UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Informatica

Sistemi Distribuiti

Cluster Resource Management

Prof. Pizzutilo Sebastiano

Christopher Piemonte

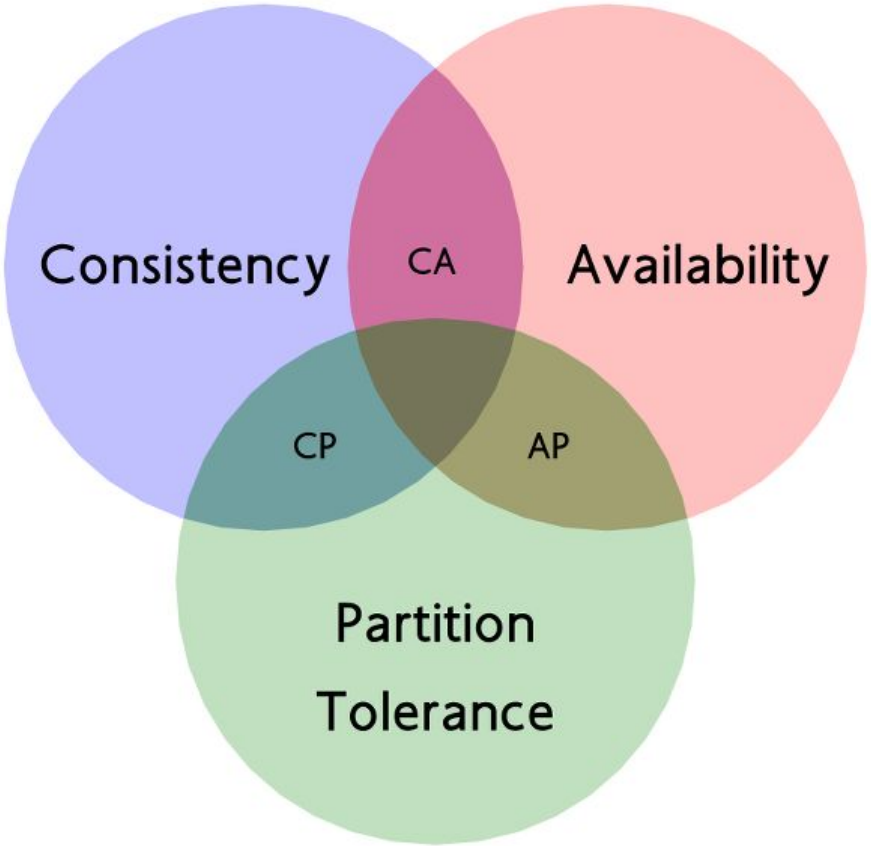
659723

---

a.a. 2016/2017

1 - Introduzione

Un Sistema Distribuito è un insieme di processori indipendenti, ciascuno con proprie risorse, che comunicano attraverso una rete di comunicazione e interagiscono tra loro per condividere risorse ovunque distribuite, apparendo ai suoi utenti come un singolo sistema coerente. L'obiettivo principale di un sistema distribuito è rendere l'accesso alle risorse remote facile per gli utenti e condividerle in maniera efficiente e controllata. Questo è spinto da motivazioni economiche e tecnologiche ed equivale a garantire proprietà quali la *trasparenza*, l'*apertura*, la *scalabilità*, introducendo una serie di problematiche poco diffuse nei sistemi stand-alone. Alla base della implementazione di un Sistema Distribuito vi è infatti la scelta, dipendentemente dai requisiti di business, tra coerenza (**Consistency**), disponibilità (**Availability**) e tolleranza di partizione (**Partition tolerance**) espressa dal CAP Theorem.



I vantaggi derivanti dall'utilizzo di un Sistema Distribuito spesso portano inevitabilmente ad una maggiore complessità del software ed una maggiore attenzione alla sicurezza. Nonostante questo esiste un grande interesse mondiale nella costruzione e installazione di sistemi distribuiti. La necessità di trovare un compromesso è intrinseca nella progettazione e nella architettura del sistema e rende le soluzioni adatte a domini specifici.

In seguito verrà approfondito Apache Hadoop, nato come sistema per il calcolo distribuito secondo il paradigma MapReduce ma in seguito astratto e generalizzato per fornire un framework comune sul quale sono stati sviluppati numerosi add-on e plugin dando vita a un ecosistema di applicazioni distribuite con una base comune. In particolare si parlerà dei sistemi indispensabili che forniscono le fondamenta di tutto l'ecosistema Hadoop: il File System Distribuito *HDFS* e il Cluster Resource Management *YARN*.

2 - Cluster Computer

Un computer **Cluster** è un insieme di computer omogenei connessi tramite una rete con lo scopo di distribuire una elaborazione (parallelizzabile) tra computer componenti il cluster, al costo di una maggiore complessità dell'infrastruttura.

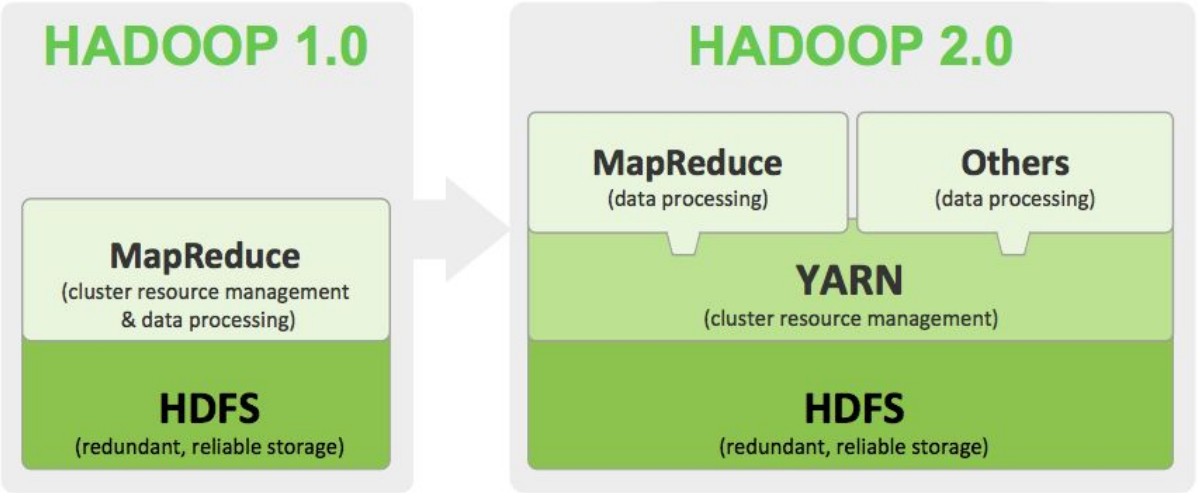
Per l'utente o i client, il cluster è assolutamente trasparente: tutta la notevole complessità hardware e software è mascherata; i servizi vengono erogati, i dati sono resi accessibili e le applicazioni elaborate come se fossero tutte provenienti da un solo computer centrale. Questo viene supportato dalle seguenti caratteristiche:

- **Fail-over Service**: il funzionamento delle macchine è continuamente monitorato e quando un host smette di funzionare un'altra macchina subentra in attività. Lo scopo è garantire dunque un servizio continuativo garantendo cioè alta disponibilità di servizio grazie all'alta affidabilità dovuta alla tolleranza ai guasti del sistema cluster per effetto della ridondanza di apparati;
- ***Load balancing***: è un sistema nel quale le richieste di lavoro sono inviate alla macchina con meno carico di elaborazione distribuendo/bilanciando così il carico di lavoro sulle singole macchine. Questo garantisce tempi minori di processamento di un servizio e minore affaticamento di una macchina;
- **High Performance Computing**: i computer sono configurati per fornire prestazioni estremamente alte. Le macchine suddividono i processi di un job su più macchine, al fine di guadagnare in prestazioni. La peculiarità saliente è che i processi sono parallelizzati e che le routine che possono girare separatamente saranno distribuite su macchine differenti invece di aspettare di essere eseguite sequenzialmente una dopo l'altra.

3 - Apache Hadoop

Apache Hadoop è uno dei più popolari framework per il calcolo distribuito, nato come versione Open Source del *Google File System* (GFS) e di *Google MapReduce* sviluppato inizialmente da Doug Cutting ha dato inizio ad un ecosistema di applicazioni distribuite. Queste se in principio si poggiavano sul solo file system distribuito, permettendo servizi di accesso ai dati (Hive, Pig, HBase), con l'introduzione di **YARN** (Yet Another Resource Navigator) sono state introdotte nuove applicazioni per il data processing.

Hadoop 2.0 ha quindi cercato di distanziarsi da un singolo modello computazionale ed offrire un livello di astrazione maggiore, cercando di diventare il Kernel del cluster distribuito e di farsi carico dei dettagli implementativi di basso livello. In seguito verranno descritte le componenti principali dell'architettura Hadoop.



3.1 - HDFS

HDFS è un file system distribuito ideato per soddisfare requisiti quali affidabilità e scalabilità, è in grado di gestire un numero elevatissimo di file, anche di

dimensioni ragguardevoli (dell'ordine dei Gigabyte o Terabyte), attraverso la distribuzione dei dati su cluster contenenti migliaia di nodi.

HDFS organizza i file in una struttura gerarchica, delegando i compiti di gestione ed immagazzinamento a diversi processi secondo una architettura master/slave. Un cluster HDFS è costituito dai seguenti tipi di nodi:

- **NameNode**: è l'applicazione che gira sul server principale. Gestisce il file system ed in particolare il namespace, ovvero l'elenco dei nomi dei file e dei blocchi e ne controlla l'accesso. Il NameNode distribuisce le informazioni contenute nel namespace su due file: il primo è *fsimage*, che costituisce l'ultima immagine del namespace; il secondo è un log dei cambiamenti avvenuti al namespace a partire dall'ultima volta in cui il file fsimage è stato aggiornato. Quando il NameNode parte effettua un merge di fsimage con il log dei cambiamenti così da produrre uno snapshot dell'ultima situazione.
- **DataNode**: applicazioni che girano sui nodi del cluster, generalmente una per nodo, e gestiscono fisicamente lo storage di ciascun nodo. Queste applicazioni eseguono, logicamente, le operazioni di lettura e scrittura richieste dai client e gestiscono fisicamente la creazione, la cancellazione o la replica dei blocchi dati.
- **SecondaryNameNode**: noto anche come **CheckPointNode**, si tratta di un servizio che aiuta il NameNode ad essere più efficiente. Infatti si occupa di scaricare periodicamente il file fsimage e i log dei cambiamenti dal NameNode, di unirli in un unico snapshot che è poi restituito al NameNode.
- **BackupNode**: è il nodo di failover e consente di avere un nodo simile al SecondaryNameNode sempre sincronizzato con il NameNode.

3.1.1 - Architettura

HDFS è organizzato secondo una architettura master/slave. Un cluster HDFS consiste in un singolo NameNode, un master server che gestisce il namespace del

FIle System e l'accesso ai file da parte dei client. Inoltre, i DataNode presenti su ogni nodo del cluster, gestiscono lo storage del nodo sul quale girano. Internamente un file è suddiviso in uno o più blocchi che saranno immagazzinati in vari DataNode. Il NameNode esegue le operazioni di namespace di apertura, chiusura, rinomina dei file e delle directory presenti nel File System. Sono i DataNode a gestire le richieste di lettura e scrittura dei client ed a eseguire, sotto istruzioni del NameNode, le operazioni di creazione, rimozione e replicazione dei blocchi.

Il NameNode ed i DataNode sono software progettati per girare su macchine generiche. Queste tipicamente sono fornite di un sistema operativo GNU/Linux e devono poter eseguire applicazioni Java.

Il NameNode possiede il repository contenente tutti i metadati del cluster HDFS. Qui è mantenuto il namespace del File System, qualunque cambiamento ad esso o alle sue proprietà è effettuato dal NameNode. Una applicazione può specificare il numero di repliche di un file che il NameNode deve mantenere, questo fattore è chiamato replication factor del file ed è una delle informazioni mantenute dal NameNode.

Replication

HDFS è progettato per immagazzinare in modo affidabile file molto grandi distribuiti su diverse macchine all'interno del cluster. Mantiene ogni file come una sequenza di blocchi; tutti i blocchi di un file, tranne l'ultimo, hanno la stessa grandezza e sono replicati per garantire la tolleranza ai guasti. Il NameNode prende tutte le decisioni riguardo la replicazione dei blocchi mentre la grandezza dei blocchi e il replication factor sono specifici per ogni file. I file HDFS hanno unicamente uno scrittore alla volta. Per garantire il corretto funzionamento del cluster, il NameNode riceve segnali, detti HeartBeat e BlockReport, da ogni DataNode nel cluster. La ricezione di un HeartBeat da parte di un DataNode implica il corretto funzionamento del DataNode, mentre il BlockReport contiene una lista di tutti i blocchi presenti su quel DataNode.

Robustness

- **Data Disk Failure, Heartbeats e Re-Replication**: Ogni DataNode manda un segnale di HeartBeat al NameNode periodicamente. Può capitare che una partizione di rete può causare la perdita di connessione fra un sottoinsieme di DataNode ed il NameNode. Questa condizione può essere individuata dall'assenza di HeartBeat. Il NameNode contrassegna i DataNode come Deade e non inoltra nuove richieste IO. Ogni blocco immagazzinato sui Dead DataNode non è più disponibile e può causare il numero di repliche di qualche blocco a scendere sotto il replication factor. Il NameNode traccia costantemente i blocchi e crea nuove replica se necessario. Questa può essere causata da diversi fattori come il fallimento di un nodo, dello storage di un nodo, dalla corruzione di un blocco o dall'aumento del replication factor per quel file. Il time-out per marcare un nodo come Dead è di solito abbastanza lungo (più di 10 minuti di default) per evitare il la creazione di repliche indesiderate.

- **Cluster Rebalancing**: HDFS è fornito di schemi di rebalancing. Uno schema può spostare automaticamente blocchi da un DataNode ad un altro se lo spazio libero di un DataNode scende al di sotto di una soglia predefinita. In un periodo di richiesta maggiore di un particolare file possono essere create e riallocate dinamicamente nuove repliche dei blocchi.

- **Data Integrity**: Può accadere che un blocco recuperato da un DataNode arrivi corrotto. Questo può essere dovuto al fallimento dello storage del nodo, a problemi di rete o a problemi software. Il client HDFS verifica il checksum del contenuto del blocco HDFS. Quando un file viene creato infatti, viene calcolato il checksum per ogni blocco del file e questi vengono memorizzati in un file nascosto separato nello stesso namespace. Quando un client richiede un file, vengono verificati i dati ricevuti da ogni DataNode e comparati con i

relativi checksum. In caso di errori, si cerca di ottenere una replica dello stesso blocco da un altro DataNode.

Data Organization

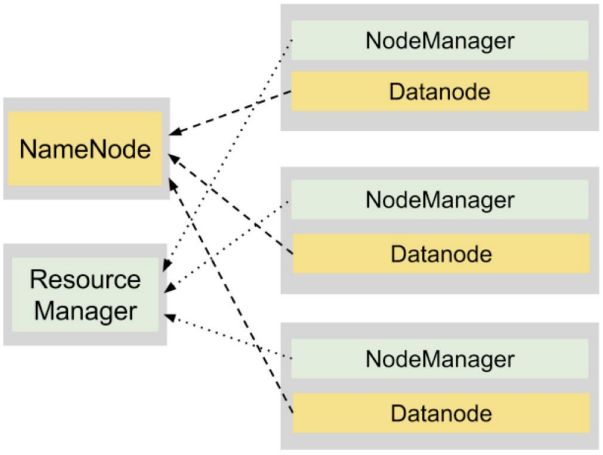
- **Data Blocks**: Il FIle System è progettato per la memorizzazione di file molto grandi. Le applicazioni compatibili con HDFS hanno bisogno di manipolare grandi data set. Di solito le operazioni di scrittura avvengono solo una volta mentre le operazioni di lettura sono più frequenti. La grandezza tipica di un blocco è di 128 MB. Quindi un file HDFS è diviso in blocchi di 128 MB, memorizzati possibilmente su DataNode differenti.
- **Replication Pipelining**: Quando un client sta scrivendo su di un file HDFS con un replication factor di tre, il NameNode recupera una lista di DataNode usando un algoritmo di scelta dei target delle repliche. La lista contiene i DataNode che ospiteranno le repliche di quel blocco. Il client quindi scrive sul primo DataNode della lista. Questo a sua volta riceve i dati in porzioni, le scrive nel suo repository locale e le manda al DataNode successivo. Il secondo DataNode, ricevute le porzioni del blocco le scrive sul repository locale e le manda al terzo DataNode. Il terzo DataNode le scrive soltanto nel repository locale in quanto è stato raggiunto il numero di repliche specificato nel replication factor. Si forma quindi una pipeline fra i DataNode nella lista i quali ricevono, scrivono e trasferiscono i dati.

3.2 - YARN

Inizialmente vi erano solo due layer a comporre l'architettura di Hadoop: lo storage layer HDFS ed il data-processing layer MapReduce. Quest'ultimo era l'unico modello computazionale e con il tempo ha rivelato i suoi difetti su determinati casi d'uso. Con l'introduzione di YARN, il cluster Hadoop dispone di un generico framework per il resource management per applicazioni distribuite, dove è possibile utilizzare diverse framework per il calcolo distribuito, implementati su misura per i diversi task.

MapReduce adesso è solo una delle possibili opzioni affiancato da altre soluzioni indirizzate al graph-processing, stream-processing, MPI, ecc.

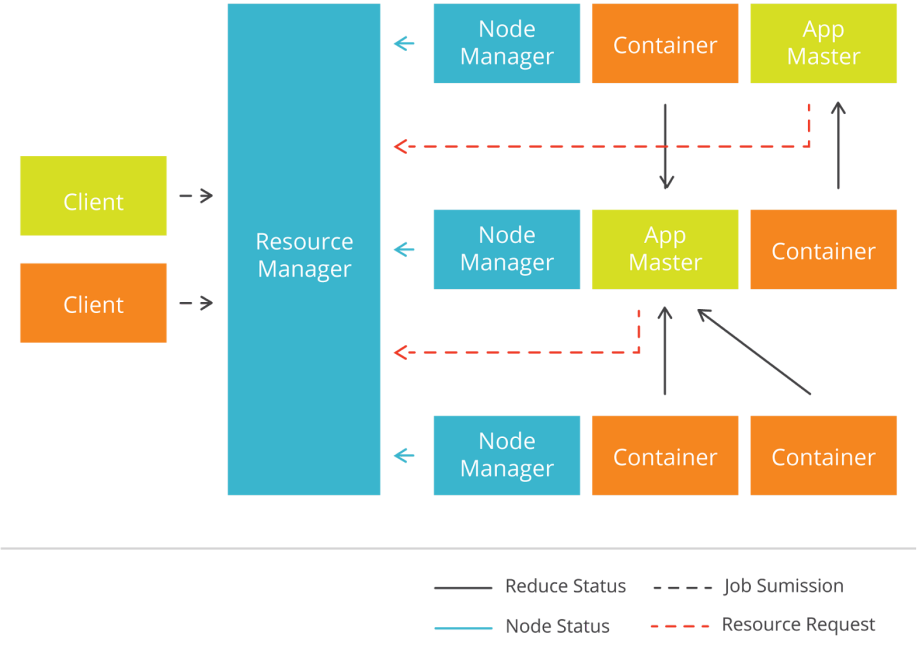
L'idea fondamentale di YARN consiste nel dividere le funzionalità di Resource Management e di Job Scheduling/Monitoring in processi differenti. Si basa sulla idea di avere di avere un ResourceManager globale, un NodeManager per ogni nodo ed un ApplicationMaster per ogni applicazione. Un'applicazione può essere un singolo job o un insieme di job (di solito rappresentati come grafi aciclici orientati o DAG). Il ResourceManager insieme con il NodeManager costituiscono l'infrastruttura distribuita per la gestione delle applicazioni.



- Il **ResourceManager** è l'autorità che gestisce le risorse per tutte le applicazioni del sistema. Ha due componenti principali: lo *Scheduler* e l'*ApplicationManager*. Lo Scheduler è responsabile per l'allocazione delle risorse alle varie applicazione in esecuzione, rispettando i limiti specificati. Non effettua nessun compito di monitoraggio o tracciamento dello stato dell'applicazione e non offre nessuna garanzia sulla riesecuzione di task falliti per application o hardware failure, delegando il lavoro ai Singoli ApplicationMaster distribuendo così il carico sui nodi. Lo scheduler assegna le risorse in base alle esigenze delle applicazioni, rilasciando Container. L'ApplicationManager è responsabile per l'accettazione delle job-submissions,

inizializzando il primo container contenente l'ApplicationMaster, al restart di questo in caso di failure e al mantenimento della lista delle applicazioni in esecuzione. Dopo la richiesta di esecuzione di un'applicazione, verifica se questa può essere soddisfatta (in base alle risorse richieste e quelle disponibili), in caso positivo, gli assegna un ID univoco e la inoltra allo Scheduler.

- Il **NodeManager**, uno per ogni nodo, è lo slave responsabile dei singoli nodi, del monitoraggio delle proprie risorse (CPU, memoria, disco, rete), dei Container e del riferimento di informazioni al ResourceManager.
- L'**ApplicationMaster**, uno per applicazione, è incaricato di negoziare le risorse con il ResourceManager (con lo Scheduler) e di cooperare con il NodeManager monitorando lo stato ed i progressi del container. È utile ricordare che è inoltre possibile configurare gli ApplicationMaster per gestire un insieme di applicazioni per volta (ad esempio un ApplicationMaster per Pig o Hive che gestisce un insieme di job MapReduce).



L'introduzione di YARN permette di esibire le seguenti caratteristiche:

- **Scalabilità**: demandare le responsabilità dei resource manager tradizionali all'ApplicationMaster, distribuendo il carico di gestione sui nodi.
- **Apertura**: spostando tutto il codice specifico all'applicazione all'interno dell'ApplicationMaster generalizza il sistema in modo da poter eseguire diversi tipi di processi.
- **Multi-tenancy**: permette l'utilizzo di diversi engine, i quali possono accedere simultaneamente allo stesso dataset utilizzando Hadoop come standard per batch, interactive e real-time processing.
- **Cluster Utilization**: l'allocazione dinamica migliora l'utilizzo delle risorse.
- **Compatibility**: applicazioni sviluppate per MapReduce su Hadoop 1.0 possono essere eseguite senza modifiche.

3.2.1 - Negoziazione delle risorse

Lo Scheduler del ResourceManager deve possedere tutte le informazioni necessarie per prendere decisioni efficienti ed ottimizzare l'accesso al File System, ad esempio cercando di ridurre il traffico dati. Questo viene effettuato attraverso le **ResourceRequest** ed il rilascio di **Container**. Essenzialmente un'applicazione richiede specifiche risorse attraverso l'ApplicationMaster allo Scheduler. Questo risponde concedendo un Container che soddisfa le richieste dell'applicazione esplicitate nella ResourceRequest.

Una ResourceRequest ha una forma del genere:

< *resource-name*, *priority*, *resource-requirement*, *number-of-containers* >

Dove:

- **resource-name**: può essere un hostname, un rackname o * (nessuna preferenza).

- **priority**: priorità relativa alle altre richieste della stessa applicazione.
- **resource-requirement**: memoria, cpu ecc.
- **number-of-container**: numero di container.

I Container rappresentano essenzialmente l'allocazione delle risorse, ovvero la riuscita di una ResourceRequest. Il Container permette l'utilizzo delle risorse in esso contenute all'applicazione su di uno specifico nodo. L'ApplicationMaster deve comunicare l'avvenuta allocazione del Container al NodeManager.

YARN permette il lancio di diverse processi, diversamente dal solo MapReduce di Hadoop 1.0, le API per l'esecuzione all'interno dei Container sono quindi platform-agnostic e possono contenere:

- Comandi per l'esecuzione di script.
- Variabili d'ambiente.
- Risorse locali necessarie al lancio dell'applicazione come jar, file, ecc.
- Token di sicurezza.

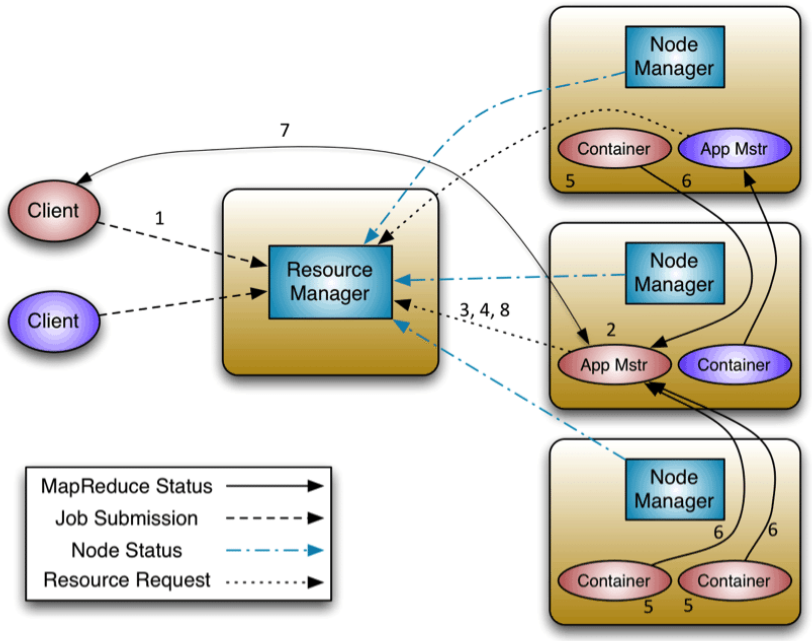
Questo permette all'ApplicationMaster di lavorare unitamente al NodeManager per l'esecuzione di diversi processi, che possono variare da semplici shell script a processi C/Java/Python o ancora a macchine virtuali a tutti gli effetti.

3.2.2 - Esecuzione di Applicazioni

L'esecuzione di applicazioni consiste nei seguenti passi:

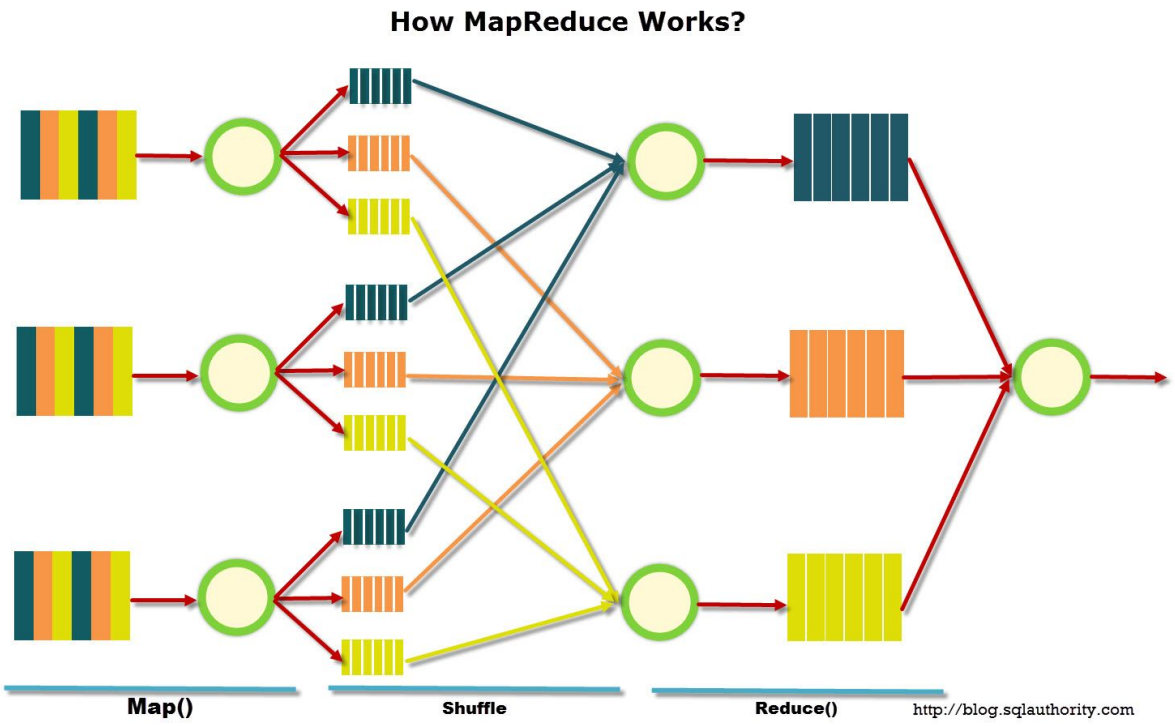
1. Il client presenta la richiesta di esecuzione di un'applicazione con tutte le specifiche necessarie alla creazione di un ApplicationMaster appropriato.

2. Il ResourceManager assume la responsabilità di negoziare uno specifico Container nel quale far partire l'ApplicationMaster. Ottenuto il Container lancia l'ApplicationMaster.
3. All'avvio l'ApplicationMaster si registra al ResourceManager permettendo al client di comunicare direttamente con lui.
4. Durante le normali operazioni l'ApplicationMaster negozia le risorse ed i Container attraverso le ResourceRequest.
5. All'avvenuta allocazione dei Container, l'ApplicationMaster lancia il Container comunicando le specifiche al NodeManager. Queste permettono al Container di comunicare con l'ApplicationMaster.
6. Il codice eseguito all'interno del Container comunica informazioni come il progresso e lo stato dell'applicazione all'ApplicationMaster.
7. Durante l'esecuzione, il client comunica direttamente con l'ApplicationMaster per conoscere lo stato o i progressi dell'applicazione.
8. Quando l'esecuzione è terminata l'ApplicationMaster annulla la sua registrazione dal ResourceManager e rilascia il Container.



3.3 - MapReduce

A grandi linee il modello MapReduce consiste in una prima fase di Map, dove l'input viene diviso in chunk per essere processato, ed in una fase di Reduce dove l'output della fase precedente viene aggregato in modo da produrre il risultato desiderato. La semplice, ed abbastanza ristretta, natura del modello di calcolo porta ad una efficiente implementazione su larga scala di applicazioni distribuite su nodi ordinari.



L'implementazione più popolare di questo paradigma è proprio quella di Apache Hadoop che, insieme al file system distribuito HDFS, ha garantito l'esplosione di popolarità di Hadoop. In particolare, MapReduce, vanta la proprietà di ridurre lo spostamento dei dati da processare attraverso la rete. Infatti i task MapReduce vengono assegnati sullo stesso nodo dove sono presenti i blocchi HDFS interessati, portando la computazione dai dati e non viceversa.

3.3.1 - Evoluzione

Inizialmente poteva essere scomposto nelle seguenti componenti:

- Le ***MapReduce API*** per sviluppare applicazioni MapReduce.
- Il ***MapReduce framework***, implementazione del runtime delle varie fasi di map e reduce (e le varie fasi di aggregazione sort/shuffle/merge).
- Il ***MapReduce system***, infrastruttura backend per la gestione delle risorse del cluster e lo scheduling dei job.

L'implementazione era composta dallo JobTracker(master) ed i TaskTracker(slave) per ogni nodo.

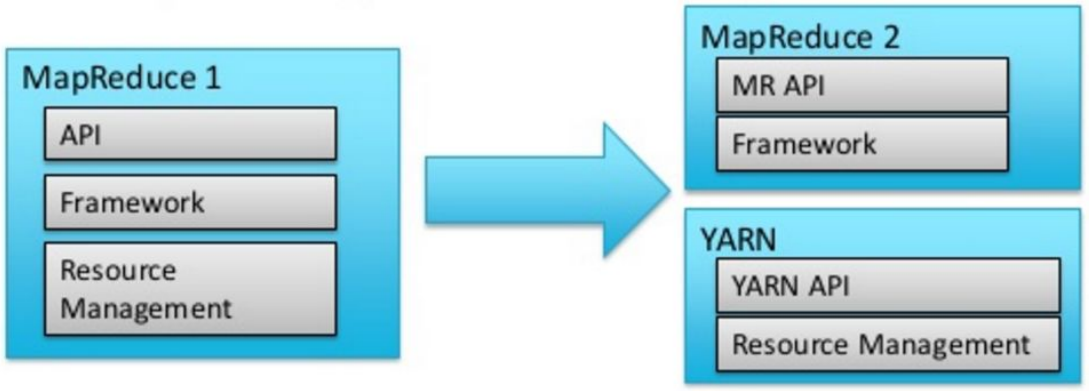
- Lo **JobTracker** era responsabile per la gestione degli TaskTracker e delle risorse concesse. In particolare assegnando task individuali ai TaskTracker e monitorando le risorse ed il ciclo di vita dei task, garantendo la fault tolerance.
- Il **TaskTracker** era responsabile per l'esecuzione e la terminazione dei task assegnati dal JobTracker, ed al riferimento periodico dello stato del task.

Questo portava a problemi di resource management quali:

- Sotto utilizzo del cluster: lo scheduling dei job organizzava ogni nodo con solo slot map o solo slot reduce.
- Impossibilità di condividere risorse con applicazioni non MapReduce.
- Limite di 4000 nodi nel cluster.

Soluzioni introdotte con YARN:

- I nodi non hanno slot ma risorse.
- Supporto ad applicazioni MapReduce e non.
- Delega di alcune funzioni dello Jobtracker (master) agli ApplicationMaster (slave).



MapReduce diventa un'applicazione YARN, il quale non ha interesse a conoscere il tipo di applicazione. Viene istanziato un ApplicationMaster per MapReduce, il quale dovrà richiedere risorse, come qualsiasi altra applicazione, attraverso ResourceRequest e l'allocazione di Container da parte del ResourceManager. Effettuato il suo compito, rimuove la registrazione del suo ApplicationMaster dal ResourceManager.

In conclusione, il lavoro dello Jobtracker viene diviso ed assegnato in parte al ResourceManager di YARN, per quanto riguarda l'assegnazione delle risorse, ed in parte ed i singoli ApplicationMaster, per la gestione della computazione. Tutto questo avviene in maniera trasparente, senza la necessità di ricompilare le applicazioni già esistenti.

4 - Cluster setup con Docker

4.1 - Tools

4.1.1 - Git

Un sistema di controllo di versione (VCS) consente di gestire le modifiche apportate ai file di un progetto su cui tipicamente lavorano più persone. Scopo di un VCS è quello di realizzare una corretta gestione delle modifiche garantendo caratteristiche di reversibilità, concorrenza e annotazione.
La reversibilità è la capacità di un VCS, di poter sempre tornare indietro in un qualsiasi punto della storia del codice sorgente, ad esempio nel caso in cui ci si è accorti di aver introdotto un errore ed è necessario ripristinare l'ultima versione stabile del software.
La concorrenza permette a più persone di apportare modifiche allo stesso progetto, facilitando il processo di integrazione di pezzi di codice sviluppati da due o più sviluppatori.
L'annotazione è la funzione che consente di aggiungere spiegazioni e riflessioni ulteriori alle modifiche apportate; in pratica è possibile "allegare" alla modifica effettuata, delle note in cui ad esempio si spiega il motivo per cui è stato necessario fare tali modifiche, eventuali criticità o qualsiasi altra informazione che si pensa possa essere utile a tutto il team di lavoro.
Con queste caratteristiche un VCS risolve i problemi più comuni dello sviluppo del software: l'evoluzione e la condivisione.

VCS Centralizzati e Distribuiti

GIT è un sistema di versionamento distribuito (DVCS) che si contrappone a quelli centralizzati (CVCS), come ad esempio CVS o SVN. Un VCS centralizzato è un sistema progettato per avere una singola copia completa del repository, ospitato in uno o più server, dove gli sviluppatori salvano le modifiche apportate.
Avere un VCS che dipende completamente da un server centralizzato ha una conseguenza ovvia: se il server o il collegamento va giù, gli sviluppatori non saranno

in grado di salvare le modifiche. O se il repository centrale viene danneggiato, e non esiste il backup, il progetto va perso.
Nei sistemi distribuiti invece ogni sviluppatore ha una propria copia locale di tutto il repository e può salvare le modifiche ogni volta che vuole. Anche in GIT esiste un server remoto che contiene l'intero repository condiviso da tutti gli sviluppatori, ma, se in un certo momento il server che ospita il repository è giù, gli sviluppatori possono continuare a lavorare senza alcun problema, rimandando la registrazione delle modifiche nel repository condiviso in seguito.
Le differenze tra i due tipi di sistemi possono essere riassunte come segue: con un CVCS abbiamo una completa dipendenza da un server remoto per svolgere il controllo di versione, mentre con un DVCS il server remoto è solo un'opzione per condividere le modifiche.

4.1.2 - Docker

Docker è un progetto open-source che automatizza il deployment di applicazioni all'interno di container software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Docker utilizza le funzionalità di isolamento delle risorse del kernel Linux come ad esempio ***cgroups*** e ***namespaces*** per consentire a "container" indipendenti di coesistere sulla stessa istanza di Linux, evitando l'installazione e la manutenzione di una macchina virtuale. I namespace del kernel Linux per lo più isolano ciò che l'applicazione può vedere dell'ambiente operativo, incluso l'albero dei processi, la rete, gli ID utente ed i file system montati, mentre i cgroups forniscono l'isolamento delle risorse, inclusa la CPU, la memoria, i dispositivi di I/O a blocchi e la rete. A partire dalla versione 0.9, Docker include la libreria **libcontainer** per poter utilizzare direttamente le funzionalità di virtualizzazione del kernel Linux.

Un Docker container, a differenza di una macchina virtuale, non include un sistema operativo separato. Al contrario, utilizza le funzionalità del kernel e sfrutta

l'isolamento delle risorse (CPU, memoria, I/O a blocchi, rete) ed i namespace separati per isolare ciò che l'applicazione può vedere del sistema operativo.

4.1.3 - Docker-compose

Docker-compose è uno strumento per lanciare applicazioni multi-container. Attraverso il file *docker-compose.yml*, è possibile specificare tutti i parametri necessari all'avvio dei container (variabili d'ambiente, eseguire comandi, esporre porte, ecc) eseguendo così il provisioning del container, ed esplicitare relazioni di dipendenza fra questi, in modo da lanciare i diversi servizi che compongono l'applicazione con un unico comando.

4.2 - Esecuzione

hdfs-site

Contiene settaggi di configurazione per il demone HDFS, informazioni sul namespace, delle directory dove memorizzare i blocchi HDFS, sull'uso o meno del NameNode secondario. Qui possiamo configurare eventuali politiche di replica e permessi di scrittura e lettura del HDFS.

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>file:///root/hdfs/namenode</value>
        <description>NameNode directory for namespace and transaction logs storage.</description>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>file:///root/hdfs/datanode</value>
        <description>DataNode directory</description>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>
</configuration>
```

Dove:

- **dfs.namenode.name.dir**: Posizione nel file system locale dove il NameNode mantiene il namespace ed i log.
- **dfs.datanode.data.dir**: lista di percorsi sul file system locale, separati da virgola, dove il DataNode può memorizzare i blocchi.
- **dfs.replication**: replication factor dei blocchi.

core-site

In questo file XML comunichiamo ad Hadoop dove il Namenode si trova nel cluster, la cartella che contiene i checkpoint e altre proprietà relative ai checkpoint.

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://hadoop-master:9000/</value>
    </property>
</configuration>
```

Dove:

- **fs.defaultFS**: URI del NameNode

yarn-site

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>hadoop-master</value>
    </property>
</configuration>
```

Dove:

- **yarn.resourcemanager.hostname**: URI del ResourceManager
- **yarn.nodemanager.aux-services**: Seleziona un servizio di shuffle necessario per l'esecuzione di MapReduce.
- **yarn.nodemanager.aux-services.mapreduce_shuffle**: In congiunzione con la proprietà yarn.nodemanager.aux-services, seleziona "direct shuffle" come default per MapReduce.

Dockerfile

```
FROM ubuntu:16.04

WORKDIR /root

# install openssh-server, openjdk and wget
RUN apt-get update && apt-get install -y openssh-server openjdk-8-jdk wget nano

# install hadoop 2.8.0
RUN wget http://mirror.nohup.it/apache/hadoop/common/hadoop-2.8.0/hadoop-2.8.0.tar.gz && \
    tar -xzvf hadoop-2.8.0.tar.gz && \
    mv hadoop-2.8.0 /usr/local/hadoop && \
    rm hadoop-2.8.0.tar.gz

# set environment variable
ENV JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
ENV HADOOP_HOME=/usr/local/hadoop
ENV PATH=$PATH:/usr/local/hadoop/bin:/usr/local/hadoop/sbin

# ssh without key
RUN ssh-keygen -t rsa -f ~/.ssh/id_rsa -P '' && \
    cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

RUN mkdir -p ~/hdfs/namenode && \
    mkdir -p ~/hdfs/datanode && \
    mkdir $HADOOP_HOME/logs

COPY config/* /tmp/

RUN mv /tmp/ssh_config ~/.ssh/config && \
    mv /tmp/hadoop-env.sh /usr/local/hadoop/etc/hadoop/hadoop-env.sh && \
    mv /tmp/hdfs-site.xml $HADOOP_HOME/etc/hadoop/hdfs-site.xml && \
    mv /tmp/core-site.xml $HADOOP_HOME/etc/hadoop/core-site.xml && \
    mv /tmp/mapred-site.xml $HADOOP_HOME/etc/hadoop/mapred-site.xml && \
    mv /tmp/yarn-site.xml $HADOOP_HOME/etc/hadoop/yarn-site.xml && \
    mv /tmp/slaves $HADOOP_HOME/etc/hadoop/slaves && \
    mv /tmp/start-hadoop.sh ~/start-hadoop.sh && \
    mv /tmp/entrypoint.sh ~/entrypoint.sh && \
    mv /tmp/run-wordcount.sh ~/run-wordcount.sh

RUN chmod +x ~/start-hadoop.sh && \
    chmod +x ~/entrypoint.sh && \
    chmod +x ~/run-wordcount.sh && \
    chmod +x $HADOOP_HOME/sbin/start-dfs.sh && \
    chmod +x $HADOOP_HOME/sbin/start-yarn.sh

# format namenode
RUN /usr/local/hadoop/bin/hdfs namenode -format

ENTRYPOINT ["./entrypoint.sh"]
```

- installazione software necessari: openssh-server openjdk-8-jdk wget nano.
- con wget viene scaricato hadoop e spostato in */usr/local/hadoop*.
- aggiunta dei file di configurazione all'interno dell'immagine.
- permessi di esecuzione.
- formattazione del datanode.
- script di entrata (entrypoint) del container, prende in input i parametri passati all'avvio.

docker-compose.yml

```yaml
version: '2'

services:
  master:
    image: chrispiemo/hadoop:1.0
    container_name: hadoop-master
    hostname: hadoop-master
    depends_on:
      - slave1
      - slave2
      - slave3
    networks:
      - hadoop
    ports:
      - "50070:50070"
      - "8088:8088"
    command: master

  slave1:
    image: chrispiemo/hadoop:1.0
    container_name: hadoop-slave1
    hostname: hadoop-slave1
    networks:
      - hadoop
    command: slave

  slave2:
    image: chrispiemo/hadoop:1.0
    container_name: hadoop-slave2
    hostname: hadoop-slave2
    networks:
      - hadoop
    command: slave

  slave3:
    image: chrispiemo/hadoop:1.0
    container_name: hadoop-slave3
    hostname: hadoop-slave3
    networks:
      - hadoop
    command: slave

networks:
  hadoop:
    external: true
```

Master:

- **image**: immagine per l'avvio del container.
- **container-name**: nome del container.
- **hostname**: nome con cui il container conosce se stesso.
- **depends_on**: garantisce che il servizio venga avviato dopo quelli specificati.
- **networks**: nome della rete.
- **ports**: porte mappate con l'host.
- **command**: input per l'entrypoint.

Slaves:

- **image**: immagine per l'avvio del container.
- **container-name**: nome del container.
- **hostname**: nome con cui il container conosce se stesso.
- **networks**: nome della rete.
- **command**: input per l'entrypoint.

Networks:

- **hadoop**: verifica la presenza del network già esistente di nome hadoop.

Esecuzione su Linux:

1. Installare docker, docker-compose and git
2. Clonare il repository:

```
git clone https://github.com/chrisPiemonte/docker-hadoop-cluster.git
cd docker-hadoop-cluster/
```

3. Creare il network :

```
docker network create hadoop
```

4. Eseguire il comando:

```
docker-compose -p hadoop up -d
```

5. Per testare:

```
docker exec -it hadoop-master ./run-wordcount.sh
```

5 - Conclusione

MapReduce si è rivelato utile per molte situazioni, ma non per tutte; altri modelli si prestano meglio alle richieste di Graph processing o iterative modeling, per citarne alcuni. Inoltre MapReduce è prevalentemente batch-oriented. La separazione delle funzionalità ha portato quindi importanti benefici, soprattutto per il client che può focalizzarsi sullo sviluppo dell'applicazione, tralasciando gli aspetti di gestione ed evitando la migrazione dei dati fra HDFS e diversi file system.

Con l'introduzione di YARN, parte delle responsabilità dello JobTracker di MapReduce sono state delegate al ResourceManager ed agli ApplicationMaster. Questo ha reso il layer di Cluster Resource Management più generale e ha reso possibile focalizzarsi maggiormente sugli aspetti implementativi. Infatti sono state raggiunte prestazioni migliori, dovute all'alleggerimento del carico del master (migliorando la scalabilità) e sono nate nuove possibilità computazionali, sfociate in una nuova ondata di progetti, Open Source e non.

GitHub repo: https://github.com/chrisPiemonte/docker-hadoop-cluster.git