ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Informatica

SVILUPPO DI UNO SMART CONTRACT IOTA

Relatore:
Prof. Dr. Marco Prandini

Presentata da:
Manuel Paris

Correlatore:
Dr. Giacomo Gori

Sessione III
Anno Accademico 2023/2024

Abstract

Le tecnologie di ledger distribuiti, in continuo sviluppo in questi ultimi anni, hanno rivoluzionato il concetto di transazioni digitali eliminando le autorità centrali e garantendo scambi di denaro diretti e sicuri. In questo ambito, IOTA si distingue dai sistemi di blockchain tradizionali grazie al Tangle, una struttura a grafo aciclico diretto che migliora la scalabilità. Elimina inoltre la figura del minatore di token e le tasse sulle transazioni per incentivare un'economia più bilanciata e sostenibile, priva di inflazione.

Un'altra rivoluzione nel settore è stata l'introduzione degli Smart Contract, programmi dal codice immutabile eseguiti sui ledger stessi per ampliare significativamente le possibilità di interazione con il sistema, estendendo i semplici scambi di denaro.

Questa tesi si concentra sullo sviluppo di uno Smart Contract IOTA, analizzando vantaggi e limitazioni pratiche della produzione di questi applicativi su piattaforme decentralizzate.

Viene descritto il processo di realizzazione dello smart contract, scritto in Solidity, e dell'applicazione web che permette agli utenti di interfacciarsi con esso, scritta in React e TypeScript. Per comunicare con lo smart contract, l'applicazione usa la libreria web3.js.

Infine, vengono valutate le prestazioni degli Smart Contract su IOTA rispetto a Ethereum, evidenziandone i punti di forza e i possibili sviluppi futuri per proseguire le sperimentazioni svolte in questa tesi.

Indice

Elenco delle Figure

Elenco delle Tabelle

Elenco dei Codici

Tabella degli Acronimi

ABI	Application Binary Interface	[ix, 54, 55, 67, 81]
BFT	Byzantine Fault Tolerant	[14]
BLOB	Binary Large OBject	[19]
DAG	grafo aciclico diretto	[i, 1, 5, 6, 24, 25, 77, 79]
EVM	Ethereum Virtual Machine	[16, 30, 32]
MEV	Miner Extractable Value	[29]
NFT	Non-Fungible Token	[20]
PoW	Proof of Work	[2, 6, 28, 29, 75]
UTXO	Unspent Transaction Output	[iii, 11, 12, 14, 15, 20, 22–25, 30]
VM	Virtual Machine	[2, 16, 18, 19, 28–30, 32]

Capitolo 1

Introduzione

Negli ultimi anni, la tecnologia dei ledger distribuiti ha rivoluzionato il concetto di transazioni digitali, eliminando la necessità di un'autorità centrale per proporre scambi di denaro veloci e sicuri in una rete peer to peer. In questo contesto, IOTA si distingue dai classici sistemi basati su blockchain grazie all'uso del Tangle, una struttura a grafo aciclico diretto (DAG) che risolve uno dei principali limiti delle blockchain, ovvero la scarsa scalabilità della struttura data dalla congestione della rete all'aumentare dell'afflusso di partecipanti. Inoltre, IOTA propone ulteriori vantaggi rispetto ad altri ledger, come la mancanza di un modo per coniare nuovi token e l'assenza di tasse sulle transazioni, in modo da contrastare l'inflazione sulla moneta.
Un'altra importante rivoluzione nel settore dei ledger distribuiti è stata l'introduzione degli Smart Contract, codici immutabili che vengono eseguiti sulle blockchain in modo da permettere di usufruire della loro elevatissima programmabilità per espandere le possibilità di interazione con il ledger. Questo ha aperto le porte al loro impiego in molteplici contesti diversi, dai sistemi di votazione alla gestione di proprietà immobiliari.
Questa tesi si concentra sullo sviluppo di uno Smart Contract IOTA, con lo scopo di analizzare i vantaggi e le limitazioni della programmazione su queste piattaforme decentralizzate. Viene sviluppata anche un'applicazione esterna che si interfaccia con lo smart contract, in modo da analizzare come questi

ultimi vengono resi disponibili agli utenti finali in un contesto reale.
Il Capitolo 2 introduce IOTA e la storia della sua evoluzione. Viene illustrato il funzionamento del Tangle e come la necessità di approvare due transazioni quando se ne effettua una nuova aumenti la sicurezza, proponendo in questo modo anche una Proof of Work che è molto ridotta rispetto a quelle di altri ledger. Viene poi descritto il modo in cui IOTA ha introdotto gli smart contract nel suo sistema, adottando una nuova struttura multi-ledger dotata di multiple blockchain che permettono di eseguire smart contract in parallelo, migliorando notevolmente la scalabilità e allo stesso tempo permettendo l'interoperabilità fra di loro. Successivamente si analizza l'aggiornamento IOTA 2.0 che propone ulteriori miglioramenti all'economia, introducendo un sistema di validazione decentralizzato e una nuova risorsa chiamata Mana necessaria per partecipare alla rete. In seguito viene anche comparata nel dettaglio la struttura di IOTA con quelle di altre blockchain, per evidenziare i maggiori miglioramenti introdotti da IOTA Foundation. Infine, viene introdotto IOTA Rebased, un nuovo progetto di IOTA annunciato ufficialmente durante la stesura di questa tesi, che punta a creare un unico ledger unificato includendo gli smart contract direttamente nel ledger principale del Tangle, grazie all'impiego di una nuova Virtual Machine MoveVM.
Il Capitolo 3 illustra le scelte progettuali per lo sviluppo dello smart contract di questa tesi e della sua applicazione esterna. Il progetto in questione è un sistema che permette a un'entità di creare una lotteria, e ad altre di parteciparvi acquistando biglietti. Alla fine del processo, si fornisce un ritorno economico al creatore e al vincitore della lotteria.
Il Capitolo 4 segue i passi del processo di implementazione dello smart contract e dell'applicazione esterna. Vengono spiegate le diverse fasi di sviluppo dello smart contract, scritto in Solidity, e dell'applicazione web, sviluppata in React e TypeScript con l'utilizzo della libreria web3.js e dell'estensione web MetaMask per interfacciarsi con lo smart contract.
Il Capitolo 5 analizza i vantaggi e le limitazioni riscontrati durante il processo di sviluppo di questo programma su ledger distribuito, e valuta l'efficienza

del ledger di IOTA comparando i tempi di esecuzione dello smart contract su tale struttura rispetto alla struttura tradizionale di Ethereum.
Il Capitolo 6 riassume il lavoro svolto in questa tesi e conclude proponendo i modi in cui quest'ultimo può essere portato avanti in futuro.

Capitolo 2

Scenari applicativi e stato dell'arte

Il concetto base su cui è sviluppata questa tesi sono i ledger distribuiti, una tecnologia che ha rivoluzionato l'approccio agli scambi di denaro digitalizzati, fornendo la possibilità di effettuare transazioni digitali senza dover passare per un'autorità centrale[1]. Questo consente scambi di denaro diretti in modalità *peer to peer*, garantendo allo stesso tempo la sicurezza necessaria. I ledger distribuiti forniscono infatti una struttura dati condivisa e immutabile, alla quale è possibile aggiungere informazioni senza richiedere un'autorità centrale: la *blockchain*.
In particolare, la tecnologia di ledger distribuiti su cui si concentra questa tesi è quella apportata da IOTA Foundation, che propone un nuovo approccio al concetto di blockchain. La tecnologia di IOTA si distingue considerevolmente dai sistemi di blockchain tradizionali, come Bitcoin o Ethereum, adottando invece una struttura basata su un grafo aciclico diretto (DAG). Ciò ha permesso la realizzazione di un ledger scalabile e libero da minatori di token, inflazione e tasse.
Questo capitolo introduce i principi fondamentali di IOTA e ne evidenzia le principali differenze rispetto alle altre tipologie di ledger distribuiti.

[1] Ad esempio, una banca.

2.1 La base di IOTA: il Tangle

Ciò che distingue maggiormente IOTA dalle altre criptovalute è il fatto che la struttura su cui si basa non è una normale blockchain, ma una struttura più complessa a grafo aciclico diretto (DAG), che prende il nome di Tangle [1]. L'idea principale alla base del Tangle è il fatto che ogni nuova transazione deve "approvare" altre due transazioni effettuate in precedenza nel ledger. La scelta di queste due transazioni è il modo in cui IOTA implementa una Proof of Work (PoW) per le nuove transazioni[2], richiedendo così un lavoro computazionale molto inferiore rispetto alle PoW di altre blockchain come Bitcoin. Allo stesso tempo viene anche migliorata la sicurezza del sistema, poiché i partecipanti sono incentivati ad approvare transazioni valide. Quindi, più approvazioni una transazione riceve, più è probabile che sia valida[3]. Grazie a questa struttura per gestire le transazioni e alla sua natura asincrona, il ledger risolve i problemi di scalabilità tipici delle blockchain tradizionali, rendendo quindi possibile l'aggiunta di transazioni anche in parallelo.

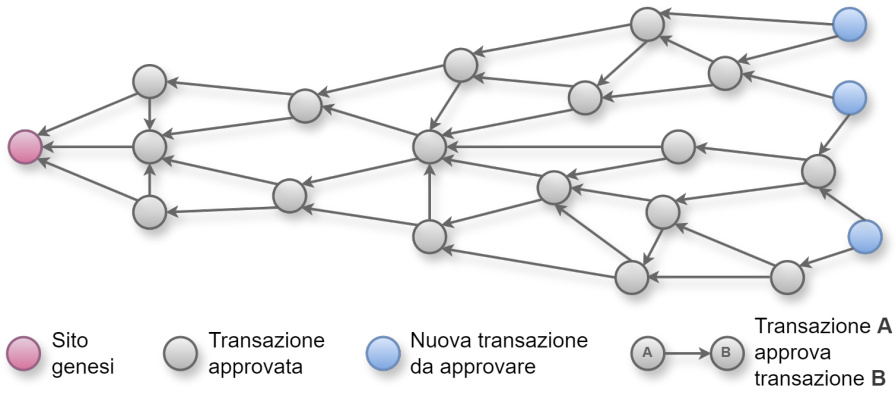


Figura 2.1: Struttura DAG del Tangle

[2]Alla quale contribuisce anche la necessità di risolvere un puzzle criptografico, sempre richiedendo un lavoro computazionale leggero.

[3]Inoltre, approvando una transazione, si approvano anche indirettamente tutte le transazioni approvate in precedenza da essa.

Nella struttura del Tangle, ogni nodo (da qui in poi definito *sito*) corrisponde a una singola transazione effettuata, mentre gli archi che collegano i siti rappresentano le approvazioni delle transazioni. Quando viene effettuata una nuova transazione, questa viene aggiunta al grafo come sito e deve approvare due transazioni già inserite in precedenza nel Tangle. La selezione di queste due transazioni avviene tramite un apposito algoritmo di selezione (vedi 2.1.2). Quando due transazioni sono in conflitto[4], una delle due viene marchiata come *orfana*, e da quel momento in poi le sue approvazioni indirette non verranno più considerate. Per scegliere quale transazione scartare, l'algoritmo di selezione viene rieseguito un determinato numero di volte, e si considera valida la transazione che fra le due viene scelta più volte.
Come si vede in Figura 2.1, il grafo parte da un sito di origine, il sito *genesi*. Questo viene inserito dall'indirizzo che ha originariamente coniato tutti i token della criptovaluta e che, a sua volta, li ha distribuiti tra vari indirizzi *fondatori* per dare inizio al sistema di scambi[5]. Da quel momento in poi, nessun altro token potrà più essere coniato e saranno infatti sempre gli stessi a circolare nel sistema. Questo è ciò che rende il ledger di IOTA libero da minatori di token e immune all'inflazione della moneta.

2.1.1 Definizioni

Prima di proseguire, è opportuno stabilire alcune definizioni che verranno utilizzate nella restante parte della sezione, riportate in Tabella 2.1.

Nodo	Un nodo è un'entità partecipante al sistema, che effettua transazioni.
Sito	Un sito è una transazione effettuata, rappresentata nel Tangle.

[4]Ad esempio, nel caso di *double spending*, approfondito in 2.1.2.

[5]La transazione genesi, di conseguenza, è approvata indirettamente da qualsiasi transazione futura inserita nel Tangle.

Peso	Il peso di una transazione è un valore direttamente proporzionale al lavoro computazionale speso per effettuarla.
Peso cumulativo	Il peso cumulativo di una transazione è la somma del suo peso più il peso di tutte le transazioni che l'hanno approvata (anche indirettamente).
Punteggio	Il punteggio di una transazione è la somma del suo peso più il peso di tutte le transazioni che ha approvato (anche indirettamente).

Tabella 2.1: Definizioni

2.1.2 Selezione delle transazioni da approvare

È necessario capire anche *come* vengono selezionate le due transazioni da approvare quando ne viene effettuata una nuova. Innanzitutto, è importante sottolineare che il sistema di IOTA non impone alcuna regola sulla scelta, bensì si basa sul concetto che per i partecipanti al sistema è più conveniente seguire lo stesso algoritmo di selezione usato dalla maggioranza.

Algoritmi di selezione base

L'algoritmo di selezione più semplice che si possa usare, e che sia anche logico, consiste semplicemente nel selezionare due *tip* casualmente. Nel caso in cui siano in conflitto, la transazione da rendere orfana viene scelta quindi sempre casualmente. Una tip non è altro che una transazione che non è ancora stata approvata da nessun'altra transazione. Logicamente quindi la scelta più sensata per una transazione da approvare è sicuramente una di queste.
Un altro algoritmo semplice, ma un po' meno aleatorio, può essere quello di selezionare le tip che hanno un punteggio maggiore. La logica di questa scelta consiste nell'aggiungere approvazioni (e quindi anche sicurezza) al maggior numero di transazioni. Questo è perché approvare transazioni con un punteggio alto significa anche approvare indirettamente un alto numero di transazioni.

Algoritmo più avanzato per prevenire attacchi double spending

Consideriamo ora un possibile attacco alla sicurezza del sistema, dove l'attaccante cerca di farsi approvare una doppia transazione in cui spende gli stessi token due volte (double spending). Analizziamo quindi come sia possibile migliorare la sicurezza per prevenire questi attacchi, e perché i semplici algoritmi di selezione visti in precedenza non siano sufficienti.
Come si vede anche in Figura 2.2, un attaccante potrebbe effettuare un pagamento in una transazione e, solo dopo che questa ha ricevuto un numero sufficiente di approvazioni e il venditore ha accettato il pagamento, eseguire un'altra transazione double spending. A questo punto, sfruttando la propria potenza computazionale, potrebbe inserire una serie di transazioni che approvino quella double spending (ma non quella originale), ancorandosi saltuariamente anche al Tangle principale per aumentare il proprio punteggio. In questo modo, l'attaccante riuscirebbe a creare una diramazione del Tangle (un subtangle) che parte dalla transazione double spending e si distacca da quella originale. Se riuscisse ad aggiungerci un numero sufficiente di transazioni in modo da superare l'altro subtangle, quest'ultimo verrebbe eventualmente reso orfano[6]. In questo scenario, il ramo valido rimarrebbe quello in cui è presente la transazione double spending, consentendo all'attaccante di spendere con successo gli stessi token due volte.

Per prevenire questo tipo di attacchi, è necessario che il flusso di transazioni effettuate nel sistema sia maggiore della quantità di transazioni che la potenza computazionale dell'attaccante gli permette di eseguire. Per questo è utile (soprattutto nelle fasi iniziali del sistema, quando ancora non è molto popolato) prevedere ulteriori misure di sicurezza, ad esempio avendo dei "checkpoint" periodici in cui controllare tutte le transazioni passate. Nelle prime versioni di IOTA, questo era il compito del *coordinatore*, un singolo nodo fidato che inserisce delle transazioni speciali che convalidano l'intero

[6]Perché, seguendo algoritmi di selezione semplici come quelli descritti in precedenza, più il subtangle parassita supera quello sano, più probabilità ha che vengano selezionate le sue tip.

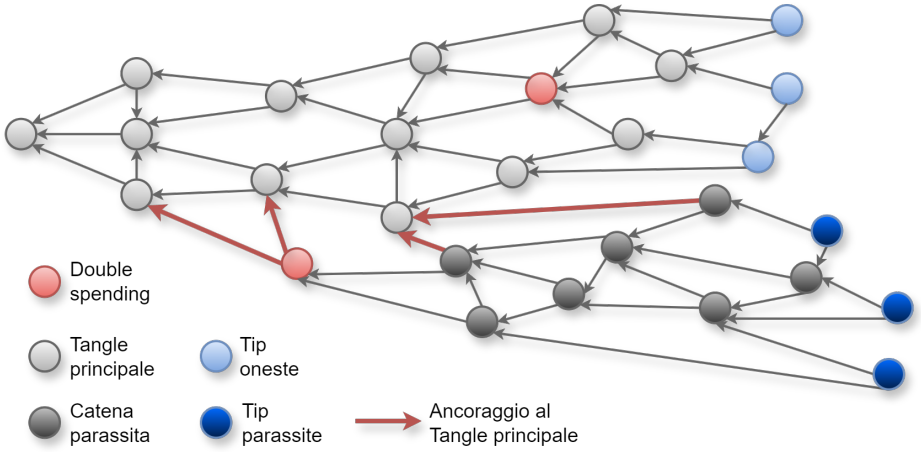


Figura 2.2: Attacco double spending con un subtangle parassita

subtangle precedente al punto in cui si ancorano. Qualsiasi convalida effettuata dal coordinatore è irreversibile [2]. In aggiornamenti successivi di IOTA, questa figura è stata sostituita da un gruppo di consenso decentralizzato (vedi 2.4.1).

Bisogna però considerare anche la possibilità in cui l'attaccante crei *in anticipo* una catena di transazioni sopra a quella double spending, per poi attaccarla al Tangle nel momento in cui la transazione originale viene approvata. In questo modo, la difesa non può più basarsi esclusivamente sulla potenza computazionale. È possibile però utilizzare un algoritmo di selezione più avanzato, in modo che non vengano approvate le transazioni di un eventuale subtangle parassita aggiunto dall'attaccante. Come illustrato in Figura 2.3, l'algoritmo consiste nel selezionare un certo numero di siti nel Tangle, e far partire da essi dei percorsi aleatori verso le tip. Questi percorsi si spostano fra i siti seguendo gli archi del Tangle. La selezione dei siti da cui partire è arbitraria, ma deve essere in un range di distanza dalle tip ragionevole: se è troppo vicino, l'algoritmo risulta poco efficace; se è troppo lontano, i tempi di calcolo diventano eccessivi. I primi due percorsi che terminano raggiungendo una tip, selezioneranno le due tip da approvare.

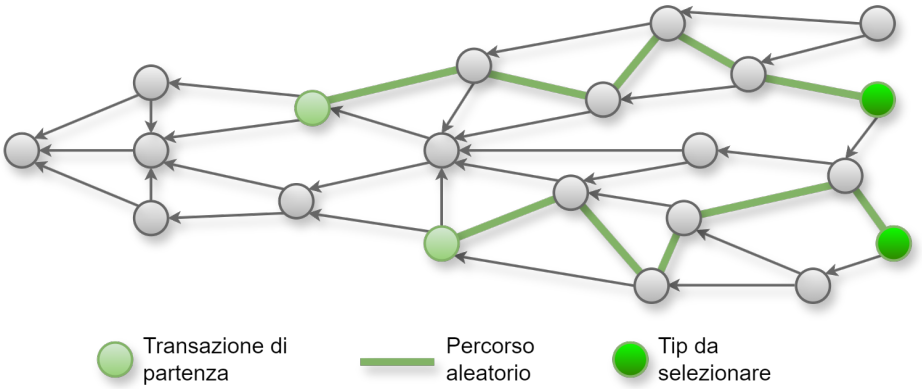


Figura 2.3: Algoritmo di selezione basato su percorsi aleatori

Per la scelta del percorso da prendere, ad ogni passo viene calcolata una distribuzione di probabilità per decidere in quale sito spostarsi. Questa distribuzione di probabilità dipende dalla differenza del peso cumulativo fra il sito attuale e quello di destinazione. Maggiore è questa differenza, minore è la probabilità che il percorso prenderà la strada verso quel sito.
Di conseguenza, la probabilità che il percorso aleatorio finisca all'interno della catena aggiunta dall'attaccante è molto bassa. Infatti, i siti appartenenti a quella singola catena avranno sicuramente un peso cumulativo inferiore rispetto ai siti dell'intero Tangle, generando quindi una differenza maggiore.

2.2 Modello del ledger di IOTA

È importante comprendere anche come vengono salvati e gestiti gli account dei nodi del sistema e i loro token. IOTA segue un modello chiamato UTXO ledger model. Questo modello si basa sul concetto di Unspent Transaction Output (UTXO). Ogni account del ledger, associato a un indirizzo, possiede un determinato numero di UTXO, ognuno dei quali contiene un certo valore di token. Quando si effettua una transazione, vengono utilizzati come input un certo numero di UTXO del mittente, che vengono poi distrutti per produrne di nuovi da assegnare al destinatario. Da qui deriva il nome

di Unspent Transaction Output: gli UTXO presenti nel ledger sono sempre quelli non ancora spesi in una transazione. La particolarità di questo modello è che permette scritture multiple: ciò significa che due o più transazioni possono modificare contemporaneamente lo stato del ledger[7]. Questo meccanismo migliora notevolmente la scalabilità del sistema, ma implica anche che lo stato del ledger non sia oggettivo: in un determinato momento, due entità possono percepirlo in modi differenti.

2.3 Evoluzione IOTA: Multi-ledger e Smart Contracts

La modifica apportata a IOTA che ha maggiormente rivoluzionato il sistema, e che costituisce la base del progetto di questa tesi, è stata l'introduzione di un secondo ledger (Layer 2) sopra al classico Tangle (Layer 1) [3]. Come illustrato anche in Figura 2.4, questo ledger è composto da multipli ledger paralleli, basati sulla struttura classica di blockchain, che si attaccano direttamente al Layer 1. Inoltre, queste blockchain sono programmabili tramite codici immutabili che vi vengono inseriti: gli Smart Contract.

La struttura multi-ledger introdotta permette quindi di avere un numero potenzialmente illimitato di chain programmabili, che possono eseguire in parallelo. Questo elimina uno dei principali difetti delle blockchain, ovvero la scarsa scalabilità. Allo stesso tempo, consente alle varie chain parallele di comunicare tra loro, grazie all'utilizzo del Layer 1 come canale comune.
L'obiettivo principale di questa struttura è fornire un metodo per programmare funzionalità anche complesse, che possano richiedere scambi di denaro. In questo modo, si favorisce l'applicazione dell'ecosistema di ledger distribuiti ad ambiti più ampi, andando oltre i semplici acquisti tramite criptovalute. Inoltre, un altro scopo è quello di consentire la tokenizzazione di qualsiasi as-

[7]Soltanto se non prendono UTXO in comune come input.

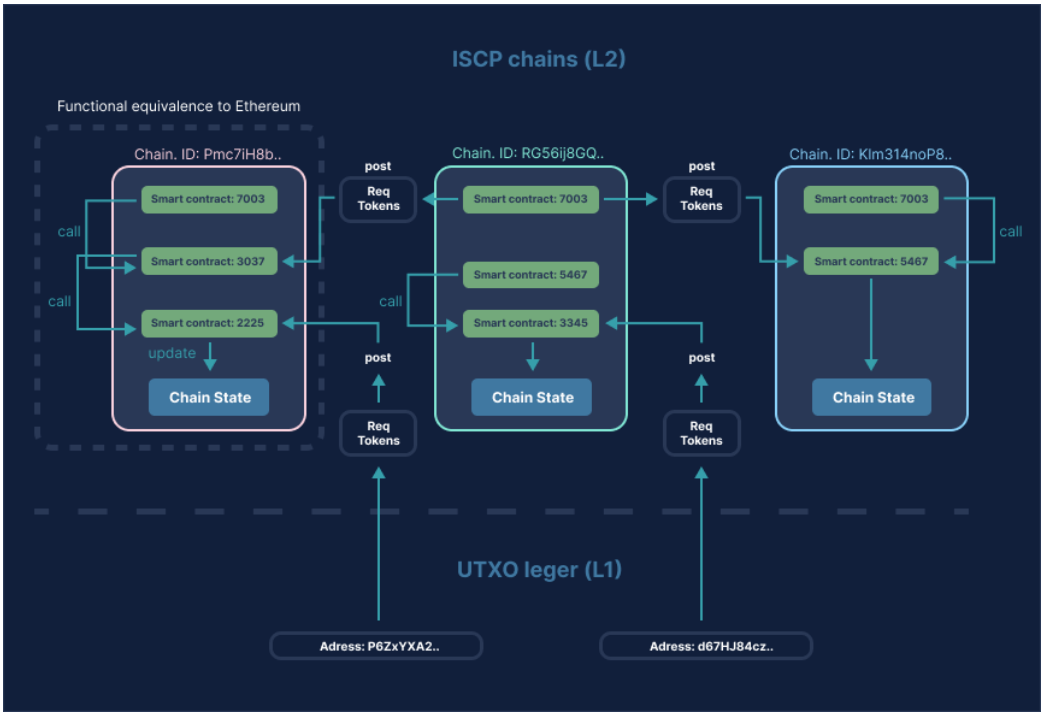


Figura 2.4: Blockchain parallele nel Layer 2 ancorate su account nel Layer 1[8]

set, non limitandosi quindi al solo token IOTA (vedi 2.3.4). Questo permette un ulteriore ampliamento dei possibili ambiti di utilizzo.

2.3.1 Struttura delle chain nel Layer 2

Ogni blockchain che fa parte del Layer 2 ha le seguenti proprietà:

- Un indirizzo proprio, chiamato chain ID, col quale si identifica la chain stessa. Tale indirizzo è immutabile.
- Un indirizzo sul Layer 1, al quale si ancora la chain. Tale indirizzo è separato dal chain ID, permettendo la possibilità di cambiare indirizzo proprietario della chain.

[8]Fonte: `https://archive.ph/DhqHS`

- Una quantità di UTXO, posseduti dall'account proprietario e utilizzabili dalla chain.
- Uno stato, che consiste in coppie chiave-valore. Lo stato è salvato in un database in tutti i nodi che hanno accesso alla chain.
- Multipli smart contract, che modificano lo stato.
- Un gruppo di validatori, nodi che mantengono la costanza dello stato nel sistema distribuito raggiungendo un consenso.

2.3.2 Aggiornare lo stato di una chain

Il meccanismo con cui una chain aggiorna il proprio stato consiste nel prendere in input degli asset (UTXO) che arrivano all'account della chain e, successivamente, aggiornare il proprio stato in base alle computazioni degli smart contract. Tuttavia, come già detto, lo stato degli account del ledger in L1 non è oggettivo. È quindi necessario concordare un unico stato del ledger, in modo da poter produrre uno stato oggettivo della chain.
È per questo motivo che ad ogni chain è assegnato un certo numero di validatori. Un validatore è un'entità, come una persona o un'azienda, che insieme ad altri validatori va a formare il comitato dei validatori della chain. Il compito di questo comitato di validatori è appunto quello di mantenere oggettivo lo stato della chain. Per raggiungere questo obiettivo, il comitato segue un algoritmo di consenso Byzantine Fault Tolerant (BFT), che gli permette di raggiungere un consenso finché ci siano almeno $\frac{2}{3}N + 1$ validatori non malevoli nel comitato di N validatori[9].
Prima che la chain prenda in input degli asset per modificare il proprio stato, i validatori concordano quindi su uno stato del ledger di L1 da considerare oggettivo e sull'ordine degli input da processare. Una volta effettuate le operazioni e modificato lo stato della chain, questo viene depositato tramite una

[9]Sono necessari $\frac{1}{3}N$ validatori malevoli per impedire la produzione di uno stato valido, e $\frac{2}{3}N + 1$ per crearne uno falsificato.

transazione come asset del chain account. La transazione è accettata soltanto se i validatori forniscono una *threshold signature* valida di tale transazione, ovvero una firma ottenuta dalla somma di tutte le firme dei validatori cooperanti (che, come prima, basta siano più di due terzi dei validatori totali). In questo modo, viene garantito il consenso globale sullo stato della chain, assicurandone l'unicità e l'oggettività per tutti i nodi.

State anchoring

Il processo di salvare lo stato della chain come asset del suo account su L1 viene definito state anchoring.
Nel dettaglio, il chain account possiede i seguenti asset:

- **Asset della chain** UTXO che sono collegati e controllati dalla chain tramite valori nel suo stato.
- **Richieste per la chain** UTXO in input da chi vuole utilizzare la chain, che devono ancora essere processati. Una volta processati saranno poi aggiunti agli asset della chain.
- **Alias output** Un UTXO contenente lo state anchoring della chain, che a sua volta contiene:
  - L'ID della chain
  - L'ID del chain account
  - Un timestamp dello state anchoring
  - L'indice di stato, che aumenta ad ogni state anchoring
  - Una prova crittografica del nuovo stato della chain. Viene utilizzata la radice del *Verkle Tree* (una versione più efficiente del Merkle Tree, vedi 2.5.2) calcolato dallo stato della chain.

Ogni volta che avviene uno state anchoring, viene eliminato il precedente alias output e sostituito con quello nuovo, in modo che ci sia sempre e solo un alias output per ogni chain.

Grazie alla prova crittografica, viene assicurato che lo stato sia protetto da manomissioni e che sia uguale per tutti i nodi.

State synchronization

In una sezione dello stato della chain chiamata *blocklog* sono anche presenti i blocchi passati della blockchain, così da poterli utilizzare come cronologia per ricostruire lo stato. Se un nodo infatti si accorge che il suo indice di stato e la sua prova crittografica dello stato sono diversi da quelli nello state anchor, significa che non è sincronizzato con lo stato attuale. A quel punto, confrontando l'indice di stato, il nodo può determinare di quanti blocchi è indietro e richiedere ad altri nodi i blocchi necessari per ricostruire lo stato attuale partendo dal proprio. Questo processo è definito state synchronization.

2.3.3 Funzionamento degli Smart Contract

Come già introdotto prima, uno Smart Contract è un programma eseguito su chain che modifica il suo stato, producendo sempre un output deterministico e oggettivo per tutti i nodi. Il suo codice, immutabile, è salvato nello stato della chain, e ogni smart contract ha a sua disposizione una partizione dello stato per salvare i propri dati.
L'esecuzione dei codici avviene grazie alla Virtual Machine (VM), una parte della chain composta da interpreti di codice e da smart contract predefiniti, necessari al funzionamento della chain. Tutti gli altri smart contract inseriti nella chain vengono aggiunti alla VM come estensione di essa. La VM di IOTA segue lo stesso modello della Ethereum Virtual Machine (EVM) di Ethereum: gli smart contract programmati per quest'ultima sono infatti compatibili anche con la VM di IOTA.
Per garantire flessibilità, la VM offre inoltre astrazioni e interfacce per le sue funzioni principali, permettendo il supporto di multipli tipi di interpreti e di smart contract, anche sulla stessa chain. Un'altra componente essenziale della VM è il *Sandbox*, un'interfaccia che permette agli smart contract di

avere accesso alle risorse della chain. L'accesso viene regolato in modo strettamente deterministico, consentendo sempre l'esecuzione di un solo smart contract alla volta. È quindi grazie al Sandbox che viene assicurato che gli smart contract della chain producano sempre un risultato deterministico e oggettivo per tutti i nodi.

Richiamare uno Smart Contract

Esistono due modi in cui le funzioni di uno smart contract possono essere richiamate:

1. Tramite semplici chiamate sincrone effettuate da altri smart contract all'interno della stessa chain.

2. Tramite chiamate dall'esterno della chain che prendono il nome di *richieste*. Sono quelle che vanno a formare gli asset in input della chain.

Le richieste possono a loro volta essere effettuate in due modi: on-ledger e off-ledger.

- **Richieste on-ledger** Richieste effettuate tramite transazioni su L1, con il chain ID come destinatario. Sono utilizzate quando il richiedente è un indirizzo del ledger e la chiamata prevede lo scambio di asset L1, oppure quando il richiedente è uno smart contract di un'altra chain.

- **Richieste off-ledger** Richieste invece effettuate senza passare per il ledger di IOTA, tramite chiamate API a uno degli *access nodes*[10] della chain. Queste richieste sono molto più veloci ed efficienti e vengono utilizzate se il richiedente è un indirizzo che non ha necessità di scambiare asset L1 tramite la chiamata[11]. In questo caso, la richiesta viene propagata al backlog della chain tramite un meccanismo di disseminazione tra gli access nodes.

[10] I nodi, fra i quali i validatori, che hanno accesso alla chain e possiedono il database con il suo stato.

[11] E quindi, non ha necessità di utilizzare il ledger.

Considerazioni sulle tasse

Nonostante il ledger di IOTA sia libero da tasse, è comunque necessario imporre in L2 dei limiti nell'utilizzo degli smart contract per prevenire abusi da parte dei richiedenti. Per questo motivo viene introdotto il concetto di *gas*. Ogni utilizzo delle risorse della chain richiede un certo quantitativo di gas, direttamente proporzionale alla quantità di risorse utilizzate. Quando viene invocata una funzione di uno smart contract che modifica lo stato della chain, viene quindi consumato il quantitativo di gas necessario. Al contrario, se viene invocata una funzione che non effettua alcuna modifica allo stato (chiamate funzioni *view*) non viene consumato nessun gas.
Il modo per ottenere gas e i limiti sul suo utilizzo sono stabiliti da chi gestisce la chain. La politica più utilizzata prevede una tassa sul consumo di gas, convertendo gli IOTA in gas. In questo scenario, le tasse pagate alla chain vengono suddivise fra i validatori e il proprietario della chain. Sono comunque possibili anche altri tipi di politiche sul gas, come ad esempio non imporre alcun limite, oppure rendere l'acquisizione di gas libera da tasse ma imporre un limite massimo prefissato sul suo consumo.

Smart Contract predefiniti

Come detto in precedenza, la VM di IOTA è fornita di base con un certo numero di smart contract (definiti *core smart contracts*) necessari al funzionamento della chain, elencati qui di seguito in Tabella 2.2.

Root	Lo smart contract radice è necessario per effettuare il deploy di nuovi smart contract e per mantenere il registro di quelli esistenti.
Accounts	Lo smart contract degli account permette la registrazione di account sulla chain stessa. A differenza di L1, viene adottato il modello classico delle blockchain, ovvero il modello basato su account (Vedi 2.5.1). Ogni account è identificato da un account ID, che corrisponde all'account L1 del proprietario. Questo permette agli utenti di spostare qualsiasi proprio asset da L1 alla chain. L'Accounts Contract, oltre a mantenere il registro degli account, si occupa anche di gestire lo scambio di asset fra di essi.
Blob	Lo smart contract dei Binary Large OBject (BLOB) mantiene il registro dei dati di grandi dimensioni.
Blocklog	Lo smart contract del log dei blocchi è quello che mantiene la traccia e la cronologia dei blocchi nella chain (Come visto in 2.3.2).
Governance	Lo smart contract della governance si occupa di gestire gli aspetti amministrativi della chain. Alcuni esempi sono la gestione dei diritti di accesso, delle politiche riguardanti tasse e gas, o della sostituzione dei validatori.
_default	Uno smart contract di default che viene chiamato automaticamente quando arriva una richiesta a uno smart contract non esistente.

Tabella 2.2: I core smart contract della VM di IOTA

2.3.4 Estensione dell'UTXO ledger model

L'UTXO ledger model di base permette soltanto di avere UTXO contenenti valori del token nativo (in questo caso, IOTA). Per poter permettere l'implementazione del Layer 2 e delle sue multi-chain, è stato quindi necessario per IOTA aggiungere un'estensione al modello che introduca tipi di UTXO speciali. Inoltre, l'estensione aggiunge anche la possibilità di destinare UTXO all'indirizzo di una chain. Normalmente ciò non sarebbe possibile perché, per permettere il cambio di proprietario, l'indirizzo della chain non è vincolato all'indirizzo del nodo su cui è ancorata.
I tipi di UTXO aggiunti con questa estensione sono i seguenti:

- **Alias output** Permette lo state anchoring e la creazione degli alias output delle chain. Come detto in precedenza, è presente anche un vincolo che impone uno e un solo alias output per ogni chain.

- **Foundry output** Rende possibile la tokenizzazione avanzata, ovvero la creazione di asset diversi dal token IOTA (chiamati *asset nativi*). In questo modo, diventa possibile la tokenizzazione di qualsiasi cosa. Può esistere un solo foundry output per ogni tipo di token creato. Tale foundry output contiene il bilancio del token e permette al suo gestore di coniarne o distruggerne qualsiasi quantità[12].

- **NFT output** Permette l'utilizzo di Non-Fungible Token (NFT) come asset nativi. È consentito un unico NFT output per uno specifico ID. Un NFT output inoltre contiene una prova di origine immutabile, ovvero la chiave pubblica di chi lo ha creato.

- **Extended value transfer output** Quello che effettivamente permette lo scambio anche di asset nativi, estendendo i normali UTXO e fornendo vincoli aggiuntivi. È sempre tramite questo output inoltre che vengono inviate le richieste alle funzioni degli smart contract sulle chain.

[12] Al contrario del token IOTA, che non è né coniabile né distruggibile dopo la creazione del sistema.

2.4 IOTA 2.0: Mana e economia sostenibile

L'aggiornamento 2.0 di IOTA, uno dei più recenti, rappresenta un'altra evoluzione importante in quanto introduce due novità di svolta:

1. Un nuovo metodo di consenso decentralizzato, che sostituisce il ruolo del coordinatore nel validare lo stato del Layer 1.

2. Una risorsa chiamata *Mana*, che rivoluziona il modo in cui vengono forniti incentivi ai nodi partecipanti al ledger [4].

Il Mana è una risorsa che viene generata da un nodo in due possibili modi:

1. Mantenendo nel proprio wallet dei token IOTA. Viene generato Mana passivamente finché ci sono IOTA nel proprio account. Più IOTA si possiedono, maggiore è la quantità di Mana generata. È importante tener conto anche che alla quantità di Mana generata nel tempo è applicata una funzione di decadimento, che riduce progressivamente la quantità di Mana ottenuto. Questa funzione è stata introdotta per disincentivare l'accumulo eccessivo di token senza una reale partecipazione alla rete, evitando che gli utenti trattengano IOTA al solo scopo di generare Mana.

2. Partecipando attivamente alla sicurezza della rete, agendo come validatore o delegatore (Vedi 2.4.1).

Allo stesso tempo, il Mana è anche la risorsa necessaria per poter utilizzare la rete ed effettuare transazioni. In questo modo, gli incentivi per partecipare alla rete sono puramente pratici e ne beneficeranno quindi soltanto coloro che sono genuinamente interessati a partecipare e contribuire al sistema.
Tutte le azioni necessarie ad accumulare Mana (e quindi, a poter utilizzare la rete) prevedono un acquisto di token IOTA[13]. Questo comporta contemporaneamente due vantaggi:

[13]Ma non necessariamente una spesa, concetto fondamentale che differenzia IOTA dagli altri token (Vedi 2.5.1).

1. Incentivando ad acquistare token IOTA, il token aumenta di valore.

2. Imponendo la necessità di possedere token per utilizzare la rete, aumenta la protezione dagli attacchi di Sybil[14].

Grazie al Mana inoltre, unito alla politica senza tasse di IOTA, si va a creare un'economia ancora più bilanciata. Tale economia infatti, evita il fenomeno in cui si arricchiscono sempre di più i validatori a scapito degli altri partecipanti tramite tasse e premi in token per i loro servizi. Si ricorre invece a un sistema in cui i premi per chi fa da validatore consistono in Mana, senza alterare il bilancio dei token di nessuna entità. In questo modo, chi aiuta a validare la rete riceve comunque dei compensi, ma senza creare uno squilibrio crescente nella distribuzione dei token.

2.4.1 Staking e sistema di consenso

Con IOTA 2.0, il ruolo di un unico coordinatore centrale è stato sostituito da un metodo di consenso decentralizzato, in cui chiunque può partecipare alla validazione delle transazioni agendo come validatore o delegatore. Per farlo è necessario fare lo *staking* di una parte dei propri token IOTA, ovvero "congelarli" nel caso dei validatori o scommetterli su un validatore nel caso dei delegatori.

Validatori

Per diventare validatore è necessario prima iscriversi a una lista di possibili validatori. Per farlo, un nodo deve inserire uno speciale UTXO contenente le informazioni necessarie, chiamato *staking feature*, nello stato del proprio account. È necessario inoltre fare lo staking di una parte dei propri token IOTA, che rimarranno congelati fintanto che si è iscritti.
Dalla lista dei possibili validatori vengono poi selezionati quelli effettivi, che opereranno per un determinato periodo di tempo chiamato *epoca*, e saranno

[14] Attacchi in cui una singola entità malevola può utilizzare multipli nodi per partecipare alla rete, e utilizzarne l'ampio numero per compiere azioni a suo vantaggio.

sostituiti a ogni nuova epoca. Il parametro principale utilizzato nella scelta è il *peso* del validatore, dato dalla quantità di token con cui ha fatto lo staking sommata alla quantità di token che gli sono stati delegati da altri nodi. È inoltre da considerare che i token delegati hanno un valore inferiore rispetto a quelli messi direttamente in staking dal validatore. Questo sistema evita che un validatore abusi della delegazione, delegando token a sé stesso per ottenere lo stesso peso dello staking ma senza congelare i propri fondi.
Il compito di un validatore è inserire *transazioni di validazione* per raggiungere il consenso sullo stato del ledger con gli altri validatori. Con ogni transazione di validazione si esprime il proprio voto su tutto il sub-tangle a cui viene ancorata.
Le transazioni di validazione sono leggere, contengono solo i dati essenziali, hanno priorità maggiore e non costano Mana. Questo garantisce di poterle sempre inserire adeguatamente nel ledger senza però congestionare la rete per le altre transazioni.
Alla fine di un'epoca, un validatore viene ricompensato se ha raggiunto la quota minima richiesta di transazioni di validazione inserite. La quantità di Mana ottenuta dal validatore dipende dal numero totale di transazioni di validazione che ha effettuato, da quanto queste hanno aiutato a raggiungere il consenso e dal peso del validatore.

Delegatori

Se dei nodi non vogliono fare da validatori ma desiderano comunque ottenere del Mana senza dover congelare i propri token, possono delegarli a uno dei validatori. Per delegare dei token è necessario inserire nel proprio account uno speciale UTXO, chiamato *delegation output*, che contiene la quantità di token delegati e l'indirizzo del validatore a cui si intende delegarli. Come detto in precedenza, questa operazione non congela i propri token, al contrario dei validatori. Di conseguenza, un delegatore è libero di spenderli anche dopo averli delegati, oppure anche di ri-delegarli a un altro validatore.
È inoltre importante specificare che è possibile delegare anche token presenti

sulle chain di L2, poiché anche una chain stessa può delegare il proprio potere a un validatore per ottenere Mana.

Distribuzione delle ricompense in Mana

Al termine di un'epoca, le ricompense vengono distribuite tra validatori e delegatori secondo una proporzione 1:2:3. Questo significa che, se il Mana generato passivamente dal possesso di x token è $1x$, quello ottenuto delegando la stessa quantità di token è $2x$, mentre quello ottenuto facendone lo staking come validatore è $3x$. Le ricompense sono adeguate in quanto aumentano proporzionalmente alla "posta in gioco" messa e al contributo fornito.

2.5 Differenze con altri ledger

Confrontiamo ora più direttamente IOTA con altre strutture di blockchain, rivedendone i concetti chiave. Ci sono molti fattori a differenziare IOTA dagli altri ledger esistenti, e che lo portano ad essere un passo importante verso l'evoluzione dei sistemi di transazioni digitali e criptovalute.

2.5.1 Layer 1

La differenza principale, come visto all'inizio di questo capitolo, consiste nella struttura stessa del ledger che si distacca dalla classica struttura di blockchain nel suo Layer 1. Come infatti illustrato in Figura 2.5, con la sua struttura a DAG, IOTA sostituisce il problema principale di scalabilità delle classiche blockchain, che sono invece limitate a poter aggiungere un solo blocco alla volta alla chain[15].

Come già visto però, rimuovendo il limite di inserimento di una transazione alla volta il DAG porta anche a uno stato del ledger non oggettivo, che in un determinato momento può non essere visto allo stesso modo da due nodi diversi. Questo ha reso naturale l'adozione dell'UTXO ledger model

[15] `https://archive.is/cZqhQ`

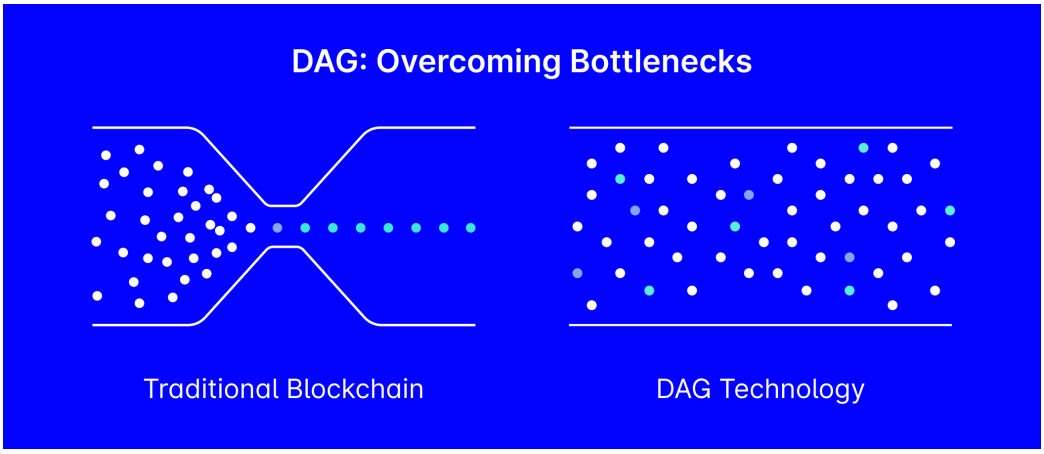


Figura 2.5: La scalabilità del DAG risolve il bottleneck delle classiche blockchain[16]

(vedi 2.2), al contrario di altri ledger basati su blockchain, che spesso (ma non sempre[17]) si affidano al modello basato su account. Questo modello, al contrario dell'UTXO ledger model che non ha uno stato oggettivo, è formato da account che sono entità atomiche con dei bilanci. Queste possono essere aggiornate soltanto una per volta, rendendo lo stato del ledger oggettivo [3]. Il metodo delle blockchain tradizionali di aggiungere un blocco alla volta introduce anche un'altra caratteristica tipica in cui IOTA si differenzia: il metodo di inserimento dei blocchi. Tipicamente infatti, nella rete di una blockchain sono presenti delle autorità centrali a cui bisogna rivolgersi per potersi far aggiungere le transazioni nei blocchi da inserire nella chain. In IOTA invece, ogni nodo ha la possibilità di inserire manualmente le proprie transazioni in qualsiasi momento.

Un'altra differenza importante, su cui IOTA basa la sua politica, è l'assenza di tasse e di mining di token. IOTA infatti si impegna a non imporre alcuna tassa per la creazione di transazioni, come invece accade tipicamente nei

[17]Bitcoin, anche se ha la struttura di una blockchain classica, utilizza UTXO ledger model. Ethereum invece utilizza il modello basato su account.

[17]Fonte: `https://archive.is/cZqhQ`

ledger distribuiti. Ne permette invece la creazione liberamente senza costi aggiuntivi, implementando poi con IOTA 2.0 uno strumento per rafforzare questo metodo: l'utilizzo di Mana per creare transazioni, ottenibile semplicemente partecipando alla rete. Come detto, in IOTA è inoltre assente anche il concetto di mining di nuovi token, in quanto tutti i token IOTA in circolazione vengono creati in un'unica volta alla creazione della rete, dopo la quale non è più possibile coniarne o distruggerne. Questo evita l'altrimenti inevitabile fenomeno di inflazione dei token, comune nelle blockchain in cui i nodi effettuano regolarmente mining per coniare nuovi token.
Con IOTA 2.0 e l'introduzione di un comitato di consenso, emergono ulteriori differenze influenti rispetto alla gestione delle autorità di validazione negli altri ledger.

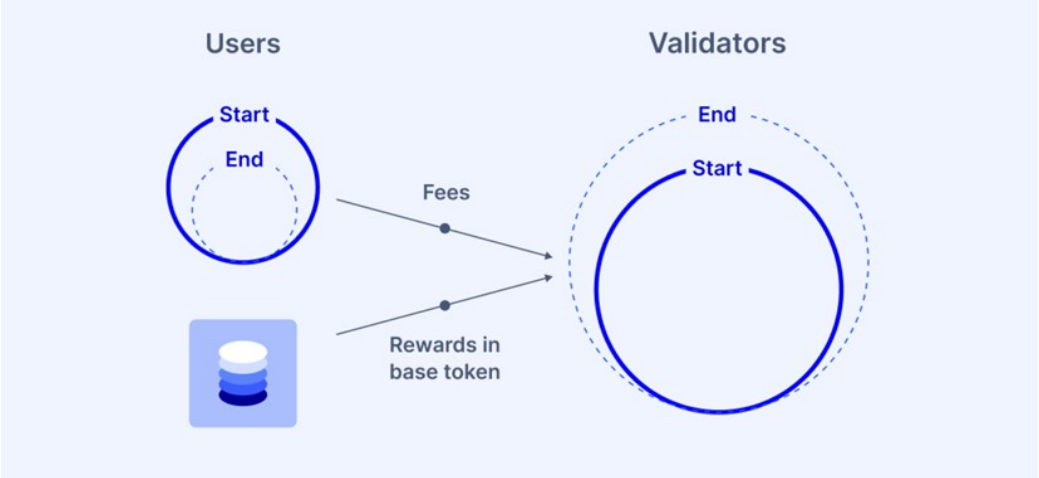


Figura 2.6: Come tasse e ricompense in token influiscono sullo sbilanciamento dell'economia nella blockchain[18]

Come infatti viene illustrato in Figura 2.6, solitamente in una blockchain i nodi di validazione vengono costantemente premiati tramite token, sia con le tasse pagate dai nodi partecipanti sia con le ricompense ottenute tramite validazione della rete. Questo porta naturalmente a uno sbilanciamento sem-

[18]Fonte: [4]

pre crescente dell'economia. È inevitabile infatti l'accumulo della ricchezza nelle mani di pochi, rendendo invece sempre più a corto la maggior parte dei normali utilizzatori della rete. IOTA invece propone un sistema, quello del Mana, che offre comunque meritati vantaggi alle autorità di validazione, ma senza basarsi su trasferimenti della valuta del ledger. In questo modo, l'economia rimane bilanciata e sostenibile per tutti.

2.5.2 Layer 2

Le chain del Layer 2, che hanno invece la classica struttura di una blockchain in cui viene inserito un blocco alla volta, adottano come modello del ledger il modello basato su account, come Ethereum.
Sempre come Ethereum, una chain L2 di IOTA ha il vantaggio di poter accedere direttamente all'intero stato della chain. Questo perché lo stato delle blockchain in L2 non contiene soltanto le transizioni per ricostruirlo, ma memorizza direttamente tutti i dati chiave-valore dello stato. Questo non è il caso ad esempio di Bitcoin, in cui per ottenere e ricostruire lo stato è necessario risalire i blocchi della blockchain applicando le transizioni [5]. Inoltre, IOTA utilizza anche un metodo differente per memorizzare l'hash dello stato. Mentre in blockchain come Bitcoin o Ethereum in un blocco viene salvato il *Merkle Tree*[19] dello stato, le chain L2 di IOTA utilizzano una versione più efficiente chiamata Verkle Tree. Tale versione utilizza un diverso schema crittografico rispetto alla funzione di hash del Merkle Tree, chiamato Vector Commitment, che permette di occupare meno spazio.
Un'altra sostanziale differenza delle chain in L2 riguarda il funzionamento della validazione. IOTA infatti si differenzia da altre blockchain, come Bitcoin ed Ethereum, adottando un metodo di validazione basato sul consenso

[19]Un albero di hash per memorizzare dati multipli. Si parte dall'hash dei dati singoli, che formano le foglie dell'albero, e si esegue l'hash delle loro somme a coppia di due (o più, nel caso di Ethereum sono 16) costruendo così l'albero, fino a raggiungere un unico hash radice [6].

piuttosto che sulle Proof of Work. Le differenze principali riguardano questi due aspetti:

1. **Selezione dei validatori.** In un sistema di validazione basato su Proof of Work, chiunque può partecipare alla validazione della blockchain, a patto di disporre di sufficiente potenza computazionale. In IOTA invece, viene effettuata un'effettiva selezione[20], tramite la quale verrà scelto un ristretto gruppo di nodi che agiranno da validatori[21]. In questo modo, sebbene la possibilità per gli utenti di partecipare alla validazione sia limitata dalla necessità di essere selezionati, questa selezione garantisce validatori fidati e quindi una maggiore sicurezza.

2. **Metodo di validazione.** In un sistema di validazione basato su Proof of Work, il validatore[22] si occupa direttamente della creazione del nuovo blocco. La metrica su cui si basa la selezione del blocco da aggiungere è la potenza computazionale impiegata dal validatore per crearlo. Nel sistema basato sul consenso implementato da IOTA invece, il contenuto del blocco è deciso in automatico dalla VM, e i validatori si limitano a esprimere il proprio voto su di esso e sui dati che la VM deve prendere in input per crearlo. La scelta del blocco da aggiungere è quindi basata sulla collaborazione dei validatori e sul consenso che raggiungono.

I vantaggi apportati da questo diverso metodo influiscono positivamente sulla sicurezza (e sulla sua scalabilità) del sistema. Per manomettere un sistema basato su Proof of Work è infatti sufficiente accumulare, anche singolarmente, una potenza computazionale che superi quella degli altri validatori. Per questo motivo, con la crescita di un sistema del genere e l'aumento della probabilità di attacchi, si cerca di avere sempre più validatori per aumentare la sicurezza della rete. Tuttavia, a lungo andare, questo causa problemi di

[20] È comunque possibile, se dovesse essercene bisogno, implementare in una chain un sistema di validazione in cui chiunque è libero di partecipare.

[21] Come visto in precedenza, il comitato di validazione.

[22] Indicato più comunemente come minatore nel sistema PoW.

scalabilità. In IOTA invece, oltre alla maggiore sicurezza garantita dalla necessità di essere selezionati e verificati per diventare validatori, manomettere la validazione è molto più complesso e richiederebbe un accordo fra molte entità. Come già detto in precedenza sarebbe infatti necessario che più di $\frac{2}{3}$ di validatori si mettano d'accordo per falsificare la firma di validazione[23]. Questa cosa, in un contesto in cui solitamente i validatori non si conoscono o possono addirittura essere in competizione fra di loro, è molto poco probabile. È poco probabile allo stesso modo anche in un contesto in cui i validatori sono in collaborazione e si fidano reciprocamente, per la natura stessa di fiducia di questo rapporto.
Grazie a questo diverso meccanismo di validazione inoltre, IOTA previene anche un fenomeno comune nelle altre blockchain, chiamato Miner Extractable Value (MEV). Questo fenomeno si verifica quando un validatore di un sistema basato su Proof of Work (minatore) analizza il flusso di transazioni in arrivo. Successivamente, poiché è lui stesso a creare il blocco, inserisce transazioni proprie che gli garantiscono un vantaggio economico in base a ciò che ha analizzato. Questo non è chiaramente possibile nelle chain L2 di IOTA, poiché appunto un validatore esprime soltanto il proprio voto sul blocco, che viene prodotto automaticamente dalla Virtual Machine.

2.6 Una nuova strada: IOTA Rebased

Durante la stesura di questa tesi, IOTA Foundation ha annunciato un nuovo approccio al loro sistema di ledger, che ne rivoluzionerà nuovamente il funzionamento. A novembre 2024, infatti, è stata pubblicata una proposta contenente modifiche importanti al ledger. Questa proposta è stata poi sottoposta a una votazione da parte della community ed è stata infine approvata a dicembre 2024 [7]. L'obiettivo principale è quello di integrare gli smart contract direttamente dentro al Layer 1.
L'approccio pensato per sviluppare IOTA Rebased è quello di includere di-

[23] O $\frac{1}{3}$ che si mettano d'accordo per impedire la produzione di una firma valida.

rettamente in L1 MoveVM, una Virtual Machine che permette di passare dall'UTXO ledger model che ha sempre avuto IOTA a un modello object-oriented. In questo modo, viene reso possibile il funzionamento di smart contract anche complessi all'interno del Layer 1 stesso, senza dover necessitare del Layer 2.
Inoltre, IOTA Rebased prevede dei cambiamenti alle politiche principali di IOTA, ovvero quelle che prevedevano un sistema libero da tasse e senza coniazione di nuovi token. Viene infatti rimosso l'utilizzo di Mana, e le ricompense per validatori e delegatori vengono distribuite direttamente in token IOTA. I token per le ricompense verranno coniati ad ogni epoca, in una quantità costante, così da influire sempre meno sull'inflazione all'aumentare del numero totale di token. Per ridurre ulteriormente il rischio di inflazione e data anche l'eliminazione del Mana, le transazioni richiederanno una piccola tassa, il cui pagamento verrà bruciato. Tuttavia, la presenza di tasse è bilanciata da ricompense per lo staking significativamente più alte, che garantirebbero guadagni superiori rispetto alla piccola quantità spesa in tasse.
Dopo l'approvazione da parte della community a dicembre, una testnet è già stata pubblicata e gli sviluppi per portare IOTA Rebased al completamento sono iniziati a tutti gli effetti.

2.6.1 Programmi futuri

IOTA Rebased è un progetto ancora agli albori, che promette varie aggiunte future migliorative. La più ambiziosa è sicuramente la volontà di introdurre multiple VM all'interno del ledger, affiancate alla principale MoveVM. In questo modo verrebbe fornito un elevato grado di universalità nel supporto agli smart contract scritti in diversi linguaggi. La Virtual Machine più ambita da portare è sicuramente la EVM utilizzata nel Layer 2. La sua integrazione permetterebbe definitivamente di includere in L1 anche tutti gli smart contract attualmente utilizzati in L2, creando un unico layer. Fino ad allora, il Layer 2 continuerà ad esistere e ad essere supportato [8].

Capitolo 3

Analisi progettuale

Questa tesi affronta lo sviluppo di uno Smart Contract IOTA, con lo scopo di analizzare vantaggi e complessità portati dallo sviluppo di applicazioni su piattaforme decentralizzate. In particolare, lo sviluppo sul ledger di IOTA metterà in risalto le prestazioni del multi-ledger introdotto dalla IOTA Foundation. Viene inoltre sviluppata anche un'applicazione esterna che si interfaccia con lo smart contract, in modo da analizzare anche i possibili utilizzi reali di questi programmi e come possono essere resi disponibili e utilizzati dagli utenti finali.
Questo capitolo illustra le scelte di progettazione per lo sviluppo dello smart contract e della sua applicazione esterna.

3.1 Analisi delle funzionalità globali

Lo smart contract che si è deciso di sviluppare deve permettere la creazione e la gestione di un sistema di lotteria. Un'entità, che corrisponde al creatore della lotteria, svolge il ruolo di gestore della lotteria. Creare una lotteria equivale ad effettuare il deploy di un'istanza dello smart contract. Al momento della creazione della lotteria, il gestore decide quanti e quali biglietti rendere disponibili e seleziona il biglietto vincente. Dopo la creazione può poi effettuare le seguenti operazioni di amministrazione e monitoraggio:

- Aggiungere un nuovo biglietto ai disponibili
- Modificare il biglietto vincente
- Monitorare lo stato dei biglietti, potendo visualizzare quanti e quali sono stati venduti
- Terminare la lotteria per porre fine all'acquisto di biglietti

Una volta terminata la lotteria, al gestore spettano dei guadagni in token IOTA. Questi corrispondono al 30% dei ricavi totali dalla vendita dei biglietti e possono essere liberamente ritirati dal gestore in qualsiasi momento.
Qualsiasi altro utente avrà invece il ruolo di partecipante alla lotteria, potendo acquistare un biglietto a un prezzo fisso di 10 IOTA. Una volta terminata la lotteria, il partecipante ha modo di visualizzare se il biglietto da lui acquistato è quello vincente. In caso positivo, può ritirare la sua vincita in token IOTA, equivalente al 70% dei ricavi totali dalla vendita dei biglietti.
In Figura 3.1 è riportato lo schema principale che illustra gli scambi di denaro della lotteria.

3.2 Progettazione dello Smart Contract

Il codice dello smart contract deve quindi fornire tutte le funzioni necessarie per effettuare le operazioni sopra elencate. Il richiedente delle funzioni viene autenticato tramite il proprio indirizzo del ledger. La gestione del denaro è effettuata direttamente dallo smart contract, utilizzando il suo bilancio interno per immagazzinare e distribuire i token.
Come linguaggio per lo sviluppo è stato scelto *Solidity*, un linguaggio di programmazione ideato per lo sviluppo di smart contract su Ethereum. Dato che la VM di IOTA implementa il modello della EVM di Ethereum, gli smart contract scritti in questo linguaggio sono completamente compatibili anche con il ledger di IOTA. Per la compilazione del codice dello smart contract si è scelto di utilizzare *Remix*, un IDE online che permette di programmare,

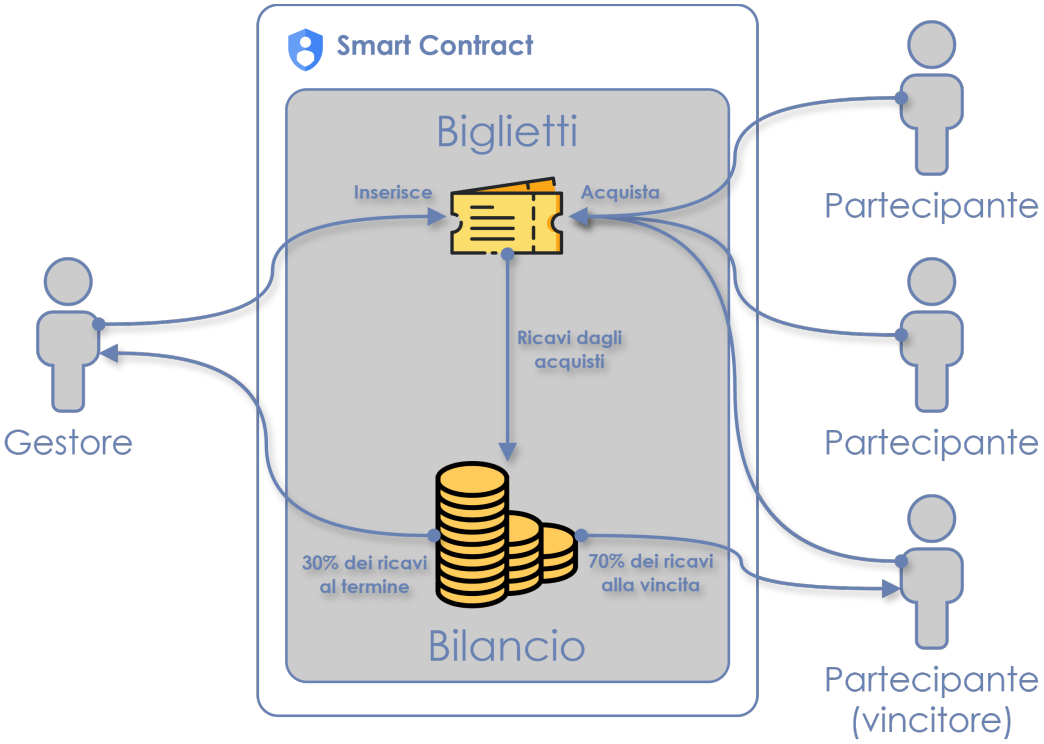


Figura 3.1: Schema del processo di lotteria e di pagamenti

compilare, fare il deploy e interagire con smart contract.
Il deploy viene effettuato su una *testnet* di IOTA fornita dalla fondazione a scopi di test. Al suo interno è infatti possibile ottenere qualsiasi quantità di token IOTA fittizi tramite *faucet*. Per collegarsi alla testnet viene utilizzato *MetaMask*, un'estensione del browser che permette di connettersi a reti di ledger distribuiti e di creare e gestire wallet con multipli account.

3.3 Progettazione dell'applicazione esterna

L'applicazione esterna deve permettere agli utenti di interagire tramite interfaccia grafica con le funzioni dello smart contract, potendone modificare e visualizzare lo stato. È stato scelto di sviluppare un'applicazione web utilizzando *React* e *Typescript*. Come libreria per interfacciarsi alla rete di IOTA e allo smart contract sono state considerate sia *ethers.js* che *web3.js*. È stato infine scelto di utilizzare web3.js data la sua maggiore rilevanza storica e supporto sviluppato nel tempo. web3.js è infatti la prima e originale libreria per l'interazione con smart contract, sviluppata direttamente dalla fondazione Ethereum[1].
Tale libreria viene poi fatta interfacciare con MetaMask, utilizzandolo come provider (vedi 4.2.1) per collegarsi alla rete IOTA in cui è presente lo smart contract.
L'applicazione fornisce due schermate principali: la schermata gestore e la schermata partecipante.

- **Schermata gestore** La schermata visualizzata da chi ha creato la lotteria. Gli permette di interagire con tutte le funzioni di amministrazione e monitoraggio dello smart contract, e di ritirare i suoi guadagni una volta conclusa la lotteria.

- **Schermata partecipante** La schermata visualizzata da tutti gli altri utenti. Permette a un utente di partecipare alla lotteria acquistando un biglietto, e di ritirare la sua vincita in caso di vittoria.

È inoltre prevista una schermata iniziale di accesso, che permette la creazione di una nuova lotteria oppure l'accesso a una già esistente.
In Figura 3.2 è riportata la sequenza di operazioni dell'applicazione finale per richiamare le funzioni dello smart contract.

[1] `https://web3js.org/`

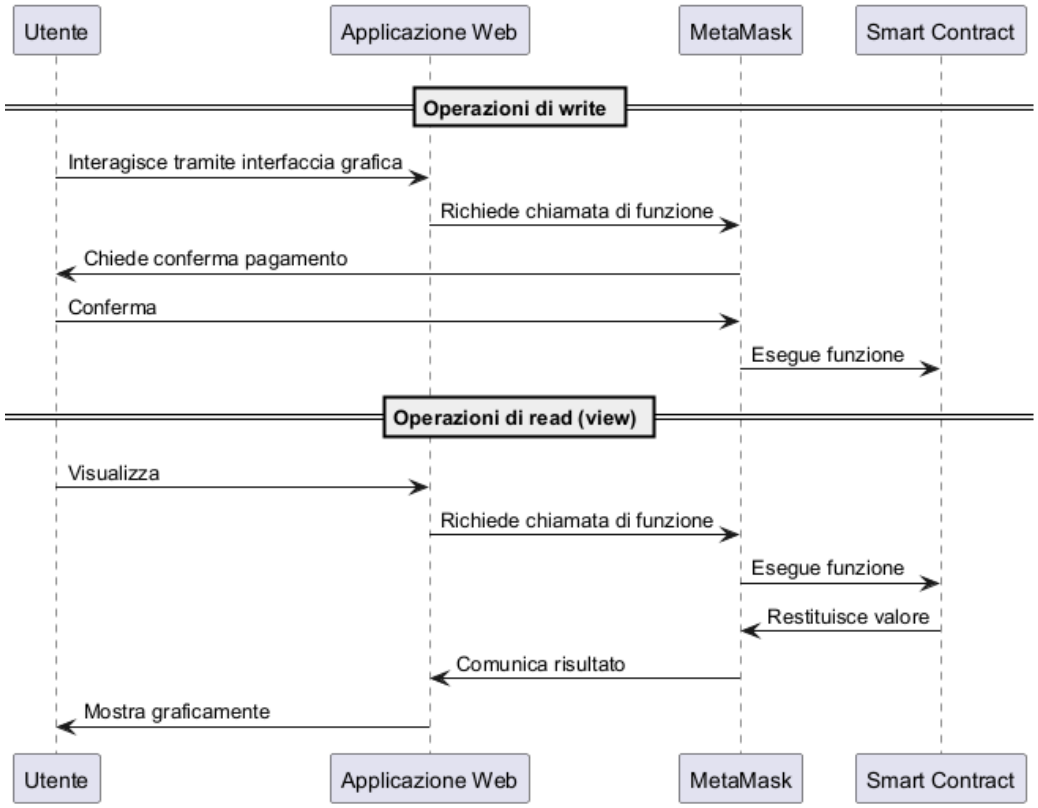


Figura 3.2: Diagramma di sequenza sulle interazioni fra utente, applicazione web e smart contract

Capitolo 4

Implementazione

In questo capitolo viene descritta nel dettaglio l'implementazione del progetto, analizzando le scelte tecniche effettuate e le difficoltà riscontrate. Il processo di implementazione è avvenuto creando una versione iniziale dello smart contract, per poi portare avanti il suo sviluppo in parallelo con quello dell'applicazione web, in base alle necessità riscontrate durante l'implementazione di quest'ultima.
La prima sezione è dedicata alla descrizione del processo di sviluppo dello smart contract, analizzando la logica di implementazione delle sue funzionalità e le soluzioni adottate per garantire efficienza e sicurezza.
Successivamente, viene illustrato lo sviluppo dell'applicazione web, soffermandosi sull'interazione con lo smart contract e sugli strumenti necessari per consentirla.
Il codice sviluppato per lo smart contract e per l'applicazione web è consultabile su GitHub[1], all'interno della cartella `project`.

4.1 Implementazione dello Smart Contract

Come appena introdotto, lo sviluppo dello smart contract ha seguito un processo di evoluzione graduale, portando allo sviluppo di più versioni del

[1] `https://github.com/UniboSecurityResearch/Paris_Manuel_BT`

programma.

1. **La prima versione** implementava la struttura base del sistema di lotteria, senza l'integrazione del sistema di pagamento.

2. **La seconda versione**, sviluppata subito dopo, incorporava nel sistema di lotteria il meccanismo di pagamento dei biglietti e di riscossione dei guadagni.

3. **La terza ed ultima versione**, sviluppata in contemporanea con l'implementazione dell'applicativo web, include modifiche e aggiunte alle funzioni dello smart contract necessarie al corretto funzionamento dell'applicazione.

4.1.1 Prima versione: struttura base

La prima fase di sviluppo è stata puramente dedicata a creare la struttura base dello smart contract e a renderlo funzionante e interagibile su blockchain. In questa versione un'entità può creare una lotteria effettuando il deploy dello smart contract, e può eseguire le funzioni amministrative e di monitoraggio elencate nel capitolo precedente. Un partecipante può ottenere un biglietto (senza effettuare un pagamento) e al termine della lotteria visualizzare se ha vinto.
I biglietti vengono memorizzati all'interno di un array `tickets` e sono identificati in due modi differenti:

1. Un ID numerico, ossia il valore con cui un biglietto viene salvato nell'array. Questo viene utilizzato principalmente per distinguere graficamente i biglietti fra di loro.

2. L'indice della posizione del biglietto nell'array. Questo è invece quello che viene utilizzato per identificare i biglietti nel codice.

Il biglietto vincente è identificato attraverso l'indice del biglietto nell'array, che viene memorizzato in una variabile `winning_ticket_id`. Lo stato della

lotteria invece viene gestito tramite la variabile `is_lottery_open`, un booleano che indica se la lotteria è ancora aperta o se è stata chiusa dal gestore. Un grande vantaggio della programmazione di smart contract è che essi rendono l'autenticazione un processo molto semplice da implementare e allo stesso tempo sicuro grazie al ledger. All'interno dello smart contract infatti, ciascuna entità viene autenticata tramite il proprio indirizzo nel ledger. Al deploy dello smart contract l'indirizzo del creatore viene registrato in una variabile chiamata `lottery_manager`. Questa variabile sarà quella utilizzata nel codice per autenticare l'indirizzo del chiamante, verificando se coincide con quello del gestore della lotteria. Per autenticare un partecipante invece è utilizzato un mapping `participants` che associa un indirizzo a una struct `Participant`. Questa struct possiede i seguenti campi:

- `ticket_id`: indice della posizione nell'array del biglietto preso dal partecipante
- `got_ticket`: booleano che indica se il partecipante ha preso un biglietto o meno. Questo campo è necessario poiché il semplice controllo che il campo `ticket_id` non sia impostato, non è sufficiente. Il motivo è che la mappatura iniziale del mapping associa a un indirizzo una struct con variabili inizializzate a 0, in questo caso quindi `ticket_id = 0` e `got_ticket = false`. Non sarebbe quindi possibile effettuare il controllo su `ticket_id` poiché il primo biglietto ha indice 0.

Sono stati inoltre definiti dei *modificatori* per semplificare l'autenticazione e la sicurezza nel codice. In Solidity, un modificatore serve ad aggiungere dei controlli aggiuntivi ad una funzione. Applicando un modificatore ad una funzione, verranno prima effettuati i controlli previsti dal modificatore, e solo in caso di esito positivo viene eseguita la funzione. Nel Codice 4.1 è mostrato un esempio di come è stato applicato un modificatore nel codice.

```
1 // Dichiarazione del modificatore
2 modifier is_participant() {
3     require(participants[msg.sender].got_ticket, "You didn't
        get a ticket yet.");
```

```
4       _;
5   }
6   // Modificatore applicato alla funzione
7   function check_ticket() public view is_participant returns (
        uint) {
8       Participant storage participant = participants[msg.sender
            ];
9       return (tickets[participant.ticket_id]);
10  }
```

Codice 4.1: Utilizzo dei modificatori

In questo estratto di codice, viene definito il modificatore `is_participant` che verifica che il chiamante sia un partecipante alla lotteria in possesso di un biglietto. Applicandolo alla funzione `check_ticket` che restituisce il biglietto posseduto dal chiamante, la funzione restituirà soltanto se il chiamante è effettivamente in possesso di un biglietto, altrimenti fallirà l'esecuzione. La logica di esecuzione è che il corpo della funzione viene inserito nel modificatore al posto dell'apposito indicatore, qui specificato in riga 4. Viene poi eseguito il modificatore con all'interno il corpo della funzione.
I modificatori definiti per questo smart contract sono i seguenti:

- `manager_rights`: controlla che il chiamante sia il gestore della lotteria, comparando il suo indirizzo con quello salvato in `lottery_manager`.
- `is_participant`: come riportato nel Codice 4.1, controlla che il chiamante sia un partecipante attivo alla lotteria. In altre parole viene controllato che, tramite il mapping `participants`, l'indirizzo del chiamante sia associato ad un `Participant` con `got_ticket = true`.
- `lottery_open`: controlla che la lotteria sia ancora aperta, tramite l'omonima variabile.
- `lottery_closed`: analogo al precedente, controllando l'inverso della variabile `lottery_open`.

Di seguito sono invece descritti i metodi di implementazione delle funzionalità che deve svolgere lo smart contract.

Deploy e creazione della lotteria

Lo smart contract ha una funzione costruttore che viene chiamata al momento del deploy. È quindi quella che viene chiamata alla creazione della lotteria. Il costruttore prende in input i seguenti parametri:

- `new_tickets`: array contenente i biglietti iniziali della lotteria. A questo array sarà successivamente possibile aggiungere biglietti, ma non rimuoverne.
- `new_winning_ticket_id`: indice corrispondente alla posizione del biglietto vincente nell'array, che potrà successivamente essere modificato.

Dopo aver verificato che l'indice fornito sia nel range di lunghezza dell'array, il costruttore assegna i parametri alle corrispondenti variabili dello smart contract: `tickets` e `winning_ticket_id`. Assegna poi alla variabile `lottery_manager` il valore di `msg.sender`, che corrisponde all'indirizzo dell'attuale chiamante della funzione. In questo modo viene permanentemente registrato come gestore della lotteria colui che ha effettuato il deploy.

Gestore: amministrazione della lotteria

Per permettere al gestore della lotteria di eseguire le funzionalità di amministrazione e monitoraggio sono state definite le seguenti funzioni, che utilizzano tutte il modificatore `manager_rights` in modo da poter essere chiamate soltanto dal gestore:

- `close_lottery`: permette di chiudere la lotteria impostando `is_lottery_open = false`. Utilizza anche il modificatore `lottery_open` per fare in modo che la funzione sia chiamabile soltanto se la lotteria non è già stata chiusa.
- `add_ticket`: permette di aggiungere un nuovo biglietto a quelli disponibili. Prende in input il parametro `new_ticket` contenente l'ID numerico del biglietto, e ne esegue il push in `tickets`. Analogamente alla precedente funzione utilizza anche il modificatore `lottery_open`.

- `change_winning_ticket`: permette di modificare l'indice del biglietto vincente. Prende in input il parametro `new_winning_ticket_id` contenente il nuovo indice del biglietto vincente, che viene assegnato alla rispettiva variabile `winning_ticket_id`. Analogamente alle funzioni precedenti viene utilizzato il modificatore `lottery_open`. È presente inoltre un controllo aggiuntivo in cui viene verificato che il nuovo indice sia valido, ovvero che rientri nella lunghezza massima dell'array `tickets`.

- `check_tickets` *(view)*: permette di visualizzare l'array di tutti i biglietti presenti nella lotteria, restituendo `tickets`.

- `check_winning_ticket` *(view)*: permette di visualizzare l'indice del biglietto vincente, restituendo `winning_ticket_id`.

- `check_ticket_allotment` *(view)*: permette di visualizzare il numero di biglietti venduti, restituendo la variabile `ticket_allotment` (descritta nel prossimo paragrafo).

Partecipante: ottenimento del biglietto e controllo della vincita

Per permettere a un utente che vuole partecipare alla lotteria di acquisire un biglietto e verificare la vincita, sono state definite le seguenti funzioni:

- `get_ticket`: permette di ottenere un biglietto fra quelli disponibili. L'assegnazione dei biglietti è stata implementata in modo sequenziale. Viene per questo utilizzata una variabile `ticket_allotment` inizializzata a 0 e incrementata ad ogni chiamata di questa funzione, che tiene traccia dell'indice del prossimo biglietto da assegnare[2]. Si utilizza quindi il mapping `participants` per ottenere l'istanza della struct `Participant` associata all'indirizzo del chiamante. Tale istanza viene impostata con `got_ticket = true` e `ticket_id = ticket_allotment`. Dopodiché `ticket_allotment` viene incrementata di 1 per la prossima

[2] Allo stesso tempo, questa variabile corrisponde anche al numero di biglietti assegnati.

assegnazione. Viene utilizzato il modificatore `lottery_open` in modo da permettere l'acquisto solamente se la lotteria è aperta. Sono inoltre presenti i seguenti controlli aggiuntivi:

  - Il chiamante non può essere il gestore, `msg.sender` deve quindi essere diverso da `lottery_manager`.
  - Devono esserci ancora biglietti disponibili, `ticket_allotment` deve quindi essere minore della lunghezza di `tickets`.
  - Il chiamante non deve aver già acquistato un biglietto, il campo `got_ticket` dell'istanza di `Participant` associata all'indirizzo `msg.sender` deve quindi essere `false`.

- `check_ticket` *(view)*: come mostrato anche nel Codice 4.1, permette di visualizzare l'ID numerico del proprio biglietto. Restituisce quindi `tickets[ticket_id]`, dove `ticket_id` è l'indice del biglietto che è stato precedentemente assegnato al chiamante tramite `get_ticket`. La funzione utilizza il modificatore `is_participant` in modo da poter essere chiamata solo da chi possiede un biglietto.

- `check_winning` *(view)*: permette di visualizzare se il proprio biglietto è quello vincente. Restituisce quindi il booleano risultante da `ticket_id == winning_ticket_id`, dove `ticket_id` è l'indice del biglietto assegnato al chiamante. È utilizzato il modificatore `is_participant`, analogo alla funzione precedente. È inoltre utilizzato il modificatore `lottery_closed`, in modo da permettere di verificare la vincita solo se la lotteria è stata chiusa.

4.1.2 Seconda versione: acquisti e pagamenti

Dopo aver portato a termine la prima versione completa dello smart contract e aver testato le sue varie componenti in modo che fosse tutto funzionante, è cominciato lo sviluppo della seconda versione. Questa versione,

come introdotto prima, introduce nello smart contract il sistema di pagamento. Diventa quindi necessario pagare un prezzo fisso per l'acquisto di un biglietto, e alla fine della lotteria vengono distribuiti pagamenti al gestore e al vincitore della lotteria. Come introdotto nel capitolo precedente, è stato deciso un prezzo di 10 IOTA per l'acquisto di un biglietto, e una retribuzione per il gestore e per il vincitore di rispettivamente il 30% e il 70% dei guadagni totali dalle vendite dei biglietti. Sono stati scelti dei valori in modo che risultassero sempre numeri interi nella divisione dei guadagni, poiché una limitazione dello smart contract è che non può gestire numeri float.
I metodi di implementazione del sistema di pagamento sono descritti qui di seguito.

Pagamento dall'utente allo smart contract

I pagamenti effettuati dall'utente verso lo smart contract, ovvero gli acquisti dei biglietti, vengono effettuati direttamente tramite la transazione stessa. Questo significa che, quando l'utente chiama la funzione per acquistare il biglietto, inserisce la somma necessaria nella stessa transazione che contiene la chiamata. Il pagamento viene ricevuto automaticamente dallo smart contract, che verifica poi se la quantità di denaro fornita è adeguata. In caso contrario, viene annullata la chiamata e restituito il denaro.

Pagamento dallo smart contract all'utente

Per effettuare dei pagamenti dall'interno dello smart contract, in Solidity sono disponibili tre diverse funzioni, comparate qui di seguito in Tabella 4.1.

Funzione	Valore restituito	Gas utilizzato	Funzionalità aggiuntive
`transfer`	Niente in caso di successo, lancia un errore in caso di fallimento.	2300 gas	Nessuna
`send`	`true` in caso di successo, `false` in caso di fallimento.	2300 gas	Nessuna
`call`	`true` in caso di successo, `false` in caso di fallimento[3].	Si adatta dinamicamente al gas necessario per la chiamata e non ha limiti.	Permette anche di chiamare funzioni di altri smart contract.

Tabella 4.1: Funzioni di pagamento in Solidity[4]

La funzione scelta per implementare i pagamenti dallo smart contract all'utente in questo progetto è `call`. Tale funzione è quella attualmente più utilizzata, data la sua flessibilità e la sua gestione dinamica e senza limiti del gas. Questo introduce però allo stesso tempo anche dei rischi nell'utilizzo, se non viene gestita con i dovuti accertamenti. Non avendo infatti limiti di gas e permettendo anche di chiamare funzioni di altri smart contract, c'è un rischio di abuso da parte del ricevente. Se il ricevente è uno smart contract, potrebbe infatti chiamare più volte ricorsivamente la funzione del nostro smart contract che lancia la `call`, ricevendo così il pagamento multiple volte. Per evitare questo comportamento è necessario strutturare la

[3]Quando si utilizza per chiamare funzioni di altri smart contract, restituisce un parametro aggiuntivo contenente l'output della chiamata.

[4]`https://dev.to/ceasermikes002/understanding-send-transfer-and-call-in-solidity-security-implications-and-preferences-4pog`

funzione in modo che *prima* vengano effettuati i controlli e le modifiche allo stato, e come ultima operazione la `call`. Un esempio del corretto utilizzo è mostrato nel Codice 4.2, che riporta l'implementazione della funzione di pagamento al vincitore (descritta successivamente in 4.1.2).

```
1  function cash_in_victory() public is_participant
       lottery_closed {
2      // Controlli
3      Participant storage participant = participants[msg.sender
           ];
4      require(participant.ticket_id == winning_ticket_id, "Your
           ticket is not the winning ticket");
5      require(!cashed_in, "Winning already cashed in.");
6      if(earnings_retrieved)
7          winning_amount = address(this).balance;
8      else
9          winning_amount = (address(this).balance * 7) / 10;
10     // Modifiche allo stato
11     cashed_in = true;
12     // Pagamento con call
13     (bool success, ) = payable(msg.sender).call{value:
           winning_amount}("");
14     require(success, "Error cashing in");
15 }
```

Codice 4.2: Corretto utilizzo di `call`

In questo modo, anche se tramite la `call` uno smart contract ricevente dovesse nuovamente chiamare `cash_in_victory` prima che termini, la seconda chiamata non andrebbe mai a buon fine. Questo perché, come prima cosa, la funzione controlla che il campo `cashed_in` non sia `true` (riga 5), valore che viene impostato *prima* del lancio di `call`[5] (riga 11). Inoltre, essendo `call` priva di un lancio automatico di errore in caso di fallimento, è necessario gestirlo automaticamente tramite il booleano restituito dalla funzione (riga 14).

[5]Se venisse impostato dopo, il ricevente potrebbe richiamare `cash_in_victory` e ricevere nuovamente un pagamento prima che venga impostato `cashed_in = true`.

Aggiunte alle funzioni dello smart contract

Per implementare il pagamento dei biglietti, sono state effettuate le seguenti modifiche alla funzione `get_ticket`:

- È stata adeguatamente rinominata in `buy_ticket`.
- È stata resa una funzione `payable`. È grazie a questo che lo smart contract può ricevere dei pagamenti attraverso la transazione di chiamata a questa funzione.
- È stato aggiunto un controllo aggiuntivo che richiede un pagamento tramite la transazione di chiamata di esattamente 10 IOTA. Tale valore è verificabile in `msg.value`. In Solidity, l'unità utilizzata dallo smart contract per gestire i bilanci è *Wei*[6]. È stato quindi necessario, nel controllo che verifica il valore del pagamento, specificare `10 ether`[7].

Inoltre, è stato aggiunto un controllo aggiuntivo alla funzione `close_lottery`, che permette la chiusura della lotteria soltanto con le seguenti condizioni:

- Nessun biglietto è ancora stato venduto. In questo caso la lotteria viene considerata annullata.
- Se dei biglietti sono stati venduti, allora è necessario che ne siano stati venduti almeno due e che il biglietto vincente sia fra quelli venduti. La prima condizione è necessaria per fare in modo che il vincitore della lotteria abbia un guadagno dalla vincita. Se infatti viene chiusa la lotteria con soltanto un biglietto acquistato, il possessore riceverà una vincita pari al 70% del prezzo dell'unico biglietto, finendo quindi per perdere il 30% di ciò che ha pagato. La seconda condizione è invece necessaria per evitare frodi da parte del gestore, assicurandosi che la lotteria

[6]Una sottounità di Ether, la valuta di Ethereum. 1 Wei equivale a 10^{-18} Ether.

[7]La specifica è in Ether e non in IOTA perché Solidity è un linguaggio inizialmente sviluppato per Ethereum. In una blockchain IOTA, la valuta Ether nel codice viene considerata come IOTA.

non venga chiusa senza che il biglietto vincente sia stato venduto a qualcuno.

Sono infine state aggiunte le seguenti funzioni per permettere la riscossione dei guadagni da parte del gestore e del vincitore:

- `cash_in_victory`: mostrata anche nel Codice 4.2, questa funzione permette al vincitore della lotteria di riscattare la sua vincita. Controlla innanzitutto che il chiamante sia effettivamente il vincitore, comparando il campo `ticket_id` del `Participant` associato al chiamante con la variabile `winning_ticket_id` dello smart contract. Verifica poi che il vincitore non abbia già riscattato la sua vincita, utilizzando una variabile booleana `cashed_in` inizializzata a `false`. In questo modo vengono impedite riscossioni multiple del denaro. Dopodiché, per decidere la quantità di bilancio dello smart contract da pagare al vincitore, viene controllato se il gestore ha già riscattato i suoi guadagni. Questo avviene tramite una variabile booleana `earnings_retrieved` inizializzata a `false`, analoga a `cashed_in` ma relativa al gestore. Se li ha già riscattati, al vincitore spetta tutta la parte rimanente del bilancio dello smart contract, altrimenti il 70%. In seguito ai controlli e prima di inviare il denaro, viene impostata `cashed_in = true`. Infine, viene lanciata l'effettiva `call` all'indirizzo di `msg.sender`, specificandolo come indirizzo `payable`, inviando la quantità di token appena calcolata. Verifica poi che il pagamento sia andato a buon fine, effettuando il revert della funzione in caso contrario. Questa funzione utilizza i modificatori `is_participant` e `lottery_closed`, in modo da permetterne la chiamata solo da chi possiede un biglietto e se la lotteria è già stata chiusa.

- `retrieve_earnings`: permette al gestore della lotteria di riscattare i suoi guadagni. Il funzionamento è speculare a `cash_in_victory` invertendo `cashed_in` e `earnings_retrieved`, e inviando una quantità di token del 30% dei ricavi invece che del 70%. Utilizza i modificatori

`manager_rights` e `lottery_closed`, in modo da poter essere chiamata solo dal gestore dopo il termine della lotteria.

4.1.3 Terza versione: integrazione con l'applicativo

Mentre le prime due versioni dello smart contract sono state sviluppate in modo autonomo rispetto all'applicazione web, la terza ed ultima versione è stata elaborata in contemporanea ad essa. Questo perché, incominciando l'effettivo sviluppo dell'applicazione web e delle sue interazioni con lo smart contract, si sono presentate necessità aggiuntive per lo smart contract che non erano state considerate durante il suo sviluppo autonomo. Durante lo sviluppo dello smart contract infatti, le funzioni venivano provate tramite l'interfaccia di Remix, che permette di visualizzare gli errori delle chiamate. Tali errori, come ad esempio quelli dati dai modificatori o da altri controlli aggiunti, venivano mostrati e differenziati fra loro senza problemi. Questo però, effettuato in un'applicazione destinata all'uso reale, non è ottimale in quanto viene sprecata una transazione che viene poi annullata a causa dell'errore. Questo causa infatti anche dei problemi quando si utilizza MetaMask per confermare la transazione, in quanto MetaMask calcola una stima di gas errata e molto elevata quando prevede che la transazione fallirà. Inoltre, gli errori personalizzati dello smart contract sono più difficili da catturare e gestire separatamente all'interno dell'applicazione esterna. Sono state quindi necessarie delle view aggiuntive per permettere all'applicazione, quando possibile, di fare i controlli *prima* di chiamare le funzioni. In questo modo è possibile impedire le chiamate a priori senza far scattare degli errori. Inoltre, ulteriori view sono state aggiunte anche per necessità grafiche e di controlli interni riscontrate nella creazione del layout dell'applicazione web, in modo da permettere agli utenti di visualizzare adeguatamente tutte le informazioni necessarie.
Le view aggiunte allo smart contract in questa versione finale sono le seguenti:

- `check_ticket_availability`: restituisce un booleano che indica se ci sono ancora biglietti disponibili o se sono esauriti. Per farlo controlla

se la variabile `ticket_allotment`, equivalente al numero di biglietti venduti, è minore della lunghezza di `tickets`. Non utilizza nessun modificatore ed è quindi chiamabile in qualsiasi situazione. Questa view è stata definita per permettere all'applicativo di controllare se ci sono ancora biglietti disponibili all'acquisto, prima di permettere di richiamare la funzione `buy_ticket`.

- `check_UUID`: definita in contemporanea a una nuova costante `contract_UUID` dello smart contract, contenente un UUID immutabile che identifica un'istanza di questo smart contract. Questa view prende in input il parametro `uuid` e confronta il suo hash[8] con quello della costante `contract_UUID`. Restituisce poi un booleano in base al risultato. Questa view è stata definita per permettere all'applicativo, quando l'utente prova ad accedere a una lotteria, di verificare che l'indirizzo fornito sia dell'istanza di uno smart contract lotteria. In questo modo, se tramite l'applicazione si tentasse di inserire l'indirizzo di uno smart contract che non è un'istanza lotteria, la chiamata a `check_UUID` non andrebbe a buon fine[9] e permetterebbe all'applicazione di gestire correttamente la cosa.

Sono inoltre state create anche delle variabili pubbliche aggiuntive, per memorizzare ulteriori dati necessari all'interfaccia grafica:

- `winning_amount`: memorizza la somma di denaro guadagnata dal vincitore. Questo avviene al momento della chiamata di `cash_in_victory`. È stata definita per permettere all'applicativo di mostrare graficamente al vincitore quanti IOTA ha vinto al momento del riscatto della vincita.

- `earned_amount`: analoga alla precedente, ma per la somma guadagnata dal gestore della lotteria.

[8] Ottenuto tramite `keccak256(abi.encodePacked(uuid))`

[9] Sia nel caso in cui la funzione non esista in quello smart contract, sia nell'eventualità in cui invece esista. È infatti quasi completamente impossibile che utilizzi lo stesso UUID.

- `lottery_name`: implementa la possibilità di dare un nome alla lotteria al momento della creazione. Questa variabile viene infatti registrata tramite un nuovo parametro `new_lottery_name` della funzione costruttore dello smart contract. È stata definita per permettere all'applicazione di visualizzare graficamente un nominativo per l'attuale lotteria, in modo da poter riconoscere a quale si sta partecipando nella schermata attuale.

Tali variabili non necessitano della creazione esplicita di una view che le restituisca, poiché per le variabili pubbliche viene automaticamente generata una view con l'omonimo nome. Rendere delle variabili pubbliche è quindi utile quando si necessita di una view che restituisca la singola variabile e che non richieda alcuna restrizione sul suo utilizzo. Per lo stesso motivo, alcune variabili che prima erano private e utilizzate solo per controlli interni al codice, in questa versione sono state rese pubbliche per permetterne la creazione di una view:

- `lottery_manager`: resa pubblica per permettere all'applicativo di comparare l'indirizzo dell'utente connesso e verificare se è il gestore della lotteria. In questo modo l'applicazione può decidere quale schermata mostrare all'utente.

- `is_lottery_open`: resa pubblica per permettere all'applicativo di mostrare graficamente lo stato della lotteria, e di controllare quali funzioni mostrare all'utente in base a quali è permesso effettuare nello stato attuale.

- `cashed_in`: resa pubblica per permettere all'applicativo di controllare se l'utente può richiamare la funzione `cash_in_victory` o se l'ha già fatto.

- `earnings_retrieved`: resa pubblica per permettere all'applicativo di controllare se il gestore può richiamare la funzione `retrieve_earnings` o se l'ha già fatto.

Un'altra implementazione che si è rivelata necessaria per la corretta integrazione con l'interfaccia grafica sono gli *eventi*. In Solidity, un evento è un segnale che viene emesso dallo smart contract in determinati punti dell'esecuzione del codice. Chiunque sia in ascolto di tali segnali, in questo caso l'applicazione web, può catturarli e gestirli in tempo reale. Sono stati quindi implementati per permettere all'applicazione di aggiornarsi in tempo reale in contemporanea a determinati cambiamenti dello stato dello smart contract. Nel Codice 4.3 è mostrato un esempio di implementazione di evento.

```
1 // Dichiarazione dell'evento
2 event ticket_published(uint ticket);
3 // Emissione dell'evento in una funzione
4 function add_ticket(uint new_ticket) public manager_rights
     lottery_open {
5     tickets.push(new_ticket);
6     emit ticket_published(new_ticket);
7 }
```

Codice 4.3: Utilizzo degli eventi

In questo estratto di codice, viene dichiarato l'evento `ticket_published` che prende in input il parametro `ticket`. Alla fine della funzione `add_ticket` viene emesso tale evento, utilizzando il biglietto appena aggiunto come input. In questo modo, chiunque sia in ascolto dell'evento `ticket_published` di questo smart contract, ogni volta che viene aggiunto un nuovo biglietto alla lotteria riceverà un segnale in tempo reale contenente l'identificativo numerico del biglietto.
Di seguito sono riportati gli eventi implementati nello smart contract:

- `closed_lottery`: viene emesso quando il gestore chiude la lotteria tramite `close_lottery`. È stato implementato per permettere all'applicativo di aggiornare in tempo reale i controlli relativi a quali funzioni sono chiamabili nello stato attuale della lotteria. Se non ci fosse questo evento e il gestore chiudesse la lotteria mentre un utente sta visualizzando l'interfaccia dell'applicazione, l'interfaccia non si aggiornerebbe

e all'utente sarebbe ad esempio permesso di chiamare la funzione per acquistare un biglietto, finendo per effettuare una transazione fallita.

- `ticket_published`: come visto nel Codice 4.3, viene emesso quando il gestore aggiunge un nuovo biglietto tramite `add_ticket`. È stato implementato per permettere all'applicativo di notificare in tempo reale gli utenti quando viene aggiunto un nuovo biglietto, e per aggiornare i controlli relativi ai biglietti disponibili. Se infatti i biglietti erano precedentemente esauriti, alla ricezione di questo evento è possibile rieffettuare il controllo per permettere nuovamente all'utente di chiamare la funzione `buy_ticket`.

- `ticket_registered`: viene emesso quando un utente acquista un biglietto tramite `buy_ticket`. Prende in input due parametri: `participant`, l'indirizzo di chi ha acquistato il biglietto, e `ticket`, l'identificativo numerico del biglietto. È stato implementato per permettere all'applicativo di notificare in tempo reale gli altri utenti quando qualcuno acquista un biglietto, e per aggiornare i controlli in modo analogo a `ticket_published`. L'assenza di questo evento potrebbe portare al tentativo di acquisto di un biglietto quando sono invece esauriti, risultando in una transazione fallita.

4.2 Implementazione dell'applicazione web

Il processo di sviluppo dell'applicazione web consiste invece in un'iniziale implementazione delle due schermate gestore e partecipante, che si concentra principalmente sul collegare correttamente l'applicazione allo smart contract. Durante questa fase di sviluppo viene usato un unico smart contract lotteria, effettuandone il deploy manualmente da Remix e inserendolo nel codice come costante. Successivamente viene sviluppata anche una schermata di accesso che permette di creare un'istanza dello smart contract direttamente dall'applicativo, o di connettersi a un'istanza esistente tramite il suo indirizzo.

4.2.1 Connessione allo Smart Contract

La prima parte dello sviluppo dell'applicativo web è stata dedicata interamente a implementare la comunicazione con lo smart contract. È stata quindi innanzitutto creata un'interfaccia grafica minimale che permettesse di richiamare le funzioni dello smart contract in modo semplice. Come descritto nel capitolo precedente, il framework scelto per lo sviluppo dell'applicazione web è React. È stato quindi creato un progetto di base React con dei `Button` che potessero essere usati per chiamare le varie funzioni dello smart contract. Per far comunicare l'applicazione con uno smart contract presente su blockchain si sono rivelati necessari i seguenti quattro componenti:

1. **Un provider**, ovvero un collegamento che mette in comunicazione l'applicazione con la rete della blockchain, permettendo di inserire transazioni e ottenere dati dai suoi smart contract. Per le prime prove è stato utilizzato direttamente l'URL del provider della testnet IOTA fornito dalla fondazione.

2. **L'indirizzo dello smart contract**. Tale indirizzo è quello con cui è identificata un'istanza di uno smart contract su una chain quando ne viene effettuato il deploy. Come introdotto precedentemente, in questa prima fase di sviluppo è stato inserito nel codice come costante `contractAddress`, ottenendolo direttamente da Remix dopo aver effettuato il deploy.

3. **L'ABI dello smart contract**. L'Application Binary Interface (ABI) è un json contenente l'interfaccia di tutte le variabili e funzioni dello smart contract, compresi gli input, gli output e i loro tipi. È necessario ad applicazioni esterne in modo da poter comunicare con lo smart contract e richiamare le sue funzioni, e può essere ottenuto dopo aver compilato il codice dello smart contract su Remix. Una volta fatto ciò è stato inserito nel progetto in un file `abi.json` che viene incluso nel file principale come costante `abi`. Nel Codice 4.4 è riportato un esempio di

come viene rappresentata una funzione all'interno dell'ABI, prendendo come campione la funzione `check_UUID`.

4. **Un collegamento a un account sul ledger**. Per le prime prove è stato inizialmente implementato inserendo nel codice la chiave privata dell'account come costante. Tale approccio, oltre a fissare l'account nel codice non permettendo a un utente di utilizzare il proprio, utilizza dati sensibili come la chiave privata e non avvisa l'utente nel momento in cui esegue operazioni che richiedono pagamenti.

```
"inputs": [
    {
        "internalType": "string",
        "name": "uuid",
        "type": "string"
    }
],
"name": "check_UUID",
"outputs": [
    {
        "internalType": "bool",
        "name": "",
        "type": "bool"
    }
],
"stateMutability": "view",
"type": "function"
```

Codice 4.4: Interfaccia di funzione nell'ABI

Come provider e collegamento agli account è stato successivamente implementato MetaMask come pianificato nel capitolo precedente. È stato quindi collegato MetaMask all'applicazione per essere utilizzato come provider. Tramite l'oggetto `window.ethereum` infatti, è possibile accedere al provider utilizzato dal browser dell'utente[10]. Per connettersi effettivamente a MetaMask

[10]Per verificare che tale provider sia MetaMask, è presente la variabile booleana `window.ethereum.isMetaMask`

si usa la funzione `window.ethereum.request( method: 'eth_requestAccounts' )`, che fornisce anche una lista contenente gli indirizzi degli account attualmente connessi a MetaMask. Il primo indirizzo della lista è quello attualmente attivo su MetaMask, e nell'applicazione viene quindi salvato in uno stato di React nominato `account`. In questo modo, MetaMask viene allo stesso tempo utilizzato anche come collegamento agli account, permettendo all'applicazione di interagire con lo smart contract tramite l'account dell'utente su MetaMask, senza richiedere dati aggiuntivi. Inoltre, sempre tramite MetaMask, l'utente è avvisato prima di compiere operazioni che richiedono pagamenti.
Una volta soddisfatti tutti i requisiti, è stato possibile collegarsi allo smart contract presente sulla blockchain utilizzando la libreria web3.js. Tramite `const web3 = new Web3(window.ethereum)` è stato creato l'oggetto Web3 utilizzando MetaMask come provider, e successivamente con `const contract = new web3.eth.Contract(abi, contractAddress)` viene creato l'oggetto tramite il quale si può comunicare con lo smart contract richiamando le sue funzioni. Vengono utilizzati due metodi per chiamare le funzioni dello smart contract, a seconda che siano funzioni view o funzioni di write:

1. Per le funzioni view, si utilizza:

   ```
   contract.methods.<nome_funzione>(<parametri_funzione>)
   .call({ from: account })
   ```

   Questa chiamata restituisce una `Promise` contenente i valori di ritorno della funzione dello smart contract. Non essendo funzioni che richiedono consumo di gas, MetaMask non richiede alcuna conferma per utilizzarle.

2. Per le funzioni write invece, è stato inizialmente riscontrato un problema riguardo all'utilizzo del gas. Si è rivelato neccessario infatti, alla chiamata di una funzione, fornire come parametro anche una stima del gas utilizzato, in modo da pagare la quantità adeguata di tasse quando si conferma la transazione da MetaMask. Dopo vari tentativi falliti

di stima del gas, anche tramite varie funzioni di web3.js, è stata infine trovata una funzione in grado di stimarlo correttamente ogni volta: `web3.eth.getGasPrice()`. Una volta stimato correttamente il gas tramite questa funzione, per chiamare funzioni write dello smart contract si utilizza:

```
contract.methods.<nome_funzione>(<parametri_funzione>)
.send({from: account, gasPrice: gasPrice})
```

Per inserire un pagamento nella transazione della chiamata, come ad esempio per chiamare la funzione `buy_ticket`, bisogna aggiungere a `send` un parametro aggiuntivo `value`. Dato che lo smart contract gestisce i bilanci in Wei (vedi 4.1.2), è necessario convertire il valore in tale unità. Sempre nel caso della funzione `buy_ticket` che richiede un pagamento di 10 IOTA, si è convertito il valore in Wei con il metodo `web3.utils.toWei(10, "ether")`[11].
Questa chiamata restituisce una Promise contenente l'hash della transazione effettuata. Prima di inviare la richiesta allo smart contract, viene chiesta conferma all'utente tramite MetaMask, specificando quanti IOTA dovrà spendere per effettuare la transazione.

Una volta definite nel file le varie funzioni che richiamassero quelle dello smart contract, è stato sufficiente assegnarle agli `onClick` dei `Button` creati inizialmente per verificarne il funzionamento corretto.
L'ultima funzionalità dello smart contract da implementare nell'applicazione sono gli eventi. Durante tale implementazione sono stati riscontrati alcuni problemi e limitazioni. La prima limitazione è stata relativa all'utilizzo di MetaMask come provider, poiché non permette l'ascolto di eventi in tempo reale. Questo perché MetaMask è un provider HTTP, ovvero comunica con la chain tramite semplici chiamate e risposte HTTP. Per ascoltare gli eventi è necessario invece un provider WebSocket, che fornisce una connessione

[11] Così come con Solidity, anche web3 è una libreria pensata per Ethereum. Allo stesso modo, l'unità Ether viene considerata come IOTA.

costante alla blockchain capace anche di ascoltare gli eventi in tempo reale[12]. È stato quindi necessario implementare un secondo provider parallelo a MetaMask, utilizzato esclusivamente per l'ascolto di eventi. Il provider WebSocket, connesso alla blockchain tramite l'URL della testnet di IOTA, è stato creato nel seguente modo:

```
const websocket = new Web3.providers.WebsocketProvider(
"wss://ws.json-rpc.evm.testnet.iotaledger.net")
```

Tramite il quale vengono creati i relativi oggetti Web3 con `const web3_wss = new Web3(websocket)` e Contract con `const contract_wss = new web3_wss.eth.Contract(abi, contractAddress)`.
In questo modo, è stato infine possibile ascoltare in tempo reale gli eventi emessi dallo smart contract tramite la definizione di un listener relativo a ogni evento. Nel Codice 4.5 è riportato un esempio di creazione di un listener, in cui viene gestito l'evento `ticket_registered` nell'applicazione finale, emesso quando un utente acquista un biglietto.

```
1  contract_wss.events.ticket_registered()
2  .on("data", async (event) => {
3      // Gestione evento
4      setLastTicketBought({
5          participant: event.returnValues.participant,
6          ticket: event.returnValues.ticket
7      })
8  })
```

Codice 4.5: Ricezione di un evento dello Smart Contract

In questo estratto di codice, viene creato un listener per la callback `data` dell'evento, che viene attivata ogni volta che l'evento viene emesso dallo smart contract e rilevato dal listener. Viene ricevuto un oggetto `event`, che all'interno di `event.returnValues` contiene gli output dell'evento. Per testare inizialmente che gli eventi venissero ricevuti correttamente dall'applicazione,

[12] https://metana.io/blog/what-are-web3-js-providers-explained/

sono stati emessi esternamente[13] e mostrati nell'applicazione tramite `alert` nei listener.
Un altro problema sorto relativo agli eventi riguarda il tempismo della ricezione. Si è riscontrato infatti che alcuni eventi vengono ricevuti dall'applicazione *prima* che lo smart contract finisca l'esecuzione e aggiorni il suo stato. In questo caso quindi, la ricezione dell'evento dall'applicazione avviene troppo presto e non viene rilevato il nuovo stato, impedendo l'aggiornamento in tempo reale. Non è stato possibile eliminare completamente questa limitazione, ma è stata aggirata implementando un timeout di alcuni secondi per la gestione degli eventi emessi da funzioni che richiedono troppo tempo per aggiornare lo stato. Viene quindi utilizzato `new Promise(resolve => setTimeout(resolve, ms))` per creare un'attesa di `ms` millisecondi prima di gestire l'evento ricevuto.

4.2.2 Sviluppo delle schermate gestore e partecipante

Dopo che l'applicazione si è rilevata capace di comunicare correttamente con lo smart contract sulla blockchain potendo utilizzare tutte le sue funzionalità, si è conclusa la fase di test ed è iniziato lo sviluppo del layout dell'applicazione. Come già indicato in precedenza, questa fase di sviluppo consente all'applicazione di interagire con un unico smart contract già presente sulla blockchain, del quale viene inserito l'indirizzo in una costante `contractAddress`.
È stato definito il file principale `Lottery.jsx` che funge da contenitore generale per intercambiare le varie schermate.
Per creare il layout grafico sono stati utilizzati i seguenti moduli npm:

- *react-bootstrap*, che permette l'utilizzo delle classi CSS di bootstrap e fornisce inoltre dei componenti React pre-stilizzati.

[13] Tramite Remix, oppure tramite un'altra istanza dell'applicazione, ad esempio da un altro browser.

- *react-toastify*, che fornisce un componente React per la visualizzazione di notifiche stilizzabili.

La prima cosa che viene visualizzata quando si accede alla pagina di `Lottery.jsx` è un componente, definito nel file `NoMetamask.jsx`, che fornisce all'utente un `Button` per connettere l'applicazione a MetaMask, oppure lo informa in caso il browser attuale non supporti o non utilizzi MetaMask. Una volta connesso MetaMask con il metodo descritto in 4.2.1, l'indirizzo dell'account attualmente attivo su MetaMask viene salvato in uno stato `account` di React. Inoltre è stato definito il listener `accountsChanged` di `window.ethereum`, che si attiva ogni volta che l'utente cambia account su MetaMask. Ad ogni attivazione di questo listener viene impostato in `account` l'indirizzo del nuovo account selezionato, in modo da permettere l'aggiornamento in tempo reale dell'applicazione. A ogni cambio dello stato `account`, viene verificato se tale indirizzo è quello del gestore della lotteria, comparandolo con l'indirizzo restituito dalla view `lottery_manager`. Il risultato di questa comparazione è salvato in uno stato `isOwner` che definisce se l'utente connesso è il proprietario o meno. Una volta deciso il valore di `isOwner`, uno stato `auth` inizializzato a `false` viene impostato a `true`, indicando che l'utente è stato autenticato con successo. Questo permette la visualizzazione di uno dei due componenti principali, contenuti nei file `OwnerView.jsx` e `UserView.jsx`. All'apertura dell'applicazione viene anche verificata l'apertura o chiusura della lotteria tramite la view `is_lottery_open`, salvandone il risultato in uno stato `isLotteryOpen`. Questo stato decide quali operazioni è possibile effettuare all'interno delle schermate gestore e partecipante, e viene mostrato tramite testo all'interno di entrambe le schermate, comunicando all'utente se la lotteria è attualmente chiusa o aperta. `Lottery.jsx` fornisce inoltre un titolo comune a entrambe le schermate contenente il nome della lotteria, ottenuto chiamando la view `lottery_name` dello smart contract. In `Lottery.jsx` sono definiti anche i listener per gestire gli eventi dello smart contract. Ogni evento fornisce all'utente un toast di notifica nel seguente modo:

- `ticket_registered`, che restituisce `participant` e `ticket`, comunica a tutti gli utenti il cui indirizzo è diverso da quello di `participant`[14] che un altro utente ha acquistato il biglietto `ticket`. Verifica inoltre se dopo questo acquisto i biglietti sono esauriti, chiamando la view `check_ticket_availability` e salvanone il risultato nello stato booleano `ticketsAvailable`. Questo stato viene poi utilizzato nella schermata partecipante (come descritto successivamente in 4.2.2).

- `ticket_published`, che restituisce `ticket`, comunica a tutti gli utenti che è stato aggiunto il nuovo biglietto `ticket` a quelli disponibili. Analogamente al punto precedente, aggiorna lo stato `ticketsAvailable`.

- closed_lottery comunica a tutti gli utenti che la lotteria è stata chiusa, e in seguito effettua la chiamata alla view `is_lottery_open` per aggiornare lo stato dell'applicazione.

Per richiamare le view alla fine della gestione degli eventi, è stata impostata una *wait* di 10 secondi prima di effettuare la chiamata, per gestire la ricezione anticipata degli eventi come descritto precedentemente in 4.2.1.

Contenuti di OwnerView

Il componente `OwnerView.jsx` che mostra la schermata del gestore viene quindi caricato in `Lottery.jsx` quando `auth` e `isOwner` sono impostati a `true`.
Quando la lotteria è aperta, mostra un pannello di controllo con un `Button` per ogni azione amministrativa che il gestore può compiere:

- **Monitorare la lista dei biglietti** Mostra un `Modal` contenente la lista di tutti i biglietti presenti nella lotteria, ottenuti chiamando la view `check_tickets` dello smart contract e salvati nello stato `tickets`. Inoltre tramite le view `check_winning_ticket` e `check_ticket_allotment`

[14] Per non comunicarlo all'utente che ha ascquistato il biglietto.

vengono salvati l'indice del biglietto vincente e il numero di biglietti venduti negli stati `winningTicketID` e `ticketsSold`. Tramite il primo stato, viene mostrato al gestore quale dei biglietti nella lista è quello vincente, evidenziandolo. Tramite il secondo invece, viene mostrato quali biglietti sono stati venduti inserendo una dicitura "Acquistato" di fianco ai primi `ticketsSold` biglietti[15].

• **Chiudere la lotteria** Permette di richiamare la funzione `close_lottery` dello smart contract, finché sono rispettati i requisiti sul numero di biglietti venduti per poter chiudere la lotteria (vedi 4.1.2). In caso questi ultimi non siano soddisfatti, il `Button` è disattivato e viene mostrato un `Tooltip` che ne notifica al gestore il motivo.

• **Modifica del biglietto vincente** Apre un `Modal` con un input che permette di specificare il nuovo indice del biglietto vincente, finché sia minore di `tickets.length`. All'invio viene chiamata la funzione `change_winning_ticket` dello smart contract utilizzando l'input fornito come parametro.

• **Aggiungere un biglietto** Apre un `Modal` con un input che permette di specificare l'identificativo numerico del biglietto da aggiungere. Nonostante lo smart contract non abbia controlli sugli ID numerici poiché la loro scelta non influisce sul suo corretto funzionamento, per rendere l'applicazione più ordinata è stato deciso di uniformarli all'interno di quest'ultima. Sono quindi richiesti identificativi di 4 cifre e che siano univoci all'interno di una lotteria. All'invio viene chiamata la funzione `add_ticket` dello smart contract utilizzando l'input fornito come parametro.

Ad ogni azione effettuata, un toast notifica al gestore il risultato dell'operazione.
Quando invece la lotteria è chiusa, al posto del pannello di controllo viene

[15] Questo è possibile grazie al fatto che i biglietti vengono venduti in ordine sequenziale. I biglietti venduti sono quindi sempre i primi x della lista, dove x è la quantità `ticketsSold`.

mostrato un resoconto della lotteria conclusa, contenente le seguenti informazioni:

- Il numero di biglietti venduti rispetto a quelli totali, rispettivamente `ticketsSold` e `tickets.length`.

- Il valore dei guadagni totali dalla vendita dei biglietti, ricavato da `ticketsSold · 10`.

- Se il vincitore ha già ritirato la vincita o meno, e il valore di tale vincita. Il primo dato è ottenuto chiamando la view `cashed_in` dello smart contract, salvandone il risultato nello stato booleano `isCashedIn`. Il valore della vincita è invece ricavato[16] da `ticketsSold · 10 · 0,7`.

- Se il gestore ha già ritirato i guadagni o meno, e il valore di tali guadagni. Analogamente al punto precedente, si ottiene lo stato booleano `earningsRetrieved` dalla view `check_earnings_retrieved`, e si ricava il valore dei guadagni[16] da `ticketsSold · 10 · 0,3`.

Viene inoltre mostrato, quando la lotteria è chiusa, il `Button` per incassare i guadagni[17]. Tramite lo stato booleano `earningsRetrieved` viene controllato se disattivare il `Button` quando il gestore ha già riscattato i guadagni. Tramite il `Button` viene quindi chiamata la funzione `retrieve_earnings` dello smart contract. Se i guadagni vengono ritirati con successo, un toast notifica il gestore indicando la quantità di IOTA ricevuti, valore ottenuto chiamando la view `earned_amount`.
In Figura 4.1 sono riportate le versioni finali della schermata gestore durante la lotteria aperta e chiusa.

[16]Non vengono utilizzate le view `winning_amount` e `earned_amount` poiché i loro valori nello smart contract vengono impostati soltanto quando i token vengono ritirati.

[17]Se invece la lotteria è stata annullata, ovvero chiusa senza vendere alcun biglietto, viene soltanto comunicato l'annullamento.

(a) Schermata gestore quando la lotteria è aperta

(b) Schermata gestore quando la lotteria è chiusa

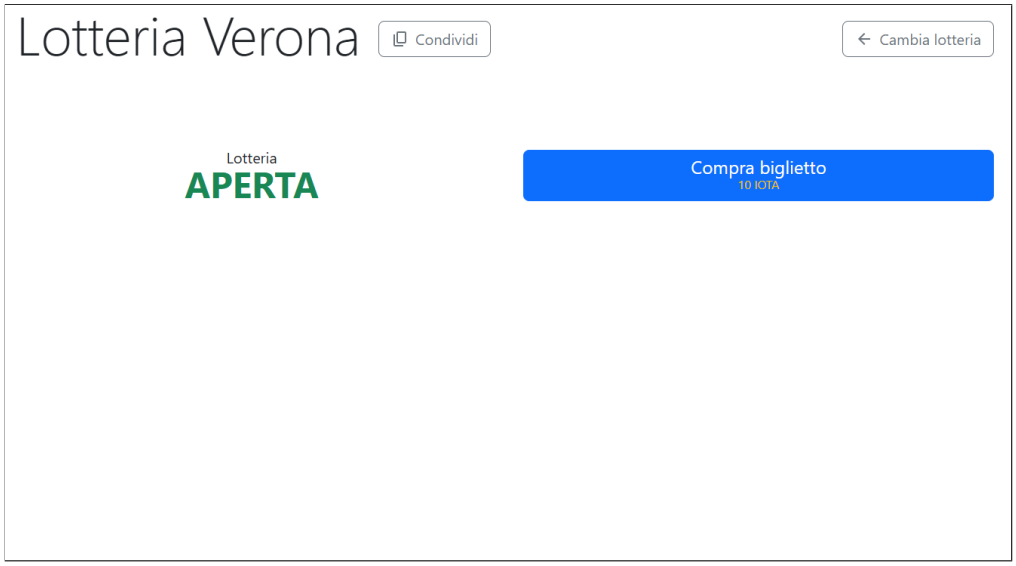
Figura 4.1: Versione finale della schermata gestore

Contenuti di UserView

Il componente `UserView.jsx` che mostra la schermata per un partecipante alla lotteria viene caricato in `Lottery.jsx` quando `auth` è impostato a `true` e `isOwner` a `false`.
Quando la lotteria è aperta, mostra ai partecipanti che ancora non hanno acquistato un biglietto un `Button` per poter effettuare l'acquisto. Permette quindi di chiamare la funzione `buy_ticket` dello smart contract, utilizzando il parametro aggiuntivo `value: web3.utils.toWei(10, "ether")` per specificare la somma del pagamento (vedi 4.2.1). Se non ci sono biglietti disponibili, verificabile dallo stato `ticketsAvailable` ottenuto dalla view `check_ticket_availability`, il `Button` è disattivato. Gli utenti che invece hanno acquistato un biglietto ne visualizzano l'identificativo numerico.
Per verificare se l'utente possiede un biglietto e ottenere l'identificativo di tale biglietto in caso positivo, si utilizza un'unica chiamata alla view `check_ticket` dello smart contract. Se la chiamata va a buon fine e restituisce il biglietto, lo mostra salvandolo nello stato `ticket`. Altrimenti, se la chiamata lancia l'errore relativo al mancato possesso del biglietto da parte del chiamante, imposta `ticket` a `undefined` che di conseguenza mostra il `Button` di acquisto. Anche se in questo modo viene permesso il fallimento della transazione, non è impattante in quanto la funzione è una view e quindi non utilizza alcun gas e non richiede nessuna tassa o conferma all'utente.
Quando la lotteria è chiusa, se un utente non ha ancora acquistato un biglietto, il `Button` di acquisto viene disattivato. I partecipanti potranno invece visualizzare un testo che li informa della vincita o perdita, che viene verificato tramite la view `check_winner`, il cui risultato viene salvato nello stato `isWinner`. Al vincitore verrà mostrato anche il `Button` per incassare la vincita. Tramite lo stato `isCashedIn` ottenuto dalla view `cashed_in`, viene controllato se disattivare il `Button` quando il vincitore ha già incassato la vincita. Tramite il `Button` viene quindi chiamata la funzione `cash_in_victory` dello smart contract. Se la vincita viene ritirata con successo, un toast notifica l'utente indicando la quantità di IOTA ricevuti, valore ottenuto chiamando

la view `winning_amount`.

In Figura 4.2 sono riportate le versioni finali della schermata partecipante durante la lotteria aperta e chiusa.



(a) Schermata partecipante quando la lotteria è aperta

(b) Schermata partecipante quando la lotteria è chiusa

Figura 4.2: Versione finale della schermata partecipante

4.2.3 Sviluppo della schermata di accesso

La seconda fase di sviluppo dell'applicazione web è stata invece dedicata ad implementare la possibilità di creare una lotteria direttamente dall'applicazione, e di accedere a qualsiasi lotteria tramite il suo indirizzo. È stato quindi creato a questo scopo un componente aggiuntivo `Access.jsx` che viene caricato in `Lottery.jsx` dopo aver effettuato il collegamento a MetaMask. Per poter creare una lotteria, è stato necessario implementare un metodo per effettuare il deploy di uno smart contract tramite l'applicazione web. Si sono innanzitutto rilevati necessari questi due componenti:

1. **L'ABI dello smart contract**, analogamente ai requisiti per connettersi a uno smart contract già inserito in blockchain (vedi 4.2.1).

2. **Il bytecode del codice dello smart contract**. È possibile ottenerlo dopo aver compilato il codice da Remix. È stato poi successivamente salvato nel file `bytecode.js` e importato in `Access.jsx` come costante `bytecode`.

Nel Codice 4.6 è riportato come è stato implementato il deploy di uno smart contract lotteria tramite l'applicazione.

```
1 // Utilizzo di ABI e bytecode per generare lo smart contract
2 const contract = new web3.eth.Contract(abi)
3 contract.options.data = bytecode
4 // Creazione della transazione per effettuare il deploy
5 const gasPrice = await web3.eth.getGasPrice()
6 const deployedContract = await contract.deploy({arguments: [
    ticketList, winningTicket, lotteryName]}).send({ from:
    account, gasPrice: gasPrice })
```

Codice 4.6: Deploy di Smart Contract da applicazione web

Nel codice viene creato un oggetto `Contract` a partire dall'ABI per gli smart contract lotteria, al quale viene poi fornito il bytecode contenente il codice compilato. Tutto questo viene caricato su blockchain tramite il metodo `deploy` dell'oggetto. I parametri del costruttore vengono inseriti nel parametro

`arguments` di `deploy` sotto forma di array. L'oggetto `deployedContract` ottenuto come risultato della funzione `deploy` contiene le informazioni dello smart contract appena inserito su blockchain. In particolare, l'indirizzo dello smart contract è reperibile in `deployedContract.options.address`.
Il componente possiede un `Button` per creare una lotteria, che apre un `Modal` contenente il form per la creazione. Viene richiesto di inserire un nome per la lotteria di massimo 50 caratteri e di inserire i biglietti uno per volta tramite un input[18]. Viene inoltre visualizzata la lista dei biglietti aggiunti fino a quel momento, dalla quale è possibile selezionare quale rendere vincente. All'invio del form, viene effettuato il deploy dello smart contract come visto nel Codice 4.6, utilizzando come parametri per il costruttore i dati inseriti. Successivamente, viene salvato il suo indirizzo in `contractAddress`, utilizzato poi per creare gli oggetti `contract` e `contract_wss`[19] in modo analogo al procedimento in sezione 4.2.1. Una volta creati questi due oggetti, viene caricata la schermata gestore o partecipante.
Per poter invece accedere a una lotteria già presente su blockchain, il componente possiede un input nel quale è possibile inserirne l'indirizzo. All'invio, l'indirizzo inserito viene salvato in `contractAddress`. In questo caso però, prima di poter creare gli oggetti `Contract` è necessario verificare che l'indirizzo inserito sia valido, e che sia effettivamente l'indirizzo di un'istanza lotteria piuttosto che di un qualsiasi altro smart contract. Viene quindi creato un oggetto `Contract` temporaneo utilizzato per convalidare l'indirizzo. Per verificare che sia un indirizzo valido, è sufficiente che la creazione dell'oggetto vada a buon fine. In caso di errore, significa che l'indirizzo non è di uno smart contract e quindi è possibile notificare l'utente dell'errore. Per verificare che invece sia un'istanza degli smart contract lotteria, viene effettuato un tentativo di chiamata alla view `check_UUID`, utilizzando come parametro una costante `contract_UUID` creata nell'applicazione, analoga a quella

[18] Come per l'aggiunta di un biglietto dalla schermata gestore, sono richiesti identificativi univoci da 4 cifre.

[19] `contractAddress`, `contract` e `contract_wss`, che prima erano costanti, ora diventano stati React in modo da poter essere modificati.

presente nello smart contract (vedi 4.1.3). Se la chiamata fallisce perché la funzione non esiste o perché l'UUID fornito non combacia con quello dello smart contract, l'indirizzo inserito non è di un'istanza lotteria e l'utente viene notificato dell'errore. Altrimenti, l'indirizzo è corretto e vengono creati gli oggetti `contract` e `contract_wss`, caricando la schermata gestore o partecipante.
Inoltre, sono stati aggiunti due `Button` alle schermate gestore e partecipante per integrare adeguatamente la nuova schermata di accesso:

1. Un `Button` per condividere l'indirizzo della lotteria. Permette di copiare nella clipboard l'indirizzo dello smart contract a cui si è connessi, per poter salvare o condividere la lotteria a dei partecipanti.

2. Un `Button` per ritornare alla schermata di accesso. Elimina `contractAddress`, `contract` e `contract_wss`, poiché la mancanza di questi stati causa il caricamento del componente `Access.jsx`.

In Figura 4.3 è riportata la versione finale della schermata di accesso.

Figura 4.3: Schermata di accesso per creare o accedere a una lotteria

Capitolo 5

Risultati

In questo capitolo viene effettuato un resoconto sui risultati riscontrati dall'utilizzo della programmazione su ledger distribuiti. In particolare, vengono analizzati vantaggi e svantaggi portati dallo sviluppo di applicazioni basate su smart contract rispetto ai classici paradigmi. Viene inoltre analizzata l'efficienza di esecuzione sul ledger scalabile di IOTA, comparandolo anche con quello di Ethereum.

5.1 Vantaggi della programmazione di Smart Contract

Sono stati riscontrati vari vantaggi dallo sviluppo di un'applicazione su ledger distribuito. Il più rilevante è sicuramente l'alto livello di sicurezza garantito dal ledger, sia per l'autenticazione sia per la gestione degli scambi di denaro, che sono al contempo semplici e veloci da implementare poiché la base per la loro implementazione è già fornita dal ledger. Come infatti descritto nel capitolo precedente, per implementare il sistema di autenticazione è stato sufficiente inserire dei semplici controlli all'interno dello smart contract. I controlli convalidano gli indirizzi degli utenti nel ledger, che sono già autenticati tramite quest'ultimo. In una normale applicazione web, sarebbe invece stato necessario implementare un backend con un sistema di

autenticazione molto più complesso, e che non raggiungerebbe comunque lo stesso livello di sicurezza. Lo stesso discorso è valido anche per l'implementazione dei pagamenti: sono infatti sufficienti semplici chiamate di funzione per utilizzare il sistema di pagamenti sicuro e senza autorità centrali del ledger. Per implementare un sistema di pagamento in un'applicazione web che non si basa su ledger distribuiti, sarebbe stato necessario creare un sistema più lungo e complesso[1], passando anche per delle autorità terze. Questo risulta spesso anche in tasse sugli acquisti superiori a quelle richieste per utilizzare gas su blockchain.

Un altro vantaggio è dato dalla possibilità di utilizzare lo smart contract anche come database dell'applicazione. Tramite i mapping è infatti possibile creare in modo semplice associazioni anche elaborate fra utenti del ledger e strutture dati, tutto salvato in modo sicuro e permanente nello stato della blockchain. Questo evita quindi di implementare un vero e proprio database all'interno di un server dell'applicazione web.

Tutti i vantaggi elencati risultano infine nella rimozione della necessità di creare un server backend per l'applicazione web. Tutto è infatti gestito dal ledger stesso, necessitando quindi della sola programmazione di uno smart contract, ma godendo allo stesso tempo di un'elevata sicurezza.

Fra gli svantaggi invece, risultano innanzitutto le conseguenze dell'immutabilità degli smart contract. Dopo il deploy non è infatti più in alcun modo possibile modificare lo smart contract, ed è necessaria quindi una grande attenzione prima del deploy per assicurarsi che non saranno necessarie modifiche future. Un altro punto importante è la necessità per gli utenti di possedere account e bilanci all'interno del ledger, cosa che attualmente non è scontata per molti utenti finali. Al momento infatti, i ledger distribuiti sono una tecnologia in crescita e non sono ancora largamente utilizzati. Inoltre, nell'applicazione sono da aspettarsi dei tempi di esecuzione per tutte le

[1]Sia per l'implementazione che per l'utilizzo finale. I pagamenti sul ledger sono infatti molto rapidi e richiedono pochi passaggi all'utente, rispetto ai sistemi di pagamento tradizionali forniti da autorità terze come banche.

operazioni che interagiscono con lo smart contract leggermente più elevati rispetto a quelli su un'applicazione tradizionale. È poi da tenere in considerazione la necessità di pagare tasse per il gas bruciato dalle modifiche allo stato della blockchain, che seppur minimali sono comunque sempre presenti.

5.2 Tempi di esecuzione

Sono stati inoltre effettuati dei test per valutare l'efficienza di esecuzione dello smart contract.
Sono stati innanzitutto testati i tempi di esecuzione delle funzioni view, prendendo come campione la funzione `is_lottery_open`. I dati sono stati raccolti effettuando la chiamata 100 volte, sia sulla testnet IOTA per la quale è stata sviluppata l'applicazione, sia sulla testnet Sepolia di Ethereum. Nella Figura 5.1 sono riportati i risultati.

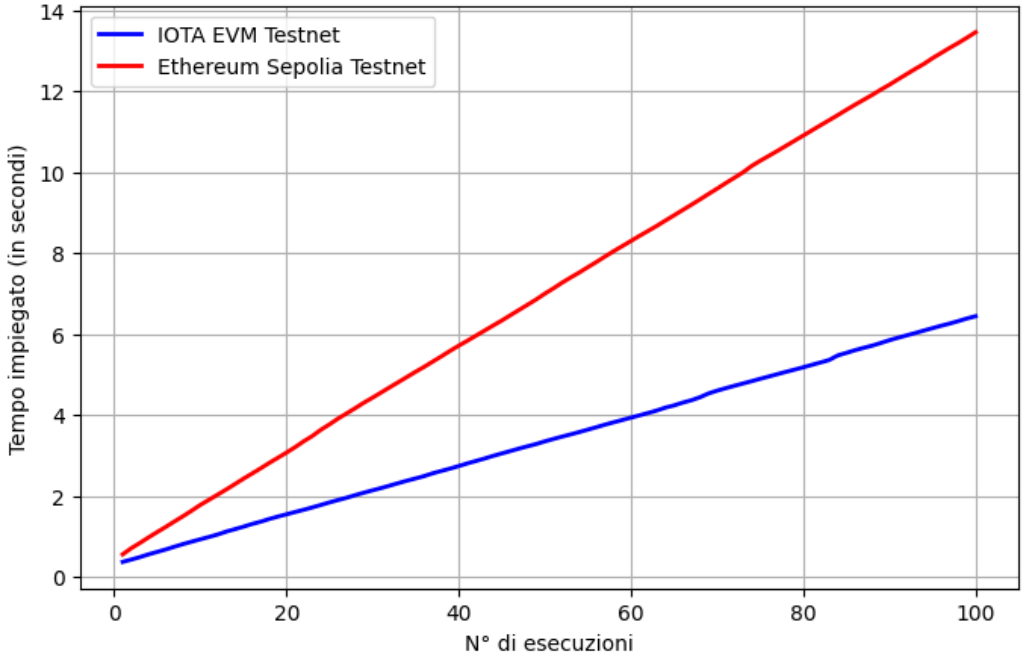


Figura 5.1: Comparazione dei tempi di esecuzione di `is_lottery_open` fra IOTA e Ethereum

Da questo grafico si evince che l'efficienza della rete IOTA nell'effettuare operazioni di lettura dello stato è superiore a quella della rete Ethereum. IOTA arriva infatti con 100 esecuzioni ad avere un tempo totale di esecuzione di circa 6 secondi rispetto ai circa 13 secondi di Ethereum, vantando quindi un aumento dell'efficienza di oltre il 100%.
Successivamente sono stati testati i tempi di esecuzione anche delle funzioni di scrittura sullo stato, usando la funzione `buy_ticket`[2] che prevede sia scritture che pagamenti. I risultati sono riportati in Figura 5.2.

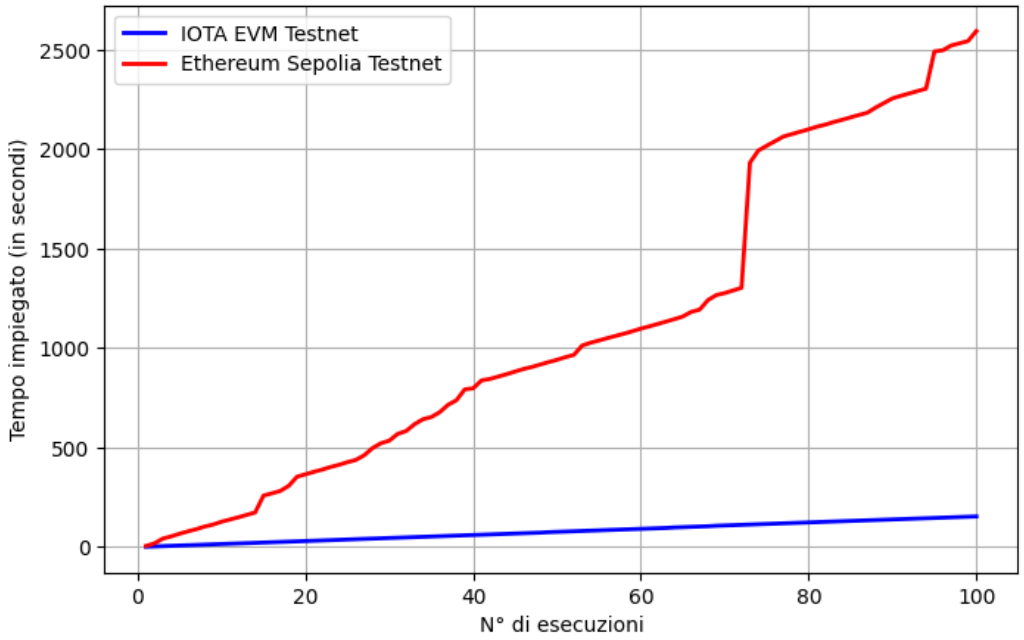


Figura 5.2: Comparazione dei tempi di esecuzione di `buy_ticket` fra IOTA e Ethereum

Qui, in un'operazione che prevede pagamenti e modifiche allo stato, è dove la vera efficienza del ledger scalabile di IOTA viene evidenziata. Mentre infatti IOTA mantiene sempre dei tempi ragionevoli, con circa 154 secondi totali dopo 100 esecuzioni, Ethereum invece presenta dei tempi nettamente

[2]Per questi test sono stati rimossi alcuni controlli della funzione, in modo da permettere multiple esecuzioni da parte dello stesso indirizzo.

più elevati. Per effettuare 100 esecuzioni infatti, ha impiegato un tempo totale di circa 2594 secondi, ovvero circa 43 minuti. Una singola esecuzione spaziava da un tempo minimo di 5 secondi fino a un tempo massimo di anche quasi 700 secondi nei momenti in cui la rete era congestionata[3], con il tempo medio più comune di 10-20 secondi. Grazie all'elevata scalabilità di IOTA e alle sue PoW estremamente leggere per creare una transazione, ogni sua esecuzione ha invece impiegato un tempo costante di circa 1-2 secondi.
Infine, sono stati testati anche i tempi di deploy, effettuando un ciclo di esecuzione di 100 deploy sia per IOTA che per Ethereum. I risultati sono riportati in Figura 5.3.

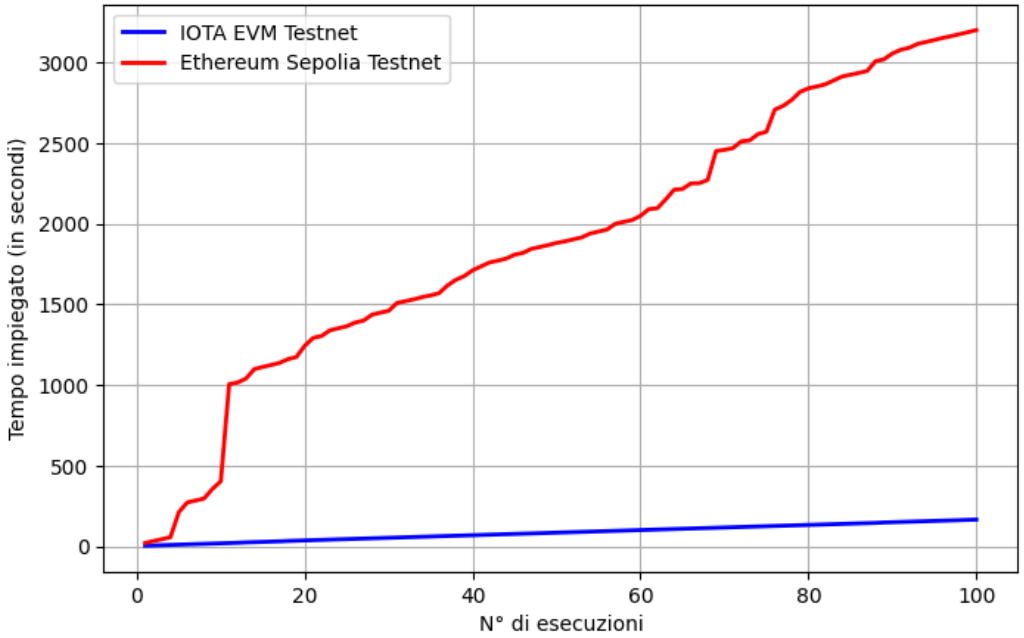


Figura 5.3: Comparazione dei tempi di esecuzione di un deploy fra IOTA e Ethereum

[3]Questi momenti di massimo congestionamento rendono il calcolo delle tempistiche estremamente variabile in base al momento in cui viene svolto il test. Infatti, svolgendo nuovamente il ciclo di 100 esecuzioni in un momento diverso in cui la rete era probabilmente più congestionata, il tempo impiegato ha raggiunto i 7541 secondi, ovvero circa 2 ore.

Le tempistiche qui sono simili a quelle delle funzioni write, con dei tempi maggiorati per entrambe le reti. Ethereum ha compiuto un tempo totale di circa 3201 secondi, ovvero circa 53 minuti. IOTA invece ha necessitato di un tempo totale di circa 165 secondi. C'è stato quindi un aumento del 23% per Ethereum rispetto all'esecuzione di una funzione di write, equivalente a 10 minuti in più[4], anche in questo caso con tempistiche estremamente variabili fra singole esecuzioni. IOTA invece, rispetto alla funzione di write ha impiegato soltanto il 6% di tempo in più, equivalente a 10 secondi, mantenendo inoltre sempre delle tempistiche molto stabili e lineari per ogni singola esecuzione. Questo prova nuovamente l'efficienza e la stabilità del ledger di IOTA rispetto ad altre blockchain come Ethereum.

[4]Questi tempi, data l'instabilità delle prestazioni di Ethereum, possono essere molto variabili.

Capitolo 6

Conclusioni e sviluppi futuri

In questa tesi sono state analizzate le possibilità e i vantaggi portati dallo sviluppo su piattaforme decentralizzate di ledger distribuiti, facendo luce in particolare su come IOTA Foundation stia portando in questo settore rivoluzioni che puntano a migliorare nettamente efficienza e usabilità delle piattaforme. A tale scopo, sono stati sviluppati uno Smart Contract e un'applicazione esterna che si interfaccia con esso.
È stata innanzitutto introdotta la tecnologia di IOTA. Si è visto come la struttura a DAG del Tangle risolva i problemi di scalabilità delle tradizionali strutture di blockchain. Con la successiva integrazione del multi-ledger, IOTA ha poi permesso la creazione di multiple blockchain sopra al Tangle, che agiscono in parallelo potendo anche comunicare fra di loro, su cui possono girare degli smart contract. Vengono analizzati anche i benefici portati dall'aggiornamento IOTA 2.0, che garantisce un'economia bilanciata e sostenibile basata sull'utilizzo di una nuova risorsa chiamata Mana. Inoltre, viene accennato il nuovissimo progetto IOTA Rebased, approvato ufficialmente durante la stesura di questa tesi, che pianifica di unificare in un unico layer il Tangle e la programmazione di smart contract.
Successivamente, sono state descritte le specifiche dello smart contract sviluppato e della sua applicazione esterna. Lo smart contract, scritto in Solidity, implementa un sistema di lotterie, permettendo a un'entità di creare

e gestire una lotteria. Qualsiasi altra entità può partecipare a una lotteria acquistando un biglietto, e al termine della lotteria sia il vincitore che il gestore possono ritirare una somma di denaro, rispettivamente del 70% e del 30% dei guadagni totali ottenuti dalla vendita dei biglietti. L'applicazione esterna permette di interagire con tale smart contract, fornendo la possibilità di creare una lotteria o di partecipare a una già creata, mostrando poi una schermata gestore o partecipante in seguito all'autenticazione dell'utente. A tale scopo viene realizzata un'applicazione web in React e TypeScript, che interagisce con lo smart contract tramite la libreria web3.js e l'estensione web MetaMask.

È stato poi illustrato il processo di sviluppo dello smart contract e dell'applicazione web. Lo sviluppo dello smart contract è avvenuto in tre fasi. Durante la prima fase è stata sviluppata la struttura base del sistema di lotteria, implementando le funzionalità progettate. Durante la seconda fase è stato implementato il sistema di pagamenti, che include l'acquisto dei biglietti e il pagamento al gestore e al vincitore alla fine della lotteria. Durante la terza fase invece, sono stati implementati eventi e view aggiuntive per permettere all'applicazione di interfacciarsi correttamente con lo smart contract. Lo sviluppo dello smart contract si è rivelato semplice e senza particolari problemi o limitazioni. Lo sviluppo dell'applicazione web è invece avvenuto implementando inizialmente le schermate gestore e partecipante, e utilizzando un unico smart contract hardcoded nell'applicazione. Successivamente, è stata implementata anche la schermata che permette di creare uno smart contract lotteria direttamente dall'applicazione e di connettersi a uno già creato. Durante lo sviluppo dell'applicazione web sono stati riscontrati alcuni problemi, principalmente legati alla gestione degli eventi emessi dallo smart contract. È stato infatti rilevato che, quando un evento viene ricevuto dall'applicazione, lo smart contract potrebbe non aver ancora salvato le modifiche effettuate allo stato dalla funzione che ha emesso l'evento. Quindi, quando l'applicazione necessitava del nuovo stato dello smart contract per gestire l'evento, è stato necessario inserire dei tempi di attesa per permettere

allo smart contract di aggiornarsi.
Infine, sono stati analizzati i principali vantaggi e svantaggi riscontrati durante lo sviluppo di un'applicazione su piattaforme di ledger distribuiti. Lo sviluppo si è rivelato particolarmente vantaggioso per quanto riguarda l'elevata sicurezza e la semplicità nell'implementarla. L'autenticazione, lo scambio di denaro, e il salvataggio di dati in un database sono infatti gestiti dal ledger e facilmente implementabili nello smart contract, rendendo sufficiente lo sviluppo di un'applicazione frontend che si interfacci con esso senza necessitare di un server backend. Inoltre, grazie alla natura del ledger, gli scambi di denaro sono sicuri, veloci, e non necessitano di autorità centrali. Le complessità da tenere in considerazione invece, includono la necessità di prestare particolare attenzione alla correttezza del codice prima di effettuare il deploy di uno smart contract, in quanto successivamente non può più essere modificato in alcun modo. Inoltre, la necessità per gli utenti di possedere un account sul ledger potrebbe influire negativamente sulla distribuzione dell'applicativo. Attualmente infatti, questo è un settore recente e ancora in fase di sviluppo e crescita, con un'utenza più ristretta rispetto ai possibili utilizzatori di un'applicazione sviluppata in modo tradizionale. Si presenta anche la necessità per gli utenti di pagare tasse in gas per l'utilizzo dell'applicazione, e di attendere tempi maggiori per il completamento delle operazioni.
È stato analizzato anche quanto il ledger di IOTA fornisca prestazioni più elevate rispetto ad altri sistemi tradizionali di blockchain. I risultati ottenuti riportano infatti tempi dimezzati per effettuare operazioni di lettura, e tempi enormemente inferiori per effettuare operazioni di scrittura. Questo dimostra quanto sia reale il bottleneck dato dalla struttura tradizionale di blockchain, e quanto invece sia positivamente impattante la struttura scalabile a DAG di IOTA.
Per portare avanti il lavoro svolto durante questa tesi, è possibile continuare ad approfondire un sistema per implementare in modo più efficiente la gestione degli eventi. Un'implementazione da tenere in considerazione potrebbe partire dall'idea di effettuare periodicamente i controlli dopo la ricezione del-

l'evento[1] per rendere più reattiva la risposta, nonostante però questo implichi multiple transazioni aggiuntive da dover effettuare. Inoltre, è da tenere in considerazione che durante lo sviluppo del progetto è stato annunciato che la libreria utilizzata web3.js verrà archiviata[2], con l'avvio del processo a marzo 2025. La libreria creata da Ethereum Foundation è stata la prima storica libreria per comunicare con smart contract. Nel 2020 fu ceduta a ChainSafe per proseguirne la manutenzione e lo sviluppo[3], e in definitiva è stato deciso a gennaio 2025 di terminarne il supporto per promuovere la crescita di altre librerie più recenti. Quindi, un passo importante per proseguire il lavoro di tesi deve essere quello di considerare l'implementazione con una nuova libreria, come ethers.js o Viem, che sono attualmente le più promettenti fra le librerie in crescita riguardanti la comunicazione con reti di blockchain.
In generale, il lavoro svolto in questa tesi propone un esempio di come i programmi sviluppati su ledger distribuiti possano essere utilizzati in contesti reali, e punta a mostrare quanto sia vantaggioso scegliere di supportare e utilizzare le tecnologie proposte da IOTA Foundation. Pertanto, questo lavoro è ampiamente proseguibile anche continuando a sviluppare smart contract su IOTA e ad analizzare possibili casi di utilizzo a cui questi possono essere applicati, implementando magari anche sistemi di tokenizzazione avanzata spiegati in sezione 2.3.4, e approcciandosi allo sviluppo sul nuovo IOTA Rebased con il più avanzato linguaggio Move della MoveVM.

[1] Ad esempio, ogni 3 secondi per 15 secondi, invece che una sola volta dopo 10 secondi.

[2] `https://blog.chainsafe.io/web3-js-sunset/`

[3] `https://medium.com/chainsafe-systems/chainsafe-receives-grant-from-ef-for-web3js-9e4376f5f36a`

Appendice A

Compilazione e deploy tramite Remix

In questa appendice sono riportati i passi da seguire per poter effettuare il deploy di uno smart contract su blockchain, tramite l'IDE di Remix.

1. Accedere all'IDE, disponibile al seguente link: `https://remix.ethereum.org/`.

2. Inserire il file con il codice dello smart contract nel file explorer online di Remix, oppure caricarlo dal proprio filesystem locale tramite il plugin *remixd*[1].

3. Compilare il codice:

   (a) Accedere alla sezione **Solidity Compiler** di Remix e avviare la compilazione.

   (b) Come risultato, è inoltre possibile copiare l'ABI e il bytecode dello smart contract.

[1] `https://remix-ide.readthedocs.io/en/latest/remixd.html`

4. Connettersi a un provider. Per questo progetto, è stato utilizzato MetaMask[2]:
   (a) Creare un wallet con uno o più account.
   (b) Collegarsi alla rete del ledger tramite il provider. Per questo progetto, è stata utilizzata la testnet di IOTA, i cui dati per la connessione sono forniti al seguente link: `https://wiki.iota.org/build/networks-endpoints/#iota-evm-testnet`.
5. Ottenere dei token nell'account desiderato. Nel caso della testnet, è possibile ottenere fondi gratuitamente tramite faucet al seguente link: `https://evm-toolkit.evm.testnet.iotaledger.net/`.
6. Effettuare il deploy:
   (a) Accedere alla sezione **Deploy & run transactions** di Remix.
   (b) Selezionare come **ENVIRONMENT** il provider ottenuto nel passo 4.
   (c) Selezionare l'account tramite il quale si vuole effettuare il deploy.
   (d) Inserire i parametri di input del costruttore e avviare **Deploy**.
7. Lo smart contract sarà poi stato inserito su blockchain e sarà possibile monitorarlo e interagirci tramite Remix nella sezione **Deployed Contracts**.

[2] `https://metamask.io/`

Appendice B

Utilizzare l'applicazione web

In questa appendice sono riportati i passi da seguire per poter avviare ed utilizzare l'applicazione web.

1. Installare MetaMask come provider e connettersi alla testnet IOTA, seguendo il passo 4 dell'Appendice A.
2. Ottenere fondi sulla testnet seguendo il passo 5 dell'Appendice A.
3. Installare Node.js e npm[1].
4. Avviare un terminale all'interno della cartella di progetto.
5. Eseguire il comando `npm install` e attendere l'installazione dei moduli dell'applicazione.
6. Eseguire `npm run start` e attendere l'avvio automatico dell'applicazione web sul browser.

[1] `https://docs.npmjs.com/downloading-and-installing-node-js-and-npm`

Bibliografia

[1] S. Y. Popov. (2015) The tangle. [Online]. Available: https://api.semanticscholar.org/CorpusID:4958428

[2] Q. Bramas. (2018) The stability and the security of the tangle. [Online]. Available: https://hal.science/hal-01716111v2

[3] E. Drasutis. (2021) Iota smart contracts. [Online]. Available: https://files.iota.org/papers/ISC_WP_Nov_10_2021.pdf

[4] O. Saa, A. Cullen, and L. Vigneri. (2023) Iota 2.0 incentives and tokenomics whitepaper. [Online]. Available: https://files.iota.org/papers/IOTA_2.0_Incentives_And_Tokenomics_Whitepaper.pdf

[5] D. Vujicic, D. Jagodic, and S. Randic, "Blockchain technology, bitcoin, and ethereum: A brief overview," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2018.

[6] H. Liu, X. Luo, H. Liu, and X. Xia, "Merkle tree: A fundamental component of blockchains," in *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*, 2021.

[7] IOTA Foundation. (2024) Iota rebased: Technical view. [Online]. Available: https://blog.iota.org/iota-rebased-technical-view/

[8] ——. (2024) Iota rebased: Fast forward. [Online]. Available: https://blog.iota.org/iota-rebased-fast-forward/