

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

**RADIX DLT:
UN'ALTERNATIVA SCALABILE
ALLE BLOCKCHAIN**

Relatore:
Chiar.mo Prof.
GABRIELE D'ANGELO

Presentata da:
ENRICO MORSELLI

Correlatori:
Chiar.mo Prof.
STEFANO FERRETTI
Dott. MIRKO ZICHICHI

IV Sessione
Anno Accademico 2018/2019

Introduzione

La tecnologia Blockchain, introdotta inizialmente come registro distribuito per le transazioni della moneta digitale Bitcoin nel 2008, ha suscitato grande interesse nel corso degli ultimi anni. In poco tempo si è capito come tale tecnologia potesse essere applicata ai campi più disparati, e non soltanto alle monete elettroniche. Fra le tante applicazioni che si prospettano per le blockchain vi sono, ad esempio, il settore energetico, il sistema sanitario, e la tracciabilità delle catene di fornitura di beni, servizi o prodotti alimentari. Il futuro insomma, sembra orientato verso un'adozione delle blockchain su scala globale. Tuttavia, è necessario risolvere una serie di problematiche affinché ciò possa avvenire. Il difetto forse più importante da questo punto di vista, è che la tecnologia blockchain, per come è stata progettata, non è scalabile. Non sarebbe pertanto capace di supportare un volume di transazioni pari o superiore a quello di circuiti di pagamento globali come Visa o MasterCard (si pensi ad esempio che Visa ha circa 736 milioni di utenti attivi, contro i circa 5,8 milioni di Bitcoin). Mentre vengono studiate possibili soluzioni per aggirare il problema nelle blockchain esistenti, sono nate piattaforme alternative alle blockchain che promettono una maggiore scalabilità, nonché facilità nel costruire applicazioni basate su esse. E' il caso di Radix DLT. In questa tesi ho deciso di concentrarmi su Radix DLT e di realizzare una piccola applicazione per provare con mano gli strumenti di sviluppo a disposizione.

La tesi è strutturata come segue:

- Nel primo capitolo si presenta una breve panoramica sulla storia e sulle caratteristiche principali della tecnologia blockchain. In particolare

si parlerà di Bitcoin come punto di partenza delle blockchain, e successivamente dell'arrivo di Ethereum, con cui la tecnologia blockchain divenne non più applicabile esclusivamente al campo delle criptovalute, ma bensì la base per lo sviluppo di applicazioni decentralizzate, dette DApps, per i più svariati compiti. Si evidenzierà poi come la mancanza di scalabilità della blockchain che ne impedisca un'adozione su scala globale. Infine, si presenterà la piattaforma Radix DLT, un'alternativa scalabile alle blockchain classiche.

- Nel secondo capitolo si presenterà l'idea per la realizzazione di un'applicazione utilizzando la libreria JavaScript messa a disposizione degli sviluppatori dal team di Radix. L'idea per l'applicazione riguarda il pagamento di royalties in campo musicale. Si vuole così verificare la fattibilità di costruire applicazioni basate sulla piattaforma Radix. Verrà presentata prima l'idea che si vuole realizzare, e successivamente l'architettura dell'applicazione.
- Nel terzo capitolo si affronteranno le scelte progettuali seguite durante l'implementazione dell'applicazione. Si esporrà poi il funzionamento dell'applicazione, con tanto di immagini di supporto. Nella conclusione si parlerà poi di quali potranno essere eventuali miglioramenti di questa applicazione.

Indice

| | |
|---|-----------|
| Introduzione | i |
| 1 Stato dell'arte | 1 |
| 1.1 La tecnologia Blockchain | 1 |
| 1.1.1 Bitcoin e la prima blockchain | 2 |
| 1.1.2 Ethereum, la blockchain general-purpose | 3 |
| 1.2 Limiti delle Blockchain: Scalabilità | 5 |
| 1.2.1 Sharding | 7 |
| 1.3 Radix DLT | 8 |
| 1.3.1 Un po' di storia | 8 |
| 1.3.2 Un milione di transazioni al secondo | 9 |
| 1.3.3 L'architettura di Radix | 10 |
| 1.3.4 Il Radix Engine | 11 |
| 1.3.5 Il Radix Ledger e lo Sharding | 13 |
| 1.3.6 Strumenti per gli sviluppatori | 16 |
| 1.3.7 Libreria JavaScript | 16 |
| 2 Presentazione dell'applicazione | 20 |
| 2.1 Cosa voglio realizzare | 20 |
| 2.1.1 L'applicazione radix-bootleg | 22 |
| 2.2 Architettura dell'applicazione | 23 |
| 2.2.1 Bootleg Service | 23 |
| 2.2.2 Front End | 24 |

| | | |
|----------|---|-----------|
| 3 | Implementazione | 25 |
| 3.1 | Utilizzo dei Token | 26 |
| 3.2 | Gestione dell'identity | 28 |
| 3.3 | Creazione di un bootleg | 31 |
| 3.4 | Acquisto di un bootleg | 32 |
| 3.5 | Visualizzazione del bootleg | 33 |
| 3.6 | Visualizzazione del saldo e delle transazioni | 35 |
| | Conclusioni | 37 |
| | Bibliografia | 40 |

Elenco delle figure

| | | |
|-----|--|----|
| 1.1 | DCS Triangle | 7 |
| 1.2 | Rappresentazione dello stack Radix | 11 |
| 1.3 | Lo stack del Engine a confronto con lo stack di una blockchain basata su smart contract (come Ethereum o Hyperledger) . . . | 13 |
| 3.1 | Funzionalità disponibili per le identity | 28 |
| 3.2 | Scelta della password per la creazione del keystore | 29 |
| 3.3 | Visualizzazione del keystore prodotto durante sportazione di un'identity locale | 30 |
| 3.4 | Creazione del bootleg | 31 |
| 3.5 | Finestra di gestione dei bootleg | 32 |
| 3.6 | Bootleg acquistato | 33 |
| 3.7 | Visualizzazione del bootleg | 35 |
| 3.8 | Visualizzazione delle transazioni che coinvolgono l'account . . | 35 |
| 3.9 | Visualizzazione del saldo dell'account | 36 |

Capitolo 1

Stato dell'arte

1.1 La tecnologia Blockchain

Una Blockchain (letteralmente “*catena di blocchi*”) è un registro pubblico e distribuito di transazioni effettuate fra partecipanti di una rete, e rientra nella famiglia delle tecnologie “Distributed Ledger” (che vuol dire appunto *Registro Distribuito*). Si tratta di un registro che non viene mantenuto da un'unica autorità centrale (come ad esempio una banca), ma ciascun partecipante della rete (ovvero, ciascun *nodo*) mantiene una copia identica del registro. Grazie a questa tecnologia, ogni nodo della rete è in grado, senza il bisogno di un'entità centrale, di concordare su una singola versione degli eventi che si verificano nella rete. Questo è un concetto molto potente, in quanto non esiste un unico punto in cui il sistema può essere attaccato, come potrebbe essere un'autorità centrale. Come si può intuire dal nome, la Blockchain è una struttura dati costituita da una catena di blocchi. Questi blocchi contengono una lista di transazioni avvenute in un certo arco di tempo, e sono interconnessi fra loro attraverso l'uso della crittografia. Oltre alle transazioni, ciascun blocco infatti contiene il codice hash del blocco che lo precede nella catena. Prima di essere aggiunto alla catena, un blocco deve essere approvato da un certo numero di nodi attraverso un meccanismo di consenso. Una volta accordatisi sul nuovo blocco, ciascun nodo aggiorna la propria copia del registro.

1.1.1 Bitcoin e la prima blockchain

La prima blockchain venne introdotta nel 2008, con la pubblicazione del whitepaper “*Bitcoin: A Peer-to-Peer Electronic Cash System*”[1], da parte di una persona (o un gruppo di persone) sotto lo pseudonimo di Satoshi Nakamoto. Nakamoto, usando un insieme di tecnologie preesistenti, creò Bitcoin, una moneta elettronica basata su di un sistema completamente decentralizzato, che non deve affidarsi a nessuna autorità centrale per l’emissione di valuta o per la validazione delle transazioni. Bitcoin consiste in:

- Una rete peer-to-peer decentralizzata
- Un registro pubblico delle transazioni (la blockchain)
- Un insieme di regole per la validazione indipendente delle transazioni e per l’emissione di valuta (le regole del consenso)
- Un meccanismo per raggiungere un consenso decentralizzato globale all’interno della rete sulla blockchain valida (l’algoritmo Proof-of-Work).

L’algoritmo di consenso PoW è stata senza dubbio l’innovazione chiave. Con l’algoritmo di consenso PoW, Bitcoin risolve elegantemente quello che è stato per anni uno dei problemi principali con cui dovettero scontrarsi le monete elettroniche, il *double spend*, ovvero quando la stessa unità di valuta può essere spesa in due o più transazioni differenti. L’invenzione di Nakamoto è anche una soluzione pratica al cosiddetto “*Problema dei Generali Bizantini*”, ovvero il problema di accordarsi sul corso degli eventi, o sullo stato di un sistema, scambiando informazioni attraverso una rete non affidabile e potenzialmente compromessa.

Un’altra componente importante di Bitcoin è senz’altro l’uso della crittografia. Come già detto infatti, ciascun blocco contiene al suo interno l’hash del blocco precedente (o *parent block*), e questa costituisce una componente chiave della sicurezza del Bitcoin. Difatti, un algoritmo di hashing produce sempre lo stesso esatto codice hash per lo stesso input, e una qualsiasi modifica dell’input produce un risultato completamente diverso. Dunque, se un

attaccante provasse a modificare un blocco, questo cambiamento si rifletterebbe nel suo codice hash. Questo causerebbe un cambiamento a catena nei blocchi successivi, che dovrebbero essere ricalcolati. Dal momento che questo richiederebbe un'enorme potenza di calcolo (e dunque consumo di energia), l'esistenza di una lunga catena rende i blocchi più vecchi *immutabili*. Inoltre, gli utenti della rete Bitcoin possiedono una coppia di chiavi crittografiche, che consente loro di provare il possesso di Bitcoin all'interno della rete. Con queste chiavi, gli utenti possono firmare transazioni per "sbloccare" una certa somma e spenderla, trasferendola ad un'altro utente.

La rete Bitcoin, avviata nel 2009, utilizza dunque la blockchain come registro pubblico distribuito delle transazioni che avvengono fra i partecipanti alla rete. Col tempo tuttavia, si cominciarono a intravedere possibili utilizzi per le blockchain che andassero oltre le criptovalute. A questo punto dunque ci si iniziò a domandare se fosse possibile costruire qualcosa di nuovo sopra Bitcoin o se invece fosse necessaria la creazione di una nuova blockchain. Costruire su Bitcoin significava essere soggetti a diverse limitazioni, dunque per avere più libertà e flessibilità, costruire una nuova blockchain era l'unica opzione, nonostante richiedesse comunque un carico di lavoro considerevole.

1.1.2 Ethereum, la blockchain general-purpose

Verso la fine del 2013, Vitalik Buterin, un giovane programmatore e appassionato di criptovalute, pubblicò un whitepaper[22] che delineava l'idea di una blockchain *turing-completa* e *general-purpose*. Successivamente, insieme a Gavin Wood, iniziò a lavorare a questa idea, finché nel 2015 venne lanciata Ethereum, una blockchain General-Purpose. Come altre blockchain, Ethereum ha una criptovaluta nativa chiamata Ether. ETH è una moneta digitale, molto simile al Bitcoin. A differenza di Bitcoin però, Ethereum può fare è programmabile, il che significa che gli sviluppatori possono usarla per creare nuovi tipi di applicazioni. Queste applicazioni decentralizzate (dette anche *DApp*, abbreviazione per *Decentralized Applications*) sfruttano i vantaggi delle criptovalute e della tecnologia blockchain.

La base per le DApp su Ethereum sono gli *smart contract*. Uno smart contract è fondamentalmente un *accordo digitale*, che viene eseguito in maniera automatica. Questo implica che qualsiasi processo che richiede un'interazione manuale fra due parti può essere automatizzato, senza il bisogno di un intermediario. Questo potrebbe portare alla cosiddetta “smart economy” in cui processi manuali soggetti ad errore vengono rimpiazzati da procedimenti automatici, completamente trasparenti ed affidabili.

Il concetto di token

Molte delle DApp costruite su Ethereum hanno la propria criptovaluta. In altre parole, ciascuna applicazione ha il proprio “token”: Per interagire con una di queste applicazioni, l'utente ha bisogno di acquistare il token nativo della DApp. Generalmente parlando, un token rappresenta qualcosa di specifico in un particolare ecosistema. Questo qualcosa potrebbe essere valore economico, un dividendo, un diritto di voto, ecc. Insomma, davvero qualsiasi cosa.

Lo standard di base per i token: ERC 20

ERC-20 definisce un'interfaccia standard che rappresenta un token. Lo standard fornisce una serie di regole comuni che tutti i token sulla rete Ethereum possono seguire per produrre i risultati attesi. In pratica l'ERC-20 consente agli sviluppatori di risparmiare tempo nella creazione di un progetto, in quanto non bisogna continuamente reinventare il modo in cui i token svolgono funzioni basilari quali i trasferimenti o il recupero di dati. Inoltre, ERC-20 consente l'interazione in maniera fluida con altri smart contracts e applicazioni decentralizzate sulla blockchain Ethereum.

ERC 721: Token non fungibili

I token definiti attraverso lo standard ERC-20 sono token *fungibili*, nel senso che ciascun token è interscambiabile. Lo standard ERC721 [6] invece

definisce un'interfaccia per token *non-fungibili* (Non-Fungible tokens o NFTs) ovvero token unici nel loro genere, che pertanto non sono interscambiabili. I NFTs consentono di rappresentare attraverso un token il possesso di dati arbitrari, incrementando in maniera drastica le possibilità di ciò che può essere rappresentato come un Token sulla Blockchain Ethereum. Ciascun NFT è legato ad un identificativo univoco, che rende ciascun token unico per il suo proprietario.

Il concetto di Token non fungibile si può applicare con facilità ad ogni tipo di oggetto collezionabile. Di fatto, i NFTs hanno creato una nuova infrastruttura per giochi basati sulla blockchain. Un esempio famoso è sicuramente il progetto CryptoKitties, un gioco che permette di comprare, vendere e scambiare carte virtuali che rappresentano CryptoKitty. Ciascun CryptoKitty è completamente unico nel suo genere, e viene rappresentato da un token non-fungibile. Lanciato nel 2017, questo gioco è stato il pioniere nell'uso dei token non fungibili per rappresentare oggetti collezionabili, e ha provato che le persone attribuiscono valore ad *oggetti digitali scarsamente disponibili* (*digitally-scarce goods*). Difatti, il valore di un singolo token può raggiungere cifre molto alte (anche più di 100mila dollari), e ad oggi sono stati scambiati CryptoKitties per un valore complessivo superiore a 27 milioni di dollari [11].

1.2 Limiti delle Blockchain: Scalabilità

Vi è sicuramente un grosso entusiasmo attorno ad Ethereum e alle blockchain in generale. Dall'applicazione esclusiva al campo delle criptovalute, si è passati, nel giro di pochi anni, da un utilizzo delle blockchain esclusivamente nel campo delle criptovalute, alla prospettiva di un utilizzo delle blockchain nei più svariati ambiti: alcuni esempi potrebbero essere il campo della sanità, la protezione del copyright o le votazioni elettroniche [14]. In futuro si ipotizza quindi una diffusione crescente di questa tecnologia. Tuttavia, uno dei problemi ormai noti delle blockchain, che costituisce un limite a tale diffusione è quello della loro bassa scalabilità. Per scalabilità si intende la capacità

di un sistema di supportare un volume crescente di lavoro aggiungendo risorse al sistema stesso. Questo aspetto non è stato fino ad ora un problema particolarmente rilevante, a causa dell'utilizzo relativamente di nicchia delle blockchain. Tuttavia, dal momento che l'insieme delle applicazioni che possono essere costruite basandosi sulle blockchain si è notevolmente espanso, la scalabilità diventa un fattore importante: Un'adozione della blockchain su scala globale porterebbe certamente ad avere un utenza sempre maggiore e di conseguenza un volume di transazioni molto più alto. L'aumento del carico di lavoro che si prospetta in futuro non sarebbe attualmente supportabile.

La scalabilità delle blockchain è complicata perché, per come questa tecnologia è stata progettata, è richiesto che ciascun nodo mantenga l'intero stato e processi ogni singola transazione che venga registrata sull'intera rete. Mentre ciò garantisce un livello molto alto di sicurezza, al tempo stesso limita notevolmente la scalabilità, poiché così facendo, una blockchain può processare tante transazioni quante ne può processare un singolo nodo. Dunque, principalmente per questo motivo, Bitcoin è limitato a circa 3-7 transazioni al secondo, mentre Ethereum a circa 7-15 [16]. Per avere un esempio numerico, si pensi che Facebook può gestire circa 175,000 richieste al secondo, il che vuol dire che Ethereum, al suo livello massimo di performance, è comunque almeno 10,000 volte più lento rispetto a Facebook. Solitamente si fa riferimento al problema della scalabilità come *scalability trilemma*. Secondo questo trilemma, non è possibile avere un sistema che sia completamente decentralizzato, consistente e scalabile allo stesso tempo, ma è necessario un *tradeoff*: si possono scegliere al massimo due di queste proprietà. Il trilemma può essere visualizzato graficamente nella forma di un *triangolo DCS*[16, 17](Figura 1.1).

Per la community di Ethereum, ricercatori e sviluppatori, la scalabilità costituisce probabilmente la sfida principale da risolvere per permettere alle applicazioni basate sulla blockchain di raggiungere un'adozione di massa. Attualmente esistono due percorsi principali che sono stati studiati per incrementare la scalabilità delle blockchain. Da un lato si cerca di costruire dei

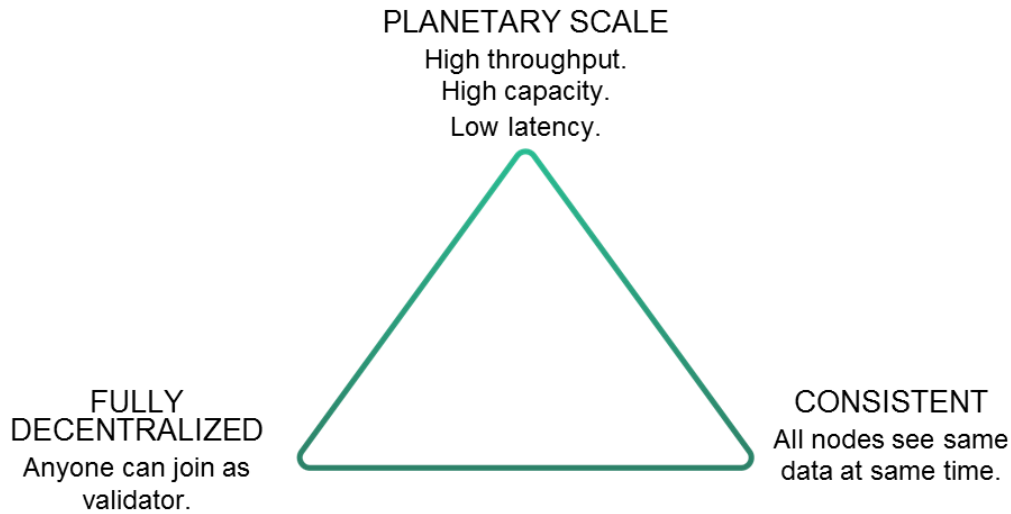


Figura 1.1: DCS Triangle

protocolli al di sopra della blockchain base (chiamati *protocolli di livello 2*) senza cambiarne la struttura, che implicano l'invio di gran parte delle transazioni su catene secondarie. Dall'altro lato si cercano soluzioni che migliorino il design della blockchain di base, e una possibile soluzione da questo punto di vista è quella di utilizzare lo *sharding*.

1.2.1 Sharding

Lo sharding (letteralmente *frammentazione*) è un concetto di lunga data nel campo dei sistemi distribuiti. Si parla di sharding quando un grande database viene partizionato in database più piccoli, in modo tale che i nodi del sistema distribuito debbano operare solo su una porzione del database, piuttosto che sull'intero database. Effettuare lo sharding della blockchain significa semplicemente partizionare la catena principale in catene più piccole e più veloci, rendendo il sistema complessivo più scalabile. Il modo per farlo sarebbe dividere lo stato e la storia delle transazioni sulla catena principale in partizioni più piccole chiamate *shard*. Ad esempio, uno schema di Sharding su Ethereum potrebbe realizzare uno shard per tutti gli indirizzi che iniziano

con 0x00, uno per quelli che iniziano con 0x01 e così via.

Nella forma più semplice di sharding, ciascuno shard ha la propria storia di transazioni, e l'effetto delle transazioni in uno shard ha un effetto solo sullo shard stesso. In forme più avanzate di sharding esiste una qualche forma di comunicazione tra uno shard e l'altro (*cross-shard*), dove una transazione all'interno di uno shard può generare eventi all'interno di altri shard.

Attualmente, Ethereum sta lavorando alla realizzazione dello sharding, ed ha rilasciato una *roadmap*, che elenca tutte le fasi che dovrebbero portare allo sharding della blockchain Ethereum. Al giorno d'oggi, queste fasi non sono ancora state ultimate. Esistono però anche altri progetti che utilizzano lo sharding per ottenere scalabilità. E' il caso di Radix DLT.

1.3 Radix DLT

Radix DLT è una nuova piattaforma DLT (*Decentralized Ledger Technology*) che si propone come alternativa alle blockchain classiche come Bitcoin o Ethereum, realizzata per essere scalabile fin dal principio e per consentire la costruzione di applicazioni su di essa con facilità. E' bene sottolineare che Radix è una tecnologia DLT, ma non si basa sulla Blockchain (quest'ultima infatti altro non è che l'implementazione più famosa di una DLT), ed utilizza un design completamente nuovo.

Radix utilizza un nuovo algoritmo di consenso chiamato Cerberus, basato su un algoritmo BFT (*Byzantine Fault Tolerant*) *three-phase commit* e sullo sharding per creare un sistema sicuro e scalabile. Attualmente, il progetto (diventato recentemente open source) è ancora in fase di sviluppo, e non è ancora disponibile una rete pubblica.

1.3.1 Un po' di storia

La storia di Radix inizia nel 2013 [12], quando il suo creatore, Dan Hughes, iniziò ad eseguire una serie di test per verificare quali fossero i limiti della scalabilità di Bitcoin. Il massimo che riuscì ad ottenere da questi test fu un

numero di transazioni per secondo (*Transactions Per Second* o *TPS*) fra 700 e 1,000. Si tratta di cifre molto basse, sapendo che, ad esempio, Visa era in grado di processare circa 24,000 TPS.

Dan, da questo punto in poi, tentò di percorrere diverse strade per aumentare il throughput di transazioni, talvolta riuscendoci, a scapito però della sicurezza. Dopo alcuni tentativi infruttuosi, Dan teorizzò ciò che divenne il primo nucleo di Radix, ovvero il registro decentralizzato Tempo. Tempo ha una struttura basata sullo sharding, che raggruppa fra di loro le transazioni correlate e separa quelle non correlate, ed utilizza un meccanismo di consenso basato sui *Logical Clock* di Leslie Lamport [9] (un meccanismo semplice per costruire un ordine parziale e relativo degli eventi). Dan iniziò quindi a cercare membri per formare un team, che iniziò a lavorare su Tempo, sulla costruzione della rete Radix e sul Radix Engine, cioè il livello applicativo di Radix (in sostanza, la parte con la quale interagiscono gli sviluppatori).

1.3.2 Un milione di transazioni al secondo

Tutto questo lavoro portò a compiere un test che replicò per intero tutte le transazioni avvenute su Bitcoin dal momento della sua creazione, durante il quale il sistema raggiunse la velocità di 1M di TPS. Questo fu certamente un grosso traguardo per Dan e il suo team, dimostrando l'enorme scalabilità ottenibile grazie a Tempo. Tuttavia, in seguito emersero delle vulnerabilità che esponevano Tempo a due possibili vettori di attacco. Il primo è stato ribattezzato dagli sviluppatori di Radix "*Weak Atom Problem*". Esso consiste in una situazione in cui un piccolo gruppo di nodi può creare una situazione in cui il consenso è sufficientemente debole da permettere di influenzare transazioni già concluse. Il secondo vettore di attacco era invece attraverso un *attacco Sybil*. Tempo infatti usava un nuovo meccanismo per proteggersi da questo tipo di attacco, chiamato *Mass*, che aumentava la reputazione dei nodi all'interno della rete per la loro buona condotta nel corso del tempo. Tuttavia, è emerso che *Mass* non avesse grande valore per i nodi onesti, e questo apriva la strada ad un possibile mercato secondario, dove attori male-

voli avrebbero potuto acquistare reputazione (attraverso Mass) per un valore inferiore al vero.

Nonostante gli sforzi del team per cercare di risolvere questi problemi, nessuna soluzione valida venne trovata. Per questo motivo, Tempo è stato abbandonato (anche se solo in parte) e la pubblicazione di una rete pubblica Radix, inizialmente prevista per la fine del 2019, è stata spostata in data da definirsi. Al momento della scrittura di questa tesi (Febbraio/Marzo 2020), il team è al lavoro su un nuovo algoritmo di consenso, chiamato *Cerberus*. In particolare, in data 3 Marzo 2020, è stato pubblicato il whitepaper su Cerberus.

1.3.3 L'architettura di Radix

Radix è fondamentalmente un software che viene eseguito su una serie di nodi (tipicamente computer) che insieme costituiscono una rete; Utilizzare la rete Radix, significa connettersi e comunicare con un nodo. Radix facilita questo compito, fornendo delle librerie in Java e JavaScript, che possono essere integrate all'interno di un'applicazione. Il *Radix Engine* e il *Radix Ledger* (Figura 1.2) sono i due livelli che formano il software che viene eseguito su ogni nodo [23]. In questo, Radix è simile ad altre piattaforme DLT, che hanno un qualche tipo di livello applicativo (tipicamente una macchina virtuale che esegue smart contracts) e un "Livello ledger" (tipicamente una blockchain). Combinati fra di loro, il Radix Engine e il Radix Ledger, forniscono una piattaforma scalabile e su cui è facile costruire applicazioni.

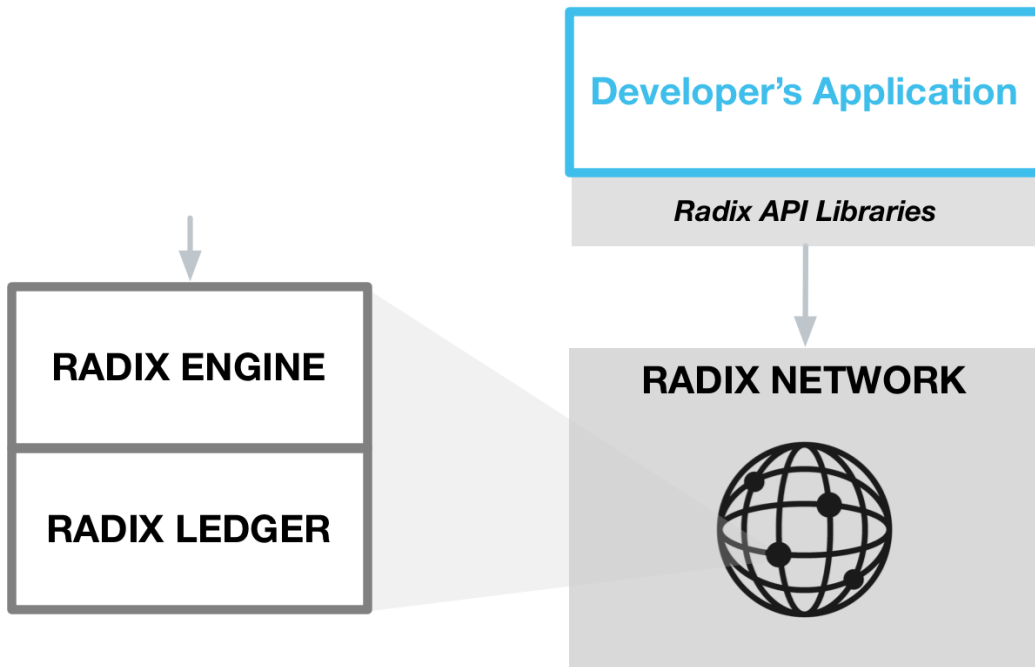


Figura 1.2: Rappresentazione dello stack Radix

1.3.4 Il Radix Engine

Il Radix Engine [19] contiene una serie di componenti pre-costruiti sicuri che sono gestiti attraverso l'API di Radix. In un certo senso, è un livello applicativo *“asset oriented”*: gli sviluppatori definiscono gli asset più importanti per la loro applicazione (che siano utenti, denaro, prodotti, ecc.), come questi asset sono correlati fra di loro e quali sono le transazioni consentite. Il Radix Engine consente agli sviluppatori di mappare facilmente questi asset in componenti Radix predefiniti e personalizzabili, definiti all'interno della Radix Engine Library [24]. La personalizzazione di questi componenti consente di definire come questi devono essere gestiti: chi può crearli o distruggerli, chi li controlla o li possiede, quali sono i passi da seguire e come esso dipende da altri asset. Una volta che gli asset sono configurati e creati, il Radix Engine si assicura che avvengano soltanto le transazioni valide fra di essi. Il Radix Ledger a sua volta elimina i conflitti possibili fra le transazioni valide, e registra il risultato. Questo implica che le garanzie di sicurezza per le transazioni

sono incorporate nella piattaforma in se. Creare asset e costruire transazioni su Radix viene gestito attraverso semplici chiamate API effettuate attraverso le librerie fornite. Non vi è dunque il bisogno di definire smart contract, dal momento che lo sviluppatore può incorporare il codice delle librerie Radix direttamente nella propria applicazione.

Come funziona il Radix Engine?

Al di sotto dell'API Radix, il Radix Engine esprime gli asset e le transazioni possibili fra di essi come macchine a stati finiti. Il vantaggio delle macchine a stati è che sostanzialmente descrivono un sistema in base a ciò che può avvenire. In breve, invece che eseguire tanti controlli “if-then” per evitare comportamenti errati, le macchine a stati semplicemente non sono in grado di seguire comportamenti errati per definizione. Il modello su cui si basa il Radix Engine è detto “Atom model”, dove gli “Atom” sono transazioni che contengono al loro interno diverse “particles”. Se tutto è valido e non vi sono conflitti, queste particles aggiorneranno tutte le macchine a stati che rappresentano gli asset dell'applicazione definiti dallo sviluppatore.

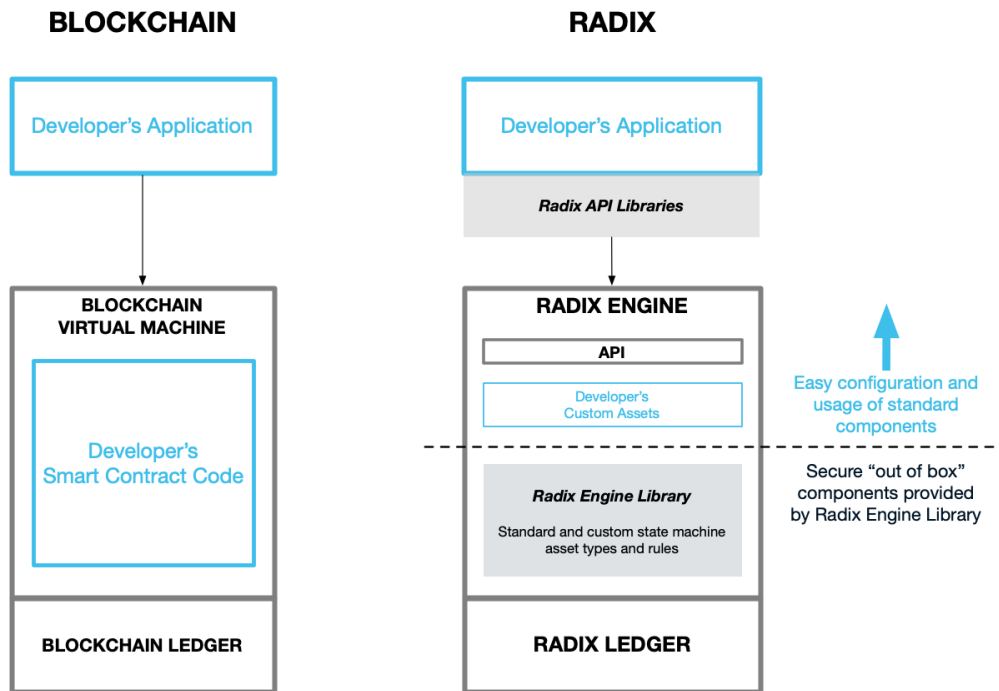


Figura 1.3: Lo stack del Engine a confronto con lo stack di una blockchain basata su smart contract (come Ethereum o Hyperledger)

1.3.5 Il Radix Ledger e lo Sharding

La maggior parte delle implementazioni dello sharding, partono da una singola struttura, come può essere una blockchain, che viene divisa in shards man mano che le sue dimensioni aumentano. Gli sviluppatori di Radix sono giunti alla conclusione che una divisione in shards effettuata in questo modo non avrebbe funzionato, in quanto ogni cambiamento successivo alla struttura degli shards, richiederebbe una riorganizzazione dell'intera rete, che più la rete è grande, e più sarebbe dispendiosa. Invece che partire da una singola struttura e poi dividerla a seconda delle necessità, quello che hanno fatto è stato stabilire fin dall'inizio come questa struttura sarebbe stata divisa in shard. Più precisamente, il Radix Ledger è stato definito in maniera da poter essere diviso in 18.4 *Quintillioni* di shards (2^{64}). Si tratta di un numero

incredibilmente grande, basti pensare che, se si dovessero distribuire equamente tutti i dati di Google su questa quantità di shard, ciascuna partizione occuperebbe soli 4 bit. Questa struttura dati non solo consente a Radix di salvare una così grossa quantità di dati, ma rende anche facile l'operazione di ricerca dei dati, grazie ad un processo *deterministico* (ovvero un processo che, dato un certo insieme di input, resituisce sempre lo stesso risultato) che consente di determinare in quale shard risiede un particolare dato. A ciascuno di questi dati è associato un indirizzo, solitamente il mittente di una transazione o di un messaggio. L'indice dello shard in cui risiede un indirizzo è calcolato in maniera deterministica, dividendo lo *shard space* totale per il modulo della chiave pubblica usata per generare quel particolare indirizzo. Questo consente a chiunque di localizzare tutti gli eventi (ad esempio, le transazioni) associati a quel particolare indirizzo.

Raggruppare le transazioni correlate

Questo modo deterministico di indicizzare i dati fornisce dunque sia un modo semplice di trovare uno specifico dato, sia un modo per raggruppare transazioni correlate fra loro, e dividere quelle non correlate. La potenza di questo approccio è più chiara se si pensa a come viene effettuata la ricerca di dati su una lockchain. Sappiamo bene che le transazioni su una blockchain vengono raggruppate in blocchi, che sono poi disposti in maniera da formare una catena. Il modo in cui le transazioni vengono disposte nei blocchi tuttavia, è di fatto casuale. Dunque, per trovare tutte le transazioni effettuate da un particolare indirizzo, è necessario cercare lungo tutta la Blockchain. Questo implica scorrere una per una anche centinaia di milioni di transazioni (come nel caso di Bitcoin) per trovare tutte quelle relative ad un singolo indirizzo. Il raggruppamento di transazioni correlate è importante anche per quanto riguarda il throughput della rete. La mia chiave pubblica, determina lo shard in cui risiede il mio account. Dunque potrò inviare soltanto transazioni da quello shard: se ad esempio invio due transazioni, una ad Alice e una a Bob, entrambe avranno come origine lo stesso shard.

Questo permette di rintracciare facilmente i tentativi di *double spend*. Allo stesso modo, questa struttura dati deterministica assicura che due transazioni che si trovano su due shard differenti non siano in alcun modo correlate. Pertanto, non vi è alcun bisogno di assicurarsi che non vi siano conflitti fra le due transazioni. Questo implica che le transazioni non correlate siano completamente asincrone, dunque tutti gli shard esistenti possono processare transazioni contemporaneamente e in parallelo.

Incentivi per i nodi

Una volta ottenuto un ledger altamente partizionato, la sfida successiva è quella di assicurarsi di avere abbastanza nodi nella rete che mantengano una quota sufficiente di shard, in maniera da assicurare la sicurezza dell'intero ledger. Quando un nodo entra a far parte della rete Radix, genera un'identity. Questa identity viene usata per identificare il *root shard* di quel nodo, ovvero lo shard che quel nodo dovrà sempre mantenere. Poi, in base alle capacità di calcolo del nodo, esso può aumentare (o diminuire) il numero di shard serviti. Maggiore è il numero di shard serviti, più è probabile ricevere ricompense e *fees* per il proprio operato. Dunque ciascun nodo cerca di mantenere più shard possibili, diminuendo se non riesce a restare sincronizzato. All'inizio, vi saranno certamente meno account e meno transazioni, per cui è probabile che tutti i nodi mantengano tutti gli shard. Più è alto il numero di nodi che si occupa di uno shard, più quello shard è sicuro, grazie al numero maggiore di copie ridondanti mantenute. Man mano che la rete cresce, la politica di incentivi di Radix è progettata in maniera tale che gli shard poco serviti diventino un'opportunità finanziaria per i proprietari dei nodi. Meno nodi servono un particolare shard, più saranno alte le ricompense per validare le transazioni che hanno origine da quello shard. I nodi avranno perciò tutto l'interesse a servire più shard possibili, privilegiando quelli meno serviti.

Cerberus

Cerberus, ossia l'algoritmo di consenso usato da Radix, si basa sullo stesso concetto di avere uno *shard space* predefinito come Tempo, ma al tempo stesso si basa su una serie di strumenti crittografici ben collaudati, che gli forniscono forti garanzie per quanto concerne la sicurezza. Attualmente, di Cerberus sappiamo che utilizza un meccanismo di consenso BFT (*Byzantine Fault Tolerant*) *three-phase commit*, e che utilizzerà la Proof-of-Stake (PoS) come meccanismo di guardia contro attacchi Sybil. Il whitepaper su Cerberus[10] è stato rilasciato soltanto recentemente, in data 3 Marzo 2020, durante la fase finale di stesura di questa tesi.

1.3.6 Strumenti per gli sviluppatori

Radix DLT offre agli sviluppatori interessati a creare applicazioni che interagiscano con il registro distribuito di Radix due librerie: una libreria Java e una libreria JavaScript, entrambe ancora in versione beta. Non essendovi una rete pubblica, per testare le librerie è necessario comunicare con la rete di testing, la *BETANET*, che può essere anche emulata in ambiente locale. Per questa tesi ho deciso di utilizzare la libreria JavaScript per realizzare una piccola applicazione che presenterò nei prossimi capitoli.

1.3.7 Libreria JavaScript

La libreria JavaScript Radix è costruita per essere eseguita sia client side che server side. E' costruita interamente utilizzando TypeScript, permettendo agli sviluppatori di costruire applicazioni più robuste sfruttando il *type checking*. Inoltre, tale libreria segue il paradigma *reactive programming*, e si appoggia alla libreria RxJS.

Concetti fondamentali

Prima di iniziare a parlare dell'applicazione, è necessario fare una panoramica sui concetti di base della libreria:

- *Universe*: Il concetto di Universe rappresenta la rete Radix. Attualmente sono disponibili diversi Universe di testing ai quali connettersi. Essendo Radix basato sullo sharding, ciascun Universe è segmentato in un certo numero di shards.
- *Atom*: Un Atom rappresenta tutti gli eventi che si verificano all'interno di un Universe, e che hanno l'effetto di lo stato del registro. Ciascun atom contiene al suo interno almeno un *enpoint address* di destinazione.
- *Address*: Un Address risiede su un particolare Shard, e costituisce punto di partenza e punto di arrivo per qualunque Atom nel Radix Universe. Si tratta di un riferimento ad un particolare Account, e permette ad un utente di ricevere fondi e/o dati sulla rete ad altri utenti. Gli indirizzi Radix vengono generati partendo da una chiave pubblica e da un checksum dell'Universe.
- *Account*: Un Account rappresenta tutti i dati salvati sul ledger per un particolare utente. Questi dati possono essere i token in possesso dell'utente, assieme ad altri dati arbitrari. Un account ha una serie di *account systems*, nei quali vengono salvati gli atom ricevuti dall'account. Gli account system di default sono:
 - *Il Transfer System*, che mantiene una lista delle transazioni in cui è coinvolto l'account, così come il saldo dell'account per tutti i differenti tipi di token.
 - *Il Radix Messaging System*, che gestisce le diverse chat di messaggistica Radix a cui l'account partecipa.
 - *Il Data System*, usato per dati personalizzati salvati sul ledger.
 - *Token Definition System*, usato per gestire i token definiti dall'utente.

E' possibile accedere ai dati di questi sistemi attraverso l'utilizzo di mappe ES6, oppure è possibile effettuare l'iscrizione (*subscribe*) ad un

subject RxJS, che emetterà un aggiornamento ogni volta che un sistema riceve un nuovo atom dalla rete.

- *Identity*: Un'Identity è una chiave privata associata ad un certo account, che può essere usata per firmare atom e per leggere dati criptati.
- *Transaction Builder*: Il Transaction builder è il componente della libreria che si occupa di creare ed inviare alla rete qualsiasi tipo di Atom che il ledger radix possa accettare. Gli Atom che si possono creare con il transaction builder sono:
 - *Transfer Atom*: realizza il trasferimento di un item (e.g. valuta) ad un certo indirizzo.
 - *Payload Atom*: contiene dati inviati ad uno o più indirizzi.
 - *Radix Message Atom*: contiene un messaggio (caso particolare di payload atom).
 - *Mint Atom*: effettua il *minting* (cioè la creazione) di una quantità specificata di token.
 - *Burn Atom*: effettua il *burning* (cioè, in sostanza, l'eliminazione) di una quantità specificata di token.

Le librerie offrono anche le funzionalità necessarie per definire dei token personalizzati. I token definiti dall'utente possono essere *single-issuance* o *multi-issuance*: i token multi-issuance, di cui si può fare il minting dopo la creazione, mentre i token single-issuance sono limitati all'importo specificato nella definizione del token.

Radix Wallet

Radix offre anche il software Desktop Wallet, un'applicazione che l'utente può utilizzare per memorizzare la propria identity. Il Wallet inoltre offre all'utente un'interfaccia da cui è possibile:

- Controllare il saldo del proprio account.

- Inviare token ad un altro account.
- Avviare una chat di messaggistica istantanea con un altro account.
- Richiedere token di test all'account Faucet.

Capitolo 2

Presentazione dell'applicazione

2.1 Cosa voglio realizzare

L'applicazione che ho deciso di realizzare prende ispirazione dal progetto open-source *Shared Royalty Non-Fungible Token* (SRNFT) [7], avviato dal Web3Studio, una organizzazione di Consensus che si occupa di Ricerca e Sviluppo nel campo delle tecnologie Blockchain e delle DApp. Lo scopo di tale progetto è quello di rendere qualunque sistema di Royalty, dall'industria del gas e del petrolio, fino all'intrattenimento, gestibile attraverso la blockchain Ethereum. Tale progetto si basa sul Token SRNFT che si presenta come una estensione del Token ERC721, ed offre tre funzionalità principali:

1. Gestire e distribuire cash flow futuri verso destinatari multipli, chiamati Franchisors.
2. Rendere unici e tracciabili asset digitali off-chain.
3. Creare incentivi per i proprietari di tali asset affinché non li diffondano in maniera impropria al pubblico.

Il SRNFT mantiene una lista di account Ethereum chiamati “franchisors” nella forma di un array, in maniera che un singolo token possa gestire cash flow verso parti multiple, in base ad una formula di distribuzione di royalties

(*Royalty Distribution Formula* o RDF). La RDF implementata può variare in base a quelli che sono gli incentivi che si vogliono creare nei diversi casi d'uso.

Vediamo come nell'implementazione di Web3Studio [7], il contratto per ciascun token SRNFT mantenga una semplice struttura dati:

```
struct Token {
    // The list of franchisors
    address[] franchisors;

    // A list of payments (in wei). Payment indices line up to
    // what payment was made for the matching franchisor
    uint256[] payments;

    // For any franchisor, what is the next payment index that
    // a
    // withdrawal should next consider.
    // (More details on this part later.)
    mapping(address => uint256) franchisorNextWithdrawIndex;
}
```

Ogni volta che viene aggiunto un nuovo Franchisor, solitamente quando un token viene trasferito da proprietario all'altro, vengono effettuate alcune operazioni:

```
function _addFranchisor (
    address franchisor, uint256 payment, uint256 tokenId)
{
    // A new franchisor is added to the list
    token.franchisors.push(franchisor);

    // Payments sent will be associated with that franchisor
    token.payments.push(msg.value);

    // We set where this franchisor can withdraw from
    token.franchisorNextWithdrawIndex[franchisor] = token.
        payments.length;
}
```

Un token ERC721 può avere un solo proprietario, e questa restrizione viene mantenuta nel SRNFT (gli stessi autori del progetto specificano che sarebbe interessante modificare tale aspetto, in modo da avere account multipli come proprietari del token). Dunque, nonostante vi possano essere più franchisor, solo quello aggiunto più recentemente ha la possibilità di trasferire il token ad un nuovo proprietario.

Il Web3Studio presenta come caso d'uso per questo token, un'applicazione specifica per il campo dell'industria musicale, chiamata *Bootleg*, che ho utilizzato come spunto per l'applicazione da realizzare.

2.1.1 L'applicazione radix-bootleg

Ispirandomi dunque all'applicazione “fittizia” Bootleg di Web3Studio, ho deciso di realizzare una mia versione di questo progetto, che fosse però basata sulla libreria JavaScript di Radix DLT.

L'idea di base dell'applicazione, è quella di consentire ai musicisti di costruire una nuova fonte di guadagno basata sulla distribuzione dei bootleg, ovvero le registrazioni degli spettacoli dal vivo realizzate dai fan (cosiddetti *bootlegger*). Mentre tradizionalmente i bootleg si diffondono in maniera non ufficiale [15], in questo caso di artisti ricevono royalties dalla vendita dei bootleg che condividono con i bootlegger. Inoltre, anche i fan che acquistano un bootleg, hanno la possibilità di condividere con l'artista le royalties generate da vendite future di tale contenuto. Questo costituisce un incentivo per i fan a non copiare il video e a diffonderlo in maniera indipendente, ad esempio attraverso altri canali non ufficiali (come ad esempio BitTorrent). Si presume comunque che i bootlegger siano in qualche modo autorizzati alla registrazione, in modo più o meno professionale, del concerto dell'artista. Tuttavia, questo discorso è al di fuori dello scopo di questa tesi.

L'applicazione permette ai bootlegger di caricare i bootleg inserendo tutte le informazioni necessarie e di metterli così a disposizione degli altri utenti. Così come nell'applicazione per Ethereum, il bootleg viene rappresentato da un token univoco. Chi accede all'applicazione, ha la possibilità di visualiz-

zare un elenco dei Bootleg disponibili e di acquistarli. Una volta acquistato un bootleg, il pagamento viene inizialmente diviso fra il bootlegger e l'artista. L'utente che acquista un bootleg riceve così il token univoco per quel bootleg, e ottiene la possibilità di visionare liberamente la registrazione del concerto. Dopo l'acquisto, l'utente viene aggiunto alla lista dei franchisor, e in quanto tale condividerà con artista e bootlegger le royalties generate dai futuri acquisti del bootleg da parte di altri utenti.

2.2 Architettura dell'applicazione

Radix Bootleg è un'applicazione strutturata principalmente in due parti: Il Frontend e il Bootleg Service. Il Frontend è la parte che si interfaccia con l'utente, e comunica con il Bootleg Service per svolgere le funzionalità relative ai token bootleg e ai pagamenti. Il Bootleg Service in sostanza implementa le funzionalità che, in una DApp per Ethereum, sarebbero realizzate con uno Smart Contract.

2.2.1 Bootleg Service

Il Bootleg Service ha le seguenti funzionalità:

- Creazione del token nativo dell'applicazione, il BTLG.
- Creazione del token univoco per il bootleg.
- Salvataggio su database delle informazioni relative al bootleg.
- Divisione del pagamento per un bootleg tra Artista, Bootlegger ed eventuali franchisors, sulla base di una Royalty Distribution Formula (RDF).
- Fornire agli utenti la lista dei bootleg presenti nell'applicazione.
- Consente ai franchisor di visualizzare i bootleg che hanno acquistato, inviando l'url della registrazione su richiesta.

Il Bootleg service dunque si interfaccia sia con il Frontend che con il database dell'applicazione. Il database ha lo scopo di mantenere i “metadati” del bootleg token, ovvero: titolo, descrizione, URL del contenuto, nonché gli indirizzi di artista, bootlegger e franchisors, in maniera tale da sapere chi sono i destinatari delle royalties generate da un particolare bootleg. Il Bootleg Service ha inoltre il compito di verificare che un franchisor, nel momento in cui richiede di visualizzare un bootleg, possieda effettivamente il token per quel bootleg.

2.2.2 Front End

Il front-end invece realizza l'interfaccia utente dell'applicazione, ed offre le seguenti funzionalità:

- Consente ai franchisor di visualizzare i bootleg che hanno acquistato.
- Autenticarsi attraverso il Radix Wallet.
- Accedere alla lista dei bootleg disponibili e quelli acquistati.
- Visionare i bootleg acquistati.
- Visualizzare il saldo dell'account.
- Visualizzare le transazioni dell'account.

Il front-end comunica con il Bootleg Service per ricevere le informazioni sui bootleg presenti nel sistema, nonché per l'invio dei pagamenti e della divisione di questi tra i diversi destinatari.

Capitolo 3

Implementazione

Sia per la realizzazione del front end che del Bootleg Service ho utilizzato TypeScript, sfruttando le funzionalità offerte dalla libreria JavaScript di Radix.

Per velocizzare lo sviluppo del Bootleg Service, ho usato come base di partenza un server realizzato con TypeScript e WebPack. Ad esso ho aggiunto il codice per la connessione ad un database locale MongoDB, per tenere traccia di tutte le informazioni relative ai bootleg. Dunque, il Bootleg Service è a tutti gli effetti un Server. Dunque, per comunicare con il Bootleg Service il Frontend utilizza delle normali richieste HTTP. Per realizzare queste richieste, ho utilizzato la libreria Axios [25].

Per il front end invece ho utilizzato il framework Vue.js, in quanto permette di realizzare un interfaccia utente in maniera piuttosto semplice, e inoltre consente di mantenere le astrazioni di Radix (Atom, Transazioni, ecc.) separate dal codice HTML.

Altri strumenti utili per lo sviluppo del progetto sono stati:

- La documentazione ufficiale della libreria JavaScript [26]:
- Gli esempi di codice forniti nella documentazione, per mostrare le funzionalità basilari implementabili con la libreria JavaScript: [27]

- Applicazione `radixdlt-js-showcase`, come esempio di frontend `Vue.js` che utilizza le funzionalità della libreria `JavaScript`: [28]
- I canali di comunicazione, principalmente `Telegram` e `Discord`, messi a disposizione della community, che consentono di interagire direttamente con gli sviluppatori del team di `Radix`.

Scelta dell'Universe

Come già accennato nel primo capitolo di questa tesi, il concetto di `Universe` rappresenta la rete `Radix`. Attualmente sono disponibili diversi `Universe` di testing ai quali connettersi per testare la propria applicazione (un po' come le reti messe a disposizione di `Ethereum` per le `DApp` in fase di sviluppo). La prima cosa che deve fare un'applicazione all'avvio è connettersi con uno di questi `Universe` a disposizione. Attualmente gli `Universe` disponibili per la libreria `JavaScript` sono: `BETANET_EMULATOR`, `BETANET`, `LOCALHOST_SINGLENODE`, `LOCALHOST` e `SUNSTONE`. In una prima versione dell'applicazione ho deciso di utilizzare l'`Universe` `BETANET_EMULATOR`, in quanto consentiva di avviare l'applicazione senza dover prima avviare il `Betanet Emulator` sulla propria macchina.

Solo successivamente ho deciso di aggiungere anche una versione che utilizzasse l'`Universe` `LOCALHOST_SINGLENODE`, che richiede l'avvio del `Betanet Emulator` per simulare la rete `betanet` sulla propria macchina. E' possibile avviare l'emulatore attraverso un'immagine `Docker`, seguendo la procedura spiegata nella documentazione ufficiale di `Radix` [29].

3.1 Utilizzo dei Token

Token Nativo (BTLG)

Per effettuare i pagamenti dei bootleg all'interno dell'applicazione, viene usato il token `BTLG`. Il `Bootleg Service` definisce il token `BTLG` al suo primo avvio come un token multi-issuance. Nel momento in cui un utente crea una

nuova identity, il frontend invia una richiesta al Bootleg Service per ricevere una somma iniziale di token BTLG token. In sostanza, il Bootleg Service prende il posto del Faucet, ovvero un account particolare che ha lo scopo di inviare dei token nativi di test. La scelta di creare un token nativo per l'applicazione con il Bootleg Service, e di inviarne una quantità predefinita a ciascuna nuova identity è stata presa soltanto durante l'implementazione del progetto. Inizialmente l'idea era quella di utilizzare il token nativo della piattaforma Radix (denominato XRD) richiedendone l'invio al Faucet Service all'avvio dell'applicazione. Tuttavia, dal momento che l'Universe in Radix influenza la creazione degli indirizzi, in ciascun Universe esiste un Faucet con un indirizzo differente. Non trovando da nessuna parte l'indirizzo del faucet per l'Universe BETANET_EMULATOR, ho dunque optato per la creazione di un token nativo all'interno dell'applicazione.

Token univoco per il bootleg

Il Bootleg Service ha anche il compito di definire un Token univoco per ciascun bootleg nel momento della sua creazione. Il possesso di questo Token da parte di un utente è ciò che indica al sistema che tale utente ha acquistato il relativo bootleg, e pertanto è autorizzato a visualizzarlo. Inizialmente il bootleg token viene mantenuto dall'account Radix del Bootleg Server, fino a che un utente non decide di acquistare il bootleg.

3.2 Gestione dell'identity

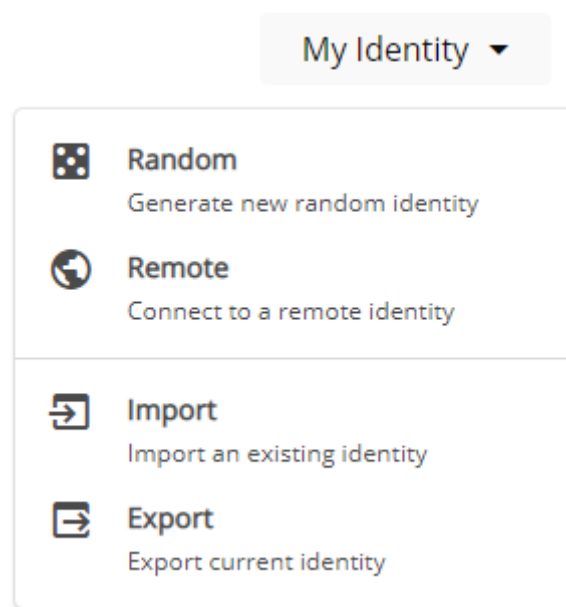


Figura 3.1: Funzionalità disponibili per le identity

La libreria Radix offre delle funzionalità per la gestione delle identity all'interno di un'applicazione. La funzione “encryptKey”, date in input un'identity e una password, cifra la prima producendo in output un oggetto JSON detto anche *keystore*. L'identity viene decifrata usando la funzione “decryptKey”, fornendo in input il keystore e la password utilizzata per produrlo. Per la gestione dell'identity all'interno dell'applicazione sono disponibili due diverse opzioni:

- Generare una nuova identity locale.
- Connettersi ad un'identity remota pre esistente, salvata nel Wallet software dell'utente.
- Importare un'identity locale esistente.
- Esportare l'identity locale corrente.

Identity locale

Quando l'utente genera una nuova identity locale, questa viene automaticamente cifrata con una password di default e il keystore viene salvato nella memoria del browser utilizzando la proprietà "Window.localStorage" dell'API Web Storage. Quando il frontend viene caricato, controlla la proprietà localStorage, e se è presente un keyStore salvato in precedenza, lo decifra e carica l'identity nell'applicazione. In questo modo, se l'utente chiude e riapre il browser, ritroverà la stessa identity. Quando l'utente genera una nuova identity locale, l'identity presente in localStorage viene sovrascritta e non si può più recuperare. Se l'utente crea un'identity e vuole salvarla per riutilizzarla in un secondo momento, ha la possibilità di *esportare* la propria identity: Facendo con la funzione "Export" l'applicazione chiede all'utente di inserire una password per cifrare la chiave, e una volta scelta la password, mostra all'utente il keystore JSON prodotto.

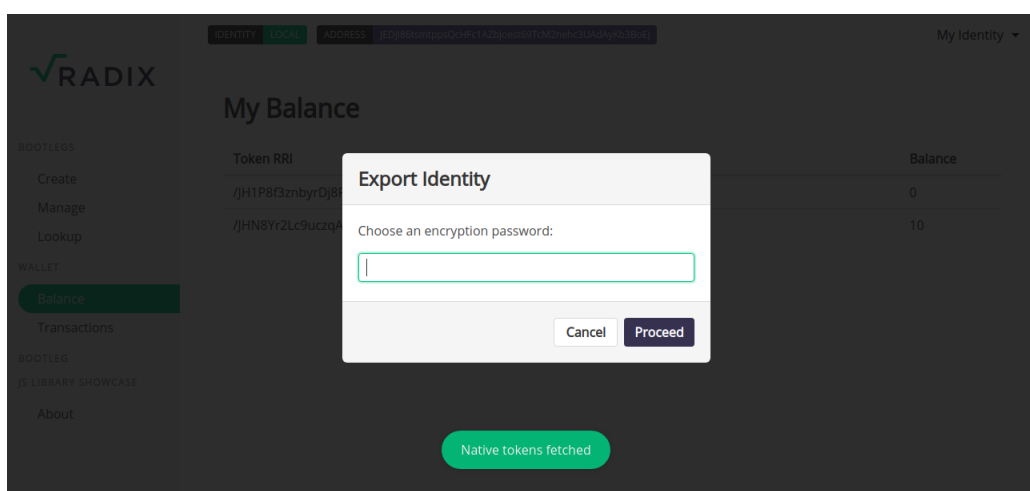


Figura 3.2: Scelta della password per la creazione del keystore

L'utente può così copiare il contenuto del keystore e salvarlo all'interno di un file, in modo da poterlo recuperare in un secondo momento. L'utente può successivamente importare l'identity creata in precedenza con la funzione

import: L'utente inserisce il keystore e la password associata per decriptare il keystore, e così facendo l'identity viene importata nell'applicazione.

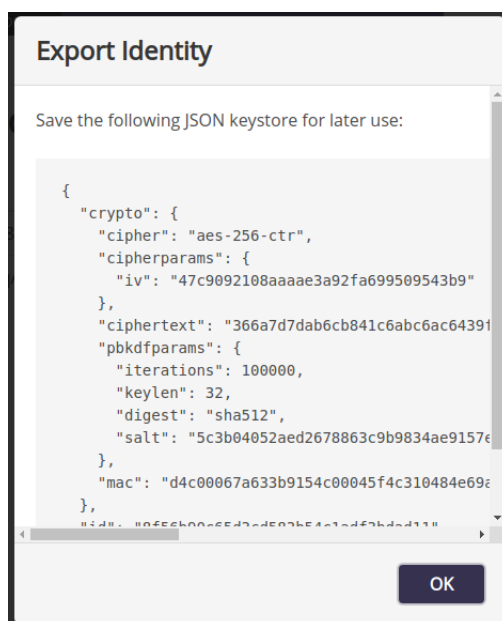


Figura 3.3: Visualizzazione del keystore prodotto durante sportazione di un'identity locale

Identity remota

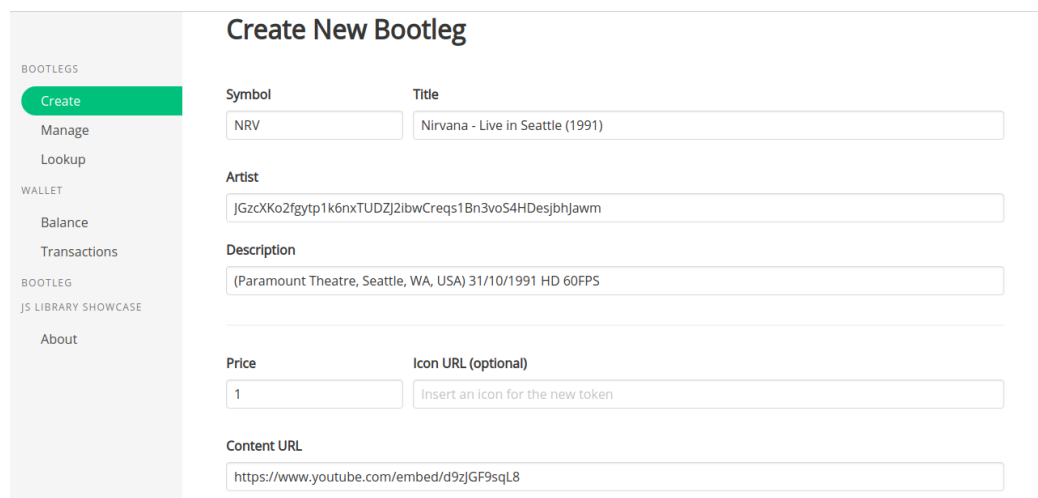
Usando un'identity remota invece, l'applicazione userà la chiave privata salvata nel Radix Wallet dell'utente, che pertanto non necessita di essere salvata nel browser. In questo caso, l'applicazione invia una richiesta al Radix Wallet. Una volta che l'utente accetta la richiesta, l'applicazione potrà usare la chiave privata mantenuta nel Wallet per firmare atom per conto dell'utente. Mentre il Frontend consente la gestione di un'identity per conto dell'utente, il Bootleg Service ha una propria identity, che viene creata al suo primo avvio, e ripristinata agli avvii successivi. Anche in questo caso vengono utilizzate le funzioni `encryptKey` e `decryptKey`, combinate con la scrittura del keystore su un file `.json`. Al suo avvio, il Bootleg Service cerca il file contenente il

keystore, e se lo trova, ne decifra il contenuto e carica l'identity, altrimenti ne crea una nuova.

3.3 Creazione di un bootleg

Per creare un bootleg, è necessario inserire:

- Simbolo usato per il token;
- Titolo del bootleg;
- Indirizzo dell'artista;
- Prezzo del bootleg;
- Url del bootleg (Per semplicità sono stati utilizzati dei link di video su YouTube);



Create New Bootleg

Symbol **Title**

NRV Nirvana - Live in Seattle (1991)

Artist

JGzcXKo2fgyp1k6nxTUDZJ2libwCreqs1Bn3voS4HDesjbhJawm

Description

(Paramount Theatre, Seattle, WA, USA) 31/10/1991 HD 60FPS

Price **Icon URL (optional)**

1 Insert an icon for the new token

Content URL

https://www.youtube.com/embed/d9zjGF9sqL8

Figura 3.4: Creazione del bootleg

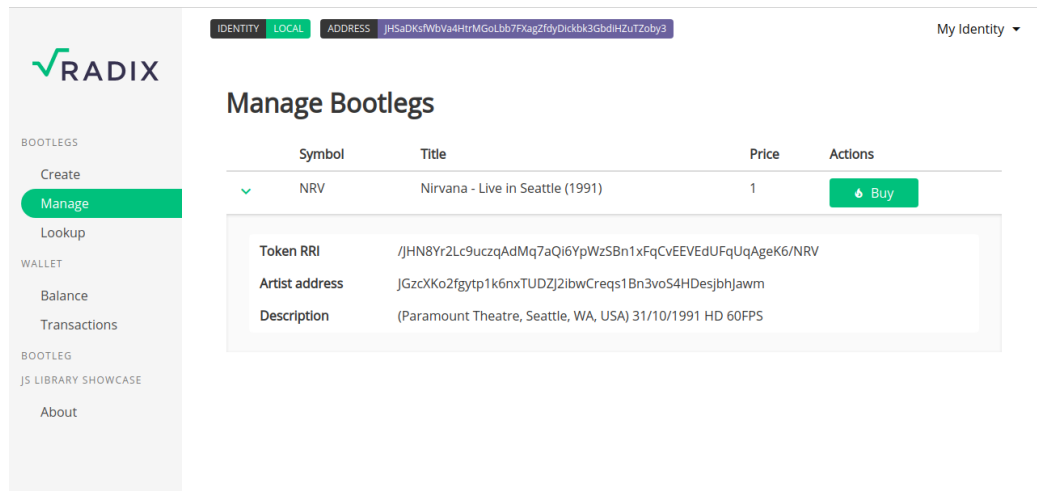
Quando l'utente dà la conferma, questi vengono inseriti in una richiesta http ed inviati al Bootleg Service, che provvederà a creare il token per il bootleg e salverà tutti i dati inviati dall'utente sul database, insieme all'identificativo del token appena creato, ed agli indirizzi di bootlegger e artista.

Terminato l’inserimento, il Bootleg Service invierà all’utente l’identificativo del token come conferma.

3.4 Acquisto di un bootleg

L’applicazione recupera la lista dei bootleg inseriti sul database inviando una richiesta al Bootleg Service, che esegue una query e recupera tutti i dati dei bootleg (ad eccezione dell’url del video), e li invia al frontend.

Per ciascun bootleg nella lista, il front end verifica se sono presenti nell’account dell’utente, e in caso contrario mostra un pulsante “Buy”.



The screenshot shows the RADIX application interface. On the left is a sidebar with the RADIX logo and navigation links: BOOTLEGS (Create, Manage, Lookup), WALLET (Balance, Transactions), BOOTLEG, JS LIBRARY SHOWCASE, and About. The 'Manage' link is highlighted. The main content area is titled 'Manage Bootlegs'. At the top, there's a header with 'IDENTITY LOCAL' and 'ADDRESS' followed by a long alphanumeric string. Below this is a table with columns: Symbol, Title, Price, and Actions. A single bootleg is listed with Symbol 'NRV', Title 'Nirvana - Live in Seattle (1991)', and Price '1'. The 'Actions' column contains a green 'Buy' button. Below the table, a detailed view of the bootleg is shown with fields: Token RRI, Artist address, and Description.

| Symbol | Title | Price | Actions |
|--------|----------------------------------|-------|---------|
| NRV | Nirvana - Live in Seattle (1991) | 1 | Buy |

| | |
|----------------|---|
| Token RRI | /JHN8Yr2Lc9ucqAdMq7aQi6YpWzSBn1xFqCvEEVEdUFqUqAgeK6/NRV |
| Artist address | JGzcKko2fgytp1k6nxTUDZJ2ibwCreqs1Bn3voS4HDesjbhJawm |
| Description | (Paramount Theatre, Seattle, WA, USA) 31/10/1991 HD 60FPS |

Figura 3.5: Finestra di gestione dei bootleg

Quando un utente acquista un bootleg, l’applicazione verifica che il suo account possieda un ammontare sufficiente di BTLG. Se sì, il pagamento viene inviato al Bootleg Service, insieme ai dati del bootleg da acquistare. Il Bootleg Service può così dividere il pagamento in parti uguali tra artista, bootlegger ed eventuali franchisor.

Una volta che il pagamento è stato diviso tra i vari destinatari, il Bootleg Service procede con l’invio del token all’utente ed inserisce quest’ultimo nella lista dei franchisor per quel token, aggiornando i dati sul database. In

questo modo, quando verrà effettuato un nuovo acquisto del bootleg, l'utente riceverà una percentuale del pagamento come franchisor.

3.5 Visualizzazione del bootleg

Una volta che l'utente possiede il token, nella lista dei bootleg compare il pulsante “Watch” di fianco al bootleg, che consente all'utente di visualizzare il bootleg. Dal momento che il front-end non possiede di per se l'url del video, deve richiederlo al Bootleg Service. Il modo in cui il frontend richiede ed ottiene il link al contenuto del bootleg si basa su uno schema challenge-response.

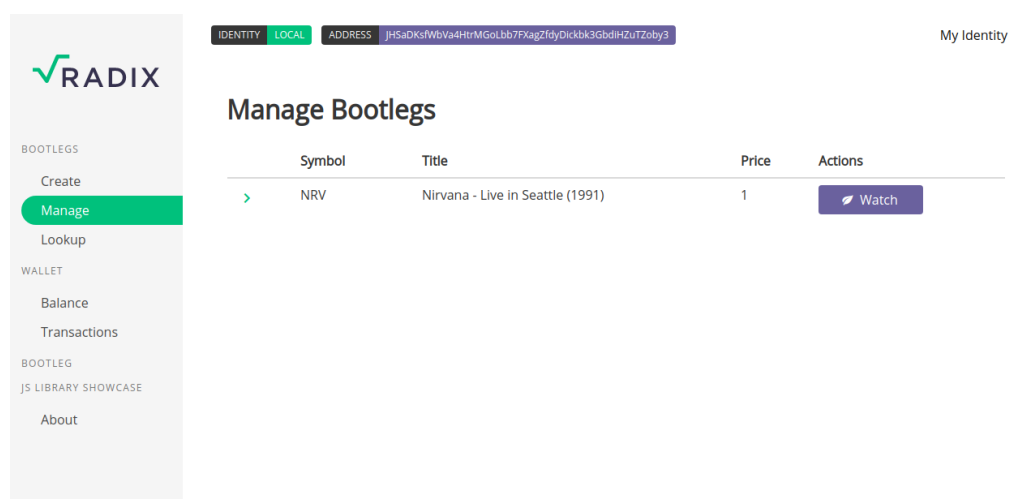


Figura 3.6: Bootleg acquistato

Cliccando su “Watch” il front end invia una richiesta al Bootleg Service per il link al bootleg. Quando il DS riceve questa richiesta, genera una *challenge* casuale, che nel caso specifico dell'applicazione, è un UUID generato con il package uuidv4. La challenge, prima di essere inviata, viene nel database insieme ad un attributo boolean *consumed*, inizialmente settato a “false”, per indicare che la challenge non è stata ancora utilizzata per accedere ad un bootleg.

Quando il frontend riceve la challenge dal Bootleg Service, crea un nuovo Payload Atom, vi inserisce la challenge ottenuta dal Bootleg Service come payload, firma questo Atom con la propria identity e lo invia al Bootleg Service con una richiesta http insieme all'URI del token del bootleg da visualizzare.

Una volta che il Bootleg Service riceve anche questa richiesta:

1. Estrae l'atom dalla richiesta e verifica la validità della firma dell'atom, cioè controlla se è stata prodotta dall'account che ha creato l'atom. Se la firma viene verificata con successo, il DS procede con la verifica della challenge.
2. Verifica la validità della challenge: il DS estrae la challenge dal payload dell'atom, ed esegue una query sul database per trovare la challenge ricevuta. Se non viene trovata la challenge, oppure viene trovata ma l'attributo "consumed" ha valore "true" (dunque è già stata utilizzata), la challenge non viene considerata valida e il DS restituisce un errore al frontend. Altrimenti, aggiorna il valore di "consumed" a true sul database e prosegue al passo successivo.
3. Verifica il possesso del bootleg: Il DS controlla il saldo dell'account dell'utente che ha inviato la richiesta e verifica se è presente il token del bootleg richiesto. Se il token è presente, allora il DS recupera il link del video dal database e lo invia come risposta al frontend.

Una volta che il frontend riceve il link del bootleg, mostra all'utente il video YouTube in modalità embedded, attraverso un elemento '<iframe>'.

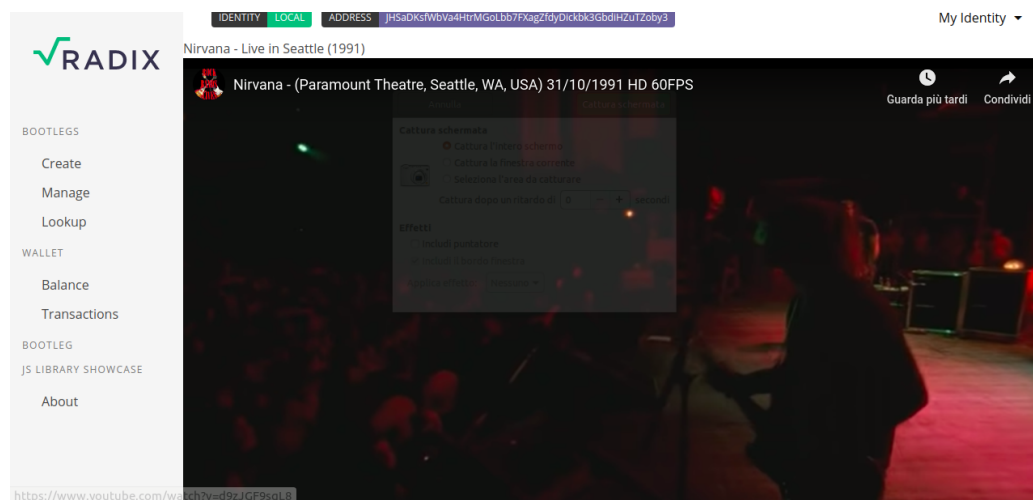


Figura 3.7: Visualizzazione del bootleg

3.6 Visualizzazione del saldo e delle transazioni

L'applicazione consente inoltre all'utente di visualizzare le transazioni ricevute e inviate, nonché il saldo dei token presenti sul proprio account.

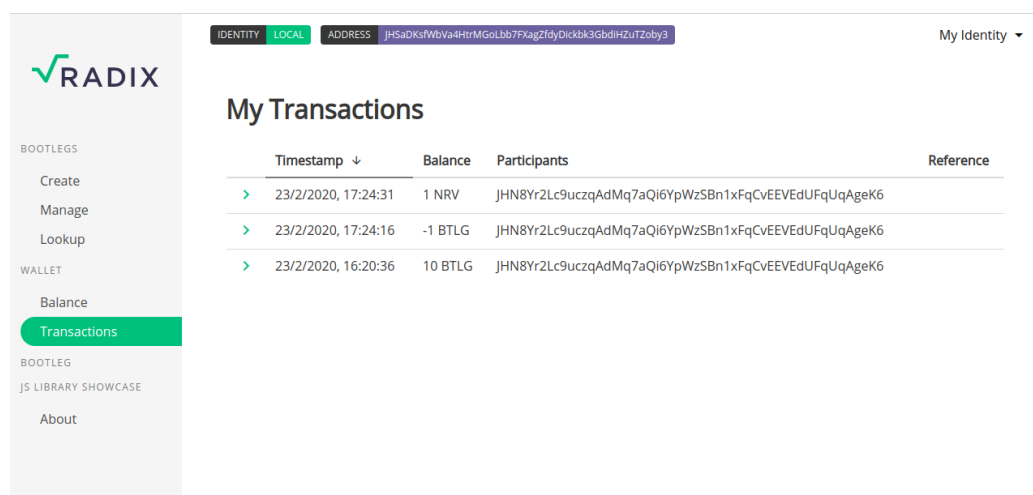
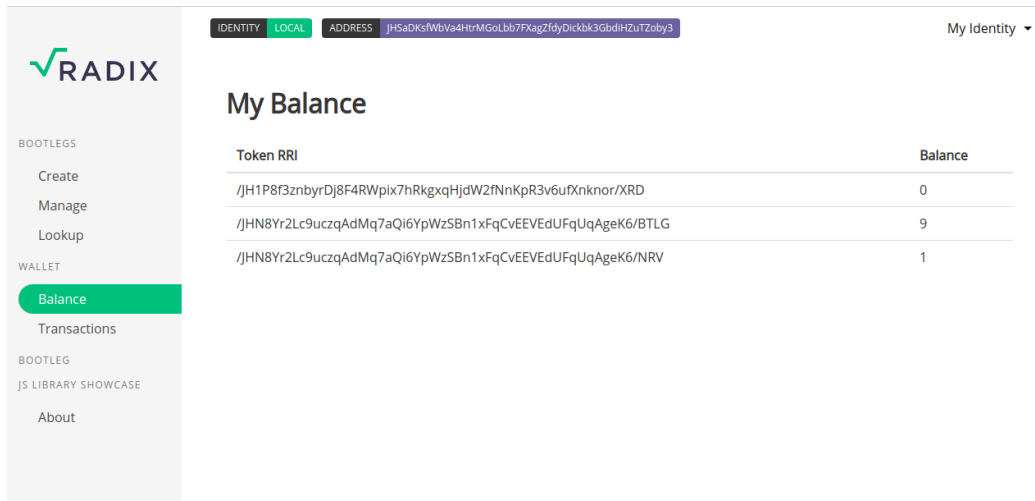


Figura 3.8: Visualizzazione delle transazioni che coinvolgono l'account



The screenshot shows the RADIX wallet interface. On the left is a sidebar with the RADIX logo and navigation links: BOOTLEGS (Create, Manage, Lookup), WALLET (Balance, Transactions), BOOTLEG, JS LIBRARY SHOWCASE, and About. The 'Balance' option is highlighted. The main content area is titled 'My Balance' and shows a table of token balances. At the top, it displays the user's identity as 'LOCAL' and their address. A dropdown menu 'My Identity' is visible in the top right.

| Token RRI | Balance |
|---|---------|
| /JH1P8f3znbyrDj8F4RWpix7hRkgxqHjdW2fNnKpR3v6uFxnknor/XRD | 0 |
| /JHN8Yr2Lc9uczqAdMq7aQi6YpWzSBn1xFqCvEEVEdUFqUqAgeK6/BTLG | 9 |
| /JHN8Yr2Lc9uczqAdMq7aQi6YpWzSBn1xFqCvEEVEdUFqUqAgeK6/NRV | 1 |

Figura 3.9: Visualizzazione del saldo dell'account

Conclusioni

In conclusione, in questa tesi è stata esposta una panoramica generale sulle blockchain, sulla loro storia e sulle caratteristiche principali di questa tecnologia, esponendo i casi di Bitcoin ed Ethereum, rispettivamente la prima e la seconda tra le criptovalute con maggior capitalizzazione di mercato. Si è visto poi come questa tecnologia, fornendo sicurezza e di consenso su di un unico ordine degli eventi in una rete distribuita senza bisogno di un'autorità centrale, abbia visto espandere i suoi possibili utilizzi. Dalle criptovalute, si è passati ad avere possibili applicazioni nei campi più disparati. Partendo da questa base, è stato esposto il problema della scalabilità delle blockchain, e di quali sono state alcune fra le soluzioni proposte, in particolar modo, lo *sharding*. Dunque è stata presentata una nuova tecnologia DLT, ovvero Radix. Essa si presenta come una piattaforma alternativa alle blockchain, ma caratterizzata da scalabilità e facilità nel costruire applicazioni che sfruttino il suo registro distribuito come base. Dopo aver visto come Radix riesce ad essere scalabile, per verificare con mano la facilità nella costruzione di applicazioni, è stata implementata un'applicazione utilizzando la libreria JavaScript di Radix.

Implementazione

Durante l'implementazione di tale applicazione, si è effettivamente riscontrato come la presenza di librerie che utilizzano linguaggi altamente diffusi consenta ad uno sviluppatore di implementare facilmente applicazione basate sul ledger Radix. Le difficoltà riscontrate durante lo sviluppo certamente

includono una documentazione che forse a volte non è abbastanza dettagliata, e l'assenza di una community di sviluppatori folta ed attiva attorno alla piattaforma Radix. La causa di ciò è probabilmente da ricercare nel fatto che si tratta ancora di un progetto in via di sviluppo. Tuttavia, i canali di comunicazione presenti (principalmente Telegram e Discord) sono stati utili per ricevere supporto durante la fase di implementazione.

Limiti dello stato attuale

Uno dei limiti dello stato attuale di Radix è certamente quello di non avere ancora a disposizione una rete pubblica stabile e robusta, nonché la necessità di dover simulare una rete in locale per poter testare le proprie applicazioni. Attualmente, non si sa ancora quando la rete pubblica di Radix sarà disponibile. Il suo rilascio è stato recentemente posticipato a causa di problemi di sicurezza che hanno portato gli sviluppatori a cambiare l'algoritmo di consenso alla base del ledger Radix, non senza generare qualche scetticismo da parte della community esistente intorno al progetto. Questo probabilmente anche per il fatto che, nonostante diversi canali di comunicazione disponibili (fra cui Telegram e Discord), si ha come membri della community la sensazione di non avere veramente un'idea chiara al 100% su ciò che succede dietro le quinte. Certamente la scelta di rendere il progetto open source [20] è stata effettuata anche con la volontà di maggiore trasparenza verso l'esterno, nonché per fornire la possibilità a soggetti esterni di dare il loro contributo al progetto.

Possibili sviluppi futuri

L'applicazione che è stata realizzata può fornire un esempio di ciò che è possibile realizzare utilizzando le librerie Radix, e dimostra come tali librerie possano essere facilmente integrate con un framework JavaScript, in questo caso Vue.js. Certamente si tratta di un'applicazione che potrebbe essere migliorata sotto molti aspetti. Alcuni sviluppi futuri dell'applicazione potrebbero includere:

- Miglioramento della gestione del token nativo.
- Differenziare le funzionalità dell'applicazione disponibili per artista, bootlegger e franchisor.
- Consentire agli artisti di autorizzare il caricamento di un bootleg sulla piattaforma.
- La memorizzazione dei bootleg non più come video di YouTube, ma su un file system distribuito (ad esempio IPFS: [30]).

Bibliografia

- [1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” p. 9.
- [2] A. M. Antonopoulos (2017, June). Mastering Bitcoin, Programming the open blockchain. O’Reilly Media.
- [3] OECD, OECD Blockchain Primer.
- [4] Consensys Academy. Blockchain Basics Book.
- [5] District0x Education Portal, What Is An ERC20 Token?
- [6] ERC721 Standard (2018) <http://erc721.org/>
- [7] Consensys Web3Studio. Bootleg, A Community for Royalty Sharing. <https://consensys.net/web3studio/bootleg>.
- [8] A. B. Bondi (2000), Characteristics of Scalability and Their Impact on Performance.
- [9] L. Lamport (1978), Time, Clocks, and the Ordering of Events in a Distributed System.
- [10] F. Cäsar, D. P. Hughes, J. Primero, S. J. Thornton, Cerberus, A Parallelized BFT Consensus Protocol for Radix, v1.0 (2020, March).
- [11] G. Minello (2019), Metodologie per la realizzazione di una Security Token Offering.

-
- [12] Sophie Donkin (2020, February), Dan And Radix's Tech Journey. Radix DLT Blog. <https://www.radixdlt.com/post/dan-and-radixs-tech-journey/>
 - [13] Matthew Hine (2019, November), Radix Engine and Ledger: The Road to Adoption. Radix DLT Blog. <https://www.radixdlt.com/post/radix-engine-and-ledger-the-road-to-adoption/>
 - [14] Carlotta Balena, Liberi dagli intermediari. Fortune Italia n. 5, Maggio 2019, p. 60.
 - [15] Bootleg, Wikipedia. <https://it.wikipedia.org/wiki/Bootleg>
 - [16] Ethereum GitHub Wiki (2019, April) Sharding FAQ.
 - [17] Trent McConaghy (2016), The DCS Triangle: Decentralized, Consistent, Scalable. Pick any two. Medium.
 - [18] The Radix Basics (2018) <https://www.radixdlt.com/post/the-radix-basics/>
 - [19] Radix Engine: A Simple, Secure Smart Contract Alternative (2019). <https://www.radixdlt.com/post/radix-engine-a-simple-secure-smart-contract-alternative/>
 - [20] Why we are open sourcing (2020). <https://www.radixdlt.com/post/why-we-are-open-sourcing/>
 - [21] Consensys Academy (2018, January), Ethereum Limitations and scaling solutions.
 - [22] Buterin, V. (2013). Ethereum white paper. GitHub repository
 - [23] Matthew Hine (2019) Radix Engine and Ledger, The road to adoption. Radix DLT Blog
 - [24] Radix Engine Library. radixdlt GitHub repository

- [25] Axios Github Repository <https://github.com/axios/axios>
- [26] <https://radixdlt.github.io/library-api-docs/radixdlt-js/2.0.0-beta.1/>
- [27] <https://docs.radixdlt.com/radixdlt-js/examples/code-examples>
- [28] <https://github.com/radixdlt/radixdlt-js-showcase>
- [29] <https://docs.radixdlt.com/kb/develop/betanet-emulator>
- [30] <https://ipfs.io/>