Alma Mater Studiorum · Università di Bologna

Campus di Cesena
Corso di Laurea in Ingegneria e Scienze Informatiche

Utilizzo della tecnologia blockchain applicata alle licenze software

Relatore:
Gabriele D'Angelo

Presentata da:
Yuqi Sun

III Sessione
Anno Accademico 2019-2020

Sommario

Negli ultimi anni la blockchain ha attratto molto interesse da varie aziende e ricercatori, studiando i miglioramenti che una tecnologia come questa potrebbe avere in vari settori, da quello finanziario a quello medico o immobiliare. Abbiamo scelto una delle possibili aree di applicazione, la gestione delle licenze software, e ne abbiamo sviluppato un sistema basato sulla blockchain di Ethereum. La tesi inizierà con un'introduzione al concetto di blockchain, fornendo le conoscenze necessarie per comprendere questa tecnologia; in seguito, illustrerà il processo che ha portato alla costruzione di questo sistema e infine analizzerà i possibili vantaggi e svantaggi che la blockchain può offrire in un ambito come le licenze software.

Indice

Introduzione

Il software è diventato uno strumento essenziale per persone e aziende per svolgere vari tipi di task, da quelli per leggere file in certi formati a quelli che monitorano le vendite di prodotti. L'uso di questi software è regolato dalle **licenze**, un accordo legale tra il licenziante (il soggetto che ne è proprietario) e il licenziatario (il soggetto che ne vuole beneficiare l'uso) che definisce le condizioni di utilizzo del programma.

Molto spesso, il licenziante mantiene il diritto esclusivo di riproduzione, di modifica, di utilizzazione economica e possesso del codice sorgente, mentre il licenziatario ha il diritto all'uso del software nei limiti permessi dalla licenza [1]. Un esempio di limite può essere il periodo di tempo entro cui è possibile usare il software o il numero massimo d'installazioni permesse. Licenze come queste sono spesso a pagamento e sono dette *licenze proprietarie*. Esistono anche licenze per i cosiddetti *software open-source*, dove viene provvisto il codice sorgente del programma; il licenziatario può avere il diritto di ridistribuzione, riproduzione o modifica e non è obbligato a pagare un corrispettivo [2].

Con la diffusione di Internet, è diventata più facile la distribuzione di software pirata-ti, ovvero software che è stato modificato per bypassare le misure che sono state inserite nel programma per impedirne la copia e la ridistribuzione non autorizzata. Uno studio del 2018 [3] afferma che globalmente il 37% degli software installati su PC non ha una licenza valida e che l'uso o l'installazione di programmi piratati aumenta il rischio d'incontrare malware; lo stesso studio sostiene che lo sviluppo di un robusto **Software Asset Manager** (SAM) potrebbe ridurre i rischi e i costi legati alla loro gestione.

Vogliamo quindi analizzare se gestire i software e le licenze usando la blockchain può avere vantaggi o meno. Spiegheremo nel primo capitolo il concetto di blockchain, descrivendo alcune sue caratteristiche chiave, per poi approfondire Ethereum, la blockchain che useremo in questa tesi; nel secondo analizzeremo l'attuale stato d'arte riguardo sistemi che usano la blockchain per gestire le licenze. Nei capitoli successivi viene effettuata l'analisi dei requisiti, la progettazione e l'implementazione della soluzione. La tesi si concluderà con l'analisi dei vantaggi, svantaggi e costi della soluzione sviluppata.

Capitolo 1

Background

1.1 Introduzione alla blockchain

La blockchain è un database aperto e distribuito che può essere rappresentato come una catena di blocchi collegati tra loro e resi sicuri mediante crittografia: ogni blocco contiene dei dati, un proprio codice identificativo e il riferimento al blocco precedente. Inventata da Satoshi Nakamoto nel 2008 per registrare le transazioni della cripto valuta *Bitcoin* [4], a oggi esistono molte blockchain, sia pubbliche sia private.

Vediamo di seguito le caratteristiche principali di una blockchain.

Decentramento e trustlessness

Diversamente dai sistemi centralizzati tradizionali, la blockchain è mantenuta in una network decentralizzata peer-to-peer [5]. Per esempio, se Alice vuole inviare a Bob €10 per mezzo di una banca, Alice dovrà prima identificarsi e poi aspettare che la banca validi e confermi la transazione; sarà poi la banca a trasferire il denaro dall'account di Alice all'account di Bob; similmente, se Alice desidera visualizzare il suo bilancio o la storia delle sue transazioni, dovrà fare una richiesta alla banca. Per fare qualunque azione, dobbiamo fidarci della banca, l'autorità centrale di questo sistema: è compito suo conservare in modo sicuro i dati di tutti gli account, verificare l'identità degli utenti, effettuare e validare transazioni. Con la blockchain e la sua network peer-to-peer, invece, viene eliminato questo intermediario; non solo, ogni utente possiede una propria copia dell'intera blockchain ed è lui stesso a eseguire e validare le transazioni.

È per questa ragione che la blockchain è definita **trustless**, in quanto diminuisce i *third-party services* di cui un utente deve fidarsi. In più, il sistema è **trasparente**, in quanto tutti possono consultarlo e leggere le transazioni che sono state fatte.

Meccanismo di consenso

Proprio perché è assente un server centrale che verifica le transazioni, i sistemi decentralizzati come la blockchain fanno uso dei cosiddetti **meccanismi di consenso**, protocolli che garantiscono la sincronizzazione tra tutte le copie, verificando l'ordine e la legittimità delle transazioni, in modo che tutti i nodi siano d'accordo sul medesimo stato della blockchain [6].

Per esempio, la blockchain di Bitcoin fa uso di ***proof of work***, un meccanismo di consenso dove un nodo, per aggiungere transazioni, deve dimostrare di aver speso un certo "sforzo", in questo caso risolvendo un problema matematico molto difficile: prima di tutto, le transazioni che non sono ancora state validate vengono raccolte in una struttura dati chiamata "blocco"; i nodi poi competono tra loro e il primo che individua la soluzione potrà creare il blocco, che verrà mandando in *broadcast* agli altri nodi; questi dovranno confermare la correttezza della soluzione e, se valida, aggiungono il blocco alla propria blockchain [5]. Dal momento che per trovare la soluzione sono richieste molte risorse computazionali, i nodi vincitori, in cambio del lavoro svolto, ricevono cripto valute come ricompensa. Questo processo di validazione viene detto ***mining***, mentre i nodi partecipanti sono chiamati *miners*.

Il problema che i *miners* devono risolvere consiste nel trovare l'hash del blocco tale per cui, se trattato come un numero, sia minore di un certo target. Ricordiamo che le funzioni hash, dato un input, ritornano sempre lo stesso output; quindi per cambiare l'output e ottenere un hash che verifichi le condizioni necessarie, si usa come input il blocco da validare insieme a un numero arbitrario di 32 bit, detto *nonce*. Bitcoin usa SHA-256 [7] applicata due volte, una al blocco, l'altra all'output del primo hash; essendo una funzione pseudo-randomica, l'unico modo per risolvere questo problema è effettuare tentativi su tentativi con vari numeri finché non viene trovato il *nonce* che ritorna l'hash giusto. Il blocco alla fine conterrà le seguenti informazioni: l'hash appena trovato, che farà da identificativo del blocco, l'hash del blocco precedente, il *nonce*, il *timestamp*, deciso dal *miner*, e le transazioni.

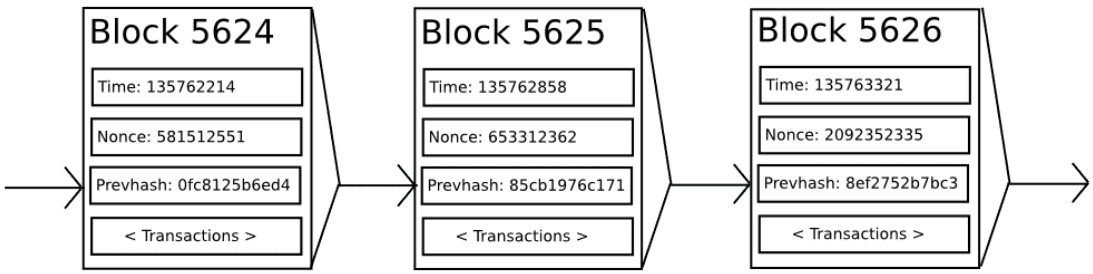

Figura 1.1: Blocchi di una blockchain [8]

Nel caso vengano trovati più blocchi validi allo stesso momento, la blockchain potrebbe dividersi in più *branch*; per decidere quale delle due è quella autentica si usa la regola della *longest chain*: in quanto è raro che due *branch* diverse generino il prossimo blocco di nuovo allo stesso tempo, quella che diviene più lunga sarà considerata quella corretta e i *miners* convergeranno a quella [5].

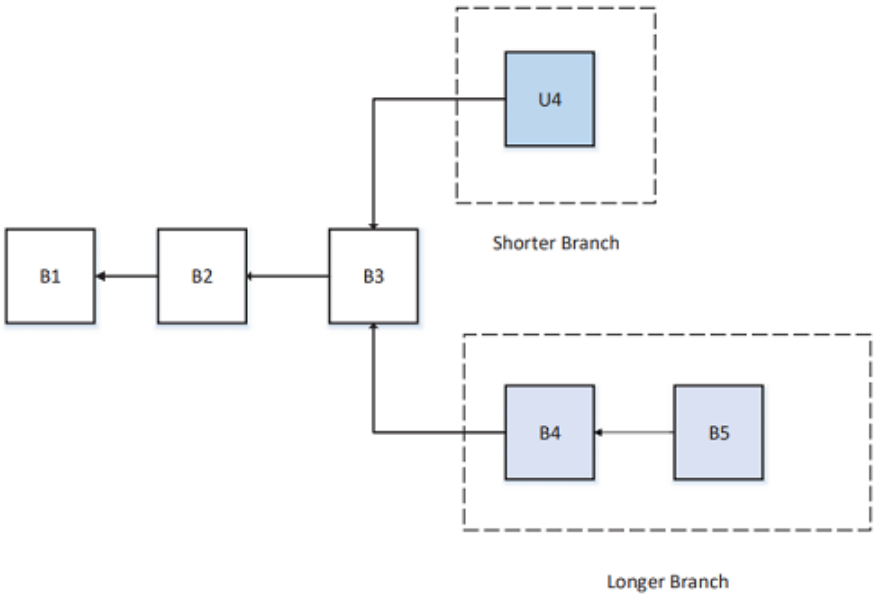

Figura 1.2: Una possibile divisione in diverse branch della blockchain [5]

Immutabilità

Una volta che un blocco è stato verificato e aggiunto alla blockchain, è molto difficile che venga modificato. Questa caratteristica è data dal fatto che i blocchi sono legati tra loro tramite crittografia: alterare una delle transazioni di un certo blocco k vorrebbe dire modificare l'input della funzione hash e di conseguenza cambierebbe anche il suo output e quindi l'hash del blocco; ma poiché il blocco $k+1$ ha un riferimento a quello precedente, anche questo dovrà avere un hash diverso e, a sua volta, anche il suo blocco "figlio". Questo effetto a cascata garantisce che un'eventuale modifica venga notata facilmente: se conserviamo in modo sicuro l'hash dell'ultimo blocco, anche se una persona riesce a modificare tutti gli hash fino alla coda della blockchain, possiamo determinare se c'è stato un cambiamento o meno [9].

Attualmente molti siti, libri e articoli descrivono che la blockchain è immutabile. Alcuni però [10, 11] fanno notare che tale affermazione è fuorviante, sostenendo che l'immutabilità non è una proprietà intrinseca della blockchain e che un gruppo di persone con abbastanza potenza computazionale possono cambiare la blockchain e decidere che la propria versione è quella vera. Comunemente, però, la blockchain pubblica continua a essere ritenuta "immutabile", in quanto modifiche alla sua storia sono molto difficili. C'è stato però un episodio, che approfondiremo nella sezione "Ethereum forks", dove ciò è accaduto.

Blockchain pubbliche e DLT

Le blockchain possono essere divise in varie categorie a seconda del loro grado di decentramento:

- Blockchain pubbliche non hanno limiti d'accesso e chiunque con una connessione Internet può partecipare alla sua network, effettuare, validare e consultare le transizioni.
- Per partecipare a una blockchain privata è necessario un invito dall'amministratore della network, tipicamente un'organizzazione; le transazioni possono essere pubbliche o private e la validazione è effettuata dall'organizzazione.
- *Consortium blockchain* sono delle blockchain ibride che hanno una struttura semi-decentralizzata. Similmente alle blockchain private, le transazioni possono essere pubbliche o private, mentre la validazione è eseguita da un numero ristretto di nodi.

Per differenziare dalla blockchain pubbliche, le altre blockchain sono anche chiamate *Distributed Ledger Technology* (DLT). Generalmente, le DLT sono più efficienti, ma a differenza di quelle pubbliche, sono più facili da modificare in quanto hanno un numero minore di nodi.

1.2 Ethereum

Ethereum è una blockchain proposta da Vitalik Buterin nel 2013 [8] e finalizzata da Gavin Wood nel 2015 con il suo Yellow Paper [12]. A differenza di Bitcoin, che è stata pensata come un registro delle transazioni della sua cripto valuta, Ethereum è stata ideata con l'intento di provvedere agli sviluppatori un ulteriore framework su cui costruire applicazioni decentralizzate. Può essere considerata come una macchina a stati basata su transazioni, dove "stato" è l'insieme di tutto ciò che è attualmente rappresentabile da un computer, come il bilancio di un account, i dati delle applicazioni etc; fornisce una macchina virtuale decentralizzata, l'*Ethereum Virtual Machine* (EVM), che esegue le transazioni, cambiando lo stato di Ethereum.

La criptovaluta di Ethereum è detta Ether (ETH) e attualmente fa uso di *proof of work*.

1.2.1 Account Ethereum e crittografia

In Ethereum esistono due tipi di account: *externally owned accounts* (EOAs) e *contracts*. Entrambi possiedono un indirizzo pubblico e possono conservare, inviare e ricevere Ether, ma solo gli EOAs possiedono una "chiave privata", grazie alla quale un utente può controllare i propri fondi, interagire con i contratti e inviare transazioni. Un contratto, invece, compie azioni a seconda della logica del suo codice; poiché non possiede una chiave privata, non può fare transazioni di sua iniziativa. Si può, quindi, considerare gli EOAs come entità "attive" controllate da utenti e i contratti come entità "passive" che rispondono alle chiamate degli EOAs eseguendo il proprio codice [13].

La chiave privata di un EOAs è molto importante: in Ethereum e altre blockchain si usa la **crittografia asimmetrica**, un tipo di crittografia che fa uso di particolari funzioni matematiche che sono facilmente calcolabili, ma di cui è difficile calcolarne l'inverso senza conoscere certe informazioni [14]. Usando la crittografia asimmetrica, deriviamo la chiave pubblica, l'indirizzo Ethereum di un EOAs, dalla chiave privata, che non è altro che un numero generato casualmente. Inoltre, unendo il contenuto di una transazione

con la chiave privata e applicando la crittografia asimmetrica, viene prodotto un codice riproducibile soltanto se si è in possesso della chiave privata. Questo codice è detto **digital signature**, con la quale viene firmata ogni transazione; in questo modo si verifica che la transazione proviene proprio da quell'account la cui chiave privata combacia con la chiave pubblica. Proprio perché è fondamentale per la propria identificazione, la chiave privata non deve essere mai condivisa: chiunque venga in possesso della chiave privata può accedere ai propri fondi ed effettuare transazioni.

In Ethereum si usa l'"Elliptic Curve Cryptography" [15], un particolare tipo di crittografia asimmetrica basato su curve ellittiche. Le chiavi private sono lunghe 256 bits, rappresentate come una stringa esadecimale da 64; le chiavi pubbliche sono gli ultimi 20 byte risultanti dall'applicazione della crittografia.

1.2.2 Smart Contract e Dapp

Gli **smart contract** non sono altro che un insieme d'istruzioni scritti con un linguaggio ad alto livello come Solidity [16] o Vyper [17]. Per essere eseguiti dall'EVM, vengono compilati in byte-code. In generale, è preferibile scrivere contratti semplici e piccoli: contratti molto grandi hanno alti costi di esecuzione, mentre la loro complessità può causare errori e bug, rendendoli vulnerabili ad attacchi e malfunzionamenti.

Dopo aver fatto il deploy sulla blockchain, un contratto non è più modificabile; può però essere eliminato implementando *Selfdestruct*, una funzione che cancella il codice e tutti i dati del contratto. Questo non elimina le transazioni che sono state effettuate precedentemente, semplicemente rende l'account del contratto un account vuoto: inviare transazioni non farà eseguire nessun codice e trasferire somme vorrà dire perdere per sempre gli Ether inviati, dal momento che non è più possibile riceverli indietro [13].

Una **decentralized applications** (Dapp) è un'applicazione che gestisce certi aspetti, come il back-end o il *data storage*, in modo decentralizzato. Possiamo quindi usare smart contracts per definire la logica back-end di una dapp e usare tecnologie tradizionali per svilupparne il front-end. Usare contratti per immagazzinare grandi quantità di dati può risultare molto costoso, per questo motivo molte dapp scelgono di archiviare i dati in altre piattaforme di storage [13]; anche qui, si può decidere se usare piattaforme centralizzate (per esempio, un cloud tradizionale) o decentralizzate, come IPFS [18] o Swarm [19].

1.2.3 Ether e Gas

L'Ether è la criptovaluta di Ethereum. Si definisce *transaction fee* la quantità di Ether che bisogna pagare ogni volta che viene creato un contratto o eseguita una transazione, mentre chiamiamo **Gas** l'unità che misura le risorse computazionali usate per la transazione [20]. Il costo finale della *transaction fee* è dato dalla quantità totale di gas usato moltiplicato per il prezzo del gas in ETH; questa somma è data come ricompensa ai *miners*.

L'utente provvede a definire la quantità massima di gas che è disposto ad usare. La quantità di gas fornita è pagata in anticipo: se il gas finisce prima del completamento della transazione, si incorre in una *Out of Gas exception*, tutte le operazioni vengono annullate e viene pagato ai *miners* il gas usato; se la transazione va a buon fine, viene pagato il gas dovuto ai *miners* e il rimanente viene rimborsato all'utente. In generale un contratto più è complesso, più è costoso da eseguire.

1.2.4 Transazione

La *transaction* registra l'azione che un EOAs vuole compiere, mentre la *transaction receipt* è il suo outcome. I principali campi sono:

- **From**: l'indirizzo dell'utente che ha iniziato l'azione
- **To**: l'indirizzo destinatario dell'azione
- **Value**: la somma di Ether inviata dal mittente al destinatario
- **Data**: campo opzionale per includere dati nella transazione

In più, la *transaction receipt* ha altri campi, tra cui:

- **Block hash**: hash del blocco
- **Block**: numero del blocco in cui è contenuta la transazione
- **Gas used**: quantità totale di gas usato

Quando si crea un contratto, il valore *To* della transazione è nullo, mentre *Data* contiene il byte-code del contratto; se la transazione va a buon fine, la sua ricevuta presenta un campo che contiene l'indirizzo del contratto. Quando invece si vuole chiamare un contratto, *To* contiene l'indirizzo del contratto, mentre *Data* contiene il nome della funzione e i suoi parametri.

1.2.5 Token

Nella blockchain di Ethereum, oltre agli Ether, si possono possedere anche dei token. I token sono la rappresentazione digitale di un *asset*. Possono essere di due tipi, fungibile o non fungibile: un token fungibile non ha caratteristiche proprie che lo possono discernere dagli altri e può quindi essere scambiato senza distinzioni con altri token fungibili; un token non fungibile, invece, è unico e non può essere scambiato indiscriminatamente. Un'altra differenza è che i token fungibili sono divisibili, quindi se un token fungibile vale 10ETH, si può ricevere o trasferire anche solo una parte di esso, mentre un token non fungibile può essere ricevuto o trasferito solo nella sua interezza.

Esistono due standard principali per rappresentare i token fungibili e non fungibili, rispettivamente **ERC-20** [21] e **ERC-721** [22], dove ERC sta per "Ethereum Request for Comments". Per creare un token che rispetti questi standard basta implementare nei contratti tutte le funzioni dichiarate.

Lo standard ERC-20 è stato sviluppato nel 2015 ed è uno dei token usati maggiormente; presta molto ad essere usato come nuova criptovaluta costruita sopra Ethereum. Golem, per esempio, permette di prestare le proprie risorse computazionali in cambio di valuta. Questa valuta non è altro che un token, detto GNT, che rispetta lo standard ERC-20. Attualmente Golem ha una propria blockchain, mentre in origine era eseguito sulla blockchain di Ethereum.

Lo standard ERC-721 è stato introdotto alla fine del 2017 e finalizzato nel 2018 ed è lo standard usato maggiormente per rappresentare asset distinguibili; possono essere fisici, come proprietà immobiliari o pezzi d'arte, o digitali, come item di collezione. Asset

come questi sono intrinsecamente univoci e dunque non scambiali liberamente: il quadro ad olio A rappresentato come token A567 è diverso dalla scultura B rappresentato come token B738. Cryptokitties [23], per esempio, è un video game dove si collezionano e si creano gatti digitali, ognuno dei quali è rappresentato come un token ERC-721 indivisibile e univoco.

Esiste anche un ulteriore standard, ERC-1155 [24], che gestisce token sia fungibili sia non fungibili e qualunque altra loro combinazione.

1.2.6 Client

Abbiamo detto che la blockchain è distribuita in una network peer-to-peer dove i nodi possono eseguire transazioni e validare i blocchi. Per eseguire il proprio nodo, è necessario installare un *client*, ovvero un programma che implementa le specifiche di Ethereum e comunica con gli altri nodi [25]. I client possono eseguire diversi tipi di nodo:

- *Full node*: conservano l'intera blockchain, fornendo dati agli altri quando richiesto; partecipano alla validazione di ogni blocco arrivati in broadcast dalla rete.
- *Light node*: conservano soltanto l'*header* dei blocchi (hash, *nonce*, *timestap* etc.), senza i dati delle transazioni; non verificano tutti i blocchi. Sono più veloci e leggeri, in quanto conservano una quantità minore di dati, e quindi possono essere usati in dispositivi che possiedono quantità limitata di memoria.
- *Archive Nodes*: conservano tutta la blockchain come nel *Full node*; in più, conservano l'intera storia dello stato della blockchain, ovvero l'insieme di tutti i dati degli account, i loro bilanci, assieme a tutti gli *smart contract*. In questo modo è possibile chiedere, per esempio, il bilancio di un account in un determinato blocco. È molto usato per servizi come *blockchain explorer*.

Client molto usati sono Geth e OpenEthereum.

Eseguire personalmente un *client* ci permette di usare Ethereum in modo anonimo, auto sufficiente e sicuro, dal momento che siamo noi stessi a validare i dati. Eseguire un client, però, può essere difficile e può necessitare di molta memoria. In alternativa quindi ci si può avvalere di *third-party API* con la quale possiamo collegarci alla blockchain come Infura o Alchemy.

1.2.7 Wallet

Data l'importanza della chiave privata, questa dev'essere custodita in modo sicuro. Conservare la chiave privata senza criptarla o proteggerla con password o con altri mezzi può essere molto rischioso. In generale, gli utenti gestiscono le proprie chiavi per mezzo di *wallets* [13], un termine che può riferirsi a:

- Applicazioni ad alto livello che offrono all'utente un'interfaccia per gestire i propri fondi e le chiavi private ed effettuare transazioni; possono anche interagire con contratti e dapp.
- Strettamente il mezzo con la quale l'utente conserva le chiavi private.

Keystore File (UTC-JSON)

Un file *keystore* è un file JSON che contiene la chiava privata criptata da una password. Una volta generato, l'unico modo per recuperare la chiave privata è decriptarla usando la password scelta. È consigliabile avere backup multipli. Molti client Ethereum come Geth e OpenEthereum usano *keystore*.

Wallet deterministici e Mnemonic phrases

Un *wallet* si dice deterministico se le chiavi private derivano da una singola *seed*: il *seed* è un numero randomico che, combinato con altri dati, genera delle chiavi private. Attualmente l'*hierarchical deterministic wallet* è lo standard più usato [26]: le chiavi private sono rappresentate in una struttura dati ad albero, dove il *seed* genera una *master key*, che a usa volta genera le chiavi "figlie" e così via, grazie all'uso di particolari funzioni.

Il metodo di back-up più diffuso per questi tipi di *wallet* è la *mnemonic phrase*, detta anche *recovery phrase* o *seed words* [27], una lista ordinata contenente dalle 12 alle 24 parole, che unite possono ricreare il *seed*.

```
//Example of a seed for a deterministic wallet, in hex
FCCF1AB3329FD5DA3DA9577511F8F137

//Example of a seed as a mnemonic phrase of 12 words:
wolf juice proud gown wool unfair wall cliff insect more detail hub
```

Hardare wallet

Uno dei metodi più sicuri per conservare il proprio account è usare un *hardware wallet*, un dispositivo fisico che conserva le chiavi private [28]. Non essendo mai connesse online, hanno un minor rischio di subire attacchi e ad oggi non è ancora avvenuto un attacco in cui la chiave è stata rubata con successo. Non sono comunque infallibili, dal momento che un hacker molto abile potrebbe ottenere la chiave se riesce prendere possesso del device fisico. In ogni caso, è comunque consigliato fare il backup della *mnemonic phrase*.

A differenza di altri metodi, posso essere meno comodi da usare e più costosi.

Altri Wallets

Eventualmente è possibile utilizzare applicazioni ad alto livello con la quale è possibile gestire il proprio account Ethereum. Molto spesso queste applicazioni permettono di creare un nuovo account o importarne uno già in nostro possesso. Possono essere applicazioni mobile, desktop o browser.

Esempi di questo tipo di *wallet* sono MetaMask, MyEtherWallet o Portis.

1.2.8 Ethereum forks

Abbiamo detto che una blockchain è comunemente ritenuta immutabile. In realtà, esistono dei rari casi in cui viene modificata.

Dal momento che sono in continuo sviluppo, a volte c'è necessità di fare upgrades alle blockchain. In casi come questi viene fatta una *fork*, ovvero una "divisione": se i cambiamenti effettuati sono retro-compatibili, si parla di *soft fork* e rimane valida una singola blockchain, altrimenti parliamo di *hard fork*, creando invece due blockchain distinte. Quando gli sviluppatori annunciano una *hard fork*, informano tutta la comunità, in modo che abbia il tempo necessario per effettuare gli upgrades al software, e specificano il numero del blocco in cui verranno fatte le modifiche; in generale, la nuova blockchain viene adottata da tutti ma esiste un caso dove non è stato così.

Nell'aprile del 2016 era stato creato su Ethereum "DAO", una collezione di *smart contract* che collezionava fondi per finanziare progetti su Ethereum: un utente poteva inviare Ether a DAO e ricevere il diritto di votare quale progetto finanziare. Due mesi dopo, un hacker sfruttò una vulnerabilità in uno dei contratti di DAO, rubando con successo una quantità di Ether pari a quasi €40 milioni [29]. Data la somma persa, gli sviluppatori di Ethereum proposero due soluzioni: una *soft fork* che avrebbe bloccato indefinitamente gli Ether trasferiti da DAO, inclusi quelli dell'hacker, o una *hard fork*, ritornando a un blocco antecedente all'attacco. Alla fine, si decise di effettuare una *hard fork*, ma non tutta la comunità era contenta della decisione, in quanto andava contro i principi su cui era basata Ethereum; non solo, avrebbe stabilito un pericoloso precedente dove in futuro casi simili sarebbero potuti risolti nello stesso modo [30]. Coloro che rifiutarono la *fork* continuarono a lavorare nella vecchia blockchain, attualmente ancora attiva con il nome di "Ethereum Classic" [31].

Capitolo 2

Stato d'arte

In questa tesi, parleremo strettamente di licenze per software proprietari dove l'azienda mantiene diritto esclusivo del programma e il suo uso è sottoposto a particolari condizioni, come il divieto di modifica e di riproduzione; anche il numero di installazioni permesse potrebbe essere limitato. Si possono avere licenze a iscrizione, dove si può usare il software solo per un certo periodo di tempo, o perpetue, dove l'utente può usare il software per un tempo indeterminato. Esistono diversi modi per validare una licenza software, per esempio attraverso una chiave di validazione, online o usando device hardware.

Integrare la blockchain per gestire le licenze software potrebbe migliorare il tracking delle licenze e prevenire la pirateria. Nel loro articolo "A Novel Method for Decentralised Peer-to-Peer Software License Validation Using Cryptocurrency Blockchain Technology" [32] pubblicato nel gennaio 2015, Jeff Herbert e Alan Litchfield illustrarono due possibili metodi per validare una licenza software: il "Master Bitcoin Method" e il "Bespoke Model".

2.1 Master Bitcoin Method

Il "Master Bitcoin Method" è un modello proposto da Christian Fortin (2011) [33] e implementato sperimentalmente da Aaron Lebo nel 2014 [34]. Questo modello si basa sull'ipotesi che un utente è legittimo titolare di una licenza se esiste una transazione tra l'azienda proprietaria del software e l'utente. Dati i seguenti attori:

- A: l'azienda che vende il software S
- S: il software
- I: l'indirizzo sulla blockchain che rappresenta S
- U: l'utente

1. A crea nella blockchain l'indirizzo I e trasferisce a I dei bitcoin che chiameremo *Master bitcoin*.
2. Una volta che U ha comprato una licenza per S con una transazione tradizionale, A invia da I a U un Master bitcoin.
3. Quando S dovrà validare la licenza, andrà proprio a verificare l'esistenza di questa transazione tra I e U.

La quantità di bitcoin inviata non è rilevante in quanto la transazione in sé è prova del possesso di un Master bitcoin e quindi il diritto all'uso di S. Poiché stiamo parlando di criptovalute, il trasferimento di un Master bitcoin a un utente U2 comporterà anche il trasferimento della licenza: U2 potrà utilizzare il software S finché S verifica che la transazione del Master bitcoin proveniente da U è partita da I. Fortin precisa inoltre che, in caso il Master bitcoin venga combinato con un bitcoin normale e poi nuovamente diviso dopo una transazione, viene considerato come Master bitcoin la somma maggiore; se è stato diviso esattamente a metà, il titolare del Master bitcoin diviene colui che ha l'indirizzo alfanumerico minore (1abc... > 2abc).

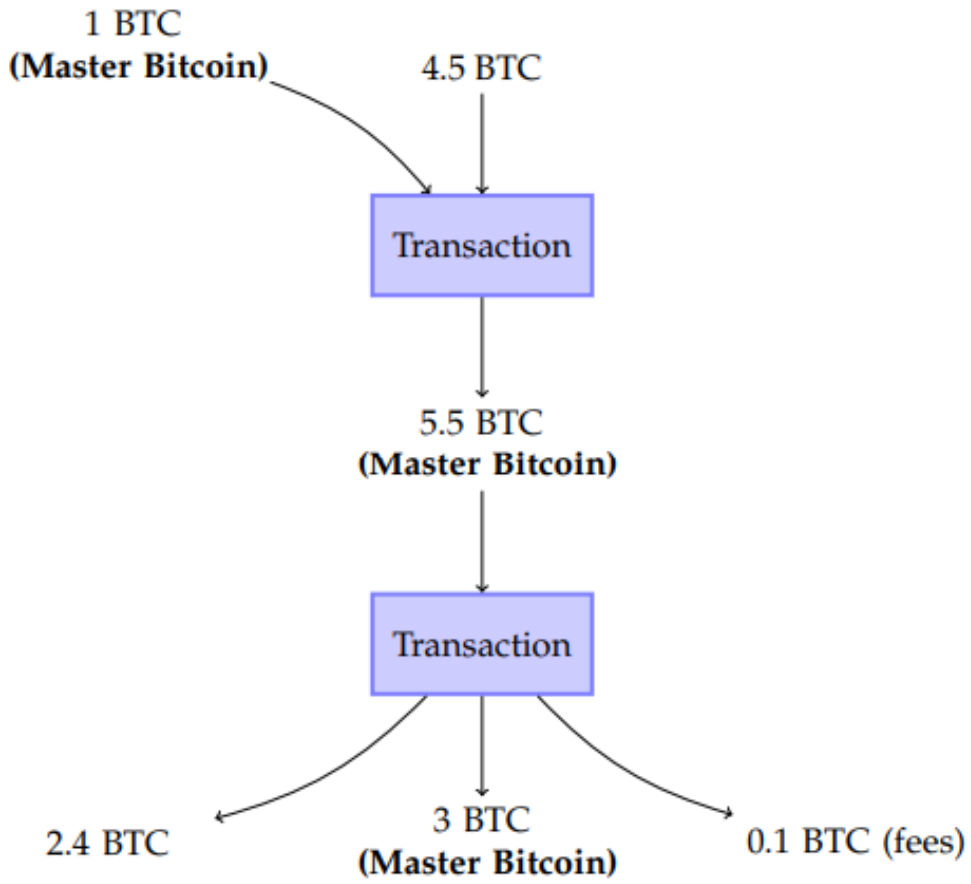

Figura 2.1: Esempio di transazioni di Master Bitcoin [33]

Herbert e Litchfield fanno presente però che il software dovrà essere in grado di leggere la storia delle transazioni, anche quando questa può essere molto lunga.

2.2 Bespoke Model

Helbert e Litchfield propongono invece la possibilità di creare una blockchain ex-novo con delle specifiche caratteristiche che permettono di validare una licenza software.

Il Bespoke Model propone di utilizzare i token per rappresentare la titolarità a una licenza: così come in Bitcoin un utente conserva nel proprio account delle criptovalute, questa nuova blockchain permette di conservare dei token. Un utente che possiede un token proveniente dall'indirizzo dell'azienda possiede una licenza e ha dunque diritto all'uso del software. Per verificare l'identità dell'utente e validare la licenza, si usa la chiave privata.

Helbert e Litchfield propongono anche di aggiungere ulteriori campi alle transazioni in modo da rendere la validazione più flessibile, oltre ad altri meccanismi come upgrades e trasferimento della licenza. Queste idee sono state poi approfondite ulteriormente nel loro successivo articolo "ReSOLV: Applying Cryptocurrency Blockchain Methods to Enable Global Cross-Platform Software License Validation", pubblicato nel 2018 [35].

2.3 Ethereum Token

Creare una nuova blockchain è molto costoso. È possibile, invece, implementare le basi dell'idea di Helbert e Litchfield con i token di Ethereum. Nel 2018 cryppadotta [36] sviluppò *dottabot*, un loro prodotto la cui gestione e vendita è interamente affidata a Ethereum. La licenza per usare *dottabot* è rappresentata come un token ERC-721, ma diversamente dal Bespoke Model, il token non è posseduto dall'utente, ma dal programma stesso: ogni software possiede una chiave privata e al momento dell'acquisto, l'utente assegnerà come titolare della licenza l'indirizzo pubblico del software. Il software, avendo una chiave privata, può interagire con il contratto, controllare se è in possesso di un token o se la licenza è scaduta o meno. Questa chiave privata deve rimanere segreta, altrimenti la licenza può essere trasferita, rubata o venduta ad altri. Se si vuole usare il programma su più PC, basta importare la chiave privata nella nuova copia del software.

2.4 Altre implementazioni

Altre aziende hanno introdotto la blockchain per la gestione totale o parziale di licenze software.

- Nel novembre 2018, l'Accenture ha inaugurato una nuova applicazione che usa tecnologia DLT per modellare e rispettare gli eventi delle licenze software durante il loro ciclo di vita, con lo scopo di facilitare funzioni di tracking, uso e auditing [37].
- La software start-up Neocor annuncia nel luglio 2020 Fusion Ledger, un sistema software cloud-based per la gestione di licenze software e prodotti tecnologi usando DLT [38].
- license.rocks offre 3 prodotti per la creazione, gestione e rappresentazione in JSON files delle licenze software. Fa uso di token ERC-1155, token per rappresentare assets fungibili e non [39].

Multiven Open Marketplace è invece un online market basata sulla blockchain dove è possibile comprare e vendere licenze software.

Capitolo 3

Analisi dei requisiti

Vogliamo creare un sistema completo dove i dati dei prodotti e delle licenze sono conservati tramite blockchain, mentre la validazione della licenza avverrà tramite chiave privata dell'utente stesso.

3.1 Requisiti funzionali

- Creare un contratto
- Mettere in pausa e non-pausa il contratto
- Limitare a certe condizioni la chiamata di certe funzioni
- Inserire e visualizzare prodotti
- Modificare prodotti
- Comprare e rinnovare una licenza tramite *wallet*
- Validare la licenza nell'applicazione
- Visualizzare informazioni essenziali del contratto: indirizzo, indirizzo del titolare del contratto, bilancio, numero totale di prodotti, numero totale di licenze vendute
- Visualizzare le informazioni della licenza: id, titolare della licenza, data d'emissione e di scadenza
- Prelevare il bilancio del contratto

3.2 Requisiti non funzionali

- Trasportare il sistema dalla testnet locale a una testnet pubblica

3.3 Funzionalità del sistema in dettaglio

- Molte funzioni sono chiamabili soltanto dal creatore del contratto (*contract owner*). Un utente diverso dall'*owner* può solo comprare, rinnovare e validare la propria licenza.

- Un prodotto ha le seguenti informazioni: id univoco, prezzo, periodo d'iscrizione, numero di prodotti venduti, disponibilità, rinnovabilità. Solo prezzo, periodo d'iscrizione e disponibilità di un prodotto sono modificabili; in particolare, la disponibilità è necessario in caso l'*owner* decida di interrompere la vendita di un certo prodotto.

- La validazione di una licenza utilizza direttamente la firma digitale dell'utente. Una licenza è valida se e solo se la chiave privata da cui proviene la richiesta combacia con la chiave pubblica titolare della licenza.

Capitolo 4

Progettazione

Il sistema che vogliamo sviluppare dovrà avere una parte back-end modellata con *smart contracts*, mentre il front-end sarà composto da un applicativo desktop, un sito web e l'applicazione vera e propria. Introduciamo anche due attori: l'azienda, che sarà l'owner del contratto, e il cliente. Per distinguere il programma per l'azienda dal software del cliente, d'ora in poi ci riferiremo al primo come *SLM* (*Software License Management*) e al secondo come *app*.

SLM e *app* sono eseguiti sul sistema operativo Windows, mentre il sito funzionerà su browser normali come Chrome, Firefox etc. La licenza verrà acquistata o rinnovata con applicazioni *wallet*, dove l'utente autorizzerà la transazione direttamente usando l'interfaccia browser. Inoltre assumiamo che l'azienda e l'utente abbiano già effettuato l'autenticazione nei rispettivi programmi.

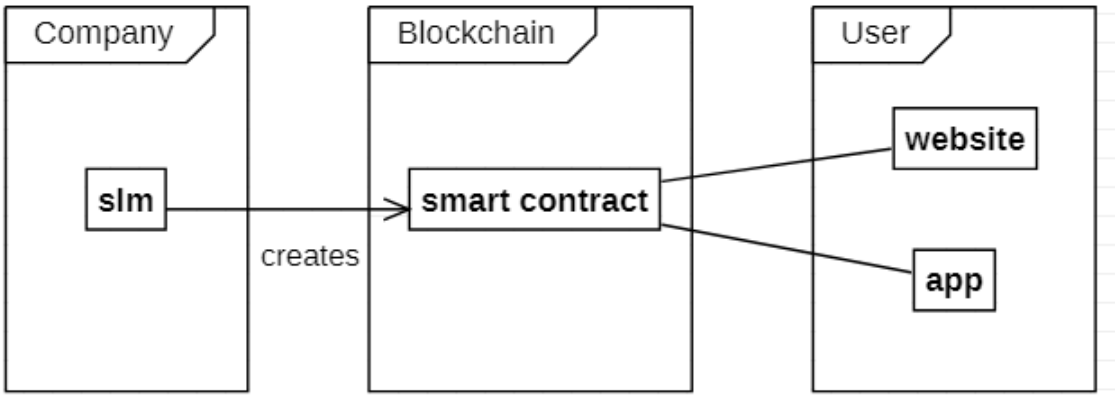

Figura 4.1: Schema dell'intero sistema

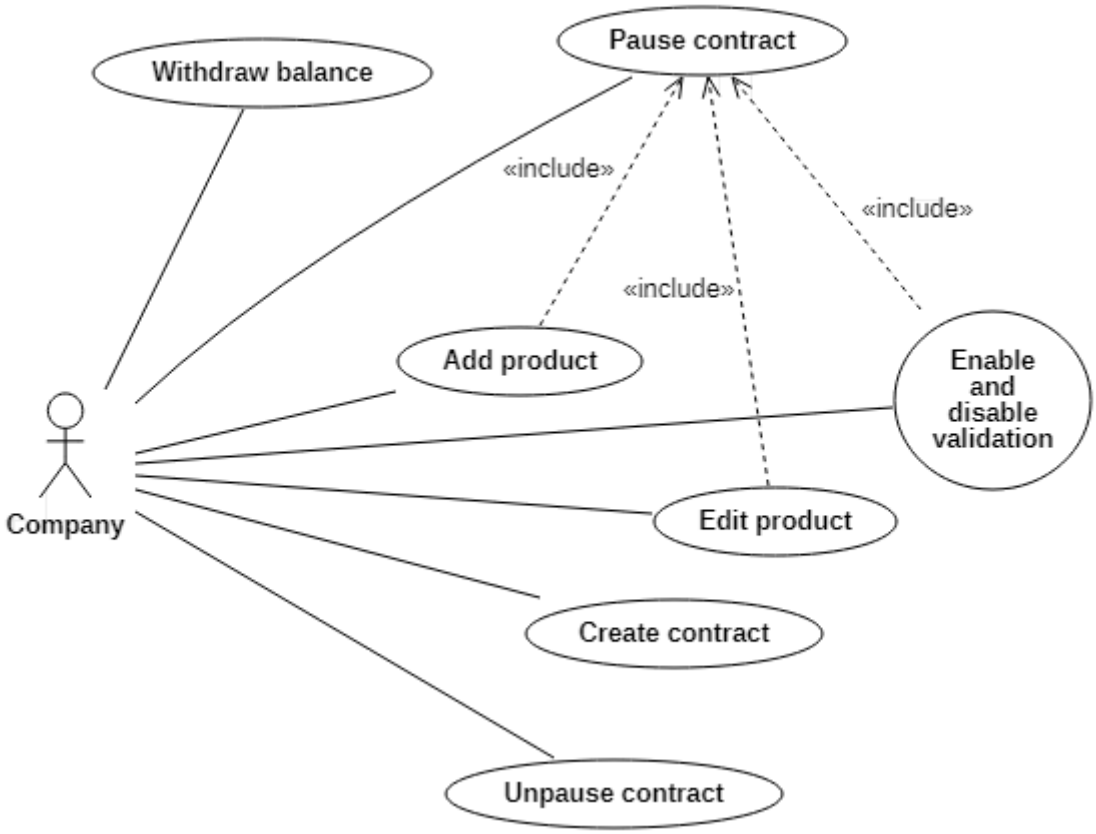

Figura 4.2: Use case - lato azienda

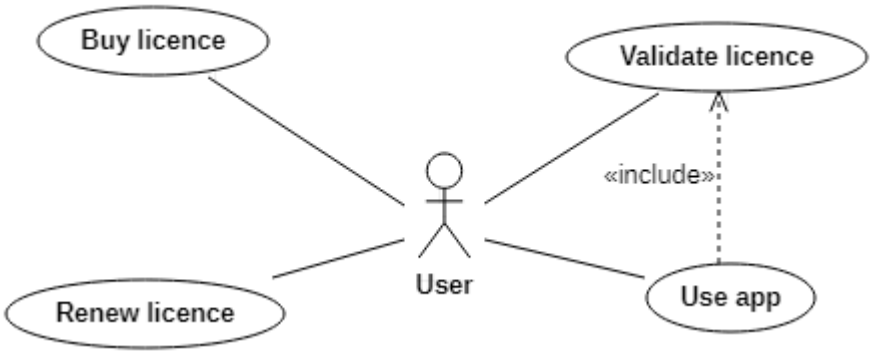

Figura 4.3: Use case - lato utente

4.1 Smart Contract

Lo *smart contract* contiene tutta la logica del sistema. Ha tutte le funzioni necessarie per creare e modificare un prodotto; comprare, rinnovare e validare una licenza; limitare l'uso di certe funzioni a seconda dell'utente che la chiama. Tutti e tre i sottosistemi dovranno avere un riferimento a questo contratto, in modo da usare le varie funzionalità senza usare altri programmi.

4.1.1 ERC721

L'ERC721 è il token scelto per rappresentare le licenze software in quanto una licenza è univoca e indivisibile. Per essere considerato tale, il contratto deve implementare le seguenti funzioni:

- **balanceOf**(address owner): ritorna il numero totale di token posseduto da *owner*
- **ownerOf**(uint256 tokenId): ritorna l'indirizzo che possiede il token con id *tokenId*
- **safeTransferFrom**(address from, address to, uint256 tokenId, bytes data): trasferisce il token *tokenId* da *from* a *to*; opzionalmente, si posso aggiungere dei dati
- **safeTransferFrom**(address from, address to, uint256 tokenId)
- **transferFrom**(address from, address to, uint256 tokenId): a differenza delle due funzioni safeTransferFrom, transferFrom non controlla se il destinatario *to* è in grado di ricevere token ERC-721; sarà responsabilità dell'utente controllare che *to* può ricevere token
- **approve**(address approved, uint256 tokenId): approva l'indirizzo *approved* a gestire il token *tokenId*
- **setApprovalForAll**(address operator, bool approved): approva l'indirizzo *operator* a gestire tutti gli asset dell'utente che ha chiamato questa funzione
- **getApproved**(uint256 tokenId): ritorna l'indirizzo approvato per il token *tokenId*
- **isApprovedForAll**(address owner, address operator): ritorna se l'indirizzo *operator* è approvato per gestire i token di *owner*

Non essendo oggetto della tesi, le licenze al momento non sono trasferibili. Attualmente, non esistono standard ufficiali per token che non possono essere trasferiti, anche se nel 2018 sulla repository ufficiale di Ethereum è stato aperto un issue per la possibile creazione di un *ERC1238* per questo tipo di token [40]. Ad oggi però non ci sono stati ulteriori sviluppi.

Si è deciso quindi di implementare comunque ERC721, ma disabilitando le funzioni di trasferimento; in questo modo i token degli utenti sono facilmente rintracciabili da vari tools come *blockchain explorer*.

4.1.2 SelfDestruct

Dobbiamo decidere se implementare o meno la funzione *Selfdestruct*. Nell'articolo "Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum" [41], è stata condotta un'indagine dove sono emersi i seguenti motivi a favore di questa funzione:

- Eliminare il contratto quando presenta errori che possono comprometterne la sua sicurezza; una volta che sono stati risolti, è possibile fare il deploy di un nuovo contratto.

- È utile per pulire l'environment di Ethereum quando si pensa di non usare più il contratto.
- La funzione permette di specificare un indirizzo a cui inviare tutti gli Ether conservati in un contratto; usare *SelfDestruct* come metodo per prelevare il bilancio del contratto può risultare vantaggioso in quanto consuma meno gas rispetto a una tradizionale funzione *withdraw*.

Allo stesso tempo, gli autori dell'articolo fanno presente che:

- *Selfdestruct* può essere motivo di preoccupazione dal momento che aggiunge un ulteriore livello di complessità al codice.
- La funzione può anche essere causa di "sfiducia" in certe applicazioni: supponiamo di avere una dapp che simula una scommessa, dove gli utenti inviano 1 ETH al contratto; dopo che è stata raggiunta una certa somma, viene scelto casualmente un vincitore che riceverà tutti gli Ether inviati. In un caso come questo, gli utenti potrebbero temere che l'*owner* del contratto possa trasferire i soldi ad ogni momento usando *Selfdestruct*.
- La distruzione di un contratto va contro il concetto d'immutabilità della blockchain.
- Se gli utenti non sono informati in tempo dell'eliminazione del contratto, questi possono inviare erroneamente Ether, subendo una perdita.

Per minimizzare gli svantaggi di questa funzione, gli autori hanno suggerito come possibile alternativa, o come opzione usata insieme a *Selfdestruct*, l'aggiunta di uno stato al contratto, che abilita e disabilita le funzioni più critiche come quelle che gestiscono Ether. È stato quindi deciso di non implementare questa funzione; in più, nel nostro caso, non cancellare il contratto può permettere agli utenti di continuare ad usare un software anche quando questo non è più in vendita.

```
function deleteContract() public onlyOwner {
  selfdestruct(ownerAddress);
  /*because we specified the owner address, at the destruction of the
   contract, all Ether will be send to them*/
}
```

4.1.3 Stati di un contratto

Al posto di *SelfDestruct* usiamo delle variabili "stato" che controllano che l'esecuzione di certe funzioni avvenga solo in certe condizioni. La possibilità di cambiare gli "stati" verrà data solo all'*owner* del contratto, ma potrebbe essere estesa a più utenti fidati. Nel contratto useremo i seguenti stati:

- **Contratto in pausa e non in pausa**: alla creazione del contratto, il contratto è in pausa. In questo stato, l'*owner* può aggiungere prodotti, cambiarne il prezzo, il periodo d'iscrizione o la disponibilità. Solo quando il contratto non è più in pausa, gli utenti sono in grado di comprare e rinnovare una licenza, mentre le funzioni di aggiunta e modifica dei prodotti sono disabilitate.

- **Validazione abilitata**: abilitare e disabilitare la validazione di una licenza.

Quando l'azienda mette in pausa il contratto per modificare i prodotti, l'utente deve essere comunque in grado di validare la licenza e usare il software. Se necessario però, bisogna avere la possibilità di disabilitare la validazione delle licenze. Questi stati, infatti, svolgono anche un ruolo di *Circuit Breaker*. Il *Circuit Breaker*, detto anche *Emergency Stop* [42], è un pattern dove si aggiunge una funzionalità per fermare l'esecuzione di funzioni critiche o dell'intero contratto in caso di bug e altre vulnerabilità; mettendo il contratto in pausa e disabilitando la validazione, blocchiamo il contratto in modo definitivo, in quanto l'unico utente in grado di interagire con esso e modificare il suo stato è l'*owner*, ovvero l'azienda.

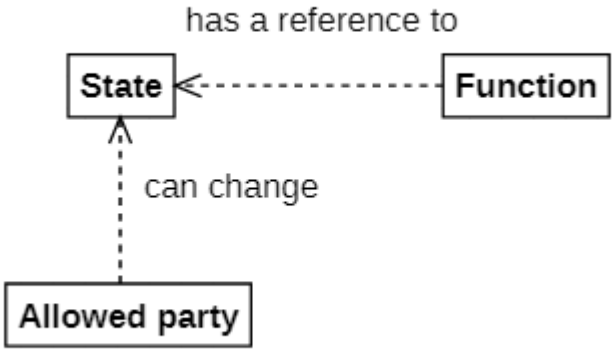

Figura 4.4: Diagramma di funzionamento di uno stato

4.2 SLM

Al primo avvio di *SLM*, l'azienda deve avere la possibilità di creare un nuovo contratto o di caricarne uno già in suo possesso; le mostreremo quindi una schermata iniziale con queste due opzioni. Fatto questo, l'azienda ha a disposizione due *view*, in una i prodotti che vuole mettere in vendita, nell'altra le informazioni generali sul contratto.

Dal momento che i prodotti hanno pochi dettagli, possiamo mostrare tutte le informazioni con una semplice tabella. Le informazioni sono:

- id univoco
- prezzo in ETH
- periodo di iscrizione
- numero di prodotti venduti
- se il prodotto è disponibile o meno alla vendita
- se la licenza per quel prodotto è rinnovabile o meno

In questa pagine l'azienda avrà la possibilità di aggiungere o modificare un prodotto. Nella *view* del contratto, l'azienda potrà visualizzare:

- l'indirizzo del contratto
- il proprio indirizzo
- se il contratto è in pausa o meno
- se è possibile validare la licenza
- il bilancio
- il numero totale dei prodotti
- il numero totale delle licenze vendute

SLM inoltre dovrà fornire all'utente la possibilità di cambiare lo stato del contratto, abilitare e disabilitare la validazione delle licenze e prelevare il bilancio.

Una volta che un'azienda ha creato uno o più prodotti, può mettere il contratto in *non-pausa*, permettendo agli utenti l'acquisto delle licenze.

4.2.1 Caricamento di *SLM*

Dopo che l'azienda ha creato per la prima volta un contratto, si salva il suo indirizzo in locale per semplificare il caricamento di *SLM*: in questa maniera, invece di chiedere ogni volta all'azienda di inserire manualmente l'indirizzo del contratto a cui vuole accedere, sarà *SLM* stesso a caricarlo automaticamente. All'avvio del programma, questo controlla l'esistenza di questo indirizzo: se non esiste, verrà mostrata la schermata iniziale; altrimenti, il programma leggerà l'indirizzo e mostrerà la schermata dell'inventario dei prodotti. Non avendo particolari esigenze, l'indirizzo del contratto è salvato in un file di testo.

4.2.2 Periodo di iscrizione

Dovendo verificare il corretto funzionamento dell'*app* quando scade la licenza, per ridurre i tempi di attesa, i periodi di iscrizione sono in minuti. In una versione ufficiale si potrebbero usare periodi mensili.

4.2.3 Comportamento sincrono

Quando si effettuano operazioni in Ethereum, non è possibile conoscere subito l'esito delle transazioni. L'azienda non può aggiungere o modificare un prodotto e poi mettere il contratto in *non-pausa*, senza essere sicura che la transazione precedente abbia avuto successo. *SLM* quindi avrà un comportamento strettamente sincrono, con un preciso insieme di azioni che l'azienda può fare a seconda dello stato corrente del contratto; ogni transazione richiederà un tempo di attesa durante la quale *SLM* verrà temporaneamente disabilitato.

Paused contract	Unpaused contract
Add a product Change price Change subscription period Change availability Withdraw balance Pause contract Disable or allow license validation	Pause contract Withdraw balance

Figura 4.5: Le possibili azioni che può fare l'azienda a seconda dello stato del contratto

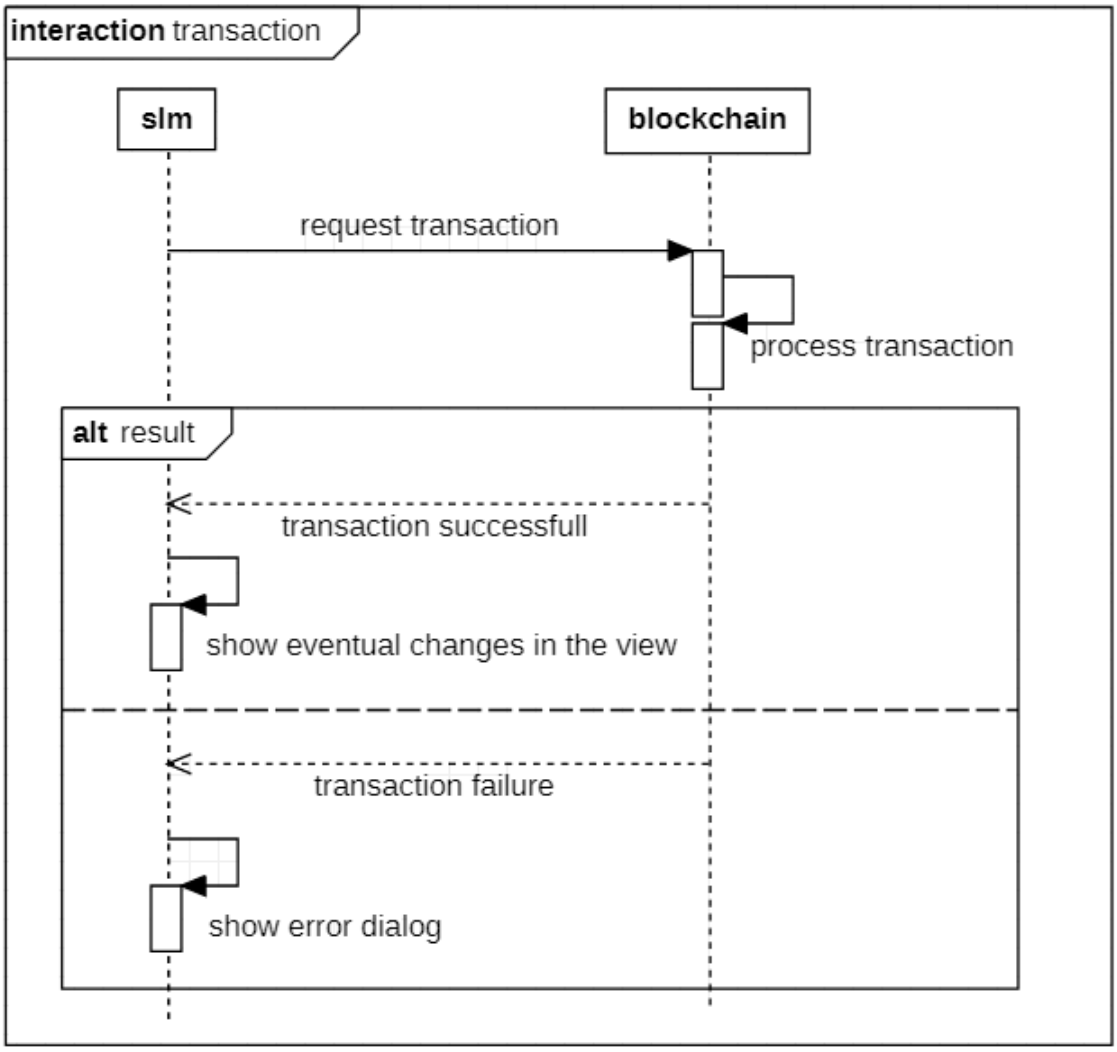

Figura 4.6: Diagramma di sequenza di una transazione generica

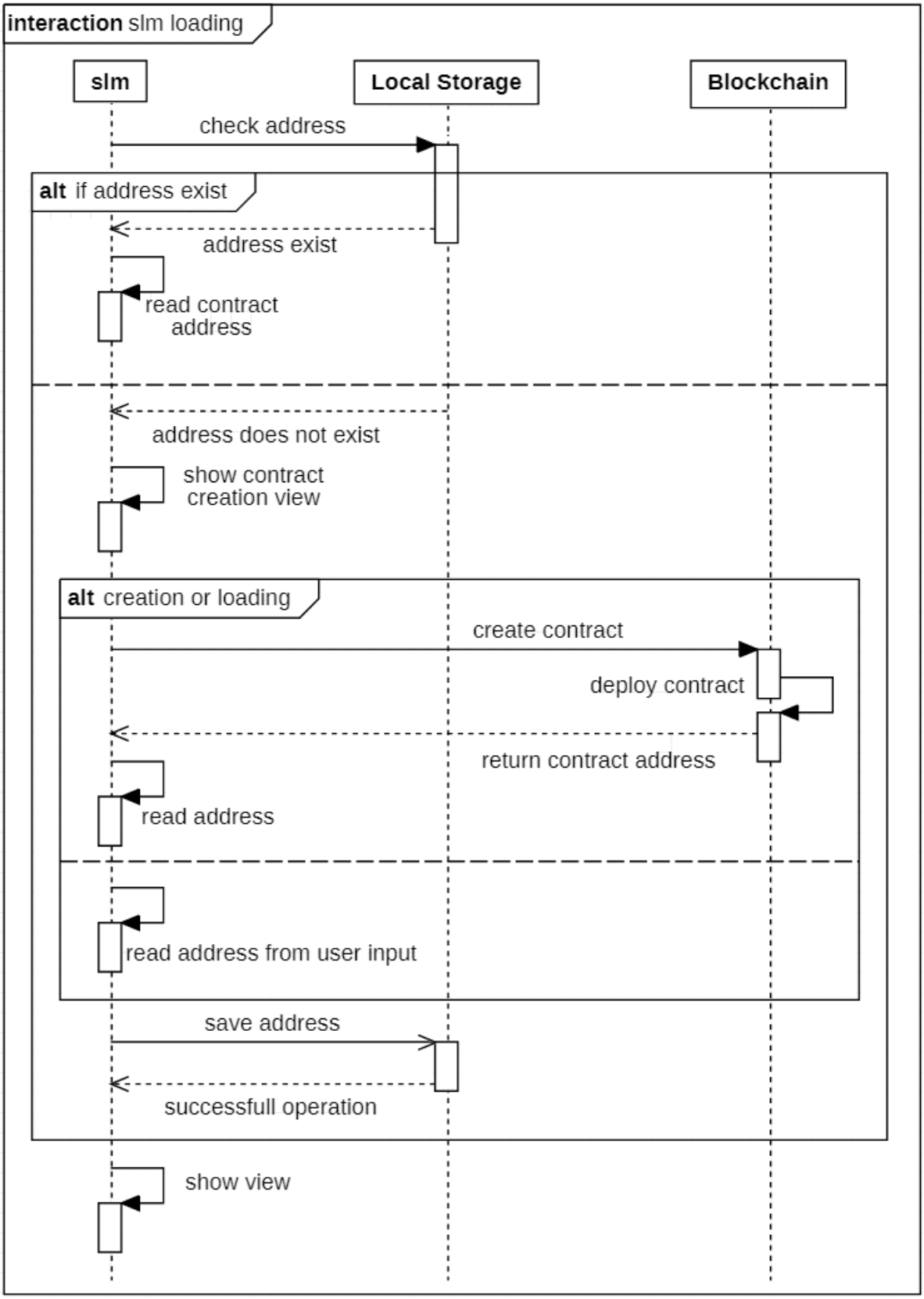

Figura 4.7: Diagramma di sequenza del loading iniziale di *SLM*

4.3 Website

Una volta che il contratto è pronto all'uso, un cliente interessato utilizza il sito web precedentemente preparato dall'azienda. Nel prototipo sviluppato per questa tesi, per semplicità il sito sarà una sola pagina web, con una breve presentazione di un prodotto immaginario seguito da due bottoni, *buy* e *renew*. Cliccando su uno di questi, è richiesto al cliente di connettersi al *wallet* se non si è già autenticato o di scaricarne uno se non è stato rilevato nessuno; allo stesso tempo viene mostrata una *form* dove il cliente inserirà le informazioni necessarie per acquistare o rinnovare una licenza e visualizzare il costo totale. Se gli input sono corretti, il cliente clicca sul bottone *proceed*; apparirà quindi un'interfaccia con le informazioni della transazione di Ethereum. Se l'utente ha intenzione di procedere, attenderà la validazione della transazione, alla fine della quale il cliente sarà informato dell'id della sua licenza o il successo del rinnovo. In tutti gli altri casi, verrà mostrato un messaggio di errore.

La licenza che viene registrata nella blockchain contiene le seguenti informazioni:

- Id
- Id del prodotto
- Indirizzo del cliente
- Data di emissione
- Data di scadenza

4.4 App

L'*app* sarà una *toy application* composta da un'area di testo che abiliteremo e disabiliteremo a seconda della presenza o meno di una licenza valida. L'*app* permette all'utente di inserire la propria licenza e visualizzarne le seguenti informazioni: id, indirizzo del cliente, data di emissione, data di scadenza. L'id del prodotto, invece, verrà confrontato con l'id integrato nell'*app*: in questo modo l'azienda può mettere in vendita più prodotti, anche molto diversi, e la licenza sarà valida solo per i software che hanno lo stesso id compreso nella licenza.

4.4.1 Comportamento

Anche l'*app* avrà un comportamento sincrono come *SLM*, in quanto l'utente attenderà il completamento della validazione prima di usufruire del software. Similmente, anche l'id della licenza viene salvato in un file di testo per risparmiare il cliente la necessità di doverlo inserire manualmente tutte le volte.

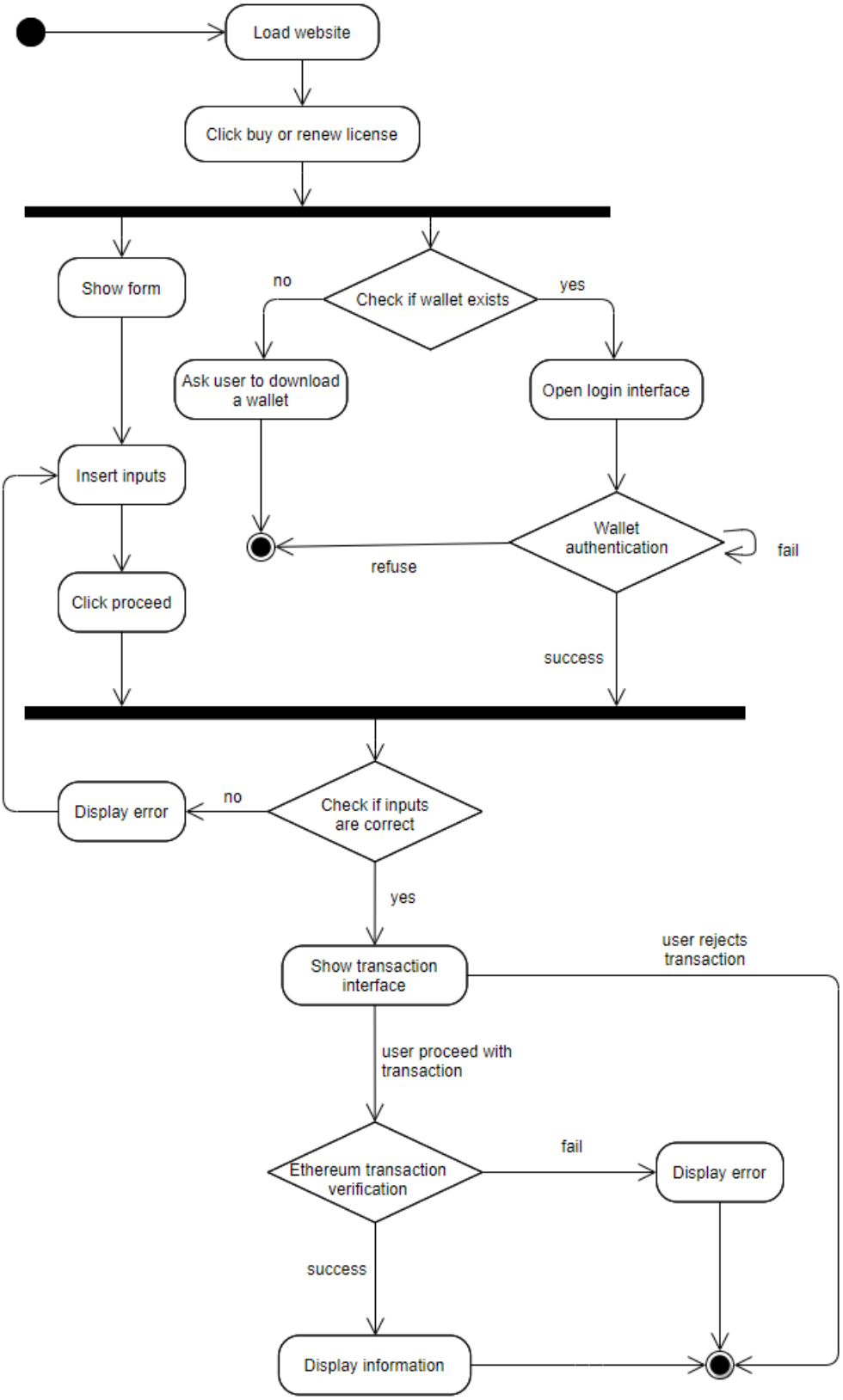

Figura 4.8: Diagramma di attività del sito web

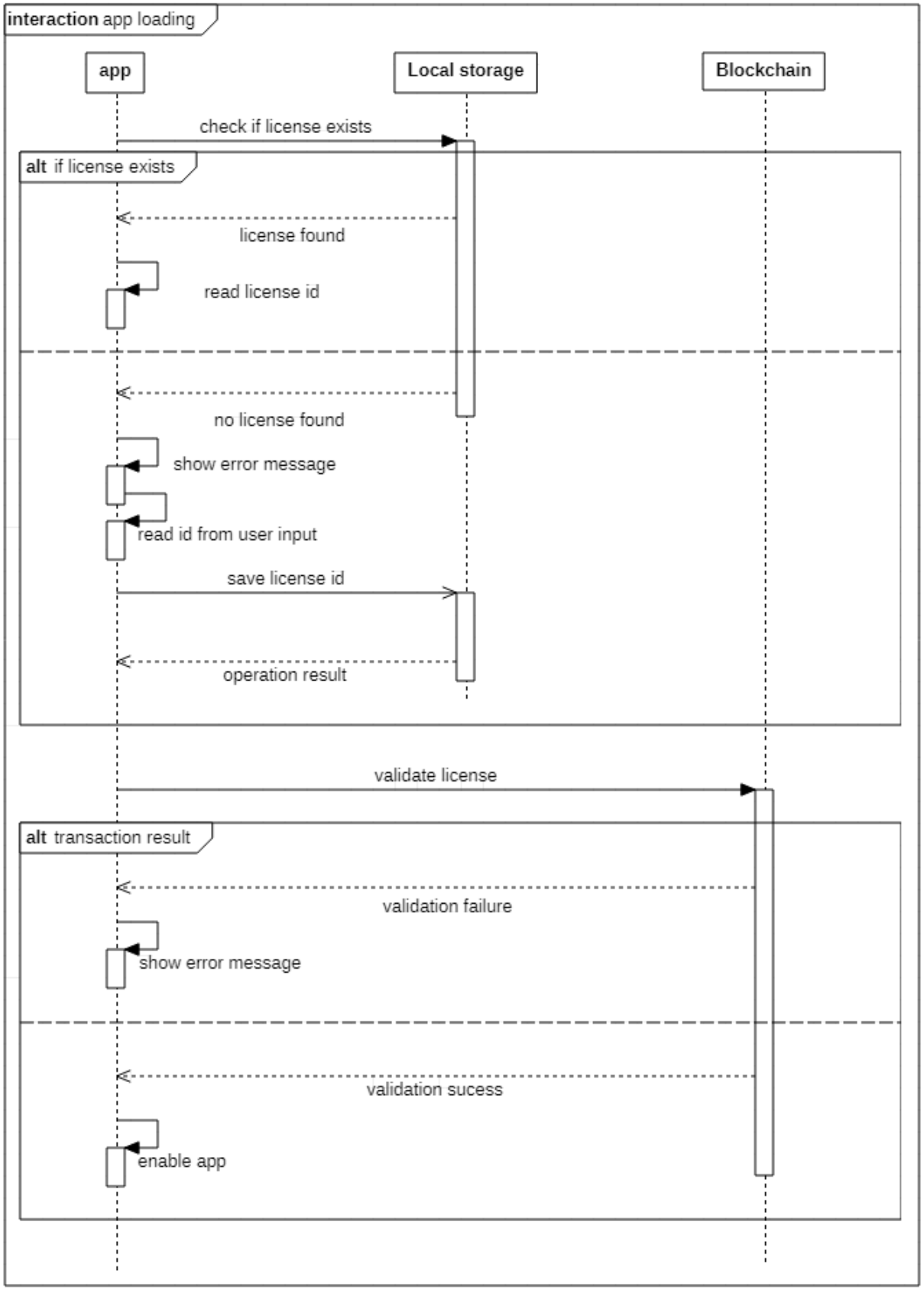

Figura 4.9: Diagramma di sequenza del loading di app

Capitolo 5

Tecnologie

5.1 Testnet, blockchain explorer e clients

Gli *smart contract* e le dapp possono essere testate su testnet locali, create con un client, o su testnet pubbliche che simulano il funzionamento di Ethereum. In questo prototipo abbiamo usato **Ganache** [43], un *client* che crea una blockchain locale e fornisce 10 account con le loro chiavi private e pubbliche, ognuno con un bilancio di 100 ETH. Per visualizzare i costi delle transazioni, abbiamo anche fatto il deploy del contratto su una testnet pubblica: molte testnet pubbliche usano un meccanismo di consenso diverso da quello di Ethereum; è stata quindi utilizzata la testnet di **Ropsten**, in quanto fa uso di *Proof of Work* ed è attualmente quella più simile a Ethereum [44]. Invece di un client, per connetterci a *Ropsten* è stato utilizzato Infura [45]: dopo essersi registrati sul sito, bisogna creare un "progetto", provvisto di un id; ogni progetto ha un *URL* con la quale possiamo connetterci alla blockchain di Ethereum o una delle testnet pubbliche. Un URL di Infura ha il seguente formato:

```
https://ropsten.infura.io/v3/[Project-id]
```

La *blockchain explorer* **Etherscan** [46] è servita per ispezionare la blockchain di Ropsten. Una *blockchain explorer* permette agli utenti di cercare facilmente le transizioni che avvengono nella blockchain, oltre a visualizzare il bilancio e la storia delle transazioni di EOAs e contratti.

5.2 Smart contract

Per lo sviluppo degli dei contratti sono stati impiegati Truffle, Ganache, Solidity e Visual Studio Code.

Truffle [47] è un framework molto usato per lo sviluppo su Ethereum che facilita la gestione, la compilazione e il deployment dei contratti. È stato usato insieme a Ganache per effettuare gli *unit tests* dei contratti.

Solidity [16] è uno dei linguaggi più usati per lo sviluppo di *smart contracts*. È un linguaggio *object-oriented*, per cui un contratto è strutturato in modo molto simile alle classi: ha un costruttore, può contenere variabili e funzioni; supporta ereditarietà, librerie e interfacce e tipi di dato complessi.

```
//Extract of a contract
contract AccessControl {

    address payable public contractOwner;

    function withdraw() external onlyOwner() {
        (bool success,) = contractOwner.call{value:
            address(this).balance}("");
        require(success, "Transfer failure");
    }

    function contractBalance()
        external
        view
        returns (uint256 balance)
    {
        return address(this).balance;
    }
}
```

5.3 Website

La logica interna del sito web è stato creato con **HTML**, **JavaScript** e **JQuery**, una libreria JavaScript molta usata perché facilita la minipolazione di HTML/DOM, CSS e la realizzazione di animazioni, mentre per l'interfaccia è stato usato **CSS**; come browser è stato usato Google Chrome.

Per interagire con gli *smart contract* è stata usata la libreria JavaScript **Web3.js** [48], mentre l'estensione browser **MetaMask** [49] farà da *wallet* per eseguire le transazioni necessarie. Per eseguire il sito web, è stato usato il pacchetto "live-server" di **Node.js**, creando un server local a cui MetaMask può connettersi.

5.4 Applicativi

Gli applicativi sono stati sviluppati in **C#**. Per interagire con la blockchain è stato usato **Nethereum** [50], una libreria open source che semplifica molto il deployment e le chiamate delle funzioni dei contratti.

Non avendo particolari esigenze per il front-end, è stata usata la piattaforma **UWP**, Universal Windows Platform.

Capitolo 6

Implementazione

In questa capitolo si discutono alcune scelte implementative che sono state fatte durante lo sviluppo del sistema. Inoltre, si mostreranno certe particolarità delle librerie e dei linguaggi utilizzati.

6.1 Smart Contract

Secondo la documentazione di Solidity [51], per minimizzare errori è consigliato scrivere contratti piccoli, modulari e facilmente comprensibili. Complessivamente il contratto finale è composto da 5 contratti:

- AccessControl: istruzioni necessarie per limitare la chiamata di certe funzioni
- Inventory: creare, modificare e ritornare le informazioni di un prodotto
- Token: creare e trasferire un token ERC721
- License: creare, rinnovare, validare e ritornare le informazioni di una licenza
- SoftwareManagement: costruttore dell'intero cotratto

Solidity supporta l'ereditarietà multipla. Un contratto può ereditare da un altro usando la keyword *is*; quando si compila il contratto "figlio", viene creato un unico contratto, contenente tutte le funzioni e le variabili.

Per essere conforme allo standard ERC721, il contratto implementa le interfacce **ERC721** e **ERC165**. ERC165 è un'interfaccia standard per ritornare le interfacce implementate di un contratto. In più, è stata usata **ERC721TokenReceiver**, un'interfaccia per verificare che, in caso di trasferimento di token a un contratto, questo possa ricevere token ERC721.

Sono state utilizzate anche le librerie *SafeMath* e *Address*: la prima serve per svolgere operazioni aritmetiche integrate con un controllo dell'*overflow*, in quanto Solidity non provoca eccezioni per questi errori; la seconda è stata utilizzata per verificare se un certo indirizzo pubblico è un contratto.

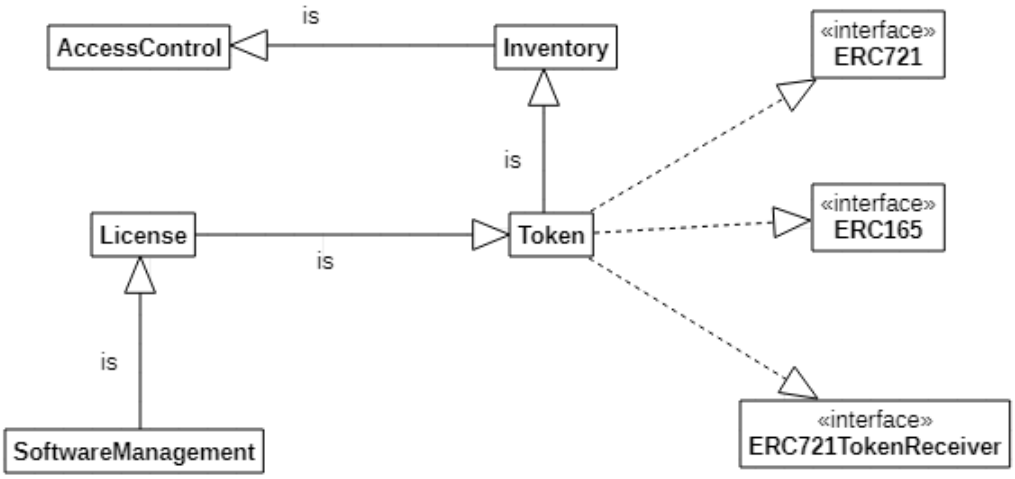

Figura 6.1: Diagramma del contratto

6.1.1 Withdraw

Per prelevare il bilancio di uno smart contract lo sviluppatore deve implementare una funzione. Le tre diverse funzioni per trasferire gli Ether da un account all'altro sono **send**, **transfer**, **call.value** e la loro differenza sta nella quantità di gas usato e la gestione delle eccezioni:

- Gas: l'account che chiama le funzioni *send* e *transfer* spende solo 2300 gas, mentre *call.value* usa tutto il gas
- Eccezioni: *send* e *call.value* ritornano *false* in caso di fallimento, mentre *transfer* lancia un'eccezione

```
1    address.send(amount);
2
3    address.transfer(amount);
4
5    address.call{value: amount}("");
```

Il limite di gas permette di evitare *reentrancy attacks* [51], un attacco che può avvenire ogni volta che un contratto chiama funzioni poche sicure di altri contratti. Per esempio, quando un contratto B interagisce con un contratto A per trasferire Ether, il controllo ricade sul contratto chiamante, ovvero B; se non sono state inserite le giuste misure, B può richiamare la funzione di *withdraw* di A senza che questa termini e potenzialmente prelevare tutto il bilancio di A. Qualunque trasferimento di Ether a un contratto, infatti, esegue una *fallback function*; questa può essere implementata inserendo codice che porta all'esecuzione di un *reentrancy attack*. Affinché B non chiami *withdraw* ricorsivamente, fino al 2019 è stato consigliato di usare le funzioni *send* e *transfer*, in quanto una quantità di gas pari a 2300 non è sufficiente per effettuare un attacco simile.

```
1
2 //example of reentrancy attack
3 contract ReceiveEther {
4
5     function () public payable {
6         /*malicious code to carry out the attack, like calling again
7         sendEther*/
8     }
9 }
10
11
12 contract SendEther {
13
14     function sendEther() public payable {
15         (bool success,) = msg.sender.call{value:
16             balance[msg.sender]}("");
17         /*calling sendEther again before the following lines are
18         executed allows the multiple retrievals of the senders' balance
19         since balance[msg.sender] is not yet set to 0*/
20         if (success)
21             balance[msg.sender] = 0;
22     }
23 }
```

Le cose sono cambiate quando nel dicembre 2019 è stato effettuato il cosiddetto "Hardfork di Istanbul" [52]. Tra i vari cambiamenti effettuati al protocollo di Ethereum, questo hardfork può rendere alcune istruzioni, come il trasferimento di Ether, più costose in termini di gas. Ciò potrebbe causare funzioni come *send* e *transfer* ad incorrere in *out-of-gas exception*. Dal momento che non è più possibile affidarci a un prezzo stabile di gas, per evitare questo problema si consiglia di non usare queste due funzioni [53] e cercare di usare altre misure per prevenire *reentrancy attacks*.

Tuttavia non ci sono informazioni riguardo eventuali problemi quando *send* e *transfer* sono chiamate da un EOA. Alla fine è stata usata la funzione *call.value*, nell'evenienza che le altre due funzioni causino *out-of-gas exception*.

```
1 function withdraw() external onlyOwner() {
2     (bool success,) = contractOwner.call{value:
3         address(this).balance}("");
4     require(success, "Transfer failure");
5 }
```

6.1.2 Uso di block.timestamp

La validazione di una licenza avviene per mezzo della funzione "costante" *validateLicense(licenseId, productId)*, ovvero una funzione che non modifica lo stato del contratto. Questa funzione verifica le seguenti condizioni:

- *licenseId* deve essere un id esistente
- *productId* del software deve combaciare con l'id associato alla licenza

- l'utente che chiama questa funzione deve avere una chiave privata che combacia con la chiave pubblica registrata nella blockchain e titolare della licenza
- la data di scadenza della licenza deve essere antecedente alla data attuale

Per controllare che la licenza non sia scaduta si è scelto di usare *block.timestamp*, una variabile globale che ritorna il *timestamp* del blocco attuale in *Unix epoch time*, ovvero il numero di secondi dal 1 gennaio 1970. È una variabile poco consigliata per accertare brevi scadenze in quanto è possibile per un *miner* manipolare il *timestamp* di un blocco di pochi secondi – ricordiamo che il *timestamp* è settato dal miner [54]; inoltre, una funzione "costante", non modificando lo stato della blockchain, non crea nessun blocco, ma ritorna il *timestamp* dell'ultimo blocco e di conseguenza una data che non rispecchia perfettamente quella attuale [1]. Nel nostro caso però, l'uso di questa funzione è accettabile per i seguenti motivi:

1. Dal momento che i periodi di iscrizione finali saranno in mesi, non è necessario che le date siano perfettamente accurate fino al secondo.
2. La blockchain di Ethereum è in continua crescita e possiamo quindi assumere che tra uno o più mesi esisterà un blocco il cui *timestamp* è successivo alla data di scadenza della licenza.
3. Il protocollo di Ethereum in sé non invalida un blocco con un *timestamp* molto lontano nel futuro, ma controlla solo che sia maggiore del blocco parente [12]; lavorando con una network peer-to-peer però, gli altri nodi potrebbero rifiutare un nodo con una data molto lontana a favore di date più vicine.

6.1.3 Limitazione delle funzioni

Solidity possiede dei costrutti particolari con la quale è possibile richiedere che la funzione verifichi certe condizioni prima di essere eseguita, lanciando eccezioni se queste non sono rispettate.

Alcune funzioni devono controllare la presenza di certi requisiti. Per esempio, non è possibile aggiungere un prodotto il cui id è già esistente nella blockchain. Per questi vincoli è stato usato *require*, una funzione che valuta un parametro *bool* e se questo risulta falso, genera una eccezione con il relativo messaggio di errore, annullando la transazione e gli eventuali cambiamenti che sono stati fatti.

```
1 function _createProduct(uint256 _productId, uint256 _price, uint256
    _subTime)
2     internal
3 {
4     require(!_doesProductExist(_productId),
5         "The product already exists.");
```

[1] Non esistono documentazioni ufficiali su cosa ritorna block.timestamp in funzioni constanti; dopo un issue su Github però, è stato appurato con test manuali che client ufficiali come Geth e OpenParity ritornano il timestamp dell'ultimo blocco. Ganache è stato dunque modificato per accordarsi con il comportamento degli altri client.

```
    require(_price > 0, "The price must be more than 0");

    ...
}
```

Altre funzioni invece sono limitate da certi vincoli: per esempio, solo l'owner del contratto può prelevare il bilancio totale del contratto o è possibile fare un acquisto solo quando il contratto non è in pausa. Per questi vincoli, sono stati usati i *function modifiers*.

```
/*The function can be called only by the owner*/
modifier onlyOwner() {
    require(msg.sender == contractOwner, "You are not authorised to use
    this function.");
    _;
}

/*only owner should be able to pause and unpause contract*/
function togglePause() external onlyOwner() {
    isPaused = !isPaused;
}
```

L'underscore "_" indica dove deve essere inserito il corpo della funzione a cui è stato aggiunto il *modifiers*. Alla stessa funzione, possono essere applicati più *modifiers*.

require e *modifiers* svolgono funzioni molto simili; l'uso di uno rispetto a un altro dipende dallo sviluppatore, considerando fattori come frequenza della condizione e leggibilità.

6.2 Applicativi

Le due applicazioni desktop *slm* e *app* sono state realizzate in C# usando il *Model-View-ViewModel* [55].

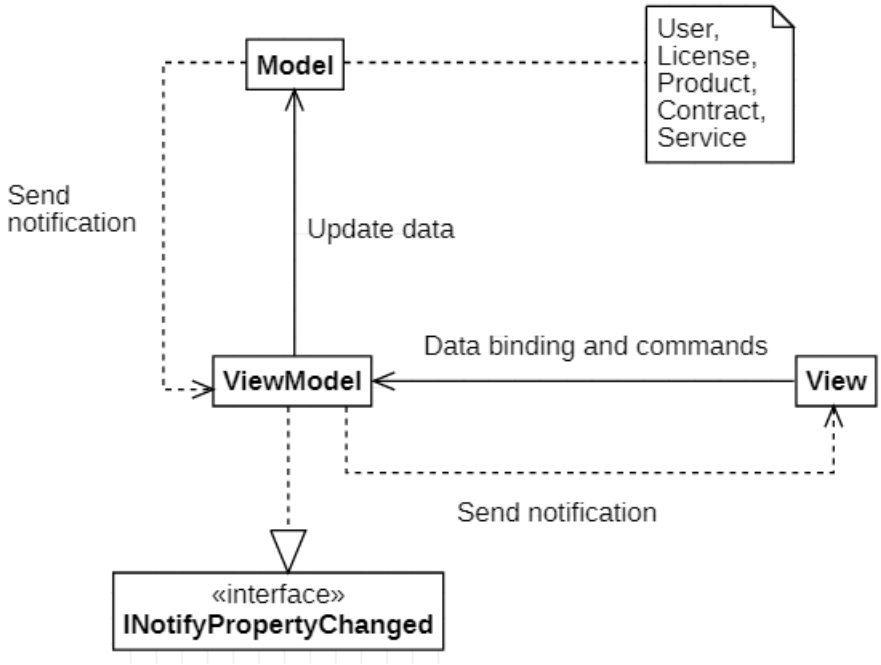

Figura 6.2: Model-View-ViewModel pattern

Il *Model-View-ViewModel* è un pattern architetturale che serve a facilitare la separazione della *view*, l'interfaccia grafica dell'utente, dal *model*, i dati e la logica del progetto. Il *view model*, invece, implementa le funzioni e le proprietà a cui sono associati i dati visualizzati ("data binding"), notificando la *view* di eventuali cambiamenti nel *model* per mezzo dell'interfaccia *INotifytPropertyChanged*; il *view model*, inoltre, svolge il ruolo di intermediario tra la view e le classi del model con cui vuole interagire.

6.2.1 Nethereum

Account

Per comunicare con la blockchain Nethereum provvede due tipi di oggetto account, *Account* e *ManagedAccount*. Queste sono classi che rappresentano un account Ethereum: hanno infatti due campi, *PrivateKey* e *PublicKey* e un *TransactionManager*, una classe con le funzioni necessarie per firmare e inviare transazioni.

Account è una classe che prende come parametro la chiave privata, in chiaro o decriptandola da un *keystore*; permette la comunicazione con la blockchain senza installare un client, in quanto le transazioni vengono firmate da *TransactionManager*. Si presta molto a essere usato con API come Infura.

ManagedAccount, invece, prende come parametri l'indirizzo pubblico e l'eventuale password con cui è stata criptata la chiave privata. *ManagedAccount* è gestito dal *client* installato sul PC: la transazione verrà creata dal *TransactionManager*, che la invierà al *client* per essere firmata.

```
1 Account account = Account.LoadFromKeyStoreFile(keystoreFilePath,
    password);
```

```
account = Account.LoadFromKeyStore(keystorefile, password);
account = Account("privateKey");

ManagedAccount account = ManagedAccount("publicKey", "password");
```

Sono stati utilizzati entrambi i tipi di account: *ManagedAccount* è stato utilizzato per eseguire i test sulla blockchain locale di *Ganache*, mentre *Account* è stato usato con la testnet pubblica di Ropsten.

Web3

Una volta creato un account, è necessario creare un oggetto *Web3*, la classe che racchiude i metodi forniti dai *client* Ethereum per interagire con i contratti. *Web3* prende come parametro un oggetto *Account* o *MenagedAccount* e l'URL a cui connettersi per comunicare con la blockchain: di default, l'URL è settato a *http://localhost:8545*, altrimenti è possibile specificare un nuovo indirizzo.

```
Web3 web3 = new Web3(account); /*implied http://localhost:8545*/

Web3 web3 = new Web3(account,
    "https://ropsten.infura.io/v3/[Project-id]");

//for example, we can use TransactionManager, the class part of Account
   /ManagedAccount, to send a transaction
await web3.TransactionManager.SendTransactionAsync(account.Address,
   addressTo, amount);
```

Service e Definitions

Il plugin di Visual Studio Code utilizzato per Solidity è stato sviluppato dallo stesso creatore di *Nethereum*, Juan Blanco. In questo plugin è integrata la compilazione e la generazione del contratto in C#, creando due file: *[ContractName]*Definition.cs e *[ContractName]*Service.cs.

SoftwareManagementDefinition contiene il bytecode del contratto *SoftwareManagement.sol* e la definizione di tutte le sue funzioni.

```
/*Excerpt of SoftwareManagementDefinitions.cs*/
public partial class SoftwareManagementDeployment :
   SoftwareManagementDeploymentBase
{
    public SoftwareManagementDeployment() : base(BYTECODE) { }
    public SoftwareManagementDeployment(string byteCode) : base(
   byteCode) { }
}

public class SoftwareManagementDeploymentBase :
   ContractDeploymentMessage
{
    public static string BYTECODE = /*contract bytecode*/;
    public SoftwareManagementDeploymentBase() : base(BYTECODE) { }
```

```
12      public SoftwareManagementDeploymentBase(string byteCode) : base(
        byteCode) { }
13  }
14
15  public partial class CreateProductFunction : CreateProductFunctionBase
        { }
16
17  [Function("createProduct")]
18  public class CreateProductFunctionBase : FunctionMessage
19  {
20      [Parameter("uint256", "_productId", 1)]
21      public virtual BigInteger ProductId { get; set; }
22      [Parameter("uint256", "_price", 2)]
23      public virtual BigInteger Price { get; set; }
24      [Parameter("uint256", "_subTime", 3)]
25      public virtual BigInteger SubTime { get; set; }
26  }
```

SoftwareManagementService è una classe che contiene le funzioni per fare il deploy del contratto ed eseguire transazioni.

```
/*Excerpt of SoftwareManagementService.cs*/

public static Task<TransactionReceipt>
    DeployContractAndWaitForReceiptAsync(Nethereum.Web3.Web3 web3,
    SoftwareManagementDeployment softwareManagementDeployment,
    CancellationTokenSource cancellationTokenSource = null)
{
     return web3.Eth.GetContractDeploymentHandler<
    SoftwareManagementDeployment>().SendRequestAndWaitForReceiptAsync(
    softwareManagementDeployment, cancellationTokenSource);
}

public SoftwareManagementService(Nethereum.Web3.Web3 web3, string
    contractAddress)
{
     Web3 = web3;
     ContractHandler = web3.Eth.GetContractHandler(contractAddress);
}

public Task<TransactionReceipt>
    CreateProductRequestAndWaitForReceiptAsync(BigInteger productId,
    BigInteger price, BigInteger subTime, CancellationTokenSource
    cancellationToken = null)
{
     var createProductFunction = new CreateProductFunction();
     createProductFunction.ProductId = productId;
     createProductFunction.Price = price;
     createProductFunction.SubTime = subTime;

     return ContractHandler.SendRequestAndWaitForReceiptAsync(
         createProductFunction, cancellationToken);
}

/*other functions*/
```

Per fare il deploy del contratto, si chiama la funzione statica di *SoftwareManagementService*; una volta che la transazione si è conclusa, ricaviamo l'indirizzo dalla *transaction receipt*. In seguito, si crea un oggetto *SoftwareManagementService*, che servirà per chiamare e interagire con il contratto.

```
TransactionReceipt receipt = await SoftwareManagementService.
    DeployContractAndWaitForReceiptAsyn(ownerAccount, new
    SoftwareManagementDeployment());

string ContractAddress = receipt.ContractAddress;
Service = new SoftwareManagementService(ownerAccount, ContractAddress);

Service.CreateProduct...
Service.PauseContract...
```

6.3 Website

Quando l'utente clicca *buy* e *renew*, bisogna controllare che nel browser esista un *wallet*. Se esiste, si chiede all'utente di accedere al proprio account; in seguito, si crea un'istanza di contratto. In *Web3.js* un contratto prende come parametri l'indirizzo e l'ABI (*Application Binary Interface*): l'ABI è la lista delle funzioni del contratto, codificate secondo specifiche precise (tipo, nome, inputs, outputs etc); simile alle API, è un'interfaccia che definisce le regole per interagire con il byte-code conservato nella blockchain, indicando i parametri necessari e il formato in cui devono essere ritornati i risultati.

```
/*Excerpt of contract ABI*/
{
    "inputs": [
        {
            "internalType": "uint256",
            "name": "_productId",
            "type": "uint256"
        },
        {
            "internalType": "address",
            "name": "_owner",
            "type": "address"
        },
        {
            "internalType": "uint256",
            "name": "_period",
            "type": "uint256"
        }
    ],
    "name": "purchaseLicense",
    "outputs": [
        {
            "internalType": "uint256",
            "name": "newLicenseId",
            "type": "uint256"
        }
    ],
    "stateMutability": "payable",
    "type": "function"
}
```

```
if (window.ethereum) { //check if there's a wallet
    window.web3 = new Web3(web3.currentProvider); //get current
   provider; MetaMask uses Infura
    try {
        ethereum.enable(); //ask connection to user
        contract = new web3.eth.Contract(abi, contractAddr);
    } catch (error) {
    }
} else {
}
```

Per le transazioni che cambiano lo stato del contratto, viene usato il metodo *send*, mentre *call* è usato per le transazioni che ritornano un valore o lo stato del contratto. Per l'acquisto e il rinnovo di una licenza quindi usiamo *send*; in aggiunta, usiamo anche i *promiEvent*: in JavaScript, *Promise* è un oggetto con la quale possiamo associare a certi eventi di una funzione asincrona, come il suo successo o fallimento, certi *handlers*; in *Web3.js* un *promiEvent* è un *Promise* combinato con un *event emitter* per eseguire certe azioni a seconda dello stadio attuale della transazione.

```
//example for the method purchaseLicense
contract.methods.purchaseLicense(1, userAddress, period)
    .send({
        from: web3.givenProvider.selectedAddress,
        to: contractAddr,
        value: totalPrice
    })
    .on('receipt', function (receipt){
        console.log(receipt);
        id = receipt.events.LicensePurchase.returnValues.id
        //show license id to user
    })
    .on('error', function(error, receipt){
        //show error message to user
    });
```

6.4 Testing

Il testing dei contratti è stato effettuato con Truffle e Ganache. Effettuare il testing è stato necessario in quanto possono evidenziare eventuali errori o assicurare che costrutti come *require* e *function modifiers* si comportino correttamente prima di iniziare lo sviluppo front-end. Sono stati realizzati tre *unit test* in JavaScript: uno per l'aggiunta e la modifica dei prodotti; uno per l'acquisto e il rinnovo di una licenza; uno per verificare che il trasferimento dei token sia stato correttamente disabilitato. Per eseguire i test, si usa il terminale: si lancia prima la testnet locale, poi si esegue il comando *"truffle test"*, che in automatico compila il contratto e svolge i test.

```
//Excerpt of JavaScript unit test
contract("Inventory", accounts => {
    let sm;
    let productId = 1;

    it("should create a product", async() => {
        sm = await SoftwareManagement.deployed();

        await sm.createProduct(productId, price, subTime);

        let paused = await sm.isContractPaused();
        assert.equal(paused, true, "Contract should be paused");
        assert.equal(await sm.getPrice.call(productId), price,
            "Price should match");
        [...]
```

```
16      });
17
18      it("should not create product with same id", async() => {
19          await truffleAssert.reverts(
20              sm.createProduct(productId, pPrice, subTime),
21              "The product already exists"
22          )
23      });
24  });
```

6.5 Esempio di sessione

In questa sezione descriviamo i passi per comprare e validare un prodotto. Per utilizzare il sistema apriamo il terminale e lanciamo il server locale per il sito web con il comando *live-server*; con *"ganache-cli -p 8545"* eseguiamo la testnet locale, altrimenti possiamo usare Ropsten effettuando i necessari cambiamenti al codice sorgente.

1. Aprire *slm* e creare un nuovo contratto. Si aprirà la *view* dell'inventario, ancora vuota. Sopra a destra cliccare su *Add product*: comparirà una finestra di dialogo dove si inserisce l'id, il prezzo e il periodo di iscrizione del nuovo prodotto che si vuole inserire. Attendere il completamento della transazione.

2. Completata l'operazione, la tabella conterrà una nuova riga contenente le informazioni del prodotto appena creato. Fare click destro sulla riga del prodotto per fare eventuali cambiamenti (cambiare il prezzo o il periodo di iscrizione; mettere il prodotto disponibile o non disponibile).

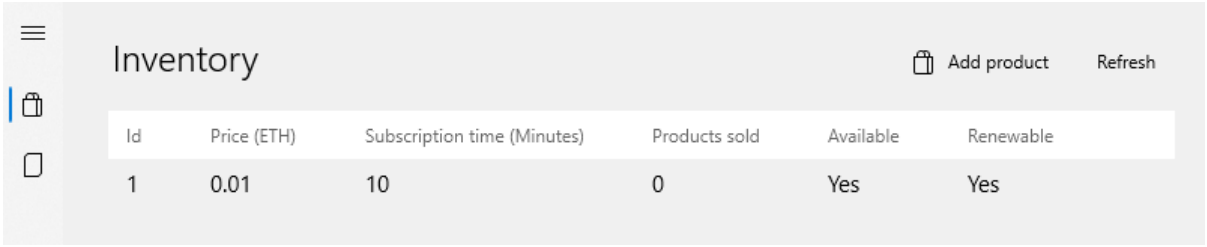

Figura 6.3: View dell'inventario

3. Usando il menù laterale a sinistra dirigersi alla *view* del contratto e cliccare su *Unpause contract*.

4. Aprire il sito web, scorrere in fondo alla pagine e cliccare su *Buy*. Apparirà la notifica di MetaMask dove si inserisce la password per accedere al proprio account. Nella *form* inserire il periodo di iscrizione e premere qualunque *Enter* per visualizzare il prezzo totale; cliccare *Proceed* per procedere l'ordine.

5. Si aprirà la finestra di MetaMask dove si possono vedere i dettagli della transazione. Confermare per eseguire la transazione. Al suo completamento, comparirà l'id del licenza.

6. Il procedimento per rinnovare una licenza è analogo.

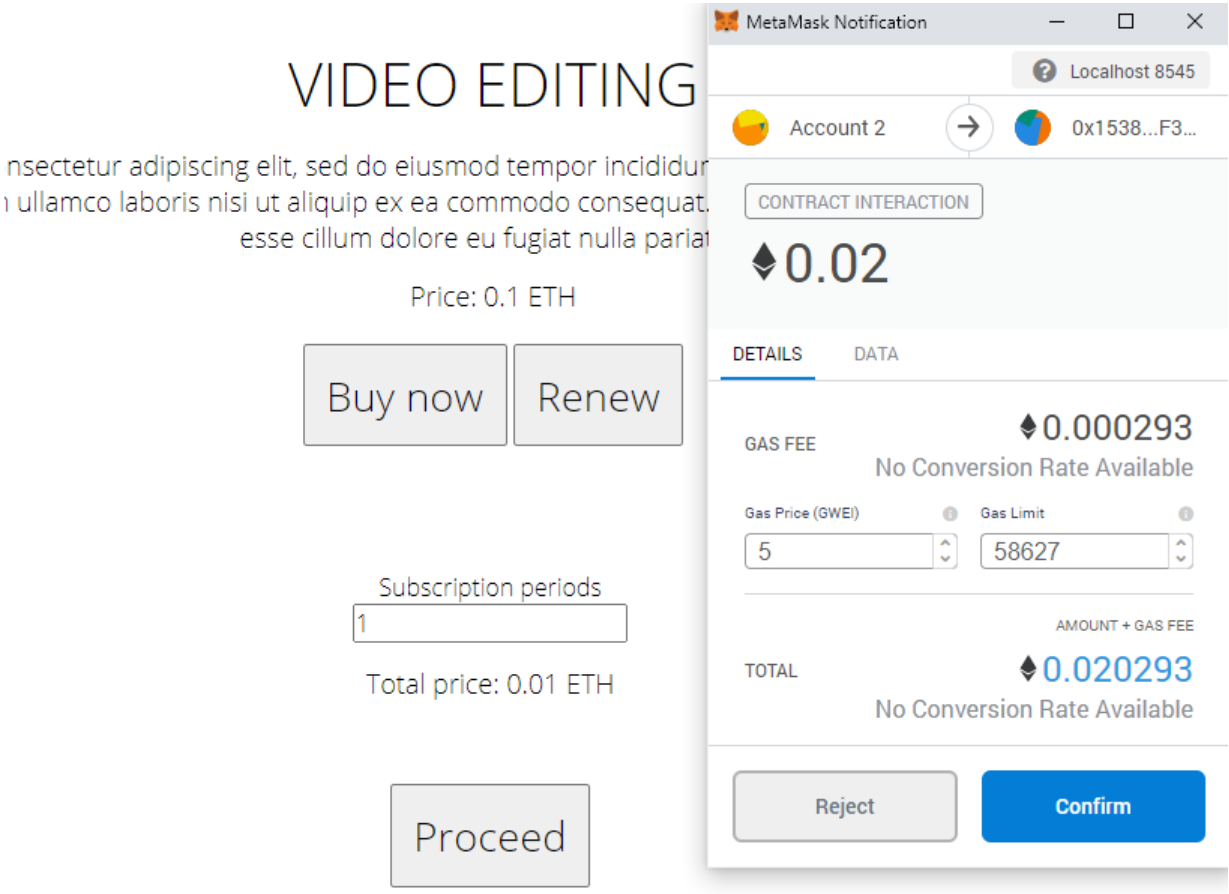

Figura 6.4: Acquisto della licenza e notifica di MetaMask

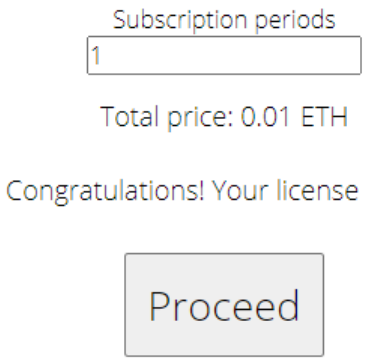

Figura 6.5: Id della licenza

7. Aprire *Application*. Non trovando una licenza nella cartella locale dell'applicazione, comparirà un messaggio di errore. Chiudere e cliccare in alto a destra *Login* e inserire l'id. Attendere la validazione della licenza.

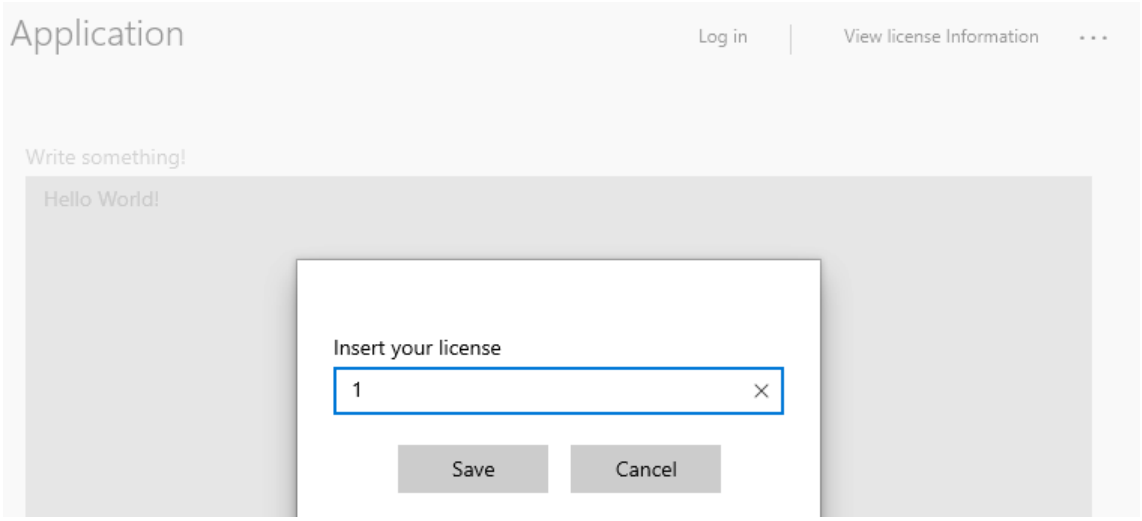

Figura 6.6: Inserimento della licenza

8. Una volta validata la licenza, l'area di testo viene abilitata. Cliccare su *View license information* per vedere le informazioni della licenza.

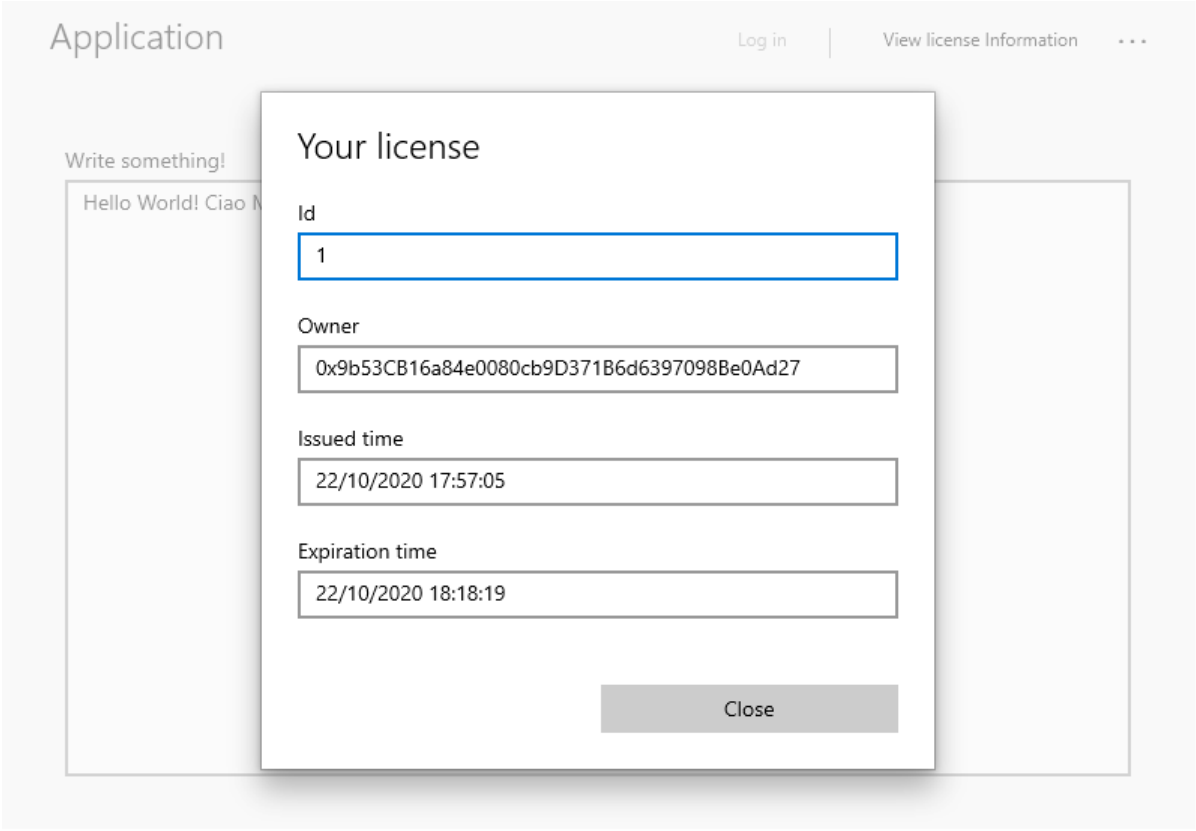

Figura 6.7: Inserimento della licenza

Capitolo 7

Conclusioni

La soluzione che abbiamo progettato è soltanto un prototipo applicato alle applicazioni desktop.

- L'azienda e gli utenti sono assicurati che tutti i dati saranno praticamente immutabili, sempre disponibili, verificabili e facilmente rintracciabili una volta registrati sulla blockchain; un confronto tra i dati inseriti dall'utente e quelli conservati nella blockchain può stabilire con semplicità se l'utente è l'effettivo titolare della licenza.
- In un sistema centralizzato, molto spesso l'utente deve affidarsi a vari *third-party services*; la blockchain, invece, elimina questa necessità, permettendo all'utente di interagire direttamente con il contratto.
- La licenza è validata tramite chiave privata. L'azienda può essere certa che se nella blockchain una certa chiave non è in possesso della licenza, allora il software non potrà essere utilizzato. Anche se gli indirizzi pubblici titolari delle licenze sono facilmente accessibili, un utente non è in grado di utilizzare il software conoscendo soltanto la chiave pubblica, in quanto è praticamente impossibile ricavare la chiave privata da quella pubblica [13].

In riguardo al caso specifico degli smart contract, un utente ha la sicurezza che un contratto non cambierà, d'altra parte deve "fidarsi" di esso – si ricordi per esempio l'uso di *SelfDestruct*. È possibile decompilare il byte-code per ottenere un codice più leggibile all'essere umano, ma comunque lontano dal contratto originale. Altrimenti, se questo è open-source, un utente diffidente può studiare il codice, compilarlo e confrontare il byte-code ottenuto con quello conservato nella blockchain. Un procedimento come questo può essere eseguito anche da terze parti: un esempio è il contratto "Golem: Multisig" che nel 2016 ha richiesto la verifica del contratto su Etherscan [56].

Un altro aspetto che bisogna considerare è che eseguire un contratto sulla blockchain ha un costo. Questo argomento sarà analizzato meglio nella sezione "Transaction fee".

7.1 Threath model

Alcune delle vulnerabilità di una gestione di licenze per un software desktop non sono legate alla blockchain stessa.

7.1.1 Conservazione delle chiavi private

Usare le chiavi private è un elemento addizionale per assicurarci che la licenza venga usata solo e soltanto dall'utente che ha comprato il prodotto. È compito suo conservarla in modo sicuro per evitarne il furto. Nessuno però può assicurarci che l'utente, a suo rischio e pericolo, non condivida la propria chiave privata con altre persone, magari creando appositamente un account condiviso con terzi. Per limitare ciò, si potrebbe fissare un numero massimo di installazioni permesse e registarne ogni uso sulla blockchain, con rilevanti ricadute sia sulla privacy dell'utente, sia sulla sua sicurezza a causa della diffusione di informazioni che potrebbero essere utili per attaccarlo.

Usare una chiave privata ha un ulteriore svantaggio: nel caso in cui l'utente perda la propria chiave, non potrà più entrare nel proprio account e quindi validare la propria licenza. In un sistema tradizionale dove magari si fa uso di e-mail, password e chiave d'attivazione, un utente è in grado di visualizzare tutti i prodotti registrati nel proprio account; se si perde la chiave e la password del proprio account, l'utente può semplicemente contattare il supporto clienti, che manderanno una mail con il link del ripristino della password. Una cosa del genere non può accadere con la chiave privata: se si perde la chiave privata, gli Ether contenuti nell'account sono perduti per sempre e con esso anche i Token; non ci sarà nessun supporto clienti in grado di aiutarci nel recuperare la chiave.

Un altro problema è la portabilità delle chiavi. Al momento, l'applicazione desktop manca di un sistema di autenticazione, mentre è necessario che il sistema permetta gli utenti di accedere ai programmi installati e validare la licenza qualunque sia il modo in cui sono conservate la chiave. Affinché il processo sia automatizzato, il programma deve essere in grado di leggere la chiave privata a seconda del suo formato (in chiaro, *keystore*, *hardware wallet*, *mnemonic phrase*), oppure comunicare con tool terzi (*digital wallet*) che effettuano la validazione e ritornano il risultato all'applicazione. Se invece vogliamo che sia il software stesso a possedere una chiave privata, come nel caso di *cryppadotta*, l'utente dovrebbe conoscere comunque la chiave: se è costretto a cambiare PC, dovrebbe essere in grado di installare il software e importare al suo interno la chiave privata già in suo possesso, senza essere costretto a comprare un'altra licenza.

7.1.2 Software piracy

Si definisce software "craccato" un software che è stato modificato per bypassare le misure che sono state inserite nel programma per impedirne l'uso, la copia e la distribuzione non autorizzata [57].

Attraverso un processo di *reverse engineering*, viene studiato il funzionamento del software per creare tools che permettono di raggirare i meccanismi di validazione inseriti nei programmi. Supponiamo, per esempio, di avere un software commerciale a pagamento che usa *product keys*: il software viene disassemblato, studiando quali sono i controlli che effettua per verificare se una chiave è valida e meno; da questa analisi vengono sviluppati programmi che generano *product keys* che sono correttamente validate dal software. Programmi di questo tipo sono detti *keygen*. Altrimenti, è possibile creare programmi, detti *patch*, che modificano direttamente il codice macchina del software, cambiandone le condizioni di validazione [58].

L'uso della blockchain può aiutare a tenere traccia delle licenze e del loro uso, ma il *software cracking* rimane comunque una possibilità.

7.1.3 Vulnerabilità nel contratto

Anche se le tecnologie dietro la blockchain possono essere sicure, lo stesso non può essere detto degli smart contract.

Se un contratto fallisce, bisogna provvedere a risolvere le vulnerabilità nel minor tempo possibile per minimizzare i danni. Benché siano già stati individuati varie raccomandazioni e pattern per rendere i contratti più sicuri [42] [51], uno smart contract può comunque contenere errori e bug che utenti malintenzionati possono abusare; anche se siamo molto convinti che il contratto sia bug-free, non è comunque detto che il compiler o le varie piattaforme che interagiscono con il contratto non abbiano vulnerabilità che gli hacker possono sfruttare.

7.2 Limiti

7.2.1 Connessione a Internet

Una validazione della licenza che usa solo la blockchain necessita di una connessione Internet: ad ogni suo avvio, il software interagisce con la blockchain per controllare che la chiave privata dell'utente sia l'effettivo titolare della licenza. Esistono in commercio invece programmi che al primo avvio vengono "attivati", per esempio con una chiave, e che poi possono essere liberamente usati offline.

7.2.2 Efficienza e consumi

In generale blockchain che fanno uso di *Proof of Work* usano molte risorse, sono generalmente lente nella verifica delle transazioni e tendono a concentrare l'*hashing power*, ovvero quanto è potente e veloce un computer a risolvere problemi computazionali [6].

- Spreco di risorse: poiché i miners ricevono criptovalute come ricompensa, le persone tendono a fare upgrade del proprio hardware per avere maggior l'*hashing power*; ne risulta di conseguenza un uso maggiore di elettricità
- Lentezza delle transazioni: i tempi di validazione di una transazione possono essere molto lunghi. Bitcoin per esempio produce un blocco ogni 10 minuti.
- Concentrazione dell'*hashing power*: effettuare il *mining* di un blocco potrebbe essere molto difficile per un singolo *miner*; molte organizzazioni quindi hanno creato dei *mining pool*, ovvero gruppi di miners che risolvono insieme il problema matematico e la ricompensa viene divisa a seconda dei contributi del singolo utente. Se l'*hashing power* di una o più mining pool è maggiore del 50%, questi hanno il monopolio della blockchain e la possibilità di effettuare un attacco, in quanto avrebbero il potere di bloccare, validare e annullare transazioni a loro piacimento. Un attacco del genere è detto *51% attack*.

A causa della scarsa quantità di dati disponibili, condizioni variabili e varie metodologie, è difficile stimare la quantità di elettricità consumata dalle blockchain: ricerche del 2018-2019 hanno provato a stimare i consumi di Bitcoin, ottenendo valori tra 20-80 terawatt-hours (Twh) all'anno, circa l'0.1-0.3% dell'elettricità usata globalmente [59]:

quantità come queste sono pari a un paese come l'Irlanda (26 TWh) ma meno di quella usata per esempio per l'aria condizionata e ventilatori elettrici, che nel 2018 era pari a 2075 TWh [60]. A confronto, Ethereum è più veloce e consuma meno energia elettrica: con un prezzo medio del gas pari a 0.000000025 ETH, riesce a generare un blocco ogni 15 secondi [61] e benché sia difficile fare misure corrette, si stima che usi un terzo dell'energia impiegata da Bitcoin [59]. Confrontandoli con i consumi dei *data centers* globali, che nel 2018 sono stati stimati a 205Twh [62], l'elettricità usata da Bitcoin ed Ethereum sono abbastanza alti se si considera che uno studio dello stesso anno [63] ha calcolato che il numero di utenti attivi che fanno uso di blockchain – quindi non solo Bitcoin ed Ethereum – sono almeno 35 milioni di persone.

Per far fronte a questo problema, nel 2018 il fondatore di Ethereum, Vitalik Buterin, ha annunciato che aveva intenzioni di passare dal *Proof of Work* al *Proof of Stake*, un meccanismo di consenso che usa meno energia e riduce il rischio di un *51% attack* [64]. Ad oggi, non è ancora stata fissata una data per il lancio della nuova blockchain di Ethereum, detta *Ethereum 2.0*, e probabilmente ci vorrà anche qualche anno prima che sia completamente funzionante [65].

7.2.3 Transaction fee

Abbiamo detto che ogni transazione su una blockchain costa una certa quantità di gas. Una volta fatto il deploy del contratto sulla testnet di Ropsten, possiamo visualizzare il gas usato.

Function	Gas Used
Contract deployment	2,854,782
addProduct	149,894
togglePause	27,966
setPrice	29,385
setSubTime	32,602
setAvailable	26,914
purchaseLicense	310,003
renewLicense	40,009
withdraw	30,523

Figura 7.1: Quantità di gas usato dalle principali funzioni

Le funzioni più costose sono il deploy del contratto, l'aggiunta di un prodotto e l'acquisto di una licenza, in quanto sono le funzioni dove si salvano dati. Con un prezzo pari a circa €320 per ETH (ottobre 2020), se consideriamo che il costo del gas in Ropsten al momento delle transazioni era pari a 0.000000005 ETH, l'owner per fare il deploy del contratto e aggiungere un prodotto dovrà pagare rispettivamente 0.01427391 ETH e 0.00074947 ETH, €4.60 e €0.24. Per comprare una licenza, l'utente dovrà pagare una somma pari a 0.001550015 ETH, ovvero €0.50.

Se usassimo il prezzo medio usato in Ethereum, pari a 0.000000025 ETH, i prezzi salirebbero. Inoltre, bisogna far presente che costi come questi sono solo indicativi, in quanto il prezzo dell'Ether e del gas è molto volatile.

Function	**Ropsten (0.000000005 ETH)**	**Ethereum (0.000000025 ETH)**
Contract deployment	0.01427391 ETH €4.60	0.0713695 ETH €22.81
addProduct	0.00074947 ETH €0.24	0.0037473 ETH €1.20
purchaseLicense	0.001550015 ETH €0.50	0.0075007 ETH €2.40
renewLicense	0.00020004 ETH €0.06	0.0010002 ETH €0.32

Figura 7.2: Confronto dei costi delle transazioni dato il prezzo del gas in Ropsten e Ethereum

7.3 Considerazioni finali

Il sistema implementato ha ancora molti aspetti da migliorare. Un'applicazione desktop è vulnerabile alla pirateria e non bisogna pensare che usare la blockchain possa improvvisamente risolvere questo problema; inoltre è una soluzione che consuma molta energia elettrica ed è poco adatta per conservare grandi quantità di dati. Ricerche specifiche sulla blockchain di Ethereum e le sue possibili applicazioni, anche se in continuo aumento, sono ancora scarse se confrontate a Bitcoin o a DLT generali. I possibili vantaggi dati dalla blockchain, come decentramento e *trustlessness*, sono apprezzabili più dagli utenti che dalle aziende, che invece potrebbero preferire soluzioni più perfomanti e scalabili. Nonostante ciò, è sicuramente una tecnologia da tenere d'occhio, in quanto ancora in via di sviluppo.

Bibliografia

[1] Sofia Giancone. Il contratto di licenza d'uso del software. *IUS in Itinere*, 2020-05-27.

[2] GNU. Licenses.

[3] Business Software Alliance (BSA). Software management: Security imperative, business opportunity. *BSA Global Software Survey*, 2018.

[4] Satoshi Nakamoto. Bitcoin Whitepaper.

[5] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017.

[6] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[7] Quynh H Dang. Secure hash standard. Technical report, Information Technology Laboratory, National Institute of Standards and Technology (USA), 2015.

[8] Vitalik Buterin. Ethereum Whitepaper: Mining, 2013.

[9] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*, chapter 1. Introduction to Cryptography and Cryptocurrencies. Princeton University Press, 2016.

[10] Daniel Conte de Leon, Antonius Q Stalick, Ananth A Jillepalli, Michael A Haney, and Frederick T Sheldon. Blockchain: properties and misconceptions. *Asia Pacific Journal of Innovation and Entrepreneurship*, 2017.

[11] Michèle Finck. Blockchains and Data Protection in the European Union. *Max Planck Institute for Innovation & Competition Research Paper*, 2017.

[12] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2020. 4.3.4. Block Header Validation.

[13] Andreas M. Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. Oreilly & Associates Inc, 1 edition, 2018.

[14] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*, chapter 1.Overview of Cryptography. CRC press, 2018.

[15] Neal Koblitz, Alfred Menezes, and Scott Vanstone. The State of Elliptic Curve Cryptography. *Designs, codes and cryptography*, 19(2-3):173–193, 2000.

[16] Solidity. Documentazione.

[17] Vyper. Documentazione.

[18] Ipfs. Sito ufficiale.

[19] Swarm. Sito ufficiale.

[20] ConsenSys. Ethereum, gas, fuel and fees. ConsenSys Media, 2016-06-23.

[21] Ethereum. EIP-20: ERC-20 Token Standard.

[22] Ethereum. EIP-721: ERC-721 Non-Fungible Token Standard.

[23] Criptokitties. Sito ufficiale.

[24] Ethereum. EIP-1155: ERC-1155 Multi Token Standard.

[25] Ethereum. Nodes and clients.

[26] Pieter Wuille. Bitcoin Improvement Proposal BIP-0032.

[27] Marek Palatinus, Pavol Rusnak, Aaron Voisine, and Sean Bowe. Bitcoin Improvement Proposal BIP-0039.

[28] Binance Academy. What Is a Hardware Wallet (and Why You Should Use One).

[29] Lucianna Kiffer, Dave Levin, and Alan Mislove. Stick a fork in it: Analyzing the Ethereum network partition. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 94–100, 2017.

[30] David Siegel. Understanding The DAO Attack. *Coindesk*, 2016-06-27.

[31] Ethereum Classic. Sito ufficiale.

[32] Jeff Herbert and Alan Litchfield. A Novel Method for Decentralised Peer-to-Peer Software License Validation Using Cryptocurrency Blockchain Technology. In *38th Australasian Computer Science Conference (ACSC 2015)*, volume 27, 2015.

[33] Christian Fortin. Master Bitcoin - The Proof of Ownership.

[34] Aaron Lebo. Implementation of a decentralized, transferable, and open software license system using the Bitcoin protocol.

[35] Jeff Herbert and Alan Litchfield. ReSOLV: Applying Cryptocurrency Blockchain Methods to Enable Global Cross-platform Software License Validation, 2018.

[36] Dotta Bot. Contratti.

[37] Accenture. Accenture Deploys New Software License Management Application on Digital Asset's Distributed Ledger Platform.

[38] Neocor. Neocor Launches Fusion Ledger.

[39] license.rocks. Sito Ufficiale.

[40] Nicola Greco. ERC1238: Non-transferrable tokens.

[41] Jiachi Chen, Xin Xia, David Lo, and John Grundy. Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. *arXiv preprint arXiv:2005.07908*, 2020.

[42] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018.

[43] Ganache. Sito ufficiale.

[44] Ethereum. Networks.

[45] Infura. Sito ufficiale.

[46] Etherscan. Sito ufficiale.

[47] Truffle. Sito ufficiale.

[48] Web3. Documentazione.

[49] MetaMask. Sito ufficiale.

[50] Nethereum. Documentazione.

[51] Solidity. Solidity: Securety consideration.

[52] William Foxley. Ethereum's Istanbul Hard Fork Is Now Live. *Coindesk*, 2019-12-08.

[53] Francisco Giordano. Reentrancy After Istanbul. *Open zeppelin*, 2019-12-11.

[54] diligence.consensys. Secure Development Recommendations: Timestamp.

[55] Microsoft. The MVVM Pattern.

[56] Golem: Multisig. Contratto verificato su Etherscan.

[57] Mitch Tulloch. *Microsoft encyclopedia of security*. Microsoft Press, 2003.

[58] Ron Honick. *Software piracy exposed*, chapter 4. Crackers. Elsevier, 2005.

[59] George Kamiya. Bitcoin energy use - mined the gap. *IEA - International Energy Agency*, 2019-07-05.

[60] IEA. Cooling.

[61] Etherscan. Ethereum Average Block Time Chart.

[62] Energy Innovation. How Much Energy Do Data Centers Really Use?

[63] Michel Rauchs, Apolline Blandin, Kristina Klein, Gina Pieters, Martino Recanatini, and Bryan Zhang. *2nd Global Cryptoasset Benchmarking Study*. University of Cambridge - Judge Business School: Cambridge Centre for Alternative Finance, 2018.

[64] Shiraz Jagati. Ethereum's Proof of Stake Protocol Under Review. *Cryptoslate*, 2018-04-22.

[65] Ethereum. Ethereum 2.0 (Eth2). Last update: 2020-09-28.