UNIVERSITÀ DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Database e Blockchain: un approccio decentralizzato all'integrità dei dati

Laureando

Riccardo Bragaglia

Matricola 521638

Relatore

Prof. Maurizio Pizzonia

Correlatore

Dr. Diego Pennino

Anno Accademico 2020/2021

Introduzione

Nella moderna era dell'informazione digitale è la norma che enti e sistemi di varia estensione abbiano a che fare con una certa quantità di dati informatici che rappresentano le varie informazioni che il mondo continuamente scambia, utilizza e modifica. Non esiste, infatti, strumento informatico che in un modo o nell'altro non utilizzi o memorizzi dati internamente ad esso o che ne scambi di continuo con l'esterno. Normalmente questo avviene tramite i *database*, uno strumento centralizzato che memorizza enormi quantità di dati e che è in grado di scambiarne con altri dispositivi ad esso connessi tramite l'utilizzo di tecnologie che ne garantiscono la stabilità e soprattutto la rapidità di accesso. I database permettono, tramite l'utilizzo del linguaggio *SQL*, di memorizzare, modificare, eliminare e duplicare i dati al loro interno organizzandoli come una tabella. Un utente che intende modificare la frazione di database dedicata ai suoi dati deve semplicemente comunicare i comandi che intende eseguire tramite le cosiddette *query SQL*. Quello di cui si tratterà di seguito, tuttavia, è piuttosto l'approccio alla sicurezza che queste tecnologie propongono e come si potrebbe fare per migliorare non solo l'isolamento delle informazioni ritenute personali o comunque sensibili, ma anche e soprattutto come si può garantire che questi dati restino effettivamente immutati all'interno del database. Se è tanto semplice eseguire una modifica sulla propria collezione di dati, allora è altrettanto semplice falsificare tale modifica, soprattutto considerando che gli strumenti di autenticazione principalmente utilizzati in questi casi sono, a loro volta, memorizzati in un database. Nell'era digitale troppo spesso le differenze tra un 1 ed uno 0 possono sconvolgere lo stato delle cose e avere il potere di forzare queste differenze risulta tanto una potente arma quanto un grave pericolo. A tale scopo negli ultimi anni arriva ad affermarsi una nuova tecnologia che propone all'interno del teatro

dei sistemi di sicurezza informatici un approccio decentralizzato basato su una condivisione di informazioni, regole ed atteggiamenti volta a formare una sorta di consorzio di enti che, seppure senza conoscersi o potersi effettivamente fidare tra di loro, hanno la garanzia che l'elevato numero di partecipanti e la condivisione di strette regole per la sicurezza possa garantire la sicurezza de propri dati. Tale tecnologia è detta *blockchain*. La blockchain è costituita da un registro decentralizzato composto da vari nodi, ciascuno contenente una certa quantità di informazioni, uniti come in una catena. La comunicazione interna ed esterna avviene tramite delle transazioni autenticate da un sistema di consenso distribuito in tutti i nodi della rete. Tramite questo, il sistema della blockchain garantisce l'immutabilità delle informazioni e l'accesso protetto ad ogni dato non secondo il giudizio di un ente singolo, il quale può essere aggirato, ma tramite un sistema di accordi interni e comuni a tutti i componenti della rete. Con questo tipo di sistema si cerca di emulare una struttura sociale dove ogni individuo partecipa attivamente alla collettività, potendosi fidare delle regole di quest'ultima e sapendo di avere una traccia delle operazioni che vengono eseguite, partecipando sia come controllore che come utilizzatore. La blockchain è già in uso come tecnologia in molti sistemi di memorizzazione di dati dal contenuto sensibile come diversi sistemi bancari, tra cui il famoso sistema delle criptovalute. In questa trattazione, si propone l'utilizzo della blockchain come metodo per garantire la sicurezza di un sistema di memorizzazione dati dimostrandone l'applicabilità in ambienti reali tramite un prototipo semplificato basato sul caso d'uso di una lista di opposizione, una tabella che assegna a ciascun utente un valore binario "Sì" o "No" per rappresentare il consenso di quell'utente al trattamento dei dati personali, a ricevere pubblicità o a qualsiasi scelta analoga, sempre modificabile dall'utente. Le funzionalità di gestione dei dati proprie dei database non sono facilmente realizzabili con l'attuale tecnologia blockchain. D'altronde una blockchain garantisce un livello di sicurezza che non è possibile ottenere da un database. L'obiettivo del lavoro è stato dunque quello di proporre un approccio ibrido che cercasse di sfruttare i vantaggi di entrambi i sistemi del database e della blockchain tramite l'*hash* crittografico dell'intero contenuto del database e la condivisione di questo tramite una blockchain, la quale assicura che le modifiche ai valori avvengano solo per mano di un utente che effettivamente ne possiede i diritti. Partendo da un prototipo già precedentemente realizzato,

si sono cercate le principali criticità ed i punti in cui intervenire per risolverle, cercando di simulare il più possibile un sistema reale immaginando come questo potrebbe essere utilizzato. Si è cercato, ad esempio, di migliorare il più possibile la fruizione parallela del servizio immaginando un caso d'uso in cui anche più utenti contemporaneamente potessero accedere alle informazioni, in qualsiasi momento e mantenendo i principi di sicurezza.

Nel primo capitolo di questa trattazione si presenteranno le principali tecnologie alla base del lavoro svolto. Verrà inoltre analizzato lo stato del progetto e l'analisi delle criticità in su cui intervenire. Nel secondo capitolo verrà presentato il processo progettuale alla base del sistema proposto analizzando l'organizzazione dei singoli elementi e le soluzioni che si sceglie di adottare per realizzare quanto visto nel capitolo precedente. Nel terzo capitolo verrà mostrata la realizzazione di un prototipo che propone un sistema più semplice ma che si basa su quanto progettato nel capitolo precedente, allo scopo di dimostrarne l'efficacia. Alla fine di questo capitolo, inoltre, verrà mostrato l'effettivo funzionamento del sistema tramite la simulazione di un caso d'uso.

Indice

Elenco delle figure

Capitolo 1

Contesto e obiettivi

1.1 La tecnologia Blockchain

Il sistema blockchain porta nel campo dell'informatica e della memorizzazione dei dati una soluzione opposta al sistema centralizzato proponendo un paradigma basato sulla condivisione di dati e programmi secondo un sistema interno di "fiducia" reciproca. Ad una blockchain fanno parte, a seconda delle regole scelte, un numero più o meno grande di partecipanti o nodi. L'idea è quella di simulare la capacità computazionale di un database o comunque di una macchina complessa condividendo la capacità individuale di ogni partecipante. Di conseguenza una rete di nodi più estesa è in grado di comunicare e gestire i dati in maniera sempre più efficace.

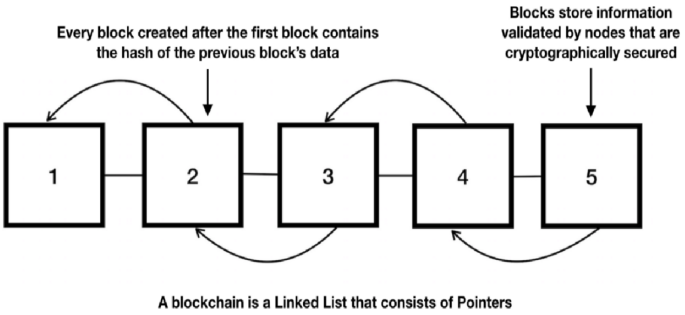

Figura 1.1: Esempio di Blockchain

Una caratteristica fondamentale di questo tipo di sistemi è la garanzia che il dato resti immutato e protetto dalle rigide regole di consenso adottate dalla rete. Se un dato viene memorizzato nella blockchain, infatti, ogni nodo si accorda con i restanti affinché quell'informazione sia valida o meno secondo le loro regole. In questo modo tutta la comunità può certificare la validità di un dato e, nel suo piccolo, ogni nodo partecipa individualmente a questa collettività decidendo di affidare le proprie informazioni alla sicurezza stipulata dal regolamento in vigore come accade, ad esempio, con le criptovalute. Quando, invece, un dato deve essere modificato, la blockchain come comunità verifica prima la validità della transazione secondo le regole e, solo dopo, la esegue. In questo modo tutti i nodi sono d'accordo su quali modifiche sono avvenute e quindi sul nuovo stato delle cose. La transazione viene quindi registrata in una specie di registro comune a cui tutti i nodi hanno accesso. Velocità e tempo di esecuzione dipendono strettamente dalla capacità di svolgere transazioni ad un ritmo elevato e con capacità molto variabili.

1.1.1 Confronto con un'architettura centralizzata

Le blockchain propongono un approccio all'organizzazione dei dati fondamentalmente opposto a quanto si è abituati a vedere con le strutture dati centralizzate. La sicurezza e l'integrità sono importanti prerogative di entrambe le tipologie, ma ciascuna le mette in atto in modo diverso con vantaggi e svantaggi. Se da un lato, infatti, la blockchain riesce a rendere sicuri i dati, dall'altro risulta che il tempo impiegato per la gestione dei dati cresca molto velocemente in funzione della quantità dei dati da gestire e dell'estensione della blockchain. Un alto numero di nodi richiede infatti molti controlli di integrità che favoriscono a rallentare ulteriormente le tempistiche di ciascuna transazione, il cui numero può aumentare molto nel caso in cui la mole di dati possa essere estremamente alta, a causa della grandezza fissa dei blocchi di cui è possibile eseguire transazioni.

I database relazionali, invece, sono basati sulla classica architettura *client/server* in cui un unico ente centralizzato riceve richieste dai client, le elabora e gestisce i dati di conseguenza. Questo permette la gestione di grandi quantità di dati in tempi più o meno noti, garantendo una crescita lineare del tempo di esecuzione in funzione dell'aumento

del volume di dati. Le tempistiche aumentano e la velocità diminuisce ma comunque è garantito un flusso di informazioni congruo a quelle tempistiche. Queste caratteristiche hanno affermato nel tempo l'affidabilità del database relazionale che rimane ad oggi lo standard principalmente utilizzato.

1.2 ADS e DB-Tree

Le *ADS (Authenticated Data Structures)* sono particolari strutture dati che garantiscono la sicurezza dei dati memorizzati tramite la continua autenticazione delle richieste che elabora. Questo è possibile grazie alla generazione di una prova (*proof*) che accompagna ogni richiesta di modo che si possa capire in ogni momento se l'accesso sia valido o meno.
L'implementazione delle ADS può avere molte forme a seconda degli utilizzi, uno in particolare è l'*hash tree* o Merkle tree.

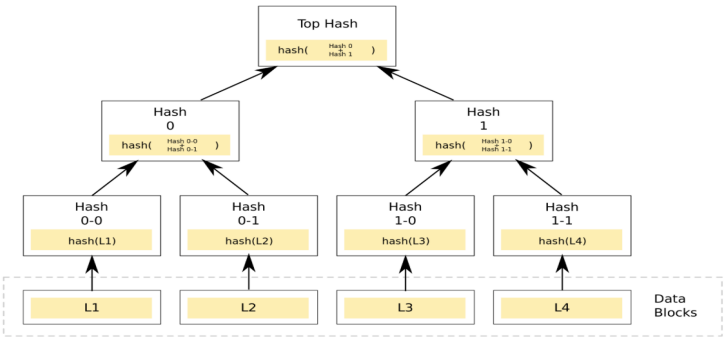

Figura 1.2: Esempio di Hash Tree o Merkle Tree

Secondo questa struttura, i dati sono organizzati in un albero in cui ciascuno dei nodi è identificato da un hash crittografico, ovvero una stringa di valori, che rappresentano il contenuto dell'albero sottostante. In questo modo, salendo di livello, è possibile ottenere un unico valore che effettivamente rappresenta l'intero corredo di informazioni memorizzate all'interno della struttura. Tale valore è detto root hash. Questo valore rappresenta, nei termini dettati dalla configurazione ADS, la proof con cui autenticare le richieste. È possibile determinare se una richiesta avviene o meno da un ente fidato

che conosce il root hash semplicemente confrontando i valori. L'organizzazione ad albero del database, denominata DB-Tree, garantisce anche il vantaggio di una persistenza maggiore rispetto alla controparte tabellare. Secondo la struttura ad albero, infatti, è possibile determinare in ogni momento se avviene o meno un cambiamento in un qualsiasi punto del contenuto del database dato che ne risulterebbe una proof differente, essendo questa composta da una combinazione di tutti i nodi che compongono l'albero. I dati, che normalmente verrebbero memorizzati in un sistema a tabelle in cui ogni riga corrisponde ad un record, sono quindi più sicuri e persistenti.

Un ultimo vantaggio, infine, è la diminuzione del numero di operazioni per operare una richiesta. Se in un database organizzato in maniera tabellare, infatti, è necessario scorrere tutti i record per trovare quelli di interesse, l'organizzazione ad albero riesce a separare le sezioni dell'albero che contengono il record desiderato finendo per eseguire meno operazioni.

1.3 Struttura del Root Hash

Per poter accordare i nodi della blockchain su una versione del database che tenga conto del completo corredo di informazioni al suo interno si usa un protocollo di hash possibile proprio grazie alla configurazione ad albero del database.

Per hash si intende una stringa crittografica ottenuta da un processo che riceve input di grandezza variabile. L'hash può rappresentare in modo esaustivo l'informazione di partenza senza che il processo inverso sia possibile e, inoltre, qualora una frazione delle informazioni in ingresso dovesse cambiare, il suo valore cambierebbe di conseguenza permettendo di riconoscere la differenza.

Grazie a questo processo è possibile eseguire un processo di hashing per ciascun nodo combinando in particolare l'indirizzo del nodo ed il valore dei suoi dati insieme ad eventuali hash di nodi "figli". Così facendo è possibile calcolare un singolo hash per ciascuno dei nodi dell'albero e, ripetendo l'operazione, si compone il root hash, un codice

ottenuto dalla combinazione di tutti i nodi sottostanti alla radice e che quindi sintetizza l'intero contenuto dell'albero.

1.3.1 Processo di autenticazione

Ogni nodo della blockchain decide quindi di accordarsi su di uno stesso root hash che rappresenta l'ultima versione accettata del database.

A questo punto, qualora un nodo volesse eseguire un'operazione, occorre che si calcoli l'hash relativo alla versione che intende modificare e, se risulta uguale a quello comunemente accettato allora può proseguire. Questo perché il root hash tiene conto sia dei valori che degli indirizzi dei nodi, per cui un hash da un altro nodo esterno alla rete risulta per forza di cose estraneo e viene bloccato.

Questo processo è denominato autenticazione e rappresenta il primo scoglio da superare per un nodo che intende accedere ai dati della blockchain.

1.4 Sistema basato su un approccio decentralizzato

L'obiettivo di questa trattazione è quello di teorizzare e progettare un sistema di memorizzazione dati che sia il quanto più possibile analogo ai comuni sistemi basati su architetture centralizzate basati su database relazionali, ma che sfrutti una proposta ibrida costituita dalla sintesi di blockchain e DB-Tree. Il sistema in questione deve garantire tutti i sistemi di sicurezza propri della blockchain per la gestione dei dati di una popolazione di utenti anche per grandi transazioni, permettendo accessi complessi e multipli dall'alto ordine di esecuzioni al secondo.

I casi d'uso da prendere in considerazione provengono da classici sistemi informatici come e-commerce e sistemi bancari in cui ciascun utente possiede dei dati di cui solo lui è proprietario. L'utente deve poter accedere a quei dati in ogni momento, anche contemporaneamente ad altri utenti e deve poter eseguire modifiche anche complesse come l'aggiunta o la rimozione di prodotti da un carrello, la compilazione di documenti

complessi o transazioni monetarie con altri utenti. In questo contesto il sistema garantisce che le operazioni avvengono secondo il paradigma decentralizzato basato sulla blockchain di modo che ciascuna transazione possa essere registrata e certificata.

Per fare ciò è necessario descrivere un ambiente in cui il bacino di utenza sia molto ampio in modo da garantire un numero elevato di transazioni per unità di tempo. È inoltre importante avere una blockchain che sia in grado di elaborare funzioni estremamente complesse, legate alla natura complessa delle transazioni, ad una latenza minima anche avendo un'elevata popolazione. Si suppone che il sistema sia ben scalabile per poter permettere l'ingresso o l'uscita di un utente in qualsiasi momento sia dal punto di vista di organizzazione della blockchain, che da quello relativo ai protocolli di hashing, autenticazione e memorizzazione dati.

1.4.1 Intervento sul progetto

Nel corso dello svolgimento dell'attività di tirocinio l'obiettivo è stato quello di intervenire su di un sistema che già in modo preliminare rappresentava una prima scrematura del progetto vero e proprio.

Il prototipo realizzato su cui è stato svolto il lavoro consisteva in un sistema elementare che permetteva l'interazione di un solo utente alla volta per poter modificare le sue informazioni. Il sistema era inoltre sprovvisto di una metodologia per restare in attività e simulare lunghe sessioni di funzionamento, caratteristica utile per valutare la stabilità a lungo termine del sistema. Lo studio di questo primo modello si è svolto in due fasi principali:

- Studio ed analisi delle tecnologie, dei linguaggi e dei paradigmi usati per la realizzazione e la progettazione del modello, principalmente concentrato su come le varie parti interagissero in modo da capirne il funzionamento. Dopo questo è stato necessario progettare un modello di refactoring per adattare il sistema alle modifiche che si intendevano aggiungere: la possibilità di avere più di un client e questi potessero interagire liberamente, e la possibilità che il sistema restasse attivo per sessioni anche più lunghe.

- Intervento e refactoring di quanto descritto precedentemente, il cui dettaglio sarà descritto con maggiore cura nei capitoli seguenti. Le modifiche non hanno richiesto l'aggiunta di tecnologie aggiuntive a quelle utilizzate. Le due modifiche principali hanno riguardato la problematica descritta che verrà descritta in seguito come "problematica delle multiple operazioni" e la possibilità di eseguire un load delle informazioni legate alla sessione che verrà descritta con cura nel capitolo dedicato alla blockchain ed alle sue componenti.

1.4.2 Problematica delle multiple operazioni

Dopo che un nodo si è autenticato ed ha eseguito la sua modifica, viene ricalcolato il nuovo root hash e tutti i nodi si accordano su questo valore. In situazioni reali, però, dove l'afflusso di utenza di un sistema simile può essere anche molto vasto, è necessario considerare che se due nodi richiedono entrambi contemporaneamente di eseguire una modifica, per forza di cose ciascuno calcolerà un nuovo root hash che non tiene conto delle modifiche avvenute in contemporanea da parte dell'altro nodo e questo rappresenta un problema per il sistema.

Si deve cercare, quindi, un sistema di gestione delle richieste che possa tener conto delle eventuali operazioni multiple che possono avvenire da parte di due o più nodi. Il sistema proposto è dunque responsabile di costruire una lista di modifiche da attuare tutte a cascata, rispettando l'architettura ad albero, per far sì che ogni modifica avvenuta anche in brevissimi intervalli di tempo possa essere memorizzata effettivamente all'interno del root hash e che questo, quindi, possa essere usato in seguito. Questo processo verrà chiamato di seguito "*queue*" o "coda".

L'idea infatti è quella di poter conservare un certo numero di transazioni in una coda per poterle poi applicare in maniera sequenziale di modo che nessuna modifica vada perduta o sovrascritta da altre.

Capitolo 2

Progettazione

In questo capitolo si descrivono nel dettaglio le scelte progettuali per la realizzazione, in linea teorica, del sistema complesso descritto nel capitolo precedente. È importante precisare che la progettazione del sistema si basa sulle premesse di un ambiente che possa garantire la capacità di mettere in atto quanto proposto.

2.1 Struttura della rete

La rete è concettualmente composta da quattro entità fondamentali che comunicano tra loro attraverso canali diversi.

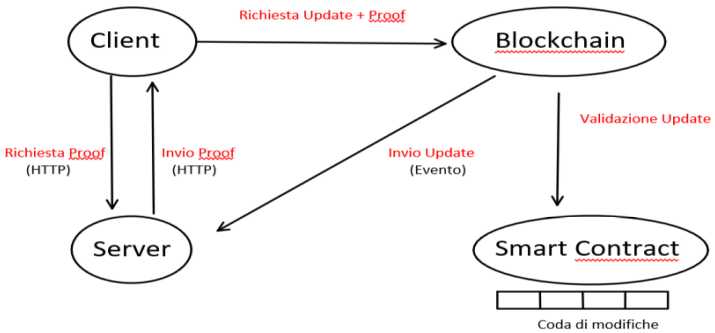

Figura 2.1: Schema concettuale del sistema

2.2 Client

Le funzioni relative al client sono responsabili di gestire le richieste di interazione con i dati appartenenti all'utente.

Va garantito innanzitutto un modo per assicurare la sicurezza dell'accesso e che questo sia possibile solo per mano di utenti che effettivamente possiedono i diritti sui dati. Questo è possibile tramite la generazione di credenziali che permettono di identificare un nodo all'interno della blockchain così da garantire che nessun altro nodo possa in qualche modo sostituirsi o manomettere informazioni che non possiede.

Dopo che il nodo è autenticato esso procede con la modifica delle informazioni (un prelievo o un deposito nel caso bancario, un'aggiunta al carrello o un pagamento nel caso e-commerce). Per farlo, il software deve eseguire prima una serie di controlli volti a certificare l'integrità delle informazioni memorizzate nel server con i quali si sta interagendo eseguendo un controllo incrociato dell'ultima versione del database.

A questo punto viene richiesto l'intervento dello *smart contract*, una funzione che rappresenta l'autorità condivisa della blockchain. È sua responsabilità quella di gestire le richieste del client seguendo le regole stabilite dal consenso comune adottato dalla blockchain. Il client invia allo smart contract una richiesta accompagnata da una proof, ovvero una lista di hash che combinati danno luogo al root hash del server.

Il software solleva l'utilizzatore dalla responsabilità di gestire le comunicazioni con la blockchain che è assegnata allo smart contract allo scopo di garantire alla comunità la sicurezza promessa.

L'architettura deve essere costruita in modo da prevedere un numero anche molto alto di client diversi, ciascuno con un codice univoco, e di un indirizzo che permetta di differenziarsi dagli altri. La chiave di questo progetto, infatti, è ampliare il prototipo da cui si partiva aggiungendo chiavi personalizzate a ciascun client. Il sistema inoltre sarà capace di differenziare le richieste e suddividere l'intervento a seconda del richiedente.

2.3 Blockchain

La blockchain è costituita da una serie di nodi il cui compito è certificare la sicurezza delle operazioni che avvengono nel sistema. È compito suo assicurarsi che tutte le funzioni principali e più delicate come la modifica dei dati memorizzati dal server avvenga secondo delle regole prefissate.

Ciascun nodo infatti dispone di un root hash che rappresenta l'ultimo stato valido dei dati memorizzati nel server, allo scopo di riconoscere velocemente qualsiasi cambiamento, sia esso certificato o meno.

La blockchain utilizzata per il progetto ed il prototipo è Quorum [GoQ], una tipologia basata su Ethereum [Eth]. All'interno di Quorum il principale metodo di comunicazione avviene tramite transazioni, destinate o ad altri nodi o ad altre entità, gli smart contract.

2.4 Smart Contract

Lo smart contract è la funzione principalmente responsabile di rappresentare e gestire il consenso comunemente adottato dalla blockchain. Si presenta come una funzione scritta in Solidity, un linguaggio appositamente pensato per la scrittura di contratti all'interno dell'ambiente Quorum. La particolarità sta nel fatto che ogni nodo della rete si accorda sull'utilizzare un'unica istanza del contratto della quale viene eseguito il deploy, ciò significa che la funzione viene eseguita di comune accordo tra tutti i nodi allo scopo di decentralizzare il compito di controllare la sicurezza delle operazioni.

Le principali responsabilità dello smart contract sono quelle di autenticare una richiesta proveniente da un Client secondo l'ultimo root hash valido, immagazzinare le richieste che i client autenticati propongono, comunicarle al server ed aggiornare il root hash in vigore tenendo conto di tutte le modifiche avvenute.

Come prima cosa lo smart contract riceve una richiesta da un client abbinata ad una

proof, ovvero ad una lista di hash la cui combinazione, si suppone, dia luogo al root hash del server. Il primo controllo di sicurezza consiste nell'accertarsi che il root hash memorizzato ed accettato dalla comunità sia uguale a quello proposto dal client che sta richiedendo l'accesso.

Quando la richiesta del client è stata autenticata lo smart contract procede con la memorizzazione della modifica e del client che l'ha eseguita. In questo modo quando vengono eseguite un certo numero di richieste, il contratto le applica a cascata combinando le modifiche nel nuovo root hash che diventa a questo punto, il nuovo standard per i futuri controlli.

A questo punto, infatti, il contract non cessa la sua esecuzione ma rimane in attesa di altre interazioni da parte di altri client, che dunque ne dovranno caricare i dati attualmente validi. Questa funzione è stata aggiunta in quanto, precedentemente, il sistema cessava l'esecuzione. Si è ritenuto, quindi, che fosse fondamentale inserire la possibilità di caricare i dati dello smart contract in modo dinamico per poter simulare anche una lunga serie di operazioni in un caso d'uso simile alla situazione reale.

All'interno dello smart contract è stata anche realizzata la principale modifica al progetto originale che prevede la possibilità di risolvere la problematica delle multiple operazioni costruendo una coda di modifiche, descritta con cura di seguito.

Il contratto è una delle parti più importanti dell'intero progetto dato che rappresenta l'intera filosofia della blockchain facendo rispettare le regole di consenso e attuando la maggioranza di controlli.

2.4.1 Coda delle modifiche

Una delle parti fondamentali della progettazione dello smart contract consiste nel fornire un metodo di memorizzazione delle modifiche che vengono proposte. Questo allo scopo di evitare il sovrascrivere, o la perdita, delle modifiche che avvengono in rapida successione.

Per farlo si è pensato di usare una tabella indicizzata che potesse abbinare un hash alla sua versione calcolata sulla base della modifica. Così facendo quando il contratto dovrà applicare le modifiche potrà basarsi sulle coppie di hash vecchi e nuovi che sono stati memorizzati nel corso delle esecuzioni precedenti.

Questo metodo di modifica permette di risolvere il problema delle multiple operazioni dato che il contratto è in grado di collezionare le modifiche che avvengono in rapidissima successione e di applicarle con criterio in modo da non perdere nessuna informazione.

2.4.2 Calcolo del Root Hash

Il root hash svolge un ruolo fondamentale all'interno dell'esecuzione dello smart contract. Esso viene calcolato e memorizzato dal contratto di modo che ci si possa far riferimento nel caso sia necessario eseguire l'autenticazione di una richiesta.

Il processo per calcolare il root hash è quindi di fondamentale importanza ed è basato sulle funzioni di hashing messe a disposizione da Quorum, gestibili attraverso il linguaggio di programmazione ed inseribili all'interno della funzione che li deve utilizzare.

Il processo di calcolo consiste nel generare un hash parziale combinando i dati propri di un nodo con una chiave univoca che lo rappresenti. In questo modo, la stringa di lunghezza fissa rappresenta in modo univoco una determinata configurazione di dati all'interno di un determinato nodo. Seguendo poi la struttura ad albero si applica lo stesso algoritmo di hashing su tutti gli hash parziali di un certo livello, combinando quello che è l'hash di un sottoalbero di nodi. Ripetendo questa operazione fino alla radice si genera il root hash che, di conseguenza, deve risultare univoco a prescindere dal punto di partenza e dal percorso svolto.

Per aiutarsi con questo processo, lo smart contract segue le linee guida della proof, il valore di prova abbinato alla richiesta che contiene sottoforma di liste questi hash

parziali, ordinati secondo la sequenza per cui vanno combinati rispettando la struttura.

Il contratto può eseguire questo calcolo con o senza modifica, a seconda dell'uso che si deve fare del root hash. In caso di autenticazione, infatti, si limita a calcolare il root hash proposto da un nodo secondo la proof ricevuta allo scopo di autenticarlo. Se invece si deve aggiornare il root hash corrente perché sono state abilitate delle modifiche, l'esecuzione avviene tenendo conto dei valori presenti nella coda delle modifiche, sostituendo di volta in volta un valore con il corrispettivo aggiornato.

2.5 Server

La seconda parte fondamentale dell'architettura del sistema è formata dal lato server. I nodi responsabili di gestire questa funzione non devono solo essere in grado di memorizzare le informazioni, ma soprattutto di gestire le diverse richieste che il client ed il contratto immettono nella rete.

Le tempistiche del server dettano il ritmo e la sequenza di esecuzioni sulla rete, ma esso non ha alcuna responsabilità in termini di sicurezza o gestione centralizzata della connessione, che restano una responsabilità dello smart contract.

2.5.1 Differenziare le richieste

Per attuare le modifiche al progetto originale e quindi per permettere l'esecuzione di più richieste in rapida successione, o anche contemporaneamente, la complessità progettuale del server è stata aumentata. In particolare si deve permettere di differenziare la provenienza di determinate modifiche per poter indicizzare i dati, e si deve poter distinguere una proposta di update da una conferma di update.

La prima avviene quando un client, attraverso lo smart contract, propone una modifica ai suoi dati. In quel caso il server deve poter memorizzare la proposta in modo da poterla applicare ai dati persistenti solo in seguito, quando avrà ricevuto la conferma.

La seconda, la conferma di update, è proprio il segnale che comunica al server di applicare in modo permanente tutte le modifiche ricevute fino a quel momento.

La differenziazione di queste due procedure avviene a seconda della capacità del server di distinguere le comunicazioni dello smart contract, dette eventi.

2.5.2 Struttura della Proof

Un dato fondamentale che il server deve poter fornire in base alle richieste, oltre al contenuto dei registri relativi a ciascun utente, è la proof.
La prova si presenta come una lista di hash ciascuno rappresentante un sottoalbero della struttura e che, combinati insieme, formano il root hash.
Insieme alla proof è presente il corredo di informazioni relative al nodo, in questo modo è sempre possibile eseguire l'algoritmo di hasing per combinare il contenuto con gli altri hash parziali.

Di seguito un esempio di proof per come è stata strutturata in fase di realizzazione del prototipo. Le informazioni relative al client, per facilità di calcolo, sono limitate ad essere semplicemente di carattere binario, ed i livelli di profondità del percorso composto dagli hash parziali sono limitati ad un numero minimo.

```
[
        "s",
        [
                "0x0f4196168aa62037d5b261178a791453715f2628f4bf934ae26c2fd6e8be4a0d",
                " ",
                "0x8fb6f46cda60b29b34a8549d1ae716b1313e4993b0ec2fb7b41e2cf7ef224ba9"
        ],
        [
                "0xb4d8b04eb42e3cb6544ab2c0cbabe75cf2d729df63bd0769bb5c767a941d5f88",
                ["0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C","s"],
                " "
        ],
        0
]
```

Figura 2.2: Esempio di proof

Si noti che in alcune sezioni l'hash parziale è assente. In quel caso viene posto un segno che sta a rappresentare di dover considerare l'hash parziale ottenuto dalla combinazione dell'indirizzo del nodo in questione ed il suo valore. Questo si fa al solo scopo di diminuire il carico informativo ed evitare la ridondanza di informazioni.

In altre sezioni, invece, è possibile vedere una coppia formata da un valore ed un indirizzo, rappresentante un nodo appartenente a quella sezione di albero.

L'ultimo valore della lista è un intero rappresentante la versione del database. Serve a riconoscere quante modifiche avvengono in fase di test del prototipo.

2.6 Fasi della comunicazione

Quanto descritto rappresenta la linea progettuale di un sistema che è in grado di gestire una certa quantità di dati seguendo una filosofia decentralizzata, utilizzando l'architettura ad albero del database per autenticare le interazioni.

Il caso d'uso principale per il sistema progettato riguarda una popolazione di utenti, titolari di una quantità di dati memorizzati all'interno del sistema, che richiede l'accesso ad essi con tempistiche e latenze molto diverse tra loro.

1. Ciascun nodo della blockchain mette a disposizione dell'utente che vi si autentica come proprietario, un'interfaccia software per interagire con la parte server. Ciascun nodo verifica quindi l'integrità del server che sta contattando richiedendo la proof relativa alla sua area di competenza.
2. Il server cattura la richiesta del client.
3. La richiesta è soddisfatta, il server fornisce la proof desiderata costruendola seguendo il percorso della struttura ad albero.
4. Il client riceve la proof.
5. Il nodo client prepara una richiesta di modifica ai dati in suo possesso inviando la sua proposta allo smart contract con in allegato la proof che ha ricevuto.

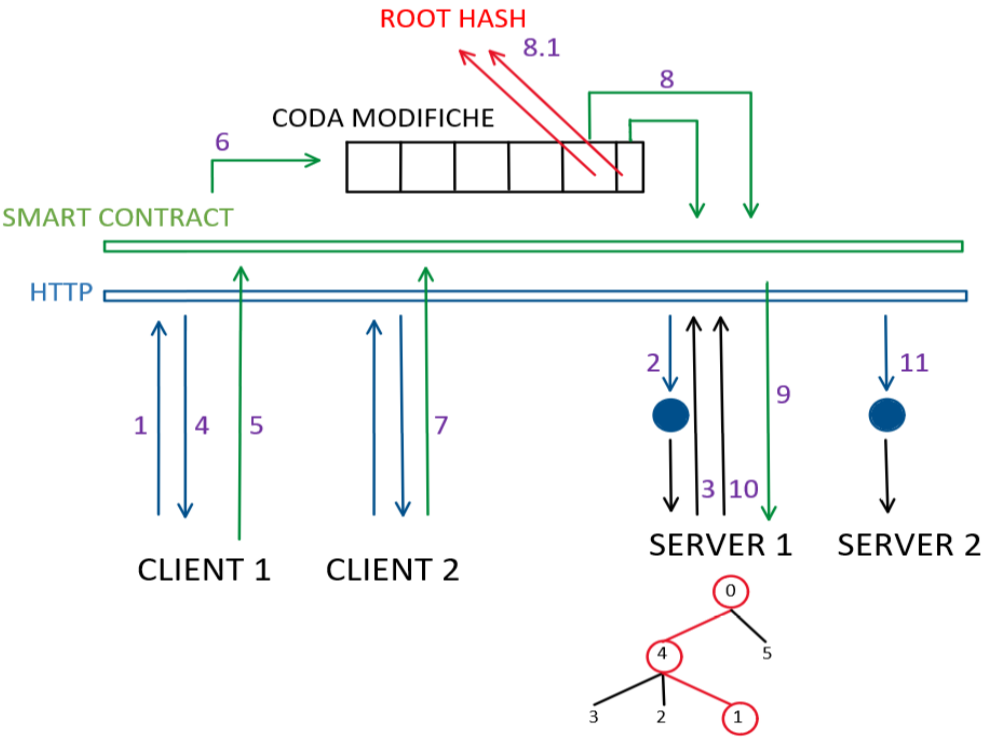

Figura 2.3: Sequenza delle operazioni. (Vedere elenco numerato)

6. Lo smart contract prende in carico la richiesta e si accerta che provenga da un nodo fidato, confrontando la prova allegata con lo stato ultimo del database tramite l'algoritmo di hashing. Se la modifica è approvata dalle regole dettate dal contratto questa viene memorizzata in attesa della conferma nella coda delle modifiche.

7. Secondo il caso d'uso si suppone che in qualunque momento un altro client possa eseguire la stessa richiesta contemporaneamente alle operazioni che si stanno svolgendo.

8. Quando sono state collezionate una certa quantità di modifiche, il contratto procede a contattare il server per inviare ogni modifica memorizzata, procedendo poi con l'aggiornamento del root hash (8.1) in modo da poter autenticare eventuali accessi futuri.

9. Il Server cattura la comunicazione dello smart contract che contiene l'update da effettuare e lo esegue sui suoi record. Da notare che in questa fase il Server non ha

potere decisionale sulla modifica, essendo questa arrivata da un ente fidato, viene applicata senza ulteriori passaggi.

10. Il server procede quindi ad aggiornare gli altri server del sistema dell'avvenuta modifica.

11. Gli altri server ricevono la modifica e la applicano ai loro record.

A questo punto il sistema è pronto per eseguire ulteriori modifiche seguendo la stessa sequenza di operazioni. Ogni entità nella rete è ora accordata su una nuova versione dei dati memorizzati e tale modifica è avvenuta secondo le regole definite dallo smart contract condiviso sul sistema.

Capitolo 3

Realizzazione

In questo capitolo si presentano nel dettaglio le tecnologie utilizzate per produrre un prototipo basato sul caso d'uso semplice derivato dalla generalizzazione descritta precedentemente. L'idea è quella di dimostrare che la linea di pensiero progettuale sia valida dimostrandone l'efficacia in un ambiente ristretto e semplificato.

Il prototipo realizza il caso d'uso generico visto precedentemente, ma nell'ambiente specifico semplificato di una lista di opposizione. Le liste di opposizione sono record che abbinano ad ogni utente un valore binario vero/falso che rappresenta l'adesione dell'utente a ricevere una pubblicità o ad abilitare il trattamento dei dati personali [pdoM].

Analogamente al sistema progettato, l'utente ha la possibilità in ogni momento di modificare la sua scelta. Inoltre è compito della blockchain garantire che l'accesso sia certificato e sicuro come già discusso.

La realizzazione del prototipo è partita da una versione precedentemente realizzata, alla quale sono state modificate ed aggiunte sezioni relative alla gestione delle multiple operazioni anche da vari client in sessioni di durata molto estesa.

3.1 7Nodes e Docker

Per simulare ed organizzare l'infrastruttura su cui vengono eseguiti i programmi si utilizza *7Nodes* [Git], una blockchain composta da sette nodi, ciascuno identificabile da un file chiave che ne riporta l'autenticità e che non è possibile duplicare.

Ciascun nodo è rappresentato da una macchina virtuale messa a disposizione da *Docker* [fD], un sistema che permette di avviare più virtualizzazioni sulla stessa macchina per simulare piccole reti.

```
> Task :upDocker
Creating network "quorum-examples-net" with driver "bridge"
Creating quorum-examples_txmanager6_1 ...
Creating quorum-examples_txmanager7_1 ...
Creating quorum-examples_txmanager5_1 ...
Creating quorum-examples_txmanager1_1 ...
Creating quorum-examples_cakeshop_1   ...
Creating quorum-examples_txmanager2_1 ...
Creating quorum-examples_txmanager4_1 ...
Creating quorum-examples_txmanager3_1 ...
Creating quorum-examples_node7_1      ...
Creating quorum-examples_node3_1      ...
Creating quorum-examples_node1_1      ...
Creating quorum-examples_node5_1      ...
Creating quorum-examples_node4_1      ...
Creating quorum-examples_node2_1      ...
Creating quorum-examples_node6_1      ...
```

Figura 3.1: Output del comando "upDocker" per avviare le macchine virtuali

Nel caso specifico vengono creati sette Docker per costruire la blockchain, in aggiunta ai quali il programma avvia anche altre macchine di controllo che fungono da manager del sistema.

3.1.1 Distribuzione dei ruoli nella rete

Partendo dal progetto precedentemente realizzato in cui 6 nodi avevano il ruolo di server ed era quindi disponibile un solo client, si è deciso di riorganizzare la struttura.

La distribuzione dei ruoli per i 7 nodi è così organizzata:

- I nodi dal numero uno al numero quattro svolgono il ruolo di server, in particolare sui nodi uno e due il server è gestito da Postgres [wmaosd], sui nodi tre e quattro è gestito da MariaDBSQL [Fou]. Questi nodi hanno il compito di catturare le comunicazioni di lettura (*GET*) e scrittura (*POST*) dei dati conservati al loro interno.
- I nodi dal numero 5 al numero 7 svolgono il ruolo di client. Questi hanno il compito di simulare l'utente del sistema che partecipa alla blockchain e intende modificare il suo valore all'interno del database.

3.2 Client

Il programma in esecuzione sul client ha lo scopo di eseguire un update del valore relativo al client stesso. È sua responsabilità effettuare il deploy o il load dello Smart Contract ed effettuare le richieste HTTP destinate al server.

3.2.1 Prime fasi dell'esecuzione

Come prima cosa, quando viene avviato, il client esegue una verifica dell'integrità dei server ai quali richiederà l'update. Per farlo richiede la proof dai nodi uno e tre e confronta che le proof ottenute siano identiche tra loro. Questa operazione funge da test preliminare della connettività alla rete e di integrità dei server. Questo, ed altri test che avvengono successivamente fungono da sbarramento, infatti se per qualsiasi motivo dovesse capitare che il client non ottiene una conferma positiva, l'esecuzione sarebbe annullata, per evitare rotture della sicurezza.

Stabilita l'integrità della connessione, il client si collega alla Blockchain tramite le credenziali contenute nel file "key" che all'interno dell'ambiente Quorum sono chiamate *"Wallet"*.

```
private fun loadWallet(): Credentials {
    return WalletUtils.loadCredentials(
        password: "",
            source: "/home/key$clientNumber"

    )
}
```

Figura 3.2: Metodo "loadWallet" per caricare le credenziali del nodo che sta eseguendo la funzione

Questo impedisce che qualunque nodo possa eseguire la modifica ai dati di qualsiasi altro nodo semplicemente fingendo di avere il suo indirizzo IP. Ogni nodo è unico nella blockchain e possiede le proprie credenziali per accedere solo ai propri dati.

3.2.2 Deploy dello Smart Sontract

Dopo aver eseguito l'autenticazione, il client esegue il deploy dello smart contract sulla blockchain. In questo modo viene creata un'istanza di smart contract che da ora in avanti sarà lo standard nella rete. Questa operazione avviene una sola volta, dato che lo stato iniziale in cui lo smart contract viene pubblicato cambia durante le esecuzioni. Questa operazione la esegue solo il primo client che effettua l'update di simulazione.

Il codice del contratto, scritto in Solidity, viene compilato e si ottengono due oggetti: il file binario *EVM* e l'*ABI*. L'ABI costituisce l'interfaccia binaria del contratto, ne descrive le funzioni ed i parametri, il file EVM descrive invece l'insieme di operazioni di basso livello necessarie ad attuare le funzioni d creazione del contract.

Questi due component vengono poi combinati tramite Web3J, un'applicazione di interfaccia che permette la comunicazione tra lo smart contract ed il sistema [Doc]. L'interfaccia genera un *wrapper*, un'applicazione che funge da collegamento e presenta tutte le funzioni principali che lo smart contract mette a disposizione.

Avendo a disposizione il wrapper compilato, il programma esegue la funzione di deploy

definita nell'interfaccia, inviando le informazioni di autenticazione. La comunicazione avviene tramite il metodo *send*.

```
private fun deployContract(web3j: Web3j, credentials: Credentials): OppositionSmartContract {
    val contract =
            OppositionSmartContract.deploy(
            web3j,
            credentials,
            BigInteger.ZERO,
            DefaultGasProvider.GAS_LIMIT
        ).send()
    return contract
}
```

Figura 3.3: Metodo "deploy" che richiede la pubblicazione del contact tramite Web3J

L'interfaccia Web3J è responsabile di tutta la comunicazione tra i programmi del sistema e lo smart contract, soprattutto quando si tratta di gestire gli eventi, ovvero le comunicazioni in uscita dal contratto.

3.2.3 Load dello Smart Contract

La fase di *load* del contract è un'alternativa alla fase di deploy, la quale viene svolta solo dal primo nodo che inizia la comunicazione. Questa opzione viene aggiunta per garantire una certa longevità al sistema che può caricare i dati dello smart contract e quindi simulare lo stato operativo reale. Da quel punto, infatti, i dati dello smart contract memorizzati in tempo reale devono essere mantenuti e ogni nodo deve potervi accedere ed agire sulla base di quei valori. Per fare ciò il programma del client riconosce la richiesta di load del client che, tramite il terminale, specifica l'indirizzo del wallet relativo al nodo che per primo ha eseguito il deploy del contract.

```
private fun loadContract(contractAddress: String, web3j: Web3j, credentials: Credentials): OppositionSmartContract {
    val contract =
            OppositionSmartContract.load(
                    contractAddress,
                    web3j,
                    credentials,
                    BigInteger.ZERO,
                    DefaultGasProvider.GAS_LIMIT
            )
    return contract
}
```

Figura 3.4: Metodo "loadContract" del client. Differisce dal delploy per l'indirizzo da cui caricare i dati

Questa operazione all'interno del sistema prototipo che si sta descrivendo avviene manualmente alla richiesta, tramite terminale, dell'esecuzione della modifica. Questo perché per automatizzare il processo sarebbe necessario l'inserimento di un ente software che fosse a conoscenza dei legami tra un nodo ed i dati del suo wallet, per ciascun nodo all'interno della Blockchain.

3.2.4 Esecuzione dello Smart Contract

Lo smart contract è la parte principale del prototipo. È responsabile di eseguire le principali operazioni di autenticazione e modifica che viaggiano attraverso il canale di comunicazione tra client e server. La funzione principale è la funzione set che funge da main di tutte le funzioni ausiliarie del contract.

Come prima cosa il contract calcola il root hash partendo dalla proof che gli è stata fornita dal client e lo confronta con quello che ha memorizzato, per controllare che la richiesta provenga da un ente fidato.

```
function set(bytes memory proof) public returns (bool) {
    bytes memory old_hash = compute_hash(proof, false);
    emit rootHashDebug(root_hash);
    if(!compare(old_hash, root_hash, 32)) {
            emit isChanged(false);
            return false;
    }

    returnValue = queue_proof(proof);
    if(returnValue){
        emit nodeNumber(addressToBytes(tx.origin));
        emit isChanged(true);
        return true;
    }
    emit nodeNumber(addressToBytes(tx.origin));
    emit isChanged(false);
    return false;
}
```

Figura 3.5: Funzione "set" dello smart contract

Dopo, il contract esegue la funzione *queue-proof()*, che segue due tipologie diverse di esecuzione, a seconda della quantità di richieste precedentemente ricevute. Tale funzione dispone infatti di un contatore che pone un limite alle richieste che si possono memorizzare prima di poter confermare la modifica al server. Tale contatore è stato impostato su 3, ciò significa che la conferma delle tre modifiche può avvenire solo quando tutte e tre sono state processate. Per ragioni di progettazione del prototipo si è scelta questa metodologia del contatore perché, ancora una volta, per fare diversamente sarebbe stato necessario l'inserimento di una particolare entità che facesse da gestore centrale.

A seconda del valore del contatore la funzione segue due diverse strade esecutive. Se il contatore è minore di 3 significa che la modifica va memorizzata ma non confermata. La funzione scorre quindi la proof elemento dopo elemento, memorizzando di volta in volta gli hash parziali soggetti a modifica in una lista associativa che lega la versione non modificata di un hash parziale alla sua stessa versione dopo l'avvenuta modifica.

Il contatore viene incrementato ed il valore di ritorno falso comunica al contract che le modifiche sono state memorizzate, ma che se ne attendono altre prima di confermare.

```
while(RLPReader.hasNext(subIter)) {     //iterate on the sublist
    RLPReader.RLPItem memory elem = RLPReader.next(subIter);
    if(RLPReader.isList(elem)) {        //key value pair
        RLPReader.RLPItem[] memory p = elem.toList();
        old_hash = bytes32ToBytes(keccak256(concatKV(p[0].toAddress(),p[1].toBytes())));
        old_s=appendHash(old_s,old_hash,length);
        modify_hash = modifyMap[old_hash];
        if(modify_hash.length > 1){ //need to replace previous updates if any
            new_s=appendHash(new_s, modify_hash, length);
        }else{
            new_s=appendHash(new_s,old_hash,length);
        }
        length += 32;
    }
    else{   //else it could be both a " " or an hash value
        if (elem.toBytes()[0]==" ") {   //if it's " " register the updated hash in the map
            old_s=appendHash(old_s, old_hash, length);  //old_hash is still a string without
                                                        //modifications to serve as a key in the map
            new_s=appendHash(new_s, new_hash, length);  //new_hash is the new string to serve as
                                                        //value in the map
            modifyMap[old_hash]=new_hash;   //it saves old_hash --> new_hash in the map
        }
        else{   //it is simply a string from another node. If it is present in the map it is changed
            old_s = appendHash(old_s, elem.toBytes(), length);
            modify_hash = modifyMap[elem.toBytes()];
            if(modify_hash.length > 1){
                new_s=appendHash(new_s, modify_hash, length);
            }else{
                new_s = appendHash(new_s, elem.toBytes(), length);
            }
        }
        length += 32;
    }
}
```

Figura 3.6: Parte 1 della funzione "queue-proof"

Una volta che il contatore ha raggiunto la quantità desiderata di modifiche, nel caso di questo prototipo tre, la seconda parte della funzione queue-proof si occupa di scorrere la proof ricevuta e di modificare gli hash parziali precedentemente memorizzati nella lista associativa. In questo modo la struttura ad albero è rispettata e la costruzione nidificata della proof può tener conto di tutte le modifiche effettuate.

```
while(RLPReader.hasNext(subIter)) { // iterate on the sublist .
    RLPReader.RLPItem memory elem = RLPReader.next(subIter);
    if(RLPReader.isList(elem)) { // key value pair in a node.
        RLPReader.RLPItem[] memory p = elem.toList();
        hash = bytes32ToBytes(keccak256(concatKV(p[0].toAddress(),p[1].toBytes())));
        modify_hash = modifyMap[hash];
        if(modify_hash.length > 1){
            s=appendHash(s, modify_hash, length);   //here the value is replaced as it was stored
                                                    //before
            modifyMap[hash] = "0";
        }else{
            s=appendHash(s,hash,length);    //if not present just copy
        }
        length += 32;
    }
    else { // the aggregate value in a node.
        if (elem.toBytes()[0]==" ") {
            s=appendHash(s, old_hash, length);
        }
        else {
            modify_hash = modifyMap[elem.toBytes()];
            if(modify_hash.length > 1){
                s=appendHash(s, modify_hash, length);   //here the value is replaced as it was
                                                        //stored before
                modifyMap[elem.toBytes()]="0";
            }else{
                s=appendHash(s, elem.toBytes(), length); //if not present just copy
            }
        }
        length += 32;
    }
}
```

Figura 3.7: Parte 2 della funzione "queue-proof"

La compilazione risultante di questo processo è, di fatti, il nuovo root hash che entra in vigore e viene memorizzato dal contract per eseguire ulteriori modifiche in futuro. Il valore di ritorno positivo comunica alla funzione set che la collezione delle modifiche attese terminata e che quindi può essere confermata.

La funzione set invia dunque l'evento di avvenuta modifica che comunica al server di memorizzare i nuovi dati.

3.3 Server

Il programma che viene avviato sui nodi server ha tre compiti fondamentali: instaurare una connessione con il DBTree creando un servizio HTTP, la gestione delle richieste riguardanti i dati memorizzati tramite API REST e la gestione degli eventi comunicati dallo smart contract.

3.3.1 Gestione delle richieste tramite API-Rest

La *API REST* è un sistema che permette di catturare alcune richieste HTTP tramite la porta 8080 e di conseguenza eseguire diverse funzioni a seconda della richiesta. Questo avviene nella funzione *DBTreeController()* che utilizzando il servizio di connessione instaurato precedentemente dal programma, cattura varie richieste http tra cui *"retrieveProof"*, ovvero la richiesta di ottenere la proof relativa ad un determinato nodo, oppure *"updateValue"* ovvero la richiesta di modificare un certo valore relativo ad un determinato nodo.

```
/**
 * This method is used for mapping HTTP GET requests onto specific handler method,
 * which returns proof of the queried key.
 *
 * @param queriedKey the key whose proof is required
 */
@GetMapping( ...value: "/retrieveProof/{queriedKey}")
fun retrieveProof(@PathVariable( value: "queriedKey") queriedKey: String):
        Pair<String?, MutableList<LinkedList<String>>> {
    return service.dbtree.authenticatedQuery(
            Address(queriedKey))
}
```

Figura 3.8: Metodo "retrieveProof" per gestire la richiesta di ottenere la proof da parte del client

```
/**
 * This method is used for mapping HTTP POST requests onto specific handler method,
 * which updates the value of the key passed as a parameter into the DBTree.
 *
 * @param keyToUpdate the key whose value needs to be updated
 */
@PostMapping( ...value: "/updateValue/{keyToUpdate}")
fun updateValue(@PathVariable( value: "keyToUpdate") keyToUpdate: String) {

    val proof = service.dbtree.authenticatedQuery(Address(keyToUpdate))

    if(proof.first?.takeLast( n: 2) == String.format("%02x", 's'.toByte()))
        service.dbtree.update(
                Pair(
                        Address(keyToUpdate),
                        Address(keyToUpdate).addrValue.drop( n: 2) + (String.format("%02x",'n'.toByte()))
                ))
    else
        service.dbtree.update(
                Pair(
                        Address(keyToUpdate),
                        Address(keyToUpdate).addrValue.drop( n: 2) + (String.format("%02x",'s'.toByte()))
                ))
}
```

Figura 3.9: Metodo "updateValue" per gestire la richiesta di modifica del valore da parte del clien"

3.3.2 Gestione degli eventi dello Smart Contract

La seconda funzione fondamentale che deve essere svolta lato server è la gestione degli eventi risultanti dalle esecuzioni dello smart contract.

Tramite il metodo *listen* della funzione DBHandler, che viene chiamato dopo la configurazione del server ed è in attesa durante tutta l'esecuzione del programma, il server si mette in ascolto degli eventi che il contract emette sulla rete.

Questi riguardano la comunicazione dell'indirizzo del client che richiede un update e l'effettiva conferma degli update.

Si noti che la funzione è responsabile di collezionare all'interno di una lista denominata *"transactionTarget"* tutti gli indirizzi dei client che hanno richiesto l'update e che, quando l'update dovrà effettivamente avvenire, questo avvenga su ciascuno degli indirizzi presenti all'interno della lista suddetta.

```
val web3j = Web3j.build(HttpService(urlList[index-1]))
var transactionTarget = mutableListOf<String>()

web3j.ethLogFlowable(debugFilter).subscribe {
    t: Log? ->
    if (t != null) {
        transactionTarget.add(t.data.toString())
        print("\n\nTransaction target: $transactionTarget")
    }
}
web3j.ethLogFlowable(filter).subscribe {
        t: Log? ->
    if (t != null) {
        print("\n\nlogCompareTransaction: "+t.toString()+"\n\n")
        if (t.data.endsWith( char: '1')) {
            for(i in transactionTarget){
                updateClientValue(i)
            }
            transactionTarget.clear()
            updateDBTreeVersion()
        }
    }

}
```

Figura 3.10: Funzione "listen" con due tipi di risposta ai determinati eventi

3.4 Simulazione e test

Di seguito si espone nel dettaglio una simulazione svolta con il prototipo che precedentemente descritto. Lo scopo della simulazione è quello di osservare e verificare il corretto funzionamento del prototipo che rappresenta il prodotto finale.

La simulazione si basa su due passaggi: come prima cosa il sistema dovrà svolgere l'update del dato del nodo 5 per poter eseguire il deploy del contract sulla blockchain e per preparare il sistema. A questo punto il prototipo può svolgere le operazioni in rapida successione, e quindi verrà richiesto un update simultaneo dei dati dei nodi 6 e 7. Per osservare l'avvenuta modifica dei dati si esegue una query sul server verificando l'efficacia delle operazioni svolte. Man mano che la simulazione procede si può controllare il log dello smart contract per verificare i cambiamenti sul root hash.

Inizialmente con il comando *java -jar* si avvia il software per tutti i nodi che fanno da server. Con questo comando si specifica anche la tipologia di server che si avvia e le credenziali. L'output colleziona dei log relativi all'avvio e poi mette la console in stato di *idle*, in attesa che arrivino richieste.

```
13:31:38.276 [main] INFO it.myproject.dbtree.ApplicationKt - Started ApplicationKt in
8.503 seconds (JVM running for 9.54)
```

Figura 3.11: Output della fase idle dell'applicazione server

Quando l'applicazione è avviata e in attesa su tutti i nodi server, dal nodo 5 viene inviata la prima richiesta di update.

```
Proofs match!
Connecting to the blockchain...

Proof formatted according to the DB-tree standard:
[s, [ , 3c27036801dc8a0c78980d7cf97dc4f27d7dd2f7f48dff04cf58bc38353b67ee], [ , 04c1bd1
79cbaad6ea1d8a09c43626b066de304058e5c6352f2bf9674746cd821, 6223143c35d24c582fdb0b29309
79756aee822081f81c44a08c1253f1429c433], 0.0]

Proof encoded in RLP:
f86a73e220a03c27036801dc8a0c78980d7cf97dc4f27d7dd2f7f48dff04cf58bc38353b67eef84320a004
c1bd179cbaad6ea1d8a09c43626b066de304058e5c6352f2bf9674746cd821a06223143c35d24c582fdb0b
2930979756aee822081f81c44a08c1253f1429c43380

Opposition contract DEPLOYED at address: 0x3f217e1fe69d1b188385b761a2b17827616b9bdb

Copy this address for next operations!!
Calling set function on the smart contract...
```

Figura 3.12: Output della richiesta di update del nodo 5

Come specificato precedentemente, il programma comunica la riuscita del controllo di integrità sul server dopo la connessione alla blockchain e ottiene la proof, la quale mostra il valore binario S o N del nodo in questione, susseguito da tutti gli hash parziali degli altri nodi fino alla radice. Infine è presente un numero intero che rappresenta la versione del database, incrementa quando vengono eseguiti update per dimostrare l'avvenuta riuscita delle modifiche.

L'esecuzione termina con il deploy dello smart contract e la chiamata delle sue funzioni. Lo smart contract comunica quindi due eventi al server, uno contenente l'indirizzo del nodo che ha richiesto l'update, in questo caso *"0638e1574728b6d862dd5d3a3e0942c3be47d996"*, l'indirizzo che identifica il nodo 5.


```
13:44:38.352 [pool-2-thread-3] DEBUG org.web3j.protocol.http.HttpService -
13:44:38.352 [pool-2-thread-3] DEBUG org.web3j.protocol.http.HttpService - {"jsonrpc":
"2.0","id":9,"result":[{"address":"0x3f217e1fe69d1b188385b761a2b17827616b9bdb","topics
":["0x9f88864d350d350f5e8ebb7002c8690f04e46e907aa1bc7e8071e198f14eadc4"],"data":"0x000
0000000000000000000000000000000000000000000000000000000000000200000000000000000000000000000
0000000000000000000000000000000000000140638e1574728b6d862dd5d3a3e0942c3be47d9960000000
0000000000000000","blockNumber":"0x3f4","transactionHash":"0x6e568a2e468193bf73323ad9
db0f9c65602b843bb8f1be15f337c80eedc63476","transactionIndex":"0x0","blockHash":"0x5b1e
7d938341773b0a171459d38add1ecd5cf56be9d2ccc29dbc421cf35bcc17","logIndex":"0x1","remove
d":false}]}
```


Figura 3.13: Log lato server della transazione per l'indirizzo del nodo richiedente update

Il secondo evento comunica un valore binario vero o falso che, se vero esegue l'update dei dati, altrimenti non esegue nulla. In questa esecuzione il valore è falso "0" dato che il sistema è in attesa di altre richieste prima di eseguire l'update di conferma.


```
logCompareTransaction: Log{removed=false, logIndex='0x2', transactionIndex='0x0', tran
sactionHash='0x616ab8277eb1d6c9cd469cf6d817133e7e580aa755aecee478b3ec066ef031cc', bloc
kHash='0x4c8a535098aa173ca951de6b7e49ccbca9278a8382f8d7db1ef08c60b40f99a5', blockNumbe
r='0x667', address='0x3f217e1fe69d1b188385b761a2b17827616b9bdb', data='0x0000000000000
0000000000000000000000000000000000000000000000000', type='null', topics=[0xddf9af4de
a5a1e91aedccbe564ef0e3d744f418e8328b062167100036b84f085]}
```


Figura 3.14: Log lato server della transazione per l'esecuzione dell'update

In questa fase è possibile anche visualizzare il root hash attualmente in vigore sullo smart contract. Un hash che, dunque, ancora non tiene conto di quest'ultima modifica richiesta.

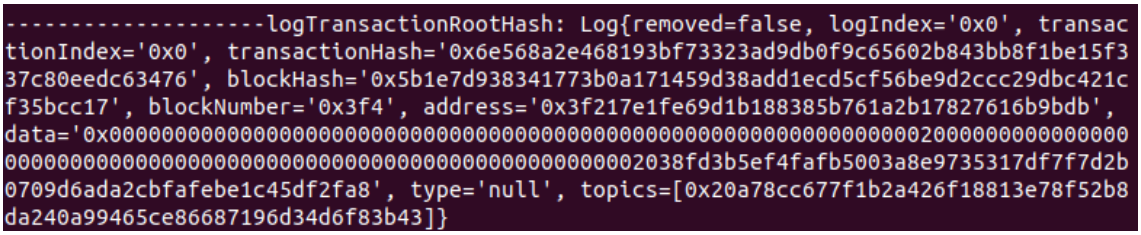

Figura 3.15: Transazione contenente il valore del root hash prima delle modifiche

A questo punto viene eseguito l'update contemporaneamente sui nodi 6 e 7. Ci si aspetta che l'output di ciascuno riporti la proof senza nessun cambiamento, nonostante sia stato richiesto l'update del dato del nodo 5, dato la conferma non è stata data.

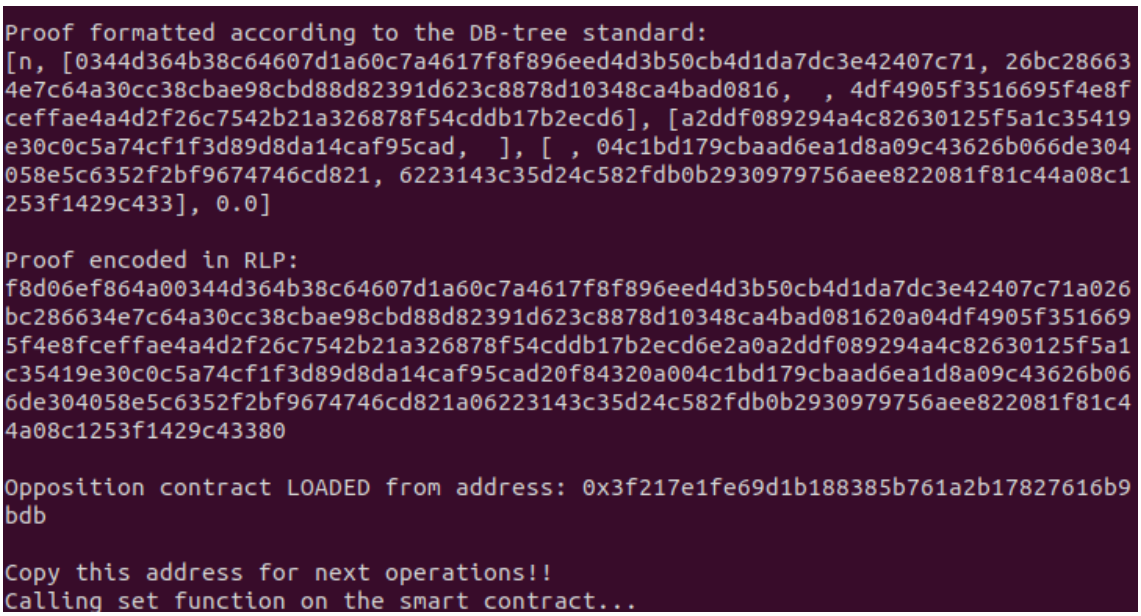

Figura 3.16: Log della fase di update del nodo 6

Da notare che in entrambi i casi è stato eseguito il load dello smart contract. La versione del database, inoltre, rimane 0.0. Il root hash in questa fase non è modificato.

Quando il contract riceve la seconda e la terza richiesta allora, oltre che a comunicare l'indirizzo dei nodi che hanno eseguito la richiesta, emette l'evento di conferma con

```
Proofs match!
Connecting to the blockchain...

Proof formatted according to the DB-tree standard:
[s, [d89f3edbcd4c81d62a5637164dbc6f9e1fd09ab60cb70c9a01a08c7d04ba7dae,  , 622314
3c35d24c582fdb0b2930979756aee822081f81c44a08c1253f1429c433], 0.0]

Proof encoded in RLP:
f84773f843a0d89f3edbcd4c81d62a5637164dbc6f9e1fd09ab60cb70c9a01a08c7d04ba7dae20a0
6223143c35d24c582fdb0b2930979756aee822081f81c44a08c1253f1429c43380

Opposition contract LOADED from address: 0x3f217e1fe69d1b188385b761a2b17827616b9
bdb

Copy this address for next operations!!
Calling set function on the smart contract...
```

Figura 3.17: Log della fase di update del nodo 7

valore 1, segnale per il server di applicare tutte le modifiche memorizzate fino ad ora.

```
logCompareTransaction: Log{removed=false, logIndex='0x2', transactionIndex='0x0', tran
sactionHash='0xda7464baa7059c7136971dc356854288c12dcd07826a8b9cee8117dc56d19516', bloc
kHash='0x6513a0836a477eaf9ca0d05a3e9b7e8617149725c983fe793b7f576d2fb4a4d0', blockNumbe
r='0x679', address='0x3f217e1fe69d1b188385b761a2b17827616b9bdb', data='0x0000000000000
0000000000000000000000000000000000000000000000001', type='null', topics=[0xddf9af4de
a5a1e91aedccbe564ef0e3d744f418e8328b062167100036b84f085]}
```

Figura 3.18: Log lato server della transazione di update con valore 1

Il server, ricevendo questa comunicazione, esegue l'update dei valori che gli erano stati precedentemente specificati ed incrementa il valore della versione. Contemporaneamente lo smart contract aggiorna il valore del suo root hash con quello nuovo, ora in vigore.

Successivamente è possibile controllare anche che il valore e la proof dei nodi che

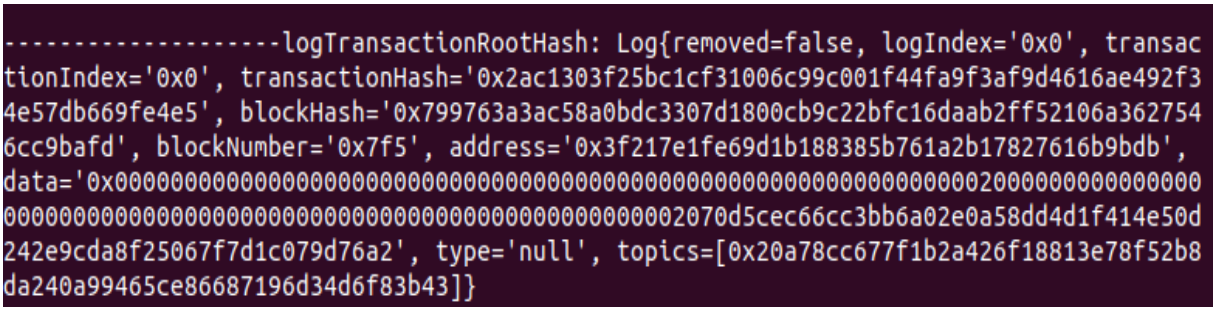

Figura 3.19: Log della transazione contenente il valore del nuovo root hash

hanno eseguito l'update siano effettivamente cambiati.

Figura 3.20: Valore della proof del nodo 6 dopo la modifica

Da questa immagine, ad esempio, si nota come il valore della scelta sia passato da N a S, segno che l'update è avvenuto correttamente. Gli hash parziali che rappresentano gli altri nodi, inoltre, sono anch'essi cambiati, segno che il cambiamento è avvenuto secondo la struttura ad albero. Infine la versione 1.0 fornisce ulteriore conferma dell'avvenuta modifica.

In conclusione si dimostra come eseguendo delle richieste anche in tempistiche molto brevi l'una dall'altra, il sistema sia in grado di eseguirle tutte seguendo il regolamento dettato dalla blockchain e la struttura ad albero, essendo quindi in grado di poter costruire un root hash rappresentante la nuova versione dei dati.

Conclusioni e sviluppi futuri

Nel corso di questa trattazione si è presentato il prototipo di un sistema per la memorizzazione sicura dei dati, basato sulla blockchain. Il sistema in questione memorizza i dati degli utenti in modo completo e ne garantisce la sicurezza tramite un sistema decentralizzato di consenso di modo che qualsiasi accesso non convalidato risulti estraneo al sistema, che quindi è al sicuro da modifiche esterne. Il database diventa dunque una struttura dati autenticata che riceve le istruzioni da uno smart contract, una funzione condivisa dalla comunità di utenti, decentralizzata e quindi inattaccabile. Ciascun client richiede la modifica dei propri dati autenticandosi ed il sistema applica tale modifica seguendo una serie di regole e ottenendo di volta in volta nuove chiavi necessarie ad autenticazioni future.

Un sistema del genere sarebbe in grado di garantire un elevato grado di sicurezza ed una forte scalabilità, adattandosi ad ogni tipo di dato, l'efficacia di questi algoritmi di sicurezza decentralizzata, inoltre, cresce esponenzialmente al crescere del numero di aderenti al sistema, dato che una maggior condivisione significa, in questo caso, una maggior sicurezza.

Bibliografia

[Doc] Web3J Documentation. http://docs.web3j.io/latest/.

[Eth] Ethereum. https://ethereum.org/en/what-is-ethereum/.

[fD] Docker: Empowering App Development for Developers. https://www.docker.com/.

[Fou] MariaDB Foundation. https://mariadb.org/.

[Git] GitHub7Nodes. https://github.com/ConsenSys/quorum-examples/tree/master/examples/7nodes.

[GoQ] GoQuorumDocumentation. https://docs.goquorum.consensys.net/en/stable.

[pdoM] Registro pubblico delle opposizioni (MdSE). http://www.registrodelleopposizioni.it/.

[wmaosd] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/.