Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea triennale in Informatica

Un'implementazione di Blockchain in Go

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Filippo Bregoli

Correlatore:
Dott.ssa
Adele Veschetti

Parole chiave: blockchain, consenso distribuito, sistemi distribuiti, go

II Sessione
Anno Accademico 2017-2018

Prefazione

Questa tesi viene redatta con l'obiettivo di creare una simulazione di blockchain e di inserirla all'interno di una cornice industriale realistica: quella della raccolta differenziata porta a porta. Viene a tal proposito presentato il caso di Tesium, azienda fittizia occupata nel campo appena citato, che desidera dotarsi di blockchain per registrare in maniera distribuita e in tempo reale lo svolgersi dell'attività di raccolta dei bidoni dei rifiuti. Della blockchain risultante vengono quindi descritti i dati che la attraversano e il protocollo di consenso distribuito basato su Bully Election Algorithm e progettato ad hoc per l'occasione. La simulazione offerta è implementata attraverso l'utilizzo del linguaggio di programmazione Go, linguaggio sviluppato dagli ingegneri Google.

Indice

Capitolo 1

Introduzione

Solo il tempo sarà in grado di dire se blockchain sarà una supernova o una meteora nella fitta galassia delle innovazioni informatiche. Allo stato attuale, però, è impossibile negare la forza dirompente con cui questo nuovo modo di concepire l'informatica sta cercando di imporsi sul mercato delle idee, nel tentativo di raggiungere un buon numero di consensi.
Cosa può rendere blockchain affascinante?
I più estranei al mondo dell'informatica vedono le potenzialità che questa offre in alcune sue implementazioni legate al mondo finanziario o a quello politico: la capacità di eseguire scambi commerciali in ambienti sicuri ma senza necessità di intermediari, l'opportunità (seppure per ora molto remota) di affrancarsi dai sistemi monetari basati sulla moneta stampata attraverso l'adozione delle nuove cryptocurrencies, l'utopia che le tasse versate da ogni contribuente possano essere utilizzate dai governanti in modo trasparente e tracciabile.
Gli addetti ai lavori sono colpiti dalle novità introdotte sotto il profilo del consenso distribuito, da come queste si amalgamano elegantemente con tecniche crittografiche per la sicurezza dei dati che devono circolare sulla rete e dalle possibili porte che blockchain può aprire nell'ambito del developing di nuove applicazioni.
Ma è tutto oro quel che luccica?
Questa tesi non si impone né l'onere né l'onore di rispondere a questa domanda dando giudizi su un paradigma che, checché se ne dica, ha ancora tanto da dimostrare. Piuttosto, questa tesi viene scritta con l'obiettivo di simulare efficientemente ed efficacemente una plausibile blockchain, cucita su misura addosso a un caso aziendale che verrà introdotto più nel dettaglio nel corso della trattazione. L'approccio utilizzato non sarà ovviamente industriale o guidato da tool blockchain-based, ma cercherà di rispettare al massimo il dogma di decentralizzazione predicato da blockchain.
Nel dettaglio, l'elaborato si struttura come segue: il primo capitolo introduce il rivoluzionario paradigma blockchain, spiegando quali sono le sue caratteristiche principali e dando qualche cenno storiografico; il secondo capitolo serve come rassegna per il linguaggio di programmazione che verrà utilizzato per costruire la piccola blockchain obiettivo di questa tesi, Go; il terzo capitolo entra nel dettaglio della progettazione, esaminando in maniera certosina gli aspetti principali affrontati durante l'ideazione della blockchain; il quarto capitolo mostra l'implementazione definitiva della soluzione, fornendo snippet del codice commentati nel dettaglio;

infine, il quinto e ultimo capitolo lascia al lettore un esempio di esecuzione della simulazione, in cui si affrontano casi ordinari e eccezioni.

1.1 La blockchain

A distanza di quasi dieci anni dalla sua prima effettiva implementazione, la definizione di blockchain è ancora abbastanza liquida e dipendente dal contesto in cui la si vuole inserire. Spesso, per descriverla, viene usato il termine "nuovo": "nuovo concetto di Trust", "nuova Internet", "nuova piattaforma tecnologica", "nuova democrazia". Una definizione certamente più "context-free" di blockchain potrebbe essere questa:

Definizione 1.1. Una blockchain è un database totalmente distribuito e pubblico, implementato, come suggerisce il nome, attraverso una catena di blocchi (ovvero una lista di record, ognuno dei quali collegato al precedente) di dimensione sempre crescente.

Un blocco è formato da una sequenza di transazioni (la misura "atomica" per descrivere le informazioni che circolano sulla rete tramite blockchain), e al momento della sua creazione da parte di uno dei client viene inoltrato a tutti i nodi appartenenti al sistema, che adeguano il contenuto della "replica" del loro database applicando i cambiamenti descritti dalle transazioni secondo l'ordine in cui sono inserite nel blocco. I blocchi e le transazioni vengono validate in modo totalmente decentralizzato, ovvero su ogni nodo della blockchain: se un'informazione viene ritenuto invalida secondo le regole del client che si esegue per connettersi alla blockchain, questa non viene considerata e non viene inoltrata dal client ai peer conosciuti.

1.2 Blockchain nella storia

Il primo esempio in termini cronologici (e anche in termini di fama ottenuta) di implementazione pratica di blockchain è *Bitcoin*, esposta al mondo da Satoshi Nakamoto in concomitanza con la pubblicazione del whitepaper "Bitcoin: A Peer-to-Peer Electronic Cash System", che è ormai eretto a "manifesto" di blockchain. L'idea fondamentale alla base della proposta di Nakamoto (che ha visto ufficialmente la luce nel 2009) è quella della creazione di una nuova cryptocurrency da poter scambiare liberamente e in modo totalmente decentralizzato tramite l'utilizzo di blockchain, che viene impiegata per registrare le varie transazioni finanziarie nei blocchi che la costituiscono. L'innovazione che Nakamoto vuole introdurre nel mondo informatico pone in realtà le sue radici su concetti già precedentemente presentati in vetrine accademiche: il *proof-of-work*, in particolare una sua implementazione denominata *Hashcash* [1].
Sull'onda del successo riscosso da Bitcoin sono state sviluppate decine e decine di cryptocurrencies basate su blockchain, che hanno preso il nome di "altcoins". Alcune di queste nascono come *fork* di Bitcoin (come *Litecoin, Zcash* e *Dash*), mantenendone gran parte del codice sorgente e delle features originali ma cambiandone alcune funzionalità cardine (ad esempio modificando le politiche di sicurezza); altre

invece sono implementate partendo da zero: è il caso di *Ether*, la celebre cryptocurrency della blockchain *Ethereum*.
Attualmente, la rosa di applicazioni pratiche di blockchain sta cercando di espandersi verso ulteriori contesti di indirizzo politico-economico: vengono oggi proposte soluzioni per una gestione smart dell'attività di riscossione delle imposte e delle tasse, per gestire un'elezione politica, oppure per gestire l'emissione di moneta da parte delle banche centrali. Recentemente, il Kenya, giovane e fragile democrazia dell'Africa orientale, in cui più volte si sono denunciati clamorosi brogli elettorali, ha deciso di dotarsi di un sistema basato su blockchain per gestire la prossima tornata elettorale.
Alle elezioni politiche italiane del 4 marzo 2018, la lista denominata "10 volte meglio", candidata alla Camera dei Deputati in alcune regioni della penisola, ha presentato un programma elettorale in cui è manifesta l'intenzione di incentivare lo sviluppo di applicazioni basate su tecnologie nuove e potenzialmente interessanti, tra cui blockchain [2].

1.3 Caratteristiche principali

Blockchain fonda la propria essenza su alcune parole chiavi molto accattivanti per le menti delle persone a cui essa si propone, e che spesso risultano essere features cruciali quando si affronta la costruzione di architetture per grossi sistemi: sicurezza, trasparenza, decentralizzazione. Ma, di preciso, come riesce a fornire questi vantaggi?

1.3.1 Decentralizzazione

Come anticipato nelle pagine precedenti, la decentralizzazione è il cuore pulsante di qualsiasi blockchain. Dire che blockchain è un database distribuito, come fatto nella definizione precedente, non significa altro che affermare come blockchain non segua i normali protocolli client-server (in cui il database è memorizzato interamente sul server ed è accessibile totalmente o parzialmente dai client tramite query ad esso inoltrate). Difatti, per evitare di riprodurre un singolo point-of-failure, ogni "nodo" che decide di entrare a far parte della rete che si affida su una blockchain scarica sul proprio client una copia del database, mantenendolo (grazie all'ausilio di particolari software che interrogano altri peer o ricevono comunicazioni dagli altri nodi) costantemente up-to-date tramite il download di nuovi blocchi. Il database è quindi accessibile in lettura con relativa semplicità, una volta scaricato. Non è altrettanto vero il concetto per quanto riguarda le operazioni in scrittura: non esistendo più, in questa particolare architettura, un elemento terzo centralizzato in grado di verificare, ordinare e regolare le varie richieste di modifica al database, blockchain deve fare uso di *protocolli di consenso distribuito* per assolvere a questa necessità. Questi protocolli di consenso distribuito prevedono che, per ottenere l'autorizzazione a pubblicare un nuovo blocco (e quindi ad aggiornare il contenuto di ogni replica del database), un nodo debba risolvere particolari puzzle crittografici, solitamente molto complicati e risolvibili solamente attraverso un dispendioso approccio brute-force. Chi riesce a risolvere il puzzle crittografico

può pubblicare un blocco, e viene generalmente ricompensato del lavoro svolto con una "reward", sotto forma di moneta virtuale. In gergo tecnico, questo processo di risoluzione del puzzle crittografico viene chiamato *mining*, e i nodi che prendono parte a questa procedura vengono di contro denominati *miners*.
La ragione per cui le modalità di accesso in scrittura sono così complicate è semplice: si vuole rendere sconveniente l'agire in modo fraudolento o in mala fede, operato pubblicando blocchi non validi con il solo intento di orientare i contenuti dei ledger in una certa direzione auspicata. Introducendo puzzle crittografici complicati, nessun nodo con cattive intenzioni spenderebbe mai una gran quantità del suo potere computazionale per creare un blocco che verrebbe subito dichiarato invalido (in quanto contenente transazioni fittizie) dagli altri peer, non dando così diritto alla ricompensa.

1.3.2 Trasparenza dei dati

Ogni copia del ledger, replicato sui nodi appartenenti alla rete, contiene tutte le transazioni, liberamente consultabili. Questo fa in modo che chiunque voglia informarsi sulla "storia" di un certo dato abbia pieno accesso alle sue origini e a tutte le evoluzioni subite fino al momento attuale. Inoltre, la possibilità di consultare ogni dato senza censure può conferire certezza che questo non è stato "corrotto" o modificato fraudolentemente.
Sotto questo punto di vista, la blockchain può essere vista come un "certificato di garanzia", in cui ogni transazione (incorruttibile e immodificabile) costituisce una verità assoluta e definitiva.
Non esiste limite al numero di ambienti che beneficerebbe della trasparenza che promette blockchain: i rapporti tra i cittadini e la Pubblica Amministrazione verrebbero sburocratizzati, per i consumatori sarebbe più facile informarsi circa la veridicità delle informazioni su un certo prodotto il cui processo produttivo è totalmente documentato sulla blockchain, per le aziende più facile gestire rapporti inter e intra-aziendali.

1.3.3 Sicurezza

Un primo modo per ottenere sicurezza, su blockchain, coinvolge accorgimenti classici: tramite l'utilizzo di crittografia e firma digitale, il mittente di ogni transazione è verificato (risolvendo quindi il problema di man-in-the-middle attack) e il contenuto di ognuna viene reso sicuro.
Oltre a metodi "da antologia", blockchain fornisce sicurezza dotandosi di una forte *immutabilità*, ottenuta come effetto del suo particolare design. Essendo infatti diversa da un sistema client/server, per corrompere una certa informazione contenuta nel database occorre modificare non solo una replica, ma bensì tutte le istanze della base di dati salvate su ogni nodo. Se si pensasse, per esempio, di corrompere una transazione contenuta nella blockchain Bitcoin, sarebbe necessario modificare con successo quasi diecimila database distribuiti in tutto il pianeta [3]. Come è facile immaginare, sarebbe un'operazione ostica, se non impossibile.
Un rischio concernente la sicurezza, seppure oggi ritenuto ancora molto teorico, che riguarda blockchain come Bitcoin viene chiamato *51% attack*, in cui si delinea

la possibilità che un gruppo di nodi che controlli la maggior parte dell'hashrate (ovvero della potenza di calcolo) di tutta la blockchain possa imporre coercitivamente sulle repliche dei ledger la propria visione, monopolizzando l'attività di creazione dei blocchi, bloccando a piacimento le transazioni di alcuni utenti (non includendole deliberatamente nei blocchi) e facendo *double spending*. Quest'ultima espressione, double spending, indica la possibilità (non auspicata) di pagare con la stessa moneta differenti persone: sarebbe come usare la stessa banconota per pagare due diversi creditori.
Il problema del double spending viene risolto dotando ogni moneta di un identificatore: in questo modo, ogni volta che un client tenta di trasferire la stessa moneta a due client differenti (che sia in buona o in mala fede), uno dei due scambi viene invalidato, in quanto si può verificare che una transazione contenente, come metodo di pagamento, una moneta con un dato identificatore è già stata emessa.

Capitolo 2

Il linguaggio Go

Introdotta in maniera sintetica ma essenziale blockchain, tramite l'illustrazione delle caratteristiche che le conferiscono appetibilità rispetto a un sistema totalmente centralizzato, il focus si sposta sullo strumento principale che consente a questo elaborato di simulare una blockchain: il linguaggio di programmazione Go.
Tramite una panoramica veloce ma auspicabilmente esaustiva, si daranno pillole introduttive su questo linguaggio, per poi entrare nel dettaglio dei formalismi da esso implementati che risulteranno utili in sede di produzione del progetto.

2.1 Breve panoramica

Go è un linguaggio di programmazione imperativo open source sviluppato da Google, annunciato nel 2009 ma la cui prima release è datata Marzo 2012. Dotato di una sintassi minimale e immediata a metà tra Python e C, si differenzia da linguaggi come PHP essendo un linguaggio compilato ad alte prestazioni. Nonostante ciò, non necessita di direttive di allocazione e deallocazione della memoria come le famose *malloc* e $free$ in quanto si dota autonomamente di un garbage collector che dinamicamente controlla le porzioni di memoria eventualmente da liberare.
A differenza di gran parte dei linguaggi imperativi "moderni", inoltre, non fornisce nessun supporto alla programmazione object-oriented: non esistono classi, non esiste ereditarietà (feature che rende il codice sia più veloce da eseguire sia più mantenibile), non esistono costruttori.

2.2 I tipi

Go è un linguaggio staticamente tipato [4]. Offre un insieme di tipi primitivi abbastanza ristretto:

- le *stringhe*: sequenze di caratteri con una lunghezza definita usate per rappresentare un testo;
- i *booleani*: usati per rappresentare i valori di verità *true* e *false*;
- i *numeri*, che possono essere ulteriormente divisi in due sotto-categorie:

  - gli *interi*, numeri senza la parte decimale, di dimensione variabile (*int8, int16, int32, int64*), con o senza segno (per ogni *int* illustrato prima esiste la sua controparte *uint*, ovvero *unsigned*);
  - i *numeri a virgola mobile*, anche detti *floating point*, ovvero i numeri reali.

2.3 Le strutture dati

Go offre anche un set di strutture built-in per agglomerare più dati tra di loro:

- gli *array*, una sequenza di elementi omogenei di lunghezza prefissata e immutabile il cui indexing comincia da 0 anziché 1;
- gli *slices*, omologhi agli array, ma che offrono la possibilità di variare la lunghezza della sequenza a run-time;
- le *maps*, un insieme di coppie chiave/valore;
- le *struct*, attraverso le quali si possono costruire tipi di dato compositi partendo dai tipi o dalle strutture dati finora citati.

2.4 Il controllo di sequenza

I formalismi offerti da Go per il controllo di sequenza sono semplicemente tre:

- il costrutto *if-else* che serve a imporre un blocco condizionale, in cui si valuta l'espressione dopo l'*if* per decidere quale ramo eseguire:

```
1 ...
2 var x int = 0
3 if x > 0{
4     x = 1
5 } else {
6     x = 2
7 }
8 ...
```

- il costrutto *switch-case*, che serve per costruire una serie di blocchi (ognuno individuato da un *case*). L'unico blocco che verrà eseguito tra quelli definiti sarà quello per cui l'etichetta accanto al *case* fa match col valore della variabile posta accanto alla keyword *switch*:

```
1 ...
2 var x int = 1
3 switch x{
4     case 1:
5         x++
6     case 2:
7         x = 1
8     case 3:
9         x = 5
```

```
10      case 4:
11          x = 16
12  }
13  ...
```

- il costrutto *for*, che permette di eseguire una moltitudine di volte lo stesso blocco di istruzioni, finché la guardia che segue la clausola *for* mantiene un valore di verità uguale a *true*. Può essere usato in diverse forme:

```
1  ...
2  var x int = 0
3  for i := 0; i < 10; i++{
4      x += i
5  }
6  ...
```

```
1  ...
2  var x [3]int
3  var sum int = 0
4  x[0] = 0
5  x[1] = 1
6  x[2] = 2
7  for pos, value := range x{
8      sum = sum + pos*value
9  }
10 ...
```

```
1  ...
2  var x bool = true
3  var y int = 0
4  for x {
5      if y < 10{
6          y++
7      } else {
8          x = false
9      }
10 }
11 ...
```

2.5 Le Goroutine

La feature più evidente e utile fornita da Go riguarda la programmazione concorrente: in un mondo in cui l'ottenimento di una forte efficienza è fondamentale, i linguaggi di programmazione devono dotarsi di supporti per la concorrenza altamente maneggiabili e duttili. Gran parte dei linguaggi usati comunemente non sempre si rivelano all'altezza di questa richiesta, implementando meccanismi efficienti ma complessi oppure di facile intuizione ma ingolfati. La proposta di Go, da questo punto di vista, si chiama Goroutine.
Una Goroutine è una funzione che è possibile eseguire su un light-weight thread in modo concorrente rispetto ad altre funzioni. Ad ogni Goroutine viene associato uno stack di memoria di dimensione espandibile on-demand, e ognuna di esse può

essere mappata su più thread di sistema [5]. Le Goroutine condividono gli stessi indirizzi di memoria, lavorando quindi in regime di memoria condivisa.
Per invocare una Goroutine occorre aggiungere la clausola *go* prima di una qualsiasi funzione (a patto che questa non abbia valori di ritorno). Sia ad esempio data questa semplice funzione Go:

```
1 func increment(x int){
2     x = x+1
3 }
```

può essere invocata sotto forma di Goroutine scrivendo questa semplice linea di codice:

```
1 go increment(0)
```

oppure, una Goroutine può anche eseguire funzioni anonime, un-named:

```
1 go func increment(x int){
2     x = x+1
3 }(0)
```

Definito l'ambiente in cui eseguire porzioni di codice in modo concorrente, è possibile fare in modo che diverse Goroutine comunichino tra loro?
La risposta è ovviamente affermativa, e i formalismi che si occupano dello scambio di dati tra diverse routine prendono il nome di *channels*, ovvero "canali".

2.6 I canali

Sono tipi di dato che vengono rappresentati dalla parola *chan*, seguita dalla keyword che individua il tipo dei dati che devono "navigare" al loro interno. Prima di poter essere utilizzati, devono essere "creati":

```
1 var c chan string = make(chan string)
```

Le operazioni possibili sui canali sono due:

- *send*: l'invio di un'informazione lungo il canale, rappresentato sintatticamente in questo modo:

  ```
  1 c <- "Hello World!"
  ```

- *receive*: la lettura di un'informazione dal canale, così rappresentata:

  ```
  1 v := <-c
  ```

Normalmente, *send* e *receive* sono operazioni bloccanti: la prima si blocca fino a quando qualcuno non esegue una *receive* sul canale, mentre la seconda si blocca fino a quando qualcuno non esegue una *send* sul canale. Si può "arginare" limitatamente il blocco creando un canale *buffered*:

```
1 var c chan string = make(chan string, 100)
```

in questo caso, le operazioni di *send* si bloccano solamente quando il buffer del canale sarà pieno, mentre le *receive* solo quando il buffer sarà vuoto.
Si possono effettuare operazioni di *send* su un canale fino a quando questo è "aperto": in qualsiasi momento, infatti, si può chiudere un canale tramite la funzione *close*. Siccome effettuare *send* su canali chiusi causa situazioni di panic, l'attività di chiusura di un canale è sempre raccomandata al sender, in modo tale che sappia il tempismo giusto in cui agire.
Sebbene la funzione di *close* ricorda le system call Unix adibite alla chiusura dei file descriptor, i canali non necessariamente debbono essere chiusi, in quanto il garbage collector interviene di volta in volta liberando le porzioni di memoria non più utilizzate.
Per visualizzare le informazioni contenute in un canale, quando è chiuso, si può utilizzare un normale ciclo *for*:

```
1  ...
2  var c chan int = make(chan int, 100)
3
4  for i := 0; i < 10; i++{
5      c <- i
6  }
7
8  close(c)
9  var sum int = 0
10
11 for elem := range c{
12     sum += elem
13 }
14 ...
```

Capitolo 3

Un caso pratico: Tesium

Dopo aver introdotto gli elementi teorici necessari per trattare l'argomento, il focus passa dalla parte di rassegna a quella implementativa, con la progettazione e la simulazione di una blockchain attraverso l'utilizzo del linguaggio Go. Di seguito verrà costruito ad hoc un caso aziendale fittizio, in cui contestualizzare la soluzione che si vuole proporre. Questo capitolo si concentrerà nella descrizione ad alto livello delle decisioni prese in sede implementativa, lasciando al successivo i dettagli tecnici.

3.1 Tesium e la raccolta differenziata porta a porta

Il caso più o meno pratico che verrà preso come esempio per questa trattazione è quello di una fantomatica società denominata Tesium. Nell'immaginario di questo elaborato, Tesium verrà dipinta come un'azienda italiana a partecipazione pubblica (comunale) che si occupa della raccolta differenziata porta a porta dei rifiuti. Nonostante il monopolio oramai quinquennale esercitato sui borghi di sua pertinenza, Tesium sta affrontando diverse difficoltà economiche, contabilizzate nell'ultimo bilancio con una severa perdita d'esercizio. Il consiglio comunale, a fronte della situazione, decide di operare un "giro di vite" ai vertici della sfortunata società, incaricando un nuovo amministratore delegato di portare avanti azioni necessarie alla riduzione delle perdite.
Il nuovo manager si opera fin da subito ad ottemperare ai suoi obblighi, optando per una politica di "spending review", ovvero di tagli agli sprechi. Inizia prima con l'eliminare le spese da lui reputate inutili, per poi lanciarsi in un'ostica battaglia con l'obiettivo di rinegoziare i contratti con i fornitori. L'ultima sfida (in ordine cronologico) che si pone è quella di organizzare sotto una nuova luce le spese per il personale.

La sua idea è tanto rischiosa quanto innovativa e passa dalla tecnologia: motivato da alcuni suoi fidi consiglieri, propone di dotare l'azienda di un nuovo sistema informatico basato su blockchain, col fine di adottare una nuova politica salariale basata su premi che, a lungo termine, possa dare luogo a maggiori risparmi. La blockchain, in questo caso, verrebbe utilizzata come grande database distribuito

in cui raccogliere le informazioni riguardanti i cassonetti svuotati durante una sessione di raccolta.
Il concetto è il seguente: ogni furgone viene reso un "peer" della blockchain (perciò è dotato di un sistema informatico embedded su cui viene scaricata copia del ledger), con cui può interagire tramite la dotazione di un lettore di codici a barre. Su ogni cassonetto della raccolta differenziata viene apposto quindi un codice a barre: ad una scansione di un codice da parte del lettore corrisponde la pubblicazione sulla blockchain di una nuova transazione (tramite meccanismi di IOT già oggi in progettazione [6]), in cui viene specificato il codice a barre letto e il camion che ha operato alla lettura. Le transazioni vengono raccolte in blocchi, i cui "miner" sono di volta in volta scelti tramite l'esecuzione di un protocollo di consenso distribuito progettato per premiare il merito. Al camion-peer delegato alla pubblicazione di un blocco viene conferita una reward, accreditata in un "wallet" legato al furgone. Viene quindi a delinearsi l'occasione che consentirebbe al manager di proporre un intervento di revisione della componente fissa di ogni salario, per lasciare spazio ad una componente variabile fatta di premi generati dal nuovo sistema, che occorrono anche come meccanismo di competizione e stimolo.

3.2 L'idea generale

Dopo aver descritto la realtà in cui si vuole immedesimare la progettazione della blockchain che si vuole andare a implementare, si indirizzerà la trattazione in una direzione esplicativa che parta dalle strutture dati di base, per poi esaminare i meccanismi con cui queste dialogano.
In ogni blockchain, gli agglomerati informativi di granularità minima sono principalmente due: le transazioni che devono viaggiare sulla rete, e i blocchi che devono raggrupparle.

3.2.1 Le transazioni

Contrariamente a gran parte delle blockchain che oggi dominano il loro settore di mercato, questa non conterrà transazioni che regoleranno pagamenti in denaro (o, per meglio dire, in cripto valute). Bensì, il contenuto di ogni transaction certifica la lettura del codice a barre di un cassonetto da parte del lettore (quindi da parte del camion a cui questo è collegato).
Perciò ogni transazione può denotarsi facilmente tramite tre campi principali:

- **l'identificatore del bidone dei rifiuti** scansionato: un attributo univoco collegato al codice a barre di ogni bidone della spazzatura;

- **l'address del client** montato sul sistema embedded del camioncino, il cui lettore ha effettuato la scansione: un attributo univoco tramite cui ogni nodo viene identificato sulla blockchain;

- **un timestamp**, ovvero una marca temporale che identifichi la data e l'ora della scansione.

3.2.2 I blocchi

Se, normalmente, i blocchi sono strutture dati addette all'agglomerazione di larghi quantitativi di transazioni, in questa blockchain assumono una doppia funzionalità:

- una funzione di aggiornamento della catena, assolta facendo da "magazzino" alle rilevazioni dei lettori effettuate di volta in volta dai dipendenti di Tesium;
- una funzione di salvaguardia e validazione della catena, ottemperata raccogliendo i dati necessari all'esecuzione del protocollo di consenso distribuito che verrà successivamente descritto.

Perciò, un blocco per la blockchain Tesium (prende lo stesso nome dell'azienda per cui verrà modellata) sarà così strutturato:

- **l'epoca**: campo di cui si darà spiegazione dettagliata in sede di descrizione del protocollo di consenso distribuito che andrà implementato per Tesium;
- **il sender**, o publisher, del blocco: campo che conterrà l'indirizzo di chi ha pubblicato il blocco sulla blockchain;
- **l'altezza** del blocco: un intero che definisce la distanza di un blocco dal primo blocco nella blockchain, situato ad altezza zero;
- **l'hash del blocco precedente**, così da poter mostrare un'effettiva correlazione tra ogni blocco della blockchain;
- **un array di transazioni**;
- **un array di "messaggi di elezione"**, strutture dati che serviranno a convergere verso l'unanime decisione su chi avrà diritto di pubblicare un certo blocco in un periodo di tempo futuro, come verrà spiegato tra poche righe.

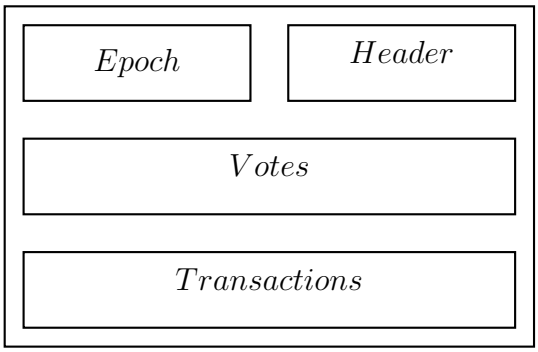

Figura 3.1: Struttura base del blocco.

3.3 I nodi della blockchain

Ogni nodo facente parte della blockchain viene dotato della sua "istanza" del ledger e di un wallet, su cui salvare eventuali remunerazioni garantite durante le ore lavorative. Di volta in volta, un nodo pubblicherà transazioni per ogni cassonetto

dei rifiuti il cui codice a barre viene scansionato dal lettore corrispondente.
Ogni nodo che vuole collegarsi alla blockchain, nel caso inferisca di non essere l'unico al momento collegato sulla rete, interroga gli altri nodi per conoscere la storia pregressa della blockchain, in modo tale da poter aggiornarsi e poter lavorare partendo dalla stessa base conoscitiva degli altri peer. Un nodo che riceve una richiesta di "catch-up" da un altro appena inseritosi nella rete invia a quest'ultimo i dati al momento in suo possesso, facendo uno "snapshot" della situazione attuale del ledger e di altre variabili di ausilio.
Ogni nodo partecipa attivamente al processo di scelta del peer che dovrà forgiare il blocco successivo eseguendo una procedura di "elezione".

3.4 L'algoritmo di leader election

Il punto nevralgico della progettazione riguarda la modellazione del protocollo di consenso distribuito, in cui si decide chi è autorizzato a pubblicare uno o più blocchi sulla blockchain e le reward che ad esso spettano. Così spiegato, il sotto-problema di trovare, tra un folto numero di nodi, uno tra loro che si erga quasi a "leader" per portare a termine un determinato compito sembra essere l'istanza di un problema di *leader election*, ovvero una classe di problemi che si pone l'obiettivo di nominare un leader tra una serie di nodi, il cui compito diventa quello di organizzare l'esecuzione di vari task distribuiti sui peer del sistema in modo tale da evitare conflitti in caso, per esempio, di eventuali accessi concorrenti a risorse.
Questa analisi trova fondamento: nelle blockchain *proof-of-work based* (ovvero quelle che concedono l'autorizzazione a pubblicare un blocco solamente a fronte della risoluzione di un complesso puzzle crittografico), per esempio, l'algoritmo di leader election è appunto quello che prevede di trovare, tramite brute-force, la soluzione a un puzzle. L'algoritmo di leader election scelto per essere incastonato nel cuore del protocollo di consenso distribuito di Tesium altri non è che una modifica del già noto Bully Election Algorithm.

3.4.1 Il Bully Election Algorithm

Il Bully Election Algorithm è tra i più famosi algoritmi di elezione nei sistemi distribuiti. Lavora sotto le assunzioni che il sistema sia sincrono, che il sistema di scambio dei messaggi sia affidabile (ovvero che nessun messaggio vada perso o venga consegnato a un destinatario sbagliato) e che ogni nodo possa incorrere in un crash in un qualsiasi momento (e che possa essere riavviato ricominciando l'esecuzione del protocollo), ma che non possa "stopparsi" per un tempo indefinito ripartendo dallo stesso "punto di vista" in cui si era bloccato. Ogni nodo che non è in una situazione di crash deve rispondere senza alcun delay a ogni messaggio. Ciascun "agente" ha un identificatore univoco che lo distingue dagli altri, e conosce tutti gli id degli altri nodi nella rete [7].
L'algoritmo viene eseguito quando un nodo N capta un crash del leader (che viene rilevato se il leader non ha informato la rete di essere vivo entro una certa finestra temporale e non ha risposto a un sollecito di un nodo). La sequenza di operazioni segue questo schema:

1. Se N ha l'id maggiore rispetto a tutti gli altri nodi, invia direttamente un messaggio di vittoria a ogni altro nodo e diventa il leader. In caso contrario, invia a ogni nodo con id maggiore al suo un messaggio in cui annuncia l'inizio dell'elezione.

2. Ogni nodo che riceve un messaggio che annuncia l'inizio di un'elezione da parte di N, manda un messaggio di stop a N e inizia a sua volta un processo di elezione inviando a ogni nodo con id maggiore al proprio un messaggio.

3. Se N non riceve messaggi di stop entro un certo timeout, invia un messaggio di vittoria a ogni altro nodo e diventa il leader. In caso contrario, si mette in attesa di un messaggio di vittoria da parte di un nodo con id maggiore. Se non riceve nessun messaggio di vittoria entro un certo timeout, ricomincia la procedura di elezione.

4. Se un nodo riceve un messaggio di vittoria da un altro nodo, imposta quest'ultimo come suo leader.

Esempio 3.1. Sia dato l'esempio di un sistema costituito da 5 processi (con gli id degli agenti che coprono il range [1, 5]). Sia dato come leader il processo con id uguale a 5, si supponga che tale leader subisca un crash e che il processo con id uguale a 3 sia il primo a rilevare il failure. Il processo 3 manda quindi un messaggio di elezione ai processi 4 e 5, mettendosi in attesa. Il processo 4 riceve il messaggio di elezione del processo 3 e gli risponde con un messaggio di stop. Il processo 5 non ha ancora effettuato crash-recovery quindi non risponde. A questo punto il processo 4 inizia a sua volta un processo di elezione e invia a 5 un messaggio, a cui però non fa seguito alcuna risposta. Scaduto il timeout, 4 si "incorona" leader e informa tutti gli altri processi con un messaggio di vittoria.

Sia n il numero di agenti del sistema, e sia supposto che un solo nodo rilevi il crash del leader. La complessità dell'algoritmo, espressa nel numero dei messaggi scambiati, è:

- nel caso pessimo (ovvero nel caso in cui è il processo con id minimo ad accorgersi del crash del leader): $O(n^2)$;
- nel caso ottimo (ovvero nel caso in cui il processo che inizia l'elezione è quello che diverrà poi il leader): $O(n)$.

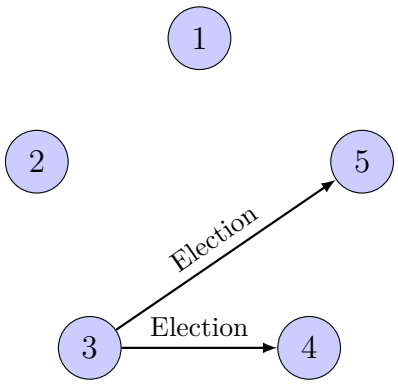

Figura 3.2: 3 inizia l'elezione.

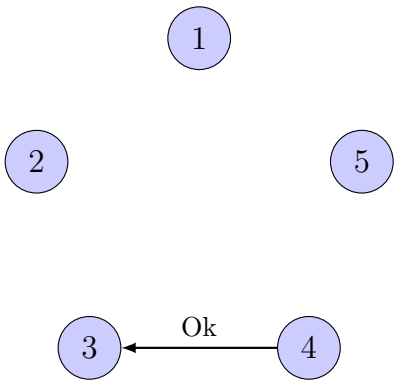

Figura 3.3: 4 risponde a 3.

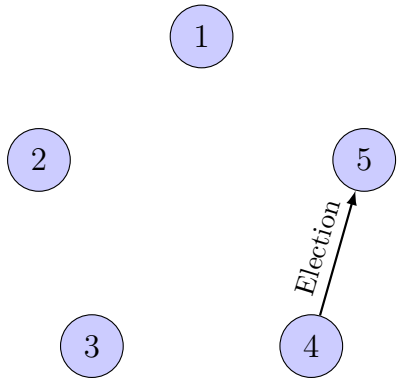

Figura 3.4: 4 inizia l'elezione.

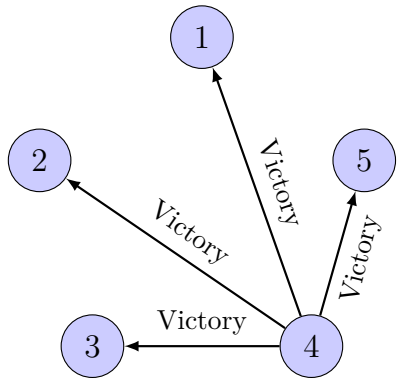

Figura 3.5: 4 si dichiara leader.

Come precedentemente enunciato, questa è una delle soluzioni più conosciute nell'ambito dei problemi di leader election, e sulla base di questo algoritmo sono state costruite numerose varianti che cercano di migliorarlo, riducendone ad esempio la complessità in termini di messaggi da scambiare o aumentandone la fault tolerance. Una di queste, il *Modified Bully Algorithm*, introduce alcuni cambiamenti interessanti, e sarà l'implementazione del Bully Algorithm che verrà utilizzata come base per la creazione dell'algoritmo di leader election progettato per Tesium.

3.4.2 Il Modified Bully Algorithm

Il Modified Bully Algorithm nasce con lo specifico intento di ridurre al minimo lo scambio di messaggi tra i vari nodi richiesto dall'algoritmo originale per determinare il leader. Come prima evidenziato, infatti, la complessità espressa nel numero dei messaggi per il Bully Election Algorithm è nell'ordine di $O(n^2)$ con n pari al numero di agenti appartenenti al sistema.
Questa nuova soluzione conserva tutte le assunzioni che sono fatte per l'originale: la sincronia del sistema, l'affidabilità del canale di comunicazione di cui questo si serve e il fatto che ogni nodo goda di un univoco identificatore. Supponendo sia il nodo N quello che per primo si accorge del crash del leader, la sequenza di operazioni da eseguire per l'elezione si imposta secondo il seguente modus-operandi [8]:

1. Se il nodo N ha l'id maggiore rispetto a tutti gli altri nodi, invia direttamente un messaggio di vittoria a ogni altro nodo e diventa il leader. In caso contrario, invia a ogni nodo con id maggiore al suo un messaggio in cui annuncia l'inizio dell'elezione.

2. Ogni nodo che ha ricevuto da N un messaggio di elezione risponde ad N, allegando al messaggio il proprio identificativo univoco.

3. Se N non riceve messaggi dai destinatari dei suoi precedenti messaggi di elezione, dichiara se stesso come leader agli altri nodi coinvolti. In caso contrario, il nodo N sceglierà il nodo a cui corrisponde l'id maggiore, a cui invierà un messaggio di "grant", ovvero un messaggio che lo autorizzerà a proclamarsi leader al resto della rete.

4. Il nodo che ha ricevuto il messaggio di "grant" da N si dichiara al sistema come nuovo leader.

L'algoritmo si pone inoltre l'obiettivo di risolvere un ulteriore problema del Bully Election Algorithm: nella soluzione originale, infatti, più nodi potevano eseguire parallelamente l'elezione, generando un'immane quantità di traffico sulla rete. Il problema, in questo caso, viene risolto tramite questi accorgimenti:

- Due nodi M e N che si accorgono simultaneamente del crash del leader iniziano in maniera separata la procedura di elezione.

- Se un processo riceve più messaggi di elezione da diversi nodi con id minori rispetto al suo, risponde solamente al nodo con id minimo tra quelli che lo hanno "contattato".

- Se un nodo M sta tenendo un processo di elezione e viene ricevuto da un messaggio di elezione da parte di un nodo N, in cui l'id di N è minore dell'id di M, allora M termina la sua procedura di elezione.

Esempio 3.2. Si prenda nuovamente ad esempio il caso precedente: 5 processi (i cui id sono distribuiti sul range [1,5]), in cui il processo con id uguale a 5 è il current leader. Ancora, il leader subisce un crash e il nodo identificato dal numero 3 è il primo ad accorgersi del guasto, e quindi ad intraprendere la procedura di elezione.
Come prima, 3 manda i messaggi di elezione ai nodi con id maggiori rispetto al suo, ovvero i nodi 4 e 5. 4 risponde a 3 allegandogli il suo identificatore, mentre 5 (che non si è ancora ripreso dal crash) non risponde. Avendo ricevuto una risposta, 3 osserva (in questo caso abbastanza banalmente) che 4 è il nodo con id maggiore dal quale ha ricevuto un "ack". A questo punto, incarica 4 di assumersi le responsabilità di leader mandandogli un messaggio di "grant". Infine, 4 invierà a tutti i nodi della rete un messaggio in cui annuncia la sua vittoria.

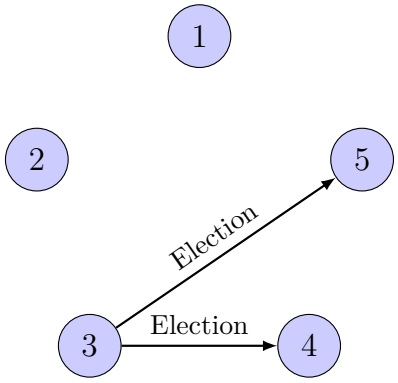

Figura 3.6: 3 inizia l'elezione.

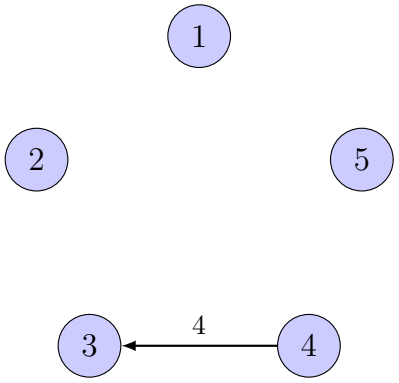

Figura 3.7: 4 risponde a 3.

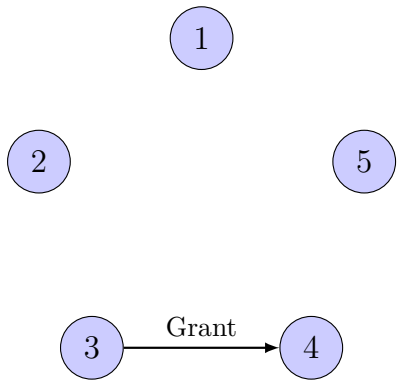

Figura 3.8: 3 incorona 4.

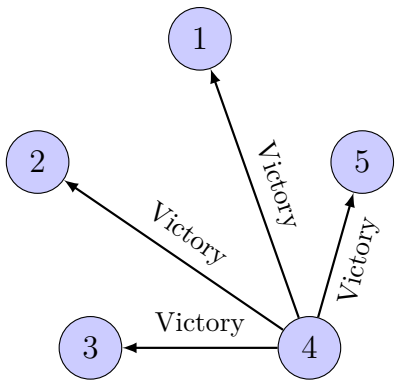

Figura 3.9: 4 si dichiara leader.

Se si passa all'analisi della complessità nel numero dei messaggi, il vantaggio introdotto da questa particolare implementazione è evidente. Supponiamo che un solo nodo si accorga del crash e si faccia di nuovo la distinzione tra i due casi esaminati anche in precedenza:

- caso pessimo (il nodo con id minore si accorge del crash): la complessità è un $O(n)$, che migliora la complessità di $O(n^2)$ della precedente soluzione;
- caso ottimo (il nodo che diventerà leader si accorge del crash): la complessità rimane nell'ordine di $O(n)$, come per la precedente implementazione.

3.4.3 Implementazione per Tesium

Sulla base di questi due algoritmi di leader election, ma soprattutto sulla base dell'ultimo esaminato, verrà costruito l'algoritmo di leader election che verrà scelto per fare parte del protocollo di consenso distribuito di Tesium. Ovviamente, per meglio comprendere l'algoritmo che si andrà a proporre, bisogna premettere alcune differenze tra le implementazioni sopra citate e la soluzione che verrà presto esposta:

- se il Bully Election Algorithm e il suo omologo modificato vengono eseguiti solamente nel caso in cui il leader precedente subisca un crash, l'algoritmo la cui discussione è prossima dovrà essere eseguito in loop all'infinito, in modo tale da garantire l'esistenza di nodi che aggiornino la blockchain continuamente;
- se il Bully Election Algorithm è progettato per essere eseguito in sistemi sincroni, l'implementazione qui fornita deve dialogare con blockchain, che è un sistema totalmente asincrono. A tale scopo, quindi, nel protocollo di consenso distribuito in cui verrà amalgamato questo algoritmo andranno inseriti opportuni parametri di sincronia per regolare lo svolgimento di ogni fase dell'algoritmo;
- se nelle implementazioni prese dalla letteratura lo scambio dei messaggi viene gestito tramite un non meglio specificato meccanismo di message passing diretto (ovvero in cui il sender manda il messaggio solo al receiver), in questo ambiente i messaggi vengono supposti come trasferiti secondo le regole della comunicazione blockchain (quindi inviati a tutti i nodi, che li devono validare)

e, come spiegato in sede di descrizione della struttura dei blocchi di Tesium, devono essere registrati sui blocchi, in modo tale che ogni nodo possa basare ogni fase dell'algoritmo sullo stesso set di messaggi disponibile anche agli altri.

Infine, a differenza dell'implementazione originale, questo algoritmo si pone ancora l'obiettivo di eleggere un unico leader, ma di nominare anche una lista di "sostituti", nel caso in cui questo non pubblichi il blocco (o i blocchi) per cui viene nominato. Rimanendo valido l'assegnamento di identificatori univoci a ogni nodo che fa parte del sistema, è possibile delineare l'algoritmo di leader election che verrà utilizzato suddividendolo in quattro fasi principali:

1. la fase di *election*: in questa fase, tutti i nodi che partecipano all'elezione (quindi, in genere, tutti i nodi che sono attivi al momento dell'inizio della suddetta) inviano i messaggi di elezione impostando come destinatari i nodi con identificatori maggiori rispetto ai loro. Nel caso in cui un nodo non trovi nessun peer con identificatore maggiore al suo, non si dichiara immediatamente come leader come nei precedenti casi, bensì si mette in attesa della seconda fase;

2. la fase di *response*: in questa fase, ogni nodo che è destinatario di un messaggio risponde al mittente con identificatore minimo tra quelli che lo hanno "interpellato" costruendo un messaggio che imposta quest'ultimo come destinatario;

3. la fase di *grant*: fase che può evolversi in due possibili sotto-casi:

   - il mittente con il minimo identificatore tra quelli che hanno spedito il messaggio di elezione (la cui identità, si preme di ricordare, è uguale per ogni nodo, in quanto i messaggi di elezione vengono raccolti nei blocchi e ogni nodo può ricavarla in modo totalmente distribuito) non riceve alcuna risposta: invia un unico messaggio impostando se stesso come destinatario, in cui si autorizza a proclamarsi leader di fronte ai nodi della blockchain durante la fase successiva;
   - il mittente con la proprietà appena descritta è destinatario di uno o più messaggi di risposta: estrapola i mittenti di ogni messaggio, raccogliendoli in una lista, che procede ad ordinare in modo decrescente secondo l'identificatore di ogni nodo. A questo punto crea, per ognuno di quei mittenti, un messaggio in cui allega la loro "posizione" occupata nella "classifica" dei leader. Dopodiché, crea un ultimo messaggio di grant, in cui imposta se stesso a destinatario e si attesta in ultima posizione. Il nodo in prima posizione sarà quello che diverrà il leader, il secondo diverrà il sostituto del leader, e così via.

4. la fase di *leader*: l'ultima fase dell'algoritmo, in cui chi è destinatario di un messaggio di grant invia un messaggio in cui accetta sostanzialmente la posizione nella lista che gli è stata assegnata.

3.5 I messaggi di elezione

Ora che si è definito il meccanismo di elezione, si può dare la semplice descrizione della struttura di ogni messaggio. Ognuno di essi sarà caratterizzato da:

- **il tipo di messaggio** inviato, ognuno identificante una fase dell'algoritmo: "election", "response", "grant", "leader";
- **l'identificatore del mittente** del messaggio, da non confondersi con l'indirizzo, che invece serve a denotare un nodo all'interno della blockchain: l'identificatore, potenzialmente, può cambiare ogni volta che l'algoritmo di leader election viene eseguito, pur rimanendo univoco. Saranno dati maggiori dettagli in sede di discussione del protocollo di consenso distribuito;
- **l'identificatore del destinatario**, per cui valgono le stesse disposizioni date per il punto precedente;
- **un payload**, che serve a specificare la posizione di un nodo nella lista dei leader in alcuni tipi di messaggio.

3.6 Il protocollo di consenso distribuito

In questo paragrafo, si spiegherà come amalgamare l'algoritmo di leader election prima ampiamente spiegato nel protocollo di consenso distribuito della blockchain Tesium.
In sintonia con le premesse che erano state fatte nel momento della formalizzazione del Bully Algorithm usato per l'evenienza, occorre introdurre dei parametri di sincronia, attraverso i quali "obbligare" i nodi a sbrigare la procedura di elezione in un tempo limitato e a scadenze ripetute.
A questa necessità si cerca di porre soddisfazione tramite la seguente modellazione, prendendo esempio dalla blockchain Cardano e dal suo algoritmo di consenso distribuito Ouroboros [9] [10].

Si immagini di frazionare lo scorrere del tempo in slot, ognuno lungo un lasso di tempo x. Ogni slot così identificato viene chiamato "epoca", e per ogni "epoca" esiste un solo ed unico leader, e una lista di eventuali sostituti. Durante l'epoca N si terranno tutte le procedure necessarie per l'elezione del leader e dei sostituti dell'epoca $N + 1$. Il compito di ogni leader durante l'epoca per cui viene eletto è quello di pubblicare sulla blockchain quattro blocchi (uno per ogni fase del Bully Algorithm prima implementato), in cui raccogliere mano mano i messaggi di elezione (ogni blocco può contenere un solo tipo di messaggi di elezione) e le transazioni che nel frattempo sono state registrate dai vari nodi. Ogni epoca è internamente divisa in due ulteriori slot temporali: uno in cui effettuare l'elezione, e uno in cui accettare nuovi nodi che vogliono connettersi alla blockchain. Lo schema usato per la modellazione di ogni epoca è il seguente:

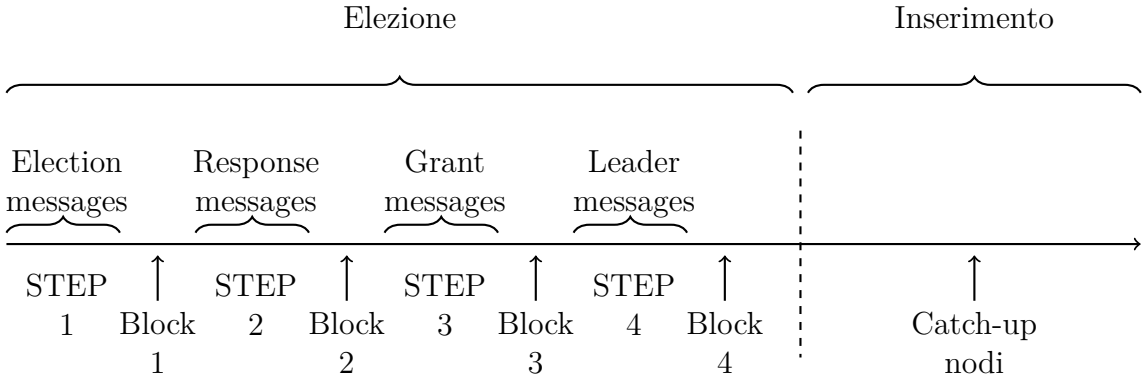

Figura 3.10: Rappresentazione schematica di un'epoca.

La prima parte dell'epoca è certamente quella che merita una spiegazione più approfondita: ogni fase del Bully Algorithm viene intervallata dalla pubblicazione di un blocco contenente tutti i messaggi di elezione raccolti dal leader incaricato per quell'epoca entro un certo lasso di tempo (il precedentemente e abbondantemente citato parametro di sincronia). Alla (eventuale) pubblicazione del quarto blocco di ogni epoca, nel caso in cui l'elezione abbia avuto esito positivo, si aggiornerà il nome del leader dell'epoca successiva e la lista di eventuali sostituti; in caso di esito negativo, invece, si possono adottare due possibili contromisure:

- nel caso in cui il fallimento dell'operazione di elezione non dipenda da chi pubblica i blocchi (i.e.: il nodo delegato a mandare i messaggi di grant subisce un crash, non inviando niente): vengono mantenuti per l'epoca successiva lo stesso leader e la stessa lista dell'epoca corrente;

- nel caso in cui il fallimento dell'operazione di elezione dipenda da chi pubblica i blocchi (i.e.: il delegato a pubblicare il blocco in cui è contenuto il nuovo leader e la lista dei suoi sostituti subisce un crash, non pubblicando il blocco): il leader viene rimpiazzato con il primo nodo inserito nella lista dei sostituti valida per l'epoca corrente, che rimarrà valida anche per la successiva.

La seconda parte dell'epoca invece è adibita all'inserimento di nuovi nodi che desiderano entrare nel sistema: la fase di inserimento (o di "catch-up") prevede l'interrogazione di altri nodi, alla quale questi rispondono inviando al nodo interrogante l'attuale stato della blockchain (ovvero i blocchi che la compongono, l'epoca corrente, il leader corrente e la lista dei sostituti, il saldo attuale di ogni nodo). Ogni nodo che proverà ad inserirsi durante la prima fase dell'epoca rileverà la necessità di aspettare la fine dell'elezione, per poi procedere nell'effettuare la procedura di catch-up. Inserire un nuovo nodo durante la procedura di elezione può risultare in incongruenze forti nei ledger distribuiti.

Esempio 3.3. Durante la procedura di elezione, tra il primo e il secondo blocco, un nodo tenta di connettersi alla blockchain. Inizia quindi la procedura di catch-up, interrogando un nodo per sapere lo stato attuale della blockchain. Appena il nodo interrogato termina la sua risposta, viene pubblicato il secondo blocco dell'elezione. Questo non viene inviato dal nodo interrogato (in quanto ha già terminato la sua fase di risposta), e non viene neanche intercettato dal nodo appena collegatosi

(in quanto inizia ad "ascoltare" nuovi blocchi in entrata solamente terminata la procedura di catch-up, che sta ancora terminando, in quanto ancora impegnato a inserire i blocchi inviatigli nella sua copia della catena). Così, ogni nodo della blockchain avrà N nodi, tranne quello appena collegatosi alla rete, che ne avrà $N-1$ e non potrà validare nessun blocco che verrà aggiunto, in quanto i link ai blocchi precedenti rappresentati dagli hash verranno sempre ritenuti invalidi.

Ad ogni blocco che il leader pubblica con successo sulla blockchain, ogni nodo aggiorna la sua copia del saldo contabile, aggiungendo un token nel "wallet" del leader. In questo modo, in base agli addetti che hanno lavorato su un certo camioncino-peer e alle ricompense per ogni volta che quel camioncino è stato "leader", verranno calcolati i premi-produzione individuali.

Ma come, effettivamente, si decide chi è il leader nella blockchain Tesium? Il processo di elezione tramite Bully Algorithm modificato viene ripetuto in loop nel tempo, e necessita, per essere portato a termine, di identificatori univoci per ogni agente che ne prende parte. Occorre trovare, quindi, per ogni processo di elezione un identificatore sì univoco, ma che potenzialmente cambi ad ogni elezione, in modo tale da non avere una situazione in cui un solo nodo è perennemente in stato di leadership e monopolizza l'attività di creazione dei blocchi.
É quindi da scartare l'idea di poter utilizzare l'address di ogni nodo come identificatore, in quanto attributo statico assegnato al momento dell'accensione del sistema sul camioncino. La necessità ricade sulla ricerca di un attributo che sia calcolato dinamicamente, in maniera distribuita dai nodi ma anche in modo tale che il suo metodo di calcolo non possa lasciare adito ad ambiguità.
L'idea implementata a soluzione di questo problema è la seguente: al sorgere dell'epoca N, ogni nodo consulta la blockchain alla ricerca dei blocchi pubblicati nell'epoca $N-1$. Una volta trovata questa "sottocatena", per ogni blocco controlla tutte le transazioni, costruendo una struttura dati di queste fattezze:

```
{
  "address1": numberOfTransactions,
  "address2": numberOfTransactions,
  "address3": numberOfTransactions,
  ...
}
```

che altri non è che un dizionario in cui, per ogni address, viene calcolato il numero di transazioni che questo ha effettuato nell'epoca $N-1$, quindi il numero di cassonetti che ha scansionato con il lettore associato al camioncino dei rifiuti. Grazie a questa struttura dati, ciascun peer è in grado di costruire gli identificatori univoci per ogni nodo della blockchain tramite questa concatenazione:

```
{
  "address1": "numberOfTransactions.ADDRESS.address1",
  "address2": "numberOfTransactions.ADDRESS.address2",
  "address3": "numberOfTransactions.ADDRESS.address3",
}
```

Gli identificatori, ovviamente, rimangono univoci, in quanto in ogni blockchain ogni address è già a sua volta univoco. In questo modo, salvo casi di crash indesiderati, ad ogni epoca viene eletto il nodo che nella precedente epoca è stato il

più produttivo, incentivando e soprattutto premiando la competizione, in quanto il primo parametro di comparazione è costituito dal numero di transazioni pubblicate da ogni nodo.

Esempio 3.4. Si supponga di avere il seguente insieme di indirizzi, ognuno dei quali associato al numero di transazioni eseguite durante una certa epoca N:

```
{
  "ac3478d69a3c81fa": 18,
  "62e60f5c3696165a": 6,
  "4e5e6ac486dsg36b": 12
}
```

Nell'epoca $N + 1$, gli identificatori che verranno usati in sede di elezione saranno, quindi:

```
{
  "ac3478d69a3c81fa": "18.ADDRESS.ac3478d69a3c81fa",
  "62e60f5c3696165a": "6.ADDRESS.62e60f5c3696165a",
  "4e5e6ac486dsg36b": "12.ADDRESS.4e5e6ac486dsg36b"
}
```

Capitolo 4

L'implementazione

Il capitolo precedente ha esplicato i dettagli progettuali circa le informazioni che viaggiano sulla blockchain Tesium, come operano i nodi che fanno circolare le informazioni e come questi dialogano tra loro. Questo capitolo mette nero su bianco le scelte implementative, in assoluta fedeltà rispetto alla progettazione, dando istanze a valori che nel precedente capitolo erano rimasti aleatori e fornendo frammenti di codice che permettano di meglio comprendere le parti fondamentali della simulazione.

4.1 La comunicazione tra i nodi

Il primo passo per descrivere questa blockchain è quello di farlo tramite la trattazione dell'architettura di base che andrà costruita per fare dialogare i nodi.
Essendo questa una simulazione, il processo di networking non è gestito tramite un effettivo protocollo peer-to-peer, ma simulato.
L'idea principale è di creare un server TCP tramite Go, a cui tutti i nodi possono connettersi per registrarsi nella blockchain ed entrare a fare parte della simulazione. Il termine "server" tradisce il sentimento di blockchain, ma in questo caso non viene usato per salvare variabili (che verranno salvate in ogni nodo) o per prendere decisioni a livello centralizzato (anch'esse operate a livello distribuito), ma piuttosto per creare una piattaforma su cui ogni nodo può connettersi e dialogare con gli altri nodi (o simulare il dialogo, con tecniche che a breve verranno spiegate). L'architettura dell'implementazione che si andrà a costruire può essere rappresentata con questa iconografica:

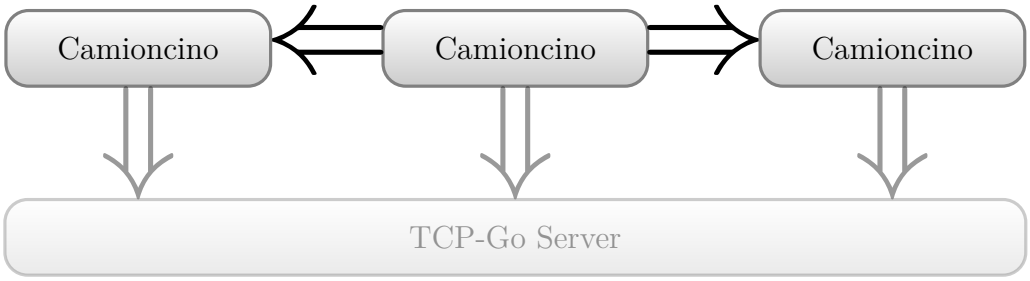

Figura 4.1: Rappresentazione grafica dell'architettura.

in cui le frecce tra un camioncino e l'altro indicano che la comunicazione "attiva" avviene solamente tra i nodi, mentre le frecce sfocate in direzione del server rappresentano il fatto che la simulazione dipende comunque da un organismo centralizzato che, seppur non operando in maniera attiva, fornisce le attività di networking necessarie per semplificare l'implementazione. Dal punto di vista del codice, per ottenere l'architettura qui mostrata si è operato nella seguente maniera:

```
...
func main(){
    httpPort := os.Getenv("PORT")
    log.Println("Port:", httpPort)
    tcpConn, err := net.Listen("tcp", ":"+httpPort)
    if err != nil {
        log.Fatal(err)
    }

    log.Println("Connect here :", httpPort)
    defer tcpConn.Close()

    for {
        conn, err := tcpConn.Accept()
        if err != nil {
            log.Fatal(err)
        }
        go handleConn(conn)
    }
}

func handleConn(conn net.Conn){
    defer conn.Close()
    ...
}
```

Listing 4.1: Gestione delle connessioni.

Lo snippet appena fornito testimonia il fatto che l'unico compito del server (che è qui individuato dalla funzione *main*) è quello di fornire una porta a cui connettersi (tramite la ricerca del valore della variabile d'ambiente "PORT", di cui viene fatto il display a terminale) per poi accettare connessioni di client (che vengono poi gestite dalla funzione *handleConn*) finché possibile.

4.1.1 I canali di comunicazione

Descritta l'architettura, come fanno i client a dialogare tra di loro?
I client si scambiano informazioni e messaggi tramite l'utilizzo di canali, ognuno adibito al trasporto di dati di un particolare tipo o al fine dell'esecuzione di una particolare operazione. La lista dei canali usati per gestire le comunicazioni tra i "peer" della blockchain è la seguente:

```
var blockChannel = make(map[string] (chan Block))
var transactionChannel = make(map[string] (chan Transaction))

var catchUpChannel = make(map[string] (chan string))
var catchUpBlockChannel = make(map[string] (chan Block))
var catchUpLeaderChannel = make(map[string] (chan []string))
```

```
7  var catchUpEpochChannel = make(map[string] (chan int))
8
9  var electionChannel = map[string]map[int](chan ElectionMessage){}
10 var responseChannel = map[string]map[int](chan ElectionMessage){}
11 var grantChannel = map[string]map[int](chan ElectionMessage){}
12 var leaderChannel = map[string]map[int](chan ElectionMessage){}
```

Listing 4.2: Canali per la comunicazione.

in cui:

- le prime due variabili sono maps in cui la chiave è una stringa rappresentante l'address di un nodo e in cui il valore è un canale, in cui le informazioni da scambiare sono blocchi per *blockChannel* e transazioni per *transactionChannel*;
- il secondo blocco di variabili è costituito da maps in cui la chiave è sempre una stringa che indica l'address di un nodo e in cui il valore è ancora un canale, ma le informazioni che circolano al suo interno servono per gestire la fase di catch-up di un nodo che sta effettuando la procedura di inserimento;
- l'ultimo blocco di dichiarazioni fornisce un elenco di maps annidate: la chiave della map esterna è di nuovo una stringa rappresentante l'address di un nodo, la chiave della map interna è un intero che rappresenta l'epoca corrente e il valore della map interna è un canale in cui scambiare messaggi di elezione per l'epoca corrente.

4.2 Le strutture dati di base

Se nel capitolo terzo di questa tesi si è proceduto a descrivere la struttura base di un blocco, di una transazione e di un messaggio di elezione sulla blockchain Tesium, in questo paragrafo verrà posto il focus su come questi concetti vengono effettivamente implementati e resi strutture Go.
Partendo dalle tre strutture di cui sopra:

```
1  ...
2  type Transaction struct{
3      trashcan string
4      sender string
5      timestamp string
6  }
7
8  type ElectionMessage struct{
9      messageType int
10     payload int
11     sender string
12     dest string
13 }
14
15 type Block struct{
16     transactions []Transaction
17     electionMessages []ElectionMessage
18     epoch int
```

```
19      sender string
20      previousBlock string
21      height int
22  }
23  ...
```

Listing 4.3: Blocchi, transazioni, messaggi di elezione.

Nel caso dei messaggi di elezione, il campo *messageType* può prendere quattro diversi valori, che richiamano le quattro fasi del meccanismo di elezione implementato dall'algoritmo di leader election:

```
1  const(
2      ELECTION int = 0
3      RESPONSE int = 1
4      GRANT int = 2
5      LEADER int = 3
6  )
```

Listing 4.4: Tipi dei messaggi di elezione.

Oltre a queste strutture, ne vengono introdotte altre, in modo tale da facilitare il processo di aggiornamento di alcune variabili fondamentali, come la blockchain, l'epoca, il leader. Ognuna di esse necessita non solo del contenitore delle informazioni (nel caso della blockchain un array di *Block*, o nel caso dell'epoca un semplice intero), ma anche di strumenti che permettano di manipolare questi dati in modo tale da evitare race condition, in quanto l'implementazione dei client si articola su più Goroutines che usufruiscono degli stessi dati. Le strutture in oggetto sono:

```
1  ...
2  type Blockchain struct{
3      chain []Block
4      mempool []Transaction
5      electionpool []ElectionMessage
6      mux sync.Mutex
7  }
8
9  type PeerList struct{
10     peers []string
11     mux sync.Mutex
12 }
13
14 type Leader struct{
15     leader string
16     leaderList []string
17     mux sync.Mutex
18 }
19
20 type Epoch struct{
21     epoch int
22     mux sync.Mutex
23 }
24
25 type Wallet struct{
26     money map[string] int
27     mux sync.Mutex
28 }
```

```
29
30 type ContinueActivities struct{
31     canContinue bool
32     mux sync.Mutex
33 }
34 ...
```

Listing 4.5: Ulteriori strutture dati.

É palese l'esistenza di un campo che accomuna ogni struct qui dichiarata: il campo *mux*, istanza di *sync.Mutex*, fornita direttamente dalle standard libraries di Go per regolare mutua esclusione tramite i metodi *Lock* e *Unlock*.
Analizzando ogni nuovo elemento appena introdotto nel dettaglio:

- la *Blockchain* comprende, oltre all'ovvia sequenza di blocchi, due ulteriori array ausiliari:
  - la *mempool*: un set di transazioni che sono state ricevute e validate dai nodi del sistema, ma non ancora raccolte in un blocco;
  - l'*election pool*: l'omologo della mempool per quanto riguarda i messaggi di elezione;
- la *peerList* rappresenta l'insieme degli address dei nodi che si sono connessi alla blockchain: a parte i canali prima citati, è l'unica variabile globale dell'intera implementazione. Viene aggiornata ogni volta che un peer si connette al sistema. Serve a simulare la conoscenza dei nodi della blockchain a priori, permettendo di evitare politiche di connessione e scoperta dei nodi che sarebbero state esuli dallo scopo di questa tesi;
- *Leader* modella sia il leader corrente che la lista di leader "di scorta";
- *Wallet* contiene una map in cui la chiave è una stringa (che rappresenta l'address di ogni camioncino-peer) e il valore è un intero che rappresenta l'importo di token associato al momento al client rappresentato da quell'address;
- in *ContinueActivities* è incapsulato un booleano che, quando impostato a *false*, obbliga a non eseguire più attività: sarà meglio spiegato in seguito.

4.3 I nodi: variabili e Goroutines di base

4.3.1 Le variabili di base

Data una spiegazione dell'architettura attraverso la quale avviene la simulazione della comunicazione tra i nodi e delle strutture che contengono i dati fondamentali per il funzionamento e l'aggiornamento del processo di scansione dei cassonetti su blockchain, si può continuare questa fase descrittiva tramite la raffigurazione di come queste variabili vengono gestite, e di come interagiscono con la rete architetturale progettata.

Tutte le attività sulla blockchain vengono delegate ai client che, come prima accennato, sono implementati attraverso la redazione della funzione *handleConn*, che riceve come parametro di input *conn*, ovvero la connessione TCP che individua il nodo. Nella fase iniziale della funzione vengono dichiarate le principali variabili locali, come la copia della blockchain riservata a ogni nodo, la copia della lista dei leader, l'epoca, fino ad arrivare alle ultime strutture dati precedentemente mostrate:

```
1  func handleConn(conn net.Conn){
2      defer conn.Close()
3      var blockchain Blockchain
4      blockchain.createFirstBlock()
5
6      var leader Leader
7      var currentEpoch Epoch
8      var activities ContinueActivities
9      activities.set(true)
10     var wallet Wallet
11     wallet.money = make(map[string]int)
12
13     var address string = calculateHash(time.Now().String())
14     setUpChannels(address)
15     io.WriteString(conn, "Here we are: your address is "
16         +address+ "\n")
17     ...
18 }
```

Listing 4.6: La dichiarazione delle variabili in *handleConn*.

Questo frammento di codice introduce alcuni aspetti di cui è utile far menzione:

- all'avvio della connessione, subito dopo l'istanza di una copia della blockchain, viene creato un "blocco zero", come fatto in altre blockchain [11]. Questo blocco è creato vuoto (ovvero non contiene nessuna transazione e/o messaggio di elezione), è ad altezza zero e il suo sender è "Tesium";
- l'address di ogni nodo è il risultato dell'applicazione della funzione hash crittografica SHA256 ad una stringa che rappresenta il timestamp attuale;
- la funzione *setUpChannels* serve per creare, nelle maps rappresentanti i canali prima analizzate, l'entry relativa all'address del nodo che la invoca, in modo tale che possa venire utilizzata anche dagli altri nodi per mandare informazioni;
- la prima riga della funzione serve a ritardare la chiusura della connessione, in modo tale che questa avvenga solamente nell'eventualità che tutte le Goroutines ausiliarie vengano fermate.

É inoltre meritevole di sottolineatura un ulteriore aspetto che dal codice può non trarre il giusto risalto: l'address, fino al punto mostrato, non viene ancora aggiunto alla *peerList* globale. Quest'operazione verrà fatta solamente in seguito alla fase di catch-up, in modo tale che il nuovo nodo venga incluso nelle operazioni che si andranno a descrivere solo quando sarà in uno stato di coerenza della propria blockchain pari a quello di ogni altro nodo.

4.3.2 La procedura di catch-up

Terminata la fase di dichiarazione delle variabili, ogni nodo procede con la fase di catch-up, ovvero la fase in cui si informa dello stato attuale della blockchain contattando altri nodi.
Nel terzo capitolo, si era specificato come la fase di catch-up venisse effettuata solamente durante la seconda parte di ogni epoca, finito il periodo dedicato all'elezione del leader per l'epoca successiva. Non si sono dati però valori circa le effettive durate di ognuno di questi periodi di tempo.
Per questa simulazione, ovviamente, non si poteva tener conto di parametri temporali troppo fedeli alla realtà: siccome lo spostamento tra un'abitazione e un'altra per il carico della spazzatura può richiedere, nel migliore dei casi, almeno qualche minuto, la lunghezza adeguata delle epoche sarebbe dovuta essere intorno ai 60 minuti. Lavorando però all'interno di un ambiente simulato, si è deciso di operare come segue:

- le epoche vengono modellate come slot temporali della durata di 60 secondi. L'inizio di ogni epoca viene fatto coincidere con l'inizio di ogni minuto;
- la prima fase dell'epoca, la fase di elezione, occupa i primi 40 secondi di ogni minuto. Verrà maggiormente specificata la suddivisione temporale di ogni momento dell'elezione in sede di analisi dell'implementazione dell'algoritmo di leader election e del protocollo di consenso distribuito;
- la seconda fase, la fase di catch-up, occupa i restanti 20 secondi.

Finito questo cappello introduttivo, si può descrivere, anche attraverso il codice, come questa fase di inserimento viene gestita:

1. per prima cosa si controlla se la lista dei peers è occupata da qualche nodo: in caso affermativo, si procede con la fase di catch-up; in caso negativo, si è nella situazione in cui il client che esegue questa procedura è il primo che si è connesso. Così, imposterà leader e epoca di conseguenza:

```
1 temppeers := peerList.getList()
2 if len(temppeers) > 0{
3     /* Catchup Phase */
4     ...
5 } else {
6     leader.set(address)
7     currentEpoch.set(0)
8 }
```

2. nel caso in cui la lista dei peers non sia vuota, si aspetta (se necessario) fino al momento giusto dell'epoca tramite l'utilizzo di un Timer, che blocca il chiamante fino allo scattare del secondo preso come input (in questo caso, il quarantunesimo secondo del minuto in cui si entra):

```
1 if len(temppeers) > 0{
2     if time.Now().Second() < 41{
3         io.WriteString(conn, "Waiting for safer moments...\n")
4         newTicker := elseTicker(41)
```

```
5           <- newTicker.C
6       }
7       io.WriteString(conn, "Catching up old blocks\n")
8       ...
9   }
10  ...
```

3. una volta terminata l'eventuale attesa, si procede con l'effettiva interrogazione dei nodi:

```
1   if len(temppeers) > 0{
2       ...
3       var foundInfos bool = false
4       for i := 0; i < len(temppeers) && !foundInfos; i++{
5           addressReceiver := temppeers[i]
6
7           var tempChain []Block
8           catchUpChannel[addressReceiver] <- address
9           time.Sleep(1500 * time.Millisecond)
10          var lengthChannelBlock int =
11              len(catchUpBlockChannel[address])
12
13          var lengthChannelLeader int =
14              len(catchUpLeaderChannel[address])
15
16          var lengthChannelEpoch int =
17              len(catchUpEpochChannel[address])
18
19          if lengthChannelBlock > 0 ||
20              lengthChannelLeader > 0 || lengthChannelEpoch > 0{
21
22              close(catchUpBlockChannel[address])
23              close(catchUpLeaderChannel[address])
24              close(catchUpEpochChannel[address])
25              for i := 0; i<lengthChannelBlock; i++{
26                  gotBlock := <-catchUpBlockChannel[address]
27                  tempChain = append(tempChain, gotBlock)
28              }
29              leaderList := <- catchUpLeaderChannel[address]
30              leader.create(leaderList)
31
32              leader.shootList(conn)
33              newEpoch := <- catchUpEpochChannel[address]
34
35              currentEpoch.set(newEpoch)
36              blockchain.insertBlock(false, tempChain...)
37              wallet.create(tempChain)
38
39              foundInfos = true
40          }
41      }
42      ...
43  }
44  ...
```

il meccanismo è semplice: per ogni nodo nella lista dei peers, si manda una stringa nel canale di query corrispondente, rappresentante il proprio address. Si blocca quindi il chiamante per 1500 millisecondi, al termine dei quali si controlla se almeno un canale mostra di non essere vuoto. In caso affermativo, si procede ad aggiornare le strutture dati di competenza estraendo le informazioni ricevute dai canali;

4. nel caso in cui non si riceva nessun dato da nessun peer il cui address è contenuto nella *peerList* al momento dell'avvio della fase di catch-up, si presume che i nodi precedenti siano incorsi in un crash. Non potendo quindi recuperare la "storiografia" della blockchain, dei leader, delle epoche, si abbandona la simulazione:

```
if len(temppeers) > 0{
    ...
    if !foundInfos{
        conn.Close()
    }
}
...
```

5. qualora la procedura di catch-up sia andata a buon fine, si inserisce l'indirizzo del client nella lista dei peer:

```
if len(temppeers) > 0{
    /* Catchup Phase */
    ...
} else {
    ...
}
peerList.insert(address)
io.WriteString(conn, "Setup completed\n")
...
```

Listing 4.7: La fase di catch-up.

Terminata con successo anche la fase di catch-up, ogni nodo innesca una serie di Goroutines, invocate in parallelo, con lo scopo di aggiornare la blockchain e le strutture dati ad essa connesse, di gestire il processo di creazione dei blocchi (nel caso in cui il nodo in questione sia l'attuale leader) e il processo di elezione per il leader successivo.
A gran parte di queste procedure, ora, è dedicata la spiegazione seguente.

4.3.3 La Goroutine di crash

La prima routine che si andrà a descrivere è una routine che serve a simulare un crash. La scelta di descriverla per prima può sembrare paradossale, ma verrà presto giustificata.
I motivi che hanno portato alla progettazione di una routine che simula un crash sono principalmente due:

- per potere, in fase di testing, dimostrare il funzionamento e la validità dell'implementazione fornita, anche sottoponendo il sistema a situazioni inusuali, come appunto i crash dei nodi che lo compongono;
- per necessità tecniche: per poter simulare Tesium, infatti, occorre prima di tutto far partire il server tramite il comando shell:

```
$ go run chainLibrary.go clients.go goroutines.go election.go
```

che permette l'esecuzione della funzione *main* e il linking a tutte le altre funzioni "di libreria" implementate per questo progetto. Invece, per ogni nodo che si vuole emulare, occorre scrivere su un terminale:

```
$ nc localhost PORTNUMBER
```

in cui PORTNUMBER deve combaciare il valore della variabile d'ambiente PORT mostrata in precedenza. *nc* è un comando che gestisce e fa da listener per connessioni TCP, come quella che viene creata dal *main* di questo progetto. Una volta fatto partire il listener per emulare un nodo, però, l'unico modo per "spegnerlo" è quello di fermare anche l'esecuzione del server, oppure di farlo deliberatamente entrare in una routine che, in questo caso, chiami la funzione *conn.Close*().

In questo modo, premendo il tasto "q" della tastiera seguito da un submit con "Invio", la routine che è qui sotto mostrata imposterà la variabile *activities* (prima dichiarata, e di tipo *ContinueActivities*) a $false$, per poi chiudere la connessione. Il fatto di impostare il parametro *activities* a $false$ serve per evitare che nel mentre vengano eseguite ulteriori operazioni (tra poco se ne avrà un esempio) nel caso in cui la chiusura della connessione richieda un tempo non immediato.

```
1  go func(){
2      for{
3          for scanQuit.Scan(){
4              cmd:= scanQuit.Text()
5              if cmd == "q"{
6                  activities.set(false)
7                  io.WriteString(conn, "CRASH\n")
8                  conn.Close()
9              }
10         }
11     }
12 }()
```

Listing 4.8: Goroutine di crash.

Goroutine di crash a parte, quasi tutte le altre presenti in questo progetto seguono uno schema fisso:

```
1 go func(){
2     for activities.get(){
3         time.Sleep(someTime)
4         exampleGoroutine(params...)
5     }
6 }()
```

Listing 4.9: Schema generale di ogni Goroutine.

4.3.4 La Goroutine di simulazione delle scansioni

La funzionalità richiesta da questa Goroutine è quella di gestione della simulazione della lettura di un codice a barre, quindi della creazione di una transazione:

```
func trashcanGoroutine(address string){
    var randomInt int = int(random(100000, 1))
    var trashcanChecked string =
        calculateHash(calculateHash(strconv.Itoa(randomInt)))
    rightNow := time.Now().String()
    newTransaction :=
        createTransaction(trashcanChecked, address, rightNow)
    sendTransaction(newTransaction)
}
```

Listing 4.10: Goroutine adibita alla creazione delle transazioni.

La routine è tanto breve quanto semplice: si tratta di immettere nel canale adibito alla raccolta delle transazioni di ogni nodo, in media una volta ogni tre secondi (i secondi che si devono aspettare sono pari a un valore ogni volta scelto casualmente in un intervallo che va da 2 a 4 compresi), una nuova transazione, in cui il valore del codice a barre del bidone dell'immondizia è simulato da un valore casuale, ottenuto eseguendo l'hash dell'hash di un numero intero positivo compreso tra 1 e 100000, per evitare il più possibile casi di "omonimia" tra i cassonetti (che verranno comunque gestiti).

4.3.5 La Goroutine di inserimento delle transazioni

Questa Goroutine regola l'inserimento delle transactions nella mempool:

```
func transactionGoroutine(address string,
    blockchain *Blockchain){

    for i := 0; i < len(transactionChannel[address]); i++{
        tr := <-transactionChannel[address]
        (*blockchain).insertIntoMempool(tr)
    }
}
```

Listing 4.11: Goroutine adibita all'inserimento delle transazioni.

Anch'essa può vantare una costruzione piuttosto elementare: all'invocazione (effettuata circa allo scoccare di ogni secondo), controlla la presenza di transazioni nel canale adibito al loro "trasporto", per poi inserirle nella blockchain attraverso la funzione *insertIntoMempool*, anch'essa meritevole di qualche parola di approfondimento:

```
func (b *Blockchain) insertIntoMempool(transaction Transaction){
    b.mux.Lock()
    if !b.checkIfPresent(transaction){
        b.mempool = append(b.mempool, transaction)
    }
    b.mux.Unlock()
}
```

Listing 4.12: Gestire l'inserimento delle transazioni nella mempool.

la funzione in oggetto aggiunge una transazione alla mempool solo se non ne esiste un'altra in uno dei blocchi della blockchain o nella mempool che esibisce lo stesso trashcan.

4.3.6 La Goroutine di inserimento di un blocco

Tramite questa Goroutine si gestisce l'inserimento "safe" di un blocco all'interno della blockchain:

```
1  func blockGoroutine(address string, blockchain *Blockchain,
2      conn net.Conn, wallet *Wallet){
3
4      (*blockchain).mux.Lock()
5      for i := 0; i < len(blockChannel[address]); i++{
6          if err := blockchain.handleInsertIntoChain(
7              <-blockChannel[address], address, wallet); err == nil{
8
9              io.WriteString(conn,
10                 "New block inserted into the blockchain\n")
11         } else {
12             io.WriteString(conn,
13                 "Block invalid: " + err.Error())
14         }
15     }
16     (*blockchain).mux.Unlock()
17 }
```

Listing 4.13: Gestire l'inserimento dei blocchi nella blockchain.

La procedura è omologa a quella di inserimento delle transazioni nella mempool: anche questa funzione prevede un caso di successo e un caso di fallimento, ma occorre esaminare la funzione *handleInsertIntoChain* per avere una panoramica completa del caso:

```
1  func (b *Blockchain) handleInsertIntoChain(block Block,
2      address string, wallet *Wallet) error{
3
4      err := b.checkBlockValidity(block)
5      if err == nil{
6          b.insertBlock(true, block)
7          if(block.sender != address){
8              b.mempool = b.deleteFromMempool(block.transactions)
9              b.electionpool =
10                 b.deleteFromElectionPool(block.electionMessages)
11         }
12         (*wallet).increment(block.sender)
13     } else {
14         for _, t := range block.transactions{
15             if !b.checkIfInmempool(t){
16                 b.mempool = append(b.mempool, t)
17             }
18         }
19     }
20     return err
21 }
```

Listing 4.14: Dettaglio dell'inserimento.

Come mostrato dal codice, questa funzione inserisce il blocco nella catena solo se il blocco passa tutti i test di validità imposti da *checkBlockValidity*, che impongono:

- che il blocco che si sta inserendo abbia come valore del campo *previousBlock* una stringa ottenuta tramite l'hash della concatenazione del campo *epoch* e del campo *height* del blocco precedente:

```
if newTop.previousBlock != calculateHash(string(oldTop.epoch) +
    string(oldTop.height)){

    return errors.New("Hash problem: got "+newTop.previousBlock+
        " instead of " +calculateHash(string(oldTop.epoch) +
        string(oldTop.height))+"\n")
}
```

- che il blocco che si sta inserendo abbia come valore del campo *height* un intero pari al valore del campo *height* del blocco precedente aumentato di 1;
- che non esista già un altro blocco all'interno della blockchain avente lo stesso valore del campo *height* del blocco che si vuole inserire;
- che il blocco che si sta inserendo non esibisca transazioni già inserite in altri blocchi presenti nella blockchain;
- che il blocco che si sta inserendo esibisca solamente messaggi sintatticamente e semanticamente validi.

Nel caso in cui tutti i check vengano passati con successo, si procede all'inserimento nella blockchain e all'applicazione degli effetti del blocco sulle strutture dati ausiliarie, tramite:

- la cancellazione dalla mempool delle transazioni che sono state inserite nel blocco (solamente se il sender è diverso dal client che esegue questa funzione);
- la cancellazione dall'electionpool dei messaggi inseriti (anche in questo caso solamente se il sender è diverso dal client che esegue questa funzione);
- l'aggiornamento dell'entry relativa al publisher nel portafogli, con l'incremento del suo saldo di un'unità.

4.3.7 La Goroutine di aggiornamento del portafogli

Questa Goroutine inserisce nuove entry nel portafogli quando rileva nuovi peers collegati alla blockchain:

```
func peersGoroutine(wallet *Wallet, conn net.Conn){
    temppeers := peerList.getList()
    for _, peer := range temppeers{
        if val := (*wallet).getEntry(peer); val ==-1{
            (*wallet).setEntry(peer)
        }
    }
}
```

Listing 4.15: Goroutine per l'aggiornamento del wallet.

Il funzionamento di questa procedura segue uno schema abbastanza semplice: si fa un "freeze" della lista dei peers in un certo momento, e per ogni peer per cui non è presente un'entry nel portafogli la si aggiunge, impostando il saldo a 0.

4.3.8 La Goroutine di catch-up

L'ultima Goroutine di questa rassegna è impiegata per gestire le risposte ai messaggi di catch-up:

```
1  func catchupGoroutine(blockchain *Blockchain, address string,
2      leader *Leader, epoch *Epoch, conn net.Conn){
3
4      var tempChain []Block
5      var senderLength int = len(catchUpChannel[address])
6
7      for i := 0; i < senderLength; i++ {
8          sender := <-catchUpChannel[address]
9          io.WriteString(conn, "CatchUp message arrived\n")
10
11         (*blockchain).mux.Lock()
12         tempChain = make([]Block, len((*blockchain).chain))
13         copy(tempChain, (*blockchain).chain)
14         (*blockchain).mux.Unlock()
15
16         for _, block := range tempChain{
17             if block.height > 0{
18                 io.WriteString(conn, "Sending Block\n")
19                 catchUpBlockChannel[sender] <- block
20             }
21         }
22
23         var lead []string = (*leader).packLeaders()
24         var ep int = (*epoch).get()
25         io.WriteString(conn, "Sending Leaders\n")
26         catchUpLeaderChannel[sender] <-lead
27         io.WriteString(conn, "Sending Epoch\n")
28         catchUpEpochChannel[sender] <-ep
29         io.WriteString(conn, "Informations sent\n")
30     }
31 }
```

Listing 4.16: Goroutine per le risposte ai messaggi di catch-up.

Non viene esposto niente di complicato: temporaneamente si controlla lo stato del canale in cui vengono trasmesse informazioni che riguardano richieste di catch-up. Se il canale contiene informazioni, significa che è giunta al nodo una richiesta di catch-up da evadere: in tal caso, si fa uno "snapshot" della situazione attuale della blockchain e si provvede a inserire nei canali collegati al mittente della richiesta sopra illustrati le informazioni necessarie a terminare la fase di inserimento nel sistema.

4.4 La creazione dei blocchi da parte del leader

Descritte le routine di base eseguite in loop da ogni nodo che prende parte alla simulazione, si può ora concentrare la trattazione sulla parte più meritevole di interesse: il protocollo di consenso distribuito. In sede di progettazione e modellazione, nel capitolo precedente, si è notato come l'elezione sia addetta a più entità:

- ai nodi della blockchain, che hanno il compito di inviare messaggi conformi alle regole dell'algoritmo di leader election utilizzato;
- al leader dell'epoca in cui l'elezione viene condotta (ovviamente anch'esso facente parte dell'insieme dei nodi della blockchain), che ha il compito di raccogliere tutti i messaggi di elezione scambiati per una certa fase all'interno di un blocco, in modo tale che tutti i nodi impegnati nel processo di scelta della lista per l'epoca successiva possano basarsi su un unico set di informazioni.

In questo paragrafo, verrà accentuato il focus sul lato del processo di elezione che compete al leader: quello della creazione dei blocchi.
Tuttavia, prima di poter mostrare, anche grazie all'ausilio del codice, come questa fase è stata implementata, occorre fare un passo indietro. Nel capitolo terzo si è data profonda illustrazione di come suddividere un'epoca in due fasi, quella di elezione e quella di inserimento dei nuovi nodi (vedesi Figura 3.10). In questo capitolo si sono dati valori temporali concreti per epoca e "macro-slot" temporali: un'epoca dura 60 secondi (parte all'inizio di ogni minuto), di cui i primi 40 secondi sono addetti alla fase di elezione e i rimanenti 20 sono adibiti alla finestra di catch-up.
Entrando più nel dettaglio della fase di elezione, si può delineare questa timeline:

1. al secondo 0 di ogni minuto inizia l'epoca (e quindi il processo di elezione);
2. dal secondo 0 al secondo 5 si esegue la fase di "election", ovvero la prima fase dell'algoritmo di leader election;
3. al secondo 5, il leader pubblica un blocco (contenente i messaggi della fase di "election");
4. dal secondo 10 al secondo 15 si esegue la fase di "response";
5. al secondo 15, il leader pubblica un blocco (contenente i messaggi della fase di "response");
6. dal secondo 20 al secondo 25 si esegue la fase di "grant";
7. al secondo 25, il leader pubblica un blocco (contenente i messaggi della fase di "grant");
8. dal secondo 30 al secondo 35 si esegue la fase di "leader";
9. al secondo 35, il leader pubblica un blocco (contenente i messaggi della fase di "leader");

10. al secondo 40, i nodi traggono le informazioni necessarie contenute nel blocco appena pubblicato, e impostano il nuovo leader.

Grazie a queste specifiche più precise, si può procedere all'analisi della Goroutine che permette a un leader di pubblicare i blocchi giusti nei momenti giusti.
Innanzitutto, può essere utile spiegare la differente clausola invocativa che la differenzia da quelle precedenti descritte:

```
go func(){
    t := minuteTicker()
    for activities.get(){
        <- t.C
        leaderGoroutine(&blockchain, address, conn,
            &currentEpoch, &leader, &wallet, &activities)
        t = minuteTicker()
    }
}()
```

Può essere evidente la differenza tra questo frammento di codice e quello introdotto dal Listing 4.9: nel primo frammento la Goroutine viene lasciata "riposare" per un periodo di tempo sempre definito (anche nel caso di quella che gestisce la creazione delle transazioni), mentre in quello appena esposto viene creato un Ticker puntato ogni volta all'inizio di ogni minuto.
La procedura *leaderGoroutine* lavora secondo il seguente modus-operandi:

1. se il client che la sta eseguendo non è il leader (ovvero la variabile *leader* ha il campo omonimo diverso dal suo address) esce e si mette in attesa del minuto successivo;

2. nel caso in cui, invece, la procedura venga eseguita dal leader, si innesca un ciclo in cui si aspettano a ogni iterazione le scadenze prima stabilite (5, 15, 25 e 35 secondi), all'occorrenza delle quali il leader avvia la procedura di mining tramite una chiamata alla funzione *tryToMine*.

```
func leaderGoroutine(blockchain *Blockchain, address string,
    conn net.Conn, epoch *Epoch, leader *Leader,
    wallet *Wallet, activities *ContinueActivities){

    if (*leader).get() == address{
        var curEpoch int
        var waitTicker *time.Ticker
        stillHere := true

        for i := 0; i < 4 && stillHere; i++{
            waitTicker = elseTicker(10*i + 5)
            <- waitTicker.C
            if i == 0{
                curEpoch = (*epoch).get()
            }
            if (*activities).get(){
                io.WriteString(conn, "Making block "
                    + strconv.Itoa(i+1) + " out of 4\n")

                if (*blockchain).tryToMine(address, curEpoch,
```

```
21                      wallet, conn){
22
23                      io.WriteString(conn, "Block mined!\n")
24                  } else {
25                      io.WriteString(conn, "Mining error!\n")
26                  }
27              } else {
28                  stillHere = false
29              }
30          }
31      }
32  }
```

Listing 4.17: Mining dei blocchi di un'epoca.

Alcuni dettagli non elementari e degni di un approccio più esaustivo:

- viene usata, come funzione generatrice dei Ticker che devono scadere ogni $10 * i + 5$ secondi, una funzione differente da *minuteTicker*, *elseTicker*. Si potrebbe obiettare, a prima vista, che mantenere due funzioni diverse che eseguono la stessa operazione possa essere solamente uno spreco di codice. Si è deciso di mantenere entrambe le funzioni, perché *elsteTicker* effettua controlli in più circa il tempo passato, non necessari per quanto riguarda *minuteTicker*;

- spesso viene controllato che il valore della variabile puntata da *activities* sia impostato a *true*: il motivo è da ricercarsi nello snippet di codice incastonato nel Listing 4.8, in cui si mostra come si può simulare un crash premendo il tasto "q" della tastiera. Nel caso in cui la chiusura della connessione richieda più tempo del previsto, si vuole lo stesso evitare che nel frattempo il nodo "in crash" (che potrebbe essere il leader) compia ulteriori operazioni;

- la funzione *tryToMine* è investita di una forte importanza: è la funzione attraverso la quale un nodo "forgia" effettivamente un blocco. Più precisamente, il nodo cerca di creare, date le informazioni ricavabili dalla blockchain e dalle sue strutture ausiliarie nell'istante in cui la funzione viene invocata, un blocco che sia compatibile con le regole di validazione. Al termine della creazione del blocco si tenta l'inserimento nella blockchain (tramite l'invocazione della funzione precedentemente introdotta *handleInsertIntoChain*) e l'invio del blocco a tutti gli altri nodi appartenenti al sistema (a meno che non occorra un errore):

```
1  func (b *Blockchain) tryToMine(address string, epoch int,
2      wallet *Wallet, conn net.Conn) bool{
3
4      b.mux.Lock()
5      block := b.mine(address, epoch)
6      if err := b.handleInsertIntoChain(block,
7          address, wallet); err == nil{
8
9          sendBlock(block, address)
10         b.mux.Unlock()
11         return true
12     } else {
```

```
13            io.WriteString(conn, err.Error())
14            b.mux.Unlock()
15            return false
16        }
17    }
```

Listing 4.18: Focus sul processo di creazione dei blocchi.

la funzione *mine* altro non fa che creare un blocco che include nell'array delle transazioni tutte le transactions presenti al momento nella mempool del "miner" (eventualmente, anche nessuna), e nell'array dei messaggi tutti i messaggi di elezioni presenti al momento nell'election pool (eventualmente, anche nessuno). Dopodiché, svuota queste due strutture dati.

4.5 L'elezione

Infine, si procede con la descrizione dell'implementazione del processo di elezione, eseguito con lo scopo di determinare il leader e i suoi eventuali sostituti per l'epoca ventura.
Anch'esso è gestito attraverso una Goroutine, simile a quella che serve a invocare la funzione attraverso cui il leader pubblica i quattro blocchi relativi all'epoca per cui viene nominato:

```
1 go func(){
2     t := minuteTicker()
3     for activities.get(){
4         <- t.C
5         electionGoroutine(&blockchain, address,
6             &currentEpoch, &leader, &activities, conn)
7         t = minuteTicker()
8     }
9 }()
```

Anch'essa, infatti, viene invocata sempre all'inizio di ogni minuto attraverso l'impostazione di un ticker che "sveglia" la routine al momento giusto.
La spiegazione di ciò che avviene nella procedura *electionGoroutine* verrà fatta per gradi, fornendo il codice ogni qualvolta sarà necessario spiegare una fase dell'algoritmo.
Per poterla meglio comprendere è però necessario introdurre due semplici procedure, utilizzate nella gestione di eventuali errori che possono occorrere durante il processo di elezione. Come introdotto sommariamente nel capitolo terzo, due sono i possibili casi che possono portare ad un fallimento del meccanismo elettivo:

- la non pubblicazione di un blocco contenente i dati su cui si deve basare una fase dell'algoritmo di leader election;
- la mancanza dei messaggi necessari al continuo della procedura.

Per ambo le casistiche era stata proposta la soluzione che sarebbe poi stata adottata nell'implementazione di Tesium: nel primo caso si sarebbe scelto come nuovo

leader il nodo in prima posizione nella lista dei sostituti, nel secondo caso si sarebbe mantenuto lo stesso leader.
La funzione che gestisce il primo caso di fallimento può essere così modellata:

```
func handleBlockErrors(leader *Leader, conn net.Conn,
    activities *ContinueActivities){

    io.WriteString(conn, "BLOCK ERROR -
        Trying to set next leader in line\n")

    if (*leader).setNextInLine(){
        io.WriteString(conn, "Setting as new leader: "
            +(*leader).get()+"\n")

        (*leader).shootList(conn)
    } else {
        io.WriteString(conn, "Failure - closing everything\n")
        (*activities).set(false)
        conn.Close()
    }
}
```

Listing 4.19: Gestione degli errori derivanti da mancanza di blocchi.

Nel caso in cui la lista dei leader sia vuota ovviamente occorre bloccare la simulazione, in quanto non è possibile eleggere in maniera distribuita un nuovo leader.
La funzione che invece gestisce il secondo caso problematico è molto più semplice: mostra solamente un messaggio di errore tramite la stampa di una stringa adeguata:

```
func handleElectionErrors(conn net.Conn){
    io.WriteString(conn, "ELECTION ERROR
        - maintaining the current Leader\n")
}
```

Listing 4.20: Gestione degli errori derivanti da mancanza di messaggi.

Avendo a disposizione anche queste due procedure, si può passare alla descrizione di *electionGoroutine*, che fa da involucro alle quattro fasi dell'algoritmo di leader election. Si parta dall'inizio:

1. all'invocazione della funzione, viene aggiornata l'epoca tramite l'incremento della variabile che ne tiene la "contabilità", viene svuotata l'election pool (per motivi di sicurezza: se l'elezione precedente non è andata a buon fine, questa struttura dati potrebbe ancora conservare messaggi di quell'elezione) e viene analizzata l'attuale situazione del sistema:

    - se il sistema non è l'unico nodo del sistema, allora si continua la procedura di elezione;
    - altrimenti, il nodo chiamante imposta se stesso come leader (in quanto è l'unico) per l'epoca ventura.

```
func electionGoroutine(blockchain *Blockchain, address string,
    epoch *Epoch, leader *Leader,
    activities *ContinueActivities, conn net.Conn){
```

```
4
5       (*epoch).increment()
6       temppeers := peerList.getList()
7       ep := (*epoch).get()
8       (*blockchain).flushElectionPool()
9       if len(temppeers) > 1{
10          /* Election Phase */
11          ...
12      } else {
13          t := elseTicker(40)
14          <- t.C
15          (*leader).set(address)
16          (*leader).leaderList = []string{}
17      }
18  }
```

2. Nel caso in cui non sia l'unico peer del sistema, viene effettuato un ulteriore check: si controlla che la blockchain non sia vuota. Il motivo è di facile comprensione: se la blockchain è ancora vuota al momento di inizio di un'epoca, allora non conterrà alcuna transazione. Di conseguenza, ogni identificatore usato dai nodi per l'elezione di quell'epoca comincerebbe con il numero "0". Quindi si evita di eseguire il meccanismo di elezione, mantenendo come leader il nodo che lo è già al momento e lasciandogli creare i blocchi che potranno fungere da "base" per l'elezione da tenere durante l'epoca successiva (da notare che il caso in cui gli identificatori di ogni nodo possano essere "0" all'inizio di un'epoca è possibile anche quando il leader dell'epoca precedente subisce un crash prima di pubblicare il primo blocco: in quel caso l'elezione viene comunque eseguita).

```
1  ...
2  if len(temppeers) > 1{
3      top := (*blockchain).getTop(false)
4      if top.height > 0{
5          /* Election Phase */
6          ...
7      } else {
8          io.WriteString(conn, "Empty Blockchain...\n")
9      }
10 }
11 ...
```

3. Nel caso in cui l'altezza della blockchain sia quindi maggiore di 0, viene attivata una piccola Goroutine, adibita alla ricezione dei messaggi dai canali rappresentanti ogni fase dell'elezione. In questo modo, in maniera del tutto parallela, viene continuamente aggiornata l'electionpool di ogni nodo, in modo tale che al momento in cui uno di essi (ovvero il leader) dovrà creare un blocco, avrà a disposizione tutti i messaggi di elezione da inserire al suo interno.

4. Viene costruita la map degli identificatori relativi all'epoca corrente (basati quindi sull'epoca precedente).

5. Vengono invocate, l'una dopo l'altra, le quattro fasi dell'algoritmo di leader election. Per ognuna viene controllato, prima di procedere con la sua invocazione, che il parametro *activities* punti ancora a un valore *true*: in caso contrario ovviamente viene fermato il meccanismo di elezione, in maniera analoga a quanto spiegato per la creazione dei blocchi da parte del leader. Ogni fase dell'algoritmo è implementata come una funzione che restituisce un intero rappresentante l'esito della fase, in modo tale che *electionGoroutine* possa analizzare il responso per prendere le adeguate contromisure in caso di errore.

4.5.1 La fase di "election"

La prima fase dell'algoritmo è modellata da questa funzione:

```
1  func firstPhase(address string, identifiers map[string]string,
2      ep int, peers []string){
3
4      var majorIds []string = getMajorpeers
5          (identifiers, identifiers[address])
6
7      for _, id := range majorIds{
8          newElectionMessage := createElectionMessage
9              (ELECTION, identifiers[address], id, 0)
10
11         sendMessage(newElectionMessage, peers, ep)
12     }
13 }
```

Listing 4.21: La fase di "election".

come si può vedere, questa funzione crea, per ogni peer avente un identificatore maggiore di quello del chiamante, un messaggio di elezione, da mandare poi a tutti i nodi (in quanto deve essere validato da tutti). Il payload del messaggio viene lasciato uguale a 0, non avendo particolare significato in questa fase.

4.5.2 La fase di "response"

Terminata la prima fase, si punta un timer che serva a svegliare la routine in tempo per la seconda fase, ovvero quella di "response":

```
1  func secondPhase(minGlobalSender *string, address string,
2      currentTop Block, previousTop Block, peers []string,
3      ep int, identifiers map[string]string) int{
4
5      var minRelativeSender string
6      if !compBlocks(currentTop, previousTop){
7          if len(currentTop.electionMessages) > 0{
8              *minGlobalSender =
9                  getGlobalMin(currentTop.electionMessages)
10
11             if(isThereAMessageForMe(identifiers[address],
12                 currentTop)){
13
14                 minRelativeSender =
```

```
15                      getMinElectionIdentifier(currentTop.
16                      electionMessages, identifiers[address])
17
18                  newResponseMessage :=
19                      createElectionMessage(RESPONSE,
20                      identifiers[address], minRelativeSender, 0)
21
22                  sendMessage(newResponseMessage, peers, ep)
23              }
24              return OK
25          } else {
26              return ELECTION_ERROR
27          }
28      } else {
29          return BLOCK_ERROR
30      }
31  }
```

Listing 4.22: La fase di "response".

in questa fase viene dapprima controllata la differenza tra due blocchi dati in input: *previousTop* e *currentTop*. Entrambi rappresentano l'ultimo blocco in ordine cronologico pubblicato sulla blockchain, ma in due momenti distinti: *previousTop* rappresenta un momento precedente alla fase di creazione del blocco che contiene i messaggi di tipo "election", mentre *currentTop* rappresenta il momento successivo. Il caso in cui i blocchi due coincidano significherebbe che non è stato pubblicato il blocco su cui basare la fase di "response", e quindi si necessiterebbe dell'intervento della funzione *handleBlockErrors* descritta nel Listing 4.19. Nel caso in cui i blocchi siano diversi, si controlla l'effettiva presenza di messaggi di elezione nel blocco in oggetto. Se anche questo controllo dà esito positivo, si procede con la vera procedura di "response", rispondendo ai messaggi eventualmente ricevuti emettendo un messaggio rivolto al mittente con identificatore minimo. Anche in questo caso, il payload del messaggio viene lasciato a 0, in quanto non interessante ai fini della fase successiva.

4.5.3 La fase di "grant"

Si passa quindi alla terza fase dell'algoritmo di leader election (sempre dopo aver aspettato la scadenza prestabilita): la fase di "grant", in cui il delegato a formulare la lista crea un messaggio per ogni nodo da inserire, in cui il payload indica la posizione che quello andrà ad occupare all'interno della suddetta:

```
1  func thirdPhase(address string, currentTop Block,
2      previousTop Block, minGlobalSender string,
3      ep int, identifiers map[string]string,
4      peers []string) int{
5
6      if !compBlocks(currentTop, previousTop){
7          if len(currentTop.electionMessages) == 0 &&
8              minGlobalSender == identifiers[address]{
9
10             newGrantMessage :=
11                 createElectionMessage(GRANT,
12                 identifiers[address], identifiers[address], 1)
```

```
13
14              sendMessage(newGrantMessage, peers, ep)
15          } else {
16              if len(currentTop.electionMessages) > 0 &&
17                  minGlobalSender == identifiers[address]{
18
19                  var senders []string =
20                      getOrderedSenders(currentTop.electionMessages)
21
22                  for pos, sender := range senders{
23                      newGrantMessage := createElectionMessage
24                          (GRANT, identifiers[address],
25                           sender, pos+1)
26
27                      sendMessage(newGrantMessage, peers, ep)
28                  }
29                  newGrantMessage :=
30                      createElectionMessage(GRANT, identifiers[address],
31                      identifiers[address], len(senders)+1)
32
33                  sendMessage(newGrantMessage, peers, ep)
34              }
35          }
36          return OK
37      } else {
38          return BLOCK_ERROR
39      }
40  }
```

Listing 4.23: La fase di "grant".

Valgono anche per questa fase le considerazioni fatte per la fase di "response": nel caso in cui non venga pubblicato il blocco su cui basare le azioni da eseguire in questa fase, si restituisce un errore che provocherà la chiamata di *handleBlockError* all'interno della procedura *electionGoroutine*. In questo caso invece è ammissibile che non esistano messaggi nel blocco: si prenda ad esempio il caso in cui nessuno risponda al nodo con minimo id che ha inviato un messaggio di elezione. Il nodo dovrà pubblicare un messaggio di grant destinato a se stesso, in modo tale da incoronarsi unico leader, senza eventuali sostituti.
Nel caso in cui l'ordine degli eventi segua invece la linea prestabilita, il nodo con identificatore minimo tra quelli che avevano inviato i messaggi di elezione nella prima fase costruisce la lista dei leader tramite l'ordinamento dei mittenti dei messaggi di "response" a lui indirizzati, inserendo poi se stesso in coda.

4.5.4 La fase di "leader"

Nella quarta fase dell'algoritmo, i nodi devono solamente accettare il loro posto nella gerarchia imposta dal peer che ha inviato i vari messaggi di grant. Spingendosi più nel dettaglio:

```
1  func fourthPhase(address string, currentTop Block,
2      previousTop Block, ep int, identifiers map[string]string,
3      peers []string) int{
4
```

```
5       if !compBlocks(currentTop, previousTop){
6           if len(currentTop.electionMessages) > 0{
7               for _, message := range currentTop.electionMessages{
8                   if message.dest == identifiers[address]{
9                       newLeaderMessage :=
10                          createElectionMessage(LEADER,
11                          identifiers[address], "", message.payload)
12
13                      sendMessage(newLeaderMessage, peers, ep)
14                  }
15              }
16              return OK
17          } else {
18              return ELECTION_ERROR
19          }
20      } else {
21          return BLOCK_ERROR
22      }
23  }
```

Listing 4.24: La fase di "leader".

Nell'evenienza in cui non vengano registrati errori, la funzione controlla che il nodo chiamante sia destinatario di uno dei messaggi di grant spediti nella fase precedente dell'algoritmo. In tal caso, crea un messaggio senza destinatari e con payload uguale al payload del messaggio di grant da lui ricevuto, per confermare la posizione conferitagli all'interno della lista dei leader.

4.5.5 Aggiornamento del leader

A conclusione della fase di elezione dell'epoca vi è la fase di aggiornamento di tutte le strutture dati collegate alla memorizzazione del leader dell'epoca successiva e della relativa lista di sostituti. Questa operazione è gestita dalla seguente funzione:

```
1   func setLeaderFromElection(currentTop Block, previousTop Block,
2       identifiers map[string]string, leader *Leader) int{
3
4       if !compBlocks(currentTop, previousTop){
5           if len(currentTop.electionMessages) > 0{
6               var senders []string = getOrderedLeaders
7                   (currentTop.electionMessages)
8
9               var addressSenders []string =
10                  getListFromIds(identifiers, senders)
11              (*leader).create(addressSenders)
12              return OK
13          } else {
14              return ELECTION_ERROR
15          }
16      } else {
17          if (*leader).setNextInLine(){
18              return BLOCK_ERROR
19          } else {
20              return FAILURE
21          }
```

```
22        }
23    }
```

Listing 4.25: Aggiornamento del leader.

Operati i controlli ormai largamente spiegati in precedenza, nel caso in cui questi restituiscano esito positivo, la procedura interviene creando una lista ordinata di address ricavata dagli identificatori dei sender di ogni messaggio di "leader" e dal payload di quel messaggio. Così, tramite la funzione *create*, la lista in input viene divisa in due parti:

- una, composta da un solo elemento, che andrà a individuare l'address del nuovo leader;
- un'altra, composta dagli altri elementi, che andrà a individuare la lista dei sostituti.

Nel caso in cui, invece, sia occorso un errore scaturito dalla mancanza del blocco dedito alla promulgazione di tale lista, si tenta la "dequeue" dalla lista dei leader, nominando come nuovo incaricato a creare i blocchi per l'epoca successiva il nodo denotato dall'address appena tolto dalla lista. Nel caso in cui anche questa operazione non dia esito positivo, si restituisce un codice indicante la necessità dello shutdown della simulazione, in quanto non sarà possibile eleggere un nuovo leader senza nessuno incaricato di pubblicare i blocchi durante l'elezione seguente.

Capitolo 5

Un esempio

A conclusione di questo lavoro, viene lasciato al lettore un esempio dell'esecuzione del progetto appena esposto. Per facilitare la comprensione dell'ordine degli avvenimenti di ogni epoca, sono state aggiunte, nei punti del codice in cui vengono gestite parti salienti dell'implementazione, ulteriori stampe.
I log e i comandi che verranno riportati sono tutti stati mostrati o eseguiti su una shell Unix.

5.1 L'avvio del server

La prima cosa da fare in ordine cronologico per avviare la simulazione è, come si può immaginare, avviare il server.
Supponendo sia la prima volta in cui ci si approccia all'avvio di questa piccola blockchain, il primo passo da compiere è quello di dare un valore concreto alla variabile d'ambiente PORT, cercando di scegliere una porta che non sia conosciuta come fondamentale per il funzionamento di altri protocolli (i.e.: la porta 80). Per esempio, si può usare la porta 9000. Il comando che deve essere digitato sul terminale è il seguente:

```
$ export PORT=9000
```

Dato un valore concreto a PORT, è possibile avviare il server, scrivendo il seguente comando:

```
$ go run chainLibrary.go clients.go goroutines.go election.go
```

A comando eseguito, il server comincerà la sua attività di ascolto di nuove connessioni in entrata, e sul terminale verranno stampate queste linee:

```
PORT: 9000
Connect here : 9000
```

5.2 La connessione del primo nodo

Il passo successivo è quello di tentare la connessione del primo nodo alla blockchain. Per fare questo, si necessita l'apertura di un nuovo terminale (a cui ci si

rivolgerà tramite il nome *Terminale 1* nelle prossime pagine), in cui eseguire il seguente comando:

```
$ nc localhost 9000
```

Grazie a questo comando, si creerà una connessione TCP sulla porta 9000, verrà notificato il server della connessione e eseguita la funzione *handleConn*. Essendo questo il primo nodo collegato, non sarà necessario effettuare la procedura di catch-up: imposterà l'epoca corrente a "zero" e se stesso a leader. Il suo set-up sarà quindi abbastanza sintetico:

```
Here we are: your address is 97da0373b04086055415fc9f2dd021d8cc0c4
8f8ac8539d4da7b784b7912b89c
Setup completed
```

Subito dopo aver mostrato l'address del nodo appena creato, sul *Terminale 1* viene stampata anche questa riga, figlia dell'esecuzione di *peersGoroutine*:

```
New entry for the wallet - 97da0373b04086055415fc9f2dd021d8cc0c4
8f8ac8539d4da7b784b7912b89c
```

Dopodiché, il nodo inizia a creare transazioni ad intervalli più o meno regolari di tempo (in media una ogni tre secondi), che vengono stampate a terminale. Eccone un esempio concreto:

```
Passed by this trashcan: 8cfb8fb3c17cb2ac31926144d78242b630de26
3be510752a901e4f6969e5037f at this time:
2018-09-22 21:01:16.940733491 +0200 CEST
```

5.3 La connessione di nuovi nodi e il catch-up

Sempre nello stesso minuto in cui viene introdotto il primo nodo, se ne vogliono introdurre altri due. Naturalmente è sufficiente mostrare una sola volta la procedura di inserimento, in quanto entrambi condurrebbero procedure di inserimento pressoché analoghe.
Si chiami quindi *Terminale 2* il terminale in cui si esegue un nuovo comando di "accensione", e si analizzino i log da esso stampati:

```
Here we are: your address is 261fde26de17c6ffbbcd348ed7
b823f2bbba7f467ad0059127b1155e25468807
Waiting for safer moments...
Catching up old blocks
Querying address 97da0373b04086055415fc9f2dd021d8cc0c4
8f8ac8539d4da7b784b7912b89c
Synchronizing chain...
Got leader: 97da0373b04086055415fc9f2dd021d8cc0c4
8f8ac8539d4da7b784b7912b89c
Got epoch: 0
Setup completed
```

Come si può facilmente osservare, il nodo appena collegatosi al sistema aspetta il momento giusto per avviare la procedura di inserimento. Giunto il momento, il nodo interroga l'unico altro client già collegato, dal quale riceve i blocchi da inserire (in questo caso nessuno) e mostra di avere ottenuto anche le informazioni

riguardanti il leader corrente e l'epoca vigente.
Si mostrino anche i log stampati dal nodo interrogato:

```
CatchUp message arrived from 261fde26de17c6ffbbcd348ed7
b823f2bbba7f467ad0059127b1155e25468807
Sending Leaders
Sending Epoch 0
Informations sent
```

Questi risultano speculari a quelli del nodo interrogante.
Come asserito poc'anzi, la procedura che seguirà l'altro nodo (creato nel *Terminale 3*) nel connettersi alla blockchain sarà analoga a quella appena mostrata. Tuttavia, per completezza di informazioni, si ritiene comunque necessario fornire l'address del nuovo nodo, in modo tale da evitare ambiguità in futuro in merito alle identità dei client:

```
Here we are: your address is c25d3c2c64e5a56470dfa4f4ab
996da13b7b78f28be4c7cfb060fae88aa4d6c0
```

5.4 La prima epoca

Iniziato il minuto successivo a quello in cui vengono creati tutti e tre i nodi tramite i loro rispettivi terminali, si entra nell'epoca successiva, ovvero l'epoca "uno".
Durante quest'epoca, come si può intuire dalle disposizioni date nel capitolo precedente, non avverrà il processo di elezione: non esiste ancora alcun blocco su cui basare il calcolo degli identificatori per l'esecuzione del Bully Algorithm implementato per l'occasione. L'unica azione che compirà *electionGoroutine* per ogni nodo, oltre a quella di aggiornare l'epoca, sarà stampare questo log:

```
Empty Blockchain - waiting...
```

Questo non vuol dire che in questa epoca non vengano pubblicati blocchi: il nodo "residente" sul *Terminale 1* è il leader corrente, e ad ogni scadenza (5, 15, 25, 35 secondi) stampa sul terminale un log di questo tipo:

```
Mining block 1 out of 4
Block mined!
```

Mentre i terminali *Terminale 2* e *Terminale 3*, ogni volta che il leader pubblica un blocco, provvedono ad informare la ricezione di quest'ultimo stampando:

```
New block inserted into the blockchain
```

5.5 La seconda epoca: l'elezione

Allo scatto del minuto successivo, inizia l'epoca "due", questa volta ospitante un vero e proprio processo di elezione, che sarà descritto passo per passo, in modo tale da poter fornire una maggiore chiarezza.

1. secondo 0 - *Terminale 1, Terminale 2, Terminale 3*: inizia la prima fase, quella di "election". Vengono creati gli identificatori univoci e mandati i messaggi di "election":

```
THIS SHOULD BE ELECTION PHASE
ID: 32.ADDRESS.97da0373b04086055415fc9f2dd021d8
cc0c48f8ac8539d4da7b784b7912b89c
ID: 21.ADDRESS.261fde26de17c6ffbbcd348ed7b823f2
bbba7f467ad0059127b1155e25468807
ID: 21.ADDRESS.c25d3c2c64e5a56470dfa4f4ab996da1
3b7b78f28be4c7cfb060fae88aa4d6c0
```

2. secondo 5 - *Terminale 1*: viene creato il primo blocco:

```
Mining block 1 out of 4
Block mined!
```

3. secondo 10 - *Terminale 1, Terminale 2, Terminale 3*: inizia la fase di "response":

```
THIS SHOULD BE RESPONSE PHASE
```

4. secondo 15 - *Terminale 1*: viene creato il secondo blocco:

```
Mining block 2 out of 4
Block mined!
```

5. secondo 20 - *Terminale 1, Terminale 2, Terminale 3*: inizia la fase di "grant":

```
THIS SHOULD BE GRANT PHASE
```

6. secondo 25 - *Terminale 1*: viene creato il terzo blocco:

```
Mining block 3 out of 4
Block mined!
```

7. secondo 30 - *Terminale 1, Terminale 2, Terminale 3*: inizia la fase di "leader":

```
THIS SHOULD BE LEADER PHASE
```

8. secondo 35 - *Terminale 1*: viene creato il quarto blocco:

```
Mining block 4 out of 4
Block mined!
```

9. secondo 40 - *Terminale 1, Terminale 2, Terminale 3*: viene consultato il quarto blocco per inferire la lista dei leader per l'epoca successiva:

```
LEADER BLOCK:
And the leader is 97da0373b04086055415fc9f2dd021d8cc0c4
8f8ac8539d4da7b784b7912b89c
Pos: 1 address: 261fde26de17c6ffbbcd348ed7b823f2bbba7f4
67ad0059127b1155e25468807
Pos: 2 address: c25d3c2c64e5a56470dfa4f4ab996da13b7b78f
28be4c7cfb060fae88aa4d6c0
```

A questo punto il processo di elezione può dichiararsi completato.

5.6 La terza epoca: il crash del leader

Aspettato anche lo scorrere dei 20 secondi dell'epoca precedente, scatta il nuovo minuto e, con esso, una nuova epoca. L'elezione dell'epoca "tre" si esegue sulla falsa riga di quella precedente.
Si usi questa fase di elezione per introdurre un'ulteriore casistica come esempio: il crash del leader. Per descrivere questa situazione, verrà analizzata nuovamente nel dettaglio di ogni passo la procedura di elezione:

1. secondo 0 - *Terminale 1, Terminale 2, Terminale 3*: inizia la fase di "election". Vengono creati gli identificatori univoci (stavolta basati sull'epoca "due") e mandati i messaggi di "election":

```
THIS SHOULD BE ELECTION PHASE
ID: 24.ADDRESS.97da0373b04086055415fc9f2dd021d8
cc0c48f8ac8539d4da7b784b7912b89c
ID: 24.ADDRESS.261fde26de17c6ffbbcd348ed7b823f2
bbba7f467ad0059127b1155e25468807
ID: 22.ADDRESS.c25d3c2c64e5a56470dfa4f4ab996da1
3b7b78f28be4c7cfb060fae88aa4d6c0
```

2. secondo 5 - *Terminale 1*: viene creato il primo blocco;

3. secondo 7 - *Terminale 1*: viene invocata la procedura di crash tramite il submit della lettera "q";

4. secondo 10 - *Terminale 2, Terminale 3*: inizia la fase di "response";

5. secondo 20 - *Terminale 2, Terminale 3*: i nodi notano che il secondo blocco non è stato pubblicato, e perciò invocano la procedura di errore *handleBlockError*:

```
BLOCK ERROR - Trying to set next leader in line
Setting as new leader: 261fde26de17c6ffbbcd348ed7b823f2bbba7
f467ad0059127b1155e25468807
Pos: 1 address: c25d3c2c64e5a56470dfa4f4ab996da13b7b78f28be4
c7cfb060fae88aa4d6c0
```

La procedura di elezione, per quest'epoca, è già terminata, anche se in maniera brusca. Come si può notare, il nodo che era precedentemente in prima posizione nella lista dei sostituti diventa il leader per quella che sarà l'epoca "quattro", mentre il nodo che occupava il secondo posto diventa il diretto sostituto del nuovo leader.

Capitolo 6

Conclusioni

La simulazione della blockchain proposta in questo elaborato è un lavoro modesto, ma che fa affidamento su un'architettura in cui si può facilmente supporre un insieme di canali di comunicazione pienamente affidabili. Nel caso in cui si volesse estendere questo progetto al fine di offrire una vera e propria implementazione per soddisfare le esigenze di un'autentica realtà aziendale già collaudata (si supponga sempre nell'ambito scelto in questa tesi, ovvero la raccolta differenziata dei rifiuti), si dovrà porre particolare attenzione sull'implementazione di metodologie di comunicazione più reali tra i peer, introducendo e gestendo anche i rischi collegati alla perdita dei messaggi che queste possono comportare. Cionondimeno, occorrerà definire in modo differente i vari parametri che regolano lo scorrere degli eventi in questa blockchain, come la lunghezza delle epoche o delle procedure di elezione. Infine, si potrebbe creare un meccanismo di valutazione delle transazioni non puramente aritmetico ma ponderato: scansionare quattro bidoni dell'immondizia in una via del centro può richiedere un tempo decisamente inferiore rispetto a quello che può servire per scansionarne altrettanti in aperta campagna. Con un meccanismo di creazione degli identificatori puramente aritmetico si rischia di avvantaggiare in maniera conclamata chi è addetto alla raccolta differenziata nelle aree urbane del comune di competenza a svantaggio di chi invece opera nelle zone rurali.

Nonostante le varie semplificazioni introdotte all'interno di questa simulazione, la blockchain Tesium riesce a soddisfare i due principali bisogni descritti: quello di mantenere la "contabilità" del processo di raccolta dell'immondizia a domicilio, e quello di fungere da "distributore" di premi nei confronti dei dipendenti più operosi. Quest'ultimo requisito in particolare è realizzato tramite l'implementazione di un protocollo distribuito orientato alla valorizzazione del merito, che riesce a fare dialogare algoritmi di leader election sincroni con un sistema totalmente asincrono come blockchain.

Ringraziamenti

Devo innanzitutto ringraziare il Professor Cosimo Laneve per l'opportunità concessami di approfondire un interessante ambito di studio e per la pazienza sempre dimostrata nei miei confronti. Desidero inoltre ringraziare la Dottoressa Adele Veschetti per la sua infinita disponibilità e per i preziosi consigli spesso elargiti.
Voglio dire grazie alla mia famiglia, per non avermi mai fatto mancare il sostegno necessario durante questo lungo e tortuoso percorso: anche quando le nubi erano scure, non avete mai smesso di ricordarmi che la pioggia è passeggera, e che dopo il temporale torna sempre il sereno.
Infine mi sento in dovere di ringraziare gli amici: ognuno, a modo suo, ha contribuito a farmi raggiungere questo traguardo quasi impensabile. Siete il sale della mia vita.

Glossario

blocco struttura dati che funge da contenitore per le informazioni che vengono trasmesse sulla blockchain.

blockchain database decentralizzato la cui rappresentazione é effettuata tramite una sequenza di blocchi back-linked.

Bully Election Algorithm algoritmo di leader election per sistemi distribuiti sincroni proposto da Hector Garcia-Molina nel 1982, in cui si propone di trovare il leader come il nodo non in crash avente l'identificatore massimo nella rete.

elezione meccanismo con cui viene scelto il leader per l'epoca successiva.

epoca lasso di tempo arbitrariamente ampio in cui viene suddiviso lo scorrere del tempo fisico. É suddivisibile in due ulteriori fasi: la fase di elezione, e la fase di catch-up.

Go linguaggio di programmazione imperativo sviluppato dagli ingegneri di Google. Utilizzato in questo elaborato come base per costruire la blockchain Tesium.

leader nodo della blockchain addetto alla creazione dei blocchi per una certa epoca.

miner in blockchain come Bitcoin, é un nodo che si cimenta nell'attivitá di risoluzione di un puzzle crittografico per pubblicare un blocco, a fronte di eventuali ricompense.

nodo uno dei tanti peer che decide di scaricare una copia del ledger sul proprio dispositivo e di eseguire un client che permetta di interagire con la blockchain.

Tesium nome dell'azienda fittizia per cui sviluppare la blockchain qui trattata; nome della blockchain implementata.

transazione l'informazione di granularitá minima che puó transitare sulla blockchain: puó riguardare transazioni finanziarie oppure semplici operazioni di "catasto".

Bibliografia

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] "Dieci Volte Meglio: una visione dell'Italia." `https://www.diecivoltemeglio.com/programma`, 2018.

[3] "GLOBAL BITCOIN NODES DISTRIBUTION." `https://bitnodes.earn.com/`.

[4] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*. Addison-Wesley Professional, 1st ed., 2015.

[5] K. Patel, "Why should you learn Go?." `https://medium.com/exploring-code/why-should-you-learn-go-f607681fad65`, January 2017.

[6] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Generation Comp. Syst.*, vol. 88, pp. 173–190, 2018.

[7] H. Garcia-Molina, "Elections in a Distributed Computing System," *IEEE Trans. Computers*, vol. 31, no. 1, pp. 48–59, 1982.

[8] M. Kordafshari, M. Gholipour, M. Jahanshahi, A. T. Haghighat, and M. DEHGHAN, "Modified bully election algorithm in distributed systems," vol. 2, 08 2005.

[9] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pp. 66–98, 2018.

[10] J. Siim, "Proof-of-Stake," 2017.

[11] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 1st ed., 2014.