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


**1 Introduzione** **3**
- 1.1 Ambito del progetto . . . 3
- 1.2 Descrizione dell'azienda . . . 4
- 1.3 Architettura generale del progetto . . . 5
  - 1.3.1 Soluzione tecnologica . . . 6
- 1.4 Obbiettivo della tesi . . . 7

**2 Blockchain** **9**
- 2.1 Struttura di una blockchain . . . 9
  - 2.1.1 Transazioni . . . 11
  - 2.1.2 Algoritmi di consenso . . . 11
- 2.2 La blockchain Ethereum . . . 13
  - 2.2.1 Smart contract . . . 13
  - 2.2.2 Solidity: un linguaggio per lo sviluppo di smart contract 13
  - 2.2.3 Perché la blockchain Ethereum per BLINC? . . . 16

**3 Identità digitale** **19**
- 3.1 Analisi dei requisiti . . . 19
- 3.2 Elenco dei prodotti e delle tecnologie disponibili . . . 20
- 3.3 uPort . . . 22
  - 3.3.1 uPort e dati on-chain: smart contract per l'identità . . 24
  - 3.3.2 uPort off-chain: JWT per lo scambio di informazioni . 35
  - 3.3.3 JSON Web Token . . . 35
  - 3.3.4 uPort e la sovranità del dato . . . 37
  - 3.3.5 Perché è stato scelto uPort? . . . 41
  - 3.3.6 Stato del progetto . . . 41
- 3.4 Integrazione nel progetto BLINC . . . 42
  - 3.4.1 Gestione di dichiarazioni ed attestazioni . . . 42
  - 3.4.2 Creazione di uPort Identity . . . 47

**4 Conclusioni** **57**
4.1 Risultati ottenuti . . . . . 57
4.2 Problemi aperti . . . . . 57
4.2.1 Su Ethereum e blockchain in generale . . . . . 57
4.2.2 Su uPort . . . . . 58
4.2.3 Su BLINC . . . . . 58
4.3 Possibili scenari futuri . . . . . 62
4.3.1 Su Ethereum e blockchain in generale . . . . . 62
4.3.2 Su uPort . . . . . 62
4.3.3 Su BLINC . . . . . 63

# 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.

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[1] 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

[1] Consoft Sistemi: https://www.consoft.it

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 Architettura generale del progetto

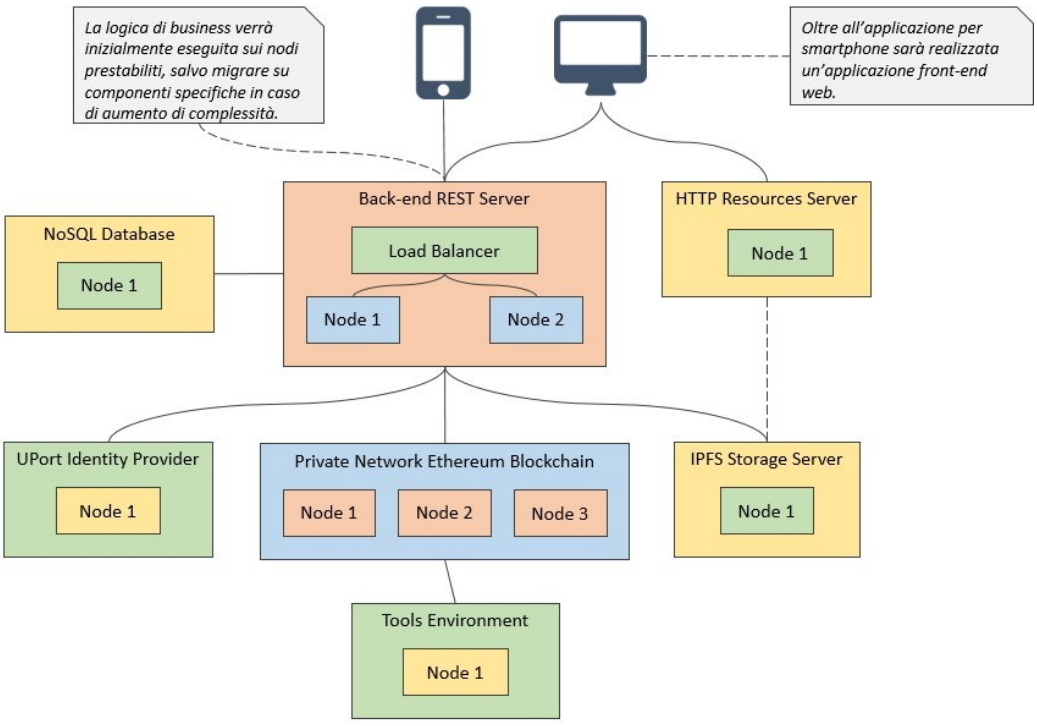


Figura 1.1: Diagramma dell'architettura di BLINC.

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[2]): necessario per memorizzare informazioni e wallet criptati degli utenti
- Server di back-end (Loopback[3], 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 macchine che eseguono istanze del client Ethereum Geth.
- IPFS[4] 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.

### 1.3.1 Soluzione tecnologica

Data l'immaturità del SDK di uPort che non ha permesso la creazione 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 MongoDB (che contiene i wallet degli utenti criptati). In questa prima versione il flusso d'interazione base è quindi questo:

[2]MongoDB: https://www.mongodb.com/
[3]LoopBack: https://loopback.io/
[4]InterPlanetary File System: https://ipfs.io/

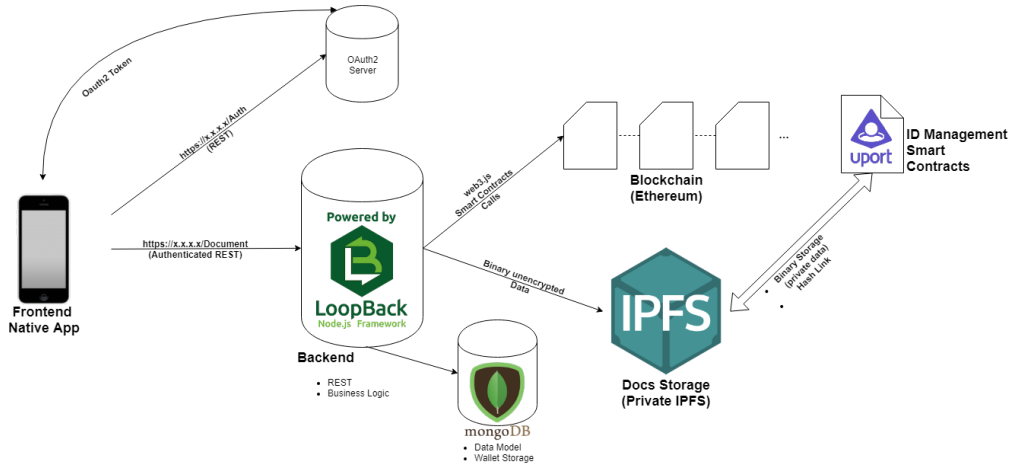


Figura 1.2: Soluzione tecnologica per BLINC.

### Utente registrato

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.

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.

## 1.4 Obbiettivo della tesi

L'obbiettivo della tesi è quello di spiegare in dettaglio il mio lavoro di ricerca ed applicazione di soluzioni blockchain e di identità digitale decentralizzata nell'ambito del progetto BLINC.

Gran parte del mio tirocinio è stata dedicata alla ricerca e poi alla sperimentazione delle tecnologie abilitanti: in primissimo luogo ho studiato cos'è una blockchain [1] per poi iniziare l'approfondimento su Ethereum, la blockchain definita come *world computer* [2]. Da qui le prime sperimentazioni: installazione del client Ethereum Geth[5] con invio di transazioni e deploy di smart contract prima da linea di comando e poi tramite la libreria JavaScript web3.js[6]. Una volta preso confidenza con i concetti e gli strumenti base per poter comprendere e lavorare su Ethereum ho iniziato a focalizzarmi sulla parte di mia competenza per il progetto: l'identità digitale. Dopo una ricerca iniziale sull'identità a sovranità personale ho approfondito la soluzione specifica più adatta a BLINC selezionata dopo una scelta condivisa tra tutte le parti interessate nel progetto, ovvero uPort[7]. In seguito ad uno studio dell'architettura e delle librerie che uPort a disposizione, ne è stata scelta una che venisse incontro alle nostre esigenze, almeno per una prima versione di BLINC. Da qui inizia il lavoro pratico sul progetto in collaborazione con i miei colleghi tirocinanti, ovvero lo sviluppo di una prima versione del sistema secondo l'architettura descritta precedentemente: in particolare mi sono occupato della creazione di endpoint API in NodeJS per la creazione e gestione di identità ed attestazioni su di esse sfruttando la libreria `uport-js-client`.

[5]Go Ethereum: https://geth.ethereum.org/
[6]web3.js: https://github.com/ethereum/web3.js
[7]uPort: https://uport.me

# Capitolo 2

# Blockchain

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 ha avuto sui modi di comunicare e acquisire informazioni. La posta in gioco riguarda industrie che generano oltre il 20% del PIL degli Stati Uniti [3]. Chiunque sarà in grado di creare coupon, titoli al portatore sistemi di pagamento, *smart contract* 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 diffusione di dispositivi robotici sarà un altro importantissimo driver: operatori come IBM e Samsung prevedono che gli oggetti negozieranno tra di loro titoli per l'accesso a dati, energia e rapporti di cooperazione nelle loro funzioni. [4] 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.

## 2.1 Struttura di una blockchain

Una **blockchain** è una architettura di calcolo distribuito funzionante come un registro elettronico senza intermediari che garantisce l'immutabilità e la permanenza delle informazioni salvate su di essa.

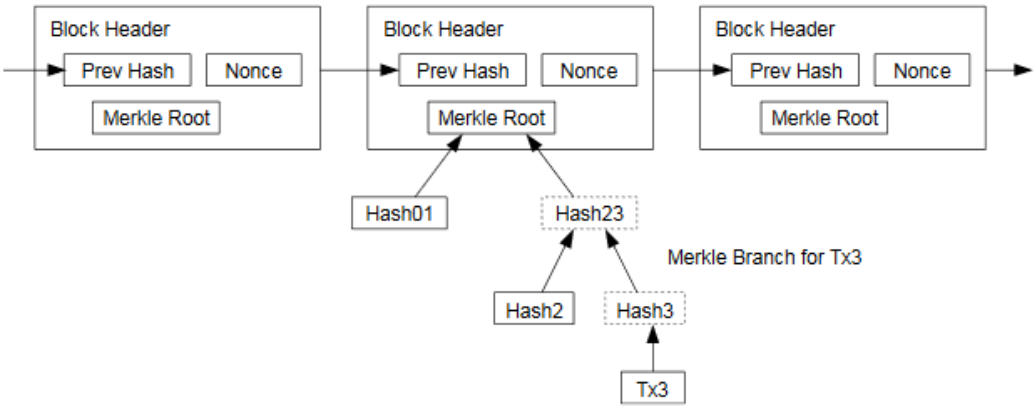


Figura 2.1: Struttura di una blockchain.

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 viene utilizzato l'algoritmo di hashing 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, permette di ottenere un hash del blocco che è valido se è numericamente inferiore al numero che rappresenta la difficoltà della rete.

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 e la difficoltà della rete ciò sarebbe computazionalmente quasi impossibile.

La blockchain è nata con Bitcoin, sistema peer-to-peer per una valuta digitale ideato da Satoshi Nakamoto nel 2008 basato su una blockchain distribuita e decentralizzata. [5] 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 gestisce un wallet (una chiave privata, una chiave pubblica ed un indirizzo Bitcoin) e connette il dispositivo su cui è installato come nodo della rete.

### 2.1.1 Transazioni

Lo scambio di valore (e opzionalmente dati) tra due account in una blockchain avviene tramite una **transazione**. Una transazione è un messaggio inviato dall'indirizzo mittente alla blockchain contenente un indirizzo destinatario, il valore della transazione (anche 0), un nonce (solo per Ethereum, un numero unico per account incrementale per evitare il doppio invio di transazioni), un campo dati (anche vuoto) e le tre componenti di una firma ECDSA $r$, $s$ e $v$.

Una volta inviata, la transazione entra in un *transaction pool*, da dove i miner andranno a selezionare randomicamente transazioni da includere nel prossimo blocco. Una transazione che viene correttamente inclusa in un blocco è definitiva e irreversibile.

### 2.1.2 Algoritmi di consenso

Affinché tutti i nodi partecipanti ad una rete come Bitcoin raggiungano un accordo su quale sarà il prossimo blocco da aggiungere alla blockchain e quindi ottenere un unico stato del registro elettronico condiviso è necessario implementare un **algoritmo di consenso**.

#### Proof of Work

Nel caso di Bitcoin ed Ethereum l'algoritmo utilizzato è **Proof of Work**: per raggiungere consenso sullo stato della blockchain Proof of Work (in breve PoW) si avvale di particolari nodi chiamati *miner*. I miner prendono delle transazioni dal *transaction pool* e le inseriscono in un blocco: affinché questo venga accettato dal resto della rete e aggiunto alla blockchain un miner deve trovare un *nonce*, ovvero un numero che, se inserito in una funzione di hash assieme al contenuto del blocco, permette di ottenere un hash numericamente minore della difficoltà della rete. Questo calcolo, essendo computazionalmente molto costoso, permette di evitare lo spam di blocchi sulla rete, ma essendo anche economicamente parecchio oneroso a causa della grande quantità di elettricità usata nel calcolo dell'hash corretto, è necessario garantire un incentivo economico ai miner che aggiungono con successo un blocco alla blockchain: da qui nascono le criptovalute. Quando un blocco viene aggiunto alla blockchain il miner che l'ha sottoposto allo scrutinio del

resto dei nodi che l'hanno validato e approvato viene ricompensato con della criptovaluta (Ad oggi un miner su Bitcoin viene ricompensato con 12,5 BTC a blocco e 5 Ether a blocco su Ethereum). Il Proof of Work ha però dei problemi, principalmente due: l'enorme impatto ambientale a causa della grande quantità di energia elettrica utilizzata dai miner e la centralizzazione della potenza di hashing causata dalla creazione di potenti gilde di miner che stanno monopolizzando sia la rete Bitcoin che quella Ethereum.

### Proof of Stake

A causa degli svantaggi sopra elencati di Proof of Work si stanno ricercando algoritmi di consenso alternativi che vadano ad eliminare (o almeno a ridurre drasticamente) i problemi che lo affliggono.

L'algoritmo che molto probabilmente andrà a sostituire PoW è **Proof of Stake** (in breve PoS). In Proof of Stake i miner vengono sostuituiti dai *validatori*, i quali hanno lo stesso compito dei primi, ovvero quello di validare transazioni ed inserirle in un blocco aggiunto alla blockchain. La differenza sta nel modo in cui vengono selezionati i validatori: se prima più potenza di calcolo significava pià probabilità di aggiungere un blocco, ora è la quantità di criptovaluta congelata (lo stake) in uno smart contract a definire la probabilità di validare un blocco ed aggiungerlo alla blockchain. Con PoS diventa ancora più difficile inserire transazioni fraudolente nella blockchain: con PoW il nodo maligno prova ad inserire transazioni fasulle al massimo perde tempo, con PoS perde direttamente la criptovaluta congelata sullo smart contract. PoS va a ridurre drasticamente sia l'impatto ambientale, dato che non serve particolare potenza di calcolo per validare i blocchi, sia il rischio di centralizzazione, cosa che può sembrare controintutiva dato che più si mette criptovaluta nello stake e più si ha probabilità di essere selezionati per validare un blocco. Questo vantaggio è dato dalla minore rilevanza delle **economie di scala**: \$10 milioni di criptovaluta daranno esattamente ritorni 10 volte migliori di \$1 milione di criptovaluta, senza altri guadagni aggiuntivi che invece si hanno quando si va a comprare in massa hardware per milioni di dollari di qualità maggiore rispetto alla media. [6]

Ethereum sta implementando Proof of Stake tramite il suo progetto Casper.[1]

[1]Casper: https://github.com/ethereum/casper

## 2.2 La blockchain Ethereum

Ethereum[2] prende tutte le caratteristiche di Bitcoin e le amplia, a partire dalla criptovaluta nativa del sistema, l'*Ether*.

L'Ether ha una serie di multipli e sottomultipli, tra cui il più utilizzato in ambito *smart contract* **wei**, che vale $10^{-18}$ Ether.

L'Ether, oltre a fungere da scambio di valore tra un utente della rete ed un altro, serve anche a pagare il *gas*, ovvero la tassa introdotta per evitare lo spam di transazioni sulla rete e necessaria per inviare transazioni che possono anche non avere valore, ma possono ad esempio contenere del codice.

### 2.2.1 Smart contract

Ethereum infatti rende la sua blockchain facilmente programmabile tramite **smart contract**, ovvero programmi scritti in un linguaggio di programmazione ad alto livello come Solidity (simil-JavaScript) o Vyper (simil-Python), compilati in bytecode e distribuiti sulla blockchain tramite speciali transazioni inviate ad un generico indirizzo `0x0`. 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, estrema sicurezza ed immutabilità.

### 2.2.2 Solidity: un linguaggio per lo sviluppo di smart contract

Solidity[3] è un linguaggio procedurale ad alto livello ispirato a JavaScript per lo sviluppo di smart contract. È di gran lunga il linguaggio più diffuso e supportato per lo sviluppo di smart contract, ed è anche quello più maturo.

La sintassi è familiare, ma ha alcune peculiarità derivanti dal fatto che ha a che fare con transazioni monetarie e non.

[2]Ethereum: https://ethereum.org/
[3]Solidity: https://github.com/ethereum/solidity

### Oggetti globali

Solidity ha definiti come oggetti globali `msg`, `block` e `tx`, che contengono rispettivamente informazioni riguardo alla chiamata che ha iniziato l'esecuzione dello smart contract (come valore della transazione e indirizzo chiamante), al blocco corrente (come numero e timestamp del blocco) e alla transazione (come origine della transazione e costo in gas).

### Tipi di dato

I tipi sono quelli standard, con la particolarità che il tipo `string` è stato aggiunto solo molto dopo la creazione del linguaggio e ad ora non ci sono metodi nativi del linguaggio per lavorare sulle stringhe. Un tipo di dato che è fondamentale in Solidity è `address`, che ha una serie di proprietà e metodi che permettono di lavorare su indirizzi Ethereum tra cui:

- `address.balance`: restituisce il bilancio in wei dell'indirizzo
- `address.transfer(wei da inviare)`: invia il dato numero di wei all'indirizzo Ethereum
- `address.call(payload)`: crea una chiamata di basso livello ad una funzione

### Ereditarietà

Solidity supporta l'ereditarietà tra contratti, permettendo di conseguenza una maggiore modularità del codice andando però a rendere più difficile l'analisi di contract, motivo per cui sta prendendo molto piede Vyper[4], il secondo linguaggio più diffuso che ha molti più vincoli sintattici e non supporta ereditarietà ma è indicato per contesti dove la sicurezza e la facilità di revisione del codice sono particolarmente importanti.

### Modificatori di funzione

Un'altra particolarità di Solidity sono i **modificatori di funzione**, come questo:

```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

[4]Vyper: https://github.com/ethereum/vyper

In particolare questo modificatore permette l'esecuzione della funzione a cui è applicato soltanto se il chiamante della funzione è il proprietario, ovvero chi l'ha distribuito sulla blockchain o un delegato, dello smart contract. I modificatori sono infatti molto utili per regolare gli accessi a funzioni, come in questo caso:

```
function adminFunction() public onlyOwner {
    // Code accessible only by admin
}
```

### Eventi

Gli eventi sono costrutti Solidity per creare log di transazioni andate a buon fine e non. Gli eventi sono particolarmente utili per dare un feedback visivo ad un utente di una dApp sulle'esito di una transazione, in quanto possono essere impostati dei listener su di essi ed è possibile fare query anche sugli eventi passati.

### Bytecode ed ABI

Il compilatore `solc` permette di ottenere l'ABI di uno smart contract ed il bytecode eseguibile dalla EVM a partire da un codice sorgente in Solidity. Una **ABI** (Application Binary Interface) è come un'API a basso livello, funge quindi da interface del contract, specificando in un formato JSON le funzioni richiamabili con i parametri accettati e le variabili di ritorno.

### Esempio di contract con la rispettiva ABI

```
pragma solidity ^0.4.22;

contract SimpleKeyValue {
  mapping (int => string) keyValueMapping;

  function set (int _key, string _value) public {
    keyValueMapping[_key] = _value;
  }

  function get (int _key) public view returns (string _value)
      {
    return keyValueMapping[_key];
  }
}
```

```
[
    {
```

```
        "constant": false,
        "inputs": [
        {
            "name": "_key",
            "type": "int256"
        },
        {
            "name": "_value",
            "type": "string"
        }
        ],
        "name": "set",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [
        {
            "name": "_key",
            "type": "int256"
        }
        ],
        "name": "get",
        "outputs": [
        {
            "name": "_value",
            "type": "string"
        }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    }
]
```

### 2.2.3 Perché la blockchain Ethereum per BLINC?

Oltre a motivi di sicurezza e trasparenza di caricamento e validazione di informazioni (documenti, certificati, contratti), la blockchain è stata una scelta tecnologica ovvia per BLINC, in quanto permette di creare una rete di fiducia distribuita e condivisa tra migranti, ACLI e aziende, nella quale ognuno dei soggetti interessati può partecipare equamente ed avere accesso ad un'informazione trasparente.

Tra tutte le possibili alternative è stato scelto Ethereum perché, nonostante la giovinezza del progetto, 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 altri progetti come EOS 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[5], che facilita e automatizza task di compilazione e migrazione di contratti su diversi tipi di blockchain, e Ganache [6], una simulazione in JavaScript di una blockchain locale che, in quanto non ha necessità 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.

[5]Truffle: https://truffleframework.com/

[6]Ganache: https://truffleframework.com/ganache

# Capitolo 3

# Identità digitale

## 3.1 Analisi dei requisiti

Come da requisiti, l'obbiettivo principale di BLINC è quello di creare un portadocumenti digitale dove i documenti inseriti rimangono sotto il controllo del proprietario.

Questo requisito introduce il concetto di **sovranità del dato** e di **identità a sovranità personale** (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, purtroppo ancora molto diffuso.

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.

L'identità a sovranità personale 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.

## 3.2 Elenco dei prodotti e delle tecnologie disponibili

- **Civic**[1]: 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 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, i quali verranno in parte anche allocati agli utenti soggetti delle attestazioni in modo da incentivare la partecipazione al sistema.

[1] Civic: https://www.civic.com/

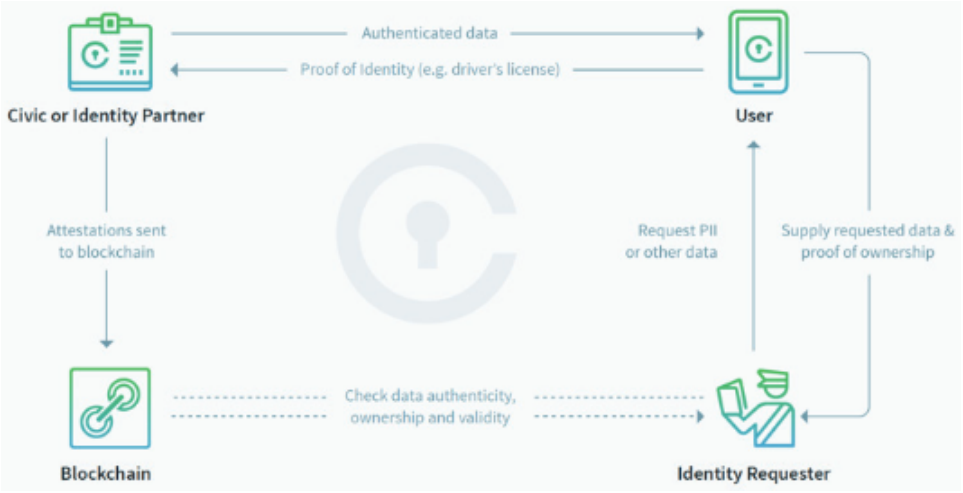


Figura 3.1: Diagramma di funzionamento di Civic.

- **SelfKey**[2]: 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[3] 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.

[2]SelfKey: https://selfkey.org/
[3]JSON-LD: https://json-ld.org/

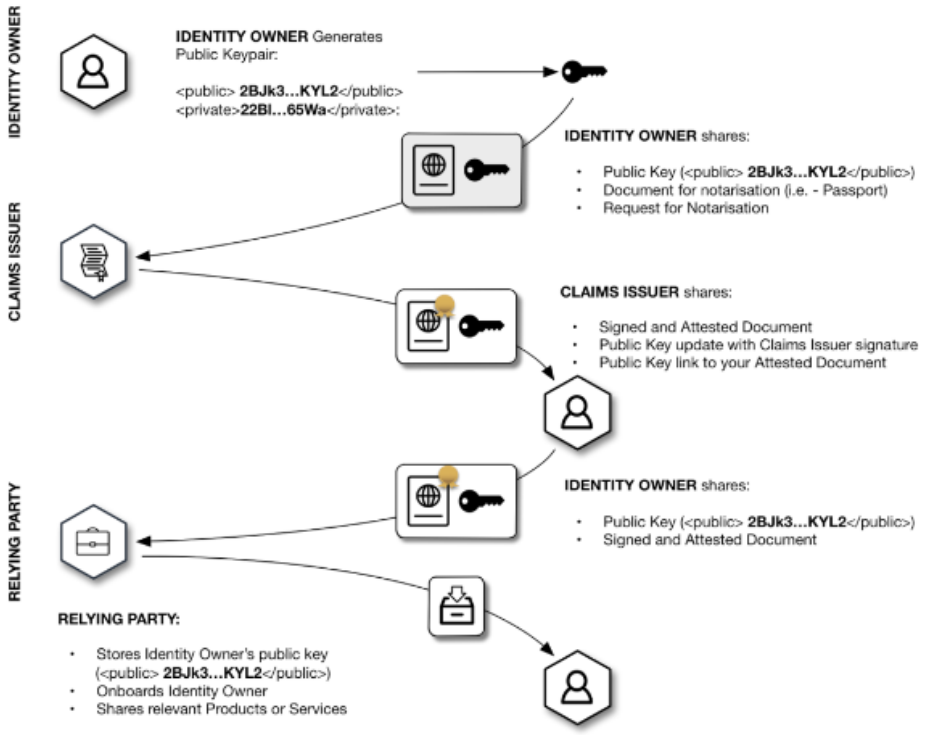


Figura 3.2: Diagramma di funzionamento di SelfKey.

- **uPort**[4]: sviluppato sotto l'egida di ConsenSys, uPort è il sistema per la self-sovereign identity basato su Ethereum scelto per la gestione delle identità su BLINC.

## 3.3 uPort

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.

[4]uPort: http://uport.me

- 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.

Un **DID Document** è un insieme di dati, come le chiavi pubbliche, rappresentati tipicamente come oggetto JSON tramite i quali un'identità decentralizzata può autenticarsi. Possono contenere anche attributi e dichiarazioni sull'identità decentralizzata. [7]

### Esempio di DID Document

```
{
  "@context": "https://w3id.org/did/v1",
  "id": "did:uport:2nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX",
  "publicKey": [{
    "id": "did:uport:2nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX#keys
       -1",
    "type": "Secp256k1VerificationKey2018",
    "owner": "did:uport:2nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX",
    "publicKeyHex": "04613
       bb3a4874d27032618f020614c21cbe4c4e47..."
  }, {
    "id": "did:uport:2nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX#keys
       -2",
    "type": "Curve25519EncryptionPublicKey",
    "owner": "did:uport:2nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX",
    "publicKeyBase64": "QCFPBLm5pwmuTOu+haxv0+Vpmr6Rrz/
       DEEvbcjktQnQ="
  }],
  "authentication": [{
    "type": "Secp256k1SignatureAuthentication2018",
    "publicKey": "did:uport:2
       nQtiQG6Cgm1GYTBaaKAgr76uY7iSexUkqX#keys-1"
  }]
}
```

Dato che tutto ciò che viene salvato sulla blockchain è pubblicamente visibile, sarebbe un grave errore quello di riportare informazioni private dell'identità direttamente sulla blockchain anche se criptate in quanto, essendo le informazioni sulla blockchain come scolpite nella pietra, ciò creerebbe problemi in un sicuro futuro nel quale le tecniche di crittazione odierne saranno facilmente superabili anche da semplici calcolatori.

uPort risolve ciò salvando su blockchain soltanto il **DID** (Decentralized IDentifier) dell'identità digitale, ovvero un identificatore non direttamente

riconducibile all'utente. Tutte le informazioni private vengono invece salvate su una parte dedicata dell'archiviazione del device dell'utente o su server personali dedicati chiamati *Identity Hub*[5]. Le informazioni private vengono poi condivise solo sotto consenso dell'identità proprietaria sotto forma di JWT tramite procedure che approfondisco nella sottosezione riguardante la gestione di dati off-chain. [8]

### 3.3.1 uPort e dati on-chain: smart contract per l'identità

Per quanto riguarda i dati salvati su Ethereum uPort prevede un insieme di smart contract per la gestione delle identità che sono:

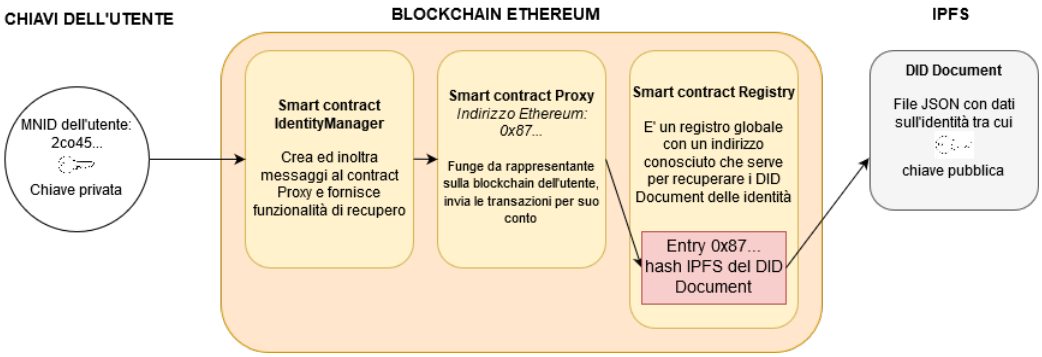


Figura 3.3: Diagramma di interazione tra gli smart contract di uPort.

- `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. E' un contract molto semplice: la funzione anonima è una funzione di fallback sul contratto a cui è possibile trasferire Ether (contrassegnata con il modificatore `payable`) e che registra la chiamata come evento. La funzione `forward` inoltra semplicemente all'indirizzo destinazione una transazione con il dato `value` ed eventuali dati passati nel parametro `data`, dopodiché registra l'avvenuto inoltro come evento.

```
// Proxy.sol
pragma solidity 0.4.15;
import "./libs/Owned.sol";
```

[5]Identity Hub: github.com/decentralized-identity/identity-hub

```
contract Proxy is Owned {
    event Forwarded (address indexed destination, uint
       value, bytes data);
    event Received (address indexed sender, uint value);

    function () payable { Received(msg.sender, msg.value)
       ; }

    function forward(address destination, uint value,
       bytes data) public onlyOwner {
        require(executeCall(destination, value, data));
        Forwarded(destination, value, data);
    }

    // copied from GnosisSafe
    // https://github.com/gnosis/gnosis-safe-contracts/
       blob/master/contracts/GnosisSafe.sol
    function executeCall(address to, uint256 value, bytes
        data) internal returns (bool success) {
        assembly {
            success := call(gas, to, value, add(data, 0
               x20), mload(data), 0, 0)
        }
    }
}
```

- `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.

  Il costruttore inizializza tre campi del contract che hanno queste funzioni:

  - `userTimeLock`: quando un `owner` è stato aggiunto si deve aspettare questo tempo prima di poter inviare transazioni
  - `adminTimeLock`: quando un `owner` è stato aggiunto si deve aspettare questo tempo prima di poter avere capacità amministrative (aggiungere o togliere proprietari o cambiare chiavi di recupero)
  - `adminRate`: quando un `owner` esegue una funzione amministrativa si deve aspettare questo tempo prima di poter eseguire di nuovo funzioni amministrative

La funzione cruciale di questo contract è `createIdentity`, la quale crea una nuova istanza di un contract Proxy che funge da rappresentante on-chain dell'identità a cui vengono assegnati un proprietario `owner` ed un account di recupero `recoveryKeys`.

La funzione successiva `createIdentityWithCall` fa la stessa cosa ma in più fa una chiamata ad un contract il cui indirizzo è specificato nel parametro `destination` e la funzione richiamata è specificata nel parametro `data`. Un caso in cui questa ulteriore chiamata ad un contract è utile e quello in cui si voglia subito richiamare il contract Registry per salvare il did Document dell'identità.

La funzione `registerIdentity` serve per trasferire il controllo di un contract Proxy all'IdentityManager, deve essere richiamata direttamente dal Proxy.

La funzione `forwardTo` permette all' `owner` di un contract Proxy di inoltrare una transazione tramite il proprio Proxy.

La funzione `addOwner` permette al proprietario di un Proxy di aggiungere un comproprietario, il quale può iniziare subito a fare transazioni senza aspettare `userTimeLock`.

La funzione `addOwnerFromRecovery` permette all'indirizzo

`recoveryKey` di aggiungere un nuovo proprietario in modo da riottenere il controllo della propria identità. In questo caso il proprietario aggiunto deve aspettare `userTimeLock` prima di poter fare transazioni.

La funzione `removeOwner` permette ad un proprietario con funzioni di amministratore di rimuovere un altro proprietario.

La funzione `changeRecovery` permette ad un proprietario di sostituire la `recoveryKey` con un'altra.

`initiateMigration`, `finalizeMigration`, `cancelMigration` permettono ad un proprietario di un Proxy di migrare il proprio contract da un IdentityManager ad un altro. La migrazione è finalizzata dallo stesso proprietario dopo un tempo di `adminTimeLock` tramite la `finalizeMigration`, mentre la `cancelMigration` può essere richiamata in qualsiasi momento.

```
// IdentityManager.sol
pragma solidity 0.4.15;
import "./Proxy.sol";


contract IdentityManager {
```

```
    uint adminTimeLock;
    uint userTimeLock;
    uint adminRate;

    event LogIdentityCreated(
        address indexed identity,
        address indexed creator,
        address owner,
        address indexed recoveryKey);

    event LogOwnerAdded(
        address indexed identity,
        address indexed owner,
        address instigator);

    event LogOwnerRemoved(
        address indexed identity,
        address indexed owner,
        address instigator);

    event LogRecoveryChanged(
        address indexed identity,
        address indexed recoveryKey,
        address instigator);

    event LogMigrationInitiated(
        address indexed identity,
        address indexed newIdManager,
        address instigator);

    event LogMigrationCanceled(
        address indexed identity,
        address indexed newIdManager,
        address instigator);

    event LogMigrationFinalized(
        address indexed identity,
        address indexed newIdManager,
        address instigator);

    mapping(address => mapping(address => uint)) owners;
    mapping(address => address) recoveryKeys;
    mapping(address => mapping(address => uint)) limiter;
    mapping(address => uint) public migrationInitiated;
    mapping(address => address) public
       migrationNewAddress;

    modifier onlyOwner(address identity) {
        require(isOwner(identity, msg.sender));
```

```
        _;
    }

    modifier onlyOlderOwner(address identity) {
        require(isOlderOwner(identity, msg.sender));
        _;
    }

    modifier onlyRecovery(address identity) {
        require(recoveryKeys[identity] == msg.sender);
        _;
    }

    modifier rateLimited(address identity) {
        require(limiter[identity][msg.sender] < (now -
            adminRate));
        limiter[identity][msg.sender] = now;
        _;
    }

    modifier validAddress(address addr) { //protects
        against some weird attacks
        require(addr != address(0));
        _;
    }

    /// @dev Contract constructor sets initial timelock
        limits
    /// @param _userTimeLock Time before new owner added
        by recovery can control proxy
    /// @param _adminTimeLock Time before new owner can
        add/remove owners
    /// @param _adminRate Time period used for rate
        limiting a given key for admin functionality
    function IdentityManager(uint _userTimeLock, uint
        _adminTimeLock, uint _adminRate) {
        require(_adminTimeLock >= _userTimeLock);
        adminTimeLock = _adminTimeLock;
        userTimeLock = _userTimeLock;
        adminRate = _adminRate;
    }

    /// @dev Creates a new proxy contract for an owner
        and recovery
    /// @param owner Key who can use this contract to
        control proxy. Given full power
    /// @param recoveryKey Key of recovery network or
        address from seed to recovery proxy
    /// Gas cost of 289,311
```

```
    function createIdentity(address owner, address
        recoveryKey) public validAddress(recoveryKey) {
        Proxy identity = new Proxy();
        owners[identity][owner] = now - adminTimeLock; //
            This is to ensure original owner has full
            power from day one
        recoveryKeys[identity] = recoveryKey;
        LogIdentityCreated(identity, msg.sender, owner,
            recoveryKey);
    }

    /// @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
    /// @param owner Key who can use this contract to
        control proxy. Given full power
    /// @param recoveryKey Key of recovery network or
        address from seed to recovery proxy
    /// @param destination Address of contract to be
        called after proxy is created
    /// @param data of function to be called at the
        destination contract
    function createIdentityWithCall(address owner,
        address recoveryKey, address destination, bytes
        data) public validAddress(recoveryKey) {
        Proxy identity = new Proxy();
        owners[identity][owner] = now - adminTimeLock; //
            This is to ensure original owner has full
            power from day one
        recoveryKeys[identity] = recoveryKey;
        LogIdentityCreated(identity, msg.sender, owner,
            recoveryKey);
        identity.forward(destination, 0, data);
    }

    /// @dev Allows a user to transfer control of
        existing proxy to this contract. Must come through
        proxy
    /// @param owner Key who can use this contract to
        control proxy. Given full power
    /// @param recoveryKey Key of recovery network or
        address from seed to recovery proxy
    /// Note: User must change owner of proxy to this
        contract after calling this
    function registerIdentity(address owner, address
        recoveryKey) public validAddress(recoveryKey) {
        require(recoveryKeys[msg.sender] == 0); // Deny
            any funny business
        owners[msg.sender][owner] = now - adminTimeLock;
```

```
        // This is to ensure original owner has full
           power from day one
        recoveryKeys[msg.sender] = recoveryKey;
        LogIdentityCreated(msg.sender, msg.sender, owner,
            recoveryKey);
    }

    /// @dev Allows a user to forward a call through
       their proxy.
    function forwardTo(Proxy identity, address
       destination, uint value, bytes data) public
       onlyOwner(identity) {
        identity.forward(destination, value, data);
    }

    /// @dev Allows an olderOwner to add a new owner
       instantly
    function addOwner(Proxy identity, address newOwner)
       public onlyOlderOwner(identity) rateLimited(
       identity) {
        require(!isOwner(identity, newOwner));
        owners[identity][newOwner] = now - userTimeLock;
        LogOwnerAdded(identity, newOwner, msg.sender);
    }

    /// @dev Allows a recoveryKey to add a new owner with
        userTimeLock waiting time
    function addOwnerFromRecovery(Proxy identity, address
        newOwner) public onlyRecovery(identity)
       rateLimited(identity) {
        require(!isOwner(identity, newOwner));
        owners[identity][newOwner] = now;
        LogOwnerAdded(identity, newOwner, msg.sender);
    }

    /// @dev Allows an owner to remove another owner
       instantly
    function removeOwner(Proxy identity, address owner)
       public onlyOlderOwner(identity) rateLimited(
       identity) {
        // an owner should not be allowed to remove
           itself
        require(msg.sender != owner);
        delete owners[identity][owner];
        LogOwnerRemoved(identity, owner, msg.sender);
    }

    /// @dev Allows an owner to change the recoveryKey
       instantly
```

```
    function changeRecovery(Proxy identity, address
        recoveryKey) public
        onlyOlderOwner(identity)
        rateLimited(identity)
        validAddress(recoveryKey)
    {
        recoveryKeys[identity] = recoveryKey;
        LogRecoveryChanged(identity, recoveryKey, msg.
            sender);
    }

    /// @dev Allows an owner to begin process of
        transfering proxy to new IdentityManager
    function initiateMigration(Proxy identity, address
        newIdManager) public
        onlyOlderOwner(identity)
        validAddress(newIdManager)
    {
        migrationInitiated[identity] = now;
        migrationNewAddress[identity] = newIdManager;
        LogMigrationInitiated(identity, newIdManager, msg
            .sender);
    }

    /// @dev Allows an owner to cancel the process of
        transfering proxy to new IdentityManager
    function cancelMigration(Proxy identity) public
        onlyOwner(identity) {
        address canceledManager = migrationNewAddress[
            identity];
        delete migrationInitiated[identity];
        delete migrationNewAddress[identity];
        LogMigrationCanceled(identity, canceledManager,
            msg.sender);
    }

    /// @dev Allows an owner to finalize migration once
        adminTimeLock time has passed
    /// WARNING: before transfering to a new address,
        make sure this address is "ready to recieve" the
        proxy.
    /// Not doing so risks the proxy becoming stuck.
    function finalizeMigration(Proxy identity) public
        onlyOlderOwner(identity) {
        require(migrationInitiated[identity] != 0 &&
            migrationInitiated[identity] + adminTimeLock <
             now);
        address newIdManager = migrationNewAddress[
            identity];
```

```
        delete migrationInitiated[identity];
        delete migrationNewAddress[identity];
        identity.transfer(newIdManager);
        delete recoveryKeys[identity];
        // We can only delete the owner that we know of.
           All other owners
        // needs to be removed before a call to this
           method.
        delete owners[identity][msg.sender];
        LogMigrationFinalized(identity, newIdManager, msg
           .sender);
    }

    function isOwner(address identity, address owner)
       public constant returns (bool) {
        return (owners[identity][owner] > 0 && (owners[
           identity][owner] + userTimeLock) <= now);
    }

    function isOlderOwner(address identity, address owner
       ) public constant returns (bool) {
        return (owners[identity][owner] > 0 && (owners[
           identity][owner] + adminTimeLock) <= now);
    }

    function isRecovery(address identity, address
       recoveryKey) public constant returns (bool) {
        return r  }
}
```

- **UportRegistry**: uno per chain, serve per aggiungere attributi in forma chiave:valore alle identità, in particolare l'hash del DID Document caricato su IPFS. Il metodo `get` serve per andare a recuperare dal mapping `registry` il valore corrispondente alla determinata chiave `registrationIdentifier` inserito dal dato `issuer` su un determinato `subject`. Il metodo `set` serve per andare a inserire o aggiornare nel mapping `registry` un valore corrispondente alla determinata chiave `registrationIdentifier` inserito dal dato `issuer` su un determinato `subject`.

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

contract UportRegistry{
  uint public version;
  address public previousPublishedVersion;
  mapping(bytes32 => mapping(address => mapping(address
     => bytes32))) public registry;
```

```
  function UportRegistry(address
      _previousPublishedVersion) {
    version = 3;
    previousPublishedVersion = _previousPublishedVersion;
  }

  event Set(
    bytes32 indexed registrationIdentifier,
    address indexed issuer,
    address indexed subject,
    uint updatedAt);

  //create or update
  function set(bytes32 registrationIdentifier, address
      subject, bytes32 value){
       Set(registrationIdentifier, msg.sender, subject,
          now);
       registry[registrationIdentifier][msg.sender][
          subject] = value;
  }

  function get(bytes32 registrationIdentifier, address
      issuer, address subject) constant returns(bytes32){
       return registry[registrationIdentifier][issuer][
          subject];
  }
}
```

- `EthereumClaimsRegistry`: uno per chain, serve per aggiungere dichiarazioni ed attestazioni alle identità. La funzione `setClaim` inserisce una dichiarazione nel mapping registry riguardante l'indirizzo `subject` in forma chiave:valore, stessa cosa per `setSelfClaim` che però inserisce una dichiarazione sul chiamante stesso della funzione.

  `getClaim` recupera una dichiarazione dal registry fatta da un dato `issuer` al dato `subject`.

  `removeClaim` controlla prima di tutto che chi cerca di eliminare la dichiarazione sia il dichiarante o il soggetto della dichiarazione, se è così procede con l'eliminazione e l'emissione dell'evento `ClaimRemoved`.

```
//EthereumClaimsRegistry.sol
pragma solidity 0.4.19;


/// @title Ethereum Claims Registry - A repository
   storing claims issued
```

```
///        from any Ethereum account to any other
   Ethereum account.
contract EthereumClaimsRegistry {

    mapping(address => mapping(address => mapping(bytes32
        => bytes32))) public registry;

    event ClaimSet(
        address indexed issuer,
        address indexed subject,
        bytes32 indexed key,
        bytes32 value,
        uint updatedAt);

    event ClaimRemoved(
        address indexed issuer,
        address indexed subject,
        bytes32 indexed key,
        uint removedAt);

    /// @dev Create or update a claim
    /// @param subject The address the claim is being
       issued to
    /// @param key The key used to identify the claim
    /// @param value The data associated with the claim
    function setClaim(address subject, bytes32 key,
       bytes32 value) public {
        registry[msg.sender][subject][key] = value;
        ClaimSet(msg.sender, subject, key, value, now);
    }

    /// @dev Create or update a claim about yourself
    /// @param key The key used to identify the claim
    /// @param value The data associated with the claim
    function setSelfClaim(bytes32 key, bytes32 value)
       public {
        setClaim(msg.sender, key, value);
    }

    /// @dev Allows to retrieve claims from other
       contracts as well as other off-chain interfaces
    /// @param issuer The address of the issuer of the
       claim
    /// @param subject The address to which the claim was
        issued to
    /// @param key The key used to identify the claim
    function getClaim(address issuer, address subject,
       bytes32 key) public constant returns(bytes32) {
        return registry[issuer][subject][key];
```

```
    }

    /// @dev Allows to remove a claims from the registry.
    ///      This can only be done by the issuer or the
       subject of the claim.
    /// @param issuer The address of the issuer of the
       claim
    /// @param subject The address to which the claim was
        issued to
    /// @param key The key used to identify the claim
    function removeClaim(address issuer, address subject,
        bytes32 key) public {
        require(msg.sender == issuer || msg.sender ==
           subject);
        require(registry[issuer][subject][key] != 0);
        delete registry[issuer][subject][key];
        ClaimRemoved(msg.sender, subject, key, now);
    }
}
```

### 3.3.2 uPort off-chain: JWT per lo scambio di informazioni

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

### 3.3.3 JSON Web Token

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 siano quelle inviate inizialmente e che non siano state modificate nel frattempo.

### Formato di un JWT

Un esempio di JWT codificato è questo:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIyblF0aVFHNkNnbTFHWVRCYWFLQWdyNzZ1WTdpU2V4VWtxWCIsImlzcyI6IjJvRFp2TlVnbjc3dzJCS1RrZDlxS3BNZVVvOEVMOTRRTDVWIiwia2V5IjoiRG9jdW1lbnRvRGlJZGVudGl0YSIsInZhbHVlIjoiUW1UaHJXWmhHeEV5ZnBkWEtrUzRDMm1Nckd TZkw5dUhzemFyQkFqdHkyNVNRUiIsImlhdCI6MTUzNjI0MzEyMH0.7fdplWJBD5aYAZ10gWohyNljO618ue-fmK3sdRMs5Ug

Figura 3.4: Esempio di JWT codificato.

Come si vede un JWT è diviso in tre parti separate dai punti:

- Header: un oggetto JSON codificato in Base64 composto dal tipo di token (JWT) e dall'algoritmo di hash usato, nel caso in foto HMAC SHA256
- Payload: un oggetto JSON codificato in base64 che contiene le dichiarazioni che vengono fatte su un soggetto, nel caso di uPort su un'identità, e le informazioni su quella dichiarazione, ad esempio:

```
{
    "key": "CartaDiIdentita",
    "value": "hashIPFSdellacartadiidentita"
}
```

- Signature: un hash dell'Header codificato in base64, un ".", Payload codificato in base64 e un secret.

Come si deduce dall'immagine sopra, i JWT permettono di inviare molte informazioni in un formato molto ristretto, adatto quindi ad essere incluso in un URL di una richiesta HTTP, ad esempio.

uPort utilizza i JWT esattamente in questo modo: si inviano ad un unico endpoint `https://id.uport.me/req/[JWT]`, il quale si occuperà poi di decodificare e, in base al contenuto del payload, agire di conseguenza richiedendo informazioni, aggiungendo dichiarazioni o richiedendo di firmare transazioni all'utente specificato nel JWT.

### 3.3.4 uPort e la sovranità del dato

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 utilizzarle 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 le 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:

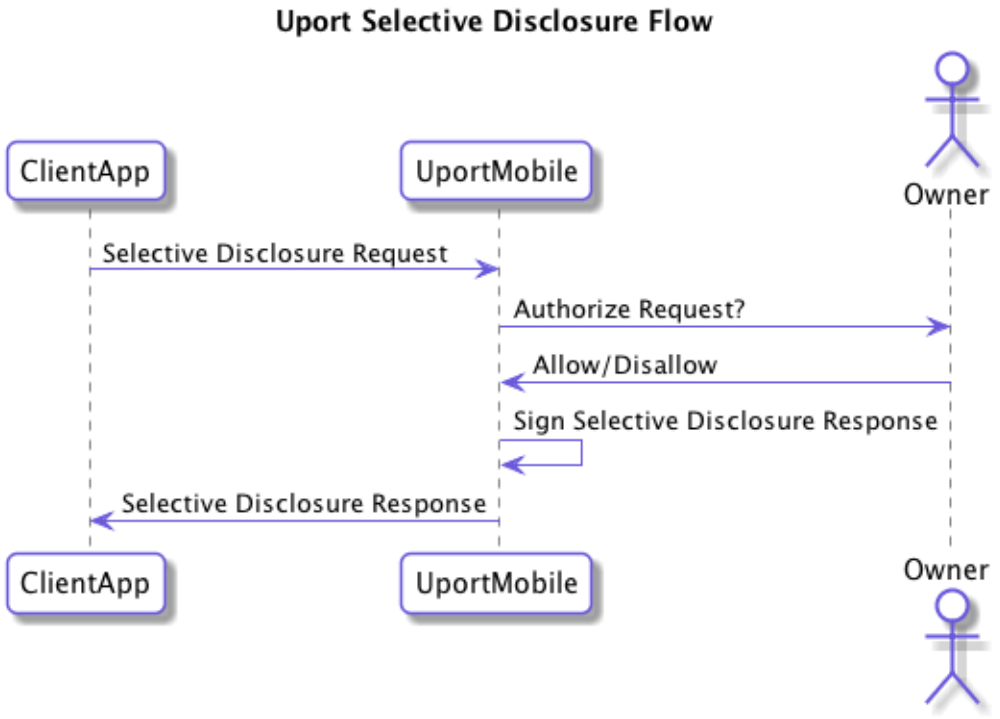


Figura 3.5: Diagramma di sequenza per la richiesta di informazioni ad un utente uPort.

La **Selective Disclosure Request** fatta dal servizio che richiede le credenziali all'utente è 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: 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.

| Nome | Descrizione | Obbligatorio |
|---|---|---|
| `type` | Deve avere valore shareReq | Sì |
| `iss` | Il MNID dell'identità firmataria | Sì |
| `iat` | Il momento del rilascio | Sì |
| `exp` | Momento di scadenza del JWT | No |
| `callback` | URL di callback per restituire la risposta ad una richiesta | No |
| `net` | Id della rete della chain Ethereum dell'Identità. Es. 0x4 per rinkeby | No |
| `act` | Tipo dell'account Ethereum:<br>- General: scelta dell'utente (default)<br>- Segregated: un account basato su uno smart contract unico sarà creato per l'app richiedente<br>- Keypair: un account basato su una coppia di chiavi unica sarà creato per l'app richiedente<br>- Devicekey: richiede una nuova device key per un account su chain privata<br>- None: non viene restituito nessun account | No |
| `requested` | Le attestazioni autofirmate da un utente. Vettore di tipi di attestazioni<br>per attestazioni autofirmate. Ad es: [“name, “email”] | No |
| `verified` | Le attestazioni verificate richieste da un utente. Vettore di tipi di attestazioni per attestazioni autofirmate. Es: [“name”, “email”] | No |
| `permissions` | Un vettore di permessi richiesti. Al momento sono solo supportate le notifications | No |
| `boxPub` | Chiave pubblica del'identità richiedente, usata per criptare i messaggi inviati all'URL di callback | No |
| `issc` | Le dichiarazioni auto firmate dal `iss` di questo messaggio, sia come oggetto di tipi di dichiarazioni per le dichiarazioni autofirmate oppure l'hash IPFS dell'oggetto equivalente. | No |
| `vc` | Un vettore di dichiarazioni verificate (JWT) o l'hash IPFS dell'oggetto equivalente riguardanti il `iss` del messaggio | No |

Tabella 3.1: Campi di una Selective Disclosure Request

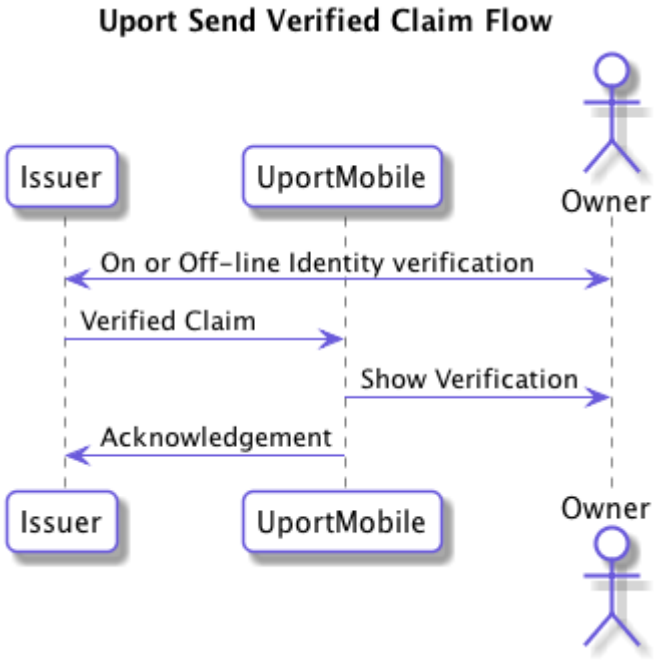


Figura 3.6: Diagramma di sequenza per l'aggiunta di dichiarazioni su un'identità uPort.

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.

| Nome | Descrizione | Obbligatorio |
|---|---|---|
| `iss` | Il DID dell'identità firmataria | Sì |
| `sub` | Il DID dell'identità interessata | Sì |
| `type` | Il tipo di attestazione | No |
| `exp` | Momento di scadenza della dichiarazione | Sì |
| `claim` | Un oggetto contenente una o più dichiarazioni riguardanti `sub` | No |
| `issc` | Le dichiarazioni auto firmate dal `iss` di questo messaggio, sia come oggetto di tipi di dichiarazioni per le dichiarazioni autofirmate oppure l'hash IPFS dell'oggetto equivalente. | No |
| `vc` | Un vettore di dichiarazioni verificate (JWT) o l'hash IPFS dell'oggetto equivalente riguardanti il `iss` del messaggio | No |

Tabella 3.2: Campi di un Verified Claim

### 3.3.5 Perché è stato scelto uPort?

Tra le tre opzioni esplorate si è deciso di adottare uPort per i seguenti motivi:

- Basato su piattaforma Ethereum: per motivi di facilità di sviluppo e di deploy di blockchain privata è stato scelto Ethereum per BLINC, quindi Civic, basato su Rootstock, è stato scartato per questo motivo.
- Semplice e interoperabile con altri smart contract: uPort è un semplice layer di identità basato su smart contract Ethereum senza servizi o token aggiuntivi, motivo per cui è stato preferito a SelfKey.
- Grado di maturità: tra le tre opzioni esplorate, uPort è quello in stato più avanzato di sviluppo e ha il miglior supporto sia dal team di sviluppo che dalla community.

### 3.3.6 Stato del progetto

uPort è ancora in una fase prematura dello sviluppo, nella quale sono possibili stravolgimenti radicali dell'architettura sottostante, come è successo a luglio di quest'anno. [9]

Nonostante ciò, uPort fornisce giù una vasta documentazione ed una grande quantità di librerie e smart contract da poter integrare nelle proprie dApp.

Al momento è possibile creare identità ed aggiungere attributi e attestazioni ad esse tramite le loro librerie `uport-connect`[6] e `uport-credentials`[7], la prima da integrare in dApp composte da solo frontend, la seconda da utilizzare in dApp che sfruttano un backend in NodeJS.

È disponibile una terza libreria in JavaScript `uport-js-client` [8] che in teoria dovrebbe solo servire per simulare il comportamento dell'app mobile, ma che nel caso di BLINC nella sua prima versione è stata cruciale in quanto l'unica in grado di gestire l'intero ciclo di creazione di identità e di aggiunta di attributi ed attestazioni su di esse senza l'utilizzo dell'app mobile.

Sono in fase di sviluppo anche `uport-android-sdk` [9] e `uport-ios-sdk`[10], che sarebbero state fondamentali per BLINC fin da subito, ma dato che ancora ad oggi sono in fase molto embrionale siamo dovuti scendere ai compromessi architetturali che spiegherò nelle conclusioni.

## 3.4 Integrazione nel progetto BLINC

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

### 3.4.1 Gestione di dichiarazioni ed attestazioni

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

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.

[6] uPort Connect: https://github.com/uport-project/uport-connect
[7] uPort Credentials: https://github.com/uport-project/uport-credentials
[8] uPort JS Client: https://github.com/uport-project/uport-js-client
[9] uPort Android SDK: https://github.com/uport-project/uport-android-sdk
[10] uPort iOs SDK: https://github.com/uport-project/uport-ios-sdk

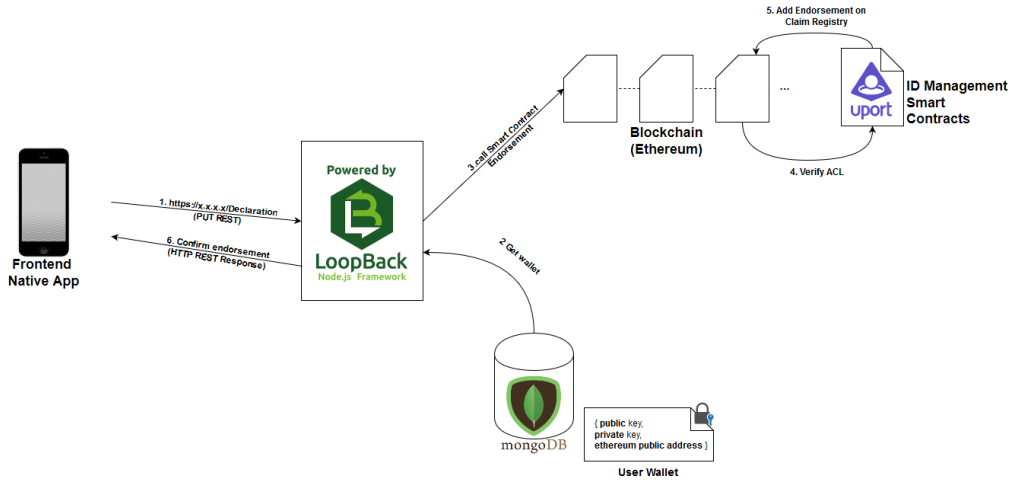


Figura 3.7: Flusso di gestione delle dichiarazioni sull'utente.

### Implementazione

Queste due funzioni da me implementate utilizzano la struttura delle funzioni richiamate dagli endpoint imposta da LoopBack basata su modelli, in questo caso il modello `Declaration` così formato:

```
{
  "name": "Declaration",
  "base": "Model",
  "idInjection": true,
  "options": {
    "validateUpsert": true
  },
  "properties": {
    "issuer": {
      "type": "string",
      "required": true
    },
    "subject": {
      "type": "string",
      "required": true
    },
    "key": {
      "type": "string",
      "required": true
    },
    "value": {
      "type": "string",
```

```
      "required": true
    }
  },
  "validations": [],
  "relations": {},
  "acls": [
    {
      "accessType": "*",
      "principalType": "ROLE",
      "principalId": "$everyone",
      "permission": "DENY"
    }
  ],
  "methods": {}
}
```

`getAllDeclarations` recupera il wallet di un dato migrante da MongoDB tramite il suo `id` e, se trovato, istanzia una uPort Identity la quale interroga l'`ethereum-claims-registry` istanziato sulla blockchain privata per ottenere tutte le attestazioni riguardanti l'identità e le ritorna in JSON.

```
/**
* 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);
    }
    );
};
```

`addDeclaration` permette ad un'identità uPort di aggiungere una attestazione ad un'altra identità. Recupera il wallet di una data identità da MongoDB tramite il suo `id` e, se trovato, istanzia una uPort Identity la quale, tramite le librerie di utilità uPort, crea un JWT firmato che vale da attestazione tramite la `credentials.attest`. La funzione `consume` si occupa di parsificare lo URI passato che contiene il JWT e inoltrarlo alla apposita funzione `addAttestationRequestHandler` di uPort JS Client che permette di aggiungere credenziali sull'`ethereum-claims-registry`.

```
/**
* Adds a declaration for a specific issuer
* @param {string} issuer the id of the owner of the
    declarations.
* @param {string} subject the subject of the declaration
* @param {string} key
* @param {string} value the value of the declaration itself
*
* @returns {Object} response the added declaration.
*/
Declaration.addDeclaration = (issuer, subject, key, value, cb
    ) => {
    let issuerUportIdentity, signer, credentials, declaration
        , response;

    Declaration.app.models.migrant.findOne(
    { where: { id: issuer } },
    async (err, instance) => {
        if (err) cb(err, null);

        try {
        issuerUportIdentity = initialize(instance);

        // Gets the SimpleSigner object with the private key
            of the user, which is needed to sign transactions
        signer = SimpleSigner(issuerUportIdentity.deviceKeys.
            privateKey);

        // Instantiates the Credentials class, a uPort class
            which simplifies the creation of signed
            attestation JWTs
        credentials = new Credentials({
            address: issuerUportIdentity.mnid,
            signer: signer,
            networks: {
            [issuerUportIdentity.network.id]: {
                ...issuerUportIdentity.network
            }
            }
        });

        // The attest function creates a signed attestation
            JWT
        declaration = await credentials.attest({
            sub: subject,
            claim: { [key]: value }
        });

        // The consume function is a UPortClient function
```

```
            which parses
        // uPort uris and relays them to the responsible
            functions
        response = await issuerUportIdentity.consume(
            `me.uport:add?attestations=${declaration}`
        );

        cb(null, response);
        } catch (error) {
        cb(error, null);
        }
    }
    );
};
```

### 3.4.2 Creazione di uPort Identity

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

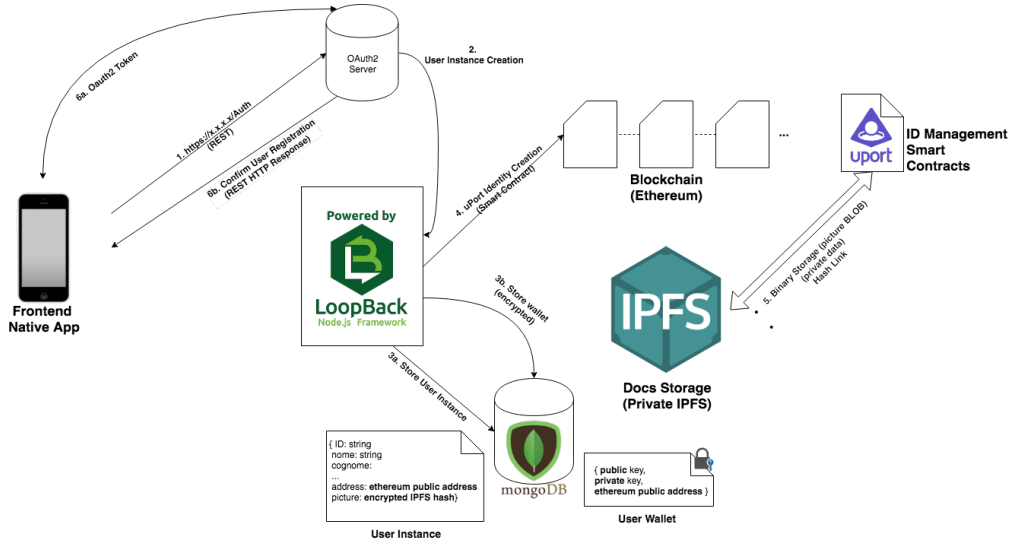


Figura 3.8: Flusso di creazione di identità.

Durante la registrazione viene istanziato sulla blockchain Ethereum privata lo smart contract Proxy associato all'utente, 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 blockchain.

#### Implementazione

Queste due funzioni da me implementate utilizzano la struttura delle funzioni richiamate dagli endpoint imposta da LoopBack basata su modelli, in questo caso il modello `Migrant` così formato:

```
{
    "name": "migrant",
    "base": "Model",
    "idInjection": false,
    "options": {
        "validateUpsert": true
    },
    "properties": {
```

```
        "id": {
        "type": "string",
        "required": "true"
        },
        "info": {
        "type": "object",
        "required": "true"
        },
        "deviceKeys": {
        "type": "object",
        "required": "true"
        },
        "recoveryKeys": {
        "type": "object",
        "required": "true"
        },
        "mnid": {
        "type": "string",
        "required": "true"
        },
        "initialized": {
        "type": "boolean",
        "required": "true"
        }
    },
    "validations": [],
    "relations": {
        "documents": {
        "type": "hasMany",
        "model": "Document",
        "foreignKey": ""
        },
        "accessTokens": {
        "type": "hasMany",
        "model": "token",
        "foreignKey": ""
        },
        "wallet": {
        "type": "hasOne",
        "model": "wallet",
        "foreignKey": ""
        }
    },
    "acls": [
        {
        "accessType": "*",
        "principalType": "ROLE",
        "principalId": "$authenticated",
        "permission": "ALLOW"
```

```
        }
    ],
    "methods": {}
    }
```

`getMigrant` va semplicemente a recuperare un Migrant in base alla sua email e password da MongoDB e lo restituisce.

```
/**
* Gets a migrant from the datasource
* @param {String} email
* @param {String} password
*
* @returns {Object} the migrant instance
*/
Migrant.getMigrant = (email, password, cb) => {
    Migrant.app.models.migrant.findOne(
    { where: { 'info.email': email, 'info.password': password
        } },
    (err, instance) => {
        if (err) cb(err, null);

        cb(null, instance);
    }
    );
};
```

`addMigrant` è la funzione fulcro di BLINC: crea o aggiorna un'identità uPort con gli attributi passati nel body della richiesta HTTP.

```
/**
* 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) {
        if (err) {
        cb(err, null);
        }
    });

    cb(null, migrantID);
    });
};
```

Le funzioni `createUportId` e `initializeUportId` fanno parte di una libreria scritta da me che funge da middleware tra la business logic raggiungibile tramite chiamate API e la libreria `uport-js-client` che simula il funzionamento dei protocolli uPort.

`createUportId` crea un'istanza di UPortClient con le informazioni passate dalla richiesta HTTP

precedente, crea un wallet per essa tramite `uportClient.initKeys()` e invia una transazione monetaria all'indirizzo dell'identità in modo che possa pagare il gas per le transazioni di istanziazione del Proxy e dell'inserimento del DID Document all'interno dello UportRegistry.

```
/**
 * creates a UportClient instance.
 * @param {Object} info personal info of the user.
```

```
 *
 * @returns {Promise<UPortClient>} the promise to have a
    UPortClient with specified infos.
 */
const createUportId = (info = {}) => {
  return new Promise(async (resolve, reject) => {
    try {
      const uportClient = new UPortClient(config, { info });

      // Creates a keypair for the uPort client
      uportClient.initKeys();

      const accounts = await web3.eth.getAccounts();

      // Gets the first Ethereum account in the private
         blockchain which is the one who is mining hence
      // the one with funds
      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);
    }
  });
};
```

`initializeUportId` richiama `initializeIdentity`,

metodo di `UPortClient` che chiama la `createIdentity` dello smart contract IdentityManager, dopodiché salva il DID Document dell'identità su IPFS e ne registra l'hash sullo smart contract Registry.

```
/**
* Initializes a UportClient instance, meaning that it will
   get 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) {
        // Crates an instance of the IPFS class, which
           allows us to communicate with the IPFS node
           specified in the configuration
        this.ipfs = new IPFS(networkConfig.ipfsConfig);

        this.registryNetwork = {
            [this.network.id]: {
                registry: this.network.registry,
                rpcUrl: this.network.rpcUrl
            }
        }
    };

    const registry = new UportLite({
        networks: this.registryNetwork
    });
    // Function that uses the UportLite library
    this.registry = address =>
        new Promise((resolve, reject) => {
            registry(address, (error, profile) => {
                if (error) return reject(error);

                resolve(profile);
            });
    });

    this.verifyJWT = jwt => verifyJWT({ registry: this.
       registry, address: this.mnid }, jwt);
    // Sets the HttpProvider, necessary object for EthJS
       library in order
    // to interact with the Ethereum blockchain
```

```
        this.provider = config.provider || new HttpProvider(
            this.network.rpcUrl);
        // Creates the EthJS object with the just created
            HttpProvider as a parameter
        this.ethjs = this.provider ? new EthJS(this.provider)
             : null;
        // Sets addresses of contracts passed in the
            configuration object
        this.claimsRegistryAddress = this.network.
            claimsRegistry;
        this.registryAddress = this.network.registry;
        this.identityManagerAddress = this.network.
            identityManager;
        // Necessary to do some actions, if identity is
            recreated from MongoDB config.initialized has a
            value,
        // else identity is initialized later
        this.initialized = config.initialized || false;

        this.consume = this.consume.bind(this);
    }
}
```

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

```
initializeIdentity(initDdo) {
    if (!this.network)
        return Promise.reject(new Error('No network
            configured'));
    const IdentityManagerAddress = this.
        identityManagerAddress;
    // Creates an object contract with a specific ABI
    // (Application Binary Interface, a low level API-like
        for contracts,
    // written in JSON as result of contract compilation)
    // and at a specific address for the IdentityManager
        contract
    const IdentityManager = Contract(IdentityManagerArtifact.
        abi).at(
        IdentityManagerAddress
    );
    // Creates keypair and an ethereum address for the user
        and for recovery
    this.initKeys();
    // Calls contract function to create an Identity
    const uri = IdentityManager.createIdentity(
        this.deviceKeys.address,
        this.recoveryKeys.address
    );
```

```
    // The consume function is a uport-js-client function
       which, given a URI which conforms to the uPort
       Protocol specs,
    // parses it and, basing on the given URI format, sends a
        tx, adds an attestation or requests credentials.
    return this.consume(uri)
    .then(this.getReceipt.bind(this))
    .then(receipt => {
        const log = receipt.logs[0];
        const createEventAbi = IdentityManager.abi.filter(
            obj => obj.type === 'event' && obj.name === '
               IdentityCreated'
        )[0];
        // Gets the Proxy contract address for the identity
           from
        // the event emitted by the createIdentity function
        this.id = decodeEvent(createEventAbi, log.data, log.
           topics).identity;
        // Creates the MNID for the identity, which is the
           base58 encoding of the network id and the identity
            Proxy address
        this.mnid = mnid.encode({ network: this.network.id,
           address: this.id });
        this.initTransactionSigner(IdentityManagerAddress);
        const baseDdo = {
            '@context': 'http://schema.org',
            '@type': 'Person',
            publicKey: this.deviceKeys.publicKey
        };
        const ddo = Object.assign(baseDdo, initDdo);
        // The code for the writeDDO is below, at a high
           level it uploads the ddo object to IPFS
        // and adds it to the Registry smart contract
        return this.writeDDO(ddo);
    })
    .then(this.ethjs.getTransactionReceipt.bind(this.ethjs))
    .then(receipt => {

        this.initialized = true;
        return;
    });
}

writeDDO(newDdo) {
    // Creates an object contract with a specific ABI
    // and at a specific address for the Registry contract
    const Registry = Contract(RegistryArtifact.abi).at(this.
       network.registry);
    return this.getDDO()
```

```
        .then(ddo => {
        // If the Identity Document already exists it doesn't
            overwrite with the given one,
        // otherwise it adds to IPFS the newDdo
        ddo = Object.assign(ddo || {}, newDdo);
        return new Promise((resolve, reject) => {
            this.ipfs.add(Buffer.from(JSON.stringify(ddo)), (
                err, result) => {
            if (err) reject(new Error(err));
            resolve(result);
            });
        });
        })
        .then(res => {
        const hash = res[0].hash;
        const hexhash = new Buffer(base58.decode(hash)).
            toString('hex');
        // Removes Qm from ipfs hash, which specifies length
            and hash
        const hashArg = `0x${hexhash.slice(4)}`;
        const key = 'uPortProfileIPFS1220';
        // Writes on the Registry contract the association
            between the Proxy contract address
        // for the identity and the IPFS hash of its Identity
            Document which contains its pubkey
        return Registry.set(key, this.id, hashArg);
        })
        .then(this.consume.bind(this));
}
```

# Capitolo 4

# Conclusioni

## 4.1 Risultati ottenuti

Nella prima versione di BLINC io ed i miei colleghi siamo riusciti a creare una versione base del sistema di gestione di identità ed attestazioni su di esse e sui documenti che le riguardano (salvati su storage decentralizzato IPFS) secondo l'architettura descritta in precedenza.

In particolare ho sviluppato il sistema di creazione e gestione delle identità e di creazione e visualizzazione di attestazioni e dichiarazioni creando delle apposite API REST consumabili da web e da mobile.

## 4.2 Problemi aperti

### 4.2.1 Su Ethereum e blockchain in generale

#### Scalabilità

Eseguire calcoli e salvare molti dati su una blockchain è troppo lento e costoso.

Per quanto riguarda il salvataggio di moli di dati importanti si ha una soluzione che consiste nel memorizzare soltanto i riferimenti crittografici ai dati che sono poi salvati su altri tipi di storage decentralizzato, come IPFS o Swarm[1].

Per quanto riguarda invece la velocità di processamento delle transazioni, per fare un confronto, il circuito VISA processa fino a 24.000 transazioni al secondo [10], Ethereum ne processa al massimo 15 al secondo, con grandi problemi quando la rete è molto utilizzata (come nel dicembre 2017 con

[1]Swarm: http://theswarm.eth.show/

il fenomeno CryptoKitties [11]) che causano ritardo nel processamento di transazioni e aumento del costo del gas.

#### Privacy

La blockchain garantisce immutabilità, correttezza e trasparenza dei dati e delle computazioni: se le prime due proprietà sono ben accette in tutti i casi in cui la blockchain è un valore aggiunto, l'ultima è più una criticità che un punto di forza in molte applicazioni che hanno a che fare con dati sensibili, come ad esempio applicazioni sanitarie.

### 4.2.2 Su uPort

#### SDK per mobile

Al momento sono in fase di sviluppo SDK sia per Android che per iOS, ma al momento in cui si doveva implementare una prima versione dell'architettura di BLINC non erano abbastanza mature per le nostre necessità ( non permettevano ad esempio di poter interagire con una rete Ethereum privata ma soltanto con le testnet pubbliche) e soprattutto non era ancora disponibile l'SDK iOS, una tegola insormontabile per il progetto in quanto è necessario sviluppare una app per entrambi i maggiori sistemi operativi mobile.

#### Parziale centralizzazione

Al momento uPort utilizza ancora dei microservizi centralizzati per rendere possibili alcune parti fondamentali dell'architettura come il server di messaging Chasqui o il server che finanzia le transazioni Sensui.

### 4.2.3 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.

### Utilizzo di librerie e protocolli uPort datati o deprecati

Dato che nella prima versione di BLINC non è stato possibile spostare il wallet degli utenti sul loro telefono a causa dell'immaturità degli SDK mobile, è stato necessario spostare la gestione dell'identità sul server.

Questa necessità, oltre a quella di dover creare le identità su una blockchain privata, ha precluso l'utilizzo di molte librerie (che erano anche quelle più supportate) di uPort, portandomi di fatto a dover utilizzare l'unica che soddisfacesse i nostri requisiti, ovvero la `uport-js-client` già citata in precedenza.

Usando la libreria sono stati però riscontrati diversi problemi:

- Errori strutturali e sintattici del codice: utilizzo di alcune variabili non dichiarate e errori nell'esecuzione di alcune funzioni, sistemati velocemente.

- Utilizzo di URI deprecati: gli URI di richiesta di informazioni o di creazione di attestazioni utilizzati nella libreria erano diversi da quelli specificati nella documentazione di uPort e deprecati da tempo. Dato che l'utilizzo è solo temporaneo per il progetto ho preferito mantenere gli URI deprecati per non perdere troppo tempo a rifattorizzare l'intera libreria.

- La memorizzazione delle attestazioni non veniva fatta *on chain*: come da requisiti, le attestazioni sugli utenti devono essere immutabili e quindi salvate su blockchain, ma la versione iniziale di `uport-js-client` non permetteva ciò, come si vede nel codice seguente, che va semplicemente ad aggiungere le attestazioni come campi di un oggetto `UPortClient`:

```
addAttestationRequestHandler(uri) {
  const params = getUrlParams(uri)

  // Gets the JWT from the URI
  const attestations = Array.isArray(params.attestations)
      ? params.attestations : [params.attestations]

  for (let jwt in attestations) {
    jwt = attestations[jwt]

    // Decodes the JWT passed in the URI and gets the
       payload part
    const json = decodeToken(jwt).payload
    const key = Object.keys(json.claim)[0]
```

```
    if (this.network) {
      this.verifyJWT(jwt).then(() => {

        // This just adds attestations to the UportClient
            object, while we want attestations to be
           stored on the blockchain
        this.credentials[key] ? this.credentials[key].
           append({jwt, json}) : this.credentials[key] =
           [{jwt, json}]
      }).catch(console.log)
    }
  }
}
```

Ciò che si vuole per BLINC è però che tutte le attestazioni siano visibili a tutti e approvate su blockchain, perciò ho dovuto effettuare delle modifiche direttamente alla libreria per inserire correttamente attestazioni sull'`ethereum-claims-registry`:

```
async addAttestationRequestHandler(uri) {

  // Creates an object contract with a specific ABI
  // and at a specific address for the
     EthereumClaimsRegistry contract
  const ClaimsReg = Contract(
     EthereumClaimsRegistryArtifact.abi).at(
    this.network.claimsRegistry
  );

  const params = getUrlParams(uri);
  const attestations = Array.isArray(params.attestations)
    ? params.attestations
    : [params.attestations];

  let i = 0;

  return Promise.all(

    // This method also adds the possibility to add more
       than one attestation a time
    attestations.map(async (jwt, i) => {
      const json = decodeToken(jwt).payload;
      const issAddress = json.iss;
      const subAddress = json.sub;
      const key = Object.keys(json.claim)[i];
      const value = Object.values(json.claim)[i];

      // Creates the transaction URI that will be given
         to the consume function
```

```
      // which will call send a transaction to the
         EthereumClaimsRegistry contract on the private
         blockchain
      // with the parameters passed in the JWT
      const tx = ClaimsReg.setClaim(subAddress, key,
         value);

      try {
        const txHash = await this.consume(tx);
        const receipt = await this.getReceipt(txHash);
        const log = receipt.logs[0];

        const claimSetEventAbi = ClaimsReg.abi.filter(
          obj => obj.type === 'event' && obj.name === '
             ClaimSet'
        )[0];


        // Gets the ClaimSet event emitted by the
           setClaim function, to give feedback to the
           NodeJS backend
        // that the claims were correctly set
        const decodedEvent = decodeEvent(
          claimSetEventAbi,
          log.data,
          log.topics
        );

        // Returns an object which is added to the
           response array
        return {
          key: ethutil.toUtf8(decodedEvent.key),
          value: ethutil.toUtf8(decodedEvent.value),
          issuer: decodedEvent.issuer,
          sub: decodedEvent.subject
        };
      } catch (error) {
        return Promise.reject(error);
      }
    })
  );
}
```

## 4.3 Possibili scenari futuri

### 4.3.1 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 (state channels) [2]
- Plasma (side-chains) [3]
- Casper (Proof-of-Stake) + Sharding [4]

#### Privacy

Diverse aziende stanno lavorando per rendere possibile la computazione e il salvataggio di dati privati, tra cui Keep[5] [12] ed Enigma[6]: la prima avvalendosi di contenitori off-chain di dati privati e la seconda tramite l'uso di smart contract privati [13].

### 4.3.2 Su uPort

#### SDK mobile

Lo sviluppo delle SDK procede come si vede sui repository GitHub di uPort, quindi si raggiungerà il livello di maturità 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

[2]Raiden: https://raiden.network/
[3]Plasma: http://plasma.io/
[4]Sharding: https://github.com/ethereum/wiki/wiki/Sharding-FAQs
[5]https://keep.network/
[6]https://enigma.co/

smart contract IdentityManager e registrazione dell'Identity Document su smart contract Registry), la registrazione di un'identità nella nuova architettura richiederà soltanto la creazione di un account Ethereum, ed è quindi gratuita, dettaglio non da poco perchè permetterà di rimuovere il loro server di finanziamento delle transazioni Sensui, andando di fatto ad avvicinarsi ad una architettura per la gestione di identità completamente decentralizzata.

### 4.3.3 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 privati e non 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 sfrutterà 3Box[7], ovvero l'implementazione del team di uPort della specifica degli Identity Hub.

[7] 3Box: http://alpha.3box.io/

# Bibliografia


[1] A. Brownworth. https://anders.com/blockchain/.

[2] A. M. A. e Gavin Wood, "Mastering ethereum." https://github.com/ethereumbook/ethereumbook.

[3] P. Rizzo, "Wedbush report projects 400usd bitcoin price by 2016." https://www.coindesk.com/wedbush-report-projects-400-bitcoin-price-by-2016/, Luglio 2015.

[4] S. Higgins, "Ibm reveals proof of concept for blockchain-powered internet of things." https://www.coindesk.com/ibm-reveals-proof-concept-blockchain-powered-internet-things/, Gennaio 2015.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Ottobre 2008.

[6] https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs#what-are-the-benefits-of-proof-of-stake-as-opposed-to-proof-of-work, 2018.

[7] w3c, "Decentralized identifiers (dids)." https://w3c-ccg.github.io/did-spec/#did-documents, Agosto 2018.

[8] M. Sena, "Privacy preserving identity system for ethereum dapps." https://medium.com/uport/privacy-preserving-identity-system-for-ethereum-dapps-a3352d1a93e8, Aprile 2018.

[9] P. Braendgaard, "Preparing your apps for our new standards based identity architecture." https://medium.com/uport/preparing-your-apps-for-our-new-standards-based-identity-architecture-41c9a55b1
Luglio 2018.

[10] VISA. https://usa.visa.com/dam/VCOM/download/corporate/media/visanet-technology/visa-net-booklet.pdf, 2018.

[11] BBC, "Cryptokitties craze slows down transactions on ethereum." https://www.bbc.com/news/technology-42237162, Dicembre 2017.

[12] M. Luongo and C. Pon, "The keep network: A privacy layer for public blockchains," 2017.

[13] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," 2017.