Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Gestione della disponibilità nella catena di distribuzione del sangue con Smart Contract in Ethereum

Relatore:
Chiar.mo Prof.
Davide Sangiorgi

Correlatore:
Dott.
Stefano Pio Zingaro

Presentata da:
Michael Carchesio

Sessione I
Anno Accademico 2021/2022

Ai miei nonni.

Sommario

Ad oggi, la questione della fiducia sta diventando sempre più un problema. Molti si preoccupano dei propri dati personali e altri hanno timore di possibili attacchi e fanno fatica ad avere fiducia di qualsiasi ente terzo che offre servizi, oppure opera per conto di altri soggetti.

La tecnologia Blockchain e le applicazioni che si basano sul DLT - Distributed Ledger Technology, propongono nuovi metodi e rivoluzionano, oggi, il sistema economico, modificando i concetti di transazione, fiducia e proprietà, permettendo un nuovo sistema di validazione degli eventi, processi e scambi, basato sul raggiungimento di un consenso distribuito tra più parti denominate nodi o peer. L'idea che si cela dietro la Blockchain è quella di avere un sistema di fiducia, decentralizzato, in cui ogni entità ha lo stesso peso all'interno della catena distribuita (si vedano i vari riferimenti [Ant14][AW18], i quali contengono molte più informazioni).
Il sistema è basato sulla crittografia, la quale è diventata sempre più disponibile e compresa alla fine degli anni '80. Molti ricercatori hanno iniziato ad usare la crittografia per realizzare monete digitali, ma l'idea iniziale era di avere un sistema molto simile a quello utilizzato oggi dalle banche: un sistema centralizzato e, di conseguenza, facile da attaccare.

L'applicazione di tale tecnologia sta portando a numerosi cambiamenti, sia in ambito informatico e sia per la gestione logistica e la tracciabilità del dato in ambito farmaceutico e alimentare. Il caso di studio della seguente tesi si basa, dopo una attenta analisi delle tecnologie prese in considerazione, quali Bitcoin ed Ethereum, sulla realizzazione di alcuni smart contract[AW18], che permettono di verificare la disponibilità delle sacche di sangue nei centri trasfusionali e nelle aziende sanitarie pubbliche e private (per maggiori approfondimenti [APi19][AC20]). L'idea nasce dal problema di fiducia e di disponibilità in tempo reale di prodotti e, più nello specifico, di dati sanitari: ad oggi, non abbiamo a disposizione un sistema in grado di aggregare tutti i dati sanitari di ogni singolo paziente e ricostruire la *vita clinica* di questo può diventare molto complicato. Successivamente, abbiamo considerato un caso di studio più ristretto che ci permettesse di sfruttare, sperimentalmente, la tecnologia Blockchain di Ethereum e la distribuzione degli smart contract sulla rete tenendo in considerazione l'intera catena di validazione e l'algoritmo di consenso applicato a tale tecnologia.

L'analisi effettuata ci ha permesso di garantire le proprietà messe a disposizione dalla Blockchain: **decentralizzazione**, **immutabilità**, **verificabilità**, **accertabilità** e **trasparenza**. La produzione degli smart contract per la disponibilità delle sacche di sangue, è stata realizzata utilizzando l'ambiente Hardhat[1], i nodi messi a disposizione da tale ambiente per operare su una rete di test, la rete Testnet Ropsten utilizzando il wallet

MetaMask[2] e Ganache Provider[3] per avere, in alcuni casi, un ambiente in grado di tracciare le varie transazioni e l'inserimento di quest'ultime all'interno di blocchi verificati, validati dai miner e aggiunti alla catena. Infine, abbiamo utilizzato anche l'ambiente Remix per testare i vari smart contract, utilizzando diversi nodi per effettuare richieste ed interrogare il codice distribuito. Remix[4] è un ambiente che permette a chi lo utilizza, di interagire con varie reti di test e di utilizzare dei nodi validatori e una infrastruttura molto simile alla Mainnet, cioè alla rete principale di una Blockchain che, nel nostro caso, si identifica nella rete principale di Ethereum.

Gli smart contract, sono stati testati e sono stati utilizzati alcuni codici messi a disposizione dalla comunità, OpenZeppelin, la quale si occupa di sviluppare e rilasciare codici aggiornati, i quali vanno di pari passo con le versioni del linguaggio di programmazione per lo sviluppo degli smart contract denominato *Solidity*.

Gli smart contract presentati nella tesi sono soltanto una prima versione, e gli orizzonti di miglioramento ed eventuale sviluppo sono molteplici: dalla realizzazione di test in grado di verificare le varie lacune presenti negli smart contract, all'analisi di casi più complessi sfruttando librerie o codici già testati.

[1]Hardhat: https://hardhat.org/
[2]MetaMask: https://metamask.io/
[3]Ganache: https://docs.nethereum.com/en/latest/ethereum-and-clients/ganache-cli/
[4]Remix: https://remix.ethereum.org/

Indice

1 Introduzione **1**

2 Bitcoin **3**

2.1 Introduzione a Bitcoin . . . 3
2.1.1 B-money . . . 3
2.1.2 HashCash . . . 4
2.2 Transazioni in Bitcoin . . . 4
2.2.1 Transazioni . . . 4
2.2.2 Input e output delle transazioni . . . 5
2.2.3 Commissioni nelle transazioni . . . 6
2.2.4 Creazione di una catena di transazioni . . . 7
2.2.5 Script di blocco e di sblocco per le transazioni . . . 7
2.3 Generazione di chiavi e indirizzi . . . 10
2.3.1 Crittografia a chiave pubblica (crittografia asimmetrica) . . . 11
2.3.2 Secp256k1 . . . 13
2.3.3 Indirizzi in Bitcoin . . . 14
2.4 La rete e i nodi . . . 15
2.4.1 Network discovery . . . 16
2.4.2 Nodi completi o Full Node . . . 17
2.4.3 Nodi leggeri . . . 17
2.5 Il consenso . . . 19
2.5.1 Problema dei generali Bizantini . . . 19
2.5.2 Consenso decentralizzato . . . 21

3 Ethereum **23**

3.1 Introduzione ad Ethereum . . . 23
3.1.1 La criptovaluta in Ethereum . . . 24
3.1.2 Ethereum Virtual Machine - EVM . . . 24
3.1.3 Gas . . . 25
3.2 Generazione delle chiavi . . . 26
3.3 Transazioni Ethereum . . . 26

3.3.1 EOA e Smart contract . . . 26
3.3.2 Come è strutturata una transazione . . . 27
3.3.3 Propagazione di una transazione . . . 28
3.4 Smart contract Ethereum . . . 28
3.4.1 Esecuzione di uno smart contract . . . 28

4 Blockchain e algoritmi di consenso 30
4.1 Blockchain pubblica o privata? . . . 31
4.2 Protocollo di consenso . . . 31
4.2.1 Proof-of-Work (PoW) . . . 32
4.2.2 Proof-of-Stake (PoS) . . . 32

5 Introduzione alla gestione della filiera del sangue 33
5.1 Descrizione del caso di studio . . . 33
5.1.1 Le principali componenti del sangue . . . 34
5.1.2 La fase di test e di distribuzione . . . 34
5.1.3 La fase di raccolta . . . 34
5.1.4 La fase di trasporto . . . 35
5.1.5 Utilizzo di una sacca . . . 35
5.2 Problematiche riscontrate e motivazione . . . 36
5.3 Proposta di alcuni smart contract . . . 36

6 Solidity 37
6.1 Introduzione al linguaggio . . . 37
6.1.1 Gestione delle versioni . . . 37
6.1.2 Gestione di una transazione in Solidity . . . 38
6.1.3 Componenti utilizzati . . . 38
6.2 Tipi e dati . . . 39
6.2.1 Unsigned integer . . . 39
6.2.2 Mapping . . . 39
6.2.3 Modifier per la gestione di criticità . . . 40
6.2.4 Gestione degli eventi . . . 40
6.2.5 Constructor per operazioni iniziali . . . 41
6.2.6 Comunicazione tra smart contract . . . 41

7 Declinazione della soluzione al caso di studio 44
7.1 Introduzione . . . 44
7.2 Fase di testing . . . 45
7.2.1 Verifica creazione di una sacca di sangue . . . 45
7.2.2 Verifica disponibilità di una sacca di sangue . . . 46
7.2.3 Verifica di una sacca non assegnata al paziente . . . 47

7.3 Standard OpenZeppelin 48
7.4 Proprietà di una sacca di sangue 49
7.4.1 Stato dello smart contract 49
7.4.2 Funzioni utili 50
7.5 Smart contract per gestire la disponibilità 51
7.5.1 Stato dello smart contract 51
7.5.2 Utilizzo di *modifier* 52
7.5.3 Aggiunta di una sacca di sangue 53
7.5.4 Usabilità della sacca 54
7.5.5 Assegnare una sacca di sangue 55
7.5.6 Eliminare una sacca assegnata 55
7.6 Eventi negli smart contract proposti 57

8 Conclusioni **58**
8.1 Sviluppi futuri 58
8.1.1 Concetto di multi-firma 59
8.1.2 Standard token ERC-20 59

Bibliografia **63**

Elenco delle figure

2.1 Transazione come partita doppia [Ant14] . . . 5
2.2 Catena di transazioni [Ant14] . . . 7
2.3 Catena di transazioni [Nak08] . . . 8
2.4 Script di sblocco e blocco [Ant14] . . . 9
2.5 Esecuzione degli script [Ant14] . . . 10
2.6 Codifica Base58 . . . 15
2.7 Nodi nella rete Bitcoin . . . 15
2.8 Richiesta di entrata nella rete . . . 16
2.9 Funzionamento Bloom filter [RD14] . . . 18
2.10 Comandante leale nello scenario Bizantino [LP82] . . . 20
2.11 Comandante malintenzionato nello scenario Bizantino [LP82] . . . 20

3.1 Costo del gas [Woo14] . . . 25

4.1 Proprietà nella Blockchain . . . 30

6.1 Ciclo di vita di uno smart contract . . . 38
6.2 Operazione di call . . . 41
6.3 Operazione di delegatecall . . . 42
6.4 Esecuzione degli smart contract . . . 43

7.1 Workflow per inserire una sacca . . . 53
7.2 Workflow per modificare lo stato di una sacca . . . 54
7.3 Workflow per eliminare una sacca di sangue . . . 56

Capitolo 1

Introduzione

L'idea di utilizzare una Blockchain per la gestione di valute venne proposta nel 2008 con la pubblicazione del documento *"Bitcoin: A Peer-to-Peer Electronic Cash System"* da parte di Satoshi Nakamoto. Questi, ha combinato diverse tecnologie precedenti come *B-money* e *HashCash* per creare un sistema di criptovaluta completamente decentralizzato, il quale non si basa su un'autorità centrale per l'emissione o la regolamentazione di una valuta e la convalida delle transazioni [Ant14].

Bitcoin è fondato su un sistema peer-to-peer completamente distribuito. In quanto tale, non esiste un server centrale o un unico punto di controllo che si occupa di verificare tutte le transazioni e di creare o coniare nuova moneta. La creazione avviene attraverso un processo chiamato *mining*, il quale implica la ricerca di una soluzione ad un cosiddetto *"puzzle crittografico"*. A partire dagli anni '80 si è iniziato a pensare alla realizzazione di moneta che fosse del tutto digitale ma, come già accennato in precedenza, tutte le valute erano centralizzate ed erano, quindi, facili da attaccare da parte di governi o hacker.

Successivamente, si è avuta la necessità di estendere il protocollo Bitcoin per realizzare applicazioni e codice distribuito. Nel dicembre del 2013 un programmatore, appassionato della tecnologia Bitcoin, Vitalik Buterin, pubblicò un *white paper* con alcune idee, come quella di avere una blockchain Turing completa per tutti gli usi. Dopo alcuni feedback e considerazioni da parte di appassionati e programmatori, uno in particolare, chiamato Gavin Wood, propose l'idea di una blockchain come piattaforma per la creazione di denaro programmabile, con la capacità di supportare applicazioni contenenti risorse digitali, sviluppando il protocollo Ethereum. Questo introdusse il concetto di *EOA (Externally Owned Accounts)* e di *smart contract* e la comunicazione tra questi attraverso transazioni (per maggiori approfondimenti [AW18]).

Dopo alcuni anni, venne estratto, nel 2015, il primo blocco di Ethereum che ha permesso l'introduzione della nuova tecnologia proposta e avviato l'EVM (Ethereum Virtual

Machine): una macchina virtuale, single thread, che permette di eseguire codice distribuito tra tutti i componenti della piattaforma Ethereum.

Quando trattiamo la tecnologia Bitcoin o Ethereum, abbiamo a che fare con diverse parti:

- **Protocollo**: una rete peer-to-peer totalmente decentralizzata e senza nessun controllo da parte di una autorità.
- **Blockchain**: una catena di blocchi contenente tutte le transazioni verificate, validate e inserite da un miner in un blocco; anch'esso validato dall'intera comunità. In questo caso, abbiamo a che fare con il *Distributed Ledger Technology (DLT)*, cioè un registro distribuito tra tutti i partecipanti alla rete, i quali mettono a disposizione risorse di calcolo in comune che permettono di validare le transazioni, evitando la comunicazione con intermediari terzi, raggiungendo un *consenso distribuito* ottenuto attraverso sistemi crittografici che garantiscono l'unicità e la resilienza.
- **Algoritmo di consenso**: un insieme di regole per convalidare, indipendentemente, delle transazioni e per controllare l'emissione di nuova valuta. L'algoritmo utilizzato da Bitcoin è denominato *Proof-of-Work (PoW)* ma, oggi, ci si sta spostando verso un algoritmo che abbia un impatto energetico più basso, vale a dire l'algoritmo di consenso *Proof-of-Stake (PoS)* già adottato, il giorno 8 giugno del 2022, dalla Blockchain di Ethereum effettuando un'operazione di **merge** sulla rete testnet Ropsten. Il passaggio definitivo alla *PoS* avverrà attraverso ***The Merge*** interrompendo l'attività di *mining*.

Capitolo 2

Bitcoin

In questo capitolo, verrà illustrato il funzionamento della Blockchain di Bitcoin assieme a tutte le specifiche che esso adotta: partendo dal funzionamento con la gestione delle transazioni e il mining, fino ad arrivare a toccare vari punti riguardanti l'intera rete di nodi e il consenso, utilizzato da tale tecnologia per validare i blocchi da aggiungere alla catena.

2.1 Introduzione a Bitcoin

Bitcoin è una raccolta di concetti e tecnologie, nello specifico tutte le unità di valuta denominate Bitcoin vengono utilizzate da tutti i partecipanti o nodi della rete. Essi possono inviare denaro ad altri indirizzi validi, acquistare dei beni oppure estendere il proprio credito ricevendo BTC da altri partecipanti alla rete.

Come spiegato precedentemente in breve, la tecnologia Blockchain di Bitcoin non è nata dal nulla, ma Satoshi Nakamoto, pseudonimo del fondatore la cui identità è ancora oggi sconosciuta, ha combinato varie tecnologie, tra cui possiamo trovare *B-money* e *HashCash* per realizzare un sistema totalmente decentralizzato.

2.1.1 B-money

La tecnologia B-money venne inventata dallo scienziato e informatico Wei Dai nel 1998. Tale progetto non venne lanciato ufficialmente e offriva delle funzionalità simili a quelle fornite dal Bitcoin di oggi. Il concetto sul quale si fonda tale tecnologia è quello di avere il requisito della validazione di transazioni sfruttando la *Proof-of-Work* utilizzata, come sappiamo, da Bitcoin. Inoltre, tale progetto proponeva l'utilizzo di firme digitali basate su una crittografia a chiavi pubbliche per: autenticare le transazioni e verificare i legittimi proprietari di moneta virtuale.

Lo stesso scienziato, però, presume che l'ideatore di Bitcoin non abbia nemmeno letto il suo report e che solo dopo aver ideato la tecnologia, messa a disposizione di tutta la comunità mondiale, abbia scoperto l'esistenza di B-money e abbia accreditato nel suo articolo relativo a Bitcoin, l'ideatore di tale tecnologia.

2.1.2 HashCash

Il sistema HashCash è stato inventato da Adam Back nel 1997 e tale tecnologia, oggi, viene utilizzata come funzione di mining in Bitcoin. Venne creata per rispondere a possibili attacchi DoS (*Denial-of-Service*) e per proteggersi da eventuali e-mail *spam.* La prima versione di HashCash utilizzava come funzione crittografica hash lo standard *SHA-1*, mentre la tecnologia Bitcoin, lanciata nel 2008 con il "whitepaper" di Satoshi Nakamoto, utilizza lo SHA-256 assieme all'algoritmo crittografico basato su ECDSA[Fab] [Cut] (algoritmo di crittografia sulle curve ellittiche).
Ciò che avveniva con HashCash era di aggiungere, ad ogni e-mail inviata, un'intestazione basata su funzione hash che poteva essere prodotta solo dalla CPU del computer dal quale veniva lanciata tale e-mail. Tale *sigillo* è un identificatore per mostrare che l'utente ha utilizzato il processore per un breve periodo di tempo.

La funzione hash utilizzata in Bitcoin, permette di calcolare in modo semplice l'hash di un dato passato come parametro, ma rende molto difficile, se non impossibile, trovare la pre-immagine (o *pre-digest*) di tale dato partendo dal valore calcolato (*digest*); questo, perchè la funzione hash è anche denominata *funzione non bigettiva* e, quindi, non può essere invertita. Più precisamente, stiamo dicendo che invertire tale funzione comporta una potenza di calcolo molto elevata e una complessità computazionale anch'essa elevata pari a $O(2^k)$, dove il parametro k corrisponde all'hash generato: per una funzione SHA-256 avremo $k = 256$ e una complessità computazionale pari a $O(2^{256})$. Ovviamente, per risalire al valore originale si può optare per un attacco basato su forza bruta.

2.2 Transazioni in Bitcoin

In questa sezione cerchiamo di comprendere come vengono gestite le transazioni, cosa sono gli input e gli output delle transazioni e come avviene la firma e la verifica delle transazioni quando viene scambiata una quantità di bitcoin.

2.2.1 Transazioni

Le transazioni sono alla base del sistema della Blockchain sia di Bitcoin che di Ethereum, anche se differiscono per qualche caratteristica: ad esempio, in Ethereum è possibile inoltrare una transazione che contiene al suo interno un campo *data* popolato con il *bytecode*

dello smart contract eseguito sulla EVM - Ethereum Virtual Machine; parleremo di questi concetti nelle sezioni successive.

Riprendendo in mano il concetto di transazioni nel sistema Bitcoin, abbiamo a che fare con elementi che vengono inoltrati nella rete che possono essere verificati da tutti i nodi della comunità per identificare il proprietario dei bitcoin presenti nella transazione. I proprietari di bitcoin hanno la possibilità di inviare questi, firmati con la propria chiave privata, ad altri partecipanti alla rete e, quindi, autorizzare il trasferimento così da estendere la catena di proprietà.

2.2.2 Input e output delle transazioni

Quando abbiamo a che fare con le transazioni, possiamo vederle come un registro a partita doppia, in cui possiamo notare dei debiti verso il fornitore e dei crediti verso il cliente. Quando effettuiamo una transazione, stiamo trasferendo dei bitcoin e possiamo dire che ogni output precedente diventa input per tutte le transazioni successive [Ant14]. Inoltre, quando effettuiamo un pagamento, dobbiamo aggiungere dei costi di transazione comunemente identificati come ***transaction fee***, i quali verranno riscossi dal miner nel momento in cui aggiungerà la nostra transazione al blocco da validare e da propagare a tutta la comunità. Quindi, se il minatore riuscirà a risolvere il blocco con le transazioni all'interno, sarà ricompensato con dei nuovi bitcoin generati dalla prima transazione inserita da lui stesso e denominata **coinbase**, e dalla somma di tutte le *transaction fee* in ogni transazione.

Transaction as Double-Entry Bookkeeping

Inputs	Value	Outputs	Value
Input 1	0.10 BTC	Output 1	0.10 BTC
Input 2	0.20 BTC	Output 2	0.20 BTC
Input 3	0.10 BTC	Output 3	0.20 BTC
Input 4	0.15 BTC		
Total Inputs:	0.55 BTC	Total Outputs:	0.50 BTC

Inputs *0.55 BTC*
- *Outputs* *0.50 BTC*
Difference *0.05 BTC (implied transaction fee)*

Figura 2.1: Transazione come partita doppia [Ant14]

Come si nota dalla *figura 2.1* riportata, la differenza tra tutti gli input e tutti gli output corrisponde ad un valore di 0.05 *BTC*. Tale differenza corrisponde alla commissione

che verrà pagata al minatore come discusso in precedenza.

Quindi, la componente fondamentale in una transazione è l'output di transazione: tutti gli output sono visti come blocchi indivisibili, registrati sulla Blockchain e riconosciuti validi dall'intera comunità. Tali output sono denominati **UTXO - Unspent Transaction Output**, questi vengono ricevuti ogni volta che un qualche utente Bitcoin invia delle nuove criptovalute. Possiamo dire che l'intero saldo contenuto in un wallet Bitcoin, corrisponde alla totalità degli UTXO che sono stati pagati a quell'indirizzo e che l'utente potrà spendere come input di transazioni successive.

Abbiamo detto che ogni UTXO è **indivisibile**: pensiamo, ad esempio, di avere a disposizione un UTXO pari a 10 BTC. Se effettuassimo un pagamento per un determinato prodotto dal costo di 2 BTC, dovremmo mandare la totalità dell'UTXO: all'indirizzo del fornitore una quantità di 2 BTC, mentre a noi stessi una quantità di 8 BTC.

Per concludere, diciamo che un output assegna un nuovo proprietario al valore inviato, associandolo ad una chiave: ogni utente Bitcoin ha l'obbligo di firmare le transazioni affinché tutta la comunità lo veda come legittimo proprietario; quindi, gli output di una transazione possono essere utilizzati come input in una nuova transazione, creando così una catena di proprietà mentre il valore viene spostato da un indirizzo all'altro.

2.2.3 Commissioni nelle transazioni

Sappiamo che le transazioni includono un costo di commissione denominato *transaction fee*. Riprendendo il concetto visionato all'inizio della trattazione, possiamo dire che per disincentivare gli utenti ad inondare il sistema di transazioni, così come avviene ad esempio per lo spam e il sistema di HashCash, viene inserito un costo di transazione che permette di scoraggiare tale attacco. Inoltre, le commissioni fungono anche da incentivo per permettere al miner di includere la transazione all'interno di un blocco successivo da minare.

Ovviamente, tutto questo garantisce la sicurezza della rete: Satoshi Nakamoto sapeva che per avere dei validatori avrebbe dovuto incentivarli in qualche modo per avere sicurezza nel sistema. L'idea di guadagno dietro le *transaction fee* e la creazione di nuovi bitcoin permette ai miner di guadagnare una quantità di criptovaluta e, il lavoro da loro svolto, assicura all'intera comunità l'immutabilità, la validazione e la sicurezza delle transazioni inserite in un blocco della Blockchain.

2.2.4 Creazione di una catena di transazioni

Cosa molto importante da sottolineare è che un utente che possiede un wallet, non dispone direttamente di bitcoin nel proprio portafoglio, ma indirettamente: nel *wallet* sono presenti le chiavi utili per firmare la transazione e comunicare a tutti di essere il proprietario di una certa quantità di bitcoin.

Per comprendere il corretto funzionamento di una catena di transazioni, riportiamo il seguente esempio. Supponiamo che l'utente Joe inoltri una quantità pari a 0.1000 *BTC* sul conto dell'utente Alice; verrà pagata una commissione al miner per aggiungere la transazione al blocco di validazione. Successivamente, supponiamo che l'utente Alice, dopo aver ricevuto i bitcoin, inoltri una quantità di 0.0150 *BTC* sul conto dell'utente Bob, pagando anch'essa una *transaction fee*. Possiamo notare quanto descritto nella *figura 2.2*.

Figura 2.2: Catena di transazioni [Ant14]

2.2.5 Script di blocco e di sblocco per le transazioni

Entriamo ora nei dettagli che riguardano le transazioni nella Blockchain di Bitcoin. Abbiamo visto che nel momento in cui un soggetto A trasferisce bitcoin ad un soggetto B, ciò che fa è firmare digitalmente un hash della transazione precedente e la chiave pubblica del destinatario. Quest'ultimo, quando riceve i bitcoin potrà verificare le firme digitali per validare la catena di proprietà.

Gli script di transazione, in Bitcoin, per verificare e stabilire il legittimo proprietario dell'UTXO non presentano loop o funzionalità di controllo del flusso complesse oltre al semplice flusso condizionale. Abbiamo di fronte un linguaggio non Turing completo e questo significa che la complessità di tali script risulta essere limitata. Tali limitazioni non permettono attacchi DoS alla rete Bitcoin, cosa che nella Blockchain di Ethereum è stata gestita diversamente.

Nella *figura 2.3*, ripresa dal *white paper*[5] di Satoshi Nakamoto, possiamo notare come il proprietario identificato dal numero uno firmi la transazione successiva, la quale verrà inoltrata all'*utente 2*. Viene preso l'hash della transazione precedente inoltrata dall'*utente 0* all'*utente 1*, viene combinata con la chiave pubblica del destinatario (*utente 2*) e dopo aver ottenuto l' hash, viene firmato digitalmente dall'*utente 1* legittimo proprietario. Analoga operazione verrà effettuata dall'*utente 2* se utilizzerà l'UTXO ricevuto come input di una transazione [Nak08].

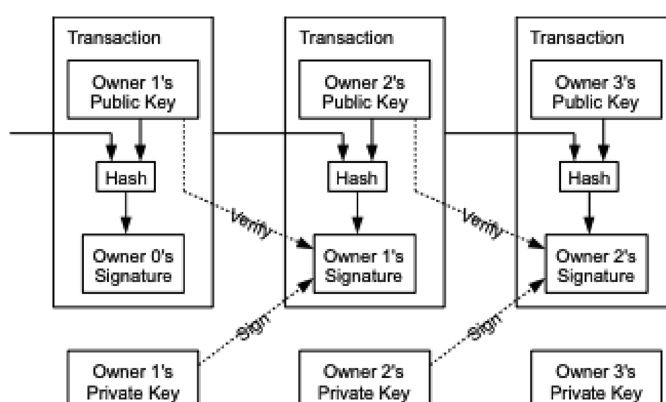


Figura 2.3: Catena di transazioni [Nak08]

Ricordiamo che sulla Blockchain troviamo tutte le transazioni generate dai vari nodi della rete Bitcoin; ognuna delle quali, inserite in un blocco validato da un utente miner. Quindi, nel momento in cui un utente vuole utilizzare una quantità di bitcoin ricevuti, dovrà attingere ad una quantità di criptovalute che possiede realmente e che ha ricevuto, magari, da altri utenti.

Guardando nel dettaglio la *figura 2.3*, supponiamo che Alice voglia inviare una quantità di criptovalute della quale dispone all'utente Bob. La prima cosa che dovrà fare sarà eseguire l'*unlocking script*, conosciuto come *scriptSig* sull'UTXO che avrà ricevuto, per sbloccare i bitcoin contenuti. Successivamente, dovrà bloccare la quantità di bitcoin

[5]Bitcoin: A peer to peer Electronic Cash System, https://bitcoin.org/bitcoin.pdf

scelti come input di transazione, utilizzando il *locking script o scriptPubKey* e rappresentando tali bitcoin come "output" per Bob. Infine, Alice applicherà la propria firma digitale sulla transazione, la quale verrà inoltrata a tutti i peer presenti nella rete Bitcoin.

Ciò che accade a basso livello è utilizzare il linguaggio di scripting di Bitcoin, il quale è chiamato *linguaggio basato su stack*[6] perchè utilizza una struttura dati a pila per validare le transazioni. Di seguito sono elencate le varie operazioni che verranno effettuate:

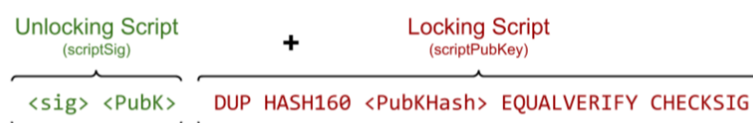


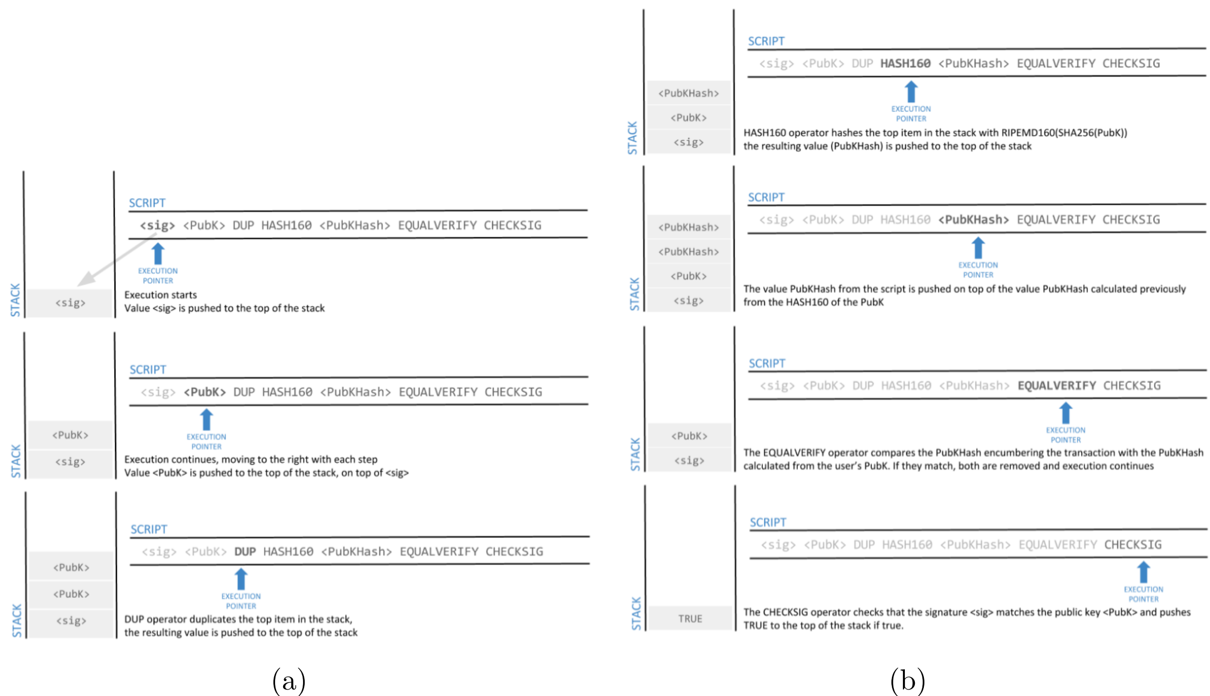
Figura 2.4: Script di sblocco e blocco [Ant14]

- viene effettuata l'operazione *push* dello *scriptSig*: vengono inserite nello stack i rispettivi valori di <*sig*> e <*PubK*>.
- Inseriamo nello stack lo *scriptPubKey* eseguendo la prima operazione sullo stack *OP_DUP*. Tale operazione ci permette di duplicare l'elemento in testa allo stack; più precisamente: viene estratto il primo elemento, viene duplicato e viene eseguito *push* del risultato.
- La seconda operazione sullo stack è *OP_HASH160*: esegue *pop* del primo elemento, applica le due funzioni hash utilizzate in Bitcoin, vale a dire *SHA256* e *RIPEMD160* ed esegue nuovamente *push* del risultato ottenuto.
- Successivamente viene inserito nello stack <*PubKHash*>, cioè l'hash della chiave pubblica del destinatario.
- Viene eseguita la chiamata all'operazione *OP_EQUALVERIFY* che effettua due volte *pop* dallo stack e confronta i valori: se sono uguali si passa all'operazione successiva, altrimenti lo stack fallisce.
- Infine, l'ultima operazione è *OP_CHECKSIG* che si occupa di verificare la firma: estrae i due valori rimasti nello stack, cioè <*sig*> e <*PubK*>, esegue la verifica della firma e se tutto va a buon fine inserisce il valore 1 (true) nello stack, altrimenti inserisce il valore 0 (false).

[6]Per questioni legate alla sicurezza in Bitcoin, dal 2010 non è possibile eseguire i due script in sequenza. Tutto questo è legato ad una vulnerabilità che permetteva di compromettere lo script di blocco utilizzando uno script di sblocco non valido.

Considerazione importante da fare è che l'utente che riceve una quantità di bitcoin spendibili, per i quali riesce ad eseguire lo script di sblocco, potrà utilizzarli solo se la transazione precedentemente eseguita è stata inserita in un blocco validato da un miner e che, per tale blocco, siano occorse almeno sei conferme. Questo vuol dire che la relativa quantità di bitcoin potrà essere spesa se e solo se altri miner avranno convalidato altre transazioni inserite in altri blocchi, per un totale di sei blocchi sopra il blocco contenente la transazione corrente.

Tale logica, permette di avere sicurezza nel sistema: la complessità computazionale per permettere ad un possibile attaccante di modificare la transazione contenuta nel blocco validato, dopo che sono avvenute sei conferme, è troppo elevato e modificare una transazione all'interno della Blockchain, ricalcolando tutti gli hash dei blocchi precedenti e successivi, risulterebbe inutilmente complesso se non impossibile. Di seguito la *figura 2.5* permette di comprendere i passaggi descritti nell'elenco precedente:



(a) (b)

Figura 2.5: Esecuzione degli script [Ant14]

2.3 Generazione di chiavi e indirizzi

Come enunciato nella sezione precedente, il sistema Bitcoin utilizza diverse funzioni di hash per combinare la catena di transazioni, combinare blocchi validati dai miner e altre

operazioni; in questa sezione cercheremo di spiegare come vengono generate le chiavi e i relativi indirizzi in Bitcoin per: assicurare immutabilità e confidenzialità nell'intero sistema.

2.3.1 Crittografia a chiave pubblica (crittografia asimmetrica)

A differenza dell'algoritmo RSA, Bitcoin utilizza l'algoritmo ECDSA (Elliptic Curve Digital Signature Algorithm), cioè la crittografia a curva ellittica. Per generare una chiave privata è possibile utilizzare un generatore di numeri casuali o, per avere maggiore sicurezza nella creazione di questa, è possibile generare una stringa casuale molto grande e passarla ad una funzione hash come SHA256.

Per generare una possibile chiave privata, quindi, bisogna fare in modo che essa sia lunga più di 256 bit e che sia compresa tra $[1, n-1]$. Stiamo dicendo che, presa una stringa generata casualmente più lunga di 256 bit, la passiamo alla funzione hash SHA256, la quale ci restituisce una stringa di lunghezza 256 bit; il valore di n da rispettare è definito come una costante denominata ***ordine della curva ellittica***:

$$n = 1.158 \cdot 10^{77}, \textit{un pó meno di } 2^{256} \tag{2.1}$$

La generazione della chiave pubblica avviene nel modo seguente: partendo dalla chiave privata, è possibile applicare l'algoritmo ECDSA per ricavare una nuova chiave utilizzando operazioni di raddoppio e somma di punti, partendo da un punto base. Viene utilizzato un punto denominato *punto generatore o di generazione* che, moltiplicato per la chiave privata generata casualmente, permette di ottenere una nuova chiave.

$$K_{pub} = k_{priv} \cdot G \tag{2.2}$$

Tale punto di generazione, è specificato dallo standard ***secp256k1*** e tutte le chiavi generate usano lo stesso punto G. La moltiplicazione di $k \cdot G$, equivale a ripetere l'addizione di $G + G + ... + G$ per k volte e, applicando quanto descritto, otteniamo due coordinate che rappresentano il punto K.

Di seguito riportiamo alcune definizioni e un esempio per comprendere meglio il funzionamento di generazione di una possibile chiave pubblica (maggiori approfondimenti in [Fab]):

Definition 2.3.1 (Somma tra due punti sulla curva ellittica). *Si definisce somma fra due punti P, Q su una curva ellittica il punto: $P + Q = R$ dove, dati $P = (x_P, y_P), Q = (x_Q, y_Q), R = (x_R, y_R)$, vale che:*

$$x_R = m^2 - x_P - x_Q \tag{2.3}$$

$$y_R = m(x_P - x_R) - y_P \tag{2.4}$$

con

$$m = \frac{y_Q - y_P}{x_Q - x_P} \tag{2.5}$$

Definition 2.3.2 (Punto doppio)**.** *Si definisce punto doppio il punto $2P = P + P = R$ dove, dati $P = (x_P, y_P)$ e $R = (x_R, y_R)$ vale che:*

$$x_R = m^2 - 2x_P \tag{2.6}$$

$$y_R = m(x_P - x_R) - y_P \tag{2.7}$$

con

$$m = \frac{3x_P^2 + a}{2y_P} \tag{2.8}$$

Per comprendere meglio quanto enunciato, possiamo effettuare un esempio su una curva ellittica generica, scegliendo dei valori semplici per la riuscita dell'esempio.

Esempio 1.
Supponiamo di avere a disposizione i seguenti dati:

- Modulo: 67
- Punto di generazione G: (2, 22)
- Ordine n: 79
- k_{priv}: 2

Come notiamo dai dati, abbiamo preso una chiave privata molto semplice e per questo effettuiamo un'operazione riprendendo la definizione di punto doppio riportato precedentemente. Per cercare la chiave pubblica corrispondente, bisogna calcolare il valore di m:

$$m = \frac{3 \cdot 2^2 + 0}{2 \cdot 22}\, mod\, 67 \tag{2.9}$$

$$m = \frac{12}{44}\, mod\, 67 \tag{2.10}$$

$$44^{-1} \cdot 12\, mod\, 67 \tag{2.11}$$

Ovviamente, se provassimo ad effettuare il modulo dell'operazione sopra indicata, non avremmo un valore intero. Per risolvere tale problematica, dobbiamo trovare il

reciproco del valore 44 nel campo finito. Per fare ciò, bisogna trovare quel numero x che, moltiplicato per il valore 44, eguaglia 1 mod 67.

$$44 \cdot x = 1 \; mod \; 67 \tag{2.12}$$

$$44 \cdot 32 = 1 \; mod \; 67 \tag{2.13}$$

L'unico valore che moltiplicato per 44 rispetta la condizione è il valore 32; eseguiamo, ora, i passi successivi per ottenere il valore di m.

$$m = 12 \cdot 32 \; mod \; 67 = 384 \; mod \; 67 = 49 \tag{2.14}$$

A questo punto, possiamo ricavare i valori di x_R e y_R:

$$x_R = 49^2 - 2 \cdot 2 \; mod \; 67 = (2401 - 4) \; mod \; 67 = 52 \tag{2.15}$$

$$y_R = 49(2 - 52) - 22 \; mod \; 67 = (-2450 - 22) \; mod \; 67 = -2472 \; mod \; 67 = 7 \tag{2.16}$$

Ricordiamo che, per trovare il valore in modulo di un numero negativo, dobbiamo eseguire l'operazione che segue:

$$-2472 \; mod \; 67 = -2472 + 67 \cdot x = -2472 + 67 \cdot 37 = 7 \tag{2.17}$$

Possiamo concludere che la chiave pubblica risulta essere $(52, 7)$.

2.3.2 Secp256k1

Come già detto, Bitcoin utilizza la crittografia a curva ellittica che rispetta lo standard *secp256k1*, stabilita dal NIST (National Institute of Standards and Technology[7]). Tale curva è definita nella maniera che segue dalla seguente funzione:

[7]NIST: https://www.nist.gov/

$$y^2 = x^3 + 7 \tag{2.18}$$

oppure

$$y^2 \ mod \ p = (x^3 + 7) \ mod \ p \tag{2.19}$$

2.3.3 Indirizzi in Bitcoin

Trattiamo, ora, le due funzioni hash che vengono utilizzate in Bitcoin per generare l'indirizzo sul quale un nodo della rete può ottenere o ricevere eventuali bitcoin. Le due funzioni sono SHA (Secure Hash Algorithm) e RIPEMD (Race Integrity Primitives Evaluation Message Digest).

Alla creazione di un wallet Bitcoin, vengono generate le chiavi: privata, pubblica e l'indirizzo Bitcoin. Tale indirizzo, viene generato facendo passare la chiave pubblica prima nella funzione *SHA256* e poi nella funzione *RIPEMD160*, producendo un valore a 160 bits (o 20 bytes). Per avere una maggiore leggibilità dell'indirizzo, viene fatto passare per la codifica *Base58Check*[8]: utilizza un checksum di 4 byte aggiunti alla fine dei dati che vengono codificati e viene utilizzato per prevenire errori di scrittura o battitura [Ant14] [GM18]; vediamo insieme il funzionamento.

Come prima operazione, viene aggiunto un prefisso ai dati denominato *"version byte"*, esso server per identificare in modo semplice i dati che vengono codificati: per un indirizzo Bitcoin si utilizza il prefisso *0x00*. Il prossimo passo è quello di utilizzare per due volte la funzione SHA256 passando i relativi dati:

$$checksum = SHA256\ (SHA256\ (prefisso + dati))$$

Infine, vengono presi gli ultimi 4 byte della codifica e vengono aggiunti alla fine dei dati a disposizione.

Come per gli indirizzi, anche le chiavi pubbliche possono essere di due tipi: normali o compresse. Questo secondo caso è stato introdotto per ridurre la grandezza delle transazioni, risparmiando spazio sui nodi che inglobano internamente l'intera blockchain.

La *figura 2.6* chiarisce il funzionamento della codifica in *Base58*:

[8]https://tools.ietf.org/id/draft-msporny-base58-01.html

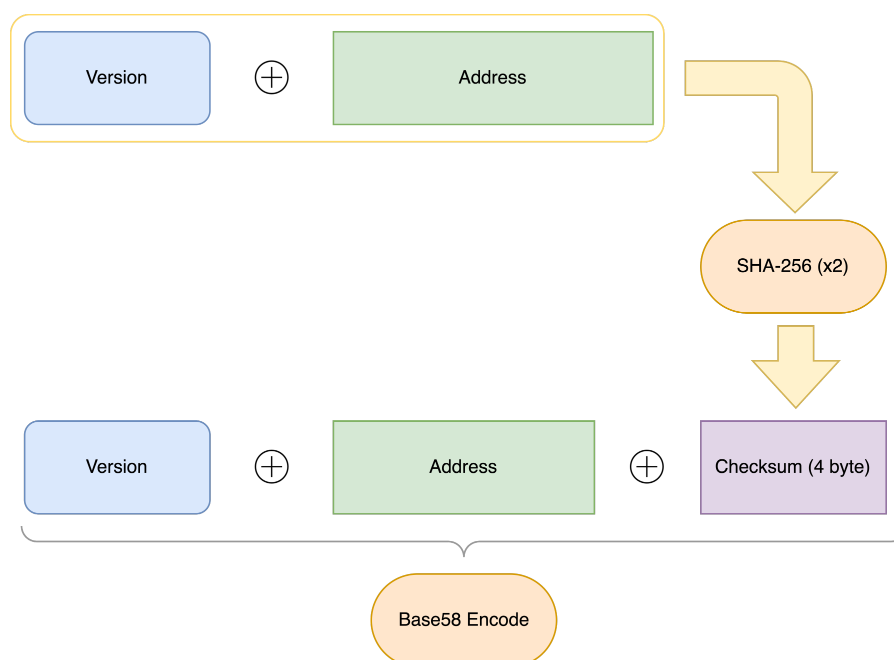


Figura 2.6: Codifica Base58

2.4 La rete e i nodi

Quando parliamo di Bitcoin, abbiamo a che fare con una struttura Peer-to-Peer (P2P), non avendo un'autorità centrale e con tutti i nodi partecipanti alla rete autonomi e liberi da qualsiasi vincolo. Tutti i nodi si connettono ad altri nodi formando così una rete che sia: resiliente, decentralizzata e soprattutto aperta a tutti.

I vari nodi che possiamo trovare all'interno della rete possono assumere diversi ruoli, tra cui: routing, database blockchain, miner o wallet. Tutti i nodi eseguono la funzione di *routing* per partecipare alla rete, altrimenti non potrebbero propagare le varie transazioni, connettersi agli altri peer e visualizzare i blocchi di transazione e, quindi, l'intera Blockchain.

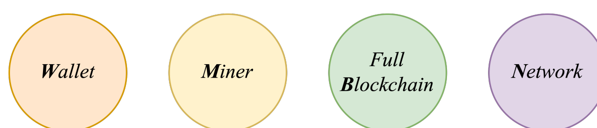


Figura 2.7: Nodi nella rete Bitcoin

Ci sono varie tipologie di nodi nella rete Bitcoin: i nodi completi, mantengono un riferimento ad una copia completa di tutta la Blockchain, possono verificare le varie

transazioni autonomamente, senza collegarsi a servizi esterni. Altri nodi, invece, sono leggeri, cioè si occupano solo di implementare il routing e gestiscono il proprio wallet, non avendo una cronologia di tutte le transazioni e quindi una copia dell'intera Blockchain; per avere ciò, sono obbligati a consultare servizi esterni.

Possiamo anche trovare altri nodi che hanno un riferimento alla Blockchain, svolgono il servizio di routing e convalidano le transazioni cercando di risolvere il problema di creazione di un nuovo blocco da aggiungere alla Blockchain per: ricevere come premio una quantità di bitcoin come pagamento, assieme alle *transaction fee* di tutte le transazioni per blocco. Tali nodi vengono chiamati Miner.

2.4.1 Network discovery

Ogni nuovo nodo che partecipa alla rete Bitcoin deve comunicare a tutti gli altri di essere entrato. Come prima cosa, il nodo entrante deve cercare almeno un nodo della rete e connettersi, inviando un cosiddetto *"message version"* o messaggio di versione, stabilendo una connessione TCP (porta 8333). Se il peer contattato ha intenzione di collegarsi con il peer che ha effettuato la richiesta, allora risponderà con una propria versione, inviando un *verack* che riconosce e stabilisce una nuova connessione nella rete.

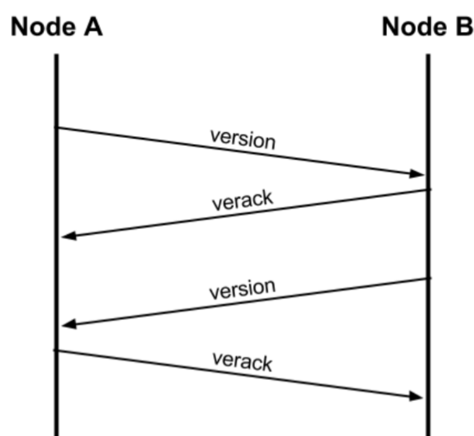


Figura 2.8: Richiesta di entrata nella rete

Dopo che la prima connessione è avvenuta con successo, bisogna fare in modo che il nuovo nodo venga conosciuto dall'intera rete. Per fare ciò si sfrutta la rete peer-to-peer: il nodo che ha risposto con successo alla chiamata dell'entrante comunica ai nodi vicini tale entrata, i quali, a loro volta eseguiranno la stessa operazione. Una volta stabilite una o più connessioni, il nuovo nodo invia un messaggio, ai nodi ai quali si è collegato, inoltrando il proprio indirizzo IP, allo stesso modo i nodi che ricevono tale messaggio, lo

inoltrano agli altri nodi ai quali sono collegati; così facendo il nuovo nodo viene conosciuto dall'intera rete. Un nuovo nodo ha due modi per trovare altri peer ai quali connettersi:

1. Utilizzare il metodo attraverso il quale è possibile consultare degli indirizzi IP di nodi Bitcoin stabili.

2. Prendere l'ultimo indirizzo IP dell'ultimo nodo Bitcoin.

È molto importante che il nodo entrante, continui a scoprire nuovi nodi nella rete: i nodi in Bitcoin vanno e vengono; quindi, c'è la possibilità di perdere le vecchie connessioni. Inoltre, il nuovo nodo entrante potrà fare da tramite e assicurare l'entrata ad altri nodi.

2.4.2 Nodi completi o Full Node

Tutti i nodi all'interno della Blockchain hanno lo stesso peso e vengono creati tutti allo stesso modo. Un nodo completo è più dispendioso da eseguire: i dati relativi a maggio 2020 comunicano che un *"Full Node Bitcoin"*, doveva avere a disposizione una quantità di memoria pari a 277 GB per immagazzinare l'intera Blockchain [GM18], ed è comprensibile che non tutti gli utenti siano in grado di disporre di risorse sufficienti per l'archiviazione e la verifica.

I *Full Node* hanno diverse connessioni sia in entrata che in uscita: possono chiedere di connettersi ad altri utenti, oppure possono essere interpellati da utenti che, nuovi nella rete, hanno bisogno di effettuare delle connessioni per pubblicizzare la loro entrata.

2.4.3 Nodi leggeri

I nodi leggeri[9], come introdotto, non tengono traccia dell'intera Blockchain come i full node, ma tengono un riferimento ad essa solo per recuperare le transazioni, e i dati, di cui hanno bisogno. Per fare ciò stabiliscono una connessione ai full node e possono effettuare solo connessioni verso l'esterno.

Un nodo leggero, per recuperare tutte le informazioni necessarie, dovrà verificare che una specifica o più transazione si trovi al di sotto di almeno sei blocchi per essere certo che essa non possa più essere modificata; mentre un nodo completo si occupa di collegare tutti i blocchi presenti nella Blockchain al blocco genesi, cioè il primo blocco di altezza uno. Successivamente, dovrà inviare una richiesta esplicita per recuperare le intestazioni dei blocchi e il peer che risponderà potrà inoltrare fino ad un numero massimo di 2000 intestazioni.

[9]SPV - Simplified Payment Verification, verifica la catena di blocchi e non la catena delle transazioni.

Per evitare di appesantire la memoria di transazioni richieste, un nodo leggero utilizza un filtro denominato *Bloom filter*, attraverso il quale filtra tutti i possibili blocchi di transazione di cui necessita, cercando di mascherare l'indirizzo che sta cercando: in altre parole, seleziona gli elementi di interesse senza però rivelare a quale sia direttamente interessato (per eventuali approfondimenti sul funzionamento, consultare [Ant14]).

Ciò che avviene è inserire nel bloom filter gli indirizzi o pattern di interesse contenuto in un wallet di un nodo SPV: se un peer nella rete trova una transazione che rispetta la condizione del bloom filter, invia l'intestazione del blocco, che contiene la transazione di interesse, al peer che ha lanciato il filtro. Infine, il nodo ricevente, verifica che la transazione si trova nel blocco indicato e aggiunge tale transazione al blocco, senza la necessità di ricostruire l'intera catena.

Dimostrazione sul funzionamento di un *bloom filter*

Il bloom filter è una struttura dati utilizzata per rappresentare uno specifico insieme di n elementi $E = \{e1, e2, ..., en\}$. Tale struttura dati è formata da un array contenente un numero massimo di m bit e da k funzioni hash indipendenti tra loro $H = \{h1, h2, ..., hk\}$ e, le cui uscite, sono distribuite in modo uniforme nel range {0, 1, 2, ..., m - 1}.

Supponiamo che un nodo SPV applichi un pattern s_i di ricerca utilizzando un bloom filter settato inizialmente con tutti valori 0. Come prima cosa, verrà fatto passare tale valore (che rappresenta uno specifico indirizzo contenuto nel wallet) all'interno delle k funzioni hash, le quali restituiranno ognuno un valore indipendente che verrà utilizzato per settare l'array degli $m - 1$ elementi: $h_1(s_i), h_2(s_i), ..., h_k(s_i)$.

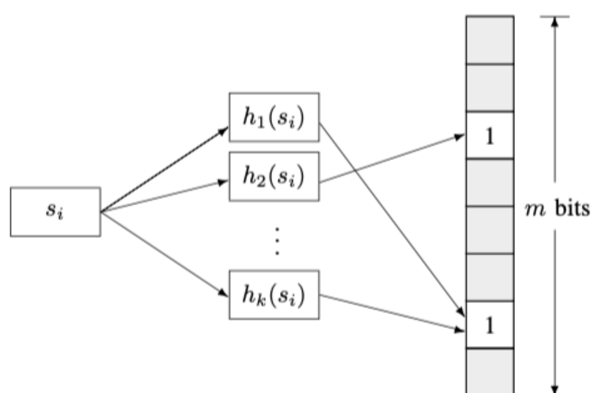


Figura 2.9: Funzionamento Bloom filter [RD14]

Successivamente, verranno settate le corrispondenze dei valori generati dalle funzioni hash al valore 1; questo porta ad una criticità: se la maggior parte delle posizioni nell'array sono già state settate, allora si avranno dei *falsi positivi*; per ovviare a tale problema, si cerca di utilizzare un array molto grande e, quindi, con un gran numero di posizioni settabili (approfondimenti [RD14] [AC14]). Ovviamente, la presenza di valori già settati al valore 1 ci indica che l'elemento che si sta prendendo in considerazione "potrebbe" trovarsi già in memoria, ma non possiamo affermarlo con certezza.

2.5 Il consenso

Quando si tratta di consenso in un sistema distribuito e decentralizzato come quello della Blockchain di Bitcoin, la domanda è: come possiamo raggiungere un accordo definitivo, che sia uguale per tutti, sullo stato del sistema? Come possiamo raggiungere un consenso comune? Per rispondere a questa domanda, viene preso in considerazione il problema noto come ***problema dei Generali Bizantini*** [AC20].

2.5.1 Problema dei generali Bizantini

Tale problema pone l'attenzione su un gruppo di generali Bizantini che vogliono mettersi d'accordo su un eventuale attacco ad un castello: tutti i generali possono usufruire di messaggeri per inviare vari messaggi ad altri generali. Quindi, il compito di ogni generale è quello di inoltrare i messaggi agli altri $n - 1$ generali, ma questo non vieta che generali malintenzionati possano inviare più di un messaggio per confondere le idee su un possibile attacco o meno (tutte le dimostrazioni e il materiale proposto possono essere consultati visionando l'articolo [LP82]).

Impossibilità del problema

Se i generali comunicassero solamente attraverso messaggi orali, non ci sarebbe una soluzione e quindi un consenso comune, questo a meno che i $\frac{2}{3}$ dei generali non siano leali; prendiamo in considerazione solo due possibili decisioni: "attacco" o "ritirata" e consideriamo le seguenti condizioni.

1. Tutti i fedeli luogotenenti obbediscono allo stesso ordine.
2. Se il comandante è leale, allora ogni fedele luogotenente obbedisce all'ordine che invia.

Di seguito, riportiamo due semplici esempi che possono essere consultati nell'articolo [LP82].

Esempio 1. Supponiamo di avere inizialmente un comandante fedele che comunica al luogotenente 2 un ordine di attacco, a questo punto il luogotenente 2 ricopre la figura del malintenzionato e comunica al luogotenente 1 un ordine di ritirata. Allo stesso modo, il comandante comunica al luogotenente 1 un ordine di attacco e, in questo scenario, il luogotenente 1 obbedirà all'ordine del comandante (secondo il punto 2 riportato nelle condizioni), senza sapere chi tra esso e il luogotenente 2 sia il malintenzionato.

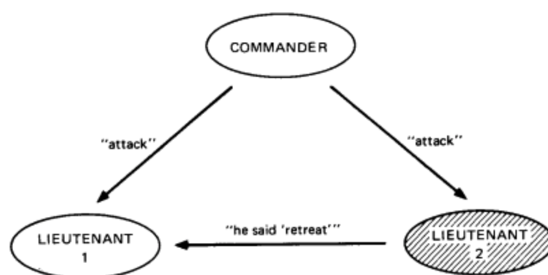


Figura 2.10: Comandante leale nello scenario Bizantino [LP82]

Esempio 2. Supponiamo in questo esempio che il comandante sia un malintenzionato e comunica al luogotenente 1 di attaccare e al luogotenente 2 l'ordine di ritirata. Anche in questo caso, il luogotenente 1 non conosce l'identità del traditore e non sa effettivamente quale messaggio sia stato mandato al luogotenente 2.

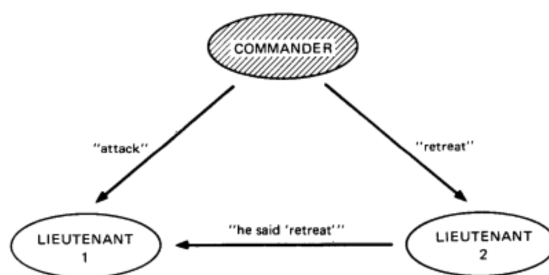


Figura 2.11: Comandante malintenzionato nello scenario Bizantino [LP82]

Come si nota dai due esempi riportati, se esiste un traditore nel sistema non è possibile per il luogotenente 1 distinguere le due situazioni, obbedendo sempre al comandante. Inoltre, se il luogotenente 2 riceve l'ordine di ritirata dal comandante malintenzionato dovrà rispettare quanto detto: se comunicherà al luogotenente 1 la ritirata, quest'ultimo obbedirà al comandante non soddisfacendo la prima condizione.

Per le soluzioni e dimostrazioni discusse dagli autori, rimandiamo al seguente articolo [LP82].

2.5.2 Consenso decentralizzato

Secondo Satoshi Nakamoto, il problema dei generali Bizantini può essere riproposto nel sistema Bitcoin risolvendolo. Ogni nodo nella rete è paragonato ad un generalo bizantino e, ovviamente, ci possono essere generali e messaggeri malintenzionati e il sistema deve assicurare di arrivare ad un accordo comune nonostante questi. Il problema più grande di tale soluzione è il seguente: più partecipanti alla rete ci sono e maggiore deve essere il numero di messaggi per il consenso scambiati; in questo modo il tempo di esecuzione aumenta in modo quadratico con il numero di utenti nel sistema.

La soluzione proposta permette a tutti gli utenti di essere validatori onesti: il primo che trova la soluzione basata su hash del blocco successivo, potrà prendere una decisione. In Bitcoin tale sistema è denominato Proof-of-Work e si basa sull'utilizzo di hardware molto efficiente che permetta di proporre in un solo secondo 15˙000˙000 di hash possibili; con un dispendio di energia molto elevato per qualsiasi attaccante.

Affinché ci sia consenso, bisogna che si verifichino quattro eventi:

- Tutte le transazioni sulla rete devono essere verificate in maniera indipendente da ogni *full node*.
- Aggiunta delle transazioni verificate ad un blocco da validare sfruttando l'algoritmo di *Proof-of-Work*.
- Verifica di nuovi blocchi da aggiungere alla catena da parte di ogni nodo.
- Scelta della catena con il maggior numero di nodi, o con l'altezza maggiore nell'intera storia.

Il mining

I nodi presenti nella blockchain che si occupano di validare le transazioni sono chiamati nodi *miner*. Tali nodi, oltre alla verifica delle transazioni, si occupano di inserire queste all'interno di una *memory pool* o *transaction pool*, in attesa di aggiungerle ad un *candidate block* o *blocco candidato*. Ovviamente, non tutte le transazioni presenti nella *transaction pool* vengono inserite in un blocco, le rimanenti saranno aggiunte al blocco successivo tenendo in considerazioni le *transaction fee*.

Le transazioni vengono scelte tra quelle con priorità maggiore: età maggiore e con un valore alto rispetto agli input più piccoli e più "giovani"; inoltre, le transazioni con priorità più alta rispetto alle altre, devono avere un valore intero che sia maggiore di 57.600.000.

$$\frac{100.000.000\ satoshi * 144\ blocchi}{250\ byte} = 57.600.000$$

Nel processo di mining ogni blocco viene generato, in media, ogni dieci minuti. Per cercare di rispettare tale tempistica viene monitorato costantemente un parametro di difficoltà denominato *retargeting*: ogni 2016 blocchi, tutti i nodi settano nuovamente la difficoltà, la quale misura il tempo impiegato per trovare gli ultimi 2016 blocchi e li confronta con il tempo previsto di 20160 minuti.

$$Nuovo\ target = vecchio\ target \cdot \frac{Tempo\ corrente\ per\ aver\ trovato\ gli\ ultimi\ 2016\ blocchi}{20160\ minuti}$$

Evento di Halving

Nel sistema decentralizzato, il numero massimo di bitcoin che possono essere estratti o minati corrisponde a 21 milioni. Ogni 210.000 blocchi oppure ogni 4 anni, tale fornitura viene dimezzata attraverso un evento denominato *halving*. Tale evento dimezza in maniera costante il rilascio di nuovi bitcoin nella rete, influenzando il tasso di emissione e la quantità di criptomonete in circolazione.

Nel momento di scrittura di questa tesi, la quantità di bitcoin che viene rilasciata al miner che risolve e distribuisce il blocco di transazioni è pari a 6.25 BTC e, come descritto nel paragrafo precedente, tale distribuzione regolare è permessa anche grazie al *retargeting*.

Capitolo 3

Ethereum

In questo capitolo verrà illustrato, in breve, il funzionamento di Ethereum e le caratteristiche aggiuntive portate da Vitalik Buterin. Questo, nel 2013, cercò di presentare a tutta la comunità un protocollo innovativo che permettesse di sfruttare le potenzialità di una rete peer-to-peer come in Bitcoin, ma con la capacità di sviluppare smart contract e applicazioni decentralizzate.

Come accennato nell'introduzione a questa tesi, dopo la pubblicazione del *white paper* [But13], un programmatore chiamato Gavin Wood pubblicò uno *yellow paper* [Woo14] di risposta a Vitalik Buterin, in cui descrisse la macchina a stati basata su transazioni.

3.1 Introduzione ad Ethereum

Ethereum può essere presentata come una piattaforma con un sistema di esecuzione condiviso, la Ethereum Virtual Machine. Stiamo parlando di una piattaforma decentralizzata che gestisce smart contract ed è basata su un registro distribuito di tipo *permissionless*.

Nel 2016 la piattaforma venne divisa in: *Ethereum Foundation* e *Ethereum Classic*. La prima ha il compito di gestire le applicazioni decentralizzate (*DApp*) ed occuparsi della fase di sviluppo, supporto e di ricerca. Il sistema distribuito è basato sul consenso Proof-of-Work come in Bitcoin, ma esattamente due anni fa (nel 2020) venne rilasciata la versione Ethereum 2.0 attraverso la *Beacon Chain* che ha permesso l'inizio della migrazione all'algoritmo di consenso *Proof-of-Stake*, del quale tratteremo nelle sezioni successive. La seconda, Ethereum Classic, si occupa di limitare i vari rischi nel sistema e ha lo scopo di limitare il più possibile i rischi di deflazione della criptovaluta, gestendo l'emissione dei token in proporzione alla crescita della piattaforma.

3.1.1 La criptovaluta in Ethereum

Nella piattaforma Ethereum troviamo una nuova criptovaluta denominata *Ether*. Tale moneta virtuale ha un duplice compito: permette di effettuare transazioni tra account, smart contract o tra account e smart contract, e permette l'acquisto del ***gas*** per pagare le varie commissioni. Eseguire uno smart contract, vuol dire pagare una quantità specifica di gas a seconda delle operazioni che vengono riportate sul codice; nello *yellow paper*[Woo14] si trovano i costi delle varie operazioni.

Quindi, la differenza con bitcoin è che l'*ether* può essere utilizzato anche come mezzo di pagamento all'interno del sistema Ethereum, oltre che ad essere utilizzato come moneta virtuale di scambio tra i vari account e smart contract. Effettuando un velocissimo paragone, possiamo dire che il validatore in Ethereum potrà scegliere, così come in Bitcoin, di includere o meno la transazione all'interno del *candidate block*: maggiore è la quantità di gas in eccesso inviata alla transazione, più si ha possibilità che tale transazione venga validata velocemente.

3.1.2 Ethereum Virtual Machine - EVM

L'EVM in Ethereum rappresenta l'ambiente in cui sviluppare ed eseguire il *bytecode* degli smart contract. Attraverso il processo di virtualizzazione, viene eseguita una macchina fisica e, nello specifico, EVM è una macchina a stati, single thread e strutturata a *stack*.

Grazie all'EVM è possibile verificare la quantità di gas fornita ad una transazione per l'esecuzione di uno smart contract, eseguendo il codice fornito oppure interrompendo l'esecuzione evitando di aggiornare lo stato di Ethereum; se in modo accidentale o voluto, la quantità di gas fornita dovesse scendere a 0, l'EVM restituirà un'eccezione *"Out-of-Gas (OOG)"*, verrà interrotta l'esecuzione e abbandonata la transazione presa in carico.

Problema dell'arresto in Ethereum

Alan Mathison Turing (1912 - 1954) è l'ideatore della macchina Universale di Turing e ha fornito contributi anche alla crittoanalisi, aiutando durante la Seconda Guerra Mondiale. Turing aveva affrontato il problema della calcolabilità e se le macchine potessero avere un comportamento intelligente; pubblicando nell'ottobre del 1950 uno studio sulla rivista *Mind*.

L'idea di Turing era studiare i limiti della calcolabilità e, attraverso il problema dell'arresto, Turing dimostrò che esistono delle funzioni che non possono essere calcolate dalla MdT (Macchina di Turing). Tutto questo è stato affrontato e "risolto" attraverso

l'utilizzo del gas nella EVM.

Alcuni programmi impiegano un'eternità per essere eseguiti, ma non possiamo affermare con certezza se, dato un programma, esso porterà a termine la propria esecuzione o meno se non lo eseguissimo. Per rispondere a tale problema, Vitalik Buterin assieme al cofondatore Gavin Wood, decisero di introdurre il gas per ogni operazione effettuata da uno smart contract sulla EVM.

3.1.3 Gas

Come letto dal paragrafo precedente, il gas è l'unità di Ethereum in grado di misurare le risorse computazionali per eseguire azioni sulla blockchain. Ogni operazione presenta uno specifico costo in unità di gas. La logica del gas impedisce che uno smart contract rimanga nello stato di esecuzione per un tempo indefinito: questo riduce i possibili attacchi DoS, in quanto il codice terminerà la propria esecuzione una volta terminata la quantità di gas fornita alla transazione.

$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT 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.
$G_{callvalue}$	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 SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by the number of bytes in the exponent 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.

Figura 3.1: Costo del gas [Woo14]

Riprendendo il concetto esposto all'inizio della trattazione, possiamo affermare che il gas disincentiva i malintenzionati dall'invio di transazioni malevole, poichè dovranno pagare in proporzione alle risorse di calcolo utilizzate, memoria che consumano e qualsiasi altra operazione che effettueranno.

Ogni volta che viene eseguita una transazione, l'utente inserisce una sorta di mancia per il miner che aggiungerà la transazione al blocco e, se l'EVM esegue con successo tutte le operazioni senza esaurire il gas fornito, la quantità in eccesso di gas verrà pagata al miner come commissione sulla transazione e convertito in ether in base al prezzo del gas. Mentre, il gas rimanente verrà riconsegnato al mittente sempre effettuando prima la conversione in ether.

$$Commissione\ miner = prezzo\ del\ gas \cdot costo\ del\ gas$$

Nel dettaglio possiamo vedere in cosa consiste il gas inserito in una transazione:

- Costo del gas: è l'unità di gas necessaria ad eseguire una o più operazioni per conto di uno smart contract.
- Prezzo del gas: corrisponde alla quantità di ether che si è disposti a pagare per unità di gas quando viene inviata la transazione su Ethereum.

Inoltre, possiamo trovare in ogni blocco una certa quantità di gas o, anche definita, *gas limit*; questo limite vincola i miner ad inserire un numero di transazioni per blocco che non consumino tutto il gas limit del blocco (maggiori approfondimenti relativi al gas in [AW18] [Woo14]).

3.2 Generazione delle chiavi

La generazione delle chiavi avviene sempre sfruttando l'algoritmo basato sulle curve ellittiche. L'unica differenza è che per generare l'indirizzo Ethereum partendo dalla chiave pubblica, viene applicato l'algoritmo di hashing *Keccak-256* prendendo gli ultimi 20 bytes (byte meno significativi a destra). La rappresentazione degli indirizzi che possiamo visualizzare su un wallet come MetaMask oppure attraverso delle transazioni, è fatta con una codifica esadecimale: 40 bit per ogni indirizzo che identificano un account oppure uno smart contract.

3.3 Transazioni Ethereum

A differenza di Bitcoin, in Ethereum le transazioni non vengono utilizzate solo per scambiare una certa quantità di criptovalute tra più account, ma anche per eseguire un nuovo smart contract da parte di un EOA, aggiungendo il bytecode alla transazione, oppure per effettuare richieste su uno smart contract esistente (approfondimenti sui costi di transazioni in [XS18]). Inoltre, si ha la possibilità di far comunicare più smart contract gestendo la quantità di ether da trasferire.

3.3.1 EOA e Smart contract

Come si nota dalla descrizione sopra riportata, in Ethereum possiamo avere due tipi di account: un account *EOA (Externally Owned Account)* che possiamo paragonare ad un semplice account in Bitcoin, e uno smart contract o *CA (Contract Account)*, il quale non possiede una chiave privata per firmare le varie transazioni, ma il tutto viene gestito dal codice in esso contenuto. Cerchiamo di capire come EVM gestisce a basso livello i vari indirizzi nella rete.

Possiamo associare ad ogni indirizzo Ethereum valido uno specifico account, oppure uno smart contract; le parti che compongono un account sono le seguenti:

- Il bilancio degli ether che dispone (espresso in Wei, dove 1 ETH corrisponde a 10^{18} Wei).
- Un *nonce*, cioè un campo intero che rappresenta il numero di transazioni inviate con successo e registrate sulla blockchain di Ethereum (il *nonce* previene i *reply-attacks*, quindi non è possibile mandare una transazione con un nonce maggiore se la transazione corrente non è stata registrata sulla blockchain).
- Deposito o archivio di dati permanente, utilizzato dagli smart contract e per un EOA sarà vuoto.
- Codice dello smart contract; un EOA avrà lo spazio riservato al codice sempre vuoto.

Invio di ether ad un indirizzo valido

È possibile, in Ethereum, inviare una certa quantità di *ether* ad un indirizzo valido: un qualsiasi indirizzo rappresentato da una stringa a 20 byte. La cosa da comprendere è che, se vengono inoltrati degli *ether* ad un indirizzo che non è associato ad un EOA; quindi, non si ha una chiave privata per sbloccare gli ether, oppure ad uno smart contract, non sarà possibile usarli e rimarranno bloccati. Quando quel determinato indirizzo verrà assegnato, allora si potranno utilizzare gli ether che erano stati inviati con la transazione precedente.

3.3.2 Come è strutturata una transazione

Una transazione in Ethereum è strutturata nel modo che segue:

- Nonce: il numero di sequenza dato dall'account o EOA che ha inviato la transazione.
- Gas price: prezzo del gas espresso in Wei.
- Gas limit: come per i blocchi, anche le transazioni hanno un gas limit che rappresenta la massima quantità di gas che il mittente è disposto a pagare per eseguire la transazione.
- Value: rappresenta la quantità di ether da inviare al destinatario (EOA o CA).
- Receiver: contiene l'indirizzo del destinatario.

- Data: può contenere o meno il bytecode di uno smart contract.
- Le componenti per eseguire la firma digitale.

I campi *data, value, gas e sender* saranno molto utili nella programmazione in *Solidity*[10] per: interagire con dati che EOA o CA si scambiano, effettuando controlli di proprietà.

3.3.3 Propagazione di una transazione

Come avviene in Bitcoin, la propagazione di una transazione parte dal nodo di origine e viene inoltrata ai nodi vicini (per un massimo di 13 nodi collegati): essi verificano la transazione e la aggiungono al *transaction pool.* Solo i nodi minatori aggiungeranno le varie transazioni contenute nella *transaction pool* ad un *candidate block* e cercheranno di convalidarle per poi inoltrare a tutta la rete la soluzione al blocco trovata [AW18].

3.4 Smart contract Ethereum

Ogni smart contract deve essere prima compilato e successivamente eseguito nella EVM. Ricordiamo che uno smart contract non può nascere dal nulla: solo un EOA può lanciare un nuovo smart contract, il quale potrà solo successivamente avere delle interazioni con altri smart contract già presenti in rete.

Inoltre, gli smart contract presentano diverse proprietà come *immutabilità*, infatti una volta distribuiti non possiamo più modificare i codici, l'unico modo che abbiamo per farlo è distruggere lo smart contract e proporne uno nuovo alla rete; per distruzione si intende l'eliminazione del bytecode dal campo data e non l'eliminazione della transazione. Altra proprietà è l'essere *deterministico*, cioè l'esito dell'esecuzione è uguale per tutti coloro che lo eseguono, contattando la transazione avviata per la prima volta da un EOA.

Infine, ogni smart contract opera in un piccolo contesto di esecuzione e viene aggiornato ogni volta che lo stato in esso contenuto cambia; tale stato è rappresentato da variabili nel codice che possono essere modificate da vari attori (EOA e altri smart contract).

3.4.1 Esecuzione di uno smart contract

Quando la EVM esegue uno smart contract viene anche creato il contesto di esecuzione o *execution context*, il quale è formato da varie regioni di memoria [HM22] [AW18]:

1. **Stack**: lista di elementi di 32 byte

[10] Linguaggio di programmazione di smart contract, https://docs.soliditylang.org/en/v0.8.15/

2. **Code**: parte di memoria in cui si conservano le istruzioni.

3. **Memory**: esiste durante l'esecuzione di uno smart contract, inizializzata a 0.

4. **Storage**: parte di memoria persistente. Implementata come una mappa a 32 byte e con valori a 32 byte.

5. **Return data**: restituisce un valore dopo che è stata effettuata una chiamata. La restituzione di valori avviene con *return* oppure *revert* nel caso in cui una chiamata non possa essere eseguita.

6. **Calldata**: contiene i dati di una transazione la prima volta che essa viene propagata: contiene i dati presenti nel costruttore.

Capitolo 4

Blockchain e algoritmi di consenso

In questo capitolo verranno introdotte le diverse tipologie di Blockchain esistenti e i vari algoritmi di consenso utilizzati, effettuando un rapido paragone tra le potenzialità offerte dalla tecnologia; applicando le potenzialità di una blockchain assieme al registro distribuito (DLT) e all'algoritmo di consenso, possiamo assicurare alla comunità una fiducia decentralizzata, basata sulla conoscenza di tutti i dati presenti nel sistema [XS18].

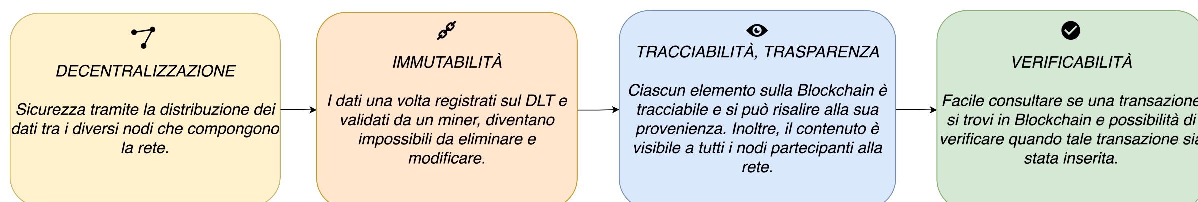


Figura 4.1: Proprietà nella Blockchain

1. **Decentralizzazione**: in un sistema centralizzato come una banca, ci affidiamo ad essa nel momento in cui effettuiamo un bonifico o effettuiamo determinate operazioni come il risalire al saldo corrente. In blockchain, non dobbiamo fidarci l'uno dell'altro e possiamo raggiungere un accordo comune come nel sistema centralizzato: come visto si possono consultare le informazioni di un nodo completo, per risalire all'intera storia delle transazioni; inoltre, si tiene aggiornato un DLT - *Distributed Ledger Technology* come registro distribuito tra tutti i peer.

2. **Immutabilità**: la catena immutabile delle transazioni, firmate crittograficamente e viste all'inizio di questa tesi, permette il non ripudio dei dati registrati su blockchain. Inoltre, si garantisce l'integrità di tutti i dati presenti in una transazione.

3. **Tracciabilità e verificabilità**: una volta registrate su blockchain possiamo sia verificare l'intera storia di una transazione e sia rintracciarla sul DLT per verificarne le varie proprietà.

4.1 Blockchain pubblica o privata?

Introducendo le varie blockchain presenti oggi, possiamo effettuare una semplice distinzione in due tipologie differenti:

- Blockchain pubblica o *permissionless*.
- Blockchain privata o *permissioned*.

La prima, blockchain pubblica, viene utilizzata sia da Bitcoin che da Ethereum. In essa chiunque può essere un validatore che permette di aiutare la comunità rispettando le proprietà sopra riportate, utilizzando la propria potenza computazionale o di calcolo ed essendo ripagato con una quantità di criptovaluta. La seconda, blockchain privata, utilizza dei soggetti preselezionati da una entità esterna e anche loro hanno il potere di convalidare le transazioni. In quest'ultima tecnologia, non abbiamo alcun tipo di competizione tra i miner, ma i nodi validatori scelti devono rispettare alcune condizioni obbligatorie:

- Dimostrare di utilizzare sistemi sicuri e certificati per poter operare sull'intera blockchain. Un esempio potrebbe essere l'utilizzo di comunicazioni *off-chain* solo utilizzando sistemi crittografici adeguati, così da prevenire accessi indesiderati ai dati.
- Evitare di effettuare un'analisi di dati che potrebbero fare uscire allo scoperto l'identità dei soggetti presenti nella rete.

Utilizzando una blockchain permissioned non avremo una vera e propria decentralizzazione totale di tutte le informazioni, ma ci dovremmo fidare di soggetti terzi o esterni che ricoprono il ruolo di validatori. In una blockchain permissionless, invece, si utilizzano gli algoritmi di consenso che inducono il miner ad effettuare uno sforzo per validare le transazioni e tenere sicuro il sistema; quindi, si ha una maggiore sicurezza nei sistemi permissionless piuttosto che nei sistemi permissioned.

4.2 Protocollo di consenso

Scegliere un protocollo di consenso a discapito di un altro può influire su livelli di sicurezza e scalabilità. L'approccio generale che viene utilizzato quando si ha difficoltà nello scegliere quale catena sia quella valida all'interno di blockchain, è denominato *consenso di Nakamoto (Nakamoto consensus)*, il quale permette di scegliere la catena di blocchi più lunga osservata.

4.2.1 Proof-of-Work (PoW)

La Proof-of-Work è attualmente il protocollo di consenso utilizzato sia in Bitcoin e sia in Ethereum: il protocollo di Bitcoin è denominato *HashCash*, mentre quello di Ethereum è *Ethash* [XS18]. La più grande criticità di questo algoritmo è la maggiore potenza di calcolo che deve essere utilizzata da un miner: al crescere della blockchain, si avrà una crescita della potenza di calcolo per trovare l'*hashing* del blocco successivo. Tale crescita è denominata *bomba di difficoltà* e si sta cercando di proporre nuovi algoritmi di consenso che cercano di superare tale limite.

4.2.2 Proof-of-Stake (PoS)

Questo algoritmo non presenta una capacità di calcolo da avere per validare i singoli blocchi. Esso si fonda sulla quantità di criptovalute tenute da un miner: maggiore è la quantità di cripto detenuta e maggiori saranno le possibilità di validare i blocchi. Ovviamente la sicurezza sta nel fatto che un possibile attaccante, dovrebbe detenere la maggior parte delle criptovalute in circolazione per effettuare un attacco (attacco al 51%), ma tutto questo sarebbe controproducente proprio per il miner che detiene la quota più alta; vediamo insieme perchè.

Come prima operazione, un nodo che vuole diventare validatore invia una transazione speciale che blocca la propria quantità di ether in un deposito. Successivamente, quando il sistema ha identificato un numero massimo di validatori, inizierà il processo di validazione: si vota sul blocco da validare e il peso del singolo voto per il singolo miner dipende dalla quantità di ether depositato. Se i validatori sono onesti, allora i miner guadagnano una piccola ricompensa, proporzionale al valore depositato; altrimenti, si applica una sorta di ritorsione in cui viene persa l'intera quantità di cripto depositata [AW18].

Capitolo 5

Introduzione alla gestione della filiera del sangue

In questo capitolo verrà introdotto il caso di studio relativo alla disponibilità delle sacche di sangue preso in considerazione e quali problematiche sono state riscontrate. Inoltre, verrà proposta una parte introduttiva del lavoro svolto operando sulla blockchain di Ethereum.

5.1 Descrizione del caso di studio

La gestione della catena per la disponibilità delle sacche di sangue è di vitale importanza: la non disponibilità può comportare problemi gravi al paziente e addirittura causare la morte o varie complicazioni. La disponibilità del sangue risulta fondamentale per il centro medico, che può usufruirne in vario modo: chirurgia, trapianto di organi, tumori e malattie del sangue. Altro aspetto fondamentale è che il sangue è un prodotto deteriorabile; quindi, si ha la necessità di garantire differenti proprietà come la gestione della temperatura, sopra la quale una sacca di sangue potrebbe essere invalidata e non resa disponibile per una possibile trasfusione.

Nel caso di studio preso in considerazione si è cercato di tenere traccia delle varie sacche presenti in un centro trasfusionale, il quale le rende disponibili ad una struttura sanitaria che si occuperà, solo dopo aver passato diversi controlli, di prenderla in carico e utilizzarla per eventuali operazioni. Nel paragrafo successivo verranno introdotte alcune informazioni per la gestione di una sacca, quali tipologie di sacche ci sono e verranno illustrati alcuni problemi di pianificazione nella catena di approvvigionamento.

5.1.1 Le principali componenti del sangue

Le componenti principali che possiamo trovare in una sacca di sangue sono: *globuli rossi*, *piastrine*, *globuli bianchi* e *plasma*. I componenti citati possono essere recuperati in una sacca di sangue intero, oppure scissi attraverso un processo meccanico denominato *aferesi*: permette di recuperare solo le componenti necessarie, mentre il resto viene inserito nuovamente nel corpo del donatore [APi19].

5.1.2 La fase di test e di distribuzione

Dopo la raccolta, il centro trasfusionale si occupa di verificare ed effettuare i relativi test per rendere disponibili le sacche presso le strutture sanitarie. Il test è fondamentale per la sicurezza della distribuzione del sangue: viene effettuato un controllo in laboratorio per identificare il gruppo sanguigno (A^+, A^-, B^+, B^-, AB^+, AB^-, 0^+ e 0^-).

Successivamente avvengono dei controlli per testare le varie unità, verificando che siano prive di malattie: il sistema utilizzato potrebbe presentare dei falsi positivi o falsi negativi, per questo si ha la necessità di selezionare una serie di test aggiuntivi di sicurezza [APi19]. Infine, una volta ultimati tutti i controlli necessari, il sangue viene depositato all'interno di apposite emoteche, in attesa di essere distribuito. A tal proposito, il sangue intero e le varie componenti non possono rimanere inutilizzate per un periodo troppo lungo; la tabella di seguito fornisce la temperatura e i relativi giorni di conservazione.

Componente	Periodo di conservazione	Condizioni
Sangue intero (WB)	21/35 giorni	18-24°C
Globuli rossi (RBC)	42 giorni	2-10°C
Piastrine (PLT)	3-7 giorni	20-24°C
Plasma	1 anno	$\leq$ -30°C
Cellule staminali (Cryo)	1 anno	$\leq$ -30°C

5.1.3 La fase di raccolta

Come illustrato nel paragrafo precedente, abbiamo due modalità di raccolta del sangue e per questo abbiamo varie tipologie di sacche utilizzate nella raccolta:

- *Sacca singola*: viene depositato il sangue intero senza scinderlo nelle varie componenti enunciate. Questa raccolta può essere fatta per singolo paziente, ma bisogna rispettare le norme Europee vigenti per effettuare una nuova raccolta: in Italia la frequenza di donazione di sangue intero può essere effettuata solo dopo che siano trascorsi 90 giorni (*tempo di differimento*) dall'ultima donazione. Inoltre, nel giro

di un anno possono essere effettuate un numero massimo di donazioni pari a quattro sia per uomo che per donna: in quest'ultimo caso, se fertile, allora solo due donazioni come tetto massimo.

- *Sacca doppia*: contiene globuli rossi e plasma contenuto in due sacche differenti.
- *Sacca tripla*: contiene globuli rossi, piastrine/cryo e plasma.
- *Sacca quadrupla*: contiene globuli rossi, plasma e buffy coat (successivamente strato leucocitario-piastrinico). Il buffy coat successivamente viene centrifugato per produrre delle unità di piastrine e globuli bianchi, le quali possono essere utilizzate nel processo di trasfusione; possono essere uniti dai quattro a cinque buffy coat dello stesso gruppo sanguigno prima di scinderlo nelle varie componenti.

Secondo gli studi effettuati da [APi19], la resa degli emoderivati per aferesi è considerevolmente maggiore di quella della raccolta di unità di sangue intero.

5.1.4 La fase di trasporto

I veicoli impiegati per il trasporto di sacche di sangue, possono trasportare tali unità dai centri di raccolta mobili (CSs - Mobile Collection Sites) verso i centri trasfusionali (BCs - Blood Centers), oppure dai centri trasfusionali verso le strutture sanitarie locali. I centri di raccolta mobili, di solito, riescono a raccogliere una maggiore quantità di sangue intero e componenti, sensibilizzando attraverso campagne pubblicitarie e collocandosi nei centri della città o nei poli universitari delle regioni.

Di solito, il trasporto avviene durante la notte e vengono effettuati, da parte della struttura sanitaria, eventuali controlli giorno per giorno sulle disponibilità interne di sangue. Le strutture possono richiedere ordini di routine (pratica preferibile, evitando criticità e non disponibilità per i pazienti) oppure, effettuare degli ordini di emergenza per soddisfare la domanda in cui non c'è disponibilità di prodotto.

Una volta raggiunta la destinazione, viene effettuato un controllo tra il paziente e la sacca ricevuta (tale processo è denominato *cross matching*) in cui viene determinata la compatibilità con il donatore e, in caso di esito positivo, viene assegnata la sacca e restituito un documento al centro trasfusionale, il quale può eliminare la sacca dal proprio inventario.

5.1.5 Utilizzo di una sacca

Per concludere, introduciamo la politica di emissione e utilizzo delle sacche di sangue da parte della struttura sanitaria. Trovandoci di fronte ad un prodotto deteriorabile,

la politica utilizzata per la somministrazione e utilizzo di una specifica sacca, è quella *FIFO (First-in First-out)*, la quale utilizza il prodotto più vecchio ricevuto, cioè quello entrato per primo in struttura [APi19].

5.2 Problematiche riscontrate e motivazione

Il caso di studio sulla disponibilità delle sacche di sangue, presenta i seguenti possibili problemi. Nella fase di disponibilità, ci sono alcune criticità: ad oggi, è il centro trasfusionale che ha il controllo delle disponibilità dei vari prodotti, e gestisce la distribuzione e l'eliminazione delle sacche utilizzate, attraverso un documento che attesta l'effettivo utilizzo della sacca in una struttura sanitaria. Anche nella fase di trasporto potrebbero esserci alcuni problemi, ad esempio la temperatura di conservazione potrebbe salire per un guasto tecnico o per altri problemi legati al trasporto; inoltre, potrebbero verificarsi anche dei problemi legati alla gestione dell'inventario e alle emoteche che si occupano di tenere ad una temperatura costante i vari prodotti.

5.3 Proposta di alcuni smart contract

Proponiamo uno smart contract in grado di depositare nuove sacche di sangue, magari gestendo il semplice identificativo della sacca, registrando il tutto sulla blockchain (operazioni sulle proprietà della sacca potrebbero essere eseguite off-chain). Così facendo, sarà possibile risalire a tutte le transazioni contenenti una sacca di sangue con le relative proprietà e, decidere se renderla disponibile per una struttura sanitaria. Per di più, proponiamo varie funzionalità, le quali permettono, in modo efficiente, di: validare la sacca da distribuire alla struttura richiedente, assegnare una sacca di sangue, effettuare i cambiamenti di stato su una sacca presente nel sistema, eliminare la sacca utilizzata, notificando a tutta la comunità tale cambiamento.

Ovviamente, dovrebbero essere effettuati dei controlli prima di rilasciare una sacca e per questo proponiamo altre funzionalità le quali, dopo aver ricevuto *off-chain* le varie informazioni per l'assegnazione della sacca e dopo averla identificata internamente, permettono il rilascio di una tipologia di questa, tra quelle enunciate nel caso di studio, alla struttura sanitaria. In ultima analisi, dopo aver effettuato l'operazione di *cross matching*, sarà possibile comunicare al centro trasfusionale l'assegnazione di una sacca ad un paziente: tale funzionalità è stata realizzata facendo comunicare alcuni smart contract, ma è possibile farlo anche attraverso un *EOA*.

Il processo proposto sfrutta la blockchain già esistente di Ethereum, il protocollo di consenso *PoW* e il linguaggio *Solidity* per la programmazione degli smart contract.

Capitolo 6

Solidity

Prima di introdurre la parte sperimentale della tesi, in questo capitolo verrà illustrato il funzionamento di Solidity e quali componenti del linguaggio sono state utilizzate per la realizzazione degli smart contract proposti.

6.1 Introduzione al linguaggio

Solidity è un linguaggio di programmazione ad alto livello introdotto per scrivere codice distribuito o smart contract, da inoltrare a tutta la comunità della blockchain. Una parte del funzionamento, per quanto riguarda la distribuzione del codice, è stata trattata precedentemente; di seguito viene riportato un quadro generale di ciò che accade.

6.1.1 Gestione delle versioni

Cosa fondamentale quando si programma in Solidity è scegliere una specifica versione del linguaggio. Ci sono stati durante la storia vari cambi e modifiche al linguaggio, ad esempio, passando dalla versione *v0.6.0* alla versione attuale, durante la scrittura di questa tesi, *v0.8.0*, si è risolto il problema di *overflow* e *underflow*. Precedentemente, se veniva utilizzata una variabile di tipo intero senza segno, nel caso di overflow veniva settata la variabile al valore iniziale e cioè 0, questo perchè veniva effettuata una operazione di *wrap*, e quindi di avvolgimento, del risultato presentato.

Giustamente, come visto nelle sezioni precedenti, tutto questo ha un costo e la comunità di Solidity ha pensato di non far più gestire tale problematica direttamente all'utente, utilizzando dei codici di controllo, ma effettuando tale gestione internamente, aumentando di una piccola quantità il costo del gas per l'operazione. Quindi, quando si

programma in *Solidity*, bisogna prestare particolare attenzione alla scelta della versione, tenendo in considerazione i pro e contro del codice che si andrà ad utilizzare.

6.1.2 Gestione di una transazione in Solidity

Tutte le transazioni si trovano in un blocco che viene minato e successivamente validato e accettato dagli altri nodi. Quando compiliamo uno smart contract, lo facciamo attraverso una transazione che viene lanciata a tutta la rete, con associato i dati necessari. Una volta lanciata una transazione con il codice di uno smart contract, questo si troverà nello stato *start*: abbiamo la compilazione in *bytecode* e i relativi dati di una transazione. Nella fase di interazione con lo smart contract avremo lo stato *running*: richiamiamo le funzioni e mandiamo transazioni. Infine, possiamo trovarci in uno stato *stop* al chiamare di una particolare funzione denominata *selfdestruct*, la quale rimuove il bytecode dalla transazione.

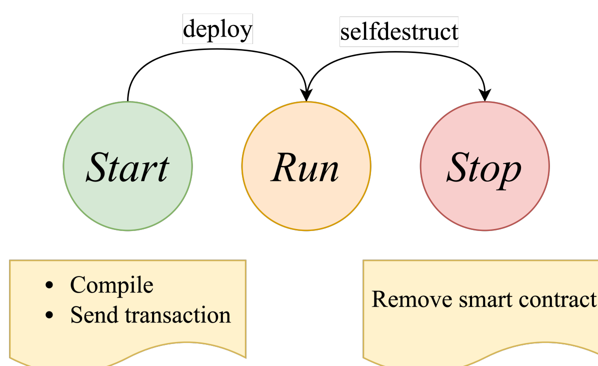


Figura 6.1: Ciclo di vita di uno smart contract

Per concludere, quando dobbiamo salvare dei dati su blockchain con uno smart contract meglio non farlo direttamente. Fare ciò risulterà molto costoso e inutilmente oneroso, meglio utilizzare un qualcosa di *off-chain* registrando, sulla blockchain, solo un identificativo per recuperare tutte le informazioni al di fuori della catena.

6.1.3 Componenti utilizzati

Oltre all'utilizzo del linguaggio Solidity sono state utilizzate altre componenti come *Hardhat*, *Ganache*, *Remix* e il tutto è stato testato utilizzando la rete Testnet Ropsten messa a disposizione della comunità di Ethereum per testare gli smart contract, utilizzando il wallet virtuale *MetaMask* per visualizzare le transazioni.

MetaMask è un wallet che permette di utilizzare una quantità di ether di prova per testare le funzionalità di uno smart contract. L'idea è quella di avere una replica

dell'intera blockchain principale di Ethereum in cui è possibile visualizzare le transazioni e i codici degli smart contract presenti. Successivamente, per una maggiore flessibilità e senza attendere il tempo necessario di validazione di una transazione, si è preferito utilizzare Hardhat, la quale mette a disposizione degli sviluppatori, dei nodi con una quantità di ether predefinita (1000 ETH), assieme all'ambiente Remix che permette di compilare e distribuire nell'immediato uno smart contract, verificando e proponendo all'utente eventuali problemi.

6.2 Tipi e dati

In questa sezione verranno introdotti i vari tipi di dato utilizzati e i vari controlli sul codice effettuati attraverso asserzioni e altre operazioni.

6.2.1 Unsigned integer

Durante la scrittura degli smart contract si è preferito l'utilizzo di valori interi senza segno. Utilizzando la *v0.8.0* e superiore, non ci sono stati problemi relativi alla gestione dell'*avvolgimento*; ovviamente, Solidity mette a disposizione un metodo per annullare tale *feature aggiuntiva* e permettere di far gestire all'utente l'operazione di wrap; il codice da inserire per permettere quanto detto è *unchecked* {*...*}.

6.2.2 Mapping

Il mapping che viene introdotto in *Solidity*, permette di avere a disposizione un array all'interno dello smart contract, il quale si occupa di mappare specifici dati. Il mapping può essere fatto sia con tipi nativi come *int*, *uint*, *address* (identificatore di un indirizzo Ethereum valido), e sia utilizzando delle strutture definite dallo sviluppatore del codice utilizzando, come nel linguaggio C++, le *struct*. Vediamo un semplice esempio di mapping:

```
// SPDX-License-Identifier: unlicensed
pragma solidity ^0.8.0;

contract MappingExample {
    mapping (uint => string) public values;

    function addValue (uint _index, string memory _newValue) public {
        values[_index] = _newValue;
    }
}
```

Come si nota, abbiamo un indice rappresentato da un valore intero senza segno, il quale mappa un valore di tipo stringa. Tale *mapping* è di tipo pubblico in modo che qualsiasi utente possa controllare, passando un indice, se il valore stringa è presente o meno all'interno dell'array. La funzione *addValue* permette di inserire un nuovo valore, passando un intero senza segno come indice e una stringa, e aggiunge tale valore alla posizione specificata; in questo caso viene permesso sovrascrivere un valore già presente, ma il controllo potrebbe essere effettuato da alcuni *modifier* che vediamo nel paragrafo successivo.

All'interno di *Solidity* è possibile anche utilizzare gli array, la scelta di utilizzare il mapping è molto semplice: si ha un consumo di gas inferiore e, di conseguenza, un pagamento più basso riguardante gli ether.

6.2.3 Modifier per la gestione di criticità

Negli smart contract, sono stati utilizzati anche i *modifier*: strutture che permettono la non replicabilità del codice e l'aggiunta in essi di eventuali controlli da effettuare su più parti o funzioni. All'interno dei *modifier*, sono stati inseriti eventuali controlli che permettono di terminare l'esecuzione di uno smart contract, nel momento in cui alcune condizioni non sono soddisfatte; i controlli possibili sono:

- *Require*: termina l'esecuzione di uno smart contract, ritornando la quantità di gas rimanente. Per fare ciò, valuta una condizione come input e se non soddisfatta effettua una chiamata che blocca l'esecuzione, effettuando un *revert*. Solitamente, tale operazione viene utilizzata per effettuare dei controlli sugli input passati da un EOA o da uno smart contract.

- *Assert*: termina l'esecuzione di uno smart contract ma, a differenza di *require*, non restituisce il gas rimanente, consumandolo. Tale operazione viene utilizzata per validare le invarianti ed effettuare controlli come divisione per zero, convertire un grande numero ecc.

6.2.4 Gestione degli eventi

In Solidity è possibile stampare, attraverso delle operazioni denominate *Event*, informazioni sui *logs* degli smart contract. Questo costrutto è molto utile per: verificare se l'operazione è andata a buon fine e per restituire all'utente dati che identificano tale operazione. Ad esempio, una volta effettuata una chiamata ad una funzione, sarà possibile visualizzare l'indirizzo che ha effettuato tale chiamata e, in aggiunta, i relativi dati inviati.

6.2.5 Constructor per operazioni iniziali

È anche possibile dichiarare un costruttore dello smart contract: nel momento in cui viene lanciato il codice da un EOA, è possibile salvare l'indirizzo di questo ed effettuare determinate chiamate di funzioni o distruggere uno smart contract, se e solo se è il proprietario a richiedere tale operazione. Aggiungere un costruttore è anche utile nel momento in cui si deve effettuare un'operazione di multi-firma per inviare specifici dati: a differenza del sistema Bitcoin, il quale permette di firmare più volte una specifica transazione prima di sbloccare una certa quantità di bitcoin, in Ethereum tale operazione non è consentita, ma è possibile scrivere uno smart contract che permette di farlo.

6.2.6 Comunicazione tra smart contract

call e delegatecall

Altra funzionalità è la comunicazione tra uno o più smart contract; possono essere effettuate chiamate attraverso due metodi presenti in Solidity: *call* e *delegatecall*, ma bisogna prestare particolare attenzione al loro utilizzo.

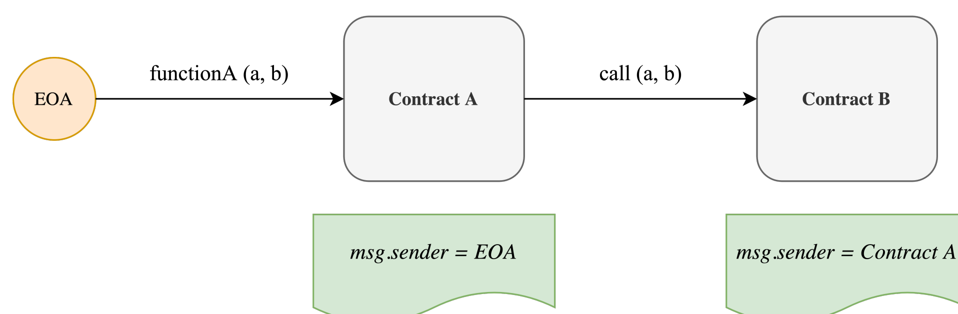


Figura 6.2: Operazione di call

Come notiamo dalla *figura 6.2*, nel momento in cui viene effettuata una *call* da uno smart contract ad un altro, il contesto di esecuzione sarà quello del secondo smart contract. Quindi, supponiamo di avere a disposizione una funzione nello smart contract chiamato, la quale permette di aggiornare due campi: un indirizzo e un valore *uint*. Effettuando una call, verranno aggiornati i dati presenti nel secondo smart contract e il *msg.sender*[11] di riferimento risulterà quello dello smart contract chiamante.

Sottolineiamo sempre che, per essere lanciato, uno smart contract deve essere chiamato da un EOA che genera una transazione di creazione. Nell'esempio, l'EOA comunica

[11]**msg.sender** è un dato che possiamo trovare nel momento in cui viene lanciata una nuova transazione: restituisce l'indirizzo del mittente che ha contattato lo smart contract.

con il *Contract A* il quale solo successivamente potrà attivare la comunicazione con il *Contract B*; inoltre, il *Contract A* riceverà sempre come indirizzo quello dell'EOA che ha effettuato la richiesta.

Nel caso di *delegatecall*, il contesto di esecuzione sarà quello del primo smart contract. Quindi, è come se venisse eseguita la funzione di aggiornamento dei due dati all'interno del primo smart contract. Mentre, nel secondo smart contract, i dati non verrebbero aggiornati; il *msg.sender* in questo caso non sarà quello dello smart contract chiamante, bensì quello dell'EOA che ha lanciato una transazione.

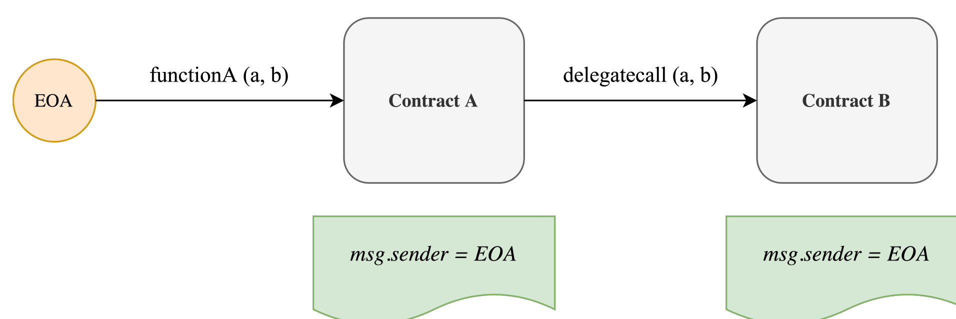


Figura 6.3: Operazione di delegatecall

È stato fornito il seguente esempio per comprendere che, se venisse lanciata una richiesta di *selfdestruct* del *Contract B* utilizzando una *delegatecall*, ci sarebbe un'autodistruzione involontaria del chiamante. Nel paragrafo successivo è riportato lo smart contract che esegue le operazioni descritte.

Smart contract

```
// SPDX-License-Identifier: unlicensed
pragma solidity ^0.8.0;

contract ContractA {
    uint256 public number;
    address public _address;

    function sendData (address _contract) external {
        (bool success, ) = _contract.call(abi.encodeWithSignature("
           print(uint256)", 10));
        require (success, "Call error...");
    }
}

contract ContractB {
```

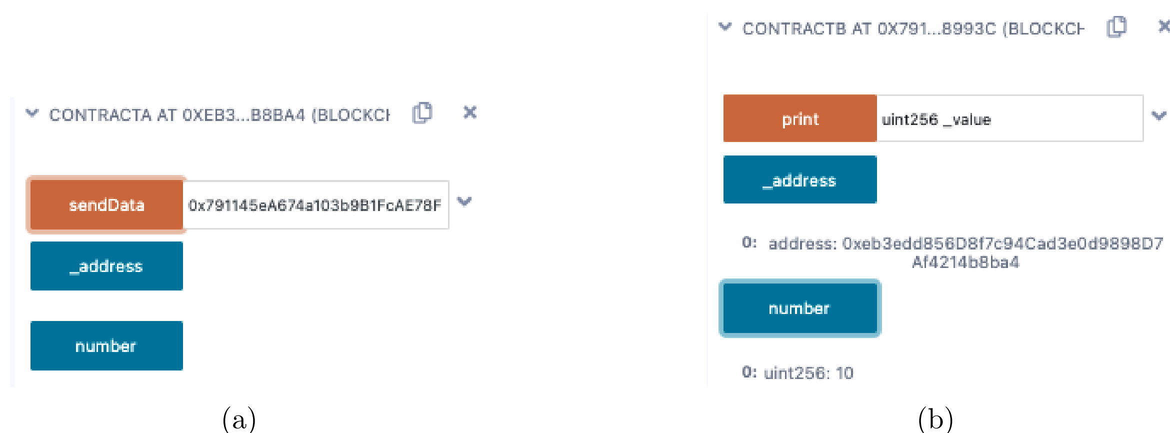
```
    uint256 public number;
    address public _address;

    event Print (address sender, uint256 value);

    function print (uint256 _value) external {
        _address = msg.sender;
        number = _value;
    }
}
```

Dal codice si notano due funzioni esterne[12]: nel primo smart contract viene effettuata un'operazione di *call* e viene restituito un valore booleano che è settato a *true* nel caso di successo della chiamata, altrimenti restituisce un *false*. Come si nota, è stato inserito un controllo di *require*, per effettuare, eventualmente, un *revert* e comunicare all'utente che l'operazione non è andata a buon fine. Il secondo smart contract riceve un valore di tipo intero (senza segno) e va a settare due variabili pubbliche: l'indirizzo di chi ha lanciato la richiesta, in questo caso il primo smart contract, e un valore passato di default pari a 10.

Come conclusione, viene riportato il risultato generato: dopo aver lanciato i due smart contract, è possibile richiamare la funzione *sendData* passando l'indirizzo dello smart contract da chiamare. Dopo aver lanciato una nuova transazione, possiamo vedere che l'operazione di aggiornamento va a buon fine e i dati vengono aggiornati.



(a) (b)

Figura 6.4: Esecuzione degli smart contract

[12]Le chiamate *external*, permettono di richiamare solo esternamente le funzioni, le quali sono inutilizzabili in contesti interni al codice.

Capitolo 7

Declinazione della soluzione al caso di studio

In questo capitolo verranno introdotti gli smart contract che abbiamo cercato di realizzare a sostegno della tesi discussa. Il codice riportato nelle seguenti sezioni può essere liberamente consultato e potrà essere migliorato aggiungendo nuove funzionalità.

7.1 Introduzione

Si è cercato di realizzare gli smart contract utilizzando come ambiente di sviluppo Hardhat, come già annunciato all'inizio della tesi. I casi di test sono stati realizzati in *Javascript* utilizzando *hardhat-waffle* come libreria di testing messa a disposizione dall'ambiente utilizzato. Tali test hanno permesso di controllare il codice durante tutta la fase di sviluppo, verificando tutte le proprietà ad ogni modifica effettuata.

Le informazioni, durante la seguente esposizione, saranno organizzate nella maniera che segue:

1. Trattazione dei test realizzati e spiegazioni di essi.
2. Utilizzo di codice testato dalla comunità OpenZeppelin per effettuare controlli di proprietà.
3. Trattazione del codice proposto, con alcune parti spiegate nel dettaglio attraverso schemi riassuntivi per facilitarne l'apprendimento.

7.2 Fase di testing

Come primo passo, proponiamo alcuni test realizzati per controllare la distribuzione del codice anche in futuro; è stato scritto il minimo necessario che permette di superare il test e solo successivamente sono stati aggiunti degli accorgimenti, cercando di mantenere il codice il più semplice possibile. Di seguito, vengono proposte alcune parole chiave che troviamo nei vari test:

- *getContractFactory (...)*: prende come parametro una stringa che identifica uno smart contract da lanciare. Così facendo, si ottiene una astrazione del contratto che sarà utilizzato per creare una nuova istanza.
- *deploy()*: esegue lo smart contract e assegna il risultato ad una eventuale variabile.
- *deployed()*: permette di creare l'istanza del contratto, consentendo di accedere alle funzioni che propone in fase di test.
- *wait()*: attende che una transazione lanciata venga aggiunta ad un nuovo blocco, successivamente minato e confermato.

7.2.1 Verifica creazione di una sacca di sangue

Il caso di test proposto permette di verificare se una sacca di sangue viene creata correttamente. Viene lanciata una transazione, la quale contatta lo smart contract *Availability*, si attende che la transazione venga minata e viene effettuata un'operazione di *expect*:

- Se il risultato ritorna *true*, allora il test è passato.
- Se il risultato ritorna un'operazione di *revert*, allora vuol dire che si ha avuto un problema in fase di creazione.

```
it("Should return that the new blood sack exist", async function () {
    const Availability = await ethers.getContractFactory("Availability"
       );
    const availability = await Availability.deploy();

    // Deployed smart contract
    await availability.deployed();

    const tx = await availability.addNewBloodSack(0, 3, 0);

    // Wait until the transaction is mined
    await tx.wait();

    expect(await availability.isExist(0)).to.equal(true);
```

```
    await expect(availability.isExist(1)).to.be.revertedWith('The blood
        sack number not exist');
});
```

In ultima istanza, viene effettuato un controllo per verificare se nella posizione successiva, si ha una sacca di sangue creata precedentemente. In questo caso, l'operazione *"reverta"* e viene restituito un messaggio di errore, passando il test.

7.2.2 Verifica disponibilità di una sacca di sangue

Il seguente caso di test verifica se una sacca di sangue creata rispetta le proprietà inserite: una volta creata una nuova sacca, non è possibile utilizzarla prima di eventuali controlli che il centro trasfusionale dovrà compiere (le verifiche sono riportate nella "Descrizione del caso di studio"). I passi da seguire in questo test sono:

- Si verifica l'usabilità della sacca, se rispetta la proprietà allora verrà fatto *revert* e verrà restituito un messaggio di errore.
- Se la prima parte del test viene passata, allora si lancia una nuova transazione per *settare* ad *usabile* lo stato della sacca creata.
- Si controlla se l'operazione è andata a buon fine, provando anche a lanciare nuovamente la transazione di settaggio sulla sacca. In quest'ultimo caso, il test passerà se e solo se non verrà permesso di nuovo di cambiare lo stato.

```
it("Should return blood sack is usable", async function () {
    const BloodSack = await ethers.getContractFactory("BloodSack");
    const bloodSack = await BloodSack.deploy(0, 0, 3, 0);

    // Deployed smart contract
    await bloodSack.deployed();

    // Blood sack is unusable
    await expect(bloodSack.isUsable()).to.be.revertedWith('This blood
       sack cannot be used');

    // Transaction to set state
    const txSetState = await bloodSack.setBloodSackState();
    await txSetState.wait();

    // Check if blood sack is usable
    expect(await bloodSack.isUsable()).to.equal(true);
    await expect(bloodSack.setBloodSackState()).to.be.revertedWith('
       Blood sack already usable');
});
```

Dal codice notiamo la distribuzione di una sacca di sangue, senza utilizzare come intermediario di creazione il centro trasfusionale. In questo caso non si creano problemi relativi ad eventuali controlli, in quanto stiamo testando solo le proprietà presenti in ogni sacca creata.

7.2.3 Verifica di una sacca non assegnata al paziente

In quest'ultimo test che proponiamo, abbiamo cercato di testare la non eliminazione della sacca una volta assegnata alla struttura sanitaria; infatti, solo dopo aver effettuato l'eventuale assegnazione al paziente, si potranno inviare le informazioni al centro trasfusionale, il quale si occuperà della cancellazione. Le istruzioni sono le seguenti:

- Viene creata l'istanza di riferimento per il centro trasfusionale e vengono lanciate tre transazioni per: creare una sacca di sangue, settare lo stato della sacca ad *usabile* e, infine, approvare una specifica sacca per una struttura sanitaria.
- Si tenta di eliminare la sacca assegnata: se l'operazione porta ad un *revert*, allora il test passa con successo, in quanto la struttura sanitaria non ha ancora assegnato la sacca ricevuta.

```
it("Should return that blood sack cannot be deleted", async function ()
    {
    const Availability = await ethers.getContractFactory("Availability"
       );
    const availability = await Availability.deploy();

    const [owner, secondAccount] = await ethers.getSigners();

    // Deployed smart contract
    await availability.deployed();

    // Transaction to create a blood sack
    const tx = await availability.addNewBloodSack(0, 3, 0);
    const txUsable = await availability.changeBloodSackState(0);
    const txApprove = await availability.approve(secondAccount.address,
        0);

    // Wait until the transaction is mined
    await tx.wait();
    await txUsable.wait();
    await txApprove.wait();

    // Transaction to delete a blood sack
    await expect(availability.deleteBloodSack(secondAccount.address)).
       to.be.revertedWith('Blood sack cannot be deleted');
});
```

7.3 Standard OpenZeppelin

Nella fase di sviluppo, sono state prese in considerazione alcune parti di codice già testate. La comunità di OpenZeppelin[13], mette a disposizione funzioni e classi sicure, da utilizzare all'interno di progetti e per la gestione di applicazioni decentralizzate. Nello specifico, è stata utilizzata la funzionalità che permette di identificare un utente proprietario di un relativo smart contract utilizzando un *abstract contract* implementato nel codice proposto.

```
address private _owner;

/**
 * @dev Initializes the contract setting the deployer as the initial
    owner.
*/
constructor() {
    _transferOwnership(_msgSender());
}

/**
 * @dev Returns the address of the current owner.
 */
function owner() public view virtual returns (address) {
    return _owner;
}

/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
    require(owner() == _msgSender(), "Ownable: caller is not the owner"
        );
    _;
}

/**
 * @dev Transfers ownership of the contract to a new account (`newOwner
    `).
 * Can only be called by the current owner.
 */
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero
        address");
    _transferOwnership(newOwner);
}

/**
```

[13] La comunità di OpenZeppelin: https://github.com/OpenZeppelin

```
 * @dev Transfers ownership of the contract to a new account (`newOwner
    `).
 * Internal function without access restriction.
 */
function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

Bisognerà estendere lo smart contract di riferimento con l'*abstract contract Ownable* e utilizzare le funzionalità proposte come la funzione *owner()*, la quale restituisce l'indirizzo del proprietario oppure *onlyOwner()* che rappresenta un *modifier* di controllo per verificare se, chi ha effettuato un'interazione con uno smart contract, è il legittimo proprietario.

7.4 Proprietà di una sacca di sangue

In questa sezione verranno presentate le proprietà e il relativo smart contract di una sacca di sangue. La trattazione sarà divisa in parti: stato dello smart contract e alcune funzioni utili sia per il centro trasfusionale che per la struttura sanitaria.

7.4.1 Stato dello smart contract

Lo stato è composto da:

- *_type*: rappresenta la tipologia di una sacca di sangue.
- *_group*: identifica il gruppo sanguigno di riferimento.
- *_Rh*: identifica il valore dell'Rh.
- *_inventory*: stabilisce se una sacca è stata assegnata o meno.
- *_state*: stabilisce se una sacca è utilizzabile o meno.

```
/// @notice type
enum BloodSackType { Single, Double, Triple, Quadruple }
/// @notice group
enum BloodGroup { A, B, AB, ZERO }
/// @notice Rh
enum BloodRh { PLUS, MINUS }
/// @notice assigned or unassigned
enum BloodInventory { Assigned, Unassigned }
/// @notice state
```

```
enum State { Usable, Unusable }

/// @notice Smart contract state
uint private id;
BloodSack.BloodSackType private _type;
BloodSack.BloodGroup private _group;
BloodSack.BloodRh private _Rh;
BloodSack.BloodInventory public _inventory;
BloodSack.State private _state;
```

7.4.2 Funzioni utili

In tutte le funzioni che proponiamo, viene bloccato qualsiasi altro utente che non sia il proprietario della sacca di sangue inserita nel sistema. La prima funzione realizzata controlla se una sacca si trova nello stato *unusable* e solo in quel caso permette, ad un centro trasfusionale, di modificare tale campo.

```
/// @notice Change blood sack state
function setBloodSackState () external onlyOwner {
    require (_state == BloodSack.State.Unusable, "Blood sack already
       usable");
    _state = BloodSack.State.Usable;
}
```

La seconda funzione permette di modificare lo stato di assegnazione ad un paziente della sacca. Tale operazione inizia con la richiesta da parte di un nodo di domanda (struttura sanitaria pubblica o privata) verso il centro trasfusionale, il quale ha il compito di modificare tale stato.

```
/// @notice Change blood sack inventory
function setInventory () external onlyOwner {
    require (_state == BloodSack.State.Usable, "This blood sack is
       unusable");
    require (_inventory == BloodInventory.Unassigned, "This blood sack
       is already assigned");
    _inventory = BloodInventory.Assigned;
}
```

L'ultima funzione che proponiamo, elimina una sacca di sangue solo se è stata assegnata e trasfusa ad un paziente. L'operazione viene svolta dal centro trasfusionale e viene effettuata un'operazione di *selfdestruct* dello smart contract restituendo, ad un indirizzo passato come parametro, la quantità di ether che lo smart contract possedeva. Quest'ultima operazione è fondamentale per controllare la quantità di ether in circolazione, evitando il bloccaggio in qualche indirizzo Ethereum.

```
/// @notice Destroy a blood sack if it is assigned
function destroyBloodSack (address _to) external onlyOwner {
    require (_inventory == BloodInventory.Assigned, "This blood sack
       isn't assigned");
    selfdestruct(payable(_to));
}
```

7.5 Smart contract per gestire la disponibilità

In questa sezione introdurremo parti di codice degli smart contract proposti come soluzione al caso di studio riportato. Verrà introdotta la logica di comunicazione tra varie parti di codice, cercando di spiegarne nel dettaglio le varie funzionalità.

7.5.1 Stato dello smart contract

Lo stato dello smart contract per la distribuzione e la verifica della disponibilità delle sacche di sangue è composto da tre variabili:

- Una variabile *uint256* che identifica l'indice di una specifica sacca; il numero massimo di sacche possibili è pari a $2^{256} - 1$.
- Un *mapping* che tiene traccia di tutte le sacche raccolte ed inserite dal centro trasfusionale.
- Un *mapping* di un *mapping* utilizzato per assegnare una specifica sacca di sangue ad una struttura sanitaria. Dopo l'assegnazione, il centro trasfusionale terrà sempre traccia della sacca, fino al momento di trasfusione.

Lo stato verrà aggiornato tutte le volte che avverrà un'interazione con lo smart contract. Nel primo caso, quando avverrà un nuovo inserimento nel sistema di una nuova sacca, il valore dell'indice sarà incrementato automaticamente; allo stesso modo, anche il primo mapping, ”mapperà” il nuovo dato inserito.

Quando avverrà l'assegnazione di una sacca presso una struttura, lo stato relativo al mapping cambierà e avverrà l'eliminazione interna di questa: verrà ”svuotato” il campo di riferimento, il quale conterrà una *struct* con i dati vuoti della sacca eliminata. Infine, il *mapping composto* verrà aggiornato sia in fase di assegnazione che in fase di eliminazione da parte del centro trasfusionale.

Struct di identificazione

La struct annunciata nel paragrafo precedente contiene: un riferimento al codice della sacca (nel momento di creazione verrà generata una transazione che conterrà tutti i dati di una sacca, cioè un nuovo smart contract con un proprio indirizzo) e un campo *uint256* per l'identificazione interna al centro trasfusionale.

```
/// @notice A new Blood sack
struct Sack {
    BloodSack _sack;
    uint256 _identifier;
}
```

7.5.2 Utilizzo di *modifier*

I *modifier* utilizzati ci hanno permesso di effettuare controlli di vario tipo: controllare se, dato un identificativo, la sacca è presente nel centro trasfusionale, controllare se una sacca può essere assegnata e, infine, controllare se la struttura sanitaria ha effettuato una richiesta di una nuova sacca da trasfondere.

```
/// @notice Modifier
modifier isValidBloodSack (uint256 _bloodSackNumber) {
    require (_bloodSackNumber < index, "The blood sack number not exist
        ");
    require (address(blood[_bloodSackNumber]._sack) != address(0), "
        Blood sack not exist");
    _;
}

modifier isUsableBloodSack (uint256 _bloodSackNumber) {
    (bool isUsable, ) =
        address(blood[_bloodSackNumber]._sack).call(abi.
            encodeWithSignature("isUsable()"));
    require (isUsable, "Blood sack unusable");
    _;
}

modifier checkSackAddress (address sender) {
    require (organized[owner()][sender] != address(0), "There isn't an
        inventory");
    _;
}
```

Come riportato, tutti i controlli sono stati eseguiti utilizzando l'operatore *require* per, eventualmente, bloccare lo smart contract effettuando un'operazione di *revert*.

7.5.3 Aggiunta di una sacca di sangue

Aggiungere una sacca di sangue, vuol dire creare un nuovo smart contract, passando le informazioni iniziali della sacca come la tipologia, il gruppo sanguigno e Rh. La comunicazione parte da un EOA che, attraverso una transazione, esegue la funzione *addNewBloodSack* la quale permetterà di settare lo stato dello smart contract, restituendo un evento di conferma contenente: l'indirizzo dello smart contract della sacca di sangue, l'indice e l'indirizzo del mittente.

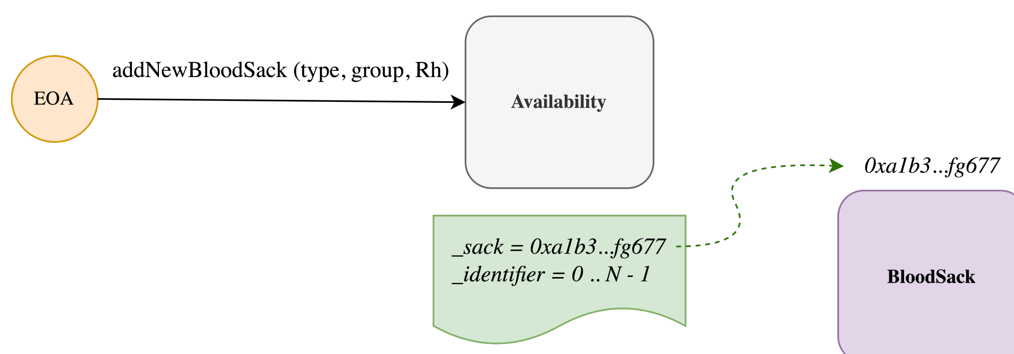


Figura 7.1: Workflow per inserire una sacca

Come si nota dalla *figura 7.1*, è l'EOA che si occupa di far partire la transazione per risvegliare il comportamento dello smart contract *Availability* e, quest'ultimo, si occuperà di effettuare la creazione della sacca di sangue. Bisogna prestare particolare attenzione all'indirizzo del proprietario della sacca di sangue inserita. Durante la creazione, verrà preso come proprietario non l'EOA che ha effettuato la prima richiesta, ma *Availability*.

```
/// @notice Add a new blood sack
function addNewBloodSack (BloodSack.BloodSackType _type, BloodSack.
   BloodGroup _group, BloodSack.BloodRh _Rh) external {
    BloodSack sack = new BloodSack (index, _type, _group, _Rh);
    blood[index]._sack = sack;
    blood[index]._identifier = index;

    /// @notice Emit to return an event in logs
    emit NewBloodSack(address(sack), index, msg.sender);
    index++;
}
```

Nel momento di creazione viene inizializzato un nuovo smart contract, prendendo tra i parametri anche l'identificativo per il tracciamento all'interno del centro trasfusionale, settando lo stato interno del contratto.

```
constructor (
    uint _index,
```

```
    BloodSack.BloodSackType _bloodSackType,
    BloodSack.BloodGroup _bloodSackGroup,
    BloodSack.BloodRh _bloodSackRh
) {
    id = _index;
    _type = _bloodSackType;
    _group = _bloodSackGroup;
    _Rh = _bloodSackRh;
    _inventory = BloodInventory.Unassigned;
    _state = BloodSack.State.Unusable;
}
```

7.5.4 Usabilità della sacca

Una volta inserita la sacca, sarà possibile renderla usabile per una particolare struttura. Per fare ciò, sempre il centro trasfusionale, si occupa di inviare una richiesta allo smart contract della sacca e cambiare lo stato di questa al valore *usable*. La comunicazione avviene sempre attraverso una *call* senza passare nessun parametro. I controlli effettuati riguardano:

- La verifica del proprietario che ha effettuato la chiamata.
- Se la sacca è realmente presente nel centro.

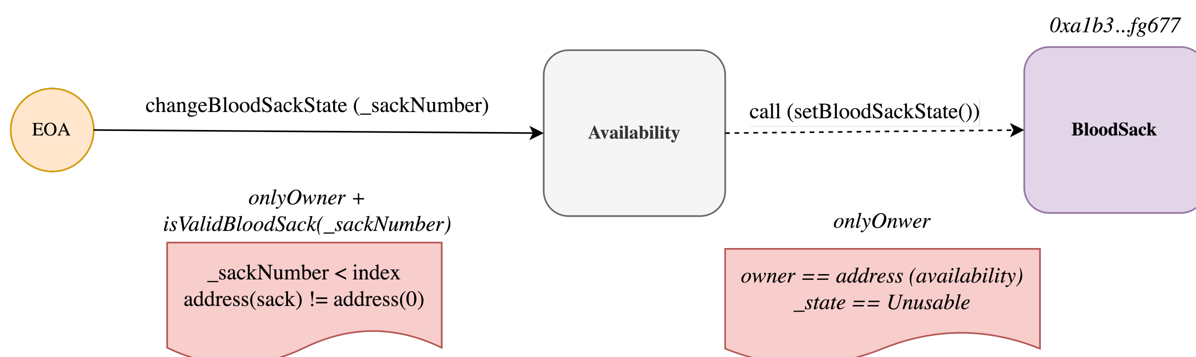


Figura 7.2: Workflow per modificare lo stato di una sacca

```
/// @notice Change blood sack state
function changeBloodSackState (uint256 _bloodSack) external onlyOwner
   isValidBloodSack(_bloodSack) {
    (bool success, ) = address(blood[_bloodSack]._sack).call(abi.
       encodeWithSignature("setBloodSackState()"));

    require(success, "Blood sack state not change");
}
```

Dal codice[14] si nota che, se la richiesta non dovesse andare a buon fine, l'operazione porterebbe ad un *revert*, bloccandola e restituendo un messaggio di errore. A quel punto, bisognerà verificare cosa sia successo all'interno di *BloodSack* per comprendere tale arresto.

7.5.5 Assegnare una sacca di sangue

Prima di assegnare una sacca di sangue, bisognerà verificare la sua disponibilità e la sua usabilità e solo successivamente si potrà procedere.

```
function approve (address demandNode, uint256 sack) external onlyOwner
isValidBloodSack(sack)
isUsableBloodSack(sack)
returns(bool)
{
    require (organized[msg.sender][demandNode] == address(0), "Blood
       sack already assigned");
    organized[msg.sender][demandNode] = address(blood[sack]._sack);
    emit Organized(msg.sender, demandNode, address(blood[sack]._sack));

    delete blood[sack];

    return true;
}
```

La funzione *approve* esegue un ulteriore controllo sul doppio *mapping* per verificare che il nodo di domanda al quale assegnare la sacca non ne abbia già una in fase di richiesta. Dopo aver effettuato tale controllo e aver assegnato la sacca, bisognerà eliminarla dal *mapping* del centro trasfusionale; bloccando eventuali richieste per quella specifica sacca, la quale risulterà con:

- Il campo indirizzo, corrispondente al valore di *address(0), 0x00...0*.
- Il campo dell'identificatore azzerato.

7.5.6 Eliminare una sacca assegnata

L'eliminazione di una sacca, e quindi di uno smart contract in Ethereum, sarà permesso solo una volta assegnata ad un paziente. Tale assegnazione potrà essere fatta solo dall'indirizzo al quale è stato attribuito la sacca, sia esso un EOA o uno smart contract. Nel nostro caso, viene chiamata una funzione esterna che comunica con lo smart contract del centro trasfusionale, il quale dopo aver effettuato le opportune verifiche, si occuperà della cancellazione.

[14] *abi.encodeWithSignature (..., ...)*: permette di codificare una specifica chiamata ad un metodo di uno smart contract, prendendo una stringa rappresentante il nome della funzione da chiamare.

Richiesta esterna di trasfusione

La funzione riportata è presente in uno smart contract, quindi la comunicazione avviene tra diversi smart contract fino ad arrivare alla fase di distruzione nel *BloodSack* di appartenenza.

```
function assignedInventory (address availability) external onlyOwner {
    (bool success , ) = availability.call(abi.encodeWithSignature("
       demandNodeAssignedInventory()"));
    require (success , "You cannot assign inventory");
}
```

Tale funzione comunicherà con *Availability smart contract*, il quale si occuperà di mandare una richiesta al *BloodSack*, modificando lo stato *_inventory* e settandolo al valore *assigned*.

```
/// @notice Set inventory in a blood sack
function demandNodeAssignedInventory () external checkSackAddress(msg.
   sender) {
    (bool success , ) = organized[owner()][msg.sender].call(abi.
       encodeWithSignature("setInventory()"));
    require (success);
}
```

Eliminazione definitiva della sacca

Una volta eseguite le operazioni sopra citate, sarà possibile eliminare la sacca sempre effettuando una *call* verso lo smart contract da distruggere. Come riportato nel **capitolo 6**, utilizzare una *delegatecall* non permetterebbe di usare come contesto di esecuzione quello di *BloodSack*, ma quello di *Availability*, distruggendo lo smart contract che tiene traccia delle varie disponibilità.

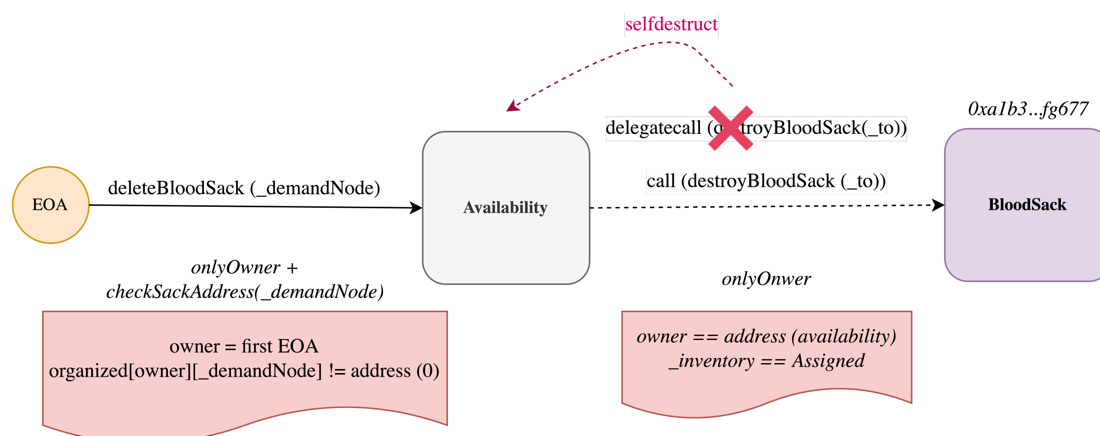


Figura 7.3: Workflow per eliminare una sacca di sangue

Nel codice che segue si può verificare che, come per l'eliminazione della sacca nel centro trasfusionale, anche in questo caso, l'eliminazione dovrà avvenire per rendere nuovamente disponibile il campo *address* per identificare una nuova sacca da assegnare ad una struttura sanitaria; verrà effettuata un'operazione di *delete*, la quale permetterà di "resettare" il campo citato.

```
/// @notice Delete an existing blood sack
function deleteBloodSack (address demandNode) external onlyOwner
    checkSackAddress(demandNode) {
    (bool success, ) = organized[owner()][demandNode].call(abi.
        encodeWithSignature("destroyBloodSack(address)", owner()));
    require (success, "Blood sack cannot be deleted");

    emit DeleteBloodSack(msg.sender, organized[owner()][demandNode]);
    delete organized[owner()][demandNode];
}
```

7.6 Eventi negli smart contract proposti

Gli eventi sono molto utili per notificare le varie modifiche effettuate sui contratti, e possono essere utilizzati per gestire la logica nell'applicazione. All'interno di alcune funzioni, abbiamo inserito degli eventi che vengono scatenati solo quando la funzione chiamata modifica lo stato di un contratto.

Ogni volta che viene inserita una nuova sacca, viene lanciato un evento che comunica l'indirizzo della sacca creata, l'identificatore e il mittente. Allo stesso modo, quando avviene l'eliminazione della sacca, viene preso il mittente che ha effettuato tale operazione (sempre il proprietario dello smart contract nel nostro caso) e l'indirizzo della sacca. Infine, quando viene assegnata una sacca, oltre a visualizzare il mittente e l'indirizzo assegnato, è anche possibile visionare l'indirizzo relativo al nodo di domanda che trasfonderà la sacca richiesta.

```
/// @notice Events
event NewBloodSack (address sackAddress, uint256 identifier, address
    sender);
event DeleteBloodSack (address sender, address sackAddress);
event Organized (address sender, address receiver, address sack);
```

Capitolo 8

Conclusioni

Sicuramente gli smart contract proposti hanno un margine di miglioramento, ma attraverso l'esperimento abbiamo voluto confermare le proprietà che, in un sistema centralizzato, non possono essere rispettate. Quindi, date le premesse e la parte sperimentale, la tecnologia proposta potrebbe essere una possibile soluzione al problema del caso di studio presentato.

Il codice proposto è una prima versione degli smart contract che si occupano della gestione della disponibilità delle sacche di sangue. Una parte aggiuntiva, potrebbe essere quella di fornire effettivamente all'utente una interfaccia grafica sulla quale effettuare le varie operazioni: aggiungere una sacca, distribuirla, eliminarla. Inoltre, potrebbe risultare di aiuto l'aggiunta di alcuni controlli e possibili funzionalità, visionati durante lo studio della tecnologia, di seguito riportati.

8.1 Sviluppi futuri

Gli smart contract presentati potrebbero essere migliorati apportando opportune modifiche. In futuro ci proponiamo di provare a risolvere altri problemi utilizzando il concetto di multi-firma e l'applicazione dello standard token *ERC-20*. Il primo potrebbe essere utile nel momento in cui più responsabili debbano decidere di assegnare una sacca di sangue; quindi, si avrebbe un maggior controllo sulle scelte da fare. Mentre, lo standard token *ERC-20* permette di distribuire un proprio token all'interno della comunità di Ethereum. Il nuovo token proposto potrebbe essere utilizzato per usufruire di trattamenti medici o alcuni medicinali, invogliando le persone a donare una parte del loro sangue o alcune componenti utili.

8.1.1 Concetto di multi-firma

L'idea potrebbe essere quella di avere più responsabili all'interno di un centro trasfusionale che si occupano di validare particolari operazioni su una sacca di sangue. Ad esempio, per approvare una sacca di sangue da consegnare ad una struttura sanitaria, è possibile che tutti i responsabili del centro si debbano mettere d'accordo e firmare la richiesta in modo esplicito.

Possibile idea da sviluppare

Supponiamo di avere a disposizione le seguenti informazioni per firmare una particolare richiesta:

- *mapping (uint256 => mapping (address => bool))*: per ogni sacca di sangue identificata da un valore intero, è possibile verificare se un particolare indirizzo (responsabile nel centro) abbia firmato o meno una richiesta.
- *mapping (address => bool)*: dato un indirizzo, restituisce *true* se è un responsabile, altrimenti false.

```
event Sign (address _signer, uint256 _identifier);

function sign (uint256 _identifier) external {
    require (_identifier <= counter, "The sack number not exist");
    require (signers[msg.sender], "Not signer");
    require (!signed[_identifier][msg.sender], "Signed");

    signed[_identifier][msg.sender] = true;
    sacks[_identifier].sigs += 1;

    emit Sign (msg.sender, _identifier);
}
```

La funzione proposta potrebbe essere un punto di partenza per sviluppare l'idea. In essa possiamo avere dei controlli che ci permettono di verificare, in maniera indipendente, se: l'identificatore della sacca è presente, colui che ha inviato la richiesta di firma è uno dei firmatari e se non ha già firmato una specifica sacca di sangue. Infine, viene applicata la firma e inseriti nei *logs* i dati relativi al firmatario e l'identificatore della sacca firmata.

8.1.2 Standard token ERC-20

Definiamo *ERC-20* uno standard per scrivere un token, il quale espone una interfaccia imposta dalla comunità di *Solidity*. Le interfacce permettono di interagire con altri contratti: grazie ad esse, possiamo avere delle parti astratte del codice, da implementare nel nostro contratto e parti prevedibili di codice che seguano uno standard ben definito.

In generale, un token è qualcosa da distribuire e che tutti gli indirizzi Ethereum possono gestire.

Interfaccia di un ERC-20

Tale standard, fornisce funzioni di base per permettere il trasferimento e l'approvazione, ma anche altre operazioni:

```
event Transfer(address indexed from, address indexed to, uint256 value)
    ;
event Approval(address indexed owner, address indexed spender, uint256
    value);

function totalSupply() external view returns (uint256);
function balanceOf(address account) external view returns (uint256);
function transfer(address to, uint256 amount) external returns (bool);
function allowance(address owner, address spender) external view
    returns (uint256);
function approve(address spender, uint256 amount) external returns (
    bool);
function transferFrom(address from, address to, uint256 amount)
    external returns (bool);
```

Oltre ai metodi proposti che dovranno essere implementati, ci sono anche dei metodi opzionali:

- *name()*: il nome del token generato.
- *symbol()*: il simbolo identificativo del token.
- *decimals()*: la parte decimale da passare al costruttore per avere un numero massimo di token da distribuire.

Sicurezza con lo standard

Inizialmente, lo standard proposto non era presente e alcuni dei token creati precedentemente, non ritornavano un valore di tipo *bool*; non notificando lo stato della modifica apportata sul contratto distribuito. Oggi è obbligatorio notificare tutti i cambiamenti di stato quando si decide di distribuire un nuovo token e interagire con esso.

Implementazione di alcune parti di codice

Di seguito proponiamo alcune implementazioni *grezze* di funzioni, realizzate per proseguire eventuali studi futuri. Le funzioni riportate, in breve, sono: **transfer** che permette

di trasferire una quantità di *token* da un indirizzo ad un altro e, **transferFrom** che permette, una volta resa disponibile una certa quantità di *token* per un indirizzo, di ricevere la sola quantità depositata in una sorta di archivio, distribuendola ad un altro indirizzo oppure a sé stesso. Quest'ultima funzione, quindi, permette di delegare la distribuzione e il controllo di *token* ad un altro indirizzo.

```
function transfer (address to, uint256 amount) external override
    returns(bool) {
     balanceOf[msg.sender] -= amount;
     balanceOf[to] += amount;

     emit Transfer(msg.sender, to, amount);
     return true;
}

function transferFrom (address from, address to, uint256 amount)
    external override returns(bool) {
     allowance[from][msg.sender] -= amount;
     balanceOf[from] -= amount;
     balanceOf[to] += amount;

     emit Transfer(from, to, amount);
     return true;
}
```

La quantità totale dei *token* sarà invece detenuta dall'account che renderà disponibile lo smart contract la prima volta, il quale si occuperà solo in futuro dei vari trasferimenti.

Ringraziamenti

Ci tengo a ringraziare in modo particolare il Prof. Davide Sangiorgi e il Dott. Stefano Pio Zingaro per avermi guidato e consigliato durante tutto il percorso di studio, fornendomi informazioni per lo sviluppo di questa tesi.

Un grazie anche a tutti coloro che mi hanno sostenuto e con i quali ho potuto collaborare nell'ambiente universitario.

Infine, un grazie particolare ai miei familiari, i quali mi hanno permesso di studiare e formarmi presso l'Università di Bologna.

Bibliografia

[LP82] Robert Shostak Leslie Lamport e Marshall Pease. «The Byzantine Generals Problem». In: *SRI International* (1982), pp. 382–401. DOI: `https://lamport.azurewebsites.net/pubs/byz.pdf`.

[Nak08] Satoshi Nakamoto. «Bitcoin: A Peer-to-Peer Electronic Cash System». In: *bitcoin.org* (2008), pp. 1–9. DOI: `https://bitcoin.org/bitcoin.pdf`.

[But13] Vitalik Buterin. «A next generation smart contract and decentralized application platform». In: (2013), pp. 1–36. DOI: `https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf`.

[Ant14] Andreas M. Antonopoulos. *Mastering Bitcoin, Unlocking digital crypto-currencies.* O'Reilly Media Inc., 2014. ISBN: 9781449374044.

[AC14] Damian Gruber Arthur Gervais Ghassan O. Karame e Srdjan Capkun. «On the Privacy Provisions of Bloom Filters in Lightweight Bitcoin Clients». In: (2014), pp. 1–11. DOI: `https://eprint.iacr.org/2014/763.pdf`.

[Woo14] Dr. Gavin Wood. «ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER». In: (2014), pp. 1–41. DOI: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[AW18] Andreas M. Antonopoulos e Dr. Gavin Wood. *Mastering Ethereum, building Smart Contract and DApps.* O'Reilly Media Inc., 2018. ISBN: 9781491971949.

[GM18] Tatiana Gayvoronskaya e Christoph Meinel. *Blockchain Hype or Innovation.* Springer, 2018. ISBN: 9783030615581, 9783030615598.

[XS18] Ingo Weber Xiwei Xu e Mark Staples. *Architecture for Blockchain Applications.* Springer, 2018. ISBN: 9783030030346, 9783030030353.

[APi19] N.Labadiec A.Pirabánac W.J.Guerrerob. «Survey on blood supply chain management: Models and methods». In: *Elsevier* (2019), pp. 1–23. DOI: `https://www.sciencedirect.com/science/article/pii/S030505481930190X`.

[AC20] Paulo Mendes1 Antonio Miguel Rosado da Cruz Francisco Santos1 e Estrela Ferreira Cruz. «Blockchain-based Traceability of Carbon Footprint: A Solidity Smart Contract for Ethereum». In: *Instituto Politecnico de Viana do Castelo, 4900-347* (2020), pp. 258–268. DOI: https://www.scitepress.org/Papers/2020/94126/94126.pdf.

[HM22] Amin Milani Fard Haozhe Zhou e Adetokunbo Makanju. «The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support». In: (2022), pp. 358–378. DOI: https://www.mdpi.com/2624-800X/2/2/19/pdf?version=1654438241.

[Cut] Gioavnni Cutolo. *Una introduzione all'aritmetica modulare.* URL: http://www.dma.unina.it/cutolo/didattica/PLS/2016/AritmeticaModulare.pdf.

[Fab] Francesco Fabris. *Introduzione alla crittografia delle curve ellittiche.* URL: https://moodle2.units.it/pluginfile.php/464885/mod_resource/content/0/0_Dispensa_Curve_Ellittiche_16_05_22.pdf.

[RD14] Pedro B. Velloso Rafael P. Laufer e Otto Carlos M. B. Duarte. «Generalized Bloom Filters». In: (2005, 2014), pp. 1–13. DOI: https://www.researchgate.net/publication/228949877_Generalized_bloom_filters.