Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica Per Il Management

Un sistema di micropagamenti su piattaforma Ethereum per il wireless roaming

Relatore:
Chiar.mo Prof.
GABRIELE D'ANGELO

Presentata da:
GIOVANNI PIETRUCCI

Correlatori:
Chiar.mo Prof.
STEFANO FERRETTI

Chiar.mo Dott.
MIRKO ZICHICHI

Sessione III
Anno Accademico 2018/2019

Introduzione

Viviamo in un'era di grandi innovazioni tecnologiche, tra queste, stanno prendendo piede i pagamenti digitali surclassando i normali metodi di pagamento. In particolare, nell'ultimo decennio stanno avanzando nuove tecniche di pagamento decentralizzato tramite ciptovalute. Tali metodologie sono fondate su una tecnica rivoluzionaria chiamata blockchain, presentata assieme a bitcoin per la prima volta da Satoshi Nakamoto nel 2008. Tale tecnologia si propone come un registro distribuito, gestito da una rete peer-to-peer, in grado di archiviare immutabilmente le transazioni in una rete di computer attraverso un algoritmo di consenso. I movimenti all'interno di questo sistema sono trasparenti, sicuri e verificabili. Lo sviluppo di queste nuove tecniche di pagamento, ha portato alla luce Ethereum, una piattaforma decentralizzata che permette l'implementazione di Dapp e smart contracts. Questi ultimi rappresentano un insieme di istruzioni descritte in un linguaggio di programmazione (Solidity) le quali vengono eseguite automaticamente da una macchina virtuale. L'implementazione di tali tecnologie ha alla base una struttura incentrata sul consenso di tutti i nodi della rete. Specie le transazioni che hanno bisogno di essere elaborate dai partecipanti della rete. La crescita di tali piattaforme ha portato nel tempo un incrementale numero di utenti che causato un rallentamento e un aumento del prezzo per transazione. La soluzione a questo problema risiede in delle tecniche di pagamento off-chain chiamate payment channel che limitano l'inserimento di alcune transazioni nella blockchain. Questo progetto nasce con l'intento di regolamentare delle microtransazioni tra un device gestito da un utente e un insieme di router al quale il device richiede la connessione. Tali richieste vengono gestite da smart contracts, i quali affiancati da un meccanismo di messaggistica firmata, vanno a fornire una soluzione al problema della scalabilità realizzando un sistema che limiti l'inserimento di transazioni ripetute in una blockchain, riducendo i costi e il tempo di attesa.

Indice

Elenco delle figure

Capitolo 1

Stato dell'arte

1.1 Web Decentralizzato

Prima di introdurre il concetto di *Web decentralizzato* o *Web 3.0* è bene fare un passo indietro. A partire dagli anni '90, con l'avvento del *World Wide Web*[1], si aveva la possibiltà di consultare con accesso diretto ad internet, pagine web statiche senza la possibilità di interazioni. Con l'avvento del *Web 2.0*, il web venne rivoluzionato implementando funzionalità più dinamiche che permisero una maggiore interconnessione tra le parti che componevano la rete. Con l'arrivo delle più grandi multinazionali in campo informatico ed e-commerce, si è pian piano generata una sorta di centralizzazione dei servizi attraverso i server resi disponibili da essi. L'obbiettivo del Web 3.0 è dunque quello di riportare il web ad una fase di decentralizzazione che eviti di avere dei server intermedi obbligatori per poter effettuare una determinata operazione. Un altro punto fondamentale[2], è quello che con l'attuale sistema i rischi di attacchi ai dati, oltre ad essere maggiori, sono anche più pericolosi, in quanto tutte le informazioni sono raggruppate nelle mani di pochi. Il Web 3.0 si propone come soluzione a tale scelta garantendo un maggiore controllo e privacy, distribuendo il *cloud* e la memoria tra i computer che compongono la rete. L'architettura rivoluzionaria che ha dato il via a questo nuovo tipo di sistemi è la blockchain, che come vedremo, non viene più collegata a transazioni ma verrà utilizzata nello sviluppo del Web 3.0 includendo gli spostamenti e la registrazione di dati. La blockchain viene vista come *distributed ledger* aperto a tutti venendo descritta dagli autori di un report[3] del governo del Regno Unito come *"essentially an asset da-*

tabase that can be shared across a network of multiple sites, geographies or institutions", ossia "come un database di risorse che può essere condiviso su una rete di più siti, aree geografiche o istituzioni".

1.1.1 Architettura Peer-to-peer

La rete *Peer-to-peer* (P2P) è un sistema di entità (*Peers*) connessi tra loro tramite internet. Si distacca dal modello *Client-Server* in cui il *server* funge da entità centrale a cui i *client* si connettono creando un sistema centralizzato, dove tutto dipende dal comportamento del *server*, come illustrato in figura 1.1. Nel modello P2P i peers possono comunicare tra loro senza un server centrale. L'unico vincolo posto ad un'entità per accedere ad una rete P2P è quella avere una connessione internet e un software P2P

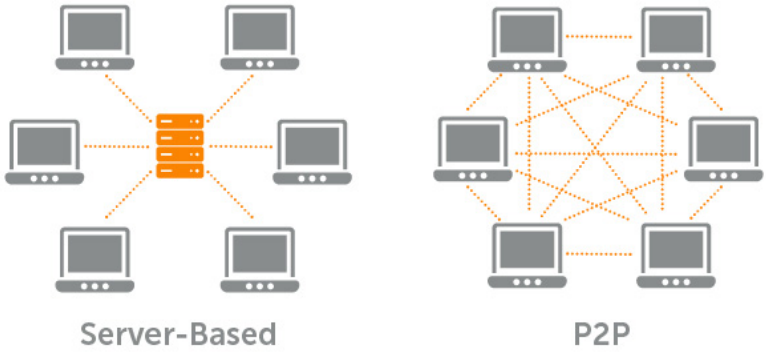

Figura 1.1: Rappresentazione del modello client-server e del modello P2P

La caratteristica più importante è quella di presentare un modello parzialmente o completamente decentralizzato, in quanto i nodi presenti nel sistema hanno la possibilità di auto-organizzarsi e condividere risorse distribuite senza l'ausilio di un server su cui basare tutto il calcolo e la potenza computazionale, evitando il *single point of failure*[1]. La decentralizzazione in questo sistema permette di stabilire una rete fondata sulla distribuzione e collaborazione tra i vari nodi. Tale modello è alla base dell'archittettura

[1]Single point of failure: In un sistema, rappresenta la parte più vulnerabile che in caso di malfunzionamento o attacchi DoS può portare ad anomalie

su cui si fondano alcuni tra i più importanti sistemi di crittovaluta al mondo, ossia la *Blockchain*.

1.2 Blockchain

Era l'11 Marzo 2008 quando Satoshi Nakamoto[2] scrisse: "*I've been working on a new electronic cash system that's fully peer-to-peer, with no trusted third party.*" che letteralmente significa: "Ho lavorato su un nuovo sistema elettronico di pagamento, che è completamente peer-to-peer, senza terze parti attendibili". Attraverso questa frase introduttiva, Nakamoto, aprì le porte ad un rivoluzionario sistema di pagamento decentralizzato, descrivendolo in un whitepaper[6]. Il tutto era basato su una tecnologia o idea originariamente descritta[5] in forma embrionale da un gruppo di ricercatori guidati da Stuart Haber e W. Scott Stornetta. Essi svilupparono una tecnica che prevedeva un marcatore temporale (timestamp) di documenti digitali, in tal modo, si evitava una possibile revisione di questi ultimi con conseguente modifica in scrittura, diventando dunque immutabili. Tale modello non venne più utilizzato fino al 2008, quando Nakamoto ne riprese e modificò il concetto per poter creare la blockchain e i Bitcoin, la prima crittovaluta al mondo.
Prima di procedere con la descrizione dei meccanismi di una blockchain è bene descrivere cosa essa sia.
Con blockchain[7][6] si intende un'architettura che implementa un sistema di archiviazione immutabile di transazioni su una rete di computer. Nel dettaglio si tratta di una sequenza digitale di blocchi in cui sono memorizzate delle transazioni. Tale sistema permette uno scambio sicuro, una registrazione di transazioni ed un *broadcasting* di queste ultime tra individui in tutto il mondo, premesso che abbiano accesso ad internet. Si può ricondurre al concetto di libro mastro[3] pubblico dove tutte le transazioni vengono archiviate in un elenco o catena di blocchi, quest'ultima è in costante crescita nel momento in cui vengono aggiunti nuovi elementi. L'implementazione della blockchain ha permesso lo sviluppo di nuovi sistemi autonomi decentralizzati evitando dunque una centralizzazione ed un collegamento a terze parti o intermediari.

[2]Satoshi Nakamoto: Pseudonimo dell'inventore della criptovaluto Bitcoin

[3]Libro Mastro: È un registro della contabilità formato da tutti i conti movimentati da un sistema contabile

Il sistema di decentralizzazione viene garantito da un sistema peer-to-peer in cui chiunque può partecipare. Quando qualcuno si aggiunge alla rete riceve una copia completa della blockchain, in questo modo può verificarne il suo corretto ordine. Se ad esempio viene creato un blocco durante una transazione, quest'ultimo verra inviato a tutti i partecipanti della rete i quali verificheranno che non sia stato manomesso. Se la verifica termina positivamente, allora tale blocco verrà aggiunto alle blockchain della rete. Poco prima di essere inserito, viene firmato dal nodo tramite una firma digitale usando la sua chiave privata. Ogni utente nella rete possiede due chiavi. Una chiave privata che è usata per creare le firme digitali ed una chiave pubblica la quale possiede due casi d'uso, il primo è che funge da address nella rete ed il secondo è che viene utilizzato per verificare una firma o convalidare l'identità di un inviante. Attraverso la tecnologia blockchain la gestione delle transazioni avviene in modo completamente decentralizzato migliorandone la trasparenza, andando a ridurre i costi e ed integrando tecnologie quali hash crittografico, firma digitale (crittografia asimmetrica) ed un meccanismo di consenso distribuito. Lo sviluppo della blockchain è volto dunque ad abilitare scambi di digital assets tra parti senza una particolare fiducia gli uni negli altri, riducendo il numero di intermediari. Questo meccanismo di creazione e gestione dati ha avuto un forte impatto tra i vari settori in quanto risulta molto efficiente, automatizza i processi riducendo i costi e permette la creazione di nuovi modelli di business.
È possibile distinguere le funzionalità essenziali di una blockchain attraverso tre punti focali:

- **Decentralizzazione**: Grazie al suo sistema decentralizzato, la blockchain, permette di superare il problema della centralizzazione, evitando quindi l'utilizzo di intermediari o di punti centrali fissi su cui tutto il sistema pone le sue basi e dal quale dipende. Tramite questo metodo, viene eliminato un grosso punto debole nei sistemi centralizzati più famosi al giorno d'oggi, ossia evitare e limitare gli attacchi ai server o database da cui dipende il sistema. Il tutto viene implementato eliminando i singoli punti centrali (single point of failure) e distribuendo la potenza computazionale in più calcolatori appartenenti agli utenti del sistema, andando così a creare una solida struttura a rete. è dunque un sistema autonomo basato sul consenso generale dei suoi nodi e sulla trasparenza delle operazioni effettuate;

- **Immutabilità**: Grazie alla presenza di un distributed ledger, è possibile ricondurre

le operazioni effettuate ad un unico proprietario evitando inoltre che esso possa modificarne il contenuto. Ciò che viene scritto nella blockchain è difficilmente modificabile e ne garantisce un accesso grazie ad un'identità univoca nel sistema;

- **Fiducia**: Essendo un sistema decentralizzato e soprattutto sicuro, il rapporto tra i membri della blockchain viene fondato su una struttura fiduciaria basata sul fatto che non esiste più una figura intermediaria (ad esempio le banche) ma di fatto un sistema decentralizzato che permette ai membri di essere parte integrante del distributed ledger. Le regole di gestione delle transazioni sono predefinite, pubbliche e trasparenti.

Con Blockchain [7] (come mostrato in figura 1.2) si intende dunque una sequenza di blocchi contenenti una lista di transazioni in un ordine particolare. Ciascun elemento è legato al successivo tramite un solo collegamento, questo significa che le entità seguono un ordine preciso e non possono avere più collegamenti tra loro.
Chiaramente il primo blocco della catena non contiene un puntatore all'elemento precedente in quanto essendo il primo non avrà punti di riferimento precedenti.
Due strutture dati essenziali vengono utilizzate per la realizzazione di una blockchain:

- **Pointers**: Una variabile che memorizza le informazioni relative alla posizione di un'altra variabile.

- **Linked List**: Una sequenza di blocchi in cui ogni unità ha dati specifici collegandosi tramite il pointer al blocco successivo.

Una volta vista la struttura di una blockchain, è possibile passare all'analisi delle singole unità che la compongono, come illustrato in figura 1.3.
L'unità ripetente si può suddividere in due sezioni: *Header* e *Body*.
Per quanto riguarda l'*Header*, può essere suddiviso come segue:

- **Hash del blocco**: Rappresenta una firma univoca ottenuta attraverso l'algoritmo Hash (SHA 256) che può essere usata per generare un valore Hash di 256bit. Tale firma (o Hash) viene generata nel momento in cui viene creato un blocco. Nel caso in cui quest'ultimo venisse modificato, anche l'Hash subirebbe una variazione.

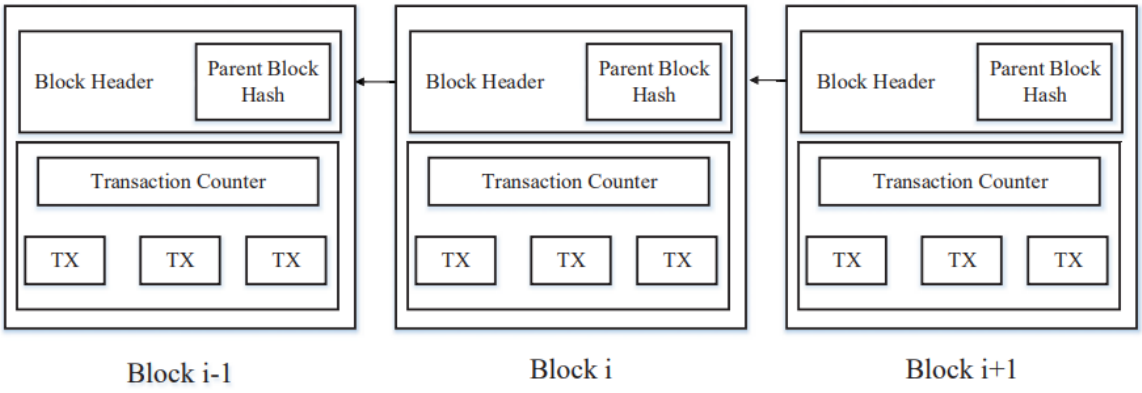

Figura 1.2: Rappresentazione schematica di una blockchain

- **Hash del blocco precedente**: Come appunto dice il titolo stesso, esso rappresenta il puntatore al blocco precedente creando così la struttura a catena. Se un Hash viene modificato vengono ricalcolati tutti gli hash della catena.

- **Merkle Tree Root Hash**: Viene collocato all'interno del blocco e dipende dal tipo della blockchain. Per quelle relative a Bitcoin ed Ethereum ad esempio, rappresenta i dettagli della transazione quali inviante,ricevente e la quantità di denaro. Tutte le transazioni contenute in un blocco possono essere aggregate in un valore hash.

- **Timestamp**: Unità di misura temporale (che ha come termine minimo 1970-01-01 T00:00 UTC) la quale registra il momento in cui è stato creato il blocco.

- **Nonce**: Rappresenta una variabile di 4-byte incrementata dalla proof of work. Viene implementato nel processo di mining.

- **Block Version**: Indica le regole di validazione che deve seguire il blocco durante il suo processo.

- **nBits**: Indica la forma codificata della soglia target, per soglia target si intende la soglia al di sotto della quale deve trovarsi un hash dell'header del blocco affinchè esso stesso sia valido.

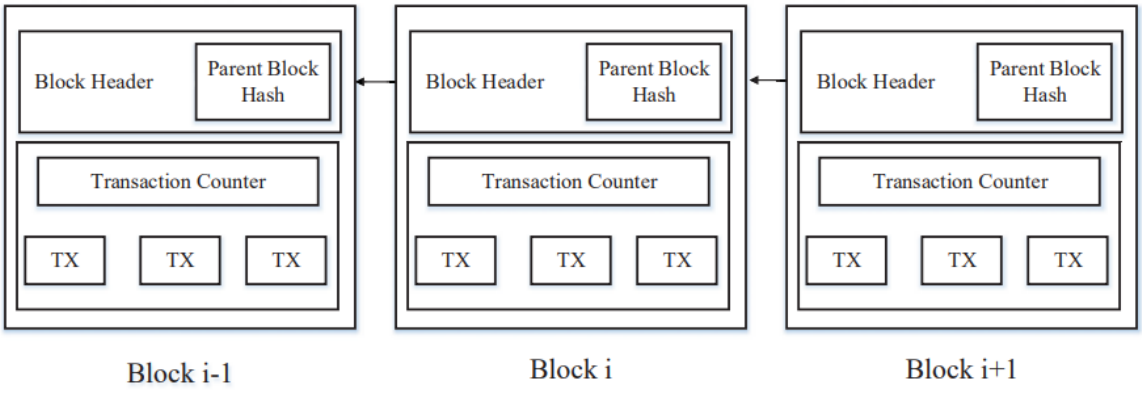

Figura 1.3: Rappresentazione schematica di un blocco della blockchain

Il *Body* invece è composto da un contatore che rappresenta le transazioni ed il numero di transazioni. Il numero massimo che un blocco può contenere dipende dalla grandezza sia del blocco e che delle transazioni.

1.2.1 Distributed Ledger Technology

Il *Distributed Ledger Technology* (*DLT*) è un libro mastro digitale che risiede nella rete dei computer, possiamo intenderlo come un database distribuito tra gli attori di un sistema che ne salvano e replicano una propria copia. Al suo interno vengono gestite, scritte e modificate tutte le operazioni che avvengono in un sistema. La sua funzione è quella di decentralizzare il controllo e di autenticare le transazioni su di esso (dipendenti altrimenti da una figura intermedia), rendendolo pubblico e trasparente a chiunque. Tutte le modifiche che avvengono sul DLT vengono ripubblicate simultaneamente in tutti i nodi del sistema mentre le informazioni vengono memorizzate ed autenticate tramite una firma crittografica. La blockchain rappresenta uno dei primi esempi di distributed ledger tecnology questi infatti garantiscono la memorizzazione delle transazioni in maniera distribuita e permanente, inoltre, tramite un meccanismo di consenso tra i nodi della rete, vanno ad eliminare la necessità di un intermediario che verifichi le transazioni.

1.2.2 Proof of Work

La *Proof of Work (PoW)*[9] è un algoritmo di consenso distribuito tra la rete Blockchain. È un meccanismo volto a proteggere quest'ultima da attacchi *Denial of Service (DoS)* indesiderati. Come definito in precedenza, all'interno di una Blockchain avvengono molteplici transazioni di crittovalute tra gli utenti. Tali transazioni necessitano di una validazione, la quale viene gestita da un gruppo di nodi della blockchain chiamati *Miners*. Quest'ultimi danno il nome a tutto il processo su cui si basa la PoW, ossia *Mining*. Fu introdotto per la prima volta da Bitcoin ed il concetto su cui si struttura è quello di far sfruttare ai Miners la loro potenza di calcolo per risolvere il problema della ricerca di un valore tramite funzioni hash, rendendo possibile la creazione e l'inserimento di nuovi blocchi. Bitcoin ad esempio impiega la PoW basandola su funzioni hash volte alla ricerca di un valore. Il processo di mining dunque, è un meccanismo che implementa funzioni hash quali hashcash per inserire un nuovo blocco nella blockchain. Il tutto garantisce una sicurezza da attacchi DoS in quanto chiunque tenti di modificare le transazioni registrate in un blocco deve eseguire il ricalcolo di quest'ultimo e di tutti i blocchi successivi aumentando il costo e la richiesta di calcolo computazionale.

Capitolo 2

Ethereum

2.0.1 Introduzione Ethereum

L'idea di *Ethereum* venne presentata per la prima volta attraverso un *whitepaper*[11] di Vitalik Buterin nel 2013. Seguendo le orme del sistema di Bitcoin e Mastercoin (una prima forma di Smart Contract), Buterin decise di proporre un protocollo alternativo per la realizzazione di applicazioni decentralizzate basate su un linguaggio Turing-completo implementato su una blockchain. Ethereum è dunque una piattaforma open source decentralizzata e basata su una rete peer-to-peer di calcolatori che implementa lo sviluppo di smart contracts e di applicazioni decentralizzate. Questa struttura permette una decentralizzazione nel momento in cui ciascun utente implementa una macchina virtuale (EMV) per sincronizzarsi nella blockchain. Tale modello permette una distribuzione di server e di potenza di calcolo garantendo una maggiore sicurezza, flessibiltà ed affidabilità. Ethereum è dunque costruito su una blockchain la quale è estremamente potente ed ha un'infrastruttura globale. I nodi interagiscono nel sistema attraverso le transazioni, quest'ultime sono il punto centrale e permettono al sistema di generare una sequenza corretta e validata di blocchi che registrano e salvano le informazioni.
Poco dopo la nascita di Ethereum, il cofondatore Gavin Wood pubblicò uno *yellowpaper*[13] in cui estese il *whitepaper* di Buterin andando a descrivere nello specifico i comportamenti del sistema. Quest'utlimo infatti può essere considerato come una macchina a stati basata su transazioni. Partendo da una fase di stato iniziale si possono eseguire incrementalmente nuove transazioni per portare lo stato ad una forma finale (come è possibile notare in figura 2.1). Quest'ultimo include informazioni quali saldo dei

conti, reputazione, accordi di fiducia e dati relativi ad informazioni reali.

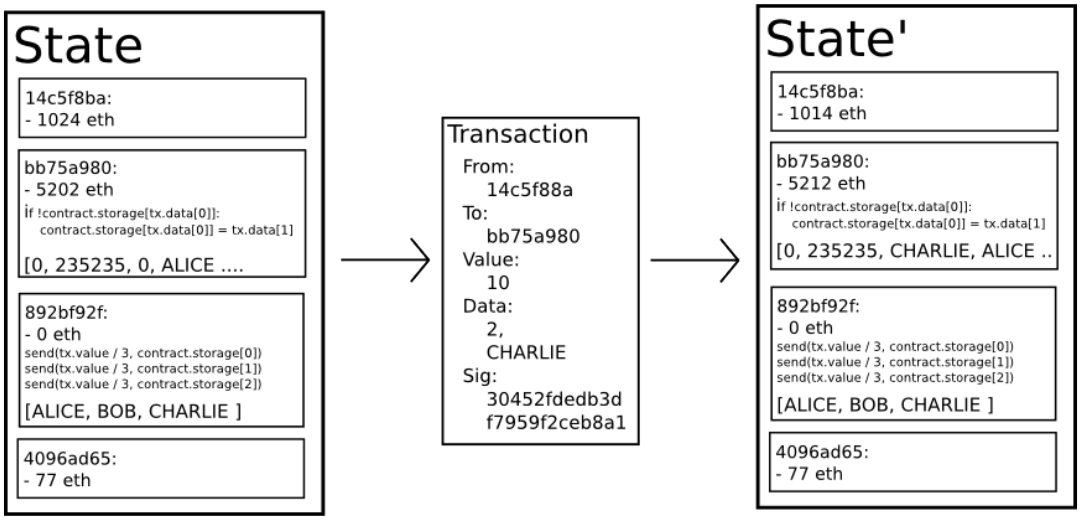

Figura 2.1: Cambio di stato[11]

2.0.2 Ether

Vitalik introdusse nel suo sistema una nuova criptovaluta chiamata *Ether* (ETH). Una criptovaluta è un mezzo di scambio digitale utilizzato su internet basato su funzioni crittografiche, rendendo possibili transazioni tra due o più parti. L'ether viene utilizzato per pagare le commissioni in Ethereum, rappresenta un *"Crypto-Fuel"* ossia una sorta di carburante che alimenta il sistema Ethereum. L'Ether può essere paragonato al Bitcoin con la differenza che può essere anche il sistema di pagamento interno ad Ethereum.
Nella Figura 2.2 è rappresentato l'andamento di Ether dal 2015 al 2020.

2.0.3 Ethereum Virtual Machine

L'*Ethereum Virtual Machine (EVM)* è una macchina virtuale strutturata a *stack* ed è progettata per essere l'infrastruttura per l'esecuzione del bytecode di smart contract basati su ethereum. Per Macchina Virtuale, si intende un software che simula in tutto e

Figura 2.2: Andamento del prezzo ETH[17]

per tutto una macchina fisica attraverso un processo di virtualizzazione. L'EMV rende possibile l'esecuzione effettiva dei programmi su una blockchain. Possiamo pensarla come una macchina virtuale quasi-turning completa in quanto le computazioni performanti della macchina sono vincolate dal Gas. Quest'ultimo è un'unità che misura la quantità di sforzo computazionale necessario per l'esecuzione di specifiche operazioni.

2.0.4 Smart Contracts

L'idea di fondo quando si parla di *smart contracts*[4] è quella di pensarli come dei contratti nel mondo reale con l'unica differenza che sono digitali. Facendo chiarezza, un contratto rappresenta un accordo tra due o più parti per costruire, regolare o estinguere un insieme di clausole che definiscono il contratto. Uno smart contract è un contratto auto-eseguito in cui i termini dell'accordo sono scritti in un linguaggio di programmazione specifico (*Solidity*). Rappresenta dunque una serie di istruzioni che mettono in accordo due o più parti del sistema garantendo sicurezza ed immutabilità durante l'esecuzione. La creazione degli smart contracts in Ethereum avviene tramite un codice scritto nel linguaggio di programmazione chiamato Solidity, il quale è un linguaggio ad alto livello orientato ad oggetti; il codice generato, viene eseguito dall'Ethereum Virtual Machine.
L'implementazione degli smart contracts è dunque rivoluzionaria se si pensa all'impatto che possono avere nella vita di tutti i giorni, a partire dall'ambito assicurativo fino a quello finanziario. Nello specifico, gli smart contracts sono degli script eseguibili senza possibilità di censura o manipolazione che risiedono nella blockchain. Nel momento in cui essi vengono trasmessi in quest'ultima, assumono un *address* univoco. L'esecuzione

degli smart contracts viene attivata associando il codice a delle transazioni garantendo immutabilità e distribuibiltà. Sono immutabili in quanto nel momento in cui ne viene creato uno, esso non può più essere modificato e sono distribuibili in quanto l'output generato da essi è convalidato dai nodi del client per evitare manomissioni sul risultato finale. Essi hanno l'obiettivo di facilitare operazioni tra persone ed istituzioni garantendo sia un sistema decentralizzato sia transazioni sicure tra parti che non si conoscono, evitando l'intervento di intermediari. Essendo basati su una blockchain, vi è la garanzia che tutte le transazioni vengano effettuate in maniera sicura e trasparente, ovviamente tale transazione ha validità nel momento in cui i punti stipulati nel contratto vengano rispettati. L'unione di più smart contracts può dare vita a quella che viene definita *dApp*, ossia applicazione decentralizzata che verrà analizzata successivamente.

2.0.5 Account

In ethereum un *account* è un oggetto costituito da un *address* di 20-byte ed è colui che funge da inviante e da ricevente in una transazione, quest'ultima aggiorna direttamente il bilancio (*balance*) e ne mantiene lo stato. Una transazione permette uno scambio di valori, messaggi e dati tra più account.
Un account contiene quattro campi:

- Nonce rappresenta un contatore per essere sicuri che ogni transazione sia processata una sola volta;
- Il bilancio attuale di Ether;
- Il codice del contratto (se presente);
- Il deposito (vuoto di default).

Esistono due tipi di account: l'*Externally Owned Accounts(EOA)* ed il *Contract Account(CA)*. Il primo è controllato da una chiave privata, mentre il secondo è controllato dal codice dello smart contract e può essere attivato solo da un EOA. L'Externally Owned Account non possiede un codice ed è necessario per far parte della rete Ethereum. Interagisce con la blockchain usando transazioni le quali rendono possibili l'invio di messaggi ad altri account se create o firmate. Nel Contract Account invece, ogni volta che il contratto riceve un messaggio ne viene attivato il codice permettendo ad esso di leggere

e scrivere internamente al deposito o di inviare altri messaggi o creare contratti. Per concludere, un account ethereum ha la possibilità di gestire lo stato di appartenenza di ether, le transazioni consistono nello spostamento di tale stato da un account all'altro.

2.0.6 Transazioni

Quando si parla di *transazioni* nel mondo Bitcoin, si fa riferimento al semplice scambio di criptovalute tra due parti. Le transazioni in Ethereum, invece, estendono tale concetto in ulteriori possibilità. Vi è come in bitcoin, la possibilità di effettuare un trasferimento di valore ed è la più semplice forma di transazione di Ether tra account. Un altro tipo di transazione è relativo invece alla creazione di nuovi smart contracts includendone il codice nella transazione. Per finire, una transazione può essere utilizzata per attivare uno smart contract. Le transazioni sono dunque una forma di comunicazione effettiva tra i vari account, come scrisse Butarin nel suo whitepaper con il termine "transazione" si intende un pacchetto di dati firmati, contenenti un messaggio da inviare da un account esterno. Le transazioni sono strutturate nel seguente modo:

- Il destinatario del messaggio;
- Una firma che identifica il mittente;
- L'ammontare di ether e di dati da inviare al destinatario;
- Un valore *STARTGAS*, che rappresenta il massimo numero di steps computazionali che l'esecuzione della transazione può impiegare;
- Un valore *GASPRICE*, che rappresenta la commissione che il mittente paga per lo step computazionale.

Per contenere il problema di loops infiniti, accidentali o uno spreco computazionale nel codice, Buterin introdusse due nuovi concetti: STARTGAS e GASPRICE. Per STARTGAS si intende un valore che rappresenta il massimo numero di passi computazionali effettuabili da una transazione. Ciascuna transazione necessita inoltre di un costo chiamato *GAS*(o GASPRICE) da intendere come il carburante che serve ad un veicolo per potersi spostare. Per GAS si intende una sorta di commissione ad un miner che approva la transazione aggiungendola al ledger della blockchain. L'idea di fondo è quella di far

pagare agli utenti i costi computazionali (dall'energia alla CPU) necessari per eseguire, creare ed approvare le transazioni. Il GAS può essere acquistato in cambio di Ether ed è necessario per il corretto svolgimento di un contratto, va specificato all'inizio di esso e non può essere ricaricato durante un l'esecuzione. Se questa termina, terminerà anche il contratto.

2.0.7 Token

I token vivono nella blockchain ethereum, come anticipato la crittovaluta nativa di ethereum è Ether, oltre agli ether è possibile supportare altri tokens che hanno un funzionamento simile ad una valuta. Possono rappresentare una valuta, un'azione di una società o una certificazione. Per ottenere dei token si necessita l'invio di ether allo smart contract il quale restituisce una determinata quantità di token. Ogni token può essere diverso dall'altro, a discrezione dello smart contract che lo crea. Se tale scenario dovesse presentarsi ci si ritroverebbe in un sistema in cui la comunicazione via token tra smart contract dei nodi della rete ethereum risulterebbe complessa e costosa in termini di tempo. La community decise di estendere il concetto di token, presentato nella prima versione di Ethereum, propondendo quello che è conosciuto come token ERC20 ossia un token standard che presenta delle caratteristiche funzionanti in qualunque smart contract appartenente alla rete Ethereum.

Token ERC20

Il termine *ERC20*[8][10] sta per *"Ethereum Request for Comments"* ovvero "richiesta di commenti ethereum". Rappresenta una linea guida o standard quando si vuole creare un proprio token e definisce sei funzioni che lo smart contract deve implementare e tre opzionali. Per quanto riguarda le funzioni opzionali vengono descritte come segue:

- **Nome**: Rappresenta il nome del Token;
- **Simbolo**: Rappresenta il nome abbreviato del Token;
- **Decimals**: Rappresenta il numero di cifre decimali che supporta (massimo 18).

Per quanto riguarda le funzioni obbligatorie, esse vengono rappresentate come segue:

- **totalSupply**: Definisce il numero massimo di token sostenibili dal sistema;
- **transfer**: Prende una determinata quantità di token da totalSupply ;
- **approve**: Controlla che lo smart contract possa garantire il trasferimento di una determinata quantità di token ad un utente controllando il total supply dell'utente;
- **transferFrom**: Trasferisce tokens tra due utenti che ne possiedono;
- **balanceOf**: Restituisce il numero di token appartenenti ad un dato address;
- **allowance**: Controlla che un utente possa effettivamente inviare una quantità di token ad un altro.

2.0.8 Dapp

Le Dapp sono applicazioni decentralizzate eseguibili in una rete di computer peer-to-peer. Rappresentano una novità proposta da ethereum, in quanto è un'applicazione strutturata su una blockchain ma con un'UI che permette agli utenti di collegarsi ed utilizzare gli smart contracts implementati. Essendo decentralizzate non hanno bisogno di una figura intermedia che le gestisca in quanto sono autonome ed auto-gestite. Inoltre si elimina il problema di avere una struttura centralizzata che si basa su un unico server superando il problema del single point of failure.

2.1 Payment Channel

2.1.1 Problema della scalabilità

Per scalabilità si intende la capacità di un sistema di aumentare o diminuire di scala in funzione delle necessità e delle disponibiltà, il sistema che gode di tali proprietà viene detto scalabile. Le transazioni si pongono alla base del problema della scalabilità[15] nel mondo delle crittovalute. Al momento ethereum elabora circa 15 transazioni al secondo mentre Visa ne elabora 45.000 al secondo. Con il crescente utilizzo di tali piattaforme

decentralizzate, si è arrivati ad un rallentamento della rete con conseguente crescita del GAS. Il limite principale è che le blockchain pubbliche come ethereum richiedono che ogni transazione sia elaborata da ogni singolo nodo della rete, ogni operazione che si svolge sulla blockchain deve essere eseguita da ogni singolo partecipante. Ciò pone un limite fondamentale nelle transazioni ossia che la richiesta del lavoro non può essere superiore a quello che supporta un singolo nodo. Si potrebbe presentare la situazione in cui si richiede ad ogni singolo nodo di fare più lavoro di quello supportato. Se ad esempio dimensione del blocco (cioè GASLIMIT) rappoppiasse singnificherebbe richiedere più lavoro ai nodi, ed i computer meno potenti abbandonerebbero la rete, facendo diventare il mining un processo sempre più centralizzato. Serve dunque una soluzione che risolva tale problema senza aumentare il carico dei nodi.

2.1.2 Funzionamento del payment channel

Il *Payment Channel* nasce come possibile soluzione al problema della scalabilità di una blockchain.
"*Un albero che cade nella foresta fa rumore anche se non c'è nessuno in ascolto?*" è la chiara metafora utilizzata nel paper[12] in cui venne introdotto il concetto di payment channel. Il concetto alla base è quello di capire se è necessario rilevare avvenimenti non osservati. Se nessuno sente la caduta dell'albero o se ha emesso o meno un suono, non ha alcuna conseguenza, allo stesso modo se una transazione avviene più volte e costantemente non è necessario salvarla tutte le volte nella blockchain ma è necessario salvare solo alcune informazioni importanti. Questo viene fatto per evitare un bottleneck delle transazioni nella rete.
La soluzione proposta è dunque quella di creare un sistema in grado di gestire i pagamenti off-chain tra due o più parti. Nel caso in cui due parti necessitino di un canale di pagamento, una delle due effettua una prima transazione nella blockchain in cui inserisce delle valute in deposito, si impegna dunque verso la seconda parte offrendo una garanzia. La creazione di questo deposito è necessaria per non introdurre nel sistema terze parti che possano creare situazioni d'attrito, costi di mantenimento o di transazione. Una volta accettate le condizioni iniziali, il sistema si sposta fuori dalla blockchain (*off-chain*) e le transazioni vengono effettuate tramite uno scambio di messaggi firmati. Al termine del processo di trasferimento dati, una delle due parti decide di chiudere il payment channel

inserendo il riepilogo finale delle trasazioni nella blockchain.

Con *Payment Channel o Micropayment channel* si intende una classe di tecniche svilppate per peremttere agli utenti di effettuare molteplici transazioni senza inserirle tutte nella blockchain. In un payment channel solo due transazioni sono aggiunte alla blockchain, quella di apertura del canale e quella di chiusura, tra queste due parti possono avvenire molteplici transazioni chiamate off-chain.
Tali transazioni risultano più veloci, non hanno bisogno di GAS e non sono visibili nella blockchain. In figura 2.3 è rappresentato il funzionamento di un payment channel tra due nodi.

2.1.3 Payment channel network

Per ridurre i costi ed i tempi di apertura di un payment channel tra più parti, viene in aiuto il concetto di payment channel network, rappresentato in figura 2.4. Si tratta infatti di una combinazione di più payment channel strutturati a rete che connettono tutti i nodi del sistema. Tale implementazione abilita le transazioni tra i partecipanti della rete che non hanno un canale aperto tra loro. Il tutto avviene purchè i nodi che collegano due parti abbiano un payment channel attivo. Supponiamo che Alice abbia un canale aperto con Bob, il quale a sua volta ne ha uno collegato a Charlie. Nel caso in cui Alice voglia effettuare delle transazioni con Charlie, può evitare di aprire un channel diretto con lui utilizzando Bob come intermediario (il quale riceverà una commissione). Ciò riduce costi e velocizza il sistema ma necessita di intermediari che abbiano abbastanza fondi nei payment channels.

2.1.4 State Channel

Gli *state channel*[14] sono una generalizzazione dei payment channel che vanno ad estenderne il funzionamento. Infatti la differenza principale è che non riguardano solo le transazioni tra due entità ma anche la modifica dei dati relativi agli stati del sistema. Gli utenti di uno state channel possono eseguire complessi smart contract *off-chain*. Immaginiamo di avere due users nel sistema. Questi stabiliscono uno state channel e ne mantengono un *ledger* simulato per il contratto. L'esecuzione di alcune operazioni di quest'ultimo non verranno registrate sulla blockchain. Questa fase rimane attiva finchè non si entra in una ipotetica fase di conflitto. Se dovesse accadere infatti, ciascuna delle

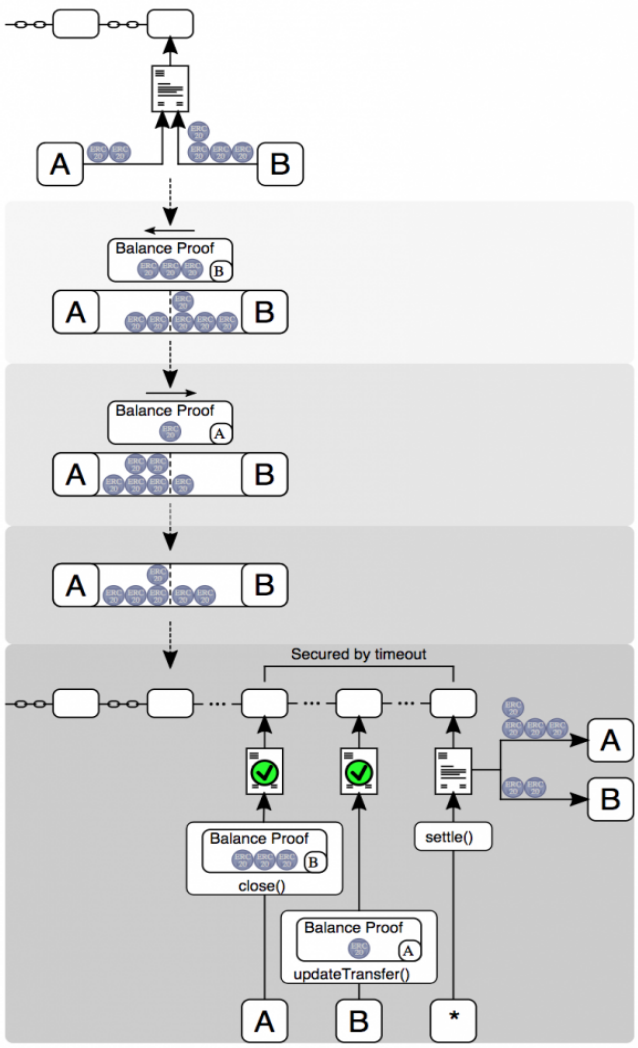

Figura 2.3: Funzionamento di un payment channel[16]

due parti può interrompere lo state channel andando ad inserire l'ultima operazione nella blockchain.
Il funzionamento può essere così descritto:

- Gli utenti si spostano off-chain ed inviano i token ad un contratto tramite firma, quest'ultimo ha la capacità di ripagare tutte le parti al termine del processo.
- Un utente firma le transazioni o messaggi e li invia ad un altro utente. Ognuno di essi fa una copia della firma dell'inviante.

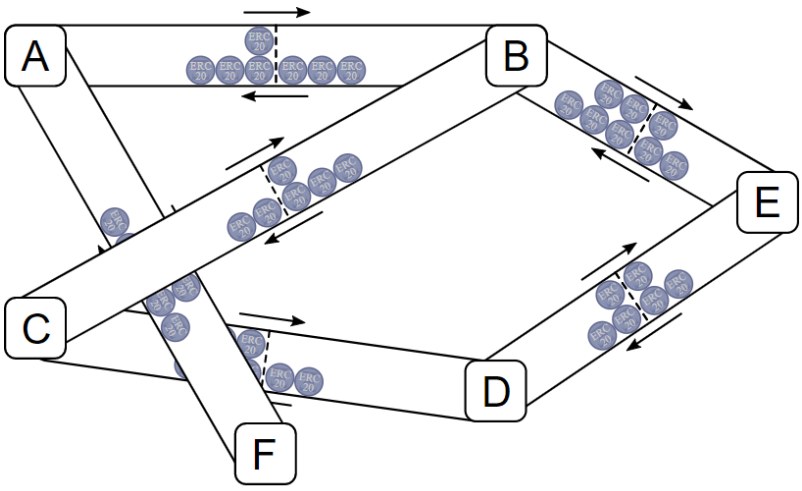

Figura 2.4: Rappresentazione di un network di payment channel[16]

- Ogni transazione contiene un nonce, in questo modo lo smart contract è in grado di stabilire un ordine cronologico delle transazioni.
- Il canale viene chiuso con conseguente inserimento dell'ultima transazione nella blockchain.
- Lo stato viene aggiornato e sbloccato, lo smart contract invia i token finali alle relative parti.

Gli state channel sono stati implementati meno rispetto ai payment channel. Raiden è uno dei pochi progetti che implementa e supporta sia payment channel che specifici protocolli per formare un network di state channel. Infatti consente trasferimenti di token ERC20 tra partecipanti senza la necessità del consenso globable. Il punto forte di Raiden è che non è necessario un canale di pagamento diretto tra due utenti, l'importante è che siano collegati tramite altri nodi, generando uno spostamento di dati indiretto.

Capitolo 3

Progettazione

In questo capitolo si andranno a definire tutte le fasi progettuali svolte per la realizzazione di questo progetto. Verranno divise in specifiche, suddivisione del lavoro, componenti del sistema, tecnologie e strumenti. La premessa è quella di simulare un contesto in cui vegano implementate metodologie di pagamento decentralizzato utilizzando le tecnologie descritte nei capitoli precedenti.

3.1 Specifiche

Lo scopo di questo progetto è quello di simulare un sistema in cui chiunque attraverso il proprio dispositivo possa connettersi ad una rete Wi-Fi a pagamento tramite microtransazioni gestite da smart contract. Il sistema è stato pensato per una città in cui sono installati un insieme di router indipendenti che permetteno la completa copertura della superficie della città.
Posto che il dispositivo abbia più interfacce di rete implementate, lo scopo è quello di consentire ad un utente di connettersi ad un router. Per permettere ciò, l'user deve effettuare delle transazioni verso il router, il quale ne abiliterà l'accesso tramite una password evitando preconfigurazioni o autorizzazioni per l'utilizzo di esso, permettendone poi, una connessione il più stabile possibile. Tutto ciò può essere ottenuto utilizzando degli smart contracts che permettano di regolamentare uno scambio di valori per creare le condizioni in cui una persona può connettersi ad una rete wi-fi pagando delle microtransazioni per una data unità di tempo. Da una parte l'utente pagherà per il tempo di utilizzo della

connessione, dall'altra, il router ne permetterà l'uso rendendo disponibile una password specifica.
Il sistema si estende nel momento in cui una persona si allontana dalla copertura della rete a cui è connesso, in questo caso infatti è possibile creare un altro smart contract con un ulteriore router andando dunque a creare un sistema di copertura il più stabile possibile ed evitando all'utente di perdere denaro e connessione.

3.2 Suddivisione del lavoro

Una volta impostate le specifiche di tale sistema è possibile suddividere il lavoro svolto in due parti.

- Implementazione di Smart Contracts: Una serie di smart contracts implementati per gestire le richieste di pagamento on-chain ed off-chain tramite l'utilizzo di token ERC20. Essi vengono distinti come segue:
  - ERC20: Rappresenta l'interfaccia standard per token ERCO20 che fa da schema al token effettivo utilizzato nel sistema;
  - WiFiToken: Rappresenta l'implementazione dello standard ERC20 in un token sfruttabile all'interno del sistema;
  - PaymentChannel: Implementazione di un payment channel nel sistema, è la struttura su cui si basa il progetto in quanto permette di regolare tutti i pagamenti del sistema.
- Simulazione del comportamento degli attori: Attraverso Nodejs è stata simulata una comunicazione tra due nodi in modo tale da poter creare un caso d'uso più realistico possibile.

3.3 Componenti del sistema

Gli attori principali che compongono il sistema sono i seguenti:

- User Device: Interpretato come client, rappresenta il dispositivo dell'utente che ricerca le reti disponibili e ne attiva una connessione attraverso l'apertura di un payment channel;

- Router: Interpretato come Server, rappresenta il router in attesa di richieste di connessione. È il secondo attore del sistema ed è il destinatario delle transazioni. Una volta aperto il payment channel si occupa di convalidare i messaggi inviati dal client e di chiudere il canale quando richiesto;
- Payment Channel: È lo smart contract che gestisce il tutto, al suo interno troviamo metodologie di apertura e chiusura del canale di pagamento e di verifica delle condizioni necessarie al corretto svolgimento del sistema;
- WiFiToken: È un token ERC20, ed è l'oggetto di scambio tra client e server.

A seguire, sono rappresentati il diagramma di sequenza (figura 3.1) del sistema relativo alla comuicazione dei nodi con lo smart contract e ed una rappresentazione grafica del progetto(figura 3.2).

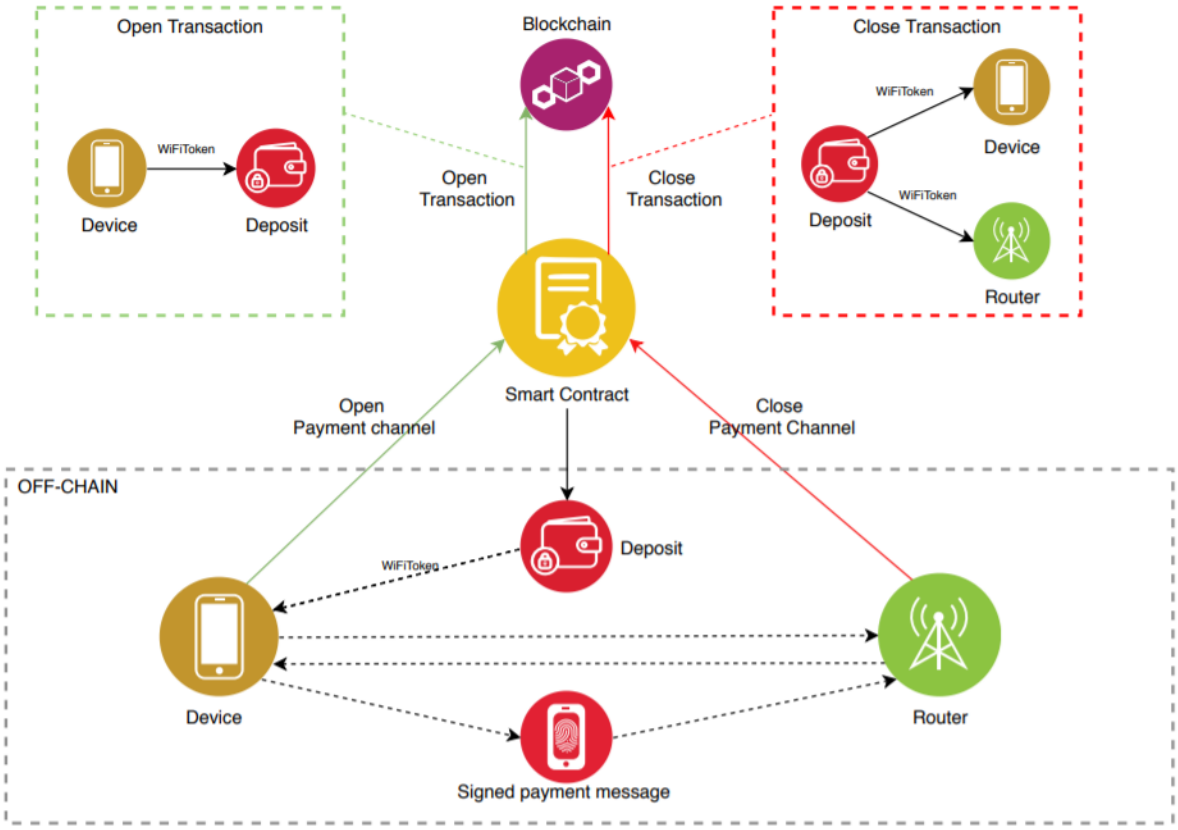

Figura 3.1: Rappresentazione schematica del payment channel applicato al progetto

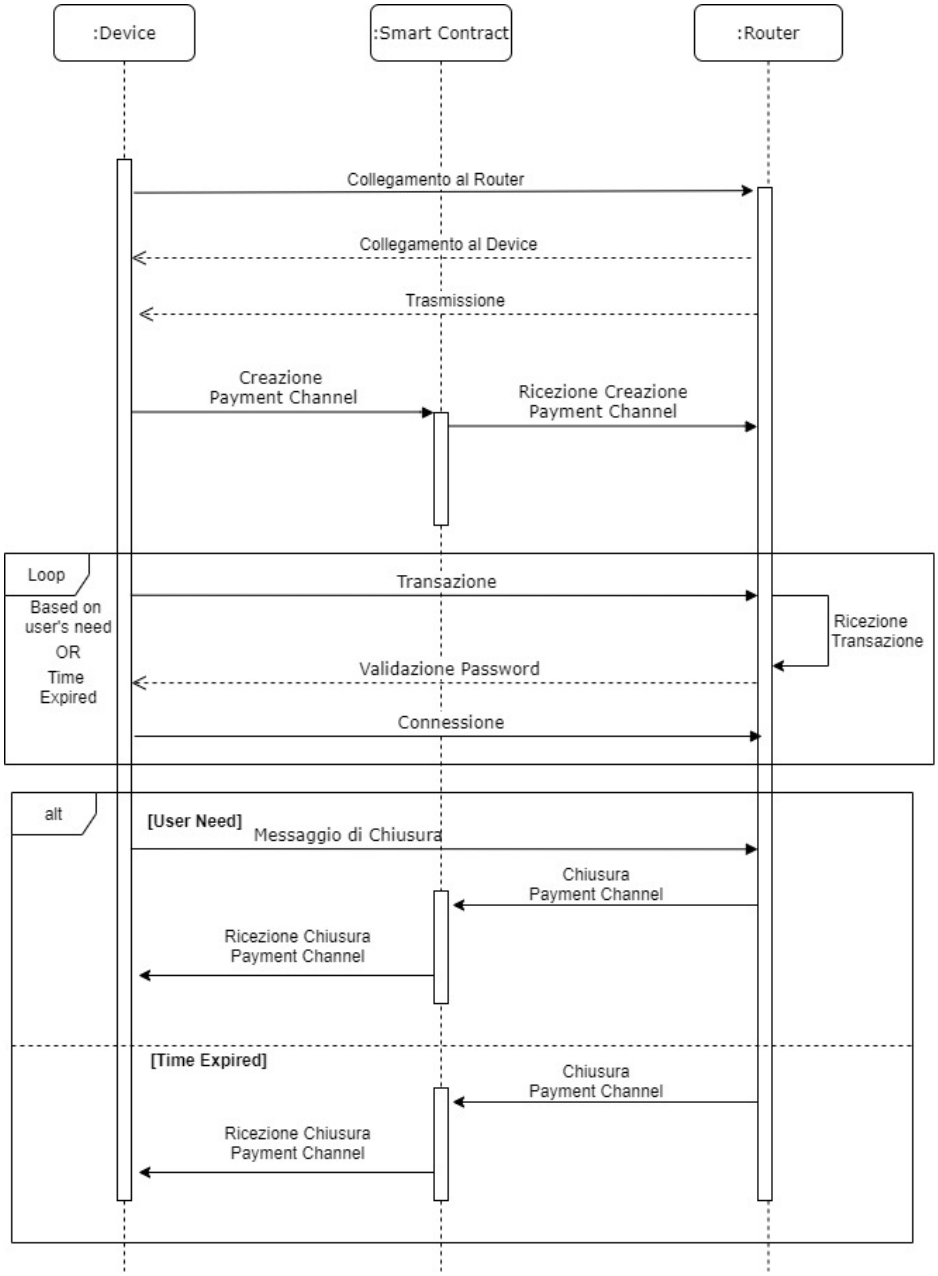

Figura 3.2: Diagramma di sequenza del payment channel

3.3.1 Comportamento del Device

In questo sottoparagrafo verranno analizzate le possibili operazioni che un Device può svolgere all'interno del sistema. Per device si intende il componente utilizzato dall'utente

per poter comunicare con il Router. Nella figura 3.2 è possibile vedere le operazioni svolte dal device e dal router e sono descritte a seguire.
Le operazioni effettuabili dal Router sono:

- Collegamento Al Router: Il device si collega al sistema, i primi passi che effettua sono il collegamento al provider in locale di Web3 ed al Router (server) tramite l'indirizzo IP ed una porta predefinita. La connessione a Web3 è necessaria per simulare una blockchain etheruem in locale e per analizzare il comportamento dei nodi in essa, andando a ricreare in locale tutte le transazioni relative, in questo caso, agli smart contracts implementati.

- Ricezione: Una volta collegato, il device, riceve la tariffa proposta dal router.

- Creazione Payment Channel: L'utente decide di aprire un payment channel con il router, richiamando la funzione relativa alla creazione del canale scritta nello smart contract PaymentChannel, in questa fase viene stabilito l'intero balance (deposito) utilizzabile nel canale da entrambi gli attori. L'inserimento di tale deposito è necessario per definire l'intero importo utilizzabile dal device. Quest'ultimo invia al router un messaggio in cui specifica il deposito usufruibile, il blocco di riferimento ed il suo address.

- Transazione: L'utente decide dunque di voler pagare il server, così facendo si aspetta la password valida per potersi connettere al router. Per farlo invia un messaggio in cui è presente la quantità di token da trasferire firmati dall'utente che inoltra la transazione.
  L'operazione di transazione può essere ripetuta affinchè il balance resti positivo. I passaggi di transazione avvengono fuori dalla blockchain per limitare il problema della scalabilità relativo alle transazioni on-chain. è necessario dunque definire un sistema off-chain che gestisca i trasferimenti in modo sicuro. Ciò viene definito da degli algoritmi di crittografici per firmare dei messaggi in cui sono contenuti i dettagli di una transazione.

- Connessione: Se la fase precedente va a buon fine, allora il device si aspetta che il server gli invii una password per potersi connettere. La connessione rimane attiva fintantoché l'utente continui ad inviare microtransazioni verso il router.

- Ricezione Chiusura Payment Channel: La fase di chiusura effettiva avviene nel server che deciderà se chiudere o meno il canale, ciò avviene se l'utente non continua il pagamento dopo una certa unità di tempo o se l'user decide di interrompere il pagamento inviando un semplice messaggio di avviso al Router. La funzione relativa alla chiusura del canale viene gestita dallo smart contract PaymentChannel il quale termina le operazioni off-chain andando ad inserire nella blockchain l'ultima transazione aggiornata dopo aver effettuato dei controlli relativi alle firme dei messaggi inviati nella fase di transazione.

3.3.2 Comportamento del Router

Per quanto riguarda il Router esso è rappresentato come un nodo in ascolto e in comunicazione con il device utilizzato dall'utente.Nella figura 3.2 è possibile vedere le operazioni svolte dal device e dal router e sono descritte a seguire.
Le operazioni effettuabili dal Router sono:

- Collegamento Al Device: Il Router si collega al sistema, i primi passi che effettua sono il collegamento al provider in locale di Web3 ed al Device (client) nel momento in cui questi ne richiede la connessione. La connessione a Web3 è necessaria per simulare una blockchain etheruem in locale e per analizzare il comportamento dei nodi in essa, andando a ricreare in locale tutte le transazioni relative, in questo caso, agli smart contracts implementati.

- Trasmissione: Una volta collegato, il Router, invia la propria tariffa al device.

- Ricezione apertura canale: Il Router viene notificato della creazione del payment channel. Successivamente riceve le informazioni relative a quest'ultimo tramite un messaggio del device. Da questo momento il sistema si sposta off-chain limitando l'inserimento nella blockchain di transazioni ripetute, risparmiando in termini di costi per transazione e per tempo di attesa relativo al caricamento di queste.

- Ricezione Transazione: Il Router riceve un messaggio firmato con i dettagli della transazione per ottenere una password. Procede dunque con una fase di controllo relativo alla firma del massaggio. Una volta recuperati i dati ed effettuato il controllo, procede all'invio della password al client facendo partire un timer relativo

al tempo di utilizzo; fintantochè l'utente invia transazioni verso il server, questo si impegna a convalidare la propria password.

- Chiusura Payment Channel: In questa fase il Router procede alla chiusura del payment channel. Questa può accadere se scade il timer impostato nella fase precedente o se il client decide di interrompere il pagamento. Il sistema ritorna dunque on-chain andando ad inserire l'ultima transazione aggiornata nella blockchain tramite la funzione di chiusura descritta nel paymentChannel, una volta controllati i parametri relativi alle firme delle parti, lo smart contract elimina dal sistema il canale ed attraverso il contratto WiFiToken trasferisce i token aggiornati al router ed al device (in quest'ultimo se il balance non è stato esaurito completament).

Capitolo 4

Implementazione

L'implementazione di tale progetto è presente nella seguente repository `https://github.com/giovanniPi997/paymentchannelTESI`.
Come anticipato nel capitolo precedente, il progetto deve simulare un sistema in cui due entità, Utente e Router, comunichino tra di loro effettuando delle microtransazioni tramite token. Poichè Ethereum supporta 15 transazioni al secondo è necessario trovare una soluzione che vada a limitare il più possibile le trasazioni nel sistema presentato. Tale modello proposto, in un caso reale, richiederebbe un numero considerevole di transazioni per richiesta di connessione, alimentando costi e tempi d'esecuzione. Si è cercato di risolvere il problema basando il sistema sui payment channels. Come spiegato nel capitolo 1.4, il payment channel è una tecnica che permette di spostare le transazioni off-chain.
La valuta implementata nel sistema è un token che segue lo standard di ERC20.

```
interface IERC20 {
    function transfer(address to, uint256 value) external returns (bool);

    function approve(address spender, uint256 value) external returns (bool);

    function transferFrom(address from, address to, uint256 value) external returns (bool);

    function totalSupply() external view returns (uint256);

    function balanceOf(address who) external view returns (uint256);
```

```
11
12     function allowance(address owner, address spender) external view
    returns (uint256);
13
14     event Transfer(address indexed from, address indexed to, uint256
    value);
15
16     event Approval(address indexed owner, address indexed spender,
    uint256 value);
17 }
```

Attraverso tale struttura è possibile implementare il token effettivo che verrà utilizzato nel progetto, il nome di tale token è WiFiToken (WFT). Quest'ultimo ha le funzionalità di un token ERC20 con la possibilà di essere *mintable* ed *ownable*, i token mintable possono essere creati in qualunque momento e aggiunti alla totalSupply ossia alla quantità di token posseduti. Con ownable, invece, si rende chiaro chi sia il proprietario di uno o più token presenti nel sistema.

4.1 Tecnologie e strumenti utilizzati

Per sviluppare una soluzione alla proposta descritta nelle Specifiche, è necessario creare un ambiente di sviluppo in grado di poter simulare nel modo più veritiero possibile tutte le azioni necessarie per poter giungere ad un modello ottimale. A seguire verranno brevemente descritte tutte le tecnologie e strumenti utilizzati:

4.1.1 Solidity

Il linguaggio di programmazione utilizzato per programmare smart contract è Solidity[18]. è un linguaggio ad alto livello ed è orientato ad oggetti. Le sue influenze derivano da linguaggi quali C++, Pyton e JavaScript. Il codice scritto verrà poi eseguito dall'Ethereum Virtual Machine (EVM).

4.1.2 Truffle

Truffle[22] è un ambiente di sviluppo e framework volto ai test. Il suo obiettivo è quello di semplificare le azioni e le operazioni effettuate in un sistema Ethereum. È possibile

compilare, collegare ed effettuare un deploy di Smart Contracts. Una delle prime operazioni da effettuare usando truffle è quella di fare una "migrazione" tra i vari contratti implementati nel sistema, rendendo operativi gli smart contracts una volta che si entra nel fulcro dell'applicazione. è possibile effettuare operazioni sui contratti direttamente da console. Inoltre è possibile testare degli script che implementano alcune funzioni degli smart contracts.

4.1.3 Ganache

Ganache[20] è un software personalizzato che permette di simulare in locale una blockchain Ethereum. è un ambiente di sviluppo in cui è possibile testare, effettuare un deploy e sviluppare applicazioni. Una volta attivato vengono generati 10 account con un balance di 100ETH ciascuno. Aggiungendo il "truffle-config.js" al Workspace è possibile collegare un progetto truffle a Ganache. Così facendo ogni operazione svolta nell'applicazione dallo Smart Contract verrà visualizzata nella Home del sistema.

4.1.4 Web3.js - Ethereum JavaScript API

Web3.js[19] è una collezione di librerie (Javascript e Pyton) che permettono ad un utente di interagire in locale o remoto con un nodo ethereum implementando connessioni HTTP O IPC, in questo progetto è stata implementata la connessione HTTP.

4.1.5 Node.js

Node.js[**?**] è una framework per la realizzazioni di applicazioni web in JavaScript. Si basa sul JavaScript Engine V8.

4.2 Connessione a Ganache

Il primo passaggio per la realizzazione di questo progetto, è quella di impostare un ambiente il più simile e coerente possibile a quello di ethereum. Per farlo è stato utilizzato Ganache. Come spiegato nei capitoli precedenti, Ganache è un software che permette la creazione di una blockchain in locale. La sua interfaccia è rappresentata nella seguente figura 4.1.

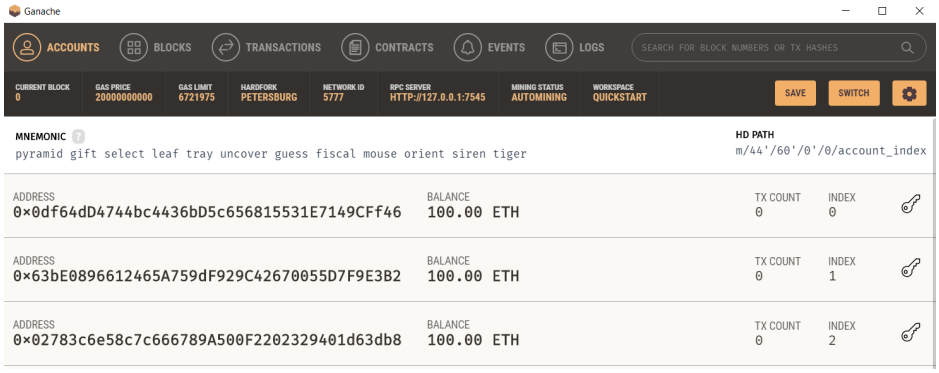

Figura 4.1: Interfaccia Ganache [20]

Per permettere ai nodi di poter comunicare con ganache, si è impostata un RPC Server in locale con porta 7545. Andando nello specifico, entrambi i nodi implementeranno le seguenti linee di codice per poter stabilire una connessione con ganache:

```
const web3Provider = 'ws://127.0.0.1:7545';

web3 = new Web3(web3Provider);
accounts = await web3.eth.getAccounts();
myAccount = accounts[accountNr];
web3.defaultAccount = myAccount;
```

In questo modo le informazioni degli address generati da ganache verranno copiati su entrambi i nodi.

4.3 Connessione Client-Server

Una volta stabilita la connessione dei nodi a ganache tramite la libreria web3.js, è possibile passare all'implementazione di un canale di comunicazione tra client e server. Poichè il sistema di comunicazione è in locale, si userà lo stesso ip usato in ganache, cambiandone però la porta, ossia 7546.
Per impostare un canale tra client e server, è stato deciso di utilizzare la libreria socket.io la quale abilita le comunicazioni in tempo reale, bidirezionalmente e basandole su eventi. Per quanto riguarda il lato client avremo dunque:

```
var socket = require('socket.io-client')('http://127.0.0.1:7546');
```

Mentre per il lato server avremo:

```
var server = http.createServer(function(req, res){ });
server.listen(7546);
var socket = io.listen(server);
```

in questo modo il sistema riconosce due nodi, un server in ascolto ed un client , entrambi comunicanti sulla stessa porta, ossia 7546.

Una volta definiti i protocolli di connessione dei due nodi, è necessario collegarli. Poichè il server è in ascolto, è possibile definirlo come in uno stato di attesa di un evento da parte del client, in questo caso l'evento richiesto è per la connessione. Dunque il client implementa la seguente funzione per stabilire una connessione con il server.

```
socket.on('connect',function() {});
```

Una volta connessi i due nodi, è possibile passare ad uno scambio di messaggi. In questo caso, si tratta di uno scambio di informazioni tra il Device ed il Router. Il primo invia il proprio address ethereum, mentre il secondo invia sia il proprio address che il prezzo tariffario proposto per l'utilizzo della connessione wi-fi.

4.4 Creazione del payment channel

L'implementazione del payment channel è la struttura sulla quale si basa la regolamentazione delle transazioni tra due parti. In questo paragrafo si descrive la creazione di esso, andandone a definire i parametri e le condizioni necessarie perchè questa avvenga. La causa che ne implica la creazione è relativa alla scelta di una delle due parti di iniziare un canale di transazioni con l'altra. Nel caso specifico, avviene se l'user (sender) decide di voler connettersi con il router (receiver). I parametri necessari per la creazione di un payment channel sono l'address ricevente ed un deposito. Quest'ultimo ha un ruolo chiave poichè pone un limite massimo ed immutabile di token prelevabili durante l'attività del payment channel. L'inserimento dei due parametri è necessario per la creazione dell'oggetto relativo al canale, questo infatti viene inserito nel mapping dei canali, ed avrà come chiave il risultato della funzione keccack256 avente come parametri l'address

dell'inviante, l'address del destinatario ed il numero del blocco su cui risiede il payment channel.

La creazione di quest'ultimo, viene notificata ad entrambe le parti tramite un *event* nello smart contract:

```
event ChannelCreated( address indexed senderAddr,
                      address indexed receiverAddr,
                      uint32 indexed blockNumber,
                      uint192 deposit );
```

- address indexed senderAddr: Rappresenta l'address dell'inviante (Device);
- address indexed senderAddr: Rappresenta l'address del ricevente (Router);
- uint32 indexed blockNumber: Rappresenta il blocco in cui risiede il payment channel;
- uint192 deposit: Rappresenta il deposito dal quale è possibile prelevare i token.

Rappresenta l'ultima transazione registrata nella blockchain, le prossime transazioni avverranno off-chain andando così a risolvere il problema dell'elevato numero di transazioni al secondo registrate nella blockchain, limitando il tempo di attesa del sistema.

4.5 Transazioni off-chain

Come anticipato nel paragrafo precedente, da questo momento (fino alla chiusura del payment channel) le transazioni avverrano off-chain. È tuttavia necessario chiarire il concetto di transazioni che, in questo caso, si distacca leggermente dalla definizione esplicata nei capitoli precedenti. Nel momento in cui è stato aperto il payment channel, lo smart contract, perde di utilità per quanto riguarda le transazioni. Queste infatti, avvengono tramite uno scambio di messaggi a multi-firma tra device e router. è possibile infatti firmare messaggi anche se non si è nella rete ethereum, La firma e verifica dei messaggi firmati tramite l'algoritmo ECDSA consentono una comunicazione a prova di attacchi fuori dalla blockchain.

4.5.1 Lato client

Andando nello specifico, il client trasmette un messaggio al server dove specifica il suo balance (ossia la quantità di token inviati) ed un messaggio firmato. La funzione per firmare quest'ultimo è la seguente:

```
const senderHash = web3.utils.soliditySha3(
  {
    type: 'address',
    value: routerAccount
  },
  {
    type: 'uint32',
    value: blockNumber
  },
  {
    type: 'uint192',
    value: balance
  },
  {
    type: 'address',
    value: PaymentChannelAddr
  }
);
return await web3.eth.sign(senderHash, myAccount);
```

La fase relativa alla firma del messaggio è necessaria per poter soddisfare un pagamento in un contratto. è essenziale, tuttavia, garantire una protezione contro gli attacchi replay, in quanto significherebbe permettere il riutilizzo del messaggio per richiedere l'autorizzazione per una seconda volta. Per evitare ciò, si simula lo stesso meccanismo che utilizza ethereum nelle transazioni, ossia utilizzando il nonce (in questo caso è il blockNumber) che è un contatore che specifica il corretto ordine delle transazione. Lo smart contract dunque, utilizza il nonce per verificare se questo viene utilizzato più volte.
Per garantire una maggiore sicurezza viene inserito nel messaggio anche l'address dello smart contract *PaymentChannelAddr*, in tal modo verranno accettati solo i messaggi che contengono l'address del contratto stesso. Una volta definite le informazioni da inserire nel messaggio da firmare, è necessario impostarlo, crearne un hash e firmarlo. La struttura è quella rappresentata nel codice sovrastante, per quanto riguarda l'hash invece, la

libreria ethereumjs-abi fornisce una funzione chiamata soliditySHA3 che imita il comportamento della funzione keccak256 di solidity applicata agli argomenti da codificare. Attraverso la funzione sign è possibile firmare un dato attraverso l'algoritmo ECDSA. Si tratta dunque di una funzione che accetta un address proprietario ed un hash restituendo una firma, in questo caso è relativa al messaggio.
Il messaggio completo da inviare al server sarà dunque strutturato nel seguente modo:

```
var message = {
  balance: balance,
  signature: senderSign
}
```

Per dare una maggiore sicurezza alla comunicazione tra i nodi è stata implementata una libreria di socket.io che permette di cirfrare e decifrare i messaggi più importanti tra due nodi. In questo modo il client invia il messaggio al server con una maggior certezza.

```
encrypt('secret')(socket);

socket.emit('message', message);
```

4.5.2 Lato server

Per quanto riguarda il router, esso deve verificare che il firmatario del messaggio appena arrivato sia il device, deve dunque effettuare un ripristino dei dati per controllare la veridicità del messaggio.

Le firme ECDSA in ethereum sono costituite da tre parametri r,s e v. La libreria Web3 permette di concatenare questi tre elementi in un'unica stringa, cioè la firma che è stata ottenuta in precedenza.

In questo modo è possibile procedere alla fase di ricostruzione del messaggio a partire dalle informazioni che il client ha fornito al server.

```
const balanceHash = web3.utils.soliditySha3(
      {
        type: 'address',
        value: myAccount
      },
      {
        type: 'uint32',
```

```
        value: blockNumber
      },
      {
        type: 'uint192',
        value: balance
      },
      {
        type: 'address',
        value: PaymentChannelAddr
      }
    );
```

Proprio come nel client, si inseriscono le informazioni all'interno della funzione soliditySHA3 che imita il comportamento della funzione keccak256 di solidity. Una volta ottenuto l'hash, questo va inserito nella funzione di ripristino fornita da web3, ossia web3.eth.accounts.recover():

```
const accountRecovered = await web3.eth.accounts.recover(
      balanceHash,
      message.signature
    );

```

Attraverso la funzione di recover, verrà restituito un valore, questo deve essere uguale all'account inviante, ossia del device. Una volta accertata la verifica e l'uguaglianza tra l'account inviante e quello ricostruito dal server, quest'ultimo procede all'invio della password per permettere al device di connettersi alla rete wi-fi. Come spiegato nella progettazione, l'implementazione vera e propria di un sistema di connessione non è stata effettuata in questo progetto, vi è solo uno scambio di messaggi in cui il router definisce la password ed una variabile booleana che ne indichi lo stato. Se questa è *true* allora la password è disponibile, se è *false* allora non è valida. Una volta inviato il tutto, il server implementa un timer per regolare il tempo di utilizzo della password. Se prima del termine di esso non viene inoltrato nessun altro messaggio da parte del client, allora il router procede con la chiusura del payment channel.

4.5.3 Aggiornamento del balance

Le transazioni appena descritte possono essere ripetute fino all'esaurimento del deposito inizializzato dal client. Esso serve infatti per gestire la quantità di token utilizzabili nel sistema. Nel momento in cui il client invia un secondo messaggio contenente il balance, il server ha il compito di aggiornare quest'ultimo sommando il nuovo valore con il balance attuale:

```
    balance=balance+message.balance;
```

Così facendo viene memorizzato la quantità effettiva utilizzata dal client.

4.6 Chiusura payment channel

La fase di chiusura del payment channel serve a trasferire il risultato delle transazioni off-chain nella blockchain. La fase di chiusura può avere due cause, una è per scelta del client e l'altra per scelta del server.

- Il client può decidere di chiudere il payment channel in qualsiasi momento, se ciò dovesse accadere allora deve notificare tale scelta tramite messaggio al server, il quale inizierà la fase di chiusura.
- Il server implementa un timer nel momento in cui arriva un messaggio firmato contente il balance, andando a calcolare il tempo di utilizzo della password. Se allo scadere del timer il client non ha ancora inoltrato un messaggio per una seconda transazione, allora il server procede con la chiusura del payment channel.

Prima di poter procedere con la chiusura del payment channel è necessario determinare la firma del destinatario, in questo caso del router. Il procedimento relativo alla firma è uguale a quello del client, tuttavia cambiano alcuni parametri. Il codice che ne implementa il servizio è il seguente:

```
const signBalance = async () => {
 const receiverHash = web3.utils.soliditySha3(
   {
     type: 'address',
     value: otherAccount
   },
```

```
    {
      type: 'uint32',
      value: blockNumber
    },
    {
      type: 'uint192',
      value: balance
    },
    {
      type: 'address',
      value: PaymentChannelAddr
    }
  );
  return await web3.eth.sign(receiverHash, myAccount);
};
```

Nell'hash generato dal web3.utils.soliditySha3 troviamo i seguenti parametri:

- otherAccount: Rappresenta l'altro address partecipante al payment channel visto dal router, ossia il device;
- blockNumber: Rappresenta il blocco sul quale risiede il payment channel;
- balance: Rappresenta il bilancio finale aggiornato, è la somma di tutti i token inviati dal client
- PaymentChannelAddr: Rappresenta l'address dello smart contract.

Una volta generato l'hash è possibile creare la firma del messaggio tramite la funzione web3.eth.sign() così facendo otteniamo la firma del router. La funzione che gestisce la chiusura è racchiusa nel seguente codice:

```
const close = async senderSign => {
  const receiverSign = await signBalance();
  try {
    await PaymentChannel.methods
      .closeChannel(
        myAccount,
        blockNumber,
```

```
        balance,
        senderSign,
        receiverSign)
      .send(myAccountOptions);

  } catch (e) {
    console.log(e);
  }
};
```

è possibile notare cinque parametri inseriti nella funzione closeChannel(), essi sono descritti come segue:

- myAccount: Rappresenta l'address del router;
- blockNumber: Rappresenta il blocco sul quale risiede il payment channel;
- balance: Rappresenta il bilancio finale aggiornato, è la somma di tutti i token inviati dal client;
- senderSign: Indica la firma dell'utente;
- receiverSign: Indica la firma del Router.

Al termine della funzione closeChannel viene inserita una seconda funzione, ossia .send(), tramite tale funzione è possibile specificare chi implementa i metodi in questione, in questo caso ci riferiamo al router.

La funzione closeChannel vista precedentemente appartiene allo smart contract PaymentChannel. Essa si compone come segue;

```
function closeChannel(address receiver, uint32 blockNumber, uint192
    balance,bytes calldata senderBalanceSign,bytes calldata
    receiverBalanceSign) external {

        address senderAddr = extractBalanceProofSignature(
            receiver,
            blockNumber,
            balance,
            senderBalanceSign
```

```
        );

        address receiverAddr = extractBalanceProofSignature(
            senderAddr,
            blockNumber,
            balance,
            receiverBalanceSign
        );

        require(receiver == receiverAddr);

        _closeChannel(senderAddr, receiverAddr, blockNumber, balance);

        emit ChannelClosed(receiver, receiver, blockNumber, balance);

    }
```

All'interno di tale funzione è possibile evidenziare due fasi, una prima di controllo ed una seconda di chiusura effettiva del payment channel.

Le condizioni per poter chiudere il canale sono relative al controllo finale sulle firme, è necessario infatti che il processo di recupero dell'address destinatario tramite firma dia come risultato l'address in questione. Ciò avviene richiamando la funzione *extractBalanceProofSignature* la quale riprende lo stesso concetto applicato a web3 ma utilizzandolo su solidity, ossia calcola l'hash di un insieme di dati tramite SHA3 e successivamente ne deriva l'address combinando l'hash appena ottenuto ed il balance firmato dall'inviante.

Una volta terminati tali controlli, si passa alla chiusura effettiva del canale. Essa avviene richiamando la funzione _closeChannel(). In questa funzione avviene il passaggio più importante ossia il trasferimento dei token dal deposito alle parti componenti il sistema.

```
function _closeChannel(address sender, address receiver, uint32
  blockNumber, uint192 balance) private {
        bytes32 key = keccak256(abi.encodePacked(sender, receiver,
  blockNumber));
        Channel memory channel = _channels[key];
        require(channel.blockNumber > 0);
        require(balance <= channel.deposit);

```

```
        delete _channels[key];

        require(_token.transfer(receiver, balance));
        require(_token.transfer(sender, channel.deposit.sub(balance)));
    }
```

Il processo di chiusura avviene ricreando la stessa chiave utilizzata nell'apertura del payment channel, tale chiave verrà poi inserita nel mapping dei canali per ricercare il payment channel che si vuole eliminare, una volta trovato si procede con la cancellazione di esso.

Per concludere, non resta che trasferire i token dal deposito ai nodi.

```
function _transfer(address from, address to, uint256 value) internal {
        require(to != address(0));
        require(value <= _balances[from]);

        _balances[from] = _balances[from].sub(value);
        _balances[to] = _balances[to].add(value);
        emit Transfer(from, to, value);
    }
```

In questo modo i token verranno detratti dal deposito ed inviati al balance del router e del device.

Capitolo 5

Risultati

I risultati analizzati son stati ricavati tramite una simulazione dei processi attraverso il provider ropsten, una test network che simula le tempistiche ed i costi della rete ethereum. Per prima cosa è stato impostato un gasPrice pari a 5 Gwei che è il prezzo standard per le transazioni in una normale blockchain. Nella seguente tabella sono rappresentati i tempi ed i costi relativi al deploy degli smart contracts del sistema tramite migrations. Il processo di deploy degli smart contract del progetto, in questo caso ha una durata

Name	Time(s)	GasUsed	Cost(ETH)	Cost(EUR)
Migrations	9	263741	0.001318705	0.27
WiFiToken	12	1288268	0.00644134	1.33
PaymentChannel	13	1173185	0.005865925	1.22
TOTAL	34	2686690	0.01343345	2.78

Tabella 5.1: Costi e tempi di Deploy degli Smart Contracts.

media ha 34 secondi con 2686690 Gas usati, il costo complessivo sarà dunque 0.01343345 Eth ossia di 2,78 Euro.

Per quanto riguarda le operazioni degli smart contracts effettuabili dai nodi, esse si presentano come mostrato in figura 5.2. I punti che richiedono maggior interesse sono quelli relativi ai costi ed ai tempi d'esecuzione. Quest'ultimi non sono stati inseriti per la volatilità dei dati, infatti va specificato che i tempi di attesa variano a seconda dell'affluenza di richieste in un determinato momento della giornata, più richieste ci sono e più il tempo di attesa aumenta.

Name	GasUsed	Cost(ETH)	Cost(EUR)
transferFrom	52046	0.000260	0.056
mint	66738	0.00033369	0.069
approve	44111	0.00022733	0.047
createChannel	97310	0.000478905	0.099
closeChannel	75862	0.00036931	0.077

Tabella 5.2: Costi e tempi relativi alle funzioni più importanti degli Smart Contracts.

Seguendo i dettagli del sito *EthGasStation* è possibile stimare che in genere le transazioni dal gasPrice di 5 Gwei in media hanno un tempo di inserimento minore di 5 minuti. Tramite dei test effettuati in vari momenti della giornata è possibile definire i tempi d'attesa più significativi (transferFrom e createChannel) da 9 secondi fino ad 1 minuto e 8 secondi, con una media di 12 secondi circa per il trasferimento dei token e di 16 secondi per l'apertura di canale.
Per quanto riguarda i costi, l'obbiettivo dell'implementazione dei payment channel è relativo al fatto che si devono limitare le operazioni ripetitive da inserire nella blockchain (in questo caso del trasferimento di token). Per permetterlo, il payment channel, utilizza gli scambi di messaggi firmati evitando dunque di inserire i trasferimenti. L'operazione relativa ad essi è rappresentata nella tabella come *transferFrom* ed ha un costo di 0.000260 ETH. Se si ripete dunque tale operazione per n volte, il risultato avrà una crescita lineare dell'ordine di n detraendolo dal balance dell'utente che ne implementa l'utilizzo. Il procedimento è analogo se si pensa ai tempi d'attesa relativi all'inserimento della transazione nella blockchain, infatti la media relativa ad essi è di circa 12 secondi che, se ripetuta, può portare a rallentamenti considerevoli. Grazie ai payment channel è possibile dunque evitare tutte le n operazioni e tempi di caricamento, inserendo l'ultimo balance aggiornato nei messaggi nella transazione di chiusura del canale, ossia in *closeChannel*. Nella tabella 5.3 e 5.4 sono riportati rispettivamente i casi in cui si decide di effettuare 100 trasazioni on-chain ed off-chain tramite payment channel.

Name	Call	GasUsed	Cost(ETH)	Cost(EUR)
transferFrom	100	5204600	0.0260	5.6

Tabella 5.3: Costi relativi a 100 transazioni effettutate on-chain.

Name	Call	GasUsed	Cost(ETH)	Cost(EUR)
approve	1	44111	0.00022733	0,047
createChannel	1	97310	0.000478905	0,099
closeChannel	1	75862	0.00036931	0.077
TOTAL	3	214283	0.00107554	0.233

Tabella 5.4: Costi relativi a 100 transazioni effettutate off-chain.

Si noti dunque che le normali transazioni on-chain ripetute 100 volte, portano ad un totale di 5.6 Euro ed un tempo di attesa complessivo di circa 20 minuti, mentre lo stesso numero di transazioni off-chain ha un costo complessivo di 0.223 Euro con tempo di attesa complessivo di 32 secondi.
Un altro punto che richiede attenzione, è quello relativo al tempo di attesa dovuto all'inserimento di una transazione nella blockchain, in questo caso relativo all'apertura del payment channel. Si pensi infatti, ad un utente che si allontana da un router, nel momento in cui l'user perde la connessione, è necessario che ne attivi una seconda con un altro router. In questo caso, l'unico tempo d'attesa incisivo è quello relativo all'apertura del canale (*openChannel* nella tabella), che in media è di 16 secondi.

Capitolo 6

Conclusioni

Per concludere, il progetto soddisfa i requisiti iniziali mantenendo stabili le transazioni off-chain garantendone sicurezza e validità. Il sistema inoltre permette di ridurre i costi relativi alle commissioni per le transazioni ed il tempo di attesa relativo ad esse. Attraverso i risultati si è notato che l'unico problema relativo ai payment channels riguarda l'apertura di essi, infatti vi è in media un ritardo per transazione ci circa 16 secondi. Nel caso in cui una persona si allontani dal router connesso infatti, può esserci la possibilità in cui l'utente richieda una nuova connessione con un secondo router. Il problema si presenta durante un cambio di connessione, provocando, perdita di alcuni secondi e perdita di pacchetti dati, rendendo la connettività poco uniforme. Una possibile soluzione può essere quella di implementare un payment channel network tra i router della città. In questo modo un utente può risparmiare in termini di costi e di attesa, utilizzando i payment channels dei nodi per indirizzare una specifica transazione. Gli unici costi in questo caso sarebbero quelli di apertura del canale e delle commissioni dei nodi intermediari sul quale passa la transazione. Andando invece a ridurre i tempi di attesa che provocherebbero una perdita di pacchetti ed un'interruzione della connessione. Una seconda soluzione potrebbe essere quella di implementare una *Mesh Network* in una città, ossia una sorta di *Municipal wireless network* privata. In questo modo il sistema si presenta con una singola fonte di connessione alla quale sono legati un insieme di ripetitori che ne amplificano la copertura. Attraverso un sistema del genere, è possibile limitare l'utilizzo di operazioni di apertura del payment channel. Questa infatti, avviene solo durante la prima connessione, successivamente il sistema garantisce copertura con-

tinua, andando ad eliminare il problema di creare nuovi payment channel e garantendo pagamenti off-chain tramite scambio di token.

Bibliografia

[1] *Cohen-Almagor, Raphael. (2011). Internet History. International Journal of Technoethics.* Vol. 2. 45-64. 10.4018/jte.2011040104.

[2] Tripathi, Nikhil & Mehtre, Babu. (2013) *DoS and DDoS Attacks: Impact, Analysis and Countermeasures.* 1-6.

[3] Government Office for Science, The Rt Hon Matt Hancock MP, Ed Vaizey *Distributed Ledger Technology: beyond block chain.* 2016.

[4] Mark Gates. *Ethereum: Complete Guide to Understanding Ethereum, Blockchain, Smart Contracts, ICOs, and Decentralized Apps. Includes guides on buying Ether, Cryptocurrencies and Investing in ICOs.*

[5] Haber, S., Stornetta,W.S. *How to time-stamp a digital document. J.* Cryptology 3, 99–111 (1991).

[6] Nakamoto, Satoshi. (2009). *Bitcoin: A Peer-to-Peer Electronic Cash System.*

[7] Zheng, Zibin & Xie, Shaoan & Dai, Hong-Ning & Chen, Xiangping & Wang, Huaimin. (2017). *An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends.* 10.1109/BigDataCongress.2017.85.

[8] Vujicic, Dejan & Jagodic, Dijana & Ranđić, Siniša. (2018). *Blockchain technology, bitcoin, and Ethereum: A brief overview.* 1-6. 10.1109/INFOTEH.2018.8345547.

[9] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. *On the Security and Performance of Proof of Work Blockchains.* In Proceedings of the 2016 ACM SIGSAC Conference on Computer and

Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 3–16. DOI:https://doi.org/10.1145/2976749.2978341

[10] Shahar Somin, Goren Gordon and Yaniv Altshuler *ERC20 Transactions over Ethereum Blockchain: a Network Perspective* MIT Media Lab, Cambridge, MA, USA

[11] Vitalik Buterin. *Ethereum: A Next-Generation Cryptocurrency and Decentralized Application Platform* . 2014.

[12] Poon, Joseph, and Thaddeus Dryja. *"The bitcoin lightning network: Scalable off-chain instant payments."* (2016).

[13] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger."

[14] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. *2018. General State Channel Networks. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18).* Association for Computing Machinery, New York, NY, USA, 949–966. DOI:https://doi.org/10.1145/3243734.3243856

[15] V. Buterin,"Sharding faq."[Online].Available:https://github.com/ethereum/wiki/wiki/Sharding-FAQ

[16] Raiden Network, https://raiden.network/101.html

[17] Ethereum Price, https://ethereumprice.org/

[18] Documentazione Solidity, https://solidity.readthedocs.io/en/v0.6.3/

[19] Web3.js, https://web3js.readthedocs.io/en/v1.2.6/

[20] Ganache, https://www.trufflesuite.com/ganache

[21] Truffle, https://www.trufflesuite.com/docs/truffle/overview

[22] Node.Js, https://nodejs.org/it/

Ringraziamenti

Ringrazio il Professor Stefano Ferretti ed il Dottor Mirko Zichichi per tutti i consigli, la disponibiltà e l'aiuto datomi.

Ringrazio mia madre, mio padre e mia sorella Martina per esserci sempre stati, per il sostegno costante e per tutte quelle volte che mi hanno saputo ascoltare ed aiutare. Vi voglio bene.

Ringrazio Rosa per tutti i momenti passati insieme, per tutte le parole di incoraggiamento, per tutto l'amore che riesce a trasmettermi, per avermi fatto crescere ed avere più fiducia in me. Questo risultato è soprattutto merito suo.

Ringrazio tutti i miei amici, per avermi accompagnato in questo percorso, chi da vicino e chi da lontano. Ogni piccolo gesto è stato fondamentale per me.

Infine ringrazio i miei compagni di corso Andrea, Francesco e Michele per avermi fatto passare questi anni nel migliore dei modi, un ringraziamento in particolare a Federico, una costante nel mio percorso universitario.