

Ethereum Dapp

Silvia Zandoli: silvia.zandoli2@studio.unibo.it

Progetto per l'anno di corso 2019/2020

Contents

1	Introduzione	4
1.1	Cos'è una Blockchain?	4
1.2	Permissioned vs Permissionless Blockchain	5
1.3	Definizione di Smart Contract	7
2	Obiettivi	8
2.1	Piano di lavoro	8
2.2	Scenari	9
2.3	Politica di autovalidazione	10
3	Analisi dei Requisiti	10
3.1	Requisiti funzionali	11
3.2	Requisiti non funzionali	12
4	Stato dell'arte	12
4.1	Architettura di Ethereum	12
4.2	I nodi di mining	13
4.2.1	Come funziona il mining	14
4.3	Gli Account	14
4.3.1	Gli Account di proprietà esterna	14
4.3.2	Contract Accounts	15
4.4	Transazioni	15
4.5	Blocchi	16
4.6	Una transazione end to end	17
4.7	Ciclo di vita di uno Smart Contract in Ethereum	18
4.8	Recap di ciò che ho utilizzato	18
5	Design	20
5.1	Struttura	20
5.1.1	Dominio applicativo	21
5.2	Client Node	22
5.3	Comportamento	22
5.4	Interazione	23
6	Dettagli implementativi	23
6.1	Gli Smart Contract in Solidity	23
6.2	Client NodeJs	26
7	Validazione	29
8	Istruzioni per il deploy	31
8.1	Come eseguire la Dapp	31
8.2	Come compilare e testare i contratti su Remix	31
8.3	Link contratti su Remix e Github	35
9	Esempi d'uso Dapp	35
10	Interazione con Geth	39
10.1	Geth console	39
10.2	Interazione con Geth	39
10.2.1	Deploy di un contratto	40
10.2.2	Invio di una transazione	40

10.2.3 Modifica account di default	41
11 Conclusioni	42
11.1 Lavori futuri	42
11.2 Cosa ho imparato	42

1 Introduzione

Questa sezione sarà dedicata a contestualizzare il lavoro svolto per questo progetto.

Si introdurrà l'emergente tecnologia Blockchain, per poi definire gli Smart Contract ed infine la piattaforma Ethereum.

In questo capitolo introduttivo sarà riportato il lavoro svolto nella fase di studio individuale e di approfondimento.

1.1 Cos'è una Blockchain?

Quando si parla di Blockchain si pensa subito al Bitcoin. Questo perchè il Bitcoin è stato il più grande caso di successo della tecnologia. La Blockchain è l'infrastruttura che sta sotto il Bitcoin (da cui ha preso il nome) e anche alle altre cryptocurrency.

La Blockchain è un modello di computazione distribuita, decentralizzata e replicata.

Essa è basata su una struttura dati a blocchi. La Blockchain salva i dati in un registro distribuito (Libro Mastro), che viene replicato tra i partecipanti di una stessa rete.

Si tratta di un'architettura peer to peer decentralizzata, dove ogni partecipante sarà in possesso di una copia degli stessi dati. Tali dati vengono distribuiti tra i partecipanti, che saranno i nodi della rete.

Per quanto riguarda la sicurezza questa architettura va a limitare i rischi derivanti da errori o modifiche sui dati, ma anche da possibili attacchi a singoli database.

Esistono due grandi famiglie di Blockchain. Quelle pubbliche (Permissionless) e quelle private (Permissioned). Le prime garantiscono a tutti i partecipanti una totale trasparenza sui dati presenti sulla rete, invece nelle private i partecipanti hanno diversi livelli di visibilità sui dati. In questo caso per entrare a far parte della rete può essere richiesta una forma di autorizzazione oppure semplicemente si avrà un accesso limitato a solo alcuni dei dati presenti sulla rete. Recentemente stanno sempre più emergendo quelle ibride che cercano di tranne i vantaggi delle Blockchain pubbliche e private.

Per quanto riguarda la crittografia, Blockchain usa la firma digitale e l'hashing.

La firma digitale usa un sistema a chiave pubblica per codificare e certificare i messaggi inviati. Ogni partecipante ha dunque due chiavi: una privata e segreta che vedrà soltanto lui e l'altra pubblica, quindi accessibile da chi è autorizzato a comunicare con lui. Con una chiave il messaggio viene criptato e con l'altra viene decriptato rendendolo comunque leggibile. Ogni partecipante possiede quindi una chiave pubblica e privata. Ethereum si basa su una Blockchain di tipo permissionless quindi ogni transazione una volta scritta e sincronizzata nella Blockchain (dopo essere stata minata) può essere letta da tutti in maniera trasparente.

La seconda tecnica crittografica è l'hashing. Esso codifica ogni messaggio in input in messaggio in codice. Il messaggio in codice resterà sempre uguale tutte le volte che gli passeremo lo stesso dato in input. Ma se proviamo a cambiare anche solo una lettera o aggiungiamo una virgola il messaggio cambierà totalmente. Il secondo motivo per cui l'hashing è importante è perchè possiamo passargli qualsiasi tipo di dato in input. Indipendentemente dalla sua grandezza o complessità, il risultato sarà sempre un messaggio in codice con lo stesso formato. L'hashing quindi permette anche di verificare l'uguaglianza di dati anche molto grandi semplicemente confrontando i messaggi in codice che vengono generati.

Blockchain utilizza la crittografia sia per sicurezza sia per permettere una verifica veloce e precisa dell'integrità dei dati, indipendentemente dalla loro complessità.

L'accordo sul da farsi in un contesto così distribuito si basa sul consenso. Esso è un meccanismo automatico che definisce una conoscenza comune dei processi tra i partecipanti della stessa rete di gestione e controllo di essa.

L'algoritmo di consenso più famoso utilizzato da Bitcoin e dalla prima versione di Ethereum è il Proof of Work. Un ruolo fondamentale in questo algoritmo di consenso è svolto dai minatori (miners). I miners hanno il compito di aggiungere le transazioni in Blockchain validandole. La

validazione è accettata dalla rete solo se il miner è riuscito a risolvere un problema matematico computazionalmente sempre più complesso. I miners si danno così da fare nel cercare di validare la transazione perchè chi riuscirà ad aggiungere la transazione in Blockchain si aggiudicherà una ricompensa monetaria(in ether, bitcoin, etc). Una volta che il blocco è stato aggiunto alla catena tutti i partecipanti ne riceveranno automaticamente una copia direttamente nel loro registro condiviso. Questo algoritmo però, come anticipato, richiede davvero tanta potenza di calcolo. Per questo attualmente stanno nascendo nuovi algoritmi di consenso che possono essere implementati su nuove piattaforme a seconda delle diverse esigenze dei partecipanti della rete. Ad esempio Ethereum sta migrando verso un nuovo algoritmo di consenso, che è il Proof of Stake.

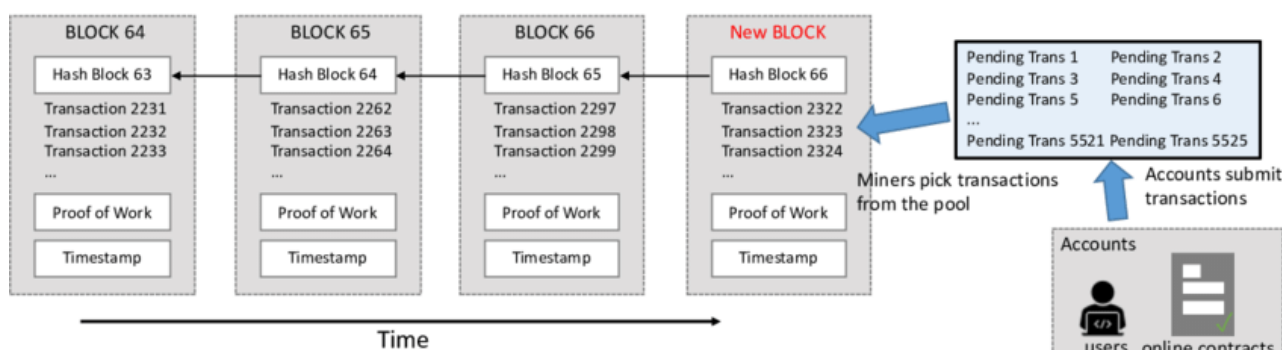
I dati di una Blockchain vengono gestiti utilizzando una catena di blocchi condivisa. Ogni blocco quindi contiene un gran numero di transazioni. Oltre ai dati i blocchi contengono altri parametri molto importanti, come ad esempio il times della creazione del blocco e il riferimento al blocco precedente.

Una volta che tutti i dati sono stati inseriti il blocco deve essere chiuso o certificato. Per fare questo il blocco viene passato in input ad un algoritmo di hashing che quindi genererà il suo messaggio in codice. Blockchain poi salverà questo codice come parametro all'interno del blocco stesso. In questo modo il blocco ora è chiuso e certificato. Quindi ogni modifica di qualsiasi dato o parametro andrà ad invalidare il blocco stesso.

Quando si genererà un secondo blocco oltre ai parametri che abbiamo già visto sarà importante includere il codice del blocco precedente. In questo modo ogni blocco farà sempre riferimento al proprio blocco precedente che verrà certificato generando il codice del nuovo blocco e così via per tutti i blocchi successivi. Con questo meccanismo Blockchain fa in modo che tutti i blocchi siano collegati tra loro. Quindi qualsiasi modifica all'interno di un blocco non solo invaliderà il blocco stesso ma invaliderà l'intera catena di blocchi.

Ovviamente un malintenzionato potrebbe modificare il blocco e rigenerare a cascata tutti gli hash della catena. Blockchain però, come abbiamo visto, mantiene sincronizzato il registro condiviso e lo fa ogni volta che un blocco viene aggiunto alla catena. Quindi per rendere la modifica efficace deve essere in qualche modo effettuata su buona parte dei nodi della rete. Questo significa che più nodi ci sono più la Blockchain garantisce persistenza dei dati limitando contraffazioni, frodi o eventuali modifiche.

Concludendo, Blockchain è quindi un tipo di "Distributed ledger technology" che quindi gestisce i dati attraverso un registro distribuito tra i partecipanti di una rete.



Struttura a blocchi di una Blockchain

1.2 Permissioned vs Permissionless Blockchain

Le Blockchain, come già accennato, vengono classificate anche in base alla presenza o assenza di permessi di accesso ad esse. Ovvero, chi è autorizzato a:

- Leggere tutti i record presenti sulla Blockchain?
- Scrivere su di essa?

- Mantenere la coesione, la stabilità e l'integrità della rete, ovvero svolgere il lavoro di miner?

Esistono principalmente due tipologie di Blockchain: Permissionless (o pubbliche) e Permissioned. Non si tratta di una classificazione rigida. Anzi, gli elementi caratterizzanti di queste declinazioni possono essere combinati, per creare registri personalizzati per applicazioni specifiche.

Le Blockchain Permissionless o Pubbliche vengono definite così perché non richiedono alcuna autorizzazione per poter accedere alla rete, eseguire delle transazioni o partecipare alla verifica e creazione di un nuovo blocco.

Le più famose sono sicuramente Bitcoin ed Ethereum, dove non vi sono restrizioni o condizioni di accesso. Chiunque può prenderne parte. Si tratta di una struttura completamente decentralizzata, in quanto non esiste un ente centrale che gestisce le autorizzazioni di accesso. Queste sono condivise tra tutti i nodi allo stesso modo. Nessun utente della rete ha privilegi sugli altri, nessuno può controllare le informazioni che vengono memorizzate su di essa, modificarle o eliminarle, e nessuno può alterare il protocollo che determina il funzionamento di questa tecnologia. Le Blockchain Permissionless sono pubbliche. Infatti sarebbe un controsenso avere una Blockchain privata dove però non viene richiesta un'autorizzazione per accedere ai dati registrati; per questo motivo tutte quelle che non richiedono un'approvazione sono definite pubbliche. Nonostante i dati registrati su queste Blockchain siano pubblici, questi vengono crittografati per mantenere un sufficiente livello di privacy. Ad esempio tutti i nodi di Bitcoin conoscono gli indirizzi wallet degli altri utenti e le transazioni che sono avvenute tra di loro. In linea di principio questi indirizzi sono semplicemente pseudonimi e, a meno che non vengano ricondotti all'identità della persona del mondo reale che ne è il proprietario, viene garantito un livello di privacy sufficiente. Un metodo per proteggere ulteriormente la propria identità consiste nell'utilizzare più di un singolo indirizzo wallet.

La principale preoccupazione legata alle Blockchain Pubbliche è il tema della scalabilità, ovvero la capacità di un sistema di migliorarsi all'aumentare del numero di partecipanti. Questo tipo di rete non è una tecnologia scalabile: al crescere della quantità di nodi, la velocità delle transazioni rimane invariata ma aumenta la stabilità del sistema che diventa così più sicuro.

Le Blockchain Permissioned invece sono soggette ad un'autorità centrale che determina chi possa accedervi. Oltre a definire chi è autorizzato a far parte della rete, tale autorità definisce quali sono i ruoli che un utente può ricoprire all'interno della stessa, definendo anche regole sulla visibilità dei dati registrati. Le Blockchain Permissioned introducono quindi il concetto di governance e centralizzazione in una rete che nasce come assolutamente decentralizzata e distribuita.

Chiamata comunemente Blockchain del Consorzio, invece di consentire a qualsiasi persona con una connessione Internet di partecipare alla verifica del processo di transazione, affida il compito ad alcuni nodi selezionati ritenuti degni di fiducia.

Un esempio esplicativo di questa tipologia di rete può essere costituito da un consorzio composto da 10 aziende, ognuna di esse collegata alla Blockchain grazie un computer. Se la società "7" ha rapporti lavorativi solo con "1", "3" e "6" condividerà le fatture solo con queste tre senza che sia necessario autorizzare le altre società a leggere i dati tra loro condivisi.

Nelle reti pubbliche come Bitcoin, viene richiesto ai miner di dimostrare una proof-of-work, ovvero svolgere un task molto dispendioso in termini di tempo ed energia. Quindi più una transazione si trova in profondità nel registro maggiore è il lavoro che un nodo deve fare per poterla eliminare o modificare e ricostruire tutti i blocchi successivi ad essa, mentre la catena continua a crescere grazie al lavoro degli altri utenti. Il protocollo di Bitcoin rende quindi estremamente svantaggioso sul piano economico tentare di riscrivere la Blockchain, garantendone così l'immutabilità.

Le Blockchain autorizzate (Permissioned) non garantiscono questa proprietà, poiché non sono in grado di assicurare l'immutabilità. Ogni volta che l'algoritmo che determina la modalità per la generazione di un nuovo blocco e del consenso non richiede ai nodi di spendere in modo irreversibile una qualche forma di energia e di tempo, in realtà ci stiamo affidando alla loro

buona fede, piuttosto che su un algoritmo prevedibile.

In una blockchain Permissioned esistono diversi livelli di accesso che riguardano:

- La lettura del registro, che può essere soggetta a diverse restrizioni come ad esempio poter visionare solo le transazioni che coinvolgono direttamente l'utente.
- La possibilità di proporre ed effettuare nuove transazioni che vengano poi validate ed inserite nella Blockchain.
- La possibilità di partecipare attivamente alla rete svolgendo l'attività di mining per la creazione di nuovi blocchi.

Mentre l'ultimo livello di accesso, quello relativo all'attività di mining, è concesso solo ad un insieme limitato di utenti in questo tipo di Blockchain, gli altri due non sono obbligatoriamente sottoposti ad un'autorizzazione.

Le caratteristiche delle Blockchain Permissioned le rendono più interessanti agli occhi delle grandi imprese e dalle istituzioni poiché vengono ritenute più sicure di quelle pubbliche e permettono di avere il livello di segretezza richiesto, controllando chi può accedervi e chi può visualizzare i dati registrati.

Le Blockchain autorizzate sono inoltre più performanti, veloci, scalabili e meno costose di quelle pubbliche, dato che hanno una dimensione e diffusione minore di esse e che le transazioni vengono verificate da un limitato numero di utenti.

1.3 Definizione di Smart Contract

A Febbraio 2019 (D.l. n.135/2018) il legislatore italiano ha formulato una definizione di Smart Contract che per definirsi tale deve:

- Essere un programma per elaboratore.
- Operare su tecnologie basate su registri distribuiti.
- L'esecuzione deve vincolare automaticamente due o più parti sulla base di effetti predefiniti dalle stesse.
- Soddisfare il requisito della forma scritta previa identificazione informatica delle parti interessate.

Uno Smart Contract è paragonabile a un contratto nella vita reale. Esso è un programma, scritto in un linguaggio di programmazione specifico (Solidity è tra questi), salvato in Blockchain. Uno Smart Contract, applicato in un contesto Blockchain, è una struttura logica immutabile, deterministica (con gli stessi input viene prodotto sempre lo stesso output), affidabile e user defined (definita da un utente).

Sono chiamati Smart Contract perché permettono ad uno sviluppatore di definire una sorta di contratto o interfaccia di operazioni da offrire agli utenti. Queste, una volta invocate, rilasceranno un determinato servizio, definito dallo sviluppatore del contratto.

Più in generale, gli Smart Contract ad esempio aiutano le persone a scambiare denaro, trasferire proprietà e qualsiasi altra cosa di valore. Ciò viene fatto in modo trasparente e senza ricorrere ai servizi di un intermediario.

La caratteristica principale di Ethereum è consentire ai partecipanti della rete di lavorare con i contratti intelligenti. L'Ether (la criptovaluta di Ethereum) viene usata per pagare le transazioni (che spesso su Ethereum coinvolgono l'uso di Smart Contract).

Il primo passaggio in un contesto reale che vuole utilizzare la Blockchain con gli Smart Contract è la stipula del contratto. Le due parti poi codificano le clausole in un linguaggio di programmazione (ad esempio Solidity).

Successivamente, lo Smart Contract viene inserito nella Blockchain (la cosiddetta fase di deploy dello Smart Contract). Il deploy è una transazione a tutti gli effetti. Come ogni transazione

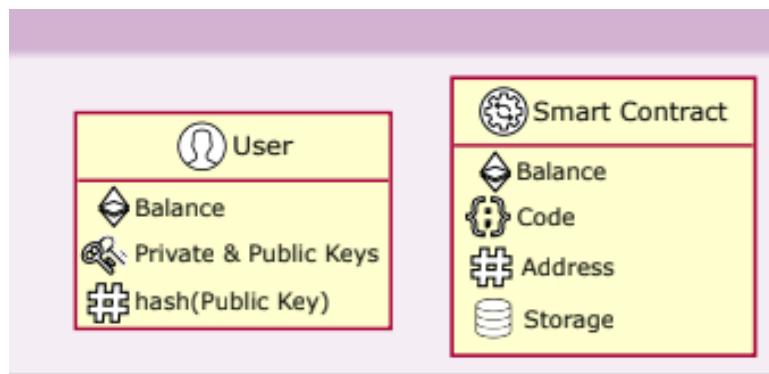
nelle Blockchain che usano l'algoritmo di consenso Proof of Work (PoW), essa avviene dopo che i nodi miners la validano per ricevere il compenso.

La rete, validate un numero sufficiente di transazioni, le inserisce in un blocco che verrà aggiunto alla catena (da qui la parola Block-Chain).

Un' applicazione basata su Smart Contract Ethereum ha la possibilità di avere un sistema per la gestione degli account, dei portafogli e dei pagamenti del tutto integrata.

Per effettuare la computazione viene definita un'unità di pagamento (Gas), che di solito è proporzionale alla complessità della computazione.

L'invocazione di una funzione di uno Smart Contract che effettuerà una transazione, determinerà il suo cambio di stato. Inoltre richiederà un consumo di Gas da parte dell'account che l'ha invocata. Più gas viene usato e più costerà la transazione. Più si è disposti a pagare per la validazione della transazione e più in fretta essa verrà minata perchè farà più gola a seguito della maggiore ricompensa.



2 Obiettivi

Il presente progetto consiste nella creazione di una Dapp la quale rappresenti un caso d'uso che permetta di valorizzare i vantaggi di una Blockchain pubblica e di risaltarne le potenzialità attraverso la creazione di Smart Contract complessi. In particolare, si è deciso di creare diversi Smart Contract che interagiscono tra di loro in un contesto puramente distribuito come quello delle Blockchain.

Un altro obiettivo del mio lavoro, svolto in maniera indipendente dalla Dapp, è stato quello di imparare a usare Geth da linea di comando per interagire con la Blockchain e con gli Smart Contract di Remix. Si troveranno diverse informazioni lungo la relazione, e in particolare il capitolo 10 sarà riservato a Geth.

2.1 Piano di lavoro

Vengono qui definite in maniera chiara le fasi del lavoro.

Per imparare a scrivere gli Smart Contract, dopo uno studio approfondito del linguaggio di programmazione Solidity, ho utilizzato Remix. Ho così iniziato su Remix la fase di sviluppo degli Smart Contract. Mi sono avvalsa di test automatizzati per verificare la correttezza degli Smart Contract. Infine ho creato i due contratti che andranno a modellare il mio dominio applicativo.

Ultimata questa fase ho sviluppato i test automatizzati utilizzando il framework Truffle e la libreria NodeJS. Ho poi verificato che il deploy degli Smart Contract vada a buon fine e ho validato il funzionamento delle funzionalità cuore del sistema, come l'aggiunta dei partecipanti al sondaggio e l'estrazione del vincitore.

Ho effettuato anche una fase di testing non automatizzato. Esso mi ha permesso di verificare il corretto funzionamento e la corretta interazione tra quello che avevo prodotto e la rete Blockchain. Ho utilizzato Ganache per eseguire gli Smart Contract prodotti precedentemente.

Poi mi sono concentrata sullo sviluppo della mia applicazione: ho creato una pagina web minimale e mi sono appoggiata a Truffle per il deploy e l'interazione tra la mia pagina e gli Smart Contract. E' stato infatti possibile interfacciarsi con la rete Blockchain mediante le funzionalità riportate all'interno degli Smart Contract sviluppati precedentemente. Per fare ciò ho utilizzato la libreria web3js che viene usata tramite Node JS.

Per evitare di pagare fee reali per le transazioni ho fatto il deploy degli Smart Contract su Ropsten.

Il wallet utilizzato per pagare le transazioni è quello di MetaMask. Per diventare nodo Ethereum ed effettuare il deploy si è usato il servizio Infura.io, evitando così di scaricarsi una copia locale della Blockchain pubblica e permettendomi di concentrarmi più sugli aspetti legati alla Blockchain e agli Smart Contract.

Ultimata la Dapp l'ultima fase è stata quella di interagire con gli Smart Contract su Remix e la Blockchain attraverso Geth. Allo stesso modo in cui avevo fatto precedentemente nella mia demo usando web3js ho effettuato il deploy dei contratti e l'esecuzione di transazioni. Ho fornito a parte una documentazione con tutti i comandi che ho utilizzato (file istruzioniGethProgetto).

2.2 Scenari

La Dapp registrerà in Blockchain, attraverso un primo Smart Contract, i dati di diversi Clienti di un importante marchio di alta moda che hanno partecipato ad un sondaggio in negozio sul loro brand.

Lo Smart Contract registrerà così i dati dell'utente in Blockchain (se e solo se avrà verificato che il codice fiscale dell'utente non esista già per quel particolare sondaggio).

Al termine del periodo predisposto al sondaggio si simulerà, con un altro Smart Contract, l'estrazione di un vincitore tra i partecipanti al sondaggio. Questi vincerà un oggetto di valore messo in palio dal famoso brand.

Solo l'amministratore del sistema potrà avviare l'estrazione del vincitore ma tutti potranno verificarlo e verificare il processo di estrazione.

Con un caso d'uso come questo si vogliono valorizzare la certezza (grazie agli Smart Contract), la trasparenza e l'immutabilità delle transazioni in un contesto decentralizzato e distribuito.

E si vuole che il processo di selezione del vincitore sia anch'esso pubblico e trasparente in modo da dare fiducia a tutti i partecipanti del sondaggio.

Sia i clienti del negozio che l'amministratore hanno a disposizione un'interfaccia web minimale e user-friendly per effettuare tutte le operazioni.

Oltre alla verifica della presenza o meno del codice fiscale, gli Smart Contract verificano che il numero di parametri passati sia corretto e che non manchino dei dati prima di avviare il mining.



Diagramma dei casi d'uso

2.3 Politica di autovalidazione

Il prodotto sviluppato sarà valutato in base a diversi criteri. Innanzitutto è fondamentale garantire l'affidabilità e correttezza degli Smart Contract prodotti. Essendo lo Smart Contract per definizione una struttura immutabile, nel caso in cui si effettuasse il deploy di un contratto "sbagliato", l'intera struttura del sistema sarebbe compromessa. Si rischierebbe così di perdere la consistenza dell'informazione. Un approccio test driven darà la possibilità di verificare il completo funzionamento.

Sarà molto importante distinguere le proprietà appartenenti al dominio applicativo per poi inserirle all'interno degli Smart Contract. Non si dovranno mai gestire problematiche di dominio all'interno dell'applicazione client. Le logiche applicative devono infatti essere univoche e deterministiche, e non di certo dipendenti dalla tecnologia di interfacciamento che si utilizza.

Sarà inoltre importante capire il funzionamento delle tecnologie usate, definendo in maniera chiara il loro scopo, funzionamento, ruolo e l'interazione tra di esse, proponendo una chiara architettura del sistema sviluppato.

Molto importante è anche lo sviluppo di un'architettura NodeJS modulare, che dà la possibilità di aggiungere facilmente nuove operazioni di seguito ad un'eventuale evoluzione degli Smart Contract.

Durante tutto l'iter di sviluppo sono stati eseguiti innumerevoli test automatizzati e non.

La correttezza dei risultati di progetto è quindi verificata e garantita dall'esito positivo dei test.

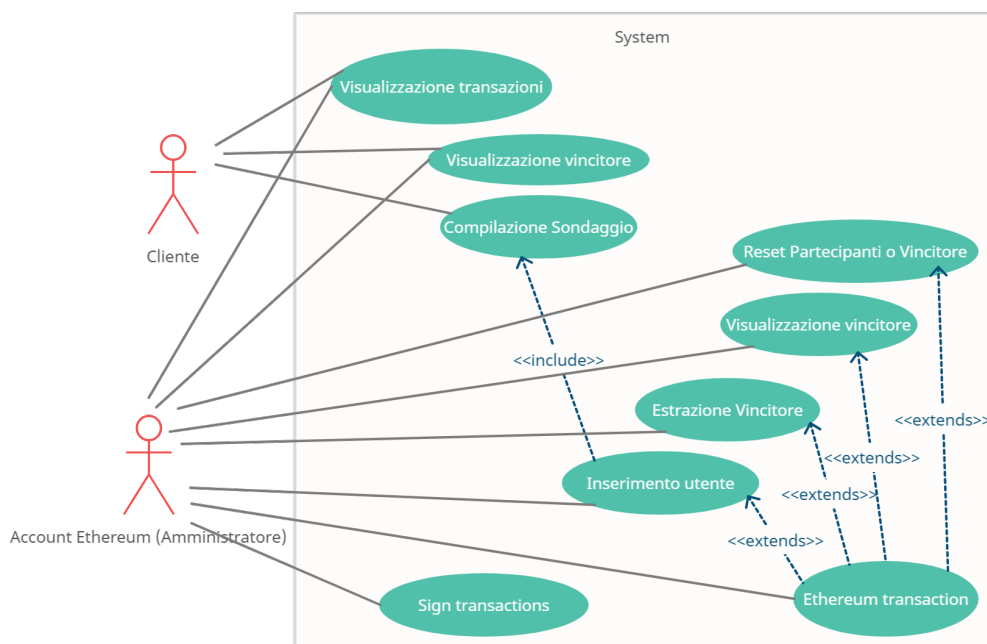
3 Analisi dei Requisiti

L'obiettivo del progetto per il corso di Sistemi Distribuiti è sviluppare una Dapp per un negozio di abbigliamento di alta moda che gestisce un concorso a premi. L'applicazione sarà basata sulla Blockchain pubblica di Ethereum.

Ethereum è un framework open-source e permette la creazione e pubblicazione di Smart Contract per creare applicazioni decentralizzate general purpose ed account based.

Si vuole dare la possibilità ai clienti di partecipare al sondaggio e all'amministratore di fare una serie di operazioni per gestire questo concorso a premi e di poter aggiornare i dati, salvando tutto in Blockchain. A questa finalità, si svilupperà un client per l'interazione con la rete Ethereum che consisterà in un'interfaccia web minimale.

Per poter modificare lo stato del sistema sarà necessario possedere un wallet Ethereum. Con il wallet Ethereum si possono effettuare operazioni di lettura in Blockchain in quanto non hanno un costo. Ma per effettuare transazioni che modificano lo stato della Blockchain bisogna anche avere degli ether per poterle pagare. Come anticipato il costo della transazione dipenderà da quanto gas verrà usato e dal prezzo di mercato del gas in quel momento. Ovviamente l'utente può indicare un prezzo massimo di gas che è disposto a pagare per eseguire la transazione. Ritorniamo qui sul concetto che più si è disposti a pagare e prima la transazione verrà minata e messa in Blockchain.



Caso d'uso con le principali funzionalità del sistema

3.1 Requisiti funzionali

Di seguito i requisiti funzionali della mia Dapp:

- Possibilità di avere a disposizione un'interfaccia web user-friendly
- Possibilità di effettuare transazioni, firmandole con chiave privata
- Possibilità di visualizzare o cambiare l'owner di un contratto
- Possibilità di associare un account Ethereum all'owner del contratto (amministratore)
- Possibilità da parte del cliente di compilare il sondaggio inserendo le proprie generalità
- Possibilità dell'owner di inserire l'utente che ha partecipato al sondaggio nella Blockchain o controllare se è già presente
- Possibilità dell'owner e dei partecipanti di visualizzare il numero di partecipanti al sondaggio e trovare un partecipante in base all'ID
- Possibilità dell'owner di estrarre il vincitore del sondaggio

- Possibilità dell'owner di resettare i partecipanti e il vincitore del sondaggio
- Possibilità da parte di owner e partecipanti di visualizzare tutti i dettagli delle transazioni effettuate (ether consumati, gas speso, hash delle transazioni, indirizzi, eventi, blocchi etc)
- Possibilità da parte dei partecipanti del sondaggio di avere una totale trasparenza sui dati presenti sulla rete, di controllare e visualizzare tutte le varie transazioni prodotte etc.

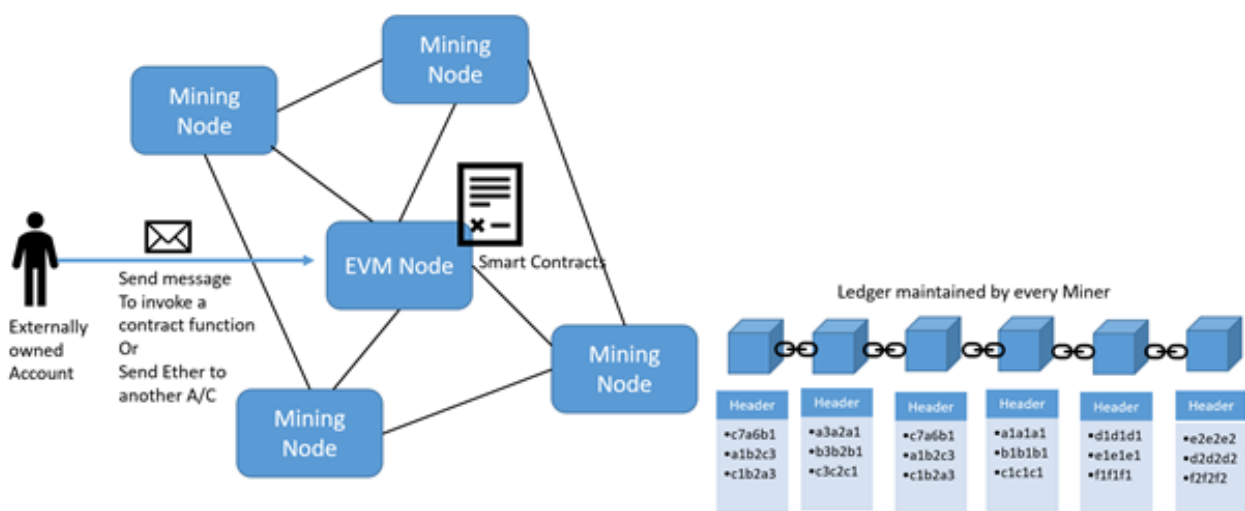
3.2 Requisiti non funzionali

- Utilizzare le best practice di Solidity per lo sviluppo degli Smart Contract.
- Non gravare sull'intensità computazionale delle operazioni principali per ridurre il Gas richiesto dalle funzioni Ethereum. In tal senso si è provveduto all'ottimizzazione del codice degli Smart Contract. Per esempio, durante lo sviluppo degli Smart Contract è stato prediletto, quando possibile, l'uso della chiave "external" per una funzione piuttosto che la chiave "public" perchè consuma meno gas. Invece con l'utilizzo della chiave "view" si sono potute implementare anche operazioni che non cambiano lo stato della Blockchain e quindi non consumano Ether.
- Utilizzare un approccio di sviluppo test-driven

4 Stato dell'arte

Alla luce degli aspetti sopra riportati, prima di introdurre concetti legati alla realizzazione del progetto, è necessario fare leva su alcuni aspetti teorici e tecnici che stanno alla base dell'applicazione e verranno poi ripresi nei capitoli riguardanti il Design e i Dettagli implementativi.

4.1 Architettura di Ethereum



Architettura di Ethereum

Una Blockchain è un'architettura formata da diverse componenti e quello che la rende unica è il modo in cui questi componenti interagiscono tra di loro.

Alcune componenti importanti in Ethereum sono la EVM (Ethereum Virtual Machine), i nodi di mining, i Blocchi, le transazioni, l'algoritmo di consenso, gli Smart Contract, gli account, gli Ether e il Gas.

Una rete di Blockchain consiste in tanti nodi che appartengono ai miners e in altri nodi che si occupano dell'esecuzione degli Smart Contract e delle transazioni. Questi nodi fanno parte della EVM. Ogni nodo della EVM è collegato a un altro nodo sulla rete. Questi nodi usano un protocollo peer to peer per parlare tra di loro e usano di default il numero di porta 30303.

Abbiamo quindi una rete strutturata in nodi chiamati EVM (Ethereum virtual machine), ossia macchine virtuali eseguite all'interno di ognuno dei server replica, i quali ci forniscono un ambiente isolato e protetto dal resto della rete host locale, in grado di interagire con le risorse della macchina fisica.

L'EVM è un centro di calcolo che ospita gli Smart Contract e permette la loro esecuzione. Questi Smart Contract vengono eseguiti come parte di una transazione e poi segue il processo di mining.

Una persona che ha un account sulla rete può inviare un messaggio per trasferire ether o per invocare una funzione del contratto. Ethereum considera entrambi delle transazioni.

La transazione deve essere firmata digitalmente con la chiave privata dell'account che ha inviato la richiesta. Questo viene fatto per verificare l'identità dell'utente mentre si verifica la transazione e si cambia il saldo degli account coinvolti nella transazione.

L'EVM può essere visto come l'ambiente di esecuzione per una rete Ethereum. Esso permette di eseguire il codice scritto all'interno degli Smart Contract e può accedere agli account (ai suoi dati archiviati, ai contratti, etc).

Quando viene invocata una funzione di uno Smart Contract e viene sottomessa una transazione, essa non viene eseguita immediatamente, ma viene incodata in un pool di transazioni. Poi quando queste transazioni verranno eseguite verranno inserite nei nodi della rete.

Le EVM avranno anche la possibilità di interfacciarsi con un database (uno per ogni EVM) nel quale è memorizzata una copia della Blockchain. Saranno anche responsabili dell'esecuzione del bytecode Ethereum (output della compilazione di uno Smart Contract).

Lavorando con gli Smart Contract, Ethereum ci permette di astrarre da problematiche quali l'implementazione di protocolli per lo scambio di messaggi tra i nodi all'interno della rete Blockchain, lo sviluppo di protocolli di consenso, di validazione delle transazioni e meccanismi per la fault tolerance (gestione di guasti). Tutte queste meccaniche saranno implementate all'interno di ogni EVM.

4.2 I nodi di mining

Un miner è responsabile della scrittura delle transazioni sulla catena Ethereum. Esso è interessato a scrivere transazioni nel registro per ottenere la ricompensa ad esso associata.

I miners ottengono una ricompensa per la scrittura di un blocco nella catena e anche delle commissioni cumulative sul gas da tutte le transazioni nel blocco.

Ci sono generalmente molti miners disponibili all'interno di una rete Blockchain, ognuno dei quali cerca di competere con gli altri per scrivere transazioni. Tuttavia, solo un miner può scrivere il blocco nel Libro Mastro. La scelta del miner che scriverà su un blocco avviene tramite una sfida.

La sfida è data in ogni nodo e ogni miner cerca di risolvere il puzzle usando la sua potenza di calcolo. Il miner che risolve il puzzle scrive per primo il blocco contenente le transazioni nel Libro Mastro e riceve anche 5 ether come ricompensa.

Ogni nodo di mining mantiene la propria istanza del Libro Mastro di Ethereum e il Libro Mastro è in definitiva lo stesso per tutti i miners. È compito dei miners assicurarsi che il loro registro sia aggiornato con i blocchi più recenti.

Ogni miner svolge tre importanti funzioni:

- Minare o creare un nuovo blocco con transazione e scriverlo sul libro mastro di Ethereum
- Avvisare che ha appena estratto un blocco e inviarlo agli altri miners.
- Accettare nuovi blocchi estratti da altri miners e mantenere aggiornata la propria istanza di registro.

I nodi di Mining si riferiscono ai nodi che appartengono ai Miners. Questi nodi fanno parte della stessa rete in cui è ospitato EVM. Ad un certo punto i miners creano un nuovo blocco, raccolgono tutte le transazioni dal pool di transazioni e le aggiungono al blocco appena creato. Infine, questo blocco, viene aggiunto alla catena.

4.2.1 Come funziona il mining

Il processo di mining è applicabile a tutti i miners sulla rete e ogni miner continua a eseguire regolarmente le attività qui spiegate.

I miners non vedono l'ora di estrarre nuovi blocchi e stanno attivamente in ascolto per ricevere nuovi blocchi da altri miners. A un certo punto, il miner raccoglie tutte le transazioni dal pool di transazioni. Questa attività viene svolta da tutti i miners.

Il miner costruisce un nuovo blocco e vi aggiunge tutte le transazioni. Prima di aggiungere queste transazioni, controllerà se una qualsiasi delle transazioni non è già scritta in un blocco che potrebbe ricevere da altri miners. In tal caso, eliminerà tali transazioni. Il miner aggiungerà la propria transazione al fine di ottenere la ricompensa per l'estrazione del blocco.

L'attività successiva del miner è generare l'intestazione del blocco e in seguito:

- Il miner esegue l'hash di tutte le transazioni nel blocco, questi hash vengono ulteriormente combinati a coppie per generare un nuovo hash. Il processo continua fino a quando non c'è un solo hash per tutte le transazioni nel blocco. L'hash è indicato come hash della 'transaction root'. Questo hash viene aggiunto all'intestazione del blocco.
- Il miner identifica anche l'hash del blocco precedente. Il blocco precedente diventerà padre del blocco corrente e anche il suo hash verrà aggiunto all'intestazione del blocco.
- Il miner calcola in modo simile gli hash degli stati e delle ricevute della 'transaction root' e li aggiunge all'intestazione del blocco.
- All'intestazione del blocco vengono aggiunti anche un nonce e un timestamp.

Il processo di mining inizia nel momento in cui il miner continua a modificare il valore del nonce e cerca di trovare un hash soddisfacente come risposta al puzzle dato. È da tenere presente che tutto questo viene eseguito da ogni miner nella rete. Alla fine uno dei miner sarà in grado di risolvere il puzzle e avvisare gli altri miner della rete. Questi verificheranno la risposta e, se sarà corretta, controlleranno ulteriormente ogni transazione accettando il blocco e aggiungendolo alla loro istanza di registro. L'intero processo è anche noto come Proof of Work in cui un miner fornisce la prova che ha lavorato sul calcolo della risposta finale che potrebbe essere la soluzione del puzzle. Esistono anche altri algoritmi come Proof of Stake e Proof of authority.

4.3 Gli Account

Gli account sono il principale elemento di Ethereum. E' l'interazione tra gli account che Ethereum vuole memorizzare sul suo Libro Mastro. Ethereum supporta due tipi di account. Ogni account(o conto) ha un saldo che restituisce il valore corrente in esso memorizzato.

4.3.1 Gli Account di proprietà esterna

Gli account di proprietà esterna su Ethereum sono account che appartengono a persone reali. Quando un account di proprietà esterna viene creato da una persona, viene generata una chiave pubblica-privata. La chiave privata viene tenuta al sicuro dall'individuo mentre la chiave pubblica diventa l'identità di questo account. Questa chiave pubblica è generalmente di 256 caratteri, tuttavia Ethereum utilizza i primi 160 caratteri per rappresentare l'identità di un account. Se Bob crea un account sulla rete Ethereum avrà la sua chiave privata a sua disposizione mentre i suoi primi 160 caratteri di chiave pubblica diventeranno la sua identità. Altri account sulla rete possono quindi inviare ether a questo account.

Un esempio di Account Ethereum viene mostrato qui sotto.

0xa57de277ede9c1521f51f6989ed2497a5b9c1926

Un account di proprietà esterna può contenere Ether nel proprio saldo e non ha nessun codice associato. Può eseguire transazioni con altri account di proprietà esterna e può anche eseguire transazioni invocando funzioni all'interno dei contratti.

4.3.2 Contract Accounts

I Contract Accounts sono molto simili agli account di proprietà esterna. Vengono identificati utilizzando il loro indirizzo pubblico. Non hanno alcuna chiave privata. Possono contenere ether in modo simile agli account di proprietà esterna, tuttavia contengono codice: il codice per gli Smart Contract costituito da funzioni che possono modificare lo stato.

4.4 Transazioni

Come abbiamo anticipato, per poter effettuare modifiche allo stato dell'applicazione decentralizzata è necessario eseguire transazioni.

Una transazione può essere eseguita da un account registrato all'interno del sistema di Blockchain distribuito.

Definiamo ora le componenti principali di una transazione Ethereum.

Per poter effettuare una transazione è necessario stabilire l'account che la andrà ad eseguire, tramite un indirizzo Ethereum pubblico (from), ed un indirizzo "to", che sarà il destinatario della transazione. Il destinatario di una transazione, nel nostro caso sarà lo Smart Contract che abbiamo implementato e "deployato" dentro la rete Blockchain, che in particolare sarà identificato da un indirizzo Ethereum pubblico.

Oltre a "from" e "to" una transazione ha altri parametri.

Il parametro "value" si riferisce alla quantità di ether che è stata trasferita da un account a un altro.

Il parametro "input" si riferisce al bytecode del contratto compilato ed è usato durante il deploy del contratto in EVM. Esso serve anche per memorizzare i dati relativi alla funzione dello Smart Contract chiamata con i suoi parametri.

Il parametro "nonce" rappresenta il numero di transazioni prodotte dall'account che sta effettuando la transazione. Transazioni con nonce non coerenti verranno respinte, ad esempio infatti, non sarà possibile scrivere transazioni con nonce 1, prima che la transazione con nonce 0 sia stata inviata e validata.

Il parametro "hash" indica l'hash della transazione.

All'interno di una transazione sono specificati anche dettagli relativi al Gas. Il prezzo da pagare da un'utente per scrivere una transazione è dato da $ETH = GasLimit * GasPrice$, che di fatto è l'importo che verrà assegnato al miner che andrà a validare la transazione e non al provider del servizio.

Un altro parametro è GasPrice ed è utilizzato per definire un livello di priorità per la validazione di una transazione. Infatti, più è alto, e prima la transazione verrà validata ed applicata all'interno della Blockchain. Quindi, nel caso in cui un utente abbia necessità di scrivere una transazione il prima possibile, dovrà pagare un prezzo maggiorato.

Con il Gas Limit invece si può definire il costo in gas per una determinata operazione.

All'interno della transazione abbiamo infine la descrizione dell'operazione che viene effettuata. Nel nostro caso, essa rappresenterà una chiamata ad una funzione dello Smart Contract, con eventuali argomenti. Nei logs abbiamo tutte le informazioni sull'operazione effettuata (i logs sono dati dagli eventi scritti nei contratti che stampavano le informazioni delle operazioni effettuate).

Una volta generata la transazione però sarà necessario applicargli la firma digitale per renderla identificabile e di conseguenza possibile da validare. Come già sappiamo, ogni account Ethereum possiede un identificatore pubblico ed una chiave privata, i quali ci permetteranno

Altri parametri importanti di una transazione sono "BlockNumber" (il blocco a cui appartiene la transazione), e V, R e S che sono relativi alla firma digitale e alla firma delle transazioni.

Esempio di transazione dopo l'aggiunta di un utente

<https://ropsten.etherscan.io/tx/0x048dbbabaf54af8b45ea787e1fdb62d10015a3655565c099d71b7a9e72027145>

I blocchi sono contenitori di transazioni. Un blocco contiene più transazioni. Ogni blocco ha diversi numeri di transazioni basate sul Gas Limit.

Tra i principali parametri del blocco si ricorda la difficulty. Essa è la complessità della challenge data ai miners per questo blocco.

16

[This is a Ropsten Testnet block only]	
⑦ Block Height:	9780871 < >
⑦ Timestamp:	⌚ 3 hrs 40 mins ago (Mar-05-2021 02:07:09 PM +UTC)
⑦ Transactions:	28 transactions and 31 contract internal transactions in this block
⑦ Mined by:	0x2f93b2f047e05cdf02820ac4b3178efc2b43d55 in 20 secs
⑦ Block Reward:	2.079588265882141 Ether (2 + 0.017088265882141 + 0.0625)
⑦ Uncles Reward:	1.5 Ether (1 uncle at Position 0)
⑦ Difficulty:	420,507,431
⑦ Total Difficulty:	32,946,571,492,736,252
⑦ Size:	133,962 bytes
⑦ Gas Used:	6,616,975 (82.71%)
⑦ Gas Limit:	8,000,029
⑦ Extra Data:	030100/OpenEthereum/1.47.0/li (Hex:0xdb830301008c4f70856e457468657265756d86312e34372e30826c69)

Informazioni sul blocco

Si può visualizzare il blocco in figura anche qui: <https://ropsten.etherscan.io/block/9780871>

4.6 Una transazione end to end

Avendo visto i concetti fondamentali, si vede ora una transazione end-to-end completa per capire come si sussegue attraverso più componenti e come viene archiviata nel Libro Mastro. In questo esempio, Sam vuole inviare pochi Ether a Mark. Sam genera un messaggio di transazione come mostrato prima contenente i campi 'from', 'to', 'value' e lo invia alla rete Ethereum.

La transazione non viene scritta immediatamente nel Libro Mastro e viene invece inserita in un pool di transazioni.

Il nodo di mining crea un nuovo blocco e prende tutte le transazioni dal pool rispettando i criteri del Gas Limit e le aggiunge al blocco. Questa attività viene svolta da tutti i miners della rete. Anche la transazione di Sam farà parte di questo processo.

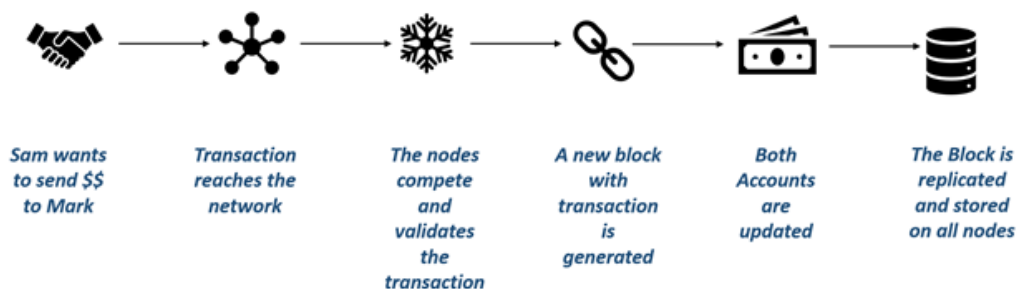
I miners competono cercando di risolvere la sfida loro lanciata. Il vincitore è un miner che può risolvere per primo la sfida.

Dopo un periodo (10 secondi in Ethereum) uno dei miner annuncerà di aver trovato la soluzione alla sfida e che è il vincitore e dovrebbe quindi scrivere il blocco sulla catena.

Il vincitore invia la soluzione della sfida insieme al nuovo blocco a tutti gli altri miners. Il resto dei miners convalida e verifica la soluzione e, una volta soddisfatti della soluzione, accettano il nuovo blocco contenente la transazione di Sam da aggiungere alla loro istanza di Libro Mastro. Questo genera un nuovo blocco sulla catena che viene mantenuto nel tempo e nello spazio.

Intanto i conti di entrambe le parti vengono aggiornati con un nuovo saldo. Infine, il blocco viene replicato su ogni nodo della rete.

Lo stesso processo è mostrato anche nell'immagine successiva.



4.7 Ciclo di vita di uno Smart Contract in Ethereum

Ethereum mette a disposizione un linguaggio di Programmazione turing completo (Solidity). Gli Smart Contract, sviluppati con linguaggi di alto livello, prima di essere eseguiti, devono essere compilati producendo il bytecode che viene poi interpretato dalla EVM.

Una volta che il contratto è compilato viene fatto il deploy e poi viene messo in esecuzione all'interno delle EVM.

Sarà necessario un account con alcuni ether presenti ed un nodo connesso alla Blockchain per poter effettuare la transazione di migrazione, che permetterà di memorizzare il contratto all'interno dei nodi della rete. Una transazione di migrazione consiste di fatto in una normale transazione Ethereum, effettuata dall'account che ha sviluppato lo Smart Contract, che andrà a produrre un nuovo indirizzo pubblico, ossia quello del contratto all'interno delle EVM.

Il contratto compilato verrà poi allocato ad un indirizzo virtuale univoco all'interno delle EVM, il quale dovrà essere utilizzato come indirizzo target per effettuare transazioni dagli account.

Una volta che il contratto sarà "deployato" all'interno della macchina virtuale Ethereum, si potranno richiamare le sue funzioni.

Ricapitolando, quando si compila un contratto, il compilatore genera due artefatti: l'ABI e il bytecode. Bisogna pensare all'ABI come un'interfaccia composta da tutte le dichiarazioni di funzioni pubbliche e esterne insieme ai loro parametri e ai tipi di ritorno. L'ABI definisce il contratto e qualsiasi chiamante che desidera richiamare qualsiasi funzione di contratto può utilizzare l'ABI per farlo.

Il bytecode è quello che rappresenta il contratto ed è deployato nell'environment Ethereum.

Quindi il bytecode è richiesto durante il deploy mentre l'ABI è richiesto durante l'invocazione di funzioni.

Viene creata una nuova transazione che utilizza la nuova istanza di contratto passandola nel bytecode del contratto e l'appropriata quantità di gas per l'esecuzione del contratto. Dopo che la transazione viene minata, il contratto è disponibile all'indirizzo determinato da Ethereum.

4.8 Recap di ciò che ho utilizzato

In questa sezione si fa un recap dei vari tool e piattaforme utilizzate per la mia Dapp.

- **Remix**: Si tratta di un IDE web che integra una macchina virtuale Blockchain di test per il deploy degli Smart Contract e l'esecuzione di transazioni. Il linguaggio di programmazione usato per scrivere gli Smart Contract è Solidity.
- **Truffle**: Fornisce un set di strumenti per aiutare gli sviluppatori nello sviluppo e deploy di Smart Contract.
- **Ganache**: Si tratta di un applicativo del framework Truffle che permette di avere una Blockchain Ethereum locale. Il tool permette anche di generare un numero limitato di account con portafoglio non vuoto. In questo modo si può effettuare il testing verificando la correttezza dell'interazione (tra la Blockchain e l'applicativo NodeJs) per quelle operazioni che richiedono di effettuare transazioni, dove necessariamente vengono spesi degli Ether.

- Ropsten: Si tratta di una Blockchain pubblica creata da Ethereum per gli sviluppatori che permette di testare la Dapp su una vera Blockchain pubblica Ethereum ma utilizzando ether di test. Ropsten ha il suo Explorer e la sua rete è pubblica esattamente come Ethereum quindi tutti possono andare su questo Explorer e vedere le transazioni. Il vantaggio di usare Ropsten sta anche nella possibilità di usare un sito internet pubblico per verificare tutte le transazioni e gli aspetti ad esse legate (quello che in gergo tecnico si chiama etherscan per Ethereum). Si trova a questo indirizzo: <https://ropsten.etherscan.io/>.
- Web3.js: è una libreria in Javascript da includere nelle nostre DApps, permettendoci di interagire con tutto il sistema Ethereum, in particolar modo con la Blockchain e gli Smart Contract.
Javascript e soprattutto NodeJS in questi anni ci hanno portato nel magico mondo dell'asincrono. Web3.js ci porta, a partire dalla versione 1.0.0, ad usare promises e callback, in quanto l'esecuzione su una rete come Blockchain è l'opposto di una esecuzione sincrona. Web3.js offrirà nella versione 1.0.0 la possibilità di effettuare dei subscribe a molti tipi di eventi: transazioni in fasi di pending, syncing del nodo, nuovi blocchi e molto altro. Web3 ci permette una gestione completa dei nostri Smart Contract. Possiamo infatti fare il deploy di uno Smart Contract senza scomodare tools come Truffle: ci basterà fornire alla funzione `web3.eth.contract()` l'ABI del nostro Smart Contract, ottenendo un oggetto di cui chiamare la funzione `new`. Questa funzione, che si occuperà di eseguire la transazione richiede un oggetto con le seguenti proprietà:

- `from`: l'account da utilizzare per la transazione
- `gasPrice`: prezzo del gas in WEI
- `gas`: il limite di gas fornito alla rete per la transazione
- `data`: bytecode dello Smart Contract

Per interagire con uno Smart Contract abbiamo bisogno dell' ABI (Abstract Binary Interface) e dell'indirizzo su cui risiede lo smart contract.

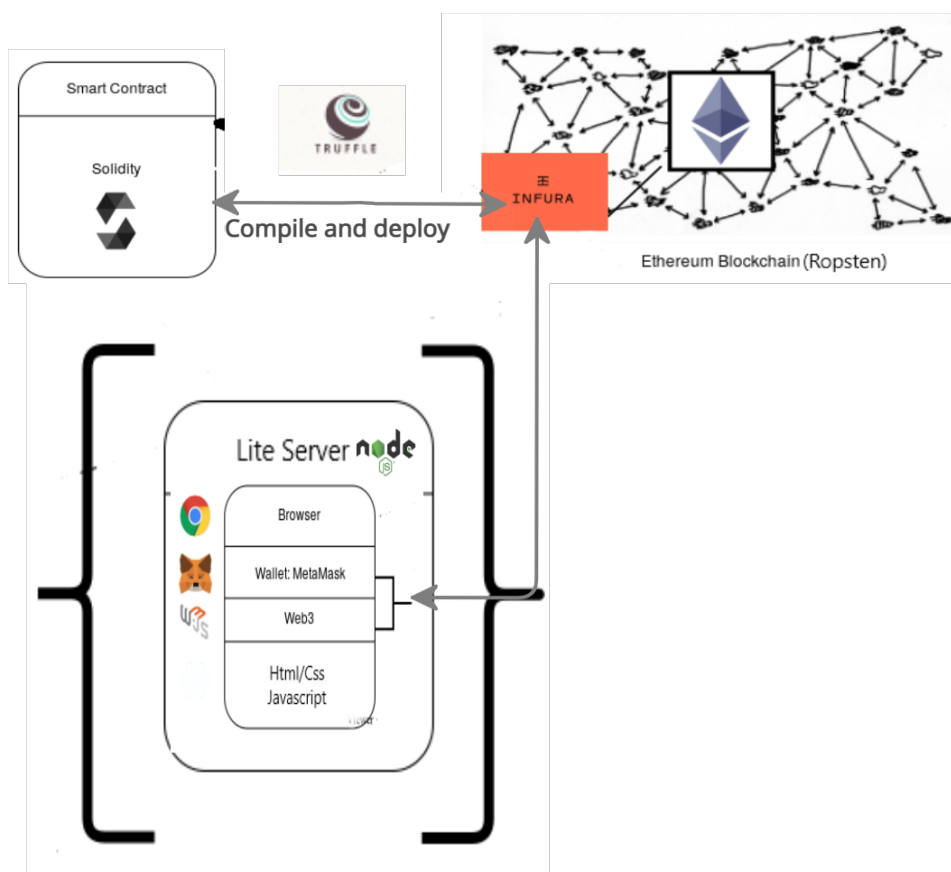
- Metamask: Metamask funziona grazie all'utilizzo di web3.js. E' stato creato per essere un portafoglio per Ethereum e uno strumento per interagire con le DApp. Per ottenere entrambi, MetaMask stabilisce un canale di comunicazione tra l'estensione e la DApp in questione. Una volta che l'applicazione riconosce che MetaMask sia presente, viene abilitato e può essere utilizzato dall'utente.
Una volta abilitata la DApp, l'utente può eseguire ciascuna delle azioni o degli eventi consentiti, dall'acquisto o dalla vendita di token, all'accesso alle risorse o a qualsiasi servizio fornito. Ognuna di queste azioni ha un costo, che deve essere pagato in Ethereum o nel token indicato per esso. In entrambi i casi, MetaMask possiede gli strumenti necessari per gestire tale interazione.
Vale a dire, MetaMask non solo genera un portafoglio di criptovaluta, ma controlla anche ogni interazione dell'utente con la DApp ed esegue le operazioni necessarie affinché tali operazioni vengano eseguite. Tutto questo avviene in un mezzo di comunicazione sicuro e utilizzando una forte crittografia. MetaMask ha la capacità di generare le proprie chiavi asimmetriche, salvarle localmente e gestirne l'accesso. Grazie a questo, MetaMask è un'estensione altamente sicura.
- Infura: E' un sistema che permette alle dApp di processare informazioni sulla blockchain di Ethereum senza avere un full node installato sul proprio device e quindi permette un accesso veloce alla Blockchain da un nodo centralizzato. Supporta JSON-RPC su interfacce HTTPS e WebSockets, fornisce request and subscription-based connection. Esso è un servizio che da un lato ci avvicina ad un contesto centralizzato (essendo il nodo di Infura quello da cui partono tutte le transazioni pagate con il nostro wallet) dall'altro

ci permette di interfacciarci alla Blockchain pubblica senza scaricarci una copia locale di essa.

5 Design

Come definito negli obiettivi lo scopo di questo progetto non è stata l'implementazione di una rete Blockchain, ma è stata sfruttata la struttura fornita da Ethereum, e si sono ereditate tutte le astrazioni e logiche di interazione fornite da essa. Si analizzeranno più nel dettaglio le logiche della mia Dapp.

5.1 Struttura



Struttura della mia Dapp

La struttura di questa Dapp viene riassunta nella figura sopra.

Come già detto questa Dapp si appoggia a una tecnologia decentralizzata, quale è la piattaforma Ethereum. Gli Smart Contract vengono deployati avvalendosi di Infura, per effettuare le transazioni utilizziamo MetaMask e il suo wallet. MetaMask a sua volta, in modo trasparente per l'utente, usa Infura.

Il deploy dell'applicazione viene fatto sulla rete testnet Ropsten. Grazie a Ropsten possiamo utilizzare monete (ether) di sviluppo (prelevati da Metamask).

Si potrebbe benissimo migrare da Ropsten a Ethereum. L'unica differenza è che dovrei usare soldi reali (ether) che hanno un costo.

Il front-end, ovvero la parte di applicazione parte utente, è composta da un web browser, da un wallet Metamask, da web3, da Javascript e da una pagina web minimale. Il client è composto da un interfaccia web minimale. L'interazione sarà basata sull'utilizzo della libreria web3JS, il

cui funzionamento è basato sul protocollo sincrono JSON-RPC. Essa dà al client la possibilità di accedere ai contratti "deployati" e alle sue funzioni, e di poter così effettuare transazioni, autenticandole grazie al meccanismo crittografico basato su firma digitale. Il codice che scrive la parte del software che verrà eseguito in FrontEnd sarà scritto in Javascript. Lite server è un server molto leggero che ci serve per rendere le pagine dinamiche ed effettuare le chiamate HTTP.

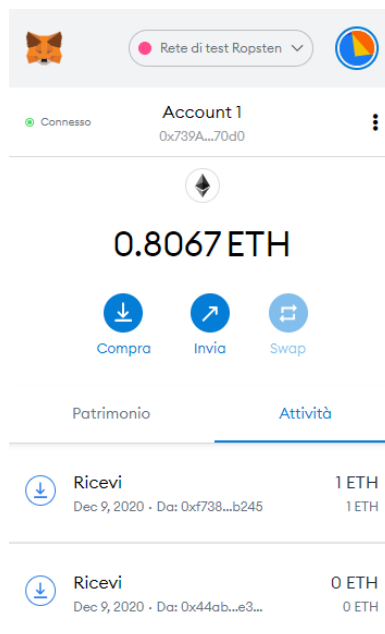
Si userà Truffle per la compilazione e il deploy dei contratti.

Come wallet si utilizza Metamask installato come pug-in in Google Chrome. Il mio progetto quindi funziona solo utilizzando il browser Chrome.

Il wallet di MetaMask consentirà agli utenti di archiviare i propri account Ethereum, comprese le chiavi private, in modo sicuro all'interno del browser.

Le transazioni, prima di essere inviate, vengono firmate con la chiave privata del wallet lato client. In questo modo Metamask invia la transazione su Ropsten senza chiedere all'utente nessuna conferma. Se non si firmassero le transazioni, ad ogni richiesta, dopo aver fatto il Login al proprio account MetaMask, si dovrebbe approvare manualmente ogni transazione. Per una demo più usabile si è pensato di procedere come descritto.

Solo perchè il progetto è accademico si è deciso di mettere la chiave privata in chiaro all'interno del progetto. In uno scenario reale si dovrebbe gestire in modo sicuro il meccanismo di cifratura e decifratura delle chiavi private, come anche di altri dati sensibili, tra cui le generalità di un utente. Ma ciò esula dagli obiettivi di questo progetto. Si veda la sezione 11.1.



Wallet di Metamask per la mia Dapp

5.1.1 Dominio applicativo

All'interno degli Smart Contract l'entità core che viene modellata è Partecipante. Le informazioni che lo definiscono sono il codice fiscale (l'id), nome, cognome, email. Un Partecipante può diventare il Vincitore del sondaggio. Ogni contratto deployato, come d'altronde ogni transazione, ha un hash che lo identifica univocamente.

Di seguito i due hash rispettivamente per dello Smart Contract Partecipanti e Vincitore deployati su Ropsten ed accessibili a tutti:

PartecipazioneSondaggioBFB.sol: "0x54850326CE45516A79fa82C6dCF7aB25B2246C63"

VincitoreSondaggioBFB.sol: "0x0F8190810667903A36B04c325A3b79c7B651f1BE"

Il contratto Owner.sol serve a definire le operazioni riservate all'amministratore. L'amministratore è l'utente che ha deployato gli Smart Contract la prima volta, a meno di modifiche successive.

Solo dall'account dell'amministratore è possibile aggiungere utenti in Blockchain, estrarre il Vincitore oltre ad altre operazioni di amministrazione quali: reset partecipanti al sondaggio e reset vincitore del contest.

5.2 Client Node

Il core del client node si trova nella cartella "src" del progetto.

In particolare App.js gestisce tutte le operazioni di interazione con la Blockchain. Queste operazioni sono asincrone. In questo modo si possono gestire le operazioni richieste dall'utente, eseguendo il task associato all'operazione richiesta.

Per richiamare le istanze di contratto serve l'interfaccia json del rispettivo contratto (l'abi del contratto). I json dei contratti sono contenuti anch'essi nella cartella src.

5.3 Comportamento

Grazie all'utilizzo degli Smart Contract è possibile lavorare ad un'astrazione tale da stabilire il comportamento funzionale dei nodi della rete Ethereum, definendo ed implementando servizi che sono erogabili dal sistema tramite la definizione di funzioni.

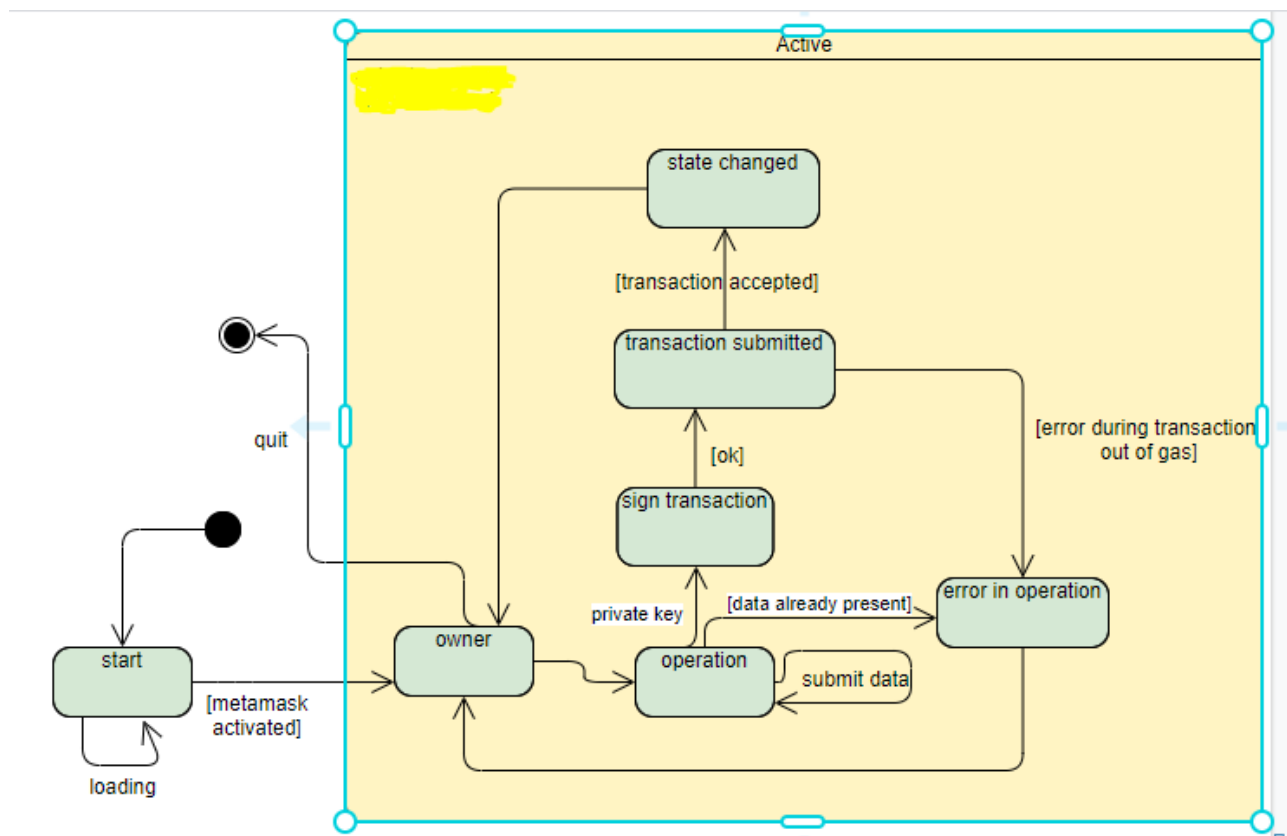
E' importante capire tuttavia il livello di astrazione sul quale si sta lavorando.

Utilizzando una piattaforma Blockchain quale Ethereum, possiamo lavorare senza preoccuparci della reale complessità della rete Blockchain sulla quale vengono eseguiti i nostri Smart Contract. Il vero comportamento di un nodo di rete quando vengono ricevuti i messaggi ci viene nascosto.

Passiamo ora a definire il comportamento dell'interfaccia web sviluppata nella mia Dapp.

Qui sotto è riportata un diagramma di stato che riassume il comportamento del client sviluppato per l'interazione con la rete Blockchain. Esso si basa sulla continua ricezione di comandi da standard input, per poi tradurli in operazioni di interfacciamento con la rete Blockchain.

L'utilizzo di uno meccanismo asincrono ci permette gestire in maniera piuttosto semplice la ricezione di comandi e la gestione delle numerose operazioni.



Inciso: tra gli errori durante la transazione oltre all'out of gas e ai dati già presenti c'è anche il revert in caso di input errati o mancanti, etc. Dei casi si revert se ne è comunque parlato in questa relazione.

5.4 Interazione

Nella figura della sezione 5.1 possiamo già vedere come avviene l'interazione delle varie parti della mia Dapp.

L'interazione con la rete Blockchain avverrà mediante l'utilizzo della libreria web3JS.

La comunicazione sarà sincrona e basata sull'utilizzo del protocollo JSON-RPC, che ci darà la possibilità di interagire con la rete Ethereum.

Un client NodeJS ci darà la possibilità di accedere allo stato dei contratti, e ci consentirà di effettuare chiamate alle funzioni definite in essi.

Lavorando ad un alto livello di astrazione, non saremo noi a definire il modello di interazione tra le EVM, ma utilizzeremo la struttura implementata da Ethereum, spiegata nel capitolo 4.

6 Dettagli implementativi

In questo parafraso riporto qualche dettaglio implementativo della Dapp creata.

Solidity inoltre non fornisce tool per generare documentazione, quindi darò una breve descrizione implementativa sui contratti sviluppati.

Ricordiamo che il codice di tali contratti è comunque opportunamente commentato.

Si daranno dettagli anche sul funzionamento del client Node.

6.1 Gli Smart Contract in Solidity

Tutti i contratti della mia Dapp si trovano nella cartella "contracts" del progetto.

Entrambi i contratti, sia quello relativo ai partecipanti del sondaggio, sia quello relativo al vincitore, estendono un contratto denominato Owner.sol.

Owner è un contratto già messo a disposizione da Remix, definisce le operazioni di un Owner. Owner corrisponde al proprietario, ovvero all'indirizzo di chi ha deployato il contratto. Nel nostro caso corrisponde con il proprietario del negozio che gestisce il sondaggio.

Come possiamo vedere in figura sotto, nel contratto PartecipazioneContrattoBFB.sol dichiaro una variabile di stato owner (l'address del proprietario), una variabile che definisce il numero di utenti, una variabile mapping che mappa tutti gli utenti che partecipano al contest in (Id/Utente). Nel costruttore l'owner viene settato come sender (proprietario). L'utente è rappresentato come una struct.

```
pragma solidity ^0.7.0;
import "./Owner.sol";

@title il contratto per la partecipazione al sondaggio
contract PartecipazioneSondaggioBFB is Owner {
    address private owner;

    uint private userCount = 0;

    mapping(uint => User) private users;

    constructor() {
        owner = msg.sender;
    }

    struct User {
        uint id;
        string cdf;
        string nome;
        string cognome;
        string email;
    }
}
```

L'operazione principale di questo contratto che riporto è senz'altro l'aggiunta di un utente.

```
/*@dev Aggiunta utente alla Blockchain
* @param _cdf univoco dell'utente(lo smart contract verifica la sua univocita')
* @param _nome nome utente
* @param _cognome cognome utente
* @param _email email utente
*/
function addUser( string memory _cdf, string memory _nome, string memory _cognome, string
memory _email) external isOwner() {
    require(bytes(_cdf).length > 0 , "Il parametro codice fiscale e' vuoto");
    require(bytes(_nome).length > 0 , "Il parametro Nome vuoto");
    require(bytes(_cognome).length > 0 , "Il parametro Cognome vuoto");
    require(bytes(_email).length > 0 , "Il parametro Email e' vuoto");

    if(checkCDFExisting(_cdf)==0) //se il codice fiscale non esiste allora inserisco l'
        utente
    {
        userCount ++;
        users[userCount] = User(userCount, _cdf, _nome, _cognome, _email);
        emit UserAdded(userCount, _cdf, _nome, _cognome, _email);
    }
    else //se il cdf esiste gia' l'utente ha gia' partecipato al sondaggio quindi non puo'
        piu' partecipare
    {
        emit Error("L' utente ha gia' partecipato al Sondaggio ");
        revert("L'utente ha gia' partecipato al Sondaggio");
    }
}
```

Dovendo rendere tutte le funzioni del contratto attivabili unicamente da account Ethereum, si è deciso di utilizzare il modificatore modifier fornito dal linguaggio Solidity per poter definire dei criteri per l'accesso alle funzionalità del contratto.

Una funzione Solidity, una volta chiamata, entrerà in esecuzione solo dopo avere eseguito le proprietà modifier alle quali aderisce. Nel caso una clausola modifier contenga clausole require, essa sarà eseguita solamente se quest'ultima andrà a buon fine.

Utilizzare clausole require all'interno delle funzioni modifier è un comune pattern Solidity utilizzato per definire restrizioni per l'accesso, evitando utenti senza permessi.

Come possiamo vedere nella funzione addUser, l'utilizzo del modifier isOwner permetterà che questa funzione sia solo invocata dall'owner del contratto (quindi dall'amministratore).

Inoltre, affinché l'inserimento dell'utente vada a buon fine, si richiedono dei parametri validi per codice fiscale, etc. Inoltre, con il controllo sul codice fiscale, si può controllare se un utente è già stato inserito, evitando inserimenti doppi.

addUser è una funzione che se invocata cambia lo stato di una Blockchain, ed è invocabile solo dall'esterno (infatti si utilizza la key "external"). Se l'operazione non va a buon fine(un campo non è corretto o un codice fiscale è già stato inserito) viene emesso un evento errore e innesca un'eccezione con revert (esso interrompe l'esecuzione e ripristina i cambiamenti di stato, fornendo una stringa esplicativa).

Nella scrittura delle funzioni ho posto molta attenzione nell'assegnare la giusta visibilità, prediligendo le key "external" o "private" alla key "public" quando possibile. Questo per ridurre il consumo di gas durante l'invocazione della funzione.

Funzioni "external" sono invocabili esternamente da tutti, funzioni "private" sono funzioni private che non possono essere richiamate da altri contratti e sono invocabili solo dall'owner, funzioni "public" sono invocabili da chiunque, ovvero dall'owner ma anche dagli utenti del sondaggio. Tra quest'ultime ricordiamo la funzione che permette di visualizzare il numero di utenti o di trovare un utente dato il suo ID.

Molto importante è anche "view", un modificatore da inserire nelle funzioni di visualizzazione che non altera lo stato della Blockchain, in modo tale che all'invocazione della funzione non ci sia consumo di Gas. Si è applicata la chiave view ad esempio nelle funzioni che mi permettono la visualizzazione dei partecipanti, la visualizzazione di un partecipante dato il suo ID, il controllo se un partecipante è già presente.


```
//----- EVENTI

event Error(
    string error
);

event UserAdded(
    uint id,
    string cdf,
    string nome,
    string cognome,
    string email
);
```

In Solidity molto importanti sono gli eventi.

Da uno Smart Contract si può emettere un evento ogni volta che c'è un'invocazione di funzione e una persona esterna possa venire informata in real time.

Gli eventi non possono essere letti da uno Smart Contract, essi vengono solo emessi e verranno letti dal di fuori.

Quando un evento viene emesso salva gli argomenti passati in dei logs di transazione. Questi logs vengono salvati nella Blockchain e sono accessibili usando l'indirizzo del contratto.

Nella figura del paragrafo 4.2 possiamo vedere che in etherscan oltre a visualizzare le informazioni di una transazione, possiamo visualizzare anche i logs cliccando su Logs(1). Nel nostro esempio tale log contiene le generalità dell'utente inserito nella Blockchain.

Un altro contratto importante è VincitoreSondaggioBFB.sol. Tale contratto andrà ad interagire con l'altro, quindi è molto importante creare una referenziazione tra i due contratti, come possiamo vedere in figura.

```
pragma solidity ^0.7.0;
import "./Owner.sol";
import "./PartecipazioneSondaggioBFB.sol";

contract VincitoreSondaggioBFB is Owner {

    PartecipazioneSondaggioBFB partecipazioneSondaggioBFBContract;

    address private owner;
    uint private userCountVincitore = 0;

    Vincitore private vincitore;

    struct Vincitore {
        string cdf;
        string nome;
        string cognome;
        string email;
    }

    constructor (PartecipazioneSondaggioBFB _partecipazioneSondaggioBFBContract) {
        //creao una referenziazione/dipendenza forte tra i due smart contract
        partecipazioneSondaggioBFBContract = PartecipazioneSondaggioBFB(
            _partecipazioneSondaggioBFBContract);
        owner = msg.sender;
    }
}
```

Molto importante è la funzione estraiVincitore. E' possibile estrarre un vincitore solo se ci sono già dei partecipanti al sondaggio. Viene generato un numero ID random e viene estratto l'utente a cui quell>ID appartiene.

```
function estrazioneVincitore () external isOwner() returns (string memory)
{

    uint contUtentiSondaggio= partecipazioneSondaggioBFBContract.getUserCount();
```

```

uint indiceVincitore;

if (contUtentiSondaggio<1)
{
    revert("Non ci sono abbastanza utenti per estrarre un vincitore");
}
if(contUtentiSondaggio==1)
{
    indiceVincitore=1;
}
else
{
    indiceVincitore= random(contUtentiSondaggio); //genero un numero random compreso
    tra 1 e il numeri di utenti che ha partecipato al sondaggio
}

(string memory cdfVincitore,string memory nomeVincitore,string memory cognomeVincitore
,string memory emailVincitore)= partecipazioneSondaggioBFBContract.getUserByID(
    indiceVincitore);

vincitore= Vincitore(cdfVincitore, nomeVincitore, cognomeVincitore, emailVincitore);
userCountVincitore++;
emit Winner ("Il Vincitore", cdfVincitore, nomeVincitore, cognomeVincitore,
    emailVincitore );

return(cdfVincitore);
}

```

Tutti potranno visualizzare il vincitore.

```

//----- FUNZIONI EXTERNAL INVOCABILI DA TUTTI

//funzione che ritorna un vincitore gia' estratto in precedenza
function getVincitore () external returns (string memory, string memory,string memory,
    string memory )
{
    require(userCountVincitore>0, "Non e' stato sorteggiato nessun vincitore");
    emit Winner ("Il Vincitore", vincitore.cdf, vincitore.nome, vincitore.cognome,
        vincitore.email );
    return (vincitore.cdf, vincitore.nome, vincitore.cognome, vincitore.email);
}

```

Tutti i miei Smart Contract su Remix sono presenti a questo indirizzo: <https://remix.ethereum.org/#version=soljson-v0.7.0+commit.6c089d02.js&optimize=false&gist=c7ed67e8f0979df1f940af84ddea8d24&runs=200&evmVersion=null>

6.2 Client NodeJs

Questa sezione darà dettagli implementativi sulla tecnologia client sviluppata. E' pensata, in particolare, per un'eventuale sviluppatore che dovrà effettuare modifiche evolutive o adattive. Si tralascia la parte relativa allo sviluppo grafico dell'interfaccia web minimale, di non interesse per questo corso.

L'obiettivo è quello di mostrare come sono state applicate le nozioni teoriche introdotte precedentemente mediante la libreria web3JS per interfacciarsi alla Blockchain. App.js (nella cartella src) è il file javascript che si invoca per fare tutte le chiamate per la connessione con la Blockchain.

Il metodo load viene invocato al caricamento della pagina internet. L'user decide il wallet da usare (nel nostro caso quello di Metamask). Questo wallet espone il provider come l'oggetto globale in javascript. Questo oggetto si interfaccia con la Blockchain. Per interfacciarsi alla Blockchain bisogna essere ad un nodo della rete, Metamask usa Infura.

```
load: async () => {
```

```

    await App.loadWeb3(); //settaggio Web3 (lo prende da Metamask)
    await App.loadAccount(); //settaggio Account utente
    await App.loadContracts(); //caricamento e settaggio Smart Contracts
  },

```

loadWeb3 - Quando un utente carica una pagina Metamask direttamente fornisce un Web3 e un Ethereum Provider. Web3 si collegherà al Provider per interfacciarsi alla Blockchain. Se non si trova nessuna istanza web3 ci si collega a Ganache.

loadAccount - mi crea un account usando la chiave privata del mio wallet. L'account quindi viene firmato con la chiave privata in modo che posso chiamare il mio Smart Contract con Ropsten. Si possono fare le chiamate direttamente alla mia Blockchain.

loadContracts - I due Smart Contract vengono caricati. Viene presa la loro rappresentazione json creata in fase di compilazione (con truffle compile)

```
App.account = web3.eth.accounts.privateKeyToAccount(App.PRIVATE_KEY);
```

Ora prendiamo come riferimento l'operazione di aggiunta di un utente nel sondaggio.

```

/*Metodo che aggiunge alla blockchain i dati del partecipante al Sondaggio*/
aggiungiAllaBlockchain: async (_cdf, _n, _c, _e) => {

  try {
    const web3 = new Web3(App.web3Provider);
    const contract_PartecipazioneSondaggio = new web3.eth.Contract(JSON.parse(App.ABI_PartecipazioneSondaggio), App.ADDR_PartecipazioneSondaggio);
    const account = App.account;
    let hash;
    //INIZIO CREAZIONE TRANSAZIONE DA INVIARE
    const transaction = contract_PartecipazioneSondaggio.methods.addUser(_cdf, _n, _c, _e);
    //ether da inviare: nel nostro progetto il value e' sempre 0
    value = 0;
    //options => altri paramentri della transazione
    const options = {
      to: transaction._parent._address, //indirizzo dello Smart contract
      //destinatario (il padre del metodo)
      data: transaction.encodeABI(), //a ABI byte string containing the data of the
      //function call on a contract,
      gas: await transaction.estimateGas({
        from: account.address,
        value: value
      }), //quantita' di gas da usare (stimato) nella transazione
      gasPrice: await web3.eth.getGasPrice(), //il prezzo del gas per questa
      //transazione e' in wei
      value: value //valore trasferito dalla transazione in wei
    };
    const signed = await web3.eth.accounts.signTransaction(options, account.privateKey);
    /*the method will return a promise combined event emitter. Resolves when the
    transaction receipt is available (hash della transazione per l'utente a video)
    */
    return new Promise(resolve => {

      //invio nuova transazione firmata e come parametro ci passo l'hash della
      //transazione codificataa tramite RLP (rawtransaction)
      web3.eth.sendSignedTransaction(signed.rawTransaction)
      //EVENTI di callback del metodo
      .on('transactionHash', function(hash_returned) {
        //la Promise ritorna immediatamente l'hash della transazione per
        //fornire un feedback all'utente
        resolve(hash_returned);
      })
      //evento emesso quando la transazione viene ricevuta e l'utente viene
      //aggiunto
      .on('receipt', function(receipt) {
        console.log("Aggiunta Utente in blockchain avvenuta con successo");
      })
      .on('error', console.error); //errore, la transazione non va a buon fine.
    });

  } catch (error) { //evento di errore

    console.log("Errore aggiunta utente alla Blockchain: " + error);
  }
}

```

```
    },  
  },  
},
```

Dopo che viene dichiarata un'istanza di contratto, viene dichiarata una variabile transaction che è la rappresentazione del metodo da invocare nello Smart Contract con i suoi parametri.

In options passo i parametri della transazione.

Quindi una volta creato l'oggetto transazione, esso è firmato con chiave privata, usando la funzione signTransaction.

La transazione bisogna inviarla firmata, ciò mi permette di non stare a passare da Metamask per la singola approvazione.

Come primo parametro di signTransaction passo i parametri precedenti settati per la transazione e come secondo parametro la chiave privata dell'account che serve per firmare la transazione.

Questo metodo mi ritorna una Promise che ritorna subito l'hash della transazione per fornire un feedback all'utente.

Inoltre quando la transazione verrà minata e sarà disponibile la "fattura" della transazione, l'utente verrà aggiunto in Blockchain (condizione .on('receipt', function(receipt)).

Per ultimo, analizziamo come catturare i dati emessi dall'evento di un metodo e decodificarli.

Nella foto sottostante un esempio per far visualizzare le generalità del vincitore dopo che è stato estratto.

```
.on('receipt', function(receipt) {  
    /*Non appena la transazione e' stata minata (e quindi e' disponibile la "  
    ricevuta" della transazione)  
    vengono catturati i dati del vincitore (codice, nome, cognome, email)  
    emessi dall'evento nel metodo*/  
    const typesArray = [{  
        type: 'string',  
        name: 'messenger'  
    },  
    {  
        type: 'string',  
        name: 'cdf'  
    },  
    {  
        type: 'string',  
        name: 'nome'  
    },  
    {  
        type: 'string',  
        name: 'cognome'  
    },  
    {  
        type: 'string',  
        name: 'email'  
    }  
    ];  
  
    //Dalla ricevuta della transazione (che e' un JSON) acediamo al logs  
    const data = receipt.logs[0].data;  
    const decodedParameters = web3.eth.abi.decodeParameters(typesArray,  
        data);  
  
    //estraiamo tutti i valori ritornati dall'evento  
    const cdf = decodedParameters.cdf;  
    const nome = decodedParameters.nome;  
    const cognome = decodedParameters.cognome;  
    const email = decodedParameters.email;  
  
    //log di sviluppo (F12 in Chrome)  
    console.log("cdf " + cdf);  
    console.log("nome " + nome);  
    console.log("cognome " + cognome);  
    console.log("email " + email);  
  
    //resolve della Promise con i valori da visualizzare nel front end  
    resolve([cdf, nome, cognome, email, hash]);  
})  
.on('error', console.error); // If a out of gas error, the second  
parameter is the receipt;
```

```

    });
  } catch (error) {
    console.log("error estraiVincitore: " + error);
    alert("Si e' verificato un errore. Probabilmente non ci sono partecipanti al
          Contest");
  }
},

```

7 Validazione

Per la fase di testing è stato fondamentale l'utilizzo della suite Truffle.

Si è utilizzato in particolare il tool Ganache per riprodurre in locale (localhost) un'istanza della rete Blockchain Ethereum, sui cui nodi poter applicare le logiche degli Smart Contract prodotti. Questo tool ci ha anche dato la possibilità di creare account Ethereum (dedicati unicamente al testing), potendo anche generare un portafoglio ETH a nostro piacimento, indispensabile per poter effettuare transazioni.

Avremo la possibilità di specificare la porta sulla quale esporre un EVM client, con la quale il nostro applicativo NodeJS si interfacerà.

Il framework Truffle ci darà anche la possibilità di compilare lo Smart Contract Solidity ed effettuare il deploy all'interno della Blockchain sull'indirizzo nel quale è abilitato Ganache. Truffle ci darà anche la possibilità di eseguire i test automatizzati.

Il testing dell'applicativo prodotto è stato diviso in due fasi.

La prima fase è stata quella del testing automatizzato. Durante la fase di analisi si sono progettati test automatizzati per verificare la correttezza dello Smart Contract che si sarebbe dovuto sviluppare. E' stato fatto sull'ide Remix e poi con Ganache. Le funzionalità testate sono le seguenti:

- Corretta esecuzione della fase di deploy.
- Corretto inserimento di un utente.
- Corretta estrazione di un vincitore.
- Corretto reset partecipanti e vincitori

Si può concludere dicendo che la totalità dei test è andata a buon fine.

Per ultimo è stato fatto del testing non automatizzato. Per verificare il funzionamento dell'interfaccia sviluppata si è testato l'applicativo su rete Ropsten, verificando che le operazioni implementate andassero a buon fine, ossia che interagissero correttamente con la rete Blockchain.

```
//truffle-config.js
networks: {
  // Useful for testing. The `development` name is special - truffle uses it by default
  // if it's defined here and no other network is specified at the command line.
  // You should run a client (like ganache-cli, geth or parity) in a separate terminal
  // tab if you use this network and you must also set the `host`, `port` and `network_id`
  // options below to some value.
  //DEPLOY SU GANACHE
  development: {
    host: "127.0.0.1",      // Localhost (default: none)
    port: 7545,            // Standard Ethereum port (default: none)
    network_id: "5777",    // Any network (default: none)
  },

  //DEPLOY SU ROPSTEN
  ropsten: {
    provider: function() {
      return new HDWalletProvider(mnemonic, `https://ropsten.infura.io/v3/${infuraKey}`)
    },
    network_id: 3,        // Ropsten's id
    gas: 5500000,         // Ropsten has a lower block limit than mainnet
    confirmations: 2,     // # of confs to wait between deployments. (default: 0)
    timeoutBlocks: 200,   // # of blocks before a deployment times out (minimum/default: 50)
    skipDryRun: true      // Skip dry run before migrations? (default: false for public nets )
  }
}
```

Deploy su Ganache e Ropsten

Interfaccia principale di Ganache

Visualizzazione transazioni (andate a buon fine) su Ganache

8 Istruzioni per il deploy

In questo capitolo si descrivono le istruzioni per il deploy. Il capitolo 10 sarà poi riservato a Geth.

8.1 Come eseguire la Dapp

Seguirà una lista da seguire passo per passo per l'esecuzione dell'ambiente sviluppato all'interno del proprio computer.

Per iniziare bisogna scaricare il progetto, che si trova nel mio repository su gitlab (<https://gitlab.com/pika-lab/courses/ds/projects/sd-project-zandoli-aa2021>).

Poi bisogna installare npm a questo indirizzo: (<https://www.npmjs.com/get-npm>).

E poi installare la libreria web3 con il comando: **npm install web3**.

Poi bisogna installare Truffle con il comando **npm install -g truffle**.

Bisogna avere il Browser Google Chrome, averlo settato come browser predefinito e installare il plugin di MetaMask.

In Metamask fare l'import del mio wallet, è quello che ho usato come amministratore e ha degli ether di test da usare su Ropsten.

Per importarlo, tramite la funzione apposita del plug-in, bisogna inserire il valore della variabile "mnemonic" che trovate nel file truffle-config.js.

Avere il wallet è fondamentale per testare il progetto perchè altrimenti si dovrebbe creare un nuovo wallet, modificare la variabile "mnemonic", ricompilare il progetto con il comando "truffle compile", ri-deployare gli Smart Contract, con il comando "truffle migrate --network ropsten" e inserire gli indirizzi degli Smart Contract deployati nelle variabili ADDR_PartecipazioneSondaggio e ADDR_VincitoriDelSondaggio che si trovano in testa al file App.js.

Se si usa un nuovo wallet si deve anche modificare la variabile PRIVATE_KEY, che si trova sempre in testa al file App.js, con la chiave privata del wallet creato.

Il prototipo, come descritto, usa il servizio Infura per il deploy degli Smart Contract. La chiave del progetto creato su Infura è stata inserita all'interno della variabile "infuraKey" del file truffle-config.js.

Si consiglia quindi di non modificare queste variabili o creare nuovi progetti o accounts, ma qualora lo si volesse fare bisogna poi aggiornare la variabile "infuraKey" con la nuova chiave di progetto assegnata da Infura.

Una volta che tutto è installato e correttamente configurato, posizionandosi con il prompt dei comandi all'interno della cartella del progetto bisogna eseguire il comando che avvierà lite server: **npm run dev**.

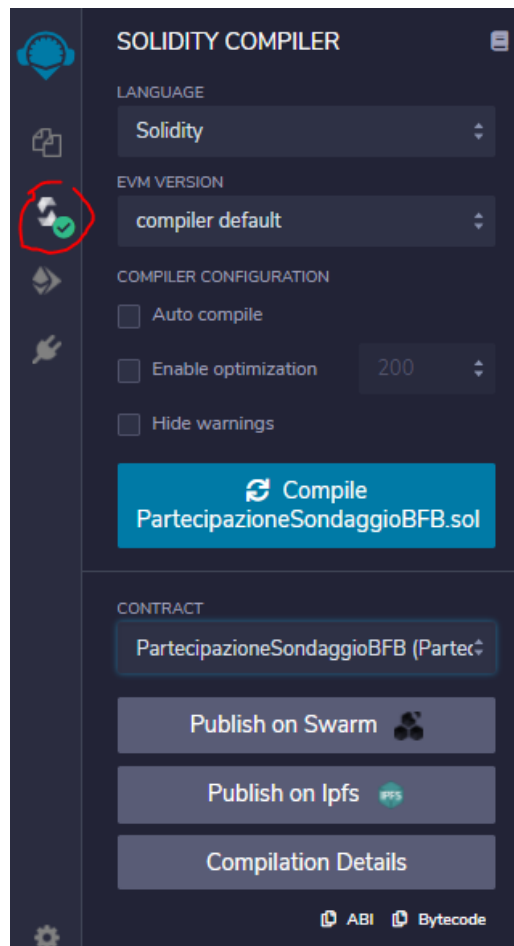
Il comando lancerà il browser Chrome e si collegherà a MetaMask (potrebbe chiedere la password di Metamask creata dopo l'import del mio wallet e la frase di seed). Per la frase di seed si può utilizzare quella che ho lasciato nel file Metamask.txt. Dopo di che si potrà testare l'applicativo come descritto nella sezione apposita.

8.2 Come compilare e testare i contratti su Remix

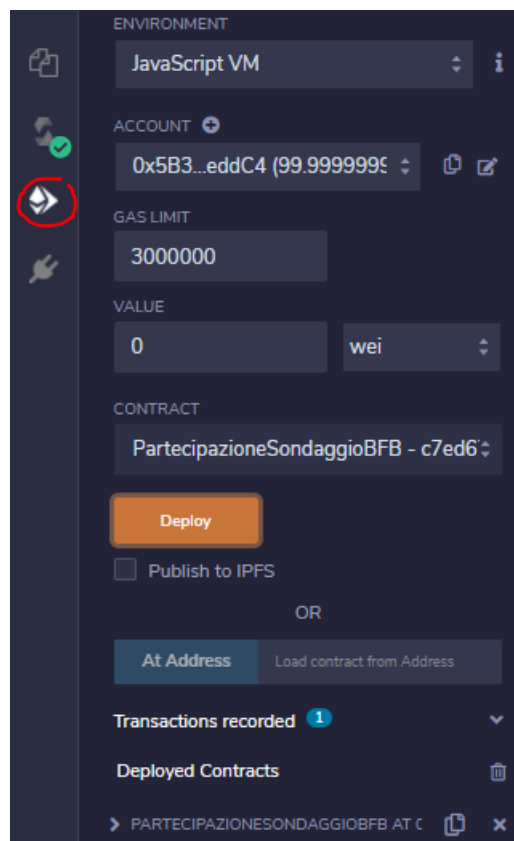
Qualora si volesse solo compilare e testare il corretto funzionamento degli Smart Contract su Remix riporto qui di seguito i passaggi per la compilazione e deploy con immagini in allegato. In fondo al paragrafo ho anche riportato il link di accesso a tali Smart Contract su Remix.

Prima di tutto bisogna compilare i nostri Smart Contract. Owner.sol non necessita di essere compilato essendo incluso nei nostri due Smart Contract principali.

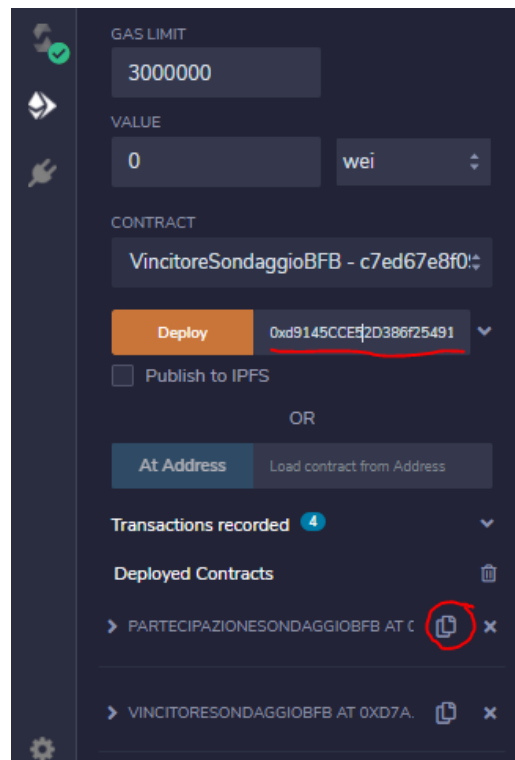
Da compilare nel seguente ordine: PartecipazioneSondaggioBFB.sol e poi VincitoreSondaggioBFB.sol.



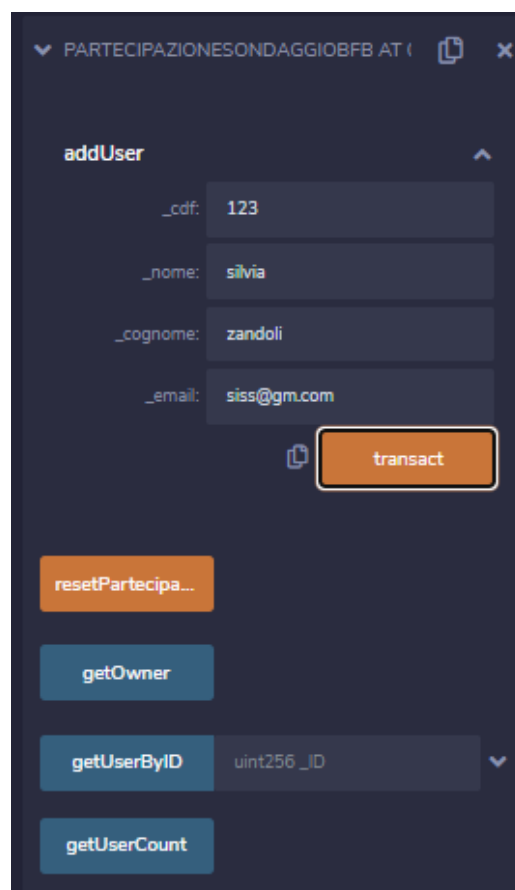
Dopo la compilazione come si può vedere nella figura sopra si può memorizzare anche l'ABI e il bytecode del contratto compilato (cliccando sui rispettivi bottoni in basso a destra). Dopo aver compilato entrambi si procede con il deploy di PartecipazioneSondaggioBFB.sol



Ora si può fare il deploy di VincitoreSondaggioBFB.sol. Come si vede dalla figura in fase di deploy bisogna indicargli l'indirizzo di PartecipazioneSondaggioBFB.sol precedentemente deployato. Infatti, come già spiegato nel Capitolo 5.1, si è creata una referenziazione tra i due contratti in modo da attingere allo stesso bacino di utenti.



Nella figura sottostante si possono vedere le funzioni che si possono richiamare di PartecipazioneSondaggioBFB. Ora si prova ad aggiungere un utente e quindi eseguire una transazione.



La transazione è andata a buon fine come si può qui vedere (si possono anche visualizzare tutte le informazioni della transazione):

```
✓ [vm] from: 0x5B3...eddC4 to: PartecipazioneSondaggioBFB.addUser(string,string,string,string) 0xd91...39138 value: 0 wei
data: 0x01c...00000 logs: 1 hash: 0x6c0...77f60

status      true Transaction mined and execution succeed
transaction hash  0x6c0cb242a65fa96d61ccf16b4c1bbaf91cd9ddd226105e4e4673b8f070877f60
from        0x5B380a6a701c568545dCfcB03FcB875f56beddC4
to          PartecipazioneSondaggioBFB.addUser(string,string,string,string) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas         3000000 gas
transaction cost 164675 gas
execution cost  139563 gas
hash         0x6c0cb242a65fa96d61ccf16b4c1bbaf91cd9ddd226105e4e4673b8f070877f60
input        0x01c...00000
decoded input { "string _cdf": "123", "string _nome": "silvia", "string _cognome": "zandoli", "string _email": "siss@gn.com" }
decoded output {}
logs        [ { "from": "0xd9145CCE52D386f254917e481eB44e9943F39138", "topic": "0x65389231ab6740a3523c340f54c9e2d674a6d4fe071d34f132f471fe681a066b", "event": "UserAdded", "args": { "0": "1", "1": "123", "2": "silvia", "3": "zandoli", "4": "siss@gn.com", "id": "1", "cdf": "123", "nome": "silvia", "cognome": "zandoli", "email": "siss@gn.com" } } ]
```

Dopo aver aggiunto qualche partecipante, si può procedere all'estrazione del vincitore e alla sua visualizzazione (getVincitore). Nella figura sottostante ci sono tutte le funzioni che si possono richiamare di VincitoreSondaggioBFB.sol.

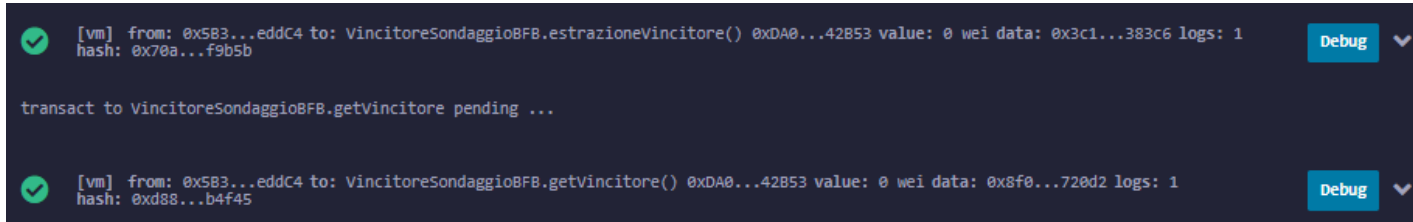


Ora mostro dei casi di revert. Motivi di revert nelle transazioni possono essere inserimento di dati errati o mancanti (per esempio nella funzione di aggiunta di un utente), out of gas e anche richiamare funzioni prima di chiamarne altre. Ad esempio, non posso fare il reset dei partecipanti o estrarre un vincitore se non è stato inserito alcun partecipante al sondaggio, etc. Se per esempio si prova a visualizzare un vincitore nel caso in cui prima non se ne sia estratto uno, la transazione non andrà a buon fine, come si può vedere in figura.

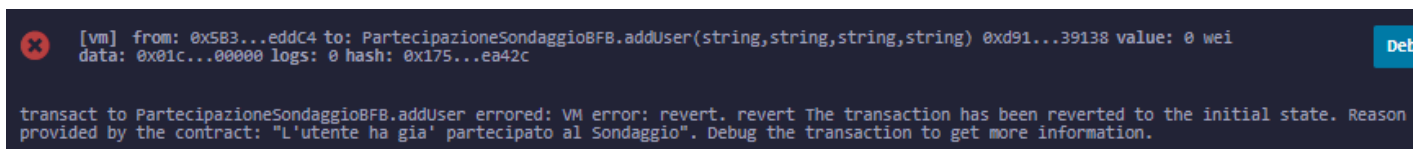
```
✗ [vm] from: 0x5B3...eddC4 to: VincitoreSondaggioBFB.getVincitore() 0xDA0...42B53 value: 0 wei data: 0x8f0...720d2 logs: 0
hash: 0x2ad...1d83f

transact to VincitoreSondaggioBFB.getVincitore errored: VM error: revert. revert The transaction has been reverted to the initial state. Reason provided by the contract: "Non è stato sorteggiato nessun vincitore". Debug the transaction to get more information.
```

Si estrae allora prima il vincitore. Prima di estrarre un vincitore si deve controllare però che siano stati inseriti dei partecipanti al sondaggio. Dopo aver estratto si potrà visualizzare anche il vincitore. Le transazioni di estrazione e di visualizzazione ora andranno a buon fine. Cliccando sulla piccola freccia a destra di Debug si può allargare e vedere tutte le informazioni relative alla transazione di interesse.



In figura sotto c'è un altro caso di revert della transazione perchè durante l'inserimento di un utente si è inserito un codice fiscale già presente.



8.3 Link contratti su Remix e Github

Fornisco il link per accedere ai miei contratti su Remix oppure su Github:

- il link che accede a Remix aprendo gli Smart Contract è: <https://remix.ethereum.org/#version=soljson-v0.7.0+commit.6c089d02.js&optimize=false&gist=c7ed67e8f0979df1f940af84ddea8d24>
- il link per l'accesso su github è: <https://gist.github.com/c7ed67e8f0979df1f940af84ddea8d24>

9 Esempi d'uso Dapp

Si mostrano ora alcune immagini dell'interfaccia client, con qualche spiegazione.

Sondaggio di Belle Folie Boutique

Inserisci i tuoi dati e compila il sondaggio

Codice Fiscale:

Enter your Fiscal Code

Nome:

Enter your name

Cognome:

Enter your surname

Email:

Enter your email

Il mio sondaggio

Qual'è la sua valutazione generale

☐ Ottima

del negozio?

☐ Buona

☐ Poco Buona

Con quanta frequenza compra

☐ Una o più volte in un mese

presso il nostro negozio?

☐ Più volte in un anno

☐ Una volta all'anno o anche

☐ meno

È soddisfatto dei prodotti

☐ Molto

acquistati?

☐ Piacentissimo

☐ Per niente

Altri commenti/suggerimenti?

Enter your comment here...

Invia

Pagina principale sondaggio per l'utente.

L'amministratore catturerà i dati inseriti dall'utente e provvederà a inserirli in Blockchain. Ogni utente potrà controllare di essere stato inserito nel sondaggio controllando la sua transazione sulla Blockchain. Per fare ciò deve cliccare sul tasto oro ticket.



Sondaggio di Belle Folie Boutique

Grazie! Parteciperai all'estrazione del vincitore!

La tua partecipazione al sondaggio è stata inserita in Blockchain

Verifica la tua partecipazione prendendo nota della tua transazione o clicca sull'immagine del ticket

0x0f12ea56b3e7680f5f34b3632be0c550c7c6c177c697f2a421ef8306b317c808



Link alla transazione della figura:

<https://ropsten.etherscan.io/tx/0x0f12ea56b3e7680f5f34b3632be0c550c7c6c177c697f2a421ef8306b317c808>

E' prevista una pagina di amministrazione (che possiamo vedere nelle due immagini in basso).

L'amministratore può estrarre un vincitore, visualizzarlo il vincitore e visualizzare l'hash della transazione. Riporto il link alla transazione dell'estrazione del vincitore che è stata fatta in questo esempio:

<https://ropsten.etherscan.io/tx/0x672a972c46fced16da7df1b46ab8c9be07142c06b75b34218ce0b748cc27f65e>

Estrazione vincitore

Avviare l'estrazione del vincitore tra i partecipanti al sondaggio.

[Estrai Vincitore in BlockChain](#)


Dati Vincitore

Codice Fiscale

Nome

Cognome

Email

Hash Transazione 

[Visualizza vincitore precedentemente estratto](#)

Pagina amministratore: funzionalità di estrazione e visualizzazione vincitore

Reset Contest

Nella sezione seguente è possibile resettare i dati raccolti per il seguente sondaggio del Brand.

Con "Reset Vincitore" si cancella solo il vincitore estratto e sarà possibile far partire un nuovo contest con gli stessi partecipanti.

Con "Reset Partecipanti" si cancelleranno tutti i dati relativi al contest.

[Reset Vincitore](#) [Reset Partecipanti](#)

Smart Contract su Ropsten

[Estrazione Vincitore](#) [Partecipazione al Sondaggio](#)

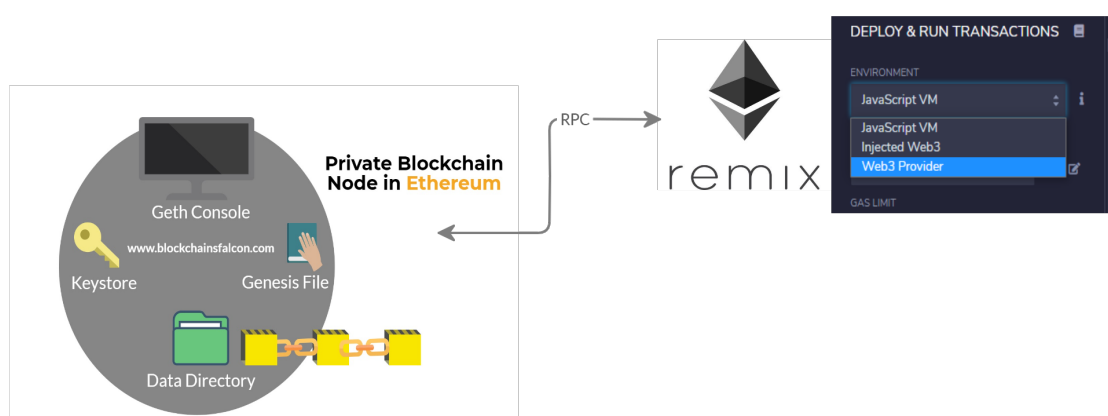
Per il progetto non è stato prevista una vera e propria autenticazione alla pagina dell'amministratore. Si veda il capitolo 11.1.

10 Interazione con Geth

Si ricorda che il lavoro fatto con Geth è completamente indipendente dallo sviluppo della mia Dapp, ed è stato fatto anche per imparare a interagire con questo tool per interfacciarsi alla Blockchain. Il link per scaricare Geth sul proprio computer è il seguente: <https://geth.ethereum.org/downloads/>.

10.1 Geth console

Geth permette di interfacciarsi con la Blockchain usando direttamente la linea di comando.



10.2 Interazione con Geth

In questo capitolo si riportano una serie di comandi per interagire con Geth. Comunque tutti i comandi usati si possono trovare anche nel file **istruzioniGethProgetto** che si trova nel progetto.

Come già visto si è usato web3 per l'interfacciamento con la Blockchain. Con web3 si è potuta creare un'interfaccia web semplice e usabile dove si poteva interagire facilmente con gli Smart Contract e controllare le transazioni su Ethereum.

Oltre a web3 si è anche usato Geth che permette di interfacciarsi con la Blockchain usando direttamente la riga di comando.

Qui di seguito si riportano le principali operazioni utilizzate per interagire con gli Smart Contract.

E' possibile tenere in una cartella del computer tutti gli account che vengono creati in modo che non siano temporanei e rimangano solo in memoria, usando l'opzione `--datadir`. Si consiglia di restare nella cartella di default dove si avvia il prompt dei comandi.

```
mkdir test-chain-dir //creazione cartella
```

Geth viene inizializzata in modalità dev e viene abilitato RPC che viene utilizzato per connettere applicazioni remote al nodo della rete usando la connessione HTTP. In questo caso si viene connessi all'ide di Ethereum Remix, dove sono presenti i contratti (ho lasciato il link nel capitolo 8.3).

```
geth --datadir test-chain-dir --rpc --allow-insecure-unlock --dev --rpccorsdomain "https://remix.ethereum.org,http://remix.ethereum.org"
```

Si apre un altro prompt dei comandi. Ora si deve lavorare da qui. Si effettua la connessione alla locazione IPC del nodo digitando:

```
geth attach ipc:\\.\pipe\geth.ipc
```

Una volta che Geth viene connesso in modalità dev si può interagire con esso in qualsiasi modo. E viene creato automaticamente un account/wallet. Si possono già fare diverse operazioni a linea di comando. Ad esempio si può creare un nuovo account che verrà salvato nella cartella test-chain-dir.

```
personal.newAccount()
```

Si possono già trasferire Ether dall'account coinbase al nuovo account creato:

```
eth.sendTransaction({from:eth.coinbase, to:eth.accounts[1], value: web3.toWei(0.05, "ether")})
```

E si può controllare l'attuale saldo dell'account creato:

```
eth.getBalance(eth.accounts[1])
```

Ora bisogna accedere a Remix, dove ci sono i contratti. Si devono compilare manualmente sull'Ide i due contratti e si abilita, come Environment, Web3 Provider che permette di collegarsi a un nodo remoto della rete (come si vede anche nella figura del paragrafo 9.1). In questo modo avviene il collegamento remoto. Verranno visualizzati nell'IDE nella sezione Account i due wallet creati.

10.2.1 Deploy di un contratto

Geth fa capire meglio i vari processi di deploy di un contratto. Invece usando Remix o altre piattaforme il meccanismo viene nascosto.

Il contratto da deployare viene compilato su Remix. Da Remix bisogna salvarsi l'ABI e il bytecode.

Prima di tutto su linea di comando bisogna salvare in una variabile address tra "" l'indirizzo con cui si vuole fare il deploy. Si trova su Remix (sezione Deploy, barra 'Account') ed è il primo account (che sarebbe quello di default, quello "unlocked"). Si deve poi specificare l'account di default come:

```
account=eth.accounts[0]
```

Si deve specificare poi una variabile bytecode e tra "" si salva solo il campo object del bytecode del contratto che si è compilato su Remix inserendo un 0x davanti. Di seguito sulla riga di comando si scrive questo codice:

```
simpleContract = web3.eth.contract(abi);
simple = simpleContract.new(42, {
  from: account,
  data: bytecode,
  gas: 0x47b760
}, function(e, contract) {
  if (e) {
    console.log("err creating contract", e);
  } else {
    if (!contract.address) {
      console.log("Contract transaction send: TransactionHash: " + contract.transactionHash
        + " waiting to be mined...");
    } else {
      console.log("Contract mined! Address: " + contract.address);
      address = contract.address;
      console.log(contract);
    }
  }
});
```

10.2.2 Invio di una transazione

Una volta che il contratto è "deployato" nella EVM è ora di invocare il contratto. Usando il nuovo indirizzo generato, un'istanza di contratto può essere creato e le sue funzioni invocate. Per inviare una transazione prima di tutto si deve impostare un account di default scrivendo


```
web3.eth.defaultAccount = eth.accounts[0]
```

Si deve salvare poi nella variabile abi l'ABI del contratto compilato che trovo su Remix.

Si definisce poi una variabile `testcontractContract` che definisce il contratto, sul quale si può interagire e richiamare le funzioni a riga di comando:

```
testcontractContract=web3.eth.contract(abi).at(address)
```

Ora si può provare a richiamare la funzione `addUser` del contratto `PartecipazioneSondaggioBFB`.

```
count=testcontractContract.addUser("12","silvia","z","s@z.com")
```

E si salva l'hash della transazione.

```
contractTx = web3.eth.getTransaction(count)
```

Si possono così visionare tutte le informazioni della transazione. Sono le stesse informazioni che si potevano visualizzare su etherscan usando web3. Ad esempio si può vedere il numero di blocco, il gas utilizzato, il prezzo del gas, l'address da cui è stata inviata la transazione, l'hash della transazione etc.

Si può vedere una transazione su Geth nella figura sotto.

Dopo avere invocato una funzione e dopo che il contratto è stato minato si stampa la transazione.

[illegible]

Ora si vogliono leggere i log della transazione. In input si può leggere il risultato ottenuto, ma esso necessita di essere decodificato.

```
console.log(web3.toAscii(contractTx.input).replace(/\u0000/g, ''))
```

10.2.3 Modifica account di default

Se poi si volesse cambiare l'account di default con un altro, questi sono i passaggi:

```
// create a account that use empty password
account = personal.newAccount();
// using an existing account in node
// var account = eth.accounts[0]
// unlock account for 300 seconds with empty password
personal.unlockAccount(account, "pwd", 300);
// set it as default account
web3.eth.defaultAccount = account;
```

11 Conclusioni

La parte più interessante è stata sicuramente la fase di studio iniziale, grazie alla quale sono riuscita ad addentrarmi nel contesto Blockchain per comprenderne il reale funzionamento. La Blockchain è una tecnologia all'avanguardia, che sta prendendo sempre più piede. Ethereum è solo una delle Blockchain con le sue caratteristiche e peculiarità. Una Blockchain pubblica come Ethereum si sposa molto bene per un caso d'uso come questo che avevo come obiettivo evidenziarne i punti di forza.

Penso che la fase di sviluppo degli Smart Contract sia invece stata tra le più difficoltose, poichè Solidity è un linguaggio ancora in fase di sviluppo, per cui ancora carente per alcuni aspetti. Mi sono trovata un attimo in difficoltà nel capire come far interagire i diversi contratti sviluppati e nel fare la referenziazione tra il contratto della partecipazione e il contratto del vincitore. In Solidity operazioni banali diventano difficili, come il confronto tra stringhe o la possibilità di ritornare strutture dati più complesse. Questo perchè purtroppo il linguaggio non fornisce librerie per fare ciò. L'aspetto positivo è che il linguaggio di programmazione è sempre in continua evoluzione.

Il framework Truffle si è rivelato molto utile e soprattutto facile da utilizzare.

Anche la parte javascript non è stata tanto facile da scrivere però sono rimasta stupita del funzionamento del framework NodeJS, che non avevo mai utilizzato prima. Mi sono trovata bene con la gestione di funzioni asincrone e delle promise. E' stato comunque meno complicato utilizzare la libreria web3js in quanto molto documentata.

Essendo riuscita a portare a termine gli obiettivi definiti nella sezione 2 si può quindi concludere di avere portato a termine con successo il progetto.

11.1 Lavori futuri

Dovendo rientrare all'interno di un monte ore stabilito in partenza (90h), purtroppo, non si ha avuto la possibilità di sviluppare alcune funzionalità, che sarebbero state interessanti.

Sarebbe stato interessante estendere Lite Server come un server di backend più complesso che potrebbe essere di appoggio anche per salvare delle transazioni, per salvare qualcosa anche in un database relazionale per fare per esempio data analytics, machine learning, etc. Nella mia corrente applicazione ho fatto solo un lite server che gira in locale e lancia la mia applicazione con Metamask. Un server e un DB permetterebbero anche di gestire utenti e autenticazione.

Sarebbe interessante rendere disponibile questa applicazione anche per sistemi mobile e tablet. Inoltre sarebbe interessante in un futuro dare la possibilità di scegliere i parametri di una transazione relativi al Gas, in particolare si potrebbe lasciare all'utente la decisione del Gas Price nel caso volesse una transazione più veloce.

Infine sarebbe opportuno gestire la crittografia dei dati sensibili quali chiavi private, mnemonic, infuraKey e tutti i dati ritenuti sensibili dopo un'accurata valutazione. Sarebbe opportuno anche cifrare i dati sensibili dell'utente quali codice fiscale, nome, cognome e email prima di inviarli in Blockchain. Infatti queste informazioni sensibili non andrebbero mai messe in chiaro in un database in cui tutti possono avere accesso in lettura. Se si volesse mai mettere in produzione questa applicazione bisognerebbe quindi fare un re-design che eviti la memorizzazione in chiaro di dati sensibili sulla Blockchain.

11.2 Cosa ho imparato

Mi è piaciuto molto avere la possibilità di poter spaziare su varie tipologie di progetto.

Sono contenta di avere avuto la possibilità di studiare tecnologie e paradigmi che non si sarebbero visti in altri corsi di studio, quali la tecnologia Blockchain, la piattaforma Ethereum e gli Smart Contract e Solidity (uno dei linguaggi di programmazione che permette di crearli).

References

- [1] Ethereum, <https://ethereum.org/en/>.
- [2] Solidity, <https://docs.soliditylang.org/en/v0.7.0/>
- [3] Truffle, <https://www.trufflesuite.com/>
- [4] Web3Js, <https://web3js.readthedocs.io/en/v1.3.0/getting-started.html>
- [5] Andrea Omicini Giovanni Ciatto, *Blockchain and Smart Contract*, 2019/2020.
- [6] Blockchain, https://blog.osservatori.net/it_it/blockchain-spiegazione-significato-applicazioni
- [7] Medium, <https://medium.com/coinmonks/https-medium-com-ritesh-modi-solidity-chapter1-63dfaff08a11>
- [8] Geth, <https://geth.ethereum.org/docs/>
- [9] textbook Alibaba per Geth, <https://www.alibabacloud.com/>