ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UNA LIBRERIA IN SCALA DI SUPPORTO ALLA CREAZIONE E CONFIGURAZIONE DI UNO STACK SMACK

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. MIRKO VIROLI

Co-relatore
Dott. ROBERTO CASADEI

Presentata da
STEFANO SALVATORI

Terza Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Scala

SMACK

Software-Defined Infrastructure

Mesos

Docker

Indice

Il codice sviluppato è disponibile all'indirizzo
`https://github.com/StefanoSalvatori/SMACK-Library`.

Introduzione

Negli ultimi anni lo sviluppo dei sistemi IoT e delle applicazioni real-time, ha portato ad un grande cambiamento nel mondo delle applicazioni big data. Si è passati dai sistemi basati in gran parte su operazioni batch offline in cui i dati venivano acquisiti e archiviati periodicamente, all'analisi di informazioni in tempo reale, in cui più che di dati singoli, si preferisce parlare di flussi (o stream) di dati. L'esigenza di gestire tante informazioni e subito, ha potato quindi allo sviluppo di nuovi strumenti ed architetture per analizzare, processare e salvare i dati in maniera scalabile e con alte performance. In questo contesto la combinazione di Spark, Mesos, Akka, Cassandra e Kafka è diventata dominante per quanto riguarda lo sviluppo di applicazioni big data e molti parlano addirittura di un nuovo LAMP stack[1].

Anche se SMACK è diventato quindi uno stack molto popolare negli ultimi anni, rimangono comunque poche le tecnologie di supporto relative alla sua gestione e configurazione. Le risorse già esistenti online sono per lo più frammentarie e richiedono una conoscenza di basso livello riguardo scripting bash, reti e sistemi operativi. Molto spesso ci si ritrova per esempio a dover modificare dei file di testo con stringhe di configurazione oppure a dover scaricare diversi programmi, rendendo l'installazione un'operazione molto complessa, nella quale è facile compiere degli errori.

Lo scopo di questa tesi è quello di fornire una libreria che semplifichi la creazione e la gestione dello stack con la possibilità di interfacciarsi con questo via software tramite delle API in Scala. L'approccio scelto è quello che potremmo definire di Software-Defined Infrastructure (SDI). Con una Software-Defined Infrastructure i requisiti relativi all'infrastruttura di un'applicazione vengono dichiarati a livello software, contando sul fatto che l'hardware appropriato venga fornito in automatico da un'infrastruttura sottostante. La virtualizzazione delle risorse semplifica le operazioni di creazione e configurazione di un'infrastruttura fornendo inoltre maggiore flessibilità e tolleranza ai guasti.

In quest'ottica la libreria sviluppato permette, in primo luogo, di creare un cluster Mesos che si occupi di gestire le risorse hardware; e in secondo luogo

[1] `https://mesosphere.com/blog/smack-stack-new-lamp-stack/`

di definire su questo un'infrastruttura scalabile composta da Spark, Kafka e Cassandra. É stata strutturata in modo che queste operazioni richiedano la scrittura di poche righe di codice, producendo un guadagno in termini di tempo rispetto ad un'installazione manuale. Va comunque precisato che i metodi messi a disposizione permettono una configurazione "standard" dei componenti dello stack; si è preferito infatti non fornire un elevato grado di personalizzazione dell'infrastruttura per favorire l'usabilità e la semplicità di utilizzo.

Capitolo 1

Background

1.1 SMACK

SMACK è uno stack tecnologico utilizzato per lo sviluppo di applicazioni Big Data ad alte performance. Comprende una serie di strumenti open source che, combinati insieme, permettono di gestire in maniera efficiente grandi quantità di dati. Il termine è l'acronimo di:

- **Spark**: framework open source per il calcolo distribuito e l'analisi di dati su larga scala.
- **Mesos**: un sistema kernel distribuito per la gestione di cluster di computer.
- **Akka**: libreria open source per il modello di programmazione ad attori.
- **Cassandra**: database management system non relazionale distribuito con licenza open source e ottimizzato per la gestione di grandi quantità di dati.
- **Kafka**: piattaforma open source di stream processing di dati in tempo reale.

SMACK sta riscuotendo molto successo ultimamente in quanto fornisce agli sviluppatori e alle aziende in generale, una soluzione affidabile ed open source per l'utilizzo e l'analisi di big data (sistemi IoT, analisi di dati in tempo reale, analisi predittiva...). Uno schema dell'architettura ad alto livello di SMACK è mostrato in figura 1.1.

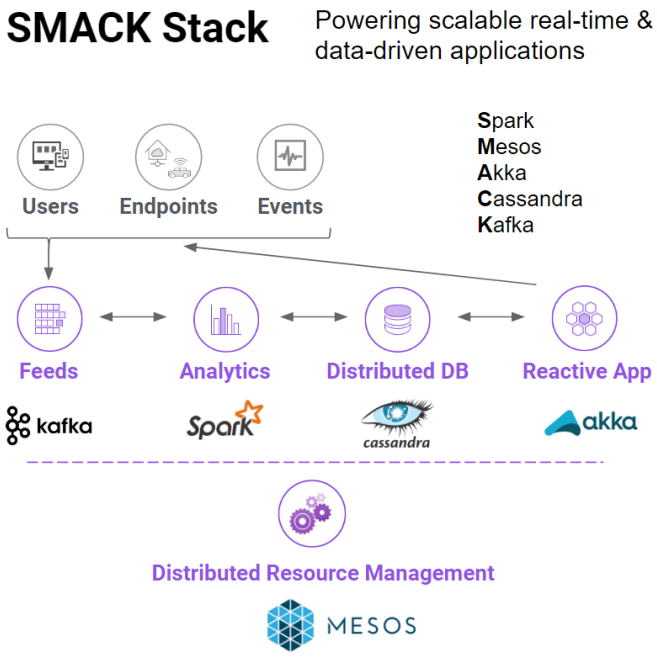

Figura 1.1: Esempio di infrastruttura SMACK[1]

1.1.1 Spark

Spark[1] è un framework open-source per il calcolo distribuito con svariate applicazioni in machine learning, stream processing ed applicazioni big data. Si basa su un'architettura di tipo master/worker ed al centro del suo funzionamento ci sono i cosiddetti Resilient Distributed Datasets o RDD. Un RDD è una collezione distribuita e immutabile di oggetti suddivisa in più partizioni all'interno di un cluster. Un RDD può essere creato parallelizzando una collezione di oggetti già esistente, oppure caricando dei dati dal file system. Un'applicazione che fa uso di spark si divide quindi sostanzialmente in 3 parti: creazione di un RDD, esecuzione di operazioni sull'RDD, eventuale salvataggio dei risultati.

Anche con le performance raggiunte dai soli RDD, spark rappresenta un ottimo strumento di elaborazione di dati, ma la sua popolarità si deve principalmente ad una funzionalità chiamata Spark Streaming. Spark Streaming fornisce infatti un'astrazione chiamata DStream o stream discretizzato che permette di gestire sequenze di dati nel tempo. Spark Streaming utilizza un'architettura "micro-batch", in cui le operazioni sullo stream vengono trattate come una serie continua di operazioni su piccoli batch di dati (figura 1.2).Ogni batch

[1]Apache Spark: `http://spark.apache.org/`

viene creato a intervalli di tempo regolari e tutti i dati che arrivano durante tale intervallo vengono aggiunti a quel batch. La dimensione degli intervalli di tempo è determinata da un parametro configurabile chiamato intervallo di batch (compreso in genere tra 500 millisecondi e alcuni secondi).

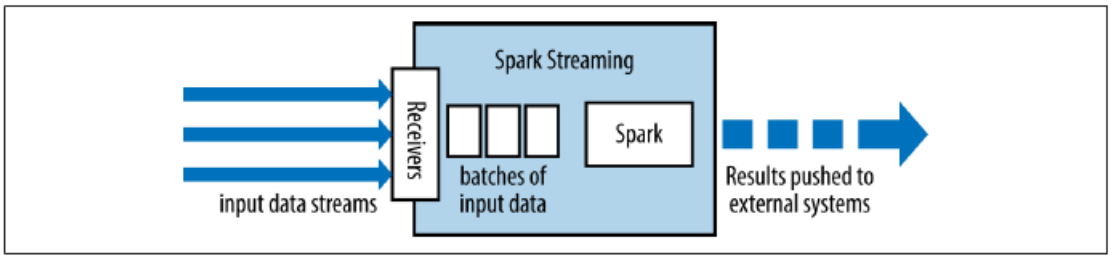

Figura 1.2: Architettura di Spark Streaming[1]

Spark può funzionare su un cluster hardware gestito da Apache Mesos con alcuni vantaggi che includono:

- partizionamento dinamico tra Spark e altri framework
- partizionamento scalabile tra più istanze di Spark

1.1.2 Mesos

Mesos[2] è una piattaforma open-source che permette di gestire CPU, memoria e altre risorse in un cluster[2]. Il motivo per cui viene utilizzato come cluster manager all'interno dello stack SMACK è che la sua architettura rende molto semplice scalare orizzontalmente applicazioni per disporre di più capacità computazionale.

É costituito da una serie di nodi che eseguono task, chiamati Agent, e da altri nodi, Master, che coordinano il cluster(figura 1.3). Le applicazioni eseguite su Mesos sono dette framework e si dividono in due componenti:

- uno scheduler, che comunica con il master per richiedere risorse di esecuzione
- un executor, che viene avviato sui nodi agent per eseguire le attività del framework

I master sono quindi responsabili di offrire risorse ai framework garantendo un'elevata disponibilità e un'allocazione efficiente. Mesos impone che solo un master alla volta sia eletto leader, così da essere in grado di calcolare e prendere

[2]Apache Mesos:`http://mesos.apache.org/`

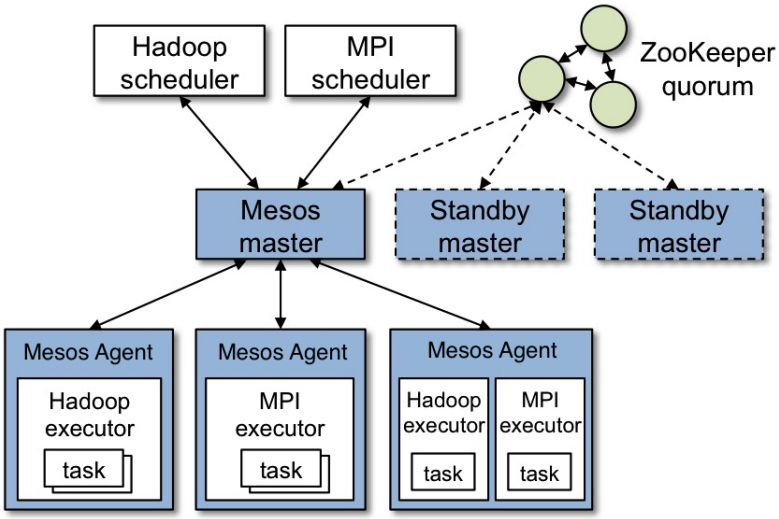

Figura 1.3: Architettura di Mesos[2]

decisioni senza essere rallentato. Gli eventuali altri master entrano quindi in funzione solo nel caso in cui, per qualche motivo, il leader corrente non sia più 'funzionante' (questa procedura viene chiamata leader election ed è gestita da Zookeeper).

Mentre il master determina quante risorse offrire a ciascun framework, gli scheduler selezionano quale delle risorse offerte utilizzare. Una volta scelta, passano a Mesos una descrizione dell'attività che vogliono eseguire, la quale, a sua volta, viene avviata sui nodi Agent.

Marathon

Marathon[3] è un framework Mesos progettato per l'avvio di applicazioni con una durata indefinita nel tempo. In particolare garantisce che queste continuino la loro esecuzione anche in caso di malfunzionamento dell'agente sul quale stanno girando. Ha molte caratteristiche che semplificano l'esecuzione di applicazioni in un ambiente cluster, come l'alta disponibilità, l'auto-recovery, la scalabilità e i controlli di stato. Un'applicazione all'interno di Marathon viene chiamata Task e può essere definita attraverso un file JSON che ne descrive il comportamento e le risorse necessarie all'esecuzione. Un'importante funzionalità di Marathon è la possibilità di eseguire applicazioni 'containerizzate', ovvero che vengono lanciate come container Docker all'interno di Mesos. Viene infatti spesso definito come una 'piattaforma di orchestrazione di container'.

[3]Apache Marathon: `https://mesosphere.github.io/marathon/`

Consente di monitorare le applicazioni in esecuzione attraverso un API REST che comprende varie funzioni tra cui:

- creazione, esecuzione e arresto dei task sul cluster
- lista di tutti i task in esecuzione
- recupero delle informazione relative ad uno specifico task

Zookeeper

Sia Marathon che Mesos richiedono uno strumento esterno per funzionare chiamato Zookeeper[4]. ZooKeeper è un servizio centralizzato per il mantenimento delle informazioni di configurazione, la denominazione e la sincronizzazione in sistemi distribuiti. Offre un servizio ad alte performance e viene spesso utilizzato per la gestione della leader election in sistemi fault-tolerance.

1.1.3 Akka

Akka[5] è una libreria open source per costruire applicazioni concorrenti e distribuite su JVM. Supporta diversi modelli di programmazione concorrente ma enfatizza quello ad attori. Gli attori sono oggetti concorrenti che comunicano attraverso lo scambio di messaggi asincrono; sono caratterizzati da

- una coda di ricezione dei messaggi
- un comportamento (stato dell'attore, variabili interne)
- un protocollo di ricezione e risposta ai messaggi
- un ambiente di esecuzione
- un indirizzo

vengono organizzati gerarchicamente in strutture chiamate Actor System, ovvero gruppi di attori che condividono le medesime opzioni di configurazione. Lo scopo del sistema ad attori è suddividere i compiti e delegarli, in modo che ognuno abbia una minima parte di computazione da eseguire. Caratteristica fondamentale del modello ad attori è che l'invio di un messaggio non comporta il passaggio del flusso di controllo dal mittente al destinatario, rendendo possibile l'esecuzione concorrente di più task. Akka è diventata ormai uno strumento standard per la creazione rapida di applicazioni reattive, concorrenti e distribuite; per questo motivo è un componente molto importante dello stack SMACK.

[4]Apache Zookeeper: `https://zookeeper.apache.org/`
[5]Akka: `https://akka.io/`

1.1.4 Cassandra

Cassandra[6] è un database management system distribuito non relazionale eseguito su un cluster di nodi omogenei. Nell'ambito del CAP theorem è classificato come un sistema AP in cui disponibilità (availability) e tolleranza alle partizioni (partition tolerance) sono mantenuti a discapito della coerenza. É comunque possibile configurare Cassandra per soddisfare la consistenza dei dati tramite un parametro chiamato consistency level. Questo determina infatti il numero di nodi che devono confermare una modifica, prima che questa possa essere notificata ad un eventuale client (QUORUM è un livello di consistenza comunemente usato che indica 'la maggior parte dei nodi';può essere calcolato utilizzando la formula (n / 2 +1) dove n è il fattore di replicazione).

In Cassandra, i record all'interno del database vengono distribuiti in modo uniforme tra tutti i nodi partecipanti con un partizionamento basato su hashing dei dati. Per aumentare la tolleranza ai guasti, Cassandra dà la possibilità di creare copie dei dati, note come repliche, in modo che ogni record sia salvato su più macchine all'interno del cluster. Per quanto riguarda l'architettura interna Cassandra utilizza un protocollo di gossip per la comunicazione intra cluster e il rilevamento degli errori. Un nodo scambia informazioni di stato con un massimo di altri tre nodi. Le informazioni di stato vengono scambiate ogni secondo e contengono informazioni su se stesso e su tutti gli altri nodi conosciuti. Ciò consente a ciascun nodo di apprendere informazioni sugli altri anche se comunica con un piccolo sottoinsieme di nodi.

Cassandra rappresenta quindi un ottimo strumento quando si ha bisogno di scalabilità, alte prestazioni e disponibilità continua; per questo motivo viene utilizzato come DBMS all'interno dello stack SMACK.

1.1.5 Kafka

Apache Kafka[7] è una piattaforma di streaming distribuita scritta in Scala e Java eseguibile su un cluster di uno o più server. L'obiettivo è quello di fornire un servizio ad alto throughput e bassa latenza per gestire feed di dati in tempo reale. Mette a disposizione tre funzionalità chiave:

- Publish e Subscribe di messaggi.
- Memorizzazione di flussi di record in modo duraturo e tollerante ai guasti.
- Elaborazione dei flussi di record man mano che si verificano.

Viene generalmente utilizzato per due ampie classi di applicazioni:

[6]Apache Cassandra: `http://cassandra.apache.org/`
[7]Apache Kafka: `https://kafka.apache.org/`

- Realizzazione di pipeline di flussi di dati in tempo reale che ottengono in modo affidabile dati da sistemi o applicazioni
- Creazione di applicazioni di streaming in tempo reale che trasformano o reagiscono ai flussi di dati

Alla base del suo funzionamento ci sono i topic: un topic è un nome di categoria in cui i messaggi vengono pubblicati. Un topic può avere zero, uno o più consumatori che leggono i dati scritti su di esso. Per ogni topic, Kafka mantiene un log di partizioni come mostrato in figura 1.4.

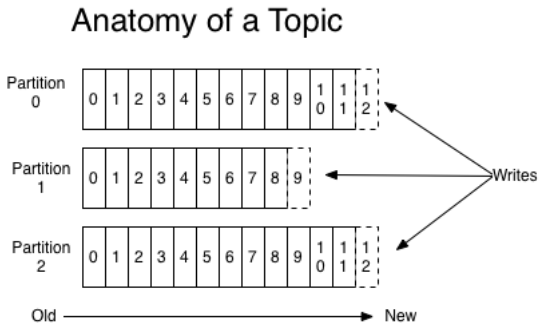

Figura 1.4: Log delle partizioni di Kafka[3]

Una partizione è una sequenza ordinata e immutabile di record. Ogni partizione viene replicata su un numero configurabile di server per la tolleranza agli errori ed ha un server che funge da "leader" e zero o più server che fungono da "follower". Il leader gestisce tutte le richieste di lettura e scrittura per la partizione mentre i follower replicano passivamente il leader. Se il leader fallisce, uno dei follower diventerà automaticamente il nuovo leader (leader election tramite Zookeeper). Ogni server funge da leader per alcune delle sue partizioni e follower per altre, quindi il carico è ben bilanciato all'interno del cluster.I record all'interno delle partizioni sono unicamente identificati da un numero ID sequenziale chiamato offset. Il cluster Kafka mantiene in memoria i record pubblicati, indipendentemente dal fatto che siano stati consumati o meno, utilizzando un periodo di conservazione configurabile. Ad esempio, se il periodo di conservazione è impostato su due giorni, per i due giorni successivi alla pubblicazione di un record, esso sarà disponibile per il consumo, dopodiché verrà scartato per liberare spazio. Le prestazioni di Kafka sono effettivamente costanti per quanto riguarda le dimensioni dei dati, quindi la memorizzazione su lungo periodo di tempo non è un problema.

Kafka però non viene utilizzato all'interno di SMACK solamente per le sue capacità di leggere, scrivere e archiviare di dati, ma anche perché consente un

l'elaborazione di questi in tempo reale e con alte performance. Mette infatti a disposizione una funzionalità per lo streaming chiamata stream processor, che prende in input flussi continui di dati, esegue alcune elaborazioni su questo input e produce flussi di dati continui in output. Ciò consente di creare applicazioni real-time che eseguono elaborazioni non banali calcolando aggregazioni di flussi o eseguendo operazioni sui dati con alte performance.

1.2 Scala

Scala[8] è un linguaggio di programmazione general-purpose che unisce il paradigma ad oggetti con quello funzionale. Alcuni dei punti di forza del linguaggio sono:

- Chiarezza: Incentivando la scrittura di codice senza *side effects*, il paradigma funzionale porta a scrivere codice su cui è più semplice fare debugging, più facile da testare e più facile da riutilizzare.
- Brevità: Scala mette a disposizione una serie di costrutti e possibilità sintattiche che rendono il codice più breve e conciso.
- Interoperabilità con Java: La compilazione del codice sorgente produce Java bytecode, questo fa si che Scala sia completamente compatibile con del codice Java.

É stato introdotto per la prima volta agli utenti nel 2003 ed ha iniziato negli ultimi anni a sviluppare un seguito significativo tra gli sviluppatori grazie ad alcune caratteristiche che lo rendono particolarmente adatto al modello di programmazione su Big Data. Per questo motivo la S di SMACK, oltre che a Spark, viene spesso associata a Scala. Anche se trova quindi il suo più naturale utilizzo all'interno di applicazioni real-time o applicazioni data intensive, Scala rimane un linguaggio general-purpose e può essere quindi utilizzato per vari scopi come:

- Sviluppo di applicazioni web
- Applicazioni che lavorano con stream di dati
- Applicazioni distribuite
- Sviluppo di Domain Specific Lenguages

[8]Scala: `https://www.scala-lang.org/`

1.3 Docker

Docker[9] è un progetto open source che consente la creazione e l'utilizzo di container per il deployment di applicazioni. Utilizza il kernel del sistema operativo e le sue funzionalità per isolare i processi in modo da poterli eseguire in maniera indipendente. Questa indipendenza è l'obiettivo dei container: la capacità di eseguire più processi e applicazioni in modo separato per sfruttare al meglio l'infrastruttura esistente pur conservando il livello di sicurezza che sarebbe garantito dalla presenza di sistemi separati.

Gli strumenti per la creazione di container, come Docker, consentono il deployment a partire da un'immagine[10]. Ciò semplifica la condivisione di un'applicazione o di un insieme di servizi, con tutte le loro dipendenze, nei vari ambienti. Docker automatizza anche la distribuzione dell'applicazione (o dei processi che compongono un'applicazione) all'interno dell'ambiente containerizzato. Gli strumenti sviluppati partendo dai container, responsabili dell'unicità e della semplicità di utilizzo di Docker, offrono agli utenti accesso alle applicazioni, la capacità di eseguire un deployment rapido, e il controllo sulla distribuzione di nuove versioni. L'utilizzo di Docker per creare e gestire i container può semplificare la creazione di sistemi distribuiti, permettendo a diverse applicazioni o processi di lavorare in modo autonomo sulla stessa macchina fisica o su diverse macchine virtuali. Ciò consente di effettuare il deployment di nuovi nodi solo quando necessario, permettendo uno stile di sviluppo del tipo platform as a service (PaaS) per sistemi come Apache Cassandra, MongoDB o Riak.

1.4 Tecnologie esistenti per l'utilizzo di SMACK

Non esistono attualmente strumenti creati appositamente per l'installazione e la configurazione automatica di uno stack SMACK. Ci sono però varie risorse in rete che offrono un servizio più generale che può essere utilizzato per la creazione dello stack. Vengono elencati qui le principali.

1.4.1 Mesosphere DC/OS

DC/OS[11] è un sistema operativo distribuito basato su Apache Mesos. Consente la gestione di più macchine come se fossero un singolo computer. Automatizza la gestione delle risorse, pianifica il posizionamento dei processi,

[9] Docker: `https://www.docker.com/`

[10] Un'immagine è un file che include tutti i componenti necessari per eseguire un'applicazione: codice, librerie, variabili di ambiente e file di configurazione.

[11] Mesosphere DC/OS: `https://mesosphere.com/`

facilita la comunicazione tra task e semplifica l'installazione e la gestione di servizi distribuiti. Tra le varie funzionalità che mette a disposizione c'è quella di eseguire servizi, tra cui Spark, Cassandra e Kafka, all'interno del sistema operativo distribuito tramite l'installazione di pacchetti. Per questo motivo DC/OS può essere un valido strumento per l'installazione e la configurazione di uno stack SMACK.

1.4.2 Amazon ECS

Amazon Elastic Container Service (Amazon ECS)[12] è un servizio di orchestrazione di contenitori altamente dimensionabile ad elevate prestazioni che supporta i contenitori Docker e consente di eseguire e ridimensionare facilmente le applicazioni suddivise in contenitori su AWS. Amazon ECS consente di sviluppare qualsiasi tipo di applicazione in contenitori, dalle applicazioni a lunga esecuzione e microservizi alle applicazioni per attività in batch e per Machine Learning. É quindi possibile configurare uno stack SMACK completo incapsulando ogni suo componente in un container docker che sarà gestito poi dal servizio di Amazon.

[12] Amazon ECS: `https://aws.amazon.com/it/ecs/`

Capitolo 2

Analisi dei Requisiti

2.1 Requisiti Funzionali

Le funzioni messe a disposizione dalla libreria saranno:

1. **Installazione e configurazione automatica dei componenti**

   I componenti necessari al corretto funzionamento di SMACK verranno automaticamente installati nelle macchine che andranno a formare il cluster. Si provvederà ad:

   - Installare e configurare i file Mesos
   - Installare e configurare Marathon
   - Installare e configurare Zookeeper
   - Installare Spark
   - Installare Scala
   - Installare Docker

2. **Creazione di un cluster Mesos**

   Un cluster è caratterizzato da nome simbolico, lista di nodi master e lista dei nodi Agent. Sarà possibile crearlo in 2 modalità equivalenti:

   - Attraverso delle chiamate via software
   - Tramite un file di configurazione in formato JSON

3. **Esecuzione di task all'interno del cluster**

   Ogni task è definito da: ID, numero di CPU richieste per l'esecuzione, memoria RAM disponibile, spazio su disco riservato, un set di variabili d'ambiente e numero di istanze da eseguire; può rappresentare

- un comando bash: richiede che sia specificata la stringa che si vuole eseguire
- un container Docker: in questo caso è necessario fornire il nome dell'immagine docker

4. **Setup e configurazione di un cluster Cassandra**

   Sarà possibile creare un cluster Cassandra su Mesos scegliendo il nome ed il numero di nodi che apparterranno al cluster. Si potrà decidere anche quante CPU e quanta memoria assegnare ad ogni nodo.

5. **Setup e configurazione di un cluster Kafka**

   In maniera analoga, sarà possibile creare un cluster Kafka definendo il numero di broker e quante risorse assegnare ad ognuno.

6. **Setup e configurazione di Spark**

   Verrà messa a disposizione una funzionalità per eseguire il framework che gestisce automaticamente l'interazione tra Spark e Mesos. In questo modo funzionerà come cluster manager per Spark e gli agenti Mesos potranno essere utilizzati come esecutori Spark.

7. **Scalabilità real-time**

   Sarà possibile scalare orizzontalmente l'infrastruttura potendo in particolare:

   - Incrementare il numero di agenti Mesos (e contemporaneamente di esecutori Spark)
   - Incrementare il numero di nodi Cassandra
   - Incrementare il numero di broker Kafka

2.2 Requisiti non funzionali

1. **Tolleranza ai guasti**

   Il malfunzionamento di una agente Mesos non comprometterà, per quanto possibile, il funzionamento di Cassandra, Spark o Kafka al suo interno (vedi figura 2.1) e sarà possibile configurare Mesos per funzionare con più di un master.

2. **Usabilità**

   Un altro punto da tenere presente sarà la facilità di utilizzo della libreria sviluppata; questa dovrà essere semplice ed essenziale in modo che,

poche righe di codice, permettano di costruire un infrastruttura funzionante. L'utilizzo della libreria dovrà effettivamente fornire un guadagno in termini di tempo e risorse rispetto alla configurazione manuale dello stack.

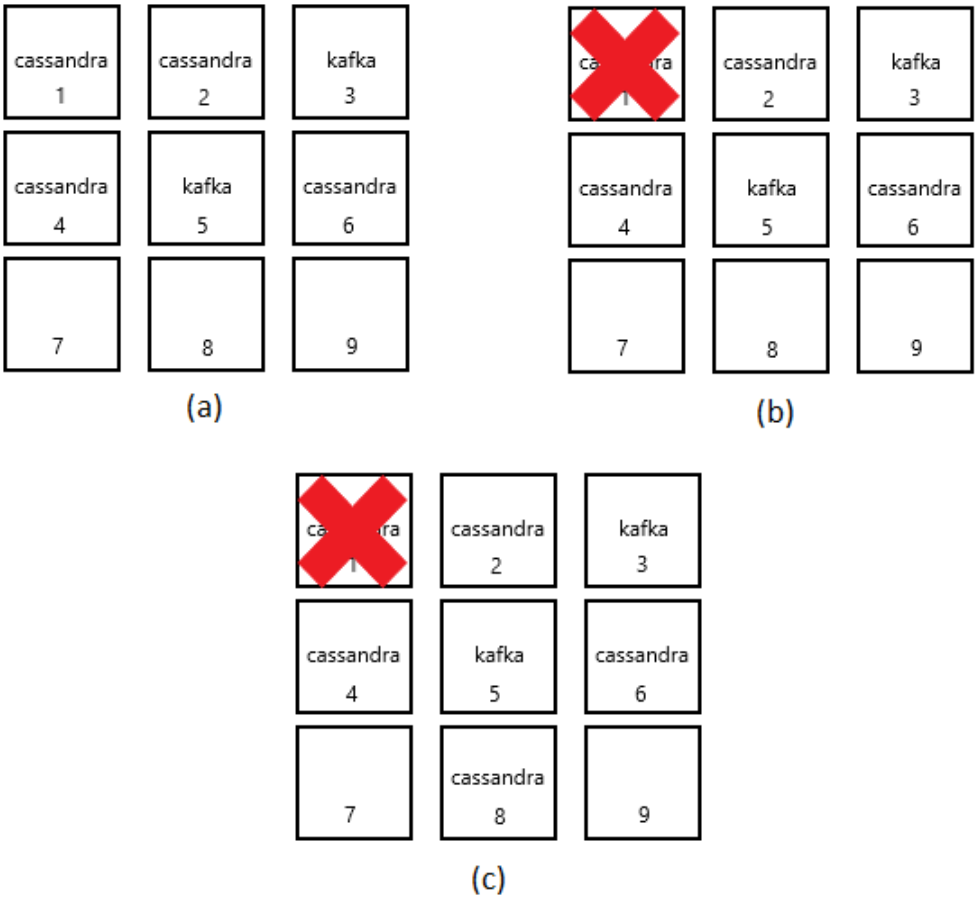

Figura 2.1: Esempio di recupero da uno stato di malfunzionamento di uno nodo: (a) Cluster funzionante con 2 broker Kafka e 4 nodi Cassandra; (b) Il nodo 1 ha un malfunzionamento e non permette all'istanza di Cassandra di continuare; (c) Il processo che era attivo sul nodo 1 viene automaticamente recuperato su un altro nodo libero (8 in questo caso)

Capitolo 3

Progettazione

Prima di procedere con la descrizione dell'architettura generale della libreria, vale la pena discutere e soffermarsi sulle caratteristiche dell'infrastruttura SMACK presa come riferimento durante la fase di progettazione (figura 3.1) che hanno poi influito in molte delle scelte effettuate.

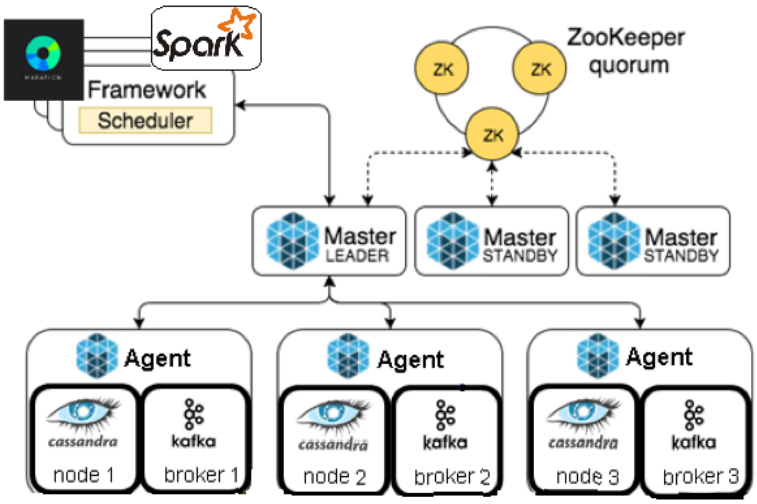

Figura 3.1: Infrastruttura generale dello stack. Cassandra e Kafka vengono eseguiti come container docker (rappresentati con i riquadri neri all'interno degli agent) attraverso Marathon; Spark utilizza un framework separato per l'interazione con il cluster.[4]

Una prima caratteristica da notare è che si è deciso di trattare i cluster Cassandra e Kafka come " virtuali": i nodi e i broker al loro interno non

girano fisicamente su una specifica macchina, ma vengono invece eseguiti su Mesos attraverso dei container Docker gestiti dal framework Marathon. In questo modo l'astrazione e l'isolamento fornito dai container, permette loro di non essere "vincolati" ad un agente in particolare, rendendo allo stesso tempo possibile un'allocazione efficiente delle risorse del cluster. Grazie alle funzionalità di recovery messe a disposizione da Marathon, viene inoltre garantita una tolleranza agli errori aggiuntiva: in caso di malfunzionamento di un agente, una particolare immagine docker verrebbe automaticamente recuperata su un altro nodo disponibile nel cluster. Per quanto riguarda l'esecuzione di Spark, si è invece scelto di utilizzare un approccio differente: non vengono eseguiti, come per Cassandra e Kafka, vari container all'interno del cluster, ma, come si vede dalla figura, viene utilizzato un framework che gestisce automaticamente l'interazione tra Spark e Mesos. I Mesos Agent possono essere in questo modo utilizzati come 'esecutori' Spark.

Avendo quindi in mente questo tipo di architettura, andiamo ora a descrivere la struttura e gli schemi UML della libreria.

3.1 Cluster e Nodi

L'interfaccia Cluster visibile in figura 3.2 modella il concetto di cluster come insieme di nodi logicamente connessi. Un Nodo, definito nella classe Node, rappresenta una macchina fisica o virtuale che abbia uno specifico indirizzo IP e sulla quale sia possibile eseguire comandi tramite SSH. Rifacendosi all'architettura di Mesos, si è scelto di mantenere all'interno dell'interfaccia Cluster la suddivisione tra nodi Master e nodi Agent.

Un cluster è responsabile della configurazione dei nodi al suo interno e mette a disposizione le funzionalità di base per:

- creare e distruggere un cluster
- eseguire dei task
- aggiungere e rimuovere nodi

L'interfaccia Cluster viene implementata dalla classe MesosCluster, che incapsula l'effettivo funzionamento di Mesos. In questo caso è stato ritenuto opportuno dare la possibilità di utilizzare un builder per la creazione di un MesosCluster in modo da semplificare la scrittura di codice ed aumentarne la leggibilità (figura 3.3). Oltre al builder, un altro modo rapido per la creazione di un cluster è l'utilizzo di un file in formato JSON in cui sono specificati i vari parametri di configurazione (nome, ip dei nodi Master, ip dei nodi Agent...).

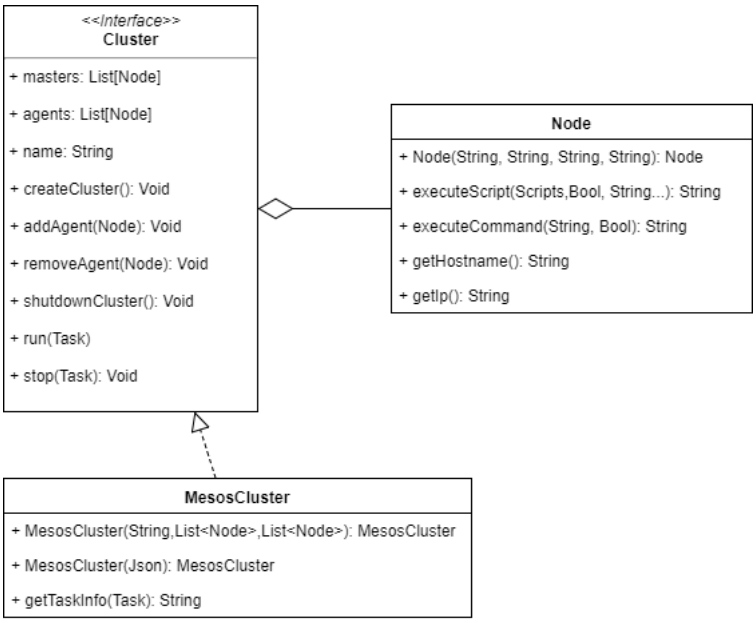

Figura 3.2: Diagramma UML dell'interfaccia Cluster e la classe Node.

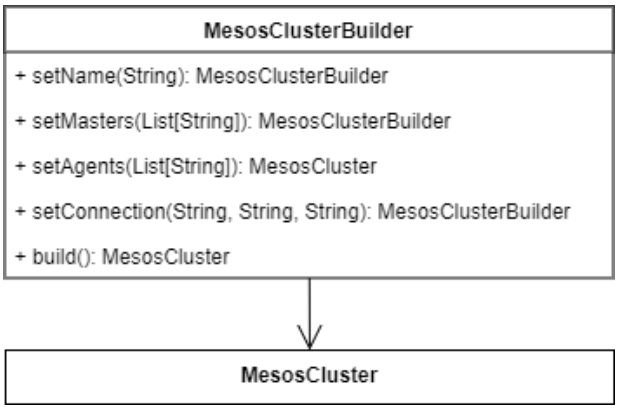

Figura 3.3: Builder per la classe MesosCluster

3.2 Task

I task sono stati modellati all'interno della libreria attraverso l'interfaccia Task; rappresentano oggetti eseguibili su Mesos tramite Marathon e vengono descritti da vari parametri tra cui: numero di cpu, quantità di memoria, spazio su disco ecc...Possono essere utilizzati anche come esecutori di container Docker e possono essere salvati come file JSON. L'implementazione dell'interfaccia avviene tramite la classe GenericTask, per la quale, visto l'elevato numero di parametri e la possibilità che alcuni di questi abbiano un valore di default,

viene messo a disposizione un builder TaskBuilder. Una visione più dettagliata delle relazioni fra i componenti descritti è mostrata in figura 3.4.

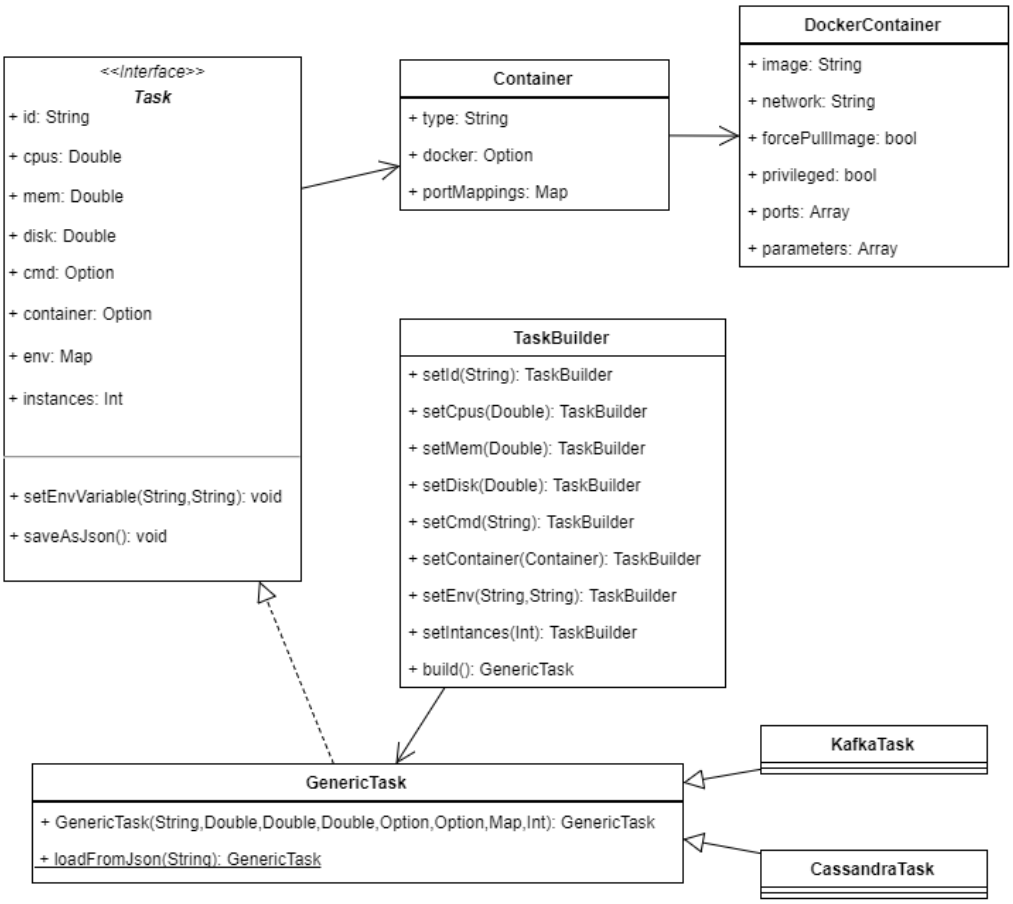

Figura 3.4: Diagramma UML dell'architettura dei Task.

Nello schema sono visibili le 2 classi di tipo GenericTask: CassandraTask e KafkaTask che gestiscono l'esecuzione di Cassandra e Kafka come task Marathon. Eseguire uno di questo task corrisponde a lanciare le immagini docker relative a un nodo Cassandra o un broker Kafka. Vengono mostrate qui per sottolineare quanto detto in precedenza riguardo la virtualizzazione: un cluster Cassandra o Kafka infatti, può essere definito a questo punto via software configurando opportunamente più istanze di queste classi, senza preoccuparsi della struttura sottostante; le risorse necessarie al funzionamento vengono infatti automaticamente fornite dai container Docker. Naturalmente per essere effettivamente funzionanti dovranno venire eseguite su un cluster, ma ne saranno tecnicamente indipendenti.

3.3 MarathonCluster

L'interfaccia MarathonCluster (figura 3.5), rappresenta quanto appena detto riguardo i cluster virtuali. L'idea è quella di fornire un livello di astrazione per cui un gruppo di task Marathon che interagiscono tra loro, viene percepito come un cluster fisico.

Le due classi CassandraCluster e KafkaCluster visibili in figura implementano appunto questo comportamento in relazione ai cluster Cassandra e Kafka.

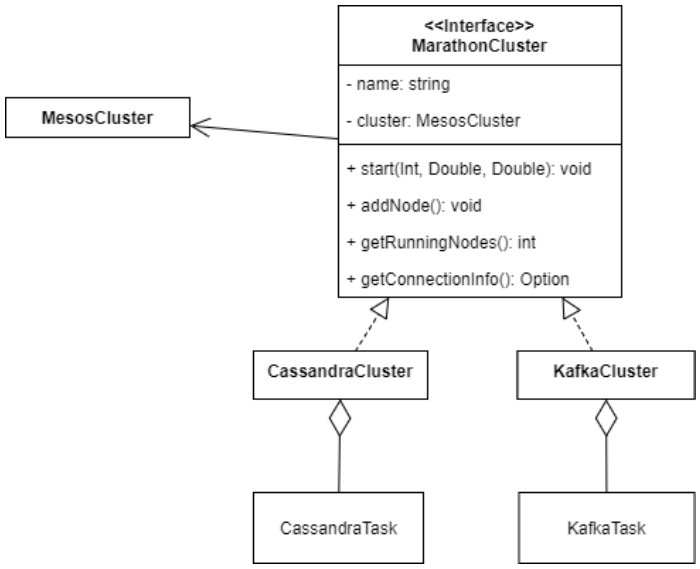

Figura 3.5: Struttura della classe MarathonCluster

3.4 Interfaccia per SMACK

Per dare la possibilità di interagire più facilmente con il sistema descritto, è stata definita una classe Facade[1] SmackEnvironment (figura 3.6) che mette a disposizione le principali funzioni di scalabilità e gestione di uno stack SMACK utilizzando le classi descritte in precedenza.

In particolare con questa classe è possibile:

- Eseguire il framework Spark
- Dichiarare i cluster Cassandra e Kafka da lanciare su Mesos

[1]Il pattern Facade ha come intento quello di fornire un'interfaccia unificata per un insieme di interfacce in un sottosistema semplificandone l'utilizzo.[3]

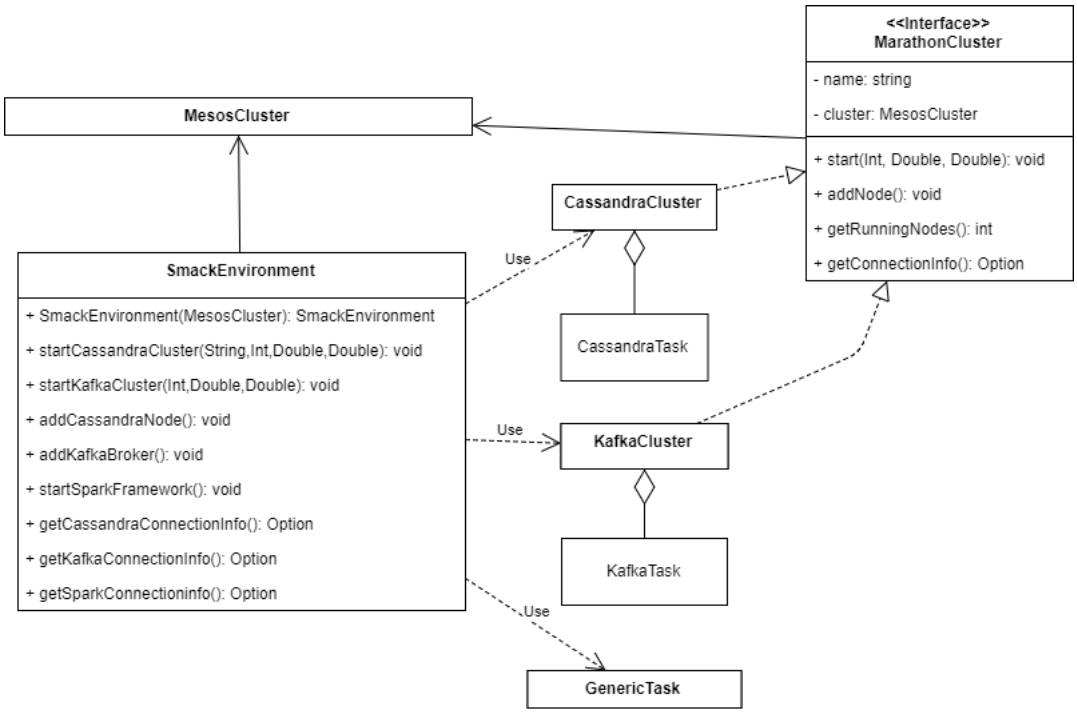

Figura 3.6: Diagramm UML del pattern Facade relativo alla classe SmackEnvironment.

- Aumentare il numero di nodi dei cluster
- Avere le informazioni di connessione (indirizzi ip e porte)

Utilizzando il Facade pattern la configurazione “standard” di SMACK viene quindi molto semplificata risultando praticamente immediata nella scrittura di codice. É importante però sottolineare che la classe SmackEnvironment non fornisce un alto livello di ’personalizzazione’; in caso di particolari esigenze, è comunque possibile costruirsi la propria infrastruttura definendo opportunamente i task da eseguire sul cluster.

Capitolo 4

Implementazione

4.1 Scala

In questa fase dello sviluppo della libreria si è passati dalla progettazione language independent con la quale sono stati prodotti i moduli nella sezione 3, all'effettiva implementazione in Scala. Questo non ha cambiato naturalmente la logica dei componenti, ma ha portato a dover effettuare delle scelte per quanto riguarda la traduzione in codice dei moduli. In particolare:

1. Gli stereotipi *interface* sono stati implementati attraverso dei trait Scala

2. I costruttori delle classi sono stati inseriti come factory method nei companion object delle rispettive classi

4.2 Cluster

Nella prima fase dello sviluppo ci si è concentrati sulle funzionalità relative alla creazione e configurazione del cluster.

4.2.1 Installazione automatica dei componenti

La prima funzionalità implementata è quella che permette di creare in maniera automatica un cluster Mesos attraverso la classe `MesosCluster`. Inizialmente si è posto il problema di come configurare da remoto i nodi che avrebbero fatto parte del cluster: si è scelto di utilizzare la libreria Java Ganymed-SSH2[1] che fornisce delle API per effettuare connessioni SSH da codice. A questo punto è stato implementato il metodo `executeScript` nella classe `Node` che

[1]Ganymed SSH:`http://www.ganymed.ethz.ch/ssh2/`

prende in input un oggetto di tipo `Script` e lo esegue sulla macchina scelta, con la possibilità di passare dei parametri

```
def executeScript(script: Script, params: String*)
```

Potendo quindi eseguire comandi sulle macchine, si è passati a scrivere gli script di configurazione veri e propri per:

- Installare i componenti necessari al funzionamento dei nodi Master ed Agent
- Configurare la risoluzione degli hostname all'interno delle macchine
- Avviare ed arrestare i nodi Master ed Agent
- Eseguire il framework Marathon sul cluster

I file relativi all'installazione dei componenti sono stati sviluppati utilizzano il classico formato bash [4.1]

```
#!/bin/bash
[...]
sudo 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"
| sudo tee /etc/apt/sources.list.d/mesosphere.list
sudo apt-get update
sudo apt-get install mesos -y
[...]
```

Listato 4.1: Frammento di script bash per l'installazione dei componenti

mentre gli altri sono stati scritti con una particolare sintassi che permette di utilizzate Scala come linguaggio di scripting [4.2]

```
#!/bin/sh
exec scala "$0" "$@"
!#
object Main extends App {
 <codice dello script>
}
Main.main(args)
```

Listato 4.2: Esempio di utilizzo di Scala come linguaggio di scripting

Avendo quindi a disposizione gli script, sono stati definiti i relativi oggetti all'interno del companion object della classe `MesosCluster` [4.3]

```
object MesosCluster {
 object InstallMaster extends Script("scripts/install_master.sh")
 object InstallAgent extends Script("scripts/install_agent.sh")
 object StartMaster extends Script("scripts/start_master.scala")
 object StartAgent extends Script("scripts/start_agent.scala")
 object StartMarathon extends Script("scripts/start_marathon.scala")
 object SetHosts extends Script("scripts/set_hosts.scala")
 object StopMaster extends Script("scripts/stop_master.scala")
 object StopAgent extends Script("scripts/stop_agent.scala")
}
```

Listato 4.3: Lista degli script disponibili definiti attraverso degli `object` scala

l'implementazione del metodo `createCuster` è risultata a questo punto immediata [4.4]

```
masters.foreach(_.executeScript(InstallMaster))
agents.foreach(_.executeScript(InstallAgent))
[...]
masters.foreach(_.executeScript(StartMaster))
agents.foreach(_.executeScript(StartAgent))
```

Listato 4.4: Frammento di implementazione del metodo `createCluster`

Gli stessi script sono stati utilizzati anche per sviluppare i metodi di arresto del cluster e aggiunta/rimozione di nodi.

4.2.2 Configurazione di un cluster da un file JSON

Per quanto riguarda la gestione dei file JSON è stata utilizzata la libreria lift-json[2] che mette a disposizione varie funzionalità per utilizzare questo tipo di file. In questo modo è stato implementato il metodo `loadFromJson` per la classe `MesosCluster`, inserito poi all'interno del companion object come factory method: `MesosCluster.loadFromJson("file.json")`

4.2.3 Gestione dei task sul cluster

Per poter eseguire, arrestare e gestire i task su Mesos è stata utilizzata l'API REST messa a disposizione dal framework Marathon (che viene avviato di default durante la creazione del cluster). Questa offre varie funzionalità che devono essere chiamate passando come argomento il task in formato JSON; per

[2]Lift Json: `https://github.com/lift/lift/tree/master/framework/lift-base/lift-json`

questo motivo è stata implementata la classe `TaskSerializer` che, combinata con le funzionalità di lift-json, permette di salvare i task su file in questo formato.

Le chiamate all'API di Marathon hanno una sintassi di questo tipo [4.5]

```
"curl -X POST http://<MARATHON-IP>:8080/v2/apps -d @task.json -H
   'Content-type: application/json'"
```

Listato 4.5: Esempio di una chiamata all'API REST di Marathon

Visto che la libreria sviluppata deve poter essere compatibile con qualsiasi sistema operativo, il comando curl da utilizzare cambia a seconda che si stia usando una macchina Windows based o Linux based [4.6].

```
curl = System.getProperty("os.name") match {
     case s if s.startsWith("Windows") => "curl.exe"
     case _ => "curl"
```

Listato 4.6: Utilizzo del comando `curl`

4.3 Task

Una volta conclusa la parte riguardante la configurazione del cluster, si è passati a sviluppare tutto ciò che riguarda i task. Dopo aver implementato il trait Task rifacendosi all'interfaccia definita in fase di progettazione [3.4], sono state sviluppate le due classi `Container` e `DockerContainer` che contengono i parametri da fornire a Marathon per l'esecuzione dei container [4.7]

```
object DockerContainer {
  case class DockerParameter(key: String, value: String)
}
case class DockerContainer(image: String, network: String, forcePullImage:
    Boolean, privileged: Boolean, ports: Array[Int], parameters:
    Array[DockerParameter])

object Container {
  implicit def dockerContainerToOption(docker: DockerContainer) =
      Some(docker)
  implicit def containerToOption(container: Container) = Some(container)
}
case class Container(`type`: String, docker: Option[DockerContainer],
    portMappings: Option[Map[Int, Int]])
```

Listato 4.7: Implementazione delle classi `Container` e `DockerContainer`

è stata in seguito implementata la classe `GenericTask`, che rappresenta un generico task Marathon da eseguire sul cluster [4.8]

```
class GenericTask(var id: String,
                  var cpus: Double,
                  var mem: Double,
                  var disk: Double,
                  var cmd: Option[String],
                  val container: Option[Container],
                  var env: Map[String, String],
                  var instances: Int = 1) extends Task
```

Listato 4.8: Implementazione della classe `GenericTask`

Utilizzando questa come base, sono state poi sviluppate le classi `CassandraTask` e `KafkaTask`.

4.3.1 CassandraTask

La classe `CassandraTask` è un `GenericTask` in cui viene eseguito di default il container docker relativo ad un nodo Cassandra [4.9].

```
class CassandraTask(id: String, cpus: Double, mem: Double, disk:
    Double, cmd: Option[String], instances: Int = 1)
  extends GenericTask(id, cpus, mem, disk, cmd,
      CassandraTask.CASSANDRA_CONTAINER, Map(), instances) {
  [...]
 }
```

Listato 4.9: Implementazione della classe `CassandraTask`

Il container eseguito è definito nell'oggetto `CASSANDRA_CONTAINER` del suo companion object [4.10]

```
val CASSANDRA_CONTAINER = Container("DOCKER",
    DockerContainer("cassandra:latest", [...]))
```

Listato 4.10: Definzione del container docker relativo ad un nodo Cassandra

"cassandra:latest" indica il nome dell'immagine docker da eseguire all'interno del container. In rete sono disponibili varie immagini già configurate per l'esecuzione di Cassandra su container; in questo caso si è scelto di utilizzare quella ufficiale[3].

[3] `https://hub.docker.com/_/cassandra/`

All'interno della classe CassandraTask sono state in seguito definite, tramite un `Enumeration`, alcune variabili d'ambiente da utilizzare per la configurazione del cluster Cassandra [4.11].

```
object CassandraVariable extends Enumeration {
  type CassandraVariable = Value
  val CASSANDRA_CLUSTER_NAME = Value("CASSANDRA_CLUSTER_NAME")
  val CASSANDRA_LISTEN_ADDRESS = Value("CASSANDRA_LISTEN_ADDRESS")
  val CASSANDRA_BROADCAST_ADDRESS = Value("CASSANDRA_BROADCAST_ADDRESS")
  val CASSANDRA_RPC_ADDRESS = Value("CASSANDRA_RPC_ADDRESS")
  val CASSANDRA_START_RPC = Value("CASSANDRA_START_RPC")
  val CASSANDRA_SEEDS = Value("CASSANDRA_SEEDS")
  val CASSANDRA_NUM_TOKENS = Value("CASSANDRA_NUM_TOKENS")
  val CASSANDRA_DC = Value("CASSANDRA_DC")
  val CASSANDRA_RACK = Value("CASSANDRA_RACK")
  val CASSANDRA_ENDPOINT_SNITCH = Value("CASSANDRA_ENDPOINT_SNITCH")
}
```

Listato 4.11: Enumeration per le variabili d'ambiente principali di un container Cassandra

4.3.2 KafkaTask

In maniera analoga è stata implementata la classe `KafkaTask`. In questo caso invece di utilizzare l'immagine docker di Cassandra è stata utilizzata quella di un broker Kafka[4] [4.12]

```
val KAFKA_CONTAINER = Container("DOCKER",
    DockerContainer("wurstmeister/kafka", [...]))
```

Listato 4.12: Definzione del container docker relativo ad un broker Kafka

anche qui sono state definite delle variabili d'ambiente predefinite per la configurazione di Kafka [4.13].

```
object KafkaVariable extends Enumeration {
  type KafkaVariable = Value
  val HOSTNAME_COMMAND = Value("HOSTNAME_COMMAND")
  val KAFKA_ADVERTISED_LISTENERS =
      Value("KAFKA_ADVERTISED_LISTENERS")
  val KAFKA_LISTENERS = Value("KAFKA_LISTENERS")
  val KAFKA_ZOOKEEPER_CONNECT = Value("KAFKA_ZOOKEEPER_CONNECT")
```

[4] https://hub.docker.com/r/wurstmeister/kafka/

```
  val KAFKA_LISTENER_SECURITY_PROTOCOL_MAP =
      Value("KAFKA_LISTENER_SECURITY_PROTOCOL_MAP")
  val KAFKA_INTER_BROKER_LISTENER_NAME =
      Value("KAFKA_INTER_BROKER_LISTENER_NAME")
}
```

Listato 4.13: Enumeration per le variabili d'ambiente principali di un container Kafka

4.4 Cluster virtuali su Mesos

Una volta definiti quindi i task per Cassandra e Kafka, si è passati all'implementazione dei cluster virtuali `CassandraCluster` e `KafkaCluster`. Inizialmente, rifacendosi a quanto detto durante la fase di progettazione, si era pensato di adottare una soluzione in cui il cluster virtuale viene creato eseguendo su Mesos tanti oggetti task connessi tra loro; si è poi deciso, invece, di sfruttare la possibilità che offre Marathon di definire il numero di istanze attive di un certo task: per creare un cluster Cassandra di 10 nodi per esempio, invece di istanziare 10 oggetti di tipo `CassandraTask`, se ne può eseguire uno solo impostando il numero di istanze a 10 [4.14].

```
val cassandraCluster = CassandraTask([...], instances=10)
```

Listato 4.14: Creazione di un cluster Cassandra a partire da un task. (Vengono omessi nel costruttore i parametri non rilevanti)

Con la stessa filosofia, aggiungere o rimuovere nodi dal cluster corrisponde a modificare attravero l'API di Marathon il numero di istanze attive del task.

Si può notare come con questa politica di gestione dei cluster su Marathon l'unica cosa che differenzia due cluster è il task che rappresenta il singolo nodo. Per questo motivo tutta la logica appena descritta è stata inserita all'interno del trait `MarathonCluster` (Scala permette di implementare i metodi direttamente all'interno dei trait) e le classi `CassandraCluster` e `KafkaCluster` che lo estendono hanno il solo compito di implementare il metodo `nodeTask` che definisce quale task eseguire (vedi [4.15] e [4.16]).

```
class CassandraCluster(override val mesos: MesosCluster,
     override val clusterName: String)
  extends MarathonCluster {
  [...]
  override def nodeTask(cpus: Double, memory: Double) = {
```

```
    val task = CassandraTask(this.clusterName, cpus, memory, 0, None,
        nodes)
    task.set(CassandraVariable.CASSANDRA_CLUSTER_NAME, this.clusterName)
    task.set(CassandraVariable.CASSANDRA_SEEDS,
        mesos.agents.map(_.getIp).mkString(","))
    task
  }
  [...]
}
```

Listato 4.15: Definizione del task relativo ad un nodo Cassandra

```
private class KafkaCluster(override val mesos: MesosCluster,
      override val clusterName: String)
extends MarathonCluster {
 [...]
 override def nodeTask(nodes: Int, cpus: Double, memory: Double) = {
    val kafka = KafkaTask(this.clusterName, cpus, memory, disk = 0, cmd =
        None, instances = nodes)
    kafka.set(KafkaVariable.HOSTNAME_COMMAND,"ip -4 route get 8.8.8.8 | awk
        {'print $7'} | tr -d '\\n'")
    kafka.set(KafkaVariable.KAFKA_ZOOKEEPER_CONNECT,
        s"${mesos.masters.map(_.getIp).mkString(",")}:2181/kafka")
    kafka.set(KafkaVariable.KAFKA_LISTENERS,
     "INSIDE://_{HOSTNAME_COMMAND}:9092,OUTSIDE://_{HOSTNAME_COMMAND}:9094")
    kafka.set(KafkaVariable.KAFKA_ADVERTISED_LISTENERS,
     "INSIDE://_{HOSTNAME_COMMAND}:9092,OUTSIDE://_{HOSTNAME_COMMAND}:9094")
    kafka.set(KafkaVariable.KAFKA_LISTENER_SECURITY_PROTOCOL_MAP,
        "INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT")
    kafka.set(KafkaVariable.KAFKA_INTER_BROKER_LISTENER_NAME, "INSIDE")
    kafka
 }
 [...]
}
```

Listato 4.16: Definizione del task relativo ad un broker Kafka. In questo caso è necessario impostare qualche configurazione aggiuntiva in modo da permettere una corretta comunicazione dei broker

In realtà le classi che estendono `MarathonCluster` devono implementare anche il metodo `getConnectionInfo` che restituisce i parametri di connessione al cluster. Per esempio per Cassandra viene restituita la lista di indirizzi dei nodi e la porta su cui sono in ascolto

```
case class CassandraConnectionInfo(nodes: List[String], port: Int)
```

per Kafka invece, visto che i broker potrebbero essere in ascolto su porte differenti, viene restituita una mappa che associa ogni indirizzo ip ad una porta.

```
case class KafkaConnectionInfo(brokers: Map[String, Int])
```

4.5 SmackEnvironment

A questo punto si è passati all'implementazione della classe `SmackEnvironment` che, utilizzando gli strumenti descritti in precedenza, raggruppa in un'unica interfaccia le funzionalità principali per la creazione un'infrastruttura SMACK. La classe utilizza al suo interno le classi `CassandraCluster` e `KafkaCluster` per la gestione dei cluster Cassandra e Kafka, e definisce un apposito task per eseguire lo spark framework sul cluster Mesos [4.17].

```
private val sparkDispathcerTask = new TaskBuilder()
  .setId("spark-dispatcher")
  .setCpus(0.5).setMemory(512).setDisk(0)
  .setCmd(s"/root/${SmackEnvironment.SPARK_VERSION}/bin/spark-class " +
    "org.apache.spark.deploy.mesos.MesosClusterDispatcher " +
    s"--master mesos://${mesos.zkConnectionString}/mesos")
  .build()
```

Listato 4.17: Task che esegue lo Spark framework

Capitolo 5

Esempi di utilizzo ed usabilità

In questa sezione verranno mostrati degli esempi di utilizzo della libreria per dare un idea del suo funzionamento e grado di usabilità. All'interno di alcuni programmi verranno utilizzate delle `Thread.sleep` per rappresentare sequenze di codice eseguite in momenti temporali diversi.

5.1 Creazione di un cluster Mesos

La creazione di un cluster Mesos può essere effettuata in due modalità equivalenti:

1. Istanziando un oggetto di tipo `MesosClusterBuilder` e configurando gli opportuni parametri:

`clusterName`	nome simbolico da assegnare al cluster
`masters`	lista di nodi che svolgeranno il ruoli di 'coordinatori' del cluster
`agents`	lista di nodi che svolgeranno il ruoli di 'esecutori' all'interno del cluster
`connection`	username e chiave ssh per connettersi ai nodi elencati sopra

```
import cluster.MesosClusterBuilder

object ClusterFromBuilder extends App {
  override def main(args: Array[String]): Unit = {
    val mesos = new MesosClusterBuilder()
      .setClusterName("Mesos-Cluster")
      .setMasters(List("159.65.127.232"))
```

```
      .setAgents(List("159.65.114.239",
         "159.65.113.201","159.65.117.154"))
      .setConnection("root", "private_key_openssh", "")
      .build()
    mesos.createCluster()
  }
}
```

Listato 5.1: Creazione del cluster Mesos tramite builder

2. Creando un file di configurazione in formato JSON (Figura 5.1) da passare come argomento al metodo `MesosCluster.fromJson`:

```
import cluster.{MesosCluster, Node}
import task.TaskBuilder

object CusterFromJson extends App {
  override def main(args: Array[String]) = {
    val cluster = MesosCluster.fromJson("clusterConfig.json")
    cluster.createCluster()
  }
}
```

Listato 5.2: Creazione del cluster Mesos tramite JSON

```
{
  "clusterName": "Mesos-Cluster",
  "user": "root",
  "sshKeyPath": "private_key_openssh",
  "sshKeyPassword": "",
  "masters": [
    "142.93.107.161"
  ],
  "agents": [
    "142.93.107.164",
    "142.93.107.179"
  ]
}
```

Figura 5.1: Esempio di file di configurazione

In entrambi i casi è poi sufficiente richiamare il metodo `cluster.createCluster()` per rendere operativo il cluster. Tramite questo comando, vengono installati automaticamente sui nodi tutti i componenti necessari e vengono avviati i servizi minimi per la gestione del cluster e l'esecuzione di task.

E' possibile eventualmente 'spegnere' un cluster chiamando il metodo `cluster.shutdownCluster()` che ferma, in tutti i nodi, i servizi relativi al funzionamento di Mesos.

5.2 Esecuzione di task all'interno del cluster

Una volta avviato un cluster, è possibile eseguire dei task al suo interno. Gli oggetti Task possono essere creati attraverso il normale costruttore oppure un TaskBuilder, ed eseguiti con il comando `cluster.run(task)`

```
import cluster.MesosCluster
import task.TaskBuilder

object RunTask extends App {
  override def main(args: Array[String]) = {
    val cluster = MesosCluster.fromJson("clusterConfig.json")
    cluster.createCluster()
    Thread.sleep(30000)
    val task = new TaskBuilder()
      .setId("task")
      .setMemory(1024)
      .setCpus(2)
      .setDisk(0)
      .setCmd("echo 'i'm a task' ; sleep 10")
      .build()
    cluster.run(task)
    Thread.sleep(10000)
    cluster.stop(task)
  }
}
```

Listato 5.3: Esecuzione di Task sul cluster

I parametri configurabili di un task sono:

`id` id o nome del task

`cpus` numero di cpus dedicate(default = 0.5)

`memory` memoria RAM dedicata (default = 512)

`disk` spazio su disco riservato (default = 0)

`cmd` comando bash da eseguire (default = None)

`container` container docker da eseguire (default = None)

`env` mappa chiave valore di variabili d'ambiente (default = Map())

`instances` numero di istanze del task (default = 1)

Una volta avviato il cluster Mesos è possibile utilizzare la classe `SmackEnvironment` per configurare i cluster Spark, Cassandra e Kafka.

5.3 Setup e configurazione di un cluster Cassandra

Per creare un cluster Cassandra si può utilizzare il metodo `startCassandraCluster` passando come argomenti il numero di nodi che faranno parte del cluster e quante CPU e memoria assegnare a ciascun nodo [5.4] Una volta avviato il cluster è possibile conoscere le informazioni di connessione tramite il metodo `getCassandraConnectionInfo`

```
import cluster.MesosClusterBuilder
import smack.SmackEnvironment

object StartCassandra extends App {
  override def main(args: Array[String]) = {
    val mesos = new MesosClusterBuilder()
      .setClusterName("Mesos-Cluster")
      .setMasters(List("159.65.127.232"))
      .setAgents(List("159.65.114.239", "159.65.113.201", "159.65.117.154"))
      .setConnection("root", "private_key_openssh", "")
      .build()
    val smack = new SmackEnvironment(mesos, "cassandra-cluster",
        "kafka-cluster")
    smack.startCassandraCluster(serversCount = 2, cpus = 2, memory = 2048)
    val cassandraNodes = smack.getCassandraConnectionInfo()
  }
}
```

Listato 5.4: Creazione di un cluster Cassandra

5.4 Setup e configurazione di un cluster Kafka

In maniera analoga si può creare un cluster Kafka utilizzando il metodo `startKafkaCluster` ed avere i parametri d connessione con `getKafkaConnectionInfo` [5.5].

```
import cluster.MesosClusterBuilder
import smack.SmackEnvironment

object StartKafka extends App {
  override def main(args: Array[String]) = {
    val mesos = new MesosClusterBuilder()
      .setClusterName("Mesos-Cluster")
      .setMasters(List("159.65.127.232"))
      .setAgents(List("159.65.114.239", "159.65.113.201", "159.65.117.154"))
      .setConnection("root", "private_key_openssh", "")
      .build()
    val smack = new SmackEnvironment(mesos, "cassandra-cluster",
        "kafka-cluster")
    smack.startKafkaCluster(brokers = 4, cpus = 1, memory = 1024)
    val brokers = smack.getKafkaConnectionInfo()
  }
}
```

Listato 5.5: Creazione di un cluster Cassandra

5.5 Setup e configurazione di Spark

E' possibile eseguire il framework che permette di utilizzare Mesos come cluster manager di Spark chiamando il metodo `startSparkFramework` [5.6]

```
import cluster.MesosClusterBuilder
import smack.SmackEnvironment

object StartKafka extends App {
  override def main(args: Array[String]) = {
    val mesos = new MesosClusterBuilder()
      .setClusterName("Mesos-Cluster")
      .setMasters(List("159.65.127.232"))
      .setAgents(List("159.65.114.239", "159.65.113.201", "159.65.117.154"))
      .setConnection("root", "private_key_openssh", "")
      .build()
```

```
    val smack = new SmackEnvironment(mesos, "cassandra-cluster",
        "kafka-cluster")
    smack.startSparkFramework()

  }
}
```

Listato 5.6: Creazione di un cluster Cassandra

5.6 Scalabilità dell'infrastruttura

Una volta avviato un cluster Mesos è possibile aggiungere e rimuovere degli agenti a seconda delle esigenze. Per farlo basta utilizzare i metodi `addAgent` e `removeAgent` di un oggetto di tipo MesosCluster

```
import cluster.{MesosCluster, Node}

object ScaleCluster extends App {
  override def main(args: Array[String]) = {
    val cluster = MesosCluster.fromJson("clusterConfig.json")
    cluster.createCluster()
    Thread.sleep(1000)
    val node1: Node = Node("178.43.54.212","root","sshKeyPath","")
    val node2: Node = Node("178.43.54.213","root","sshKeyPath","")
    cluster.addAgent(node1)
    cluster.addAgent(node2)
    Thread.sleep(1000)
    cluster.removeAgent(node2)
  }
}
```

Listato 5.7: Scalabilità del cluster Mesos

Anche per quanto riguarda Cassandra e Kafka è possibile aggiungere nodi e broker attraverso la classe `SmackEnvironment` richiamando i metodi `smack.addCassandraNode()` e `smack.addKafkaBroker()`

```
import cluster.MesosCluster
import smack.SmackEnvironment

object ScaleSmack extends App {
  override def main(args: Array[String]) = {
    val mesos = MesosCluster.fromJson("clusterConfig.json")
```

```
    mesos.createCluster()
    val smack = new SmackEnvironment(mesos, "cassandra-cluster",
        "kafka-cluster")
    smack.startCassandraCluster(serversCount = 2, cpus = 2, memory =
        2048)
    smack.startKafkaCluster(brokersCount = 1, cpus = 2, memory =
        2048)
    smack.startSparkFramework()
    Thread.sleep(1000)
    smack.addCassandraNodes(1)
    smack.addKafkaBrokers(2)
  }
}
```

Listato 5.8: Scalare un'infrastruttura SMACK

Conclusioni

La libreria sviluppata fornisce uno strumento semplice e rapido per la creazione e configurazione di uno stack SMACK. Rispetto alle tecnologie già esistenti, tra le quali troviamo per esempio Mesosphere DC/OS ed Amazon ECS, fornisce funzionalità più specifiche in merito alla configurazione di Mesos, Spark, Cassandra e Kafka e permette di avere con questi un controllo diretto via software. L'installazione e configurazione automatica dei componenti di basso livello riduce di molto i tempi che sarebbero normalmente necessari alla creazione dello stack e mette anche un utilizzatore "meno esperto" nella condizione di poter utilizzare la libreria. La struttura delle API permette di avere un approccio di tipo dichiarativo sulla definizione dell'infrastruttura, rendendo la creazione dello stack un'operazione semplice e realizzabile con poche righe di codice.

Utilizzando gli script bash e Scala si è riusciti ad automatizzare il processo di installazione e configurazione di Mesos: sia per quanto riguarda l'installazione a partire da zero, che per la gestione di un cluster esistente. Grazie all'API REST del framework Marathon è stato in seguito possibile implementare le funzionalità di esecuzione e monitoraggio di task e container; fondamentali per lo sviluppo successivo della libreria.

Il supporto alla conteinerizzazione fornito da Mesos, combinato con la tecnologia di Docker, permette di gestire in maniera flessibile e scalabile i cluster Kafka e Cassandra, utilizzando i container come "astrazione" per i nodi. Tramite lo Spark framework è inoltre possibile usare Mesos come cluster manager di Spark. In questo modo tutti gli agenti Mesos, se necessario, vengono utilizzati come esecutori Spark per il calcolo distribuito.

Grazie alle funzionalità di recovery di Mesos e Marathon l'infrastruttura creata con la libreria risulta essere scalabile e tollerante ai guasti permettendo l'aggiunta e rimozione di nodi in tempo reale.

L'utilizzo di Scala come linguaggio di programmazione rappresenta poi un valore aggiunto per il progetto in quanto è un linguaggio molto usato per lo sviluppo di applicazioni in ambito IoT e big data dove SMACK trova ampio utilizzo. Uno dei possibili sviluppi futuri potrebbe essere proprio quello di estendere

il codice attuale con funzionalità relative alla creazione di applicazioni su stack SMACK.

Bibliografia

[1] Karau H., Konwinski A., Wendell P., Zaharia M. *Learning Spark: Lightning-Fast Big Data Analysis*, o'reilly, 2015.

[2] Greenberg D. *Building Applications on Mesos - Leveraging resilient, scalable and distributed systems*, o'reilly, 2015.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., 1995.

Sitografia

[1] *https://mesosphere.com/blog/smack-stack-new-lamp-stack/*

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

[3] *https://kafka.apache.org/intro*

[4] *https://blog.thecodeteam.com/2017/07/17/data-analytics-using-container-persistence-smack/*