



Università degli Studi di Napoli "Parthenope"

DIPARTIMENTO DI SCIENZE E TECNOLOGIE

Corso di laurea in Informatica

Tesi di Laurea Triennale

**PROGETTAZIONE E SVILUPPO DI UN
SISTEMA DISTRIBUITO PER L'ANALISI
DELLE TRANSAZIONI BITCOIN**

Relatore

prof. Alessio Ferone

Laureando

Antonio Riccardi
mat. 0124000411

Anno accademico 2017 - 2018

Ai miei genitori.

Ringraziamenti

Sommario

La seguente tesi ha come obiettivo la realizzazione di un'applicazione distribuita in grado di fare analisi di transazioni provenienti dalla blockchain di Bitcoin.

Il nocciolo della questione affrontata è quello di studiare la struttura di ogni singola piattaforma distribuita e di implementare componenti in grado di ricavare informazioni utili per l'analisi delle transazioni. In particolare, ci si soffermerà sulle principali tecnologie informatiche che permettono di costruire sistemi distribuiti, come Spark ed Hadoop, e sulle strutture dati che facilitano l'immagazzinamento delle informazioni come Neo4j.

Nella prima parte della tesi si procede con l'introduzione alla tecnologia alla base della moneta virtuale Bitcoin, facendo luce su cos'è la Blockchain e quali vantaggi/svantaggi presenta rispetto ad altri sistemi che gestiscono transazioni monetarie. A seguire, sono presi in esame le principali soluzioni di analisi di transazioni già esistenti.

Il secondo capitolo dell'elaborato mostra una panoramica del disegno dell'architettura del sistema realizzato, mettendo in luce i principali componenti del sistema. A tal fine è presente un'ampia e dettagliata descrizione di ogni singolo componente del sistema specificando i vantaggi ottenuti dal proprio utilizzo.

Il terzo capitolo descrive come è stata fatta l'implementazione del sistema, mostrando porzioni di codice e screen dell'applicazione che ci aiutano nella comprensione di come funziona il sistema.

Infine nell'ultimo capitolo vengono trattate le considerazioni finali del lavoro svolto soffermandosi anche sui possibili sviluppi.

Indice

Elenco delle figure	6
Elenco delle tabelle	8
Elenco dei listati	9
1 Bitcoin: Fonte di bigdata	10
1.1 Blockchain	10
1.2 Bitcoin	12
1.2.1 Come avviene una transazione	13
1.2.2 Come avviene una transazione dal punto di vista tecnico	14
1.3 Analisi dati su sistemi distribuiti	18
1.3.1 Caratteristiche di un sistema distribuito	19
1.3.2 Vantaggi e Svantaggi	19
2 Overview e progettazione di sistema	21
2.1 Scopo del progetto	21
2.2 Architettura del progetto	21
2.3 Sistema distribuito	23
2.3.1 Bitcoind	24
2.3.1.1 ZeroMQ	25
2.3.2 Apache Spark	26
2.3.2.1 Spark Streaming	28
2.3.2.2 GraphX	29
2.3.3 Hadoop HDFS	30
2.3.4 Neo4j	33
2.3.5 Zookeeper	37
2.3.5.1 Kafka	39
2.4 WebApp	40
2.4.1 Node.js	42
2.4.1.1 Express.js ed Handlebars	44
2.4.1.2 WebSocket	46
2.4.1.3 MaterializeCSS	49
2.4.1.4 D3.js	49

3 Scelte tecniche ed implementazione	52
3.1 Diagramma delle classi	54
3.2 Codice	57
3.3 Interfaccia utente	74
4 Conclusioni e sviluppi futuri	80
Riferimenti bibliografici	81

Elenco delle figure

1.1	Rete distribuita di nodi paritari.	11
1.2	Catena di blocchi nella blockchain.	11
1.3	Logo Bitcoin.	13
1.4	Come avviene una transazione Bitcoin.	14
1.5	Azioni del ricevente.	15
1.6	Prima parte. Azioni del pagante.	15
1.7	Seconda parte. Azioni del pagante.	16
1.8	Azione dei miner.	16
1.9	Merkle tree.	17
1.10	Differenza tra sistema distribuito e centralizzato.	18
2.1	Architettura completa.	22
2.2	Architettura in dettaglio del sistema distribuito.	24
2.3	Messaggio inviato con la modalità Publish-Subscribe.	26
2.4	Visualizzazione di un RDD partizionato.	27
2.5	Infrastruttura Spark.	28
2.6	Come vengono gestiti i dati in Spark Streaming.	29
2.7	Logo GraphX.	29
2.8	Visualizzazione invio dati ad HDFS.	31
2.9	Architettura HDFS.	33
2.10	Interfaccia web di Neo4j nella quale è eseguita una query.	35
2.11	Funzionamento di Zookeeper.	38
2.12	Come funziona Kafka	40
2.13	Architettura in dettaglio della webapp.	42
2.14	Gestione eventi con Node.js.	43
2.15	Visualizzazione della catena di funzioni per rispondere ad una richiesta	45
2.16	Comunicazione client/server tramite WebSocket	48
2.17	Logo D3.js	51
3.1	UML delle classi (Sistema distribuito).	55
3.2	Alberatura file di Blochchain Explorer.	56
3.3	Dettaglio transazione.	65
3.4	Grafo delle transazioni completo.	67
3.5	Grafo delle transazioni completo.	73

3.6	Home page Blockchain Explorer.	74
3.7	Elenco ultime transazioni.	75
3.8	Dettaglio transazione.	76
3.9	Visualizzazione intero grafo con calcolo del Page Rank.	77
3.10	Dettaglio nodo centrale.	78
3.11	Dettaglio transazione all'interno del grafo.	79

Elenco delle tabelle

Elenco dei listati

2.1	Query Cypher.	36
2.2	Esempio di HTML scritto con Handlebars.	45
2.3	Esempio di variabile passata dal controller al template engine.	46
3.1	Metodo readProperties.	58
3.2	Inizializzazione Spark Streaming.	58
3.3	Creazione oggetto sparkConf.	59
3.4	Metodo della libreria Spark Streaming ZeroMQ.	59
3.5	Funzione bytesToObjects.	59
3.6	Salvataggio Bytes su HDFS.	60
3.7	Array di Byte trasformato in oggetto Block.	60
3.8	Prelievo transazioni e salvataggio in Neo4j.	61
3.9	Metodo che crea o modifica una transazione in Neo4j.	61
3.10	Calcolo PageRank e salvataggio su Neo4j.	62
3.11	Invio dati a Kafka.	63
3.12	Creazione server in NodeJS.	64
3.13	Associazione URL-Callback in Express.	64
3.14	Creazione subscriber Kafka.	66
3.15	Creazione di un Server WebSocket.	66
3.16	Inizializzazione svg per il grafo.	67
3.17	Creazione linee.	68
3.18	Creazione dei nodi del grafo.	69
3.19	Utilizzo DataTable.	70
3.20	Associazione Express-Handlebars.	71
3.21	Template Handlebars.	71

Capitolo 1

Bitcoin: Fonte di bigdata

Nel lontano 2009 uno pseudonimo Satoshi Nakamoto pubblicò il manifesto "*Bitcoin: A Peer-to-Peer Electronic Cash System*"^[21] sancendo la nascita di una nuova moneta digitale chiamata Bitcoin, che fa del suo punto di forza la libertà e la sicurezza. Nel corso degli anni il valore della moneta è cresciuto sino a raggiungere picchi di 19290 dollari nel 17 Dicembre 2017 ^[5]. Di pari passo, anche il numero di transazioni giornaliere sulla rete bitcoin ha raggiunto cifre esorbitanti riuscendo a validare 5 transazioni al secondo^[4], questa enorme mole di dati è possibile processarla solo con sistemi distribuiti che gestiscono i Big Data. Quest'ultimi rappresentano tutti quei dati che possono essere disponibili in enormi volumi, possono presentarsi con formati semistrutturati o addirittura destrutturati e possono essere prodotti con estrema velocità. Uno degli aspetti che caratterizzano i big data è la loro quantità. Questi dati infatti, sono generati dall'utente attraverso gli strumenti del Web 2.0, sistemi gestionali, oppure generati automaticamente da macchine (sensori, sistemi di calcolo) assumendo volumi rilevanti, non più gestibili con strumenti di database tradizionali. Non a caso l'elaborato di tesi, utilizza un sistema distribuito creato ad hoc per i big data per gestire l'enorme volume di transazioni che circolano nella rete Bitcoin.

Bitcoin, oltre all'impatto economico, ha avuto un importante ruolo nel campo della ricerca, donandoci un sistema che gestisce le transazioni tra due parti in un network peer-to-peer, diversamente dai sistemi tradizionali, denominato Blockchain.

1.1 Blockchain

La blockchain come dice il nome è una catena di blocchi che implementano un database aperto e distribuito, atto a memorizzare le transazioni tra due parti in modo sicuro, verificabile e permanente. In altre parole, la blockchain rappresenta il libro contabile (o libro mastro), ossia il registro sul quale sono riportati tutti gli scambi tra le parti. Questo libro mastro è distribuito (Distributed Ledger) replicato e sincronizzato tra tutti i membri di una rete. In questo database vengono registrate le transazioni (come lo scambio di beni o informazioni) tra i partecipanti alla rete.

I dati non sono memorizzati su un solo computer ma su più macchine collegate tra loro attraverso una rete peer-to-peer [1.1] sottoforma di blocchi.

Ciascun nodo è autorizzato ad aggiornare e gestire il libro contabile distribuito in modo indipendente, ma sotto il controllo consensuale degli altri nodi. Infatti, gli aggiornamenti non sono più gestiti, come accadeva tradizionalmente, sotto il controllo rigoroso di un'autorità centrale, ma sono invece creati e caricati da ciascun nodo in modo appunto indipendente. In questo modo ogni partecipante è in grado di processare e controllare ogni transazione ma, nello stesso tempo ogni singola transazione, essendo gestita in autonomia, deve essere verificata, crittografata e approvata dalla maggioranza dei partecipanti alla rete.

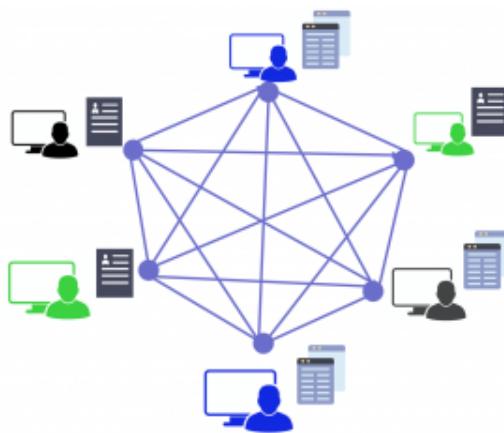


Figura 1.1: Rete distribuita di nodi paritari.

Ogni blocco contenuto nella blockchain archivia un insieme di transazioni validate correlate da un Marcatore Temporale (Timestamp) ed un hash (una stringa alfanumerica) che identifica il blocco in modo univoco e che permette il collegamento con il blocco precedente[1.2].

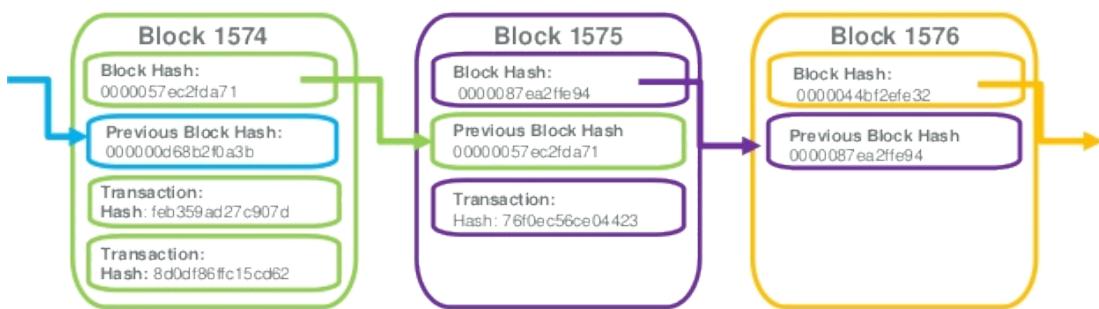


Figura 1.2: Catena di blocchi nella blockchain.

Perché un nuovo blocco di transazioni sia aggiunto alla Blockchain è necessario appunto che sia controllato, validato e crittografato. Solo con questo passaggio può poi

diventare attivo ed essere aggiunto alla Blockchain. Per effettuare questo passaggio è necessario che ogni volta che viene composto un blocco venga risolto un complesso problema matematico che richiede un cospicuo impegno anche in termini di potenza e di capacità elaborativa. Questa operazione viene definita come “Mining” ed è svolta dai “Miner”. La risoluzione del problema è un processo irreversibile per cui il blocco valido aggiunto alla catena diviene immutabile.

In conclusione, questa tecnologia può essere utilizzata in tutti gli ambiti in cui è necessaria una relazione tra più persone o gruppi. Ad esempio può garantire il corretto scambio di titoli e azioni o sostituire un atto notarile, perché ogni transazione viene sorvegliata da una rete di nodi che ne garantiscono la correttezza senza l’ausilio di intermediari.

Bitcoin sfrutta questa tecnologia per gestire le transazioni finanziarie in modo sicuro e del tutto autonomo.

1.2 Bitcoin

Bitcoin (codice: BTC o XBT) è una criptovaluta e un sistema di pagamento mondiale creato nel 2009 da un anonimo inventore, noto con lo pseudonimo di Satoshi Nakamoto, che sviluppò un’idea da lui stesso presentata su Internet a fine 2008. Per convenzione se il termine Bitcoin è utilizzato con l’iniziale maiuscola si riferisce alla tecnologia e alla rete, mentre se minuscola (bitcoin) si riferisce alla valuta in sé.^[2]

Il progetto Bitcoin è nato con l’intento di risolvere i problemi di fiducia, trasparenza e responsabilità tra due parti nello scambio di denaro per beni e servizi su Internet, senza l’utilizzo di intermediari. Bitcoin, infatti, rappresenta la prima rete di pagamento che utilizza una tecnologia distribuita peer-to-peer per operare senza un’autorità centrale: la gestione delle transazioni e l’emissione di denaro sono effettuati collettivamente dalla rete.

La rete Bitcoin utilizzando la tecnologia blockchain, è una catena di blocchi concatenati tra di loro, infatti ogni blocco contiene il riferimento al hash del blocco precedente. Questo sistema è immutabile perché se si volesse modificare un blocco si dovrebbero modificare tutti i blocchi successivi ad esso, per fare ciò si dovrebbe creare una catena di blocchi che è più lunga di quella esistente. Ovviamente, la distribuzione e la natura delle tempistiche del processo rendono praticamente impossibile che ciò accada.

Inoltre, utilizzando un sistema distribuito, permette di tenere traccia di tutti i trasferimenti in modo da evitare il problema del double spending (spendere due volte). Tutti gli utenti sono a conoscenza di ciò che accade e pertanto non c’è bisogno di una autorità centrale che gestisca le transazioni.

In aggiunta, Bitcoin consente il possesso e il trasferimento anonimo delle monete, ed infatti i dati necessari per usufruire dei propri bitcoin sono salvati in uno o più personal computer sotto forma di digital wallet (portafoglio). I bitcoin sono trasferiti tramite Internet a chiunque disponga di un indirizzo bitcoin. La struttura peer-to-peer della rete e l’assenza di un ente centrale rende impossibile a qualsiasi autorità il blocco dei trasferimenti, il sequestro dei bitcoin senza il possesso delle relative chiavi o la svalutazione dovuta all’immissione di nuova moneta. Per questo motivo i bitcoin sono la moneta più utilizzata nel Deep Web.

Ricapitolando, le principali caratteristiche su cui si basa questo sistema di enorme successo sono le seguenti:

- **Nessuna autorità centrale**: non dipende da nessuna terza parte privata o ente governativo, e, pertanto, il valore dei BTC è liberamente contrattato sul mercato. Si tratta quindi di un sistema decentralizzato;
- **Irreversibile e non falsificabile**: una volta che una transazione è stata effettuata ed è inclusa nella blockchain, non può più essere annullata, nemmeno dal mittente;
- **Anonimo**: chiunque può scaricare il software ed iniziare ad effettuare transazioni senza registrarsi, comunicare dati personali e senza svelare la propria identità.
- **Sistema distribuito su rete Peer-to-peer (rete paritaria o paritetica)**: qualsiasi nodo è in grado di avviare e completare una transazione in modo autonomo.
- **Inflazione determinata a priori**: L'emissione di nuovi BTC è determinata dall'algoritmo stesso del programma, e non può essere modificata.

Per capire meglio questi punti, verrà illustrata in modo dettagliato una transazione tra due utenti Alice e Bob che vogliono scambiarsi bitcoin.



Figura 1.3: Logo Bitcoin.

1.2.1 Come avviene una transazione

Un esempio pratico di scambio di moneta aiuterà nel capire come funziona il sistema. I nostri attori saranno Alice (pagante) e Bob (ricevente).

I passi da eseguire sono:

1. Bob comanda alla sua applicazione (wallet) su PC o smartphone di creare un indirizzo. Il software restituisce una sequenza alfanumerica da 26 a 35 caratteri. Questo è un esempio: 12gXGyXWkvyDAjVKZHyGGstVYyXJ6ZjqqV
2. Bob copia l'indirizzo e lo mostra al mittente tramite qualunque mezzo: via mail, messaggio, QR code, pubblicato su un sito internet, scritto su carta, dettato al telefono etc.
3. Alice inserisce nel suo software l'indirizzo di Bob e la quantità di Bitcoin da inviare, inoltre specifica l'ammontare della commissione da pagare al minatore (detto "miner") per convalidare la transazione. Ad oggi i più comuni software stabiliscono automaticamente una commissione fissa a 0,00001 bitcoin.
4. Bob ritrova sul suo software la transazione avvenuta, ma prima di considerare il pagamento effettuato attende che la transazione sia inserita nella blockchain.

5. In un massimo di 10 minuti circa la transazione viene inserita in un blocco della blockchain da parte del miner, che ha convenienza a inserirla perché in questo modo ottiene la commissione pagata da Alice.
6. Bob può sentirsi sicuro che la transazione è confermata con l'aumentare di blocchi che vengono aggiunti alla blockchain conseguentemente a quello che contiene la transazione.
7. Bob avrà nel proprio portafoglio i bitcoin inviati da Alice per spenderli in una nuova transazione.



Figura 1.4: Come avviene una transazione Bitcoin.

1.2.2 Come avviene una transazione dal punto di vista tecnico

La stessa transazione viene affrontata con un occhio più tecnico, soffermando l'attenzione sulla parte di creazione delle chiavi e della conferma ed aggiunta del blocco.

I passi da eseguire sono:

1. Tramite il software che si occupa del wallet, Bob genera in modo del tutto casuale una chiave privata, che viene salvata sul suo computer.
2. La chiave privata viene convertita in una chiave pubblica tramite un procedimento matematico. Comunemente è il software di Bob a generare automaticamente la chiavi quando Bob chiede all'applicazione di creare un indirizzo da comunicare al mittente. In realtà Bob potrebbe utilizzare una chiave privata facilmente ricordabile, come "sum qui sum", e ricavare Public key e indirizzo da questa. Il procedimento matematico si basa su un algoritmo chiamato "Elliptic Curve Digital Signature Algorithm"^[2] che utilizza la curva ellittica.
È teoricamente possibile ma statisticamente impraticabile scoprire la chiave privata partendo da quella pubblica: poiché il procedimento matematico applicato è unidirezionale, il processo inverso per indovinare la chiave privata richiederebbe una quantità di tentativi e una potenza di calcolo talmente enorme da essere al di là di ogni possibilità
3. La chiave pubblica viene a sua volta crittografata e accorciata tramite un hash. Possiamo chiamare la nuova chiave pubblica Public Key Hash. La chiave pubblica originaria invece è detta Full Public Key.
4. L'hash della chiave pubblica viene convertito in una riga di massimo 35 caratteri (per comodità pratica), che costituisce l'indirizzo del portafoglio di Bob. Per esempio: 12gXGyXWkvyDAjVKZHyGGstVYyXJ6ZjqqV.

5. Bob spedisce l'indirizzo ad Alice.

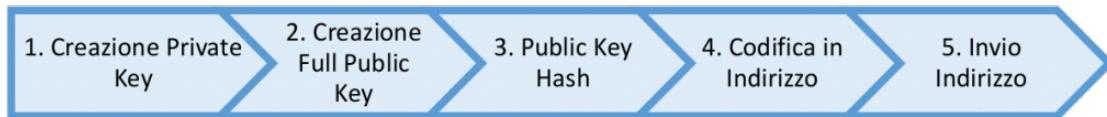


Figura 1.5: Azioni del ricevente.

6. Il software di Alice decodifica immediatamente l'indirizzo in una normale Public Key Hash
7. Alice crea la transazione. Si può pensare la transazione come un codice che contiene diverse informazioni, ciascuna rappresentabile come una stringa composta da molti caratteri:
 - l'input: uno o più output di una transazione precedente fatta nei confronti di Alice, da cui ella attinge i bitcoin che «spedisce» nel nuovo output
 - l'output: la quantità di bitcoin spediti. Possono esserci più output per ogni transazione, ciascuno identificato con un ID specifico
 - l'istruzione per la firma (la "signature script"): ovvero le istruzioni che Bob dovrà fornire per convalidare la transazione, dimostrando di essere il possessore del nuovo output. È proprio per la creazione dello script che il software di Alice ha bisogno del Public Key Hash fornito da Bob. Le informazioni necessarie per validare la firma sono due, entrambe già in possesso di Bob: la full Public Key e la Private Key, che dovranno combaciare col Public Key Hash specificato da Alice nello script.

Queste informazioni vengono processate insieme nella creazione di un unico hash chiamato txid (transaction identifier)



Figura 1.6: Prima parte. Azioni del pagante.

8. Tutti i bitcoin che Alice ha a disposizione su un particolare input vengono "spediti" nella transazione. Infatti nella transazione è coinvolta sempre l'intera quantità di bitcoin presenti nell'input anche se Bob ne ha richiesti molti meno. Se Alice dispone di un input di 100 bitcoin e ne trasferisce 20 a Bob, l'input è sempre trasferito nella sua interezza di 100 bitcoin. In questo caso avrà due output diversi, uno di 80 bitcoin (al lordo della commissione per il miner) che tornano al portafoglio di Alice (il change output), l'altro di 20 bitcoin che vanno all'indirizzo di Bob. L'unico caso

di transazione che abbia un solo input e un solo output è quello in cui l'input corrisponde esattamente all'ammontare richiesto da chi riceve i bitcoin. Spesso le transazioni hanno più output, e quindi i bitcoin trasferiti vanno ad indirizzi con diverse chiavi pubbliche e private.

9. Alice trasmette via internet al software di tutti gli altri nodi tutte le informazioni relative alla transazione. I nodi sono rappresentati da tutti coloro che hanno il software Bitcoin Core sui propri pc/dispositivi o numerosi altri software che permettono di "collegarsi" al network Bitcoin.



Figura 1.7: Seconda parte. Azioni del pagante.

10. I minatori inseriscono le transazioni ancora non confermate nella blockchain. Per inserire le transazioni all'interno della blockchain il miner deve creare un nuovo blocco, processo che richiede una quantità di calcolo molto elevata e dunque una spesa in energia elettrica e strumenti. Un miner ha interesse a inserire quante più transazioni nel blocco che vuole creare poiché guadagnerà tutte le commissioni pagate su ciascuna transazione. Se una transazione non include alcuna commissione, il miner non ha alcun interesse economico nell'inserirla nel blocco.

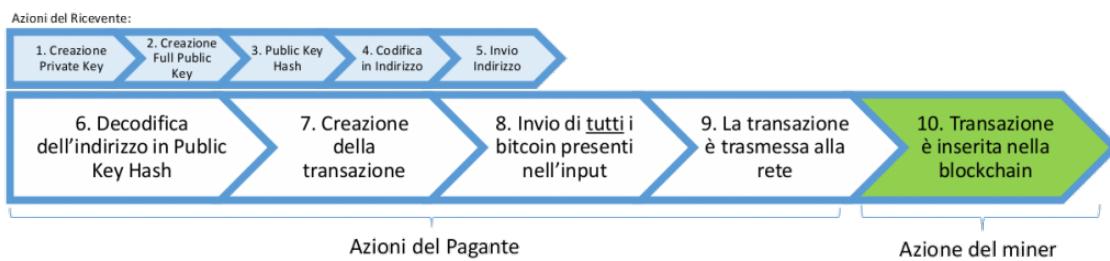


Figura 1.8: Azione dei miner.

Per inserire le transazioni nel blocco, i minatori partono dagli id delle transazioni (txid), ciascuno dei quali rappresenta l'hash di tutte le informazioni inerenti una singola transazione (passo 7).

I txid sono accoppiati due a due, creando un hash per ogni coppia di transazioni. Ogni hash viene poi accoppiato con un altro hash, creando un hash figlio dei due hash precedenti, e così via finché non si arriva a un unico hash. In caso di numeri dispari un hash viene processato con una sua copia identica. Questo procedimento può essere rappresentato come un albero, il cosiddetto Merkle tree [1.9], dove le foglie sono le transazioni txid, i rami (biforcuti) gli hash intermedi e la radice l'hash finale, prodotto di tutti gli altri hash: la Merkle root. L'hash finale è come l'ultimo

di una stirpe e porta con sé il “DNA” di tutti gli hash precedenti[11].

Grazie alla struttura del Merkle tree, non è necessario conoscere tutte le transazioni incluse in un blocco per verificare che una singola transazione ne faccia parte, è invece sufficiente seguire un particolare ramo che collega una foglia (una transazione) alla merkle root.

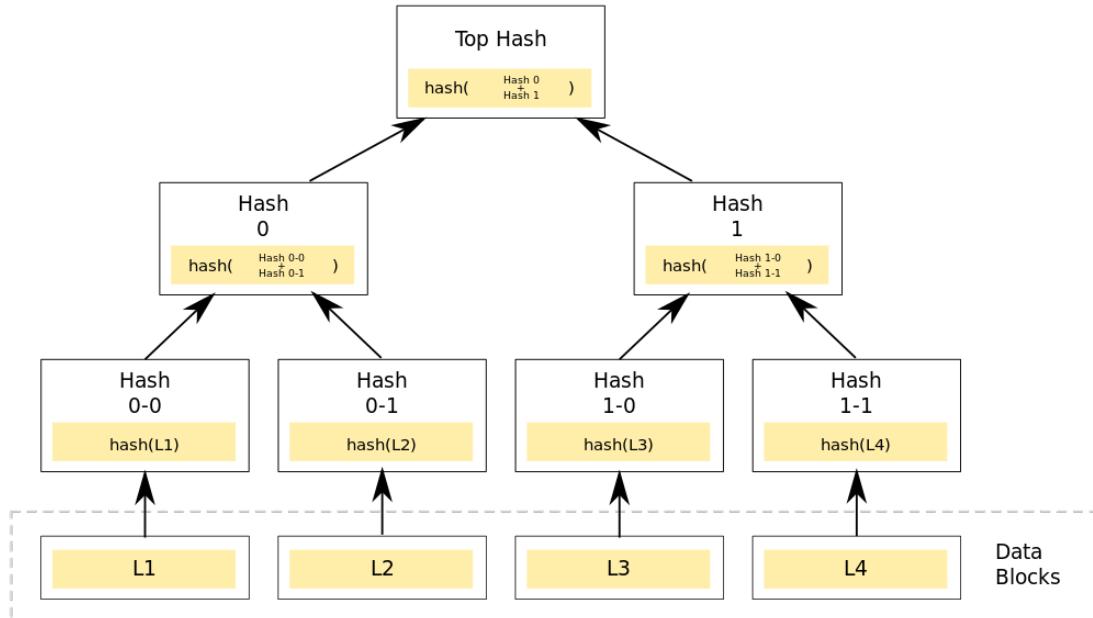


Figura 1.9: Merkle tree.

11. Bob ora vuole spendere i suoi nuovi bitcoin in una nuova transazione, il destinatario è Charlie. Bob segue la stessa procedura che ha seguito Alice, creando una transazione specificando output, signature script (per cui gli serve il public key hash di Charlie), timestamp e versione del software.

Bob però deve dimostrare di essere il possessore dei bitcoin che invia a Charlie, ovvero i bitcoin presenti nell'input della transazione. Tale input è anche l'output della transazione fra Alice e Bob. Quest'ultimo per dimostrare di possedere l'output di quella transazione deve porre la sua firma (signature). Bob inserisce quindi la sua Full Public Key, verificando che corrisponde al Public Key Hash dato in precedenza ad Alice, e la sua Private Key, che rappresenta la conferma che Bob solo è la persona che ha originato inizialmente quella Public Key. Infatti seppur sia teoricamente possibile, è per motivi statistici «infattibile» scoprire la Private Key partendo dalla Public Key. Il procedimento di firma è del tutto automatizzato dal software Bitcoin[28].

1.3 Analisi dati su sistemi distribuiti

Il numero di utenti che ogni giorno spende bitcoin è in costante crescita. La tecnologia Bitcoin quindi è costretta a lavorare con volumi di dati sempre crescente. Per questo motivo, le applicazioni che fanno analisi devono adattarsi scegliendo sistemi consoni a queste moli di dati: i Sistemi distribuiti.

Esistono più definizioni (più o meno equivalenti fra loro) di un sistema distribuito, fra cui:

- "Un sistema distribuito è una porzione di software che assicura che un insieme di calcolatori appaiano come un unico sistema coerente agli utenti del sistema stesso" (Maarten van Steen, 2016).[1]
- "Un sistema distribuito consiste di un'insieme di calcolatori autonomi, connessi fra loro tramite una rete e un middleware di distribuzione, che permette ai computer di coordinare le loro attività e di condividere le risorse del sistema, in modo che gli utenti percepiscano il sistema come un unico servizio integrato di calcolo" (Wolfgang Emmerich, 1997).[9]
- "Un sistema distribuito è un sistema in cui il fallimento di un computer che non sapevi neppure esistere può rendere il tuo computer inutilizzabile" (Leslie Lamport, 1987).[17]

Sintetizzando, un sistema distribuito è un insieme di processori indipendenti (con proprie risorse hardware/software) interconnessi da una rete di comunicazione, che cooperano per condividere alcune delle risorse ovunque distribuite ed eseguire algoritmi parallelamente. Questi sistemi riescono ad apparire all'utente come un singolo sistema, permettendo di avere una certa estrazione dall'hardware dei singoli nodi.

I sistemi distribuiti [1.10a] si contrappongono ai sistemi centralizzati, nella quale tutto il lavoro è eseguito da un solo calcolatore [1.10b]. Un esempio di sistema distribuito è la rete Internet stessa, che si estende a livello mondiale comprendendo risorse fisicamente molto distanti tra loro, in cui processi con funzioni diverse e connessi da reti di vario tipo si scambiano messaggi informativi basati su disparati protocolli di comunicazione.

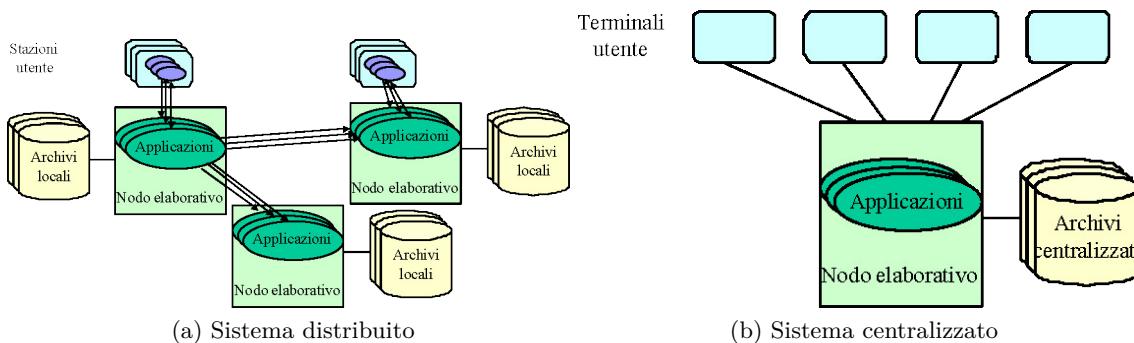


Figura 1.10: Differenza tra sistema distribuito e centralizzato.

L'applicazione creata è eseguita su un sistema distribuito questo perché abbiamo bisogno di tempi di risposta bassi ed alta affidabilità dei dati. Per questo motivo il codice viene partizionato in piccoli problemi ed eseguiti dai nodi del nostro cluster. Il processo di partizione dei dati viene automatizzato dal framework Spark [2.3.2] che si occupa della gestione, invio e recupero dei dati tra i vari nodi.

1.3.1 Caratteristiche di un sistema distribuito

Un sistema distribuito si definisce tale se rispetta alcune caratteristiche come:

- **Remoto:** le componenti di un sistema distribuito devono poter essere trattate allo stesso modo sia che siano in locale che in remoto.
- **Concorrenza:** è possibile eseguire contemporaneamente due o più istruzioni su macchine differenti.
- **Malfunzionamenti parziali:** Ogni componente del sistema può smettere di funzionare correttamente indipendentemente dalle altre componenti del sistema; questo non deve compromettere le funzionalità dell'intero sistema. I fallimenti che possono affliggere i processi possono essere di varia natura.
- **Eterogeneità:** un sistema distribuito è eterogeneo per tecnologia sia hardware che software. Si realizza in tutti i contesti come rete di comunicazione, protocollo di rete, linguaggi di programmazione, applicazioni, etc.
- **Autonomia:** un sistema distribuito non ha un singolo punto dal quale può essere controllato, coordinato e gestito. La collaborazione va ottenuta inviando messaggi tra le varie componenti del sistema e gestita tramite politiche di condivisione e di accesso che devono essere rigorosamente seguite.
- **Evoluzione:** un sistema distribuito può cambiare sostanzialmente durante la sua vita, sia perché cambia l'ambiente sia perché cambia la tecnologia utilizzata. L'obiettivo è quello di assecondare questi cambiamenti senza costi eccessivi.[\[26\]](#)

1.3.2 Vantaggi e Svantaggi

I sistemi distribuiti hanno dei pro e contro che sono da prendere in considerazione quando si vogliono adottare soluzioni di questo tipo.

I vantaggi nell'utilizzo dei sistemi sono:

- **Connattività e collaborazione:** possibilità di condividere risorse hardware e software (compresi dati e applicazioni)
- **Prestazioni e scalabilità:** la possibilità di aggiungere risorse, fornisce la capacità di migliorare le prestazioni e sostenere un carico che aumenta (scalabilità orizzontale)
- **Tolleranza ai guasti:** grazie alla possibilità di replicare risorse e dati

- **Apertura:** l'uso di protocolli standard aperti favorisce l'interoperabilità di hardware e software di fornitori diversi
- **Economicità:** i sistemi distribuiti offrono spesso (ma non sempre) un miglior rapporto prezzo/qualità che i sistemi centralizzati basati su mainframe

Gli svantaggi invece sono:

- **Complessità:** i sistemi distribuiti sono più complessi di quelli centralizzati e quindi risultano più difficili da capire, inoltre, lo sviluppo delle applicazioni deve essere implementato ad hoc.
- **Sicurezza:** l'accessibilità in rete pone problematiche di sicurezza
- **Gestibilità:** è necessario uno sforzo maggiore per la gestione del sistema operativo e delle applicazioni

Capitolo 2

Overview e progettazione di sistema

In questo capitolo viene messo in luce lo scopo dell'elaborato realizzato con particolare attenzione alle scelte progettuali adottate e alle ricerche effettuate per implementare l'argomento trattato, onde presentare un quadro generale completo. Viene, inoltre, descritta l'architettura del software, sottolineando ed approfondendo le sue principali componenti.

2.1 Scopo del progetto

Lo scopo principale del lavoro di tesi è quello di creare un sistema distribuito che permetta la visualizzazione real time, la storicizzazione e l'analisi delle transazioni provenienti dalla blockchain di bitcoin. L'obiettivo, dunque, è quello di riuscire a creare un sistema che gestisca grandi quantità di dati in un ambiente distribuito, garantendo affidabilità e consistenza dei dati anche in caso di guasti.

Il sistema si rivolge ad un pubblico che vuole fare analisi delle transazioni processate dalla rete bitcoin. L'utente infatti, può controllare in real time le ultime transazioni elaborate, monitorare l'intera rete blockchain per risalire a tutta la catena di transazioni, oppure controllare gli hash (indirizzi) che hanno avuto maggior punteggio di PageRank.

Le ultime transazioni elaborate, hanno una serie di informazioni di dettaglio come il blocco di appartenenza, l'hash che ha generato quella transazione, il timestamp ed i destinatari dei bitcoin. Mentre per quanto riguarda il monitoraggio dell'intera blockchain, l'utente può navigare con l'utilizzo del proprio mouse, in un grafo orientato rappresentante la storia di tutte le transazioni. Inoltre, il fruitore può controllare i nodi con il maggior punteggio di PageRank e localizzarli all'interno del grafo.

2.2 Architettura del progetto

Le principali scelte di progetto sono state prese coerentemente con lo stato dell'arte e si è tentato di non introdurre nuova complessità al panorama esistente, ricorrendo a tecnologie, protocolli e standard già esistenti ed affermati, senza definirne di nuovi. Quindi, l'architettura dovrà essere sufficientemente generale in modo da poter garantire

nuovi sviluppi ed evoluzioni future e da non comportare l'esclusione a priori di determinate soluzioni e tecnologie.

Il sistema, quindi, si divide logicamente in due moduli:

- **Il Sistema distribuito (back-end):** E' la parte non visibile all'utente. Si occupa del recupero dei dati dalla blockchain di bitcoin, della storicizzazione e della pubblicazione sui topic di kafka. Interamente scritto in Java, comprende Bitcoind, Spark, Hadoop, Neo4j, Zookeeper.
- **Webapp (front-end):** E' la parte visibile all'utente finale. Si occupa della rappresentazione grafica delle transazioni. Scritto in principalmente in Javascript, utilizza la potenza di NodeJS per creare l'interfaccia grafica. Integra nel proprio ecosistema Express Handlebars, WebSocket, MaterialCSS e D3js.

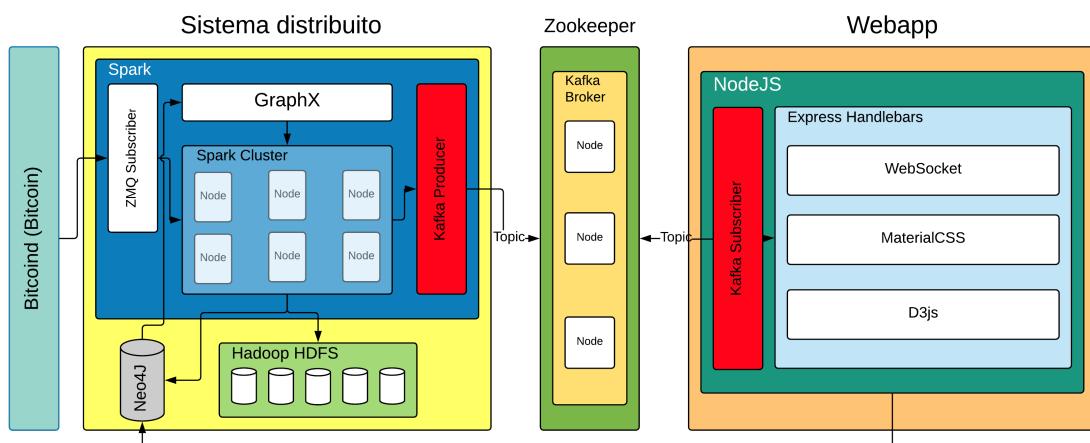


Figura 2.1: Architettura completa.

Per la costruzione di questa infrastruttura, la prima grande sfida, è stata quella di trovare un framework o un tool che permetesse di creare e programmare su di un sistema distribuito, senza complicarci la vita. Facendo ricerche sul web, la tecnologia che più si accostava meglio al mio problema è stata Apache Spark [2.3.2]. Questo strumento riesce a garantire a pieno i vincoli che ci siamo imposti, regalandoci la possibilità di creare un sistema distribuito su vari nodi impostando opportuni file di configurazione.

Risolto il problema infrastrutturale si è proceduto all'analisi dei singoli sottoproblemi. L'applicazione, per testare il carico di lavoro sul sistema, fa uso di blocchi grezzi provenienti dal software nativo del progetto Bitcoin: Bitcoind [2.3.1]. Bitcoind è un demone che invia blocchi o transazioni (a seconda di come lo si imposta) su di una coda di tipo publisher-subscriber tramite protocollo ZeroMQ [2.3.1.1]. I dati, quindi, sono prelevati da Bitcoind grazie all'implementazione di un connettore creato ad hoc.

Ottenuti i blocchi dalla coda, è nata l'esigenza di conservare i dati ottenuti così da poterli processare ed analizzare. Fortunatamente, Spark offre una nativa collaborazione con il FileSystem distribuito Hadoop [2.3.3], permettendomi di tenerli salvati su una memoria

di massa distribuita.

Oltre ad Hadoop, i dati sono stati immagazzinati in Neo4j [2.3.4]. Un database NoSQL che permette il salvataggio dei dati sottoforma di grafo, così da poter gestire facilmente i collegamenti tra le varie transazioni.

L'ultimo step, è stato quello di fare analisi delle transazioni, trovando i nodi con il maggior PageRank [2.3.2.2]. Anche in questo caso Spark è venuto in conto grazie al modulo GraphX, contenuto nel framework, il quale contiene algoritmi (come il PageRank) già sviluppati per l'analisi sui grafi.

Una volta che il sistema distribuito è completo, non resta che mostrare i risultati ottenuti. Le scelte nel campo del front-end sono migliaia ma per semplicità ed una forte attitudine ai sistemi real-time si è preferito usare NodeJS [2.4.1]. NodeJS ha dei moduli che permettono l'accesso a Kafka, il tramite tra la parte di back-end e front-end. Quindi, con NodeJS è stato creato un sito web consentendo agli utenti dal proprio browser, di visualizzare lo stato delle transazioni, i valori del PageRank e le transazioni che arrivano in real time.

2.3 Sistema distribuito

Come si accennava in precedenza, per lo sviluppo dell'applicazione è stato necessario ricorrere ad un sistema distribuito. Questo sistema, perno di tutta la trattazione, è invisibile all'utente ma di fondamentale importanza. Infatti, è colui che si occupa del processamento dei dati provenienti da Bitcoind, garantendo che nessun guasto influenza il risultato. Il sistema elabora su più macchine distribuite per ottimizzare i tempi di risposta. In particolare, utilizza un "cluster Spark" (insieme di macchine) per ridistribuire il carico di lavoro equamente sui diversi nodi. Così facendo, ha la possibilità di elaborare grandi quantità di dati in real-time. Inoltre, utilizza la tecnica della replicazione dei dati così che, in caso di rottura di uno dei nodi, si possano recuperare le informazioni perse.

Terminata l'elaborazione dei dati, il cluster invia i risultati ai sistemi dediti al salvataggio e alla pubblicazione. Due sono quelli dedicati alla storicizzazione permanente dei dati: Neo4j e Hadoop. Mentre, per quanto riguarda la pubblicazione, i dati saranno inviati a Kafka (tramite un producer) che li renderà disponibili per essere fruiti. La scelta dei framework Hadoop e Kafka, è data dal fatto che sono entrambi capaci di lavorare in ambienti distribuiti. Diversa invece è la scelta di Neo4j, l'utilizzo di tale sistema, infatti, è dovuto al fatto che si adatta al meglio con l'idea di transazione, salvando i dati come grafi.

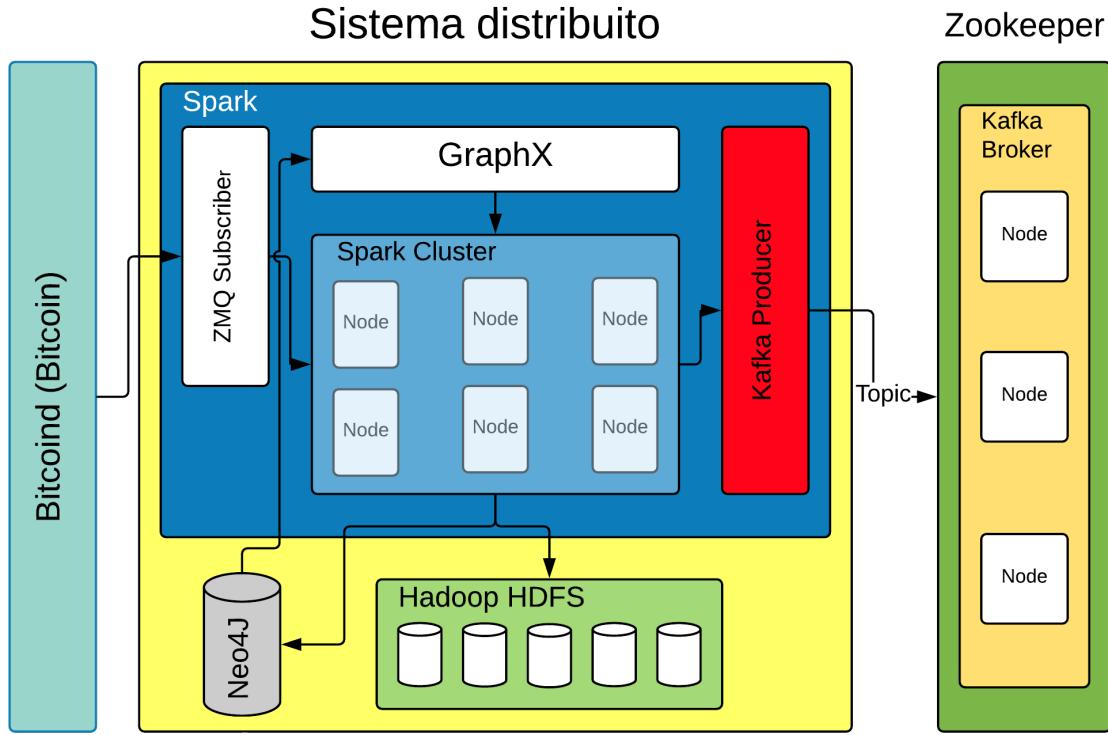


Figura 2.2: Architettura in dettaglio del sistema distribuito.

2.3.1 Bitcoind

Il primo componente del sistema distribuito ha il compito di fornire i dati da elaborare. Questa funzione è svolta dal demone Bitcoind.

Bitcoind, formalmente, è un software che implementa il protocollo Bitcoin per l'utilizzo delle remote procedure call (RPC). Esso è anche il secondo client Bitcoin nella storia del network [10]. Per sua natura, è eseguito come processo in background quindi l'utente per interagire con esso ha bisogno di farlo tramite una interfaccia da riga di comando chiamata *bitcoin-cli*. Il demone, inoltre, funge da nodo della rete Bitcoin, infatti si sincronizza con la blockchain, verifica le transazioni ed invia blocchi. Esiste una versione anche con interfaccia grafica del demone chiamata *Bitcoin-QT* o *Bitcoin Core*, ma per lo scopo dell'elaborato si è preferito utilizzare la versione lite per limitare l'utilizzo di risorse.

La versione demone, inoltre, ha il vantaggio di creare una coda ZeroMQ per la comunicazione con applicazioni esterne. Il sistema distribuito, utilizza questa funzione per recuperare i blocchi in formato grezzo (sequenza di byte) ogni qualvolta sono validati dalla blockchain.

2.3.1.1 ZeroMQ

ZeroMQ (anche conosciuto come ØMQ, 0MQ, o zmq) è una libreria di messaggistica asincrona ad alte prestazioni, destinata all'uso in applicazioni distribuite o concorrenti. Fornisce code di messaggi, ma a differenza dei middleware orientati ai messaggi, il sistema ZeroMQ può essere eseguito senza un broker di messaggi dedicato [14]. La libreria, inoltre, funziona molto bene grazie al suo modello interno di threading, che può superare le tradizionali applicazioni TCP in termini di throughput utilizzando una tecnica di batching automatico dei messaggi [30]. Il suo modello I/O asincrono offre la possibilità di creare applicazioni multicore scalabili, costruite come attività asincrone di elaborazione dei messaggi. Inoltre, la comunità di sviluppatori, ha creato una quantità enorme di wrapper per la libreria, così da renderla funzionante sulla maggior parte dei sistemi operativi. All'interno del sistema distribuito, infatti, è stato utilizzato il wrapper *jzmq* che permette l'utilizzo della libreria tramite Java API.

ZMQ, può avere diverse modalità di invio di messaggi atomici:

- Request-reply: Connnette un insieme di clienti ad un insieme di servizi. Questo è una Remote Procedure Call (RPC).
- Publish-subscribe: Connnette un insieme di produttori ad un insieme di consumatori.
- Push-pull (pipeline): Connnette i nodi in un pattern fan-out/fan-in che può avere più passaggi e cicli. Distribuisce in maniera parallela i messaggi.
- Exclusive pair: Collega due socket in maniera esclusiva.

Bitcoind utilizza una socket di tipo Publish-subscribe. Il publisher, in questo caso bitcoind, può spedire un messaggio a molti consumers attraverso un canale virtuale chiamato topic. Uno stesso topic può essere condiviso da più subscriber (client). Per ricevere i messaggi, i client devono "sottoscriversi" al topic creato da bitcoind. Ovviamente, questo implica che c'è relazione temporale tra i publisher e i subscribers, nel senso che, un client che si sottoscrive ad un topic può consumare solamente i messaggi pubblicati dopo la sua sottoscrizione. Qualsiasi messaggio spedito sul topic, viene consegnato a tutti i consumer sottoscritti, ciascuno dei quali riceve una copia identica di ciascun messaggio inviato. I messaggi quindi, vengono automaticamente inviati in broadcast ai consumer, senza che questi ne abbiano fatto esplicita richiesta.

Nel caso del sistema distribuito, il client che si sottoscrive al topic *pubrawblock* di bitcoind è il connettore-subscriber implementato all'interno di Spark.

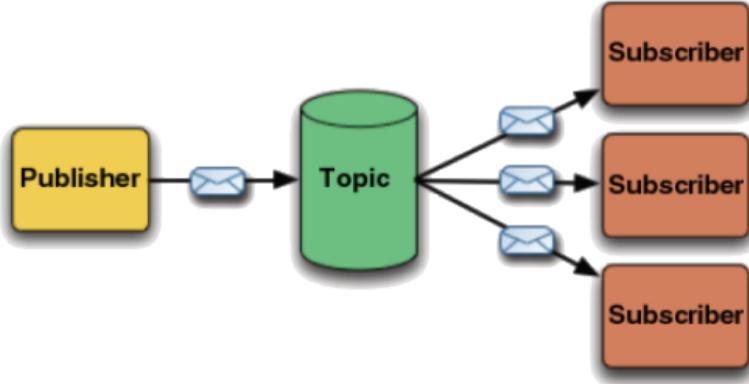


Figura 2.3: Messaggio inviato con la modalità Publish-Subscribe.

2.3.2 Apache Spark

Apache Spark è un framework per la computazione di grandi moli di dati su cluster, progettato e implementato nel 2010 da un gruppo di ricercatori dell’Università di Berkeley a San Francisco [27]. Viene scherzosamente descritto come "quello più veloce di Hadoop" perché nel confronto con il suo predecessore ha prestazioni 10-100 volte maggiori.

Questo progetto nasce dall’esigenza di migliorare le prestazioni dei sistemi distribuiti “MapReduce”. Ad alto livello infatti, un’applicazione Spark è formata da un driver program, che contiene la funzione main scritta dall’utente, e di una serie di parallel operation definite nel programma che verranno eseguite sui vari nodi worker che compongono il cluster. Fin qui nulla di nuovo, è il modello del calcolo distribuito master/slave. La vera innovazione è stata introdotta nel modo di definire i dati da elaborare.

Infatti Spark mette a disposizione un’astrazione molto potente, il *resilient distributed dataset* (RDD), che rappresenta una collezione di dati "immutabili" a cui il programmatore si può riferire direttamente tramite l’oggetto associato. Un RDD rappresenta un set di dati che è suddiviso in partizioni (Una tabella chiave-valore suddivisa in tante sottotabelle o un file diviso in tanti segmenti).

L’RDD realizza la fault tolerance grazie all’informazione di lineage: quando una partizione di un RDD si perde a causa di un guasto o di un altro errore, l’RDD ha tutte le informazioni riguardo la storia di quella partizione, in termini di operazioni effettuate su di esso che gli consentono di ricostruirla. La creazione, avviene a partire dai dati su disco (presi da HDFS) o da altre fonti di dati. Una volta creato, un RDD può restare in memoria centrale oppure può essere materializzato su disco.

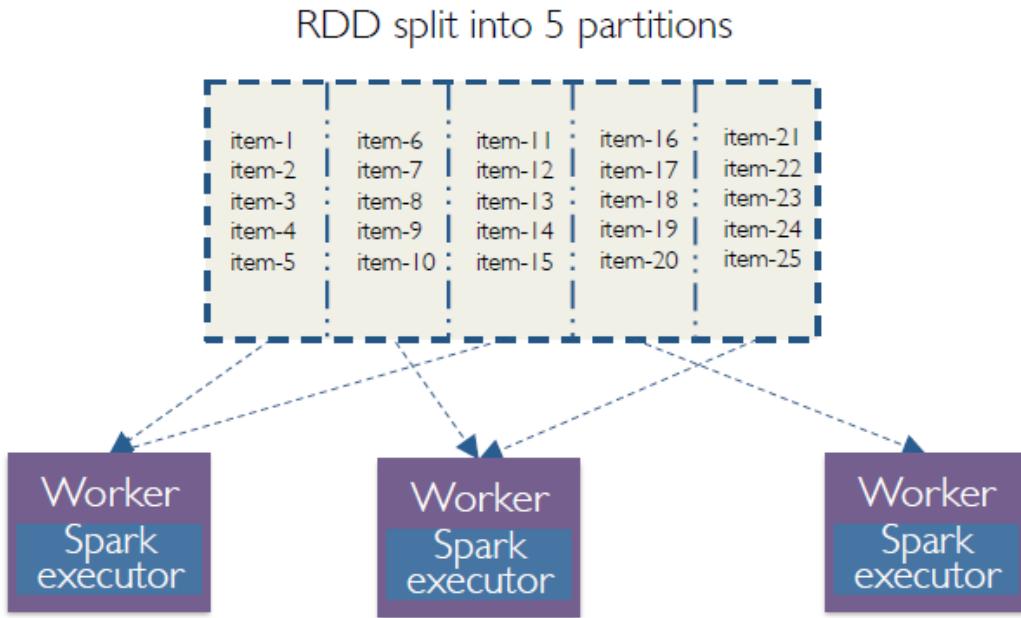


Figura 2.4: Visualizzazione di un RDD partizionato.

Spark nasce come un sistema per creare e gestire job di analisi basati su trasformazioni di RDD. Dato che gli RDD nascono e vivono in memoria, l'esecuzione di lavori iterativi, o che trasformano più volte un set di dati, sono immensamente più rapide di una sequenza di MapReduce; questo perchè il disco non viene mai (o quasi mai) impiegato nell'elaborazione.

Il vantaggio principale dell'utilizzo di Spark è la sua estrema velocità nell'eseguire programmi di elaborazione dati. Il motivo di queste prestazioni risiedono in una miglior gestione della memoria; aspetto che lo diversifica da Hadoop. Prendendo come esempio una generica elaborazione fatta con il paradigma MapReduce, in Hadoop questa produce il seguente schema di lavoro, che può essere iterato più volte (semplificando):

- Load dei dati da disco locale verso i nodi worker del cluster;
- Esecuzione della funzione assegnata;
- Store dei dati su disco locale.

Le ripetute fasi di load/store rendono il sistema complessivamente lento. Spark, invece, cerca di mantenere in memoria i dati, esegue le operazioni di trasformazione e solo alla fine memorizza i dati sul disco.

Come detto in precedenza, Spark non utilizza MapReduce come motore di esecuzione; invece, utilizza il proprio runtime distribuito (DAG) per l'esecuzione di jobs su un cluster. Quando viene invocata un'azione su un RDD, viene creato un “job”. Un Directed Acyclic Graph o DAG è un grafo aciclico in cui ogni nodo è una partizione di RDD e ogni vertice è

una trasformazione. A differenza di MapReduce, il motore DAG di Spark può processare pipeline arbitrarie di operatori e tradurle in un unico “job” per l’utente.

Spark, infine, sta dimostrando di essere una buona piattaforma su cui costruire strumenti di analisi, infatti ha moduli per il Machine learning (MLlib), Elaborazione grafica (GraphX), Elaborazione di stream (Spark Streaming) ed SQL (Spark SQL) [27].

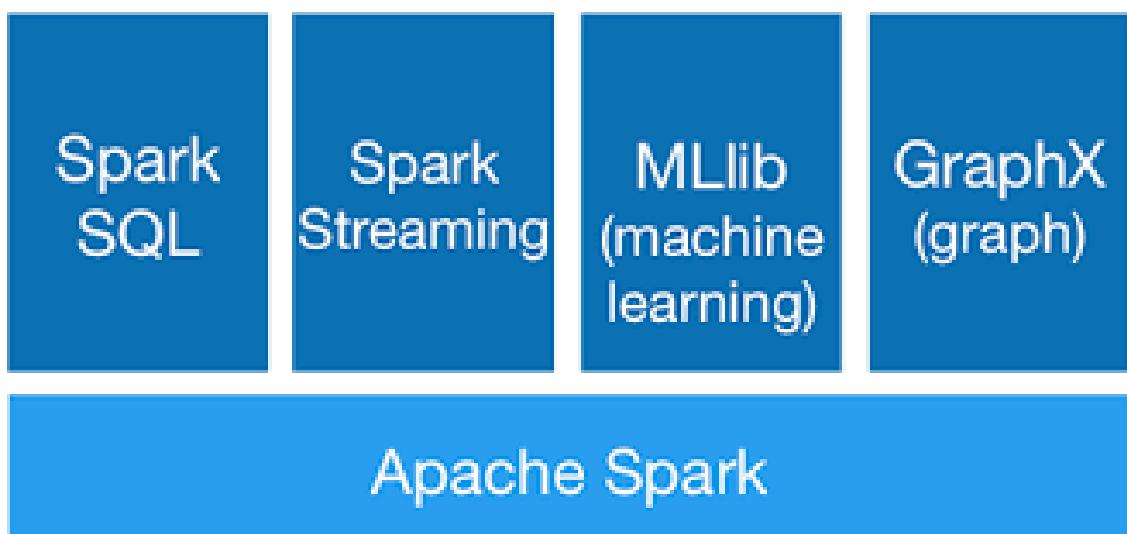


Figura 2.5: Infrastruttura Spark.

2.3.2.1 Spark Streaming

Nel sistema distribuito, poichè c’è l’esigenza di recuperare i dati in real time dalla coda di bitcoind, viene utilizzato Spark Streaming. Questo modulo è un’estensione dell’API Spark di base che consente l’elaborazione streaming, scalabile, ad alto throughput e con tolleranza agli errori dei flussi di dati in tempo reale. I dati, che possono provenire da diverse fonti, sono elaborati utilizzando algoritmi complessi espressi con funzioni di alto livello come *map*, *reduce*, *join* e *window*. I dati processati, infine possono essere inviati a filesystem (Hadoop) o database (Neo4j) per il salvataggio oppure ad altri moduli di Spark, dediti all’analisi, tipo machine learning (MLlib) o graph processing (GraphX).

Internamente, Spark Streaming riceve streams di dati di input e li divide in batch, che vengono quindi elaborati dal motore Spark per generare il flusso finale di risultati. Per consentire il facile utilizzo di questi dati, fornisce un’astrazione di alto livello chiamata stream discretizzato o DStream, che rappresenta un flusso continuo di dati. E’ possibile creare Dstreams da flussi di dati di input da sorgenti come Kafka, Flume e Kinesis o applicando operazioni di alto livello su altri Dstreams [13]. Internamente, un Dstream è rappresentato come una sequenza di RDD sulla quale possono essere effettuate le operazioni descritte in precedenza. Infatti, i Dstream possono essere trasformati usando le operazioni di trasformazione simili a quelle dei RDD, come map e filter.



Figura 2.6: Come vengono gestiti i dati in Spark Streaming.

2.3.2.2 GraphX

La diffusione dei grafi nei sistemi informatici ha portato a un grande lavoro di analisi su di essi. Facendone un utilizzo sempre più frequente, ci si trova a dover fare ricerche, interrogazioni e misure su questa struttura dati e di trovare un modo per memorizzare efficientemente questi oggetti. Anche Spark si è occupato del problema e ha reso disponibile uno strumento, basato sul framework principale, che ottimizza la gestione dei grafi e consente di applicare ad essi funzioni e metodi in modo molto intuitivo. Si tratta del progetto GraphX. Questo modulo è usato nel sistema distribuito, per fare analisi dei dati provenienti dall'elaborazione di Spark.

GraphX è un nuovo componente di Apache Spark per grafi ed il calcolo parallelo su di essi. Spark ha introdotto l'RDD, un'astrazione comoda per memorizzare i dati e risparmiando al programmatore parecchio lavoro. GraphX estende il concetto di RDD introducendo il *Resilient Distributed Graph* (RDG). Inoltre, per aiutare nell'analisi, espone un insieme di operatori fondamentali (sottografo, joinVertices e aggregateMessages) come variante ottimizzata dell'API Pregel. In più, Graphx include una crescente collezione di algoritmi e costrutti per grafi per semplificare le attività di analisi [19]. Attualmente le sue API sono scritte in Scala, Java e Python.

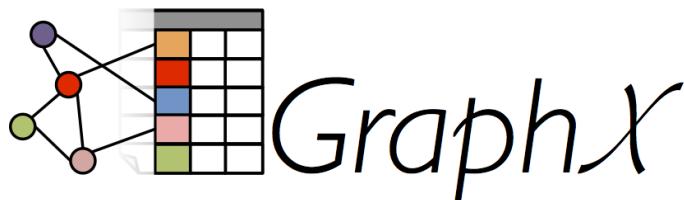


Figura 2.7: Logo GraphX.

GraphX, come detto in precedenza, gestisce i dati in memoria come se fossero grafi. Infatti, utilizza archi e vertici che hanno delle proprietà. Ogni vertice possiede un identificativo univoco a 64bit (VertexID), mentre, allo stesso modo, gli archi contengono gli identificativi di origine e partenza. Queste proprietà sono definite dal programmatore e tenuti in memoria come oggetti. Su di essi si possono eseguire metodi per l'analisi già inclusi nella libreria di GraphX come Connected components, Triangle Counting, PageRank, etc. Nell'elaborato di tesi, per trovare i nodi con maggior importanza all'interno

del grafo, è stato usato l'algoritmo *PageRank*.

Il PageRank è un algoritmo di analisi che assegna un peso numerico ad ogni elemento di un collegamento ipertestuale di un insieme di documenti, come ad esempio il World Wide Web, con lo scopo di quantificare la sua importanza relativa all'interno della serie. L'algoritmo di PageRank è stato brevettato (brevetto US 6285999) dalla Stanford University; è inoltre un termine ormai entrato di fatto nel lessico dei fruitori dei servizi offerti dai motori di ricerca. Il nome PageRank è un marchio di Google ed il suo nome si deve a Larry Page [29], uno dei due fondatori di quell'azienda [16].

L'algoritmo completo per il calcolo del PageRank fa ricorso all'uso della teoria dei processi Markov ed è classificato nella vera categoria degli algoritmi di Link Analysis Ranking. Dalla formula inizialmente sviluppata dai fondatori di Google, Sergey Brin e Larry Page, è possibile comprendere come il PageRank viene distribuito tra le pagine:

$$PR[A] = \frac{1-d}{N} + d \left(\sum_{K=1}^n \frac{PR[P_k]}{C[P_k]} \right)$$

Dove:

- **PR[A]** è il valore di PageRank della pagina A che vogliamo calcolare.
- **N** è il numero totale di pagine note.
- **n** è il numero di pagine che contengono almeno un link verso A. **Pk** rappresenta ognuna di tali pagine.
- **PR[Pk]** sono i valori di PageRank di ogni pagina **Pk**.
- **C[Pk]** sono il numero complessivo di link contenuti nella pagina che offre il link.
- **d (damping factor)** è un fattore deciso da Google e che nella documentazione originale assume valore 0,85. Può essere aggiustato da Google per decidere la percentuale di PageRank che deve transitare da una pagina all'altra e il valore di PageRank minimo attribuito ad ogni pagina in archivio.

Dalla formula si nota quindi che all'aumentare del numero di link complessivi dei siti che puntano ad A il PageRank aumenta.

2.3.3 Hadoop HDFS

Apache Hadoop è un framework open-source che supporta applicazioni distribuite con elevato accesso ai dati, è uno dei primi sistemi per l'analisi di Big Data, ed è stato considerato un modello da seguire per tutti i sistemi che sono stati creati successivamente. Apache Hadoop è stato ideato per la memorizzazione e la gestione di grandi quantità di dati in parallelo, su cluster di grandi dimensioni (costituiti da migliaia di nodi) assicurando un'elevata affidabilità, scalabilità e disponibilità (fault-tolerant). Tutti i moduli, infatti, sono progettati in modo tale che il software del framework gestisca automaticamente gli eventuali “down” dell'hardware, molto frequenti in un sistema parallelo.

Hadoop nacque per sopperire a un grave problema di scalabilità di Nutch, un crawler

open source basato sulla piattaforma Lucene di Apache. I programmatore Doug Cutting e Michael J. Cafarella hanno lavorato a una versione iniziale di Hadoop a partire dal 2004; proprio in quell'anno furono pubblicati documenti tecnici riguardanti il Google File System e Google MapReduce, documenti da cui Doug e Michael attinsero le competenze fondamentali per lo sviluppo di HDFS e di un nuovo e innovativo pattern per l'elaborazione distribuita di elevate moli di dati, MapReduce, che oggi rappresenta uno dei componenti fondamentali di Hadoop. Circa quattro anni più tardi, nel 2008, nacque la prima release come progetto Open Source indipendente di Apache. A oggi Hadoop è un insieme di progetti tutti facenti parte della stessa infrastruttura di calcolo distribuito. Hadoop offre librerie che permettono la suddivisione dei dati da elaborare direttamente sui nodi di calcolo e permette di ridurre al minimo i tempi di accesso, questo perché i dati sono immediatamente disponibili alle procedure senza pesanti trasferimenti in rete. Il framework garantisce inoltre un'elevata affidabilità: le anomalie e tutti gli eventuali problemi del sistema sono gestiti a livello applicativo anziché utilizzare sistemi hardware per garantire alta disponibilità. Un'altra caratteristica di Hadoop è la scalabilità che è realizzabile semplicemente aggiungendo nodi al cluster in esercizio. I principali vantaggi di Hadoop risiedono nelle sue caratteristiche di agilità e di flessibilità. L'architettura base del framework Hadoop, si compone dei seguenti elementi principali:

- **HDFS:** Il filesystem distribuito di Hadoop, nella quale vengono fisicamente salvati i file.
- **MapReduce:** Un framework per la creazione di applicazioni in grado di elaborare grandi quantità di dati in parallelo basandosi sul concetto di functional programming, ed è, quindi, il principale responsabile del processo di calcolo.
- **Yarn:** Si occupa della schedulazione dei task, ossia delle sotto-attività che compongono le procedure map e reduce eseguite nel processo di calcolo.

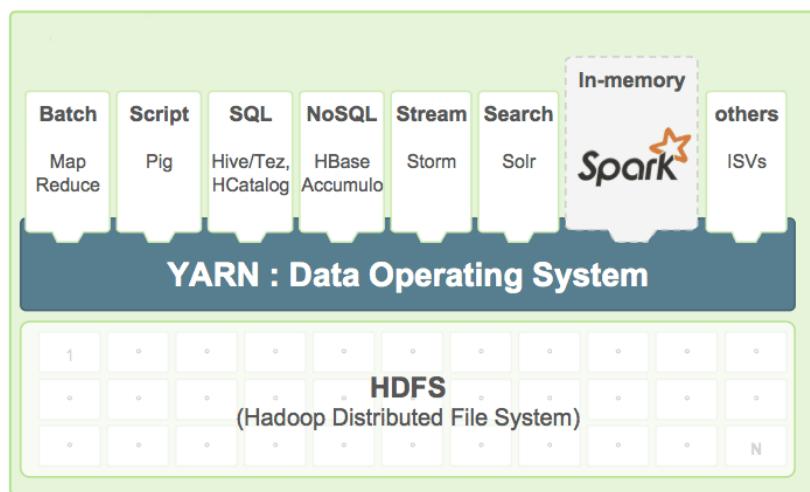


Figura 2.8: Visualizzazione invio dati ad HDFS.

All'interno del sistema distribuito, per il salvataggio veloce e distribuito dei dati su dispositivi fisici, viene utilizzato l'HDFS di Hadoop.

L'HDFS (Hadoop Distributed File System), come riportato nella documentazione ufficiale, è il file system distribuito di Hadoop, progettato appositamente per immagazzinare un'enorme quantità di dati (dell'ordine dei Gigabyte e Terabyte), in modo da ottimizzare le operazioni di archiviazione e accesso a un ristretto numero di file di grandi dimensioni, a differenza dei tradizionali file system che sono ottimizzati per gestire numerosi file di piccole dimensioni; infatti, quando la mole dei dati diventa non più gestibile da una singola macchina, si rende necessario partizionare gli stessi su un certo numero di macchine separate, interconnesse da una rete (cluster), rendendo il filesystem di fatto distribuito. HDFS presenta i file organizzati in una struttura gerarchica di cartelle. Dal punto di vista dell'architettura, un cluster è costituito dai seguenti tipi di nodi:

- **NameNode:** è l'applicazione che gira sul server principale. Gestisce il file system e in particolare il namespace. Inoltre, determina come i blocchi dati siano distribuiti sui nodi del cluster e la strategia di replicazione che garantisce l'affidabilità del sistema. Il NameNode monitora anche che i singoli nodi siano in esecuzione senza problemi e in caso contrario decide come riallocare i blocchi.
- **DataNode:** applicazione/i che girano su altri nodi del cluster, generalmente una per nodo, e gestiscono fisicamente lo storage di ciascun nodo. Queste applicazioni eseguono, logicamente, le operazioni di lettura e scrittura richieste dai client e gestiscono fisicamente la creazione, la cancellazione o la replica dei blocchi dati.
- **SecondaryNameNode:** si tratta di un servizio che aiuta il NameNode a essere più efficiente
- **BackupNode:** è il nodo di failover e consente di avere un nodo simile al SecondaryNameNode sempre sincronizzato con il NameNode.

In HDFS le richieste di lettura dati seguono una politica relativamente semplice [15] avvengono scegliendo i nodi più vicini al client che effettua la lettura e, ovviamente, in presenza di dati ridondanti, risulta più semplice soddisfare questo requisito. Inoltre, occorre precisare che la creazione di un file non avviene direttamente attraverso il NameNode. Infatti, il client HDFS crea un file temporaneo in locale e solo quando tale file supera la dimensione di un blocco, è preso in carico dal NameNode. Quest'ultimo crea il file all'interno della gerarchia del file system, identifica un DataNode e i blocchi su cui posizionare i dati. Successivamente DataNode e blocchi sono comunicati al client HDFS che provvede a copiare i dati dalla cache locale alla sua destinazione finale.

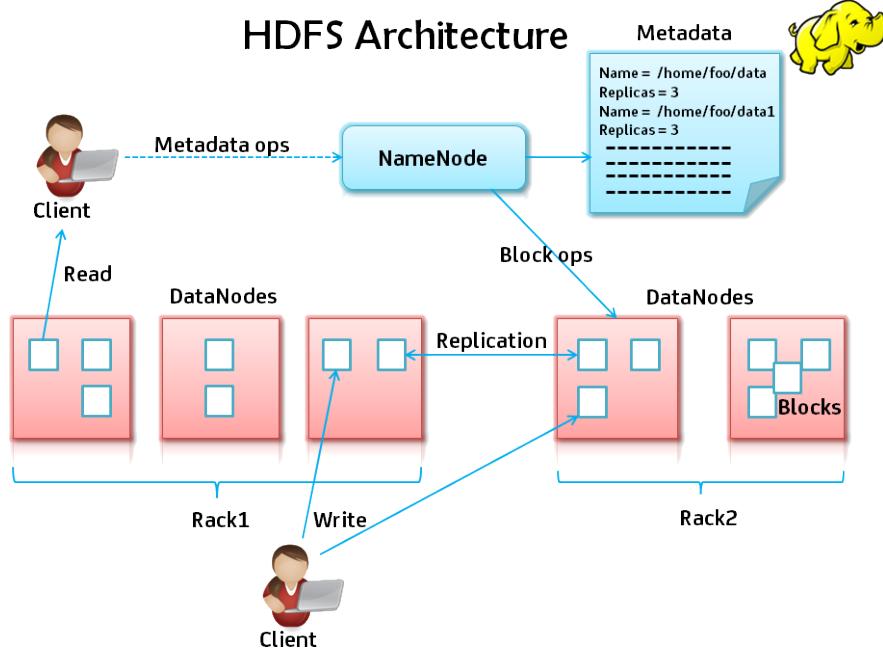


Figura 2.9: Architettura HDFS.

Quanto detto fino a ora, ci permette di concludere che quando vengono trattati file di grandi dimensioni, HDFS è molto efficiente. Invece con file di piccole dimensioni, dove per piccole dimensioni s'intendono dimensioni inferiori al blocco, il trattamento dei file è molto inefficiente, questo perché i file utilizzano spazio all'interno del namespace, cioè l'elenco dei file mantenuti dal NameNode, che ha un limite dato dalla memoria del server che ospita il NameNode stesso.

2.3.4 Neo4j

Negli ultimi anni è incredibilmente aumentata la popolarità delle tecnologie di immagazzinamento di informazioni conosciute con il nome di *NoSQL*, acronimo che sta per Not only SQL [8]. I NoSQL *Database Management System (DBMS)* sono sistemi software che consentono di immagazzinare e organizzare i dati senza fare affidamento sul modello relazionale, solitamente impiegato da database tradizionali.

I NoSQL DBMS sono inoltre contraddistinti dal fatto che non utilizzano un sistema transazionale ACID e spesso sono schema-less [8], ovvero non possiedono uno schema fisso a cui devono attenersi, evitando spesso così le operazioni di join.

Le strutture dei dati memorizzati nei sistemi NoSQL (ad esempio documenti, grafi o coppie chiave/valore) rendono alcune operazioni più veloci rispetto a generici database relazionali. Si utilizzano infatti diversi tipi di database NoSQL in base al problema da risolvere. Le tipologie di database NoSQL più utilizzate sono:

- **Key/Value Store:** rappresentano una tipologia di database NoSql che si basa sul concetto di *associative array*, implementati attraverso HashMap. In un array associativo si hanno un insieme di record che possono essere identificati sulla base

di una chiave univoca. La tipologia di memorizzazione adottata dai key/value stores garantisce tempi di esecuzione costanti per tutte le operazioni applicabili sui dati: *add*, *remove*, *modify* e *find*.

- **Document-oriented database:** questo tipo di database permette di memorizzare in maniera efficiente dati semistrutturati. Essi possono essere considerati una specializzazione dei database key/value, nei quali vengono permesse strutture innestate. Sono spesso usati i formati JSON o XML per la memorizzazione dei dati. MongoDB è il più diffuso database orientato ai documenti ed è uno dei DBMS NoSQL più utilizzati.
- **Column-oriented database:** i DBMS colonnaari, a differenza dei tradizionali RDBMS, che memorizzano i dati riga per riga, sfruttano la memorizzazione dei dati per colonna. Per ogni colonna si memorizzano coppie chiave/valore, dove la chiave è l'identificativo di riga ed il valore è il valore associato a quella colonna per la specifica riga. Questa rappresentazione permette di risparmiare una notevole quantità di spazio in caso di sparsità dei dati. I DBMS di tipo colonnaire più diffusi sono HBase e Cassandra.
- **Graph database:** un database a grafo utilizza un insieme di nodi con un insieme di archi che li connettono per memorizzare le informazioni, in cui le “relazioni” vengono rappresentate come grafi. Le strutture a grafo si prestano molto bene per la rappresentazione di determinati dati semistrutturati e altamente interconnessi come, ad esempio, i dati dei social network e del Web. I database a grafi più diffusi sono GraphDB, OrientDB e Neo4j.

La struttura dati che più si avvicina all’idea di transazione è quella del grafo, con nodi che rappresentano i mittenti/destinatari dei bitcoin e gli archi che identificano l’ammontare della transazione. Per questo motivo, nel sistema distribuito, si è preferito fare utilizzo del database NoSQL Neo4j per il salvataggio. Inoltre, le performance dei Graph DBMS tendono ad essere ottimali quando i dati da archiviare sono altamente connessi e la mole del dataset è estremamente grande. Questo perché al contrario degli RDBMS (Relational Database Management System), la loro natura gli consente di evitare le onerose operazioni di join semplicemente attraversando le relazioni che connettono i nodi.

Neo4j, dunque, è un software per basi di dati a grafo open source sviluppato interamente in Java. È un database totalmente transazionale, che viene integrato nelle applicazioni permettendone il funzionamento stand alone e memorizza tutti i dati in una cartella. È stato sviluppato dalla Neo Technology, una startup di Malmö, Svezia e della San Francisco Bay Area [6]. Il database può essere usato sia in modalità embedded che server. Nella modalità embedded si incorpora il database nell’applicazione (con maven o includendo i file JAR) e questo viene eseguito all’interno della JVM, quindi nello stesso processo ma accettando vari thread concorrenti. Nella modalità server invece il database è un processo a sé stante a cui si accede tramite REST facendo delle query e ricevendo i dati in remoto. È robusto, scalabile e ad alte prestazioni. È dotato di:

- Transazioni ACID.

- High Availability.
- Può memorizzare miliardi di nodi e relazioni.
- Alta velocità di interrogazione tramite traversamenti.
- Linguaggio di interrogazione dichiarativo e grafico chiamato Cypher
- DBMS schema-less, ciò sta a significare che i suoi dati non devono attenersi a alcuna struttura di riferimento prefissata.

La index-free adjancency è alla base delle sue alte prestazioni di attraversamento, d'interrogazione e di scrittura, ed è uno degli aspetti chiave della sua architettura. L'index-free adjancency è una lista (o tabella), ove ogni suo elemento è composto da un nodo del grafo e dai puntatori ai nodi connessi ad esso.

Neo4j, inoltre, salva i dati dentro di una serie di store file, contenuti all'interno di un'unica cartella. Ognuno di questi file contiene al suo interno le informazioni relative ad una singola parte del grafo (e.g. nodi, relazioni, proprietà). Questa separazione della struttura del grafo facilita il suo attraversamento. In Neo4j le unità fondamentali che compongono un grafo, dunque, sono i nodi e le relazioni. I nodi vengono solitamente impegnati per rappresentare le entità, ma a seconda della sfera delle relazioni possono essere utilizzati per scopi differenti. A parte proprietà e relazioni, i nodi possono anche essere etichettati con zero o più Label. Le relazioni tra i nodi, invece, sono una parte chiave dei database a grafo. Ci permettono di trovare le informazioni connesse. Come per i nodi, le relazioni possono avere le proprietà, hanno sempre una direzione e possiedono sempre un nodo di partenza e uno di arrivo. Infine, una label è un *named graph construct*, viene usata per raggruppare i nodi in sottoinsiemi; tutti i nodi etichettati con la stessa label fanno parte dello stesso insieme.

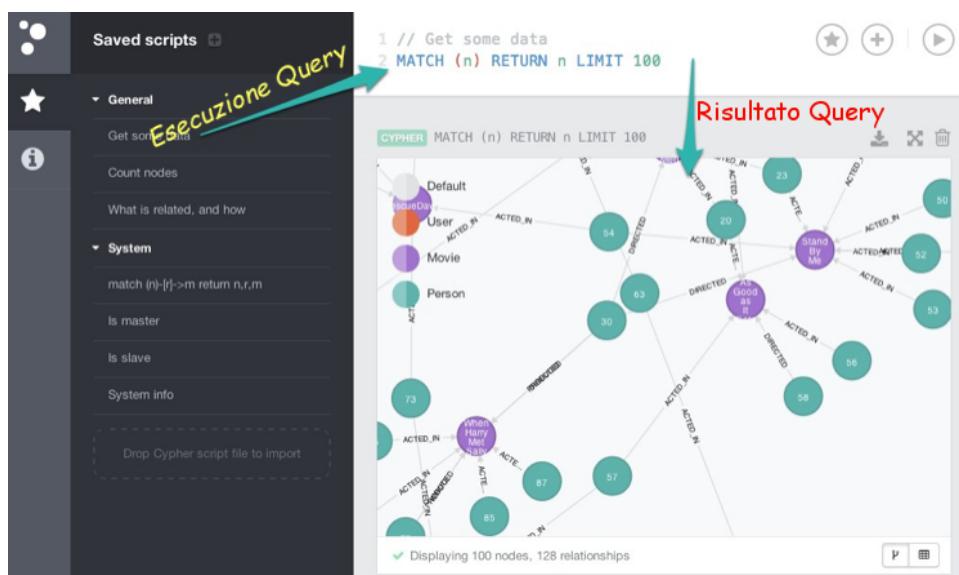


Figura 2.10: Interfaccia web di Neo4j nella quale è eseguita una query.

Tra i vantaggi che possiamo ottenere nell'utilizzo di Neo4j troviamo:

- **Distribuito:** Qualora si voglia garantire che il proprio sistema sia in grado di fornire un servizio di erogazione dei dati continuo, senza failure point e in grado di bilanciare e gestire un'enorme mole di richieste, Neo4j High Available (HA) è la risposta a questa esigenza. Neo4j HA, infatti, è stato progettato per rendere semplici, le transazioni da una singola macchina ad un'una macchina multipla, senza dover cambiare la tipologia delle istanze che andranno a comporre il cluster.
- **REST API:** Il server è munito di una ricca REST API che permettono ai client di spedire richieste in formato JSON per mezzo del protocollo HTTP. Le risposte vengono restituite all'interno di documenti JSON arricchiti con Hypermedia Links che mettono in risalto ulteriori caratteristiche del dataset. Sono molteplici le funzionalità messe a disposizione dalla REST API, ma il suo più grande vantaggio è quello di potervi accendere per mezzo di una semplice applicazione browser, come Firefox, Chrome o Internet Explorer.
- **Platform Independence:** Dato che le informazioni contenute nel server vengono accedute per mezzo di documenti JSON spediti attraverso l'HTTP, le applicazioni client possono essere costruite su qualsiasi tipo di piattaforma, basta possedere delle librerie client HTTP.
- **Scaling Independence:** Quando Neo4j viene eseguito in modalità server possiamo aumentare o diminuire il numero di componenti del cluster indipendente dal tipo di applicazione.
- **Per-request transactional:** Ogni richiesta da parte del client viene eseguita come una singola transazione, atomicamente separata dalle altre. Tuttavia, la REST API fornisce un supporto per l'esecuzione di operazioni in batch (ovvero l'esecuzione "accorpata" delle operazioni).

Neo4j, infine utilizza un sistema di query dichiarativo e grafico che prende il nome di Cypher. Esso è un linguaggio grafico, ovvero si basa sulla riproduzione grafica del sotto-grafo che si vuole estrarre. Con le query è possibile creare, modificare, eliminare e interrogare i dati del database. Il sotto-grafo riprodotto nelle query viene chiamato pattern, e per produrlo non servono strumenti particolari, ma basta seguire delle semplici regole che permettono di disegnarlo impiegando i caratteri ASCII (i caratteri presenti sulla tastiera).

Cypher permette, quindi, agli utenti (o ad una applicazione che agisce per conto dell'utente) di interrogare il database cercando i dati che corrispondono ad una specifica struttura. In termini da profano, chiediamo al database di *"cercare tutti quegli oggetti simili o che assomigliano"* ad un certo pattern.

```
1 Match (a: NODE_LABEL) -[r: RELATION_LABEL] -> (b: NODE_LABEL)
2 RETURN a, b, r;
```

Listato 2.1: Query Cypher.

Questo è un esempio di query scritta col linguaggio Cypher. Questo pattern descrive un path (percorso), che connette (a) a (b) ritornando sia i nodi che le relazioni che soddisfano la query.

2.3.5 Zookeeper

ZooKeeper è un servizio, centralizzato e open source, di coordinamento affidabile per applicazioni distribuite [12]. I servizi di coordinamento sono notoriamente difficili da ottenere; sono, infatti, particolarmente inclini a errori derivanti, ad esempio, da condizioni di stallo. ZooKeeper, invece, fornisce servizi operativi per cluster di grandi dimensioni. Esempi di questi servizi includono: informazioni di configurazione distribuita, un servizio di sincronizzazione e un registro di denominazione per i sistemi distribuiti. Le applicazioni sfruttano questi servizi per coordinare l'elaborazione distribuita tra cluster di grandi dimensioni. ZooKeeper inoltre, mira a semplificare la natura di questi diversi servizi in un'interfaccia molto semplice creando un servizio di coordinamento centralizzato. Il servizio stesso è distribuito e altamente affidabile. Questo perché, questo tipo di servizi risulta essere di difficile implementazione per le applicazioni distribuite.

Lo spazio dei nomi è costituito da registri di dati, chiamati znodes, in linguaggio ZooKeeper, che sono simili ai file e alle directory. A differenza di un tipico file system, progettato per l'archiviazione, i dati di ZooKeeper vengono conservati in memoria, il che significa che ZooKeeper può ottenere alti throughput e bassi numeri di latenza. Il database replicato di ZooKeeper comprende un albero di znode, entità che rappresentano approssimativamente i nodi del file system (simili a directory). Ogni znode può essere arricchito da una matrice di byte, che memorizza i dati. Inoltre, ogni znode può avere altri znode sotto di esso, praticamente formando un sistema di directory interno.

Zookeeper è considerato un servizio robusto, dal momento che i dati persistenti sono distribuiti tra più nodi (questo insieme di nodi è chiamato "ensemble") e un client si connette a uno di essi (cioè un "server" specifico), migrando se un nodo fallisce; Finché funziona la stragrande maggioranza dei nodi, l'insieme dei nodi ZooKeeper è vivo. In particolare, un nodo master viene scelto dinamicamente per consenso all'interno dell'ensemble; se il nodo principale non funziona, il ruolo del master passa a un altro nodo. È interessante notare che il nome di uno znode può essere sequenziale, il che significa che il nome che il client fornisce quando si crea lo znode è solo un prefisso: il nome completo è dato anche da un numero sequenziale scelto dall'ensemble. Ciò è utile, ad esempio, per scopi di sincronizzazione: se più client desiderano ottenere un blocco su una risorsa, possono contemporaneamente creare uno znode sequenziale su una posizione: chi ottiene il numero più basso ha diritto al blocco.

Inoltre, uno znode può essere “effimero” (temporaneo): questo significa che viene distrutto non appena il client che lo ha creato si disconnette. Ciò è utile soprattutto per sapere quando un client fallisce, il che può essere rilevante quando il client stesso ha delle responsabilità che dovrebbero essere prese da un nuovo client. Tutte le richieste di scrittura dei client vengono inoltrate a un singolo server, chiamato leader: in questo modo è possibile garantire che le scritture siano mantenute in ordine, vale a dire che le scritture siano lineari. Il resto dei server ZooKeeper, chiamati follower, riceve proposte di messaggi dal leader e concorda la consegna dei messaggi. Il livello di messaggistica si occupa di sostituire i

leader in caso di fallimento e di sincronizzare i follower con il leader. ZooKeeper utilizza un protocollo di messaggistica atomica personalizzato. Poiché il livello di messaggistica è atomico, ZooKeeper può garantire che le repliche locali non divergano mai. Quando il leader riceve una richiesta di scrittura, calcola lo stato del sistema quando deve essere applicata la scrittura e lo trasforma in una transazione che cattura questo nuovo stato. Ogni volta che un client scrive sull'ensemble, la maggior parte dei nodi mantiene l'informazione: questi nodi includono il server per il client e ovviamente il leader. Ciò significa che ogni scrittura rende il server aggiornato con il leader. Significa anche, tuttavia, che non è possibile avere scritture simultanee. La garanzia delle scritture lineari è la ragione del fatto che ZooKeeper non offre prestazioni ottimali per i carichi di lavoro dominanti dalla scrittura. In particolare, non dovrebbe essere usato per l'interscambio di grandi dati, come i media. Fintanto che la comunicazione coinvolga dati condivisi, ZooKeeper è utile. Quando i dati possono essere scritti contemporaneamente, ZooKeeper interviene, perché impone un rigoroso ordine di operazioni anche se non strettamente necessario dal punto di vista di chi scrive. Per la lettura, invece, ZooKeeper eccelle, questo perché le letture sono simultanee poiché sono servite dallo specifico server al quale il client si connette.

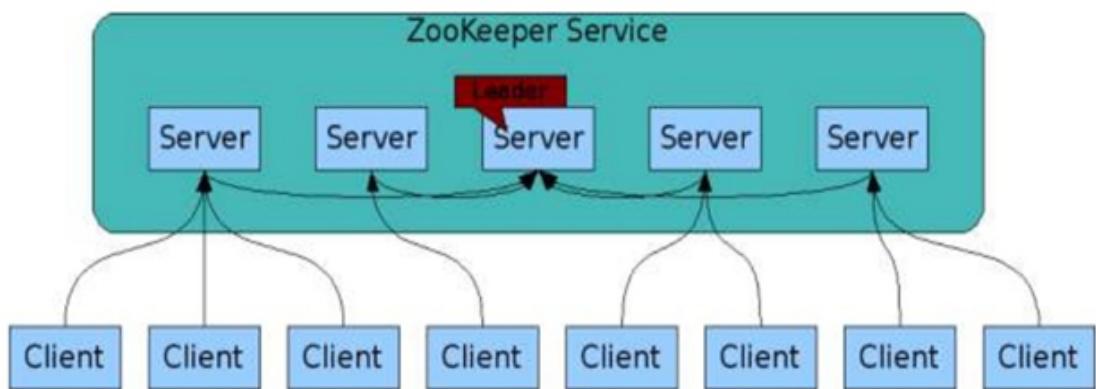


Figura 2.11: Funzionamento di Zookeeper.

ZooKeeper, in sintesi, è servizio molto veloce e molto semplice. La velocità di Zookeeper è data da carichi di lavoro in cui le letture dei dati sono più comuni delle scritture. Il rapporto di lettura / scrittura ideale è di circa 10:1. Inoltre, Zookeeper mantiene uno spazio dei nomi gerarchico standard, simile a file e directory, questo lo rende un sistema sostanzialmente semplice. ZooKeeper viene replicato su un insieme di host (chiamato “ensemble”) e i server sono a conoscenza l’uno dell’altro. Finché sarà disponibile una massa critica di server, sarà disponibile anche il servizio ZooKeeper. Non c’è un singolo punto di errore. Per questo Zookeeper è detto: affidabile.

Dal momento che il suo obiettivo, però, è quello di essere una base per la costruzione di servizi più complicati, come la sincronizzazione, fornisce un insieme di garanzie tra cui:

- *Consistenza sequenziale*: gli aggiornamenti da un client verranno applicati nell’ordine in cui sono stati inviati.

- *Atomicità*: gli aggiornamenti hanno esito positivo o negativo. Nessun risultato parziale.
- *Immagine del sistema singolo*: un client vedrà la stessa vista del servizio indipendentemente dal server a cui si connette.
- *Affidabilità*: una volta applicato un aggiornamento, esso persisterà da quel momento fino a quando un client non sovrascriverà l'aggiornamento.
- *Tempestività*: la visualizzazione dei client del sistema è garantita per essere aggiornata entro un certo limite di tempo.

2.3.5.1 Kafka

Apache Kafka è una piattaforma di streaming open source distribuita che può essere utilizzata per compilare applicazioni e pipeline di dati in streaming in tempo reale. Kafka offre anche una funzionalità di broker di messaggi simile a una coda di messaggi, dove è possibile pubblicare e sottoscrivere flussi dei dati denominati [20]. Apache Kafka permette la gestione di centinaia di megabyte di traffico in lettura e scrittura al secondo da parte migliaia di Client. È stato sviluppato per la prima volta su LinkedIn e viene spesso utilizzato in combinazione con Apache Spark Streaming per l'elaborazione dei flussi in tempo reale.

Kafka ha un'architettura che differisce significativamente da altri sistemi di messaggistica. Ogni nodo prende il nome di broker. Kafka offre un'API Producer per la pubblicazione di record in un topic Kafka e un'API Consumer che viene utilizzata quando si sottoscrive un topic. Un topic è un nome di categoria o feed a cui i record sono pubblicati. I Topic in Kafka sono sempre multi-abbonato; cioè, un argomento può avere zero, uno o più consumatori che si abbonano ai dati scritti su di esso [22]. Kafka archivia i record in topic. Un topic è un insieme di messaggi della stessa categoria; per ogni topic il cluster Kafka mantiene un registro partizionato. Ogni partizione è una sequenza ordinata ed immutabile di messaggi che vengono aggiunti in continuazione all'interno del registro. I record, che come abbiamo visto sono archiviati in topic, vengono prodotti da un producer e usati dai consumer. I Producers o produttori pubblicano i messaggi (i dati) all'interno di un topic. Il Producer è responsabile di scegliere in quale partizione del registro del topic inserire un proprio messaggio. Ogni Producer sceglie il proprio algoritmo di assegnamento (per esempio un semplice round robin). I Consumers o consumatori leggono (consumano) i dati presenti all'interno del topic. Nel caso del sistema distribuito il produttore è integrato nell'ecosistema Spark, mentre il consumatore è una libreria creata ad-hoc presente all'interno della Webapp, in particolare nell'ecosistema NodeJS.

Ogni partizione è una sequenza ordinata e immutabile di record che viene continuamente aggiunta a un registro di commit strutturato. Ai record nelle partizioni viene assegnato un numero ID sequenziale chiamato offset che identifica in modo univoco ogni record all'interno della partizione [22].

In Kafka le partizioni sono replicate tra i nodi per garantire protezione in caso di interruzioni dei nodi (broker). La replicazione di una partizione viene gestita automaticamente in modo che sia assegnata a diversi brokers. Kafka elegge per ogni Broker una partizione

“Leader” (indicato con (L)) e tutte le scritture e le letture dovranno passare alla partizione “Leader” scelta. Il traffico dei producer viene indirizzato al leader di ogni nodo, usando lo stato gestito da ZooKeeper.

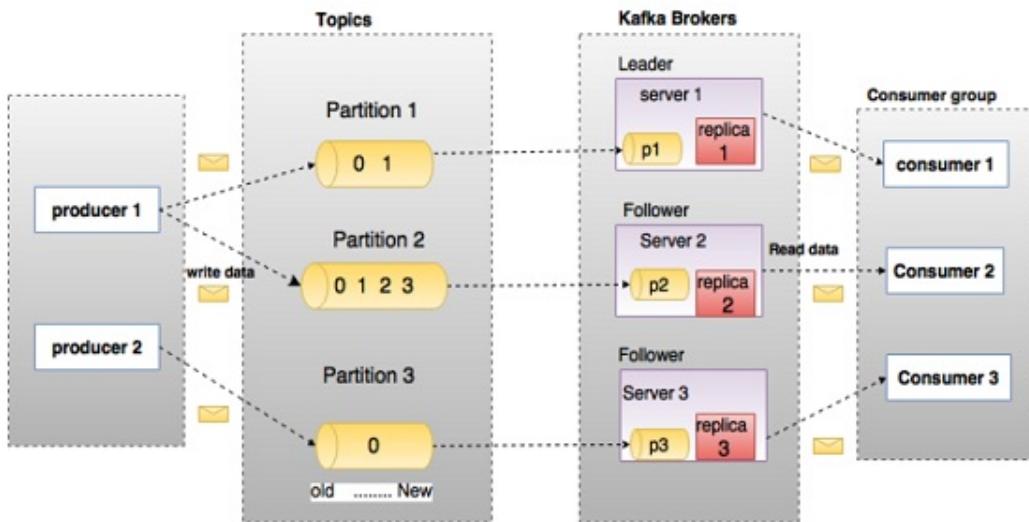


Figura 2.12: Come funziona Kafka

Infine, la messaggistica in Kafka prevede solo due di tipi di modelli:

- *Queuing (Coda)*: un pool di Consumers può leggere dal Server e ciascuno può leggere i dati solamente durante il suo turno;
- *Publish - subscribe*: il messaggio viene trasmesso a tutti i Consumers.

Kafka offre una sola implementazione dell’entità Consumer che generalizza entrambi i modelli, il consumer group. Un Consumer etichetta se stesso con il nome del gruppo a cui decide di far parte e ciascun messaggio pubblicato in un topic, seguito dal gruppo, viene consegnato a ciascun Consumer presente. Se tutti i Consumer hanno lo stesso consumer group, il sistema si reduce ad una semplice coda con priorità first in - first out (FIFO). Se tutti i consumer hanno un consumer group diverso, il sistema automaticamente diventa di tipo publish-subscribe.

2.4 WebApp

Una applicazione web è una applicazione client/server per un ambiente stateless, cioè senza memoria, che utilizza le tecnologie internet. Con il termine Webapp si descrive un’applicazione accessibile via web per mezzo di una network, come ad esempio una Intranet o attraverso la rete Internet. Pertanto, saper programmare per il web significa conoscere i diversi meccanismi e strumenti per conservare o passare i dati, detti parametri, tra le diverse pagine dell’applicazione web. In pratica una Web-application, è un

programma che non necessita di essere installato nel computer in quanto esso si rende disponibile su un server in rete e può essere fatto funzionare attraverso un normale Web browser (es. Google Chrome, Mozilla Firefox, Opera, ecc.). Il client, dopo aver instaurato una connessione con il server, invia la richiesta per un servizio; il server dopo aver elaborato i dati necessari rende disponibile al client il servizio richiesto. A differenza dei siti web statici (HTML), l'applicazione web viene realizzata con una o più tecnologie (Javascript, Ajax, Servlet, Database ecc.) che permettono la creazione di un sito dinamico, cioè di un sito nel quale il contenuto delle pagine varia durante l'interazione.

Le applicazioni Web si pongono come valida alternativa alle tradizionali applicazioni Client/Server per vari motivi:

- *Facilità di distribuzione e aggiornamento*: un'applicazione Web risiede interamente sul server, per cui la sua pubblicazione coincide con la distribuzione e l'aggiornamento ed è automaticamente disponibile a tutti gli utenti.
- *Accesso multi-piattaforma*: l'accesso all'applicazione è indipendente dall'hardware e dal sistema operativo utilizzato dagli utenti;
- *Riduzione del costo di gestione*: l'uso di Internet come infrastruttura per un'applicazione Web riduce notevolmente sia i costi di connettività che i costi di gestione dei client;
- *Scalabilità*: un'applicazione Web ben progettata può crescere insieme alle esigenze dell'azienda senza particolari problemi;

Per poter mostrare l'elaborazione di Spark nel sistema distribuito, si è preferito utilizzare una webapp per i motivi sopra citati. Questa applicazione web fa uso di tecnologie di ultima generazione quali NodeJs ed HTML5. Questo la rende fruibile a tutti gli utenti provvisti di Pc, Tablet o Smartphone con un browser installato. Nel seguente capitolo dunque vengono messi in luce i principali componenti che compongono la webapp, mostrando i punti forti di ogni tecnologia.

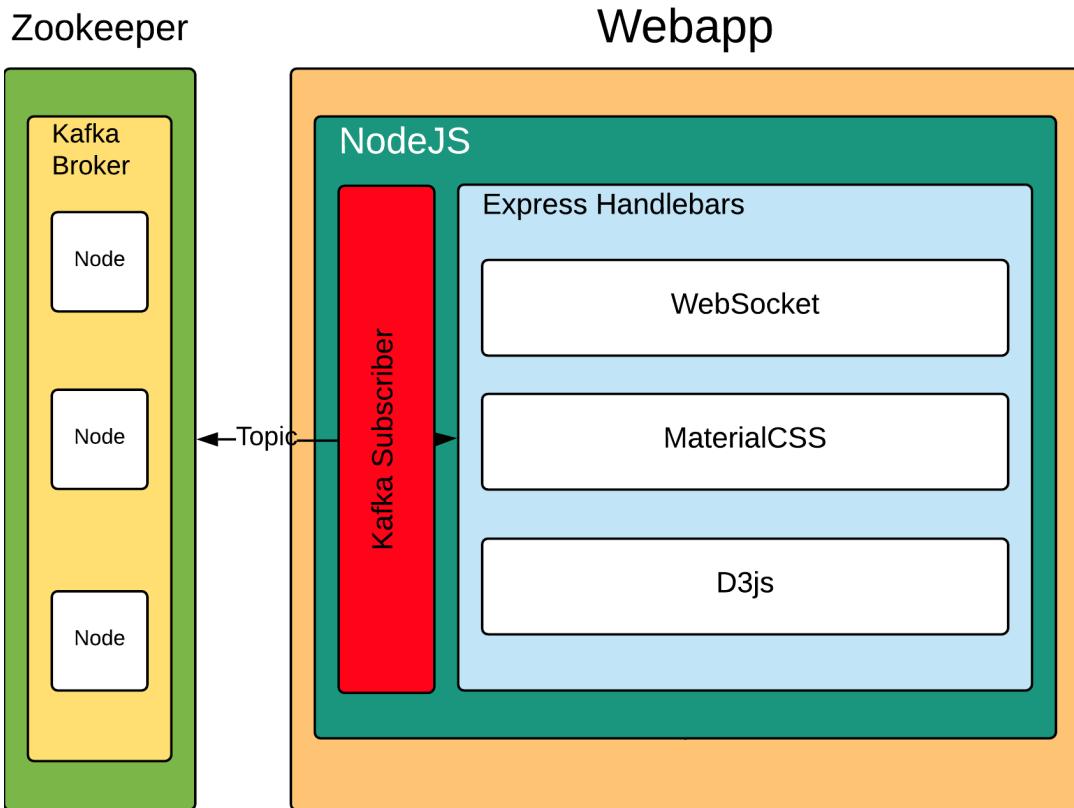


Figura 2.13: Architettura in dettaglio della webapp.

2.4.1 Node.js

Node.js (anche conosciuto come Node o Nodejs) è un framework/piattaforma molto potente basato sul motore *JavaScript V8* di Google Chrome, per creare facilmente applicazioni di Web veloci e scalabili. Pubblicata da Ryan Dahl nel 2009, viene utilizzata per sviluppare applicazioni Web intensive di I/O come siti di streaming video, applicazioni a pagina singola e altre applicazioni Web. Node.js è un ambiente open source, multi-piattaforma per lo sviluppo di applicazioni lato server completamente gratuito, utilizzato da migliaia di sviluppatori in tutto il mondo. Le applicazioni Node.js sono scritte in Javascript e possono essere eseguite all'interno del runtime di Node.js su OSX, Microsoft Windows e Linux.

Utilizza un modello I/O non bloccante e basato sugli eventi che lo rende leggero ed efficiente, perfetto per applicazioni in tempo reale ad alta intensità di dati che funzionano su dispositivi distribuiti [23]. Il modello di networking su cui si basa Node.js non è quello dei processi concorrenti, ma I/O event-driven: ciò vuol dire che Node richiede al sistema operativo di ricevere notifiche al verificarsi di determinati eventi, e rimane quindi in

sleep fino alla notifica stessa. Solo in tale momento torna attivo per eseguire le istruzioni previste nella funzione di *callback*, così chiamata perché da eseguire una volta ricevuta la notifica, che il risultato dell'elaborazione è disponibile. Tale modello di networking, implementato anche nella libreria Event machine per Ruby e nel framework Twisted per Python, è ritenuto più efficiente nelle situazioni critiche in cui si verifica un elevato traffico di rete [23]. Di fronte alle esigenze di migliorare le performance dei software di rete Ryan Dahl ha creato una piattaforma che esegue le operazioni di I/O particolarmente lente (comunicazioni di rete o accesso al disco) in modo asincrono, rendendo la programmazione su Node JS diversa da qualsiasi esperienza con altri linguaggi. In definitiva, l'obiettivo di Node.js è quello di fornire un modo veloce per realizzare applicazioni web scalabili in termini di gestione delle connessioni da parte dei client verso il web server.

Nel momento in cui una operazione di I/O considerata lenta (di solito lo è se riguarda la rete o il disco fisso) viene eseguita da un programma in Node JS, V8 si occupa di trasferire la chiamata su un thread non bloccante fra quelli che ha a disposizione nella sua *thread-pool base*. In questo modo, il thread principale con il codice può continuare la sua esecuzione senza context switch. Nel momento in cui una operazione collegata ai thread non bloccanti è terminata il kernel segnala che questo thread può tornare in coda di esecuzione. A questo punto però V8 si occuperà di intercettare il messaggio, mettere nella propria coda di esecuzione la funzione di callback specificata con l'operazione di I/O terminata e di rimettere il thread non bloccante a disposizione per altre operazioni di I/O. Così facendo, virtualmente il thread che esegue codice non si ferma mai, avvicendando le funzioni di callback delle varie operazioni terminate [2.14].



Figura 2.14: Gestione eventi con Node.js.

Bisogna però sempre tenere a mente, quando si progetta di scrivere un programma con Node.js, che questa architettura ha un effetto collaterale molto pesante; le operazioni

che occupano per lungo tempo il thread in cui viene eseguito il codice (operazioni di calcolo onerose) bloccano l'interno software. Per questo motivo Node.js è assolutamente sconsigliato in caso di operazioni di calcolo complesse e nella fase di progettazione di programmi che usano questa piattaforma è sempre necessario utilizzare questa caratteristica come criterio fondamentale per la scrittura del codice. Di seguito sono alcune delle caratteristiche importanti che rendono Node.js la prima scelta di architetti del software:

- **Asynchronous and Event Driven:** Tutte le API della libreria Node.js sono asincrone, ovvero non bloccanti. Significa essenzialmente che un server basato su Node non aspetta mai che un'API restituisca i dati. Il server passa all'API successiva dopo averlo chiamato ed un meccanismo di notifica di eventi consente al server di ottenere una risposta dalla precedente chiamata.
- **Molto veloce:** Essendo costruito sul motore Javascript V8 di Google Chrome, Node.js è molto veloce nell'esecuzione del codice.
- **Thread singolo ma altamente scalabile:** Node utilizza un singolo thread con loop di eventi. Il meccanismo degli eventi aiuta il server a rispondere in modo non bloccante e rende il server altamente scalabile rispetto ai server tradizionali che creano thread limitati per gestire le richieste. Quindi, Node utilizza un singolo programma con thread e lo stesso programma può fornire il servizio a un numero molto più grande di richieste rispetto ai server come Apache HTTP Server.
- **Nessun Buffering:** Le applicazioni create con node non bufferizzano mai alcun dato. Queste applicazioni generano semplicemente i dati in blocchi.
- **Licenza:** Node è rilasciato sotto licenza MIT.

Esiste un mondo attorno a Javascript composto da librerie che ne estendono le funzionalità. Stesso discorso vale per Node.js, in quanto è attiva una comunità di sviluppo che ha realizzato in questi anni molte librerie per realizzare particolari tipi di supporto (database, Network, . . .) ed un sistema di installazione di questi moduli che si occupa anche di eventuali dipendenze: *npm*. Nei prossimi capitoli saranno analizzati alcuni progetti legati a Node.js usati nella creazione della webapp.

2.4.1.1 Express.js ed Handlebars

Express.js (anche chiamato semplicemente Express) è un framework basato su Node.js che offre un insieme robusto di utilità per realizzare agilmente un'architettura *MVC (Model-View-Controller)* sul lato server di applicazioni web single-page, multi-page ed ibride.

Basato sul modulo di Node chiamato *connect*, risulta essere un ottimo "connettore" o *middleware* tra le diverse librerie che possono popolare una webapp, tra cui WebSocket, Passport, Mustache.js, Handlebars, etc.

Le funzionalità offerte sono il *routing*, la possibilità di gestire le configurazioni dell'applicazione ed un motore di templating.

Al centro di questa libreria c'è il concetto di flusso di funzioni, o come piace dire a certe persone, un set di *livelli di funzione*. Infatti, per creare un'applicazione Express c'è bisogno di creare una sequenza di funzioni (livelli) che il framework può navigare.

Quando una di queste decide di entrare in gioco, può completare il processo e fermare la sequenza. Dopodiché Express riprende a scorrere la sequenza fino alla fine.

Queste funzioni hanno delle callback associate che sono eseguite nel momento in cui un client effettua una richiesta. Questo processo prende il nome di Routing.

Il routing, dunque, è una funzionalità di Express.js, che determina la risposta ad una richiesta client ad un endpoint particolare; il quale può essere un URI (o percorso) o un metodo di richiesta HTTP specifico. In pratica, con Express, si possono gestire le richieste e le risposte *in the middle*, cioè fra server e client. Quando il server riceve una richiesta HTTP la racchiude all'interno di un oggetto *ServerRequest*. Questo oggetto, insieme all'oggetto *ServerResponse*, viene passato al primo middleware che ne può modificare il contenuto, o aggiungere proprietà. Una volta terminata la modifica, il middleware richiamerà il successivo nell'eventuale catena (di funzioni) presente.

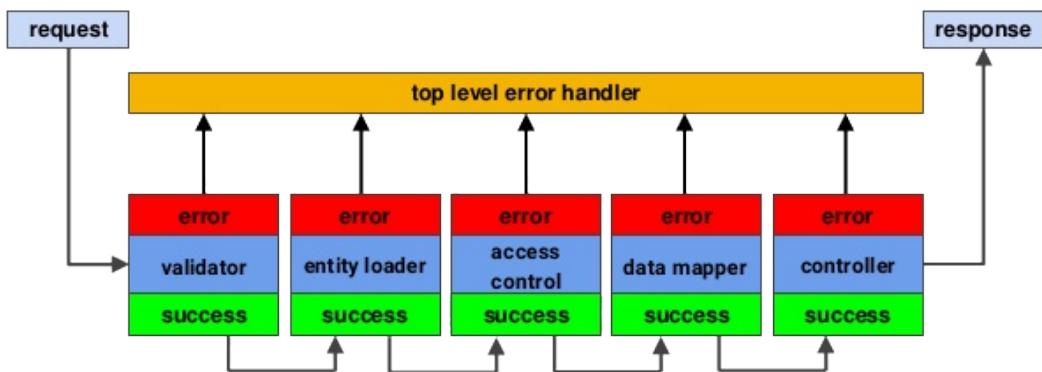


Figura 2.15: Visualizzazione della catena di funzioni per rispondere ad una richiesta

Se l'ultima funzione della catena deve restituire un HTML come risultato, Express chiama in causa il *template engine*. Questo motore, infatti si occupa di fare il tramite tra i dati presenti nel layer model e quello di presentazione. In particolare elabora la richiesta producendo, in fase di run-time, un HTML dinamico partendo da una struttura definita (template). Esistono diversi motori che possono essere utilizzati con Express ognuno con le proprie particolarità e specifiche.

Per questo elaborato la scelta del templating engine è caduta su *Handlebars.js*. La libreria di templating Handlebars consente di creare un'interfaccia utente ricca includendo HTML statico e contenuto dinamico, che possono essere specificati nelle doppie parentesi graffe. Handlebars.js è molto popolare, semplice da usare e con una grande community. È basato sul linguaggio dei modelli di Mustache, ma lo migliora in molti aspetti. Con Handlebars, si può separare la generazione di HTML dal resto del JavaScript e scrivere codice più pulito. Inoltre, aggiunge costrutti (if e cicli for) che permettono di creare dinamicamente l'HTML. Infine, introduce un sistema di *partial* che permette allo sviluppatore di inserire nelle proprie pagine, porzioni di HTML provenienti da file esterni.

Express, quindi, per costruire la pagina di risposta da inviare al client, chiama Handlebars che preleva i file con estensione *.handlebars* li "compila" e genera il risultato finale.

```
1 <div class="entry">
2   <h1>{{title}}</h1>
3   <h2>By {{author.name}}</h2>
4   <div class="body">
5     {{body}}
6   </div>
7 </div>
```

Listato 2.2: Esempio di HTML scritto con Handlebars.

Il listato 2.2 è un esempio di HTML scritto con la sintassi di Handlebars. I valori *title*, *author.name* e *body* saranno sostituiti, in fase di run-time, con i valori provenienti dal controller che passerà un oggetto all'engine templating. vedi listato 2.3.

```
1 var context = {
2   title: "My First Blog Post!",
3   author: {
4     id: 47,
5     name: "Yehuda Katz"
6   },
7   body: "My first post. Wheeeee!"
8 };
```

Listato 2.3: Esempio di variabile passata dal controller al template engine.

2.4.1.2 WebSocket

L'applicazione web dell'elaborato di tesi, come detto in precedenza, prevede una interfaccia grafica per la visualizzazione delle ultime transazioni provenienti da Bitcoin. Per poter implementare questa funzionalità, c'è bisogno di utilizzare tecniche che permettano l'invio di dati tra client e server. Dal semplice request/response di HTTP, l'evoluzione del web e delle sue tecnologie ha portato alla nascita di nuove tecnologie per migliorare sempre di più la comunicazione remota.

Il modello tradizionale di comunicazione, derivato dalle specifiche standard di HTTP, prevedeva una comunicazione sincrona: in seguito a una azione dell'utente (request), il server eseguiva l'operazione richiesta e restituiva il risultato (response). Dopo la richiesta iniziale, il client si poneva in uno stato di attesa fino a quando la risposta non era ricevuta, risultando in uno spreco di tempo e risorse. Il refresh della pagina peggiorava inoltre la user-experience. Allo stesso tempo, il server non manteneva nessuna informazione riguardo alla comunicazione appena avvenuta. Più richieste della stessa operazione dunque venivano ogni volta re-processate e rigenerate per ogni client che le richiedeva. Una comunicazione di questo tipo, semplice da effettuare dal punto di vista implementativo, risulta tuttavia inefficiente e inadatta ad applicazioni di larga scala moderne.

Il primo fondamentale passo è stato rendere la comunicazione da sincrona ad asincrona. Ciò è stato possibile attraverso l'uso di plugin esterni, come Flash, oppure tramite nuovi meccanismi come Ajax. Tuttavia, entrambi presentano dei problemi: nel primo caso, un utente poteva non aver intenzione di installare software esterno, rendendo così inutile

il plugin; nel secondo caso, la gestione stessa della comunicazione attraverso Javascript poteva velocemente raggiungere livelli di complessità non accettabili per grandi applicazioni. Per questi motivi la comunicazione tra client e server distribuito, è implementata con l'utilizzo della nuova tecnologia associata ad HTML5: i *WebSocket*. Formalmente, *WebSocket* è una tecnologia web che fornisce canali di comunicazione full-duplex attraverso una singola connessione TCP. L'API del *WebSocket* è stata standardizzata dal W3C e il protocollo *WebSocket* è stato standardizzato dall'IETF come RFC 6455 [18].

Dunque, tra le novità portate da HTML5, i *WebSocket* rappresentano quella di maggior importanza dal punto di vista dell'interazione tra client e server. *WebSocket* è una tecnologia per effettuare comunicazioni bidirezionali in tempo reale. Essi prevedono un canale di comunicazione sempre attivo, a bassa latenza, tra client e server, utilizzabile da entrambi sia in scrittura che in lettura. Tale canale è costituito da una connessione TCP persistente, garantito da un handshaking client-key iniziale ed un modello di sicurezza originbased. Per la protezione dei dati trasmessi contro lo sniffing sono applicate apposite maschere.

Le caratteristiche principali dei *WebSocket*:

- **Bidirezionali:** Quando il canale di comunicazione è attivo, sia il client che il server sono connessi ed entrambi possono inviare e ricevere messaggi.
- **Full-duplex:** Dati inviati contemporaneamente dai due attori (client e server) non generano collisioni e vengono ricevuti correttamente.
- **Basati su TCP:** Il protocollo usato a livello di rete per la comunicazione è il TCP, che garantisce un meccanismo affidabile (controllo degli errori, re-invio di pacchetti persi, ecc) per il trasporto di byte da una sorgente a una destinazione.
- **Client-key handshake:** All'apertura di una connessione, il client invia al server una chiave segreta di 16 byte codificata con base64. Il server aggiunge a questa un'altra stringa, detta *magic string*, specificata nel protocollo (“258EAFA5-E914-47DA-95CA-C5AB0DC85B11”), codifica con SHA1 e invia il risultato al client. Così facendo, il client può verificare che l'identità del server che ha risposto corrisponda a quella desiderata.
- **Sicurezza origin-based:** Alla richiesta di una nuova connessione, il server può identificare l'origine della richiesta come non autorizzata o non attendibile e rifiutarla.
- **Maschera dei dati:** Nella trama iniziale di ogni messaggio, il client invia una maschera di 4 byte per l'offuscamento. Effettuando uno XOR bit a bit tra i dati trasmessi e la chiave è possibile ottenere il messaggio originale. Ciò è utile per evitare lo sniffing, cioè l'intercettazione di informazioni da parte di terze parti.

WebSocket Protocol

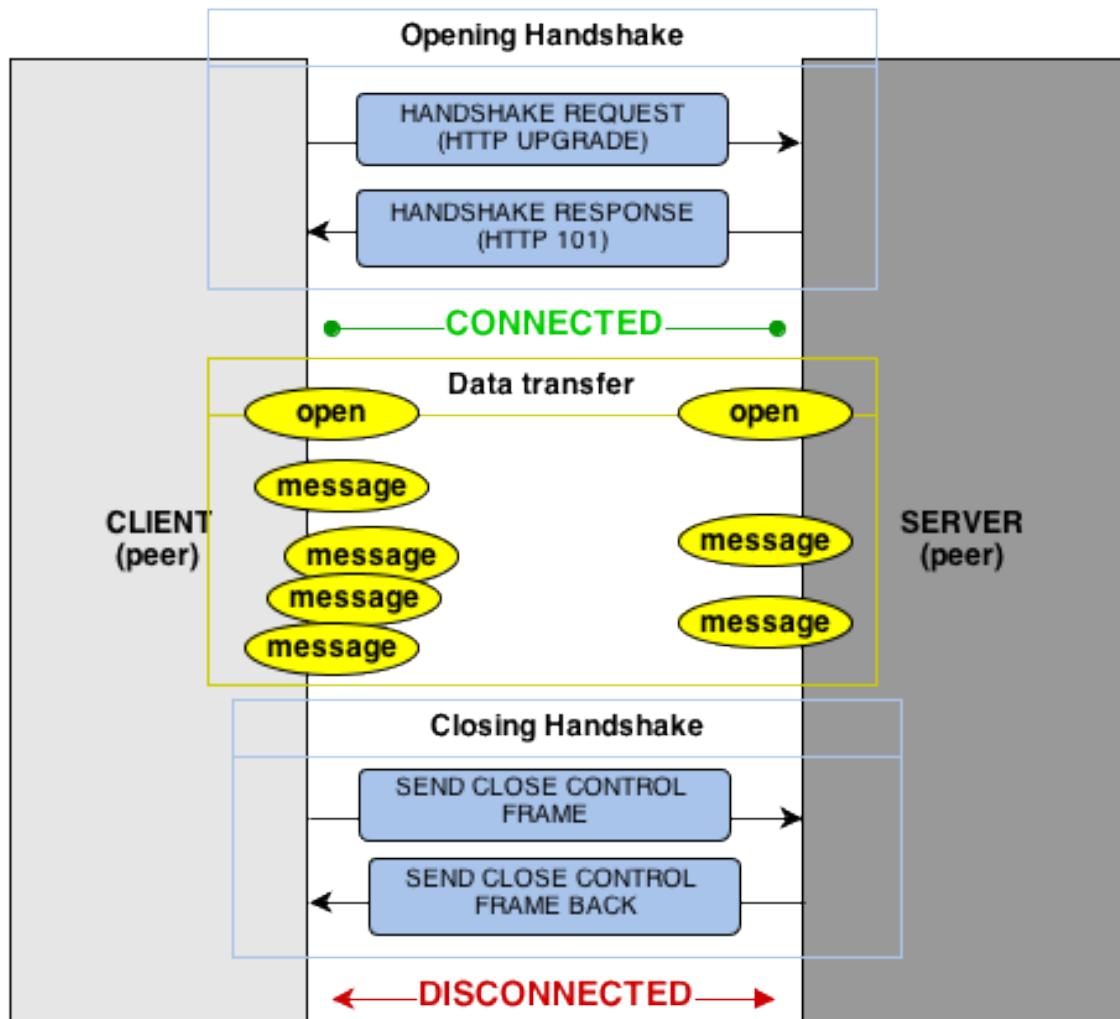


Figura 2.16: Comunicazione client/server tramite WebSocket

WebSocket è disegnato per essere implementato sia lato browser che lato server, ma può essere utilizzato anche da qualsiasi applicazione client-server. L'utilizzo dei WebSockets lato client, è possibile attraverso l'uso di specifiche API Javascript che consentono di ottenere informazioni sullo stato della connessione (aperta, chiusa, in apertura o in chiusura), di interagire con essa (inviare dati o chiudere la comunicazione) o di gestire particolari eventi come la ricezione di errori. Lato server, invece, esistono implementazioni dei WebSockets per la maggior parte dei linguaggi più utilizzati (Node.js, Java, C#, Python, Ruby).

2.4.1.3 MaterializeCSS

L’interfaccia grafica, *GUI (Graphical User Interface)* o *UI (User Interface)*, di una applicazione web determina, in molti casi, il suo successo. Progettare e programmare bene questo componente, quindi, ha un peso molto importante sulla buona riuscita anche del sistema sottostante che lo alimenta.

Material CSS è una libreria di componenti UI creata con CSS, Javascript e HTML che permette di creare interfacce in *Material Design*. Creato e progettato da Google, Material Design è un linguaggio di progettazione che combina i principi classici del design di successo con l’innovazione e la tecnologia. Sebbene si possa scegliere tra milioni di librerie, la scelta di questo tool risulta molto promettente ed è una valida alternativa a prodotti affermati, quali Bootstrap o Foundation, con i quali è molto più complicato produrre un’interfaccia con una vera identità. MaterialCSS offre layout, animazioni e molti altri componenti che rendono semplice la creazione di pagine web. I componenti aiutano a costruire pagine Web e applicazioni *responsive*, consistenti e funzionali, nel rispetto dei principi di progettazione Web moderni quali portabilità del browser e indipendenza del dispositivo. Tra i componenti di maggiore rilievo troviamo: *Table* utile alla creazione responsive di tabelle, *NavBar* che permette la creazione di una barra di navigazione per qualsiasi dispositivo e *Icons* un set di icone disegnate con il CSS, facendo a meno così delle immagini.

Le principali caratteristiche del framework sono:

- **Installazione:** L’installazione del framework è estremamente semplice, infatti bisogna solo importare i CSS e i javascript nel tag head nelle pagine html del proprio sito.
- **Grid System:** Materialize divide il layout di un pagina web in 12 colonne. Questo sistema prende il nome di *grid system* e permette di utilizzare classi *container* che rendono il sito responsive e quindi utilizzabile su ogni dispositivo.
- **Componenti:** Il framework ha una moltitudine di componenti *pronti all’uso*.
- **Temi built-in:** Nasce con una serie di temi grafici che possono essere utilizzati.
- **Gratuito:** E’ gratuito e rilasciato con licenza MIT.

In definitiva, Materialize è nato per facilitare lo sviluppo grafico utilizzando un design sobrio e minimale. Rendendo così la realizzazione di interfacce grafiche semplice ma funzionale.

2.4.1.4 D3.js

Ultimo componente di maggiore interesse è la libreria *D3.js* [2.17]. Questa libreria è usata all’interno di MaterilizeCSS per disegnare i grafi delle transazioni. D3.js (o solo D3 per Data-Driven Documents) è una libreria Javascript scritta da Mike Bostock come progetto successore di un precedente tool di visualizzazione chiamato Protovis [31]. E’ basata sugli standard web e sfrutta appieno le tecnologie dei browser per manipolare gli elementi. Come per JQuery, si utilizza la sintassi CSS per i selettori e si applicano gli stili agli

elementi tramite fogli CSS. D3 a differenza delle altre librerie di grafici, non offre un insieme di grafici già pronti all’uso, bensì un potente framework che permette di realizzare praticamente qualsiasi tipo di grafico manipolando gli elementi di una pagina web di tipo HTML, SVG o Canvas in base al contenuto di un dataset.

La libreria JavaScript D3, incorporata in una pagina web HTML, utilizza funzioni JavaScript prefatte per selezionare elementi del DOM, creare elementi SVG, aggiungergli uno stile grafico, oppure transizioni, effetti di movimento e/o tooltip. Questi oggetti possono essere largamente personalizzati utilizzando lo standard web dei "fogli di stile a cascata", chiamati in inglese CSS. In questo modo grandi collezioni di dati possono essere facilmente convertiti in oggetti SVG usando semplici funzioni di D3 e così generare ricche rappresentazioni grafiche di numeri, testi, mappe e diagrammi. I dati utilizzati possono essere in diversi formati, il più comune è il JSON, valori separati da virgola CSV o geoJSON, ma, se necessario, si possono scrivere funzioni JavaScript apposta per leggere dati in altri formati.

Il concetto centrale del design di D3 è permettere al programmatore di usare dei selettori, come per i CSS, per scegliere i nodi all’interno del DOM Document Object Model e quindi usare operatori per manipolarli, similmente alla libreria jQuery. La selezione può essere basata su tag, elementi, classi, identificatori, attributi o punti della gerarchia. Una volta che gli elementi sono selezionati possiamo applicare operazioni su di essi. Questo comprende leggere ed impostare attributi, mostrare testi, formattare. Gli elementi possono anche essere aggiunti e rimossi. Questo processo di modifica, creazione ed eliminazione di elementi HTML, può essere eseguito in base ai set di dati forniti, che è il concetto di base di D3.js.

D3.js è uno dei migliori framework in circolazione di visualizzazione dei dati e può essere utilizzato per generare visualizzazioni semplici e complesse insieme all’iterazione dell’utente e agli effetti di transizione. La sua potenza è dovuta alcune caratteristiche come l’estrema flessibilità, la facilità d’utilizzo e rapidità, il supporto a grandi dataset, codice riutilizzabile, un’ampia varietà di funzioni ed infine la possibilità di associare dati ad elementi del DOM. D3, infine è un progetto open source e funziona senza plugin. Richiede molto meno codice e offre i seguenti vantaggi:

- Ottima visualizzazione dei dati.
- È modulare. Infatti si può importare, nel proprio progetto, solo una parte di D3, senza dover scaricare l’intera libreria ogni volta.
- Facilità nella creazione di componenti per grafici.
- Manipolazione diretta del DOM e collegamento uno a uno con i dati in memoria.
- È gratuito.



Figura 2.17: Logo D3.js

Capitolo 3

Scelte tecniche ed implementazione

Data la natura dell’elaborato, ossia il recupero e l’analisi delle transazioni provenienti da Bitcoin, si è deciso di implementare una soluzione che prevede i seguenti progetti:

- **Sistema Distribuito (SD in futuro):** Applicazione distribuita che implementa la parte back-end;¹
- **Blockchain Explorer:** applicazione web che implementa il componente front-end.²

Le due applicazioni sono sviluppate con tecnologie e linguaggi diversi. In questo capitolo, viene esposta l’implementazione dei software, attraverso porzioni di codice che aiuteranno a capire al meglio come le due applicazione cooperano nel raggiungimento dell’obiettivo finale.

I linguaggi utilizzati per sviluppare i progetti sono Java 8 per la parte Beck-end , Java-script e HTML5 per la parte front-end.

- **Java 8:** Questo linguaggio orientato agli oggetti, è il più diffuso al mondo grazie alla sua portabilità. Infatti permette di scrivere codice in grado di essere eseguito indipendentemente dalla piattaforma e quindi sui principali sistemi operativi quali Microsoft, Linux e macOS. Questa peculiarità rende Java uno tra i più famosi linguaggi conosciuti e distribuiti al mondo.

Java ha le seguenti caratteristiche che lo rendono unico nel suo genere:

- **Orientato agli oggetti:** In Java, tutto è un oggetto. Java può essere facilmente esteso poiché si basa sul modello Object.
- **Platform Independent:** A differenza di molti altri linguaggi di programmazione, Java viene compilato in un codice byte indipendente dalla piattaforma.

¹**Sistema Distribuito:** <https://github.com/Antonio90/BitcoinSpark.git>.

²**Blockchain Explorer:** <https://github.com/Antonio90/bitcoin-webapp.git>.

Questo codice viene distribuito sul web e interpretato dalla Virtual Machine (JVM) su qualsiasi piattaforma su cui viene eseguito.

- **Semplice:** Java è progettato per essere facile da imparare, perché si basa sul concetto della programmazione orientata agli oggetti (OOP).
- **Sicuro:** Consente di sviluppare sistemi privi di virus e senza manomissioni.
- **Neutrale alle architetture:** Il compilatore Java genera un formato di file "neutro" rispetto all'architettura che rende il codice compilato eseguibile, grazie all'utilizzo del sistema *Java runtime (JRE)* su molti processori.
- **Portatile:** Essendo neutro per l'architettura e senza aspetti di implementazione dipendenti dalle specifiche può essere eseguito su macchine con diversi sistemi operativi.
- **Multithread:** È possibile scrivere programmi in grado di eseguire più attività contemporaneamente. Questa funzionalità di progettazione consente agli sviluppatori di creare applicazioni interattive che possono essere eseguite senza intoppi.
- **Interpretato:** Il codice byte Java viene tradotto al volo per le istruzioni della macchina nativa e non viene memorizzato da nessuna parte. Il processo di sviluppo è più rapido e analitico poiché il collegamento è un processo incrementale e leggero.
- **Funzioni Lambda (solo nella 1.8):** Sono funzioni anonime, ossia una funzione che ha un corpo ma non un nome. In pratica un metodo senza una dichiarazione e quindi senza modificatori d'accesso e dichiarazione del tipo del valore di ritorno [24]. Molto utilizzate in Spark.

L'utilizzo di questo linguaggio per il back-end ha permesso di creare un sistema distribuito indipendente dalle specifiche hardware delle macchine su cui viene eseguito il codice. Inoltre, Spark ha un intero set di API pronte all'uso scritte in Java.

- **Javascript e HTML:** JavaScript è un linguaggio di programmazione leggero e interpretato. Comunemente usato come parte delle pagine Web, le cui implementazioni consentono allo script sul lato client di interagire con l'utente e creare pagine dinamiche. È un linguaggio di programmazione interpretato con funzionalità orientate agli oggetti.

La versione Client-side di JavaScript è la forma più comune di questo linguaggio ma esiste anche la Server-side che viene eseguita sulla macchina su cui è installato il software. La principale differenza delle due forme è che il codice nel caso di Client-side viene integrato in una pagina HTML ed eseguito dal browser dei vari client, mentre quello Server-side viene eseguito sulla sola macchina che ha il sorgente.

I vantaggi nell'utilizzo di questo linguaggio sono:

- **Interfacce più ricche e dinamiche:** È possibile utilizzare JavaScript per includere elementi come componenti e cursori con trascinamento della selezione per fornire un'interfaccia ricca ai visitatori del sito.

- **Meno carico al server:** E' possibile convalidare l'input dell'utente prima di inviare la pagina al server. Ciò consente di risparmiare il traffico del server, il che significa meno carico sul server.
- **Platform Independent:** Il codice, in qualsiasi forma, può essere eseguito sui principali browser o eseguito su qualsiasi macchina poiché interpretato.
- **Tipizzazione dinamica:** In Javascript è possibile dichiarare una variabile senza indicarne il tipo, il quale verrà deciso in fase di runtime [7].

Javascript quindi è stato usato in accoppiata con l'HTML e CSS per rendere l'interfaccia grafica della applicazione web più performante e gradevole. Inoltre, è stato utilizzato anche in forma Server per generare risposte dinamiche da restituire ai client.

3.1 Diagramma delle classi

Durante lo sviluppo della parte back-end, sono state create diverse classi di supporto allo sviluppo del componente. In figura 3.1, diagramma delle classi (UML), è possibile vedere tutte le classi che entrano in gioco per la realizzazione dell'applicativo.

L'applicativo sviluppato ha come punto di partenza (*main*) un Job di Spark. Dalla documentazione ufficiale un job di Spark è un calcolo parallelo composto da più attività che vengono generate in risposta a un'azione [25]. In parole povere, è una serie di istruzioni che vengono eseguite in parallelo su più macchine, ottimizzando i tempi di risposta dell'applicativo.

I passi fondamentali per creare un Job di Spark sono:

- **Caricare le librerie:** Immettere nel proprio *classpath* le librerie di Spark.
- **Inizializzare il contesto di Spark:** La prima operazione di codice da effettuare è quella di inizializzare lo *SparkContext*, in cui comunica la locazione dei cluster da utilizzare ed altre impostazioni quali il numero, i driver di connessione, etc.
- **Parallelizzare le collezioni di dati:** I dati di input devono essere parallelizzati tramite il metodo di *SparkContext parallelize*. Con questo metodo gli elementi della collezione vengono copiati per formare un set di dati distribuito che può essere gestito in parallelo. I dati parallelizzati diventano RDD.
- **Operazioni sugli RDD:** Ai dati parallelizzati si applicano le principali operazioni di Spark come map, filter, reduce, etc.

Nell'applicativo questi passi sono svolti dalla classe principale che contiene il main: "*App*". La classe in questione però utilizza delle classi create ad-hoc per gestire al meglio alcuni aspetti. Di seguito sono riportate le principali classi con una breve descrizione delle loro funzionalità:

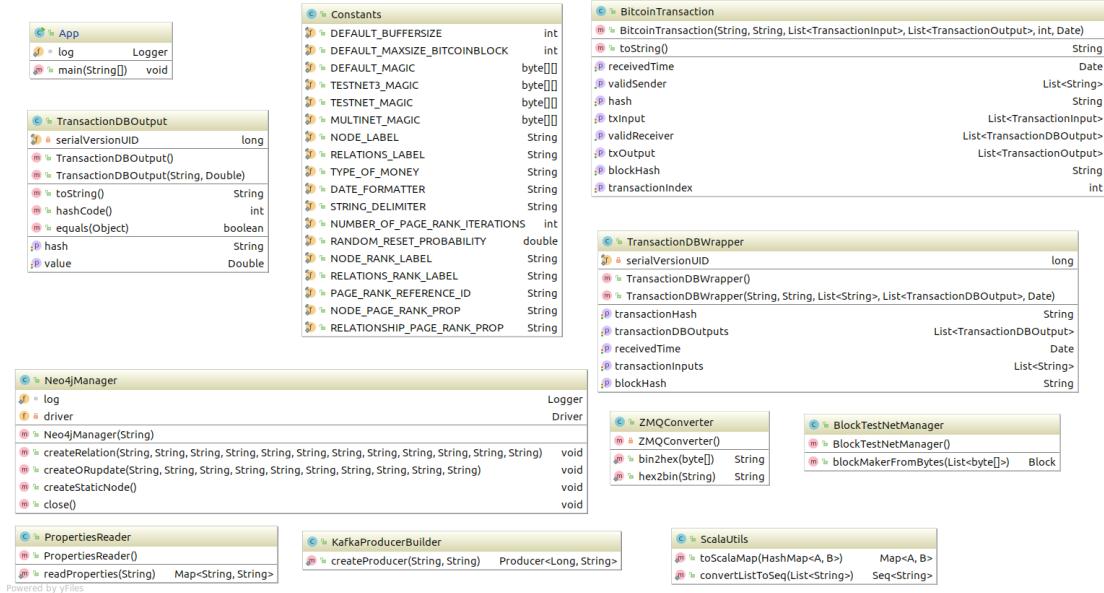


Figura 3.1: UML delle classi (Sistema distribuito).

- **App**: Questa classe è il fulcro di tutto l'applicativo. Infatti, è colei che si occupa della creazione del Job di Spark, del salvataggio nelle periferiche esterne e della messa a disposizione dei dati elaborati. In particolare, la classe App è la prima ad essere eseguita lanciando il Job di Spark Streaming che costantemente attende dati provenienti da blockchain. Ottenuti i dati, li invia al cluster di Spark che li elabora li salva sul filesystem Hadoop e nella base dati Neo4j. Questo primo passaggio è poi accompagnato dall'analisi dello stato delle transazioni esistenti, effettuato tramite la libreria GraphX, la quale applica l'algoritmo del Page Rank sui nodi presenti in Neo4j. Terminata questa fase, le ultime righe di codice creano un collegamento con Kafka per inviare i dati sui topic.
- **Constants**: Questa classe è utilizzata per il salvataggio ed il recupero delle costanti come gli IP delle macchine utilizzate (hadoop, kafka, neo4j), nomi dei nodi per il database etc.
- **BitcoinTransaction**: La classe in questione viene utilizzata da App per creare una struttura dati in memoria che semplifica la gestione delle transazioni. Infatti, essa rappresenta l'astrazione di una transazione Bitcoin.
- **TransactionDBWrapper**: Come dice il nome, questa classe è un wrapper (contenitore) per le transazioni inviate a Neo4j.
- **TransactionDBOutput**: Analogamente alla precedente, questa classe è utilizzata per avere una idea di nodo quando vengono recuperate le transazioni dal database.

- **Neo4jManager:** La classe Neo4jManager è fondamentale per il salvataggio dei dati in Neo4j. Infatti, si occupa della creazione di una connessione con il database, delle query da eseguire su di esso e dell'estrazione dei dati.

Sul fronte visualizzazione il discorso cambia. Il linguaggio Javascript infatti, difficilmente utilizza il paradigma ad oggetti evitando quindi di creare classi. In contrapposizione però utilizza procedure che si attivano in risposta ad una azione o ad una specifica richiesta. Per questo motivo creare un diagramma delle classi non avrebbe senso.

In Blockchain explorer quindi sono stati creati una serie di file Javascript che vengono richiamati all'occorrenza dal framework NodeJS. I file in questione, figura 3.2, sono i componenti principali che compongono la webapp. Occorre però fare una divisione del Javascript che viene eseguito lato client e quello lato Server. Infatti tutti i file contenuti nella cartella *public* sono script che vengono eseguiti solo dai browser dei client, poiché sono integrati nell'HTML che il server genera dinamicamente. I restanti file invece sono eseguiti dal motore di Chrome V8 tramite NodeJS solo sul server.

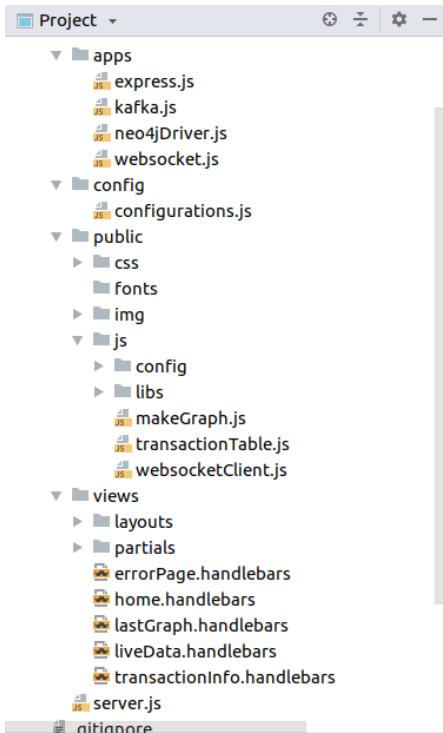


Figura 3.2: Alberatura file di Blockchain Explorer.

Ogni file ha un funzione specifica definita di seguito:

- **server.js:** Questo è un file di inizializzazione, nella quale viene inizializzato il server in ascolto su di una porta e caricato il framework Express. Esso rappresenta dunque il punto iniziale di tutte le richieste che verranno smistate.

- **configuration.js**: E' un file che contiene le principali costanti dell'applicativo come la porta su cui far partire il server, la porta della websocket ed altri settaggi utili per il funzionamento dell'applicazione.
- **express.js**: Questo file è il cuore della webapp. Essa contiene tutte le *route* possibili. Infatti, contiene tutta la catena di funzioni per gestire una richiesta da parte dei client. In altre parole, in questo file sono fatte le associazioni tra path richiesto dal client e callback eseguita in risposta. In questo file troviamo il codice che viene eseguito ad ogni richiesta client, dalla visualizzazione del grafo alle ultime transazioni. Le callback, per restituire ai client una valida risposta, richiamano a loro volta funzioni provenienti da altri file.
- **kafka.js**: Contiene l'implementazione del Kafka Subscriber, quindi si occupa della connessione a Kafka ed alla ricezione dei dati da parte del topic.
- **neo4jDriver.js**: Analogamente a Neo4jManager, viene usato per la connessione ed il recupero dei dati presenti in base dati Neo4j. In particolare, utilizza il linguaggio Cypher per ottenere i risultati dal database.
- **websocket.js**: Questo script crea un server WebSocket per permettere ai vari client di ricevere le transazioni in real-time tramite protocollo WebSocket.
- **Cartella views**: In questa cartella sono presenti tutti i template utilizzati dal sito. Infatti, contiene tutti i file con estensione *.handlebars* utili al framework per creare pagine HTML dinamiche. All'interno di questi template sono linkati tutti gli script e i fogli di stile presenti nella cartella *public*.
- **Cartella public**: Questa cartella contiene tutte le risorse statiche che vengono caricate ed eseguite dai browser. I file Javascript presenti in questa cartella stabiliscono una connessione con il server tramite WebSocket, disegnano il grafo delle transazioni dinamicamente e creano le tabelle all'interno del sito.

Gli script di particolare importanza sono:

- **MakeGraph.js**: Lo script contenuto in questo file, utilizza la libreria D3.js per disegnare i grafi delle transazioni presenti sul sito.
- **transactionTable.js**: Questo file contiene lo script che genera le tabelle. Grazie all'utilizzo della libreria DataTable riesce a disegnare in modo efficiente tabelle anche di grandi dimensioni.
- **websocketClient.js**: Come dice il nome, questo file permette la connessione col server WebSocket istanziato da NodeJS.

3.2 Codice

L'elaborato di tesi, come detto in precedenza, è suddiviso in due distinti progetti: Sistema Distribuito e Blockchain Explorer. In questo capitolo, l'attenzione sarà focalizzata sulla reale implementazione, visualizzando e commentando righe di codice con maggiore interesse.

Ogni applicazione Java ha un *entry point*, cioè una funzione principale che viene richiamata all'avvio dell'applicativo. All'interno del sistema distribuito l'entry point è definita nella classe *App*. Questa classe ha il compito di:

- **Recupero Costanti:** Il primo passo che effettua l'applicazione è il recupero delle costanti definite all'interno di un file di *properties*: *bitcoin.properties*. Per leggere le costanti di questo file utilizza un metodo statico della classe *PropertiesReader* chiamato *readProperties*. Questo metodo [3.1], prende in input un file di properties e restituisce una mappa chiave-valore con tutte le proprietà definite nel file.

```

1 public static Map<String, String> readProperties(String
2   propFileName){
3
4     Map<String, String> propMap = new
5       HashMap<String, String>();
6     Properties prop = new Properties();
7
8     try {
9       prop.load(PropertiesReader.class.getClassLoader()
10      .getResourceAsStream(propFileName));
11
12     for (String key : prop.stringPropertyNames())
13     {
14         String value = prop.getProperty(key);
15         propMap.put(key, value);
16     }
17
18   } catch (IOException e) {
19     e.printStackTrace();
20   }
21
22   return propMap;
23 }
```

Listato 3.1: Metodo *readProperties*.

- **Inizializzare Spark e connessione con Bitcoind:** Caricate le costanti in una mappa, non resta che inizializzare Spark. Per fare ciò, viene creato un oggetto di tipo *JavaStreamingContext* [3.2].

```

1 JavaStreamingContext streamingContext = new
  JavaStreamingContext(sparkConf, new Duration(2000));
```

Listato 3.2: Inizializzazione Spark Streaming.

Questa classe si occupa di creare il contesto streaming di Spark secondo le configurazioni presenti nell'oggetto *sparkConf* e di creare Job eseguiti ogni due secondi.

All’oggetto *sparkConf* viene dato un nome che lo identifica in caso di più Job, gli viene impostato il tipo di cluster da creare ed infine l’URL di connessione col database Neo4j [3.3].

```
1 SparkConf sparkConf = new
  SparkConf().setAppName(propMap.get("appSparkName"));
2
3 if (!sparkConf.contains("spark.master")) {
4     /*local [K] (Run Spark locally with K threads,
5      usually k is set up to match the number of
6      cores on your machine)*/
7     sparkConf.setMaster("local[2]");
8 }
9
10 /**
11  * Configuration of Neo4j
12  */
13 sparkConf.set("spark.neo4j.bolt.url", neo4jConnectionUrl);
```

Listato 3.3: Creazione oggetto *sparkConf*.

Una volta che il contesto di Spark è stato creato, non resta che creare il collegamento tra il Sistema Distribuito e Bitcoind. Questa operazione viene fatta tramite l’utilizzo della libreria *spark-streaming-zeromq* di Apache Bahir. La libreria in questione permette di creare una connessione tra una coda ZMQ (utilizzata da Bitcoind) e Spark [3.4].

```
1 JavaDStream<byte[]> lines =
  ZeroMQUtility.createStream(streamingContext, host,
    subscribe, bytesToObjects);
```

Listato 3.4: Metodo della libreria Spark Streaming ZeroMQ.

Il metodo *createStream* associa, quindi, la ricezione dei dati ad una funzione di callback, che viene richiamata per la gestione dei dati. In questo caso, la funzione in questione è *bytesToObjects* 3.5, che estrae i byte provenienti dalla coda di Bitcoind e li trasforma in oggetti parallelizzati: *JavaDStream*.

```
1 Function<byte[][] , Iterable<byte[]>> bytesToObjects = new
  Function<byte[][] , Iterable<byte[]>>() {
2     @Override
3     public Iterable<byte[]> call(byte[][] bytes)
4         throws Exception {
5             Iterable iterable = Arrays.asList(bytes[0]);
6             return iterable;
7         }
};
```

Listato 3.5: Funzione *bytesToObjects*.

- **Salvare i dati nell’HDFS:** Ottenuti i byte provenienti da Bitcoind può partire l’elaborazione. Il primo step da effettuare è il salvataggio sul filesystem distribuito Hadoop. Questo framework, precedentemente citato, si occupa della storicizzazione distribuita dei dati. Spark, abituato a lavorare su architetture distribuite, ha al suo interno metodi nativi che permettono di salvare dati in Hadoop. Infatti è bastato chiamare il metodo *saveAsTextFile* della classe *RDD* per salvare i dati all’interno del filesystem.

```
1 lines.foreachRDD((bytes, time) -> {
2
3     List<byte[]> blockAsByte = bytes.collect();
4     if (!blockAsByte.isEmpty()) {
5         bytes.coalesce(1).saveAsTextFile(hadoopHdfs +
6             File.separator + "blocks" +
7             File.separator);
8     }
9 });

```

Listato 3.6: Salvataggio Bytes su HDFS.

Nel listato [3.6] per ogni RDD, viene salvato un file di testo sul filesystem di Hadoop.

- **Lanciare il job di Spark:** Terminate le operazioni preliminari, Spark è pronto per eseguire il Job.
Tutti i dati, come visto nel listato 3.4, sono contenuti in un oggetto chiamato *lines*. Questo oggetto contiene la rappresentazione in byte dei blocchi provenienti da bitcoind, parallelizzati sui vari nodi del cluster. Il primo step da eseguire dunque, è la trasformazione da byte in oggetto.

- **Trasformare i Byte in oggetti:** La trasformazione di una sequenza di byte in blocco è demandata a *Bitcoinj*. Questa libreria, creata da Google, è usata per lavorare con il protocollo Bitcoin. Può mantenere un portafoglio, inviare/ricevere transazioni senza bisogno di una copia locale di Bitcoin Core e ha molte altre funzionalità avanzate [3]. Implementato in Java, offre oggetti e metodi per gestire al meglio i dati provenienti da Bitcoind.

Nel Job di Spark, quindi, la trasformazione dei byte è fatta dal metodo *blockMakerFromBytes* di *BlockTestNetManager*. Il metodo in questione, partendo da un array di byte restituisce un oggetto che prende il nome di *Block*. Il listato 3.7 mostra la trasformazione dei byte provenienti di Bitcoin ad oggetto Block.

```
1 BlockTestNetManager blockManager = new
2     BlockTestNetManager();
3 Block block =
4     blockManager.blockMakerFromBytes(blockAsByte);

```

Listato 3.7: Array di Byte trasformato in oggetto Block.

- **Salvare in Neo4j:** Per il recupero e l’analisi, le transazioni sono storicizzate nel database a grafi Neo4j. Dall’oggetto *block* precedentemente inizializzato, quindi, vengono recuperate tutte le transazioni tramite il metodo *getTransactions()*. Il metodo restituisce una lista di transazioni (*Transaction*) associate al blocco che il Job salva nel database.

```
1 for (int i = 0; i < block.getTransactions().size(); i++) {  
2     BitcoinTransaction bTx = new  
3         BitcoinTransaction(tx.getHashAsString(),  
4             block.getHashAsString(),  
5                 tx.getInputs(),  
6                     tx.getOutputs(),  
7                         i, new Date());  
8     Transaction tx = block.getTransactions().get(i);  
9     for(TransactionDBOutput tOut :  
10        bTx.getValidReceiver()) {  
11         log.info("####_Saving_transaction_in_  
12             Neo4j_####");  
13         neo4jManager.createORupdate(Constants.NODE_LABEL,  
14             String.join(Constants.STRING_DELIMITER,  
15                 bTx.getValidSender(), Constants.NODE_LABEL,  
16                     tOut.getHash(), Constants.RELATIONS_LABEL,  
17                         Constants.TYPE_OF_MONEY,  
18                             Double.toString(tOut.getValue()), bTx.getHash()  
19                                 , bTx.getBlockHash(),  
20                                     df.format(bTx.getReceivedTime()));  
21     }  
22 }
```

Listato 3.8: Prelievo transazioni e salvataggio in Neo4j.

Come si vede nel listato 3.8, le transazioni recuperate dal blocco vengono nuovamente trasformate, per comodità, in un oggetto creato ad-hoc *BitcoinTransaction*. Da questo oggetto sono recuperate tutte le transazioni che hanno un valido destinatario (*getValidReceiver()*) e per ognuna di esse viene effettuata una query Cypher 3.9 che crea o modifica i nodi nel database.

```
1 public void createORupdate(String $fromLabel, String  
2     $hashFrom, String $toLabel, String $hashTo, String  
3     $relationLabel, String $relType, String $relValue,  
4     String $transactionHash, String $blockHash, String  
5     receivedDate){  
6     try (Session session = driver.session()) {  
7         String greeting =  
8             session.writeTransaction(new  
9                 TransactionWork<String>(){  
10                     @Override  
11                     public String execute( Transaction tx ) {  
12                         StatementResult result =
```

```
9
10          tx.run("Merge『a:" + $fromLabel +
11                  " {hash:'" + $hashFrom + "'}\』\n" +
12                  "+ \"Merge『b:" + $toLabel +
13                  " {hash:'" + $hashTo + "'}\』\n" +
14                  "+ \"Merge『a)-[r:" +
15                  $relationLabel + "『type:'" +
16                  $relType + "',『value:'" +
17                  $relValue + "',『
18                  transactionHash:'" +
19                  $transactionHash +
20                  "+',blockHash:'" + $blockHash +
21                  "+',『receivedTime:'" +
22                  receivedDate + "'}]->(b);",
23                  parameters( "", "" ) );
24          return "Relationship『Saved";
25      }
26  });
27}
28}
```

Listato 3.9: Metodo che crea o modifica una transazione in Neo4j.

- **Calcolo del PageRank:** L’analisi delle transazioni è un’operazione che richiede molta potenza di calcolo. Il sistema distribuito però, tramite il cluster di Spark, riesce a gestire al meglio queste situazioni garantendo alta affidabilità e rapidità d’esecuzione. Nel progetto di tesi, per mostrare la potenza di Spark è stata utilizzato l’algoritmo di PageRank applicato su milioni di nodi.

```
1 log.info("###『Start『PageRank『calculation『by『Graphx『###");
2
3 Graph graph = Neo4jGraph.loadGraph(
4 streamingContext.sparkContext().sc(),
5 Constants.NODE_LABEL, ScalaUtils.convertListToSeq(
6 (Arrays.asList(Constants.RELATIONS_LABEL)),
7 Constants.NODE_LABEL);
8
9 Graph pageRankGraph =
10 PageRank.run(graph, Constants.NUMBER_OF_PAGE_RANK_ITERATIONS,
11 Constants.RANDOM_RESET_PROBABILITY,
12 stringClassTag, stringClassTag);
13
```

```
12 Neo4jGraph.saveGraph(streamingContext.sparkContext().sc(),
    pageRankGraph, Constants.NODE_PAGE_RANK_PROP, new
    Tuple2<String, String>(Constants.RELATIONS_RANK_LABEL,
    Constants.RELATIONSHIP_PAGE_RANK_PROP) ,
    scala.Option.apply(new
    Tuple2<String, String>(Constants.NODE_RANK_LABEL,
    Constants.PAGE_RANK_REFERENCE_ID)),
    scala.Option.apply(new
    Tuple2<String, String>(Constants.NODE_RANK_LABEL,
    Constants.PAGE_RANK_REFERENCE_ID)), true,
    stringClassTag, stringClassTag);
```

Listato 3.10: Calcolo PageRank e salvataggio su Neo4j.

Nel codice 3.10 sono riportare le operazioni che il Job di spark esegue sui nodi esistenti nel sistema. Il primo passo è quello di caricare in una struttura dati tutti i nodi salvati precedentemente. GraphX per questo scopo, mette a disposizione la classe *Graph*.

Caricati i nodi delle transazioni nella struttura dati di Spark viene invocato il metodo *PageRank.run* che esegue il calcolo del PageRank su tutti i nodi del grafo. Questo metodo ritorna un nuovo grafo contenente, per ogni nodo, il valore del PageRank ottenuto.

Terminata l'esecuzione del metodo, non resta che salvare il nuovo grafo ottenuto nella base dati per un impiego futuro.

- **Pubblicazione dei dati:** L'ultima operazione del Job è quella di pubblicare i dati per essere fruiti dalle applicazioni in real-time. Il sistema distribuito quindi, riceve da Bitcoin le informazioni, le storicizza, le analizza e le rimette a disposizione per altri consumatori. Le righe di codice 3.11, mostrano come sono inviati i dati processati, sul topic *transaction-topic* di Kafka.
-

```
1 log.info("####_Sending_data_to_kafka####");
2 final Producer<Long, String> producer =
    KafkaProducerBuilder.createProducer(kafkaHost + ":" +
    kafkaPort, kafkaAppID );
3 try {
4     log.info("Sending_JSON:_" + json);
5     final ProducerRecord<Long, String> record =
        new ProducerRecord<>(kafkaTopic,
        new Random().nextLong(),
        json);
6     RecordMetadata metadata =
        producer.send(record).get();
7 } finally {
8     producer.flush();
9     producer.close();
10 }
```

Listato 3.11: Invio dati a Kafka.

In particolare, in queste poche righe di codice, viene creato un oggetto KafkaProducer, il quale ha il compito di connettersi al *kafkaHost* e di inviare, tramite il metodo *send*, i dati sul topic settato nel *ProducerRecord*.

Per visualizzare le elaborazioni del sistema distribuito, è stata creata una applicazione web. Blockchain Explorer infatti, è un sito web che mette a disposizione strumenti per controllare le ultime transazioni, i risultati dell’analisi e la navigazione dell’intera Blockchain. Il codice di questo applicativo può essere riassunto tramite una serie di funzioni:

- **Creazione del server:** NodeJs per servire le varie richieste, provenienti dai client, ha bisogno di creare un server. Il modulo che permette questa funzionalità è *http*. Per questo elaborato però, il server è demandato alla libreria Express.js che ci facilita l’implementazione.

```
1 const app = express();
2 const port = process.env.PORT || 3000;
3
4 app.listen(port, function () {
5   console.log('express-handlebars example server listening
6   on: ' + port);
7 }) ;
```

Listato 3.12: Creazione server in NodeJS.

Il listato 3.12 mostra come sia possibile creare un server in NodeJS. Nello specifico, viene creata una variabile chiamata *app*, inizializzata ad *express*, che contiene l’intero web server. L’applicazione con queste poche righe di codice è in grado di rispondere a migliaia di richieste HTTP da parte dei client.

- **Associazione URL-Callback:** Ogni richiesta effettuata da i client ad uno specifico endpoint (URL), nel server, è associata ad una funzione di callback. La funzione dunque viene eseguita ogni qualvolta un utente richiede una specifica pagina. L’associazione URL-Callback è gestita tramite il modulo Express.js.

```
1 app.get('/infotransaction', function (req, res, next) {
2
3   var transactionID = req.query.id || '';
4
5   if(transactionID != ''){
6
7     findTransaction(res, req, next, transactionID);
8
9   } else {
10
11     return next('No transaction id');
12
13   }
14
15 }) ;
```

Listato 3.13: Associazione URL-Callback in Express.

Nel listato 3.13, è possibile notare l’associazione tra l’URL *infotransaction* e la funzione anonima descritta dopo la virgola. Lo scopo di questo URL è quello di ottenere informazioni di una precisa transazione, noto l’ID, all’interno della base dati. Il compito della funzione dunque, è quello di cercare la transazione e di ritornare una pagina HTML con le informazioni richieste. Se la ricerca non dovesse ottenere un risultato, allora la funzione chiama il metodo *next*, il quale permette ad Express.js di eseguire la prossima funzione nella catena di funzioni. In questo caso, viene passata all’handler che si occupa di generare un errore. L’immagine 3.3 mostra la pagina web che l’utente visualizza, quando richiede il dettaglio di una transazione.

The screenshot shows a web application interface for a blockchain explorer. At the top, there is a blue header bar with the 'BLOCKCHAIN Explorer' logo. Below the header, the main content area has a yellow header bar containing the text 'Transaction hash:' followed by a long string of characters: '3099179a36d5b50b9992d89c5be8e9dfb1561ac3a7110c95bd0e049ff8cf1d1'. The main body of the page is divided into several sections:

- Summary**: A table with two rows: 'Received Time' (16/09/2018 20:18:37) and 'Block Hash' (000000000000000a6adb7640d57780286fe24ec2499a8598e1b8a8907ea96a1).
- Total BTC**: Shows a value of 0.42726511 BTC.
- Outputs**: A large gray box containing three transaction outputs. Each output is represented by a colored circle (green, blue, orange) and a Bitcoin address: 'mvtohcrSMwXfB5p3h33xQTGBsq1tCrCqF4', 'mhNnSNiHPu41Dm1fWZ5tt2pqD4MPahuP6U', and '2N3M2yMH63zg9jQyP1CKriKRvvre21LmCTm'.
- Inputs**: A table showing the transaction inputs. It lists two inputs from the same address: 'mhNnSNiHPu41Dm1fWZ5tt2pqD4MPahuP6U' (Value: 0.125 BTC) and 'mhNnSNiHPu41Dm1fWZ5tt2pqD4MPahuP6U' (Value: 0.30226511 BTC).
- Footer**: Contains a section for 'Antonio Riccardi' (Developer for passion), links to 'Pages' (Home, Live data, Graph viewer), and a 'Connect' section with links to Facebook, LinkedIn, and Twitter. It also includes a footer note: 'Made with ❤ by Antonio Riccardi'.

Figura 3.3: Dettaglio transazione.

- **Prelievo dati da Kafka:** Come detto in precedenza, il sistema distribuito al termine della propria elaborazione, pubblica i dati sui topic di Kafka. Blockchain Explorer, preleva i dati messi a disposizione dal topic di Kafka implementando un *HighLevelConsumer*. L’oggetto in questione infatti si connette a Kafka e preleva i dati dal topic *transaction-topic*. Infine li invia tramite websocket ai client che sono in ascolto sul server.

```
1
2 const kafkaBroker = config.kafka.host + ":" +
  config.kafka.port;
3 const client = new kafka.Client(kafkaBroker);
4 const topics = [
5   {
6     topic: config.kafka.topic
7   }
8 ];
9 const options = {
10   autoCommit: true,
11   encoding: 'utf8',
12   groupId: 'bitcoin-webapp' //Math.random().toString()
13 };
14
15 const consumer = new kafka.HighLevelConsumer(client, topics,
  options);
16
17 consumer.on('message', function(message){
18   console.log('Incoming message: ' +
    message.value.toString());
19   wss.sendBroadcast(message.value.toString());
20 });


```

Listato 3.14: Creazione subscriber Kafka.

Il codice 3.14 mostra come la WebApp crea un HighLevelConsumer. Nello specifico viene creato un client, *kafka.Client*, contenente l'IP e la porta di Kafka. Inoltre viene creato un oggetto *topic* nella quale viene specificato il topic alla quale connettersi. I due oggetti, insieme ad altre opzioni *options*, sono passate al costruttore dell'oggetto HighLevelConsumer il quale instaura una connessione con la coda e preleva i dati. Infine, all'oggetto *consumer* viene associata una funzione da eseguire ogni qualvolta un nuovo dato è disponibile. Questa funzione ha il compito di inviare i dati, tramite WebSocket, a tutti i client connessi al server.

- **Comunicazione tramite Websocket:** Per permettere la comunicazione real-time delle transazioni provenienti da Kafka ai vari client, la WebApp crea un server WebSocket. La creazione del server è visibile nel listato 3.15.

```
1 var wss = new webSocket.Server({
2   port: config.webSocket.port
3 });
4
5
6 wss.sendBroadcast = function(message){
7   wss.clients.forEach( function (client) {
8     client.send(message);
9   });
10};


```

Listato 3.15: Creazione di un Server WebSocket.

Infine, al Server WebSocket viene aggiunta la funzione *sendBroadcast*, richiamata nel listato di Kafka 3.14, la quale ha il compito di inviare a tutti i client connessi al server il messaggio, *message*, proveniente da Kafka.

- **Renderizzazione dei Grafi:** La renderizzazione dei grafi è demandata ai browser dei client. Il codice che segue viene inglobato nelle pagine HTML inviate dal server ai vari client. Il browser quindi, riceve dal server sia i dati che il codice da eseguire, generando una pagina come in figura 3.5.

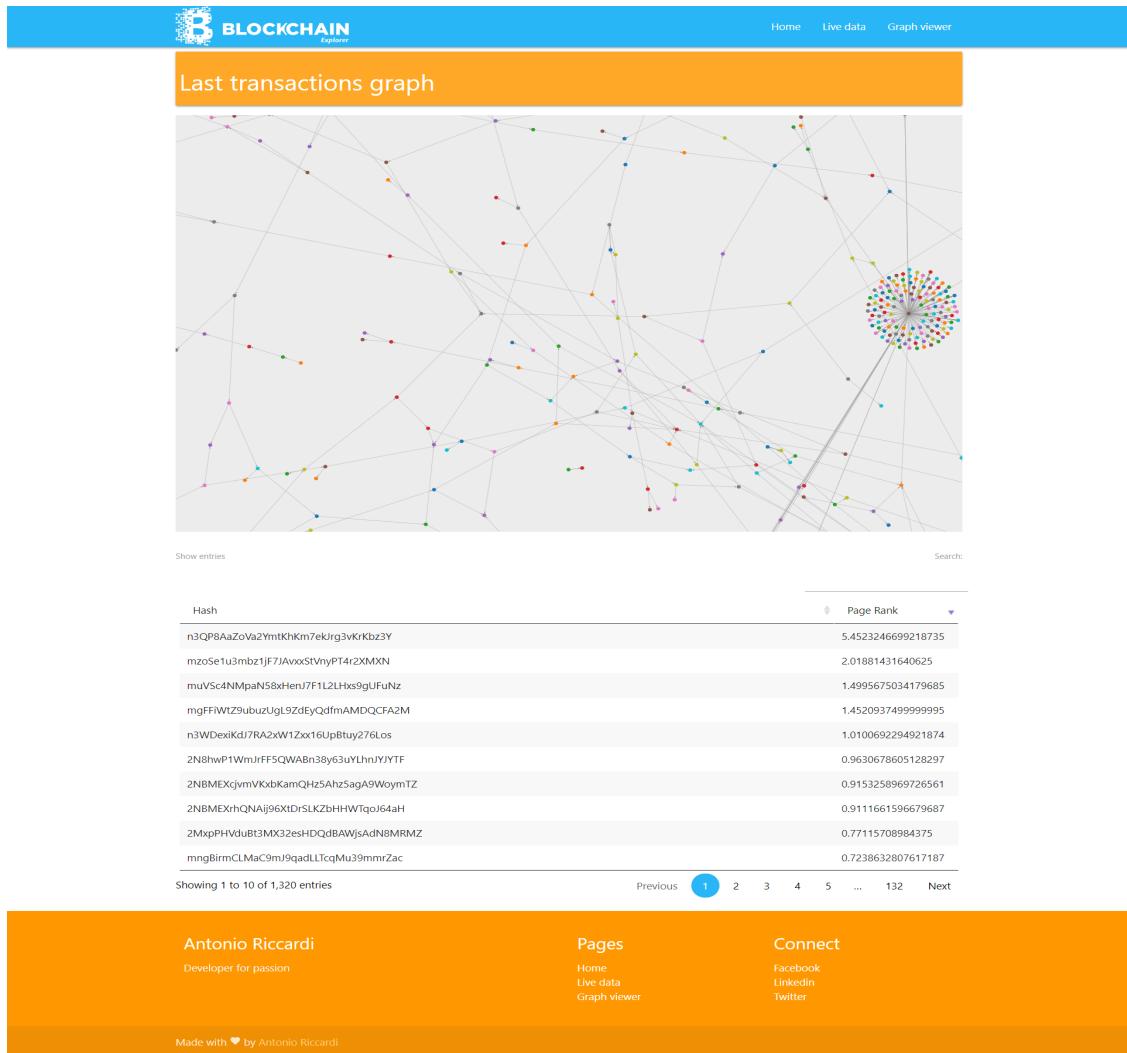


Figura 3.4: Grafo delle transazioni completo.

```
1 var svg = d3.select("#" + idSvg),
```

```
2         width = + $("#" + idSvg).width(),
3         height = +svg.attr("height");
4
5     var zoomLayer = svg.append('g');
6
7     svg.call(d3.zoom().on('zoom', function(){
8         zoomLayer.attr('transform', d3.event.transform);
9 }));
10
11    zoomLayer.append('defs').append('marker')
12        .attrs({'id':'arrowhead',
13            'viewBox': '-0 -5 10 10',
14            'refX':13,
15            'refY':0,
16            'orient':'auto',
17            'markerWidth':13,
18            'markerHeight':13,
19            'xoverflow':'visible'})
20        .append('svg:path')
21        .attr('d', 'M 0,-5 L 10 ,0 L 0,5')
22        .attr('fill', '#999')
23        .style('stroke','none');
```

Listato 3.16: Inizializzazione svg per il grafo.

Nel listato 3.16 sono riportate le righe di codice che servono per l'inizializzazione del tag *svg*, presente nella pagina HTML, nella quale verranno disegnate cerchi e le linee rappresentati rispettivamente nodi ed archi del grafo delle transazioni di bitcoin. Inoltre, su di esso viene aggiunto un layer per lo zoom, che permette all'utente finale di avere una fluida navigazione all'interno del grafo.

```
1             simulation = d3.forceSimulation(nodes)
2                 .force("link", d3.forceLink()
3                     .id(function (d) { return
4                         d.hash;}))
5                     .force("charge",
6                         d3.forceManyBody().strength(-80))
7                     .force("center",
8                         d3.forceCenter(width / 2,
9                             height / 2))
10                    .on("tick", tick)
11                    .stop();
12
13
14             simulation.force("link")
15                 .links(links);
16
17
18             for(i=0; i < 300; i++) simulation.tick();
19
20             link = zoomLayer.selectAll(".link")
21                 .data(links)
22                 .enter()
```

```
17          .append("line")
18          .attr("class", "link")
19          .attr('marker-end', 'url(#arrowhead)')
20          .attr("x1", function(d){ return
21              d.source.x; })
22          .attr("y1", function(d){ return
23              d.source.y; })
24          .attr("x2", function(d){ return
25              d.target.x; })
26          .attr("y2", function(d){ return
27              d.target.y; })
28
29      link.append("title")
30      .text(function (d) {
31          var t = "Transaction Hash: " +
32              d.transactionHash + "\n" +
33              "Received Time: " + d.receivedTime +
34              "\n" +
35              "Block Hash: " + d.blockHash + "\n" +
36              "Value: " + d.value;
37
38          return t;
39
40      });
41
```

Listato 3.17: Creazione linee.

Invece, nel listato 3.17 sono riportate le righe di codice utili alla creazione degli archi tra i vari nodi del grafo. In particolare, alla riga 9 viene utilizzato il metodo *link* di *d3.js* che prende in input i dati da renderizzare, i quali sono trasformati in elementi grafici dell'svg chiamati *line*. A questi sono aggiunti le coordinate di partenza e destinazione, l'icona a forma di freccia e l'attributo *title* contenente tutte le informazioni relative alla transazione (Hash, Timestam, valore totale della transazione e l'hash del blocco di appartenenza).

```
1
2     node = zoomLayer
3         .selectAll(".node")
4         .data(nodes)
5         .enter()
6         .append("g")
7         .attr("class", "node")
8         .attr("transform", function(d){
9             return "translate(" + d.x + ", " + d.y + ")";})
10        .on("contextmenu", function(data, index){
11            console.info("Selected hash: " + data.hash);
12            $("input[type='search']").val(data.hash);
13            $("input[type='search']").keyup();
14            d3.event.preventDefault();
15
16        });
17
18    
```

```
16          })
17          .call(d3.drag()
18              .on("start", dragstarted)
19              .on("drag", dragged)
20              .on("end", dragended)
21          );
22
23      node.append("circle")
24          .attr("r", 5)
25          .style("fill", function (d, i) {return
26              colors(i);})
27
28      node.append("title")
29          .text(function (d) {return d.hash;});
```

Listato 3.18: Creazione dei nodi del grafo.

Analogamente a quanto fatto per gli archi del grafo, nel listato 3.18 viene mostrato come con D3.js sia possibile associare i dati esistenti in memoria ad elementi grafici sullo schermo. In questo caso, viene utilizzato il metodo *data*, riga 4, per passare i nodi da disegnare nella pagina HTML al framework. I nodi, quindi, vengono disegnati tramite il tag *circle* e correlati dalla classe *node* che darà un colore diverso per ogni nodo. Infine, come accade agli archi, viene aggiunto un *title* il quale contiene l'hash dei destinatari o mittenti delle transazioni.

- **Costruzione delle tabelle:** Altro aspetto importante è la creazione delle tabelle. In Blockchain explorer è possibile visionare in forma tabellare le ultime transazioni processate, la sintesi di una transazione singola oppure i valori del PageRank per ogni hash. Nel codice 3.19 è possibile vedere come sia facile trasformare una semplice *<table>* HTML in una tabella con colonne ordinabili, paginazione e filtri di ricerca. Il tutto viene fatto dalla funzione *DataTable(dataTableConfig)* il quale invoca la libreria DataTable che trasforma la tabella *realTimeTransactions* con le proprietà inserite nella variabile *dataTableConfig*.
-

```
1 var initializeTable = function(){
2     transactionsTable =
3         $('#realTimeTransactions').DataTable(dataTableConfig);
4 }
5 var dataTableConfig = {
6     retrieve: true,
7     data: [],
8     dom: 'ftrip',
9     order: [[2, 'desc']],
10    columns:[
11        {title: 'Transaction hash', className: '', orderable:
12            false, visible: true},
12        {title: 'Value Out', className: '', orderable: false,
13            visible: true},
```

```
13         {title: '', className: '', orderable: true, visible:
14             false},
15         {title: '', className: '', orderable: false, visible:
16             false}
17     ],
18     oLanguage: {
19         sEmptyTable: 'Waiting for transactions...'
20     }
21 };
```

Listato 3.19: Utilizzo DataTable.

- **HTML Templating:** Come detto in precedenza il server invia pagine HTML dinamiche partendo da template statici. Questa funzionalità è espletata da Express.js. In particolare la libreria permette di utilizzare un motore grafico per renderizzare le pagine HTML. Nell'elaborato di tesi viene utilizzato Handlebars. Il codice 3.20 mostra come sia semplice associare questo tool come motore grafico di Express. L'associazione viene fatta tramite il metodo *engine*, il quale prende in ingresso una serie di informazioni quali: il template di default, le cartelle contenenti i template ed i partials ed una lista di funzioni, chiamate *helpers*, che possono essere richiamate all'interno dell'HTML.

```
1 // Create 'ExpressHandlebars' instance with a default layout.
2 var hbs = exphbs.create({
3     defaultLayout: 'main',
4     layoutsDir: 'src/views/layouts/',
5     partialsDir: 'src/views/partials/',
6     helpers: {
7         json : function(content) {
8             return JSON.stringify(content);
9         }
10    }
11 });
12 app.engine('handlebars', hbs.engine);
```

Listato 3.20: Associazione Express-Handlebars.

Un template quindi non è altro che una pagina statica HTML, la quale in fase di runtime viene processata e modificata in base alle informazioni provenienti dal server. Un esempio di template è il listato 3.21, il quale mostra la creazione dinamica delle righe della tabella, partendo dall'oggetto *transaction* ricevuto dal server. Come si può notare, viene usato il costrutto *each* per iterare sull'intero oggetto *transaction* e di prelevare solo le proprietà da inserire all'interno dei vari *<td>*.

```
1 <div class="row">
2     <table class="table striped">
3
4         <thead>
5             <tr>
6                 <td>Source</td>
```

```
7             <td>Destination</td>
8             <td>Value</td>
9         </tr>
10        </thead>
11        <tbody>
12            {{#each transaction}}
13            <tr>
14                <td>{{source.properties.hash}}</td>
15                <td>{{destination.properties.hash}}</td>
16                <td>{{relation.properties.value}} BTC</td>
17            </tr>
18            {{/each}}
19        </tbody>
20    </table>
21 </div>
```

Listato 3.21: Template Handlebars.

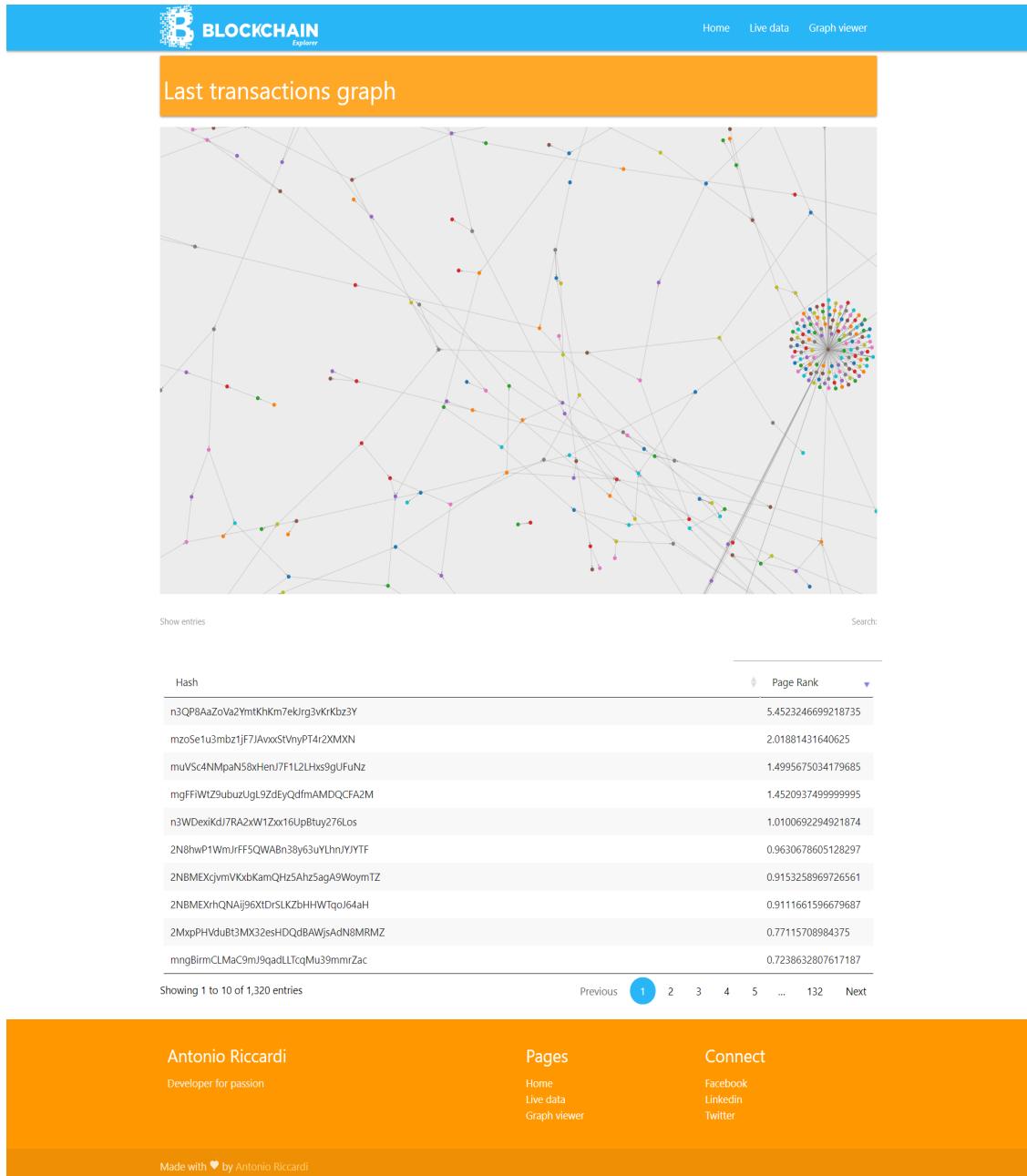


Figura 3.5: Grafo delle transazioni complete.

In figura 3.5 è possibile visualizzare la pagina HTML che viene generata dall'elaborazione del codice 3.21.

3.3 Interfaccia utente

L’interfaccia utente è il luogo d’incontro tra il sistema distribuito e l’utilizzatore finale. In questo elaborato di tesi, come detto in precedenza, l’utente che vuole utilizzare e controllare i risultati del sistema distribuito deve utilizzare un qualsiasi browser e digitare l’indirizzo web del server di Blockchain explorer. L’utente dunque da qualsiasi tipo di dispositivo può accedere alla home page del portale ed iniziare ad utilizzare il sistema. L’immagine 3.6 mostra la prima schermata che l’utente visualizza quando accede al portale.

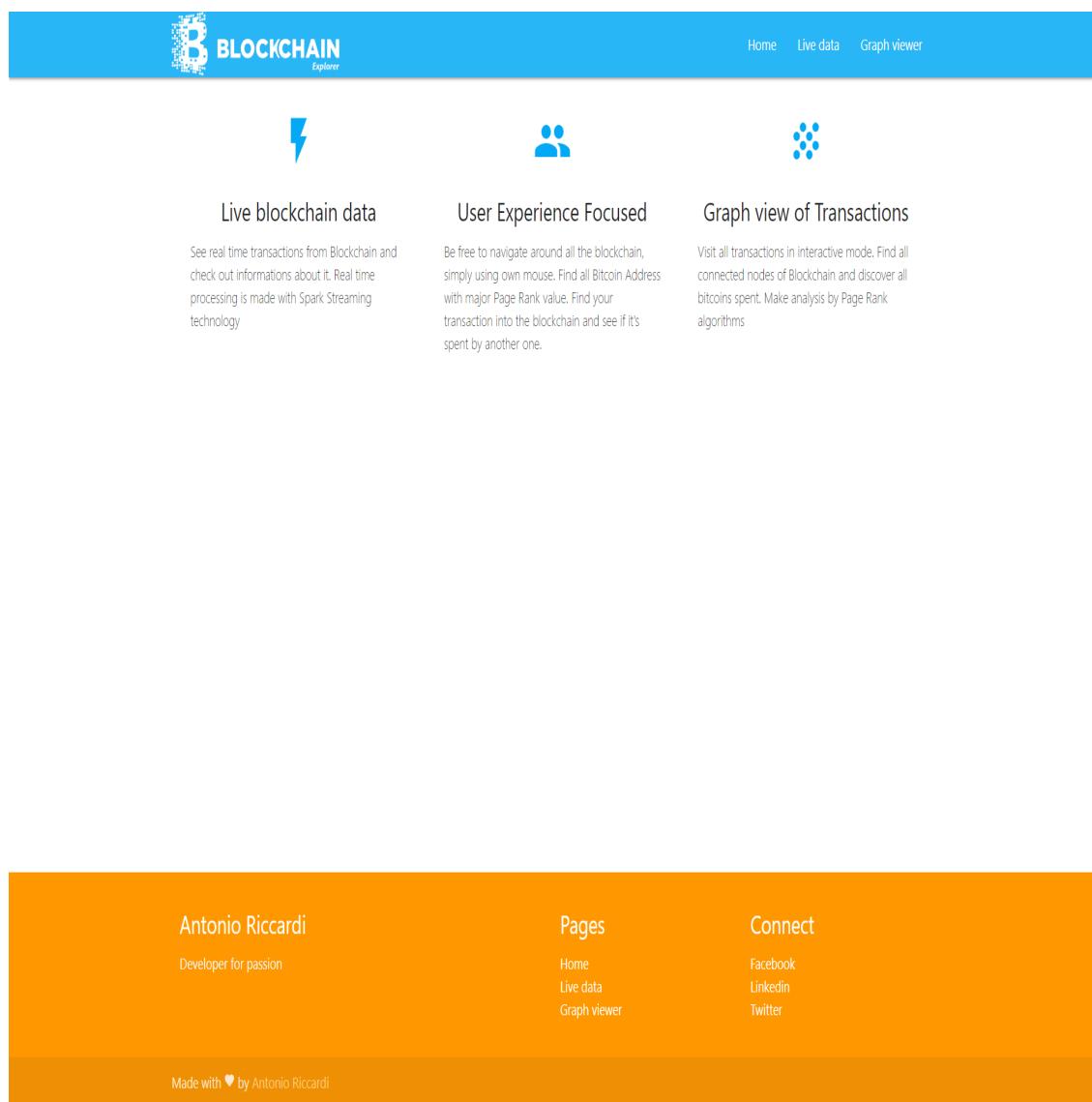


Figura 3.6: Home page Blockchain Explorer.

Nella prima pagina mostrata agli utenti, sono descritte brevemente tutte le funzionalità offerte dal sito. Per accedere a queste funzionalità il sito mette a disposizione due menù: il primo nella parte alta ed il secondo nel footer. Entrambi i menù forniscono i link alle pagine per visualizzare i dati in real time (Live Data) ed il grafo delle transazioni con i relativi page rank (Graph viewer).

La ricezione dei dati in real time è mostrata in figura 3.7 ed accessibile dai menù tramite la voce *Live Data*. Questa funzionalità quindi, permette di controllare le ultime transazioni avvenute sulla Blockchain, in maniera tabellare, senza dover ricaricare la pagina.

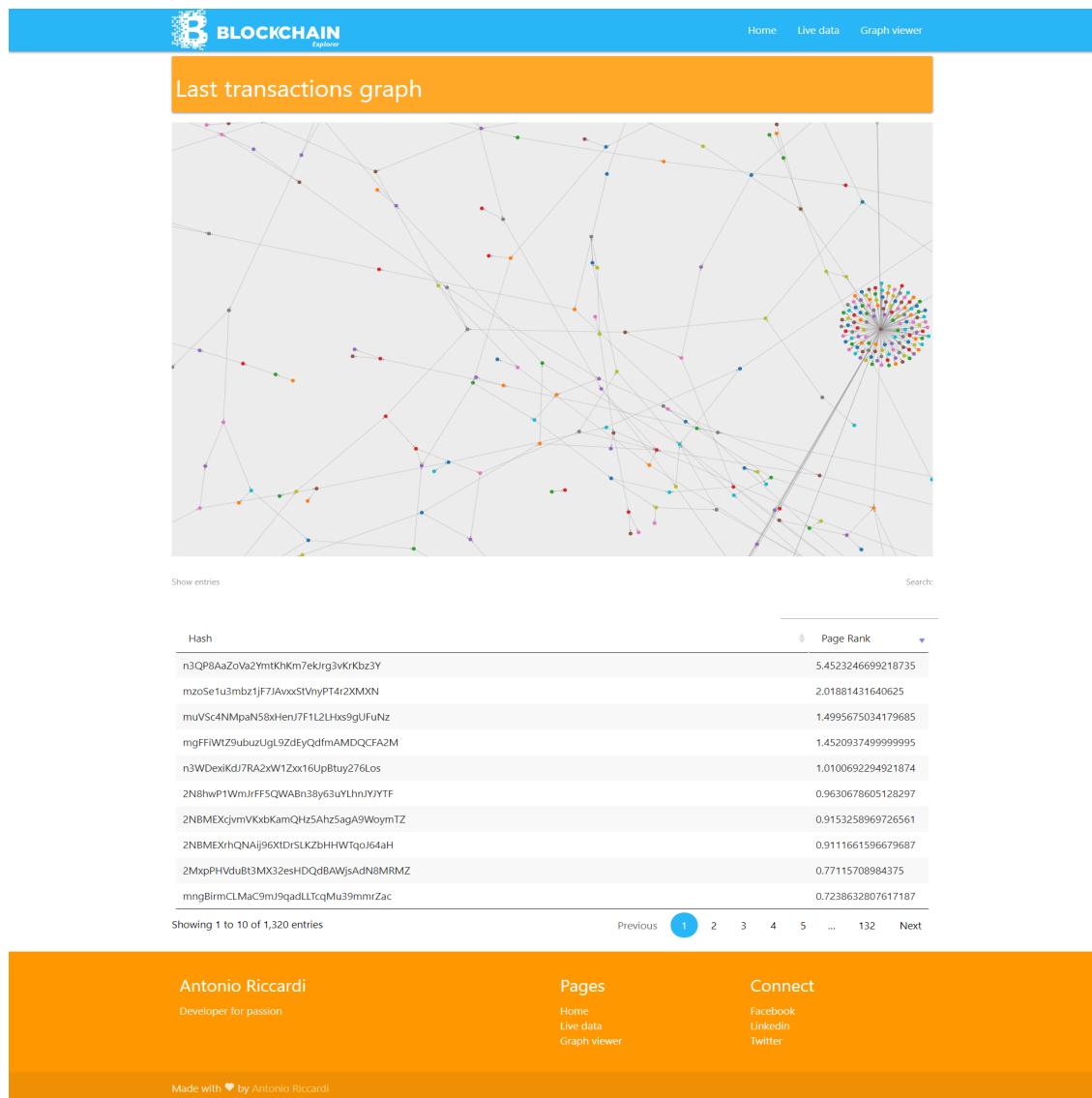


Figura 3.7: Elenco ultime transazioni.

Live data, inoltre offre la possibilità di mostrare il dettaglio di una transazione semplicemente selezionando l'hash dalla tabella. Questa operazione apre una nuova scheda

del browser, che mostra le informazioni di dettaglio e la ricostruzione grafica della transazione selezionata. L’immagine 3.8 mostra una scheda di dettaglio per la transazione con hash **3099179a36d5b50b9992d89c5be8e9dfb1561ac3a7110c95bd0e049ff8cfe1d7**

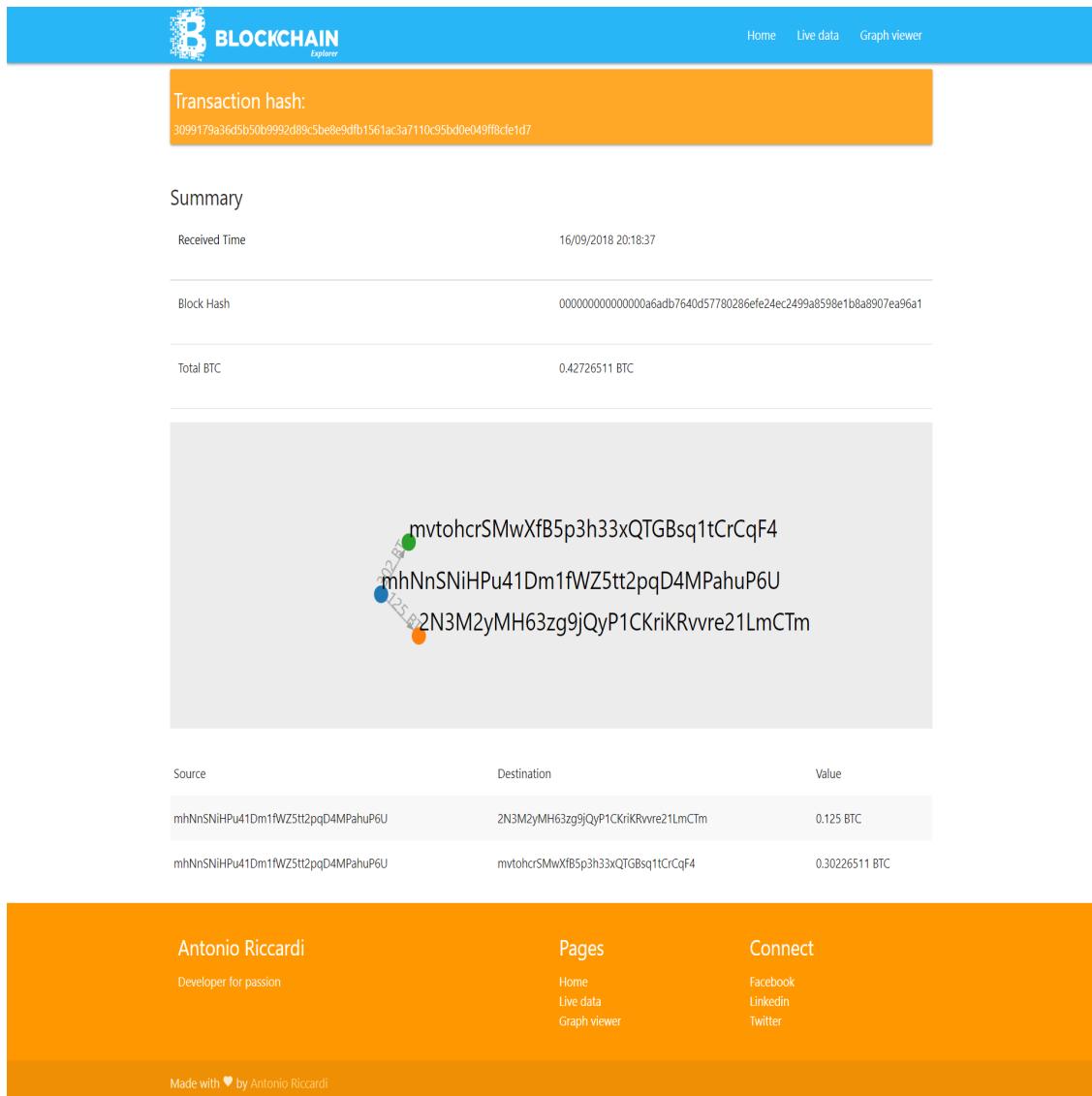


Figura 3.8: Dettaglio transazione.

In particolare la figura sopra citata aggiunge informazioni relative al timestamp della transazione, l’hash del blocco di appartenenza e il totale dei Bitcoin spesi in questa transazione. Infine, è presente un grafico che raffigura lo scambio di bitcoin ed una tabella che descrive l’andamento di tale transazione. Nell’immagine 3.8 è possibile notare che indirizzo hash **mhNnSNiHPu41Dm1fWZ5tt2pqD4MPahuP6U** ha avviato la transazione inviando 0.125 BTC al destinatario **2N3M2yMH63zg9jQyP1CKriKRvvre21LmCTm** ed i restanti 0.30226511 BTC all’hash **mvtohcrSMwXfb5p3h33xQTGBsq1tCrCqF4**;

in virtù del protocollo Bitcoin, il quale stabilisce che il resto di una transazione genera un nuovo hash e quindi un nuovo input per una successiva transazione, uno dei due hash citati precedentemente si presuppone essere il resto di questa transazione e che quindi appartenga alla stessa persona che ha avviato la transazione.³

Altra funzionalità molto importante fornita dall'applicazione è la possibilità di controllare lo stato della blockchain. Questa particolare funzione è accessibile dai menù cliccando sulla voce *Graph viewer*.

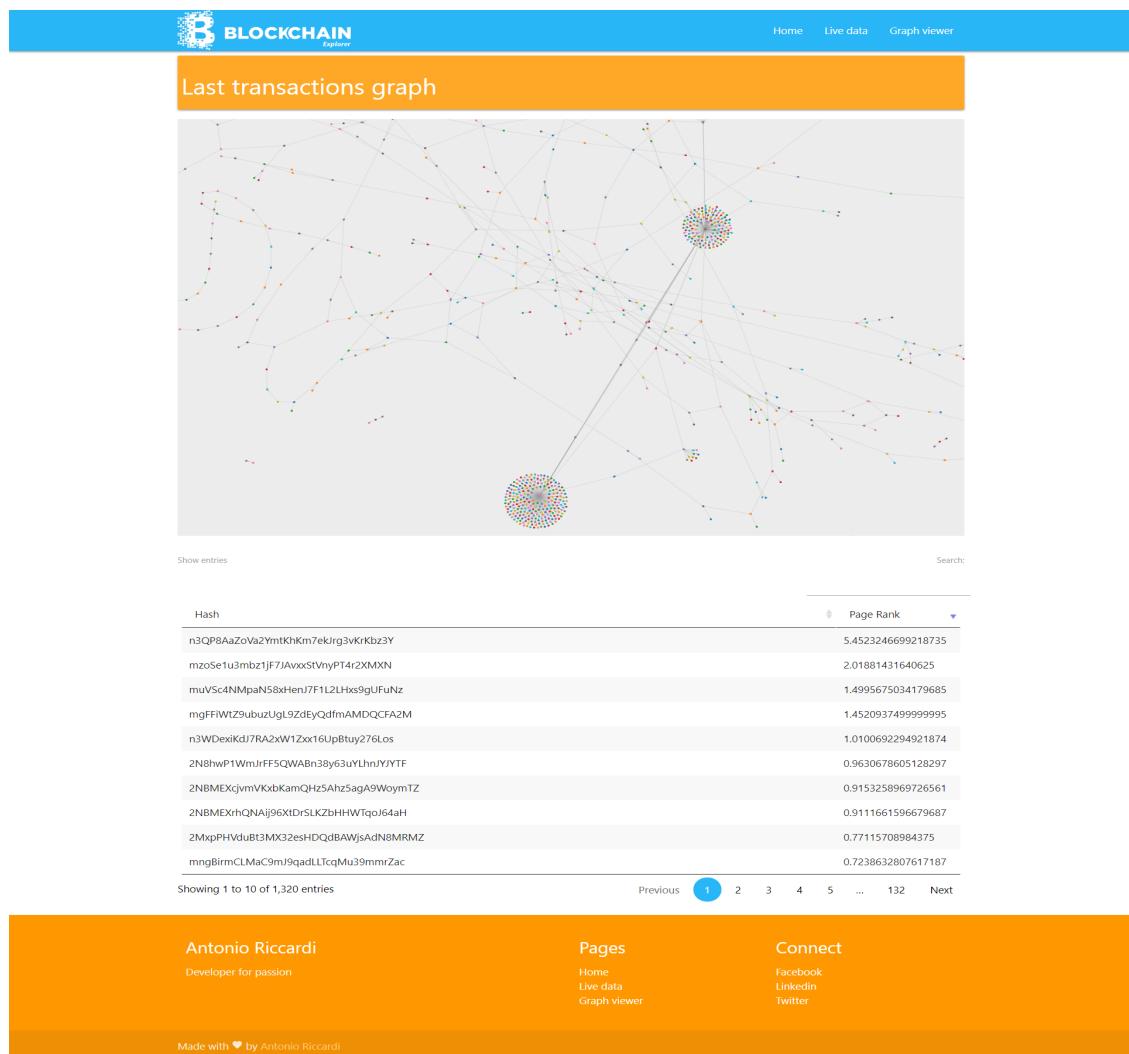


Figura 3.9: Visualizzazione intero grafo con calcolo del Page Rank.

L'immagine 3.9 mostra lo stato della blockchain all'interno della base dati del sistema

³Questa transazione può essere verificata sul sito ufficiale di *BlockCypher* all'indirizzo <https://live.blockcypher.com/btc-testnet/tx/3099179a36d5b50b9992d89c5be8e9dfb1561ac3a7110c95bd0e049ff8cf1d7/>

tramite grafo orientato ed il valore del PageRank per ogni indirizzo hash calcolato dal sistema distribuito.

Tramite l'utilizzo del mouse, l'utente può navigare attraverso il grafo andando ad ingrandire per trovare un particolare indirizzo hash [3.10] oppure cercare una transazione ed ottenere i dati più significativi [3.11] semplicemente passando con il mouse sopra.

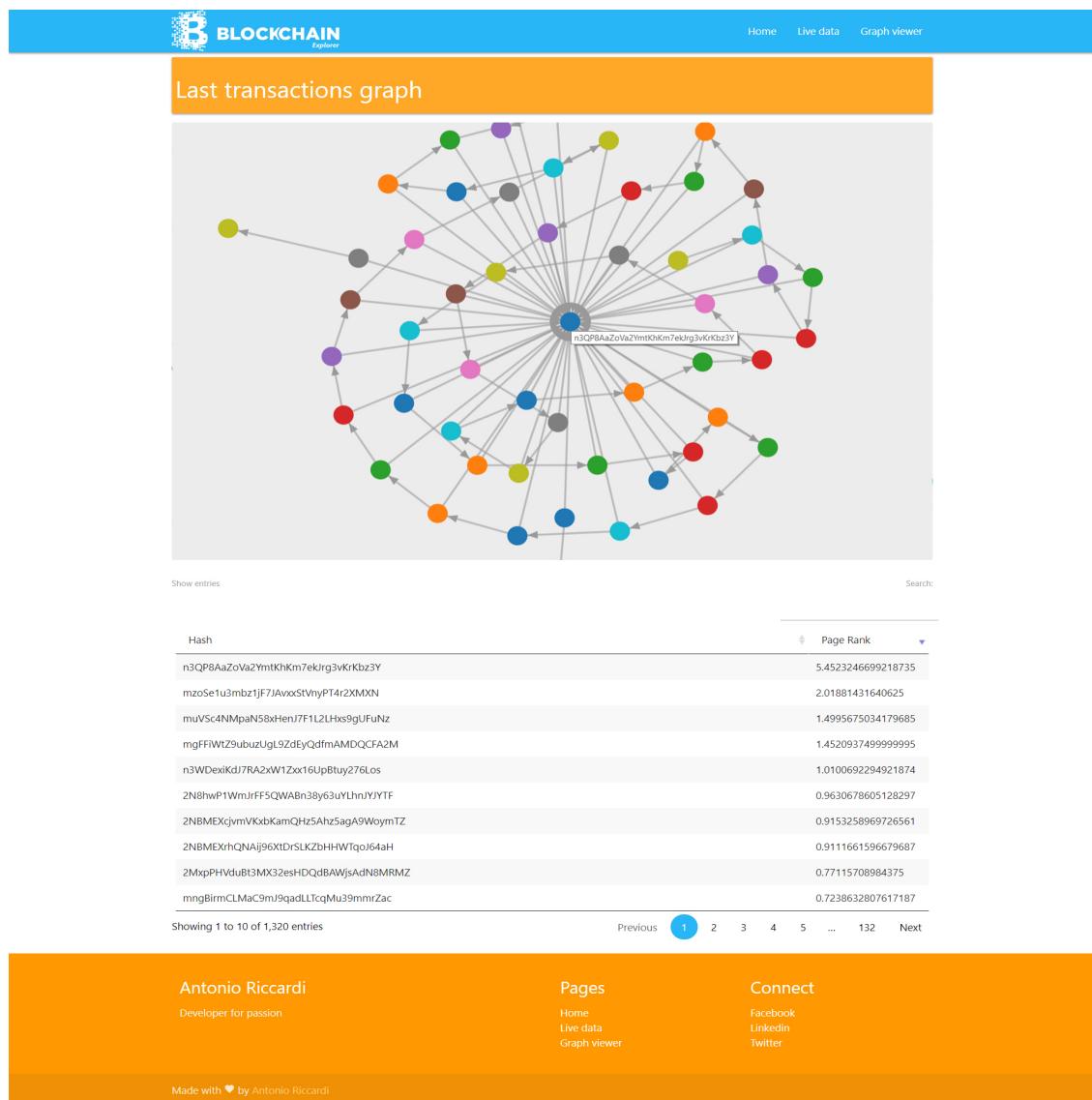


Figura 3.10: Dettaglio nodo centrale.

3 – Scelte tecniche ed implementazione

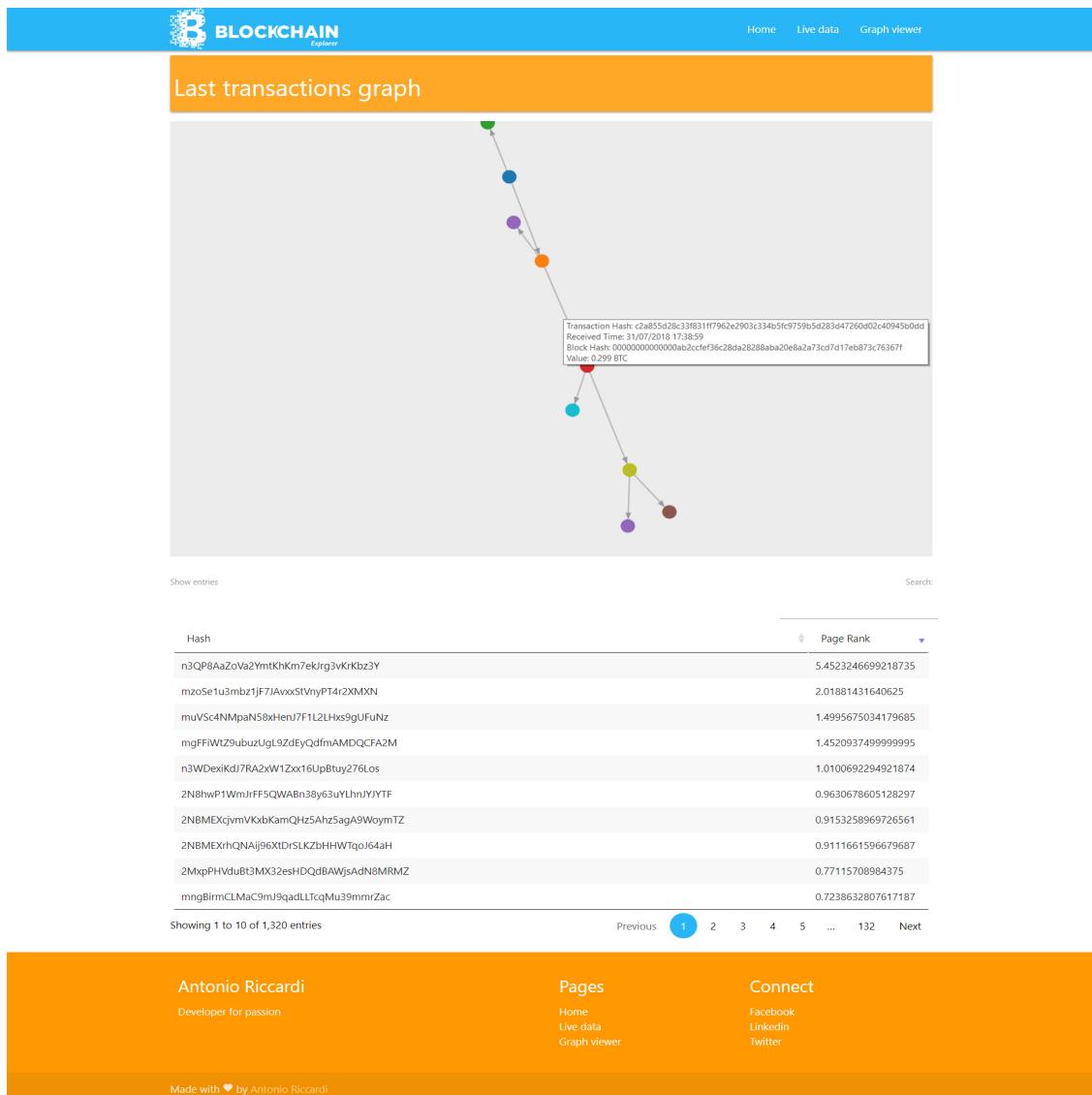


Figura 3.11: Dettaglio transazione all'interno del grafo.

Infine, è possibile ricercare hash all'interno della tabella del PageRank semplicemente cliccando con il tasto destro sul nodo all'interno del grafo. Questa funzionalità cercherà all'interno del database il valore del page rank associato al nodo e lo mostra nella tabella sottostante.

Capitolo 4

Conclusioni e sviluppi futuri

Il fenomeno delle monete virtuali sta avendo molto successo anche tra le persone non vicine all'informatica. Bitcoin è la prima moneta elettronica ed anche la più famosa. Fautrice di una rivoluzione nel campo della finanza, bitocoin ha portato un innovativo sistema per la decentralizzazione e validazione delle transazioni: Blockchain. La blockchain e l'impossibilità di una sua manomissione è sicuramente l'elemento più innovativo del sistema, la cui applicazione alternativa a Bitcoin sembra in grado di rivoluzionare tutti i sistemi di gestione centralizzata a cui siamo abituati.

L'elaborato di tesi svolto aveva come obiettivo quello di sviluppare un sistema distribuito per la gestione dei Big Data provenienti da Bitcoin. Esso è una possibile soluzione che permette di elaborare grosse moli di dati come quelle di Bitcoin. In corso d'opera inoltre, è stata pensata anche una soluzione che permettesse agli utenti meno esperti di visualizzare le transazioni ed i risultati ottenuti dall'elaborazione: Blockchain Explorer.

Il sistema distribuito nei test effettuati, con carichi elevati, ha ottenuto delle ottime prestazioni. In 24 ore di utilizzo non ha mai subito dei rallentamenti nel processare i dati. Altro discorso per quanto riguarda la parte web, infatti non sono stati effettuati test di carico eccessivi, ma le tecnologie utilizzate nella costruzione garantiscono ottime performance.

Tuttavia entrambi i progetti sono resi pubblici sul repository di GitHub, per eventuali suggerimenti o migliorie da apportare. Il sistema distribuito però, è stato pensato ed implementato per non essere legato solo ai Bitcoin. Infatti, Spark può essere facilmente riadattato per ricevere stream di dati diversi dai bitcoin. Questo dà al sistema distribuito una elasticità tale da poter facilmente cambiare la fonte dati ma di lasciare invariato il cluster sottostante. Questo potrebbe essere non uno sviluppo futuro ma un "improvement" da effettuare.

Bitcoin, come detto in precedenza, ha lo scopo di essere anonimo. Col sistema distribuito invece questa peculiarità potrebbe venir meno. Infatti, si potrebbe sviluppare un sistema per capire chi si cela dietro un indirizzo bitcoin. Questa funzionalità potrebbe essere sviluppata collegando le transazioni salvate nel database del sistema distribuito con un algoritmo di ricerca sul web dell'indirizzo preso in esame. Infine un altro sviluppo che si potrebbe prendere in considerazione è quello di sviluppare un'interfaccia grafica per smartphone e tablet così da tener sempre a portata di mano lo stato di Bitcoin.

Riferimenti bibliografici

Manuali cartacei

- [1] Maarten van Steen e Andrew S. Tanenbaum. *A brief introduction to distributed systems.* 2008.
- [2] Andreas M. Antonopoulos. *Mastering Bitcoin.* "O'Reilly Media, Inc."
- [6] Rik Van Bruggen. *Learning Neo4j.* "O'Reilly Media, Inc.", 2014.
- [7] Douglas Crockford. *JavaScript. Le tecniche per scrivere il codice migliore.* Tecniche Nuove, 2009.
- [9] Wolfgang Emmerich. *Distributed System Principled.* 1997.
- [12] Benjamin Reed Flavio Junqueira. *ZooKeeper: Distributed Process Coordination.* "O'Reilly Media, Inc.", 2013.
- [13] Gerard Maas Francois Garillot. *Stream Processing with Apache Spark.* "O'Reilly Media, Inc.", 2017.
- [15] Alex Holmes. *Hadoop in practice.* Manning Publications Co., 2012.
- [17] Leslie Lamport. «Distribution». In: (1997). URL: <https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Distribution.pdf>.
- [18] Andrew Lombardi. *WebSocket: Lightweight Client-Server Communications.* "O'Reilly Media, Inc.", 2015.
- [19] Robin East Michael S. Malak. *Spark GraphX in Action.* MANNING, 2016.
- [21] Satoshi Nakamoto. «Bitcoin: A Peer-to-Peer Electronic Cash System». In: (2009). URL: <https://bitcoin.org/bitcoin.pdf>.
- [22] Todd Palino Neha Narkhede Gwen Shapira. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale.* "O'Reilly Media, Inc."
- [23] Packt Publishing. *Learning Node.js.* "O'Reilly Media, Inc.", 2016.
- [24] Herbert Schildt. *Java: A Beginner's Guide, Sixth Edition.* McGraw-Hill Education, 2014.
- [26] Andrew Stuart Tanenbaum. *Sistemi distribuiti. Principi e paradigmi.* 2001.
- [27] Tom White. *Hadoop: The definitive guide.* "O'Reilly Media, Inc.", 2012.
- [31] Nick Qi Zhu. *Data Visualization with D3.js Cookbook.* PACKT PUBLISHING, 2013.

Siti Web consultati

- [3] bitcoinj.github.io. *Bitcoinj*. URL: <https://bitcoinj.github.io/>.
- [4] Blockchain.com. *Chart bitcoin transactions*. URL: <https://www.blockchain.com/it/charts/n-transactions?timespan=all> (visitato il 2018).
- [5] Blockchain.com. *Chart value of Bitcoin*. URL: <https://www.blockchain.com/charts/market-price?timespan=all> (visitato il 2018).
- [8] www.nosql database.org. *NoSQL Database*. URL: <http://www.nosql-database.org/> (visitato il 2018).
- [10] en.bitcoinwiki.org. *Bitcoind*. URL: <https://en.bitcoinwiki.org/wiki/Bitcoind> (visitato il 2018).
- [11] en.wikipedia.org. *Merkle Tree*. URL: https://en.wikipedia.org/wiki/Merkle_tree (visitato il 2018).
- [14] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. 2013.
- [16] it.wikipedia.org. *PageRank*. URL: <https://it.wikipedia.org/wiki/PageRank> (visitato il 2018).
- [20] Microsoft. *Introduzione ad Apache Kafka*. URL: <http://docs.microsoft.com/it-it/azure/hdinsight/kafka/apache-kafka-introduction>.
- [25] spark.apache.org. *Cluster mode overview*. URL: <https://spark.apache.org/docs/latest/cluster-overview.html>.
- [28] www.albertodeluigi.com. *Come avviene una transazione bitcoin*. URL: <http://www.albertodeluigi.com/index/bitcoin/transazione-bitcoin#panel-108-0-0-1> (visitato il 2018).
- [29] www.google.com. *Google Press Center: Fun Facts*. URL: <https://web.archive.org/web/20090424093934/http://www.google.com/press/funfacts.html> (visitato il 2018).
- [30] zeromq.org. *How come ØMQ has higher throughput than TCP although it's built on top of TCP?* URL: <http://zeromq.org/area:faq#toc6> (visitato il 2018).

]