ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea in Ingegneria e Scienze Informatiche

TECNOLOGIA BLOCKCHAIN: ANALISI TECNICA E SVILUPPO DI UN'APPLICAZIONE DECENTRALIZZATA PER VOTAZIONI ELETTRONICHE

Elaborato in

SISTEMI EMBEDDED E INTERNET-OF-THINGS

Relatore

Prof. ANDREA OMICINI

Co-relatore

Dott. GIOVANNI CIATTO

Presentata da

ANDREA CARDIOTA

Terza Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

crittografia

blockchain

decentralizzazione

ethereum

solidity

A nonno Vincenzo

Abstract

Scopo della tesi è la progettazione e sviluppo di un'applicazione distribuita e decentralizzata, basata su tecnologia Blockchain, il cui obiettivo è di fornire una possibile alternativa a quello che è il sistema di votazioni elettorali attualmente in uso, sfruttando la piattaforma Ethereum ed il linguaggio di programmazione Solidity. Mediante l'utilizzo della suddetta applicazione, è possibile registrare il proprio voto con la certezza che la sua integrità e segretezza non vengano compromesse, escludendo inoltre la possibilità che possa essere considerato nullo. Queste funzionalità saranno rese disponibili agli utenti finali mediante l'utilizzo di uno strumento, Metamask, che permette di accedere e sfruttare servizi web distribuiti senza l'impiego di un client apposito, ma tramite un comune web browser. È quindi presentata un'analisi approfondita delle tecnologie sin qui citate, utile alla comprensione e formulazione di un modello di software distribuito moderno che sfrutti una tecnologia Blockchain, al fine di introdurre considerevoli vantaggi non raggiungibili con le tecnologie attuali.

Indice

Introduzione

Nell'ultimo decennio abbiamo assistito alla crescita esponenziale di una tecnologia nata contestualmente all'invenzione del Bitcoin e delle cripto-valute in generale: la *Blockchain*. Quest'ultima rappresenta un'importante evoluzione ed innovazione non solo in informatica, ma anche in ambito finanziaro, medico e politico.
La Blockchain è un "incorruttibile" registro digitale di transazioni, di svariato tipo, che può essere programmato per registrare, in maniera definitiva e quindi incancellabile, dati di qualsiasi natura e valore.
Tale tecnologia viene etichettata da molti come rivoluzionaria per diversi motivi, prima fra tutti la restituzione del totale controllo delle informazioni e transazioni all'utente finale. Dal punto di vista puramente tecnico esistono una serie di vantaggi che rendono la Blockchain una tecnologia migliore (per molto aspetti, ma non per tutti) delle altre attualmente disponibili.
I dati memorizzati sulla Blockchain sono consistenti, accurati, temporizzati e largamenti disponibili. Inoltre, data la natura decentralizzata della tecnologia, la difesa da potenziali attacchi è considerevolmente più robusta e, in aggiunta, non esiste un singolo punto di rottura.
La Blockchain permette intrinsecamente agli utenti di sapere che le transazioni verranno eseguite seguendo le specifiche del protocollo, eliminando del tutto il bisogno di terze parti che introducono - nelle tecnologie meno recenti - complessità e ne riducono l'integrità. Quest'ultima infatti è il punto di forza della tecnologia oggetto di studio, in quanto ogni cambiamento ed aggiornamento sulla Blockchain è pubblicamente visibile ad ogni utente. Tutte le transazioni sono inoltre immutabili, quindi non possono essere alterate o cancellate. Altri

vantaggi di rilevante importanza sono la velocità delle transazioni e la consistente riduzione delle commissioni. Le transazioni vengono infatti eseguite nel giro di qualche ora o addirittura qualche istante - a seconda della Blockchain utilizzata - rispetto agli svariati giorni necessari per una transazione interbancaria. Eliminando il bisogno di terze parti, si riducono drasticamente i costi di *overhead* portando ad un totale di commissioni necessarie significativamente inferiore per l'utente finale.

L'obiettivo di questa tesi è sviluppare un'applicazione web che si occupi di gestire una votazione elettronica mediante l'utilizzo di tale tecnologia.
La Blockchain non fu inizialmente pensata e progettata per memorizzare informazioni, bensì per processare transazioni senza intermemdiari e per eliminare il problema del *double-spending*; ci troviamo quindi di fronte ad un importante limite della tecnologia per quanto riguarda la sicurezza - che non rappresenta un elemento critico - lasciandoci quindi intuire che questo progetto non si presta allo specifico caso di votazioni elettroniche elettorali.
L'unico aspetto che un'applicazione decentralizzata ed una web app hanno in comune è il front-end. Come è facilmente immaginabile infatti, la differenza sostanziale risiede nel back-end, dove al posto dei classici strumenti e linguaggi di programmazione che si utilizzerebbero di solito, come *PHP* che opera su un database, viene utilizzato un contratto scritto in linguaggio *Solidity* che opera su una Blockchain Ethereum. Verrà abbondantemente sfruttata libreria *Web3*, la quale permette l'interfacciamento con i contratti Solidity e l'interazione con i nodi della rete Ethereum.
Decidere di sviluppare un'applicazione decentralizzata invece di una classica web app permette di godere di svariati vantaggi e solo qualche svantaggio.
A causa della natura della tecnologia, l'utilizzo delle DApps spesso costringe l'utente al pagamento di una piccola tassa, di solito in *cryptovaluta*, che permette di utilizzare il servizio offerto. Il motivo di ciò è che le DApps vengono eseguite su migliaia di nodi in giro per il mondo, rendendo la tassazione e necessaria per mantenere la rete operativa ed a tenere lontani gli spammers.
Un grande vantaggio risiede nella persistenza delle DApps che, di fatto, non

possono essere oscurate da nessun'entità; una volta completato il deploy di un'applicazione decentralizzata, quest'ultima rimarrà sulla Blockchain per sempre. L'altra faccia della medaglia è che, se usato per scopi illegali, rappresenta anche un grande svantaggio in quanto le autorità non sarebbero in grado di fermare l'erogazione del servizio.
Un'altra corposa differenza risiede nella possibilità di poter memorizzare informazioni all'interno delle DApps; è ad esempio possibile creare un'applicazione decentralizzata che si occupi di accumulare degli *Ether* - cryptovaluta della rete Ethereum - e distribuirli casualmente ad alcuni degli utenti finali una volta raggiunto una pre-definita somma di cryptovaluta. Fondamentale è la differenza con una simile web app che di fatto non memorizza alcun dato al suo interno, forzando l'intervento di un intermediario per portare a termine il compito.
La rete Ethereum può, *attualmente*, gestire circa 15 transazioni al secondo, cosa che implica forti limitazioni in scalabilità.
Nel prossimo futuro è però previsto un radicale aumento delle performance a circa 10000 transazioni al secondo, che rappresenta un incremento di più del 65000%.

La tesi è organizzata come segue:

Capitolo 1 Questo capitolo, di carattere più teorico-matematico, è necessario alla comprensione dei meccanismi sottostanti al sistema oggetto di studio. Si partirà dalle proprietà base della crittografia moderna, fino ad arrivare ad analizzare gli aspetti teorici della tecnologia Blockchain come gli Smart Contracts.

Capitolo 2 In questo capitolo si analizzeranno gli aspetti di decentralizzazione e consenso della tecnologia, con una panoramica sui meccanismi di mining.

Capitolo 3 All'interno di questo capitolo si studierà il linguaggio di programmazione Solidity, soffermandosi sugli aspetti più significativi, utilizzato per programmare Smart Contracts su Blockchain Ethereum.

Capitolo 4 In questo capitolo verranno esposti tutti gli step di analisi dei requisiti ed implementazione del progetto di tesi, fino ad arrivare alle fasi di testing e deploy sull'applicazione realizzata.

Conclusioni e sviluppi futuri In quest'ultima parte della tesi, viene analizzato il lavoro svolto nella sua interezza, andando ad analizzare possibili sviluppi futuri e la risoluzione di alcune problematiche nate durante le fasi di analisi e sviluppo.

Capitolo 1

Stato dell'arte

1.1 Crittografia

La crittografia è un campo di ricerca accademica che utilizza molte ed avanzate tecniche matematiche che sono notoriamente conosciute per essere particolarmente complesse e difficili da capire a fondo.
In particolare, in informatica, col termine crittografia ci si riferisce alla tecnica di comunicazione derivata da concetti matematici e da un set di algoritmi per trasformare il messaggio in *capsule* difficili da decifrare. Questi algoritmi deterministici sono usati per generare chiavi crittografiche e firme digitali, con il fine di assicurare la privacy delle suddette comunicazioni.
La crittografia è strettamente legata alle discipline di *cryptology* - studio di codici - e *cryptanalisys* - decryption e analisi di codici o cyphers - che includono tecniche come microdots e fusione di parole e immagini.
Per assicurare la totale trasparenza, veridicità e sicurezza dei dati presenti sul cosiddetto *ledger*, la Blockchain utilizza una porzione molto ridotta di alcune famose tecniche crittografiche, che andremo ora ad analizzare.

1.1.1 Funzioni Hash

Una funzione hash è una funzione matematica che gode delle tre seguenti proprietà:

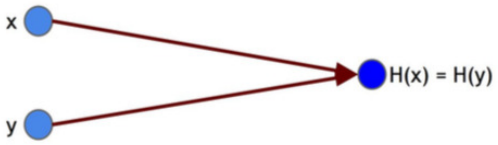


Figura 1.1: Rappresentazione grafica di Collision Resistance. x e y sono valori differenti, tuttavia quando vengono usati come input per la funzione hash, producono lo stesso valore di output.

1. L'input può essere una stringa di qualsiasi dimensione;
2. Restituisce un output di dimensione fissa;
3. Tempo computazionale O(n).

L'insieme di queste proprietà definisce una funzione hash che risulta essere molto utile per la costruzione di una hash table.
Una funzione hash, per essere crittograficamente sicura, deve rispettare le ulteriori seguenti proprietà:

1. Collision-resistance;
2. Hiding;
3. Puzzle-friendliness.

Collision-resistance Una funzione hash H è detta *resistente alle collisioni* se risulta impossibile trovare due valori, x e y, tale che

$$x \neq y$$

ma

$$H(x) = H(y).$$

Dato che il numero di valori di input supera il numero di valori di output, sappiamo con assoluta certezza che ci sarà almeno un valore di output alla quale la funzione hash associa più di un valore di input.

Una possibile applicazione della proprietà di collision-resistance è: se sappiamo che due valori di input x e y applicati su una funzione H sono diversi, allora possiamo affermare che i valori *H(x)* e *H(y)* sono diversi. Se x e y fossero stati diversi ma i loro valori di hash uguali, allora H non avrebbe soddisfatto la proprietà di collision-resistance.

Hiding La seconda proprietà che vogliamo che la nostra funzione hash rispetti è l'*hiding*. Questa proprietà afferma che: dato l'output della funzione

$$y = H(x)$$

non esiste nessun modo di risalire all'input. Una funzione hash H rispetta questa proprietà se avendo un valore segreto v scelto da una distribuzione di probabilità che ha un'alta entropia minima, tale che dato H (v concatenato ad x) è impossibile risalire ad x.
In teoria dell'informazione, l'entropia minima stima quando sia predicibile un certo output.

Puzzle-friendliness La terza proprietà che una funzione hash H deve rispettare per essere pienamente sicura, è la puzzle-friendliness.
Una funzione hash H è detta puzzle-friendly se per ogni valore y di output composto da n-bit, se k è scelto da una distrubuzione con elevata entropia minima, allora è impossibile trovare il valore x tale che

$$H(k) = y$$

con k concatenato ad x, in un tempo inferiore a 2^n.
Sono molte le funzioni hash disponibili in letteratura, ma una molto diffusa nelle applicazioni che riguardano la tecnologia blockchain, è *SHA256*. Quando possiamo costruire una funzione hash che lavora su input di lunghezza prefissata, tramite la trasformazione di *Merkle-Damgard*, è possibile convertirla in una funzione hash che lavora su input di lunghezza arbitraria. SHA-256 usa una funzione che prende 768 bit di input e produce 256 bit di output, la lunghezza del blocco è di 512 bit.

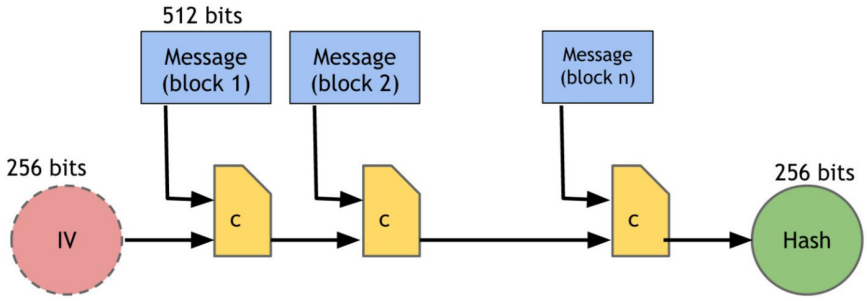


Figura 1.2: SHA-256 usa la trasformazione di Merkle-Damgard per convertire una funzione collision-resistant di lunghezza fissa in una funzione hash che accetta input di lunghezza arbitraria. Viene effettuato una sorta di padding per uniformare la lunghezza in multipli di 512 bit.

Figura 1.3: L'hash pointer punta ad un'area nella quale vengono memorizzate delle informazioni insieme al valore crittografico dei dati in un certo istante temporale.

1.1.2 Hash pointer e strutture dati

Un *hash pointer* è una funzione crittografica che punta ad un blocco di dati. A differenza di una *linked list*, che permette di identificare il blocco successivo, un hash pointer rende possibile verificare che il precedente blocco di dati non sia stato compromesso.

1.1.3 Firme digitali

Le firme digitale sono di fondamentale importanza quando si vuole costruire dei blocchi in ambito blockchain. Una firma digitale è la versione digitale della

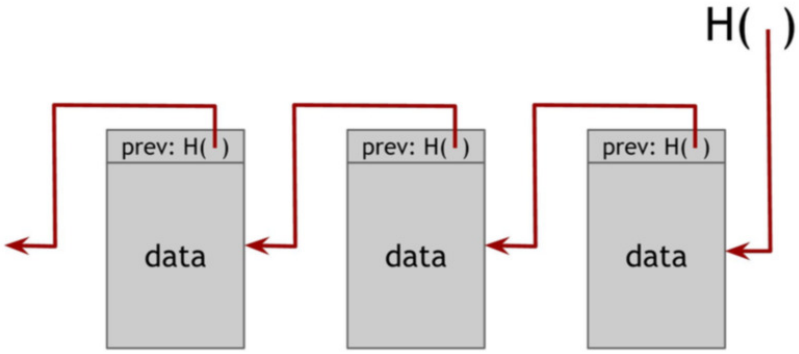


Figura 1.4: Una *blockchain* è una linked list composta da hash pointers.

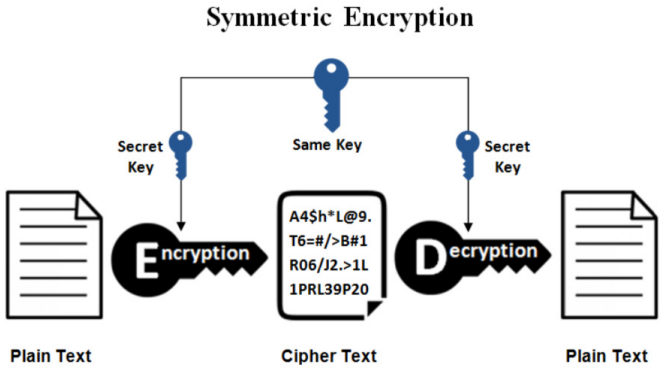


Figura 1.5: Encryption/decryption tramite chiave privata/pubblica.

firma per iscritto.
I concetti fondamentali sono due: una certa firma può essere apposta solo dal legittimo proprietario e contemporaneamente essere verificata da tutti i componenti della rete. In seconda battuta, si deve poter verificare che la firma apposta sia legata solo ad un particolare documento in modo che la firma non possa essere apposta su documenti differenti.

Digital Signature Scheme

Il Digital Signature Scheme si compone dei tre seguenti algoritmi:

1. `(sk, pk) := generateKeys(keySize)` prende in input la dimensione di una chiave e genera una coppia di chiavi. La chiave segreta *sk* viene mantenuta privata in quanto è l'effettiva firma che viene utilizzata per apporre le firme. La chiave pubblica *pk* viene "inviata" a tutti che hanno bisogno di effettuare la verifica di veridicità della chiave privata;

2. `sig := sign(sk, message)` è il metodo che si occupa di apporre la firma. Prende in input una chiave privata ed un messaggio e restituisce una firma per il messaggio e la chiave fornitegli;

3. `isValid := verify(pk, message, sig)` prende in input un messaggio, una chiave pubblica ed una firma. Restituisce **TRUE** o **FALSE** a seconda che la firma sia risultata valida o meno, considerate le condizioni.

   Una chiave pubblica, di default, sembrerà un numero totalmente casuale e nessuno sarà in grado di risalire ad una chiave privata data una certa chiave pubblica.

La chiavi utilizzate all'interno della tecnologia Blockchain sono caratterizzate dalle seguenti lunghezze:

1. Chiave privata: 256 bit;
2. Chiave pubblica (non compressa): 512 bit;
3. Chiave pubblica (compressa): 257 bit;
4. Messaggio da firmare: 256 bit;
5. Firma: 512 bit.

1.1.4 Zero-knowledge proof

La zero-knowledge-proof (ZKP) permette ad un *prover* di assicurare ad un *verifier* che si abbia conoscenza di un segreto o di uno statement senza rivelare il documento in sè.
Un protocollo, per qualificarsi come zero-knowledge proof, deve rispettare le seguenti proprietà:

- Completeness: se lo statement è veritiero, un verifier *onesto* sarà convinto da un prover onesto;
- Soundness: se lo statement è falso, nessun prover non onesto può convincere un verifier del contrario;
- Zero-knowledge: se lo statement è veritiero, nessun verifier non onesto viene a conoscenza di qualcosa se non del fatto che lo statement in questione sia appunto veritiero.

Le ZKP sono di carattere probabilistico, dato che esiste sempre una - supper molto ridotta - possibilità che sia possibile imbrogliare un verifier onesto. Il verifier può essere convinto che, con un probabilità del 99.9%, il prover sarà in grado di portare a termine la verifica. Questa tecnica non è però perfetta, un terzo soggetto infatti potrebbe contestare che le due parti abbiano colluso e falsato l'intero processo.

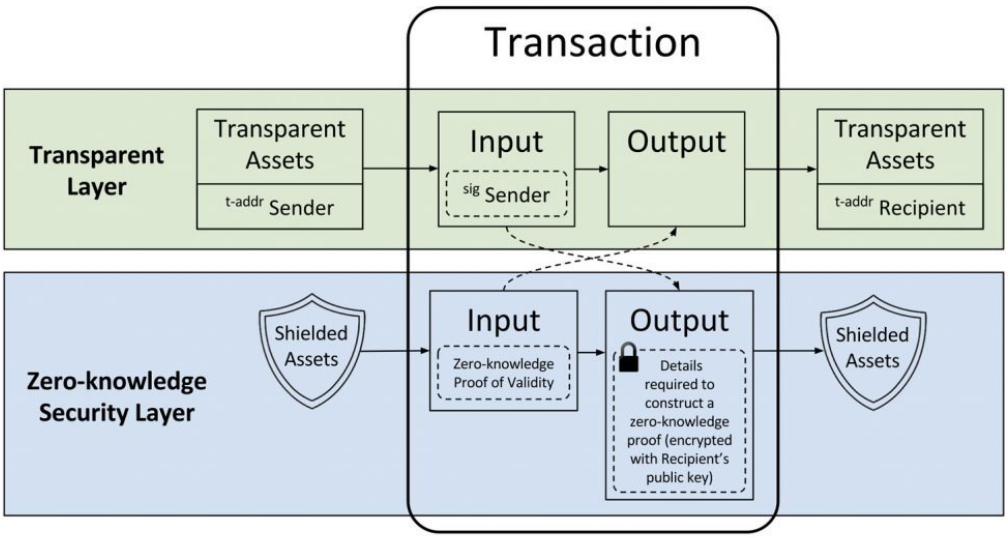


Figura 1.6: Meccanismo di funzionamento del protocollo a conoscenza zero.

1.2 Meccanismi di funzionamento

1.2.1 I blocchi

La Blockchain non è altro che una combinazione di due differenti strutture dati hash, di cui la prima è una catena di blocchi hash. Ogni blocco ha un header, un hash pointer che punta a delle transazioni ed un altro hash pointer che punta al precedente blocco della catena.
La seconda struttura è un albero, in particolare un **Merkle-tree**, di tutte le transazioni contenute all'interno del blocco che ci permette di risalire ad una sorta di "riassunto" delle transazioni in modo molto efficiente. Per provare che una transazione sia contenuta all'interno di un blocco, possiamo fornire un percorso all'interno dell'albero (la cui altezza è logaritmica).

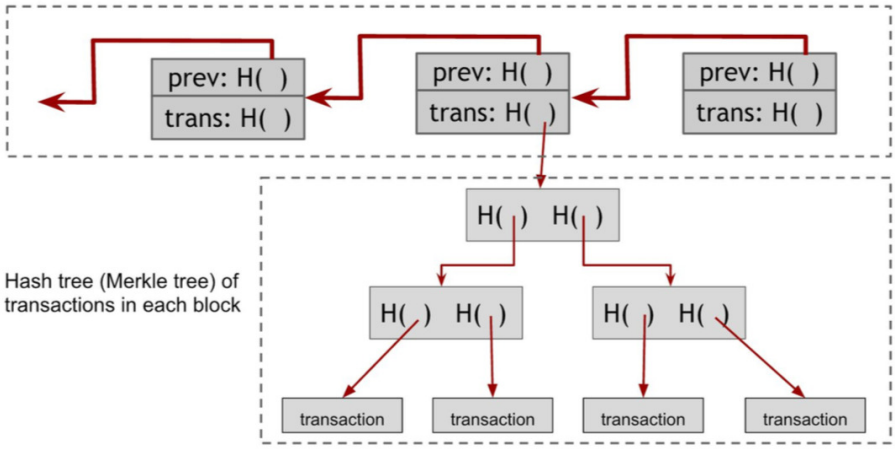


Figura 1.7: La parte superiore della figura rappresenta la catena di blocchi hash, mentre la seconda è il Merkle-tree, struttura interna di ogni blocco.

1.2.2 Transazioni

La transazioni definiscono i cambiamenti di stato all'interno della Blockchain che devono ancora essere seguiti. Esse sono costituite da sei diversi attributi:

- txID - identificatore progressivo della transazione;
- issuerID - l'indirizzo dell'entità emittente della transazione;
- Signature - la firma crittografica della transazione;
- recipientID - l'indirizzo dell'entità ricevente della transazione;
- value - numero intero indicante l'ammontare della cryptovaluta inviata;
- data - informazioni aggiuntive arbitrarie.

Inizialmente, un *issuer* compila una transazione, la firma con la sua chiave privata e la invia in modalità broadcast all'intera rete P2P. I cosiddetti *peers* prenderanno poi in considerazione sono le transazioni che verranno considerate valide. Per risultare tale, quest'ultima dev'essere ben formata, la firma deve combaciare con l'indirizzo dell'issuer e deve certificare l'integrità della

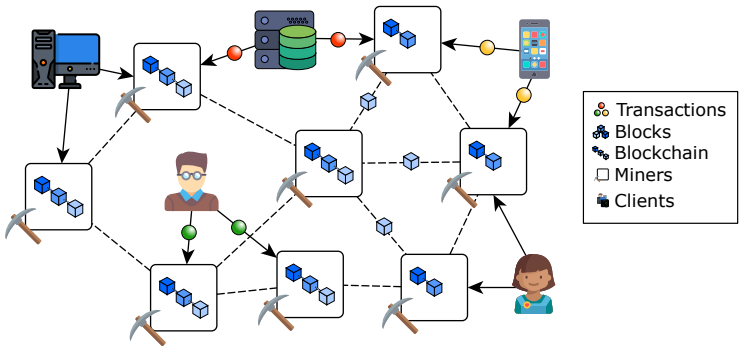


Figura 1.8: Rappresentazione grafica della tipologia di rete sottostante alla tecnologia Blockchain. [11]

transazione stessa. E' bene tenere a mente che nonostante una transazione sia considerata valida, non c'è alcuna certezza che essa sarà eseguita.

1.2.3 La rete

La blockchain è, di fatto, una rete peer-to-peer all'interno della quale tutti i nodi sono *uguali*. Non esiste una gerarchia e non esistono nodi con privilegi maggiori di altri. Si basa su tecnologia TCP ed è caratterizzata da una topologia *random* in cui ogni nodo comunica con altri nodi casuali. Nuovi nodi possono aggiungersi alla rete in qualsiasi momento. Difatti, scaricando un qualsiasi client Blockchain su un qualsiasi computer, quest'ultimo diventa un nodo a tutti gli effetti.
Questo rende la rete particolarmente *dinamica*, a seguito delle continue attività di entrata e uscita dei nodi. Non esiste un modo particolare per abbandonare la rete, ma se un nodo risulta inattivo per più di tre ore, i nodi cominciano a "dimenticarlo".
Quando un nodo entra a far parte della rete, si comunica con un nodo che già si conosce inizialmente, detto *seed node*; esistono svariati modi di cercare un seed node. Il nodo entrante invia un messaggio in cui chiede che gli sia inviata la lista di indirizzi dei nodi conosciuti presenti sulla rete, successivamente il nodo sceglie a quale connettersi.

Gossip protocol Per pubblicare una transazione, l'intera rete dovrà essere in grado di vederla. Questo viene reso possibile attraverso un classico algoritmo di flooding, anche conosciuto come *Gossip Protocol*.
Se A vuole inviare dei soldi a B, il client di A crea e il nodo invia la transazione a tutti i nodi a cui è collegato e successivamente ogni nodo esegue una serie di verifiche per determinare se accettare o meno la transazione. Se il controllo viene superato, il nodo propaga, a turno, tutte le informazioni verso gli altri nodi.
I nodi che ricevono la transazione la inseriscono in una cosiddetta *transactions pool*, che contiene le transazioni ricevute ma non ancora inserite in un blocco della catena. Se un nodo invece dovesse ricevere una transazione che si trova già all'interno della transactions pool, non la propaga oltre. Questo garantisce che il protocollo di gossip termini e che le transazioni non rimangano bloccate in un loop nella rete indefinitamente.

Zero-knowledge proof in ambito blockchain

La tecnologia blockchain non può essere considerata del tutto anonima, infatti tutte le transazioni sono pubblicamente disponibili a chiunque voglia vederle; ciò è dovuto intrinsecamente alla natura di tale tecnologia.
Questo però rappresenta un problema per alcune aree di applicazione della tecnologia, come le industrie della finanza e della medicina, dove è fondamentale mantenere la segretezza dei dati.
Nel particolare caso di *Ethereum*, quest'ultima viene raggiunta mediante l'utilizzo di tecniche di *zero-knowledge proof* attraverso l'utilizzo di *zkSnark*, dove Snark sta per Succinct Non-Interactive Arguments of Knowledge.

- Succint: la dimensione dei messaggi è molto ridotta se comparata alla lunghezza della computazione vera e propria;

- Non-interactive: esiste poca o nessuna interazione. Esiste di solito una fase di setup e successivamente un singolo messaggio dal prover al verifier;

- Arguments: il verifier è protetto soltalto da prover con potenza computazionale limitata. I prover con sufficiente potenza a disposizione possono creare prove (o appunto argomenti) riguardanti falsi statement. Questo fenomeno è anche noto come computational soundness;

- Knowledge: il prover non ha possibilità di costruire prove/argomenti senza essere a conoscenza del cosiddetto *witness*.

I protocolli che implementano la zero-knowledge proof, quindi, sono in grado di effettuare transazioni su Blockchain conservando la segretezza necessaria e richiesta da specifici ambiti applicativi.

1.2.4 Smart Contracts

Proposto per la prima volta nel 1996 da Nick Szabo [3], uno Smart Contract (SC) è un protocollo pensato per facilitare, verificare e far rispettare la negoziazione o l'esecuzione di un contratto.
Se volessimo dare una definizione più formale del concetto di Smart Contract, questa sarebbe: *Stateful, reactive, user-defined, immutable, and deterministic processes executing some arbitrary computation on the blockchain, i.e., while being replicated over the blockchain network.* Approfondendo, possiamo ulteriormente approfondire la definizione:

- Stateful - gli SC incapsulano al loro interno il proprio stato, come gli oggetti in O.O.P.;

- Reactive - possono essere chiamati facendo uso di *invocation transitions*;

- User-defined - gli utenti possono fare deploy degli SC implementando una logica arbitraria mediante *deployment transitions*;

- Immutable - il loro codice sorgente non può essere alterato una volta effettuato il deployment;

- Arbitrary - sono espressi medianti una logica Turing-complete;

- Replicated - possiamo intendere la Blockchain come un interprete *replicato*, che utilizza protocolli di consenso per sincronizzare le varie repliche degli SC.

Sostanzialmente, uno Smart Contract è un programma che può automaticamente eseguire i termini di un contratto. Quando si verifica una condizione *pre-configurata* in uno Smart Contract, le parti coinvolte nell'accordo contrattuale eseguiranno/riceveranno ciò che viene specificato nell'oggetto e nelle clausole del contratto stesso. Da ciò si intuisce che, sfruttando contratti di questo tipo, si ha una garanzia di gran lunga superiore che l'oggetto del contratto venga effettivamente rispettato da chi di dovere, rispetto a un contratto tradizionale.
Uno Smart Contract risiede sulla Blockchain ed è identificabile da un indirizzo univoco, definito da un set di variabili di stato e funzioni. Esso viene fatto valere grazie al protocollo Blockchain sottostante.
Un ulteriore caratteristica interessante è che gli Smart Contracts, oltre che dalle macchine, possono essere capiti anche dagli esseri umani; questo è un enorme vantaggio poichè, se sia gli umani che le macchine sono in grado di interpretare il contenuto di questi contratti, è molto più probabile che vengano valutati positivamente in contesti legali e sociali.
Uno Smart Contract può essere stateless, ciò vuol dire che non conserva nessuno stato al suo interno. Un contratto stateful, invece, può essere eseguito all'interno di costrutti più potenti ed elaborati come i loops, che permettono di mantenere informazioni interne sullo stato.
Dopo vari tentativi di integrare tutti questi principi all'interno della tecnologiaBblockchain, è emersa quello che oggi conosciamo come *Ethereum*, una piattaforma decentralizzata i cui programmi interni sono proprio gli Smart Contracts, scrivibili in un linguaggio che è *Turing-complete*.

Problemi degli Smart Contracts Gli Smart Contracts rappresentano un mezzo mediante il quale diventa possibile esploitare in maniera efficiente tutte le potenzialità della tecnologia Blockchain; ma ciò porta con sé alcune problematiche non di poco conto. In questa sezione studieremo alcuni dei più

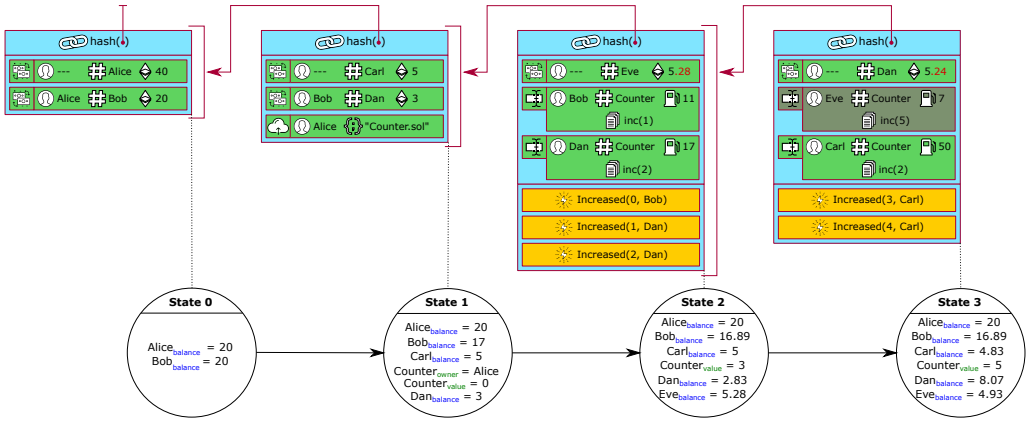


Figura 1.9: Esempio di meccanismo di funzionamento di uno Smart Contract. [11]

importanti problemi riguardo lo sviluppo e implementazione di SC.
Alcuni dei problemi più importanti relativi allo sviluppo ed implementazione di SC sono la poca garanzia di privacy, la scarsa randomness e comunicazione inter-contratto, impossibilità di fixare i bugs, mancanza di proactivness, disembodiement, mancanza di tecniche di concorrenza e granularità di costi.

Segretezza dei dati negli Smart Contracts

I campi all'interno dei contratti possono essere pubblici o privati; dichiarare un campo come privato tuttavia non garantisce la segretezza dei dati contenuti all'interno di esso. Questo accade perché per settare il valore di un campo, gli utenti devono obbligatoriamente inviare una transazione ai *miners* i quali la pubblicheranno sulla Blockchain. Dato però che la Blockchain è pubblica per sua natura, chiunque può ispezionare il contenuto della suddetta transazione e risalire al valore del campo.
Alcuni contratti sviluppati per determinati tipi di applicazione richiedono che i campi siano mantenuti segreti per un certo intervallo temporale. Questo permette di mantenere il campo segreto fino a che un certo - precedentemente specificato - evento si verifichi.
Nel corso degli anni sono stati studiati numerosi metodi per cercare di mante-

nere la segretezza all'interno dei contratti, uno degli approcci più interessanti è senza dubbio il *commit-reveal pattern*. In questa tecnica, l'utente invia inizialmente un hash di informazioni segrete e, quando tutti gli altri utenti hanno inviato anche le loro, il partecipante svela i propri dati che possono essere quindi verificati. Il commit-reveal patten non si presta per tutti i tipi di applicazione, in quanto aggiunge molta complessità agli utenti, ma rappresenta un punto di partenza per future implementazioni.

Randomness Per generare un numero causale, i contratti spesso utilizzano l'hash o il timestamp del blocco in uno specifico istante temporale futuro, visto che il valore sarà lo stesso per tutti i componenti della rete. Dato però che i miner possono controllare i contenuti e l'ordine dei blocchi, un miner potrebbe condizionare il processo di generazione di un numero random. Una soluzione proposta in letteratura è l'implementazione di *timed-commitment protocols*. Tutte le parti inviano il loro commitment insieme a un deposito; successivamente i partecipanti "aprono" i loro commitment. Il numero random è poi calcolato dai valori segreti contenuti all'interno dei commitment. Se un partecipante non rileva il proprio commitment, perde il deposito inviato.
Questo tipo di meccanismo è molto utilizzato in alcuni ambiti applicativi come aste o firma degli Smart Contract e garantisce inoltre un livello di sicurezza molto elevato, in tutti i tipi di applicazioni, contro attacchi di tipo parallelo.

Smart contract inter-communication

Nel caso di Ethereum, gli SC sono di fatto degli oggetti che comunicano attraverso delle chiamate di funzione sincrone. Lo SC chiamato è referenziato dai chiamanti utilizzando i loro indirizzi:

- Il flusso di controllo originato da un utente può attraversare più di uno SC;
- Il chiamante aspetta il chiamato;
- La *unattended re-entrancy* è difficilmente evitabile;

- Possibilità di esiti non previsti - e non desiderati - abbastanza consistente.

Impossibilità di fixare i bugs Come già menzionato, gli SC ed il loro codice sono immutabili. Ciò implica che non possono essere fixati, aggiornati o rimpiazzati richiedendo quindi un nuovo deployment del contratto stesso nel caso in cui si presentasse un bug o la necessità di implementare un servizio aggiuntivo. I contratti "abbandonati" restano però sempre sulla Blockchain, il che vuol dire portare ad un occasionale spreco di risorse di mining.

Mancanza di proactiveness, disembodiment e assenza di concorrenza Gli SC sono entità computazionali di tipo reattivo, in quanto hanno costantemente bisogno di "prendere in prestito" il flusso di controllo di un utente x e sono *time-aware* ma non reattive al tempo, non possono inoltre programmare o posticipare computazioni.
Le computazioni sono distribuite per natura e le transazioni sono strettamente sequenziali. Questo, in alcuni casi, potrebbe rappresentare un tipo di approccio dispendioso e superfluo in quanto le computazioni indipendenti non possono essere eseguite in maniera concorrente e quindi si presenta il bisogno di ripeterle su scala globale. Inoltre, le computazioni che risultino essere molto pesanti ed esigenti di risorse non possono essere separate in parti distinte per essere eseguite in maniera concorrente.

Smart Property

La *Smart Property* è un concetto che ridefinisce il controllo di una proprietà o di un bene attraverso la tecnologia Blockchain e mediante l'utilizzo degli Smart Contracts. La proprietà può essere fisica come una casa, un'auto, un computer oppure non fisica, come ad esempio le azioni di una società.
Rendere una proprietà "smart" tendenzialmente vuol dire che può essere scambiata senza aver bisogno di coinvolgere terze parti. In questo modo si riduce di molto il rischio di frodi e la necessità di dover pagare commissioni di mediazione.
Un banale esempio: richiedere un prestito ad un'istituzione o ad un privato su

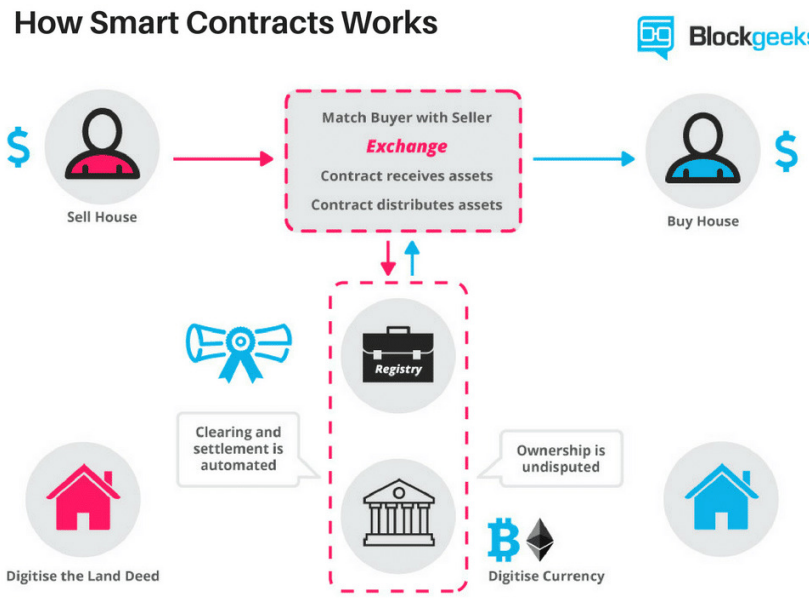


Figura 1.10: Smart property. [11]

internet mediante Blockchain, prendendo una smart property come garanzia. Questo renderebbe l'accesso al credito molto più accessibile e concorrenziale.

Capitolo 2

Decentralizzazione e Consenso

La decentralizzazione è uno dei concetti fondamentali dietro l'ideologia della Blockchain; quasi nessun sistema al mondo, infatti, può vantare di essere *totalmente* decentralizzato.
Ad esempio, l'email è quasi del tutto decentralizzata ma basata su SMTP, che è un protocollo standardizzato.

Esistono tre diversi tipi di decentralizzazione:

- Decentralizzazione architetturale - *"di quanti computer fisici è composto un sistema?"*;
- Decentralizzazione politica - *"Quante organizzazioni/individui controllano i computer di cui è fatto il sistema?"*;
- Decentralizzazione logica - *"Quale topologia hanno le interfacce e le strutture dati che compongono il sistema?"*.

Una qualsiasi Blockchain è sia politicamente che architetturalmente decentralizzata, in quanto nessuno la controlla e non esiste un punto centrale di vulnerabilità. Al tempo stesso è anche logicamente centralizzata in quanto non esiste uno stato del sistema comunemente accordato; il sistema si comporta quindi come un singolo computer.

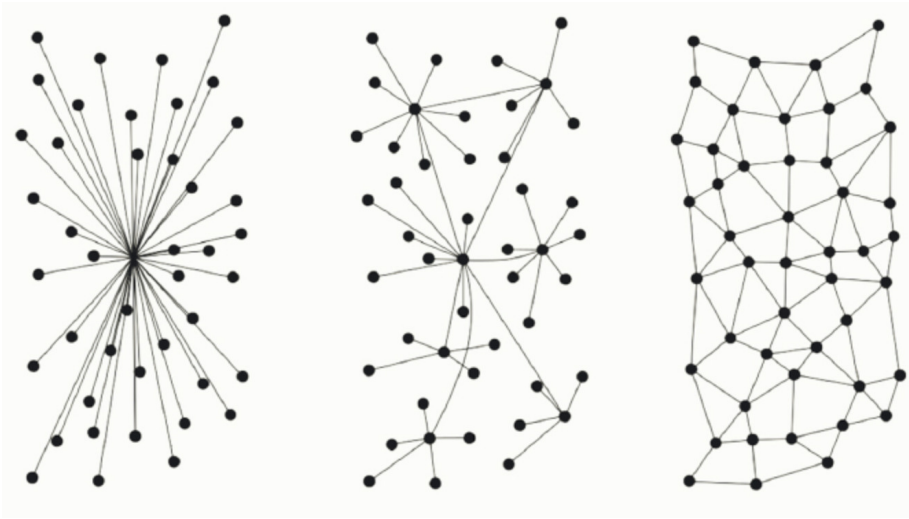


Figura 2.1: Centralizzato (A) - Decentralizzato (B) - Distribuito (C)

2.1 Consenso

Il consenso è un componente fondamentale per l'utilizzo di una tecnologia Blockchain, a prescindere da quale sia l'area di applicazione.
Il consenso distribuito ha svariate applicazioni ed è stato studiato per decenni. Quando, ad esempio in una grande azienda, si ha a che fare con centinaia di migliaia di server sparsi su tutto il pianeta - che insieme formano un gigantesco database distribuito - ogni informazione dev'essere memorizzata su svariati nodi della rete, che devono essere quindi sincronizzati sullo stato del sistema. Se, invece, avessimo un sistema basato sul consenso distribuito, potremmo costruire un grande sistema di memorizzazione *key-value*, che mappa chiavi arbitrarie a valori arbitrari. Un sistema di questo tipo potrebbe, ad esempio, permettere lo sviluppo di un DNS distribuito, dato che si tratterebbe soltanto di un'associazione indirizzo-nome.

2.1.1 Distributed Consensus Protocol

Dando una definizione tecnica, il Distributed Consensus Protocol contiene n nodi, ognuno avente un certo valore di input.
Un protocollo di consenso distribuito deve rispettare le seguenti proprietà:

1. Deve terminare con tutti i nodi *onesti* in accordo sul valore;

2. Il valore deve essere generato da un nodo onesto.

A questo punto ci si potrebbe chiedere, effettivamente, su cosa tutti i nodi debbano raggiungere un consenso. Dato che n-nodi stanno trasmettendo transazioni in modalità broadcast sulla rete, essi devono accordarsi su quali transazioni siano state trasmesse ed in quale ordine. Il risultato di questa operazione sarà un grande, distribuito registro globale.

2.2 Mining e meccanismi di compensazione

Una domanda che a questo punto sorge spontanea è: *come determiniamo se un nodo è onesto o meno?*
O, in altre parole, qual è il processo da seguire per stabilire se una specifica transazione sia lecita?
Nel momento in cui, però, si fornisce un incentivo di carattere economico ai nodi in cambio di onestà, il problema non si pone.

2.2.1 Block reward

Viene introdotto quindi il meccanismo di *block reward*. Prendendo in analisi il caso *Bitcoin*, il nodo che crea un blocco contenente n-transazioni, ha la possibilità di aggiungere una speciale ulteriore transazione al blocco stesso. Quest'ultima consiste in un'operazione di generazione di cripto-moneta, il nodo può inoltre scegliere a quale indirizzo inviare queste frazioni di Bitcoin. Si può pensare a questo meccanismo come una sorta di compenso pagato al nodo in cambio del servizio di creazione del blocco della catena. L'unico modo però in cui questo nodo riceverà la ricompensa, è se il blocco creato viene approvato dal restante insieme di nodi, come avviene per le "normali" transazioni; questo è il punto chiave dietro i meccanismi di compensazione. In questo modo quindi si 'convincono" tutti i nodi a comportarsi in maniera onesta, dato che in caso contrario non riceverebbero nessuna ricompensa.

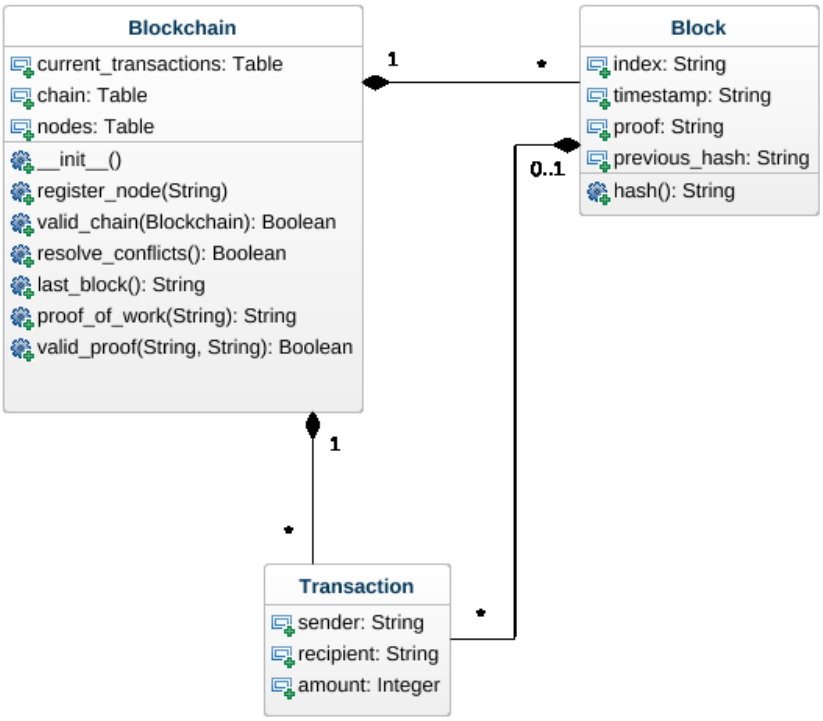


Figura 2.2: Diagrammi delle classi rappresentante il funzionamento di base di una Blockchain.

Questo è un furbo e semplice meccanismo che però introduce ben tre nuovi problemi:

1. Scegliere un nodo in maniera completamente randomica non è scontato;
2. Il sistema, dati gli incentivi, potrebbe diventare instabile in quanto tutti vorrebbero ottenere le stesse ricompense;
3. Un nodo, a seguito del problema precedente, potrebbe creare una serie di nodi fratelli per cercare di sovvertire il processo di consenso.

2.3 Proof-of-Work

Tutti questi problemi sono risolvibili attraverso il meccanismo di *Proof-of-Work* (PoW). Ciò consiste nell'approssimazione del processo di selezione di un nodo randomico, selezionando invece il nodo in base alla risoluzione di un complesso calcolo matematico; per cui in base alle risorse computazionali che un nodo possiede. Il meccanismo è inoltre dotato di una caratteristica di correzione automatica della difficoltà di mining di tipo *self-adaptive* molto interessante. Il processo di selezione, contrariamente alla risoluzione del calcolo, avviene mediante l'esecuzione di un singolo hash, quindi richiede pochissime risorse. La risoluzione di questo puzzle matematico, chiamato *mining*, non è triviale e la complessità del problema può essere modificata in modo che, ad esempio, un nodo necessiti di qualche minuto per generare il blocco.
Il primo nodo a risolvere il problema, invia un messaggio broadcast al resto della rete.
Per referenziare un singolo blocco della catena, l'header viene hashato due volte tramite la funzione SHA256; l'intero restituito sarà compreso nell'intervallo [0, 2^{256} - 1]. La *block reference* viene utilizzata dal protocollo proof-of-work per permettere ad ogni blocco di essere convalidato. Essa non dove superare una certa soglia, definita dalla seguente funzione:

$$hash(B) \leq M/D$$

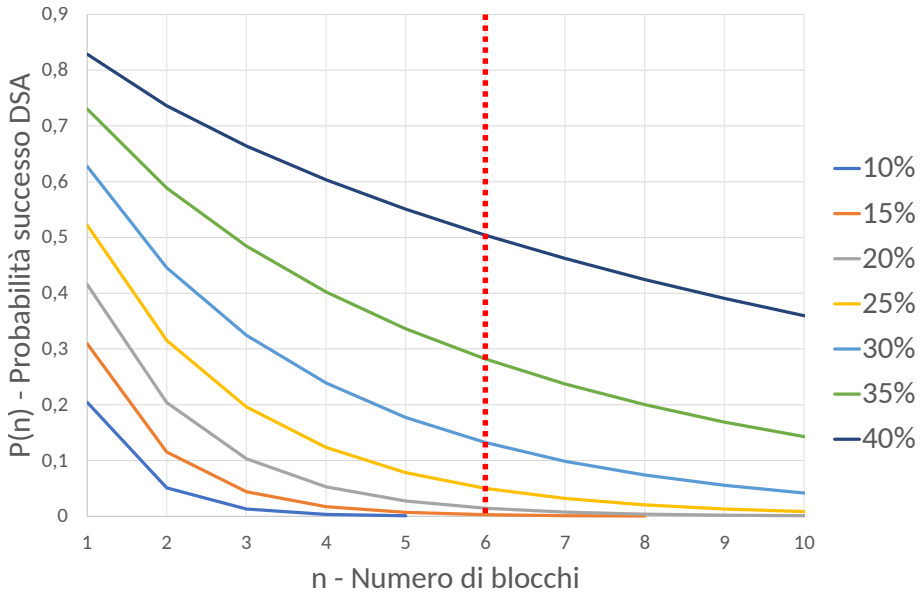


Figura 2.3: Sicurezza del protocollo Proof-of-Work. [11]

dove D $\in$ [1, M] è la *difficoltà target*. L'unico modo attualmente conosciuto per trovare B è iterare attraverso tutte le possibili variabili nell'header del blocco in maniera ripetuta.
Più è alto D, più iterazioni si renderanno necessarie per trovare un blocco valido; infatti il numero stimato di operazioni necessarie è esattamente D.
Il periodo T(r), per un miner capace di eseguire r operazioni al secondo per trovare un blocco valido, è distribuito esponenzialmente attraverso il rapporto r/D:

$$P[T(r) \leq t] = 1 - exp(-rt/D)$$

2.3.1 Mining pools

Se una persona singola decidesse di effettuare del mining con il proprio computer domestico, non avrebbe alcuna chance di risolvere il problema matematico, necessario per ricevere la ricompensa, prima dei tantissimi supercomputer presenti nella rete.
La sua opzione migliore quindi sarebbe quella di unirsi ad una *mining pool*, un gruppo di *miners* che formano una *pool* nel tentativo di, unendo le proprie

risorse, riuscire a competere con i più potenti nodi della rete. Il pool manager riceverà l'eventuale ricompensa del blocco minato e lo distribuirà a tutti i miners partecipanti alla pool in base a quanta potenza di calcolo hanno dedicato per la risoluzione del puzzle matematico.
Spesso le mining pools offrono una *variable share difficulty function* il cui compito è quello di assegnare il cosiddetto *share target* ai miners, in funzione della potenza computazionale che sono in grado di offrire. Questa tecnica permette di assicurare che non vengano assegnati dei compiti con difficoltà troppo elevata rispetto alla potenza di calcolo che si è in grado di offrire e al tempo stesso che la difficoltà non sia troppo bassa, in quanto questo potrebbe ridurre considerevolmente l'overhead di una mining pool.
Quest'ultima pagherà una ricompensa ai miners in relazione al numero di *shares* inviate ed accettate. Una delle policy più comuni adottate dalle mining pools è la *Pay Per Share* (PPS), la quale richiede che sia pagata una commissione alla pool.
Dato che il calcolo finale per il pagamento di share è dipendente dall'effettiva attività di mining della pool, risulta complesso stimare una somma esatta di shares da pagare. Tuttavia, questa problematica viene risolta mediante l'adozione di un protocollo di conversione Hashrate-to-Bitcoin, conosciuto come *Hardening Stratum.*
Stratum è un *Clear Text Communication Protocol* che si interpone tra una mining pools ed i suoi miners. Si basa su TCP/IP e utilizza il formato JSON-RPC. Questo protocollo permette quindi ai miners di trovare, in modo efficiente, lavori da svolgere in relazione alla propria potenza di calcolo disponibile.
Nel corso degli anni, sono nate molte controversie riguardo l'impatto ambientale causato dal Proof-of-Work in quanto, specialmente nei paesi asiatici dove il costo dell'elettricità è molto basso, si sono costruite delle vere e proprie *mining farm*, contenenti centinaia di migliaia di GPU atte a risolvere calcoli matematici complessi, che consumano una quantità smisurata di energia elettrica; si stima infatti che l'intera Bitcoin Network consumi più energia di quella consumata da 159 paesi, *messi insieme* [15].

Figura 2.4: Impatto ambientale del protocollo *Proof-of-Work*.

Da quì nasce il protocollo *Proof-of-Stake* (PoS), che consiste nella scelta del nodo creatore del blocco in modo deterministico, in base alla quantità di valuta che possiede. Grazie al PoS, si raggiungono gli stessi risultati del PoW, facendo a meno delle grandi risorse di calcolo necessarie per il corretto funzionamento di quest'ultima. Non esiste quindi un block reward e i miners, anche detti *forgers* in questo caso, vengono ricompensati con le commissioni necessarie per inviare transazioni sulla Blockchain.

Capitolo 3

Solidity

Gli Smart Contracts di Ethereum sono scritti con i cosiddetti *Specialized Contract Specification Languages*. Sono attualmente disponibili tre alternative: *LLL* (simile a LIPS), *Serpent* (simile a Python) e *Solidity*, che ricorda molto Javascript. Quest'ultimo è il linguaggio ufficiale di Ethereum, che lo rende di fatto il più diffuso. Si tratta di un linguaggio Object-Oriented nel quale la definizione del concetto astratto di *contratto* è molto simile a quello di classe.

3.1 Punti chiave

Il linguaggio è dotato di alcune caratteristiche chiave:

- Types: Solidity supporta molti data types che devono però essere noti a tempo di compilazione dato che il linguaggio è *statically typed*.
  Sono supportati booleans, integer (signed o unsigned da 8 a 256 bit) e array di byte di dimensione fissa. Le stringhe possono essere usate come array di byte a dimensione dinamica ma, ad oggi, non è stato ancora introdotto il supporto per variabili floating point. Un ulteriore data type presente all'interno di Solidity è l'Ethereum Address che contiene i 20 byte necessari per memorizzare un indirizzo Ethereum. Presente anche il supporto per le structs, enumerazioni e mappings. Il linguaggio Solidity possiede le seguenti caratteristiche:

- State variables: sono delle variabili che che vengono memorizzate permanentemente all'interno del contratto. Possono essere di tipo diverso e sono soggette a scopes e visibilità come in ogni altro linguaggio.

- Funzioni: le funzioni costanti hanno il solo scopo di restituire un valore e non possono modificare lo stato del contratto. Possono essere chiamate direttamente senza necessità di spendere Ethereum per la scrittura dato che non modificano la Blockchain. Le funzioni di transazione invece sono usate per alterare lo stato del contratto e quindi comportano un costo di scrittura.

- Function modifiers: sono costrutti che vengono utilizzati per alterare il comportamento di specifiche funzioni e per controllare se una data condizione sia stata soddisfatta prima che una funzione venga eseguita. I function modifiers sono considerati proprietà di ereditarietà dei contratti, ogni funzione può appartenere a molteplici modifiers che possono essere sovrascritti da contratti derivati.

- Eventi: rappresentano un modo, all'interno di Solidity, di fornire informazioni all'esterno di uno Smart Contract. Utilizzano i log delle transazioni dell'Ethereum Virtual Machine (EVM). Le funzioni possono invocare gli eventi e inviare degli *event messages* che saranno memorizzati sulla Blockchain. Gli event messages non sono accessibili dall'interno dei contratti, nemmeno dal contratto che li ha generati.

All'interno di Solidity esistono due tipologie di memoria all'interno del quale è possibile manipolare e gestire dati:

- Memory: un'area di memoria "infinitamente" espandibile, sotto forma di array di byte lineare non persistente che è inizializzato ad una nuova istanza ogni volta che il contratto riceve una message-call. Ogni nuova word di 256 bit di memoria richiesta ha un prezzo (in *gas*) che deve essere pagato, il cui costo cresce quadraticamente al crescere dell'utilizzo.

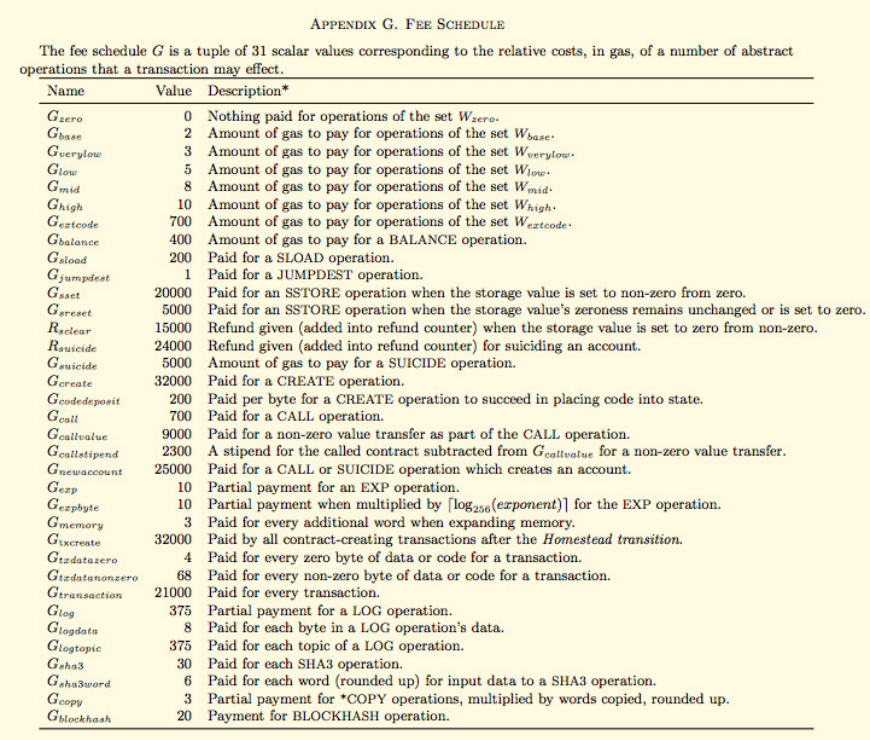
- Storage: è una mappa key-value di parole di 256 bit. Questa tipologia di memoria non viene resettata al termine della computazione, rimane quindi persistentemente in memoria.

I contratti possono essere ereditati da altri contratti e possono chiamare porzioni di codice che "vivono" in altre porzioni della Blockchain. Ogni volta che un contratto effettua una message-call, il codice chiamato viene eseguito all'interno del suo ambiente utilizzando il suo spazio di memoria; il chiamante è inoltre responsabile di pagare il costo della transazione. Il codice chiamato può inoltre accedere al valore, al chiamante ed ai dati del messaggio in entrata, mentre il nodo può restituire un valore o un array di byte di dimensione fissa come output.
Uno dei principali difetti di Solidity è la totale mancanza di un completo sistema di gestione della sicurezza; questo porta a lasciare totalmente nelle mani dello sviluppatore gli aspetti sopracitati, il che conduce spesso a comuni errori di programmazione così frequenti da essere citati per altro all'interno della documentazione ufficiale.

Immutabilità La natura immutabile di Solidity tende ad aggravare i noti problemi di sicurezza delle tecnologie Blockchain. Una volta effettuato il deploy di un contratto, è impossibile tornare indietro. Gli sviluppatori sono quindi incentivati a non compilare il codice tanto quanto farebbero con un'applicazione centralizzata. In accordo con le best practices, gli sviluppatori dovrebbero testare il proprio codice sia nella TestNet che in una rete di test privata prima di effettuare il deploy sulla MainNet. Conseguentemente, ha preso piede una pratica che consiste nell'inserimento di una funzione *kill* del contratto, nel caso che quest'ultimo dovesse essere dismesso per svariate motivazioni.

3.2 Il gas

Il gas è stato inizialmente pensato per evitare *DDoS* alla rete Ethereum. All'atto di esecuzione di un contratto, ogni transazione o funzione costa un



APPENDIX G. FEE SCHEDULE

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
$G_{suicide}$	5000	Amount of gas to pay for a SUICIDE operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SUICIDE operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txcreate}$	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.

Figura 3.1: Schema quote gas di Ethereum [3]

certo numero di unità di gas. L'esatto ammontare di gas necessario per la transazione è determinato da quante risorse (numero di accessi in memoria, memoria occupata, computazioni) si rendono necessarie per l'invio della transazione.

Il sistema di funzionamento del gas non è molto differente dall'uso del Kw per la misurazione dell'energia elettrica consumata. Una differenza con il mercato dell'energia consiste nel fatto che il mittente della transazione stabilisce il costo espresso in gas che il miner può decidere di accettare o meno. Questo significa che più saremo disposti a pagare il gas, maggiori saranno le probabilità che la transazione venga *minata* - e quindi ricevuta - in tempo breve. Essenzialmente, ogni riga di codice Solidity all'interno di uno Smart Contract, necessità di una - seppur minima - quantità di gas per essere eseguita.

3.3 La funzione di fallback

Quando un contratto sviluppato in Solidity viene compilato in bytecode EVM, le sue funzioni vengono inviate all'inizio del contratto compilato e sono identificate da una firma derivata dal nome della funzione e dai suoi parametri in ingresso. Quando una funzione viene chiamata, la firma viene infatti fornita nella chiamata. Se non vengono trovate funzioni corrispondenti oppure non viene fornita nessuna firma, la funzione di *fallback* di un contratto viene invocata.
La funzione di fallback non ha argomenti e non restituisce nessun tipo di dato. Se un contratto riceve una firma vuota - insieme ad una commissione - ma non viene definita nessuna funzione di fallback, verrà generata un'eccezione. Una best practice in Solidity consiste nel non permettere ad una funzione di fallback di contenere istruzioni che utilizzino molto gas poichè questo potrebbe generare un'ulteriore eccezione al momento dell'invocazione della funzione *send* che viene utilizzata per trasmettere criptovaluta. In questo caso il contratto non sarà in grado di ricevere criptovaluta.
Dato che la funzione di fallback può essere eseguita improvvisamente, si viene a creare una vulnerabilità la quale può essere sfruttata da un eventuale attaccante; essa consiste nell'eseguire una funzione non ricorsiva in un loop. In letteratura questa problematica viene definita come *reentrancy*.

3.4 Problemi di sicurezza

Gli Smart Contracts sono vulnerabili al phishing ed altri tipi di attacchi in maniera uguale, se non maggiore, alle controparti centralizzate. Ad esempio, è buona norma non eseguire troppe chiamate esterne all'interno del codice; le chiamate a dei contratti *untrusted* possono infatti portare all'esecuzione di codice malevole all'interno del contratto stesso. Nel caso in cui la chiamata ad un contratto sia strettamente necessaria, è opportuno utilizzare un flag che indichi la potenziale presenza di un'interazione **unsafe**. Quando si utilizzano chiamate *raw* (della forma indirizzo.call()) o chiamate di contratto (della

forma contrattoEsterno.qualcheMetodo()), è buona prassi assumere che del codice malevolo possa essere eseguito. Anche nel caso in cui contrattoEsterno non sia malevolo, del codice che lo sia può ancora essere eseguito da ogni contratto che viene invocato. Solidity offre chiamate di metodi a basso livello su indirizzi *raw*: indirizzo.call(), indirizzo.callcode(), indirizzo.delegatecall() ed indirizzo.send(). La loro particolarità è che non generano in nessun caso un'eccezione, ma restituiranno invece **false** se la chiamata dovesse generarne una.
Nel caso questi metodi vengano utilizzati, è opportuno gestire la possibilità che la chiamate fallisca, controllando il valore restituito:

```
// pericoloso
indirizzo.send(55);
indirizzo.call.value(55);
// invia tutto il restante Gas senza controllare il risultato
indirizzo.call.value(100)(bytes4(sha3("deposito()")));
/* se il deposito dovesse generare un'eccezione, la chiamata raw
    restituisce FALSE e gli Ether non vengono restituiti */

// maggiormente sicuro
if (!indirizzo.send(55)) {
   // codice che gestisca l'eccezione nel caso di FALSE
}
contrattoEsterno(indirizzo).deposit.value(100);
```

Le chiamate esterne possono fallire accidentalmente o deliberatamente. Per minimizzare i conseguenti danni, è buona norma isolare ogni chiamata all'interno della propria transazione che può essere inizializzata del destinatario della chiamata. Questa pratica è particolarmente importante nel caso dei pagamenti, dove risulta essere migliore lasciar prelevare all'utente i fondi invece di inviarli automaticamente.

3.5 Gestione delle eccezioni

Se un account non contiene codice, una funzione genera un'eccezione o il contratto termina il gas a disposizione, una chiamata a funzione genererà un'eccezione. Un'ulteriore eccezione fermerà l'esecuzione della funzione attuale e ripristinerà tutte le successive modifiche allo stato del contratto precedente alla generazione dell'eccezione. Al momento di stesura di questa tesi, Solidity non contiene ancora un sistema per il catch delle eccezioni, detto questo, però, l'unico effetto collaterale è quello che la transazione finirà il gas a disposizione. Un altro problema consiste nel fatto che quando si utilizza la funzione *send* di un contratto, lo stato del contratto non viene ripristinato e la funzione potrebbe restituire **FALSE**. Queste incongruenze nella gestione delle eccezioni in Solidity possono avere importanti conseguenze sulla sicurezza del contratto stesso, ad esempio se un contratto non controllasse il valore restituito da send, potrebbe assumere che la transazione sia andata a buon fine dato che non è stata generata nessuna eccezione.

Unpredictable state Lo stato dei contratti è determinato dal valore dei campi e dal *balance*. Quando un utente invia una transazione sulla rete per chiamare un contratto, non può avere l'assoluta certezza che la transazione sarà eseguita nello stesso stato in cui si trovava il contratto nel momento in cui è stata inviata la transazione stessa.
La variazione di stato può verificarsi poiché, nel lasso di tempo trascorso dalla chiamata, altre transazioni potrebbero aver alterato lo stato del contratto. Quando i gruppi di miners raggruppano le transazioni all'interno dei blocchi, non sono tenuti a conservare l'ordine temporale delle suddette transazioni, potrebbero inoltre decidere di non includere alcuna transazione.
Esiste inoltre un'ulteriore circostanza in cui l'utente potrebbe non conoscere lo stato in cui si troverà la sua transazione; ovvero quando due miners trovano contemporaneamente un nuovo blocco da *minare*. Quando questo accade, la Blockchain si divide in due diversi branch, ovvero effettua una *fork*. I "successivi" miners potranno quindi allegare le loro transazioni ad uno o all'altro

branch. Trascorso del tempo però, solo il branch più lungo sarà considerato parte della Blockchain, mentre quello più corto verrà di fatto abbandonato e con lui tutte le sue transazioni. Si deduce quindi che per un utente, conoscere lo stato attuale, può essere determinante per pubblicare nuove transazioni. Tuttavia, lo stato potrebbe sempre cambiare poiché le transazioni che lo hanno generato potrebbero trovarsi all'interno del branch più corto.
In alcuni casi, non conoscere lo stato all'interno del quale verranno eseguite le transazioni può portare a problemi di sicurezza e vulnerabilità importanti. Infatti, nonostante il codice non possa essere alterato una volto effettuato il deploy sulla Blockchain, è possibile costruire un contratto i cui componenti possono essere aggiornati dal proprietario linkando successivamente il contratto ai componenti malevoli.

Capitolo 4

Analisi e sviluppo dell'applicazione

Finora abbiamo analizzato alcuni aspetti teorici con il fine di studiare e comprendere la tecnologia Blockchain, quali: crittografia, decentralizzazione ed il linguaggio di programmazione Solidity. Una volta acquisite queste conoscenze, è stata condotta una fase di analisi dei requisiti con conseguente scelta operativa delle tecnologie, al momento disponibili, che si adattassero nel miglior modo possibile agli obiettivi posti in fase di analisi.

4.1 Analisi dei requisiti

La parte progettuale di questa tesi verte quindi sullo sviluppo di un'applicazione decentralizzata, o *Dapp*, il cui obiettivo è quello di simulare il funzionamento di una piattaforma per esprimere il proprio voto durante una votazione elettronica. Quest'ultima, così come il classico sistema elettorale, permettere di visualizzare la lista dei candidati e di votare una sola volta.
L'ideaologia dietro questa applicazione consiste nel fatto che in futuro non sarà necessario utilizzare carta e penna per esprimere il proprio voto durante un'ipotetica votazione elettronica, bensì sarà sufficiente utilizzare un computer/smartphone ed una connessione ad internet. Questo permetterà di far risparmiare tempo agli elettori ed alla pubblica amministrazione, senza contare il notevole risparmio economico e la garanzia che i voti non possano essere in nessun modo contraffatti o nulli.

Ad oggi esistono svariate proposte per usare la Blockchain come piattaforma di e-voting; le soluzioni attualmente in circolazione raggiungono però l'obiettivo di garanzia della privacy del votante attraverso l'introduzione di un intermediario o *third party*, fattore che si scontra con i principi fondamentali della tecnologia Blockchain.
Il requisito principe per la realizzazione del software è quindi automatizzare l'intero processo di scrutinio dei voti per ridurre drasticamente tempi e costi di un'ipotetica votazione elettronica evitando l'introduzione di terze parti. L'applicazione, inoltre, dovrebbe essere in grado di rispettare le regole generali di una classica votazione, come la necessità di essere identificati prima del voto e l'impossibilità di esprimere il proprio voto più di una volta.
Per raggiungere questi obiettivi e garantire al tempo stesso il livello di sicurezza più alto possibile, contestualmente ai limiti della tecnologia, si è deciso di utilizzare la Blockchain Ethereum a causa delle sue caratteristiche intrinsiche di gestione attraverso gli Smart Contracts e del supporto disponibile online, in quanto circa il 90% delle DApps sono sviluppate attraverso tale piattaforma. Come visibile in 4.1, l'applicazione è stata quindi realizzata seguendo una semplice astrazione basata su un diagramma UML dei casi d'uso.

4.2 Strumenti utilizzati

4.2.1 Truffle Framework

Truffle Framework è un insieme di librerie per la compilazione, linking, deployment e gestione dei sorgenti per i contratti Ethereum, che semplifica di molto lo sviluppo di applicazioni decentralizzate in quanto contiene al suo interno tutto il necessario per la gestione di Smart Contracts. Ogni volta che si vuole compilare ed effettuare il deployment dell'applicazione, non è possibile rimuovere la vecchia versione e testare la nuova, ma sarà necessario inviare la nuova versione sulla Blockchain, avendo di fatto svariati duplicati fino al raggiungimento dello stadio di sviluppo per il rilascio finale. Questo, insieme al costo in termini di cryptovaluta che comporta ogni scrittura su una *live*

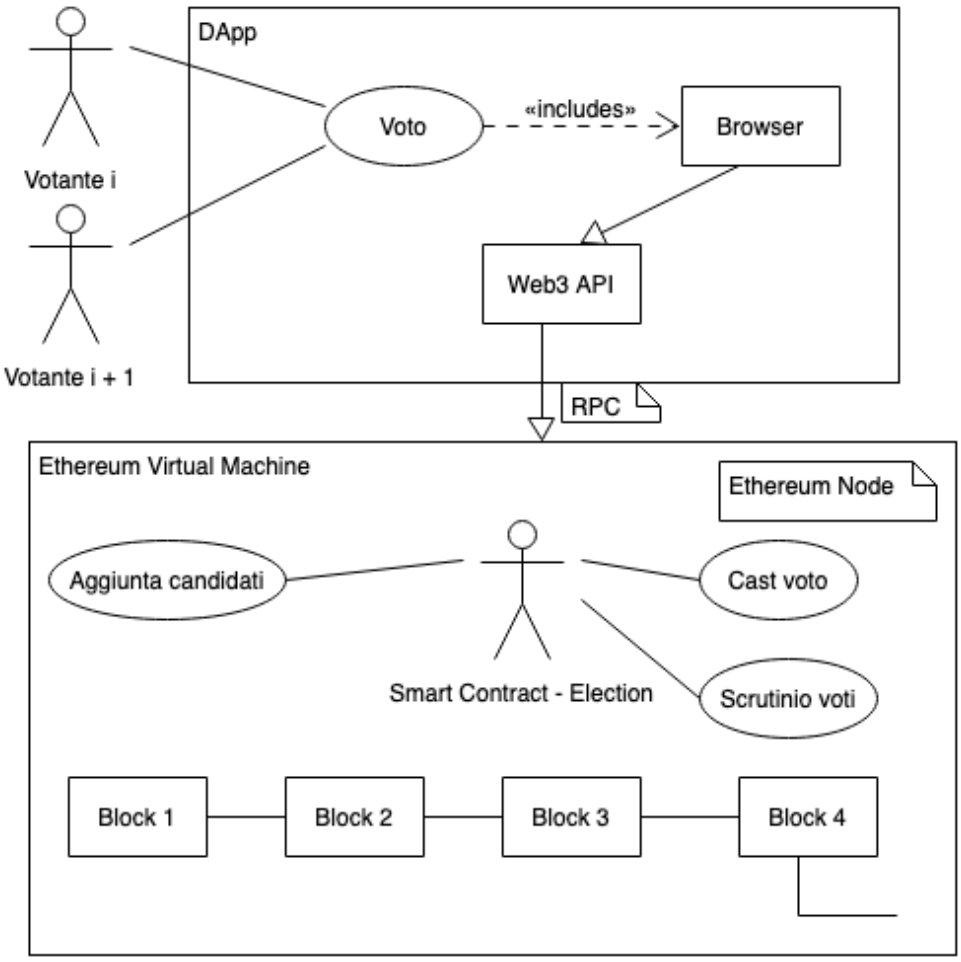


Figura 4.1: Diagrammi dei casi d'uso progettato durante l'analisi dei requisiti.

Blockchain, giustifica l'utilizzo di una cosiddetta *Test-Net*.
A tal fine, è stata utilizzata l'utility *Geth* la quale esegue una Blockchain di test in memoria, con degli account fittizi dotati di un numero predeterminato di Ether (cryptovaluta utilizzata su blockchain Ethereum) ciascuno.
Al suo interno sono inoltre presenti funzionalità di testing e unit testing, sia in Javascript che in Solidity, che rendono molto semplice prevedere il comportamento della dapp che si sta sviluppando testandola in svariate casistiche. I test sono avviabili da console con il comando: *truffle test test.js*.

4.2.2 Web3

Web3.js è una collezione di librerie che permettono di interagire con un nodo Ethereum, sia locale che remoto, attraverso l'uso di HTTP o IPC.
All'interno di Web3, uno dei concetti più potenti è la ridefinizione delle strutture dati. E' importante notare che la Blockchain è solo una di svariate tecnologie nel *Decentralized Web Stack*; infatti è un ottimo modo per memorizzare chi ha fatto cosa e quando in P2P, ma non è ideale per memorizzare grandi quantitativi di dati per due principali motivazioni: scalabilità (le Blockchain sarebbero troppo lente in questo scenario) e privacy, che non è stata prevista in fase di design. Negli scorsi anni era molto difficile - se non impossibile - sviluppare applicazioni decentralizzate, almeno per la grande maggioranza dei programmatori, ma questo è cambiato quando nel 2017 è stato introdotto l'eco-system Web3.
Grazie a Web3, infatti, è possibile sviluppare web apps che interagiscano con la Blockchain, leggendo e scrivendo dati tramite gli Smart Contracts. Web3 comunica con la Blockchain Ethereum attraverso *JSON RPC*, acronimo di "Remote Procedure Call Protocol", che permette a Web3 di effettuare richieste individuali ai noti Ethereum per leggere e scrivere sulla rete, similmente a quanto fatto da jQuery con le JSON API per interagire con i web server.

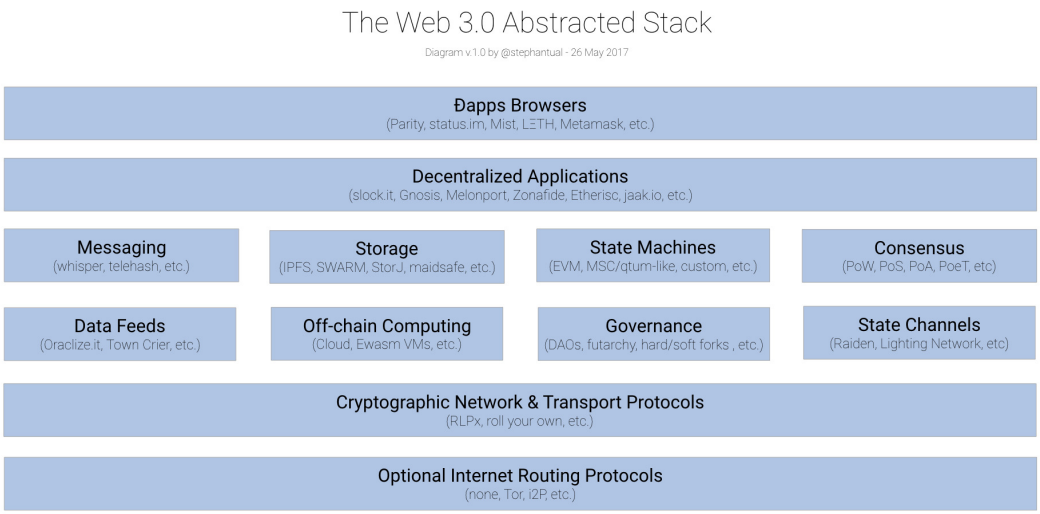


Figura 4.2: Web 3.0 stack

4.2.3 Ganache-CLI

Ganache-CLI, parte della suite Truffle tra gli strumenti di sviluppo per Ethereum, è una versione a riga di comando di una Blockchain locale; utilizza *EthereumJS* per simulare il comportamento di un client e rende lo sviluppo di applicazioni decentralizzate molto rapido e (più) sicuro. Include inoltre tutte le funzioni RPC più diffuse e le sue funzionalità principali - come il trigger di eventi - possono essere eseguite deterministicamente.
Una volta avviata la Blockchain in locale, Ganache-CLI ci fornisce degli indirizzi fittizzi intesi per un utilizzo di test durante le fasi di sviluppo.

Geth Per l'utilizzazione nel mondo reale, un'utility come *Geth* risulterebbe più appropriata. Geth è una *Multipurpose Command Line Interface*, implementata nel linguaggio di programmazione *Go* sviluppato da Google, che si occupa di far girare un nodo Ethereum. Utilizzando Geth è quindi possibile effettuare mining della criptovaluta Ethereum, spostare fondi, creare contratti e transazioni visualizzandone l'intera cronologia, sia sulla *MainNet* che sulla *TestNet*.
E' possibile interagire con Geth attraverso tre differenti interfacce, quali: una console che fornisce un ambiente javascript a runtime per interfacciarsi con

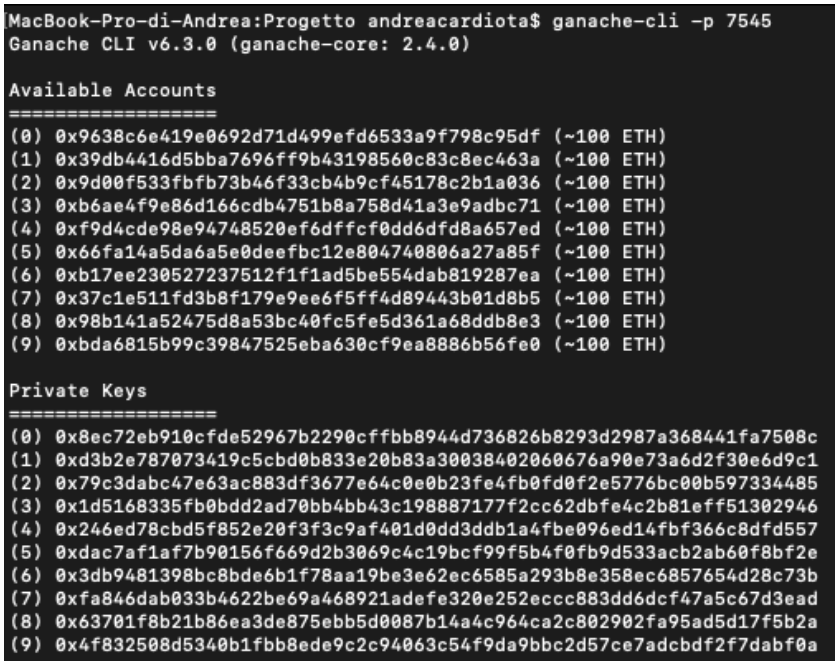


Figura 4.3: Indirizzi fittizzi con relative chiavi private messi a disposizione da Ganach-CLI.

il nodo stesso, attraverso la *Javascript Console API*; un server JSON-RPC e infine utilizzando dei command line *options and subcommands*. Quando si avvia il nodo Ethereum con il comando *geth*, quest'ultimo non sta effettuando nessun mining task; per far ciò basta utilizzare il comando *geth –mine –minerthreads=x*, modificando il paramento x a seconda dei core e thread che si hanno a disposizione.
Fondamentale è l'assegnazione di un **etherbase address**, senza il quale il processo di mining non può essere inizializzato.
E' inoltre possibile aggiungere dei dati supplementari (limitati a 32 bytes) al blocco minato. Per convienzione viene utilizzata una stringa Unicode in modo da poter settare un tag personalizzato. Segue un esempio:

```
miner.setExtra("Andrea")
...
debug.printBlock(131805)
BLOCK(be465b020fdbedc4063756f0912b5a89bbb473d1d1df84363e05ade0195cb1):
```

```
   Size: 531.00 B TD: 643485290485 {
NoNonce:
   ee48752c3a0bfe3d85339451a5f3f411c21c8170353e450985e1faab0a9ac4cc
Header:
[
...
       Coinbase:         a4d8e9cae4d04b093aac82e6cd355b6b963fb7ff
       Number:           131805
       Extra:            Andrea
...
}
```

4.2.4 MetaMask

MetaMask è un utility che permette di utilizzare applicazioni decentralizzate all'interno dei classici browser, sotto forma di estensione o plug-in, rendendo quindi superfluo l'utilizzo di browser dedicati.
La sua potenza risiede nella possibilità di connessione ad un nodo Ethereum, chiamato *INFURA*, con il fine di eseguire Smart Contracts; ciò rimuove di fatto la necessità di eseguire un intero nodo sulla macchina locale.
Combinando queste funzionalità con la sua facilità d'uso, MetaMask è stato adottato in maniera massiccia in tempo relativamente breve, aumentando significativamente il supporto degli sviluppatori e della comunità online, fattore che ha influito in maniera non indifferente nella scelta di utilizzarlo in questo progetto di tesi.
Ai fini dell'aplicazione sviluppata, attraverso questa utility è possibile importare gli account che Geth mette a disposizione, simulando di volta in volta ogni utente come se mostrasse al seggio la propria carta d'identità.
Una volta importato l'account, all'interno dei sorgenti javascript verrà controllato se l'utente in questione abbia votato o meno, permettendogli di votare in caso non lo abbia ancora fatto.

4.3 Solidity Smart Contract

Il contratto di questa applicazione decentralizzata, come già menzionato, è scritto in Solidity.
Di seguito sono analizzati i segmenti più significativi:

- Memorizzare gli account che hanno già votato in modo da impedire voti ripetuti:

```
mapping(address => bool) public voters;
```

- Trigger che viene invocato quando un voto è stato registrato correttamente:

```
emit votedEvent(_candidateId);
```

- Memorizzare un votante:

```
voters[msg.sender] = true;
```

4.3.1 Logica di controllo

Come affermato in precedenza, la logica di controllo principale dell'applicazione risiede all'interno dei sorgenti javascript, in particolare al loro interno si gestiranno le seguenti attività:

- Istanziazione di un nuovo contratto Truffle:

```
initContract: function() {
    $.getJSON("Votazione.json", function(election) {
      eVote.contracts.Votazione = TruffleContract(election);
      eVote.contracts.Votazione.setProvider(eVote.web3Provider);
      eVote.listenForEvents();
      return eVote.render();
    });
  }
```

Questo tipo di istanziazione serve a settare il provider in modo tale da essere in grado di interagire col contratto istanziato sulla Blockchain quando richiesto;

- Implementazione di un listener per gli eventi provenienti dal contratto:

```
listenForEvents: function() {
    eVote.contracts.Votazione.deployed().then(function(instance)
        {
      instance.votedEvent({}, {
        fromBlock: 'latest'
      }).watch(function(error, event) {
        console.log("Evento innescato", event)
        // refresh della pagina una volta espresso un voto
        eVote.render();
      });
    });
  }
```

Viene effettuato un controllo attivo sull'ultimo blocco scritto della catena; una volta rilevato il trigger dell'evento la pagina viene ricaricata per mostrare il conteggio aggiornato;

- Caricamento dati dal contratto:

```
eVote.contracts.Votazione.deployed().then(function(instance) {
      voteInstance = instance;
      return voteInstance.candidatesCount();
    }).then(function(candidatesCount) {
       // salvataggio promises per prelevare dati del candidato i
      const promises = [];
      for (var i = 1; i <= candidatesCount; i++) {
        promises.push(voteInstance.candidates(i));
      }
        ...
```

```
}
```

- Caricamento dati importati da MetaMask attraverso una funzione di libreria di Web3:

```
web3.eth.getCoinbase(function(err, account) {
      if (err === null) {
        eVote.account = account;
        $("#accountAddress").html("Account in uso: " + account);
      }
    });
```

- Funzionalità di casting del voto:

```
castVote: function() {
    var candidateId = $('#candidatesSelect').val();
    eVote.contracts.Votazione.deployed().then(function(instance)
        {
      return instance.vote(candidateId, {
        from: eVote.account
      });
    }).then(function(result) { // attesa aggiornamento del voto
      ...
  }
};
```

In questo modo andiamo a creare un'istanza del contratto che è legata ad uno specifico indirizzo della rete Ethereum; ogni istanza è caratterizzata da un mapping *1-to-1* dalla funzione Javascript alle funzioni del contratto stesso. Una volta creata l'istanza, viene resituita l'istanza del voto legato al votante ed al candidato votato.

4.4 Deployment

La fase di deploy dell'applicazione avviene sfruttando, in sequenza, tre strumenti. Come primo passo è necessario lanciare un'istanza locale di Blockchain Ethereum; per far ciò utilizziamo l'utility *Ganache CLI* mediante il comando:

```
ganache-cli -p 7545
```

la quale sarà raggiungibile tramite porta 7545 come indicato.
Una volta che la Blockchain è stata istanziata, è possibile compilar il contratto e ed effettuarne il deploy sulla Blockchain con un unico comando:

```
truffle migrate --reset
```

Effettuata questa operazione, la Blockchain - come visto in 4.3 - ci metterà a disposizione degli account fittizzi pre-caricati con un quantitativo di Eth sufficiente ad effettuare operazioni di test. Il flag reset sovrascrive tutte le precedenti istanziazioni dello stesso contratto; ciò risulta essere molto utile soprattutto in fase di sviluppo.
Infine, tramite la libreria Javascript *npm*, lanciamo il server locale che si occuperà di gestire tutte le richieste HTTP. Per far ciò utilizziamo il comando:

```
npm run dev
```

Tramite una breve analisi delle console messe a disposizione dai suddetti strumenti, si nota come gli ID delle transazioni per il deploy dei contratti - così come le transazioni di ogni singolo evento di voto - risultino combaciare una volta terminata l'intera fase di deploy dell'applicazione.

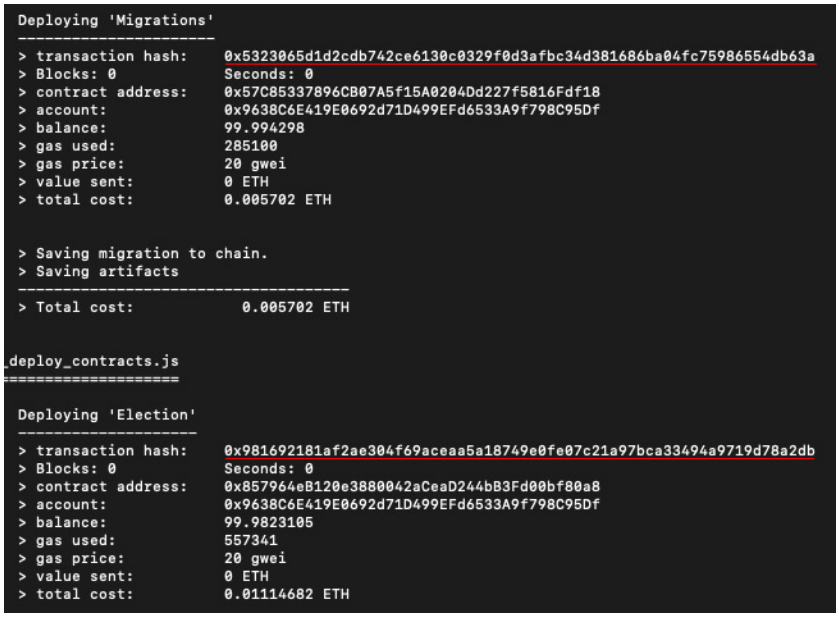


Figura 4.4: ID transazioni per il deploy dei contratti su console Truffle.

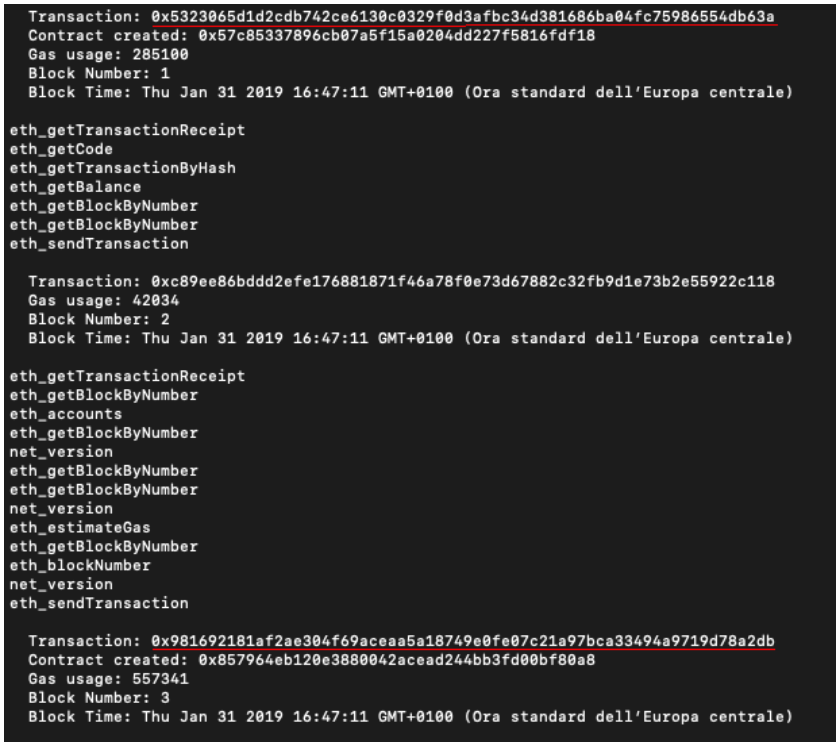


Figura 4.5: ID transazioni per il deploy dei contratti su console Blockchain.

Testing E' stata inoltre sviluppata una piccola utility di test. Tra le funzionalità più significative troviamo il controllo sulla correttezza dei parametri attribuiti ad ogni singolo candidato:

```
it("Inizializza il candidato con i valori corretti", function() {
    return Votazione.deployed().then(function(instance) {
      voteInstance = instance;
      return Votazione.candidates(1);
    }).then(function(candidate) {
      assert.equal(candidate[0], 1, "Contiene l'ID corretto");
      // itero per tutti i candidati
      return voteInstance.candidates(3);
    }).then(function(candidate) {
      ...
    });
  });
```

Di seguito una seconda funzionalità di testing che permette di assicurarsi che un utente non voti più di una volta:

```
  it("Genera un'eccezione nel caso di doppio voto", function() {
    return Votazione.deployed().then(function(instance) {
      voteInstance = instance;
      candidateId = 2;
      voteInstance.vote(candidateId, {
        from: accounts[1]
      });
      return voteInstance.candidates(candidateId);
    }).then(function(candidate) {
      var voteCount = candidate[2];
      assert.equal(voteCount, 1, "Accetta primo voto");
      // voto doppio
      return voteInstance.vote(candidateId, {
        from: accounts[1]
      });
```

```
  }).then(assert.fail).catch(function(error) {
    ...
});
```

4.5 Demo e User Flow

Dopo aver eseguito correttamente tutte le fase di deploy del software ed aver effettuato la connessione alla Blockchain locale, accedendo al Localhost ci troveremo davanti al front-end dell'applicazione:

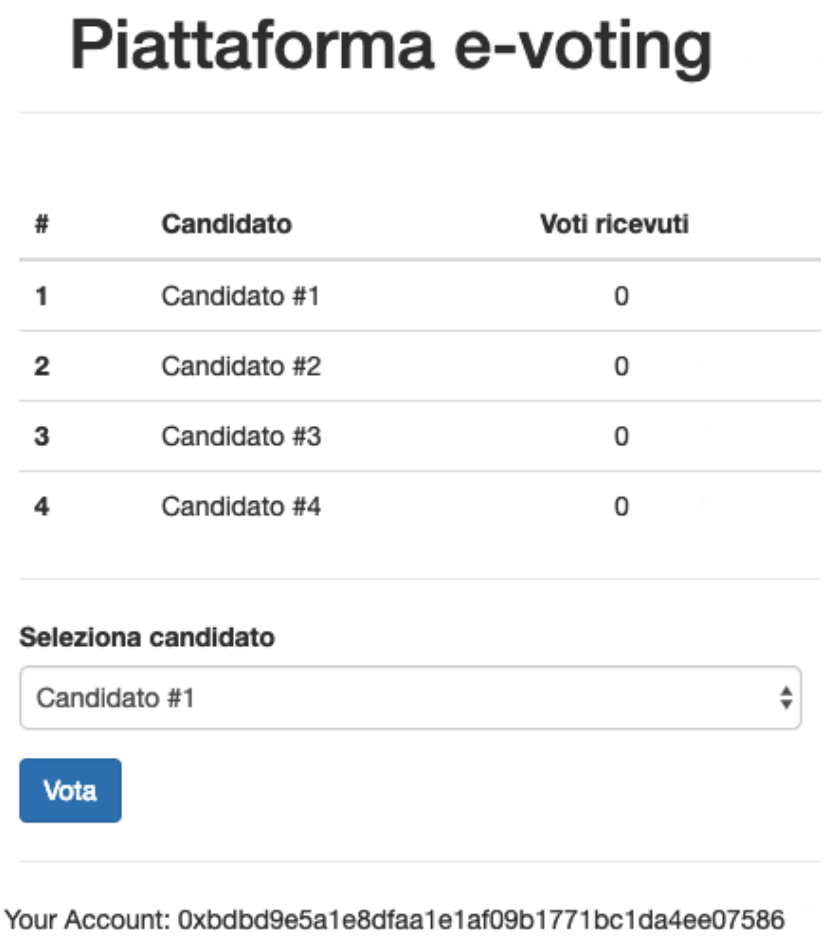


Figura 4.6: Come è possibile notare nella parte bassa dell'immagine, l'applicazione mostra l'indirizzo relativo all'account importato in MetaMask; esso rappresenta un indentificativo dell'elettore che esprime il proprio diritto di voto; potrebbe quindi essere sostituito da un codice fiscale o equivalente.
Una volta selezionato il proprio candidato ed aver inviato il proprio voto, all'utente viene proposto un dialog il quale mostra il costo della transazione che si sta per inviare e chiede conferma finale del voto, dalla quale non si può tornare indietro. Terminata questa fase, la pagina viene automaticamente ricaricata mostrando il conteggio dei voti aggiornato, privando inoltre l'utente della possibilità di esprimere un secondo voto.

Conclusioni

Nello sviluppo di questa tesi è stata fornita una visione di base dei meccanismi crittografici necessari al corretto funzionamento della tecnologia Blockchain. Si è successivamente discusso di come svariati campi di applicazione possano trarre considerevoli vantaggi dall'adozione di tale tecnologia, sia all'interno di nuove classi di applicativi software che in altre già esistenti e ben radicate sul mercato. Si è sviluppata un'applicazione decentralizzata il cui obiettivo era, oltre all'acquisizione di un set di conoscenze che non possedevo, quello di dimostrare come un sistema *Blockchain-based* possa semplificare un'operazione complessa come una votazione e far risparmiare, seppur non precisamente quantificato, tempo e denaro garantendo al tempo stesso un'integrità e una sicurezza difficilmente raggiungibili con i metodi attualmente utilizzati.

Si può affermare che i punti di forza dell'applicazione sviluppata siano la veridicità e integrità dei voti; quindi che i requisiti inizialmente posti siano stati rispettati, seppur esistano importanti margini di miglioramento. Nello specifico, in un eventuale utilizzo nel mondo reale, gli utilizzatori del software saranno, con probabilità molto alta, persone non addette ai lavori; è quindi certamente possibile alleggerire la fase di deployment semplicandone gli step. L'opzione più adatta in tal senso riguarda l'utilizzazione una piattaforma che si occupi di automatizzare deployment, scalabilità e manutenzione di applicazioni decentralizzate in *Content-Addressable Data Sharing Networks*.

Effettuando il deployment dell'applicazione mediante l'utilizzo di un protocollo P2P come *IPFS*, ogni nodo può memorizzare collezioni di hashed files che permettono il setacciamento dell'informazione richiesta attraverso l'intero

set di nodi. Restano aperte alcune problematiche come sicurezza e segretezza della tecnologia; allo stato attuale la sicurezza dei sistemi peer-to-peer non è perfetta, nonostante sia di gran lunga superiore ad un sistema centralizzato. Un approccio differente ma ugualmente funzionale è l'utilizzo delle cosiddette *Layer-2 Solutions*, che consistono nel costruire protocolli che lavorino al di sopra del base-layer aumentandone esponenzialmente la scalabilità.

Mi ritengo complessivamente soddisfatto del lavoro svolto durante la stesura di questa tesi, in quanto ho avuto la possibilità di approfondire una tematica dal mio punto di vista molto interessante sia allo stato attuale della tecnica ma soprattutto in ottica futura; credo infatti che possedere il set di conoscenze acquisito possa rivelarsi molto utile sia durante il percorso magistrale che nel mio futuro professionale.

Bibliografia

[1] Satoshi Nakamoto: Bitcoin: A peer-to-peer electronic cash system (Bitcoin whitepaper), 2008,
`https://bitcoin.org/bitcoin.pdf`

[2] Ethereum Foundation: Ethereum white paper, 2014,
`https://github.com/ethereum/wiki/wiki/White-Paper`

[3] Dr. Gavin Wood: Ethereum yellow paper,
`http://paper.gavwood.com/`

[4] Andreas M. Antonopoulos: Mastering Bitcoin 2nd edition, 2017,
`https://github.com/bitcoinbook/bitcoinbook`

[5] Solidity Documentation,
`https://solidity.readthedocs.io/en/develop`

[6] Geth Documentation,
`https://github.com/ethereum/go-ethereum/wiki/geth`

[7] Web3 Documentation,
`https://github.com/ethereum/web3.js`

[8] MetaMask Documentation,
`https://metamask.io/`

[9] Notes on the EVM, 2016,
`https://github.com/CoinCulture/evm-tools/blob/master/analysis/guide.md`

[10] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, Steven Goldfeder: Bitcoin and Cryptocurrency Technologies. Princeton University Press, 2016.

[11] Giovanni Ciatto: A gentle introduction to the Blockchain and Smart Contracts, 2018.

[12] State of Blockchain, Coindesk, 2018.

[13] Bitcoin Wiki, 2018,
`https://en.bitcoinwiki.org`

[14] Awesome Ethereum Virtual Machine, 2017,
`https://github.com/pirapira/awesome-ethereum-virtual-machine`

[15] Bitcoin power usage, 2018,
`https://powercompare.co.uk/bitcoin`

[16] Ethereum for web developers, 2017,
`https://medium.com/@mvmurthy/ethereum-for-web-developers-890be23d1d0c`

[17] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli: A survey of attacks on Ethereum Smart Contracts.

[18] Smart Contracts best practices,
`https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/recommendations.md`