Tesi laurea triennale

Daniel Arduini

Luglio 2023

Indice

1 Introduzione

La blockchain è una tecnologia che sta riscuotendo molto successo nell'ultimo periodo grazie ad alcune sue caratteristiche come la decentralizzazione e la sicurezza. Questo elaborato di tesi esplora in modo dettagliato la blockchain fornendo anche un esempio di implementazione reale.

Nella prima parte verrà effettuata un'analisi della tecnologia spiegando tutti i concetti chiave che la contraddistinguno. Verrà discusso il concetto di blocco, di decentralizzazione e di sicurezza, ma verranno anche analizzati argomenti più specifici come gli smart contract.
L'obiettivo è fornire una visione sufficientemente ampia della tecnologia in modo da poter continuare l'analisi tramite un esempio pratico di implementazione.

La seconda parte della tesi si concentrerà sull'applicazione pratica della blockchain attraverso l'analisi di un progetto concreto. Sarà presentato un caso d'uso specifico in cui la blockchain è stata implementata in un contesto aziendale per garantire la sicurezza e la tracciabilità dei dati.
Questa sezione fornirà un'opportunità per valutare i vantaggi e gli svantaggi dell'implementazione di questa tecnologia in un contesto reale, affrontando considerazioni pratiche come i costi di esecuzione.

In definitiva, questa tesi mira a fornire una panoramica completa sulla blockchain, dal suo fondamento teorico alla sua applicazione concreta. Il risultato sarà un'analisi completa che consentirà di comprendere appieno l'impatto di questa tecnologia e di valutarne le possibilità di implementazione in diversi scenari.

2 Blockchain

2.1 Cos'è?

Una blockchain è una struttura pubblica, condivisa e immutabile che permette di salvare dati all'interno di blocchi. I blocchi sono collegati tra loro in modo tale che ogni blocco verifichi il successivo e per questo modo se un nodo della catena si rompe, allora tutta la catena viene invalidata.

2.2 Parole chiave

Si usa parola **blocco** per indicare un insieme di informazioni che vengono salvate all'interno della blockchain. Infatti i dati non sono inseriti singolarmente come in un database tradizionale, ma vengono raggruppati in blocchi di dimensione predefinita e una volta che viene soddisfatta la dimensione vengono salvati.

Una **catena** è l'insieme dei blocchi collegati tra loro. Ogni blocco è collegato al blocco precedente tramite un codice hash creando in questo modo un legame inviolabile, dato che la modifica di un blocco porterebbe al cambiamento del codice hash e quindi all'invalidazione di tutti i collegamenti successivi.

Un **hash code** è un codice che viene generato da una funzione matematica che a partire da una stringa di lunghezza variabile, ne genera una di lunghezza fissa che identifica univocamente la stringa di partenza. Può essere visto come se fosse l'impronta digitale dell'elemento di partenza in quanto è unica e rappresenta in modo diretto l'input iniziale. Il codice hash risulta differente anche se un solo carattere della stringa di partenza viene modificato, quindi è facile intuire che se un blocco cambia hash code, allora sicuramente è stato mutato.

Ogni computer all'interno della rete è chiamato **nodo** e tiene in memoria una copia della rete completa. Questo permette alla tecnologia di essere distribuita e decentralizzata, in quanto non esiste un server centrale.

Le operazioni che vengono svolte all'interno di questa struttura sono chiamate **transazioni** e sono salvate all'interno dei blocchi.

Ora che abbiamo definito le parole chiave della blockchain possiamo andare ad analizzare in dettaglio il funzionamento di questa tecnologia.

2.3 Come funziona?

In questo capitolo andremo ad analizzare il funzionamento di una blockchain, spiegando i vari passaggi che la contraddistinguono. Prenderemo come riferimento la blockchain Ethereum in quanto è quella che verrà utilizzata nell'esempio pratico, tuttavia la maggior parte dei concetti è applicabile a tutte le altre blockchain.

2.3.1 Transazione

Una transazione, come detto prima, è un'operazione svolta da un utente all'interno della blockchain.

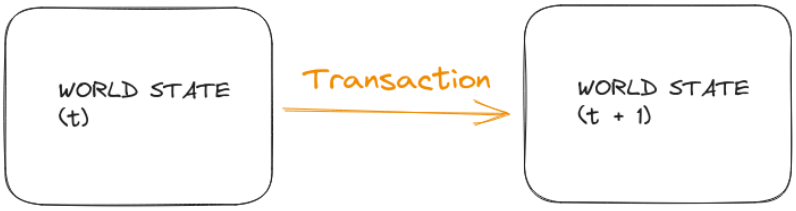

Figura 1: Cambio di stato con transazione

Le transazioni che modificano lo stato della struttura devono essere notificate a tutti gli utenti che vi partecipano. Per fare ciò si utilizza un sistema di notifiche chiamato **broadcast** che permette di inviare un messaggio a tutti i nodi della rete, in questo modo ogni nodo può mandare una richiesta di esecuzione di una transazione a tutti e in seguito un validatore la eseguirà e propagherà il cambiamento di stato su tutta la rete.
Le transazioni una volta eseguite vengono salvate all'interno di un blocco che verrà poi aggiunto alla catena.

2.3.2 Blocco

La blockchain è per certi versi simile a un database, ossia il suo scopo è salvare dati di vario tipo, ma ciò che li differenzia è il modo in cui i dati vengono salvati.
Come detto prima la blockchain è una serie di blocchi che contengono informazioni, in particolare vengono salvati tre dati:

- **Dati**: sono i dati che devono essere salvati. La tipologia di dati può variare in base al tipo di blockchain utilizzata.
- **Hash del blocco precedente**: è l'hash del blocco precedente e serve per collegare i blocchi tra loro. Se il blocco precedente viene modificato e il suo hash code cambia, allora non sarà più uguale a quello salvato nel blocco successivo e quindi la catena viene invalidata.
- **Hash**: è un codice univoco che identifica il blocco e viene calcolato in base ai dati contenuti nel blocco stesso. Per poterlo calcolare si una appunto una funzione di hashing che prende in input i dati e restituisce l'hash.

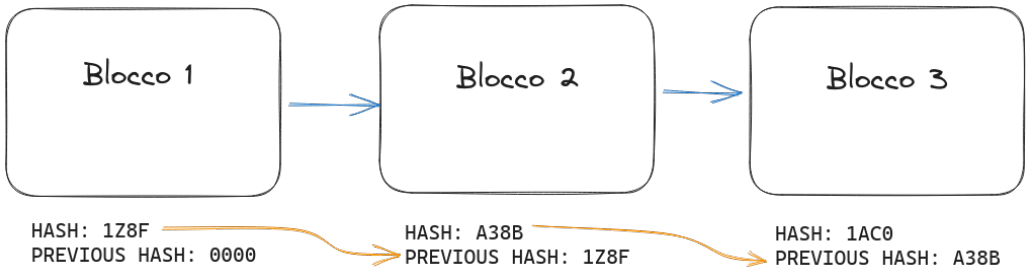

Figura 2: Schema dei blocchi Blockchain

Dall'immagine precedente 2 si può notare che ogni blocco è collegato al blocco prima tramite l'hash del blocco precedente. Il primo blocco ha come hash precedente *0000* perchè essendo il blocco iniziale non esistono blocchi prima di lui.

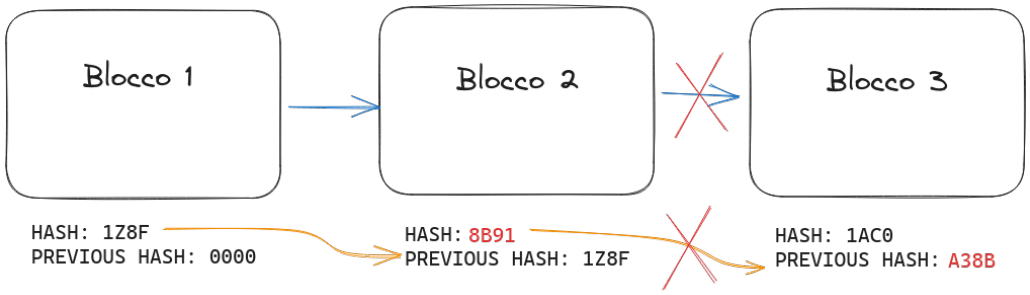

Figura 3: Catena invalida

In questo caso 3 si può notare che il blocco 2 è stato modificato e quindi il suo hash code è cambiato. Questo ha portato alla invalidazione della catena in quanto il blocco 3 non ha più come hash del blocco precedente quello del blocco 2.

2.3.3 Account

Un account è un'entità all'interno della blockchain che ha un saldo in ETH e che ha la possibilità di eseguire transazioni. Esistono due tipi di account: **Externally-Owned-Account (EOA)** che è un account che può essere gestito da chiunque abbia la chiavi private e **Contract Account** che è uno smart contract all'interno della rete, può essere gestito solo dal codice.

Entrambi i tipi di account possono eseguire operazioni di scambio moneta in valuta (ETH) e possono interagire con gli smart contract, ma ci sono delle differenze tra i due:

- **EOA:** creare un account non ha un costo e le operazioni che si possono eseguire sono solo transazioni e scambi di ETH. Sono composti da una coppia di chiavi criptografiche, una pubblica e una privata, che permettono di controllare le attività dell'account.
- **Contract Account:** la creazione dell'account ha un costo in quanto c'è bisogno di caricare il codice dello smart contract all'interno della blockchain. Possono eseguire

transazioni solo in risposta a richieste ricevute e le azioni eseguite sono quelle definite all'interno del codice. Non hanno chiavi come gli account EOA

Ogni account ha una serie di elementi al suo interno:

- **nonce:** valore che conta il numero di transazioni eseguite per un EOA, oppure il numero di contratti creati per un contract account. Solo una transazione con un certo nonce può essere eseguito da un account, questo aiuta la protezione da attacchi di ripetizione.
- **balance** il numero di ETH che l'account possiede
- **codeHash:** il codice dell'account che viene anche salvato all'interno della EVM. Non può essere in nessun modo modificato.
- **storageRoot** l'hash code che punta al nodo contenente lo storage per l'account.

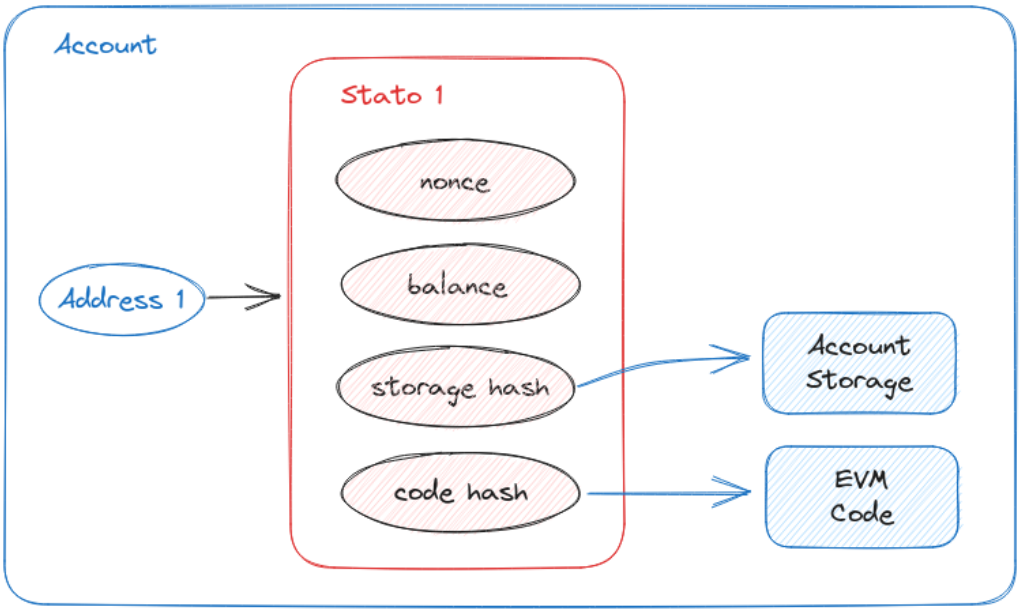

Figura 4: Account

2.3.4 Ethereum Virtual Machine (EVM)

Nella blockchain di Ethereum c'è un solo computer che ha uno stato a cui ogni partecipante della rete deve fare riferimento. Ogni partecipante ne tiene una copia e ha inoltre la possibilità di richiedere al computer principale di eseguire delle operazioni. Quando questa richiesta viene avanzata, altri nodi della rete si prendono in carico l'operazione e la verificano, validano ed eseguono notificando poi agli altri ciò che è avvenuto. Ogni operazione porta al cambiamento dello stato della EVM, e quando ciò avviene, il nuovo stato viene aggiornato su tutti i nodi della rete.
Questo modello di **macchina a stati distribuita** permette l'esistenza degli smart contract, ossia dei programmi scritti dagli utenti che possono essere eseguiti all'interno della blockchain. Questa caratteristica differenzia la blockchain di Ethereum da quella di Bit-

coin in quanto quest'ultima non permette l'esecuzione di programmi, limitando di molto le possibilità di utilizzo della rete.

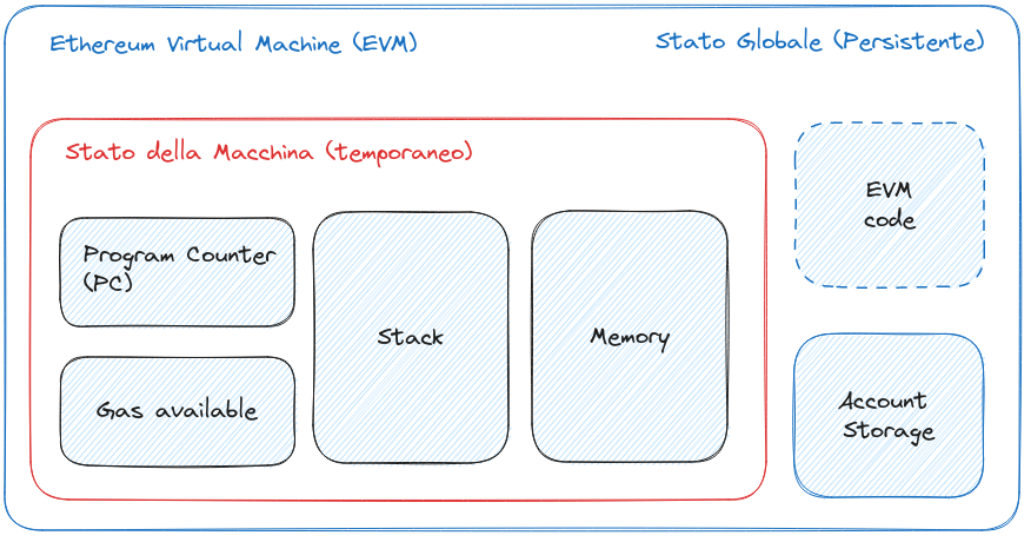

Figura 5: Ethereum Virtual Machine

Lo schema 5 mostra che la EVM ha uno stato globale che è immutabile ed è quello su cui tutti devono fare riferimento, ma poi ogni singola macchina ha il suo stato locale che può essere modificato e poi in caso notificato alle altre macchine.

Il cambio di stato si comporta come una funzione matematica, ossia dato un input restituisce un output in modo deterministico. In modo più formale possiamo descrivere la funzione come segue:

$$S_{t+1} = \Upsilon(S_t, T) \tag{1}$$

Dato uno stato S_t e una transazione T viene restituito un nuovo stato $\S_{t+1}$.

2.3.5 Gas fee

Gas si riferisce all'unità che misura la quantità di sforzo computazionale richiesto per eseguire operazioni specifiche sulla rete Ethereum.
Poiché ogni transazione di Ethereum richiede risorse computazionali per essere eseguita, tali risorse devono essere pagate per garantire che Ethereum non sia vulnerabile allo spam e non si blocchi in loop computazionali infiniti. Il pagamento per la computazione avviene sotto forma di tassa sul gas.
Gas fee è la quantità di gas necessaria per eseguire una transazione moltiplicata per il prezzo di un singolo gas. Il prezzo del gas è espresso in **gwei**, che è un sottomultiplo di un ether (1 gwei = 10^{-9} ETH).

Come vengono calcolate le gas fee?
È possibile impostare la quantità di gas che si è disposti a pagare quando si invia una transazione. Offrendo una certa quantità di gas, si fa un'offerta per includere la transazione nel blocco successivo. Se si offre troppo poco, è meno probabile che i validatori scelgano la transazione per l'inclusione, il che significa che la transazione potrebbe essere eseguita in

ritardo o non essere eseguita affatto. Se si offre troppo, c'è il rischio di sprecare un po' di ETH.

Il costo totale in gas che si deve pagare è diviso in due parti: *base fee* e *priority fee (tip)*.
La **base fee** è la quantità richiesta dal protocollo che bisogna pagare per poter ritenere l'operazione valida. La **priority fee** è la quantità di gas che si offre in più per far eseguire la transazione prima di altre.
Per esempio, se si vuole inviare una transazione che richiede 21.000 gas, la *base fee* è di 10gwei e si offre 1 gwei come *priority fee*, allora il costo totale sarà:

$$\text{units of gas used} * (\text{base fee} + \text{priority fee})$$

$$21.000 * (10 + 1) = 0.000021 ETH \tag{2}$$

2.3.6 Ether

Abbiamo visto che le gas fee vengono pagate in ETH, ma cos'è l'ETH?
Ether (o ETH) è la criptovaluta nativa della blockchain di Ethereum, ciò significa che gli scambi di moneta all'interno di questa struttura avvengono con questa valuta.
Gli Ether vengono creati come ricompensa per l'azione di proposta e validazione di un nuovo blocco. L'importo emesso dipende dal numero di validatori e dalla quantità di Ether che hanno depositato nei loro portafogli digitali.
Di solito 1/8 del valore viene destinato a chi ha proposto il blocco e il resto viene suddiviso tra i validatori.
Ovviamente, per evitare il deprezzamento dovuto a inflazione, alcuni Ether devono anche essere eliminati, per questo motivo quando un utente effettua una transazione, la *base fee*, il cui valore viene impostato dalla rete in base alla domanda, viene eliminata.

2.3.7 Meccanismo del consenso

Il termine meccanismo di consenso si riferisce all'intera serie di protocolli, incentivi e idee che consentono a una rete di nodi di concordare lo stato di una blockchain.
Ci sono due tipi di meccanismi di consenso:

- **Proof of Work (PoW)**
- **Proof of Stake (PoS)**

Proof of Work
Questo meccanismo viene utilizzato all'interno della blockchain di Bitcoin e veniva utilizzato anche in quella di Ethereum prima del passaggio a Proof of Stake. Si basa sul concetto di **mining**, ossia il lavoro svolto per aggiungere un blocco valido all'interno della catena3

- **Nuovi blocchi:** I minatori competono per creare nuovi blocchi pieni di transazioni elaborate. Il vincitore condivide il nuovo blocco con il resto della rete e guadagna alcune criptovalute appena coniate (ad esempio BTC nella blockchain Bitcoin). La gara è vinta dal computer che riesce a risolvere più velocemente un puzzle matematico e la soluzione di questo puzzle è il lavoro di "proof-of-work". Questo produce il collegamento crittografico tra il blocco corrente e quello precedente.

- **Sicurezza:** La rete è sicura grazie al fatto che servono il 51% della potenza di calcolo totale della rete per poter alterare la catena, una quantità di risorse ed energia talmente grande che rende assolutamente nulli i possibili benefici di un attacco.

Proof of Stake
Meccanismo utilizzato in questo momento da Ethereum, in cui i validatori sono solo nodi che hanno depositato una certa quantità di ETH (in questo momento 32 ETH).

- **Nuovi blocchi:** I validatori vengono scelti in modo casuale basandosi sulla quantità di ETH che hanno depositato. Una volta scelti creano un nuovo blocco e lo condividono con il resto della rete, ricevendo una ricompensa per il loro lavoro oppure una sanzione economica in caso di comportamento scorretto, come ad esempio la distruzione di una parte o la totalità degli ETH salvati nel portafoglio.
- **Sicurezza:** La rete è sicura perché nel caso un attaccante volesse commettere un'azione malevole, metterebbe a rischio gran parte dei suoi ETH e quindi molto spesso non è conveniente.

2.3.8 Smart Contract

Uno smart contract è un programma che gira sulla blockchain di Ethereum. È una raccolta di *codice* (le sue funzioni) e di *dati* (il suo stato) che risiede ad un indirizzo specifico sulla blockchain di Ethereum.
I contratti intelligenti sono un tipo di conto Ethereum, ciò significa che hanno un saldo e possono essere oggetto di transazioni, tuttavia non sono controllati da un utente, ma vengono distribuiti sulla rete ed eseguiti come programmato. Gli account utente possono quindi interagire con uno smart contract inviando transazioni che eseguono una funzione definita nello in esso. Gli smart contract possono definire regole, come un normale contratto, e applicarle automaticamente tramite il codice e le interazioni con essi sono irreversibili.
Uno smart contract può essere scritto da chiunque tramite i linguaggi di programmazione **Solidity** o **Vyper** e può essere eseguito da chiunque abbia accesso alla blockchain di Ethereum.
La loro principale differenza rispetto a un programma sviluppato per esempio in Java è che lo smart contract non può accedere a dati che non siano presenti nella blockchain, quindi non può leggere dati dallo standard input o scrivere dati sullo standard output, non può effettuare chiamate http o accedere a database esterni. Questa scelta è stata fatta perché tutti i nodi devono poter condividere e validare le informazioni, quindi devono essere in grado di eseguire lo stesso codice e ottenere lo stesso risultato.

Durante il proseguo della tesi verrà utilizzato come linguaggio *Solidity* per la scrittura degli smart contract. Di seguito un esempio di codice:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleStorage {
    string public text;

    function set(string calldata _text) external {
        text = _text;
    }

    function get() external view returns (string memory) {
        return text;
    }
}
```

Per il momento non ci interessa sapere a cosa serve ogni componente di questo codice, ci basta capire che è un contratto che permette di salvare una stringa e di leggerla.

2.4 Approfondimento di Solidity

Per capire meglio i capitoli successivi di questa tesi, è necessario fare un approfondimento sul linguaggio Solidity, in particolare sulla gestione della memoria.

2.4.1 Memory e Storage

Quando vengono dichiarate delle variabili all'interno di uno smart contract, si può decidere dove salvare i dati, in *memory* o in *storage*, questo perchè le operazioni in blockchain hanno sempre un costo e spesso elevato, qundi questa possibilità aiuta gli sviluppatori a gestire la parte economica relativa alla scrittura e lettura dei dati.

Storage
I dati persistenti sono rappresentati dalle variabili di stato e questi dati vengono salvati direttamente sulla blockchain in modo permanente. È neccessario dichiarare il tipo della variabile in quanto lo smart contract ha bisogno di sapere quanto spazio allocare in fase di compilazione.

```solidity
contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

Memory
Sono i dati temporanei che esistono per la durata dell'esecuzione di una funzione all'interno dello smart contract. Dato che non vengono salvati in modo permanente, sono molto più economici da utilizzare rispetto a quelli di stato.

```solidity
contract SumCalculator {
    function sum(uint256 _num1, uint256 _num2) public returns (uint256) {
        uint256 memory result = _num1 + _num2;
        return result;
    }
}
```

Come si può vedere dal codice, la variabile *result* è stata dichiarata come variabile di memoria tramite la keyword *memory* e quindi il valore verrà cancellato non appena la funzione

finirà l'esecuzione. La parola *memory* in quseto caso poteva anche essere evitata in quanto implicita, ma è stata inserita per chiarezza.

2.4.2 Salvataggio dati di grandezza dinamica (Storage)

Una domanda che può sorgere spontanea è: *Come si fa a salvare dati di grandezza dinamica in modo permanente?*
Se per esempio instanziamo un *mapping* (l'equivalente di un hashmap) come quello mostrato di seguito, non possiamo sapere a priori quanti elementi saranno inseriti e quindi non possiamo sapere quanto spazio allocare in fase di compilazione.

```
1  pragma solidity ^0.8.0;
2
3  contract MappingExample {
4      mapping(address => uint256) public balances;
5
6      function setBalance(uint256 amount) public {
7          balances[msg.sender] = amount;
8      }
9
10     function getBalance() public view returns (uint256) {
11         return balances[msg.sender];
12     }
13 }
```

Ogni smart contract ha accesso alla memoria permanente che può essere pensata come se fosse un array in cui ogni elemento è di lunghezza **32 byte** e l'array ha lunghezza $\mathbf{2^{256}}$. Nonostante questo modello mentale ci aiuti a capire la grandezza della memoria, dobbiamo ricordare che in realtà la memoria è sparsa e popolata in modo casuale e il metodo di accesso utilizza una struttura a **chiave/valore**. Tutte le celle non utilizzate sono riempite con uno zero, che a differenza di quello che potremmo pensare, non occupa nessuno spazio e non serve allocare memoria per esso grazie alla struttura dati utilizzata che si chiama **Merkle Patricia Trie**.
Ovviamente, per quanto possa sembrare che con uno smart contract si possa salvare una quantità di dati praticamente infinita, bisogna ricordare che l'aggiunta di ogni dato ha un costo e questo disincentiva all'utilizzo massivo di questo metodo di storage.

2.5 Applicazioni nel mondo reale

La blockchain è una tecnologia che ha preso molto piede negli ultimi anni e un esempio pratico di questa espansione lo troviamo nel mondo dell'automotive con Alfa Romeo. Infatti con il lancio della Alfa Romeo Tonale è stata introdotto un servizio basato sulla blockchain che permette di registrare tutti i dati del veicolo per poi creare un certificato che può essere utilizzato come garanzia dello stato generale dell'automobile.
Questa applicazione dimostra tutti i punti di forza di questa tecnologia, ossia il fatto che i dati non possano essere modificati e che siano accessibili da tutti rende il dato salvato nella rete un dato di valore e di fiducia. Inoltre, in questo caso specifico vengono anche utilizzati gli **NFT** che sono dei certificati digitali sempre basati sulla blockchain che servono appunto come atto di garanzia rispetto ad un bene, sia fisico che digitale.

3 Stage

3.1 Analisi dei requisiti

L'azienda in cui ho svolto l'attività di stage universitario opera principalmente nel settore assicurativo producendo gestionali per semplificare l'amministrazione degli utenti, delle scadenze delle polizze, degli avvisi di pagamento, della rateizzazione...
In particolare si cerca di semplificare tutto il procedimento che viene svolto dai vari broker assicurativi durante il procedimento di ricerca e stipula di una polizza, dalla realizzazione di preventivi basati sulle richieste specifiche del cliente fino alla stipula del contratto. Tra le varie funzionalità che offrono i loro prodotti c'è anche la possibilità di salvare i documenti dei clienti, in particolare i contratti firmati direttamente all'interno del programma, Ovviamente questi documenti sono salvati in modo sicuro e protetto da accessi non autorizzati e per essi deve essere garantita l'integrità e l'autenticità.
La tecnica che viene utilizzata per garantire l'integrità è quella dell'hashing, in particolare viene calcolato l'hash di alcuni meta-dati del file e viene poi salvato all'interno del database, in questo modo è possibile verificare che il file non sia stato modificato in alcun modo e sia quindi integro.
Per quanto riguarda l'autenticità invece viene utilizzata una firma digitale che viene apposta sul file, in questo modo è possibile verificare che il file sia stato firmato da una persona autorizzata e che quindi sia autentico.

3.1.1 Problemi da risolvere

Il sistema utilizzato dall'azienda è abbastanza solido e viene utilizzato da anni, tuttavia si è cercato un modo per migliorarlo e renderlo più sicuro e affidabile.
Tra le varie problematiche che sono emerse c'è quella relativa al salvataggio dell'hash-code all'interno del database, in particolare se un utente malintenzionato riuscisse ad accedere al database potrebbe modificare l'hash-code e quindi invalidare il file. Inoltre se l'utente riuscisse ad accedere al file potrebbe modificarlo e poi aggiornare l'hash-code nel database in modo da rendere il file integro.
Serviva trovare un modo per garantire in modo trasparente al cliente che l'hash-code salvato non potesse essere in nessun modo modificato.

3.1.2 Soluzioni proposte

Sono state proposte varie soluzioni tra cui quella di salvare in molteplici database l'hash-code in modo da rendere più difficile la modifica, oppure quella di salvare l'hash-code in un database esterno ad esempio online, ma tutte queste soluzioni non risolvevano il problema di base, ovvero che l'hash-code poteva essere modificato.
Infatti queste proposte non andavano a cambiare l'idea di fondo, ossia l'utilizzo di un database per salvare il codice, ma cercavano solamente di aumentare il coefficiente di difficoltà per un utente malintenzionato, ma si portavano dietro tutti i problemi che c'erano prima.
Quindi dopo varie discussioni è stata avanzata l'idea di utilizzare la tecnologia blockchain per risolvere il problema.

3.2 Soluzione: Blockchain

3.2.1 Introduzione

La blockchain è una tecnologia che permette di salvare dati in modo sicuro e affidabile, in particolare i dati vengono salvati in blocchi che vengono poi concatenati tra loro in modo da formare una catena, da qui il nome blockchain.
L'integrità dei dati è garantita dal fatto che ogni blocco contiene l'hash-code del blocco precedente, in questo modo se un utente malintenzionato volesse modificare un blocco dovrebbe modificare anche tutti i blocchi successivi, rendendo praticamente impossibile la modifica. Inoltre i blocchi vengono salvati in modo distribuito, in questo modo non c'è un unico punto di accesso ai dati, ma sono distribuiti in modo che tutti gli utenti possano accedervi. Ogni utente ha una copia della blockchain e può verificare che i dati siano corretti e che non siano stati modificati e questo garantisce la trasparenza del sistema.

3.2.2 Quali dati salvare

La blockchain è una tecnologia molto potente e versatile, ma non è adatta a qualsiasi tipo di dato, infatti è stata progettata per salvare transazioni di denaro e non per salvare file.
Quindi, l'opzione di salvare l'intero documento all'interno della blockchain è stata scartata, in quanto sarebbe stato troppo dispendioso in termini di risorse e non sarebbe stato possibile salvare grandi moli di dati. Inoltre sarebbe stato anche uno spreco di risorse dato che i documenti vengono già salvati in modo sicuro all'interno del database dei clienti.
Quindi si è deciso di salvare solamente l'hash-code del documento all'interno della blockchain in modo da ridurre l'utilizzo di risorse e allo stesso tempo garantire l'integrità del file. Quindi il processo utilizzato in precedenza viene preservato, ma serviva solamente cambiare il modo in cui viene salvato il codice di verifica.

3.2.3 Come interagire con la blockchain

La tecnologia della blockchain è sembrata fin da subito una buona soluzione, ora serviva trovare un modo per integrarla con il sistema già esistente in modo da non alterare l'utilizzo del programma da parte degli utenti finali. Per implementare l'invio dei dati lato applicazione bastava creare un nuovo processo come quelli che erano già presenti che periodicamente inviasse i nuovi dati. Quindi alla fine si trattava solamente di adattare il codice già esistente che salvava sul database. Tuttavia serviva capire come creare un punto d'accesso dal lato blockchain a cui poter inviare le richieste.

3.2.4 Smart Contract

Alcune blockchain come Ethereum permettono di creare degli smart contracts, ovvero dei programmi che vengono eseguiti all'interno della blockchain e che possono interagire con essa.
In realtà gli smart contracts non sono dei veri e propri programmi in quanto non possono interagire con l'input dell'utente, non possono fare delle chiamate http e non possono accedere a risorse esterne, ma possono solamente eseguire una lista di operazioni descritta dal programmatore. In sostanza sono delle macchine a stati finiti che vengono eseguite all'interno della blockchain e con la quale possono interagire direttamente.
La caratteristica di questa tecnologia è che gli smart contracts sono dei contratti senza intermediari che vengono eseguiti in modo automatico e senza nessun supervisore umano,

quindi potenzialmente potrebbero essere utilizzate per stipulare, ad esempio, delle polizze assicurative totalmente gestite da sistemi informatici. Nel nostro caso bastava questa caratteristica non è stata necessaria in quanto dovevamo solo salvare dei dati. Le interfacce che sono state create sono le seguenti:

- **saveHashCode(...)**: per inserire un nuovo hash-code all'interno della blockchain.
- **getHashCode(...)**: per ottenere l'hash-code di un file.

3.3 Struttura del progetto

Il gestionale su cui ho dovuto lavorare si divideva principalmente in due parti logiche separate:

- **Parte grafica e funzionale**: è la parte che comprende il front-end e il back-end, si occupa di fornire l'interfaccia grafica all'utente e di gestire le richieste che arrivano dal client.
- **Parte dei processi**: è la parte che si occupa di eseguire periodicamente una serie di processi, principalmente di invio notifiche o di sincronizzazione dei dati.

Il mio compito è stato quello di modificare la parte dei processi in modo da aggiungere il processo che si occupa di inviare i dati alla blockchain.
Il resto del codice non aveva bisogno di essere modificato in quanto il tutto era già utilizzato per mandare i documenti e gli hash-code al database, quindi abbiamo solo dovuto modificare il processo che salva il codice di verifica.

3.3.1 Analisi struttura del codice

Dovendo lavorare solo sulla parte dei processi, ho dovuto analizzare il codice per capire come funzionava e come potevo integrare la blockchain.
Il progetto sostanzialmente contiene una serie di classi che contengono la logica che deve essere eseguita dal processo e poi una libreria che si occupa della configurazione e dell'esecuzione dei processi.
Ovviamente il progetto contiene anche delle utility per poter interagire tramite chiamate http e per poter eseguire operazioni sul database.

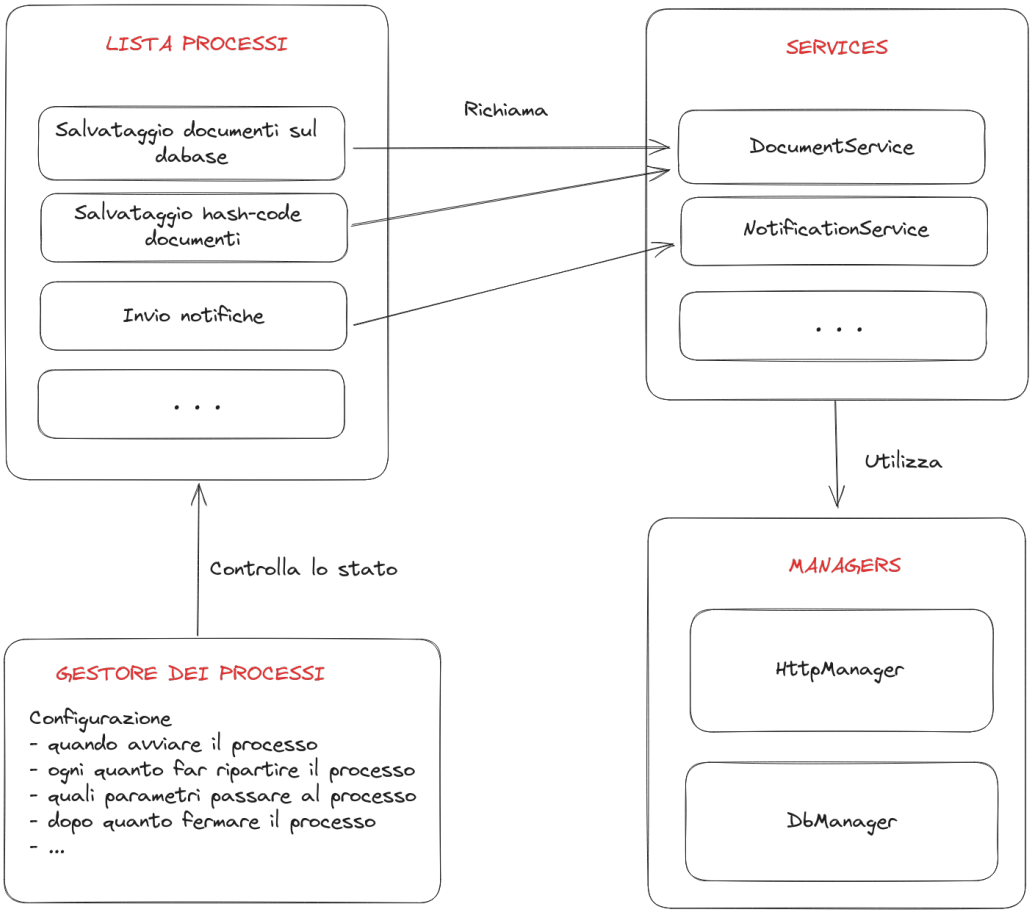

Figura 6: Struttura del progetto di gestione dei processi

Come si può vedere dalla figura 6 il progetto ha quattro parti fondamentali:

- **Gestore dei processi**: è la libreria che si occupa di gestire i processi, in particolare si occupa di leggere la configurazione e di eseguire i processi. La configurazione è un file json che contiene informazioni riguardo al nome del processo, i dati da passare, quando eseguire e ogni quanto riavviare il processo ...
- **Processi**: sono le classi che contengono la logica che deve essere eseguita dal processo, in particolare ogni classe deve implementare l'interfaccia *IProcess* che contiene il metodo *execute()* che viene eseguito dal gestore dei processi. Il processo non contiene effettivamente il codice che implementa le funzioni che devono essere eseguite, ma richiama le interfacce fornite dal Service di riferimento.
- **Services**: sono le classi che vengono richiamate dal processo e che contengono le interfacce di comunicazione tra il processo e il manager. Queste classi esistono per nascondere le implementazioni del manager e per racchiudere più istruzioni in un unica chiamata. Per esempio un interfaccia può gestire l'intero salvataggio del documento, ma al suo interno richiama due funzioni del manager, una per il salvataggio dei dati del documento e una per il salvataggio dell'hash-code

- **Magagers**: sono le classi che si occupano di interagire direttamente con dei servizi tramite chiamate http, o di eseguire operazioni sul database, oppure di comunicare con altri servizi.

3.3.2 Analisi funzionamento dei processi

La maggior parte dei processi implementati dal software funzionano tramite la logica di una coda di richieste (FIFO).

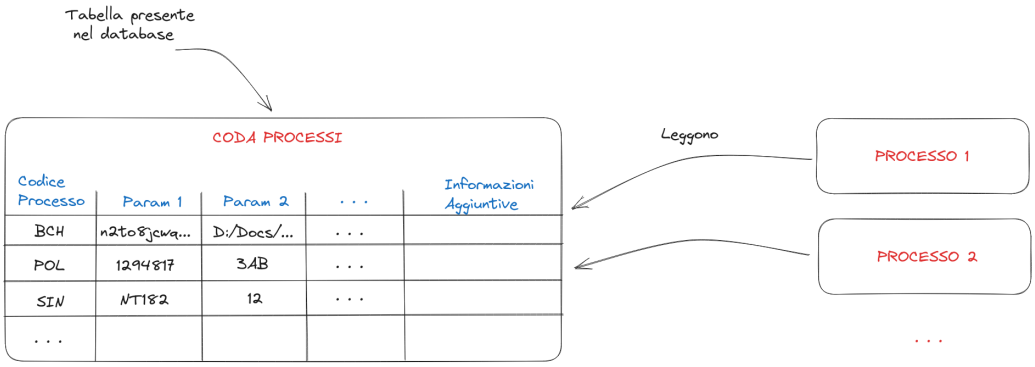

Figura 7: Tabella contenente la lista dei processi

La figura 7 mostra lo schema di funzionamento della coda dei processi.
Sostanzialmente esiste una tabella nel database che su ogni record contiene dei dati che possono essere letti ed interpretati dai singoli processi e che ne determinano il funzionamento. Ogni record contiene una serie di dati obbligatori e altri dati che invece sono opzionali. I dati obbligatori sono:

- **Codice Processo**: è un codice che identifica il processo che deve essere eseguito. Questo codice viene utilizzato per capire quale classe deve essere eseguita dal gestore dei processi. Viene scelto arbitrariamente dal programmatore e salvato in un file di configurazione all'interno del codice del programma.
- **Parametri**: una serie di colonne che sono adibite al passaggio di parametri al processo. Questi parametri possono essere utilizzati per passare dati che sono necessari al processo per poter funzionare. Un esempio di parametro può essere il nome del file che deve essere salvato o il nome della cartella in cui deve essere salvato.
- **Dati utente**: una serie di dati che identificano l'utente che ha richiesto l'esecuzione del processo. Questi dati sono utili in caso di problemi o di errori, in modo da poter risalire all'utente che ha fatto partire la richiesta e poterlo contattare.
- **Dati aggiuntivi**: sono dati che come quelli dell'utente servono in caso di problemi. Possono essere dati aggiuntivi le informazioni riguardo alla data di inserimento e di esecuzione del processo, il nome della macchina che ha aggiunto e quella che ha eseguito il processo ...

La tabella viene popolata automaticamente da altri processi oppure vengono inseriti nuovi record in base alle operazioni eseguite dall'utente. Per esempio se un utente richiede il salvataggio di un nuovo documento tramite l'interfaccia grafica, allora viene inserito dal

programma una nuova richiesta nella tabella con i dati necessari a far funzionare il processo di conservazione.

3.4 Implementazione

3.4.1 Processo BCH

Per sviluppare questo progetto ho dovuto per prima cosa creare un mio processo e il relativo codice-processo nel file di configurazione. Al processo è stato assegnato il codice BCH (che identifica il salvataggio su blockchain) e il processo è stato chiamato *BlockchainConservationProcess*.
Il processo non ha nulla di particolare al suo interno, semplicemente va a leggere i dati dal database e quando trova nella coda dei processi il suo codice allora estrae i parametri dal record sul database e richiama le funzioni per eseguire il salvataggio su blockchain.
Al processo vengono passati i seguenti valori:

- **Hash-Code**: il codice viene generato da un algoritmo già utilizzato all'interno del programma che estrae alcuni dati dalle informazioni del documento e ne genera un codice univoco.
- **Codici aggiuntivi**: una serie di parametri aggiuntivi che servono sempre all'interno della logica del gestionale, ma che per me non hanno nessun importanza in quanto non vengono utilizzati, è solo importante salvarli in modo sicuro.

Ora che il processo è stato creato, ho dovuto creare il codice che si occupa di interagire con la blockchain, il punto focale di questo progetto.

3.4.2 Smart Contract

Per poter sviluppare questa parte ho utilizzato il linguaggio Solidity, un linguaggio realizzato da Ethereum appositamente con lo scopo di sviluppare smart contract.

Solidity è un linguaggio di programmazione ad alto livello utilizzato per scrivere smart contract sulla blockchain Ethereum. Grazie a Solidity, gli sviluppatori possono creare applicazioni decentralizzate (DApps) sulla blockchain Ethereum, che possono essere utilizzate per una vasta gamma di scopi, come ad esempio la gestione di token, la registrazione di proprietà intellettuale, la gestione dei contratti e molto altro ancora.

Per poter scrivere codice Solidity ho utilizzato l'editor di testo Remix, un editor online che permette di testare i propri smart-contract su delle blockchain di test, in modo da non aver nessun costo di transazione. Infatti un fattore da considerare è che ogni transazione che viene eseguita sulla blockchain ha un costo, che viene pagato in Ether, la valuta della blockchain Ethereum. Anche il deploy di uno smart-contract è una transazione e perciò ha un costo. Per questo motivo esistono delle blockchain di test, realizzate appositamente per permettere agli sviluppatori di testare i propri codici senza dover ogni volta pagare. L'IDE online Remix aiuta proprio a fare questo, permettendo di scegliere in modo facile su quale blockchain salvare il proprio smart-contract. Questo ovviamente sarebbe stato possibile anche senza Remix, ad esempio scaricando dei software che eseguono una blockchain di test sulla propria macchina come ganache-cli, ma l'IDE online è molto più comodo e veloce da utilizzare.

Inoltre Remix ha una grafica che offre un'interfaccia che aiuta anche a visualizzare i costi effettivi del deploy e delle transazioni, oltre ad avere già installate tutte le versioni del compilatore e del debugger, in modo da poterli cambiare velocemente e senza doverli scaricare.

Il codice Solidity è contenuto in un file con estensione *.sol* ed è composto come segue:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3 contract MyContract {
4     function helloWorld() public pure returns (string memory) {
5         return "Hello, World!";
6     }
7 }
```

In questo esempio si possono vedere alcune parti fondamentali del codice:

- **Versione di Solidity**: con il codice *pragma solidity ^0.8.0* si indica la versione del compilatore di Solidity che deve essere utilizzata per compilare il codice. Questo è importante in quanto ogni versione del compilatore ha delle caratteristiche diverse e quindi è importante specificare la versione che si vuole utilizzare.
- **Contratto**: con la parola chiave *contract* si indica che si sta definendo un nuovo contratto. Il contratto è l'oggetto principale di Solidity, è l'oggetto che viene salvato sulla blockchain e che contiene tutte le funzioni e le variabili che si vogliono utilizzare. Il nome del contratto è il nome inserito dopo la keyword *contract*, in questo caso *MyContract*.
- **Funzioni**: Le funzioni in Solidity sono come le funzioni in altri linguaggi di programmazione, ma hanno alcune caratteristiche particolari. Per prima cosa le funzioni possono avere due livelli di visibilità: public o internal. Le funzioni pubbliche possono essere chiamate da chiunque, mentre le funzioni internal possono essere chiamate solo da altre funzioni all'interno del contratto. Inoltre le funzioni possono avere delle keyword aggiuntive: *view* o *pure*. Le funzioni *view* sono funzioni che non modificano lo stato del contratto, mentre le funzioni *pure* sono funzioni che non modificano lo stato del contratto e non leggono neanche lo stato del contratto. Esistono anche altre keyword come *payable* che permette di ricevere Ether, ma non sono state utilizzate in questo progetto.
- **Tipo di ritorno**: Le funzioni possono ritornare dei valori, come nel caso dell'esempio, in cui la funzione ritorna una stringa. Per specificare il tipo di ritorno della funzione si utilizza la keyword *returns* seguita dal tipo di ritorno. Nel caso in cui la funzione non ritorni nulla, allora non si utilizza la keyword *returns*.

Il codice mostrato nell'esempio è un semplice contratto che quando viene richiamata la funzione *helloworld()* ritorna la stringa "Hello World".

Partendo da questo esempio, il codice che ho dovuto scrievere io è il seguente:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DocumentStore {
    struct HashCode {
        bytes32 value;
        bool present;
    }

    mapping(string => HashCode) private data;

    function addDocument(string calldata key, bytes32 hash_code) public {
        require(!data[key].present, "The document is already present");
        data[key] = HashCode(hash_code, true);
    }

    function getDocumentHashCode(string calldata key) public view returns (
        bytes32) {
        require(data[key].present, "The document is not stored yet");
        return data[key].value;
    }
}
```

Questo codice non ha nulla di troppo complicato, ma è comunque necessario descrivere alcune peculiarità del linguaggio Solidity.
Per prima cosa ho inserito uno struct che contiene due valori: il primo è il valore dell'hash del documento, mentre il secondo è un booleano che indica se il documento è presente o meno. Questo booleano è necessario perché, come si può vedere dalla funzione *addDocument*, non è possibile inserire un valore in una mappa se questo è già presente.
Provenendo da altri linguaggi di programmazione potrebbe venire naturale chiedersi perchè non abbia utilizzato un array e poi cercato se il valore fosse presente al suo interno. Questa decisione è stata presa a causa dei costi di esecuzione delle operazioni di ricerca all'interno di un array. Infatti, mentre la ricerca all'interno di una mappa ha un costo di esecuzione pari a $O(1)$, la ricerca all'interno di un array ha un costo di esecuzione pari a $O(n)$, dove n è la lunghezza dell'array. Questo significa che, se il numero di documenti inseriti nella mappa dovesse diventare molto grande, il costo economico di esecuzione aumenterebbe proporzionalmente in quanto ogni operazione all'interno della blockchain ha un costo in termini di gas fee (tasse di esecuzione).

Il codice che avrei dovuto scrivere nel caso avessi dovuto utilizzare un array per memorizzare i documenti sarebbe stato il seguente:

```
struct Document {
    string key;
    bytes32 hash_code;
}

Document[] private documents;

function addDocument(string calldata key, bytes32 hash_code) public {
    for (uint i = 0; i < documents.length; i++) {
        require(documents[i].key != key, "The document is already present");
    }
    documents.push(Document(key, hash_code));
}

function getDocumentHashCode(string calldata key) public view returns (
    bytes32) {
    for (uint i = 0; i < documents.length; i++) {
        if (documents[i].key == key) {
            return documents[i].hash_code;
        }
    }
    revert("The document is not stored yet");
}
```

Come si può notare in entrambi i casi ho dovuto iterare sugli elementi dell'array tramite un ciclo for e come detto prima questo porta a un costo in termini di risorse e di gas fee.

Un altro punto degno di attenzione del codice è la keyword `calldata` all'interno della definiione dei parametri delle funzioni. Esistono due tipi principali di keyword per i parametri di una funzione: `memory`, e `calldata`.
La loro differenza è il fatto che `memory` è un'area di memoria temporanea dove vengono salvati i dati durante l'esecuzione di una funzione, mentre `calldata` è un'area di memoria in sola lettura dove vengono salvati i dati che vengono passati alla funzione.
Il vantaggio di usare `calldata` è che essendo il dato in sola lettura non porta a nessun costo in termini di gas feei, in quanto se ad esempio un dato di tipo memory viene passato ad un ulteriore funzione questo viene interamente copiato in memoria, mentre se viene passato un dato di tipo `calldata` questo viene passato per riferimento.

3.4.3 Integrazione Smart Contract con Processo

Avendo realizzato le due estremità del sistema, ovvero il processo e lo smart contract, ho dovuto integrarle tra di loro. Per fare ciò ho utilizzato un tool per C# e applicativi .NET chiamato Nethereum. Questo tool permette di realizzare delle classi che fungono da wrapper per gli smart contract e che permettono di interagire con essi. In pratica basta inserire nel progetto il codice realizzato in Solidity nell'apposito file con estensione .sol e poi eseguire un comando che genera le classi wrapper. Queste classi sono realizzate in C# e permettono di interagire con gli smart contract come se fossero delle normali classi. In particolare, per ogni smart contract viene generata una classe che ha lo stesso nome dello smart contract e che contiene tutte le funzioni e le variabili pubbliche dello smart contract. Per esempio, per il mio smart contract `DocumentStore` viene generata la classe `DocumentStoreService` che

contiene tutte le funzioni e le variabili pubbliche di `DocumentStore`. Per utilizzare questa classe basta istanziarla e passare al costruttore l'indirizzo dello smart contract e l'istanza di Web3 che permette di interagire con la blockchain. In questo modo si ottiene un'istanza della classe che permette di interagire con lo smart contract.

I due codici che ho realizzato sono i seguenti:

```
1  public async Task<byte[]> GetHashCodeFromSmartContract(string[] keyToSearch)
2  {
3      if (keyToSearch.Length == 0)
4          throw new System.ArgumentException("No key to search found");
5
6      Web3 web3 = new Web3(_chainURL);
7      var contractHandler = web3.Eth.GetContractHandler(_smartContractAddress)
           ;
8
9      string key = ConcatStrings(keyToSearch);
10
11     var getDocumentFunction = new GetDocumentHashCodeFunction() { Key = key
           };
12
13     var returnValue = await contractHandler
14         .QueryAsync<GetDocumentHashCodeFunction,byte[]>(getDocumentFunction)
               ;
15
16     return returnValue;
17 }
```

In questo codice si può vedere come viene utilizzata la classe generata automaticamente da Nethereum. In particolare, si può vedere come viene istanziata la classe `GetDocumentHashCodeFunction` che rappresenta la funzione `getDocumentHashCode` dello smart contract. Questa classe viene istanziata passando come parametro la chiave da cercare. Questa classe viene poi passata al metodo `QueryAsync` che permette di eseguire la query sulla blockchain. Il metodo restituisce un oggetto di tipo `Task<byte[]>` che rappresenta il risultato della query.

```
public async Task<TransactionReceipt> StoreDocumentToSmartContract(
    string[] keyToStore,
    byte[] hashCodeToStore
)
{
    if (keyToStore.Length == 0)
        throw new System.ArgumentException("No key to store found");
    else if (hashCodeToStore.Length == 0)
        throw new System.ArgumentException("No hash code to store found");

    Account account = new Account(_userPrivateKey, _chainID);
    Web3 web3 = new Web3(account, _chainURL);

    var contractHandler = web3.Eth.GetContractHandler(_smartContractAddress)
        ;

    var addDocumentFunction = new AddDocumentFunction()
    {
        Key = ConcatStrings(keyToStore),
        Hash_code = hashCodeToStore,
    };

    var addDocumentFunctionTxnReceipt = await contractHandler
        .SendRequestAndWaitForReceiptAsync(addDocumentFunction);

    return addDocumentFunctionTxnReceipt;
}
```

Invece nel salvataggio del documento nello smart contract, si può vedere come viene istanziata la classe `AddDocumentFunction` che rappresenta la funzione `addDocument` dello smart contract. Questa classe viene istanziata passando come parametri la chiave e l'hash code del documento da salvare. Questa classe viene poi passata al metodo `SendRequestAndWaitForReceiptAsync` che permette di eseguire la transazione sulla blockchain. Il metodo restituisce un oggetto di tipo `Task<TransactionReceipt>` che rappresenta il risultato della transazione.

Ora che anche la comunicazione tra le due parti è stata realizzata, il progetto è stato testato per vedere che tutto funzionasse come previsto, rimaneva solo da valutare se fosse possibile vendere questo prodotto all'interno del pacchetto del gestionale già presente.

3.5 Analisi prodotto

Una volta terminato il progetto ho effettuato un'analisi con il mio tutor aziendale per capire se effettivamente il software avesse del potenziale. Le conclusioni che sono state prese sono le seguenti:

- Il prodotto utilizza una tecnologia che i clienti non sono abituati ad utilizzare, quindi andrebbero educati e andrebbe spiegata loro l'utilità di queste operazioni. Di sicuro si potrebbe fare, ma il vantaggio che si otterrebbe non giustifica lo sforzo che si dovrebbe fare.
- I costi di transazione per l'upload di un documento sono troppo elevati per gli standard a cui sono abituati i clienti, si parla di decine di centesimi fino ad arrivare all'euro per ogni documento salvato, cifra non vendibile considerando che per ogni utente vengono caricati decine di file.

- Il costo per scrittura e lettura di un documento olre ad essere elevato è anche molto fluttuante e dipende da molti fattori come il prezzo dei gas fee, il prezzo valore del token Etheri, il costo per ogni operazione eseguita che dipende anche dagli aggiornamenti della piattaforma Ethereum ... Questo fattore rende difficile anche vendere il prodotto in quanto risulta complicato dare una stima al cliente dei costi effettivi che dovrà sostenere e per questo motivo il cliente potrebbe non essere interessato.

A causa di queste complicazioni si è deciso di archiviare il progetto fino a quando la tecnologia non sarà più diffusa e i costi saranno più contenuti.
In ogni lo sviluppo da parte mia è stato molto interessante perchè ha portato alla realizzazione di un prodotto che utilizza una tecnologia innovativa e che ha permesso di approfondire le conoscenze su questa tecnologia. Inoltre, la sperimentazione di paradigmi nuovi porta sempre ad effettuare un'analisi degli attuali problemi in cerca di soluzioni più efficaci e questo non può che essere un bene per l'azienda.