Alma Mater Studiorum · Università di Bologna
Campus di Cesena

Scuola di Scienze
Corso di laurea in ingegneria e scienze informatiche

BLOCKCHAIN E INTERNET OF THINGS: REALIZZAZIONE DI UN'APPLICAZIONE DECENTRALIZZATA PER L'AFFITTO DI CASE VACANZA

Tesi di Laurea in
Sistemi Embedded e Internet of Things

Relatore:
Prof. Ing.
ANDREA OMICINI

Correlatore:
Dott.
GIOVANNI CIATTO

Presentata da:
MATTEO SACCOMANNI

Terza Sessione di Laurea
Anno Accademico 2017 - 2018

PAROLE CHIAVE

Blockchain

Smart Contract

Peer-to-peer networks

Internet of Things

Ai miei genitori
che hanno sempre creduto in me
e mi hanno permesso di arrivare dove sono oggi.

Grazie

Abstract

Negli ultimi anni, con l'evoluzione del World Wide Web, Internet ha subito una crescita esponenziale e ogni giorno la rete diventa sempre più pervasiva nelle nostre vite grazie agli smartphone e agli innumerevoli dispositivi smart che sono entrati nelle nostre case dando vita all'Internet of Things. Per realizzare servizi sempre più veloci, evoluti e interattivi Internet nel corso del tempo, e in particolar modo nell'ultimo periodo con l'avvento del cloud, è diventato contemporaneamente sempre più centralizzato, divenendo così più fragile e concentrando nelle mani di pochissime aziende il funzionamento delle applicazioni e dei servizi che utilizziamo ogni giorno, oltre ai nostri dati. Questa tesi si pone come obiettivo l'analisi, la progettazione e la realizzazione di un'applicazione web totalmente decentralizzata basata su blockchain adibita all'affitto di casa vacanza a breve termine, in maniera similare a quanto già realizzato da piattaforme online quali Booking e Airbnb, divenute molto popolari e utilizzate negli ultimi anni. La tesi si concentra inoltre sull'ambito Internet of Things, analizzando come queste due tecnologie possono interagire tra loro per offrire servizi tangibili nel mondo reale. A tale scopo si vuole simulare, attraverso l'utilizzo di un RaspBerry Pi, la realizzazione di una serratura smart key-less, che gestisca gli accessi interrogando il sistema precedente per verificare se l'utilizzatore dispone dei permessi necessari ad accedere. La tesi mira infine a individuare, oltre ai vantaggi, anche quali sono i limiti attuali delle tecnologie che sono state utilizzate per la realizzazione di questo progetto.

Indice

Introduzione **1**

1 Background **5**
1.1 Blockchain 5
1.1.1 Blockchain pubbliche 7
1.1.2 Blockchain permissioned 8
1.1.3 Generazioni di blockchain 8
1.2 Ethereum 9
1.3 Ethereum e Web 3.0 11
1.4 Whisper 13
1.5 IPFS 17

2 Requisiti e Analisi **23**
2.1 Descrizione del caso di studio 23
2.2 Analisi dei requisiti 24
2.2.1 Requisiti funzionali 25
2.2.2 Requisiti non funzionali 26

3 Progettazione architetturale **29**
3.1 Scelta delle tecnologie 29
3.2 Architettura del sistema 32

4 Implementazione **37**
4.1 Configurazione della testnet 37
4.2 Creazione di un annuncio 39
4.3 Creazione di una prenotazione 42
4.4 Interazione con la serratura smart 45

5 Valutazione **49**
5.1 Valutazione dell'applicazione 49
5.2 Validazione 51

Conclusioni e lavori futuri **52**

Bibliografia **57**

Introduzione

Dalla nascita del World Wide Web Internet ha conosciuto una crescita esponenziale e negli ultimi anni si può dire di aver assistito, non solo a una rapidissima rivoluzione tecnologica, ma anche a un grandissimo mutamento della nostra società. Internet ha cambiato le vite di tutti, dal modo di comunicare, ricercare, intrattenersi e acquistare, ogni giorno siamo sempre più connessi alla rete. Complice di questa pervasività di Internet nelle nostre vite è stato l'avvento dello smartphone, e da allora di tutta una serie di dispositivi dotati di unità di elaborazione sempre più piccole ed economiche che sono entrati nelle nostre case, creando una rete di oggetti sempre più intelligenti e sempre più connessi, dando vita a quello che oggi è conosciuto come l'Internet of Things (IOT).

I servizi che Internet è in grado in offrire sono divenuti nel corso del tempo sempre più numerosi, evoluti e interattivi, e di pari passo sono aumentate anche le aspettative dei suoi utilizzatori. Mentre Internet, nella sua concezione originaria, era un'infrastruttura fortemente decentralizzata, progettata esplicitamente per trasferire dati anche in caso di caduta di uno o più dei suoi nodi, oggigiorno è divenuta una rete molto più centralizzata e con diversi punti critici. Il web non funziona senza i server DNS e il modello adottato dal web 2.0 rende le applicazioni dipendenti da server remoti che eseguono le funzioni e memorizzano i dati. Nel corso del tempo, e in particolar modo negli ultimi anni con l'avvento del cloud, ci si è orientati sempre più verso un modello centralizzato che, se da parte ha consentito la creazione e una così rapida evoluzione delle applicazioni e dei servizi che utilizziamo ogni giorno, dall'altra ha reso il web un po' più fragile concentrando nelle mani di pochissime aziende il loro funzionamento e i nostri dati.

Nel 2008 viene però presentato da Satoshi Nakamoto Bitcoin, un sistema di pagamento mondiale completamente decentralizzato che elimina per la prima volta la necessità di dover utilizzare un intermediario per effettuare lo scambio di asset digitali. Alla base del funzionamento di Bitcoin vi è una tecnologia chiamata blockchain le cui applicazioni non si limitano soltanto alla possibilità di creare valute digitali. Negli ultimi anni sono nati sulla scia

di Bitcoin centinaia di progetti che si prefiggono come scopo la risoluzione dei problemi più svariati. Tra questi ve ne sono anche alcuni, tra cui ad esempio Ethereum, che mirano a utilizzare la blockchain come tecnologia per sviluppare applicazioni e una nuova versione del web decentralizzati.

Questa tesi si pone come obiettivo la realizzazione di un'applicazione web totalmente decentralizzata basata su blockchain adibita all'affitto di casa vacanza a breve termine, in maniera similare a quanto già realizzato da piattaforme online quali Booking e Airbnb, divenute molto popolari e utilizzate negli ultimi anni. La tesi si concentra inoltre sull'ambito Internet of Things, analizzando come queste due tecnologie possono interagire tra loro per offrire servizi tangibili nel mondo reale. A tale scopo si vuole realizzare anche un'applicazione per un sistema embedded che simulerà il funzionamento di una serratura smart key-less e dovrà interagire con l'applicazione precedente, risiedente sulla blockchain, per verificare se concedere il diritto d'accesso. Lo scopo di questa tesi non si limita alla realizzazione del sistema sopra descritto, mira infatti anche a determinare quali sono gli attuali limiti delle tecnologie che sarà necessario individuare, approfondire e utilizzare per lo svolgimento di questo progetto.

La tesi, che ripercorre il lavoro svolto e riassume le nozioni apprese, si articola in cinque capitoli:

Capitolo 1 Nel primo capitolo viene descritto lo stato dell'arte delle tecnologie che sono state successivamente utilizzate nella fase di sviluppo per la realizzazione del sistema oggetto di questa tesi.

Capitolo 2 Nel secondo capitolo viene descritto nel dettaglio il caso di studio e viene affrontata l'analisi dei requisiti.

Capitolo 3 Nel terzo capitolo vengono illustrate le principali scelte architetturali e tecnologiche che sono state prese per soddisfare i requisiti definiti nella precedente fase di analisi.

Capitolo 4 Nel quarto capitolo viene mostrato come sono state sviluppate le parti più significative del sistema cercando di dare spazio a tutte le tecnologie utilizzate e concentrandosi in particolare al modo in cui queste interagiscono tra loro.

Capitolo 5 Nel quinto capitolo si fornisce una valutazione del lavoro svolto e si analizzano i limiti della tecnologia blockchain emersi dalla realizzazione del progetto. Viene inoltre discusso anche come è stata svolta la validazione dell'applicazione.

Nelle conclusioni viene infine analizzato il lavoro svolto, verificando se sono stati raggiunti tutti gli obiettivi che questa tesi si prefissa, e si evidenziano i miglioramenti e le funzionalità che potrebbero essere introdotti in lavori futuri.

Capitolo 1

Background

Per la realizzazione dell'applicazione decentralizzata oggetto di questa tesi si è reso necessario lo studio preliminare di una serie di tecnologie e piattaforme, che hanno fortemente influenzato le scelte fatte nelle successive fasi di analisi e progettazione.

Data la connotazione fortemente innovativa della maggior parte di queste tecnologie, in questo primo capitolo verrà fornita una panoramica non esaustiva su di esse, al fine di fornire tutti gli strumenti necessari a comprendere il progetto di questa tesi. Le varie sezioni forniranno dapprima un'introduzione generale sull'attuale stato dell'arte e successivamente dettaglieranno maggiormente le piattaforme che si è scelto di adottare nello specifico per la realizzazione dell'applicazione.

1.1 Blockchain

Una blockchain è una base di dati distribuita, condivisa e immutabile.

Originariamente formulata nel 1991 da un gruppo di ricercatori [1] come soluzione per marcare i documenti digitali in modo che non fosse possibile retrodatarli o manometterli, questa tecnologia è rimasta tuttavia inutilizzata fino al 2009 quando Satoshi Nakamoto l'ha impiegata per creare Bitcoin [2], la prima criptovaluta digitale.

Come suggerisce il termine "blockchain" stesso, si tratta di un registro digitale aperto e distribuito dove le informazioni sono raggruppate in blocchi tra loro concatenati mediante meccanismi crittografici, che formano così una lista crescente di record in grado di resistere a manomissioni.

Tra i vari meccanismi crittografici utilizzati per garantire la sicurezza di una blockchain, vi sono le funzioni hash crittografiche, ovvero degli algoritmi matematici in grado di mappare dei dati di dimensione arbitraria in una

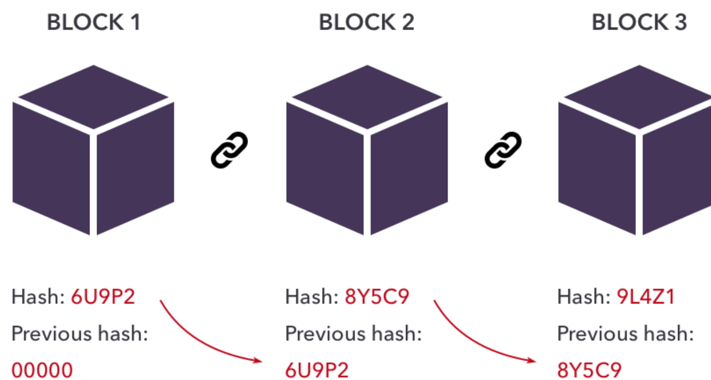

Figura 1.1: Rappresentazione semplificata di una blockchain [3]

stringa di bit di dimensione prefissata (un hash), e progettati per essere funzioni unidirezionali, cioè impossibili da invertire.

Una volta che i dati sono stati registrati all'interno della blockchain diventa estremamente difficile modificarli retroattivamente: ogni blocco infatti oltre a memorizzare dei dati (tipicamente transazioni) mantiene anche un timestamp e l'hash del blocco precedente. L'hash, calcolato alla creazione di ogni blocco, rende estremamente semplice rilevare se c'è stato un tentativo di alterazione del passato, in quanto una qualunque modifica al contenuto di un blocco provocherebbe anche una modifica del suo hash value e quindi un'invalidamento dell'intera blockchain dal momento che il blocco successivo non conterrebbe più un riferimento valido al blocco precedente. Affinché un attaccante sia dunque in grado di manomettere i dati memorizzati nella blockchain è necessario che questo sia in grado di ricalcolare in cascata anche gli hash di tutti i blocchi successivi a partire dal blocco che desidera modificare. Ciò rende l'operazione di manomissione tanto più difficile quanto è maggiore il numero di blocchi che l'attaccante dovrà necessariamente tentare di modificare.

Un altro modo con cui la blockchain garantisce la sua sicurezza è attraverso la sua ridondanza e distribuzione: anziché utilizzare un'entità centrale per la gestione della catena, la blockchain utilizza un sistema distribuito e decentralizzato, costituito da una rete di nodi peer-to-peer ognuno dei quali possiede una copia privata dell'intera blockchain.

Per garantire la coerenza tra le varie copie, l'aggiunta di un blocco è globalmente regolata da un protocollo condiviso che realizza il consenso tra i nodi. Tipicamente nelle blockchain pubbliche la collaborazione di massa è alimentata da interessi collettivi che consentono di superare il problema dell'infinita riproducibilità di un bene digitale e della doppia spesa, senza

che sia necessario l'utilizzo di un server centrale o di un'autorità.

Blockchain è una tecnologia in continua evoluzione: dall'avvento di Bitcoin la ricerca è esplosa e ha fatto venire alla luce diverse varianti ed evoluzioni, tanto che oggi è già possibile farne una prima classificazione suddividendo le blockchain in due famiglie principali: blockchain pubbliche e blockchain permissioned (o private).

1.1.1 Blockchain pubbliche

Le blockchain pubbliche sono tipicamente open-source e non prevedono alcun tipo di restrizione, chiunque vi può prendere parte senza che sia necessario alcun permesso. In particolare:

- chiunque può scaricare l'intera blockchain e il programma per diventare un nodo della rete iniziando così a partecipare alla validazione delle transazioni da aggiungere alla blockchain secondo il protocollo di consenso adottato;
- chiunque può inviare transazioni alla rete che verranno inserite nella blockchain se valide;
- chiunque può visualizzare tutte le transazioni memorizzate all'interno della blockchain attraverso un block explorer. A seconda del tipo di blockchain le transazioni possono essere più o meno trasparenti garantendo un certo livello di privacy e anonimato.

Tipicamente questo tipo di blockchain adotta delle forme di incentivi economici per coloro che si occupano della validazione delle transazioni (i cosiddetti "miner") adottando sistemi come Proof of Work o Proof of Stake [4]. Questi schemi consentono di mantenere la blockchain al sicuro e di funzionare in un contesto totalmente *trustless*, ovvero in un ambiente dove i nodi della rete non si conoscono e non si fidano l'un l'altro.

La principale caratteristica delle blockchain pubbliche è dunque quella di offrire un mezzo tecnologico per poter realizzare la de-intermediazione nella fornitura di molteplici servizi e consentono la creazione e l'esecuzione di applicazioni decentralizzate senza che sia necessario sostenere alcun costo per l'infrastruttura. A questi vantaggi si contrappongono alcuni svantaggi quali la possibilità di gestire un numero di transazioni al secondo significativamente più basso rispetto a quelle che sono in grado di gestire le blockchain permissioned e l'impossibilità di memorizzare all'interno della blockchain informazioni riservate dal momento l'intera catena di blocchi è pubblicamente consultabile [5].

Le blockchain pubbliche più note sono Bitcoin [2] e Ethereum [6].

1.1.2 Blockchain permissioned

Le blockchain permissioned si contrappongono in maniera netta a quelle pubbliche in quanto prevedono restrizioni sull'accesso contemplando la presenza di un sottosistema per l'identificazione e la gestione dei permessi associati ai vari nodi. Soltanto chi è stato autorizzato dall'amministratore a prendere parte alla rete può quindi, in base ai diritti che gli sono stati concessi, esplorare il contenuto della blockchain, inviare o verificare nuove transazioni.

Questa tipologia di blockchain trova impiego principalmente nei settore bancario e industriale, e più in generale in tutti quei contesti dove vi è un consorzio di attori che lavorano insieme e che desiderano scambiarsi informazioni in modo riservato ma che non si fidano completamente l'uno dell'altro. Una delle applicazioni più frequenti che vengono citate quando si parla di blockchain permissioned, è infatti quella della gestione della filiera logistico produttiva, un contesto nel quale vi sono diverse parti (fornitori, trasportatori, intermediari finanziari, catene di distribuzione) che hanno necessità di condividere fra loro informazioni senza che però siano pubbliche e visibili a tutto il mondo (si consideri il caso di segreti o strategie aziendali che rappresentano un vantaggio rispetto alla concorrenza) o anche soltanto ad altre parti coinvolte nella filiera stessa (si pensi ad esempio ad un prezzo speciale concordato con una certa catena di distribuzione che non vuole essere reso noto alle altre) [7].

Le blockchain permissioned risolvono efficacemente questo problema, permettendo la costituzione una base di dati distribuita e condivisa tra tutte le parti dove tutte le modifiche rimangono registrate e immutabili e dove tutti partecipano alla validazione delle transazioni evitando così di concentrare tutto il potere presso un unico ente in cui tutte le parti dovrebbero riporre la fiducia, esattamente come accadrebbe adottando un database condiviso tradizionale.

La ricerca nel campo delle blockchain permissioned è iniziata solo negli ultimi anni e risulta essere ancora agli albori. Fra le blockchain più promettenti di questo tipo oggi la più nota è probabilmente Hyperledger Fabric [8].

1.1.3 Generazioni di blockchain

Un'altra modalità adottata per la classificazione delle blockchain è la suddivisione in generazioni in base alle caratteristiche e funzionalità offerte dalla blockchain stessa. Ad oggi si possono contare tre generazioni:

1. **Prima generazione: criptovalute**

   La prima applicazione della blockchain è stata la realizzazione di criptovalute come Bitcoin e altre semplici alt-coin come Litecoin.

   Con la loro introduzione è stato reso possibile per la prima volta lo scambio diretto di denaro tra due parti senza la necessità di nessun intermediario (peer-to-peer) in modo sicuro, veloce ed economico.

2. **Seconda generazione: digital assets, smart contract e dApp**

   L'avvento della seconda generazione di blockchain si è avuto con la nascita di Ethereum, la prima blockchain ad introdurre il concetto di smart contract, ovvero semplici programmi che possono essere scritti, distribuiti ed eseguiti all'interno di un sistema informatico decentralizzato, sicuro, immutabile e affidabile.

   Le blockchain di seconda generazione non si limitano più quindi a permettere lo scambio di denaro ma permettono anche la definizione e lo scambio di un qualsiasi asset digitale. Gli sviluppatori possono realizzare su di esse nuovi token e applicazioni decentralizzate (dApps).

3. **Terza generazione: scalabilità, interoperabilità e IOT**

   La definizione esatta di blockchain di terza generazione è un tema ancora dibattuto, ma sono già diversi i progetti che si sono dati questa etichetta, il più noto fra questi è Cardano [9].

   I problemi che le blockchain di terza generazione stanno cercando di risolvere sono legati alla scalabilità, in particolare attraverso la creazione di molteplici layer (tendenza che ha portato alla anche nascita di Lightning Network, layer di secondo di livello per transazioni istantanee su Bitcoin), all'interoperabilità tra blockchain diverse e allo sviluppo di tecnologie ad hoc per la realizzazione di applicazioni blockchain M2M (machine to machine) in ottica Internet of Things.

1.2 Ethereum

Ethereum è una piattaforma decentralizzata, pubblica e open-source basata su blockchain e dotata di funzionalità di creazione e pubblicazione di contratti intelligenti scritti in un linguaggio di programmazione Turing completo. Proposto inizialmente da Vitalik Buterin nel 2013, il suo sviluppo è iniziato l'anno seguente e oggi è principalmente curato dall'Ethereum Foundation.

Ethereum dispone di una criptovaluta nativa, l'Ether, che oltre a poter essere scambiato fra account, è generato dalla piattaforma stessa come ricompensa ai miner per il lavoro computazionale svolto. A ciascun account (o wallet) sono associati una chiave privata e una chiave pubblica che prende anche il nome di indirizzo. Per poter effettuare una transazione è necessario che il mittente conosca l'indirizzo, ovvero la chiave pubblica, del wallet del destinatario e che firmi digitalmente la transazione prima di inviarla alla rete Ethereum con la propria chiave privata, dimostrando così che il richiedente dell'operazione è l'effettivo titolare del wallet.

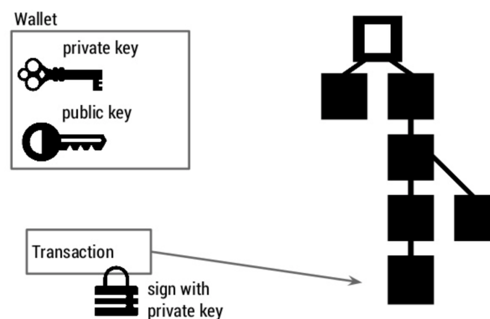

Figura 1.2: Private key e public key

Ethereum mette a disposizione anche l'Ethereum Virtual Machine (EVM), una macchina virtuale decentralizzata all'interno della quale vengono eseguiti gli smart contract utilizzando la potenza computazionale dei nodi che costituiscono la rete. Gli smart contract possono essere scritti utilizzando diversi linguaggi di programmazione, il più utilizzato dei quali è al momento Solidity [1]. Il codice sorgente scritto in Solidity, o in un qualsiasi altro linguaggio ad alto livello fra quelli supportati, deve essere poi compilato per produrre un bytecode pronto per essere eseguito dalla EVM.

Ciascuna transazione generata sulla rete Ethereum, sia essa un semplice trasferimento di Ether, o l'invocazione di una funzione di uno smart contract, comporta un costo che è proporzionale alla complessità computazionale, alla banda utilizzata e/o alla quantità di storage necessario. Questo meccanismo interno adottato da Ethereum prende il nome di "gas", ed è stato ideato per evitare possibili attacchi spam o DDoS (Distributed Denial of Service) che potrebbero compromettere il funzionamento dell'intera rete.

Per ogni transazione l'utente può dunque definire un *gas price* e un *gas limit*, ovvero rispettivamente l'ammontare che decide di pagare per ogni unità di gas consumata (valore espresso solitamente in Gwei, un sottomultiplo del-

[1] https://solidity.readthedocs.io/en/latest/index.html

l'Ether) e la quantità massima di gas consumabile. In questo modo il mittente della transazione conosce a priori quale sarà il costo massimo dell'operazione, e garantisce al tempo stesso che ogni computazione giungerà sempre a termine. Qualora infatti si verificasse ad esempio un loop infinito all'interno del codice di uno smart contract, la computazione verrà interrotta non appena si sarà consumato tutto il gas messo a disposizione al momento della creazione della transazione. Il costo effettivo sarà determinato dall'effettiva quantità di gas utilizzato moltiplicato per il gas price indicato e quindi detratto al mittente dal saldo del suo wallet in Ether. La possibilità di specificare il gas price fa si che sia possibile assegnare una priorità alle transazioni, i miner preferiranno infatti inserire all'interno del prossimo blocco le transazioni con un gas price più elevato, per ottenere una maggiore ricompensa.

Mediamente la rete Ethereum genera un nuovo blocco ogni 14/15 secondi, un tempo molto più breve rispetto a quello impiegato da Bitcoin che si aggira intorno ai 10 minuti.

Per la messa in sicurezza della rete e per il conseguimento del consenso distribuito, Ethereum, come Bitcoin, basa il suo funzionamento sull'utilizzo di un protocollo Proof-of-work (PoW) ma è già in programma nei prossimi mesi un hard-fork (major update) per passare all'utilizzo di un protocollo Proof-of-stake (PoS) che risolverà il problema dell'enorme quantità di energia elettrica attualmente consumata per l'attività di mining [10].

1.3 Ethereum e Web 3.0

Il progetto Ethereum non concerne soltanto lo sviluppo della nota blockchain, il fondatore Vitalik Buterin ha infatti obiettivi molto più ambiziosi. La roadmap di Ethereum prevede infatti lo sviluppo di altre tecnologie complementari che si andranno ad affiancare alla blockchain per realizzare quello che secondo l'Ethereum Foundation diverrà il Web 3.0, ovvero la prossima generazione del Web.

Se con Web 1.0 ci si riferisce al primo stadio del World Wide Web, costituito da pagine web statiche, connesse tra loro da semplici hyperlink, e da prime semplici applicazioni come blog e forum, la transizione al Web 2.0 si è avuta con l'avvento di AJAX e altre tecnologie che hanno reso il web sempre più interattivo e dinamico, consentendo agli sviluppatori di realizzare web application sempre più complesse, dove gli utenti possono interagire e modificare i contenuti delle pagine web.

Secondo la visione del progetto Ethereum la caratteristica principale della prossima generazione del World Wide Web sarà la decentralizzazione, un web senza più alcun server centralizzato e fatto di dApps.

Qualsiasi applicazione non banale per offrire i suoi servizi richiede però tipicamente diverse risorse per poter essere eseguita: potenza computazionale, un database, memoria di massa e un sistema di comunicazione per poter interagire con altre applicazioni.

Come discusso nelle precedenti sezioni, blockchain come quella di Ethereum sono in grado di offrire solo i primi due tipi di queste risorse: la EVM fornisce potenza computazionale per eseguire il codice contenuto negli smart contract che realizzano la business logic delle applicazioni decentralizzate, mentre la blockchain stessa può memorizzare dati e transazioni esattamente come un database.

Tuttavia ciò non è sufficiente, infatti per poter sviluppare applicazioni complesse e totalmente decentralizzate è necessario disporre anche di una memoria di massa dove poter memorizzare file dati, anche di grandi dimensioni, e di un sistema di comunicazione.

A tal scopo il progetto mira a fornire:

- **Smart contract**: logica decentralizzata
- **Swarm:** storage decentralizzato
- **Whisper:** messaggistica decentralizzata

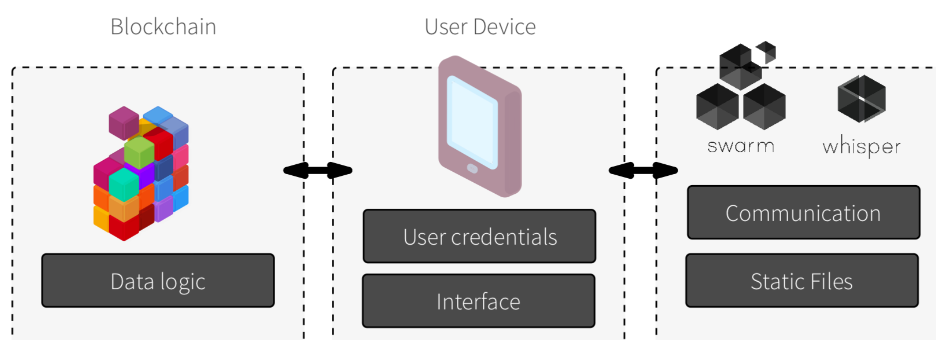

Figura 1.3: Ecosistema Ethereum [11]

L'insieme di queste tecnologie può consentire lo sviluppo di applicazioni totalmente decentralizzate e arbitrariamente complesse, andando a completare la visione di Ethereum come metafora di un computer condiviso, globale e decentralizzato.

Attualmente Swarm e Whisper sono ancora in fase di sviluppo e proseguono la loro evoluzione molto lentamente dal momento che sono posti in secondo piano rispetto allo sviluppo della blockchain di Ethereum. In particolare Swarm risulta essere ancora estremamente acerbo e nella pratica non ancora utilizzabile per sviluppo di applicazioni che non siano dei semplici test, mentre Whisper, pur essendo ancora in alpha stage, risulta già essere utilizzabile dagli sviluppatori di dApps.

Nelle prossime sezioni verrà approfondito più nel dettaglio il funzionamento di Ethereum Whisper, protocollo per la comunicazione tra applicazioni distribuite, e di IPFS, un protocollo esterno al progetto Ethereum e utilizzabile in sostituzione a Swarm per l'archiviazione decentralizzata di file e la creazione di un Web distribuito.

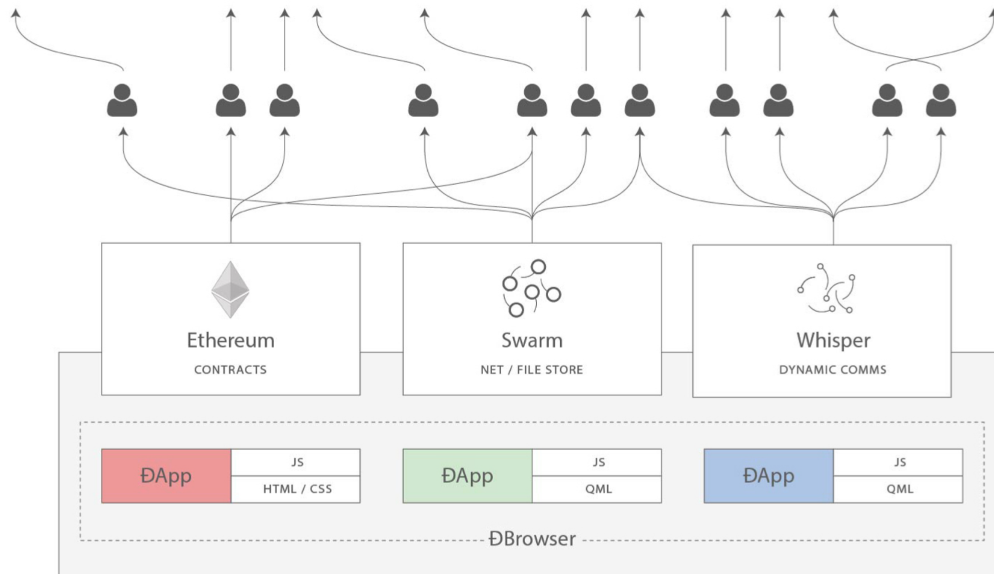

Figura 1.4: Contracts, Swarm e Whisper, le tecnologie messe a disposizione da Ethereum per la realizzazione di un Web 3.0 [12]

1.4 Whisper

Whisper è un protocollo di messaggistica peer-to-peer per applicazioni decentralizzate che fornisce agli sviluppatori di dApps delle semplici API per poter inviare e ricevere messaggi. Le comunicazioni avvengono off-chain, quindi Whisper è un protocollo totalmente indipendente dalla blockchain di Ethereum.

La caratteristica principale di Whisper è la cosiddetta *darkness*, ovvero la possibilità di scambiare messaggi quasi in totale segretezza. Whisper non si limita a garantire la confidenzialità sul contenuto dei messaggi mediante crittografia, ma nasconde anche l'identità del mittente e del destinatario rendendo impossibile agli altri nodi della rete sapere se due entità stanno intrattenendo tra loro una conversazione. Questa scelta ha costretto i progettisti di Whisper a fare alcuni sacrifici in termini di performance per tutelare maggiormente la privacy. Per questo motivo non è un protocollo adatto ad ogni caso d'uso, in particolare non è indicato per comunicazioni in tempo reale e per l'invio di blocchi di dati di grandi dimensioni. Al contrario questo protocollo è stato progettato per inviare piccole quantità di informazioni non persistenti tra dApps o sviluppare ad esempio app di messaggistica focalizzate sulla privacy. Whisper è in ogni caso un protocollo altamente configurabile e consente agli sviluppatori una grande flessibilità nel controllo dei parametri di sicurezza e privacy dei loro messaggi.

Alla base del funzionamento di Whisper vi è una rete decentralizzata di computer chiamati nodi. Per costituire questa rete, un nodo trova i suoi pari utilizzando un algoritmo di node-discovery simile a quello utilizzato da BitTorrent per la ricerca dei peer.

Whisper è incluso di default all'interno dei principali client Ethereum, Geth e Parity, ma è attualmente configurato come feature opzionale da attivare manualmente se lo si desidera. L'utilizzo sulla rete principale è quindi limitato dal numero di nodi Ethereum in esecuzione che hanno il protocollo Whisper abilitato.

Ogni nodo connesso alla rete Whisper dispone di un'identità. Un'identità in Whisper è un'entità (un individuo o un gruppo) titolare di una chiave crittografica che consuma messaggi. Senza una chiave crittografica è impossibile inviare e ricevere messaggi. A seconda del caso d'uso Whisper supporta sia la crittografia simmetrica (AES-256) che quella asimmettrica (SECP-256k1). La crittografia garantisce che solo gli effettivi destinatari possano accedere al contenuto di un messaggio. Se un nodo può decodificare un messaggio, allora questo è destinato a un utilizzatore di quel nodo.

Come anticipato Whisper rende possibile la comunicazione tra due peer in totale riservatezza, senza lasciare alcuna prova né ad eventuali analizzatori del traffico né agli altri peer, anche se questi hanno partecipato all'instradamento dei messaggi. Per realizzare ciò, il meccanismo di scambio dei messaggi risulta essere piuttosto complesso.

Innanzitutto per poter inviare un messaggio questo va prima crittografato utilizzando una chiave simmetrica condivisa o la chiave pubblica del destinatario. E' impossibile inviare messaggi non crittografati tramite Whisper. Se lo desidera, il mittente può allegare al messaggio anche una firma digitale che

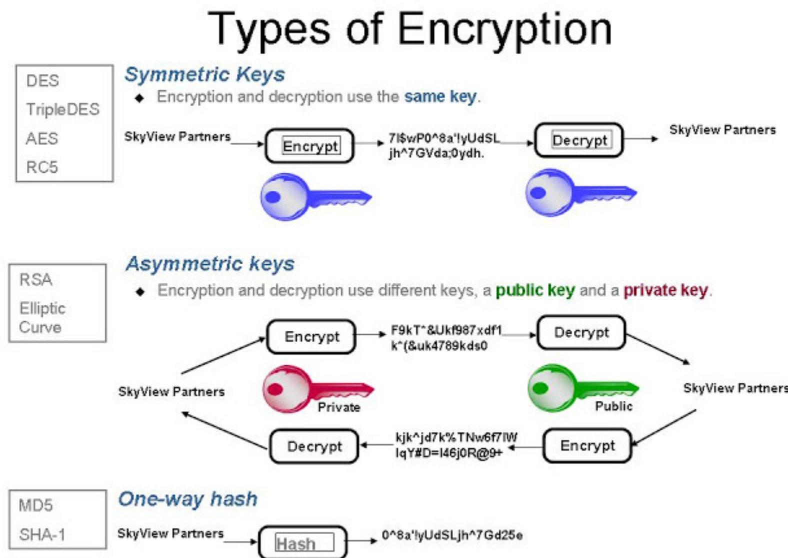

Figura 1.5: Funzionamento di tutti i tipi di crittografia

garantirà al destinatario l'identità del mittente. La firma, qualora fornita, è la signature ECDSA dell'hash Keccak-256 dei dati non crittografati ottenuta utilizzando la chiave privata del mittente. Una volta crittografato il messaggio questo viene inserito all'interno di un pacchetto che prende il nome di *envelope*. Questo aggiunge una serie di metadati necessari per l'instradamento del messaggio. Contrariamente però a quanto avviene nella maggior parte dei protocolli, gli envelope di Whisper non contengono alcuna informazione sul destinatario. Il formato di un envelope è quindi tipicamente il seguente:

```
[ Version, Expiry, TTL, Topic, AESNonce, Data, EnvNonce ]
```

Il contenuto di questi campi sarà reso chiaro nei prossimi paragrafi, ora si noti soltanto che l'unica informazione interessante per un attaccante sarebbe il contenuto del campo *Data* contenente il messaggio, il quale è stato però precedentemente crittografato.

Whisper per sconfiggere l'analisi del traffico e per recapitare il messaggio a destinazione senza conoscere l'identità del destinatario fa sì che ogni messaggio sia instradato su ogni nodo della rete. In questo senso, Whisper si comporta come il protocollo UDP (User Datagram Protocol) quando opera in modalità broadcast. Ogni nodo inoltra dunque il messaggio ai nodi vicini fino a quando il destinatario non lo riceve. Il destinatario sarà l'unico in grado di decifrare il messaggio poiché l'unico a disporre della chiave critto-

grafica necessaria. In ogni caso anche il nodo destinatario effettuerà l'inoltro del messaggio ai suoi nodi vicini per essere completamente irrintracciabile.

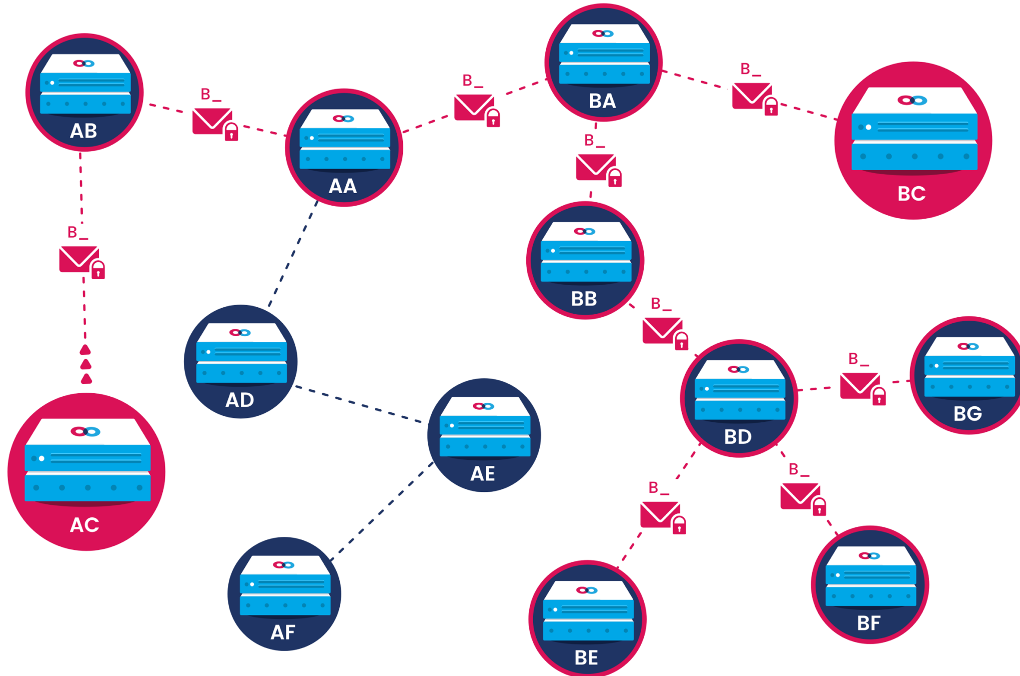

Figura 1.6: Routing dei messaggi con Whisper

Poiché in questo modo ciascun nodo dovrebbe tentare di decifrare ogni messaggio che transita sulla rete per scoprire se era a lui il destinatario della comunicazione, e dal momento che la decrittazione è un lavoro computazionalmente molto costoso, i progettisti di Whisper hanno risolto questo problema richiedendo che a ciascun messaggio sia associato un argomento da memorizzare all'interno del campo *Topic*. Più precisamente gli argomenti dei messaggi vengono automaticamente sottoposti a hash SHA3-256 e solo i primi 4 byte del risultato vengono memorizzati all'interno dell'envelope. Ciascuna identità registra gli argomenti a cui è interessata e utilizzando la sua chiave crittografica crea un filtro per i messaggi su un nodo. Quest'efficiente filtro probabilistico, noto come bloom filter [13], può indicare con un grado molto elevato di certezza se un messaggio appartiene a uno degli argomenti di interesse. Il nodo tenterà la decrittazione solo se un filtro segnala una possibile corrispondenza. In questo modo grazie all'utilizzo del pattern di comunicazione publish–subscribe, Whisper diventa un protocollo di comunicazione estremamente semplice ed efficiente.

Whisper adotta infine un protocollo proof-of-work per contrastare possibili attacchi di tipo Denial of Service (DoS). Senza questa misura sarebbe infatti possibile attaccare la rete inviando una raffica di messaggi a tutti i nodi (flood attack), o messaggi con un *TTL* (Time-to-Live espresso in secondi) molto elevato che ne causerebbe una persistenza estremamente duratura all'interno della rete (expiry attack). Per questo motivo è stato previsto all'interno dell'envelope un campo chiamato *EnvNonce* all'interno del quale viene memorizzato il risultato della proof-of-work. La PoW di Whisper consiste nell'esecuzione ripetuta di un semplice algoritmo SHA3 con cui si cerca di trovare il numero più piccolo possibile entro un determinato periodo di tempo. Il mittente del messaggio può decidere quanto tempo dedicare a questa operazione attraverso la specificazione di un parametro e tanto più lavoro si sceglie di eseguire localmente, tanto più a lungo verrà mantenuto il messaggio presso i vari nodi e tanto più velocemente questo si propagherà attraverso la rete, acquisendo una priorità maggiore rispetto agli altri messaggi. Infatti ad ognuno di essi viene assegnato un punteggio secondo i seguenti criteri:

- i messaggi più piccoli hanno voti più alti;
- i messaggi con PoW più alta hanno punteggi più alti;
- i messaggi con TTL inferiore hanno punteggi più alti.

Sulla base di questo punteggio ciascun nodo elaborerà i messaggi (e li inoltrerà ai nodi vicini) solo se questo valore supererà una certa soglia, altrimenti verranno eliminati.

1.5 IPFS

IPFS (InterPlanetary File System) è un protocollo e un file system peer-to-peer distribuito pensato per la memorizzazione e la condivisione di contenuti ipermediali [14]. IPFS fornisce un modello di archiviazione a blocchi ad alto throughput e una localizzazione delle risorse basata su collegamenti ipertestuali indirizzati al contenuto. IPFS è stato ideato con l'ambizioso obbiettivo di voler essere la base per la creazione di un nuovo Web completamente distribuito, più veloce, più sicuro e più aperto, ponendosi come alternativa all'attuale architettura client-server e al celeberrimo protocollo HTTP.

I vantaggi che si potrebbero trarre dall'utilizzo di IPFS sarebbero molteplici, ad esempio:

- download più veloci ed efficienti: mentre HTTP può scaricare una risorsa da un solo computer alla volta, IPFS può recuperare parti diverse di un file da molteplici computer contemporaneamente;

- un web più robusto: l'architettura di IPFS elimina i server centrali che rappresentano un *single point of failure* permettendo la creazione di un Web "permanente". IPFS crea una rete resiliente dove ogni risorsa è sempre disponibile grazie al mirroring dei dati su più nodi;

- resistenza alla censura: mentre può essere molto semplice per un governo bloccare l'accesso a un determinato sito internet ospitato presso un server centralizzato, IPFS crea un Web immune alla censura.

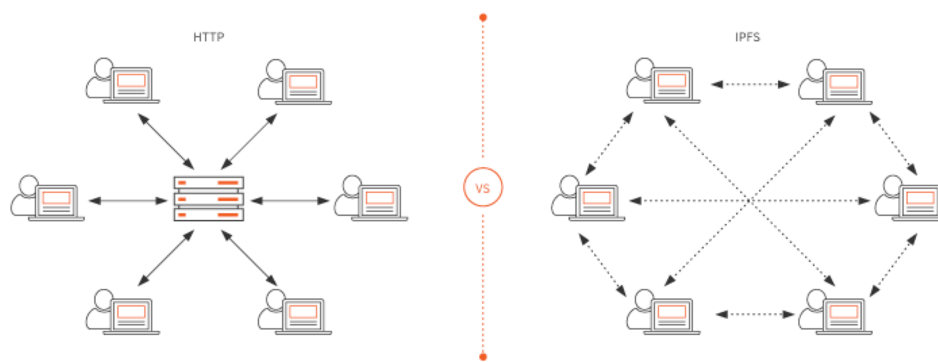

Figura 1.7: Confronto tra HTTP con architettura client server e IPFS con architettura peer-to-peer [15]

IPFS basa il suo funzionamento sull'utilizzo di molteplici tecnologie, racchiudendo in sè le idee di successo che erano alla base di precedenti sistemi peer-to-peer, inclusi DHT, BitTorrent, Git e SFS [14].

Per cercare di spiegare come funziona IPFS, il primo concetto da introdurre è quello di indirizzamento basato sul contenuto, che si contrappone al paradigma di indirizzamento basato sulla posizione utilizzato oggigiorno nel Web. All'interno del World Wide Web infatti, le risorse sono sempre state identificate per mezzo di un URL, che esprime in modo esatto dove è localizzata la risorsa desiderata, specificando l'host a cui è necessario connettersi (mediante dominio o indirizzo IP) e il path per l'individuazione del file all'interno del file system. Questo approccio ha il chiaro svantaggio che qualora il server non risulti raggiungibile per un qualsiasi motivo, la risorsa desiderata non sarà disponibile nonostante è estremamente probabile che almeno un altro computer nel mondo ne stia mantenendo una copia. IPFS supera questo problema identificando le risorse, non mediante la propria posizione, bensì attraverso il loro contenuto. Ogni file in IPFS è infatti identificato attraverso un hash univoco che viene calcolato sul contenuto del file stesso. Qualora un peer desideri scaricare una determinata risorsa, sarà sufficiente

che conosca l'hash di quel file e attraverso un'interrogazione della rete otterrà risposta da uno o più peer che detengono una copia di quella risorsa. L'utilizzo dell'hash come chiave per l'identificazione dei file ha inoltre il vantaggio di garantirne l'integrità: il nodo che ha richiesto una risorsa, per assicurarsi che il file ottenuto sia effettivamente quello richiesto e che non abbia subito alcuna alterazione, non dovrà far altro che ricalcolarne l'hash e verificare che questo corrisponda con quello che aveva inizialmente richiesto. Oltre ad aggiungere questa misura di sicurezza gratuitamente, la scelta di utilizzare l'hash come chiave porta un secondo vantaggio, ovvero la deduplicazione dei file: ogni volta che un utente pubblica un nuovo file su IPFS, la rete verifica attraverso il suo hash se questo è già presente, evitando automaticamente che ne vengano mantenute molteplici copie qualora più utenti caricassero la medesima risorsa.

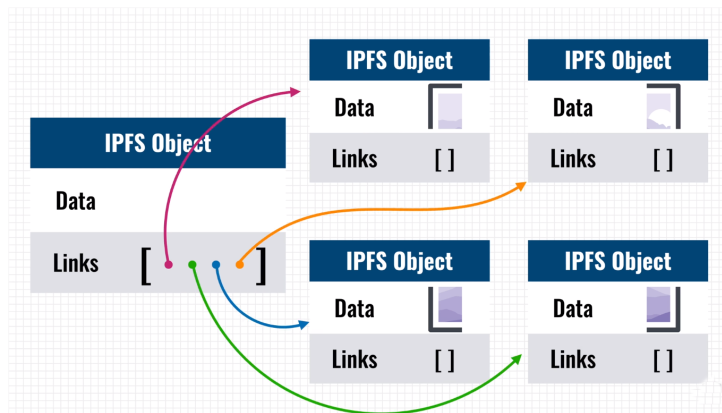

Figura 1.8: Memorizzazione di un file di grandi dimensioni [16]

IPFS memorizza i file all'interno di contenitori chiamati *IPFS object*, ciascuno dei quali può contenere fino a 256KB di dati oltre a una serie di collegamenti ad altri oggetti. Qualora il file da memorizzare avesse dunque una dimensione superiore a 256KB questo verrebbe suddiviso in tanti IPFS object, ciascuno con una dimensione di 256KB, ai quali si aggiungerebbe un ulteriore IPFS object vuoto contenente tutti i collegamenti ordinati ai vari altri oggetti che compongono il file (figura 1.8). Questa architettura permette ad IPFS di modellare efficacemente anche la struttura di un tipico file system, creando un oggetto per ciascun file e directory come viene mostrato in figura 1.9.

Dal momento che IPFS utilizza l'indirizzamento basato sul contenuto, i file memorizzati su di esso non possono più essere modificati, infatti qualunque aggiornamento al file produrrebbe anche una modifica dell'hash che lo

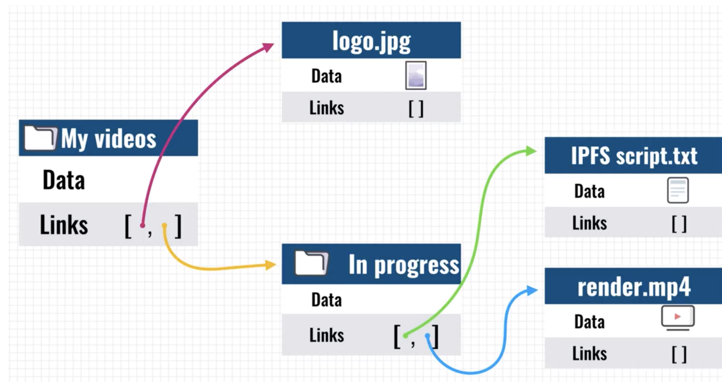

Figura 1.9: File system modellato attraverso IPFS objects [16]

identifica. Per questa ragione IPFS, esattamente come Git, supporta il versioning dei file. Ogni volta che il contenuto di un file viene aggiornato, IPFS crea un oggetto chiamato *commit* che rappresenta un particolare snapshot nella cronologia delle versioni di un determinato IPFS object. Ogni commit contiene un riferimento al commit precedente e un link alla specifica versione dell'IPFS object, come mostrato in figura 1.10. IPFS tiene traccia in questo modo dell'ultima versione del file e di tutte le versioni precedenti.

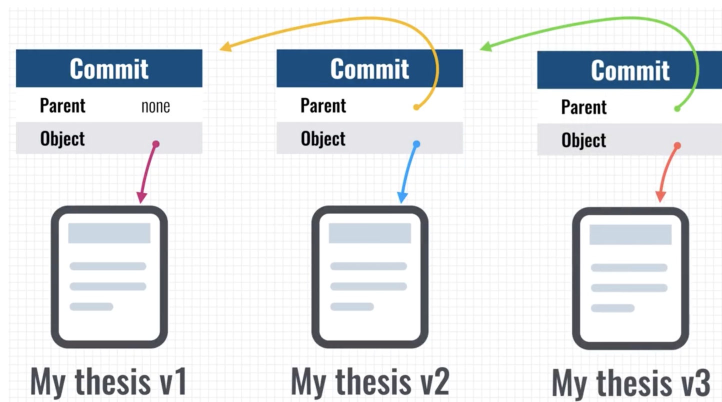

Figura 1.10: File versioning in IPFS [16]

Ogni nodo all'interno della rete memorizza una copia dei file che ha scaricato, oltre ad alcune informazioni di indicizzazione utili a capire quali risorse sta memorizzando ciascun nodo. Il problema principale di IPFS è quello di riuscire a mantenere i file sempre disponibili. Infatti se un file è condiviso da pochi nodi della rete questo diventerà non disponibile non appena questi nodi andranno offline, esattamente come capita su BitTorrent per i file con pochi seeders. L'unico modo per risolvere questo problema è offrire un in-

centivo ai nodi per rimanere online quanto più tempo possibile e distribuire proattivamente i file sulla rete in modo che ci sia sempre un certo numero minimo di nodi che ne mantenga una copia disponibile. Per questo gli stessi sviluppatori di IPFS hanno creato Filecoin, una blockchain basata su IPFS che punta alla creazione di un mercato decentralizzato per lo storage di dati. In questo modo chi offre il proprio spazio di archiviazione per il salvataggio di dati su IPFS è incentivato economicamente a farlo e a mantenere il proprio nodo online per quanto più tempo possibile.

In sintesi IPFS può essere visto come un singolo sciame BitTorrent che scambia oggetti all'interno di un repository Git. È possibile accedere al file system IPFS in vari modi, tra cui anche FUSE e HTTP.

Capitolo 2

Requisiti e Analisi

In questo capitolo verrà descritto il caso di studio e successivamente affrontata l'analisi dei requisiti, ovvero si descriveranno nel dettaglio tutte le funzionalità e tutte le caratteristiche che dovranno essere implementate dall'applicazione decentralizzata oggetto di questa tesi.

2.1 Descrizione del caso di studio

Questa tesi si pone come obiettivo la realizzazione di un'applicazione web totalmente decentralizzata basata su blockchain adibita all'affitto di casa vacanza a breve termine, in maniera similare a quanto già realizzato da piattaforme online quali Booking e Airbnb, divenute molto popolari e utilizzate negli ultimi anni.

La piattaforma sarà utilizzata e dovrà essere in grado di gestire due tipologie di utenti: i clienti, in cerca di una sistemazione per la loro prossima vacanza, e i proprietari delle strutture, coloro che hanno deciso di mettere a disposizione il proprio appartamento per ospitare i visitatori in vacanza.

L'applicazione dovrà quindi innanzitutto permettere ai soggetti ospitanti di creare degli annunci dove presentare la loro struttura e la loro offerta. Tali annunci dovranno presentare tutte le informazioni di base utili ai clienti, quali ad esempio: una descrizione, una galleria fotografica, il prezzo per notte, i servizi disponibili e l'indirizzo. I proprietari delle strutture dovranno anche essere in grado di modificare in ogni momento i propri annunci, di rimuoverli e di consultare per ciascuna sistemazione l'elenco delle prenotazioni ricevute.

I clienti invece dovranno poter esplorare tutte le sistemazioni disponibili, applicando eventualmente anche dei filtri di ricerca per scremare e visualizzare solo gli appartamenti che rispettano determinati requisiti di interesse. Per ogni alloggio il cliente dovrà poter accedere alla scheda completa contenente

tutte le informazioni e le fotografie messe a disposizione dall'inserzionista. Una volta individuata una sistemazione disponibile di proprio gradimento, l'utente dovrà poter effettuare attraverso il sito una prenotazione, andando a specificare il periodo e il numero di notti del soggiorno. Il cliente dovrà anche poter consultare in ogni momento lo storico delle sue prenotazioni.

La piattaforma dovrà inoltre provvedere alla gestione dei pagamenti dei soggiorni, andando ad offrire anche un servizio di tutela e garanzia per entrambe le parti. All'atto della prenotazione il sito dovrà richiedere al cliente il versamento della somma necessaria a coprire le spese dell'intero soggiorno, garantendo così al soggetto ospitante la completa solvibilità da parte del cliente. Tali fondi rimarranno però custoditi all'interno della piattaforma stessa fino a tre giorni prima della data di check-in prevista, in modo che il cliente possa esercitare fino a tale termine il diritto di cancellazione della propria prenotazione e quindi ottenere il rimborso della somma versata. Solo quando mancheranno meno di 72 ore alla data di check-in l'applicazione effettuerà il pagamento del soggiorno al titolare dell'annuncio.

Infine si intende sviluppare, oltre alla suddetta applicazione web, un software per un sistema embedded in grado di simulare il funzionamento di una serratura smart key-less, che sia in grado di concedere automaticamente il diritto ad accedere all'appartamento soltanto al proprietario e agli eventuali ospiti che vi hanno prenotato un soggiorno in quel periodo. Il software dovrà quindi interagire con il sistema precedente e interrogare la blockchain per determinare se l'utilizzatore ha il diritto di impartire comandi alla serratura stessa. Le uniche operazioni permesse saranno l'apertura e la chiusura della serratura. La realizzazione di questa semplice applicazione embedded avrà lo scopo di analizzare come il mondo blockchain e il mondo Internet of Things possono interagire tra loro per offrire servizi smart tangibili nel mondo reale, in questo caso specifico la possibilità per il cliente di effettuare il check-in e il check-out presso la struttura in modo completamente autonomo senza che il proprietario dell'appartamento debba incontrarsi fisicamente con l'ospite per consegnargli e ritirare le chiavi dell'alloggio.

2.2 Analisi dei requisiti

In questa sezione si andranno a schematizzare per punti i requisiti delle applicazioni che dovranno essere sviluppate.

2.2.1 Requisiti funzionali

Di seguito verranno elencati i requisiti funzionali richiesti in fase di creazione del progetto, ovvero le funzionalità che dovranno essere rese disponibili dalle applicazioni. I punti sono stati suddivisi in due insiemi, ovvero in funzionalità dedicate ai proprietari degli appartamenti, coloro che sulla piattaforma ricopriranno il ruolo di inserzionisti, e in funzionalità dedicate ai clienti, coloro che invece prenoteranno soggiorni presso le strutture disponibili per le loro vacanze. Si noti che un utente della piattaforma potrà essere contemporaneamente sia inserzionista che cliente.

Requisiti funzionali per gli inserzionisti:

- **creazione di un nuovo annuncio:** il sito dovrà permettere l'aggiunta di una nuova struttura richiedendo l'inserimento di una serie di informazioni essenziali quali ad esempio una descrizione, una galleria fotografica, il prezzo per notte, i servizi disponibili e l'indirizzo;
- **modifica di un annuncio:** il titolare di un annuncio dovrà poterlo modificare in qualsiasi momento e in ogni sua parte tranne per quanto riguarda campi particolari immutabili quali ad esempio l'indirizzo;
- **rimozione di un annuncio:** il titolare di un annuncio dovrà poter cancellare in ogni momento un proprio annuncio. La cancellazione provocherà la rimozione della struttura dal catalogo di quelle disponibili e impedirà che possano essere fatte nuove prenotazioni, ma quelle già in essere continueranno ad essere valide. Opzionalmente potrebbe essere prevista la possibilità di recuperare un'inserzione disabilitata;
- **possibilità di consultare le prenotazioni di una struttura:** il titolare di un annuncio dovrà poter visualizzare un'elenco delle prenotazioni previste presso la propria struttura in modo da poter preparare l'abitazione al meglio per poter accogliere i nuovi ospiti;
- **possibilità di richiedere il pagamento di una prenotazione:** il titolare di un annuncio potrà ottenere il pagamento relativo ad una prenotazione soltanto a partire da tre giorni prima della data di check-in prevista;
- **accesso alla struttura:** la serratura smart dovrà riconoscere il diritto di accesso al proprietario della struttura in ogni momento.

Requisiti funzionali per i clienti:

- **possibilità di esplorare e ricercare fra le strutture disponibili:** il cliente dovrà poter visualizzare un'elenco di tutte le strutture disponibili e dovrà essere in grado di effettuare anche delle ricerche con vari filtri per riuscire a individuare più velocemente una sistemazione idonea per il suo prossimo soggiorno;

- **visualizzazione di un annuncio:** il cliente per ogni struttura dovrà poter visualizzare una scheda contenente tutte le informazioni e le fotografie messe a disposizione dall'inserzionista;

- **possibilità di visualizzare le date disponibili per una determinata struttura:** il cliente dovrà poter consultare le date ancora disponibili e quindi prenotabili per una determinata struttura;

- **creazione di una prenotazione:** il cliente dovrà poter effettuare una prenotazione, previa scelta di un periodo disponibile, effettuando il pagamento del costo dell'intero soggiorno;

- **cancellazione di una prenotazione:** il cliente dovrà essere in grado di annullare una prenotazione e di ottenere dunque un rimborso della quota versata fino a tre giorni prima della data di check-in prevista;

- **accesso alla struttura:** la serratura smart dovrà riconoscere diritto di accesso al cliente di turno per tutto il tempo che intercorre dalla data di check-in alla data di check-out concordate.

2.2.2 Requisiti non funzionali

Di seguito verranno raccolti tutti i requisiti non funzionali, ovvero tutte le caratteristiche che pur non essendo funzionalità dovranno essere soddisfatte dai sistemi realizzati. I requisiti non funzionali individuati sono i seguenti:

- **totale decentralizzazione dell'applicazione:** la piattaforma da sviluppare dovrà essere completamente decentralizzata, non solo per quanto riguarda l'aspetto computazionale attraverso l'uso di una blockchain che offra funzionalità di smart contract ma anche per quanto riguarda l'eventuale storage necessario in cui memorizzare file caricati dagli utenti o anche il codice sorgente stesso dell'applicazione. Non dovrà quindi essere presente un web server centrale;

- **possibilità di visualizzare il codice sorgente:** l'utente della piattaforma dovrà poter visualizzare il codice sorgente dell'applicazione web per poterne verificare di persona il funzionamento, in particolare come funziona il servizio di escrow (custodia dei depositi) dei pagamenti relativi alle prenotazioni;

- **utilizzo gratuito della piattaforma:** all'utente non dovrà essere richiesto di sostenere alcuna spesa per l'effettuazione di operazioni di semplice consultazione o ricerca sul sito web, e per l'utilizzo della serratura smart dell'appartamento.

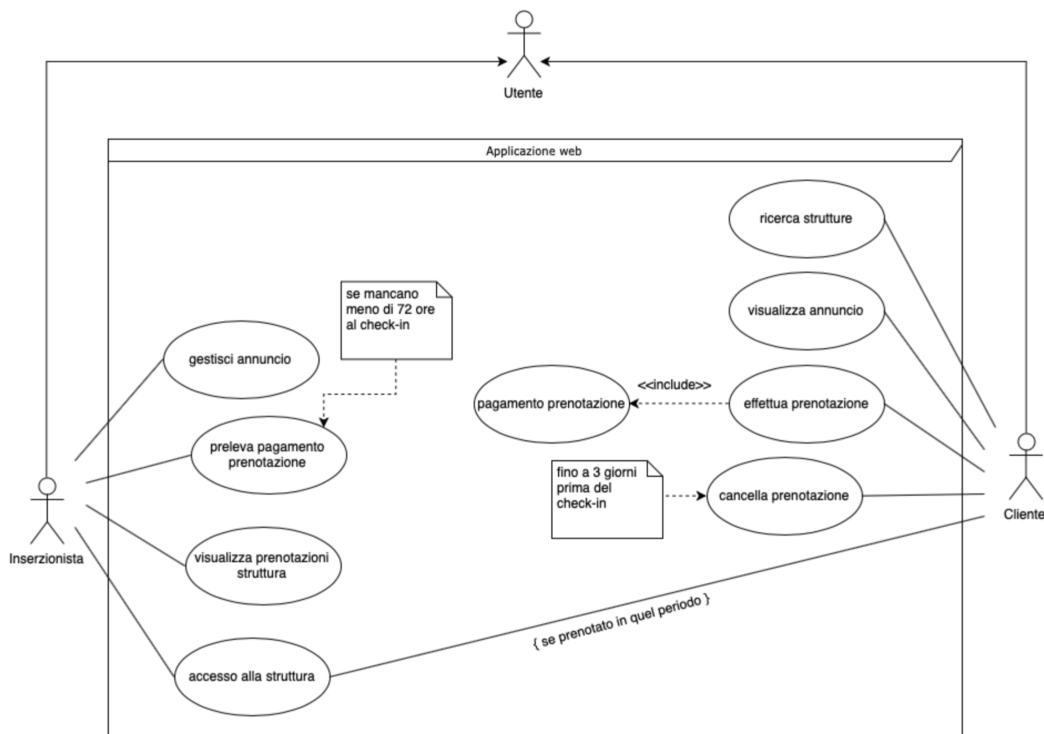

Figura 2.1: Descrizione dei requisiti funzionali mediante diagramma UML dei casi d'uso

Capitolo 3

Progettazione architetturale

In questo capitolo verranno descritte le principali scelte architetturali e tecnologiche che sono state prese per fare in modo di soddisfare i requisiti definiti nella precedente fase di analisi.

3.1 Scelta delle tecnologie

Prima di poter procedere alla progettazione dell'architettura del sistema da realizzare si è resa necessaria l'individuazione delle tecnologie da utilizzare in fase di sviluppo per poter comprendere come queste potessero interagire tra loro e soddisfare tutti i requisiti funzionali e non funzionali emersi dalla precedente fase di analisi.

La prima decisione è stata chiaramente quella relativa alla scelta della blockchain su cui basare la logica dell'intera applicazione. In particolare si è dovuto decidere se optare per una blockchain pubblica, come ad esempio Ethereum, o una blockchain permissioned, come invece Hyperledger Fabric. Per lo sviluppo dell'applicazione web oggetto di questa tesi, alla fine si è deciso di optare per l'utilizzo di una blockchain pubblica, dal momento che tra i requisiti individuati era richiesta anche la gestione dei pagamenti relativi alle prenotazioni e una totale trasparenza sul contenuto delle transazioni e sul codice sorgente dello smart contract memorizzati sulla blockchain stessa. Contrariamente, l'utilizzo di una blockchain privata avrebbe richiesto anche la creazione di un nuovo token di scambio, dal momento che solitamente questa tipologia di blockchain non dispone di una criptovaluta nativa, scelta che avrebbe aumentato in maniera significativa i tempi di sviluppo per la realizzazione di un aspetto di scarso interesse per gli obiettivi di questa tesi.

In particolare si è scelto dunque di utilizzare la blockchain di Ethereum, in quanto tra le blockchain pubbliche è quella ad oggi più matura, e quin-

di quella con il più grande numero di sviluppatori e il più ampio parco di framework, tool e documentazione di supporto allo sviluppo di applicazioni decentralizzate esistente. La criptovaluta di Ethereum, l'Ether, è inoltre al momento la seconda criptovaluta più capitalizzata al mondo dopo Bitcoin e quindi una delle più note e utilizzate sul web, permettendo così di raggiungere una platea di possibili utilizzatori molto più ampia rispetto a quella che si sarebbe ottenuta utilizzando una piattaforma minore.

Un'altra decisione che è stata fondamentale operare dal principio, è stata quella relativa alla scelta della tecnologia per lo storage decentralizzato del codice sorgente della web app e degli altri file dati che fosse stato necessario memorizzare, come ad esempio le fotografie degli appartamenti. Un altro requisito non funzionale emerso nel precedente capitolo, richiedeva infatti che l'applicazione fosse totalmente decentralizzata, e che quindi non utilizzasse nessun server centrale, nemmeno per l'hosting del sito web. Per risolvere questo problema si è pensato di utilizzare IPFS, un file system decentralizzato il cui funzionamento è già stato descritto all'interno della sezione 1.5. Combinando l'uso di Ethereum e di IPFS è dunque possibile realizzare applicazioni totalmente decentralizzate la cui architettura risulta essere quella mostrata in figura 3.1.

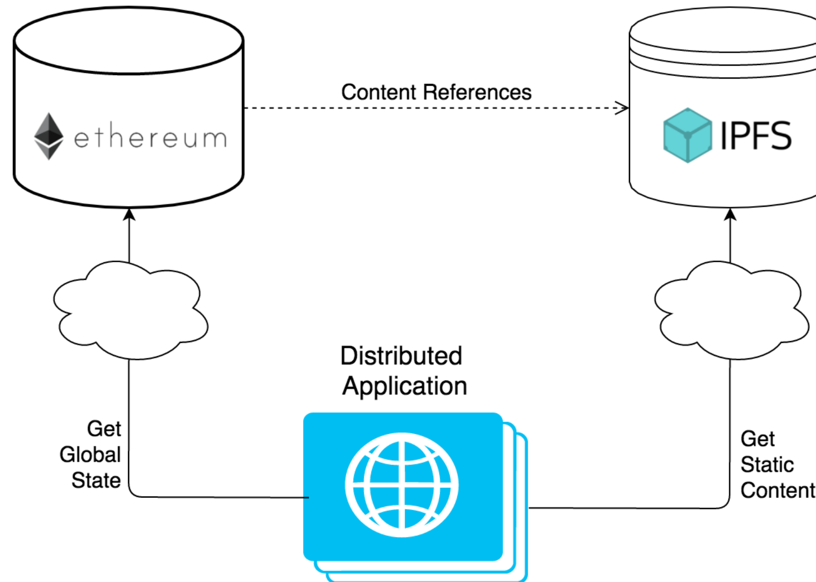

Figura 3.1: Architettura di un'applicazione decentralizzata basata su Ethereum e IPFS

La scelta di non utilizzare alcun web server e di ospitare i sorgenti dell'applicazione su IPFS pone alcune limitazioni importanti che influenzeranno

drasticamente la fase di sviluppo. Tali misure precluderanno infatti l'utilizzo di un qualunque linguaggio di programmazione server-side tradizionale (come ad esempio PHP o ASP.NET) e di conseguenza l'applicazione web dovrà essere interamente sviluppata utilizzando solo linguaggi eseguibili client-side quali HTML e Javascript. Questa architettura prevede infatti che la computazione server-side dovrà essere totalmente realizzata dallo smart contract in esecuzione sulla blockchain.

Queste due tecnologie combinate possono quindi risultare sufficienti per realizzare gran parte delle applicazioni decentralizzate, con Ethereum che si occupa di memorizzare all'interno della blockchain lo stato corrente dell'applicazione, realizzando le stesse funzioni di un database, e di eseguirne la logica business, codificata all'interno di uno smart contract, grazie alla sua EVM (Ethereum Virtual Machine), mentre IPFS mantiene in modo persistente tutte le risorse statiche che non potrebbero essere memorizzate direttamente all'interno della blockchain in quanto risulterebbe estremamente costoso. Questo è dovuto al fatto che tutti i nodi della rete devono mantenere in locale una copia dell'intera blockchain, e se quest'ultima fosse utilizzata come un repository, invece che come un database per la memorizzazione di semplici transazioni, la sua dimensione esploderebbe esponenzialmente nel giro di pochissimo tempo divenendo completamente ingestibile. Pertanto all'interno della blockchain ci si limita a memorizzare dei riferimenti ai file, ovvero gli hash IPFS che identificano le risorse su di esso memorizzate.

Per la realizzazione di questo sistema, l'architettura individuata risulta però essere ancora non soddisfacente, in quanto persiste un problema che viola uno dei requisiti evidenziati in fase di analisi. La scelta di utilizzare una blockchain pubblica ha portato con sé infatti lo svantaggio di richiedere il pagamento di una piccola commissione per l'esecuzione di ogni operazione, destinata a ricompensare i miner del lavoro computazionale svolto per aver validato una transazione ed aver eventualmente eseguito del codice di uno smart contract. Ciò è chiaramente un problema, dal momento che l'utente utilizzatore dell'applicazione non si aspetta di dover pagare per l'utilizzo della piattaforma, in particolare per quanto riguarda alcune funzionalità. Infatti mentre l'utente può tollerare di pagare una piccola commissione per la pubblicazione di un annuncio o contestualmente al pagamento di una prenotazione, difficilmente sarà disposto a pagare per effettuare una ricerca o inviare dei comandi alla serratura smart. Fortunatamente su Ethereum è necessario pagare le commissioni di mining solo quando la computazione richiesta comporta un aggiornamento dei dati memorizzati all'interno della blockchain, mentre se si opera in sola lettura tale pagamento non è richiesto. Questo comportamento è dovuto al fatto che un'operazione di aggiornamento deve essere eseguita necessariamente da tutti i nodi della rete, mentre per

leggere i dati è sufficiente fare una richiesta al solo nodo a cui l'applicazione si connette direttamente e che funge come punto di ingresso alla rete Ethereum. Analizzando dunque tutti i requisiti funzionali emersi in fase di analisi, si nota che il problema persiste soltanto per quanto riguarda le interazioni con la serratura smart.

Dal momento che l'idea di utilizzare una funzione dello smart contract per aprire e chiudere la serratura non può essere praticabile, in quanto è impensabile che un cliente sia disposto a pagare ogni qual volta desideri aprire o chiudere il portone dell'appartamento, si è resa necessaria l'introduzione di una terza tecnologia volta ad introdurre lo possibilità di scambiare messaggi direttamente tra due entità, in questo caso tra l'applicazione web e la serratura smart. Per questa ragione è stato introdotto Whisper, un protocollo di messaggistica peer-to-peer per applicazioni decentralizzate facente parte della suite di tecnologie sviluppate da Ethereum per la realizzazione del Web 3.0. Il funzionamento di Whisper è già stato affrontato all'interno della sezione 1.4. Whisper, con il suo servizio di messaggistica decentralizzata, rappresenta l'ultimo elemento necessario per riuscire a sviluppare una dApp arbitrariamente complessa, andando quindi a completare il set di tecnologie necessarie per progettare l'architettura del sistema da realizzare.

3.2 Architettura del sistema

L'architettura finale del sistema risulta dunque essere quella descritta in figura 3.2. Tale strutturazione rispecchia quella del noto design pattern architetturale MVC (Model-View-Controller) [17], caratterizzato da tre tipi di componenti:

- il **model**: incapsula la business logic e i dati utili all'applicazione fornendo attraverso la sua interfaccia dei metodi per accedervi e manipolarli;
- la **view**: realizza l'interfaccia grafica, quindi visualizza i dati ottenuti dal model e si occupa dell'interazione con l'utente;
- il **controller**: funge da intermediario tra view e model, raccoglie quindi i comandi dell'utente provenienti dalla view e li attua modificando lo stato degli altri due componenti.

La potenza di questo pattern risiede nel fatto che riesce a disaccoppiare l'interazione fra view e model, aspetto particolarmente importante in applicazioni di questo tipo dal momento che tutte le interazioni con il model avvengono in modo asincrono.

Nell'applicazione web oggetto di studio, la componente model è realizzata dallo smart contract Ethereum, mentre la componente view è realizzata dalle pagine web HTML. Il ruolo di controller è invece eseguito dal layer Javascript che interfaccia la view e il model attraverso l'utilizzo della libreria web3.js.

Si noti come l'applicazione web interagisce anche con il sistema IOT che realizza la serratura smart mediante lo scambio di messaggio diretti attraverso l'utilizzo del protocollo Whisper. Il dispositivo embedded resta sempre in attesa di ricevere nuovi messaggi contenenti i comandi per aprire o chiudere la serratura e una volta ricevuti interroga direttamente lo smart contract per verificare se il mittente dei messaggi dispone dei permessi necessari ad impartire i comandi richiesti.

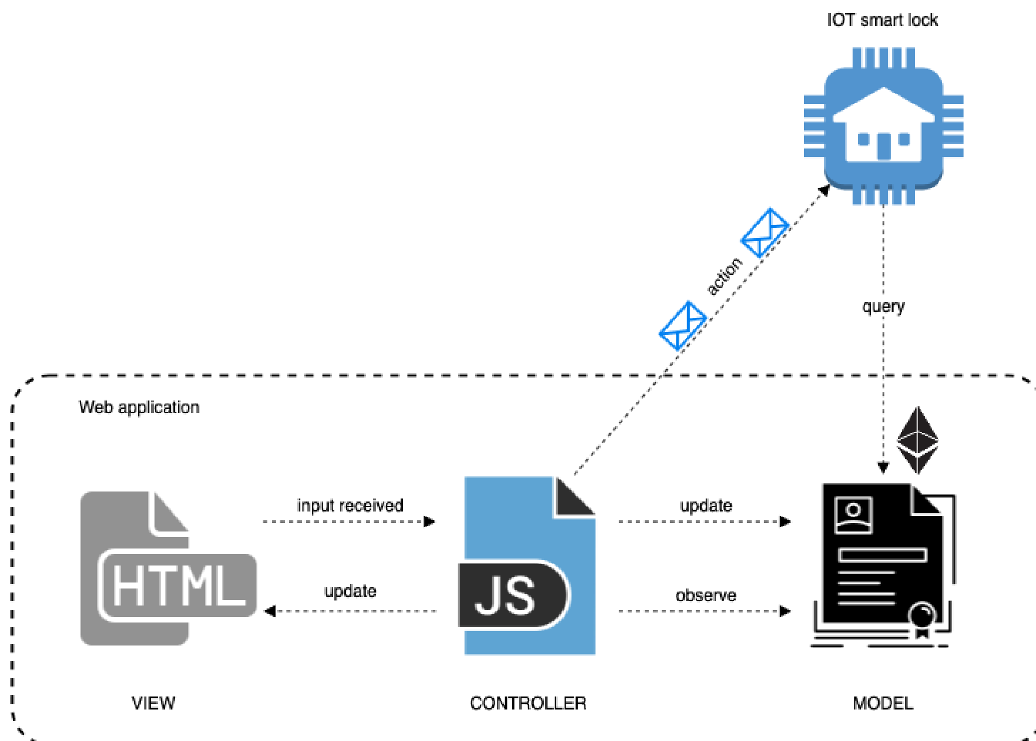

Figura 3.2: Architettura software del sistema

Per poter utilizzare l'applicazione web decentralizzata l'utente dovrà disporre di un browser in grado di interagire con la blockchain di Ethereum. A tale scopo le soluzioni possibili sono molteplici, una delle quali è messa a disposizione direttamente dalla Ethereum Foundation, ovvero il browser Mist.

Mist è un software che integra al suo interno tutti gli strumenti necessari per la gestione di un wallet, per il deployment di smart contract e persino

un browser abilitato al web 3.0, ovvero all'utilizzo di dApps, pensato appositamente per un pubblico di non addetto ai lavori. Mist è però un software ancora in fase di sviluppo, molto lento e con alcune criticità di sicurezza tuttora irrisolte, pertanto una possibile migliore alternativa, spesso suggerita, è rappresentata da Metamask, un'estensione per i browser tradizionali come Google Chrome, Mozilla Firefox e Opera che li abilita all'utilizzo delle dApps, senza che l'utente abbia nemmeno la necessità di eseguire un client Ethereum sul proprio computer, come Geth o Parity.

Metamask rende l'accesso al mondo delle dApps basate su Ethereum estremamente facile e alla portata di tutti. Anche Metamask mette a disposizione degli strumenti per una gestione completa dei propri wallet, consentendo di effettuare transazioni, firmare messaggi, aggiungere ed esportare chiavi private, gestire molteplici account e passare dalla rete principale di Ethereum a una qualsiasi testnet con un click.

Sono molteplici dunque i layer che costituiscono l'architettura di un'applicazione web decentralizzata basata su Ethereum. Lo stack risultante viene mostrato graficamente dall'immagine 3.3.

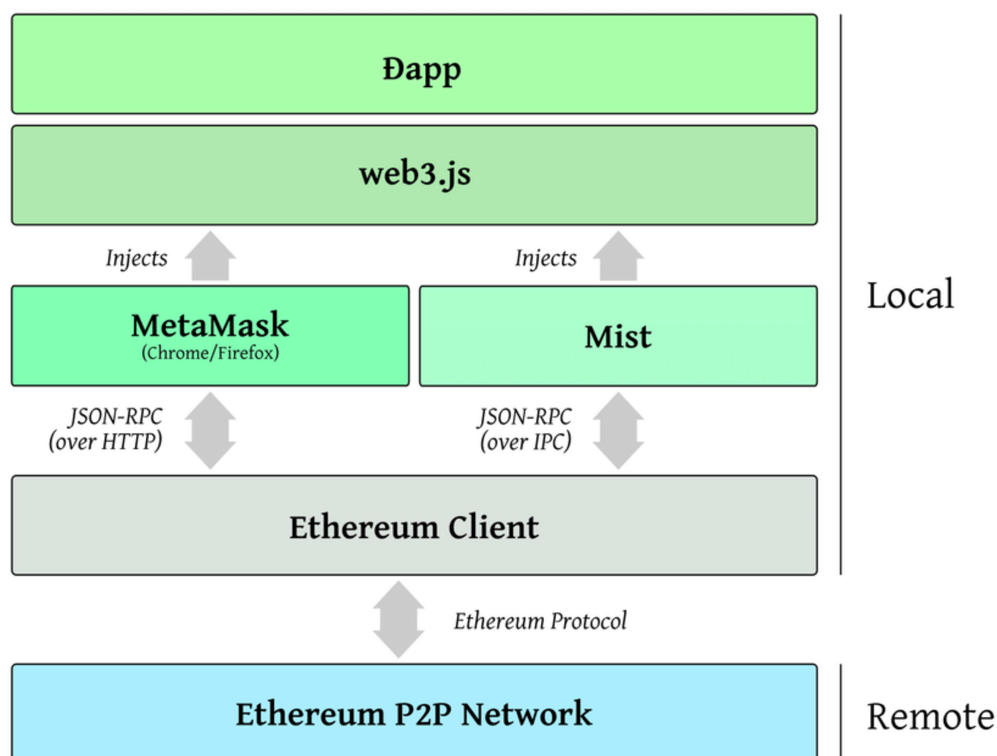

Figura 3.3: Stack architetturale di una dApp Ethereum

Grazie a servizi come Infura [1], è inoltre possibile evitare l'installazione e la configurazione di un client Ethereum e IPFS in locale, risparmiando così risorse sia in termini computazionali che di tempo e di denaro per la realizzazione dell'infrastruttura necessaria. Infura fornisce infatti agli sviluppatori di applicazioni decentralizzate strumenti ed API di facile utilizzo per consentire un accesso sicuro, affidabile e scalabile a Ethereum e IPFS. In questo progetto Infura è stato però utilizzato solo per la parte riguardante IPFS in quanto gli endpoints pubblici messi a disposizione da Infura per collegarsi alla rete Ethereum non espongono ancora le API necessarie per il funzionamento di Whisper. Per l'esecuzione di questa specifica applicazione è pertanto richiesto che l'utente abbia in esecuzione sul proprio computer un'istanza di un client Ethereum, come Geth o Parity, con il protocollo Whisper abilitato.

[1] https://infura.io

Capitolo 4

Implementazione

In questo capitolo verrà mostrato come sono state sviluppate le parti più significative del sistema. Si cercherà di dare spazio a tutte le tecnologie utilizzate e in particolare al modo in cui queste interagiscono tra loro, con l'obiettivo di chiarire ulteriormente i dettagli e le scelte descritti in fase di progettazione.

4.1 Configurazione della testnet

Prima di analizzare l'implementazione di alcune parti del sistema, è importante chiarire il contesto nel quale viene eseguita l'applicazione.

Per sviluppare applicazioni decentralizzate basate su Ethereum, è necessario per prima cosa configurare una blockchain di test che utilizzi una criptovaluta fittizia, ovvero un ambiente che consenta di effettuare molteplici deployment di smart contract e transazioni di prova senza che sia necessario sostenere un esborso economico reale. E' possibile utilizzare sia testnet pubbliche online, che costruirsi delle proprie testnet private in locale. Quest'ultima possibilità è messa a disposizione direttamente dal client Ethereum Geth ufficiale.

Per configurare una nuova blockchain Ethereum in locale è necessario quindi creare un file JSON contente una serie di proprietà che caratterizzeranno la blockchain stessa, tra le quali si evidenziano:

- `difficulty`: valore scalare corrispondente al livello di difficoltà applicato durante il processo di mining di un blocco, e quindi proporzionale al tempo che sarà necessario per generarlo;

- `gaslimit`: valore scalare che definisce, a livello di blockchain, la quantità massima di gas utilizzabile per blocco;

- `chainId`: valore che identifica la catena corrente. Viene utilizzato per implementare una protezione contro i replay attack.

Per lo sviluppo di questo progetto è stata utilizzata una configurazione standard, ovvero la seguente:

```
{
    "difficulty" : "0x80000",
    "extraData"  : "",
    "gasLimit"   : "0x8000000",
    "alloc": {},
    "config": {
        "chainId": 15,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
    }
}
```

Una volta creato il file di configurazione, è possibile procedere all'inizializzazione della blockchain, attraverso l'esecuzione del comando:

```
geth --datadir=./blockchain/ init ./genesis.json
```

Completato il processo di inizializzazione dovrà essere generato anche un primo account Ethereum che fungerà da coinbase, ovvero da wallet principale all'interno del quale Geth custodirà gli Ether falsi utilizzabili per i vari test, ottenuti come ricompensa per il mining. Dopodiché sarà possibile lanciare l'esecuzione del client Ethereum specificando tutti i parametri che saranno necessari per il funzionamento dell'applicazione. Il comando da eseguire risulterà dunque essere il seguente:

```
geth --datadir=./blockchain/ --shh --rpc
--rpcaddr "0.0.0.0" --rpcport "8545" --rpccorsdomain '*'
--networkid=15 --rpcapi "db,eth,net,web3,shh,personal"
--mine --minerthreads "1"
```

Di seguito viene descritto il significato di ciascun flag utilizzato:

- `datadir`: directory dove è memorizzata la blockchain;
- `shh`: abilita il protocollo Whisper;
- `rpc`: avvia l'interfaccia HTTP JSON-RPC;
- `rpcaddr`: interfaccia di ascolto del server HTTP-RPC;

- `rpcport`: porta di ascolto del server HTTP-RPC;
- `rpccorsdomain`: lista di domini da cui accettare richieste;
- `networkid`: identificatore della rete (deve corrispondere al `chaidId` affinché Metamask non sollevi eccezioni);
- `rpcapi`: elenco di API offerte attraverso l'interfaccia HTTP-RPC.
- `mine`: abilita il mining;
- `minerthreads`: numero di thread dedicati al processo di mining;

Con questa configurazione si esaurisce il lavoro necessario a creare l'infrastruttura tecnologica di test necessaria per il deployment dell'applicazione decentralizzata.

4.2 Creazione di un annuncio

La creazione di un annuncio consiste nell'inserimento, da parte del proprietario di un appartamento, di tutte le informazioni relative a una struttura da aggiungere al catalogo del sito.

Si ricorda che il form di inserimento di una nuova struttura prevede anche la possibilità di arricchire l'annuncio allegando una o più fotografie dell'appartamento, che dovranno quindi essere caricate su IPFS. Di seguito vengono mostrate alcune parti di codice sorgente che realizzano questa funzionalità.

```
1  function initIPFS() {
2    ipfs = new window.IpfsHttpClient('ipfs.infura.io', '5001',
       { protocol: 'https' });
3    ipfs.id(function(err, res) {
4      if (err) {
5        console.error(err);
6      } else {
7        console.log("Connected to IPFS node!", res.id,
8          res.agentVersion, res.protocolVersion);
9      }
10   });
11 }
```

Non appena la pagina di inserimento viene caricata, la componente JavaScript esegue la funzione `initIPFS` che, attraverso l'utilizzo della libreria `IpfsHttpClient`, inizializza una variabile globale con un riferimento alla connessione instaurata con un nodo IPFS remoto messo a disposizione da Infura.

Al momento dell'invio del form viene invece eseguita la porzione di codice che si occupa dell'upload delle immagini vero e proprio.

```
var imageHashes = [];
var buffer = window.IpfsHttpClient().Buffer;
var uploadCounter = 0;

for(i = 0; i < images.length; i++) {
  var reader = new FileReader();
  reader.readAsArrayBuffer(images[i]);
  reader.onload = function() {
    var mybuffer = buffer.from(this.result);
    ipfs.add(mybuffer, function(err, result) {
      if(err) {
        console.error(err);
        showSection("permission_denied");
      } else {
        var ipsfHash = result[0].hash;
        ipfs.pin.add(ipsfHash);
        imageHashes.push(ipsfHash);
        if(++uploadCounter == images.length) {
          if($("#btnSubmit").text() == "Crea annuncio") {
            createContract(imageHashes);
          } else {
            updateContract(imageHashes);
          }
        }
      }
    });
  }
}
```

All'interno del vettore `images` vi sono memorizzati i riferimenti alle risorse che l'utente ha precedentemente indicato di voler caricare, mediante l'utilizzo del componente HTML per il caricamento di file. Ogni elemento dell'array viene letto dal disco per mezzo di un oggetto FileReader e successivamente inserito all'interno di un buffer che viene passato come parametro di input alla funzione di upload su IPFS visibile a riga 10. Se non si verificano errori, IPFS restituirà come risultato un oggetto contenente l'hash del file, che verrà aggiunto ad un array chiamato `imageHashes`. Si ripete dunque la procedura di upload per ogni immagine selezionata dall'utente, prima di procedere al salvataggio della struttura sulla blockchain richiamando il metodo `createContract` (o `updateContract` in caso di aggiornamento di un annuncio preesistente).

Il contenuto del metodo `createContract`, che realizza il salvataggio sulla blockchain, è a questo punto piuttosto semplice: dopo aver recuperato dal form HTML tutti i dati inseriti, si limita ad invocare la funzione

addStructure messa a disposizione dallo smart contract, passandogli tutti i parametri richiesti. Sulla blockchain, per quanto riguarda le immagini, verranno memorizzati soltanto i loro hash IPFS. Questi, identificando univocamente le immagini, saranno infatti sufficienti per riuscire a recuperarle in secondo momento quando dovranno essere mostrate.

Si noti ora come avviene il passaggio di questi hash, contenuti all'interno dell'array imageHashes: poiché Solidity non accetta array bidimensionali come parametri di una funzione, e un array di stringhe viene considerato tale, tutti gli hash vengono concatenati in un'unica stringa utilizzando il punto come elemento separatore.

```
1 await App.contractInstance.addStructure(title, description,
    services, guestsNum, bedsNum, dailyPrice, address, city,
    district, postalCode, imageHashes.join('.'),
2     { from: App.userAccount });
```

Si omette ora l'implementazione del metodo addStructure, contenuto nello smart contract, in quanto ritenuto di scarso interesse per il lettore, dal momento che si limita ad effettuare il salvataggio dei dati ricevuti in input.

Si mostra invece come è stato modellato il concetto di struttura ospitante: dal momento che Solidity non è un linguaggio di programmazione orientato agli oggetti, si è dovuto ricorrere all'utilizzo di una serie di strutture che raggruppassero tutti gli attributi necessari al loro interno.

```
1  struct Services {
2      bool hasWifi;
3      bool hasPark;
4      bool hasAirConditioning;
5      bool hasTV;
6      bool hasKitchen;
7  }
8
9  struct Location {
10     string structureAddress;
11     string city;
12     string district;
13     string postalCode;
14 }
15
16 struct Structure {
17     address owner;
18     uint structureID;
19     string title;
20     string description;
21     Services services;
22     uint guestsNum;
23     uint bedsNum;
```

```
24      uint dailyPrice;
25      Location location;
26      string[] photosHashes;
27      uint[] reservationIDs;
28      bool active;
29  }
```

4.3 Creazione di una prenotazione

La possibilità di creare prenotazioni è una delle funzionalità principali messe a disposizione dall'applicazione web decentralizzata. L'utente, una volta individuata la struttura presso la quale vorrebbe soggiornare, seleziona attraverso un datepicker una data per il check-in e una data per il check-out disponibili, dopodiché effettua immediatamente il pagamento per saldare l'intera prenotazione, versando un importo congruo per il numero di notti del soggiorno. Per realizzare questa funzionalità sono quindi molteplici i controlli necessari per verificare che non ci siano state sovrapposizioni o anomalie nella scelta del periodo e che sia stato versato correttamente l'importo richiesto.

```
1  $("#btnSubmit").click(async function() {
2    if($("#period").val().length == 0) {
3      return;
4    }
5
6    var dates = datePicker.getValue().split(" - ");
7    var checkInTimestamp = new Date(dates[0]).getTime() / 1000;
8    var checkOutTimestamp = new Date(dates[1]).getTime() /
      1000;
9    var numDays = (checkOutTimestamp - checkInTimestamp) / 60 /
       60 / 24;
10   var balance = price * numDays;
11   checkInTimestamp += 14 * 60 * 60;
12   checkOutTimestamp += 10 * 60 * 60;
13
14   try {
15     await App.contractInstance.addReservation(contractID,
16                                checkInTimestamp,
17                                checkOutTimestamp,
18                                { from: App.userAccount,
19                                  value: balance });
20     showSection("success");
21   } catch(err) {
22     console.error(err);
23     showSection("permission_denied");
24   }
25 });
```

All'invio del form di prenotazione, viene scatenato il precedente handler JavaScript che dopo aver effettuato qualche operazione di validazione e conversione invoca il metodo `addReservation` dello smart contract passandogli in input i dati della prenotazione. Si noti che le date di check-in e check-out vengono passate sotto forma di unix timestamp e che tra i parametri aggiuntivi della transazione, visibili alle righe 20 e 21, oltre ad esserci indicato l'indirizzo pubblico del mittente, questa volta vi è anche un attributo value. Il valore memorizzato all'interno di questo campo corrisponde alla somma di Ether che verrà allegata alla transazione, realizzando così il pagamento. La somma specificata, verrà quindi trasferita dal wallet del mittente a quello dello smart contract.

```
1  function addReservation(uint _structureID, uint
       _checkInTimestamp, uint _checkOutTimestamp) payable public
        {
2
3    //check structureID is valid and structure ad is active
4    require(_structureID < structures.length);
5    require(structures[_structureID].active);
6
7    //check timestamps are valid:
8    //- check that check-in is not in the past
9    dateUtils.DateTime memory currentTime = dateUtils.
      parseTimestamp(block.timestamp);
10   uint todayTimestamp = dateUtils.toTimestamp(currentTime.
      year, currentTime.month, currentTime.day);
11   require(_checkInTimestamp >= todayTimestamp);
12
13   //- check that check-in is set at 14 and check-out at 10
14   dateUtils.DateTime memory checkInTime = dateUtils.
      parseTimestamp(_checkInTimestamp);
15   dateUtils.DateTime memory checkOutTime = dateUtils.
      parseTimestamp(_checkOutTimestamp);
16   require(checkInTime.hour == 14);
17   require(checkOutTime.hour == 10);
18
19   //- check that reservation is at least for one night
20   uint checkInDayTimestamp = dateUtils.toTimestamp(
      checkInTime.year, checkInTime.month, checkInTime.day);
21   require(_checkOutTimestamp > checkInDayTimestamp + 1 days);
22
23   //check structure is available in that period
24   require(checkStructureAvailability(_structureID,
      _checkInTimestamp, _checkOutTimestamp));
25
26   //check that paid amount is correct for number of nights
27   uint checkOutDayTimestamp = dateUtils.toTimestamp(
```

```
        checkOutTime.year, checkOutTime.month, checkOutTime.day);
28    uint nights = (checkOutDayTimestamp - checkInDayTimestamp)
        / 60 / 60 / 24;
29    require(msg.value == structures[_structureID].dailyPrice *
        nights);
30    //End of checks
31
32    Reservation memory newRes;
33    newRes.reservationID = reservations.length;
34    newRes.structureID = _structureID;
35    newRes.guest = msg.sender;
36    newRes.checkInTimestamp = _checkInTimestamp;
37    newRes.checkOutTimestamp = _checkOutTimestamp;
38    newRes.price = msg.value;
39    newRes.balance = msg.value;
40    newRes.canceled = false;
41
42    reservations.push(newRes);
43    structures[_structureID].reservationIDs.push(newRes.
        reservationID);
44    myReservations[msg.sender].push(newRes.reservationID);
45  }
```

Il primo aspetto che si può notare, analizzando l'implementazione della funzione `addReservation` esposta dallo smart contract, è l'utilizzo della keyword payable nella signature del metodo, un modificatore necessario per abilitare il contratto alla ricezione di Ether.

Da notare inoltre il modo in cui è stato strutturato il codice sorgente, ovvero con tutti i controlli per la validazione dei dati passati in ingresso eseguiti all'inizio, e gli aggiornamenti delle strutture dati svolti tutti in coda al metodo. Tale pattern, seguito anche nell'implementazione di tutte le altre funzioni, rende più efficiente l'eventuale rollback delle modifiche eseguite fino a quel momento, qualora si verificasse un errore o l'invocazione non dovesse superare con successo tutti i controlli iniziali. Per la verifica dei requisiti viene infatti invocata la funzione `require` che prende in input un'espressione booleana, la quale nel caso risulti essere falsa provoca la sospensione dell'esecuzione del metodo e l'annullamento di tutte le modifiche apportate.

Infine, come previsto, il contratto non invia direttamente il pagamento ricevuto al proprietario dell'appartamento, ma lo custodisce all'interno del suo wallet. Il proprietario potrà richiederne il prelievo non appena avrà soddisfatto tutti i requisiti necessari, invocando il metodo `withdrawReservation` (si ricorda che non è possibile effettuare il prelievo della cifra fintantoché mancano più di 72 ore alla data di check-in).

```
function withdrawReservation(uint _reservationID) public {
  require(_reservationID < reservations.length && !
   reservations[_reservationID].canceled);
  Reservation storage res = reservations[_reservationID];
  Structure memory house = structures[res.structureID];
  require(house.owner == msg.sender);
  require(block.timestamp > (res.checkInTimestamp - 3 days));

  uint amount = res.balance;
  res.balance = 0;
  msg.sender.transfer(amount);
}
```

Questo metodo è stato implementato seguendo le indicazioni del pattern "Withdrawal from Contracts" [18], volto a prevenire la possibilità di re-entrancy attacks.

4.4 Interazione con la serratura smart

Si analizza ora come è stata implementata l'interazione tra l'applicazione web e la serratura smart, simulata utilizzando un Raspberry Pi 3, facendo uso del protocollo Whisper per l'invio dei messaggi contenenti i comandi per il controllo della serratura stessa.

Come già spiegato in sezione 1.4, Whisper per poter inviare un messaggio richiede che questo venga prima crittografato utilizzando una chiave simmetrica condivisa o la chiave pubblica del destinatario. In questo caso si è scelto di utilizzare la crittografia simmetrica, in quanto diversamente sarebbe stato necessario prevedere anche un meccanismo per lo scambio delle chiavi pubbliche tra l'utente e la serratura. Quindi sia l'applicazione web che il software Node.js eseguito dal Raspberry, non appena vengono caricati, generano una chiave crittografica per Whisper a partire da una password condivisa, ovvero "ethertravel".

```
web3.shh.generateSymKeyFromPassword("ethertravel", function(
   error, keyId) {
  symKeyId = keyId;
});
```

Fatto ciò l'applicazione web è già in grado di inviare messaggi, e per farlo è sufficiente che invochi la seguente funzione `sendWhisper`.

```
function sendWhisper(structureID, message) {
  web3.personal.sign(encodeToHex(JSON.stringify(message)),
   App.userAccount, function(error, signature) {
    if(!error) {
      let msg = {
```

```
        content: encodeToHex(JSON.stringify(message)),
        signature: signature
      }
      let postData = {
        ttl: 7,
        topic: getTopic(structureID),
        powTarget: 2.01,
        powTime: 100,
        payload: encodeToHex(JSON.stringify(msg))
      };

      postData.symKeyID = symKeyId;
      web3.shh.post(postData, function() {
        console.log("Message sent");
      });
    } else {
      console.error(error);
    }
  });
}
```

Passato un messaggio in ingresso, questo viene fatto firmare digitalmente all'utente utilizzando la chiave privata del suo account Ethereum e la firma risultante viene allegata al messaggio originale. Il tutto viene poi inserito all'interno di un envelope che aggiunge una serie di attributi tra cui il topic, ovvero l'oggetto del messaggio, che sarà fondamentale in fase di ricezione. Come oggetto si scelto di impostare l'identificatore della struttura a cui si sta tentando di accedere, in questo modo ogni serratura smart considererà soltanto i messaggi effettivamente riguardanti la struttura di appartenenza.

Inizialmente la serratura smart dovrà essere necessariamente configurata: a tale scopo è stato previsto un file chiamato `settings.json`, contenente tutti i parametri da configurare affinché l'applicazione sappia a che struttura appartiene la serratura e possa così mettersi in ascolto dei relativi messaggi in arrivo.

```
{
  "url": "http://192.168.1.200:8545",
  "contractAddress": "0
   x34CaEad1632255d05FDC492055879cF29713229a",
  "structureID": 0
}
```

I tre parametri di configurazione da impostare sono:

- `url`: l'indirizzo di un'istanza Geth con API Whisper abilitate;
- `contractAddress`: l'indirizzo Ethereum dello smart contract;

- `structureID`: l'identificatore univoco della struttura a cui la serratura si riferisce.

Completata la configurazione iniziale, il software del dispositivo IOT è pronto per restare in ascolto di messaggi Whisper in arrivo, aventi come topic lo `structureID` specificato.

```
web3js.shh.generateSymKeyFromPassword("ethertravel",
  function(error, keyId) {

  let filter = {
    topics: [getTopic(structureID)],
    symKeyID: keyId
  };
  startListen(filter);
});

async function startListen(filter) {

  var msgFilter = await web3js.shh.newMessageFilter(filter);

  msgFilter.watch(function(error, result) {
    if (!error) {
      let message = decodeFromHex(result.payload);
      let guestAddress=contract.getCurrentGuest(structureID);
      let res = contract.getStructureDetails(structureID);
      let ownerAddress = res[0];
      web3js.personal.ecRecover(message.content,
        message.signature, function(er, address) {
        var allowed = (address == guestAddress
            || address == ownerAddress);
        var command = decodeFromHex(message.content);
        if(allowed && command == "open") {
          LED.writeSync(1);
        } else if(allowed && command == "close") {
          LED.writeSync(0);
        }
      });
    }
  });
}
```

Ricevuto un messaggio, il software interroga lo smart contract richiamando le funzioni `getCurrentGuest` e `getStructureDetails`, per ottenere gli indirizzi pubblici di coloro che hanno i permessi necessari per poter impartire comandi alla serratura, ovvero gli indirizzi del proprietario dell'appartamento e dell'eventuale cliente che vi sta soggiornando in quel preciso momento. Dopodiché, richiamando la funzione `ecRecover`, ricalcola a partire dalla firma allegata al messaggio, l'indirizzo Ethereum pubblico del mittente, che

verrà confrontato con quelli restituiti dalla blockchain. Soltanto se il mittente del messaggio corrisponderà a una delle due entità autorizzate, la serratura verrà aperta o chiusa secondo quanto richiesto nel messaggio. L'apertura e la chiusura viene simulata attraverso l'accensione e lo spegnimento di un led collegato al Raspberry Pi.

Capitolo 5

Valutazione

In questo capitolo si fornirà una valutazione del lavoro svolto andando a verificare se sono stati raggiunti tutti gli obiettivi che questa tesi si prefissava. Nella prima parte si valuterà l'applicazione realizzata e in particolare i limiti della tecnologia blockchain che sono emersi durante lo sviluppo del progetto. Nella seconda parte si discuterà invece come è stata svolta la validazione dell'applicazione.

5.1 Valutazione dell'applicazione

Gli obiettivi di questa tesi sono stati raggiunti con successo, l'applicazione web decentralizzata realizzata risulta infatti soddisfare tutti i requisiti che erano stati definiti durante la fase iniziale di analisi. La sua progettazione e realizzazione hanno richiesto lo studio di diverse tecnologie innovative e un'analisi per capire come integrarle tra loro per riuscire a concretizzare il risultato atteso.

La piattaforma web per l'affitto di case vacanza implementata può ritenersi quindi soddisfacente, in quanto realizza tutte le funzionalità previste in modo totalmente trasparente per l'utente finale rispetto a un sito web tradizionale, se non fosse che per il suo funzionamento è richiesta l'installazione dell'estensione Metamask o l'utilizzo di un browser specifico come Mist per poter gestire il wallet Ethereum e inviare le transazioni alla blockchain.

Questo progetto ha permesso inoltre di evidenziare i limiti che, ad oggi, le tecnologie utilizzate per la realizzazione di applicazioni decentralizzate come questa, ancora presentano. Tra i limiti più importanti si evidenziano:

- **scalabilità limitata**: tra i più grandi vantaggi che si ottengono sviluppando un'applicazione decentralizzata vi è la mancata necessità di dover costruire, configurare e manutenere un'infrastruttura sulla quale

rilasciare ed eseguire l'applicazione stessa, in quanto si sfrutta l'infrastruttura della rete preesistente scelta. Questa caratteristica può però risultare anche un enorme limite, infatti il livello massimo di scalabilità dell'applicazione sarà determinato dalla capacità massima della blockchain scelta. In particolare, Ethereum in questo momento è in grado di gestire un throughput di circa 15 transazioni al secondo. Se si considera che sulla sola piattaforma Booking vengono ogni giorno fatte oltre 1.500.000 prenotazioni [1], è semplice verificare che una singola applicazione con questo genere di volumi sarebbe già in grado di saturare e congestionare l'intera rete. La blockchain di Ethereum non è quindi ancora pronta ad ospitare applicazioni reali con un consistente numero di utenti senza che non si verifichino gravi stalli della rete, come già successo nel dicembre 2017 con il rilascio dell'applicazione CryptoKitties [19] che nel giro di pochi giorni diventò virale. A tal proposito la roadmap di Ethereum prevede già aggiornamenti per aumentare in futuro il numero di transazioni al secondo che potrà essere in grado di gestire. La scalabilità delle blockchain resta comunque uno dei temi più dibattuti e su cui si stanno concentrando la maggior parte degli sforzi della ricerca;

- **impossibilità di aggiornare facilmente gli smart contract**: dal momento che il codice sorgente degli smart contract è memorizzato all'interno della blockchain, questo diventa immutabile e permanente, rendendo di fatto impossibile aggiornare direttamente il contenuto di uno smart contract già pubblicato. Questo non impedisce in maniera assoluta l'aggiornamento di un'applicazione decentralizzata, ma lo complica notevolmente, costringendo gli sviluppatori a ricorrere a diversi stratagemmi più articolati che comportano la creazione di un nuovo contratto che dovrà interagire in qualche modo con il precedente, divenendo di fatto una specie di strato di secondo livello;

- **impossibilità di comunicare con l'esterno**: uno smart contract non può interagire con servizi offerti dal mondo esterno come email o notifiche sms. Se si desidera realizzare un'applicazione decentralizzata in grado di fornire anche questo tipo di funzionalità allora è necessario prevedere l'utilizzo di un server centrale che realizzi questi servizi in risposta ad eventi scatenati dallo smart contract per i quali rimane costantemente in ascolto. Inoltre, è da notare che uno smart contract può fornire risposte o scatenare eventi soltanto a seguito di richieste effettuate da un client, in quanto la sua esecuzione non può essere

[1] https://www.booking.com/content/about.en-gb.html

programmata per eseguire compiti periodici o avviare per primo una comunicazione;

- **impossibilità di memorizzare dati sensibili o confidenziali**: la scelta di utilizzare una blockchain pubblica come Ethereum consente da una parte di poter offrire all'utente una completa trasparenza sull'intero funzionamento dell'applicazione, dall'altra impedisce però di poter raccogliere dati sensibili o confidenziali in quanto memorizzandoli nella blockchain verrebbero esposti pubblicamente. Una possibile soluzione potrebbe essere quella di ricorrere alla crittografia, cifrando i dati prima di inviarli alla blockchain. Questo non impedirebbe però ad un eventuale attaccante di provare a violare la crittografia cercando di individuare la chiave utilizzata attraverso un attacco brute-force, che per quanto lento e dispendioso possa essere in base all'algoritmo scelto, può comunque non risultare un deterrente sufficiente in base al valore delle informazioni che l'attaccante sta cercando di carpire;

- **tempi di attesa**: la validazione delle transazioni, realizzata attraverso il processo di mining, richiede sempre un certo periodo di tempo, dipendente dal livello di congestione della rete in un dato momento e dal *gas price* configurato dall'utente. Dal momento che la blockchain di Ethereum genera un nuovo blocco mediamente ogni 14 secondi, l'utente dovrà attendere circa 10-20 secondi per vedere la propria transazione approvata (sia essa un trasferimento di Ether o l'invocazione di un metodo di uno smart contract che ne provoca un aggiornamento dello stato). Oggigiorno gli utenti hanno aspettative molto elevate quando navigano sul web e si aspettano che tutte le pagine e tutti i servizi vengano sempre erogati in modo praticamente istantaneo. L'attesa necessaria per il mining presente nelle applicazioni web decentralizzate può quindi compromettere parzialmente l'esperienza utente.

Risulta quindi chiaro che la blockchain non può essere impiegata in qualunque contesto e che presenta, ad oggi, ancora diversi problemi dovuti alla scarsa maturità della tecnologia.

5.2 Validazione

Quando si realizza un'applicazione decentralizzata basata su blockchain, la fase di validazione ricopre un ruolo molto importante all'interno del ciclo di sviluppo, in quanto una volta pubblicato sulla blockchain il codice sorgente di uno smart contract risulterà essere per sempre immutabile. E' quindi

necessario accertarsi in maniera rigorosa della correttezza dei sorgenti, dal momento che eventuali bug saranno molto difficili da correggere. Svolgere una validazione completa e rigorosa nel caso di applicazioni non banali può però risultare un compito molto arduo e costoso in termini di tempo.

Per quanto riguarda l'applicazione oggetto di questa tesi è stata svolta una validazione parziale, mediante l'implementazione di test automatici relativi alla parte di gestione degli annunci delle strutture ospitanti. Il testing automatico delle funzioni relative alla gestione delle prenotazioni è stato lasciato a sviluppi futuri in quanto subentrano anche aspetti relativi alla gestione del tempo che complicano in maniera significativa la costruzione di test esaustivi. I test automatici sono stati implementati utilizzando il linguaggio di programmazione Javascript insieme al framework di test Mocha, e Chai per quanto riguarda le asserzioni, entrambi integrati all'interno della suite di sviluppo Truffle [2] che è stata utilizzata per la realizzazione dell'intero progetto.

Infine, prima di pubblicare sulla mainnet uno smart contract che prevede di mantenere all'interno del proprio wallet una quantità di Ether potenzialmente cospicua, come nel caso di questa applicazione, sarebbe buona norma far condurre a terze parti un audit sulla sicurezza del codice implementato.

[2]https://truffleframework.com

Conclusioni e lavori futuri

La realizzazione di questa tesi ha consentito l'approfondimento di diverse tecnologie fortemente innovative per lo sviluppo di applicazioni totalmente decentralizzate, tra cui in particolare la tecnologia blockchain che negli ultimi anni sta riscuotendo un sempre maggior interesse nell'ambito della ricerca.

Il caso di studio scelto è risultato essere sufficientemente complesso, tanto da richiedere l'utilizzo di diverse tecnologie e l'individuazione di un'architettura per farle interagire tra loro e realizzare il sistema richiesto. Oltre allo sviluppo di un'applicazione web decentralizzata per l'affitto di case vacanza, la tesi si è concentrata anche su come far comunicare il mondo blockchain con il mondo Internet of Things: a tal fine è stata sviluppata una semplice applicazione per Raspberry Pi che simula il funzionamento di una serratura smart che interagisce con il sistema precedente, mostrando come queste due tecnologie possono interagire tra loro per offrire servizi tangibili nel mondo reale.

Il risultato finale del lavoro svolto si è rivelato valido raggiungendo tutti gli obiettivi prefissati da questa tesi. In particolare, la realizzazione del progetto ha permesso di comprendere meglio anche quali sono i limiti attuali e le problematiche che si possono dover affrontare quando si lavora con tecnologie nuove e quindi non ancora del tutto consolidate. Pur svolgendo quanto richiesto, l'applicazione realizzata non può comunque ritenersi conclusa e sono diversi i miglioramenti e le funzioni che potrebbero essere introdotti in lavori futuri. Tra questi si evidenziano:

- predisporre lo smart contract per semplificare aggiornamenti futuri;
- aggiungere la possibilità di far recensire ai clienti le strutture presso le quali hanno soggiornato;
- riscrivere l'applicazione web come app per smartphone iOS e Android;
- rendere l'architettura ibrida per riuscire a implementare servizi di e-mailing e notifiche push;

- confrontare questo sistema con un'implementazione che adotta tecnologie diverse, come ad esempio una blockchain permissioned.

In conclusione, il risultato ottenuto da questa tesi è stato molto soddisfacente e di grande interesse, oltre a rappresentare una base di partenza per numerosi sviluppi futuri che potrebbero portare all'approfondimento di altrettante numerose nuove tecnologie complementari o alternative.

Bibliografia

[1] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, pp. 99–111, 1991.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf, 2008, [Online; ultimo accesso 8 marzo 2019].

[3] W. Hall-Smith, "What is blockchain technology?" https://www.ig.com/uk/trading-strategies/what-is-blockchain-technology-180312, 2018, [Online; ultimo accesso 8 marzo 2019].

[4] F. A. Saleh, "Blockchain without waste : Proof-of-stake*," https://www.ivey.uwo.ca/cmsmedia/3783185/11-30-18-saleh.pdf, 2018, [Online; ultimo accesso 8 marzo 2019].

[5] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *Open Problems in Network Security.* Springer International Publishing, 2016, pp. 112–125.

[6] "A next-generation smart contract and decentralized application platform," https://github.com/ethereum/wiki/wiki/White-Paper, [Online; ultimo accesso 8 marzo 2019].

[7] D. Karatkevich, "Hyperledger enterprise solutions: Top 5 real use cases," https://openledger.info/insights/hyperledger-enterprise-solutions-top-5-real-use-cases/, 2018, [Online; ultimo accesso 8 marzo 2019].

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys*

Conference, ser. EuroSys '18, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[9] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 357–388.

[10] D. Langley, "Ethereum 2.0," https://medium.com/rocket-pool/ethereum-2-0-76d0c8a76605, 2018, [Online; ultimo accesso 8 marzo 2019].

[11] D. Leung, "Decentralized application messaging with whisper — part 1," https://blog.enuma.io/update/2018/08/08/decentralized-application-messaging-with-whisper.html, 2018, [Online; ultimo accesso 8 marzo 2019].

[12] T. Gerring, "Building the decentralized web 3.0," https://blog.ethereum.org/2014/08/18/building-decentralized-web/, 2014, [Online; ultimo accesso 8 marzo 2019].

[13] J. Talbot, "What are bloom filters?" https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff, 2015, [Online; ultimo accesso 8 marzo 2019].

[14] J. Benet, "Ipfs - content addressed, versioned, p2p file system (draft 3)," https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf, 2014, [Online; ultimo accesso 8 marzo 2019].

[15] P. Vowell, "What is the interplanetary file system?" https://www.maxcdn.com/one/visual-glossary/interplanetary-file-system/, 2016, [Online; ultimo accesso 8 marzo 2019].

[16] X. Decuyper, "Ipfs - simply explained," https://www.savjee.be/videos/simply-explained/ipfs-interplanetary-filesystem/, 2018, [Online; ultimo accesso 8 marzo 2019].

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[18] "Common patterns for solidity," https://solidity.readthedocs.io/en/latest/common-patterns.html, [Online; ultimo accesso 8 marzo 2019].

[19] T. O'Ham, "Cryptokitties is totally wrecking the ethereum network," https://bitsonline.com/cryptokitties-wrecking-ethereum/, 2017, [Online; ultimo accesso 8 marzo 2019].