Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di applicazioni distribuite con lo stack SMACK

Relatore
Prof. Ing. Mirko Viroli

Presentata da
Emiliano Ciavatta

Correlatore
Dott. Roberto Casadei

Seconda Sessione di Laurea
Anno Accademico 2017-2018

Sommario

SMACK è uno stack composto da un insieme di software open source che possono essere utilizzati per sviluppare applicazioni distribuite. I componenti dello stack sono: Spark, un framework per effettuare calcolo distribuito, Mesos, serve per gestire e coordinare le risorse del cluster, Akka, una libreria che implementa l'architettura basata sul modello ad attori, Cassandra, un gestore di database non relazionale e distribuito e Kafka, una piattaforma di stream processing. Ciascuno di questi componenti è indipendente: ognuno ha le proprie API, una propria configurazione e non interagiscono fra loro.

Lo scopo del progetto è quello di costruire una solida base di partenza su cui sarà possibile progettare e implementare applicazioni che lavorino utilizzando lo stack SMACK. Si vuole realizzare quindi un framework che non contenga logica applicativa, ma che permetta di costruire applicazioni distribuite utilizzando un'architettura già pronta, senza che sia necessario preoccuparsi di come i componenti dello stack interagiscano fra loro e senza dover ricorrere a tecniche di progettazione per lo sviluppo di sistemi distribuiti. Il framework dovrà semplificare al massimo la progettazione del sistema distribuito.

Tramite il framework dovrà essere possibile realizzare applicazioni web-based. L'applicazione potrà interagire con gli utenti attraverso l'architettura REST, un'architettura ideale per i sistemi distribuiti. Il framework dovrà permettere di sviluppare applicazioni elaborate mantenendo bassa la complessità del sistema. Dovranno essere garantite tutte le proprietà di un sistema distribuito, quali la scalabilità, la modularità, la tolleranza agli errori, l'alta affidabilità e la consistenza dei dati, l'efficienza nell'uso delle risorse e le migliori performance. L'applicazione inoltre dovrà essere necessariamente sicura.

Indice

Introduzione

Quando si sviluppa un'applicazione è necessario tenere in considerazione il numero di utenti che devono essere serviti. È necessario che l'ordine di grandezza venga definito già in fase di analisi perché potrebbe stravolgere l'architettura del sistema. Quando il numero di utenti che devono usufruire simultaneamente dell'applicazione è limitato è possibile costruire un sistema con un'architettura centralizzata, che utilizzi soltanto le risorse di una macchina. Questo è il caso ad esempio di una azienda medio/piccola perché il numero dei dipendenti che devono utilizzare il sistema è costante. Quando si deve sviluppare un'applicazione che deve soddisfare un numero indefinito e sempre crescente di utenti le risorse di una singola macchina non bastano più: è necessario creare un sistema distribuito. Questo è il caso di un'applicazione che lavora nell'ambito dell'Internet of Things, dove il numero degli oggetti intelligenti connessi al sistema può crescere esponenzialmente, o nel caso di un social network, dove non si conosce il numero delle persone connesse.

Lo scopo di questa tesi è quello di realizzare una struttura di un sistema distribuito. La caratteristica principale che il sistema dovrà avere è la scalabilità, una proprietà dell'ingegneria del software che indica la capacità di un sistema di gestire e processare una quantità sempre crescente di lavoro. Il sistema dovrà quindi essere in grado di espandere le sue capacità quando ci sarà maggior richiesta e restringersi quando le risorse hardware saranno inutilizzate. Occorre che questo processo non implichi un degradamento delle performance o un aumento della complessità del sistema. Per realizzare il sistema verrà utilizzato lo stack SMACK, un insieme di software open source

indipendenti che ricoprono ciascuno una funzionalità diversa di un sistema distribuito. Lo scopo del progetto è quello di integrare e connettere i componenti per poter realizzare un'applicazione concreta. Il progetto sarà composto da uno scheletro che si preoccuperà di orchestrare il sistema distribuito sul quale si potrà implementare la logica dell'applicazione.

Di seguito sono elencati i capitoli trattati dalla tesi.

- **Componenti dello stack**. In questo primo capitolo è stata effettuata una rassegna dei cinque componenti che fanno parte dello stack. Ciascun componente verrà analizzato singolarmente e verranno descritti i requisiti, le caratteristiche e le funzionalità di ciascuno di essi.
- **Analisi dei requisiti**. È la descrizione dei requisiti del progetto che con questa tesi si intende realizzare. Verranno descritte le caratteristiche e le funzionalità che il framework dovrà avere.
- **Design dell'architettura**. Descrive la fase di progetto del framework prestando particolare attenzione all'architettura utilizzata dal sistema, ovvero il modello ad attori. Verranno descritti i paradigmi e i pattern di progettazione utilizzati.
- **Implementazione e testing**. Vengono descritte le tecniche di implementazione e le modalità di testing del sistema. Saranno introdotte specifiche da seguire per rendere il sistema sicuro.
- **Un caso di studio**. È presentata un'applicazione realizzata con il framework progettato. Verranno descritti alcuni strumenti realizzati per gestire il cluster e per misurare le performance dell'applicazione.

Il codice sorgente del progetto è rilasciato sotto licenza MIT ed è disponibile al seguente indirizzo: `https://github.com/eciavatta/smack-template`. Il framework è stato realizzato interamente in Scala, un linguaggio di programmazione multi-paradigma studiato per integrare le caratteristiche e le funzionalità dei linguaggi orientati agli oggetti e dei linguaggi funzionali[1]; è particolarmente adatto per lo sviluppo di sistemi distribuiti.

[1] https://www.scala-lang.org/

Capitolo 1

Componenti dello stack

1.1 Apache Spark

Una delle caratteristiche principali delle applicazioni moderne è la quantità di dati che esse generano. Dato il basso costo dei supporti di memorizzazione salvare tutti i possibili dati che un'applicazione può generare è fondamentale, perché da essi si può estrarre conoscenza. Per poter estrarre informazioni i dati però devono essere elaborati. Apache Spark fornisce un motore di elaborazione di dati che opera in maniera distribuita per permettere di processare grandi volumi di dati in poco tempo.

Cos'è Spark

Spark è un motore per il processamento di dati facile da usare, efficiente, veloce e capace di adattarsi a qualsiasi tipo di computazione che è necessario effettuare. È una piattaforma che opera in modo distribuito e che è progettata per gestire autonomamente le risorse del cluster. Spark estende MapReduce, uno dei più popolari modelli utilizzati per l'elaborazione dei Big Data. Lo fa nel modo più efficiente possibile supportando la maggior parte delle più comuni tipologie di computazione, permettendo ad esempio di eseguire query interattive, fare *stream processing* o effettuare *batch processing*. La velocità è una delle caratteristiche principali di Spark ed è una proprietà

fondamentale per i software che devono processare dataset di notevoli dimensioni. Un'altra funzionalità di Spark è quella di poter processare grandi quantità di dati direttamente in memoria; questo permette un aumento di velocità notevole se si confronta con altri motori che operano tramite lettura e scrittura su disco, come ad esempio MapReduce. Nonostante questo le prestazioni sono aumentate anche per le applicazioni che operano sfruttando dischi come strumenti di memorizzazione temporanea.

Una delle caratteristiche fondamentali di Spark è quella di permettere di effettuare una vasta quantità di operazioni che possono essere effettuate in modo distribuito all'interno del cluster con la stessa piattaforma. Per poter estrarre informazioni utili da dati grezzi spesso sono necessarie più operazioni: Spark permette di eseguirle tutte utilizzando lo stesso tipo di API, quindi abbattendo costi di progettazione, sviluppo e manutenzione, concatenando i processi da effettuare. Tramite la concatenazione i dati possono essere trasformati in modo distribuito tramite una *pipeline* di processi.

Le sorgenti di dati utilizzati come input della fase di elaborazione possono essere le più disparate: tramite Spark è possibile leggere grandi file testuali o binari, prelevare dati da sorgenti Hadoop distribuite o da altri database più comuni, come Apache HBase e Apache Cassandra. Il collegamento con quest'ultimo è fondamentale per poter sviluppare un'applicazione con lo stack SMACK.

Spark è composto da più componenti che possono interagire fra loro per creare un processo di elaborazione dei dati. Di seguito verranno descritti brevemente i componenti principali. L'elenco completo delle funzionalità di ciascun componente è disponibile sulla documentazione ufficiale sul sito Apache[1].

- **Spark Core**: è il componente principale e comprende il motore computazionale che si occupa di organizzare, distribuire e monitorare le applicazioni realizzate con Spark. Le applicazioni sono composte da

[1] http://spark.apache.org/docs/latest/

task che vengono eseguiti in modo distribuito nelle macchine targate come *worker*. Lo Spark Core si occupa di coordinare questo processo. Inoltre comprende altre funzionalità di base, come il supporto per il *task scheduling*, la gestione della memoria, il supporto per la gestione degli errori e l'interazione con i sistemi di memorizzazione.

- **Spark SQL**: è il modulo che può essere utilizzato per lavorare con i dati strutturati. Permette di eseguire query tramite SQL in modo distribuito utilizzando diversi linguaggi di programmazione a scelta. Sono disponibili diversi driver che permettono a Spark di collegarsi ai principali gestori di database.

- **Spark Streaming**: è un componente che permette di processare in tempo reale flussi di dati proveniente da più sorgenti. Le API fornite da questo componente sono molto simili a quelle utilizzate dallo Spark Core, permettendo quindi di passare da un sistema all'altro facilmente.

- **MLlib**: attraverso questo modulo è possibile applicare algoritmi di *machine learning* sui dati direttamente con Spark. MLlib fornisce diversi tipi di algoritmi di apprendimento automatico che funzionano in modo distribuito senza che sia necessario implementarli a mano o utilizzare librerie esterne.

- **GraphX**: è la libreria che permette di lavorare con i grafi. Tutte le operazioni che è possibile effettuare sui grafi, come la creazione di un sotto grafo o la trasformazione dei vertici, vengono effettuate in maniera distribuita dai nodi del cluster.

- **Cluster Manager**: è il componente che si occupa implicitamente di gestire il cluster. Spark può essere eseguito su diversi tipi di cluster o in modalità standalone. Tra i tipi di cluster disponibili c'è Hadoop

YARN e Apache Mesos; quest'ultimo è utilizzato proprio dallo stack SMACK.

Come funziona

Spark è una libreria che permette di sviluppare applicazioni che possono eseguire delle operazioni distribuite nel cluster. Sono fornite le API per tre linguaggi: Scala, Java e Python. L'applicazione sviluppata con Spark consiste in un *driver* che esegue la funzione main del programma. Il *driver* ha a disposizione l'oggetto `SparkContext` che serve per coordinare tutti i processi eseguiti su più nodi all'interno del cluster. L'oggetto `SparkContext` può connettersi a diverse tipologie di *cluster manager*, al quale viene richiesto di allocare le risorse per eseguire la computazione. Il *driver* si connette con i nodi dei cluster tramite gli esecutori, dei processi avviati soltanto sui nodi *worker* che hanno il compito di eseguire a comando porzioni di lavoro e memorizzare dati dell'applicazione. Ciascuna applicazione ha i suoi esecutori che vengono creati e persistono per tutta la durata dell'applicazione e utilizzano tutti i processori fisici e logici messi a disposizione dall'esecutore. Questo permette di isolare le applicazioni che utilizzano Spark nel caso sia utilizzato lo stesso cluster per eseguirne più di una. L'isolamento avviene sia nei *driver* sia negli esecutori: è quindi impossibile scambiare dati fra due applicazioni senza utilizzare sistemi di trasmissione o memorizzazione esterni. In figura 1.1 è presente un diagramma che mostra quanto spiegato.

Cosa eseguire è definito nel codice dell'applicazione che deve essere pacchettizzata in archivi in formato JAR che devono essere inviati agli esecutori. Quando gli esecutori sono avviati tramite l'oggetto `SparkContext` è possibile inviare task, unità di lavoro che devono essere eseguite in modo distribuito sfruttando tutte le risorse allocate. Le operazioni lanciate dai task che devono essere eseguite parallelamente possono essere raggruppate in *job*, che a loro volta sono suddivisi in *stage* che definiscono le unità elementari.

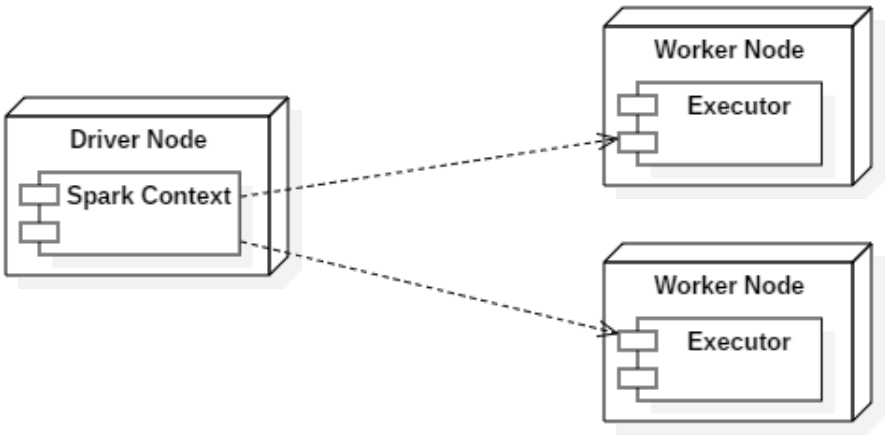

Figura 1.1: Diagramma di deployment che mostra le connessioni tra l'oggetto `SparkContext` e gli esecutori lanciati nei nodi *worker*

Dall'applicazione è possibile accedere ad un qualsiasi *resilient distributed dataset* (RDD), una collezione di elementi partizionata e suddivisa fra i nodi che formano il cluster. Tramite gli RDD è possibile eseguire operazioni in parallelo sulla collezione. Gli RDD rappresentano una collezione astratta. La sorgente dei dati può essere qualsiasi, ad esempio può essere un file presente nel file system del cluster Hadoop o una semplice collezione in memoria presente nell'applicazione *driver*. Tramite gli RDD è possibile effettuare qualsiasi tipo di operazione sui dati. È possibile sempre tramite il *driver* richiedere a Spark di memorizzare i dati della collezione nei nodi *worker* attraverso un sistema di caching per poterli riutilizzare senza doverli ogni volta ritrasmettere. Gli RDD implementano un sistema autonomo per ripristinare lo stato della collezione in caso di fallimento di uno o più nodi del cluster.

È possibile che i nodi *worker* necessitino di informazioni aggiuntive per effettuare operazioni sugli RDD, come variabili che contengono informazioni sullo stato del sistema. Il *driver* può utilizzare delle variabili condivise per inviare informazioni ai *worker*, effettuando una copia delle variabili dichiarate nel *driver* sui task del *worker*. Possono essere utilizzati due tipi di variabili: le

variabili *broadcast*, che sono utili per memorizzare informazioni temporanee su tutti i nodi, e gli accumulatori, che sono variabili intere che possono essere solo incrementate per essere utilizzate come contatori, ad esempio per contare il numero di operazioni effettuate da tutti i *worker*.

Chi lo utilizza

Spark può essere utilizzato in diversi ambiti. Tramite la shell interattiva può essere utilizzato dai *data scientist* per analizzare e modellare i dati presenti su database distribuiti utilizzando SQL come linguaggio di interazione. L'utilizzo principale di Spark è quello di fornire un sistema per realizzare applicazioni che processano i dati, sia in real-time sia in modalità batch. Data la sua vasta gamma di funzionalità, la sua facilità d'utilizzo e d'apprendimento e il fatto che sia ormai diventata una tecnologia stabile e matura, Spark è diventato uno dei software più utilizzati per effettuare *data processing*.

Spark è nato attraverso un progetto elaborato da ricercatori che hanno lavorato per lo sviluppo di Hadoop MapReduce[1]. Lo scopo del progetto era quello di creare un sistema che risolvesse i problemi di inefficienza di MapReduce che sorgevano quando venivano processati i job. Spark è stato progettato fin da subito per essere veloce ad effettuare query interattive ed eseguire algoritmi iterativi, introducendo il supporto alla memorizzazione dei dati su memoria principale ed un sistema efficiente di gestione degli errori. Successivamente è stato pubblicato come progetto open source e trasferito alla fondazione Apache, dove è diventato uno dei progetti più popolari e più utilizzati.

1.2 Apache Mesos

Un sistema distribuito si basa sul presupposto che ci siano diverse macchine indipendenti che lavorino insieme per risolvere uno stesso problema. Quando il numero delle macchine è basso si può tentare di configurare ciascuna macchina manualmente e avviare i componenti dell'applicazione in modo statico, cioè utilizzando sempre la stessa configurazione. Quando si ha a che fare con un numero elevato di macchine, nei sistemi distribuiti più grandi, configurare manualmente ciascuna macchina diventa un problema. Apache Mesos nasce per risolvere questo problema. Questo strumento permette di gestire e organizzare un numero infinito di macchine in modo rapido e affidabile.

Cos'è Mesos

Mesos è una piattaforma per condividere le risorse disponibili nel cluster con diverse applicazioni. È un sistema che permette di coordinare tutte le macchine presenti all'interno del cluster e trattarle come un'unica entità logica. Questo permette di risolvere numerosi problemi, come quello di non dover gestire manualmente l'hardware dopo aver configurato il servizio di Mesos, o non dover più assegnare delle applicazioni ad un insieme statico di macchine. Con Mesos è possibile assegnare dinamicamente compiti da eseguire a macchine con certe specifiche, oppure è possibile lasciare scegliere direttamente a Mesos come organizzare al meglio le risorse. In questo modo si risparmia del tempo che può essere utilizzato per problemi più importanti.

Mesos può essere visto anche come un sistema di dispiegamento delle applicazioni. Esistono altri tipi di software che si occupano di questo, come Ansible e Puppet, che ricoprono le stesse funzionalità di altri sistemi di scripting, come Bash e Perl, in modo semplice, intuitivo e schematico. Questi tool hanno però una limitazione: la configurazione che viene scritta sulle macchine del cluster è statica. Quando la configurazione viene completata non è più possibile riallocare risorse in modo dinamico per soddisfare ad esempio le necessità dell'applicazione.

Con Mesos viene introdotto il concetto di framework, un'entità a parte che deve essere lanciata all'interno del cluster che contiene la logica necessaria per installare, avviare e monitorare una o più applicazioni. Esistono framework già pronti per essere utilizzati, come Marathon, un framework open source sviluppato da Mesosphere per la gestione di applicazioni di lunga durata. Esistono anche altri tipi di framework, ad esempio per gestire istanze di web server o per avviare job in intervalli di tempo prefissati.

La potenza di Mesos risiede nel fatto di permettere di eseguire framework dinamici che possono effettuare decisioni in tempo reale basandosi sulle caratteristiche del cluster e sulle condizioni attuali del traffico nella rete. I framework possono utilizzare tutte le risorse disponibili nel cluster o solo parte di esse e possono scegliere di utilizzare una strategia specifica di organizzazione delle risorse per adattarsi alle esigenze di una applicazione. Il framework si deve occupare di intercettare gli errori dell'applicazione e reagire ad essi in modo istantaneo, eseguendo ad esempio nuove istanze dei servizi che sono falliti e rimuovendo dal sistema quelli difettosi.

Mesos è una piattaforma per ospitare applicazioni. Invece di pagare servizi cloud per ottenere soluzioni "Platform as a Service" pronte è possibile utilizzare il cluster Mesos per lanciare, gestire e monitorare tutti i tipi di servizio che si necessitano. Alcuni framework avanzati, come Marathon, includono diverse funzionalità che permettono di raggiungere in autonomia elevati livelli di affidabilità, riavviando ad esempio le macchine che falliscono o pianificando la manutenzione. Altri servizi permettono di implementare il *service discovering*, un esempio sono Mesos-DNS e Marathon-lb, oppure bilanciare le risorse e distribuirle equamente fra le richieste, come fa HAProxy.

Come funziona

Mesos è un sistema distribuito che utilizza un *controller* centrale e tanti *worker*. I *worker* sono progettati per funzionare indipendentemente dal *controller* in modo tale da non esercitare pressioni sul *controller* e non dover attende-

re la disponibilità quando il numero dei *worker* è elevato. Le applicazioni che vengono eseguite su Mesos sono chiamate *framework*. Un *framework* è suddiviso in due parti: lo *scheduler* e gli esecutori.

Per eseguire un *framework* su Mesos occorre prima eseguire il suo *scheduler*, ovvero un processo che è in grado di dialogare con Mesos utilizzando un protocollo stabilito. Quando lo *scheduler* viene avviato si connette immediatamente a Mesos in modo tale da recuperare le informazioni sulle risorse che è possibile utilizzare. Quando lo *scheduler* vuole eseguire del lavoro lancia un esecutore, che è semplicemente il *worker* che è assegnato allo *scheduler*. L'esecutore è in grado di eseguire uno o più *task*. Gli esecutori presenti in macchine diverse funzionano in modo indipendente; non dialogano fra loro ma soltanto con lo *scheduler* che viene aggiornato costantemente sulle informazioni dei *task* in esecuzione.

Il cluster Mesos è composto da due componenti: il servizio *master* e il servizio *slave*, che è rinominato in *agent* dalla versione 1.0. I *master* coordinano il cluster, mentre gli *slave* eseguono le applicazioni dentro i container. Le macchine che vengono utilizzate come *slave* condividono le risorse che hanno a disposizione, come la CPU e la memoria. Per suddividere risorse fra varie applicazioni viene utilizzata la containerizzazione attraverso Docker oppure tramite container Linux (LXC).

Nella figura 1.2 vengono mostrate le connessioni che intercorrono fra due macchine *master*.

I *master* sono il cervello del cluster. Assolvono la funzione di coordinazione delle macchine *slave* e hanno diverse responsabilità, elencate di seguito.

- Mantengono informazioni sullo stato dei *task* attivi. Per fornire assistenza alle macchine *slave* e per gestire il cluster, tutti i nodi *master* memorizzano lo stato dell'intero cluster. I dati vengono mantenuti in memoria principale per diminuire la latenza delle richieste. Le macchine che fungono da *master* devono essere dotate quindi di molta memoria.

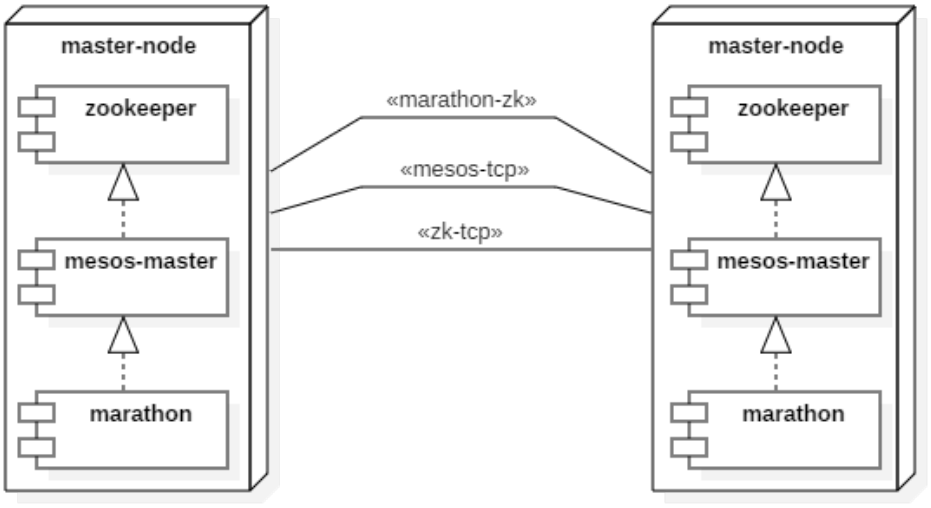

Figura 1.2: Diagramma di deployment che mostra le connessioni tra due nodi master

- Suddividono equamente le risorse fra i diversi *framework* connessi. I *master* sono responsabili di gestire le risorse del cluster e offrirle ai *framework* in base al loro diritto di possedere certe risorse. Ad ogni *framework* è associato un ruolo e ad ogni ruolo è possibile assegnare determinati tipi di risorse in quantità modificabili. È possibile anche assegnare un peso che indica l'importanza e che viene utilizzato al momento dell'allocazione delle risorse.

- Gestiscono l'interazione con gli amministratori. A tal scopo ogni *master* possiede un servizio raggiungibile dal browser per ispezionare e interagire direttamente con il cluster. L'interfaccia utente permette di vedere quante risorse sono disponibili all'interno del cluster, i *framework* connessi con i relativi dettagli di esecuzione, i *task* in esecuzione e i dettagli sulle risorse utilizzate da ciascun *task*. È possibile ispezionare ogni *task* per poter accedere alla *sandbox* dove sono contenuti anche i file di log.

- Garantiscono sempre la disponibilità. In ambienti di produzione è consigliato eseguire tre o cinque *master*. Finché il numero dei *master* in esecuzione è maggiore della metà dei nodi *master* totali il servizio è garantito e Mesos continua a funzionare correttamente. Dato che tutti i *master* conoscono le stesse informazioni ciascun nodo *master* può servire le richieste. Per le decisioni che devono essere effettuate da un solo nodo, come l'allocazione delle risorse, viene eletto un nodo *leader* che assolve questo compito.

Coloro che eseguono le applicazioni sono gli *slave*, che hanno diverse funzionalità, elencate di seguito.

- Gestiscono i container. I container sono un modo semplice e leggero per garantire l'accesso da parte di un'applicazione a una quantità specifica di risorse. Mesos permette l'utilizzo di due tipologie di container, i container standard LXC e i container gestiti dal software Docker, il più popolare software di containerizzazione. Con i container è possibile anche gestire gli accessi e i permessi dell'applicazione in riferimento all'ambiente virtuale creato tramite containerizzazione.

- Mantengono informazioni sui *task* di loro competenza. Ciascuno *slave* ha anche il compito di mantenere in memoria le informazioni sullo stato di esecuzione dei *task* ad esso assegnati. Per rendere più facili le operazioni di debug è possibile interagire con ciascuno *slave* per effettuare diverse operazioni, tra cui recuperare le informazioni di log relative all'intero servizio o ad uno specifico *task*.

- Avvisano sulle condizioni della macchina. Ciascuno *slave* ha il compito di informare i nodi *master* sulle condizioni del servizio. Vengono riportati diversi dati, tra cui la tipologia e la quantità delle risorse disponibili, il numero e le informazioni dei task attivi o terminati. In questo modo si possono aggiungere nuovi *slave* in maniera molto semplice

senza dover riconfigurare i nodi *master*, perché i nodi *slave* vengono riconosciuti all'avvio in automatico.

- Aggiornare sullo stato dei *task*. Ogni *task* invia delle informazioni allo *scheduler* che lo ha avviato.

Nella figura 1.3 vengono mostrate le connessioni che intercorrono fra *master* e *slave*.

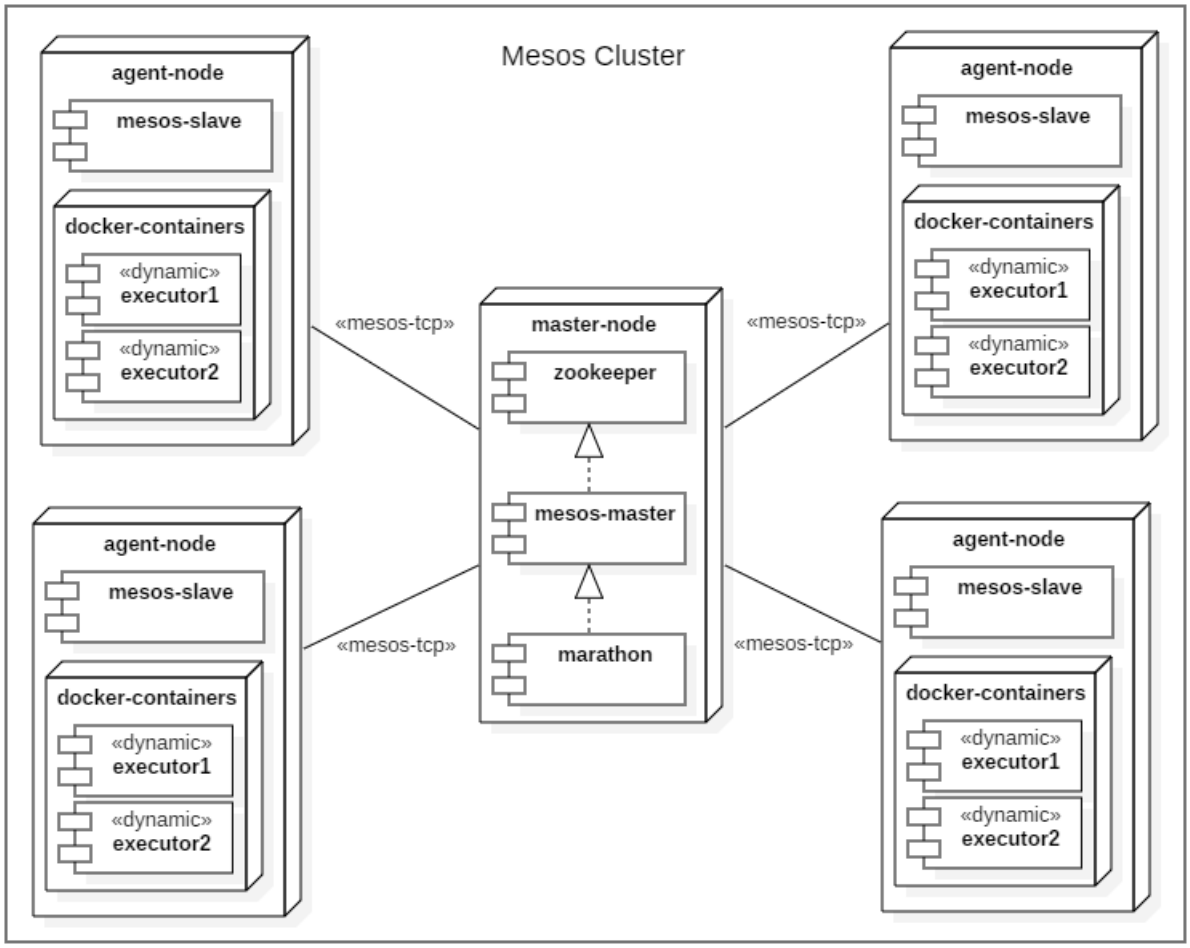

Figura 1.3: Diagramma di deployment in cui sono rappresentati un singolo nodo master e quattro nodi agent

Come è nato e chi lo utilizza

Mesos è stato ideato nel laboratorio della Berkeley AMP e l'obiettivo iniziale era quello di creare una piattaforma per la condivisione delle risorse del cluster fra più applicazioni[2]. Dopo la creazione diverse aziende, fra cui Twitter, si sono interessate al progetto e hanno contribuito al miglioramento di Mesos. Raggiunta una certa popolarità è stato inglobato da Apache e ora è uno dei più popolari software open source per la coordinazione delle macchine di un cluster. Tra le aziende più famose che usano Apache Mesos c'è sicuramente Apple, che lo utilizza per gestire la sua infrastruttura formata da decine di migliaia di macchine. Anche Cisco, eBay, Netflix, Paypal utilizzano Mesos per il coordinamento dei nodi dei loro cluster suddivisi anche in più datacenter sparsi per il pianeta.

1.3 Akka

Prima dell'esplosione del web e dei Big Data era normale per una applicazione funzionare utilizzando una sola macchina e una sola CPU. Quando l'applicazione non era veloce abbastanza o richiedeva ulteriori risorse si sostituiva il processore con uno più potente e si risolveva il problema. Ad un certo punto la velocità dei processori ha smesso di crescere esponenzialmente e per soddisfare tutte le richieste degli utenti le applicazioni hanno dovuto iniziare ad utilizzare metodi alternativi per risolvere questo problema. Sono così nati i primi sistemi distribuiti che permettono di sfruttare le risorse di più macchine con più processori in modo parallelo.

Definizioni e proprietà di un sistema distribuito

Un sistema distribuito per supportare le richieste di più utenti deve essere concorrente. La concorrenza è una proprietà che si riferisce all'abilità di parti differenti di un programma, di un algoritmo o di un problema di essere eseguite senza seguire una sequenza di operazioni, quindi fuori dall'ordine, senza influenzare il risultato. Se la proprietà di concorrenza viene rispettata le stesse parti di programma o di un algoritmo possono essere eseguiti su più macchine in modo parallelo, facendo diminuire il tempo di esecuzione se confrontato con lo stesso problema risolto utilizzando un singolo flusso di istruzioni.

La proprietà che misura l'efficienza di un sistema distribuito è la scalabilità e viene utilizzata per indicare i casi in cui un aumento di risorse per risolvere lo stesso problema provocano un impatto sulle performance. È possibile scalare un sistema in due modi: verticalmente (*scaling up*) e orizzontalmente (*scaling out*). Un sistema distribuito scala verticalmente quando vengono incrementate risorse sulla stessa macchina in cui è eseguita l'applicazione, ad esempio aumentando il numero di processori fisici o logici. Si parla di scalabilità orizzontale quando vengono aggiunte dinamicamente nuove risorse

indipendenti all'interno del cluster. Lo *scaling up* raggiunge velocemente un limite, mentre lo *scaling out* può essere infinito.

Un'applicazione realizzata tramite un'architettura distribuita deve idealmente aumentare le risorse in modo lineare quando aumentano le richieste degli utenti. Per progettare un sistema distribuito efficiente occorre avvicinarsi il più possibile alla proprietà di linearità, per far sì che all'aumentare delle risorse non si sprechino unità di calcolo. All'aumentare delle risorse è necessario mantenere costante anche la complessità del sistema. Un sistema che aumenta di complessità quando scala è difficile e costoso da mantenere; occorre quindi mantenere la complessità bassa per usare efficientemente le risorse del cluster.

In un sistema distribuito i componenti interagiscono fra loro. Due componenti che vogliono comunicare possono trovarsi entrambi sulla stessa macchina oppure su due macchine differenti. Se si trovano sulla stessa macchina è possibile utilizzare un metodo di comunicazione sincrono per permettere ai componenti di scambiarsi informazioni tramite comunicazione tra processi. Se due componenti si trovano su macchine differenti e si utilizza una comunicazione sincrona la latenza introdotta dalla comunicazione in rete produce un ritardo che fa sprecare risorse ai componenti che devono attendere delle risposte. È necessario quindi utilizzare una connessione asincrona fra i componenti che si trovano su macchine diverse.

Per non far aumentare la complessità del sistema è possibile unificare i metodi di comunicazione e utilizzare la comunicazione asincrona sia nel caso in cui due componenti si trovino sulla stessa macchina sia nel caso in cui si trovino su macchine diverse. Si introduce quindi un livello di astrazione che permette a più componenti di comunicare fra loro senza che sia necessario sapere dove essi siano collocati fisicamente e quindi senza dover utilizzare metodi di comunicazione diversi.

Cos'è Akka

Akka è una libreria scritta in Scala che fa parte di un progetto open source che permette di realizzare applicazioni distribuite tramite un'architettura che utilizza il modello di programmazione ad attori[3]. Con Akka è possibile utilizzare gli attori per progettare un sistema distribuito che scali sia verticalmente sia orizzontalmente utilizzando la stessa interfaccia. Permette di utilizzare le risorse efficientemente e l'architettura con cui è stata progettata permette di realizzare applicazioni che non aumentino di complessità quando scalano. Oltre ad implementare il modello ad attori, con Akka sono disponibili numerosi strumenti che permettono di agevolare operazioni comuni in un sistema distribuito.

Akka è stato progettato e sviluppato seguendo le idee del *Reactive Manifesto*, un'iniziativa che è stata ideata per stendere le basi della programmazione reattiva. Un sistema reattivo è un sistema robusto, elastico e flessibile. Il manifesto è incentrato nell'usare efficientemente le risorse e fornire l'opportunità alle applicazioni di scalare automaticamente. I punti cardine del manifesto sono i seguenti[2]:

- Le comunicazioni I/O bloccanti sono un limite per il parallelismo, quindi le comunicazioni I/O non bloccanti sono preferibili.
- Le interazioni sincrone fra i componenti limitano le opportunità per il parallelismo, le comunicazioni asincrone sono quindi preferibili.
- Effettuare il *polling* riduce le opportunità di usare meno risorse, un'architettura basata su eventi è preferibile.
- Un nodo che termina deve non può far fallire il sistema. È necessario isolare gli errori per non perdere informazioni e dati.
- Il sistema deve essere elastico. Se c'è meno richiesta deve essere possibile utilizzare meno risorse. Se c'è più richiesta deve essere possibile incrementare il numero di risorse senza superare la quantità richiesta.

[2]https://www.reactivemanifesto.org/

Come funziona

La maggior parte delle funzionalità di Akka sono incentrate sul modello ad attori. Gli attori sono dei componenti elementari che incapsulano una logica applicativa e che comunicano con altri attori tramite dei messaggi. Ogni attore può sia inviare messaggi che riceverli. I messaggi ricevuti vengono memorizzati temporaneamente in una coda di messaggi prima che vengano processati. Finché la coda di messaggi non è piena l'attore è abilitato alla ricezione, indipendentemente dal fatto che l'attore stia elaborando o meno un messaggio. I messaggi vengono memorizzati secondo l'ordine di arrivo ed elaborati uno per volta.

Un messaggio è una struttura dati che contiene informazioni e che non può essere cambiata dopo che è stata creata; è quindi immutabile. Un attore può scegliere dove processare i messaggi; può scegliere come unità di elaborazione un altro thread sulla stessa macchina o può scegliere di elaborare il messaggio in una macchina remota. Con Akka si possono inviare messaggi di qualunque tipo, a patto che siano immutabili. Akka quindi non effettua controlli sul tipo e sulla classe degli oggetti inviati come messaggio. Il *type checking* in un'architettura con modello ad attori è limitativo perché aggiunge costi e limita la flessibilità. Per scelta progettuale Akka ha deciso di non effettuare *type checking*.

Un attore può effettuare quattro operazioni: inviare messaggi, creare nuovi attori, cambiare stato e supervisionare altri attori. Ciascuna di queste operazioni deve essere eseguita in modo asincrono. Di seguito vengono analizzate le operazioni in dettaglio.

- **Inviare messaggi**. Gli attori possono interagire fra loro scambiandosi messaggi. Non è possibile per un attore estendere o utilizzare parte delle funzionalità di un altro attore. Akka non permette ad un componente esterno di modificare lo stato interno di un attore. Questa tipologia di architettura adottata obbliga ad incapsulare la logica all'interno degli attori per ridurre la complessità e le dipendenze del sistema.

La garanzia che i messaggi inviati ad altri attori arrivino non esiste: occorre quindi implementare meccanismi in autonomia nel caso si debba avere la certezza che un messaggio venga consegnato. Un modo semplice ed intuitivo per ovviare al problema è quello di utilizzare dei messaggi di risposta per confermare la presa in carico di un messaggio.

- **Creare nuovi attori**. Akka organizza gli attori in una gerarchia ad albero. Un attore può quindi creare altri attori che diventeranno i suoi figli. L'attore padre è responsabile dei suoi figli; se termina l'attore padre gli attori figli muoiono. Ciascun attore può scegliere autonomamente di terminare o può decidere di far terminare un qualsiasi altro attore[3]. Il primo attore non ha genitore e viene creato direttamente dall'`ActorSystem`.

- **Cambiare stato**. La logica interna di un attore può essere rappresentata tramite una macchina a stati finiti. Un attore può implementare quindi più stati. Le transizioni fra gli stati vengono effettuate dai messaggi inviati dagli altri attori. Cambiando di stato un attore reagisce quindi ai messaggi in maniera differente.

- **Supervisionare altri attori**. Ogni attore è responsabile degli attori che crea. Ogni attore padre può scegliere in che modo reagire quando i suoi attori figli ad esempio falliscono. L'attore padre può scegliere quindi una strategia che permette ad esempio di riavviare gli attori figli quando falliscono per una determinata causa.

Ogni processo dell'applicazione è supervisionato da un `ActorSystem`. Tramite questo oggetto è possibile effettuare la maggior parte delle operazioni. L'`ActorSystem` è incaricato di creare il primo attore, ovvero l'attore di primo livello che non ha genitore. È possibile creare più attori di primo livello, ma solitamente è sconsigliato. Tramite l'`ActorSystem` è possibile inizializzare

[3]Sono presenti dei meccanismi per impedire che questo avvenga in alcuni casi delicati

anche altri componenti ad esempio per permettere la comunicazione remota e per formare e gestire il cluster.

Quando il primo attore viene creato viene generato un indirizzo che permette di inviare messaggi a quel determinato attore. Ogni attore ha un indirizzo diverso. Gli indirizzi seguono la stessa gerarchia ad albero utilizzata per la creazione degli attori e indicano un percorso che è simile a quello utilizzato dagli URL. L'ultima parte del percorso indica il nome dell'attore. Il nome dell'attore è un parametro che può essere scelto durante la creazione; c'è un unico limite: due attori sullo stesso livello e con lo stesso genitore devono avere un nome diverso. Come gli URL, gli indirizzi degli attori possono essere relativi o assoluti. Con un indirizzo assoluto è possibile inviare un messaggio ad un attore che si trova in una macchina remota.

1.4 Apache Cassandra

Quando si lavora con i Big Data deve essere presente un sistema che permetta di memorizzare i dati in maniera strutturata. La maggior parte delle applicazioni moderne raccolgono enormi quantità di dati e incentrano il loro business nel cercare di catalogare più informazioni possibili. È quindi necessario un sistema distribuito che si faccia carico di tutte le richieste per memorizzare dati nel modo più veloce ed efficiente possibile. Cassandra è un database management system non relazionale che fa parte dei database NoSQL che si occupa proprio di questo.

Problemi e benefici dei database relazionali

In un sistema distribuito una delle proprietà fondamentali è la scalabilità. I database più diffusi e utilizzati fino a questo momento, ovvero i database relazionali, non rispettano appieno questa proprietà. Il loro problema principale sono le operazioni di *join*, che sono uno dei principi cardine dei database relazionali. Quando è presente una quantità elevata di dati le operazioni di *join* possono diventare lente. Inoltre se i dati da incrociare si trovano su macchine diverse effettuare un'operazione di *join* può essere un processo costoso.

Un altro problema riguarda le transazioni, che sono delle operazioni per garantire la consistenza dei dati quando devono essere effettuate più query che riguardano gli stessi dati. Le transazioni funzionano bloccando l'accesso dei client ai dati per la porzione di tempo che impiega la transazione ad essere eseguita. Questa condizione diventa inaccettabile quando si devono gestire un numero elevato di transazioni, perché si creerebbe una coda di richieste che aumenterebbe in modo spropositato la latenza, impedendo ai client di leggere e scrivere dati senza aspettare.

È possibile tentare di aggirare questi problemi ma non è possibile risolverli completamente senza cambiare strategia. Si possono migliorare le caratteristiche hardware, aumentando memoria e CPU, ma si arriva ad un limite. È possibile aggiungere nuove macchine, ma si creano problemi di replicazione e

consistenza dei dati. Si può cercare di migliorare gli indici e le query ma non si risolve il problema se la quantità di richieste aumenta esponenzialmente. Aggiungendo infine un livello di *caching* o duplicando i dati si possono creare problemi di consistenza e si violano i principi della progettazione degli schemi relazionali.

Cos'è Cassandra

Apache Cassandra è un sistema distribuito open source per memorizzare i dati. Cassandra fornisce numerose funzionalità ed è altamente personalizzabile. Fra le sue caratteristiche chiave ci sono la durabilità dei dati e la facilità di scalare. È un sistema ottimizzato sulla scrittura che è decentralizzato, sempre disponibile e permette ai client di interagire tramite un linguaggio di query creato apposta, chiamato Cassandra Query Language (CQL). L'architettura distribuita di Cassandra proviene da Amazon Dynamo, mentre il modo con cui è possibile interagire proviene da Google Bigtable.

Di seguito sono elencate le principali proprietà di Cassandra e per ogni proprietà sono spiegati pregi e difetti, paragonando Cassandra ad un normale database relazionale.

- **Nasce come un sistema distribuito**. Cassandra è progettato per essere eseguito su più macchine e per essere visto dai client come un'unica entità. Può essere eseguito in più macchine dello stesso cluster o in macchine distribuite geograficamente in più datacenter.

- **È un sistema decentralizzato**. A differenza dei principali database relazionali e non, Cassandra non si basa sull'architettura master/slave. Cassandra è interamente decentralizzato, ciò significa che ogni nodo del cluster è identico. Il vantaggio è che non è necessario configurare manualmente i nodi per stabilire il ruolo che devono avere ed è più facile manutenere il cluster. La coordinazione del cluster Cas-

sandra avviene tramite un protocollo peer-to-peer interno. Tramite la decentralizzazione è più facile ridondare i dati e distribuirli nel cluster.

- **Scala elasticamente**. Questa proprietà permette di aggiungere e rimuovere nodi del cluster senza causare perdite di dati e garantendo in qualsiasi momento la disponibilità dei dati. Il cluster è in grado di accettare nuove istanze che potranno partecipare ricevendo una copia parziale dei dati e iniziando a servire gli utenti in modo trasparente senza dover modificare la configurazione. Lo stesso è valido in caso contrario: è sempre possibile rimuovere nodi Cassandra dal cluster per diminuire la capacità totale; lo si può fare ad esempio per diminuire i costi nei periodi in cui è richiesta meno potenza computazionale.

- **È sempre disponibile**. Anche quando si aggiungono o rimuovono istanze dal cluster si ha sempre la garanzia che non ci siano momenti di down-time. Il cluster continua a servire le richieste degli utenti senza che sia possibile scoprire in alcun modo i cambiamenti interni.

- **È tollerante agli errori**. Se si imposta, come consigliato, una ridondanza dei dati, quando un nodo fallisce per un qualsiasi motivo, ad esempio per caduta della connessione o per altri problemi hardware o software, i dati non vengono persi e il cluster continua a servire le richieste. In automatico Cassandra copia i dati persi in un altro nodo in modo tale da ritornare allo stesso stato precedente.

- **Adotta un certo livello di consistenza**. La consistenza serve per indicare il fatto che alla stessa richiesta effettuata in qualsiasi momento debbano ritornare sempre gli ultimi dati scritti. Per garantire totale affidabilità Cassandra deve fare alcuni sacrifici sulla consistenza. Il livello di consistenza da garantire è un parametro modificabile; aumentandolo però si rischiano di perdere garanzie sull'affidabilità.

- **Orientato sulle righe**. A differenza di alcuni database Cassandra memorizza i record delle tabelle per riga e non per colonna. Le righe vengono salvate in tabelle hash multidimensionali. Non è presente il vincolo per cui tutte le righe all'interno della stessa tabella debbano necessariamente avere le stesse colonne. Ciascuna riga deve obbligatoriamente avere una chiave unica che viene utilizzata per distribuire le righe in più istanze Cassandra. La chiave utilizzata per stabilire dove deve essere memorizzata una riga si chiama chiave di partizione. Un'altra chiave può essere indicata per stabilire l'ordine in cui devono essere salvate più righe con la stessa chiave di partizione e si chiama chiave di *clustering*. È necessario che all'interno della stessa tabella il risultato della concatenazione fra chiave di partizione e chiave di *clustering* sia univoco.

- **Performante al massimo**. Cassandra deve essere in grado di sfruttare al massimo tutte le risorse a lei allocate. Anche sotto carichi eccezionali Cassandra garantisce tutte le proprietà elencate sopra e non spreca risorse per coordinare il cluster. La proprietà di scalabilità di Cassandra è lineare, cioè quando si raddoppiano le risorse hardware viene raddoppiata anche la potenza computazionale, senza che ci siano sprechi o *overhead*.

Come funziona

Ogni nodo Cassandra all'interno del cluster è indipendente e responsabile di una certa porzione del dataset. Quando un utente richiede di salvare o modificare un dato la prima operazione che viene eseguita viene definita *commit log*. I dati inviati tramite query vengono salvati su disco per fare in modo che non vengano persi attraverso una semplice e veloce operazione di scrittura in modalità di sola aggiunta. Subito dopo i dati vengono scritti in una `Memtable`, una struttura dati organizzata per righe ma che può contenere un numero elevato di colonne. I dati inizialmente vengono mantenuti in

memoria principale. Dato che Cassandra ha comunque i dati scritti nel file system può inviare al client una conferma della memorizzazione dei dati. Questo meccanismo permette una scrittura dei dati decisamente rapida che può sopportare un carico di lavoro elevato.

I record contenuti nella `Memtable` devono essere trasferiti su disco tramite un processo chiamato *flush*. I dati vengono scritti in una tabella chiamata `SSTable`, ovvero una tabella ordinata di stringhe, tramite un'operazione sequenziale di scrittura. A differenza di una scrittura casuale adottata dalla maggior parte dei database relazionali, la scrittura sequenziale permette di trasferire interi blocchi di memoria su disco in maniera efficiente e rapida, soprattutto se si utilizzano dischi magnetici come unità di memorizzazione. Per questo motivo è necessario memorizzare i dati secondo un ordine preciso che deve essere utilizzato anche in fase di lettura dei dati.

Per modificare record già presenti all'interno del cluster non si effettuano operazioni di sovrascrittura o di cancellazione. Quando si modifica una riga viene semplicemente aggiunto un nuovo record con la stessa chiave e con gli altri dati modificati. Ad ogni record è associato un timestamp che viene utilizzato per stabilire quale record sia il più recente. Quando si hanno più versioni dello stesso record, tramite lettura sequenziale i record duplicati vengono caricati in memoria e ordinati attraverso un'operazione di *merge sort*. I record ordinati vengono successivamente riscritti su disco tramite un'operazione di scrittura sequenziale. I vecchi record possono quindi essere eliminati.

I record vengono memorizzati in partizioni che possono risiedere anche in nodi Cassandra differenti. Ad ogni record è associata una chiave primaria. Tramite una funzione di hashing sulla chiave primaria viene stabilito in quale partizione salvare il record. La chiave primaria deve essere la più consistente possibile: calcolato l'hash i record si devono distribuire equamente tra le partizioni disponibili.

Per garantire che i dati non vengano persi è possibile impostare un fattore

di replicazione dei dati. I record devono essere replicati in partizioni diverse su nodi Cassandra diversi. Per supportare la replicazione dei dati vengono creati dei nodi virtuali. Ciascuna istanza Cassandra è responsabile della sua parte di dati che memorizza nella sua partizione principale; inoltre mantiene repliche dei dati di altre partizioni attraverso dei nodi virtuali.

Inizialmente quando un client deve inserire dei dati si collega ad un nodo qualsiasi del cluster ed effettua la richiesta. Cassandra si preoccupa poi di ridondare i dati in autonomia copiando i dati appena inseriti in altri nodi del cluster. Se il client effettua una richiesta per richiedere i dati appena inseriti in un nodo diverso da quello utilizzato per la scrittura nel periodo di tempo fra la prima scrittura e la copia dei dati per la replicazione si possono ottenere dei dati non aggiornati. È possibile limitare o risolvere questo problema modificando il livello di consistenza. Questo livello si imposta in base al numero di conferme che il client deve ottenere quando effettua un'operazione di scrittura. Se si impostano meno conferme le operazioni di scrittura sono più rapide ma ci possono essere problemi di consistenza, mentre se si decide che bisogna aspettare tutte le conferme da parte di tutti i nodi che devono contenere la replica dei dati la latenza aumenta e il client deve aspettare più tempo per ricevere la conferma, ma la proprietà di consistenza è garantita.

Chi lo utilizza e come è nato

Il progetto è nato all'interno di Facebook per risolvere il problema della ricerca dei dati. La compagnia aveva grandi volumi di dati che difficilmente erano gestibili da un sistema tradizionale[4]. Cassandra è nato con l'intenzione di creare un sistema che fosse scalabile per gestire una quantità sempre crescente di dati. Vista l'importanza che stava ottenendo il progetto Facebook ha deciso di renderlo open source pubblicando i sorgenti. Poco dopo il progetto è stato inglobato dall'incubatore Apache ed è diventato presto un progetto di primo livello con una community di sviluppatori e collaboratori sempre più vasta. Ad oggi è utilizzato dalle più grandi aziende del pianeta in ambienti di produzione, come Facebook, Twitter e Netflix.

1.5 Apache Kafka

Ogni applicazione moderna deve essere in grado di gestire grossi volumi di dati in tempo reale. I dati generati sono la parte più importate di un'applicazione, perché da essi viene estrapolata la conoscenza. I dati possono essere di diversi tipi, ad esempio dei messaggi di log, di metrica o le attività degli utenti. Qualsiasi byte generato può contenere informazioni. Ad esempio le visite degli oggetti in un negozio virtuale possono essere trasformate in raccomandazioni. Più velocemente si effettuano queste operazioni maggiore è il vantaggio che possiamo trarne.

Cos'è Kafka

Kafka è un software open source che ha come scopo principale quello di effettuare lo streaming dei dati. Viene presentato come un *messaging system* ma con tre differenze sostanziali. A differenza di altri sistemi di messaggistica che implementano solamente code di messaggi, come ActiveMQ o RabbitMQ, Apache Kafka:

- è progettato per essere un sistema distribuito e può scalare orizzontalmente per gestire le richieste di qualsiasi tipo di applicazione;

- è un vero sistema di memorizzazione: i dati vengono scritti su disco e cancellati dopo un periodo di tempo. Inoltre i dati possono essere replicati per poterli recuperare in caso di problemi;

- il livello di astrazione che si può ottenere attraverso lo streaming dei dati è massimo.

Kafka può essere visto anche come una versione in real-time di Hadoop. Hadoop viene utilizzato per memorizzare i dati all'interno del cluster che verranno poi processati in modo periodico. Kafka permette di processare dati in tempo reale, appena essi vengono generati.

Kafka utilizza il pattern *puslish/subscribe* e i dati che vengono scambiati tra chi pubblica i dati, i produttori, e chi li riceve, i consumatori, vengono chiamati messaggi. I produttori possono scegliere di pubblicare soltanto una tipologia di messaggio; lo stesso vale per i consumatori quando effettuano l'operazione di sottoscrizione.

Come altri sistemi che implementano il pattern *publish/subscribe* anche Kafka utilizza una coda di messaggi per la memorizzazione e la comunicazione fra processi per lo scambio di messaggi. A differenza di altri sistemi però Kafka può essere visto come un sistema centralizzato singolo che permette la pubblicazione di messaggi generici. I dati all'interno di Kafka sono memorizzati nell'ordine in cui arrivano ed è garantita la presenza grazie a sistemi di protezione contro gli errori.

Come funziona

L'unità minima all'interno di Kafka è il messaggio, che viene visto dal sistema come un semplice array di byte che non ha alcun significato. Ad ogni messaggio può essere associata una chiave, anch'essa priva di significato per Kafka e che consiste in un array di byte. La chiave può essere utilizzata per determinare in quale partizione deve essere pubblicato un messaggio attraverso una funzione di hashing. Se la funzione di hashing utilizzata è consistente, cioè per ogni chiave viene sempre generato lo stesso hash, due messaggi con la stessa chiave verranno sempre pubblicati nella stessa partizione.

Per la pubblicazione di dati in maniera efficiente è possibile utilizzare operazioni batch. Le operazioni batch riguardano un insieme di messaggi che devono far parte dello stesso *topic* e della stessa partizione. Questo sistema viene utilizzato per ridurre la latenza e il volume di traffico generato nella rete. Se i messaggi vengono raggruppati e trasmessi insieme invece che pubblicati singolarmente, vengono risparmiate diverse richieste. I batch possono essere compressi per ridurre ulteriormente il volume di dati da scambiare in rete.

Seppur i messaggi non sono altro che semplici array di byte è consigliato aggiungere una struttura, ovvero uno schema. Per la costruzione di uno schema è possibile utilizzare diversi linguaggi tra cui il JSON o l'XML, che sono facili da usare e da leggere ma non sono comprimibili. È necessario tenere in considerazione anche altri problemi, come il problema dell'evoluzione dello schema. Per non creare problemi di compatibilità fra schemi differenti esistono software che permettono di gestire l'evoluzione dello schema, come Apache Avro.

I messaggi sono suddivisi in *topic*, che a loro volta sono suddivisi in partizioni. Un messaggio generalmente è scritto in una sola partizione all'interno di un *topic*. Un *topic* può avere più partizioni, e in ciascuna partizione è garantito l'ordine di arrivo dei messaggi. I messaggi all'interno di una partizione vengono letti dal più vecchio al più recente, mentre fra più partizioni dello stesso *topic* possono non essere rispettati i vincoli temporali. Le partizioni vengono utilizzate da Kafka per garantire ridondanza e scalabilità. Ciascuna partizione può essere situata in server diversi, ciò significa che un singolo *topic* può essere distribuito in diversi nodi del cluster per fornire una scalabilità orizzontale.

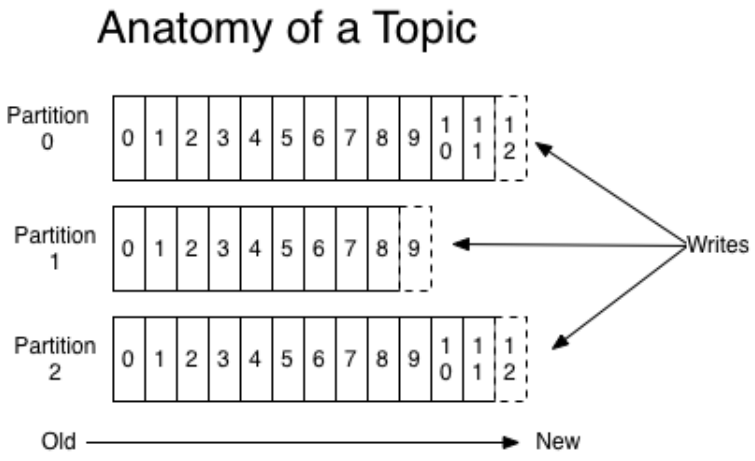

Figura 1.4: Anatomia di un topic di Apache Kafka [Apache Kafka official documentation]

I client del sistema sono i produttori e i consumatori. I produttori producono nuovi messaggi di uno specifico *topic*. Solitamente il produttore non si preoccupa di scegliere quale partizione utilizzare, ma i messaggi vengono distribuiti equamente fra le partizioni del *topic*. Il consumatore si occupa di sottoscrive uno o più *topic* e leggere i messaggi nell'ordine in cui sono prodotti. Per tenere traccia dei messaggi letti da un consumatore ad ogni messaggio viene aggiunto un *offset*, ovvero un intero auto-incrementante che è univoco all'interno della partizione.

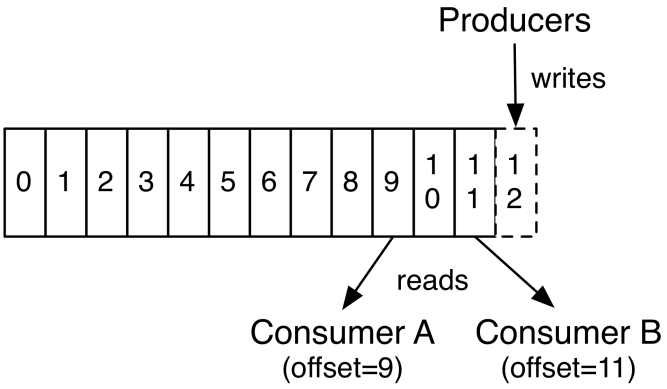

Figura 1.5: Produttori e consumatori di una partizione [Apache Kafka official documentation]

L'*offset* può essere memorizzato in due modi:

- **Esternamente**. L'applicazione deve implementare una logica per mantenere l'*offset* per tutte le partizioni dei *topic* sottoscritti. La memorizzazione dell'*offset* deve essere permanente per non perdere il punto in cui si è arrivati ad elaborare i dati in caso di fallimento del contenitore degli *offset*.

- **Internamente**. In questo caso l'*offset* di ciascuna partizione è salvato all'interno di Zookeeper o di Kafka, quindi il consumatore può fermarsi o riavviarsi senza che l'offset venga perso.

I consumatori devono far parte di un gruppo di consumatori; identificato da una stringa, in un gruppo di consumatori possono operare più consumatori che sottoscrivono lo stesso *topic*. Il gruppo assicura che ciascuna partizione sia letta da solo un membro. In questo modo i consumatori possono essere scalati orizzontalmente per consumare *topic* con un numero elevato di messaggi. Quando un consumatore fallisce le partizioni del *topic* vengono ribilanciate e distribuite agli altri membri del gruppo.

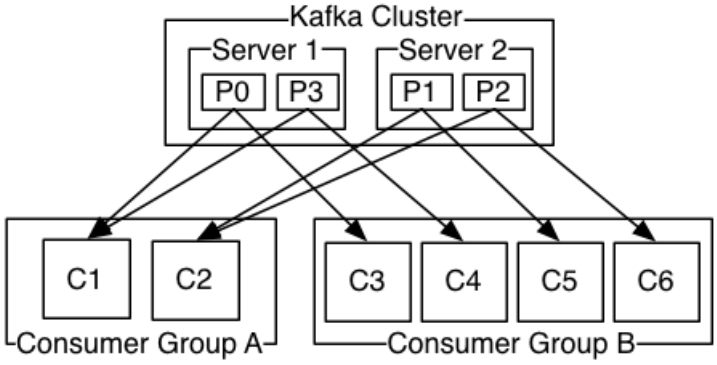

Figura 1.6: Gruppi di consumatori per delle partizioni di uno stesso topic [Apache Kafka official documentation]

Kafka può essere installato su più server; ciascuna istanza è chiamata *broker*. Il *broker* si occupa di ricevere i messaggi dai produttori, assegnare loro un *offset* e salvare il messaggio coi metadati sul disco. Inoltre serve i consumatori rispondendo alle richieste di lettura. I *broker* di Kafka avviati su server diversi con l'ausilio di Apache Zookeeper formeranno un cluster. All'interno del cluster ci sarà un *broker* che avrà la funzione di controllore. Il controllore, eletto fra i *broker* disponibili all'interno del cluster, ha la responsabilità di amministrazione; si occupa ad esempio di assegnare le partizioni ai *broker* quando un *topic* ne ha più di una e monitorare gli errori degli altri *broker*. Anche le partizioni sono controllate da un *leader*, un *broker* all'interno del cluster. Una partizione può essere assegnata a più *broker*; questo succede quando si imposta un fattore di replicazione per garantire la ridondanza dei

messaggi. Se ci sono più copie di una stessa partizione e il *leader* della partizione fallisce, viene eletto un nuovo *leader* fra i *broker* rimanenti.

A ciascun *topic* può essere impostato un tempo di ritenzione che indica un periodo di tempo durante il quale devono essere mantenuti i messaggi prodotti. In sostituzione o in aggiunta alla durata di ritenzione si può impostare la soglia massima di peso di un determinato *topic*. Scaduto il tempo di ritenzione o superata la soglia massima di peso, Kafka si occupa di cancellare i messaggi dal disco e renderli inaccessibili dai consumatori.

Per applicazioni che devono essere distribuite in diverse aree geografiche per requisiti di isolamento di sicurezza o per fare il recovery di un intero cluster Kafka permette di organizzare i *broker* in più datacenter. Viene fornito uno strumento, chiamato MirrorMaker, che viene utilizzato per questo scopo: è un client di Kafka che funziona contemporaneamente come produttore e consumatore e serve per copiare i messaggi fra cluster di diversi datacenter.

Prestazioni e alternative

Kafka ha numerosi punti di forza che lo rendono uno dei software più utili per effettuare lo streaming dei dati. Esistono altre soluzioni open source per lo streaming e il processamento dei dati, come Apache Flume, Apache Flink e Apache Samza, ma sono meno conosciute e utilizzate. Apache Kafka ha il vantaggio di:

- avere la possibilità di gestire più produttori, che lo rende un sistema ideale per aggregare dati da più sistemi;

- avere la possibilità di gestire più consumatori contemporaneamente, cosa che molti sistemi di streaming di dati non permettono;

- salvare i dati sul disco per un tempo e con criteri prestabiliti, in modo tale da permettere ai produttori e ai consumatori di stopparsi e riavviarsi senza perdere i dati;

- scalare orizzontalmente in modo lineare ed istantaneo. È possibile formare il cluster con uno o pochi *broker* fino ad arrivare a migliaia di nodi, scalando senza interruzioni;

- mantenere alte prestazioni anche se i *broker* sono sotto carico, avendo tempi di latenza inferiori al secondo tra la produzione di un messaggio e la sua consumazione. In base alle caratteristiche hardware e alle performance della macchina, un singolo *broker* può facilmente gestire migliaia di partizioni e milioni di messaggi al secondo.

Chi lo utilizza e come è nato

Kafka è utilizzato da decine di migliaia di aziende, almeno un terzo di quelle indicate nel Fortune 500[5]. È uno dei progetti open source che è cresciuto più rapidamente negli ultimi anni e che ha creato un immenso ecosistema attorno. È diventato il cuore dei progetti che gestiscono e processano stream di dati. Kafka è un progetto nato all'interno di LinkedIn per gestire flussi continui di dati. Il prototipo di sistema utilizzato inizialmente da LinkedIn era formato da servizi ActiveMQ che al tempo erano difficilmente scalabili. L'azienda ha quindi deciso di formare un team per creare un software che utilizzasse il modello *push-pull* attraverso il pattern *consumer/producer* per fornire la persistenza dei messaggi, per garantire affidabilità e consumatori multipli, per ottimizzare il *throughput* dei dati e che avesse la capacità di scalare orizzontalmente al momento del bisogno e quando i dati e le connessioni crescevano. È nato così Kafka che è stato successivamente inglobato dalla fondazione Apache.

Capitolo 2

Analisi dei requisiti

2.1 Requisiti

Lo scopo del progetto è quello di creare una solida base di partenza su cui sarà possibile progettare e implementare applicazioni che lavorano utilizzando lo stack SMACK. Si vuole realizzare quindi un framework che non contiene logica applicativa, ma che permette di costruire applicazioni distribuite senza che sia necessario utilizzare le API dei componenti dello stack e quindi senza conoscere il loro funzionamento.

L'architettura utilizzata dal framework dovrà essere di tipo client-server, mentre per l'interazione fra l'utente e l'applicazione dovrà essere utilizzato il protocollo HTTP. Il framework dovrà permettere di realizzare applicazioni *web-based*. Sarà quindi possibile modellare un qualsiasi dominio reale che sia applicabile ad una logica client-server. Sarà necessario utilizzare durante la fase di progettazione un livello di astrazione e di generalizzazione tale da poter permettere a chi utilizzerà il framework di implementare anche logiche applicative complesse. Nel contempo la struttura base del framework dovrà essere il più semplice possibile, in modo tale da poter permettere di realizzare semplici applicazioni senza dover applicare particolari tecniche di progettazione; il framework dovrà quindi fornire linee guida da utilizzare per agevolare la progettazione senza che siano però in alcun modo vincolanti.

Il framework è destinato a diverse tipologie di applicazioni. Potrebbe essere utilizzato per progettare sistemi nell'ambito dell'Internet of Things, dove i client sono i singoli sensori e gli attuatori presenti negli oggetti intelligenti e il server è l'entità centrale, ma distribuita, che elabora i dati ricevuti e trasmette eventualmente informazioni per azionare sistemi automatici. Può essere utilizzato per implementare un social network. Data la vasta quantità di dati che devono essere gestiti, soltanto un sistema distribuito può farsi carico della moltitudine delle richieste degli utenti.

Attraverso il framework dovrà essere possibile realizzare un'applicazione che potrà essere lanciata all'interno del cluster Mesos. Apache Mesos alloca le risorse per eseguire l'applicazione in modo dinamico, quindi le informazioni dell'ambiente e della macchina dove l'applicazione è lanciata sono disponibili solamente a *runtime*. Questo è un problema comune di tutti i sistemi distribuiti che deve essere studiato e analizzato attentamente perché è uno dei punti cruciali.

Requisiti funzionali

- Il framework deve permettere di implementare logiche applicative complesse, ma la struttura del software deve essere semplice e schematica.

- Le API dei software che sono alla base dello stack, come Apache Kafka e Apache Cassandra, devono essere decorate dal framework per permettere un utilizzo a più alto livello delle stesse.

- Le richieste devono poter essere processate in due modi, in real-time e in modalità batch. In entrambi i casi il framework deve assicurare alcune garanzie, come la sicurezza di non perdere i dati in caso di errore.

- Il framework deve utilizzare il modello delle API REST per l'interazione client-server, fondamentale quando si devono progettare sistemi distribuiti per la mancanza del concetto di sessione.

- Deve essere presente un sistema per l'esecuzione di operazioni, definite come job, attraverso uno schedulatore di processi che pianifica le operazioni nel tempo. I job devono essere eseguiti in modo distribuito all'interno del cluster tramite il motore Apache Spark.

Requisiti non funzionali

- Il framework deve essere progettato per rispettare il più possibile la proprietà di modularità. Data la complessità del sistema, questa proprietà diventa una delle proprietà da tenere più in considerazione durante la progettazione. Inoltre ogni parte del sistema deve essere il più possibile indipendente dalle altre; è necessario quindi mantenere un alto livello di incapsulamento.

- Permettendo di sviluppare un'applicazione distribuita, la proprietà di scalabilità è fondamentale. Un sistema perfetto dovrebbe garantire una scalabilità lineare, il framework deve cercare di avvicinarsi il più possibile a questa caratteristica.

- Il sistema dovrà necessariamente essere *fault tolerance*. Deve essere tenuto il considerazione quindi che gli errori possono capitare, ma deve essere trovata una soluzione per reagire in caso di fallimento per isolare dal sistema le parti problematiche e continuare a servire le richieste normalmente.

- Durante la fase di progetto è necessario tenere in considerazione la sicurezza. Il sistema è composto da più componenti che interagiscono fra loro attraverso la rete; è necessario che gli accorgimenti sulla sicurezza debbano essere presi in considerazione fin da subito.

- Il framework deve garantire un'alta affidabilità. Costruendo un sistema con una minima dipendenza fra i componenti in caso di problemi

generalizzati o manutenzione deve essere possibile isolare la parte non funzionante e garantire che il resto del sistema non affetto da guasti funzioni correttamente.

- Essendo un sistema distribuito la quantità di dati che transiteranno all'interno del cluster sarà elevata. Sarà necessario utilizzare sistemi di serializzazione degli oggetti, dei messaggi e degli eventi che siano i più efficienti possibili. Per serializzazione efficiente si intende che i dati devono essere serializzati e deserializzati velocemente senza impiegare per troppo tempo la CPU. Al tempo stesso devono essere utilizzate tecniche di compressione che permettono di ridurre il traffico di rete all'interno del cluster o tra più datacenter.

- Bisogna riuscire a mantenere bassi i tempi di processamento delle richieste da effettuare in real-time. Quando non è necessario aspettare l'elaborazione dei dati, come nel caso dei sensori che raccolgono informazioni, il sistema deve provvedere alla memorizzazione temporanea dei dati per poi elaborarli in un secondo momento, in modo da ridurre al minimo le risorse utilizzate dai sistemi di ricezione.

2.2 Analisi e modello del dominio

Il framework deve poter permettere di creare un'applicazione distribuita *web-based* che abbia un'architettura client-server. Il sistema deve quindi essere composto da un componente che raccoglie le richieste dei client, definito come *frontend* e un componente che elabora le richieste, chiamato *backend*. Essendo un sistema distribuito è necessario aggiungere un ulteriore componente per permettere il coordinamento delle istanze attive che compongono il sistema, che viene definito *seed*. Infine un componente chiamato *service* deve essere utilizzato per analizzare le richieste non processate in real-time.

Suddividendo il framework in più parti indipendenti si riduce la complessità del sistema, perché ogni parte può essere progettata, sviluppata e testata singolarmente. Ogni parte si occupa di una funzionalità diversa che deve essere autonoma. Suddividendo fin dalla fase di analisi il problema in più parti sarà più facile poi progettare il sistema distribuito. Le parti saranno definite come ruoli. Per funzionare il sistema ha bisogno che ciascun ruolo sia attivo e funzionante. Nella sezione di seguito è spiegato in dettaglio il funzionamento di ogni ruolo.

Ruoli del sistema

Backend

È il ruolo che si occupa di processare le richieste degli utenti raccolte dal frontend per trasformarle in risposte. Le richieste ricevute vengono inviate dal frontend sotto forma di messaggi che contengono il tipo di azione da compiere e il contenuto della richiesta. Il backend è dotato di un supervisore che intercetta il tipo di azione da eseguire riconoscendo la classe dell'oggetto che contiene il messaggio, e lo inoltra al controllore che deve gestire quel tipo di azione. Il controllore ha il compito di verificare innanzitutto se la richiesta è valida. Se il client effettua una richiesta sbagliata un messaggio di errore deve essere inviato al frontend che si occupa di rispondere al client che ha effettuato la richiesta. Per le richieste valide, il controllore può scegliere fra le seguenti opzioni:

- gestire localmente la richiesta ed effettuare la computazione necessaria, rispondendo immediatamente al frontend con i risultati;

- inviare un nuovo evento ad un broker Kafka se non è necessario processare la richiesta subito e notificare il frontend della presa in carico;

- salvare in modo permanente i dati ricevuti nel cluster Cassandra e fornire al frontend i riferimenti della transazione.

Questo ruolo si deve occupare anche dell'autenticazione degli utenti nel caso l'utente accedesse ad una risorsa protetta. Questo è dovuto al fatto che il frontend, il ruolo che si occupa di raccogliere le richieste dei client, non deve poter accedere al database per una questione di sicurezza, dato che è l'unico componente che deve essere raggiungibile dall'esterno del cluster.

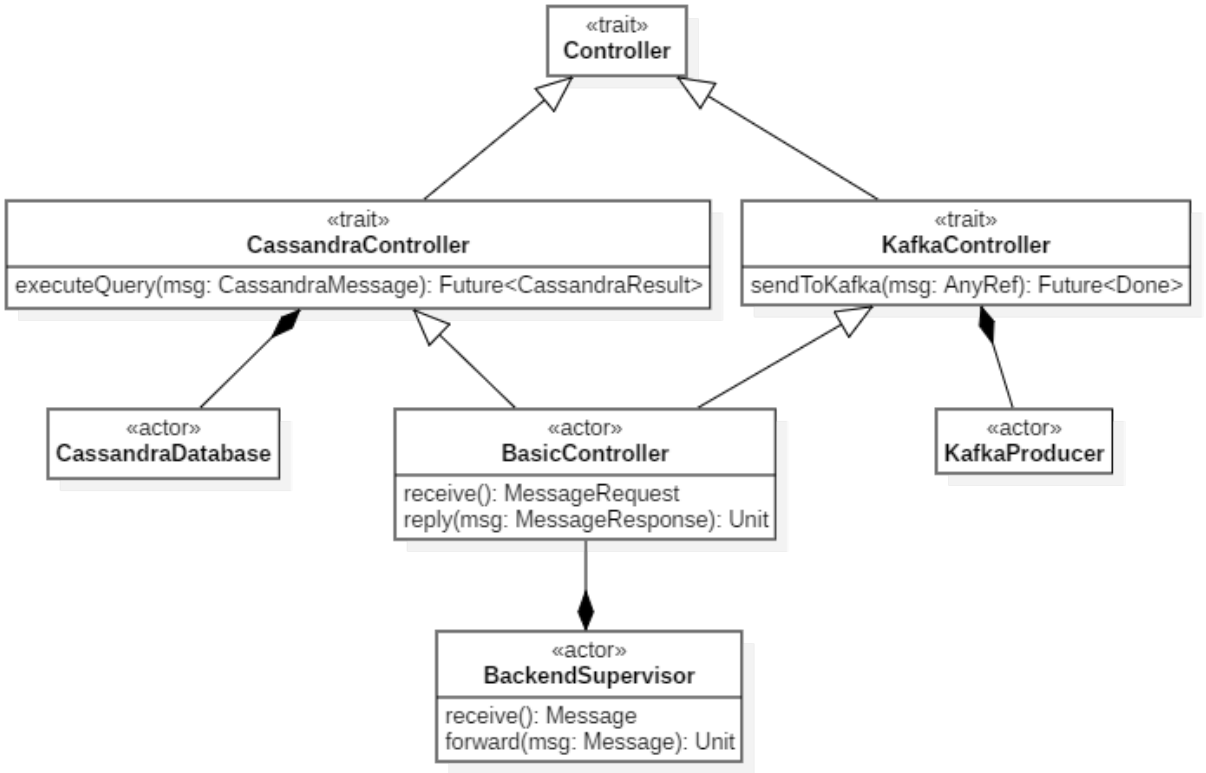

Figura 2.1: Diagramma delle classi che mostra l'analisi del modello del ruolo Backend

Frontend

Deve avviare un'istanza di un web server che si occupa di raccogliere le richieste dei client e di fornire loro una risposta. Il web server può essere esposto direttamente all'esterno del cluster o può essere collegato ad un *load balancer*

hardware o software. Un esempio di *load balancer* software è HAProxy[1]. Il frontend non gestisce direttamente le richieste ma le inoltra al backend. L'interazione del sistema con gli utenti deve essere realizzata tramite il modello delle API REST. Seguendo le specifiche REST[2] gli oggetti dell'applicazione dovranno essere modellati come risorse. Ogni risorsa deve dichiarare un percorso (*route*) e indicare quale sia il servizio che dovrà processare la richiesta. Una *route* dovrà contenere le specifiche della risorsa, come ad esempio l'URL che deve essere utilizzato per raggiungere tale risorsa, e le modalità di richiesta, ovvero quali azioni si devono compiere. Le operazioni disponibili devono essere quelle indicate da CRUD, ovvero la possibilità di creare una risorsa, selezionarla, aggiornarla o rimuoverla. Il servizio che si occupa di processare la richiesta deve essere un'istanza di un backend. Alla ricezione di una richiesta il web server seleziona la *route* corrispondente alla risorsa voluta: se esiste la richiesta deve essere processata, altrimenti deve essere generata una risposta di errore. Nel frontend deve essere presente una logica per la validazione delle richieste prima che esse vengano inoltrate al backend, in modo tale da scartare immediatamente le richieste non valide.

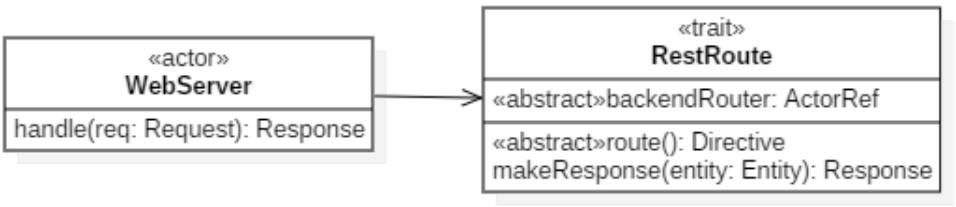

Figura 2.2: Diagramma delle classi che mostra l'analisi del modello del ruolo Frontend

Service

È il servizio che si occupa di processare le richieste che non vengono immediatamente processate dal backend. Serve ad esempio per consumare eventi

[1]http://www.haproxy.org/
[2]http://jsonapi.org/format/

dai broker Kafka, processarli e salvarli in modo permanente nel cluster Cassandra. Deve essere utilizzato quindi per elaborare dati che richiedono più risorse computazionali o che richiedono più tempo per essere processati, oppure nei casi in cui non si voglia far attendere il client quando si elabora una certa risorsa. Il ruolo service deve essere dotato di un supervisore che si occupa di monitorare lo stato dei servizi attivi. Quando in un servizio succede un errore imprevisto, il supervisore ha il compito di riavviare il servizio e di ristabilire le connessioni con i servizi esterni.

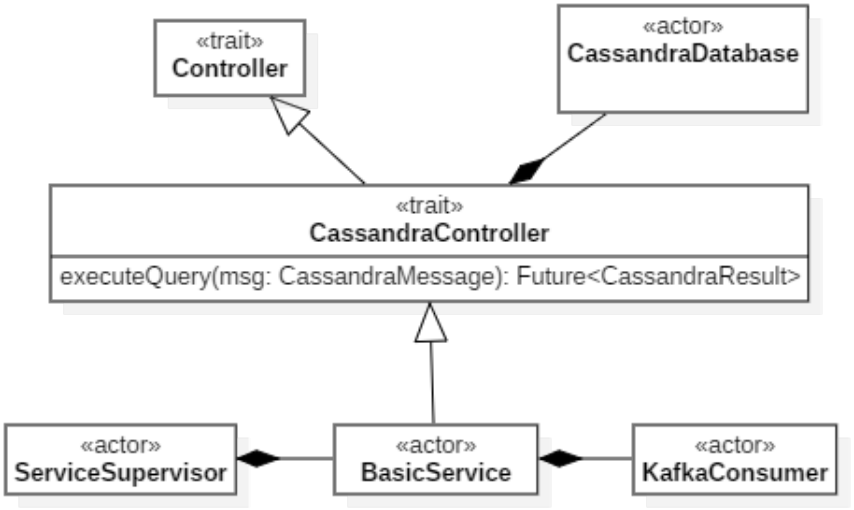

Figura 2.3: Diagramma delle classi che mostra l'analisi del modello del ruolo Service

Seed

È il ruolo che si occupa di coordinare i nodi del cluster. È fondamentale per la formazione del cluster e per il congiungimento dei nuovi nodi con i nodi già esistenti. È ideato per essere utilizzato come nodo statico all'interno del cluster in modo tale da essere raggiunto dagli altri ruoli sempre allo stesso indirizzo.

Analisi dei dati con Apache Spark

Il framework deve poter permettere di analizzare i dati salvati nel cluster di Apache Cassandra. Il motore utilizzato per eseguire le operazioni in modo distribuito è Apache Spark.

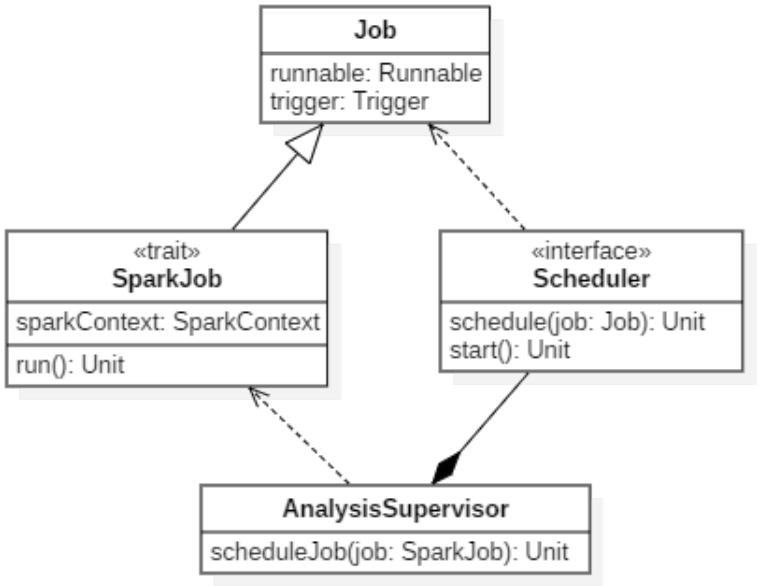

Figura 2.4: Diagramma delle classi che mostra le principali interfacce del modulo di analisi

Occorre progettare ed implementare un sistema per incapsulare unità di lavoro, che verranno definite come job, per eseguirle in intervalli di tempo regolari o in orari precisi. È necessario quindi un pianificatore di attività, definito *scheduler*, che si preoccupa di eseguire i job quando viene scatenato un evento, o *trigger*, che decide il momento di esecuzione di ogni job. Lo *scheduler* deve essere contenuto in un supervisore che si deve preoccupare anche di lanciare il driver di Spark, il componente del motore di analisi che si preoccupa di suddividere i job da eseguire in *steps*, ovvero unità di lavoro elementari che verranno inviate in maniera distribuita agli esecutori Spark. Questo sistema deve poter essere utilizzato dall'applicazione che estende il

framework per effettuare semplici analisi sulle tabelle del database, ma che richiedono più risorse computazionali, come il raggruppamento di dati in una tabella per una colonna che non è la chiave di partizione. Deve essere possibile implementare anche strumenti di analisi più complessa come ad esempio il *machine learning*.

Capitolo 3

Design dell'architettura

3.1 Modello ad attori

L'architettura utilizzata per progettare il framework segue il modello ad attori, un modello matematico ideato per sistemi concorrenti che utilizza gli attori per la comunicazione e lo scambio di informazioni tra thread o processi situati sulla stessa macchina o su macchine differenti. L'astrazione del modello ad attori permette di focalizzarsi sulla comunicazione che è l'aspetto più importante dei sistemi distribuiti.

Il modello ad attori adotta la filosofia che qualsiasi cosa è un attore. Un attore è un'unità indipendente che incapsula una propria logica e che comunica con gli altri attori, indipendentemente da dove si trovino, tramite messaggi asincroni. Il vantaggio principale di questo approccio è quello di forzare un utilizzo intensivo dell'incapsulamento che è una proprietà chiave nei sistemi distribuiti. Questo è necessario per ridurre la complessità del sistema. Ogni attore ha una sua logica ed è responsabile di una funzionalità del sistema, questo permette di scomporre il problema originale in più sotto-problemi ed assegnare ad ogni attore un problema più semplice da risolvere.

Il modello ad attori si differenzia dalle principali architetture tradizioni dal tipo di comunicazione che utilizza. In un sistema tradizione che utilizza un paradigma di programmazione orientato ad oggetti l'interazione fra i com-

ponenti viene effettuata tramite la chiamata a metodo. Ogni metodo ha un valore di ritorno che può essere nullo, ma che deve obbligatoriamente essere presente. Il difetto di questo approccio risiede nel fatto che l'operazione di chiamata a metodo deve aspettare un valore di ritorno, ma questa procedura è bloccante. Per bloccante si intende che il componente che ha chiamato il metodo deve bloccarsi ed attendere fino a che non viene ritornato un valore. Questo può essere problematico in un sistema distribuito in cui i componenti del sistema possono essere situati anche su macchine diverse e le chiamate a metodo devono essere quindi effettuate attraverso la rete. Oltre a dover aspettare il tempo di processamento vanno quindi aggiunti i tempi di latenza della rete, che provocano un *overhead* tale da rendere il sistema distribuito non efficiente.

Per ovviare a questo problema il modello ad attori rivoluziona il sistema di comunicazione. Invece di utilizzare chiamate a metodo per interagire con componenti esterni vengono utilizzati messaggi asincroni. Un messaggio non è altro che un vettore di byte che contiene informazioni che devono essere trasmesse fra componenti diverse. Un attore utilizza i messaggi per delegare del lavoro ad un altro attore. Rispetto ad una architettura basata sulla chiamata di metodo, il componente che deve spedire o richiedere informazioni ad un componente esterno non deve aspettare la risposta e quindi interrompere l'esecuzione. Se il messaggio viene perso ad esempio per un problema di rete non è necessario implementare tecniche di ripristino della connessione ma è sufficiente ritrasmettere il messaggio. Questo riduce drasticamente la complessità del sistema e permette di costruire applicazioni *fault-tolerance* in maniera gratuita. Il valore di ritorno può essere incapsulato in un messaggio e trasmesso al componente che ha richiesto informazioni o che necessita di una conferma.

Ogni attore processa i messaggi reagendo in base al suo stato interno. Gli attori possono quindi essere rappresentati da diagrammi a stati finiti, dove le transizioni da uno stato all'altro sono rappresentate dai messaggi. I messaggi vengono processati da ogni attore sequenzialmente secondo l'ordine di arrivo,

ma la ricezione dei messaggi avviene in modo concorrente. Il vantaggio nell'utilizzo di questo approccio risiede nel fatto che quando si progetta la logica di un attore non si devono tenere in considerazione problemi di concorrenza. L'unico vincolo che è necessario introdurre è che i messaggi devono essere oggetti immutabili: non deve essere possibile inviare messaggi che possono essere modificati dopo l'invio.

L'architettura ad attori permette quindi a più attori di lavorare parallelamente senza che sia necessario prendere precauzioni per evitare problemi di concorrenza, come problemi di *deadlock* o di *starvation*.

La struttura dell'intero framework si basa su questo modello ed implementa questa architettura.

3.2 Modello REST

L'utente interagisce col sistema attraverso il modello delle API REST (Representational State Transfer). REST è uno stile architetturale che definisce una serie di specifiche che devono essere utilizzate quando si crea un servizio web. REST è quindi un protocollo che permette a due sistemi di dialogare attraverso la rete, utilizzando come protocollo di comunicazione HTTP. REST è un protocollo *stateless*, che significa che nessuna delle due parti, ovvero il mittente e il ricevitore, mantengono salvate informazioni sullo stato di connessione.

L'architettura REST si basa sul concetto di richiesta e risposta, che non sono altro che semplici messaggi HTTP che possono contenere dati (*payload*) da trasferire al sistema con cui si sta dialogando. Solitamente il *payload* è formattato utilizzando un linguaggio come XML o JSON e segue uno schema prestabilito. Il framework dovrà implementare l'architettura REST e il *payload* dovrà essere formato da oggetti JSON. Il modello delle API REST permette quattro tipi di operazioni:

- Creare una risorsa attraverso il metodo HTTP POST.

- Leggere una risorsa attraverso il metodo HTTP GET.
- Aggiornare una risorsa attraverso il metodo HTTP PUT.
- Rimuovere una risorsa attraverso il metodo HTTP DELETE.

Nel diagramma dei casi d'uso mostrato in figura 3.1 sono riassunte le operazioni che un utente può effettuare quando interagisce con il sistema.

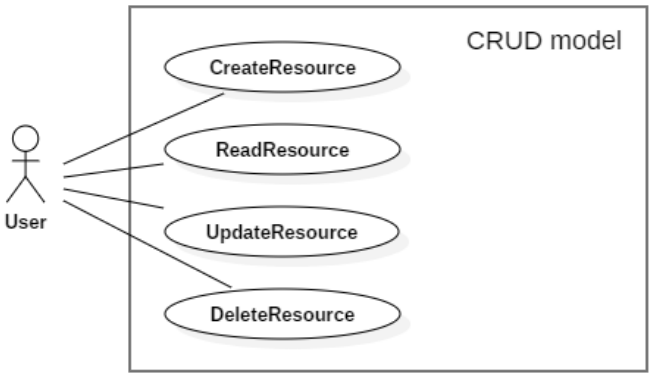

Figura 3.1: Caso d'uso del modello CRUD utilizzato nelle api REST

3.3 Architettura

I componenti del framework, definiti come ruoli, sono realizzati attraverso attori. Per soddisfare le richieste degli utenti i ruoli interagiscono tra loro scambiandosi messaggi. I messaggi sono composti dal tipo di azione da svolgere, rappresentata dalla classe del messaggio, e dai dati che devono essere trasportati. I messaggi all'interno del sistema seguono dei flussi. Nel framework i flussi principali sono due: il primo rappresenta il percorso dei messaggi da quando l'utente effettua la richiesta a quando gli viene fornita una risposta; il secondo flusso è utilizzato per processare i messaggi memorizzati temporaneamente su Apache Kafka, elaborarli e salvarli in modo permanente nel database Cassandra. Di seguito sono descritti in dettaglio i due flussi principali.

Flusso di ricezione dei messaggi

Il modello delle richieste/risposte utilizzato da qualsiasi servizio web può essere visto come un flusso di dati che inizia dal client che effettua la richiesta e termina nello stesso quando aspetta una risposta. Nel caso in cui il servizio web sia attivo in una sola macchina il percorso che i dati seguono è semplice: la richiesta arriva al server, viene elaborata e viene trasmessa una risposta. In un sistema distribuito il flusso dei messaggi si complica. Il framework separa il compito di ricezione delle richieste da quello di elaborazione.

Tramite protocollo HTTP il client si collega ad una macchina abilitata alla ricezione delle richieste, nel caso del framework un nodo che ha il ruolo di frontend. La richiesta HTTP contiene un oggetto in formato JSON dove sono presenti i dati che devono essere elaborati. Il frontend ha il compito di verificare l'integrità degli oggetti che gli sono stati inviati. Se la richiesta è mal posta o contiene errori, il frontend la scarta e informa il client del problema. In tutti gli altri casi il frontend invia l'oggetto incapsulato in un messaggio al backend che ha il compito di elaborarlo. Nel caso in cui la risorsa da creare debba essere elaborata in un momento successivo rispetto alla ricezione il messaggio viene inviato ad un broker di Apache Kafka; in caso contrario i dati vengono salvati in modo permanente nel cluster Apache Cassandra. Si può posticipare l'elaborazione del messaggio quando sono necessarie molte risorse computazionali per effettuare l'elaborazione, quando l'elaborazione richiede troppo tempo oppure quando la richiesta è di sola generazione di informazioni e non di richiesta di informazioni. Il backend dopo aver scelto uno dei due metodi può quindi informare il frontend dell'avvenuta presa in carico nel caso in cui i dati siano inviati a Kafka o inviando i riferimenti della transazione nel caso in cui la risorsa debba essere salvata in modo permanente nel cluster Cassandra. Il frontend ha infine il compito di trasformare il messaggio ricevuto dal backend in risposta HTTP che invierà al client. La connessione HTTP quindi viene chiusa. Nelle figure 3.2 e 3.3 sono mostrati due esempi che applicano il flusso descritto rispettivamente per creare una nuova risorsa e cercare una risorsa esistente.

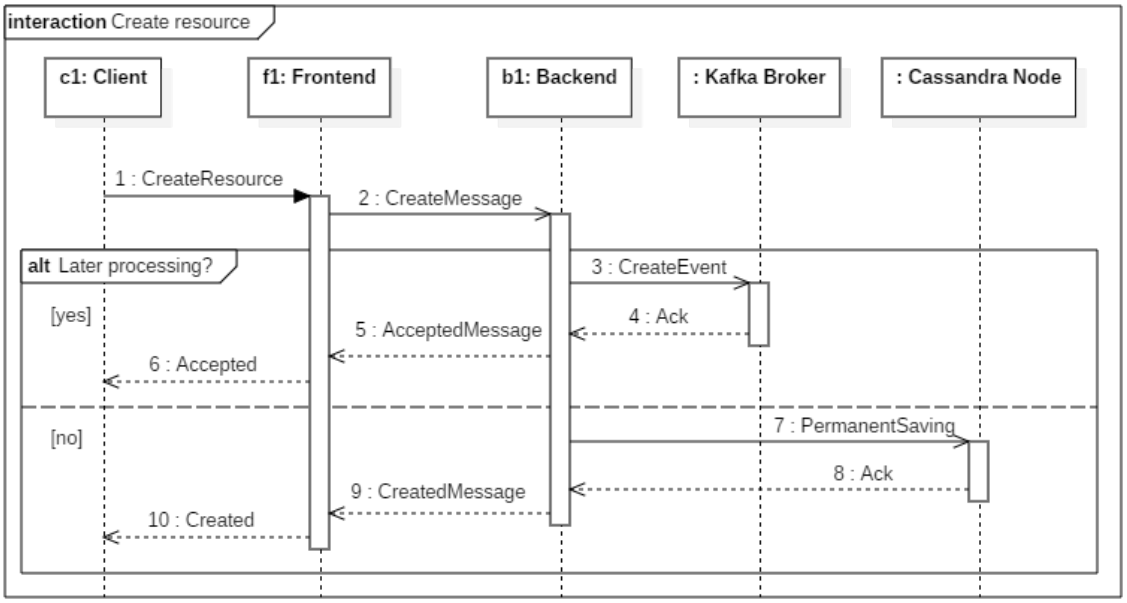

Figura 3.2: Diagramma di sequenza del flusso per creare una risorsa

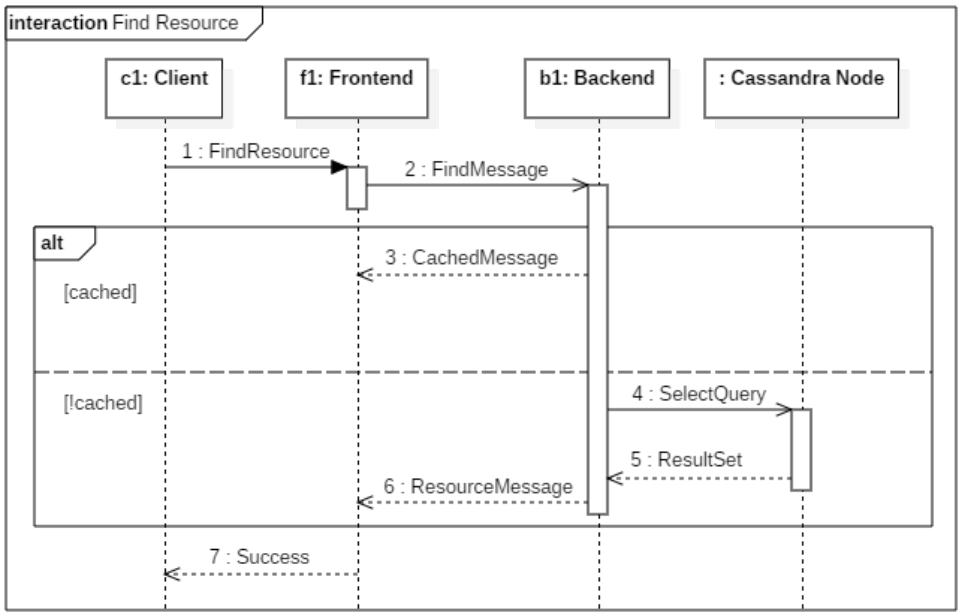

Figura 3.3: Diagramma di sequenza del flusso per cercare una risorsa

Flusso di elaborazione dei messaggi

Le richieste che non sono processate subito perché richiedono troppo tempo per essere valutate o perché non è necessario fornire immediatamente una risposta vengono memorizzate temporaneamente su Apache Kafka. Viene utilizzato un flusso per indicare il percorso che i messaggi compiono per essere elaborati quando vengono consumati da Kafka.

Il ruolo *service* si registra a Kafka come consumatore e i messaggi grezzi non ancora elaborati iniziano a transitare. Il ruolo *service* ha il compito di elaborare localmente i dati. Se la richiesta contiene informazioni da salvare in modo permanente nel database, il messaggio viene convertito in query e inviato al cluster Cassandra. Ricevuta la conferma di memorizzazione da Cassandra, *service* può inviare un messaggio al broker Kafka comunicando l'*offset* del messaggio originale per confermare l'avvenuta elaborazione dello stesso. Nella figura 3.4 è mostrato un diagramma di sequenza che mostra il percorso dei messaggi appena descritto.

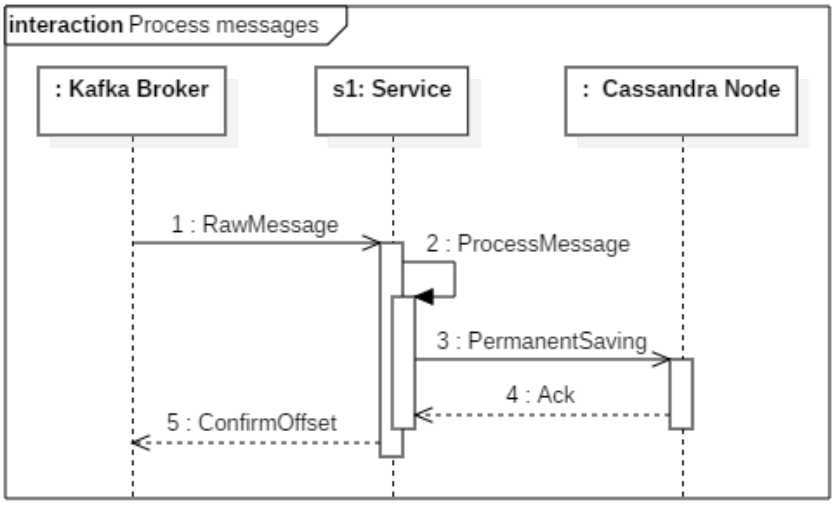

Figura 3.4: Diagramma di sequenza che mostra il flusso dei messaggi durante la fase di elaborazione

3.4 Dipendenze fra i componenti

Utilizzando un modello ad attori le dipendenze fra i componenti di uno stesso sistema devono essere minime. Le uniche dipendenze che è necessario tenere in considerazione sono quelle fra i macro-componenti che compongono il framework descritti in fase di analisi come ruoli. Nella figura 3.5 è presente un diagramma di deployment che mostra come i ruoli previsti dal framework interagiscono fra loro.

Ogni ruolo è materializzato attraverso un task Mesos e viene eseguito attraverso un esecutore presente in un agente di Mesos scelto dinamicamente dal cluster. Ogni ruolo deve conoscere gli indirizzi delle proprie dipendenze in modo da poter inviare messaggi attraverso la rete. L'indirizzo della macchina dove un task viene eseguito lo si conosce solamente a runtime, è necessario quindi un meccanismo di *service discovering* per scoprire gli indirizzi degli altri componenti. Questo meccanismo può essere realizzato utilizzando dei ruoli statici che non fanno altro che informare gli altri ruoli del posizionamento di ciascun task all'interno del cluster. Questo speciale ruolo definito già in fase di analisi è seed. Tutti i componenti del framework al momento dell'avvio dovranno collegarsi ad un nodo seed per poter richiedere di far parte del cluster e conoscere la rete di tutti gli altri componenti.

Ci sono alcuni nodi che sono collegati a più componenti. Un esempio è il ruolo frontend che è collegato a più nodi backend. Questo significa che quando il frontend deve spedire un messaggio al backend deve scegliere uno dei nodi disponibili all'interno del cluster. Deve essere presente quindi un componente speciale che sceglie dove inoltrare il messaggio. Tale componente viene definito *router* e può implementare diverse strategie di indirizzamento. La strategia più semplice e più utilizzata è quella del *round robin*: ogni volta che un nuovo messaggio deve essere inviato il destinatario viene prelevato da una lista circolare, in modo tale da distribuire equamente i messaggi da inviare. Possono essere utilizzate anche altre strategie, come quella dell'estrazione casuale o tramite indirizzamento intelligente sfruttando le metriche dei nodi

del router.

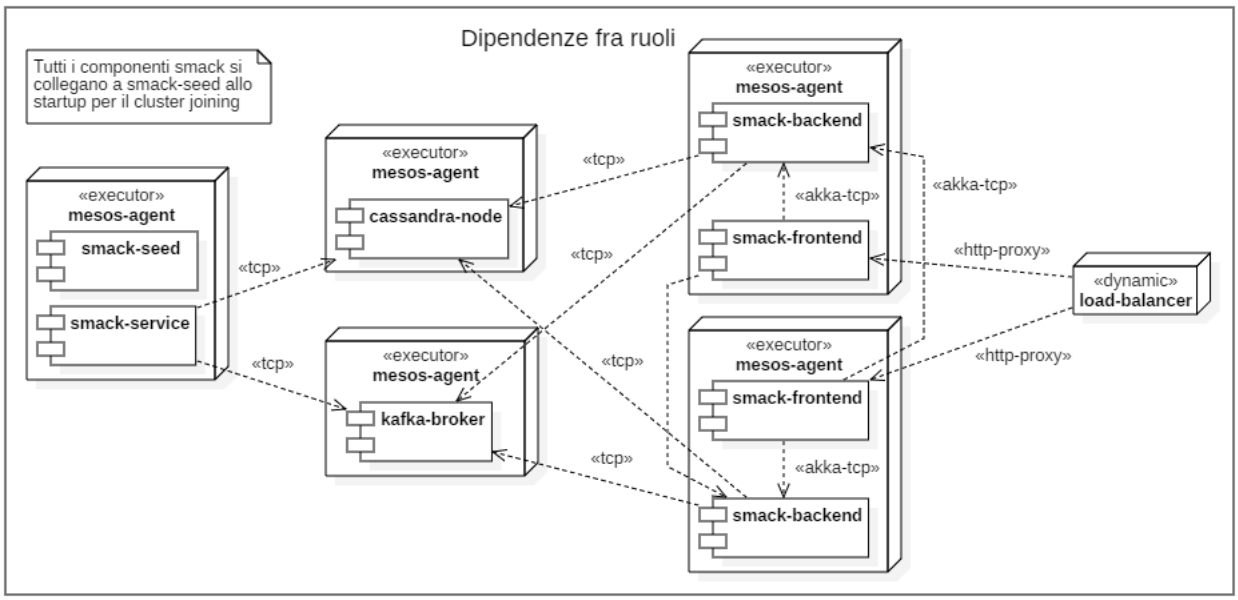

Figura 3.5: Diagramma di deployment che mostra le principali connessioni fra i componenti dello stack

3.5 Paradigmi e pattern di progettazione

Un paradigma utilizzato per la progettazione del framework è lo *stream processing*. Lo stream è un concetto astratto che rappresenta ad alto livello il flusso dei dati. In un sistema distribuito la quantità di dati trasferita è elevata. Vengono trasferiti dati quando un utente richiede una risorsa, quando dei nodi all'interno dello stesso cluster si coordinano scambiandosi informazioni e quando si devono ridondare i dati copiandoli in un altro datacenter, ad esempio. Diventa quindi utile un paradigma ad alto livello come quello degli stream per implementare in modo efficiente il trasferimento di dati.

Gli stream devono essere innanzitutto intuitivi e sicuri. È necessario che siano intuitivi per poter essere utilizzati per progettare sistemi complessi astraendo al massimo il concetto di flusso di dati. Devono essere sicuri perché

devono utilizzare meccanismi che reagiscono in caso di errore, ad esempio ritrasmettendo le parti mancanti. Devono infine essere efficienti per utilizzare al meglio le risorse disponibili, ad esempio implementando sistemi per il controllo del flusso. La funzionalità per permettere di controllare il flusso dei dati e modificare la velocità di trasmissione è uno dei punti cardine del Reactive Streams[1].

Uno stream è un processo che trasporta dati e opzionalmente li trasforma. Trasformazioni comuni sono la serializzazione degli elementi, la compressione o la cifratura. Le unità che uno stream trasporta sono chiamate elementi e sono considerate unità base anche se sono costituite da più byte. Uno stream è quindi una catena di elementi o, più appropriatamente, un grafo. Il grafo di uno stream è la descrizione della topologia, quindi del percorso che gli elementi devono percorrere quando lo stream viene eseguito. Il flusso degli elementi è controllato tramite la tecnica del *back-pressure*, un meccanismo che il consumatore ha per notificare il produttore per regolare la velocità di trasferimento degli elementi in base alle risorse che si dispongono. Per garantire la proprietà di efficienza questa tecnica deve essere asincrona e non bloccante.

Nella figura 3.6 è riportato un diagramma di attività che mostra i passaggi dello stream che il framework utilizza per inviare messaggi ai broker di Apache Kafka. La sorgente dei messaggi è rappresentata dalle richieste degli utenti che vengono effettuate tramite API REST. I messaggi che contengono l'azione da svolgere ed eventuali argomenti devono essere serializzati per essere trasportati in rete. L'operazione di serializzazione può fallire e i messaggi vengono separati in due canali differenti: da una parte i messaggi serializzati con successo e dall'altra i messaggi che contengono errori. I messaggi serializzati possono essere inviati a Kafka che in maniera asincrona risponde con l'esito dell'operazione.

[1] http://www.reactive-streams.org/

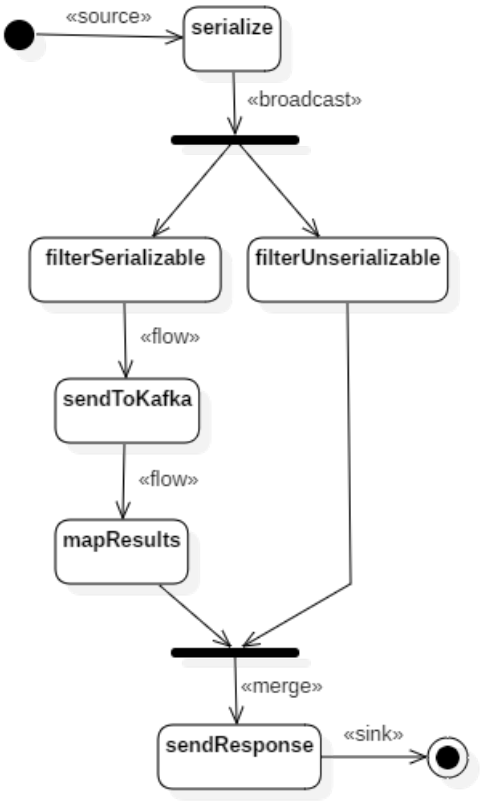

Figura 3.6: Diagramma di attività che mostra il flusso dei messaggi inviati ad Apache Kafka

La figura 3.7 mostra il procedimento inverso rispetto alla figura 3.6. In questo caso è rappresentato un diagramma di attività che indica i passaggi dello stream che il framework utilizza per consumare messaggi precedentemente inviati ad Apache Kafka. I messaggi devono innanzitutto essere deserializzati. Se il processo di deserializzazione fallisce il messaggio problematico viene scartato per non far fallire lo stream nei processi successivi. Il messaggio convertito in oggetto in memoria viene quindi processato dal componente che è in ascolto. Tutti i messaggi consumati da Kafka devono essere esplicitamente confermati attraverso il numero identificativo del messaggio, ovvero l'*offset*. Inviando la conferma ad ogni messaggio viene generato un traffico di rete

inutile e vengono sprecate risorse. Il modo corretto per confermare i messaggi è quello di raggrupparli in *chunk* e confermarli tutti insieme. Questa operazione viene definita *batching*. Anche in questo caso l'elemento terminale dello stream può essere un semplice consumatore che scarta i messaggi.

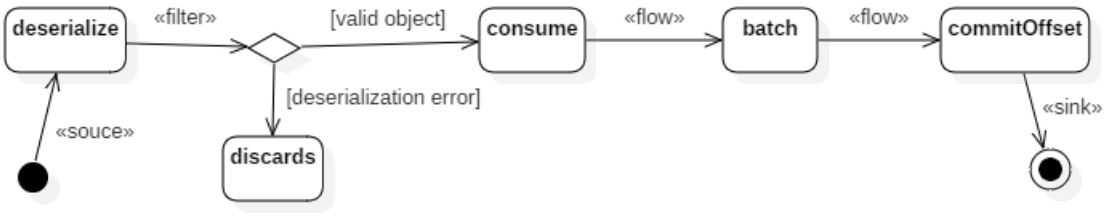

Figura 3.7: Diagramma di attività che mostra il flusso dei messaggi ricevuti da Apache Kafka

Per eseguire le query in modo asincrono si può utilizzare lo stesso paradigma, come mostrato nella figura 3.8. Le query da effettuare nel database distribuito Apache Cassandra possono essere incapsulate in messaggi e inserite in una coda di messaggi utilizzata come sorgente dello stream. Ogni messaggio viene processato sequenzialmente dallo stream ed inviato in modo asincrono al database Cassandra che esegue la query. Il risultato della query viene incapsulato in un messaggio di risposta e spedito al componente che ha chiesto di eseguire la query. Un elemento terminale dello stream provoca una pressione per fare in modo che i messaggi vengano processati il più velocemente possibile.

Figura 3.8: Diagramma di attività che mostra il flusso delle query da eseguire su Apache Cassandra

Capitolo 4

Implementazione e testing

4.1 Tecniche d'implementazione

Serializzazione

In un sistema distribuito i componenti sono dislocati su macchine diverse connesse tra loro tramite una rete che può essere locale interna al cluster o pubblica. È necessario un meccanismo che permetta di trasferire strutture dati od oggetti da una macchina all'altra, questo processo viene definito serializzazione. Tramite la serializzazione gli oggetti devono essere convertiti in un formato che è possibile trasportare via rete, solitamente in un vettore di byte. Il processo di serializzazione deve essere reversibile: un oggetto serializzato deve poter essere convertito nell'oggetto originale o in una sua copia identica tramite un processo di ricostruzione. Devono essere serializzati soltanto i campi degli oggetti e non i metodi, dato che le operazioni sugli oggetti dipendono dallo stato contenuto nei campi. Per gli oggetti complessi questo procedimento non è semplice, perché occorre serializzare non solo l'oggetto genitore ma anche tutti gli oggetti contenuti all'interno.

Per la serializzazione dei messaggi scambiati fra i componenti e i servizi il framework utilizza Protocol Buffers[1], una libreria sviluppata da Google. I

[1] https://developers.google.com/protocol-buffers/

Protocol Buffers sono un meccanismo estensibile neutro rispetto alla piattaforma, ed indipendente dalla piattaforma per serializzare i dati strutturati. Tramite Protocol Buffer è possibile implementare una serializzazione efficiente: gli oggetti in memoria vengono convertiti in vettori di byte nel modo più rapido possibile e riducendo al massimo la dimensione utilizzata. I Protocol Buffers utilizzano un linguaggio che permette di definire qualsiasi tipo di struttura dati per la maggior parte dei linguaggi di programmazione utilizzati. Includono un compilatore che trasforma le definizioni delle strutture dati in codice sorgente od oggetto utilizzabile direttamente dall'applicazione finale.

Marshalling

Un particolare tipo di serializzazione viene utilizzato per convertire oggetti in memoria in oggetti JSON e viceversa. Per l'interazione con i client il framework implementa il modello delle API REST. Il formato degli oggetti utilizzato per lo scambio di informazioni è il JSON; è necessario quindi implementare un meccanismo possibilmente automatico per effettuare la conversione. Questo processo di conversione è definito dal framework come *marshalling* ed è implementato in modo leggero, pulito ed efficiente dalla libreria `spray-json`[2].

`spray-json` permette di convertire: tra documenti in formato JSON, oggetti in memoria strutturati nello stesso modo in cui vengono rappresentati gli oggetti in JSON e tipi definiti in Scala, in particolare i `case class`. Nella figura 4.1 è rappresentato un diagramma che mostra il procedimento di conversione fra le tre tipologie citate.

[2]https://github.com/spray/spray-json

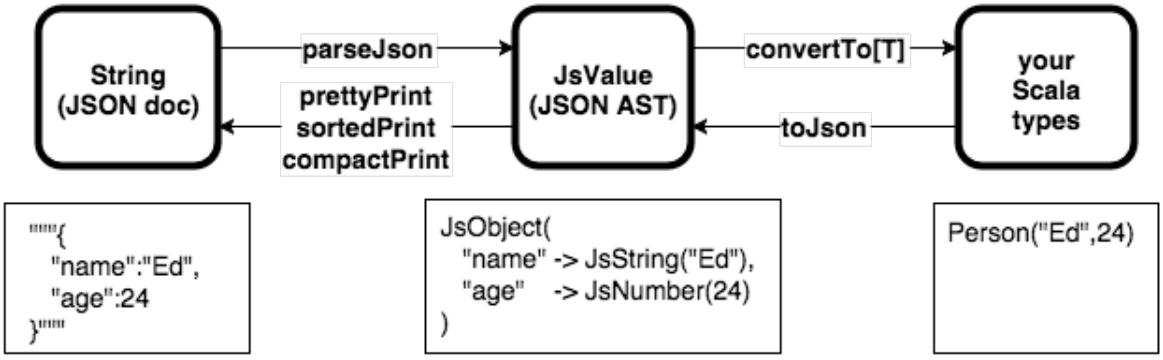

Figura 4.1: Diagramma di conversione fra diversi tipi di oggetto presente nella documentazione ufficiale di spray-json

4.2 Testing del sistema

Testing unitario

Il testing unitario è stato utilizzato ampiamente per sviluppare il framework e per verificare la correttezza dei singoli componenti del framework. Per lo sviluppo è stato utilizzato un approccio di Test Driven Development (TDD), che è un processo di sviluppo del software che consiste nello scrivere prima i test e poi implementare la logica. È un procedimento ciclico che impedisce di progredire se è presente almeno un test che fallisce. I test vengono scritti a partire dai requisiti stilati in fase di analisi; questo permette di ridurre in unità minime il codice per essere più semplice e conciso. Inoltre tramite questo metodo è più facile focalizzarsi solamente a scrivere il codice necessario per passare i test e tralasciare parti che non erano state previste in fase di analisi.

Per i test unitari è stato utilizzato Scalatest che è il più flessibile e il più usato strumento per effettuare i test nell'ecosistema di Scala[3]. Scalatest è progettato per essere facilmente estendibile per adattarsi a tutte le tipologie di progetti. Supporta nativamente diversi stili di testing, ma per il framework

[3]http://www.scalatest.org/

è stato utilizzato soltanto lo stile `FlatSpec`, perché è semplice e permette di focalizzarsi a scrivere test che si descrivono da soli, senza che sia necessario inserire ulteriore documentazione.

Il framework è predisposto per contenere i test dell'applicazione che dovrà essere sviluppata. È possibile scegliere di utilizzare anche uno stile di test differente da quello utilizzato per testare il framework.

Testing d'integrazione

Per testare il framework che consente di creare un'applicazione tramite un sistema distribuito i test unitari non bastano. I test unitari non consentono infatti di testare l'interazione fra i vari componenti che compongono il framework, ma si focalizzano soltanto sulla logica interna. Per verificare che più componenti interagiscano correttamente sono necessari i test d'integrazione. Generalmente sono test più complessi dei test unitari e soprattutto più lenti perché devono tenere in considerazione la latenza della comunicazione fra processi diversi o addirittura macchine diverse.

Per effettuare i test d'integrazione è necessario eseguire i componenti da testare in processi differenti. C'è una libreria che è compresa in Akka che serve per eseguire i test in più Java Virtual Machine (JVM) nello stesso momento. Questa tipologia di testing è stata chiamata Multi JVM Testing[4] ed è stata utilizzata per effettuare i test d'integrazione del framework.

4.3 Librerie utilizzate

Per sviluppare il framework sono state utilizzate diverse librerie. Sono state utilizzate soltanto librerie open source. Di seguito sono elencate le principali.

- **akka**: è la libreria che implementa il modello ad attori e che è già stata ampiamente discussa nel primo capitolo nella descrizione dei componenti dello stack. Oltre al modulo principale, `akka-actor`, sono

[4] https://doc.akka.io/docs/akka/2.5/multi-jvm-testing.html

compresi diversi moduli che ricoprono altre funzionalità di un sistema distribuito, tra cui: `akka-cluster` (permette di coordinare i nodi del cluster), `akka-http` (fornisce le API per creare un web server e gestire le richieste/risposte), `akka-remote` (permette la comunicazione fra due componenti su nodi diversi) e `akka-stream` (fornisce le API per lavorare con gli stream).

- **cassandra-driver**: sono i driver che permettono di connettersi al cluster Cassandra ed eseguire query.

- **kamon**: è un toolkit per monitorare le applicazioni che vengono eseguite sulla JVM. Fornisce le metriche per monitorare lo stato dei componenti del sistema distribuito e invia automaticamente i dati a un servizio di monitoraggio supportato.

- **scopt**: è una libreria che facilita il *parsing* degli argomenti dell'eseguibile.

- **sentry**: raccoglie tutti i log dell'applicazione e li invia ad un servizio esterno dove è possibile monitorarli in real-time e filtrarli.

- **uuidGenerator**: è una semplice libreria per generare gli UUID utilizzati come chiavi primarie nello schema del database. È possibile generare anche TIMEUUID, un particolare tipo di UUID che è possibile convertire in timestamp e ordinare facilmente.

- **quartz**: implementa uno *scheduler* per poter gestire più task e lanciarli in orari precisi o ad intervalli regolari.

- **spark**: è il motore che fa parte dello stack SMACK che si occupa di analizzare e processare grandi quantità di dati in maniera distribuita, sfruttando le risorse del cluster.

4.4 Dispiegamento e cenni sulla sicurezza

Per lanciare l'applicazione all'interno del cluster occorre prima installare e configurare Apache Mesos per la coordinazione e la gestione delle macchine. Per installare Mesos è possibile fare riferimento alla guida presente nell'Appendice A. Inoltre occorre uno strumento per gestire le applicazioni che verranno lanciate all'interno del cluster Mesos. Per lo sviluppo di questo progetto è stato utilizzato Marathon, un framework creato da Mesosphere per la gestione di applicazioni a lunga durata, che è possibile configurare seguendo le istruzioni riportate nell'Appendice B.

Attraverso Marathon è possibile eseguire qualsiasi tipo di applicazione. Per lanciare un'applicazione è necessario definire alcuni parametri che serviranno a Marathon per allocare le risorse necessarie e avviare l'applicazione. I più importanti parametri solitamente indicati sono: un id univoco che rappresenta l'applicazione, le risorse hardware necessarie per il dispiegamento (tempo di CPU, memoria principale, disco), le informazioni sul tipo di container da utilizzare (Docker o LCX) e il nome delle immagini che contengono i binari da prelevare, la definizione delle porte utilizzate per comunicare con le altre applicazioni, i volumi utilizzati nel caso l'applicazione debba memorizzare unità di supporto permanente, gli argomenti da passare all'applicazione quando si avvia e le variabili d'ambiente da definire.

È possibile interagire con Marathon attraverso API REST. Definiti i parametri per ogni componente del sistema distribuito è possibile scalare ogni istanza del sistema in modo indipendente. Ciascuna istanza è definita da un *task* che rappresenta il *task* Mesos ed è lanciata in un container in una macchina scelta tramite un criterio, o scelta casualmente all'interno del cluster. Le istanze sono quindi indipendenti: ognuna salva i propri log nel proprio contenitore e se la singola istanza fallisce viene ricoverata soltanto l'istanza difettosa. Mesos effettua un controllo sulla salute delle singole istanze. Se un'istanza fallisce ritornando un codice di errore Mesos si preoccupa di distruggere il container e creare una nuova istanza con le stesse caratteristiche.

Dispiegamento dell'applicazione

Per poter eseguire un'applicazione realizzata con il framework oggetto di questa tesi è necessario che nel cluster vengano lanciate almeno un'istanza di Apache Kafka e una di Apache Cassandra. Di questi due nodi si deve conoscere l'indirizzo e la porta utilizzati per poter estendere i due servizi e connettere altre istanze. Le prime due istanze lanciate vengono definite *seed* perché consentono alle altre istanze create successivamente di unirsi e far parte dello stesso cluster. Possono essere lanciati anche più nodi di *seed* per essere sicuri che ce ne sia almeno uno disponibile oppure utilizzare metodi di *service discovering* più complessi. Le istruzioni per effettuare queste operazioni sono riportate nell'Appendice D.

Avendo già formato un cluster Cassandra e un cluster Kafka si può procedere a lanciare l'applicazione. Anche l'applicazione deve avere una o più istanze *seed* che hanno il compito di coordinare tutti i componenti dell'applicazione e aggiungere nuove istanze. Gli altri tre ruoli del framework, ovvero *backend*, *frontend* e *service* possono essere lanciati in qualsiasi ordine. Per funzionare l'applicazione ha bisogno di almeno un'istanza di ciascun ruolo. I ruoli possono essere scalati in modo indipendente: se il *backend* dell'applicazione impiega diverse risorse e tempo per processare le richieste, è lecito stanziare molti nodi *backend* e pochi nodi *frontend* abilitati alla ricezione.

Il ruolo di ogni istanza è deciso all'avvio tramite un parametro. Gli argomenti con cui devono essere lanciati i componenti dell'applicazione vengono impostati attraverso Marathon. Oltre al ruolo che l'istanza deve assumere, sono presenti altri argomenti che è obbligatorio indicare, tra cui: l'indirizzo dei nodi *seed* per effettuare l'operazione di *cluster joining* dell'applicazione, l'indirizzo dei punti di contatto Cassandra e l'indirizzo delle macchine di bootstrap di Kafka. È possibile indicare inoltre se il debug debba essere attivato, l'ambiente in cui il componente viene eseguito e il livello dei log.

È possibile distribuire i binari dell'applicazione tramite immagini Docker che possono essere caricate sul registro pubblico o su un registro privato.

Bilanciamento del carico

Nella maggior parte delle situazioni si vuole nascondere il fatto che l'applicazione si basi su un sistema distribuito. Questo è il caso ad esempio di quando si vuole raggiungere l'applicazione da un solo nome di dominio. Per distribuire equamente le richieste che provengono dai client ai nodi che si occupano di riceverle si possono utilizzare due metodi: si effettua la distribuzione tramite DNS oppure si utilizza un *load balancer*.

La distribuzione tramite DNS si basa su un meccanismo semplice: quando il DNS viene interpellato per risolvere il nome a dominio dell'applicazione il DNS risponde fornendo un indirizzo sempre diverso. Quando finisce gli indirizzi dei nodi abilitati alla ricezione delle richieste inizia da capo. Questo algoritmo di smistamento viene chiamato *round robin*. Questo sistema non sempre è adatto per vari motivi. Uno fra questi è che il DNS è lento a propagarsi e se una macchina abilitata alla ricezione smette per un qualsiasi motivo di funzionare il DNS continuerà a risolvere il dominio in quell'indirizzo per un periodo di tempo. Il client che riceverà quell'indirizzo non avrà quindi il servizio funzionante.

Il metodo più adatto per distribuire le richieste è utilizzare un *load balancer*. Esistono due tipi di *load balancer*: hardware o software. I *load balancer* hardware sono i più performanti ma anche quelli più costosi. Alcuni di questi più avanzati possono essere utilizzati con connessioni cifrate attraverso il protocollo https; in questo modo si risparmiano risorse computazionali del cluster che sarebbero invece utilizzate per cifrare i dati. Di *load balancer* software ce ne sono di diversi tipi anche open source. Non sempre sono la soluzione al problema perché quando si hanno carichi estremamente elevati le risorse computazionali di una macchina non bastano per smistare le risorse.

Nella figura 4.2 è mostrato un diagramma di sequenza che mostra l'utilizzo del *load balancer* in un'applicazione realizzata con il framework.

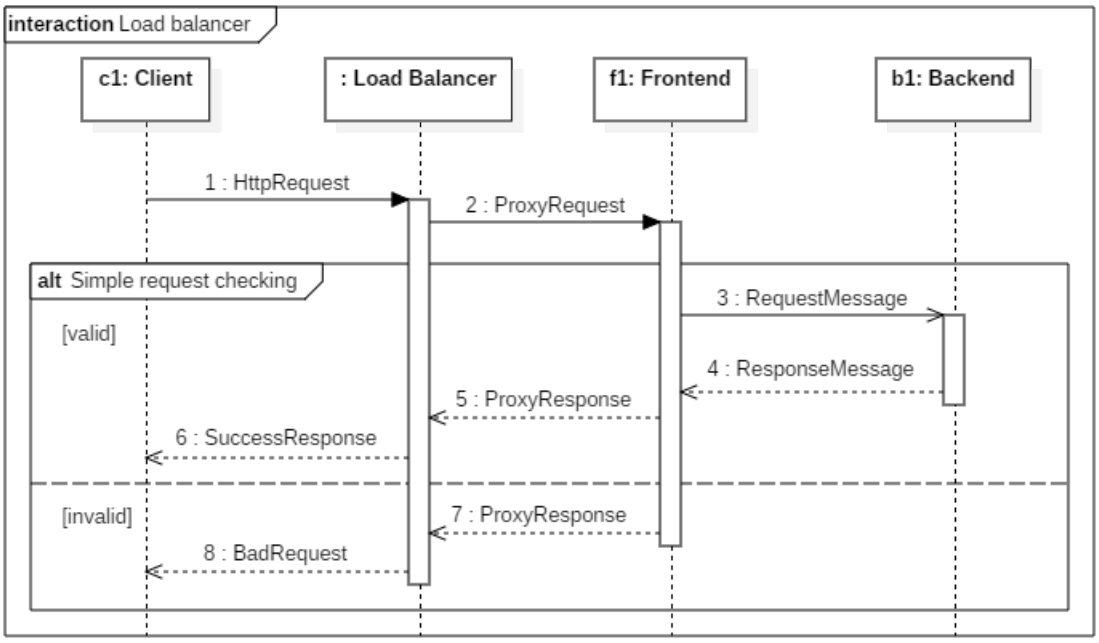

Figura 4.2: Diagramma di sequenza che mostra l'utilizzo del load balancer per smistare le richieste

Protezione attraverso l'uso di firewall

In un sistema distribuito i componenti interagiscono fra loro attraverso delle connessioni di rete. Ciascun componente se vuole interagire deve avere almeno una porta aperta in modo che altri componenti si possano connettere. Occorre proteggere queste porte e impedire che dall'esterno del cluster qualcuno si possa connettere violando il sistema.

Un modo semplice per risolvere il problema è utilizzare un'interfaccia di rete locale per connettere i nodi fra loro, lasciando l'interfaccia pubblica disponibile solo per i componenti che devono accettare connessioni dall'esterno. Spesso quando si affittano risorse da gestori cloud questa opzione non è possibile. Un modo alternativo per proteggere le porte dei nodi del cluster è utilizzare un firewall. Esistono due tipi di firewall: hardware o software. Il

firewall hardware è più performante ma generalmente è più costoso di un firewall software, che è disponibile open source. Il firewall va configurato a dovere per ciascuna macchina del cluster in modo tale da autorizzare le connessioni tra nodi interni nel cluster e bloccare le connessione provenienti dall'esterno.

Criptazione dei canali di comunicazione

Quando si affittano delle risorse cloud pubbliche occorre tenere in considerazione che la rete utilizzata da quelle macchine è condivisa. È buona norma oltre che proteggere le macchine con firewall criptare i canali di comunicazione fra componenti del sistema distribuito che non si trovano sulla stessa macchina.

Tutti i software e le librerie utilizzate dal progetto permettono di creare linee di comunicazione punto a punto criptate. Per criptare la comunicazione può essere utilizzato il protocollo crittografico SSL/TLS. Ciascuna macchina per poter interagire con le altre deve avere installati i certificati di sicurezza e le chiavi pubbliche e private per poter accettare e iniziare le comunicazioni. Occorre quindi progettare un sistema per distribuire queste chiavi in maniera sicura.

Autenticazione e autorizzazione dei componenti

Per aggiungere un ulteriore livello di sicurezza è possibile autenticare ciascun componente. In questo modo se attraverso falle si riesce ad entrare nella stessa rete del cluster non sarà possibile fingersi un componente sano senza essere autenticati. Sono disponibili diversi modi per configurare l'autenticazione che può essere effettuata anche in maniera centralizzata. Inoltre è possibile anche assegnare a ciascun componente un'autorizzazione che permette di limitare le operazioni che può fare. In questo modo se si devono proteggere parti del sistema sensibili si impedisce l'accesso a componenti non autorizzati che potrebbero anche involontariamente creare dei danni.

Capitolo 5

Un caso di studio

5.1 Un'applicazione realizzata con il framework

Il caso di studio tratta un'applicazione sviluppata per funzionare con lo stack SMACK tramite il framework oggetto della tesi.

Scopo e requisiti dell'applicazione

L'applicazione da realizzare è un semplice raccoglitore di click distribuito. Dovranno essere fornite delle API pubbliche tramite architettura REST per poter interagire con l'applicazione. Il sistema dovrà tenere traccia delle pagine visitate dagli utenti di altre applicazioni *web-based* e salvare le informazioni degli utenti che le visitano. Tramite quest'applicazione deve essere possibile generare report sulle statistiche che mostrano il numero degli accessi ad un sito in un determinato periodo di tempo e gli indirizzi URL più richiesti.

Il sistema traccia le richieste solo dei siti registrati. Ciascun sito appartiene ad un utente, che può registrare più siti. Per utente si intende l'entità che può interagire con il sistema, non il visitatore di cui vengono raccolti i click nelle pagine dei siti registrati. Ad ogni sito è associato obbligatoriamente un solo nome di dominio. Le visite delle pagine vengono raccolte sotto forma di

log che contengono l'URL della pagina visitata, l'indirizzo IP del visitatore e l'user-agent del visitatore. Tramite IP si potrà scoprire la regione geografica di provenienza, mentre dall'user-agent è possibile stabilire il browser e il sistema operativo utilizzati dal visitatore.

Funzionamento dell'applicazione

Per poter utilizzare l'applicazione è necessario essere utenti registrati. Un utente non registrato può creare il proprio utente, mentre un utente già registrato può richiedere le informazioni del suo account e può modificarlo, come mostrato nel diagramma dei casi d'uso in figura 5.1.

A ciascun utente deve essere associato un indirizzo e-mail e il nome completo.

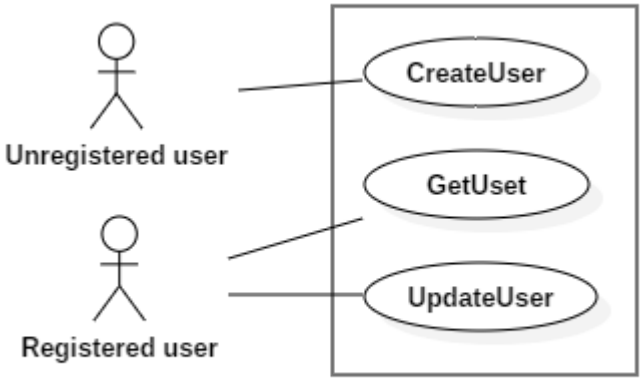

Figura 5.1: Diagramma dei casi d'uso per l'entità utente

Un utente può ottenere la lista dei siti che ha creato oppure può creare nuovi siti. Inoltre deve essere possibile ottenere le informazioni di dettaglio di un sito già creato, deve essere possibile modificarlo e rimuoverlo, come mostrato nel diagramma dei casi d'uso in figura 5.2.

A ciascun sito è associato uno ed un solo dominio. Deve essere inoltre generato un codice di tracking che deve essere utilizzato come riferimento quando vengono inviati i dati sui click delle pagine di quel determinato sito.

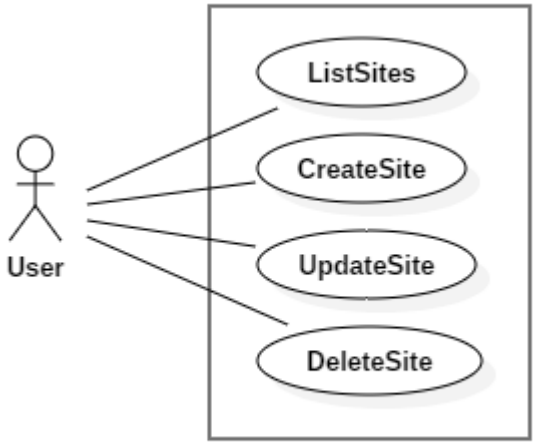

Figura 5.2: Diagramma dei casi d'uso per l'entità sito

Infine deve essere possibile per i siti registrati inviare log sugli accessi alle pagine. È possibile, attraverso uno script in Javascript, inviare in modo asincrono le informazioni dell'utente e l'indirizzo della pagina visitata all'applicazione.

Le operazioni per gestire gli utenti e i siti vengono eseguite immediatamente. Ciò significa che le richieste effettuate modificano le informazioni presenti nel database e la risposta conferma l'avvenuta modifica. I log che contengono le informazioni sulle visite delle pagine invece non vengono processati subito ma vengono inviati a Kafka e processati in un momento successivo.

Si è scelta questa strategia perché le richieste che interagiscono con le entità utenti e siti sono limitate ed è possibile processarle subito perché non richiedono particolari risorse. I log devono invece essere analizzati prima di essere memorizzati nel database Cassandra e vengono impiegate risorse. Non è necessario far attendere il client durante la fase di analisi quindi al client viene inviata solo la notifica di presa in carico e il log viene analizzato in un momento successivo.

Le tre entità descritte vengono gestite da tre componenti separati, `UserController`, `SiteController` e `LogController`. Ciascuno di questi componenti, realizzati tramite attori, è supervisionato dal `BackendSupervisor` come mostrato nel diagramma delle classi in figura 5.3.

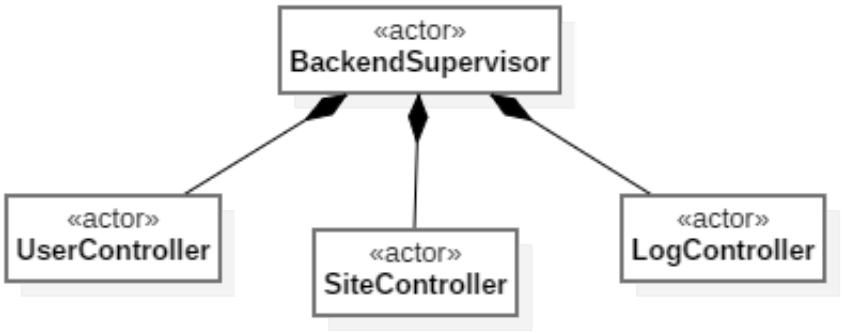

Figura 5.3: Diagramma delle classe dei componenti del backend

Il servizio che si occupa di processare e analizzare i log contenenti le visite delle pagine è implementato dal `LogService` supervisionato dal `ServiceSupervisor`, come mostrato nel diagramma delle classi in figura 5.4

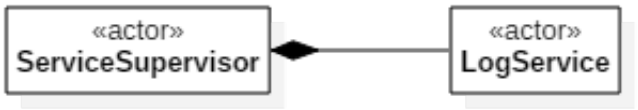

Figura 5.4: Diagramma delle classi dei componenti del service

5.2 Progettazione della base di dati

Per progettare la base di dati è necessario tenere in considerazione che Cassandra è un database distribuito non relazionale. Le regole consigliate per la progettazione di schemi per i database relazionali non sono adatte per modellare dati in Cassandra.

Prima di modellare il dominio reale è necessario capire a fondo i dati: tutto il dominio dell'applicazione deve essere analizzato e compreso prima di iniziare la fase di progettazione dello schema. È necessario inoltre conoscere a priori in che modo devono essere interrogati i dati. Non potendo effettuare operazioni di join alcune operazioni di selezione non possono essere eseguite; una soluzione a questo problema è quella di ridondare i dati. È necessario definire quindi quali colonne per ogni tabella deve essere possibile selezionare. La filosofia di Cassandra è quella di scrivere più dati per leggerli più velocemente. Se ottimizzati a dovere, la duplicazione dei dati in Cassandra non è quindi un problema.

Per progettare una corretta base di dati con Cassandra è necessario seguire un procedimento. Si parte dalla creazione del modello concettuale con l'ausilio degli schemi e-r utilizzando lo stesso metodo della progettazione di schemi relazionali. Utilizzando una metodologia basata sulle query si costruisce il modello logico. Applicando un'analisi e validando il modello logico si costruisce infine il modello fisico dal quale creare le strutture dati nel cluster Cassandra.

La metodologia basata sulle query serve per progettare il modello logico basandosi sulle query che devono essere utilizzate dall'applicazione finale. Occorre innanzitutto utilizzare una chiave primaria consistente in modo che i dati possano essere distribuiti in maniera equa all'interno del cluster. Occorre minimizzare il numero delle partizioni da leggere evitando però di creare partizioni che possono diventare troppo grandi.

Si ricorda che in Cassandra la chiave primaria è formata da due sotto-chiavi: la chiave di partizione, utilizzata per stabilire in quale partizione si deve

trovare un dato, e la chiave di *clustering*, che indica in che modo devono essere ordinati i dati all'interno di una partizione. Per trasformare il modello concettuale in modello logico il procedimento in linea generale è il seguente.

- Le entità e le relazioni si convertono in tabelle.
- Gli attribuiti chiave si trasformano in chiavi primarie.
- Gli attributi di ricerca tramite eguaglianza devono essere chiavi di partizione.
- Gli attributi di ricerca tramite ineguaglianza e gli attributi di ordinamento diventano chiavi di *clustering*.

Creato il modello logico è fondamentale effettuare una fase di analisi e validazione. È necessario tenere in considerazione che una volta che lo schema è stato creato non sempre è possibile modificarlo senza dover sostituire intere parti. Prima di convertire il modello logico in modello fisico occorre essere sicuri che:

- I dati vengano distribuiti in maniera equa fra le partizioni;
- Quando si effettua una query di lettura, i dati si trovino in un'unica partizione;
- Non ci siano conflitti di scrittura che comportino sovrascritture. È necessario quindi essere sicuri che non possa essere possibile inserire due record diversi con la stessa chiave primaria.
- La grandezza delle partizioni sia contenuta. Per verificarlo è possibile risolvere la seguente equazione:

$$n_{cells} = n_{row} \cdot (n_{cols} - n_{pk} - n_{static}) + n_{static} < 1M$$

- Non si duplichino i dati più di tre volte.

5.3 Schema versioning con smack-migrate

Quando si realizza qualsiasi tipo applicazione si seguono diverse fasi che indicano il percorso da seguire per sviluppare un software. Le tre fasi che si seguono dopo la fase di progettazione sono la fase di sviluppo, la fase di testing e la fase di dispiegamento. In ciascuna di queste fasi l'applicazione viene eseguita in un ambiente diverso. Nella fase di sviluppo l'applicazione viene eseguita sulla macchina locale dello sviluppatore, nella fase di testing il codice viene eseguito in un server remoto tramite un processo di integrazione continua mentre nella fase di dispiegamento l'applicazione è lanciata nelle macchine di produzione.

In ciascuno di questi tre ambienti l'applicazione deve avere accesso alla base dati. È assolutamente necessario però che la base dati non sia in comune, ma che abbia almeno una versione diversa per ciascuna fase. Per unificare lo schema di ciascuna base di dati e per poterlo creare e distruggere in modo automatico è stato creato lo strumento smack-migrate.

Lo strumento permette di effettuare migrazioni dello schema del database gestito da Cassandra. Per migrazione si intende il processo di evoluzione dello schema del database. Ogni migrazione deve essere una semplice operazione, ad esempio la creazione di una tabella o l'inserimento di un indice.

Ad ogni migrazione deve essere associato un tag univoco a tutte le migrazioni, una query per creare la struttura dati elementare e una query per distruggerla. Le migrazioni vengono eseguite mantenendo l'ordine stabilito. Lo strumento tiene traccia di tutte le migrazioni effettuate salvando lo stato dello strumento nella tabella `migrations` del database.

È possibile inserire nuove migrazioni in qualsiasi momento; quando se ne inseriscono di nuove occorre rieseguire la migrazione in modo da creare soltanto le strutture mancanti. Non è possibile alterare vecchie migrazioni già effettuate o modificare tag perché si comprometterebbero i risultati dello strumento.

5.4 Verificare le performance con smack-client

Questo strumento è stato creato per verificare se l'applicazione soddisfa parte dei requisiti non funzionali stabiliti in fase di analisi. In particolare attraverso questo strumento e altri sistemi di misurazione è possibile misurare le performance del sistema distribuito per verificare ad esempio se la proprietà di scalabilità è stata rispettata.

smack-client è uno strumento di stress testing: serve per inviare all'applicazione un numero personalizzabile di richieste. La tipologia delle richieste e i dati da inviare sono modificabili. All'avvio dello strumento vengono creati un numero di attori pari al parametro *parallelism*. Ogni attore contiene una coda nella quale vengono inseriti ogni secondo un valore di *requests-per-second* richieste. Ogni richiesta viene effettuata all'indirizzo `http://$host:$port/$uri`. Il client elabora solo le richieste che riesce ad effettuare, le altre le scarta. Ogni 5 secondi vengono stampate le statistiche parziali che contengono:

- Il numero di risposte ottenute, suddivise per il codice di risposta
- Il numero di richieste inserite nella coda delle richieste
- Il numero di richieste scartate dalla coda delle richieste (queue overflow)
- Il numero di richieste fallite
- Il numero totale di richieste
- Il tempo minimo, in millisecondi, fra richiesta e risposta
- Il tempo massimo, in millisecondi, fra richiesta e risposta
- Il tempo medio, in millisecondi, fra richiesta e risposta

Al termine dell'esecuzione del client, quando sono state effettuate *count* richieste o quando il programma è interrotto forzatamente, vengono stampate le statistiche globali.

Capitolo 6

Analisi delle performance

In questo capitolo verranno analizzate approfonditamente le performance del framework sviluppato che utilizza tutti i componenti dello stack SMACK. Sono stati realizzati due tipi di test: nel primo è stato stanziato un numero fisso di componenti di `backend` e sono state misurate le performance all'aumentare dei nodi di `frontend`. Nel secondo test è stato utilizzato un numero fisso di istanze `frontend` e un numero variabile di istanze `backend`.

Per effettuare i test sono state utilizzate macchine virtuali create tramite un micro-servizio localizzato ad Amsterdam. È stata utilizzata una sola macchina master dove è stato eseguito il servizio master di Apache Mesos e dove è presente il load balancer. Per effettuare il bilanciamento del carico è stato utilizzato il software open source HaProxy. Il load balancer non fa altro che smistare le richieste fra le istanze `frontend` attive tramite un algoritmo round robin. Ciascuna macchina in cui è stato lanciato il servizio slave di Mesos è dotata di due CPU virtuali e quattro gigabyte di RAM. Sono state utilizzate in totale otto macchine slave.

Per misurare le capacità e l'efficienza del sistema è stato utilizzato soltanto un tipo di richiesta HTTP inviata al sistema tramite lo strumento `smack-client`. Ciascuna richiesta esegue il percorso seguente: un'istanza `frontend` riceve la richiesta e verifica che sia composta correttamente; viene inviato un messaggio ad un'istanza `backend` che effettua una query al cluster

Cassandra con i dati ricevuti e aspetta una risposta; viene eseguita un'elaborazione locale dei dati e viene inviato un messaggio a Kafka per terminare l'elaborazione del messaggio in un secondo momento; il `backend` notifica il `frontend` dell'avvenuta presa in carico, il quale risponde al client.

Per ciascuna richiesta viene misurato il periodo di tempo dall'invio della richiesta alla ricezione della risposta. Come riferimento per ogni misurazione è stato preso un intervallo di tempo di 25 secondi, nel quale vengono misurati il tempo minimo, il tempo massimo e il tempo medio che impiegano le richieste ad essere eseguite. Inoltre viene misurata la media dell'utilizzo delle risorse sempre nello stesso periodo di tempo delle istanze oggetto di misurazione.

6.1 Misurazione efficienza del ruolo frontend

Per eseguire questo test sono state utilizzate un numero variabile di istanze `frontend` in cinque misurazioni. Nella prima misurazione è stata lanciata una sola istanza `frontend`, mentre nelle successive è stato incrementato il numero di istanze. Sono stati utilizzati un numero costante di istanze di altri componenti per semplificare i test. In ogni misurazione il sistema era composto da otto istanze di `backend`, tre broker Kafka e tre nodi Cassandra. I client hanno sempre cercato di inviare il massimo numero di richieste possibile. Tutte le richieste sono state inviate al load balancer.

Nella figura 6.1 è riportato il grafico che mostra il numero di richieste effettuate per secondo. Nel grafico è stato incluso con una linea tratteggiata l'andamento ottimale all'aumentare del numero delle istanze di `frontend`. Più le due linee si avvicinano più la proprietà di scalabilità è stata rispettata. Dal grafico si può notare che con pochi nodi si ha una scalabilità quasi perfetta. Con l'aumentare del numero delle istanze la linea che indica il numero di richieste effettuate si distacca sempre di più dalla linea tratteggiata che indica il numero delle richieste ottimali: questo è dovuto al fatto che il load balancer non riesce più a reggere il carico di richieste e gli altri componenti del sistema, dovendo gestire più richieste, aumentano la latenza.

Figura 6.1: Numero delle richieste effettuate per secondo dal ruolo frontend

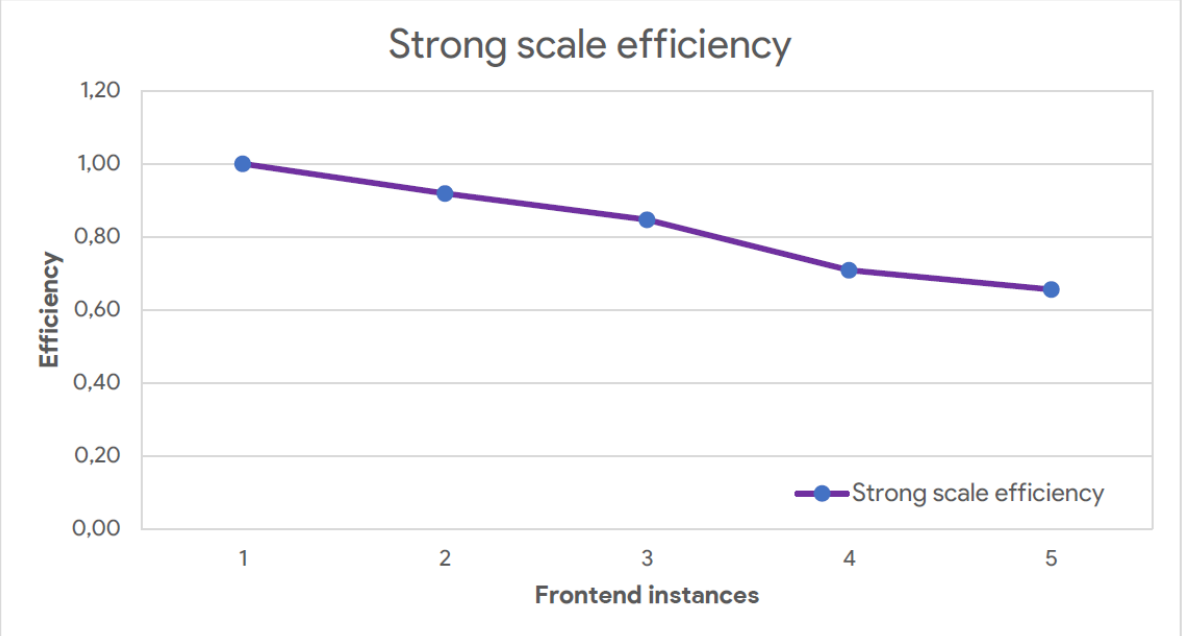

Figura 6.2: Misurazione strong scale efficiency del ruolo frontend

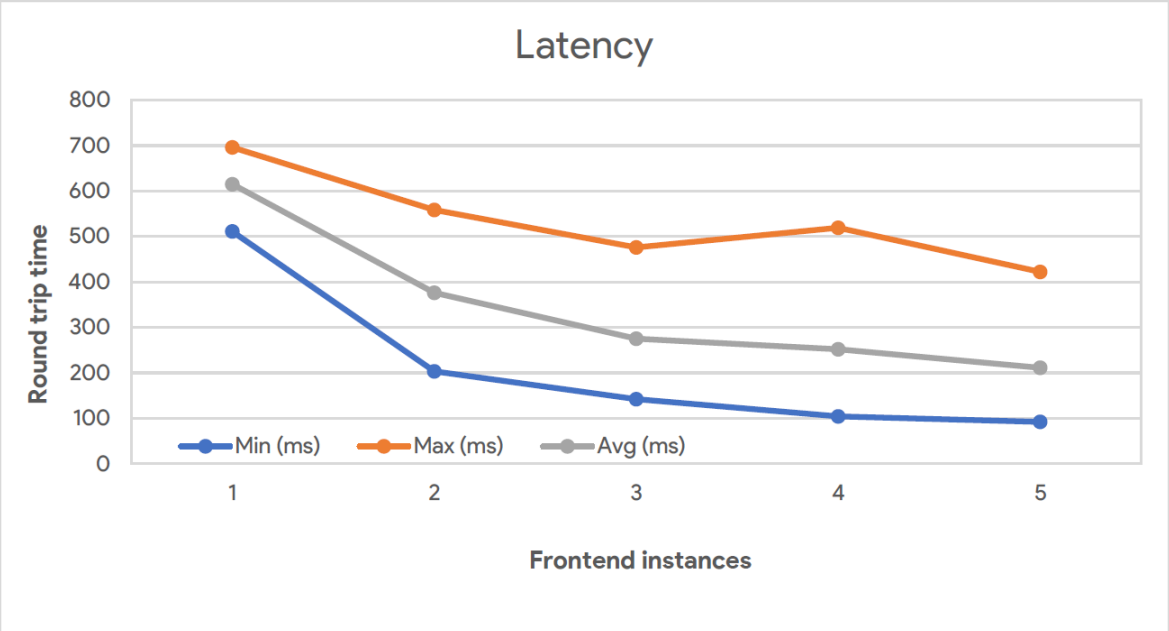

Figura 6.3: Latenze delle richieste per il ruolo frontend

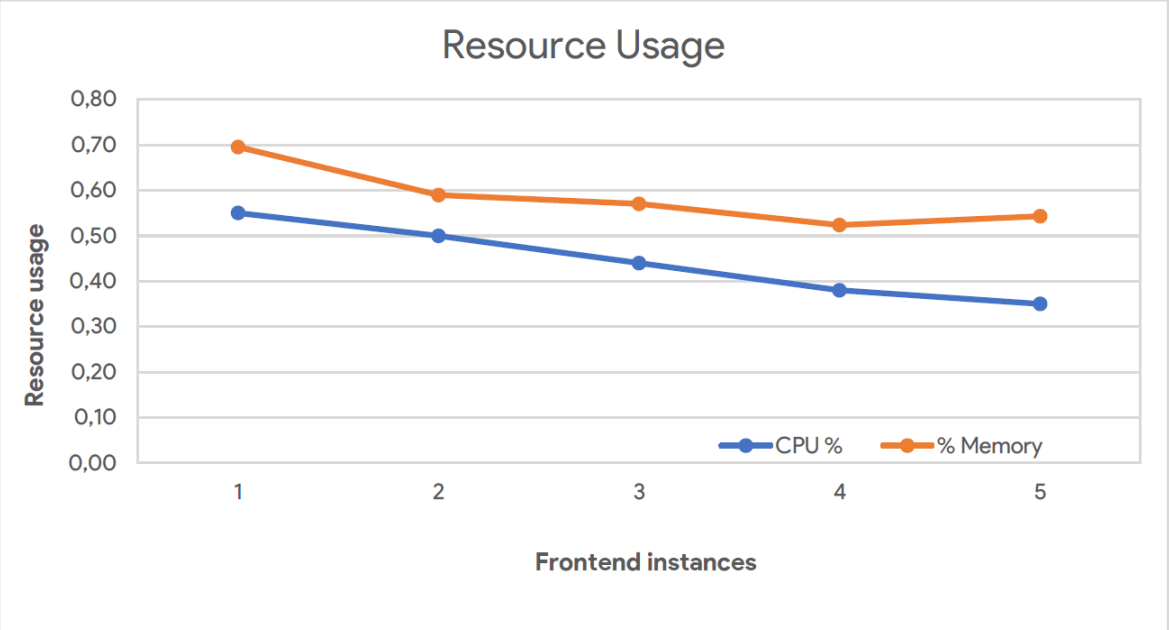

Figura 6.4: Utilizzo delle risorse per il ruolo frontend

Nella figura 6.2 è riportato il grafico che mostra la *strong scale efficiency* per ogni misurazione. Anche da questo risultato si deduce quello che è stato spiegato precedentemente. Nella figura 6.3 è mostrato il grafico della latenza minima, massima e media di ciascuna richiesta. Si può notare che quando è attiva solo un'istanza di `frontend` i tempi di latenza sono elevati. Aumentando il numero di istanze `frontend` il carico si distribuisce tramite l'algoritmo di round robin del load balancer fra le istanze attive e la latenza scende fino a raggiungere livelli accettabili. Per completezza, nel grafico mostrato in figura 6.4 è presente l'utilizzo delle risorse. La misurazione indica una media dell'utilizzo delle risorse delle istanze attive.

6.2 Misurazione efficienza del ruolo backend

Per effettuare questo tipo di test è stato utilizzato un numero variabile di istanze `backend` mantenendo fisso il numero di istanze `frontend`. Durante le misurazioni nel sistema erano presenti otto istanze `frontend` connesse a load balancer, tre broker Kafka e tre nodi Cassandra. Nella figura 6.5 è riportato il grafico che mostra il numero di richieste effettuate dal client per secondo. Dal grafico si può notare che aumentando il numero di istanze di `backend` il numero delle richieste soddisfatte aumenta in modo poco significativo. Infatti la linea tratteggiata che indica l'andamento ottimale si discosta enormemente dall'andamento effettivo delle richieste. Questo è dovuto al fatto che si è creato un collo di bottiglia nel load balancer che non riesce a gestire più di un certo numero di richieste. Questo rende il test insignificante. Dal grafico mostrato in figura 6.6 viene mostrato lo stesso problema ma da una prospettiva diversa. È possibile notare chiaramente che all'aumentare del numero delle istanze di `backend` l'efficienza scende rapidamente. In figura 6.7 viene invece mostrato il grafico della latenza. Da questo grafico si può notare che inizialmente, all'aumentare delle istanze di `backend`, la latenza diminuisce rapidamente, mentre superata una certa soglia diventa stazionaria. Infine nel grafico in figura 6.8 è riportato l'utilizzo delle risorse.

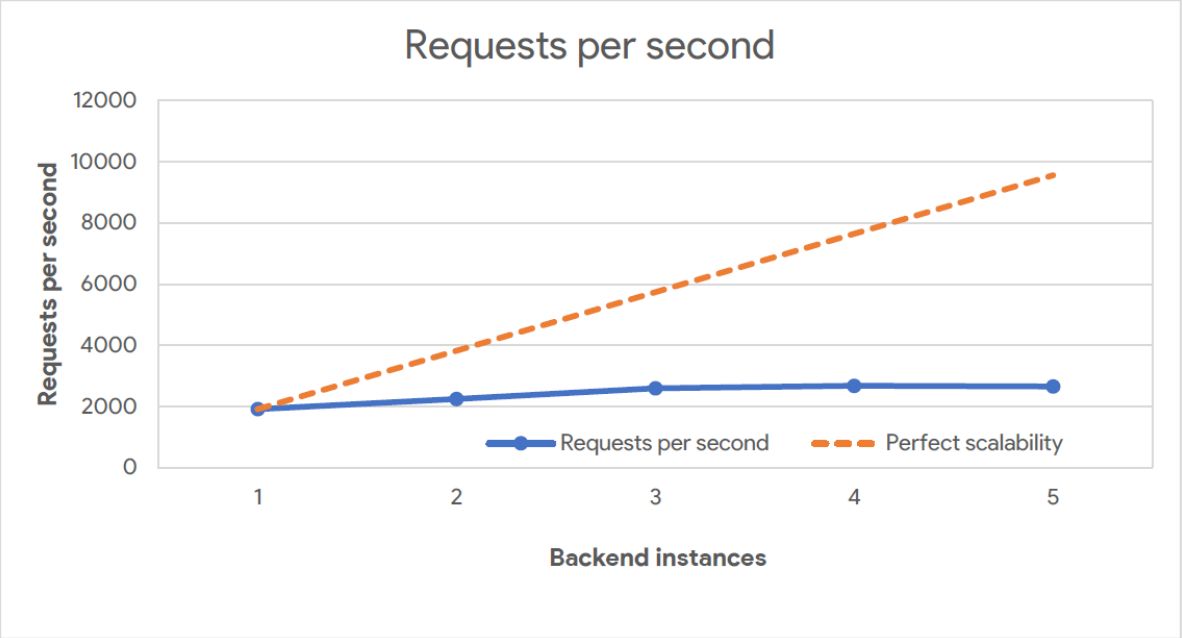

Figura 6.5: Latenze delle richieste per il ruolo backend

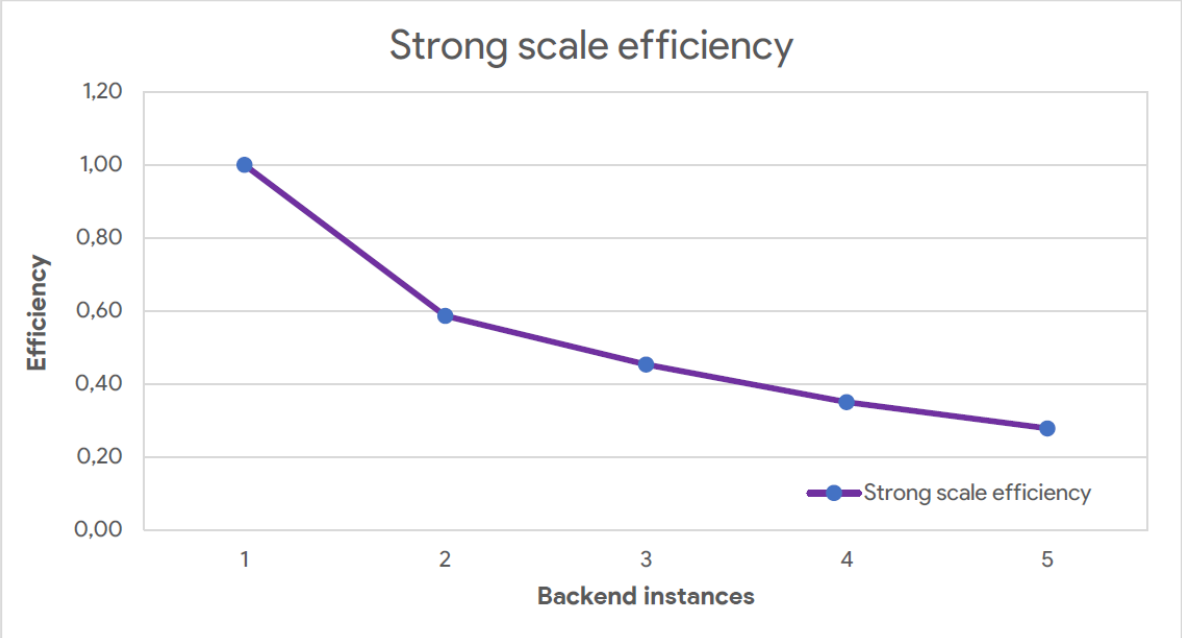

Figura 6.6: Misurazione strong scale efficiency del ruolo backend

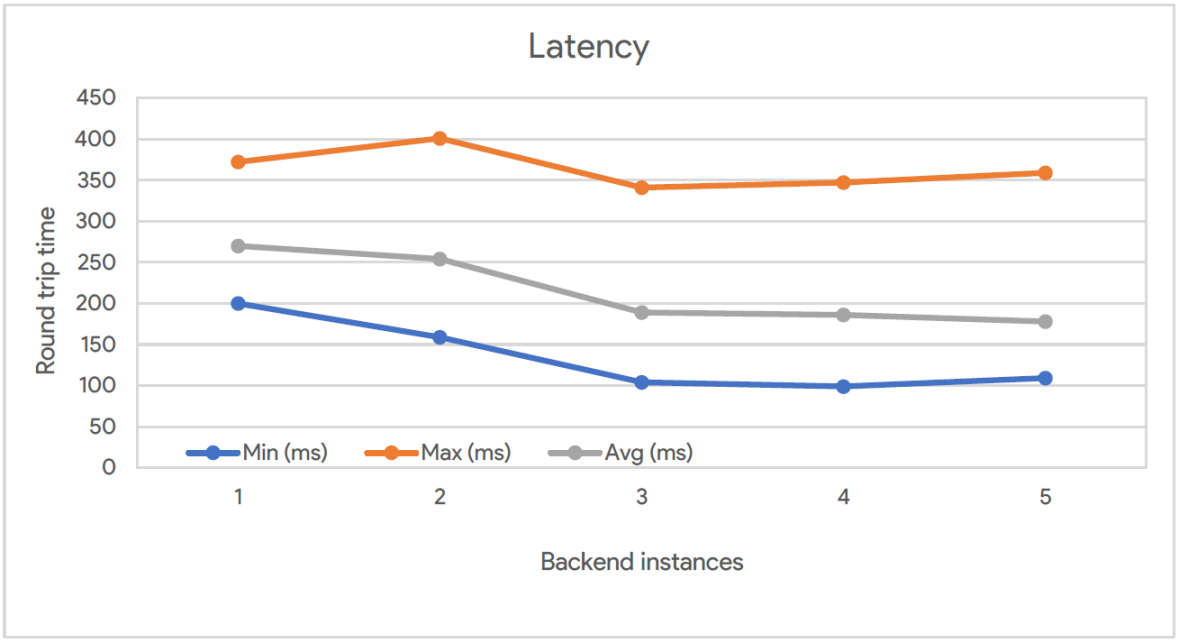

Figura 6.7: Numero delle richieste effettuate per secondo dal ruolo backend

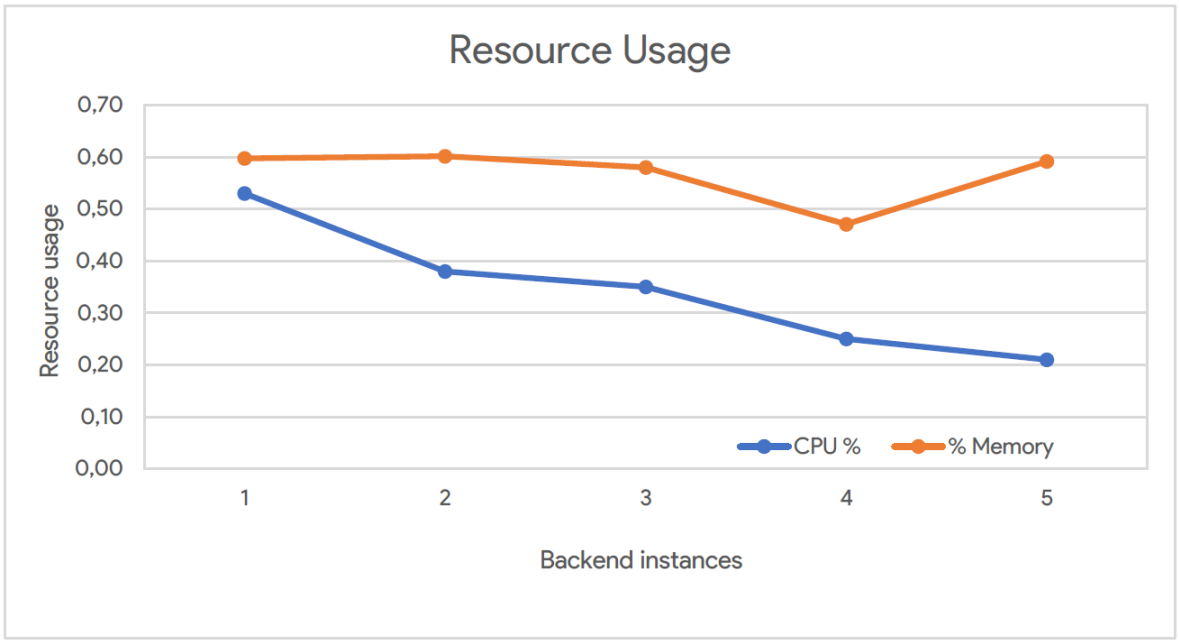

Figura 6.8: Utilizzo delle risorse per il ruolo backend

Conclusioni

Il framework sviluppato oggetto di questa tesi è stato realizzato rispettando i requisiti posti in fase di analisi. Tramite il framework sarà possibile sviluppare applicazioni distribuite che utilizzano i componenti dello stack SMACK. Sia in fase di progettazione sia durante la fase di sviluppo si è prestata attenzione a realizzare un progetto che rispettasse tutte le qualità del software, privilegiando le proprietà dei sistemi distribuiti.

Essendo un framework il progetto è riusabile per sua natura. Tutti i componenti del progetto sono facilmente estendibili e si adattano a qualsiasi logica applicativa. Si è cercato di realizzare un sistema il più modulare possibile incapsulando a dovere la logica di ciascun componente in modo tale da ridurre al minimo le dipendenze.

Il sistema distribuito che può essere realizzato con il framework è efficiente ed è affidabile. Sfruttando lo strumento realizzato per effettuare test sulle performance si è notato che anche la proprietà di scalabilità è stata rispettata. Il sistema distribuisce il carico di lavoro in maniera equa fra le risorse disponibili in modo tale da ridurre la latenza di ciascuna risposta. Anche le risorse hardware vengono utilizzate in maniera efficiente senza che si creino problemi di *overhead* quando si aumentano le risorse disponibili nel cluster.

Le caratteristiche e le funzionalità del framework sono facilmente verificabili tramite le tue tipologie di test adottate, ovvero i test unitari e i test d'integrazione. Il software è stato inoltre sviluppato utilizzando software di versionamento del codice e strumenti di integrazione continua, particolarmente adatti per il metodo di sviluppo seguito.

Sviluppi futuri

Anche se tramite il framework è possibile sviluppare ogni tipo di applicazione di qualsiasi complessità, al progetto mancano diverse funzionalità comuni a tutti i sistemi distribuiti. Queste funzionalità potrebbero essere generalizzate ed implementate nel sistema, in modo tale che chi utilizza il framework per sviluppare un'applicazione non le debba implementare.

Di seguito viene riportato un elenco degli eventuali lavori futuri che è possibile effettuare.

- Implementare un sistema di autenticazione degli utenti del sistema.
- Realizzare un meccanismo per l'autorizzazione di accesso alle risorse.
- Migliorare le API per interagire con il cluster Cassandra sostituendo il sistema attuale basato su query con un sistema di Data Modeling.
- Creare un sistema per raccogliere i log di tutte le macchine e per poterli analizzare da un unico servizio.
- Migliorare il sistema di validazione dei dati.
- Fornire un supporto completo alla localizzazione delle stringhe.
- Implementare un sistema per eseguire task automatici programmabili.
- Implementare altri meccanismi di interazione con gli utenti, come ad esempio quello delle WebSocket.
- Permettere il caricamento e la memorizzazione di file.
- Aggiungere sistemi efficienti di caching utilizzabili per sviluppare l'applicazione.

Appendice A

Installazione Apache Mesos

La configurazione riguarda due tipi di nodi: mesos-master e mesos-slave (o agent). La documentazione ufficiale per l'installazione e per la configurazione di Mesos è disponibile sul sito Apache Mesos[1].

A.1 Prerequisiti

Per fare riferimento a questa guida in ogni nodo deve essere installata una distribuzione Ubuntu 16.04. La maggior parte delle operazioni sono da effettuare con utente con privilegi root, si presuppone quindi che la shell sia avviata in modalità sudo. È consigliabile impostare l'hostname della macchina in modo che possa essere risolto da un server DNS esterno o utilizzando BIND in un nome di dominio valido e che l'indirizzo risolto possa essere raggiungibile dagli altri nodi del cluster. Per la configurazione del firewall è possibile fare riferimento all'Appendice C.

[1] http://mesos.apache.org/

A.2 Installazione

Installazione di Java 8

È innanzitutto necessario installare una versione di Java 8. Il modo più facile per installarlo è scaricarlo direttamente dal repository di default di Ubuntu, dove è contenuta l'ultima versione di OpenJDK 8. Per fare ciò è necessario aggiornare l'indice dei pacchetti ed installare il JDK.

```
apt-get update
apt-get install -y default-jdk
```

È consigliato invece installare la versione Oracle di Java 8, ma è necessario aggiungere prima ai repository locali l'archivio privato Oracle ed accettare la licenza.

```
add-apt-repository -y ppa:webupd8team/java
apt-get update
echo "oracle-java8-installer shared/accepted-oracle-license-v1-1 \\
  select true" | debconf-set-selections
apt-get install -y oracle-java8-installer
```

Occorre anche aggiungere la variabile d'ambiente che indica il percorso della directory di Java.

```
echo -e "JAVA_HOME=""/usr/lib/jvm/java-8-oracle""" >> /etc/environment
source /etc/environment
```

È possibile modificare la versione di Java da utilizzare nel seguente modo.

```
update-alternatives --config java
```

Installazione di Apache Mesos

Sia nei nodi master che nei nodi agent occorre installare l'ultima versione disponibile di Apache Mesos. È possibile scaricare i sorgenti di Mesos dal

sito ufficiale e compilarli manualmente. Le istruzioni per compilare i sorgenti ed installare Mesos in questo modo sono disponibili sul sito ufficiale di Mesos[2].

Per installare Mesos attraverso il repository di Ubuntu è necessario effettuare le seguenti operazioni.

```
apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" \
  | tee /etc/apt/sources.list.d/mesosphere.list
apt-get update
apt-get install -y mesos
```

La procedura per configurare i nodi master e i nodi agent è diversa. Sono forniti gli esempi per due tipi di configurazioni.

- **Un sono nodo master**. Nella seguente configurazione si assume che il master sia raggiungibile dagli altri nodi dall'indirizzo `master1.example.com`

- **Più nodi master**. È consigliabile utilizzare questo tipo di configurazione in ambienti di produzione e avere un numero dispari di nodi master (es. 3, 5, 7). Nella configurazione di esempio si assume che ci siano 3 nodi master raggiungibili dagli altri nodi dagli indirizzi `master[1-3].example.com`

Mesos Master

Per il coordinamento e il salvataggio di dati in modo permanente Mesos utilizza Apache Zookeeper, che viene installato in automatico quando si installa Apache Mesos attraverso il metodo indicato sopra. I file di configurazione di Zookeeper sono presenti nella directory `/etc/zookeeper/conf`. Occorre modificare le seguenti impostazioni.

[2]http://mesos.apache.org/documentation/latest/building/

- L'id del servizio Zookeeper. Deve essere un intero e deve essere unico all'interno del cluster. Aprire con un editor il file `myid` e inserire l'id scelto.

- Le impostazioni di Zookeeper contenute all'interno del file di configurazione `zoo.cfg`. È possibile visionare l'elenco di tutte le possibili opzioni nella documentazione Zookeeper[3]. Di seguito è mostrata la configurazione minima.

```
# la directory dove sono salvati gli snapshot
dataDir=/var/lib/zookeeper

# la porta alla quale i client si connettono
clientPort=2181

# gli altri server zookeeper, da impostare soltanto se si
# utilizzano più nodi master
# la prima porta è usata per connettersi al server leader
# la seconda porta è usata per l'elezione del leader
server.1=master1.example.com:2888:3888
server.2=master2.example.com:2888:3888
server.3=master3.example.com:2888:3888
```

Mesos per connettersi a Zookeeper deve conoscere tutti gli indirizzi dei servizi Zookeeper presenti. Aprire con un editor il file `/etc/mesos/zk` e inserire la stringa `zk://master1.example.com:2181/mesos` se si utilizza una configurazione con un solo nodo master, oppure
`zk://master1.example.com:2181,master2.example.com:2181,`
`master3.example.com:2181/mesos` se si utilizzano ad esempio 3 nodi master.

La configurazione del servizio master di Mesos è collocata nella directory `/etc/mesos-master`. Il nome di ogni file creato in questa directory verrà

[3]http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

passato come argomento durante l'esecuzione del servizio, mentre il contenuto verrà passato come valore. È possibile visualizzare la lista degli argomenti disponibili sulla documentazione sul sito di Apache Mesos[4]. Di seguito sono elencati i parametri principali.

- `quorum`: il numero minumo di nodi master attivi. È consigliabile rispettare la seguente regola: `quorum` > `(number of masters)/2`. Parametro **obbligatorio** solo nel caso in cui si scelga una configurazione con più nodi master.

- `work_dir`: il percorso della directory del master. Impostare `/var/lib/mesos`. Parametro **obbligatorio**.

- `cluster`: il nome da utilizzare per il cluster.

- `hostname`: il nome di dominio comunicato agli altri master. Se non viene specificato parametro `ip` o `advertise_ip`, il master deve essere raggiungibile dagli altri master attraverso questo valore. Di default viene utilizzato l'hostname del sistema.

- `ip`: l'indirizzo ip che gli altri master devono utilizzare per comunicare e che viene utilizzato al posto dell'`hostname`. Per ottenere l'indirizzo ip del sistema sull'interfaccia `eth0` utilizzare il comando `ifconfig eth0 | awk '/inet addr/split($2,a,":"); print a[2]'`.

- `advertise_ip`: l'indirizzo ip che gli altri master devono utilizzare per comunicare se risolvendo l'`hostname` non è possibile raggiungere il master.

Quando si installa Mesos tramite repository Ubuntu come nel procedimento indicato sopra viene impostato l'avvio in automatico del servizio `mesos-slave`. Disabilitare questo servizio nei nodi master utilizzando i seguenti comandi.

[4]http://mesos.apache.org/documentation/latest/configuration/master/

```
service mesos-slave stop
echo "manual" > /etc/init/mesos-slave.override
```

Occorre infine riavviare i servizi `zookeeper` e `mesos-master` dopo averne modificato la configurazione.

```
service zookeeper restart
service mesos-master restart
```

Collegandosi all'indirizzo `http://master1.example.com:5050` è disponibile l'interfaccia utente del master di Mesos, mostrata di seguito.

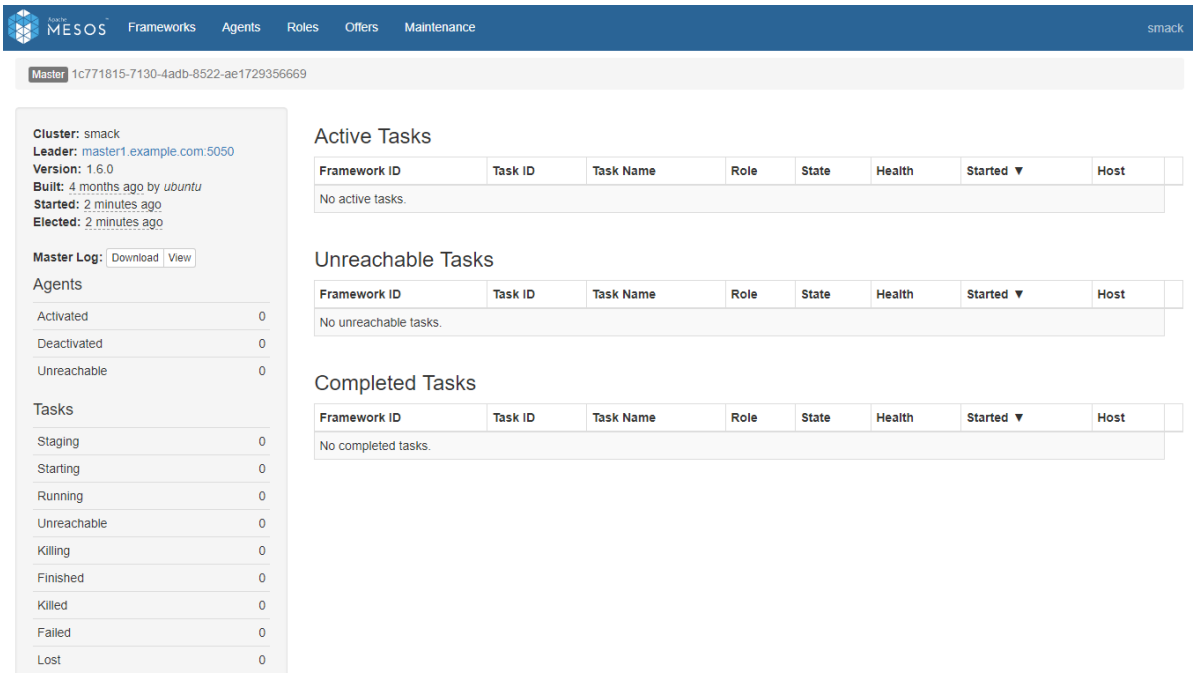

Figura A.1: Interfaccia utente del master di Mesos

Mesos Agent

Per connettersi ai nodi master gli agenti Mesos devono conoscere gli indirizzi Zookeeper dei master. È necessario modificare il file `/etc/mesos/zk` e inserire la stringa `zk://master1.example.com:2181/mesos` se si utilizza una configurazione con un solo nodo master, oppure

`zk://master1.example.com:2181,master2.example.com:2181,master3.example.com:2181/mesos` se si utilizzano ad esempio 3 nodi master.

La configurazione del servizio slave di Mesos è collocata nella directory `/etc/mesos-slave`. Il nome di ogni file creato in questa directory verrà passato come argomento durante l'esecuzione del servizio, mentre il contenuto verrà passato come valore. È possibile visualizzare la lista degli argomenti disponibili sulla documentazione sul sito di Apache Mesos[5]. Di seguito sono elencati i parametri principali.

- `work_dir`: il percorso della directory del master. Impostare `/var/lib/mesos`. Parametro **obbligatorio**.

- `hostname`: il nome di dominio comunicato al master e utilizzato dai task. Se non viene specificato parametro `ip` o `advertise_ip`, l'agente deve essere raggiungibile dal master attraverso questo valore. Di default viene utilizzato l'hostname del sistema.

- `ip`: l'indirizzo ip che il master deve utilizzare per comunicare e che viene utilizzato al posto dell'`hostname`. Per ottenere l'indirizzo ip del sistema sull'interfaccia `eth0` utilizzare il comando `ifconfig eth0 | awk '/inet addr/split($2,a,":"); print a[2]'`.

- `advertise_ip`: l'indirizzo ip che il master deve utilizzare per comunicare se risolvendo l'`hostname` non è possibile raggiungere l'agente.

- `containerizers`: la tipologia di container che i task possono utilizzare. Inserire `docker,mesos` per utilizzare sia container nativi Mesos sia container Docker.

Quando si installa Mesos tramite repository Ubuntu come nel procedimento indicato sopra viene impostato l'avvio in automatico dei servizi `zookeeper`

[5] http://mesos.apache.org/documentation/latest/configuration/agent/

e `mesos-master`. Disabilitare questi servizi nei nodi agent utilizzando i seguenti comandi.

```
service zookeeper stop
echo "manual" > /etc/init/zookeeper.override
service mesos-master stop
echo "manual" > /etc/init/mesos-master.override
```

Occorre infine riavviare il servizio `mesos-slave` dopo aver modificato la configurazione.

```
service mesos-slave restart
```

A.3 Conclusione

Il cluster Mesos è pronto per essere utilizzato. È ora possibile installare il framework Marathon per la gestione dei task.

Appendice B

Installazione Marathon

Marathon è un framework di Mesos mantenuto da Mesosphere. La documentazione ufficiale è disponibile sul sito Mesosphere Marathon.

B.1 Prerequisiti

Per fare riferimento a questa guida in ogni nodo master deve essere installata una distribuzione Ubuntu 16.04 e deve essere installato il servizio `mesos-master`, come indicato nella guida Installazione Apache Mesos riportata nell'Appendice A. La maggior parte delle operazioni sono da effettuare con utente con privilegi root, si presuppone quindi che la shell sia avviata in modalità sudo. Per la configurazione del firewall nell'Appendice C è riportata la lista delle porte utilizzate.

B.2 Installazione

Ci sono due modi per installare Marathon: scaricandolo dal sito ufficiale[1] o installandolo da repository Ubuntu. Se si sceglie la seconda opzione e si è già seguito il procedimento indicato nella guida Installazione Apache Mesos

[1] https://mesosphere.github.io/marathon/

riportata nell'Appendice A per l'aggiunta del repository dove è contenuto anche Mesos, saltare questo passaggio.

Nota: è necessario installare Marathon soltanto nei nodi dove è presente il servizio master di Mesos.

Per aggiungere il repository di Ubuntu dove è contenuto Marathon effettuare le seguenti operazioni.

```
apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" \
  | tee /etc/apt/sources.list.d/mesosphere.list
apt-get update
```

Per installare Marathon eseguire:

```
apt-get install -y marathon
```

La configurazione di Marathon è contenuta nella directory `/etc/marathon/conf`. Il nome di ogni file creato in questa directory verrà passato come argomento durante l'esecuzione del servizio, mentre il contenuto verrà passato come valore. Di seguito sono elencati soltanto i parametri obbligatori; tutte le possibili opzioni sono disponibili nella documentazione ufficiale nella sezione Command line flags[2].

- `hostname`: il nome di dominio comunicato a Mesos e alle altre istanze Marathon. Se non specificato viene utilizzato l'hostname del sistema.

- `master`: l'indirizzo Zookeeper del master di Mesos in una configurazione con singolo master (`zk://master1.example.com:2181/mesos`) o gli indirizzi Zookeeper dei master di Mesos in una configurazione con master multipli

[2]https://mesosphere.github.io/marathon/docs/command-line-flags.html

(`zk://master1.example.com:2181,master2.example.com:2181,master3.example.com:2181/mesos`).

- `zk`: l'indirizzo Zookeeper di tutti i nodi dove è eseguito il servizio Marathon. In una configurazione con singolo master indicare `zk://master1.example.com:2181/marathon`, mentre in una configurazione con master multipli indicare `zk://master1.example.com:2181,master2.example.com:2181,master3.example.com:2181/marathon`.

Per applicare la configurazione occorre riavviare il servizio `marathon`.

```
service marathon restart
```

Marathon per eseguire i task non utilizza l'utente root. In ogni agente mesos è necessario aggiungere un nuovo utente di nome `marathon`.

```
useradd -m -s /bin/bash marathon
```

B.3 Conclusione

Terminata l'installazione, connettendosi all'indirizzo `http://master1.example.com:8080` è possibile visualizzare l'interfaccia utente di Marathon.

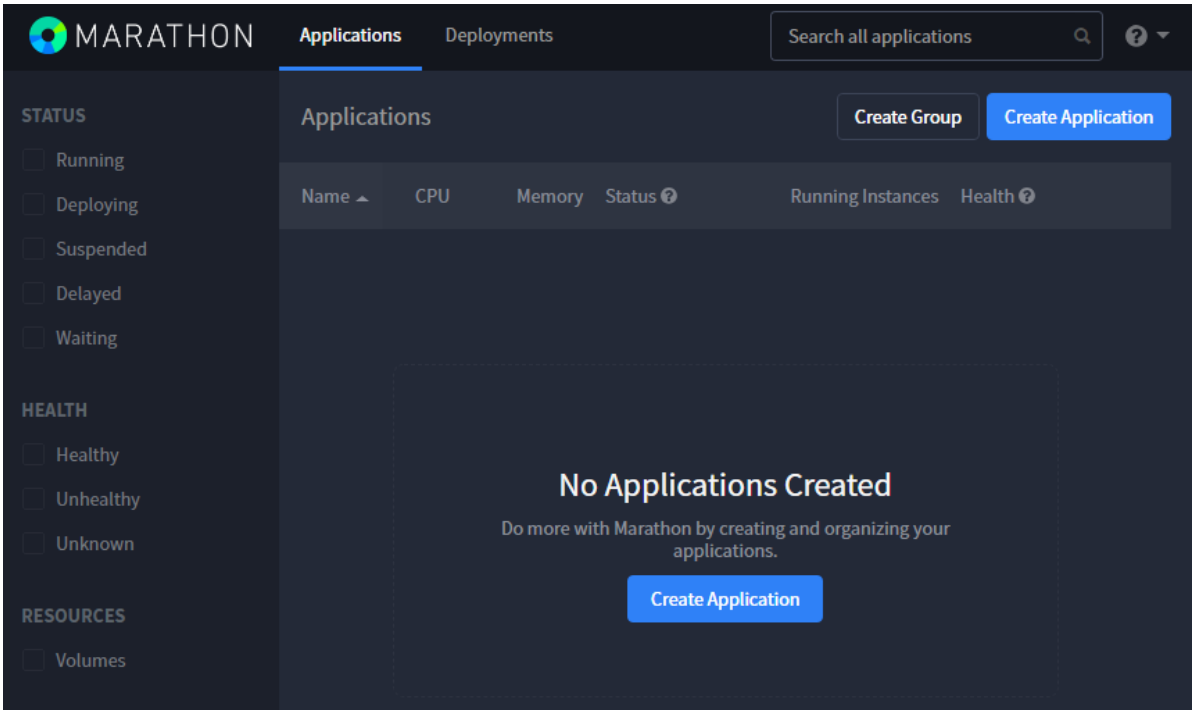

Figura B.1: Interfaccia utente Marathon

Appendice C

Configurazione firewall

Nella tabella di seguito sono elencate tutte le porte utilizzate dall'intero sistema.

Le porte da 1025 a 32000 possono venire scelta da Mesos durante l'allocazione dinamica delle porte.

Port	**Protocol**	**Service**	**Source**	**Destination**	**Description**
22	tcp	ssh	admin	master,agent	admin port
80	tcp	http	all	master	service port
443	tcp	https	all	master	service port
2181	tcp	zookeeper	master,agent	master	client port
2888	tcp	zookeeper	master	master	leader connection
3888	tcp	zookeeper	master	master	leader election
4040	tcp	spark	admin	master	admin port
5050	tcp	mesos-master	master,agent,admin	master	master port
5051	tcp	mesos-slave	master	agent	agent port
7000	tcp	cassandra	agent	agent	internal port

7199	tcp	cassandra	agent,jmx	agent	jmx port
8080	tcp	marathon	master,admin	master	admin port
8082	tcp	haproxy	admin	master	stats port
9042	tcp	cassandra	agent,app	agent	client port
9092	tcp	kafka-manager	admin	master	admin port
9092	tcp	kafka	agent,app	agent	client port
9093	tcp	kafka	agent,jmx	agent	jmx port
9160	tcp	cassandra	agent	agent	thrift port

Appendice D

Dispiegamento cluster

Per lanciare l'applicazione è necessario avere a disposizione un insieme di nodi che devono formare un cluster Mesos.

D.1 Prerequisiti

- Avere a disposizione un insieme di nodi che andranno a formare il cluster. È possibile utilizzare qualsiasi micro-servizio per la creazione di istanze. Viene spiegato come creare semplici droplet con DigitalOcean.
- Suddividere le risorse in nodi master e nodi agent e installare Apache Mesos (come indicato nell'Appendice A) su tutti i nodi.
- Installare Marathon (come indicato nell'Appendice B) sui nodi dove è stato installato il servizio mesos-master.
- Configurare il firewall (seguendo le regole riportate nell'Appendice C) per permettere ai nodi di dialogare.

Creazione droplet con DigitalOcean

Negli scripts allegati è presente un semplice tool per la creazione di droplet su DigitalOcean. Si chiama `droplets-compose` e ha due sotto comandi:

- `./droplets-compose up` <`num`> per la creazione di `num` droplet

- `./droplets-compose down <num>` per la distruzione di `num` droplet

A ciascuna istanza verrà assegnato il nome `mesos-agent$index`, con `$index` che va da 1 a `num`. I nodi `mesos-master` devono essere creati in precedenza manualmente. Ad ogni droplet viene associato un record di dominio che ha nome uguale a `mesos-agent$index.do.example.com`. È possibile modificare le opzioni del droplet e del record di dominio da creare rispettivamente nei file `create-droplet.json.template` e `create-domain-record.json.template`. Per eseguire lo script è necessario prima modificare alcune variabili d'ambiente all'interno del file `smack-env.sh`.

```
# Il token fornito da DigitalOcean per l'autenticazione
DO_TOKEN=""
# L'id della chiave ssh caricata su DigitalOcean
DO_SSH_KEY_ID=0
# L'id dell'immagine snapshot di mesos-agent creata in precedenza
DO_IMAGE_ID=0
# Il numero massimo di droplets che è possibile creare
MAX_DROPLETS=10
# Il dominio registrato su DigitalOcean da utilizzare come
# riferimento per le istanze
DO_DOMAIN="do.example.com"
```

D.2 Formazione del cluster

Dopo aver configurato tutti i nodi e avendo a disposizione risorse Mesos è possibile lanciare task su Marathon attraverso degli oggetti JSON. Gli oggetti per questa applicazione e per le sue dipendenze sono disponibili nella directory `marathon`. Sono disponibili anche tre script per la formazione automatica del cluster, e sono `cluster-init-compose`, `cluster-smack-compose` e `cluster-full-compose`. Ciascuno script contiene due sotto-comandi, `up` utilizzabile per la formazione del cluster e `down` per la sua distruzione. Di default, `cluster-init-compose` si occupa di:

- generare il cluster id, che viene utilizzato nel nome del cluster Cassandra (`cassandra_$id`) e come percorso Zookeeper di Kafka (`kafka_$id`);
- lanciare una istanza `kafka-seed` che si occupa della formazione e del congiungimento di broker Kafka;
- lanciare una istanza `cassandra-seed` che si occupa della formazione e del congiungimento di nodi Cassandra;
- lanciare due istanze `kafka-node` che portano il numero di istanze Kafka a 3;
- lanciare due istanze `cassandra-node` che portano il numero di istanze Cassandra a 3;
- eseguire la migrazione dello schema del database Cassandra, utilizzando `smack-migrate`;
- creare dei topic di Kafka con le relative partizioni.

Sempre di default, `cluster-smack-compose` si occupa di:

- lanciare una istanza `smack-seed` che si occupa di congiungere i nuovi nodi con i nodi già esistenti che formano il cluster;
- lanciare tre istanze `smack-frontend` che si occupano di avviare un'istanza di un web server per raccogliere le richieste dei client e di fornire loro una risposta;
- lanciare tre istanze `smack-backend` che si occupano di raccogliere le richieste degli utenti inviate al frontend e trasformarle in risposte;
- lanciare tre istanze `smack-service` che si occupano di processare le richieste che non vengono immediatamente processate dal backend.

Lo script `cluster-full-compose` non fa altro che eseguire `cluster-init-compose` prima e `cluster-smack-compose` dopo.

È possibile modificare i parametri di questi script modificando le variabili d'ambiente contenute all'interno del file `smack-env.sh`, mostrato di seguito.

```
# L'indirizzo del master dove è presente anche il servizio Zookeeper
MESOS_HOST=127.0.0.1
# La chiave privata di Sentry (opzionale)
SENTRY_DNS=""
# Il percorso locale dove sono contenuti i pacchetti in formato jar
# dell'applicazione
APP_PATH=/app
# Il percorso dove sono presenti i binari di Apache Kafka
KAFKA_HOME=/opt/kafka
# La versione corrente del progetto
VERSION="0.3.0-SNAPSHOT"
# Attiva o disattiva il debug dell'applicazione
DEBUG=false
# Imposta l'ambiente di produzione
ENVIRONMENT=production
# Imposta il livello dei log
LOG_LEVEL=warning
# Imposta il numero dei nodi Cassandra
CASSANDRA_NODE_INSTANCES=2
#Imposta il numero dei nodi Kafka
KAFKA_NODE_INSTANCES=2
# Imposta il numero delle istanze smack-frontend
SMACK_FRONTEND_INSTANCES=3
# Imposta il numero delle istanze smack-backend
SMACK_BACKEND_INSTANCES=3
# Imposta il numero delle istanze smack-service
SMACK_SERVICE_INSTANCES=3
# Il nome del topic relativo ai log
LOG_TOPIC="logs"
# Il numero delle partizioni del topic relativo ai log
LOG_PARTITIONS=3
# Il fattore replicante delle partizioni del topic relativo ai log
LOG_REPLICATION_FACTOR=2
```

Bibliografia

[1] H. Karau et al. *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015.

[2] D. Greenberg. *Building Applications on Mesos: Leveraging Resilient, Scalable, and Distributed Systems*. O'Reilly Media, 2015.

[3] R. Roestenburg, R. Bakker e R. Williams. *Akka in Action*. Manning, 2015.

[4] E. Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, 2016.

[5] N. Narkhede, G. Shapira e T. Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, 2017.

Elenco delle figure