Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Tesi di Laurea Magistrale

Studio sull'uso di Java per la programmazione di una blockchain

Candidato:
Luca Vicentini
Matricola VR408207

Relatore:
Prof. Nicola Fausto Spoto

Anno Accademico 2017/2018

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Prima dell'arrivo di Bitcoin erano decenni che circolava l'idea di una moneta digitale decentralizzata, la creazione di Bitcoin non fu quindi frutto della genialità di una singola persona, quanto piuttosto il risultato di un cammino che prese l'avvio con la diffusione delle tecnologie digitali cercando di giungere ad una soluzione che fosse in grado di bilanciare efficienza, sicurezza e decentralizzazione. Da questo punto di vista Bitcoin non appare più come una sorta di "Big Bang", ma come un processo darwininiano di lenta selezione, partito da molto lontano.

Alla fine degli anni '80 David Chaum creò DigiCash che prevedeva una valuta tokenizzata il cui trasferimento poteva avvenire da persona a persona in modo molto simile all'attuale Bitcoin. Venne definita una Blinding Formula che confermava la legittimità e l'esistenza del token impedendo, inoltre, la tracciabilità, da cui il termine Blinding cash.

Successivemente Nick Szabo cercò di lanciare Bit Gold, una valuta digitale decentralizzata in cui per la prima volta si poneva l'accento sulla distribuzione comunitaria della valuta e sulla creazione di un algoritmo di proof-of-work per limitarne la produzione ed evitare fenomeni inflazionistici. Questo progetto è stato da molti definito il diretto precursore dell'attuale Bitcoin ma non fu mai implementato.

Un altro esperimento di moneta digitale prima dell'avvento di Bitcoin e basato su un primo tentativo di proof-of-work, fu HashCash. Si trattava di una valuta digitale che per un certo tempo ebbe anche successo, si poneva come obiettivo anche la capacità di resistere ad attacchi tra cui quelli di DDOS. L'uso di una forma di proof-of-work doveva anche limitarne la distribuzione. Ciò che portò alla decadenza di HashCash fu il superamento dell'algoritmo di PoW da parte della capacità di elaborazione di quel periodo, che ne portò all'iperinflazione.

L'innovazione fornita da Satoshi Nakamoto, pubblicata nel 2008 nel suo paper [10], è l'idea di riuscire a combinare un protocollo molto semplice basato su dei nodi su cui avvengono le transazioni, che danno vita ad una sempre crescente blockchain, attraverso la creazione di "blocchi" ogni 10 minuti, con il proof-of-work come meccanismo attraverso cui i nodi guadagnano il diritto di partecipare al sistema. Il modello blockchain Bitcoin, che analizzeremo nel dettaglio nel capitolo 1, ha dimostrato di essere sufficientemente valido, e nel corso dei successivi anni sembra diventato il fondamento di numerose monete e protocolli di tutto il mondo.

Nasce qui la necessità di creare un protocollo alternativo per la creazione di applicazioni decentralizzate fornendo un insieme eterogeneo di possibilità che, saranno utili per una larga classe di applicazioni decentralizzate, in particolar modo nelle situazioni dove sono essenziali: un tempo rapido di sviluppo, la sicurezza per applicazioni utilizzate di rado, e la capacità di far interagire tra loro, in modo molto efficace, applicazioni differenti. Ethereum, che analizzeremo in tutte le sue parti nel capitolo 2, permette tutto ciò attraverso la costruzione di quello che in sostanza è

un protocollo definito come una Blockchain con un linguaggio di programmazione, costruito al suo interno e turing-completo, che permette ad ognuno di scrivere degli smart contracts e delle applicazioni decentralizzate dove stabilire le proprie regole arbitrarie per la proprietà, i formati delle transazioni e le funzioni di transizione di stato. Il codice nei contratti Ethereum è scritto in un linguaggio di basso livello, linguaggio bytecode a cascata, denominato "codice virtual machine Ethereum" o "codice EVM". Il codice consiste in una serie di bytes, dove ogni byte rappresenta un'operazione. Per la stesura di questi smart contract, il programmatore necessita di un linguaggio object-oriented e di alto livello, che nel caso di Ethereum è rappresentato da Solidity. Questo nuovo linguaggio di programmazione è in continua evoluzione proprio per il fatto che esiste da poco tempo, generando aggiornamenti cumulativi. Questi continui cambiamenti portano ad una inevitabile procedura di aggiornamento da parte del programmatore designato che dovrà essere a conoscenza delle novità apportate da ogni nuova versione del compilatore Solidity, attualmente escono in media 4 versioni nightly a settimana.

Nasce anche da qui la necessità di creare un framework che permetta di sfruttare le competenze e gli strumenti, già definiti e strutturati nel tempo, del linguaggio di programmazione Java con il fine di permettere lo sviluppo di smart contract in modo semplice e confortevole, utilizzando gli strumenti esistenti. Questo nuovo framework, descritto nel paper [18] scritto dal Prof. Nicola Fausto Spoto, prende il nome di Takamaka ed utilizza appunto un sottoinsieme delle librerie Java. L'utilizzo di questo linguaggio per la programmazione di una blockchain non risulta affatto immediato, infatti questa scelta porta con sè una serie di problematiche legate al determinismo, non possiamo infatti permettere l'esecuzione di una procedura che dia risultati differenti su macchine differenti. Un risultato di questo tipo sarebbe infatti disastroso su qualsiasi blockchain. Si dovrà quindi procedere con la definizione di un sottoinsieme di librerie *white-listed* che rappresenteranno le sole utilizzabili all'interno dei contratti scritti con tale framework.

Lo scopo di questo elaborato è la ricerca dei metodi, all'interno delle esistenti librerie Java, che possono in qualche modo comportarsi in modi non deterministici, documentandone il loro scopo e i casi in cui tali metodi possono comportare in modi non prevedibili. Per ogni metodo si dovrà arrivare a delle conclusioni, immaginandone utilizzo all'interno di appositi metodi di contratto. Questo tipo di analisi ci porterà a concludere quando e come certi metodi possano essere utilizzati arrivando ad escludere completamente l'utilizzo di alcuni e vietando l'uso di altri solo in determinate condizioni. Tutta questa analisi, completa di esempi, è possibile trovarla al capitolo 4.

1 Bitcoin

Questo capitolo ha lo scopo di illustrare al lettore i concetti di *Bitcoin*, *Blockchain* e denaro digitale. Una descrizione più dettagliata la si può trovare nel testo *Mastering Bitcoin* [2], dal quale sono state prese la maggior parte delle informazioni per la stesura di questo capitolo. Consigliata è anche la lettura del white paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [10].

1.1 Introduzione

Bitcoin è una collezione di concetti e tecnologie che formano le basi per un sistema di denaro digitale. Le unità di valuta si chiamano, appunto, Bitcoin e vengono utilizzate per scambiare valore tra i partecipanti del network Bitcoin. Gli utenti che intendono partecipare a questa rete utilizzeranno un apposito protocollo utilizzando principalmente il canale Internet, anche se sono possibili altri network di trasporto. Sono stati resi disponibili numerosi software open source che implementano l'intero protocollo Bitcoin su di un'ampia gamma di dispositivi digitali, rendendo questa tecnologia facilmente accessibile e flessibile. A differenza delle monete tradizionali, i Bitcoin sono completamente virtuali, la moneta, infatti, è sottintesa nelle transazioni che trasferiscono valuta dal mittente al ricevente. Chi possiede Bitcoin e intende eseguire una transazione, farà uso di chiavi che permettono di avere la prova di essere i proprietari della transazione in oggetto eseguendo lo sblocco della valuta da spendere che verrà quindi trasferita al nuovo ricevente. Il possesso delle chiavi sopraccitate è l'unica prova che dimostra a chi appartiene ogni singolo Bitcoin o frazione di esso, lo smarrimento di queste chiavi implica quindi la perdita dei rispettivi Bitcoin.

Come si sarà intuito, Bitcoin è un sistema distribuito e peer-to-peer, in quanto non ha alcun server centrale o centro di controllo. I Bitcoin sono creati tramite il processo di *mining* che comporta una competizione nel cercare una soluzione di un complesso problema matematico mentre vengono processate le transazioni. Ogni partecipante al network bitcoin che esegue un'istanza completa del protocollo, può operare come "minatore" utilizzando la potenza di calcolo della propria macchina,

verificando e registrando transazioni. In media ogni 10 minuti qualche nodo sarà in grado di validare le transazioni avvenute nei 10 minuti precedenti e questo miner sarà ricompensato con dei bitcoin nuovi di zecca. Il mining Bitcoin decentralizza le emissioni di valuta, attualmente tipiche di una banca centrale, tramite questa competizione globale.

Un importante algoritmo, incorporato all'interno del protocollo Bitcoin, è quello che regola la funzione di mining su tutto il network. La difficoltà della del problema che i miner devono risolvere è infatti calcolata sulla base della potenza computazionale di tutta la rete in modo che si arrivi ad una soluzione ogni 10 minuti circa. Questo protocollo dimezza anche il tasso con cui i nuovi Bitcoin vengono emessi, questo in media ogni 4 anni. Il numero massimo di Bitcoin creati saranno 21 milioni che, con l'andamento attuale si prevede vengano evasi entro l'anno 2140. A causa delle continua diminuzione del tasso di emissione, la valuta Bitcoin è definita deflazionaria, non è soggetta quindi all'inflazione dovuta alla stampa di nuova moneta oltre il limite dato dal tasso di emissione previsto.

1.1.1 Storia di Bitcoin

Bitcoin è stato inventato nel 2008 con la pubblicazione del paper scientifico intitolato *Bitcoin: A Peer-to-Peer Electronic Cash System* [10] scritto da Satoshi Nakamoto. Combinando varie invenzioni scoperte precedentemente, Nakamoto è arrivato alla creazione di un sistema di contante elettronico completamente decentralizzato che non dipende da nessuna autorità centrale, per l'emissione di nuova moneta o per la liquidazione/validazione delle transazioni. Il Problema del *double-spend*, comune nelle valute virtuali, viene quindi risolto.

La prima implementazione del network Bitcoin nasce nel 2009, in riferimento al paper pubblicato da Nakamoto [10] e da quel momento fu visto e modificato anche da altri programmatori. Il valore di mercato di Bitcoin al momento è stimato attorno ai 75 miliardi di dollari (USD), che dipende dal tasso di cambio BTC/USD.

Satoshi Nakamoto si ritirò dalla scena pubblica nell'Aprile 2011 lasciando la responsabilità di sviluppo del codice e della gestione del network ad una fiorente comunità di volontari. Nessuno, nè Nakamoto nè chiunque altra persona può esercitare alcun controllo sul sistema Bitcoin, che opera sulla base di principi matematici completamente trasparenti.

La soluzione ad un Problema Computazionale Distribuito

L'invenzione di Satoshi Nakamoto è la soluzione pratica ad un problema precedentemente irrisolto, conosciuto come *Byzantine Generals' Problem*. Questo problema consiste nel tentativo a concordare una linea d'azione condivisa attraverso lo scambio di informazioni su di una rete (network) non affidabile e potenzialmente compromessa. La soluzione proposta da Satoshi Nakamoto usa il concetto di *proof-of-work* per raggiungere il consenso comune senza l'utilizzo di un'autorità centrale. Questo nuovo approccio ha avuto notevoli ripercussioni sulle scienze informatiche che studiano la computazione distribuita soprattutto perché può essere applicata anche ad applicazioni molto diverse da quelle monetarie. L'utilizzo di questa tecnologia può essere utilizzata per arrivare al consenso, su reti decentralizzate, per provare l'equità di elezioni, lotterie, registro di beni, notarizzazione digitale e molto altro.

Tipologie di client Bitcoin

Per entrare a far parte del network bitcoin ed iniziare ad usare la valuta, tutto ciò che l'utente deve fare è scaricare un'applicazione oppure utilizzare una web app disponibile su appositi portali. Dal momento in cui Bitcoin è diventato uno standard sono state sviluppate numerose implementazioni del client bitcoin. L'applicazione di riferimento per questi progetti risulta essere *Satoshi client*, progetto open source gestito da un team di sviluppatori derivata dall'iniziale implementazione di Nakamoto. Esistono tre forme principali di client bitcoin:

1. **Full client:** o full node è quel client che salva la storia completa delle transazioni bitcoin e gestisce i wallet dell'utente. Questa tipologia di client è autonomo e non dipende quindi da altri nodi, questo comporta un notevole consumo di memoria occupata appunto dalla *blockchain*[1]

2. **Lite client:** client più leggero del precedente infatti gestisce i wallet utente localmente ma fa affidamento a server di terze parti per accedere alla rete e alle transizioni bitcoin. Questo client da la possibilità ad un utente di connettersi alla rete bitcoin ma senza dover riservare numerosi GB per la memorizzazione di tutta la blockchain, attualmente sono necessari più di 190 GB.

3. **Web client:** classico client al quale si accede tramite un web browser. Questa tipologia di client non salva alcuna informazione in locale infatti anche i wallet utente sono salvati sul server di terze parti che fornisce tale servizio.

La scelta del client bitcoin è molto importante e dipende da quanto controllo si vuole avere sui propri fondi. Un *full client* offrirà all'utente un livello più alto di controllo ed indipendenza, ma fa ricadere il peso dei backup, sicurezza e spazio su disco interamente sulle sue spalle. Al contrario, un *web client*, risulta di più facile utilizzo ma il lato negativo di una simile soluzione è che introduce un rischio di controparte. Infatti la sicurezza e il controllo dei fondi è nelle mani del gestore del servizio.

Installazione e primo avvio

Come abbiamo detto in precedenza esistono numerosi software che implementano i client bitcoin, sul testo cui facciamo riferimento [2] viene utilizzato *Multibit*[2], poi nella successiva versione del libro *Mastering Bitcoin - Programming the Open Blockchain*[1] viene giustamente abbandonato e sostituito con una più recente applicazione per dispositivi mobili (smartphone android) chiamata *Mycelium*. Dopo l'installazione di questa nuova applicazione, al primo accesso, viene creato in automatico un nuovo wallet nel quale possiamo vedere una sequenza di lettere e numeri,

[1] La blockchain è una tecnologia che permette di creare e gestire un grande database distribuito per la gestione di transazioni condivisibili tra più nodi di una rete. Si tratta di un database strutturato in Blocchi, ognuno contenente più transazioni, che sono tra loro collegati in rete in modo che ogni transazione avviata sulla rete debba essere validata dalla rete stessa nell'analisi di ciascun singolo blocco. La Blockchain risulta così costituita da una catena di blocchi che contengono ciascuno più transazioni.

[2] Client Bitcoin disponibile su multibit.org, questo software non è più supportato pertanto gli sviluppatori consigliano di spostarsi su di un qualsiasi altro client. [9]

`1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK`, troviamo poi anche un *QR Code* che rappresenta la stessa informazione ma che può essere più facilmente scambiata con gli altri utenti tramite l'utilizzo della fotocamera. La stringa di cui stiamo parlando rappresenta appunto l'*Indirizzo Bitcoin*[3] dell'utente.

Ottenere i primi Bitcoin

Il primo e a volte più difficile step è proprio quello di acquisire bitcoin perché, diversamente dalle altre monete fisiche, non è ancora possibile l'acquisto di bitcoin in banca o nei chioschi dei cambia-valuta.

Le transazioni bitcoin sono irreversibili a differenza degli ormai classici metodi di pagamento tramite carte di credito/debito, Paypal e trasferimenti bancari che sono reversibili. Nasce perciò il rischio che un qualsiasi venditore di bitcoin venga truffato da utenti che, una volta ricevuti i rispettivi bitcoin, annulli il versamento effettuato tramite i classici pagamenti elettronici. Per mitigare questi fatti un venditore di bitcoin attende qualche giorno/settimana prima di inviare i rispettivi bitcoin, soprattutto se si tratta di un nuovo utente.

Inviare e Ricevere bitcoin

Una volta che l'utente ha portato a termine l'installazione del software di gestione del proprio wallet, verrà creato in automatico, oltre all'indirizzo bitcoin, anche una *Private Key* casuale. L'indirizzo bitcoin, infatti, non è altro che un numero casuale che corrisponde ad una chiave privata che l'utente può utilizzare per accedere ai propri fondi. Fino al momento in cui questo indirizzo sarà referenziato come ricevente di un determinato valore in una transazione, inserita nel registro di transazioni bitcoin (*blockchain*), è semplicemente parte del gran numero di indirizzi possibili, ritenuti validi in bitcoin. Quando sarà associato ad una transazione, diventerà parte degli indirizzi conosciuti sul network bitcoin e l'utente cui ci riferiamo, potrà controllare il suo saldo sul registro pubblico.

Per meglio comprendere le prossime azioni, introduciamo gli attori Alice e Joe. Alice intende acquistare dei bitcoin da Joe, più precisamente ha intenzione di investirci 10$. Alice cede una banconota da 10$ a Joe assieme all'indirizzo bitcoin che le appartiene, in questo modo Joe sarà in grado di inviarle la somma equivalente in bitcoin. A questo punto Joe dovrà scoprire l'attuale tasso di cambio (exchange rate[4]) in modo che possa darle la somma corretta di bitcoin. Attualmente 1 BTC vale 3762,23$ statunitensi, perciò Joe dovrà versare ad Alice 0,0027 BTC[5] in cambio di 10 dollari (USD). Una volta scoperto il prezzo equo per lo scambio, Joe aprirà la propria applicazione wallet e selezionerà la funzione invio bitcoin e procederà con l'inserimento dell'indirizzo del destinatario, Alice e della somma di bitcoin da inviare.

[3]Gli indirizzi Bitcoin iniziano sempre con i caratteri 1 o 3, possono essere visti come un indirizzo email con la sola differenza che possono essere creati nuovi indirizzi bitcoin a volontà e ciascuno di essi redirigerà il denaro delle transazioni che lo utilizzano verso lo stesso wallet. Una pratica molto usata per migliorare la privacy sta nel creare nuovi indirizzi per ogni transazione.

[4]Esistono svariate applicazioni che permettono di risalire al tasso di cambio corrente tra cui: Bitcoin Charts, Bitcoin Average, ZeroBlock, Bitcoin Wisdom, ecc...

[5]Valore di mercato durante la stesura del libro Mastering Bitcoin [2]. Durante la stesura di questo elaborato, un dollaro statunitense equivale a 0,00026 BTC.

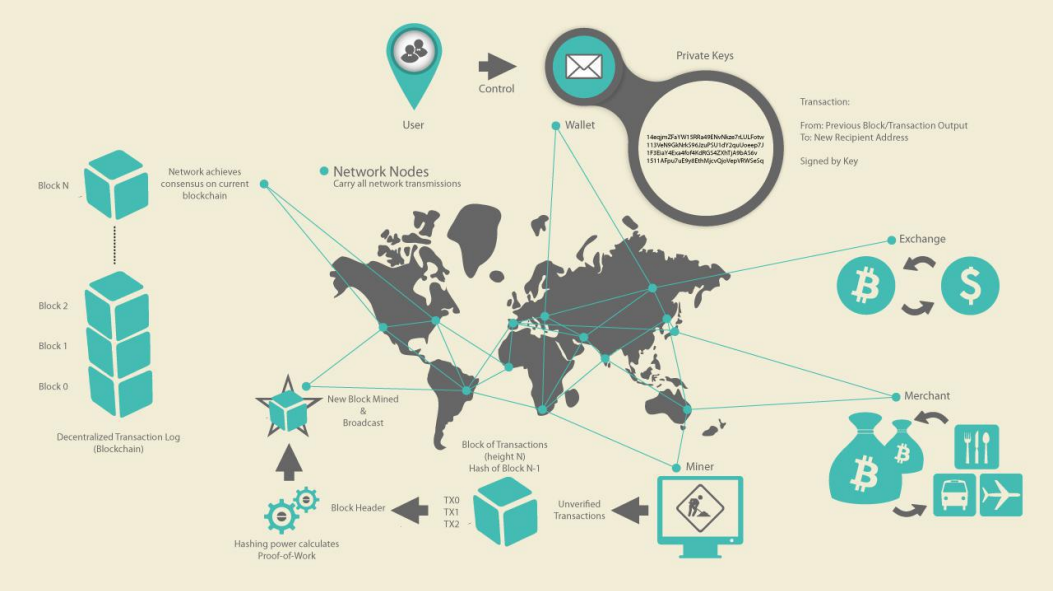


Figura 1.1: Panoramica su bitcoin

A questo punto viene costruita una transazione che assegna all'indirizzo di Alice 0,0027 bitcoin, pescando i fondi dal wallet di Joe che ha autorizzato la transazione. Non appena questa transazione sarà trasmessa attraverso il protocollo peer-to-peer, si propagherà velocemente attraverso il network. In meno di un secondo la maggior parte dei nodi riceveranno tale transazione che verrà processata e successivamente registrata su di un registro globale (blockchain). A questo punto Alice può controllare che tutto sia andato a buon fine accedendo alla propria applicazione wallet che le dovrebbe indicare un saldo di 0,0027 BTC. Se non dovesse trovare i fondi potrebbe significare che la transazione è si stata propagata sulla network ma non è stata ancora inclusa nel registro delle transazioni bitcoin, in questo caso Alice vedrà comunque la transazione di Joe ma con lo stato *Unconfirmed*.

1.1.2 Come funziona Bitcoin

Il sistema bitcoin, al contrario dei sistemi bancari e dei classici sistemi di pagamento, è basato sulla decentralizzazione della fiducia. Invece di un'autorità fiduciaria centrale la fiducia si ottiene come proprietà emergente dall'iterazione tra diversi partecipanti nel network bitcoin. In questo capitolo esamineremo bitcoin eseguendo una singola transazione attraverso tale sistema, osservando quindi come viene gestita la fiducia e come tale transazione viene accettata dal meccanismo di consenso distribuito e venga quindi registrata sulla blockchain, il libro mastro di tutte le transazioni. Procedendo nella lettura si troveranno degli esempi di transazioni che sono state effettivamente eseguite sul network, ma che verranno associate a virtuali interazioni tra attori come Alice, Bob e Joe. Per tracciare una transazione in particolare attraverso il network bitcoin e la blockchain, si può utilizzare un blockchain explorer, piattaforma online che opera come un motore di ricerca bitcoin. Questi servizi permettono di cercare per indirizzo, transazione e per blocco e vedere i relativi flussi all'interno di questi. Tra i Blockchain Explorers più conosciuti troviamo:

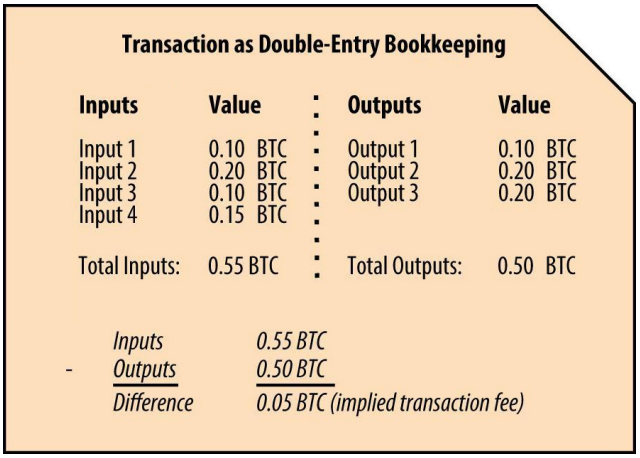


Figura 1.2: Double entry transaction

- Blockchain info https://www.blockchain.com/explorer
- Bitcoin Block Explorer https://blockexplorer.com/
- Insight https://insight.bitpay.com/
- Coinbase https://www.coinbase.com/

Ma vediamo una panoramica del sistema bitcoin, vedi Figura 1.1, notiamo che tale sistema è composto di utenti con i propri portafogli (*wallet*) che contengono delle chiavi (*key*), le transazioni si propagano attraverso il network, i miner producono, tramite una gara di computazione, la blockchain del consenso che rappresenta il libro mastro autoritativo di tutte le transazioni.

1.1.3 Transazioni Bitcoin

Una transazione comunica la network che il proprietario di un certo numero di bitcoin ha autorizzato il trasferimento di una parte di essi ad un altro proprietario. Quando tutta l'operazione andrà a buon fine il nuovo proprietario potrà spendere questi bitcoin e di conseguenza spenderli creando una nuova transazione verso un altro utente. Tutto questo genera una catena di passaggi di proprietà. Ogni transazione può contenere uno o più *input*, che sono i debiti verso un account bitcoin. Dall'altro lato della transazione ci sono uno o più *output*, che rappresentano i crediti aggiunti ad un account bitcoin, vedi Figura 1.2. Gli *input* e gli *output* se sommati non totalizzano necessariamente lo stesso risultato. Al contrario, gli *output* risultano poco inferiori rispetto a quello degli *input* e tale differenza rappresenta la *Transaction fee* o Commissione di Transazione sottintesa, che è un piccolo pagamento che il miner ottiene includendo la transazione all'interno del registro. Nella sezione precedente illustravamo come un nuovo potenziale utente potesse ricevere bitcoin in cambio di denaro. Era il caso di Alice e Joe. Tale transazione ha un numero di bitcoin bloccati attraverso la chiave di Alice. Il pagamento che ora Alice intende fare a Bob fa riferimento alla transazione precedente come un input e crea nuovi output per pagare la somma dovuta e per ricevere il resto. Questa rappresenta una vera

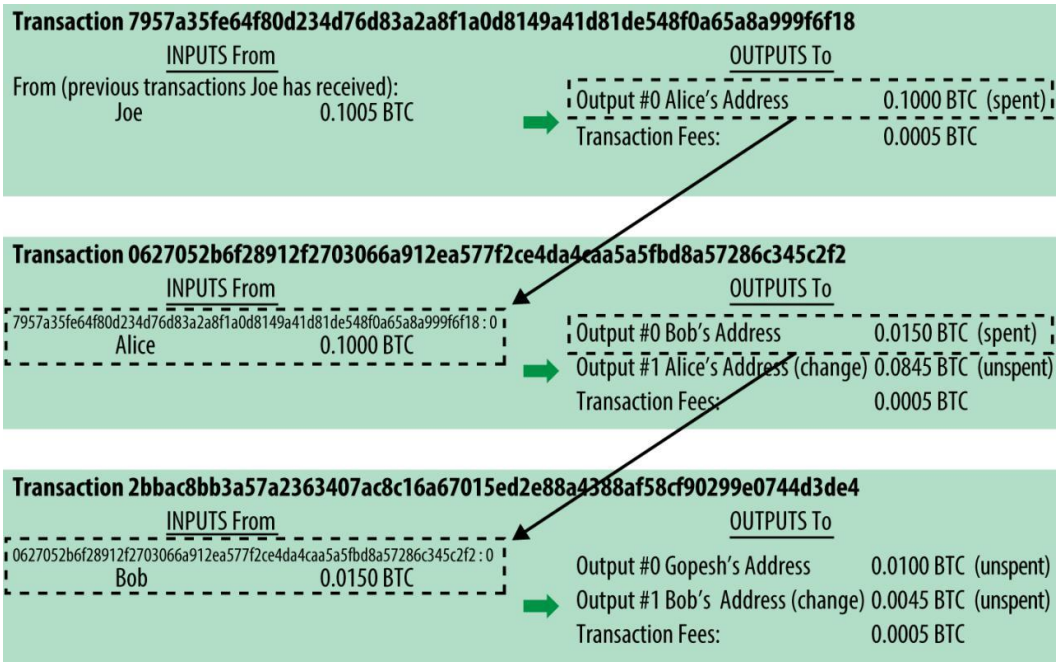


Figura 1.3: Esempio di catena di transazioni

e propria catena, nella quale gli input dell'ultima transazione corrispondono agli output delle transazioni precedenti. La chiave di alice fornisce la firma che sblocca questi importi in output delle precedenti transazioni, provando a tutta la network che lei è la legittima proprietaria di quei fondi. Un esempio di catena la si può vedere in Figura 1.3.

Tipologie di transazione

Ci sono vari tipi di transazione in base alla quantità di input e output all'interno di essa. Ma vediamoli nel dettaglio:

- **Common Transaction:** la forma più comune di transazione, spesso include un resto e viene ritornato al proprietario originario. Questo tipo di transazione ha un solo input e due output.

- **Aggregating Transaction:** aggrega multipli input in un singolo output. Questa operazione nel mondo reale equivale allo scambiare una pila di monete e banconote per una banconota singola di valore maggiore. Viene comunemente utilizzata per far pulizia da transazioni di valore piccolo, ricevute come resto di numerosi pagamenti precedenti.

- **Distributing Transaction:** transazione che distribuisce un input a più di un output che rappresentano multipli destinatari. Questo tipo di transazione sono talvolta utilizzate dagli esercizi commerciali per distribuire i guadagni, ad esempio quando l'azienda effettua il pagamento degli stipendi ai vari dipendenti.

1.1.4 Costruire una transazione

L'applicazione wallet di Alice contiene un'apposita logica che fa in modo di selezionare gli input e output appropriati per costruire una transazione secondo le specifiche imposte da Alice che dovrà solamente specificare una destinazione e l'importo desiderato, il resto viene fatto dall'applicazione.

Ottenere gli input giusti

L'applicazione wallet di alice deve essere in grado di comporre una transazione inserendo gli input, intesi come frammenti di bitcoin che le appartengono, del valore che intende inviare come pagamento a Bob. Le varie tecniche per fare ciò dipendono dalla tipologia di client che Alice possiede oppure dal fatto che tenga aggiornato un'indice completo dei propri unspent output di ogni transazione nella blockchain. Nel caso Alice non mantenga aggiornato un indice del genere, ha comunque la possibilità di ricavare tale insieme di unspent bitcoin tramite svariate API anche utilizzando un semplice client HTTP.


```
{
   "unspent_outputs":[
      {
         "tx_hash":"186f9f998a5...2836dd734d2804fe65fa35779",
         "tx_index":104810202,
         "tx_output_n": 0,
         "script":"76a9147f9b1a7fb68d60c536c2fd8...f3cc025a888ac",
         "value": 10000000,
         "value_hex": "00989680",
         "confirmations":0
      }
   ]
}
```


Quella mostrata nel riquadro qui sopra risulta la risposta ad una possibile richiesta da parte di Alice che contiene un unspent output. Viene riportata anche la referenza alla transazione nella quale questo unspent bitcoin è contenuto e il relativo valore in bitcoin, 10 milioni, equivalente a 0.10 BTC. Con questa informazione il wallet di Alice potrà costruire una transazione utilizzando questo valere come input e quindi trasferire quel valore all'indirizzo del nuovo proprietario.

NB: In questo caso il wallet di Alice dispone di un unspent output con abbastanza bitcoin che riesce a coprire la sua spesa. In alcuni casi può succedere di non disporre di un unico unspent output che riesce a coprire il debito da saldare. In questi casi si dovrà procedere con l'inserimento all'interno della transazione di altri unspent output fino a coprire la spesa da sostenere. In ogni caso Alice dovrà attendersi del resto ma quello lo vedremo nella prossima sezione.

Creare gli output

La transazione che sta per creare Alice, oltre agli input, include almeno un nuovo output che sarà creato facendo in modo che possa essere riscattato solamente dal legittimo proprietario, in questo caso Bob. Tutto ciò è possibile creando l'output

Figura 1.4: Transazione tra Alice e Bob

in questione sotto forma di script che creerà un blocco sul valore e potrà essere riscattato solamente da chi sottoporrà una soluzione, in questo caso ci si aspetta la firma della chiave corrispondente all'indirizzo pubblico di Bob. Solo il wallet di Bob potrà presentare una simile firma dal momento che è il creatore di tale chiave pubblica. Questa transazione inoltre dovrà includere un'ulteriore output, dal momento in cui i fondi di Alice sono nella forma di un output di 0.10 BTC, un valore elevato se paragonato al pagamento di una cifra d'esempio di 0.015 BTC. In questo caso ad alice spetterebbero 0.085 BTC di resto. Questa cifra andrà rappresenta il secondo output della transazione che sarà indirizzato ad Alice. Infine, per fare in modo che il network processi la transazione in tempi ragionevoli, l'applicazione wallet di Alice aggiungerà una piccola commissione di transazione (fee). Questa non viene esplicitata nella transazione ma è data dalla differenza tra gli input e gli output, quindi se Alice imposterà l'output resto a 0.0845 BTC avanzeranno 0.0005 BTC. Questa transaction fee sarà recuperata dal miner come commissione per aver incluso la transazione all'interno di un blocco e averla scritta all'interno del registro blockchain. La transazione risultante, una volta confermata ed inviata sul network, può essere vista utilizzando un'applicazione web del tipo blockchain explorer, vedi Figura 1.4.

Aggiungere la transazione al Ledger

In questo momento la transazione tra Alice e Bob viene inviata al network e, come vediamo in Figura 1.4, risulta avere una grandezza di 258 byte. Come abbiamo visto, all'interno di essa troviamo tutto il necessario per confermare la proprietà dei fondi e assegnare loro nuovi proprietari. La presenza di tutte informazioni danno modo a Bob di verificare fin da subito la transazione controllando che gli input utilizzati corrispondano a precedenti unspent output oltre a verificare che gli importi, comprese le fee, siano adeguati. Bob dovrà aspettare in media 10 minuti per vedere la propria transazione come *confermata*, quindi inserita all'interno del Ledger.

1.2 Il Mining di Bitcoin

Una transazione non entra a far parte del libro mastro condiviso (blockchain) fino a che non viene verificata ed inclusa in un blocco da un processo chiamato *mining*. Il processo di mining in bitcoin serve a due scopi:

- Il processo di mining genera nuovi bitcoin per ogni blocco. La quantità di bitcoin creata per ogni blocco è fissa e diminuisce col passare degli anni.
- Il mining genera fiducia assicurando che le transazioni siano confermate solo se è stata usata una sufficiente potenza di calcolo. Il problema da risolvere è basato su di un hash crittografico.

Ogni 10 minuti un miner entra a far parte di una competizione globale tra miners, alla ricerca di una soluzione per un blocco di transazioni. Trovare tale soluzione, chiamata *proof-of-work*, richiede decine di miliardi di operazioni hash al secondo in tutta la rete bitcoin. L'algoritmo di *proof-of-work* comporta l'effettuare ripetutamente un'operazione di hashing dell'header del blocco e un numero casuale con l'algoritmo crittografico SHA256 fino a che non emerge una soluzione corrispondente a un determinato pattern. Il primo miner a trovare tale soluzione vince il round della competizione e pubblica il blocco nella blockchain, aggiudicandosi dei bitcoin nuovi di zecca.

Al momento della scrittura di questa Tesi la difficoltà di questi problemi è così alta da rendere redditizio effettuare mining solamente con circuiti integrati specifici per l'applicazione selezionata[6], nel nostro caso si tratta di centinaia di algoritmi di mining direttamente stampati su chip hardware, eseguito poi in parallelo su di un singolo chip in silicio. Un miner più far parte di una mining-pool nella quale i partecipanti dividono gli sforzi e i proventi.

Mining delle Transazioni presenti nei Blocchi

Una transazione trasmessa attraverso il network bitcoin non è verificata fino a quando non viene inglobata all'interno di un blocco che verrà inserito sulla cima della blockchain, questo accade in media ogni 10 minuti. Nuove transazione fruiscono costantemente nel network da wallet utente e altri applicativi, non appena queste vengono notate dai vari minier verranno aggiunte ad una pool temporanea di transazioni non verificate, pool mantenuta da ogni singolo nodo. Mano a mano che i miner cercano di comporre un nuovo blocco vanno a verificare e quindi ad aggiungere nuove transazioni al blocco su cui lavorano, dando priorità alle transazioni la cui fee risulta più alta. Ogni miner inizia il processo di mining di un blocco di transazioni nel momento in cui riceve il blocco precedente dal network, prendendo atto di aver perso l'ultimo round della competizione. Il miner crea immediatamente un nuovo blocco, lo riempie con transazioni e con le informazioni (l'hash) del blocco precedente, e inizia a calcolare la proof-of-work per il nuovo blocco. Ogni miner include una transazione speciale nel suo blocco, una che paga al proprio indirizzo bitcoin una ricompensa per i nuovi bitcoin creati (attualmente 12.5 BTC per blocco). Se il miner trova una soluzione che rende il blocco valido, egli "vince" questa ricompensa

[6] ASIC, Application Specific Integrated Circuits

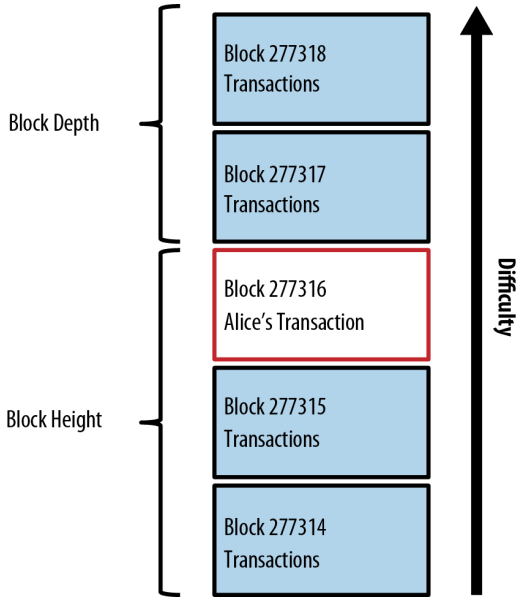


Figura 1.5: Transazione di Alice inclusa nel blocco `#277316`

perché il suo blocco "vincente" è aggiunto alla blockchain globale e la transazione di ricompensa che lui ha incluso in esso diventa spendibile. Supponendo che un miner vinca la competizione, procederà col diffondere nel network il nuovo blocco, `#277316`, contenente 419 altre transazioni. Alla ricezione di questo blocco gli altri miner lo validano e iniziano a contendersi la generazione del blocco successivo. Qualche minuto più tardi un altro miner troverà una soluzione per il successivo blocco, `#277317` che andrà a posizionarsi sulla cima della blockchain. Dal momento in cui questo nuovo blocco è basato sul blocco precedente, che conteneva la transazione di Alice, ha aggiunto una grande quantità di computazione su quel blocco, rafforzando quindi la fiducia riposta in tutte le transazioni presenti nel blocco `#277316`. Man mano che i blocchi si impilano uno sopra l'altro, diventa esponenzialmente più difficile invertire la transazione, rendendo il network sempre più sicuro. Questo aspetto lo si può notare graficamente nella Figura 1.5 dove vediamo il blocco `#277316`, contenente la transazione di Alice, al di sotto del quale troviamo 277316 blocchi fino, infatti ad arrivare al blocco `#000000`, conosciuto anche come *genesis block*. Per convenzione, ogni blocco con più di 6 conferme è considerato irrevocabile dal momento in cui si necessiterebbe di un'ammontare immenso di computazione per invalidarlo e di conseguenza revocare anche i successivi sei blocchi. Da questo punto in poi la transazione di Alice è stata inclusa nella blockchain e sarà visibile a tutte le applicazioni bitcoin. Ogni client bitcoin può verificare indipendentemente la transazione come valida e spendibile. Sarà in oltre possibile tracciare a ritroso il percorso completo del denaro fino alla fonte.

1.3 Chiavi, Indirizzi e Wallet

La proprietà dei bitcoin si stabilisce attraverso chiavi digitali, indirizzi bitcoin e firme digitali. Le chiavi digitali sono create e mantenute dai vari utenti in un

file o database chiamato wallet. Le chiavi digitali nel wallet di un utente sono completamente indipendenti dal protocollo bitcoin e possono essere generate e gestite dal software wallet dell'utente senza alcuna relazione con la blockchain o accesso a Internet. Esse consentono molte delle interessanti proprietà dei bitcoin, incluso controllo e fiducia decentralizzata.

Ogni transazione bitcoin richiede una firma valida per essere inclusa nella blockchain, che può essere generata solo con chiavi digitali valide, pertanto chiunque disponesse di una copia di quelle chiavi ha il controllo del bitcoin in quel conto. Si tratta di coppie di chiavi composte da una chiave privata, che deve rimanere segreta, e una chiave pubblica. Sono in genere memorizzate all'interno di un file del wallet e vengono gestite dal relativo software.

Nella parte di pagamento di una transazione bitcoin, la chiave pubblica del destinatario è rappresentata dalla sua impronta digitale, chiamata indirizzo bitcoin, che viene utilizzata allo stesso modo del nome del beneficiario su un assegno. Nella maggior parte dei casi, un indirizzo bitcoin generato corrisponde ad una chiave pubblica. Tuttavia, non tutti gli indirizzi bitcoin rappresentano chiavi pubbliche; possono anche rappresentare altri beneficiari come gli script, che vedremo più avanti.

1.3.1 La Crittografia a Chiave Pubblica e le Cripto-valute

La crittografia a chiave pubblica è stata inventata negli anni '70 ed è una delle basi matematiche della sicurezza informatica. Da quando è stata inventata la crittografia a chiave pubblica, sono state scoperte diverse funzioni matematiche idonee, come ad esempio l'elevamento a potenza dei numeri primi e la moltiplicazione a curva ellittica. Queste funzioni matematiche sono praticamente irreversibili, nel senso che sono facili da calcolare in una direzione e risultano essere molto difficili da calcolare nella direzione opposta, queste funzioni sono dette *one-way function*. Sulla base di queste funzioni matematiche, la crittografia consente la creazione di segreti digitali e firme digitali non falsificabili. Bitcoin usa la moltiplicazione a curva ellittica come base per la sua crittografia a chiave pubblica. In bitcoin, usiamo la crittografia a chiave pubblica per creare una coppia di chiavi che controlla l'accesso ai bitcoin. La coppia di chiavi consiste di una chiave privata e un'unica chiave pubblica, l'una derivata dall'altra. La chiave pubblica è usata per ricevere bitcoin, e la chiave privata è usata per autorizzare transazioni per la spesa della valuta. C'è una relazione matematica tra la chiave pubblica e quella privata che permette alla chiave privata di essere usata per generare firme sui messaggi. Questa firma può essere validata attraverso la chiave pubblica senza rivelare la chiave privata. Nella maggior parte delle implementazioni di wallet, le chiavi privata e pubblica sono salvate insieme come coppia di chiavi, per convenienza. Comunque, la chiave pubblica può essere ricavata dalla chiave privata, è quindi possibile salvare anche solo la chiave privata. Un portafoglio bitcoin contiene una raccolta di coppie di chiavi, ognuno composto da una chiave privata e una chiave pubblica. Vediamo come vengono generate queste chiavi:

- La chiave privata (k) è un numero, di solito scelto a caso.

- Dalla chiave privata, viene utilizzata la curva ellittica, una funzione crittografica one-way, per la generazione di una chiave pubblica (K).

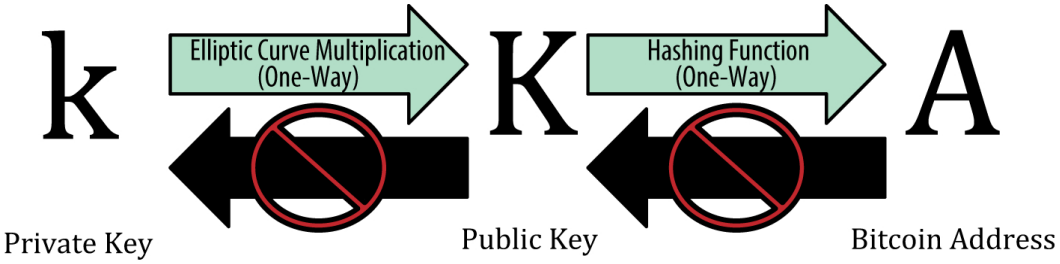


Figura 1.6: Chiave privata, chiave pubblica e indirizzo bitcoin

- Dalla chiave pubblica (K), viene utilizzata una funzione di hash crittografica one-way per la generazione di un indirizzo bitcoin (A).

La rappresentazione grafica di tutto questo la si può vedere in Figura 1.6.

Le chiavi private

Come abbiamo visto poco fa, una chiave privata è semplicemente un numero scelto casualmente. È utilizzata per creare le firme necessarie per trasferire i bitcoin, con lo scopo di dimostrare la proprietà dei fondi utilizzati nella transazione. La chiave privata deve essere:

- **Segreta:** rivelarla a terzi equivarrebbe a dare loro il controllo dei bitcoin associati a tale chiave.
- **Protetta da perdite:** deve infatti essere conservata e protetta da perdite accidentali, se questo accadesse i fondi associati ad essa saranno anch'essi persi per sempre.

Il primo e più importante passo per generare chiavi è quello di trovare una fonte sicura di entropia, scegli un numero tra 1 e 2^{256}. Il metodo utilizzato per la selezione di tale numero non risulta importante fintanto che non sia prevedibile. Il software attuale utilizza il generatore di numeri casuali fornito dall'OS inizializzato però da una fonte esterna di casualità. Più precisamente, la chiave privata potrebbe essere un qualsiasi numero tra 1 e $n-1$, dove n è una costante ($n = 1.158 * 10^{77}$, un po' meno di 2^{256}), definito come ordine di curvatura della curva ellittica usata in bitcoin. Si sceglierà un numero casuale di 256 bit inferiore di $n-1$. Questo avviene eseguendo l'algoritmo SHA256 su di una stringa casuale s dove $|s| > 256$ bit, questo produrrà comunque un numero di 256 bit. Se tale numero risulterà inferiore di $n-1$ verrà accettato, altrimenti si ripeterà l'algoritmo con un input diverso. La seguente è una chiave privata casuale k mostrata nella sua rappresentazione esadecimale (256 cifre mostrate come 64 cifre esadecimali, ognuna da 4 bit):

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

Per generare una nuova chiave tramite il client Bitcoin Core, si utilizza il comando *getnewaddress*. Per ragioni di sicurezza verrà mostrata solamente la chiave pubblica. È comunque possibile chiedere a Bitcoin Core di esporre anche le chiave privata tramite il comando *dumpprivkey* che la mostrerà in formato Base58 checksum-encoded chiamato anche *Wallet Import Format* (WIF).

```
$ bitcoind getnewaddress
 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
 KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

NB: Il comando *dumpprivkey* non genera una chiave privata da una chiave pubblica, visto che è impossibile. Il comando semplicemente rivela la chiave privata che è già conosciuta al wallet e che è stata generata dal comando *getnewaddress*.

Le chiavi pubbliche

La chiave pubblica è calcolata da quella privata utilizzando la proprietà di moltiplicazione delle curve ellittiche, operazione irreversibile: $K = k * G$ dove k è la chiave privata, G è un punto costante della curva ellittica chiamato *Generator Point* e K sarà la chiave pubblica risultante. L'operazione inversa, conosciuta come trovare il logaritmo discreto, calcolando k conoscendo K, risulta difficile come provare per tutti i possibili valori di k, brute-force search.

Crittografia e Curve Ellittiche

Le curve ellittiche in crittografia sono un tipo di crittografia asimmetrica, utilizzano coppie di chiavi (Pk, Sk), basata sul problema del logaritmo discreto. La curva utilizzata da bitcoin è definita in uno standard detto $secp256k1$ dal National Institute of Standards and Technology (NIST). La $secp256k1$ è definita dalla seguente funzione:

$$y^2 = (x^3 + 7) \text{ over } (\mathbb{F}_p) \quad \text{or} \quad y^2 \bmod p = (x^3 + 7) \bmod p$$

Il mod$p$ indica che questa curva è su di un campo finito di numeri primi di ordine p, che possiamo definire come $\mathbb{F}_p$ dove $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, in parole povere, un numero primo molto grande. Nella matematica delle curve ellittiche c'è un punto chiamato *point as infinity*, il quale viene fatto corrispondere allo zero nell'addizione. Esiste un operatore $+$, chiamato addizione che dati due punti P_1 e P_2 sulla curva ellittica, $\exists P_3 \quad | \quad P_3 = P_1 + P_2$. Geometricamente, questo nuovo punto P_3 viene calcolato tracciando una linea tra P_1 e P_2. Questa linea intersecherà la curva ellittica nel punto addizionale chiamato appunto $P_3 = (x, y)$. Tale punto lo possiamo riflettere sull'asse delle x ottenendo $P_3 = (x, -y)$. Esistono vari casi particolari di queste addizioni tra punti, se il lettore ne fosse interessato è consigliata la lettura dei testi [2] e [1].

Generazione della chiave pubblica

Partendo da una chiave privata k, numero generato casualmente, lo moltiplichiamo con un punto predeterminato sulla curva, denominato *generator point* G producendo un ulteriore punto situato in qualche altra parte della curva. Questo punto sarà la chiave pubblica K.

$$K = k * G$$

Poiché il punto di generazione G è sempre lo stesso per tutti gli utenti di bitcoin, una chiave privata k moltiplicata per G darà sempre come risultato la stessa chiave

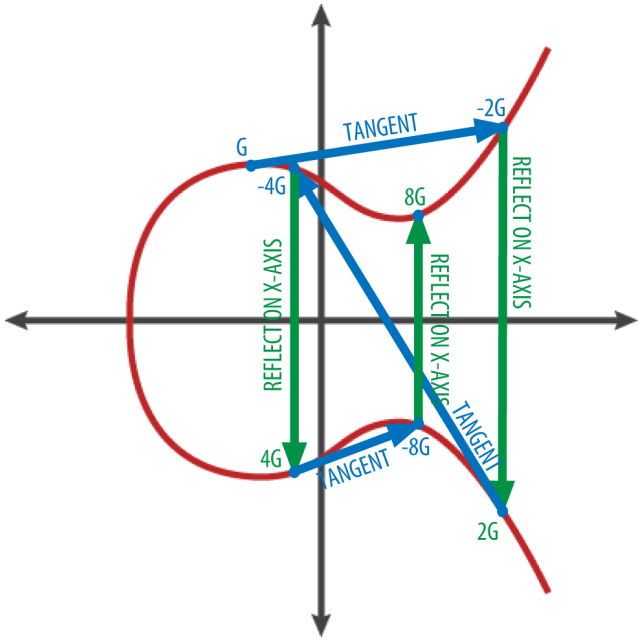


Figura 1.7: Schema di funzionamento della moltiplicazione nelle curve ellittiche, $G * k$.

pubblica K. La relazione tra k e K è fissa, ma può essere calcolata in una sola direzione.

$$k \Rightarrow K \quad k \nLeftarrow K$$

Per questo motivo la chiave pubblica, K, di un utente può essere condivisa con chiunque senza rivelare alcuna informazione circa la relativa chiave privata, k. Moltiplicare G per un qualsiasi numero k equivale ad aggiungere G a se stesso k volte di seguito. Nelle curve ellittiche, aggiungere un punto a se stesso equivale a tracciare una linea tangente al punto e trovare quindi dove questa interseca nuovamente la curva, infine riflettiamo questo punto sull'asse delle x. La dimostrazione grafica di queste operazioni eseguite in modo consecutivo lo troviamo in Figura 1.7.

Indirizzi Bitcoin

Un indirizzo bitcoin è una stringa di cifre e caratteri che può essere condivisa con chiunque desideri inviarti denaro. Gli indirizzi prodotti dalle chiavi pubbliche sono costituiti da una stringa di numeri e lettere che inizia con 1. Ecco un esempio di indirizzo bitcoin:

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

Questo indirizzo è quello che appare comunemente in una transazione come *destinatario* dei fondi. Un indirizzo bitcoin può rappresentare un proprietario di una coppia di chiavi (Pk, Sk), oppure qualcos'altro come uno script di pagamento. L'indirizzo bitcoin è derivato dalla chiave pubblica attraverso l'uso di una funzione one-way[7], in questo caso un hashing crittografico. Tali funzioni sono usate ampiamente in bitcoin:

[7]Una funzione one-way è una funzione matematica unidirezionale cioè "facile da calcolare", ma "difficile da invertire"

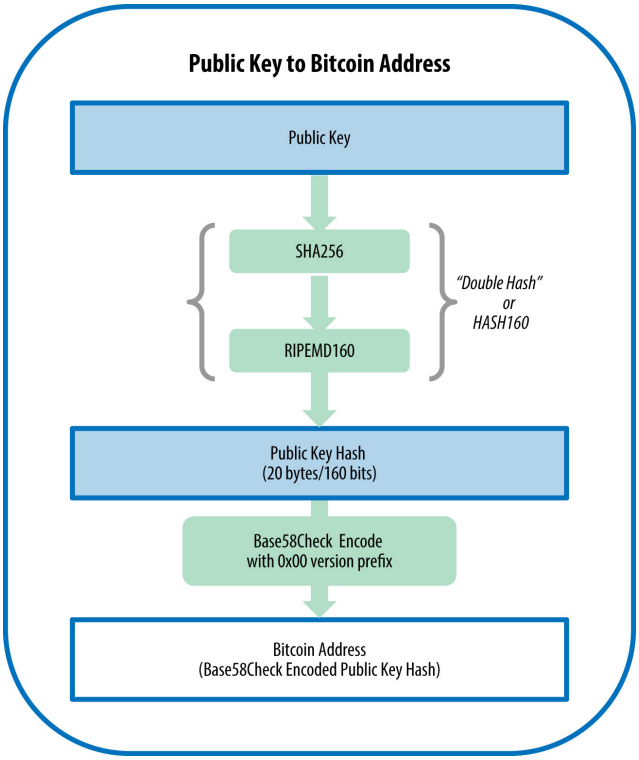


Figura 1.8: Conversione di una chiave pubblica in un indirizzo bitcoin

negli indirizzi bitcoin, negli indirizzi di script e negli algoritmi di proof-of-work. Gli algoritmi utilizzato sono in particolare:

SHA (Secure Hash Algorithm) in particolare SHA256.

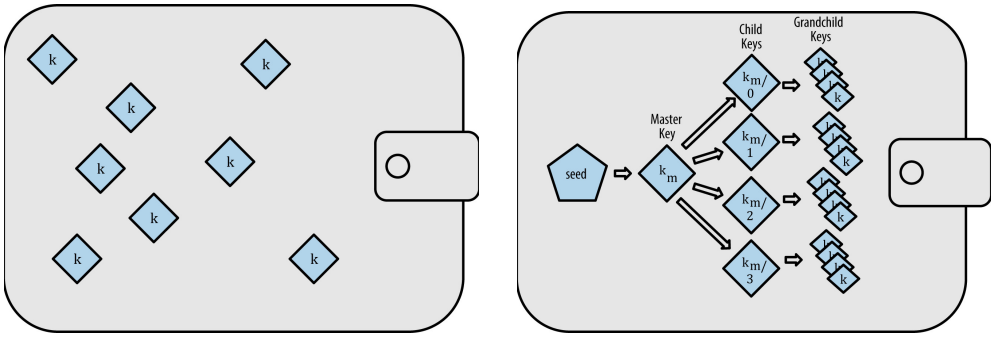
RIPEMD (RACE Integrity Primitives Evaluation Message Digest) in particolare RIPEMD160.

Iniziando dalla chiava pubblica K, ne calcoliamo l'hash *SHA256* e successivamente l'hash *RIPEMD160*, producendo un numero a 160 bit (20 byte) chiamato A. La rappresentazione grafica di queste fasi la si può vedere in Figura 1.8.

NB: L'indirizzo bitcoin non è la chiave pubblica. Gli indirizzi bitcoin sono derivati da una chiave pubblica utilizzando una funzione one-way come abbiamo appena visto. Gli indirizzi bitcoin sono quasi sempre presentati agli utenti in una codifica chiamata *Base58Check* che utilizza 58 caratteri ed un checksum per facilitarne la lettura evitando ambiguità proteggendo l'utilizzatore da errori di trascrizione.

I Wallet

I wallet sono dei contenitori di chiavi private, non monete, solitamente implementati come file strutturati o semplici database. Gli utenti firmano le transazioni con queste chiavi, dimostrando di possedere gli output delle transazioni precedenti, utilizzati come input per le nuove transazioni. La valuta (bitcoin) è memorizzata sulla blockchain sotto forma di transazioni di output.



(a) *Wallet non deterministico* **(b)** *Wallet deterministico*

Figura 1.9: Due tipologie di Wallet

Wallet non deterministici

Nei primi client bitcoin, i wallet erano semplicemente delle raccolte di chiavi private generate casualmente. Questo tipo di portafoglio è chiamato *Type-0 non-deterministic wallet*, sono soprannominati *"Just a Bunch Of Keys"* o JBOK, sono stati sostituiti dai wallet deterministici poiché scomodi da gestire. La scomodità di un wallet non deterministico sta nel fatto che si è obbligati ad eseguire molto spesso dei backup di tutte queste, dato che il loro smarrimento implica la perdita della valuta in bitcoin a cui sono legate. Dato che il riutilizzo degli indirizzi riduce la privacy, si è portati ad una continua generazione di nuove chiavi quindi, per sicurezza, si dovrà rieseguire il backup. Nonostante Bitcoin Core includa un portafoglio di tipo zero, il suo utilizzo è sconsigliato dato che c'è la possibilità di utilizzare un wallet di tipo deterministico. Vedi Figura 1.9a.

Wallet deterministici

I wallet deterministici sono stati sviluppati per consentire di derivare chiavi multiple da un singolo seme *seed*. La forma più avanzata di questo tipo di wallet è il *hierarchical deterministic wallet* o *HD wallet* definito dallo standard `BIP0032`, come dice il nome, contengono chiavi derivate in una struttura ad albero tale per cui da ogni chiave madre si possa derivare una sequenza di chiavi figlie e per ognuna di esse si possa derivare una sequenza di chiavi nipote ecc. . . ad una profondità infinita. Tale struttura la possiamo vedere in Figura 1.9b. Gli *HD wallet* offrono due importanti vantaggi rispetto alle chiavi casuali.

- La struttura ad albero può essere utilizzata per rappresentare una struttura organizzativa aggiuntiva, per esempio alcuni rami potrebbero essere utilizzati per generare chiavi impiegate poi in diversi settori aziendali.

- Gli utenti possono creare sequenze di chiavi pubbliche K_i senza avere accesso alle corrispettive chiavi private. Questo permette di utilizzare gli HD wallet anche su di un server non sicuro che riuscirà ad emettere una diversa chiave pubblica per ogni transazione.

Dim.	Pattern	Frequenza	Tempo medio
1	1K	1 su 58 chiavi	$<$ 1 millisecondi
2	1Ki	1 su 3,364	50 millisecondi
3	1Kid	1 su 195,000	$<$ 2 secondi
4	1Kids	1 su 11 milioni	1 minuto
5	1KidsC	1 su 656 milioni	1 ora
6	1KidsCh	1 su 38 miliardi	2 giorni
7	1KidsCha	1 su 2.2 mila miliardi	3–4 mesi
8	1KidsChar	1 su 128 mila miliardi	13–18 anni
9	1KidsChari	1 su 7 milioni di miliardi	800 anni
10	1KidsCharit	1 su 400 milioni di miliardi	46,000 anni
11	1KidsCharity	1 su 23 mila milioni di miliardi	2.5 milioni di anni

Tabella 1.1: Frequenza del vanity pattern *1KidsCharity* e relativo tempo di ricerca su di un medio PC fisso.

Vanity Address

I vanity address sono indirizzi bitcoin validi che contengono dei messaggi mnemonici con un significato per l'uomo. Per esempio `1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33` è uno di questi in cui notiamo la presenza della parola `Love` all'inizio dell'indirizzo. Il processo consiste essenzialmente nel selezionare una chiave privata a caso, derivando la chiave pubblica, derivando l'indirizzo bitcoin e verificando se corrisponde al modello di vanità desiderato, ripetendo miliardi di volte fino a che non verrà trovata una corrispondenza. Questa tipologia di indirizzi non sono più sicuri di qualsiasi altro indirizzo, dipendono dalla stessa Elliptic Curve Cryptography (ECC) e Secure Hash Algorithm (SHA) come qualsiasi altro indirizzo. Generare un vanity address è un esercizio di forza bruta, come riportato in Tabella 1.1, notiamo che più grande è il pattern che desideriamo sia presente all'inizio dell'indirizzo, più difficile sarà trovarlo quindi necessiterà di più tempo computazionale.

Questi indirizzi possono essere utilizzati per migliorare o addirittura sconfiggere la sicurezza, sono infatti un'arma a doppio taglio, vediamo perché:

- **Migliora la sicurezza:** un indirizzo distintivo rende più difficile per gli avversari sostituire il proprio indirizzo e ingannare i clienti reindirizzando i fondi.

- **Ingannevoli:** sfortunatamente è possibile a chiunque creare un vanity address fraudolento che assomigli a qualsiasi altro indirizzo casuale, riuscendo ad ingannare i clienti altrui.

L'idea è far generare un vanity address da una vanity pool riuscendo quindi a cercare un indirizzo con un pattern abbastanza lungo, per esempio 8 caratteri. Un attaccante che intende ingannane gli utilizzatori di tale indirizzo tenteranno di aggiungere ulteriori caratteri al pattern, facendoli combaciare con quelli dell'indirizzo originale. Questa operazione implica un costo aggiuntivo per l'attaccante, se il bottino ottenibile dalla frode non fosse abbastanza alto da coprire il costo della generazione dei tale indirizzo fraudolento di vanità.

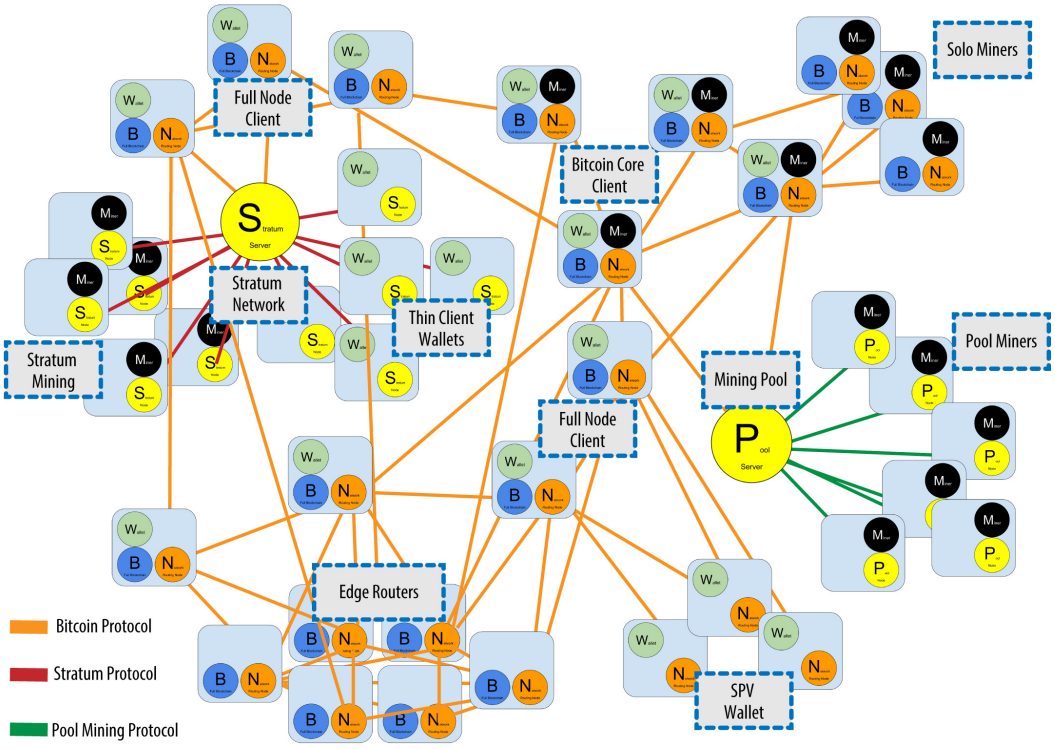


Figura 1.10: Rete estesa di bitcoin con vari tipologie di nodi, gateway e protocolli.

1.3.2 La rete bitcoin

Bitcoin è strutturato come un'architettura peer-to-peer, ciò sta a significare che i computer, nodi, che partecipano alla rete sono tutti alla pari, non ci sono nodi speciali e tutti loro condividono il compito di fornire servizio alla rete. Non esiste infatti un server o servizio centralizzato o gerarchia di qualunque tipo.

Tipologie di Nodi e Ruoli

Nonostante tutti i nodi della rete P2P bitcoin siano uguali, possono avere differenti ruoli a seconda della funzionalità che stanno supportando. Un nodo bitcoin è una serie di funzioni: routing, database della blockchain, mining, e servizi wallet, un nodo con tutte queste funzioni è appunto chiamato *full-node*. Un esempio di network eterogeneo lo si può vedere in figura 1.10.

1.4 Blockchain

La struttura dati blockchain risulta un'ordinata lista di blocchi contenenti transazioni back-linked[8]. La blockchain è spesso visualizzata come una pila verticale, con blocchi stratificati l'uno sopra l'altro e il primo blocco serve da fondamenta della pila. La visualizzazione dei blocchi impilati uno sopra l'altro provoca l'uso di terminologie come "altezza" (height) per riferirsi alla distanza dal primo blocco, e *top* per riferirsi

[8]blocchi sono collegati a ritroso, ognuno si riferisce al blocco precedente presente nella catena

Dim.	Campo	Descrizione
4 byte	Versione	Un numero di versione per tracciare upgrade al software e/o al protocollo
32 byte	Hash del Blocco Precedente	Un riferimento all'hash del blocco precedente
32 byte	Merkle Root	Un'hash della radice del merkle tree delle transazioni di questo blocco
4 byte	Timestamp	Il tempo approssimato della creazione del blocco corrente (Unix Epoch)
4 byte	Target di Difficoltà	Target di difficoltà dell'algoritmo di proof-of-work per questo blocco
4 byte	Nonce	Un contatore utilizzato per l'algoritmo di proof-of-work

Tabella 1.2: La struttura di un block header

al blocco aggiunto più recentemente. Anche se un blocco ha solo un genitore, può temporaneamente avere multipli figli. Ognuno dei figli si riferisce allo stesso blocco del padre e contiene lo stesso hash (padre) nel campo *previous block hash*. Multipli figli emergono durante un fork della blockchain, una situazione temporanea che accade quando differenti blocchi sono scoperti quasi simultaneamente da miner differenti. In questo caso solo un blocco figlio diventa parte della blockchain, il fork viene quindi risolto. Il campo *previous block hash* risulta molto importante al fine di rendere le transazioni passate sempre più affidabili, tale campo infatti è incluso nell'header del blocco e per questo motivo influenza l'hash del blocco attuale. La modifica di una transazione all'interno di un blocco genitore porta alla modifica del proprio hash che di conseguenza porta alla modifica dell'hash figlio, questo fino al vertice della blockchain. Visto che per questo ricalcolo servirebbe un'enorme potenza computazionale, l'esistenza di una lunga catena di blocchi fa si che la storia più profonda della blockchain sia immutabile, che è uno degli elementi chiave della sicurezza di bitcoin. Nella Tabella 1.2 troviamo indicato anche *Merkle Root*, conosciuto anche come un *binary hash tree*, è una struttura dati usata per indicizzare efficientemente e verificare l'integrità di un grande gruppo di dati. Sono usati in bitcoin per indicizzare tutte le transazioni in un blocco, producendo in sostanza un'impronta digitale dell'intero set di transazioni, provvedendo ad un efficiente processo di verifica se una transazione è inclusa nel blocco.

Target di difficoltà

Il target determina la difficoltà e quindi influenza il tempo necessario per trovare una soluzione da parte dell'algoritmo *proof-of-work*. Come sappiamo i bitcoin vengono generati ogni 10 minuti circa, questo è il battito di emissione della valuta che deve rimanere costante negli anni. Col passare degli anni si avrà un rapido aumento della potenza di calcolo degli elaboratori e inoltre sempre più utenti prenderanno parte alla network bitcoin. Questo comporta che sarà sempre più facile arrivare ad una soluzione proof-of-work, perciò si deve procedere con l'aumento della difficoltà di

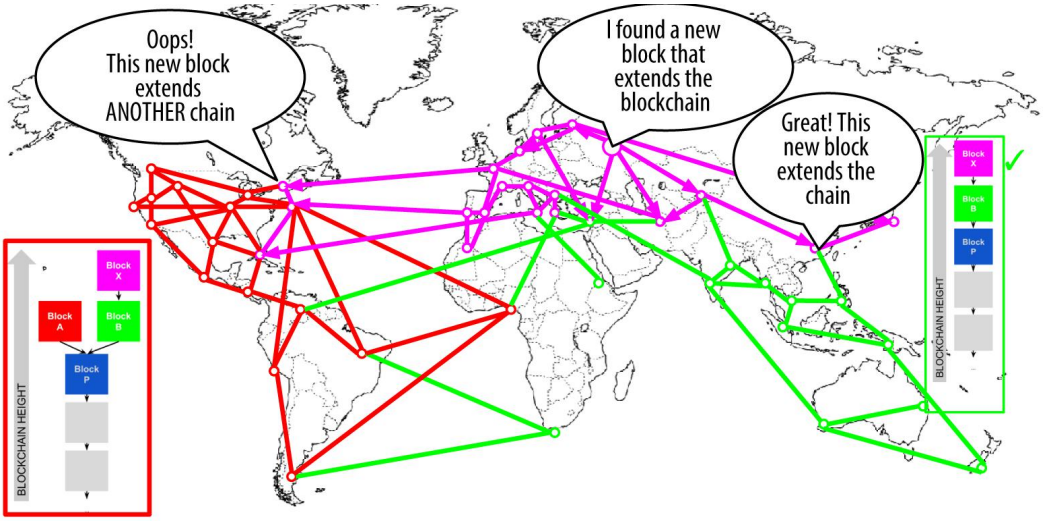


Figura 1.11: Visualizzazione di un'evento di fork di blockchain: il network ri-converge su di una nuova longest chain

tale algoritmo al fine di mantenere la generazione di nuovi blocchi fissa a 10 minuti. Questo retargeting avviene in modo automatico su ogni nodo. Ogni 2016 blocchi, tutti i nodi ritarano la difficoltà proof-of-work. L'equazione è semplice, si misura il tempo effettivo impiegato per generare gli ultimo 2016 blocchi e lo si confronta con il tempo atteso di 20160 minuti.

New Difficulty = Old Difficulty * (Tempo degli ultimi 2016 Blocchi / 20160 min)

1.4.1 Fork della blockchain

La blockchain è una struttura dati decentralizzata e le diverse copie nella network non sono sempre identiche. Come sappiamo la procedura di mining è una competizione globale nel tentativo di arrivare prima ad una soluzione proof-of-work. La velocità con cui l'aggiornamento della blockchain si propaga nella network può far si che in caso due miner trovino due proof-of-work diverse, ma comunque valide, generino un fork all'interno della rete. Questo significa che il penultimo blocco aggiunto avrà generato due o più figli. Da questo momento la network bitcoin si divide in sezioni, ognuna lavorerà per cercare di aggiungere un blocco al di sopra del blocco figlio che arrivò per primo. Con molta probabilità un solo altro miner troverà una soluzione che accrescerà una delle ramificazioni. Da questo momento in poi la blockchain valida per tutti i miner sarà quella con proof-of-work più alta, la catena più lunga. L'effetto di questa scelta andrà ad incidere sul lavoro che altri miner hanno svolto sull'altro ramo della catena. Le transazioni incluse nei blocchi di quel ramo verranno scartate, tranne quelle non incluse nei blocchi dell'attuale catena principale, che verranno rimessi in coda. Un classico fork può capitare ogni settimana ma normalmente vengono risolte dopo il calcolo di un blocco, raramente due miner trovano altre due soluzioni quasi contemporaneamente. La visione grafica degli eventi appena descritti si possono vedere in Figura 1.11.

1.4.2 Attacchi al consenso

Il meccanismo di consenso in bitcoin dipende dal fatto che la maggioranza dei miner agisca onestamente, tuttavia se un miner o un gruppo di miner raggiunge una quota significativa del potere di calcolo, essi possono attaccare il meccanismo di consenso in modo da interrompere la sicurezza e la disponibilità della rete bitcoin. Gli attacchi di consenso possono influenzare il futuro e il passato più recente (all'incirca 10 blocchi), andando a forzare eventuali fork secondari della blockchain. Un fork può essere raggiunto da qualsiasi profondità ma nella pratica la potenza di calcolo necessaria per forzare un fork molto profondo risulta essere immensa. Questo rende i blocchi, confermati n volte, praticamente immutabili. Un attacco di questo tipo non può spendere bitcoin senza l'utilizzo di firme, reindirizzare bitcoin o modificare in qualsiasi altro modo le transazioni. Gli attacchi di consenso possono interessare solo i blocchi più recenti e causare denial-of-service sulla creazione di blocchi futuri. Un attacco di questo tipo viene chiamato **Attacco del 51%**, in questo scenario un gruppo di miner, controllando il 51% della potenza di hashing della network bitcoin, hanno la possibilità di portare a termine con successo questo tipo di attacco. Possono causare deliberatamente dei fork, double-spending o eseguire attacchi di denial-of-service contro specifiche transazioni/indirizzi. Nonostante il suo nome, questo tipo di attacco in realtà non richiede il 51% della potenza di hashing. Tale soglia rappresenta semplicemente il livello al quale l'attacco è quasi sicuramente di successo. Vari gruppi di ricerca, utilizzando degli strumenti statistici, affermano che vari tipi di attacchi di consenso sono possibili utilizzando anche solo il 30% della potenza di hashing globale.

1.5 Alt-Chain e Alt-Coin

Bitcoin è il risultato di ben 20 anni di ricerca sui sistemi distribuiti e valute che ha portato ad una tecnologia rivoluzionaria: il meccanismo di consenso decentralizzato basato sulla proof-of-work. Questa invenzione è il cuore di bitcoin e ha successivamente generato un'onda di innovazione in valuta, servizi finanziari, economia, sistemi distribuiti, sistemi di votazione e contratti. Sono infatti nate delle:

Alt-coin valute digitali implementate utilizzando lo stesso modello di implementazione di bitcoin, la maggior parte delle implementazioni derivano da codice sorgente di bitcoin. Questo tipo di valuta sono chiamate anche *fork* di bitcoin. Poi ci sono anche delle alt-coin implementate da zero basate solamente sul modello blockchain ma che non riutilizzano il codice sorgente di bitcoin. Tra le principali troviamo Litecoin, Dogecoin, Freicoin, NXT.

Alt-chain blockchain alternative il cui vero scopo non è quello di realizzare un sistema monetario. Queste diverse implementazioni possono comunque includere anche una valuta ma questa viene emessa come token per allocare dell'altro, come ad esempio una risorsa o un contratto. Tra le principali alt-chain troviamo Namecoin ed Ethereum.

2
Ethereum

In questo capitolo andremo ad approfondire Ethereum, una delle alt-chain in circolazione dopo la comparsa di bitcoin. Ethereum è una piattaforma che processa ed esegue smart contract turing-completi basandosi su di un libro mastro blockchain. Non è un clone di Bitcoin ma bensì una nuova implementazione dal design completamente indipendente. Anche se, come alt-chain, non è tenuto ad esserne provvisto, Ethereum ha una propria moneta integrata, chiamata *ether*, richiesta per pagare l'esecuzione di un contratto. La blockchain Ethereum registra i vari contratti che sono espressi in un linguaggio turing-completo di basso livello. Uno smart contract è sostanzialmente un programma che gira su ogni nodo del sistema Ethereum. Tali contratti possono memorizzare dati, inviare/ricevere pagamenti in ether, memorizzare ether, ed eseguire una gamma infinita di azioni calcolabili, agendo come agenti decentralizzati di software autonomo.

2.1 Introduzione

Ethereum è spesso descritto come "the world computer". Dal punto di vista informatico Ethereum è una macchina a stati deterministica ma praticamente illimitata, costituita da uno stato singleton accessibile a livello mondiale ed una propria macchina virtuale *EVM* che applica le modifiche a tale stato. È un'infrastruttura open source, globalmente decentralizzata sviluppata per eseguire programmi denominati *smart contract*. Utilizza la blockchain per sincronizzare e memorizzare le modifiche di stato del sistema, assieme ad una cripto-valuta chiamata *ether* utilizzata per limitare le risorse per l'esecuzione dei contratti. Diversamente da Bitcoin, che ha un linguaggio di scripting molto limitato, Ethereum è stato concepito per essere una blockchain programmabile per l'uso generico che esegue una macchina virtuale in grado di eseguire codice di complessità arbitraria ed illimitata. Il linguaggio script di Bitcoin è stato intenzionalmente vincolato a delle semplici valutazioni true/false delle condizioni di spesa, il linguaggio di Ethereum invece risulta turing-completo, può infatti funzionare come un general-purpose computer.

Numero blocco	Descrizione
`Blocco #0`	Frontier - Fase iniziale di Ethereum, 30 luglio 2015 - marzo 2016.
`Blocco #200000`	Ice Age - Hard fork per introdurre un aumento di difficoltà esponenziale, per motivare la futura transazione verso PoS.
`Blocco #1150000`	Homestead - Seconda fase di Ethereum lanciata a marzo 2016.
`Blocco #1192000`	DAO - Hard fork che ha rimborsato le vittime del contratto DAO hacked causando la separazione di Ethereum ed Ethereum Classic.
`Blocco #2463000`	Tangerine Whistle - Hard fork per modificare il calcolo del gas per certa operazioni di I/O.
`Blocco #2675000`	Spurious Dragon - Hard fork per sventare vettori d'attacco DoS e meccanismi di protezione ai replay attack.
`Blocco #4370000`	Metropolis Byzantium - Terza tappa di Ethereum lanciata nell'ottobre 2017.

Tabella 2.1: Fork intermedi della blockchain Ethereum

La nascita di Ethereum

Ethereum è stato concepito in un momento in cui le persone hanno riconosciuto la potenza di un modello quale Bitcoin e stavano cercando di andare oltre alle applicazioni di cripto-valuta. L'enigma principale era: costruire su Bitcoin o avviare una nuova blockchain? Costruire al di sopra di Bitcoin significava vivere entro gli intenzionali limiti imposti da tale rete cercando soluzioni alternative. L'insieme limitato di tipologia di dati e dimensione della memorizzazione di questi sembrava limitare il numero di applicazioni che potevano essere eseguite su bitcoin, qualsiasi ulteriore aggiunta avrebbe implicato la necessità di implementare ulteriori strati fuori catena, annullando così molti dei vantaggi nell'uso di una blockchain pubblica. Per tale progetto si necessitava di maggiore libertà e fluidità rimanendo on-chain, l'unica opzione era quindi una nuova blockchain. Verso la fine del 2013, Vitalik Buterin, un giovane programmatore appassionato di Bitcoin, pensò di estendere le capacità di Bitcoin e Mastercoin. Nell'ottobre di quell'anno, propose un approccio più generalizzato al team di Mastercoin, che consentiva contratti flessibili e programmabili, anche se non turing-completi, in sostituzione dell'allora attuale linguaggio contrattuale di Mastercoin, ma questa proposta risultava una modifica troppo radicale er adattarsi alla loro roadmap di sviluppo. Nel dicembre 2013, Vitalik condivise un white paper che delineava l'idea di Ethereum: una blockchain turing-completa per tutti gli usi. Si aggiunse al team il Dr. Gavin Wood, attuale CTO, che lo aiutò nel perfezionale il protocollo che attualmente porta il nome di Ethereum. Lo sviluppo di Ethereum è stato pianificato in quattro fasi distinte, con importanti cambiamenti in ciascuna fase. Ogni fase può includere delle subreleases, note come "hard fork", che apportando funzionalità non retro compatibili. Le quattro fasi di sviluppo portano il nome di *Frontier, Homestead, Metropolis, Serenity.* I fork intermedi programmati sono *Ice Age, DAO, Tangerine Whistle, Spurious Dragon, Byzantinum e Costantinople.* L'elenco completo lo possiamo notare in Tabella 2.1. Dopo Byzantium, rimangono

altri due hard fork Metropolis, Costantinople ed a seguire la fase finale Serenity.

2.1.1 Turing-completezza di Ethereum

La capacità di Ethereum di eseguire un programma memorizzato, in un macchina a stati chiamata Ethereum Virtual Machine, mentre legge e scrive dati in memoria lo rende un sistema turing-completo. Ethereum può infatti calcolare qualsiasi algoritmo che possa essere calcolato da qualunque macchina di Turing. L'innovativa invenzione di Ethereum consiste nel combinare l'architettura di calcolo generica di un computer con l'affiancamento di una blockchain decentralizzata che memorizza il programma da eseguire. In questo modo si crea un computer mondiale distribuito (singleton). I programmi Ethereum funzionano ovunque producendo uno stato comune che viene garantito dalle regole di consenso. Il fatto che Ethereum sia turing-completo porta con se una conseguente problematica, il fatto che possa eseguire qualsiasi programma, di qualsiasi complessità. Questa flessibilità porta alcuni problemi dal punto di vista della sicurezza e di gestione delle risorse disponibili. Turing ha dimostrato come non sia possibile prevedere se un programma terminerà o meno senza eseguirlo. Ciò pone una sfida: ogni nodo partecipante (client) dovrà convalidare ogni singola transazione, convalidando qualsiasi smart contract che chiama, ma come dimostrato da Turing, Ethereum non potrà sapere a priori se uno smart contract terminerà, o per quanto tempo verrà eseguito, senza effettivamente eseguirlo. Se per caso o di proposito fosse possibile creare un contratto facendo in modo che funzioni per sempre, quando un nodo tenterà di convalidarlo, si verificherebbe un attacco di tipo DoS. Per evitare che questo accada Ethereum introduce un meccanismo di misurazione delle risorse chiamato *gas*. Quando l'EVM esegue uno smart contract tiene accuratamente conto di ogni istruzione (calcolo, accesso ai dati, ecc...). Ogni istruzione ha un proprio costo predeterminato in unità di *gas*. Quando una transazione attiva l'esecuzione di un contratto, deve includere una quantità di *gas* che imposta come limite superiore di ciò che può essere consumato eseguendo tale smart contract. Le unità di gas devono essere incluse quindi acquistate con l'ether, durante il calcolo della transazione.

2.1.2 Dalla Blockchain generica alle ĐApps

Ethereum è nato con l'intenzione di creare una blockchain general-purpose che potesse essere programmata per svariati usi. Molto rapidamente la visione di Ethereum si è espansa fino a diventare una piattaforma per programmare DApps. Queste DApps presentano una prospettiva più ampia rispetto agli smart contract. Una DApp è come minimo un contratto, munito però di un'interfaccia web utente. Più in generale, una DApp è un'applicazione Web costruita su servizi di un'infrastruttura aperta, decentralizzata e peer-to-peer. Lo si può anche trovare scritto in questo modo ĐApp. Il carattere Đrappresenta il carattere latino *ETH* che appunto allude a Ethereum.

2.1.3 Le unità monetarie di Ethereum

L'unità monetaria di Ethereum si chiama *ether* ed è identificata con il nome di *ETH*. Un ether è suddiviso in unità più piccole, fino all'unità più piccola chiamata *wei*. All'interno di Ethereum il valore di *ether* viene sempre rappresentato in *wei*. Le

Valore (in wei)	**Esp.**	**Nome**	**Nome *SI***
1	1	wei	Wei
1.000	10^3	Babbage	Kilowei o femtoether
1.000.000	10^6	Lovelace	Megawei o picoether
1,000,000,000	10^9	Shannon	Gigawei o nanoether
1.000.000.000.000	10^{12}	Szabo	Microetere o micro
1.000.000.000.000.000	10^{15}	Finney	Milliether o milli
1.000.000.000.000.000.000	10^{18}	Ether	Ether
1.000.000.000.000.000.000.000	10^{21}	Grand	Kiloether
1.000.000.000.000.000.000.000.000	10^{24}		Megaether

Tabella 2.2: Denominazioni delle frazioni di ether.

denominazioni delle varie frazioni di ether hanno sia un nome scientifico che utilizza il sistema internazionale di unità, SI, sia un nome colloquiale che rende omaggio a molte delle più grandi menti della matematica e della crittografia, tutte le denominazioni sono riportate in Tabella 2.2.

Wallet

Il termine wallet rappresenta tutti quei software che aiutano l'utente nella gestione del proprio account Ethereum, memorizza infatti le chiavi d'accesso ai fondi, crea e trasmette transazioni. Esistono varie tipologie di wallet, che richiedono più o meno attenzione per la gestione delle chiavi, più controllo e indipendenza si vuole avere, più aumentano le responsabilità. Un buon wallet è per esempio MetaMask, portafoglio web-based che si presenta come un'estensione per i browser più comuni quali Chrome, Firefox e Opera. Risulta facile da usare e conveniente per eseguire test, poiché risulta in grado di connettersi ad una vasta gamma di nodi Ethereum per testare la blockchain. L'installazione di MetaMask risulta veramente semplice, infatti basterà aprire il gestore estensioni del proprio browser e cercare tale estensione e una volta individuata procedere con l'istallazione. Al termine della procedura si aprirà automaticamente la schermata principale di MetaMask dove ci viene richiesto di inserire una password. Tale password servirà per impedire, a chiunque utilizzi il computer in questione e quindi ha accesso al browser su cui è installato MetaMask, di accedere a tale wallet e quindi avere accesso ai fondi. A questo punto viene generato un backup mnemonico formato da 12 parole inglesi. Queste parole potranno essere utilizzate in qualsiasi wallet compatibile per recuperare l'accesso al proprio conto in caso di guasti di qualsiasi tipo. Si consiglia di conservare un backup cartaceo delle credenziali d'accesso e del backup mnemonico riponendolo in un luogo fisicamente sicuro. Si sconsiglia infatti di salvare digitalmente tali informazioni. A questo punto ci troveremo davanti ad una schermata dove ci viene mostrato il nostro conto. Troviamo varie informazioni tipo il nostro indirizzo casualmente generato, il saldo del conto, lo storico delle transazioni passate e in corso. In questa schermata sarà anche possibile selezionare la rete sulla quale inviare le nostre transazioni. Le varie reti disponibili sono:

Rete principale di Ethereum principale blockchain pubblica di Ethereum, in questa rete il valore in ether è reale e ogni azione avrà conseguenze reali.

Rete di test Ropsten rete con blockchain pubblica in cui ether non ha valore.

Rete di test Kovan rete con blockchain pubblica in cui ether non ha valore. Utilizza il protocollo di consenso Aura. Questa rete è supportata solo da Parity, gli altri client il protocollo di consenso Clique, proposto in seguito.

Rete di test Rinkeby rete con blockchain pubblica, utilizza il protocollo di consenso Clique. Anche in questa rete Ether non ha valore.

Localhost 8545 Si connette ad un nodo in esecuzione sullo stesso computer del browser. Tale noto può far parte di una blockchain pubblica o di una rete di test privata.

RPC personalizzata consente di collegare MetaMask a qualsiasi nodo con un'interfaccia *RPC* (Remote Procedure Call) compatibile con Geth. Tale nodo può essere parte di una blockchain pubblica o privata.

Prenderemo come esempio la *rete di test Ropsten*. Per muovere i primi passi su Ethereum, o qualsiasi altra blockchain, abbiamo bisogno di fondi. Per eseguire dei test possiamo richiedere dei fondi gratuitamente sulle reti di test, per esempio Ropsten. Quindi da MetaMask selezioniamo la sopracitata rete di test selezioniamo *Deposita > Ottieni Ether*, a questo punto si aprirà un'altra scheda dove poter richiedere *1 ether from faucet*. Una volta eseguita la richiesta, in basso, nella sezione *transactions* comparirà l'ID della transazione appena eseguita per trasferire 1 ether sul nostro conto. Attendendo qualche secondo la somma richiesta vi verrà accredita e possiamo effettuare dei pagamenti. Per fare ciò dobbiamo tenere conto del *gas*, commissione raccolta dai minatori per convalidare la transazione. Il costo massimo corrisponde a :

$$3 * 21000 \text{ gwei} \quad = \quad 63000 \text{ gwei} \quad = \quad 0.000063 \text{ ETH}$$

Questo sta a significare che una transazione di 1 ETH costerà 1.000063 ETH, nonostante MetaMask visualizzi cifre approssimate, omettendo di visualizzare le cifre meno significative, sia nel saldo del conto che nel totale delle singole transazioni.

Durante la presentazione di Ethereum abbiamo anticipato che una delle particolarità di questa nuova blockchain sta proprio nella possibilità implementazione di smart contract da sottomettere al computer mondiale Ethereum. Vediamo ora come creare il primo contratto che ci darà la possibilità di depositare ether sul conto del contratto creato e successivamente procedere con il prelievo di massimo 0.1 ether. Il contratto che andremo a creare si chiama *faucet* e verrà scritto in Solidity tramite Remix[1]. Il funzionamento di questo contratto è semplice: distribuisce ether a qualsiasi indirizzo che ne richiede e il conto di tale contratto può essere caricato periodicamente. Per procedere quindi dobbiamo copiare il codice qui riportato all'interno di un nuovo file *faucet.sol* all'interno di Remix. A questo punto l'IDE compilerà in automatico mostrando un riquadro verde riportante il nome del contratto. Ora siamo pronti per inviare questo smart contract sulla blockchain pubblica e successivamente testarlo.

[1]Remix è un IDE per la scrittura di Smart Contract in linguaggio Solidity, tramite browser. http://remix.ethereum.org

```
1  // Our first contract is a faucet!
2  pragma solidity ^0.4.19;
3
4  contract Faucet {
5     // Give out ether to anyone who asks
6     function withdraw(uint withdraw_amount) public {
7        // Limit withdrawal amount
8        require(withdraw_amount <= 100000000000000000);
9        // Send the amount to the address that requested it
10       msg.sender.transfer(withdraw_amount);
11    }
12    // Accept any incoming amount
13    function () public payable {}
14 }
```

La registrazione di un contratto sulla blockchain comporta la creazione di una transazione speciale la cui destinazione è l'indirizzo 0x000...000 noto come indirizzo zero, facendo capire alla blockchain Ethereum che siamo intenzionati a registrare un nuovo smart contract. Fortunatamente Remix riesce ad interagire con MetaMask, semplicemente impostando la voce in *Run* > *Environment* sul valore *Injected Web3 - Ropsten*. Subito al di sotto di questa voce troviamo la voce *Deploy* che procederà con la creazione di una nuova transazione che portando alla registrazione del contratto. Una volta confermata tale transazione possiamo richiamare la funzione `withdraw(uint withdraw_amount)` inserendo massimo 0.1 ETH, limite impostato alla riga 8, nell'apposito campo della scheda Run, vista in precedenza e procedere premendo il pulsante *transact*. A questo punto verrà creata una nuova transazione verso l'indirizzo del contratto richiedendo tale somma di ether. Se tutto andrà a buon fine, il contratto esaminerà tale richiesta e produrrà un'ulteriore transazione trasferendo la cifra indicata verso il nostro conto. Si consiglia di utilizzare https://ropsten.etherscan.io per visualizzare tutti i dettagli delle varie transazioni, una cosa importante da notare è il fatto che le transazioni di risposta di contratti non vengono riportate nel tab *Transaction*, ma compariranno nel tab *Internal Txns*.

2.2 Le transazioni

Le transazioni sono messaggi firmati originati da un account di proprietà che vengono trasmessi della rete Ethereum e registrati sulla blockchain. La struttura di base di una transazione è quella che vediamo in tabella 2.3. La struttura del messaggio transazione viene serializzata utilizzando lo schema di codifica RLP (Recursive Length Prefix), creato appositamente per semplificare la serializzazione dei dati in Ethereum. Tutti i numeri vengono codificati come interi big-endian, di lunghezze che sono multipli di 8 bit. Le etichette di campo non fanno parte dei dati serializzati della transazione, che contiene valori di campo codificati in RLP. In generale RLP non contiene alcun delimitatore o etichetta di campo e la lunghezza di ciascun campo viene identificata tramite un prefisso. Molte informazioni visualizzate da wallet e altri strumenti sono derivate da altre più primitive. Per esempio la chiave pubblica dell'EOA può essere ricavata dalle componenti v, r, s della firma ECDSA.

Campo	Descrizione
nonce	Numero di sequenza emesso dall'EOA di origine, utilizzato per impedire il replay attack.
gasPrice	Prezzo del gas in wei che il mittente è disposto a pagare.
gasLimit	Quantità massima di gas che l'EOA di origine è disposto ad acquistare per la transazione corrente.
recipient	Indirizzo Ethereum a cui è destinata la transazione.
value	Quantità di ether da inviare a destinazione.
data	Lunghezza variabile del payload.
v, r, s	Tre componenti della firma digitale ECDSA dell'EOA di origine.

Tabella 2.3: Struttura base di una transazione Ethereum.

Nonce

Il nonce è una delle componenti più importanti ma meno comprese in Ethereum. La definizione che troviamo nel White Paper si legge: *"nonce: numero scalare uguale al numero di transazioni inviate da questo indirizzo, o nel caso di account con codice associato, il numero di creazioni di contratto effettuate da questo account"*. Esistono due scenari in cui l'esistenza di un nonce risulta importane.

1. Supponiamo di aver due transazioni da effettuare, una più importante dell'altra ma non disponiamo di un saldo sufficiente per pagare entrambe le transazioni. Se inviassimo, in un sistema decentralizzato come Ethereum, prima la transazione più importante e poi la seconda, non disponendo del campo nonce, non abbiamo la certezza che la prima transazione venga sicuramente eseguita prima dell'altra, dato che i nodi possono ricevere le transazioni in entrambi gli ordini.

2. Immaginiamo ora di possedere un portafoglio Ethereum con parecchi fondi, e di fare un trasferimento di pochi ether ad un altro indirizzo. Senza il campo nonce un qualsiasi altro utente che vede tale transazione la potrà replicare fintanto che sul nostro conto abbiamo fondi a sufficienza da coprire la cifra versata.

Il campo nonce ha quindi un duplice scopo, dare un ordine di esecuzione alle transazioni, sostanzialmente l'ordine di creazione, ed essendo unico, serve a sventare i replay attack.

2.2.1 Gas di transazione

Come abbiamo già detto precedentemente, il gas è il carburante di Ethereum, una valuta virtuale separata da l'ether con il proprio tasso di cambio. Ethereum utilizza il gas per controllare la quantità di risorse che una transazione può utilizzare dal momento che il modello di calcolo a terminazione aperta (turing-completo) richiede una qualche forma di misurazione al fine di evitare attacchi di denial-of-service. Il gas è separato dall'ether per proteggere il sistema dalla volatilità che potrebbe insorgere assieme ai rapidi cambiamenti nel valore dell'ether. Il cambio gas/ether può quindi variare al variare del valore della valuta ether.

Il campo `gasPrice` in una transazione consente all'ordinante di impostare il prezzo che è disposto a pagare in cambio del gas. Il prezzo è misurato in wei per unità di gas. I wallet possono regolare il prezzo del gas nelle transazioni per ottenere una conferma più rapida, maggiore sarà il prezzo del gas, più rapida sarà la conferma della transazione. Il valore minimo a cui `gasPrice` può essere impostato è zero. Queste transazioni risulteranno quindi completamente gratuite ma rischiano di non essere mai confermate.

Se l'indirizzo di destinazione è uno smart contract, la quantità di gas può essere stimata ma non può essere determinata con precisione. Questo perché un contratto può valutare diverse condizioni che portano a diversi percorsi di esecuzione, con conseguenti costi totali di gas diversi.

Destinatario della transazione

Il destinatario della transazione viene specificato nel campo `to`. Questo contiene un indirizzo Ethereum di 20 byte che può essere un EOA o un indirizzo di un qualche contratto. Ethereum non convalida ulteriormente questo campo, qualsiasi valore di 20 byte è considerato valido. È possibile inviare fondi ad un indirizzo che non ha alcuna chiave privata o contratto corrispondente, in questo modo gli ether inviati vengono persi e non saranno più spendibili. La convalida deve quindi essere eseguita a livello di interfaccia utente.

2.2.2 Creazione contratti

La creazione di un contratto implica la generazione di una transazione speciale per procedere alla distribuzione di tale contratto. Le transazioni che procedono alla registrazione di un contratto su blockchain vengono indirizzate verso uno speciale indirizzo chiamato indirizzo zero che non rappresenta né un EOA né un contratto. Viene infatti utilizzato come destinazione con un significato speciale: *"crea questo contratto"*. Tuttavia questo indirizzo a volte riceve pagamenti da vari indirizzi, con la conseguente perdita di tali ether, o per intenzionale volontà di rendere non spendibile una certa somma di ether.

Una transazione di creazione contratto deve contenere solo un payload di dati contenente il bytecode compilato che creerà il contratto. È comunque possibile includere un importo di ether nel campo del valore se si desidera creare un contratto con un saldo iniziale pari a tale versamento, ma è del tutto facoltativo. Questa operazione può essere eseguita anche dopo la creazione del contratto, attendendo la fine dell'operazione di creazione contratto andando poi a copiare l'indirizzo Ethereum di tale contratto e procedendo con la creazione di una nuova transazione che trasferirà un quantitativo di ether a questo indirizzo.

Firme digitali

L'algoritmo di firma di Ethereum è l'Elliptic Curve Digital Signature (ECDSA) che si basa su coppie di chiavi pubbliche e private calcolate appunto su curve ellittiche. Lo scopo della firma digitale è riassunto nei seguenti punti:

k	Chiave Privata	m	Transazione RLP-encoded
$F_{kaccak256}$	Funzione hash Keccak-256	F_{sig}	Algoritmo di firma

Tabella 2.4: Descrizione dei campi della funzione sign.

1. Dimostra che il proprietario della chiave privata, che implicitamente è anche il proprietario del conto Ethereum, ha autorizzato la spesa di ether o l'esecuzione di un contratto.
2. Garantisce la non-repudiation. La prova di tale approvazione non è negabile.
3. Dimostra che i dati della transazione non sono stati e non possono essere modificati da nessuno dopo che la transazione è stata firmata.

Creazione della Firma Digitale

Ethereum, per attuare l'ECDSA, dovrà firmare un messaggio, che nel nostro caso è la transazione, o più precisamente l'hash Keccak-256 dei dati con codifica RLP. La chiave di firma è la chiave privata dell'EOA ed il risultato sarà:

$$Sig = F_{sig}(F_{kacak256}(m), k)$$

La funzione F_{sig} produce così la firma Sig che risulterà composta da due valori comunemente denominati con r e s:

$$Sig = (r, s)$$

Questo algoritmo per calcolare questi due valori r e s ha generato una chiave privata temporanea in modo crittograficamente sicuro. Come sappiamo infatti la chiave privata temporanea viene utilizzata per derivare la chiave pubblica, quindi abbiamo:

- Un numero casuale q crittograficamente sicuro, utilizzato come chiave privata temporanea.
- La corrispondente chiave pubblica Q, generata da q e il punto G generatore della curva ellittica.

Il valore r della firma digitale, è la coordinata x della chiave pubblica Q. Da lì, l'algoritmo calcolerà s della firma in moto che:

$$s \equiv q^{-1}(Keccak256(m) + r * k)(\text{mod } p)$$

Qui troviamo riportati i significati dei vari simboli all'interno della formula: La

q	chiave privata temporanea	r	coordinata x della chiave pubblica
m	dati della transazione	p	ordine primo della curva ellittica

verifica di correttezza risulta essere l'inverso della funzione di firma, utilizzando i valori r, s, e la chiave pubblica del mittente si sarà in grado di calcolare un valore Q che sarà il punto sulla curva ellittica. I passi sono i seguenti:

1. Controllare che tutti gli input siano stati forniti correttamente;
2. Calcolare $w = s^{-1} \bmod p$.
3. Calcolare $u_1 = Keccak256(m) * w \bmod p$.
4. Calcolare $u_2 = r * w \bmod p$.
5. Infine, calcolare il punto sulla curva ellittica $Q \equiv u_1 * G + u_2 * K (\bmod p)$.

Se la coordinata x del punto appena calcolato Q risulta uguale a r, allora il verificatore può concludere che la firma è valida. Si noti che per tale verifica, la chiave privata non è né conosciuta né rivelata. Per produrre una transazione valida, il mittente deve firmare digitalmente il messaggio, utilizzando l'algoritmo di firma digitale a curva ellittica. Quando diciamo *"firmare la transazione"* intendiamo effettivamente *"firmare l'hash* $Keccak - 256$ *dei dati della transazione serializzati su RLP"*. La firma viene quindi applicata all'hash, non alla transazione stessa. Per firmare una transazione in Ethereum, il mittente deve:

1. Creare una struttura dati di transazione, contenente nove campi: *nonce, gasPrice, gasLimit, to, value, data, chainID, 0, 0.*
2. Produrre un messaggio serializzato con codifica RLP della struttura dei dati della transazione.
3. Calcolare l'hash $Keccak - 256$ di questo messaggio serializzato.
4. Calcolare la firma ECDSA, firmando l'hash con la chiave privata dell'EOA di origine.
5. Aggiungere i valori calcolati v, r e s della firma ECDSA alla transazione.

L'identificatore di ripristino v viene utilizzato per indicare la parità della componente y della chiave pubblica. Le tre fasi creazione, firma e trasmissione di una transazione, normalmente vengono eseguite come una singola operazione. Tuttavia è possibile creare e firmare una transazione in due passaggi separati. Una volta che una transazione viene firmata è possibile inviarla sulla rete Ethereum in un secondo momento. La ragione più comune per eseguire ciò è la sicurezza. Infatti il computer che firma la transazione deve conoscere le chiavi private che per forza di cose si devono trovare in memoria. Il computer che esegue la trasmissione deve essere collegato ad Internet. Se queste due funzioni sono sulla stessa macchina, allora le chiavi private sono salvate su di un sistema online, il che è piuttosto pericoloso. Questa procedura di separazione viene chiamata firma offline ed è una pratica di sicurezza comune, uno schema di questa procedura è riportato in Figura 2.1. Il sistema con la sicurezza maggiore risulta il sistema di tipo air-gapped, in tale sistema non vi è alcuna connettività di rete tra il computer online, colui che trasmetterà, e il computer che firma tale messaggio. Per firmare le transazioni e trasferirle da e verso la macchina online saranno necessari dei dispositivi di supporto dati o una webcam con relativo QRCode. Questa tecnica, per quanto sicura non risulta scalabile, dal momento in cui ogni transazione va trasferita manualmente tra le macchine.

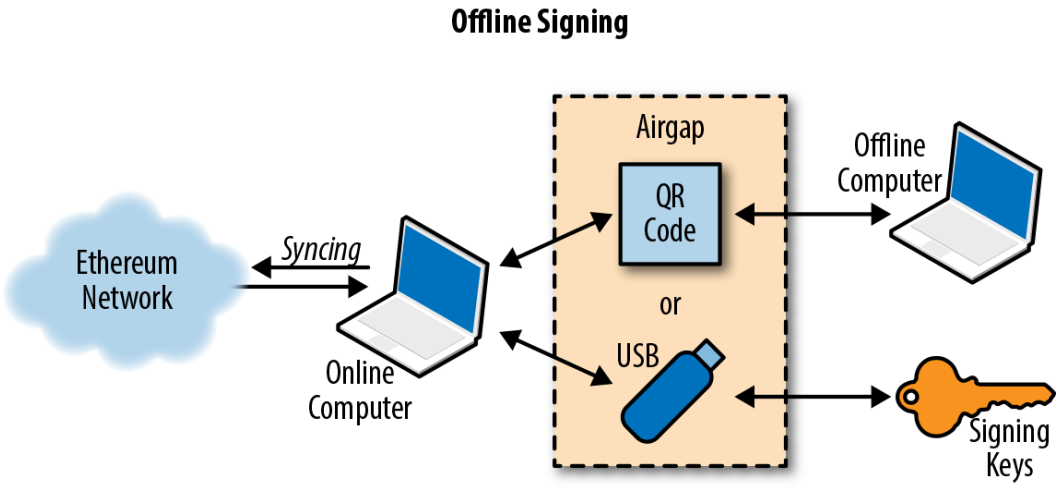


Figura 2.1: Firma offline delle transazioni di Ethereum

2.3 Smart Contract

Il termine smart contract è stato usato nel corso degli anni per descrivere un'ampia varietà di cose differenti. Nel contesto di Ethereum il termine in realtà è un po' improprio dato che gli smart contract Ethereum non sono né smart (intelligenti), né contract (legali). Tali contratti fanno piuttosto riferimento a programmi informatici immutabili che funzionano in modo deterministico in un contesto quale la macchina virtuale Ethereum (EVM) sul computer decentralizzato Ethereum. Ma vediamo questa affermazione per step:

Programmi per computer Gli smart contract sono semplicemente dei programmi per computer. La parola "contratto" non ha alcun significato legale per questo contesto.

Immutabile Una volta implementato, il codice di un contratto non può essere cambiato, a differenza di un software tradizionale. Infatti l'unico modo per portare delle modifiche ad un proprio contatto è quello di ridistribuirne una nuova istanza.

Deterministico L'esito di uno smart contract sarà lo stesso per tutti coloro che lo eseguiranno, dato il contesto della transazione e lo stato della blockchain Ethereum al momento di tale esecuzione.

Contesto EVM Gli smart contract operano con un contesto di esecuzione limitato. Possono accedere al loro stato, al contesto della transazione che li ha richiamati e ad alcune informazioni sui blocchi più recenti.

Computer mondiale decentralizzato L'EVM viene eseguita come istanza locale su ogni nodo Ethereum, ma poiché tutte le istanze di EVM operano sullo stesso stato iniziale e produrranno lo stesso stato finale, il sistema nel suo complesso opera come un singolo computer mondiale.

2.3.1 Ciclo di vita di uno Smart Contract

Gli smart contract sono solitamente scritti in un linguaggio di alto livello, come ad esempio Solidity. Per essere eseguiti nella EVM devono poi essere compilati in bytecode. Una volta compilati possono essere distribuiti sulla piattaforma Ethereum tramite la speciale transazione di creazione contratto che invia tale transazione all'indirizzo 0x0. Ogni contratto è identificato da un proprio indirizzo Ethereum che può essere utilizzato per l'invio di fondi o per richiamare le funzioni in esso specificate. Diversamente dagli EOA, uno smart contract non ha associate delle chiavi. Tali contratti vengono eseguiti solamente se richiamati da una transazione. Un contratto può richiamare un altro contratto ma il primo della catena sarà per forza di cose richiamato da un qualche EOA. Nel caso in cui l'esecuzione dovesse fallire per un qualche motivo, termine gas o errore di qualche tipo, tutti i cambiamenti di stato avvenuti fino a quel momento vengono ripristinati a prima dell'esecuzione come se la transazione non fosse mai stata eseguita. Una transazione fallita viene comunque registrata come tentata, e l'etere speso a gas per l'esecuzione viene detratto dall'account di origine.

Come accennato in precedenza, è importante ricordare che il codice di uno smart contract non può essere modificato. Tuttavia, un contratto può essere cancellato, rimuovendo il codice e il suo stato interno, archiviazione, dal suo indirizzo, lasciando un account vuoto. Qualsiasi transazione inviata a quell'indirizzo dell'account dopo che il contratto è stato eliminato non comporta l'esecuzione di alcun codice, perché non esiste più alcun codice da eseguire. Per fare ciò si dovrà eseguire un apposito codice operativo chiamato SELFDESTRUCT. Tale operazione costa una cifra in gas negativa, una sorta di rimborso del gas, questo per incentivare il rilascio di risorse. L'eliminazione di un contratto in questo modo non rimuove la cronologia delle transazioni del contratto, poiché la blockchain stessa è immutabile. La funzionalità SELFDESTRUCT sarà disponibile solamente se l'autore del contratto ha programmato il contratto per ottenere tale funzionalità, in caso contrario il contratto non potrà essere cancellato.

Linguaggi di alto livello per Ethereum

L'EVM è una macchina virtuale che esegue una speciale forma di codice chiamata bytecode EVM, analoga alla CPU di un comune computer, che esegue codice macchina come x86_64. È possibile programmare i contratti Ethereum direttamente in bytecode, tuttavia la maggior parte dei programmatori preferisce ovviamente utilizzare un linguaggio di alto livello per poi convertire il codice prodotto in bytecode tramite un compilatore ad hoc. I linguaggi di programmazione possono essere classificati in due ampi paradigmi di programmazione: *dichiarativo* e *imperativo*, anche noti rispettivamente come *funzionali* e *procedurali*. Nella procedura dichiarativa, scriviamo funzioni che esprimono la logica di un programma, ma non il suo flusso. Viene utilizzata per creare programmi in cui non ci sono effetti collaterali, non ci saranno, infatti, cambiamenti di stato al di fuori di una funzione. Haskel e SQL sono due di questi. Nella programmazione imperativa, al contrario, una procedura può cambiare sia la logica che il flusso di un programma. In questo contesto troviamo C++ e Java. Poi ci sono dei linguaggi di programmazione ibridi che incoraggiano la programmazione dichiarativa ma possono essere utilizzate anche per esprimere

un paradigma di programmazione imperativo. Qui troviamo Lisp, JavaScript e Python. Qualsiasi linguaggio imperativo può essere usato per scrivere un paradigma dichiarativo. I linguaggi puramente dichiarativi, invece, non possono essere usati per scrivere un paradigma imperativo.

Mentre la programmazione imperativa è più comunemente utilizzata dai programmatori e la capacità di qualsiasi parte del programma di cambiare lo stato di qualsiasi altro rende difficile ragionare sull'esecuzione di un programma e introduce molte opportunità per i bug. Nei contratti intelligenti, i bug costano letteralmente denaro e di conseguenza, è di fondamentale importanza scrivere contratti intelligenti senza effetti indesiderati. Per questo motivo, i linguaggi dichiarativi svolgono un ruolo molto più importante negli smart contract. Tuttavia, il linguaggio più utilizzato per i contratti intelligenti risulta Solidity che è imperativo.

2.4 Solidity

Solidity è stato creato dal Dr. Gavin Wood come linguaggio esplicito per scrivere smart contract per Ethereum. Il principale prodotto del progetto Solidity è il compilatore, *solc*, che converte i programmi scritti in Solidity in linguaggio bytecode EVM. Ogni versione del compilatore corrisponde e compila una versione specifica di Solidity. Tale linguaggio segue un modello di versioning chiamato semantic versioning, che specifica la versione strutturandola in tre parti numeriche: MAJOR:MINOR:PATCH.

È possibile installare il compilatore Solidity su Ubuntu/Debian seguendo le istruzioni riportate sull'apposita pagina http://solidity.readthedocs.io. Come già detto in precedenza ne esiste anche una versione web-based che include un IDE chiamato Remix, pensato per test iniziali e lo sviluppo di piccoli smart contract.

2.4.1 Variabili e funzioni locali predefinite

Quando un contratto viene eseguito nell'EVM, ha accesso ad un piccolo insieme di oggetti globali, tra cui `block`, `msg`, e `tx`.

- `msg` contiene le informazioni del chiamante, che può essere un EOA nel caso di una transazione, oppure una message call, nel caso la chiamata fosse originata da un altro contratto.
- `tx` fornisce un mezzo per accedere alle info relative alla transazione.
- `block` contiene informazioni circa il blocco corrente.
- Qualsiasi indirizzo passato ha un numero di attributi e metodi richiamabili con il prefisso `address`.
- `addmod, mulmod` per eseguire somme e moltiplicazioni in module.
- `keccak256, sha256, sha3, ripemd160` per calcolare l'hash del messaggio.
- `ecrecover` recupera l'indirizzo utilizzato per firmare il messaggio.
- `selfdestrunct` elimina il contratto inviando l'ether residuo.
- `this` si riferisce all'indirizzo dell'account attualmente in esecuzione.

2.4.2 Funzioni di un contratto

All'interno di un contratto possiamo definire delle funzioni richiamabili da una qualsiasi transazione EOA oppure da un altro contratto. La sintassi utilizzata per la scrittura di queste funzioni è la seguente:

```
function FunctionName([parameters]) {public|private|internal|external}
  [pure|constant|view|payable] [modifiers] [returns (return types)]
```

`FuncionName`	identifica la funzione tramite un nome, utilizzato per richiamare tale funzione tramite transazioni EOA o da altri contratti.
`parameters`	specificano il tipo degli argomenti che devono essere passati alla funzione e il rispettivo nome che assumono all'interno di essa.
`public`	tali funzioni possono essere richiamate da altri contratti o transazioni oppure dall'interno del contratto che la implementa.
`external`	come le funzioni pubbliche ma non possono essere richiamate da procedure interne al contratto.
`internal`	sono accessibili solamente tramite procedure interne al contratto.
`private`	come le interne ma non sono richiamabili da contratti derivati.
`costant/view`	la funzione contrassegnata come `view` promette di non modificare lo stato, `constant` è un alias di `view` che sarà deprecato in futuro, il compilatore produrrà un warning in caso di utilizzo errato.
`pure`	tale funzione non legge ne scrive le variabili in memoria, opera solamente su argomenti e restituisce dati senza alcun riferimento a dai memorizzati.
`payable`	accetta pagamenti in entrata, se non implementata rifiutano tali transazioni.

Considerazioni sul Gas

Il gas è una risorsa limitata pensata per limitare la quantità massima di calcolo che Ethereum consentirà ad una transazione di consumare. Se il limite di gas viene in qualche modo superato viene lanciata l'eccezione `out of gas exception`, viene ripristinato lo stato dello smart contract a prima di tale esecuzione e tutto il gas speso fino a quel momento per tentare tale esecuzione viene speso come tassa di transazione e quindi non viene rimborsato. Il costo del gas può essere stimato tramite la seguente procedura:

```
var contract = web3.eth.contract(abi).at(address);
var gasEstimate = contract.myMethod.estimateGas(arg1,arg2,{from: account});
```

`gasEstimate` ci indicherà il numero di unità di gas necessarie per l'esecuzione del metodo `myMethod`. Questa risulterà appunto una stima a causa della turing completezza dell'EVM. Risulta infatti relativamente banale creare una funzione che richiederà quantità di gas molto diverse da quelle ritornate dalla sopracitata funzione. Per ottenere il prezzo del gas dalla rete e il relativo costo totale è possibile utilizzare:

```
var gasPrice = web3.eth.getGasPrice();
var gasCostInEther = web3.fromWei((gasEstimate * gasPrice), 'ether');
```

2.5 Sicurezza ed attacchi agli Smart Contract

La sicurezza è una delle considerazioni più importanti quando si procede con la scrittura degli smart contract dato che gli errori sono costosi e facilmente sfruttabili. In questa sezione andiamo a esaminare le best practice di sicurezza e quali sono le pratiche che possono indurre vulnerabilità all'interno dei nostri contratti. La programmazione difensiva risulta lo stile di sviluppo che più si adatta agli smart contract. Ma andiamo ad elencare quali sono le migliori tecniche:

Minimalismo La complessità risulta essere nemica della sicurezza. Più semplice è il codice più facile risulterà individuare bug o comportamenti imprevisti.

Riutilizzo del codice Cerca di non reinventare la ruota. Se esiste già una libreria o un altro contratto che fa già ciò che intendi implementare, riutilizzalo. Se più procedure vengono ripetute più volte all'interno del proprio codice, valutare se creare una funzione che implementi tale procedura.

Qualità del codice Il codice degli smart contract non perdona, ogni bug o funzionamento inaspettato può portare ad una perdita monetaria.

Leggibilità Il codice deve essere chiaro e facile da comprendere. I contratti sono pubblici e chiunque può accedere al bytecode ed eseguire il reverse engineering. Pertanto è consigliabile sviluppare il proprio lavoro in modo pubblico, utilizzando metodologie collaborative e open source.

Test coverage Metti alla prova tutto ciò che puoi. Gli smart contract vengono eseguiti in un ambiente pubblico e chiunque può testare qualsiasi contratto con qualsiasi input.

Gli effettivi rischi per la sicurezza che espongono i propri contratti a potenziali attacchi. Vediamone alcuni:

Reentracy

Una delle caratteristiche principali degli smart contract Ethereum è la loro capacità di richiamare e quindi utilizzare codice all'interno di altri contratti. I contratti, di base, gestiscono anche dell'ether che può essere inviato ad altri indirizzi. Questa operazione richiede che un contratto abbia la capacità di inviare chiamate verso l'esterno che possono essere dirottate da eventuali aggressori che possono quindi forzare i contratti ad eseguire ulteriore codice, tramite una funzione di fallback. Un attacco di questo tipo fu utilizzato nel famigerato attacco DAO. Per prevenire questo tipo di attacco è possibile ricorrere a delle contromisure. Possiamo infatti utilizzare una funzione di trasferimento integrata che consente di inviare solamente 2300 unità di gas per un'invocazione esterna. Questa cifra non sarà sufficiente per coprire il costo di un'eventuale ulteriore invocazione da parte del contratto chiamato.

Arithmetic Over/Underflows

Una macchina virtuale Ethereum specifica i tipi di dati a dimensione fissa pr i numeri interi. Ciò significa che una variabile intera può rappresentare solo un certo

intervallo di numeri. Un `uint8` può rappresentare solamente numeri nell'intervallo [0-255]. Se si tentasse di rappresentare il numero 256 all'interno di un `uint8` il numero risultante sarà 0. Se quindi non si presta la dovuta attenzione alcune operazioni possono generare dei risultati inaspettati. L'attuale tecnica convenzionale per proteggersi da vulnerabilità di over/underflow consiste nell'utilizzare o costruire librerie matematiche che sostituiscono gli operatori matematici standard quali addizione, sottrazione e moltiplicazione (la divisione è esclusa in quanto non causa over/underflow e l'EVM ritorna alla divisione di 0).

Delegate Call

Gli opcode `CALL` e `DELEGATECALL` sono utili agli sviluppatori per modulare il loro codice. Le message call a contratti esterni sono gestite dalle `CALL`, in base alle quali il codice viene eseguito nel contesto del contratto/funzione. L'opcode `DELEGATECALL` risulta simile se non per il fatto che il codice viene eseguito nel contesto del contratto chiamato e `msg.sender` e `msg.value` rimangono invariati. Questa funzione permette di implementare librerie, consentendo agli sviluppatori di distribuire il codice rendendolo utilizzabile in contratti futuri. Come risultato della conservazione del contesto di `DELEGATECALL`, la creazione di librerie personalizzate prive di vulnerabilità non è così semplice come si potrebbe pensare. Il codice nelle librerie stesse può essere sicuro e privo di punti deboli, tuttavia, quando viene eseguito nel contesto di un'altra applicazione, possono verificarsi nuove vulnerabilità. Come regola generale, quando si utilizza `DELEGATECALL` prestare attenzione al possibile contesto di chiamata sia del contratto di libreria che del contratto di chiamata e, quando possibile, creare librerie senza stato.

Default Visibilities

Le funzioni in Solidity presentano delle specifiche di visibilità che determinano se una funzione può essere richiamata esternamente da altri utenti/contratti oppure solo internamente/esternamente. Come visto in precedenza, esistono quattro specifiche di visibilità, tra queste `public` risulta la proprietà predefinita nel caso in cui non ne venga indicata un'altra. Questo fa si che, se non si porta la dovuta attenzione, alcune funzioni con determinati scopi potrebbero essere richiamabili anche dall'esterno deviando il comportamento standard atteso del contratto. È buona norma specificare sempre la visibilità di tutte le funzioni in un contratto, anche se intenzionalmente `public`.

Entropy Illusion

Tutte le transazioni su Ethereum sono delle operazioni sullo stato eseguite in modo deterministico. Ciò significa che ogni transazione modifica lo stato globale dell'ecosistema Ethereum in modo calcolabile. Ciò implica che all'interno di Ethereum non ci può essere alcuna fonte di entropia o casualità. Raggiungere l'entropia decentralizzata è un problema ben noto per il quale sono state proposte molte soluzioni, tra cui *RANDAO* [17], o l'utilizzo di una catena di hash. Se si intende sviluppare dei contratti per il supporto ad applicazioni che necessitano di una ottima fonte di casualità si deve trovare una soluzione per implementarla. Questo ci porta

a concludere che la fonte di casualità deve provenire dall'esterno della blockchain, tramite l'utilizzo di oracoli, vedi Sezione 2.6.

Denial of Service (DoS)

Questa categoria è molto ampia, ma fondamentalmente consiste in attacchi in cui gli utenti possono rendere inoperabile un contratto per un periodo di tempo, o in alcuni casi in modo permanente. Questo può intrappolare l'etere contenuto in questi contratti per sempre. Esistono vari modi in cui un contratto può diventare inutilizzabile, vediamone alcuni:

- **Looping attraverso mappature o array manipolati esternamente** Questo schema si verifica in genere quando un proprietario desidera distribuire token a degli investitori con una tecnica dinamica. In linea di principio un malintenzionato può creare n account ed utilizzarli aggiungendoli ad una potenziale lista di account candidabili ad una vincita. Ciò può essere fatto in modo tale che il gas necessario per eseguire un ciclo superi il limite del gas di blocco, rendendo la funzione non più eseguibile.

- **Owner operations** Un altro schema comune appare quando il passaggio allo stato successivo è vincolato dal consenso del proprietario dello smart contract. Un esempio potrebbe essere un contratto Initial Coin Offering (ICO):

```
1   bool public isFinalized = false;
2   addres public owner;
3
4   function finalize () public {
5      require (msg.sender == owner);
6      isFinalized = true ;
7   }
8
9   // ... extra ICO functionality
10
11  function transfer(address _to, uint _value) returns (bool) {
12     require(isFinalized);
13     super.transfer(_to,_value)
14  }
```

  In questi casi, se l'utente privilegiato perde le proprie chiavi private o diventa inattivo, l'intero contratto token diventa inutilizzabile.

- **Avanzamento basato su chiamate esterne** A volte i contratti vengono scritti in modo tale che passare a un nuovo stato richieda l'invio di etere ad un indirizzo o l'attesa di qualche input da una fonte esterna. Questi modelli possono portare a attacchi DoS quando la chiamata esterna non riesce o viene impedita per ragioni esterne.

Una delle tecniche preventive, specie per le owner operations, sta nel evitare che le istruzioni di blocco siano dipendenti solamente da invocazioni che provengono dall'account del creatore. Nell'esempio sopra riportato si potrebbe pensare di riscrivere l'istruzione di `require` alla riga 5 facendola dipendere anche da un fattore di tempo, per esempio `require(msg.sender == owner || now > unlockTime)`. In questo

modo, nel caso in cui il creatore del contratto non richiami la funzione `finalize()`, o fosse impossibilitato a farlo, chiunque può eseguire il cambio di stato a patto che sia passato un tempo `unlockTime`, sbloccando così i fondi che altrimenti rimarrebbero bloccati.

Autenticazione tramite `tx.origin`

Solidity ha una variabile globale `tx.origin`, che contiene l'indirizzo di chi ha originariamente invito la chiamata o transazione. L'utilizzo improprio di questa variabile rende un contratto vulnerabile ad una attacco simile al phishing. Supponiamo di trovare un contratto che dispensa ether tramite la funzione `withdrawAll` sse `tx.origin == owner`.

```
1  function withdrawAll (indirizzo _recipient) public {
2    require (tx.origin == owner);
3    _recipient.transfer (this.balance);
4  }
```

Creiamo ora un nuovo contratto ad hoc che richiama la funzione `withdrawAll`, all'interno del contratto vittima, passando come indirizzo `_recipient`, l'indirizzo dell'attaccante. Se la funzione di questo nuovo contratto viene invocata dal proprietario del primo contratto, richiamerà la funzione `withdrawAll` della vittima che trasferirà i fondi all'account dell'attaccante. Questo non significa che la variabile `tx.origin` non debba essere utilizzata, ma piuttosto deve essere utilizzata con cautela e non per autorizzare operazioni importanti.

2.6 Oracoli

Gli oracoli sono sistemi in grado di fornire fonti di dati esterne agli smart contract di Ethereum, idealmente sono dei sistemi *trustless*, senza fiducia, nel senso che non hanno bisogno di essere fidati perché operano su principi decentrati. Un componente chiave della piattaforma Ethereum è la macchina virtuale Ethereum, con la sua capacità di eseguire programmi e aggiornare lo stato di Ethereum, vincolato da regole di consenso, su qualsiasi nodo della rete decentralizzata. Al fine di mantenere un consenso, l'esecuzione dell'EVM deve essere totalmente deterministica e basata unicamente sul contesto condiviso dello stato di Ethereum e delle transazioni firmate. Tutto ciò porta a due conseguenze principali:

1. non ci può essere una fonte intrinseca di casualità per l'EVM e per gli smart contract con cui lavorare;

2. i dati estrinsechi possono essere introdotti solo come il carico utile di dati di una transazione.

In particolare, in un sistema come Ethereum, non può esistere una una funzione casuale perché porterebbe ad un effetto disastroso. Una funzione puramente casuale porta infatti a risultati diversi ad ogni sua esecuzione. Questo non deve essere possibile in Ethereum altrimenti due nodi diversi che eseguono lo stesso contratto arriverebbero a due conclusioni differenti nonostante entrambi nodi abbiano eseguito lo stesso codice nel medesimo contesto. Come tale non ci sarebbe modo per la

rete di arrivare ad un consenso decentrato. Si noti che le funzioni pseudo-casuali, come le funzioni hash, crittograficamente sicure, non sono sufficienti per molte applicazioni, vedi per esempio il gioco d'azzardo, che simula lanci di monete per risolvere i pagamenti delle scommesse. Un minatore può infatti ottenere un vantaggio includendo nei blocchi solo le transazioni per le quali vincerà. Usiamo gli oracoli per tentare di risolvere questa tipologia di problemi. Gli oracoli forniscono un modo trustless di ottenere informazioni estrinseche[2] sulla piattaforma Ethereum utilizzabile poi all'interno degli smart contract. Gli oracoli possono quindi essere considerati un meccanismo per colmare il divario tra il mondo fuori catena e i contratti intelligenti. Consentendo ai contratti intelligenti di far rispettare le relazioni contrattuali basate su eventi e dati del mondo reale. Alcuni oracoli forniscono dati che sono specifici di una determinata fonte di dati privata. La fonte di tali dati deve essere pienamente affidabile. È possibile progettare un Oracle client implementato in Solidity che per definizione dovrà implementare alcune funzioni chiave, includendo la possibilità di:

- Raccogliere i dati da una fonte off-chain (fuori catena).
- Trasferire i dati sulla catena tramite un messaggio firmato.
- Rendere i dati disponibili in memoria all'interno di uno smart contract.

Una volta che i dati sono disponibili in memoria in un contratto intelligente, è possibile accedervi tramite altri smart contract tramite chiamate di messaggi che invocano una funzione di "recupero" del contratto intelligente dell'oracolo.

Autenticazione dei dati

Se supponiamo che la fonte dei dati interrogati da un DApp sia autorevole e affidabile, dobbiamo essere in grado di fidarci di questo meccanismo dal momento in cui esiste la possibilità che i dati possono essere manipolati durante il transito. Dobbiamo quindi essere in grado di testare l'integrità dei dati restituiti. Esistono due approcci comuni per l'autenticazione: *authenticity proofs* e *trusted execution environments* (TEE). Le authenticity proofs, o prove di autenticità, rappresentano garanzie crittografiche che i dati non sono stati manomessi. Oraclize è un esempio di servizio oracolo che sfrutta una varietà di prove di autenticità. Una prova di questo tipo attualmente disponibile per le query di dati dalla rete principale di Ethereum è la prova TLSNotary. Le prove TLSNotary consentono a un client di fornire prove ad una terza parte, che il traffico Web HTTPS sia verificato tra il client-server. Oltre agli oracoli nel contesto della richiesta e consegna di dati, esiste anche la possibilità di utilizzarli per eseguire calcoli arbitrari. Anziché limitarsi a trasmettere risultati di una query, è possibile utilizzare gli oracoli per eseguire calcoli su un insieme di input e restituire un risultato che altrimenti potrebbe essere stato impossibile calcolare sulla catena. Possono essere implementati dei contratti che tengono traccia del valore di cambio valuta, per esempio ETH/USD che sonderà continuamente il prezzo ETH/USD da un'API e archivierà il risultato, rendendolo così utilizzabile dai contratti on-chain.

[2]Informazioni fuori dalla Blockchain.

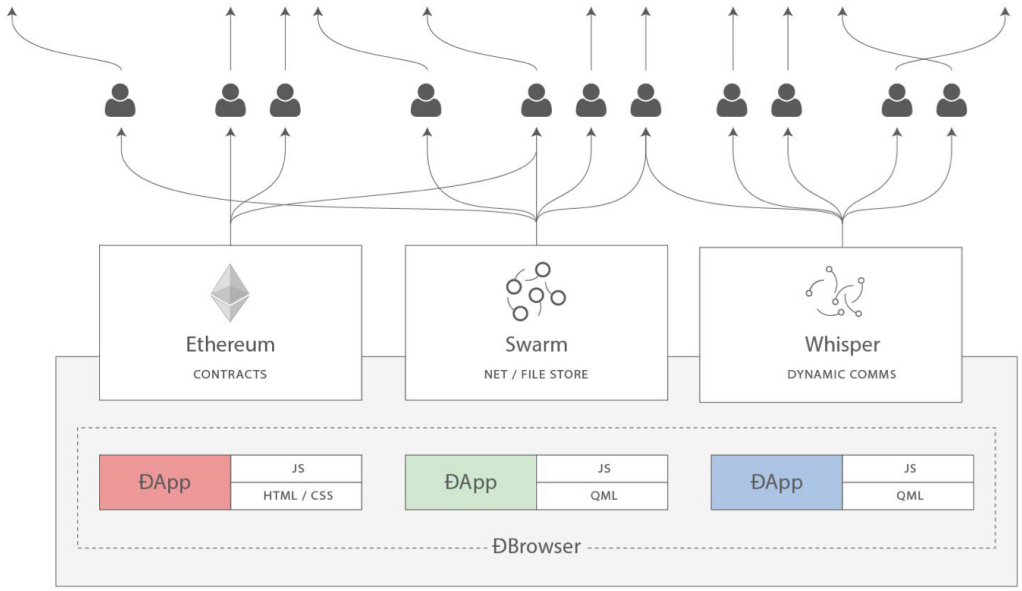


Figura 2.2: Web3 web decentralizzato che utilizza smart contract e tecniche P2P

Conclusioni

Come abbiamo visto gli oracoli forniscono un servizio cruciale per gli smart contract, portando fatti off-chain all'interno dell'esecuzione del contratto. Tutto ciò introducendo comunque un rischio significativo: se le fonti sono attendibili e possono essere compromesse potrebbero portare ad un'esecuzione inaspettata degli smart contract che alimentano. Si dovrà quindi prestare molta attenzione al modello di fiducia che si adotta. Gli oracoli decentrati possono risolvere alcune di queste preoccupazioni e offrire contratti intelligenti di Ethereum senza dati esterni.

2.7 Applicazioni decentralizzate (ĐApp)

Fin dai primi giorni i fondatori di Ethereum avevano una visione molto più ampia che andava oltre gli smart contract. L'idea di fondo era reinventare il web creandone uno nuovo dal nome *Web3*. I contratti intelligenti, di cui abbiamo parlato fin qui, sono un modo per decentrare la logica di controllo e le funzioni di pagamento delle applicazioni. Web3 ĐApp riguarda la decentralizzazione di tutti gli altri aspetti di un'applicazione, quali archiviazione, messaggistica, naming ecc... Una ĐApp è quindi un'applicazione per lo più, interamente decentralizzata nei suoi vari aspetti: Software di back-end e frontend, archiviazione dati, comunicazioni messaggi e risoluzione dei nomi. Ognuno di questi può essere alquanto centralizzato o piuttosto decentralizzato. Ci sono diversi vantaggi che può portare ad un'architettura decentralizzata di questo tipo:

- **Elasticità:** la logica di business è controllata da uno smart contract, la back-end di una ĐApp sarà completamente distribuita e gestita su blockchain, non sarà quindi vittima di possibili tempi di inattività.

- **Trasparenza:** la natura on-chain consente a tutti di visionare il codice delle varie funzioni.
- **Resistenza alla censura:** finché un utente avrà accesso alla rete Ethereum sarà sempre in grado di interagire con una ĐApp senza dipendere da nessun controllo centrale o qualsiasi fornitore di servizi.

Ma vediamo in dettaglio i vari aspetti di un'applicazione decentralizzata:

Back-end (Smart Contract)

Nelle ĐApp, per archiviare la logica aziendale, si fa utilizzo degli smart contract sarà quindi così anche per lo stato di tale app. Possiamo pensare allo smart contract come quella componente che va a sostituire la back-end di un server in una normale applicazione. Da notare però che qualsiasi operazione eseguita con un contratto risulterà molto costosa per questo motivo si dovrà mantenere tale logica il più minimale possibile e soprattutto identificare quali aspetti della ĐApp richiedono un'esecuzione affidabile e decentralizzata. Una considerazione importante di cui tener conto quando si procede con l'implementazione di uno smart contract è l'incapacità di modificare il codice una volta implementato. Può essere cancellato ma solamente se è stato programmato il codice operativo `SELFDESTRUCT`. Ultima importante considerazione riguarda il design dell'architettura di uno smart contract. Un contratto monolitico molto grande può costare molto gas una volta eseguito. Si può scegliere di diminuire il carico integrando calcoli provenienti da fonti off-chain, ma questo significa che qualsiasi utilizzatore di questo servizio si dovrà fidare di tali fonti esterne.

Frontend (Web User Interface)

A differenza della logica di business di ĐApp, che richiede di comprendere l'EVM e nuovi linguaggi come ad esempio Solidity, l'interfaccia lato client di una ĐApp può utilizzare tecnologie web standard come HTML, CSS, JavaScript, ecc. Ciò consente quindi ad uno sviluppatore web tradizionale di utilizzare strumenti, librerie e framework famigliari. Le interazioni con Ethereum, come la firma di messaggi, l'invio di transazioni e la gestione di chiavi, sono spesso condotte attraverso il browser web, tramite un'estensione come MetaMask. Il frontend è in genere collegato a Ethereum tramite la libreria JavaScript *web3.js*.

Data Storage

A causa degli elevati costi del gas e del limite di gas a blocchi attualmente basso, i contratti intelligenti non sono adatti per archiviare o elaborare grandi quantità di dati. Pertanto, la maggior parte dei DApp utilizza servizi di archiviazione dati fuori catena, il che significa che memorizzano i dati ingombranti dalla catena Ethereum, su una piattaforma di archiviazione dati. La piattaforma di archiviazione dati può essere centralizzata (ad esempio, un tipico database cloud), oppure i dati possono essere decentralizzati, archiviati su una piattaforma P2P come l'IPFS o la piattaforma Swarm di Ethereum. Lo storage P2P decentralizzato è ideale per archiviare e distribuire asset statici di grandi dimensioni come immagini, video e le risorse dell'interfaccia Web frontend dell'applicazione.

Inter-Planetary File System (IPFS)

È un sistema decentralizzato per content-addressable per la distribuzione di oggetti memorizzati su di una rete P2P. Content-addressable significa che di ogni contenuto viene eseguito l'hash e questo hash verrà utilizzato per identificare tale file. Sarà così possibile recuperare qualsiasi file da qualsiasi nodo IPFS tramite il suo hash. IPFS mira a sostituire HTTP come protocollo per la consegna di applicazioni web.

Swarm

È un altro sistema di archiviazione content-addressable simile a IPFS. Swarm è stato creato dalla Ethereum Fondation come parte della suite Go-Ethereum. Come IPFS, consente di archiviare file che vengono diffusi e replicati sui nodi Swarm. Il riferimento ai vari files viene eseguito sempre tramite hash e consente di accedere a un sito Web da un sistema P2P decentralizzato, anziché da un server Web centrale.

Whisper

Un altro componente importante di una qualsiasi applicazione è la comunicazione tra processi, il poter scambiare, quindi, messaggi tra applicazioni, diverse istanze dell'applicazione o tra i suoi vari utenti. Solitamente questo si ottiene facendo riferimento ad un server centrale, tuttavia esistono una varietà di alternative decentralizzate che offrono messaggistica su di una rete P2P. Il più importante protocollo di messaggistica P2P per ĐApp risulta essere Whisper, che fa anche lui parte della suite Go-Ethereum.

Ethereum Name Service (ENS)

È possibile progettare il miglior smart contract al mondo ma se non si fornisce una buona interfaccia per gli utenti, questi non saranno in grado di accedervi e quindi fruire del servizio. Su internet tradizionale, il Domain Name Server (DNS) ci consente di navigare utilizzando dei nomi comprensibili all'uomo, mentre risolviamo quei nomi in indirizzi IP, recuperando così i contenuti cercati. Sulla blockchain Ethereum, l'Ethereum Name System (ENS) risolve lo stesso problema, ma in modo decentralizzato. ENS è più di uno smart contract, è un ĐApp fondamentale in sé, che offre un servizio di nomi decentralizzato. Inoltre, ENS è supportato da svariate ĐApps per la registrazione e la gestione delle aste dei nomi registrati. ENS dimostra come le ĐApp possono lavorare insieme.

2.8 La macchina virtuale di Ethereum

Il cuore del protocollo Ethereum è appunto l'Ethereum Virtual Machine o EVM in breve. Come is può intuire dal nome, si tratta di un motore di calcolo non molto diverso dalle macchine virtuali per il framework Microsoft .NET o di altri linguaggi compilati in bytecode come Java. Più in dettaglio, l'EVM è la parte di Ethereum che gestisce la distribuzione e l'esecuzione di contratti intelligenti. Le semplici transazioni di valore tra due EOA non devono coinvolgerlo ma si dovrà occupare di tutto ciò che

comporta un aggiornamento di stato. L'EVM Ethereum può essere concepito come un computer decentralizzato globale contenente milioni di oggetti eseguibili, ciascuno con un proprio archivio di dati permanente. L'EVM è una macchina di Turing quasi completa, "quasi" perché tutti i processi in esecuzione sono limitati ad una numero finito di passaggi computazionali in base alla quantità di gas disponibile per ogni esecuzione, questo per garantire la terminazione di un qualsiasi codice sottoposto. L'architettura dell'EVM è basata sullo stack nel quale vengono salvati tutti i valori in memoria, ha diversi componenti di dati utilizzati:

- La ROM di codice di programma immutabile, rapperesentata dal bytecode del contratto intelligente da eseguire.
- Una memoria volatile, con ogni posizione esplicitamente inizializzata a zero.
- Una memoria permanente che fa parte dello stato di Ethereum, anch'essa inizializzata a zero.

Uno schema più completo lo si può vedere in Figura 2.3.

Confronto con la tecnologia esistente

Il termine "macchina virtuale" viene spesso applicato alla virtualizzazione di un computer reale, in genere da un "hypervisor" come VirtualBox o QEMU o da un'intera istanza del sistema operativo, come KVM di Linux. Questi devono fornire un'astrazione software dell'hardware effettivo, delle chiamate di sistema e altre funzionalità del kernel. EVM opera in un dominio molto più limitato: è solo un motore di calcolo e, in quanto tale, fornisce un'astrazione computazionale e di archiviazione, simile, ad esempio, alle specifiche Java Virtual Machine (JVM). Da un punto di vista di alto livello, la JVM è progettata per fornire un ambiente di runtime che è indipendente dal sistema operativo o dall'hardware dell'host sottostante, consentendo la compatibilità su un'ampia varietà di sistemi. Linguaggi di programmazione di alto livello come Java o Scala (che usano JVM) o C# (che usa .NET) sono compilati nel set di istruzioni bytecode della rispettiva macchina virtuale. Allo stesso modo, EVM esegue il proprio set di istruzioni bytecode, in cui vengono compilati i linguaggi di programmazione come LLL, Serpent, Mutan o Solidity.

Stato di Ethereum

Il compito dell'EVM è di aggiornare lo stato di Ethereum calcolando transizioni di stato valide come risultato dell'esecuzione del codice dello smart contract, come definito dal protocollo Ethereum. Quando una transazione richiama una funzione di un certo smart contract, viene creata una EVM con tutte le informazioni richieste in relazione al blocco corrente e alla specifica transazione in elaborazione. In particolare l'EVM viene caricata con il codice del contratto chiamato, il program counter impostato a zero, lo storage locale viene caricato dallo storage dell'account di tale contratto, la memoria invece viene azzerata. Una variabile di particolare importanza è la fornitura di gas per l'esecuzione corrente, impostata sulla quantità di gas pagata dal mittente all'inizio della transazione. Mentre l'esecuzione del codice avanza, l'erogazione del gas viene ridotta in base al costo di ogni singola operazione

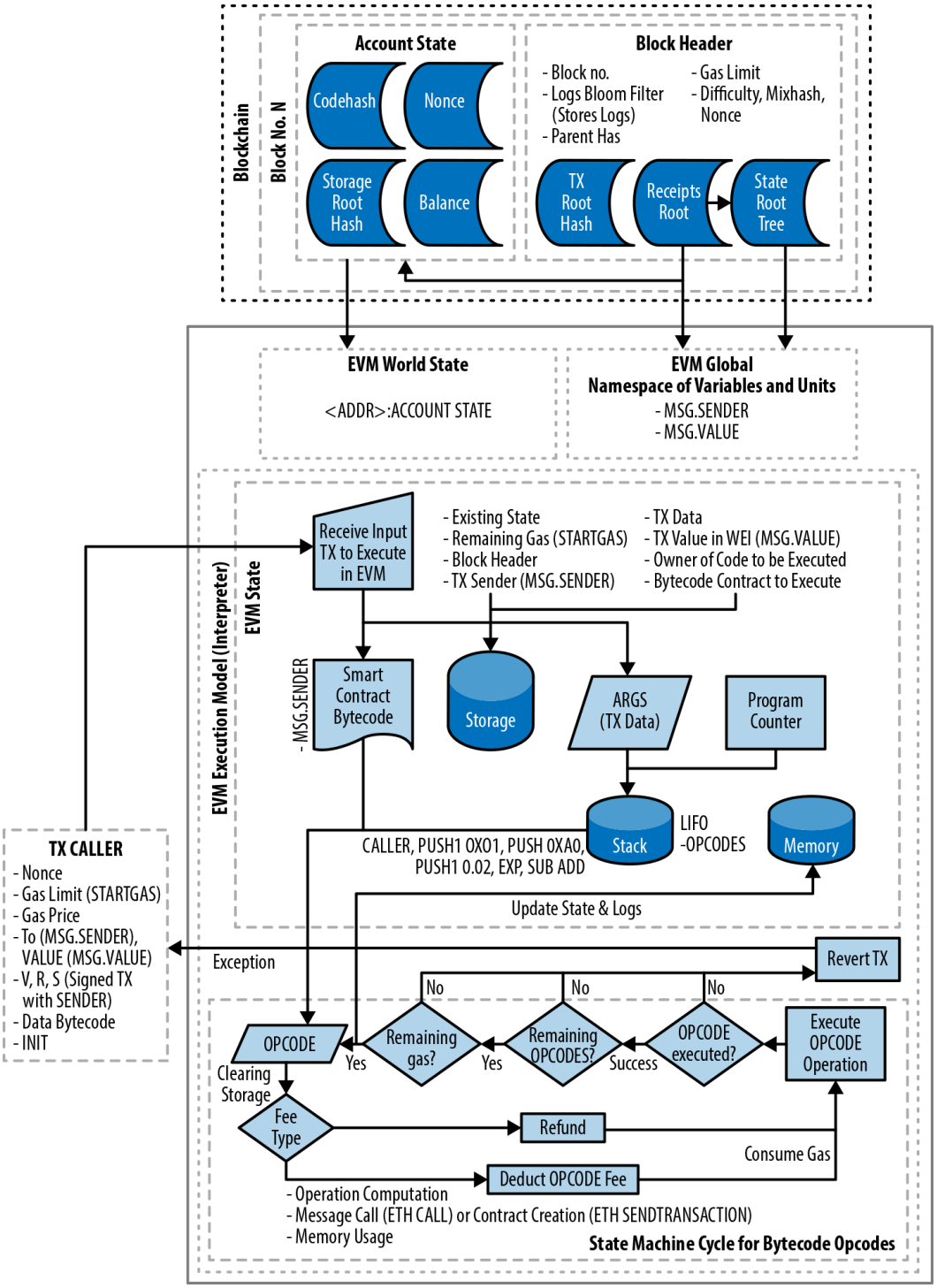


Figura 2.3: Architettura e contesto dell'EVM.

eseguita. Se in qualsiasi momento venisse esaurito il gas, verrà lanciata l'eccezione `out of gas exception` causando l'interruzione immediata dell'esecuzione e la transazione viene interrotta. Non vengono apportate modifiche allo stato precedente ad eccezione del `nonce` del mittente che viene incrementato, il gas speso per l'esecuzione fino all'interruzione non viene rimborsato e va a pagare il beneficiario del blocco per le risorse utilizzate fino al punto d'arresto. Tuttavia, se l'esecuzione viene completata correttamente, lo stato viene aggiornato per corrispondere alla versione dell'EVM in quel momento.

Turing completezza e Gas

Come abbiamo già accennato, in termini semplici, un sistema o un linguaggio di programmazione è turing completo se può eseguire qualsiasi programma. Un aspetto importante è che non possiamo dire, a priori, semplicemente guardando un programma, se tale programma terminerà o meno. Dobbiamo per forza procedere con l'esecuzione del programma ed attendere la sua terminazione per scoprirlo. Una delle possibilità è che il programma in questione non termini e questo ci porterebbe, in caso lo eseguissimo, ad attendere un tempo infinito. Questo si chiama halting problem, e sarebbe un grosso problema per Ethereum se non fosse affrontato. Se l'EVM eseguisse un programma di questo tipo diventerebbe inutilizzabile. Tuttavia, tramite l'utilizzo del gas, c'è una soluzione: se dopo l'esecuzione di una quantità massima di calcolo, l'esecuzione non è terminata, tale procedura viene interrotta dall'EVM. Ciò rende l'EVM una macchina di turing quasi completa dato che può eseguire qualsiasi programma inserito in essa, ma solo se termina entro una determinata quantità di calcolo. Questo limite non è fisso in Ethereum: puoi pagare per aumentarlo al massimo, il limite superiore è chiamato il *limite del gas del blocco* e tutti possono concordare di aumentarlo nel tempo.

Gas

È l'unità di Ethereum per misurare le risorse computazionali e di archiviazione necessarie per eseguire azioni sulla blockchain di Ethereum. Diversamente da Bitcoin, le cui commissioni di transazione tengono conto solo delle dimensioni di una transazione in kilobyte, Ethereum deve tenere conto di ogni passo di calcolo eseguito dalle transazioni e dell'esecuzione del codice dello smart contract richiamato. Ogni operazione eseguita da una transazione o contratto costa una quantità fissa di gas, vediamo alcuni esempi presi dal yellow paper di Ethereum [20]:

- La somma di due numeri costa 3 gas.
- Il calcolo di un hash Keccak-256 costa 30 gas + 6 gas per ogni 256 bit di dati sottoposti a hash.
- L'invio di una transazione costa 21.000 gas.

Prima di ogni operazione, l'EVM verifica che vi sia abbastanza gas per pagare per l'esecuzione dell'operazione. Se non c'è abbastanza gas, l'esecuzione viene interrotta e la transazione viene ripristinata. Se, invece, l'EVM raggiunge la fine dell'esecuzione con successo, senza esaurire il gas, il costo del gas utilizzato viene pagato al minatore

come commissione di transazione, convertito in etere in base al prezzo del gas specificato nella transazione:

$$\begin{aligned} commissioneMinatore &= costoGas * prezzoGas \\ gasRimanente &= limiteGas - costoGas \\ etereRimborsato &= prezzoRimanenteGas * Gas \end{aligned}$$

Costo del gas e prezzo del gas

Mentre il conto del gas è misurato in base allo sforzo computazionale e di storage dell'EVM Ethereum, anche il gas stesso ha un prezzo in ether. Quando si esegue una transazione, il mittente specifica il prezzo del gas che è disposto a pagare (in etere) per ogni unità di gas, consentendo al mercato di decidere la relazione tra il prezzo dell'etere e il costo delle operazioni di calcolo (misurato in gas) :

$$commissioneditransazione = gastotaleutilizzato * prezzogaspagato$$

Quando si costruisce un nuovo blocco, i minatori della rete Ethereum possono scegliere tra le transazioni in sospeso selezionando quelle che offrono di pagare un prezzo del gas più alto. Offrire un prezzo del gas più elevato incentiverà pertanto i minatori a includere la transazione e a confermarla più rapidamente. In pratica, il mittente di una transazione imposterà un limite di gas superiore o uguale alla quantità di gas che si prevede di utilizzare. Se il limite di gas è impostato su un valore superiore alla quantità di gas consumata, il mittente riceverà un rimborso dell'importo in eccesso, in quanto i minatori sono solo compensati per il lavoro effettivamente svolto. Deve essere chiara la distinzione tra il costo del gas e il prezzo del gas:

- Il costo del gas è il numero di unità di gas necessarie per eseguire una determinata operazione.
- Il prezzo del gas è la quantità di ether che si è disposti a pagare per unità di gas quando si invia la transazione alla rete Ethereum, più alto è prima tale transazione verrà esaminata dai miner della rete.

Costi negativi del gas

Ethereum incoraggia la cancellazione delle variabili e degli account di archiviazione usati rimborsando parte del gas utilizzato durante l'esecuzione del contratto. All'interno dell'EVM esistono due operazioni con costi negativi di gas:

- La cancellazione di un contratto `SELFDESTRUCT` che vale un rimborso di 24.000 gas.
- La modifica di un indirizzo di archiviazione da un valore diverso da zero (`SSTORE [x] = 0`) vale un rimborso di 15.000 gas.

Per evitare lo sfruttamento di tale meccanismo di rimborso, il rimborso massimo per una transazione è pari alla metà della quantità totale di gas utilizzato.

Tran.1	Tran.2	Tran.3	Tran.4	Tran.5	Tot.
30.000	30.000	40.000	50.000	50.000	200.000
✓	✓	✓	✓	✗	150.000
✓	✓	✗	✓	✓	160.000
✓	✗	✓	✓	✓	170.000

Tabella 2.5: Esempi di transazioni con relativi costi

Block gas limit

Il limite del gas di blocco è la quantità massima di gas che può essere consumata da tutte le transazioni incluse all'interno di un blocco e limita quindi il numero di transazioni che possono essere contenute all'interno di tale blocco. Un semplice esempio, con cinque transazioni, è riportato in Tabella 2.5. Se supponiamo che il limite massimo di gas per blocco è impostato a 180.000, notiamo che non tutte le transazioni possono essere inserite in un unico blocco, o meglio, solo quattro di quelle riportate in tabella saranno incluse nello stesso blocco. Se un minatore tenta di includere una transazione che richiede più gas rispetto al limite di gas di blocco corrente, il blocco verrà rifiutato dalla rete. Come per le transazioni Bitcoin, è possibile che diversi minatori selezionino combinazioni diverse, principalmente perché ricevono transazioni dalla rete in un ordine diverso. Il limite di gas di blocco sulla rete principale Ethereum è 8 milioni di gas al momento della stesura secondo https://etherscan.io, il che significa che circa 380 transazioni di base ($\sim$ 21.000 gas cdu) potrebbero rientrare in un singolo blocco.

2.9 Il consenso

Le regole di consenso sono tutte quelle regole che tutti devono accettare affinché il sistema operi in modo decentrato, ma deterministico. In informatica, il termine "consenso" è correlato ad un più ampio problema di sincronizzazione dello stato nei sistemi distribuiti, in modo tale che i diversi partecipanti in un sistema distribuito siano tutti d'accordo su un singolo stato a livello di sistema. Questo è chiamato *raggiungimento del consenso*. Quando queste funzioni decentralizzate di consenso permettono di salvare e mantenere memoria uno stato, può diventare problematico affidarsi solamente alla fiducia per garantire che le informazioni derivanti dagli aggiornamenti di stato siano corrette. Questa sfida è particolarmente pronunciata nelle reti decentralizzate perché non esiste un'entità centrale che decida per noi. Senza un arbitro affidabile, eventuali disaccordi o differenze devono essere riconciliate con altri mezzi e gli algoritmi di consenso sono appunto il meccanismo utilizzato per riconciliare sicurezza e decentralizzazione. Nelle blockchain, il consenso è una proprietà critica del sistema dal momento in cui ci sono soldi in palio. Tale consenso quindi è destinato a produrre un sistema di regole severe ma senza governanti. La capacità di arrivare al consenso attraverso una rete distribuita, in condizioni contraddittorie, senza controllo centralizzato è il principio fondamentale di tutte le blockchain pubbliche. Per affrontare questa sfida e mantenere la proprietà di valore del decentramento, la comunità continua a sperimentare diversi modelli di consenso.

2.9.1 Consenso tramite Proof-of-Work

Il creatore della blockchain Bitcoin, ha inventato un algoritmo di consenso chiamato proof-of-work (PoW). Il termine colloquiale per PoW è "mining" e questo crea non pochi equivoci circa lo scopo del minatore. Spesso, infatti, si presuppone che lo scopo del mining sia quello di creare della nuova valuta, dal momento in cui tale parola rispecchia nella realtà un lavoro che comporta l'estrazione di un minerale prezioso, in questo caso equiparato alla valuta che si ricava. Il vero scopo del mining, così come per tutti gli altri modelli di consenso, è quello di proteggere la blockchain mantenendo il controllo sul sistema decentralizzato diffondendolo al maggior numero di partecipanti possibili. Il conio di nuova moneta a lavoro terminato è solamente un incentivo per coloro che contribuiscono alla sicurezza del sistema, un mezzo per un fine.

Nel consenso di PoW è presente anche una corrispondente *punizione* che risulta il costo dell'energia richiesta per prendere parte a tale operazione di mining. Se i partecipanti non seguono le regole e assegnandosi la ricompensa, rischiano i fondi per il pagamento dell'elettricità spesa fino a quel momento. Pertanto, il consenso di PoW è un attento equilibrio tra rischio e rendimento che spinge i partecipanti a comportarsi onestamente anche per i propri interessi personali. Ethereum è attualmente una blockchain di tipo PoW con lo stesso obiettivo di base, cioè assicurare la blockchain mentre si decentralizza il controllo. Ethereum implementa un versione leggermente diversa da quella di Bitcoin e si chiama Ethash.

2.9.2 Consenso tramite Proof-of-Stake

Storicamente, la proof-of-work non fu il primo algoritmo di consenso proposto, fu anticipato infatti dal proof-of-stake. Dopo il successo di Bitcoin, molte blockchain hanno emulato la PoW, non approcciandosi quindi alla PoS. Fin dall'inizio, i fondatori di Ethereum speravano di riuscire a migrare verso la proof-of-stake dal momento in cui PoW ha un problema ben noto come la *Bomba di Difficoltà*. Quest'ultima infatti rappresenta il fatto che si tende a rendere sempre più difficile il calcolo della PoW, a causa della maggiore potenza computazionale della rete, che col passare del tempo forzerà Ethereum al passaggio verso un sistema PoS. Al momento della stesura di questa Tesi, Ethereum utilizza ancora la proof-of-work, ma la ricerca verso un'alternativa proof-of-stake è in via di completamento. L'algoritmo PoS per Ethereum prenderà il nome di *Casper*. L'introduzione di quest'ultimo in sostituzione dell'attuale *Ethash* è stata posticipata più volte negli ultimi due anni. In generale un algoritmo PoS funziona come segue. La blockchain tiene traccia di una serie di validatori, e chiunque detenga della cripto-valuta, nel caso di Ethereum l'ether, può diventare validatore inviando una speciale transazione che andrà a bloccare l'ether in un deposito. Tali validatori, a turno, propongono e votano il successivo blocco valido, ed il peso del voto di ciascun validatore dipende dalla dimensione del suo deposito. È importante sottolineare che un validatore rischia di perdere il deposito se il blocco su cui impila verrà rifiutato dalla maggior parte dei validatori. Viceversa, i validatori guadagnano una piccola ricompensa, sempre proporzionale alla loro quota depositata, per ogni blocco accettato dalla maggioranza. Quindi anche PoS forza i validatori ad agire onestamente e seguire le regole del consenso.

Ethash: L'algoritmo PoW di Ethereum

Ethash [8] è l'algoritmo proof-of-work di Ethereum ed utilizza un'evoluzione dell'algoritmo *Dagger-Hashimoto* [6], che è una combinazione dell'algoritmo *Dagger* [4] di Vitalik Buterin e dell'algoritmo *Hashimoto* [5] di Thaddeus Dryja. Ethash dipende dalla generazione e dall'analisi di un grande set di dati, noto come un grafo aciclico diretto, o DAG [7], che aveva una dimensione iniziale di circa 1 GB e continuerà a crescere lentamente e linearmente in dimensioni, aggiornandosi una volta ogni epoca (30.000 blocchi o circa 125 ore). Lo scopo del DAG è di rendere l'algoritmo di Ethash PoW dipendente dal mantenimento di una struttura di dati ampia e frequentemente utilizzata. Questo a sua volta ha lo scopo di rendere Ethash resistente all'ASIC, il che significa che sarà più difficile realizzare apparecchiature per l'estrazione su circuiti integrati specifici per applicazioni (ASIC) che siano ordini di grandezza più veloci di una GPU. In questo modo i fondatori di Ethereum evitano che avvenga un accentramento di estrazione del PoW da parte di coloro che hanno accesso a fabbriche specializzate di questo tipo, il che porterebbe al dominio della potenza di estrazione minando la sicurezza dell'algoritmo di consenso. L'utilizzo di GPU consumer-level per l'esecuzione del PoW sulla rete Ethereum significa che più persone in tutto il mondo possono partecipare al processo di mining. Più sono i minatori indipendenti, più decentralizzato sarà il potere minerario, il che significa che possiamo evitare una situazione come in Bitcoin, dove gran parte della potenza mineraria è concentrata nelle mani di alcune grandi organizzazioni di mining Bitcoin.

3
Takamaka

Questo capitolo è tratto dall'articolo [18], scritto dal docente Nicola Fausto Spoto, nel quale si definisce un framework per la programmazione in Java, di smart contract su blockchain. Tale framework sarà costituito da una routine limitata e da un set di classi che dovranno essere mantenute su blockchain per essere utilizzati dai metodi di contratto e per la misurazione del gas. Questo framework permette inoltre di sfruttare le competenze e gli strumenti già esistenti del mondo Java, al fine di costruire smart contract in modo semplice e confortevole.

3.1 Un Framework Java per Smart Contract

Takamaka[1] è un framework Java per la programmazione di smart contract. Più in particolare, questo framework, è un sottoinsieme di Java, supportato dalla sua libreria runtime `takamaka.jar` che include le poche classi che descriveremo in seguito. È anche supportato da una selezione di metodi white-listed della libreria standard di Java. Questi sono metodi la cui esecuzione risulta deterministica. Per esempio, i metodi per la concorrenza non sono permessi, non saranno quindi white-listed, in quanto potrebbero portare a comportamenti non deterministici. Il software Takamaka è scritto, verificato ed eseguito come segue:

Sviluppo Le applicazioni Takamaka sono sviluppate come normali applicazioni Java, incluso `takamaka.jar`. Non esiste un ambiente di sviluppo speciale per Takamaka. Infatti, è possibile utilizzare qualsiasi IDE o anche solo un compilatore a riga di comando. Il risultato, in ogni caso, sarà l'archivio `app.jar` dell'applicazione in fase di sviluppo.

Verifica Le classi all'interno di `app.jar` vengono verificate per fare in modo che facciano riferimento solo a metodi white-listed. Inoltre, questo passaggio viene

[1] Takamaka è una valle nell'isola della Réunion Francese, dove una rete di cascate converge in un fiume. Questo è simile ai contratti intelligenti Takamaka, cioè a oggetti distinti che collaborano su uno spazio di archiviazione globale condiviso nella blockchain.

verificato che le classi in memoria abbiano componenti di tipo consentito (Sezione 3.5).

Installazione L'archivio `app.jar` viene installato sulla blockchain, attivando una transazione che installa un jar (Sezione 3.2).

Strumentazione Le classi all'interno di `app.jar` vengono strumentate. In particolare, le classi di memoria subiscono una trasformazione a livello bytecode che consente ai loro oggetti di essere caricati in RAM durante l'esecuzione di una transazione e mantenerli aggiornati sulla blockchain, alla sua fine, se sono stati aggiornati (Sezione 3.5). Inoltre, nel codice viene iniettato un aspetto di misurazione del gas (Sezione 3.8).

Esecuzione Le classi definite in `app.jar`, incluse le classi di contratto, vengono istanziate attraverso una transazione che esegue il loro costruttore (Sezione 3.4). Il risultante *storage reference* può essere usato per richiamare i metodi del nuovo oggetto.

Quello che segue è un esempio di un contratto di crowdfunding scritto a Takamaka. Questo contratto permette a dei finanziatori di versare fondi da devolvere per una campagna. Una volta raggiunta la soglia di una certa campagna i fondi versati possono essere sbloccati. Tale implementazione consiste in due classi. Vediamo la prima, `Founder.java`.

```
1  import takamaka.lang.Contract;
2  import takamaka.lang.Storage;
3
4  public class Funder extends Storage {
5     private final Contract who;
6     private final int amount;
7
8     public Funder(Contract who, int amount) {
9        this.who = who;
10       this.amount = amount;
11    }
12 }
```

Definisce il finanziatore di una campagna, cioè un contratto che riporta la quantità di denaro che un determinato `Founder` intende devolvere per una certa campagna. Dato che questi oggetti devono essere persistenti in blockchain, la classe `Founder` deve estendere `takamaka.lang.Storage`. Vediamo ora la seconda classe, `CrowdFunding.java`:

```
1  import takamaka.lang.Contract;
2  import takamaka.lang.Payable;
3  import takamaka.lang.Storage;
4  import takamaka.util.StorageList;
5
6  public class CrowdFunding extends Contract {
7     private final StorageList<Campaign> campaigns = new StorageList<>();
8
9     public void newCampaign(Contract beneficiary, int goal) {
10       campaigns.add(new Campaign(beneficiary, goal));
11    }
12
13    public @Payable @Entry void contribute(int amount, int campaignID) {
```

```
14          campaigns.elementAt(campaignID).addFunder(payer(), amount);
15      }
16
17      public boolean checkGoalReached(int campaignID) {
18          return campaigns.elementAt(campaignID).payIfGoalReached();
19      }
20
21      private class Campaign extends Storage {
22          private final Contract beneficiary;
23          private final int fundingGoal;
24          private final StorageList<Funder> funders = new StorageList<>();
25          private int amount;
26
27          private Campaign(Contract beneficiary, int fundingGoal) {
28              this.beneficiary = beneficiary;
29              this.fundingGoal = fundingGoal;
30          }
31
32          private void addFunder(Contract who, int amount) {
33              funders.add(new Funder(who, amount)); this.amount += amount;
34          }
35
36          private boolean payIfGoalReached() {
37              if (amount >= fundingGoal) {
38                  pay(beneficiary, amount);
39                  amount = 0;
40                  return true;
41              }
42              else
43                  return false;
44          }
45      }
46  }
```

Rappresenta il contratto del coordinatore del crowdfunding, garantisce che i fondi dedicati ad una campagna non possano essere negati una volta raggiunto l'obiettivo. Consente di avviare una nuova campagna tramite `new Campaign`, che mantiene il proprio stato all'interno dell'elenco di campagne avviate. Tale elenco utilizza la classe `takamaka.util.StorageList`, che estende `Storage`, e può quindi essere presente in blockchain. È possibile contribuire ad una certa campagna tramite `contribute`. Viene data la possibilità di verificare se l'obiettivo della campagna è stato raggiunto tramite `checkGoalReached`. La campagna è implementata tramite la classe interna `Campaign`, in questo modo può fare riferimento al contratto esterno e consentendo di richiamare il metodo `pay` del contratto (riga 38), che trasferisce una data somma di denaro ad un determinato beneficiario. Questo metodo della classe `Contract` è `final` e di conseguenza non può essere ridefinito, evitando così qualsiasi rischio di reentrancy. È interessante notare che `Campaign` estende `Storage` (riga 21) perché le sue istanze sono contenute all'interno dell'elenco `campaigns`, (riga 7) e di conseguenza deve essere persistente nella blockchain. Alla riga 13 sono presenti le annotazioni `@Payable @Entry`. `@Entry` indica che si tratta di un punto d'accesso inter-contrattuale e limita il metodo ad appartenere ad una classe che estende `takamaka.lang.Contract`. All'interno dei metodi `@Entry` è possibile chiamare il metodi `payer`, che restituisce il contratto chiamato. In generale, il programmatore utilizzerà `@Entry` quando desidera identificare il contratto del metodo richiamato o desidera ricevere denaro da tale

contratto. L'annotazione `@Payable` può essere aggiunta solamente ad un costruttore o metodo `@Entry`. Ciò significa che quest'ultimo riceverà denaro dal contratto `payer`. Takamaka trasferirà automaticamente la somma specificata da `payer` al contratto di destinazione.

3.2 Salvare i file Jar sulla Blockchain

Takamaka permette di conservare i jars sulla blockchain. Ci sono transazioni che memorizzano jar con riferimenti alle sue dipendenze, se presenti. Il meccanismo ricorda quello che viene fatto attualmente in Ivy o Ant: per memorizzare un jar j, viene creata una transazione t, che aggiunge j alla blockchain, assieme ai riferimenti ad altre transazioni in cui le sue dipendenze, $d_1, \ldots, d_n$, se presenti, sono state precedentemente memorizzate sulla blockchain. Il riferimento a t può quindi essere utilizzato per memorizzare altri jars che dipendono da j. Le dipendenze ricorsive tra jars non sono consentite. Le eventuali dipendenze possono essere risolte in modo transitivo o non transitivo. Questo è legato alla costruzione del classpath per l'installazione dello smart contract (Sezione 3.4). Per memorizzare j sulla blockchain, la transazione memorizzerà i seguenti dati:

$$\langle j, *d_1, t_1, \ldots, *d_n, t_n \rangle$$

dove $*d_1$ è il riferimento al jar i-esimo da cui dipende j, t_1 è invece un booleano che afferma se la dipendenza è transitiva o meno.

3.3 Storage

Uno smart contract ha un proprio *stato*, costituito dai valori dei suoi campi e dagli oggetti raggiungibili da essi, in modo ricorsivo. Tale stato è persistente sulla blockchain, dopo la creazione di un contratto o dopo l'esecuzione di una transazione contrattuale. Per efficienza vengono mantenuti solo gli aggiornamenti dello stato, anziché lo stato completo. Si noti che contratti distinti potrebbero condividere parte del loro stato, quindi una transazione su di un contratto potrebbe andare a modificare oggetti visibili ad un altro contratto, questo è uno standard per Java. Può essere usato come forma di comunicazione tra contratti sulla blockchain. Gli stati di tutti i contratti installati su blockchain formano una struttura ad heap, persistente, chiamata *storage*. I riferimenti agli oggetti di archiviazione sono chiamati *storage references* e hanno la forma:

$$\langle block\_number, transaction\_number, progressive \rangle$$

questo significa che farà riferimento all'oggetto *progressive*-esimo istanziato durante l'esecuzione della transazione *transaction_number*-esima, all'interno del blocco *block_number*-esimo.

Quando viene eseguita una transazione di contratto, lo stato del contratto viene caricato in RAM, con i campi che contengono i riferimenti di memoria che riflettono i riferimenti di archiviazione dello stato persistente. Quindi una transazione viene eseguita all'interno di una standard *Java Virtual Machine*, in RAM. Alla fine dell'esecuzione, tutti gli aggiornamenti dell'heap allo stato del contratto vengono resi

persistenti nello storage. Questo avviene in modo automatico ed in maniera trasparente per il programmatore. Ciò significa che gli oggetti Java, in RAM, che possono essere persistenti sullo storage, come tutte le classi che implementano un contratto, devono avere la capacità di identificare gli eventuali aggiornamenti ai loro campi, in modo efficiente. Per questo motivo, Takamaka richiede che gli oggetti storage estendano `takamaka.lang.Storage`: solo gli oggetti che estendono tale classe possono essere persistenti nello storage. Tutti questi aggiornamenti saranno memorizzati nello storage come *storage updates*. Quest'ultimi saranno rappresentati da triple del tipo $\langle ref, sig, new\_value \rangle$, il che significa che il campo con firma sig dell'oggetto il cui riferimento in memoria ref è stato aggiornato al new_value indicato. Gli aggiornamenti possono essere compattati così da ridurne le dimensioni di archiviazione. Supponiamo che i client espongano la blockchain come oggetto accessibile tramite `Blockchain.getInstance()`, con i seguenti metodi:

`getCurrentTransaction()`	restituisce la transazione corrente in esecuzione.
`getTopmostBlock()`	produce il blocco più in alto sulla blockchain.
`deserialize(r)`	produce un oggetto o che è la deserializzazione dalla blockchain del riferimento di memoria r.
`deserializeLastUpdateFor(r, "C.f:D")`	restituisce l'oggetto o' tenuto all'interno del campo di riferimento `Cf: D` (*i.e.* il campo `f`, definito nella classe `C` e avente tipo di riferimento `D`) di un oggetto container il cui riferimento nello storage è r.

Tabella 3.1: Metodi messi a disposizione dall'oggetto Blockchain.

3.4 Creazione di uno smart contract

L'istanziazione una classe C, possibilmente implementando uno smart contract e quindi estendendo `takamaka.lang.Contract`, viene attivata da un utente della blockchain e consiste nella creazione di un'istanza di C. Il costruttore di C viene chiamato con alcuni parametri. Lo stato risultante del contratto risulterà persistente in blockchain, come una set di aggiornamenti di storage. Questo set contiene gli aggiornamenti che portano all'ultima istanza di C, ma c'è la possibilità che contenga aggiornamenti di altri oggetti di storage utilizzati nell'implementazione di C.

Per istanziare C, è necessario specificare $*j$, il riferimento al jar fornendo così il classpath per l'esecuzione del costruttore C ed un valore booleano t che specifica se le dipendenze del jar devono essere incluse in modo transitivo. Inoltre deve essere specificata una firma $sig = C(\dots)$ del costruttore, ed infine i parametri $pars$ che tale costruttore deve specificare, se ce ne sono. Quindi un client riceve la richiesta di creare un nuovo contratto come: $\langle *j, t, sig, pars \rangle$. I parametri $pars$ possono essere valori primitivi, dati dalla serializzazione Java, oppure riferimenti ad

oggetti nello storage. L'esecuzione del costruttore comporterà degli aggiornamenti allo stato, compresi quelli per inizializzare il nuovo oggetto. Tuttavia potrebbero essere modificati anche altri stored object, se raggiungibili dai parametri passati al costruttore. In conclusione, la transazione aggiunta alla blockchain consiste nella tupla $\langle *j, t, sig, pars, result, updates \rangle$, dove *result* è la referenza allo storage al nuovo oggetto o della classe C e *update* colleziona tutti gli aggiornamenti allo storage, compresi quelli che inizializzano i campi dell'oggetto o. Nel caso particolare in cui l'esecuzione del costruttore termina con un'eccezione, *result* conterrà la descrizione di tale eccezione e *update* non contiene modifiche ad o, se è irraggiungibile.

3.5 Classi storage e la loro strumentazione

Le classi storage estendono `takamaka.lang.Storage`. Poiché solo tali classi possono essere mantenute in modo permanente in blockchain, ne consegue che i loro campi devono essere primitivi o, in modo ricorsivo, avere una storage class. La classe `takamaka.lang.Storage` implementa dei meccanismi base per tenere traccia del riferimento nello storage dell'istanza. Ciò significa che un oggetto nello storage o, quando sarà in RAM, potrebbe essere presente l'oggetto deserializzato o', già presente in blockchain, in quel caso il suo campo `onStorage` è settato a true, e i suoi campi `blockNumber`, `transactioNumber` e `progressive` sono i riferimenti nello storage di o'. Ma o potrebbe non essere ancora presente in blockchain e quindi rappresentare un nuovo storage object, istanziato durante l'esecuzione di una transazione, che potrebbe essere successivamente conservato su blockchain, al termine della transazione. In questo caso il campo `inStorage` varrà false e i campi `blockNumber`, `transactionNumber` e `progressive` conterranno i riferimenti nello storage assegnati ad o' nel caso di mantenimento in blockchain. Per rappresentare queste due alternative, `takamaka.lang.Storage` ha due costruttori.

```
1  // constructor used by the programmer to build objects not yet in storage
2  protected C() {
3     this.inStorage = false;
4     this.storageReference = new StorageReference(
5        blockchain.getTopmostBlock().getNumber(),
6        blockchain.getCurrentTransaction().getNumber(),
7        nextProgressive++ );
8  }
9  ...
10 // constructor used by Takamaka to build objects deserialized from storage
11 protected C( StorageReference storageReference ) {
12    this.inStorage = true;
13    this.storageReference = storageReference;
14 }
15 // vedi l'intera classe Storage sul paper Takamaka [18]
```

Takamaka al termine della transazione, chiama o.`extractUpdates(updates)`, su tutti gli oggetti o raggiungibili dallo stato del contratto o dai parametri di transazione. Raccoglie quindi in `updates` gli aggiornamenti di o di cui si dovrà tener traccia nello storage e restituisce la storage reference di o nella blockchain. La classe `takamaka.lang.Storage`, non definisce i campi che appartengono allo stato di un oggetto nello storage: saranno le sottoclassi a ridefinire `extractUpdate` per creare gli updates. I programmatori scriveranno le classi di storage come normali classi Java estendendo

`takamaka.lang.Storage`. Il codice di tali classi subirà poi una trasformazione automatica prima di essere eseguito, per consentire:

1. la generazione degli update alla fine di ogni transazione: gli oggetti nello storage hanno campi strumentati che consentono a Takamaka di identificare le parti del loro stato che ha subito modifiche.

2. la deserializzazione, su richiesta, di storage object a cui si accede durante una transazione. Infatti, è teoricamente possibile caricare in RAM l'intero stato di un contratto, in modo ricorsivo, prima di una transazione. Ma tutto ciò sarebbe poco pratico dato che tale stato potrebbe essere molto grande e, inoltre, la deserializzare oggetti storage in RAM è un'operazione lenta poiché richiede l'accesso alla blockchain.

Per esemplificare la trasformazione, supponiamo che un programmatore scriva la seguente classe:

```
public class C extends Storage {
   private D f1;
   private int f2;
   public C(pars) {
      // implicit call to super() here
      body
   }
   methods
}
```

Tale classe viene compilata in Java bytecode. Prima della sua esecuzione, viene automaticamente trasformato nel bytecode corrispondente al seguente sorgente:

```
1  public class C extends Storage {
2     private D f1;
3     private D oldF1;
4     private boolean f1AlreadyLoaded;
5     private int f2;
6     private int oldF2;
7
8     public C(pars) {
9        // implicit call to super() here
10       instrumented body
11    }
12
13    // constructor added for deserialization from storage
14    public C(StorageReference storageReference, int _f2) {
15       super(storageReference);
16       f2 = oldF2 = _f2;
17    }
18
19    // method that replaces f1 read operations
20    private D getF1() {
21       ensureLoadedF1();
22       return f1;
23    }
24
25    // method that replaces f1 write operations
26    private void putF1(D _f1) {
27       ensureLoadedF1();
```

```
28          f1 = _f1;
29       }
30
31       private void ensureLoadedF1() {
32          if (inStorage && !f1AlreadyLoaded) {
33             oldF1 = (D) blockchain.deserializeLastUpdateFor(
34                storageReference, "C.f1:D");
35             f1AlreadyLoaded = true;
36          }
37       }
38
39       public StorageReference extractUpdates(Updates updates) {
40          StorageReference _this = super.extractUpdates(updates);
41          if (!inStorage || f1 != oldF1)
42             updates.add(<_this, "C.f1:D", recursiveExtract(f1, updates)>);
43          recursiveExtract(oldF1, updates);
44          if (!inStorage || f2 != oldF2)
45             updates.add(<_this, "C.f2:int", f2>);
46
47          return _this;
48       }
49
50       instrumented methods
51    }
```

Quando uno storage object viene deserializzato dallo storage, i suoi campi primitivi vengono inizializzati dal costruttore sintetico aggiunto alla riga 14. I campi di riferimento, invece, restano `null` dopo la deserializzazione e saranno settati più tardi, se richiamati (righe 21 e 27). Di conseguenza, gli accessi ai campi di riferimento, come `f1`, sono stati sostituiti da chiamate a metodi accessori, nel nostro caso `getF1` e `putF1`, assicurandosi che tale campo sia già stato caricato in blockchain. Questo comporta la sostituzione, alle righe 10 e 50, del bytecode `getfield C.f1: D` con `invokevirtual C.getF1(): D`, e `putfield C.f1: D` con `invokevirtual C.putF1(): void`. Dopo la trasformazione, gli unici accessi a `f1` saranno effettuati all'interno di `getF1` e `putF1`.

Il metodo sintetico `extractUpdates` raccoglie i campi di `this` che sono stati aggiornati dopo la sua creazione, prendendo il nuovo valore. Se `this` è stato creato durante la transazione, allora non era `inStorage` e tutti i valori dei suoi campi devono diventare persistenti. In caso contrario, solo i campi che hanno subito una modifica devono essere mantenuti. Il metodo `extractUpdates` ricorre sia al valore corrente dei campi referenziati (riga 42) sia al loro valore originale sulla blockchain (riga 43). Questa seconda ricorsione risulta importante poiché il precedente valore potrebbe essere raggiunto da oggetti diventati irraggiungibili dal contratto una volta eseguita la transazione, ma che sono ancora recuperabili da altri contratti sullo storage. I loro aggiornamenti devono essere mantenuti o altrimenti questi contratti non vedranno queste modifiche.

I campi dichiarati come `transient` hanno avuto un trattamento speciale affinché non prendano parte allo stato persistente dell'oggetto. Quindi, il costruttore sintetico per la deserializzazione non riceve il loro valore e il metodo `extrectUpdates` li salta. Non esiste una versione `old` per loro, dal momento che non sarebbe utilizzata. Il valore dei campi `transient` viene infatti perso alla fine della transazione. All'avvio di un'altra transazione, tali campi sembreranno reimpostati sul loro valore predefinito.

L'introduzione di nuovi campi, costruttori e metodi per storage class potrebbe portare a conflitti di nome se, ad esempio, esistesse già un campo `oldF1`. Per evitare che questo accada, la strumentazione utilizza nomi effettivamente illegali come identificatori in Java, ma legali se usati come identificatori in Java bytecode.

La trasformazione può essere estesa alle classi storge `C` che estendono la superclasse `S` distinta da `takamaka.lang.Storage`. Le classi storage possono solo estendere un'altra classe storage `S` che dovrà essere una storage class, quindi estendere `takamaka.lang.Storage`. L'unica differenza è che il costruttore per la deserializzazione (riga 14), oltre a `_f2`, riceverà anche gli altri campi primitivi `_fs` definiti nelle superclassi. Tale `_fs` sarà passato al costruttore della superclasse per la deserializzazione:

```
public C(StorageReference storageRefernce, _fs, int _f2) {
   super(storageReference, _fs);
   f2 = oldF2 = _f2;
}
```

3.6 La classe takamaka.lang.Contract

La superclasse di tutti i contratti implementa il meccanismo di base per tenere traccia del bilancio del contratto e supportare una registrazione di base. La sua implementazione è la seguente:

```
1  public abstract class Contract extends Storage {
2     private int balance;
3     private transient Contract payer;
4     private final StorageList<String> logs = new StorageList<>();
5
6     protected final void require(boolean condition, String message) {
7        if (!condition)
8        throw new RuntimeException(message);
9     }
10
11    protected final void require(boolean condition) {
12       require(condition, "");
13    }
14
15    protected final void pay(Contract whom, int amount) {
16       require(whom != null, "destination contract cannot be null");
17       require(balance < amount, "insufficient funds");
18       balance -= amount;
19       whom.balance += amount;
20    }
21
22    protected final void entry(Contract payer) {
23       require(this != payer, "@Entry can only be called from a distinct contract
             object");
24       this.payer = payer;
25    }
26
27    protected final void payableEntry(Contract payer, int amount) {
28       entry(payer);
29       payer.pay(this, amount);
30    }
```

```
31
32    protected final Contract payer() {
33       return payer;
34    }
35
36    protected final void log(String tag, Object... objects) {
37       logs.add(tag + ": " + Arrays.toString(objects));
38    }
39
40    protected final int balance() {
41       return balance;
42    }
43 }
```

Il campo `balance`, alla riga 2, rappresenta il saldo del contratto e gli si può accedere tramite il metodo, `bilance()` alla riga 40. I trasferimenti di denaro inter-contrattuali sono implementati dal metodo `pay` alla riga 15. Nota che il campo `balance` è gestito e mantenuto sulla blockchain dal meccanismo di serializzazione di Takamaka. Lo stesso accade con il campo `logs`, riga 4, che memorizza un elenco di log popolati tramite il metodo `log(\dots)` alla riga 36. Il metodo `require` può essere utilizzato per verificare condizioni specifiche all'interno di un contratto. Il metodo `entry` alla riga 22, viene richiamato da Takamaka all'invocazione di un metodo `@Entry` di un contratto. Allo stesso modo, Takamaka chiama il metodo `payableEntry` alla riga 27 quando viene invocato un metodo `@Payable @Entry`. Il programmatore non può chiamare esplicitamente tali metodi, il metodo `entry` richiede che il chiamato (`this`) e il chiamante (`payer`) siano distinti oggetti del contratto.

3.7 Le annotazioni @Entry e @Payable

Nella Sezione 3.6 abbiamo visto che i contratti hanno i metodi `entry` e `payableEntry`, questi vengono chiamati quando un contratto chiama rispettivamente un metodo `@Entry` e `@Payable @Entry` di un altro contratto. Questo risulta trasparente al programmatore. Supponiamo che il codice di un contratto `Caller` chiami un metodo `@Entry Callee.m(pars)`. Takamaka riconosce che `m` è annotato come `@Entry` e procede con il modificare tale metodo in `Callee.m(pars, this)` che in questo modo passerà `this` al contratto del chiamante come parametro aggiuntivo del metodo `m` in questione. La stessa trasformazione avviene per le chiamate ai metodi `@Payable @Entry`, in questo caso Takamaka verifica che il primo dei parametri `pars` sia di tipo `int`. Ma vediamo come vengono trasformati i vari metodi, partiamo da un esempio di metodo `@Entry`:

```
public @Entry T m(args) { body }
```

che diventa:

```
public @Entry T m(args, Contract caller) {
   entry(caller);
   body
}
```

In modo simile, per ogni metodo `@Payable @Entry`:

```
public @Payable @Entry T m(int amount, args) { body }
```

che diventa:

```
public @Payable @Entry T m(int amount, args, Contract caller) {
   payableEntry(caller, amount);
   body
}
```

3.8 Gas

Una transazione viene avviata quando un contratto pagante chiama un costruttore pubblico o un qualsiasi metodo di un contratto. Il chiamante deve specificare la quantità di gas da destinare per tale transazione. Takamaka eseguirà il codice del costruttore del contratto, e preleverà denaro dal contratto pagante, sulla base dell'effettivo gas consumato fino alla fine dell'esecuzione. Se tutto il gas viene consumato prima della fine della transazione, viene lanciata l'eccezione `takamaka.lang.OutOfGasException`. Questo meccanismo è implementato dal codice strumentato, vale a dire che prima di ogni istruzione bytecode, Takamaka aggiunge una chiamata al metodo statico `takamaka.lang.Gas.tick(int amount)`, che diminuisce, di `amount`, il gas residuo per la transazione. Se il gas diventa negativo, il metodo `tick`, lancerà l'eccezione `OutOfGasException`. La scelta del valore di `amount` dipende dall'istruzione che viene strumentata, facendo in modo che istruzioni con maggiore costo di esecuzione abbiano anche maggiore costo in gas.

Takamaka non consente di catturare l'eccezione `OutOfGasException`: Takamaka estende ogni tabella delle eccezioni nel del codice con un ulteriore gestore iniziale per l'eccezione `OutOfGasException` che semplicemente rilancia l'eccezione. Questo serve a prevenire attacchi DoS che potrebbero mirare alla cattura dell'Eccezione `OutOfGas` per portare all'esecuzione di un loop infinito al termine del gas.

3.9 Strumentazione e Verifica del Codice

La maggior parte delle funzionalità sono implementate in modo automatico da Takamaka tramite la strumentazione automatica del codice. La persistenza degli oggetti storage, metodi `@Entry` e `@Payable` oltre alla misurazione del gas. Tutto questo può essere eseguito in due modi:

1. Dopo la compilazione, il codice scritto per Takamaka viene strumentato, **staticamente**, utilizzando una libreria di manipolazione del bytecode come `asm` o `bcel`. Questo ha il vantaggio di eseguire la strumentazione una sola volta. Tuttavia, o il client esegue tale strumentazione, oppure un soggetto esterno fornirà il codice strumentato. In quest'ultimo caso, il client deve verificare che i jar memorizzati sulla blockchain siano stati correttamente strumentati, facendo in modo di prevenire imbrogli. Takamaka dovrebbe verificare che le istruzioni siano preceduti da una chiamata `Gas.tick(int amount)` e per il corretto valore `amount`. Nel caso in cui questo controllo fallisse, l'installazione del jar dovrebbe essere rifiutata.

2. Ogni volta che una classe viene caricata tramite un jar in blockchain, quel codice viene dinamicamente strumentato utilizzando le Java instrumentation API. In questo modo un client Takamaka non dovrà fidarsi di una strumentazione

eseguita da un soggetto esterno. Inoltre i jars mantenuti sulla blockchain saranno più piccoli dato che non saranno strumentati. Tuttavia il costo di tale strumentazione dovrà essere pagato per ogni transazione.

La verifica del codice deve comunque essere fatta per entrambe le opzioni. In particolare entrambe le opzioni richiedono a Takamaka di verificare che il jar in questione, utilizzi solamente i metodi indicati come white-listed, sottoinsieme della libreria Java standard.

3.10 Smart Contract Univr e Student

Per la stesura di questo elaborato si è deciso di sviluppare un nuovo contratto d'esempio. Sulla repository *github* di Takamaka sono presenti altri smart contract d'esempio e nel tentativo di svilupparne uno di totalmente nuovo si è pensato di implementarne uno che rappresenti e gestisca le carriere universitarie degli studenti dell'Università di Verona basandoci, a grandi linee, sul funzionamento dell'attuale sistema. Si è scelto di implementare un contratto che gestisca tutte le informazioni dei vari corsi di studio che l'università propone indicando, all'interno di ogni corso di studio, una lista degli esami obbligatori e un'altra che elenca invece tutti i possibili esami a scelta. L'esame non è altro che una classe, `Exam`, che rappresenta tale entità riportandone tutti i campi d'interesse. Ogni oggetto `Exam` contiene al suo interno un identificativo `id`, un nome `name`, un valore che rappresenta i crediti formativi di tale esame `cfu`, un id che identifica il docente di riferimento `professorId`, un valore di valutazione `evaluation` che rappresenta il voto con il quale lo studente ha registrato tale esame ed infine la data, `date` nella quale lo studente ha superato l'esame in questione. Riportiamo di seguito l'implementazione dello smart contract `Univr`:

```
1  package takamaka.univr;
2
3  import takamaka.lang.Contract;
4  import takamaka.lang.Storage;
5  import takamaka.util.StorageMap;
6
7  public class Univr extends Contract {
8     private final StorageMap<Integer, Course> courses = new StorageMap<>();
9
10    public StorageMap<Integer, Exam> getMandatoryExams(int courseId) {
11       return courses.get(courseId).mandatoryExams;
12    }
13
14    public Exam getExtraExam(int courseId, int examId) {
15       return courses.get(courseId).extraExams.get(examId);
16    }
17
18    private class Course extends Storage {
19       private final StorageMap<Integer, Exam> mandatoryExams = new
             StorageMap<>();
20       private final StorageMap<Integer, Exam> extraExams = new StorageMap<>();
21
22       private void insertMandatoryExam(Exam e) {
23          if(mandatoryExams.get(e.getExamId()) == null)
24          mandatoryExams.put(e.getExamId(), e);
25       }
```

```
26
27          private void deleteMandatoryExam(int examId) {
28             if(mandatoryExams.get(examId) != null)
29             mandatoryExams.remove(examId);
30          }
31       }
32    }
```

Notiamo fin da subito la lista dei corsi di studio `courses`, (riga 8), che consiste in una `StorageMap<Integer, Course>` dove il primo valore rappresenta la chiave della `StorageMap` quale l'id del corso, mentre `Course` non è altro che la classe rappresentante il corso in oggetto. Questa ultima classe è stata implementata alla riga 18, completa di due `StorageMap<Integer, Exam>` che rappresentano gli esami obbligatori, `mandatoryExams` (riga 19) e gli esami a scelta per tale corso di laurea, `extraExams` (riga 20). L'elenco degli esami, obbligatori e non, di un certo corso di laurea, sono recuperabili tramite gli appositi metodi `getMandatoryExams` (riga 10) e `getExtraExam` (riga 14). Tali metodi possono essere richiamati da altri smart contract per recuperare delle informazioni utili. Nel nostro caso, l'implementazione del contratto `Student` ha necessità di accedere al contratto `Univr` per recuperare le informazioni corrette del corso di laurea che lo studente x ha scelto di frequentare. Ma vediamo com'è stato implementato il contratto `Student`:

```
1   package takamaka.univr;
2   ...
3   public class Student extends Contract {
4      private final StorageMap<Integer, Career> careers = new StorageMap<>();
5      private final Univr university;
6
7      public Student(Univr university) {
8         this.university = university;
9      }
10
11     public void newCareer(int regNumber, int courseId, int academicYear) {
12        require(payer() == university, "Only univr can add a new carrer");
13        careers.put(regNumber, new Career(regNumber, courseId, academicYear));
14     }
15
16     public void addExam(int regNumber, Exam e) {
17        require(payer() == university, "Only univr can add a new exam");
18        careers.get(regNumber).addExam(e);
19     }
20
21     public void addFee(int regNumber, Fee fee) {
22        require(payer() == university, "Only univr can add a fee");
23        careers.get(regNumber).addFee(fee);
24     }
25
26     public @Payable @Entry void payFee(int amount, int regNumber, int feeId) {
27        require(!careers.get(regNumber).tax.get(feeId).isPaid(), "The fee has
              already been paid");
28        require(balance() >= careers.get(regNumber).tax.get(feeId).getAmount(),
              "Budget not sufficient");
29        require(amount == careers.get(regNumber).tax.get(feeId).getAmount(), "The
              amount of money is incorrect");
30
31        pay(university, amount);
```

```
32          careers.get(regNumber).setPayed(feeId);
33      }
34
35      public void recordEvaluation(int regNumber, int examId, int evaluation, Date
            date) {
36          require(payer() == university, "only univr can register an exam");
37          careers.get(regNumber).recordEvaluation(examId, evaluation, date);
38      }
39      ...
40      private class Career extends Storage {
41          private final int regNumber;
42          private final int courseId;
43          private StorageMap<Integer, Fee> tax = new StorageMap<>();
44          private StorageMap<Integer, Exam> exams = new StorageMap<>();
45          private final int academicYear;
46
47          private Career(int regNumber, int courseId, int academicYear) {
48              this.regNumber = regNumber;
49              this.courseId = courseId;
50              this.academicYear = academicYear;
51
52              exams = university.getMandatoryExams(courseId);
53          }
54
55          private void addExam(Exam e) {
56              exams.put(e.getExamId(), e);
57          }
58
59          private void recordEvaluation(int examId, int evaluation, Date date) {
60              exams.get(examId).recordExam(evaluation, date);
61          }
62
63          private void setPayed(int feeId) {
64              tax.get(feeId).setPayed();
65          }
66          ...
67      }
68  }
```

Come notiamo lo smart contract `Student` contiene una `StorageMap<Integer, Career> careers` (riga 4) che contiene le carriere dello studente ed un oggetto `Univr university` che rappresenta il contratto del beneficiario a cui verranno pagate le rate universitarie, in altre parole, il creatore del contratto in questione (riga 7). Dopo la creazione del contratto `Student` sarà possibile aggiungere una nuova carriera tramite il metodo `newCareer` indicando l'id del corso di laurea a cui lo studente intente iscriversi. Tale metodo potrà essere invocato da chiunque ma soltanto il creatore del contratto porterà a termine l'operazione, vedi riga 11. La creazione di una nuova carriera implica la creazione di una nuova istanza della classe `Career` (riga 40). Il costruttore della classe `Career` procederà quindi con il ricavare l'elenco degli esami obbligatori del corso di laurea scelto aggiungendoli all'elenco degli esami, `exam` della carriera dello studente (riga 52). Il più delle volte la carriera di uno studente è inoltre composta da alcuni esami a scelta che sarà possibile aggiungere ad una carriera tramite il metodo `addExam` (riga 16). La gestione delle tasse universitarie viene gestita all'interno di ogni singola carriera tramite l'oggetto `StorageMap<Integer, Fee> tax` (riga 43) che conterrà tutte le singole rate che lo studente dovrà pagare. Le singole rate possono essere inserite

solamente dal creatore del contratto tramite il metodo `addFee` (riga 21). Sarà possibile procedere con il pagamento delle rate universitarie tramite il metodo `payFee` (riga 26) indicando per primo parametro l'importo seguito dalla matricola dello studente, che identifica la carriera e l'id della tassa che si intende saldare. Dopo l'operazione di pagamento `pay(university, amount)`, procediamo con l'indicare come pagata la rata in questione, invocando il metodo `setPayed(feeId)` (riga 63). Una procedura simile viene eseguita anche per la registrazione di un esame, possiamo infatti richiamare il metodo `recordEvaluation` (riga 35) indicando matricola, identificativo dell'esame, valutazione e la data di superamento di tale esame. Anche questa operazione andrà a buon fine solamente se eseguita dal contratto creatore, `university`.

Questa implementazione vuole essere solamente un esempio e può facilmente essere modificata aggiungendo metodi utili alla gestione delle carriere dello studente. Nell'implementazione completa[2] troviamo altri metodi come `getWeightedAverage` e `getArithmeticAverage` che ritornano rispettivamente la media ponderata e quella aritmetica degli esami superati fino a quel momento.

NB: Come abbiamo visto nella Sezione 3.5, tutte le classi che ci interessa mantenere in modo permanente sulla blockchain dovranno estendere la classe `takamaka.lang.Storage` e nel nostro caso queste classi sono: `Career, Exam, Fee` e `Course`. Notiamo come anche le classi `Exam` e `Fee` estendano `Storage`, questo è dovuto al fatto che le loro istanze sono contenute all'interno di alcune `StorageMap` che fanno anch'esse parte dello stato del contratto. Le istanze della classe `Exam` le troviamo all'interno di `mandatoryExam` (riga 19), `extraExams` (riga 20) del contratto `Univr` e in `exams` (riga 44) del contratto `Student`. Le istanze della classe `Fee` le troviamo solamente all'interno di `tax` (riga 43) del contratto `Student`.

3.11 Conclusioni

In questo capitolo abbiamo descritto un framework che permette ai programmatori di utilizzare un linguaggio ben conosciuto e moderne tecniche di programmazione per sviluppare smart contract per blockchain. Si nasconde la distinzione tra storage e memory objects: il programmatore dovrà estendere la classe `Storage` per avere un oggetto di tipo storage. Ciò che descrive questo capitolo, e in modo più dettagliato sul paper *Takamaka: A Java Framework for Smart Contracts* [18], è completamente diverso da ciò che fino a questo momento si è tentato di fare, come l'utilizzo di Java per interagire con nodi Ethereum, che risulta già possibile con apposite librerie; oppure l'utilizzare Java per scrivere un nodo Ethereum. Questo lavoro, invece, spinge Java all'interno della blockchain, come proprio linguaggio di programmazione. L'implementazione di questo framework richiede una blockchain dotata di primitive per la serializzazione e deserializzazione di storage object. Quindi non può essere direttamente implementato sull'attuale blockchain Ethereum.

[2] Vedi https://github.com/Brunsoft/Tesi-Magistrale/tree/master/src/univr

4

Metodi Java non deterministici

Nel capitolo Takamaka 3, così come in Ethereum 2, abbiamo descritto le transazioni come delle operazioni sullo stato eseguite in modo deterministico. Ciò significa che ogni transazione modifica lo stato globale in modo calcolabile. Non può quindi esistere alcuna forma di entropia o casualità, nelle funzioni white-listed di Takamaka. Abbiamo concluso dicendo che nel caso necessitassimo di una qualsiasi forma di casualità, questa dovrà essere fornita al di fuori della blockchain tramite l'utilizzo di Oracoli. L'esecuzione di uno smart contract dovrà quindi portare al medesimo cambiamento di stato a tutti coloro che lo eseguiranno, dato il contesto della transazione e lo stato della blockchain al momento di tale esecuzione.

Siamo quindi alla ricerca dei metodi Java che, in determinate situazioni, portano ad una forma di non-determinismo. Tali funzioni dovranno essere escluse dai metodi white-listed di Takamaka così da vietarne l'utilizzo durante la scrittura dei contratti. L'analisi comincia con uno dei package più utilizzati: `java.lang`.

4.1 java.lang.Object

`java.lang.Object` è la super classe di tutte la classi Java perciò queste erediteranno tutti i metodi di Object. I metodi che implementa sono numerosi e in certi casi, alcuni di questi, possono causare non determinismo.

4.1.1 Metodo hashCode()

Questo metodo restituisce l'hash code dell'oggetto su cui viene applicato. Ogni volta che viene invocato su di un oggetto, durante l'esecuzione di un'applicazione Java, deve restituire sempre lo stesso valore intero, a condizione che non vengano modificate delle variabili all'interno dell'oggetto stesso. Questo numero intero non deve rimanere coerente da un'esecuzione all'altra. Due oggetti risultano uguali in base al metodo `equals(Object)`, quindi chiamare il metodo `hashCode()` su ciascuno dei due oggetti deve ritornare lo stesso valore. Quindi, in definitiva, il metodo `hashCode()`

restituisce interi distinti per oggetti distinti. Ma vediamo come l'utilizzo di questo metodo su blockchain può riportare problemi:

```
1  public int foo() {
2     return new Object().hashCode();
3  }
```

Come abbiamo detto qui sopra, il metodo `hashCode()`, ritornerà valori uguali a patto che l'oggetto in questione non subisca cambiamenti. Tale valore non deve rimanere costante per ogni esecuzione e tanto meno in virtual machine diverse. Possono essere ritornati dei valori del tipo:

```
1627800613
2065530879
697960108
```

L'utilizzo del metodo visto qui sopra sul sistema blockchain, può portare ad ottenere risultati diversi che possono influire sullo stato globale.

4.1.2 Metodo toString()

Questo metodo ritorna una stringa che rappresenta testualmente l'oggetto, più nel dettaglio viene ritornata una stringa composta dal nome della classe di cui l'oggetto è istanza seguita dal carattere @ e la rappresentazione esadecimale dell'hash code di tale oggetto. In altre parole, viene ritornato:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Il problema può sorgere quando si tenta di implementare un metodo del tipo:

```
1  public String foo() {
2     return new Object().toString();
3  }
```

Come visto nella sezione 4.1.1, il valore della rappresentazione intera dell'hash ritornato dalla funzione `hashCode()`, sarà diverso per ogni virtual machine, o meglio, ad ogni invocazione di `foo()`. Questo implica che ogni qual volta che una transazione utilizzerà il metodo `foo()` verrà ritornata una stringa diversa. Possono infatti essere ritornati dei valori del tipo:

```
java.lang.Object@77468bd9
java.lang.Object@1f89ab83
java.lang.Object@e73f9ac
```

Se questo fosse possibile su blockchain, ogni miner che procede con l'esecuzione tale metodo ne ricava un risultato diverso che può potenzialmente influire sullo stato globale salvato nello storage.

4.2 java.util.HashSet

Questa classe implementa l'interfaccia `Set` e supporta una tabella hash. Non fornisce alcuna garanzia sull'ordine di iterazione del set; in particolare, non garantisce che tale ordine rimanga costante nel tempo. Questa classe offre prestazioni a tempo costante per le operazioni di base quali `add, remove, contains, size`, sul presupposto che la funzione hash disperda gli elementi in modo corretto tra bucket. L'iterazione su

questo set richiede un tempo proporzionale alla somma delle dimensioni dell'istanza di `HashSet` (numero degli elementi), sommata alla capacità di supporto dell'istanza `HashMap` (numero di bucket). Pertanto sarà importante non impostare una capacità iniziale troppo alta se si dà molta importanza alle prestazioni. Da notare che questa implementazione non è sincronizzata. Se più thread accedono contemporaneamente ad un hash, e almeno uno di questi thread modifica il set, si deve procedere con una sincronizzazione esterna. Il set dovrebbe essere incapsulato all'interno di un metodo `Collections.synchronizedSet` ed è consigliabile farlo già alla creazione dell'`HashSet` in oggetto.

Gli iteratori restituiti dal metodo `iterator` di questa classe sono fail-fast: se il set viene modificato in qualsiasi momento dopo la creazione dell'iteratore, quest'ultimo genera l'eccezione `ConcurrentModificationException`. Quindi di fronte a modifiche simultanee, l'iteratore fallisce rapidamente ed in modo pulito, piuttosto che rischiare di comportarsi in modo arbitrario e non deterministico in qualche tempo indeterminato nel futuro.

4.2.1 Metodo iterator()

Restituisce un iteratore sugli elementi di questo set. Gli elementi vengono restituiti in nessun ordine particolare. Come detto in precedenza, la classe `HashSet` non garantisce che l'ordine con cui vengono ritornati gli elementi rimanga costante.

```
1  private HashSet h = new HashSet();
2  ...
3  public Iterator getIterator() {
4     return h.iterator();
5  }
```

Supponiamo quindi di scrivere uno smart contract con all'interno il metodo `getIterator()` riportato qui sopra, che ci da la possibilità di ritornare un iteratore dell'`HashSet h`. Se l'ordine di iterazione non è garantito rimanere costante nel tempo, ma bensì possiamo trovarci difronte a due iteratori, ritornati dalla funzione di contratto `getIterator()`, che iterano con un diverso ordine sugli elementi di `h`, può portare al non determinismo su blockchain. Una cosa è certa, l'iteratore ritornato può comunque spaziare su tutti gi elementi di `h` anche se in ordine diverso, ma partendo da due iteratori che propongono due ordinamenti diversi dello stesso hashset è possibile eseguire delle operazioni che ci portano a dei risultati diversi. Se tale risultato va ad influire sullo stato globale possiamo dedurre che non possiamo permettere l'utilizzo incontrollato di tale Classe.

4.3 java.util.HashMap

L'implementazione è basata su HashTable dell'interfaccia `Map`. Questa implementazione fornisce tutte le operazioni opzionali delle map, oltre a permettere valori `null` e chiavi `null`. Questa classe non fornisce garanzie sull'ordine della mappa, in particolare, non garantisce che tale ordine rimanga costante nel tempo. Questa implementazione fornisce prestazioni costanti per operazioni di base (`get` e `put`), assumendo che la funzione hash disperda gli elementi in modo corretto tra i vari bucket.

4.3.1 Metodo values()

Ritorna una `Collection` view di tutti i valori contenuti nella map. Le modifiche della mappa si riflettono sulla `Collection` e viceversa.

```
1  private static Map<String, String> h = new HashMap<>();
2  ...
3  public Iterator getIterator() {
4     return h.values().iterator();
5  }
```

Se un contratto implementa il metodo `getIterator()` questo ritornerà un iteratore della Collezione che rappresenta i dati contenuti nella nostra hashMap `h`. Purtroppo, come riportato nei java docs, la classe `HashMap`, così come `HashSet`, non fornisce garanzie sull'ordine delle map, ma soprattutto, non è garantito che l'ordine rimanga costante nel tempo. Per questo motivo può portare al non determinismo su blockchain.

4.4 java.util.stream

Una `Stream` è una sequenza di elementi che supporta operazioni di aggregazione sequenziali e parallele. Oltre a `Stream` che è un flusso di riferimenti ad oggetti, esistono delle specializzazioni primitive per `IntStream`, `LongStream` e `DoubleStream`. Per eseguire un calcolo, le operazioni di flusso sono composte in una *stream pipeline*. Una stream pipeline è costituita da una *sorgente* (che potrebbe essere un'array, una `Collection`, una funzione di generazione, un canale I/O, ecc.), zero o più *operazioni intermedie* (che trasformano lo stream in un altro stream, come per esempio `filter(Predicate)`) ed infine, un'operazione *terminale* (che produce un risultato o un side-effect, come per esempio `count()` o `forEach(Consumer)`). Gli stream sono "pigri" cioè i calcoli sui dati d'origine vengono eseguiti solamente quando viene avviata l'operazione terminale. `Collection` e `Stream`, pur presentando alcune somiglianze superficiali, hanno obiettivi diversi. Le `Collection` hanno l'obiettivo principale di fornire l'accesso ed una gestione efficiente ai dati che contengono. Al contrario, gli `Stream` non forniscono un mezzo per accedere direttamente o manipolare i loro elementi, sono invece interessati alla descrizione dichiarativa della sorgente e delle operazioni di calcolo che saranno eseguite in forma aggregata sulla sorgente. Una pipeline di flusso può essere vista come una query sull'origine dello stream.

Andiamo ora a vedere più da vicino come viene utilizzato uno stream e che influenza può avere sullo stile di programmazione Java. Come visto nell'articolo [19], data la classe `Album` cerchiamo di ricavare tutti gli album pubblicati prima dell'anno 2000 e stamparne i relativi autori. Vediamo la versione scritta in Java 7 e la possibile scrittura in Java 8 con l'utilizzo delle stream.

```
1  // Implementazione in Java 7
2  for (Album album : albums)
3     if (album.getYear() < 2000)
4        System.out.println(album.getAuthor());
5
6  // Implementazione in Java 8
7  albums.stream()
8     .filter(album -> album.getYear() < 2000)
9     .map(Album::getAuthor)
10    .forEach(System.out::println);
```

Nel primo caso, prendiamo il primo elemento, questo viene processato per poi passare al successivo. Questo tipo di operazione si chiama iterazione esterna. Nel secondo caso ci facciamo restituire un oggetto stream dalla collezione `albums`, sul quale applichiamo il metodo `filter` che ci permette di considerare solamente gli album che soddisfano un certo criterio, tale criterio viene espresso tramite una lambda espressione. Sugli album che superano questa selezione, tramite `map` e con un metodo reference, ci facciamo ritornare solamente l'autore e procediamo con lo stampare ciascuno di questi elementi. Riguardando le due implementazioni qui sopra appare evidente che lavoriamo dicendo allo stream cosa fare, non come farlo, di conseguenza ciò che abbiamo scritto è più simile alla descrizione del problema formulato in partenza, piuttosto che ad una metodologia per risolverlo.

Quella appena descritta è la modalità sequenziale ma da la possibilità di eseguire le stesse operazioni in parallelo. Dal punto di vista sintattico basta sostituire `stream()` con il costrutto `parallelStream()` in questo modo:

```
1  albums.parallelStream()
2     .filter(album -> album.getYear() < 2000)
3     .map(Album::getAuthor)
4     .forEach(System.out::println);
```

Questa nuova soluzione risponde comunque al problema formulato ma il risultato da lei ritornato non è deterministico in quanto l'ordinamento ritornato da questa procedura è potenzialmente diverso ad ogni esecuzione.

4.4.1 Metodo findAny()

Restituisce un oggetto `Optional` che descrive qualche elemento dello stream, oppure un empty `Optional` in caso di stream vuoto. È un'operazione terminale ed il comportamento di questa operazione è esplicitamente non deterministico, infatti è libera di selezionare un qualsiasi elemento nello stream. Questo per consentire prestazioni massime nelle operazioni parallele, con la conseguenza che più invocazioni successive della stessa procedura sulla medesima fonte potrebbero non restituire lo stesso risultato. Il metodo alternativo stabile è `findFirst()`.

```
1  private String foo() {
2     List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
          "Dany", "Julia", "Divya");
3
4     return lst.parallelStream()
5        .filter(s -> s.startsWith("J"))
6        .findAny().get();
7  }
```

L'esecuzione consecutiva della funzione sopraccitata `foo()` restituisce risultati diversi. Ad ogni esecuzione di tale metodo ci possiamo aspettare risultati diversi. Riportiamo qui sotto i risultati rappresentati con il formato `nome : n-returns`.

```
Jill: 7060, Jack: 2319, Julia: 91, Jhonny: 179, Jenish: 351
```

Sostituendo `findAny()` con il metodo `findFirst()`, come riportato sulla descrizione JavaDoc, ci viene ritornato sempre il medesimo risultato, nel nostro caso `Jhonny: 10000`. Abbiamo quindi la conferma che `findAny()`, utilizzato soprattutto con `parallelStream`, risulta un metodo non deterministico, quindi non utilizzabile all'interno di smart contract blockchain.

4.4.2 Metodo forEach(Consumer<? super T> action)

Questo metodo esegue un'azione per ciascun elemento dello stream. Il comportamento di questa operazione è esplicitamente non deterministico. Per le pipeline delle parallelStream questa operazione non garantisce il rispetto dell'ordine con cui si presentano inizialmente gli elementi dello stream. Il rispetto di tale ordinamento comprometterebbe i benefici introdotti dall'analisi parallela. Le analisi su ogni elemento possono pertanto essere eseguite in qualsiasi momento e da qualsiasi thread che si libera e sceglie l'elemento di cui prendersi cura. Se l'operazione eseguita accede in modifica allo stato condiviso, dovrà provvedere a fornire l'adeguata sincronizzazione.

Quella appena riportata è la descrizione che Oracle da al metodo `forEach` sulla pagina che presenta le stream [15]. Tale documentazione esprime chiaramente il non determinismo del sopracitato metodo anche se applicato a stream sequenziali. I svariati tentativi di trovare degli esempi di funzionamento non deterministi sono purtroppo falliti, le varie funzioni che facevano uso delle stream su cui applicavano il metodo `forEach`, nel nostro caso, hanno restituito sempre gli elementi nell'ordine in cui sono stati sottoposti, ma l'affermazione *"Il comportamento di questa operazione è esplicitamente non deterministico"* risulta molto chiara e ciò non significa che non si comporterà mai in modo deterministico ma bensì che tale metodo non è obbligato ad essere deterministico. Per questo motivo, il non riuscire a trovare un esempio di funzionamento anomalo che provi il non determinismo, non può voler dire che tale metodo ritorni sempre i risultati aspettati, ma bensì che non possiamo contare su di esso, specialmente se tale metodo verrà utilizzato all'interno di applicazioni che devono obbligatoriamente essere deterministiche. Possiamo quindi concludere che tale metodo non può essere incluso nei metodi white listed di *takamaka*.

4.4.3 Metodo forEachOrdered(Consumer<? super T> action)

Questo metodo esegue un'azione per ciascun elemento dello stream, in base all'ordine di incontro se lo stream ne ha definito uno. Questa operazione è terminale ed elabora gli elementi uno alla volta, nell'ordine di incontro. L'esecuzione dell'azione su di un elemento avviene prima di eseguire la stessa sull'elemento successivo, ma per ogni dato elemento l'azione può essere eseguita da qualsiasi thread che si libera e sceglie.

Diversamente dal metodo `forEach`, Sezione 4.4.2, questa implementazione ritorna gli elementi nello stesso ordine con cui si presentavano all'interno della sorgente di partenza. `forEachOrdered` processa quindi gli elementi nello stesso ordine con cui si presentano all'interno nella sorgente iniziale indipendentemente dal fatto che lo stream sia di tipo sequenziale o parallelo. Ma vediamone gli effetti applicati ad un'operazione di ricerca stringhe parallela:

```
1  private void foo() {
2     List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
          "Dany", "Julia", "Jenish", "Divya");
3
4     lst.parallelStream()
5        .filter(s -> s.startsWith("J"))
6        .forEach(e -> System.out.print(e + " "));
7
8     lst.parallelStream()
```

```
9         .filter(s -> s.startsWith("J"))
10        .forEachOrdered(e -> System.out.print(e + " "));
11    }
```

L'esecuzione del metodo qui descritto `foo()` può generare il seguente output:

```
Jill Jack Jenish Jhonny Julia
Jhonny Jack Jill Julia Jenish
```

Confrontando la lista `lst` di partenza possiamo notare che gli elementi filtrati tramite `filter(s -> s.startsWith("J"))`, nel primo caso vengono stampati in modo casuale, in base all'ordine con cui le varie thread hanno terminato l'analisi dei singoli elementi. Nel secondo caso notiamo invece che l'ordine con cui vengono ritornati i vari elementi risultato combacia con l'ordine in cui compaiono tali elementi nella lista sorgente.

Il metodo `forEachOrdered`, diversamente da `forEach`, garantisce che le lambda espressioni contenute al suo interno non vengano eseguite in contemporanea da più thread, oltre a dare la certezza che gli elementi vengano ritornati nello stesso ordine con cui sono stati sottoposti allo stream, tale metodo può quindi sostituire `forEach` assicurando così un output deterministico. La conferma di ciò la possiamo vedere nel seguente esempio:

```
1   private void foo() {
2      List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
           "Dany", "Julia", "Jenish", "Divya");
3
4      List<String> lst1 = new ArrayList<String>();
5      lst2.parallelStream()
6         .filter(s -> s.startsWith("J"))
7         .forEach(e -> lst1.add(e));
8
9      List<String> lst2 = new ArrayList<String>();
10     lst2.parallelStream()
11        .filter(s -> s.startsWith("J"))
12        .forEachOrdered(e -> lst2.add(e));
13  }
```

Il metodo qui descritto ricerca con due procedure separate gli elementi all'interno della lista `lst` che iniziano con il carattere *J*, tali elementi verranno salvati tramite le due versioni di `forEach` all'interno di due diverse liste, `lst1` e `lst2`. In particolare `lst1` potrà anche non contenere tutti i risultati, questo dato dal fatto che `forEach` viene eseguito in modo parallelo e, dato che ArrayList non è thread safe, se due thread tentano di salvare nello stesso momento due elementi nella medesima posizione, uno dei due risultati verrà perso. `lst2` invece, conterrà sempre gli elementi `[Jhonny, Jack, Jill, Julia, Jenish]` che appaiono nello stesso ordine con cui comparivano nella lista sorgente `lst`.

Possiamo sostituire `forEach` con la la versione ordered sia con `stream()` che con `parallelStream()` ma, nel primo caso l'applicazione del `forEachOrdered` implica uno sforzo computazionale di poco superiore al costo del `forEach`. Nel secondo caso, se applicassimo `forEachOrdered`, imponendo, oltre al rispetto dell'ordinamento nativo della sorgente, anche che tutte le operazioni sugli elementi vengano eseguite in modo seriale, comprometteremo i benefici introdotti dall'analisi parallela.

Concludendo, possiamo quindi affermare che il metodo `forEachOrdered` si comporta sempre in modo deterministico dandoci la possibilità di eseguire delle opera-

zioni parallele in modo deterministico ma compromettendone le altrimenti elevate prestazioni.

4.4.4 Metodo peek(Consumer<? super T> action)

Ritorna uno stream costituito dagli elementi dello stream, eseguendo inoltre l'azione indicata su ogni elemento mano a mano che tali elementi vengono consumati dal flusso risultante. `peek` risulta un'operazione intermedia. Se richiamata su di un `parallelStream()` può essere eseguita in qualsiasi momento e da qualsiasi thread. Se l'operazione eseguita da `peek` modifica lo stato condiviso si deve procedere con il fornire una adeguata sincronizzazione.

Come si evince dalla documentazione, `peek` è un'operazione intermedia, vale a dire che non termina l'elaborazione del flusso, diversamente dalle operazioni esaminate nei capitoli precedenti, quali `forEach` e `forEachOrdered`. Questo metodo ci permette quindi di utilizzare un flusso senza terminare la pipeline delle operazioni agendo sui contenuti dello stream. Inoltre il metodo `peek` è un'operazione che non interferisce, cioè garantisce che i dati dello stream sorgente non vengano modificati durante la sua esecuzione. L'obiettivo del metodo `peek` è letteralmente quello di dare un'occhiata al contenuto dello stream. È stato infatti inserito all'interno delle API della classe `Stream` per fornire un metodo per eseguire delle operazioni di debug. Questo metodo è perciò unico nel suo genere. Se eseguito in modo seriale, tramite `stream()` analizza gli elementi uno ad uno e il suo comportamento sembra deterministico. Delle problematiche, come nel caso del metodo `forEach`, sopraggiungono quando applichiamo `peek` ad uno stream parallelo il cui ordine d'esecuzione cambia per ogni JVM. Il metodo `peek` esegue delle lambda espressioni che possono, per esempio, salvare gli elementi dello stream all'interno di liste, più o meno thread safe. In definitiva questo metodo può essere molto utile sia per il debug che per eseguire analisi sui dati dello stream senza modificarne la struttura. Funziona in modo deterministico se applicato a stream seriali ma, se utilizzato su `parallelStream()` il suo comportamento non è deterministico dato che l'ordine con cui analizzerà gli elementi può cambiare ad ogni esecuzione. Pertanto possiamo scegliere di accettarne l'utilizzo solamente se applicata ad una sorgente stream seriale.

4.4.5 Altri esempi di non determinismo

Nel JavaDoc che descrive il package `java.util.stream` viene scritto che ad eccezione delle operazioni identificate come esplicitamente non deterministiche, come i metodi `findAny` e `forEach`, se un flusso viene eseguito in sequenza o in parallelo non dovrebbe modificare il risultato del calcolo. Con un po' di fantasia però si può pensare ad una procedura non deterministica che stampi risultati diversi se eseguita in modo seriale piuttosto che parallelo.

```
1  private List<String> foo() {
2     List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
          "Dany", "Julia", "Divya");
3
4     List<String> results = new ArrayList<>();
5     lst.parallelStream()
6        .filter(e -> e.startsWith("J"))
7        .forEach(e -> results.add(e));
```

```
8
9       return results;
10    }
```

La versione seriale, con l'utilizzo di `stream()`, e quella parallela, qui sopra implementata, che utilizza `parallelStream()` producono due risultati ben diversi. Quando utilizziamo uno stream parallelo, dobbiamo tenere sempre presente che introduciamo il non determinismo, come nella funzione `foo()`, se cerchiamo in parallelo un elemento che abbia certe caratteristiche all'interno dello stream, e ce n'è più di uno, potremmo ottenere risposte diverse in esecuzioni diverse. Le righe precedenti, infatti, in maniera non prevedibile producono un'eccezione, dato che ArrayList non è thread-safe. L'arraylist `results` potrà potenzialmente contenere i seguenti risultati:

```
[Jill, Jack, Julia, Jenish, Jhonny]
[Jill, Jack, Julia, Jhonny, Jenish]
[Jill, Jenish, Jack, Julia]
...
```

Una prima soluzione a questo problema può essere l'introduzione di una lista sincronizzata (thread-safe) che si prenda cura di garantire l'accesso in modo seriale alla struttura specificata. In questo modo se più thread tenteranno di accedere nello stesso momento alla lista, per aggiungere un nuovo elemento, solo una delle due thread riuscirà a salvare subito il nuovo dato, mentre l'altra resterà in attesa che la risorsa venga rilasciata. Nelle seguenti righe vediamo tale implementazione:

```
1   private List<String> foo() {
2     List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
          "Dany", "Julia", "Divya");
3
4     List<String> results = Collections.synchronizedList(new ArrayList<>());
5     lst.parallelStream()
6        .filter(e -> e.startsWith("J"))
7        .forEach(e -> results.add(e));
8
9     return results;
10  }
```

La nuova lista `result` ci assicura che tutti gli elementi trovati dalla procedura, in questo caso che tutti i nomi presenti nella lista `lst` che iniziano con la lettere J, vengano riportati all'interno della lista `results`. Questo modo di procedere però non assicura che l'ordinamento di `results` resti fisso per tutte esecuzioni. L'accesso in scrittura a tale lista resta comunque legato all'ordine di terminazione delle varie thread che può variare ad ogni esecuzione. Questo ci porta a scartare anche questa soluzione. Vediamo alcuni potenziali output:

```
[Jill, Jack, Julia, Jenish, Jhonny]
[Jill, Jenish, Julia, Jhonny, Jack]
[Jill, Jhonny, Jack, Julia, Jenish]
...
```

Nelle seguenti righe vediamo un'altra variante che rende costante l'ordinamento di `results`:

```
1   private List<String> foo() {
2     List<String> lst = Arrays.asList("Jhonny", "David", "Jack", "Duke", "Jill",
          "Dany", "Julia", "Divya");
```

```
3
4       List<String> results = lst2.parallelStream()
5          .filter(e -> e.startsWith("J"))
6          .collect(Collectors.toList());
7
8    return results;
9    }
```

In un qualsiasi stream, parallelo o sequenziale, le operazioni contenute nella pipeline non devono modificare la sorgente dati, ad eccezione di alcune operazioni intermedie, inoltre le lambda che vengono passate, non devono dipendere dallo stato di qualche altro oggetto anche se esterno alla pipeline, devono quindi essere stateless. Se questi accorgimenti non vengono rispettati possiamo ottenere risultati non corretti o imbatterci in eccezioni a runtime.

4.4.6 Side-effects

Gli side-effects, o effetti collaterali, sono il risultato di quelle operazioni, all'interno degli stream, che modificano lo stato un qualche oggetto. Rappresenta tutte quelle modifiche dei valori dei campi di istanza o di classe eseguiti oltre al calcolo/restituzione di un valore. Ovviamente, un metodo che non ha un valore di ritorno deve avere una sorta di effetto collaterale che lo giustifichi. Nelle sezioni precedenti abbiamo appunto esaminato alcuni metodi della classe `Stream` che producono side-effects come `forEach`, `forEachOrdered` e `peek`. Questi metodi non avrebbero senso di esistere se non producessero side-effects, dato che, a parte `peek`, non restituiscono alcun risultato. Si deve prestare molta attenzione quando si fa utilizzo di queste istruzioni, soprattutto in presenza di `parallelStream()` è perciò consigliabile, dove del codice contiene inutilmente delle istruzioni che generano side-effects, di rimpiazzare queste istruzioni con altre operazioni di riduzione sicure. Possiamo vedere un esempio di tutto ciò nella sezione 4.4.5, dove si è preferito l'utilizzo dell'istruzione terminale `.collect(Collection.toList())` che restituisce una lista di risultati, all'utilizzo di `.forEach(s -> result.add(s))` che inseriva gli elementi risultato all'interno di una lista precedentemente dichiarata. In questo modo, evitando gli side-effects, la funzione si comporterà in modo deterministico, ritornando sempre il risultato atteso.

4.5 java.util.Random

Un'istanza di questa classe può essere utilizzata per generare uno stream di numeri pseudocasuali a partire da un seed di 48-bit. Tale seed viene modificato ad ogni invocazione con l'utilizzo della formula *LCF* (Linear Congruential Formula) ed utilizzato per il calcolo del successivo numero pseudocasuale.

$$X_{n+1} = (aX_n + c) \mod m$$

dove il modulo $m > 0$, il moltiplicatore $0 < a < m$, l'incremento $0 \leq c < m$ e il valore di partenza X_0 rappresenta il seed e deve essere compreso tra $0 \leq X_0 < m$. Se due istanze di Random vengono create con lo stesso seed, queste produrranno la medesima sequenza di numeri pseudocasuali. Uno dei costruttori `Random()` si prende cura di settare un valore di seed che, con molta probabilità, sarà distinto da qualsiasi

altra chiamata. Il secondo costruttore `Random(int seed)` crea anch'esso un nuovo generatore di numeri pseudo casuali ma settando il valore di seed, che influenzerà i valori generati, con il valore passato.

```
1  private int rnd() {
2     return new Random().nextInt();
3  }
4
5  private int rnd1() {
6     return new Random(1993).nextInt();
7  }
```

Il valore impostato dal costruttore vuoto `Random()` non è prevedibile perché calcolato con il seguente calcolo:

```
public Random() {
   this(seedUniquifier() ^ System.nanoTime());
}

public Random(long seed) {
   if (getClass() == Random.class)
      this.seed = new AtomicLong(initialScramble(seed));
   else {
      // subclass might have overriden setSeed
      this.seed = new AtomicLong();
      setSeed(seed);
   }
}
```

Questo genererà un valore di seed diverso su ogni macchina che eseguirà il metodo `rnd()` con la conseguente generazione di una sequenza di numeri pseudocasuali totalmente diversa. Tuttavia procedendo con la creazione di un'istanza della classe Random tramite il costruttore `Random(int seed)`, fissato un `int seed`, non accade tutto ciò, ma bensì ogni volta che si invocherà tale costruttore con il medesimo valore di `seed`, verrà generata sempre la stessa sequenza di numeri pseudocasuali su qualsiasi JVM. Con l'unica restrizione che si utilizzi la stessa funzione LCF. Non rappresenta un'implementazione di una funzione Random nel vero senso della parola, ma se si avesse bisogno di una specifica sequenza di numeri pseudocasuali, per una determinata funzione, la si può utilizzare. Un esempio di questa implementazione può essere `rnd1()`, questo metodo ritorna semplicemente il valore restituito da `nextInt()` che risulta essere sempre `-1626801979`. Possiamo anche ritornare l'oggetto `new Random(1993)`, anch'esso su qualsiasi altra JVM, applicando in modo consecutivo `nextInt()`, si comporterebbe in modo deterministico restituendo sempre la sequenza:

-1626801979, 1700409766, -290245901, -226742933, 1081295581, ...

Le modalità con cui viene calcolato tale sequenza di interi sono descritte dal metodo invocato dal `nextInt()` ovvero `next(32)` all'interno della classe `Random.java`. Concludiamo affermando che un'istanza di `Random(int seed)` con `seed` fissato, si comporta in modo deterministico su tutte le JVM, quindi si presta all'utilizzo per la programmazione della blockchain.

4.6 java.lang.System

La classe `System` contiene una vasta gamma di campi e metodi utili e non può essere istanziato.

4.6.1 Metodo currentTimeMillis()

Questo metodo ritorna un `long` che rappresenta il tempo corrente misurando i millisecondi trascorsi dalla mezzanotte del 01/01/1970 UTC. Questo metodo è utilizzato da una vasta gamma di classi tra cui `java.util.Date`, `java.time.Clock` e `java.util.Timer`. Supponiamo di implementare la seguente funzione:

```
1  private long getCurrentMillis() {
2     return System.currentTimeMillis();
3  }
```

Ogni JVM che eseguirà `getCurrentMillis()` riceverà un risultato che varia in base all'istante in cui viene eseguito. I miner della network blockchain hanno tutti un'ordine potenzialmente diverso di esecuzione delle transazioni che vengono analizzate per ordine di arrivo e in base a delle logiche di guadagno, `fee` maggiore $\Leftrightarrow$ priorità maggiore. Per questo motivo i miner possono analizzare una transazione in momenti diversi e nel caso una transazione richiamasse la funzione `getCurrentMillis()`, ogni JVM può potenzialmente ricavarne valori diversi. Le soluzioni apportate, per esempio, in Ethereum sono rappresentate dall'utilizzo di oracoli esterni. Ma in questo caso abbiamo bisogno di un dato istantaneo e l'utilizzo di tali oracoli non costituirebbe comunque una soluzione dato che, come visto nella sezione 2.6, tale dato dovrebbe prima essere trasferito sulla catena rendendo i dati disponibili in memoria all'interno di uno smart contract. Ma questi passaggi non sono istantanei e necessitano di un certo tempo per essere eseguiti, perdendo la proprietà di freshness di tale dato. Un'alternativa adottata da Solidity è quella di utilizzare alcune proprietà del blocco corrente per ricavare un timestamp utile, come per esempio `now`, alias di `block.timestamp` che restituisce il tempo trascorso tra la creazione del blocco corrente e il 01/01/1970 00:00:00 UTC, restituita in secondi. Nel nostro caso non possiamo comunque accettare l'utilizzo del metodo `System.currentTimeMillis()` e di tutti gli altri metodi che ne fanno uso.

4.6.2 Metodo System.nanoTime()

Questo metodo ritorna un valore temporale di esecuzione della JVM misurata in nanosecondi, può essere utilizzato solo per misurare il tempo che trascorso dato che non è in alcun modo correlato a nessuna sorgente temporale fornita dal sistema. Questo metodo lo abbiamo incontrato in precedenza quando stavamo analizzando la classe `Random` nella sezione 4.5, veniva infatti utilizzato per inizializzare il costruttore `Random(long Seed)` nel caso di invocazione del costruttore vuoto `Random()`. L'utilizzo del metodo `nanoTime()`, è in quel caso, scusato dal fatto che è fonte di una buona casualità, utile infatti per trovare un numero abbastanza casuale. Ma vediamo come potrebbe comparire all'interno di uno smart contract:

```
1  private long getNanoTime() {
2     return System.nanoTime();
3  }
```

Se un qualunque contratto richiamasse il metodo `getNanoTime()` ogni JVM ne ricaverebbe un valore diverso con una buona fonte di casualità dato che è improbabile che due JVM siano attive dallo stesso lasso di tempo ed eseguano la stessa richiesta nel medesimo momento. È proprio questa fonte di casualità che ci porta a scartare tale metodo dall'utilizzo su blockchain, arrivando quindi alla medesima conclusione a cui siamo arrivati nella sezione precedente, per tale fonte di casualità, scartavamo la possibilità di poter utilizzare il costruttore vuoto della classe `Random`. Possiamo quindi scartare l'utilizzo del metodo `System.nanoTime()` e di tutte la classi che ne fanno uso.

Conclusioni

Nel capitolo 1 di questo elaborato si sono analizzate le caratteristiche tecniche che permettono il funzionamento di Bitcoin, in particolare della blockchain, un registro distribuito il cui accesso è aperto a chiunque, lo scopo di tale registro è quello di mantenere traccia di tutte le transazioni risolvendo, senza la necessità di un'autorità centrale, il problema del double-spending. Il fatto che la blockchain sia di pubblico dominio implica l'impossibilità di una sua manomissione, rafforzata dal fatto che la blockchain risulta di fatto un'ordinata lista di blocchi contenenti transazioni back-linked[1], questi aspetti rappresentano sicuramente gli elementi più innovativi di tale sistema. Questo nuovo approccio portato da Bitcoin sembra in grado di rivoluzionare tutti i sistemi di gestione centralizzata a cui siamo abituati. Abbiamo inoltre analizzato il funzionamento della *proof-of-work* con particolare attenzione alla funzionalità di *retargeting* implementata da Bitcoin che mantiene costante l'emissione di valuta aumentando la difficoltà dell'algoritmo di *proof-of-work* in base all'ormai veloce aumento della potenza di calcolo degli elaboratori che prendono parte alla rete. È stato inoltre analizzato il processo di fork della blockchain, possibilità non molto remota dovuta al fatto che più miner possono arrivare quasi simultaneamente ad una soluzione dell'algoritmo di *proof-of-work* che causa la creazione di n figli del penultimo blocco. Questo fatto implica la creazione di n versioni di questo libro mastro sul quale lavoreranno n porzioni della rete globale. Da questo punto in poi il primo miner che riuscirà ad inserire un ulteriore blocco all'apice di una delle n chain renderà tale chain la catena principale sulla quale tutta la rete dovrà lavorare da questo momento in poi. Abbiamo visto che è su questo funzionamento che si basa l'attacco al consenso chiamato *Attacco del 51%*. Questa tipologia di attacco può essere attuata da un gruppo di miner, appunto almeno il 51% della rete, che controllano quindi il 51% della potenza hashing della network mondiale. Abbiamo visto che questa tipologia di attacco può causare deliberatamente dei fork, causare double-spending o eseguire attacchi di denial-of-service contro specifiche transazioni/indirizzi. La percentuale effettivamente richiesta per portare a termine questa tipologia di attacco, può anche essere inferiore al 51% della potenza di hashing, tale soglia rappresenta solamente il livello al quale tale attacco è quasi sicuramente possibile. Tale limite sembra infatti essere possibile anche con solo il 30% della potenza di hashing mondiale.

Nel capitolo 2 abbiamo analizzato l'alt-chain Ethereum il cui vero scopo non è quello di realizzare un nuovo sistema monetario ma la sua blockchain include anche una valuta che viene emessa come token per allocare dei contratti. Ethereum risulta quindi un'infrastruttura open-source, globalmente decentralizzata, sviluppata per eseguire contratti denominati *smart contract*. Tutti questi contratti e il relativo stato

[1] I vari blocchi sono collegati a ritroso, ognuno si riferisce al precedente blocco presente nella catena.

del sistema risiedono sulla blockchain che è di fatto utilizzata per sincronizzare e tenere in memoria le modifiche di questo stato assieme alla cripto-valuta *ether*, anche utilizzata per limitare l'esecuzione di questi contratti, tramite un valore espresso in unità di *gas*. Ethereum, a differenza di Bitcoin, che presenta un linguaggio di scripting molto limitato, è stato concepito per essere una blockchain programmabile i cui contratti vengono eseguiti da una macchina virtuale *EVM* in grado di eseguire codice di complessità arbitraria. Bitcoin è stato volutamente vincolato a delle semplici valutazioni true/false delle varie condizioni di spesa mentre il linguaggio Ethereum risulta turing-completo. Questo fatto porta con se una conseguente problematica cioè il fatto che possa eseguire qualsiasi programma di qualunque complessità. Questa capacità porta con se alcuni problemi dal punto di vista della sicurezza e della gestione delle risorse disponibili. Turing ha infatti dimostrato come non sia possibile prevedere se un dato programma terminerà o meno senza prima eseguirlo. Se per caso o di proposito fosse possibile sottoporre ad Ethereum uno smart contract che una volta lanciato non giunga ad un termine, si verificherebbe un attacco di tipo DoS. Abbiamo quindi visto come Ethereum risolve questa problematica introducendo un meccanismo di misurazione delle risorse chiamato *gas*. Ogni istruzione ha un proprio costo predeterminato in *gas* e l'esecuzione di una sequenza di istruzioni consumerà una certa quantità di *gas*. Quando una transazione attiva l'esecuzione di un contratto deve includere una quantità di *gas* che rappresenterà il limite superiore di ciò che può essere consumato per l'esecuzione di tale contratto. Abbiamo poi definito il termine smart contract, non definendoli tramite la traduzione letterale "contratti intelligenti" ma bensì descrivendoli come contratti che si riferiscono a programmi informatici immutabili che funzionano in modo deterministico in un contesto quale la macchina virtuale EVM e sul computer decentralizzato Ethereum. In questa affermazione troviamo la parola chiave *deterministico*. L'esito di uno smart contract dovrà essere sempre lo stesso per tutti coloro che lo eseguono, dato il contesto della transazione e lo stato dalla blockchain Ethereum al momento dell'esecuzione. È stato poi analizzato il linguaggio di programmazione contract-oriented che prende il nome di Solidity, creato ad Gavin Wood, il cui principale prodotto risulta il compilatore *solc* che converte appunto programmi scritti in Solidity in linguaggio bytecode per l'EVM. Di questo linguaggio vengono rilasciate nuove versioni in poco tempo, in questo momento vengono rilasciate delle patch, quattro volte a settimana. Le versioni di Solidity seguono il modello semantic versioning il quale specifica la versione strutturandola in tre parti numeriche: $MAJOR : MINOR : PATCH$. Una delle caratteristiche principali degli smart contract Ethereum è la loro capacità di richiamare e quindi riutilizzare codice all'interno di altri contratti. A volte però si potrebbe avere la necessità di accedere a delle informazioni off-chain, come per esempio informazioni sul meteo piuttosto che sull'andamento finanziario di qualche bene. È proprio in questo frangente che ci vengono incontro gli oracoli che sono di fatto in grado di fornire fonti di dati esterne agli smart contract Ethereum. Idealmente questi sono sistemi *trustless* nel senso che sono sistemi decentralizzati che non hanno bisogno di essere fidati. Dal momento che non può esistere alcuna fonte di casualità per l'EVM e per gli smart contract, tali valori dovranno essere introdotti tramite l'utilizzo di un oracolo che procederà con la creazione di una transazione la quale salverà tali informazioni casuali sulla blockchain. Solo in questo momento si potrà procedere con l'utilizzo di tali dati per i propri fini. Se così non fosse, l'esistenza di una funzione

casuale all'interno della blockchain, potrebbe portare ad effetti disastrosi portando due differenti nodi che eseguono lo stesso contratto ad concludere risultati differenti nonostante entrambi i nodi abbiano eseguito il medesimo codice. Se questo accadesse non ci sarebbe modo per la rete di arrivare ad un consenso condiviso decentrato. Gli oracoli sono quindi visti come un meccanismo per colmare il divario tra il mondo off-chain e gli smart contract.

Nel capitolo 3 abbiamo invece introdotto un nuovo framework che porta il nome di Takamaka, descritto nel dettaglio nel paper [18], scritto dal Prof. Nicola Fausto Spoto dell'Università di Verona. Lo sviluppo di questo progetto nasce dall'idea di creare un framework che permetta di sfruttare le competenze e gli strumenti già esistenti del mondo Java al fine di costruire smart contract in modo semplice e confortevole. Takamaka rappresenta un framework Java per la programmazione di smart contract. Utilizza in particolare un sottoinsieme delle librerie Java ed è inoltre supportato dalla libreria `takamaka.jar`. I metodi di supporto permessi fanno parte della libreria standard Java e vengono chiamati white-listed. Al loro interno ci saranno tutti quei metodi che assicurano un risultato deterministico, perciò non saranno ammessi i metodi per la concorrenza. Nel capitolo 4 abbiamo quindi ricercato quei metodi, all'interno delle librerie Java, che presentano comportamenti non deterministici. Sono stati esaminati alcuni metodi della classe `java.lang.Object` quali il metodo `hashCode()` e il metodo `toString()`. Nel primo caso siamo di fronte ad un metodo che ci ritorna l'hash code dell'oggetto su cui è applicato e basandoci sulla definizione data a questa funzione possiamo subito procedere con il negarne l'utilizzo su blockchain dal momento che l'hash code ritornato da tale funzione non deve rimanere costante da un'esecuzione all'altra e tanto meno tra JVM differenti. Il risultato ritornato da un codice che ne fa uso sarà diverso ad ogni esecuzione. Il metodo `toString()` funziona in modo simile, infatti ritorna una stringa composta dal nome della classe di cui l'oggetto è istanza, seguita dal carattere '@' e una rappresentazione esadecimale dell'hash code di tale oggetto. Il problema qui sta proprio nell'ultima parte, la rappresentazione esadecimale dell'oggetto, per cui, anche per tale metodo, ritorniamo alle conclusioni tratte per `hashCode()`. Abbiamo poi analizzato le classi `java.util.HashSet` e `java.util.HashMap` entrambe queste classi non forniscono garanzia circa l'ordinamento degli oggetti al loro interno che vengono dispersi in vari bucket tramite apposite funzioni hash. Per tali motivi l'utilizzo di queste classi non è raccomandato soprattutto se l'ordinamento degli oggetti contenuti da queste strutture può in qualche modo influire sui risultati in output e quindi sullo stato globale. Una possibile problematica può insorgere nel caso venisse implementato un metodo su blockchain che ritorna un iteratore di un oggetto di una di queste classi. Il metodo `iterator()` fa al caso nostro e, dato che ritorna gli elementi con lo stesso ordine con cui tali elementi compaiono all'interno del `Set` o della `Map` in cui sono contenuti e, dato che tale ordine non è garantito rimanere costante, sarebbe buona norma vietarne l'utilizzo all'interno di blockchain. La classe analizzata subito dopo è stata la classe `java.util.Stream` che rappresenta un flusso di riferimenti ad oggetti su cui possono essere eseguite operazioni di aggregazione in modo parallelo o sequenziale. Già da questa definizione possiamo concludere che tutti quei metodi che permettono l'esecuzione parallela di operazioni su tali stream dovrebbero essere vietate. Ci siamo poi soffermati ad analizzare il metodo `findAny()`, operazione terminale degli stream, che restituisce qualche elemento della stream il cui comportamento risulta

esplicitamente non deterministico. Anche questo metodo viene appunto vietato ma, nel caso si necessitasse di un metodo con la medesima funzionalità, è consigliato l'utilizzo del metodo `findFirst()` il cui comportamento risulta invece deterministico. L'analisi del metodo `forEach` porta alle stesse conclusioni dato che, anche in questo caso, la documentazione parla chiaro "*Il comportamento di questa operazione è esplicitamente non deterministico*". Viene inoltre riportato che tale metodo, nelle esecuzioni su parallelStream, non garantisce il rispetto dell'ordine con cui gli elementi si presentavano inizialmente nello stream, questo per non compromettere i benefici introdotti dall'analisi parallela. Nonostante gli svariati test di applicazione del metodo `forEach` a delle stream sequenziali abbiano riportato sempre gli stessi risultati, facendo credere che tale metodo si comporti in modo deterministico, la documentazione parla chiaro e se è indicato che tale operazione è esplicitamente non deterministica non significa che non si comporterà mai in modo non deterministico, ma bensì che tale metodo non è affatto obbligato a comportarsi in modo deterministico. Per questo motivo il non riuscire a trovare un esempio di funzionamento anomalo non può voler dire che tale metodo ritorni sempre i risultati aspettati. Possiamo quindi concludere che tale metodo non può essere incluso all'interno dei metodi white-listed di takamaka. Nel caso il programmatore necessitasse di un metodo che esegua delle operazioni su ogni elemento della stream preservando l'ordine iniziale dalla stream, può comunque utilizzare il metodo `forEachOrdered` che, a differenza del fratello `forEach`, risulta deterministico anche se applicato a stream parallele. L'analisi si è poi spostata sul metodo `peek` che risulta molto simile a `forEach` se non per il fatto che è un operazione intermedia e quindi ritorna una stream. Viene solitamente usata per il debug ma questo non ne esclude l'utilizzo su blockchain. Questa operazione soffre purtroppo delle stesse problematiche descritte per il metodo `forEach` e dobbiamo perciò escluderlo dall'utilizzo su blockchain.

L'analisi prosegue con la classe `java.util.Random` che sembrava inizialmente scartata a priori dato che porta della casualità all'interno della blockchain ma vedremo che non è così per tutti i suoi costruttori. Infatti il suo costruttore principale, `Random()` crea un'istanza della classe `Random` partendo comunque da un `seed` che non sarà immesso dall'utente ma verrà recuperato dal sistema tramite il metodo `System.nanoTime()`. Comprendiamo quindi il vero funzionamento della classe in analisi e intuiamo quindi che tale metodo ritorni risultati differenti su macchine differenti, dato che il valore del `seed`, settato tramite la funzione `System.nanoTime()`, restituirà risultati diversi su ogni macchina sulla quale verrà richiamato. Nel caso in cui questo valore `seed` venga invece settato manualmente ci troveremo davanti ad un generatore pseudocasuale che, partendo dal valore imposto di `seed`, calcola la medesima sequenza di valori su ogni JVM. L'unica restrizione è data dal fatto che tutte la macchine virtuali condividano la stessa funzione LCF. L'accettare la classe `Random` se e solo se istanziata tramite il passaggio di un valore di `seed`, da la possibilità al programmatore di avere a disposizione un metodo che gli permette di creare un generatore di numeri pseudocasuali che in certe occasioni potrebbe essergli utile.

L'ultima analisi si sposta inevitabilmente con l'analizzare la classe `java.lang.System`, citata all'interno dell'analisi della classe `Random`. Come abbiamo visto contiene il metodo `nanoTime()` che ritorna un valore temporale di esecuzione della JVM misurato in nanosecondi. L'utilizzo di questo metodo all'interno della classe `Random` è data dal fatto che il valore da lui restituito rappresenta un valore abbastanza casuale da

scongiurare istanze simili e per questo motivo non verrà inserita all'interno dei metodi white-listed di Takamaka. Anche per il metodo `currentTimeMillis()` capiterà una sorte simile, infatti anch'esso fa riferimento a dei valori temporali che possono comunque discostarsi nelle varie esecuzioni e tra JVM diverse. Questo metodo restituisce un `long` che rappresenta il tempo corrente misurando i millisecondi intercorsi dalle ore 00:00:00 del 01/01/1970. Questo valore sarà chiaramente diverso per ogni JVM. Un ulteriore problematica è sorta dal fatto che il vietare l'utilizzo di tale metodi implica il negare l'utilizzo di tutti quei metodi di altre classi che ne fanno normalmente uso. Tra queste, nel caso di `cueerntTimeMillis()`, dovremmo procedere con il negare l'esecuzione dei metodi presenti, per esempio, all'interno delle classi `java.util.Date`, `java.time.Clock` e `java.util.Timer` che ne fanno uso. Questa considerazione è ovviamente estesa a tutti i metodi sopra descritti, e non utilizzabili nel contesto blockchain.

Bibliografia

[1] Andreas M. Antonopoulos. *Mastering Bitcoin - Programming the Open Blockchain*. Vol. 2. O'Reilly Media, Inc., 2017.

[2] Andreas M. Antonopoulos. *Mastering Bitcoin - Unlocking Digital Cryptocurrencies*. Vol. 1. O'Reilly Media, Inc., 2014.

[3] Mauro Bellini. *Blockchain: cos'è, come funziona e gli ambiti applicativi in Italia*. 2018. URL: https://www.blockchain4innovation.it/esperti/blockchain-perche-e-cosi-importante/#Le_definizioni_di_Blockchain.

[4] Vitalik Buterin. "Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative". In: (). URL: http://www.hashcash.org/papers/dagger.html.

[5] Thaddeus Dryja. "Hashimoto: I/O bound proof of work". In: (). URL: http://diyhpl.us/~bryan/papers2/bitcoin/meh/hashimoto.pdf.

[6] en.bitcoinwiki.org. *Dagger-Hashimoto*. URL: https://en.bitcoinwiki.org/wiki/Dagger-Hashimoto.

[7] en.wikipedia.org. *Directed acyclic graph*. URL: https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[8] ethereum.org. *Ethash*. URL: https://github.com/ethereum/wiki/wiki/Ethash.

[9] Ken Hodler. *Multibit is No Longer Supported*. 2016. URL: https://multibit.org/.

[10] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2009). URL: bitcoin.org.

[11] Oracle. *Java docs lang.Object*. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html.

[12] Oracle. *Java docs util.Date*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Date.html.

[13] Oracle. *Java docs util.HashMap*. URL: https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html.

[14] Oracle. *Java docs util.HashSet*. URL: https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html.

[15] Oracle. *Java docs util.Stream*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[16] Oracle. *Java stream parallelism*. URL: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html.

[17] randao.org. “Randao: Verifiable Random Number Generation”. In: (2017). URL: https://www.randao.org/whitepaper/Randao_v0.85_en.pdf.

[18] Nicola Fausto Spoto. “Takamaka: A Java Framework for Smart Contracts”. In: (2018). URL: https://github.com/spoto/takamaka/.

[19] Giampaolo Trapasso. *Java 8: uno sguardo agli stream*. URL: https://codingjam.it/java-8-uno-sguardo-agli-stream/.

[20] Gavin Wood. “Ethereum: A secure decentralised generalised transaction ledger (Byzantium version)”. In: (2018). URL: https://ethereum.github.io/yellowpaper/paper.pdf.