

UNIVERSITÀ  
DI PAVIA

FACOLTA' DI INGEGNERIA  
DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE  
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA E INFORMATICA

TESI DI LAUREA

## **Un sistema basato su Blockchain per la protezione di una Internet of Things**

*Candidato:*

Luca Fornasari

*Relatore:*

Prof. Antonino Nocera

Anno accademico  
2019/2020

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Protezione IoT . . . . .	2
<b>2</b>	<b>Blockchain</b>	<b>4</b>
2.1	Quando nasce . . . . .	4
2.2	Struttura di una blockchain . . . . .	4
<b>3</b>	<b>Soluzione proposta</b>	<b>7</b>
3.1	Punto di partenza . . . . .	7
3.2	Trust e Reputation . . . . .	8
<b>4</b>	<b>Implementazione</b>	<b>10</b>
4.1	Modello semplificato . . . . .	10
4.2	Requisiti Client e Smart Contract . . . . .	11
4.2.1	Client . . . . .	11
4.2.2	Smart Contract . . . . .	12
4.3	Linguaggio e Blockchain . . . . .	13
<b>5</b>	<b>Codice</b>	<b>15</b>
5.1	Client . . . . .	15
5.2	Smart Contract . . . . .	18
5.3	Demo e test . . . . .	24
<b>6</b>	<b>Criticità</b>	<b>25</b>
6.1	Soluzioni . . . . .	26
<b>7</b>	<b>Conclusioni</b>	<b>28</b>
	<b>Bibliografia</b>	<b>29</b>

# Capitolo 1

## Introduzione

### 1.1 Protezione IoT

Negli ultimi anni il mondo dell'Internet of Things ha subito un'evoluzione rapida e senza precedenti. I dispositivi *smart* sono entrati a far parte della vita e della quotidianità di tutti, offrendo comfort e vantaggi di cui ormai è difficile privarsi.

Tuttavia, questa rapida crescita dell'IoT porta con sé nuove sfide relative alla protezione degli oggetti che ne fanno parte. Infatti quando parliamo di IoT facciamo riferimento ad un network composto da un gruppo estremamente eterogeneo di oggetti smart: si va dal più semplice sensore fino alla più complessa automobile con guida autonoma. Comprendiamo subito, dunque, che i limiti da tenere in considerazione sono molti, quali ad esempio:

- (i) batteria;
- (ii) memoria;
- (iii) potenza computazionale.

Il problema fondamentale è garantire un sistema *protetto, autonomo, trasparente e decentralizzato* a un insieme di dispositivi eterogenei dal punto di vista di specifiche tecniche e servizi offerti.

Con il concetto di *protezione* e *autonomia* si vuole sottolineare il fatto che a comporre l'IoT vi sono oggetti che non gestiscono solo dati personali (come il sonno, i passi, i battiti cardiaci, ecc...), ma con una prospettiva molto più ampia; lo sviluppo tecnologico ci porta verso un futuro che vedrà le nostre città piene di sensori smart, semafori smart, videocamere smart, ecc.

E' fondamentale, dunque, affrontare il problema nel migliore dei modi.

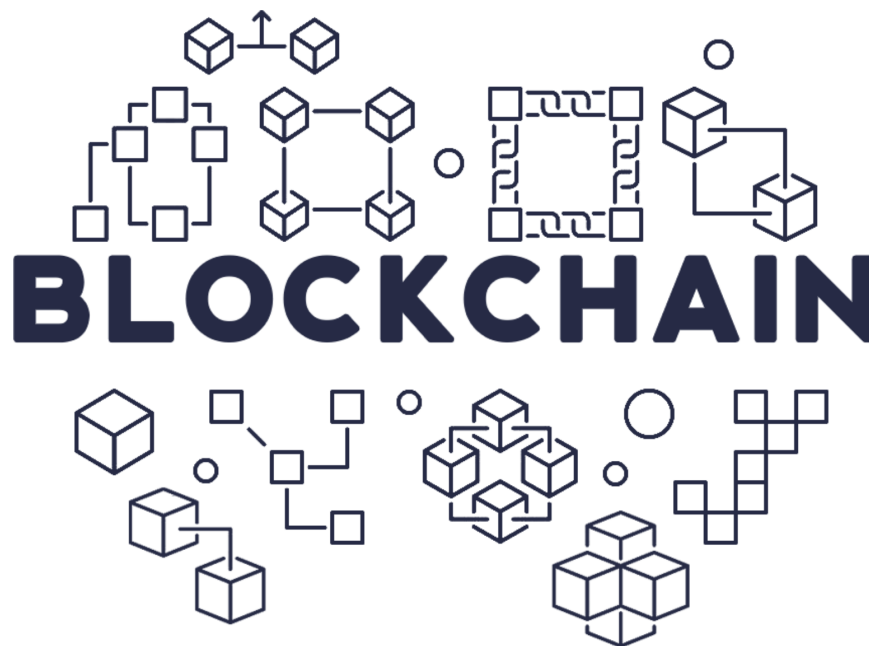
L'idea di sicurezza e autonomia si traduce in un ambiente in cui ogni dispositivo lavora in modo totalmente indipendente ed con una visione globale dell'operato di tutti gli altri dispositivi, con la possibilità di evitare contatti e comunicazioni nel momento in cui si

nota un comportamento non corretto.

La *trasparenza* e la *decentralizzazione* sono alla base di un network in cui non esistono nodi con diritti in più di altri: tutti devono essere sullo stesso piano di autorizzazione. Inoltre è importante che i dati scambiati nelle comunicazioni tra oggetti siano completamente leggibili per permettere la valutazione dei comportamenti come accennato precedentemente.

Per raggiungere tali obiettivi è necessario definire un meccanismo che associ una sequenza di azioni ad ogni singolo oggetto per identificarne il comportamento ed un metodo di autenticazione per mappare ogni singola azione.

La tecnologia su cui fondare tale sistema è la *Blockchain* [7].



## Capitolo 2

# Blockchain

### 2.1 Quando nasce

La tecnologia **blockchain** viene presentata nel 2008 in occasione della nascita del Bitcoin; nasce quindi per supportare un network capace di garantire transazioni economiche sicure e soprattutto che non coinvolgano attori terzi (come le banche ad esempio) [1].

Blockchain, oltre a garantire transazioni sicure, introduce una importante rivoluzione riguardo allo scambio di "beni" digitali; infatti, quando si vuole condividere un file con un'altra persona, ciò che avviene tecnicamente è una copia di byte che da un computer passano ad un altro, così come quando attraverso il servizio di home banking si decide di effettuare una transazione, vengono modificati dei byte all'interno di un enorme database salvato sui server centrali della banca.

Con l'avvento di blockchain è diventato possibile lo scambio di informazioni digitali in maniera univoca: è possibile decidere di rinunciare alla proprietà di un particolare bene (ad esempio un Bitcoin, un immobile, un'opera d'arte ecc...) cedendola a qualcun'altro.

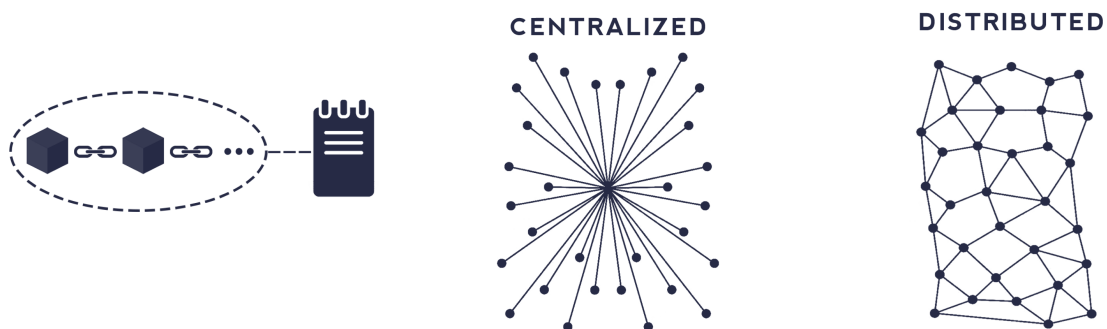
### 2.2 Struttura di una blockchain

In un white paper pubblicato nel novembre 2008, Satoshi Nakamoto ha proposto Bitcoin come sistema di pagamento elettronico fondato su un network peer-to-peer decentralizzato, pienamente operativo senza la necessità di una parte terza di fiducia. In contemporanea, viene introdotta la blockchain, ad oggi riconosciuta come rivoluzionaria e sulla quale sono riposte grandi aspettative.

Tuttavia, pur essendo ormai sulla bocca di tutti, la tecnologia blockchain non è ancora compresa da molti e spesso viene associata solamente a Bitcoin come strumento di pura speculazione.

Per definizione, una blockchain può essere descritta come un **distributed ledger**, ovvero un "registro distribuito" su cui vengono annotate tutte le transazioni avvenute, dalla prima fino all'ultima. Possiamo quindi immaginare un grosso libro mastro organizzato in *blocchi*, dove ogni blocco è collegato al precedente (come una *catena*) e contiene un insieme di transazioni.

Quindi, in che modo si può garantire la tanto famosa sicurezza nel momento in cui si decide di far parte di un network di questo tipo? La caratteristica vincente di tale tecnologia è il suo essere "distribuita": il ledger di cui abbiamo parlato non è centralizzato, non è (ad esempio) salvato all'interno dei server principali di una banca, ma è a disposizione di tutti i peer della rete. Capiamo subito che un sistema del genere è estremamente più sicuro, infatti nel caso in cui un nodo decidesse di modificare qualcosa all'interno della copia del registro in suo possesso, a quel punto avrebbe una versione non più valida della blockchain. In termini di sicurezza, più una blockchain è decentralizzata (ovvero più nodi ne fanno parte), più essa aumenta.



Per comprendere a pieno i meccanismi che tengono in piedi un network del genere sarebbe necessario un paper scientifico a se, tuttavia ai fini di capire il lavoro di tesi è importante vedere quali sono i principi più importanti. I concetti alla base della blockchain sono 2: **crittografia** e **consenso**.

- **Crittografia**: la codifica Hash è uno strumento prezioso, per definizione è una funzione che associa ad un elemento di un dominio un solo elemento di un codominio e l'operazione inversa è enormemente più costosa. In una blockchain quasi tutte le informazioni sono codificate in Hash (id del blocco, address di un nodo, id di una transazione ecc...) e inoltre ogni peer è provvisto di **chiave privata** (per firmare le transazioni) e di **chiave pubblica** (ottenuta a partire dalla privata, per permettere di ricevere transazioni).
- **Consenso**: per definizione è la conoscenza comune dei processi di gestione e controllo del network. I nodi che compongono una blockchain possono assumere il

ruolo dei cosiddetti **miners**, ovvero possono decidere di fornire la propria potenza computazionale al fine di validare nuovi blocchi e transazioni. Volendo garantire sicurezza è importante che non esistano attori terzi che detengono permessi "in più" rispetto ad altri: col meccanismo degli algoritmi di consenso è così, ogni peer collabora risolvendo algoritmi complessi (PoW, PoS ecc...) con l'obiettivo di raggiungere un "accordo generale" per accodare alla catena nuovi blocchi. Ritornando per un attimo al problema della sicurezza IoT, è impensabile immaginare uno scenario dove ogni oggetto risolve costantemente tali algoritmi, ovviamente esistono diversi tipi di peer che compongono una blockchain e nella nostra visione ogni dispositivo dovrà avere capacità computazionali necessarie alla sola connessione e lettura.

I settori di applicazione del paradigma blockchain sono potenzialmente infiniti, dal momento che essa permette "la disintermediazione e la decentralizzazione di transazioni di qualsiasi tipo e tra tutte le parti a livello globale" (Swan 2015, p. x) ed è "potenzialmente in grado di riconfigurare tutte le attività umane in modo pervasivo come ha fatto il Web" (Swan 2015, p. vii). Per questa ragione la blockchain è ritenuta da alcuni "fondamentale per il progresso della società alla pari della Magna Charta o della Stele di Rosetta" (Swan 2015, p. viii) ed è spesso definita come un "Cigno Nero" – cioè un avvenimento storico di grande impatto che non può essere previsto, crea sorpresa nell'osservatore e può essere razionalizzato soltanto a posteriori (Taleb, 2007).

## Capitolo 3

# Soluzione proposta

### 3.1 Punto di partenza

Il lavoro su cui si fonda questa tesi è la ricerca del professore Antonino Nocera:

*A two-tier Blockchain framework to increase protection and autonomy of smart objects in the IoT.*

[5]

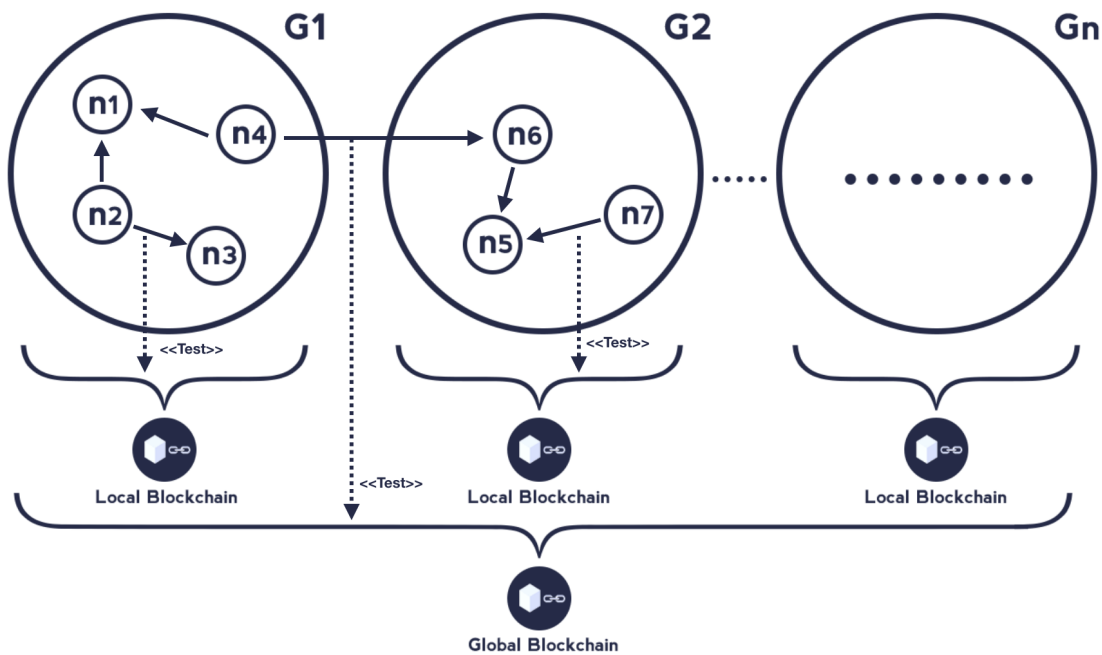
Nel paper viene descritto un approccio **community-based** al problema della sicurezza IoT: il grande network dell'internet delle cose viene visto diviso in gruppi di dispositivi. Ogni gruppo è composto da oggetti che presentano un certo livello di ridondanza dal punto di vista dei servizi offerti. Questa visione non è del tutto distante dalla realtà, infatti è facile riuscire a formare diversi gruppi: oggetti domotici all'interno di una casa, oggetti installati all'interno di un'università, oggetti di una città smart ecc...

Tra di loro, gli oggetti si scambiano informazioni, dunque avviano delle comunicazioni più o meno frequenti. Durante questi normali contatti, c'è la possibilità che un dispositivo decida di testarne un altro, avviando quindi una comunicazione di test. Tale struttura è importante per garantire l'autonomia del mondo IoT descritta precedentemente: non vogliamo che esista un amministratore che avvia tali test, ma devono essere i nodi stessi.

Iniziamo a capire che un sistema in cui le informazioni scambiate sono fondamentali per valutare l'operato dei dispositivi e garantire un certo livello di protezione, necessita a sua volta solidità e sicurezza. E' qui che entra in gioco il concetto di blockchain, che in questo approccio assume un ruolo determinante.

Nella visione del paper i nodi dell'IoT si scambiano informazioni affidandosi ad una blockchain per tenere traccia di tali comunicazioni, al fine di avere uno scenario trasparente in cui si può verificare il comportamento di ogni oggetto.





La soluzione proposta tiene conto delle limitazioni delle blockchain in ambito IoT, infatti la mole di comunicazioni (anche non di test) è enorme, e sarebbe impensabile salvare una così grande quantità di dati che aumenta costantemente. Inoltre la frequenza di contatti congestionerebbe il network rendendo il lavoro dei miners troppo pesante e riversandosi in un impatto ambientale non trascurabile. Per questo motivo l'approccio prevede due diversi tier:

- Una blockchain locale, di tipo lightweight [2], pensata apposta per comunicazioni IoT. Il meccanismo di mining è differente da una blockchain normale affinché il network non venga congestionato dalla mole di transazioni. Ne esistono diverse, come ad esempio *IOTA*[8]. Ogni community è provvista del livello locale, in cui vengono salvati tutti i dati di tutte le comunicazioni che avvengono.
- Una blockchain globale, che ha una visione generale di ogni community. Su questo livello vengono salvate aggregazioni di dati ottenuti durante le varie comunicazioni locali. In questo caso è possibile usare blockchain tipo *Ethereum* o *Hyperledger*.

## 3.2 Trust e Reputation

Capiamo ora come è possibile monitorare il comportamento degli oggetti.

Abbiamo introdotto il concetto di comunicazione di test, infatti è possibile che un oggetto, invece di avviare una comunicazione normale con un altro oggetto, decida di iniziare un test per valutarne il livello di **Trust**.

A livello operativo, un test funziona in questo modo: (i) vengono chieste informazioni

al dispositivo da testare (ii) vengono chieste informazioni alla community (iii) vengono confrontati i dati tornati dall'oggetto testato con la media dei dati della community e si calcola un valore in funzione della "distanza".

Come risultato si ottiene un valore di Trust relativo al test di un nodo  $n_1$  su  $n_2$ , salvato sulla blockchain locale della relativa community. E' intuibile, quindi, che col tempo il numero di valori di Trust calcolati per un nodo è molto grande.

Per avere una visione corretta del comportamento di un oggetto, bisognerebbe dunque leggere ogni singolo valore calcolato, ma questo è insostenibile sia dal punto di vista computazionale che di tempo. E' per questo che sulla blockchain globale vengono salvati i cosiddetti valori di **Reputation** per ogni nodo, calcolati come aggregazione dei valori di Trust. Queste aggregazioni possono essere eseguite (i) allo scadere di una time window, (ii) dopo un certo numero di test.

I calcoli appena descritti non possono essere eseguiti dai dispositivi in prima persona, infatti i già citati limiti computazionali, di memoria e di batteria non ne permetterebbero la fattibilità.

Questo problema è facilmente risolvibile attraverso gli **smart contract** delle blockchain. Il concetto di smart contract è stato introdotto in un secondo momento, quando si è sentita la necessità di avere un meccanismo per garantire uno scambio di beni o servizi (diversi dalla criptovaluta) attraverso la blockchain. Gli smart contract, letteralmente contratti intelligenti, sono un'incorporazione di clausole contrattuali codificate in linguaggio informatico, in software o protocolli informatici, che vengono utilizzati per la conclusione di rapporti di natura contrattuale conferendo autonoma esecuzione ai termini programmati al verificarsi di certe condizioni definite ex ante.

E' quindi possibile immaginare uno smart contract come un software che viene eseguito su una grossa "virtual machine" della blockchain che si fonda sulla potenza computazionale offerta dai miners.

Esisterà dunque, secondo il modello proposto, uno smart-contract che ha il compito di svolgere il calcolo e di gestire dei livelli di Trust e Reputation di ogni nodo.

## Capitolo 4

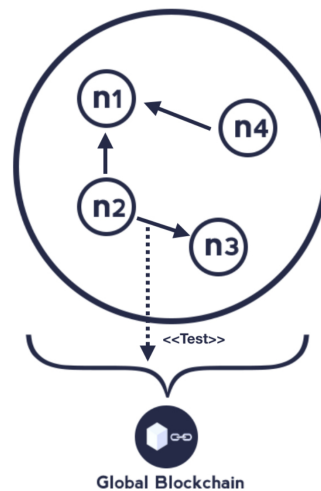
# Implementazione

L'obiettivo di questa tesi è lo sviluppo di un pacchetto software che implementa l'approccio descritto nel paper, utilizzando tecnologie disponibili per rendere la soluzione fruibile. Con pacchetto software si intende un **Client** da installare sugli oggetti, e uno **Smart-Contract**.

### 4.1 Modello semplificato

Al fine di adattare il progetto al lavoro di una tesi di laurea triennale, e considerati i soli tre mesi di tempo per l'implementazione, è stato necessario semplificare il modello descritto nel paper.

- La semplificazione più importante è quella relativa al doppio tier di blockchain: si è optato per una sola blockchain globale su cui salvare i dati di Trust e Reputation; infatti l'implementazione di un secondo livello "locale" con una blockchain come IOTA avrebbe richiesto un lavoro a se, che non avrebbe permesso per questioni di tempo una cura sufficiente dei dettagli.
- Una seconda semplificazione è stata quella di ipotizzare di avere le community già organizzate, concentrando il lavoro sulla gestione di un singolo gruppo di dispositivi. L'operazione di service discovery, ovvero l'identificazione dei servizi offerti dagli oggetti e quindi la formazione delle community è già ampiamente descritta in molti paper di ricerca scientifica.
- Come conseguenza dell'eliminazione del tier locale c'è l'impossibilità di salvare i dati delle comunicazioni non di test su blockchain, quindi le comunicazioni "classiche" avvengono tramite socket TCP/IP.



Volendo riassumere l'approccio descritto nel paper semplificato nel modo appena descritto, possiamo dire che c'è una sola community di dispositivi smart che comunicano tra di loro e, al verificarsi di una condizione particolare che vedremo dopo, possono decidere di avviare dei test per valutare i livelli di Trust e Reputation del gruppo.

## 4.2 Requisiti Client e Smart Contract

Vediamo ora quali sono i requisiti del Client e dello Smart Contract

### 4.2.1 Client

Vogliamo affidare ai dispositivi la minore responsabilità possibile.

E' necessario che essi siano in grado di:

- Connettersi alla blockchain
- Generare transazioni su blockchain
- Utilizzare le funzioni dello Smart-Contract
- Fornire i servizi una volta richiesti

Inoltre ogni oggetto è caratterizzato da alcune informazioni che lo contraddistinguono: **servizi offerti**, **IP** per la comunicazione TCP/IP, **caratteristiche fisiche** come ad esempio la posizione fisica in cui esso è collocato e infine **address su blockchain**.

La posizione fisica è un dato che servirà per eseguire un'operazione che successivamente vedremo in dettaglio (*pruning*); anche in questo caso è stata eseguita una semplificazione ipotizzando che ogni dispositivo sia "collocato" lungo un asse X.



Volendo anticipare ciò che vedremo successivamente, l'operazione di pruning ci permette di scegliere un sottoinsieme di oggetti della community a cui chiedere informazioni durante un test. Grazie alla semplificazione della posizione come dato scalare, sarà possibile fissare un intorno in prossimità dell'oggetto testato.

### 4.2.2 Smart Contract

Come detto precedentemente, lo Smart Contract ha un ruolo chiave all'interno del sistema, esso è infatti ciò che gestisce la visione globale del comportamento degli oggetti e ne deve calcolare gli indicatori (Trust e Reputation).

Possiamo identificare i requisiti più importanti:

- Gestione mappatura dei nodi della community
- Gestione dei valori di Trust e Reputation
- Calcolo dei valori di Trust e Reputation

Su blockchain saranno presenti due mappature fondamentali. Quindi, riprendendo quanto accennato nel precedente capitolo, durante la vita di una community alcuni nodi vengono testati e i valori di Trust salvati in una tabella organizzata come la tabella 4.1. Ogni time-window (oppure dopo tot. test) tale mappatura viene svuotata e i valori vengono aggregati.

Tabella 4.1: Mappatura risultati dei test

ID	Trustor	Trustee	Trust
01	$n_1$	$n_3$	0.87
02	$n_3$	$n_2$	0.92
03	$n_6$	$n_3$	0.85
...	...	...	...

Tabella 4.2: Mappatura dei livelli di Reputation

Node	Reputation
$n_1$	0.95
$n_2$	0.90
$n_3$	0.80
...	...

Il valore che interessa agli oggetti è la Reputation, essendo essa un'aggregazione che tiene conto della valore passato e della media dei Trust calcolati durante l'ultima time-window. Quindi, prima di avviare una comunicazione, un dispositivo legge su blockchain una mappatura organizzata come la tabella 4.2 con l'obiettivo di controllare se un nodo si sta comportando bene oppure no; in caso il valore di Reputation sia troppo basso, significa che un oggetto non fornisce dati corretti e quindi viene escluso dai futuri contatti.

### 4.3 Linguaggio e Blockchain

Per implementare la soluzione bisogna capire quali tecnologie ad oggi disponibili ha senso utilizzare.

Per affrontare questa tematica si è iniziato considerando i più diffusi sistemi operativi per oggetti smart. Una nota doverosa riguarda il grado di complessità degli oggetti di cui parliamo: come descritto precedentemente, il mondo IoT comprende una grande varietà di oggetti, ma per supportare la connessione ad una blockchain è chiaro che non si fa riferimento a sensori con pochi kB di memoria a disposizione, ad esempio. Ormai la maggior parte di oggetti smart prodotti, pur considerandone la categoria dei più semplici, hanno quasi tutti un minimo di potenza computazionale che permette la connessione ad una blockchain o la comunicazione TCP/IP.

Nel settore di sviluppo software per tali dispositivi, i linguaggi più utilizzati sono: **Java**, **C/C++** e **Python**.

I secondi perchè quasi tutti i S.O. sono scritti con tali linguaggi.

Tuttavia, dovendo scegliere un linguaggio sapendo di doverlo utilizzare per permettere la connessione ad una tecnologia moderna e in continuo sviluppo quale la blockchain, si è optato per *Python* [6].

Infatti, Python è uno dei linguaggi di programmazione più popolare; è open source, multi piattaforma e può essere utilizzato per sviluppare perssochè qualunque tipo di applicativo. Una regola non scritta cita che esiste sempre un package Python che può renderti la vita più semplice.

Per quanto riguarda la blockchain invece abbiamo già introdotto i nomi di **Ethereum** e **Hyperledger**. La scelta è ricaduta sulla prima, essendo *Ethereum* la blockchain più supportata dal punto di vista dello sviluppo di DApps (Decentralized-Applications) e con una community solida e strutturata.



Oltre al vantaggio del supporto per lo sviluppo, Ethereum è anche una blockchain che permette di partecipare al network come *light node*, ovvero da' la possibilità di essere un nodo che non salva l'intera blockchain in locale. Citando la documentazione di Ethereum, un light node:

- *"Stores the header chain and requests everything else."*
- *"Can verify the validity of the data against the state roots in the block headers."*
- *"Useful for low capacity devices, such as embedded devices or mobile phones, which can't afford to store gigabytes of blockchain data."*

Ritornando alla problematica della connessione Client-Blockchain, esiste un package Python **Web3.py** [4] che funziona da interfaccia con Ethereum.

Mentre per quanto riguarda lo sviluppo dello smart contract, il linguaggio da utilizzare è obbligatoriamente **Solidity** [3].

E' ancora un linguaggio in fase embrionale, come vedremo successivamente per compiere operazioni che con linguaggi comuni sarebbero semplici, con Solidity bisogna ricorrere ad ulteriori passaggi.

# Capitolo 5

## Codice

In questo capitolo verranno illustrati alcuni pezzi di codice del Client e dello Smart Contract, in funzione di mostrare quali soluzioni sono state adottate per risolvere i problemi riscontrati principalmente con il linguaggio Solidity.

### 5.1 Client

```
1 import json
2 import random
3 from web3 import Web3
4
5 class Client(threading.Thread):
6
7     # Everything needed to connect to the blockchain and to define the
    contract
8     endpoint = "http://127.0.0.1:7545"
9     web3 = Web3(Web3.HTTPProvider(endpoint))
10    contract_address = web3.toChecksumAddress("0
    x622abCc594a578ced6924ACA6397170B58AaA025")
11    contract_abi = json.loads(' [{"inputs": [{"internalType"... '}
12    contract = web3.eth.contract(abi=contract_abi, address=contract_address
    )
13
14    ...
15
16    # Node informations
17    position = 15
18    service = "Temperature"
19    ip = "1002"
20
21    prun_condition = 30
22    pact = 0.7
```



In questo primo snippet di codice vediamo le informazioni discusse nello scorso capitolo che caratterizzano un dispositivo. In particolare possiamo notare che per collegarsi alla blockchain si è utilizzato un endpoint associato ad un IP locale, questo perchè i test sono stati effettuati su una rete locale e non sulla blockchain reale come vedremo più avanti.

Tra i dati possiamo notare anche "*pruncondition*" e "*pact*", il primo valore lo vedremo meglio dopo, mentre il secondo possiamo pensarlo come "probabilità che questo nodo esegua un test". Nel dettaglio:

**Data:** probabilità *pact* che un nodo *tr* avvii un test su un nodo *te*

Viene generato un valore casuale *vact*

**if** *vact* < *pact* **then**

    il test viene avviato;

**else**

    il test non viene avviato;

**end**

#### Algorithm 1: Avvio test

Nell'algoritmo abbiamo fatto riferimento a *tr* e *te*; d'ora in poi quando parleremo di nodo che avvia un test lo chiameremo **Trustor** (*tr*), mentre **Trustee** (*te*) un nodo che viene testato.

```

1      ...
2      """
3      Method that generates a random number, and if it matched with the
4      probability, it starts a probingTransaction
5      """
6      def probingTransaction(self, trustee):
7          if random() <= self.pact:
8              myData = self.askInformation(int(trustee))
9              pruned_nodes = self.pruning(trustee)
10             datas = self.askInformations(pruned_nodes)
11             self.contract.functions.calculateTrust(myData, datas, trustee).
12             transact()
13             return True
14
15     """
16     Method that iterates through the mapped informations in the smart
17     contract and returns the pruned nodes' IPs
18     """
19     def pruning(self, _trustee):
20         prun_result = []
21         for i in self.community_nodes:
22             if(i != _trustee):
23                 if (abs(int(self.contract.functions.getPosition(i).call())
24 - int(self.contract.functions.getPosition(_trustee).call())) < self.
```

```

21 prun_condition) and (self.contract.functions.getService(i).call() ==
22 self.contract.functions.getService(_trustee).call()):
23     prun_result.append(i)
24
25 return prun_result

```

La funzione **probingTransaction** implementa la dinamica descritta nell'algoritmo, ovvero viene generato un numero casualmente (compreso tra 0 e 1) e confrontato con la *pact* locale. Se si soddisfa la condizione dell'*if* si inizia il test: prima si richiedono i dati al trustee, poi si interrogano i nodi della community ottenuti come risultato del pruning e infine vengono inviati i dati allo smart contract.

Il concetto importante da capire è che il trustee non può (e non deve) in nessun modo sapere di essere testato; infatti la comunicazione trustor-trustee avviene come se fosse una normale comunicazione di routine, mentre l'operazione di pruning prevede una lettura su blockchain che non genera alcuna transazione, di conseguenza è impossibile capire se tale operazione è in corso.

La funzione **pruning** è quella che serve per capire a quale sotto-insieme della community chiedere informazioni per il confronto. Operativamente ciò che succede è una lettura delle informazioni degli oggetti che si trovano su blockchain, che se soddisfano una particolare condizione (funzione delle informazioni del trustee) vengono compresi nel gruppo di nodi a cui chiedere i dati.

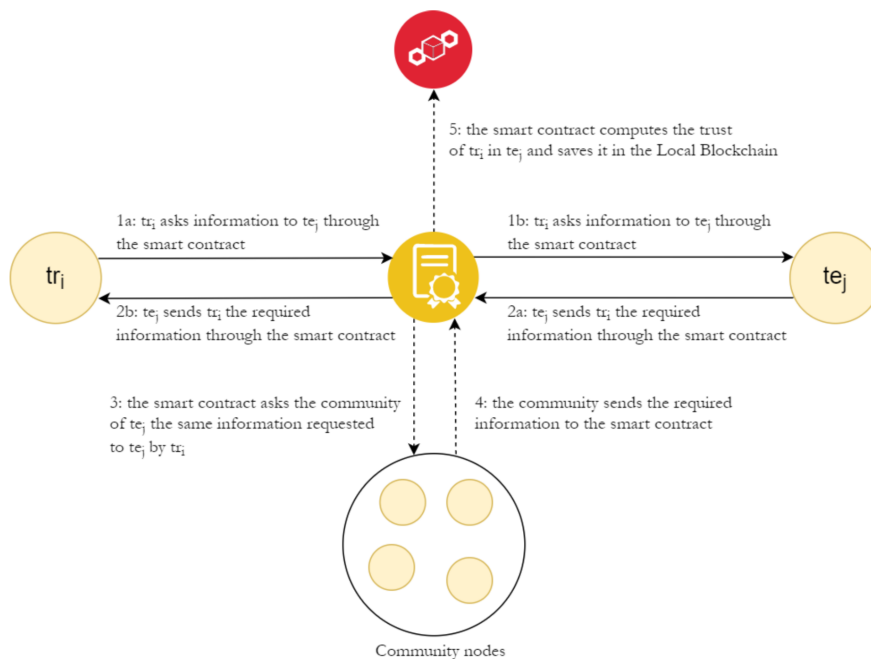


Figura 5.1: Schema approccio problematico

Una prima versione del paper illustrava un approccio in cui il pruning veniva interamente gestito dallo smart contract che, una volta elaborato il sotto-insieme, avrebbe

dovuto comunicarlo al trustor. Questa soluzione presentava un grosso problema, difatti è impossibile nascondere un'interazione di questo tipo utilizzando una blockchain. Un approccio del genere richiederebbe che il contratto conosca su quale oggetto è in corso un test. Ipotizziamo che l'eventuale funzione di pruning dello smart contract necessiti dell'IP del trustee, allora il trustor dovrebbe generare una transazione per comunicarlo e quindi sarebbe scoperto facilmente. Discorso diverso invece per un'operazione di lettura che non genera alcuna transazione.

## 5.2 Smart Contract

```
1 contract testContract {
2
3     address[] nodes;
4     mapping (string => nodeInformation) nodesInformations;
5     mapping (uint => trustResult) trusts;
6     mapping (address => string) nodesIps;
7     uint nCounter = 0;
8     uint probingThreshold = 5;
9     uint trustTolerance = 1;
10
11
12     // Struct that will contain all the informations about a connected node
13     struct nodeInformation{
14
15         uint position;
16         string service;
17
18     }
19
20     // Struct that will contain a trust calculation result
21     struct trustResult{
22
23         string trustor_ip;
24         string trustee_ip;
25         uint trust;
26
27     }
28
29     mapping (string => uint) reputations;
```

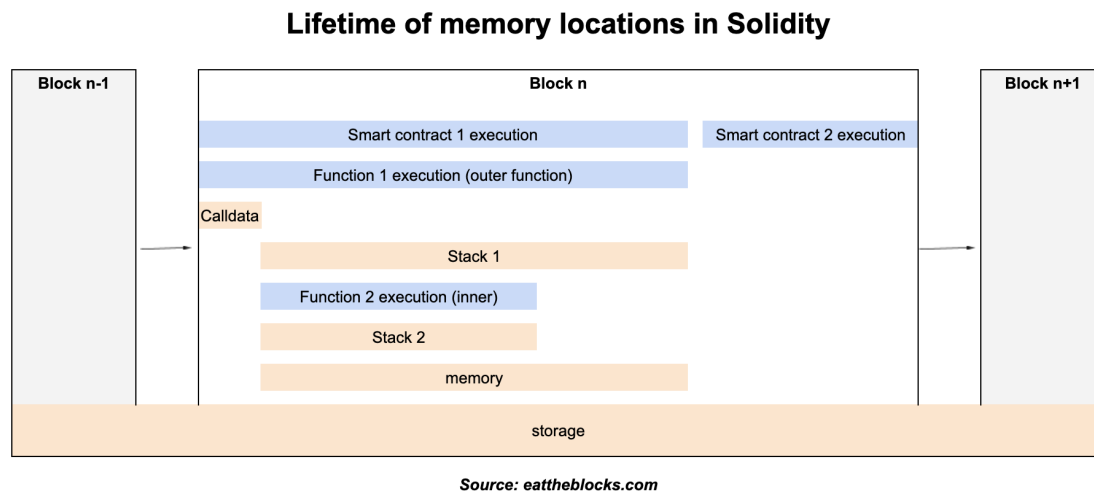
Come analizzato nel precedente capitolo, su blockchain il contratto deve avere una visione globale della community e in particolare dei valori di trust e reputation per ogni dispositivo. Notiamo subito un vettore **nodes** di tipo *address* che contiene gli indirizzi di ogni nodo del gruppo; una mappatura **nodesInformations** che associa una *string* (ovvero l'ip TCP/IP dei nodi) ad un'istanza della struct *nodeInformation*; e due mappature **trusts** e **reputations** che rappresentano le tabelle 4.1 e 4.2.

Notiamo un'ulteriore mappatura **nodesIps**, che associa un *address* ad una *string* (IP TCP/IP). Tale "dizionario" aggiuntivo ci serve perchè in una funzione del contratto ci servirà ottenere l'IP di un oggetto conoscendo il suo indirizzo su blockchain. Purtroppo Solidity è un linguaggio ancora in forte evoluzione, di conseguenza molte operazioni che con altri linguaggi più famosi sono scontate, con quest'ultimo non le sono; una su tutte è l'impossibilità di effettuare cicli sulle mappature.

```
1 // Function called from a node when it connects to the blockchain
2 function connect(uint position, string calldata service, string
   calldata ip) public{
3
4     bool flag = true;
5
6     for(uint i = 0; i < nodes.length; i++){
7
8         if(nodes[i] == msg.sender){
9             flag = false;
10            break;
11        }
12
13    }
14
15    if(flag){ // Checks that node is not already connected and the
   IP is not already in
16
17        nodes.push(msg.sender);
18        nodesInformations[ip].position = position;
19        nodesInformations[ip].service = service;
20        nodesIps[msg.sender] = ip;
21
22    }
23
24 }
```

La funzione **connect** viene usata dagli oggetti per fornire le proprie informazioni al contratto; serve durante la prima connessione alla blockchain, per permettere a un nodo di "registrarsi" come membro di una community.

Nella firma della funzione notiamo che essa accetta in input una *position*, un *service* e un *ip*, in particolare vediamo l'identificatore **calldata** nella definizione della stringa. E' importante capire cosa significa e come Solidity (e quindi la Blockchain Ethereum) gestisce la memoria a disposizione per ogni contratto.



Per ogni smart contract di Solidity esistono 4 *memory locations*:

- **storage**
- **memory**
- **calldata**
- **stack**

Hanno tutti un tempo di vita diverso e non è sempre semplice scegliere quale utilizzare. Nel grafico qua sopra possiamo vedere in che modo sono gestiti i diversi dati caratterizzati da tali identificatori. **Storage** è l'unica zona di memoria che permane ed è comune per ogni blocco e per ogni contratto, per definire un dato nello storage basta non esplicitare nulla. **Memory** invece è associato ad un contratto particolare ed è comune a più esecuzioni di funzioni. **Stack** è invece associato ad una singola esecuzione di un'unica funzione e **calldata** è lo spazio di memoria più volatile in cui vengono temporaneamente salvati i dati relativi agli argomenti passati ad una funzione.

Ritornando al discorso della fase ancora "embrionale" di Solidity, è stato necessario definire due funzioni: una per la media e una per il valore assoluto di una sottrazione.

```

1  function absSubtraction(uint x1, uint x2) private pure returns (uint) {
2      if(x1 > x2){
3          return (x1 - x2);
4      } else {
5          return (x2 - x1);
6      }
7  }
8
9

```

```

10     function avg(uint[] calldata d) private pure returns (uint) {
11         uint sum = 0;
12         for (uint i = 0; i < d.length; i++) {
13             sum += d[i];
14         }
15         return sum/d.length;
16     }

```

Analizziamo ora le funzioni per il calcolo vero e proprio di trust e reputation. Come spiegato precedentemente, la funzione di trust confronta il dato ottenuto dal trustee con i dati ottenuti dalla community, in particolare con la media di tali dati.

In particolare, possiamo scrivere che il risultato del test  $T_{ij}$  del trustor  $tr_i$  sul trustee  $te_j$  è funzione del dato  $out_j$  fornito da  $te_j$ , della media dei risultati  $\overline{out_{ij}}$  ottenuti dal sottoinsieme della community, e dalla tolleranza  $\tau$ .

$$T_{ij} = 1 - F(out_j, \overline{out_{ij}}, \tau)$$

dove

$$F(out_j, \overline{out_{ij}}, \tau) = \frac{|out_j - \overline{out_{ij}}|}{\max(out_j, \overline{out_{ij}})} \cdot (1 - \tau)$$

La funzione dello smart contract che si occupa di effettuare questo calcolo la troviamo di seguito:

```

1     // Function called from a node when wants to calculate the trust
2     function calculateTrust(uint tdata, uint[] calldata datas, string
3     calldata _trustee) public{
4
5         uint avrg = avg(datas);
6         uint max;
7
8         if(avrg > tdata){
9             max = avrg;
10        } else {
11            max = tdata;
12        }
13
14        uint trust = 100 - (((absSubtraction(tdata, avrg) * 100) / max) /
15        trustTolerance);
16
17        trustResult memory result;
18        result.trustor_ip = nodesIps[msg.sender];

```

```

17     result.trustee_ip = _trustee;
18     result.trust = trust;
19
20     trusts[nCounter] = result;
21     nCounter++;
22
23     if(nCounter == probingThreshold){
24         calculateReputations();
25         resetTrusts();
26     }
27 }

```

Ricordiamo che i risultati dei singoli test a un certo punto verranno aggregati; si può decidere di utilizzare un meccanismo che si basa su time-window (più efficace all'interno di una comunità grossa in cui i test avvengono in gran numero e in modo regolare), oppure utilizzando una soglia di transazioni di test, quest'ultimo approccio è il più semplice ed è quello implementato nel codice.

Difatti, quando il numero di test  $nCounter$  raggiunge la soglia  $probingThreshold$  il contratto avvia la funzione del calcolo delle reputazioni, aggregando i valori di trust calcolati nell'ultimo periodo. E' importante capire che in una visione del mondo reale, tali comunicazioni di controllo e di conseguenza i trust calcolati formano una mole di dati considerevole, è quindi impensabile dovere leggere così tante informazioni per comprendere il comportamento di un singolo oggetto.

Vediamo adesso le formule per il calcolo delle aggregazioni di tali valori. Definiamo  $TrS_j$  come l'insieme dei nodi che nell'ultima time-window hanno effettuato almeno un test su  $te_j$ . La reputazione  $R_j^\omega$  di  $te_j$  dopo la time-window  $\omega$  viene computata come:

$$R_j^\omega = \begin{cases} \alpha \cdot R_j^{\omega-1} + (1 - \alpha) \bar{T}_j^\omega & \text{se } TrS_j \neq \emptyset \\ R_j^{\omega-1} & \text{altrimenti} \end{cases}$$

- $R_j^{\omega-1}$  è la reputazione di  $te_j$  dopo la time-window precedente
- $\alpha$  è un valore di tolleranza usato per pesare l'importanza del valore passato
- $\bar{T}_j^\omega$  è calcolato come:

$$\bar{T}_j^\omega = \frac{\sum_{tr_i \in TrS_j} T_{ij}}{|TrS_j|}$$

Il valore di reputazione è cruciale all'interno di questo sistema di protezione, infatti se un oggetto dovesse avere una reputazione troppo bassa significa che non fornisce i dati attesi, quindi è stato attaccato oppure ha qualche guasto. Ogni oggetto, effettuando una

semplice lettura su blockchain, riesce a capire in che modo lavora un altro con cui deve comunicare. Nell'approccio completo (non semplificato) l'esclusione di nodi con poca reputazione è affidata al contratto, mentre nell'implementazione descritta in questa tesi ogni oggetto ha conoscenza degli IP degli altri nodi della community in locale.

Possiamo quindi vedere la funzione che calcola l'aggregazione:

```
1  function calculateReputations() private {
2
3      uint trCount = 0;
4      uint trustSum = 0;
5
6      for(uint i = 0; i < nodes.length; i++){
7
8          string memory te = nodesIps[nodes[i]];
9
10         for(uint j = 0; j < nCounter; j++){
11
12             if(keccak256(abi.encodePacked(trusts[j].trustee_ip)) ==
13                keccak256(abi.encodePacked(te))){
14
15                 trustSum += trusts[j].trust;
16                 trCount++;
17             }
18         }
19     }
20
21     if(trCount != 0){
22         uint reputation = trustSum / trCount;
23         reputations[te] = (reputations[te] / 4) + (reputation * 3)
24         /4;
25         trCount = 0;
26         trustSum = 0;
27     }
28 }
```

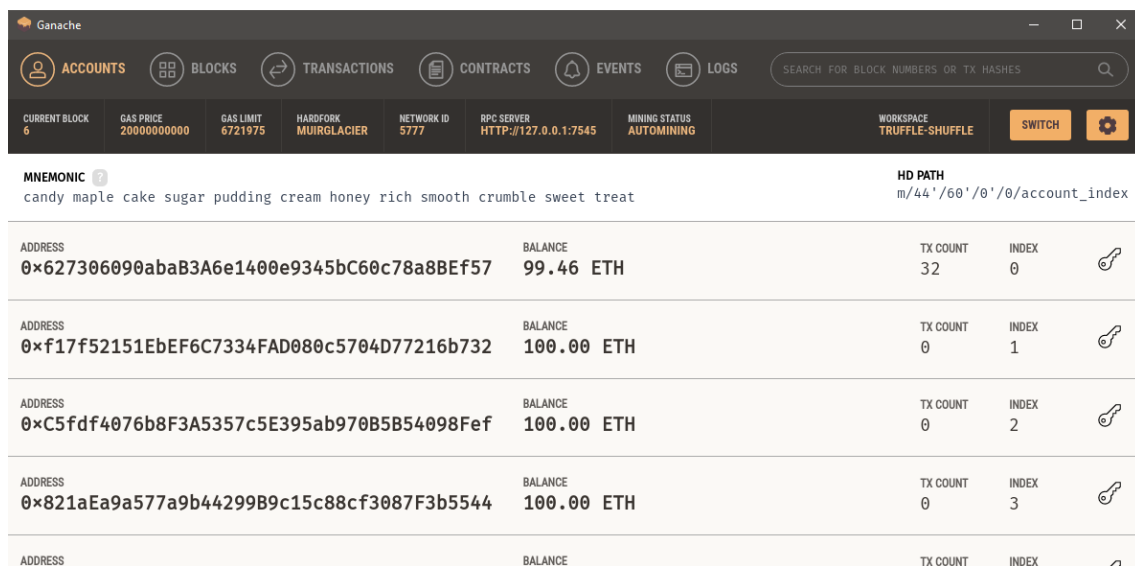
Diversamente dalle precedenti è una funzione privata, essa infatti viene usata solo dal contratto, più nello specifico dalla funzione *calculateTrust* una volta raggiunta la soglia.



## 5.3 Demo e test

Per testare l'implementazione del client e dello smart contract la necessità era avere un'istanza locale della blockchain. Sarebbe inutile e dispendioso utilizzare la vera Ethereum per diversi motivi: innanzitutto i costi che era importante evitare, e soprattutto per la scarsa scalabilità della Blockchain di questo ultimo periodo.

La soluzione è stata utilizzare **Ganache**: esso è un software open source che permette di creare una blockchain in ram. La blockchain che verrà generata sarà realizzata in Javascript, che replica comportamento e caratteristiche della famosa blockchain di Ethereum.



The screenshot shows the Ganache application window. At the top, there's a navigation bar with icons for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS. Below this is a status bar with various network parameters like CURRENT BLOCK, GAS PRICE, GAS LIMIT, HARDFORK, NETWORK ID, RPC SERVER, MINING STATUS, and WORKSPACE. The main area displays a list of accounts with their addresses, balances, transaction counts, and indices. A mnemonic phrase is also visible at the top left of the main area.

ADDRESS	BALANCE	TX COUNT	INDEX
0x627306090abaB3A6e1400e9345bC60c78a8BEf57	99.46 ETH	32	0
0xf17f52151EbEF6C7334FAD080c5704D77216b732	100.00 ETH	0	1
0xC5fdf4076b8F3A5357c5E395ab970B5B54098Fef	100.00 ETH	0	2
0x821aEa9a577a9b44299B9c15c88cf3087F3b5544	100.00 ETH	0	3

Lato client è bastato eseguire più istanze del software sviluppato in Python ed associare ogni esecuzione ad una porta TCP/IP (per permettere lo scambio di dati) e ad un address sulla blockchain locale.

## Capitolo 6

### Criticità

Come illustrato precedentemente, il lavoro svolto in questa tesi si basa su una ricerca più complessa e completa; tuttavia l'implementazione del modello completo avrebbe richiesto molto più tempo di quello a disposizione, dunque si è dovuto ricorrere a delle semplificazioni. Nel modello completo gli oggetti scrivono su blockchain qualunque dato scambiato, non solo quelli che fanno parte di una comunicazione di test. Ricordiamo che i livelli di blockchain previsti sono 2:

- Uno locale, realizzato con una blockchain **lightweight** pensata per gestire il carico di dati generati dall'IoT. Su questo livello vengono salvate tutte le informazioni scambiate all'interno di una community.
- Uno globale, realizzato con una blockchain normale, su cui si salvano i valori aggregati delle informazioni salvate sul livello locale.

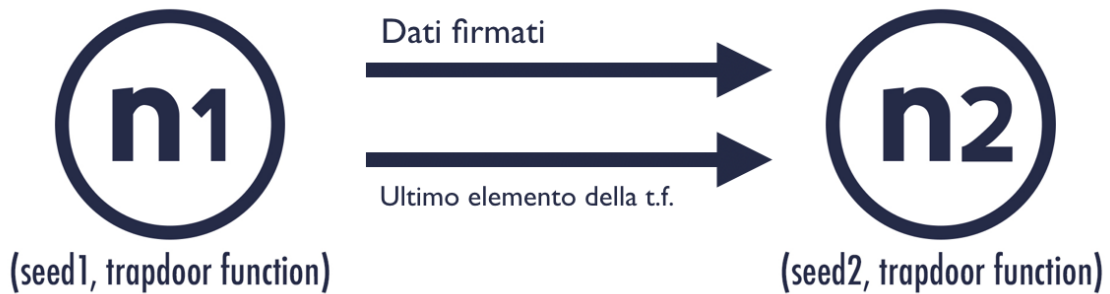
E' facilmente intuibile che implementare due livelli di questo tipo richiede più tempo e lavoro di quello a nostra disposizione. La scelta obbligata è stata quella di semplificare tale modello scegliendo di valutare le problematiche relative all'utilizzo di una blockchain particolare, in questo caso Ethereum.

Come da previsione, semplificare un sistema di protezione non è semplice e soprattutto genera, per forza di cose, un modello meno sicuro. Infatti il sistema presentato in queste pagine porta con se una importante criticità dovuta all'eliminazione del livello locale di blockchain. Decidendo di non salvare su blockchain i dati scambiati dagli oggetti in comunicazioni "di routine" ci si è trovati ad affrontare il problema della impossibilità di controllare che, una volta iniziato un test, i dati comunicati al contratto dal trustor siano effettivamente corretti. Nulla vieta ad un oggetto di fornire informazioni false col fine di abbassare la reputation di un altro oggetto. Necessitiamo di un ulteriore strumento di controllo.

## 6.1 Soluzioni

Una prima soluzione è sicuramente quella di implementare il modello completo, in questo modo è impossibile mentire sui dati scambiati. Come già ampiamente discusso precedentemente è stato però impossibile.

Una seconda soluzione è invece individuata nell'utilizzo di una **trapdoor function** unita allo scambio di dati firmati. Immaginiamo la trapdoor come un array che contiene  $n$  elementi, allora l'idea è che durante le comunicazioni un oggetto invia le informazioni richieste firmate e l'ultimo elemento non ancora inviato della trapdoor.



Vediamo nel dettaglio perchè tale modifica ci fornisce una protezione migliore. Per quanto riguarda i dati firmati, è necessario al fine di essere sicuri che essi provengano dal nodo interessato. Per comprendere l'utilità dell'elemento della trapdoor è importante vedere come quest'ultima è costruita.

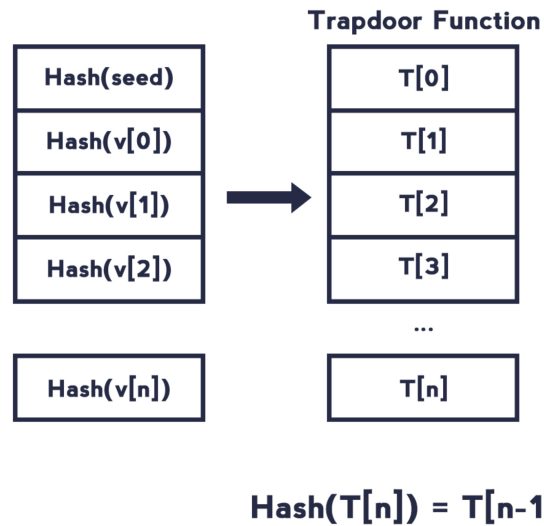
Innanzitutto notiamo che ogni oggetto è caratterizzato da un **seed**; esso è un valore numerico o alfanumerico privato, conosciuto solamente dal nodo stesso. Partendo dal seed ogni oggetto costruisce la propria trapdoor. Per prima cosa viene ipotizzato il numero di comunicazioni che un particolare oggetto farà durante la propria vita, successivamente, partendo dal seed, si genera un vettore di  $n$  elementi calcolando la codifica hash del valore precedente.

$$\boxed{\text{Hash}(\text{seed})} \mid \boxed{\text{Hash}(v[0])} \mid \boxed{\text{Hash}(v[1])} \mid \boxed{\text{Hash}(v[2])} \mid \dots \mid \boxed{\text{Hash}(v[n])}$$

Ottenuto questo vettore basta ribaltarlo, portando come primo elemento l'ultimo e viceversa. In questo modo otteniamo la trapdoor function, che per costruzione è caratterizzata da una proprietà interessante, ovvero preso l' $n_{\text{simo}}$  elemento della trapdoor function  $T[]$ :

$$\text{Hash}(T[n]) = T[n - 1]$$

Poichè abbiamo costruito la trapdoor a partire dal seed, privato e conosciuto solo dal nodo, è impossibile per un oggetto ottenere il successivo valore  $T[n + 1]$  di un altro oggetto. Tale valore è ottenibile solamente in seguito ad una comunicazione con esso.



E' semplice quindi introdurre un controllo aggiuntivo sul contratto in cui si verifica che i dati utili al calcolo del test soddisfino le condizioni della proprietà della trapdoor e che siano firmati dal nodo testato.

## Capitolo 7

# Conclusioni

In conclusione, il lavoro presentato in questa tesi aveva come scopo la creazione di una demo che trasporta il lavoro di ricerca del professore Nocera in un contesto industriale. L'implementazione ha evidenziato i vantaggi e i limiti della blockchain Ethereum:

- fortemente supportata dal punto di vista dello sviluppo di applicazioni decentralizzate, con la community di developers più ampia;
- **solidity** con ancora molti limiti ma l'integrazione con altri linguaggi (Python, JavaScript) estremamente semplice;
- la bassa scalabilità di Ethereum porta ad escluderla per implementare l'approccio completo utilizzandola da sola, fees troppo alte e tempi troppo lunghi;
- test semplici da effettuare grazie alle diverse test-net e a **Ganache**.

Si può considerare la tesi come un punto di partenza per implementare in futuro il modello completo, integrando un livello locale di blockchain lightweight; oppure implementando lo stesso modello semplificato ma utilizzando una blockchain diversa da Ethereum.

# Bibliografia

- [1] T. Ahram, A. Sargolzaei, S. Sargolzaei, J. Daniels, and B. Amaba. Blockchain technology innovations. In *2017 IEEE Technology Engineering Management Conference (TEMSCON)*, pages 137–141, 2017.
- [2] A. Dorri, S. S. Kanhere, and R. Jurdak. Towards an optimized blockchain for iot. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 173–178, 2017.
- [3] Ethereum. Solidity language. <https://docs.soliditylang.org/en/v0.8.6/>.
- [4] Ethereum. Web3.py. <https://web3py.readthedocs.io/en/stable/>.
- [5] S. Nicolazzo, A. Nocera, D. Ursino, and L. Virgili. A privacy-preserving approach to prevent feature disclosure in an iot scenario. *Future Generation Computer Systems*, 105:502–519, 2020.
- [6] P. P. Ray. A survey on visual programming languages in internet of things. *Scientific Programming*, 2017:1231430, Mar 2017.
- [7] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz. On blockchain and its integration with iot. challenges and opportunities. *Future Generation Computer Systems*, 88:173–190, 2018.
- [8] B. Shabandri and P. Maheshwari. Enhancing iot security and privacy using distributed ledgers with iota and the tangle. In *2019 6th International Conference on Signal Processing and Integrated Networks (SPIN)*, pages 1069–1075, 2019.