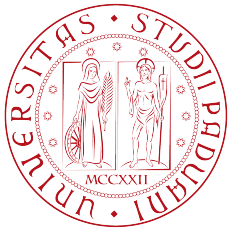
UNIVERSITÀ DEGLI STUDI DI PADOVA

Ingegneria Informatica
Sistemi Distribuiti



Consenso applicato nelle blockchain

Professore:
Mauro Migliardi

Studenti:
Davide Martini
Matricola: 1183732
Simone Nigro
Matricola: 1183535

Anno Accademico 2018-2019

Sommario

Attualmente l'uso di tecnologie distribuite nella rete è molto diffuso in vari ambiti. Uno di questi riguarda le transazioni monetarie, o la gestione di database, nei quali si sta passando da una gestione centralizzata con alti costi di mantenimento, a dei *Distributed Ledger Technology*. Questi sono molto più flessibili e hanno un costo minore ma offrono un servizio con una gestione meno sicura in quanto avendo più copie, si può presentare incoerenza tra i dati. In questa tesina verranno trattati ed analizzati i problemi del consenso e dei Bizantini per sistemi distribuiti, questi verranno messi in relazione al mondo delle blockchain, per poi interpretare i risultati ottenuti e mettere in risalto i limiti di questi sistemi.

Indice

Elenco delle figure

Capitolo 1

Problema dei Generali Bizantini

1.1 Introduzione

Il problema dei generali Bizantini sorge nelle situazioni in cui occorre trovare un accordo tra più partecipanti della rete. In queste situazioni possono verificarsi degli errori di inconsistenza tra le informazioni o comportamenti maligni dei partecipanti che possono comunicare tra loro solamente tramite l'invio di messaggi. I sistemi affidabili devono gestire malfunzionamenti di componenti, i quali possono inviare informazioni che possono creare delle situazioni di conflitto in diverse parti del sistema. Nel caso in cui, in una rete, uno o più partecipanti si comportino in maniera da creare volontariamente delle situazioni di inconsistenza, o creino situazioni di indecisione, questi vengono definiti partecipanti maligni. Questo problema del consenso nella rete, si può astrarre prendendo in considerazione un gruppo di generali Bizantini accampati con le loro truppe attorno ad una città nemica. I generali devono accordarsi per trovare un piano comune d'attacco. Il problema sorge dal fatto che uno o più generali possano essere traditori che cercheranno di confondere gli altri tramite l'invio di messaggi manipolati. Occorre quindi trovare un algoritmo che risolva questo tipo di situazioni. Un sistema deve essere in grado di colmare un malfunzionamento o un fallimento di uno o più dei suoi componenti. I generali devono possedere un algoritmo che garantisca le seguenti caratteristiche:

1. Tutti i generali leali devono decidere lo stesso piano di azione.
2. Un piccolo numero di traditori non deve poter portare ad un piano di azione pessimo per i generali leali.

Sia $v(i)$ l'informazione comunicata dall'i-esimo generale. Ognuno di essi deve usare lo stesso metodo per combinare i valori $v(1), \ldots, v(n)$ in un singolo piano d'azione, dove n è il numero di generali. La condizione 1 è soddisfatta sapendo che ogni generale usa lo stesso metodo per elaborare le informazioni, la condizione 2 viene

soddisfatta dall'uso di un metodo robusto. Un piccolo numero di traditori può influenzare la decisione degli altri generali, solamente se quelli leali sono equamente divisi nello schieramento. Un generale comunica i propri valori $v(i)$ ad un altro. Il metodo banale è che ogni i-esimo generale invii l'informazione $v(i)$ tramite un messaggero ad un altro generale. Questo metodo non funziona, poichè la condizione 1 richiede che ogni generale leale ottenga gli stessi valori $v(1), \ldots, v(n)$, non si tiene in considerazione che un traditore possa inviare valori differenti a diversi generali. Per fare in modo che la condizione 1 venga soddisfatta occorre che le seguenti affermazione siano vere:

- Ogni generale leale deve ottenere la stessa informazione $v(1), \ldots, v(n)$. Un generale non può necessariamente usare il valore $v(i)$ ottenuto direttamente dall'i-esimo generale, perchè potrebbe essere un traditore e inviare messaggi differenti a diversi generali.

- Se l'i-esimo generale è leale, il valore che invia deve essere usato da ogni generale come valore definitivo per l'informazione $v(i)$.

1.2 Soluzione tramite l'invio di messaggi autenticati

Viene analizzato il caso in cui i vari generali possano inviare messaggi riportanti la propria firma. In questo caso bisogna fare le seguenti assunzioni:

- La firma di un generale leale non può essere riprodotta ed ogni alterazione del contenuto dell'autenticazione può essere rilevato.

- Ogni generale può verificare l'autenticità della firma di un altro generale.

Nell'algoritmo sviluppato per risolvere il problema, viene assunta una funzione *choice* che viene applicata al set di ordini ricevuti per ottenere quello finale. In questo algoritmo il comandante invia un messaggio autenticato ad ognuno dei suoi tenenti, che successivamente aggiungendo la loro firma all'ordine ricevuto lo inviano agli altri tenenti, che aggiungeranno la loro firma e lo invieranno agli altri. Il solo requisito richiesto per la funzione *choice* è che se il set V consiste in un singolo elemento v, allora $choice(V) = v$. Di default $choise(\emptyset)$ viene impostata al valore $RITIRATA$. Un'implementazione possibile della funzione *choice* è data dall'elemento mediano del set V, assumendo che gli elementi siano ordinati. Dato $x : i$, i definisce il valore x firmato dall'i-esimo tenente, mentre $v : j : i$ denota il valore v firmato da j e che il valore $v : j$ viene firmato anche dal tenente i. Supponiamo che il generale di indice 0 sia il comandante e che ogni tenente i manterrà un set V_i, contenente il set di ordini

firmati che ha ricevuto. Un algoritmo che risolve questo problema è:

Algoritmo $SM(m)$

Inizialmente $V_i = \emptyset$.

1. Il comandante firma ed invia il suo valore ad ogni tenente.
2. Per ogni i:
   A. Se il tenente i riceve un messaggio della forma $v : 0$ dal comandante e non ha già ricevuto altri ordini, allora:
      a. $V_i = \{v\}$.
      b. Il tenente invia il messaggio $v : 0 : i$ ad ogni altro tenente.
   B. Se il tenente i riceve un messaggio della forma $v : 0 : j_1 : \cdots : j_k$ e $v \notin V_i$, allora:
      a. $V_i = V_i \bigcup \{v\}$.
      b. Se $k < m$, il tenente invia il messaggio $v : 0 : j_1 : \cdots : j_k : i$ ad ogni altro tenente, esclusi $j_1, \ldots, j_k$.
3. Per ogni i: quando il tenente i non riceverà più messaggi, calcolerà l'ordine definitivo ottenuto da $choice(V_i)$.

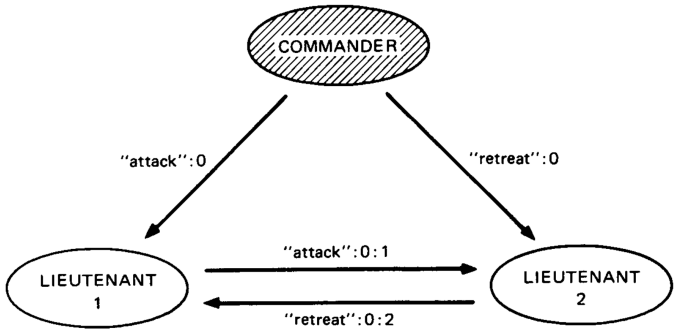


Figura 1.1. Algoritmo $SM(1)$.

Il tenente i ignora nel passo 2 ogni messaggio contenente un ordine v già contenuto nel set V_i. Per induzione su k, si dimostra che per ogni sequenza di tenenti $j_1, \ldots, j_k$ con $k \leq m$, un tenente può ricevere al massimo un messaggio della forma $v : 0 : j_1 : \cdots : j_k$ nel passo 2. Può essere usato un time-out per determinare

quando non arriveranno più messaggi dagli altri tenenti e garantire la terminazione del metodo. Nell'algoritmo $SM(m)$, un tenente firma con il suo codice identificativo per confermare la ricezione dell'ordine. Se il tenente considerato è il m-esimo ad aggiungere la firma all'ordine, allora questa autenticazione è superflua, in quanto il messaggio non viene spedito. Per ogni m, l'algoritmo $SM(m)$ risolve il problema dei Bizantini se ci sono al massimo m traditori su un totale di almeno $m + 2$ generali.

1.3 Missing Comunication Path

Nei problemi precedenti, abbiamo ipotizzato che un generale possa inviare messaggi direttamente ad ogni altro generale. Ora viene considerata la situazione in cui questo possa non verificarsi. Supponiamo che esistano delle barriere fisiche che inducano a delle restrizioni nell'invio dei messaggi. Consideriamo ogni generale come un nodo in un grafo semplice non orientato G, dove un arco tra due nodi indica che questi due generali possano scambiarsi messaggi. Nel caso estremo in cui il comandante non possa comunicare con i tenenti, il problema dei Bizantini non ammette soluzione in quanto il messaggio non può essere inviato e quindi i tenenti traditori non possono essere scoperti senza una verifica dell'ordine del generale. Nel caso siano collegati solamente i generali leali, lasciando i traditori isolati, si è in grado di ottenere una soluzione al problema dei Bizantini in quanto non ci sono intrusi nella propagazione degli ordini e questi non vengono modificati. Un'altra situazione in cui non esiste garazia di soluzione è nel caso in cui ci siano due tenenti che possano comunicare solamente con gli altri tramite traditori. Per ogni m e d, se ci sono al massimo m traditori e il sottografo dei generali leali ha diametro d, l'algoritmo $SM(m + d - 1)$ risolve il problema dei Bizantini. Il diametro di un grafo è considerato il più piccolo numero d, in modo che due nodi siano connessi da un percorso con al massimo d archi.

Capitolo 2

Problema del consenso

2.1 Introduzione

Il problema del consenso riguarda tutti i processi incaricati della gestione di dati che partecipano ad una particolare transazione per approvare oppure no l'installazione del risultato all'interno di un database. Qualsiasi decisione venga presa, tutti i gestori dei dati devono prendere la stessa decisione, in modo tale da preservare la consistenza del database. I sistemi reali sono soggetti a diversi tipi di fallimenti come un *crash* di un processo, rottura della connessione della rete e messaggi distorti o duplicati. In questi casi il problema del consenso non è semplice da risolvere. Viene riportata la trattazione che dimostra l'impossibilità di raggiungere un consenso tra processi che ammettano anche un solo fallimento della rete. Per affrontare questo problema vengono fatte diverse assunzioni, in modo da focalizzarsi sul problema direttamente connesso con il consenso e senza divagare in altre possibili configurazioni della rete. Vengono quindi fatte le seguenti ipotesi:

- Il sistema di messaggi è affidabile, invia tutti i messaggi correttamente ed esattamente una sola volta.

- I processi non hanno accesso a clock sincronizzati, quindi algoritmi basati su time-out non possono essere usati.

Lo stop di un processo in un momento non opportuno può causare disturbi durante la ricerca di un accordo tra i processi rimanenti. Per poter permettere un evento simile occorre fare delle grandi restrizioni sul tipo di fallimenti che possano essere tollerati. Non si ha la capacità di rilevare la morte di un processo, un processo cioè non è in grado di comunicare la morte di un altro o che la sua esecuzione si sta svolgendo molto lentamente in maniera anomala.

2.2 Definizioni iniziali

Ogni processo parte con un valore iniziale $\{0,\ 1\}$. Un processo che non fallisce, decide un valore in $\{0,\ 1\}$ entrando così in uno stato di decisione appropriato. Tutti i processi che non falliscono e che devono effettuare una scelta, devono ottenere lo stesso valore. I processi considerati sono modellati come degli automi che comunicano tramite l'invio di messaggi. In un passo atomico, un processo può aspettarsi di ricevere un messaggio, effettuare un calcolo locale ed inviare un set arbitrario ma finito di messaggi ad un altro processo. Questo può inviare lo stesso messaggio in un passo a tutti gli altri con la conoscenza che se un processo attivo riceve il messaggio, lo riceveranno tutti gli altri ancora attivi.

2.3 Consensus protocol

Per poter effettuare la dimostrazione, occorre avere delle notazioni comuni, in modo da poter definire chiaramente ogni possibile situazione della rete all'interno del protocollo. Viene definito un *consensus protocol* P come un sistema asincrono di N processi ($N \geq 2$). Ogni processo p ha un registro di input (di un bit) x_p e un registro di output y_p con valori in $\{b,\ 0,\ 1\}$ e un illimitato quantitativo di memoria interna. I registri di input ed output assieme al *program counter* e alla memoria interna compongono lo stato interno. Lo stato iniziale definisce dei valori iniziali fissati per tutto lo stato interno con il registro di output settato al valore b. Gli stati in cui il registro di output assume il valore 1 o 0 sono distinti da uno stato di decisione. Il processo p si comporta in maniera deterministica in accordo con la funzione di transizione. Un messaggio è una coppia $(p,\ m)$ dove p è il nome del processo di destinazione ed m è il valore del messaggio contenuto in un insieme finito M. Il sistema di invio messaggi mantiene un set, chiamato *message buffer*, di tutti i messaggi che sono stati inviati ma non ancora ricevuti. Questo sistema contiene due operazioni:

- *send*$(p,\ m)$: inserisce il messaggio $(p,\ m)$ nel *message buffer*.
- *receive*(p): elimina alcuni messaggi $(p,\ m)$ dal *buffer* e restituisce il valore m nel caso $(p,\ m)$ sia stato consegnato, o restituisce un valore speciale $(\emptyset)$ che lascia il *buffer* invariato.

Una configurazione consiste nello stato interno di ogni processo e nel contenuto del *message buffer*. In una configurazione iniziale ogni processo comincia la sua esecuzione e non presenta alcun messaggio nel *message buffer*. Uno step porta da una configurazione ad un'altra, lo step primitivo è il più breve possibile ed è rappresentato da un processo p.

Data una configurazione C, uno step è composto da due fasi:

1. *receive*(p) viene eseguito sul *message buffer* restituendo il valore m in $M \cup \{\emptyset\}$.
2. In base allo stato interno contenuto in C e al valore m, il processo p entra in un nuovo stato interno ed invia un set distinto di messaggi agli altri processi.

Lo step è completamente determinato dalla coppia $e = (p,\ m)$ che viene chiamata evento. $e(C)$ denota la configurazione risultante e viene detto che l'evento e viene applicato alla configurazione C. L'evento $(p,\ \emptyset)$ può sempre essere applicato a C, e quindi è sempre possibile per un processo avere un altro step. Una schedulazione da C è una sequenza finita o infinita σ di eventi che possono essere applicati, uno alla volta, partendo da C. La sequenza associata di step viene chiamata run e se σ è finita, $\sigma(C)$ viene chiamata configurazione raggiungibile da C. Una configurazione raggiungibile da una configurazione iniziale viene detta accessibile.

Lemma 1. *Si suppone che da una certa configurazione C le schedulazioni σ_1 e σ_2 portino alle rispettive configurazioni C_1, C_2. Se i set di processi contenuti in σ_1 e σ_2 sono disgiunti, allora σ_2 può essere applicato a C_1, σ_1 può essere applicato a C_2 ed entrambi produrranno la stessa configurazione C_3.*

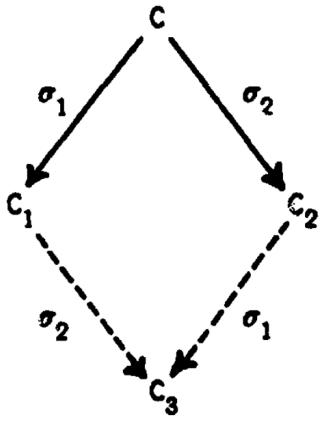


Figura 2.1. Lemma 1.

Dimostrazione. Le schedulazioni σ_1 e σ_2 non interagendo, possono essere applicate liberamente. □

Una configurazione C ha un valore di decisione v se qualche processo p è in uno stato di decisione con $y_p = v$. Un *consensus protocol* è parzialmente corretto se soddisfa le due condizioni seguenti:

1. Una configurazione accessibile non deve avere più di un valore di decisione.

2. Per ogni $v \in \{0,\ 1\}$, qualche configurazione accessibile ha valore di decisione v.

Un processo p non fallisce in un run se è in grado di essere eseguito infinite volte, in caso contrario, fallisce. Un run viene definito ammissibile se al massimo un processo fallisce, ed è definito decisivo se alcuni processi raggiungono uno stato di decisione. Un protocollo P è totalmente corretto a dispetto di un fallimento se è parzialmente corretto ed ogni run ammissibile è un run decisivo.

Teorema 1. *Non esiste un consensus protocol totalmente corretto che ammetta un fallimento.*

Dimostrazione. Assumiamo per assurdo che P sia un *consensus protocol* totalmente corretto a dispetto di un fallimento.
L'idea è quella di mostrare alcune circostanze per le quali il protocollo rimane indecisivo.
Questo viene evidenziato in due passi:

1. Esiste una configurazione iniziale nella quale la decisione non è predeterminata (i processi sono deterministici).

2. Viene costruito un run ammissibile che eviti che uno step porti il sistema ad una particolare configurazione di decisione.

Partiamo con la dimostrazione definendo C come una configurazione e V il set di valori di decisione delle configurazioni raggiungibili da C. V è quindi l'insisme di tutti i valori possibili che possono essere raggiunti applicando a C qualsiasi tipo di evento. C è bivalente se $|V| = 2$, mentre è univalente se $|V| = 1$. Nel caso sia univalente, C viene detta 0-valente o 1-valente in accordo con il valore di decisione corrispondente, in quanto V può contenere solo il valore 0 o il valore 1. Per la totale correttezza di P e per il fatto che ci sono sempre run ammissibili, $V \neq \emptyset$. Per continuare la dimostrazione, si ha bisogno di provare altri Lemmi.

Lemma 2. *Il protocollo P possiede una configurazione iniziale bivalente.*

Dimostrazione. Viene ipotizzato che P non possegga una configurazione iniziale bivalente. In questo caso, P deve avere entrambe le configurazioni 0-valente e 1-valente, in quanto la correttezza parziale è stata ipotizzata. Due configurazioni inziali vengono chiamate adiacenti se differiscono solamente nel valore iniziale x_p di un singolo processo p. Esiste quindi una configurazione C_0, 0-valente, adiacente ad una configurazione iniziale C_1, 1-valente. Viene definito p, il processo con il valore iniziale differente tra le due configurazioni.
Consideriamo un run decisivo ammissibile per C_0 nel quale il processo p non viene eseguito e σ la schedulazione associata a questo run. σ può essere applicata anche a

C_1 rendendo i due run identici ad eccezione dello stato interno del processo p. Non avendo applicato p, si nota che i due run raggiungono lo stesso stato di decisione. Se il valore di decisione è 1, allora C_0 è bivalente, se il valore di decisione è 0, risulta bivalente la configurazione C_1. Queste due situazioni contraddicono l'assunzione della non esistenza di una configurazione iniziale bivalente. □

Lemma 3. *Sia C una configurazione bivalente di P e sia $e = (p,\ m)$ un evento applicabile a C. Viene definito* **C** *come l'insieme di tutte le configurazioni raggiungibili da C senza che sia applicato l'evento e. Sia* $\mathbf{D} = e(C) = \{e(E) \mid E \in \mathbf{C} \wedge e \text{ applicabile ad } E\}$. *Allora* **D** *contiene una configurazione bivalente.*

Dimostrazione. Abbiamo quindi una configurazione C e quasiasi operazione venga eseguita su questa configurazione può produrre due risultati differenti. Consideriamo un evento applicabile a C. **C** è l'insieme di tutte le configurazioni ottenute partendo da C applicando qualsiasi evento tranne e. **D** è l'insieme di tutte le configurazioni in **C** a cui dopo viene applicato e. Considero una configurazione ottenuta applicando x eventi partendo da C, a cui successivamente viene applicato e. Occorre dimostrare che questa configurazione finale è bivalente.
L'evento e per definizione è applicabile a C, per definizione di **C** e per il fatto che i messaggi possono essere ritardati arbitrariamente, ed è applicabile ad ogni configurazione $E \in \mathbf{C}$.
Viene assunto che **D** non contenga configurazioni bivalenti, quindi ogni configurazione $D \in \mathbf{D}$ è univalente. Dimostriamo per assurdo.
Sia E_i una configurazione i-valente raggiungibile da C, $i = 0,\ 1$ (E_i è univalente). Se $E_i \in \mathbf{C}$, allora $F_i = e(E_i) \in \mathbf{D}$, in altre parole l'evento e viene applicato ad E_i, quindi esiste $F_i \in \mathbf{D}$, tale che F_i sia raggiungibile da E_i. F_i è i-valente per ipotesi e uno tra E_i e F_i è raggiungibile dall'altro. $F_i \in \mathbf{D}$, $i = 0,\ 1$, allora **D** contiene sia la configurazione 1-valente che 0-valente. Due configurazioni vengono chiamate vicine

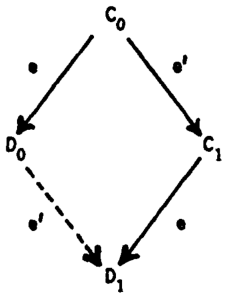


Figura 2.2. Lemma 3.

se una è il risultato dell'altra dopo che le è stato applicato uno step. Allora esistono due configurazioni vicine $C_0,\ C_1 \in \mathbf{C}$, tali che $D_i = e(C_i)$, con D_i i-valente con

$i = 0,\ 1$. Essendo C_0 e C_1 vicine possiamo dire che $C_1 = e'(C_0)$, con $e' = (p',\ m')$. A questo punto ci si può trovare in due casi differenti:
CASO 1. Se $p' \neq p$, $D_1 = e'(D_0)$ (per il Lemma 1). Conosco che D_1 e D_0 sono configurazioni vicine, quindi esiste un evento che trasforma una configurazione in un'altra. Questa situazione è impossibile perchè la configurazione successiva ad una configurazione 0-valente deve essere ancora 0-valente.
CASO 2. Se $p' = p$, consideriamo un run decisivo al quale p non prende parte, partendo da C_0 con schedulazione σ e definiamo $A = \sigma(C_0)$. σ è applicabile a D_i (per il Lemma 1) e si ottiene una configurazione $E_i = \sigma(D_i)$, i-valente, $i = 0,\ 1$. Se applichiamo l'evento e ad A, otteniamo $e(A) = E_0$ ed $e(e'(A)) = E_1$. A risulta essere bivalente e quindi è impossibile che A sia univalente.
Abbiamo ottenuto una contraddizione, quindi **D** contiene configurazioni bivalenti. □

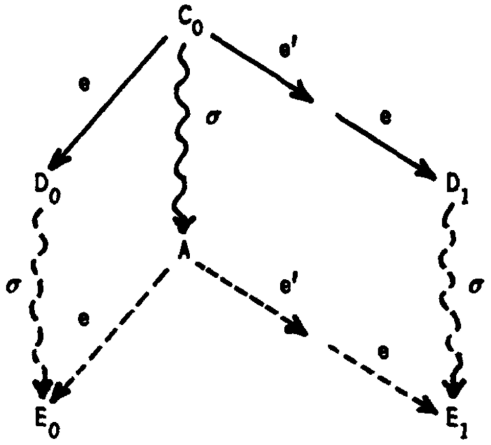


Figura 2.3. Non totale correttezza del protocollo P.

Terminiamo ora la dimostrazione sapendo che qualsiasi run decisivo applicato ad una configurazione bivalente raggiunge una configurazione univalente, essendo decisivo e possiamo considerarlo come uno step. A questo punto vogliamo verificare che è sempre possibile avere un run che eviti questo step, in modo che sia un run non decisivo ma ammissibile.
Questo run viene costruito in varie fasi partendo da una configurazione iniziale.
Viene mantenuta una coda di processi, inizialmente in ordine arbitrario e il *message buffer* viene ordinato in base al tempo nel quale i messaggi sono stati spediti. Ogni fase consiste in uno o più step. Le fasi terminano quando il primo processo presente nella coda compie uno step. Se la coda dei messaggi non è vuota, il primo messaggio viene ricevuto dal processo. Questo viene poi spostato alla fine della coda dei processi. In una sequenza infinita di fasi, ogni processo compie infiniti step e riceve

ogni messaggio che gli è stato invitato. Questo run risulta essere quindi ammissibile. Partendo da questo run dobbiamo evitare il raggiungimento di uno stato di decisione.
Sia C_0 una configurazione iniziale bivalente da cui comincia l'esecuzione, in cui ogni fase risulta partire da una configurazione bivalente. Supponiamo che la configurazione C sia bivalente, che il processo p sia all'inizio della coda e che m sia il primo messaggio per p nel *message buffer* di C se sono presenti messaggi, $\emptyset$ altrimenti. Sia $e = (p,\ m)$, per il Lemma 3, c'è una configurazione bivalente C' raggiungibile da C attraverso una schedulazione nella quale l'evento e viene applicato per ultimo. La sequenza corrispondente di step definisce la fase.
Finchè ogni fase termina con una configurazione bivalente, ogni fase è la costruzione di una successione infinita di schedulazioni. Il run risultante è ammissibile ma nessuna decisione viene raggiunta. Quindi otteniamo il risultato della nostra dimostrazione, cioè che P non è totalmente corretto. □

2.4 Initially Dead Processes

Viene proposta una soluzione per il problema del consenso nel caso siano presenti N processi. Di questi, la maggioranza non deve essere fallita e nessun processo può morire durante l'esecuzione del protocollo. Non si è a conoscenza inizialmente di quali processi siano morti e quali siano ancora attivi. Questo protocollo viene eseguito in due fasi:

- Nella prima fase i processi costruiscono un grafo diretto G nel quale ogni nodo corrisponde ad un processo. Ogni processo invia in maniera *broadcast* un messaggio contenente il suo numero identificativo e il suo valore iniziale, rimanendo in attesa di ricevere messaggi da $L-1$ processi, dove $L = \left\lceil \frac{N+1}{2} \right\rceil$. G ha un arco da i a j solamente se j può ricevere messaggi da i. G, ha quindi un numero di archi entranti in un nodo pari a $L-1$.

- Nella seconda fase i processi costruiscono G^+, la chiusura transitiva di G, definita come un grafo composto dagli stessi vertici di G, con la caratteristica che se G ha un cammino orientato da a a b, G^+ ha un arco orientato da a a b. Alla fine di questa fase ogni processo k è a conoscenza di tutti gli archi (i, j) incidenti su k contenuti in G^+. Ogni k conosce inoltre i valori iniziali per ogni processo j.

Per il calcolo di G^+, ogni processo invia tramite *broadcast* a tutti gli altri processi il suo numero identificativo e il valore iniziale, assieme ai nomi degli $L-1$ processi di cui è a conoscenza al termine della prima fase. A questo punto, il processo attende finchè non avrà ricevuto tutti i messaggi della prima e seconda fase da tutti

gli altri che conosce. Inizialmente il processo k sarà a conoscenza solamente degli $L-1$ processi collegati direttamente, ma successivamente potrebbe scoprire altri nodi presenti nella rete.
A questo punto ogni processo conosce tutti i nodi e tutti gli archi in G e può calcolare tutti gli archi presenti in G^+. Questo permette di determinare quali nodi appartengano ad una clique iniziale in G^+. Finchè ogni nodo in G^+ ha almeno $L-1$ predecessori ci può essere solamente una clique iniziale con cardinalità almeno L ed ogni processo che completa la seconda fase è a conoscenza del set esatto di questi processi.
Ogni processo prende una decisione basata sui valori iniziali di tutti i processi, presenti nella clique di partenza, usando una regola comune. Finchè tutti i processi conoscono questi valori, otterranno la stessa decisione.

Teorema 2. *Esiste un protocollo, per il consenso parzialmente corretto, nel quale tutti i processi attivi raggiungono una decisione a patto che durante l'esecuzione nessun processo muoia e che inizialmente almeno la maggioranza dei processi sia attiva.*

Capitolo 3

Blockchain

3.1 Introduzione

La blockchain può essere considerata una tecnologia che appartiene alla categoria delle tecnologie di archivi distribuiti (*Distributed Ledger Technology*, DLT). Le DLT possono essere definite come un insieme di sistemi caratterizzati dal fatto di fare riferimento a un registro distribuito, governato in modo da consentire l'accesso e la possibilità di effettuare modifiche da parte di più nodi di una rete. Un registro distribuito è un sistema nel quale non è presente un sistema centrale di validazione. Le DLT prevedono pertanto un meccanismo di validazione a sua volta distribuito basato sul concetto di consenso. Si possono definire le blockchain come un esempio di *Distributed Ledger Technology* caratterizzate da un registro impostato e strutturato in modo da gestire le transazioni all'interno di una catena di blocchi. Ciascun blocco si aggiunge alla catena sulla base di un processo basato sul consenso distribuito su tutti i nodi della rete, ovvero con la partecipazione di tutti i nodi che vengono chiamati a contribuire alla validazione delle transazioni presenti in ciascun blocco e alla loro inclusione nel registro.

3.2 Modello distribuito per blockchain

Viene considerato un grafo $G = (V,\ E)$, nel quale i processi V sono connessi tra loro tramite dei link fissati E. I processi fanno parte di un sistema di blockchain S e questi possono comportarsi come dei client che vogliono emettere una o più transazioni al sistema e/o come server tramite *mining*, un'azione che cerca di combinare le transazioni in blocchi. Ogni processo possiede un singolo account e una transazione emessa dal processo p_i è considerata un trasferimento di attività digitali o monete dall'account sorgente, al processo p_j destinatario, con $p_j \neq p_i$. Ogni transazione è identificata in maniera univoca e viene inviata in maniera *broadcast* a tutti i processi

nel miglior modo possibile. Un problema può sorgere in quanto una transazione emessa più volte, può creare diverse transazioni distinte. I processi che compongono il *consensus protocol* sono chiamati *miner* e fanno partire il protocollo tramite una funzione che permette loro di proporre nuovi blocchi. I processi decidono su un nuovo blocco ad un dato indice in base ad una funzione che specifica il tipo di sistema *proof-of-work* blockchain in uso. La potenza di calcolo dei *miner* viene espressa come *minning power* e viene definita la potenza totale dei *miner* t come la somma delle potenze di tutti quelli presenti in V. Ognuno di questi cerca di raggruppare un set T di transazioni, di cui riceve la proposta, in un blocco b che contiene T fino al momento che le transazioni T non vanno in conflitto e i bilanci degli account non vanno in passivo.

Algorithm 1 The general proof-of-work blockchain consensus algorithm at process p_i

1: $\ell_i = \langle B_i, P_i \rangle$, the local blockchain at node p_i is a directed acyclic
2: graph of blocks B_i and pointers P_i
3: b, a block record with fields:
4: *parent*, the block preceding b in the chain, initially $\perp$
5: *pow*, the proof-of-work nounce of b that solves the cryptopuzzle, initially $\perp$
6: *children*, the successor blocks of b in the chain

7: propose()$_i$: ▷ *function invoked to solve consensus*
8: **while** true **do** ▷ *do forever*
9: *nounce* = local-random-coin() ▷ *toss a local coin to get a nounce*
10: create block b : $b.parent$ = last-block(ℓ_i) and $b.pow$ = *nounce* ▷ *create a new block*
11: **if** solve-cryptopuzzle(*nounce*, b) **then** ▷ *if the chosen nounce solves the puzzle*
12: broadcast($\langle \{b\}, \{\langle b, b.parent \rangle\} \rangle$) ▷ *broadcast to all (including to himself)*

13: deliver($\langle B_j, P_j \rangle$)$_i$: ▷ *upon reception of blocks*
14: $B_i \leftarrow B_i \cup B_j$ ▷ *update vertices of blockchain*
15: $P_i \leftarrow P_i \cup P_j$ ▷ *update edges of blockchain*
16: $\langle B'_i, P'_i \rangle \leftarrow$ get-main-branch() ▷ *recompute the main branch*
17: **if** $b_0 \in B'_i \wedge \exists b_1, ..., b_m \in B_i : \langle b_1, b_0 \rangle, \langle b_2, b_1 \rangle ..., \langle b_m, b_{m-1} \rangle \in P_i$ **then** ▷ *if enough blocks*
18: decide(b_0) ▷ *consensus is reached*

Figura 3.1. Algoritmo 1.

Il grafo G è statico, cioè nessun processo può entrare o uscire dal grafo nonostante i processi possano fallire. I *miner* devono risolvere un *cryptopuzzle* prima della

creazione di un nuovo blocco. Il *miner* seleziona ripetutamente un *nonce* (in crittografia questo termine indica un numero, generalmente casuale, che ha un utilizzo unico) ed applica una funzione pseudo-casuale a questo blocco e al *nonce* selezionato, finchè non ottiene un risultato minore di una data soglia. Se ha successo, il *miner* crea un blocco che contiene il *nonce* vincitore come *proof-of-work* e inoltre fissa l'indice del blocco inviandolo via *broadcast*.
Questa funzione di *broadcast* richiama la funzione con cui è stato invocato ogni processo p_j corretto (incluso p_i, se risulta corretto, che ha generato il blocco e il puntatore a questo) su ricezione del messaggio di *broadcast*.
La difficoltà di questo *cryptopuzzle* viene definta dalla soglia che limita la frequenza con cui nuovi blocchi vengono generati dalla rete.
Ci sono vari modelli di fallimenti che separano i processi falliti da quelli corretti. In un *crash failure model*, i processi che falliscono lo possono fare in un determinato punto fermando la loro elaborazione e la loro comunicazione per il resto dell'esecuzione. In un *Byzantine failure model*, i processi che falliscono possono non seguire le specifiche del protocollo ma avere un comportamento arbitrario. Viene considerato un *Byzantine failure model* e si assume che la presenza di un avversario, che può controllare i processi, possa farlo soltanto per una piccola frazione $\rho < 0.5 \cdot t$ della potenza totale dei *miner* del sistema. Questi processi vengono chiamati maligni o Bizantini. Si può ipotizzare inoltre che l'avversario possa decidere di distruggere le comunicazioni in un sottoinsieme di archi selezionati E_0 del grafo di comunicazione G.

3.3 Il problema del consenso nelle blockchain

Per un dato indice, tutti i processi corretti si accordano su un unico blocco di transazioni a quell'indice. I nodi possono proporre blocchi differenti per lo stesso indice perchè i *miner* remoti risolvono il *cryptopuzzle* nel momento in cui altri *miner* ricevono i blocchi dal resto della rete.

Definizione 1. *Il problema del consenso dei Bizantini è quello di garantire la congiunzione di queste tre proprietà per un dato indice:*

1. *Accordo (due processi corretti non possono decidere blocchi diversi).*
2. *Terminazione (tutti i processi corretti devono decidere un blocco).*
3. *Validità (il blocco deciso è un blocco proposto da un processo).*

I sistemi di blockchain operano su una rete che si ipotizza sincrona, ogni messaggio viene inviato in un periodo di tempo, ma questo potrebbe essere non realistico. Sfortunatamente il problema del consenso è irrisolvibile nelle reti asincrone anche

in caso di un solo *crash*. Per far fronte a questa impossibilità, varie proposte promettono di garantire il consenso Bizantino tramite garanzie probabilistiche ottenute con la randomizzazione.
Una proposta usa il consenso Bizantino nel caso asincrono e lo fa terminare in un tempo costante atteso con la combinazione di firme digitali e un leader leale tramite la randomizzazione.
Nei Bitcoin la probabilità di un accordo su un blocco ad un dato indice della blockchain aumenta esponenzialmente con la profondità della catena. I Bitcoin risolvono la fase di accordo, ma non la terminazione, con probabilità 1 quando il tempo di esecuzione è infinito.
Se dopo un tempo finito, o un numero finito $i + k$ di blocchi, dove il blocco $i + k$ è stato proposto da un *miner*, il protocollo Bitcoin afferma che il consenso è stato ottenuto per il blocco all'indice i, ottenendo una probabilità per l'accordo < 1 il quale non viene garantito.

3.4 Requisito di terminazione

L'applicazione deve rispondere al cliente che se emette una transazione per comprare dei beni, deve ricevere una risposta che indica il successo o il fallimento della transazione.
Viene adottata l'idea di garantire la terminazione con una certa probabilità. Se la terminazione non viene garantita in maniera deterministica, l'applicazione potrebbe non rispondere.
La creazione di blocchi distinti allo stesso indice della blockchain, come una violazione di un accordo, viene rapppresentata in Figura 3.2. Sotto l'ipotesi della rete sincrona, una riorganizzazione garantisce comunque con alta probabilità che il blocco all'indice i sia deciso in modo univoco quando la catena raggiunge la profondità di $i+k$. Le applicazioni possono considerare la profondità $i+k$ come la terminazione per il consenso del blocco all'indice i e indicano che la transazione di questo venga consegnata con successo.
Selezionando un parametro k appropriato, Bitcoin può risolvere il *Monte Carlo Byzantine consensus problem*. Per definizione degli algoritmi Monte Carlo, questi garantiscono la loro terminazione ma possono restituire un risultato incorretto.

Definizione 2. *Il Monte Carlo Byzantine consensus problem garantisce la congiunzione di queste tre proprietà:*

1. *Accordo probabilistico (due processi corretti non decidono differenti blocchi con probabilità $P(agreement) > \delta$).*
2. *Terminazione (tutti i processi corretti devono decidere un blocco).*

3. Validità (il blocco deciso è un blocco proposto da qualche processo).

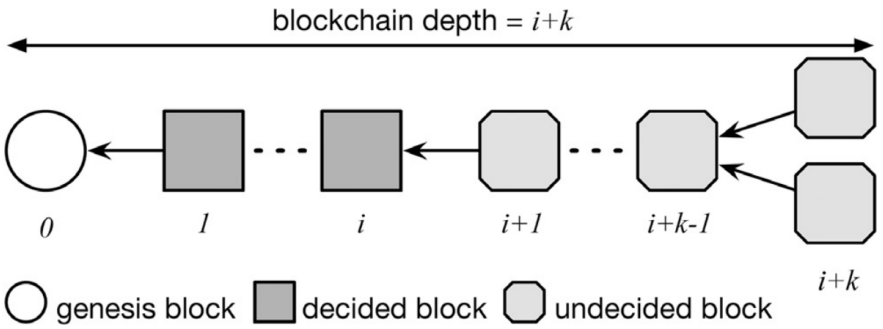


Figura 3.2. Struttura di una blockchain.

Questa variante del consenso vuole garantire che la blockchain restituisca un risultato al cliente (anche se questo potrebbe essere incorretto). Un venditore potrebbe attendere per un determinato periodo nel quale osserva ogni possibile invalidazione della transazione dalla blockchain. Dopo questo periodo fissato, se nessuna invalidazione è apparsa, la transazione viene considerata valida.
Nel peggior caso, il venditore potrebbe sbagliare e la transazione potrebbe eventualmente essere considerata invalida, ma senza la comunicazione al venditore. In questo caso il venditore perderebbe i suoi beni. Questo caso accade però con una probabilità sufficientemente bassa da garantire un buon risultato sull'emissione delle transazioni.

3.5 Directed Acyclic Graph (DAG)

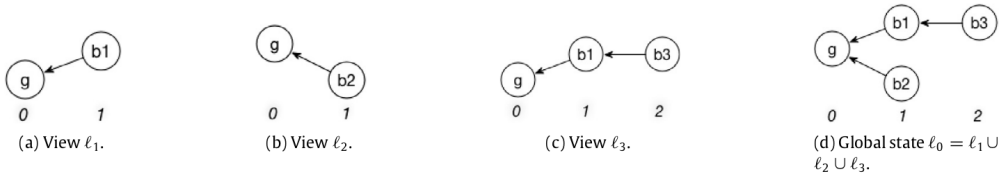
Sia la blockchain un grafo aciclico diretto (DAG) $l = (B,\ P)$ dove i blocchi di B puntano ad ogni altro tramite i puntatori P ed esiste un blocco speciale $g \in B$ chiamato *genesis block*.

1. *Collision-resilience assumption*: la funzione di hash usata per calcolare la codifica del blocco è resiliente alle collisioni e il contenuto di ogni blocco è unico.

2. *Unique-pointer-per-block assumption*: ogni blocco *non-genesis* contiene esattamente una hash di un altro e si considera che il suo grado in uscita sia pari a 1.

L'Algoritmo 1, riportato in Figura 3.1, descrive la progressiva costruzione della blockchain ad un particolare nodo p_i su ricezione di alcuni blocchi dagli altri processi.

I blocchi nuovi vengono aggregati semplicemente ai blocchi già noti ed ognuno di questi viene associato ad un indice fissato. Il *genesis block* possiede l'indice 0.



(a) View ℓ_1. (b) View ℓ_2. (c) View ℓ_3. (d) Global state $\ell_0 = \ell_1 \cup \ell_2 \cup \ell_3$.

Figura 3.3. Fork.

Nelle situazioni l_1, l_2 ed l_3, i processi possono avere differenti viste dello stato corrente della blockchain. Due *miner* p_1 e p_2 possono minare due differenti blocchi chiamati b_1 e b_2. Se il blocco b_1 o il blocco b_2 non sono stati propagati in tempo ai processi p_1 e p_2, entrambi i blocchi punteranno allo stesso blocco g. Essendo il ritardo presente nella rete non prevedibile, un terzo nodo p_3 potrebbe ricevere il blocco b_1 e minarne uno nuovo senza essere a conoscenza di b_2. I tre processi p_1, p_2 e p_3 hanno tre differenti viste locali della stessa blockchain, chiamate $l_1 = (B_1,\ P_1)$, $l_2 = (B_2,\ P_2)$, $l_3 = (B_3,\ P_3)$. Ci riferiamo alla blockchain globale come un grafo diretto aciclico $l_0 = (B_0,\ P_0)$ che rappresenta l'unione di queste blockchain locali, $l_0 = l_1 \cup l_2 \cup l_3$, con:

$$\begin{cases} B_0 = \cup\ b_i \\ P_0 = \cup\ p_i \end{cases} \tag{3.1}$$

Possiamo definire il punto dove blocchi distinti della blockchain globale hanno lo stesso predecessore viene chiamato *fork*. Per risolvere i *fork* e definire uno stato di accordo in modo deterministico da tutti i processi, un sistema di blockchain deve selezionare un *main brach* come un'unica sequenza di blocchi.

3.6 Bitcoin Consensus Algorithm

Nel caso del protocollo Bitcoin, la difficoltà dei *cryptopuzzle* genera solitamente un blocco ogni 10 minuti. Il vantaggio di questo lungo periodo è la produzione di pochi punti di *fork* perchè i blocchi sono minati raramente e durante il periodo di calcolo vengono propagati a tutti gli altri nodi della rete. *Bitcoin consensus protocol* decide un particolare blocco ad un certo indice e la scelta di un parametro m. Quando si è in presenza di un *fork*, il protocollo Bitcoin lo risolve selezionando il branch più profondo come il *main branch*. Dopo questo calcolo, ogni processo p_i ottiene il *main branch* e quindi tutti i processi ottengono la stessa vista della blockchain.

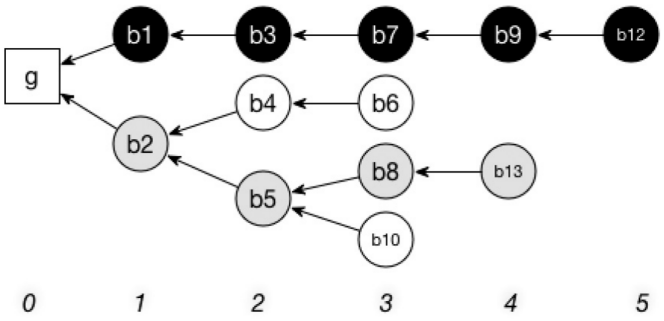


Figura 3.4. *Main branch* Bitcoin (nero), *main branch* Ethereum (grigio).

3.7 Ethereum Consensus Protocol

A differenza di Bitcoin, Ethereum genera un blocco ogni 12-15 secondi, questo favorisce la presenza di *fork* e i *miner* continuano a generare blocchi prima di venire a conoscenza degli altri già presenti, a causa dell'alto rate di generazione e il tempo di propagazione.
GHOST (*Greedy Heaviest Observed Subtree*) *consensus algorithm* viene usato da Ethereum. Questo protocollo seleziona iterativamente, come successore di un blocco, la radice del sottoalbero che contiene il maggior numero di nodi.

3.8 Blocchi decisi e transazioni approvate

Un sistema S di blockchain deve definire quando un blocco ad un dato indice viene accordato dai processi. Deve quindi definire un punto nella sua esecuzione dove un prefisso del *main branch* può essere ragionevolmente considerato come persistente. Più precisamente deve esistere un parametro m, deciso dal sistema S, in modo che ogni applicazione consideri un blocco come deciso e la transazione come emessa. Questo parametro è tipicamente $m_{bitcoin} = 5$ e $m_{ethereum} = 11$.

Definizione 3. *Sia $l_i = (B_i,\ P_i)$ la vista della blockchain al nodo p_i nel sistema S. Una transazione tx per essere localmente approvata da p_i, deve ottenere la congiunzione delle seguenti proprietà nella vista l_i di p_i:*

1. *La transazione tx deve essere in un blocco $b_0 \in B_i$ del main branch del sistema S.*

2. *Dovrebbe esserci una sottosequenza di m blocchi $b_1,\ \ldots,\ b_m$ collegata dopo il blocco b.*

Una transazione tx è emessa se esiste un processo p_i dove tx è localmente consegnata. La prima proprietà è necessaria affinchè i processi possano accordarsi sul *main branch*, che definisce lo stato corrente degli account nel sistema. I blocchi che non fanno parte del *main branch* verrebbero ignorati.
La seconda proprietà invece è necessaria per garantire che i blocchi e le transizioni che sono attualmente nel *main branch* siano persistenti e vi rimangano.

3.9 Problemi di sicurezza per blockchain

Le proprietà di sicurezza dei principali sistemi di blockchain possono essere violate. I ritardi introdotti dalla rete aumentano la probabilità di violazione degli accordi tra i *miner* della blockchain. Occorre quindi trovare un *tradeoff* tra accordo e terminazione. Una soluzione potrebbe far dipendere la terminazione dai fattori ambientali, come i ritardi della rete e le variazioni di potenza dei *miner*, in modo da garantire una soglia minima di successo.
Sfortunatamente esistono applicazioni semplici che fanno affidamento su parametri fissati di questi fattori, come l'attesa per un numero fissato di m blocchi minati, per decidere la terminazione.
Ci sono più modi in cui i processi Bizantini possono variare i ritardi e la potenza dei minatori, per produrre più transazioni simultanee sia in Bitcoin che in Ethereum. Le soluzioni esistenti per il consenso sono inefficienti a causa del problema che viene risolto in quanto vengono applicate delle restrizioni per un sottoproblema.
Assumendo che ogni processo corretto proponga un valore valido, ognuno di questi deve decidere su un valore in modo che le seguenti proprietà vengano soddisfatte.

Definizione 4. *Una soluzione del blockchain Byzantine consensus problem deve garantire la congiunzione di queste tre proprietà per un dato indice:*

1. *Accordo (due processi corretti devono decidere gli stessi blocchi).*
2. *Terminazione (tutti i processi corretti devono decidere un blocco).*
3. *Validità (un blocco deciso è valido).*

La prima proprietà permette alla blockchain di decidere un blocco di transazioni che sono state proposte dai processi Bizantini.
Per risolvere il problema del consenso per le blockchain occorre essere in grado di riconoscere un blocco minato da processi Bizantini oppure da processi corretti. Questo tipo di operazione permette di decidere se un blocco di transazioni può essere considerato valido oppure no. Senza questa distinzione sulla provenienza dei blocchi non si è in grado di distinguere le transazioni valide da quelle inviate da processi Bizantini.

Per risolvere il problema classico del consenso dei Bizantini occorre limitare a 1 il numero di blocchi decisi si $n - t$ blocchi di transazioni proposti dai partecipanti corretti. Per risolvere il consenso nelle blockchain, il blocco deciso potrebbe rappresentare l'unione di tutti gli $n - t$ blocchi che sono stati proposti e verificati come validi.

Capitolo 4

Conclusioni

Il problema del consenso risulta essere molto attuale e si sono trovate delle tecniche che sotto alcune restrizioni del problema possono risolverlo. Se viene considerato il problema generale nella sua interezza, la dimostrazione presentata in [1] rimane ancora valida.
La blockchain riesce comunque, date le sue assunzioni, a risolvere il problema del consenso con probabilità 1 a discapito di un tempo infinito di esecuzione. Si deve quindi compiere una scelta tra accordo e tempo di esecuzione, trovando un *tradeoff* tra questi due aspetti garantendoli entrambi. Questo può portare purtroppo ad avere dei casi nei quali l'accordo, non essendo garantito, può causare delle perdite di informazioni o di denaro.

Bibliografia

[1] M. J. Fischer, N. A. Lynch, M. S. Paterson, *Impossibility of Distributed Consensus with One Faulty Process*, Journal of the ACM, 32(2):374-382, April 1985.

[2] L. Lamport, R. Shostak, M. Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, Pages 382-401

[3] V. Gramoli, *From blockchain consensus back to Byzantine consensus*, Elsevier B.V, September 2017.