ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DISI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Sistemi Distribuiti

ONLINE STREAM PROCESSING DI BIG DATA SU APACHE STORM PER APPLICAZIONI DI INSTANT COUPON

CANDIDATO

Ardit TAHIRI

RELATORE:

Chiar.mo Prof. Paolo BELLAVISTA

CORRELATORE:

Chiar.mo Prof. Antonio CORRADI

Anno Accademico 2014/2015

Sessione III

Për familjen time..

Introduzione

Big data è il termine usato per descrivere una raccolta di dati così estesa in termini di volume, velocità e varietà da richiedere tecnologie e metodi analitici specifici per l'estrazione di valori significativi. Molti sistemi sono sempre più costituiti e caratterizzati da enormi moli di dati da gestire, originati da sorgenti altamente eterogenee e con formati altamente differenziati, oltre a qualità dei dati estremamente eterogenei.

Un altro requisito in questi sistemi potrebbe essere il fattore temporale: sempre più sistemi hanno bisogno di ricevere dati significativi dai *Big Data* il prima possibile, e sempre più spesso l'input da gestire è rappresentato da uno stream di informazioni continuo. In questo campo si inseriscono delle soluzioni specifiche per questi casi chiamati *Online Stream Processing*.

L'obiettivo di questa tesi è di proporre un prototipo funzionante che elabori dati di *Instant Coupon* provenienti da diverse fonti con diversi formati e protocolli di informazioni e trasmissione e che memorizzi i dati elaborati in maniera efficiente per avere delle risposte in tempo reale. Le fonti di informazione possono essere di due tipologie: XMPP e Eddystone. Il sistema una volta ricevute le informazioni in ingresso (che contengono dati che riguardano il cliente vero e proprio e la locazione in cui egli si trova), estrapola ed elabora codeste fino ad avere dati significativi che possono essere utilizzati da terze parti. Lo *storage* di questi dati è fatto su Apache Cassandra. Il problema più grosso che si è dovuto risolvere riguarda il fatto che Apache Storm non prevede il ribilanciamento delle risorse in maniera automatica, in questo caso

specifico però la distribuzione dei clienti durante la giornata è molto varia e ricca di picchi. Il sistema interno di ribilanciamento sfrutta tecnologie innovative come le metriche e sulla base del throughput e della latenza esecutiva decide se aumentare/diminuire il numero di risorse o semplicemente non fare niente se le statistiche sono all'interno dei valori di soglia voluti.

Il primo capitolo si occupa di dare una panoramica generale sul mondo di *Big Data*, caratteristiche principali, dettagli funzionali, rami della società in cui è principalmente utilizzato e stime di investimenti futuri e stime sulle moli di dati su cui si andrà a lavorare nel futuro. Inoltre viene presentata anche la prima piattaforma di successo in ambito *Big Data* che ha influenzato con il suo approccio e la sua architettura le piattaforme più moderne: Apache Hadoop.

Nel secondo capitolo si parla in generale dell'elaborazione di grandi quantità di dati e dei diversi approcci possibili. In seguito si analizzano le diverse piattaforme di *Online Stream Processing* con particolare focus su Apache Storm che rappresenta la piattaforma di riferimento di questo lavoro di tesi. Vengono analizzati i suoi componenti architetturali, la sua struttura fisica, il parallelismo, approcci possibili in un discorso *Cloud*, e altre sottoparti secondarie.

Il terzo capitolo tratta quelle che sono le tecnologie principali utilizzate nella tesi oltre ad Apache Storm. In primis vengono visti Google Eddystone e XMPP e analizzati nello specifico in quanto sono utilizzati come formato per le informazioni che vengono trasmesse. Infine viene spiegato Apache Cassandra che rappresenta l'unità di memorizzazione del sistema.

Nel quarto capitolo viene mostrato in tutta la sua completezza ICoS (Instant Coupon on Storm): partendo dall'architettura fino ad arrivare all'implementazione vera e propria.

Il quinto capitolo è incentrato sui risultati sperimentali derivanti dai diversi test eseguiti su ICoS e sulle conseguenze e riflessioni in merito agli esiti degli esperimenti. Offre anche degli spunti sull'assegnazione statica di risorse iniziali, in modo da avere meno cambiamenti possibili in corso d'opera.

Nell'ultimo capitolo si traggono le conclusioni del lavoro di tesi effettuato, si analizzano i vari aspetti che hanno caratterizzano il suddetto progetto e si cerca di fare delle stime e delle previsioni su un lavoro futuro.

Capitolo 1 - Big Data & Apache Hadoop

1.1 Big Data

1.1.1 Introduzione

Big data è il termine usato per descrivere una raccolta di dati così estesa in termini di volume, velocità e varietà da richiedere tecnologie e metodi analitici specifici per l'estrazione di valore. Molti sistemi sono sempre più costituiti e caratterizzati da enormi moli di dati da gestire, originati da sorgenti altamente eterogenee e con formati altamente differenziati, oltre a qualità dei dati estremamente eterogenee. Il progressivo aumento della dimensione dei dataset è legato alla necessità di analisi su un unico insieme di dati, con l'obiettivo di estrarre informazioni aggiuntive rispetto a quelle che si potrebbero ottenere analizzando piccole serie, con la stessa quantità totale di dati. Ad esempio, l'analisi per sondare gli "umori" dei mercati e del commercio, e quindi del trend complessivo della società e del fiume di informazioni che viaggiano e transitano attraverso Internet. Big data rappresenta anche l'interrelazione di dati provenienti potenzialmente da fonti eterogenee, quindi non soltanto i dati strutturati, come i database, ma anche non strutturati. Con i big data la mole dei dati è dell'ordine dei Zettabyte, ovvero miliardi di Terabyte. Quindi si richiede una potenza di calcolo parallelo e massivo con strumenti dedicati eseguiti su decine, centinaia o anche migliaia di server.

Tipicamente le aree applicative che riguardano i Big Data sono: scenari smart city, applicazioni social, large-scale streaming information, data center e high-performance computing, sistemi e servizi mobili ecc.

I Big Data hanno tre origini principalmente:

1. Le transazioni originate dall'utilizzo dei dispositivi da parte degli individui, come carte di credito, cellulari e carte fedeltà;
2. Le interazioni e i messaggi che si sviluppano attraverso le reti sociali e il web 2.0;
3. Le rilevazioni svolte da innumerevoli sensori digitali presenti negli oggetti che utilizziamo quotidianamente.

I sistemi informativi richiedono una visione quality-aware che possa organizzare e rendere efficiente l'intero data lifecycle. 6V viene utilizzato per il nuovo processamento e trattamento dei dati:

- Volume: riferito alle grandi quantità di dati generati ogni secondo. Non si parla di Terabyte ma Zettabyte o Brontobyte. Se prendiamo tutti i dati digitali generati nel mondo tra l'inizio del tempo fino al 2008, la stessa quantità viene generata ogni minuto oggigiorno. I nuovi strumenti di Big Data utilizzano sistemi distribuiti in modo da poter memorizzare e analizzare i dati tra i database che sono sparsi in tutto il mondo.
- Variety: si riferisce ai diversi tipi di dati che si possono usare. In passato ci si è sempre concentrati solo su dati strutturati che si trovano in tabelle o database relazionali, come i dati finanziari. L'ottanta percento dei dati di tutto il mondo però è non strutturato (testo, immagini, video, voce, ecc) con tecnologie tipo Big Data si possono ora analizzare e portare insieme i dati di tipo diverso, come messaggi, conversazioni social media, le foto, i dati di sensore, video o registrazioni vocali;
- Velocity: s'intende la velocità con cui si processano i dati, ci sono diversi tipi di approcci, batch, micro-batching, real/near time, stream. Ci si riferisce anche alla velocità con cui i nuovi dati vengono generati (basti pensare semplicemente a quanto velocemente un tag diventa virale all'interno di un social network) e la velocità con cui si muovono.
- Value: s'intendono i valori che si riescono a estrarre dalle mole di dati da analizzare;
- Veracity: s'intende la "veridicità" dei dati, ossia la qualità dei dati intesa come il valore informativo che si riesce a estrarre;

- Variability: questa caratteristica può essere un problema; si riferisce alla possibilità di inconsistenza dei dati;

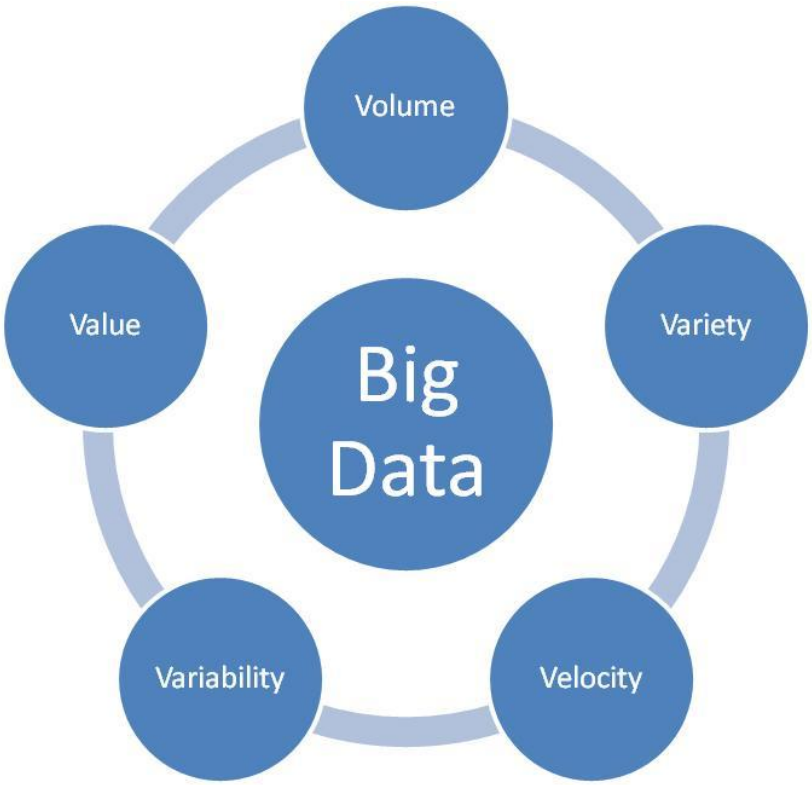


Figura 1.1 – le 5V di Big Data

Per quanto riguarda in specifico i Big Data si è passati da un modello delle 3V (Volume, Velocity, Variety) a uno a 5V e poi a 6V anche se non tutti in letteratura concordano su ciò.

I Big Data sono un argomento interessante per molte aziende, le quali negli ultimi anni hanno investito più di 15 miliardi di dollari, finanziando lo sviluppo di software per gestione dei dati e analisi. Questo è accaduto perché le economie più forti sono molto motivate all'analisi di enormi quantità di dati: basti pensare che ci sono oltre 4,6 miliardi di smartphone attivi e circa due miliardi di persone hanno accesso a internet. Il volume dei dati in circolazione si è evoluto come segue:

- Nel 1986 i dati erano 281 PetaByte;
- Nel 1993 i dati erano 471 PetaByte;
- Nel 2000 i dati erano 2,2 ExaByte;
- Nel 2007 i dati erano 65 ExaByte;
- Nel 2014 si è superato quota 650 ExaByte.

Gli investimenti in tematiche Big Data sono stati sui 6.3 miliardi di dollari nel 2012 e nel 2018 si prevede un investimento sui 48.3 miliardi di dollari con un incremento atteso annuo del 45% annuo, sia da parte di investimenti pubblici che privati.

Ogni minuto nel 2014 mediamente venivano mandate 204 milioni di e-mail, generate 1.8 milioni di "mi piace" su Facebook, pubblicati 278mila tweet su Twitter e caricate più di 200mila foto su Facebook, Google processa una media di 40mila query al secondo andando oltre 3.5 miliardi al giorno, su YouTube vengono caricati all'incirca 100 ore di video ogni minuto (ci vorrebbero 15 anni per guardare ogni video caricato dagli utenti in un giorno). L'ammontare di dati trasferiti tramite reti mobile è incrementato dell'81% al mese tra il 2012 al 2014, ed è in continuo aumento. Con il boom derivato dall'Internet of Things (IoT) i dispositivi collegati a internet sono a oggi sui tredici miliardi, ma supereranno quota cinquanta miliardi entro il 2020. Questo sono alcuni dei dati che rendono l'idea su cosa Big Data lavora e lavorerà. Uno degli esempi più recenti proviene da Tesco, la celebre catena di supermercati inglesi, che è riuscita a risparmiare 16 milioni di sterline all'anno monitorando le abitudini di acquisto dei suoi clienti e misurando l'efficienza delle offerte speciali.

1.1.2 Big Data e aree applicative

Le aree applicative di Big Data sono molteplici:

- Sicurezza e Law Enforcement: basta che si considerino tutti i video di sorveglianza, comunicazioni, recording di chiamate. Questi rappresentano milioni di dati al secondo con bassa densità di dati critici. In quest'area applicativa si tende a identificare pattern e relazioni fra sorgenti di informazioni dal volume imponente e molto numerose. US Government e IBM hanno un progetto insieme per high-performance analytics ad alta scalabilità su multimedia stream "in motion" di tipo eterogeneo;
- Prevenzioni di frodi: rilevazione delle frodi con analisi incrociate in real-time;
- e-Science: si parla di previsioni del meteo sempre più accurate, sincronizzazione della ricerca atomica, rilevamento di pericoli atmosferici ecc;
- Sistemi sanitari e salute: la potenza di calcolo di sistemi Big Data Analytic permette di trovare nuove cure e comprendere meglio e prevedere modelli di malattia. Si possono usare tutti i dati da smartwatch e dispositivi wearable per comprendere meglio i legami tra gli stili di vita e le malattie. Ciò permettono inoltre di monitorare e prevedere epidemie e focolai di malattie, semplicemente quello che le persone pubblicano/dicono/sentono;

- Smart Grid & Energy: si parte da concetti di micro-generazione di energia e difficoltà di storage, si possono creare per esempio di sistemi elettrici di monitoraggio smart grid per la prevenzione dei black-out;
- Smart City & Trasporti: Big Data viene utilizzato per migliorare molti aspetti delle città e paesi. Ad esempio, consente alle amministrazioni comunali di ottimizzare i flussi di traffico sulla base di informazioni sul traffico in tempo reale, così come i social media e dati meteo. Un certo numero di città stanno attualmente utilizzando grandi analisi dei dati con l'obiettivo di trasformarsi in Smart Cities, in cui i processi di infrastrutture di trasporto e di utilità sono tutti uniti;
- Telefonia;
- Supermercati: analisi dei dati di mercato e dei profili dei clienti per ad esempio instant coupon;
- Aumento delle performance sportive: la maggior parte degli sport d'elite hanno ormai abbracciato il campo del Big Data. Molte utilizzano video analytics per monitorare le prestazioni di ogni giocatore in una partita di calcio o di baseball, la tecnologia del sensore è costruito in attrezzature sportive come palloni da basket o mazze da golf, e molte squadre sportive d'elite tracciano i loro atleti al di fuori dell'ambiente sportivo - grazie alla tecnologia intelligente per monitorare la nutrizione e il sonno, così come le conversazioni social media per monitorare il benessere emotivo.

1.1.3 Evoluzione delle architetture per Big Data

Nel 2000, Seisint Inc. ha sviluppato un framework per il file-sharing distribuito basato su C++ per l'archiviazione e la ricerca dei dati. Il sistema memorizzava dati strutturati, semi-strutturati, e dati non strutturati su più server. Gli utenti potevano creare query in un C++ modificato chiamato ECL, il quale utilizzava un metodo "apply schema on read" per dedurre la struttura dei dati memorizzati al momento della query. Nel 2004, LexisNexis ha acquisito Seisint Inc. e nel 2008 ha acquisito ChoicePoint Inc. e la loro piattaforma di elaborazione parallela ad alta velocità. Le due piattaforme sono state fuse nel sistema HPCC il quale nel 2011 era open source sotto la licenza Apache v2.0. Attualmente, HPCC e Quantcast File System sono le uniche piattaforme a disposizione del pubblico in grado di analizzare quantità superiori agli exabyte di dati.

Nel 2004, Google ha pubblicato un documento su un framework chiamato MapReduce. Tale framework fornisce un modello di elaborazione parallela e un’implementazione associata per

elaborare enormi quantità di dati. Con MapReduce, le query sono divise e distribuite attraverso i nodi paralleli e trattati in parallelo (Map step). I risultati vengono poi raccolti e consegnati (Reduce step). Il framework è stato un grande successo, tanto che in molti volevano replicare l'algoritmo. Pertanto, un'implementazione del framework MapReduce è stato adottato da un progetto Apache open-source chiamato Hadoop.

Recenti studi dimostrano che l'uso di un'architettura multistrato è un'opzione per trattare con i big data. L'architettura parallela distribuita ripartisce i dati su più unità di elaborazione, e le unità di elaborazione parallela forniscono dati molto più velocemente, migliorando la velocità di elaborazione. Questo tipo di architettura inserisce i dati in un DBMS parallelo, che implementa l'uso di framework come MapReduce e Hadoop. Questi tipi di framework cercano di rendere la potenza di elaborazione trasparente all'utente finale utilizzando un server front-end.

Un'applicazione per il big data analytics può essere basata su un'architettura 5C (connessione, conversione, cyber, cognizione e configurazione). La mole di dati consente a un'organizzazione di spostare l'attenzione dal controllo centralizzato di un modello condiviso per rispondere alle dinamiche mutevoli di gestione delle informazioni. Questo consente una rapida separazione dei dati riducendo così il tempo di overhead.

1.1.4 Tecnologie e Big Data

I big data richiedono tecnologie eccezionali per elaborare in modo efficiente grandi quantità di dati all'interno di lassi di tempo tollerabili. Un rapporto di McKinsey del 2011 suggerisce tecnologie appropriate che includono A/B testing, crowdsourcing, la fusione e integrazione di dati, algoritmi genetici, apprendimento automatico, elaborazione del linguaggio naturale, elaborazione del segnale, simulazione, analisi di serie temporali e visualizzazione. I big data multimediali possono anche essere rappresentati come tensori, che possono essere più efficacemente gestite mediante calcolo tensor-based, come l'apprendimento sottospaziale multilineare. Le tecnologie aggiuntive possono essere applicato a big data inclusi database con elaborazioni in parallelo massicce (MPP), applicazioni di search-based, data mining, file system distribuiti, database distribuiti, infrastrutture cloud-based.

Alcuni ma non tutti i database relazionali MPP hanno la capacità di memorizzare e gestire petabyte di dati. Implicita è la possibilità di caricare, monitorare, eseguire il backup, e ottimizzare l'uso di grandi tabelle di dati in RDBMS.

Gli utenti di big data analytics sono generalmente ostili allo storage condiviso lento, preferendo Direct-Attached Storage (DAS) nelle sue varie forme da solid state drive (SSD) a dischi a elevata capacità come SATA all'interno dei nodi di elaborazione in parallelo. La percezione dell'architettura delle aree di memoria condivise - storage area network (SAN) e network-attached storage (NAS) - è che sono relativamente lente, complesse e costose. Queste qualità non sono coerenti con i grandi sistemi di analisi dei dati che prosperano sulle prestazioni dei sistemi, delle infrastrutture, e sul basso costo.

La consegna di informazioni in tempo reale o quasi è una delle caratteristiche distintive di big data analytics. La latenza è quindi da evitata quando e dove possibile. Il costo di una SAN necessaria per applicazioni di analisi è molto più elevato rispetto ad altre tecniche di memorizzazione. Gli utenti di big data analytics preferiscono sempre meno però lo storage condiviso.

1.2 Apache Hadoop

1.2.1 Introduzione

Apache Hadoop è un framework che supporta applicazioni distribuite con elevato accesso ai dati sotto una licenza libera. Sviluppato con tecnologia Java, è stato concepito per scrivere facilmente applicazioni che elaborano 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). Apache Hadoop è stato praticamente il primo sistema per l'analisi di Big Data di successo, e tutti i sistemi che sono venuti successivamente prendono ispirazione dalla sua architettura e da come processa i dati in parallelo.

Hadoop nacque per sopperire a un grave problema di scalabilità di Nutch, un crawler open source basato sulla piattaforma Lucene di Apache. I programmatori 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 il quale 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.

Per garantire queste caratteristiche, Hadoop utilizza numerosi macro-sistemi tra cui HDFS (Hadoop Distributed File System), un file system distribuito, progettato appositamente per immagazzinare un'enorme quantità di dati, in modo da ottimizzare le operazioni di archiviazione e accesso a un ristretto numero di file di grandi dimensioni, ciò a differenza dei tradizionali file system che sono ottimizzati per gestire numerosi file di piccole dimensioni.

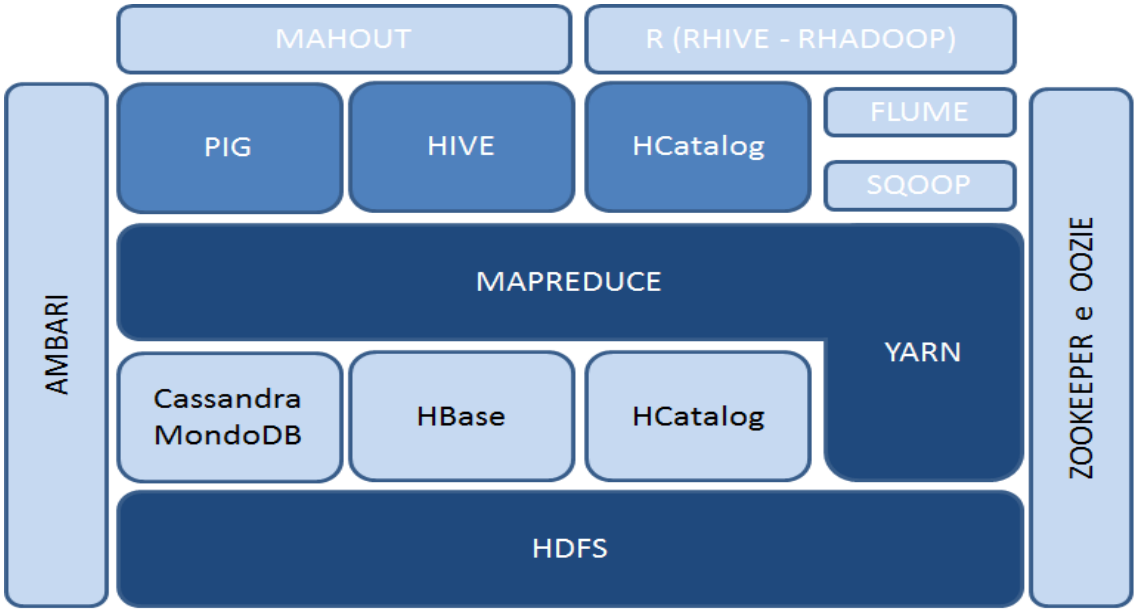


Figura 1.2 – Ecosistema Hadoop

1.2.2 Caratteristiche di Hadoop

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. Per comprendere meglio l'utilità di questi vantaggi, confrontiamo le peculiarità di storage e data management offerte da Hadoop con quelle offerte da un "classico" RDBMS.

RDMS	**Hadoop**
Schema *on Write*: lo schema dei dati deve	Schema *on Read*: i dati sono

essere creato prima che i dati stessi vengano caricati	semplicemente copiati nel file system, nessuna trasformazione è richiesta
Ogni dato da caricare deve essere trasformato nella struttura interna del database	I dati delle colonne sono estratte durante la fase di lettura
Nuove colonne devono essere aggiunte esplicitamente prima che i nuovi dati per tali colonne siano caricate nel database	I nuovi dati possono essere aggiunti ed estratti in qualsiasi momento

Tabella 1.1 – Confronto tra RDBMS e Hadoop

1.2.3 HDFS: Architettura e funzionamento

HDFS è un file system distribuito ideato per soddisfare requisiti quali affidabilità e scalabilità ed è in grado di gestire un numero elevatissimo di file, anche di dimensioni ragguardevoli (dell'ordine dei gigabyte o terabyte), attraverso la realizzazione di cluster che possono contenere migliaia di nodi. 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: avvengono scegliendo i nodi più vicini al client che effettua la lettura e, ovviamente, in presenza di dati ridondati 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.

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.

1.2.4 MapReduce

MapReduce è un framework per la creazione di applicazioni in grado di elaborare grandi quantità di dati in parallelo basandosi sul concetto di functional programming. A differenza della programmazione multithreading, in cui i thread condividono i dati oggetto delle elaborazioni presentando così una certa complessità proprio nel coordinare l'accesso alle risorse condivise, nel functional programming, invece, la condivisione dei dati è eliminata, e questi sono passati tra le funzioni come parametri o valori di ritorno.
MapReduce lavora secondo il principio del *divide et impera*, suddividendo l'operazione di calcolo in diverse parti processate in modo autonomo. Una volta che ciascuna parte del problema è stata calcolata, i vari risultati parziali sono "ridotti" (cioè ricomposti) a un unico risultato finale. È MapReduce stesso che si occupa dell'esecuzione dei vari task di calcolo, del loro monitoraggio e della ripetizione dell'esecuzione in caso si verifichino problemi.
Il framework lavora attraverso i compute node cioè dei nodi di calcolo che si trovano assieme ai DataNode di HDFS: infatti, lavorando in congiunzione con HDFS, MapReduce può eseguire i task di calcolo sui nodi dove i dati sono già presenti, aumentando così l'efficienza di calcolo.
Fondamentalmente, un job MapReduce è costituito da quattro componenti:

- I dati di input, su HDFS;
- Una funzione map, che trasforma i dati di input in una serie di coppie chiave/valore;

- Una funzione reduce che, per ogni chiave, elabora i valori a essa associati e crea, come output, una o più coppie chiave valore. L'esecuzione della funzione reduce è preceduta da una fase di raccolta delle coppie chiave/valore prodotte dalla funzione map. Le coppie sono ordinate per chiave e i valori con la stessa chiave sono raggruppati;
- L'output, scritto su un file HDFS.

MapReduce si presta bene all'esecuzione di numerose operazioni sui dati. Data la varietà di operazioni che MapReduce può effettuare in modo efficiente, i potenziali utilizzi del framework sono moltissimi: creazione di liste di parole da documenti di testo, indicizzazione e ricerca. Appartengono a questa categoria i seguenti esempi applicativi: conteggi, somme, estrazione di liste univoche di valori (ad esempio analisi dei log dei Web server) e applicazioni di filtri ai dati; analisi di strutture dati complesse, come grafi (ad esempio per applicazioni di social network analysis); data mining e machine learning; esecuzione di task distribuiti (come calcoli matematici complessi e analisi numeriche); correlazioni, operazioni di unione, intersezione, aggregazione e join (ad esempio analisi di mercato, analisi predittive e previsione dei trend).

Queste numerose applicazioni confermano la potenza informativa che è possibile estrarre con la Big Data analytic, grazie alla quale non ci si limita più a un business intelligence orientata all'analisi dati strutturati per trarre conclusioni a posteriori. Grazie alla varietà dei dati che è possibile analizzare, all'enorme quantità che è possibile immagazzinare con le moderne tecnologie e allo stesso tempo alla velocità che non è più un limite nell'analisi di enormi moli di informazioni, si possono elaborare modelli di analisi che possono fornire predizioni e non limitarsi ad analisi descrittive.

Capitolo 2 - Online Stream Processing & Apache Storm

2.1 Online Stream Processing

2.1.1 Presentazione

La domanda di stream processing è in aumento. Quantità immense di dati devono essere elaborati velocemente da una serie sempre crescente di fonti di dati disparate. Questo spinge ai limiti le infrastrutture di elaborazione tradizionali. Le applicazioni stream-based includono il commercio, i social network, internet of things, il monitoraggio di sistemi, e molti altri esempi. Un certo numero di piattaforme open source sono emerse per affrontare questo problema che può essere risolto in modi diversi, anche se talvolta molti approcci sono simili.

Per online stream processing s'intende l'analizzare e il processamento di grandi quantità di dati provenienti da un streaming in tempo reale. Essenziale per lo streaming processing è Streaming Analytics, o la capacità di calcolare continuamente analisi matematiche o

statistiche al volo all'interno del flusso. Soluzioni di stream processing sono progettate per gestire grandi volumi in tempo reale con un'architettura scalabile, altamente disponibile e fault tolerant.

Parlando invece in generale ci sono tre approcci principali per elaborazione di grandi quantità di dati: elaborazione batch, stream processing e micro-batching.

Un sistema di elaborazione batch ha le seguenti caratteristiche:

- Processamento di dati in massa;
- Accesso a tutti i dati;
- Possibili operazioni di calcolo grandi e complesse;
- Latenza alta (nell'ordine di minuti);
- Si è più interessati a un throughput che alla latenza;
- Tipico esempio è MapReduce di Hadoop.

Un sistema per lo stream processing ha seguenti caratteristiche:

- Un modello di elaborazione one-at-a-time;
- I dati vengono elaborati immediatamente dopo l'arrivo;
- Le operazioni di calcolo sono relativamente semplici e generalmente indipendenti fra di loro;
- Latenza nell'ordine dei sub-secondi;
- Difficoltà nel mantenere lo stato in modo efficiente;
- Esempio tipico è Apache Storm Core.

Infine, i sistemi Micro-batching:

- Rappresentano un caso speciale di elaborazione batch, con batch di dimensione veramente piccola;
- Mix tra il batching e streaming;
- Latenza nell'ordine di secondi;
- Tipico esempio è Spark Streaming.

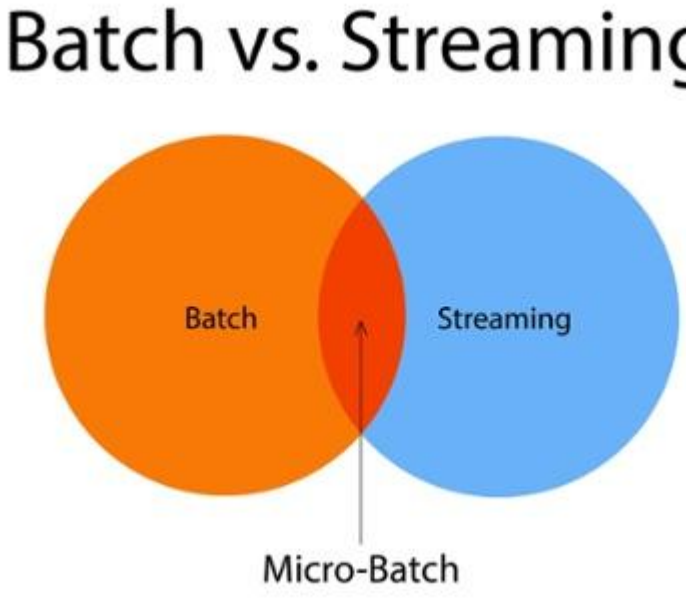


Figura 2.1 – Confronto fra i diversi tipi di approccio

Una soluzione per l'online stream processing deve risolvere diverse sfide tra cui:

- L'elaborazione di grandi quantità di streaming events;
- Risposta in tempo reale
- Prestazioni e scalabilità (i volumi di dati aumentano di dimensioni e complessità);
- Integrazione rapida con infrastrutture esistenti;
- La produttività degli sviluppatori in tutte le fasi del ciclo di vita di sviluppo delle applicazioni, offrendo dei buoni tool di supporto e sviluppo rapido;
- Comunità attiva;
- Alerting.

Per quanto riguarda l'elaborazione di dati ad alta velocità si dispone di due diversi use cases di elaborazione: Distributed Stream Processing (SP) o talvolta chiamato Event Stream Processing (ESP) e Distributed Complex Event Processing (CEP).

SP/ESP è stateless, l'elaborazione dei dati in ingresso è fatto in modo distribuito con "continue query" (vede in sostanza che i dati che arrivano siano con determinati parametri). Questo avviene senza alcun I/O o data storage. Sui flussi di dati si possono applicare trasformazioni, join, filtraggio, ecc secondo topologia definita. Lo stato finale è di solito memorizzato da qualche parte per ulteriori elaborazioni. Caso d'uso tipico è il calcolo continuo e analisi matematiche o statistiche al volo all'interno dello stream. Soluzioni di stream processing sono

progettati per gestire grandi volumi di dati in tempo reale con una soluzione scalabile, altamente disponibile e un'architettura fault tolerant.

CEP è stateful, micro o mini raggruppamenti di dati che coinvolgono elaborazione complessa di dati in cui è coinvolto anche il mantenimento dello stato. Lo stato è persistente, non deve essere necessariamente memorizzato in un disco rigido, potrebbe essere organizzato in memoria o altro. CEP è ottimizzato per elaborare gli eventi discreti in topologie definite. Classico esempio può essere il trend dei tweet Twitter secondo hashtag definiti per una determinata finestra di tempo.

Non possiamo prendere questi approcci come qualcosa di completamente distinto. Un sistema CEP può essere costruito su di un sistema ESP. Apache Storm Core è l'ideale per ESP, ma con l'astrazione Trident riesce anche a soddisfare CEP.

2.1.2 Apache Flume

Flume è un servizio distribuito affidabile per la raccolta in modo efficiente, l'aggregazione, e lo spostamento in generale di grandi quantità di dati di log. Esso ha un'architettura semplice e flessibile basato su streaming data flows. E' robusto e fault tolerant con meccanismi di affidabilità sintonizzabili e molti meccanismi di failover e recovery. Esso utilizza un semplice modello di dati estensibile che consente di fare applicazione analytics online.

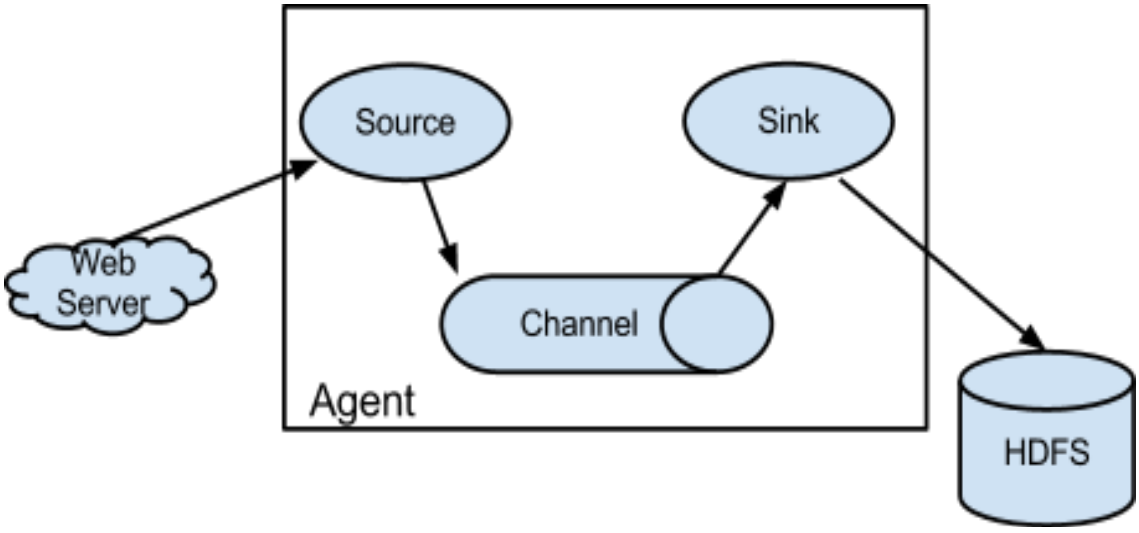


Figura 2.2 – Funzionamento generale di Apache Flume

L'uso di Apache Flume non è solo limitato all'uso sui dati di log. Dal momento che le fonti di dati sono personalizzabili, Flume può essere utilizzato per il trasporto di grandi quantità di

dati generati da eventi come ad esempio dai social-media, messaggi e-mail e praticamente qualsiasi fonte possibile di dati.

Un Event è un'unità di dati che viaggia attraverso un agente Flume. I flussi di Event si muovono attraverso Source poi Channel e poi al Sink, ed è rappresentato da un'implementazione dell'interfaccia Event. Un Event trasporta un payload che è accompagnato da un set opzionale di intestazioni (attributi sotto forma di String). Un agente Flume è un processo (JVM) che ospita i componenti che permettono agli Event di viaggiare da una sorgente esterna a una destinazione esterna.

Una Source consuma eventi avente un formato specifico, e quelli eventi sono consegnati alla Source da una fonte esterna, come ad esempio un server web. Quando una Source riceve un evento, lo memorizza in uno o più canali. Il Channel è uno store passivo che contiene l'evento fino a quando quell'evento è consumato da un Sink. Un tipo di canale disponibile in Flume è il FileChannel che utilizza il file system locale come il suo archivio di backup. Un Sink è responsabile per la rimozione di un evento dal Channel e la messa in un repository esterna come HDFS (nel caso di un HDFSEventSink) o il suo inoltro al prossimo Source che rappresenta il next hop del flusso. La Source e Sink funzionano in maniera asincrona.

Vantaggi nell'utilizzo di Apache Flume:

- Utilizzando Apache Flume possiamo memorizzare i nostri dati su qualsiasi store centralizzato (HBase, HDFS);
- Quando il tasso di dati in ingresso supera la velocità con cui i dati possono essere scritti in output, Flume funge da mediatore tra i produttori di dati e gli store centralizzati e fornisce un flusso costante di dati tra loro;
- Utilizzando Flume si è in grado di ottenere i dati da più server e trasportarli immediatamente in Hadoop;
- Supporta un set molto ampio di tipologie di sorgenti e destinazioni;
- Supporta la scalabilità orizzontale;

Il principale svantaggio nell'utilizzo di Apache Flume risiede nel fatto che esso non è ottimizzato per il real-time stream-processing, ma è costruito per supportare principalmente il migraggio di dati da delle sorgenti che possono essere dei server web in piattaforme per l'analisi come Apache Hadoop.

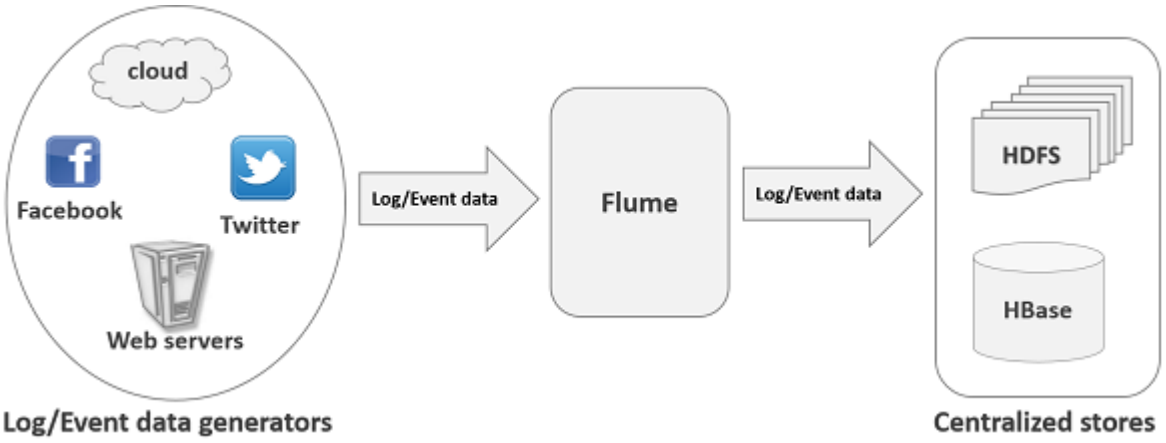


Figura 2.3 – esempio di utilizzo di Apache Flume

2.1.3 Apache Storm

Apache Storm è un sistema open source distribuito per computazioni in real-time. Storm rende facile lavorare in modo affidabile gli stream illimitati di dati, facendo per l'elaborazione in tempo reale quello che ha fatto Hadoop per l'elaborazione batch. E' utilizzato da colossi come Yahoo, twitter, Spotify, Flipboard, Baidu ecc. Storm ha molti casi d'uso: l'analisi in tempo reale, machine learning online, calcolo continuo, RPC distribuito, ETL, e altro ancora. Storm è veloce, scalabile, fault-tolerant. Una topologia Storm consuma flussi di dati e processa tali flussi in modi arbitrariamente complessi, ripartizionandoli tra ogni fase del calcolo se necessario.

Prima di Storm, per fare online stream processing usando solamente Hadoop, i dati venivano salvati in HDFS processati secondo lo schema Map/Reduce e in seguito risalvati in HDFS generando così della latenza causato dalle code di stream di dati che aspettano di essere salvati e processati, Storm propose un approccio innovativo basato sui concetti di Topologia, Spout e Bolt. Per stream s'intende una sequenza di tuple illimitata, si considerino spout le sorgenti degli stream, e per bolt invece s'intendono i processi che eseguono gli stream che arrivano in input e ne producono altri in output dopo aver applicato allo stream d'ingresso delle funzioni di filtraggio/join/trasformazioni ecc. Le topologie sono rappresentate dalle diverse combinazioni che si possono fare tra i bolt e gli spout formando così dei grafi che possono essere ciclici.

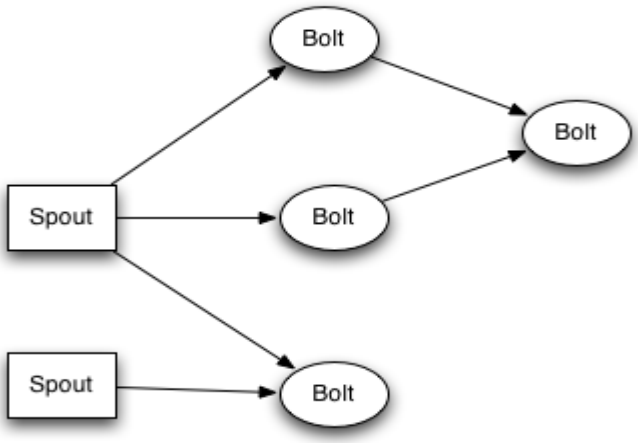


Figura 2.4 – Funzionamento generale di Apache Storm

I principali vantaggi che espone Apache Storm sono:

- Modello di programmazione semplice: simile all'approccio di MapReduce per ridurre la complessità dell'elaborazione di batch in parallelo;
- Esegue qualsiasi linguaggio di programmazione: clojure, java, ruby, python sono supportate come predefinite. Il supporto per gli altri linguaggi può essere aggiunti mediante l'implementazione di un semplice protocollo di comunicazione di Storm;
- Fault-tolerant: Storm gestisce i processi e i fallimenti dei nodi;
- Scalabilità orizzontale: l'elaborazione può essere estesa in parallelo usando thread, processi e server aggiuntivi;
- Processamento dei messaggi garantito: viene garantito che ogni messaggio sarà elaborato almeno una volta. Si occupa di riprodurre i messaggi dalla sorgente quando un task non riesce;
- Veloce: il sistema è progettato in modo che i messaggi vengono elaborati rapidamente e utilizza ØMQ come coda per i messaggi;
- Modalità locale: Storm ha una modalità "locale" dove si simula completamente un cluster Storm in-process. Ciò consente di sviluppare e testare topologie molto velocemente;
- Debug in Java/Eclipse;
- Comunità molto attiva.

Gli unici svantaggi che ci sono nell'utilizzo di Apache Storm sono rappresentati dal fatto che il load balancing non è automatico e che in generale è lasciato tanto lavoro da fare all'utente. Rimane comunque la piattaforma di riferimento per il real-time stream processing.

2.1.4 Apache S4

Apache S4 (Simple Scalable Streaming System) è un sistema general-purpose, distribuito, scalabile, parzialmente fault-tolerant con una piattaforma che permette ai programmatori di sviluppare facilmente applicazioni per l'elaborazione dei flussi continui di dati illimitati. Viene utilizzata per analisi di dati real time (tweet, dati finanziari, notizie), social network, ricerche real time, processa mento di eventi complessi ecc.

Gli eventi contenenti dati con chiave vengono indirizzati con affinità al Processing Element (PE), che consumano gli eventi e eseguono uno o entrambi i seguenti passi: emettere uno o più eventi che possono essere consumati da altri PE, e/o pubblicare i risultati. L'architettura assomiglia al modello con Attore che fornisce una semantica d'incapsulamento e di trasparenza consentendo alle applicazioni di essere concorrenti esponendo una semplice interfaccia di programmazione per gli sviluppatori di applicazioni.

Utilizzando S4 si hanno i seguenti vantaggi:

- Decentralizzazione: tutti i nodi sono simmetrici senza nessun servizio centralizzato e nessun point of failure. Questo semplifica le implementazioni e le modifiche di configurazione dei cluster;
- Scalabilità: se il throughput aumenta linearmente allora i nodi possono venire aggiunti al cluster. Non vi è alcun limite predefinito al numero di nodi supportati;
- Estendibilità: le applicazioni possono essere scritte e distribuite utilizzando delle semplici API. Dei Building block della piattaforma possono essere sostituiti da delle implementazioni personalizzate;
- Fault-tolerance: quando un server va in down, automaticamente uno stand-by server viene attivato. Checkpoint e recuperi minimizzano le perdite di stato;
- Concettualmente molto potente;
- Fa parte di Apache Incubation;
- Interfaccia client build-in;
- Load balancing automatico.

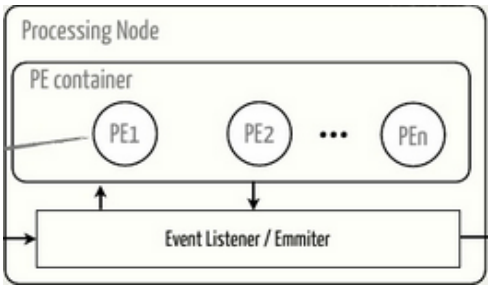


Figura 2.5 – Architettura generale di Apache S4

Chi sceglie di lavorare con Apache S4 deve mettere in conto che c'è una configurazione molto complessa da fare, potenzialmente ci può essere perdita di dati, e il debug è molto lento.

2.1.5 Apache Samza

Apache Samza è un framework di stream processing distribuito. Esso utilizza Apache Kafka per la messaggistica, e YARN per fornire tolleranza ai guasti, sicurezza e gestione delle risorse. Il progetto mira a fornire un framework per lo stream processing asincrono quasi in real-time. Samza è stato costruito insieme a Kafka che originariamente erano entrambi di LinkedIn.

I concetti base dell'architettura di Samza sono:

- Lo stream: composto da messaggi immutabili di un tipo o categoria simile. Ad esempio, un flusso potrebbe essere tutti i clic su un sito Web, o tutti gli aggiornamenti di una particolare tabella di database, o tutti i log prodotti da un servizio, o qualsiasi altro tipo di dati di evento;
- Gli jobs: rappresenta il codice che esegue una trasformazione logica su una serie di flussi di input per aggiungere i messaggi di output allo stream in uscita;
- Le partizioni: ogni stream è diviso in una o più partizioni. Ogni partizione nel flusso è una sequenza totalmente ordinata di messaggi.
- I tasks: un job è ridimensionato scomponendolo in più task. Il task è l'unità di parallelismo del job, così come la partizione è al flusso. Ogni task consuma i dati da una partizione per ogni dei job che consuma lo stream in entrata.

Abbiamo poi uno streaming layer responsabile di fornire flussi partizionati che vengono replicati e resi duraturi; un execution layer che è responsabile per la pianificazione e le funzioni di coordinamento fra le macchine e un processing layer che si occupa del trattamento del flusso di input e delle trasformazioni che vengono applicate ai dati. L'effettiva attuazione dello streaming execution layer è pluggable. Implementazione streaming può essere fornita

tramite una delle implementazioni esistenti: Kafka (topics) o Hadoop (una directory di file in HDFS) o un database (tabella). Samza ha un supporto in-built per Apache YARN e Apache Kafka.

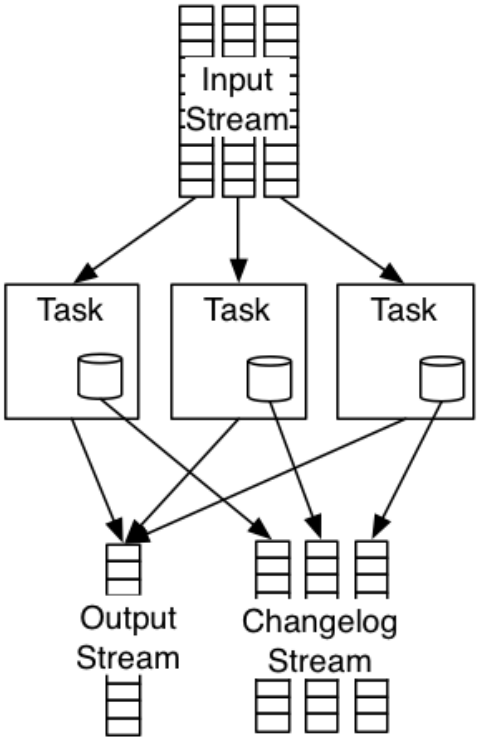


Figura 2.6 – Funzionamento generale di Apache Samza

Tra le sue caratteristiche ci sono:

- API semplici;
- Managed state: Samza gestisce lo snapshotting e il ripristino dello stato di uno stream processing. Quando il processo viene riavviato, Samza ripristina il suo stato con snapshot corretto;
- Fault tolerant: Ogni volta che una macchina del cluster va in down, Samza lavora con YARN in modo trasparente per migrare le attività in un'altra macchina;
- Durability: Samza utilizza Kafka per garantire che i messaggi vengono elaborati nell'ordine in cui sono stati scritti in una partizione, e che nessun messaggio venga mai perduto;

- Scalabilità: Samza viene partizionato e distribuito a tutti i livelli. Kafka fornisce stream ordinati, partizionati, replicabili, con fault-tolerant. YARN fornisce un ambiente distribuito per l'esecuzione del container Samza;
- Pluggable: Samza funziona anche senza Kafka e YARN, fornendo API che consentono di eseguire Samza con altri sistemi di messaggistica e di ambienti di esecuzione;
- Isolamento processore: Samza funziona con Apache YARN, che supporta il modello di sicurezza di Hadoop, e l'isolamento delle risorse attraverso Linux CGroups.

Il più grande svantaggio di Samza è che comunque ha bisogno di architettura come Kafka, o che simili a esso per funzionare. Successivamente c'è da considerare che è tra i più recenti framework open-source che si occupa di stream processing e questo ovviamente non può essere considerato un vantaggio.

2.1.6 Apache Spark Streaming

Apache Spark è un framework generale per l'elaborazione dei dati su larga scala che supporta tanti diversi linguaggi di programmazione e concetti come MapReduce, processamento in-memory, stream processing, elaborazione grafica e machine learning.

Spark Streaming è un'estensione di Spark che sfrutta la capacità di programmazione veloce del suo Core per eseguire analisi in streaming. Prende dati in mini-batch (micro-batching) ed esegue trasformazioni RDD su essi. Questo design permette che lo stesso insieme di codice applicativo scritto per l'analisi di batch si possa utilizzare per analisi in streaming, su un unico engine.

Il problema principale di Spark streaming è che non è stato costruito per supportare al massimo dell'efficienza lo stream processing ma si adatta di più ad un modello di micro-batching.

2.1.7 Apache Flink

Apache Flink è un framework open source gestito dalla comunità Apache per Big Data Analytics. Il nucleo di Apache Flink è concentrato per gestire flusso di dati in streaming scritto in Java e Scala. Ha lo scopo di colmare il divario tra i sistemi di MapReduce o simili e sistemi di DB in parallelo. Il sistema runtime in pipeline di Flink consente l'esecuzione di programmi di bulk/batch e stream processing. Inoltre, il runtime di Flink supporta

nativamente l'esecuzione di algoritmi iterativi. I programmi in Flink possono essere scritti in Java o Scala e vengono compilati automaticamente e ottimizzati in modo tale che i dati vengono eseguiti in un ambiente cluster o cloud. Flink non fornisce un proprio sistema di archiviazione dei dati, dati di input devono essere conservati in un sistema di storage distribuito come HDFS o HBase. Per l'elaborazione del flusso di dati, Flink consuma i dati da code di messaggi (affidabile) come Kafka.

Il principale vantaggio di Flink è che permette in uno stesso programma l'elaborazione batch e lo stream processing secondo un'architettura a Lambda, ovviamente questo può essere visto anche come uno svantaggio se si vuole solo il massimo dell'efficienza sviluppata per stream processing.

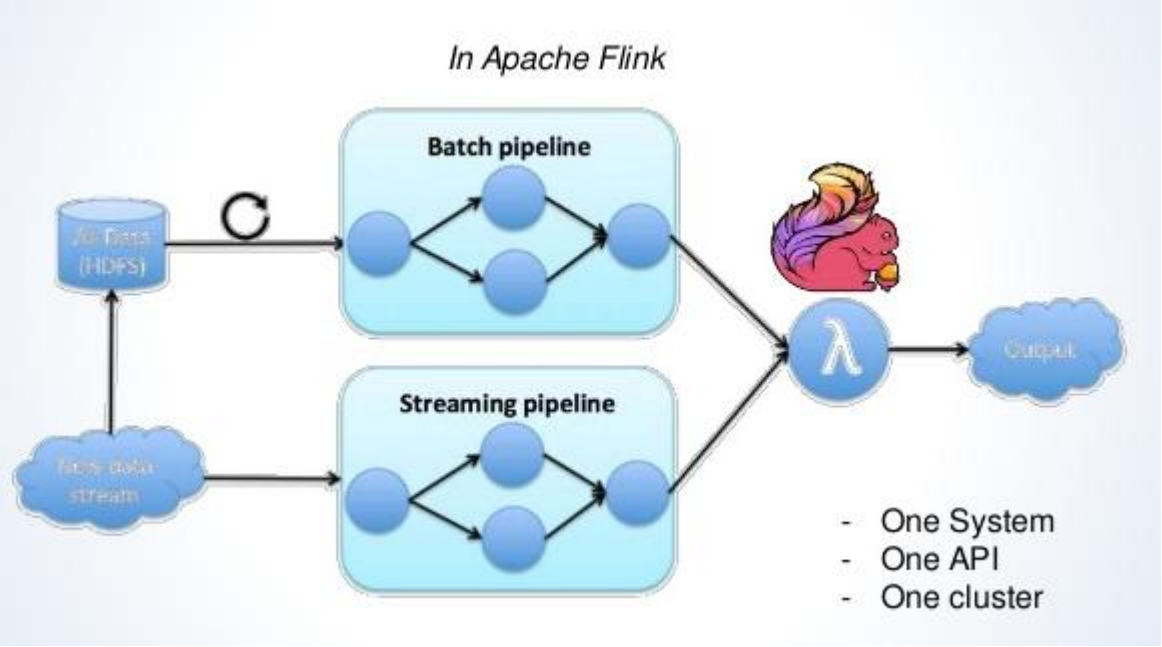


Figura 2.7 – Funzionamento di Flink secondo architettura Lambda

2.2 Apache Storm

2.2.1 Architettura

Storm è un sistema di computazione real-time, distribuito, gratuito e open source che rende semplice lavorare in maniera affidabile con flussi di dati illimitati, effettuando l'elaborazione in tempo reale alla stessa stregua dell'elaborazione che sistemi come Hadoop effettuano in batch. Storm è stato originariamente creato Nathan Marz e il suo team presso BackType la quale è una società di analisi sociale. In seguito, Storm è stata acquisita e fatta diventare open-

source da Twitter. In breve tempo, Apache Storm è diventato uno standard per il sistema di elaborazione in tempo reale distribuita. Inoltre è scritto in Java e Clojure.

Tipici casi di applicabilità di Storm sono:

- Stream Processing: elaborazione di nuovi flussi di dati o aggiornamento di dati memorizzati in database in tempo reale. A differenza dei tradizionali approcci di stream-processing con reti di code e worker, Storm è fault-tolerance e scalabile.
- Continuous Computation: interrogazioni continue e presentazione dei risultati in real-time. Un esempio è l'analisi dei trend dei topic su Twitter: i risultati dell'elaborazione di uno stream di tweet, che mostra gli argomenti di tendenza, verranno presentati in tempo reale.
- Distributed RPC: Storm può essere utilizzato per parallelizzare un'intensa interrogazione on the fly. L'idea si basa sull'utilizzo di remote procedure call distribuite che attendono messaggi di invocazione: quando si riceve una chiamata, una query viene elaborata e vengono restituiti i risultati. Un esempio di Distributed RPC è la parallelizzazione di query di ricerca.

Fondamentalmente Hadoop e Storm sono dei framework utilizzati per l'analisi dei Big Data. Entrambi si completano a vicenda e si differenziano per alcuni aspetti. Apache Storm fa tutte le operazioni ad eccezione di persistenza, mentre Hadoop si occupa di tutto, ma è in ritardo nel calcolo in tempo reale. La seguente tabella confronta gli attributi di Storm e Hadoop.

Storm	**Hadoop**
Processamento di stream in tempo reale	Processamento in batch
Stateless	Stateful
Architettura Master/Slave con ZooKeeper come coordinatore di base. Il nodo master è chiamato nimbus e i nodi slave supervisor.	Architettura Master/Slave con/senza Zookeper come coordinatore di base. Il nodo master si chiama job tracker e i nodi salve task tracker.
La mole di dati si aggira su decine di migliaia di messaggi al secondo su cluster.	Hadoop Distribuited File System (HDFS) utilizza il framework MapReduce per processare la mole di dati che può impiegare minuti o ore.

La topologia Storm esegue fino allo spegnimento da parte dell'utente o un fallimento irrecuperabile.	MapReduce esegue in ordine sequenziale e quando finisce di ferma.
Se nimbus/supervisor va in down, quando viene effettuato il riavvio, si continua da dove ci si era fermati, senza nessuna perdita.	Se il job tracker va in down, tutti i job in esecuzione vanno persi.

Tabella 2.1 – Confronto tra Storm e Hadoop

Tralasciando tutto quello che si è detto in precedenza su vantaggi/svantaggi di Storm, dal punto di vista logico, la computazione parallela e distribuita è svolta da diversi componenti, ognuno dei quali è responsabile di processare un semplice task specifico. Uno stream in input allo Storm Cluster è raccolto e gestito da un componente chiamato spout. Storm considera gli stream come infinite sequenze di tuple. Secondo Storm le tuple sono liste ordinate di oggetti, l'ordine è quello di arrivo delle informazioni. Uno spout, legge lo stream in ingresso e incapsula i dati in tuple.

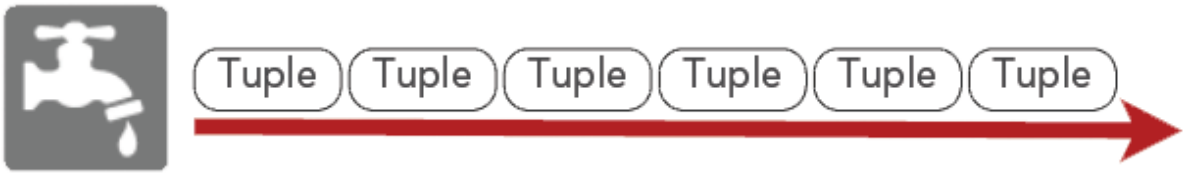


Figura 2.8 – Lettura dello stream in ingresso

All'estrema sinistra della figura proposta è stato inserito il simbolo che i progettisti di Storm utilizzano per rappresentare uno spout. Le tuple vengono passate a uno o più componenti chiamati bolt, che processano opportunamente queste tuple. A sua volta, un bolt può distribuire il risultato dell'elaborazione all'esterno dello Storm Cluster (quindi a un sistema di terze parti) oppure passarli a un altro bolt che effettuerà altre operazioni sui dati già elaborati.

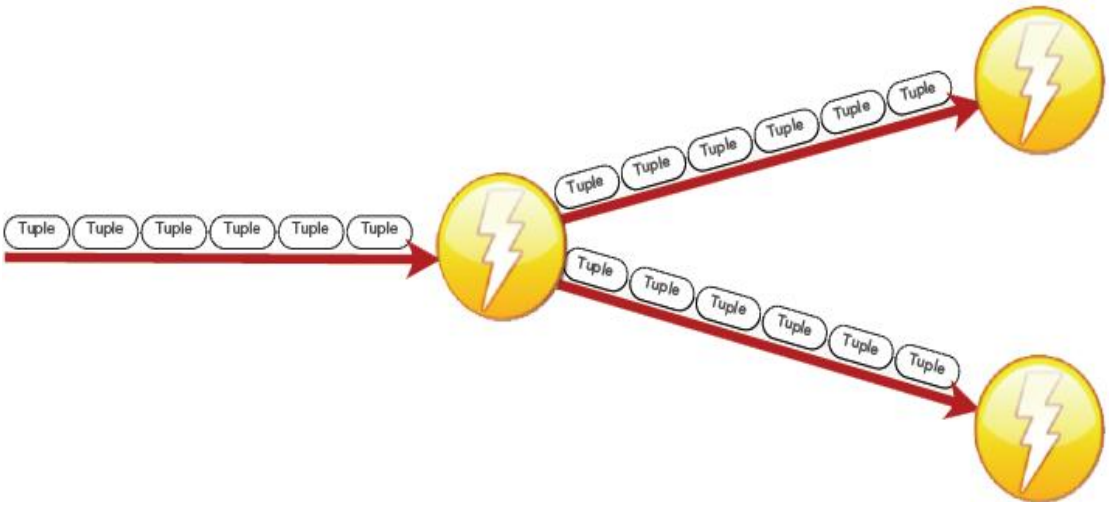


Figura 2.9 – Distribuzione del risultato dell'elaborazione

Il fulmine è il simbolo che i progettisti di Storm utilizzano per rappresentare un bolt. L'insieme di tutti i componenti e delle connessioni è chiamata topologia.

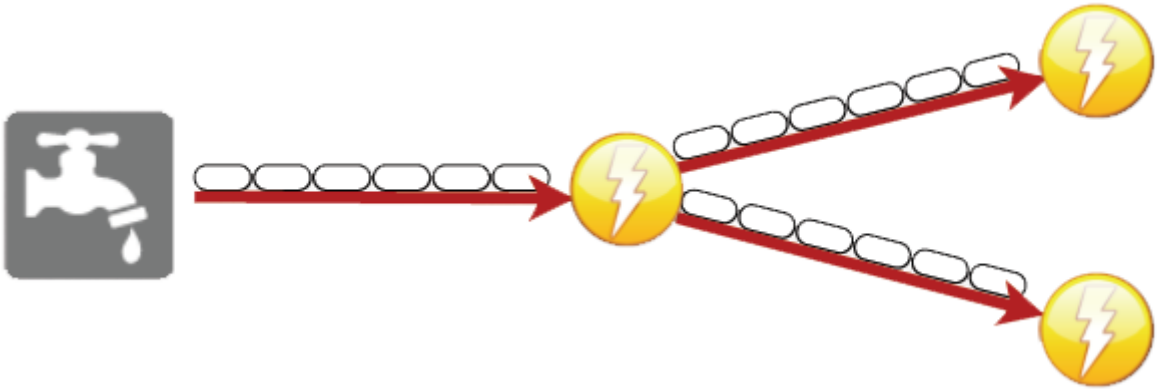


Figura 2.10 – Insieme di componenti e connessioni (topologia)

Nella figura proposta in precedenza è stata fornita la rappresentazione di una topologia costituita da uno spout e da bolt.

Un semplice esempio pratico potrebbe essere quello di essere di fronte ad un problema di streaming word count, cioè contare le occorrenze delle parole all'interno di uno stream continuo, e di voler aggiornare e presentare i risultati in real-time. Con Storm, è possibile progettare un topologia che contenga:

- Uno spout che emetta continuamente frasi (o le legga da una risorsa esterna e le inoltri);
- Un bolt che prenda in ingresso il testo delle frasi e ne estrapoli le parole;
- Un bolt che effettui il conteggio delle occorrenze e restituisca il risultato.

2.2.2 Topologia

Una caratteristica fondamentale da considerare quando si progetta una topologia, è definire come i dati sono scambiati tra i vari componenti, cioè come gli stream vengono consumati dai bolt.

A tal proposito si definisce uno stream grouping, una tecnica che specifica quali stream sono consumati da ogni bolt e come lo stream verrà consumato. Un nodo, infatti, può emettere più di uno stream di dati. Una tecnica di Stream Grouping, permette di scegliere quali stream ricevere. Esistono diversi tipi di Stream Grouping:

- Shuffle Grouping: ogni tupla emessa viene inviata a un bolt scelto a caso, garantendo però che ogni consumatore riceverà lo stesso numero di tuple;
- Fields Grouping: permette di controllare in che modo le tuple sono inviate ai bolt, basandosi su uno o più campi della tupla stessa. Garantisce che un dato set di valori per una certa combinazione di campi, sia sempre inviata allo stesso bolt;
- All Grouping: una singola copia di ogni tupla viene inviata a tutte le istanze dei bolt. Generalmente, questo tipo di grouping è usato per inviare segnali ai bolt. Ad esempio, se è necessario effettuare un refresh di una cache, è possibile inviare un refresh cache signal a tutti i bolt. In tale contesto, potrebbe essere necessario identificare i segnali e da quali sorgenti provengono: in questo caso Storm permette di identificare gli stream associandogli un nome;
- Custom Grouping: attraverso l'implementazione dell'interfaccia CustomStreamGrouping, è possibile implementare una tecnica di grouping personalizzata. Ciò fornisce il potere di decidere quali bolt riceveranno ogni tupla;
- Direct Grouping: con questo grouping, è la sorgente che decide quale componente riceverà la tupla;
- Global Grouping: permette di inviare le tuple generate da tutte le istanze di una sorgente a una singola istanza target;
- Partial Key Grouping: lo stream è partizionato dai campi specificati nel grouping, come Fields grouping, ma il carico è bilanciato tra due bolt, fornendo così un migliore utilizzo delle risorse quando i dati in ingresso sono non allineati;
- None Grouping: Questo grouping specifica che ci si deve curare di come il flusso è raggruppato. Attualmente, none grouping è equivalente a shuffle grouping.

2.2.3 Spout

Gli spout si limitano a prelevare un flusso da una certa sorgente e a inoltrarlo ai bolt. Dunque è possibile far riferimento a diverse architetture per decidere il modo più efficiente per prelevare i flussi dalla sorgente:

- Direct Connection: uno spout è direttamente connesso alla sorgente che emette un flusso. L'architettura è semplice da implementare e adatta quando la sorgente è nota e rimane costante. Viene considerata sorgente non nota se questa viene aggiunta dopo che una topologia è stata già messa in esecuzione. Quest'architettura è semplice da implementare, in particolare quando l'emettitore è un dispositivo ben noto o un gruppo di dispositivi noti. Un dispositivo si dice noto se lo è all'avvio e rimane lo stesso per tutta la durata della topologia. Un dispositivo sconosciuto è uno che viene aggiunto dopo che la topologia è in esecuzione. Un gruppo di dispositivi noto è quello in cui tutti i dispositivi del gruppo sono noti al momento dall'inizio (un esempio è la lettura di uno stream da Twitter);

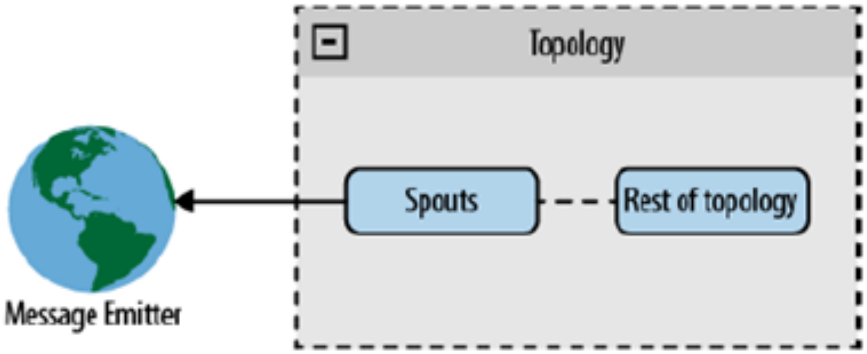


Figura 2.11 – Esempio di Direct Connection

- Direct Connection Hashing: questo tipo di architettura, è adatto quando esistono diverse sorgenti eterogenee che emettono flussi e si vuole parallelizzare l'elaborazione tra più spout. In un contesto del genere, Storm offre una feature interessante: permette l'accesso al contesto della topologia da qualsiasi componente (spout/bolt). Utilizzando questa feature, è possibile suddividere i diversi stream tra più istanze di spout;

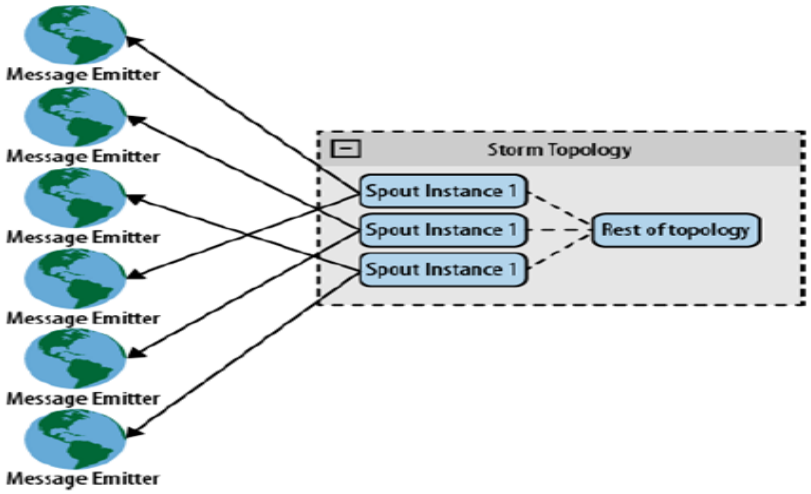


Figura 2.12 – Esempio di Direct Connection Hashing

- Direct Connection Coordinator: nel caso precedente si può collegare lo spout a un dispositivo noto. È possibile utilizzare lo stesso approccio per la connessione a dispositivi sconosciuti utilizzando un sistema di coordinamento per mantenere l'elenco dei dispositivi. Il coordinatore rileva le modifiche apportate alla lista e crea e distrugge le connessioni. Ad esempio, quando si raccolgono i file di log da server web, l'elenco dei server Web può cambiare nel tempo. Quando si aggiunge un server Web, il coordinatore rileva la modifica e crea un nuovo spout per esso;

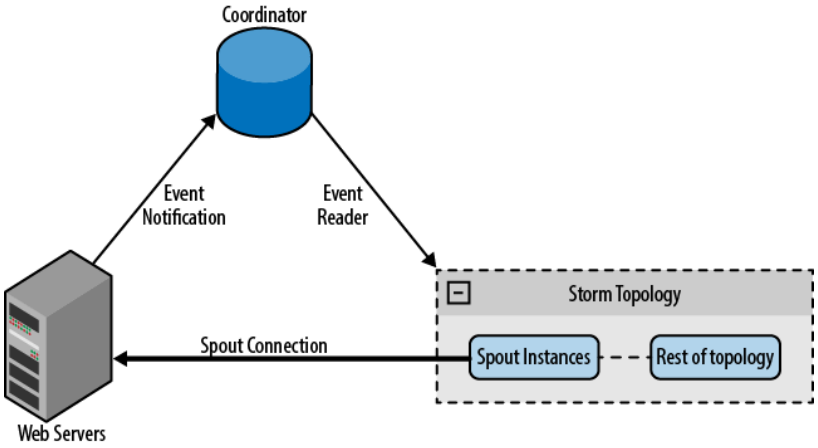


Figura 2.13 – Esempio di Direct Connection Coordinator

- Enqueued Messages: un approccio differente dai precedenti, consiste nel collegare uno spout a un sistema che mantiene una coda dei flussi emessi dalle sorgenti. Tale approccio, svincola la progettazione dalla conoscenza delle sorgenti in quanto il sistema intermedio può fungere da middleware. In molti casi, è possibile utilizzare la

coda per essere affidabili utilizzando la funzionalità replay messages di molti sistemi di che gestiscono code. Questo significa che non c'è bisogno di sapere nulla sugli emettitori di messaggi, e il processo di aggiunta e rimozione degli emettitori sarà più facile che con il collegamento diretto. Allo stesso tempo il sistema intermedio può essere un collo di bottiglia dell'intero sistema.

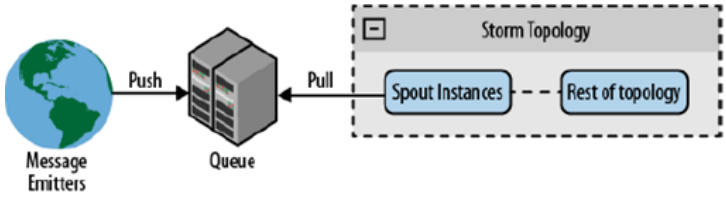


Figura 2.14 – Esempio di Enqueued Messages

2.2.4 Bolt

Un bolt è un componente che prende le tuple in input, le elabora e produce delle tuple in output. I bolt sono creati su una macchina client, serializzati in una topologia e inviati a un nodo master del cluster. Il cluster lancia i nodi worker che deserializzano il bolt e successivamente inizia l'elaborazione delle tuple.

2.2.5 Metric

Storm 0.9 ha aggiunto la nozione di metrics consumer. Spesso è utile sapere che cosa sta succedendo nella topologia. L'interfaccia di Storm (Storm UI) fornisce una certa comprensione, ma questo può essere considerato a livello generale, vale a dire che mostra quante tuple sono state trasmesse tra spout e bolt, e quanti sono andati a buon fine o quanti no. Non può dire nulla circa lo stato interno degli spout e bolt, perché queste informazioni dipendono dall'applicazione specifica. Questo è il punto in cui i metrics vengono utilizzati. Il framework per le metriche permette di creare variabili metriche negli spout e nei bolt. Questi parametri sono trasmessi al metric consumer. Come per i bolt, è responsabilità del programmatore poi mettere in output questi dati per un sistema di storage e/o di visualizzazione esterna.

È utilizzata internamente per tenere traccia dei numeri presenti nella console di Nimbus: conta il numero di execute e ack, latenza media dei processi per bolt, utilizzo dell'heap da parte dei worker, e così via.

Storm mette a disposizione i seguenti metrics:

- AssignableMetric: imposta la metrica per il valore esplicito che gli viene fornito. Utile se si tratta di un valore esterno o nel caso in cui si sta già calcolando il riassunto statistico da soli. È molto utilizzato per statsd Gauges;
- CombinedMetric: interfaccia generica per le metriche che possono essere aggiornate in modo associativo.
- ReducedMetric
    - MeanReducer: traccia una media parziale di valori dati al suo metodo reduce(). Accetta double, integer e long, e mantiene la media interna come un double.
    - MultiReducedMetric: un hashmap di metriche ridotte.
- CountMetric - un totale parziale dei valori forniti. Si chiama incr() per incrementare di uno, incrBy(n) per aggiungere/sottrarre il numero dato.
    - MultiCountMetric - un hashmap di parametri metrici.

I metric consumer sono oggetti destinati a processare/fare dei report/log/etc in uscita dagli oggetti Metric (rappresentate come oggetti DataPoint) per tutti i vari spout/bolt in cui sono stati registrati, fornendo anche i metadati utili come ad esempio dove la metrica è stata raccolta, il worker ospite, il numero della porta del worker, il ComponentID del bolt/spout, il taskID, timestamp, e updateInterval (tutti rappresentati come oggetti TaskInfo). I MetricConsumers sono registrati nella configurazione della topologia di Storm (usingbacktype.storm.Config.registerMetricsConsumer(...)) o nel sistema di configurazione di Storm (topology.metrics.consumer.register). I metric consumer devono implementare l'interfaccia backtype.storm.metric.api.IMetric.

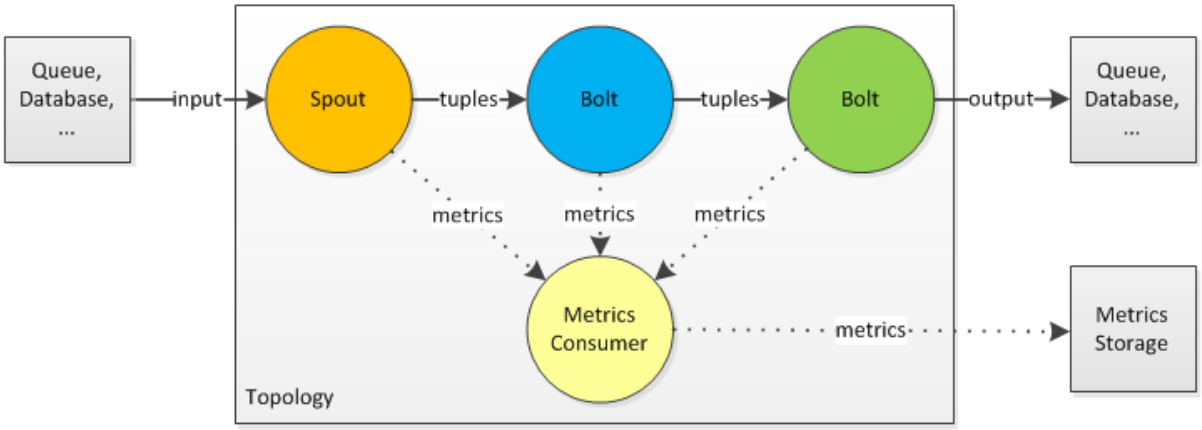


Figura 2.15 – Illustrazione di una topologia classica in Storm

2.2.6 Stream

Lo stream è il core dell'astrazione di Storm. Uno stream è una sequenza illimitata di tuple che vengono processate e create in parallelo in modo distribuito. Per impostazione predefinita, le tuple possono contenere interi, long, short, byte, stringhe, double, float, boolean e array di byte. È anche possibile definire i propri tipi personalizzabili basta che essi siano serializzabili e possono essere utilizzati in modo nativo all'interno tuple. A ogni stream è dato un ID quando viene dichiarato. Poiché i single-stream spout a volte sono comuni, OutputFieldsDeclarer ha metodi di convenienza per dichiarare un unico flusso senza specificare un ID. In questo caso, il flusso viene dato l'ID predefinito di "default".

Le risorse che si riferiscono agli stream sono:

- Tuple: gli stream sono composti da tuple;
- OutputFiledsDeclarer: usa gli stream dichiarati e i loro schemi;
- Serialization: utilizzato per avere informazioni sulla tipizzazione dinamica delle tuple personalizzabili;
- ISerialization: i tipi serializzabili devono implementare questa interfaccia;
- CONFIG.TOPOLOGY_SERIALIZATIONS: i tipi serializzabili possono essere registrati utilizzando questa configurazione.

2.2.6 Organizzazione fisica Storm Cluster

Dal punto di vista fisico, uno Storm Cluster è costituito da nodi organizzati in nodi master e worker. I nodi master eseguono un demone chiamato Nimbus, i nodi worker eseguono un demone chiamato Supervisor. Una topologia viene eseguita attraverso molti nodi worker su differenti macchine. Infine, Storm mantiene lo stato di tutti i cluster in un server ZooKeeper, rappresentante anche l'interfaccia tra Nimbus e i vari Supervisor.

In dettaglio:

- Nimbus: è responsabile di distribuire il codice all'interno del cluster, assegnare i task a ogni nodo worker e di monitorare i guasti. Se Nimbus va in down ed era in esecuzione sotto process supervision (come raccomandato) verrà riavviato come se nulla fosse successo. Mentre Nimbus è down le topologie esistenti continuano a eseguire tranquillamente ma non si riesce a creare nuove topologie. Anche i nodi worker continuano a eseguire senza problemi, il Supervisor potrà riavviare i propri worker

locali se lo ritiene necessario. Tuttavia, i task che falliscono non saranno riassegnati alle altre macchine in quanto questo è compito di Nimbus;

- ZooKeeper: è un servizio centralizzato per il mantenimento d'informazioni di configurazione. È preferibile utilizzare una macchina dedicata per ZK in quanto di norma rappresenta il collo di bottiglia per Storm. Le operazioni di I/O rappresentano invece il collo di bottiglia per ZK;
- Supervisor: esegue una porzione di una topologia. Se un Supervisor va in down ed era in esecuzione sotto process supervision (come raccomandato) verrà riavviato come se nulla fosse successo. I worker in esecuzione non saranno interessanti dal down del Supervision;
- Worker: sono i veri e propri nodi che eseguono la logica. Quando un worker va in down se ha come genitore un Supervisor verrà riavviato. Se l'esecuzione fallisce continuamente sull'avvio ed è attivo l'heartbeat con Nimbus, egli lo riassegnerà a un'altra macchina.

I demoni elencati (tranne i worker che non sono dei demoni) sono stateless e per questo motivo il loro stato viene mantenuto in questo servizio, cosi possibili fail o restart non incidono sul sistema in produzione.

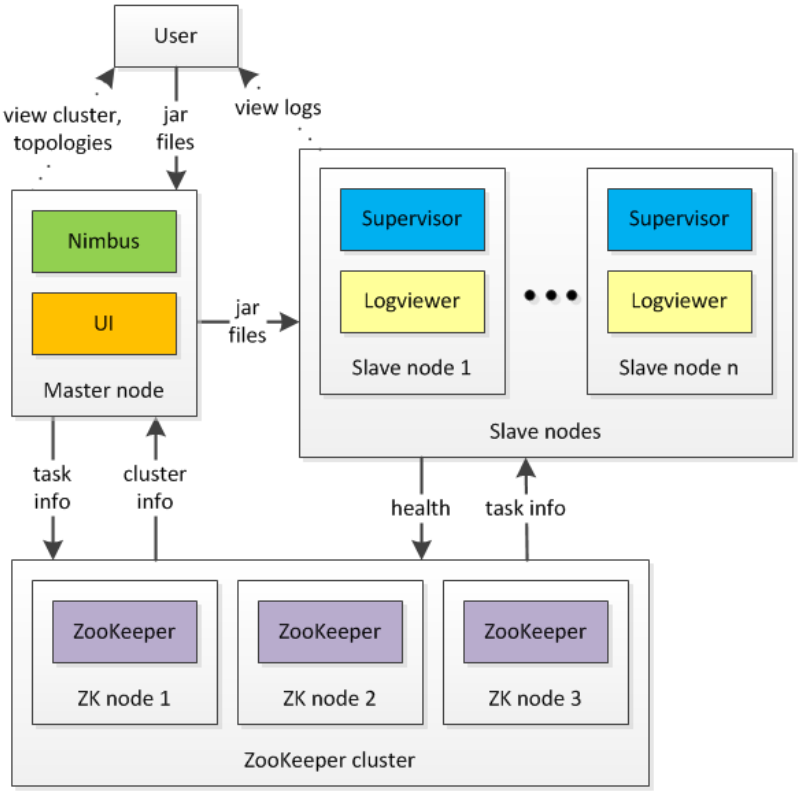


Figura 2.16 – Architettura fisica di Storm

Per quanto riguarda il workflow di Storm funziona nel seguente modo:

- Inizialmente, nimbus attenderà la topologia che sarà sottoposto a esso;
- Una volta che una topologia è presentata, verrà elaborata la topologia e raccolti tutti i task che devono essere eseguite e l'ordine in cui verranno eseguiti;
- In seguito, nimbus distribuirà uniformemente i task a tutti i supervisori disponibili;
- In un particolare intervallo di tempo, tutti i supervisori invieranno i loro "battiti cardiaci" al nimbus per informarlo che sono ancora in esecuzione;
- Quando un supervisore va in down e non invia un "battito cardiaco" al nimbus, il nimbo assegna i task a un altro supervisore;
- Quando il nimbus stesso va in down, i supervisor lavoreranno sui task già assegnati senza alcun problema;
- Una volta che tutti i task sono completati, il supervisor attenderà assegnamento di un nuovo task;
- Nel frattempo, il nimbus che è un down verrà riavviato automaticamente dagli strumenti di monitoraggio del servizio;
- Il nimbus appena riavviato continuerà da dove si era fermato. Allo stesso modo, il supervisor in down può anche essere riavviato automaticamente. Poiché sia il nimbus che il supervisor possono essere riavviati automaticamente ed entrambi continueranno da dove si erano fermati, Storm garantisce l'elaborazione di tutti i task almeno una volta.
- Una volta che tutte le topologie vengono elaborate, nimbus attende l'arrivo di una nuova topologia e allo stesso modo il supervisor attende nuovi task;

Storm opera in due modalità:

- Local mode: le topologie vengono eseguiti su una macchina locale, in una singola JVM. Generalmente, questa modalità è usata per sviluppo, test e debbugging in quanto è il modo più semplice per vedere come le tipologie lavorano insieme.
- Remote mode: in questa modalità, le topologie sono inviate allo Storm cluster che sarà costituito da più processi eseguiti su macchine differenti. In questa modalità non vengono fornite informazioni di debbugging in quanto viene considerata come una modalità di produzione. Tuttavia, è possibile creare uno Storm cluster su una singola

macchina di sviluppo ed effettuare il deploy del progetto che integra Storm, simulando una modalità di produzione.

2.2.7 Parallelismo

Storm fa una distinzione tra le seguenti tre principali entità che vengono utilizzate per far eseguire le topologie in un cluster:

- Processi worker: un processo worker esegue un sottoinsieme di una topologia, all'interno della propria JVM. Esso appartiene a una topologia specifica e può eseguire una o più esecutori per uno o più componenti (spout o bolt). Una topologia in esecuzione consiste in molti di tali processi in esecuzione su molte macchine all'interno di un cluster Storm.
- Esecutori (thread): Un esecutore è un thread che viene generato da un processo worker e viene eseguito all'interno della JVM dello stesso. Un esecutore può eseguire una o più task per lo stesso componente (spout o bolt). Un esecutore ha sempre un thread che utilizza per tutti i suoi task, il che significa che le operazioni vengono eseguite in serie su un esecutore.
- Task: Un task esegue l'elaborazione effettiva dei dati ed è gestito all'interno del thread del suo esecutore. Ogni spout o bolt che s'implementa nel codice esegue il maggior numero di task all'interno del cluster. Il numero di task per un componente è sempre lo stesso per tutta la durata di una topologia, ma il numero di esecutori (thread) per un componente può cambiare nel tempo. Ciò significa che la seguente condizione vale: #threads <= #tasks. Per default, il numero di task è destinato a essere lo stesso del numero di esecutori, cioè Storm eseguirà un task per thread (che di solito è quello che si vuole comunque).

Una macchina all'interno di uno Storm cluster può eseguire uno o più processi worker per una o più topologie. Ogni processo worker fa girare più esecutori per una specifica topologia. Uno o più esecutori girano all'interno di un processo worker e per ogni esecutore viene generato un thread dal processo worker. Ogni esecutore svolge il suo compito per uno o più task dello stesso componente (sia esso uno spout o un bolt). Il task esegue la vera e propria elaborazione dei dati.

In Storm la terminologia "parallelismo" è specificamente usato per descrivere il cosiddetto parallelism hint cioè il numero iniziale di esecutori (thread) di un componente. C'è più di un

modo per impostare le opzioni di parallelismo però, e Storm ha attualmente il seguente ordine di precedenza per le impostazioni di configurazione: configurazione component-specific esterno -> configurazione component-specific interno -> configurazione topology-specific -> storm.yaml -> defaults.yaml.

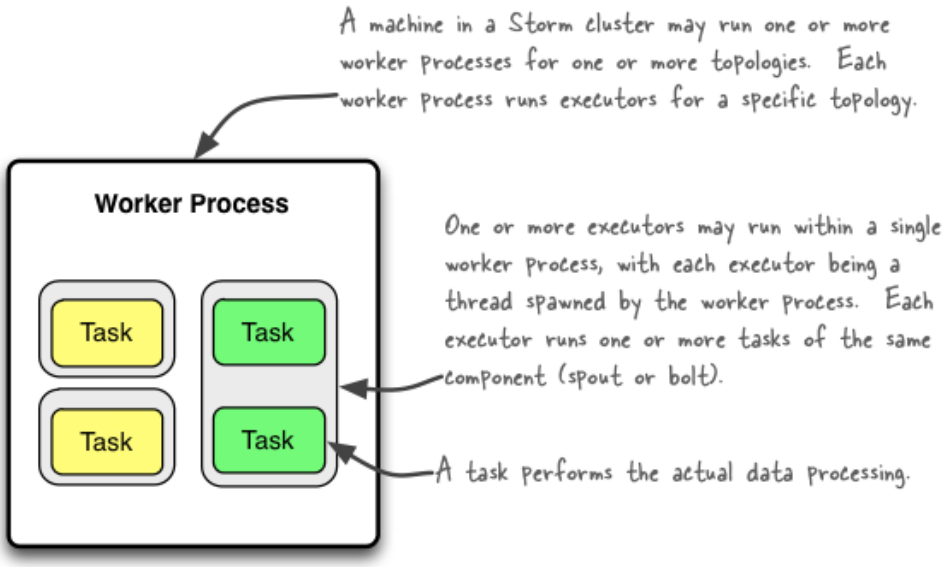


Figura 2.17 – Illustrazione della relazione fra le tre entità

Storm 0.8.2 ha introdotto l'Isolation Scheduler che rende facile e sicuro condividere un cluster tra molte topologie, cioè risolve il problema multi-tenancy - evitando risorse contese tra topologie - fornendo un completo isolamento tra le topologie. Quando si utilizza l'isolation scheduler si consiglia di impostare un numero di workers multiplo del numero di macchine. E il parallelism hint a un multiplo del numero di worker. Se lo si fa chiamando setNumTasks () (che la maggior parte dei clienti utilizza), poi quello che succede è che il carico di lavoro è uniformemente distribuito. Ogni macchina e ogni JVM avrà lo stesso numero di thread, e circa lo stesso carico di lavoro.

Nella seguente tabella si spiega meglio come settare le varie componenti.

Componente	Descrizione	Settaggio
Worker	Numero di processi worker per una data topologia attraverso il cluster Storm	Config#setNumWorkers
Executor	Numero di esecutori per	TopologyBuilder#setSpout

	componenti	TopologyBuilder#setBolt
Task	Numero di task per componente	ComponentConfiguration Decla- rer#setNumTasks

Tabella 2.2 – Componenti base per il parallelismo

Una bella funzionalità di Storm è quella che rende possibile l'aumentare o il diminuire il numero di worker e/o esecutori senza dover riavviare il cluster o la topologia. Si hanno due opzioni per riequilibrare una topologia:

- Utilizzare lo Storm web UI;
- Utilizzare il tool CLI di Storm.

Gli utenti di Storm che richiedono di creare una topologia che guidi il sistema a eseguire un insieme di task oltre a creare le fonti di dati e i processi che elaborano in maniera effettiva i dati, deve occuparsi anche del grado di parallelismo che è necessario per completare i carichi di lavoro in tempo (sennò non sarebbe un sistema che funziona in real-time) e ciò deve essere purtroppo determinato manualmente. Si tratta di un problema ben noto il far scalare grandi sistemi di lavoro a una soglia accettabile, è un processo a tentativi, porta errori ed è time-consuming. In Storm, questo lavoro deve essere fatto per ogni topologia che viene creata.

Quello che si può fare in questo momento è:

- Decidere il parallelism hint degli spout e bolt in base al valore di soglia che vogliamo raggiungere;
- Usare il tool CLI per cambiare tale parallelismo se vediamo che non si raggiungono quelli che sono gli obiettivi prefissati;
- Utilizzare l'interfaccia di Storm per controllare anche tramite le metriche se sono soddisfatti i valori di soglia che ci si è preposti.

Tutti gli altri metodi che ci sono per far scalare in maniera automatica Storm sono dei metodi costruiti ad hoc, anche se lo stesso progetto Storm sta lavorando a una versione dinamica di tale framework ma per il momento non c'è nessuna release.

Quasi tutti i sistemi ad hoc però seguono le seguenti linee guida:

- Eliminare il bisogno di fare scalare il sistema manualmente;

- Permettere di far scalare le topologie in base a delle regole: cioè cambiare configurazione in maniera automatica nel momento in cui il sistema rileva che si è superato un certo valore di soglia;
- La configurazione voluta deve essere raggiunta in pochi minuti da quando si riceve la prima tupla: per far scalare manualmente un sistema bisogna aspettare ore per raggiungere i valori di soglia raggiunti in una topologia grande. Ogni volta che il carico in ingresso aumenta il sistema deve essere in grado di cambiare automaticamente, e deve essere vero anche il viceversa, cioè nel momento in cui il carico diminuisce, devono diminuire anche le risorse utilizzate.

2.2.8 Storm on Cloud

In generale una terza generazione di sistemi per il processamento real time sta venendo fuori: con una potenza di calcolo senza precedenti grazie alla scalabilità orizzontale, adattamento del carico dinamico on-demand, modelli di programmazione semplificati attraverso potenti semantiche per la gestione degli eventi, degrado delle prestazioni lineare in caso di guasti. Alcune questioni rimangono da risolvere per rendere i sistemi per il real time processing pronto per la cloud-era. La maggior parte dei sistemi esistenti sono infrastrutture che non tengono conto di strategie di distribuzione e delle caratteristiche peculiari dell'hardware a disposizione. La connessione fisica e le relazioni tra gli elementi infrastrutturali è noto per essere un fattore chiave sia per migliorare le prestazioni del sistema sia per la tolleranza ai guasti. Gli utenti al giorno d'oggi non sono interessanti solo alle performance ma anche al costo monetario. Questi sistemi se sono cloud-based dovrebbero prendere in considerazione i costi di gestione come variabile per l'ottimizzazione delle prestazioni. Bellavista et al. hanno proposto un primo prototipo, che permette all'utente di trovare un compromesso tra il costo monetario e la tolleranza ai guasti.

La piattaforma cloud di Google fornisce l'infrastruttura per gestire stream di dati alimentati da milioni di dispositivi intelligenti. L'architettura di questo tipo di stream processing in tempo reale deve fare i conti con l'input, il processing, lo storage e l'analisi di centinaia di milioni di eventi all'ora. L'architettura sottostante descrive un tale sistema:

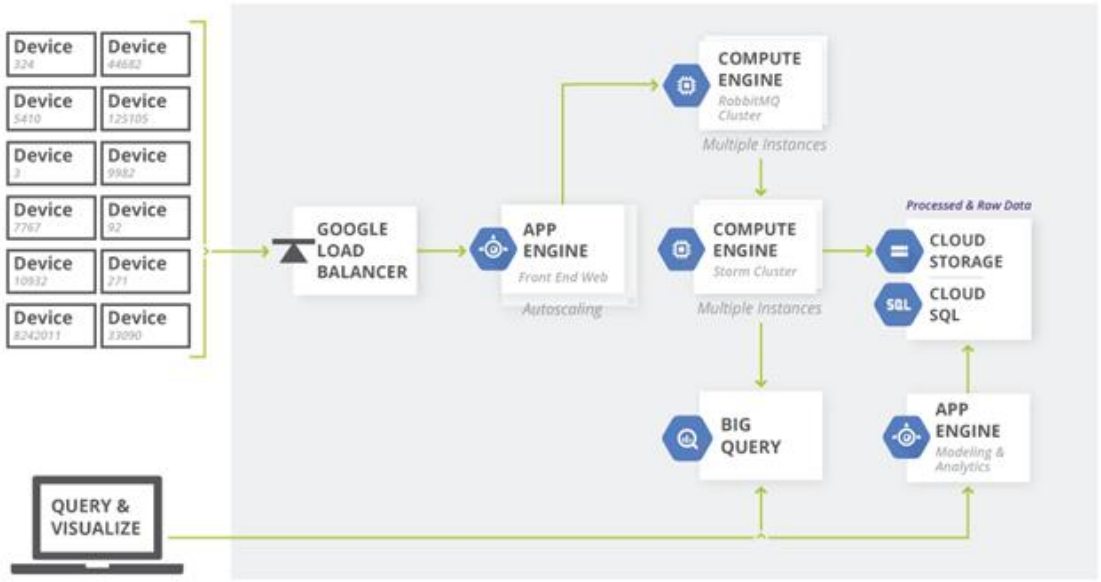


Figura 2.18 – Storm sulla piattaforma cloud di Google

Milioni di dispositivi inviano le loro informazioni di stato per il front-end App Engine. Load Balancer Google assicura che il carico sia distribuito attraverso le istanze di App Engine, la piattaforma PaaS di Google. La funzione Autoscaling viene utilizzata per aumentare o diminuire il numero di istanze App Engine in funzione dei carichi derivanti dai dispositivi in maniera automaticamente. I server Front End inviano i messaggi a un cluster che esegue un sistema di accodamento come RabbitMQ per gestire la pubblicazione dei dati. RabbitMQ, in esecuzione su Compute Engine, fornisce un sistema di pub/sub che garantisce affidabilità ed elasticità. Da lì i messaggi vengono letti da un cluster di nodi che eseguono software stream processing come Apache Storm. Si possono utilizzare una varietà di opzioni esistenti tra cui BigQuery, Google Cloud SQL e Google Cloud Storage. Google BigQuery supporta l'inserimento ad alta velocità di righe in tabelle che possono essere interrogati con un linguaggio SQL.

Azure cloud rende Apache Storm facile e conveniente da implementare, con nessun hardware da acquistare, nessun software da configurare, la scelta di strumenti di sviluppo (Java o C #), e con una profonda integrazione con Visual Studio.

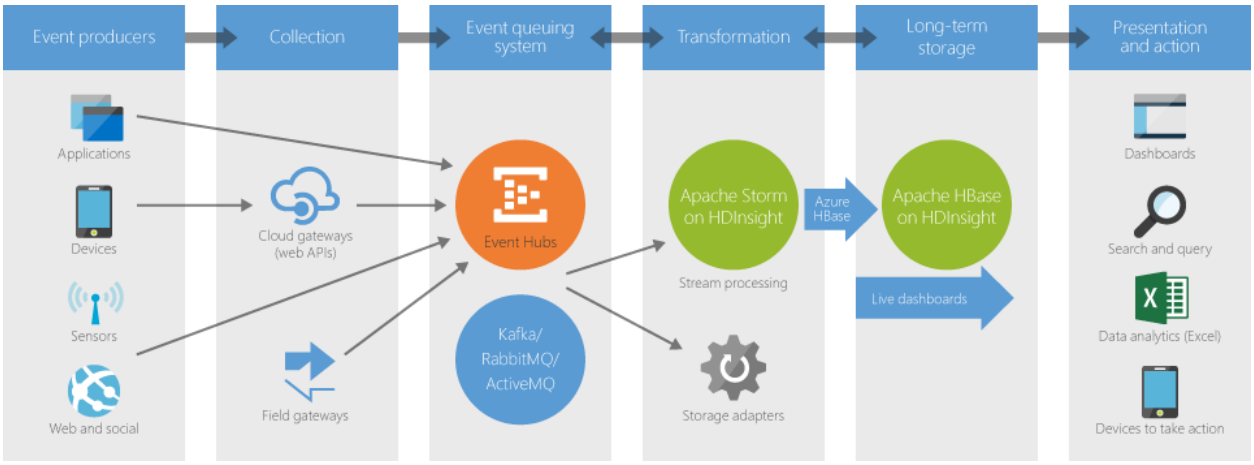


Figura 2.19 – Storm sulla piattaforma Azure

Con Storm su HDInsight, non c'è nessuna installazione o impostazione da fare, tutto è fatto da Azure. Storm verrà installato in pochi minuti senza dover acquistare nuovo hardware o incorrere in altri costi up-front. Il built-in di integrazione con l'IDE di Visual Studio significa che è possibile sviluppare, implementare ed eseguire in debug le topologie Storm rapidamente e facilmente. Si può anche mescolare spout scritti in altre linguaggi, il che significa che è possibile sfruttare il vasto universo di bolt e spout esistenti come parte della topologia. Storm per HDInsight sfrutta la potenza del cloud Azure, rendendo più facile la creazione di gruppi di qualsiasi dimensione e di elaborare qualsiasi quantità di dati su richiesta. Il costo viene calcolato solo per l'elaborazione e lo storage effettivamente utilizzati. Storm per HDInsight porta un ulteriore passo in avanti per garantire al 99,9% il tempo di esecuzione. Azure inoltre offre anche il supporto enterprise 24x7 e cluster monitoring.

Per quanto riguarda Amazon Elastic Compute Cloud (EC2), lo stesso Storm mette a disposizione istruzione per tenere attivo e funzionante un cluster. All'interno di storm-deploy, una volta scaricato provvederà in maniera automatica il provisioning, la configurazione e l'istallazione di un cluster Storm su EC2. Inoltre configura Ganglia in modo tale da poter monitorare CPU, disco e rete.

Invece per OpenStack esiste un progetto che si chiama Sahara che ha come obiettivo quello di fornire agli utenti un modo semplice di integrazione per i framework di trattamento di grandi quantità di dati (come Hadoop, Spark precedentemente e dal 2014 anche per Storm). Questo si ottiene specificando parametri di configurazione, come la versione del framework, la topologia del cluster, i dettagli hardware del nodo e altro ancora. Dopo l'installazione è necessario attivare il plugin di Storm aggiungendolo alla lista dei plugin nel file di

configurazione. Infine è necessario garantire che le immagini che si stanno installando con Sahara abbiano installato Storm.

2.2.9 Buffer dei messaggi interni

Storm fa uso di ZeroMQ (0MQ), una libreria di messaging asincrono a elevate prestazioni, utilizzata in sistemi distribuiti, che offre le seguenti caratteristiche:

- Una socket library che agisce come un framework concorrente;
- Consegna di messaggi attraverso IPC, TCP e multicast;
- Operazioni di I/O asincrone per il passaggio di messaggi tra le applicazioni scalabili e multicore;
- Connessioni N-to-N, publish/subscribe, pipeline, request-reply.

In particolare, Storm utilizza delle socket push/pull.

In generale invece per la comunicazione intra-worker in Storm (inter-thread sullo stesso nodo Storm) viene utilizzato LMAX Disruptor. Per la comunicazione inter-worker (nodo a nodo attraverso la rete) si utilizza invece ZeroMQ o Netty. Per la comunicazione inter-topologia non c'è niente di integrato in Storm di default, si usano delle soluzioni ad hoc utilizzando un sistema di messaggistica come Kafka/RabbitMQ, un database, ecc.

Nella figura sottostante si mostra una panoramica della coda interna del messaggio di un worker in Storm. Le code relativi a un processo worker sono colorate in rosso, le code relative invece agli executor sono colorate in verde.

Per quanto riguarda i processi worker per gestire i messaggi in entrata e uscita ogni processo worker ha un singolo thread in ascolto sulla porta TCP del work. Il parametro topology.receiver.buffer.size determina la dimensione del buffer che il thread di ricezione utilizza per inserire messaggi in arrivo nelle code in entrata dei thread dei worker. Allo stesso modo, ogni worker ha un singolo thread di invio che è responsabile per la lettura dei messaggi nella coda di trasferimento del worker e di inviarli in rete per i consumatori a valle. La dimensione della coda di trasferimento viene configurata attraverso topology.transfer.buffer.size.

- Il topology.receiver.buffer.size è il numero massimo di messaggi che vengono ammassati insieme in una sola volta per l'aggiunta di coda in entrata al thread del worker. L'impostazione di questo parametro troppo elevato può causare problemi, il

valore predefinito è otto elementi, e il valore deve essere una potenza di 2 (questo requisito viene indirettamente da LMAX Disruptor).

- Ogni elemento della coda di trasferimento configurato con topology.transfer.buffer.size è in realtà una lista di tuple. Il valore di default è di 1024 elementi.

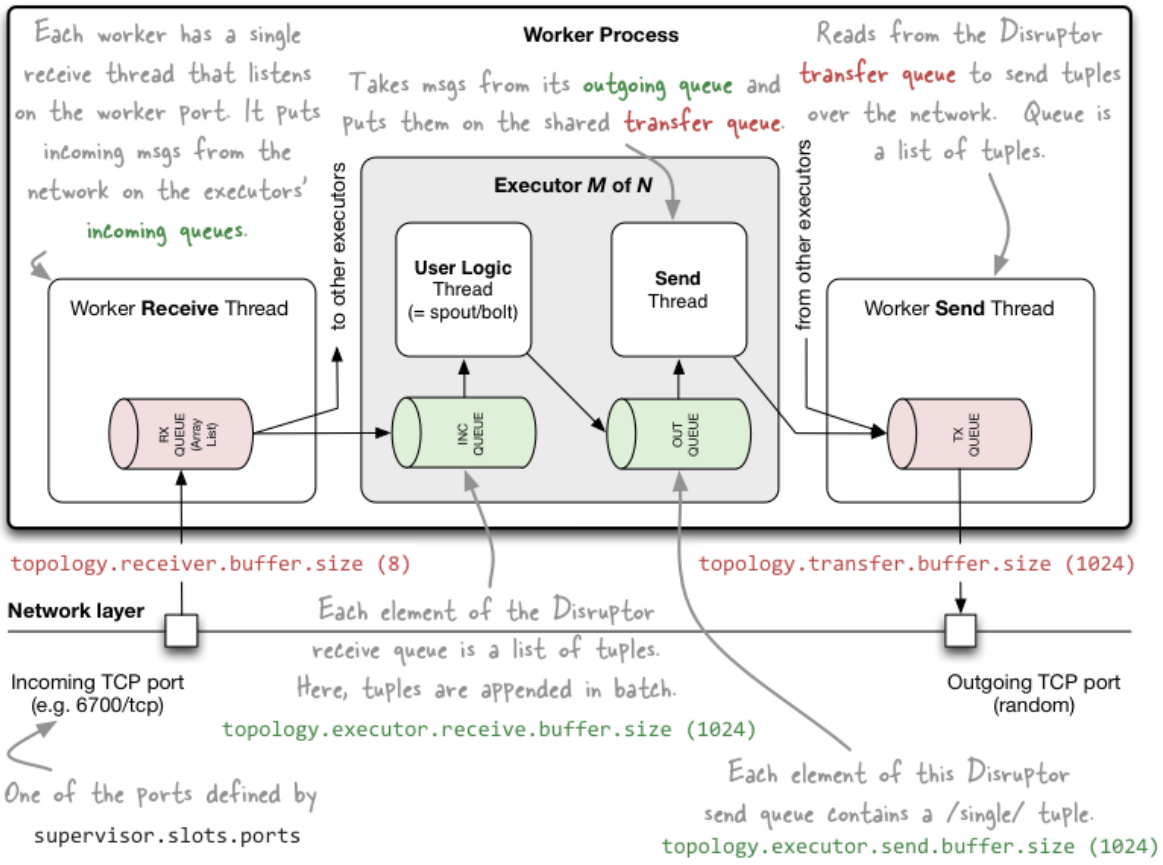


Figura 2.20 - Gestione dei buffer

Per quanto riguarda gli executor invece ogni thread esecutore ha la sua coda in entrata e in uscita. Come descritto in precedenza, per un processo worker viene eseguito un thread dedicato alla ricezione che è responsabile per lo spostamento di messaggi in arrivo alla coda in ingresso dell'appropriato esecutore. Allo stesso modo, ogni esecutore ha il suo thread di trasmissione dedicato che sposta i messaggi in uscita dalla sua coda in uscita alla coda del processo worker 'genitore'. Le dimensioni delle code in entrata e uscita degli esecutori sono configurate tramite topology.executor.receive.buffer.size e topology.executor.send.buffer.size rispettivamente.

- Ogni esecutore ha un unico thread che muove i messaggi dalla coda in uscita dell'esecutore alla coda del processo worker. La topology.executor.receive.buffer.size è la dimensione della coda in ingresso per un esecutore. Ogni elemento di questa coda è una lista di tuple. Il valore di default è 1024 elementi, e il valore deve essere una potenza di 2 (questo requisito viene da LMAX Disruptor).
- Il topology.executor.send.buffer.size è la dimensione della coda in uscita per un esecutore. Ogni elemento di questa coda conterrà una singola tupla. Il valore di default è 1024 elementi, e il valore deve essere una potenza di 2.

2.2.10 Distributed Messaging System

Apache Storm elabora i dati in tempo reale e l'input normalmente viene da un message queuing system. Un sistema di messaggistica distribuito esterno può fornire i dati necessari per l'elaborazione in tempo reale. Gli spout leggeranno i dati dal sistema di messaggistica e le convertono in tuple e ingresso per i bolt. Come detto precedentemente invece, Storm utilizza internamente un proprio sistema di messaging per la comunicazione fra le varie entità.

La messaggistica distribuita si basa sul concetto di accodamento dei messaggi in maniera affidabile. I messaggi vengono accodati asincronamente tra applicazioni client e sistemi di messaggistica. Un sistema di messaggistica distribuito offre i vantaggi di affidabilità, scalabilità, e persistenza. La maggior parte dei modelli di messaggistica seguono il modello publish-subscribe (semplicemente Pub-Sub), dove i mittenti dei messaggi sono chiamati publisher e coloro che vogliono ricevere i messaggi sono chiamati subscriber. Una volta che il messaggio è stato pubblicato dal mittente, i subscriber possono ricevere il messaggio selezionato con l'aiuto di un'opzione di filtraggio. Di solito si hanno due tipi di filtro, uno è il filtraggio basato su argomenti e un altro è filtro basato sul contenuto. La seguente tabella descrive alcuni dei sistemi di messaggistica con elevato throughput e tra i più popolari e usati in Storm.

Sistema distribuito di messaging	Descrizione
Apache Kafka	Kafka è stato sviluppato da LinkedIn e successivamente è diventato un sotto-progetto di Apache. È persistente e con modello pub/sub distribuito. Inoltre è veloce, scalabile

	e altamente efficiente
RabbitMQ	RabbitMQ è un applicativo di messaggistica distribuita, robusta e open source. È facile da usare ed eseguire su tutte le piattaforme
JMS	JMS (Java Message Service) è un insieme di API open-source che supportano la creazione, lettura e invio di messaggi da un'applicazione a un'altra.
ActiveMQ	ActiveMQ è un sistema di messaggistica open-source di JMS.
ZeroMQ	ZeroMQ è un sistema di messaggistica peer-to-peer. Fornisce patern come push-pull, router-dealer (già descritta in precedenza)
Kestrel	Kestrel è un sistema di accodamento di messaggi distribuito facile, veloce e sicuro.

Tabella 2.3 – Sistemi distribuiti di messaggistica per Storm

2.2.11 Trident

Trident è un'astrazione di alto livello per fare calcolo in tempo reale sopra Storm. Esso consente di mischiare perfettamente throughput elevato (milioni di messaggi al secondo), lo stream processing stateful con l'esecuzione di query distribuite a bassa latenza. È familiare a strumenti di elaborazione in batch di alto livello come il Pig o Cascading. Trident ha join, aggregazioni, raggruppamenti, funzioni e filtri. Oltre a questi, Trident aggiunge primitive per fare elaborazione incrementale stateful in cima a qualsiasi database o archivio di persistenza. Trident ha consistenza, semantica exactly-once, si ha quindi facilità nel ragionamento sulle topologie Trident. Il principale vantaggio offerto da Trident rimane comunque la garanzia che ogni messaggio introdotto nella topologia viene elaborato una sola volta. Questo risultato è difficile da ottenere in una topologia Java di tipo raw, che garantisce che i messaggi verranno elaborati almeno una volta. Esistono altre differenze, ad esempio la disponibilità di

componenti predefiniti che possono essere usati senza che sia necessario creare bolt. I bolt sono infatti sostituiti da componenti meno generici, ad esempio filtri, proiezioni e funzioni.

Le API di Trident espongono un'opzione facile per creare una topologia Trident con la classe "TridentTopology". In sostanza, la topologia Trident riceve il flusso di input da uno spout e ne fa una sequenza di operazioni ordinate (filtro, aggregazione, raggruppamento, ecc) sul flusso. Le tuple di Storm sono sostituite dalle tuple di Trident e i bolt sono sostituiti dalle operazioni. Una tupla Trident è una lista di valori con nome. L'interfaccia TridentTuple è il modello di dati di una topologia Trident ed è l'unità di base di dati che possono essere elaborati da una topologia Trident. Uno spout Trident è simile a uno spout in Storm, con opzioni aggiuntive per utilizzare le caratteristiche nuove. In realtà, si può utilizzare ancora l'interfaccia IRichSpout, che si è sempre utilizzato in una topologia Storm, ma con ciò perdiamo le proprietà transazionali e non si è in grado di utilizzare i vantaggi forniti da Trident. Lo spout di base con tutte le funzionalità di Trident è "ITridentSpout". Supporta sia la semantica transazionale sia la transazionale opaca. Gli altri spout sono IBatchSpout, IPartitionedTridentSpout, e IOpaquePartitionedTridentSpout. Oltre a questi spout generici, Trident ha molti altri esempi di implementazione di spout. Uno di loro è FeederBatchSpout, che si può utilizzare per inviare lista di tuple facilmente senza che ci si preoccupi di elaborazione in batch, parallelismo, etc. Tridente si basa sulle 'operazioni trident' per elaborare il flusso di input di tuple. Le API hanno un certo numero di operazioni in-built per gestire l'elaborazione del flusso. Queste operazioni vanno dalla semplice validazione di raggruppamenti complessi ad aggregazioni di tuple.

Le più importanti operazione che si possono fare sono:

- Filtraggio: Filter è un oggetto utilizzato per eseguire l'operazione di convalida dell'input. Un filtro Trident ottiene un sottoinsieme di tuple come input e restituisce true o false a seconda che determinate condizioni siano soddisfatte o meno. Se viene restituito vero, allora la tupla viene mantenuta nel flusso di uscita; altrimenti, la tupla viene rimosso dal flusso. Filter eredita dalla classe BaseFilter e implementare il metodo isKeep;
- Funzioni: la Function è un oggetto utilizzato per eseguire una semplice operazione su una singola tupla. Prende in ingresso un sottoinsieme di tuple e restituisce zero o più tuple (anche nuove). Function eredita dalla classe BaseFunction e implementa il metodo execute;

- Aggregazione: Aggregation è un oggetto utilizzato per eseguire operazioni di aggregazione su un lotto di ingresso o una partizione o flusso. Trident ha tre tipi di aggregazione:
    - aggregate: durante il processo di aggregazione, le tuple sono inizialmente ripartizionate usando un raggruppamento globale per combinare tutte le partizioni dello stesso lotto in una singola partizione;
    - partitionAggregate: aggrega ogni partizione anziché l'intero lotto di tuple. L'uscita del partition aggregate sostituisce completamente le tuple in ingresso e contiene un singolo campo tupla;
    - persistentAggregate: aggrega tutte le tuple attraverso tutti i lotti e salva il risultato in memoria o in un database.
- Raggrupamenti: un'operazione di raggruppamento è un'operazione in-built e può essere chiamato dal metodo groupBy, il quale ripartiziona il flusso facendo una partizione sui campi indicati, e poi all'interno di ogni partizione, raggruppa le tuple insieme in base al fatto se hanno campi sono uguali. Normalmente, si utilizza "groupBy" insieme a "persistentAggregate" per ottenere aggregazioni raggruppate;
- Merging e Joining: il merging e lo joining possono essere fatti utilizzando rispettivamente i metodi "merge" e "join". Il merge unisce uno o più flussi. Lo join è simile al merge, tranne per il fatto che lo join utilizza i campi delle tuple da entrambi i lati per controllare e unire i due flussi.

Trident fornisce un meccanismo per la manutenzione dello stato. Le informazioni di stato possono essere memorizzate nella topologia stessa, altrimenti è possibile memorizzare in un database separato. Il motivo per cui si fa ciò è per recuperare una tupla se essa fallisce durante l’elaborazione. Questo crea un problema durante l'aggiornamento dello stato, perché non si è certi che lo stato di queste tuple è stato aggiornato in precedenza o meno. Se la tupla fallisce prima di aggiornare lo stato, facendo ritentare la tupla potrebbe far diventare lo stato stabile. Tuttavia, se la tupla fallisce dopo l'aggiornamento dello stato rientrando la stessa tupla aumenterà di nuovo il conteggio nel database e rendere lo stato instabile.

Come in molti casi d'uso, se il requisito è quello di elaborare una query solo una volta, si è in grado di raggiungere quest'obiettivo scrivendo una topologia a Trident. D'altra parte, sarà difficile ottenere esattamente una volta elaborazione nel caso di tempesta. Quindi Trident sarà utile per i casi d'uso in cui si richiede una sola volta l'elaborazione. Trident invece lavora male

nei casi d'uso in cui si vuole ottenere performance alte in quanto aggiunge complessità a Storm oltre alla gestione dello stato.

Capitolo 3 – Eddystone, XMPP e Apache Cassandra

3.1 Google Eddystone

3.1.1 Introduzione

Eddystone è una tecnologia a beacon rilasciato da Google del mese di luglio 2015. È open-source, cross-platform e offre la posizione e la vicinanza di un utente via formato beacon utilizzando Bluetooth a basso consumo energetico (BLE). La tecnologia permette a smartphone, tablet e altri dispositivi di eseguire azioni quando sono in prossimità di un dispositivo che supporta Eddystone. I bluetooth beacon fanno parte del trend di Internet of Things (IoT). I beacon sono piccoli trasmettitori (di solito a batteria) che inviano informazioni su un punto specifico d'interesse, e queste informazioni vengono poi passivamente prelevati da uno smartphone o tablet a portata del trasmettitore. Una fermata dell'autobus attrezzato con beacon potrebbe inviare i tempi di transito, negozi potrebbero inviare promozioni ai clienti in quel periodo in negozio, o un museo potrebbe inviare alle persone informazioni sulla mostra che hanno davanti a loro. In tandem con l'Eddystone, Google ha rilasciato le API di prossimità Beacon (Proximity Beacon API) progettate per aiutare gli utenti a coordinare un insieme di beacon nella gestione e organizzazione, incluso verificare il loro stato di salute.

Google Eddystone si contrappone ad uno standard già presente dal 2013 che si chiama iBeacon messo appunto dalla Apple, che presenta però un numero significante di problemi, cui il primo che è uno standard proprietario che funziona solo con gli iDevices, questo

significa che chi lo adotta taglia fuori dal proprio range di utilizzatori circa il cinquanta percento degli smartphone in USA ed oltre l'ottanta percento livello mondiale.

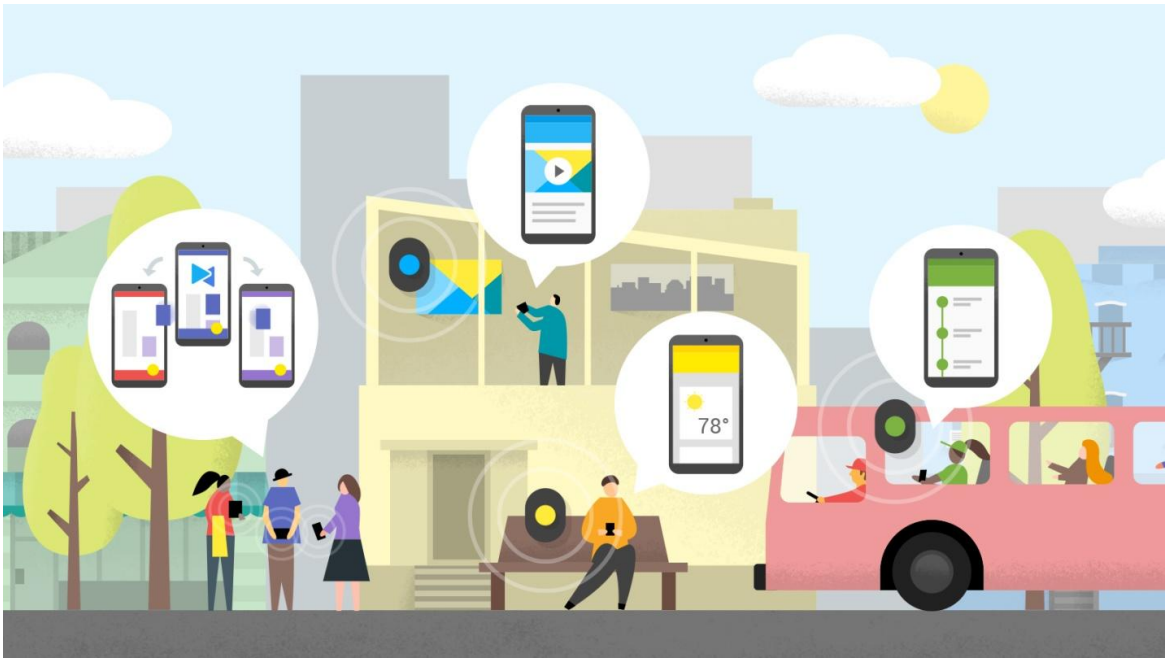


Figura 3.1 – Ambiente con Beacon

Per garantire aggiornamenti tempestivi basate sul contesto attraverso i beacon, le API di prossimità dei beacon consentono di associare dati allegati con i beacon, anche dopo che sono stati distribuiti. Utilizzando l'API nelle vicinanze, un'applicazione Android o iOS può quindi rilevare beacon vicini e utilizzare gli allegati per offrire agli utenti un'esperienza di proximity-aware migliore. I dispositivi degli utenti possono anche utilizzare i beacon come un segnale per migliorare altri strumenti basati sulla localizzazione, come la Place Picker, che assiste gli utenti nella scelta dei business o punti di interesse nelle vicinanze.

Le PBA (Proximity Beacon API) in combinazione con la telemetria broadcast di Eddystone (Eddystone-TLM), aiuta a garantire che l'insieme beacon si comporti come dovrebbe. È possibile utilizzare l'endpoint di diagnostica per individuare eventuali comportamenti errati, come ad esempio un beacon con batteria scarica, assicurando che gli utenti abbiano una grande esperienza in modo continuo. Queste statistiche di salute ed altro sono disponibili attraverso strumenti di monitoraggio del Proximity Beacon API.

I beacon bluetooth rappresentano delle comunicazioni unidirezionali, quindi di solito l'obiettivo è l'invio di una notifica che, quando premuta, faccia partire un form in grado di visualizzare o di trasferire dati.

L’UUID (Universal Unique Identifier) è un valore di 128 bit che identifica ogni beacon nel mondo, che un’app può ascoltare ed eseguire per determinate operazione, ed è esattamente ciò che Apple iBeacon mandano, e possono mandare solo questo tipo di informazione. L’aspetto negativo dell’UUID è che esso è legato allo sviluppatore o all’azienda proprietaria del trasmettitore dei beacon con vincolato l’unica app che può utilizzare quell’informazione, e ciò è limitante pensando alle potenzialità di questa tecnologia. Eddystone utilizza molte tipologie di frame tra cui gli URL, infatti, tramite l’invio di un URL invece di un UUID può essere considerato molto più universale, si apre solo in un browser Web. Infatti non tutti gli utenti vogliono installare e tenere installate tutte le app in posti in cui ci potrebbero essere dei beacon. Un esempio può essere considerato una persona che sta davanti alle macchinette e non vuole ovviamente scaricare un’app se vuole comprare solo una bibita, ma può essere interessato ad altre tipi di informazioni che avrebbe tramite Browser Web. Quindi per transazioni one-time è preferibile usare l’URL. Gli URL sono fondamentalmente la versione "QR Code" di un beacon. I vantaggi rispetto al codice QR sono che non hai bisogno di un lettore QR Code e non si deve scattare una foto di un codice QR. Un EID (Ephemeral Identifier) è un frame sicuro, è un beacon personale che solo gli utenti autorizzati possono leggere. Google non offre molti dettagli su questo nuovo tipo di frame se non informazioni del tipo "cambia frequentemente e permettere solo ai clienti autorizzati di decodificarlo." L'azienda dice che questo si potrebbe usare per cose come trovare il vostro bagaglio quando si scendere l'aereo o la ricerca di chiavi perse. I dati di telemetria - quest'ultimo tipo di frame (come spiegato in precedenza) è per le aziende che devono gestire grandi insiemi di beacon. I beacon funzionano spesso con alimentazione a batteria, il che significa che le batterie devono essere cambiate, in questo modo il frame di tipo telemetrico andrebbe ad inviare i dati diagnostici e la batteria residua al centro di gestione.

Mentre Eddystone, il formato dei beacon è open source, il metodo raccomandato di Google per ricevere i frame non lo è. Google Play Services, le Nearby API, e il servizio cloud di PBA (Proximity Beacon API) sono tutti sistemi proprietari. Pure in questo caso Google sta ancora seguendo il modello Android/Google Play Service, dove la piattaforma è open source, ma per la migliore esperienza si ha anche bisogno di alcuni pezzi di proprietà di Google. Play Service funge da singola applicazione che monitora la connessione Bluetooth per beacon e gli avvisi delle applicazioni appropriate, al contrario dell’approccio in cui ogni singola applicazione individualmente apre la connessione Bluetooth per la ricerca di beacon.

Utilizzando la piattaforma per i beacon, tra cui PBA e Eddystone è possibile:

- Abilitare le app a reagire agli allegati beacon in dipendenza del contesto;
- Monitorare lo stato dell'insieme dei frame telematici e degli endpoint di diagnostica;
- Sfruttare il Physical Web.

Le PBA (Proximity Beacon API) possono essere utilizzate per registrare qualsiasi beacon che supporti una delle seguenti specifiche:

- Eddystone;
- iBeacon;
- AltBeacon.

La piattaforma beacon di Google consiste nei seguenti componenti:

- Dispositivo che supportano i beacon;
- Formato Eddystone;
- Proximity Beacon API.

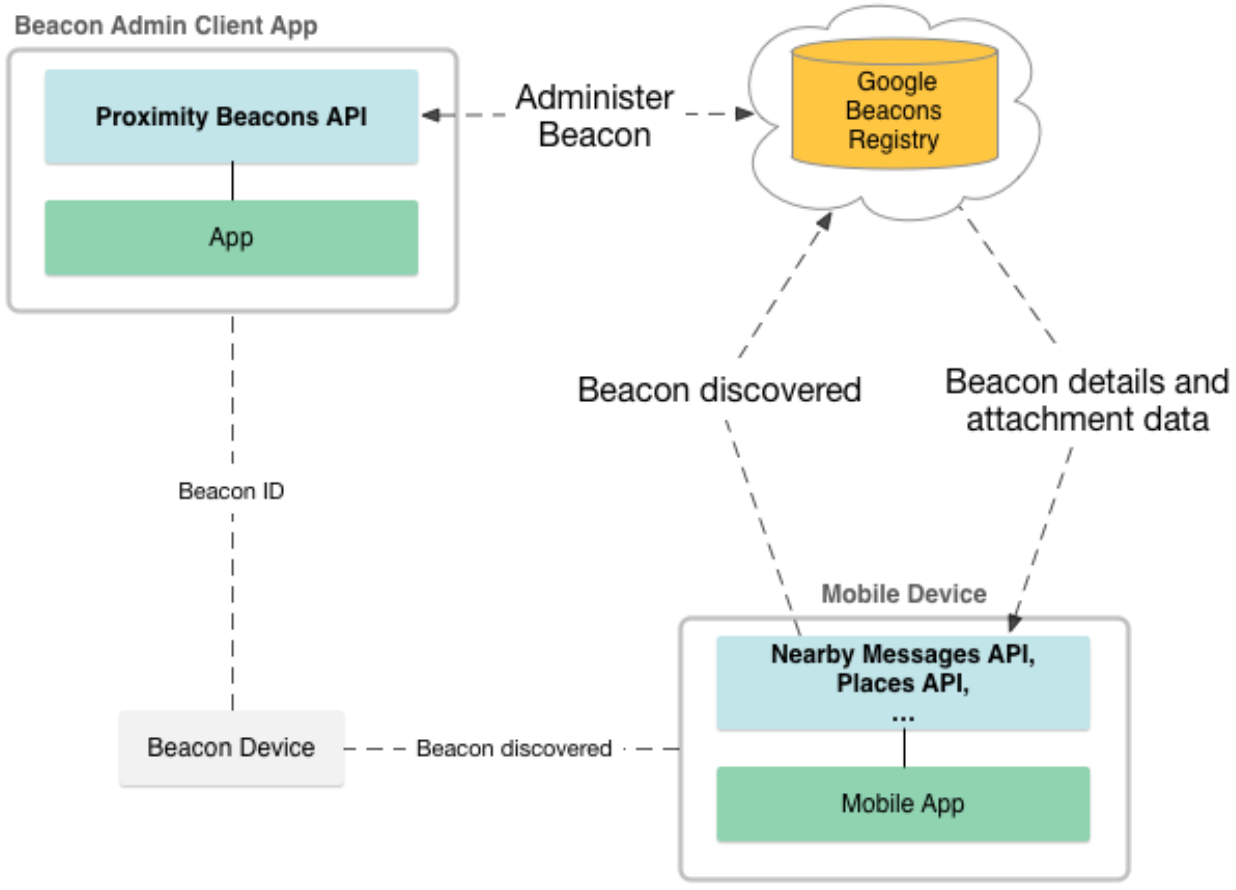


Figura 3.2 – Utilizzo delle PBA

3.1.2 Dispositivi Beacon

I dispositivi beacon sono dispositivi leggeri che trasmettono i dati in forma di frame beacon Bluetooth, come Eddystone, a intervalli predefiniti. I dispositivi beacon possono essere alimentati da una batteria che può durare per oltre un anno. Inoltre si può anche utilizzare una fonte di alimentazione esterna, per esempio se il beacon è incorporato in un dispositivo più grande come un televisore o di un veicolo. La durata della batteria del dispositivo beacon è influenzata dalla velocità con cui trasmette il suo messaggio, e il livello di potenza delle trasmissioni. Questi parametri influenzano anche la latenza di rilevamento. Per lavorare con le API di Google, i dispositivi beacon devono prima essere forniti con alcune impostazioni iniziali (namespace ID, instance ID, frame format, e così via), e poi registrati utilizzando le PBA.

3.1.3 Formato Eddystone

Come anche detto precedentemente Eddystone può essere rilevato da dispositivi come Android e iOS. Il formato Eddystone si basa sulla collaborazione con i partner del settore e nell'analisi implementazioni esistenti. Come descritto ampiamente in precedenza diversi tipi di payload possono essere inclusi nel formato di frame, tra cui:

- Eddystone-UID: un ID univoco opaco;
- Eddystone-URL: un URL compresso che, una volta analizzato e decompresso, è direttamente utilizzabile dal cliente sul suo browser web;
- Eddystone-TLM: un blocco di informazioni di telemetria contenente stato del beacon e valori runtime.

L'architettura di Eddystone è fatta in modo tanto flessibile da lasciar aperto a nuovi formati di frame futuri, non limitandosi ai tre appena esposi.

3.1.4 Proximity Beacon API

L'PBA consente di registrare beacon con Google, e di amministrare i dati associati con i beacon che il progetto utilizza. È inoltre possibile associare degli allegati ai beacon. Gli allegati contengono dati di qualsiasi tipo che sono ospitati in modo sicuro sui server di Google. Quando la tua applicazione rileva un particolare beacon o un insieme di beacon, è possibile mandare i dati degli allegati al l'utente con le Nearby API (scelta consigliata) o il metodo beaconinfo.getforobserved da parte di PBA.

Si possono utilizzare diversi metodi di PBA per inviare i dati degli allegati:

- Dati di registrazione e i dati degli allegati vengono utilizzati per consentire esperienze context-aware nelle tue applicazioni, e tutte le altre applicazioni che si sceglie di condividere.
- Dati di registrazione possono essere utilizzati anche per migliorare l'API e servizi di Google correlati. Ad esempio, se si associa il beacon con un ID luogo tramite l'API di prossimità questi dati verranno utilizzati per migliorare la precisione di Google Place Picker.

Dal momento che gli allegati vengono memorizzati nel cloud, le PBA forniscono un modo scalabile, a bassa latenza per gestire e aggiornare i dati associati con i beacon distribuiti. Questo assicura che gli utenti vedono sempre gli ultimi dati disponibili, ed elimina la necessità di fornitura degli allegati in maniera manualmente ai beacon.

3.2 XMPP

3.2.1 Introduzione

Extensible Messaging and Presence Protocol (XMPP) (precedentemente noto come Jabber) è un insieme di protocolli aperti estremamente semplici basati su scambio di messaggi, pensato per scambiarsi informazioni in applicazioni per messaggistica istantanea e presenza basato su XML. Jeremie Miller, il suo fondatore, iniziò il progetto nel 1998; il suo primo rilascio pubblico principale avvenne nel marzo 2000. Il prodotto principale del progetto è jabberd, un server al quale i client XMPP si connettono per rendere possibile la conversazione. Questo server può creare una rete XMPP privata (dietro a un firewall, ad esempio), o può far parte di una rete XMPP globale e pubblica. È nato nel periodo in cui fare istant messaging significava scambiarsi messaggi testuali fra utenti correntemente connessi alle reti tramite una qualche applicazione. XMPP è un formato standardizzato e questo permette che i messaggi scambiati sono interoperabili su diverse applicazioni inoltre si possono scambiare informazioni sulla presenza.

Tutte le social networking application di rilievo usano questo formato per scambiarsi i messaggi fra i server e i clienti quando fanno istant messaging. Come dice il nome stesso, serve a fare scambio di messaggi pensato anche per la Presence: pensato per trasferire l'informazione applicativa sulla corrente connessione di un utente alla rete/applicazione e se un utente è in grado di essere raggiunto da un messaggio di chat oppure no (per esempio nelle

chat di un social network come Facebook quando qualcuno si collega viene mandato un messaggio XMPP verso l'infrastruttura che notifica che l'utente è correntemente connesso e viene notificato a tutti gli "amici" dell'utente). È a tutti gli effetti una tecnologia a scambio di messaggi. Vengono scambiati messaggi (tipicamente di presenza e stato) anche utilizzando meccanismi pub-sub. Nel protocollo è previsto che un'entità possa diventare subscriber di messaggi di un certo tipo (per esempio tutti i vostri amici Facebook di un utente sono subscribed al messaggio XMPP di presenza di quell'utente multicasted 1 a N a tutti i subscribed). Dal punto di vista architetturale il modello è puro C/S: Il cliente manda il messaggio al servitore che lo distribuisce. Se abbiamo bisogno di più servitori, questi son organizzati tra loro peer2peer fino a che non arriviamo all'ultima foglia che è chi deve ricevere il messaggio stesso.

Esistono tre tipologie di utilizzo di XMPP:

- Presence stanza: modalità XMPP secondo modello pub-sub. Chi manda il messaggio lo manda ad un gruppo di subscriber interessati a quel messaggio. Concetto di topic di JMS portato su XMPP;
- Message stanza: in cui lo scambio di messaggi XMPP è pensato 1 a 1. A tutti gli effetti è la coda JMS;
- Info/Query stanza: usare lo scambio di messaggi per modellare invocazioni C/S con semplice interazione request/response. Il cliente produce un messaggio XMPP, chi legge capisce che deve svolgere un'operazione, produrre un messaggio di risposta e rispedirlo indietro in info/query stanza (invocazione remota di metodo come in corba).

Nello standard non è specificato su quale protocollo sottostante debba essere trasferito il messaggio. Il formato dei messaggi XMPP è un formato verboso XML basato su stream: ci si può collegare a messaggi precedenti. I vantaggi e gli svantaggi di un formato XML sono: verboso, costoso nel processamento ma molto generale e molto dinamico (ci sono possibilità di estendere l'uso dello scambio dei messaggi a finalità diverse).

Un cliente che voglia inviare messaggi XMPP deve collegarsi al server e seguire il protocollo. Un punto particolarmente inefficace è che lo standard prevede che ogni volta che cade la connessione di trasporto che stiamo utilizzando per collegarci all'XMPP server, si debba iniziare una nuova sessione di interazione. Anche se è molto utilizzato nel mobile, è piuttosto inadatto allo stesso: ogni sessione d'interazione deve iniziare da capo tutte le volte che si perde la connessione di trasporto (ad esempio tutte le volte che ci si sposta e si cambia la rete

di accesso). Questa cosa è stata così sentita che in Android si è deciso di usare il supporto XMPP ma implementato in una sua variante (fuori dallo standard):

- Non si è più obbligati a usare messaggi formattati xml;
- Tutte le volte che s'interrompe la connessione non c'è una nuova sessione d'interazione: si riconosce la mobilità della sessione d'interazione.

XMPP al giorno d'oggi si usa soprattutto per propagare informazioni di presenza. I sistemi classici sono: avete un cliente applicativo che si connette a un XMPP server con l'informazione di presenza, qui ci sono tutte le informazioni di sottoscrizione di tutti quelli interessati all'informazione di presenza e da qui partono i messaggi per tutti gli interessati.

In uno schema banale con un unico XMPP server centralizzato, partono direttamente verso gli altri subscriber. Molto più spesso abbiamo XMPP server distribuiti e un minimo di coordinamento tra loro. Ci saranno aggregatori di tutte le subscription per un dominio. Quando arriva un messaggio che fa match con la sottoscrizione il messaggio va a tutti i sottoscrittori locali con uno schema a grafo a due livelli.

Livello alto dato dai server e livello basso dato dai client che sono in grado solo di inviare messaggi XMPP e riceverli, mentre i server sono in grado di coordinarsi e scambiarsi informazioni di sottoscrizione Non è detto che i messaggi XMPP viaggino sullo stesso protocollo di trasporto. Esistono già dei server XMPP chiamati gateway capaci di tramutare un messaggio XMPP in formato SMS o SMTP. Messaggi che possono diventare quindi sms o email.

Tra i punti di forza di XMPP ci sono:

- Sistema decentralizzato: l'architettura di XMPP è simile alle email; chiunque può realizzare il proprio server XMPP e non s'identificano server centrali;
- Standard aperto: l'Internet Engineering Task Force ha formalizzato XMPP come tecnologia approvata per la messaggistica istantanea (definito negli RFC 6120 e RFC 6121). Non sono previste royalty per l'implementazione di queste specifiche;
- Diffusione: le tecnologie XMPP sono utilizzate dal 1999. Esistono molte implementazioni dello standard XMPP per client, server e sono stati realizzati molti componenti e librerie;
- Sicurezza: i server XMPP possono essere isolati dalla rete pubblica, e la sicurezza viene affidata a protocolli come SASL e TLS;

- Flessibilità: si possono realizzare funzioni proprietarie usando XMPP come base; per mantenere interoperabilità, la XMPP Standards Foundation gestisce estensioni al protocollo. Le estensioni permettono di realizzare funzionalità come chat room, gestione di rete, groupware, file sharing, videogiochi, controllo remoto di sistemi e monitoraggio, geolocalizzazione, middleware, cloud computing e VoIP;
- Molto utilizzato oramai in ambiente IoT;
- Estrema semplicità dei modelli;
- Richiama concetti usati in sistemi come JMS e CORBA.

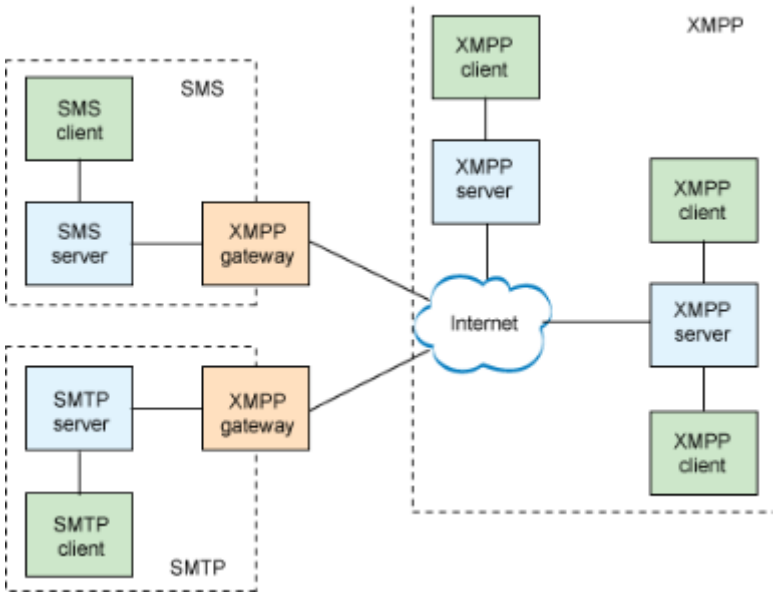


Figura 3.3 – Utilizzo di XMPP

Tra le debolezze invece troviamo:

- Non supporta Quality of Service (QoS): consegna assicurata di messaggi deve essere costruita in cima allo strato XMPP. Ci sono due XEPS proposte per affrontare questo problema, XEP-0184 e XEP-0333.
- Comunicazione text-based: poiché XML è text-based XMPP nella versione normale crea un elevato sovraccarico alla rete rispetto alle soluzioni puramente binarie.
- Trasferimento dati binario in-band è limitata: i dati binari devono essere prima codificate in base64 prima che possa essere trasmesso in-band. Pertanto, qualsiasi quantità significativa di dati binari (ad esempio, il trasferimento di file) è più conveniente trasferirlo out-of-band, utilizzando messaggi in-band solo per il coordinamento. Il miglior esempio di questo è il protocollo di estensione Jingle XMPP: XEP-0166.

- Non supporta crittografia end-to-end: a partire dal giugno 2015, ad XMPP manca il supporto nativo end-to-end di crittografia. XEP-0210 e XEPS associati hanno proposto una implementazione, ma sono stati rinviati. Off-the-Record Messaging (OTR) può essere utilizzato in cima a XMPP per la crittografia end-to-end, anche se supporta solo chat di testo per singolo utente.

3.2.2 Aspetti tecnici

In XMPP ogni entità in rete è indirizzabile in maniera univoca attraverso un indirizzo chiamato JID che può essere:

- Bare: node@domain (identifica solamente il nodo);
- Full: node@domain/resource (identifica un'entità all'interno di un nodo);

L'XML Stream è un contenitore per lo scambio di elementi XML tra due entità attraverso la rete, identificato da un tag di apertura <stream> e uno di chiusura </stream>. Al suo interno è possibile trasmettere diversi elementi XML, tra cui delle XML stanza o degli elementi di negoziazione TLS e/o SASL. L'XML stanza è un blocco d'informazioni strutturate che vengono spedite da un'entità ad una'altra all'interno dell'XML Stream. Ci sono diversi tipi di XML stanza che si differenziano a seconda del messaggio che si vuole inviare, di default il core di XMPP prevede:

- <message>

```
<message to='eve@space.com'
from='walle@earth.com/lime' type='chat' xml:lang='en'>
<body> Hi, Eve </body>
</message>
```

- <presence>

```
<presence from='walle@earth.com/lime'

 to='eve@space.com' />
```

- <iq>

```
<iq from='walle@earth.com/lime'
type='get' id='roster_1'>
<query xmlns='jabber:ip:roster'/> </iq>
```

I passi fondamentali per stabilire una comunicazione con XMPP sono:

1. Determinare l'hostname e la porta a cui connetterci;
2. Aprire una connessione TCP;
3. Aprire un XML stream;
4. Completare la negoziazione TLS per criptare la comunicazione (raccomandato);
5. Completare la negoziazione SASL per l'autenticazione;
6. Collegare una risorsa allo stream;
7. Scambio di un numero illimitato si XML stanza con altre entità della rete;
8. Chiudere l'XML stream;
9. Chiudere la connessione TCP.

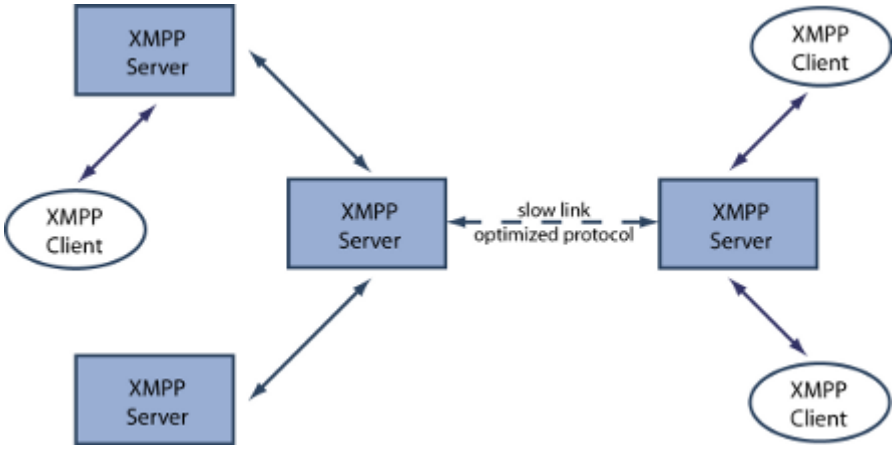


Figura 3.4 – Schema funzionamento XMPP

Tradizionalmente, le applicazioni di Instant Messagging uniscono i seguenti fattori:

- Il punto centrale di focus è una lista dei propri contatti o "amici" (in XMPP questa lista è chiamata "roster");
- Lo scopo di utilizzare tale applicazione è scambiare relativamente brevi messaggi di testo con particolari contatti in quasi tempo reale; spesso un numero elevato di tali messaggi in rapida successione, sotto forma di one-to-one 'chat session';
- Il catalizzatore per lo scambio di messaggi è la "presenza": vale a dire, informazioni sulla disponibilità della rete di particolari contatti (sapendo quindi chi è online e disponibile per una sessione di chat uno a uno).
- Le informazioni sulla presenza sono fornite solo ai contatti che sono autorizzati per mezzo di un accordo esplicito chiamato "presence subscription".

Così a un livello più alto un utente deve essere in grado di completare i seguenti casi d'impiego:

- Gestire gli elementi nella propria lista dei contatti;
- Scambiare messaggi con i propri contatti;
- Avere informazioni sulla presenza di uno o più contatti;
- Gestione delle sottoscrizioni di presenza da e per i contatti.

3.2.3 Presence stanza

Il concetto di presenza si riferisce alla disponibilità di un'entità per la comunicazione su una rete. Al livello più semplice, la presenza è una variabile booleana on/off che segnala se un soggetto è disponibile o non disponibile per la comunicazione. In XMPP, la disponibilità di un'entità viene segnalato quando il suo client genera un 'stanza' <presence/> con nessun attributo 'type', e la mancanza di disponibilità di un'entità viene segnalato quando il suo client genera una stanza <presence /> il cui attributo 'type' ha un valore di "unavailable ".

Il concetto di presenza in XMPP segue generalmente un modello "publish-subscribe" o il pattern "observer", in cui l'entità invia la sua presenza al suo server, e il server poi trasmette tali informazioni a tutti i contatti dell'entità che hanno una sottoscrizione alla presenza dell'entità (nella terminologia dettata dal RFC 2778 un'entità che genera presenza è un "presentity" e le entità che ricevono presenza sono "subscribers "). Un client genera presence in broadcast a tutte le entità sottoscritte con l'invio di una presence stanza al suo server senza specificare nessun indirizzo, dove la presence stanza non ha alcun attributo 'type' o con attributo 'type' con valore 'unavailable'. Questo tipo di presenza è chiamato "broadcast presence" (un client può anche inviare "directed presence ", vale a dire, una presence stanza con un 'to' che contiene l'indirizzo, questo è meno comune, ma a volte è utilizzato per inviare la presenza ad entità che non sono sottoscritte alla presenza dell'utente nel server).

Dopo che il cliente ha completato le pre-condizioni specificate di XMPP-CORE si può stabilire un 'presence session' presso il proprio server inviando un messaggio di presenza iniziale, invece per far terminare la sessione di presenza basta che s'invii una presence stanza con 'type=unavailable'. In XMPP generalmente le applicazioni combinano le funzioni di messaggistica con quelli di presenza, ma ciò non è obbligatorio.

Per quanto riguarda il cliente, dopo aver completato le precondizioni descritte nel XMPP-CORE (obbligatorio) e aver fatto richiesta al roster (raccomandato) può segnalare la propria

disponibilità a comunicare inviando il suo “initial presence” al suo server, vale a dire, invia una presence stanza senza l’attributo “to” che indica l’indirizzo e senza nessun attributo “type”.

```
UserClient: <presence/>
```

Il server dal canto suo, una volta ricevuto la presenza iniziale dal cliente, deve mandare un “initial presence stanza” con lo JID completo dell’user a tutti i contatti che si sono sottoscritti alla presenza di quell’utente, tali contatti sono quelli per i quali uno JID è presente nel roster dell’utente con l’attributo ‘subscription’ impostato su un valore “from” o “both”.

```
UserServer:<presence from=’examplefrom/res’ to=’exampleto1’>
UserServer:< presence from=’examplefrom/res’ to=’exampleto2’>
```

Successivamente è il server del singolo contatto che deve mandare la presence stanza al vero e proprio utente che è sottoscritto. Quando il client del contatto riceve l’informazione di presenza da parte dell'utente, è consigliato il seguente comportamento per i client interattivi:

- Se lo JID dell'utente è nel roster del contatto, si possono visualizzare le informazioni sulla presenza in un'interfaccia roster adeguato;
- Se l'utente non è nel roster del contatto ma il contatto e l'utente si scambiano attivamente messaggi o IQ stanza, visualizzare le informazioni sulla presenza nell'interfaccia utente per quella sessione di interazione;
- In caso contrario, ignorare le informazioni sulla presenza e non visualizzarlo al contatto.

3.2.4 Presence probe

Un "presence probe" è una richiesta d’informazioni sull’attuale presenza di un contatto, inviata a nome di un utente dal server dell'utente; sintatticamente è una presence stanza il cui attributo 'type' ha un valore di "probe". Nel contesto delle sottoscrizioni alle presenze, il valore del 'from' deve essere lo JID dell'utente sottoscritto e il valore del 'to' deve essere lo JID del contatto a cui è sottoscritto l'utente, dal momento che le sottoscrizioni delle presenze sono basate sullo JID.

```
UserServer: <presence from=’examplefrom’ id=’ign291v5’ to=’exampleti2’ type=’probe’/>
```

Presence probe non deve essere inviato da un client, perché in generale non avrà mai bisogno di mandarli in quanto il compito della raccolta delle presenza dei contatti di un utente viene gestito dal server dell'utente. Tuttavia, se il client di un utente genera un presence probe in uscita poi il server dell'utente dovrebbe inviare il probe (se il contatto è in un altro server) o elaborare il probe (se il contatto è allo stesso server).

Quando un server ha bisogno di scoprire la disponibilità di un contatto dell'user, manda un presence probe con 'from' lo JID dell'user e con 'to' lo JID del contatto. Il server manda di norma periodicamente dei presence probe ai contatti se non riceve nessuna informazione sulla presenza di essi o del traffico da quel contatto, questo approccio serve a prevenire contatti cosiddetti 'fantasma' che compaiono online quando invece non lo sono. Una volta che il server del contatto riceve una presence prove con JID del contatto del server dell'utente, deve rispondere in uno dei seguenti modi:

- Se il contatto non esiste, o se lo JID dell'user nel roster del contatto ha uno stato di sottoscrizione diverso da "From", "From + Pending Out" o "Both" il server del contatto dovrebbe rispondere con una presence stanza di tipo 'unsubscribed' in risposta al presence probe;
- Se invece il contatto è stato rimosso temporaneamente o permanentemente su un altro indirizzo, il server dovrebbe rispondere con una presence stanza di tipo 'error' con un tag error in cui si spiega il tipo di errore e si mette eventualmente il nuovo indirizzo del contatto;
- Oppure se il contatto esiste, ma non è presente, il server dovrebbe risponde con una presence stanza di tipo 'unavailable' con eventualmente un tag in cui ci sono delle informazioni sull'ultimo momento in cui il contatto ha generato l'ultima presence stanza;
- Altrimenti, se il contatto è disponibile, il server deve rispondere con una presence probe in cui è compreso l'ultima presence stanza senza l'attributo 'to' che ha ricevuto da quel contatto.

Dopo l'invio della presenza iniziale, in qualsiasi momento durante la sua sessione lo user può aggiornare la propria disponibilità inviando una presence stanza con nessun valore per l'attributo 'to' e l'attributo 'type', però può contenere elementi <priority/>, <show/> e una o più istanza dell'elemento <status/>. Il server, dal canto suo, appena riceve una presence stanza di aggiornamento, deve trasmette l'intero XML a tutti i contatti che sono nel roster dell'utente

con una subscription di tipo ‘from’ o ‘both’. Il server del contatto dopo aver ricevuto la presenza dell’utente deve consegnare la presence stanza al vero e proprio contatto. Dal punto di vista del contatto, non vi è alcuna differenza significativa tra la trasmissione di initial presence e il successivo update, quindi il contatto segue le stesse regole che segue quando arriva per la prima volta un presence stanza da quell’utente.

3.2.5 Unavailable presence

Prima di terminare la sua sessione di presenza con un server, il client dovrebbe chiudere la sessione mandando un unavailable presence, vale a dire, invia una presence stanza che non ha l’attributo ‘to’ a che ha l’attributo ‘type’ con valore ‘unavailable’.

```
UserClient: <presence type=’unavailable’> <status>going on vacation</status> </presence>
```

Opzionalmente l’unavailable prensence stanza può contenere uno o più elementi <status/> che specificano il motivo per cui il client non è più disponibile.

Per quanto riguarda il server dello user se una risorsa diventa indisponibile per qualsiasi motivo (sia con chiusura gracefully sia senza) deve trasmettere il nuovo stato della risorsa a tutti i contatti che sono nel roster dello user con sottoscrizione del tipo ‘from’ o ‘both’.

```
UserServer: <presence from=’example@web.com/res’ to=’exampleto@web.com’ type=’unavailable’> <status>going on vacation</status> </presence>
```

Una volta che il server dei contatti riceve questa notifica dallo user, il server deve consegnare la presence stanza a tutti i contatti attivi.

3.2.6 Directed Presence

In generale, un client invia directed presence quando desidera condividere le informazioni sulla disponibilità con una entità che non è sottoscritta alla sua presenza, in genere su base temporanea. Gli usi comuni di presenza diretta includono sessioni casuali one-to-one di chat, e stanze di chat multi-utente. Il rapporto temporaneo stabilito condividendo la presenza diretta con un'altra entità è secondario al rapporto permanente stabilito attraverso una sottoscrizione. Pertanto, gli atti di creazione, modifica o cancellazione di una sottoscrizione presenza deve prevalere sulle regole specificate per la directed presence. Ad esempio, se un utente condivide la presenza diretta con un contatto, ma poi aggiunge il contatto al roster dell'utente

completando la sottoscrizione, il server dell'utente deve trattare il contatto proprio come sarebbe qualsiasi sottoscrittore normale, per esempio, inviando successive trasmissioni di presenza al contatto. Un server XMPP tipicamente implementa le presenze dirette tenendo un elenco delle entità (bare JID o full JID) a cui un utente ha inviato presenza diretta durante la sessione corrente dell'utente per una data risorsa (full JID), quindi viene cancellata la lista quando l'utente va offline (ad esempio, mediante l'invio di una presence stanza di tipo "unavailable"). La directed presence è una presence stanza generato dal client con un attributo 'to' il cui valore è un bare JID o full JID dell'altra entità e con nessun attributo 'type' o un attributo 'type' il cui valore è "unavailable".

Quando il server dell'utente riceve una directed presence stanza, dovrebbe elaborarlo secondo le seguenti regole:

1. Se l'utente invia una disponibilità/indisponibilità attraverso directed presence ad un contatto che è nel roster dell'utente con un tipo di abbonamento di "from" o "both" dopo aver inviato la presenza iniziale e prima dell'invio della presenza (cioè, durante la presence session dell'utente), il server dell'utente deve localmente o in remoto consegnare l'intero XML di quella presence stanza ma non deve modificare lo stato per quanto riguarda la presenza di trasmissione del contatto.
2. Se l'utente invia il directed presence ad un'entità che non è nel roster dell'utente con una sottoscrizione di tipo "from" o "both" durante la sessione presenza dell'utente, il server dell'utente deve localmente o in remoto consegnare l'intero XML di quella presence stanza a quell'entità, ma non deve modificare lo stato del contatto per quanto riguarda la trasmissione della presenza; tuttavia, se la risorsa disponibile da cui l'utente ha inviato la presenza diretta diventa non disponibile, il server dell'utente deve notificare l'indisponibilità all'entità del contatto (se l'utente non ha ancora inviato una directed presence di non disponibilità);
3. Se l'utente invia un directed presence senza prima inviare la presenza iniziale o dopo aver inviato la trasmissione di presenza di non disponibilità (ad esempio, la risorsa è collegato ma non è disponibile), il server dell'utente deve trattare l'entità a cui l'utente invia il directed presence come nel precedente caso #2.

Dal punto di vista del server del contatto, non vi è alcuna differenza significativa tra la trasmissione di presenza e la presenza diretta, quindi il server del contatto segue le regole per l'elaborazione di presenza in entrata semplice. Dal punto di vista del cliente del contatto, non

vi è alcuna differenza significativa tra la trasmissione di presenza e la presenza diretta, pure in questo caso il cliente del contatto segue le regole per l'elaborazione di presenza in entrata.

Se un client ha inviato directed user ad un'altra entità (ad esempio, un partner di una chat one-to-one chat o in una chat room multi-utente), dopo un certo tempo l'entità o il suo server potrebbero voler sapere se il cliente è ancora in linea. Questo scenario è particolarmente comune nel caso di chat room multi-utente, in cui l'utente potrebbe essere un partecipante per un lungo periodo di tempo. Se il client passa offline senza che la chat room sia informata (sia da parte del client o server del client), l'utente nella chat room potrebbe diventare un "fantasma". Per rilevare questi "fantasmi", alcune implementazioni chat room multiutenti mandano delle presence probes agli utenti che si sono uniti nella chat. Nel caso di directed presence, l'entità che esegue il presence probe dovrebbe inviarlo dallo JID da cui ha ricevuto la directed presence (che sia full JID o un bare JID). Il probe deve essere inviata al full JID dell'utente, non al bare JID dell'utente perché l'autorizzazione temporanea sulla directed presence si basa sul full JID da cui l'utente l'ha inviato presenza all'entità. Quando il server dell'utente riceve un probe, è necessario applicare una logica associata con i sottoscrittori di presenza come descritto precedentemente. Se l'entità di probe non ha una sottoscrizione alla presenza dell'utente, il server deve controllare se l'utente ha inviato una presenza diretta al soggetto durante la sessione in corso; in tal caso, il server dovrebbe rispondere al probe con la sola mera presenza di tipo "disponibile" o "non disponibile" (cioè non inclusi gli elementi figlio) e solo per quel full JID (vale a dire, non per tutte le altre risorse che potrebbero essere attualmente associati al bare JID dell'utente).

3.3 Apache Cassandra

3.3.1 Introduzione

Cassandra è un database management system non relazionale distribuito con licenza open source. È altamente scalabile, ad alte performance progettato per gestire grandi quantità di dati su più commodity server, fornire il servizio ad alta disponibilità senza un singolo punto di fallimento. È fault-tolerant, elastico e con consistenza dei dati regolabile sia in read che in write. Si tratta di un tipo di database NoSQL.

Il codice di Cassandra è stato inizialmente sviluppato all'interno di Facebook per potenziare la ricerca all'interno del sistema di posta. Nel luglio del 2008 sono stati resi disponibili le sorgenti su Google Code; nel marzo 2009 è entrato a far parte del progetto Incubator di

Apache, data in cui l'interno progetto ha iniziato a essere distribuito sotto l'Apache License 2. Nel febbraio 2010 è diventato uno dei progetti top-level di Apache.

Quindi le caratteristiche principali sono:

- È scalabile, fault-tolerant e consistente;
- È un database column-oriented;
- La sua architettura distribuita è ispirata ad Amazon Dynamo (sistema sviluppato da Amazon a partire dal 2004 ed è un servizio in hosting all'interno dell'infrastruttura AWS. È stato creato per aiutare ad affrontare alcuni problemi di scalabilità del sito web Amazon.com. Al giorno d'oggi è usato per aumentare ala potenza di Amazon Web Services. È basato sui principi di scalabilità incrementale, simmetria, decentralizzazione e eterogeneità.) invece il modello per i suoi dati è basato su Google Bigtable (sistema di storage sviluppato da Google a partire dal 2004. Associa 2 valori di stringa arbitrari – valore chiave di riga e valore chiave di colonna – e un timestamp in un array di byte. Non è un database relazionale e può essere definito come una mappa ordinata, multidimensionale e sparsa.);
- Cassandra implementa un modello Dynamo-style per la replicazione senza nessun punto di fallimento, ma aggiungendo un data model per le 'column family' più potente;
- È usato da grosse compagnie come Facebook, Twitter, Cisco, Rckspace, eBay, Netflix e altro.

Cassandra è diventata così popolare per le sue eccellenti caratteristiche tecniche:

- Scalabilità elastica: Cassandra è altamente scalabile; permette di aggiungere più hardware per ospitare più clienti e più dati per soddisfare ogni requisito.
- Sempre su architettura: Cassandra non ha singolo punto di guasto ed è continuamente disponibile per le applicazioni business-critical che non possono permettersi un fallimento.
- Performance che scalano linearmente: Cassandra è linearmente scalabile, cioè, aumenta la produttività come si aumenta il numero di nodi del cluster. Pertanto mantiene un tempo di risposta rapido.
- Memorizzazione dei dati flessibile: Cassandra ospitare tutti i possibili formati di dati, tra cui: strutturati, semi-strutturati e non strutturati. Può ospitare dinamicamente modifiche alle strutture di dati secondo necessità.

- Distribuzione facile dei dati: Cassandra offre la flessibilità necessaria per distribuire i dati quando è necessaria la replicazione dei dati su più data center.
- Supporto alle transazioni: Cassandra supporta le proprietà ACID (atomicity, consistency, isolation, durability) a differenza dei maggiori altri database NoSQL.
- Scritture rapide: Cassandra è stato progettato per funzionare su hardware commodity a basso costo. Esegue scritture incredibilmente veloci ed è in grado di memorizzare centinaia di terabyte di dati, senza sacrificare l'efficienza di lettura.

3.3.2 Database NoSQL

Un database NoSQL (a volte chiamato come Not Only SQL) è un database che fornisce un meccanismo per archiviare e recuperare dati come per le tabelle relazionali utilizzate nei database relazionali. Questi database sono schema-free, supportano la replica facile, hanno delle API semplici, eventualmente consistenti, e in grado di gestire enormi quantità di dati.

L'obiettivo primario di un database NoSQL è avere: semplicità di design, scaling orizzontale, e un controllo più preciso sulla disponibilità. I database NoSQL utilizzano strutture di dati diverse rispetto ai database relazionali. Ciò rende alcune operazioni più velocemente in NoSQL. L'idoneità di un determinato database NoSQL dipende dal problema che deve risolvere.

La seguente tabella metterà in confronto i punti principali di differenza tra i database relazionali e quelli NoSQL:

Database relazionali	Database NoSQL
Supporta un potente linguaggio di query	Supporta un semplice linguaggio di query
Ha uno schema fisso	Nessuno schema fisso
Segue ACID (atomicity, consistency, isolation e durability)	Solo eventualmente consistenza
Supporta le transazioni	Transazioni con supportate

Tabella 3.1 – Confronto Database relazionali e NoSQL

Oltre Cassandra, ci sono i seguenti database NoSQL che sono abbastanza popolari:

- Hbase: è un database open source, non relazionale, distribuito sul modello di Google BigTable ed è scritto in Java. È sviluppato come parte del progetto Apache Hadoop e si esegue all'inizio del HDFS, fornendo funzionalità BigTable simile per Hadoop.
- MongoDB: è un sistema di database multipiattaforma document-oriented che evita di usare la tradizionale struttura del database relazionale basato su tabelle a favore dei documenti JSON-like con schemi dinamici rendendo l'integrazione dei dati in alcuni tipi di applicazioni più semplice e veloce.

3.3.3 Architettura

L'obiettivo del progetto di Cassandra è quello di gestire i carichi di lavoro di grandi quantità di dati su più nodi senza alcun single point of failure. Cassandra ha un sistema peer-to-peer distribuito attraverso i suoi nodi, e i dati sono distribuiti tra tutti i nodi di un cluster. Tutti i nodi di un cluster svolgono lo stesso ruolo. Ogni nodo è indipendente e allo stesso tempo interconnesso ad altri nodi. Ogni nodo in un cluster può accettare le richieste di lettura e scrittura, indipendentemente da dove il dato si trova effettivamente nel cluster. Quando un nodo va in down, le richieste di lettura/scrittura possono essere servite da altri nodi della rete.

In Cassandra, uno o più nodi in un cluster attuano le repliche dei vari pezzi di dati. Se viene rilevato che alcuni dei nodi rispondono con un valore out-of-data, Cassandra restituirà il valore più recente al client, successivamente, Cassandra esegue una riparazione di lettura in background per aggiornare i valori non aggiornati.

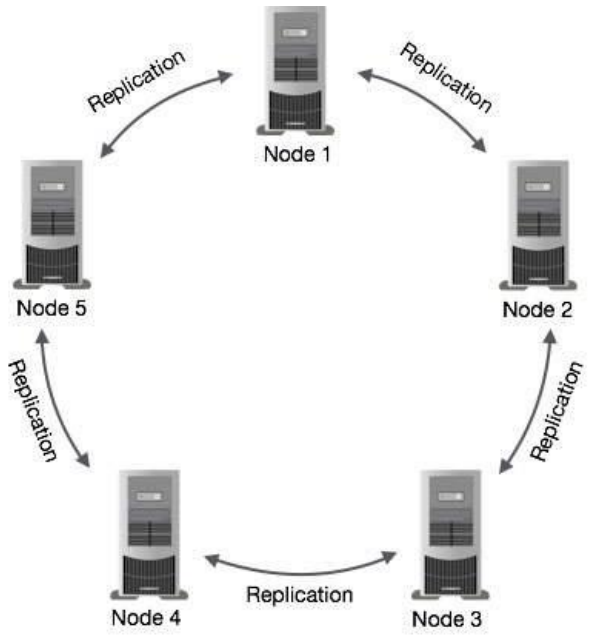


Figura 3.5 – Replicazione in Cassandra

La figura precedente mostra una vista schematica di come Cassandra utilizza la replica dei dati tra i nodi di un cluster per garantire l'assenza di single point of failure. Cassandra utilizza un protocollo chiamato Gossip in background per permettere ai nodi di comunicare fra loro e rilevare nodi che sono in down nel cluster.

I componenti chiave di Cassandra sono le seguenti:

- Node: è il luogo in cui sono memorizzati i dati.
- Data center: si tratta di un insieme di nodi correlati.
- Cluster: un cluster è un componente che contiene uno o più data center.
- Commit log: il registro dei commit è un meccanismo crash-recovery in Cassandra. Ogni operazione di scrittura è scritta nel registro dei commit.
- Mem-table: un mem-table è una struttura di dati residente in memoria. Dopo il commit log, i dati verranno scritti nei mem-table. A volte, per una singola colonna, ci saranno più mem-table.
- SSTable - è un file su disco in cui i dati vengono eliminati da mem-table quando il contenuto raggiunge un valore di soglia.
- Filtro Bloom: questi non sono che filtri rapidi, non deterministici, algoritmi per testare se un elemento è un membro di un set. Si tratta di un particolare tipo di cache. Ai filtri bloom si accede dopo ogni query.

3.3.4 Cassandra Query Language

Gli utenti possono accedere a Cassandra attraverso i suoi nodi utilizzando Cassandra Query Language (CQL). CQL tratta il database (keyspace) come un contenitore di tabelle. I programmatori usano cqlsh: un prompt per lavorare con CQL. I clienti si avvicinano a uno qualsiasi dei nodi per le loro operazioni di lettura e scrittura. Quel nodo (coordinatore) svolge da proxy tra il client e nodi contenenti i dati. Ogni attività di scrittura sui nodi viene catturato dai commit log nei nodi. Successivamente i dati saranno acquisiti e memorizzati nella mem-table. Ogni volta che il mem-table è pieno, i dati verranno scritti nel file di dati SStable. Tutte le scritture sono partizionate automaticamente e replicati in tutto il cluster. Cassandra consolida periodicamente gli SSTable, scartando i dati non necessari. Durante le operazioni di lettura, Cassandra ottiene valori dalla mem-table e controlla il filtro bloom per trovare lo SSTable opportuno che contiene i dati richiesti.

3.3.5 Data Model

Il modello di dati di Cassandra è significativamente diverso da quello che normalmente si vede in un RDBMS. Il database di Cassandra è distribuito su più macchine che operano insieme. Il contenitore esterno è conosciuto come cluster. Per la gestione dei guasti, ogni nodo contiene una replica, e in caso di guasto, la replica si prende cura. Cassandra organizza i nodi di un cluster, in un formato ad anello, e assegna i dati a essi.

Keyspace è il contenitore più esterno per i dati in Cassandra. Gli attributi di base di un keyspace in Cassandra sono:

- Fattore replica: è il numero di macchine in cluster che riceveranno copie degli stessi dati.
- Strategia di posizionamento: non è altro che la strategia per posizionare le repliche sull'anello.
- Column families: lo keyspace è un contenitore per un elenco di una o più famiglie di colonne. Una famiglia di colonne, a sua volta, è un contenitore di un insieme di righe. Ogni riga contiene colonne ordinate. Le famiglie di colonne rappresentano la struttura dei dati. Ogni keyspace ha almeno uno e spesso molte famiglie di colonna.

Una famiglia di colonna è un contenitore per un insieme ordinato di righe. Ogni riga, a sua volta, è una collezione ordinata di colonne.

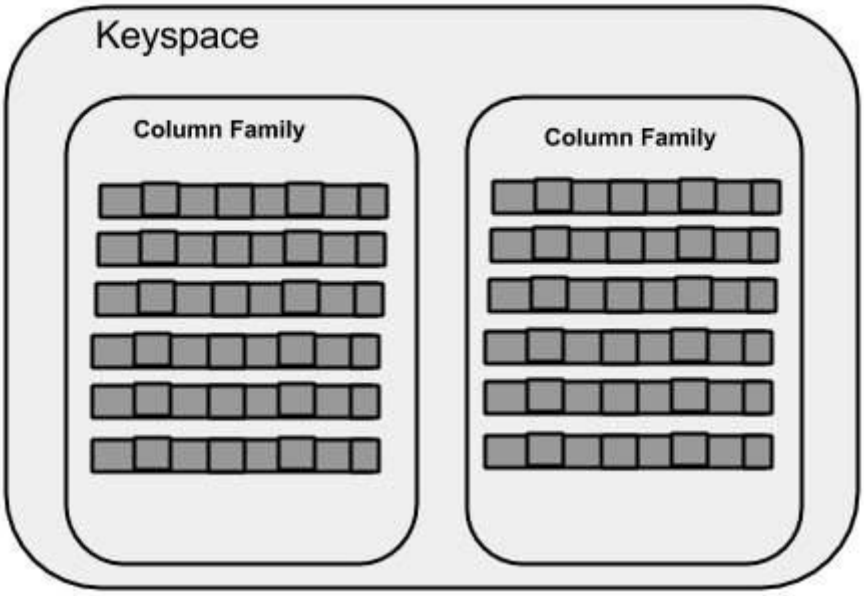


Figura 3.6 – vista schematica del keyspace

Nella tabella seguente sono elencati i punti che differenziano una famiglia colonna da una tabella di database relazionali.

Tabelle relazionali	Famiglie di colonne in Cassandra
Uno schema di un modello relazionale è fisso. Una volta che definiamo alcune colonne per una tabella, durante l'inserimento di dati, in ogni riga tutte le colonne devono essere riempiti, almeno con un valore nullo.	In Cassandra, anche se le famiglie di colonna sono definite, le colonne invece no. È possibile aggiungere liberamente qualsiasi colonna a ogni famiglia di colonne in qualsiasi momento.
Le tabelle relazionali definiscono solo colonne e l'utente compila la tabella con i valori.	In Cassandra, una tabella contiene colonne, o può essere definito come una super famiglia di colonna.

Tabella 3.2 – Confronto le tabelle e le famiglie di colonne

Una column family ha i seguenti attributi:

- keys_cached: esso rappresenta il numero di posizioni per tenere in cache per SSTable.
- rows_cached: rappresenta il numero di righe il cui intero contenuto verrà memorizzato nella cache in memoria.
- preload_row_cache: specifica se si desidera precompilare la row cache.

A differenza di tabelle relazionali in cui lo schema di una column family non è fisso, Cassandra non può forzare le singole righe ad avere tutte le colonne. Una colonna (column) è la struttura dati di base di Cassandra con tre valori chiave, nome colonna, valore, e un timestamp. Una super colonna (super column) è una colonna speciale che è anche una coppia di valori-chiave. Ma una super colonna memorizza una mappa di sub-colonne. Generalmente le famiglie delle colonne sono memorizzati su disco in singoli file. Pertanto, per ottimizzare le prestazioni, è importante mantenere le colonne che si rischiano di interrogare insieme nella stessa famiglia colonna e una colonna super può essere utile in questo caso. Nella seguente tabella verranno esposte le differenze tra il modello di dati di Cassandra e quello di un generico RDBMS.

RDBMS	Cassandra
RDBMS lavora con dati strutturati	Cassandra lavora con dati non strutturati
Hanno uno schema fisso	Hanno uno schema flessibile

Una tabella è un array di array (riga x colonna)	Una tabella è una lista di coppie chiave-valori innestate (riga x chiave-colonna x valore-colonna)
Il database è il contenitore esterno che contiene i dati corrispondenti a un'applicazione	Lo keyspace è il contenitore esterno che contiene i dati corrispondenti a un'applicazione
Le tabelle sono entità di un database	Le tabelle o le famiglie di colonne sono le entità dello keyspace
Una riga è un record individuale nel RDBMS	Una riga è un'unità di replicazione in Cassandra
Le colonne rappresentano gli attributi di una relazione	Le colonne rappresentano le unità di memorizzazione
RDBMS supporta il concetto di chiave esterna e join	Le relazioni sono rappresentate utilizzando le collezioni

Tabella 3.3 – Confronto tra un RDBMS e Cassandra

Un esempio di column è:

```
{
  name:  "age",
  value: "26",
  timestamp: 123456743
}
{
  name:  "fullname",
  value: "Sandro Paganotti",
  timestamp: 836123423
}
```

Gruppi di column possono essere raggruppati all'interno di super column, che altro non sono se non column il cui campo value contiene un array di column; un esempio di supercolumn potrebbe essere il seguente:

```
{
  name:  "telephone number",
  value: {
    {
      name:  "prefix",
      value: "349",
      timestamp: 123123141241
    },
    {
      name:  "number",
      value: "0516558976",
      timestamp:    231231231434
```

```
        }
    }
}
```

Una piccola differenza che si può notare che nelle super colonne non c’è il campo timestamp.

Capitolo 4 – ICoS (Instant Coupon on Storm)

4.1 ICoS: Requisiti e ipotesi

Il prototipo è un'applicazione Storm che processa dati in tempo reale. I dati possono essere di due tipologie: provenienti tramite protocollo XMPP o dati presi da Eddystone. In tutte e due i casi l'obiettivo è fare profilazione dell'utente, o dati statistici per le diverse sedi in cui i clienti possono stare.

Le ipotesi che si fanno per il seguente prototipo sono le seguenti:

1. Esiste un'app per smartphone in cui il login è rappresentato dal codice della carta fedeltà;
2. Per quanto riguarda XMPP: nel momento in cui un cliente utilizza l'app per usufruire di offerte stando al supermercato questa invia una presence stanza al server che gestisce l'applicazione che poi provvederà semplicemente a inviarlo al server front-end dove esegue l'applicazione Storm tramite directed presence stanza. L'applicazione Storm riceverà e processerà in maniera intelligente le diverse presence stanza; la 'resource' che un utente mette a disposizione tramite la propria presenza è composta nel seguente modo: example@example/id_carta_fedeltà, il roster può essere considerato il server a cui si collega l'app che prende questi dati semplicemente;
3. Per quanto riguarda Eddystone: siccome ogni dispositivo Eddystone possiede un ID univoco di 128 bit, si utilizza codesto per fare processamento con Storm, l'ID del

dispositivo lo si prende tramite le Proximity Beacon API e successivamente lo si manda al server che gestisce l'app il quale provvederà a trasmetterlo al server front-end su cui gira l'applicazione Storm in maniera analoga a XMPP il quale provvederà a elaborare i dati che arrivano.

Quindi in generale ci sono due tipi di informazioni, grazie al protocollo XMPP si hanno informazioni riguardanti la posizione di utente all'interno di una filiale, invece grazie ai mobile beacon possiamo sapere sia quale utente sia dove sta facendo la spesa o anche meglio se si vuole anche in quale reparto.

Il prototipo prende queste informazioni e le raggruppa in due insiemi, un riguardante il cliente e un riguardante la locazione. I raggruppamenti riguardano semplicemente il numero di clienti che ci sono in una certa filiale o quante volte il cliente va nelle filiali. Periodicamente, in base a specifiche si memorizzano questi dati in un database non relazionale come Apache Cassandra.

4.1.1 Architettura

Per la realizzazione del prototipo si sono utilizzate le seguenti tecnologie:

- Storm 0.9.0;
- Maven 3.2.3;
- Cassandra 2.1.0;

Nella topologia sono presenti due tipi di spout, uno per ricevere i messaggi XMPP (XMPPReader) in un formato XML-based (come da standard) e uno che riceve i messaggi Eddystone in formato JSON (EddystoneReader – in questo caso il formato JSON per la scambio di messaggi è una scelta basata sul fatto di scegliere un formato leggero, facile da generare e analizzare).

XMPPReader analizza lo stream di dati che arriva in ingresso e invia un messaggio in cui sono poste le informazioni riguardanti locazione e cliente a due bolt, uno che si occupa di prendere dal messaggio solo la parte riguardante il posto (PlaceSplitterXMPP) e l'altro il cliente (ClientSplitterXMPPP). Discorso analogo per EddystoneReader, solo che per il posto c'è un bolt in più in quanto nel messaggio in formato JSON viene passato l'ID del dispositivo con cui si è comunicato tramite Eddystone (si ricorda che ogni dispositivo secondo lo standard Eddystone deve avere un ID univoco di 128 bit), quindi da quell'ID univoco bisogna

ricavare il posto in cui si trova il dispositivo. Per fare ciò viene effettuato un accesso in lettura su Cassandra in cui sono memorizzate le associazioni ID/locazione.

PlaceSplitterXMPP quindi si occupa di prendere la parte riguardante il luogo dal messaggio che riceve dallo spout. Analogamente per PlaceSplitterEddystone ricordando che esiste un altro bolt chiamato PlaceFromID che da un ID di 128 bit riesce a prendere il nome univoco della locazione corretta. Ci sono poi ClientSplitterXMPP e ClientSplitterEddystone che si occupano di prendere la parte riguardante i clienti da messaggi che ricevono dagli spout.

Successivamente ci sono due bolt predisposti per il conteggio dei clienti (ClientCounter) e delle locazioni (PlaceCounter), i due contatori, nel prototipo, si azzerano a intervalli di tempo e memorizzano quelli che sono i loro risultati parziali.

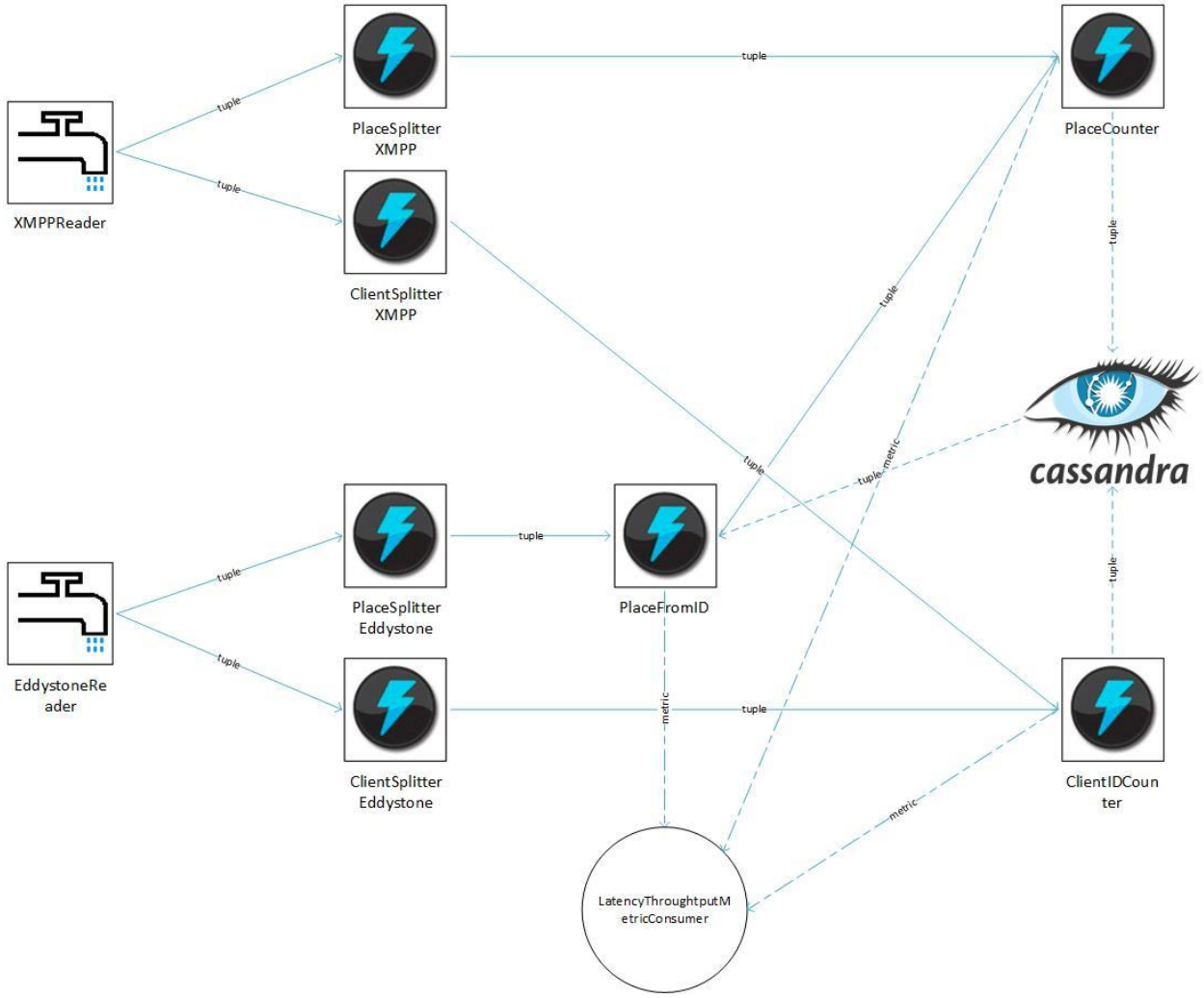


Figura 4.1 – Il data flow di ICoS

C’è inoltre una classe che funge da metric consumer che è LatencyThroughputMetricConsumer che riceve metriche dalle classi PlaceFromID, PlaceCounter e ClientCounter e utilizza queste informazioni per applicare un bilanciamento di carico se è necessario.

4.1.2 Introduzione al problema del parallelism hint

Il parallelism hint deve essere deciso in maniera manuale e statica, per questo è importante fare il maggiore numero di test prima di decidere i valori di soglia. Questo ovviamente rappresenta un problema in quei sistemi in cui la mole di dati in ingresso possono variare e di molto nell’arco della giornata. E questo prototipo rientra tra questi casi. Infatti per quanto riguarda questo caso specifico la maggior parte dei clienti va a fare spesa di solito negli stessi orari, quindi si possono prevedere dei picchi nella distribuzione della mole di dati in ingresso relativamente alti. Sarebbe auspicabile quindi un sistema per il load balancing in modo tale da non avere uno spreco di risorse e soprattutto in modo tale da far funzionale l’applicativo al massimo delle sue potenzialità. Ciò permette di eliminare il bisogno di fare scalare il sistema manualmente, permettere di far scalare le topologie in base a delle regole prefissate o anche dinamiche. Questo permette di cambiare configurazione in maniera automatica nel momento in cui il sistema rileva che si è superato un certo valore di soglia. La configurazione voluta deve essere raggiunta in pochi minuti da quando si riceve la prima tupla: per far scalare manualmente un sistema bisogna aspettare ore per raggiungere i valori di soglia in una topologia grande. Ogni volta che il carico in ingresso aumenta il sistema deve essere in grado di cambiare automaticamente, e deve essere vero anche il viceversa, cioè nel momento in cui il carico diminuisce, devono diminuire anche le risorse utilizzate.

Si è utilizzato un approccio abbastanza singolare nel nostro caso specifico: invece di creare un sistema di load balancing completamente esterno il quale leggendo i file di log creati dai metric consumer e analizzandoli potesse decidere le azioni da compiere, si è deciso di mettere la logica di bilanciamento direttamente all’interno del metric consumer stesso. Questo approccio ha vantaggi e svantaggi:

- Velocità e prontezza di risposta: nessun sistema esterno può essere più veloce dello stesso metric consumer che è il primo ad analizzare i dati che arrivano dalle varie entità quindi può agire in maniera tempestiva;

- Collo di bottiglia: il principale svantaggio di tale approccio è il fatto che si andrebbe ad appesantire il metric consumer che potrebbe diventare il collo di bottiglia dell'intero sistema se non riesce ad analizzare in tempo tutti i metric che gli arrivano. C'è da considerare che l'assegnazione delle risorse del metric consumer avviene in maniera statica e ciò non può essere cambiato dinamicamente quindi bisogna essere ancora più attenti nell'assegnazione iniziale delle risorse al metric consumer. Un altro motivo per cui è importante avere un'assegnazione iniziale ottimale riguarda il fatto che si risparmia del lavoro al metric consumer non rischiando di portarlo al limite;
- Un limite invece riguarda la non possibilità di aumentare il parallelism hint all'infinito. Per quanto riguarda gli executor si ricorda che non possono superare il numero di thread iniziali assegnati a quella determinato componente quindi il numero di thread deve essere maggiore o uguale al numero di executor per ogni componente. Per quanto riguarda il numero di worker, questo dipende in maniera diretta dalla cardinalità dei supervisor. Tutti questi parametri sono importanti nella verifica se è possibile ribilanciare o meno;
- È un sistema tutt'uno non ha bisogno di sistemi fuori da esso per funzionare.

Come parametri per il load balancing sono stati scelti il throughput ovvero il numero di tuple per un intervallo di tempo che vengono processati da un'entità e la latenza esecutiva che rappresenta il tempo che trascorre da quando un'entità riceve una tupla a quando questa finisce di essere eseguita ed eventualmente inviata a un'altra entità. Si potevano scegliere anche altri parametri come le lunghezze delle code in ingresso e in uscita dei vari componenti, e grazie a queste si poteva capire se quel componente era carico eccessivamente o meno, o l'utilizzo della memoria heap e non heap da parte della topologia o dei componenti stessi.

Un aspetto da considerare è quello riguardante il carico parziale della topologia ovvero potrebbe accadere che non tutta la topologia sia eccessivamente carica ma solo una parte di essa. Ciò potrebbe essere causata ad esempio da un semplice bolt che ci mette troppo tempo rispetto alle altre entità nella topologia in quanto l'accesso a un database lo rallenta particolarmente, o perché succede qualcosa che non era previsto. In questi casi specifici bisogna considerare il fatto di poter aumentare solo il numero di executor in quanto aumentare il numero di worker porta si a risolvere il problema ma porta anche a uno spreco di risorse che potrebbero essere utilizzate in maniera più intelligente. Spesso però non è facile rendersi conto se è un caso di carico parziale o meno. Uno modo per evitare ciò risiede sempre nello

tuning iniziale delle risorse e nel fare tanti test in modo tale da rendersi conto se ci sono parti più lente di altre nella topologia.

Un altro concetto importante da considerare è il fatto che non si possono creare troppe metriche da tutte le entità in gioco nella topologia, bisogna limitarsi a quelle entità che sono più a rischio: cioè quei componenti che hanno più probabilità di altri di raggiungere il carico massimo computazionale, quindi dopo la fase iniziale di test oltre al tuning delle risorse si decide di ricevere le metriche soltanto dalle entità più a rischio. Facendo in questo modo si ha il grosso vantaggio di alleggerire il metric consumer che non riceve dati non significativi da analizzare, quindi si riduce la probabilità di fault dello stesso metric consumer.

Per decidere se bisogna bilanciare (in positivo o in negativo) si usano dei valori di soglia di riferimento. Ovviamente sarebbe preferibile che questi valori di soglia siano variabili esse stesse in base a regole stabilite ad esempio sugli orari, perché le esigenze sulle prestazioni potrebbero variare. Avere dei valori di soglia statici potrebbe rendere il sistema rigido, il minimo che si potrebbe fare è che essi possano almeno essere cambiati su comando, modificando un file di configurazione ad esempio.

4.1.3 Maven

Maven è un progetto open source, sviluppato dalla Apache, che permette di organizzare in modo molto efficiente un progetto java. Può essere paragonato all'altro progetto più conosciuto della Apache, Ant, ma fornisce funzionalità più avanzate. I vantaggi principali di Maven sono i seguenti:

- Standardizzazione della struttura di un progetto compilazione;
- Test ed esportazione automatizzate;
- Gestione e download automatico delle librerie necessarie al progetto;
- Creazione automatica di un semplice sito di gestione del progetto contenente informazioni.

Usare Maven porta a dei vantaggi tra i quali:

- Ci si rende indipendente dalla struttura dei progetti dei varie IDE utilizzati da tutti gli sviluppatori;

- Semplifica i cicli di vita dei progetti che possono avere iniziali parti comuni (dipendenze, librerie sviluppate internamente), ma che poi hanno vita diversa come aggiornamenti, variazioni e altro;
- Uniforma lo sviluppo di nuovi progetti a una serie di convenzioni predefinite.

Il processo si semplifica perché non è necessario conoscere nel dettaglio i meccanismi di compilazione, basta usare i numerosi plugin gestiti dalla comunità open-source. Grazie al POM (Project Object Model) tutte le informazioni riguardanti il progetto sono racchiuse in un solo punto. La struttura di un progetto Maven, inoltre, porta lo sviluppatore a seguire gli standard di qualità del software industriale, ad esempio: test separati dall'applicativo, documentazione aggiornata, ambiente di esecuzione di test il cui ambiente di compilazione/esecuzione non sia legato all'ambiente applicativo. Maven si occupa anche di dependency management che include: update automatico e dipendenze transitive.

I comandi, le fasi, i tipi di configurazione plug-in e i tipi di progetto sono tutti standard. Un file pom.xml di Maven specifica tutto il necessario per costruire un progetto cioè quali librerie devono essere disponibili per la compilazione, quali servono al compilatore, quale parte di codice li utilizza. STS include funzionalità (vale a dire, il plugin m2eclipse, che fornisce supporto a Maven per i derivati Eclipse) per importare direttamente i progetti Maven.

La gestione dei progetti con maven è composta da diverse fasi:

1. Validate: Maven controlla che il progetto sia configurato correttamente. Controlla la correttezza del file di configurazione e la disponibilità di tutto ciò che è richiesto dalle altre fasi.
2. Compile: questa fase compila tutti i file Java contenuti nel progetto. È utile per gestire e separare correttamente le interdipendenze tra le varie parti del progetto. Non è necessario invocare manualmente il compilatore. I progetti sono compilati utilizzando le informazioni contenute nel file POM. Maven è in grado automaticamente di capire dove trovare i file sorgenti, le librerie richieste e altre informazioni necessarie per compilare l'applicazione.
3. Test: esegue i test automatizzati e crea un rapporto con il loro esito. Un fallimento in un test bloccherà l'esecuzione delle fasi successive. Il testing è un'attività indispensabile per assicurarsi che l'applicazione funzioni correttamente. Maven si integra con JUnit. Ogni volta che viene ricreato il pacchetto con l'applicazione, tutti i test sono eseguiti.

4. Package: crea un pacchetto contenente il codice compilato. I file compilati sono “pacchettizzati” in un unico archivio. Maven supporta diversi tipi di pacchetti: jar, war, ecc. I pacchetti creati sono disponibili nella directory target. Rende più semplice generare rapporti e documentazione per i progetti: documentazione API, risultati dei test e copertura del codice, qualità del codice. Le informazioni contenute nei rapporti sono essere rese disponibili attraverso un sito web.
5. Install: copia il pacchetto nel repository locale. Una volta che il pacchetto è installato nel repository locale, può essere utilizzato come una dipendenza per altri progetti.
6. Deploy: pubblica il pacchetto in un repository remoto in modo che il pacchetto sia disponibile pubblicamente.

Le fasi sono sequenziali. Ogni fase richiede l’esecuzione corretta della fase precedente. Maven si basa su una serie di convenzioni: struttura delle directory standard, stesso modo per gestire tutti i progetti, tutto si configura attraverso un file Project Object Model (POM). Senza Maven dovremmo: cercare le librerie sul web, scaricare e configurare le librerie richieste e le loro dipendenze, ripetere il procedimento per poter eseguire l’applicazione. Usando Maven basta semplicemente specificare quali librerie servono. Per quanto riguarda la gestione delle dipendenze Maven offre supporto per più versioni delle stesse librerie. Esiste un repository a livello di sistema per evitare che librerie richieste da più progetti siano duplicate. Inoltre ci sono repository remoti con una gran quantità di software disponibile.

I comandi principali che si eseguono per questo progetto sono stati:

- clean: come dice il nome, tenta di pulire i file e le directory generati da Maven durante la sua costruzione. Mentre ci sono plugin che generano file aggiuntivi, il plugin Clean presuppone che questi file vengono generati all'interno della directory di destinazione;
- install: Maven ha una strategia a due livelli per risolvere e distribuire file, in base agli artefatti. Il primo livello è chiamato repository locale, che è la cache sul vostro sistema. Durante l'esecuzione di Maven, per la prima volta si guarderà nella cache locale per gli artefatti. Se non riesce a trovarlo in locale, Maven accederà ai repository remoti per trovare l'artefatto. Una volta trovato verrà memorizzato all'interno della repository locale, in modo che sia disponibile per l'uso attuale e futura;
- assembly:assembly: Il plugin Assembly per Maven è principalmente destinato a consentire agli utenti di aggregare l'output del progetto insieme con le sue dipendenze,

moduli, la documentazione del sito e altri file in un unico archivio distribuibile. Viene utilizzato nel nostro caso per creare l'archivio da montare su Storm poi;

4.2 Implementazione

I due spout presenti nella topologia (XMPPReader e EddystoneReader) vanno a ereditare la classe BaseRichSpout, una classe standard in Storm per la creazione di spout con già qualche funzionalità data. La classe espone tre metodi principali:

- Il metodo open() viene utilizzato per inizializzare la configurazione dello spout. Il parametro conf contiene tutte le informazioni per la configurazione del cluster Storm il parametro context viene utilizzato per ottenere informazioni relative al processo in esecuzione sul cluster; il parametro collector viene utilizzato per emettere le tuple dallo spout ai bolt (operazione svolta nel metodo nextTuple()). All'interno di questo metodo si è solo inizializzato il collector.
- Il metodo nextTuple() ha un duplice ruolo: leggere lo stream continuamente e incapsulare i dati letti in tuple per poi emetterle verso i bolt. La lettura continua dello stream viene gestita invocando il metodo ripetutamente durante tutta l'esecuzione dello spout. Questa esecuzione ripetuta pone un problema: se da un lato permette di leggere lo stream facilmente, dall'altro genera attesa attiva anche quando non ci sono dati aggiornati da elaborare. Per rendere l'esecuzione del metodo più efficiente, si può mettere in sleep il thread che esegue il metodo per poi risvegliarlo quando serve. Nella prima parte del codice, viene gestita l'attesa attiva mettendo in sleep il thread se la variabile booleana completed è true. Questa variabile viene settata a true quando termina il ciclo di lettura (seconda parte del codice). Ovviamente è necessario dichiarare questa variabile all'interno della classe. La seconda parte del codice invece effettua la lettura dello stream e a ogni riga letta, viene invocato emit(): questo è il metodo che inoltra le tuple all'esterno dello spout. I dati letti dal file vengono incapsulati in una tupla istanziando l'oggetto Values. Terminata la lettura dello stream (con successo o meno), la variabile completed viene settata a true e dunque la successiva invocazione del metodo innesca l'invocazione del metodo sleep che blocca l'attesa attiva. Ovviamente ciò non è obbligatorio e dipende dal tipo di stream che si ha in ingresso.
- Un particolare che fino ad ora non è stato sottolineato è il fatto che un bolt può ricevere tuple da più spout. Si possono differenziare le diverse tuple con il metodo

declareOutputFields() nel quale si associa un'etichetta alla tupla che lo spout emette. Possiamo incapsulare l'etichetta in un oggetto Fields e associarla alla tupla con il metodo declare() dell'oggetto OutputFieldsDeclarer. Nel nostro caso, come si può vedere nel codice sottostante, mettiamo un'etichetta di nome "from_element" in modo tale che il bolt possa fare le dovuto operazioni.

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
          declarer.declare(new Fields("from_element"));
}
```

Per quanto riguarda tutti i bolt ereditano tutti dalla classe BaseRichBolt in maniera analoga come succedeva per gli spout con BaseRichSpout. Ci sono tre metodi principali anche in questo caso nei bolt:

- Analogamente al metodo open() di uno spout, il metodo prepare() viene utilizzato per inizializzare la configurazione del bolt, infatti i parametri di ingresso sono gli stessi. Possiamo utilizzare il metodo prepare per configurare il collector.
- Il metodo execute() permette di eseguire l'elaborazione per cui viene creato il bolt. Come anticipato, le stringhe vengono incapsulate in tuple; una tupla arriva ad un bolt attraverso l'oggetto Tuple (in ingresso al metodo execute). Come è facile intuire, l'estrazione descritta avviene tramite il metodo getStringByField() dell'oggetto input, al quale passiamo come parametro l'etichetta assegnata nel metodo declare dello spout. Successivamente si effettua l'elaborazione definita. Dato che l'elaborazione non si ferma a questi bolt, dobbiamo invocare il metodo emit() in maniera duale a come si è fatto negli spout.

```
public void execute(Tuple input) {
          String sentence = input.getStringByField("from_element");
        String[] words = sentence.split(" ");
        for(String word : words){
          StringTokenizer st = new StringTokenizer(word, "@");
          String from = st.nextToken();
          from = from.trim();
            if(!from.isEmpty()){
                from = from.toLowerCase();
                collector.emit(new Values(from));
            }
        }
    }
```

- Come fatto nello spout, si può associare un'etichetta alla tupla implementando il metodo declareOutputFields():

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
```

```
        declarer.declare(new Fields("place"));
    }
```

Tutti e due i counter (PlaceCounter e ClientCounter) ereditano anche loro da BaseRichBolt. I metodi sono gli stessi che sono stati descritti precedentemente, quello che cambia profondamente è la logica.

```
public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
        this.counters = new HashMap<String, Integer>();
        time_int = System.currentTimeMillis();
}

public void execute(Tuple input) {
    String str = input.getStringByField("clientID");
    if(!counters.containsKey(str)){
        counters.put(str, 1);
    }else{
        Integer c = counters.get(str) + 1;
        counters.put(str, c);
    }
    if( (TimeUnit.MILLISECONDS.toSeconds(System.currentTimeMillis() -
this.time_int)) >= 5){
      org.apache.log4j.BasicConfigurator.configure(new NullAppender());
      //creating Cluster object
      cluster = Cluster.builder().addContactPoint("127.0.0.1").build();
      //Creating Session object
      session = cluster.connect("tesi_ardit");
       for(Map.Entry<String, Integer> entry : counters.entrySet()){
           String output = entry.getKey()+": "+entry.getValue()+"\n";
          //Query
           String query = "INSERT INTO client (name, timestamp, count)"
               +" VALUES('"+ entry.getKey()+"',"+this.time_int+",
                "+entry.getValue()+");";
           session.execute(query);
       }
       session.close();
       cluster.close();
       this.counters = new HashMap<String, Integer>();
     time_int = System.currentTimeMillis();
    }
}
```

Come si può vedere dal codice all'interno possiamo trovare anche il codice di riferimento per salvare i dati su Cassandra.

4.2.1 Implementazione Apache Cassandra

Per lavorare con Cassandra occorre innanzitutto avere un'istanza di Cluster.builder della classe com.datastax.driver.core, al quale si aggiunge un punto di contatto (l'indirizzo IP del nodo). Per avere un'istanza di Session invece basta che s'invochi il metodo connect della classe cluster, si può invocare senza argomenti o passando come argomento il nome del keyspace su cui si andrà a lavorare.

Si può usare in seguito le query CQL chiamando il metodo execute della classe Session. L'argomento del metodo execute() si può passare in due modi: o utilizzando il formato stringa per scrivere tutta la query o passando una stanza della classe Statement. Qualsiasi soluzione delle due si scelga la query poi sarà eseguita sul clqsh. Successivamente si prosegue con la chiusura della sessione e del cluster.

Per quanto riguarda invece lo keyspace su cui si lavora: nel momento della sua creazione si possono settare due proprietà che sono replication e durable_writes. La prima proprietà si riferisce alla strategia di replicazione e al numero di repliche che si vogliono scegliere. Ci possono essere tre scelte possibili per quanto riguarda la strategia di replicazione:

- Simple Strategy: specifica un fattore di replicazione semplice per il cluster;
- Network Topology Strategy: usando questa opzione si può settare un fattore di replicazione per ogni data-center in maniera indipendente;
- Old Network Topology Strategy: questa è una strategia di replicazione legacy.

Quindi avremo una riga simile alla sottostante nel momento in cui creiamo un keyspace:

```
CREATE KEYSPACE xxx WITH replication = {'class':'SimpleStrategy',
'replication_factor':3}
```

Per default invece, la proprietà durable_writes di un keyspace è settato a vero, tuttavia può assumere anche il valore falso. Non si può invece settare se si utilizza una simple strategy.

Successivamente si devono creare le due columnfamily, un riguardante i luoghi e l'altro i clienti. La sintassi è la seguente:

```
CREATE COLUMNTABLE client(
   name text,
   timestamp timestamp,
   count varint,
   PIMARY KEY(name, timestamp)
   );
```

La columnfamily dei luoghi che si chiama place ha una struttura del tutto equivalente, con tre proprietà che sono:

- Name: s'intende il nome della struttura o l'ID del cliente;
- Timestamp: è l'attimo in cui viene effettuata la memorizzazione e insieme alla proprietà name va a costituire la chiava primaria;
- Count: il numero di occorrenze del cliente o del luogo.

Tutte le operazioni su keyspace e sulle columnfamily posso essere fatte tramite le API messe a disposizione da Cassandra da codice, ma in questo caso si è preferito utilizzare CLQ per effettuarle.

Infine si è utilizzato TopologyBuilder per quanto riguarda la costruzione della topologia, in modalità locale tramite la classe LocalCluster o in modalità remoto utilizzato la classe StormSubmitter per far caricare la topologia su Storm. Nella prima parte si implementa la creazione di una topologia che avviene istanziando l'oggetto TopologyBuilder. L'associazione degli spout e dei bolt alla topologia è resa possibile rispettivamente dai metodi setSpout() e setBolt(). Entrambi i metodi ricevono in ingresso due parametri: un ID con il quale si identifica univocamente l'entità passata e l'oggetto vero e proprio. Esiste anche un terzo argomento opzionale che riguarda l'impostazione del numero di executor che di default è uno. Inoltre tramite il metodo setNumTasks() si possono impostare anche il numero effettivo di task, che di default sarebbe stato uguale al numero di executor. In questo modo si può decidere il parallelism hint iniziale in maniera statica. Con il metodo shuffleGrouping() implementiamo la tecnica dello Shuffle Grouping e in ingresso si fornisce l'ID del componente da cui riceve le tuple. Nella seconda parte viene creato un oggetto Config in cui vengono passati dei parametri, tra cui la registrazione del metric consumer tramite il metodo registerMetricsConsumer che come argomento oltre la classe vera e propria ha bisogno di un parametro che indica quanti thread devono essere creati per quel metric consumer. Questo parametro è importante se si considera il fatto che il metric consumer stesso potrebbe diventare il collo di bottiglia nella gestione delle metrics che riceve, quindi è importante scegliere la cardinalità giusta in questo caso. Successivamente, come si è detto in precedenza, grazie all'argomento che si passa nella fase di invocazione si decide se andare in modalità locale o remota. Nella modalità locale si imposta un tempo in cui la topologia deve girare e poi terminare. Nella modalità remota si imposta sempre tramite l'oggetto Config il numero di worker iniziali nel sistema. Con questo ultimo parametro si può dire completata la fase di impostazione statica delle risorse iniziali della topologia. È molto importante fare dei test e avere dei valori di riferimento per fare una scelta ottimale delle risorse iniziali. Il prototipo è si dotato di un sistema di load balancing ma è sempre meglio che sia utilizzato il meno possibile perché comunque l'operazione di rebalancing di Storm è un'operazione non immediata e soprattutto costosa in termini computazionali quindi è buona norma fare il minor numero possibile di cambiamenti.

```
public class PrototypeTopology {
```

```
    //spouts
    public final static String XMPP_READER_SPOUT_ID = "xmpp-reader";
    public final static String EDDYSTONE_READER_SPOUT_ID = "eddystone-
reader";
    //place bolts
    public final static String PLACE_COUNTER_BOLT_ID = "place_counter";
    public final static String XMPP_PLACE_SPLITTER_BOLT_ID =
"xmpp place splitter";
    public final static String EDDYSTONE_PLACE_SPLITTER_BOLT_ID =
"eddystone_place_splitter";
    public final static String PLACE_FROM_ID_BOLT_ID = "place_from_id";

    //client id bolts
    public final static String CLIENTID_COUNTER_BOLT_ID = "place_counter";
    public final static String XMPP_CLIENTID_SPLITTER_BOLT_ID =
"xmpp_clientID_splitter";
    public final static String EDDYSTONE_CLIENTID_SPLITTER_BOLT_ID =
"eddystone_clientID_splitter";

    public static void main(String[] args) throws Exception {
        //Prima parte - Definizione del TOPOLOGY
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout(XMPP_READER_SPOUT_ID, new XMPPReader(), 2);
        builder.setSpout(EDDYSTONE_READER_SPOUT_ID, new EddystoneReader(),
2);

        builder.setBolt(EDDYSTONE_PLACE_SPLITTER_BOLT_ID, new
PlaceSplitterEddystone()).
          shuffleGrouping(EDDYSTONE_READER_SPOUT_ID);
        builder.setBolt(PLACE_FROM_ID_BOLT_ID, new PlaceFromID(), 6)
          .setNumTasks(8)
          .shuffleGrouping(EDDYSTONE_PLACE_SPLITTER_BOLT_ID);
        builder.setBolt(EDDYSTONE_CLIENTID_SPLITTER_BOLT_ID, new
ClientIDSplitterEddystone()).
          shuffleGrouping(EDDYSTONE_READER_SPOUT_ID);

        builder.setBolt(XMPP_PLACE_SPLITTER_BOLT_ID, new
PlaceSplitterXMPP()).
          shuffleGrouping(XMPP_READER_SPOUT_ID);
        builder.setBolt(XMPP_CLIENTID_SPLITTER_BOLT_ID, new
ClientIDSplitterXMPP()).
          shuffleGrouping(XMPP_READER_SPOUT_ID);


        builder.setBolt(PLACE_COUNTER_BOLT_ID, new PlaceCounter(), 1)
          .setNumTasks(4)

    .shuffleGrouping(XMPP_PLACE_SPLITTER_BOLT_ID).shuffleGrouping(PLACE_FROM
_ID_BOLT_ID);
        builder.setBolt(CLIENTID_COUNTER_BOLT_ID, new ClientIDCounter(), 1)
          .setNumTasks(4)

    .shuffleGrouping(XMPP_CLIENTID_SPLITTER_BOLT_ID).shuffleGrouping(EDDYSTO
NE_CLIENTID_SPLITTER_BOLT_ID);

        //Seconda parte  - Configurazione Storm Cluster
        Config conf = new Config();
        conf.setDebug(true);

conf.registerMetricsConsumer(LatencyThroughtputMetricConsumer.class, 2);

        if (args != null && args.length > 0) {
```

```
                conf.setNumWorkers(1);
                StormSubmitter.submitTopologyWithProgressBar(args[0], conf,
builder.createTopology());
          }else {

                LocalCluster cluster = new LocalCluster();
                cluster.submitTopology("test", conf,
builder.createTopology());
                Utils.sleep(30000);
                cluster.killTopology("test");
                cluster.shutdown();
          }
    }

}
```

4.2.2 LatencyThroughputMetricConsumer

La classe che funge da metric consumer e da load balancing come detto anche in precedenza è la classe LatencyThroughtputMetricConsumer, a seguire viene riportato il codice della classe.

```
public void prepare(@SuppressWarnings("rawtypes") Map stormConf, Object
registrationArgument, TopologyContext context,
                IErrorReporter errorReporter) {
          nimbusClient =
NimbusClient.getConfiguredClient(Utils.readDefaultConfig());
        try {
                List<TopologySummary> ts =
nimbusClient.getClient().getClusterInfo().get_topologies();
                for(TopologySummary list : ts){
                      if(list.get_name().equalsIgnoreCase("test")){
                            this.topS = list;
                      }
                }
                this.top =
nimbusClient.getClient().getTopology(topS.get_id());
          } catch (TException e) {
                e.printStackTrace();
          } catch (NotAliveException e) {
                e.printStackTrace();
          }
        init();
        takeBolts();
```

Intanto bisogna dire che un metric consumer deve ereditare l'interfaccia IMetricConsumer da cui si ereditano dei metodi, tra cui il metodo prepare() che è il primo metodo invocato quando si instanzia l'oggetto. All'interno di esso si prendono delle informazioni importanti tra cui un oggetto TopologySummary in cui sono presente tutte le informazioni riguardanti la topologia e un oggetto StormTopology che descrive una topologia in generale, ovviamente tutte e due sono riferite alla topologia del prototipo. Questi oggetti possono essere presi in maniera dinamica tramite l'oggetto NimbusClient che descrive un client che accede a Nimbus tramite

tecniche di reflection. Dall'oggetto TopologySummary si può ottenere anche la lista dei bolt e degli spout associati a quella topologia, tutto in maniera dinamica.

```
public void handleDataPoints(TaskInfo taskInfo, Collection<DataPoint>
dataPoints) {
         double value = - 1;
         if(taskInfo.srcComponentId.equalsIgnoreCase("place_counter") ||

    taskInfo.srcComponentId.equalsIgnoreCase("clientID_counter") ||

    taskInfo.srcComponentId.equalsIgnoreCase("place_from_id")){
              for (DataPoint p : dataPoints) {
                   if(p.name.equalsIgnoreCase("__execute-latency") &&

taskInfo.srcComponentId.equalsIgnoreCase("place_from_id")){

                        value = getValueofLatency(p.value.toString());

                        if(value > this.highLatency){
                             this.latencyInc = true;
                        }else if (value < this.lowLatency){
                             this.latencyDec = true;
                        }
                   }
                   if(p.name.equalsIgnoreCase("PlaceCounter") ||
p.name.equalsIgnoreCase("ClientIDCounter")){
                        if(count == nrTotalWriters-2){
                              count=0;
                              tot = tot +
Integer.parseInt(p.value.toString());
                              if(tot > this.highThroughput){
                                   this.throughputDec = true;

                              }else if (tot < this.lowThroughput){
                                   this.throughputInc = true;
                              }
                              tot=0;
                              TOT=tot;
                         }else{
                              count++;
                              tot = tot +
Integer.parseInt(p.value.toString());
                         }
                   }
              }
         }
         if((latencyDec || latencyInc) && (throughputDec ||
throughputInc)){
              if(latencyDec) decreaseParallelism(taskInfo.srcComponentId);
              else increaseParallelism(taskInfo.srcComponentId);
         } else if(latencyDec || latencyInc ){
              if(percLatency(value) > 0.5){
                   if(latencyDec)
decreaseParallelism(taskInfo.srcComponentId);
                   else increaseParallelism(taskInfo.srcComponentId);
              }
         } else if (throughputDec || throughputInc){
              if(percThrou(TOT) > 0.5){
                   if(throughputDec)
decreaseParallelism(taskInfo.srcComponentId);
```

```
                    else increaseParallelism(taskInfo.srcComponentId);
                }
            }
        }
```

Un altro metodo ereditato da IMetricConsumer è il metodo handleDataPoint a cui vengono passati due argomenti, un oggetto TaskInfo che descrive un po' tutte le caratteristiche del task che ha inviato la metrica tra cui il timestamp in cui è stato inviato, il nome della macchina host e la sua porta, l'ID del task e l'ID del componente dell'entità che ha inviato la metrica, e una collezione di oggetti DataPoint che rappresentano i dati veri e propri inviate al metric consumer. Prima di continuare la spiegazione del metodo ci sono da fare delle premesse: tutti i bolt/spout inviano in maniera automatica delle metriche a ogni metric consumer, e oltre a loro anche il sistema Storm. Queste metriche riguardano proprietà come la latenza, il numero di ack, uso della memoria heap e non heap ecc. Oltre a queste metriche per ogni bolt/spout si possono registrare altre metriche da inviare come spiegato nel capitolo di Apache Storm nella sezione Metric. Nel nostro caso si considerano due tipi di metriche: uno riguardanti le metriche generali che arrivano in maniera automatica di cui prendiamo solo quelle riguardanti la latenza esecutiva; e uno riguardanti le metriche che estendono la classe CountMetric che serviranno per il calcolo del throughput. Come spiegato in precedenza non avrebbe senso analizzare tutte le metriche da tutte le entità nella topologia perché di norma solo poche di esse possono essere in condizioni critiche. Nella nostra topologia i tre bolt che hanno bisogno di essere controllati sono PlaceCounter, ClientIDCounter e PlaceFromID, in quanto dopo una prima fase di test e di analisi sono gli unici che hanno relazioni con Apache Cassandra, l'accesso alla quale può portare alle maggiori criticità dal punto di vista temporale. Quindi nella prima parte del metodo si effettua un filtraggio sulla tipologia di componente che invia la metrica, poi successivamente si analizzano i datapoint selezionando solo quelli che trattano di latenza esecutiva o solo quelli costruiti ad hoc per il throughtput. Nel primo caso si verifica semplicemente se il valore letto dalla metrica supera la soglia superiore o quella inferiore. Nel caso si superi la soglia superiore significa che si deve incrementare il parallelism hint e viceversa se è al di sotto della soglia minore. Per quanto riguarda il throughput prima di avere un valore reale bisogna ricevere il numero totale di tuple elaborate in un lasso di tempo da tutti i thread di una determinato componente, successivamente si può verificare se ha superato uno delle due soglie, nel caso che si sia superata la soglia superiore significa che vengono elaborate più tuple del dovuto quindi si può diminuire il parallelism hint e viceversa se si è al di sotto della soglia minore.

Un caso particolare è quello in cui solo uno dei due valori sia oltre il limite cioè può capitare ad esempio che la latenza esecutiva sia oltre una soglia ma che invece il throughput sia nella media. In questi casi si è pensato di usate un approccio singolare basato su quanto il valore di soglia ha superato il limite. Se lo ha superato più del 50% allora si può ribilanciare la topologia senza che si aspetti che anche l'altro valore non sia nella media. Ovviamente il problema non si pone se tutte e due i valori sono fuori soglia. Il metodo infine dopo aver fatto tutto i vari controlli decide se aumentare o diminuire il parallelism hint o stare semplicemente in attesa.

```
private void increaseParallelism(String name) {
          RebalanceOptions opt = new RebalanceOptions();
          opt.set_wait_secs(0);

          if(this.bolts.get(name).get_common().get_parallelism_hint() <
this.bolts_ex.get(name)){
                int value =
this.bolts.get(name).get_common().get_parallelism_hint()+

    calcolateKExecutors(this.bolts.get(name).get_common().get_parallelism_hi
nt());
                if(value > this.bolts_ex.get(name)) value =
this.bolts_ex.get(name);
                opt = setNumExecutors(opt, value, name);
          }else{

                int value = this.nrTotalWriters+calcolateKWorkers();
                if(value > UtilTopologyStorm.MAXWORKERS) value =
UtilTopologyStorm.MAXWORKERS;
                opt = setNumWorkers(opt, value);
          }
          rebalanceTopology(opt);
          resetBools();
    }
```

Il metodo increaseParallelism (come il suo duale decreaseParallelism) non è un metodo ereditato dall'interfaccia IMetricConsumer, ma è un metodo interno chiamato per ribilanciare il carico quando serve. Il metodo per prima cosa valuta se è possibile aumentare il numero di executor così da evitare casi di carico parziali e quindi di aumentare il numero di worker inutilmente. Successivamente per decidere di quanto aumentare il parallelism hint si basa sul suo valore attuale e cerca di aumentarlo in maniera proporzionale, senza ovviamente superare il massimo possibile o andare a zero. Successivamente se non è possibile modificare il numero di executor, prova ad aumentare il numero di worker ed effettua un ragionamento duale a quanto fatto per gli executor. Durante tutto il metodo si va a completare con le informazioni giuste un oggetto chiamato RebalanceOptions che semplicemente contiene tutte

le modifiche che saranno tramandate a Storm. Come ultima operazione chiama il metodo rebalanceTopology passando come argomento l'oggetto RebalanceOptions.

```
public void rebalanceTopology(RebalanceOptions options){
          try {
                nimbusClient.getClient().rebalance("test", options);
          } catch (NotAliveException e) {
                e.printStackTrace();
          } catch (InvalidTopologyException e) {
                e.printStackTrace();
          } catch (TException e) {
                rebalanceTopology(options);
                e.printStackTrace();
          }
    }
```

Il metodo rebalanceTopology chiama semplicemente sul cliente che accede a Nimbus il metodo rebalance, che ha bisogno del nome della topologia come primo argomento, e di un oggetto RebalanceOptions che contiene tutte le modifiche da apportare alla topologia. Ovviamente l'operazione può andare male per diversi motivi raccolti in delle eccezioni catturate tramite try-catch, la più interessante è TException che può nascere nel momento in cui la topologia sta già facendo delle modifiche e non riesce a rispondere in tempo alla richiesta di ribilanciamento; in questo caso si richiamo semplicemente il metodo per riprovare l'operazione.

```
private void init(){
        this.tot_workers = UtilTopologyStorm.NR_WORKERS_INI;
        this.nrTotalWriters = UtilTopologyStorm.NR_TOT_WRITERS;
        this.highLatency = UtilTopologyStorm.HIGHLATENCY;
        this.lowLatency = UtilTopologyStorm.LOWLATENCY;
        this.lowThroughput = UtilTopologyStorm.LOWTHROUGHPUT;
        this.highThroughput = UtilTopologyStorm.HIGHTHROUGHPUT;
        resetBools();
        this.count = 0; this.tot=0;
}
```

Nel metodo init() vengono semplicemente presi i valori si soglia e altri valori come le cardinalità di partenza delle risorse utilizzate, e per fare ciò su usa una classe Utility chiamata UtilTopologyStorm. Oltre a ciò vengono resettati dei valori che verranno utilizzati successivamente per il throughput e per l'algoritmo di bilanciamento.

```
private int calcolateKWorkers(){
          int result=0;
          result = this.tot_workers/3;
          if(result < 1) result = 1;
          return result;
```

```
    }
    private int calcolateKExecutors(int value){
          int result=0;
          result = value/3;
          if(result < 1) result = 1;
          return result;
    }
```

calcolateKWorkers e calcolateKExecutors sono i due metodi tramite i quali calcoliamo il valore 'k' con cui aumentare/diminuire la cardinalità dei worker e degli executor rispettivamente. Per fare ciò si considera in partenza il valore attuale delle risorse e poi si calcola un numero intero che rappresenta circa un terzo delle risorse attuali. Di norma quando si tratta di bilanciamento di carico le risorse che devono aumentare oscillano tra il 10% e il 50% quindi la scelta del 33.3% sembra una scelta sensata in tale senso. Con delle versioni più avanzate di tale prototipo si potrebbe pensare di fare dei ragionamenti più complessi su come calcolare 'k', basati anche su una prima fase di test nella quale si cerca di capire mediamente di quanto si dovrebbe ribilanciare il sistema, seguita anche da un'analisi matematica in cui si cerca di capire quale funzione matematica/probabilistica descrive al meglio il comportamento di questi cambiamenti.

Per quanto riguarda i bolt che utilizzano i CountMetric devono seguire la seguente procedura:

- `transient CountMetric _countMetric`: questa è la variabile dichiarata all'interno del bolt. Si noti che la variabile è dichiarata come transient. Questo è necessario perché nessuno dei Metric è serializzabile e tutte le variabili non-transient nei bolt/spout devo essere serializzabili;
- ```
  _countMetric = new CountMetric();
  context.registerMetric("ClientIDCounter", _countMetric, 60);
  ```
  All'interno del metodo prepare si va a instanziare la classe CountMetric, e attraverso il metodo registerMetric della TopologyContext si va a registrare il Metric. Ha bisogno di tre argomenti: il primo rappresenta il nome dei dati DataPoint che invierà, il secondo è l'istanza del CountMetric, e il terzo campo rappresenta l'intervallo di tempo in cui si deve inviare la metrica. È importante notare che la scelta del valore dell'intervallo di tempo con cui contattare il metric consumer è molto importante, perché scegliendo un valore molto basso significherebbe che si andrebbe a caricare in maniera inutile il metric consumer, viceversa se l'intervallo è troppo alto potrebbe
  ```

succedere che il metric consumer si accorge in maniera tardiva che bisogna ribilanciare la topologia. Anche in questi casi è importante la fase di test per fare delle scelte ottimali;

- `_countMetric.incr();` Chiamando il metodo incr si va semplicemente a incrementare il valore della variabile. Quest'operazione si effettua alla fine del metodo execute del bolt, e successivamente allo scadere del periodo di tempo si procederà all'invio del valore del CountMetric al metric consumer che riceverà quindi il numero totale di tuple che sono state elaborate da quel bolt.

Capitolo 5 – Risultati Sperimentali

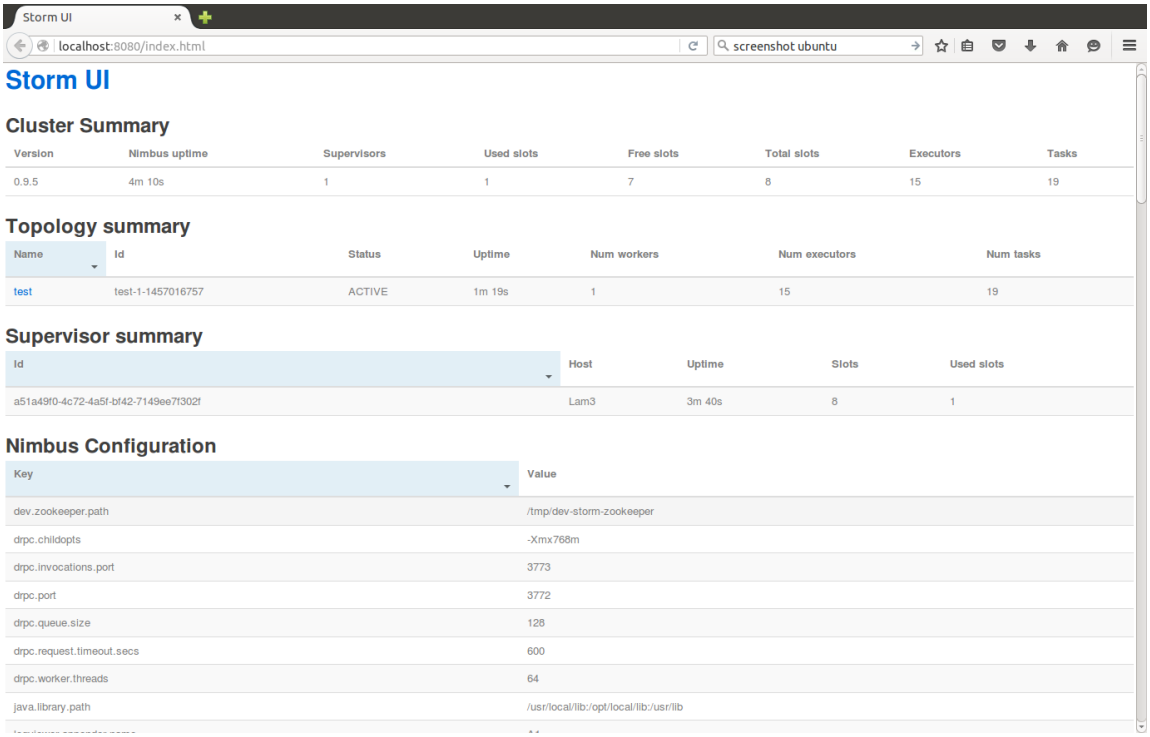
I test sono stati portati a termine su un computer con le seguenti caratteristiche:

- Processore Intel Core Duo E8400 con frequenza 3.00GHz;
- 128MB di cache di primo livello;
- 64MB di cache di secondo livello;
- 4GB di RAM.

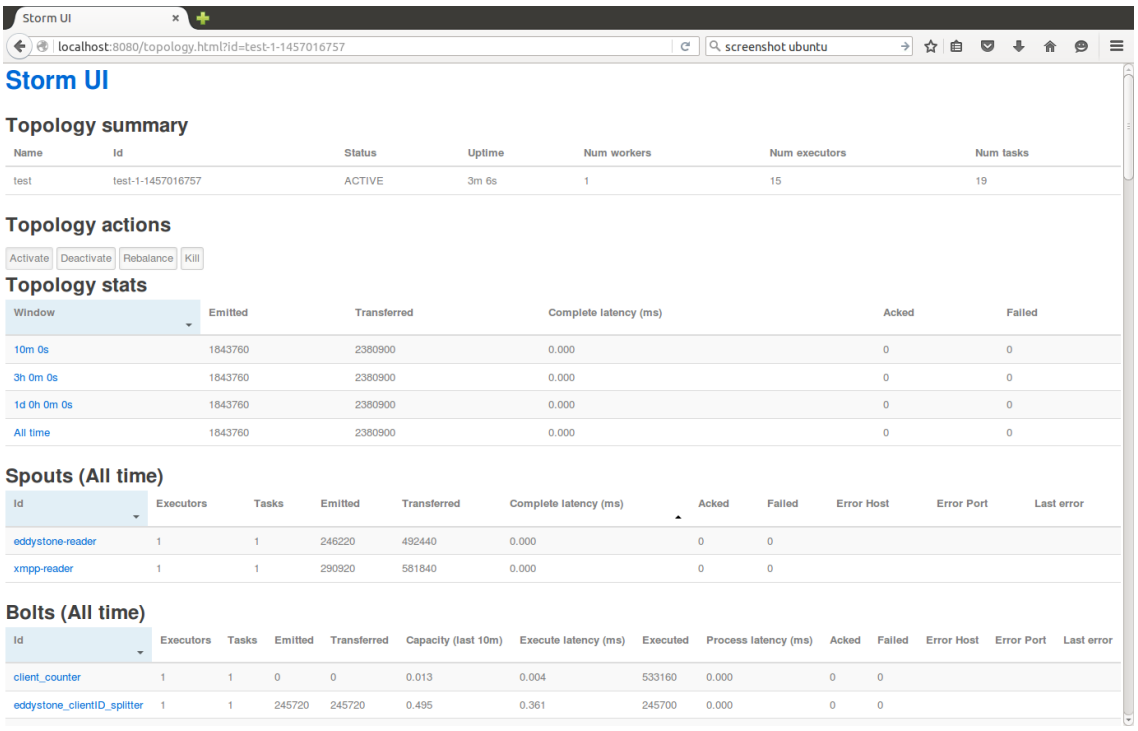
La fase di sperimentazione prevede undici batterie da dieci prove (tranne gli ultimi due test che sono formati da cinque prove) che cercano di analizzare quasi tutti i casi tipici che possono accadere. Le prove sono identiche fra di loro e hanno una durata di dieci minuti (tranne nel caso del decimo test). Dopo vengono presi il parallelism hint finale, il throughput totale e la latenza esecutiva media di tutti i componenti.

Per prima cosa si controlla se il demone di Apache Cassandra è in esecuzione. In seguito per far partire Apache Storm bisogna far eseguire nell'ordine ZooKeeper, Nimbus, Supervisor e infine UI tramite il quale possiamo avere delle info su tutto quello che Storm sta eseguendo. Successivamente tramite Maven e il comando 'mvn clean install assembly:assembly' si crea il package adatto per montarlo su Storm. Dopo grazie al comando 'jar' di Storm e avendo passato gli argomento in maniera corretta si può far eseguire finalmente la topologia. Per

controllare che tutto sia andato in maniera corretta basta andare su un Browser Web all'indirizzo localhost:8080 (o da remoto <indirizzo_macchina>:8080) è il risultato dovrebbe essere il seguente:



In seguito se si clicca sulla topologia 'test' si possono avere tutte le info utili sulla topologia in esecuzione, con un layout come il seguente:



Tramite il quale possiamo avere dei valori statistici o compiere anche delle operazioni.

Controllando i file di log (in modalità debug possiamo vedere quello che è il processo di elaborazione dei dati. EddystoneReader riceve e poi invia a PlaceSplitterEddystone e ClientSplitterEddystone tuple come le seguenti:

```
[id_place:123456,id_client:1111]
[id_place:345678,id_client:1111]
[id_place:123456,id_client:3333]
[id_place:345678,id_client:3333]
[id_place:234567,id_client:2222]
```

Invece XMPPReader riceve e inoltra a PlaceSplitterXMPP e ClientSplitterXMPP tuple come le seguenti:

```
divisione1@web.com/1111
divisione2@web.com/2222
divisione3@web.com/2222
divisione1@web.com/2222
divisione1@web.com/3333
```

ClientSplitterEddystone prenderà solo il valore dell'id_client, invece PlaceSplitterEddystone prenderà il valore di id_place, farà una query di lettura su Cassandra e successivamente avrà una stringa univoca per la locazione. Le tuple che produrranno per ClientIDCounter e PlaceCounter sono uguale nel formato a quelle prodotte dai bolt PlaceSplitterXMPP e ClientSplitterXMPP. Per quanto riguarda ClientIDCounter riceverà le seguenti tuple:

```
1111      1111
3333      3333
2222      1111
2222      2222
2222      3333
```

Invece PlaceCounter riceverà le seguenti tuple:

```
divisione1      divisione3
divisione1      divisione3
divisione2      divisione1
divisione2      divisione3
divisione1      divisione1
```

In qualsiasi momento possiamo fare delle query su Cassandra e possiamo vedere che i dati di conteggio sono memorizzati correttamente. I dati ovviamente sono memorizzati ogni sessanta secondi dai bolt ClientIDCounter e PlaceCounter e tramite cqlsh si possono vedere:

```
cqlsh:tesi_ardit> select * from client;

 name | timestamp                       | count
------+---------------------------------+-------
 1111 | 2016-03-03 14:52:44+0000 | 37033
 1111 | 2016-03-03 14:53:46+0000 | 68748
 1111 | 2016-03-03 14:54:46+0000 | 75987
 2222 | 2016-03-03 14:52:44+0000 | 36920
 2222 | 2016-03-03 14:53:46+0000 | 68783
 2222 | 2016-03-03 14:54:46+0000 | 75543
 3333 | 2016-03-03 14:52:44+0000 | 37368
 3333 | 2016-03-03 14:53:46+0000 | 68110
 3333 | 2016-03-03 14:54:46+0000 | 75808
```

```
cqlsh:tesi_ardit> select * from place;

 name       | timestamp                       | count
------------+---------------------------------+-------
 divisione3 | 2016-03-03 14:52:44+0000 | 35724
 divisione3 | 2016-03-03 14:53:46+0000 | 68109
 divisione3 | 2016-03-03 14:54:46+0000 | 76431
 divisione2 | 2016-03-03 14:52:44+0000 | 35559
 divisione2 | 2016-03-03 14:53:46+0000 | 68522
 divisione2 | 2016-03-03 14:54:46+0000 | 76760
 divisione1 | 2016-03-03 14:52:44+0000 | 35205
 divisione1 | 2016-03-03 14:53:46+0000 | 68893
 divisione1 | 2016-03-03 14:54:46+0000 | 76403
```

Figura 5.1 – Le tabelle client e place in Cassandra

5.1 Test 1

Il primo test si incentra sul capire quali sono le prestazioni del sistema avendo attivato la modalità debug. Si ricorda che nella modalità debug ogni componente scrive nei file di log qualsiasi operazione effettuata, questo porta a un rallentamento delle prestazioni rispetto al caso senza debug. L'obiettivo di questo test è capire di quanto è la differenza. Il throughput medio deve essere fra 35mila e 45mila al minuto (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms. Il numero di worker nell'intera prova rimane lo stesso cioè uno, invece il numero di executor è inizialmente 17 (21 thread) fino ad arrivare alla prova finale di 33 (35 thread).

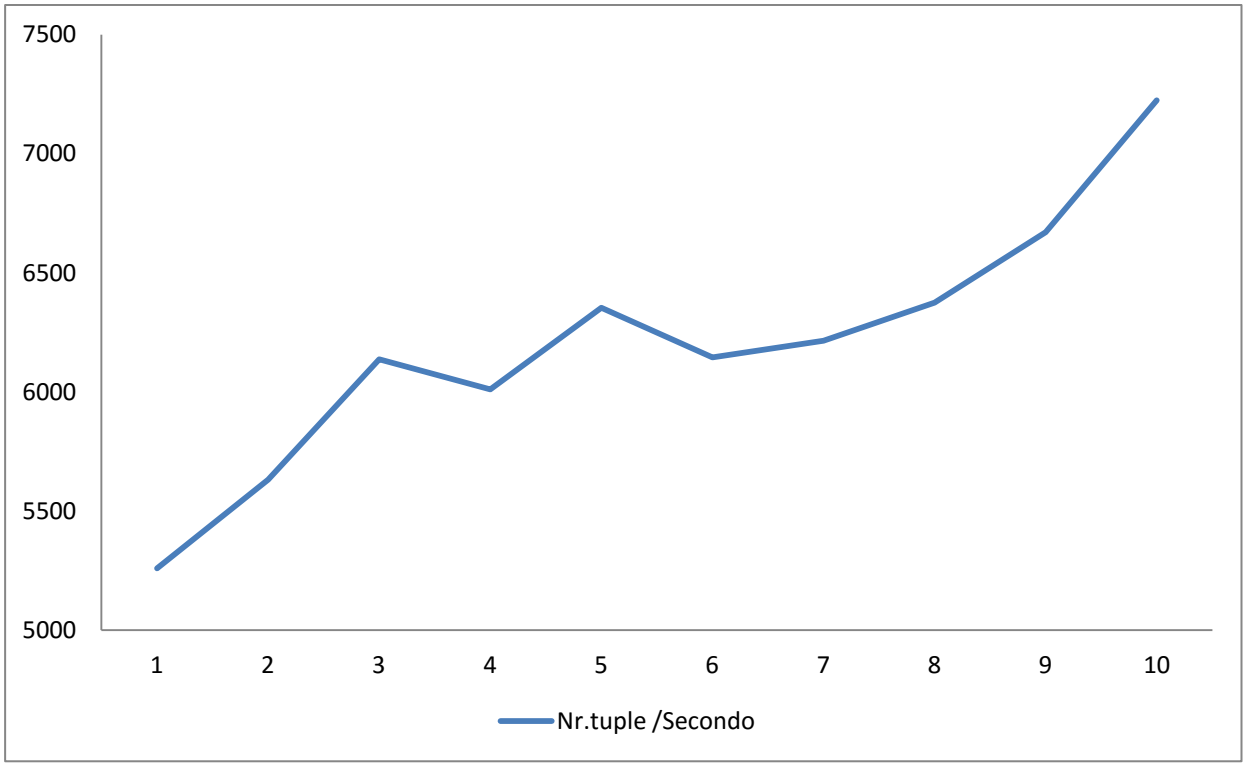


Grafico 5.1 – Throughput Test 1

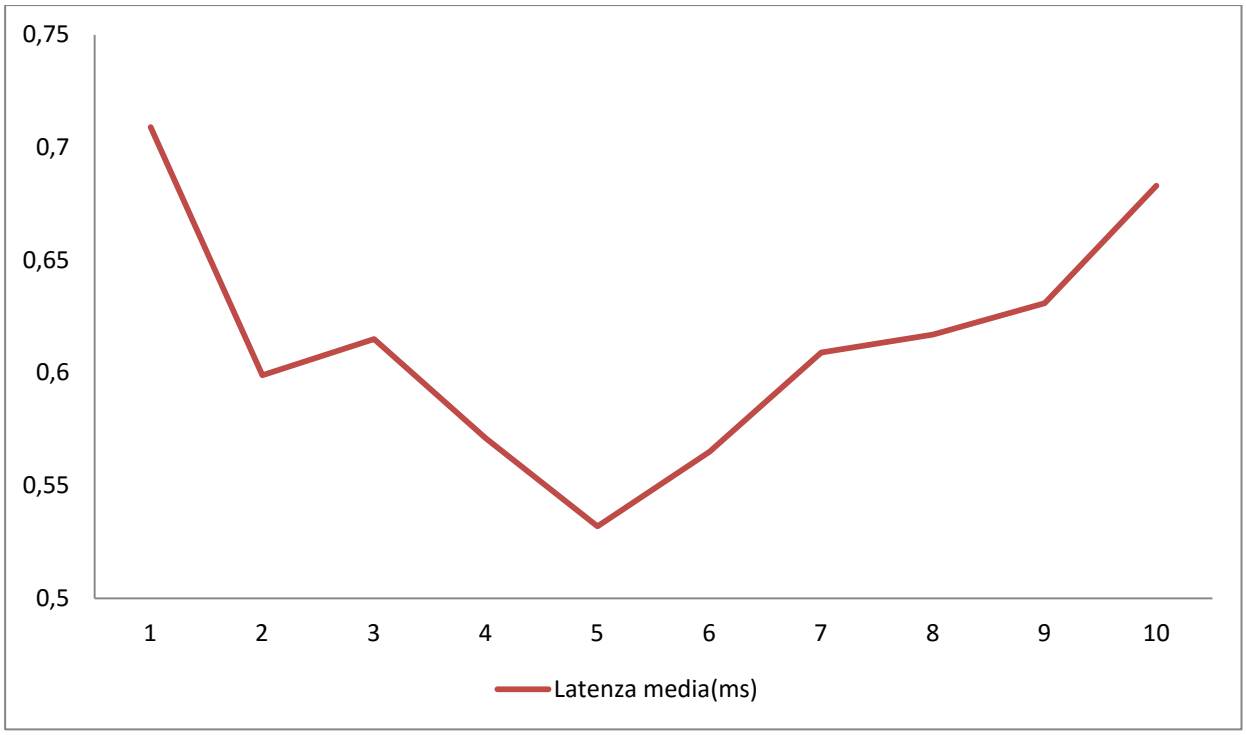


Grafico 5.2 – Latenza media Test 1

Alla fine della prima batteria di test si può notare che:

- Il throughput aumenta in maniera quasi lineare all'aumentare delle risorse a disposizione (che in questa batteria era rappresentato dall'aumento del numero di executor); si nota che tra la prova quattro e la prova sette si ha un comportamento anomalo: ciò si potrebbe spiegare con il fatto che le risorse in quei casi non aumentano di tanto, aggiungendo anche fattori aleatori (che dipendono dalla natura stessa delle tuple) e il tempo necessario al sistema per trovare l'assetto migliore grazie al bilanciamento di carico, che nei casi detti è maggiore rispetto che negli altri casi.
- La latenza media invece non segue lo stesso andamento del throughput ma parte da un livello alto, subisce una diminuzione per poi tornare in sostanza a livello iniziale. Questo può sembrare un comportamento strano poiché dopo aver aumentato le risorse ci si aspetta che tutte e due i valori (throughput e latenza esecutiva) migliorino ma se si analizzano meglio i dati a disposizione si possono notare le cause le quali possono essere:
    - Componente particolarmente lento: se un certo componente ha una latenza media di 'x' ms per eseguire una certa operazione e ha raggiunto il suo limite anche aggiungendo componenti non si riesce ad aumentare le prestazioni di

quel componente, però nel sistema in generale ci sono più tuple rispetto al caso precedente questo porta come conseguenza quella che le tuple in coda nelle diverse code sono di più quindi come risultato finale si ha che il numero di tuple totali elaborate aumenta (ed è quello che ci si aspetta) invece la latenza aumenta anche perché le tuple ci mettono di più a essere elaborate completamente;

  - Limite di componenti esterni: nel nostro caso potrebbe essere rappresentato dal numero di connessioni massime che si possono avere con Apache Cassandra, per superare questo limite si potrebbe aumentare la cardinalità dei componenti di Cassandra in modo da dividere meglio il carico di chiamate, in questo caso però se questo non viene fatto si posso notare le anomalie visto nei grafici precedenti;
  - O semplicemente avendo più dati in ingresso mediamente i bolt sono più occupati rispetto che in precedenza quindi si allunga anche la latenza; ovviamente la latenza deve sempre rimanere nella media: caso tipico in cui questo avviene è il caso in il throughput non è rispettato quindi aumentando le risorse si riesce a elaborare più tuple al secondo a discapito delle prestazioni della latenza.

- Una topologia in Storm non parte in maniera istantanea ma ci mette circa 60-75 secondi per arrivare a regime, in questo minuto si potrebbero accumulare delle tuple nelle code dei bolt, quindi i primi minuti di test si può notare una latenza esecutiva particolarmente alta, latenza che a regime poi si abbassa e raggiunge i valori medi aspettati;
- Si nota che il valore di soglia del throughput dell'ultimo minuto di norma è più alto del throughput medio, questo si nota in quei casi in cui è stato praticato il ribilanciamento (che in questa batteria è capitato nove volte su dieci);
- Può succedere che anche cambiando le risorse (di poco) non si apprezzino miglioramenti significativi, la causa di ciò risiede probabilmente nel fatto che la criticità per essere risolta ha bisogno di ancora più risorse (ciò spiega in parte la parte centrale del grafico del throughput);

5.2 Test 2

Nel secondo test si vanno a controllare quello che è il comportamento del sistema in una situazione normale. Si incentra sul capire quali sono le prestazioni del sistema. Si può anche

vedere di quanto è la differenza rispetto alla modalità debug avendo lo stesso input e le stesse risorse. Il throughput medio deve essere fra 190mila e 250mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

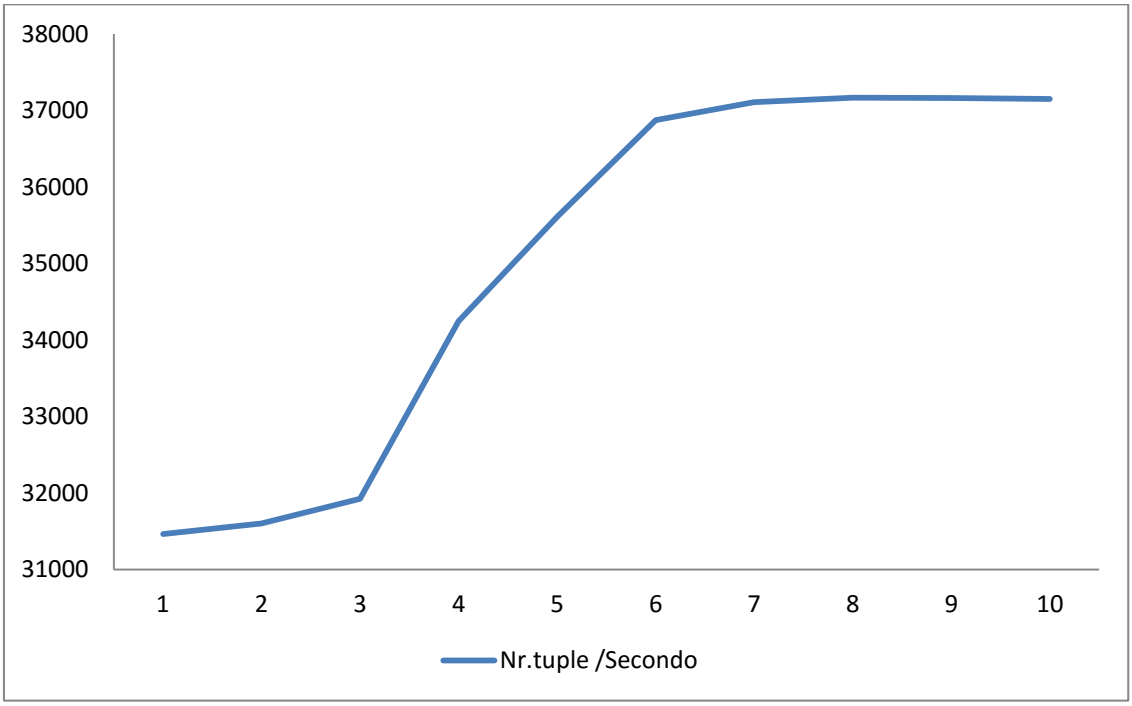


Grafico 5.3 – Throughput media Test 2

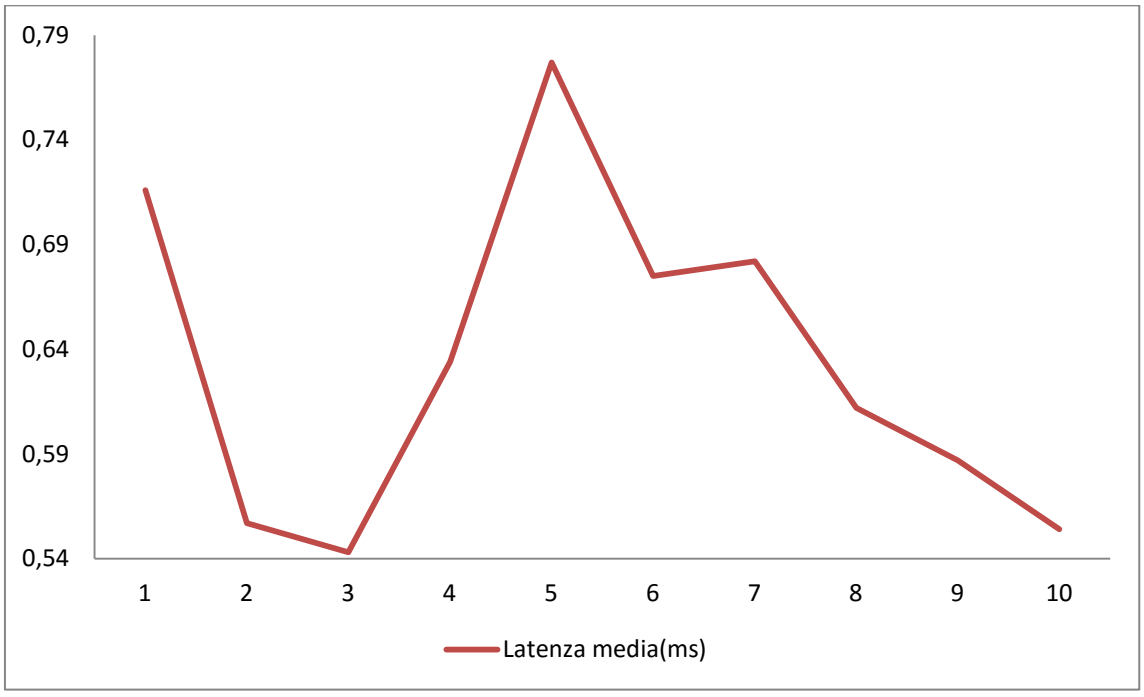


Grafico 5.4 – Latenza media Test 2

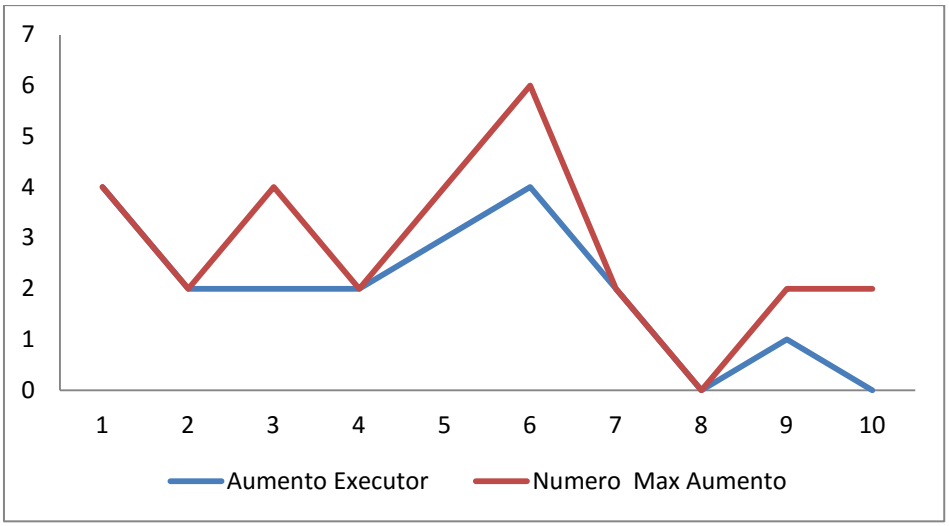


Grafico 5.5 – Variazione executor Test 2

Dai risultati della seconda batteria si può notare che:

- Il throughput aumenta in maniera proporzionale all'aumentare delle risorse: questo fino alla prova numero sei, dalla quale in poi si vede che l'andamento dei risultati rimane quasi sullo stesso livello. Questo comportamento anomalo ha una spiegazione molto semplice: il sistema riesce a consumare tutto l'input senza che ci siano molti elementi nelle code di attesa. A conferma di ciò si può notare dalla sesta prova in poi anche un crollo nei valori della latenza in quanto una tupla ci mette di meno a essere elaborata completamente. Un ulteriore prova di ciò si può avere dal grafico degli executor che inizialmente vengono aumentati praticamente al massimo fino ad arrivare nelle ultime prove in cui l'aumento è minimo;
- Si nota un picco nella latenza nella quinta prova, questa oltre ad essere dovuto a elementi aleatori proprie delle tuple potrebbe essere spiegato con il fatto che nella quinta prova vengono elaborate tantissime tuple con relativamente poche risorse rispetto alle prove che ne seguono;
- Come detto in precedenza il numero degli executor all'inizio raggiunge quasi sempre il massimo perché non è rispettato il throughput, fino ad arrivare nelle ultime prove in cui la variazione tra gli executor iniziali e finali è minima in quanto la latenza e il throughput sono nella media;
- Rispetto alla modalità debug si possono vedere che si riesce a elaborare circa cinque/sei volte più tuple.

5.3 Test 3

Nel terzo test si aumenta l'input a uno solo dei due spout in ingresso (in questo caso XMPPReader) per vedere come variano le prestazioni se una sua parte è più carica rispetto al resto della topologia. In seguito si carica in più l'altro spout per vedere se cambia qualcosa in dipendenza di quale parte della topologia è più carica. Il throughput medio deve essere fra 190mila e 250mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

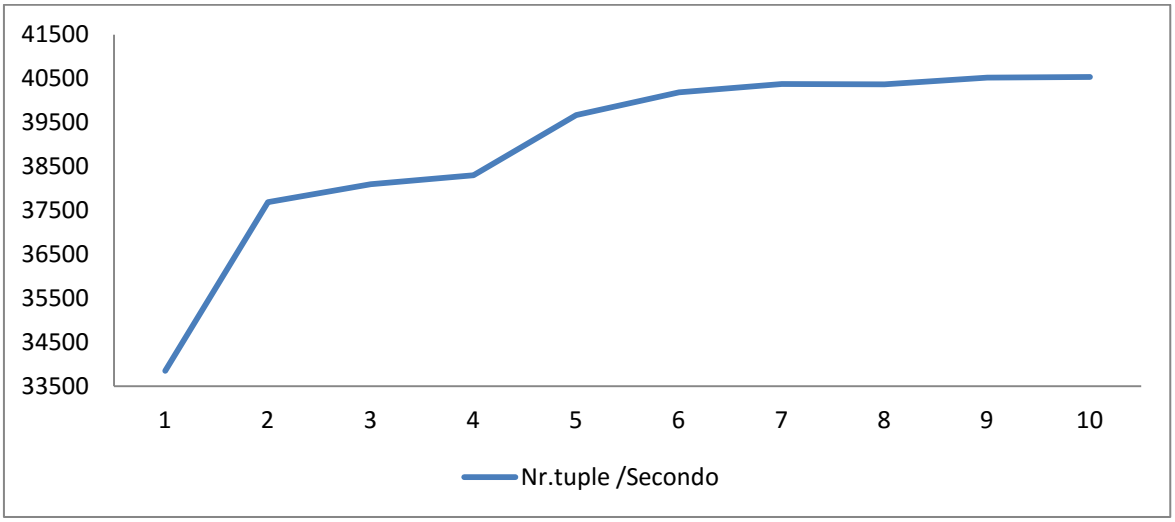


Grafico 5.6 - Throughput media Test 3

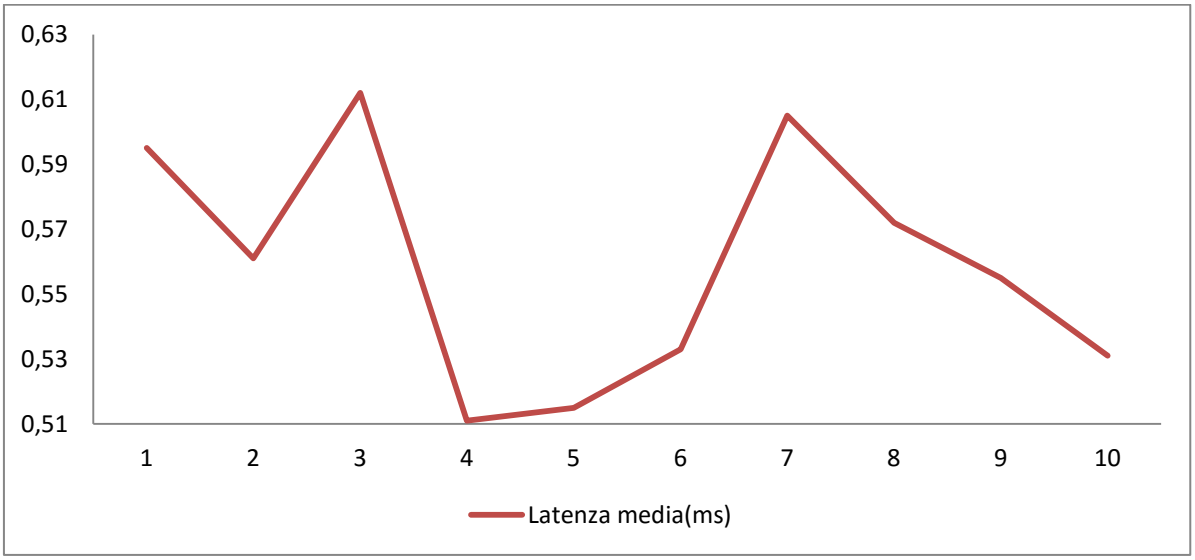


Grafico 5.7 – Latenza media Test 3

Dai risultati del terzo possiamo si può notare che:

- L'andamento dello throughput è simile al test due come ci si poteva aspettare, anche se il valore di sogli raggiunto è più alto. Questo significa che la parte della topologia che elabora XMPP reagisce bene anche se si aumenta l'ingresso;
- L'andamento della variazione del numero di executor è del tutto similare al test due;
- L'andamento invece della latenza è simile al test due ma con importanti varianti: ci sono due picchi, il primo al terzo tentativo che si può spiegare con il fatto che semplicemente si raggiunge quel livello di latenza prima rispetto al test due, ed è presente un secondo picco al settimo tentativo; l'interpretazione di questo picco si potrebbe spiegare come fenomeno similare a quello del picco terzo (o del picco al quinto tentativo del test due).

5.4 Test 4

Nel quarto test si aumenta l'input a uno solo dei due spout in ingresso come nel test tre solo che in questo caso lo spout sarà EddystoneReader. Grazie a questo test quindi si può fare un confronto fra i due spout per vedere se ci sono differenze e soprattutto per vedere se le due parti della topologia hanno discrepanze significative fra di loro. Il throughput medio deve essere fra 190mila e 250mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

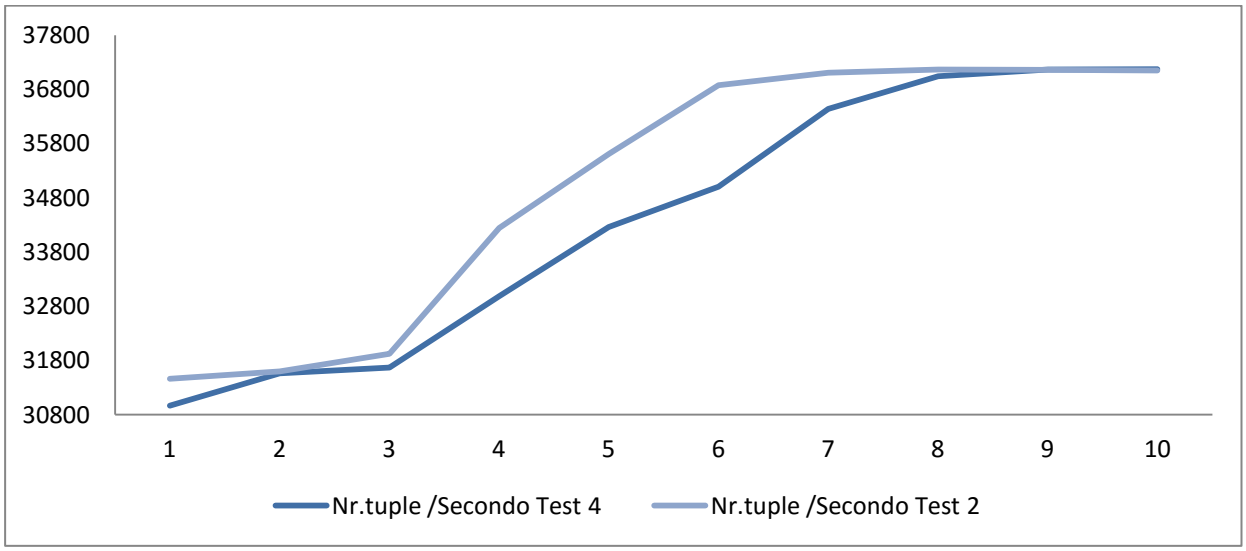


Grafico 5.8 – Confronto throughput Test 4 e Test 2

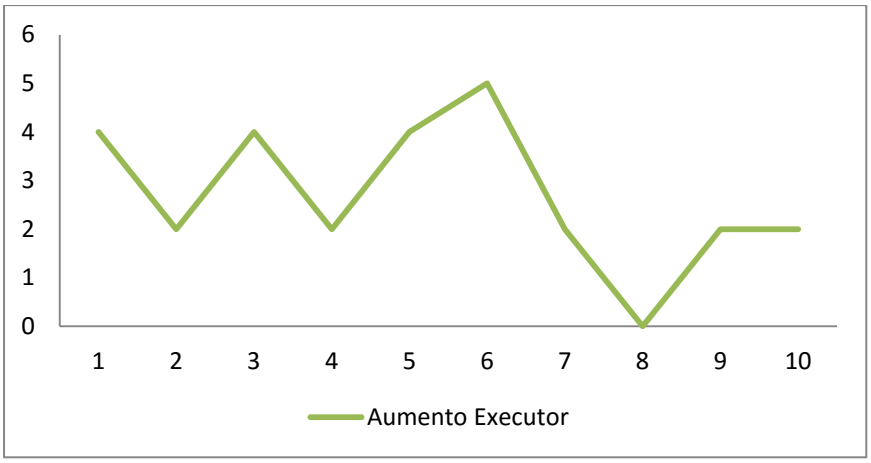


Grafico 5.9 – variazione Executor nel Test 4

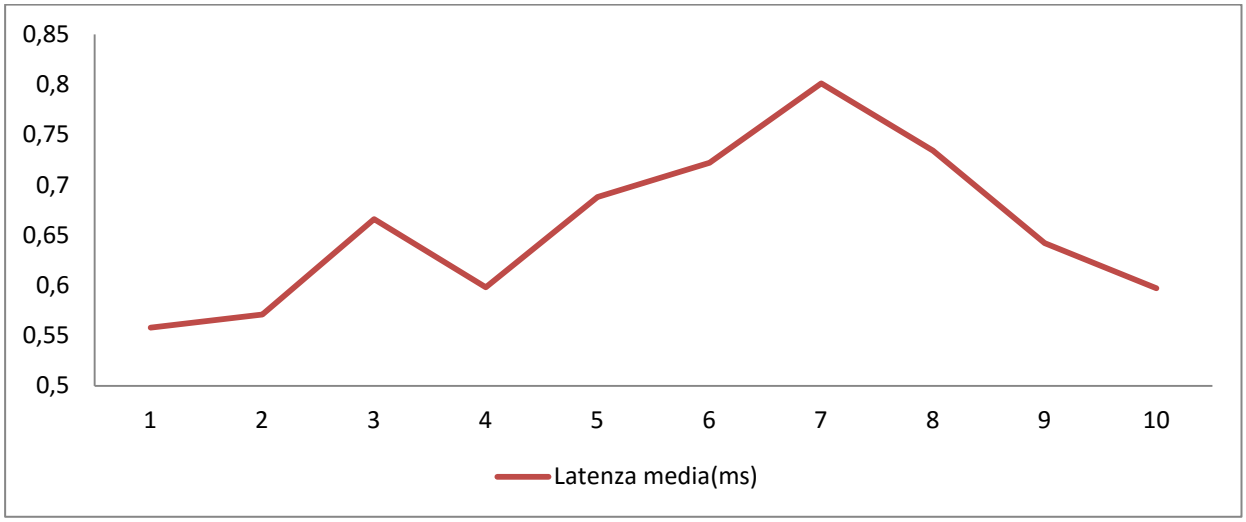


Grafico 5.10 – Latenza media Test 4

Dai dati del test quattro possiamo notare:

- A differenza del test tre non c'è un aumento significativo nelle performance, anzi se si mette a confronto con le prestazioni del test due si può notare che l'andamento delle performance è praticamente uguale; questo ci fa capire che anche aumentando l'input la parte di topologia riguardante Eddystone non riesce a migliorare le proprie statistiche in maniera positiva. Il collo di bottiglia è rappresentato come detto più volte dalla classe PlaceFromID che a ogni esecuzione accede in lettura ad Apache Cassandra e non riesce a migliorare le proprie prestazioni. Ci possono essere due soluzioni a questo problema:
    - Come detto anche in precedenza si potrebbe aumentare la cardinalità delle risorse Cassandra, che non ha problemi di scalabilità;

- Si potrebbe utilizzare un database ottimizzato per le letture come ad esempio LDAP.

- Siccome ci sono sempre troppe tuple in attesa, come si può notare dal grafico 5.9 il numero di executor finale è sempre maggiore di quelli iniziali;
- Per quanto riguarda la latenza mediamente è più alta rispetto agli altri test è la spiegazione risiede sempre nelle code di attesa che sono sempre piene. L'andamento è molto simile al grafico della latenza del test numero due, i motivi di ciò sono stati spiegati precedentemente.

5.5 Test 5

Nel quarto test si aumenta l'input a tutti e due gli spout in ingresso per vedere come reagisce la topologia soprattutto in confronto al terzo e al quarto test. Il throughput medio deve essere fra 190mila e 250mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

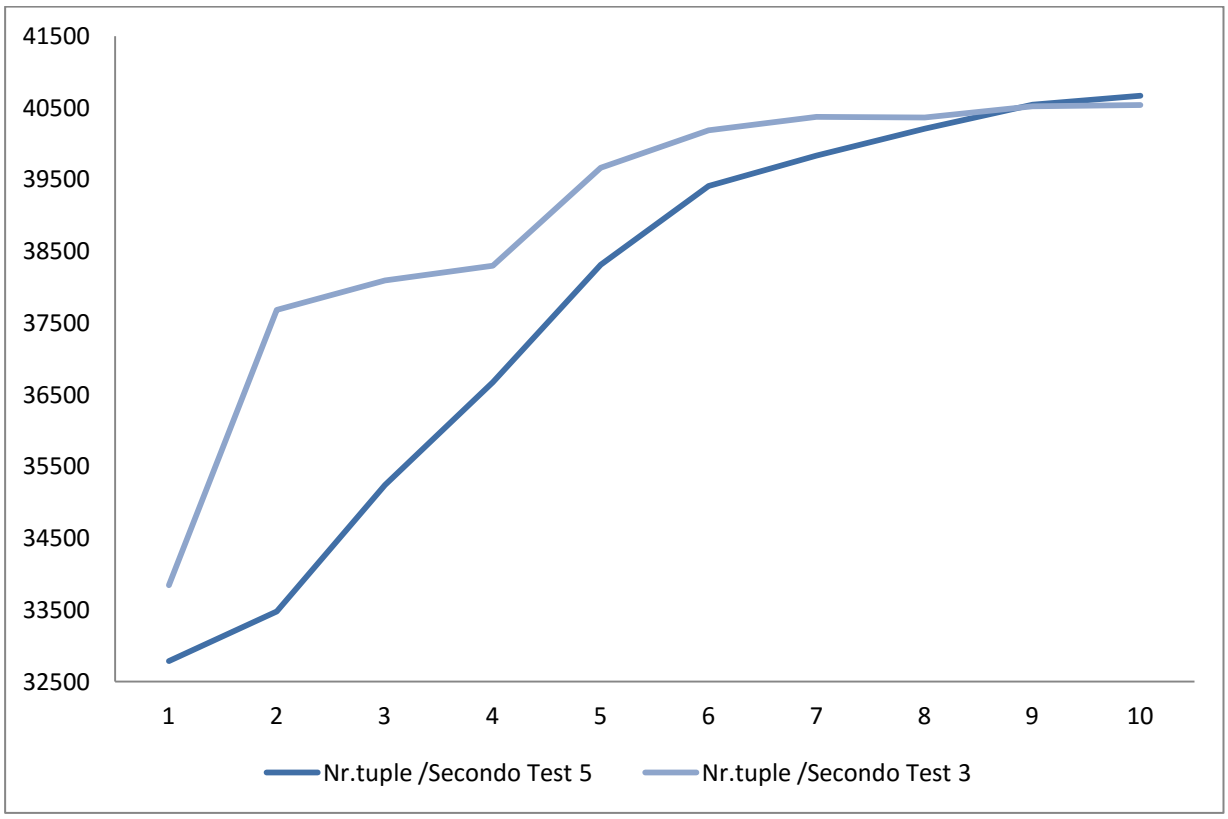


Grafico 5.11 – Confronto throughput Test 5 e Test 3

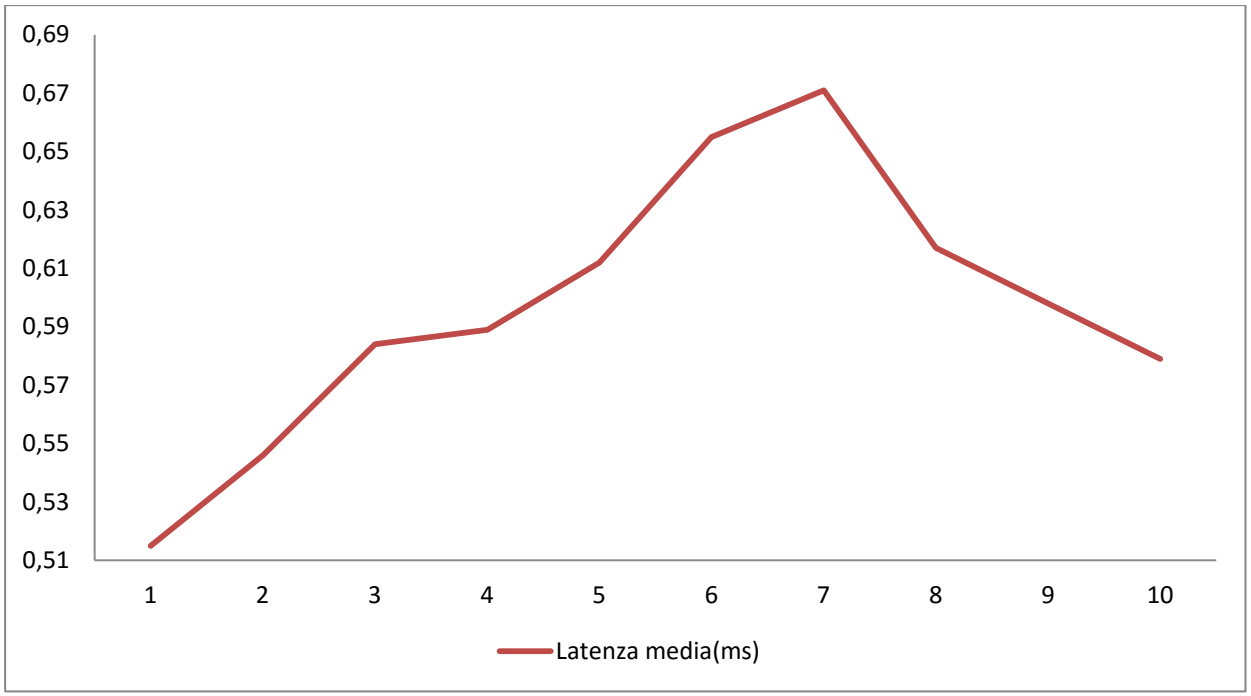


Grafico 5.12 – Latenza media Test 5

Dai risultati del quinto test si può notare:

- Il throughput segue lo stesso andamento del terzo test, come si poteva prevedere in quanto aumentando l'input sullo spout XMPPReader abbiamo un aumento del throughput cosa che non succede invece aumentando l'input allo spout EddystoneReader;
- Per quanto riguarda l'andamento della latenza mediamente è più alta del terzo test in quanto le code sono più piene per i motivi esposti per il quarto test;
- Stesso discorso per la differenza del numero di executor fra l'inizio e la fine del test.

5.6 Test 6

Nel sesto test si aumenta il numero di worker, passando da uno a due. Questo cambiamento è particolarmente significativo in quanto quando si aumenta il numero di worker aumentano in maniera indiretta anche il numero di executor e dei thread. È importante notare quanto cambiano le prestazioni, soprattutto sapendo che l'algoritmo di bilanciamento prova prima ad aumentare il numero di executor e poi il numero di worker, quindi è importante vedere la differenza fra queste due modifiche. Il throughput medio deve essere fra 250mila e 350mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

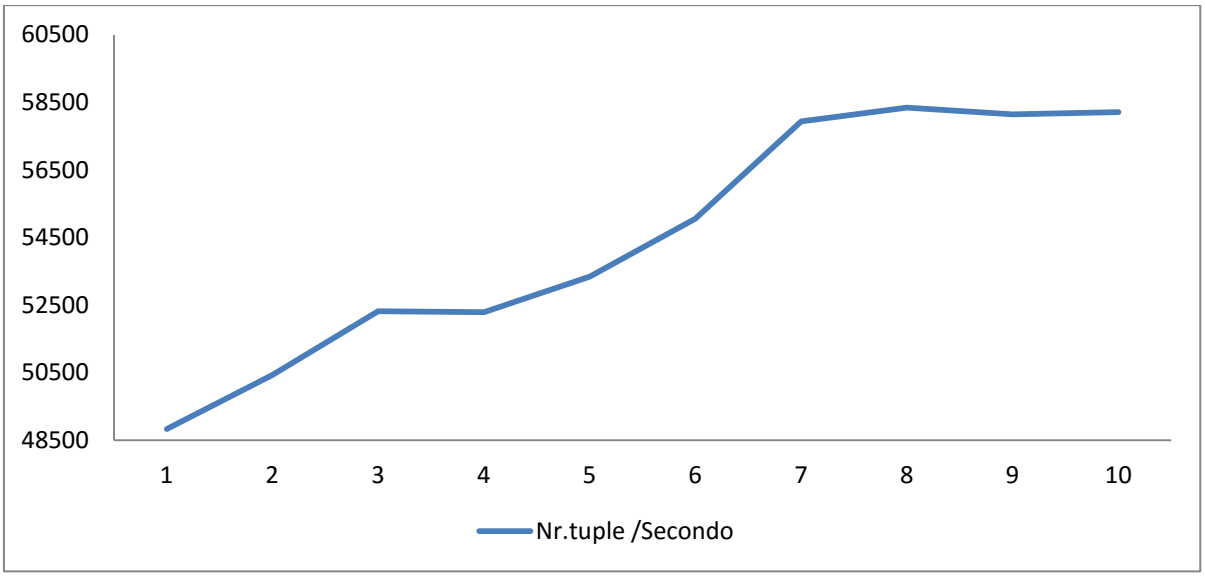


Grafico 5.13 - Throughput media Test 6

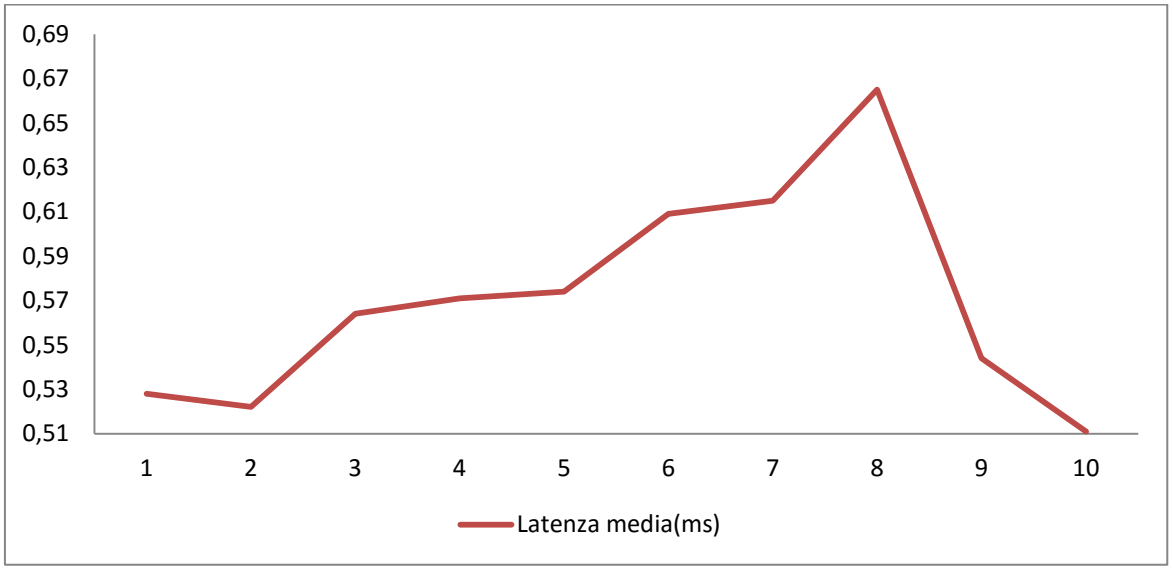


Grafico 5.14 – Latenza media Test 6

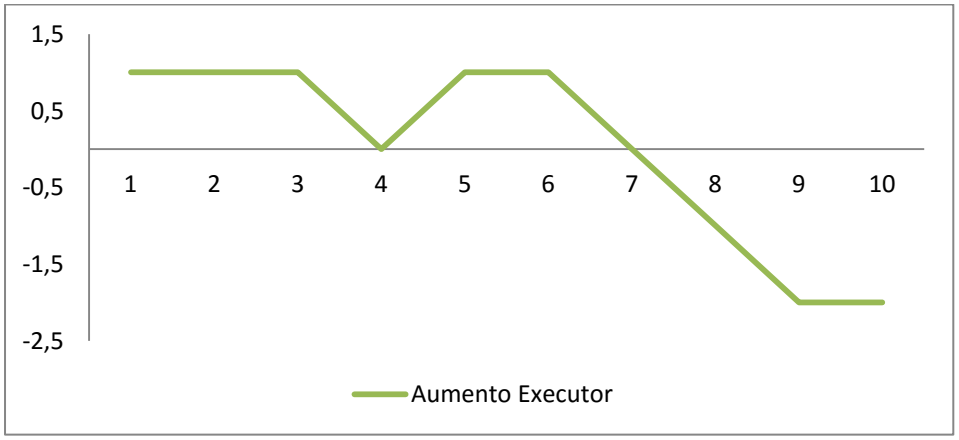


Grafico 5.15 – variazione Executor nel Test 6

Dai risultati del sesto test si può notare che:

- Il throughput cresce fino alla settima prova per poi assestarsi a un valore di soglia avendo raggiunto di limite. Dai grafici successivi si può notare che non si è raggiunto il limite di tuple in ingresso ma semplicemente si è raggiunto la soglia superiore del throughput;
- La latenza ha un andamento classico cresce giacché elabora più tuple al secondo per poi abbassarsi quando le tuple elaborate sono le stesse però le risorse sono di più;
- Il numero di executor all'inizio aumenta, poi dalla settima prova in poi comincia a diminuire. Questo perché probabilmente si è superato la soglia superiore dello throughput quindi il sistema per bilanciarsi diminuisce il numero di executor;
- Raddoppiando il numero di worker (passando da uno a due worker in questo caso specifico) non si ha un raddoppio delle performance, ma più un aumento proporzionale;
- Il numero di tuple elaborate aumenta di circa 1.5 volte rispetto al caso con un solo worker.

5.7 Test 7

Nel settimo test si aumenta l'input a uno solo dei due spout in ingresso (in questo caso XMPPReader) in maniera del tutto similare al terzo test. Grazie a questo test si può vedere se il comportamento della topologia sarà lo stesso come nel caso del terzo test o avrà un comportamento diverso. Il throughput medio deve essere fra 300mila e 400mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

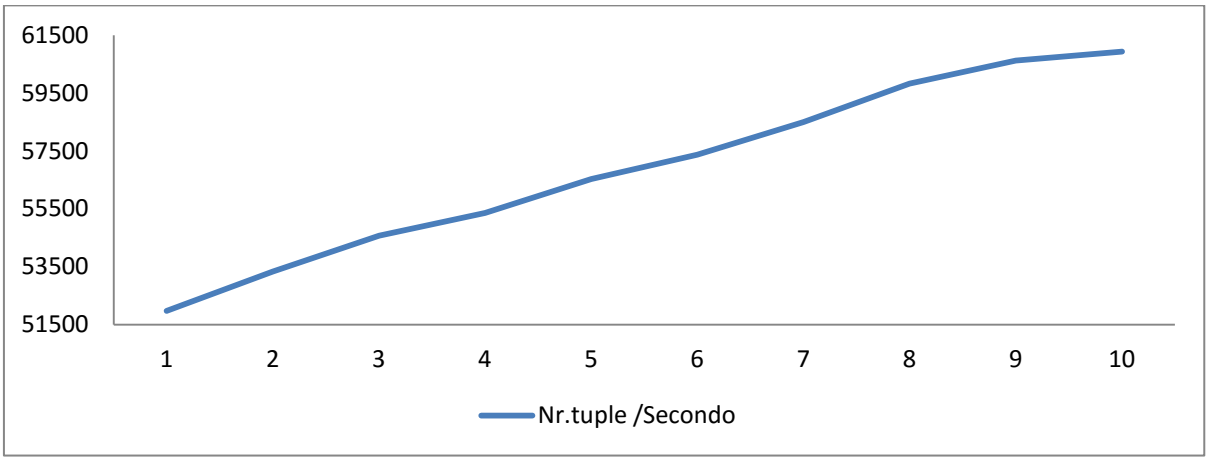


Grafico 5.16 - Throughput media Test 7

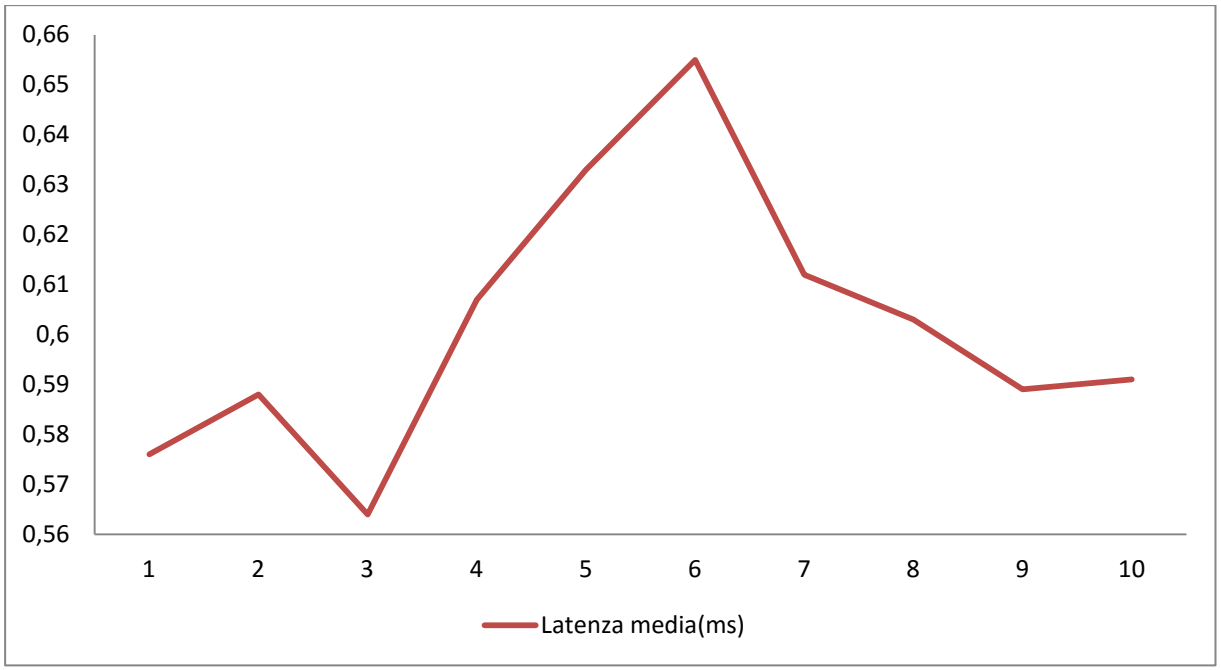


Grafico 5.17 – Latenza media Test 7

Dai dati del settimo test si può notare che:

- L'andamento del throughput segue un andamento lineare, all'aumento delle risorse aumentano anche le tuple elaborate. In questo test avendo aumentato la soglia superiore del throughput non abbiamo lo schiacciamento del grafico nella fase finale;
- Discorso analogo per la latenza, che raggiunge un picco all'incirca a metà delle prove, in seguito si abbassa poiché le tuple in ingresso sono di meno in rapporto all'aumento delle risorse.
- Il comportamento del settimo test è praticamente uguale al quarto test, quindi non ci sono cambiamenti significativi in questo caso specifico

5.8 Test 8

Nell'ottavo test si aumenta l'input a uno solo dei due spout in ingresso come nel settimo solo che in questo caso lo spout sarà EddystoneReader. Con questo test si può vedere se ci saranno differenze di comportamento rispetto al settimo test. Il throughput medio deve essere fra 300mila e 400mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

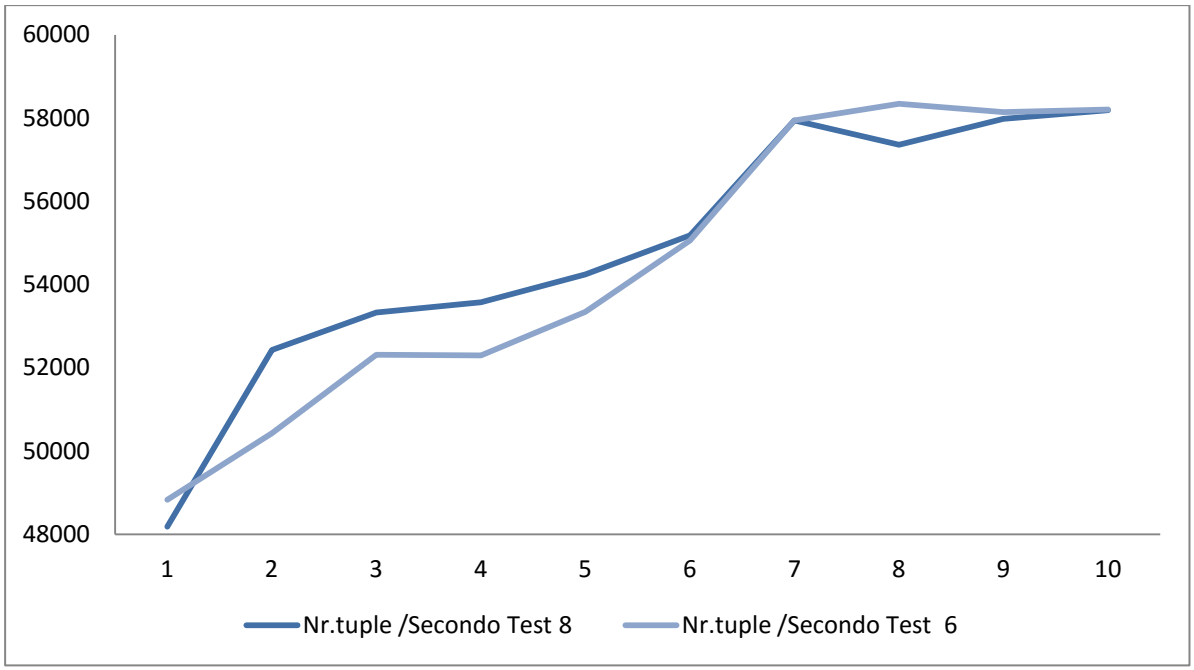


Grafico 5.18 – Confronto throughput Test 8 e Test 6

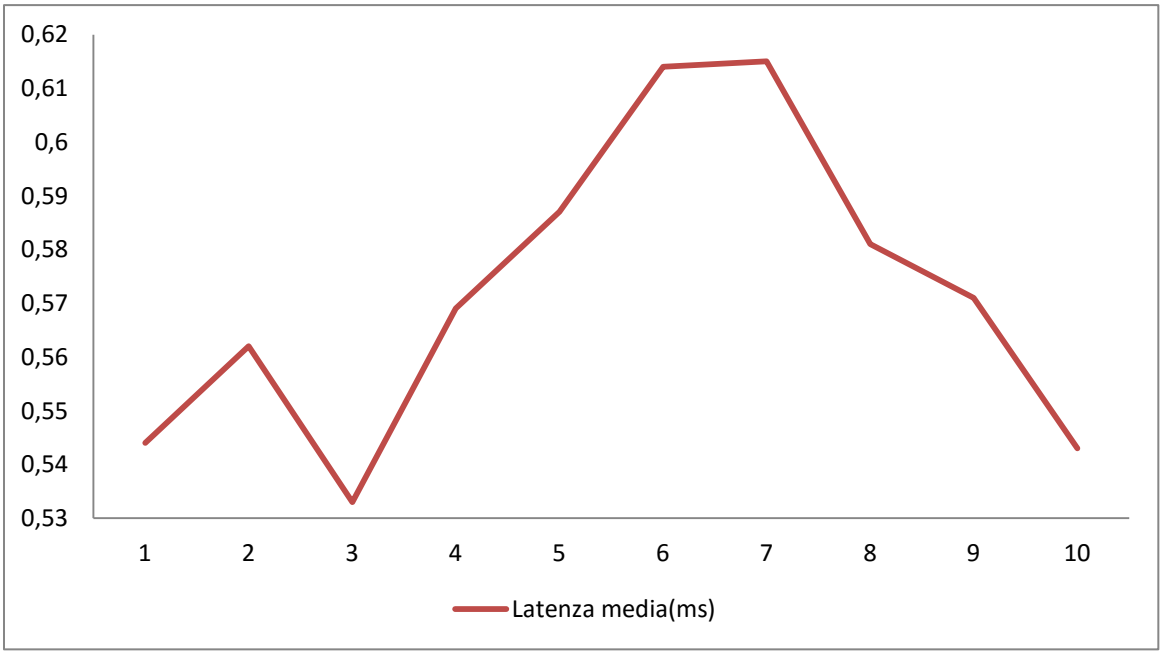


Grafico 5.19 – Latenza media Test 8

Dall'ottava batteria di test si può notare che:

- L'andamento del throughput è simile a quello del sesto test; quindi pure in questo caso aumentando l'input dello spout Eddystone non si ha un incremento significavo delle performance anzi rimangono praticamente uguali al sesto test. Il collo di bottiglia è rappresentato come detto più volte dalla classe PlaceFromID che a ogni esecuzione accede in lettura ad Apache Cassandra per avere una stringa 'place' partendo dall'ID

di 128 bit e non riesce a migliorare le proprie prestazioni. Ci possono essere due soluzioni a questo problema come detto anche precedentemente:

  - Si potrebbe aumentare la cardinalità delle risorse Cassandra, che non ha problemi di scalabilità, spostandolo su un cluster di risorse invece che in un solo nodo;
  - Si potrebbe utilizzare un database ottimizzato per le letture come ad esempio LDAP.

- Il grafico della latenza presenta per la prima volta un picco che dura per due prove; ciò potrebbe essere spiegato che fra le due prove non c'è un grande cambiamento di risorse, le tuple in ingresso sono le stesse quindi ci si può aspettare una latenza media in sostanza uguale.

5.9 Test 9

Nel nono test si aumenta l'input a entrambi gli spout in ingresso per vedere come reagisce la topologia soprattutto in confronto ai test numero sette e otto. Il throughput medio deve essere fra 300mila e 400mila (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

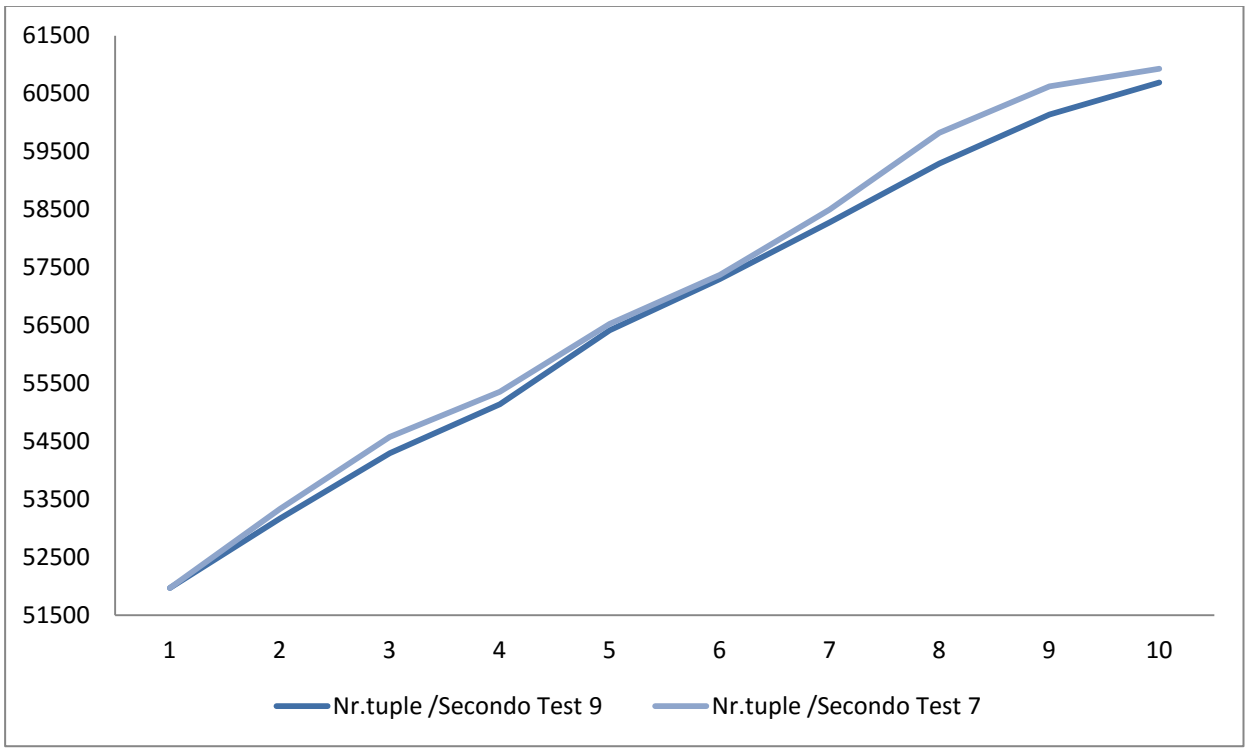


Grafico 5.20 – Confronto throughput Test 9 e Test 7

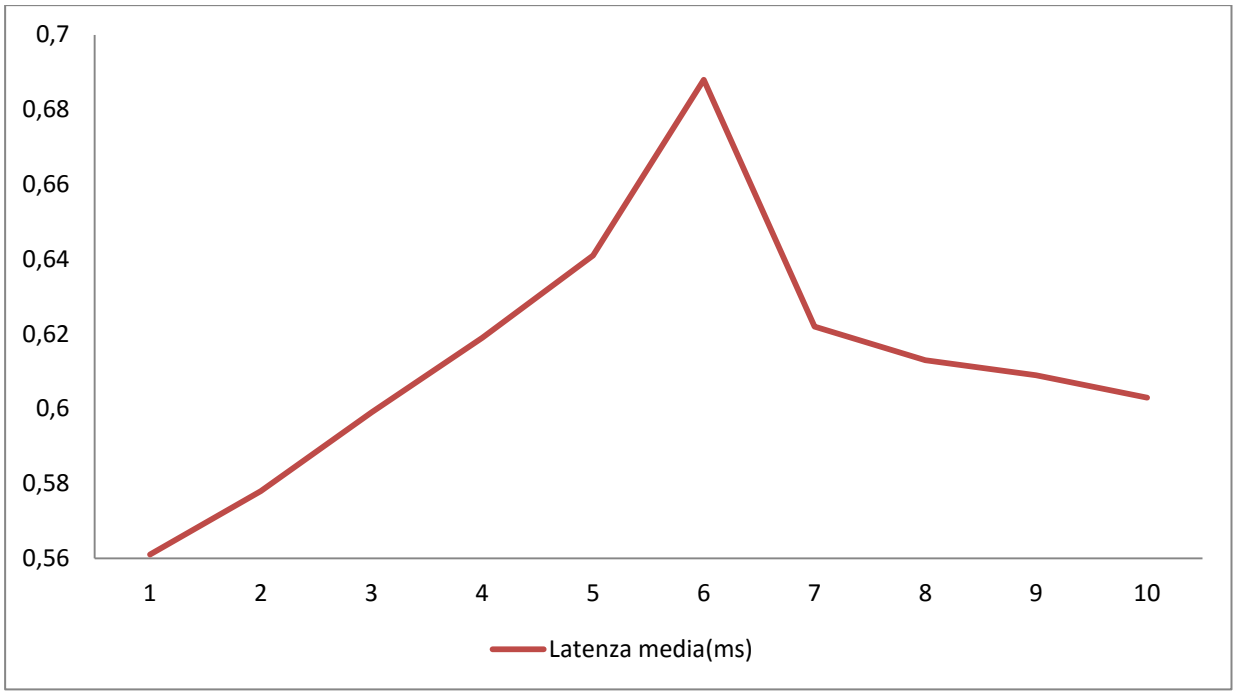


Grafico 5.21 – Latenza media Test 9

Dai risultati del nono test si può notare:

- Il throughput come nel caso del quinto test segue un andamento similare a quando si aumenta solo l'input allo spout XMPPReader;
- La latenza media ha un picco nella sesta prova anche se generalmente è più alta del settimo test, questo ovviamente per i motivi di carico eccessivo delle code.

5.10 Test 10

In questi due ultimi test si andranno a vedere degli aspetti non visti nei test precedenti e anche le modalità saranno diverse. Il decimo test è composto da cinque prove di durata diversa partendo dalla prima che dura dieci minuti fino all'ultima che dura cinquanta minuti aumentando sempre di dieci minuti alla volta per ogni prova. Le risorse iniziali saranno sempre uguali per ogni prova, quindi quello che si vuole vedere con questo test è quanto cambiano le performance quando si va a regime o meglio quanto penalizza la fase di start le prestazioni in tempi brevi di elaborazione. In questa prova ci sarà solo un worker e quindici executor (ventuno thread) inizialmente. Il throughput medio deve essere fra 200mila e 300mila al minuto (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.2 ms e 2.5 ms.

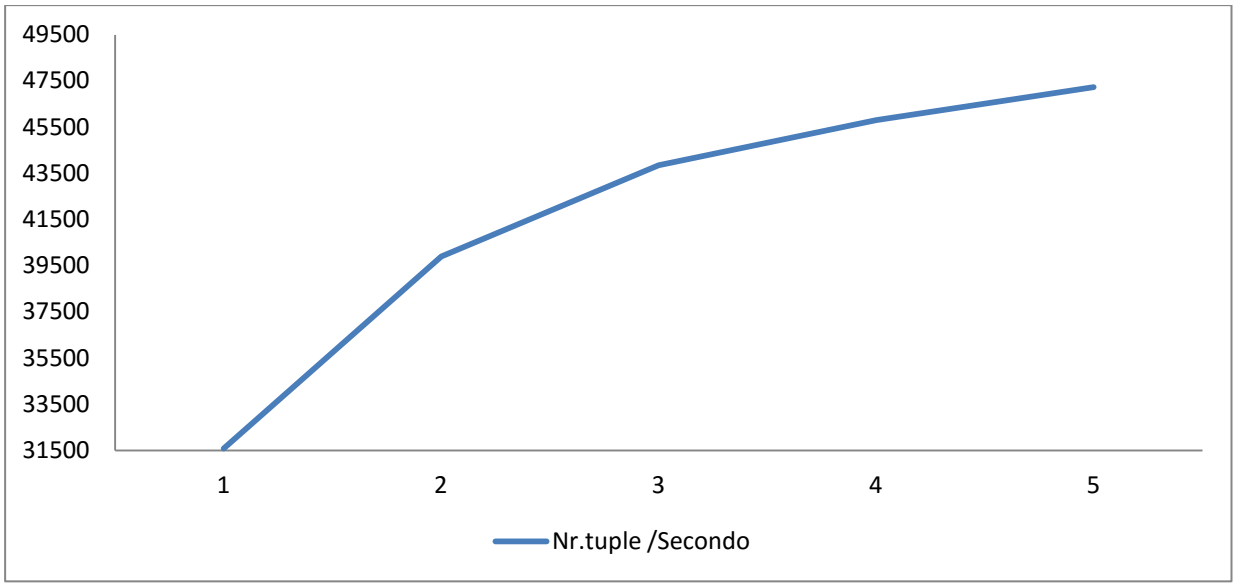


Grafico 5.22 – Throughput Test 10

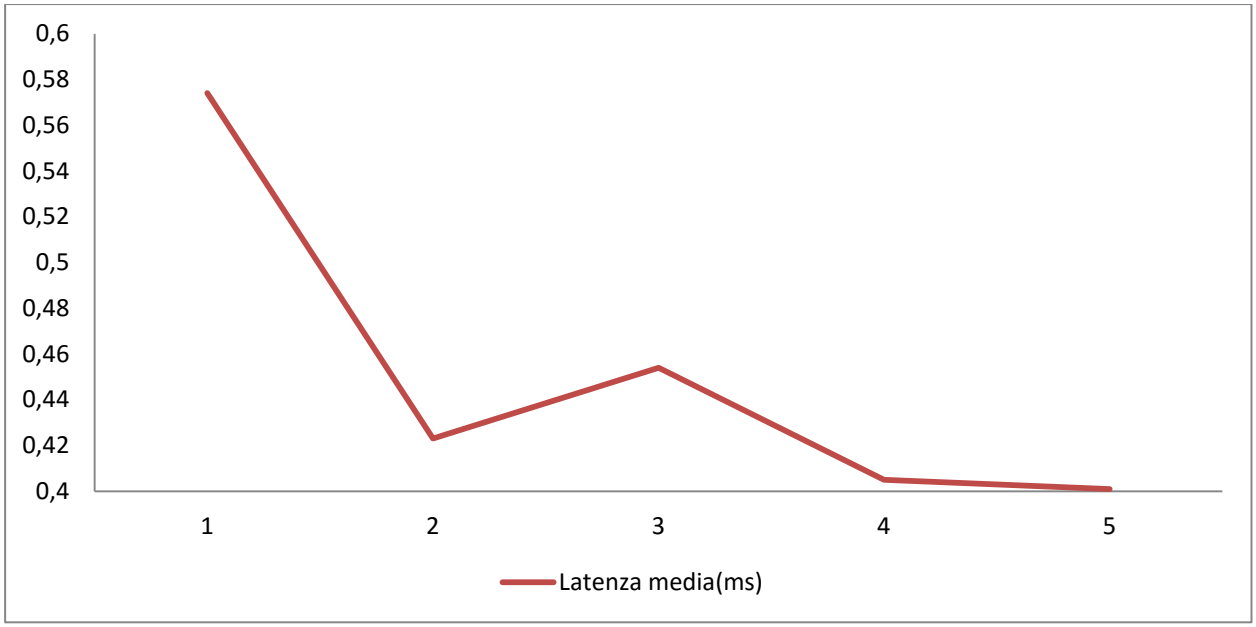


Grafico 5.23 – Latenza media Test 10

Dal decimo test possiamo vedere che:

- A regime le prestazioni migliorano o meglio probabilmente le tuple prodotte ogni dieci minuti sono le stesse solo che nei primi dieci minuti per 60-75 secondi vengono elaborate poche tuple e ovviamente nel calcolo del throughput medio nei dieci minuti questo fa cambiare le statistiche.
- Il discorso per latenza è simile, più tempo si trascorre più regolarmente le tuple vengono elaborate portando ad abbassarne il valore.

5.11 Test 11

Nell'ultimo test si mette il sistema sotto sforzo per quanto riguarda il ribilanciamento a ribasso, cioè le tuple in ingresso e quindi elaborate sono di più di quelle che si vuole che vengano elaborate. Pure questa batteria è composta da cinque prove, però le risorse cambiano fra i diversi test invece la durata è sempre dieci minuti. Il throughput medio deve essere fra 200mila e 300mila al minuto (valori di soglia minore e maggiore rispettivamente), invece la latenza esecutiva deve essere tra 0.5 ms e 2.5 ms.

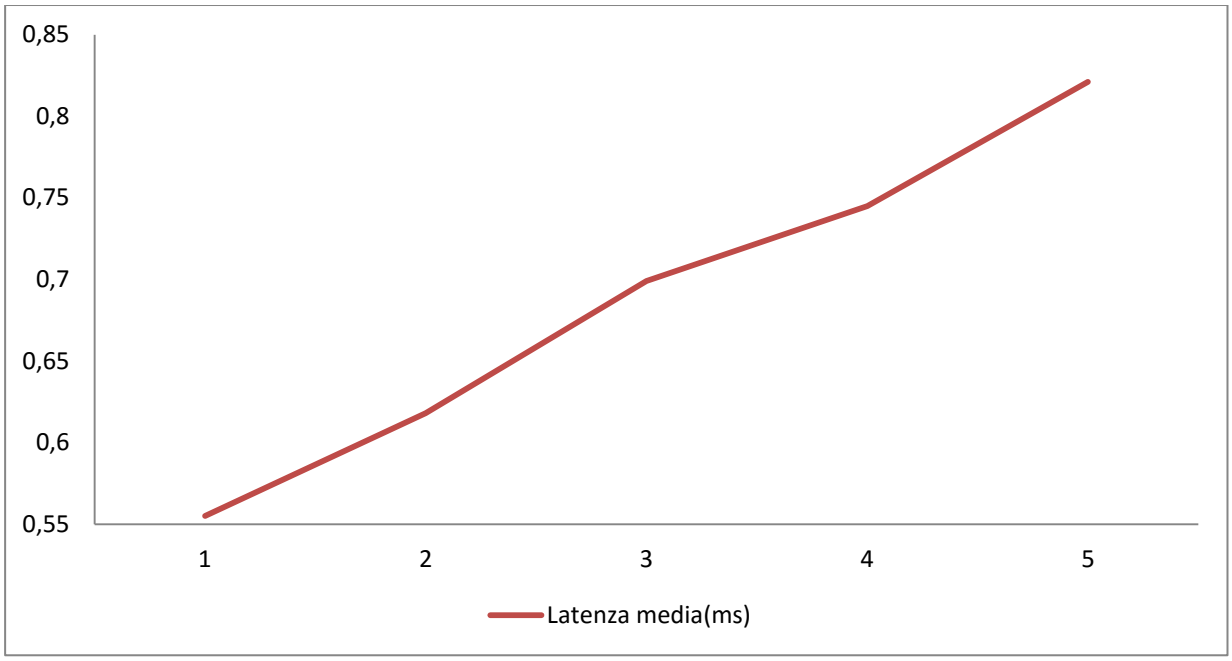


Grafico 5.24 – Latenza media Test 11

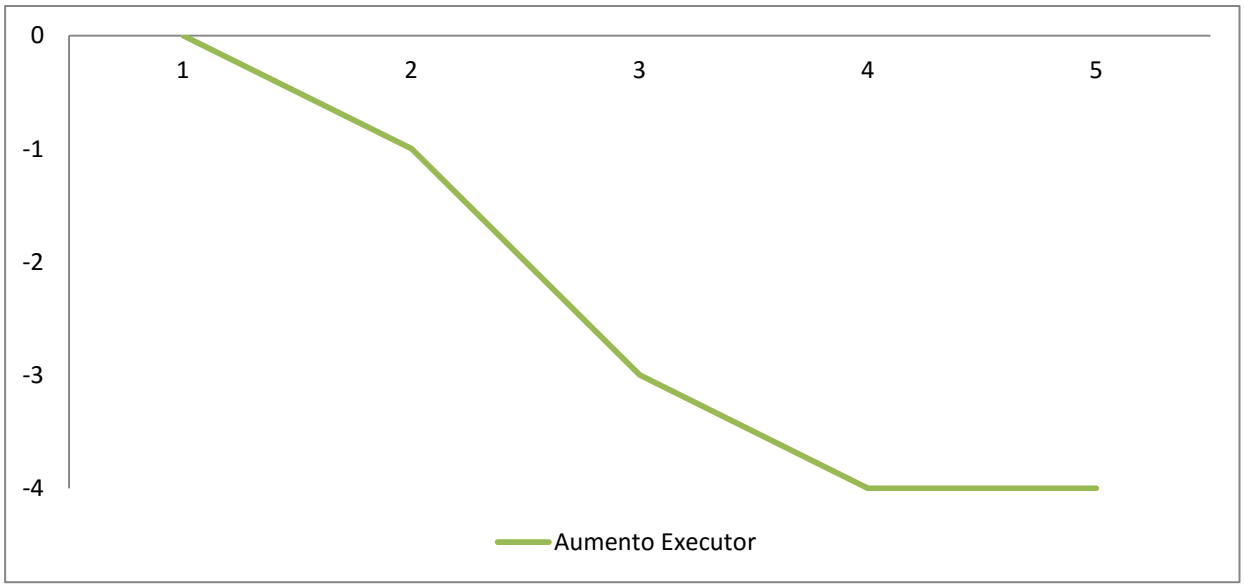


Grafico 5.25 – variazione Executor nel Test 11

Dai risultati dell'ultimo test possiamo notare:

- Come c'era da aspettarsi aumentando le risorse teoricamente si riescono a elaborare più tuple però siccome in questo caso c'è un limite superiore molto stringente per il throughput questo porta ad attese più lunghe nelle code per le tuple per essere elaborate e quindi ciò porta a un aumento progressivo della latenza media.
- Per quanto riguarda il numero di executor quasi sempre il sistema di ribilanciamento arriva al limite (cioè non può diminuire più di così) o arriva in un punto in cui le risorse producono il numero di tuple giuste in uscita. Ovviamente più le risorse aumentano più diminuisce il numero di executor rispetto alla fase iniziale.

Dai dati di questi test si posso fare importanti riflessioni:

- Con l'aumento degli executor le prestazioni migliorano ma non di tanto (quindi se possibile si dovrebbe sempre cercare di aumentare il numero di executor) invece aumentando il numero di worker le prestazioni ricevono un'impennata;
- Per scegliere in maniera ottimale le risorse iniziali bisogna avere conoscenza del flusso di ingresso, partire con un'assegnazione casuale delle risorse potrebbe portare il sistema a lavora non in maniera ideale e sotto di prestazioni rispetto a quello che potrebbe fare;
- Per i diversi input:
    - Per i test dal numero due al numero cinque le migliori prestazioni in termini di throughput e latenza media si hanno nel terzo test e in maniera più approfondita nella decima prova che potrebbe rappresentare un setting iniziale per quel determinato stream di dati;
    - Per i test dal numero sei al numero nove le migliori prestazioni in termini di throughput e latenza media si hanno nel settimo test in particolare si ha un possibile setting ottimale nella nona prova;
- Il sistema di load balancing sembra che risponde in maniera ottimale sia a condizioni in cui si deve aumentare il parallelism hint sia quando lo si deve diminuire.

Conclusioni

Il mercato dei Big Data e dei sistemi di stream processing è in continuo aumento con significativi investimenti sia da parte del settore pubblico che privato. Insieme a questo i requisiti per manipolare ed estrarre informazioni sono sempre più stringenti in termini di volume di dati, della varietà di essi e della velocità di elaborazione e risposta. Sempre nuove piattaforme e nuove metodologie vengono pubblicate per affrontare in maniera più intelligente questi tipi di problema.

Gli *Instant Coupon* cominceranno a fare parte della nostra quotidianità nel fare spese, nel comprare materiale *hi-tech*, vestiti, libri, *videogame* in quanto si è dimostrato l'efficacia di avere delle offerte ad hoc per ogni cliente più che offerte generali, portando vantaggi ai clienti stessi e alle aziende.

Il mio lavoro di tesi si concentra sulla costruzione di un prototipo pienamente funzionante che si occupi soprattutto della ricezione ed elaborazione delle tuple che posso arrivare dai diversi clienti quando si trovano nelle strutture. Per avere questi tipi di informazioni si utilizzano due principali tecnologie che sono Google Eddystone e XMPP. In seguito vengono estratte le informazioni principali e memorizzate in un DBMS NoSQL come Apache Cassandra.

Si è cercato di tenere la topologia il meno complesso possibile poiché aggiungendo altri elementi si aumenta solamente il tempo di risposta. Siccome è un sistema che lavora in tempo reale si è cercato di dare molta importanza a questo vincolo.

La più grande difficoltà che si è riscontrata in questo lavoro è stata sicuramente sulla parte di *load balancing* in quanto Apache Storm nelle versioni fino a ora non offre strumenti per fare un bilanciamento del carico automatico in base allo stream di ingresso e performance volute. Si è cercato di provare un approccio nuovo e innovativo ma che nello stesso tempo garantisse buone prestazioni. Dai risultati sperimentali in seguito ai test eseguiti su ICoS si può notare che il sistema risponde molto bene a cambi della mole di ingresso di dati sia ribilanciando in positivo sia in negativo. Le prestazioni di norma aumentano linearmente all'aumentare delle risorse totali. In maniera specifica aumentando il numero di *executor* le prestazioni aumentano di poco poiché si va a incrementare le risorse di solo una parte della topologia invece aumentando il numero di *worker* le prestazioni crescono in maniera decisa ma lineare, questo perché aumentando il numero di *worker* si aumenta in maniera indiretta anche il numero di *executor*. Unica pecca è rappresentata dalla lettura in Cassandra in quanto non è un sistema ottimizzato per continue letture e ciò rappresenta probabilmente il limite principale di questo prototipo.

Un'altra fase critica è comunque rappresentata dal ribilanciamento stesso che non è un'operazione immediata poiché Storm ci mette circa sessanta secondi a creare nuovi componenti o a toglierne da quelli esistenti, questo si poteva notare nei test nei quali dopo che si è stabilizzata la topologia (quindi senza cambiamenti di risorse o stream) le performance migliorano gradualmente.

Dei test possibili ulteriori che si potrebbero fare è mettere sia Storm che Cassandra in un cluster di risorse e vedere se il trend di prestazioni rimane in linea con quelli effettuati o ci sono dei miglioramenti (considerando che sia Storm che Cassandra sono altamente scalabili).

In futuro si potrebbe usare ICoS in un sistema più grande che grazie a questi dati incrociati con altri sulla storia commerciale e dati di trend di mercato potrebbe creare a pieno un servizio di *Instant Coupon.* Un possibile miglioramento per rendere ICoS ancora più performante è un usare un sistema come ad esempio LDAP in modo da eliminare l'unica criticità vera che mina le prestazioni del sistema.

Per quanto riguarda le fonti d'informazioni la direzione futura sarà sicuramente passare completamente a tecnologie e beacon (non solo Eddystone) in quanto i trend di mercato spingono in questa direzione oltre al fatto che le tecnologie a beacon sono ottimali per dare informazioni di posizione in ambienti indoor e riescono a dare delle info più specifiche in confronto ad esempio a XMPP.

Un’aggiunta possibile al sistema è mettere Trident on top alla topologia, solo che questa scelta bisogna che sia verificata dal punto di vista delle prestazioni, giacché con Trident si ha una diminuzione delle performance ma un aumento dell’espressività e della potenza di elaborazione, se con la diminuzione delle prestazioni si rimane in un range ancora accettabile sicuramente Trident sarebbe una scelta da fare.

Immagini

Tabelle

Grafici