UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di laurea magistrale in Ingegneria Elettronica (D.M. 270/04)

Implementazione di protocolli crittografici per la certificazione di dati su blockchain.

Implementation of cryptographic protocols for the certification of data on blockchain.

Relatore:

Prof. Marco Baldi

Candidato:

Rosati Alessandro

Anno Accademico 2022/2023

Alla mia famiglia

Indice

Abstract

Nel contesto dinamico e sempre più interconnesso del mondo aziendale contemporaneo, la gestione affidabile dei dati si erge come uno dei pilastri fondamentali per il successo, la trasparenza e la sostenibilità delle operazioni. Le imprese, immerse in un panorama in costante digitalizzazione, si trovano ad affrontare una sfida cruciale: garantire l'autenticità e l'integrità dei dati che costantemente raccolgono, elaborano e condividono. In un'epoca in cui l'informazione rappresenta un asset vitale, emerge con forza la necessità di soluzioni innovative che non solo proteggano, ma certifichino l'affidabilità e l'integrità dei dati aziendali. È in questo scenario che la tecnologia blockchain si è rivelata una pietra miliare, rivoluzionando il modo in cui vengono concepite la sicurezza e la validità dei dati nelle dinamiche aziendali. Inizialmente associata al mondo delle criptovalute, la blockchain si è rivelata una soluzione all'avanguardia per garantire la certificazione e l'immutabilità dei dati.

Questo elaborato si pone come obiettivo lo sviluppo di parte di una piattaforma innovativa, specificamente dedicata alla gestione dei dati aziendali. La peculiarità di questa piattaforma risiede nella sua fondamentale adozione della tecnologia blockchain per certificare l'autenticità e garantire l'immutabilità dei dati raccolti. Il suo obiettivo primario è fornire alle aziende uno strumento affidabile, efficiente e trasparente per la gestione dei dati, mirando a promuovere la fiducia e la sicurezza nelle interazioni e nello scambio di informazioni tra le varie parti coinvolte. Il nucleo centrale di questa tesi è la data verification, ovvero il processo di verifica dell'integrità di un dato, e su come questo possa essere velocizzato in alcuni casi specifici. Verrà analizzato il caso in cui ai Merkle Trees, già considerati precedentemente, vengano accostate delle strutture Patricia Trie (Practical Algorithm To Retrieve Information Coded In Alphanumeric) per velocizzare la ricerca del dato di interesse all'interno del Merkle Tree stesso, così da velocizzare la fase di ottenimento della proof. Tale approccio può risultare conveniente quando i Merkle Trees sono costituiti da molti elementi, dal momento che i Patricia Tries sono particolarmente efficienti nell'indicizzazione di grandi set di dati.

Introduzione

In questo elaborato, l'obiettivo è quello di studiare una piattaforma (Fig. (a)), destinata alle aziende, che consenta loro di caricare dati, pertinenti al loro settore operativo, all'interno di un ambiente sicuro. La peculiarità risiede nella possibilità di certificare questi dati sfruttando la tecnologia blockchain.

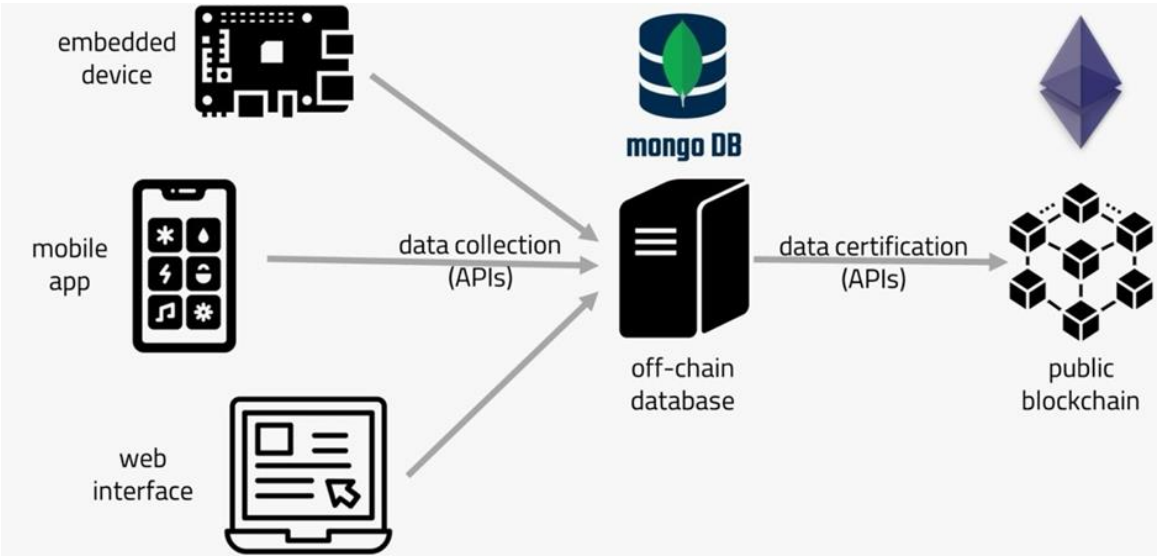


Fig. (a): schema della piattaforma oggetto di studio.

Grazie a questa piattaforma, l'utente finale avrà il potere di verificare l'autenticità di un dato mediante una query al database. Nel dettaglio, viene creato un Merkle Tree che contiene le informazioni da certificare. La radice (root hash) di questo albero costituisce il dato pubblicato sulla blockchain (Ethereum) attraverso una singola transazione.

La verifica dell'integrità richiede:

- l'informazione stessa, per calcolarne l'hash;
- il root hash della transazione nella blockchain;
- l'albero associato (memorizzato nel database);
- l'indice di posizionamento dell'informazione all'interno dell'albero.

Il caso che verrà trattato è quello di database **trustworthy**, ovvero di un database sul quale l'utente può riporre affidamento; quindi, l'utente confida nel fatto che, in un momento successivo alla certificazione, nel caso in cui voglia verificare l'integrità di un dato, può fare una query al database ed ottenere le informazioni di cui ha bisogno (proof) per poi procedere in autonomia con la verifica di integrità. In questo caso, dunque, una volta che l'utente avrà fatto una query per il dato del quale vuole verificare l'integrità, gli verrà restituita una "proof" specifica che gli permetterà di determinare l'hash root del Merkle Tree. Successivamente, questo hash sarà confrontato con l'hash registrato sulla

blockchain: se vi è corrispondenza, il dato è considerato integro e affidabile. Questo è il caso che prenderemo in considerazione da qui in avanti.
Una possibile variante di questa procedura è quella per cui, al termine della fase di certificazione, la piattaforma restituisce all'utente una label, contenente l'indice nel quale è stato posizionato il dato certificato all'interno del Merkle Tree. In fase di verifica, nella query al database l'utente dovrà anche inserire il valore di tale indice. Ciò permette di velocizzare il calcolo della proof e, di conseguenza, il processo di verifica.
Altrimenti, in caso di database **trustless**, si tiene conto anche dell'eventualità in cui un utente, che ha caricato dati e li ha certificati sulla piattaforma, in un momento successivo possa non avere più la possibilità di comunicare con la stessa: in questo caso, la scelta migliore è quella di restituire all'utente, già al termine della certificazione, tutte le informazioni (proof) di cui ha bisogno, per permettergli in futuro di procedere con la verifica, senza la necessità di fare la query alla piattaforma.
Nel caso trattato di database trustworthy, si è studiato un metodo per ottimizzare l'indicizzazione del Merkle Tree, facendo uso di una struttura Patricia Trie, così da velocizzare la ricerca del dato di interesse all'interno del Merkle Tree e, di conseguenza, la restituzione della proof in fase di verifica. Sono stati svolti dei confronti tra i due approcci (ovvero tra l'uso del solo Merkle Tree e l'uso di Merkle Tree e Patricia Trie) al variare del numero di elementi inclusi nel Merkle Tree e sono state fatte delle considerazioni in merito alle prestazioni raggiunte nei due casi (in termini di tempi richiesti per la generazione degli alberi, per la ricerca del dato e per il calcolo della proof e in termini di spazio richiesto per il salvataggio dei due alberi).
L'organizzazione seguita per esaminare questo argomento è la seguente:

- nel primo capitolo vengono mostrate le fondamenta della tecnologia blockchain, esplorando in dettaglio i vantaggi derivanti da questa innovazione: in particolare, viene posto l'accento su Ethereum, la blockchain utilizzata per questo progetto; inoltre, vengono introdotte le strutture dati (Merkle Tree e Patricia Trie) che saranno necessarie per l'implementazione della piattaforma;
- nel secondo capitolo sono mostrate alcune soluzioni che offrono un servizio di certificazione dati tramite l'uso di tecnologia blockchain;
- il terzo capitolo è dedicato a una panoramica del funzionamento della piattaforma stessa, delineando le varie componenti coinvolte, offrendo una visione d'insieme dei meccanismi sottostanti e delle loro interconnessioni;
- nel quarto capitolo viene analizzata l'applicazione del Patricia Trie alla piattaforma oggetto di studio e viene effettuato il confronto col caso d'uso del solo Merkle Tree;
- nel quinto capitolo sono presentate le conclusioni dell'analisi svolta.

Capitolo 1

Blockchain: principi di funzionamento e analisi di Ethereum

1.1 Introduzione alla blockchain

La tecnologia blockchain ha inaugurato un nuovo capitolo nell'evoluzione della gestione dei dati digitali. Alla base di questa trasformazione vi è il concetto innovativo di registri distribuiti [1], un nuovo paradigma che ha infranto le barriere tradizionali, permettendo a una rete di partecipanti di condividere e sincronizzare un registro digitale in modo distribuito, abolendo, di conseguenza, la dipendenza da un'autorità centrale.

Questa struttura di dati ha rivoluzionato il modo stesso in cui archiviamo, gestiamo e proteggiamo le informazioni. La sua nascita ha dato il via ad un ecosistema in cui la sicurezza non è affidata a un unico punto di vulnerabilità, ma piuttosto alla condivisione e alla verifica diffusa tra i partecipanti della rete: la blockchain, infatti, attraverso i suoi registri distribuiti, garantisce un'immunità senza precedenti contro la manipolazione dei dati. Ogni transazione, ogni dato che viene inserito all'interno della blockchain viene sigillato all'interno di un blocco crittograficamente connesso al precedente, formando una catena di dati. Questo meccanismo impedisce qualsiasi tentativo di alterazione o compromissione dei dati, garantendo la loro integrità nel tempo.

L'essenza della blockchain si radica nella sua struttura distribuita, in cui ogni nodo contribuisce alla validazione e alla conservazione delle informazioni: questa natura decentralizzata conferisce, di conseguenza, resilienza al sistema. La blockchain, pertanto, ha implicazioni che non sono limitate alle sole transazioni finanziarie, ma che si espandono in una vastità di settori e applicazioni. La sua adozione è testimone di una trasformazione in atto nell'economia digitale, influenzando settori quali l'istruzione, la logistica, la sanità, l'intrattenimento e, persino, la governance.

1.2 Distributed Ledger Technology *(DLT)*

La Distributed Ledger Technology (DLT) è l'infrastruttura tecnologica e i protocolli che consentono l'accesso, la convalida e l'aggiornamento simultaneo di dati, che sono registrati in più luoghi contemporaneamente. A differenza dei database tradizionali, infatti, i registri distribuiti non hanno un archivio dati centrale, ma sfruttano una rete di computer distribuita su più entità, posizioni o nodi. In un registro distribuito, ciascun nodo elabora e verifica ogni elemento, generando così un record di ciascun elemento e creando un consenso sulla sua veridicità. Un registro distribuito può essere utilizzato per registrare dati statici, come un registro, e dati dinamici, come le transazioni finanziarie.

La DLT funziona sulla base dei principi di decentralizzazione: a differenza dei tradizionali database centralizzati, la DLT opera su una rete peer-to-peer (P2P), in cui più nodi archiviano una copia completa del registro e partecipano al processo di aggiornamento e validazione delle transazioni senza l'ausilio di un'autorità centrale, riducendo, di fatto, il rischio di Single Point of Failure. Per garantire che tutti i nodi della rete abbiano una copia accurata e aggiornata del registro, è essenziale che tutti i partecipanti alla rete concordino su quali transazioni debbano essere aggiunte al registro (Fig. 1.1) e in quale ordine.

Fig. 1.1: Proprietà della Digital Ledger Technology [2].

Questo viene effettuato attraverso un algoritmo di consenso, ovvero l'insieme completo di protocolli, incentivi e idee che consente a una rete di nodi di trovare un accordo sulla validità di ciascuna transazione. Ethereum, ad esempio, usa un meccanismo di consenso basato sul Proof of Stake che trae la propria sicurezza cripto-economica da una serie di ricompense e sanzioni applicate al capitale bloccato dagli stakers. Questa struttura di incentivi incoraggia i singoli stakers a gestire validatori onesti, punisce coloro che non lo fanno, e crea un costo estremamente elevato per attaccare la rete. Quindi, esiste un protocollo che governa il modo in cui i validatori onesti sono selezionati per proporre o convalidare i blocchi, elaborare le transazioni e decidere l'ordine delle transazioni all'interno di questi blocchi. La DLT utilizza la crittografia per archiviare in modo sicuro i dati: i dati vengono crittografati prima di essere memorizzati sulla rete distribuita, garantendo che siano inaccessibili a chiunque non abbia le dovute autorizzazioni; inoltre, ogni partecipante alla rete possiede una coppia di chiavi crittografiche, una privata e l'altra pubblica: la chiave privata viene utilizzata per firmare digitalmente le transazioni o le informazioni, mentre la chiave pubblica viene utilizzata per verificare l'autenticità della firma. Ciò assicura che solo chi possiede la chiave privata

possa firmare le transazioni, garantendo l'autenticità e l'integrità dei dati. Per concludere, una volta che vengono registrati sulla DLT, i dati diventano immutabili, ovvero non possono essere cancellati o modificati retroattivamente.

Uno dei sistemi DLT maggiormente conosciuti è la blockchain, che raggruppa le transazioni in blocchi concatenati insieme e poi le trasmette ai nodi della rete.

1.3 Blockchain

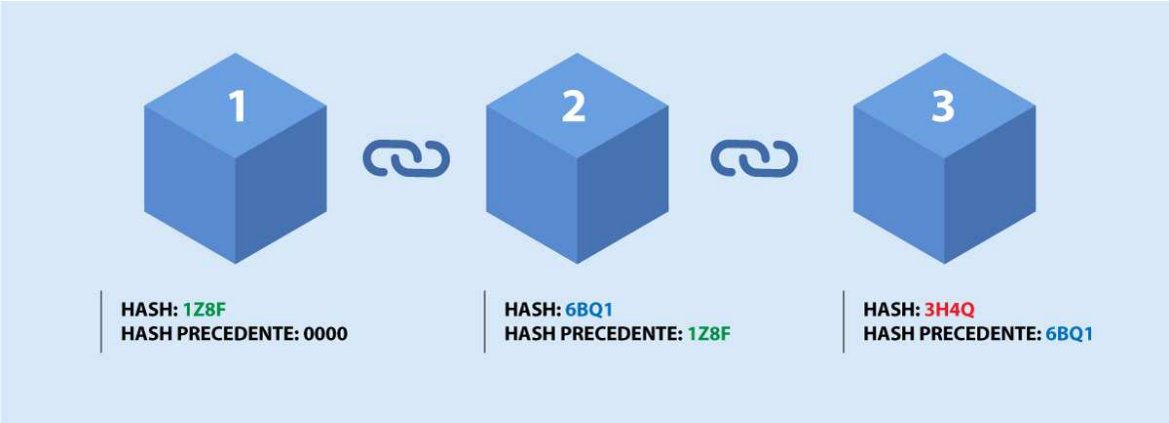


Fig. 1.2: Blockchain technology example [3].

La blockchain è un registro digitale distribuito che consente la registrazione sicura e immutabile delle transazioni o dei dati. Funziona come una catena di blocchi, ciascuno contenente un insieme di dati, collegati in modo crittografico in ordine cronologico: ad esempio, in una blockchain di criptovalute, un blocco può includere informazioni sulle transazioni, come mittente, destinatario e importo. Ciascun blocco include, oltre ai dati, un hash crittografico unico che lo identifica. Questo hash viene generato utilizzando un algoritmo che calcola un "digest" dei dati contenuti all'interno del blocco, così da ottenere una stringa di lunghezza fissa; qualsiasi modifica ai dati del blocco genererebbe un hash completamente diverso. Ogni blocco contiene, inoltre, anche l'hash crittografico del blocco precedente. Questo fa sì che si venga a determinare una vera e propria catena di blocchi, da cui deriva il termine "blockchain". Collegando ogni blocco al precedente tramite l'hash, la blockchain assicura che nessun blocco possa essere modificato senza modificare anche tutti i blocchi successivi, garantendo in questo modo l'integrità dei dati.

Tradizionalmente, le reti blockchain sono suddivise in permissionless e permissioned [4], ma recentemente è nata la necessità di un ulteriore livello di raffinamento nella definizione della tassonomia di una rete: ecco, dunque, l'introduzione di due ulteriori discriminanti, *public e private* (vedi Fig. 1.3). In una blockchain pubblica, *chiunque* è libero di unirsi e partecipare alle attività principali della rete, operando in base a uno schema incentivante che incoraggia i nuovi partecipanti a aderire e mantenere agile la rete. Le blockchain pubbliche offrono una soluzione particolarmente

efficace dal punto di vista di una architettura veramente decentralizzata, democratizzata e priva di autorità. Non esistono entità con autorità sul sistema complessivo. Una blockchain privata, invece, consente solo l'ingresso selezionato di partecipanti verificati, ovvero che abbiano passato un predeterminato processo di controllo; l'operatore di rete (in altri contesti, l'admin) ha il diritto di convalidare, sovrascrivere, modificare o eliminare le entità attestate sulla rete. Pertanto, è presente almeno una autorità centrale dotata di alto livello di privilegio.

La distinzione principale tra blockchain pubbliche e private, dunque, consiste nel controllo di chi è autorizzato a partecipare alla rete; pertanto, una blockchain privata non è realmente decentralizzata ed è rappresentabile come un registro distribuito che opera nello stesso modo di un database chiuso e sicuro, basato su tecniche di crittografia.

Passiamo ora a definire le accezioni permissioned e permissionless, ricorrendo ad alcuni esempi.

Le public permissioned blockchain consentono a chiunque di unirsi alla rete autorizzata previa un'adeguata verifica della propria identità, nonché l'assegnazione di autorizzazioni selezionate e designate per eseguire esclusivamente determinate attività sulla rete (un esempio è Ripple [5], una delle più grandi criptovalute diffuse nel 2022). Al contrario, una private permissionless blockchain rappresenta una particolarità che recentemente si sta diffondendo come ottimale per applicazioni di tipo smart contract: in modo simile alle reti public permissionless, chiunque può partecipare alla costruzione di un nodo in una rete di questo tipo; tuttavia, a differenza della blockchain pubblica, gli altri nodi ne riconosceranno l'esistenza ma non condivideranno alcun dato, in coerenza con la natura

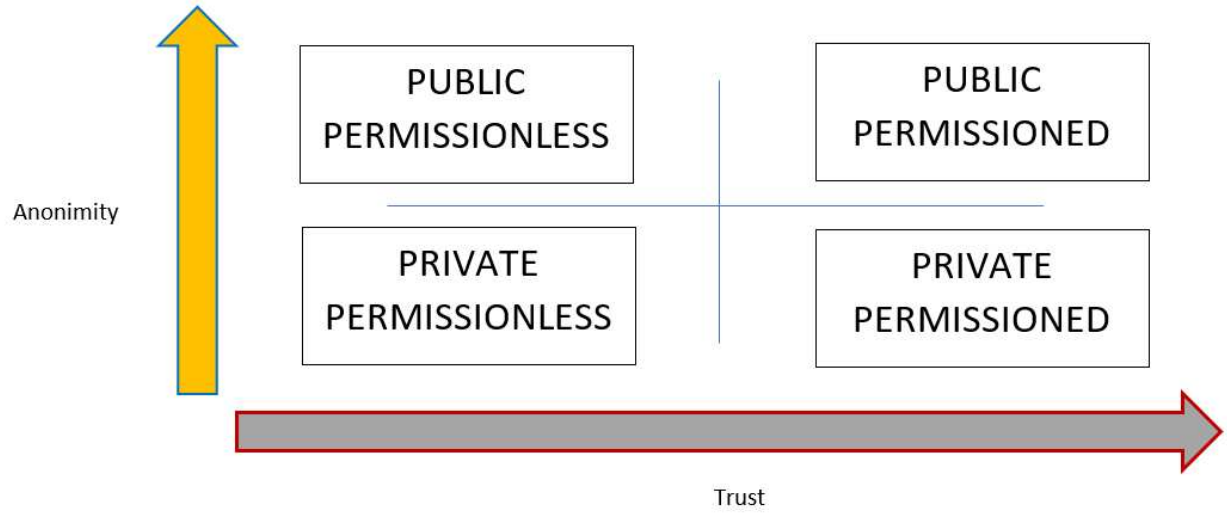


Fig. 1.3: Tipico schema anonymity/trust blockchain [4].

privata della rete. Gli smart contracts su queste reti private non solo definiscono chi è autorizzato ad eseguire azioni contrattuali, ma anche chi è autorizzato a leggere il contratto e tutti i dati relativi.

Per ottenere ciò, le soluzioni private permissionless non creano un'unica catena condivisa da tutti, ma ogni istanza di un contratto intelligente ha la sua catena ad hoc. In altre parole, la distribuzione di uno smart contract su una rete private permissionless crea automaticamente una catena (laterale) privata associata a quel contratto.

La tecnologia blockchain è stata adottata in diversi campi, sfruttando la sua struttura distribuita e crittografata per migliorare la sicurezza, la trasparenza e l'efficienza. Uno dei settori nei quali ha trovato una prima valida applicazione è quello finanziario, dove le criptovalute come Bitcoin o Ethereum consentono transazioni sicure e peer-to-peer, senza intermediari. Questa tecnologia ha anche rivoluzionato i pagamenti internazionali, accelerando e rendendo più convenienti le transazioni transfrontaliere.

La gestione della catena di approvvigionamento è un altro ambito in cui la blockchain ha suscitato interesse: la sua capacità di tracciare e autenticare ogni passaggio lungo la catena (che collega diversi attori, come fornitori, produttori, distributori, trasportatori e consumatori, e comprende molteplici attività come approvvigionamento, produzione, trasporto, stoccaggio, gestione degli inventari e consegna), garantendo la provenienza e la qualità delle merci, ha, infatti, portato a una maggiore trasparenza e riduzione delle frodi nel settore.

Nel campo della salute, la blockchain è stata adottata per creare registri medici digitali sicuri e condivisi. Ciò consente agli operatori sanitari di accedere in modo sicuro e veloce alle informazioni dei pazienti, migliorando la qualità delle cure e garantendo la privacy dei dati sensibili.

La blockchain sta anche rivoluzionando l'industria immobiliare, offrendo una piattaforma per la registrazione e la verifica delle transazioni immobiliari. Questo processo semplificato e sicuro migliora l'efficienza nelle compravendite di proprietà e assicura l'autenticità dei documenti.

Oltre a questi settori, la tecnologia blockchain trova applicazioni nell'istruzione per la gestione delle credenziali digitali, nell'energia per ottimizzare la distribuzione e il monitoraggio dell'energia, nonché nei settori dell'intrattenimento per la gestione dei diritti d'autore e la trasparenza delle transazioni.

1.4 Ethereum

Fig. 1.4: Logo di Ethereum.

Vitalik Buterin, co-fondatore di Ethereum, ha pubblicato il suo white paper nel 2013, per rispondere alle carenze di Bitcoin [6]. All'interno del white paper sono descritti in dettaglio gli smart contracts – algoritmi "if-then" automatizzati e immutabili – che consentono lo sviluppo di applicazioni decentralizzate. Sebbene le DApps (Decentralized Applications) esistessero già nel panorama delle blockchain, le piattaforme non erano tra di loro interoperabili: Buterin intendeva uniformarle attraverso Ethereum. Così è nato Ethereum 1.0, un unico spazio per decine di migliaia di applicazioni diverse, tutte conformi alle stesse regole. Questo insieme di regole è codificato nella rete e applicato autonomamente, con gli sviluppatori in grado di applicare le proprie regole all'interno delle DApps. Non c'è un'autorità centrale che modifica e fa rispettare le regole; al contrario, il potere è nelle mani delle persone che lavorano all'interno della community.

Nel corso del tempo, molti sviluppatori si sono avvicinati a Ethereum con le loro proposte decentralizzate. Nel 2016, la community ha fondato The DAO (Decentralized Autonomous Organization), un gruppo democratico che votava le modifiche e le proposte della rete. The DAO fungeva, tra le altre cose, da fondo d'investimento decentralizzato e si basava su smart contract, in modo da eliminare la necessità di un amministratore delegato per la gestione di Ethereum e prendere ogni decisione tramite votazione; tuttavia, a causa di una falla nella sicurezza, un hacker sconosciuto rubò 40 milioni di dollari dai fondi di The DAO. Per annullare il furto, The DAO ha votato per un "hard fork" di Ethereum, ovvero per una modifica sostanziale e permanente del codice della blockchain che porta alla divisione della blockchain stessa in due rami separati, abbandonando di fatto la vecchia rete e passando a un nuovo protocollo; tuttavia, questa proposta ha diviso la comunità di Ethereum, dal momento che una parte sosteneva il ripristino dei fondi tramite l'hard fork, mentre l'altra parte riteneva che la blockchain dovesse rimanere immutabile, senza alcuna interferenza esterna.

La comunità di Ethereum ha alla fine optato per l'hard fork, creando due blockchain separate:

- **Ethereum (ETH):** la versione che ha implementato l'hard fork per ripristinare i fondi rubati. Questa è la blockchain principale di Ethereum oggi.
- **Ethereum Classic (ETC):** la versione originale che non ha subito modifiche e ha mantenuto l'immagine di blockchain immutabile.

Questo evento ha dimostrato la possibilità di una divisione nella blockchain di Ethereum, creando due asset digitali separati con ideologie e visioni differenti sulla "giustezza" delle modifiche apportate alla blockchain.

1.4.1 Panoramica generale

Come Bitcoin, la rete Ethereum è costituita da migliaia di computer in tutto il mondo, grazie agli utenti che partecipano come "nodi", invece di fare affidamento su un server centralizzato. Questo rende la rete decentralizzata e altamente resistente ad attacchi esterni: è praticamente impossibile mandare il network offline. Se un computer viene disattivato, ve ne sono comunque altri che tengono in piedi la rete. Ethereum è, essenzialmente, un unico sistema decentralizzato che gestisce un computer chiamato Ethereum Virtual Machine (EVM). Ogni nodo possiede una copia di tale computer, il che significa che ogni interazione deve essere verificata in modo che tutti possano aggiornare la propria copia.

Le interazioni in rete sono considerate "transazioni" e sono memorizzate all'interno dei blocchi della blockchain di Ethereum. I validatori sono coloro che hanno il compito di validare le transazioni e creare nuovi blocchi, attraverso un sistema noto come Proof-of-Stake (PoS). Questa procedura di validazione dei blocchi è entrata in vigore a Settembre 2022, dal momento che, prima di questa data, il sistema di validazione di Ethereum era basato sulla Proof-of-Work (PoW), ossia con il lavoro svolto dai miners per validare le transazioni, proprio come accade tuttora nella rete Bitcoin; inoltre, come Bitcoin, tutte le transazioni di Ethereum sono pubbliche. I validatori trasmettono i blocchi completati al resto dei partecipanti della rete, che confermano la modifica e aggiungono i blocchi alla propria copia del registro condiviso. I blocchi confermati non possono essere manomessi e fungono da cronologia di tutte le transazioni della rete. L'Ether (ETH), token della rete Ethereum, entra costantemente in circolazione sotto forma di ricompense per i validatori derivanti dallo staking.

Ogni transazione effettuata in Ethereum prevede una commissione, chiamata "gas fee", pagata dagli utenti per interagire con la blockchain. Tutte le commissioni delle transazioni di un blocco sono parte della ricompensa per il lavoro svolto dal validatore, in modo che vi sia un incentivo a mantenere in funzione la rete. Le tariffe del gas di Ethereum possono essere piuttosto elevate in base all'attività della rete: i validatori scelgono le transazioni con le tariffe più alte e gareggiano tra loro per convalidare le transazioni per primi. Questa concorrenza spinge le tariffe sempre più in alto, congestionando la rete nei periodi di maggiore affluenza.

Per interagire con Ethereum è necessario, dunque, possedere ETH, che vengono conservati all'interno di un wallet, ovvero un'applicazione, un software o un dispositivo che consente agli utenti di gestire le loro chiavi crittografiche, interagire con la blockchain e conservare, appunto, in modo sicuro le loro criptovalute o asset digitali. Tale wallet si collega alle DApp, fungendo da "passaporto" per l'ecosistema Ethereum. Da lì, chiunque può acquistare oggetti, giocare, prestare denaro e svolgere ogni sorta di attività, proprio come su Internet tradizionale.

1.4.2 Vantaggi e svantaggi di Ethereum

Oltre alla decentralizzazione e all'anonimato, Ethereum offre anche altri vantaggi, come l'assenza di censura: se un utente pubblicasse un contenuto ritenuto offensivo su un social media tradizionale, la piattaforma potrebbe eliminare il messaggio e prendere dei provvedimenti nei confronti dell'utente; su una piattaforma social basata su Ethereum, invece, questo potrebbe accadere solo se la comunità stessa votasse per farlo. In questo modo, gli utenti con punti di vista diversi possono discutere come meglio credono: sarà poi la community a decidere cosa deve o non deve essere detto.

I membri della community impediscono, inoltre, che dei malintenzionati prendano il sopravvento della rete: qualcuno con cattive intenzioni dovrebbe, infatti, avere il controllo del 51% della rete per apportare un cambiamento, cosa praticamente impossibile. Ethereum è, pertanto, molto più sicuro di un semplice server centralizzato, che può, invece, essere hackerato.

Sebbene siano molti i risvolti positivi di questa piattaforma, Ethereum presenta alcuni problemi chiave che devono essere risolti:

- **Scalabilità**: Buterin ha immaginato Ethereum come un nuovo Internet, con milioni di utenti che interagiscono contemporaneamente; tuttavia, tale interazione è attualmente limitata in particolar modo dalle tariffe costose del gas;
- **Accessibilità**: è costoso sviluppare su Ethereum e difficile da usare per gli utenti che non conoscono la sua tecnologia. Alcune piattaforme richiedono wallet specifici, il che significa che bisogna spostare ETH dal proprio portafoglio attuale a quello richiesto.

1.4.3 Transazioni

Una transazione su Ethereum si riferisce a un'azione avviata da un conto esterno, in altre parole, da un conto gestito da un umano, non da un contratto. Ad esempio, se Bob invia 1 ETH ad Alice, il conto di Bob sarà addebitato e quello di Alice verrà accreditato. Questa azione che modifica lo stato avviene all'interno di una transazione (Fig. 1.5).

Fig. 1.5: Diagramma adattato da Ethereum EVM illustrated [7].

Le transazioni, che cambiano lo stato dell'EVM, devono essere trasmesse all'intera rete. Ogni nodo può trasmettere una richiesta di esecuzione di una transazione sull'EVM e, in seguito, un validatore eseguirà la transazione e propagherà il cambiamento di stato risultante al resto della rete.
Le transazioni richiedono una commissione e devono essere incluse all'interno di un blocco validato. Una transazione in Ethereum contiene le seguenti informazioni:

- *from* – indirizzo del mittente che firmerà la transazione. Questo sarà un indirizzo che appartiene ad un utente, in quanto gli indirizzi che corrispondono ai contratti non possono inviare transazioni;
- *recipient* – l'indirizzo ricevente (se è un indirizzo che appartiene ad un utente, la transazione trasferirà valore; se è un indirizzo che corrisponde a un contratto, la transazione eseguirà il codice del contratto);
- *signature* – l'identificativo del mittente. Viene generata quando la chiave privata del mittente firma la transazione e conferma che il mittente ha autorizzato la transazione;
- *nonce* – un contatore con incremento sequenziale, che indica il numero della transazione effettuata dal conto del mittente;
- *value* – quantità di ETH da trasferire dal mittente al destinatario (denominata in WEI, dove 1 ETH corrisponde a 10^{18} wei);
- *input data* - campo facoltativo per includere dati arbitrari;
- *gasLimit* – importo massimo di unità di gas che possono essere consumate dalla transazione. L'EVM specifica le unità di gas necessarie per ogni passaggio di calcolo;
- *maxPriorityFeePerGas* – il prezzo massimo del gas consumato da includere come mancia al validatore;
- *maxFeePerGas* – la commissione massima per unità di gas che si desidera pagare per la transazione (che include baseFeePerGas e maxPriorityFeePerGas).

Il gas è un riferimento al calcolo necessario perché un validatore elabori la transazione. Gli utenti devono pagare una commissione per questo calcolo. In Fig. 1.6 viene riportato un esempio di transazione su Ethereum:

```
{
  from: "0xEA674fdDe714fd979de3EdF0F56AA9716B898ec8",
  to: "0xac03bb73b6a9e108530aff4df5077c2b3d481e5a",
  gasLimit: "21000",
  maxFeePerGas: "300"
  maxPriorityFeePerGas: "10"
  nonce: "0",
  value: "10000000000",
}
```

Fig. 1.6: Esempio di transazione su Ethereum [7].

L'oggetto di una transazione deve essere firmato utilizzando la chiave privata del mittente. Questo prova che la transazione è stata originata solo dal mittente e non è stata inviata in modo fraudolento.

1.4.4 Smart Contracts

Uno Smart Contract è un programma che funziona sulla blockchain di Ethereum [8]. Si tratta di una raccolta di codice (le sue funzioni) e dati (il suo stato) che risiede in un indirizzo specifico sulla blockchain. Gli smart contracts sono un tipo di account presente su rete Ethereum, per cui i contratti hanno un saldo e possono essere oggetto di transazioni; tuttavia, questi non sono controllati da un utente, ma vengono invece distribuiti alla rete ed eseguiti come programmi. Gli account degli utenti possono quindi interagire con uno smart contract inviando transazioni che eseguono una funzione definita sul contratto. Gli smart contracts possono definire regole, come un contratto normale, e applicarle automaticamente tramite codice. Di default, gli smart contracts non possono essere eliminati e le interazioni con essi sono irreversibili.

Chiunque può scrivere uno smart contract e distribuirlo alla rete: per farlo, è necessario utilizzare un linguaggio di programmazione per smart contracts e avere abbastanza ETH per distribuire il contratto. La distribuzione di uno smart contract è tecnicamente una transazione, quindi bisogna pagare il gas, così come si fa nel caso di un semplice trasferimento di ETH; tuttavia, in questo caso, i costi del gas sono molto più alti. I linguaggi principalmente utilizzati per scrivere smart contracts sono due: Solidity e Vyper. Tali contratti devono essere compilati prima di poter essere distribuiti, in modo che la EVM possa interpretare e memorizzare il contratto.

Gli smart contracts sono pubblici su Ethereum e possono essere considerati come API aperte: ciò significa che uno smart contract può chiamare altri smart contracts per estendere notevolmente ciò che è possibile. I contratti possono persino distribuire altri contratti.

Una limitazione degli smart contracts è che questi, da soli, non possono ottenere informazioni sugli eventi del "mondo reale", perché non possono recuperare dati da fonti esterne alla blockchain, dal momento che fidarsi delle informazioni esterne potrebbe, infatti, mettere in pericolo il consenso, che è importante per la sicurezza e la decentralizzazione; tuttavia, è importante che le applicazioni blockchain possano utilizzare dati esterni. Per porre rimedio a questo problema, si utilizzano gli oracoli, strumenti che acquisiscono dati esterni e li rendono disponibili agli smart contracts.

Esistono, inoltre, contratti multifirma, ovvero smart contracts che richiedono più firme valide per eseguire una transazione. Questo è molto utile per evitare singoli punti di errore per contratti che detengono quantità considerevoli di ether o altri token. I contratti multifirma dividono anche la responsabilità per l'esecuzione del contratto e la gestione delle chiavi tra più parti e impediscono la perdita di una singola chiave privata che potrebbe portare a una perdita irreversibile di fondi. Per

questi motivi, i contratti multifirma possono essere utilizzati per la governance semplice di un DAO. I contratti multifirma richiedono N firme su M firme accettabili possibili (dove $N \leq M$ e $M > 1$) per essere eseguiti. N = 3, M = 5 e N = 4, M = 7 sono comunemente usati. Una multifirma 4/7 richiede quattro delle sette firme possibili valide. Questo significa che i fondi sono ancora recuperabili anche se vengono perse tre firme. In questo caso, significa anche che la maggioranza dei detentori delle chiavi deve essere d'accordo e firmare affinché il contratto venga eseguito.

1.5 Merkle Tree

In vari sistemi distribuiti e peer-to-peer, la verifica dei dati è molto importante. Questo perché gli stessi dati esistono in molteplici posizioni: quindi, se un dato viene modificato in una posizione, è importante che venga modificato ovunque. La verifica dei dati serve per assicurarsi che i dati siano uguali ovunque; tuttavia, è dispendioso in termini di tempo e computazionalmente costoso controllare l'integrità di ogni file ogni volta che un sistema vuole verificare i dati ed è per questo motivo che vengono utilizzati i Merkle Trees, così da limitare il più possibile il quantitativo di dati inviato in rete. Il Merkle Tree, definito anche come albero hash binario, è una struttura dati utilizzata per riassumere e validare in modo efficiente grandi set di dati: questo perché fa uso di hash, anziché file completi. Attualmente, il suo principale utilizzo è nelle reti peer-to-peer come Tor, Bitcoin e Git.

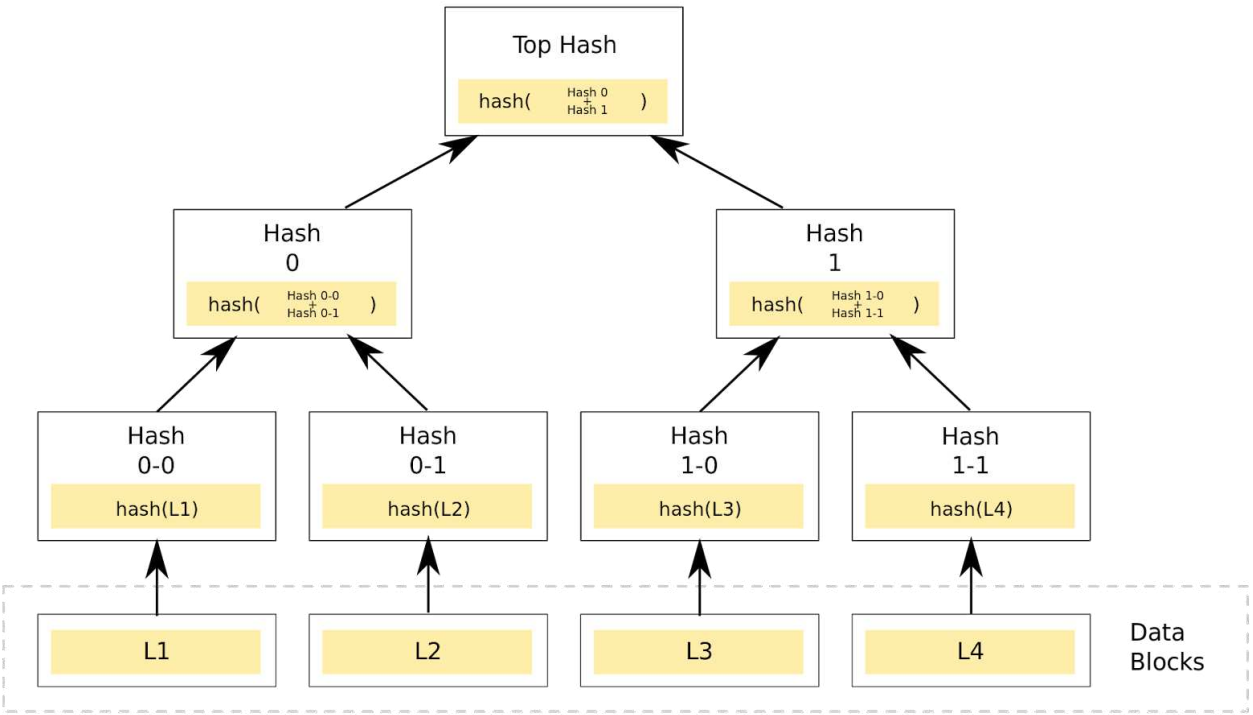


Fig. 1.7: Esempio di Merkle Tree [9].

Un Merkle Tree è un albero in cui ogni nodo foglia è etichettato con l'hash crittografico di un blocco di dati, mentre i nodi più in alto nell'albero sono gli hash dei rispettivi figli.
Nel dettaglio, inizialmente i dati vengono divisi in blocchi, ciascuno dei quali viene hashato individualmente per creare i nodi foglia dell'albero. Se il numero di foglie non fosse pari, l'ultimo blocco potrebbe essere duplicato per formare una coppia. Una volta che i nodi foglia sono stati accoppiati, l'hash di ciascun nodo non foglia viene calcolato facendo l'hash della concatenazione dei nodi foglia. Ad esempio, nella Fig. 1.7, hash 0 è il risultato dell'hashing della concatenazione di hash 0-0 e hash 0-1, ovvero hash 0 = hash (hash (0-0) + hash (0-1)), dove + denota la concatenazione. Questo processo continua ricorsivamente, combinando gli hash dei nodi non foglia per creare livelli sempre più alti dell'albero, fino a raggiungere un unico nodo radice, chiamato "Merkle Root" o "Hash Root".
Per verificare l'integrità di una porzione di dati, è sufficiente avere l'hash della foglia corrispondente e l'hash dei nodi non foglia lungo il percorso dalla foglia alla radice. Questo percorso costituisce una "proof" (o prova) che può essere utilizzata per verificare che la foglia sia parte dell'albero e che l'albero sia integro. Il vantaggio principale dei Merkle Trees è che consentono di verificare in modo efficiente l'integrità di grandi set di dati, richiedendo solo l'hash della radice e un numero limitato di hash intermedi, anziché calcolare gli hash di tutti i dati: dimostrare che un nodo foglia è parte di un dato Merkle Tree richiede il calcolo di un numero di hash proporzionale al logaritmo del numero di nodi foglia dell'albero. Ad esempio, si suppone di voler verificare l'integrità del blocco L2 nel diagramma sopra. Avendo la Merkle Root, il processo è molto semplice: si interroga la rete su (Hash 0-1) e questa restituisce (Hash 0-0) e (Hash 1). Il Merkle Tree consente di verificare che tutto sia contabilizzato con tre hash: dato Hash 0-0, Hash 1 e la radice, Hash 0-1 (l'unico hash mancante) deve essere presente nei dati.

1.6 Patricia Trie

Un Trie, o albero dei prefissi (Fig. 1.8), è un tipo di albero di ricerca, una struttura dati ad albero utilizzata per individuare chiavi specifiche all'interno di un insieme. Queste chiavi sono spesso stringhe, con collegamenti tra i nodi definiti non dall'intera chiave, ma dai singoli caratteri. Per accedere a una chiave (per recuperarne il valore, modificarla o rimuoverla), si attraversa il trie in profondità, seguendo i collegamenti tra i nodi, che rappresentano ciascun carattere nella chiave.
A differenza di quanto avviene in un albero di ricerca binario, nel trie il valore effettivo della chiave non è memorizzato direttamente nei nodi dell'albero, ma viene rappresentato dalla posizione dei nodi stessi lungo il percorso dalla radice al nodo foglia.

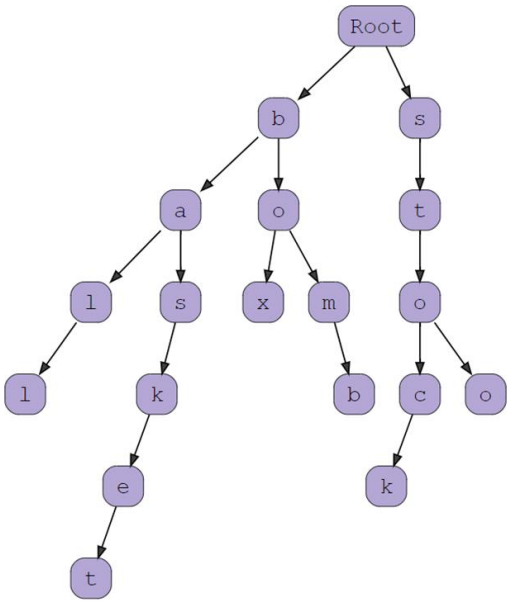


Fig. 1.8: Esempio di albero dei prefissi.

Ad esempio, supponendo di avere un trie di parole "cat", "dog" e "car", ogni carattere di queste parole diventa un nodo nel trie. Per cui:

- la radice del trie rappresenta un nodo vuoto;
- il primo livello contiene nodi per tutte le possibili prime lettere delle parole nell'insieme ("c", "d");
- il secondo livello contiene nodi per le seconde lettere delle parole ("a", "o");
- il terzo livello, infine, conterrà nodi per le terze lettere delle parole ("t", "g", "r").

Pertanto, cercando "dog" all'interno del trie, si inizia dalla radice e si segue il percorso "d" -> "o" -> "g". A questo punto, si arriva alla fine della parola e si trova un nodo foglia che indica la presenza della parola "dog". La presenza della parola è determinata dalla completezza del percorso, non dal valore memorizzato in un singolo nodo.

Questo approccio consente ricerche efficienti basate sui prefissi comuni e consente di risparmiare spazio, in quanto non è necessario memorizzare esplicitamente il valore associato ad ogni nodo, ma solo la struttura dell'albero che rappresenta le chiavi.

Un Radix Trie (Fig. 1.9) è un trie ottimizzato per lo spazio, in cui ogni nodo, che è l'unico figlio, è unito al suo genitore. Il risultato è che il numero di figli di ogni nodo interno è al massimo la radice r del Radix Trie, dove r è un numero intero positivo e una potenza x di 2, con $x \geq 1$. A differenza degli alberi regolari, i bordi possono essere etichettati con sequenze di elementi oltre che con singoli elementi. Questo rende i Radix Tries molto più efficienti per insiemi piccoli (specialmente se le stringhe sono lunghe) e per insiemi di stringhe che condividono lunghi prefissi.

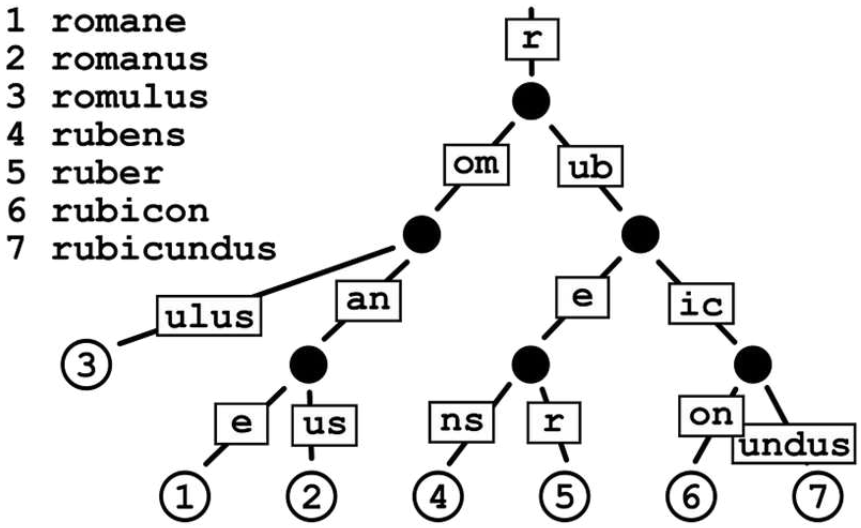


Fig. 1.9: Esempio di Radix Trie [10].

In un trie "standard", la chiave è un percorso che viene seguito passo dopo passo (cioè, un valore esadecimale alla volta) per raggiungere una destinazione, ovvero il valore di interesse. Ogni volta che viene fatto un passo lungo quel trie, ci si posiziona su un nodo.

Nei Patricia Tries esistono diversi tipi di nodi:

- Null: un nodo inesistente;
- Branch: un nodo che si collega ("si dirama") a un massimo di 16 nodi figli distinti. Un nodo di tipo "branch" può anche avere un valore proprio;
- Leaf: un "nodo terminale" che contiene la parte finale del percorso e un valore associato;
- Extension: è una novità introdotta con i Patricia Tries e, di conseguenza, ciò che li distingue dai radix tries: consiste in un nodo "shortcut" che fornisce un percorso parziale e una destinazione. I nodi di estensione vengono, infatti, utilizzati per "bypassare" nodi non necessari quando esiste solo una diramazione valida per una sequenza di nodi. Ciò permette di migliorare l'efficienza rispetto ai tries standard.

I Patricia Tries consentono una rappresentazione efficiente e compatta delle chiavi, dove la ricerca, l'inserimento e la cancellazione dei dati avvengono con complessità temporale ottimale rispetto alla lunghezza delle chiavi stesse. Questa caratteristica si traduce in prestazioni elevate in termini di tempo di accesso, rendendo i Patricia Tries una scelta ideale per applicazioni che richiedono operazioni frequenti di ricerca e aggiornamento dei dati. Nel Capitolo 4 studieremo l'implementazione di tale struttura dati per migliorare le prestazioni della piattaforma per la certificazione dei dati.

Capitolo 2

Blockchain e Certificazione: studio di piattaforme preesistenti che adottano la blockchain per la certificazione dei dati

Esistono diversi progetti e piattaforme che si occupano di utilizzare la blockchain per la certificazione dei dati. In questo capitolo verranno riportati i dettagli di due di queste piattaforme, per mostrare le soluzioni che sono state adottate e quelli che possono rappresentare dei limiti di tali architetture.

2.1 Tierion, IPFS e tecnologia blockchain

In [11], per effettuare il check dell'integrità dei dati caricati su cloud, si dà priorità all'archiviazione in sicurezza dei logs di accesso degli utenti, che sono inclusi all'interno dei cosiddetti dati di provenienza che, contenendo informazioni private degli utenti, dovrebbero essere immutabili e inaccessibili agli avversari. Questo modello è stato implementato, testato ed analizzato utilizzando IPFS (InterPlanetary File System), che è un meccanismo di archiviazione decentralizzato, è supportato dalla blockchain per archiviare i dati di provenienza del cloud e utilizza l'API pubblicamente disponibile di Tierion per archiviare il valore hash delle voci di provenienza.

2.1.1 Dati di provenienza nel Cloud: approccio decentralizzato con blockchain e IPFS

Per migliorare la disponibilità dei file, di solito il CSSP (Cloud Storage Service Provider) mantiene una copia dei dati in più posizioni. A causa di diversi motivi, come guasti o aggiornamenti delle macchine, i dati cloud possono essere spostati su un'altra macchina, sia all'interno dello stesso data center che tra data center situati in posizioni remote. Questa transizione dei dati archiviati può creare problemi/difficoltà, dal momento che raccogliere i dati di provenienza e individuare gli errori in un sistema del genere è difficile. Un modo per risolvere questo problema è raccogliere i registri da tutte le macchine in cui si spostano i dati; tuttavia, le diverse macchine che vengono utilizzate in questi casi possono avere architetture diverse e potrebbero eseguire software diversi. Pertanto, raccogliere i dati di provenienza è una sfida: i dati di provenienza devono, infatti, includere ogni singola azione eseguita sui dati stessi o sulla macchina associata a tali dati. Di solito, i dati di provenienza vengono archiviati in un database centralizzato, sotto la sorveglianza di un ente fidato che generalmente è il fornitore di archiviazione cloud, ovvero il CSSP stesso; tuttavia, non è saggio fidarsi completamente del CSSP, poiché potrebbe alterare i dati per trarne un vantaggio. Ad esempio, il CSSP potrebbe

modificare i dati di provenienza archiviati per nascondere eventuali accessi non autorizzati, poiché ha pieno controllo su quel database. Questo problema può essere risolto archiviando i dati di provenienza in modo distribuito. La tecnologia blockchain può essere una soluzione adeguata ad affrontare il problema menzionato. Poiché i dati archiviati nella blockchain sono immutabili, la fiducia tra gli utenti e il CSSP può essere aumentata se i dati di provenienza vengono conservati nella blockchain, anziché nel database centralizzato del CSSP; tuttavia, lo storage di grandi quantità di dati sulla blockchain è molto costoso. Questo problema può essere affrontato utilizzando un sistema di archiviazione di file decentralizzato, come IPFS. I vantaggi di questo approccio sono i seguenti: (1) Il database di provenienza è distribuito tra tutti gli utenti e il CSSP utilizzando IPFS. I dati archiviati in IPFS sono immutabili e quindi non possono essere alterati dall'avversario. (2) Poiché i dati di provenienza sono archiviati con ogni utente nella rete blockchain, non esiste un singolo punto di fallimento, a differenza del CSSP centralizzato. Ciò garantisce anche un'alta disponibilità. (3) Il modello proposto è stato implementato, testato ed analizzato e i risultati dell'implementazione dimostrano l'usabilità pratica del modello proposto in termini di computazione e overhead di archiviazione.

2.1.2 Schema proposto

I dati di provenienza corrispondenti ad ogni azione eseguita sui dati archiviati vengono memorizzati in un file distribuito basato su IPFS e un valore hash corrispondente ai dati di provenienza viene memorizzato in blockchain come transazione. Una lista di queste transazioni forma un blocco. Successivamente, utilizzando i dati archiviati in IPFS e gli hash nella blockchain, l'Auditor di provenienza può verificare l'integrità dei dati archiviati. L'utilizzo di tecniche decentralizzate come la tecnologia blockchain e IPFS per verificare l'integrità dei file archiviati su CSSP fornisce diversi vantaggi:

- il database di provenienza è distribuito tra tutti gli utenti e il CSSP, il che porta ancora più fiducia tra gli utenti del cloud;
- non esiste un singolo punto di fallimento;
- i valori di digest memorizzati nella blockchain, corrispondenti ai dati di provenienza archiviati in IPFS, sono completamente immutabili;
- non è possibile per nessun nodo trovare l'ubicazione di archiviazione esatta di una particolare voce di provenienza archiviata in IPFS, poiché i dati archiviati in IPFS sono anonimizzati.

Il sistema proposto è costituito da alcune entità principali, i cui ruoli vengono spiegati brevemente come segue:

- **Cloud User:** come mostrato nella Figura 2.1, tutti gli attori che utilizzano il servizio di archiviazione cloud sono utenti del cloud. Per utilizzare il servizio di archiviazione cloud, gli utenti del cloud devono prima completare la registrazione dell'utente presso il CSSP. Un utente può modificare i dati e le autorizzazioni associati ai dati archiviati. Una volta caricato il file, l'utente del cloud può optare per il servizio di provenienza dei dati;
- **Cloud Service Provider:** CSP o CSSP fornisce archiviazione come servizio agli utenti del cloud. È anche responsabile della registrazione degli utenti. Solo il CSSP conosce la vera identità degli utenti del cloud. Il gestore di provenienza all'interno del CSSP è responsabile della memorizzazione dei dati di provenienza nel database archiviato e dell'upload del valore hash dei dati di provenienza sulla rete blockchain;
- **IPFS:** questo database è distribuito tra CSSP, auditor di provenienza e tutti gli utenti del cloud. Alla rilevazione di uno dei dati di provenienza, il CSSP aggiunge i nuovi dati di provenienza ricevuti al database distribuito di IPFS. Solo il CSSP e l'auditor di provenienza avranno il permesso di scrittura su questo database, mentre tutti gli utenti del cloud potranno solo leggere e verificare l'attività del CSSP e dell'auditor di provenienza. Una volta che l'auditor avrà convalidato alcuni dati, aggiornerà lo stato corrispondente all'ingresso nel dato distribuito;
- **Blockchain Network:** memorizza il valore hash dei dati di provenienza. Solo il CSSP può scrivere su questo database, dato che tutti gli altri attori saranno in grado solo di leggere;
- **Provenance Auditor:** nel ricevere una richiesta di convalida dei dati di provenienza, l'auditor raccoglierà i dati di provenienza dalla blockchain e da IPFS per convalidare la provenienza dei dati. L'auditor può convalidare, ma non collegare i dati alla vera identità degli utenti.

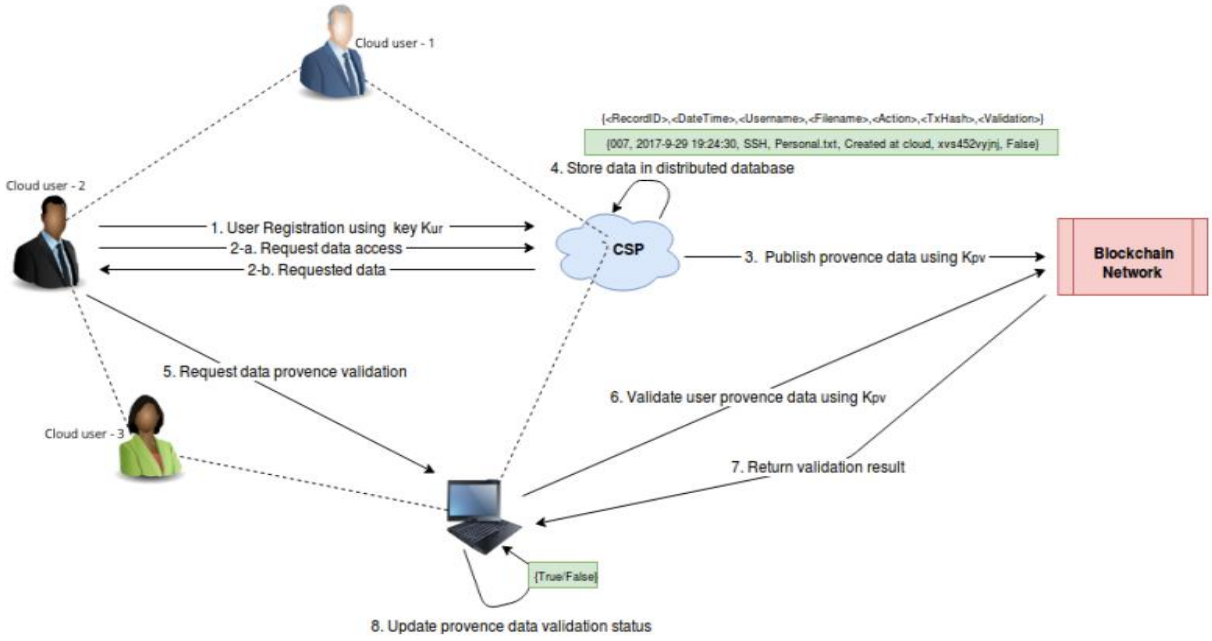


Fig. 2.1: Interazioni nel sistema [11].

2.1.3 Algoritmo proposto

Una volta che i dati di provenienza vengono raccolti, vengono archiviati nel network IPFS e nella rete blockchain; inoltre, il file di mappatura verrà aggiornato di conseguenza. L'algoritmo è mostrato nelle Fig. 2.2 (a) e (b):

Algorithm 1 Data Storage

```
// Adding provenance entry to IPFS and collecting its hash
IPFS_hash = echo<provenance_entry> | ipfs add

// Add SHA256 value of provenance entry into blockchain
Blockchain_id = add_Hash(sha256(provenance_entry))

// Update mapping.txt file
update_mappingFile(sha256(UserID), Blockchain_id, IPFS_hash, timestamp) {
  append(sha256(UserID, Blockchain_id, IPFS_hash, timestamp) -> mapping.txt file
}
```

Algorithm 2 Provenance algorithm

```
// User can request for provenance data validation by sending its mapping file
request_validation(Report)

// Provenance auditor will validate
bool validate_report(Report) {
   For all entry R in Report {
   IF { Valid(Blockchain_id) == true) {
            X = get_hash_from_Blockchain(Blockchain_id)
            Y = get_value_from_IPFS(IPFS_hash)
                     if(X != SHA256(Y))
                          return False; // Data is tampered
                     } // if
                } // for all
      return True; // All good
   }
```

Fig. 2.2 (a): Algoritmo di data storage.

Fig. 2.2 (b): Algoritmo di validazione.

• **Raccolta dei dati:** una volta che un'azione (scrittura, aggiornamento, ecc.) viene avviata sui file da parte di un utente, i dati di provenienza generati vengono raccolti nei loggers. Questi dati di provenienza raccolti vengono aggiunti a IPFS e il loro valore hash viene aggiunto alla blockchain. Questo è mostrato nell'Algoritmo 1.

• **Archiviazione dei dati di provenienza:** una volta raccolti i dati di provenienza, viene calcolato il valore SHA-256 dell'ingresso di provenienza raccolto. Si inseriscono l'ingresso di provenienza in IPFS e il valore SHA-256 nella blockchain come spiegato nell'Algoritmo 1. La rete IPFS risponderà con IPFS_id e la rete blockchain risponderà con Blockchain_id. A questo punto si archiviano la mappatura di IPFS_id e Blockchain_id insieme al timestamp nel file di mappatura che verrà archiviato presso l'utente del cloud.

• **Provenienza dei dati del cloud:** una volta che l'utente ha richiesto la convalida dei dati di provenienza, l'auditor di provenienza raccoglierà i dati di provenienza associati a quel particolare utente dal file di mappatura. Nell'algoritmo 2, vengono recuperati i dati di provenienza da IPFS e il valore SHA-256 dalla blockchain e vengono confrontati per ogni riga. Se i valori di ciascuna riga corrispondono, l'auditor di provenienza restituisce True (cioè, i dati non sono stati modificati), altrimenti restituisce False (i dati sono stati modificati).

2.1.4 Flusso del sistema

La Figura 2.3 mostra il flusso del processo. Una spiegazione dettagliata è la seguente:

- **Registrazione:** l'utente del cloud avvia il processo registrandosi presso il CSSP. Il CSSP richiederà tutte le informazioni necessarie all'utente del cloud per registrarsi al suo servizio;
- **Ottenere le chiavi:** in risposta alla registrazione, l'utente del cloud otterrà una coppia di chiavi che verranno utilizzate per crittografare e decrittare i dati durante l'archiviazione e il recupero;
- **Caricamento dei dati:** l'utente del cloud caricherà i propri files sul cloud;

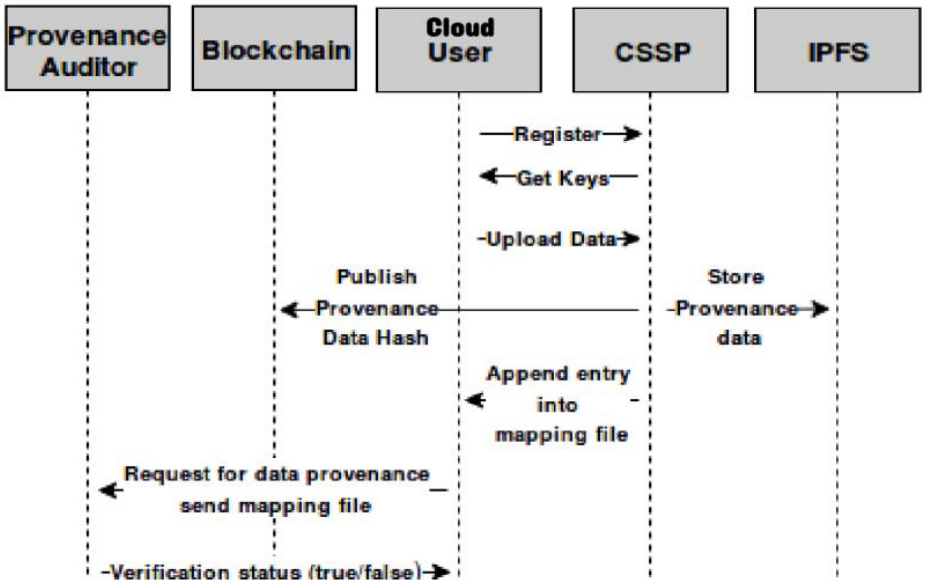


Fig. 2.3: Flusso dell'architettura [11].

- **Archiviare i dati di provenienza:** quando un utente del cloud o un avversario tenta di accedere al file archiviato, il CSSP genera i dati di provenienza del cloud che vengono archiviati nelle reti IPFS e blockchain;
- **Pubblicare i dati di provenienza:** viene calcolato il valore SHA-256 dei dati di provenienza e viene archiviato in blockchain. In risposta a questa archiviazione, la blockchain risponderà con l'identificativo BlockchainId;
- **Archiviare i dati di provenienza in IPFS:** i dati di provenienza raccolti verranno archiviati in IPFS. Tale rete risponderà con un IPFS_hash, che potrà essere utilizzato per recuperare i dati in futuro;
- **Aggiungere una voce al file di mappatura:** si aggiunge una voce contenente la mappatura tra l'identificativo BlockchainId ricevuto e l'IPFS_hash nel file di mappatura;

- **Richiesta di provenienza dei dati:** l'utente può richiedere la validità dei propri dati archiviati all'auditor di provenienza. Insieme a questa richiesta, l'utente invierà anche il proprio file di mappatura;
- **Stato di verifica:** accedendo al file di mappatura, l'auditor di provenienza richiederà i dati da IPFS e dalla blockchain e verificherà i dati archiviati in IPFS, rispondendo con la validità dei dati all'utente corrispondente.

2.2 Factom

Factom [12] fornisce al mondo un audit trail preciso, verificabile e immutabile, offrendo alle imprese l'accesso alla tecnologia blockchain. Viene utilizzata la cosiddetta "Factom Chain" per registrare i dati sulla blockchain e, in secondo luogo, viene effettuato l'ancoraggio degli hash dei dati su blockchain Bitcoin ed Ethereum. La Factom Chain è costituita da una catena di blocchi dedicata che funge da livello di base per registrare le informazioni; utilizza, inoltre, una struttura di blocchi appositamente progettata per archiviare dati hash e prove di esistenza.

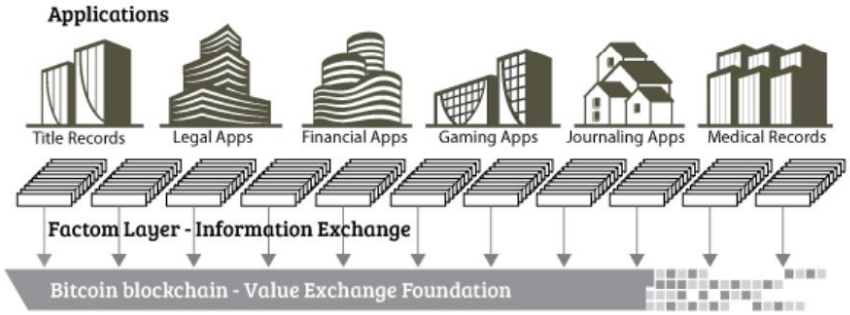


Fig. 2.4: utilizzo di applicazioni "ancorate" su blockchain Bitcoin [12].

2.2.1 Sicurezza dei dati registrati

Factom organizza i dati inseriti (Entries) utilizzando la struttura del Merkle tree: ogni voce viene hashata singolarmente e questi hash vengono combinati per creare una Merkle root. La Merkle root viene successivamente inclusa in un blocco (denominato blocco di directory), garantendo l'integrità e l'immutabilità delle voci. Factom, inoltre, impiega un gruppo di server affidabili, chiamati Factom Authority Set o server federati: questi server partecipano al processo di consenso per convalidare e garantire la sicurezza delle voci. Tali server sono responsabili della creazione e della firma dei blocchi di directory, garantendo l'accuratezza delle informazioni registrate su Factom. Factom, infine, effettua un ancoraggio periodico dell'hash del blocco di directory sulla blockchain di Bitcoin. Questo processo aggiunge un ulteriore livello di sicurezza: l'ancoraggio funge da prova crittografica che le

voci registrate su Factom in un determinato momento sono protette dalla sicurezza sottostante della blockchain di Bitcoin. Combinando la struttura del Merkle tree, il meccanismo di consenso che coinvolge i server federati e l'ancoraggio a Bitcoin, Factom assicura la sicurezza e l'integrità delle voci registrate sulla sua blockchain.

2.2.2 Panoramica del sistema

Factom è costituito da un insieme gerarchico di livelli, che possono essere raggruppati in quattro categorie:

1. Livello di Directory: organizza Merkle roots dei blocchi di Entry;
2. Livello di Entry: organizza i riferimenti ai dati inseriti;
3. Catene: raggruppamento di dati specifici per un'applicazione;
4. Voci/dati: contengono i dati grezzi di un'applicazione o l'hash dei suoi dati privati.

2.2.3 Livello di directory e organizzazione delle Merkle Roots

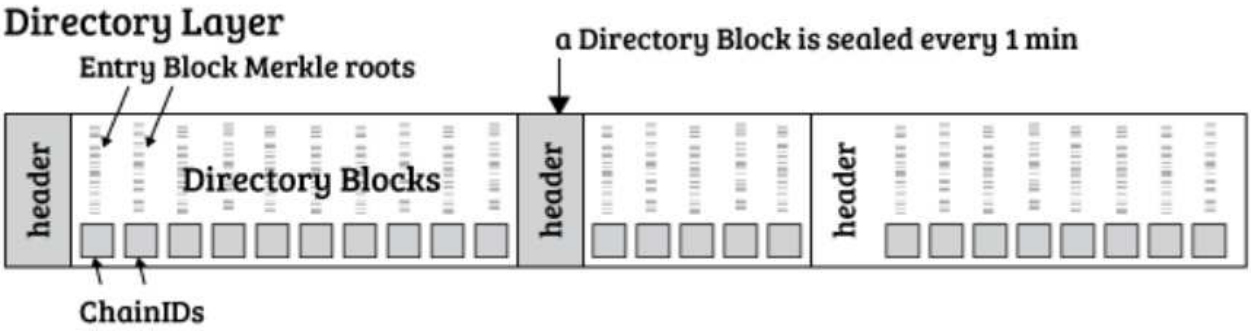


Fig. 2.5: struttura dei Directory Blocks [12].

Il livello di directory è il primo livello nella gerarchia del sistema Factom: consiste principalmente in una lista che associa un ChainID alla Merkle root del blocco di voci contenente i dati per quel ChainID (Fig. 2.5). Se un'applicazione dispone solo dei blocchi di Directory, può individuare i blocchi di voci di cui è interessata senza doverle scaricare tutte.

I server di Factom, una volta raccolte le Merkle roots e raggruppate nei blocchi di directory, registrano l'hash di tali blocchi su blockchain di Bitcoin, con un processo che viene chiamato "ancoraggio".

2.2.4 Strato di "Entry Block": organizzazione di hash e dati

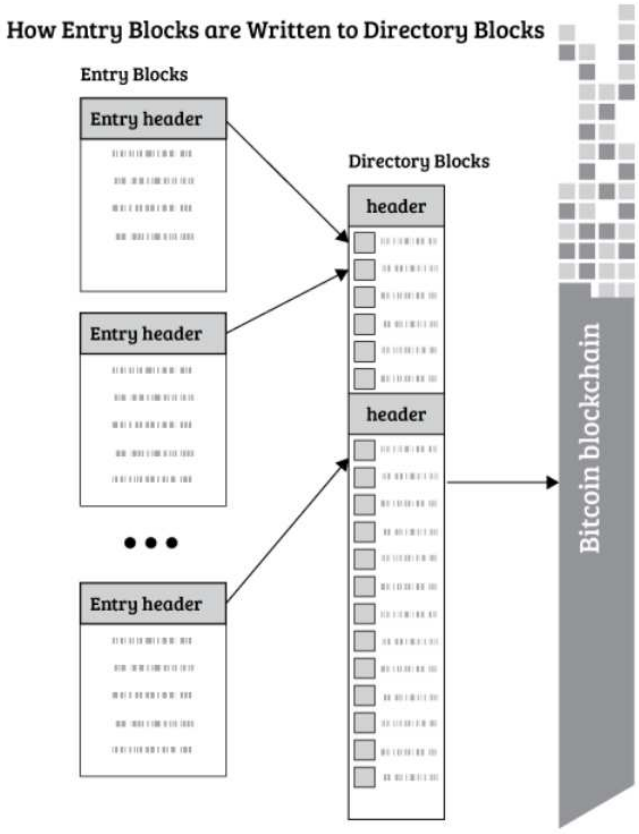


Fig. 2.6: scrittura degli Entry Blocks all'interno dei Directory Blocks [12].

Gli entry blocks sono il secondo livello di gerarchia nel sistema. Essi contengono gli hash delle singole voci, che provano l'esistenza dei dati e forniscono una chiave per trovare le voci in una rete Distributed Hash Table (DHT).

2.2.5 Entries: come vengono generate

Le voci sono create dagli utenti e inviate a Factom. È possibile garantire all'utente la privacy delle voci attraverso l'hashing o l'encoding delle informazioni, ma è anche possibile lasciare tutto in chiaro, se non è necessario codificare o oscurare i dati. Registrando l'hash di un documento, Factom può fornire una prova di pubblicazione. Presentando il documento in un momento successivo, è possibile creare il suo hash e confrontarlo con l'hash registrato in passato; c'è, inoltre, molta flessibilità sulla tipologia dei dati che possono essere caricati (Fig. 2.7): possono essere qualcosa di leggero, come un collegamento ipertestuale, o anche più grandi (ci sono comunque dei limiti, dal momento che le commissioni limitano la dimensione dei dati accettati).

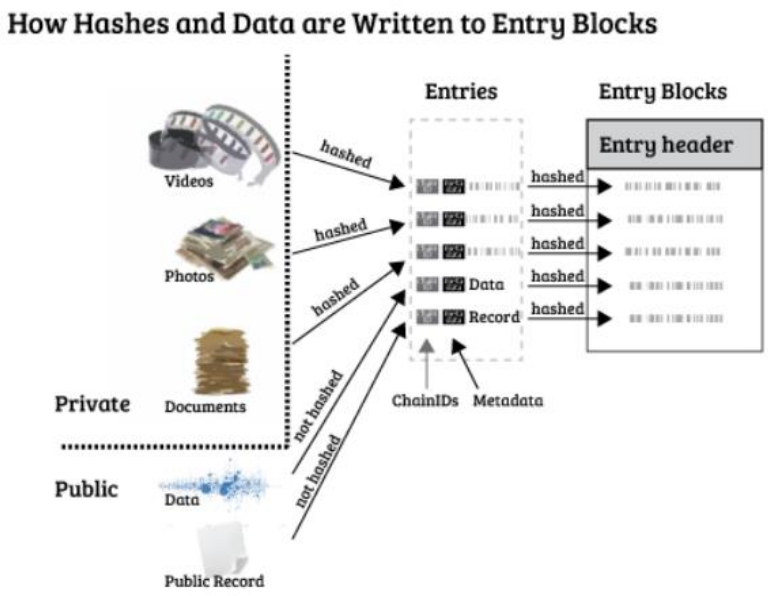


Fig. 2.7: generazione delle Entries [12].

2.2.6 Organizzazione delle Entries all'interno delle Chains

Le catene in Factom sono sequenze di voci che riflettono degli eventi rilevanti per un'applicazione. Le catene documentano queste sequenze di eventi e forniscono una traccia di audit che registra l'avvenimento di una sequenza di eventi. Le catene sono interpretazioni logiche dei dati inseriti all'interno dei blocchi di directory e dei blocchi di voci. I blocchi di directory indicano quali catene vengono aggiornate, mentre i blocchi di voci indicano quali voci sono state aggiunte alla catena.

Capitolo 3

Panoramica della piattaforma per la certificazione dei dati su blockchain

3.1 Strumenti utilizzati

La piattaforma che è stata sviluppata si basa sull'impiego di vari strumenti fondamentali: tra questi, Node.js, MongoDB e Ganache giocano un ruolo di rilievo nell'architettura e nello sviluppo del sistema, fornendo risorse essenziali per garantire sicurezza, affidabilità e gestione efficiente dei dati certificati.

3.1.1 Node.js

Node.js è un ambiente di runtime JavaScript open-source [13] che ha rivoluzionato il modo in cui vengono sviluppate le applicazioni web. Fondamentalmente, consente l'esecuzione di codice JavaScript lato server, offrendo una piattaforma potente e versatile per la creazione di applicazioni scalabili e performanti. La sua caratteristica distintiva risiede nella capacità di utilizzare JavaScript sia sul lato client che sul lato server, consentendo agli sviluppatori di scrivere codice in un unico linguaggio su entrambi i lati dell'applicazione. Questa unificazione semplifica lo sviluppo e la manutenzione delle applicazioni, migliorando l'efficienza del processo di sviluppo.

Node.js è noto per il suo modello di programmazione asincrona, che consente alle operazioni I/O di essere eseguite in modo non bloccante. Ciò significa che Node.js può gestire molteplici richieste simultaneamente, rendendo l'applicazione altamente scalabile e reattiva. Grazie alla sua architettura event-driven, Node.js è particolarmente adatto per applicazioni che richiedono una gestione rapida e distribuita dei dati, come le applicazioni in tempo reale, le applicazioni di streaming o i servizi web ad alte prestazioni; inoltre, Node.js dispone di un vasto ecosistema di librerie e moduli che ampliano le sue funzionalità di base. Il gestore dei pacchetti Node.js, npm (Node Package Manager), consente agli sviluppatori di accedere a una vasta gamma di moduli che possono essere facilmente integrati nelle loro applicazioni, accelerando così lo sviluppo e migliorando l'efficienza.

3.1.2 MongoDB

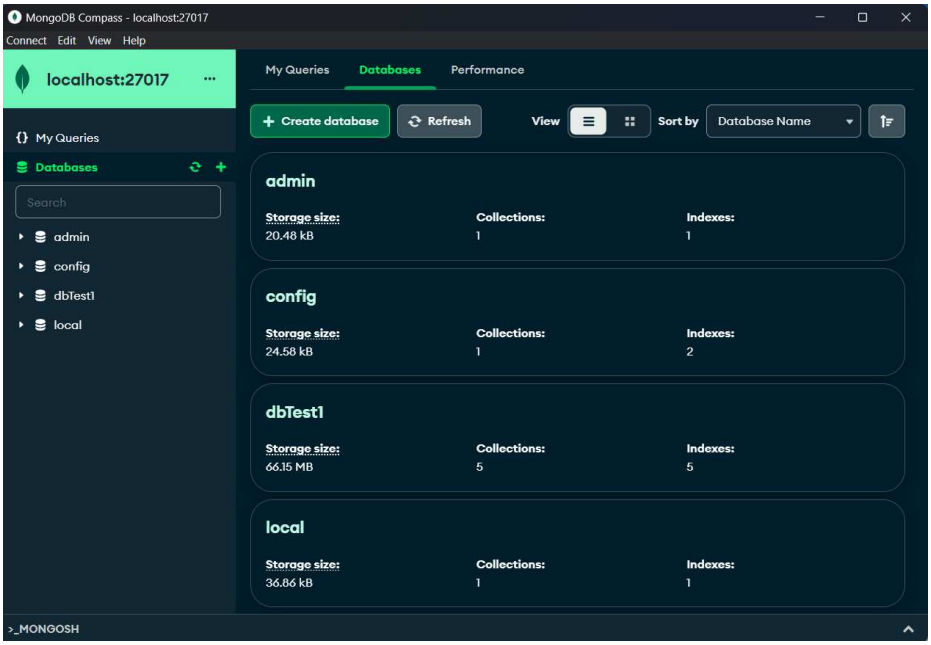


Fig. 3.2: Interfaccia grafica di MongoDB Compass.

MongoDB è un database NoSQL flessibile e scalabile che fa uso di un modello di dati basato su documenti [14]. Questo sistema di gestione dei dati è progettato per gestire volumi enormi di dati non strutturati o semi-strutturati, offrendo una soluzione dinamica e adattabile alle esigenze delle moderne applicazioni. La caratteristica principale di MongoDB è la sua struttura di archiviazione basata su documenti BSON (Binary JSON), simile a JSON, che consente una modellazione dei dati flessibile e senza schema rigido. Questo significa che i dati possono essere memorizzati in documenti che possono variare nella struttura e contenere campi diversi, consentendo una maggiore agilità nello sviluppo e nell'adattamento delle applicazioni alle mutevoli esigenze aziendali; inoltre, MongoDB supporta una vasta gamma di tipi di dati, inclusi array, oggetti nidificati e tipi di dati geospaziali, consentendo una gestione sofisticata dei dati.

Per interagire con i database MongoDB si utilizza MongoDB Compass, un'interfaccia grafica utente (Fig. 3.2) intuitiva e potente che funge da strumento di gestione e analisi dei dati, offrendo agli sviluppatori e agli amministratori di database una visione chiara e interattiva del loro ambiente di sviluppo. MongoDB Compass semplifica notevolmente l'esplorazione, l'interrogazione e la manipolazione dei dati: consente agli utenti di visualizzare e interagire con i dati tramite una varietà di strumenti, come query builder, aggregazioni visive e visualizzazioni grafiche dei dati. Questo strumento è prezioso per esplorare e comprendere la struttura dei dati all'interno del database, facilitando l'analisi e l'ottimizzazione delle query; inoltre, MongoDB Compass offre funzionalità avanzate come l'indicizzazione dei dati, la gestione degli schemi e la creazione di query complesse con facilità.

3.1.3 Ganache

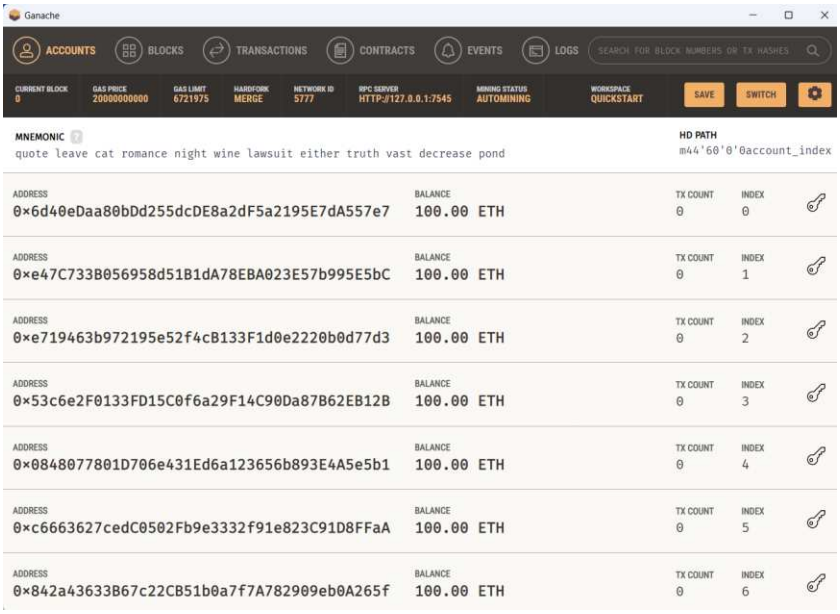


Fig. 3.3: Interfaccia grafica di Ganache.

Ganache è uno strumento fondamentale per la blockchain Ethereum, dato che fornisce un ambiente di sviluppo locale che consente agli sviluppatori di creare, testare e interagire con applicazioni blockchain senza la necessità di connettersi alla rete Ethereum principale e, di conseguenza, senza preoccuparsi dei costi o dei rischi associati all'interazione con la rete Ethereum live [15].

Ganache offre una serie di funzionalità utili per il testing e lo sviluppo delle applicazioni blockchain: consente di creare account, generare transazioni, deployare smart contract e simulare il comportamento della blockchain in modo controllato. Gli account generati da Ganache dispongono di Ether fittizio (ETH) per simulare le transazioni e le interazioni con la blockchain, consentendo agli sviluppatori di testare le proprie applicazioni senza utilizzare valuta reale; inoltre, Ganache offre un'interfaccia utente intuitiva che permette agli sviluppatori di esplorare e comprendere il comportamento della blockchain in modo visivo e dettagliato. La dashboard fornisce informazioni sullo stato della blockchain, sui blocchi, sulle transazioni, sugli account e sugli eventi generati, facilitando l'analisi e il debug delle applicazioni.

Un altro aspetto fondamentale di Ganache è la sua flessibilità: offre, infatti, varie opzioni di configurazione, tra cui la regolazione della velocità della blockchain, la simulazione di reti private, la gestione degli account e altro ancora.

3.2 Data Collection

Il problema della data collection si suddivide in due sotto-problemi:

1. Progettazione del **Front-end**: è la parte visibile e interattiva di un'applicazione web con cui gli utenti interagiscono direttamente. Include l'interfaccia utente, il design, la disposizione degli elementi visivi e tutti gli elementi con cui gli utenti possono interagire direttamente. Le tecnologie front-end comunemente usate includono HTML, CSS e JavaScript;
2. Progettazione del **Back-end**: è la parte non visibile di un'applicazione web che gestisce le funzionalità dietro le quinte. Si occupa dell'elaborazione dei dati, della logica dell'applicazione, dell'accesso al database e dell'invio delle informazioni al front-end. Utilizza linguaggi di programmazione come Python, Ruby, PHP, Java, C# e framework come Node.js, Django e Ruby on Rails.

3.2.1 Sviluppo del Front-End

Il front-end è stato scritto principalmente in HTML, con aggiunte di CSS; inoltre, include anche l'utilizzo di Bootstrap, framework CSS che fornisce una serie di strumenti e componenti predefiniti per la creazione di interfacce web responsive e stilisticamente gradevoli.

L'interfaccia è stata sviluppata in modo molto minimale, solo a titolo di esempio, per mostrare quello che è il principio di funzionamento della piattaforma (Fig. 3.4). Il codice è riportato in Fig. 3.5:

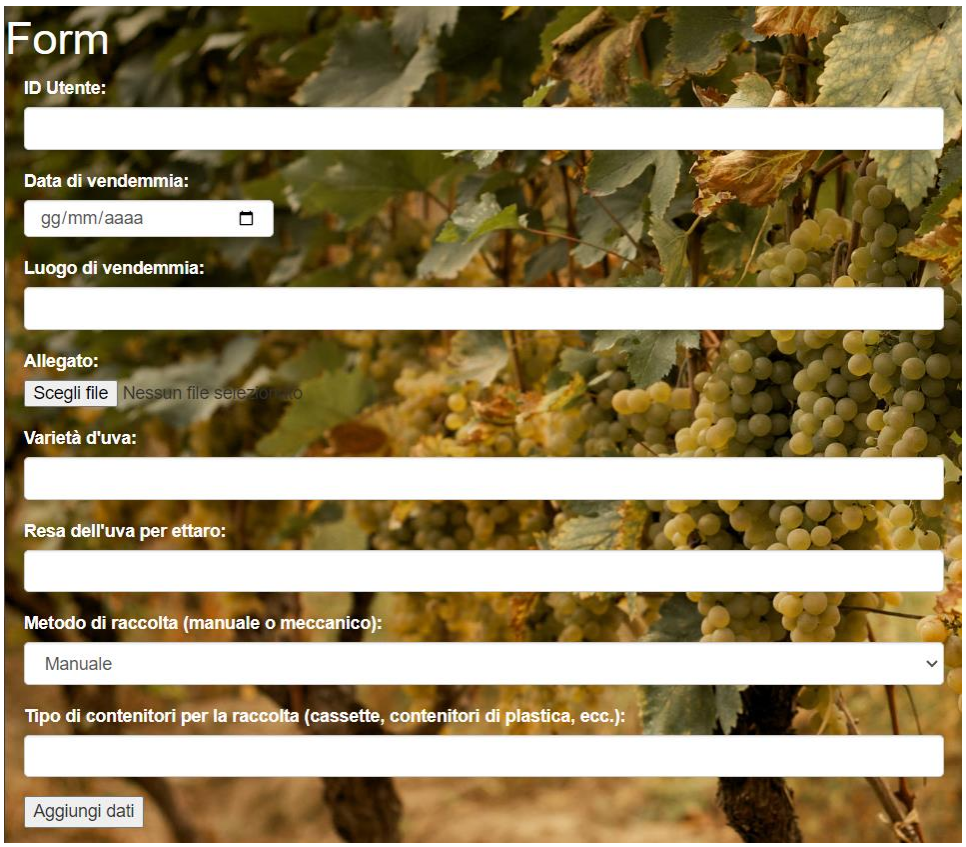


Fig. 3.4: Interfaccia per il caricamento dei dati.

```html
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Form</title>
    <style>
        /* Stile per il corpo della pagina con sfondo */
        body {
            background-image: url("uva-da-tavola.jpg");
            background-size: cover;
            background-repeat: no-repeat;
            background-attachment: fixed;
            color: white; /* Imposta il colore del testo a bianco */
        }

        /* Imposta il colore del titolo */
        body h1 {
            color: white;
        }

        /* Imposta larghezza e altezza delle caselle di testo e menu a discesa */
        input[type="text"],
        input[type="date"],
        select {
            width: 200px; /* Imposta la larghezza delle caselle di testo e menu a discesa */
            height: 30px; /* Imposta l'altezza delle caselle di testo e menu a discesa */
            font-size: 14px; /* Imposta la dimensione del testo delle caselle di testo e menu a discesa */
        }

        /* Imposta il colore del label e degli input */
        label,
        input,
        select {
            color: white;
        }
    </style>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
</head>

<body>
    <h1>Form</h1>
    <form class="container" method="post" action="/" enctype="multipart/form-data">
        <div class="form-group">
            <label for="idUtente">ID Utente:</label>
            <input class="form-control" name="idUtente" id="idUtente">
        </div>
        <div class="form-group">
            <label for="dataVendemmia">Data di vendemmia:</label>
            <input class="form-control" name="dataVendemmia" id="dataVendemmia" type="date">
        </div>
        <div class="form-group">
            <label for="luogoVendemmia">Luogo di vendemmia:</label>
            <input class="form-control" name="luogoVendemmia" id="luogoVendemmia">
        </div>
        <div class="form-group">
            <label for="allegato">Allegato:</label>
            <input type="file" class="form-control-file" name="allegato" id="allegato">
        </div>
        <div class="form-group">
            <label for="varietàUva">Varietà d'uva:</label>
            <input class="form-control" name="varietàUva" id="varietàUva">
        </div>
        <div class="form-group">
            <label for="resaUvaEttaro">Resa dell'uva per ettaro:</label>
            <input class="form-control" name="resaUvaEttaro" id="resaUvaEttaro">
        </div>
        <div class="form-group">
            <label for="metodoRaccolta">Metodo di raccolta (manuale o meccanico):</label>
            <select class="form-control" name="metodoRaccolta" id="metodoRaccolta">
                <option value="manuale">Manuale</option>
                <option value="meccanico">Meccanico</option>
            </select>
        </div>
        <div class="form-group">
            <label for="tipoContenitoriRaccolta">Tipo di contenitori per la raccolta (cassette, contenitori di plastica, ecc.):</label>
            <input class="form-control" name="tipoContenitoriRaccolta" id="tipoContenitoriRaccolta">
        </div>
        <button>Aggiungi dati</button>
    </form>
</body>
</html>
```

Fig. 3.5: Codice HTML e CSS per l'implementazione del Front-end.

Questo codice HTML definisce una pagina web che contiene un formulario per raccogliere informazioni sulla vendemmia, tra cui ID utente, data di vendemmia, luogo, varietà d'uva, resa, metodo di raccolta e tipo di contenitori. Quando il modulo viene inviato, i dati verranno inviati al server Node.js per l'elaborazione tramite il metodo POST.

3.2.2 Sviluppo del Back-End

Il codice che segue è un'applicazione Node.js che si occupa di gestire le informazioni che vengono raccolte tramite form HTML, attraverso un server web. Ecco una panoramica dei passaggi principali:

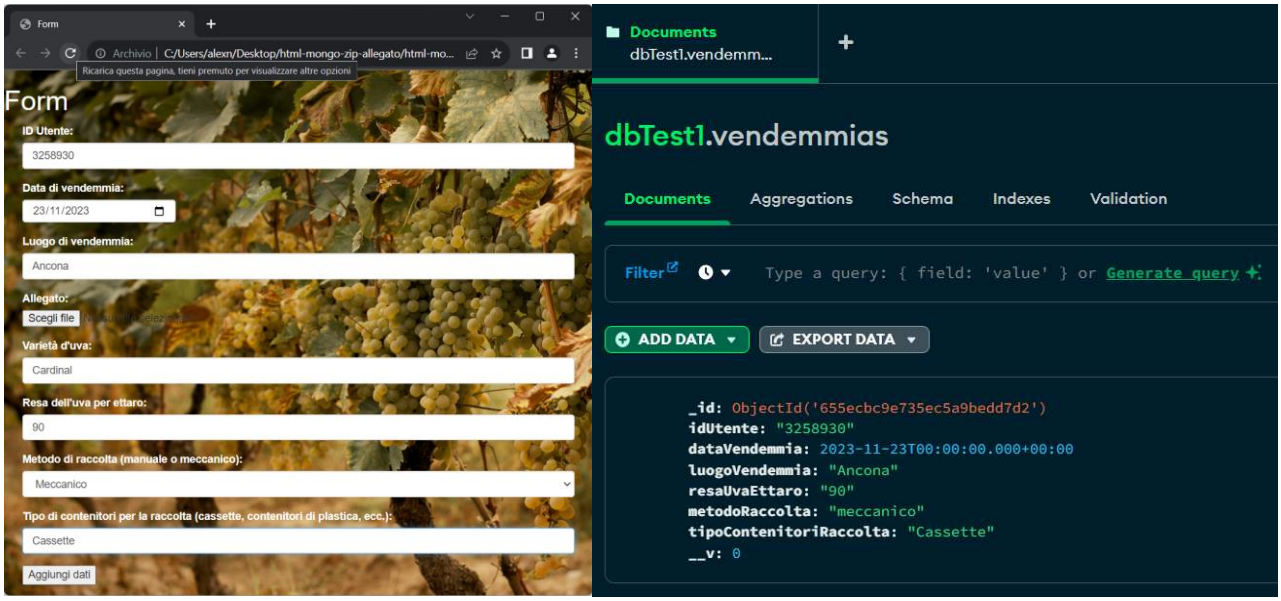
- vengono importate le librerie necessarie come Express per la gestione del server, Mongoose per comunicare con il database MongoDB, Multer per la gestione degli allegati e altre librerie per la manipolazione dei file;
- viene creato un server Express e specificata la porta su cui il server ascolta le richieste;
- Multer viene configurato per gestire gli upload di file, specificando la destinazione dei file caricati;
- viene stabilita la connessione al database MongoDB utilizzando Mongoose e l'URL di connessione specificato;
- viene creato uno schema per modellare i dati che arrivano dal form HTML, specificando i vari campi come idUtente, data, luogo, ecc.;
- quando viene effettuata una richiesta GET alla radice del server, viene restituito un file HTML;
- se viene inviata una richiesta POST alla radice del server (il modulo HTML), i dati raccolti vengono estratti dalla richiesta e salvati come un nuovo oggetto;
- se è presente un allegato nel form, questo viene letto, compresso in ZIP utilizzando la libreria archiver, il percorso del file ZIP viene salvato nell'oggetto e il file temporaneo viene eliminato;
- l'oggetto, con o senza allegato, viene salvato nel database MongoDB;
- viene configurata una route per rendere accessibili i file caricati nella cartella "uploads" attraverso il percorso "/uploads";
- il server Express viene avviato, in modo che possa iniziare ad ascoltare le richieste sulla porta specificata.

In Fig. 3.6 è riportato il codice:

```
const express = require('express');
const mongoose = require('mongoose');
const multer = require('multer');
const path = require('path');
const app = express();
const port = 3000;
const fs = require('fs');
const archiver = require('archiver'); // Importa la libreria per la compressione

// Connetti al database MongoDB
const url = 'mongodb://0.0.0.0/dbTest1';

// Configura multer per gestire gli allegati
const storage = multer.diskStorage({
    destination: path.join(__dirname, 'uploads'),
    filename: function (req, file, cb) {
        cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
    }
});

const upload = multer({ storage: storage });

const vendemmiaSchema = {
    idUtente: String,
    dataVendemmia: Date,
    luogoVendemmia: String,
    varietàUva: String,
    resaUvaEttaro: String,
    metodoRaccolta: String,
    tipoContenitoriRaccolta: String,
    allegato: String
};

const Vendemmia = mongoose.model("Vendemmia", vendemmiaSchema);

app.use(express.json());

mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true });

app.get("/", function (req, res) {
    res.sendFile(__dirname + "/index.html");
});

app.post("/", upload.single('allegato'), function (req, res) {
    const newVendemmia = new Vendemmia({
        idUtente: req.body.idUtente,
        dataVendemmia: req.body.dataVendemmia,
        luogoVendemmia: req.body.luogoVendemmia,
        luogoVendemmia: req.body.luogoVendemmia,
        varietàUva: req.body.varietàUva,
        resaUvaEttaro: req.body.resaUvaEttaro,
        metodoRaccolta: req.body.metodoRaccolta,
        tipoContenitoriRaccolta: req.body.tipoContenitoriRaccolta,
    });

    if (req.file) {
        // Leggi il contenuto del file allegato
        const allegatoData = fs.readFileSync(req.file.path);

        // Crea un file compresso ZIP
        const archive = archiver('zip', {
            zlib: { level: 9 } // Imposta il livello di compressione (opzionale)
        });
        const archivePath = path.join(__dirname, 'uploads', 'allegato.zip');
        const output = fs.createWriteStream(archivePath);
        archive.pipe(output);

        // Aggiungi il file allegato compresso al file ZIP
        archive.append(allegatoData, { name: req.file.originalname });

        // Finalizza il file ZIP
        archive.finalize();

        newVendemmia.allegato = archivePath; // Salva il percorso del file ZIP nell'oggetto

        // Elimina il file temporaneo dell'allegato
        fs.unlinkSync(req.file.path);
    }

    newVendemmia.save()
        .then(savedData => {
            res.json(savedData); // Invia una risposta JSON con i dati salvati
        })
        .catch((error) => {
            console.error('Errore nell\'inserimento dei dati in MongoDB:', error);
            res.status(500).json({ error: 'Errore nell\'inserimento dei dati' });
        });
});

app.use('/uploads', express.static('uploads'));

app.listen(port, function () {
    console.log(`Server in ascolto sulla porta ${port}`);
});
```

Fig. 3.6: Codice per la gestione delle informazioni raccolte.

È possibile fare un test per verificare se funziona tutto correttamente, andando a compilare il form HTML (Fig. 3.7(a)) e verificare, dopo aver proceduto con l'invio, se tale dato è stato aggiunto all'interno del database MongoDB (Fig. 3.7(b)):



(a) (b)

Fig. 3.7: (a) Compilazione del form; (b) Documento all'interno di MongoDB.

3.3 Costruzione del Merkle Tree per il Data Certification

Il vantaggio che deriva dall'impiego del Merkle Tree nel contesto di questo progetto è quello di limitare il numero di transazioni su blockchain che sono necessarie per certificare i dati di interesse: grazie a questa soluzione, si pubblica su Ethereum, in modo periodico, soltanto la Merkle Root che fa riferimento ad un albero contenente una quantità variabile di dati (l'alternativa sarebbe, invece, quella di calcolare gli hash dei dati singolarmente ed effettuare tante transazioni su blockchain quanti sono i dati da certificare). Ovvio che, nel caso di adozione del Merkle Tree, sarà necessario più spazio sul database per memorizzare, oltre ai dati, anche i Merkle Trees stessi (è già stato anticipato il motivo per cui si è preferito non salvare direttamente le proof dei dati): si rende, pertanto, necessario effettuare delle opportune valutazioni, dipendenti da quelle che sono le specifiche di progetto, riguardo quella che rappresenta la soluzione più adeguata al caso specifico e quali sono i parametri da impostare (con quale frequenza inviare transazioni su Ethereum, quanti elementi inserire all'interno del Merkle Tree,…). Tutte queste considerazioni sono presenti in [16], all'interno del quale è presente il codice che implementa la generazione del Merkle Tree, il salvataggio dello stesso su database (Fig. 3.8), la ricerca dell'indice (Fig. 3.9) e la generazione della proof (Fig. 3.10). Tale codice viene riportato di seguito:

```
// Costruzione dell'albero di Merkle e calcolo della radice
function buildMerkleTree(hashedData) {
    if (hashedData.length === 0) {
        return [null, null];  // Se la lista    vuota, restituisci [null, null]
    }

    if (hashedData.length === 1) {
        return [hashedData[0], hashedData[0]];  // La radice    anche il dato singolo
    }

    const tree = [hashedData];

    while (tree[tree.length - 1].length > 1) {
        const level = [];
        for (let i = 0; i < tree[tree.length - 1].length; i += 2) {
            const left = tree[tree.length - 1][i];
            const right = i + 1 < tree[tree.length - 1].length ? tree[tree.length - 1][i + 1] : tree[tree.length - 1][i];
            level.push(crypto.createHash('sha256').update(left + right).digest('hex'));
        }
        tree.push(level);
    }

    return [tree, tree[tree.length - 1][0]];
}

// Funzione per salvare l'intero albero di Merkle nel database
async function saveMerkleTreeToDatabase(db, merkleTree) {
    try {
        const collection = db.collection('MerkleTree');

        // Sovrascrivi con il nuovo albero di Merkle
        await collection.replaceOne({}, {
            merkleTree: merkleTree
        }, {
            upsert: true // Crea il documento se non esiste
        });

        console.log('Albero di Merkle salvato nel database con successo.');
    } catch (error) {
        console.error('Errore durante il salvataggio dell\'albero di Merkle nel database:', error);
    }
}
```

Fig. 3.8: Costruzione e salvataggio di un Merkle Tree [15].

```
    // Cerca l'hash del documento nell'albero di Merkle
    const dataPosition = merkleTree[0][0].indexOf(calculatedHash);
```

Fig. 3.9: Ricerca dell'indice del dato di interesse [15].

```
// Calcola la prova di Merkle
function getMerkleProof(tree, index) {
    const proof = [];
    let levelIndex = index;

    for (const level of tree) {
        if (levelIndex % 2 === 0 && levelIndex + 1 < level.length) {
            const siblingIndex = levelIndex + 1;
            proof.push('R' + level[siblingIndex]);
        } else if (levelIndex % 2 === 1 && levelIndex - 1 >= 0) {
            const siblingIndex = levelIndex - 1;
            proof.push('L' + level[siblingIndex]);
        }

        levelIndex = Math.floor(levelIndex / 2);
    }

    return proof;
}
```

Fig. 3.10: Generazione della proof [15].

Riguardo la generazione della proof, gli step che vengono seguiti sono i seguenti:

1. **Funzione getMerkleProof(tree, index):**
    - Questa funzione prende in input il Merkle Tree (tree) precedentemente salvato su database e l'indice del dato di cui si desidera ottenere la prova (index).
2. **Inizializzazione delle variabili:**
    - **proof**: è un array vuoto in cui verranno inserite le proof (hash) man mano che vengono calcolate.
    - **levelIndex**: è l'indice del dato all'interno del Merkle Tree e viene inizializzato con l'indice fornito come input alla funzione.
3. **Iterazione attraverso i livelli dell'albero:**
    - Il codice utilizza un ciclo for...of per attraversare i vari livelli (level) del Merkle Tree.
4. **Calcolo della proof di Merkle:**
    - La logica principale si trova all'interno di questo ciclo. Il codice verifica se l'index del dato è pari o dispari:
        - Se **index** è pari (levelIndex % 2 === 0) e c'è un "sibling" (un nodo fratello) disponibile nel livello, aggiunge l'hash del nodo fratello segnato come "R" (destro) nella proo.

- Se **index** è dispari (levelIndex % 2 === 1) e c'è un nodo fratello disponibile, aggiunge l'hash del nodo fratello segnato come "L" (sinistro) nella proof.
- In entrambi i casi, l'hash del nodo fratello viene aggiunto all'array proof.

5. **Aggiornamento dell'indice del livello:**
    - Dopo ogni iterazione, levelIndex viene aggiornato dividendo per 2, ottenendo così l'indice del livello superiore nel Merkle Tree (Math.floor(levelIndex / 2**)**).
6. **Ritorno della proof di Merkle:**
    - Una volta terminato il ciclo attraverso i livelli dell'albero, la funzione restituisce l'array proof, che contiene la proof di Merkle per il dato specifico all'interno del Merkle Tree.

È possibile evidenziare che, per generare la proof di un dato specifico, è necessario conoscere la posizione che tale dato occupa all'interno del Merkle Tree. In questo caso specifico, per ottenere l'indice del dato, si ricorre al metodo .indexOf(), un metodo di JavaScript che viene utilizzato per trovare la posizione di un elemento all'interno di un array e restituirne l'indice corrispondente. Tale metodo esegue, pertanto, un confronto tra ogni singolo elemento dell'array e il valore che gli viene fornito come input per cercare la sua presenza all'interno dell'array, restituendo l'indice della prima occorrenza dell'elemento corrispondente. In uno scenario in cui si utilizzano Merkle Trees con molti elementi e si ricevono richieste di certificazione con elevata frequenza, andare a confrontare l'hash di interesse con tutti quelli inseriti all'interno del Merkle Tree può risultare poco efficiente, in termini di tempo richiesto dal processo di verifica. Una soluzione per risolvere questo problema può essere quella di ricorrere all'uso di un Patricia Trie, associato al Merkle Tree di partenza, all'interno del quale andare a memorizzare le posizioni che i dati occupano all'interno del Merkle Tree stesso. Uno studio dettagliato di questa soluzione sarà mostrato nel prossimo capitolo.

Capitolo 4

Ottimizzare l'indicizzazione: Patricia Tries come supporto all'utilizzo dei Merkle Trees

4.1 Definizione della classe PatriciaTrie in Javascript

Un esempio di codice che definisce la struttura di un Patricia Trie è mostrato in Fig. 4.1:

```javascript
const crypto = require('crypto');

class PatriciaTrie {
    constructor() {
        this.root = {};
    }

    add(transaction) {
        let temporaryRoot = this.root;
        let str = transaction.hash;

        for (let i = 0; i < str.length; i++) {
            let character = str[i];
            if (temporaryRoot[character] === undefined) {
                temporaryRoot[character] = {};
            }
            temporaryRoot = temporaryRoot[character];
        }

        // Creazione di un nuovo oggetto che contiene la posizione del dato all'interno di data
        const position = transaction.data;
        temporaryRoot["DATA"] = { position }; // Associare la posizione come valore

        // Aggiunta del dato completo se necessario
        //temporaryRoot["DATA"]["fullData"] = transaction.data;
    }

    get(hash) {
        let temporaryRoot = this.root;

        for (let index = 0; index < hash.length; index++) {
            if (temporaryRoot) temporaryRoot = temporaryRoot[hash[index]];
            else return null;
        }

        if (temporaryRoot && temporaryRoot["DATA"]) {
            return temporaryRoot["DATA"];
        } else {
            return null;
        }
    }

    remove(hash) {
        let temporaryRoot = this.root;

        for (let index = 0; index < hash.length; index++) {
            if (temporaryRoot) temporaryRoot = temporaryRoot[hash[index]];
            else return false;
        }

        if (temporaryRoot && temporaryRoot["DATA"]) {
            delete temporaryRoot["DATA"];
            return true;
        } else {
            return false;
        }
    }
```

Fig. 4.1: Patricia Trie in Javascript.

Il primo metodo che viene definito è il metodo 'constructor', che inizializza la struttura Trie con un nodo radice vuoto; il secondo metodo è denominato 'add' ed esegue i seguenti steps:

- aggiunge una transazione alla struttura Trie;
- itera la stringa hash della transazione;
- per ogni carattere della stringa hash, se il percorso non esiste nel Trie, crea un nuovo nodo per rappresentare quel carattere;
- aggiunge un'associazione temporaryRoot["DATA"], contenente la posizione della transazione nella struttura dati.

Il terzo metodo è quello di 'get(hash)' ed è quello che verrà utilizzato più avanti per ricavare la posizione di un dato all'interno del Merkle Tree a partire dal suo hash. Questo metodo:

- recupera una transazione (dunque, un dato, ovvero il valore registrato) dal Trie, dato l'hash;
- segue il percorso corrispondente all'hash all'interno del Trie;
- se trova un nodo terminale che contiene la chiave "DATA", restituisce la posizione della transazione.

L'ultimo metodo è quello 'remove(hash)' e procede alla rimozione di un dato dal Trie, dato l'hash.
È possibile definire molti altri metodi all'interno della classe Patricia Trie (ad esempio, per visitare e raccogliere tutti i dati contenuti nel Patricia Trie, per effettuare una stampa su console della struttura dell'albero o anche per calcolare una stima della dimensione richiesta dall'albero).

4.2 Implementazione di un Patricia Trie per la memorizzazione degli indici del Merkle Tree

In questo paragrafo verranno messi a confronto due scenari:

- nel primo, per certificare il dato e calcolare la proof, si farà uso del solo Merkle Tree, così come visto nel Capitolo 3. Verranno misurati i tempi di generazione degli alberi, della ricerca dell'indice e del calcolo della proof al variare del numero di elementi inseriti nel Merkle Tree ed eseguendo 10000 iterazioni (per ciascuna iterazione, si varia in modo random il dato del quale si vuole ottenere la proof);
- nel secondo, si ricorre all'uso di un Patricia Trie per salvare gli indici inerenti alle posizioni dei dati all'interno del Merkle Tree; così come per il primo scenario, si valuteranno le prestazioni al variare del numero di elementi inseriti nel Merkle Tree, eseguendo 10000 iterazioni per il calcolo di media e deviazione standard dei tempi.

Inoltre, sarà svolta una valutazione dello spazio aggiuntivo richiesto per il salvataggio del Patricia Trie all'interno del database.

4.2.1 Calcolo della proof nel caso di utilizzo di Merkle Tree: tempi e spazio richiesti

Il codice in Fig. 4.2 (a) e (b) si occupa di valutare le prestazioni del Merkle Tree, in termini di tempi necessari per la generazione dell'albero, per la ricerca dell'indice e per la restituzione della proof; viene, inoltre, calcolato lo spazio richiesto per la memorizzazione dell'albero all'interno del database:

Desktop > JS MerkleTreeTest.js > ...

```
1  const { StandardMerkleTree } = require("@openzeppelin/merkle-tree");
2  const fs = require("fs");
3  const { performance } = require('perf_hooks');
4
5  const iterations = 10000; // Numero di iterazioni da eseguire
6  const dataSize = 1000; // Numero di dati da inserire nel Merkle Tree
7
8  const fixedData = [];
9
10 for (let i = 0; i < 1000; i++) { // Dati da inserire nel Merkle, sono gli stessi anche
11   const address = `0x${i.toString().padStart(40, '0')}`; // Esempio: 0x0000000000000000
12   const value = `${i * 100000000000000000}`; // Esempio: 0, 100000000000000000, 200000
13   fixedData.push([address, value]);
14 }
15
16 const times = { // Necessario per calcolo media e dev. standard dei tempi
17   treeCreation: [],
18   findIndex: [],
19   obtainProof: [],
20 };
21
22 for (let i = 0; i < iterations; i++) {
23   const values = fixedData.slice(0, dataSize);
24
25   // Misura il tempo iniziale
26   const startTime = performance.now();
27
28   // Genera il Merkle Tree
29   const tree = StandardMerkleTree.of(values, ["address", "uint256"]);
30   const treeCreationTime = performance.now();
31   times.treeCreation.push(treeCreationTime - startTime);
32   console.log('Tempo per la creazione del Merkle Tree', treeCreationTime - startTime);
33   //console.log(tree.render());
34
35   // Salva l'albero in un file JSON dopo la creazione
36   //fs.writeFileSync(`tree_${i}.json`, JSON.stringify(tree.dump()));
```

Fig. 4.2 (a): generazione e salvataggio del Merkle Tree.

```
  // Scegli il dato di interesse in modo casuale
  const randomIndex = Math.floor(Math.random() * dataSize);
  const dataOfInterest = fixedData[randomIndex][0];
  //console.log('dataofinterest', dataOfInterest);

  let index = -1;
  const indexOfInterestStartTime = performance.now();
  for (const [i, v] of tree.entries()) {
    if (v[0] === dataOfInterest) {
      index = i; // Indice trovato
      //console.log('indice',i);
      break;
    }
  }
  const indexOfInterestEndTime = performance.now();
  times.findIndex.push(indexOfInterestEndTime - indexOfInterestStartTime);
  console.log('Tempo per trovare indice', indexOfInterestEndTime - indexOfInterestStartTime);

  if (index !== -1) {
    const proofObtainStartTime = performance.now();
    // Ottieni la proof utilizzando la posizione determinata dall'indice i
    const proof = tree.getProof(index);
    //console.log('proof:',proof);
    const proofObtainEndTime = performance.now();
    times.obtainProof.push(proofObtainEndTime - proofObtainStartTime);
    console.log('Tempo per ottenere la Proof',proofObtainEndTime - proofObtainStartTime);
  }
}

function calculateStatistics(timesArray) { // Calcolo medie e dev. standard tempi
  const sum = timesArray.reduce((acc, cur) => acc + cur, 0);
  const average = sum / timesArray.length;

  const squaredDifferences = timesArray.map(time => Math.pow(time - average, 2));
  const variance = squaredDifferences.reduce((acc, cur) => acc + cur, 0) / timesArray.length;

  const standardDeviation = Math.sqrt(variance);

  return { average, standardDeviation };
}

console.log("Tempi per la creazione del Merkle tree:");
console.log(calculateStatistics(times.treeCreation));

console.log("Tempi per trovare l'indice del dato di interesse:");
console.log(calculateStatistics(times.findIndex));

console.log("Tempi per ottenere la proof:");
console.log(calculateStatistics(times.obtainProof));
```

Fig. 4.2 (b): ricerca dell'indice e restituzione della proof.

Gli step eseguiti sono, nell'ordine:

- **Inizializzazione dei parametri:**
  - *iterations* rappresenta il numero di iterazioni da eseguire;
  - *dataSize* indica il numero di dati da inserire nel Merkle Tree;
  - *fixedData* è un array vuoto che verrà riempito con coppie di dati (indirizzo, valore);

- **Generazione dei dati:**
  - il codice genera un insieme di dati di esempio (fixedData) da utilizzare per popolare il Merkle Tree. Questi dati consistono in coppie di indirizzi e valori associati a scopo dimostrativo.
- **Misurazione dei tempi:**
  - il codice esegue un ciclo, che viene ripetuto 10000 volte;
  - per ciascuna iterazione, si seleziona un sottoinsieme di dati del dataset creato in precedenza;
  - si misura il tempo richiesto per creare un Merkle Tree utilizzando tale sottoinsieme;
  - si seleziona, in modo casuale, un dato di interesse dal sottoinsieme e viene calcolato il tempo necessario per trovare l'indice di questo dato all'interno del Merkle Tree;
  - a questo punto, si calcola il tempo necessario per ottenere la proof relativa a quel dato.
- **Statistiche sui tempi:**
  - è definita una funzione, *calculateStatistics()*, che riceve in ingresso un array di tempi e calcola media e deviazione standard dei tempi registrati nella fase precedente;
- **Stampa dei tempi:**
  - le statistiche sui tempi vengono stampate su console;
- **Calcolo della dimensione del Merkle Tree:**
  - all'occorrenza, è possibile stabilire le dimensioni richieste per l'archiviazione del Merkle Tree salvandolo in locale.

4.2.2 Calcolo della proof nel caso di uso congiunto di Merkle Tree e Patricia Trie: tempi e spazio richiesti

Il codice riportato in Fig. 4.3 (a) e (b) si occupa di valutare le prestazioni del Merkle Tree, usato congiuntamente al Patricia Trie, secondo le modalità precedentemente mostrate. Le metriche utilizzate sono le stesse già viste nello scenario precedente, ovvero: tempi necessari per la generazione dell'albero, per la ricerca dell'indice e per la restituzione della proof e spazio richiesto per la memorizzazione dell'albero all'interno del database:

Desktop > JS MerkleTreewithPatriciaTest.js > ...

```
const { StandardMerkleTree } = require("@openzeppelin/merkle-tree");
const crypto = require('crypto');
const fs = require("fs");
const PatriciaTrie = require('./PatriciaTrie7');
const { performance } = require('perf_hooks');
const path = require('path');

const iterations = 10000; // Numero di iterazioni da eseguire
const dataSize = 1000; // Numero di dati da inserire nel Merkle Tree

const fixedData = [];

for (let i = 0; i < 1000; i++) { // Dati da inserire nel Merkle, sono gli stessi anche nello scenario con solo
  const address = `0x${i.toString().padStart(40, '0')}`; // Esempio: 0x0000000000000000000000000000000000000000
  const value = `${i * 10000000000000000}`; // Esempio: 0, 10000000000000000, 20000000000000000, ...
  fixedData.push([address, value]);
}

const times = { // Necessario per calcolo media e dev. standard dei tempi
  treeCreation: [],
  findIndex: [],
  obtainProof: [],
};

for (let i = 0; i < iterations; i++) {
    const values = fixedData.slice(0, dataSize);

    // Misura il tempo iniziale
    const startTime = performance.now();

    // Genera il Merkle Tree
    const tree = StandardMerkleTree.of(values, ["address", "uint256"]);

    // Salva l'albero Merkle in un file JSON dopo la creazione
    //const folderPath = "C:/Users/alexn/Desktop/1000 Elements Patricia Trie and Merkle";
    //const filePath2 = path.join(folderPath, `tree_${i}.json`);
    //fs.writeFileSync(filePath2, JSON.stringify(tree.dump()));

    // Inserimento dei valori nel Patricia Trie
    const patriciaTrie = new PatriciaTrie();
    for (const [i, v] of tree.entries()) {
      const hashedKey = crypto.createHash('md5').update(v[0]).digest('hex'); // Chiave hashata, uso md5 per av
      patriciaTrie.add({ hash: hashedKey, data: i }); // Associare la posizione come valore, mentre la chiave
    }

    //const memoryUsed = patriciaTrie.calculateMemoryUsage();
    //console.log(`Stima della memoria occupata dal Patricia Trie: ${memoryUsed} byte`);

    // Salva l'albero Patricia in un file BSON (miglior compressione rispetto a JSON) dopo la creazione
    //const { BSON } = require('bson');
    //const serializedPatriciaTrie = BSON.serialize(patriciaTrie);
    //const filePath = path.join(folderPath, `PatriciaTrie_${i}.bson`);
    //fs.writeFileSync(filePath, serializedPatriciaTrie);

    const treeCreationTime = performance.now();
    times.treeCreation.push(treeCreationTime - startTime);
    console.log('Tempo per la creazione di Merkle e Patricia trees:', treeCreationTime - startTime, 'ms');
```

Fig. 4.3 (a): generazione e salvataggio del Merkle Tree e del Patricia Trie.

```
    // Calcola l'hash del dato di interesse, scelto in modo casuale
    const randomIndex = Math.floor(Math.random() * dataSize);
    const dataOfInterest = fixedData[randomIndex][0];

    const indexOfInterestStartTime = performance.now(); // Prendi i tempi per il calcolo dell'indice
    const dataHash = crypto.createHash('md5').update(dataOfInterest).digest('hex'); // Uso md5, così come fatto per
    //console.log('dataHash', dataHash);

    const valuepos = patriciaTrie.get(dataHash); // Indice ottenuto
    const indexOfInterestEndTime = performance.now();
    times.findIndex.push(indexOfInterestEndTime - indexOfInterestStartTime);
    console.log('Tempo per trovare l\'indice:', indexOfInterestEndTime - indexOfInterestStartTime, 'ms');

    console.log('Position', valuepos.position);

    if (valuepos.position !== null) {
      const proofObtainStartTime = performance.now();
      // Ottieni la proof utilizzando la posizione trovata nel Patricia Tree
      const proof = tree.getProof(valuepos.position);
      console.log('Proof:', proof); //se scelgo dataOfInterest in modo non casuale (in questo scenario e in quello
      const proofObtainEndTime = performance.now();
      times.obtainProof.push(proofObtainEndTime - proofObtainStartTime);
      console.log('Tempo Prova',proofObtainEndTime - proofObtainStartTime);
    } else {
      console.log('Dato non trovato nel Patricia Trie.');
    }
  }

function calculateStatistics(timesArray) { // Calcolo medie e dev. standard tempi
    const sum = timesArray.reduce((acc, cur) => acc + cur, 0);
    const average = sum / timesArray.length;

    const squaredDifferences = timesArray.map(time => Math.pow(time - average, 2));
    const variance = squaredDifferences.reduce((acc, cur) => acc + cur, 0) / timesArray.length;

    const standardDeviation = Math.sqrt(variance);

    return { average, standardDeviation };
  }

  console.log("Tempi per la creazione di Merkle e Patricia trees:");
  console.log(calculateStatistics(times.treeCreation));

  console.log("Tempi per trovare l'indice del dato di interesse:");
  console.log(calculateStatistics(times.findIndex));

  console.log("Tempi per ottenere la proof:");
  console.log(calculateStatistics(times.obtainProof));
```

Fig. 4.3 (b): ricerca dell'indice e restituzione della proof.

Gli steps eseguiti sono analoghi a quelli del caso precedente, con qualche aggiunta:

- Importazione delle librerie:
  - @openzeppelin/merkle-tree: è una libreria che permette di gestire Merkle Trees;
  - crypto: è il modulo di crittografia di Node.js, utilizzato per hash e altre operazioni crittografiche;
  - fs: è il modulo del file system di Node.js, usato per operazioni di lettura/scrittura dei file;
  - PatriciaTrie: è il modulo, mostrato nel Paragrafo 4.1, per la gestione dei Patricia Tries;

- performance: è un modulo per misurare le prestazioni in termini di tempo;
- path: è il modulo per gestire i percorsi dei file e delle directory.

- Definizione di variabili:
  - iterations e dataSize: numero di iterazioni e di dati da inserire nel Merkle Tree;
  - fixedData: array dove vengono preparati dati, a solo scopo dimostrativo, per essere inseriti nel Merkle Tree.
- Ciclo di esecuzione principale: il ciclo for esegue un numero specificato di iterazioni. In ciascuna iterazione:
  - viene selezionato un subset di dati da fixedData per popolare il Merkle Tree;
  - viene creato un Merkle Tree utilizzando la libreria @openzeppelin/merkle-tree con i valori selezionati;
  - viene creato un Patricia Trie, per il quale si utilizzano gli hash dei dati come chiavi e gli indici (ovvero, il riferimento sulla posizione occupata all'interno del Merkle Tree) come valori;
  - viene misurato il tempo per la creazione dell'albero e la memorizzazione dei dati;
  - viene selezionato in modo casuale un dato di interesse;
  - si misura il tempo per trovare l'indice del dato nel Patricia Trie;
  - se il dato viene trovato, viene calcolata la proof dal Merkle Tree e viene misurato il tempo impiegato per ottenere questa prova;
- Funzione calculateStatistics: vengono calcolati media e deviazione standard dei tempi suddetti e vengono stampati su console.

4.2.3 Risultati

I risultati ottenuti sono stati organizzati in 3 tabelle, così da valutare i vantaggi e gli svantaggi dei due approcci. Ci interessa, in particolare, stabilire quanto spazio aggiuntivo in memoria è richiesto, nel caso d'uso del Patricia Trie e qual è il guadagno che si ottiene, in questo caso, in termini di tempo necessario alla fase di data verification.

Nella prima tabella (Tab. 4.1) sono riportati lo spazio richiesto per l'archiviazione degli alberi e il tempo per la generazione degli alberi di entrambi gli approcci, in funzione del numero di elementi inclusi all'interno del Merkle Tree: come si può constatare, ovviamente al crescere del numero di elementi contenuti all'interno del Merkle Tree, cresce anche la dimensione richiesta per il salvataggio dei due alberi;

	# Elementi	Memory Usage (kB)	Tree Generation Time (ms)
Merkle	100	22.8	Avg: 39.78, StdDev: 11.87
Merkle with Patricia	100	49.4	Avg: 44.45, StdDev: 7.89
Merkle	300	68.7	Avg: 120.10, StdDev: 33.06
Merkle with Patricia	300	147.7	Avg: 126.27, StdDev: 24.84
Merkle	1000	231	Avg: 363.15, StdDev: 76.32
Merkle with Patricia	1000	491	Avg: 378.58, StdDev: 50.64
Merkle	1500	344	Avg: 512.35, StdDev: 61.02
Merkle with Patricia	1500	732	Avg: 538.96, StdDev: 74.28

Tab. 4.1: memoria e tempi richiesti per la generazione degli alberi.

inoltre, è interessante notare che, nell'approccio che fa uso anche del Patricia Trie, sarà necessario uno spazio circa pari a 2.1 volte lo spazio richiesto nel caso di salvataggio del solo Merkle Tree (anche il tempo richiesto per la generazione degli alberi aumenta, ma in modo decisamente meno rilevante). I risultati ottenuti in termini di tempo richiesto per la generazione dell'albero ci interessano in modo marginale, dal momento che le differenze tra i due casi non sono sostanziali e che, una volta generati gli alberi, questi tempi non influenzano in alcun modo le prestazioni che si ottengono in fase di data verification.

Nella seconda tabella (Tab. 4.2) invece, sono riportate le informazioni sui tempi di ricerca dell'indice e di calcolo della proof. Infine, nella Tab. 4.3 è riportato il tempo medio richiesto per la fase di verifica (qui, si tiene conto solo dei tempi necessari per la ricerca dell'indice e per il calcolo della proof: non viene preso in considerazione il tempo necessario affinchè l'utente, data la proof, calcoli la Merkle Root e la confronti con quella pubblicata su blockchain).

Ad ogni modo, trascurando i tempi richiesti per le proof, che nei due casi sono paragonabili (infatti, una volta a disposizione del Merkle Tree e dell'indice di riferimento, l'algoritmo per il calcolo della proof è lo stesso nei due casi), è possibile riscontrare un deciso miglioramento nel caso che fa uso di Patricia Trie: mentre, nel caso del solo Merkle Tree, si nota che il tempo per la ricerca dell'indice corrispondente al dato cresce con il numero di elementi inclusi nel Merkle Tree, nel caso di uso di Patricia Trie il tempo per la ricerca dell'indice è pressochè indipendente rispetto al numero di elementi contenuti nel Merkle Tree.

	# Elementi	Time required to find the index (ms)	Time required for proof (ms)
Merkle	100	Avg: 0.05, StdDev: 0.04	Avg: 0.58, StdDev: 0.26
Merkle with Patricia	100	Avg: 0.03, StdDev: 0.01	Avg: 0.56, StdDev: 0.18
Merkle	300	Avg: 0.07, StdDev: 0.05	Avg: 0.62, StdDev: 0.30
Merkle with Patricia	300	Avg: 0.04, StdDev: 0.01	Avg: 0.61, StdDev: 0.23
Merkle	1000	Avg: 0.15, StdDev: 0.12	Avg: 0.63, StdDev: 0.21
Merkle with Patricia	1000	Avg: 0.04, StdDev: 0.03	Avg: 0.64, StdDev: 0.24
Merkle	1500	Avg: 0.18, StdDev: 0.13	Avg: 0.64, StdDev: 0.19
Merkle with Patricia	1500	Avg: 0.05, StdDev: 0.02	Avg: 0.62, StdDev: 0.18

Tab. 4.2: tempi richiesti per la ricerca degli indici e per il calcolo della proof.

	# Elementi	Avg Time required for verification (ms)
Merkle	100	0,63
Merkle with Patricia	100	0,59
Merkle	300	0,69
Merkle with Patricia	300	0,65
Merkle	1000	0,78
Merkle with Patricia	1000	0,68
Merkle	1500	0,82
Merkle with Patricia	1500	0,67

Tab. 4.3: tempo medio richiesto per la fase di verifica.

Capitolo 5

Conclusioni

In questo elaborato è stata presentata una piattaforma, destinata all'uso aziendale, che permette il caricamento di dati e la conseguente certificazione degli stessi, sfruttando la tecnologia blockchain e le caratteristiche di due strutture dati, ovvero il Merkle Tree ed il Patricia Trie: in particolare, viene creato un Merkle Tree che contiene le informazioni da certificare. L'hash root del Merkle Tree costituisce il dato pubblicato sulla blockchain (Ethereum) attraverso una singola transazione.

Nel momento in cui un utente voglia verificare l'integrità di un dato, procede facendo una query al database, tramite cui può ottenere le informazioni di cui ha bisogno (proof), così da poter verificare, in modo autonomo, che il dato in suo possesso sia integro. In questo caso, dunque, una volta che l'utente avrà fatto la query al database e avrà ricevuto la proof specifica del dato del quale vuole effettuare la verifica di autenticità, potrà utilizzare tale proof per calcolare l'hash root del Merkle Tree. Successivamente, questo hash sarà confrontato con l'hash registrato sulla blockchain: se vi è corrispondenza, il dato è considerato integro e affidabile. Questo è il caso che è stato preso in considerazione, ovvero quello di database trustworthy.

In questo contesto, si è studiata la possibilità di velocizzare la data verification ricorrendo al Patricia Trie, ovvero ad una struttura particolarmente efficiente nell'indicizzazione di grandi set di dati: nel dettaglio, è stata implementata una soluzione tramite cui si effettua il salvataggio, all'interno del Patricia Trie, degli indici, relativi alle posizioni, dei dati presenti nelle foglie del Merkle Tree. Sono state valutate le prestazioni di questo approccio, rispetto all'uso più tradizionale del solo Merkle Tree. Tali prestazioni sono state valutate al variare del numero di elementi che compongono il Merkle Tree ed in termini dei tempi richiesti per la generazione degli alberi, per la ricerca degli indici e per il calcolo della proof e dello spazio richiesto per il salvataggio di tali alberi su database; fissato il numero di elementi presenti all'interno del Merkle Tree, sono, inoltre, state effettuate 10000 iterazioni per il calcolo dei valori medi e delle deviazioni standard dei tempi in esame (ad ogni iterazione, è stato variato in modo random il dato del quale si era interessati ad ottenere la proof).

I risultati ottenuti evidenziano che nel caso di utilizzo di Patricia Trie, a fronte di un maggiore tempo ed un maggiore spazio richiesti in fase di generazione degli alberi (lo spazio richiesto è pari a circa 2.1 volte rispetto al caso d'uso del solo Merkle Tree), si velocizza di molto il processo di ricerca dell'indice del dato (e ciò è tanto più vero quanti più elementi sono presenti nel Merkle Tree).

È possibile, pertanto, concludere che, in uno scenario in cui si ricevono richieste di certificazione con elevata frequenza e i Merkle Trees contengono molti elementi, utilizzare Patricia Tries per memorizzare le posizioni dei dati nei Merkle Trees risulta conveniente, in quanto permette di

velocizzare il processo di data verification (in particolare, la ricerca dell'indice del dato); tuttavia, questo costituisce a tutti gli effetti un trade off, dal momento che il vantaggio che si ottiene in fase di verifica della certificazione comporta, appunto, un maggior tempo e un maggior spazio richiesti (di conseguenza, un maggior costo, in fase di generazione degli alberi).

Bibliografia

[1] Panoramica della blockchain. https://www.ibm.com/it-it/topics/blockchain

[2] Blockchain: unlocking the value of distributed ledger technology in Private Equity. https://lpea.lu/blockchain-unlocking-the-value-of-distributed-ledger-technology-in-private-equity/

[3] Bigini, Gioele & Freschi, Valerio & Lattanzi, Emanuele. (2020). A Review on Blockchain for the Internet of Medical Things: Definitions, Challenges, Applications, and Vision. Future Internet. 12. 208. 10.3390/fi12120208.

[4] Blockchain: Permissionless vs Permissioned. https://www.ictsecuritymagazine.com/articoli/blockchain-permissionless-vs-permissioned/

[5] The Underlying Architecture Of Ripple's Permissioned Blockchain. https://www.rain.com/learn/the-underlying-architecture-of-ripples-permissioned-blockchain

[6] Ethereum Network Explained. https://www.lcx.com/ethereum-network-explained/#:~:text=Vitalik%20Buterin%20co%2Dcreated%20Ethereum,the%20creation%20of%20decentralized%20applications.

[7] Transazioni. https://ethereum.org/it/developers/docs/transactions/

[8] Introduction to Smart Contracts. https://ethereum.org/en/developers/docs/smart-contracts/#:~:text=A%20%22smart%20contract%22%20is%20simply,be%20the%20target%20of%20transactions.

[9] Albero di Merkle. https://it.wikipedia.org/wiki/Albero_di_Merkle

[10] Radix Tree. https://en.wikipedia.org/wiki/Radix_tree

[11] Cloud Data Provenance using IPFS and Blockchain Technology, July 8, 2019.

[12] Factom: Business Processes Secured by Immutable Audit Trails on the Blockchain, April 25, 2018.

[13] Node.js. https://it.wikipedia.org/wiki/Node.js

[14] MongoDB. https://it.wikipedia.org/wiki/MongoDB

[15] What is Ganache? https://trufflesuite.com/docs/ganache/

[16] Sviluppo e validazione di strumenti opensource per protocolli di certificazione e tracciabilità basati su DLT, Marco Farinasso