Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Studio e implementazione di smart contract per la piattaforma Ethereum in ambito contact tracing

Tesi di laurea triennale

Relatore

Prof.Lamberto Ballan

Laureando

Matteo Infantino

Anno Accademico 2019-2020

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

— Oscar Wilde

Dedicato a ...

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Pinco Pallino presso l'azienda Azienda S.p.A. Gli obbiettivi da raggiungere erano molteplici.
In primo luogo era richiesto lo sviluppo di ... In secondo luogo era richiesta l'implementazione di un ... Tale framework permette di registrare gli eventi di un controllore programmabile, quali segnali applicati Terzo ed ultimo obbiettivo era l'integrazione ...

"Life is really simple, but we insist on making it complicated"

— Confucius

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. NomeDelProfessore, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per il sostegno, il grande aiuto e per essermi stati vicini in ogni momento durante gli anni di studio.

Ho desiderio di ringraziare poi i miei amici per tutti i bellissimi anni passati insieme e le mille avventure vissute.

Padova, Luglio 2020 Matteo Infantino

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Introduzione al contesto applicativo.

Esempio di utilizzo di un termine nel glossario
Application Program Interface (API).

Esempio di citazione in linea
site:agile-manifesto.

Esempio di citazione nel pie' di pagina
citazione[1]

1.1 L'azienda

Sync Lab S.R.L. è una società fondata nel 2002 a Napoli come software house e diventata rapidamente un'azienda di consulenza nel dominio dell'Information and Communication Technology(ICT). Oggi Sync Lab ha raggiunto un'ampia diffusione sul territorio attraverso le sue cinque sedi: Napoli, Roma, Milano, Padova e Verona. L'organico aziendale è andato aumentando in modo continuo e rapido, in relazione all'apertura delle varie sedi ed alla progressiva crescita delle stesse, raggiungendo le cento collaborazioni nel 2007 e superando le duecento nel 2016. Con l'aiuto dei suoi specialisti che lavorano continuamente in maniera sincronizzata, collaborativa e disciplinata, Sync Lab propone ai suoi clienti un'ampia gamma di prodotti nei settori mobile, videosorveglianza e sicurezza delle infrastrutture informatiche aziendali.
Le politiche di assunzione hanno reso Sync Lab un punto di riferimento per coloro che intendono avviare o far evolvere in chiave professionale la loro carriera: l'alto tasso di assunzione post-stage ed il basso turn-over testimoniano la voglia di condividere il progetto comune, assumendo ruoli e responsabilità che possono essere offerti solo da un processo evolutivo così intenso.

[1] **womak:lean-thinking**.

Figura 1.1: Logo Sync Lab

1.1.1 Servizi offerti

La principale attività di Sync Lab è la consulenza tecnologica, un processo continuo di identificazione e messa in opera di soluzioni su misura, finalizzate alla creazione di valore. I principali servizi che fornisce l'azienda sono:

* Business Consultancy;
* Project Financing;
* IT Consultancy.

L'offerta di consulenza specialistica trova le punte di eccellenza nella progettazione di architetture software avanzate, siano esse per applicativi di dominio, per sistemi di Supporto al Business (BSS), per sistemi di integrazione (EAI/SOA) o per sistemi di monitoraggio applicativo/territoriale. Il laboratorio Ricerca e Sviluppo (RD) dell'azienda è sempre al passo con i nuovi paradigmi tecnologici e di comunicazione, ad esempio Big Data, Cloud Computing, Internet of Things (IoT), Mobile e Sicurezza IT, per supportare i propri clienti nella creazione ed integrazione di applicazioni, processi e dispositivi. Le attività in ambito Educational ed RD hanno permesso di acquisire una profonda conoscenza degli strumenti di finanza agevolata fruendone direttamente ed interagendo con enti di supporto ai progetti innovativi dei propri clienti. L'azienda, grazie alla rete di relazioni a livello nazionale ed internazionale, ha ottenuto importanti finanziamenti in progetti RD europei (FP7 e H2020).

1.1.2 Prodotti offerti

Dalla sua creazione fin ad oggi, Sync Lab ha sviluppato diversi prodotti garantendone sempre la qualità grazie alle certicazioni ISO 9001, ISO 14001, ISO 27001, OHSAS 18001.

I prodotti offerti sono i seguenti:

* **SynClinic:** rappresenta il sistema informativo sanitario per la gestione integrata di tutti i processi clinici e amministrativi ospedalieri, cliniche e case di cura. Utilizzabile sia in sia on premises, gestisce, organizza e monitora tutte le fasi del percorso di cura del paziente, diventando supporto indispensabile per il personale nella gestione del rischio clinico;
* **DPS 4.0:** rappresenta una soluzione web per una compliance continua che fa uso della GDPRg Privacy acronimo inglese General Data Protection Regulation che

è un regolamento attraverso il quale la Commissione Europea intende rafforzare la protezione dei dati personali di cittadini dell'Unione Europea ;

* **StreamLog:** rappresenta un sistema finalizzato al soddisfacimento dei requisiti fissati dal Garante, ovvero è in grado di effettuare il controllo degli accessi degli utenti ai sistemi in modo semplice ed efficace. Il sistema è basato su framework open source allo stato dell'arte e, in particolare, su un'innovativa tecnologia di "streaming", frutto del laboratorio di Ricerca e Sviluppo Sync Lab;
* **StreamCrusher:** tecnologia che aiuta ad essere ben informati su quando bisogna prendere decisioni di business, a identificare velocemente criticità e a riorganizzare i processi in base a nuove esigenze;
* **Wave:** nato dal laboratorio di Ricerca e Sviluppo Sync Lab, si propone come integrazione sinergica tra i mondi della Videosorveglianza e quello dei Sistemi Informativi

L'approfondita conoscenza di processi e tecnologie, maturata in esperienze altamente significative e qualificanti, fornisce l'expertise e il know-how necessari per gestire progetti di elevata complessità, dominando l'intero ciclo di vita:

* Studio di fattibilità;
* Progettazione;
* Implementazione;
* Governance;
* Post Delivery.

1.1.3 Settori di impiego

Sync Lab si sta sempre più specializzando in vari settori d'impiego: dal mondo banking all'assurance con una nicchia importante nell'ambito sanità in cui vanta un prodotto d'eccellenza per la gestione delle cliniche private. L'azienda inoltre ha recentemente fondato un reparto collegato Sync Security che si occupa del mondo della cyber security e sicurezza informatica in genere.

1.2 Organizzazione del testo

Il secondo capitolo descrive ...

Il terzo capitolo approfondisce ...

Il quarto capitolo approfondisce ...

Il quinto capitolo approfondisce ...

Il sesto capitolo approfondisce ...

Nel settimo capitolo descrive ...

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

* gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
* per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *parola*[g];
* i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

In questo capitolo si introduce il contesto in cui lo stage viene svolto e viene fornita una panoramica del progetto. Successivamente si descrivono le principali tecnologie utilizzate per raggiungere gli obiettivi prefissati. Inoltre sono analizzati i vincoli, gli obiettivi e la pianificazione dello stage

2.1 Introduzione al progetto

L'obiettivo di SyncLab è sviluppare un'applicazione per il contact tracing. Il software deve poter registrare i contatti tra le persone sfruttando il bluetooth LE. Le informazioni da raccogliere riguardano il tempo e la distanza di contatto con gli altri dispositivi, assicurando il rispetto della privacy. In questo modo quando una persona risulta infetta è possibile notificarlo a tutti i suoi contatti.

L'azienda ha incaricato diversi stagisti per sviluppare un'applicazione mobile e una web application che gestiscano il contact tracing, la prima lato utente generico, la seconda lato utente sanitario. Lo scopo della applicazione mobile è quello di registrare i contatti, mentre lo scopo della webapp è fornire l'autorità al personale medico di segnalare una persona risultata positiva a un tampone, sempre dietro conferma del malato.

Questo è il quadro generico dove si inserisce il mio percorso di stage. Infatti mi è stato proposto di studiare e implementare uno smart contract per registrare i contatti nella piattaforma ethereum. Lo scopo del mio percorso si può sintetizzare nei seguenti punti:

* Studio tecnologico delle blockchain (in particolare ethereum);
* Analisi e sviluppo di uno smart contract per il contact tracing in linguaggio solidity;
* Studio e utilizzo dello smart contract in ambiente mobile (Android).

2.2 Studio tecnologico

Lo studio richiesto per lo svolgimento dello stage parte dalla tecnologia blockchain in tutti i suoi aspetti, per poi passare nello studio specifico di ethereum, solidity e

tutti gli strumenti di supporto per lo sviluppo e il testing di smart contract. Infine è necessario uno studio di ethereum applicato ad un ambiente mobile.

2.2.1 Blockchain

Una blockchain è un registro pubblico, distribuito e decentralizzato. Può essere utilie immaginarlo come un grande database distribuito su molti nodi, in continua espansione. Il registro è strutturato come una catena di blocchi contenenti le transazioni. Le principali caratteristiche di una blockchain sono l'immutabilità, la trasparenza, la tracciabilità delle transazioni e la sicurezza.

2.2.1.1 Permissionless e permissioned

Le blockchain si dividono in due categorie: blockchain permissionless (o pubbliche) e blockchain permissioned (o private). Le prime permettono l'accesso a qualsiasi utente che decida di connettersi e partecipare, generando nuove transazioni, effettuando il compito di miner o semplicemente leggendo il registro delle transazioni memorizzate. Bitcoin ed ethereum sono esempi di blockchain pubbliche. Le seconde invece permettono l'accesso solo ad utenti autorizzati ed autenticati poiché la Blockchain in questione opera esclusivamente entro i limiti di una comunità ben definita dove tutti i partecipanti sono noti. Solitamente a capo di questi sistemi si trovano istituti finanziari o agenzie governative che definiscono chi possa accedere o meno alla rete. Questo vuol dire che tutti i miner sono considerati fidati.

2.2.1.2 Nodi, transazioni, blocchi, ledger

Per comprendere al meglio il funzionamento di una blockchain è utile spiegare i componenti basilari da cui è formata:

* Nodi: sono i partecipanti alla blockchain. Qualsiasi dispositivo che si connette alla blockchain è considerato un nodo, ma ne esistono di diversi tipi in base al ruolo che hanno nella rete.

* Transazione: è costituita dai dati che rappresentano i valori oggetto di scambio e che necessitano di essere verificati, approvati e poi archiviati.

* Blocco: è rappresentato dal raggruppamento di un insieme di transazioni che sono unite per essere verificate, approvate e poi archiviate dai partecipanti alla blockchain.

* Ledger: il registro pubblico nel quale vengono annotate con la massima trasparenza e in modo immutabile tutte le transazioni effettuate in modo ordinato e sequenziale. Il Ledger è costituito dall'insieme dei blocchi che sono tra loro incatenati tramite una funzione di crittografia e grazie all'uso di hash.

2.2.1.3 Generali bizantini

La blockchain nasce per eliminare il bisogno di una terza parte per validare una transazione. Per farlo è necessario che tutti i nodi siano concordi sulla validità e l'ordine delle transazioni. È dunque fondamentale introdurre un algoritmo di consenso per validare i blocchi e quindi le transazioni. Per capire a pieno questo concetto è ragionevole introdurre il problema matematico dei generali bizantini.

Diversi generali, durante un assedio, sono sul punto di attaccare una città nemica. Essi sono dislocati in diverse aree strategiche e possono comunicare solo mediante messaggeri al fine di coordinare l'attacco decisivo. Il problema è che tutti sanno che tra di loro si trovano uno o più traditori. Come può ciascun luogotenente avere la certezza che una volta ricevuto l'ordine di attacco, questo sia stato inviato anche agli altri? Dunque come può essere certo di non essere l'unico ad averlo ricevuto in quella forma e dunque che tutti gli altri luogotenenti siano nella condizione di trasmettere e condividere lo stesso ordine? Come è possibile garantire ai generali e a tutti i luogotenenti che nessuno cerchi di minare il piano? Lo scopo del problema è far sì che ai luogotenenti onesti arrivi il piano d'attacco corretto, senza che i traditori possano compromettere l'operazione facendo arrivare loro le informazioni sbagliate. In una blockchain, la soluzione è quella di non avere più un generale che comanda sugli altri. Dunque non c'è più un centro che prevale gerarchicamente. Ma si assegna la stessa gerarchia a tutti i partecipanti. Tutti i generali e tutti i luogotenenti, ovvero tutti i nodi, che partecipano a questo modello, concordano ogni singolo messaggio trasmesso tra i nodi, lo vedono e lo condividono. Tutti devono concordare ogni messaggio che viene trasmesso nella rete. Tutti vedono qualsiasi messaggio e qualsiasi modifica. Di conseguenza la blockchain risulta immutabile, immodificabile incorruttibile.

2.2.1.4 Algoritmi di consenso

L'obiettivo di un algoritmo di consenso è ottenere la validazione delle transazioni in una blockchain. Questo si traduce in un comune accordo sullo stato della blockchain tra i nodi partecipanti. Ogni transazione deve essere registrata nel ledger e l'algoritmo di consenso assicura che non ci siano state transazioni maligne o corrotte. Esistono diversi tipi di algoritmi, ma mi limiterò a spiegare i due principali, su cui si fonda la quasi totalità delle blockchain.

L'algoritmo proof of work si basa sulla generazione di blocchi attraverso complessi problemi matematici, la cui risoluzione richiede un grande sforzo computazionale e un grosso dispendio di tempo ed energia. Questa metodologia comporta la ricerca di un valore che, una volta sottoposto ad una funzione hash, restituisca un ulteriore hash che inizia con un certo numero di bit a zero. Il lavoro medio richiesto è esponenzialmente proporzionale al numero di bit a zero richiesti e può essere verificato immediatamente. Si adotta un valore hash con un numero selezionabile di bit a zero al fine di non semplificare la sua generazione, dal momento che l'avanzamento tecnologico offre calcolatori di potenza crescente. Il nodo che riesce a trovare il valore hash corretto prima degli altri crea il blocco: esso verrà aggiunto in coda alla blockchain e riceverà un compenso sotto forma di valuta virtuale, chiamata cripto-valuta. L'algoritmo Proof-of-Work non risolve in modo ottimo la Byzantine Fault Tolerance ma offre al contempo una delle implementazioni più sicure ed affidabili per le reti blockchain.

Il proof of stake cambia paradigma: i blocchi vengono validati da chi possiede più token nella blockchain. Ogni account ha una possibilità proporzionale al proprio saldo di generare un blocco valido. Rispetto al proof of work, questo approccio presenta molti vantaggi. POW, infatti, richiede un'enorme quantità di tempo ed energia, che comporta un limitato numero di transazioni al secondo e un costo elevato da parte dell'utente per ottenere la validazione della propria transazione. Con POS si ottiene maggiore scalabilità e minore costo di tasse. Inoltre la struttura della blockchain la rende più sicura. Consideriamo l'attacco del 51 percento: se un miner detiene la

maggioranza dello stake non ha nel suo interesse un attacco alla rete perché se il valore della criptovaluta crolla, crollerebbero anche tutti i suoi averi. Per questo i maggiori proprietari di tokens avranno nel loro interesse il mantenimento di una rete sicura.

Il proof of work è stato il primo algoritmo utilizzato nelle blockchain ed è tutt'ora il più diffuso. Negli ultimi anni però stanno nascendo molte blockchain che si basano su proof of stake e persino ethereum entro la fine del 2020 inizierà il passaggio al POS, che si dovrebbe completare nel 2022.

2.2.2 Ethereum

Ethereum è una piattaforma decentralizzata creata nel 2015 da Vitalik Buterik e che ha riscosso fin da subito grande successo. In ethereum vengono eseguiti programmi chiamati smart contracts. Anche Bitcoin ha un linguaggio che gli permette di la creazione di smart contract, ma a differenza della famosa blockchain, Ethereum permette la scrittura di contratti turing-completi. Negli smart contracts è possibile programmare come in qualsiasi altro linguaggio di programmazione, seppur con delle differenze di cui discuteremo più avanti.

2.2.2.1 Account, transazioni e gas

2.2.2.1.1 Account In Ethereum ogni partecipante alla blockchain possiede un account con cui interagire nella rete. Anche gli smart contracts hanno un account, fondamentale per raggiungerli e interagire con loro. Gli account hanno un *balance* in Ether, la cryptovaluta di Ethereum, che può essere modificato da transazioni inviate o ricevute.

2.2.2.1.2 Transazioni Una transazione è composta dall'account target, l'account sender, il valore trasferito in Ether, campi dati opzionali, il limite di gas che la transazione può consumare e il prezzo del gas.

2.2.2.1.3 Gas Ogni transazione consuma un certo numero di gas, che rappresenta il carburante della blockchain. Ogni operazione negli smart contract ha un costo in gas che deve essere pagato come tassa per far sì che venga eseguita in Ethereum.

2.2.2.2 Solidity

Solidity è un linguaggio tipato che permette di scrivere smart contracts. I contracts a prima vista ricordano molto le classi nella programmazione orientata a ogggetti, ma in solidity bisogna prestare particolare attenzione alla sicurezza, in quanto gli smart contracts gestiscono soldi e transazioni. Se sono presenti bug o falle di sicurezza le conseguenze potrebbero essere gravi. Inoltre è sempre opportuno tenere a mente che le operazioni non sono gratuite, ma vengono pagate con delle tasse in base al tipo di operazione richiesta. L'efficienza quindi, è cruciale nel definire smart contracts e non sempre le best practices in solidity corrispondono alla soluzione che può sembrare migliore in un altro linguaggio di programmazione.

2.2.2.3 Truffle

Truffle è una suite di supporto per la programmazione in solidity che ha l'obiettivo di rendere la vita dello sviluppatore più semplice. Fornisce un ambiente di sviluppo e un framework per testare i contratti usando una EVM (Ethereum Virtual Machine).

2.2.2.4 Web3js e web3j

Tra gli obiettivi del mio stage c'è l'integrazione dello smart contract sviluppato con un ambiente mobile e una web application. Web3.js e web3j servono proprio per raggiungere questo scopo. Sono librerie, la prima per JavaScript e la seconda per Java, che forniscono il supporto necessario a interagire con il contratto.

2.3 Analisi dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Difficoltà nell'effettuare la maggior parte dello stage in remoto
Descrizione: A causa dell'emergenza sanitaria causata dal COVID-19, gran parte dello stage verrà effettuato da remoto.
Soluzione: Pianificazione del lavoro dettagliata; frequenti aggiornamenti con tutor aziendale; utilizzo di strumenti di supporto.

2. Scarsa conoscenza delle tecnologie previste per lo svolgimento dello stage
Descrizione: Le blockchain, ethereum e lo sviluppo di smart contracts rappresentano un campo non affrontato durante il percorso di studi e poco conosciuto dal sottoscritto.
Soluzione: Dedicare molto tempo allo studio approfondito di tutte le tecnologie in gioco per essere preparato a sufficienza al momento dello sviluppo.

3. Difficoltà nell'integrazione di ethereum con l'ambiente mobile
Descrizione: Il tutor aziendale mi ha avvisato preventivamente della sua scarsa conoscenza in questo ambito e delle possibilti difficoltà che potrebbero emergere.
Soluzione: In fase di studio tecnologico, dedicare diverse ore ad approfondire questo aspetto.

4. Tempo limitato per l'integrazione finale con il prodotto sviluppato dall'azienda
Descrizione: Come riportato nel primo rischio, la maggior parte dello stage sarà effettuato da remoto. Il tempo a disposizione con gli altri sviluppatori sarà poco e deve essere sufficiente per portare a termine l'integrazione della blockchain nel progetto.
Soluzione: Otiimizzare il tempo con gli altri sviluppatori in azienda e se necessario comunicare con loro e organizzare call durante il lavoro svolto da casa.

2.4 Vincoli e obiettivi

2.4.1 Vincoli temporali

Lo stage si svolge in un periodo di 8 settimane lavorative per 8 ore al giorno. Visto il contesto lavorativo da remoto mi è stato richiesto di compilare quotidianamento un registro delle attività che consentisse al tutor aziendale di verificare il mio lavoro. Verso la fine dello stage mi è stato consentito di lavorare alcuni giorni in azienda, dalle 9:00 alle 18:00. A prescindere dalla modalità di lavoro ho avuto delle scadenze settimanali da rispettare in base alla pianificazione. Infatti, il piano di lavoro presenta

un calendario delle attività diviso per settimane che mi ha consentito di tenere sotto controllo lo stato di avanzamento del mio stage.

2.4.2 Vincoli metodologici

Per favorire il lavoro da remoto, il tutor aziendale mi ha fornito un piano delle attività presente su trello. Inoltre ogni giorno ho registrato le mie attività in un documento condiviso con il tutor e quasi quotidianamente sono state effettuate call per essere in costante aggiornamento.

2.4.3 Vincoli tecnologici

I vincoli tecnologici che ho dovuto rigorosamente rispettare sono stati:

* Ethereum e solidity per lo sviluppo dello smart contract
* Android per l'integrazione del contract in ambiente mobile
* Gitlab per la condivisione del mio lavoro

Per il resto mi è stata concessa parecchia libertà riguardo agli strumenti di sviluppo e l'uso di tecnologie. Questo ha avuto dei pro e dei contro. Inizialmente ho dovuro dedicare parecchio tempo allo studio ed è stato difficile capire quali fossero le scelte migliori. A posteriori è stato molto utile e formativo, anche se alcune cose non sono state utilizzate nell'implementazione finale.

2.4.4 Obiettivi

Si prevede lo svolgimento dei seguenti obiettivi:

Tabella 2.1: Tabella degli obiettivi obbligatori

Obiettivo	**Descrizione**
O01	Acquisizione competenze sulle tecnologie Blockchain, in particolare Ethereum
O02	Capacità di progettazione e analisi di smart contract
O03	Capacità di raggiungere gli obiettivi richiesti in autonomia, seguendo il programma preventivato)
O04	Portare a termine l'implementazione di almeno l'80% degli sviluppi previsti

Tabella 2.2: Tabella degli obiettivi desiderabili

Obiettivo	Descrizione
D01	Portare a termine il lavoro di studio della portabilità di Ethereum su dispositivi mobili
D02	Portare a termine l'implementazione completa degli sviluppi previsti

Tabella 2.3: Tabella degli obiettivi facoltativi

Obiettivo	Descrizione
F01	Completare l'installazione di un peer su un dispositivo mobile Android

2.5 Pianificazione

* **Prima Settimana (40 ore)**
  - Incontro con persone coinvolte nel progetto per discutere i requisiti e le richieste relativamente al sistema da sviluppare;
  - Studio delle diverse tipologie di catene Block Chain (permissioned/unpermissioned/public/private);
  - Studio del funzionamento delle blockchain (diverse tipologie del proof).
* **Seconda Settimana (40 ore)**
  - Studio di Ethereum, installazione peer e configurazione;
  - Ripasso Javascript;
  - Studio linguaggio Solidity.
* **Terza Settimana (40 ore)**

  - Studio linguaggio Solidity.

* **Quarta Settimana (40 ore)**

  - Analisi caso d'uso: creazione smart contract per inserire in catena informazioni sul tracing delle persone;
  - Implementazione e testing del codice di smart contract per l'inserimento in catena delle informazioni raccolte sul tracing.

* **Quinta Settimana (40 ore)**

  - Analisi Caso d'uso: creazione smart contract per recuperare da catena le informazioni sul tracing delle persone;
  - Implementazione e testing del codice di smart contract per il recupero delle informazioni raccolte sul tracing.

* **Sesta Settimana (40 ore)**

  - Studio Installazione peer Ethereum in ambienti mobile.

* **Settima Settimana (40 ore)**

  - Studio e prototipo di installazione del peer Ethereum in ambienti mobile.

* **Ottava Settimana (40 ore)**

  - Installazioni finali e test;
  - Stesura tesina.

Capitolo 3

Analisi dei requisiti

Breve introduzione al capitolo

3.1 Casi d'uso

All'inizio dello stage sono stati discussi i casi d'uso del software da sviluppare con il tutor aziendale e gli altri stagisti partecipanti al progetto. L'azienda intende sviluppare un'applicazione mobile per gestire il tracciamento dei contatti e una web application per permettere al personale sanitario di segnalare una persona risultata positiva a un tampone, sempre dietro conferma dell'infetto. Il mio ruolo nel progetto è stato integrare uno smart contract con le due parti, rispettando i casi d'uso e i requisiti pensati.
Tuttavia la blockchain non è visibile da chi utilizza l'applicazione e non richiede un'interazione con l'utente. Per questo motivo, per quanto riguarda il mio stage, non sono presenti casi d'uso, ma solamente i requisiti che lo smart contract deve soddisfare.

3.2 Tracciamento dei requisiti

Da un'attenta analisi dei requisiti effettuata sul progetto è stata stilata la tabella che traccia i requisiti.
Sono stati individuati diversi tipi di requisiti e per distinguerli si è fatto utilizzo di un codice identificativo.
Il codice dei requisiti è così strutturato R(F/Q/V)(N/D/O) dove:

R = requisito

F = funzionale

Q = qualitativo

V = di vincolo

N = obbligatorio (necessario)

D = desiderabile

Z = opzionale

Nelle tabelle 3.1, 3.2 e 3.3 sono riassunti i requisiti e il loro tracciamento con gli use case delineati in fase di analisi.

Tabella 3.1: Tabella del tracciamento dei requisiti funzionali

Requisito	**Descrizione**
RFN-1	L'address del proprietario del contract deve essere salvato
RFN-2	Una persona deve essere inserita in catena con un id univoco
RFN-3	Un inserimento deve essere rifiutato se è l'id della persona è già presente in catena
RFN-4	Una persona infetta deve essere segnalata in blockchain
RFN-5	Una segnalazione di infezione deve essere rifiutata se non è stata invocata da un medico autorizzato
RFN-6	Una segnalazione di infezione deve essere rifiutata se viene inserito un id non presente in blockchain
RFN-7	Quando viene rilevato un contatto tra due persone deve essere inserito in blockchain
RFN-8	L'inserimento di un contatto viene rifiutato se viene effettuato da un indirizzo non corrispondente a quello della persona il cui contatto viene inserito
RFN-9	L'inserimento di un contatto viene rifiutato se gli id non sono presenti in blockchain
RFN-10	Deve essere possibile ottenere i contatti di una persona negli ultimi 14 giorni
RFN-11	I contatti di una persona non devono essere forniti se vengono richiesti da qualcun altro

Requisito	Descrizione
RFN-12	Deve essere posssibile stabilire se una persona ha avuto contatti ritenuti a rischio
RFN-13	Un rischio di contagio deve essere confermato anche dai contatti della persona infetta
RFN-14	L'informazione sulla presenza di contatti a rischio deve essere effettuata solo dal proprietario dell'informazione
RFN-15	Deve essere possibile calcolare la somma degli indici di contatto di una persona negli ultimi gionri
RFN-16	La somma degli indici di contatto deve essere effettuata solo dal proprietario dell'informazione

Tabella 3.2: Tabella del tracciamento dei requisiti qualitativi

Requisito	Descrizione
RQN-1	Lo smart contract deve essere il più possibile ottimizzato per limitare i consumi di gas
RQN-2	Il codice deve essere versionato e reso disponibile nel repository aziendale
RQN-3	Deve essere fornito un documento tecnico

Tabella 3.3: Tabella del tracciamento dei requisiti di vincolo

Requisito	**Descrizione**
RVN-1	Utilizzo del linguaggio solidity e piattaforma ethereum per lo sviluppo dello smart contract
RVD-2	Installazione dello smart contract in ambiente mobille (Android)

Capitolo 4

Progettazione e implementazione

Breve introduzione al capitolo

4.1 Prima versione smart contract

La prima implementazione dello smart contract prevede l'utilizzo di due struct: Person e Contact

```
    struct Person {
        string id;
        address owner;
        bool infected;
    }
```

```
    struct Contact{
        Person p1;
        Person p2;
        uint256 contactRate;
        uint256 timestamp;
    }
```

I campi dati utilizzati dal contract sono i seguenti:

```
contract Tracing {

    address ownerAddress;
    Person[] people;
    Contact[] contacts;
```

Al momento del deploy del contratto, ownerContract viene inizializzato con l'address appartenente a chi effettua il deploy. I due array people e contacts, invece, vengono popolati in seguito. People contiene tutte le persone che utilizzano lo smart contract, mentre contacts contiene tutti i contatti registrati tra due persone.
Non entrerò nel dettaglio del funzionamento di questo contract in quanto è stato scartato per favorire un'altra soluzione. Le ragioni risiedono in alcune criticità di

questa implementazione. Avere una struttura dati che contiene tutti i contatti di tutti gli utenti non è una soluzione particolarmente furba. Per prima cosa complica inutilmente il contract per effettuare qualsiasi tipo di operazione sui contatti. In secondo luogo per il suo funzionamente sono richieste operazioni particolarmente onerose per quanto riguarda il consumo di gas, come il confronto tra stringhe. Infatti in solidity non è supportato il confronto booleano tra due stringhe. Per bypassare questo problema è necessario calcolare l'hash delle due stringhe e poi confrontarlo. Il calcolo dell'hash, però, è una delle operazioni più onerose in solidity.
Nella sezione analisi gas approfondirò l'argomento, mostrando i consumi del contract v1 a confronto con la soluzione definitiva.

4.2 Seconda versione smart contract

Come per la precedente soluzione sono stati adottati due struct, Person e Contact, ma con alcune differenze

```
    struct Person {
        address owner;
        bool infected;
    }
```

```
    struct Contact {
        string idContact;
        uint256 contactRate;
        uint256 timestamp;
    }
```

Person è una struct che ha come campi dati un address owner e un booleano infected. Owner rappresenta l'address della persona. Infected invece, rappresenta lo stato di salute. Contact è una struct che ha come campi dati una stringa idContact, un intero contactRate e un intero timestamp. IdContact rappresenta l'id della persona con cui si è entrati in contatto, contactRate è un indice di contatto tra le due persone e timestamp contiene l'unix time del momento del contatto.

I campi dati sono i seguenti:

```
contract Tracing {

    address ownerAddress;
    mapping(string => Person) mapIdToPerson;
    mapping(string => Contact[]) mapIdToContact;
```

Lo smart contract possiede un campo address ownerContract che rappresenta il proprietario del contratto. Al momento del deploy ownerContract viene inizializzato all'address appartenente a chi effettua il deploy. mapIdToPerson associa un id ad una persona. La mappa contiene tutte le persone che utilizzano il contract. mapIdToContact associa un id ad un array di contatti. Ogni array rappresenta i contatti della persona con il corrispondente id.

Le funzioni disposibili per un uso esterno nel contract sono le seguenti:

```
/*
```

```
Add a person in mapIdToPerson
If the person is already in, returns an error
*/
function addPerson(string calladata _id, address _owner) external
    ;

/*
Change the state of infected for the person of the input id
Returns an error if the id isn't registered or if there aren't
    the permission to call the function: only the owner of the
    contract can call this function.
*/
function setInfected(string calladata _id,bool _infected)
    external;

/*
Return the infected state of a person
Returns an error if the call is from a user without permission or
     if the id isn't registered
*/
function getInfected(string calladata _myId,string calladata
    _idContact) external view isOwner(_myId) returns (bool);

/*
Adds a contact in mapIdToContacts
Returns an error if one of the ids doesn't exists
*/
function addContact(string calladata _myId,string calladata
    _idContact,uint _contactRate) external isOwner(_myId);

/*
Returns an array of id contacts of the person.
*/
function getContactsById(string calladata _id) external view
    returns(string[] memory);

/*
Check if in the contacts of a person there are people infected
Returns an error if the call is from a user without permission
*/
function hasIdHadRiskyContacts(string calladata _id,uint
    _contactRate,uint _days) external view isOwner(_id) returns (
    bool);

/*
Sums a person contactRates in the last days
Returns an error if the call is from a user without permission
*/
function sumContactRate(string calladata _id,uint256 _days)
    external view isOwner(_id) returns (uint256);
```

4.3 Analisi gas

In ethereum ogni operazione effettuata che cambia lo stato della blockchain consuma gas. Il gas è un'unità di misura utilizzata per pagare la computazione effettuata dai miners. Più un operazione è dispendiosa a livello di risorse, più gas questa operazione costerà. Il concetto di gas permette di addebitare una tariffa che viene pagata ai miners, incentivandoli a prendere parte attiva nel sistema ethereum. Tuttavia, il gas rappresenta un cambio di paradigma rispetto alla programmazione classica. Non sempre un approccio vantaggioso in un normale linguaggio di programmazione rappresenta la best practice in solidity, proprio perché c'è una variabile in più da considerare. Nel corso dello stage è stata rivolta particolare attenzione a questo aspetto: sono state portate modifiche molto importanti al contract definitivo per ottimizzare il più possibile il consumo di gas.

Nei paragrafi precedenti ho mostrato la prima implementazione scartata per un eccessivo consumo di gas. Qui sotto riporto i costi relativi alle due versioni.

Figura 4.1: Primo smart contract

Solc version: 0.6.7+commit.b8d736ae	Optimizer enabled: true	Runs: 1500	Block limit: 6721975 gas	
Methods	41 gwei/gas		214.63 eur/eth	
Contract	Method	Min	Max	Avg	# calls	eur (avg)
Migrations	setCompleted	-	-	27254	2	0.24
Tracing	addContact	199566	211405	203512	3	1.79
Tracing	addPerson	75617	87455	79367	4	0.70
Tracing	setInfected	40524	52850	50111	9	0.44
Deployments					% of limit	
Migrations		-	-	137638	2 %	1.21
Tracing		-	-	1599579	23.8 %	14.08

Figura 4.2: Secondo smart contract

Solc version: 0.6.7+commit.b8d736ae	Optimizer enabled: true	Runs: 1500	Block limit: 6721975 gas	
Methods	26 gwei/gas		209.78 eur/eth	
Contract	Method	Min	Max	Avg	# calls	eur (avg)
Migrations	setCompleted	-	-	27254	2	0.15
Tracing	addContact	-	-	107741	3	0.59
Tracing	addPerson	-	-	44898	4	0.24
Tracing	setInfected	-	-	30369	7	0.17
Deployments					% of limit	
Migrations		-	-	137638	2 %	0.75
Tracing		-	-	1673623	24.9 %	9.13

Il costo del deploy del deploy dei due smart contracts è abbastanza simile: intorno a 1.500.000 gas. Questo però non rappresenta il costo maggiore per quanto riguarda

l'applicazione di contact tracing. Il deploy infatti, viene effettuato una singola volta e il suo costo rimane marginale. La parte più onerosa è certamente il consumo che riguarda le funzioni addPerson, addContact e setInfected. Si può notare che per ogni chiamata di queste funzioni il consumo di gas è all'incirca dimezzato.

* addContact: 200k prima, 100k dopo
* addPerson: 80k prima, 45k dopo
* setInfected: 50k prima, 30k dopo

4.3.1 Fattibilità applicazione reale

Anche con la massima efficienza di uno smart contract, è fondamentale considerare il caso d'uso dell'applicazione per rendersi conto della fattibilità del progetto. Come accennato prima, i costi maggiori in ethereum non sono rappresentati dal deploy del contratto, bensì dal numero di chiamate stimate alle funzioni che consumano gas. Nel nostro caso d'uso la funzione addPerson viene eseguita ogni qualvolta una persona installa l'applicazione nel proprio smartphone. Considerando metà della popolazione italiana questo vuol dire 30 milioni di chiamate. Prendendo come esempio di prezzo per gas 26 gwei, ogni chiamata costa circa 24 centesimi. Questo vuol dire che sono per l'installazione il costo stimato è di 7,5 milioni di euro. Già questo piccolo calcolo sarebbe sufficiente per rendersi conto che questo smart contract non è sostenibile in un contesto di utilizzo reale. Purtroppo però, c'è una spesa ancora maggiore. In blockchain viene inserito ogni contatto superiore a 15 minuti a distanza inferiore di 2 metri, per ogni persona che ha l'applicazione. Facendo una piccola stima, se consideriamo una media di 10 contatti di questo tipo a persona al giorno, con ogni chiamata di costo pari a 0,50 centesimi circa, si ottiene una spesa di ben 150 milioni di euro (al giorno!). Decisamente non sostenibile. Soluzioni di questo genere non sono realizzabili e questo smart contract rappresenta un lavoro puramente accademico, senza alcuna possibilità di utilizzo reale. Tuttavia ci sono delle soluzioni alternative.

4.3.2 Soluzioni

4.3.2.1 Smart contract semplificato

Come detto, in ethereum la soluzione proposta non è realizzabile nella pratica. Una possibile alternativa nell'ambito contact tracing è rappresentata dalla gestione locale di tutti i contatti tra le persone. Ogni dispositivo mobile possiede un id e i contatti con gli altri dispositivi (distanza inferiore a 2 metri per almeno 15 minuti) vengono registrati e salvati localmente. Ogni 14 giorni vengono eliminati perché considerati ininfluenti per il contagio. Quando una persona viene trovata infetta a seguito di un tampone, il medico (sotto autorizzazione del contagiato) lo segnala tramite web app. Quello che succede in seguito a questa segnalazione è che l'id del malato viene inserito nell'array degli infetti nello smart contract. Lato mobile periodicamente il dispositivo chiama una funzione dello smart contract che restituisce gli infetti degli ultimi 14 giorni. In seguito controlla se tra i suoi contatti è presente un infetto o meno. In questo modo lo smart contract è estremamente semplice e si occupa solo di conservare gli id dei malati in modo sicuro e immutabile. Tutte le altre operazioni sono effettuate localmente. Il consumo del gas è infinitamente più contenuto e permette un utilizzo reale dell'applicazione. Inoltre non sono presenti rallentamenti

dovuti ai tempi di validazione delle transazioni. Questo perché l'unica operazione che richiede consumo di gas, e dunque una transazione, è quella di inserimento dell'id nell'array, che viene effettuata dal medico solo per le persone malate a seguito di un tampone positivo. Questa operazione non è necessario che sia immediata, ma può senza problemi impiegare il tempo necessario per la validazione in una blockchain. Dal lato mobile infatti, il controllo di infetti presenti tra i contatti non è real-time, ma viene effettuato periodicamente. Le funzioni presenti in questo smart contract sono solo due: una per l'inserimento di un infetto, l'altra per ottenere la lista di infetti. Quest'ultima non richiede una transazione in quanto è una funzione di sola lettura.

Figura 4.3: Nuovo smart contract

Solc version: 0.6.7+commit.b8d736ae	Optimizer enabled: true	Runs: 1500	Block limit: 6721975 gas		
Methods					
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**eur (avg)**
Migrations	setCompleted	-	-	27254	2	-
Tracing	addInfected	69130	84130	76630	2	-
Deployments				**% of limit**		
Migrations		-	-	137638	2 %	-
Tracing		-	-	331348	4.9 %	-

Dalla tabella riportata si può notare come il costo di deploy del contratto sia notevolmente inferiore rispetto ai precedenti contratti: 300k gas contro 1500k circa. Ma la differenza la fa soprattutto il numero di chiamate alla funzione addInfected, decisamente inferiori rispetto a tutte le chiamate necessarie per registrare i contatti. In Italia i contagi certificati ad oggi sono circa 200k. Considerando un costo di circa 20 centesimi a chiamata per ogni infetto, supponendo anche che tutti gli infetti siano registrati all'applicazione e diano il consenso al proprio medico, la spesa totale dell'utilizzo del contratto si aggirerebbe intorno a 40k euro, in linea con le dApp più utilizzate.

4.3.2.2 Altre blockchain

Un'altra soluzione, che può essere anche complementare a quella appena descritta, prevede un cambio di piattaforma. Ethereum infatti, nonostante sia la blockchain più diffusa per lo sviluppo di applicazioni decentralizzate, ha un costo molto alto e destinato ad aumentare con l'aumentare degli utenti. Questo è dovuto all'algoritmo di consenso che ethereum, come la maggiorparte delle blockchain, utilizza attualmente, ossia il proof of work. La computazione richiesta per validare i blocchi e le transazioni richiedono il pagamento di una tassa, proporzionale alle risorse richieste. Tuttavia, esistono altri tipi di blockchain che utilizzano il consenso proof of stake, come la piattaforma EOS. La scalabilità e le transazioni gratuite sono i punti di forza di EOS. Il processo non è comunque gratuito, ma richiede che l'utente possieda un numero di token (stake) che gli permetta di "affittare" le risorse necessarie per validare le transazioni. Questi token però non sono spesi, ma possono essere restituiti quando si vuole al prezzo corrente della cryptovaluta. Anche ethereum sta effettuando il passaggio al proof of stake, con la piattaforma che verrà denominata ethereum 2.0, in arrivo a fine 2020. Il passaggio

definitivo, con la chiusura di ethereum 1.0 è previsto per il 2022. Ethereum 2.0 è molto atteso perché il cambio di algoritmo cambia completamente il modo di vedere le blockchain. La scalabilità è uno dei problemi maggiori delle blockchain che si basano su proof of work, limitandone le possibilità di applicazione. Invece con EOS ora e con ethereum 2.0 nel futuro sarà possibile utilizzare le blockchain per qualsiasi tipo di applicazione decentralizzata senza accusare differenze rispetto a una tradizionale applicazione.

4.4 Test smart contract

I test dello smart contract sono stati fatti con il linguaggio JavaScript sfruttando la suite truffle. Di seguito è riportata la tabella di tracciamento dei test.

Tabella 4.1: Tabella test smart contracti

Test id	**Descrizione**	**Requisiti coperti**
UT-1	Should deploy smart contract properly and save owner address	RFN-1
UT-2	Should add two person to people array and get them by id	RFN-2 - RF10
UT-3	Should not be able to add the same id person in people array	RFN-3
UT-4	Should add a contact between two id presents in people array	RFN-7
UT-5	Should not be able to add another person contacts	RFN-8
UT-6	Should not be able to add a contact with nonexisting person	RFN-9
UT-7	Should set a person infected	RFN-4
UT-8	Should not be able to set a person infected if the address is not who deploy the contract	RFN-5

Test id	**Descrizione**	**Requisiti coperti**
UT-9	Should not be able to set infected a nonexisting person	RFN-6
UT-10	Should not be able to get other people contacts	RFN-11
UT-11	Should check that the second id has a infection risk	RFN-12
UT-12	Should check that if a contact is not confirmed, then a person has not risk infection	13
UT-13	Should not be able to check another person risk infection	RFN-14
UT-14	Should check that sum of contactRates in the last day is 15 for the first id	RFN-15
UT-15	Should not be able to calculate another person sum of contactRates	RFN-16

Capitolo 5

Integrazione in applicazione di contact tracing

5.1 Applicazione android

5.1.1 Requisiti

1. Smart contract Tracing.sol
2. Address contratto
3. Chiave privata wallet
4. Address wallet
5. Compilatore solc
6. Web3j
7. Android studio

5.1.2 Installazione

Scaricare web3j dalla repo su Github
Estrarre lo zip scaricato con il seguente

```
unzip web3j-<version>.zip
```

Eseguire web3j con

```
web3j-<version>/bin/web3j
```

Creare i file .abi e .bin dello smart contract con il comando

```
solcjs ./Tracing.sol --bin --abi --optimize -o ./
```

A partire dai file .bin e .abi generare la classe java

```
web3j solidity generate -b ./Tracing.bin -a ./Tracing.abi -o ./
    -p GeneratedClasses
```

In questo modo nella directory /GeneratedClasses si troverà la classe Tracing.java, da inserire nel progetto di android per permettere l'integrazione dell'app con la blockchain. L'ultimo passaggio da effettuare per interagire con il contract è includere nel file build.gradle la dipendenza con la libreria web3j, inserendo la seguente riga

```
implementation 'org.web3j:core:<version>'
```

5.1.3 Utilizzo smart contract

Per utilizzare lo smart contract è stata creata la classe BlockchainManager che si occupa di caricare il contratto, creare delle credenziali e gestirne le funzioni. Di seguito viene riportata la classe sviluppata:

```
import android.content.Context
import android.util.Log
import androidx.preference.PreferenceManager
import com.google.gson.Gson
import it.synclab.synctrace.mobile.tracinglib.db.User
import org.bouncycastle.jce.provider.BouncyCastleProvider
import org.web3j.crypto.Credentials
import org.web3j.crypto.Keys
import org.web3j.crypto.Wallet
import org.web3j.protocol.Web3j
import org.web3j.protocol.core.DefaultBlockParameterName
import org.web3j.protocol.http.HttpService
import org.web3j.tx.Transfer
import org.web3j.utils.Convert
import java.math.BigDecimal
import java.math.BigInteger
import java.security.Security

class BlockchainManager {

    companion object {
        private const val TAG = "BlockchainManager"
        private val GAS_LIMIT = BigInteger.valueOf(6721975L)
        private val GAS_PRICE = BigInteger.valueOf(20000000000L)
    //private key metamask account
        private const val PRIVATE_KEY = YOUR_PRIVATE_KEY
    //address del contract
        private const val CONTRACT_ADDRESS = ADDRESS
        // shared preferences keys
        private const val CREDENTIALS = "bc_credentials"
        private const val TRANSACTION_CREDENTIALS = "
            bc_transaction_credentials"
    }

    private val web3j = Web3j.build(HttpService(
        "https://ropsten.infura.io/v3/YOUR_INFURA_ID"
    ))

    init {
        setupBouncyCastle()
    }
```

```
    private fun setupBouncyCastle() {
        val provider =
            Security.getProvider(BouncyCastleProvider.
                PROVIDER_NAME)
                ?:
                return
        if (provider.javaClass == BouncyCastleProvider::class.
            java) {
            return
        }
        Security.removeProvider(BouncyCastleProvider.
            PROVIDER_NAME)
        Security.insertProviderAt(BouncyCastleProvider(), 1)
    }

    fun registerPersonIfFirstTime(context: Context) {
        if (!areCredentialsPresent(context)) {
            val transactionCredentials = Credentials.create(
                PRIVATE_KEY)
            // create new private/public key pair
            val keys = Keys.createEcKeyPair()
            val uuid = User.getUuid(context)
            val wallet = Wallet.createLight(uuid, keys)
            val credentials = Credentials.create(Wallet.decrypt(
                uuid, wallet))

            sendFunds(context, 0.2, credentials,
                transactionCredentials)

            try {
                val tracingContract = loadContract(context,
                    credentials)
                tracingContract.addPerson(uuid, credentials.
                    address).sendAsync().get()
            } catch (e: Exception) {
                Log.e(TAG, "Cannot add person", e)
                return
            }

            setCredentials(context, credentials,
                transactionCredentials)
            Log.i(TAG, "Created credentials and registered person
                ")
        }
    }

    private fun areCredentialsPresent(context: Context): Boolean
        {
        val sp = PreferenceManager.getDefaultSharedPreferences(
            context)
        return sp.getString(CREDENTIALS, null) != null
    }
```

```
    private fun getCredentials(context: Context): Credentials {
        val sp = PreferenceManager.getDefaultSharedPreferences(
            context)
        val jsonCredentials = sp.getString(CREDENTIALS, null)
        return Gson().fromJson(jsonCredentials, Credentials::
            class.java)
    }

    private fun getTransactionCredentials(context: Context):
        Credentials {
        val sp = PreferenceManager.getDefaultSharedPreferences(
            context)
        val jsonCredentials = sp.getString(
            TRANSACTION_CREDENTIALS, null)
        return Gson().fromJson(jsonCredentials, Credentials::
            class.java)
    }

    private fun setCredentials(context: Context, credentials:
        Credentials,
                               transactionCredentials:
                                   Credentials) {
        val sp = PreferenceManager.getDefaultSharedPreferences(
            context)
        val gson = Gson()
        sp.edit().apply() {
            putString(CREDENTIALS, gson.toJson(credentials))
            putString(TRANSACTION_CREDENTIALS, gson.toJson(
                transactionCredentials))
        }.apply()
    }

    private fun loadContract(context: Context, credentials:
        Credentials = getCredentials(context)): TracingContract {
        return TracingContract.load(CONTRACT_ADDRESS, web3j,
            credentials, GAS_PRICE, GAS_LIMIT)
    }

    private fun sendFunds(context: Context, amount: Double,
                          credentials: Credentials =
                              getCredentials(context),
                          transactionCredentials: Credentials =
                              getTransactionCredentials(context))
                              {
        try {
            Transfer.sendFunds(
                web3j, transactionCredentials, credentials.
                    address,
                BigDecimal.valueOf(amount), Convert.Unit.ETHER
            ).sendAsync().get()
        } catch (e: Exception) {
            Log.e(TAG, "Cannot send funds", e)
        }
    }
```

```
    fun addContact(context: Context, myUuid: String, contactUuid:
        String, contactIndex: Double) {
        try {
            val tracingContract = loadContract(context)
            tracingContract.addContact(myUuid, contactUuid,
                BigInteger.valueOf(contactIndex.toLong()))
                .sendAsync().get()

            val ethGetBalance =
                web3j.ethGetBalance(CONTRACT_ADDRESS,
                    DefaultBlockParameterName.LATEST).sendAsync().
                    get()
            val wei = ethGetBalance!!.balance
            if (wei.compareTo(BigInteger.valueOf
                (10000000000000000L)) == -1) {
                sendFunds(context, 0.1)
            }
        } catch (e: Exception) {
            Log.e(TAG, "Cannot add contact to blockchain", e)
        }
    }
}
```

In questa classe per effettuare le chiamate alle funzioni dello smart contract è sufficiente richiamare la funzione seguita da una chiamata asincrona, come per il seguente caso:

```
  tracingContract.addPerson(
    uuid,credentials.address
  ).sendAsync().get()
```

Grazie a questa classe che gestisce il contratto, nell'applicazione è possibile utilizzare le funzioni dello smart contract dove desiderato.

5.2 Web application

5.2.1 Requisiti

* Node.js
* Address contratto
* Chiave privata wallet
* Address wallet

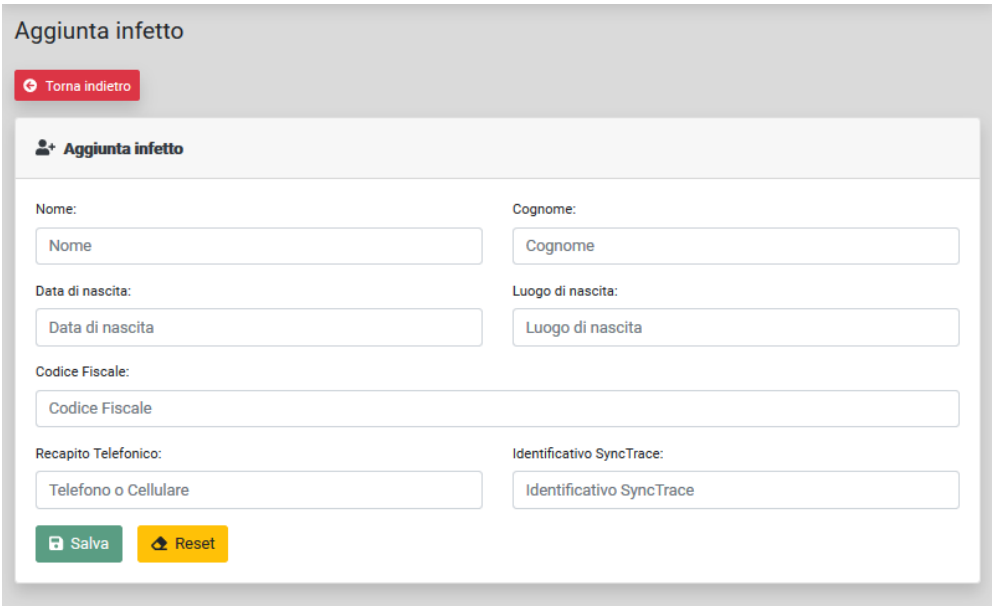
5.2.2 Utilizzo smart contract

Per interagire con lo smart contract dalla web application è stato utilizzato uno script opportunamente inserito nel codice dell'applicazione. Questo script permette di effettuare una chiamata alla funzione setInfected dello smart contract. SetInfected viene richiamata solo nella web application in quanto è autorizzato a effettuare la chiamata solo il personale medico. Per accertarsi di questo anche nello smart contract,

la funzione può essere chiamata solo dall'address di chi fa il deploy del contract. Per questo l'address che fa la chiamata nello script (salvato nella variabile addressFrom) è lo stesso indirizzo di chi fa il deploy.

Quando dalla schermata in figura 5.1 un medico inserisce un infetto, viene effettuata una chiamata alla relativa funzione nello smart contract.

Figura 5.1: Inserimento infetti da web application



Lo script utilizzato per farlo è il seguente:

```
const Web3 = require('web3')
Tx = require("ethereumjs-tx").Transaction

// connect to Infura node
const web3 = new Web3(new Web3.providers.HttpProvider('https://
    ropsten.infura.io/v3/YOUR_INFURA_ID'))

// the address that will send the test transaction
const addressFrom = 'YOUR_METAMASK_ADDRESS'
//private key of metamask account, be aware of the security
const privKey = environment.BC_PRIVATE_KEY;

// the contract address
const addressTo = 'CONTRACT_ADDRESS'

//abi of the contract
const abi = 'CONTRACT_ABI'

// Signs the given transaction data and sends it. Abstracts some
    of the details
// of buffering and serializing the transaction for web3.
function sendSigned(txData, cb) {
```

```
  const privateKey = Buffer.from(privKey, 'hex')
  const transaction = new Tx(txData, {'chain':'ropsten'})
  transaction.sign(privateKey)
  const serializedTx = transaction.serialize().toString('hex')
  web3.eth.sendSignedTransaction('0x' + serializedTx, cb)
}

const myContract = new web3.eth.Contract(JSON.parse(abi),
    addressTo);

const myData = myContract.methods.setInfected( INFECTED_ID,true).
    encodeABI();

// get the number of transactions sent so far so we can create a
    fresh nonce
web3.eth.getTransactionCount(addressFrom).then(txCount => {

  // construct the transaction data
  const txData = {
    nonce: web3.utils.toHex(txCount),
    gasLimit: web3.utils.toHex(6721975),
    gasPrice: web3.utils.toHex(20000000000),
    to: addressTo,
    from: addressFrom,
    value: web3.utils.toHex(web3.utils.toWei("0", 'wei')),
    data: myData
  }

  // fire away!
  sendSigned(txData, function(err, result) {
    if (err) return console.log('error', err)
    console.log('sent', result)
  })

})
```

Capitolo 6

Conclusioni

6.1 Raggiungimento degli obiettivi

Nella sezione 2.4.4 sono stati presentati gli obiettivi proposti dal tutor aziendale a inizio stage. Di seguito vengono riportate le tabelle degli obiettivi con il loro stato di completamento

Tabella 6.1: Tabella degli obiettivi obbligatori

Obiettivo	**Descrizione**	Stato
O01	Acquisizione competenze sulle tecnologie Blockchain, in particolare Ethereum	Completato
O02	Capacità di progettazione e analisi di smart contract	Completato
O03	Capacità di raggiungere gli obiettivi richiesti in autonomia, seguendo il programma preventivato)	Completato
O04	Portare a termine l'implementazione di almeno l'80% degli sviluppi previsti	Completato

Tabella 6.2: Tabella degli obiettivi desiderabili

Obiettivo	**Descrizione**	Stato
D01	Portare a termine il lavoro di studio della portabilità di Ethereum su dispositivi mobili	Completato
D02	Portare a termine l'implementazione completa degli sviluppi previsti	Completato

Tabella 6.3: Tabella degli obiettivi facoltativi

Obiettivo	**Descrizione**	Stato
F01	Completare l'installazione di un peer su un dispositivo mobile Android	Completato

6.2 Conoscenze acquisite

Nel co

6.3 Valutazione personale

Appendice A

Appendice A

Citazione

Autore della citazione

Bibliografia