Università degli Studi di Torino

Dipartimento di Informatica

Corso di Laurea in Informatica

Tesi di Laurea in Informatica

BLINC
Blockchain e Identità Digitale

Candidato:
Lorenzo Bersano

Relatori:
Prof. Claudio Schifanella
Prof. Alex Cordero

ANNO ACCADEMICO 2017–2018

Indice

Capitolo 1

Introduzione

1.1 Ambito del progetto

BLINC – Blockchain inclusiva per cittadinanze digitali è un progetto di ricerca industriale e sviluppo sperimentale emanato da Regione Piemonte a valere sui fondi POR FESR 2014-2020 Europei. Il progetto mira a realizzare una piattaforma Blockchain per la gestione di identità digitali, dati, transazioni di valore coinvolgendo la PA e gli operatori di servizi per i migranti. Tutti gli attori che entrano in contatto con gli utenti potranno inserire certificazioni: dal titolo di identità, al credito formativo, alla lettera di raccomandazione che il migrante potrà esibire a sua discrezione, preservandone la privacy e al contempo migliorando le potenzialità di costruzione di fiducia e inclusione sociale.

La tecnologia blockchain consente di passare da un internet dei dati ad un internet dei valori, promettendo un impatto di innovazione in ambito finanziario e commerciale paragonabile all'impatto che il web avuto sui modi di comunicare e acquisire informazioni. La posta in gioco riguarda industrie che generano oltre il 20% del PIL (stime Wedbush securities). Chiunque sarà in grado di creare coupon, titoli al portatore sistemi di pagamento, contratti automatici che regolano le relazioni tra diversi attori nella filiera produttiva e molto altro ancora. Lo sviluppo dell'internet of money è una tendenza già in atto visibile nel successo di strumenti come le gift card, i coupon, i circuiti di credito commerciale, punti fedeltà, sistemi di pagamento attraverso smartphone.

La blockchain permetterà un salto di qualità rendendo diversi circuiti interoperabili. Si può comprendere la blockchain in analogia con l'introduzione del protocollo SMTP per la posta elettronica, un'evoluzione dalle intranet aziendali all'internet aperta che conosciamo. L' internet delle cose e la diffu-

sione di dispositivi robotici sarà un altro importantissimo driver: operatori come IBM e Samsung prevedono he gli oggetti negozieranno tra di loro titoli per l'accesso a dati, energia e rapporti di cooperazione nelle loro funzioni. Altrettanto significativa è la capacità di sincronizzare le basi dati della pubblica amministrazione, anagrafe, catasto, fisco, documenti protocollati, ma anche informazioni protette come le registrazioni del sistema sanitario. La proposta presente si inserisce in questo filone di ricerca.

Tutto ciò si integra in un progetto che permette di sperimentare soluzioni originali ad un problema drammatico, di grande rilevanza politica e sociale, nonché esistenziale. Essere straniero, provenire da una cultura non europea, talvolta avere una storia di fuga da situazioni insostenibili o pericolose spesso porta a perdere l'identità formale e sociale costruita nel paese di origine, al tempo stesso la richiesta di informazione da parte dell'ambiente circostante è maggiore. Dimenticare i propri documenti può causare problemi, il carico burocratico e la frequenza presso gli uffici della PA sono più alti che per i cittadini italiani. Attraverso la Blockchain sarà possibile strutturare tecnologie per la fiducia, atte a colmare quel gap di informazione che frena i processi di inclusione dei migranti, senza portare a stigmatizzazioni o discriminazioni nel diritto alla privacy.

Il funzionamento dell'applicazione base proposta è semplice, un portadocumenti virtuale che contiene certificati auto-generati ad esempio a partire da documenti cartacei o per dichiarazione) accanto a documenti generati dei servizi privati e pubblici con cui il migrante viene in contatto. Il portadocumenti è accessibile da qualsiasi dispositivo, ma pensato per il mobile, con interfacce semplificate che rendono l'uso compatibile con scarse competenze digitali.

Un altro aspetto fondamentale del prodotto è la gestione granulare della privacy: il portadocumenti non può essere consultato nella sua interezza, ma, utilizzando tecnologie Blockchain pensate per i record sanitari, l'utente potrà decidere quali certificati ne fanno parte. L'iniziativa si colloca nel contesto della sharing economy ed il progetto valuterà possibili integrazioni con altre operazioni Blockchain condotte dall'Università di Torino, in particolar modo nel campo delle valute sociali locali e del sistema di protezione sociale, con il progetto Co-City riguardante il coinvolgimento diretto (patti di collaborazione) dei cittadini attivi nella generazione di servizi negli spazi urbani inutilizzati.

1.2 Descrizione dell'azienda

Consoft Sistemi è presente sul mercato ICT dal 1986 con sedi a Milano, Torino, Genova, Roma e Tunisi. Accanto alla capogruppo sono attive altre 4 società: CS InIT, specializzata nello scouting e distribuzione di soluzioni software, Consoft Consulting focalizzata sulla PA, Consoft Sistemi MEA e C&A Soft Consulting per espandere l'offerta della capogruppo nel mercato nord-africano e medio-orientale. Il Gruppo Consoft ha focalizzato la propria offerta su alcune aree tematiche, prevalentemente focalizzate sul tema della Digital Transformation nell'ambito delle quali è in grado di realizzare soluzioni "end to end" per i propri Clienti attraverso attività di consulenza, formazione, realizzazione di soluzioni integrate ed erogazione di servizi in insourcing/outsourcing.

Ha ottenuto la Certificazione ISO 27001 ed ha un Sistema di Gestione Qualità certificato UNI EN ISO 9001:2008. Tra le aree di specializzazione tecnologica annovera DevOps e Testing, Analytics & Big Data, Cyber Security e Internet of Things.

Consoft Sistemi è parte del CDA del Cluster Tecnologie per le Smart Cities & Communities Lombardia, è membro di Assolombarda ed Assintel (tramite CS_InIT) ed attiva nei progetti di innovazione proposti dagli Enti. Ha fatto parte dell'Osservatorio Internet of things e Osservatorio Big Data del MIP, è membro IOTitaly.

Consoft Sistemi come partner tecnologico collabora attivamente a progetti di ricerca sia regionali che nazionali ed europei con l'obiettivo di studiare e realizzare soluzioni che arricchiscano il mercato con ulteriori componenti ICT sviluppati a partire dalla realtà progettuale proposta, basati pertanto su un'esperienza che ne abbia già stimato il grado di fattibilità e sostenibilità economica.

Le attività di R&D inoltre, permettono di creare ulteriori contatti tra aziende di dimensioni diversificate, centri di ricerca, università ed operatori di settore per costruire un'offerta di servizi più completi e competitivi e per consentire l'utilizzo sinergico di risorse nell'ottica di un complessivo aumento di efficienza ed efficacia.

L'Innovazione sociale attraverso il miglioramento della qualità della vita è tra i temi di maggiore interesse di Consoft Sistemi ed è in questo ambito che si colloca il progetto su cui è stato condotto e sviluppato il tirocinio.

1.3 Descrizione della tecnologia e motivazione della scelta di Ethereum

Data la necessità di avere un sistema sicuro e trasparente di caricamento e validazione di informazioni (documenti, certificati, contratti), la blockchain è stata la scelta tecnologica più ovvia per BLINC.

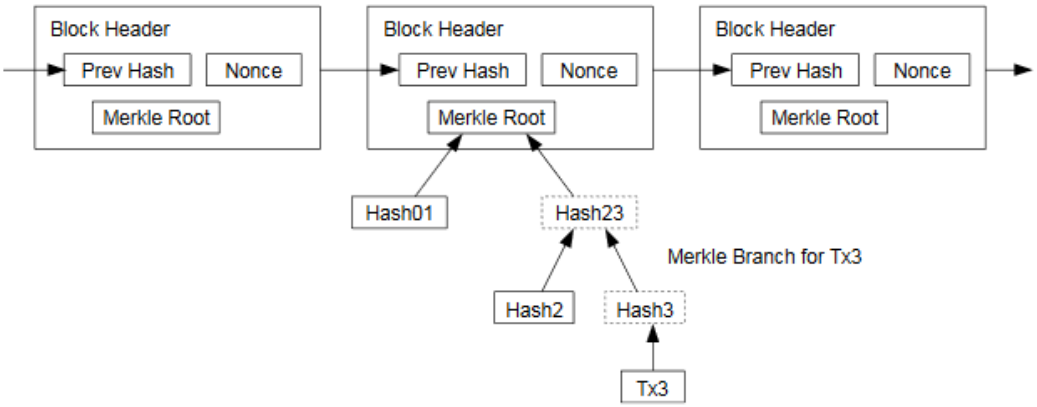


Figura 1.1: Struttura di una blockchain.

La blockchain è una struttura dati funzionante come un registro elettronico che garantisce l'immutabilità e la permanenza delle informazioni salvate su di essa. Come suggerisce il nome, la blockchain è una catena di blocchi marcati temporalmente sempre crescente, nella quale ogni blocco contiene queste informazioni:

- Un insieme di transazioni salvate in un Merkle Tree (un albero binario crittograficamente verificato)
- L'hash del blocco precedente (nel caso di Ethereum e Bitcoin un hash SHA-256)
- Il momento in cui il blocco è stato aggiunto alla catena (sottoforma di timestamp UNIX)
- Un nonce, ovvero un numero unico che, inserito in una funzione di hash assieme al resto delle informazioni del blocco, mi permette di ottenere un hash del blocco che è valido secondo dei requisiti (ad es. inizia con quattro zeri).

La sicurezza della blockchain è data dalle garanzie delle funzioni di hash: dato che l'header di ogni blocco contiene l'hash del blocco precedente, se un

qualsiasi blocco venisse modificato, il suo hash cambierebbe, e quindi invaliderebbe il collegamente al blocco successivo. Bisognerebbe quindi andare a ricalcolare l'hash di ogni blocco successivo, ma data la dimensione attuale di una blockchain come Bitcoin sarebbe computazionalmente impossibile.

La blockchain è nata con Bitcoin, sistema p2p per una valuta digitale creato da Satoshi Nakamoto basato su una blockchain distribuita e decentralizzata. Distribuzione e decentralizzazione sono fondamentali perché rendono la blockchain di fatto incensurabile e senza un unico point-of-failure. Chiunque può unirsi alla rete Bitcoin: è sufficiente eseguire un client Bitcoin che crea un wallet, ovvero una chiave privata, una chiave pubblica ed un indirizzo Bitcoin, e connette il dispositivo su cui è installato come nodo della rete.

Ethereum prende tutte le caratteristiche di Bitcoin e le amplia: rende la sua blockchain facilmente programmabile tramite smart contract. Uno smart contract è un programma scritto in un linguaggio di programmazione ad alto livello come Solidity (simil-JavaScript) o Vyper (simil-Python), compilato in bytecode e distribuito sulla blockchain tramite una transazione. Gli smart contract sono eseguiti in un ambiente isolato ed indipendente (e quindi più sicuro) chiamato Smart Contract Execution Engine, che nel caso di Ethereum prende il nome di EVM (Ethereum Virtual Machine). Esattamente come in ogni blockchain distribuita e decentralizzata ogni nodo della rete possiede una copia della blockchain stessa (e quindi ogni transazione), in Ethereum ogni nodo esegue il codice degli smart contract: questa replicazione rende molto lenta l'esecuzione delle istruzioni, ma permette lo sviluppo di applicazioni che necessitano di trasparenza, sicurezza ed immutabilità.

1.3.1 Perché Ethereum per BLINC?

Nonostante la giovinezza del progetto, Ethereum si è già imposto come standard *de-facto* per lo sviluppo di applicazioni decentralizzate basate su smart contract (è stato il primo a portare il concetto di smart contract su architetture basate su blockchain, poi è stato seguito da altre architetture come Stellar e NEO) e su un frontend che utilizza librerie come web3.js per comunicare con la rete Ethereum, è supportato dalla più grande comunità open source in ambito blockchain ed è fornito di un grande numero di strumenti per velocizzare il workflow di sviluppo di *dApp* (Decentralized Applications), tra cui Truffle, che facilita e automatizza task di compilazione e migrazione di contratti su diversi tipi di blockchain, e Ganache, una simulazione in JavaScript di una blockchain locale che, in quanto non c'è bisogno di minare le transazioni, è molto veloce e quindi adatta allo sviluppo e al testing di smart contract.

Essendo quindi un sistema stabile, ben supportato e facilmente integrabile con metodi di sviluppo tradizionali grazie alle librerie in JavaScript costantemente aggiornate, Ethereum permette di rendere BLINC, oltre che a un progetto di ricerca, un prodotto vendibile.

1.4 Architettura generale del progetto

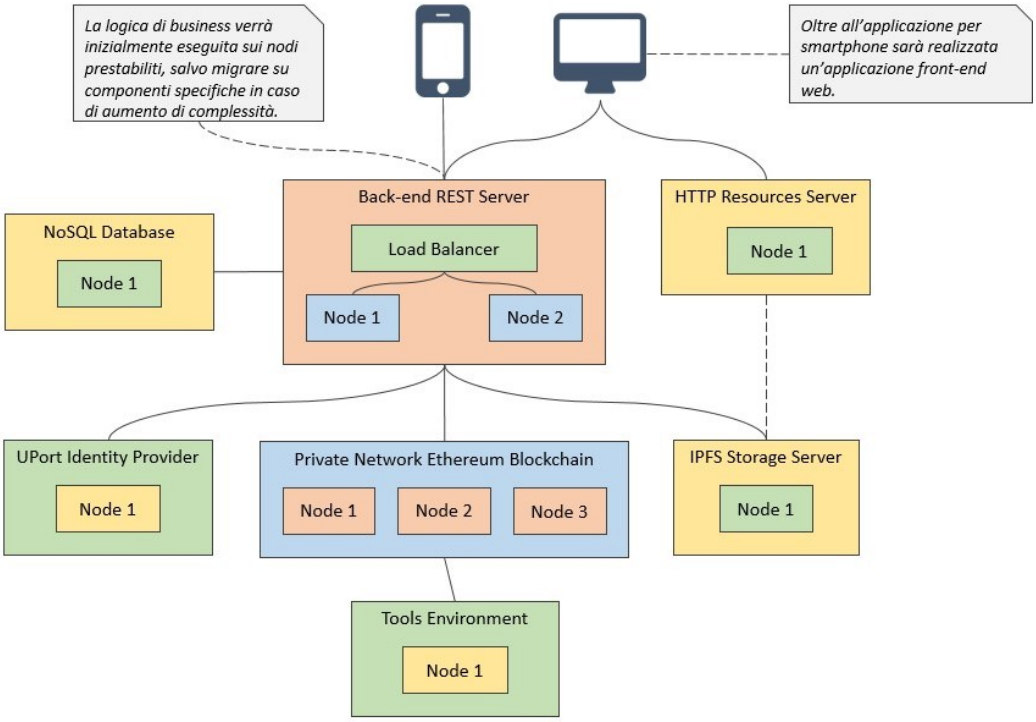


Figura 1.2: Diagramma dell'architettura di BLINC.

L'architettura è composta da (da sinistra a destra, dall'alto in basso):

- Applicazione mobile (native Android ed iOS) e web app: servono ai migranti ed agli operatori per interfacciarsi ai servizi di BLINC. comunicano con i server REST e di risorse tramite chiamate ad API
- Database NoSQL (MongoDB): necessario per memorizzare informazioni e wallet criptati degli utenti
- Server di back-end (Loopback, framework di creazione di API basato su NodeJS): espone gli endpoint API richiamabili dai client, contiene la business logic e accede alla rete Ethereum per interagire con le identità

dei migranti e effettuare dichiarazioni ed endorsement sulle generalità e documenti da essi caricati

- HTTP Resources Server: necessario per servire risorse statiche (immagini, documenti...) ai client
- uPort Identity Provider: insieme di *smart contract* uPort distribuiti sui nodi Ethereum sottostanti che rendono possibile la creazione e gestione di identità e di attestazioni relative ad esse.
- Blockchain Privata Ethereum: insieme di nodi che eseguono istanze del client Ethereum Geth.
- IPFS Storage Server: un nodo IPFS, sistema di storage decentralizzato, necessario per salvare i documenti cifrati degli utenti.
- Tools environment: strumenti di monitoraggio della chain privata Ethereum citata in precedenza, in particolare un *block explorer* ed un *netstats*, che permettono rispettivamente di visualizzare le transazioni incluse per blocco e di avere una visione di insieme sulla rete Ethereum privata, tra cui la difficoltà della rete, i nodi che partecipano ad essa e l'uptime.

1.4.1 Soluzione tecnologica

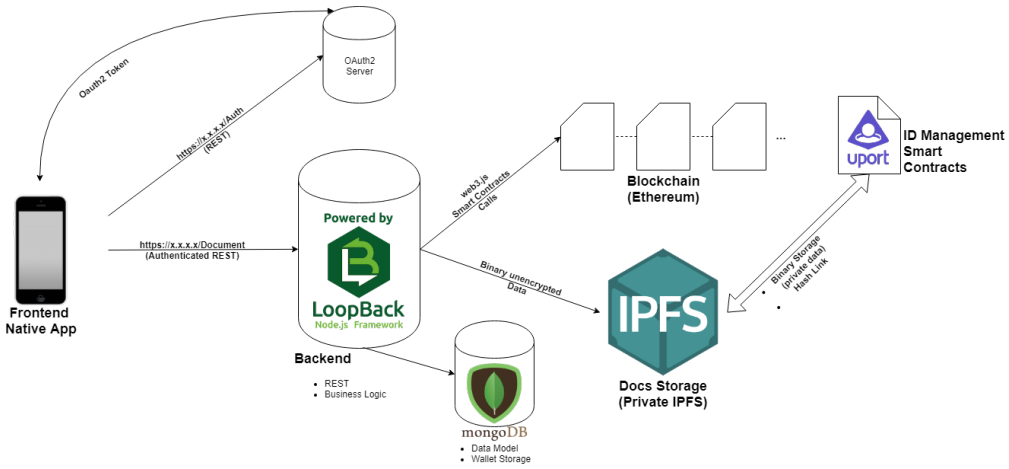


Figura 1.3: Soluzione tecnologica per BLINC.

Data l'immaturità del SDK di uPort che non ha permesso il mantenimento del wallet del migrante sul proprio telefono, la prima versione dell'architettura e di flusso di accesso ai servizi include elementi provvisori come l'OAuth

server (basato su login tramite e-mail e password e centralizzato, entrambe cose che si vorrebbero evitare) e il MongoDB (che contiene i wallet degli utenti criptati). In questa prima versione il flusso d'interazione base è quindi questo:

Utente registrato

1. L'utente apre l'applicazione di BLINC, inserisce e-mail/nome utente e password che vengono inviati all'OAuth server: se e-mail/nome utente esistono e la password associata è corretta viene restituito un token di autorizzazione tramite il quale l'utente potrà accedere alle API protette del backend.

2. Grazie al token ricevuto, l'utente può accedere innanzitutto alla propria identità uPort, salvata su database, per caricare documenti e verificare attestazioni sulle proprie credenziali e sui propri documenti.

Utente non registrato

Al primo accesso all'app BLINC il migrante inserisce informazioni base (nome, cognome, telefono...), la propria e-mail ed una password. Queste ultime vengono salvate sull'OAuth server per i login successivi, l'identità viene creata con le informazioni inserite in precedenza tramite le librerie apposite di uPort e salvata su MongoDB, dove viene anche creato, criptato e salvato il wallet Ethereum dell'utente, composto da una coppia di chiavi ed un indirizzo Ethereum, ricavato dalla chiave pubblica.

Capitolo 2

Identità digitale

2.1 Analisi dei requisiti

Come da requisiti, l'obbiettivo principale di BLINC è quello di creare un portadocumenti digitale decentralizzato i cui documenti inseriti rimangono sotto il controllo del proprietario. Quest'ultimo requisito introduce il concetto di **sovranità del dato** (o self-sovereign identity), ovvero il totale controllo della propria identità digitale senza dover dipendere da terzi come Facebook o SPID per la Pubblica Amministrazione.

La self-sovereign identity è soltanto l'ultima di una serie di evoluzioni che la gestione dell'identità digitale ha subìto nel tempo: inizialmente si aveva il modello a silos, ovvero la creazione di credenziali diverse per ogni servizio di cui un utente fruisce.

Successivamente si è passati ad un modello federato, nel quale le credenziali per un determinato servizio/sistema vengono riconosciute ed accettate anche in altri sistemi. Esempi di questo modello applicato in generale sono Facebook e Google Login, tramite i quali si può accedere ai siti/applicazioni che li supportano utilizzando le proprie credenziali Facebook/Google, mentre SPID è un esempio di un'applicazione a livello di Pubblica Amministrazione di questo modello.

La self-sovereign identity non sarebbe mai stata possibile con i tradizionali sistemi di Identity Providing centralizzati, ma con l'avvento della blockchain si è finalmente potuto iniziare ad esplorare soluzioni di sovranità del dato grazie soprattutto a due importanti proprietà: la pseudonimizzazione e la decentralizzazione.

La prima è fondamentale anche in ottica GDPR ed è data dall'intrinseca capacità della blockchain di mantenere riferimenti ad informazioni tramite il loro hash, la seconda è importante in quanto un identity provider basato su

blockchain è sempre disponibile al contrario degli identity provider centralizzati che, se non disponibili, impossibilitano l'accesso a servizi. Alcune aziende stanno implementando soluzioni di self-sovereign identity, e per BLINC sono state esaminate e valutate le seguenti:

2.2 Elenco dei prodotti e delle tecnologie disponibili

- **uPort**: Sviluppato sotto l'egida di ConsenSys, uPort è un sistema per la self-sovereign identity basato su Ethereum.

- **Civic**: Sviluppato dall'omonima azienda, Civic è una piattaforma composta da smart contract ed un token per lo scambio di valore basata su Rootstock, una side-chain di Bitcoin che permette l'esecuzione di smart contract. Civic fornisce una piattaforma sicura di self-sovereign identity accessibile dagli utenti tramite la loro app, che funge da wallet per l'identità. Civic ed i suoi identity partner possono fare una richiesta di credenziali all'utente, che può accettare o respingere, tramite un codice QR. Nel sistema Civic ci sono tre diversi soggetti: i Validatori, ovvero istituzioni finanziarie, entità governative e aziende che hanno la possibilità di validare l'identità di singoli o di altre aziende che prendono parte al sistema Civic come Utenti tramite delle transazioni di approvazione sulla blockchain chiamate attestazioni. Una attestazione è in pratica l'hash di un'informazione dell'identità più metadati relativi ad essa utilizzabile dai Fornitori di servizi per erogare servizi agli utenti abilitati. I fornitori di servizi possono acquistare le attestazioni ai validatori tramite smart contract pagando con i token del sistema Civic.

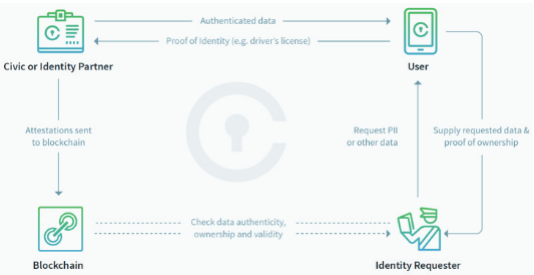


Figura 2.1: Diagramma di funzionamento di Civic.

- **SelfKey**: Sviluppato dalla SelfKey Foundation, SelfKey è un sistema basato su Blockchain Ethereum composto da un wallet personale per il possessore dell'identità, un marketplace di prodotti e servizi, un protocollo basato su JSON-LD ed un token per scambiare valore. L'identità dell'utente è salvata localmente sul suo smartphone sottoforma di wallet (coppia di chiavi), al quale vengono associate delle dichiarazioni sugli attributi dell'utente (data di nascita, nome, lavoro...) che possono essere verificate da terzi (banche, organizzazione...) in modo da poter accedere a determinati servizi. Con questo metodo si riduce la quantità di dati condivisi alle terze parti allo stretto indispensabile.

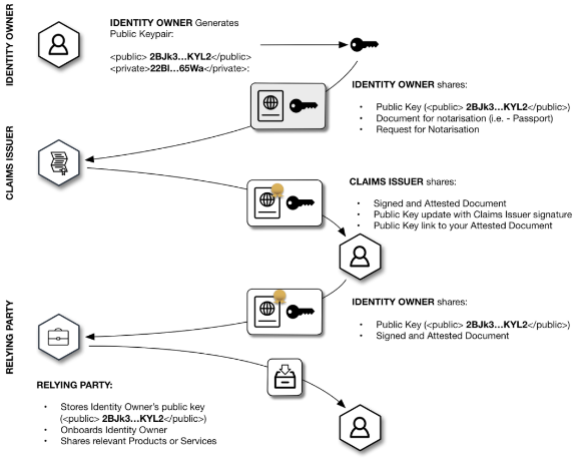


Figura 2.2: Diagramma di funzionamento di SelfKey.

2.3 Descrizione della tecnologia scelta e motivazione

uPort è un insieme di protocolli, librerie e smart contract che creano uno strato interoperabile di identità, che possono aggiungere, togliere e verificare attributi ed attestazioni a sé stesse ed alle altre identità.

Un'identità in uPort è formata da:

- Un **MNID** (Multi Network IDentifier), che è un oggetto JSON codificato in base58 formato da un campo network, ovvero l'ID della chain Ethereum su cui si trova l'account (ad esempio 0x1 per la mainnet) e un campo address il cui valore è un indirizzo Ethereum.

- Una chiave privata necessaria per firmare transazioni da inviare alla blockchain
- Una chiave pubblica inserita nel proprio DID Document salvato su IPFS e sullo smart contract Registry

uPort prevede la gestione di dati sia off-chain che on-chain. Per quanto riguarda la parte on-chain uPort prevede un insieme di smart contract per la gestione delle identità che sono:

- **Proxy**: uno per ogni identità, funge da rappresentante on-chain di essa. Ogni transazione che viene fatta sulla blockchain viene fatta dallo smart contract Proxy e non direttamente dal wallet dell'identità, permettendo di conseguenza di mantenere il possesso della propria identità digitale anche nel caso di smarrimento del device su cui è installato il wallet uPort.

```
1  // Proxy.sol
2  pragma solidity 0.4.15;
3  import "./libs/Owned.sol";
4
5
6  contract Proxy is Owned {
7      event Forwarded (address indexed destination, uint
           value, bytes data);
8      event Received (address indexed sender, uint value);
9
10     function () payable { Received(msg.sender, msg.value)
           ; }
11
12     function forward(address destination, uint value,
           bytes data) public onlyOwner {
13         require(executeCall(destination, value, data));
14         Forwarded(destination, value, data);
15     }
16
17     // copied from GnosisSafe
18     // https://github.com/gnosis/gnosis-safe-contracts/
           blob/master/contracts/GnosisSafe.sol
19     function executeCall(address to, uint256 value, bytes
           data) internal returns (bool success) {
20         assembly {
21             success := call(gas, to, value, add(data, 0
                   x20), mload(data), 0, 0)
22         }
23     }
24 }
```

- **IdentityManager**: uno per chain, funge da Factory degli smart contract Proxy e ne tiene traccia, associando, oltre all'identità vera e propria associata al Proxy, altre identità fidate che permettono il recupero del possesso del proprio contract Proxy in caso di smarrimento del wallet.

```
1  // IdentityManager.sol
2  pragma solidity 0.4.15;
3  import "./Proxy.sol";
4
5
6  contract IdentityManager {
7      uint adminTimeLock;
8      uint userTimeLock;
9      uint adminRate;
10
11     event LogIdentityCreated(
12         address indexed identity,
13         address indexed creator,
14         address owner,
15         address indexed recoveryKey);
16
17     event LogOwnerAdded(
18         address indexed identity,
19         address indexed owner,
20         address instigator);
21
22     event LogOwnerRemoved(
23         address indexed identity,
24         address indexed owner,
25         address instigator);
26
27     event LogRecoveryChanged(
28         address indexed identity,
29         address indexed recoveryKey,
30         address instigator);
31
32     event LogMigrationInitiated(
33         address indexed identity,
34         address indexed newIdManager,
35         address instigator);
36
37     event LogMigrationCanceled(
38         address indexed identity,
39         address indexed newIdManager,
40         address instigator);
41
42     event LogMigrationFinalized(
43         address indexed identity,
```

```
44          address indexed newIdManager,
45          address instigator);
46
47      mapping(address => mapping(address => uint)) owners;
48      mapping(address => address) recoveryKeys;
49      mapping(address => mapping(address => uint)) limiter;
50      mapping(address => uint) public migrationInitiated;
51      mapping(address => address) public
            migrationNewAddress;
52
53      modifier onlyOwner(address identity) {
54          require(isOwner(identity, msg.sender));
55          _;
56      }
57
58      modifier onlyOlderOwner(address identity) {
59          require(isOlderOwner(identity, msg.sender));
60          _;
61      }
62
63      modifier onlyRecovery(address identity) {
64          require(recoveryKeys[identity] == msg.sender);
65          _;
66      }
67
68      modifier rateLimited(address identity) {
69          require(limiter[identity][msg.sender] < (now -
                adminRate));
70          limiter[identity][msg.sender] = now;
71          _;
72      }
73
74      modifier validAddress(address addr) { //protects
            against some weird attacks
75          require(addr != address(0));
76          _;
77      }
78
79      /// @dev Contract constructor sets initial timelock
            limits
80      /// @param _userTimeLock Time before new owner added
            by recovery can control proxy
81      /// @param _adminTimeLock Time before new owner can
            add/remove owners
82      /// @param _adminRate Time period used for rate
            limiting a given key for admin functionality
83      function IdentityManager(uint _userTimeLock, uint
            _adminTimeLock, uint _adminRate) {
84          require(_adminTimeLock >= _userTimeLock);
```

```
85          adminTimeLock = _adminTimeLock;
86          userTimeLock = _userTimeLock;
87          adminRate = _adminRate;
88      }
89
90      /// @dev Creates a new proxy contract for an owner
            and recovery
91      /// @param owner Key who can use this contract to
            control proxy. Given full power
92      /// @param recoveryKey Key of recovery network or
            address from seed to recovery proxy
93      /// Gas cost of 289,311
94      function createIdentity(address owner, address
            recoveryKey) public validAddress(recoveryKey) {
95          Proxy identity = new Proxy();
96          owners[identity][owner] = now - adminTimeLock; //
                This is to ensure original owner has full
                power from day one
97          recoveryKeys[identity] = recoveryKey;
98          LogIdentityCreated(identity, msg.sender, owner,
                recoveryKey);
99      }
100
101     /// @dev Creates a new proxy contract for an owner
            and recovery and allows an initial forward call
            which would be to set the registry in our case
102     /// @param owner Key who can use this contract to
            control proxy. Given full power
103     /// @param recoveryKey Key of recovery network or
            address from seed to recovery proxy
104     /// @param destination Address of contract to be
            called after proxy is created
105     /// @param data of function to be called at the
            destination contract
106     function createIdentityWithCall(address owner,
            address recoveryKey, address destination, bytes
            data) public validAddress(recoveryKey) {
107         Proxy identity = new Proxy();
108         owners[identity][owner] = now - adminTimeLock; //
                This is to ensure original owner has full
                power from day one
109         recoveryKeys[identity] = recoveryKey;
110         LogIdentityCreated(identity, msg.sender, owner,
                recoveryKey);
111         identity.forward(destination, 0, data);
112     }
113
114     /// @dev Allows a user to transfer control of
            existing proxy to this contract. Must come through
```

```
        proxy
115     /// @param owner Key who can use this contract to
            control proxy. Given full power
116     /// @param recoveryKey Key of recovery network or
            address from seed to recovery proxy
117     /// Note: User must change owner of proxy to this
            contract after calling this
118     function registerIdentity(address owner, address
            recoveryKey) public validAddress(recoveryKey) {
119         require(recoveryKeys[msg.sender] == 0); // Deny
                any funny business
120         owners[msg.sender][owner] = now - adminTimeLock;
                // This is to ensure original owner has full
                power from day one
121         recoveryKeys[msg.sender] = recoveryKey;
122         LogIdentityCreated(msg.sender, msg.sender, owner,
                recoveryKey);
123     }
124
125     /// @dev Allows a user to forward a call through
            their proxy.
126     function forwardTo(Proxy identity, address
            destination, uint value, bytes data) public
            onlyOwner(identity) {
127         identity.forward(destination, value, data);
128     }
129
130     /// @dev Allows an olderOwner to add a new owner
            instantly
131     function addOwner(Proxy identity, address newOwner)
            public onlyOlderOwner(identity) rateLimited(
            identity) {
132         require(!isOwner(identity, newOwner));
133         owners[identity][newOwner] = now - userTimeLock;
134         LogOwnerAdded(identity, newOwner, msg.sender);
135     }
136
137     /// @dev Allows a recoveryKey to add a new owner with
            userTimeLock waiting time
138     function addOwnerFromRecovery(Proxy identity, address
            newOwner) public onlyRecovery(identity)
            rateLimited(identity) {
139         require(!isOwner(identity, newOwner));
140         owners[identity][newOwner] = now;
141         LogOwnerAdded(identity, newOwner, msg.sender);
142     }
143
144     /// @dev Allows an owner to remove another owner
            instantly
```

```
145     function removeOwner(Proxy identity, address owner)
            public onlyOlderOwner(identity) rateLimited(
            identity) {
146         // an owner should not be allowed to remove
                itself
147         require(msg.sender != owner);
148         delete owners[identity][owner];
149         LogOwnerRemoved(identity, owner, msg.sender);
150     }
151
152     /// @dev Allows an owner to change the recoveryKey
            instantly
153     function changeRecovery(Proxy identity, address
            recoveryKey) public
154         onlyOlderOwner(identity)
155         rateLimited(identity)
156         validAddress(recoveryKey)
157     {
158         recoveryKeys[identity] = recoveryKey;
159         LogRecoveryChanged(identity, recoveryKey, msg.
                sender);
160     }
161
162     /// @dev Allows an owner to begin process of
            transfering proxy to new IdentityManager
163     function initiateMigration(Proxy identity, address
            newIdManager) public
164         onlyOlderOwner(identity)
165         validAddress(newIdManager)
166     {
167         migrationInitiated[identity] = now;
168         migrationNewAddress[identity] = newIdManager;
169         LogMigrationInitiated(identity, newIdManager, msg
                .sender);
170     }
171
172     /// @dev Allows an owner to cancel the process of
            transfering proxy to new IdentityManager
173     function cancelMigration(Proxy identity) public
            onlyOwner(identity) {
174         address canceledManager = migrationNewAddress[
                identity];
175         delete migrationInitiated[identity];
176         delete migrationNewAddress[identity];
177         LogMigrationCanceled(identity, canceledManager,
                msg.sender);
178     }
179
180     /// @dev Allows an owner to finalize migration once
```

```
        adminTimeLock time has passed
181     /// WARNING: before transfering to a new address,
        make sure this address is "ready to recieve" the
        proxy.
182     /// Not doing so risks the proxy becoming stuck.
183     function finalizeMigration(Proxy identity) public
        onlyOlderOwner(identity) {
184         require(migrationInitiated[identity] != 0 &&
            migrationInitiated[identity] + adminTimeLock <
             now);
185         address newIdManager = migrationNewAddress[
            identity];
186         delete migrationInitiated[identity];
187         delete migrationNewAddress[identity];
188         identity.transfer(newIdManager);
189         delete recoveryKeys[identity];
190         // We can only delete the owner that we know of.
            All other owners
191         // needs to be removed before a call to this
            method.
192         delete owners[identity][msg.sender];
193         LogMigrationFinalized(identity, newIdManager, msg
            .sender);
194     }
195
196     function isOwner(address identity, address owner)
        public constant returns (bool) {
197         return (owners[identity][owner] > 0 && (owners[
            identity][owner] + userTimeLock) <= now);
198     }
199
200     function isOlderOwner(address identity, address owner
        ) public constant returns (bool) {
201         return (owners[identity][owner] > 0 && (owners[
            identity][owner] + adminTimeLock) <= now);
202     }
203
204     function isRecovery(address identity, address
        recoveryKey) public constant returns (bool) {
205         return recoveryKeys[identity] == recoveryKey;
206     }
207 }
```

- **UportRegistry**: uno per chain, serve per aggiungere attributi in forma chiave:valore alle identità, in particolare l'hash del DID Document caricato su IPFS.

```
1 //UportRegistry.sol
2 pragma solidity 0.4.8;
```

```
3
4  contract UportRegistry{
5    uint public version;
6    address public previousPublishedVersion;
7    mapping(bytes32 => mapping(address => mapping(address
        => bytes32))) public registry;
8
9    function UportRegistry(address
        _previousPublishedVersion) {
10     version = 3;
11     previousPublishedVersion = _previousPublishedVersion;
12   }
13
14   event Set(
15     bytes32 indexed registrationIdentifier,
16     address indexed issuer,
17     address indexed subject,
18     uint updatedAt);
19
20   //create or update
21   function set(bytes32 registrationIdentifier, address
        subject, bytes32 value){
22       Set(registrationIdentifier, msg.sender, subject,
            now);
23       registry[registrationIdentifier][msg.sender][
            subject] = value;
24   }
25
26   function get(bytes32 registrationIdentifier, address
        issuer, address subject) constant returns(bytes32){
27       return registry[registrationIdentifier][issuer][
            subject];
28   }
29 }
```

- **EthereumClaimsRegistry**: uno per chain, serve per aggiungere dichiarazioni ed attestazioni alle identità.

```
1  //EthereumClaimsRegistry.sol
2  pragma solidity 0.4.19;
3
4
5  /// @title Ethereum Claims Registry - A repository
       storing claims issued
6  ///         from any Ethereum account to any other
       Ethereum account.
7  contract EthereumClaimsRegistry {
8
```

```
9       mapping(address => mapping(address => mapping(bytes32
            => bytes32))) public registry;
10
11      event ClaimSet(
12          address indexed issuer,
13          address indexed subject,
14          bytes32 indexed key,
15          bytes32 value,
16          uint updatedAt);
17
18      event ClaimRemoved(
19          address indexed issuer,
20          address indexed subject,
21          bytes32 indexed key,
22          uint removedAt);
23
24      /// @dev Create or update a claim
25      /// @param subject The address the claim is being
            issued to
26      /// @param key The key used to identify the claim
27      /// @param value The data associated with the claim
28      function setClaim(address subject, bytes32 key,
            bytes32 value) public {
29          registry[msg.sender][subject][key] = value;
30          ClaimSet(msg.sender, subject, key, value, now);
31      }
32
33      /// @dev Create or update a claim about yourself
34      /// @param key The key used to identify the claim
35      /// @param value The data associated with the claim
36      function setSelfClaim(bytes32 key, bytes32 value)
            public {
37          setClaim(msg.sender, key, value);
38      }
39
40      /// @dev Allows to retrieve claims from other
            contracts as well as other off-chain interfaces
41      /// @param issuer The address of the issuer of the
            claim
42      /// @param subject The address to which the claim was
            issued to
43      /// @param key The key used to identify the claim
44      function getClaim(address issuer, address subject,
            bytes32 key) public constant returns(bytes32) {
45          return registry[issuer][subject][key];
46      }
47
48      /// @dev Allows to remove a claims from the registry.
49      ///      This can only be done by the issuer or the
```

```
        subject of the claim.
50      /// @param issuer The address of the issuer of the
            claim
51      /// @param subject The address to which the claim was
             issued to
52      /// @param key The key used to identify the claim
53      function removeClaim(address issuer, address subject,
            bytes32 key) public {
54          require(msg.sender == issuer || msg.sender ==
                subject);
55          require(registry[issuer][subject][key] != 0);
56          delete registry[issuer][subject][key];
57          ClaimRemoved(msg.sender, subject, key, now);
58      }
59  }
```

Per quanto riguarda invece le interazioni off-chain uPort utilizza JWT (JSON Web Token) firmati, che possono essere utilizzati per:

- Ricevere/fare richieste di condivisione di credenziali da/ad altre identità uPort
- Inviare/ricevere attestazioni a/da altre identità uPort

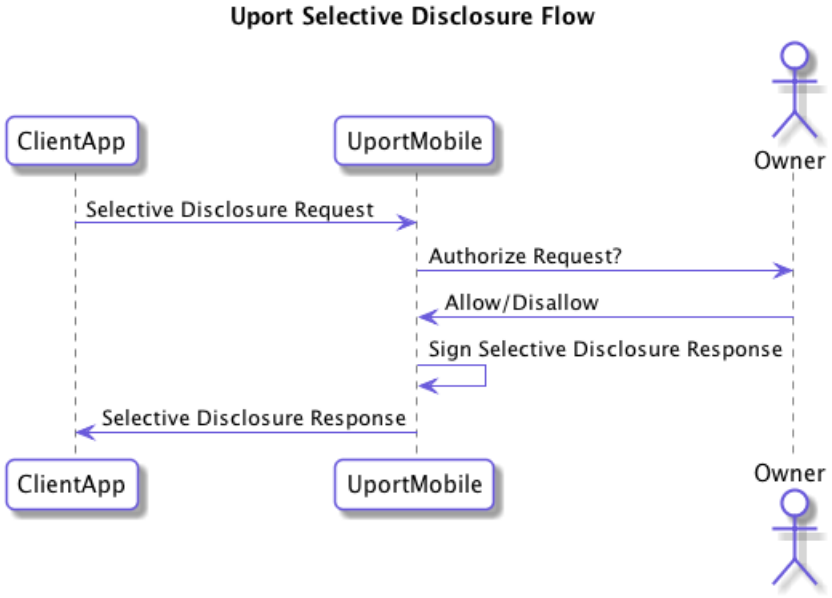
I JWT sono una maniera standardizzata dalla RFC per scambiare informazioni tra parti sotto forma di oggetto JSON firmato ed eventualmente criptato. Sono utilizzati principalmente in due ambiti:

1. **Autorizzazione**: una volta che un utente ha acceduto ad un sito/servizio, un JWT viene aggiunto ad ogni richiesta successiva per consentire l'accesso a risorse protette (API, ad esempio).
2. **Scambio di informazioni**: ambito d'interesse per uPort, un JWT è particolarmente indicato per scambiare informazioni per due motivi: è un formato di dimensioni ridotte adatto ad essere aggiunto ad un URL ed essendo un oggetto JSON firmato si ha la sicurezza che le informazioni ricevute sono quelle inviati inizialmente e che non sono state modificate nel frattempo.

uPort, come tutte le soluzioni di self-sovereign identity, inverte totalmente il paradigma dell'identità online: se prima erano i siti web e servizi su cui ci si registrava ad avere sotto controllo tutte le nostre informazioni, avendo quindi la possibilità di rivenderle al miglior offerente per scopi pubblicitari od altro, con uPort l'identità e le informazioni associate ad essa sono totalmente sotto il controllo del proprietario dell'identità e sono le terze parti a dover

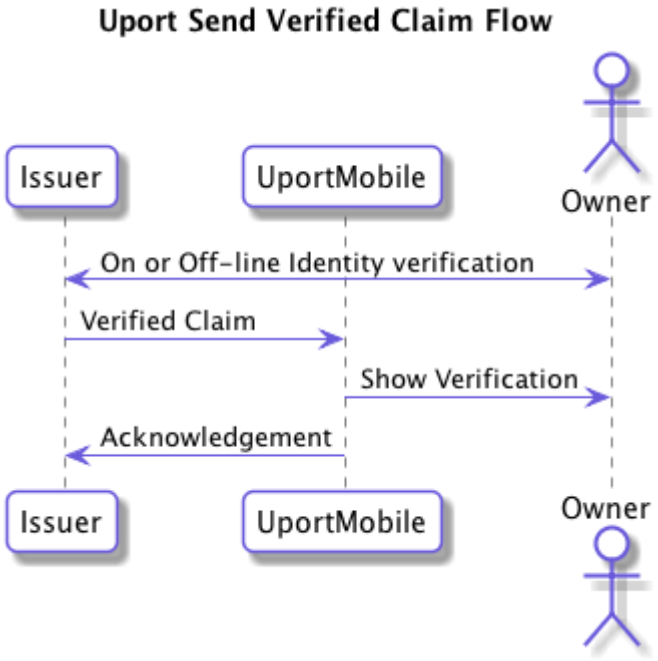
richiedere la informazioni necessarie all'utente, che di conseguenza ha anche un maggiore controllo su quali e quante informazioni fornisce.

Questo processo di rilascio di informazioni personali a terzi in uPort si chiama **Selective Disclosure Flow**, e segue questo flusso:



La Selective Disclosure Request fatta dal servizio che richiede le credenziali all'utente non è altro che un JWT che segue uno schema ben formato inviato ad un opportuno endpoint gestito da uPort. Una volta che la richiesta è arrivata al server Chasqui (che gestisce lo scambio di JWT tra app uPort e normali web app o DApp) la web app/DApp fa polling sul server fino a quando l'utente non approva o nega l'accesso alle informazioni richieste: in caso di approvazione la web app/DApp ha finalmente accesso alle informazioni richieste.

Molto spesso, come nel caso di BLINC, le informazioni vengono richieste per poi essere attestate, ad esempio un servizio di e-ticketing potrebbe richiedere nome e cognome dell'acquirente per attestare che quell'identità ha effettivamente acquistato un biglietto. Per questo motivo in congiunzione al flusso di richiesta di informazioni uPort prevede il Send Verification Flow, ovvero il processo di attestazione di attributi e credenziali dell'identità.



Il Verified Claim creato dall'issuer (che è l'identità uPort che ha creato l'attestazione) è un JWT che segue uno schema ben formato inviato allo stesso server Chasqui citato in precedenza e segue un flusso molto simile a quello della richiesta di credenziali.

2.3.1 Perché è stato scelto uPort?

Tra tutte le alternative disponibili è stato scelto uPort perché tra tutti i sistemi di gestione dell'identità digitale e di sovranità del dato è quello con un grado di maturità e supporto di sviluppatori e community maggiore. Inoltre, il fatto che sia costruito su Ethereum rende semplice l'interoperabilità con smart contract sviluppati esternamente a uPort come ad esempio, nel caso di BLINC, quello di gestione dei documenti dei migranti.

2.4 Integrazione nel progetto BLINC

All'interno del progetto uPort è stato utilizzato per la gestione di dichiarazioni ed endorsement sui migranti e la creazione di uPort Identity associate ad essi.

2.4.1 Gestione di dichiarazioni ed endorsement

Per quanto riguarda questo caso d'uso di BLINC ci si riferisce a questo flusso di interazione:

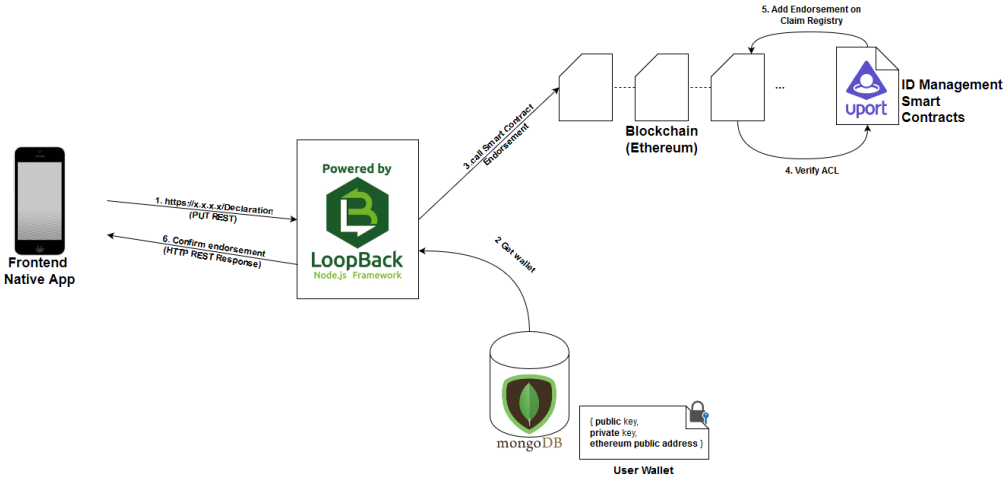


Figura 2.3: Flusso di gestione delle dichiarazioni sull'utente.

Alla rispettiva chiamata API il backend di BLINC va a recuperare dal database il wallet associato al migrante che si è autenticato e firma una transazione che ha come destinatario lo smart contract ethereum-claims-registry, che permette di associare delle dichiarazioni da parte di un indirizzo Ethereum verso un altro in forma chiave:valore.

Implementazione

```
/**
* Gets all the declaration made and returns them.
* @param {string} id the id of the owner of the declarations.
*
* @returns {Array} declarations an array of declarations.
*/
Declaration.getAllDeclarations = (id, cb) => {
    Declaration.app.models.migrant.findOne(
    { where: { id } },
    async (err, instance) => {
        if (err) cb(err, null);
        const uportIdentity = initialize(instance);
        const declarations = await uportIdentity.
           getAllAttestations();
        cb(null, declarations);
```

```
15      }
16      );
17  };
18
19  /**
20  * Adds a declaration for a specific issuer
21  * @param {string} issuer the id of the owner of the
        declarations.
22  * @param {string} subject the subject of the declaration
23  * @param {string} key
24  * @param {string} value the value of the declaration itself
25  *
26  * @returns {Object} response the added declaration.
27  */
28  Declaration.addDeclaration = (issuer, subject, key, value, cb
        ) => {
29      let issuerUportIdentity, signer, credentials, declaration
            , response;
30
31      Declaration.app.models.migrant.findOne(
32      { where: { id: issuer } },
33      async (err, instance) => {
34          if (err) cb(err, null);
35
36          try {
37          issuerUportIdentity = initialize(instance);
38
39          // Gets the SimpleSigner object with the private key
                of the user, which is needed to sign transactions
40          signer = SimpleSigner(issuerUportIdentity.deviceKeys.
                privateKey);
41
42          // Instantiates the Credentials class, a uPort class
                which simplifies the creation of signed
                attestation JWTs
43          credentials = new Credentials({
44              address: issuerUportIdentity.mnid,
45              signer: signer,
46              networks: {
47              [issuerUportIdentity.network.id]: {
48                  ...issuerUportIdentity.network
49              }
50              }
51          });
52
53          // The attest function creates a signed attestation
                JWT
54          declaration = await credentials.attest({
55              sub: subject,
```

```
56                 claim: { [key]: value }
57             });
58
59             // The consume function is a UPortClient function
                   which parses uPort uris and relays them to the
                   responsible functions
60             response = await issuerUportIdentity.consume(
61                 `me.uport:add?attestations=${declaration}`
62             );
63
64             cb(null, response);
65             } catch (error) {
66             cb(error, null);
67             }
68         }
69         );
70 };
```

2.4.2 Creazione di uPort Identity

Si segue questo flusso ogni volta che un nuovo utente si iscrive alla piattaforma:

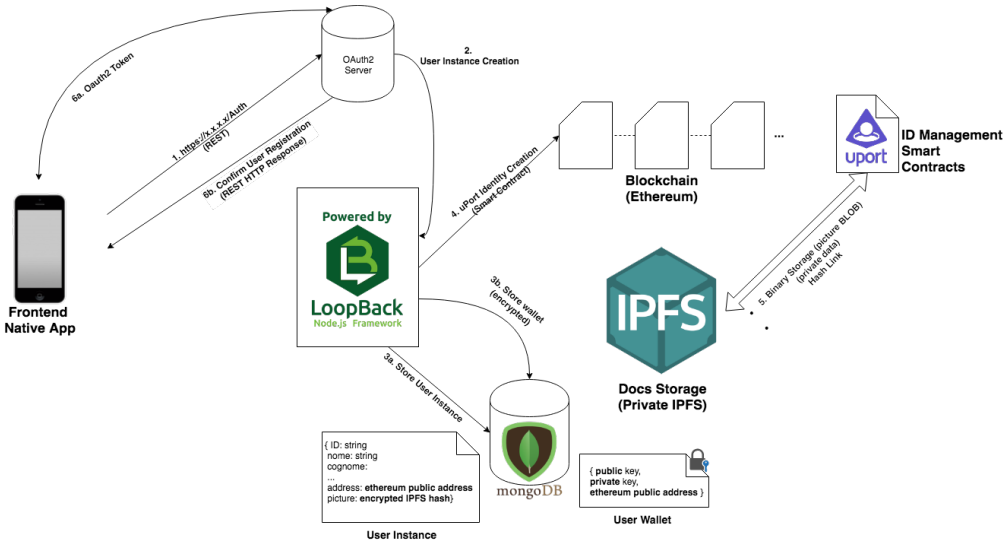


Figura 2.4: Flusso di creazione di identità.

Durante la registrazione viene istanziato sulla blockchain Ethereum privata lo smart contract Proxy associato all'utente e vengono salvate su database le credenziali dell'utente (nome, cognome...) e gli si associa un wallet che

viene poi criptato. A questo punto ad ogni login dell'utente viene deserializzata l'identità uPort ad esso associata ed il wallet, in modo da poter interagire con la chain.

Implementazione

```
/**
* creates a new migrant instance in uPort and in mongoDB.
* input params are personal info of the user.
* @param {String} name
* @param {String} familyName
* @param {string} phoneNumber
* @param {String} email
* @param {String} password
*
* @returns {Object} the migrant created instance
*/
Migrant.addMigrant = (name, familyName, phoneNumber, email,
    password, cb) => {
    Migrant.app.dataSources.BlincTestDB.autoupdate('migrant',
        async err => {
    if (err) throw err;

    let migrantID;

    const info = {
        name,
        familyName,
        phoneNumber,
        email,
        password
    };

    try {
        migrantID = await createUportId(info);
        migrantID = await initializeUportId(migrantID);
    } catch (error) {
        throw error;
    }

    migrantID = {
        id: migrantID.id,
        info: migrantID.info,
        deviceKeys: migrantID.deviceKeys,
        recoveryKeys: migrantID.recoveryKeys,
        mnid: migrantID.mnid,
        initialized: migrantID.initialized
    };

    Migrant.app.models.migrant.create([migrantID], function(
```

```
        err, migrant) {
43          if (err) {
44          cb(err, null);
45          }
46      });
47
48      cb(null, migrantID);
49      });
50  };
51
52  /**
53  * Gets a migrant from the datasource
54  * @param {String} email
55  * @param {String} password
56  *
57  * @returns {Object} the migrant instance
58  */
59  Migrant.getMigrant = (email, password, cb) => {
60      Migrant.app.models.migrant.findOne(
61      { where: { 'info.email': email, 'info.password': password
            } },
62      (err, instance) => {
63          if (err) cb(err, null);
64
65          cb(null, instance);
66      }
67      );
68  };
```

Le funzioni `createUportId` e `initializeUportId` fanno parte di una piccola libreria scritta da me che funge da middleware tra la business logic raggiungibile tramite chiamate API e la libreria `uport-js-client` (modificata ampiamente per i requisiti di BLINC), che simula il funzionamento dei protocolli uPort. Qui il codice delle due funzioni:

```
1  /**
2   * creates a UportClient instance.
3   * @param {Object} info personal info of the user.
4   *
5   * @returns {Promise<UPortClient>} the promise to have a
        UPortClient with specified infos.
6   */
7  const createUportId = (info = {}) => {
8    return new Promise(async (resolve, reject) => {
9      try {
10       const uportClient = new UPortClient(config, { info });
11       // Creates a keypair for the uPort client
12       uportClient.initKeys();
13
```

```
      const accounts = await web3.eth.getAccounts();
      const miner = accounts[0];

      const fundTx = {
        from: miner,
        to: uportClient.deviceKeys.address,
        value: 0.2 * 1.0e18
      };

      // Sends transaction to fund the uPort client in order
         to deploy uPort contracts later
      await web3.eth.sendTransaction(fundTx);

      console.log('Funded identity');

      resolve(uportClient);
    } catch (e) {
      reject(e);
    }
  });
};
```

```
/**
* Initializes a UportClient instance, meaning that it will
   deploy uPort IdentityManager contract, save the DID
   Document on IPFS and save it on Registry contract.
* @param {UPortClient} uportClient an instance of UPortClient
   .
* @param {Object} appDDO facultative object with additional
   info in UPortClient is an instance of a uPort application
*
* @returns {Promise<UPortClient>} the promise to have a
   initialized UPortClient.
*/
const initializeUportId = (uportClient, appDDO = {}) => {
    return new Promise(async (resolve, reject) => {
    if (uportClient) {
        try {
            await uportClient.initializeIdentity(appDDO);
            resolve(uportClient);
        } catch (error) {
            reject(error);
        }
    } else reject('An uPort ID must be created!');
    });
};
```

La libreria `uport-js-client` mette a disposizione una classe `UPortClient` così fatta:

```
class UPortClient {
    constructor(config = {}, initState = {}) {
        this.responseHandler = configResponseHandler(config.
            responseHandler);

        // Object that contains user's info if passed to the
            constructor, else empty object
        this.info = initState.info || {};

        // Keypair of the user necessary to sign transactions
        this.deviceKeys = config.deviceKeys;

        // Keypair of another identity, used to recover user'
            s identity (Not used in BLINC as for now
            Identities are saved on a DB)
        this.recoveryKeys = config.recoveryKeys;

        /* Object with this form:
         *  network: {
         *   id: "0x456719", this is the id of the private
             blockchain deployed for BLINC
         *   rpcUrl: "http://10.83.0.11:8545", this is the
             endpoint we connect to in order to communicate
             with the blockchain via EthJS, a JavaScript
             library
         *   claimsRegistry: "0
             xa7b3058152165c72a4dd7c4812c5964f1c26f00d", this
             is the address of the EthereumClaimsRegistry
             contract on BLINC's private chain
         *   registry: "0
             xdb571079af66edbb1a56d22809584d39c20001d9", this
             is the address of the UportRegistry contract on
             BLINC's private chain
         *   identityManager: "0
             xff37a57b8d373518abe222db1077ed9a968a5fdf", this
             is the address of the IdentityManager contract on
              BLINC's private chain
         *   storage: "0
             x7e27e8f3aa4bda26502c38ccd28a4838aeca7966", this
             is the address of the smart contract that stores
             the IPFS hashes of user's documents
         *  },
         */
        this.network = config.network
        ? configNetwork(config.network)
        : configNetwork(networkConfig.network); // have some
            default connect/setup testrpc

        if (this.network) {
```

```
29              // Crates an instance of the IPFS class, which
                   allows us to communicate with the IPFS node
                   specified in the configuration
30              this.ipfs = new IPFS(networkConfig.ipfsConfig);
31
32              this.registryNetwork = {
33                  [this.network.id]: {
34                      registry: this.network.registry,
35                      rpcUrl: this.network.rpcUrl
36                  }
37              }
38          };
39
40          const registry = new UportLite({
41              networks: this.registryNetwork
42          });
43
44          // Function that uses the UportLite library
45          this.registry = address =>
46              new Promise((resolve, reject) => {
47                  registry(address, (error, profile) => {
48                      if (error) return reject(error);
49
50                      resolve(profile);
51                  });
52          });
53
54          this.verifyJWT = jwt => verifyJWT({ registry: this.
                registry, address: this.mnid }, jwt);
55
56          // Sets the HttpProvider, necessary object for EthJS
                library in order
57          // to interact with the Ethereum blockchain
58          this.provider = config.provider || new HttpProvider(
                this.network.rpcUrl);
59
60          // Creates the EthJS object with the just created
                HttpProvider as a parameter
61          this.ethjs = this.provider ? new EthJS(this.provider)
                : null;
62
63          // Sets addresses of contracts passed in the
                configuration object
64          this.claimsRegistryAddress = this.network.
                claimsRegistry;
65          this.registryAddress = this.network.registry;
66          this.identityManagerAddress = this.network.
                identityManager;
67
```

```
68          // Necessary to do some actions, if identity is
                recreated from MongoDB config.initialized has a
                value,
69          // else identity is initialized later
70          this.initialized = config.initialized || false;
71
72          this.consume = this.consume.bind(this);
73      }
74  }
```

Questo è il codice per inizializzare una identità uPort ed è richiamato dalla funzione da me creata `initializeUportId`

```
1   initializeIdentity(initDdo) {
2       if (!this.network)
3           return Promise.reject(new Error('No network
                configured'));
4
5       const IdentityManagerAddress = this.
            identityManagerAddress;
6
7       // Creates an object contract with a specific ABI
8       // (Application Binary Interface, a low level API-like
            for contracts,
9       // written in JSON as result of contract compilation)
10      // and at a specific address for the IdentityManager
            contract
11      const IdentityManager = Contract(IdentityManagerArtifact.
            abi).at(
12          IdentityManagerAddress
13      );
14
15      // Creates keypair and an ethereum address for the user
            and for recovery
16      this.initKeys();
17
18      // Calls contract function to create an Identity
19      const uri = IdentityManager.createIdentity(
20          this.deviceKeys.address,
21          this.recoveryKeys.address
22      );
23
24      // The consume function is a uport-js-client function
            which, given a URI which conforms to the uPort
            Protocol specs,
25      // parses it and, basing on the given URI format, sends a
            tx, adds an attestation or requests credentials.
26      return this.consume(uri)
27      .then(this.getReceipt.bind(this))
28      .then(receipt => {
```

```
29          const log = receipt.logs[0];
30
31          const createEventAbi = IdentityManager.abi.filter(
32              obj => obj.type === 'event' && obj.name === '
                    IdentityCreated'
33          )[0];
34
35          // Gets the Proxy contract address for the identity
                from
36          // the event emitted by the createIdentity function
37          this.id = decodeEvent(createEventAbi, log.data, log.
                topics).identity;
38
39          // Creates the MNID for the identity, which is the
                base58 encoding of the network id and the identity
                 Proxy address
40          this.mnid = mnid.encode({ network: this.network.id,
                address: this.id });
41          this.initTransactionSigner(IdentityManagerAddress);
42
43          const baseDdo = {
44              '@context': 'http://schema.org',
45              '@type': 'Person',
46              publicKey: this.deviceKeys.publicKey
47          };
48
49          const ddo = Object.assign(baseDdo, initDdo);
50
51          // The code for the writeDDO is below, at a high
                level it uploads the ddo object to IPFS
52          // and adds it to the Registry smart contract
53          return this.writeDDO(ddo);
54      })
55      .then(this.ethjs.getTransactionReceipt.bind(this.ethjs))
56      .then(receipt => {
57
58          this.initialized = true;
59          return;
60      });
61  }
62
63  writeDDO(newDdo) {
64      // Creates an object contract with a specific ABI
65      // and at a specific address for the Registry contract
66      const Registry = Contract(RegistryArtifact.abi).at(this.
            network.registry);
67      return this.getDDO()
68        .then(ddo => {
69          // If the Identity Document already exists it doesn't
```

```
            overwrite with the given one,
70          // otherwise it adds to IPFS the newDdo
71          ddo = Object.assign(ddo || {}, newDdo);
72          return new Promise((resolve, reject) => {
73            this.ipfs.add(Buffer.from(JSON.stringify(ddo)), (
                err, result) => {
74              if (err) reject(new Error(err));
75              resolve(result);
76            });
77          });
78        })
79        .then(res => {
80          const hash = res[0].hash;
81          const hexhash = new Buffer(base58.decode(hash)).
              toString('hex');
82          // Removes Qm from ipfs hash, which specifies length
              and hash
83          const hashArg = `0x${hexhash.slice(4)}`;
84          const key = 'uPortProfileIPFS1220';
85
86          // Writes on the Registry contract the association
              between the Proxy contract address
87          // for the identity and the IPFS hash of its Identity
               Document which contains its pubkey
88          return Registry.set(key, this.id, hashArg);
89        })
90        .then(this.consume.bind(this));
91  }
```

Capitolo 3

Conclusioni

3.1 Problemi aperti

3.1.1 Su Ethereum e blockchain in generale

Scalabilità Eseguire calcoli su una blockchain è troppo lento e costoso, quindi salvare e fare calcoli su grandi quantità di dati non è fattibile. Per fare un confronto, il circuito VISA processa circa 10.000 transazioni al secondo, Ethereum ne processa al massimo 15 al secondo, con grandi problemi quando la rete è molto utilizzata (come nel dicembre 2017 con il fenomeno CryptoKitties) che causano ritardo nel processamento di transazioni e aumento del costo del gas.

Privacy I dati su una blockchain sono accessibili da tutti, rendendo impossibile il salvataggio e la computazione di dati sensibili. Ciò restringe il numero di applicazioni possibili per la blockchain.

3.1.2 Su uPort

SDK per mobile

Al momento della scrittura di questa sezione non è ancora disponibile un SDK maturo per l'utilizzo dei protocolli di uPort su applicazioni native Android e iOS.

Parziale centralizzazione

Al momento uPort utilizza ancora dei microservizi per rendere possibili alcune parti fondamentali dell'architettura come ad esempio il server di messaging Chasqui o il server che finanzia le transazioni Sensui, non è sono ancora decentralizzati in mancanza di alternative solide ai modelli centralizzati.

3.1.3 Su BLINC

Centralizzazione

A causa delle scadenze non rispettate dal team di uPort per quanto riguarda le SDK mobile e delle scadenze che il team di BLINC doveva rispettare è stato necessario spostare il wallet degli utenti su un server centrale invece che mantenerlo sul device dei migranti. Questo va parzialmente contro i principi di decentralizzazione e sovranità del dato che caratterizzano la blockchain ed il progetto uPort.

3.2 Possibili scenari futuri

3.2.1 Su Ethereum e blockchain in generale

Scalabilità

Sono in diverse fasi di sviluppo (alcune in fasi embrionali, altre già oltre il MVP) alcune possibili soluzioni per risolvere i problemi di scalabilità di Ethereum.

- Raiden
- Plasma
- Casper
- Sharding

Privacy

Diverse startup ed aziende stanno lavorando per rendere possibile la computazione e il salvataggio di dati privati, tra cui Keep ed Enigma: la prima avvalendosi di contenitori off-chain di dati privati e la seconda tramite l'uso di smart contract privati.

3.2.2 Su uPort

SDK mobile Lo sviluppo delle SDK procede come si vede sui repository GitHub di uPort, quindi si raggiungerà il livello di maturazione necessario a decentralizzare l'architettura di BLINC in relativamente poco tempo.

Rivoluzione dell'architettura di uPort L'architettura uPort cambierà radicalmente di qui a poco, passando da un'astrazione dell'identità basata su smart contract sviluppati internamente in uPort ad una architettura basata sugli standard proposti dalla Decentralized Identity Foundation.

A differenza dell'attuale architettura dove la creazione di un'identità richiede due transazioni (deploy del contratto Proxy tramite chiamata allo smart contract IdentityManager e registrazione dell'Identity Document su smart contract Registry), la registrazione di un'identità nella nuova architettura richiede soltanto la creazione di un account Ethereum, ed è quindi gratuita.

3.2.3 Su BLINC

Spostamento del wallet su telefono

Una volta che saranno rilasciate le SDK mobile di uPort si procederà a decentralizzare l'architettura, spostando i wallet degli utenti da MongoDB al loro smartphone e di fatto rimuovendo la necessità di avere un backend centralizzato.

Salvataggio degli attributi delle uPort Identity su Identity Hub

Invece di salvare gli attributi delle identità su MongoDB, l'attuale soluzione provvisoria e centralizzata, si passerà all'utilizzo di Identity Hub che sono, come descritto sul repository GitHub della Decentralized Identity Foundation, dei datastore che contengono oggetti significativi per l'identità in locazioni conosciute. Ogni oggetto in un Hub è firmato dall'identità proprietaria ed è accessibile globalmente attraverso delle API conosciute globalmente. Il vantaggio di un Hub rispetto ad un datastore tradizionale come può essere appunto un database Mongo è la decentralizzazione: un'identità può avere una o più istanze di Hub che sono indirizzabili tramite un meccanismo di routing basato su URI collegati all'identificatore dell'identità. Tutte le istanze di Hub si sincronizzano tra di loro, garantendo così la consistenza dei dati al loro interno e permettendo al proprietario dei dati di accedervi da ovunque, anche offline. Molto probabilmente si sfrutteranno i 3box, ovvero l'implementazione del team di uPort della specifica degli Identity Hub.