Ethereum Smart Contract dApp for vehicle mileage tracking

Filippo Pistocchi
filippo.pistocchi4@studio.unibo.it

December 2019

L'idea del progetto in questione è nata dalla necessità, da parte del mercato dell'usato di veicoli stradali, di rendere la compravendita il più trasparente ed affidabile possibile.

Essendo oggigiorno molto frequenti le truffe dovute alla vendita di mezzi con conta chilometro alterato, in quanto il chilometraggio di un mezzo è il fattore chiave considerato dai consumatori al momento dell'acquisto, si è deciso di sviluppare una tecnologia in grado di prevenire la modifica di quest'ultimo. Un' acquisto trasparente permetterà al consumatore di valutare i costi di manutenzione futura che ne deriveranno.

L'idea proposta è di creare una dApp (applicazione decentralizzata), basata su Smart Contract Ethereum, in grado ricevere segnali inviati dai veicoli (relativi al chilometraggio), per poi memorizzarli in un registro pubblico e distribuito. L'integrità dei dati all'interno di questa nuova tecnologia sarà estremamente affidabile, in quanto basata su blockchain e smart contracts.

Una attenta fase di analisi è stata seguita da una fase di progettazione, nella quale si sono valutate e successivamente definite le scelte architetturali adottate.

Lo sviluppo dello Smart Contract ha seguito un approccio test driven, portando a termine con successo i test progettati.

Ci si è concentrati poi sullo sviluppo di una tecnologia client in grado di poter interagire con la rete blockchain, sulla quale si sono eseguiti i test grazie al framework Truffle.

Essendo riuscito a potare a termine gli obiettivi definiti si può quindi concludere di avere portato a termine con successo il progetto entro la deadline prevista.

Contents

1 Introduzione **3**
- 1.1 Cos'è la blockchain? . . . 3
- 1.2 Smart Contracts di Ethereum . . . 3
  - 1.2.1 Ciclo di vita di uno Smart Contract Ethereum . . . 6

2 Obiettivi e risultatati attesi **6**
- 2.1 Scenari . . . 7
- 2.2 Politica di autovalutazione . . . 8

3 Analisi dei requisiti **8**
- 3.1 Requisiti funzionali del client Node . . . 9
- 3.2 Requisiti non funzionali . . . 9

4 Design **10**
- 4.1 Struttura . . . 10
  - 4.1.1 Dominio applicativo . . . 11
  - 4.1.2 La rete blockchain di Ethereum . . . 11
  - 4.1.3 Il framework Truffle . . . 12
  - 4.1.4 Client Node . . . 13
  - 4.1.5 Transazioni Ethereum . . . 13
- 4.2 Comportamento . . . 15
  - 4.2.1 Comportamento della rete Ethereum . . . 15
  - 4.2.2 Comportamento del client Node . . . 15
- 4.3 Interazione . . . 15

5 Dettagli implementativi **16**
- 5.1 Smart Contract Solidity . . . 16
- 5.2 Client NodeJS . . . 18
  - 5.2.1 Esecuzione delle operazioni . . . 18
  - 5.2.2 Gestione delle transazioni . . . 19

6 Auto valutazione **20**
- 6.1 Testing . . . 20

7 Istruzioni per il deploy **21**

8 Esempi di utilizzo **22**

9 Conclusioni **23**
- 9.1 Lavori futuri . . . 24
- 9.2 Cosa abbiamo imparato? . . . 24

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 Contracts ed infine la piattaforma Ethereum. In questo capitolo introduttivo sarà riportato il lavoro svolto nella fase iniziale del progetto, ossia la fase di approfondimento e di studio individuale.

1.1 Cos'è la blockchain?

Spesso il termine blockchain viene confuso con Bitcoin, Ethereum e Smart Contract, ma, seppure siano tecnologie strettamente dipendenti e collegate, rappresentano realtà differenti.

La blockchain è un modello di computazione distribuita, decentralizzata e replicata, basata su una struttura dati a blocchi. Tale struttura è definita da una catena di "blocchi", contenete dati, immutabili nel tempo, ordinati in ordine cronologico ed ognuno contenente un riferimento all'hash del blocco precedente. Ogni blocco memorizza le informazioni relative ad una transazione, ossia una variazione allo stato del sistema. La possibilità di eseguire transazioni all'interno di un sistema blockchain è data unicamente agli account registrati, i quali, tramite l'utilizzo di una chiave pubblica (che identifica l'account) ed una privata, saranno in grado di autenticare operazioni transazionali tramite firma digitale, generando blocchi da inserire in coda alla catena. Questo meccanismo permette di stabilire facilmente l'autenticità e la veridicità di un operazione e di identificare eventuali manomissioni o attacchi.

Una volta prodotto un blocco pertanto, lo si dovrà validare. Esistono diverse tecniche e protocolli di validazione che non saranno trattate in questa breve introduzione. In figura 1 è rappresentata la modalità di verifica tramite firma digitale dell'autenticità dei messaggi.

L'integrità della struttura blockchain è garantita anche dalla replicazione dell'informazione, infatti la struttura della rete blockchain è basata su una suddivisione in più server replica, ognuno contenente una copia della blockchain. Essendo un sistema decentralizzato (peer to peer), ossia non basato su un server centrale (a differenza della maggior parte delle applicazioni web), è molto più resistente ad attacchi e manomissioni.

Garantendo molti vantaggi a livello di sicurezza tuttavia, il prezzo da pagare, è in termini di performance. Gli algoritmi di validazione e di fault tollerance abbassano drasticamente il troughput della rete blockchain. In figura 2 è riportato un grafico di comparazione tra Bitcoin ed altri sistemi di pagamento digitale.

Tecnologie come Bitcoin ed Ethereum sono specifiche implementazioni di blockchain che si basano su questo modello computazionale.

1.2 Smart Contracts di Ethereum

Uno Smart Contract, applicato in un contesto blockchain, è una struttura logica immutabile, deterministica (con gli stessi input viene prodotto sempre lo stesso output),

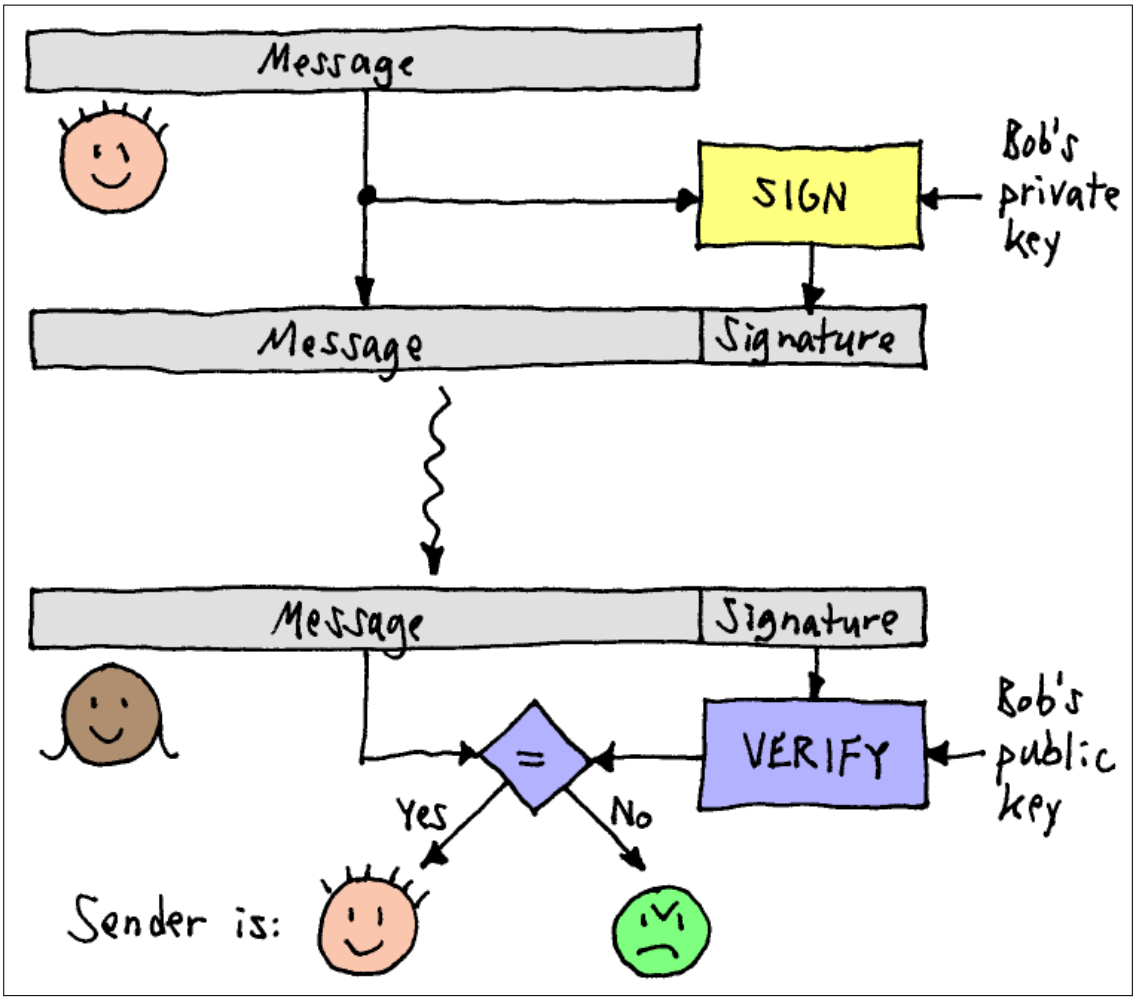

Figure 1: Firma digitale

affidabile e user defined (definita da un utente). Sono chiamati "Smart Contract", perché permettono di fatto ad uno sviluppatore di definire una sorta si contratto o interfaccia di operazioni da offrire agli utenti. Queste, una volta invocate, rilasceranno un determinato servizio, definito dallo sviluppatore del contratto.

Uno Smart Contract quindi fornisce un'astrazione in grado di modellare il dominio applicativo di una applicazione decentralizzata, ed applicare le logiche sviluppate all'interno della blockchain, permettendo allo sviluppatore di interagire con la complessità della struttura.

L'idea di Vitalik Buterin (il fondatore di Ethereum) è nata proprio dalla necessità di offrire uno strumento che desse la possibilità di sviluppare applicazioni "general purpose" ed applicarle in un contesto blockchain decentralizzato.

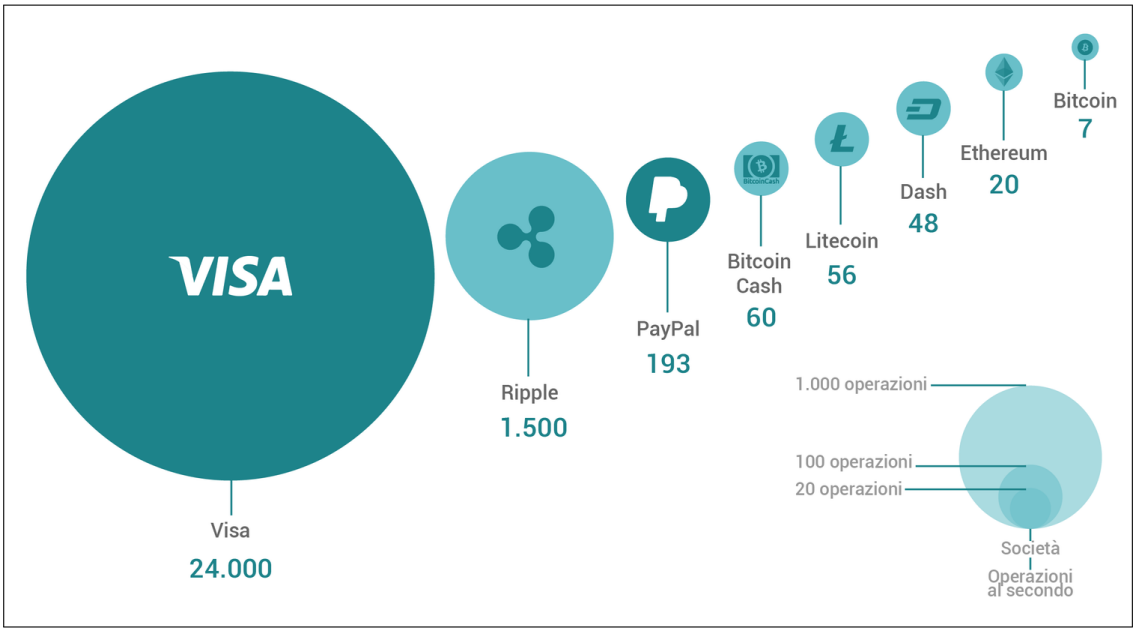

Figure 2: Throughput dei principali sistemi di pagamento digitale, misurato in operazioni al secondo. Visa e PayPal, non essendo decentralizzati garantiscono un numero più elevato di operazioni rispetto ai restanti sistemi decentralizzati che gestiscono cripto valute.

Il progetto Ethereum[1] è proprio finalizzato a risolvere questa necessità.

Un applicazione basata su Smart Contract Ethereum gode della possibilità di avere un sistema per la gestione di account, portafogli e pagamenti completamente integrata.

Ethereum definisce un unità di pagamento per effettuare della computazione (Gas), generalmente proporzionale alla complessità dell'operazione. Lo stato di un contratto Ethereum può essere modificato eseguendo transazioni, la cui computazione richiede l'utilizzo di Gas. L'invocazione di una funzione di uno Smart Contract che effettuerà transazioni, richiederà un consumo di Gas da parte dell'account che l'ha invocata. L'importo pagato verrà assegnato a colui che validerà la transazione (ad esempio un miner nel caso in cui la validazione sia basata sul proof of work), sotto forma di ETH, ossia la cripto valuta di Ethereum.

Altra caratteristica molto importante della rete Ethereum è la struttura dei nodi. Viene infatti offerto un centro di calcolo che permette l'esecuzione di Smart Contracts al di sopra della piattaforma Ethereum. All'interno dei nodi di rete viene eseguita una EVM (Ethereum virtual machine), ossia un software in grado di emulare in tutto e per tutto una macchina fisica, attraverso un processo di virtualizzazione in cui vengono assegnate le risorse fisiche (CPU, RAM, disco fisso, ecc...) alle applicazioni che vengono eseguite al di sopra della macchina virtuale. La struttura di una EVM sarà dettagliata maggiormente nella sezione 4.1.2

[1] `https://ethereum.org`

1.2.1 Ciclo di vita di uno Smart Contract Ethereum

Ethereum mette a disposizione un linguaggio di Programmazione (turing completo) chiamato Solidity[2].

Gli smart contracts, sviluppati con linguaggi di alto livello, prima di essere eseguiti, devono essere compilati producendo il bytecode interpretabile poi dalla EVM.

Una volta compilato il contratto è necessario effettuare il deploy per poterlo mettere in esecuzione all'interno delle EVM. Sarà necessario un account con bilancio ETH positivo 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 sviluppatore dello 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. In figura 3 è definito il ciclo di vita di uno Smart Contract Ethereum.

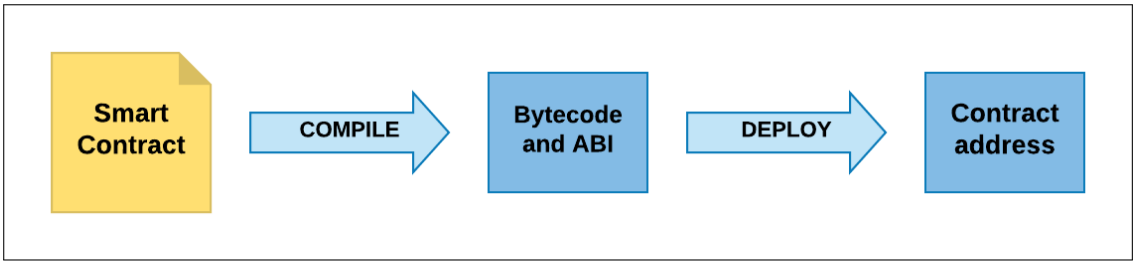

Figure 3: Ciclo di vita di uno Smart Contract Ethereum

2 Obiettivi e risultatati attesi

Si è sviluppata una lista di obiettivi incrementale per definire in maniera più chiara possibile il percorso che si seguirà per l'esecuzione e sviluppo del progetto.

- La prima fase di progetto consiste nello studio della BCT (Blockchain Technology) e degli Smart Contracts.
- Si studierà poi il framework Ethereum, con particolare attenzione al linguaggio offerto per l'implementazione degli Smart Contract (Solidity), e la relativa suite Truffle, che sarà utilizzata in fase di testing del sistema.
- Inizierà poi la fase di sviluppo dove si progetteranno i test automatizzati per verificare la correttezza dello Smart Contract, ed infine si creerà il contratto che andrà a modellare il dominio applicativo descritto nella sezione 4.1.1.

[2] `https://solidity.readthedocs.io/en/v0.5.13/`

- Ultimata questa fase si svilupperanno i test automatizzati utilizzando il framework Truffle[3] e la libreria ChaiJS[4] per NodeJS[5]. Si verificherà che il deploy dello Smart Contract vada a buon fine. Si validerà poi il funzionamento delle funzionalità core del sistema, come l'incremento del chilometraggio e la corretta gestione ed associazione dei veicoli agli account Ethereum.

- La fase successiva consisterà nello sviluppo di un interfaccia a riga di comando che permetterà di comunicare con un server copia all'interno della blockchain. Sarà infatti possibile interfacciarsi con la rete blockchain mediante le funzionalità riportate all'interno dello Smart Contract sviluppato precedentemente. Per fare ciò si studierà la libreria web3JS[6] che verrà utilizzata tramite Node JS.

- Ci sarà infine una fase di testing non automatizzato, che ci permetterà di verificare il funzionamento e la corretta interazione tra il tool prodotto nella fase precedente e la rete blockchain. In questa fase il framework Truffle (Ganache) sarà fondamentale per poter caricare una rete di EVM (Ethereum virtual machine) sulla quale eseguire lo Smart Contract prodotto precedentemente. Il tool ci permetterà inoltre di generare un numero limitato di account (finalizzati al testing) con portafoglio ETH non vuoto, da poter utilizzare per effettuare il testing e verificare la correttezza dell'interazione (tra l'applicativo NodeJS e la blockchain) per le operazioni che richiedono di effettuare di transazioni, ossia che necessiteranno di spendere ETH.

2.1 Scenari

L'idea del progetto in questione è nata dalla necessità, da parte del mercato dell'usato di veicoli stradali, di rendere la compravendita il più trasparente ed affidabile possibile.

Essendo oggigiorno molto frequenti le truffe dovute alla vendita di mezzi con conta chilometro alterato, in quanto il chilometraggio di un mezzo è il fattore chiave considerato dai consumatori al momento dell'acquisto, si è deciso di sviluppare una tecnologia in grado di prevenire la modifica di quest'ultimo. Un' acquisto trasparente permetterà al consumatore di valutare i costi di manutenzione futura che ne deriveranno.

L'idea proposta è di creare un'applicazione decentralizzata in grado ricevere segnali inviati dai veicoli (relativi al chilometraggio), per poi memorizzarli in un database pubblico e distribuito. L'integrità dei dati all'interno di questa nuova tecnologia sarà estremamente affidabile, in quanto basata su blockchain e smart contracts.

L'interazione con il sistema avverrà mediante un software, che fungerà da client, che permetterà di accedere ai servizi offerti dallo Smart Contract presente all'interno della rete Ethereum. Le funzionalità che saranno messe a disposizione verranno dettagliate nel capitolo 3.

[3] `https://www.trufflesuite.com`
[4] `https://www.chaijs.com`
[5] `https://nodejs.org/it/`
[6] `https://web3js.readthedocs.io/en/v1.2.4/`

2.2 Politica di autovalutazione

Il prodotto sviluppato sarà valutato in base ai criteri sotto indicati.

- **Affidabilità e correttezza dello Smart Contract prodotto:** Un fattore critico per la buona riuscita di un progetto di questo tipo è sicuramente la correttezza dello Smart Contract. Essendo quest'ultimo per definizione una struttura immutabile, nel caso in cui si effettuasse il deploy di un contratto "difettoso", l'intera struttura del sistema sarebbe compromessa, rischiando di perdere la consistenza dell'informazione. Un approccio test driven darà la possibilità di verificare il funzionamento del sistema.
- **Incapsulamento delle funzionalità di dominio all'interno dello Smart Contract:** Sarà molto importante distinguere le proprietà appartenenti al dominio applicativo per poi inserirle all'interno dello Smart Contract. Non si dovranno mai gestire problematiche di dominio all'interno dell'applicazione client, perché l'utilizzo di un client differente andrebbe ad influire su logiche applicative, che dovrebbero essere univoche e deterministiche, e non di certo dipendenti dalla tecnologia di interfacciamento che si utilizza.
- **Interazione delle componenti:** Sarà importante comprendere il funzionamento delle tecnologie utilizzate, definendo in maniera chiara il loro scopo, funzionamento, ruolo e l'interazione tra di esse, proponendo una chiara architettura del sistema sviluppato.
- **Possibilità di evolvere il client sviluppato:** Credo che un'altra caratteristica molto importante sia la possibilità di fare manutenzione evolutiva al client sviluppato. Lo sviluppo di un'applicazione NodeJS modulare darà la possibilità di aggiungere (più facilmente e con meno modifiche) operazioni in parallelo ad una eventuale evoluzione dello Smart Contract (ad esempio le funzionalità introdotte nella sezione 9.1).

3 Analisi dei requisiti

L'obiettivo del progetto per il corso di Sistemi Distribuiti è sviluppare una dApp per tracciare il chilometraggio dei veicoli associati al servizio. L'applicazione sarà basata su una rete blockchain sviluppata con l'emergente piattaforma Ethereum.

Ethereum è un framework open-source che permette lo sviluppo di applicazioni decentralizzate general purpose ed account based tramite l'utilizzo dell'astrazione degli Smart Contract.

Si vuole dare la possibilità ad una casa produttrice di veicoli di inserire all'interno di essi un dispositivo embedded in grado di poter monitorare l'andatura del veicolo, per poi poter inviare periodicamente segnali per aggiornare il chilometraggio dello stesso, ad un server blockchain. A questa finalità, si svilupperà un client per l'interazione con la rete Ethereum che consterà di un'interfaccia a riga di comando.

Si vuole dare la possibilità di associare un veicolo ad un account Ethereum. Ciò permetterà di poter effettuare transazioni all'interno della blockchain per modificare lo stato del veicolo, ossia (ad esempio) l'aumento del chilometraggio.

Come introdotto nel capitolo 1, per poter modificare lo stato del sistema sarà necessario possedere un account Ethereum, con un bilancio ETH sufficientemente grande per poter pagare il Gas richiesto dalla computazione e dunque, portare a termine l'operazione con successo. L'account Ethtereum sarà fondamentale anche per poter effettuare transazioni pertanto, ogni volta volta che si eseguirà un operazione che modificherà lo stato del sistema, verrà richiesto di fornire l'indirizzo pubblico dell'account che andrà ad effettuare la transazione ed anche la chiave privata, per poter applicare la firma digitale sulla transazione.

Verrà data anche la possibilità di visualizzare lo stato dello Smart Contract, per poter controllare i veicoli iscritti al servizio e verificarne i dettagli e lo stato. Questa operazione non prevede transazioni Ethereum in quanto non altererà lo stato dello Smart Contract, per tanto non sarà necessario applicare alcuna firma digitale.

Il diagramma UML dei casi d'uso in figura 4 descrive le funzionalità offerte dal sistema che si svilupperà.

3.1 Requisiti funzionali del client Node

- Possibilità di inizializzare il client tramite un file di configurazione.
- Possibilità di effettuare transazioni, firmandole con chiave privata.
- Associare un account Ethereum ad un veicolo, fornendo targa e chilometri percorsi.
- Incrementare il chilometraggio di un veicolo.
- Possibilità di effettuare un passaggio di proprietà.
- Modificare la targa di un veicolo iscritto al servizio.
- Mostrare il veicolo associato ad un account.
- Mostrare il chilometraggio di un veicolo, data la targa.
- Mostrare il numero di veicoli iscritti al servizio.

3.2 Requisiti non funzionali

- Utilizzare le best practice di Solidity per lo sviluppo dello Smart Contract.
- Non gravare sull'intensità computazionale delle operazioni core per ridurre il Gas richiesto dalle funzioni Ethereum.
- Utilizzare un approccio di sviluppo test driven.

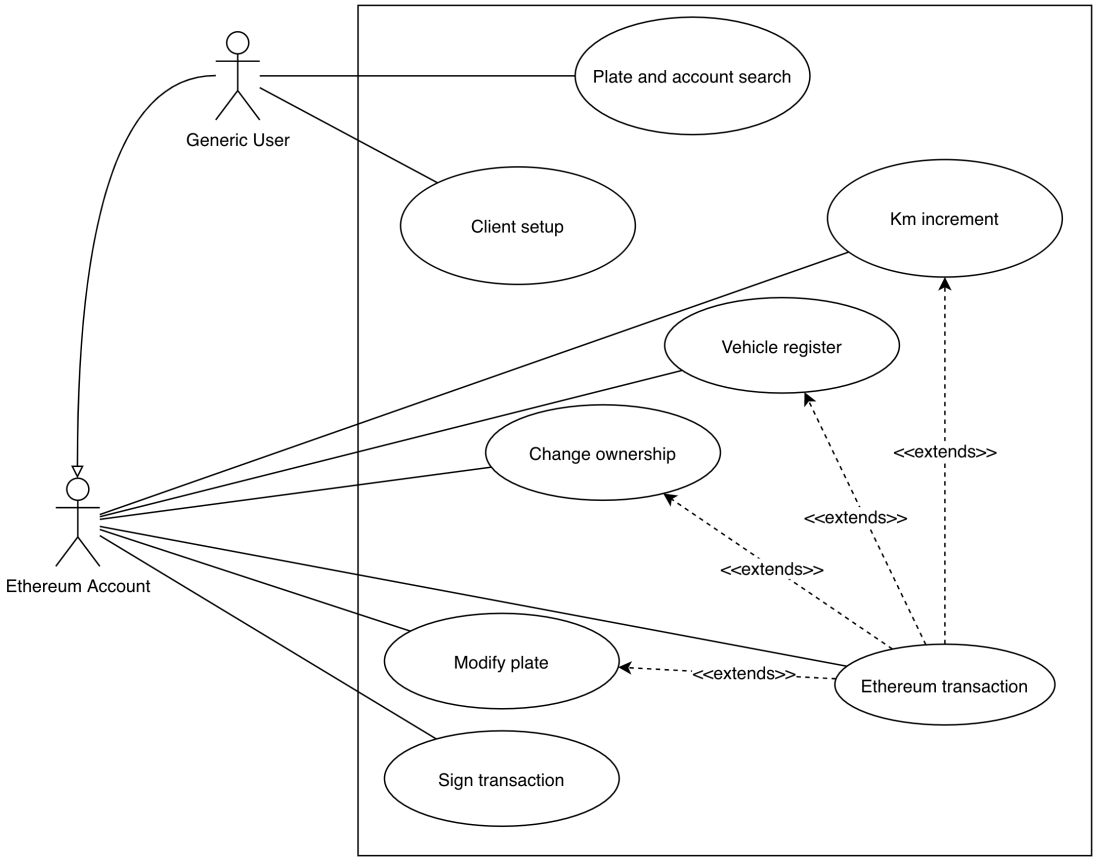

Figure 4: Use Case diagram raffigurante le funzionalità del sistema.

4 Design

In questa sezione verranno definiti gli aspetti di design salienti della nostra applicazione. Ci si concentrerà in particolare sull'architettura fisica del sistema prodotto e sul modello strutturale che si è sviluppato dato che il progetto è basato sull'utilizzo di un framework.

Come definito negli obiettivi all'interno del capitolo 2, lo scopo non è l'implementazione di una rete blockchain, ma basarsi sulla struttura fornita da Ethereum per poter sviluppare un applicativo distribuito, ereditando tutte le astrazioni e logiche di interazione fornite da essa.

È proprio per questo motivo che le sezioni dedicate al comportamento e all'interazione non saranno particolarmente ricche.

4.1 Struttura

In figura 5 è riportata l'architettura del sistema finale prodotto. Il modello architetturale utilizzato è composto una rete di EVM che eseguono lo Smart Contract sviluppato.

Esse saranno instanziate da Ganche. Un nodo copia all'interno della rete esporrà un interfaccia EVM Client (ad un URL definito). Esso esporrà le funzionalità definite nello Smart Contract. L'interazione sarà basata sull'utilizzo della libreria web3JS, il cui funzionamento è basato sul protocollo sincrono JSON-RPC. Essa ci darà la possibilità accedere alle funzioni del contratto, dandoci la possibilità di poter effettuare transazioni, autenticandole grazie al meccanismo crittografico basato su firma digitale.

Le chiamate per l'interfacciamento sono state implementate all'interno di un client NodeJS che fornisce un interfaccia a riga di comando per poter eseguire le operazioni definite nei requisisti funzionali (sezione 3.1).

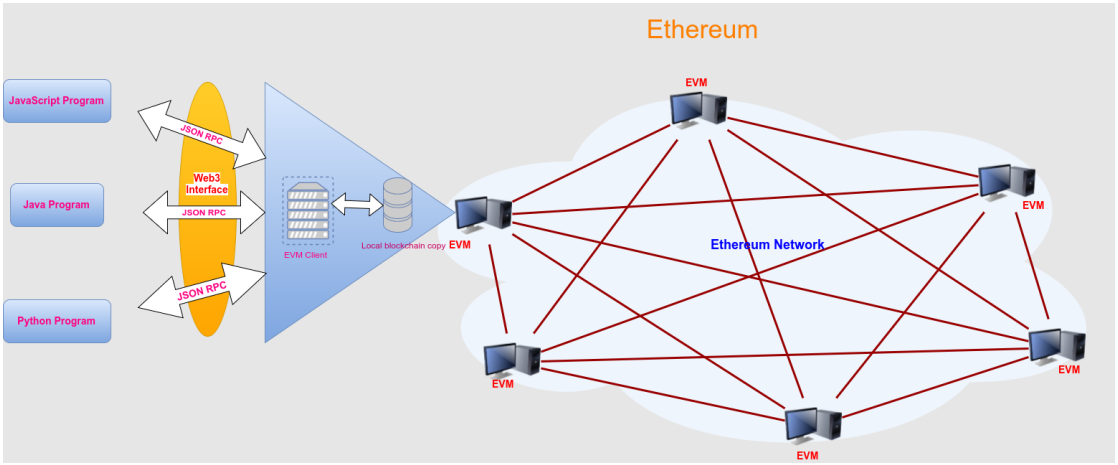

Figure 5: Architettura del sistema sviluppato. Fonte Link.

4.1.1 Dominio applicativo

L'entità di core modellata all'interno dello Smart Contract Ethereum è la struttura veicolo. Le informazioni che lo definiscono sono la targa (identificatore **non univoco**) e il numero di chilometri percorsi. Si è deciso inoltre di associare ad ogni veicolo un account Ethereum, contraddistinto da un indirizzo pubblico **univoco** ed ovviamente una chiave privata, che sarà utilizzata per firmare le transazioni, con un mapping 1:1, pertanto, ogni account potrà interagire unicamente con il mezzo associato ad esso, se presente. Il contratto sviluppato si chiamerà *CarTracker* e sarà inserito all'interno della directory *contracts*.

4.1.2 La rete blockchain di Ethereum

Proviamo ora a comprendere il funzionamento della rete di server replica offerta da Ethereum.

La piattaforma Ethereum ci permette di lavorare con 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 machcina fisica. Le EVM sono ambienti all'interno delle quali è possibile memorizzare ed eseguire uno Smart Contract, il cui comportamento sarà deterministico. Esse 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 Ethtereum (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.

In figura 6 è schematizzata l'architettura di una EVM.

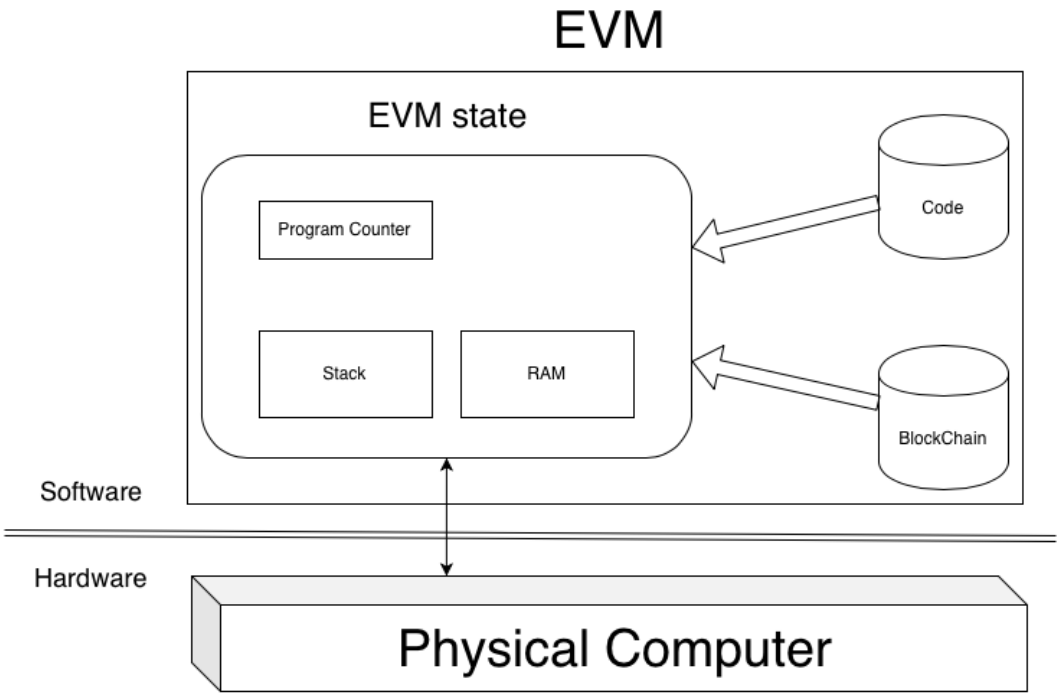

Figure 6: Architettura di una EVM.

4.1.3 Il framework Truffle

L'utilizzo del framework Truffle ci ha dato la possibilità di poter testare l'applicazione in maniera semplice ed efficace.

Partiamo definendo il ruolo del software Ganache, anch'esso incluso all'interno del framework. Ganache viene utilizzato per instanziare una rete blockchain Ethereum dedicata al testing degli Smart Contract (la struttura che Ganache ci permette di instanziare

è rappresentata dalle EVM in figura 5). Questa rete verrà sfruttata per incapsulare le funzionalità all'interno dello Smart Contract una volta effettuato il deploy del contratto all'interno dei nodi. Quando si esegue un server Ganache, si avrà la possibilità di gestire un numero limitato di account per poter di fatto interagire con l'applicazione decentralizzata. Verranno infatti forniti account Ethereum (con identificatore pubblico, chive privata e portafoglio non vuoto) che potranno essere utilizzati per effettuare transazioni. Essendo la rete blockchain prodotta, una rete Ethereum, erediterà le sue principali feature, ossia la tolleranza ai guasti e la validazione delle transazioni (e.g. minig).

Il framework Truffle ci darà anche la possibilità di gestire il ciclo di vita di uno Smart Contract (visto nella sezione 1.2.1) Sarà infatti possibile compilare il contratto ed effettuare il deploy su una rete blockchain, che ne nostro caso sarà quella instanziata con Ganache. Con Truffle sarà inoltre possibile scrivere test automatizzati in JavaScript o Solidity per l'applicazione decentralizzata. Ci verrà inoltre fornita una console che ci permetterà di interagire con la rete blockchain sulla quale abbiamo effettuato il deploy del contratto.

4.1.4 Client Node

Per gestire in maniera efficace la ricezione di comandi si è sviluppata una funzione asincrona `schedule`, che ci permette di gestire le operazioni richieste dall'utente, eseguendo il task associato all'operazione richiesta. L'utilizzo di uno scheduler ci consentirà, in un futuro, di poter sviluppare ulteriori funzionalità client, e di poterle aggiungere, senza effettuare modifiche consistenti. L'applicazione è stata suddivisa in 4 moduli ed è contenuta all'interno del package *app*.

Nel modulo principale, *app.js*, si è definito lo scheduler, che gestirà le possibili operazioni di interazione con la blockchain.

Abbiamo poi un modulo dedicato all'utilizzo della libreria web3, *contract.js*, all'interno della quale sono state "wrappate" le primitive fornite dalla libreria che permettono l'interazione con la blockchain.

All'interno del modulo *questions.js* si sono definite funzioni per gestire l'interazione con gli utenti tramite standard input.

Abbiamo poi il modulo *tasks.js*, sul quale si basa lo scheduler, il quale sfrutta le interrogazioni all'interno del modulo *questions.js* per la ricezione degli input, per poi eseguire le operazioni all'interno del modulo *contract.js*, fornendone eventuali input catturati dalle interrogazioni.

4.1.5 Transazioni Ethereum

Come abbiamo anticipato nel capitolo introduttivo, 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. Passiamo ora a definire le componenti principali di transazione Ethereum.

Per poter effettuare una transazione è necessario stabilire l'account che la andrà ad eseguire, tramite un indirizzo Ethereum pubblico, ed un indirizzo "target", che sarà il

destinatario della transazione. Il destinatario di una transazione, nel nostro caso sarà lo smart contract che abbiamo implementato e "deployato" all'interno della rete blockchain, che in particolare sarà identificato da un indirizzo Ethereum pubblico (come indicato in figura 3).

Una transazione Ethereum è anche caratterizzata da un numero chiamato "nonce". Esso rappresenta il numero di transazioni prodotte dall'account che sta effettuando la transazione. Ogni volta che un account invia una transazione, il nonce viene incrementato di uno. 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 già inviata, sia stata validata. Non sarà nemmeno possibile validare una transazione con nonce uguale a 2 ad esempio, senza avere mai inviato transazioni con nonce 0.

All'interno di una transazione sono specificati anche dettagli relativi al Gas. Abbiamo che 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. Come già detto nel capitolo 1 questo meccanismo di compenso per validazione garantisce la sicurezza del sistema. Il Gas price 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 struttura blockchain, quindi, nel caso in cui un utente abbia necessità di scrivere una transazione il prima possibile, sarà costretto a pagare un prezzo maggiorato. Con il Gas Limit invece si può definire il costo in gas per una determinata operazione. Il nostro sistema, non necessitando di transazioni immediate, utilizza un Gas Price fisso e sarà impostato tramite il software Ganache (e.g. non è importante che l'incremento del chilometraggio sia istantaneo). Per quanto riguarda invece il Gas Limit, si è deciso di tenerlo basso per le operazioni di incremento del chilometraggio, affinché non si debba pagare un importo troppo elevato per un'operazione che di fatto ha una frequenza giornaliera, mentre lo si è tenuto alto per le restanti operazioni, ossia, l'associazione di un veicolo ad un account, la modifica della targa ed il passaggio di proprietà.

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.

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 di appendere la firma all'interno dell'oggetto transazione. In figura 1 è riportata la modalità di firma, che di fatto sarà eseguita dall'autore della transazione, e la modalità di verifica, che sarà eseguita da un opportuno validatore, designato in base alla modalità di verifica del sistema (e.g. proof of work oppure proof of steak), che dovrà effettuare la validazione.

Una volta applicata la firma alla transazione, essa sarà inviata al client EVM, che di fatto, si occuperà di validarla (con le modalità definite precedentemente) ed aggiungerla alla blockchain o respingerla nel caso in cui i parametri non siano corretti.

4.2 Comportamento

Questa sezione è stata divisa in due parti, nella prima verrà descritto il comportamento implementato dai nodi Ethereum all'interno della rete blockchain e nella seconda parte si descriverà il comportamento del client cli (command line interface) sviluppato in linguaggio NodeJS

4.2.1 Comportamento della rete Ethereum

Grazie all'utilizzo degli Smart Contracts è possibile lavorare ad un'astrazione tale da stabilire il comportamento funzionale dei nodi della rete Ethereum, definendo ed implementando servizi erogabili dal sistema tramite la definizione di funzioni. Le funzionalità offerte sono state definite nella figura 4.

È importante comprendere tuttavia il livello di astrazione sul quale stiamo lavorando. Utilizzando una piattaforma blockchain quale Ethereum, possiamo lavorare senza preoccuparci della reale complessità della rete blockchain sulla quale verrà eseguito in nostro Smart Contract. Ci verrà nascosto infatti il vero comportamento di un nodo di rete in fase di ricezione di messaggi.

4.2.2 Comportamento del client Node

Passiamo ora a definire il comportamento dell'interfaccia a riga di comando sviluppata. In figura 7 è riportata una macchina a stati dettagliata che riassume il comportamento implementato dal client sviluppato per l'interazione con la rete blockchain, dove ogni stato rappresenta una richiesta su standard input da parte del client, ed ogni transazione indica l'inserimento di un input da parte dell'utente. 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 di scheduling ci permette gestire in maniera piuttosto semplice la ricezione di comandi e la gestione delle numerose operazioni.

4.3 Interazione

L'interazione con la rete blockchain avverrà mediante l'utilizzo della libreria web3JS. Come indicato in figura 5, 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 del contratto, e ci consentirà di effettuare chiamate alle funzioni definite in esso.

Come introdotto nella sezione 4.2, 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.

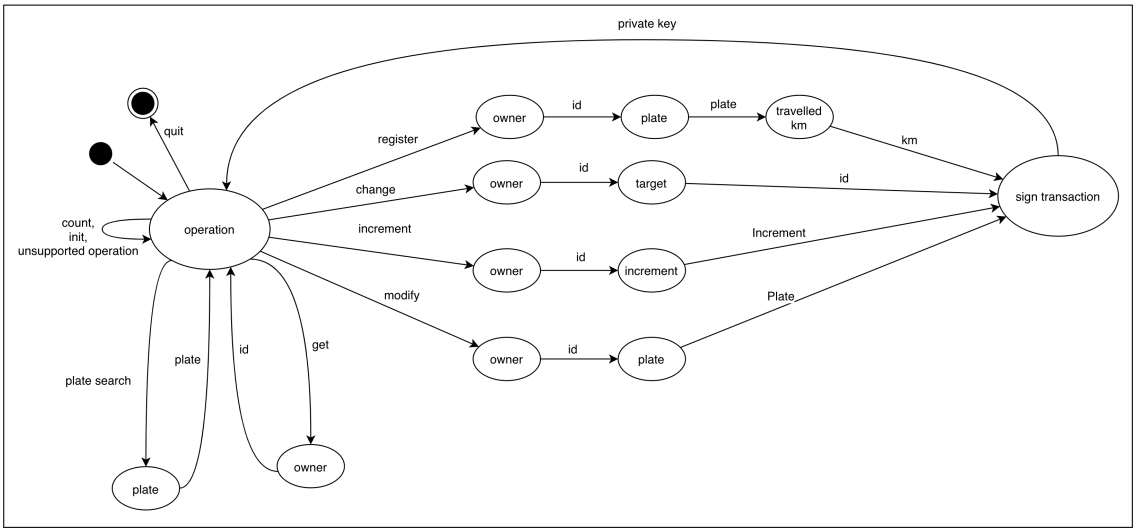

Figure 7: Macchina a stati che descrive il comportamento dell'interfaccia a riga di comando.

5 Dettagli implementativi

Dato che il linguaggio Solidity non fornisce tool per generare una documentazione, questa sezione sarà dedicata a fornire dettagli implementativi del contratto Solidity sviluppato. Si è deciso di fornire numerosi dettagli in quanto il linguaggio utilizzato, seppure object oriented, differisce da quelli tradizionali proprio perché necessita di ulteriori meccanismi ed astrazioni per poter modellare uno Smart Contract. Si sono comunque inseriti commenti all'interno del codice.

Si daranno dettagli anche sul funzionamento del client Node.

5.1 Smart Contract Solidity

La struttura dati utilizzata per memorizzare la relazione tra gli account ed i veicoli è una mappa, che associa l'indirizzo di un account Ethereum ad un veicolo, definito dalla struttura `Vehicle`.

Questa mappa rappresenta parte dello stato dello Smart Contract. Le variabili di stato all'interno di un contratto Solidity sono definite con il modificatore `public`. Il significato è differente dal public object oriented, in quanto lo stato potrà essere modificato solo con chiamate a funzione.

```
1 mapping(address => Vehicle) public vehicles;
```

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` conterrà 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.

Le proprietà `requireAccountRegistration` e `requireAccountNotRegistred` ci daranno la possibilità di stabilire rispettivamente se un account è iscritto al servizio o meno, con conseguente annullamento della funzione che le eredita.

```
modifier requireAccountRegistration() {
    require(
        vehicles[msg.sender].registered,
        "Account not registred"
    );
    _;
}

modifier requireAccountNotRegistered() {
    require(
        !vehicles[msg.sender].registered,
        "Account already registered"
    );
    _;
}
```

Solidity ci dà la possibilità di definire eventi, che una volta triggerati da funzioni, verranno inseriti all'interno del transaction log. Si sono definiti per gestire e soprattutto testare, lato client, l'inserimento dei veicoli e l'incremento del chilometraggio.

```
event InsertedVehicle (
    address owner,
    string plate
);

event KilometerIncrement (
    address owner,
    uint increment
);
```

Passiamo ora alle funzionalità core del nostro sistema.

Per effettuare l'inserimento di un veicolo è richiesto che un account Ethereum non ne abbia già uno associato, pertanto la funzione `insertVehicle` erediterà la proprietà `requireAccountNotRegistred`. Questa funzione ci permetterà di inserire un veicolo all'interno del sistema, associandolo all'account che ha effettuato la chiamata.

Prima di completare l'operazione verrà triggerato un evento tramite la proprietà `emit`.

```
function insertVehicle(string memory _plate, uint _travelledKilometers)
    public requireAccountNotRegistered {
    vehicles[msg.sender].plate = _plate;
```

```
    vehicles[msg.sender].travelledKilometers = _travelledKilometers;
    vehicles[msg.sender].registered = true;
    vehicleCount++;

    emit InsertedVehicle(msg.sender, _plate);
}
```

Abbiamo infine la funzione che ci permette di incrementare il chilometraggio di un veicolo. Essa necessita ovviamente che l'account con la quale si esegue sia registrato al sistema, implementa infatti il modificatore `requireAccountRegistration`.

```
function kilometerIncrement(uint _increment) public
    requireAccountRegistration {
    vehicles[msg.sender].travelledKilometers += _increment;
    emit KilometerIncrement(msg.sender, _increment);
}
```

5.2 Client NodeJS

Questa sezione darà dettagli implementativi sulla tecnologia client sviluppata. È pensata, in particolare, per un'eventuale sviluppatore che dovrà effettuare modifiche evolutive o adattive.

5.2.1 Esecuzione delle operazioni

L'entità principale per il funzionamento del dell'interfaccia a riga di comando è il meccanismo di scheduling sviluppato. Esso eredita il nome scheduler anche se in realtà non è basato su una gestione di task concorrenti. Il suo obiettivo è semplicemente la scelta ed esecuzione del task associato all'operazione utente richiesta. La struttura dello scheduler è sviluppata in maniera tale da poter aggiungere facilmente nuove funzionalità utente, infatti basterà sviluppare un task all'interno del modulo *tasks.js* per gestire una nuova operazione ed eventualmente sviluppare nuove interrogazioni (modulo *questions.js*) o nuove funzioni web3 (modulo *contract.js*). Sarà infine necessario creare un nuovo ramo all'interno dello scheduler per processare il task in seguito alla ricezione di una parola chiave. Nella figura sottostante è mostrata la gestione degli input principali.

```
const schedule = async () => {
  const operation = await q.getOperation();

  switch (operation) {

    case "add":
      await tasks.register();
      break;

    case "increment":
      await tasks.increment();
      break;
```

```
13
14      case "exit":
15        console.log("Quitting service...");
16        stop = !stop;
17        break;
18
19      default:
20        console.log("Unsupported operation");
21        break;
22    }
23  };
24
25  const scheduler = async () => {
26    while (!stop) {
27      await schedule();
28    }
29  };
30
31  // Entry point
32  scheduler();
```

5.2.2 Gestione delle transazioni

In questa sezione mostreremo come sono state applicate le nozioni teoriche introdotte nella sezione 4.1.5 mediante la libreria web3JS.

Prendiamo come riferimento l'operazione di incremento del chilometraggio. L'obbiettivo è quello di creare un oggetto transazione, per poi poterlo firmare con chiave privata ed infine inviarlo all'indirizzo in cui la rete Ethereum è in ascolto. Si è definita una costante `tx` che definisce al suo interno i parametri necessari per portare a termine la transazione, quali il nonce, l'indirizzo dell'account che effettua la transazione, l'indirizzo dell'account che la riceve, il Gas Limit e l'oggetto della transazione, ossia la funzione `insertVehicle` fornita dallo Smart Contract.

```
1  web3.eth.getTransactionCount(accountAddress).then(tCount => {
2    const tx = {
3      // The "nonce" in Ethtereum is the number of transaction
4      nonce: tCount,
5      // The account that execute the transaction
6      from: accountAddress,
7      // Target address of the transaction, in this case it represents the
           contract address
8      to: contractAddress,
9      // The gas limit
10     gas: 1000000,
11     // This encodes the ABI of the method and the arguements
12     data: contract.methods.insertVehicle(plate, mileage).encodeABI()
13   };
14   signAndSendTransaction(tx, pvtk);
15 });
```

Una volta creato l'oggetto transazione, esso è stato firmato con chiave privata, utilizzando la funzione `signTransaction`. Si è utilizzata poi la funzione `sendSignedTransaction` per inviare la transazione firmata. Si è sviluppata la funzione `sendSignedTransaction` per fare ciò.

```
1 function signAndSendTransaction(tx, pvtk) {
2   const signPromise = web3.eth.accounts.signTransaction(tx, pvtk);
3   signPromise.then(signedTx => {
4     web3.eth.sendSignedTransaction(signedTx.raw || signedTx.
          rawTransaction);
5   });
6 }
```

6 Auto valutazione

L'approccio utilizzato per lo sviluppo è test driven, quindi la correttezza del contratto potrà essere verificata riproducendo i test automatizzati sviluppati. Sarà anche importante verificare il funzionamento del client NodeJS per verificare l'effettiva interazione tra il tool di interfacciamento prodotto, con la rete blockchain, basata sullo Smart Contract sviluppato.

I test sviluppati si trovano all'interno della directory *tests*. Per eseguirli sarà necessario effettuare il deploy, ed infine eseguire il comando `truffle test`.

6.1 Testing

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 dello Smart Contract prodotto. 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 interfaccerà.

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.

- **Testing automatizzato**: Durante la fase di analisi si sono progettati test automatizzati per verificare la correttezza dello Smart Contract che si sarebbe dovuto sviluppare. Si è deciso di sviluppare semplici asserzioni utilizzando la libreria ChaiJS per Node. Le funzionalità testate sono le seguenti.
  - Corretta esecuzione della fase di deploy.

- Corretto incremento del numero di veicoli all'interno del servizio.
- Correttezza nell'inserimento dei veicoli all'interno del sistema.
- Coerenza con il mapping 1:1 tra account e veicolo.
- Impossibilità di accedere alle operazioni sul veicolo se assente.
- Correttezza della funzionalità core, ossia l'incremento del chilometraggio.

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

- **Testing non automatizzato**: Per verificare il funzionamento dell'interfaccia a riga di comando sviluppata si è testato l'applicativo, verificando che le operazioni implementate andassero a buon fine, ossia che interagissero correttamente con la rete blockchain.

7 Istruzioni per il deploy

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

1. Il primo passo per l'esecuzione dell'applicazione è installare npm.
2. Una volta installato npm, installare Truffle. `npm install -g truffle`.
3. Installare Ganache. È possibile utilizzare sia la versione grafica che quella da terminale. Il comando per l'installazione di quest'ultima è `npm install -g ganache-cli`.
4. Eseguire Ganache. `ganache-cli -p PORT` per il tool a riga di comando, dove al posto di `PORT` è necessario specificare la porta sulla quale si metterà in ascolto Ganache. Da notare che la porta di default per il deploy utilizzando Truffle è la 7545. Nel caso di volesse utilizzare un altra porta sarà necessario modificare il file $truffle - config.js$ inserendo il nuovo indirizzo di rete.
5. Si dovrà clonare poi il repository del progetto.
6. Compilare lo Smart Contract. `truffle compile`.
7. Fare il deploy del contratto, verificando che l'indirizzo riportato nel file $truffle - config.js$ coincida con quello di ascolto del client Ganache e che le porte coincidano. `truffle migrate --reset`

Una volta completati i passi indicati, sarà possibile avere in locale un'istanza della rete blockchain (grazie a Ganache) e sarà possibile interagirci tramite la console di Truffle, in particolare sarà possibile eseguire i test automatizzati, presenti nella cartella test, nel file $tests.js$, con il comando `truffle test`. Verranno poi indicati i test che andranno a buon fine o meno. Sarà anche possibile interfacciarsi alla rete blockchain con il comando `truffle console`, impartendo direttamente istruzioni NodeJS.

Passiamo ora alla configurazione dell'ambiente per poter permettere l'interfacciamento tramite la libreria web3JS.

1. Inizializzare Node. `npm install`
2. Sarà necessario installare anche la libreria Web3JS. Lanciare il comando `npm install web3`.
3. Si dovrà avviare poi avviare il client NodeJs `node app/app.js`.

Nella sezione successiva verranno forniti dettagli sull'utilizzo del client.

8 Esempi di utilizzo

Una volta eseguite le istruzioni indicate nel capitolo 7 sarà possibile utilizzare il client per potere interagire con la blockchain.

Una volta eseguita l'applicazione Node sarà possibile interagire con essa impartendo comandi. Il modello di interazione è definito formalmente nella sezione 4.2.2.

Il tool ci darà la possibilità di eseguire operazioni transazionali, che per essere eseguite, necessitano dell'indirizzo Ethereum che la andrà a compiere ed di una chiave privata per poter firmare la transazione stessa. L'indirizzo pubblico di un account e la chiave privata, si dovranno recuperare dal tool Ganache (se si utilizza il tool da terminale, gli indirizzi pubblici e le chiavi private verranno mostrate all'avvio).

Ricordo che per eseguire il client sarà necessario lanciare il comando `node app/app.js`.

Le operazioni offerte sono identificate da keyword, che attiveranno determinati comportamenti:

- `init`: Questa operazione consente di inizializzare il client al fine fine poterlo interfacciare correttamente con la rete fornita da Ganache, pertanto è necessario effettuarla prima di utilizzare le altre funzionalità. La configurazione è basata sulla lettura del file (*/app/config/config.json*). Sarà pertanto necessario impostare a "mano" i parametri necessari alla configurazione. Sarà necessario, una volta compilato il contratto, impostare il campo `abi` (application binary interface). L'ABI si troverà, una volta compilato il contratto, all'interno del file */build/contracts/CarTracker.json* (sotto il campo `abi`) e sarà necessario copiarla ed inserirla all'interno del file di configurazione nella corretta sezione. Sarà necessario poi impostare l'indirizzo dello Smart Contract. Sarà possibile recuperarlo con i seguenti comandi, una volta effettuato il deploy del contratto, `truffle console` e poi `CarTracker.deployed().then(c => console.log(c.address);)`. L'indirizzo recuperato dovrà essere isnerito all'interno del campo `contract_address` nel file di configurazione. Si dovrà infine impostare l'indirizzo di rete modificando il campo `network` all'interno del file JSON, scegliendo l'indirizzo di rete fornito dal Client EVM di Ganache.
- `register, insert, add`: Questa operazione sarà utilizzata per poter inserire un veicolo. Verrà poi richiesto di inserire l'indirizzo dell'account al quale lo si vuole associare, la targa, il chilometraggio ed infine la chiave privata dell'account per poter firmare la transazione.

- `increment`: Verrà data la possibilità di incrementare il chilometraggio di un veicolo. Si dovrà fornire l'indirizzo dell'account, l'incremento che si vuole effettuare ed infine la chiave privata dell'account in questione per poter applicare la firma digitale alla transazione.
- `modify`: Questa funzionalità permetterà di modificare la targa di un veicolo. Sarà necessario fornire l'identificativo dell'account Ethereum, la nuova targa ed infine la chiave privata per poter firmare la transazione.
- `change`: Questa operazione permetterà di effettuare il passaggio di proprietà di un veicolo verso un account target, che non dovrà essere associato a nessun altro veicolo. Sarà necessario fornire l'identificativo dell'account Ethereum che cede il veicolo, il nuovo account Ethtereum che riceverà il veicolo ed infine la chiave privata per poter firmare la transazione.
- `get, details-from-account`: Questa operazione permetterà di ottenere la targa ed il chilometraggio del veicolo associato ad un account Ethereum. Si dovrà semplicemente inserire l'identificativo dell'account.
- `plate-search`: Questa è la funzionalità più importante. Permette di effettuare una ricerca per targa, la quale restituirà il chilometraggio del veicolo in questione. Sarà necessario fornire la targa da ricercare.
- `count`: Questa operazione permetterà di sapere il numero di veicoli registrati al servizio.
- `exit, quit, abort`: Queste sono le istruzioni per uscire dall'applicativo.
- Ogni altra istruzione verrà rifiutata.

9 Conclusioni

La parte più interessante è stata sicuramente la fase di studio iniziale, grazie alla quale sono risuscito ad addentrarmi nel contesto blockchain per comprenderne il reale funzionamento. Studiando poi la piattaforma Ethereum sono rimasto impressionato dalla sua potenza e versatilità.

Penso che la fase di sviluppo dello Smart Contract sia invece stata la più brutta e difficoltosa, poiché Solidity è un linguaggio ancora in fase di sviluppo, per cui ancora carente secondo alcuni aspetti. Mi sono trovato molto in difficoltà nell'utilizzo della struttura dati `mapping` che non fornisce funzioni per gestire ricerche all'interno del key set e del value set, costringendomi ad utilizzate meccanismi di ricerca poco eleganti ed inefficienti. In Solidity anche alcune operazioni che possono sembrare banali, come il confronto tra stringhe, mi hanno messo in difficoltà in quanto il linguaggio non fornisce librerie per fare ciò. Altra pecca per il linguaggio Solidity è l'impossibilità di restituire dati di tipo `struct` dalle funzioni.

Per quanto riguarda il testing, posso dire di essere molto soddisfatto della suite Truffle. Il framework si è rivelato molto utile e soprattutto facile da utilizzare.

Non essendo un fan del linguaggio JavaScript, pensavo che lo sviluppo dell'interfaccia a riga di comando sarebbe stata la parte meno interessante del progetto, invece sono rimasto stupito dal funzionamento del framework NodeJS, che non avevo mai utilizzato prima. Mi sono trovato molto bene con la gestione di funzioni asincrone e delle promise. Non è stato particolarmente complicato utilizzare la libreria web3JS in quanto molto documentata.

Essendo riuscito a potare a termine gli obiettivi definiti nella sezione 2 si può quindi concludere di avere portato a termine con successo il progetto entro la deadline prevista.

9.1 Lavori futuri

Dovendo rientrare all'interno di un monte ore stabilito in partenza (90h) e dovendo includere all'interno del progetto una fase di studio, una di sviluppo ed infine una di stesura report, purtroppo, non si ha avuto la possibilità di sviluppare alcune funzionalità, che sarebbero state sicuramente molto interessanti.

Sarebbe stato molto bello poter aggiungere la possibilità di effettuare conversioni tra chilometri e miglia, in maniera tale da rendere utilizzabile l'applicazione anche in paesi basati si una differente unità di misura. Attualmente il sistema è basato unicamente sui chilometri e non supporta conversioni.

Mi sarebbe piaciuto anche poter inserire dettagli ulteriori per poter identificare più precisamente i veicoli, come ad esempio poter inserire le più disparate tipologie di veicolo (automobili, camion, mezzi di trasporto, ecc..), per poter rendere ancora più pervasiva l'applicazione. Sarebbe stato anche interessante potere definire ulteriori dettagli relativi ad un veicolo, come il modello, la casa produttrice, l'anno di immatricolazione, ecc... per poter rendere più distinguibili i veicoli iscritti.

Credo che un altro possibile miglioramento potrebbe essere aggiungere dettagli relativi al proprietario di un veicolo, dando la possibilità di indicare se un veicolo è in vendita fornendo eventualmente un contatto del venditore.

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.

9.2 Cosa abbiamo imparato?

Credo che questo progetto si differenzi in positivo rispetto ad lavori svolti per altri corsi di studio. Mi è piaciuto molto avere la possibilità di poter spaziare su varie tipologie di progetto e soprattutto sono soddisfatto di avere avuto la possibilità di poter lavorare con tecnologie molto attuali (come Ethereum).

È stato molto bello 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.

References

[1] Ethereum. https://ethereum.org, 2019.

[2] Solidity. https://solidity.readthedocs.io/en/v0.5.14/, 2019.

[3] Truffle. https://www.trufflesuite.com, 2019.

[4] Web3JS. https://web3js.readthedocs.io/en/v1.2.4/, 2019.

[5] Andrea Omicini Giovanni Ciatto. *Blockchain and Smart Contracts*. 2019/2020.