordo) Tutti i processori non guasti danno in output lo stesso valore.
- (Validità) Se tutti i processori non guasti iniziano con uno stesso valore $v \in V$, allora v è l'unico possibile output.
- (Terminazione) Tutti i processori non guasti decidono.

Le modifiche introdotte nel problema ConsensoGB riguardano il fatto che le proprietà di accordo e validità devono essere soddisfatte solo dai processori non guasti: poichè i guasti sono bizantini, non c'è modo di "forzare" un processore guasto a fare delle scelte né tantomeno a fargli rivelare il vero valore di input.

Algoritmo EIGByz**.** L'algoritmo EIGStop che abbiamo visto in precedenza può essere adattato al caso di guasti bizantini nel seguente modo. I processori propagano i valori inziali per $f+1$ rounds come in EIGStop. Messaggi non conformi vengono semplicemente ignorati e il destinatario si comporta come se il messaggio non fosse mai arrivato. Alla fine degli $f+1$ rounds i valori dell'albero EIG che sono nulli vengono rimpiazzati dal valore di default v_0. Per prendere una decisione, ogni processo visita il proprio albero EIG partendo dalle foglie per arrivare alla radice e durante la visita calcola un nuovo valore, *newval*, per ogni nodo dell'albero come segue. Per ogni foglia con etichetta x, $newval(x) := val(x)$. Per ogni nodo interno con etichetta x, $newval(x)$ è definito come il *newval* di una maggioranza stretta dei figli del nodo x, cioè si esaminano i valori *newval* dei figli di x, e se c'è un valore v presente in una maggioranza stretta si pone $newval(x) := v$. Se non esiste un valore in una maggioranza stretta, allora si pone $newval(x) := v_0$.

La Figura 6.7 mostra un esempio. I valori *newval* delle foglie sono uguali ai valori *val* calcolati da EIGStop quando non sono $\perp$; i valori $\perp$ vengono sostituiti da v_0. Per i nodi interni invece vengono calcolati con la regola descritta sopra. Ad esempio per il nodo x_1 ci sono 4 figli con $newval = 1$ e 3 figli con $newval = 0$, quindi $newval(x_1) = 1$. Per i nodi x_2 e x_3 invece c'è una maggioranza di figli con $newval = 0$ quindi si ha che $newval(x_2) = 0$ e $newval(x_3) = 0$. Procedendo verso la radice, il nodo y ha una maggioranza di figli con $newval = 0$ quindi si ha che $newval(y) = 0$. La decisione finale è $newval(\lambda)$, cioè il *newval* della radice dell'albero EIG.

L'algoritmo EIGByz funziona se $n > 3f$. La prova di correttezza dell'algoritmo EIGByz si basa sul fatto che, poichè $n > 3f$, f processori guasti non possono produrre

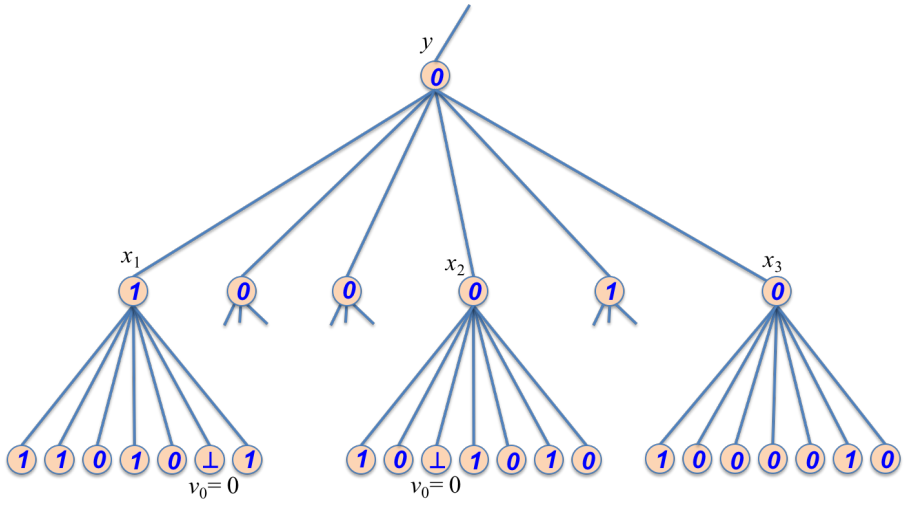

Figura 6.7: Valori *newval* in EIGBYZ.

troppi valori discordanti e quindi le informazioni diffuse dagli $n - f$ processori non guasti sono sufficienti a prendere la decisione corretta (dettagli sulla prova di correttezza possono essere trovati in L96 [26]). L'algoritmo EIGBYZ ha la stessa complessità di EIGSTOP: $f + 1$ rounds, $O((f + 1)n^2)$ messaggi e $O(n^{f+1}b)$ bit di comunicazione.

6.4.3 *Limite inferiore $n > 3f$.*

Abbiamo visto che l'algoritmo EIGBYZ risolve il problema del consenso per guasti bizantini. Esistono anche altri algoritmi come l'algoritmo TURPINCOAN, che funziona per il caso in cui l'insieme di valori iniziali è $V = \{0, 1\}$. Un altro algoritmo usa l'algoritmo TURPINCOAN come subroutine per risolvere il problema per un insieme di valori iniziali arbitrario (si veda il Capitolo 6 di L96 [26] per ulteriori dettagli su questi algoritmi). Tutti questi algoritmi hanno una caratteristica in comune: richiedono che il numero di processori n del sistema sia strettamente più grande del triplo del numero f di possibili guasti, cioè deve essere $n > 3f$.

Non è un caso. Infatti per $n \leq 3f$ il problema non può essere risolto. Intuitivamente questo significa che f guasti bizantini, quindi f processori che si comportano in modo da disturbare la computazione possono "imbrogliare" fino a $2f$ processori che funzionano bene. Prima di presentare la prova, diamo un esempio che ci dà una intuzione della limitazione.

Consideriamo il caso particolore di un sistema con $n = 3$ processori e $f = 1$. Supponiamo, per semplicità, che i processori decidano dopo 2 round e che nel primo round ogni processore spedisca il proprio valore iniziale mentre nel secondo round ogni processore spedisce il valore ricevuto nel primo round, sepcificando anche da chi lo ha ricevuto. Queste limitazioni non sono "forti" in quanto i processori si stanno scambiando tutte le informazioni a loro disposizione e per fare ciò bastano due round (in ogni caso con questo esempio stiamo solo dando un'intuizione del perché il problema non può essere risolto quando $n \leq 3f$). Consideriamo le seguenti esecuzioni:

1. Esecuzione α_1. I valori iniziali dei 3 processori p_1, p_2 e p_3 sono, rispettivamente, 1, 1 e 0. I processori p_1 e p_2 funzionano correttamente mentre p_3 è guasto. Nel primo round tutti i processori si comportano seguendo l'algoritmo e quindi spediscono il proprio valore iniziale. Nel secondo round i processori non guasti, p_1 e p_2, spediscono correttamente ciò che hanno appreso nel primo round, mentre il processore guasto, p_3 si comporta in modo fraudolento e spedisce un messaggio corretto a p_1 ed un messaggio *falso* a p_1. La Figura 6.8 mostra i messaggi spediti. Per la proprietà di validità i processori corretti devono decidere 1.

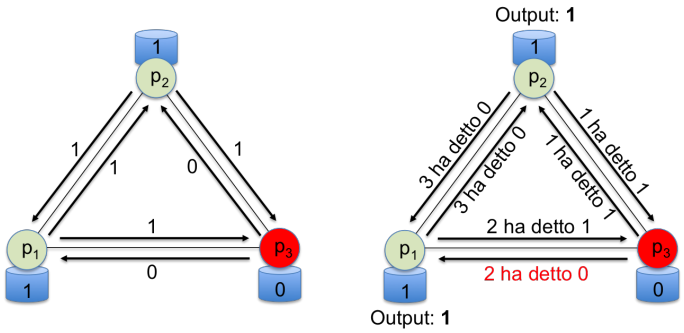

Figura 6.8: Esecuzione α_1

2. Esecuzione α_2. Questa esecuzione è simile all'esecuzione α_1: questa volta è il processore p_1 ad essere guasto e i valori iniziali sono 0 per p_2 e p_3 e 1 per p_1. Il processore guasto si comporta come nel caso precedente: nel primo round spedisce correttamente il proprio valore iniziale mentre nel secondo round manda un messaggio veritiero a p_2 ed un messaggio falso a p_3. La Figura 6.9 mostra i messaggi spediti. Per la proprietà di validità i processori corretti quest volta devono decidere 0.

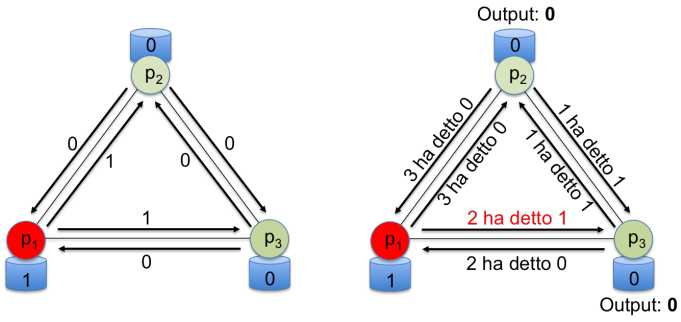

Figura 6.9: Esecuzione α_2

3. Esecuzione α_3. Consideriamo infine una terza esecuzione α_3 in cui il processore p_2 è guasto, mentre p_1 e p_3 funzionano correttamente. I valori iniziali di p_1 e p_3 sono rispettivamente 1 e 0. Questa volta il processore guasto spedisce messaggi contradditori nel primo round (a p_1 invia 0 e a p_3 invia 1) e messaggi veritieri nel secondo round. La Figura 6.10 mostra i messaggi spediti.

Le esecuzioni α_1 e α_3 sono indistinguibili per il processore p_1: esso infatti riceve e spedisce gli stessi messaggi nelle due esecuzioni. Poichè p_1 decide 1 in α_1, concludiamo che deve decidere 1 anche in α_3.

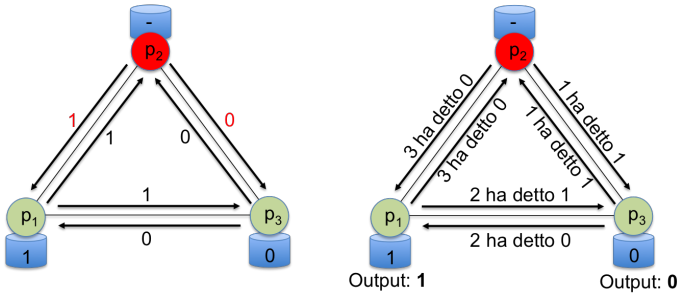

Figura 6.10: Esecuzione α_3

Analogamente, le esecuzioni α_2 e α_3 sono indistinguibili per il processore p_3: esso infatti riceve e spedisce gli stessi messaggi nelle due esecuzioni. Poichè p_3 decide 0 in α_0, concludiamo che deve decidere 0 anche in α_3.

Questo significa che la proprietà di accordo è violata nell'esecuzione α_3.

L'esempio che abbiamo fornito non costituisce una prova della limitazione $n > 3f$, ma fornisce solo l'intuizione del perché tale limitazione è necessaria. Non è una prova in quanto abbiamo stabilito uno specifico tipo di algoritmo: utilizza solo 2 round e manda uno specifico pattern di messaggi. Quindi, da quanto detto, possiamo concludere solo che non esistono algoritmi di quel tipo.

Il seguente lemma generalizza l'argomentazione usata nell'esempio fornendo quindi una prova del fatto che con soli 3 processori anche un solo guasto rendo il problema non risolvibile.

Lemma 6.4.3 *Tre processori non possono risolvere il problema del consenso se uno di essi può comportarsi in modo bizantino.*

DIMOSTRAZIONE. Per contraddizione assumiamo che esista un algortimo A che risolve il problema in un sistema con tre processori, 1, 2 e 3 nell'ipotesi che uno di essi possa essere guasto in modo bizantino. Nella prova useremo 6 processori che verranno utilizzati in gruppi di 3 per poter far girare l'algoritmo A. Consideriamo l'esecuzione dell'algoritmo A nei seguenti 3 casi:

1. Processori p_1, p_2, p_3 con p_1 guasto e input di p_2 e p_3 pari a 0. Chiamiamo α_1 l'esecuzione che ne deriva.
2. Processori p'_1, p'_2, p'_3 con p'_3 guasto e input di p'_1 e p'_2 pari a 1. Chiamiamo α_2 l'esecuzione che ne deriva.
3. Processori p'_1, p_2, p_3 con p_2 guasto, input di p'_1 pari a 1 e input di p_3 pari a 0. Chiamiamo α_2 l'esecuzione che ne deriva.

Poichè l'algoritmo A risolve il problema anche se un processore è guasto, si ha che in α_1, per la condizione di validità, i processori p_2 e p_3 decidono 0 e in α_2, sempre per la condizione di validità, i processori p'_1 e p'_2 decidono 1.

Nell'esecuzione α_3 non sappiamo quale valore viene deciso in quanto i processori partono da input diversi quindi non possiamo sfruttare la condizione di validità. Tuttavia per la proprietà di accordo devono decidere o entrambi 0 o entrambi 1; mostreremo ora che questo non è possibile.

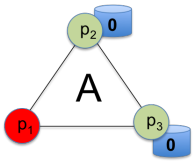

Figura 6.11: Esecuzione α_1

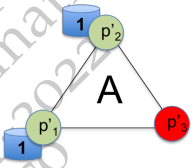

Figura 6.12: Esecuzione α_2

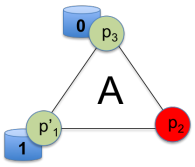

Figura 6.13: Esecuzione α_3

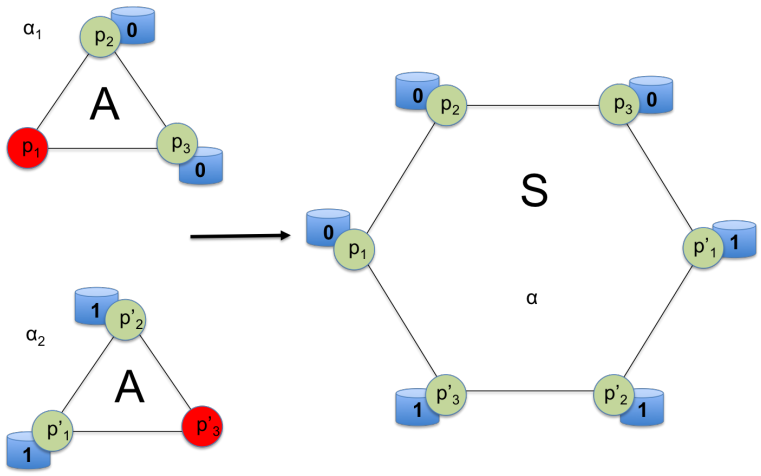

Figura 6.14: Sistema S per il Lemma 6.4.3

Costruiamo un sistema S usando due "copie" di A, come mostrato nella Figura 6.14. Nel sistema S i processori si comportano seguendo l'algoritmo A ma i processori p_1 e p_3 comunicheranno con gli omologhi dell'altra copia: l'algoritmo deve continuare a funzionare in quanto abbiamo supposto che possa tollerare un guasto bizantino e nel sistema S è come se ognuna delle due copie di A si ritrovi in una situazione reale in cui un processore (p_1 per una copia e p_3 per l'altra) sia guasto.

Osservazione: cosa è S? Non è un sistema dove risolviamo il problema del consenso usando l'algoritmo A; infatti A risolve il problema del consensi in un sistema con 3 processori non con 6. S è semplicemente un sistema distribuito sincrono con 6 processori in cui ogni processore si comporta seguendo l'algoritmo A nel modo descritto in precedenza. Il sistema S non risolve il problema del consenso! Cosa fa il sistema S non ci interessa nemmeno. Assumeremo che in S non si verifichino guasti e chiameremo α l'esecuzione dell'algoritmo A in S.

Per i processori p_2 e p_3 l'esecuzione α_1 è indistinguibile da α, mentre per i processori p'_1 e p'_2 l'esecuzione α_2 è indistingubile da α. Poichè in α_1 i processori p_2 e p_3 decidono 0 anche in α p_2 e p_3 decidono 0. Analogamente, poichè in α_2 i processori p'_1 e p'_2 decidono 1 anche in α p'_1 e p'_2 decidono 1.

Infine consideriamo le esecuzioni α e α_3 dal punto di vista dei processori p'_1 e p_3. Per tali processori α e α_3 sono indistinguibili e poichè in α decidono rispettivamente 0 e 1 si ha che anche in α_3 i processori p'_1 e p_3 decidono rispettivamente 0 e 1. Questo significa che l'algoritmo A non risolve il problema del consenso nell'esecuziuone α_3. □

Il lemma precedente mostra che non è possibile risolvere il problema se $f \geq n/3$ nel caso specifico di $n = 3$. Ovviamente questo lascerebbe la possibilità che per valori maggiori di n la situazioni cambi. Il seguente teorema mostra che il risultato è valido per ogni n.

Teorema 6.4.4 *Il problema del consenso in un sistema di n processori non può essere risolto se si possono verificare f guasti bizantini con* $n \leq 3f$.

DIMOSTRAZIONE. Il caso $n = 2$ è semplice in quanto, informalmente, ogni processore

deve decidere da solo perché l'altro protrebbe essere guasto. Più formalmente potremmo procedere per assurdo assumendo che esita un algoritmo A. Consideriamo l'esecuzione di A quando p_1 e p_2 inziano con rispettivamente 0 e 1 e nessuno dei due è guasto. Chiamiamo α tale esecuzione. In α A dovrà far prendere una decisione; senza perdere in generalità assumimamo che sia 0; se fosse 1 si potremmo precedere in modo simmetrico (e la prova potrebbe essere estesa anche al caso non binario). Ora consideriamo una nuova esecuzione α' di A in cui p_1 è guasto e p_2 funziona correttamente e ha come input 1. Il processore bizantino p_1 si comporta esattamente come in α. Dunque p_2 non vede nessuna differenza fra α e α' pertanto, visto che in α ha deciso 0 farà lo stesso in α' violando la proprietà di validità. Pertanto l'algoritmo A non può esistere e quindi il problema non può essere risolto.

Consideriamo ora il caso $3 \leq n \leq 3f$. Anche in questo caso procediamo per contraddizione assumendo che esista un algoritmo A che risolve il problema. Mostreremo come trasformare A in un algoritmo B per un sistema con 3 processori b_1, b_2 e b_3 in cui uno può essere guasto. Ognuno dei 3 processori usati in B simulerà circa un terzo dei processori usati in A. Cioè partizioniamo gli n processori in 3 insiemi I_1, I_2 e I_3 ognuno di grandezza al più f. Il processore b_i simulerà l'insieme I_i nel modo seguente: il processore b_i tiene traccia degli stati di tutti i processori in I_i, simulando tutti i passi e le spedizioni dei messaggi dei processori in I_i. I messaggi saranno spediti al processore che simula il destinatario. Se uno qualsiasi dei processori in I_i decide un valore v allora anche b_i decide v (se ci sono più valori allora b_i può decidere uno qualsiasi di tali valori).

L'algoritmo B risolve il problema del consenso bizantino in un sistema con 3 processori. Consideriamo infatti un'esecuzione α in cui al massimo un processore b_i è guasto e sia α' la corrispondente esecuzione di A nel sistema simulato. Poichè b_i simula al massimo f processori, si ha che in α' ci sono al massimo f guasti. Poichè A risolve il problema del consenso bizantino con al massimo f guasti, si ha che in α' vengono soddisfate le proprietà di accordo, validità e terminazione. Le stesse valgono anche in α. Infatti sia b_i un processore non guasto pertanto b_i simula i processori I_i che essendo non guasti, decideranno; quindi anche b_i decide.

Per la proprietà di validità, assumiamo che tutti i processori non guasti di B abbiano come input lo stesso valore v. Ovviamente anche i processori di A iniziano con lo stesso valore v. Pertanto per la validità in α' si ha che tutti processori non guasti di A decidono v. Questo significa anche che tutti i processori non guasti di B decidono v.

Infine per la proprietà di accordo, siano b_i e b_j due processori non guasti di B. Tali processori simulano solo processori non guasti di A. Questo significa che i processori in I_i e in I_j decidono lo stesso valore. Quindi anche b_i e b_j decidono lo stesso valore.

Pertanto siamo riusciti a costruire un algoritmo che risolve il problema del consenso nel caso di 3 processori ed un guasto bizantino. Ma questo contraddice il Lemma 6.4.3.
□

6.5 Consenso in sistemi asincroni con guasti stop

Consideriamo adesso il problema del consenso in sistemi asincroni. In un sistema asincrono i processori possono avere velocità diverse e, almeno dal punto di vista teorico,

estremamente diverse al punto che diventa difficile, anzi impossibile, distinguere un processore molto lento da un processore guasto. Analogamente anche i tempi di consegna dei messaggi sono variabili e non possiamo sapere se un messaggio è stato perso oppure è ancora in transito. In un sistema asincrono, in un certo senso, il tempo non esiste, o più precisamente non può essere misurato. In realtà, ogni componente del sistema può misurare il tempo, ma lo fa in modo indipendente e quindi ogni componente misura il tempo in modo diverso. L'assunzione è che queste diversità possono essere enormi, fino al caso limite di infinito che corrisponde a componenti guaste. Ovviamente questo crea una grossa difficoltà in quanto gli algoritmi dovranno rinunciare ad usare informazioni relative alla misura del tempo. Si noti come questo sia in contrapposizione a ciò che succede in un sistema sincrono, nel quale è come se esistesse una misurazione del tempo uguale per tutte le componenti (processori e canali) e ogni conponente fosse in grado di eseguire delle istruzioni in perfetta sincronia, cioè nello stesso istante, con tutte le altre componenti del sistema.

Consideriamo guasti di tipo stop (porblema ConsensoGS). Ricordiamo la definizione del problema. *Accordo*: in una qualsiasi esecuzione, tutti le decisioni sono uguali. *Validità*: in una qualsiasi esecuzione, se tutti i valori iniziali sono uguali a v, allora v è l'unico possibile valore per le decisioni. *Terminazione*: in una qualsiasi esecuzione in cui si possono verificare al massimo f guasti stop, tutti i processori non guasti decidono.

6.5.1 *Impossibilità per algoritmi deterministici*

Il seguente risultato è una delle pietre miliari nella teoria dei sistemi distribuiti, ed è noto come teorema dell'impossibilità FLP, dai nomi[2] dei ricercatori che lo hanno dimostrato.

[2] Fischer, Lynch, Patterson, 1985 [13].

Teorema 6.5.1 *Non è possibile risolvere il problema del consenso in un sistema distribuito asincrono anche in presenza della possibilità di un solo guasto.*

Prima di dimostrare il teorema, specifichiamo meglio il formalismo che utilizzeremo.

Assumiamo che $V = \{0, 1\}$. Consideriamo un sistema con n processori, $p_1, p_2, \ldots, p_n$. Ogni processore inizia la computazione con un bit di input e deve produrre un bit di output in accordo alle 3 proprietà della definzione del problema del consenso.

I processori comunicano inviando dei messaggi. Possiamo modellare la comunicazione con un buffer globale e affidabile: i messaggi non vengono persi, anche se ogni messaggio può impiegare un tempo arbitrario per arrivare alla destinazione. Il buffer quindi è semplicemente un multinsieme di coppie

$$\mathit{buffer} = \{(p, m) |\ m \text{ è un messaggio per } p\}$$

dove m è un messaggio e p è un processore, quello che riceverà il messaggio. L'invio e la ricezione dei messaggio sono modellati da due operazioni:

- $\mathit{send}(m, p)$: eseguita da un processore q, indica la spedizione del messaggio m destinato a p. L'effetto è quello di inserire l'elemento (m, p) nel *buffer*.

- *receive*(p): eseguita da un processore p, indica la ricezione di un messaggio da parte di p. L'effetto è quello di cancellare un elemento arbitrario (m, p) dal *buffer* e di consegnare m a p. Come caso particolare questa operazione può restituire il valore speciale `null` che indica l'assenza di un messaggio per p. Questo modella l'asincronia del sistema: anche se p chiede di ricevere un messaggio, il messaggio potrebbe non esistere, oppure essere ancora "in transito".

Una *configurazione* del sistema è lo stato di ogni processore più il contenuto del *buffer*. Nella configurazione iniziale, il *buffer* è vuoto e ogni processore ha il bit iniziale settato.

La computazione procede in *passi* che portano il sistema da una configurazione a quella successiva. Ogni passo viene eseguito da un processore p che può utilizzare l'operazione *receive*(p) per ricevere un messaggio $m \in M \cup \{\texttt{null}\}$, eseguire una computazione locale cambiando il proprio stato ed eventualmente spedire con l'operazione *send*(m, q) un numero finito di messaggi agli altri processori.

Un *evento* $e = (m, p)$ rappresenta la ricezione del messaggio m da parte di p. Una *esecuzione parziale* è una sequenza σ, finita o infinita, di eventi a partire da un configurazione c. Indicheremo con $\sigma(c)$ la configurazione risultante e diremo che σ è *applicabile* a c. Una *esecuzione* è una esecuzione parziale che inizia da una configurazione iniziale.

Lemma 6.5.2 *Consideriamo una configurazione c e le esecuzioni parziali σ_1 e σ_2 applicabili a c. Sia $c_1 = \sigma_1(c)$ e $c_2 = \sigma_2(c)$. Se l'insieme dei processori che eseguono passi in σ_1 e σ_2 sono disgiunti, allora $\sigma_2(c_1) = \sigma_1(c_2)$.*

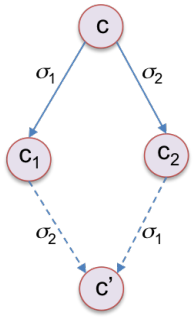

Figura 6.15: Lemma 6.5.2

DIMOSTRAZIONE. Il lemma dice che applicando entrambe le esecuzioni parziali si arriva sempre alla stessa configurazione indipentemente dall'ordine in cui le si applica. (Figura 6.15). Per provarlo, notiamo che poichè sia σ_1 che σ_2 sono entrambe esecuzioni parziali applicabili in modo indipendente a c e l'insieme dei processori che eseguono passi in σ_1 che σ_2 sono disgiunti, non è possibile "spedire" un messaggio in un passo di σ_1 e riceverlo in un passo di σ_2 o viceversa.

Dunque, poichè gli insiemi di processori sono disgiunti, si ha che i passi applicati nelle due esecuzioni non interagiscono fra loro.

Questo significa che in c_1 è possibile applicare σ_2 e in c_2 è possibile applicare σ_1; inoltre, poichè gli insiemi di processori sono disgiunti, eseguire prima σ_1 e poi σ_2 non crea nessuna differenza rispetto a eseguire prima σ_2 e poi σ_1 (l'ordine è rilevante solo se uno stesso processore esegue passi sia in σ_1 che σ_2). □

Diremo che una configurazione è 0-*valente* se da quella configurazione tutte le possibili esecuzioni parziali portano a un risultato finale di 0. Analogamente diremo che una configurazione è 1-*valente* se da quella configurazione tutte le possibili esecuzioni parziali portano a un risultato finale di 1. Inoltre diremo che una configurazione è *univalente* se è 0-valente oppure 1-valente. Una configurazione *bivalente* è una configurazione che può portare sia a una decisione di 0 sia a una decisione di 1. Si noti che quando si raggiunge una configurazione univalente, di fatto si è raggiunto la decisione finale, quindi in pratica l'algoritmo, almeno nella sua sostanza, è terminato. Da una configurazione bivalente invece sono necessari altri passi per poter raggiungere una decisione.

Per provare che il problema non può essere risolto, assumeremo che, per assurdo, esiste un algoritmo A che risolve il problema. Mostreremo che esistono dei casi in cui A non termina mai. Per fare ciò mostreremo che esiste una configurazione iniziale bivalente e che da tale configurazione è possibile eseguire passi senza mai raggiungere una configurazione univalente.

Lemma 6.5.3 *Esiste una configurazione iniziale bivalente.*

Dimostrazione. Assumiamo per assurdo che tutte le configurazioni iniziali siano univalenti. Consideriamo le seguenti configurazioni iniziali c_i, per $i = 0, 1, 2, \ldots, n$. Nella configurazione c_0 tutti gli input sono 0. Per la condizione di validità c_0 deve essere 0-valente. La configurazione c_i differisce da c_{i-1} solo per l'input del processore i-esimo: tale input è 0 in c_{i-1} e 1 in c_i. Pertanto c_n è una configurazione inizale in cui tutti gli input sono 1. Per la condizione di validità c_n deve essere 1-valente.

Poichè c_0 è 0-valente e c_n è 1-valente, deve esistere un indice $i \in [0, n-1]$ tale che c_i è 0-valente e c_{i+1} è 1-valente.

Consideriamo l'esecuzione σ_i che inizia da c_i nella quale il processore p_{i+1} si guasta immmediatamente senza eseguire nessun passo. L'input di p_{i+1}, dunque, non potrà essere noto a nessuno.

Consideriamo anche l'eseuzione σ_{i+1} che inizia da c_{i+1} nella quale il processore p_{i+1} si guasta immmediatamente senza eseguire nessun passo. Anche in questo caso l'input di p_{i+1} non potrà essere noto a nessuno. Tale bit è però l'unica differenza fra σ_i e σ_{i+1}; il fatto che nessuno lo conosca significa che le due esecuzioni sono indistinguibile per tutti i processori (tranne che per p_{i+1}, che però è guasto).

Pertanto tutti i processori non guasti decideranno lo stesso valore sia σ_i che in σ_{i+1}. Ma questo non è possibile in quanto σ_i è 0-valente e σ_{i+1} è 1-valente. □

Lemma 6.5.4 *Consideriamo una configurazione c raggiungibile e bivalente e sia $e = (p, m)$ un evento applicabile in c. Sia C l'insieme di configurazioni raggiungibili da c senza applicare e. Sia D l'insieme di configurazioni raggiungibili da una qualsisi configurazione di C applicando e (si veda la Figura 6.16). L'insieme D contiene una configurazione bivalente.*

Dimostrazione. Procediamo per assurdo assumendo che D non contenga nessuna configurazione bivalente, quindi che D contenga solo configurazioni univalenti.

Osserviamo che se l'insieme C non cotiene altre configurazioni oltre a c, si avrebbe che l'unic evento applicabile a c è l'evento e. Ma poichè possiamo far rompere il processore p che è l'unico che può applicare e, c, avremmo trovato una configurazione che non permette di proseguire l'esecuzione. Essendo c bivalente si avrebbe un'esecuzione in cui non si decide. Per cui, in questo, caso il problema non è risolvibile. Pertanto consideriamo il caso in cui oltre a e ci sono altri eventi $e' \neq e$ applicabili a C.

Poichè c è bivalente, esistono due configurazioni $\overline{c_0}$ e $\overline{c_1}$, raggiungibili da c, rispettivamente 0-valente e 1-valente. Da ciò segue anche che esistono due configurazioni $c_0 \in C$ e $d_0 = e(c_0)$ con d_0 0-valente. Infatti se $\overline{c_0} \in C$ allora basta prendere $c_0 = \overline{c_0}$ e si ottiene il risultato voluto. Consideriamo il caso $\overline{c_0} \notin C$. Poichè siamo arrivati a $\overline{c_0}$ partendo da c, visto che $\overline{c_0}$ non è in C, abbiamo usato e per raggiungere $\overline{c_0}$. Questo implica che siamo passati per una configurazione $c_0 \in C$ tale che $d_0 = e(c_0) \in D$. Ma

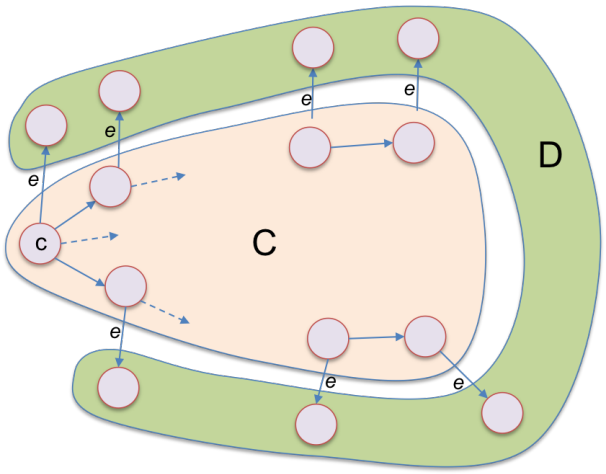

Figura 6.16: Lemma 6.5.4. Le esecuzioni da c a C non contengono $e = (p, m)$.

sappiamo anche che D contiene solo configurazioni univalenti; pertanto poichè da d_0 raggiungiamo $\overline{c_0}$ che è 0-valente, anche d_0 deve essere 0-valente.

In modo analogo possiamo provare che esistono due configurazioni $c_1 \in C$ e $d_1 = e(c_1)$ con d_1 1-valente.

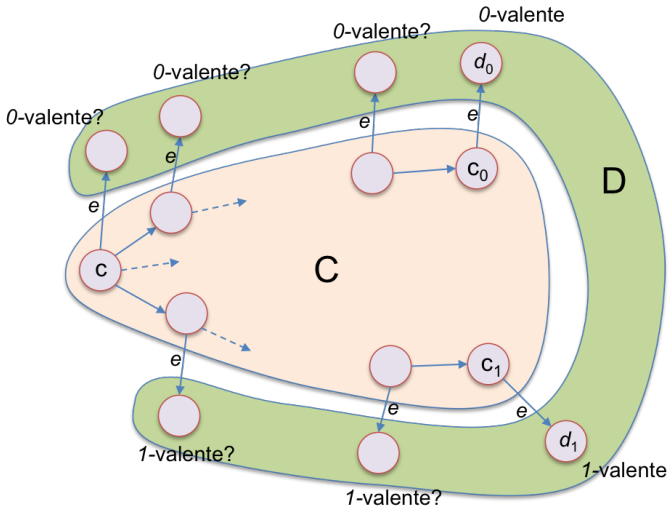

Figura 6.17: Configurazioni c_0, d_0, c_1 e d_1.

Dunque, per ipotesi (assurda) D contiene solo configurazioni univalenti e abbiamo visto che non possono essere tutte dello stesso tipo, cioè D contiene sia configurazioni 0-valenti che configurazioni 1-valenti. Il prossimo passo è quello di mostrare che esistono due configurazioni di C, le chiameremo C_0 e C_1, con $C_1 = e'(C_0)$, dove $e' = (p', m') \neq e$, e tali che $D_0 = e(C_0)$ e $D_1 = e(C_1)$ sono, rispettivamente, 0-valente e 1-valente[3]. Per provare tale affermazione sfruttiamo il fatto che D contiene sia una configurazione d_0 0-valente sia una configurazione d_1 1-valente. Entrambe d_0 e d_1 sono raggiungibili da c, come mostrato nella Figura 6.17. Chiamiamo σ_0 il cammino da c a c_1 e σ_1 il cammino da c a c_1.

Partendo dal fatto che d_1 è 1-valente, e considerando il cammino da c a c_1 fatto da

[3] Qui il ruolo di 0 e 1 potrebbe essere invertito. Il punto chiave è che si passa da una configurazione univalente di un valore (0 o 1) a un configurazione univalente del valore negato.

configurazioni successive in C, ci chiediamo se il nodo di D corrispondente al nodo di C che precede c_1 è 0-valente oppure 1-valente. Se è 0-valente abbiamo trovato la coppia di nodi C_0 e C_1. Se è 1-valente ripetiamo il procedimento con il nodo procedente nel cammino da c a c_1.

Possiamo fare lo stesso, in modo simmetrico, partendo da d_0. Poichè andando a ritroso su σ_0 e σ_1 arriviamo in entrambi i casi a c, dovremo necessariamente trovare (su almeno uno fra σ_1 e a σ_2) due configurazioni C_0 e C_1 per le quali le corrispondenti configurazioni in D passano da 0-valente a 1-valente (o da 1-valente a 0-valente). Senza perdere in generalità assumiamo che si passi da 0-valente a 1-valente: nel caso simmetrico si procede nello stesso modo scambiando i ruoli di 0 e 1. La Figura 6.18 mostra le configurazioni C_0, C_1, D_0 e D_1.

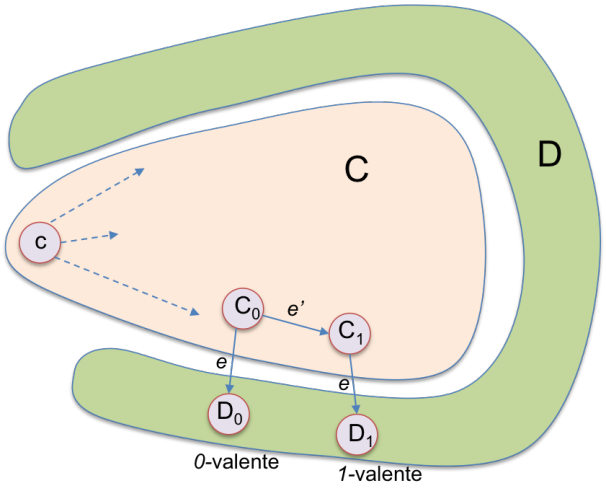

Figura 6.18: Configurazioni C_0, D_0, C_1 e D_1.

Sia p' il processore coinvolto nel passo e'.

Se fosse $p' \neq p$, potremmo applicare e' a D_0 e per il Lemma 6.5.2 si avrebbe che $e'(D_0) = D_1$, come mostrato nella Figura 6.19. Ma questo è un assurdo in quanto si passerebbe da una configurazione 0-valente a una configurazione 1-valente.

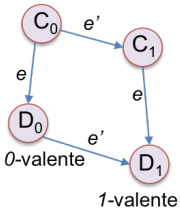

Figura 6.19: Situazione impossibile per il caso $p' \neq p$.

Pertanto deve necessariamente essere $p' = p$. Ma anche in questo caso arriviamo a un assurdo. Infatti, possiamo considerare l'esecuzione parziale σ da C_0 nella quale p non esegue più nessun passo e che arriva in una configurazione finale, cioè nella quale si prende una decisione, in altre parole una configurazione univalente, $f = \sigma(C_0)$. Questo è un punto cruciale della prova: qui invochiamo il fatto che l'algoritmo deve funzionare anche se un solo processore si guasta. Quindi l'esistenza dell'esecuzione σ è garantita dal fatto che da C_0 dobbiamo arrivare a una decisione anche se il processore p si guasta e quindi non esegue nessun passo.

Per il Lemma 6.5.2 possiamo applicare σ anche a D_0 e D_1 e parimenti l'evento e ad f e anche la sequenza di eventi (e', e) sempre a f, raggiungendo le configurazioni E_0 ed E_1, come mostrato nella Figura 6.20. Tale situazione è impossibile in quanto si avrebbe una configurazione univalente, la configurazione f, che può essere trasformata sia in 0-valente (configurazione E_0) che in 1-valente (configurazione E_1).

Pertanto D deve necessariamente contenere una configurazione bivalente. □

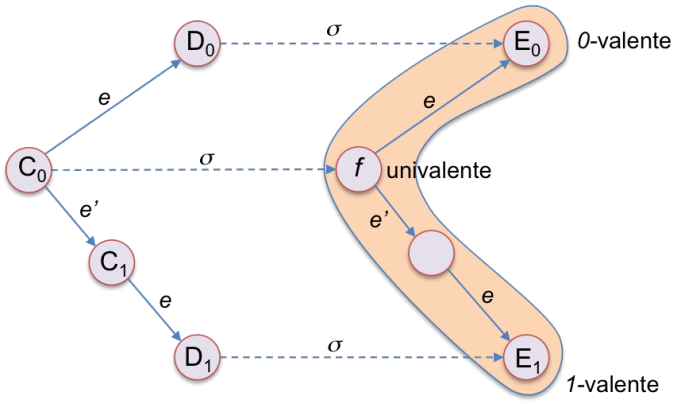

Figura 6.20: Situazione impossibile per il caso $p' = p$.

6.5.2 *Algoritmo randomizzato*

Poichè il problema del consenso non è risolvibile in sistemi asincroni con algoritmi deterministici, spostiamo l'attenzione su una soluzione randomizzata. Per poter utilizzare la randomizzazione rilassiamo leggermente il problema cambiando la condizione di terminazione nel seguente modo:

- (Terminazione) Tutti i processi non guasti decidono entro il tempo t con probabilità almeno $1 - \epsilon(t)$,

dove $\epsilon(t)$ è una funzione che diventa sempre più piccola al crescere di t.

Una soluzione semplice è l'algoritmo BENOR. Tale algoritmo richiede che $n > 2f$ e $V = \{0, 1\}$.

Algoritmo BENOR. Ogni processore mantiene due variabili locali x e y. Inizialmente la x contiene il valore di input del processore e la y è `null`. Ogni processore p_i esegue una serie di *fasi* numerate $1, 2, \ldots$. Il processore continua ad eseguire l'algoritmo anche dopo aver preso la decisione finale. In ogni fase s, il processore p_i esegue 2 rounds:

Round 1: Spedisce in broadcast il messaggio (first, s, x). Aspetta l'arrivo di almeno $n - f$ messaggi della forma (first, s, v). Se almeno $\lfloor \frac{n}{2} \rfloor + 1$ hanno tutti lo stesso valore v allora $y := v$, altrimenti $y :=$ `null`.

Round 2: Spedisce in broadcast il messaggio (second, s, y). Aspetta l'arrivo di almeno $n - f$ messaggi della forma (second, s, v).

(a) Se c'è un messaggio con un valore $v \neq$ `null` allora $x := v$;

(b) Se almeno $f + 1$ messaggi contengono uno stesso valore v allora il processore decide v

(c) Altrimenti, si sceglie uniformemente a caso 0 o 1 come valore di x.

Osserviamo che i passi del round 2 sono ben definiti in quanto si può provare che i valori y sono o tutti appartenenti a $\{0, \texttt{null}\}$ oppure tutti appartenenti a $\{1, \texttt{null}\}$. Quindi se esistono più messagi *second* per valori diversi da `null` essi contengono tutti lo

stesso valore. Osserviamo anche che la condizione $n > 2f$ è necessaria per assicurarsi di avere un valore di y diverso da `null` in una fase in cui tutti i processori iniziano con lo stesso valore di x.

Lemma 6.5.5 *Per qualsiasi fase, si ha che $y_i \in \{0, \texttt{null}\}$ per tutti i processori i, oppure che $y_i \in \{1, \texttt{null}\}$.*

Dimostrazione. Affinchè un processore i esegua $y := v$ è necessario che il processore riceva un messaggio *first* per v da più di $n/2$ processori. Poichè i processori sono n non possono esserci due valori per i quali ciò succede. □

Lemma 6.5.6 *In una fase in cui tutti i processori i non guasti iniziano con lo stesso valore $x_i = v$, tutti i processori non guasti prenderanno una decisione.*

Dimostrazione. Poichè $n > 2f$ ci sono almeno $n/2 + 1$ processori non guasti che spediranno il valore $x_i = v$ nei messaggi *first*. Quindi ogni processore non guasto riceverà almeno $n/2 + 1$ messaggi e quindi avrà $y_i = v$. Nel secondo round tutti i processori non guasti spediranno tale valore nei messaggi *second* e poichè $n/2 + 1 \geq f + 1$ tutti i processori non guasti riceveranno almeno $f + 1$ messaggi e quindi decideranno v. □

Lemma 6.5.7 *L'algoritmo* BenOr *soddisfa la proprietà di validità.*

Dimostrazione. La prorietà di validita richiede che se tutti i valori iniziali sono v allora v deve essere la decisione. Quindi assumiamo che tutti i valori di input siano uguali a v. Quindi tutti i processori avranno $x = v$; pertanto i messaggi *first* avranno tutti il valore v come terza componente. Di conseguenza l'unico valore possibile per i messagi *second* è v. Dunque l'unico valore possibile per un processo che decide è v. □

Lemma 6.5.8 *L'algoritmo* BenOr *soddisfa la proprietà di accordo.*

Dimostrazione. La proprietà di accordo richiede che non ci siano due decisioni diverse. Sia p_i il primo processore che decide e sia s la fase nella quale il processore decide e v il valore. Poichè p_i è il primo processore che decide non ci sono altri processori che decidono in una fase s', $s' < s$. Poichè p_i decide nella fase s, dal codice dell'algoritmo si ha che p_i riceve nella fase s almeno $f + 1$ messaggi della forma $(second, s, v)$. Questo implica che un qualsiasi processore p_j, $j \neq i$, che completa la fase s riceve almeno 1 di questi messaggi, visto che p_j riceverà un messaggio da tutti i processori che hanno spedito un messaggio a p_i con al massimo l'eccezione dei processori che si guastano che sono al più f. Dal Lemma 6.5.5 sappiamo che p_i riceverà messaggi *second* solo per questo valore e quindi non può decidere su un altro valore.

Inoltre p_j, avendo ricevuto almeno un messaggio *second* per v, imposterà il valore della variabile x a v. Questo è vero per tutti i processori che completano la fase s. Pertanto tutti i propcessori che iniziano la fase $s + 1$ (e anche quelle successive) avranno $x = v$ e possiamo concludere, con una giustificazione simile a quella del lemma precedente, che l'unico valore sul quale i processori possono decidere è v. □

Lemma 6.5.9 *L'algoritmo* BenOr *soddisfa la proprietà di terminazione: tutti i processori non guasti decidono nelle prime $s+1$ fasi con probabilità almeno $1-\left(1-\frac{1}{2^n}\right)^s$.*

Dimostrazione. Per $s=0$ la prova è banale vista che la probabilità con cui i processori devono decidere diventa 0. Consideriamo quindi il caso $s \geq 1$.

Dal Lemma 6.5.6 si ha che che se tutti i processori i iniziano la fase s con lo stesso valore di $x_i = v$, allora la fase avrà successo e i processori prenderanno una decisione. Quindi ragioniamo su come vengono scelti i valori per la fase s. Tali valori vengono stabiliti nel round 2 della fase $s-1$. Dal codice dell'algoritmo si ha che il valore di x può essere stabilito in due modi: o copiando il valore di un messaggio di tipo *second*, oppure scegliendo un valore casuale.

Per il Lemma 6.5.5 si ha che i valori spediti nei messaggio di tipo *second* non sono mai discordanti, pertanto nel round 2 della fase $s-1$ alcuni valori x_i saranno stabiliti in base ai messaggi di tipo *second* e saranno o tutti 0 o tutti 1 e i restanti x_i saranno stabiliti a caso. Si noti che sono possibili anche i casi estremi in cui tutti i valori x_i sono stabiliti a caso oppure sono tutti stabiliti in base al valore dei messaggi di tipo *second*. In ogni caso c'è la possibilità che siano tutti uguali.

Quale è la probabilità che siano tutti uguali? La probabilità di tale evento dipende da quanti valori vengono scelti a caso: più sono i valori scelti a caso e minore sarà tale probabilità. Quindi il caso "peggiore" è quello in cui tutti i valori sono scelti a caso e in tale caso la probabilità che tutti i valori siano uguali è $2/2^n = 1/2^{n-1}$.

Dunque la probabilità di terminare l'algoritmo nella fase s è almeno $1/2^{n-1}$, pertanto la probabilità che *non* venga raggiunta la decisione nella fase s è al massimo $\left(1-\frac{1}{2^n}\right)$. Tale affermazione è vera per ogni fase e la probabilità è indipendente dalle fasi precedenti in quanto dipende solo dalle scelte casuali fatte per stabilire i valori x_i.

Quindi la probabilità che non venga raggiunta nessuna decisione in s fasi è al massimo $\left(1-\frac{1}{2^{n-1}}\right)^s$. Per cui con probabilità almeno $1-\left(1-\frac{1}{2^{n-1}}\right)^s$ tutti i processori non guasti decidono nelle prime $s+1$ fasi. □

Si noti che il teorema fornisce una stima del "tempo" necessario a far diventare ϵ piccolo in termini di numero di fasi. Poichè siamo in un sistema asicnrono per avere una valutazione che sia una funzione del tempo reale è necessario fare delle assunzioni su quanto può durare una fase.

L'algoritmo di Ben-Or può essere adattato per far fronte a guasti di tipo bizantino ma per funzionare richiede che la percentuale di processori che possono guastarsi sia minore, più precisamente deve essere $n > 5f$.

6.6 Il problema del consenso in sistemi reali

Abbiamo visto che il problema del consenso è estremamente semplice da risolvere quando tutto funziona bene mentre può essere irrisolvibile in presenza di guasti. Lo abbiamo affrontato considerando varie assunzioni sia sulla sincronia del sistema sia sul tipo di guasti. Per quanto riguarda la sincronia siamo passati da un estremo, sistemi totalmente sincroni, all'altro, sistemi totalmente asincroni. La sincronia gioca un ruolo fondamentale per la progettazione di algoritmi distribuiti. I sistemi totalmente sincroni

sono molto più facili da gestire; tuttavia in pratica è molto difficile avere tali sistemi. È molto più comodo avere algoritmi che funzionano per sistemi totalmente asincroni in quanto ciò facilita l'implementazione fisica del sistema distribuito; purtroppo progettare algorimi che funzionino anche in assenza di sincronia è più difficile e a volte impossibile come abbiamo visto per il problema del consenso. Nella realtà i sistemi distribuiti sono "parzialmente" sincroni: i processori eseguono ogni passo in al massimo ℓ unità di tempo e i messaggi vengono consegnati in al massimo d unità di tempo. Questo tipo di sincronia è più facile da implementare rispetto alla sincronia totale. In questi sistemi diventa possibile risolvere il problema del consenso. Infatti l'assunzione di parziale sincronia permette di adattare algoritmi progettati per sistemi sincroni ai sistemi parzialmente sincroni (si veda il Capitolo 25 di [26] per approfondimenti).

Nel seguito presenteremo un algortimo, chiamato Paxos, per il consenso in sistemi parzialmente sincroni. In realtà l'algoritmo è molto più robusto in quanto la parziale sincronia è sfruttata molto poco. In pratica l'algoritmo garantisce le proprietà di safety anche se il sistema è completamente asincrono, anche se i messaggi si perdono e anche se i processori si guastano con guasti di tipo stop. Le proprietà di liveness invece sono garantite solo se per un periodo sufficientemente lungo il sistema non presenta guasti e la parziale sincronia è rispettata. Pertanto Paxos è un algoritmo molto importante dal punto di vista pratico.

Le assunzioni riguardo al sistema sono: sistema asincrono con scambi di messaggi. Ogni nodo opera a una propria velocità, può fermarsi per un guasto stop ma può anche essere riparato. I messaggi possono richiedere un tempo arbitrariamente lungo per essere consegnati, possono essere persi, duplicati ma non alterati.

6.6.1 *Overview*

L'idea di base dell'algoritmo Paxos è quella di proporre dei valori fino a che uno di essi viene accettato da una maggioranza dei processi. Il valore accettato da una maggioranza dei processi è il valore della decisione finale. Un qualsiasi nodo/processo può proporre un valore. I nodi che propongono valori verranno detti *leader*. Per proporre un valore un proponente inizia un round logico. I round sono numerati con una numerazione che permette un ordine totale (un esempio è dato da una coppia formata da un numero progressivo e l'ID del nodo). Informalmente i passi di un round sono i seguenti.

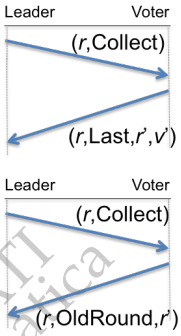

Figura 6.21: Possibili risposte al messaggio Collect

1. Il proponente (leader) spedisce un messaggio di "Collect" a tutti gli altri nodi. Il messaggio Collect serve a dichiarare l'intenzione di iniziare un round, di cui deve specificare il numero, e allo stesso tempo chiede informazioni riguardo ai round precedenti in cui i nodi sono stati coinvolti.

2. I nodi (voter) che ricevono un messaggio Collect, rispondono con un messaggio "Last", fornendo informazioni riguardo ai precedenti round, in particolare l'ultimo (cioè con il numero più alto) round in cui il nodo è stato coinvolto. Con tale risposta il nodo fa anche la promessa di non accettare nessun valore per un round con un numero minore. Se il nodo ha promesso a qualche altro proponente di non accettare il round proposto, allora la risposta sarà un messaggio di "OldRound". La Figura 6.21

mostra le possibili risposte al messaggio Collect.

3. Quando il proponente ha ricevuto messaggi "Last" da almeno una maggioranza dei processori, decide il valore da proporre nel round e invia un messaggio di "Begin" per il round specificando il valore proposto. Il valore da proporre dipende dalle informazioni ricevute: se qualche processo già conosce una decisione allora il valore proposto sarà la decisione già presa, altrimenti sarà il valore proposto nel round più recente fra quelli conosciuti o quello iniziale del proponente.

4. I nodi che ricevono il messaggio "Begin" e non hanno nessun vincolo di rifiuto per il numero di round a cui si riferisce il messaggio, accetteranno il valore proposto spedendo un messaggio di "Accept". Nel caso siano vincolati a un round con un numero più alto spediranno un messaggio di "OldRound". La Figura 6.22 mostra le possibili risposte al messaggio Begin.

5. Se il proponente riceve almeno una maggioranza di "Accept", allora può decidere sul valore proposto nel round.

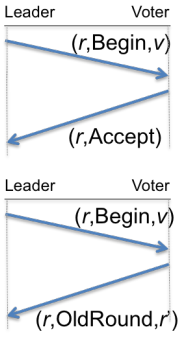

Figura 6.22: Possibili risposte al messaggio Begin

A questo punto la decisione è stata presa, ma è conosciuta solo dal proponente del round. Occorre informare gli altri nodi. Il proponente potrà farlo spedendo ulteriori messaggi "Success", come mostrato nella Figura 6.23. In realtà la cosa importante è che non è più possibile scegliere un nuovo valore. Infatti la regola per la scelta del valore da proporre e il fatto che due maggioranze hanno sempre un nodo in comune garantisce che se un valore è stato accettato in un round con un numero più piccolo allora tale valore è l'unico possibile per round con numeri più grandi.

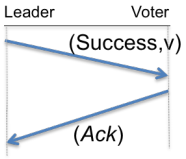

Figura 6.23: Messaggio di successo

Si noti che l'algoritmo prevede la possibilità che più leader decidano contemporaneamente di iniziare un nuovo round. I messaggi pertanto saranno sempre etichettati con il numero del round in modo tale che non ci sia confusione. I numeri dei round sono unici e sono formati da un valore intero che viene incrementato dal leader del round e dall'identificatore del leader stesso. Ad esempio il numero di round $r = (87, 3)$ è il numero di round scelto dal processo 3 con progressivo 87. Si noti che con tale numerazione per ogni coppia di numeri di round r e r', con $r \neq r'$ si ha che o $r < r'$ oppure $r' < r$, semplicemente definendo l'ordine $<$ con $(x, i) < (y, j)$ se e solo se $x < y$ oppure $x = y$ e $i < j$. I leader sceglieranno dei numeri di round sempre crescenti rispetto a quelli di cui vengono a conoscenza.

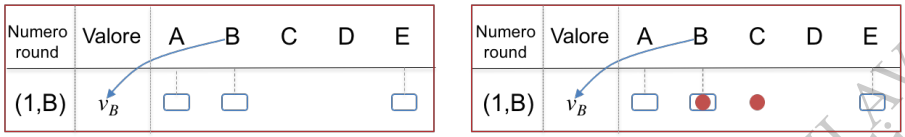

Figura 6.24: Esempio di esecuzione: Round (1,B). I riquadri vuoti indicano l'invio di un messaggio Last, i riquadri pieni l'invio di un messaggio Accept.

La Figura 6.24 mostra un esempio di esecuzione dell'algoritmo Paxos. In questo esempio il nodo B ha inziato il round $(1, B)$ e i nodi A, B e E hanno risposto con un messaggio Last, promettendo quindi di non accettare successivi round con numero inferiore. Poichè B non è a conscenza di nessun altro round precedente è libero di scegliere il proprio valore iniziale come possibile decisione da prendere nel round. I

processori B e C hanno anche accettato il valore proposto da B nel round $(1, B)$. Questo round non ha permesso il raggiungimento di una decisione in quanto non c'è stata una maggioranza dei processori che ha accettato il valore proposto.

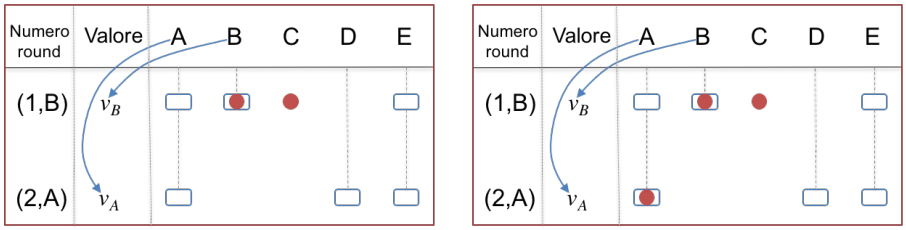

Figura 6.25: Esempio di esecuzione: Round (2,A).

Successivamente il processore A decide di iniziare un nuovo round $(2, A)$, come mostrato nella Figura 6.25. In questo round i processori A, D e E inviano un Last message impegnandosi a non partecipare in nessun round con numero inferiore. Il valore proposto in questo round viene accettato solo da A, quindi anche in questo round non viene presa nessuna decisione.

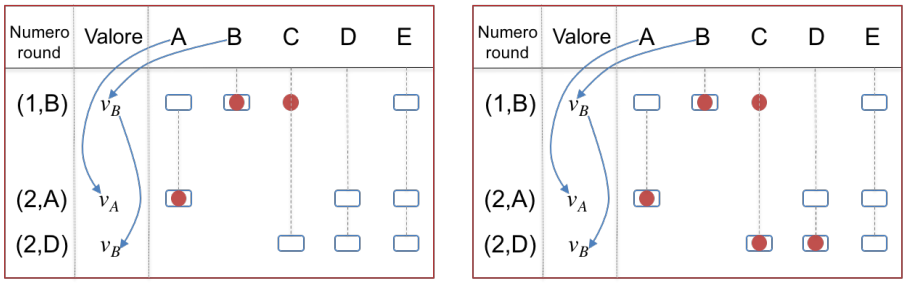

Figura 6.26: Esempio di esecuzione: Round (2,D).

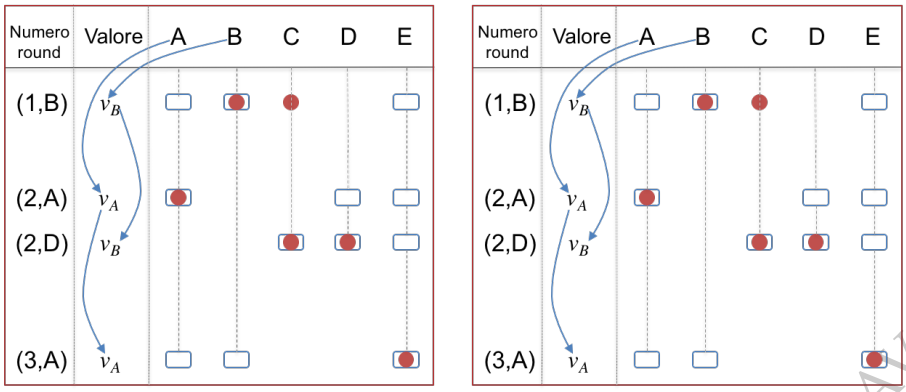

Figura 6.27: Esempio di esecuzione: Round (3,A).

Le Figure 6.26 e 6.27 mostrano altri due round che non portano a una decisione. Per questi due round è fondamentale osservare che il valore proposto è quello del round precedente con numero più alto fra quelli conosciuti. Ad esempio nel round $(2, D)$ il leader D propone il valore v_b in quanto c'è il processore C, che ha spedito un messaggio Last, che è a conoscenza del fatto che nel round $(1, B)$ è stato proposto il valore v_B. Analogamente nel round $(3, A)$ viene proposto il valore v_A in quanto fra la

maggioranza dei processori che hanno inviato un Last messagge non c'è nessuno che conosce il valore proposto nel round $(2, D)$, mentre il processore A conosce il valore proposto nel round $(2, A)$ che è il più recente fra quelli conosciuti.

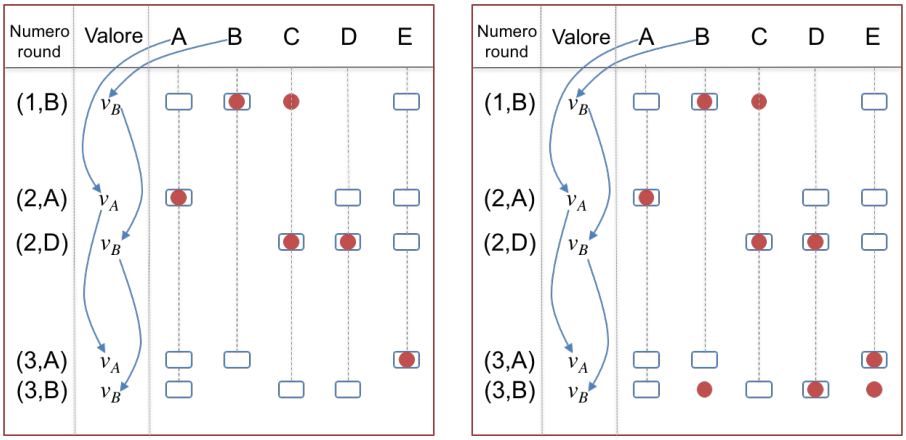

Figura 6.28: Esempio di esecuzione: Round (3,B).

La Figura 6.28 mostra il successivo round che permette di decidere sul valore v_B. Dal momento in cui il round $(3, B)$ ha ottenuto una maggioranza di Accept, successivi round con numeri inferiori non potranno mai avere una maggioranza (quindi non potranno nemmeno iniziare) mentre round con numeri superiori potranno proporre solo il valore v_B. Il fatto che il valore di un round che permette al leader di decidere deve essere l'unico valore possibile per i round successivi è cruciale ovviamente per la proprietà di accordo. Questa è la parte delicata di tutto l'algoritmo e l'intersezione garantita dall'uso delle maggioranze permette di mantenere la consistenza delle scelte dei valori da proporre nei round.

Algorithm 24: PAXOSLEADER$_i$

```
counter = 1
while decido di iniziare un nuovo round do
    counter++
    r = (counter, i)
    Spedisci (r, Collect)
    Ricevi messaggi (r, Last, r', v')
    Aggiorna counter al valore più grande
    Wait: una maggioranza di (r, Last, r', v')
          (Se troppa attesa, inizia un nuovo round)
    Scegli v come il v' nel messaggio Last con r' più grande
    Se non c'è un tale v' prendi il valore iniziale di i
    Spedisci (r, Begin, v)
    Ricevi messaggi (r, Accept)
    Wait: una maggioranza di (r, Accept)
          (Se troppa attesa, inizia un nuovo round)
    Decidi sul valore v di r
    Spedisci (Success, v)
    Ricevi messaggi (Ack)
```

6.6.2 *Pseudocodice*

La Figura 24 mostra una descrizione con pseudocodice (molto informale) del comportamento dei leader. Un nodo può può iniziare un nuovo round in qualsiasi momento, diventando così leader di quel round. Nella prima fase di un round il leader richiede informazioni a tutti gli altri nodi inviando il messaggio di Collect. La scelta del numero del round è importante. Il leader sceglie un numero di round che è maggiore di tutti i numeri di round di cui è a conoscenza. Pertanto, ogni volta che un leader riceve informazioni riguardanti altri round prende nota dei numeri di round in modo da tale poter scegliere un numero di round maggiore di tutti quelli già usati. I messaggi Last contengono anche i valori proposti nei round precedenti. Grazie a queste informazioni il leader potrà scegliere come valore da proporre nel proprio round il valore che corrisponde al numero di round più grande contenuto nel messaggi Last. Si noti che il leader sceglie tale valore solo dopo aver ricevuto messaggi Last da almeno una maggioranza dei nodi. Nel caso in cui nessuno dei messaggi Last contiene un round precedente (all'inizio nessun nodo ha partecipato a nessun round) il leader proporrà il proprio valore di input. A questo punto il leader può chiedere di accettare il valore porposto e a tal fine spedisce a tutti il messaggio Begin in cui viene specificato il valore del round. Se riceve una maggioranza di Accept il leader dichiara successo nel round e quindi può decidere sul valore proposto nel round.

L'algoritmo ammette la possibilità che più leader inizino contemporaneamente dei round. Questo significa che durante l'esecuzione un leader può ricevere messaggi da altri leader. In questo caso, poichè le attività di un leader possono interferire con quelle

degli altri leader, impedendo il raggiungimento di una decisione, è opportuno non proseguire con il round iniziato ma "lasciare il passo" al round con il numero più alto. Come vedremo la terminazione è garantita solo quando per un periodo sufficientemente lungo un solo processore si comporta da leader dando la possibilità a tale leader di terminare con successo un round.

Algorithm 25: PAXOSVOTER$_i$

```
lastR = (0, i)
lastV = input_i
Commit = (0, i)
while 1 do
    if Ricevo (r, Collect) then
        if r ≥ Commit then
            Spedisci (r, Last, lastR, lastV)
            Commit = r
        else
            Spedisci (r, OldRound, Commit)
    if Ricevo (r, Begin, v) then
        if r ≥ Commit then
            Spedisci (r, Accept)
            lastR = r
            lastV = v
        else
            Spedisci (r, OldRound, Commit)
```

La Figura 25 mostra una descrizione con pseudocodice del comportamento dei votanti (tutti i nodi che ricevono i messaggi spediti dai leader sono votanti). Un nodo accetterà i valori proposti a meno che non è stato già coinvolto in round precedenti. A tal fine ogni votante manterrà delle informazi riguardo a quanto fatto in precedenza, in particolare la variabile *Commit* contiene l'ultimo round per il quale il nodo ha risposto positivamente al messaggio di Collect, mentre le variabili *lastR* e *lastV* contengono rispettivamente il numero di round e il valore dell'ultimo round in cui il nodo ha accettato il valore proposto. Pertanto, quando arriva un nuovo messaggio di Collect per un round r, il votante controllerà se $r \geq Commit$, cioè se il nuovo round è sufficientemente "nuovo". Se lo è allora il votante spedirà le informazioni necessarie con un messaggio Last e contemporaneamente aggiornerà il valore di *Commit*. Se il round non è sufficientemente nuovo allora la risposta sarà un messaggio OldRound. Analogamente alla ricezione di un messaggio Begin, il votante risponderà con un Accept solo se il round proposto è sufficientemente nuovo.

6.6.3 *Accordo e validità*

L'algoritmo garantisce la proprietà di validità in quanto tutti i valori che vengono proposti sono sempre uguali a uno degli input. Provare che la proprietà di accordo è soddisfatta è un po' più complicato. Una prova formale va al di là degli obiettivi di questo corso. Forniamo delle argomentazioni informali.

Poichè durante l'esecuzione dell'algoritmo è possibile che dei leader inizino dei round che non saranno mai completati per il fatto che una maggioranza dei processori partecipa a round con numeri più alti, classificheremo tali round come *bloccati*. Quindi un round bloccato è un round r per il quale una maggioranza dei nodi ha fatto un commit per un round con un numero più grande di r e pertanto r non potrà procedere.

Inoltre diremo che un round r è *ancorato* se ogni altro round $r' < r$ è o bloccato oppure ha lo stesso valore di r.

Osserviamo che se il nodo i ha partecipato ad un round r_1 e spedisce un messaggio Collect per il round $r_2 > r_1$, allora il nodo i non parteciperà a nessun round r, $r_1 < r < r_2$.

Lemma 6.6.1 *Ogni round non bloccato è ancorato.*

Dimostrazione. **(sketch)** Procediamo per induzione sulla lunghezza dell'esecuzione. Quindi supponiamo che l'asserzione è vera in uno stato s e dobbiamo provare che è vera nello stato s' a cui si arriva da s eseguendo uno step dell'algoritmo. L'unico passo che può rendere falsa l'asserzione del lemma è la scelta di un nuovo valore per un round: se il valore non fosse uguale a quelli scelti per tutti i round precedenti non bloccati il lemma sarebbe falso. Sia r il numero del round. Poichè il valore viene scelto quando una maggioranza dei nodi ha inviato un messaggio Last si ha che quella maggioranza non parteciperà a round compresi fra r' e r dove r' è il valore più alto ricevuto nei messaggi Last. Si noti che poichè stiamo scegliendo il valore di r il round r' è esso stesso ancorato (per l'ipotesi induttiva). Pertanto anche r è ancorato. □

6.6.4 *Terminazione*

L'algoritmo Paxos non garantisce terminazione in presenza di leader multipli: ognuno di essi potrebbe iniziare un nuovo round impedendo a quelli precedenti di arrivare a completamento. Tuttavia questo è possibile solo in presenza di leader multipli. Se la situazione si stabilizza ed esiste un solo leader esso potrà portare a completamento un round. Infatti dal momento in cui il leader è unico un solo round iniziato dall'unico leader potrebbe essere bloccato da round iniziati precedentemente da nodi che erano leader (in quanto i numeri di quei round potrebbero essere più grande del round iniziato dal leader). Un secondo round iniziato dall'unico leader sarà portato a termine senza ostacoli.

Pertanto per garantire la terminazione dell'algoritmo è necessario garantire l'unicità di un leader e la possibilità per questo leader di comunicare con una maggioranza dei nodi per un periodo sufficientemente lungo.

Si noti anche che dopo la decisione presa dal leader è necessario che anche gli altri nodi vengano informati della decisione: è sufficiente che il leader invii un messaggio a

tutti gli altri nodi e si accerti, tramite un riscontro, che il messaggio sia stato ricevuto (si veda la Figura 6.23).

6.7 *Consenso e registri distribuiti*

In questa sezione ci occuperemo di registri distribuiti e delle loro applicazioni che recentemente hanno suscitato un notevole interesse. I registri distribuiti hanno alla loro base il problema del consenso. Che cosa è un registro distribuito? Un registro, genericamente parlando, è un elenco di annotazioni ordinate, cioè una prima annotazione, una seconda, e così via. Un registro, per esempio, può essere usato per tenere traccia di entrate e uscite di un'attività economica. Più in generale è una sequenza ordinata di elementi, che possono essere di qualsiasi natura. I registri possono essere fisici, come per esempio i libri mastro, ma anche ovviamente digitali, cioè memorizzati in file/database.

Figura 6.29: Libro mastro

Un registro distribuito è un registro digitale che è gestito tramite un sistema distribuito, solitamente con ogni nodo che mantiene una copia del registro. La difficoltà per l'implementazione di un registro distribuito è quella di non creare inconsistenze fra le varie copie.

Il problema del consenso è alla base dei registri distribuiti: ogni elemento del registro è una informazione sulla quale i nodi del sistema distribuito devono mettersi d'accordo. Quindi, molto genericamente, possiamo dire che per implementare un registro distribuito occorre risolvere una istanza del problema del consenso per ogni elemento del registro. Se tutti i nodi del sistema sono d'accordo sui singoli elementi del registro il registro è valido e può essere usato senza che ci siano inconsistenze fra le versioni dei singoli nodi.

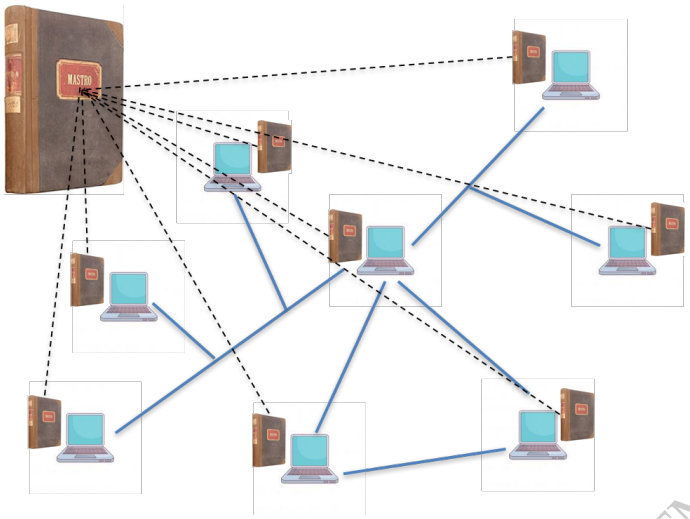

Figura 6.30: Registro distribuito

Negli ultimi anni si è visto un crescente interesse per i registri distributi. Buona parte di queto interesse è dovuto all'apparizione delle criptovalute, prime fra tutte Bitcoin. Sebbene i regsitri distribuiti possano essere utilizzati in vari contesti, consideriamo come esempio proprio il caso delle criptovalute.

Immaginiamo di voler creare una criptovaluta. Prima di tutto dovremmo chiederci che cosa é una valuta, in base a quali principi viene accettata e come la possiamo utilizzare. Senza voler approfondire troppo questo discorso, che va oltre gli obiettivi di questo capitolo, diciamo che per avere una valuta, e la cosa vale anche per una criptovaluta, è necessario che chi partecipa al sistema accetti le regole del sistema. Quindi il primo passo è quello di definire delle regole del nostro sistema per una criptovaluta.

La regola principale è che si utilizza un registro, per il momento possiamo anche pensarlo come astratto, anche se poi alla fine dovremo implementarlo come registro distribuito, se vogliamo utilizzarlo nel contesto di una rete di calcolatori. Nel registro vengono memorizzate delle *transazioni*. Le transazioni spostano somme di criptovaluta. Per essere concreti e usare degli esempi diamo un nome alla nostra criptovaluta: euro virtuali (€V). Assumiamo anche per semplicità che i partecipanti alla criptovaluta, Alice, Bruno, Carlo e Davide, inizialmente abbiano tutti 100€V.

Ora supponiamo che Alice voglia pagare 30€V a Bruno. Basterà scrivere nel registro questa transazione:

REGISTRO
Alice paga 30€V a Bruno

Poi Carlo vuole pagare 50€V a Davide, e Bruno invece vuole pagare 25€V a Davide. Il registro diventa:

REGISTRO
Alice paga 30€V a Bruno
Carlo paga 50€V a Davide
Bruno paga 25€V a Davide

Dall'analisi del registro è possibile risalire alla situazione aggiornata riguardo la somma posseduta da ognuno dei partecipanti: Alice ha 70€V, Bruno ne ha 105, Carlo ha 50€V e Davide 175. In linea di principio una criptovaluta funziona più o meno così!

Dovrebbe essere evidente però che questo approccio molto semplice presenta vari problemi. Il primo più evidente è chi scrive cosa nel registro. Ad esempio Davide potrebbe scrivere "Alice paga a Davide 100€V, poi "Bruno paga a Davide 100€V, poi "Carlo paga a Davide 100€V prendendosi tutti i soldi. Ovviamente se lo può fare Davide lo possono fare anche gli altri. E ovviamente il sistema non funzionerebbe. La soluzione a questo problema è facile e viene dalla crittografia: le transazioni vengono firmate digitalmente da chi spende e quindi possono essere create solo da chi spende. Anche firmando le transazioni però rimane un problema: se Alice paga Davide scrivendo nel registro "Alice paga a Davide 100€V", Davide può ricopiare questa transazione senza alterare nulla e quindi avere un doppio, o multiplo, pagamento da Alice.

Un altro problema evidente è capire quanti soldi ha chi spende nel momento in cui spende. Ad esempio nella situazione dell'esempio precedente Davide ha 175€V. Potrebbe scrivere nel registro "Davide paga a Bruno 200€V, spendendo così più di quel che ha.

In questo esempio semplice abbiamo assunto che i partecipanti abbiano inizialmente tutti la stessa somma di criptovaluta. Ovviamente in pratica non è così, quindi occorrono dei meccanismi per creare e/o assegnare la criptovaluta ai partecipanti.

Se si risolvono tutti questi problemi si ha un sistema che funziona. Tutte le criptovalute affrontano questi, e anche altri problemi, per avere un sistema funzionante. Ogni approccio ha le sue peculiarietà, e i suoi lati positivi e negativi.

Infine c'è il problema di dove mantenere il registro. Una soluzione banale è quella di avere un terza parte fidata che gestisce il registro. Questa soluzione semplifica la gestione del registro ma si basa su un'assunzione difficile da rendere pratica in quanto tutti si dovrebbero fidare della terza parte. Inoltre una tale soluzione non è scalabile e la terza parte è un collo di bottiglia per l'intero sistema. Ad esempio se ha un guasto, l'intero sistema si ferma. La soluzione è quella di avere un registro distribuito e per scrivere le singole voci del registro occorre usare un algoritmo di consenso. Alcune criptovalute usano un algoritmo di consenso esplicito, altre, come Bitcoin, invece raggiungono il consenso in modo implicito, cioè senza implementare un vero e proprio algoritmo di consenso, ma sfruttando altri approcci che di fatto hanno lo stesso obiettivo (fare in modo che tutti concordino su ogni elemento del registro).

6.7.1 Blockchain

Una blockchain (letteralmente "catena di blocchi") è una sequenza di elementi (blocchi) legati fra di loro da una "catena" che in pratica stabilisce l'ordinamento degli elementi. Quindi, in pratica, una blockchain è simile a una lista a puntatori in cui ogni blocco contiene un riferimento al blocco precedente creando così una catena.

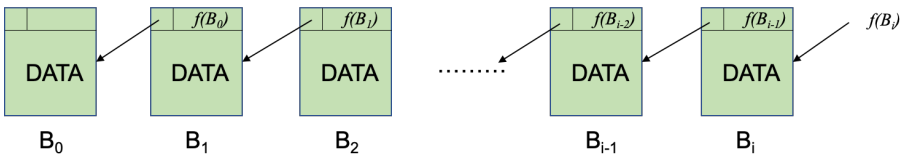

Figura 6.31: Blockchain

A differenza di una lista a puntatori, una blockchain, oltre al puntatore al blocco precedente, memorizza in ogni blocco anche un valore di hash del blocco precedente. Questo permette di verificare che i blocchi precedenti non siano stati modificati. Per le proprietà delle funzioni hash non è possibile modificare un blocco senza far cambiare il suo valore di hash e quindi una qualsiasi modifica ai blocchi precedenti comporterebbe un cambiamento di tutti i valori hash successivi fino ad arrivare all'ultimo conosciuto che permetterebbe di capire che c'è stata una modifica, a patto di poter essere sicuri di quale è/era il valore originario dell'hash contenuto nell'ultimo blocco. In pratica per garantire la sicurezza dell'intera blockchain dobbiamo preoccuparci solo di garantire la sicurezza dell'ultimo valore hash. Il primo blocco della blockchain è, in gergo, detto blocco di *genesi*, in quanto non ha un blocco che lo precede e quindi è l'origine della blockchain.

Quale è la relazione fra una blockchain e un registro distribuito? Il registro distribuito può essere implementato con una blockchain: ogni blocco potrebbe corrisponde a una

transazione e quindi il registro è dato dalla sequenza di blocchi.

In pratica, per motivi di efficienza, poichè per ogni blocco di fatto occorre raggiungere un consenso fra tutti i partecipanti, quello che si fa è raggruppare un certo numero di transazioni in un blocco e quindi anzichè mettersi d'accordo sulle transazioni una per volta ci si mette d'accordo in un solo colpo su tutte le transazioni di un blocco.

6.7.2 Bitcoin

Bitcoin implementa un registro distribuito basato su una blockchain. La blockchain assicura che gli elementi del registro, una volta scritti nel registro, non possano essere successivamente cambiati. Gli elementi stessi sono concordati attraverso un algoritmo di consenso. In bitcoin gli elementi base del registro sono le *transazioni* cioè operazioni di trasferimento della criptovaluta. I partecipanti vengono identificati da una chiave pubblica alla quale corrisponde una chiave privata che, per definizione, rimane segreta e conosciuta solo a chi ha generato la coppia di chiavi.

Una transazione in Bitcoin è una asserzione del tipo "Sposta 10 monete da Alice a Bruno". Una tale transazione deve essere firmata da Alice che, al momento della transazione, deve essere la proprietaria della somma di denaro specificata nella transazione. Ci sono delle transazioni speciali che permettono di "creare" delle nuove monete; tali transazioni sono eseguite dai cosiddetti, nel gergo Bitcoin, *minatori*. I minatori per poter creare nuove monete devono risolvere un puzzle matematico non semplice che richiede una considerevole capacità di calcolo.

Un esempio molto schematico, e semplificato, di registro Bitcoin è il seguente:

REGISTRO DISTRIBUITO
Crea 30 bitcoin e assegnali ad Alice (firmato da un minatore)
Trasferisci 20 bitcoin da Alice a Bruno (firmato da Alice)
Trasferisci 6 bitcoin da Bruno a Carlo (firmato da Bruno)
Trasferisci 3 bitcoin da Carlo a Alice (firmato da Carlo)
Trafserisci 12 bitcoin da Alice a Davide (firmato da Alice)

La transazione iniziale genera 30 bitcoin e li assegna ad Alice. Alice spende 20 dei 30 bitcoin comprando qualcosa da Bruno. Dopo questa transazione ad Alice restano 10 bitcoin mentre Bruno ne ha 20. Quindi Bruno dà 6 bitcoin a Carlo. A questo punto Alice ha ancora 10 bitcoin, mentre Bruno 14 e Carlo 6. Quindi Carlo dà 3 bitcoin ad Alice ed ora Alice possiede 13 bitcoin, Bruno 14 e Carlo 3. Infine Alice trasferisce 12 dei sui 13 bitcoin a Davide. La situazione finale vede Alice con un solo bitcoin, Bruno con 14, Carlo con 3 e Davide con 12 bitcoin.

In realtà memorizzare le transazioni in questo modo rende complicato verificare lo stato del sistema: per sapere se nell'ultima transazione Alice ha ancora abbastanza bitcoin si deve leggere a ritroso tutto il registro fino alla prima transazione. Per rendere più efficiente il controllo, Bitcoin in realtà non memorizza solo dove vanno i bitcoin spesi ma anche da dove arrivano. Quindi le transazioni avranno una parte che specifica l'input (da dove arrivano i bitcoin) e una parte che specifica l'output (dove vanno i bitcoin). Per semplificare la gestione viene imposta la condizione che il totale dell'input

deve essere uguale al totale dell'output. Questa condizione non è una limitazione in quanto in output si può far "ritornare" una parte dei bitcoin a chi li sta spendendo. Quindi una rappresentazione più veritiera, ma sempre semplificata, di come Bitcoin gestisce le transazioni nell'esempio precedente è la seguente:

	REGISTRO DISTRIBUITO
1	Input: `null` (creazione bitcoin) Output: [0] 30 ad Alice (firmato da un minatore)
2	Input: 1[0] (Transazione 1, Output [0]) Output: [0] 20 a Bruno, [1] 10 ad Alice (firmato da Alice)
3	Input: 2[0] Output: [0] 6 a Carlo, [1] 14 a Bruno (firmato da Bruno)
4	Input: 3[0] Output: [0] 3 a Alice, [1] 3 a Carlo (firmato da Carlo)
5	Input: 1[1] e 4[0] Output: [0] 12 a Davide, [1] 1 ad Alice (firmato da Alice)

Si noti come nella transazione numero 2 dei 30 bitcoin di Alice, 20 vadano a Bruno mentre 10 vanno di nuovo ad Alice. In questo modo tutti i 30 bitcoin sono stati "spesi", ma in pratica solo 20 sono passati da Alice a Bruno.

Con questo approccio è facile verificare se una nuova transazione è valida. Ad esempio per verificare la transazione 5, dobbiamo controllare le transazioni che sono richiamate nell'input: l'output 1 della transazione 1, che dà 10 bitcoin ad Alice, e l'output 0 della transazione 4 che dà altri 3 bitcoin ad Alice. Quindi Alice ha a disposizione 13 bitcoin per questa transazione e nell'output vengono spesi i 13 bitcoin. E per essere sicuri che Alice non abbia speso 2 volte gli stessi bitcoin è sufficiente controllare solo la parte di registro che inizia dalla prima transazione dell'input fino a quella che li spende. In questo particolare esempio, molto semplice, sarebbe comunque praticamente tutto il registro ma è evidente che al crescere del registro la situazione cambia e diventa un notevole vantaggio.

Un altro vantaggio è che con questo sistema è facile combinare insieme sia input che output. Ad esempio se Bruno vuole dare, dei 14 bitcoin rimasti, 2 bitcoin ad Alice, 3 a Carlo e 5 a Davide può creare la seguente transazione:

6	Input: 2[1] Output: [0] 2 ad Alice, [1] 3 a Carlo, [2] 5 a Davide, [3] 4 a Bruno (firmato da Bruno)

Se Davide vuole "consolidare" i suoi bitcoin, per poter, in futuro, far riferimento a una sola transazione per provare che possiede 17 bitcoin, potrà creare la seguente transazione:

7	Input: 5[0] e 6[2] Output: [0] 17 a Davide (firmato da Davide)

In modo simile è possibile fare un "pagamento congiunto" facendo riferimento a input che derivano da più persone. L'unica differenza in questo caso è che la transazione dovrà essere firmata da tutti i paganti.

Transazioni e blocchi. Per rendere effettiva una transazione occorre scriverla nel registro distribuito e per fare questo è necessario eseguire il protocollo di consenso fra tutti i nodi della rete Bitcoin. Questa operazione è onerosa e pertanto Bitcoin raggruppa le transazioni in "blocchi". Questi blocchi sono effettivamente i blocchi della blockchain di Bitcoin. Quindi un blocco della blockchain di Bitcoin non contiene una singola transazione ma un insieme di transazioni. Il numero di transazioni in un blocco non è fissato a priori ed è quindi variabile. Da misurazione empiriche, si ha che la grandezza media di un blocco è di circa 1MB.

Ma chi propone nuovi blocchi? Chiunque partecipi alla rete Bitcoin può proporre nuovi blocchi ma per poterlo fare deve risolvere un puzzle matematico. Chi vuole scrivere una transazione invia un messaggio in broadcast sulla rete Bitcoin. Chi è interessato a proporre nuovi blocchi memorizza le richieste di transazioni, che arrivano dagli utenti che vogliono fare le transazioni, e le organizza in un blocco.

Il puzzle matematico che bisogna risolvere per aggiungere un blocco è il seguente. Si deve trovare un numero da aggiungere al blocco in modo tale che il valore hash, calcolato con SHA256, dell'intero blocco (considerando anche il numero aggiunto), sia una stringa che inizia con 32 zeri. Chi riesce a fare questa operazione viene chiamato, nel gergo Bitcoin, *minatore*. Dunque un (potenziale) minatore deve prima "ascoltare" messaggi di richiesta di transazioni e organizzarle in un blocco, che è quindi un insieme di transazioni. Non c'è un numero prefissato di transazioni da inserire in un blocco. Su un blocco si calcola poi la funzione hash SHA256. Per poter inserire il blocco nella blockchain Bitcoin è necessario che tale valore sia più piccolo di una determinata soglia; per semplicità possiamo pensare che per essere valido, il valore has del blocco deve iniziare con una sequenza di zeri di una certa lunghezza (attualmente 32, ma questo numero è variabile). Senza un notevole sforzo aggiuntivo con altissima probabilità questo non avviene. La Figura 6.32 mostra un blocco di transazione e un esempio di valore hash SHA256; tale blocco non è valido in quanti i primi 32 bit del valore hash contengono anche degli 1.

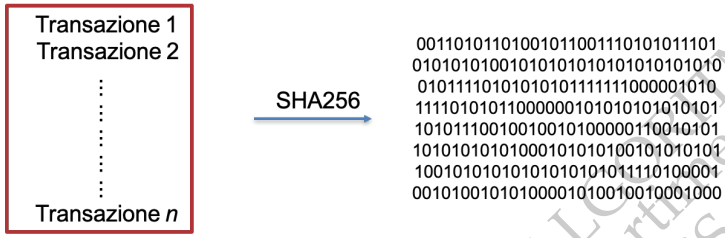

Figura 6.32: Esempio di blocco non valido

Per poter aggiungere il blocco è necessario aggiungere dei bit per fare in modo che il valore hash dell'intero blocco, quindi incluso i bit aggiuntivi, inizi con 32 zeri. La Figura 6.33 mostra un blocco valido: alle transazioni del blocco sono stati aggiunti dei bit, nella figura rappresentati da un numero con cifre decimali, che fanno sì che il valore hash del blocco inizi con 32 zeri.

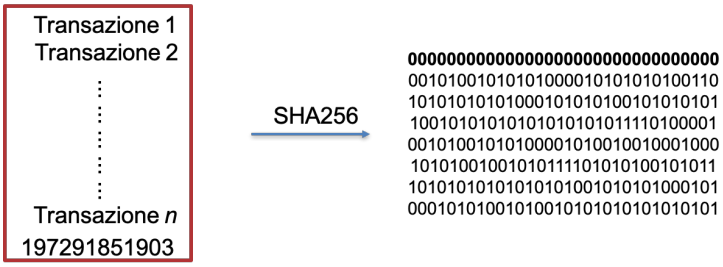

Figura 6.33: Esempio di blocco valido

Dunque un potenziale minatore deve trovare un *numero magico* (cioè i bit da aggiungere) per avere un valore di hash con la proprietà richiesta da Bitcoin. Questa operazione è computazionalmente onerosa in quanto, per le proprietà delle funzioni hash, l'unico modo per trovare il numero magico è quello di aggiungere bit provando tutte le possibilità. Cioè non c'è un algoritmo efficiente per risolvere questo puzzle matematico, ma l'unico approccio possibile – almeno per la attuali conoscenze – è quello della forza bruta.

Proof-of-work. Questo approccio viene chiamato *proof-of-work*. Infatti i minatori devono dimostrare di aver svolto del lavoro (l'individuazione del numero magico) per poter aggiungere nuovi blocchi.

Creazione di nuovi bitcoin e incentivi. Risolvere i puzzle matematici relativi al valore hash comporta un dispendio computazionale non indifferente. Questa è una scelta progettuale fatta sia per rendere non semplice questa operazione sia per incentivare la partecipazione di tutti alla rete Bitcoin: i minatori che riescono ad inserire un nuovo blocco vengono ricompensati con dei bitcoin. Cioè, quando si trova un blocco valido, vengono automaticamente "creati" nuovi bitcoin che vengono assegnati al minatore che ha trovato il blocco valido. L'ammontare esatto varia nel tempo. Inizialmente era di 50 bitcoin. Il protocollo prevede che tale valore venga dimezzato ogni 210.000 blocchi creati, approssimativamente ogni 4 anni considerando la velocità media con cui vengono creati blocchi. Attualmente (nel 2020) il suo valore è di 6.25 bitcoin.

Per creare i nuovi bitcoin, il blocco contiene una transazione speciale chiamata *coinbase* che non ha bisogno di nessun input, visto che i bitcoin vengono creati dalla transazione, e ha come output il minatore (in realtà, visto che è il minatore stesso a creare il blocco potrebbe usare qualsiasi altro indirizzo come output ma è presumibile che ogni minatore usi il proprio indirizzo per riscattare i bitcoin creati dalla transazione).

Poichè il valore dei bitcoin creati nelle transazioni coinbase viene dimezzato periodicamente è ovvio che a un certo punto sarà praticamente nullo. Il protocollo stabilisce che dal blocco 13.230.000 (che dovrebbe essere creato nel 2140 circa, stando alle attuali

stime) non verranno più creati nuovi bitcoin. Questo significa che c'è un limite superiore al numero di bitcoin che si possono creare, che è di circa 21 milioni di bitcoin.

Il ruolo dei minatori è ovviamente fondamentale: senza i minatori non si creano nuovi blocchi! Oltre ai coinbase Bitcoin prevede una ulteriore forma di incentivazione che è una tassa sulle transazioni (*transaction fee*). Tale tassa non è obbligatoria: ogni transazione può includere una transaction fee che viene riscattata dal minatore del blocco in cui la transazione viene inserita. È ovvio che i minatori daranno preferenza alle transazioni che includono una transaction fee. Pertanto pagare una transaction fee può abbreviare il tempo di attesa per far apparire la transazione nella blockchain. Dal 2140 le transaction fee saranno l'unica forma di incentivo, visto che non verranno più inserite transazioni coinbase.

Forking. Dunque, un nuovo blocco di transazioni viene aggiunto alla blockchain quando un minatore riesce a trovare un numero magico per rendere il blocco valido. A questo punto il blocco può essere messo nella blockchain. La Figura 6.34 mostra una blockchain in attesa del blocco numero 3875.

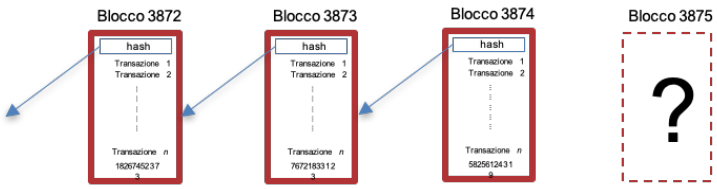

Figura 6.34: Blockchain in attesa del prossimo blocco

Il primo minatore che riesce a trovare il numero magico potrà mandare un messaggio a tutta la rete annunciando il nuovo blocco. Tutti i nodi della rete potranno verificare che il blocco è valido calcolando il suo valore hash e quindi aggiungerlo alla blockchain. Ma che succede se due (o più) minatori trovano (più o meno) contemporaneamente un numero magico? A quel punto ci sarebbero 2 candidati per essere il prossimo blocco (numero 3875 nell'esempio) e entrambi avrebbero tutto il diritto di esserlo. Questa situazione è indicata come una "biforcazione" (fork). Cioè di fatto la catena può evolvere in due, o anche più di due, direzioni diverse. E chiaramente ognuna delle biforcazioni può procedere indipendentemente dalle altre se nuovi blocchi vengono aggiunti, come mostrato nella Figura 6.35.

Ovviamente la catena deve essere unica e quindi queste biforcazioni non devono essere possibili. Per risolvere il problema Bitcoin considera come unica catena valida la biforcazione più lunga. Poiché la strategia è quella di seguire la biforcazione più lunga le altre, verranno pian piano abbandonate. Quindi è possibile che per un breve periodo ci siano più biforcazioni in competizione per essere quella valida ma solo una "sopravviverà" mentre le altre saranno considerate invalide e verranno, prima o poi, abbandondate. Con il passare del tempo solo una delle due (o più) catene sarà ritenuta valida perchè sarà quella dove più blocchi vengono aggiunti.

Questo significa che quando un nuovo blocco viene reso pubblico non possiamo subito essere sicuri che sia definitivo, cioè che sia sulla catena più lunga. Per essere

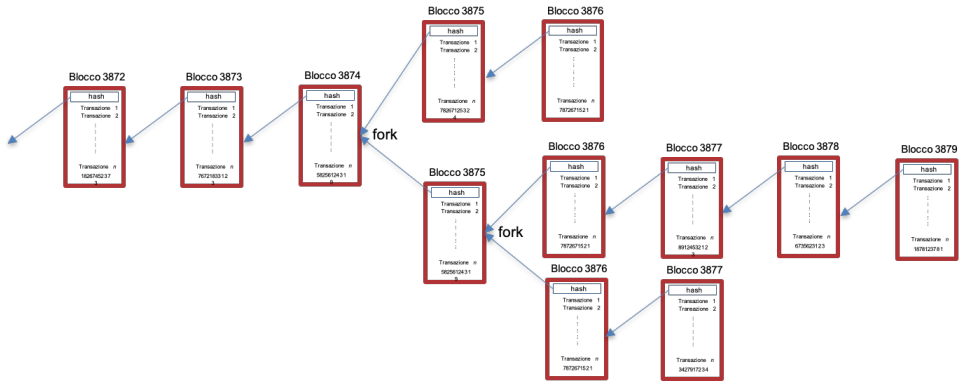

Figura 6.35: Esempi di fork della blockchain

sicuri che un blocco sia contenuto nella catena più lunga, cioè nella catena valida, è necessario aspettare che appunto ci sia una catena chiaramente più lunga delle altre. Considerando la velocità con cui nuovi blocchi vengono aggiunti, è necessario aspettare circa 10 minuti dall'approvazione del blocco e poi controllare che il blocco sia nella catena più lunga, cioè che la "diramazione" del blocco non sia stata abbandonata. Per come funziona Bitcoin l'attesa di 10 minuti è sufficiente ad avere una altissima probabilità che il blocco sia sulla diramazione più lunga e quindi sia valido.

Il "protocollo" Bitcoin. A questo punto possiamo riassumere il funzionamento generale di Bitcoin, riprendendo la descrizione usata nel white paper di Bitcoin. La rete Bitcoin funziona così:

1. Nuove transazioni vengono inviate in broadcast a tutti i nodi.
2. Ogni nodo raccoglie le richieste di transazione organizzandole in un blocco di transazioni.
3. Ogni nodo cerca un numero magico per il blocco creato.
4. Trovato un numero magico, un nodo invia il blocco in broadcast a tutti.
5. I nodi che riceveono un nuovo blocco lo accettano solo se c'è il numero magico, tutte le transazioni sono valide e non ci sono soldi spesi due volte.
6. I nodi esprimono implicitamente l'accettazione di un nuovo blocco lavorando alla creazione del blocco successivo.

In caso di biforcazioni, possibili quando due nodi inviano un nuovo blocco contemporaneamente, i nodi mantengono temporaneamente entrambe le "versioni" per, successivamente optare per la biforcazione più lunga abbondonando quella più corta.

Struttura reale delle transazioni. Come abbiamo già detto, in Bitcoin i partecipanti sono identificati da chiavi pubbliche (quindi Alice, Bruno, Carlo, Davide in realtà saranno delle chiavi pubbliche); inoltre tali chiavi pubbliche non vengono memorizzate nella loro interezza, bensì viene memorizzato solo il valore hash della chiave.

Figura 6.36: Esempio di transazione bitcoin

Un esempio reale di transazione Bitcoin è mostrato nella Figura 6.36.

Nelle transazioni sono memorizzate anche altre informazioni. In particolare ci sono informazioni relative a dei programmi che possono essere descritti con uno specifico linguaggio di scripting. Vediamo qualche dettaglio in più.

Nell'esempio la transazione contiene i seguenti campi.

`hash`: L'hash dell'intera transazione che funge anche da identificativo.

`ver`: Un numero di versione (`ver`) che specifica la versione che la transazione sta usando; la versione stabilisce il formato dell'intera transazione e il significato dei singoli campi.

`vin_sz`: Numero di "input" nella sezione `in`, cioè il numero di riferimenti alle transazione dalle quali provengono i bitcoin coinvolti in questa transazione.

`vout_sz`: Numero di "output" nella sezione `out`, cioè il numero di destinatari dei bitcoin coinvolti in questa transazione.

`lock_time`: Questo campo ha una funzione particolare che permette di controllare *quando* questa particolare transazione può essere resa attiva. E' un intero che viene interpreatato o come "altezza" di un blocco (se $<$ 500.000.000) o come una data Unix (se $>=$ 500.000.000). L'altezza di un blocco, nel gergo bitcoin, è il numero ordinale del blocco. Quindi il valore 0 indica che la transazione può essere inserita immediatamente in quanto il blocco 0 è il blocco di genesi e quindi è già stato creato. Un numero più grande implica che bisogna aspettare che quel blocco venga creato prima di poter inserire la transazione. Se è una data Unix, quindi specificata usando il numero di secondi dal 1 gennaio 1970, la transazione può essere inserita solo dopo il tempo specificato.

`size`: Grandezza in byte?

`in`: Dettagli degli input.

`out`: Dettagli degli output.

Vediamo qualche dettaglio per la sezione di input. Gli input formano un "array" di elementi, ognuno dei quali fa riferimento a un output di una precedente transazione. In questo esempio ci sono 2 input. Per ogni elemento dell'array degli input troviamo:

`prev_out`: l'hash della transazione che funziona come un puntatore alla transazione precedente.

`n`: Un indice che identifica il particolare output della transazione (è la posizine nell'array).

`scriptSig`: Una firma che permette di stabilire che si ha il diritto di usare i bitcoin a cui si sta facendo riferimento.

La sezione degli output contiene informazioni sugli output della transazione. Anche per gli output possiamo avere un array, ma in questo esempio troviamo un solo output. I campi sono:

`value`: Il valore in bitcoin di questo output alla transazione precedente.

`scriptPubKey`: Il destinatario (una chiave pubblica) dei bitcoin. In realtà questo campo è uno script che nella sua forma più semplice specifica la chiave pubblica destinataria dei bitcoin, ma può essere usato anche per fare operazioni più complesse. In ogni caso determina un destintario.

Transaction fee. Nella spiegazione informale data all'inizio abbiamo detto che la somma dei bitcoin in input deve essere uguale alla somma dei bitcoin in output. In realtà non è proprio così. Ovviamente la somma degli input non può essere inferiore alla somma degli output perché la cosa equivarrebbe a spendere soldi che non esistono. Il contrario è invece ammesso ed è un meccanismo di incentivazione per i minatori. Se l'output è più piccolo dell'input la differenza è una ricompensa per il nodo che ha convalidato il blocco.

Bitcoin Scripting. Abbiamo detto che negli output delle transazioni ci sono degli script. Nella maggior parte dei casi lo script servirà semplicemente ad assegnare l'output al corretto destinatario. Per fare questo è necessario specificare che la somma data in output può essere riscattata da una chiave pubblica il cui valore hash è specificato nello script e occorre fonire una prova di proprietà della chiave pubblica dalla quale arriva l'input, cioè una firma fatta con la corrispondente chiave privata. Di norma nello script di output troveremo le istruzione per compiere queste operazioni. Nulla vieta però altri utilizzi. Il linguaggio di scripting di bitcoin è un linguaggio molto semplice basato su uno stack. Le "istruzioni" possono essere semplici dati che vengono inseriti nello stack, oppure comandi che operano sul top dello stack. Anche se il funzionamento è semplice, le singole istruzioni possono svolgere anche azioni complesse, come il calcolo di una funzione hash o la verifica di una firma. Quindi il linguaggio in sè è semplice e compatto, ma offre un supporto per le necessarie operazioni crittografiche. La seguente tabella riporta alcune delle istruzioni de linguaggio.

Nome	Funzione
OP_DUP	Duplica il top dello stack
OP_HASH160	Calcola il valore hash (usando in sequenza SHA-256 e RIPEMD-160).
OP_EQUALVERIFY	Restituisce `ture` se gli operandi sono uguali. Altrimenti restituscie `false` e rende la transazione non valida.
OP_CHECKSIG	Controlla che la firma in input sia valida usando la chiave pubblica di input per l'hash della transazione.
OP_CHECKMULTISIG	Stesso controllo di OP_CHECKSIG ma su più firme.

Trattandosi di un linguaggio di scripting basato su uno stack, per implementarlo non servirà niente altro che uno stack. Le "istruzioni" possono essere o dati, che vengono inseriti nel top dello stack, oppure OP_CODES che specificano le istruzioni da eseguire prendendo l'operando, o gli operandi, dal top dello stack. Consideriamo, per esempio, il seguente script

<sig>
<pubKey>
OP_DUP
OP_HASH160
<pubKeyHash?>
OP_EQUALVERIFY
OP_CHECKSIG

Dove i dati sono specificati fra parentesi angolari. La Figura 6.37 mostra lo stack per l'esecuzione dello script.

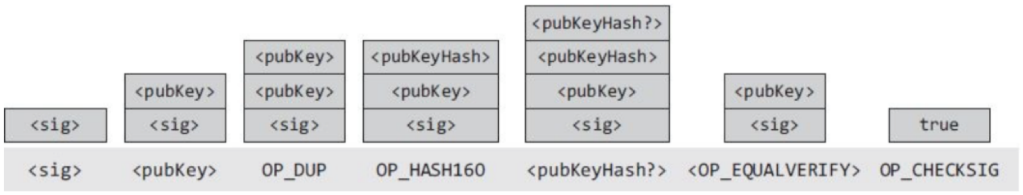

Figura 6.37: Esempio di esecuzione di uno script Bitcoin

6.8 *Note bibliografiche*

Molto del materiale presentato in questo capitolo è tratto da L1996 [26]. Il problema dei due generali, e la prova di impossibilità per il caso di sistemi deterministici con perdita di messaggi è stato presentato in [16]; l'algoritmo randomizzato è stato presentato in [32]. I risultati presentati per sistemi con guasti bizantini sono stati studiati in [27] e [24]. La prova di impossibilità per sistemi asincroni con guasti stop è dovuta a Fischer, Lynch e Paterson [13]. L'algoritmo di Ben-Or è stato presentato in [4]. L'algoritmo Paxos è dovuto a Lamport [22]; la sua iniziale stesura "archeologica" è stata rivisitata da De Prisco, Lampson e Lynch [10, 11]. L'algoritmo Paxos può essere esteso per gestire anche guasti bizantini [23].

6.9 *Esercizi*

1. Si consideri il problema dell'attacco coordinato con guasti sui canali, con la seguente variante sulla condizione di accordo: se c'è qualche processo che decide 1 allora ci devono essere almeno due processi che decidono 1 (in altre parole l'unico caso che vogliamo evitare è che un generale decida di attaccare da solo). Per $n = 2$ questa condizione è equivalente a quella originale, ma per $n \geq 3$ no. Questo problema è risolvibile? Se sì, si dia un algoritmo, se no si fornisca una prova di impossibilità.

2. Si consideri il problema dell'attacco coordinato con guasti sui canali in un sistema sincrono per il caso semplice di 2 processi p_1 e p_2 connessi da un canale di comunicazione. Il canale si comporta in maniera randomizzata e in ogni round, in maniera indipendente dagli altri round, con probabilità p entrambi i messaggi vengono consegnati e con probabilità $1 - p$ vengono entrambi perso.

Progettare un algoritmo e valutare con quale probabilità soddisfa ciascuna delle 3 proprietà richieste dal problema. Si cerchi di ottenere un algoritmo che rende quanto più alte possibile le probabilità di soddisfare le proprietà.

3. Si consideri il problema dell'attacco coordinato con guasti sui canali in un sistema sincrono per il caso semplice di 2 processi p_1 e p_2 connessi da un canale di comunicazione. Il canale si comporta in maniera randomizzata e in ogni round, in maniera indipendente dagli altri round, il messaggio spedito da p_1 a p_2 viene consegnato con probabilità q_1, e quindi perso con probabilità $1 - q_1$ mentre quello spedito da p_2 a p_1 viene consegnato con probabilità q_2 e quindi perso con probabilità $1 - q_2$.

   Progettare un algoritmo e valutare con quale probabilità soddisfa ciascuna delle 3 proprietà richieste dal problema. Si cerchi di ottenere un algoritmo che rende quanto più alte possibile le probabilità di soddisfare le proprietà.

4. Si consideri l'algoritmo FloodSet in un sistema distribuito sincrono di 4 processori p_1, p_2, p_3, p_4 che iniziano la computazione con i valori di input $1, 0, 0, 0$. Si consideri un'esecuzione α in cui il processore si ferma (guasto stop) nel primo round dopo aver spedito il proprio messaggio solo a p_2, mentre p_2 si ferma nel secondo round dopo aver spedito il messaggio a p_1 e p_3 (quindi non a p_4). Si descriva l'esecuzione dell'algoritmo specificando l'insieme di valori di input noti a ognuno dei processori in ognuno dei round; si assuma che l'algoritmo venga eseguito per il massimo numero di round.

5. Perché l'algoritmo FloodSet stabilisce l'output al round $f + 1$? Cosa succede se cambiamo l'algoritmo facendo prendere la decisione al round f? E se la prendiamo al round $f + 2$? Stabilire, in entrambi i casi, se l'algoritmo continua a funzionare fornendo una spiegazione, se funziona, o un controesempio, se non funziona.

6. Si consideri la seguente variante dell'algoritmo FloodSet: al posto dell'insieme W ogni processore tiene traccia solo del minimo valore visto, *minv*. Quindi inizialmente $minv_i = v_i$ per tutti i processori p_i, dove v_i è l'input di p_i. Alla ricezione dei messaggi x_j, il processore p_i aggiorna $minv = \min\{minv, x_j\}$, per tutti i processori p_j dai quali p_i riceve messaggi. Alla fine del round $f + 1$ p_i decide dando in output *minv*. Questo algoritmo funziona? Se sì dare una prova, se no, fornire un controesempio.

7. Si consideri l'algoritmo EIGStop. Sappiamo che esso può gestire guasti di tipo stop ma non guasti di tipo bizantino. Si fornisca un esempio in cui guasti bizantini causano una violazione della proprietà di validità.

8. Si consideri l'algoritmo EIGByz in un sistema con 7 processori. Scegliere dei valori di input per tutti i processori. Scegliere 2 processori che saranno guasti durante l'esecuzione dell'algoritmo: essi spediranno dei valori casuali al posto di quelli stabiliti dall'algoritmo: lanciare una monetina per stabilire i valori. Mostrare l'esecuzione dell'algoritmo per 3 round (uno in più del numero di guasti) e verificare che l'algoritmo funziona correttamente.

9. Si consideri l'algoritmo EIGByz. Si costruiscano delle esecuizioni in cui l'algoritmo fornisce un risultato errato nei seguenti casi:
   - 7 nodi, 2 guasti e 2 round.
   - 6 nodi, 2 guasti e 3 round.
10. Progettare un algoritmo per il problema del consenso in un sistema sincrono con al massimo f guasti stop in modo tale che venga soddisfatta anche la seguente proprietà (detta di *early stopping*): se in una esecuzione si verificano solo $f' < f$ guasti allora il tempo necessario a decidere è proporzionale a f' e non a f.
11. Si risolva l'esercizio precedente anche per il caso di guasti di tipo bizantino.
12. Si consideri il problema del consenso in sistemi asicroni con guasti stop. Si fornisca una variante dell'algoritmo di BenOr in cui i processori non guasti possono, a un certo punto, fermarsi.
13. Si consideri la segfuente variante dell'algoritmo di BenOr.

    Algoritmo ByzBenOr. Ogni processore mantiene due variabili locali x e y. Inizialmente la x contiene il valore di input del processore e la y è `null`. Ogni processore p_i esegue una serie di *fasi* numerate $1, 2, \ldots$. Il processore continua ad eseguire l'algoritmo anche dopo aver preso la decisione finale. In ogni fase s, il processore p_i esegue 2 rounds:

    Round 1: Spedisce in broadcast il messaggio (first, s, x). Aspetta l'arrivo di almeno $n - f$ messaggi della forma (first, s, v). Se almeno $n - 2f$ hanno tutti lo stesso valore v allora $y := v$, altrimenti $y :=$ `null`. Si noti che l'assunzione $n > 5f$ implica che non possono esserci due valori v in almeno $n - 2f$ messaggi.

    Round 2: Spedisce in broadcast il messaggio (second, s, y). Aspetta l'arrivo di almeno $n - f$ messaggi della forma (second, s, v).

    i. Se almeno $n - 2f$ hanno lo stesso $v \neq \mathit{null}$ allora $x = v$ e decide v se non ha già deciso.
    ii. Se almeno $n - 4f$ messaggi contengono uno stesso valore v allora $x = v$ ma non si prende una decisione.
    iii. Altrimenti, si sceglie uniformemente a caso 0 o 1 come valore di x.

    Si provi che ByzBenOr risolve il problema in sistemi asincroni con guasti bizantini nel caso $n > 5f$.
14. Si consideri l'algoritmo Paxos in un sistema con 5 processori e si descriva una esecuzione in cui occorrono 3 round per arrivare alla decisione.

Bibliografia

[1] M. Agrawal, N. Kayal, and N. Saxena. Primes is in p. *Annals of mathematics*, 160(2):781–793, 2002.

[2] J. Aspnes. Notes on randomized algorithms cpsc 469/569: Fall 2016, 2016. www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[4] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.

[5] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[6] C. J. Colbourn. The complexity of completing partial latin square. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[7] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[8] T. H. Cormen, R. L. Rivest, C. E. Leiserson, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[9] S. Dasgupta, C. H. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2008.

[10] R. De Prisco. *Revisting the Paxos Algorithm*. PhD thesis, Massachusetts Institute of Technology, 1997.

[11] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1):35 – 91, 2000.

[12] M. du Sautoy. *L'equazione da un milione di dollari*. RCS Libri, Titolo originale: The number misteries. A Mathematical Odyssey Through Every Day Life., 2010.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[16] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[17] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[18] D. R. Karger. Global min-cuts in rnc, and other ramifications of a simple min-cut algorithm. In V. Ramachandran, editor, *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 21–30. ACM/SIAM, 1993.

[19] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[20] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson Education Limited, Harlow, Essex, GB, 2014.

[21] L. Lamport. Definizione di un sistema distribuito. http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt, 1987.

[22] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] L. Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[25] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. In russo.

[26] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[27] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[28] D. Poole and A. Mackworth. *Artificial Intelligence*. Cambridge University Press, 2nd edition, 2017. Disponibile online in versione HTML: `http://artint.info/index.html`.

[29] M. Soltys. *An Introduction to the Analysis of Algorithms*. World Scientific, 2st edition, 2012.

[30] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[31] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001.

[32] G. Varghese and N. A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack. *Inf. Comput.*, 128(1):57–71, July 1996.

[33] V. V. Vazirani. *Approximation Algorithms*. Springer, 1st edition, 2001.