

Hedera

Analisi del sistema Hedera e dei costi di esecuzione delle applicazioni



PROGETTO E SVILUPPO DI APPLICAZIONI BLOCKCHAIN

Versione luglio 2023

*University of Cagliari, Italy*

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

Docente del Corso: Andrea Pinna

Luigi Mereu 70/87/46312 Mirco Aresu 60/73/65283

**

*ABSTRACT*

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.

*Indice*

*Frontespizio………………………………………………………………………………………………………………………………………………………*[0]

*Abstract ………………………………………………………………………………………………………………………………………………………….* [1]

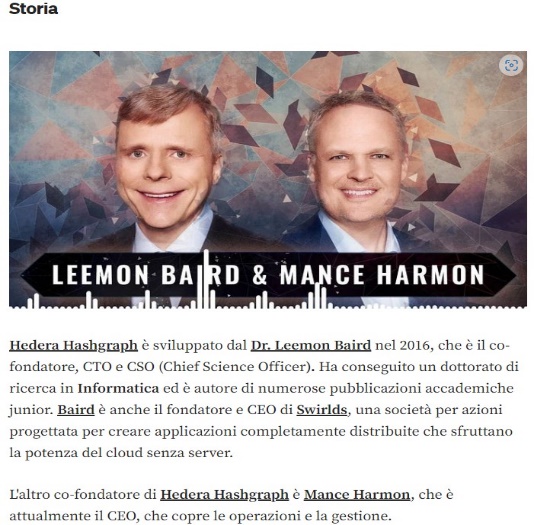
***Sezione 1*** *………………………………………………………………………………………………………………………………………………………..* [2]

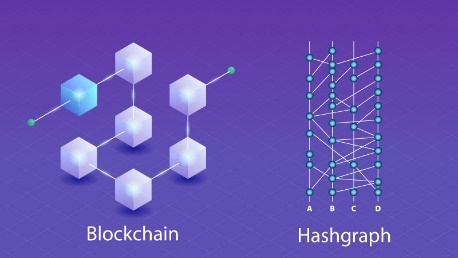
* + *HEDERA - BREVE PANORAMICA NEL SETTORE………………………………………………………………………………………...* [2]
  + *HEDERA - ENTERPRISE DISTRIBUITED LEDGER………………………………………………………………………………………* [3]
  + *ALGORITMO DI CONSENSO………………………………………………………………………………………………………………….* [4]
  + *HBAR - TOKEN ED ECONOMIA DELLA RETE……………………………………………………………………………………………* [5]
* ***Sezione 2…………………………………………………………………………………………………………………………………*** [6]
  + *CASI D’USO ED ANALISI DEGLI SMART CONTRACT…………………………………………………………………………………..* [6]
    - *Simple Transfer…………………………………………………………………………………………………………………………….* [7]
    - *Data Storage………………………………………………………………………………………………………………………………..* [8]
    - *CrowdFund………………………………………………………………………………………………………………………………….* [9]
    - *HTLC…………………………………………………………………………………………………………………………………………*[10]
* ***Sezione 3…………………………………………………………………………………………………………………………………*** [11]
  + *IMPOSTAZIONE DELL’AMBIENTE DI SVILUPPO HEDERA E DEPLOY DEGLI SMART CONTRACT(Ethereum-Like)…….* [11]
    - *Step 1: Creazione e definizione di un account sulla testnet di Hedera…………………………………………………………….* [11]
    - *Step 2: Installazione di Node.js………………………………………………………………………………………………………….* [12]
    - *Step 3: Creazione di un ambiente di sviluppo per testare gli Smart Contracts………..…………………………………………* [13]
    - *Step 4: Approccio modulare e deploy dello smart contract………………………………………………………………………….*[14]
      * *4.1) Compilazione del* contratto………………………………………………………………………………………..[14]
      * *4.2) Estrazione del file JSON dalla cartella “artifacts”………………………………………………………………………*[14]
      * *4.3) Salvataggio del file JSON nella cartella del Simple Transfer per la quale abbiamo costruito l’ambiente………* [14]
      * *4.4) Costruzione di un file “store” da aggiungere alla cartella…………………………………………………………….* [15]
      * *4.5) Costruzione dei file di interazione esempio: deploy, deposit, withdraw e balance………………………………….* [15]
      * *4.6) Deploy del contratto e valutazione del costo……………………………………………………………………………..* [17]
  + *EXTRA - TEST DEL SIMPLE TRANSFER NELLA MAINNET…………………………………………………………………………..* [19]
* ***Sezione 4****………………………………………………………………………………………………………………………………………………..* [23]
  + *ANALISI DEL CODICE, INTERAZIONE CON I CONTRATTI SULLA TESTNET HEDERA E COSTI……….…………………..* [23]
  + *Simple Transfer………………………………………………………………………………………………………..…….…………………*[23]
  + *Data Storage………………………………………………………………………..……………………………………………….…………* [27]
    - *Esperimento 1……………………………………………………………………………………………………………………………* [33]
    - *Esperimento 2……………………………………………………………………………………………………………………………* [34]
  + *CrowdFund……………………………………………………………………………………………………………………………………..* [35]
  + *HTLC…………………………………………………………………………………………………………………………………………….* [38]
* ***Conclusioni****…………………………………………………………………………………………………………………………………………….* [44]

***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 DISTRIBUITED 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 costruirci **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 diventarne 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 Bizantine Fault Tollerance**), 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 "**ħ**", quindi 5 ħ 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 "tħ", quindi è corretto dire 1 ħ = 100.000.000 tħ

Denominazioni ufficiali HBAR :  
​  
**gigabar** 1 Gℏ = 1,000,000,000 ℏ  
**megabar** 1 Mℏ = 1,000,000 ℏ  
**kilobar** 1 kℏ = 1,000 ℏ  
**hbar** 1 ℏ = 1 ℏ  
**millibar** 1,000 mℏ = 1 ℏ  
**microbar** 1,000,000 μℏ = 1 ℏ  
**tinybar** 100,000,000 tℏ = 1 ℏ

***SEZIONE 2***

*CASI D’USO ED ANALISI DEGLI SMART CONTRACT(Ethereum-Like)*

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.

Abbiamo riportato in questa sezione gli smart contract prima di essere riadattati per hedera, giusto per capirne la logica prima dell’implementazione, derivano dalla cartella privata del progetto:

[ [contracts-costs-benchmark/contracts at main · blockchain-unica/contracts-costs-benchmark (github.com)](https://github.com/blockchain-unica/contracts-costs-benchmark/tree/main/contracts)], nelle cartelle “ethereum”.

**Abbiamo preso in considerazione i seguenti:**

**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:



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.

**Data Storage**

***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 h24/7.*

***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.

******

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.

**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 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.



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 chehanno 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.

***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:



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*.

***SEZIONE 3***

*IMPOSTAZIONE DELL’AMBIENTE DI SVILUPPO HEDERA E DEPLOY DEGLI SMART CONTRACT*

*In questa sezione* ***spiegheremo*** *in maniera dettagliata, attraverso i vari* ***step****, diverse metodologie per* ***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 compositi (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:

Immagine che contiene testo, schermata, software, Sistema operativo

Descrizione generata automaticamente

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

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

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 fittizi 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.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamentePer 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 *promt* dei comandi per verificare la corretta installazione dei software e la relativa versione:

Immagine che contiene testo, software, Software multimediale, schermata

Descrizione generata automaticamente

***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:

Immagine che contiene testo, Carattere, software, schermata

Descrizione generata automaticamente

dentro la quale inseriremo il file del contratto scritto inSolidity, 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:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

A questo punto installeremo le dipendenze e gli **SDK**:Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

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*.

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



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**

Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamenteIl risultato sarà il seguente:

Immagine che contiene testo, Carattere, numero, linea

Descrizione generata automaticamentePossiamo 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 d’esempio e la cartella del progetto fino ad ora.

***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, eseguiamo un esempio per il caso di simple transfer, in questo caso non vediamo nel dettaglio il codice ma solo come procedere, le figure son quindi solo a scopo illustrativo:

*Immagine che contiene testo, schermata, software, Carattere

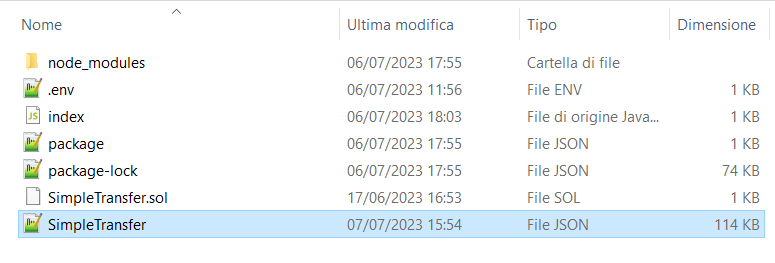
Descrizione generata automaticamente4.1) Compilazione del contratto*

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

*Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente*

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

**

*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 ***store.js****:*

Lo script **store.js** carica il **bytecode** di un contratto intelligente sulla rete test di **Hedera Hashgraph**. Utilizza le **chiavi dell'operatore** per autenticare ed eseguire le transazioni. Inizia creando un **FileCreateTransaction** con il bytecode del contratto. Successivamente, esegue questa transazione sul client, ottenendo una ricevuta. Infine, recupera l'**ID del file** dal **fileReceipt** e lo stampa. Questo ID del file è fondamentale poiché identifica il contratto sulla blockchain.

*4.5) Costruzione dei file di interazione esempio: 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****:*  
Lo script **deploy.js** **distribuisce** un contratto intelligente sulla rete di test di **Hedera Hashgraph**. Esso utilizza la **chiave privata dell'operatore** per autenticare ed eseguire le transazioni. Inizia convertendo l'**ID dell'account destinatario** in un formato compatibile con la blockchain utilizzando la funzione **toBytes32()**. Successivamente, crea una nuova **ContractCreateTransaction**, impostando l'**ID del file bytecode**, il **gas** e i **parametri del costruttore**. Quest'ultimo include l'ID dell'account destinatario trasformato. Infine, esegue questa transazione sul client, ottiene una ricevuta e recupera l'**ID del contratto** da quella ricevuta, stampando l'ID del nuovo contratto. Qui la cosa importante da notare e che stiamo facendo la deploy con l’ID ottenuto dallo store e l’account 2 dell’env che corrisponde al destinatario.

Script ***deposit.js****:*

Lo script **deposit.js** esegue una **funzione di deposito** su un contratto intelligente sulla rete di test di **Hedera Hashgraph**. Utilizza la **chiave privata dell'operatore** per autenticare ed eseguire le transazioni. Crea una nuova **ContractExecuteTransaction**, impostando l'**ID del contratto**, il **gas**, la **funzione** da eseguire ("deposit") e l'**importo** da depositare. Infine, esegue la transazione sul client, ottiene una ricevuta e stampa lo **stato** del deposito dalla ricevuta. L'importo del deposito e l'ID del contratto vengono passati come parametri alla funzione **deposit()**. Questo script tuttavia non ha dato troppi problemi durante il testing

Script ***withdraw.js****:*

Lo script **withdraw.js** esegue una funzione di **prelievo** su un contratto intelligente sulla rete di test di Hedera Hashgraph. Viene utilizzata la **chiave privata dell'operatore** per autenticare ed eseguire le transazioni. Lo script crea una nuova **ContractExecuteTransaction**, impostando l'**ID del contratto**, il **gas**, la **funzione** da eseguire ("withdraw") e l'**importo** da prelevare e l'**ID del prelevatore**. Successivamente, esegue la transazione sul client, ottiene una ricevuta e stampa lo **stato** del prelievo dalla ricevuta. L'importo del prelievo, l'ID del contratto e l'ID del prelevatore vengono passati come parametri alla funzione **withdraw()**.

Nel contesto di Hedera Hashgraph e degli smart contract Ethereum, un **bytes32** è un tipo di dato che può contenere una sequenza di 32 byte. Viene spesso utilizzato per rappresentare un hash o un identificatore univoco.

Nel codice degli script **deploy.js** e **withdraw.js**, la funzione **toBytes32()** viene utilizzata per convertire un **accountId** in un formato **bytes32**. Questo perché gli ID di account Hedera, essendo stringhe, devono essere convertiti in un formato **bytes32** per essere utilizzati come parametri negli smart contract Ethereum.

Nella funzione **toBytes32()**, viene creata una nuova istanza di **Buffer** a partire dall'**accountId**, e se la lunghezza del buffer supera i 32 byte, viene generato un errore. Successivamente, viene creato un nuovo buffer **bytes32** di lunghezza 32 e il buffer originale viene copiato in esso. Il buffer **bytes32** viene quindi restituito dalla funzione.

È importante notare che questa conversione è necessaria perché il tipo di dati **bytes32** di Solidity non corrisponde direttamente al tipo di dati **string** di JavaScript; quindi, questa conversione garantisce che l'**accountId** possa essere correttamente gestito dallo smart contract.

Script ***balance***:

Questo script Node.js controlla e stampa il saldo di tre diversi account Hedera Hashgraph. Utilizza la libreria SDK di Hedera Hashgraph per interagire con la rete Hedera.

Innanzitutto, viene impostato l'operatore del client utilizzando l'ID dell'operatore e la chiave privata, che vengono recuperati dalle variabili di ambiente.

La funzione **checkAccountBalance()** crea una nuova **AccountBalanceQuery**, un tipo di interrogazione che può ottenere il saldo di un account Hedera. Questa interrogazione viene impostata con l'ID dell'account di cui si vuole conoscere il saldo usando il metodo **.setAccountId()**.

Successivamente, la query viene eseguita utilizzando il metodo **.execute()** del client. Il saldo restituito viene quindi stampato sulla console.

Infine, la funzione **checkAccountBalance()** viene chiamata per ciascuno dei tre account specificati. *Se si verifica un errore durante l'esecuzione della query di saldo, questo viene catturato e stampato sulla console con* ***console.error****.*

*In cui ID\_1 è l’owner, ID\_2 il recipient e ID\_3 un account generico.*

Immagine che contiene testo, schermata, numero, software

Descrizione generata automaticamenteStato della cartella *Simple Transfer,* come appare il progetto al momento:

Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamenteStato del file *.env*, ricordiamo con tutti gli account necessari per usufruire di tutte le funzionalità dello smart contract:

A questo punto possiamo procedere per fare la deploy vera e propria, intendiamo, eseguire gli script. Per prima cosa apriamo il promt 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(parametro della funzione di deploy), per poi eseguirlo sempre da terminale.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

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.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

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*](mailto:0.0.14818703@1688818052.771279330)per ottenere le informazioni dei costi che stavamo cercando.

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.*

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Tuttavia, questa tecnica è utile giusto per verificare il corretto funzionamento, il nostro tool vuole stampare questi costi a schermo e questo verrà discusso nella sezione 4.

**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)

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

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.

Immagine che contiene schermata, testo, Software multimediale, software

Descrizione generata automaticamenteNell’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.

Immagine che contiene schermata, testo, software, Software multimediale

Descrizione generata automaticamente

*Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente*

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

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.

Immagine che contiene testo, elettronica, schermata, software

Descrizione generata automaticamente

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.

Immagine che contiene testo, software, numero, Software multimediale

Descrizione generata automaticamente

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***

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

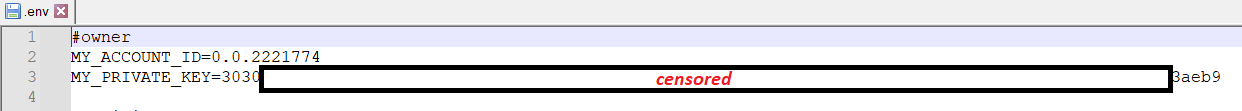
***Righe di codice modificate nel file deploy.js***

***Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente***

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:***



***Esecuzione della store e della deploy***

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

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

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

E infine aprire la pagina relativa ai costi del deploy:

Immagine che contiene testo, schermata, software, Carattere

Descrizione generata automaticamente

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*

Consideriamo il fattore di conversione **1 HBAR = 0,056290 USD**

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** con **CONTRACT\_REVERT\_EXECUTED**, indicando con

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



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

 ***Specifiche :*** *“The contract Storage allows a user to store insiede the blockchain two typologies of dynamic size data: a byte sequence and a string. After contract creation, the contract allows two actions:*

*-* ***\*\*storeBytes\*\*,*** *which allows the user to store an arbitrary sequence of bytes;*

*-* ***\*\*storeString\*\*,*** *which allows the user to store a string of arbitrary length.”*

*Stato della cartella Simple Transfer nella versione modulare:*

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

Possiamo provare le funzioni del contratto caricato nel Network nella sez*ione 3,* una volta aver sostituito negli script di *deposit* e *withdraw* l’ID del contratto precedentemente creato, due esempi su come usarlo:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamente

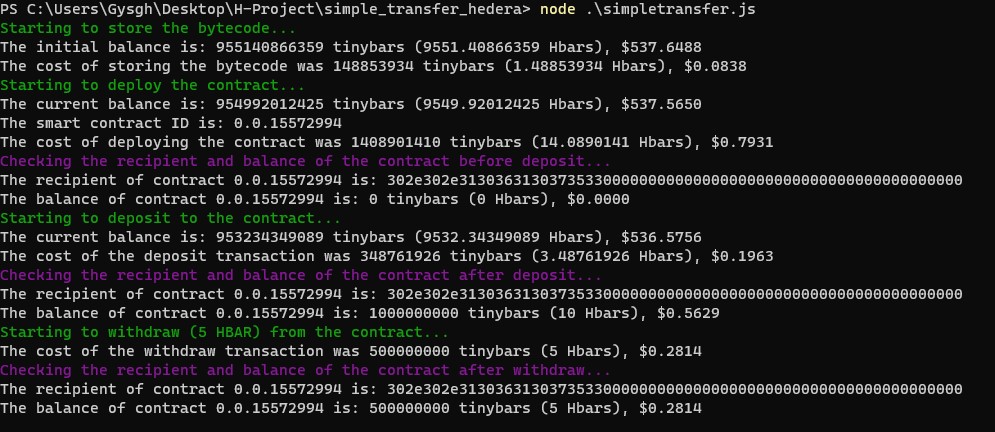
Per capire cos’è stato fatto in maniera più intuitiva descrivo la versione dello script unimodulare simpletransfer.js : Il codice fornisce una panoramica di come utilizzare l'SDK Hashgraph per interagire con un contratto intelligente su Hedera. Esso include le fasi di creazione, distribuzione, deposito e prelievo.

* **dotenv**: È quindi un modulo che carica variabili di ambiente da un file .env in process.env. **require("dotenv").config();** carica le variabili di ambiente.
* **@hashgraph/sdk**: Il pacchetto SDK per interagire con la rete Hedera Hashgraph. Fornisce funzionalità per la creazione di client, transazioni, query e altri componenti necessari per interagire con Hedera.
* **PrivateKey.fromString(process.env.MY\_PRIVATE\_KEY)**: Questa funzione converte una stringa di chiave privata nel formato corretto per l'SDK.
* **Client.forTestnet()**: Crea un nuovo client per interagire con il testnet di Hedera.
* **client.setOperator(operatorId, operatorPrivateKey)**: Imposta l'account e la chiave privata dell'operatore per il client. L'operatore è l'account utilizzato per pagare le transazioni eseguite dal client.
* **toBytes32(accountId)**: Funzione per convertire un ID account Hedera in un valore bytes32.(Padding di zeri fino a riempirlo)
* **getBalance(accountId)**: Una funzione che esegue una query per ottenere il saldo di un account.
* **tinybarsToHbar(tinybars)** e **hbarToUSD(hbars)**: Funzioni di utilità per convertire tra diverse unità di HBAR, hint: 8 zeri per passare dall’uno all’altro e abbiamo usato ***1 HBAR = 0,056290 USD.***
* ***printInGreen(message)*** *e* ***printInMagenta(message)****: Funzioni di utilità per stampare messaggi colorati sulla console.*
* **main()**: Questa è la funzione principale che gestisce l'interazione con il contratto intelligente. Include la creazione di un file per memorizzare il bytecode del contratto, la distribuzione del contratto, il deposito di HBAR nel contratto, il prelievo di HBAR dal contratto e l'esecuzione di query sul contratto per ottenere informazioni come il saldo e il destinatario.

Le funzioni principali per interagire con il contratto includono:

* **FileCreateTransaction().setContents(bytecode)**: Crea un nuovo file su Hedera che contiene il bytecode del contratto.
* **ContractCreateTransaction()**: Crea una nuova transazione per la distribuzione di un contratto su Hedera.
* **ContractExecuteTransaction()**: Esegue una funzione del contratto, come depositare o prelevare fondi.
* **ContractCallQuery()**: Esegue una query su un contratto, come ottenere il saldo o il destinatario del contratto.
* Infine, **main().catch(console.error)**; esegue la funzione principale e cattura eventuali errori che potrebbero verificarsi durante l'esecuzione.

Ognuna di queste funzioni può esser vista come uno singolo script come riportato nella pagina precedente, nel nostro tool sono implementati anche come script già pronti a uno a uno ma nella pagina successiva riportiamo un esempio di esecuzione non modulare ben strutturata dello script simpletransfer.js.

**Simpletransfer.js**

| **Action** | **Cost ℏ (Hbars)** | **Cost $ (USD)** | **Descrizione** |
| --- | --- | --- | --- |
| Initial balance | N/A | N/A | Il saldo iniziale dell'owner è 9507.38 Hbar. |
| Store bytecode | 1.49 | 0.0838 | Il contratto viene compilato e il bytecode viene memorizzato su Hedera. |
| Deploy contract | 14.09 | 0.7931 | Il contratto viene implementato sulla rete Hedera. |
| Deposit | 3.49(-10) | 0.1963(-0.5629) | L'owner deposita 10 Hbar nel contratto. |
| Withdraw | 1.51(+5) | 0.0851(+0.2814) | Il recipient ritira 5 Hbar dal contratto. |

I costi esatti possono variare leggermente a causa di vari fattori, tra cui la rete Hedera Hashgraph stessa e le fluttuazioni del tasso di cambio tra Hbar e $.

DataStorage

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



Immagine che contiene testo, software, Pagina Web, Sito Web

Descrizione generata automaticamenteAnche in questo caso vediamo come son stati eseguiti tutti i seguenti passaggi:

Abbiamo la **store.js** identica a quella della simpletransfer



* **FileCreateTransaction()**: Crea un nuovo file su Hedera che contiene il bytecode del contratto.

**deploy.js:**

****

* **ContractCreateTransaction()**: Crea una nuova transazione per la distribuzione del contratto.

**storeString.js :**  


* **ContractExecuteTransaction()**: Esegue una funzione del contratto, in questo caso stiamo facendo la store della stringa “Hello, world!”

**storeBytes.js :**

****

* **ContractExecuteTransaction()**: Esegue una funzione del contratto, in questo creiamo un buffer di dati da memorizzare nel contratto. In questo caso, stiamo creando un buffer che contiene la stringa "Hello" in formato ASCII.

La stringa "Hello" è rappresentata in codifica ASCII. Ogni carattere in ASCII ha un corrispondente valore numerico. Ecco i valori ASCII per "Hello":

"H": 72 (0x48 in esadecimale)

"e": 101 (0x65 in esadecimale)

"l": 108 (0x6c in esadecimale)

"l": 108 (0x6c in esadecimale)

"o": 111 (0x6f in esadecimale)

**getString\_And\_Bytes.js**  


* **ContractCallQuery()**: Esegue una query su un contratto, come leggere lo stato del contratto o chiamare una funzione che non modifica lo stato del contratto."

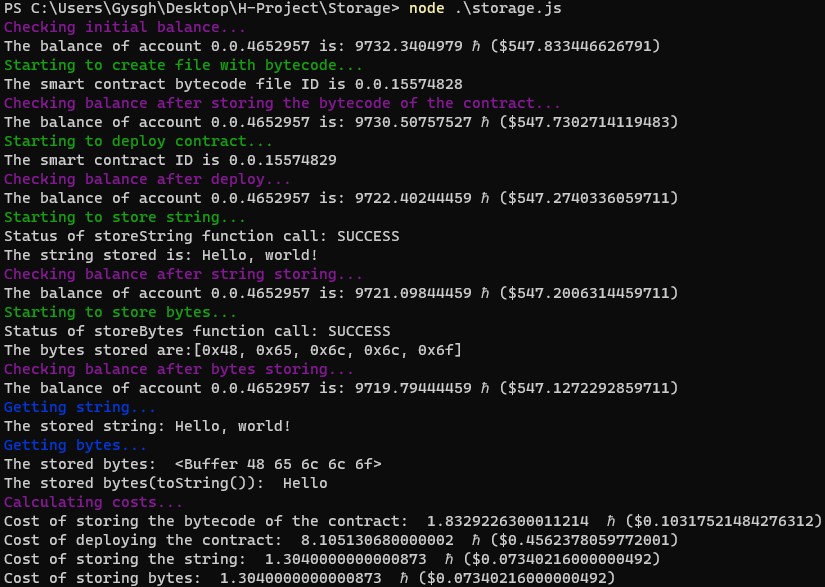
In questa porzione di codice, "ContractCallQuery()" è utilizzato due volte, una volta per ottenere una stringa dal contratto e un'altra volta per ottenere una sequenza di byte. Entrambe queste operazioni sono di sola lettura e non modificano lo stato del contratto; quindi, vengono effettuate tramite una query piuttosto che con una transazione.

**balance.js**

**getBalance(accountId)**, nel tool è sempre presente uno script pronto per ottenere il balance dell’account o degli account usati per l’operazione

**Esperimento 1**

**storage.js**

Abbiamo creato un file unico che comprende tutto il lavoro svolto da ogni singola funzione per fare l’analisi dei costi del contratto con gli stessi dati precedenti, ottenendo il seguente log:

Dalle stampe di log del nostro output, possiamo vedere che il costo delle operazioni viene calcolato correttamente in Hbar e USD. Qui è il riassunto in forma tabellare:

| **Operazione** | **Costo (in Hbar)** | **Costo (in USD)** |
| --- | --- | --- |
| Storing the bytecode of the contract | 1.83 ℏ | $0.103 |
| Deploying the contract | 8.11 ℏ | $0.456 |
| Storing the string | 1.30 ℏ | $0.073 |
| Storing bytes | 1.30 ℏ | $0.073 |

Questi numeri vengono approssimati all'ultima cifra significativa. I costi esatti possono variare leggermente a causa di vari fattori, tra cui la rete Hedera Hashgraph stessa e le fluttuazioni del tasso di cambio tra Hbar e $.

**Esperimento 2**

Per Data Storage ci aspettiamo qualche prova con diverse dimensioni del testo e del dato in bytes



Di seguito è presentato un riassunto dei costi per ogni esecuzione del tuo script di test:

| **Lunghezza della stringa** | **Costo memorizzazione stringa (ℏ)** | **Costo memorizzazione stringa ($)** | **Costo memorizzazione byte (ℏ)** | **Costo memorizzazione byte ($)** |
| --- | --- | --- | --- | --- |
| 12 | 1.2880000000004657 | $0.07250152000002621 | 1.2879999999986467 | $0.07250151999992382 |
| 13 | 1.2880000000004657 | $0.07250152000002621 | 1.2879999999986467 | $0.07250151999992382 |
| 14 | 1.2880000000004657 | $0.07250152000002621 | 1.2879999999986467 | $0.07250151999992382 |
| 15 | 1.2880000000004657 | $0.07250152000002621 | 1.2880000000004657 | $0.07250152000002621 |

Da quanto si può osservare, il costo delle operazioni di memorizzazione sembra rimanere costante indipendentemente dalla lunghezza della stringa o dei byte che stai memorizzando, aumentiamola:

**30 caratteri:**

Stringa: "abcdefghijklmnopqrstuvwxyzabcd"

Byte: Buffer.from([0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x61, 0x62, 0x63, 0x64])

**40 caratteri:**

Stringa: "abcdefghijklmnopqrstuvwxyzabcdefghijkl"

Byte: Buffer.from([0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70])

**50 caratteri:**

Stringa: "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz"

Byte: Buffer.from([0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66])

**104 caratteri:** *(come i 50 caratteri ma x2)*

| **Lunghezza della stringa** | **Costo memorizzazione stringa (ℏ)** | **Costo memorizzazione stringa ($)** | **Costo memorizzazione byte (ℏ)** | **Costo memorizzazione byte ($)** |
| --- | --- | --- | --- | --- |
| 30 | 1.2960000000002765 | $0.07295184000001556 | 1.2960000000002765 | $0.07295184000001556 |
| 40 | 1.2960000000002765 | $0.07295184000001556 | 1.2960000000002765 | $0.07295184000001556 |
| 50 | 1.2960000000002765 | $0.07295184000001556 | 1.2959999999984575 | $0.07295183999991317 |
| 104 | 1.2960000000002765 | $0.07295184000001556 | 1.2960000000002765 | $0.07295184000001556 |

Dall’analisi si può notare come il costo di memorizzazione in byte aumenta all’aumentare della lunghezza della stringa.

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.  
  

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 son 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 eseguire lo script.*

Quindi essenzialmente a livello di codice svolgiamo un lavoro molto simile alla simpletransfer, la differenza sostanziale sta nei timestamp limitati per svoglere le interazioni

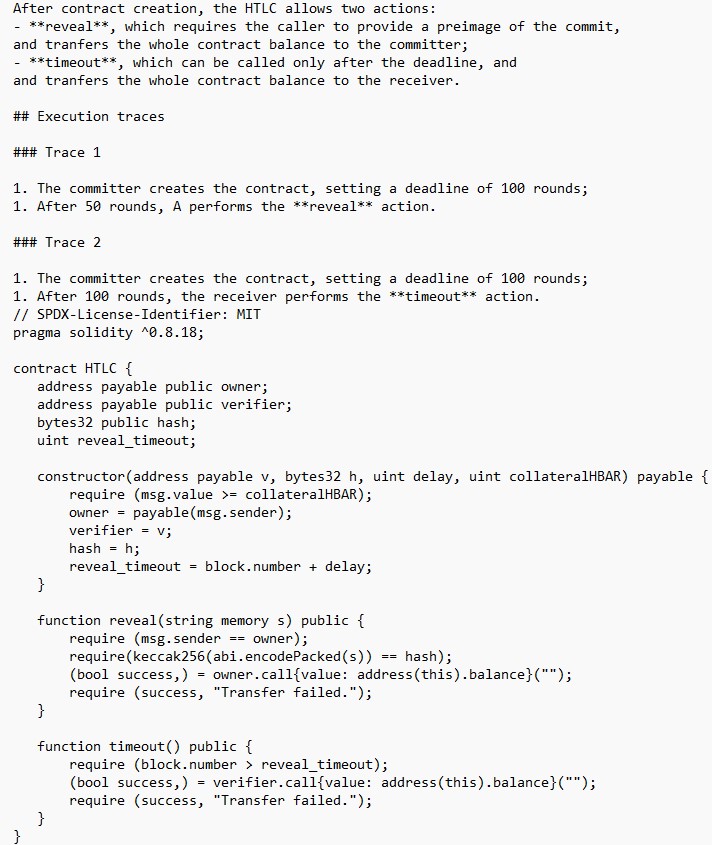
**Logs:**

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

| **Action** | **Cost ℏ (Hbars)** | **Cost $ (USD)** | **Description** |
| --- | --- | --- | --- |
| Initial balance | N/A | N/A | L'owner ha un saldo iniziale di 9507.38 Hbar. |
| Deploy contract | 14.07 | 0.7903 | Il contratto viene implementato sulla rete Hedera. |
| Donate | 3.49(-110) | 0.1963(2.8145) | L'owner dona 110 Hbar al contratto. |
| Wait | 0 | 0 | Attesa di 60 secondi prima di procedere con il prelievo. |
| Withdraw | 1.51(+110) | 0.0848(2.8145) | Il recipient ritira 110 Hbar dal contratto. |
| Deploy contract | 14.07 | 0.7903 | Viene implementato un nuovo contratto sulla rete Hedera per testare la funzione "reclaim". |
| Donate | 3.49(-50) | 0.1963(2.8145) | L'owner dona 50 Hbar al nuovo contratto. |
| Wait | 0 | 0 | Attesa di 60 secondi prima di procedere con la reclamazione. |
| Reclaim | 1.51(-50) | 0.0848(2.8145) | L'owner recupera 50 Hbar dal contratto. |

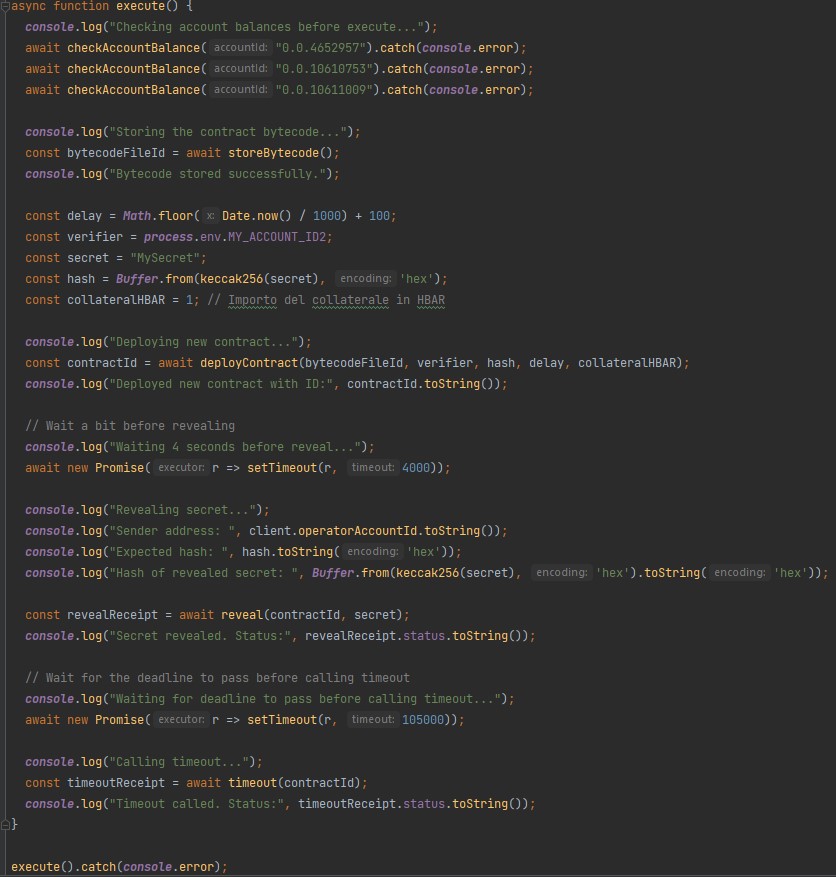
HTLC

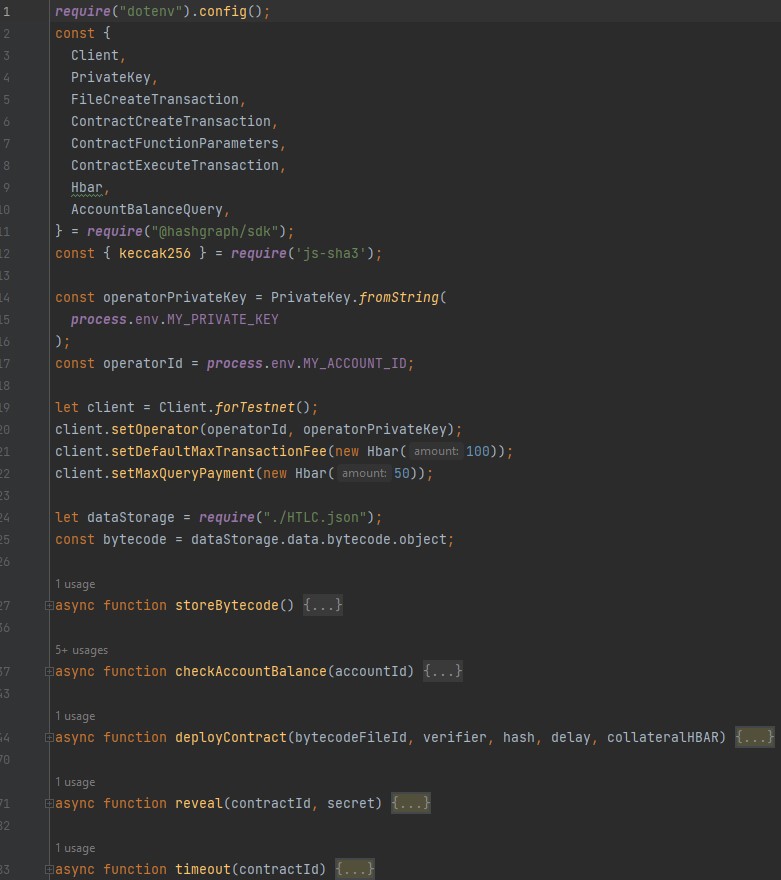
Versione di contratto riadattato:



L'Hash Timed Locked Contract (HTLC) coinvolge due utenti, consentendo di impegnarsi su un segreto per poi rivelarlo. Alla creazione, si deposita un collaterale e si definisce una scadenza. L'HTLC offre due azioni: **reveal** per rivelare il segreto trasferendo il saldo al committente, e **timeout** per trasferire il saldo al ricevitore dopo la scadenza.

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:**



**Struttura centrale**

Lo script esegue le seguenti funzioni:

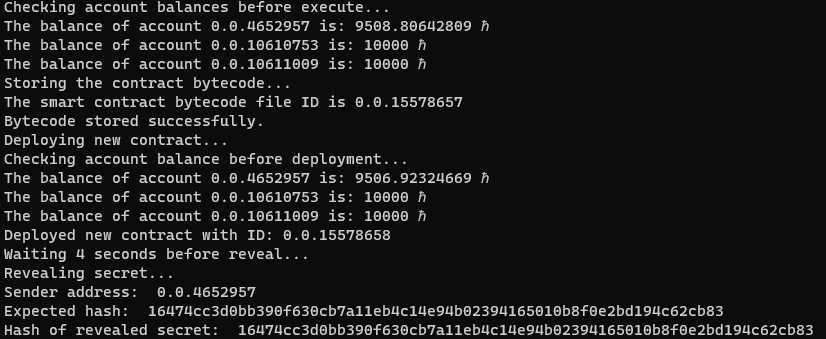
1. **Caricamento dei moduli necessari:** Il modulo **dotenv** viene utilizzato per caricare variabili d'ambiente dal file **.env** nel **process.env**. Il modulo **@hashgraph/sdk** contiene metodi per interagire con la rete Hedera, mentre il modulo 'js-sha3' fornisce una funzione per calcolare il Keccak-256 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.

| ***Azione*** | ***Costo ℏ (Hbars)*** | ***Costo $ (USD)*** | ***Descrizione*** |
| --- | --- | --- | --- |
| ***Saldo Iniziale*** | ***N/A*** | ***N/A*** | ***Il saldo iniziale dell'account owner è 10000 Hbar.*** |
| ***Memorizzazione bytecode*** | ***1.82259884*** | ***0.1026*** | ***Il contratto viene compilato e il bytecode viene memorizzato su Hedera.*** |
| ***Deploy*** | ***8.0594135*** | ***0.4537*** | ***Il contratto viene implementato sulla rete Hedera.*** |
| ***Rivelazione segreto*** | ***1.296*** | ***0.0729*** | ***L'owner esegue la funzione 'reveal' per rivelare il segreto.*** |
| ***Timeout*** | ***1.296*** | ***0.0729*** | ***L'owner esegue la funzione 'timeout'.*** |

******

*RACCOLTA DATI E ANALISI DEI COSTI DA HASHGRAPH*

Di seguito si riportano i costi di deploy degli smart contract proposti cercando di dare un senso ai risultati ottenuti, verifichiamo se ci siamo avvicinati ai valori ottenuti con le fee reali, ricordiamo che noi abbiamo usato la regola di conversione [**1 HBAR = 0,056290 USD]**, tra parentesi quadre indichiamo quindi quanto ci esce all’incirca dal nostro tool:

***Costo deploy Simple Transfer: [La nostra regola di conversione: ~ 14]***

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

***Costo deploy Data Storage: [La nostra regola di conversione: ~ 8]***

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

***Costo deploy Crowdfund: [Con la nostra regola di conversione: ~ 15]***

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

***Costo deploy HTLC: [Con la nostra regola di conversione: ~ 9]***

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

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 [amrierscuo/Hedera-cost-analysis1 (github.com)]**

Immagine che contiene computer, computer, Tastiera del computer, interno

Descrizione generata automaticamente**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.**