Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Sviluppo di un'app client per un sistema assicurativo basato su Blockchain e monitoraggio IoT

Relatore:
Prof.
FEDERICO MONTORI

Correlatore:
Dott.
LORENZO GIGLI

Presentata da:
FEDERICO TURRINI

Sessione II
Anno Accademico 2024/2025

Sommario

Negli ultimi anni si è osservato un significativo aumento dell'utilizzo dei dati provenienti da dispositivi IoT, accompagnato da crescenti problematiche legate alla sicurezza, all'affidabilità e alla trasparenza di tali dati. Per affrontare queste criticità, sono state proposte soluzioni basate sulla tecnologia Blockchain e sugli oracoli, come nel caso della piattaforma ZONIA. Tuttavia, ZONIA presenta limitazioni dovute all'assenza di un'applicazione client dedicata. In questo lavoro di tesi è stata progettata e sviluppata un'applicazione client che, sfruttando i dati IoT distribuiti e garantiti dagli oracoli di Zonia, consente la creazione di *smart insurance* affidabili e trasparenti. Inoltre, è stata condotta un'analisi dei costi e dei tempi dei flussi di lavoro dell'applicazione, al fine di fornire una valutazione completa anche sotto il profilo economico. I risultati evidenziano che l'approccio multi-chain adottato permette di bilanciare sicurezza, efficienza e costi operativi, rendendo l'applicazione sostenibile e pratica per l'uso reale.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Negli ultimi anni, l'uso dei dati provenienti da dispositivi IoT (Internet of Things) ha registrato una crescita esponenziale, rivoluzionando numerosi settori come l'industria, l'agricoltura e i servizi finanziari. Tuttavia, l'aumento del volume e della complessità dei dati ha portato con sé nuove sfide, in particolare riguardanti la sicurezza, l'affidabilità e la trasparenza delle informazioni raccolte e condivise. La mancanza di garanzie sulla qualità dei dati può compromettere l'efficacia di applicazioni critiche, come le assicurazioni intelligenti (smart insurance), che si basano su dati IoT affidabili per determinare premi, gestire sinistri e automatizzare processi contrattuali.
Per affrontare tali problematiche, sono state proposte soluzioni basate sulla combinazione di Blockchain e oracoli, strumenti che consentono di garantire l'integrità e la disponibilità dei dati distribuiti. In particolare, la piattaforma Zonia fornisce un meccanismo per distribuire dati IoT sicuri tramite oracoli Blockchain. Tuttavia, Zonia presenta ancora alcune limitazioni, tra cui l'assenza di un'applicazione client che permetta di interagire direttamente con i dati in modo user-friendly e orientato alle applicazioni finali.
In questo lavoro di tesi è stata progettata e sviluppata un'applicazione client che sfrutta i dati IoT distribuiti e garantiti dagli oracoli di Zonia, consentendo la creazione di assicurazioni intelligenti affidabili e trasparenti. L'applicazione offre un'interfaccia pratica per utenti e operatori del settore assicurativo. Inoltre, è stata condotta un'analisi dei costi e dei tempi dei flussi di lavoro dell'applicazione, al fine di fornire una valutazione completa anche dal punto di vista economico.

La tesi è strutturata come segue:

- **Capitolo 1: Stato dell'arte** – presenta una panoramica delle tecnologie IoT, Blockchain e oracoli, evidenziando le principali sfide e soluzioni attuali.
- **Capitolo 2: ZONIA** – descrive l'architettura e le funzionalità di Zonia, con particolare attenzione al funzionamento degli oracoli per la distribuzione sicura dei dati IoT.
- **Capitolo 3: Progettazione e Implementazione** – illustra la progettazione e lo sviluppo dell'applicazione client, con dettagli sulle scelte architetturali e sulle tecnologie utilizzate.
- **Capitolo 4: Analisi e Test** – fornisce una valutazione dei costi, dei tempi e delle prestazioni dei flussi di lavoro dell'applicazione, verificandone l'efficacia e l'efficienza.

Capitolo 1

Stato dell'Arte

1.1 Background

L'Internet of Things (IoT) si riferisce a una rete di dispositivi, veicoli, apparecchi e altri oggetti fisici incorporati con sensori, software e connettività di rete che consentono di raccogliere e condividere dati.
L'IoT consente ai dispositivi intelligenti di comunicare tra di loro e con altri dispositivi abilitati a Internet, come smartphone e gateway, creando una vasta rete di dispositivi interconnessi in grado di scambiare dati ed eseguire diverse attività in modo autonomo.
Alcuni esempi di modi d'uso possono essere:

- Monitoraggio delle condizioni ambientali nelle aziende agricole.
- Gestione dei modelli di traffico con smart car e altri dispositivi automobilistici intelligenti.
- Controllo di macchinari e processi nelle fabbriche.
- Monitoraggio dell'inventario e delle spedizioni nei magazzini.[1]

L'impatto dell'IoT si sta facendo sentire sempre di più negli ultimi anni, ma questo ha portato al manifestarsi di diverse problematiche, come la sicurezza,

la trasparenza e l'affidabilità dei dati, dovute all'alto grado di centralizzazione e opacità di questi sistemi, che li compromettono [2].
Sempre più aziende stanno iniziando a sfruttare la decentralizzazione e trasparenza fornita dalle Blockchain per ovviare ai problemi di sicurezza dell'IoT. Questa tecnologia è stata sviluppata per la prima volta da Satoshi Nakamoto. Il problema che Nakamoto andò a risolvere fu quello di stabilire fiducia in un sistema distribuito[3]. La Blockchain è un registro digitale distribuito che può immagazzinare dati di ogni tipo. Il nome non è casuale: questo registro viene infatti spesso descritto come una "catena" fatta di singoli "blocchi" di dati[4]. Quando dei dati, tramite una transizione, richiedono di entrare nella Blockchain, viene creato un nuovo blocco che, una volta validato, viene aggiunto alla stessa.
Uno smart contract è un programma che viene utilizzato per costruire applicazioni nella Blockchain. Nella tabella 1.1 vengono mostrati i diversi blocchi concettuali che costituiscono uno smart contract nella Blockchain Ethereum.

Campo	**Descrizione**
Balance	Quanti Ether possiede il contratto
Storage	I dati in memoria al contratto
Code	Codice macchina del contratto

Tabella 1.1: Struttura smart contract

Questo permette, ad esempio, di finanziare un pagamento verso un beneficiario istantaneamente e automaticamente una volta che le condizioni definite nel contratto sono soddisfatte.
Ci sono infatti diverse applicazioni e progetti che cercano di sfruttare la Blockchain e gli smart contracts per dati IoT, tra cui:

- Io Coltivo Italia (Farzati Tech)
  - Usano la Blockchain per tracciabilità del prodotto e sensoristica con tecnologia di BluDev (software proprietario)[5].
- CO2 Footprint Authentication Module (COFAM)
  - Mira a integrare i registri Blockchain con sensori IoT, machine learning e modelli semantici, creando un sistema affidabile per la notarizzazione e la tokenizzazione dei dati[6].
- M-Sec (UE/Japan, Smart Cities)
  - Progetto in collaborazione tra Unione Europea e Giappone che si occupa di ricercare e sviluppare tecnologie di sicurezza Multi-layered per creare *smart cities* iperconnesse con Blockchain, Big Data, Cloud e IoT [7].
- ABATA, Blockchain Applications for Authenticity and Food Traceability
  - Progetto il cui obbiettivo e sviluppare un sistema di informazioni basato su Blockchain ad accesso pubblico, capace di tracciare la catena produttiva e tutte le informazioni riguardanti le origini di prodotti alimentari[8].

1.2 Integrazione tra IoT e Blockchain

Le architetture centralizzate, come quelle utilizzate nel cloud computing, hanno contribuito in modo significativo allo sviluppo dell'Internet of Things (IoT). Tuttavia, per quanto riguarda la trasparenza dei dati, esse agiscono come delle "scatole nere", e i partecipanti alla rete non hanno una visione chiara di dove e come verranno utilizzate le informazioni che forniscono.
La Blockchain può arricchire l'IoT offrendo un servizio di condivisione affidabile, in cui le informazioni sono attendibili e tracciabili.

Le fonti dei dati possono essere identificate in qualsiasi momento e i dati rimangono immutabili nel tempo, aumentando così la loro sicurezza. Nei casi in cui le informazioni dell'IoT debbano essere condivise in modo sicuro tra molti partecipanti, questa integrazione rappresenterebbe una rivoluzione chiave [9].

1.2.1 Sfide dell'integrazione IoT–Blockchain

L'integrazione della tecnologia Blockchain con l'Internet of Things non è un processo banale.
La Blockchain è stata originariamente concepita per un contesto Internet composto da computer potenti e stabili, molto diverso dalla realtà dell'IoT, dove i dispositivi sono spesso limitati in termini di risorse computazionali, memoria e capacità energetica. Le transazioni su Blockchain richiedono firme digitali e protocolli crittografici complessi; di conseguenza, i dispositivi IoT che desiderano interagire con la catena devono essere dotati di tali funzionalità, rendendo l'integrazione difficile.
Inoltre, l'aumento del numero di attacchi alle reti IoT e le loro gravi conseguenze evidenziano la necessità di renderle più sicure e resilienti.
Molti esperti individuano nella Blockchain una tecnologia chiave per migliorare la sicurezza dell'IoT, grazie alla sua natura decentralizzata, all'immutabilità dei dati e alla trasparenza delle operazioni.
Tuttavia, una delle principali sfide di questa integrazione riguarda l'affidabilità dei dati generati dai dispositivi IoT.
La Blockchain, infatti, garantisce che le informazioni registrate siano immutabili e tracciabili, ma non può correggere o validare dati che arrivano già corrotti o falsificati. In altre parole, se l'origine del dato è compromessa, rimane tale anche dopo la scrittura sul blocco.
Dati corrotti possono derivare non solo da attacchi malevoli, ma anche da errori di sensori, malfunzionamenti hardware o problemi di comunicazione.
Un'ulteriore difficoltà è rappresentata dalla gestione della privacy dei dati. Nel contesto IoT, la tutela della privacy deve essere garantita fin dalla fase

di raccolta dei dati e mantenuta lungo tutto il ciclo di vita dell'informazione, includendo i livelli di comunicazione e applicazione.
Proteggere i dispositivi in modo che i dati vengano memorizzati in sicurezza e non siano accessibili a utenti non autorizzati richiede l'integrazione di software crittografici e meccanismi di autenticazione direttamente a livello di dispositivo, un compito complesso, soprattutto in presenza di dispositivi a basso costo o con capacità hardware limitate.
In sintesi, l'integrazione IoT–Blockchain presenta vantaggi potenziali significativi, ma è ostacolata da problematiche tecniche, computazionali e di sicurezza che richiedono soluzioni architetturali dedicate.[9]

1.2.2 Oracoli e affidabilità dei dati

Gli Oracoli sono servizi di terze parti che forniscono agli smart contract informazioni provenienti dall'esterno della rete. Essi fungono da ponte tra la Blockchain e il mondo esterno. Le Blockchain e gli smart contract, infatti, non hanno la capacità di accedere direttamente a dati *off-chain* (ovvero dati che risiedono al di fuori della rete). Tuttavia, per molte tipologie di accordi contrattuali è essenziale poter disporre di informazioni provenienti dal mondo reale per consentire l'esecuzione corretta del contratto, come accade, ad esempio, nei sistemi IoT, in cui i sensori generano dati esterni che devono essere integrati nelle logiche della Blockchain.

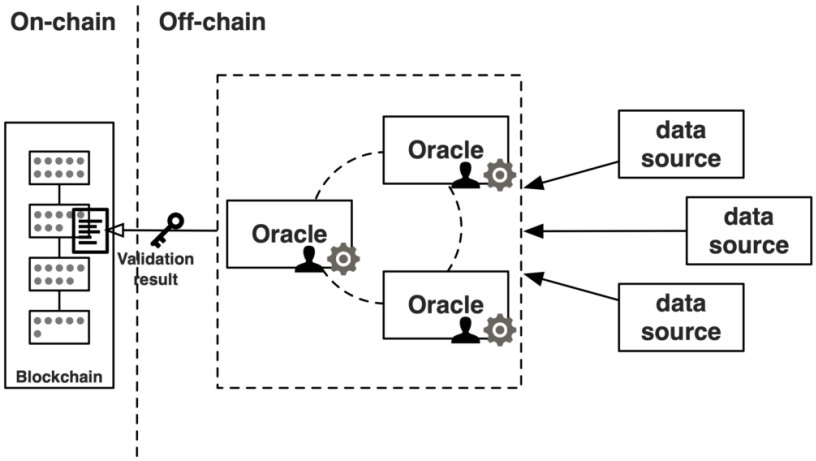


Figura 1.1: Pattern architetturale per oracoli decentralizzati

È proprio in questo contesto che entrano in gioco gli oracoli, i quali forniscono un collegamento affidabile tra dati *off-chain* e *on-chain*. Gli oracoli rivestono un ruolo fondamentale all'interno dell'ecosistema Blockchain poiché ampliano il campo di applicazione degli smart contract: senza di essi, gli smart contract avrebbero un utilizzo molto limitato, potendo accedere solo ai dati interni alla rete.

È importante sottolineare che un oracolo Blockchain non rappresenta la fonte del dato in sé, ma piuttosto il livello intermedio che interroga, verifica e autentica le fonti di dati esterne, per poi trasmettere le informazioni

validate alla Blockchain. Tuttavia, gli oracoli introducono anche un rischio significativo: se vengono considerati entità affidabili ma subiscono una compromissione, possono compromettere a loro volta l'esecuzione degli smart contract a cui forniscono dati.

Pertanto, nella progettazione di sistemi che fanno uso di oracoli, è fondamentale considerare attentamente il modello di fiducia adottato. Presupporre che un oracolo sia completamente affidabile potrebbe infatti indebolire la sicurezza dell'intero smart contract, esponendolo alla possibilità di ricevere dati falsi o manipolati. [10]

1.2.3 Introduzione a ZONIA

Si va ora a introdurre brevemente ZONIA: *A Zero-Trust Oracle System for Blockchain IoT Applications*. Un nuovo sistema di oracoli Blockchain progettato per migliorare l'integrità dei dati e la decentralizzazione negli ambienti IoT e quindi risolvere i problemi precedentemente descritti.
A differenza degli approcci tradizionali, che si basano su ambienti di esecuzione fidati (TEE) e su fonti di dati centralizzate, ZONIA adotta un modello decentralizzato che consente la partecipazione anonima e integra molteplici fonti di dati per garantire equità e affidabilità[2].
ZONIA permette di utilizzare un insieme di interfacce per richiedere dati IoT sicuri e trasparenti, che verranno utilizzati in questo contesto applicativo per produrre assicurazioni smart con dati reali. Si possono richiedere questi dati a ZONIA tramite delle query geospaziali che permettono, ad esempio, di avere informazioni su dati come umidità o temperatura.
L'applicazione qui presentata va a interagire con la declinazione multichain di ZONIA: ZONIA-MC. Questa non ha una *sidechain* predeterminata per eseguire il flusso della richiesta (Blockchain target), invece esegue un algoritmo che permette, una volta arrivata la query, di determinare quale sia la Blockchain più adatta a processarla.

1.3 Motivazioni del lavoro

Nonostante il fatto che l'architettura proposta da ZONIA introduca un approccio innovativo e decentralizzato per la gestione degli oracoli in ambito IoT, il progetto soffre attualmente della mancanza di un client dedicato che ne consenta un utilizzo diretto e intuitivo da parte degli utenti. Tale assenza limita la diffusione e l'effettiva adozione del sistema, ostacolando la possibilità di interagire in maniera semplice e trasparente con i contratti intelligenti.

L'applicazione sviluppata in questa tesi mira a colmare questa lacuna, fornendo un client user-friendly che consenta l'accesso alle funzionalità di ZONIA senza richiedere conoscenze tecniche avanzate. Inoltre, il sistema proposto introduce un meccanismo di selezione indiretta della Blockchain target, che rappresenta un elemento strategico per garantire flessibilità, efficienza e adattabilità del sistema ZONIA. Le diverse Blockchain, come vedermo, presentano caratteristiche eterogenee in termini di costi di transazione, tempi di finalizzazione, livello di sicurezza e grado di decentralizzazione. Consentire all'utente di orientare la scelta in base a questi criteri permette di ottimizzare le prestazioni complessive e adattare il sistema al contesto applicativo specifico.

In questo modo, il lavoro non solo rende più agevole l'interazione con ZONIA, ma contribuisce anche ad aumentarne l'accessibilità, favorendo la sperimentazione e la crescita dell'ecosistema.

Capitolo 2

ZONIA

ZONIA è un oracle network decentralizzato "zero-trust", dove gli individui sono liberi di entrare nella rete senza bisogno di rivelare la propria identità o usare hardware specializzato. Lo scopo di ZONIA è recuperare dati IoT affidabili e renderli disponibili on-chain per smart contract e applicazioni distribuite.
Gli obiettivi principali sono:

- Eliminare punti unici di fiducia, sfruttando la decentralizzazione.
- Supportare query semantiche e geospaziali.
- Combinare dati da molteplici sorgenti (*Producers*) tramite *Indexer / Oracle* e valutare o aggiornare la reputazione dei nodi per contrastare manipolazioni o collusioni.

I seguenti sono gli attori nell'architettura di ZONIA:

- *Consumers*
  - Sono coloro che inviano le richieste, fornendo la query e pagando, possono essere smart contracts, applicazioni o servizi.

- *Producers*
  - Sono la fonte primaria di dati che vengono processati dal sistema. Possono essere sensori, dispositivi o anche utenti reali. I produttori di questi dati vengono retribuiti per le loro contribuzioni, incentivando la qualità dei dati forniti.
- *Indexers*
  - Tengono traccia delle query inviate dai *Consumers*, per poi decidere quali *Producers* utilizzare per ottenere i dati.
- *Oracles*
  - Collezionano i dati dai *Producers*, li valutano e garantiscono l'integrità dei dati. Gli Oracoli vengono ricompensati per il loro contributo e i dati convalidati vengono scritti in modo sicuro on-chain[2].

2.1 Architettura di ZONIA

Tutta la logica di core è mantenuta da una Blockchain EVM-compatibile detta *RelayChain*, che si occupa di ospitare gli *smart contracts* principali, questi sono:

- *EntityRegistry*
  - Ha in memoria tutti gli Oracoli e gli Indici registrati nel sistema e si occupa anche del processo di *staking*.
- *ReputationTracker*
  - Gestisce un sistema di reputazione dei nodi, la cui valutazione serve in diversi contesti, come selezionare in modo ottimale l'Oracolo o l'Indice da impiegare.

- *Token*
  - Il token ERC-20 impiegato nel sistema ZONIA. Viene utilizzato per saldare l'importo delle richieste e incentivare i partecipanti alla rete.
- *Gate*
  - È il punto di entrata per le richieste in altre catene, infatti ogni catena supportata deve avere il suo contratto Gate corrispondente deployato.

Vi sono inoltre 3 interfacce implementate:

- *Gate Interface*
  - Interfaccia che tutti i contratti Gate devono implementare, definisce i metodi che permettono ai *Consumers* di sottomettere le richieste, parallelamente permette a *Oracles* e *Indexers* di leggere le richieste e scrivere i risultati nella chain.
- *Producer Interface*
  - Facilita l'integrazione dei *Producers*, consentendo l'interazione con dispositivi eterogenei, indipendentemente dall'architettura. I dispositivi IoT forniscono descrittori semantici per permettere interazioni coerenti con *Indexer* e *Oracle*.
- *Discovery Interface*
  - Definisce come i *Producers* sono richiamati dal sistema. Incorpora query geospaziali per localizzare i *Producers* basandosi sulla posizione geografica degli stessi.

La declinazione multi-chain di ZONIA, ZONIA-MC, introduce la capacità di scegliere, tramite una parametrizzazione sulla richiesta, la chain/ambiente più adatto (per costi, latenza, efficienza energetica) su cui effettuare operazioni on-chain e di interagire con più chain target tramite Gate multipli[2].

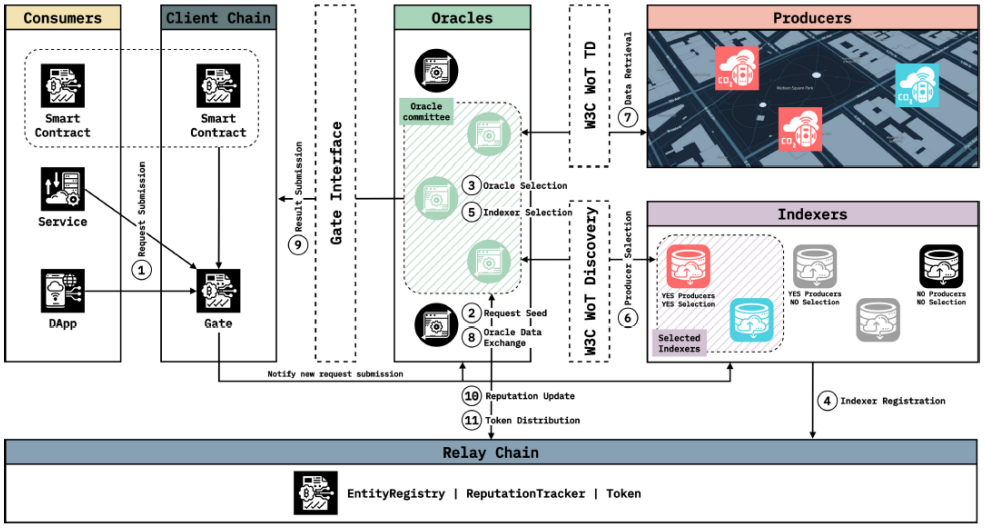


Figura 2.1: Architettura ad alto livello di ZONIA [2]

2.2 Requisiti Funzionali dell'applicazione

L'applicazione presentata va a interagire con gli smart contracts di ZONIA-MC, ha la capacità di richiedere dati da dispositivi IoT tramite transazioni sulla Blockchain, immagazzina i dati e li visualizza tramite un'interfaccia utente intuitiva.
Inoltre l'applicazione va a integrare *Metamask* come wallet manager e lo utilizza per firmare le transazioni nella testnet di Ethereum, Sepolia.
L'obiettivo è inoltre rendere concretamente utilizzabili i dati IoT forniti, integrandoli per creare assicurazioni smart automatiche, trasparenti e sicure. Si tratta quindi di creare una piattaforma per assicurazioni intelligenti, dove gli smart contract agiscono come vere e proprie polizze assicurative.

Di seguito verrà brevemente descritto il flusso tipico di esecuzione implementato nell'applicazione:

- La compagnia assicurativa crea un contratto assicurativo partendo da un Factory, nel quale inserisce indirizzo del wallet del cliente, il premio assicurativo e i parametri necessari per richiedere i dati IoT da ZONIA.
- Una volta creato il contratto l'assicuratore lo può "Fondare", ovvero trasferire l'importo assicurativo concordato nel contratto. Da questo momento è il contratto assicurativo stesso che ha il premio.
- Il cliente può firmare il contratto pagando una percentuale del premio e ora il contratto è considerato attivo a tutti gli effetti.
- Il cliente può infine richiedere una "Verifica", in cui viene controllato, tramite i dati ricevuti dal interfaccia Gate di ZONIA, se è possibile effettuare la liquidazione. Nei in casi in cui si può procedere, il contratto viene liquidato e il premio viene trasferito nel account del cliente.

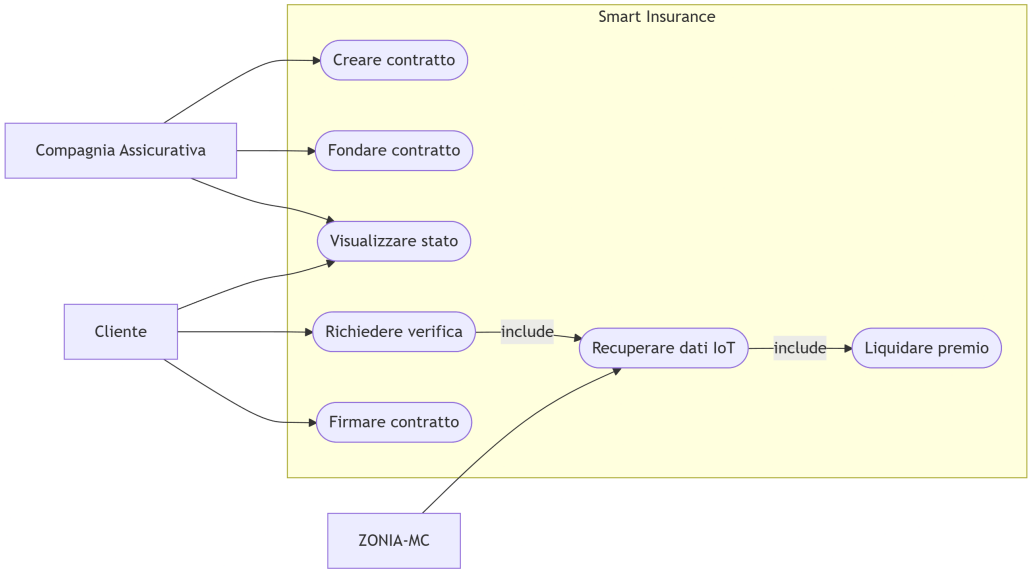


Figura 2.2: Flowchart delle interazioni tra attori e sistema smart insurance.

2.2.1 Un Esempio Concreto

Un potenziale cliente possiede un campo agricolo e decide, per tutelarsi, di stipulare una polizza contro la siccità con la nostra compagnia assicurativa. Tramite la piattaforma, l'assicuratore crea un nuovo contratto dal proprio account, specificando l'indirizzo wallet del cliente e il premio assicurativo di 1 ETH. Dopo aver "fondato" il contratto (trasferendo il premio), il cliente lo firma versando il 10% dello stesso. Durante la stagione, i sensori di umidità collegati a ZONIA-MC inviano periodicamente dati alla Blockchain. Al termine del periodo assicurato, il cliente richiede una verifica: l'applicazione interroga ZONIA-MC e rileva che i valori di umidità sono stati inferiori alla soglia critica per oltre 10 giorni consecutivi. Il contratto viene quindi liquidato automaticamente, trasferendo l'importo previsto al wallet del cliente.

Capitolo 3

Progettazione e Implementazione

In questo capitolo parleremo di come è stata implementata l'applicazione utilizzando un approccio top-down.

3.1 Architettura e Design

Al lancio dell'applicazione e dopo la scelta del ruolo tra cliente o assicuratore, all'utente viene presentata la Home Page, e viene chiesto di collegare l'account Metamask per recuperare l'indirizzo del wallet. La Home Page si presenta con un grafico a torta che mostra lo stato dei contratti creati e alcuni dati diagnostici. Da qui si può navigare verso le altre schermate che permettono di visualizzare i dettagli dei contratti creati, creare contratti e modificare i *Chain Params*.
Di seguito viene mostrato un diagramma che mostra i collegamenti tra le schermate.

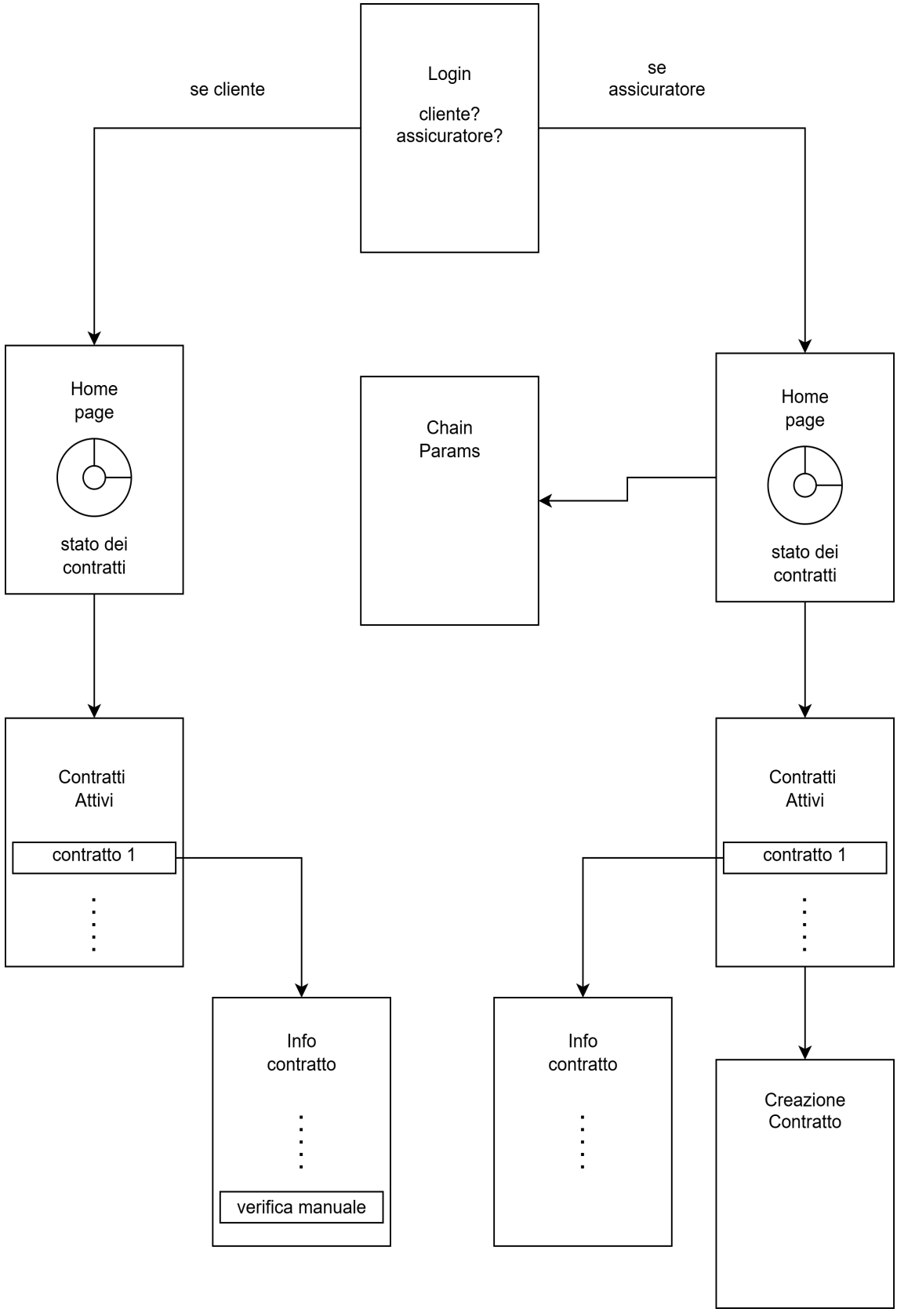


Figura 3.1: FlowChart UI dell'applicazione.

Le tecnologie utilizzate sono le seguenti:

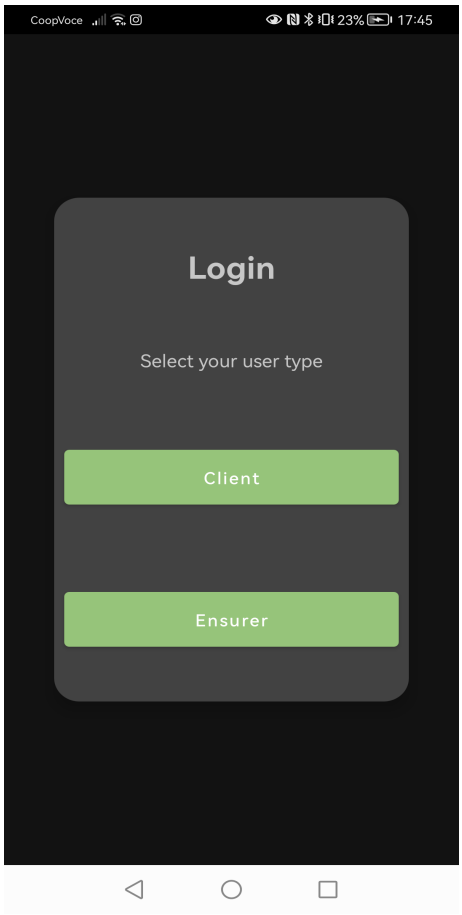
- DApp sviluppata in Android (Kotlin).
- Smart Contracts sviluppati in Solidity/Hardhat.
- La Blockchain utilizzata è Ethereum Sepolia Testnet.

3.2 Schermate e UI

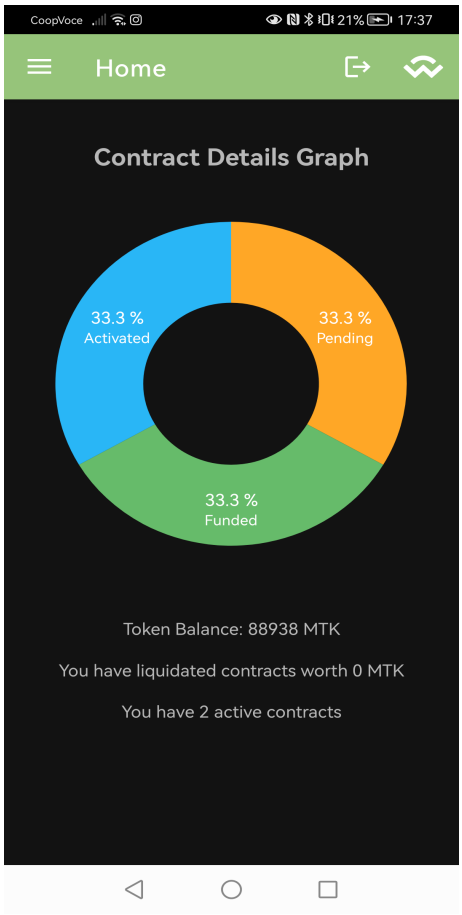
In questa sezione si va a osservare la UI più dettagliatamente, spiegandone le funzionalità.

3.2.1 Login e Home Page

Al lancio dell'applicazione, viene richiesto di scegliere uno dei ruoli che deve interpretare l'utente. Questi sono *Client* e *Ensurer*, ognuno ha uno scopo diverso e quindi anche le funzionalità dell'applicazione cambiano a seconda di questa scelta. Per esempio, il *Client* essendo il fruitore delle assicurazioni, non ne può creare.
Una volta scelto il ruolo si viene indirizzati sulla Home Page che, come detto in precedenza, presenta un grafico a torta con lo stato dei contratti, che può essere uno tra: *Pending*, *Funded*, *Activated* e *Liquidated*. Inoltre presenta diciture che mostrano il bilancio dei token, il numero dei token liquidati e il numero dei contratti attualmente attivi.



(a) Schermate - Login

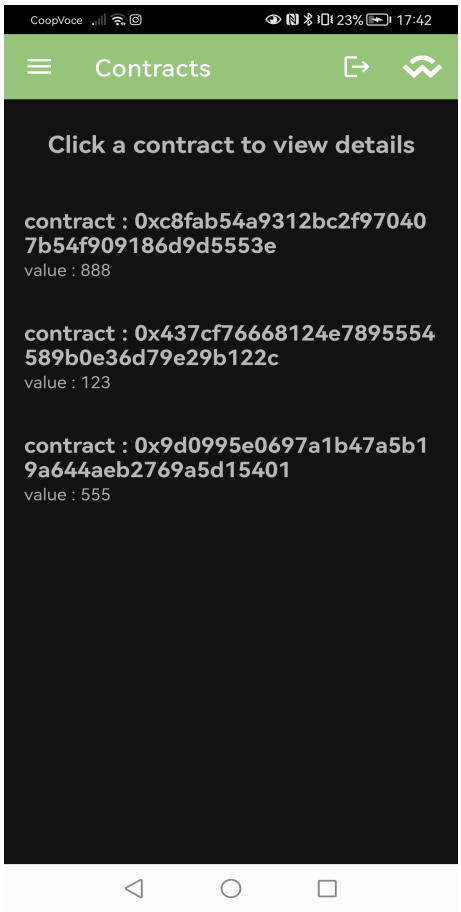


(b) Schermate - Home

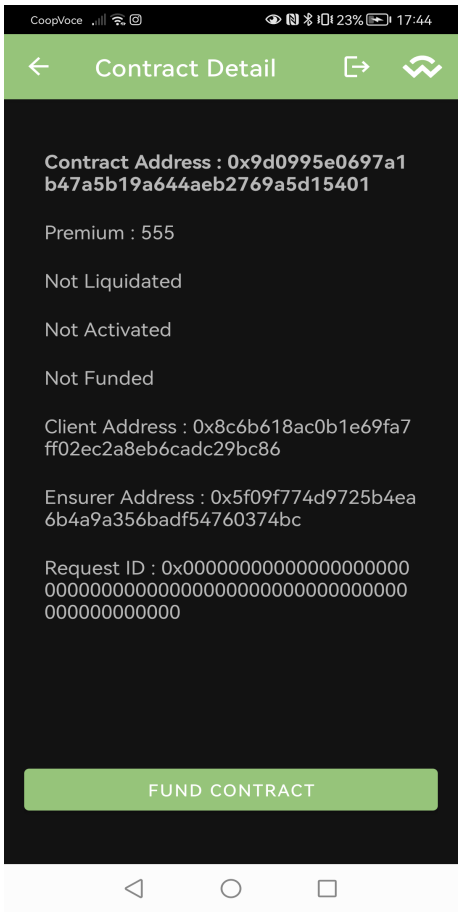
Figura 3.2: UI dell'applicazione: Schermate di Login e Home.

3.2.2 Contratti

Accedendo alla schermata *Contracts* tramite il menù di navigazione, è possibile visualizzare un riepilogo dei contratti. Nel caso in cui l'utente sia un *Ensurer*, vengono mostrati tutti i contratti creati, mentre per un utente con ruolo di *Client* vengono elencati i propri contratti. Selezionando un contratto dall'elenco si accede alla relativa pagina di dettaglio, dove sono riportate informazioni significative quali l'indirizzo dell'assicuratore, l'indirizzo del contratto e l'importo del premio. Inoltre si possono fare le operazioni di *Founding*, *Activation*, *Liquidation*. Di cui parleremo successivamente.



(a) Schermate - Contracts



(b) Schermate - Details

Figura 3.3: UI dell'applicazione: Schermate di Lista contratti e Dettagli.

3.2.3 Creazione dei Contratti

Questa schermata permette a un utente di tipo *Ensurer* di creare un nuovo contratto. Indicando l'indirizzo del beneficiario e il premio assicurativo. Inoltre si devono inserire i parametri riguardanti la query che il contratto invierà a ZONIA. Si può scegliere tra una query sull'umidità o temperatura e indicare tramite una mappa le coordinate geografiche.

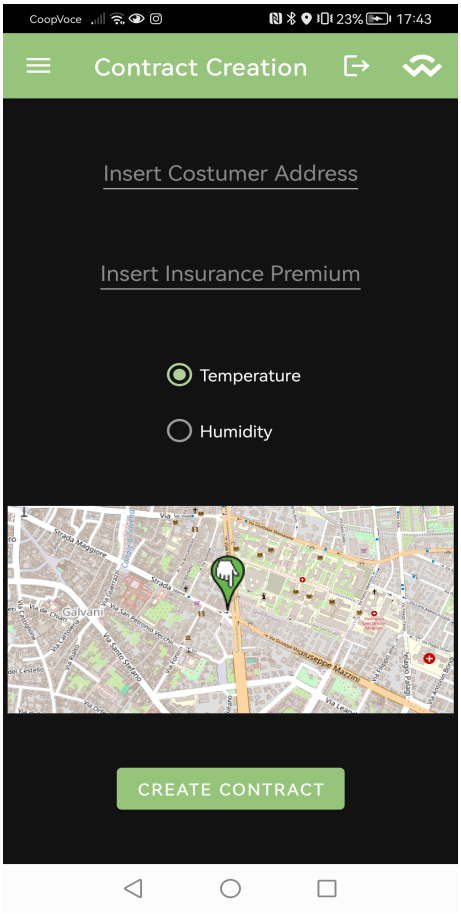


Figura 3.4: UI dell'applicazione: Schermata di creazione Contratti.

3.2.4 Chain Params

Questa schermata permette di aggiungere ulteriori parametri alla query, questi indicano i pesi con cui l'algoritmo di ZONIA sceglie la chain target tra Ethereum Sepolia, Polygon Amoy e Avalanche Fuji. I valori che possono assumere vanno da 0 a 100 e la somma dei quattro parametri non deve superare 100.

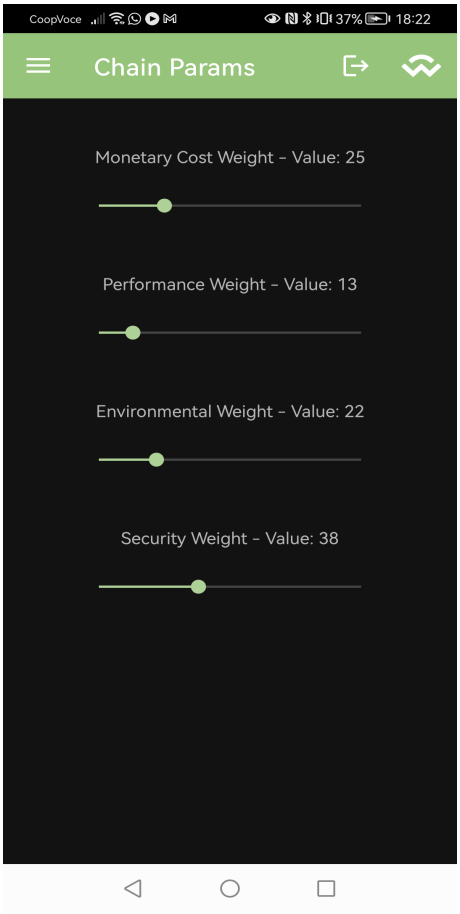


Figura 3.5: UI dell'applicazione: Schermata di scelta dei chain params.

Questi pesi rappresentano:

- **Costo monetario**: dipende dalla gas base fee corrente richiesta per scrivere i dati e dal tasso di cambio tra il token nativo della Blockchain e un'altra valuta di riferimento, come il dollaro statunitense.
- **Performance**: viene valutata utilizzando una combinazione di metriche dinamiche e statiche, al fine di catturare sia le condizioni di rete in tempo reale sia le capacità fondamentali del protocollo. Le metriche dinamiche sono fondamentali per valutare le prestazioni correnti. La più significativa tra queste è la base fee (per le Blockchain che adottano EIP-1559), la quale funge da indicatore on-chain diretto della congestione della rete. Da essa è possibile derivare un tempo medio di attesa stimato, che fornisce una misura della latenza dal punto di vista dell'utente.
- **Impatto ambientale**: conoscendo la localizzazione geografica dei miner, è possibile stimare l'impronta di carbonio utilizzando la matrice elettrica di ciascun Paese. Si tratta comunque di un'attività di natura complessa e orientata alla ricerca.
- **Sicurezza**: le Blockchain che usano Proof-of-Work come meccanismo di consenso sono generalmente considerate le più sicure. Per le Blockchain basate su Proof-of-Stake, viene analizzata la decentralizzazione dei saldi in stake, al fine di assicurare che il controllo non sia concentrato in pochi validatori[12].

3.3 Implementazione Frontend

3.3.1 Main Activity e Overview

L'applicazione si basa su una navigation drawer activity con gestione dei due ruoli che servono per caricare layout diversi, oltre che gestione del portafoglio Metamask con WalletConnect. Nel metodo onCreate() viene recuperato il ruolo selezionato dall'utente dalle SharedPreferences e configurato il Navigation Drawer. Se l'utente è un cliente, vengono nascoste le voci relative alla creazione dei contratti e alla configurazione dei Chain Params:

Listing 3.1: Navigation Drawer + Nav Controller

```
val drawerLayout: DrawerLayout = binding.drawerLayout
val navView: NavigationView = binding.navView
val navController = findNavController(R.id.
    nav_host_fragment_content_main)

appBarConfiguration = AppBarConfiguration(
    setOf(
            R.id.nav_home, R.id.nav_gallery, R.id.
                nav_contract_creation,
            R.id.nav_settings
    ), drawerLayout
)
setupActionBarWithNavController(navController,
    appBarConfiguration)
navView.setupWithNavController(navController)
if (userRole == "cliente"){
    val menu = navView.menu
    menu.findItem(R.id.nav_contract_creation).isVisible =
        false
    menu.findItem(R.id.nav_chain_params).isVisible = false
}
```

Viene poi controllato se esiste già una sessione di collegamento con Metamask. In quel caso viene riutilizzata, altrimenti viene ricreata con Appkit.connect() e facendo partire l'intent di Metamask per accettare la connessione.
I parametri di connessione sono la chain di Sepolia "eip155:11155111" e la lista di metodi dei quali si richiede il permesso.
Tutte le pagine soggette a navigazione sono implementate con Fragment e ViewModel.

Listing 3.2: Creazione sessione

```
val pairing = CoreClient.Pairing.create() ?: throw
    IllegalStateException("Failed to create pairing")

val getPairing = CoreClient.Pairing.getPairings()
val uri = "metamask://wc?uri=" + Uri.encode(pairing.uri)
val intent = Intent(Intent.ACTION_VIEW,uri.toUri())
startActivity(intent)
AppKit.connect(
    connectParams = Modal.Params.ConnectParams(
        sessionNamespaces = mapOf(
            "eip155" to Modal.Model.Namespace.Proposal(
                chains = listOf("eip155:11155111"),
                methods = listOf(
                    "eth_sendTransaction",
                    "personal_sign",
                    "eth_sign",
                    "eth_accounts",
                    "eth_call"),
                events = listOf("chainChanged", "
                    accountsChanged")
            )
        ),
        pairing = pairing
    )
)
```

3.3.2 Login e Home page

L'Activity di partenza dell'applicazione è la Login Activity, questa serve a salvare il ruolo dell'utente per poi navigare alla Main Activity. La Home Page sfrutta la libreria "MPAndroidChart" per mostrare un grafico a torta che visualizza lo stato dei contratti creati. Viene stampato il saldo dei token dell'utente e la somma dei contratti liquidati.
Per farlo è necessario recuperare i dati dei contratti dalla Blockchain partendo dall'indirizzo del portafoglio dell'utente aggiornandoli ogni volta che la Home Page viene aperta.

Listing 3.3: Fetch Contratti

```
fun loadContracts(){
    _totalLiquidated.value = 0
    _isLoading.value  = true
    viewModelScope.launch {
        try {
            val contractAddresses: List<String> = when (
               userRole) {
                "cliente" -> contractCalls.
                   getInsuranceContractsByInsured(
                   getUserAddress(),getUserAddress())
                "assicuratore" -> contractCalls.
                   getAllInsuranceContracts(getUserAddress())
                else -> emptyList()
            }
            val contractList = mutableListOf<Contract>()
            for (address in contractAddresses){
                val data = contractCalls.getContractVariables
                   (getUserAddress(),address)
                Log.d("data", "Contract␣data␣for␣$address:␣
                   $data")
                if (data["assicurato"].toString() !=
                   getUserAddress().lowercase() && (userRole
                   == "cliente")){
                    Log.wtf("loadContracts", "Contract␣
                       $address␣is␣not␣associated␣with␣the␣
```

```
                        user␣address␣${getUserAddress()}")
17                  }
18                  val version = data["version"]?.toString() ?:
                        "unknown"
19
20                  if(data["liquidato"] as Boolean){
21                      val premio = (data["premio"] as
                            BigInteger).toInt()
22                      _totalLiquidated.value = (
                            _totalLiquidated.value ?: 0) + premio
23                  }
24                  ...
25              }
26          }
27          catch (e: Exception) {
28              Log.e("HomeViewModel", "Error␣loading␣contract␣
                    addresses", e)
29              _contracts.postValue(emptyList())
30          }
31          _isLoading.value = false
32      }
33  }
```

3.3.3 Creazione del contratto

Nella pagina per la creazione dei contratti assicurativi viene inserito il wallet del beneficiario , l'importo del premio, deve venire indicato se la query sia per un controllo di umidità o temperatura e infine la geo-locazione tramite una mappa. Dopo di che viene chiamata la funzione createContract():

- Viene controllata la validità del wallet tramite un'espressione regolare e se ethGetbalance di web3 non ritorna un'eccezione.
- Se il contratto è valido viene fatta la chiamata al factory contract per la creazione di un nuovo contratto.

3.3.4 I Contratti

Listing 3.4: Classe Contratto

```
data class Contract(
    val address: String,
    val premio: UInt,
    val isLiquidato: Boolean,
    val isAttivato: Boolean,
    val isFundend: Boolean,
    val addressAssicurato: String,
    val addressAssicuratore: String,
    val requestId: String
)
```

Viene creato un Adapter per la Recicler View che mostra la lista di tutti i contratti.
Quando viene cliccato un contratto viene creata un'action che collega il contract fragment al contract detail fragment passandogli tutti gli argomenti necessari.

Listing 3.5: Action

```
holder.itemView.setOnClickListener {
    val action = ContractsFragmentDirections.
        actionContractsFragmentToContractDetailFragment(
        contract.address,
        contract.premio.toString(),
        contract.isLiquidato,
        contract.isAttivato,
        contract.isFundend,
        contract.addressAssicurato,
        contract.addressAssicuratore,
        contract.requestId
    )
    navController.navigate(action)
}
```

3.3.5 Dettagli Contrattuali

I dati precedentemente passati vengono recuperati e visualizzati. Qui inoltre si svolgono le principali azioni rivolte ai contratti:

- *Founding*
  - Per questa operazione viene innanzitutto richiesto l'approvazione del trasferimento di token da assicuratore al contratto. Se l'operazione di approve va a buon fine allora si richiama la funzione *fundContract*.
- *Activation*
  - Per questa operazione, come la per la precedente, deve essere approvato lo scambio di token per poter chiamare, una volta ricevuta l'approvazione, la funzione *activateContract*.

- *Liquidation*
  - Questa operazione richiede, oltre che l'approvazione per il trasferimento dei token, anche che vengano richiesti i dati dall'oracolo tramite comunicazione con gli smart contract forniti da ZONIA.

3.3.6 Metamask e chiamate alla Blockchain

Prima delle activity vengono inizializzati CoreClient e Appkit per la connessione alla Blockchain e al wallet.

Listing 3.6: initialize Coreclient e Appkit

```
CoreClient.initialize(
    application = this,
    projectId = projectId,
    metaData = appMetaData,
    connectionType = connectionType,
    onError = { error -> Log.e("MyApplication",error.toString
        ()) }
)

AppKit.initialize(
    init = Modal.Params.Init(CoreClient),
    onSuccess = {
        Log.i("MyApplication","appkit initialized")
    },
    onError = { error ->
        Log.e("MyApplication",error.toString())
    }
)
```

Il file ContractCalls contiene tutte le funzioni che interagiscono con la Blockchain e i contratti. Le funzioni che si occupano di leggere dati dalla Blockchain, ovvero che non hanno bisogno di essere firmate, vengono interrogate tramite ethcall di web3j.

Listing 3.7: Get token Balance

```
suspend fun getTokenBalance(address: String): String =
    withContext(Dispatchers.IO) {
     val function = Function(
         "balanceOf",
         listOf(Address(address)),
         listOf(TypeReference.create(org.web3j.abi.datatypes.
             generated.Uint256::class.java))
     )

     val encodedFunction = FunctionEncoder.encode(function)

     val response = web3.ethCall(
         Transaction.createEthCallTransaction(address,
             mytokenAddress, encodedFunction),
         DefaultBlockParameterName.LATEST
     ).send()

     val decoded = FunctionReturnDecoder.decode(response.value
         , function.outputParameters)

     return@withContext decoded[0].value.toString()
}
```

Per le transazioni che richiedono una firma, viene utilizzato AppKit, che consente di interagire con il portafoglio MetaMask. In questo modo l'utente può firmare le transazioni utilizzando la propria chiave privata, garantendo autenticità e sicurezza nell'operazione.

Listing 3.8: Create new Contract

```
suspend fun createNewContract(addrAssicuratore: String,
    addrAssicurato: String, premio: Uint256): String =
    withContext(Dispatchers.IO){
    ...
    val request = com.reown.appkit.client.models.request.
        Request(
        method = "eth_sendTransaction",
        params = paramsArray.toString()
    )
    val deferred = kotlinx.coroutines.CompletableDeferred<
        String>()
    AppKit.setDelegate(object : AppKit.ModalDelegate {
        override fun onSessionRequestResponse(response: Modal
            .Model.SessionRequestResponse) {
            val hash = (response.result as Modal.Model.
                JsonRpcResponse.JsonRpcResult).result
            Log.d("DelegateTest", "Transaction hash: $hash")
            if (!deferred.isCompleted) deferred.complete(hash
                )
        }
        ...
    })
    AppKit.request(
        request = request,
        onSuccess = { result ->
            Log.i("approveTokenTransfer", "Transaction hash:
                $result")},
        onError = { error ->
            Log.e("approveTokenTransfer", "Transaction failed
                :${error.message}")}
    )
    val txHash = deferred.await()
    Log.d("approveTokenTransfer", "Final transaction hash:
        $txHash")
    return@withContext txHash
}
```

3.4 Contratti Solidiy

3.4.1 Token

Consiste in un Token ERC20 creato con il wizard di Openzeppelin.

Listing 3.9: Token

```
contract MyToken is ERC20, Ownable, ERC20Permit {
    constructor(
        address recipient,
        address initialOwner
    ) ERC20("MyToken", "MTK") Ownable(initialOwner)
       ERC20Permit("MyToken") {
        _mint(recipient, 1000000 * 10 ** decimals());
    }
}
```

3.4.2 Factory Contract

Il factory contract è il contratto che permette all'assicuratore di deployare altri contratti direttamente dall'applicazione sulla Blockchain sepolia. Oltre ai metodi di get dei contratti fatti dal factory, il metodo createInsurance crea una nuova istanza di contratto con i dati in input.

Listing 3.10: Create Insurance

```
function createInsurance(
    address assicurato,
    uint premioAssicurativo,
    string memory topic,
    uint256 lat,
    uint256 lng
) external onlyInsurer returns (address) {
    require(premioAssicurativo > 0, "Premio must be greater
       than 0");
    InsuranceContract newInsurance = (new InsuranceContract)(
        assicuratore,
        assicurato,
```

```
12          tokenAddress,
13          premioAssicurativo,
14          gateAddress,
15          zoniaTokenAddress,
16          topic,
17          lat,
18          lng
19      );
20      allInsuranceContracts.push(address(newInsurance));
21      insuranceContractsByInsured[assicurato].push(address(
            newInsurance));
22      emit NewInsuranceContractCreated(
23          address(newInsurance),
24          assicurato,
25          premioAssicurativo
26      );
27      return address(newInsurance);
28  }
```

3.4.3 Insurance Contract

Il contratto vero e proprio ha una copia dell'interfaccia IGate per le richieste al contratto Gate. Le funzioni di gate con cui interagisce sono submitRequest, getRequest e getResult.

Listing 3.11: Request Zonia Data

```
1  function requestZoniaData() external onlyInsured {
2      require(!liquidato, "Already␣liquidated");
3      require(attivato, "Contract␣not␣activated␣yet");
4      require(funded, "Contract␣has␣to␣be␣funded");
5      require(
6          zoniaToken.allowance(assicurato, address(this)) >= 1,
7          "Insufficient␣ZoniaToken␣allowance"
8      );
9      require(
10         zoniaToken.balanceOf(assicurato) >= 1,
11         "Insufficient␣ZoniaToken␣balance"
```

```
12      );
13      IGate.InputRequest memory req = IGate.InputRequest({
14          query: string(
15              abi.encodePacked(
16                  '{"topic":"saref:',topic,",
17                  "geo":{"type":"Feature","geometry":
18                  {"type":"Point","coordinates":[',
19                  helper(lat),
20                  ",",
21                  helper(lng),
22                  ']},"properties":{"radius":1000}}}'
23              )
24          ),
25          chainParams: IGate.ChainParams(chp1, chp2, chp3, chp4
                ),
26          ko: ko,
27          ki: ki,
28          fee: fee
29      });
30      require(
31          zoniaToken.transferFrom(assicurato, address(this),
                fee),
32          "TransferFrom failed"
33      );
34      require(
35          zoniaToken.approve(address(gate), fee),
36          "Approve to Gate failed"
37      );
38      requestId = gate.submitRequest(req);
39  }
```

Capitolo 4

Analisi e Test

In questo capitolo vengono analizzate le implicazioni in termini di costo e tempo derivanti dall'esecuzione delle stesse operazioni su differenti Blockchain.

4.1 Flusso della richiesta ZONIA

Quando l'utente richiede la verifica del contratto, con conseguente produzione della richiesta verso i contratti ZONIA e successiva liquidazione, va a interrogare la funzione *submitRequest* del contratto gate esposto su Ethereum Sepolia.
A questo punto inizia il flusso di risoluzione della richiesta. Le varie fasi saranno descritte di seguito in modo sintetico, indicando anche le funzioni coinvolte, utili per la successiva analisi dei costi.

- *Request Submission*
  - L'utente sottomette la richiesta al sistema attraverso il contratto Gate.
    Le funzioni interessate on-chain sono: *gate.submitRequest*, *token.transfer*, *token.approve*.

- *Request Seed*
  - Fase in cui gli Oracoli generano un numero a partire da una VRF (Verifiable Random Function), serve come base per selezionare in modo equo e imprevedibile i nodi che parteciperanno dopo.
    Le funzioni interessate on-chain sono: *gate.submitseed* e una tantum *gate.invalidateSeed*.
- *Reputation Reconstruction*
  - Gli Oracoli raccolgono le reputazioni di tutti gli Oracoli e degli Indexer.
- *Oracle Selection, Indexer Registration*
  - Queste due fasi avvengono parallelamente: gli Oracoli vengono selezionati in modo competitivo e bilanciato per formare un comitato che gestirà la richiesta. Gli Indexer (che organizzano e trovano i dati dei produttori) si registrano se hanno dati che soddisfano la richiesta.
    Le funzioni interessate on-chain sono: *gate.submitCommitteeHash*, *gate.submitCommittee*, *gate.applyToRequest*, *gate.closeIndexerRegistration*.
- *Chain Selection*
  - Processo off-chain dove il comitato degli Oracoli, a seconda dei parametri della richiesta, sceglie su quale Blockchain continuare le operazioni.
- *Indexer Selection, Producer Selection*
  - Il comitato seleziona off-chain un sottoinsieme degli Indexer registrati che recupereranno i descrittori dei Producers.
- *Data Retrieval*
  - Processo off-chain dove il comitato richiede ai Producer i dati reali.

- *Oracle Data Exchange*
  - Gli Oracoli del comitato si scambiano i dati tra di loro sulla Blockchain scelta.
    Le funzioni interessate on-chain sono: *gate.submitHash*, *gate.submitDataPoints*.
- *Result Submission*
  - Il comitato scrive il risultato finale sulla Blockchain da cui era partita la richiesta.
    La funzione interessata on-chain è: *gate.submitResult*
- *Reputation Update*
  - Le reputazioni di Oracoli, Indexer e Producer vengono aggiornate in base a come si sono comportati.
    Le funzioni interessate on-chain sono: *gate.commitScoreHash*, *gate.updateScores*, *gate.banOracles*, *gate.banIndexers*
- *Token Distribution*
  - Fase di distribuzione dei token tra i partecipanti alla richiesta.
    Le funzioni interessate on-chain sono: *token.transfer*, *token.approve*.

4.2 Analisi del Gas Cost

In questa sezione verranno analizzati i costi in termine di gas delle richieste e quindi della smart insurance, in funzione della Blockchain selezionata dall'algoritmo di chain selection di ZONIA. Le misurazioni sono state prese usando l'ambiente di sviluppo Hardhat e il plugin hardhat-gas-reporter, permettendo di avere informazioni precise sul costo del gas di ogni funzione degli smart contracts.
Il seguente grafico rappresenta il costo di una chiamata alle funzioni della Smart Insurance.

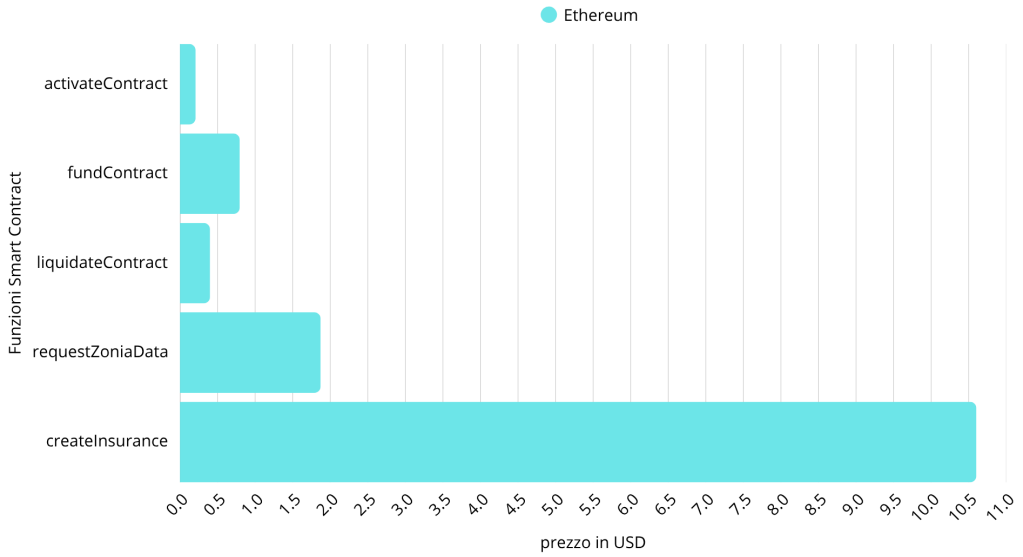


Figura 4.1: Grafico dei prezzi delle funzioni della Smart Insurance.

Si può notare come il costo di *createInsurance* è molto maggiore delle altre funzioni in quanto va a deployare uno smart contract vero e proprio che rappresenta l'assicurazione.

I seguenti grafici, invece, mostrano i prezzi in USD delle funzioni di ZONIA descritte sopra che, come precedentemente detto, sono le funzioni dei contratti che vengono eseguite quando viene sottomessa una richiesta.
Per completezza sono riportate anche le fasi eseguite off-chain o con solo operazioni di get on-chain. Queste hanno costo di gas nullo. Si nota subito che i costi del gas in Ethereum sono molto più grandi rispetto a Avalanche e Polygon che hanno costi molto più contenuti.

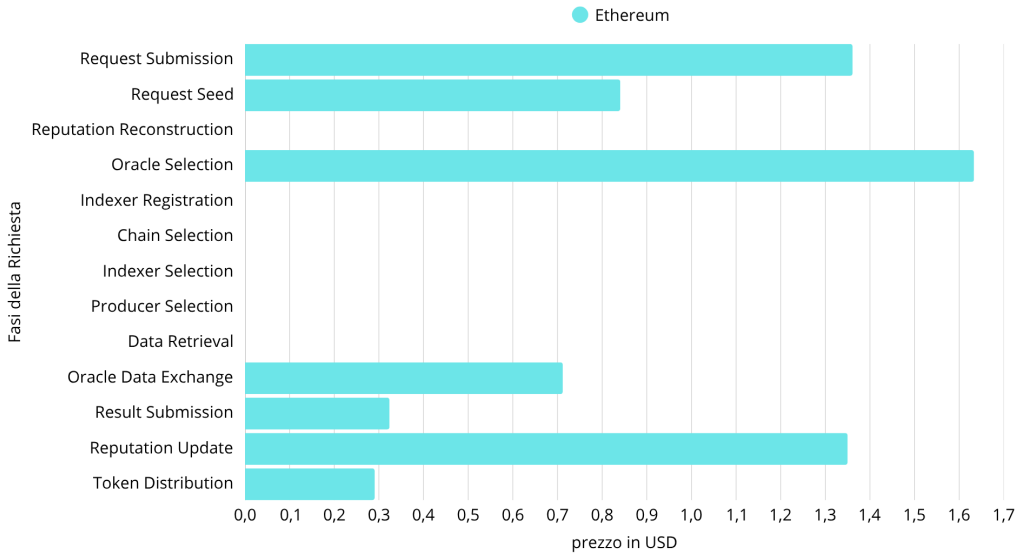


Figura 4.2: Grafico dei prezzi delle fasi della richiesta ZONIA Ethereum.

La differenza dei costi del gas tra le catene deriva da diversi fattori: ad esempio ci sono differenze di scalabilità tra le reti dovute a differenze architetturali.
Polygon e Avalanche grazie all'uso di diverse *side chains* possono permettersi di gestire molte più TPS di Ethereum[11].
Questo va infatti a congestionare meno la rete e, conseguentemente, ad abbassare i costi delle fee.

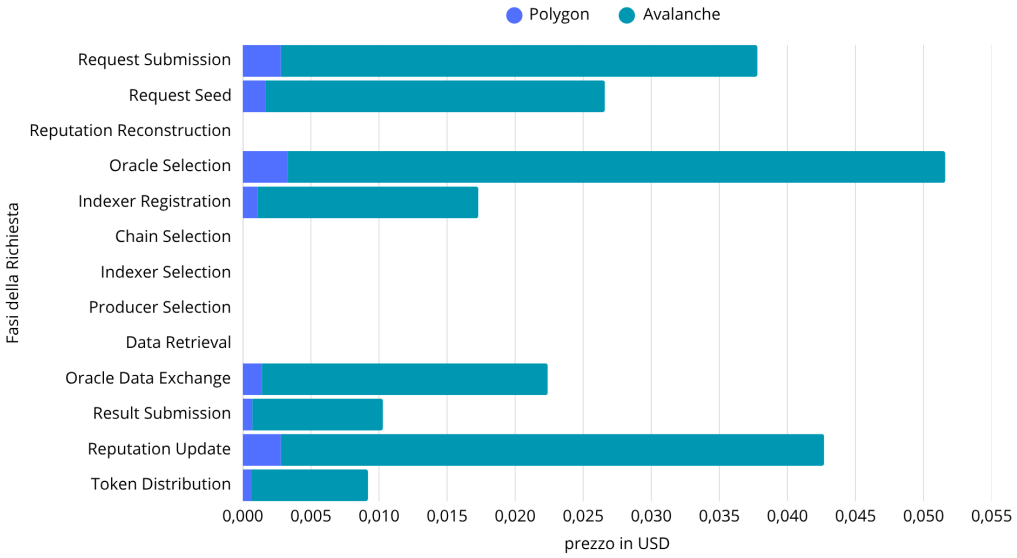


Figura 4.3: Grafico dei prezzi delle fasi della richiesta ZONIA Polygon/Avalanche.

La fase di *Oracle Data Exchange* può venire eseguita in una delle tre Blockchain dipendentemente dalla scelta dei Chain Params della richiesta. Questa fase è particolarmente delicata in termini di costo del gas in quanto il gas cost aumenta all'aumentare degli Oracoli[2].
Come si vede nel grafico seguente, il costo di questa fase viene abbattuto di molto se il comitato sceglie una rete diversa da Ethereum.

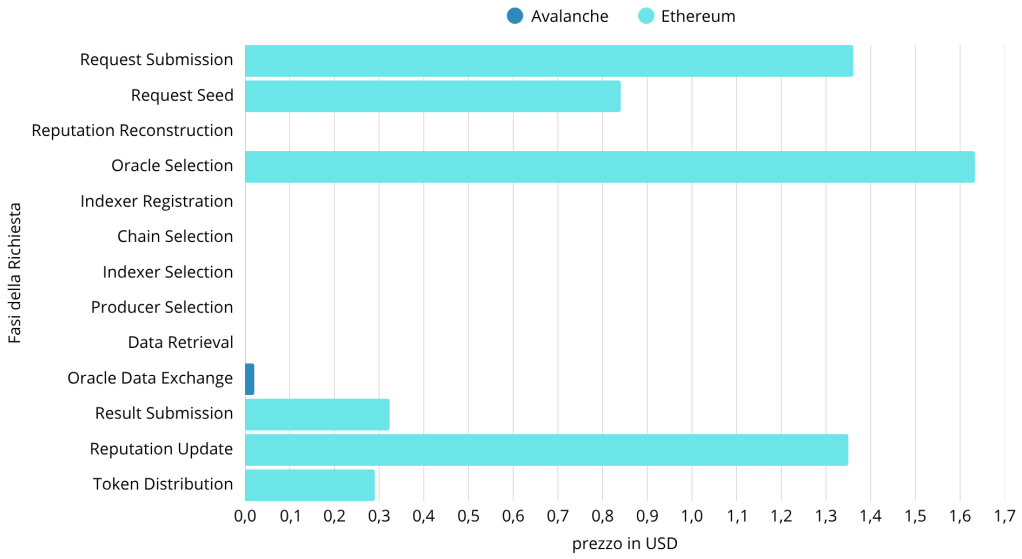


Figura 4.4: Grafico dei prezzi delle fasi della richiesta ZONIA con la fase *Oracle Data Exchange* gestita su Avalanche

4.3 Analisi di Tempo

Andiamo ora ad analizzare in termini di tempo una transizione generica su Ethereum, Polygon e Avalanche.
Il *Block Time* è il tempo in media necessario per aggiungere un nuovo blocco nella Blockchain.

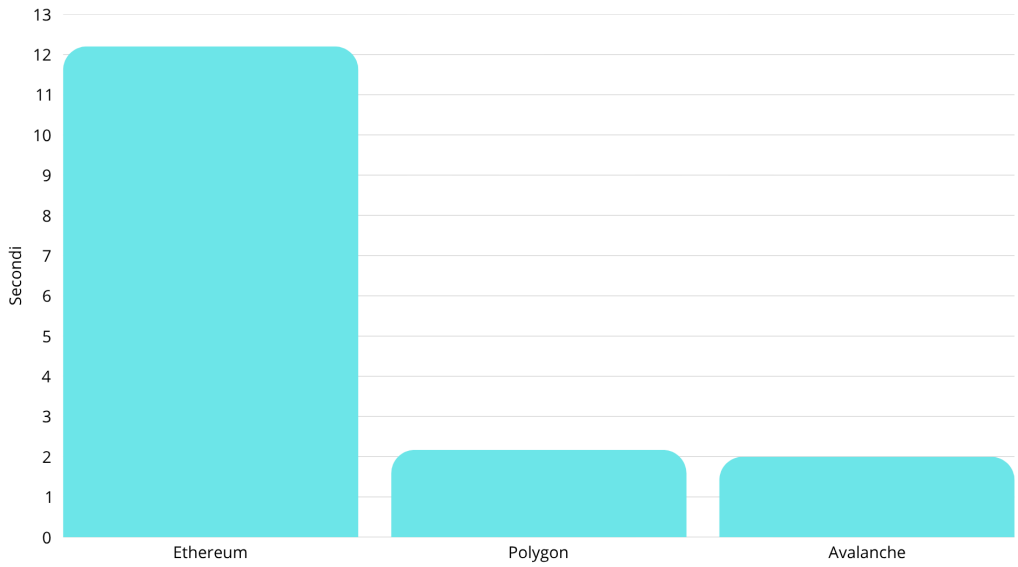


Figura 4.5: Block Time medio in secondi delle reti Ethereum, Polygon, Avalanche

Ethereum soffre di un blocktime elevato anche per la grande decentralizzazione che caratterizza la rete. Questo necessita infatti di più tempo per la trasmissione del blocco ai nodi.
Considerando di usare un gas maggiore o uguale alla media per la transazione si può dire che questa viene quasi sicuramente minata al prossimo blocco, quindi attende in media il tempo di un blocktime.

4.3.1 Considerazioni

Nonostante i costi di gas e i tempi di conferma più elevati, l'utilizzo di Ethereum può risultare preferibile in determinate situazioni per via della sua maggiore sicurezza e affidabilità. La rete Ethereum, infatti, si distingue per un elevato grado di *decentralizzazione* e per il grande numero di validatori attivi, distribuiti globalmente. Con il passaggio al meccanismo di consenso *Proof of Stake*, la rete conta ad oggi, 1 miliardo di validatori indipendenti e un valore totale in staking di decine di miliardi di dollari, rendendo economicamente e tecnicamente impraticabile un eventuale attacco del 51% [13]. Tale livello di sicurezza è superiore rispetto a soluzioni come Polygon o Avalanche, dove la sicurezza dipende da un numero inferiore di validatori. Pertanto, la scelta di Ethereum è giustificata in scenari dove la *sicurezza* e l'*immutabilità* delle transazioni sono prioritarie rispetto a costi e tempi, come ad esempio per contratti assicurativi di alto valore o per operazioni che richiedono un elevato grado di fiducia. Al contrario, reti come Polygon o Avalanche risultano più adatte per applicazioni che privilegiano la *scalabilità* e l'*efficienza economica*, in quanto offrono costi di gas sensibilmente inferiori e tempi di conferma ridotti, a fronte di un compromesso sul livello di sicurezza complessivo.

Conclusioni

Il lavoro presentato in questa tesi nasce dall'esigenza di rendere più accessibile e concretamente utilizzabile il sistema ZONIA. Nonostante la solidità concettuale dell'architettura, la mancanza di un client dedicato rappresentava un ostacolo significativo alla sua adozione. Per questo motivo, il progetto ha avuto come obiettivo principale la realizzazione di un'applicazione che permettesse di interagire in modo semplice, sicuro e intuitivo con gli smart contracts di ZONIA, senza la necessità di possedere competenze tecniche avanzate.
L'applicazione sviluppata integra il wallet *Metamask* per la gestione delle transazioni sulla testnet Ethereum Sepolia e consente di accedere alle funzionalità di richiesta e verifica dei dati provenienti da dispositivi IoT. Il sistema permette di creare, firmare e gestire contratti assicurativi intelligenti, rendendo tangibile il concetto di *smart insurance*. In questo modo, il lavoro svolto contribuisce a colmare il divario tra la complessità tecnologica della Blockchain e l'usabilità richiesta dagli utenti finali.
È stata infine condotta un'analisi approfondita dei costi e dei tempi di esecuzione delle operazioni on-chain, con l'obiettivo di comprendere l'impatto economico e prestazionale derivante dall'utilizzo di diverse Blockchain (Ethereum, Polygon, Avalanche).
Dai risultati emersi si può concludere che:

- **Ethereum** garantisce il livello più alto di sicurezza e decentralizzazione, ma presenta costi di gas più elevati e tempi di conferma maggiori;

- **Polygon** e **Avalanche** offrono costi nettamente inferiori e una maggiore rapidità di esecuzione grazie a un'architettura più scalabile;
- la presenza dell'algoritmo di *chain selection* di ZONIA consente di bilanciare in modo dinamico le esigenze di sicurezza, efficienza economica e rapidità, scegliendo la rete più adatta in funzione del tipo di operazione.

Queste osservazioni dimostrano come l'approccio multi-chain di ZONIA rappresenti un punto di forza strategico per la sostenibilità del sistema. La possibilità di spostare le fasi più onerose o intensive su Blockchain a basso costo, consente di ottimizzare sia le prestazioni sia i costi operativi.

Sviluppi Futuri

Il lavoro apre inoltre diverse prospettive future. Tra cui:

- L'estensione dell'applicazione a più Blockchain in ambiente di produzione, integrando wallet e reti mainnet.
- L'introduzione di metriche di valutazione automatica per la selezione dinamica della Blockchain target.

Bibliografia

[1] https://www.ibm.com/it-it/topics/internet-of-things

[2] GIGLI, Lorenzo, et al. ZONIA: a zero-trust oracle system for blockchain IoT applications. IEEE Internet of Things Journal, 2025.

[3] DI PIERRO, Massimo. What is the blockchain?. Computing in Science & Engineering, 2017, 19.5: 92-95.

[4] RODECK, David; CURRY, Benjamin. What is blockchain. Forbes, 2022.

[5] https://iocoltivoitalia.org/

[6] https://blockchainitalia.io/it/blockchain-italia-partner-del-progetto-cofam-per-la-certificazione-digitale-dei-servizi-ecosistemici/

[7] http://msecproject.eu/about/

[8] https://www.netservice.eu/en/research-and-development/abata

[9] REYNA, Ana, et al. On blockchain and its integration with IoT. Challenges and opportunities. Future generation computer systems, 2018, 88: 173-190.

[10] BENIICHE, Abdeljalil. A study of blockchain oracles. arXiv preprint arXiv:2004.07140, 2020.

[11] KANANI, Jaynti; NAILWAL, Sandeep; ARJUN, Anurag. Matic whitepaper. Polygon, Bengaluru, India, Tech. Rep, 2021.

[12] MONTORI, Federico; GIGLI, Lorenzo; ZYRIANOFF, Ivan; AGUZZI, Cristiano; ARTO, Manuel. D4. 'FINAL SOFTWARE DELIVERY, VALIDATION, BUSINESS MODEL, AND IMPACT ASSESSMENT. 2025

[13] https://beaconcha.in/