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

Sommario

La pandemia globale che ci ha colpito quest'anno ha avuto conseguenze in numerosi campi, oltre che nella vita di tutti noi. Il settore informatico ha cercato una soluzione che potesse favorire la ripresa della normale quotidianità: sembrava necessario trovare un modo per tracciare i contagi, riuscire a risalire alle persone entrate in contatto con malati di Covid-19 e interrompere la catena del contagio, tutto questo rivolgendo particolare attenzione al rispetto della privacy. La risposta è stata trovata nell'utilizzo di un'applicazione mobile che registri i contatti tra dispositivi con la tecnologia bluetooth LE, come nel caso della famosa applicazione Immuni attualmente utilizzata, seppur con poco successo, in Italia. Se adottate da una buona parte della popolazione, le applicazioni di contact tracing forniscono un supporto importante per il contenimento del contagio. Avere a disposizione i contatti di una persona infetta, significa agire tempestivamente segnalando ai soggetti a rischio il pericolo, al fine di limitare ulteriori contagi a catena.
L'azienda Sync Lab, dove ho svolto lo stage, ha incaricato un gruppo di tirocinanti di sviluppare un'applicazione di contact tracing chiamata SyncTrace. Come per Immuni, SyncTrace sfrutta la tecnologia bluetooth LE, ma fornisce funzionalità in più per il controllo dei contagi. Inoltre, l'azienda intende integrare nell'applicazione la tecnologia blockchain, sviluppando uno smart contract da utilizzare per il tracciamento dei contatti. Il mio ruolo all'interno del progetto è stato proprio lo studio e l'implementazione di uno smart contract per l'applicazione SyncTrace. Il presente documento descrive il lavoro da me svolto durante il periodo di stage della durata di trecento ore.

Ringraziamenti

Prima di tutto, vorrei ringraziare la mia famiglia per il sostegno che mi ha sempre espresso nel raggiungimento di questo importante traguardo.

Ringrazio i miei amici, per essere stati al mio fianco e aver reso gli anni universitari indimenticabili.

Desidero ringraziare il Prof. Ballan, relatore della mia tesi, per la gentilezza e la disponibilità che ha mostrato durante la stesura dell'elaborato.

Ringrazio l'azienda Sync Lab, dove ho svolto lo stage, e il mio tutor Fabio Pallaro per l'attenzione e l'interesse mostrati nei confronti del mio lavoro.

Padova, Luglio 2020 Matteo Infantino

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

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*[g](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 e 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 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'azienda, infatti, vanta un alto tasso di assunzione post-stage e un basso *Turn over*[g].

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, per sistemi di integrazione o per sistemi di monitoraggio. Il laboratorio ricerca e sviluppo dell'azienda è sempre al passo con i nuovi paradigmi tecnologici e di comunicazione, come *big data*[g], *cloud computing*[g], *internet of things*[g](IoT), al fine di supportare i propri clienti nella creazione e integrazione di applicazioni, processi e dispositivi. L'azienda, grazie alla rete di relazioni a livello nazionale ed internazionale, ha ottenuto importanti finanziamenti in progetti europei (FP7 e H2020).
L'approfondita conoscenza di processi e tecnologie, maturata in esperienze altamente significative e qualificanti, permette all'azienda di gestire progetti di elevata complessità, dominando l'intero ciclo di vita del software:

* Studio di fattibilità;
* Analisi dei requisiti;
* Progettazione;
* Implementazione;
* Manutenzione.

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 principali prodotti offerti sono i seguenti:

* **SynClinic:** è un sistema informativo sanitario per la gestione integrata di tutti i processi clinici e amministrativi di ospedali, cliniche e case di cura. Gestisce, organizza e monitora tutte le fasi del percorso di cura del paziente, diventando supporto indispensabile per il personale sanitario;
* **DPS 4.0:** rappresenta una soluzione web per una compliance continua, gestendo la *GDPR Privacy* (*General Data Protection Regulation*), regolamento attraverso il quale la Commissione Europea intende rafforzare la protezione dei dati personali dei cittadini dell'Unione Europea;
* **StreamLog:** è un sistema finalizzato al soddisfacimento dei requisiti fissati dal garante. 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* e 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:** è un software che si propone come integrazione sinergica tra i mondi della videosorveglianza e quello dei sistemi informativi territoriali, abilitando il controllo totale dell'area da sorvegliare.

1.1.3 Settori di impiego

Sync Lab è specializzata 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 sicurezza informatica.

1.2 Organizzazione del testo

Il secondo capitolo fornisce una panoramica del percorso di stage intrapreso, focalizzandosi poi sulle tecnologie coinvolte nel progetto.

Il terzo capitolo illustra i requisiti emersi in fase di analisi.

Il quarto capitolo approfondisce la progettazione e l'implementazione del contratto.

Il quinto capitolo approfondisce l'integrazione del contratto nel software di contact tracing sviluppato. dall'azienda

Nel sesto capitolo si analizza il percorso effettuato e si stipulano le conclusioni.

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

* 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 parte 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

2.1.1 L'idea

L'obiettivo di Sync Lab è sviluppare un'applicazione per il *contact tracing*[g]. Il software deve poter registrare i contatti tra le persone sfruttando il *bluetooth LE*[g]. Le informazioni da raccogliere riguardano il tempo e la distanza di contatto con gli altri dispositivi, assicurando il rispetto della privacy. Quando una persona risulta infetta deve essere data la possibilità di inserire il suo stato nel sistema e di avvisare chi è entrato in contatto con questa persona tramite notifica del telefono.

L'azienda ha incaricato diversi stagisti di sviluppare un'applicazione mobile e una web application che gestiscano il *contact tracing*, la prima lato utente generico, la seconda per il personale sanitario. Lo scopo dell'applicazione mobile è quello di registrare i contatti, mentre lo scopo della web app 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. Mi è stato proposto di studiare e implementare uno *smart contract* per registrare i contatti nella piattaforma *Ethereum*. I punti principali da affrontare durante il periodo del mio stage sono i seguenti:

* 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).

2.1.2 Motivazioni

La pandemia di Covid-19 ha messo alla luce il bisogno di utilizzare applicazioni mobile per il *contact tracing*, in modo da prevenire la diffusione della malattia. Le motivazioni dell'azienda per sviluppare SyncTrace, dunque, appaiono abbastanza chiare. La domanda che sorge più spontanea riguarda l'utilizzo della *blockchain* per questo scopo.

Il contact tracing può portare a problemi di privacy perché richiede che le informazioni siano conservate e distribuite tra i dispositivi; inoltre è necessario garantire la protezione dell'identità degli utenti infetti. La blockchain può giocare un ruolo neutrale nella trasmissione di questi dati sensibili, fornendo una soluzione per la privacy grazie alla sua natura che garantisce immutabilità, incorruttibilità, anonimato e sicurezza.
Per esempio, nelle soluzioni più utilizzate per il contact tracing, è possibile attaccare il sistema per corrompere le informazioni. Un *replay attack* non è da escludere: un attaccante potrebbe impossessarsi delle credenziali di un dispositivo per simulare contatti inesistenti; utilizzando la blockchain si scongiurano questo tipo di pericoli.

2.2 Studio tecnologico

Lo studio richiesto per lo svolgimento dello stage parte dalla tecnologia *blockchain* in tutti i suoi aspetti, per poi passare allo studio specifico di *Ethereum*, *Solidity* e tutti gli strumenti di supporto per lo sviluppo e il testing di *smart contract*. Infine è necessario capire come integrare lo *smart contract* sviluppato in un ambiente mobile e in una web application.

2.2.1 Blockchain

Una *blockchain* è un registro pubblico e decentralizzato che offre la possibiltà di avere un database distribuito contenente tutte le transazioni eseguite e condivise tra i partecipanti della rete. Questo database viene chiamato *distributed ledger*[g], in italiano libro mastro decentralizzato. Tutte le transazioni realizzate vengono conservate nel *ledger*, che è pubblico, sicuro e immutabile. Ogni transazione nel *ledger* pubblico è verificata dalla maggioranza dei partecipanti grazie a un *algoritmo di consenso*[g].

Le principali caratteristiche delle *blockchain* sono:

* **Immutabilità:** quando una transazione è accettata e validata non può più essere cancellata o modificata;
* **Decentralizzazione:** la rete è distribuita sui nodi di tutti i partecipanti, permettendo diversi vantaggi come l'assenza di un *single point of failure*[g];
* **Anonimato:** è garantito l'anonimato dell'identità degli utenti che effettuano una transazione;
* **Sicurezza:** grazie all'assenza di un *single point of failure* la sicurezza di una *blockchain* è superiore rispetto a un tradizionale database centralizzato;
* **Capacità:** avere a disposizione un gran numero di macchine che lavorano inseme come un'unica entità si traduce in una grande potenza di calcolo e di capacità del database.

2.2.1.1 Permissionless e permissioned

Le blockchain si dividono in due categorie: *blockchain permissionless*[g] (o pubbliche) e *blockchain permissioned*[g] (o private). Le prime permettono a qualsiasi utente che decida di connettersi, di generare nuove transazioni, effettuare il compito di *miner*[g] o leggere il registro delle transazioni memorizzate. *Bitcoin*[g] ed *Ethereum*[g] sono esempi di *blockchain* pubbliche. Le seconde, invece, permettono l'accesso solo ad utenti autorizzati e autenticati, poiché la *blockchain* 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. Nelle *blockchain permissioned*, 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; ne esistono di diversi tipi in base al ruolo che hanno nella rete;
* **Transazioni:** sono le azioni generate dai partecipanti al sistema, costituite dai dati scambiati. Necessitano di essere verificate, approvate e poi archiviate;
* **Blocchi:** sono un raggruppamento di transazioni validate che vengono registrate nella *blockchain*, assicurando la corretta sequenza e la certezza che le informazioni registrate non siano state manomesse;
* **Distributed ledger:** è 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, grazie all'uso di *hash*[g].

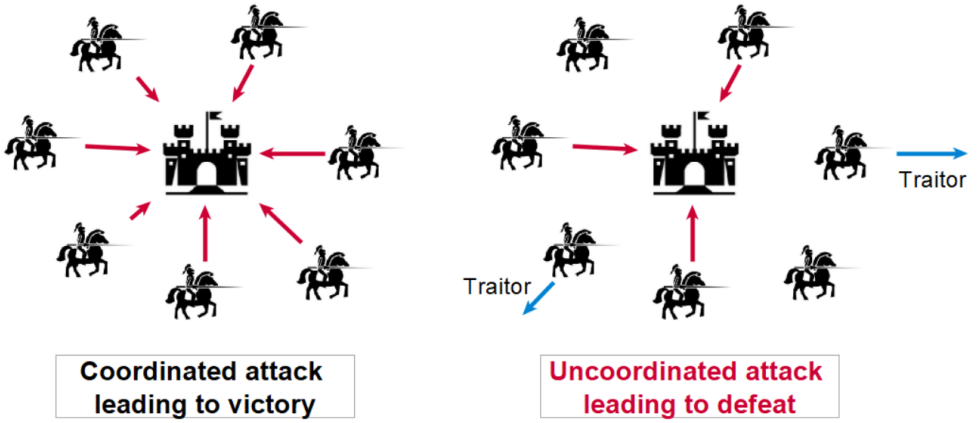
2.2.1.3 Generali bizantini

La *blockchain* nasce per eliminare il bisogno di una terza parte che validi 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 approvare i blocchi e quindi le transazioni. Per capire appieno 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? 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?

L'obiettivo da raggiungere è far sì che ai luogotenenti onesti arrivi il piano d'attacco corretto, senza che i traditori possano compromettere l'operazione facendo arrivare loro

Figura 2.1: Problema dei generali bizantini



le informazioni sbagliate. In una *blockchain*, la soluzione è quella di non avere più un generale che comanda sugli altri. 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, lo vedono e lo condividono. Di conseguenza la *blockchain* è incorruttibile se la maggioranza dei partecipanti risulta essere onesta.

2.2.1.4 Algoritmi di consenso

Gli algoritmi di consenso assicurano la sicurezza e l'integrità dei dati nei sistemi distribuiti. In una *blockchain* l'obiettivo di questi algoritmi è ottenere la validazione delle transazioni, rendere la rete affidabile e prevenire gli attacchi da parte di malintenzionati. Per ottenere questo risultato i nodi partecipanti devono essere d'accordo sullo stato della *blockchain*. 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*.

Proof of work L'algoritmo *proof of work*[g] si basa sulla generazione di blocchi attraverso complessi problemi matematici, la cui risoluzione richiede un grande sforzo computazionale. Il meccanismo su cui si basa questo algoritmo è la ricerca di un valore che sottoposto a una funzione *hash* risulti iniziare con un certo numero di zeri. Il lavoro medio è esponenzialmente proporzionale al numero di zeri richiesti. Quello che rende adatto l'*hash* per questo genere di algoritmo è il fatto che sia molto difficile trovare un valore che corrisponda alle caratteristiche richieste, ma estremamente facile verificarne la correttezza.
Generalmente si adotta un valore *hash* con un numero selezionabile di prime cifre a zero. 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 *criptovaluta*[g].
L'algoritmo *proof of work* risolve il problema dei generali bizantini se almeno il 50% + 1 dei partecipanti alla *blockchain* è onesta: soffre infatti dell'attacco del 51%.

Se la maggioranza della potenza computazionale della rete fosse controllata da un attaccante o un gruppo di attaccanti, questi avrebbero la possibilità di escludere intenzionalmente transazioni o modificarle a proprio piacimento. Un attacco di questo tipo consentirebbe all'attaccante di invertire le transazioni che ha effettuato, portando a un *double spending*[g]. Questo scenario potrebbe fare pensare a gravi problemi di sicurezza nelle *blockchain*, ma in realtà subire un attacco di questo tipo è altamente improbabile, soprattutto nelle *blockchain* di ampia diffusione. Quando una *blockchain* diventa sufficientemente grande, la possibilità che una singola persona o un singolo gruppo di persone sia in grado di controllare il 51% della rete è quasi nulla.

Proof of stake Nel *proof of stake* i blocchi vengono validati da chi possiede più *token*[g]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. Il *proof of work* richiede un'enorme quantità di tempo ed energia e, di conseguenza, un limitato numero di transazioni al secondo e un costo elevato da parte dell'utente per ottenere la validazione della propria transazione. Con il *proof of stake* si ottiene maggiore scalabilità e minor costo per le transazioni. Inoltre la struttura della *blockchain* la rende più sicura. 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 token hanno 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 *proof of stake*, che si dovrebbe completare nel 2022.

2.2.2 Ethereum

Ethereum è una piattaforma *blockchain* creata nel 2015 da Vitalik Buterik e che ha riscosso fin da subito grande successo. È nata come alternativa a *Bitcoin*, proponendo la possibilità di eseguire programmi chiamati *smart contract* per creare applicazioni decentralizzate. Anche *Bitcoin* ha un linguaggio che gli permette di creare *smart contract*, ma a differenza della famosa *blockchain*, in *Ethereum* è possibile scrivere contratti *Turing completi*[g]. Negli *smart contract* è possibile programmare come in qualsiasi altro linguaggio di programmazione, seppur con delle differenze di cui discuteremo più avanti.

2.2.2.1 Smart contract

Un contratto è un insieme di codice e dati che si trova a uno specifico *address* di *Ethereum*. È un vero e proprio programma che viene eseguito dai nodi della *blockchain*. Gli *smart contract* possono leggere o modificare il proprio stato interno, oltre a mandare o ricevere messaggi e transazioni. Esistono diversi linguaggi che permettono lo sviluppo di *smart contract* in *Ethereum*. I più famosi sono:

* *Solidity*, simile a *JavaScript*;
* *Vyper*: simile a *Python*.

Solidity è sicuramente il più diffuso per lo sviluppo di *applicazioni decentralizzate*[g]e, per questo motivo, è stato utilizzato durante lo stage.

2.2.2.2 Account, transazioni e gas

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*[g] che può essere modificato da transazioni inviate o ricevute.

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.

Gas
Le transazioni consumano un certo numero di gas, che rappresenta il carburante della blockchain. Ogni operazione negli smart contract ha un costo in gas, detto *gas price*[g], con cui viene calcolato il prezzo della *fee*[g] da pagare ai *miner* in *Ether*. Inoltre le operazioni hanno anche un limite di gas, detto *gas limit*[g], ossia la massima quantità di gas che può essere consumata in una transazione. Questo limite protegge la *blockchain* dai *loop* infiniti.

Figura 2.2: Logo di Ethereum

Figura 2.3: Logo di Solidity

2.2.2.3 Solidity

Solidity è un linguaggio tipato che permette di scrivere *smart contract*. I *contratti* a prima vista ricordano molto le classi nella programmazione orientata a oggetti. In *Solidity*, però, bisogna prestare particolare attenzione alla sicurezza, in quanto gli *smart contract* si occupano di gestire 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 *fee* in base al tipo di operazione richiesta. L'efficienza quindi, è cruciale nel definire *smart contract* e non sempre le *best practices*[g] in *Solidity* corrispondono alla soluzione che può sembrare migliore in un altro linguaggio di programmazione.

Di seguito viene riportato un contratto a titolo esemplificativo:

```
pragma solidity ^0.6.8;

contract HelloWorld {

  string hello;

  constructor() public {
      hello = "Hello World!";
  }

  function getHello() public view returns (string memory) {
    return hello;
  }

  function setHello(string memory _hello) public {
    hello = _hello;
  }
}
```

2.2.2.4 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*[g]).

2.2.2.5 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 rischi che possono essere incontrati durante lo stage. Tali rischi sono riportati di seguito insieme alle loro soluzioni.

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: Per affrontare la situazione nel migliore dei modi è opportuno pianificare il lavoro dettagliatamente, avere frequenti aggiornamenti con il tutor aziendale e utilizzare strumenti di supporto.

2. Scarsa conoscenza delle tecnologie previste per lo svolgimento dello stage
Descrizione: Le *blockchain*, *Ethereum* e lo sviluppo di *smart contract* 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: Ottimizzare il tempo con gli altri sviluppatori in azienda e, se necessario, comunicare con loro organizzando 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 quotidianamente 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*[g]. 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 dovuto dedicare parecchio tempo allo studio ed è stato difficile capire quali fossero le scelte migliori per facilitare il lavoro. A posteriori, però, è stato molto utile e formativo, anche se alcune cose non sono state utilizzate nell'implementazione finale.

2.4.4 Obiettivi

All'inizio dello stage, il tutor aziendale mi ha posto i seguenti obiettivi da raggiungere entro il termine del percorso:

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

Il tutor aziendale ha pianificato il lavoro da svolgere per ogni settimana, nel seguente modo:

* **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 blockhain;
  - Studio del funzionamento delle blockchain.
* **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

In questo capitolo vengono riportati i requisiti richiesti dal prodotto finale con l'utilizzo di una tabella per il tracciamento degli stessi.

3.1 Casi d'uso

All'inizio dello stage sono stati discussi i casi d'uso del software 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. Tuttavia la *blockchain* non è visibile da chi utilizza l'applicazione e non richiede un'interazione con l'utente. I casi d'uso sono stati stilati dai tirocinanti che hanno sviluppato l'applicazione e io mi sono limitato a rispettarli e a trovare i requisiti necessari allo *smart contract* e all'integrazione con il software.

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 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
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*[g] di una persona negli ultimi gionri

Requisito	Descrizione
RFN-16	La somma degli indici di contatto deve essere effettuata solo dal proprietario dell'informazione
RFN-17	Deve essere effettuato il deployment dello smart contract
RFN-18	Lo smart contract deve essere utilizzato dall'applicazione mobile per l'inserimento dei contatti
RFN-19	Lo smart contract deve essere utilizzato dalla web application per l'inserimento degli infetti

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*[g] 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)
RVD-3	Installazione dello smart contract in ambiente web

Capitolo 4

Progettazione e implementazione

In questo capitolo viene descritta la progettazione dello smart contract e della sua integrazione nell'applicazione SyncTrace, per poi passare all'implementazione del contratto. Infine si analizza in modo critico lo smart contract dal punto di vista dei consumi, dando possibili alternative di implementazione.

4.1 Progettazione

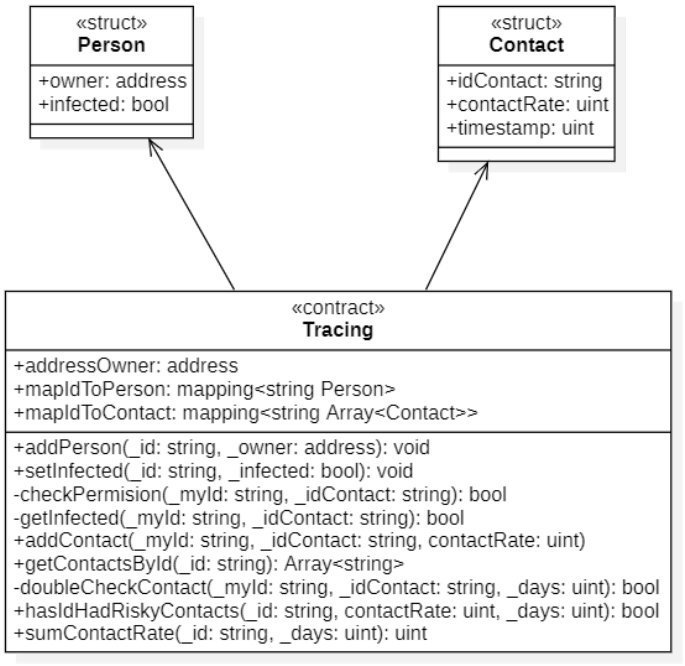
Figura 4.1: Logo di Synctrace

SyncTrace, il software di *contact tracing* ideato dall'azienda, è formato da due applicativi. Il primo è un'applicazione mobile con cui tracciare i contatti, il secondo è una web application in cui inserire gli infetti. Lo *smart contract* da sviluppare deve fornire le funzionalità necessarie per entrambi e deve essere possibile utilizzare lo *smart contract* in entrambi gli ambienti.

4.1.1 Smart Contract

Per rispettare i requisiti individuati, si è pensato di utilizzare due strutture, *Person* e *Contact*: *Person* rappresenta una persona, mentre *Contact* è un singolo contatto che una persona ha con un altro utente.
Il contratto *Tracing* avrà poi un campo dati di tipo *address* che rappresenta l'indirizzo del proprietario del contratto, una mappa che associa l'id di una persona alla struttura *Person* e una mappa che associa l'id di una persona a un *array* di contatti registrati. La figura 4.2 riporta un piccolo diagramma riassuntivo che rappresenta lo *smart contract* Tracing, con le varie funzionalità richieste dai requisiti.

Figura 4.2: Diagramma *UML*[g] smart contract



Come si può notare dal diagramma, *Person* ha due campi dati:

* *address owner*: indica l'account Ethereum di ogni persona, necessario per permettere l'utilizzo delle funzionalità solo sul proprio account;
* *bool infected*: booleano che indica lo stato di salute della persona.

Contact invece ha:

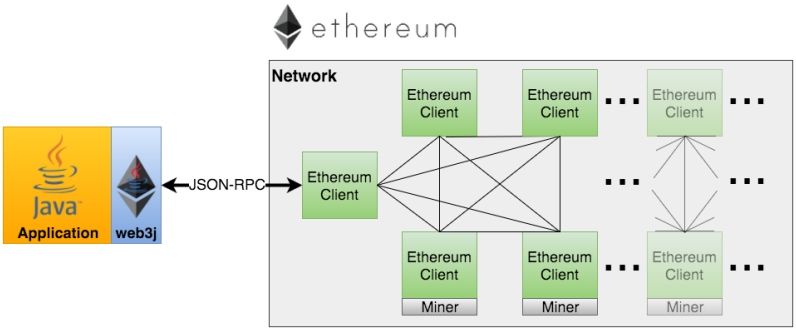
* *string idContact*: indica l'id della persona con cui si è entrati in contatto;
* *uint contactRate*: indica l'indice di contatto avuto, calcolato in base al tempo e alla distanza;
* *uint timestamp*: indica il momento in cui è stato registrato il contatto.

Una volta effettuato il *deployment*[g], il contratto Tracing deve inizializzare *addressOwner* con l'indirizzo dell'account che ha effettuato il deployment, ossia l'account dell'azienda. Questo account avrà dei privilegi attraverso i quali sarà possibile utillizzare delle funzioni critiche. Per esempio la funzione per segnalare un infetto non deve poter essere chiamata da chiunque, per scongiurare la possibilità che qualcuno possa segnalarsi infetto, anche se non lo è.
La mappa che contiene le persone registrate è stata pensata per essere popolata al primo avvio dell'applicazione con un id generato randomicamente.
I contatti di ogni persona, invece, saranno aggiunti ogni volta che l'applicazione rileva un contatto tramite la tecnologia bluetooth LE.

4.1.2 Integrazione mobile

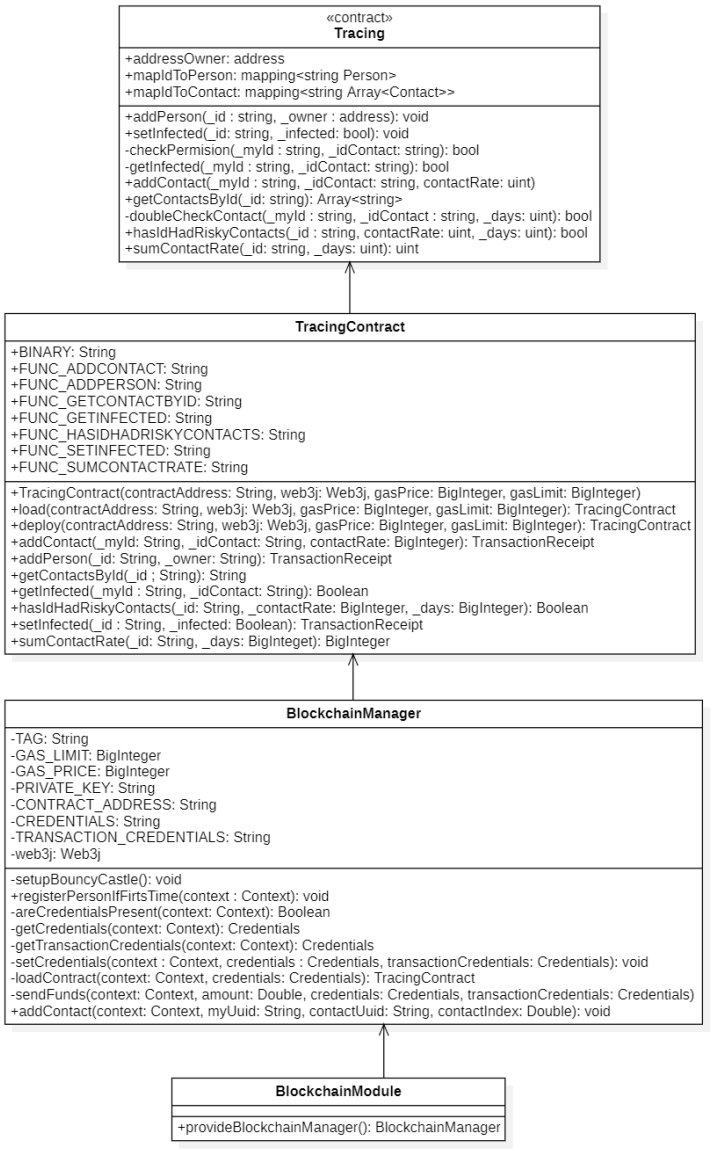
Per sfruttare la rete *Ethereum* in android si utilizza la libreria *Java web3j*, nata per lavorare con gli smart contract e interagire con i nodi di *Ethereum*

Figura 4.3: Integrazione applicazione Java - client Ethereum con web3j



La figura 4.4 riporta un diagramma *uml* che mostra l'integrazione del contratto con l'applicazione mobile SyncTrace. A partire dal contratto Tracing, grazie a *web3j*, si genera una classe Java *TracingContract* che contiene le funzioni dello *smart contract*, oltre a due metodi per effettuare il *deployment* e il caricamento del contratto. La classe *TracingContract* viene a sua volta utilizzata da un *BlockchainManager*, che gestisce la configurazione e tutte le operazioni necessarie all'applicazione. In questo modo, quando nel codice dell'applicazione è necessario effettuare una chiamata allo *smart contract*, è sufficiente instanziare la classe *BlockchainManager* e chiamare il metodo desiderato. I dettagli relativi all'implementazione nell'applicazione saranno discussi approfonditamente in seguito.

Figura 4.4: Diagramma di classe integrazione smart contract - SyncTrace app



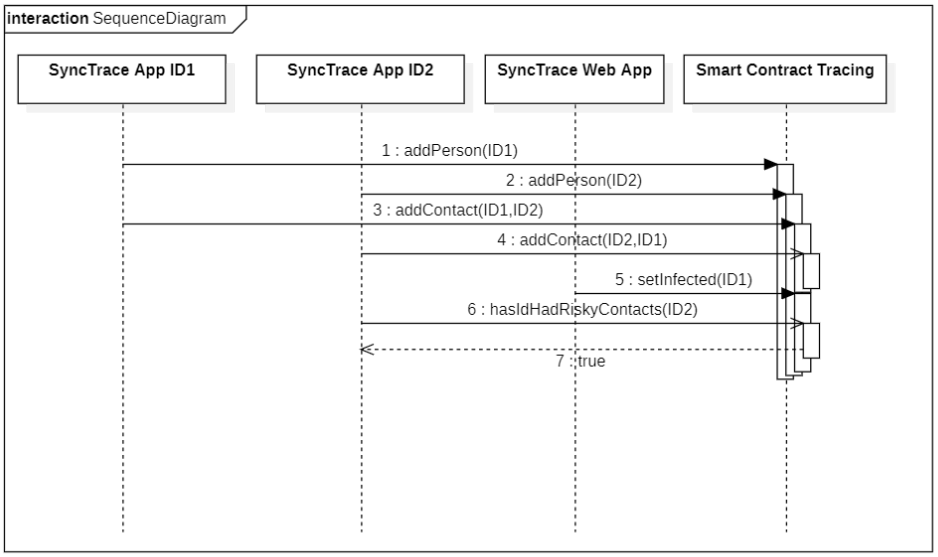
4.1.3 Integrazione web application

Come nell'ambiente mobile, anche per la web application si utilizza una libreria per interagire con *Ethereum*: *web3js*. Per integrare lo *smart contract* alla web app di SyncTrace è sufficiente implementare uno script che configuri la connesione al nodo della *blockchain* e richiami la funzione dello *smart contract* adibita all'inserimento di una persona infetta. Nel capitolo 5 verrà approfondita la questione.

4.1.4 Diagramma di sequenza

Per comprendere il funzionamento generale di SyncTrace con la *blockchain* riporto un diagramma di sequenza semplificato che mostra come le parti interagiscano nel caso d'uso principale:

Figura 4.5: Diagramma di sequenza SyncTrace



Nel diagramma sono presenti quattro partecipanti: due applicazioni SyncTrace mobile, la web application di SyncTrace e lo *smart contract*. Quando l'applicazione viene installata negli smartphone, vengono create delle credenziali e l'utente viene aggiunto in catena con il metodo *addPerson*. Il diagramma, poi, mostra l'aggiunta di un contatto tra le due applicazioni tramite la funzione *addContact*.
Quando una persona risulta positiva a un tampone, tramite web application, è possibile segnalare il suo stato di salute con la funzione *setInfected*.
A questo punto, l'applicazione che ha registrato un contatto con il malato, interrogando la*blockchain* con il suo identificativo, otterrà il valore booleano *true*, che indica che tra i suoi contatti è presente un infetto.

4.2 Proof of concept

Prima di mostrare lo smart contract sviluppato, ritengo utile mostrare una versione inizialmente pensata con il tutor aziendale e implementata come *Proof of concept*[g], per analizzare le differenze tra i due contratti, soprattutto come consumi di gas. Una parte dello stage, infatti, è stata incentrata su come ottimizzare il codice *Solidity*, cercando di progettare lo *smart contract* in modo da raggiungere i requisiti con le operazioni meno onerose nella piattaforma *Ethereum*. Le funzionalità di questa versione del contratto sono le medesime di quello definitivo e, per questo, non verranno illustrate in questa sezione. Il codice riportato nelle prossime righe mostra la differenza nelle struct e nei campi dati del contratto rispetto alla progettazione definitiva.

```
pragma solidity ^0.6.7;
pragma experimental ABIEncoderV2;

contract Tracing {

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

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

    contract Tracing {

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

  /*
    methods
  */
}
```

Come per la versione definitiva, sono presenti le due struct *Person* e *Contact*, ma con alcune differenze. *Person* contiene anche l'id dell'utente, mentre *Contact* ha due campi dati *Person* che rappresentano gli utenti entrati in contatto. Per quanto riguarda i campi dati del contratto, è sempre presente un address con lo stesso scopo, ma il modo di registrare le persone e i contatti è differente. Invece di usare delle mappe sono presenti due *array*, rispettivamente *people* e *contacts*. Lo scopo dell'*array people* è sostanzialmente lo stesso della mappa *mapIdToPerson*, ovvero contenere tutti gli utenti registrati. Per registrare i contatti, invece, c'è una grande differenza di progettazione. Invece di associare ogni persona a un'*array* che contenga tutti i contatti che ha, è presente un unico *array contacts*, che include tutti i contatti avuti tra gli utenti.
Il *Proof of concept* mostrato è stato scartato per alcune criticità. In primo luogo

avere una struttura dati che contenga i contatti di tutti gli utenti non è una soluzione particolarmente furba perché complica inutilmente il contratto in qualsiasi tipo di operazione sui contatti e, soprattutto, perché per il suo funzionamento 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 *Ethereum*.
Nella sezione 4.3.2 approfondirò l'argomento, mostrando i consumi del *proof of concept* a confronto con il contratto definitivo.

4.3 Implementazione

4.3.1 Smart contract Tracing

L'implementazione dello *smart contract*, come visto nella sezione progettazione, prevede l'utilizzo di due strutture, un address e due mappe per associare l'id di un utente alle sue informazioni e ai suoi contatti. L'intestazione del contratto, dunque, è la seguente:

```
pragma solidity ^0.6.7;
pragma experimental ABIEncoderV2;

contract Tracing {

    struct Person {
        address owner;
        bool infected;
    }

    struct Contact {
        bytes32 idContact;
        uint256 contactRate;
        uint256 timestamp;
    }

    address ownerAddress;
    mapping(bytes32 => Person) mapIdToPerson;
    mapping(bytes32 => Contact[]) mapIdToContact;

  /*
  methods
  */
}
```

È disponibile un costruttore che al momento del *deployment* assegna la variabile *addresOwner* al chiamante *msg.sender*:

```
    constructor() public {
        ownerContract = msg.sender;
    }
```

Inoltre viene dichiarato un modificatore che viene utilizzato da diverse funzioni del contratto. In *Solidity* un modificatore viene eseguito prima della funzione in cui ne viene richiesto l'utilizzo. Come un metodo, ha un nome e degli argomenti, e al suo interno si trova una condizione. Se la condizione è soddisfatta, viene eseguito il corpo della funzione che utilizza il modificatore; in caso contrario la funzione non viene eseguita.
Nel *contract Tracing* la condizione del modificatore *isOwner* è di eseguire la funzione se e solo se l'utente che chiama il metodo ha l'address che corrisponde a quello salvato nel campo dati *owner* della persona con id dato in input. In questo modo le funzioni in cui viene specificata questa condizione possono essere eseguite solo da chi ha i permessi per farlo. Il modificatore è il seguente:

```
    modifier isOwner(bytes32 _id) {
        require(msg.sender == mapIdToPerson[_id].owner,"You are
            not the owner");
        _;
    }
```

Sono state implementate delle funzioni ad uso interno, ossia non visibili e non eseguibili se non all'interno dello *smart contract*. Sono funzioni di utilità utilizzate da alcuni metodi pubblici e per questo non sono disponibili all'utente:

```
/*
Funzione di utilita' che prende in input l'id di chi chiama la
    funzione e l'id del contatto da verificare.
Prima dell'esecuzione chiama il modificatore isOwner per
    verificare i permessi.
Ritorna il valore del booleano infected per la persona di id =
    _idContact
*/
function getInfected(bytes32 _myId,bytes32 _idContact) internal
    view isOwner(_myId) returns (bool);

/*
Funzione di utilita' che prende in input l'id di chiama la
    funzione, l'id del contatto da verificare e il numero dei
    giorni da controllare.
Verifica che se il primo id ha aviuto un contatto con il secondo
    id, anche il secondo id deve aver avuto un contatto con il
    primo.
Ritorna un booleano che indica il risultato del controllo
*/
function doubleCheckContact(bytes32 _myId, bytes32 _idContact,
    uint _days) internal view returns (bool);
```

Infine le funzioni disponibili all'utente hanno tutte come visibilità *external* e sono le seguenti:

```
/*
Funzione che prende in input l'id e l'indirizzo di una persona e
    l'aggiunge alla mappa mapIdToPerson.
Se la persona e' gia' presente nella mappa, viene restituito un
    errore.
*/
function addPerson(bytes32 _id, address _owner) external;

/*
Funzione che prende in input l'id di una persona e un booleano
    che indica il suo stato di salute (infetto o meno). e cambia
    la variabile infected della persona con l'id inserito.
Puo' essere chiamata solo dall'account proprietario del contratto
     (chi ha fatto il deployment)
Restituisce un errore se l'id non e' registrato.
*/
function setInfected(bytes32 _id,bool _infected) external;

/*
Funzione che prende in input l'id del chiamante, l'id del
    contatto avuto e un indice di contatto.
Aggiunge un contatto con i dati in input alla mappa
    mapIdToContacts.
Restituisce un errore se il chiamante non ha i permessi per
    inserire il contatto o se gli id inseriti non sono registrati.
*/
function addContact(bytes32 _myId,bytes32 _idContact,uint
    _contactRate) external isOwner(_myId);

/*
Funzione che prende in input un id e restituisce un array
    contenente i contatti dell'id inserito.
Restituisce un errore se il chiamante non ha i permessi per
    richiedere i contatti oppure se l'id inserito non e'
    registrato.
*/
function getContactsById(bytes32 _id) external view isOwner(_id)
    returns(bytes32[] memory);

/*
Funzione che prende in input un id, un indice di contatto e un
    numero di giorni.
Controlla se tra i contatti della persona con l'id inserito ci
    sia un contatto con una persona infetta.
Il contatto deve avere indice superiore a quello indicato e deve
    essere avvenuto negli ultimi giorni inseriti in input.
Restituisce un errore se il chiamante non ha i permessi necessari
    .
*/
function hasIdHadRiskyContacts(bytes32 _id,uint _contactRate,uint
     _days) external view isOwner(_id) returns (bool);
```

```
/*
Funzione che prende in indice un id e un numero di giorni.
Ritorna la somma degli indici di contatto per l'id in input negli
    ultimi giorni indicati.
Restituisce un errore se il chiamante non ha i permessi necessari
    .
*/
function sumContactRate(bytes32 _id,uint256 _days) external view
    isOwner(_id) returns (uint256);
```

4.3.2 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 *miner*. Più un'operazione è dispendiosa a livello di risorse, più gas questa operazione costa. Il concetto di gas permette di addebitare una tariffa che viene pagata ai *miner*, incentivandoli a prendere parte attiva al sistema.
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: il *contract* è stato ottimizzato per cercare di limitare il più possibile il consumo di gas.

Dal *Proof of concept* allo *smart contract* finale, infatti, la differenza di gas per il *deployment* e per le operazioni è notevole. Nelle tabelle successive sono riporatati i consumi per entrambe le versioni. I costi stimati in euro fanno riferimento all'attuale tasso di conversione di 202 euro/ether. Il costo per unità di gas, invece è 26 *gwei*[g]/gas.

Tabella 4.1: Tabella gas Proof of Concept

Metodo	**Consumo gas medio**	**Costo medio (eur)**
addPerson	79367	0.41
addContact	203512	1.07
setInfected	50111	0.26
Deployment	**1599579**	**8.41**

Tabella 4.2: Tabella gas Tracing

Metodo	**Consumo gas medio**	**Costo medio (eur)**
addPerson	43859	0.23
addContact	104806	0.55
setInfected	29364	0.15
Deployment	**839416**	**4.41**

Come si può notare, il costo del *deployment* dei due *smart contract* è quasi dimezzato: 1.600.000 contro 830.000. Il risparmio c'è, ma non è il più significativo pensando al fatto che il *deployment* viene effettuato una singola volta. Il costo del *deployment*, in un'applicazione *Ethereum*, è abbastanza marginale rispetto al costo delle operazioni se, come in questo caso, le funzioni devono essere chiamate molte volte. Risulta molto più interessante analizzare il consumo delle funzioni *addPerson*, *addContact* e *setInfected*. Anche per queste operazioni il risparmio è molto importante: ogni chiamata ha un consumo all'incirca dimezzato rispetto al *Proof of Concept*:

* *addPerson*: 79367 prima, 43859 dopo
* *addContact*: 203512 prima, 104806 dopo
* *setInfected*: 50111 prima, 29364 dopo

Questi risultati sono in parte dovuti alla modifica progettuale del contratto. Nel *Proof of Concept* sono stati utilizzati *array* dinamici per contenere gli utenti registrati e i contatti. Al contrario, nello *smart contract* finale, si è fatto uso di mappe. In *Solidity* gli array dinamici hanno delle *features* che li rendono inevitabilmente più costosi come:

* La variabile *length*, per contare il numero di oggetti memorizzati;

* Il *bound-checking* per avere un controllo sull'accesso casuale.

Inoltre, grazie all'utilizzo delle mappe, le funzioni dello *smart contract* risultano essere più semplici e con meno operazioni da effettuare. Un esempio lampante è il confronto tra stringhe. Per la natura del contratto, nel *Proof of concept* è necessario confrontare più volte gli id di tipo stringa. Il confronto booleano tra stringhe, però, non esiste in *Solidity*. Si può risolvere questa problematica calcolando l'*hash* delle stringhe, per poi confrontare il risultato. Tuttavia, l'operazione di *hashing* è una tra le più onerose e aumenta sensibilmente il costo del contratto.

Sono stati presi altri accorgimenti per il risparmio di gas come:

* **Funzioni interne:** è stata utilizzzata la visibilità internal per le funzioni non utilizzate all'esterno del contratto;
* **Funzioni esterne:** è stata utilizzata la visibilità external invece di quella public;
* **Storage:** sono state limitate le operazioni di storage;
* **Eventi:** gli eventi non sono stati utilizzati. Le funzioni che emettono eventi hanno un costo molto superiore;
* **Tipo bytes:** dove possibile è stato utilizzato il tipo bytes invece del tipo stringa;
* **Assert vs require:** le eccezioni sono state gestite con la keyword require;
* **Ottimizzazione compilazione:** la compilazione Solidity è stata ottimizzata con la keyword optimize;

4.3.3 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 *deployment* 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 come campione metà della popolazione italiana, il metodo viene chiamato 30 milioni di volte. Con il prezzo attuale dell'*Ether* (202 euro/Ether) e impostando il *gas price* a 26 gwei/gas, ogni chiamata costa circa 23 centesimi. Questo vuol dire che solo 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. Con il *gas price* e il tasso di cambio indicati, la chiamata per aggiungere un contatto ha un costo di circa 50 centesimi e non è difficile immaginare che il numero di chiamate sia di svariati milioni al giorno. Per questo motivo, soluzioni di questo genere sono difficilmente utilizzabili nella realtà. Tuttavia ci sono delle soluzioni alternative.

4.3.4 Soluzioni

4.3.4.1 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; i contatti con gli altri dispositivi 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. A seguito di questa segnalazione, l'id del malato viene inserito nell'*array* degli infetti nello *smart contract*. In questo modo lo *smart contract* risulta estremamente più semplice. Gestisce un *array* di infetti con una funzione per l'aggiunta e una per la rimozione, in modo sicuro e immutabile. Lo smart contract in questione potrebbe essere implementato in questo modo:

```
pragma solidity ^0.6.7;
pragma experimental ABIEncoderV2;

contract Tracing {

    bytes32[] infected;

    function addInfected(bytes32 _id) external {
        //require only doctors
        infected.push(_id);
    }

    function removeInfected(bytes32 _id) external {
        //require only doctors
        for(uint i = 0; i < infected.length; i++) {
            if(infected[i] == _id) {
                infected[i] = infected[infected.length - 1];
                infected.pop();
            }
        }
    }

    function getInfected() external view returns (bytes32[]
        memory){
        uint length = 0;
        for(uint i = infected.length; i > 0; i--) {
            length++;
        }
        bytes32[] memory toReturn = new bytes32[](length);
        for(uint i = infected.length; i > 0; i--) {
            toReturn[i-1] = infected[i-1];
        }
        return toReturn;
    }

}
```

Lato mobile, periodicamente il dispositivo chiama la funzione dello *smart contract* che restituisce gli infetti. Gli id ritornati vengono confrontati con gli id dei contatti salvati localmente per verificare la presenza di un rischio contagio. Tutte le altre operazioni sono effettuate localmente. Il consumo del gas è infinitamente più contenuto e permette un utilizzo reale dell'applicazione. Questo perché le uniche operazione che richiedono una transazione, e dunque un consumo di gas, sono quelle di inserimento e rimozione di un id nell'*array*. Questi metodi possono essere eseguiti solo dal personale sanitario a seguito di un tampone risultato positivo.
Il consumo di gas per questo *smart contract* è riportato nella seguente tabella:

Tabella 4.3: Tabella gas smart contract semplificato

Metodo	**Consumo gas medio**	**Costo medio (eur)**
addInfected	52272	0.28
removeInfected	28353	0.15
Deployment	**196237**	**1.04**

Dalla tabella si può notare come il costo di *deployment* del contratto sia notevolmente inferiore rispetto al precedente: 1.500.000 gas contro 200.000 circa. Ma la differenza la fa soprattutto il numero di chiamate alle funzioni *addInfected* e *removeInfected*, decisamente inferiori rispetto a tutte le chiamate necessarie per registrare i contatti. In Italia i contagi certificati ad oggi sono circa 200 mila. 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 40 mila euro, in linea con le *dApp* più utilizzate.

4.3.4.2 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 *fee*, 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 desiderato al prezzo corrente della criptovaluta. 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 *proof of stake* 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 possibili applicazioni. Con *Ethereum 2.0*, invece, potrebbe essere possibile utilizzare le *blockchain* per qualsiasi tipo di *applicazione decentralizzata*, senza accusare differenze rispetto a quelle tradizionali.

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 4.4: 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 deploys the contract	RFN-5
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

Test id	**Descrizione**	**Requisiti coperti**
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 SyncTrace

In questo capitolo viene mostrata l'integrazione dello smart contract con l'applicazione mobile e la web application di SyncTrace.

5.1 Applicazione android

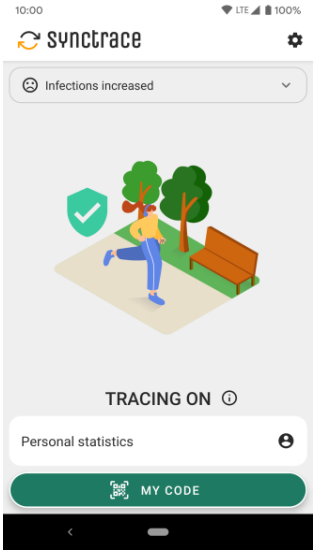


Figura 5.1: Schermata principale SyncTrace senza rischio contagio

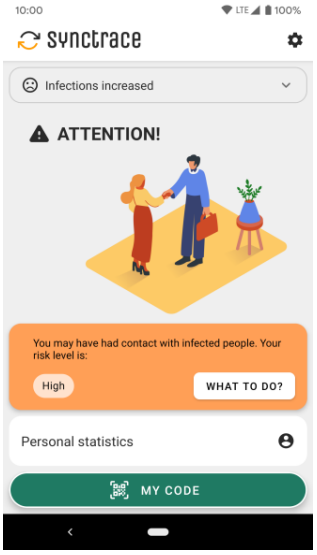


Figura 5.2: Schermata principale SyncTrace con rischio contagio

5.1.1 Obiettivo

L'obiettivo dell'integrazione con l'applicazione è implementare una classe che gestisca il contratto e tutte le operazioni legate alla *blockchain*, come visto in fase di progettazione. In particolare è necessario avere a disposizione:

* Credenziali per ogni utente da creare al primo avvio dell'app;

* Trasferimento di *Ether* a ogni utente, con ricarica all'esaurimento;
* Gestione delle funzioni dello *smart contract* richieste dall'applicazione, come l'aggiunta dei contatti.

5.1.2 Generazione classe Tracing.java

Per potere utilizzare lo *smart contract*, bisogna generare una classe Java a partire dal contratto, sfruttando *web3j*. La classe generata renderà disponibili i metodi dello *smart contract*, oltre a funzioni per effettuare il *deployment* e caricare un contratto.

La prima cosa da fare è scaricare *web3j* dalla repo su Github ed estrarre il file zip scaricato con il comando:

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

Sempre da terminale, si può avviare web3j in questo modo:

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

A questo punto è possibile 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 si può finalmente 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 del progetto la dipendenza con la libreria web3j, inserendo la seguente riga:

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

5.1.3 Implementazione

Oltre alla classe *Tracing.java*, generata da *web3j*, è necessario implementare una classe che si occupi di configurare e gestire le operazioni in *blockchain*. Nel progetto SyncTrace è stata creata la classe *BlockchainManager* per questo scopo, scritta in linguaggio Kotlin come il resto dell'applicazione. Per configurare correttamente il collegamento con lo *smart contract* sono necessarie le seguenti variabili:

```
    private val GAS_LIMIT = BigInteger.valueOf(GAS_LIMIT)
    private val GAS_PRICE = BigInteger.valueOf(GAS_PRICE)
    private const val PRIVATE_KEY = YOUR_PRIVATE_KEY
    private const val CONTRACT_ADDRESS = ADDRESS
    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"
    ))
```

GAS_LIMIT e *GAS_PRICE* sono i valori relativi al gas in *Ethereum* e devono essere passati nelle funzioni che richiedono una transazione.

La variabile *PRIVATE_KEY* è la chiave privata dell'account *Metamask*[g]che finanzia l'applicazione.

CONTRACT_ADDRESS è l'indirizzo del contratto nella *blockchain*. È possibile effettuare il *deployment* anche tramite *web3j* e ottenere l'*address* del contratto da utilizzare.

CREDENTIALS e *TRANSACTION_CREDENTIALS* rappresentano le keyword utilizzate per salvare le credenziali nelle *shared preferences* di android. Vengono create al primo avvio dell'app e salvate per essere utilizzate nel chiamare le funzioni dello *smart contract*.

web3j è un'istanza utilizzata per fornire un client eseguito tramite il provider Infura.

Nella classe sono disponibili alcune funzioni di utilità utilizzate dai metodi che verranno successivamente descritti:

```
/*
Controlla se le credenziali siano gia' state salvate nelle shared
    preferences
*/
private fun areCredentialsPresent(context: Context): Boolean


/*
Ritorna le credenziali dell'utente dalle shared preferences
*/
private fun getCredentials(context: Context): Credentials

/*
Ritorna le credenziali dell'admin dalle shared preferences
*/
private fun getTransactionCredentials(context: Context):
    Credentials

/*
Salva le credenziali nelle shared preferences
*/
private fun setCredentials(context: Context, credentials:
    Credentials,
                               transactionCredentials:
                                   Credentials)

/*
Ritorna un'istanza del contratto
*/
private fun loadContract(context: Context, credentials:
    Credentials = getCredentials(context)): TracingContract

/*
Effettua una transazione dall'account admin all'account utente
*/
private fun sendFunds(context: Context, amount: Double,
                          credentials: Credentials =
                              getCredentials(context),
                          transactionCredentials: Credentials =
                              getTransactionCredentials(context))
```

Al primo avvio dell'applicazione bisogna creare delle credenziali per il nuovo account, effettuare una transazione verso questo account, caricare il contratto *Tracing* e infine aggiungere l'utente in *blockchain* tramite la funzione *addPerson* dello *smart contract*. La funzione *registerPersonIfFirstTime* si occupa di tutto ciò

```
    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
                ")
        }
    }
```

Infine la classe *BlockchainManager* deve fornire una funzione per ogni metodo del contratto che viene utilizzato nell'applicazione, come per il metodo *addContact* che aggiunge un contatto in *blockchain* quando rilevato dal bluetooth.

```
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)
    }
}
```

È importante soffermarsi sulla funzione *addContact* per alcune considerazioni:

* La chiamata al metodo *addContact* deve essere asincrona, per permettere all'applicazione di continuare l'esecuzione senza interruzioni. La transazione in *Ethereum* non è immediata ed è opportuno che in questo lasso di tempo l'applicazione non si interrompa;
* Dopo l'esecuzione della transazione viene controllato il *balance* dell'account e, se risulta inferiore a una certa soglia, viene ricaricato;
* Possono essere sollevate eccezioni nel caso in cui la transazione non vada a buon fine; per questo è importare gestirle correttamente per evitare crash dell'applicazione.

5.2 Web application

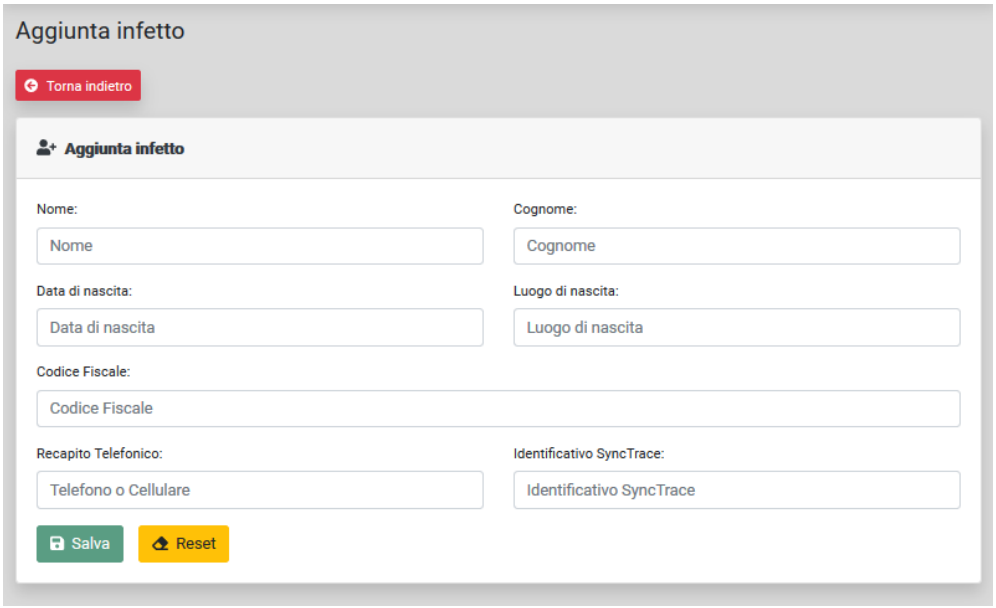
5.2.1 Obiettivo

La web application di SyncTrace è utilizzata dal personale sanitario per avere un'interfaccia di gestione con dei privilegi speciali. Tra questi spicca l'inserimento di una persona infetta dopo il risultato positivo di un tampone. Lo *smart contract* ha un metodo che permette di farlo e l'obiettivo di questa sezione è mostrare l'interazione con la *blockchain* in ambiente web.

5.2.2 Implementazione

La web app ha la seguente pagina dedicata all'inserimento di un infetto nel sistema

Figura 5.3: Inserimento infetti da web application



Al momento del salvataggio dei dati presenti nel form, deve essere effettuata una chiamata al metodo *setInfected* dello *smart contract*. Come mostrato nell'implementazione dello *smart contract*, solo l'*address* dell'admin può chiamare questa funzione. In caso contrario viene lanciata un'eccezione e l'inserimento viene rigettato. Così facendo c'è la certezza che nessuno possa modificare scorrettamente i dati in *blockchain*.

Per implementare quanto descritto si sfrutta la libreria *web3js*, con i seguenti import:

```
import Web3 from 'web3';
import * as eth from 'ethereumjs-tx';
const Tx = eth.Transaction;
```

Le variabili per configurare il contratto e completare correttamente le transazioni in *Ethereum* sono le seguenti:

```
    private web3 = new Web3(new Web3.providers.HttpProvider('
        https://ropsten.infura.io/v3/YOUR_INFURA_ID'));
    private adminAddress = 'ADMIN_ADDRESS';
    private privKey = environment.BC_PRIVATE_KEY;
    private contractAddress = ''CONTRACT_ADDRESS;
    private abiContract = 'CONTRACT_ABI'
    private myContract = new this.web3.eth.Contract(JSON.parse(
        this.abiContract), this.contractAddress);
```

web3 è un'istanza utilizzata per fornire un client eseguito tramite il provider Infura. *adminAddress* è l'indirizzo di chi ha effettuato il *deployment*, l'unico con l'autorità per

chiamare la funzione *setInfected*.
privKey rappresenta la chiave privata dell'admin.
contractAddress è l'indirizzo in cui risiede il contratto nella *blockchain*.
abiContract è il contratto in formato abi, necessario per creare l'instanza del contratto *myContract*, insieme all'indirizzo del contratto.

Per inviare la transazione in *Ethereum* è presente la seguente funzione, che prende in input i dati e effettua la transazione grazie alla funzione di web3js:

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

Infine, le seguenti righe di codice si occupano di creare i dati della transazione chiamando il metodo *setInfected* per poi inviare la transazione sfruttando la funzione *sendSigned*:

```
createInfected(infected: Infected) {

  const myData = this.myContract.methods.setInfected( infected.
      infected_id, true).encodeABI();

  this.web3.eth.getTransactionCount(this.addressFrom).then(
      txCount => {

    const txData = {
      nonce: this.web3.utils.toHex(txCount),
      gasLimit: this.web3.utils.toHex(GAS_LIMIT),
      gasPrice: this.web3.utils.toHex(GAS_PRICE),
      to: this.contractAddress,
      from: this.adminAddress,
      value: this.web3.utils.toHex(this.web3.utils.toWei('0', '
          wei')),
      data: myData
    };

    this.sendSigned(txData, (err, result) => {
      if (err) {
         return console.log('error', err);
      }
      console.log('sent', result);
    });
}
```

Dopo questa serie di operazioni, il codice identificativo della persona inserita tramite interfaccia web, viene segnalato come infetto in *blockchain* e, dall'applicazione mobile, gli utenti che hanno registrato un contatto con l'identificativo segnalato verranno avvertiti di conseguenza.

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

Le conoscenze acquisite durante lo stage sono molteplici, sia a livello teorico che pratico.

Blockchain
La prima parte del percorso è stata incentrata sullo studio della tecnologia *blockchain*, in tutte le sue sfaccettature. Essendo un campo mai affrontato durante il percorso universitario e non avendo particolari conoscenze pregresse, ho dedicato particolare attenzione allo studio delle *blockchain*, sia per interesse personale, che per comprendere al meglio i concetti generali su cui lo stage era focalizzato.

Ethereum e Solidity
La fase successiva ha riguardato lo studio della piattaforma *Ethereum* e di tutti gli strumenti utilizzati per lo sviluppo di *smart contract* per la *blockchain*. Grazie allo studio teorico delle *blockchain*, è stato facile comprendere il funzionamento della famosa piattaforma. Il linguaggio di programmazione utilizzato per gli *smart contract* è stato *Solidity*. A prima vista è risultato facile da comprendere, ma padroneggiare fino in fondo la programmazione nel contesto *blockchain* ha richiesto tempo, soprattutto per quanto riguarda l'ottimizzazione del codice.

Utilizzo smart contract in applicazione mobile e web
Dopo aver sviluppato lo *smart contract* per il *contact tracing*, mi è stato richiesto di integrarlo nelle applicazioni SyncTrace. Farlo è stato molto utile e formativo perché non mi sono limitato a sviluppare uno *smart contract*, ma ho anche imparato a sviluppare un'applicazione decentralizzata sfruttando la *blockchain*. Inoltre l'integrazione ha richiesto la collaborazione con il resto del team.

6.3 Valutazione personale

Il bilancio dello stage effettuato presso Sync Lab è senza dubbio positivo. Gli argomenti trattati mi hanno interessato sin dalla scelta del percorso e non hanno deluso le mie aspettative. La *blockchain* è una tecnologia che si sta enormemente sviluppando negli ultimi anni e questa esperienza è stata un'opportunità perfetta per approfondirla. Sono riuscito a comprendere i pregi e i difetti dello sviluppare un'applicazione decentralizzata al giorno d'oggi, quando conviene e quando no, al di là del caso d'uso affrontato nel

progetto. Sicuramente è una tecnologia in continua evoluzione e i prossimi anni possono rappresentare una svolta per questo campo. Sarà molto interessante osservare come il passaggio di *Ethereum* al *Proof of Stake* possa aumentare le opportunità e le potenzialità della *blockchain*.
Valuto positivamente l'esperienza anche per il rapporto con il mondo del lavoro. Lo stage rappresenta un'occasione per affacciarsi al contesto aziendale, completamente diverso da quello universitario. Purtroppo la pandemia ha costretto l'azienda a svolgere la maggior parte delle attività da remoto e questo ha certamente limitato il valore positivo che il tirocinio può portare. Ciononostante, ho apprezzato l'organizzazione aziendale anche in questo periodo, in particolar modo quando è stato possibile svolgere intere giornate lavorative in presenza.

Glossario

Algoritmo di consenso	Un algoritmo di consenso è un meccanismo attraverso il quale i nodi assicurano la validazione di un blocco nella rete. È fondamentale nelle blockchain perché la mancanza di un ente che convalidi le transazioni deve coincidere con la sicurezza che le transazioni avvengano correttamente e le regole del protocollo siano seguite. Gli algorimi di consenso più conosciuti sono il Proof of Work e il Proof of Stake.
Applicazioni decentralizzate	Un'applicazione decentralizzata è un programma eseguito su un sistema distribuito come le blockchain.
Best practices	Prassi che secondo l'esperienza personale o di studi abbia dimostrato di garantire il miglior risultato in determinate circostanze.
Big data	Tecniche e metodologie di analisi di grandi quantità di dati ovvero la capacità di estrapolare, analizzare e mettere in relazione un'enorme mole di dati eterogenei, strutturati e non strutturati, per scoprire i legami tra fenomeni diversi e prevedere quelli futuri. Si parla di big data quando si ha un insieme talmente grande e complesso di dati che richiede la definizione di nuovi strumenti e metodologie per estrapolare, gestire e processare informazioni entro un tempo ragionevole.
Bitcoin	Bitocoin è una blockchain creata nel 2009 da Satoshi Nakamoto che utilizza l'omonima criptovaluta. Utilizza un ledger distribuito fra i nodi per tenere traccia delle transazioni, validate tramite algoritmo Proof of Work.

Bluetooth LE

Il bluetooth LE è una tecnologia wireless con consumi energetici notevolmente ridotti rispetto al bluetooth classico, seppur con intervalli di comunicazione simili. Le sue caratteristiche si adattano a molteplici applicazioni che richiedono elevati utilizzi nel tempo del bluetooth, come per il contact tracing.

Cloud computing

Con il termine Cloud Computing (ing. "nuvola informatica") si indica un paradigma di erogazione di servizi offerti on demand da un fornitore ad un cliente finale attraverso la rete Internet.

Contact tracing

Il contact tracing è il tracciamento dei contatti nell'ambito della sanità pubblica, tramite il quale si identificano le persone che potrebbero essere venute a contatto con una persona infetta. Nell'ambito informatico con applicazione per il contact tracing si intende un programma tipicamente per smartphone che raccolga le informazioni relative ai contatti tra utenti.

Criptovaluta

La criptovaluta è una rappresentazione digitale di valore basata sulla crittografia.

Deploy

Termine con il quale si indica il caricamento di un contratto all'interno della rete Ethereum.

Distributed ledger

Il distributed ledger è un database condiviso e sincronizzato attraverso svariati nodi. Ogni nodo è autorizzato ad aggiornare il ledger in modo indipendente ma sotto il controllo consensuale degli altri. Non vi è un'autorità centrale come nei database tradizionali, ma l'incorrutibilità è garantita da un algoritmo di consenso.

Double spending

Il double spending è una potenziale criticità delle criptovalute per cui lo stesso token digitale viene speso più di una volta.

Ether

Criptovaluta utilizzata all'interno della blockchain Ethereum.

Ethereum	Piattaforma decentralizzata ideata da Vitalik Buterik nel 2015, famosa per la possibilità di eseguire smart contracts in blockchain.
Ethereum Virtual Machine	L'Ethereum Virtual Machine è la macchine virtuale per lo sviluppo e la gestione di smart contracts in Ethereum. Opera in modo separato dalla rete, ossia il codice gestito dalla Virtual Machine non ha accesso alla rete.
Fee	In ambito blockchain una fee è un pagamento richiesto per le operazioni effettuato che ha lo scopo di dare una ricompensa al miner che valida il blocco, in modo da sostenere il sistema.
Gas limit	Il gas limit è una soglia di sicurezza utilizzata per prevenire loop infiniti in blockchain; se un'operazione supera il gas limit impostato, la transazione fallisce, ma il lavoro effettuato dai miners viene comunque pagato.
Gas price	Il gas price è un valore in criptovaluta assegnato alla singola unità di gas; viene utilizzato per dare un valore monetario a un'operazione, in modo da pagare i miners.
Hash	L'hash è una funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza fissa. Viene utilizzzata in ambito blockchain per l'algoritmo Proof of Work: calcolare un'hash con delle caratteristiche definite a priori è un lavoro computazionale oneroso e viene utilizzato per validare i blocchi come prova del lavoro effettuato.
Indici di contatto	Nell'applicazione SyncTrace l'indice di contatto è un numero calcolato in funzione di distanza e tempo di contatto tra due persone, utilizzato per registrare il livello di contatto e il relativo rischio.

Information and Communication Technology	Con ICT si indica il settore legato allo sviluppo delle strutture internet e mobile, in particolare per ciò che riguarda gli aspetti di progettazione e realizzazione delle componenti fisiche o di quelle digitali. Più in generale, le ICT comprendono l'insieme dei metodi e delle tecniche utilizzate nella trasmissione, ricezione ed elaborazione di dati e informazioni digitali, ampiamente diffusesi a partire dalla cosiddetta Terza rivoluzione industriale: hardware,software e tecnologie ICT costituiscono oggi le tre componenti principali delsettore IT.
Internet of things	Nelle telecomunicazioni, Internet of things (internet degli oggetti) è il modo di riferirsi all'estensione di Internet al mondo degli oggetti e dei luoghi concreti. Al giorno d'oggi infatti sempre più oggetti comuni hanno la possibilità di interfacciarsi con il web, a partire da lavatrici, frigoriferi per arrivare a tutta la casa controllabile e accessibile da remoto.
Metamask	Metamask è un estensione browser che permette di interagire con la blockchain Ethereum. Mette a disposizione un wallet per depositare e inviare Ether nella rete.
Miner	In una blockchain con algoritmo di consenso Proof of Work, si definisce miner un nodo che risolve i problemi computazionali richiesti per la validazione di un blocco, dietro un compenso ricevuto a lavoro ultimato.
Permissioned	Una blockchain che viene così definita permette solo a utenti autenticati e autorizzati di accedere alla rete, eseguire delle transazioni o partecipare alla verifica e creazione di un nuovo blocco.
Permissionless	Una blockchain che viene così definita non richiede alcuna autorizzazione per poter accedere alla rete, eseguire delle transazioni o partecipare alla verifica e creazione di un nuovo blocco.

Proof of concept	Per Proof of Concept, abbreviato spesso in PoC, si intende una realizzazione incompleta o abbozzata di un determinato progetto o metodo, allo scopo di provarne la fattibilità o dimostrare la fondatezza di alcuni principi o concetti costituenti.
Proof of work	Algoritmo di consenso utilizzato da diverse criptovalute, come Bitcoin, Ethereum e Litecoin per raggiungere un accordo decentralizzato tra diversi nodi nel processo di aggiunta di un blocco specifico alla blockchain. Tale algoritmo obbliga i miners a risolvere dei problemi matematici estremamente complessi e computazionalmente difficili per poter aggiungere blocchi alla blockchain.
Repository	Ambiente di sistema informativo in cui vengono conservati e gestiti file, documenti e metadati relativi ad un'attività di progetto.
Token	In blockchain un token è un gettone virtuale emesso da un'organizzazione che rappresenta un'unità di valore.
Trello	Software gestionale in stile kanban che consente di lavorare in modo più organizzato e collaborativo.
Linguaggio Turing completo	Un linguaggio è detto Turing completo quando è in grado di eseguire qualunque programma che una macchina di Turing può esegiore, dato sufficiente tempo e memoria.
Turn over aziendale	Tasso di ricambio del personale, ovvero il flusso di persone in ingresso e in uscita da un'azienda.

UML

in ingegneria del software *UML, Unified Modeling Language* (ing. linguaggio di modellazione unificato) è un linguaggio di modellazione e specifica basato sul paradigma object-oriented. L'*UML* svolge un'importantissima funzione di "lingua franca" nella comunità della progettazione e programmazione a oggetti. Gran parte della letteratura di settore usa tale linguaggio per descrivere soluzioni analitiche e progettuali in modo sintetico e comprensibile a un vasto pubblico.

Wei

La più piccola unità in cui è possibile suddividere un Ether. Possiede numerosi multipli, come il gwei, molto utilizzato per comodità di espressione nel prezzo del gas.

Bibliografia

Riferimenti bibliografici

Buterik, Vitalik. *A next generation smart contract & decentralized application platform*. 2013.

Correia, Miguel. *From Byzantine Consensus to Blockchain Consensus*. 2019.

Nakamoto, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008.

Porat, Amitai et al. *Blockchain Consensus: An analysis of Proof-of-Work and its applications*. 2017.

Wood, Gavin. *Ethereum: a secure decentralized generalised transaction ledger*. 2014.

Xu, Hao et al. *BeepTrace: Blockchain-enabled Privacy-preserving Contact Tracing for COVID-19 Pandemic and Beyond*. 2020.

Siti web consultati

Come ridurre i consumi di gas. URL: https://medium.com/layerx/how-to-reduce-gas-cost-in-solidity-f2e5321e0395.

Documentazione di Ethereum. URL: https://ethereum.org/en/developers/.

Documentazione di Solidity. URL: https://solidity.readthedocs.io/en/v0.6.11/.

Documentazione di web3j. URL: https://docs.web3j.io/.

Documentazione di web3js. URL: https://web3js.readthedocs.io/en/v1.2.9/.

Ethereum 2.0. URL: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/.

Introduzione sullo sviluppo in android con web3j e infura. URL: https://medium.com/datadriveninvestor/an-introduction-to-ethereum-development-on-android-using-web3j-and-infura-763940719997.

Proof of Stake. URL: https://en.bitcoin.it/wiki/Proof_of_Stake.

Proof of work. URL: https://en.bitcoin.it/wiki/Proof_of_work.

Sito web di Sync Lab. URL: http://ww.synclab.it/.