Dispense di Sistemi distribuiti cooperativi

Emanuele Izzo, Vlad Cristinel Simon
Gabriele Benedetti, Giulia Pascale

Corso di laurea magistrale Informatica
Università Tor Vergata, Facoltà di Scienze MM.FF.NN.
13/10/2021

Documento realizzato in LaTeX

Indice

Parte I

La blockchain, Bitcoin ed Ethereum

Capitolo 1

Introduzione

1.1 La blockchain

Partiamo dal modello centralizzato banca-cliente: in questo sistema tutte le transazioni vengono gestite dalla banca, ed ogni cliente che si affida ad essa è a conoscenza solo delle proprie transazioni; possiamo dire pertanto che esiste un'intermediazione trai vari clienti della banca da parte della stessa, la quale si occupa anche di fare da garante. Per avere che tale sistema funzioni è necessario avere la fiducia dei clienti, la quale non si può ottenere se non troppo tardi. In che modo possiamo risolvere questo problema? Poniamo di effettuare un processo di disintermediazione, ossia: poniamo che la banca non si occupa più di fare da intermediario nelle varie transizioni, e che esse avvengono direttamente tra le varie persone o **peer**; per fare ciò è necessario che ci sia *fiducia da parte dei peer*, cosa che è possibile fare, ma come? Per prima cosa *tutti i peer devono lavorare sullo stesso insieme di informazioni*; inoltre *le informazioni devono essere necessarie e sufficienti per valutare indici di fiducia*. Questi tre punti appena descritti sono alla base della realizzazione e del funzionamento di una **blockchain**, il cui scopo è quello di cercare di coordinare i peer in un contesto decentralizzato, in modo tale che tutti si acccordino su un'unica opinione condivisa.

1.2 La criptovaluta

Il problema che ora ci poniamo è quello di realizzare una **criptovaluta** che sia il più possibile "simile" alle monete metalliche, e che quindi soddisfi i seguenti fondamentali di una blockchain (ossia quelli indicati sopra). L'idea è quella di eliminare le terze parti a cui si da fiducia nel caso dell'utilizzo delle monete metalliche, portando quindi ad un processo di *disintermediazione* (cosa realizzabile tramite una blockchain). Tutte le monete (sia fisiche che criptovalute) devono condividere inoltre alcune caratteristiche:

- Devono essere *portabili*.
- Devono essere *non falsificabili* (o comunque deve essere troppo oneroso rispetto al valore stesso della moneta).
- Devono essere *ben scalabili o frazionabili*.
- Devono essere *riconoscibili*.
- Devono essere *limitate/scarse* in termine di numerosità.

1.2.1 Wallet

Quando parliamo di bitcoin è necessario parlare inoltre di **wallet**, ossia gli "oggetti" che contengono i bitcoin che si posseggono. Essi sono formati da quattro parti:

- L'*identità* del possessore del wallet.
- La *quantità* di bitcoin contenuta nel wallet.
- La *chiave pubblica* (k_{pb}) che viene usata per effettuare le transizioni.
- La *chiave privata* (k_{pr}) che permette di scrivere all'interno del ledger (per poter inserire un entry al suo interno, infatti, è necessario firmare con la propria chiave privata).

Parlando nello specifico dei *bitcoin*, essi sono *anonimi* proprio come le monete metalliche (non presentano un "identificativo") e deve essere garantito che, come le monete metalliche, una volta spesi non è possibile rispenderli nuovamente (*problema della doppia spesa*). Per fare ciò, il bitcoin non associa un identificatore univoco ai singoli bitcoin, bensì li associa univocamente alla chiave pubblica del peer che li possiede.

Esempio: Poniamo di avere 5 peer che possiedono ognuno un proprio wallet con un certo quantitativo di bitcoin come indicato di seguito:

Identità	**Valore**	**Chiave pubblica**	**Chiave privata**
A	3	k_{pb_A}	k_{pr_A}
B	9	k_{pb_B}	k_{pr_B}
C	5	k_{pb_C}	k_{pr_C}
D	0	k_{pb_D}	k_{pr_D}
E	4	k_{pb_E}	k_{pr_E}

Poniamo adesso che sia avvenute le seguenti transizioni:

- D acquista da A un totale di 3 bitcoin.
- E compra da B un totale di 6 bitcoin.
- E vende a C un totale di 5 bitcoin.

Il ledger, avvenute tutte queste transazioni, conterrà le seguenti informazioni:

Infomazione	**Firma**
(k_{pb_A}, k_{pr_A}) 3	k_{pr_A}
(k_{pb_B}, k_{pr_B}) 9	k_{pr_B}
(k_{pb_C}, k_{pr_C}) 5	k_{pr_C}
(k_{pb_D}, k_{pr_D}) 0	k_{pr_D}
(k_{pb_E}, k_{pr_E}) 4	k_{pr_E}
k_{pb_D} 3 FROM k_{pb_A}	k_{pr_D}
k_{pb_E} 6 FROM k_{pb_B}	k_{pr_E}
k_{pb_C} 5 FROM k_{pb_E}	k_{pr_C}

Da notare che le transazioni vengono firmate da chi riceve i bitcoin, e non da chi li sta fornendo (il ragionamento analogo lo si può fare nella vita reale: la ricevuta di pagamento viene fornita e "firmata" da chi riceve i soldi in cambio di un servizio, e non da chi sta pagando).

1.2.2 NFT

Gli **NFT** (**Not Fungible Tokens**) sono oggetti unici, e che quindi possiedo un ID unico; questo ID unico è rappresentato dall'address dell'oggetto (ciò significa che anche uno smart contract è un NFT). Una cosa importante da notare è che nelle blockchain dove vengono scambiate valute tale requisito viene rialssato: infatti, in Bitcoin ed Ethereum, le singole unità non vengono più identificate, ma bensì si associa una certa quantità di cripotvaluta ad una chiave pubblica. Ciò non permette di distinguere due bitcoin nel caso questi vengano scambiati. É possibile scambiare un NFT tra due wallet (per esempio viene passato tra il wallet w_1 e il wallet w_2) tramite una funzione specifica; per eseguire questa funzione sono necessarie due condizioni:

- Il wallet w_1 deve avere GAS.
- il wallet w_1 deve possedere l'NFT.

Queste condizioni vengono verificate tramite la libreria *zeppelin*. Esiste inoltre uno standard *ERC-721* che descrive come costruire e operare con NFT sulla blockchain di Ethereum. Questo standard definisce le interfacce necessarie per la creazione, gestione e scambio di NFT.

1.3 Le transazioni

Le **transazioni** sono al centro dell'ecosistema delle blockchain. Le transazioni possono essere semplici come inviare criptovaluta ad un altro peer, o possono essere piuttosto complesse a seconda delle esigenze. Ogni transazione è composta da almeno un input e un output; gli input possono essere pensati come le monete che vengono spese e che sono state create in una transazione precedente, mentre gli uotput possono essere visti come monete che vengono recevute. Se una transazione è usata per coniare nuove monete, allora non è presente alcun input e quindi non è necessaria alcuna firma; se invece una transazione è usata per trasferire monete a qualche altro peer, allora deve essere firmata dal mittente con la sua chiave privata e un riferimento alla transazione precedente per mostrare l'origine delle monete. Le transazioni non sono criptate e sono visibili pubblicamente nella blockchain. I blocchi sono costituiti di transazioni e questi possono essere visualizzati utilizzando qualsiasi esploratore di blockchain online. Il ciclo di vita delle transazioni è così composto:

1. Un peer inizia una transazione, che viene firmata utilizzando la sua chiave privata.
2. La transazione viene trasmessa alla rete utilizzando un algoritmo di flooding.
3. Alcuni peer della rete si occupano della convalida dellala transazione in base a criteri prestabiliti. Di solito, più di un nodo è richiesto per convalidare le transazioni.

4. Una volta che la transazione è convalidata, viene inclusa in un blocco, che viene poi propagato alla rete. A questo punto, la transazione è considerata confermata.

5. Questo blocco viene poi aggiunto alla blockchain da un miner, il quale dovrà firmare con la sua chiave privata.

6. Nel caso in cui il blocco viene effettivamente aggiunto alla blockchain, il miner riceverà una reward.

1.3.1 Verifica del successo di una transazione

Il modello più semplice di transizione è quello in cui un utente A vuole mandare un messaggio a B e B invia una risposta ad A.

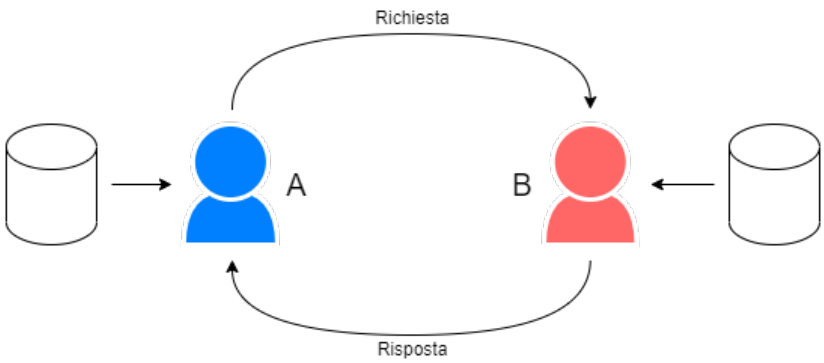

Figura 1.1: Esempio di una transazione semplice.

Notare che nel caso mostrato A e B possiedono entrambi un proprio database dove contengono i dati necessari alla transazione. Il prbolema che si vuole affrontare ora è la verifica del sucecsso della transazione: poniamo che A invia la richiesta Q a B, ma A non riceve poi alcuna risposta R da B; in questa situazione si possono individuare diverse possibilità per cui A non ha ricevuto risposta R da parte di B, soprattutto a livello di rete, tra cui:

- B non ha ricevuto Q.
- B ha smesso di funzionare.
- Il messaggio R è "morto" prima di arrivare ad A.

In queste situazioni riprovare a fare la transazione può portare effetti collaterali come richiedere due volte la stessa risorsa. Se la richiesta coinvolge inoltre un costo, questo problema non è da sottovalutare. La soluzione più semplice è quella di appoggiarsi ad una terza entità che si occupa della comunicazione tra A e B.

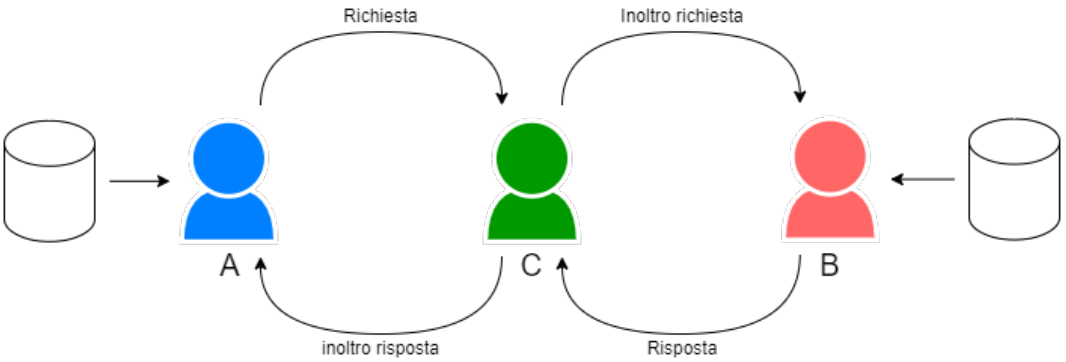

Figura 1.2: Esempio di una transazione con un intermediario.

TCP

In uno scambio effettuato dal **TCP**, ogni transazione scambia il numero di *Sequence* e il numero di *Acknowledge*. Questo permette ad ambo le parti di verificare per ogni scambio gli ultimi dati ricevuti dal destinatario.

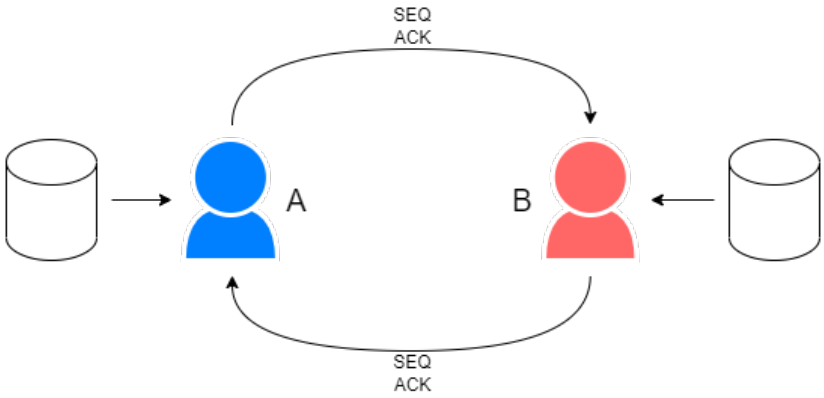

Figura 1.3: Esempio di una transazione che usa il TCP.

In questo modello di transazione, nel momento in cui l'altra parte ci informa tramite l'ACK di non aver ricevuto i dati inviati insieme al SEQ tale che $SEQ_A > ACK_B$, allora abbiamo due tipi di informazioni a disposizione per risolvere il problema. B ci ha informato qual è l'ultima porzione di dati che ha ricevuto e tramite questa informazione possiamo ricavare la porzione di dati che non ha ricevuto, permettendoci di inviare la parte mancante. Questo tipo di soluzione è il *retry* di dati non inviati. Chiaramente questo modello si applica a un contesto di più basso livello rispetto a una transazione che interessa a noi, ma rimane una soluzione interessante a un problema che nasce dal dominio della comunicazione.

CICS

Il sistema **CICS** di IBM, introdotto trami l'uso di terminali intelligenti, applica una soluzione per il problema della comunicazione e uso delle risorse condivise. Queste transazioni bancarie effettuate dai terminali provocano nel sistema un blocco delle risorse per n secondi, evitando che ci siano collisioni tra le varie transazioni. Dopo un determinato tempo, in caso in cui la transazione fallisce, esiste un sistema di rollback primoridale che annulla gli effetti della transazione sul sistema. Questo tipo di soluzione funziona perché

il sistema lavora in un ambiente controllato; infatti il mainframe conosce e controlla tutti i suoi punti di accesso, ossia i terminali del sistema, i quali sono identificati in maniera univoca poiché fisicamente sono identificabili tramite la loro posizione geografica e la porta di connessione del terminale all'interfaccia di collegamento al sistema centrale. Un operatore quando lavora sul terminale comunica il suo identificativo univoco insieme all'identificativo del terminale: questa operazione permette di generare un modello di gestione dove ogni singolo punto di accesso è noto, e questo ci permette di lavorare sulle transazioni in sospeso perché queste saranno localizzate su un terminale fisico identificabile in attesa di essere processate. Quindi il sistema può procedere a risolvere eventuali problemi perché sono noti sia le informazioni della transazione che la posizione nel sistema del terminale che ha richiesto tale transazione.

1.3.2 Validation Authority

Il compito di chi si occupa di progettare transizioni e protocolli di comunicazione è essere in grado di sapere che cosa è successo quando qualcosa non funziona come previsto. La soluzione più semplice, come accennato prima, è quella di appoggiarsi ad una terza entità che fa da intermediario, il cui compito è occuparsi di tutti gli aspetti della transazione, della sicurezza e di tutti i protocolli della comunicazione, oltre che essere un ente certificatore della transazione tra le due parti. Il problema di questo modello è che, affinché sia mantenuta la stessa performance del modello più semplice, viene richiesto che la dimensione di questa terza parte sia raddoppiata rispetto alle dimensioni delle altre due parti. In più questo modello diventa un collo di bottiglia quando si scala il traffico e la gestione di diverse transazioni. Si evince quindi che questo modello non risulta una soluzione generica al problema esposto, ma può essere solo un'idea per dei sottocasi che necessitano di questo tipo di modello per funzionare. Uno di questi modelli prende il nome di **Validation Authority** (**VA**).

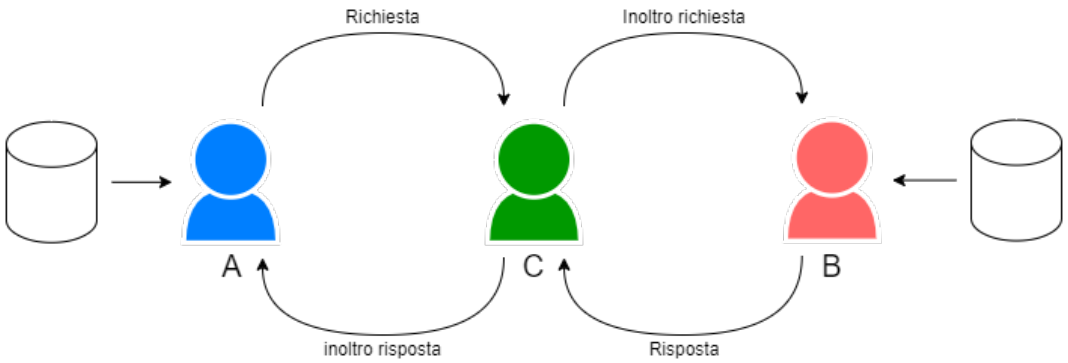

Figura 1.4: Esempio di una transazione con una validation authority (ossia C).

Il modello della VA non introduce una soluzione di tipo tecnico per la risoluzione dei problemi di comunicazione (mancanza di risposta, mancanza di ricezione), ma è una soluzione organizzativa che valida una transizione in base ai dati ricevuti dalle parti interessate. Se analizziamo il tipo di transizione più semplice, possiamo identificare 4 tipi di eventi:

- Invio della richiesta.
- Ricezione della richiesta.

- Invio della risposta.
- Ricezione della risposta.

Questi eventi possono essere validati da un'entità esterna che al termine della transazione può verificare se il processo è andato a buon fine o meno. I membri della transazione, in seguito ad ogni evento, inviano alla VA un messaggio riguardo l'evento. La VA, dopo aver ricevuto i vari messaggi, è in grado di riconoscere lo stato della transazione e può informare i membri dell'esito della stessa. Possiamo identificare 3 stati della transazione in funzione dei messaggi ricevuti:

- *Ok*: Sono presenti $[1, 2, 3, 4]$. Questo significa che ogni evento è avvenuto e la transazione è certificata.
- *Mancata conferma risposta*: Sono presenti $[1, 2, 3]$. Questo significa che la transazione è andata a buon fine ma non è stata ricevuta la risposta, pertanto è sufficiente inviare di nuovo la risposta ripetendo l'evento 3.
- *Invalido*: Sono presenti $[1]$ oppure $[1, 2]$. Questo significa che la transazione non è andata a buon fine e va ripetuta se necessario. L'assenza di invio della risposta comporta che una parte non ha partecipato alla transazione.

Questi stati permettono l'introduzione di vari step di controllo per verificare il flusso della transazione. Se usiamo degli agenti che lavorano in locale con i membri della transazione, possiamo usare dei punti di controllo per verificare lo stato effettivo della transazione. Questi agenti, che comunicano con la VA, non inviano i contenuti della transazione ma inviano dei metadati della transazione in modo da certificare ogni passo del protocollo. Questo comporta che in seguito ad errori di trasmissione, l'agente può semplicemente inviare di nuovo il messaggio alla VA senza rischiare di avere effetti collaterali che sono presenti in una normale transazione. Il registro all'interno della VA quindi si occupa di registrare questi messaggi e di creare una traccia di tutti i passaggi con relativi metadati per permettere successivamente di validare il protocollo della transazione. Una volta definito un modello di una transazione semplice, possiamo costruire una transazione complessa attraverso la definizione di un Partial Order di transazioni elementari che deve essere validato dalla VA. Una soluzione distribuita della VA è l'uso della blockchain per scrivere i messaggi ricevuti. Questo permette a ogni membro di verificare in locale sulla propria blockchain lo stato della transazione.

1.3.3 Consenso

Poniamo di avere una popolazione $P := \{p_1, p_2, ..., p_n\}$, con $|P| = n$, e poniamo che ognuna di queste entità effettua delle azioni, è necessario che venga concordato un **consenso** tra di loro per decidere quali azioni devono essere registrate da tutti. Gli **algoritmi di consenso** si occupano appunto di definire metodi di verifica dei blocchi che vengono aggiunti alla blockchain; tutti gli algoritmi di consenso si basano su 3 elementi ben definiti:

- Ogni utente che buole aggiungere blocchi, detto *validatore*, deve fornire uno *stake*, ossia un qualche valore che il validatore deve mettere in gioco (potenza computazionale, criptovaluta o reputazione); lo stake viene usato per dissuadere dall'agire in modo onesto, poiché in caso di imbroglio tale valore viene perso.

- Una *ricompensa* che viene messa in palio per i vari validatori (di solito si tratta della criptovaluta stessa).

- La *trasparenza*, questo perché si deve essere in grado di scoprire se qualcuno stia barando (idealmente, creare blocchi dovrebbe essere costoso rispetto al guadagno ottenibile, ma economico verificarli).

Una cosa che accomuna tutti gli algoritmi di consenso è il fatto che ogni individuo p_i contribuisce ad esso tramite la sua opinione, la quale può essere pesata. Indichiamo con f_i il peso dell'opinione della persona p_i:

- In un sistema *omogeneo* questo valore è uguale per tutti (ossia $f_i = \frac{1}{n}$).

- In un sistema *eterogeneo* (si riscontra nei bitcoin) questo valore varia per ogni utente (ossia $f_i = \frac{z_i}{n}$ tali che $\sum_{p_i \in P} \frac{z_i}{n} = 1$)

Affinché si abbia il consenso la somma delle fiducie deve essere pari al $50\% + 1$ (ossia si deve avere la maggioranza dei consensi, ossia, indicato con P' le persone favorevoli, $\sum_{p_i \in P'} f_i > \frac{1}{2}$).

1.3.4 Smart contracts

Gli **smart contracts** sono speciali transazioni che, oltre ad effettuare uno scambio di criptovaluta tra due peer, effettua un pezzo di codice all'interno della blockchain e il cui risultato viene salvato nella blockchain. La pubblicazione di uno smart contract richiede 3 elementi:

- l'*ABI*[1] insieme al *codice sorgente* (è necessario un compilatore per ottenere un vettore bytecode).

- La *criptovaluta* per pagare la pubblicazione (è necessario un address da cui verrà effettuato il pagamento).

- l'*identificatore* dello smart contract (è necessario indicare un address per effettuare le chiamate).

Ad ogni smart contract inoltre vengono associati dei *tools* e l'*accesso ad un blocco della blockchain*. Per effettuare una chiamata dello smart contract è necessario passare l'address del chiamante e l'addess dello smart contract.

1.4 Il problema della votazione

Per effettuare una votazione elettronica dobbiamo garantire le caratteristiche fondamentali del voto standard. Oltre ai problemi intrinseci del voto tramite strumenti fisici, esistono ulteriori considerazioni relative al mezzo usato, cioè il mondo digitale:

- *Anonimato*: Risulta una proprietà facilmente implementabile grazie all'uso della blockchain. L'uso di chiave pubbliche per idetinficare i votanti ci permette di scollegare l'informazione di chi vota dall'atto di votazione.

[1] Un'ABI definisce l'interfaccia binaria di basso livello tra due o più parti di software su una particolare architettura.

- *Certezza e immutabilità del voto*: Proprietà garantite dalla natura dello strumento usato, cioè la blockchain.
- *Segretezza*: Il problema di non poter determinare come un individuo ha votato.
- *Incoercibilità*: I votanti non devono essere in grado di comunicare come hanno votato; questo permette di evitare casi di vendita di voti o di estorsioni.
- *Fiducia pubblica*: Il processo stesso del voto genera fiducia perché il processo viene descritto pubblicamente e, dato che le assunzioni del sistema si mantengono, le proprietà descritte dal protocollo portando il votante a fidarsi.
- *Inversione del voto*: Se il voto viene trasformato in un token anonimo e pubblico, rimane il problema di decifrare il voto emesso da un votante.
- *Problemi del voto digitale*: Oltre ai fattori intrinsechi del problema del voto si introducono ulteriori criticità nel sistema dato il mezzo usato per il voto. In un sistema fisico, data la natura distribuita della modalità di voto, è molto difficile intervenire e manipolare una grande quantità di voti perché bisogna attaccare diversi punti del sistema che richiede un potere molto elevato. In un sistema digitale, la concentrazione dei dati fornisce a un eventuale attaccante un mezzo per provocare dei danni molto estesi con uno sforzo non paragonabile al sistema fisico.

1.4.1 Il teorema dei resti cinesi

Siano $n_1, n_2, ..., n_k$ interi a due a due coprimi. Per ogni $a_1, a_2, ..., a_k$ esiste una soluzione al sistema di congruenze:

$$\begin{cases} x \equiv_{n_1} a_1 \\ x \equiv_{n_2} a_2 \\ \vdots \\ x \equiv_{n_k} a_k \end{cases}$$

E tutte le soluzioni di questo sistema sono congruenti nel modulo $n = n_1 n_2 ... n_k$. L'idea dell'uso di questo teorema per il problema del voto è la seguente: se n_1 viene definito come la chiave privata del votante i allora possiamo creare un sistema di voto dove x diventa il candidato da votare e il risultato dell' operazione moluto $x \equiv_n a_i$ diventa il token del voto, cioè l'equazione i del sistema di risoluzione:

$$\begin{cases} x \equiv_{n_1} a_1 \\ x \equiv_{n_2} a_2 \\ \vdots \\ x \equiv_{n_k} a_k \end{cases}$$

Questo sistema di voto permette di capire se tutti i votanti hanno scelto il candidato x. Rimane il problema dell'estensione della soluzione per permettere di capire se la maggioranza ha votato per x.

1.5 Il cooperation agreement

Una blockchain è composta da 3 elementi essenziali:

- I *partecipanti*, ossia coloro che partecipano alla rete e che contribuiscono nella blockchain.
- Gli *assets*, ossia i valori su cui lavorano i partecipanti e che vengono manipolati nella blockchain.
- Le *transazioni*, ossia le operazioni possibili sulla blockchain che possono essere eseguite dai partecipanti (nelle transazioni sono compresi gli *smart contracts*).

Questi elementi vengono indicati nel **cooperation agreement**: esso rappresenta l'accettazione/il rinnovo delle leggi/regole definite in accordo tra tutti i partecipanti.

Esempio: Poniamo di avere un insieme di agricoltori che vogliono assicurare il loro racolto a causa delle basse temperature che possono rovinare il raccolto. L'accordo che viene stretto con gli assicuratori è il seguente:

- Ogni agricolto paga una quota di 1000€ per assicurare il proprio campo.
- Nel caso in cui siano presenti 100 giorni consecutivi in cui la temperatura è inferiore a -5°C, l' assicurazione deve pagare 3000€ all' agricoltore poiché il raccolto è andato perduto.

Vogliamo realizzare una blockchain per la memorizzazione delle temperature e la verifica delle condizioni del contratto di assicurazione. Il cooperation agreement conterrà le seguenti informazioni:

- Partecipanti:
  - Assicuratori ($k_{pb_{ass}}, k_{pr_{ass}}$).
  - Agricoltori ($k_{pb_{agr}}, k_{pr_{agr}}$).
  - Areonautica militare ($k_{pb_{am}}, k_{pr_{am}}$).
  - Associazione assicuratori ($k_{pb_{aass}}, k_{pr_{aass}}$).
  - Associazione agricoltori ($k_{pb_{aagr}}, k_{pr_{aagr}}$).
- Assets:
  - Zone geografiche.
  - Temperatura.
  - Per ogni contratto:
    - Soldi dell'agricoltore.
    - Soldi dell'assicuratore.
- Transazioni:
  - AddContract(*assicuratore*, *agricoltore*, *zonaGeografica*, *durata*, *paga*, *guadagno*, *giorni*, *temperaturaMin*)
    Dove *assicuratore* e *agricoltore* sono le chiavi pubbliche dell'assicuratore e dell'agricoltore che firmano il contratto.

- AddTemperature(*zona*, *temperatura*, *giorno*)
  Questa transazione controlla, per ogni contratto, se le condizioni per il pagamento dell'agricoltore si verificano oppure no, inoltre solo l'areonautica militare può richiame questa transazione.
- ApproveContract(*assicuratore*, *agricoltore*, *zonaGeografica*, *durata*, *paga*, *guadagno*, *giorni*, *temperaturaMin*)
- AddAssicuratore()
  Può essere eseguita solo dall'associazione di assicurazioni.
- AddAgricoltore()
  Può essere eseguita solo dall'associazione di agricoltori.

Capitolo 2

Bitcoin

Il **Bitcoin** (simbolo: ₿ , codice: BTC o XBT) è una criptovaluta e un sistema di pagamento valutario internazionale creato nel 2009 da un anonimo inventore (o gruppo di inventori), noto con lo pseudonimo di Satoshi Nakamoto, che sviluppò un'idea da lui stesso presentata su Internet a fine 2008. Per convenzione, se il termine Bitcoin è utilizzato con l'iniziale maiuscola si riferisce alla tecnologia e alla rete, mentre se minuscola (**bitcoin**) si riferisce alla valuta in sé.

2.1 La tecnologia utilizzata

Il Bitcoin si basa sul trasferimento di valuta tra conti pubblici usando crittografia a chiave pubblica. Tutte le transazioni sono pubbliche e memorizzate in un database distribuito che viene utilizzato per confermarle e impedire la possibilità di spendere due volte la stessa moneta.

2.1.1 Portafogli, chiavi e anonimato

Ogni utente che partecipa alla rete Bitcoin possiede un portafoglio che contiene un numero arbitrario di coppie di chiavi crittografiche. Le chiavi pubbliche, o "indirizzi bitcoin", fungono da punti d'invio o ricezione per tutti i pagamenti. Il possesso di bitcoin implica che un utente può spendere solo i bitcoin associati con uno specifico indirizzo. La corrispondente chiave privata serve ad apporre una firma digitale a ogni transazione facendo in modo che sia autorizzato al pagamento solo l'utente proprietario di quella moneta. La rete verifica la firma utilizzando la chiave pubblica. Se la chiave privata viene smarrita, la rete Bitcoin non potrà riconoscere in alcun altro modo la proprietà del denaro. Gli indirizzi non contengono informazioni riguardo ai loro proprietari e in genere sono anonimi. Gli indirizzi in forma leggibile sono sequenze casuali di caratteri e cifre lunghe in media 33 caratteri, che cominciano sempre per 1, per 3 oppure per *bc*1. Gli utenti possono avere un numero arbitrario di indirizzi Bitcoin, e infatti è possibile generarne a piacimento senza nessun limite in quanto la loro generazione costa poco tempo di calcolo (equivalente alla generazione di una coppia di chiavi pubblica/privata) e non richiede nessun contatto con altri nodi della rete. Creare una nuova coppia di chiavi per ogni transazione aiuta a mantenere l'anonimato.

2.1.2 Transazioni, blockchain e conferme

I bitcoin contengono la chiave pubblica del loro proprietario (cioè l'indirizzo). Quando un utente A trasferisce della moneta all'utente B rinuncia alla sua proprietà aggiungendo la chiave pubblica di B (il suo indirizzo) sulle monete in oggetto e firmandole con la propria chiave privata. Trasmette poi queste monete in un messaggio, la "transazione", attraverso la rete peer-to-peer. Il resto dei nodi validano le firme crittografiche e l'ammontare delle cifre coinvolte prima di accettarla. Per impedire la possibilità di utilizzare più volte la stessa moneta, la rete implementa un "server di marcatura oraria peer-to-peer", che assegna identificatori sequenziali a ognuna delle transazioni che vengono poi rafforzate nei confronti di tentativi di modifica usando l'idea di una catena di proof-of-work. Ogni volta che viene effettuata una transazione, essa parte nello stato di "non confermata"; diventerà "confermata" solo quando verificata attraverso una lista di marcatura oraria gestita collettivamente di tutte le transazioni conosciute, la blockchain. In particolare, ogni nodo "generatore" raccoglie tutte le transazioni non confermate che conosce in un "blocco" candidato, un file che, tra le altre cose, contiene un hash crittografico del precedente blocco valido conosciuto a quel nodo. Prova poi a riprodurre un hash di quel blocco con determinate caratteristiche, uno sforzo che richiede in media una quantità definibile di prove da dover effettuare. Quando un nodo trova tale soluzione la annuncia al resto della rete, i peer che ricevono il blocco ne controllano la validità prima di accettarlo e poi aggiungerlo alla catena. Quando una transazione viene ammessa per la prima volta in un blocco, riceve una conferma. Ogni volta che al di sopra di quel blocco vengono creati altri blocchi figli a esso collegato, riceve un'altra conferma. Quando il blocco contenente la transazione raggiunge sei conferme, ovvero vengono creati sei blocchi collegati a esso, il client Bitcoin cambia stato alla transazione portandola da "non confermata" a "confermata". La motivazione dietro a questa procedura è che a ogni conferma della transazione, ovvero a ogni nuovo blocco che viene creato al di sopra del blocco con la transazione stessa, risulta via via più difficile e costoso annullare la transazione. Un ipotetico attaccante, per annullare una transazione con un certo numero di conferme, dovrebbe generare una catena parallela senza la transazione che desidera annullare e composta da un numero di blocchi pari o superiore alle conferme ricevute dalla transazione. Ne consegue che la catena dei blocchi contiene lo storico di tutti i movimenti di tutti i bitcoin generati a partire dall'indirizzo del loro creatore fino all'attuale proprietario. Quindi, se un utente prova a riutilizzare una moneta che ha già speso, la rete rifiuterà la transazione in quanto la somma risulterà già essere spesa.

2.1.3 Generazione dei bitcoin e costi di transizione

La rete Bitcoin crea e distribuisce in maniera completamente casuale un certo ammontare di monete all'incirca sei volte l'ora ai client che prendono parte alla rete in modo attivo, ovvero che contribuiscono tramite la propria potenza di calcolo alla gestione e alla sicurezza della rete stessa. L'attività di generazione di bitcoin viene spesso definita come "mining", un termine analogo al gold mining (estrazione di oro). La probabilità che un certo utente riceva la ricompensa in monete dipende dalla potenza computazionale che aggiunge alla rete relativamente al potere computazionale totale della rete. Il numero di bitcoin creati per blocco era inizialmente di 50 BTC (aggiunti agli eventuali costi delle singole transazioni). Tale quantità è stata programmata per diminuire nel tempo secondo una progressione geometrica con un dimezzamento del premio ogni 4 anni circa (210 000 blocchi). Così dimensionata, questa serie comporta che in totale verranno creati circa 21

milioni di bitcoin nel giro di 130 anni circa, con l'80% degli stessi creati nei primi 10 anni. In seguito la ricompensa è passata a 25 BTC per blocco il 28 novembre 2012, a 12,5 BTC per blocco il 9 luglio 2016, a 6,25 BTC per blocco l'11 maggio 2020. Con la progressiva riduzione della ricompensa di generazione nel tempo, la fonte del guadagno per i minatori passerà dalla generazione della moneta alle commissioni di transazione incluse nei blocchi, fino al giorno in cui la ricompensa cesserà di essere elargita: per allora l'elaborazione delle transazioni verrà ricompensata unicamente dalle commissioni di transazione stesse. Il volume totale di emissioni dei bitcoin è limitato, in quanto è la somma dei membri di una progressione geometrica decrescente, e non supererà i 21 milioni. 12,7 milioni di bitcoin erano in circolazione nel maggio 2014, mentre 18,77 milioni di bitcoin erano in circolazione nel luglio 2021. Considerato che i miner possono decidere autonomamente quali transazioni inserire in un blocco, chi vuole inviare bitcoin dovrà pagare una tassa di trasferimento, di valore variabile, per incentivare la scelta della propria transazione. A seconda dell'importo scelto da pagare come tassa di trasferimento, il trasferimento avverrà in più o meno tempo. Questo meccanismo costituisce un importante incentivo per i miner, e permetterà loro di poter continuare con la loro attività anche quando la difficoltà per generare bitcoin aumenterà o la quantità di premio per blocco decrescerà nel tempo. I miner collezionano le tasse di transazione associate a tutte le transazioni presenti nel loro blocco dedicato.

2.2 Il bitcoin a livello economico

Figura 2.1: Storico del tasso di cambio col dollaro statunitense fino al 2020.

A differenza delle altre valute a corso legale[1], i bitcoin hanno una caratteristica unica: nessuno può controllarne il valore a causa della natura decentralizzata del metodo di creazione della valuta, sebbene sulla decentralizzazione sono stati sollevati dubbi. In Bitcoin la quantità di valuta in circolazione è limitata a priori; inoltre è perfettamente prevedibile e quindi conosciuta da tutti i suoi utilizzatori in anticipo. L'inflazione da valuta in circolazione non può quindi essere utilizzata da un ente centrale per redistribuire la ricchezza tra gli utenti. I trasferimenti sono definiti come un cambio di proprietà della valuta e vengono effettuati senza la necessità di un ente esterno che debba fare

[1]Per moneta legale (o moneta a corso legale, moneta fiduciaria o, ancora, moneta fiat) si intende uno strumento di pagamento non coperto da riserve di altri materiali (ad esempio: riserve auree), e quindi privo di valore intrinseco (anche indiretto).

da supervisore tra le parti. Tale modalità di interscambio rende impossibile annullare la transazione e quindi riappropriarsi delle monete che hanno cambiato di proprietà. Il client Bitcoin trasmette la transazione ai suoi nodi più vicini, che ne verificano l'autenticità e la disponibilità dei fondi e la ritrasmettono a loro volta ai nodi a cui sono connessi. Il numero totale di bitcoin tende asintoticamente al limite di 21 milioni. La disponibilità di nuove monete cresce come una serie geometrica ogni 4 anni; nel 2013 è stata generata metà delle possibili monete e per il 2017 saranno i tre quarti. In questo modo in circa 136 anni (dopo 33 "halving" intervallati da circa 4 anni di distanza tra loro) verranno generate tutte le monete. All'avvicinarsi di quella data e ipotizzando che la richiesta di bitcoin crescerà più che proporzionalmente rispetto alla disponibilità degli stessi, i bitcoin probabilmente subiranno una deflazione nel valore (cioè un aumento del valore reale) dovuta alla scarsità di nuova moneta. In ogni modo i bitcoin sono divisibili fino all'ottava cifra decimale (con un totale quindi di $2,1 \cdot 10^{15}$ unità), permettendo un completo aggiustamento del valore in un ambiente deflazionistico. Secondo gli sviluppatori, in un ambiente con scarsità di bitcoin i nodi, anziché finanziarsi con la creazione di nuovi bitcoin, trarranno profitto dalla loro capacità di effettuare le transazioni, competendo quindi sui prezzi e mantenendoli bassi.

Capitolo 3

Ethereum

Ethereum è una piattaforma decentralizzata del Web 3.0 per la creazione e pubblicazione peer-to-peer di smart contracts. La criptovaluta a esso legata, **Ether** (simbolo: Ξ, codice: **ETH**), è seconda in capitalizzazione dietro ai Bitcoin.

3.1 La tecnologia utilizzata

Per poter girare sulla rete peer-to-peer, i contratti di Ethereum "pagano" l'utilizzo della sua potenza computazionale tramite una unità di conto, detta Ether, che funge quindi sia da criptovaluta sia da carburante. In altre parole, contrariamente a molte altre criptovalute, Ethereum non è solo un network per lo scambio di valore monetario, ma una rete per far girare contratti basati su Ethereum. Questi contratti possono essere utilizzati in maniera sicura per eseguire un vasto numero di operazioni: sistemi elettorali, registrazione di nomi di dominio, mercati finanziari, piattaforme di crowdfunding, proprietà intellettuale, ecc. Come per le altre criptovalute, la validità di ciascun Ether è garantita da una blockchain, che è un elenco di record in continua crescita, chiamati blocchi, i quali sono collegati tra loro e protetti mediante crittografia. Per definizione, la blockchain è in sé resistente alla modifica dei dati. È un libro mastro contabile, aperto e distribuito che registra le transazioni tra due parti in modo efficiente e permanentemente verificabile. A differenza di Bitcoin, Ethereum opera utilizzando conti e saldi secondo le cosiddette transizioni di stato, che non si basano su output di transazione non spesi (unspent transaction outputs, UTXOs), ma sui saldi correnti (chiamati stati) di tutti i conti, oltre ad alcuni dati aggiuntivi. L'informazione relativa allo stato non è memorizzata nella blockchain, bensì è archiviata in un proprio albero di Merkle, vale a dire un albero binario nel quale ogni nodo è padre di due figli e il suo hash è dato ricorsivamente dalla concatenazione degli hash dei due blocchi a esso associati, secondo il seguente schema:

$$hash_0 = hash(hash_{0-0} + hash_{0-1})$$

Un portafoglio di criptovaluta memorizza le "chiavi" o "indirizzi" pubblici e privati che possono essere utilizzati per ricevere o spendere Ether. Le chiavi private (deterministiche) possono essere generate mediante il protocollo Bitcoin chiamato BIT32, a partire da una sequenza di 12 oppure di 18 parole che è memorizzata nel portafoglio Bitcoin, dalla quale è ricavata la chiave privata master di livello 0 e, a scendere, quelle dei livelli successivi. Per ogni chiave privata viene generato l'indirizzo Bitcoin associato al suo livello. Tuttavia, in Ethereum, questo procedimento non è richiesto: tramite la chiave privata è possibile scrivere nella blockchain, concludendo effettivamente una transazione Ether.

Per indirizzare l'Ether a un conto, è necessario essere in possesso dell'hash calcolato della relativa chiave pubblica, che è calcolato con l'algoritmo di crittografia Keccak-256. I conti Ether non sono nominativi, non identificano univocamente il beneficiario, quanto piuttosto uno o più indirizzi specifici. Gli indirizzi della rete Ethereum incominciano e sono identificati dal prefisso "0x", comune per i numeri numeri in base 16, seguito dai 20 byte più a destra (ordine dei byte) dell'hash Keccak-256 della chiave pubblica ECDSA dove la curva utilizzata è la cosiddetta secp256k1, la stessa di Bitcoin. Dato che in base 16 due digit corrispondono a un byte, gli indirizzi Ethereum contengono 40 cifre esadecimali (il prefisso standard "0x", che è la parte invariante, non è memorizzato, ma "reinserito" di volta in volta).

3.2 L'ether a livello economico

Ethereum ha avuto un valore stabile di circa 10 dollari fino al 2017, anno in cui ha avuto un fortissimo aumento di valore, toccando un picco di 1 261 dollari il 12 gennaio 2018, per poi scendere nuovamente, con picchi ad aprile 2018 (circa 700 dollari), giugno 2019 (circa 300 dollari), febbraio 2020 (circa 200 dollari) e dicembre 2020 (circa 600 dollari). Risalita nel 2021 con un picco di oltre 4 672 dollari.

Parte II

Software e linguaggi per la realizzazione di una criptovaluta

Capitolo 4

Hyperledger e Fabric

4.1 Hyperledger

Hyperledger (o **progetto Hyperledger**) è un progetto di blockchain open source avviato nel dicembre del 2015 dalla Fondazione Linux per supportare lo sviluppo collaborativo di registri distribuiti, con un particolare focus sul miglioramento delle performance e dell'affidabilà di questi sistemi (rispetto a design di criptovalute comparabili), in modo che siano capaci di supportare transazioni commerciali globali da parte di grandi aziende. Sono stati realizzati diversi framework importanti, tra cui:

- **Hyperledger Fabric**: è un'infratruttura a permessi di una blochckain, a cui hanno originariamente contribuito IBM e Digital Asset, fornendo un'architettura modulare con una delineazione di ruoli tra i vari nodi nell'infrastruttura, l'esecuzione di smart contract (chiamati *chaincode* in Fabric), servizi di consenso e appartenenza configurabili.

- **Hyperledger Iroha**: è usato in Cambogia per creare un nuovo sistema di pagamento oltre alla Banca Nazionale della Cambogia, e in altri vari progetti riguardo la salute, finanza e gestione delle identità.

- **Hyperledger Sawtooth**: framework a cui ha originariamente contribuito Intel, comprende una feature per il consenso dinamico che permette l'utilizzo di algoritmi di consenso a caldo in una rete in esecuzione.

- **Hyperledger Besu**: è un codebase a livello aziendale.

Hyperledger fonisce anche vari strumenti per lavorare sulle blockchain, tra cui:

- **Hyperledger Caliper**: è uno strumento di riferimento sulle blockchain e uno dei progetti Hyperldger di cui si è occupato The Linux Foundation. Hyperledger Caliper permette agli utenti di misurare le performance di una specifica implementazione di una blockchain con dei casi d'uso predefiniti.

- **Hyperledger Cello**: è un toolkit del modulo blockchain e uno dei progetti Hyperledger di cui si è occupato The Linux Foundation. Hyperledger Celio punta a portare il modello di sviluppo a richiesta "come un servizio" nell'ecosistema delle blockchain per ridurre lo sforzo richiesto per creare, gestire e terminare blockchains.

- **Hyperledger Composer**: era un insieme di strumenti di collaborazione per la costruzione di blockchain commerciali che rendevano più semplice e veloce per gli imprenditorei per gli sviluppatori creare smart contract e applicazioni blockchain per risolvere problemi di business.

- **Hyperledger Explorer**: è un modulo blockchain e uno dei progetti Hyperledger di cui si è occupato The Linux Foundation. Progettato per creare un'applicazione web facile da usare, Hyperledger Explorer può vedere, invocare, distribuire o invocare blocchi, transazioni e dati associati, informazioni della rete (nome, stato, lista di nodi), catene di codici e famiglie di transazioni, come ogni qualsiasi altra informazione rilevante salvata nel ledger.

- **Hyperledger Quilt**: è uno strumento aziendale per blockchain e uno dei progetti di cui si è occupato The Linux Foundation. Hyperledger Quilt offre interoperabilità tra sistemi di ledger implementado il *protocollo Interledger* (conosciuto come *ILP*), il quale è principalmente un portocollo di pagamento ed è progettato per transferire valori attraverso ledger distribuiti e ledger non distribuiti.

- **Hyperledger Ursa**: è una libreria di crittografia modulare, flessibile e condivisa.

4.2 Hyperledger Fabric

Hyperledger Fabric è un'infrastruttura blockchain con permessi, originariamente fornita da IBM e Digital Asset, che fornisce un'architettura modulare con una delineazione dei ruoli tra i nodi dell'infrastruttura, l'esecuzione di smart contracts (chiamati **chaincode** in Fabric) e servizi di consenso e appartenenza configurabili. Una rete Fabric comprende:

- *Peer nodes* che eseguono chaincode, accedono ai dati del libro mastro, approvano le transazioni e si interfacciano con le applicazioni.

- *Orderer nodes* che assicurano la coerenza della blockchain e consegnano le transazioni approvate ai peer della rete.

- *Membership Service Providers* (*MSPs*), ciascuno generalmente implementato come Certificate Authority, che gestisce i certificati X.509 usati per autenticare l'identità e i ruoli dei membri.

Hyperledger Fabric permette l'uso di diversi algoritmi di consenso, ma l'algoritmo di consenso che è più comunemente usato con la piattaforma è Practical Byzantine Fault Tolerance (PBFT). Fabric si rivolge principalmente a progetti di integrazione, in cui è richiesta una Distributed Ledger Technology (DLT), non offrendo servizi rivolti all'utente se non un SDK per Node.js, Java e Go. Fabric supporta chaincode in Go e JavaScript (tramite Hyperledger Composer, o nativamente dalla v1.1) out-of-the-box, e altri linguaggi come Java installando moduli appropriati. È quindi potenzialmente più flessibile dei concorrenti che supportano solo un linguaggio chiuso per gli smart contracts.

Capitolo 5

Docker

Docker è un progetto open-source che automatizza il processo di deployment di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Docker utilizza le funzionalità di isolamento delle risorse del kernel Linux come ad esempio cgroup e namespace per consentire a "container" indipendenti di coesistere sulla stessa istanza di Linux, evitando l'installazione e la manutenzione di una macchina virtuale. I namespace del kernel Linux per lo più isolano ciò che l'applicazione può vedere dell'ambiente operativo, incluso l'albero dei processi, la rete, gli ID utente e i file system montati, mentre i cgroup forniscono l'isolamento delle risorse, inclusa la CPU, la memoria, i dispositivi di I/O a blocchi e la rete. A partire dalla versione 0.9, Docker include la libreria libcontainer per poter utilizzare direttamente le funzionalità di virtualizzazione del kernel Linux, in aggiunta alle interfacce di virtualizzazione astratte come libvirt, LXC e systemd-nspawn.

5.1 Le funzionalità

Docker implementa API di alto livello per gestire container che eseguono processi in ambienti isolati. Poiché utilizza delle funzionalità del kernel Linux (principalmente cgroup e namespace), un container di Docker, a differenza di una macchina virtuale, non include un sistema operativo separato. Al contrario, utilizza le funzionalità del kernel e sfrutta l'isolamento delle risorse (CPU, memoria, I/O a blocchi, rete) e i namespace separati per isolare ciò che l'applicazione può vedere del sistema operativo. Docker accede alle funzionalità di virtualizzazione del kernel Linux o direttamente utilizzando la libreria libcontainer, che è disponibile da Docker 0.9, o indirettamente attraverso libvirt, LXC o systemd-nspawn. Utilizzando i container le risorse possono essere isolate, i servizi limitati e i processi avviati in modo da avere una prospettiva completamente privata del sistema operativo, col loro proprio identificativo, file system e interfaccia di rete. Più container condividono lo stesso kernel, ma ciascuno di essi può essere costretto a utilizzare una certa quantità di risorse, come la CPU, la memoria e l'I/O. L'utilizzo di Docker per creare e gestire i container può semplificare la creazione di sistemi distribuiti, permettendo a diverse applicazioni o processi di lavorare in modo autonomo sulla stessa macchina fisica o su diverse macchine virtuali. Ciò consente di effettuare il deployment di nuovi nodi solo quando necessario, permettendo uno stile di sviluppo del tipo platform as a service (PaaS) per sistemi come Apache Cassandra, MongoDB o Riak. Docker inoltre semplifica la creazione e la gestione di code di lavori in sistemi distribuiti.

Capitolo 6

Solidity

Solidity è un linguaggio di programmazione orientato ad oggetti a tipizzazione statica usato per la scrittura di smart contracts su varie piattaforme blockchains, tra cui in particolare Ethereum, che girano sulla Ethereum Virtual Machine, conosciuta anche come EVM. É stato inizialmente proposto nell'agosto del 2014 da Gavin Wood, per poi essere sviluppato da Christian Reitwiessner, leader della squadra Solidity del progetto Ethereum. Come specificato da Wood, Solidity è progettato intorno alla sintassi di ECMAScript per renderlo familiare agli sviluppatori web esistenti; a differenza di ECMAScript ha una tipizzazione statica e tipi di ritorno variabili. Rispetto ad altri linguaggi EVM dell'epoca, come Serpent e Mutan, Solidity contiene una serie di importanti differenze:

- Erano supportate variabili membro complesse per i contratti, incluse mappature e strutture arbitrariamente gerarchiche.
- I contratti supportano l'ereditarietà, inclusa l'ereditarietà multipla con linearizzazione C3.
- Fu introdotta un'interfaccia binaria di applicazione (ABI) che facilitava funzioni multiple type-safe all'interno di un singolo contratto (e successivamente supportata da Serpent).
- Un sistema di documentazione per specificare una descrizione centrata sull'utente delle ramificazioni di un method-call era incluso nella proposta, conosciuto come "Natural Language Specification".

6.1 Il layout di un file sorgente Solidity

I file sorgente possono contenere un numero arbitrario di **definizione di contratti**, **direttive d'importazione** e **direttive pragma**.

Esempio di un cotratto in Solidity

```
pragma solidity >= 0.7.0 <0.8.0;

contract Coin {
    // La parola chiave "public" rende le variabili accessibili ad
       altri contratti
    address public minter;
    mapping (address => uint) public balances;

    // Gli eventi permettono ai client di reagire a specifici cambi
        dei contratti che sono dichiarati
    event Sent (address from, address to, uint amount);

    // Il costruttore viene richiamato solo quando il contratto
       viene creato
    constructor() public {
        minter = msg.sender;
    }

    // Manda un ammontare di monete appena create ad un indirizzo (
       richiamabile solo dal creatore del contratto)
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }

    // Manda un ammontare di monete da qualsiasi chiamante ad un
       certo indirizzo
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient
           balance.");
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent (msg.sender, receiver, amount);
    }
}
```

6.1.1 Identificatore licenza SPDX

La fiducia negli smart contract può essere meglio stabilita se il loro codice sorgente è disponibile. Poiché rendere disponibile il codice sorgente tocca sempre problemi legali riguardo al copyright, il compilatore Solidity incoraggia l'uso di identificatori di licenza SPDX leggibili dalla macchina. Ogni file sorgente dovrebbe iniziare con un commento che indichi la sua licenza:

Esempio di identificatore licenza SPDX.

```
// SPDX-License-Identifier: MIT
```

Il compilatore non convalida che la licenza sia parte della lista permessa da SPDX, ma include la stringa fornita nei metadati del bytecode. Se non vuoi specificare una licenza o se il codice sorgente non è open-source, usa il valore speciale `UNLICENSED`. Fornire questo commento naturalmente non ti libera da altri obblighi relativi alle licenze come il dover menzionare un'intestazione di licenza specifica in ogni file sorgente o il detentore originale del copyright. Il commento è riconosciuto dal compilatore in qualsiasi punto del file a livello di file, ma si raccomanda di metterlo all'inizio del file.

6.1.2 Pragma

La parola chiave pragma è usata per abilitare certe caratteristiche o controlli del compilatore. Una direttiva pragma è sempre locale ad un file sorgente, quindi devi aggiungere il pragma a tutti i tuoi file se vuoi abilitarlo nell'intero progetto. Se importi un altro file, il pragma di quel file non si applica automaticamente al file di importazione. I file sorgenti possono (e dovrebbero) essere annotati con un pragma di versione per rifiutare la compilazione con versioni future del compilatore che potrebbero introdurre modifiche incompatibili. Cerchiamo di mantenerli al minimo assoluto e di introdurli in modo che i cambiamenti nella semantica richiedano anche cambiamenti nella sintassi, ma questo non è sempre possibile. A causa di questo, è sempre una buona idea leggere il changelog almeno per i rilasci che contengono cambiamenti di rottura. Questi rilasci hanno sempre versioni della forma `0.x.0` o `x.0.0`. Il pragma di versione è usato come segue:

Esempio di utilizzo della parola chiave `pragma`.

```
1 pragma solidity ^0.5.2;
```

Un file sorgente con la linea sopra non compila con un compilatore precedente alla versione 0.5.2, e non funziona nemmeno su un compilatore a partire dalla versione 0.6.0 (questa seconda condizione viene aggiunta usando `^`). Poiché non ci saranno cambiamenti fino alla versione 0.6.0, puoi essere sicuro che il tuo codice compili nel modo in cui volevi. La versione esatta del compilatore non è fissata, in modo che siano ancora possibili rilasci di bugfix.

6.1.3 Importazione di altri file sorgente

Solidity supporta le dichiarazioni di importazione per aiutare a modulare il vostro codice che sono simili a quelle disponibili in JavaScript (da ES6 in poi). Tuttavia, Solidity non supporta il concetto di esportazione predefinita. A livello globale, si possono usare dichiarazioni di importazione della seguente forma:

Esempio di utilizzo della parola chiave `pragma`.

```
1 import "filename";
```

La parte `filename` è chiamata **percorso di importazione**. Questa dichiarazione importa i contenuti globali da "filename" compresi i file importati a loro volta da "file-

name") nell'ambito globale corrente. Questa dichiarazione importa tutti i simboli globali da "filename" (e i simboli importati lì) nell'ambito globale corrente (diverso da ES6 ma compatibile all'indietro per Solidity). Questa forma non è raccomandata per l'uso, perché inquina imprevedibilmente lo spazio dei nomi. Se si aggiungono nuovi elementi di primo livello all'interno di "filename", essi appaiono automaticamente in tutti i file che importano in questo modo da "filename". È meglio importare simboli specifici esplicitamente. L'esempio seguente crea un nuovo simbolo globale `symbolName` i cui membri sono tutti i simboli globali da `"filename"`:

Esempio di utilizzo della parola chiave `pragma`.

```
1 import * as symbolName from "filename";
```

che ha come risultato che tutti i simboli globali sono disponibili nel formato symbolName.symbol. Una variante di questa sintassi che non è parte di ES6, ma forse utile è:

Esempio di utilizzo della parola chiave `pragma`.

```
1 import "filename" as symbolName;
```

Che è equivalente a `import * as symbolName from "filename";`.

6.1.4 Commenti

È possibile inserire commenti a singola linea (lstinline[style=Solidity]—//—) e commenti su più linee (lstinline[style=Solidity]—/* ... */—).

Esempio di utilizzo della parola chiave `pragma`.

```
1 // This is a single-line comment.
2
3 /*
4 This is a
5 multi-line comment.
6 */
```

6.2 La struttura di un contratto

I contratti in Solidity sono simili alle classi nei linguaggi orientati agli oggetti, Ogni contratto può contenere dichiarazioni di **variabili di stato**, **funzioni**, **modificatori di funzioni**, **eventi**, **errori**, **strutture dati** ed **enumeratori**. Inoltre, i contratti possono ereditare da altri contratti. Esistono anche tipi speciali di contratti chiamati **librerie** e **interfacce**

6.2.1 Variabili di stato

Le **variabili di stato** sono variabili i cui valori sono permanentemente immagazzinati nella memoria del contratto.

Esempio di dichiarazione di una variabile stato.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {
    uint storedData; // Variabile di stato
    // ...
}
```

6.2.2 Funzioni

Le **funzioni** sono unità eseguibili di codice definite solitamente all'interno di un contratto, anche se possono essere definite anche all'esterno di esso.

Esempio di dichiarazione di una funzione.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract SimpleAuction {
    function bid() public payable { // Funzione
        // ...
    }
}

// Funzione ausiliaria definita all'esterno del contratto
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

Le **chiamate alle funzioni** possono avvenire internamente o esternamente e possono avere diversi livelli di **visibilità** rispetto ad altri contratti. Le funzioni accettano **parametri** e forniscono **valori di ritorno** per scambiarsi informazioni tra loro.

6.3 Modificatori delle funzioni

I **modificatori delle funzioni** possono essere usati per moificare la semntica delle funzioni in modo dichiarativo. L'**overloading**, cioè avere lo stesso nome di modificatore con parametri diversi, non è possibile. Come le funzioni, i modificatori possono essere sovrascritti.

Esempio di dichiarazione di un modificatore di funzioni.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

    modifier onlySeller() { // Modificatore
        require(
            msg.sender == seller,
            "Solo i venditori possono chiamare questa funzione."
        );
        _;
    }

    function abort() public view onlySeller { // Uso del
        modificatore
        // ...
    }
}
```

6.3.1 Errori

Gli **errori** permettono di definire nomi e dati descrittivi alle situazioni di fallimento. Gli errori possono essere usati in una **dichiarazione di ripristino**. Rispetto alle descrizioni testuali, gli errori sono molto più economici e permettono di agganciare maggiori informazioni. Si può usare NatSpec per descrivere l'errore all'utente.

Esempio di dichiarazione di un errore.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// Fondi insufficienti per il trasferimento.
/// Richiesti `requested`, ma solo `available` disponibili.
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

6.4 Strutture dati

Le **strutture dati** sono tipi di variabile definiti in modo personalizzato che possono raggruppare diverse variabili.

Esempio di dichiarazione di una struttura dati.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

6.5 Enumeratori

Gli **enumeratori** sono tipi di variabile definiti in modo personalizzato con un numero finito di valori costanti.

Esempio di dichiarazione di un enumeratore.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // Enum
}
```

Parte III

Esempio di realizzazione di una criptovaluta

Appendice A

Creazione di uno smart contract semplice

Partiamo dalla realizzazione di un semplice smart contract chiamato `Metacoin`, il quale dovrà permettere le seguenti azioni.

- Effettuare una transazione (nel caso in cui i fondi non siano sufficienti per il trasferimento la funzione restituisce `false`, altrimenti esegue il trasferimento e restituisce `true`).
- Restituire il bilancio di un peer (ogni peer è identificato dal proprio indirizzo).
- Restituire il bilancio di un peer convertito in Ethereum.

A.1 Creazione dello smart contract

Partiamo dalla realizzazione dello smart contract vero e proprio.

Creazione dello smart contract nel file `Metacoin.sol`.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity >=0.4.25;
3
4  import "./ConvertLib.sol";
5
6  contract MetaCoin {
7      mapping (address => uint) balances;
8
9      event Transfer(address indexed _from, address indexed _to,
           uint256 _value);
10
11     constructor() public {
12         balances[tx.origin] = 10000;
13     }
14
15     function sendCoin(address receiver, uint amount) public returns
           (bool sufficient) {
16         if (balances[msg.sender] < amount) return false;
17         balances[msg.sender] -= amount;
18         balances[receiver] += amount;
```

```
19          emit Transfer(msg.sender, receiver, amount);
20          return true;
21      }
22
23      function getBalanceInEth(address addr) public view returns(uint
            ) {
24          return ConvertLib.convert(getBalance(addr),2);
25      }
26
27      function getBalance(address addr) public view returns(uint) {
28          return balances[addr];
29      }
30  }
```

Andiamo adesso ad analizzarlo pezzo per pezzo:

- Prima di tutto notiamo che viene importato il file `"ConvertLib.sol"`; questo file contiene come detto prima la funzione per convertire il bilancio di un peer in Ethereum. Non è stato utilizzato un alias per rinominare l'import, pertanto ogni volta che bisognerà richiamare la funzione nel file bisognerà scrivere `ConvertLib.convert()`.
- Viene effettuata una mappatura dei bilanci dei vari peer, usando come chiave i loro indirizzi e associando a ciascuno di essi un intero senza segno a rappresentare il loro bilancio.
- Viene dichiarato l'evento `Transfer`, il quale prendere come parametri l'indirizzo del mittente, l'indirizzo del destinatario e la quantità di valuta che viene passata. Questo evento verrà usato per inviare una quantità di criptovaluta da un peer ad un altro nella funzione `sendCoin()`.
- Il costruttore assegna all'indirizzo che istanzia il contratto 10000 criptovalute. Da notare che viene usato `tx.origin` e non `msg.sender`: questo perché `msg.sender` può indicare anche un secondo smart contract che richiama questo smart contract.
- La funzione `sendCoin()` si occupa di effettuare un trasferimento di un certa quantità di criptovaluta tra due peer usando i loro indirizzi; per eseguire questa operazione utilizza l'e'vento `Transfer()` definito precedentemente. Da notare che in caso di quantità non sufficiente per la transazione viene semplicemente restituito `false` dalla funzione, senza quindi utilizzare un errore.
- Le due funzioni `getBalance()` e `getBalanceInEth()` si occupano di restituire il bilancio di un certo peer dato il suo indirizzo: la prima restituisce semplicemente il suo bilancio, mentre la seconda usa la funzione `ConvertLib.convert()` per convertire il bilancio di quel peer in Ethereum (il tasso di conversione è 2). Per comodità, `getBalanceInEth()` usa `getBalance()` per ottenere il bilancio del peer piuttosto che andarlo a cercare direttamente.

A.2 Definizione della conversione e delle migrazioni

Di seguito sono riportati i codici dei file `ConvertLib.sol` e `Migrations.sol`.

Definizione delle funzioni di conversione nel file `ConvertLib.sol`.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.25;

library ConvertLib{
    function convert(uint amount,uint conversionRate) public pure
        returns (uint convertedAmount) {
        return amount * conversionRate;
    }
}
```

Definizione delle funzioni di migrazione nel file `Migrations.sol`.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.4.25;

contract Migrations {
    address public owner;
    uint public last_completed_migration;

    modifier restricted() {
        if (msg.sender == owner) _;
    }

    constructor() public {
        owner = msg.sender;
    }

    function setCompleted(uint completed) public restricted {
        last_completed_migration = completed;
    }
}
```

Andiamo adesso ad analizzarli come abbiamo fatto con il file `Metacoin.sol`:

- Nel file `ConvertLib.sol` viene definita una libreria, e non uno smart contract; ciò vuol dire che il suo contenuto da solo non è utilizzabile, ma può essere usato all'interno di altri smart contract.
- La funzione `convert()` semplicemente converte la quantità passata come parametro moltiplicandola per un valore di ocnversione passato anch'esso come parametro e restituendo e il valore convertito.
- Il file `Migrations.sol` semplicemente di teenrare traccia di quali migrazioni sono state effettaute sulla rete attuale

Appendice B

Creazione di uno smart contract complesso con NFT

Passiamo ora alla realizzazione di uno smart contract che permette la gestione degli NFT, e per fare ciò useremo lo standard ERC-721. Questo standard definisce per ogni NFT una varaibile `uint256` chiamata `tokenId`, e pertanto per ogni contratto che segue lo standard ERC-721 la coppia `contract address`, `uint256 tokenId` deve essere globalmente univoca. Rispetto alla creazione di un semplice contratto, per la creazione di un NFT è necessario creare due elementi differenti: la struttura dell'NFT e un gestore degli NFT.

B.1 La struttura dell'NFT

Per definire la struttura dell'NFT utilizzeremo Solidity, lavorando su due file: `bs721.sol` e `Migrations.sol`. Cominciamo dal descrivere il file `bs721.sol`.

Struttura del file `bs721.sol`.

```
1  pragma solidity >= 0.5.8;
2
3  import "../node_modules/openzeppelin-solidity/contracts/token/
       ERC721/ERC721Full.sol";
4  import "../node_modules/openzeppelin-solidity/contracts/token/
       ERC721/ERC721Mintable.sol";
5  import "../node_modules/openzeppelin-solidity/contracts/token/
       ERC721/ERC721MetadataMintable.sol";
6
7  contract bs721 is ERC721Metadata, ERC721Enumerable, ERC721Mintable,
       ERC721MetadataMintable {
8      address private _owner;     // current owner of the contract
9
10     constructor (string memory name, string memory symbol) public
           ERC721Mintable() ERC721Metadata(name, symbol) {
11         _owner = msg.sender;
12     }
13
14     function exists(uint256 tokenId) public view returns (bool) {
15         return _exists(tokenId);
16     }
17
```

```
18      function tokensOfOwner(address owner) public view returns (
            uint256[] memory) {
19          return _tokensOfOwner(owner);
20      }
21
22      function setTokenURI(uint256 tokenId, string memory uri) public
            {
23          require(msg.sender == _owner, "Only the smart contract
                creator can change this attribute");
24          _setTokenURI(tokenId, uri);
25      }
26
27      function transferFrom(address from, address to, uint256 tokenId
            ) public {
28          require(false, "You cannot transfer that token");
29      }
30
31      function safeTransferFrom(address from, address to, uint256
            tokenId, bytes memory _data) public {
32          require(false, "You cannot transfer that token");
33      }
34
35      function approve(address _approved, uint256 _tokenId) public {
36          require(false, "You cannot transfer that token");
37      }
38      function setApprovalForAll(address _operator, bool _approved)
            public {
39          require(false, "You cannot transfer that token");
40      }
41
42      function burn(uint256 tokenId) public {
43          _burn(tokenId);
44      }
45  }
```

Il file contiene tutte le definizioni per l'utilizzo di NFT. Notare che, seppur vengano incluse le 4 interfacce utilizzate `ERC721Metadata`, `ERC721Enumerable`, `ERC721Mintable` e `ERC721MetadataMintable`, le funzioni seguono la stessa struttura del file `Metacoin.sql`. Come prima, creiamo anche il file `Migration.sql` per tenere traccia di tutte le transizioni.

Struttura del file `Migrations.sol`.

```
1  pragma solidity >=0.5.2;
2  //pragma solidity ^0.5.2;
3
4  import "../node_modules/openzeppelin-solidity/contracts/ownership/
       Ownable.sol";
5
6  contract Migrations {
7    address public owner;
8    uint public last_completed_migration;
9
10   constructor() public {
```

```
    owner = msg.sender;
  }

  modifier restricted() {
    if (msg.sender == owner) _;
  }

  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }

  function upgrade(address new_address) public restricted {
    Migrations upgraded = Migrations(new_address);
    upgraded.setCompleted(last_completed_migration);
  }
}
```

B.2 Il gestore degli NFT

La parte principale di questo esempio è l'interfaccia realizzata in JavaScript: questa interfaccia permette di interlacciare un sito internet allo smart contract realizzato. Un dettaglio improtante da notare è che l'interfaccia utilizza come valuta il Wei, di cui viene riportata la conversione sotto.

Ether	Equivalente in Wei	Costo in Euro
1 Eth	10^18 Wei = (Exa)Wei	255€
10^{-3} Eth = (Milli)Ether	10^15 Wei = (Peta)Wei	0.255€
10^{-6} Eth = (Micro)Ether	10^12 Wei = (Tera)Wei	$255 * 10^{-6}$€
10^{-9} Eth = (Nano)Ether	10^9 Wei = (Giga)Wei	$255 * 10^{-9}$€
10^{-12} Eth = (Pico)Ether	10^6 Wei = (Mega)Wei	$255 * 10^{-12}$€
10^{-15} Eth = (Femto)Ether	10^3 Wei = (Kilo)Wei	$255 * 10^{-15}$€
10^{-18} Eth = (Atto)Ether	1	$255 * 10^{-18}$ Euro

Di seguito ora è riportato il codice del file `interface.js`.

Struttura del file `interface.js`.

```
'use strict';

const Web3 = require('web3');
const fs = require('fs');
const axios = require('axios');

/*********************************************/
/************** CODICE GENERALE **************/
/*********************************************/

//Path compilation, from the compiled contract in build/contracts/
    bs721.json (only the section "abi")
const bs721_json = 'bs721.json';
```

```
13
14 //Load the configurations (network, addresses, keys and contract)
15 let bs712_config = null;
16 let network = null;
17
18 //HttpProvider Endpoint
19 let web3 = null;
20
21 //Owner and contract
22 let contract_owner    = null;
23 let contractAddress   = null;
24 let contract_owner_pk = null;
25
26 try {
27     bs712_config = JSON.parse(fs.readFileSync('bs721-config.json'))
        ;
28     network = bs712_config["network"];
29
30     web3 = new Web3(new Web3.providers.HttpProvider(bs712_config[
        network]["infura"]));
31
32     contract_owner    = bs712_config[network]['contract_owner'];
33     contractAddress   = bs712_config[network]['contractAddress'];
34     contract_owner_pk = bs712_config[network]['contract_owner_pk'];
35 } catch (err)
36 {
37     console.log(JSON.stringify({ result: false, code: err.toString
        () }));
38     process.exit(1);
39 }
40
41 //Just to check if lost something
42 if (!network || !contract_owner || !contractAddress || !
    contract_owner_pk)
43 {
44     console.log(JSON.stringify({ result: false, code: "error in
        bs721-config.json" }));
45     process.exit(1);
46 }
47
48 async function getCurrentGasMediumPrice (network)
49 {
50     if (network === null || typeof(network) === 'undefined' ||
        network === "ropsten")
51     {
52         //console.log("Ropsten and others for development");
53         let gasLimit = web3.utils.toHex(8000000);
54         let gasPrice = web3.utils.toHex(web3.utils.toWei('50', '
            gwei'));
55         return {gasPrice: gasPrice, gasLimit: gasLimit};
56     }
57
58     try {
59         let response = await axios.get('https://ethgasstation.info/
            json/ethgasAPI.json');
60         let prices = {
```

```
61              low: response.data.safeLow / 10,
62              medium: response.data.average / 10,
63              high: response.data.fast / 10
64          };
65
66          let highwei = web3.utils.toWei((3*prices.high).toString(),
                'gwei');
67          return {gasPrice: web3.utils.toHex(highwei), gasLimit: web3
                .utils.toHex(8000000)};
68      } catch (err)
69      {
70          return {gasPrice: web3.utils.toHex(web3.utils.toWei('20', '
                gwei')), gasLimit: web3.utils.toHex(8000000)};
71      }
72  }
73
74 async function main(request)
75 {
76      if (request === null || request.length === 0) {
77          return { result: false, code: "request " + request + " not
                found" };
78      }
79
80      /*************************************************/
81      /************** CODICE SPECIFICO  *************/
82      /*************************************************/
83
84      //Legge abi del contratto
85      const contract_raw = fs.readFileSync(bs721_json, "UTF-8");
86      const contract_json = JSON.parse(contract_raw);
87      const contractABI = contract_json.abi;
88
89      let account = null;
90      try {
91          // Add account from private key
92          account = web3.eth.accounts.privateKeyToAccount('0x' +
                contract_owner_pk);
93          web3.eth.accounts.wallet.add(account);
94          web3.eth.defaultAccount = account.address;
95      } catch (err) {
96          return { result: false, code: err.toString() };
97      }
98
99      //si collega al contratto
100     const contract = await new web3.eth.Contract(contractABI,
            contractAddress);
101
102     //se ci sono transazioni pending e se la richiesta e' di
            aggiornamento (mint e setTokenUri, aspetta)
103     if (request[0] == "mint" || request[0] == "setTokenURI")
104     {
105         const pending    = await web3.eth.getTransactionCount(
                contract_owner, 'pending');
106         const notpending = await web3.eth.getTransactionCount(
                contract_owner);
107         if ((pending-notpending) > 0)
```

```
108         {
109             return { result: false, err: "pending" };
110         }
111     }
112 
113     //Dovrebbe essere serializable e firmata? Ci pensa web3!
114     let tx = {
115         // from: account1,
116         // to: account2,
117         // gasLimit: web3.utils.toHex(4700000),
118         // gasPrice: web3.utils.toHex(web3.utils.toWei('100', 'gwei
                ')),
119     };
120 
121     try {
122         switch (request[0]) //command name
123         {
124             case "price": {
125                 const price = await getCurrentGasMediumPrice (
                        network);
126                 return { result: true, data: { price: price } };
127             }
128             case "name": {
129                 let name = await contract.methods.name().call(tx);
130                 return { result: true, data: { name: name } };
131             }
132                 break;
133             case "symbol": {
134                 let symbol = await contract.methods.symbol().call(
                        tx);
135                 return { result: true, data: { symbol: symbol } };
136             }
137                 break;
138             case "mint": {
139                 if (request.length != 4)
140                 {
141                     return {result: false, code: "mint address
                            tokenid tokenURI"};
142                 }
143                 let gasprices = await getCurrentGasMediumPrice(
                        network);
144                 tx.gasPrice = gasprices.gasPrice;
145                 tx.gasLimit = gasprices.gasLimit;
146                 tx.to = request[1];
147                 tx.from = contract_owner;
148 
149                 let to       = request[1];
150                 let tokenId  = request[2];
151                 let tokenURI = request[3];
152                 try {
153                     let result = await contract.methods.
                            mintWithTokenURI(to, tokenId, tokenURI).send
                            (tx);
154                     return { result: true, data: { mint: true } };
155                 } catch (err)
156                 {
```

```
157                     return { result: false, code: err.toString() };
158                     }
159                 }
160                     break;
161                 case "setTokenURI": {
162                     if (request.length != 3)
163                     {
164                         return {result: false, code: "seTokenURI
                               tokenid tokenURI"};
165                     }
166                     let gasprices = await getCurrentGasMediumPrice(
                           network);
167                     tx.gasPrice = gasprices.gasPrice;
168                     tx.gasLimit = gasprices.gasLimit;
169                     tx.from = contract_owner;
170                     let tokenId  = request[1];
171                     let tokenURI = request[2];
172                     try {
173                         await contract.methods.setTokenURI(tokenId,
                               tokenURI).send(tx);
174                         return { result: true, data: { setTokenURI:
                               true } };
175                     } catch (err) {
176                         return { result: false, code: "error in
                               setTokenURI" };
177                     }
178                 }
179                     break;
180                 case "tokensOfOwner": {
181                     if (request.length != 2)
182                     {
183                         return {result: false, code: "tokensOfOwner
                               address"};
184                     }
185                     let address = request[1];
186                     let tokens = await contract.methods.tokensOfOwner(
                           address).call(tx);
187                     return { result: true, data: { tokensOfOwner:
                           tokens } };
188                 }
189                     break;
190                 case "countTokensOf": {
191                     if (request.length != 2)
192                     {
193                         return {result: false, code: "countTokensOf
                               address"};
194                     }
195                     let address = request[1];
196                     let balance = await contract.methods.balanceOf(
                           address).call(tx);
197                     return { result: true, data: { countTokensOf:
                           balance } };
198                 }
199                     break;
200                 case "balanceOf": {
201                     if (request.length != 2)
```

```
202                 {
203                     return {result: false, code: "balanceOf address
                        "};
204                 }
205                 let address = request[1];
206                 let balance = await web3.eth.getBalance(address);
207                 //console.log(typeof(balance), ": ", balance);
208                 balance = web3.utils.fromWei(balance.toString(), "
                    ether");
209                 return { result: true, data: { balanceOf: balance }
                     };
210             }
211                 break;
212             case "ownerOf": {
213                 if (request.length != 2)
214                 {
215                     return {result: false, code: "ownerOf tokenId"
                        };
216                 }
217                 let tokenId = request[1];
218                 let owner = await contract.methods.ownerOf(tokenId)
                    .call(tx);
219                 return { result: true, data: { ownerOf: owner } };
220             }
221                 break;
222             case "exists": {
223                 if (request.length != 2)
224                 {
225                     return {result: false, code: "exists tokenId"};
226                 }
227                 let tokenId = request[1];
228                 let ret = await contract.methods.exists(tokenId).
                    call(tx);
229                 return { result: true, data: { exists: ret } };
230             }
231             case "tokenURI": {
232                 if (request.length != 2)
233                 {
234                     return {result: false, code: "tokenURI tokenId"
                        };
235                 }
236                 let tokenId = request[1];
237                 let uri = await contract.methods.tokenURI(tokenId).
                    call(tx);
238                 //console.log(uri);
239                 return { result: true, data: { tokenURI: uri } };
240             }
241                 break;
242             case "transferFrom": {
243                 if (request.length != 4)
244                 {
245                     return {result: false, code: "transferFrom from
                         to tokenId"};
246                 }
247                 let from = request[1];
248                 let to = request[2];
```

```
249                 let tokenId = request[3];
250
251                 let gasprices = await getCurrentGasMediumPrice(
                        network);
252                 tx.gasPrice = gasprices.gasPrice;
253                 tx.gasLimit = gasprices.gasLimit;
254                 tx.from = contract_owner;
255
256                 let ret = await contract.methods.transferFrom(from,
                        to, tokenId).send(tx);
257                 console.log(ret);
258                 return { result: true, data: { transferFrom: ret }
                        };
259             }
260                 break;
261             case "approve": {
262                 if (request.length != 5)
263                 {
264                     return {result: false, code: "approve operator-
                            address owner-id owner-key tokenId"};
265                 }
266                 let address = request[1];
267                 let id = request[2];
268                 let pkey = request[3];
269                 let tokenId = request[4];
270                 // Add account from private key
271                 let account = web3.eth.accounts.privateKeyToAccount
                        ('0x' + pkey);
272                 web3.eth.accounts.wallet.add(account);
273                 web3.eth.defaultAccount = account.address;
274
275                 //si collega al contratto
276                 let contract = await new web3.eth.Contract(
                        contractABI, contractAddress);
277
278                 let gasprices = await getCurrentGasMediumPrice(
                        network);
279                 tx.gasPrice = gasprices.gasPrice;
280                 tx.gasLimit = gasprices.gasLimit;
281                 tx.to = address;
282                 tx.from = id;
283
284                 //Esegue il contratto
285                 let ret = await contract.methods.approve(address,
                        tokenId).send(tx);
286                 console.log(ret);
287                 return { result: true, data: { approve: ret } };
288             }
289                 break;
290         }
291     } catch (err) {
292         return { result: false, code: err.toString() };
293     }
294 }
295
296 main(process.argv.slice(2)).then(
```

```
297     ret => {
298         console.log(JSON.stringify(ret));
299     },
300     err => {
301         console.log(JSON.stringify({ result: false, code: err.
                toString() }));
302     });
```

Bibliografia

[1] Narayanan Arvind, Bonneau Joseph, Felten Edward, Miller Andrew, Goldfeder Steven (2016). *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press.

[2] Imran Bashir (2017). *Mastering Blockchain*. Packt.

[3] Antonopoulos Andreas M. (2014). *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. O'Reilly Media.

[4] Dannen Chris (2017). *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress.

[5] Androulaki Elli, Barger Artem, Bortnikov Vita, Cachin Christian, Christidis Konstantinos, De Caro Angelo, Enyeart David, Ferris Christopher, Laventman Gennady, Manevich Yacov, Muralidharan Srinivasan, Murthy Chet, Nguyen Binh; Sethi Manish, Singh Gari, Smith Keith, Sorniotti Alessandro, Stathakopoulou Chrysoula, Vukolić Marko, Weed Cocco Sharon, Yellick Jason (2018). *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains*.

[6] *Docker Documentation* [https://docs.docker.com/].

[7] *Solidity documentation* [https://docs.soliditylang.org/en/v0.8.12/]

[8] R. Kent Dybvig. *The Scheme Programming Language* [https://www.scheme.com/tspl4/]