

Principles of Cryptocurrency Design

Appunti ed Esercizi

Francesco Pasquale

11 aprile 2024

I problemi e i protocolli che abbiamo considerato finora erano formulati per un ambiente *permissioned*, ossia quando i nodi che partecipano al protocollo sono noti a tutti e non cambiano durante l'esecuzione del protocollo.

In un ambiente *permissionless* invece i nodi che partecipano al protocollo non sono noti a priori e possono variare durante l'esecuzione del protocollo. Partecipare al protocollo significa essenzialmente scaricare un *software* ed eseguirlo, entrando in comunicazione con gli altri nodi che lo stanno eseguendo.

In un ambiente *permissionless* è necessario introdurre delle misure atte a mitigare la possibilità di sybil attacks, ossia limitare la capacità di un singolo nodo *corrotto* di generare più identità e quindi “simulare” più nodi.

Il protocollo di consenso introdotto in [4] usa la *proof-of-work* [2, 3, 1] sia per impedire che un singolo nodo corrotto possa agire come più nodi onesti, sia per fare in modo che tutti i nodi onesti concordino su quali sono i dati “corretti” da tenere in memoria.

Con il termine *proof-of-work* si intende una breve “dimostrazione”, che può essere facilmente (da un punto di vista computazionale) “verificata” che una certa quantità di lavoro computazionale è stata eseguita. Per esempio, il seguente programma ha impiegato diverse ore su un normale pc prima di dare in output una stringa di testo formata dal mio nome, cognome e un numero il cui hash crittografico SHA256 espresso in esadecimale iniziasse con una sequenza di nove zeri (ossia, che rappresentasse un numero inferiore a $2^{256-9*4}$).

```
from hashlib import sha256

nonce = 0
flag = False
while not flag:
    nonce += 1
    text = 'Francesco Pasquale ' + str(nonce)
    flag = (sha256(text.encode('utf8')).hexdigest()[:9] == '000000000')
print(text)
```

Siccome SHA256 è una funzione hash crittografica, non c'è altro modo per risolvere un tale problema (trovare un numero che concatenato con la stringa Francesco Pasquale dia un *hash* al di sotto di quella soglia) che andare per tentativi.

Una volta che tale lavoro è stato fatto, chiunque può verificarlo in una frazione di secondo eseguendo il calcolo di un unico hash

```
from hashlib import sha256

nonce = '88843838094'
text = 'Francesco Pasquale ' + nonce

print(text)
print(sha256(text.encode('utf8')).hexdigest())
```

Quindi il nonce 88843838094 è una “prova” che con altissima probabilità qualche computer da qualche parte deve aver fatto miliardi di hash per trovarne uno che soddisfacesse i requisiti. Inoltre,

se qualcuno volesse trovare un **nonce** per una stringa iniziale diversa dal mio nome e cognome, non avrebbe nessun modo di avvantaggiarsi del lavoro già svolto per per la mia stringa.

Nel caso di Bitcoin il protocollo vuole fare in modo che:

- Ogni nodo onesto mantenga una copia locale di un vettore **block** a cui vengono periodicamente aggiunte nuovi elementi, che chiamiamo *blocchi*. Il vettore di blocchi è la cosiddetta *blockchain*.
- (*Eventual Consistency*): Le copie locali **block** dei vettori dei singoli nodi onesti concordino, eccetto al più per un numero limitato di blocchi alla fine del vettore.
- (*Liveness*): Se un nodo onesto riceve in input una *transazione tx* (per il momento pensiamo a una transazione semplicemente come a un “dato”), questa prima o poi viene inserita in un blocco.
- Un nodo onesto che entri a far parte della rete in qualunque momento e chieda ad altri nodi nella rete di fornirgli il vettore **block** condiviso dagli altri, deve avere un modo di discernere quale sia il vettore “corretto”, in caso riceva informazioni discordanti da nodi diversi.

Il protocollo di consenso di Nakamoto [4] può essere sintetizzato, a grandi linee, nel seguente pseudocodice.

Algorithm 1 Nakamoto Consensus

```
1: ver = 1.0
2: block[0] = (GENESIS.header, GENESIS.data)
3: t = 0
4: data = ∅
5: while True do:
6:     prev_hash = SHA256(block[t].header)
7:     timestamp = NOW()
8:     target = UPDATE_TARGET(block)
9:     nonce = 0
10:    t = t + 1
11:    while True do:
12:        data_hash = SHA256(data)
13:        header = (ver, prev_hash, data_hash, timestamp, target, nonce)
14:        if SHA256(header) < target then
15:            block[t] = (header, data)
16:            send block[t] a tutti
17:            break
18:        else
19:            nonce = nonce + 1
20:    if ricevi tx then
21:        if VERIFY_TX(tx) then
22:            Aggiungi tx a data
23:    if ricevi blk da nodo u then
24:        if VERIFY_BLK(blk) then
25:            block[t] = blk
26:            break
27:    else
28:        chiedi a nodo u tutto il vettore block_u
29:        if WORK(block_u) > WORK(block) then
30:            block = block_u
31:            t = LEN(block)
32:            break
```

Ogni blocco è diviso in **header** e **data**. Lo **header** è formato esattamente da 80 byte: 32 byte ciascuno per **prev_hash** e **data_hash**, che sono rispettivamente lo SHA256 dello **header** del blocco precedente e lo SHA256 del **data** (opportunamente serializzati), e 4 byte ciascuno per **ver**, **timestamp**, **target**, **nonce**. Il **ver** è un numero di versione che in genere non ha nessun ruolo e **timestamp** è la data e ora attuale rappresentata in Unix time, ossia un numero intero che indica il numero di secondi trascorsi dalla mezzanotte del 1 gennaio 1970. Il **target** è un elemento cruciale, di cui parleremo fra un attimo, che viene utilizzato per fare in modo che il numero di blocchi creati si mantenga, in media, a un ritmo di 1 blocco ogni 10 minuti circa, indipendentemente da quanti siano i nodi della rete e quale sia il loro potere computazionale. Il **nonce** è un numero intero qualunque a 32 bit tale che, dati gli altri 5 campi, faccia in modo che lo SHA256 dello **header** sia minore del **target**.

Esercizio 1. Si osservi che:

1. Come nel caso dell'esempio precedente in python, *trovare* un **nonce** adeguato richiede la computazione di un numero di hash che dipende dal **target** (in media questo numero sarà $2^{256}/\text{target}$), invece *verificare* che un certo **header** abbia un hash al di sotto del **target** in esso contenuto richiede la computazione di un unico hash.
2. Per *produrre* una blockchain con t blocchi, un nodo dovrà eseguire un numero di hash che in media sarà

$$2^{256} \cdot \sum_{i=1}^t \frac{1}{\text{block}[i].\text{header.target}}$$

Questo numero è quello che ho indicato come output della funzione **WORK** alla linea 29 dello pseudocodice. Per *verificare* che in una catena con t blocchi l'hash di ogni blocco è al di sotto del **target** invece un nodo deve eseguire soltanto t hash.

3. Siccome lo **header** di ogni blocco contiene l'hash del blocco precedente **prev_hash** se un nodo volesse modificare il contenuto dell' i -esimo blocco in una catena di t blocchi, dovrebbe ricalcolare un **nonce** opportuno per ognuno dei blocchi dall' i -esimo al t -esimo.

Vediamo ora come viene calcolato il **target** in Bitcoin.

Esercizio 2. Prima di andare avanti è utile fermarsi un attimo a riflettere su come implementereste voi un requisito del genere:

1. Il numero di nuovi blocchi creati sia, in media, 1 ogni 10 minuti indipendentemente da quanti hash al secondo riescano a fare i computer che stanno eseguendo il protocollo;
2. Ogni nodo che sia a conoscenza di una catena **block[1:t]** deve poter calcolare qual è il **target** che il blocco successivo **block[t+1]** deve rispettare.

La scelta implementativa di Satoshi Nakamoto in Bitcoin è stata la seguente. Il **target** iniziale è fissato nel *genesis block*, e ogni 2016 blocchi, che corrispondono a 14 giorni se i blocchi vengono prodotti a un ritmo di esattamente 1 al minuto, si ricalcola il **target** che verrà usato per i successivi 2016 blocchi: se la differenza fra i **timestamp** contenuti nell'ultimo e nel primo blocco di un'epoca di 2016 blocchi è maggiore di 14 giorni allora il **target** va aumentato di una quantità opportuna, se la differenza è minore di 14 giorni il **target** va diminuito di una quantità opportuna. La quantità opportuna è quella che garantisce che, se il numero di hash al secondo nella prossima epoca di 2016 blocchi sarà uguale a quello dell'epoca appena conclusa, allora i 2016 blocchi della prossima epoca saranno prodotti, in media, in 14 giorni.

Esercizio 3. Derivare la formula con cui si calcola il **target** della prossima epoca in funzione del **target** dell'epoca corrente e dei due **timestamp** del primo e dell'ultimo blocco dell'epoca corrente.

Un'ultima osservazione a proposito del **target** è che si tratta di un numero intero positivo minore di 2^{256} ma viene memorizzato in soli 4 byte, quindi chiaramente non tutti i numeri sono possibili **target**. In Bitcon viene utilizzata una rappresentazione in notazione scientifica *custom*, che usa l'ultimo dei quattro byte per codificare l'esponente **exp** e i primi tre byte per il **coefficiente**.

Esercizio 4. Scrivere un programma che prenda in input 4 byte $b[0 : 3]$ e restituisca il numero intero $\text{coeff} * 2^{\text{exp}}$, dove $\text{coeff} = b[0 : 2]$ e $\text{exp} = b[3]$.

Scrivere un programma che prenda in input un numero intero positivo $n < 2^{256}$ e restituisca in output 4 byte $b[0 : 3]$ tale che $b[0 : 2] * 2^{b[3]}$ coincida con n nelle 24 cifre più significative di n .

Esercizio 5. Provare a implementare un programma minimale che produca una blockchain secondo le specifiche dello pseudocodice nell'Algoritmo 1.

Riferimenti bibliografici

- [1] Adam Back. Hashcash - A denial of service counter-measure. Whitepaper, 2002.
- [2] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual international cryptology conference*, pages 139–147. Springer, 1992.
- [3] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99) September 20–21, 1999, Leuven, Belgium*, pages 258–272. Springer, 1999.
- [4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Whitepaper, 2008.