



# HEDERA



Analisi del sistema Hedera e dei costi di esecuzione delle applicazioni



2023

PROGETTO E SVILUPPO DI APPLICAZIONI BLOCKCHAIN

Luigi Mereu 70/87/46312

Mirco Aresu 60/73/65283

## ABSTRACT

Collaboration between

University of Cagliari, Italy

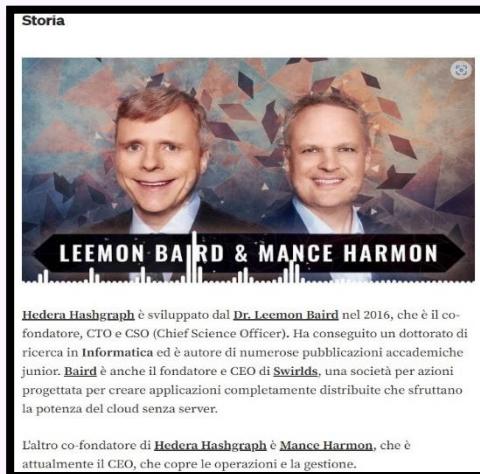
Mathematics and Computer Science Department and Department of Electrical and Electronic Engineering

Negli ultimi anni, la tecnologia **Blockchain** ha guadagnato notorietà grazie alla sua capacità di sostenere applicazioni **decentralizzate**, garantendo **sicurezza**, **trasparenza** e resistenza alla **censura**. Tuttavia, le crescenti preoccupazioni riguardo all'**efficienza energetica**, i **tempi di transazione** e alla **scalabilità** delle piattaforme blockchain tradizionali, come **Ethereum** (anche se nel mentre, la Ethereum Foundation ha cercato di arginare il problema della sostenibilità energetica modificando il proprio algoritmo di consenso da **Proof of Work** a **Proof of Stake**, con l'aggiornamento "The Merge" nel settembre 2022), hanno spinto la ricerca ad alternative più efficienti e sostenibili seppur sacrificando parte della decentralizzazione, cosa non vista benissimo da chi nel panorama si identifica come "**Maximalista**" o "**Bitcoiner**", poiché per una **criptovaluta** il cui scopo sia quello di fungere da digital cash **P2P**, come appunto **Bitcoin**, la **decentralizzazione** è uno dei tre requisiti del **trilemma blockchain** (decentralizzazione, sicurezza e scalabilità) decisamente meno minimizzabile. Lo stesso invece, può non essere vero quando si parla di infrastrutture di rete che conservano un certo grado di decentramento ma al tempo stesso, un elevato **throughput transazionale**. In questa prospettiva, **Hedera Hashgraph**, pur essendo meno decentralizzata rispetto Ethereum, emerge come una soluzione promettente, fornendo vantaggi in termini di **velocità**, **sicurezza** e **scalabilità**. Nonostante il potenziale di Hedera, è di fondamentale importanza condurre un'analisi approfondita dei costi legati all'esecuzione di applicazioni basate su **smart contract**, in quanto tali fattori influenzano direttamente l'adozione e l'efficacia delle soluzioni basate su questa piattaforma. Lo scopo di questo studio consiste nell'esaminare dettagliatamente la rete Hedera, come è organizzata e fruibile ma soprattutto i costi di esecuzione delle applicazioni attraverso lo sviluppo e il testing di diversi smart contract per vari casi d'uso e, in particolare, valutarne i costi di implementazione e di esecuzione per ciascuno di essi. Nelle sezioni seguenti, affronteremo in modo approfondito la **metodologia** adottata, i **risultati** ottenuti e le implicazioni pratiche dei nostri risultati per lo sviluppo di applicazioni basate sulla rete Hedera.

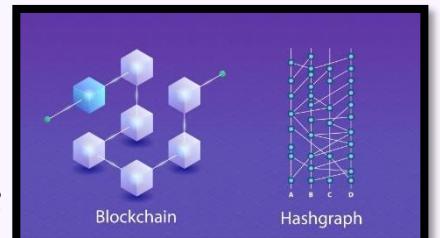
## SEZIONE 1

### HEDERA - BREVE PANORAMICA NEL SETTORE

Hedera è una **Chain Layer 1** che annovera tra i suoi validatori aziende del calibro di **Google**, **IBM**, **Ubisoft**, numerose **banche** e **Deutsche Telekom**, la compagnia di telecomunicazioni più grande della Germania ed ha come obiettivo principale quello di offrire soluzioni **aziendali**. Va da sé che, per questo motivo, ha dei **casi d'uso** orientati soprattutto ad una clientela di tipo **enterprise istituzionale** e fa proprie le **narrative** come la **scalabilità**, la **velocità di transazione** e l'essere **poco inquinante**, persino **carbon negative**, e quindi a **basso impatto ambientale**. L'azienda produttrice di dispositivi elettronici di consumo **LG**, ha affermato che rilascerà su Hedera la piattaforma **NFT** proprietaria, la quale sarà integrata nelle relative **TV** e alcuni **rumor** addirittura lasciano intendere che potrebbero essere sviluppate alcune delle future grandi **CBDC** (**Central Bank Digital Currency**). Tutto questo, ovviamente, al prezzo di sacrificare parte della **decentralizzazione**, caratteristica principale e portante della **narrativa** e della tecnologia stessa quale **Blockchain** e più in generale dei **Registri Distribuiti**.



## HEDERA - ENTERPRISE DISTRIBUTED LEDGER



Hedera è governata dall'**algoritmo di consenso Hashgraph** (da qui il nome completo **Hedera Hashgraph**).

Possiamo definire l'intera infrastruttura come un **Enterprise Distributed Ledger**, in quanto:

- **Enterprise**: perché appunto, estremamente orientata a casi d'uso di tipo aziendale istituzionale;
- **Distributed Ledger** perché non si tratta di una vera e propria Blockchain ma di un **DAG**, ovvero di un **Directed Acyclic Graph** (quindi un grafo) nel quale non esiste il **blocktime** come nelle Blockchain classiche (per definizione, catena di blocchi). L'algoritmo Hashgraph è da intendersi invece come un **registro pubblico** che tiene traccia delle transazioni, il tutto con l'ausilio di una serie di **timestamp** tra i nodi della rete. A primo impatto si direbbe quindi simile rispetto Ethereum e più in generale rispetto tutte le blockchain, ma le cose non stanno propriamente così: nell'Hashgraph, infatti, le transazioni sono validate in base alla loro posizione in relazione a tutte le altre, cosa che quindi, potrebbe essere correlata a una blockchain le cui transazioni sono contestualizzate in base sì alla loro posizione, ma in rapporto alla chain stessa, con la netta differenza che in quest'ultimo caso, la blockchain deve essere in modalità di lavoro “perpetua”, indipendentemente dal traffico di transazioni che vengono registrate in un determinato momento. Sull'Hashgraph, d'altra parte, le transazioni vengono inviate per conferma solo ai nodi attivi, e non a tutti i nodi, come su una blockchain tradizionale.

Le altre principali caratteristiche di Hedera possono essere schematizzate nel seguente modo:

- Elevata adozione **istituzionale e partnership**:

Vanta tanti grossi nomi di aziende che nell'insieme espleteranno il ruolo di validatori del Network (Governing Council), come appunto il già citato **Google**, ma anche **IBM**, **LG**, **banche istituzionali** e aziende del settore tech più in generale. Per svolgere un ruolo chiave in termini di protezione della rete, queste entità devono possedere una certa quantità di token **HBAR** che servirà appunto per eseguire l'algoritmo di convalida dei blocchi che a sua volta rientra nella grande categoria delle **Proof of Stake**; anche per poter costruirsi **applicazioni decentralizzate**, servirà necessariamente possedere il relativo token, rendendolo di fatto utile per pagare le commissioni di rete e più in generale ogni pagamento previsto per ogni applicazione. E' bene sottolineare che, proprio per via del fatto che è un Network fortemente orientato all'istituzionalità, non tutti possono diventare validatori, essendo un protocollo di tipo **permissioned** (ma comunque open), per cui prima di poter fare parte della rete dei nodi di convalida dei blocchi, bisognerà obbligatoriamente soddisfare determinati requisiti imposti da chi fa già parte del Governing Council, primo fra tutti essere un'azienda enterprise, rendendo quindi la rete meno decentralizzata rispetto altre chain. Nel tempo, a detta dell'**Hedera Foundation** stessa, cercheranno di rendere il protocollo meno decentralizzato.

- Elevata **scalabilità e tempi di convalida dei blocchi** molto contenuti:

questo è possibile proprio grazie all'architettura DAG che garantirebbe, a loro dire, una esecuzione di **diecimila transazioni semplici al secondo**, intese quindi come trasferimento di sola criptovaluta da un indirizzo (Wallet) ad un altro; Nel caso si trattassero di transazioni che richiedessero “il deploy” di **Smart Contract** e quindi di funzionalità più complesse, in realtà, questo numero scende, nonostante il throughput non sia stato precisamente individuato (in ogni caso circa un ordine di grandezza in meno).

- **Compatibilità con gli Smart Contracts**: Per l'appunto, compatibilità con gli **Smart Contract**, ovvero blocchi di codice scritto in linguaggio specifico ed eseguiti dalla **Macchina Virtuale**, in questo caso quella di Hedera.
- **Compatibilità con la EVM (Ethereum Virtual Machine)**, momentaneamente la Macchina Virtuale più utilizzata in quanto è stata la prima ad introdurre il concetto di **DApp (Decentralized Application)** e quindi l'intero paradigma attraverso il quale la maggior parte dei sistemi distribuiti decentralizzati di archiviazione di dati prende spunto: questo significa che una DApp sviluppata su Ethereum può essere eseguita anche su Hedera, rendendo di fatto il codice, e quindi il servizio, potenzialmente accessibile e usufruibile ad un'utenza maggiore.

## *ALGORITMO DI CONSENSO*

L'algoritmo di consenso volto a verificare e inserire un blocco all'interno del registro distribuito, differisce leggermente rispetto alle altre blockchain con algoritmo di consenso **Proof of Stake**: in comune a quest'ultime, nonostante i nodi validatori abbiano comunque della quantità di criptovalute da mettere in **stake** e votano sulla validità o meno delle transazioni, le stesse vengono **accodate** l'una all'altra, come già descritto, attraverso il meccanismo denominato **ABFT (Asynchronous Byzantine Fault Tolerance)**, dove il primo termine indica proprio il fatto che nei blocchi manca il **blocktime** che va scandire una tempistica di validazione, quindi nel momento in cui i nodi ricevono una transazione devono validarla in modo sequenziale vedendo soprattutto quello che c'è prima. Per il resto, il meccanismo di consenso è del tutto analogo al **BFT** implementato da altri protocolli (come Ethereum stesso, dopo l'aggiornamento Merge) dove serve il **66%** del consenso dei nodi validatori per definire una transazione valida e quindi memorizzarla all'interno del registro. Seppur trattandosi, come anticipato in precedenza, di un algoritmo che rientra nella grande famiglia delle **Proof of Stake**, in Hedera è implementato in modo particolare: introdotto relativamente da poco, prima gli unici a poter fare **Staking** erano solo i nodi validatori, mentre da adesso, attraverso delle fasi programmate tutt'ora in corso, si sta cercando di rendere il network meno centralizzato e quindi ci si aspetta che ognuno potrà fare la propria parte per quanto concerne la sicurezza del network. Inoltre, a differenza di altri algoritmi Proof of Stake, non è presente lo **Slashing**, in quanto non ci sarebbe motivo o comunque ragionabile incentivo a validare transazioni malevole oppure censurarne altre; pertanto, non è prevista una sottrazione di quantitativo di criptovaluta messa in staking nei casi descritti.





## HBAR - TOKEN ED ECONOMIA DELLA RETE

Il token della rete prende il nome di **HBAR** ed è un **Native Token**: questo significa che non segue lo standard **ERC-20**, il quale è specifico per la piattaforma Ethereum e definisce le regole per la creazione di token fungibili sulla rete stessa. HBAR ha essenzialmente due scopi e cioè:

- Lo **staking**, quindi la sicurezza del Network in termini di staker o Proxy staking
- Il **pagamento delle fees**, le quali sono estremamente basse, parliamo dell'ordine del millesimo di dollaro, motivo per cui è ottimo per DApp altamente scalabili o comunque che prevedono il registro di un numero elevato di transazioni in un certo intervallo di tempo (in genere comunque molto piccolo) ma rendendo il lavoro dello staker poco remunerativo, pertanto il validatore ha bisogno di ulteriori incentivi per rimanere tale in quanto anche il numero di transazioni non è, ad oggi, relativamente elevato, come ci mostrerebbe un **explorer** (strumento attraverso il quale possiamo monitorare in modo pratico e intuitivo lo stato di una blockchain, potendo cercare address, hash delle transazioni e altre informazioni specifiche). Per questa ragione, c'è un pool di HBAR preminati e messi da parte dalla Foundation che vengono impiegati come ricompensa aggiuntiva per i nodi di convalida. È bene evidenziare che comunque, le fees per la convalida delle transazioni, vengono per il **90%** impiegate per la ricompensa dello staker, mentre la restante parte del **10%** vengono recuperate dalla **Hedera Foundation**.

L'economia, di fatto, è estremamente semplice ed elementare: le **DApp** vengono sviluppate sopra il protocollo Hedera, chi intende interagire con gli **smart contract** paga le **transaction fees**, le quali come abbiamo visto ricompenseranno i validatori, che a loro volta beneficeranno di un network con una utenza elevata, che si traduce quasi automaticamente in un maggior volume di dati transati e quindi altre fees. In questo modo si può persino ridurre il quantitativo di criptovaluta erogato dal pool di HBAR preminati, diminuendo così la **selling pressure** che potrebbe spingere gli staker a vendere a mercato gli HBAR così guadagnati tenendo in questo modo molto basso il prezzo per unità di moneta, rendendolo infine poco attraente per futuri investitori che cercheranno un profitto economico.

È doveroso spendere qualche altro commento in merito alla **total supply** del token che si, abbiamo visto essere fissata a **50 miliardi** di unità, ma teoricamente e praticamente incrementabile, in quanto basterebbe che tutti i nodi del Council fossero d'accordo per immettere nel mercato nuovi token, ma questo, se da un lato è facile mettere d'accordo poche entità tra loro, dall'altro è altamente improbabile che ciò avvenga in quanto inflazionare il token comporterebbe la perdita di valore di HBAR stesso (a parità di domanda) e quindi, contro gli interessi degli stakeholder.

Il simbolo per gli HBAR è "**h**", quindi 5 **h** significa 5 HBAR.

Le barre minuscole sono più piccole degli HBAR. Sono utilizzati per dividere gli HBAR in quantità minori. Un HBAR equivale a cento milioni di minuscole.

Allo stesso modo, il simbolo per le barre minuscole è "**th**", quindi è corretto dire 1 **h** = 100.000.000 **th**  
Denominazioni ufficiali HBAR :

gigabar 1 **Gh** = 1,000,000,000 **h**

megabar 1 **Mh** = 1,000,000 **h**

kilobar 1 **kh** = 1,000 **h**

hbar 1 **h** = 1 **h**

millibar 1,000 **mh** = 1 **h**

microbar 1,000,000 **uh** = 1 **h**

tinybar 100,000,000 **th** = 1 **h**



## SEZIONE 2

### CASI D'USO ED ANALISI DEGLI SMART CONTRACT

In questa sezione, **esploreremo** diversi **casi d'uso** per la piattaforma **Hedera** ed **analizzeremo gli Smart Contract** oggetto di studio, scritti in linguaggio **Solidity** sull'**IDE Remix** e **commenteremo le righe di codice** più significative dal punto di vista funzionale; si cercherà di dare una spiegazione del perché ci sia bisogno di una architettura di tipo **Registro Distribuito Decentralizzato** anziché una classica architettura **Client-Server** e quindi, del problema che si cerca di risolvere.

#### 1) Simple Transfer

##### **Problema:**

Ogni qualvolta utilizziamo un mezzo di pagamento tradizionale web-based oppure semplicemente una carta di credito/debito, più o meno inconsciamente ci affidiamo a una entità terza (il caso più frequente è rappresentato da una banca). Gli scenari che si possono verificare, più o meno giustificati, sono i seguenti: transazioni bloccate in quanto ritenute sospette (anche preventivamente), cancellazione dei conti a causa di modifiche dei contratti unilaterali stipulati con gli istituti di credito, impossibilità di aprire un conto e quindi non poter usufruire di mezzi di pagamento online.

##### **Soluzione:**

Possiamo sfruttare una rete Peer-to-Peer per eliminare l'intermediario e poter eseguire pagamenti in tutta autonomia

##### **Implementazione:**

Il seguente smart contract consente a un mittente di depositare dei fondi e, al destinatario, di prelevare gli stessi:

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity ^0.8.1;
3
4 contract SimpleTransfer{
5
6     address payable recipient;
7     address owner;
8
9     constructor(address payable _recipient){    infinite gas 275200 gas
10        recipient = _recipient;
11        owner = msg.sender;
12    }
13
14     function deposit() public payable {    2579 gas
15        require(msg.sender == owner, "only the owner can deposit");
16    }
17
18     function withdraw(uint256 amount) public {    infinite gas
19        require(msg.sender == recipient, "only the recipient can withdraw");
20        require(amount <= address(this).balance, "the contract balance is less than required amount");
21
22        (bool success, ) = recipient.call{value: amount}("");
23        require(success, "Transfer failed.");
24    }
25 }
```

Attraverso le variabili globali di tipo *address* dichiariamo l'indirizzo di destinazione *recipient* (*payable*) e *owner*, cioè di colui che crea il contratto e che quindi invia (deposita) i fondi, entrambe inizializzate dal costruttore, chiamato durante il processo di deploy.

Il metodo *deposit* (*public payable*) permetterà il deposito all'*owner* attraverso l'implementazione di un controllo dato dalla *require* che si assicura che solo l'*address* proprietario possa inviare dei fondi, mentre la funzione *withdraw* consente solo al destinatario dei fondi, attraverso la prima *require*, di prelevare una certa quantità di token.

L'ultima *require* ci informerà dell'avvenuto successo o meno della transazione.

## 2) Data Transfer

### **Problema:**

Quando archiviamo e rendiamo accessibili dei dati sul Web, potremmo necessitare che gli stessi non siano oggetto di modifica indesiderata da parte di soggetti con scopo malevolo, oppure che questi siano semplicemente non cancellabili da parte di entità terze o ancora, vorremmo che i dati fossero di dominio pubblico e facilmente verificabili, a seconda del caso d'uso che vogliamo implementare. Potremmo infine aver bisogno che i dati siano sempre disponibili 24/7.

### **Soluzione:**

Potremmo sfruttare l'immutabilità dei registri distribuiti per la prima e seconda specifica, mentre per la terza possiamo sfruttare la loro trasparenza; Poiché non c'è un solo Server che in caso di inattività pregiudichi la reperibilità dei dati, essendo questi ridondanti in ciascun nodo che costituisce la rete, abbiamo la garanzia che siano accessibili in qualunque momento.

### **Implementazione:**

Il seguente smart contract consente di memorizzare una sequenza di byte e una stringa all'interno del registro distribuito.

```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity ^0.8.0;
4
5 contract DataStorage {
6
7     bytes public byteSequence;
8     string public textString;
9
10    function storeBytes(bytes memory _byteSequence) public {    ┌ infinite gas
11        byteSequence = _byteSequence;
12    }
13
14    function storeString(string memory _textString) public {    ┌ infinite gas
15        textString = _textString;
16    }
17
18 }
```

Attraverso la variabile globale *byteSequence* possiamo memorizzare un dato (immagine, file o altri dati) strutturato in formato bytes, mentre attraverso la variabile *textString* di tipo string, possiamo memorizzare una stringa di testo, come per esempio un messaggio o una descrizione.

Attraverso le funzioni *storeBytes* e *storeString* possiamo assegnare e aggiornare i valori corrispettivi.

### 3) CrowdFund

#### **Problema:**

Può capitare, come già successo in passato, che i fondi raccolti tramite una raccolta possano non arrivare a destinazione, tutti o in parte che siano; Peggio ancora, che qualche malintenzionato si finga la persona o l'organizzazione che realmente abbia dato al via la raccolta fondi, incassando ingiustamente la quantità raccolta; inoltre, sarebbe comodo poter tenere traccia di ogni donazione ma soprattutto sapere con certezza dell'uso che se ne farà del denaro raccolto, soprattutto per i più scettici.

#### **Soluzione:**

Potremmo sfruttare la trasparenza dei registri distribuiti per risolvere i problemi citati attraverso anche l'ulteriore ausilio dell'identità digitale, protocolli nati per associare (anche) un indirizzo blockchain a una specifica identità, come per esempio il popolare protocollo ENS basato su Ethereum, il tutto mantenendo la decentralizzazione e quindi facendo sempre a meno di una terza parte che faccia da tramite.

#### **Implementazione:**

Il seguente contratto consente agli utenti di donare un ammontare di criptovaluta per un determinato periodo di tempo fintanto che venga raggiunto l'obiettivo imposto dal contratto stesso.

```

1 // SPDX-License-Identifier: GPL-3.0-only
2 pragma solidity >= 0.8.2;
3
4 contract CrowdFund {
5
6     uint end_donate;    // last block in which users can donate
7     uint goal;          // amount of ETH that must be donated for the crowdfunding to be successful
8     address receiver;  // receiver of the donated funds
9     mapping(address => uint) public donors;
10
11    constructor (address payable receiver_, uint end_donate_, uint256 goal_) {    // infinite gas 256000 gas
12        receiver = receiver_;
13        end_donate = end_donate_;
14        goal = goal_;
15    }
16
17    function donate() public payable {    // infinite gas
18        require (block.number <= end_donate);
19        donors[msg.sender] += msg.value;
20    }
21
22    function withdraw() public {    // infinite gas
23        require (block.number >= end_donate);
24        require (address(this).balance >= goal);
25        (bool succ,) = receiver.call{value: address(this).balance}("");
26        require(succ);
27    }
28
29    function reclaim() public {    // infinite gas
30        require (block.number >= end_donate);
31        require (address(this).balance < goal);
32        require (donors[msg.sender] > 0);
33        uint amount = donors[msg.sender];
34        donors[msg.sender] = 0;
35        (bool succ,) = msg.sender.call{value: amount}("");
36        require(succ);
37    }
38 }
```

Attraverso la variabile globale *receiver* definiamo l'address che riceverà i fondi donati e tramite *goal* l'attributo che definisce la quota di denaro che si intende raggiungere; entrambe, sono assegnate dal costruttore del contratto. La variabile *end donate* definisce l'ultimo blocco in cui è possibile donare (a seconda della blockchain sappiamo ricavarci il tempo *T* sapendo che per ogni epoca verranno validati *N* blocchi), mentre con *donors* teniamo traccia dei donatori attraverso un mapping.

La funzione *donate* permette di fare ciò, effettuando un controllo per verificare se l'utente stia donando entro il periodo consentito e infine aggiunge la donazione all'elenco delle donazioni eseguite dall'utente stesso.

La funzione *withdraw* consente al ricevente dei fondi di prelevarli dopo il termine del periodo di donazione, effettuando un controllo per verificare se il blocco "corrente" è successivo o uguale al blocco finale entro il quale è possibile donare e se il totale donato è maggiore o uguale all'obiettivo. Inoltre, è presente un ulteriore controllo che ci informerà se la transazione è avvenuta con successo.

Infine, la funzione *reclaim*, permette agli utenti che hanno effettuato donazioni di richiedere il rimborso dei fondi non ancora prelevati dopo la scadenza del periodo di donazione, qualora l'obiettivo del finanziamento non sia stato raggiunto. Anche qui, ci saranno i dovuti controlli e verrà modificato, nel mapping, l'importo precedentemente donato dall'utente, riportandolo a zero. Ci verrà comunicata l'avvenuta restituzione.

## 4) Product & Factory

### **Problema:**

Capita spesso in ambito commerciale, specie in ambito alimentare e farmaceutico, di avere la necessità di poter tracciare un prodotto, per poterne accertare la provenienza, averne in generale tutta la storia in termini di passaggi di proprietà e al tempo stesso assicurarci che corrisponda a ciò che realmente ci aspetteremmo di ricevere. Si pensi quindi al problema della ricettazione e contraffazione, fenomeni di per sé difficili da combattere.

### **Soluzione:**

Potremmo servirci della tecnologia a registro distribuito decentralizzato per tenere una tracciabilità dei prodotti che sia impossibile da manomettere e contraffare.

### **Implementazione:**

Gli smart contract proposti permettono di registrare le informazioni di base di un prodotto, come il tag identificativo, il proprietario e la fabbrica che lo ha creato.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.1;
3
4 contract Product{
5     string tag;
6     address owner;
7     address factory;
8
9     constructor(string memory _tag){    infinite gas 195800 gas
10        owner = tx.origin;
11        factory = msg.sender;
12        tag = _tag;
13    }
14
15     function getTag() public view returns(string memory){    infinite gas
16         require(msg.sender == owner, "only the owner");
17         return tag;
18     }
19
20     function getFactory() public view returns(address){    2500 gas
21         return factory;
22     }
23 }
```

```
24
25 contract Factory{
26     mapping (address => address[]) productList;
27     function createProduct(string memory _tag) public returns(address) {    infinite gas
28         Product p = new Product(_tag);
29         productList[msg.sender].push(address(p));
30         return address(p);
31     }
32
33     function getProducts() public view returns(address[] memory){    infinite gas
34         return productList[msg.sender];
35     }
36 }
```

Per quanto riguarda il contratto Product, la variabile di stato *tag* è una stringa che rappresenta un identificatore o appunto un tag del prodotto che si desidera memorizzare all'interno del registro distribuito; la variabile di stato *owner* è l'indirizzo dell'utente che ha creato il prodotto, mentre la variabile *factory* rappresenta l'indirizzo del contratto omologo con medesime finalità.

Il costruttore viene chiamato quando viene creato un nuovo prodotto e inizializza i valori delle variabili di stato.

Infine, la funzione *getTag* restituisce il tag del prodotto solo se il chiamante della funzione è il proprietario del medesimo mentre la funzione *getFactory* restituisce l'indirizzo del contratto Factory che ha creato il prodotto.

Nel contratto Factory che tiene traccia dei prodotti fabbricati, invece, il mapping *productList* tiene traccia dei prodotti creati da ciascuno utente.

## 5) HTLC

### Problema:

Uguale a quello identificato nel Simple Transfer, con l'aggiunta di avere la necessità che il trasferimento dei fondi avvenga se e solo se venisse soddisfatta una condizione, qualsiasi essa sia. Ad esempio, nella pratica quotidiana, ci possiamo affidare ad un notaio (che comunque costituisce la terza entità di cui si vorrebbe fare a meno) il quale si occuperà del problema in questione.

### Soluzione:

Potremmo usufruire di un contratto che abbia, ad esempio, come condizione vincolante la variabile tempo, concordata tra le due parti che desiderano scambiarsi le risorse e senza che una parte terza faccia da garante, sfruttando la caratteristica trustless della rete.

### Implementazione:

Lo smart contract HTLC riportato (Hashed Timelock Contract) implementa la specifica descritta in questo modo:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.18;
3
4 contract HTLC {
5     address payable public owner;
6     address payable public verifier;
7     bytes32 public hash;
8     uint reveal_timeout;
9
10    constructor(address payable v, bytes32 h, uint delay) payable {    infinite gas 364000 gas
11        require (msg.value >= 1 ether);
12        owner = payable(msg.sender);
13        verifier = v;
14        hash = h;
15        reveal_timeout = block.number + delay;
16    }
17
18    function reveal(string memory s) public {    infinite gas
19        require (msg.sender==owner);
20        require(keccak256(abi.encodePacked(s))==hash);
21        (bool success,) = owner.call{value: address(this).balance}("");
22        require (success, "Transfer failed.");
23    }
}
```

```
24
25    function timeout() public {    infinite gas
26        require (block.number > reveal_timeout);
27        (bool success,) = verifier.call{value: address(this).balance}("");
28        require (success, "Transfer failed.");
29    }
30 }
```

La variabile di stato *owner* e *verifier* rappresentano, rispettivamente, l'indirizzo del mittente e del destinatario; la variabile *hash* è appunto un hash crittografico generato a partire da una stringa fornita in fase di creazione del contratto, mentre la variabile *reveal timeout* è il numero di blocco in cui scade il timeout per rivelare la stringa originale.

Il costruttore invece, viene chiamato in fase di creazione del contratto e richiede che venga mandato almeno 1 Ether insieme alla transazione, impostando poi i valori per *owner*, *verifier* e *hash* calcolando infine il valore del blocco di scadenza del timeout.

La funzione *reveal* permette all'*owner* di rivelare la stringa originale e richiedere il trasferimento dei fondi all' *owner*. Verifica quindi che il chiamante sia il medesimo e che l'hash della stringa fornita sia uguale all'hash salvato nel contratto; successivamente, vengono trasferiti i fondi dal contratto HTLC all'*owner*.

La funzione *timeout* consente al *verifier* di richiedere il trasferimento dei fondi a sé stesso in caso scadesse il timeout. Verifica poi che il numero di blocco corrente sia superiore al blocco di scadenza del timeout e infine verranno trasferiti i fondi dall' HTLC al *verifier*.

## 6) Escrow

### **Problema:**

Nella vita reale, le transazioni finanziarie tra acquirente e venditore possono comportare un certo grado di rischio, in particolare quando le due parti coinvolte non si conoscono bene o quando sono presenti preoccupazioni riguardo l'adempimento degli accordi, per esempio, negli acquisti online, nelle transazioni immobiliari ecc.

### **Soluzione:**

Potremmo ricorrere al registro distribuito decentralizzato il quale non ha bisogno di una terza entità che faccia da garante ma appunto sfruttare, anche in questo caso, la proprietà trustless della rete e uno smart contract che automatizzi il processo.

### **Implementazione:**

Il contratto Escrow è un esempio di come sia possibile facilitare le transazioni tra un acquirente e un venditore che potrebbero non fidarsi tra loro ma al tempo stesso offrire garanzie. Si riporta il codice:

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.0;
3
4 contract Escrow {
5     enum States{WAIT_DEPOSIT, WAIT_RECIPIENT, CLOSED}
6     address buyer;
7     address payable seller;
8     uint256 amount;
9     States state;
10
11    modifier onlyBuyer(){
12        require(msg.sender==buyer, "Only the buyer");
13        _;
14    }
15    modifier onlySeller(){
16        require(msg.sender==seller, "Only the seller");
17        _;
18    }
19
20    constructor(uint256 _amount, address payable _buyer, address payable _seller) {
21        require(_seller != address(0x0) && _buyer != address(0x0), "");
22        require(msg.sender == _seller, "The creator must be the seller");
23        amount = _amount;
24        buyer = _buyer;
25        seller = _seller;
26        state = States.WAIT_DEPOSIT;
27    }
28
29    function deposit() public payable onlyBuyer { 30929 gas
30        require(state == States.WAIT_DEPOSIT, "Invalid State");
31        require(msg.value == amount, "Invalid amount");
32        state = States.WAIT_RECIPIENT;
33    }
34
35    function pay() public onlyBuyer { infinite gas
36        require(state == States.WAIT_RECIPIENT, "Invalid State");
37        state = States.CLOSED;
38        (bool success, ) = seller.call{value: amount}("");
39        require(success, "Transfer failed.");
40    }
41
42    function refund() public onlySeller { infinite gas
43        require(state == States.WAIT_RECIPIENT, "Invalid State");
44        state = States.CLOSED;
45        (bool success, ) = buyer.call{value: amount}("");
46        require(success, "Transfer failed.");
47    }
48
49 }
```

“States” è la variabile del contratto globale che enumera i vari stati possibili del contratto, ovvero *WAIT DEPOSIT*, *WAIT\_RECIPIENT* e *CLOSED* che serviranno per tenere traccia degli sviluppi della transazione tra acquirente e venditore.

Il costruttore del contratto invece richiede che l’indirizzo del venditore e del compratore non siano vuoti, oltre che richiedere che il creatore del contratto sia il venditore; vengono impostate la quantità da transare, cioè *amount*, l’acquirente *buyer*, il venditore *seller* e lo stato iniziale *WAIT\_DEPOSIT*.

La funzione *deposit* può essere chiamata solo dall’acquirente (*onlyBuyer*) e richiede che lo stato del contratto sia *WAIT\_DEPOSIT*. L’acquirente deve inviare una quantità di Ether pari all’importo della transazione; se la condizione viene soddisfatta, lo stato del contratto viene modificato in *WAIT\_RECIPIENT*;

La funzione *pay* può essere chiamata sempre solo dall’acquirente e richiede che lo stato del contratto, stavolta, sia *WAIT\_RECIPIENT*; questa funzione conferma la transazione e trasferisce l’importo dell’escrow al venditore. Solo a questo punto, lo stato del contratto verrà quindi modificato in *CLOSED*.

La funzione *refund*, infine, può essere chiamata solo dal venditore (*onlySeller*) e richiede che lo stato del contratto sia *WAIT\_RECIPIENT*. Questa funzione restituisce l’importo dell’escrow all’acquirente, e lo stato del contratto sarà quindi modificato in *CLOSED*.

## 7) Vault

### Problema:

Potremmo aver bisogno di depositare un'importante somma presso un istituto finanziario, magari perché prevediamo di fare un viaggio e prelevare poi quel denaro in un altro stato o comunque sia in un altro momento, senza dovercelo portare appresso con i rischi associati. Ma rimarrebbe il problema di tutti i limiti imposti nel depositare e prelevare denaro presso una banca, in termini di importo e di giustificazioni, per non parlare del fatto che potremmo semplicemente non avere un conto deposito e quindi ne vorremmo usufruire senza però passare per soluzioni centralizzate.

### Soluzione:

Si potrebbe benissimo implementare uno smart contract che garantisca una custodia sicura dei fondi sfruttando la sicurezza di un registro distribuito decentralizzato.

### Implementazione:

Il contratto *Vault* permette di depositare in modo sicuro e decentralizzato i propri fondi consentendo inoltre di controllarne i flussi attraverso il codice seguente:

```
1 // SPDX-License-Identifier: GPL-3.0-only
2 pragma solidity >= 0.8.2;
3
4 contract Vault {
5     enum States{IDLE, REQ}
6
7     address owner;
8     address recovery;
9     uint wait_time;
10
11    address receiver;
12    uint request_time;
13    uint amount;
14    States state;
15
16    constructor (address payable recovery_, uint wait_time_) payable {
17        owner = msg.sender;
18        recovery = recovery_;
19        wait_time = wait_time_;
20        state = States.IDLE;
21    }
22
23    receive() external payable { } undefined gas
24
25    // IDLE -> REQ
26    function withdraw(address receiver_, uint amount_) public {
27        require(state == States.IDLE);
28        require(amount <= address(this).balance);
29        require(msg.sender == owner);
30        request_time = block.number;
31        amount = amount_;
32        receiver = receiver_;
33        state = States.REQ;
34    }
35
36    // REQ -> IDLE
37    function finalize() public infinite gas {
38        require(state == States.REQ);
39        require(block.number >= request_time + wait_time);
40        require (msg.sender == owner);
41        state = States.IDLE;
42        (bool succ,) = receiver.call{value: amount}("");
43        require(succ);
44    }
45
46    // REQ -> IDLE
47    function cancel() public 28812 gas {
48        require(state == States.REQ);
49        require (msg.sender == recovery);
50        state = States.IDLE;
51    }
52 }
```

In cui anche qua, la variabile *States* tiene traccia dello stato del contratto, e può essere *IDLE* quanto non c'è nessuna richiesta di prelievo, e *REQ* quando invece ne è stata fatta una; il proprietario del contratto è l'indirizzo che lo ha creato (*msg.sender*).

Nel costruttore, viene specificato l'indirizzo di ripristino *recovery* e il periodo di attesa (*wait\_time*); inoltre, viene richiesto un deposito iniziale di fondi al momento della creazione del contratto.

La funzione *receive* esterna e payable consente al contratto di accettare depositi di fondi.

La funzione *withdraw* invece, può essere chiamata solo dal proprietario *owner* e richiede che lo stato del contratto sia *IDLE*. Questa avvia una richiesta di prelievo specificando l'indirizzo del destinatario *receiver* e la quantità di fondi *amount* da prelevare; sarà a questo punto che la richiesta di prelievo passerà allo stato *REQ* e registra il blocco corrente come tempo di richiesta *request\_time*.

Anche la funzione *finalize* può essere chiamata solo dal proprietario e richiede che lo stato del contratto sia impostato su *REQ*. Questa finalizza la richiesta di prelievo e trasferisce l'importo specificato al destinatario; per completare l'operazione, devono essere soddisfatte le condizioni in cui il blocco corrente supera il tempo di attesa stabilito dalla somma di *request\_time* e *wait\_time*. Lo stato del contratto viene quindi ripristinato in *IDLE*.

Per concludere, la funzione *cancel* può essere chiamata solo dall'indirizzo di ripristino *recovery* e richiede che lo stato del contratto sia *REQ*; il suo compito è quello di annullare la richiesta di prelievo e ripristinare lo stato del contratto in *IDLE*.

## 8) Vesting

### Problema:

Capita spesso che un'organizzazione voglia ricevere dei fondi da parte di molti utenti in cambio di asset o frazione di fondi stessi, i quali verranno erogati con una certa cadenza, in modo controllato e automatizzato. L'esempio di un finanziamento di questo genere, in un ambiente in cui può venire meno la fiducia e al tempo stesso si ha bisogno da parte degli investitori di monitorare il flusso dei fondi in tutta trasparenza, è rappresentato dalle ICO (Initial Coin Offer) ovvero delle vendite iniziali di token da parte di una startup che appunto stabilisce chi e come riceverà il token del proprio progetto che si intende sviluppare sulla blockchain.

### Soluzione:

Un registro distribuito decentralizzato ci permette di automatizzare anche questo processo e dare fiducia agli investitori più titubanti grazie alla trasparenza della tecnologia in questione, rendendo più facile la nascita di protocolli e non per ultimo di importanza, aiutando le piccole imprese a finanziarsi senza dover passare per un'istituzione finanziaria (evitando quindi i tassi di interesse).

### Implementazione:

Il seguente smart contract propone la gestione e il rilascio programmato e controllato di fondi o token in accordi finanziari, garantendo che il beneficiario possa ottenere l'accesso ai fondi solo dopo il periodo di vesting specificato. Di seguito il codice che implementa le funzionalità descritte:

```
1 // SPDX-License-Identifier: MIT
2 // adapted from OpenZeppelin Contracts finance/VestingWallet.sol
3 pragma solidity ^0.8.0;
4
5 contract Vesting{
6     event EtherReleased(uint256 amount);
7
8     uint256 private _released;
9     address private immutable _beneficiary;
10    uint64 private immutable _start;
11    uint64 private immutable _duration;
12
13    constructor(address beneficiaryAddress, uint64 startTimestamp, uint64 durationSeconds) payable {
14        require(beneficiaryAddress != address(0), "Beneficiary is zero address");
15        _beneficiary = beneficiaryAddress;
16        _start = startTimestamp;
17        _duration = durationSeconds;
18    }
19
20    function release() public virtual {
21        uint256 amount = releasable();
22        _released += amount;
23        (bool success, ) = payable(_beneficiary).call{value: amount}("");
24        require(success, "Transfer failed.");
25    }
26
27    function releasable() public view virtual returns (uint256) {
28        return vestedAmount(uint64(block.timestamp)) - _released;
29    }
30
31    function vestedAmount(uint64 timestamp) public view virtual returns (uint256) {
32        return _vestingSchedule(_beneficiary.balance + _released, timestamp);
33    }
34
35
36    /**
37     * implementation of the vesting formula (linear curve).
38     */
39    function _vestingSchedule(uint256 totalAllocation, uint64 timestamp) internal view returns (uint256) {
40        if (timestamp < _start) {
41            return 0;
42        } else if (timestamp > _start + _duration) {
43            return totalAllocation;
44        } else {
45            return (totalAllocation * (timestamp - _start)) / _duration;
46        }
47    }
48 }
```

```
25         emit EtherReleased(amount);
26     }
27
28     function releasable() public view virtual returns (uint256) {
29         return vestedAmount(uint64(block.timestamp)) - _released;
30     }
31
32     function vestedAmount(uint64 timestamp) public view virtual returns (uint256) {
33         return _vestingSchedule(_beneficiary.balance + _released, timestamp);
34     }
35
36    /**
37     * implementation of the vesting formula (linear curve).
38     */
39    function _vestingSchedule(uint256 totalAllocation, uint64 timestamp) internal view returns (uint256) {
40        if (timestamp < _start) {
41            return 0;
42        } else if (timestamp > _start + _duration) {
43            return totalAllocation;
44        } else {
45            return (totalAllocation * (timestamp - _start)) / _duration;
46        }
47    }
48 }
```

Il contratto accetta come parametri, nel momento della sua creazione, un indirizzo del beneficiario, un timestamp di inizio e una durata in secondi; durante il periodo di *vesting*, il beneficiario può richiamare la funzione *release* per rilasciare una quantità di fondi calcolata in base alla formula del vesting implementata.

La funzione *releasable* restituisce la quantità di fondi che il beneficiario può attualmente rilasciare, tenendo conto del tempo trascorso e dei fondi già rilasciati in precedenza.

La funzione *vestedAmount* invece, calcola la quantità totale di fondi che dovrebbero essere disponibili al beneficiario in base al timestamp fornito. Anche questo calcolo si basa sulla formula di vesting implementata, che segue un andamento lineare.

## 9) Bet Oracle

### Problema:

Nel mondo delle scommesse online, i dati delle stesse potrebbero essere manomessi e quindi non veritieri; si pensi ad esempio a quel tipo di scommesse il cui esito non è noto da un evento di dominio pubblico, ad esempio una partita di calcio, ma per esempio un gioco di carte online. È evidente che i dati potrebbero essere manipolati a favore del banco, creando un ambiente ancora meno equo per gli utenti.

### Soluzione:

Potremmo ancora una volta affidarci alla trasparenza e all'immutabilità di un registro distribuito decentralizzato.

### Implementazione:

Il contratto *Bet* riportato fornisce un ambiente sicuro e trasparente per effettuare scommesse tra i partecipanti:

The image shows two side-by-side code editors displaying Solidity smart contract code. The left editor contains the full contract definition, and the right editor shows two specific functions: `bet` and `oracleSetResult`.

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.0;
3
4 contract Bet {
5
6     // bettor = bettors[choice]
7     mapping (uint => address payable) bettors;
8
9     uint timeout;
10    bool open = true;
11    address oracle;
12    uint wager;
13
14    event newBet(address bet, address bettor, uint choice);
15
16    modifier onlyOracle(){
17        require( oracle == msg.sender , "only the oracle");
18        _;
19    }
20
21
22    constructor(uint _timeout, uint _wager){    infinite gas
23        wager = _wager;
24        oracle = msg.sender;
25        timeout = block.number + _timeout;
26    }
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52 }
```

```
function bet(uint choice) payable public {    infinite gas
    require(msg.value == wager, "invalid value");
    require(choice > 0 && choice <= 2 , "invalid choice");
    require(open, "bets are closed");
    require(timeout >= block.number, "time out");
    require(bettors[choice] == address(0), "choice already selected");

    bettors[choice] = payable(msg.sender);
    emit newBet(address(this), msg.sender, choice);
}

function oracleSetResult(uint result) external onlyOracle{    infinite gas
    require(timeout < block.number, "The bets are still open");
    require(open, "Results had already been set");

    open = false;
    address payable winner = bettors[result];
    (bool success,) = winner.call{value: address(this).balance}("");
    require (success, "Transfer failed.");
}
```

Alla creazione del contratto, viene specificato un periodo di tempo limite *timeout* entro il quale i giocatori possono effettuare le scommesse e viene indicato l'indirizzo dell'oracolo, cioè l'entità non centralizzata che determinerà il risultato finale.

Il mapping *bettors*, in cui la chiave rappresenta la scelta e il valore è l'indirizzo dell'utente che ha effettuato quella determinata scelta, tiene traccia delle scommesse effettuate dai giocatori; l'evento *newBet* viene emesso ogni volta che i giocatori effettuano una scommessa.

Queste vengono effettuate chiamando la funzione *bet* e, specificando la loro scelta *choice* come parametro, potranno partecipare inviando una quantità di token pari all'importo della scommessa *wager*; le scommesse possono essere fatte se queste sono aperte e quindi la variabile *open* sarà *true*, il periodo di tempo limite non sia scaduto (*timeout >= block.number*), la scelta sia valida e cioè che valga 1 o 2 e infine che nessun altro giocatore abbia già selezionato quella stessa scelta.

Quando l'oracolo desidera impostare il risultato finale, chiama la funzione *oracleSetResult* e specifica il risultato come parametro attraverso *result*. Se il periodo di tempo limite è scaduto (*timeout < block.number*) e i risultati non sono ancora stati impostati, l'oracolo può impostare il risultato finale; una volta avvenuto ciò, le scommesse vengono chiuse (e quindi *open* diventa *false*) e il giocatore che ha fatto la scommessa corretta viene dichiarato vincitore, che infine può prelevare l'ammontare dal contratto.

## 10) Token Transfer

### **Problema:**

Il problema target è quasi del tutto analogo a quello già visto nel Simple Transfer, con la differenza che nel contratto seguente possiamo definire un token qualsiasi, purché rispetti lo standard ERC-20.

### **Soluzione e implementazione:**

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.9;
3
4 import "contracts/ERC20.sol";
5
6 contract TheToken is ERC20 {
7     constructor() ERC20("theToken", "TTK") {}    // infinite gas 830000 gas
8
9     function mint(address recipient, uint256 quantity) public{   // infinite
10        _mint(recipient,quantity);
11    }
12 }
13
14 contract TokenTransfer{
15
16     event Withdraw(address indexed sender, uint amount);
17     address payable recipient;
18     address owner;
19     address payable tokenAddress;
20     TheToken theToken;
21
22     constructor(address payable _recipient, address payable _tokenAddress){
23         recipient = _recipient;
24         owner = msg.sender;
25         tokenAddress = _tokenAddress;
26     }
}
```

```
27
28     // "approve" required before calling deposit
29     function deposit(uint256 _amount) public {   // infinite gas
30         require(msg.sender == owner, "only the owner");
31         theToken = TheToken(tokenAddress);
32         address contractAddress = address(this);
33
34         (bool success) = theToken.transferFrom(msg.sender, contractAddress,_amount);
35         require(success, "Deposit failed.");
36     }
37
38
39     function withdraw(uint256 amount) public {   // infinite gas
40         require(msg.sender == recipient, "only the recipient can withdraw");
41         theToken = TheToken(tokenAddress);
42         require(theToken.balanceOf(address(this)) > 0, "The contract balance is zero");
43
44         if ( theToken.balanceOf(address(this)) < amount){
45             amount = theToken.balanceOf(address(this));
46         }
47
48         (bool success) = theToken.transfer(msg.sender, amount);
49         require(success, "Transfer failed.");
50         emit Withdraw(msg.sender, amount);
51     }
52 }
53 }
```

Questo contratto è diviso in due parti: il contratto “*TheToken*” e il contratto “*TokenTransfer*”;

Il primo estende la libreria ERC20 e implementa un token chiamato *theToken* il cui simbolo è *TTK*; è possibile creare nuovi token ERC20 chiamando la funzione *mint* e specificando l’indirizzo del destinatario e la quantità di token da creare.

Il secondo, invece, gestisce il trasferimento del token: viene specificato un indirizzo del destinatario e un indirizzo del token al momento della creazione del contratto. Solo il proprietario dello smart contract può depositare i token tramite la funzione *deposit* e specificando l’importo da depositare. Prima di chiamare la funzione, però, è necessario eseguire l’approvazione tramite la funzione *approve* presente nel contratto ERC20 che è stato importato.

La funzione *deposit* trasferisce i token dal mittente del deposito al contratto *TokenTransfer* attraverso la funzione *transferFrom*, anch’essa presente nel contratto ERC20.

La funzione *withdraw* invece, consente al destinatario specificato di prelevare i token dal contratto *TokenTransfer*; viene verificato se il saldo del contratto è sufficiente per soddisfare la richiesta di prelievo: se il saldo del contratto è inferiore all’importo richiesto, verrà prelevato l’intero saldo disponibile, venendo così trasferiti tramite la funzione *transfer* del contratto ERC20.

L’evento *Withdraw* viene emesso quando avviene un prelievo e include l’indirizzo del mittente e l’importo prelevato.

## 11) Auction

### Problema:

Le aste tradizionali possono presentare diverse sfide e problemi, come la mancanza di trasparenza nelle offerte, la necessità di fidarsi di terze parti per gestire le transazioni finanziarie e il rischio di frodi e manipolazione.

### Soluzione:

Utilizzando un registro distribuito decentralizzato possiamo fare in modo che gli utenti che partecipino all'asta possano avere accesso ad informazioni trasparenti e immutabili, automatizzando i processi delle offerte e di vincita evitando confusioni e garantendo che solo l'offerta più alta sia valida, in modo decentralizzato(implementa quindi un sistema di aste decentralizzato) e quindi facendo completamente a meno di una terza parte che facendo da tramite possa essere di parte, sotto determinate circostanze.

Inoltre, è mitigato anche il rischio di frode.

### Implementazione:

```

1 // SPDX-License-Identifier: MIT
2 // adapted from https://solidity-by-example.org/app/engineering-auction
3
4 pragma solidity ^0.8.17;
5
6 contract auction {
7     enum States {WAIT_START, WAIT_CLOSING, CLOSED}
8
9     event Start();
10    event Bid(address indexed sender, uint amount);
11    event Withdraw(address indexed bidder, uint amount);
12    event End(address winner, uint amount);
13
14    States state;
15
16    string public object;
17    address payable public seller;
18    uint public endTime;
19    address public highestBidder;
20    uint public highestBid;
21
22    mapping(address => uint) public bids;
23

```

```

24    constructor(string memory _object, uint _startingBid) {
25        state = States.WAIT_START;
26        object = _object;
27        seller = payable(msg.sender);
28        highestBid = _startingBid;
29    }
30
31    function start(uint _duration) external {
32        require(state == States.WAIT_START, "Auction already started");
33        require(msg.sender == seller, "Only the seller");
34        state = States.WAIT_CLOSING;
35        endTime = block.timestamp + (_duration * 1 seconds);
36        emit Start();
37    }
38
39    function bid() external payable {
40        require(state == States.WAIT_CLOSING, "Auction not started");
41        require(block.timestamp < endTime, "Time ended");
42        require(msg.value > highestBid, "value < highest");
43
44        // Previous highestBid goes in the list.
45        if (highestBidder != address(0)) {
46            bids[highestBidder] += highestBid;
47        }
48

```

```

49    // If a participant makes a new bid, the previous one is automatically withdrawn
50    // (bids[msg.sender] = 0)
51    withdraw();
52
53    highestBidder = msg.sender;
54    highestBid = msg.value;
55    emit Bid(msg.sender, msg.value);
56}
57
58 function withdraw() public {
59     require(state != States.WAIT_START, "Auction not started");
60     uint bal = bids[msg.sender];
61     bids[msg.sender] = 0;
62
63     (bool success,) = payable(msg.sender).call{value: bal}("");
64     require(success, "Transfer failed.");
65
66     emit Withdraw(msg.sender, bal);
67 }
68
69
70 function end() external {
71     require(msg.sender == seller, "Only the seller");
72     require(state == States.WAIT_CLOSING, "Auction not started");
73     require(block.timestamp >= endTime, "Auction not ended");
74     state = States.CLOSED;
75
76     (bool success,) = seller.call{value: highestBid}("");
77     require(success, "Transfer failed.");
78
79     emit End(highestBidder, highestBid);
80 }
81
82

```

dove sono presenti tre possibili stati, quali *WAIT\_START*, *WAIT\_CLOSING* e *CLOSED* che indicano rispettivamente che il contratto è in attesa di iniziare l'asta, in attesa di chiusura e chiusura avvenuta.

Viene emesso un evento *Start* quando l'asta inizia, un evento *Bid* quando viene effettuata una nuova offerta, un evento *Withdraw* quando un partecipante ritira una propria offerta e un evento *End* quando l'asta termina.

Il venditore inizializza l'asta specificando un oggetto in vendita e un'offerta di partenza e può definirne la durata.

I partecipanti possono effettuare offerte durante il periodo dell'asta, le quali devono superare l'offerta più alta attuale; se lo stesso partecipante fa una nuova offerta, quella vecchia viene automaticamente ritirata e gli importi delle precedenti offerte vengono memorizzati nel mapping *bids*, associati all'indirizzo del partecipante; potranno ritirare le offerte se l'asta non è ancora partita.

Quando l'asta termina, il venditore riceve l'offerta più alta e l'indirizzo del partecipante vincitore viene registrato.

Solo a questo punto il venditore potrà ritirare l'importo dell'offerta più alta.

## 12) Decentralized Identity

### Problema:

Le identità decentralizzate sono un concetto chiave nel contesto delle tecnologie blockchain e consentono agli utenti di avere il controllo diretto delle proprie identità e delle informazioni associate ad esse, avendone la possibilità di manipolare i dati in modo indipendente ma coerente, senza dover appoggiarsi a una terza entità centralizzata, ad esempio un registro anagrafe di stato sul quale ovviamente non si avrebbe potere decisionale.

### Soluzione e implementazione:

Il contratto *SimpleEthereumDIDRegistry* propone una soluzione basata sulla blockchain Ethereum (ma potrà essere caricato anche su Hedera) il cui codice viene mostrato come segue:

```

1  /* SPDX-License-Identifier: MIT */
2  pragma solidity >0.8.6;
3
4  contract SimpleEthereumDIDRegistry {
5
6      mapping(address => uint) public nonce;
7      mapping(address => uint) public changed;
8      mapping(address => address) public owners;
9      mapping(address => mapping(bytes32 => mapping(address => uint))) public delegates;
10
11
12     modifier onlyOwner(address identity, address actor) {
13         require(actor == identityOwner(identity), "bad_actor");
14     }
15
16
17     function identityOwner(address identity) public view returns(address) {    infinite gas
18         address owner = owners[identity];
19         if (owner != owner[identity]) {
20             return owner;
21         }
22         return identity;
23     }
24

```

```

25     function checkSignature()    infinite gas
26         address identity,
27         bytes32 sigV,
28         bytes32 sigR,
29         bytes32 sigS,
30         bytes32 hash)
31         internal
32         returns(address) {
33             address signer = ecrecover(hash, sigV, sigR, sigS);
34             require(signer == identityOwner(identity), "bad_signature");
35             nonce[signer]++;
36             return signer;
37         }
38
39     function validDelegate()    infinite gas
40         address identity,
41         bytes32 delegateType,
42         address delegate)
43         internal
44         view
45         returns(bool) {
46             uint validity = delegates[identity][keccak256(abi.encode(delegateType))][delegate];
47             return (validity > block.timestamp);
48         }

```

```

49     function changeOwner()    infinite gas
50         address identity,
51         address actor,
52         address newOwner)
53         internal
54         onlyOwner(identity, actor) {
55             owners[identity] = newOwner;
56             changed[identity] = block.number;
57         }
58
59
60     function changeOwner()    infinite gas
61         address identity,
62         address newOwner)
63         public {
64             changeOwner(identity, msg.sender, newOwner);
65         }
66
67     function changeOwnerSigned()    infinite gas
68         address identity,
69         uint8 sigV,
70         bytes32 sigR,
71         bytes32 sigS,
72         address newOwner)

```

```

73     modifier {
74         bytes32 hash = keccak256(abi.encodePacked(bytes1(0x10), bytes1(0), this, nonce[identityOwner(identity)]));
75         identityOwner(identity, nonceOwner);
76         changeOwner(identity, checkSignature(identity, sigV, sigR, sigS, hash, newOwner));
77     }
78
79     function addDelegate()    infinite gas
80         address identity,
81         address actor,
82         bytes32 delegateType,
83         address delegate,
84         uint validity)
85         internal
86         onlyOwner(identity, actor) {
87             delegates[identity][keccak256(abi.encode(delegateType))][delegate] = block.timestamp + validity;
88             changed[identity] = block.number;
89         }
90
91     function addDelegate()    infinite gas
92         address identity,
93         bytes32 delegateType,
94         address delegate,
95         uint validity)
96         public {
97             addDelegate(identity, msg.sender, delegateType, delegate, validity);
98         }
99
100    function addDelegateSigned()    infinite gas
101        address identity,
102        uint8 sigV,
103        bytes32 sigR,
104        bytes32 sigS,
105        bytes32 delegateType,
106        address delegate,
107        uint validity)
108        public {
109            bytes32 hash = keccak256(abi.encodePacked(bytes1(0x10), bytes1(0), this, nonce[identityOwner(identity)], identity,
110            "addDelegate", delegateType, delegate, validity));
111            addDelegate(identity, checkSignature(identity, sigV, sigR, sigS, hash), delegateType, delegate, validity);
112        }
113
114    }

```

Il mapping *owners* collega ogni identificatore *identity* al suo proprietario *owner*, mentre Il mapping *delegates* tiene traccia dei delegati autorizzati per un dato identificatore, in base al tipo di delega *delegateType* definita e all'indirizzo del delegato *delegate*. Il valore associato a un delegato determina la validità dell'autorizzazione;

Abbiamo altri due mapping, *nonce* e *changed*: il primo tiene traccia del valore di nonce associato a ciascun proprietario e viene incrementato ogni volta che il proprietario firma un messaggio; il secondo, tiene traccia del numero di blocco in cui è stata effettuata l'ultima modifica per ciascun identificatore e viene aggiornato quando viene cambiato il proprietario o viene aggiunto un delegato.

Il contratto include anche un modificatore chiamato *onlyOwner* che verifica se l'attore che invoca una funzione è il proprietario dell'identificatore corrispondente. Questo modificatore viene utilizzato per limitare l'accesso a funzioni che lavorano su dati sensibili.

La funzione *changeOwner* consente di cambiare il proprietario di un identificatore e può essere chiamata solo dal proprietario corrente o da un attore autorizzato tramite *onlyOwner*.

La funzione *addDelegate* consente di aggiungere un delegato autorizzato per un identificatore specifico; questo può essere aggiunto solo dal proprietario corrente o sempre da un attore autorizzato. Il delegato viene specificato con il tipo di delega *delegateType*, l'indirizzo del delegato *delegate* e la durata della validità dell'autorizzazione *validity*.

Abbiamo poi le funzioni *changeOwner* e *addDelegateSigned* che consentono rispettivamente il cambio di proprietario e l'aggiunta di un delegato attraverso la firma di un messaggio. Il mittente deve fornire una firma valida per dimostrare l'autorizzazione del proprietario corrente.

La funzione *checkSignature* verifica la firma di un messaggio utilizzando l'algoritmo di firma Ethereum, se il firmatario corrisponde al proprietario dell'identificatore e incrementa il valore *nonce* associato al proprietario per evitare la riutilizzazione delle firme.

Infine, la funzione *validDelegate* verifica se un delegato specifico è ancora valido per un determinato identificatore e tipo di delega; restituisce un valore booleano che indica se l'autorizzazione è ancora valida in base al timestamp corrente.

## 13) Payment Splitter

### Problema:

Si potrebbe avere la necessità di suddividere i pagamenti tra più destinatari in modo automatico e trasparente, ad esempio un'azienda che deve pagare dei dividendi ai propri azionisti, i quali li riceveranno in proporzione alle loro partecipazioni. Ci potrebbero essere degli errori o delle iniquità nei confronti degli investitori se ciò avvenisse manualmente.

### Soluzione e implementazione:

Potremmo ancora una volta sfruttare la trasparenza della blockchain o comunque sia dei registri distribuiti decentralizzati servendoci di uno smart contract che funzioni gestisca questa funzionalità.

```

1 // SPDX-License-Identifier: MIT
2 // OpenZeppelin Contracts (last updated v4.8.0) (finance/PaymentSplitter.sol)
3
4 pragma solidity ^0.8.0;
5
6 contract PaymentSplitter {
7
8     uint256 private _totalShares;
9     uint256 private _totalReleased;
10
11    mapping(address => uint256) private _shares;
12    mapping(address => uint256) private _released;
13    address[] private _payees;
14
15    constructor(address[] memory payees, uint256[] memory shares_) payable {
16        require(payees.length == shares_.length, "PaymentSplitter: payees and shares length mismatch");
17        require(payees.length > 0, "PaymentSplitter: no payees");
18
19        for (uint256 i = 0; i < payees.length; i++) {
20            _addPayee(payees[i], shares_[i]);
21        }
22    }
23
24    receive() external payable virtual {}
25
26    function totalShares() public view returns (uint256) {
27        return _totalShares;
28    }
29
30    function totalReleased() public view returns (uint256) {
31        return _totalReleased;
32    }
33
34    function shares(address account) public view returns (uint256) {
35        return _shares[account];
36    }
37
38    function released(address account) public view returns (uint256) {
39        return _released[account];
40    }
41
42    function payee(uint256 index) public view returns (address) {
43        return _payees[index];
44    }
45
46    function releasable(address account) public view returns (uint256) {
47        uint256 totalReceived = address(this).balance + totalReleased();
48        return _pendingPayment(account, totalReceived, released(account));
49    }
50
51
52    function release(address payable account) public virtual {
53        require(_shares[account] > 0, "PaymentSplitter: account has no shares");
54
55        uint256 payment = releasable(account);
56
57        require(payment != 0, "PaymentSplitter: account is not due payment");
58
59        // _totalReleased is the sum of all values in _released.
60        // If "_totalReleased + payment" does not overflow, then "_released[account] + payment" cannot overflow.
61        unchecked {
62            _released[account] += payment;
63        }
64
65        (bool success,) = account.call{value: payment}("");
66        require(success);
67    }
68
69    function _pendingPayment(
70        address account,
71        uint256 totalReceived,
72        uint256 alreadyReleased
73    ) private view returns (uint256) {
74        return (totalReceived * _shares[account]) / _totalShares - alreadyReleased;
75    }
76
77    function _addPayee(address account, uint256 shares_) private {
78        require(account != address(0), "PaymentSplitter: account is the zero address");
79        require(shares_ > 0, "PaymentSplitter: shares are 0");
80        require(_shares[account] == 0, "PaymentSplitter: account already has shares");
81
82        _payees.push(account);
83        _shares[account] = shares_;
84        _totalShares = _totalShares + shares_;
85    }
86

```

Il contratto PaymentSplitter così implementato semplifica e automatizza la suddivisione dei pagamenti tra più destinatari in base alle quote specificate.

Al momento della creazione del contratto, che utilizza alcune funzioni e modificatori di OpenZeppelin Contracts, vengono specificati gli indirizzi *payees* dei destinatari dei pagamenti e le rispettive quote *shares*; le quote dei pagamenti vengono assegnate ai destinatari il cui totale viene calcolato e poi memorizzato nella variabile *\_totalShares*.

Payment Splitter può ricevere pagamenti attraverso la funzione *receive()* *external payable* i quali verranno memorizzati nello stesso.

I destinatari possono controllare la loro quota di pagamenti tramite la funzione *shares(address account)*, che restituisce il numero di quote assegnate all'account specificato.

I destinatari possono richiedere il pagamento dei fondi che sono loro dovuti tramite la funzione *release(address payable account)*, che calcola l'importo dei fondi che sono disponibili per il destinatario in *releasable(account)* e lo invia all'account specificato.

La funzione *releasable(address account)* calcola l'importo dei fondi che sono disponibili per il destinatario specificato, che dipende dalla somma totale dei fondi ricevuti dal contratto e dalla quantità di fondi già rilasciati al destinatario.

PaymentSplitter tiene traccia dei fondi rilasciati a ciascun destinatario nella mappatura *\_released*, del totale delle quote *totalShares*, del totale dei fondi rilasciati *totalReleased* e l'indirizzo del destinatario in base all'indice *payee(index)*.

## 14) TinyAmm

### **Problema:**

*Lo scambio di asset, comprese le criptovalute, è reso possibile tramite piattaforme centralizzate le quali non soffrono, generalmente, di problemi di liquidità, possedendo loro stesse le risorse da scambiare e quindi intervenendo in qualità di mediatori tra domanda e offerta (Market Maker); la stessa cosa non si può dire per piattaforme decentralizzate, che richiedono un ordine di acquisto e uno di vendita da parte di due parti diverse per eseguire lo scambio; questo, al contrario, può portare a problemi di liquidità, in cui può non esserci abbastanza domanda per eseguire gli scambi in maniera efficiente.*

### **Soluzione e implementazione:**

Il contratto TinyAmm risolve il problema fornendo un meccanismo automatizzato per lo scambio di token senza la necessità di un ordine di acquisto e un ordine di vendita da parte di utenti diversi (Automated Market Maker). Utilizza invece, un modello di liquidità basato su un rapporto predefinito tra due token, consentendo agli utenti di depositarli e partecipare alle pool di liquidità.

```
1 // SPDX-License-Identifier: UNLICENSED
2 pragma solidity ^0.8.18;
3
4 contract TinyAMM {
5     IERC20 public immutable t0;
6     IERC20 public immutable t1;
7
8     uint public r0;
9     uint public r1;
10
11    bool ever_deposited;
12    uint public supply;
13    mapping(address => uint) public minted;
14
15    constructor(address t0_, address t1_) {    █ infinite
16        t0 = IERC20(t0_);
17        t1 = IERC20(t1_);
18    }
19
20    function deposit(uint x0, uint x1) public {    █ infinite
21        require(x0>0 && x1>0);
22
23        t0.transferFrom(msg.sender, address(this), x0);
24        t1.transferFrom(msg.sender, address(this), x1);
25
26        uint toMint;
27
28        if (ever_deposited) {
29            require(r0 * x1 == r1 * x0);
30            toMint = (x0 * supply) / r0;
31        }
32        else {
33            ever_deposited = true;
34            toMint = x0;
35        }
36
37        require(toMint > 0);
38
39        minted[msg.sender] += toMint;
40        supply += toMint;
41        r0 *= x0;
42        r1 *= x1;
43
44        assert(t0.balanceOf(address(this)) == r0);
45        assert(t1.balanceOf(address(this)) == r1);
46    }
47
48    function redeem(uint x) public {    █ infinite
49        require(minted[msg.sender] >= x);
50        require(x < supply);
51
52        uint x0 = (x * r0) / supply;
53
54        uint x1 = (x * r1) / supply;
55
56        t0.transferFrom(address(this), msg.sender, x0);
57        t1.transferFrom(address(this), msg.sender, x1);
58
59        r0 *= x0;
60        r1 *= x1;
61        supply -= x;
62        minted[msg.sender] -= x;
63
64        assert(t0.balanceOf(address(this)) == r0);
65        assert(t1.balanceOf(address(this)) == r1);
66    }
67
68    function swap(address t, uint x_in, uint x_out_min) public {
69        require(t == address(t0) || t == address(t1));
70        require(x_in > 0);
71
72        bool is_t0 = t == address(t0);
73        (IERC20 t_in, IERC20 t_out, uint r_in, uint r_out) = is_t0
74            ? (t0, t1, r0, r1)
75            : (t1, t0, r1, r0);
76
77        t_in.transferFrom(msg.sender, address(this), x_in);
78
79        uint x_out = x_in * r_out / (r_in + x_in);
80
81        require(x_out >= x_out_min);
82
83        t_out.transfer(msg.sender, x_out);
84
85        (r0, r1) = is_t0
86            ? (r0 + x_in, r1 - x_out)
87            : (r0 - x_out, r1 + x_in);
88
89        assert(t0.balanceOf(address(this)) == r0);
90        assert(t1.balanceOf(address(this)) == r1);
91    }
92
93    interface IERC20 {
94        event Transfer(address indexed from, address indexed to, uint256 value);
95
96        event Approval(address indexed owner, address indexed spender, uint256 value);
97
98        function totalSupply() external view returns (uint256);    █ - gas
99
100       function balanceOf(address account) external view returns (uint256);    █ - gas
101
102       function transfer(address to, uint256 amount) external returns (bool);    █ - gas
103
104       function allowance(address owner, address spender) external view returns (uint256);    █ - gas
105
106       function approve(address spender, uint256 amount) external returns (bool);    █ - gas
107
108       function transferFrom(address from, address to, uint256 amount) external returns (bool);    █ - gas
109    }
110
```

Le principali funzioni di TinyAMM sono le seguenti:

La *deposit*, che prende in ingresso due token, consente agli utenti di depositare una quantità specifica di token t0 e t1; questi vengono trasferiti dal mittente al contratto. Se è la prima volta che viene effettuato un deposito, il rapporto di liquidità tra i due token viene impostato in base alla quantità depositata di t0; se invece sono stati effettuati depositi precedenti, viene verificato che il rapporto di liquidità rimanga costante e viene calcolata la quantità di token t0 da “creare” per l’utente. Vengono poi aggiornati i saldi, il rapporto di liquidità e le quantità di token creati per l’utente.

La *redeem* permette agli utenti di riscattare una quantità specifica di token creati in precedenza tramite il deposito; i token vengono trasferiti dal contratto all'utente in base alla quantità specificata. Infine vengono aggiornati i saldi, il rapporto di liquidità e le quantità di token “creati” per l'utente.

Lo *swap* consente agli utenti di scambiare uno dei due token t0 o t1 con l'altro. L'utente specifica il token di input, la quantità in input e la quantità minima di token di output richiesta. Viene a questo punto effettuato un calcolo per determinare la quantità di token di output in base al rapporto di liquidità attuale tra i due token: se la quantità di token di output è superiore o uguale alla quantità minima richiesta, viene effettuato lo scambio. Infine, come per la *redeem*, verranno aggiornati i saldi, il rapporto di liquidità e le quantità di token coinvolti nello scambio.

### 15) Upgradable Proxy

### **Problema:**

*Questo smart contract mira a risolvere un problema interno all'ecosistema blockchain, e non quindi uno per il quale, rispetto agli altri, è stata proposta la tecnologia dei registri distribuiti decentralizzati come soluzione: a volte è necessario apportare modifiche e miglioramenti all'interno degli smart contracts (per esempio correzione di bug) ma allo stesso tempo si vorrebbe garantire una continuità del servizio e inoltre che non sia necessario creare nuove istanze del contratto o richiedere agli utenti di migrare i loro fondi e dati.*

## *Soluzione e implementazione:*

Il contratto *TheProxy* e relative implementazioni, consentono di creare un proxy aggiornabile che sarà utilizzato per separare la logica del contratto dalle funzionalità di amministrazione e aggiornamento; ciò consente dunque di aggiornare il contratto senza dover modificare direttamente il proxy stesso, facilitando così la manutenzione e l'aggiornamento del contratto. Di seguito il codice:

```
1 // SPDX-License-Identifier: MIT
2 // Adapted from OpenZeppelin Contracts (last updated v4.6.0) (proxy/Proxy.sol)
3 // This code implements a simple upgradable Proxy (no beacon) with a non-upgradeable admin
4 // reference: https://docs.openzeppelin.com/contracts/4.x/api/proxy
5
6 pragma solidity >=0.8.0;
7
8
9 library StorageSlot {
10     struct AddressSlot {
11         address value;
12     }
13
14     function getAddressSlot(bytes32 slot) internal pure returns (AddressSlot storage r) {
15         assembly {
16             r.slot := slot
17         }
18     }
19
20
21 library Address{
22
23     function isContract(address account) internal view returns (bool) {    - gas
24         return account.code.length > 0;
25     }
26
27
28     function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
29         return functionDelegateCall(target, data, "Address: low-level delegate call failed");
30     }
31
32     function functionDelegateCall(  address target,
33         bytes memory data,
34         string memory errorMessage)
35         internal returns (bytes memory) {
36         (bool success, bytes memory returnData) = target.delegatecall(data);
37         require(success, errorMessage);
38         return returnData;
39     }
40
41
42     abstract contract Proxy {
43
44         function _delegate(address implementation) internal virtual {    - gas
45             assembly {
46                 calldatcopy(0, 0, calldatasize())
47                 let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)
48                 returndatcopy(0, 0, returndatasize())
49                 switch result
50                 case 0 {
51                     revert(0, 0)
52                 }
53             }
54         }
55
56         default {
57             revert(0, 0)
58         }
59     }
60
61     case 0 {
62         revert(0, returndatasize())
63     }
64
65     function _implementation() internal view virtual returns (address);
66
67     function _fallback() internal virtual {    - gas
68         _beforeFallback();
69         _delegate(_implementation());
70     }
71
72     fallback() external payable virtual {    - gas
73         _fallback();
74     }
75
76     receive() external payable virtual {    - gas
77         _fallback();
78     }
79
80     function _beforeFallback() internal virtual {}    - gas
81
82
83 abstract contract SimplifiedERC1967Upgrade {
84
85     event Upgraded(address indexed implementation);
86
87     bytes32 internal constant _IMPLEMENTATION_SLOT = 0x360894a13ba1e3120667c82492d98cda3e2076cc3735e920a3ca505d382bbc;
88     bytes32 internal constant _ADMIN_SLOT = 0xb53127684a568b3173ae13b9f8a6b1e243e53b6e8ee1178d6171785b05d6105;
89
90
91     function _getImplementation() internal view returns (address) {    - gas
92         return StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value;
93     }
94
95     function _setImplementation(address newImplementation) private {    - gas
96         require(Address.isContract(newImplementation), "ERC1967: new implementation is not a contract");
97         StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value = newImplementation;
98     }
99
100
101     function _upgradeTo(address newImplementation) internal {    - gas
102         _setImplementation(newImplementation);
103         emit Upgraded(newImplementation);
104     }
105
106
107     contract Logic {
108
109         // @dev returns true if the balance of the _toCheck address is lower than 100
110         function check(address _toCheck) public view returns(bool) {    - gas
111             if (_toCheck.balance < 100) return true;
112             return false;
113         }
114
115     }
116
117     function functionDelegateCall(address target, bytes memory data) internal returns (bytes memory) {
118         return functionDelegateCall(target, data, "Logic: low-level delegate call failed");
119     }
120
121     function functionDelegateCall(  address target,
122         bytes memory data,
123         string memory errorMessage)
124         internal returns (bytes memory) {
125         (bool success, bytes memory returnData) = target.delegatecall(data);
126         require(success, errorMessage);
127         return returnData;
128     }
129
130
131     function getAdmin() public view returns (address) {    - gas
132         return _getAdmin();
133     }
134
135     function upgradeTo(address newImplementation) public {    - gas
136         _upgradeTo(newImplementation);
137     }
138
139
140     contract Caller{
141
142         // @dev This function calls the logic function "check" passing its same address.
143         function callingLogic(address _proxy) public returns(bool,bool) {    - gas
144             string memory abi = "check(address)";
145             address param = address(this);
146             bytes memory payload = abi.encodeWithSignature(abi,param);
147             (bool success, bytes memory result) = _proxy.call(payload);
148             if (result.length>1) == byte(0x01) return (success,true);
149             return (success,false);
150         }
151
152     }
153 }
```

*TheProxy* segue il modello di upgrade semplificato del contratto *SimplifiedERC1967Upgrade*. Il proxy consente di indirizzare le chiamate a un’implementazione specifica e può essere aggiornato per indirizzare le chiamate a una nuova implementazione, senza modificare il proxy stesso; questo conserva uno slot di archiviazione per l’indirizzo dell’implementazione corrente e uno slot di archiviazione per l’amministrazione del proxy, che può essere impostato utilizzando la funzione *setAdmin*.

L'implementazione attuale invece, viene ottenuta utilizzando la funzione `_getImplementation` definita nel contratto `SimplifiedERC1967Upgrade`.

Il contratto *Caller* contiene una funzione *callLogicByProxy* che chiama la funzione *check* dell’implementazione tramite il proxy: viene creato un payload che rappresenta la chiamata della funzione *check* avente l’indirizzo del chiamante come parametro. Il payload viene quindi passato al proxy tramite la funzione *call* e viene restituito il risultato.

Infine, il contratto *Logic*, rappresenta un’implementazione di esempio con una funzione *check* che restituisce true se il saldo dell’indirizzo *toCheck* è inferiore a 100, altrimenti restituisce false.

## SEZIONE 3

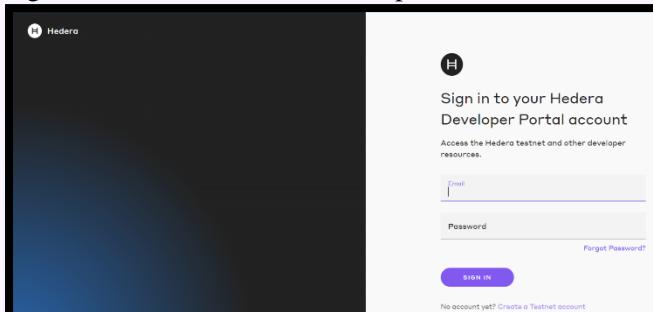
### IMPOSTAZIONE DELL'AMBIENTE DI SVILUPPO HEDERA E DEPLOY DEGLI SMART CONTRACT

In questa sezione spiegheremo in maniera dettagliata, attraverso i vari step, come sia possibile interagire con la rete Hedera, dalla creazione di un account di test allo sviluppo di un ambiente software per eseguire i contratti, fino al caricamento degli stessi nel Network al fine di renderli usufruibili nella loro interezza e quindi essere in grado di valutare i costi di deploy.

#### Step 1: Creazione e definizione di un account sulla testnet di Hedera:

Per poter interagire con la rete Hedera abbiamo necessariamente bisogno di possedere un **account**; a questo, viene associato un **ID** i quali nella rete Hedera rappresentano specifici oggetti composti (come appunto account, ma anche token, file, smart contract ecc.); ad esempio, se prendessimo in esame l'ID “**0.0.1234**”, sappiamo che il primo “**0**” rappresenta l'identificatore ID dello **shard** (frammento) a cui appartiene l'oggetto, il secondo “**0**” rappresenta l'identificatore ID del **relam** (regno) ed infine la stringa “**1234**” rappresenta l'**object**, quindi l'oggetto specifico.

A primo impatto, questa è una delle differenze che prima saltano all'occhio rispetto Ethereum, nel quale gli account sono identificati tramite degli **address**, delle stringhe alfanumeriche che iniziano con “**0x**” le quali rappresentano la chiave pubblica (a sua volta generata a partire da una chiave privata). Naturalmente, trattandosi di un registro basato su crittografia, anche Hedera utilizza il concetto di **chiavi crittografiche asimmetriche** per l'autenticazione e la firma delle transazioni, le quali semplicemente saranno associate a un certo ID fornito dalla Hedera foundation tramite i tool messi da loro a disposizione. Per ottenere il proprio ID relativo ad un account della rete di test, sarà sufficiente recarsi all'indirizzo web **portal.hedera.com** registrandosi attraverso una mail personale, come mostrato di seguito:



Una volta fatto ciò, dopo aver confermato la registrazione, potremmo accedere nella nostra area riservata che avrà il seguente aspetto:

ACCOUNT ECDSA	Refill in 21:40
EVM Address 0x7b72dfb7b024a664c8ecfaeb1705c9a83fc89bf	
HEX Encoded Private Key 0x27a2fb0f56a41b9d3200328b2ff8ef02992a8a230...	
DER Encoded Private Key 303020103030506032b657004220420b4a3e4a3b8a...	
DER Encoded Public Key 302d300706052b8104000e032200025b7d28b7c6ff378...	
Account ID 0.0.14817852	

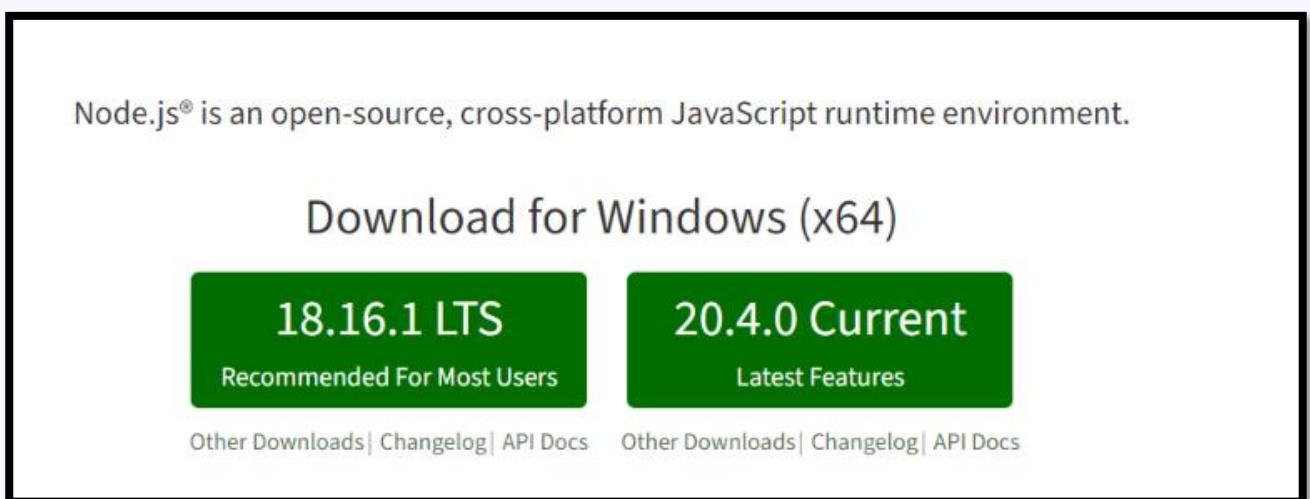
Come si evince dall'immagine, possiamo accedere alle nostre informazioni relative all'**account di test** poiché abbiamo selezionato la voce **testnet**. Volendo, si potrebbe richiedere anche l'**account di previewnet**, una rete più vicina alla **mainnet** (il network vero e proprio) in termine di funzionalità, molte delle quali ancora in fase di sperimentazione e sviluppo motivo per cui si è scelto di procedere per la **testnet**, teoricamente più stabile e più inerente agli scopi del presente studio. Nella pagina della testnet mostrata,

vengono forniti due tipi di account (associati a due **ID differenti**): il campo “**ED25519**” si riferisce all’algoritmo di firma **EdDSA** basato sulla curva ellittica ed25519, mentre il campo “**ECDSA**” si riferisce all’algoritmo di firma basato sulla curva ellittica **ECDSA (Elliptic Curve Digital Signature Algorithm)**. Per ciascuno di essi, avremo associate le rispettive **chiavi pubbliche e private**. Nonostante siano entrambi degli algoritmi crittografici robusti, ci siamo accorti che la rete non supporta temporaneamente algoritmi di firma ECDSA, per questo motivo abbiamo potuto procedere utilizzando esclusivamente il primo tipo di cifratura. Si fa notare che, poiché per testare alcuni smart contract abbiamo avuto bisogno di più account, necessariamente si è dovuto registrarsi con altre due mail differenti. Infine, a differenza di un account di **Mainnet** il quale necessita di essere “ricaricato” comprando degli HBAR a mercato nelle relative piattaforme di scambio, gli **account di test** vengono automaticamente “riempiti” dal tool di Hedera con degli HBAR finti allo scadere di un certo periodo di tempo indicato dal **refill**, proprio perché, essendo una suite di software pensati per fare delle prove, non avrebbe senso disincentivare le sperimentazioni facendo spendere denaro agli sviluppatori che invece possono ampiamente fare tutte le prove che desiderano, in quanto gli account presentano già al loro interno **10'000 HBAR**.

### Step 2: Installazione di Node.js

Per poter sviluppare smart contracts abbiamo utilizzato il linguaggio di programmazione tipizzato “**Solidity**” ma per interagire col registro distribuito di Hedera, abbiamo bisogno di **Node.js**, un ambiente di runtime JavaScript open-source per eseguire applicazioni sia client-side che server-side che appunto sono in grado di interagire con gli smart contracts.

Per installare Node.js, sarà sufficiente recarsi alla pagina web <http://nodejs.org>, scaricare la versione *LTS* (la più stabile) compatibile col Sistema Operativo ed installare il pacchetto.



L’installazione includerà anche il gestore dei pacchetti **npm** (*Node Package Manager*) il quale svolgerà un ruolo essenziale per lo scopo dello studio in quanto gestirà, tra le altre cose, le dipendenze per ogni progetto, permettendoci di specificare le librerie, i framework e gli strumenti necessari per lo sviluppo degli smart contracts occupandosi quindi di installarli correttamente.

Una volta fatto ciò, sarà sufficiente aprire il *prompt* dei comandi per verificare la corretta installazione dei software e la relativa versione:

A screenshot of a Windows Command Prompt window titled "Prompt dei comandi". The window shows the following text:

```
Prompt dei comandi
Microsoft Windows [Versione 10.0.19045.2965]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Elge> node -v
v18.16.1

C:\Users\Elge> npm -v
9.5.1

C:\Users\Elge>
```

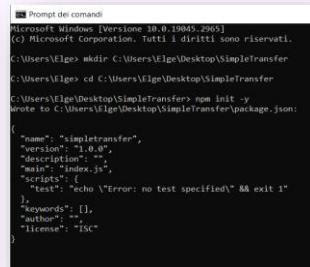
### Step 3: Creazione di un ambiente di sviluppo per testare gli Smart Contracts

Per prima cosa creiamo, da terminale, una cartella avente il nome dello smart contract da deployare:



```
Prompt dei comandi
Microsoft Windows [Versione 10.0.19045.2965]
(c) Microsoft Corporation. Tutti i diritti sono riservati.
C:\Users\Elge> mkdir C:\Users\Elge\Desktop\SimpleTransfer
```

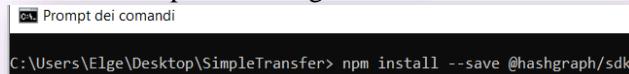
dentro la quale inseriremo il file del contratto scritto in Solidity, con estensione .sol. Fatto ciò, inizializzeremo un nuovo progetto Node.js per creare automaticamente, all'interno della cartella SimpleTransfer, un file “**package.json**” con le impostazioni predefinite:



```
Prompt dei comandi
Microsoft Windows [Versione 10.0.19045.2965]
(c) Microsoft Corporation. Tutti i diritti sono riservati.
C:\Users\Elge> mkdir C:\Users\Elge\Desktop\SimpleTransfer
C:\Users\Elge> cd C:\Users\Elge\Desktop\SimpleTransfer
C:\Users\Elge> npm init -y
wrote to C:\Users\Elge\Desktop\SimpleTransfer\package.json:

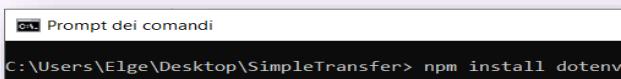
{
  "name": "simpletransfer",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "+",
  "license": "ISC"
}
```

A questo punto installeremo le dipendenze e gli **SDK**:



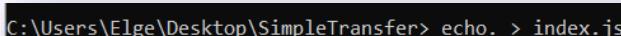
```
Prompt dei comandi
C:\Users\Elge\Desktop\SimpleTransfer> npm install --save @hashgraph/sdk
```

e tramite il comando *npm install dotenv* installeremo il pacchetto **Dotenv** nel nostro progetto, una libreria comunemente utilizzata per memorizzare e gestire le variabili di ambiente sensibili della nostra applicazione ovvero gli *ID* e le *chiavi private* con le quali firmeremo le nostre transazioni. Aggiorna l'**SDK di Hedera Hashgraph** eseguendo il comando *npm install @hashgraph/sdk@latest* o *npm update @hashgraph/sdk*, mentre per verificare la versione *npm ls @hashgraph/sdk*.



```
Prompt dei comandi
C:\Users\Elge\Desktop\SimpleTransfer> npm install dotenv
```

Tornando poi nella directory principale del progetto, creiamo un file *index.js* attraverso il comando



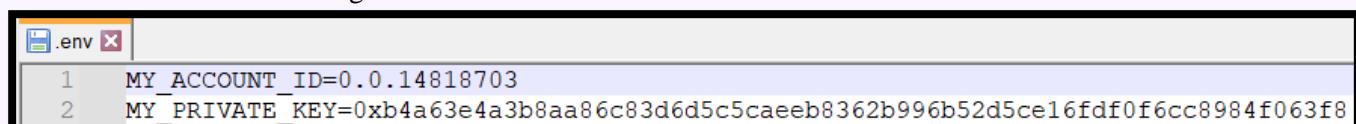
```
C:\Users\Elge\Desktop\SimpleTransfer> echo. > index.js
```

Il quale ci servirà più avanti per inserire codice JavaScript.

Non ci resta che creare il nostro file *.env* che come già anticipato, conterrà l'*ID* e la *chiave privata* (o più ID e chiavi, a seconda di quanti account interagiranno con il contratto) precedentemente forniti nello step 1, e inseriti secondo la seguente formattazione:

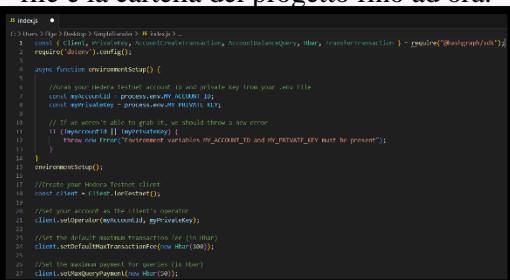
**MY\_ACCOUNT\_ID=ENTER TESTNET ACCOUNT ID**  
**MY\_PRIVATE\_KEY=ENTER TESTNET PRIVATE KEY**

Il risultato sarà il seguente:



```
.env
1 MY_ACCOUNT_ID=0.0.14818703
2 MY_PRIVATE_KEY=0xb4a63e4a3b8aa86c83d6d5c5caeb8362b996b52d5ce16fdf0f6cc8984f063f8
```

Possiamo adesso popolare il file *index.js* con lo script fornito dalla documentazione, aggiungendo il codice che ci permetterà di avere il nostro Client per la Hedera Testnet; di seguito, mostriamo come si presenta il file e la cartella del progetto fino ad ora.



```
index
1 //Create your Hedera Testnet client
2 const { Client, Operator, Account, Transaction, AccountTransaction, TransferTransaction } = require("@hashgraph/sdk");
3
4
5 async function environmentSetup() {
6
7   //With your Hedera Testnet account ID and private key from your .env file
8   const myAccountId = process.env.MY_ACCOUNT_ID;
9   const myPrivateKey = process.env.MY_PRIVATE_KEY;
10
11   //If an attempt is made to grab it, we should throw a new error
12   if (!myAccountId || !myPrivateKey) {
13     throw new Error("Environment variables MY_ACCOUNT_ID and MY_PRIVATE_KEY must be present");
14   }
15
16   environmentSetup();
17
18   //Create your Hedera Testnet client
19   const client = new Client();
20
21   //Set your account as the client's operator
22   client.operator(myAccountId, myPrivateKey);
23
24   //Set the default maximum transaction fee (in nano)
25   client.setDefaultMaxTransactionFee(1000000);
26
27   //Set the maxima payout for queries (in nano)
28   client.setMaxPayout(10000000);
```



Nome	Ultima modifica	Tipo	Dimensione
node_modules	06/07/2023 17:55	Cartella di file	
index	06/07/2023 18:03	File di origine Java...	1 KB
package	06/07/2023 17:55	File JSON	1 KB
package-lock	06/07/2023 17:55	File JSON	74 KB
SimpleTransfer.sol	17/06/2023 16:53	File SOL	1 KB
.env	06/07/2023 11:56	File ENV	1 KB

#### **Step 4: Approccio modulare e deploy dello smart contract**

Abbiamo visto come impostare, per un determinato file .sol, l'**ambiente di sviluppo** seguendo l'indicazione generica fornita dalla Hedera foundation; questa però non è sempre la strada più praticabile. Infatti, man mano che abbiamo a che fare con un contratto il cui codice cresce di complessità, avremo bisogno di uno script per ogni funzionalità, ed ecco che avere un solo file **index.js** che gestisca ogni metodo, non è la scelta migliore, nemmeno per un contratto relativamente semplice come il Simple Transfer.

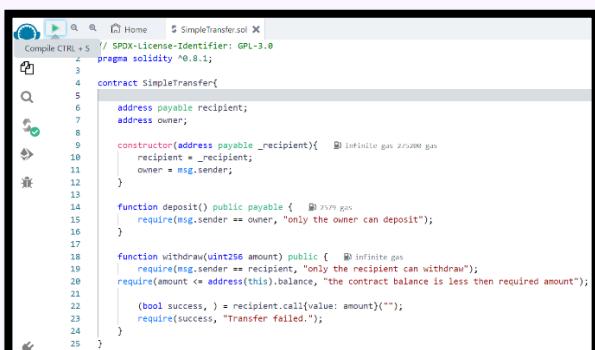
È più opportuno invece, avere un approccio alla programmazione più **modulare** per ogni interazione attraverso la quale scambiare informazioni col contratto, a partire dal **bytecode**. Il bytecode in Solidity è la **rappresentazione binaria** del codice sorgente dopo essere stato compilato da parte di un compilatore (ad esempio il solc), quindi istruzioni a basso livello eseguite dalla EVM e caricate nel registro distribuito definendo le regole e il comportamento del contratto quando viene eseguito nello stesso.

Parallelamente, l'**ABI** (Application Binary Interface) è l'interfaccia che definisce la struttura e le specifiche delle funzioni e degli eventi del contratto, fornendo una descrizione dettagliata di come interagirci inclusi i tipi di dati dei parametri delle funzioni, i tipi di dati restituiti e via dicendo.

Queste informazioni saranno presenti nel file **JSON** (Javascript Object Notation) utilizzato per rappresentare l'intera struttura dati del contratto:

Possiamo ottenerlo a partire dal file .sol servendoci dell'**IDE Remix** web, poi procedere nel seguente ordine:

#### 4.1) Compilazione del contratto



```
SPDX-License-Identifier: GPL-3.0+
pragma solidity ^0.8.1;

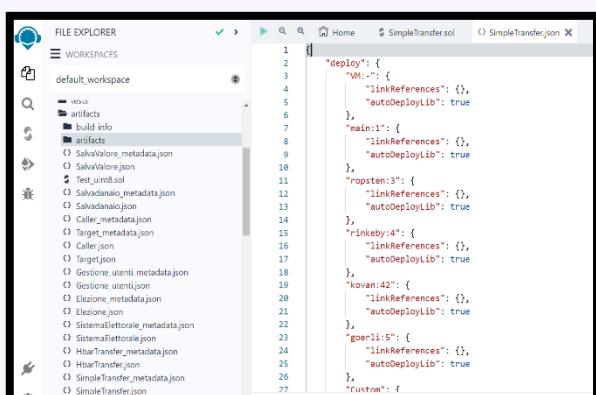
contract SimpleTransfer {
    address payable recipient;
    address owner;

    constructor(address payable _recipient) {
        recipient = _recipient;
        owner = msg.sender;
    }

    function deposit() public payable {
        require(msg.sender == owner, "only the owner can deposit");
    }

    function withdraw(uint256 amount) public {
        require(msg.sender == recipient, "only the recipient can withdraw");
        require(amount <= address(this).balance, "the contract balance is less than required amount");
        (bool success,) = recipient.call{value: amount}("");
        require(success, "Transfer failed.");
    }
}
```

#### 4.2) Estrazione del file JSON dalla cartella “artifacts”



#### 4.3) Salvataggio del file JSON nella cartella del Simple Transfer per la quale abbiamo costruito l’ambiente

Nome	Ultima modifica	Tipo	Dimensione
node_modules	06/07/2023 17:55	Cartella di file	
.env	06/07/2023 11:56	File ENV	1 KB
index	06/07/2023 18:03	File di origine Java...	1 KB
package	06/07/2023 17:55	File JSON	1 KB
package-lock	06/07/2023 17:55	File JSON	74 KB
SimpleTransfer.sol	17/06/2023 16:53	File SOL	1 KB
SimpleTransfer	07/07/2023 15:54	File JSON	114 KB

#### 4.4) Costruzione di un file “store” da aggiungere alla cartella

Il file **store** ci consente di prelevare il **bytecode** presente nel file **JSON** appena ottenuto e ci consentirà di caricarlo correttamente nella rete Hedera con la codifica in esadecimale attraverso la funzione dell’SDK “*FileCreateTransaction*”, come indicato dalle guide ufficiali del sito.

Per giungere a questo passaggio fondamentale ci è stato di aiuto consultare la guida all’indirizzo web <https://docs.hedera.com/hedera/tutorials/smart-contracts/deploy-your-first-smart-contract> la quale suggerisce come procedere dato uno smart contract generico, in questo caso *HelloHedera*, da loro fornito.

Inoltre, oltre che caricare il bytecode, ci viene fornito anche il codice javascript, da inserire nel file **store**, per creare un’istanza del contratto, attraverso la funzione dell’SDK “*ContractCreateTransaction*”, la quale ci servirà anche per determinare il costo di distribuzione dello smart contract all’interno della rete, che a sua volta dipenderà non solo dal bytecode stesso ma anche dal costo delle transazioni date dalle tariffe della rete (*gas*);

È stato quindi sufficiente modificare il codice javascript da loro fornito creando la dipendenza per il nostro specifico contratto:

Script **deposit.js**:

```
JS store.js ×
C: > Users > Elge > Desktop > SimpleTransfer > JS store.js > ...
1  require("dotenv").config();
2  const {
3    Client,
4    PrivateKey,
5    FileCreateTransaction,
6    Hbar,
7  } = require("@hashgraph/sdk");
8
9  const operatorPrivateKey = PrivateKey.fromString(
10  | process.env.MY_PRIVATE_KEY
11 );
12 const operatorId = process.env.MY_ACCOUNT_ID;
13 let client = client.forTestnet();
14 client.setOperator(operatorId, operatorPrivateKey);
15
16 // Set the default maximum transaction fee (in Hbar)
17 client.setDefaultMaxTransactionFee(new Hbar(100));
18
19 // Set the maximum payment for queries (in Hbar)
20 client.setMaxQueryPayment(new Hbar(50));
21
22 //Import the compiled contract from the SimpleTransfer.json file
23 let dataStorage = require("./SimpleTransfer.json");
24 const bytecode = dataStorage.data.bytecode.object;
25
26 async function main() {
27   //Create a file on Hedera and store the hex-encoded bytecode
28   const fileCreateTx = new FileCreateTransaction()
29     //Set the bytecode of the contract
30     .setContents(bytecode);
31
32   //Submit the file to the Hedera test network signing with the transaction fee payer key specified with the client
33   const submitTx = await fileCreateTx.execute(client);
34
35   //Get the receipt of the file create transaction
36   const fileReceipt = await submitTx.getReceipt(client);
37
38   //Get the file ID from the receipt
39   const bytecodeFileId = fileReceipt.fileId;
40
41   //Log the file ID
42   console.log("The smart contract bytecode file ID is " + bytecodeFileId);
43 }
44
45 main().catch((error) => {
46   console.error("Error:", error);
47 });
48
```

Nome	Ultima modifica	Tipo	Dimensione
node_modules	06/07/2023 17:55	Cartella di file	
.env	06/07/2023 11:56	File ENV	1 KB
index	06/07/2023 18:03	File di origine Java...	1 KB
package	06/07/2023 17:55	File JSON	1 KB
package-lock	06/07/2023 17:55	File JSON	74 KB
SimpleTransfer	07/07/2023 15:54	File JSON	114 KB
SimpleTransfer.sol	17/06/2023 16:53	File SOL	1 KB
store	05/07/2023 15:55	File di origine Java...	2 KB

#### 4.5) Costruzione dei file deploy, deposit, withdraw e balance

Servendoci della documentazione fornita, abbiamo creato lo script che ci permette di distribuire il contratto, ovvero il file *deploy*, e per il discorso di avere un progetto modulare, anche gli script relativi ai metodi *deposit*, *withdraw* e *balance* (quest'ultimo a supporto per verificare il saldo del contratto), che di fatto sostituiscono il file *index.js*.

Script *deploy.js*:

```
JS deploy.js X
C: > Users > Elge > Desktop > SimpleTransfer > JS deploy.js > ...
1 const [
2   Client,
3   ContractCreateTransaction,
4   ContractFunctionParameters,
5   PrivateKey,
6   Hbar,
7 ] = require("@hashgraph/sdk");
8
9 require("dotenv").config();
10
11 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
12 const operatorId = process.env.MY_ACCOUNT_ID;
13
14 let client = Client.forTestnet();
15 client.setOperator(operatorId, operatorPrivateKey);
16
17 client.setDefaultMaxTransactionFee(new Hbar(100));
18 client.setMaxQueryPayment(new Hbar(50));
19
20 function toBytes32(accountId) {
21   const buffer = Buffer.from(accountId, 'utf8');
22   if (buffer.length > 32) {
23     throw new Error('Account ID is too long to fit in bytes32');
24   }
25   const bytes32 = Buffer.alloc(32);
26   buffer.copy(bytes32);
27   return bytes32;
28 }
29
30 async function deployContract(bytecode fileId, recipientAddress) {
31   const contractTx = new ContractCreateTransaction()
32     .setBytecode fileId(bytecode fileId)
33     .setGas(5000000)
34     .setConstructorParameters(new ContractFunctionParameters().addBytes32(toBytes32(recipientAddress)));
35   const contractResponse = await contractTx.execute(client);
36   const contractReceipt = await contractResponse.getReceipt(client);
37
38   const newContractId = contractReceipt.contractId;
39   console.log("The smart contract ID is " + newContractId);
40 }
41
42 const recipientAddress = process.env.MY_ACCOUNT_ID2;
43 console.log("recipientAddress:", recipientAddress);
44 deployContract("ID_bytecode", recipientAddress).catch(console.error); //ID
45 |
```

Script *deposit.js*:

```
JS deposit.js X
C: > Users > Elge > Desktop > SimpleTransfer > JS deposit.js > ...
1  const {
2    Client,
3    ContractExecuteTransaction,
4    PrivateKey,
5    Hbar,
6  } = require("@hashgraph/sdk");
7
8  require("dotenv").config();
9
10 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
11 const operatorId = process.env.MY_ACCOUNT_ID;
12 let client = Client.forTestnet();
13 client.setOperator(operatorId, operatorPrivateKey);
14
15 async function deposit(contractId, amount) {
16   const depositTx = new ContractExecuteTransaction()
17     .setContractId(contractId)
18     .setGas(1000000)
19     .setFunction("deposit")
20     .setPayableAmount(new Hbar(amount));
21
22   const response = await depositTx.execute(client);
23   const receipt = await response.getReceipt(client);
24
25   console.log("Deposit status: " + receipt.status.toString());
26 }
27
28 // Call the function
29 deposit("ID", 10).catch(console.error);           //ID
```

## Script `withdraw.js`:

```
js withdraw.js ● 
C: > Users > Elge > Desktop > SimpleTransfer > js withdraw.js > ...
1  const [
2    Client,
3    ContractExecuteTransaction,
4    ContractFunctionParameters,
5    PrivateKey,
6    Hbar,
7  ] = require("@hashgraph/sdk");
8
9  require("dotenv").config();
10
11 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY2);
12 const operatorId = process.env.MY_ACCOUNT_ID2;
13 const recipientId = process.env.MY_ACCOUNT_ID2;
14
15 let client = Client.forTestnet();
16 client.setOperator(operatorId, operatorPrivateKey);
17
18 // Conversion function to turn a Hedera account ID into a bytes32 value
19 function toBytes32(accountId) {
20   const buffer = Buffer.from(accountId, 'utf8');
21   if (buffer.length > 32) {
22     throw new Error('Account ID is too long to fit in bytes32');
23   }
24   const bytes32 = Buffer.alloc(32);
25   buffer.copy(bytes32);
26   return bytes32;
27 }
28
29 async function withdraw(contractId, amount, withdrawerId) {
30   const withdrawTx = new ContractExecuteTransaction()
31     .setContractId(contractId)
32     .setGas(1000000)
33     .setFunction("withdraw", new ContractFunctionParameters().addUint256(amount).addBytes32(toBytes32(withdrawerId)));
34 }
```

```
35   const response = await withdrawTx.execute(client);
36   const receipt = await response.getReceipt(client);
37
38   console.log("Withdraw status: " + receipt.status.toString());
39 }
40
41 const hbarAmount = 5;
42 const tinybarAmount = hbarAmount * 100000000;
43
44
45 // Call the function, replace '0.0.15053101' and '5' with your own values
46 withdraw("0.0.15053101", tinybarAmount, recipientId).catch(console.error); //ID
47
48
```

## Script *balance*:

```
JS balance.js ×
C: > Users > Elge > Desktop > SimpleTransfer > JS balance.js > ...
1 const {
2   Client,
3   AccountBalanceQuery,
4   PrivateKey,
5   Hbar,
6 } = require("@hashgraph/sdk");
7
8 require("dotenv").config();
9
10 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
11 const operatorId = process.env.MY_ACCOUNT_ID;
12 let client = Client.forTestnet();
13 client.setOperator(operatorId, operatorPrivateKey);
14
15 client.setDefaultMaxTransactionFee(new Hbar(100));
16 client.setMaxQueryPayment(new Hbar(50));
17
18 async function checkAccountBalance(accountId) {
19   // Create a query to get the account balance
20   const balanceQuery = new AccountBalanceQuery()
21     .setAccountId(accountId);
22
23   // Execute the query
24   const balance = await balanceQuery.execute(client);
25
26   console.log(`The balance of account ${accountId} is: ${balance.hbars.toString()}`);
27 }
28
29 // Replace with your account IDs
30 checkAccountBalance("ID_1").catch(console.error);
31 checkAccountBalance("ID_2").catch(console.error);
32 checkAccountBalance("ID_3").catch(console.error);
```

In cui ID\_1 è l'owner, ID\_2 il recipient e ID\_3 l'account.

Stato della cartella Simple Transfer:

Nome	Ultima modifica	Tipo	Dimensione
node_modules	06/07/2023 17:55	Cartella di file	
.env	06/07/2023 11:56	File ENV	1 KB
balance	07/07/2023 19:07	File di origine Java...	1 KB
deploy	07/07/2023 19:07	File di origine Java...	2 KB
deposit	07/07/2023 19:17	File di origine Java...	1 KB
package	06/07/2023 17:55	File JSON	1 KB
package-lock	06/07/2023 17:55	File JSON	74 KB
SimpleTransfer	07/07/2023 15:54	File JSON	114 KB
SimpleTransfer.sol	17/06/2023 16:53	File SOL	1 KB
store	05/07/2023 15:55	File di origine Java...	2 KB
withdraw	07/07/2023 19:35	File di origine Java...	2 KB

Stato aggiornato del file *.env* dopo aver creato altri account necessari per usufruire di tutte le funzionalità dello smart contract:

```
#owner
MY_ACCOUNT_ID=0.0.14818703
MY_PRIVATE_KEY=0xb4a63e4a3b8aa86c83d6d5c5caeeb8362b996b52d5ce16fdf0f6cc8984f063f8

#recipient
MY_ACCOUNT_ID2=0.0.106110753
MY_PRIVATE_KEY2=0xb6d09b74f2c632add758e2ea0d279fd8100aab2ad7b3fb0afeb9cef69bff75a1

#account 3
MY_ACCOUNT_ID3=0.0.10611009
MY_PRIVATE_KEY3=0x0171482b2a3a13c0249bdcfdc0e9b89e6a3098c6496727d55c1bc7641cc369cc

#recipient 4
MY_ACCOUNT_ID4=0.0.4627289
MY_ACCOUNT_ADDRESS4=0x09ec33f3d093483e0e194436e3b2eaddbf7cd00
MY_PRIVATE_KEY4=0x74e63756f061f300d4607a0ba62a023a941442dbffca505e360456e1
```

A questo punto possiamo procedere per fare la deploy vera e propria. Per prima cosa apriamo il prompt dei comandi e spostandoci nella cartella Simple Transfer, eseguiamo tramite node il file store.js, il quale ci restituirà il bytecode del contratto nel formato ID di Hedera

#### 4.6) Deploy del contratto e valutazione del costo

Una volta ottenuto, ci basta copiarlo e incollarlo nel file *deploy.js* nel relativo campo in cui viene richiesto, per poi eseguirlo sempre da terminale.

```
Prompt dei comandi - node store.js
Microsoft Windows [Versione 10.0.19045.2965]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Elge> cd C:\Users\Elge\Desktop\SimpleTransfer

C:\Users\Elge\Desktop\SimpleTransfer> node store.js
The smart contract bytecode file ID is 0.0.15067099
```

```
Prompt dei comandi - node deploy.js
Microsoft Windows [Versione 10.0.19045.2965]
(c) Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Elge> cd C:\Users\Elge\Desktop\SimpleTransfer

C:\Users\Elge\Desktop\SimpleTransfer> node store.js
The smart contract bytecode file ID is 0.0.15067099
^C
C:\Users\Elge\Desktop\SimpleTransfer> node deploy.js
recipientAddress: 0.0.10610753
The smart contract ID is 0.0.15067100
```

Come possiamo notare, lo script ci restituisce l'ID del recipient del contratto e l'ID dello smart contract caricato in rete, pronto per essere utilizzato dagli utenti. Possiamo copiare il valore *0.0.15067100* per fare una ricerca su Hashscan verificandone così l'avvenuto deploy, tool tramite il quale, tra le altre cose, ci permette di prendere nota dei costi necessari per caricare il contratto nel registro distribuito.

The screenshot shows the HashScan interface in TESTNET mode. Under the Contracts tab, a specific contract is selected with ID **0.0.15067100**. The details page displays various parameters: Balance (0.00000000 HBAR), Admin Key (None), Memo (None), Create Transaction ([0.0.14818703@1688818052.771279330](#)), Expiry at (2:07 PM, Jul 6, 2023, GMT+2), Auto Renew Period (90 days), Auto Renew Account (None), and Max Auto Association (0). The page also includes a link to show associated account.

Il campo *Create Transaction* contiene la stringa che identifica l'avvenuta transazione del deploy; concettualmente potremmo paragonarla all'*hash* che marca ogni singola transazione, blocco o smart contract all'interno di una blockchain qualsiasi, inclusa quella di Ethereum e, poiché il deploy è di fatto una transazione vera e propria, basterà cliccare sopra la stringa [0.0.14818703@1688818052.771279330](#) per ottenere le informazioni dei costi che stavamo cercando.

The screenshot shows the HashScan interface in TESTNET mode. Under the Transactions tab, a specific transaction is selected with ID **0.0.14818703@1688818052.771279330**. The details page shows the transaction type as CONTRACT CREATE, consensus at 2:07 PM on Jul 8, 2023, and the payer account as **0.0.14818703**. Other fields include Transaction Hash, Block (7058460), Node Submitted To (0.0.4 Hosted by Hedera | East Coast, USA), and Memo (None). The page also includes a success status indicator and a summary of fees and fees paid.

Si noti che il **Payer Account** altro non è che l'owner del contratto e cioè colui che lo ha creato: infatti, l'ID corrisponde al **primo account** del file *.env* sul quale ci sono salvate le variabili globali con le quali interagiamo coi contratti.

La fees per fare il deploy dello Smart Contract ha richiesto un quantitativo di circa 16 HBAR che alla data di caricamento del contratto corrispondono circa 0,75 dollari, contro un massimo di 100 HBAR stabiliti dalla funzione dell'SDK di Hedera *SetDefaultMaxTransactionFee* utilizzata sia nello script di *store* del bytecode sia in quello di *deploy*.

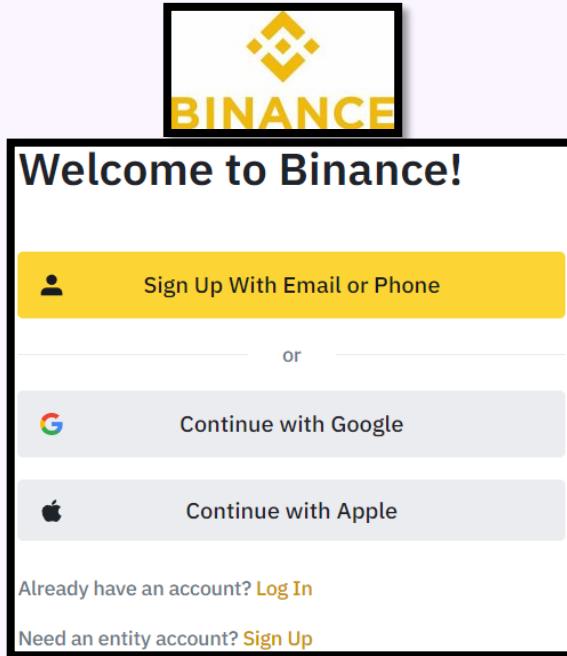
*La metodologia illustrata finora sarà la stessa che andremo ad applicare per eseguire la deploy degli altri contratti.*

Contract ID	0.0.15067100
Payer Account	0.0.14818703
Charged Fee	15.99532902 HBAR \$0.75817
Max Fee	100.00000000 HBAR \$4.73993
Valid Duration	2min
Transaction Nonce	0
Scheduled	False

#### 4.7) Extra - Test del Simple Transfer nella Mainnet

Spinti dalla curiosità di testare il contratto sulla Mainnet per stabilire se i risultati ottenuti convergessero, abbiamo deciso di aprire un account su **Binance** per comprare un po'di HBAR utili al nostro scopo. Binance è una **piattaforma di scambio** di criptovalute fondata nel 2017 da **Changpeng Zhao** che ha ottenuto un ottimo successo commerciale e mediatico. Non solo, è possibile utilizzare la piattaforma come **wallet non custodial** conservando quindi le criptovalute da noi comprate o scambiate, ma non ne saremo i veri proprietari in quanto non possederemo le chiavi private: “*not your keys, not your crypto!*”

Per poter comprare HBAR con una carta di credito o ottenerle scambiandole per **USDT**, una **stablecoin** il cui valore è ancorato a quello del dollaro, occorre registrarsi al sito [www.binance.com](http://www.binance.com) (noi faremo il login in quanto abbiamo degli USDT già presenti nel conto)



Selezionare poi la coppia di scambio **HBAR/USDT** nella sezione **Markets** ed inserire nell’**order book** il nostro ordine **SPOT** (cioè scambiamo criptovalute vere e proprie e non strumenti finanziari derivati) che sarà di tipo **BUY** in quanto vogliamo **comprare** HBAR (*numeratore*) **vendendo** USDT (*denominatore*).

Impostiamo una modica cifra di 25 USDT e mandiamo l’ordine. Essendo un mercato liquido, non avremo difficoltà a “**fillare**” l’ordine che nella pratica significa trovare un venditore disposto a darci un controvalore di HBAR pari a 25 dollari, al prezzo di mercato corrente.

Nell’ultima immagine notiamo che abbiamo così ottenuto circa **492** HBAR.



Adesso dobbiamo prelevare da Binance gli HBAR, e li trasferiremo su **Blade Wallet**, un portafoglio plug-in **custodial** consigliato dalla Hedera Foundation il quale è fruibile attraverso un qualsiasi browser, dopo averlo installato e preso nota della **chiave privata** che ci viene fornita sotto forma di una **seed phrase**, un insieme di **12 parole** che ci permettono di risalire alla chiave privata, utile oltretutto anche per ripristinare il wallet , col relativo saldo, qualora ci scordassimo la password dell'account creato o se dovessimo rompere il device, in maniera del tutto analoga per quanto concerne il **wallet/bridge** Metamask visto a lezione.

The screenshot shows the Binance Spot trading interface. At the top, there are tabs for 'Spot', 'Cross 3x', 'Isolated 5x', and 'Grid'. Below the tabs, there are two main sections: one for buying and one for selling HBAR. Each section has a 'Price' input field set to 'Market' and 'USDT'. In the 'Buy HBAR' section, the 'Total' input field is set to '25 USDT'. In the 'Sell HBAR' section, the 'Amount' input field is empty. Both sections have a slider for quantity and large green or red buttons labeled 'Buy HBAR' and 'Sell HBAR' respectively.

The screenshot shows the Binance Order History page. It displays a single recent transaction: a Market Buy order for 24.9936 USDT at a price of 0.0508 on July 3, 2023, for the HBAR/USDT pair. The transaction details include Order Time, Pair, Type, Side, Average Price, Executed Amount, Total Amount, and Trigger Conditions.

The screenshot shows the BladeWallet Extension interface. At the top, it displays the Hedera Mainnet address 0.0.2221774-xrbjf. The main screen shows the 'Main Balance' in USD (\$23.15) and HBAR (472.12). Below the balance are four buttons: Stake, Swap, Buy/Sell, and Bridge. A section for 'Manage tokens' shows a card for 'Hedera HBAR' with a balance of \$23.17 and 472.12 HBAR. A note at the bottom states: 'Before you can get any tokens, you'll need to add / associate that token with your account. Click the "Add token" button to get started.' A large orange button labeled '+ Add Token' is prominently displayed.

Si noti che nell'immagine il saldo non corrisponde al quantitativo di HBAR acquistato su **Binance**, perché i fondi sono stati trasferiti attraverso **due transazioni**; una di prova, per vedere se effettivamente i fondi non venissero persi a causa di un inserimento dell'ID scorretto o mal copiato mentre la seconda, è stata la transazione vera e propria: **entrambe** hanno richiesto un costo di rete pari a **0,8 HBAR**.

L'immagine sotto mostra le transazioni in questione comprese quelle relative al deploy del Simple Transfer, i cui costi sono anch'essi già stati scalati e verranno commentati a breve.

Recent Transactions			
ID	Type	Content	Time
0.0.2221774@1688995707.138082493	CONTRACT CREATE	Contract ID: 0.0.3140935	3:28:39.1998 PM Jul 10, 2023, GMT+2
0.0.2221774@1688995058.705620218	FILE CREATE	File ID: 0.0.3140899	3:17:50.8618 PM Jul 10, 2023, GMT+2
0.0.1030878@1688990789.718345394	CRYPTO TRANSFER	0.0.1030878 → 488.80420000 → 0.0.2221774	2:06:41.7254 PM Jul 10, 2023, GMT+2
0.0.1030878@1688990534.486101830	CRYPTO TRANSFER	0.0.1030878 → 1.20000000 → 0.0.2221774	2:02:26.5057 PM Jul 10, 2023, GMT+2

Il processo attraverso il quale è stato possibile fare il deploy dello smart contract è quasi del tutto analogo al Simple Transfert deployato sulla testnet, con la sostanziale accortezza di modificare gli script *store.js* e *deploy.js* creando un client per la Mainnet

#### Righe di codice modificate nel file *store.js*

```
13 const operatorId = process.env.MY_ACCOUNT_ID;
14 const networkName = NetworkName.Mainnet; // Imposta la rete su Mainnet
15
16 let client = Client.forMainnet(); // Crea un client per la mainnet
17 client.setNetwork(networkName);
18 client.setOperator(operatorId, operatorPrivateKey);
```

#### Righe di codice modificate nel file *deploy.js*

```
5 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
6 const operatorId = process.env.MY_ACCOUNT_ID;
7
8 let client = Client.forMainnet();
9 client.setOperator(operatorId, operatorPrivateKey);
```

Naturalmente anche il file *.env* doveva essere modificato, inserendoci come *MY\_ACCOUNT\_ID*, per l'*owner*, quello di Blade Wallet, mentre nel campo *MY\_PRIVATE\_KEY* la corrispondente chiave privata in formato ECDSA, sostanziale differenza con la rete di testnet che prevede l'uso della crittografia ED25519.

#### File *.env* per la mainnet:

```
env x
1 #owner
2 MY_ACCOUNT_ID=0.0.2221774
3 MY_PRIVATE_KEY=3030 [REDACTED] censored 3aeb9
4
```

## Esecuzione della store e della deploy

```
C:\Users\Elge\Desktop\SimpleTransfer - Copia> node store.js
L'ID del file bytecode dello smart contract è: 0.0.3140899
^C
C:\Users\Elge\Desktop\SimpleTransfer - Copia> node deploy.js
recipientAddress: 0.0.10610753
The smart contract ID is 0.0.3140935
```

Utilizzando Hashscan impostato per la mainnet, possiamo andare a cercare il nostro Simple Transfert nel registro distribuito di Hedera

The screenshot shows the HashScan interface for the MAINNET network. The top navigation bar includes links for Dashboard, Transactions, Tokens, Topics, Contracts (which is highlighted in blue), Accounts, Nodes, Staking, and Blocks. A search bar at the top right contains the placeholder "Search accounts, transactions, tokens, contracts and topics". Below the header, a large card displays the details of a deployed contract. The card title is "Contract" and the Contract ID is "0.0.3140935-hpvad". It also shows the EVM Address: "0x002fed47". The card lists various parameters: Balance (0.00000000 HBAR), Admin Key (None), Memo (None), Create Transaction (0.0.2221774@1688995707.138082493), Expires at (3:28:39.1998 PM Oct 8, 2023, GMT+2), Auto Renew Period (90 days), Auto Renew Account (None), Max. Auto. Association (0). On the right side of the card, there are fields for Obtainer (None), Proxy Account (None), Valid from (3:28:39.1998 PM Jul 10, 2023, GMT+2), Valid until (None), and File (0.0.3140899). A link "Show associated account" is located in the top right corner of the card.

E infine aprire la pagina relativa ai costi del deploy:

The screenshot shows the HashScan interface for the MAINNET network, displaying the details of a specific transaction. The transaction ID is "0.0.2221774@1688995707.138082493" and it is marked as "SUCCESS". The transaction type is "CONTRACT CREATE". The transaction was submitted at "3:28:39.1998 PM Jul 10, 2023, GMT+2". The transaction hash is a long string of hex digits: "ac2e 860b da93 9ca5 f1ac 98d9 25ba ad5c b948 2a73 ff5b 0de6 b32d fa23 886d 0e99 0376 c8c4 2516 c0fb 199f ffff 4f47 0163". The block number is "50491117". The node submitted the transaction is "0.0.12 Hosted by DLA Piper | London, UK". The memo field is "None". On the right side of the card, detailed information about the transaction is provided: Contract ID (0.0.3140935), Payer Account (0.0.2221774), Charged Fee (16.17112836 HBAR or \$0.75850), Max Fee (100.00000000 HBAR or \$4.69047), Valid Duration (2min), Transaction Nonce (0), and Scheduled (False).

È interessante notare che il **costo** di Simple Transfer *sulla mainnet* è **perfettamente in linea** con quello deployato per la testnet. Inoltre, lo script *store.js* per caricare il bytecode viene riconosciuto come una transazione semplice vera e propria, e quindi in termini di costi equivale mandare degli HBAR da un indirizzo **A** ad un indirizzo **B**, con relativa fee di **0.8** HBAR.

## SEZIONE 4

### APPROFONDIMENTI SIMPLE TRANSFER E LIMITI EMERSI

Come abbiamo visto nella sezione precedente, affinchè un contratto possa essere utilizzabile dagli utenti della rete Hedera, non basta solo caricarlo nel Network: bisogna fare in modo che le **funzioni** per cui si è progettato il software **funzionino**.

Nel caso del Simple Transfer, abbiamo avuto problemi con i metodi **deposit** e **withdraw** poiché non funzionavano così come sono stati implementati nel codice relativo che abbiamo riportato in sezione 2, e di conseguenza allo script Javascript relativo che abbiamo costruito, poiché la console riportava l'errore **ReceiptStatusError** e riportando **CONTRACT\_REVERT\_EXECUTED**, indicando quest'ultimo la reversione del contratto che in genere si verifica quando una condizione del contratto non viene soddisfatta oppure si verifica un errore durante l'esecuzione e il contratto decide di annullare l'intera operazione; i messaggi aggiuntivi che ci sono stati suggeriti, nel caso della withdraw, ovvero **TransactionReceiptQuery** ci hanno fatto ipotizzare che probabilmente c'era qualche errore di programmazione nel file Solidity, è così è stato: dopo svariati tentativi, ci siamo resi conto che nonostante gli **address** sono stati dichiarati come **payable** e quindi in grado di ricevere quantitativo di criptovaluta, per Hedera abbiamo dovuto modificare la struttura dati degli address, poiché gli ID non possono essere di tipo payable e di conseguenza andava cambiata la parola chiave di Solidity **address** con **bytes32**, unica soluzione implementabile trovata per poter rendere funzionante la withdraw.

Il codice **.sol** che viene riportato di seguito, altro non è che il file modificato di quello proposto con il quale abbiamo lavorato in tutta la sezione 3.

#### SimpleTransfer

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity ^0.8.1;
3
4 contract SimpleTransfer {
5     bytes32 public recipient;
6     address payable owner;
7
8     constructor(bytes32 _recipient) {    infinite gas 329200 gas
9         recipient = _recipient;
10        owner = payable(msg.sender);
11    }
12
13    function deposit() public payable {    2645 gas
14        require(msg.sender == owner, "only the owner can deposit");
15    }
16
17    function withdraw(uint256 amount, bytes32 withdrawerId) public {    infinite gas
18        require(recipient == withdrawerId, "only the recipient can withdraw");
19        require(amount <= address(this).balance, "the contract balance is less than required amount");
20
21        (bool success, ) = owner.call{value: amount}("");
22        require(success, "Transfer failed.");
23    }
}
```

In cui sono state aggiunte altre due funzioni per tenere meglio traccia, dal prompt dei comandi, dell'effettivo funzionamento dei metodi *deposit* e *withdraw* attraverso lo script *getRecipientAndContractBalance*

```
25     function getRecipient() public view returns (bytes32) {    2415 gas
26         return recipient;
27     }
28
29     function getContractBalance() public view returns (uint256) {    361 gas
30         return address(this).balance;
31     }
32 }
```

## Script del file *getRecipientAndContractBalance.js*:

```

JS getRecipientAndContractBalance.js ×
C: > Users > Elge > Desktop > SimpleTransfer > JS getRecipientAndContractBalance.js > ...
1 const {
2   Client,
3   ContractCallQuery,
4   ContractId,
5   PrivateKey,
6   Hbar,
7 } = require("@hashgraph/sdk");
8
9 require("dotenv").config();
10
11 const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
12 const operatorId = process.env.MY_ACCOUNT_ID;
13 let client = Client.forTestnet();
14 client.setOperator(operatorId, operatorPrivateKey);
15 client.setMaxQueryPayment(new Hbar(50));
16
17 async function main() {
18   const contractId = ContractId.fromString("0.0.15067100"); // Replace with your contract ID
19
20   // Get recipient
21   const getRecipientQuery = new ContractCallQuery()
22     .setContractId(contractId)
23     .setFunction("getRecipient")
24     .setGas(2000000);
25   const recipient = await getRecipientQuery.execute(client);
26   //console.log("The recipient of contract " + contractId + " is: " + recipient.getUint256(0).toString());
27   //console.log("The recipient of contract " + contractId + " is recipient.getString(0): " + recipient.getString(0));
28   console.log("The recipient of contract " + contractId + " is recipient.getUint256(0).toString(): " + recipient.getUint256(0).toString());
29   console.log("The recipient of contract " + contractId + " is recipient.toString(): " + recipient.toString());
30   console.log("The recipient of contract " + contractId + " is recipient.getAddress(): " + recipient.getAddress());
31   console.log("The recipient of contract " + contractId + " is process.env.MY_ACCOUNT_ID2: " + process.env.MY_ACCOUNT_ID2);
32   // console.log("The recipient of contract " + contractId + " is recipientAddress: " + recipientAddress); //ReferenceError: recipientAddress is not defined at main
33
34   // Get contract balance
35   const getContractBalanceQuery = new ContractCallQuery()
36     .setContractId(contractId)
37     .setFunction("getContractBalance")
38     .setGas(2000000);
39   const balance = await getContractBalanceQuery.execute(client);
40   console.log("The balance of contract " + contractId + " is: " + balance.getUint256(0).toString() + " tinybars (" + (balance.getUint256(0).toNumber() / 100000000) + " Hbar")
41
42 }
43
44 main().catch(console.error);
45

```

## Stato della cartella Simple Transfer:

Nome	Ultima modifica	Tipo	Dimensione
node_modules	08/07/2023 13:56	Cartella di file	
.env	08/07/2023 13:31	File ENV	1 KB
balance	08/07/2023 17:05	File di origine Java...	1 KB
deploy	08/07/2023 14:07	File di origine Java...	2 KB
deposit	08/07/2023 16:56	File di origine Java...	1 KB
getRecipientAndContractBalance	08/07/2023 18:38	File di origine Java...	3 KB
package	02/07/2023 23:02	File JSON	1 KB
package-lock	05/07/2023 12:46	File JSON	47 KB
SimpleTransfer	05/07/2023 15:52	File JSON	136 KB
SimpleTransfer.sol	05/07/2023 15:51	File SOL	1 KB
store	05/07/2023 15:55	File di origine Java...	2 KB
withdraw	08/07/2023 19:02	File di origine Java...	2 KB

Possiamo provare le funzioni del contratto caricato nel Network nella sezione 3, una volta aver sostituito negli script di *deposit* e *withdraw* l'ID del contratto precedentemente creato:

Ovviamente l'adattamento di tipo `address` a `bytes32` riguarderà tutti i contratti che richiederanno un indirizzo pagabile.

## SEZIONE 5

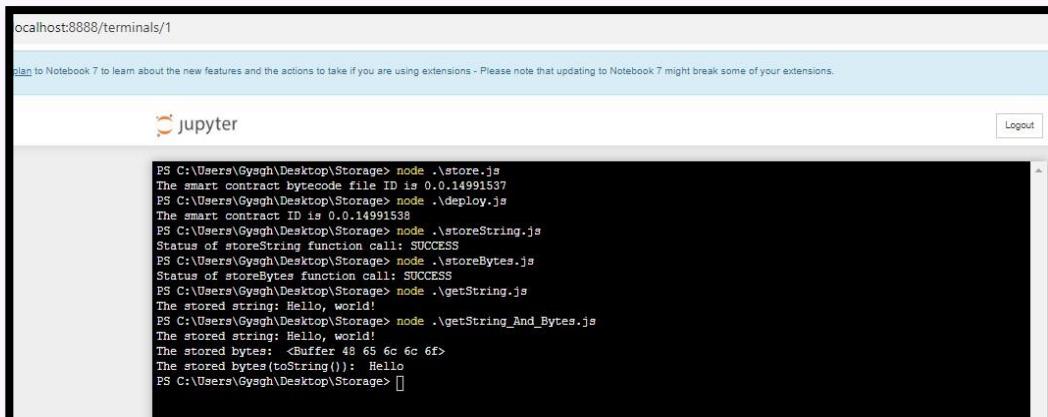
### DEPLOY DEGLI ALTRI SMART CONTRACT

#### DataStorage

Vediamo un altro esempio con il seguente contratto **dataStorage.sol**

```
5  The contract Storage allows a user to
6  store inside the blockchain two
7  typologies of dynamic size data:
8  a byte sequence and a string.
9
10 After contract creation, the contract
11 allows two actions:
12 - **storeBytes**, which allows the user
13 to store an arbitrary
14 sequence of bytes;
15 - **storeString**, which allows the user
16 to store a string of arbitrary
17 length.
18
19 ## Execution traces
20 // SPDX-License-Identifier: GPL-3.0
21
22 pragma solidity ^0.8.0;
23
24 contract DataStorage {
25
26     bytes public byteSequence;
27     string public textString;
28
29     function storeBytes(bytes memory _byteSequence) public {
30         byteSequence = _byteSequence;
31     }
32
33     function storeString(string memory _textString) public {
34         textString = _textString;
35     }
36
37 }
```

Anche in questo caso vediamo come sono stati eseguiti tutti i seguenti passaggi:



The screenshot shows a Jupyter Notebook interface with a terminal window. The terminal output is as follows:

```
localhost:8888/terminals/1
plan to Notebook 7 to learn about the new features and the actions to take if you are using extensions - Please note that updating to Notebook 7 might break some of your extensions.

jupyter
PS C:\Users\Gyagh\Desktop\Storage> node .\store.js
The smart contract bytecode file ID is 0.0.14991537
PS C:\Users\Gyagh\Desktop\Storage> node .\deploy.js
The smart contract ID is 0.0.14991538
PS C:\Users\Gyagh\Desktop\Storage> node .\storeString.js
Status of storeString function call: SUCCESS
PS C:\Users\Gyagh\Desktop\Storage> node .\storeBytes.js
Status of storeBytes function call: SUCCESS
PS C:\Users\Gyagh\Desktop\Storage> node .\getString.js
The stored string: Hello, world!
PS C:\Users\Gyagh\Desktop\Storage> node .\getString_And_Bytes.js
The stored string: Hello, world!
The stored bytes: <buffer 48 65 6e 6c 6f>
The stored bytes(toString()): Hello
PS C:\Users\Gyagh\Desktop\Storage> []
```

Abbiamo la **store.js** identica alla precedente, mentre la **deploy**:

```
const {
  Client,
  ContractCreateTransaction,
  PrivateKey,
  Hbar,
} = require("@hashgraph/sdk");

require("dotenv").config();

const operatorPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);
const operatorId = process.env.MY_ACCOUNT_ID;
let client = Client.forTestnet();
client.setOperator(operatorId, operatorPrivateKey);

// Set the default maximum transaction fee (in Hbar)
client.setDefaultMaxTransactionFee(new Hbar(100));

// Set the maximum payment for queries (in Hbar)
client.setMaxQueryPayment(new Hbar(50));

async function deployContract(bytecode fileId) {
  // Instantiate the contract instance
  const contractTx = new ContractCreateTransaction()
    .setBytecodeFileId(bytecode fileId)
    .setGas(100000);

  // Submit the transaction to the Hedera test network
  const contractResponse = await contractTx.execute(client);

  // Get the receipt of the transaction
  const contractReceipt = await contractResponse.getReceipt(client);

  // Get the smart contract ID
  const newContractId = contractReceipt.contractId;

  // Log the smart contract ID
  console.log("The smart contract ID is " + newContractId);

  // Get the transaction record to find the actual cost
  const transactionRecord = await contractResponse.getRecord(client);

  // Log the actual transaction fee
  //console.log("The transaction fee was " + transactionRecord.fee);
}

// Call the async function
deployContract("0.0.14991537").catch(console.error);
```

### storeString.js :

```
const {
  Client,
  PrivateKey,
  ContractCallQuery,
  Hbar,
  ContractExecuteTransaction,
  ContractFunctionParameters,
  ContractId,
} = require("@hashgraph/sdk");

require("dotenv").config();

const myAccountId = process.env.MY_ACCOUNT_ID;
const myPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);

const client = Client.forTestnet();
client.setOperator(myAccountId, myPrivateKey);

client.setMaxQueryPayment(new Hbar(50));

async function storeString() {
  const contractId = ContractId.fromString("0.0.14991538"); // ID

  const contractExecTx = new ContractExecuteTransaction()
    .setContractId(contractId)
    .setGas(1000000)
    .setFunction("storeString", new ContractFunctionParameters().addString("Hello, world!"));

  const submitExecTx = await contractExecTx.execute(client);
  const receipt = await submitExecTx.getReceipt(client);

  console.log("Status of storeString function call: " + receipt.status.toString());
}

storeString().catch(console.error);
```

### storeBytes.js :

```
const {
  Client,
  PrivateKey,
  ContractExecuteTransaction,
  ContractFunctionParameters,
  ContractId,
  Hbar,
} = require("@hashgraph/sdk");

require("dotenv").config();

const myAccountId = process.env.MY_ACCOUNT_ID;
const myPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);

const client = Client.forTestnet();
client.setOperator(myAccountId, myPrivateKey);

client.setMaxQueryPayment(new Hbar(50));

async function storeBytes() {
  const contractId = ContractId.fromString("0.0.14991538"); // ID

  //const data = Buffer.from("Hello, world!");
  const data = Buffer.from([0x48, 0x65, 0x6c, 0x6c, 0x6f]); // Bytes for "Hello" in ASCII

  const contractExecTx = new ContractExecuteTransaction()
    .setContractId(contractId)
    .setGas(1000000)
    .setFunction("storeBytes", new ContractFunctionParameters().addBytes(data));

  const submitExecTx = await contractExecTx.execute(client);
  const receipt = await submitExecTx.getReceipt(client);

  console.log("Status of storeBytes function call: " + receipt.status.toString());
}

storeBytes().catch(console.error);
```

**getString.js :**

```
const {
  Client,
  PrivateKey,
  ContractCallQuery,
  Hbar,
  ContractId,
} = require("@hashgraph/sdk");

require("dotenv").config();

const myAccountId = process.env.MY_ACCOUNT_ID;
const myPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);

const client = Client.forTestnet();
client.setOperator(myAccountId, myPrivateKey);

client.setMaxQueryPayment(new Hbar(50));

async function getString() {
  const contractId = ContractId.fromString("0.0.14990560"); // ID

  const contractQuery = new ContractCallQuery()
    .setGas(100000)
    .setContractId(contractId)
    .setFunction("textString")
    .setQueryPayment(new Hbar(2));

  const response = await contractQuery.execute(client);

  const storedString = response.getString(0);

  console.log("The stored string: " + storedString);
}

getString().catch(console.error);
```

## getString\_And\_Bytes.js

```
const {Client,PrivateKey,ContractCallQuery,Hbar,ContractId,} = require("@hashgraph/sdk");
require("dotenv").config();
const myAccountId = process.env.MY_ACCOUNT_ID;
const myPrivateKey = PrivateKey.fromString(process.env.MY_PRIVATE_KEY);

const client = Client.forTestnet();
client.setOperator(myAccountId, myPrivateKey);
client.setMaxQueryPayment(new Hbar(50));

async function getString() {
  const contractId = ContractId.fromString("0.0.14990560"); // ID

  const contractQuery = new ContractCallQuery()
    .setGas(100000)
    .setContractId(contractId)
    .setFunction("textString")
    .setQueryPayment(new Hbar(2));

  const response = await contractQuery.execute(client);

  const storedString = response.getString(0);

  console.log("The stored string: " + storedString);
}

async function getBytes() {
  const contractId = ContractId.fromString("0.0.14990560"); // ID

  const contractQuery = new ContractCallQuery()
    .setGas(100000)
    .setContractId(contractId)
    .setFunction("byteSequence")
    .setQueryPayment(new Hbar(2));

  const response = await contractQuery.execute(client);

  const storedBytes = response.getBytes(0);

  console.log("The stored bytes: ", storedBytes);
  console.log("The stored bytes(toString()): ", storedBytes.toString());
}

getString().catch(console.error);
getBytes().catch(console.error);
```

## Crowdfund

Abbiamo visto anche il contratto crowdfund, si compone di due parti principali: uno smart contract scritto in Solidity e uno script JavaScript.

Lo **smart contract Crowdfund** ha diverse variabili di stato:

- **end\_donate**: la data di fine per le donazioni.
- **goal**: la quantità di denaro necessaria per raggiungere l'obiettivo.
- **receiver**: l'identificativo del destinatario dei fondi.
- **donors**: una mappa che associa ogni donatore alla somma donata.
- **owner**: il proprietario del contratto, ovvero chi lo ha creato.

Il costruttore del contratto viene chiamato alla creazione e inizializza queste variabili.

Esistono tre funzioni pubbliche:

- **donate(bytes32 donorId)**: permette a un donatore di inviare fondi al contratto.
- **withdraw(uint256 amount, bytes32 withdrawerId)**: consente al beneficiario di prelevare i fondi se l'obiettivo è stato raggiunto.
- **reclaim(bytes32 claimerId)**: permette a un donatore di riprendere i suoi fondi se l'obiettivo non è stato raggiunto.

Le funzioni di ritiro e reclamo sono protette da vari **require** che assicurano che le condizioni siano soddisfatte prima che la transazione avvenga.

```
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity >= 0.8.2;

contract Crowdfund {

    uint end_donate;      // Timestamp of when users can no longer donate
    uint goal;            // amount of hbar that must be donated for the crowdfunding to be successful
    bytes32 public receiver; // receiver of the donated funds
    mapping(bytes32 => uint) public donors; // mapping of donors' addresses to their donations
    address payable owner;

    constructor (bytes32 receiver_, uint end_donate_, uint256 goal_) {
        receiver = receiver_;
        end_donate = end_donate_;
        goal = goal_;
        owner = payable(msg.sender);
    }

    function donate(bytes32 donorId) public payable {
        require (block.timestamp <= end_donate);
        donors[donorId] += msg.value;
    }

    function withdraw(uint256 amount, bytes32 withdrawerId) public {
        require (block.timestamp >= end_donate, "Withdrawal period has not started");
        require (address(this).balance >= goal);
        require (withdrawerId == receiver, "Only the receiver can withdraw");

        (bool success, ) = owner.call{value: amount}("");
        require(success, "Transfer failed.");
    }

    function reclaim(bytes32 claimerId) public {
        require (block.timestamp >= end_donate, "Reclaim period has not started");
        require(address(this).balance < goal, "Funds goal has been reached or exceeded");
        require (donors[claimerId] > 0, "No donation found for the claimer");
        uint amount = donors[claimerId];
        donors[claimerId] = 0;
        (bool success, ) = owner.call{value: amount}("");
        require(success, "Transfer failed.");
    }
}
```

Riadattamento a Hedera, Il processo di riadattamento del contratto iniziale in quello finale si è concentrato su quattro aspetti chiave:

1. **Modifica delle variabili:** Nel contratto originale, **receiver** era un indirizzo Ethereum, mentre nel contratto finale è un **bytes32**. Questo cambio è dovuto alla necessità di utilizzare identificativi specifici su Hedera Hashgraph.
2. **Cambio del metodo di misurazione del tempo:** Il contratto originale usava il numero di blocchi (**block.number**) per stabilire la durata della fase di donazione. Questo funziona su Ethereum dove il tempo è misurato in blocchi. Nel contratto finale, abbiamo sostituito **block.number** con **block.timestamp** poiché Hedera Hashgraph usa il timestamp del blocco per indicare il tempo.
3. **Introduzione del proprietario del contratto:** Nel contratto finale è stata aggiunta la variabile **owner** che rappresenta il proprietario del contratto, ovvero chi lo ha creato. Questa variabile è utilizzata nelle transazioni di prelievo e recupero, dove l'importo viene trasferito all'indirizzo del proprietario del contratto.

4. **Controllo dei requisiti nelle funzioni:** Nel contratto finale, le funzioni **withdraw** e **reclaim** includono controlli aggiuntivi rispetto al contratto originale. Ad esempio, la funzione **withdraw** richiede che l'ID del prelevatore sia uguale a quello del ricevitore, e la funzione **reclaim** verifica che l'obiettivo dei fondi non sia stato raggiunto.

Lo **script JavaScript** automatizza il processo di interazione con lo smart contract. Utilizza l'SDK di Hedera Hashgraph per interagire con la rete testnet di Hedera. All'inizio dello script, vengono impostati l'ID e la chiave privata dell'operatore, come sempre. Il codice contiene cinque funzioni asincrone principali:

- **deployContract:** crea e implementa il contratto sulla rete.
- **donate:** invia una transazione di donazione al contratto.
- **withdraw:** invia una transazione di prelievo al contratto.
- **reclaim:** invia una transazione di recupero al contratto.
- **execute:** gestisce l'intero flusso di esecuzione del contratto.

La funzione **execute** crea un nuovo contratto, attende 2 secondi, poi effettua una donazione. Dopo aver atteso che sia trascorso il periodo di donazione, tenta di prelevare i fondi o, se l'obiettivo non è stato raggiunto, di restituire i fondi ai donatori.

La funzione **execute** gestisce l'intero flusso di operazioni nel nostro contratto di crowdfunding. Inizia con la **configurazione iniziale** di vari parametri chiave per il contratto, come l'ID del bytecode, l'indirizzo del destinatario, il termine per le donazioni, l'obiettivo del crowdfunding e i dettagli del donatore, del prelevatore e del reclamante.

Successivamente, la funzione **deployContract** viene chiamata per creare il contratto sulla blockchain. Una volta che il contratto è attivo, la funzione **donate** viene utilizzata per effettuare una donazione al contratto.

Terminato il periodo di donazione, la funzione **withdraw** permette al destinatario di prelevare fondi dal contratto. In un secondo round di operazioni, viene creata un'altra istanza del contratto con un obiettivo più elevato. Dopo un'altra donazione, la funzione **reclaim** permette al donatore di recuperare la sua donazione, dato che l'obiettivo non è stato raggiunto.

Infine, l'intera esecuzione è avvolta in una chiamata a **execute().catch(console.error);** per gestire eventuali errori. In sintesi, **execute** dimostra un'efficace gestione del flusso di lavoro di un contratto di crowdfunding, da creazione, donazione, prelievo fino al recupero dei fondi, in quanto simula quello che abbiamo fatto prima ma in modo sequenziale e automatico, separando l'ordine di esecuzione degli script che in questo caso sono direttamente trasformati in funzioni, è un altro interessante approccio di lavoro durante il testing dei contratti, ecco un **esempio di come appare il file completo usato nel progetto**, abbiamo fatto diverse versioni in quanto è molto personalizzabile. *Basterà quindi fare solo node store.js, mettere l'id nella deploy del file e fare node .\all.js*

## Logs:

```

// SPDX-License-Identifier: GPL-3.0-only
pragma solidity >= 0.8.0;

contract CrowdFund {
    uint end_donate;
    uint goal;
    bytes32 public receiver;
    mapping(bytes32 => uint) public donors;
    address payable owner;
    bool goalReached = false;

    constructor (bytes32 receiver_, uint end_donate_, uint256 goal_) {
        receiver = receiver_;
        end_donate = end_donate_;
        goal = goal_;
        owner = payable(msg.sender);
    }

    function donate(bytes32 donorId) public payable {
        require(block.timestamp >= end_donate);
        donors[donorId] += msg.value;

        if(address(this).balance >= goal){
            goalReached = true;
        }
    }

    function withdraw(uint256 amount, bytes32 withdrawerId) public {
        require(block.timestamp >= end_donate, "1");
        require(address(this).balance >= goal);
        require(withdrawerId == receiver, "3");

        (bool success,) = owner.call{value: amount}("");
        require(success, "Transfer failed.");
    }

    function reclaim(bytes32 claimerId) public {
        require(block.timestamp >= end_donate, "1");
        require(gaddress(claimerId));
        require(donors[claimerId] > 0, "3");
        uint amount = donors[claimerId];
        donors[claimerId] = 0;
        (bool success,) = owner.call{value: amount}("");
        require(success, "Transfer failed.");
    }

    // Added these functions Inside the contract
    function isGoalReached() public view returns (bool) {
        return goalReached;
    }

    function getDonationAmount(bytes32 donorId) public view returns
    (uint256) {
        return donors[donorId];
    }
}

```

```
PS C:\Users\Gysgh> cd C:\Users\Gysgh\Desktop\crowdfund_Hedera-old
PS C:\Users\Gysgh\Desktop\crowdfund_Hedera-old> node .\all.js
Starting execution...
Current time: 19:59:55
Deploying contract...
Current time: 19:59:55
Contract deployed for testing with ID: 0.0.15068593
Contract link: https://hashscan.io/testnet/contract/0.0.15068593
Making donation..., the goal is 100
Current time: 19:59:58
Donation limit: 110
Waiting for donation period to end...
Donate status: SUCCESS
Amount donated: 110
Donation made.
Making withdrawal...
Current time: 20:00:19
Withdraw status: SUCCESS
Amount withdrawn: 110
Withdrawal made.
Deploying contract...
Current time: 20:00:20
Contract deployed for testing with ID: 0.0.15068594
Contract link: https://hashscan.io/testnet/contract/0.0.15068594
Making donation..., the goal is 100
Current time: 20:00:24
Donation limit: 50
Waiting for donation period to end...
Donate status: SUCCESS
Amount donated: 50
Donation made.
goal 100
Making reclaim...
Current time: 20:00:45
Waiting for donation period to end...
```

## HTLC

Versione di contratto riadattato:

```
After contract creation, the HTLC allows two actions:  
- **reveal**, which requires the caller to provide a preimage of the commit,  
and transfers the whole contract balance to the committer;  
- **timeout**, which can be called only after the deadline, and  
and transfers the whole contract balance to the receiver.  
  
## Execution traces  
  
### Trace 1  
  
1. The committer creates the contract, setting a deadline of 100 rounds;  
1. After 50 rounds, A performs the **reveal** action.  
  
### Trace 2  
  
1. The committer creates the contract, setting a deadline of 100 rounds;  
1. After 100 rounds, the receiver performs the **timeout** action.  
// SPDX-License-Identifier: MIT  
pragma solidity ^0.8.18;  
  
contract HTLC {  
    address payable public owner;  
    address payable public verifier;  
    bytes32 public hash;  
    uint reveal_timeout;  
  
    constructor(address payable v, bytes32 h, uint delay, uint collateralHBAR) payable {  
        require (msg.value >= collateralHBAR);  
        owner = payable(msg.sender);  
        verifier = v;  
        hash = h;  
        reveal_timeout = block.number + delay;  
    }  
  
    function reveal(string memory s) public {  
        require (msg.sender == owner);  
        require(keccak256(abi.encodePacked(s)) == hash);  
        (bool success,) = owner.call{value: address(this).balance}("");  
        require (success, "Transfer failed.");  
    }  
  
    function timeout() public {  
        require (block.number > reveal_timeout);  
        (bool success,) = verifier.call{value: address(this).balance}("");  
        require (success, "Transfer failed.");  
    }  
}
```

Come potremmo lavorare sul contratto in un unico script che svolge tutto quel che abbiamo imparato in precedenza in maniera semplice e organizzata? Ecco una struttura che può aiutare chi si cimenta in questi lavori a costruire una base di sviluppo per questo Smart Contract. **Execute:**

```
async function execute() {  
    console.log("Checking account balances before execute...");  
    await checkAccountBalance({ accountid: "0.0.4652957" }).catch(console.error);  
    await checkAccountBalance({ accountid: "0.0.10610753" }).catch(console.error);  
    await checkAccountBalance({ accountid: "0.0.10611089" }).catch(console.error);  
  
    console.log("Storing the contract bytecode...");  
    const bytecodefileId = await storeBytecode();  
    console.log("Bytecode stored successfully.");  
  
    const delay = Math.floor((Date.now() / 1000) + 100);  
    const verifier = process.env.MY_ACCOUNT_ID2;  
    const secret = "MySecret";  
    const hash = Buffer.from(keccak256(secret), { encoding: 'hex' });  
    const collateralHBAR = 1; // Importo del collaterale in HBAR  
  
    console.log("Deploying new contract...");  
    const contractId = await deployContract(bytecodefileId, verifier, hash, delay, collateralHBAR);  
    console.log("Deployed new contract with ID:", contractId.toString());  
  
    // Wait a bit before revealing  
    console.log("Waiting 4 seconds before reveal...");  
    await new Promise(r => setTimeout(r, 4000));  
  
    console.log("Revealing secret...");  
    console.log("Sender address:", client.operatorAccountId.toString());  
    console.log("Expected hash:", hash.toString({ encoding: 'hex' }));  
    console.log("Hash of revealed secret:", Buffer.from(keccak256(secret), { encoding: 'hex' }).toString({ encoding: 'hex' }));  
  
    const revealReceipt = await reveal(contractId, secret);  
    console.log("Secret revealed. Status:", revealReceipt.status.toString());  
  
    // Wait for the deadline to pass before calling timeout  
    console.log("Waiting for deadline to pass before calling timeout...");  
    await new Promise(r => setTimeout(r, 105000));  
  
    console.log("Calling timeout...");  
    const timeoutReceipt = await timeout(contractId);  
    console.log("Timeout called. Status:", timeoutReceipt.status.toString());  
}  
  
execute().catch(console.error);
```

## Struttura centrale

```
1  require("dotenv").config();
2  const {
3    Client,
4    PrivateKey,
5    FileCreateTransaction,
6    ContractCreateTransaction,
7    ContractFunctionParameters,
8    ContractExecuteTransaction,
9    Hbar,
10   AccountBalanceQuery,
11 } = require("@hashgraph/sdk");
12 const { keccak256 } = require('js-sha3');
13
14 const operatorPrivateKey = PrivateKey.fromString(
15   process.env.MY_PRIVATE_KEY
16 );
17 const operatorId = process.env.MY_ACCOUNT_ID;
18
19 let client = Client.forTestnet();
20 client.setOperator(operatorId, operatorPrivateKey);
21 client.setDefaultMaxTransactionFee(new Hbar({ amount: 100 }));
22 client.setMaxQueryPayment(new Hbar({ amount: 50 }));
23
24 let dataStorage = require("./HTLC.json");
25 const bytecode = dataStorage.data.bytecode.object;
26
27 1 usage
28 ④async function storeBytecode() {...}
29
30 5+ usages
31 ④async function checkAccountBalance(accountId) {...}
32
33 1 usage
34 ④async function deployContract(bytecode fileId, verifier, hash, delay, collateralHBAR) {...}
35
36 1 usage
37 ④async function reveal(contractId, secret) {...}
38
39 1 usage
40 ④async function timeout(contractId) {...}
```

Lo script esegue le seguenti funzioni:

1. **Caricamento dei moduli necessari:** Il modulo **dotenv** viene utilizzato per caricare variabili d'ambiente dal file **.env** nel **process.env**. Il modulo **@hashgraph/sdk** contiene metodi per interagire con la rete Hedera, mentre il modulo 'js-sha3' fornisce una funzione per calcolare il Keccak-256 hash di una stringa.
2. **Configurazione del client:** Il client viene configurato per utilizzare la rete di test di Hedera, con l'ID dell'operatore e la chiave privata dell'operatore definiti dalle variabili d'ambiente.
3. **Definizione delle funzioni asincrone per interagire con la rete Hedera:** Queste funzioni includono **storeBytecode** (per memorizzare il bytecode di un contratto sulla rete), **checkAccountBalance** (per ottenere il saldo di un account), **deployContract** (per creare un nuovo contratto sulla rete), **reveal** (per rivelare un segreto a un contratto) e **timeout** (per richiamare la funzione di timeout di un contratto).
4. **Esecuzione delle funzioni:** La funzione **execute** viene chiamata per eseguire le operazioni. Questa funzione controlla i saldi degli account, memorizza il bytecode del contratto sulla rete, distribuisce un nuovo contratto, attende un po' prima di rivelare un segreto al contratto, attende che scada il tempo limite prima di chiamare la funzione di timeout del contratto.

Nel dettaglio:

- **storeBytecode()**: questa funzione salva il bytecode di uno smart contract su Hedera e restituisce l'ID del file, che può essere utilizzato per riferirsi a esso in futuro.
- **checkAccountBalance(accountId)**: questa funzione controlla e stampa il saldo del conto specificato.
- **deployContract()**: questa funzione implementa un contratto utilizzando il bytecode salvato precedentemente. Viene specificato un set di parametri del costruttore, incluso l'indirizzo del verificatore, l'hash del segreto, il delay per la rivelazione del segreto e l'importo di collateralHBAR. Restituisce l'ID del contratto creato.
- **reveal()**: questa funzione chiama la funzione "reveal" dello smart contract, passando il segreto come argomento. Stampa la ricevuta della transazione.
- **timeout()**: questa funzione chiama la funzione "timeout" dello smart contract dopo che il deadline è passato. Stampa la ricevuta della transazione.
- **execute()**: questa è la funzione principale che organizza le operazioni. Memorizza il bytecode, verifica i saldi degli account, implementa il contratto, attende 4 secondi, rivela il segreto, aspetta che scada il tempo, e infine chiama il timeout.

Ogni passaggio del processo viene stampato sulla console per tener traccia dell'avanzamento del processo. Questo metodo di esecuzione automizza il processo semplificando il lavoro durante lo sviluppo, un developer può considerare questa opzione per risparmiare tempo e spazio a seconda delle proprie esigenze.

**SEZIONE 6**  
*RACCOLTA DATI E ANALISI DEI COSTI*

Di seguito si riportano i costi di deploy degli smart contract proposti cercando di dare un senso ai risultati ottenuti:

**Costo deploy Simple Transfer:**

Contract ID	0.0.15067100
Payer Account	0.0.14818703
Charged Fee	15.99532902 <small>h \$0.75817</small>
Max Fee	100.00000000 <small>h \$4.73993</small>
Valid Duration	2min
Transaction Nonce	0
Scheduled	False

**Costo deploy Data Storage:**

Contract ID	0.0.15077653
Payer Account	0.0.4652957
Charged Fee	8.89740889 <small>h \$0.42164</small>
Max Fee	100.00000000 <small>h \$4.73893</small>
Valid Duration	2min
Transaction Nonce	0
Scheduled	False

**Costo deploy Crowdfund:**

Contract ID	0.0.15068593
Payer Account	0.0.4652957
Charged Fee	16.17102200h \$0.76567
Max Fee	100.00000000h \$4.73480
Valid Duration	2min
Transaction Nonce	0
Scheduled	False

#### **Costo deploy HTLC:**

Contract ID	0.0.15077674
Payer Account	0.0.4652957
Charged Fee	9.99088525h \$0.47346
Max Fee	100.00000000h \$4.73893
Valid Duration	2min
Transaction Nonce	0
Scheduled	False

Come ci aspettavamo, i costi di deploy sono stati dello stesso ordine di grandezza, seppur con una varianza in certi casi significativa; in generale sappiamo che i costi di caricamento degli smart contract nella rete Hedera (come altre Chain EVM) dipendono principalmente dalla quantità di gas necessaria per eseguire la transazione di creazione del contratto, che rappresenta l'unità di misura per il costo computazionale delle operazioni eseguite dalla macchina virtuale. Anche se gli smart contract possono avere logica e complessità diverse, il costo sarà generalmente simile se richiedono lo stesso tipo di operazioni, che si traduce in quantità di risorse di calcolo messe in gioco: a tal proposito, non si può non trascurare il ruolo che hanno i costruttori i quali come abbiamo visto mettono in gioco fin da subito delle variabili di tipo *bytes32* le quali sappiamo che hanno un peso in termini di costi significativo per quanto concerne le risorse messe in gioco dalla macchina virtuale. Infatti, seppur vero il fatto che ci aspettavamo dei costi di deploy simili, non ci aspettavamo che fossero relativamente alti, non per un registro distribuito il cui marketing è stato da sempre focalizzato sui costi e sulle fees basse: persino un semplice trasferimento di HBAR da un account all'altro, ha richiesto circa 0,04\$, più di un ordine di grandezza di differenza rispetto al millesimo da loro garantito.

## Conclusioni

Concludendo, il lavoro con Hedera ha offerto un'opportunità unica per esplorare una tecnologia emergente. Durante lo sviluppo del progetto, siamo stati in grado di affrontare una serie di sfide che hanno permesso di ampliare le nostre competenze e la nostra comprensione delle reti blockchain e dei contratti smart. Nonostante alcune difficoltà iniziali, siamo riusciti a superare tali ostacoli attraverso la ricerca attiva e l'implementazione di soluzioni creative.

Il nostro progetto ha offerto uno sguardo sulle potenzialità e sulle limitazioni di Hedera in questo momento. Abbiamo scoperto che, sebbene la documentazione sia ancora in fase di sviluppo e possano esistere problemi di compatibilità tra le versioni, la rete offre comunque una serie di funzionalità potenti e promettenti.

In futuro, sarà interessante osservare come Hedera continua a evolvere. Siamo convinti che la piattaforma abbia molto da offrire e siamo ansiosi di vedere come il team di sviluppo risolverà le sfide attuali e migliorerà ulteriormente le funzionalità esistenti.

Anche perchè, come per tutte le tecnologie, un ingrediente dell'adozione di massa e quindi del successo commerciale di un prodotto è la semplicità attraverso la quale si utilizza il prodotto stesso, senza che l'utente si debba preoccupare di cosa avvenga "dietro le quinte" ma anzi che sia di facile comprensione e immediatezza la parte applicativa del software, in questo caso.

Riteniamo inoltre che non è strettamente necessario risolvere completamente il trilemma blockchain da sempre professato: è intelligente ragionare in termini di valore conservato da una rete e utilizzo per la quale nasce: se davvero un giorno bitcoin sarà uno Store of Value e quindi percepito e fruito come tale, e non come digital cash p2p speso quotidianamente come prevedeva l'idea originale presente nel relativo white paper, non ha senso concentrarsi sulla ricerca e sviluppo di soluzioni di scalabilità (Lightning Network potrebbe già essere sufficiente); analogamente, non ha senso sacrificare scalabilità su Hedera per avere più decentramento o sicurezza per dei dati meno sensibili.

Concludiamo il discorso dicendo che a nostro parere una soluzione implementabile e attualmente oggetto di ricerca nel settore, è quella dell'interoperabilità tra Chain, e cioè a seconda della caratteristica che si ricerca per la propria applicazione originariamente nata su una determinata Blockchain con caratteristiche differenti da un'altra, per la quale però ad un certo punto si volessero sfruttare altri aspetti come la sicurezza, ad esempio, si potrebbe rendere un protocollo multipiattaforma. Riteniamo che l'interoperabilità sarà uno dei temi che guiderà il prossimo mercato rialzista.

Per coloro che sono interessati ad esplorare ulteriormente il nostro lavoro o a sviluppare progetti simili, il nostro codice è disponibile su GitHub. Speriamo che le nostre scoperte e le soluzioni che abbiamo implementato possano essere di aiuto per altri ricercatori e sviluppatori nel campo delle reti blockchain. //Testnet [amriescuo/Hedera-cost-analysis1 (github.com)]

In definitiva, questo progetto è stato una preziosa opportunità di apprendimento e ci ha permesso di fare un passo avanti nel nostro viaggio di comprensione delle tecnologie blockchain.

