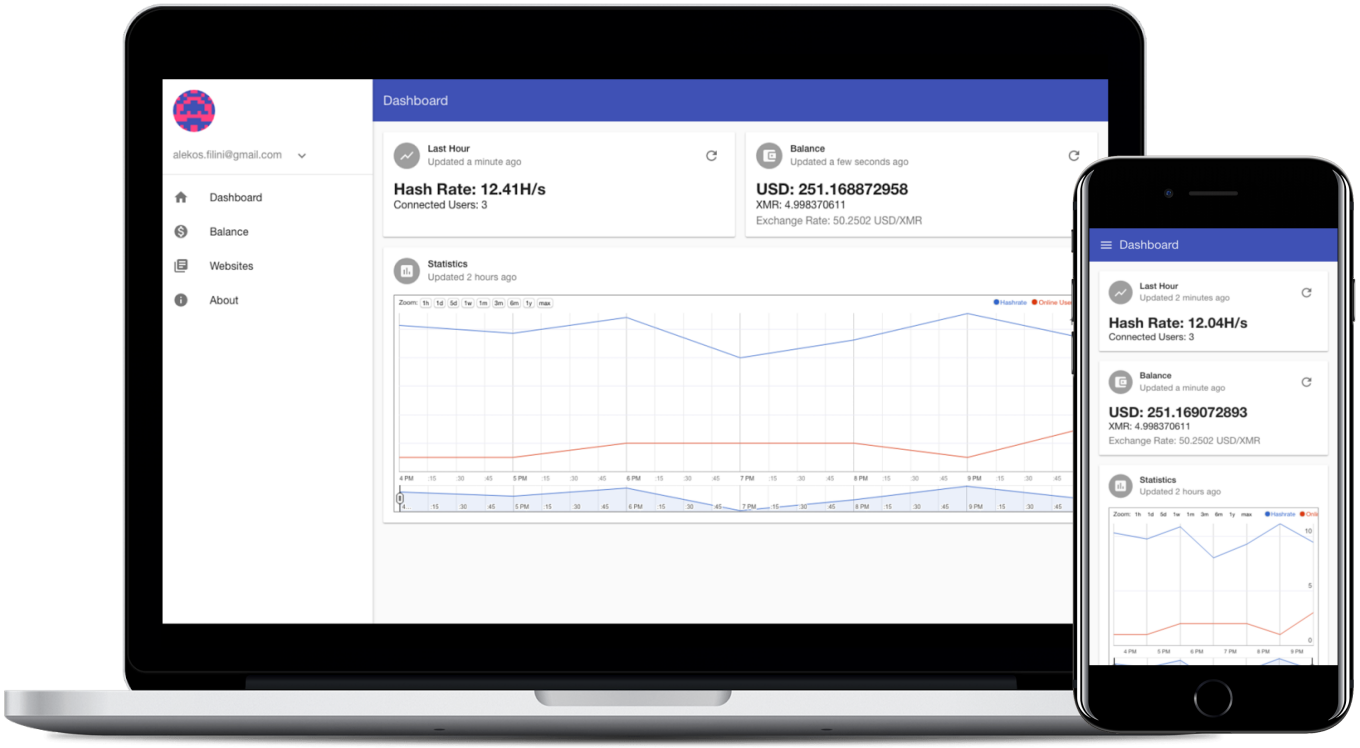
Minecash

Alekos Filini



Classe 5ª AI

IIS Benedetto Castelli

Anno Scolastico 2016/2017

Introduzione

Minecash è una piattaforma online che ha l'obiettivo di offrire un servizio di monetizzazione alternativo ai siti web che attualmente utilizzano i classici banner pubblicitari;
Viene applicato un meccanismo rivoluzionario, che consente ai siti affiliati di generare profitti senza dedicare aree delle proprie pagine web alla pubblicità.

La piattaforma sfrutta una parte della potenza di calcolo dei dispositivi che visualizzano una pagina web per creare piccole quantità di criptovaluta, che viene accumulata nell'account del gestore del sito e può essere ritirata in ogni momento.

L'idea alla base di questo progetto nasce proprio dalla mia curiosità per le criptovalute, perché penso siano una risorsa molto sottovalutata al giorno d'oggi, ma che potrà avere interessanti sviluppi per il futuro.

La prima parte di questo documento descrive in modo abbastanza approfondito, ma non eccessivamente tecnico, la moneta Bitcoin, una delle prime criptovalute create e la più famosa attualmente.
Nonostante questo progetto non utilizzi in alcun modo i Bitcoin, si è scelto di descrivere comunque questa moneta perché implementa in modo molto semplice tutti i concetti fondamentali alla base di tutte le altre criptovalute moderne e rappresenta quindi un esempio perfetto per spiegare cosa siano e come funzionino.
Altre monete, come ad esempio i Monero che vengono utilizzati in questo progetto, sono sostanzialmente "espansioni" dei Bitcoin perché riprendono da esso tutti i concetti base e allo stesso tempo implementano sistemi di sicurezza aggiuntivi, diventando così molto più lunghe e complesse da descrivere.

Successivamente vengono descritte le parti che compongono questo sistema e le soluzioni tecniche adottate per produrre una piattaforma efficiente e scalabile.

Indice

Bitcoin

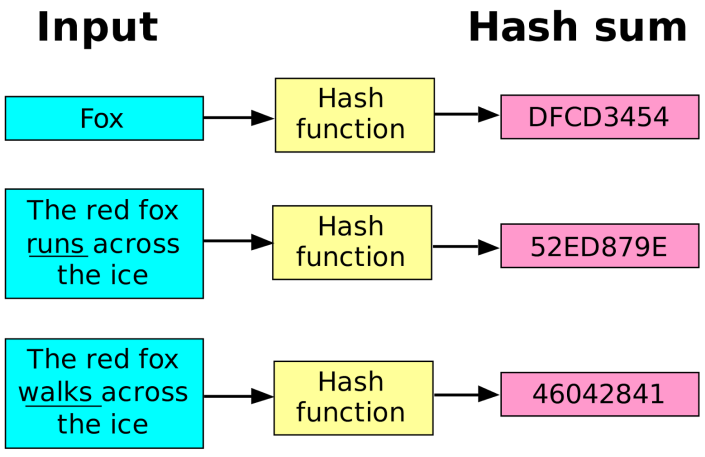
I Bitcoin sono una delle prime implementazioni del concetto di criptovaluta, ovvero una moneta elettronica gestita in modo completamente distribuito (senza entità centrale) che utilizza in modo importante la crittografia per garantire la sicurezza dell'intero sistema.

Attualmente i Bitcoin sono la criptovaluta con capitalizzazione del mercato più alta (vicina ai 50 miliardi di dollari americani al momento della scrittura), nonché la più popolare.

Per comprendere il funzionamento alla base dei Bitcoin, è necessario capire cosa siano le funzioni di hashing e il concetto e il funzionamento dei sistemi e degli algoritmi di crittografia, in particolare quelli *asimmetrici*, anche detti *a chiave pubblica.*

Hashing

In informatica e generalmente in matematica, una funzione di hashing è una funzione che, ricevuti in ingresso dati di lunghezza arbitraria, produce un risultato di lunghezza fissa che "riassume" il contenuto dei dati originali.



Essendoci una perdita di informazione, queste funzioni sono suriettive.

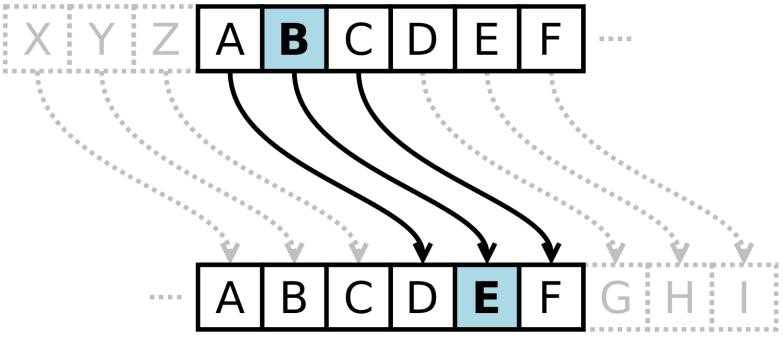
Funzioni di questo tipo sono progettate per produrre un output apparentemente casuale, che cambia radicalmente ad ogni minima modifica nei dati in ingresso e non permette di risalire ai dati originali conoscendo il loro *hash* (se non procedendo per tentativi).

Per questo motivo ogni qualvolta si rende necessario generare un "riassunto" di altre informazioni, che cambia ad ogni minima modifica delle informazioni originali si ricorre a funzioni di hashing.

Crittografia Asimmetrica

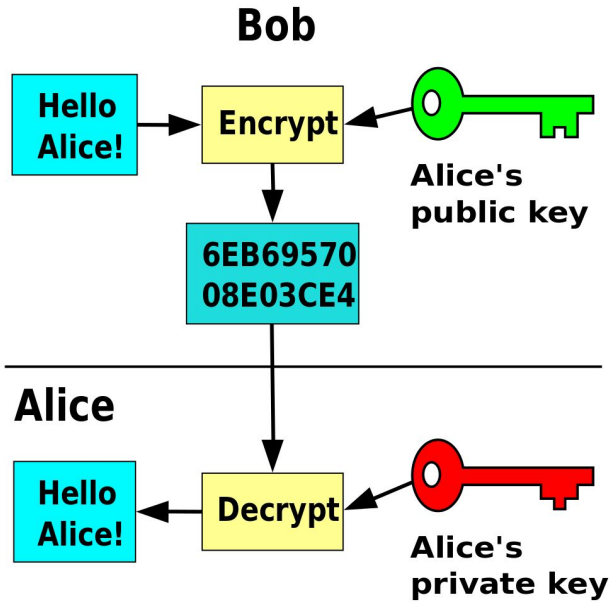
La crittografia è la scienza che studia i metodi per “offuscare” messaggi privati e permetterne la trasmissione attraverso mezzi di comunicazione non sicuri.

I primi esempi di crittografia risalgono ai tempi dei Romani, con il famoso *Cifrario di Cesare*, un codice che consiste sostanzialmente nel sostituire ogni lettera con quella che la precede di un certo numero di “posizioni” nell’ordine alfabetico. La distanza tra la lettera del messaggio *in chiaro*, e quella del messaggio cifrato è di fatto la *chiave* di questo algoritmo: chiunque sia in possesso della *chiave*, di questo numero, può decifrare i messaggi nascosti o addirittura crearne di falsi, perfettamente uguali a quelli originali.



Sistemi di cifratura simili a questo sono detti *simmetrici*, perché in modo “simmetrico” viene utilizzata la stessa chiave per cifrare e decifrare un messaggio. Per molte applicazioni questi metodi sono assolutamente validi e ne esistono di molto avanzati (AES ad esempio), ma in alcuni casi è necessario un diverso approccio alla crittografia, con metodi che abbiano altre caratteristiche oltre al semplice offuscamento di un messaggio.

I metodi di cifratura a chiave asimmetrica aggiungono funzionalità interessanti ai più semplici meccanismi simmetrici: invece che utilizzare una singola chiave per la cifratura e decifratura dei messaggi ne vengono utilizzate due diverse, strettamente legate dal punto di vista matematico.



Una delle due chiavi viene solitamente detta “pubblica” perché viene distribuita pubblicamente e viene utilizzata per cifrare messaggi diretti ad un destinatario. I messaggi cifrati non sono più decifrabili utilizzando la stessa chiave pubblica; diventa necessaria la chiave “privata”, che viene appunto mantenuta segreta dal destinatario e che viene utilizzata per decifrare messaggi diretti ad esso.

Questi sistemi di cifratura, permettono inoltre di *firmare* messaggi: al momento della trasmissione di un messaggio può essere allegato ad esso un altro piccolo

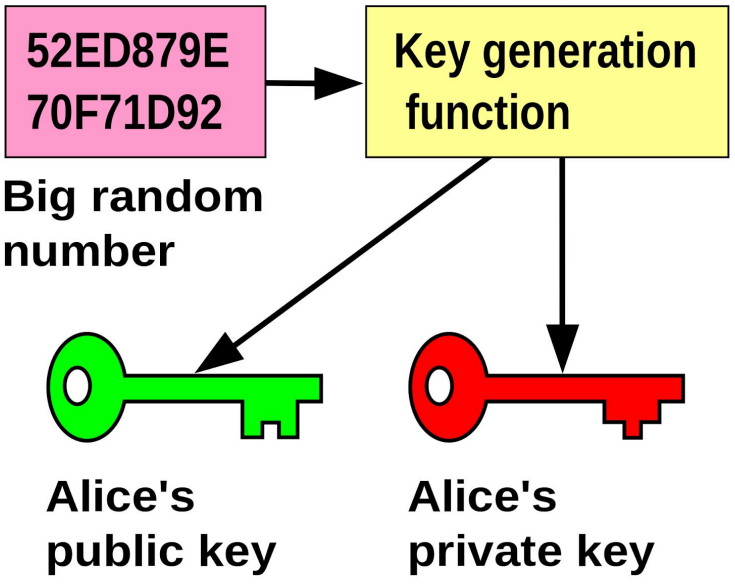
documento che contiene l'*hash* dei dati cifrato con la chiave privata del mittente. Chiunque può quindi verificare l'autenticità del messaggio utilizzando la chiave pubblica del mittente e l'integrità dei dati trasmessi (modificando il messaggio originale si modifica il suo hash).

Portafoglio

Un *wallet* Bitcoin è di fatto un *key-pair*, ovvero una coppia di chiavi pubblica e privata che permettono ad un utente di ricevere ed inviare denaro.

La chiave pubblica di un portafoglio è il suo *indirizzo*, che viene distribuito pubblicamente per ricevere denaro (come se fosse un IBAN nel sistema bancario che tutti conosciamo).
La chiave privata viene invece mantenuta segreta dall'utente, ed è necessaria per inviare denaro, perché il sistema richiede che tutte le transazioni vengano *firmate* dal mittente. Questo garantisce che solo chi possiede la chiave privata associata ad un portafoglio possa effettivamente spendere il denaro che contiene.

Essendo la moneta completamente distribuita, quindi senza un sistema di aggregazione delle informazioni centralizzato, per creare un *wallet* e scambiare denaro è sufficiente generare una chiave privata con il proprio computer. Non è necessario comunicare a nessuno la "generazione" del portafoglio, perché nel sistema Bitcoin tutti i possibili portafogli esistono dal momento della generazione della moneta.



Per comprendere meglio questo concetto, possiamo immaginare che, al momento della creazione della moneta, vengano create una serie di casseforti chiuse e posizionate in un luogo accessibile a chiunque. Quello che si chiede ora agli utenti è di costruire fisicamente una chiave in modo casuale e aprire la cassaforte con la serratura "compatibile".
Non è necessario comunicare a nessun altro che è stata trovata la chiave, semplicemente da quel momento diventa possibile prelevare denaro dalla cassaforte per spostarlo in quelle di altri utenti.

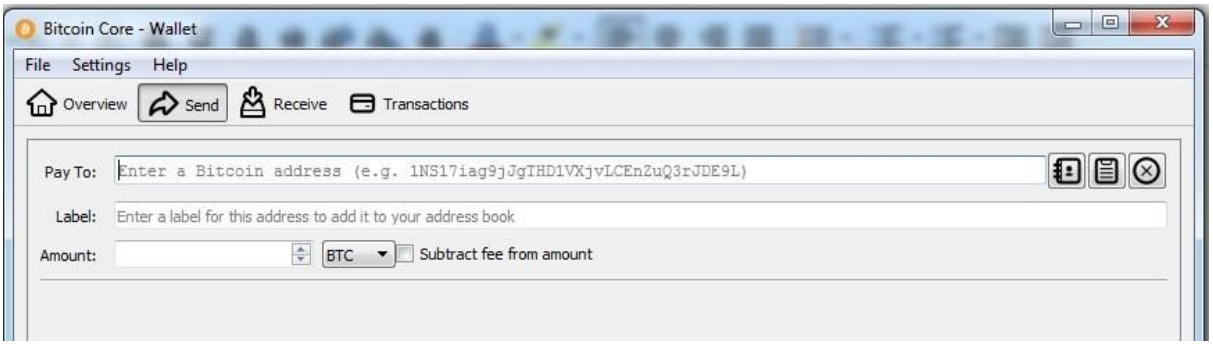
Ovviamente non conoscendo i dettagli della serratura è impossibile creare una chiave per una specifica cassaforte che sappiamo contiene denaro, esattamente come nella crittografia asimmetrica che protegge il sistema Bitcoin non è possibile generare una specifica chiave privata conoscendo la chiave pubblica (l'indirizzo) del portafoglio.

Questo meccanismo può a prima vista essere considerato pericoloso: cosa succederebbe se casualmente venisse generata una chiave corrispondente a quella di un'altra persona? La risposta è: si avrebbe pieno controllo del suo denaro e il sistema non avrebbe modo di distinguere una persona dall'altra, perché entrambe sono in possesso della stessa chiave privata.

Fortunatamente i numeri in gioco sono così grandi che è *molto* improbabile che ciò accada; esistono 2^{160} possibili indirizzi, quindi:

$$1461501637330902918203684832716283019655932542976$$

Per dare un'idea della grandezza di questo numero, possiamo paragonarlo al numero di granelli di sabbia presenti sulla terra, che è circa $7.5 \cdot 10^{18}$. Se ora immaginassimo che ogni singolo granello di sabbia fosse in realtà un intero pianeta che contiene lo stesso numero di granelli di sabbia, il loro numero totale sarebbe comunque molto più piccolo del numero di indirizzi Bitcoin possibili.



Allo stesso tempo, una conseguenza di questa politica di gestione dei portafogli è che una transazione inviata ad un indirizzo sbagliato, magari a causa di un errore di battitura, andrà sempre a buon fine, perché la rete non può sapere quali portafogli siano "attivi" e quali no, causando una perdita del denaro: come detto in precedenza, è di fatto impossibile riuscire a generare una specifica chiave privata per recuperare il denaro, perché il tempo di computazione su un computer di oggi sarebbe lunghissimo.

Per questo motivo, nonostante la moneta non possa in alcun modo essere distrutta, ma semplicemente scambiata tra portafogli, si tiene conto del fatto che un certo numero di Bitcoin verranno progressivamente persi, a causa di errori di battitura o anche per la perdita del file che contiene la chiave privata.

Blockchain

Non essendoci un'entità centrale che contiene lo storico di tutte le transazioni eseguite, è stato creato un sistema che permetta di mantenere una copia di questa lista su ognuno dei nodi della rete, che si scambiano informazioni in modo *Peer-to-Peer.*

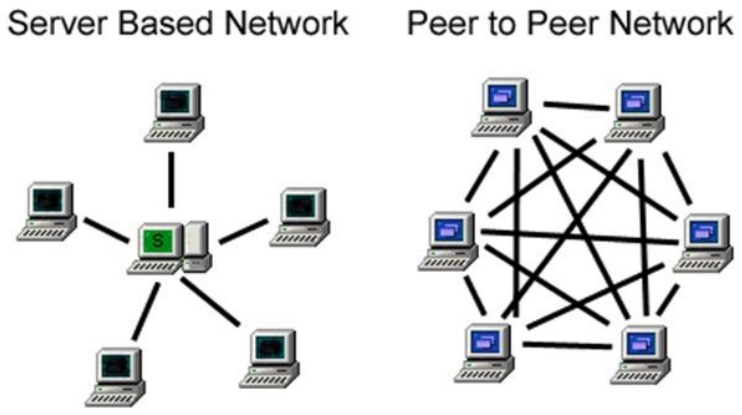
Per poter calcolare la quantità di moneta attualmente a disposizione di un dato portafoglio, è sufficiente sommare tutto il denaro contenuto nelle transazioni in ingresso e sottrarre quello uscito.

Al momento della creazione del portafoglio Bitcoin, il software provvede quindi a scaricare tutte le transazioni eseguite, da tutti gli utenti, per poter compiere questo calcolo quando necessario.

Alla creazione di una nuova transazione, essa viene comunicata ai nodi ai quali si è attualmente connessi, che provvederanno poi a propagarla a cascata ad altri nodi della rete.

È fondamentale però poter ricostruire l'ordine di invio delle transazioni, per decidere quali accettare o meno in caso il portafoglio non abbia fondi a sufficienza per coprire entrambe: ad esempio, nel sistema bancario standard, se due persone si presentassero per incassare un assegno da un conto che ha fondi a sufficienza per coprire solo uno di essi, solo il primo dei due a presentarsi riceverebbe i soldi.
Allo stesso modo si vuole che, se un utente inviasse due transazioni avendo però abbastanza denaro per coprire solo una di esse, la prima in ordine cronologico venga evasa mentre la seconda venga rifiutata.

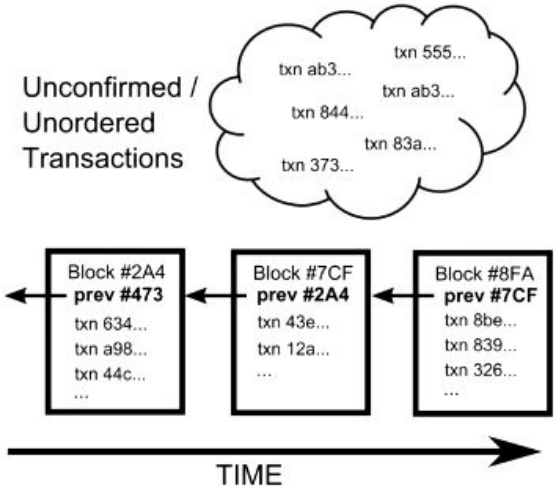
Diventa però difficile stabilire il momento di invio di una transazione, per due semplici motivi: il sistema *P2P* introduce un ritardo non indifferente nei tempi di propagazione delle informazioni riguardanti la transazione all'interno della rete. Allo stesso tempo non si può chiedere all'utente di specificare al momento dell'invio della transazione l'orario, perché essa sarebbe facilmente falsificabile.



Bisogna ricordare che in un sistema di questo genere, dove non esiste un'entità fidata centrale come una banca, è necessario che tutte le operazioni siano intrinsecamente sicure e trasparenti per tutti per non dover mai prendere decisioni (accettare o meno una transazione) in base alla *fiducia*.

Per risolvere questo problema è stato ideato un sistema che permette di raggruppare, ogni 10 minuti circa, una serie di transazioni recenti non verificate ed unirle in un contenitore, detto comunemente "blocco", in modo permanente.
Il contenuto del *blocco* è visibile a tutti, ma non più modificabile, perché insieme ad esso viene propagato il suo hash.

Oltre alle informazioni riguardo le transazioni, viene aggiunto un dato molto importante: un riferimento al blocco (il suo hash) che lo precede in termini temporali, formando così una *catena* immaginaria.
Ovviamente volendo modificare un preciso blocco della catena diventa necessario modificare tutti i successivi "a cascata", proprio perché il loro hash cambia.



L'ordine delle transazioni viene quindi stabilito in base al blocco in cui vengono inserite perché esiste un ordine rigido tra di essi.

Una transazione per poter entrare a far parte di un *blocco* ed essere quindi confermata deve essere coerente sia con le transazioni dei blocchi precedenti, sia con quelle del suo stesso blocco: controllando lo storico delle transazioni si può calcolare il "bilancio" di un portafoglio e decidere quindi se la nuova transazione sia accettabile o meno.
Se due transazioni che cercano di spendere lo stesso denaro entrassero nel *pool* delle transazioni non confermate allo stesso tempo solo una delle due, casualmente, verrebbe confermata, per mantenere la coerenza all'interno del blocco, mentre l'altra non verrà poi più accettata nei blocchi successivi per mantenere la coerenza con i blocchi precedenti.

La *blockchain* è quindi la parte del sistema che archivia in modo permanente le transazioni e allo stesso tempo ne definisce l'ordine cronologico.

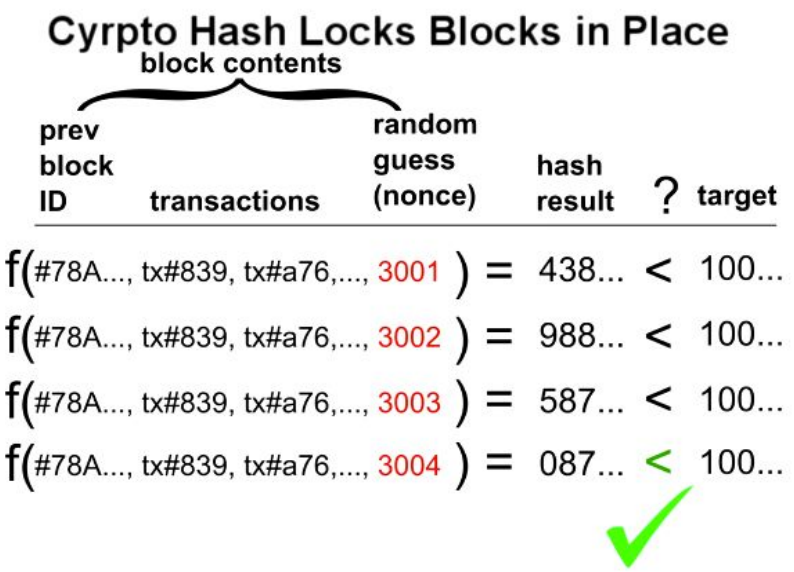
Mining

Il concetto alla base della sicurezza della blockchain è che dev'essere molto difficile generare un blocco, per impedire che una singola persona possa manipolare in solitaria le transazioni modificando un blocco in mezzo alla catena e creando velocemente tutti i successivi.

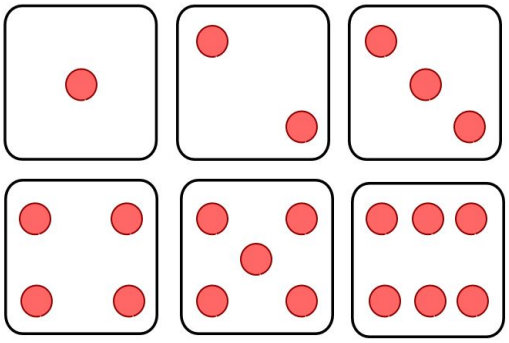
Dato che è necessario studiare un sistema distribuito per generare nuova moneta da immettere sul mercato, si è deciso di utilizzare il procedimento stesso di generazione dei blocchi, che rende sicure le transazioni, come meccanismo per sostenere una "lotteria" che genera nuova moneta al momento della creazione di un blocco.
Il risultato è che le persone che vogliono generare moneta per sé stessi mettono eseguono calcoli che rendono sicura l'intera rete.
Per specifica tra le transazioni di un blocco ne viene sempre inserita una molto particolare, che non ha un portafoglio "mittente", ma ha come destinatario il portafoglio di chi sta creando il blocco e la quantità di moneta viene stabilita dalla rete (non è quindi possibile creare un blocco e includere una transazione con una quantità enorme di moneta verso sé stessi).

Per rendere statisticamente complesso il processo di mining si è pensato di aggiungere criteri rigidi sull'hash del blocco per determinare se sia valido o meno. Per questo motivo nell'intestazione di ogni blocco viene incluso un campo di alcuni byte detto *nonce* che può essere modificato a piacere, in modo casuale, per modificare l'hash del blocco senza cambiare il suo contenuto.

A questo punto un blocco è considerato valido se il suo hash, preso in considerazione come numero esadecimale, è minore o uguale ad un determinato valore *target* che tutti i nodi della rete conoscono e provvedono ad aggiornare periodicamente.



Visto che, come detto in precedenza, le funzioni di hashing producono risultati apparentemente casuali, non è possibile risalire ad una *nonce* che crei un blocco valido e, allo stesso tempo, vista la distribuzione uniforme dei valori di output, la probabilità di *risolvere* un blocco dipende semplicemente dal rapporto tra valore target e il massimo valore possibile di un hash, nel caso dei Bitcoin $2^{256} - 1$.



Per fare un esempio, supponendo di lanciare un normale dado a 6 facce invece che calcolare un hash, due eventi statisticamente comparabili vista la distribuzione dei risultati, se il nostro valore di target fosse 3, avremo il 50% (3/6) di possibilità di ottenere un numero accettabile.

Allo stesso modo, nei Bitcoin, il valore target viene aggiornato, in modo distribuito, per rispecchiare la potenza di calcolo della rete: si è detto che questa moneta ha un *tempo di blocco* fisso, ovvero si vuole che l'intera rete *trovi* un blocco ogni circa 10 minuti.
Viene quindi stimato l'*hashrate*, ovvero il numero di hash calcolati al secondo dall'intera rete durante il *mining* dei blocchi precedenti, che è sostanzialmente il numero di tentativi fatti ogni secondo da tutti i nodi per provare a trovare un blocco valido e ricevere la ricompensa.

Questo valore è il risultato di centinaia di migliaia di dispositivi che lavorano sparsi in tutto il mondo: di conseguenza non si registrano mai oscillazioni significative, perché l'effetto di un piccolo gruppo di persone è irrilevante se comparato alla potenza di calcolo globale.

Si calcola facilmente il numero di tentativi che si vuole, in media, eseguire attraverso l'intera rete per trovare un blocco con la formula banale:

$$D = 10 \cdot 60 \cdot \text{HASHRATE}$$

Questo valore viene detto *difficoltà*, perché all'aumentare di essa diventa sempre più difficile riuscire risolvere un blocco.

Il valore target di conseguenza corrisponde a:

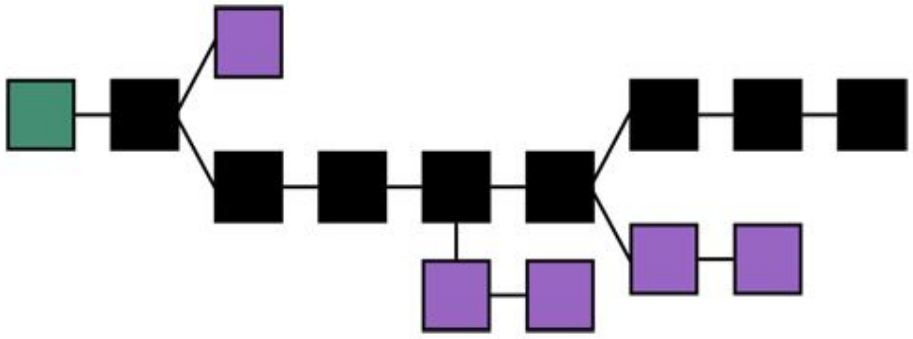
$$T = (2^{256} - 1) / D$$

Al momento della generazione di un blocco valido, esso viene immediatamente propagato all'interno della rete, che inizia quindi a lavorare al blocco successivo.

Gestione delle collisioni

Nonostante sia piuttosto raro, in alcuni casi due blocchi possono essere risolti un blocco allo stesso momento e a causa dei tempi di propagazione alcuni nodi della rete riceveranno prima uno dei due mentre altri riceveranno l'altro.

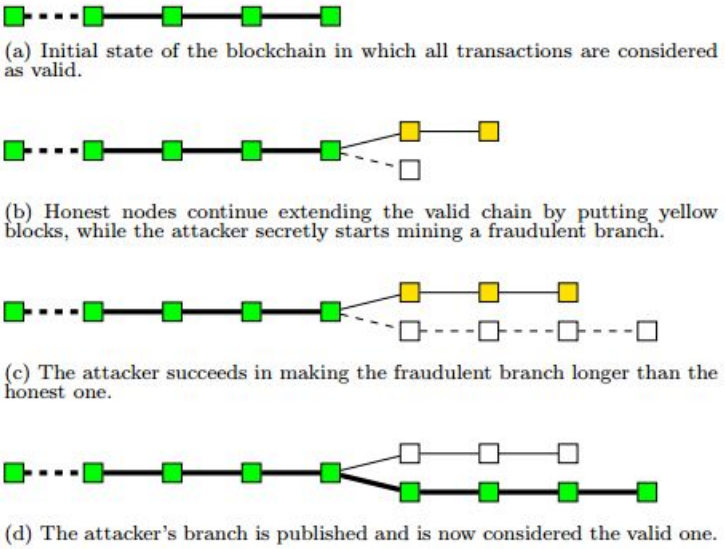
In questo caso diventa necessario raggiungere un accordo all'interno della rete, per mantenere la blockchain uguale in tutti i nodi.
Il sistema studiato è sorprendentemente semplice e allo stesso tempo molto efficace: si è pensato di lasciare che ogni nodo della rete lavori "sopra" il blocco più *alto* che conosce, ovvero quello più lontano dal primo blocco iniziale. In caso un nodo conosca più blocchi alla stessa altezza, che sono stati trovati allo stesso momento, sceglie casualmente su quale dei due lavorare.



A questo punto è sufficiente aspettare che venga trovato un nuovo blocco e che esso venga propagato nella rete; tutti i nodi che stavano lavorando su un diverso blocco ricevendone uno più alto passano ad esso e il conflitto viene così risolto.

Questo sistema funziona perché è statisticamente difficile che due blocchi vengano trovati allo stesso momento, e lo è ancora di più che questo succeda più volte consecutivamente.

La conseguenza di questo sistema è che le transazioni di fatto confermate perché inserite in un blocco che però si trova alla fine della blockchain potrebbero, in condizioni particolari, ritornare nel pool delle transazioni non confermate se il blocco che le conteneva diventa *orfano*.



(a) Initial state of the blockchain in which all transactions are considered as valid.

(b) Honest nodes continue extending the valid chain by putting yellow blocks, while the attacker secretly starts mining a fraudulent branch.

(c) The attacker succeeds in making the fraudulent branch longer than the honest one.

(d) The attacker's branch is published and is now considered the valid one.

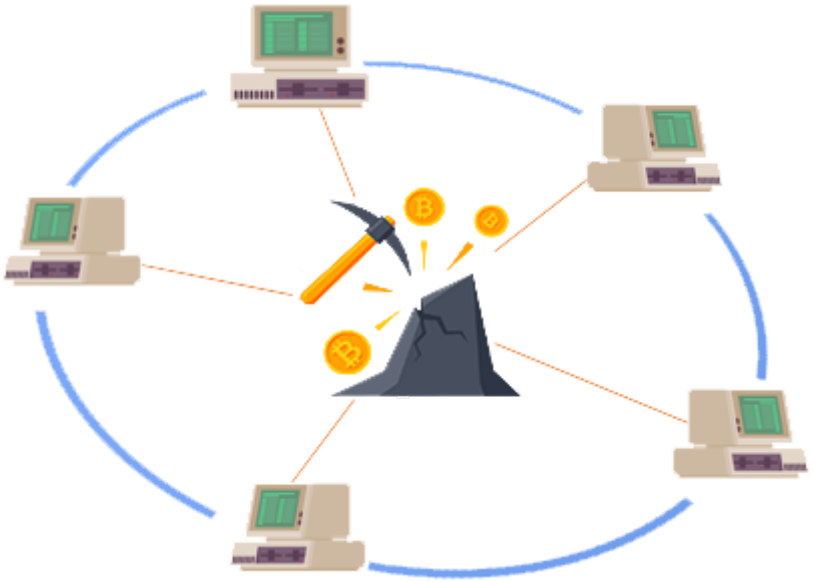
Per questo motivo è buona pratica attendere la soluzione di

diversi blocchi prima di considerare ricevuto il denaro. Più blocchi vengono risolti sopra quello che contiene la nostra transazione, più è raro che un nuovo ramo si crei e superi in lunghezza quello che noi vediamo.

In più si deve assolutamente evitare che un singolo computer o gruppo di computer possieda più del 50% della capacità di calcolo totale della rete: se ciò si verificasse, esso potrebbe in solitaria costruire rami di blockchain e manipolarla a suo piacimento.

Mining Pools

A causa dell'enorme potenza di calcolo della rete Bitcoin che porta la difficoltà a crescere, un singolo computer impiegherebbe, in media, migliaia di anni per riuscire a risolvere un blocco in solitaria.



Per questo motivo sono stati introdotti i "pool", ovvero servizi online che aggregano la potenza di calcolo di più persone e distribuiscono poi proporzionalmente la ricompensa ai computer che hanno partecipato.

Minecash

Di seguito vengono descritte le funzionalità del sistema, dedicando alcune parole ad ognuno dei moduli che compongono questo sistema.

Scelta della criptovaluta

Questo progetto sfrutta una moneta che riprende tutti i concetti descritti precedentemente nel caso dei Bitcoin, comuni a tutte le criptovalute, ma implementa allo stesso tempo alcuni cambiamenti molto interessanti: oltre a garantire un maggiore livello di privacy nascondendo di fatto il bilancio degli utenti e la quantità di moneta trasferita in ogni transazione, la caratterostoca più importante che le ha permesso di entrare a far parte di questo progetto è l'algoritmo di hashing utilizzato nel processo di mining.

Bitcoin utilizza SHA256, un algoritmo molto efficiente e soprattutto molto compatto in termini di memoria utilizzata per il suo calcolo: questo ha favorito lo sviluppo di una famiglia di dispositivi denominati *ASIC*, ovvero circuiti altamente specializzati ed efficienti che sono in grado di calcolare molti più hash al secondo, consumando allo stesso tempo meno elettricità rispetto ad un pc.
La conseguenza principale della nascita e diffusione di questi dispositivi è stata la rapida crescita della *difficoltà* nella rete Bitcoin e un brusco calo di profitti per tutti coloro che fino a quel momento avevano utilizzato computer tradizionali.
Per evitare che ciò possa accadere nuovamente sono nati diversi algoritmi di hashing pensati in modo specifico per essere sostanzialmente impossibili da calcolare con hardware dedicato, perché vanno ad attaccare i punti deboli di questi dispositivi specializzati, molto spesso la memoria, che ha costi molto elevati.

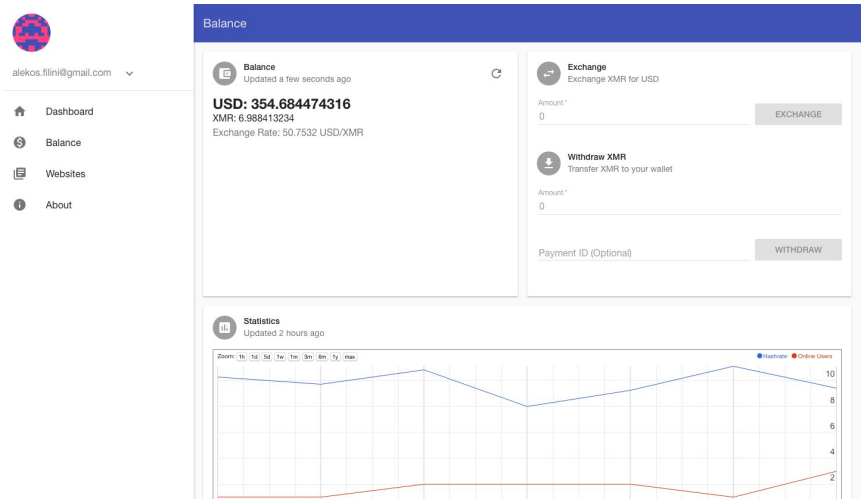
Dato che questo sistema sfrutta i browser dei visitatori per calcolare hash, si vuole utilizzare una moneta che possa essere generata in modo competitivo anche su computer, escludendo però tutte quelle monete che hanno un eccessivo memory footprint, nell'ordine dei gigabyte.
L'algoritmo perfetto per questo ambito è chiamato Cryptonight, e la moneta principale che lo implementa è denominata Monero, con simbolo XMR.

Cryptonight è un algoritmo di diversi ordini di grandezza più lento di SHA256 ed utilizza per ogni singolo hash calcolato uno *scratchpad* di 2 MiB, diventando praticamente impossibile da calcolare su ASIC, rimanendo comunque fattibile per un browser.

Funzionalità

Il sistema si interfaccia con gli utenti finali tramite un applicativo web che permette ai webmaster affiliati alla piattaforma di monitorare i profitti e gestire i siti web da loro registrati.

Monitoraggio dei profitti



Gran parte dell'applicativo web è dedicato alla raccolta e visualizzazione di statistiche che possano essere utili ai webmaster per ottimizzare i profitti migliorando le proprie pagine web.
Viene proposto all'utente un grafico che aggrega dati da tutti i siti registrati riguardo il numero totale di visitatori connessi e la potenza di calcolo complessiva, con una risoluzione oraria.

Le stesse statistiche possono essere visualizzate in real time, per ottenere dalla piattaforma un feedback più rapido ed efficace.

Gestione dei siti web

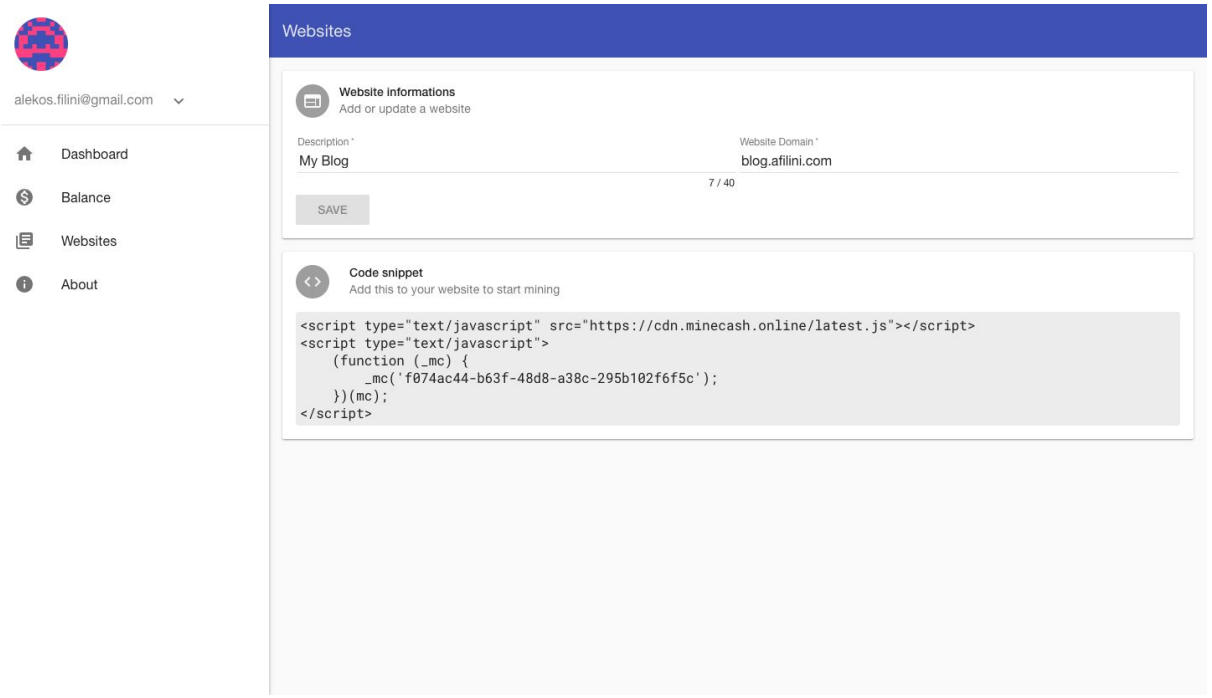
La piattaforma offre ai webmaster la possibilità di gestire da un solo account più siti web diversi, permettendo un monitoraggio più preciso dei profitti.
Al momento della creazione di un nuovo sito web viene chiesto all'utente di specificare il dominio che viene poi verificato, per evitare che questa piattaforma possa essere utilizzata per attacchi di tipo Cross Site Scripting, generando profitti grazie ad un sito vittima.

Ritiro del denaro accumulato

La moneta generata da tutti i siti web viene aggiunta al bilancio dell'account, anch'esso aggiornato in tempo reale.

L'utente, raggiunta una soglia minima, può decidere se ricevere i profitti direttamente in Monero, sotto forma di transazione verso il proprio portafoglio, o se riceverli in valuta tradizionale tramite PayPal.

Infine, una lista di tutti i ritiri passati permette di monitorare lo stato delle transazioni in attesa di essere confermate.

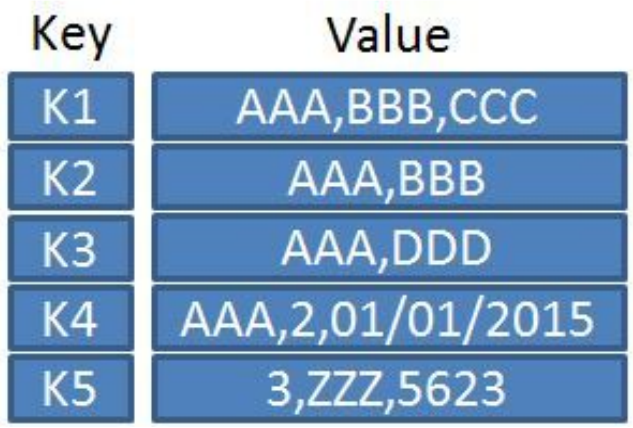


Componenti

La piattaforma è composta da diversi moduli sviluppati indipendentemente per isolarli maggiormente, che in produzione cooperano in modo scalabile per offrire un'esperienza di utilizzo veloce e funzionale all'utente finale.

Database NoSQL

Vista l'enorme quantità di dati raccolti dal sistema, principalmente utilizzati per la generazione delle statistiche, si è deciso di utilizzare un database di tipo NoSQL, che garantisce scalabilità e affidabilità superiori rispetto ad altri database relazionali più conosciuti.



In particolare si è deciso di utilizzare Couchbase, perché offre interessanti funzionalità oltre al semplice storage di documenti JSON offerto anche da altri suoi competitor, come MongoDB e CouchDB.

Couchbase implementa tramite le cosiddette *views* un engine di *Map-Reduce* altamente distribuito, basato sul motore JavaScript V8, che permette quindi di riutilizzare parti di codice con i moduli lato server, scritti in NodeJS.

Allo stesso tempo Couchbase, che include al suo interno un vero e proprio server Memcached, mantiene i documenti più utilizzati in RAM, per garantire un elevato *throughput* e una bassa latenza.

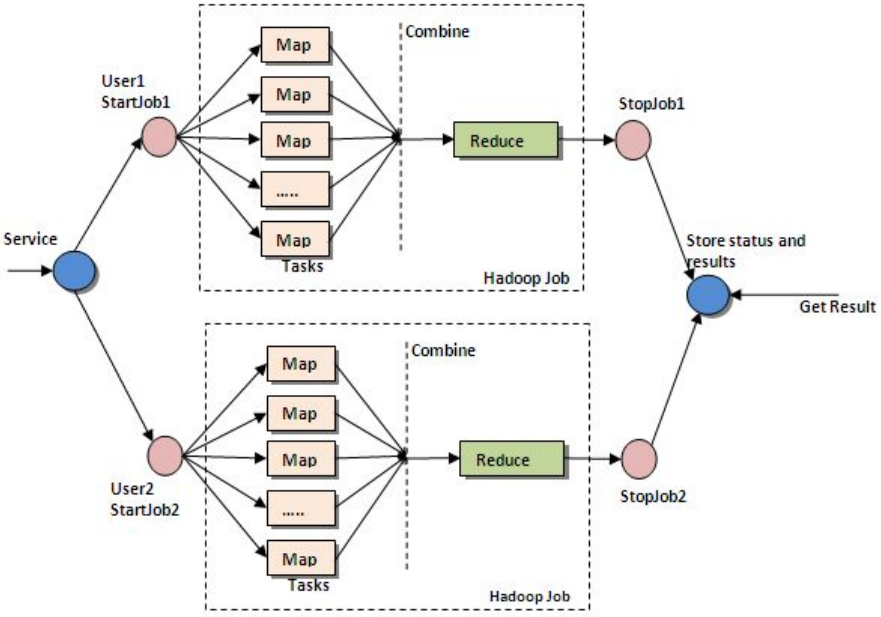
Modello Map-Reduce

Il modello di programmazione Map-Reduce nasce dalla necessità di processare enormi quantità di dati, ai quali ci si spesso riferisce con il termine *BigData*, in modo altamente distribuito, con una scalabilità quasi lineare.

Questo modello prevede la divisione dell'intero processo di elaborazione in due fasi distinte, denominate map e reduce.

La funzione *map* viene applicata una sola volta ad ogni elemento del dataset: in modo concettualmente simile ai principi dei linguaggi funzionali, il compito di questa funzione è leggere il documento ricevuto ed eventualmente produrre uno o più documenti di output che mantengano le informazioni che si vuole estrarre dai dati di input, eliminando tutto il superfluo.

La funzione di reduce viene applicata ad una serie di documenti *mappati* e il suo compito è quello di aggregare i dati grezzi e produrre un unico risultato che tenga conto di tutte le informazioni ricevute in ingresso.
In alcuni casi la funzione di reduce viene applicata più volte, quindi anche a serie di risultati di una prima reduce, fino ad arrivare ad un unico risultato che è il risultato della nostra computazione.



Questo modello prevede che la fase di *mapping* venga eseguita direttamente dal nodo del cluster che contiene il documento, scalando in modo sostanzialmente lineare anche in presenza di centinaia di computer.

La fase di *reduce* viene solitamente applicata, per la prima fase, all'interno di ogni singolo cluster, con un guadagno importante in termini di tempi di computazione, per poi essere applicata nuovamente ai risultati delle prime *reduce.*

Backend

Uno dei componenti principali del sistema è il backend che fornisce dati all'applicativo web utilizzato dagli utenti finali.
Il backend può essere immaginato come un ponte tra il database principale contenente tutti i dati raccolti e il webmaster che utilizza la piattaforma: le sue funzionalità principali sono quelle di raccolta dei dati dei siti web, aggregazione delle informazioni per la generazione delle statistiche e gestione del bilancio di ogni account.

Gestione delle sessioni

Nonostante al momento l'unico client per il backend sia l'applicativo web, si è deciso di gestire le sessioni senza utilizzare in alcun modo i cookie HTTP. Questo rende molto più semplice sviluppare in futuro applicazioni native per smartphone ed è in linea con le *best practices* REST per le API pubbliche.

Al momento dell'accesso un token univoco viene generato e inviato al client insieme ad un messaggio che conferma l'avvenuto login. Il client si occuperà poi di compilare il campo X-Authorization-Token ogni qualvolta venga eseguita una *chiamata* che necessita l'autenticazione dell'utente finale.

Webapp

La webapp è il software utilizzato per interfacciarsi con i webmaster affiliati. Esso permette di:

- Aggiungere siti web e lettura dello script da copiare nelle proprie pagine
- Monitorare le statistiche
- Creare transazioni per ritirare il denaro guadagnato
- Gestire il proprio profilo personale

La webapp è completamente *responsive*, quindi si adatta automaticamente al dispositivo che la visualizza, diventando quindi molto simile ad un'applicazione nativa se visualizzata su cellulare.

Pool

Il pool si occupa di aggregare la potenza di calcolo di più dispositivi e coordinarli al fine di distribuire il lavoro nel modo migliore possibile, tenendo in considerazione la potenza del dispositivo stesso. Pur essendo lui stesso un pool particolare, è progettato per poter comunicare con pool più grandi, apparendo sostanzialmente come un singolo dispositivo molto potente.
Questa funzionalità apre alcune possibilità interessanti: ad esempio, in caso la potenza di calcolo aggregata fosse comunque piuttosto scarsa può diventare conveniente utilizzare un pool di livello superiore per ottenere guadagni immediati. Al crescere della potenza di calcolo si può poi passare con un minimo sforzo a "lavorare" in solitaria.

Client

Il client è la parte di codice che viene inclusa nei siti web affiliati. All'inizio della sua esecuzione esso si occupa di caricare in memoria il *miner*, scegliendo la versione che garantisca le maggiori prestazioni compatibilmente con le modalità supportate dal browser del visitatore e dall'architettura utilizzata.

Contemporaneamente, il client stabilisce un canale di comunicazione bidirezionale con il *pool*, per poter ricevere del lavoro da eseguire e comunicare successivamente il risultato della computazione. Anche in questo caso viene negoziato un metodo di trasporto che garantisca una bassa latenza e allo stesso tempo sia completamente supportato dal browser.

Miner

Il miner è il componente che esegue di fatto i calcoli sul browser del client. È la parte del sistema che richiede più ottimizzazioni per massimizzare l'*hashrate* e conseguentemente i profitti.
Per questo motivo si è deciso di utilizzare la versione ufficiale scritta in C ottimizzato *compilata* con specifici tool in JavaScript e WebAssembly, piuttosto che realizzarla partendo da zero.

Tecnologie Utilizzate

Per la realizzazione di questo progetto si è deciso di utilizzare, anche per i componenti lato server, il linguaggio JavaScript, tramite NodeJS. Questo permette una maggiore efficienza nella scrittura del codice perché elimina la fase di “familiarizzazione” tipica di quando si passa da un linguaggio all’altro.

In particolare è stata scelta la versione 6.10 di NodeJS, perché supporta lo standard ECMAScript 6, che aggiunge molte funzioni interessanti per scrivere codice più leggibile.

Backend

Il backend è stato realizzato utilizzando il framework Express.js, perché gestisce internamente gran parte delle funzionalità normalmente utilizzate nella creazione di servizi REST come questo, in particolare il parsing delle rotte.
Express.js ha un’interfaccia molto semplice e leggibile, che permette anche di riutilizzare parte del codice per più chiamate.

Il codice è organizzato in diverse sottocartelle, che rappresentano le funzionalità a livello concettuale del codice che contengono. Queste sottocartelle sono:

- model: contiene i prototipi delle entità utilizzate in questo progetto
- logic: contiene tutte le funzioni che operano sulle entità, gestisce l’interazione con il database
- routes: contiene la definizione delle rotte REST
- test: contiene i test di unità
- utils: contiene funzioni utilizzate in più punti del codice, come la connessione al database

Interazione con l’esterno

Il backend, essendo il componente del progetto che prepara le informazioni da visualizzare agli utenti finali, deve in alcuni casi raccogliere informazioni dall’esterno che vengono poi integrate con quelle prodotte dalla piattaforma. In particolare diventa molto importante mantenere un valore aggiornato del cambio Monero/Dollari Americani, perché, essendo la piattaforma stessa un punto di scambio di monete è chiave mantenersi allineati con il mercato per non incorrere in gravi perdite.

In questo caso si è deciso di utilizzare le API HTTP pubbliche del sito web “Kraken”, un popolare portale di exchange di criptovalute.

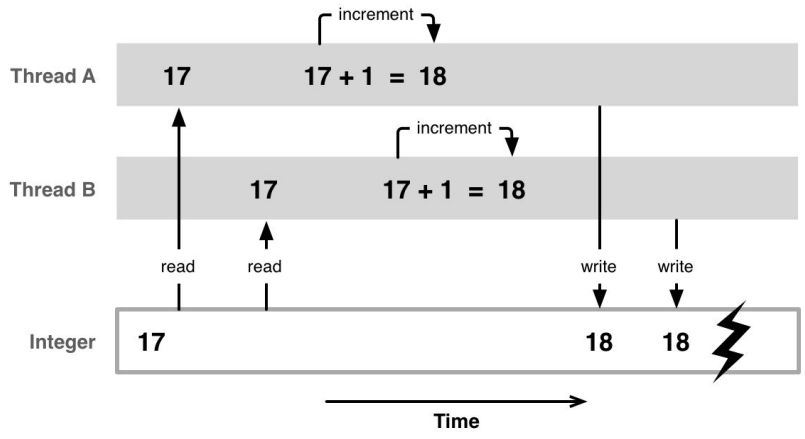
La chiamata alle API di Kraken mette a disposizione diversi indici utili per analizzare l’andamento del trading su quella coppia di monete, in particolare il valore di lettera (in

inglese “ask”) e il valore di offerta (in inglese “bid”). Questi due numeri rappresentano rispettivamente il prezzo minimo che un venditore è disposto ad accettare e il prezzo massimo che un compratore è disposto a pagare per scambiare moneta.

Nel caso del progetto descritto, viene utilizzato come valore di scambio quello di “ask”, perché, ipotizzando di vendere immediatamente la moneta scambiata, si sarebbe in linea con il resto del mercato.

Operazioni atomiche

In un progetto di questo tipo, che opera sul denaro diventa essenziale gestire in modo sicuro ogni operazione sul bilancio di un utente: si vuole evitare in ogni modo che due diverse parti del codice modifichino contemporaneamente il valore del bilancio, rendendolo non più consistente.



Considerando inoltre la necessità di aggiornare anche più volte al secondo il valore di bilancio di un utente durante la fase di mining risulta scomodo salvarlo come attributo nel documento dell’utente, perché ogni aggiornamento del bilancio porterebbe a bloccare ogni modifica sugli altri attributi del documento stesso. Per questo motivo il bilancio viene salvato come documento indipendente nel database, di tipo “counter”, ovvero come contatore atomico.
Questo permette di eseguire operazioni atomiche su di esso senza gestire manualmente le operazioni di lock e unlock, che vengono accodate ed eseguite direttamente dal database.

Generazione delle statistiche

Come descritto in precedenza, uno dei motivi per cui è stato scelto come database Couchbase è la presenza di un engine di Map-Reduce integrato tramite il sistema delle views.
Couchbase permette di isolare le risorse in termini di memoria e potenza di calcolo che verranno da esso utilizzate per mantenere le views aggiornate, in modo da non influire negativamente sui tempi di risposta ad ogni aggiornamento. Il momento in cui eseguire questi calcoli può essere scelto al momento della creazione della struttura del database: in caso sia necessario garantire l’integrità dei dati si può forzare Couchbase ad eseguire il round di map dei nuovi documenti e successivamente il reduce ad ogni aggiornamento.
In altri casi, dove è molto più importante garantire prestazioni elevate e bassa latenza si può configurare il database per eseguire il round di map-reduce a sua discrezione, quando ad esempio sono stati accumulati molti nuovi documenti.

Per questo progetto, dato che views di questo tipo sono utilizzate attualmente solo per la creazione delle statistiche si è deciso di optare per la seconda opzione, perché un ritardo di alcuni secondi nella visualizzazione dei dati statistici è più che accettabile.

Il sistema di views di Couchbase permette di produrre, tramite le operazioni di map e reduce, associazioni tra chiavi n-dimensionali e un valore sotto forma di oggetto JSON.
La possibilità di applicare filtri per range sulle chiavi multidimensionali rende molto semplice ed efficace la successiva fase di estrazione delle informazioni.

Al fine di produrre statistiche precise si è deciso di creare diversi accumulatori atomici per ogni chiave composta da quattro diversi parametri: id del client che esegue il mining, id del sito, id del proprietario del sito e ora.
Questi accumulatori contengono il numero di hash calcolati dal client durante il periodo di permanenza su un preciso sito ad una precisa ora e vengono aggiornati dal modulo “pool” che gestisce appunto la comunicazione con i dispositivi connessi per la fase di mining.

Durante la fase di map ogni chiave viene scomposta nei diversi parametri e viene creato un documento contenente come attributi i valori estratti.
Successivamente sono state definite due diverse views che hanno in comune questa prima fase di mapping, per ridurre i tempi di computazione: una prima views aggrega i dati utilizzando chiavi bidimensionali definite da id del sito e ora, mentre una seconda views utilizza chiavi bidimensionali di id del proprietario e ora.
La prima views permette di applicare efficientemente filtri ed estrarre il numero di hash calcolati e il numero di client connessi su un preciso sito ad una precisa ora o in un intervallo di ore per poter creare un grafico differenziato per ogni diverso sito.
La seconda views permette di ottenere un grafico generale per ogni proprietario, che è di fatto l’aggregazione di tutti i grafici dei singoli siti da esso gestiti.

Test di unità e CI

I test di unità sono stati scritti utilizzando Mocha e Chai, due framework molto popolari che insieme creano una suite completa per il testing di servizi REST. Vengono testate a fondo tutte le chiamate implementate per verificare che i diversi casi di errore vengano effettivamente riconosciuti e venga inviata una risposta con il corretto messaggio e codice di errore.

Grazie all’integrazione tra GitHub e Travis.CI, i test di unità sono stati configurati in modo tale da essere eseguiti automaticamente al momento del push di ogni commit su qualunque branch, con diverse versioni di NodeJS.
Dato che il branch master contiene il codice più stabile e pronto per la release, il repository è stato configurato in modo da rifiutare un merge o rebase su master senza che prima venga verificato il codice tramite il sistema di continuous integration.

Webapp

L’applicativo web è stato realizzato utilizzando il framework Angular JS che, applicando il paradigma Model-View-Controller, aiuta il programmatore a scrivere codice pulito ed organizzato.
Altra funzionalità molto utile di Angular è il two way data binding, ovvero la possibilità di associare in modo bidirezionale elementi, sia di input che di output, della view a variabili del model, lasciando al framework il compito di mantenere entrambi aggiornati.

Angular Material

L’intera interfaccia utente è stata realizzata sulla base della libreria Angular Material, che mette a disposizione decine di componenti in stile Material Design pronti da utilizzare. Seguendo le linee guida prodotte da Google si ottiene un applicativo web responsive, che si adatta ad ogni dispositivo, che su smartphone assomiglia molto ad un’applicazione nativa.

Gestione degli errori

Per semplificare la gestione degli errori si è deciso di sviluppare un filtro frapposto tra l’arrivo della risposta ad una chiamata REST e la fase di elaborazione di essa: senza questo filtro sarebbe stato necessario gestire in modo indipendente gli errori per ogni singola richiesta, allungando i tempi di sviluppo e introducendo spazio per possibili errori e dimenticanze.
Il filtro contiene una mappa tra codice di errore e messaggio di errore, attualmente solo in inglese, ma che può facilmente essere estesa per includere più lingue diverse.
In caso di errore la richiesta viene processata direttamente dal filtro che mostra il messaggio all’utente.

Grunt

Grunt.js è un “task-runner”, ovvero un programma composto da molti moduli che hanno funzioni specifiche e che possono essere composti per svolgere azioni molto complesse, dette “task”.
Viene utilizzato in questo progetto durante la fase di sviluppo, per automatizzare molte fasi che richiederebbero altrimenti ulteriore tempo di lavoro.

Il task che viene utilizzato più spesso è quello che prepara un server HTTP locale che possa servire l’applicativo e inietta un piccolo script in ogni pagina che si occupa di eseguire automaticamente il refresh ad ogni modifica.

Un altro task molto importante è quello di build: questo processo permette di ottimizzare le prestazioni unendo più file JavaScript in uno solo, riducendo quindi il numero di richieste HTTP necessarie per caricare l’intero applicativo.

Contemporaneamente, questo task riduce ulteriormente la dimensione del codice modificando i nomi delle variabili con altri più brevi, rendendo allo stesso tempo il codice più resistente ad attacchi perché più difficile da leggere.

Testing senza backend

Per semplificare lo sviluppo e isolare quando necessario l'ambiente dell'applicazione web da quello del backend, è stata inserita un'opzione impostata al momento dell'avvio dell'applicazione che permette di selezionare come endpoint per le chiamate un server remoto o una cartella locale di documenti JSON.
Questa cartella contiene un file per ogni coppia di percorso-metodo, in modo da poter specificare risposte differenti per metodi HTTP diversi sulla stessa chiamata.
Ad esempio, la chiamata GET al percorso /info viene tradotta nel file info-GET.json, che contiene una risposta pronta da servire all'applicazione. In questo modo diventa possibile sviluppare features dell'applicativo web senza che esse vengano prima aggiunte al backend.

Per motivi di sicurezza questa cartella contenente le risposte alle chiamate viene rimossa durante il periodo di "build" con Grunt, quindi viene nascosta all'utente finale con l'obiettivo di rendere più complicato un eventuale attacco.

Anche questo livello opera in modo completamente trasparente al resto dell'applicativo, che non ha modo di notare differenze tra una risposta di un verso server e una risposta proveniente dai documenti predefiniti.

Pool

Il pool è stato sviluppato in modo molto simile al backend, utilizzando NodeJS 6.10.
La parte principale del programma è la classe Pool, che implementa il protocollo JSON-RPC 2.0 per comunicare con un pool di livello superiore o direttamente con il client Monero.

socket.io

Per mantenere aperto un canale di comunicazione full duplex tra client e server si è deciso di utilizzare la libreria Socket.IO, perché implementa internamente diversi metodi di comunicazione (WebSocket, polling requests e altri), e sceglie automaticamente quello migliore per ogni browser.

Distribuzione del lavoro

La parte più significativa di questo programma è l'algoritmo per la distribuzione del lavoro tra diversi dispositivi, che deve tenere conto di molti fattori per garantire alte prestazioni e stabilità.

Per questo motivo è stato studiato un sistema per tenere traccia dei client connessi tramite cookies, in modo da poterli identificare durante il mining per due motivi:

1. Variare il carico di lavoro in base alle potenzialità del dispositivo

2. Assicurarsi che solo un'istanza del miner sia attiva su un client anche in presenza di più pagine aperte

Non possiamo infatti assumere che ogni dispositivo abbia la stessa potenza di calcolo, perché varia molto soprattutto tra un'architettura e l'altra (x86 e ARM).
Allo stesso tempo vogliamo assicurarci che anche se un browser aprisse più pagine contenenti il miner, solo un'istanza venga mantenuta attiva contemporaneamente, per non allocare troppe risorse.

Stimando l'hashrate di ogni dispositivo in base ai tempi di computazione dei job precedenti, possiamo di conseguenza stimare la grandezza del lavoro, in numero di hash da calcolare, da assegnare ad esso successivamente: il sistema cercherà di dimensionare il carico per fare in modo che venga trovato un hash valido ogni circa 10 secondi.
Il motivo per avere un tempo di hash così breve è che gli hash calcolati verranno accreditati al bilancio del sito solo se viene trovato un hash valido, per evitare che ci possano essere manipolazioni. Per questo motivo, assegnando job molto lunghi, da svariati minuti, si rischierebbe di non trovare mai un hash durante le visite brevi e quindi non accreditare nulla sul bilancio, nonostante siano stati di fatto calcolati molti hash.

Centralizzazione dei parametri di configurazione

Per semplificare il deploy di questo sistema sul server di produzione si è deciso di realizzare un sistema di caricamento dei parametri di configurazione da un file JSON.
Il file di esempio è strutturato in questo modo:

```
{
    "couchbase": {
        "connection_string": "couchbase://127.0.0.1"
    },
    "monero_pool": {
        "address": "....",
        "node_name": "server01",
        "host": "xmr.poolto.be",
        "port": 2999,
        "keepalive": 25
    }
}
```

Vengono definiti i parametri di connessione al database per l'aggiornamento delle statistiche e dei bilanci degli utenti e una serie di parametri di connessione al pool, come ad esempio l'indirizzo Monero da comunicare ad esso per ricevere la moneta generata.

Verifica delle shares

Spesso i pool di livello superiore implementano meccanismi per verificare che le shares ricevute dai client siano valide tramite un meccanismo basato sulla fiducia: client che inviano molte shares valide sono probabilmente "buoni" e la probabilità di eseguire un controllo sui futuri hash ricevuti da essi si abbassa.

Prendendo ispirazione da questo meccanismo è stato quindi implementato qualcosa di simile, per ridurre il carico di lavoro da dedicare alla verifica delle shares.

Per rendere ancora più efficiente il sistema di verifica si è pensato di sfruttare una funzionalità di NodeJS che permette, un po' come la JNI di Java, di creare dei bindings tra funzioni JavaScript e funzioni in C++ compilate per l'architettura richiesta.
Sfruttando il codice in C del miner, è stata quindi scritta una piccola classe C++ che mette a disposizione del programma in JS una funzione per verificare un hash ricevuto, con l'ulteriore vantaggio di non dover riscrivere la parte di calcolo di un hash in JavaScript.

Client

Il client è la parte più piccola dell'intero sistema e ha alcuni compiti molto precisi:

1. Gestire la comunicazione con il server
2. Caricare il miner adatto (asm.js o WebAssembly) in base alle capacità del browser
3. Mettere a disposizione un'interfaccia esterna per la configurazione del *website id*

Lo script principale, detto app.js, si occupa di creare un canale di comunicazione con il server e di avviare un tramite un Worker, ovvero su un thread separato, lo script miner.js, che a sua volta carica in memoria il vero e proprio miner e inizia a calcolare hash.
Si è deciso di eseguire su un thread separato il codice del miner per non influire negativamente sui tempi di caricamento e rendering della pagina, rendendo quindi il sistema quasi completamente trasparente all'utente finale.

Miner

Il miner implementa l'algoritmo Cryptonight, che è di fatto un'unione di più algoritmi di hashing e cifratura:

- AES 256, sia funzioni di cifratura di un blocco sia funzioni di espansione delle chiavi
- Algoritmo di hashing "Keccak"
- Algoritmo di hashing "Blake"
- Algoritmo di hashing "Groestl"
- Algoritmo di hashing "Skein"
- Algoritmo di hashing "JH"

Il miner utilizzato in questo progetto è di fatto un *fork* di una popolare versione open source, riadattata rimuovendo l'utilizzo delle istruzioni native AES che garantivano prestazioni elevate su architetture native x86 moderne.

Il codice in alcuni casi è implementato diversamente per architetture little endian e big endian, per migliorare ulteriormente le prestazioni andando a sfruttare le particolarità di ogni architettura.

Per la compilazione da C a JavaScript viene utilizzato Emscripten, che con il plugin Binaryen può anche generare eseguibili WebAssembly.

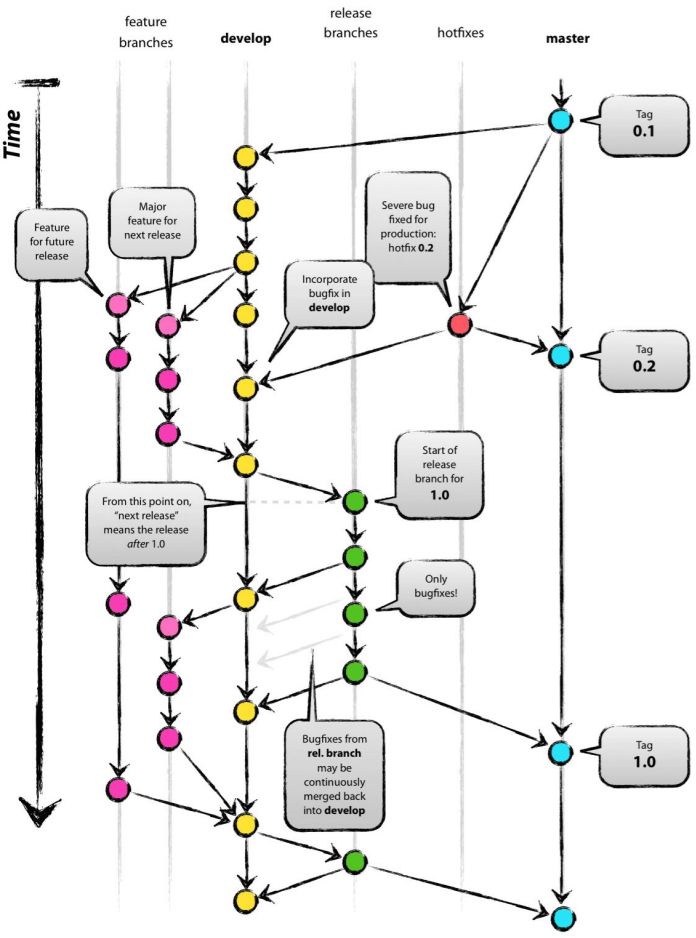
Ottimizzazione su architetture diverse

Per guadagnare ulteriormente in termini di prestazioni, molte parti del codice sono implementate diversamente per architetture big endian e little endian: per questo motivo al momento della compilazione vengono di fatto prodotti quattro diversi file, due per tipo ottimizzati per entrambe le architetture.

Strumenti Utilizzati

Per la gestione del codice scritto durante lo sviluppo si è scelto di utilizzare Git come sistema di versioning, separando ogni componente in un repository a sé stante.
Scegliendo come servizio di hosting GitHub, sono state sfruttate a fondo anche altre sue funzionalità, come l'integrazione con Travis CI per l'esecuzione automatizzata dei test di unità e del sistema di gestione delle "issues" e "pull request".
In più, si è deciso di scrivere tutti i messaggi dei commit, issues e PR in inglese, in caso diventasse necessario estendere il team di sviluppo in futuro.

Prendendo spunto da un modello di sviluppo molto popolare, sono stati utilizzati in modo particolare i branch di Git, per separare a livello pratico oltre che concettuale ogni modifica apportata al codice, che può essere così riassunto:



- Il branch master contiene il codice più stabile, ed ogni commit corrisponde ad una nuova versione rilasciata.
- Il branch develop è quello più attivo durante lo sviluppo di nuove feature da una release a quella successiva.
- I branch con prefisso hotfix_ sono sempre figli di una release e contengono principalmente bugfix rilevati in produzione su una release stabile. Una volta verificato il corretto funzionamento del codice, si procede a fare un merge su master per creare una nuova release e allo stesso tempo si includono gli hotfix in develop, per applicarli alla versione attualmente in sviluppo.
- I branch con prefisso release_ sono figli di develop, e vengono creati per apportare tutti i piccoli aggiustamenti che preparano una nuova release. Su questi branch non

si aggiungono mai nuove feature, ma si procede solamente a preparare il rilascio vero e proprio.
- I branch con prefisso feature_ vengono creati al momento dell'inizio dello sviluppo di una nuova feature e poi introdotti in develop.

Per migliorare questo modell, durante lo sviluppo sono state utilizzate le "issues" di GitHub per pianificare ulteriormente il lavoro: ogni nuova feature, prima di essere trascritta sotto forma di codice, va sempre prima inserita come issue, valutata e poi inserita all'interno di una milestone, che ha una precisa deadline.

Al momento del merge di una feature su develop o di una beta su master si apre una pull request per valutare nuovamente che tutto sia corretto prima di completare l'azione. Ovviamente questo sistema è molto più efficace se applicato da team di più persone, ma si è pensato fosse una buona idea lavorare in questo modo dall'inizio, nonostante attualmente l'intero progetto sia stato realizzato da una sola persona.

Per la scrittura dei messaggi dei commit si è scelto di adottare il metodo del titolo breve che riassuma le modifiche apportate insieme ad una descrizione estesa nelle righe successive che ne spieghi le motivazioni, per evitare i classici messaggi poco informativi, come illustrato in modo ironico in questo XKCD.

Ringraziamenti

Vorrei concludere questo documento ringraziando tutti i docenti che mi hanno seguito durante il triennio, ma in particolar modo il prof. Alessandro Bugatti, per avermi introdotto al mondo delle gare di informatica e per avermi aiutato nelle fasi di allenamento: tutte le esperienze vissute grazie a questi progetti extracurricolari mi hanno migliorato profondamente sia come tecnico sia, soprattutto, come persona, e penso che senza di lui sarebbe stato molto più difficile per me riuscire ad ottenere gli stessi successi.

Sitografia

- Hash function - Wikipedia. https://en.wikipedia.org/wiki/Hash_function
- Public-Key cryptography - Wikipedia. https://en.wikipedia.org/wiki/Public-key_cryptography
- Caesar cipher - Wikipedia. https://en.wikipedia.org/wiki/Caesar_cipher
- Bitcoin - Wikipedia. https://en.wikipedia.org/wiki/Bitcoin
- Target - Bitcoin Wiki. https://en.bitcoin.it/wiki/Target
- Difficulty - Bitcoin Wiki. https://en.bitcoin.it/wiki/Difficulty
- How Bitcoin works under the hood - YouTube (by CuriousInventor). https://www.youtube.com/watch?v=Lx9zgZCMqXE
- Monero (cryptocurrency) - Wikipedia. https://en.wikipedia.org/wiki/Monero_(cryptocurrency)
- Cryptonight Hash Function - CryptoNote Foundation. https://cryptonote.org/cns/cns008.txt
- Performance & scalability of Couchbase Server - Couchbase. https://www.couchbase.com/nosql-resources/presentations/2016/performance--scalability-of-couchbase-server.html
- MapReduce - Wikipedia. https://en.wikipedia.org/wiki/MapReduce
- Bid-ask spread - Wikipedia. https://en.wikipedia.org/wiki/Bid-ask_spread
- How to write a Git commit message - Chris Beams. https://chris.beams.io/posts/git-commit/
- Git commit - XKCD #1296. - https://xkcd.com/1296/
- A successful Git branching model - Vincent Driessen. http://nvie.com/posts/a-successful-git-branching-model/