

# Apache Hadoop : un système d'exploitation distribué pour le Big Data

**Jonathan Lejeune**

Sorbonne Université/LIP6-INRIA



# Présentation

## Qu'est ce que c'est ?

Un système open-source :

- développé en Java par la fondation Apache
- adapté pour les données massives
  - Stockage distribué et large échelle
  - Gestion de ressources distribuées
  - Applications de traitement via le modèle du Map-Reduce

## Bref historique

- 2006 : création par Doug Cutting (version 0.1.0)
- 2008 : Yahoo utilise Hadoop pour l'indexage de page web
- 2009 : Yahoo ! tri 1 To en 62 sec avec Hadoop
- fin 2013 : Hadoop 2, apparition de Yarn
- fin 2017 : 75% des données mondiales seraient stockées sur HDFS
- janv. 2019 : Hadoop 3, apport d'optimisations



# Utilisateurs d'Hadoop

- Amazon (EMR ? Elastic MapReduce )
- Yahoo !
- Criteo
- Facebook
- IBM : Blue Cloud
- EBay
- LinkedIn
- Spotify
- Twitter
- New York Times
- PowerSet (search engine natural language)
- Veoh (online video distribution)

Plein d'autres sur <http://wiki.apache.org/hadoop/PoweredBy>

# Composants Internes

## Hadoop Map Reduce

- Une application exécutant des programmes Map-Reduce sur YARN.
- Une API pour coder des programmes Map-Reduce

## YARN

- Système d'ordonnancement et de gestion de ressources de cluster

## Hadoop Distributed File System

- Système de fichiers distribué adapté aux gros volumes de données

Hadoop  
Map Reduce



Yet Another Resource Negotiator



Hadoop Distributed File System

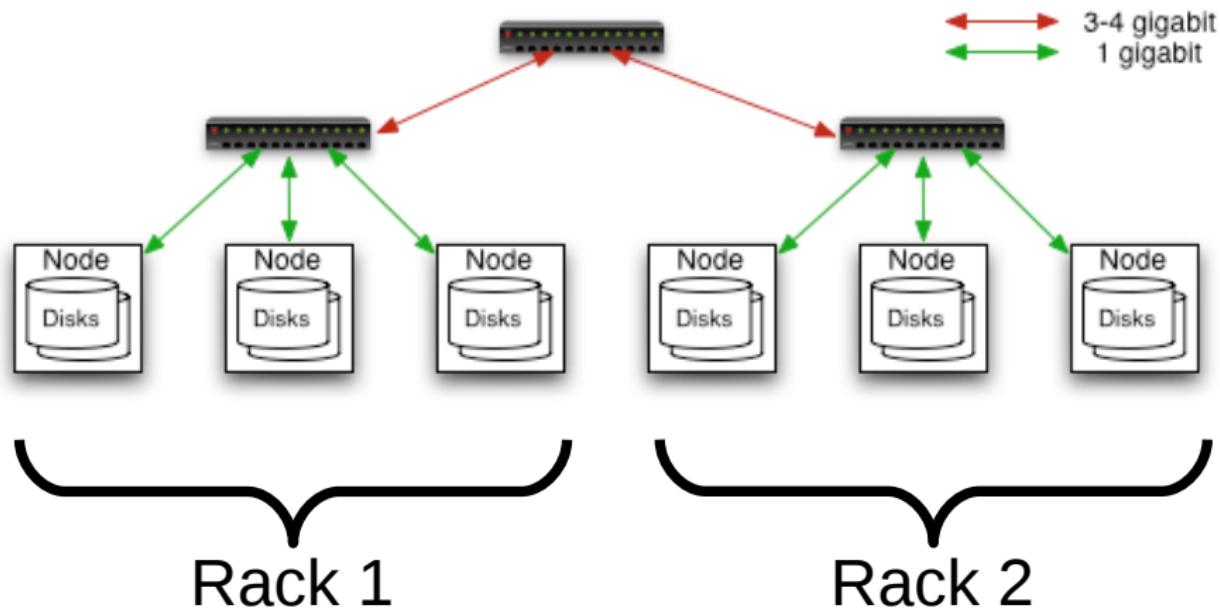


# Écosystème hadoop



# Architecture physique

- Exécution sur une grille de machines
- Les machines sont groupées par rack



# Hadoop Distributed File System



# HDFS : Généralités

## Qu'est ce que c'est ?

Système de fichiers distribué, tolérant aux pannes et conçu pour stocker des fichiers massifs (> To).

## Caractéristiques

- Les fichiers sont divisés en blocs de 128 Mo (valeur par défaut)
  - ⇒ adapté aux gros fichiers
  - ⇒ inadapté au multitude de petits fichiers
- Mode 1 écriture/plusieurs lectures :
  - Lecture séquentielle
    - ⇒ on privilégie le débit à la latence
    - ⇒ la lecture d'une donnée implique une latence élevée
  - Écriture en mode append-only en exclusion mutuelle
    - ⇒ on ne peut pas modifier les données existantes d'un fichier
    - ⇒ impossible d'avoir plusieurs écrivains, ou de modifier un endroit arbitraire du fichier

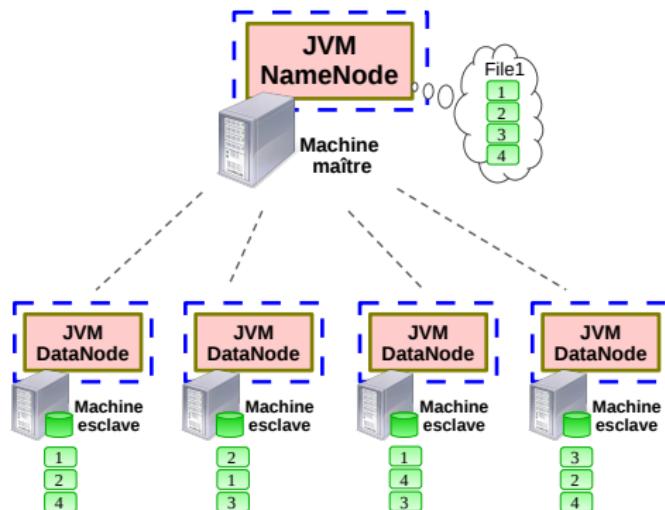
# HDFS : Aperçu global

## Une architecture maître-esclave

- la JVM maître HDFS :  
**le NameNode**
- les JVM esclaves HDFS :  
**les DataNodes**

## Stockage des blocs

- Stockage physique des blocs sur les FS locaux des Datanodes
- Chaque bloc est répliqué 3 fois par défaut
- Les répliques sont localisées sur des DataNodes différents



# HDFS : le NameNode

## Rôle du NameNode

- Expose un espace de nom arborescent
- Gère l'allocation, la distribution et la réPLICATION des blocs
- Interface du HDFS avec l'extérieur

## Information et méta-données maintenues

- liste des fichiers
- liste des blocs pour chaque fichier et du DataNode hébergeur
- liste des DataNodes pour chaque bloc
- attributs des fichiers (ex : nom, date création, facteur de réPLICATION)

**Le nb de fichiers du HDFS est limité par la capacité mémoire du NameNode**

## Journalisation des actions sur un support persistant

lectures, écritures, créations, suppressions

# HDFS : le Datanode

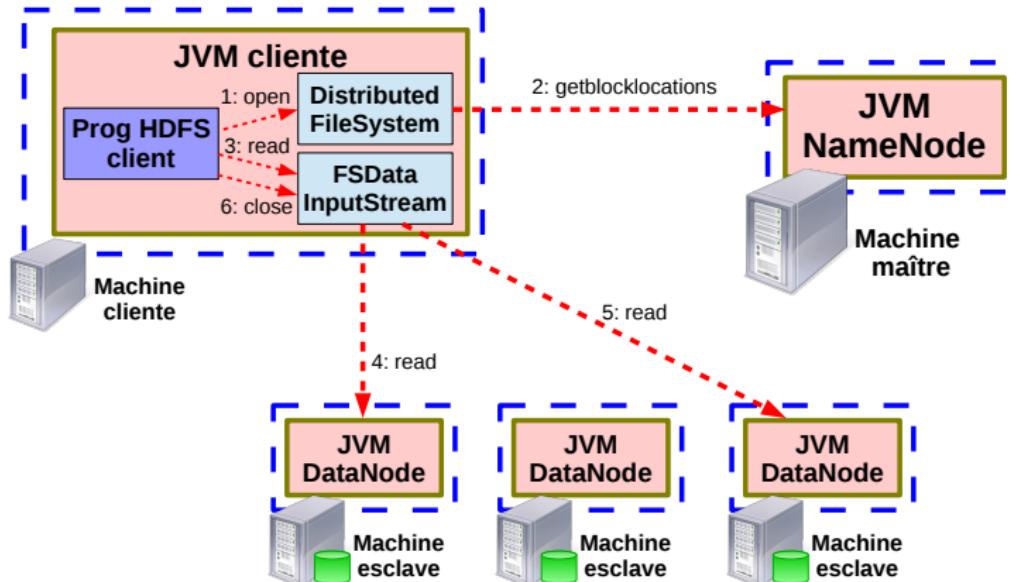
## Rôle du DataNode

- Stocke des blocs de données dans le système de fichier local
- Serveur de bloc de données et de méta-données pour le client HDFS
- Maintient des méta-données sur les blocs possédés (ex : CRC)

## Communication avec le NameNode

- Heartbeat :
  - message-aller : capacité totale, espace utilisé/restant
  - message-retour : des commandes
    - copie de blocs vers d'autres Datanodes
    - invalidation de blocs
    - ...
- Informe régulièrement le Namenode des blocs contenus localement

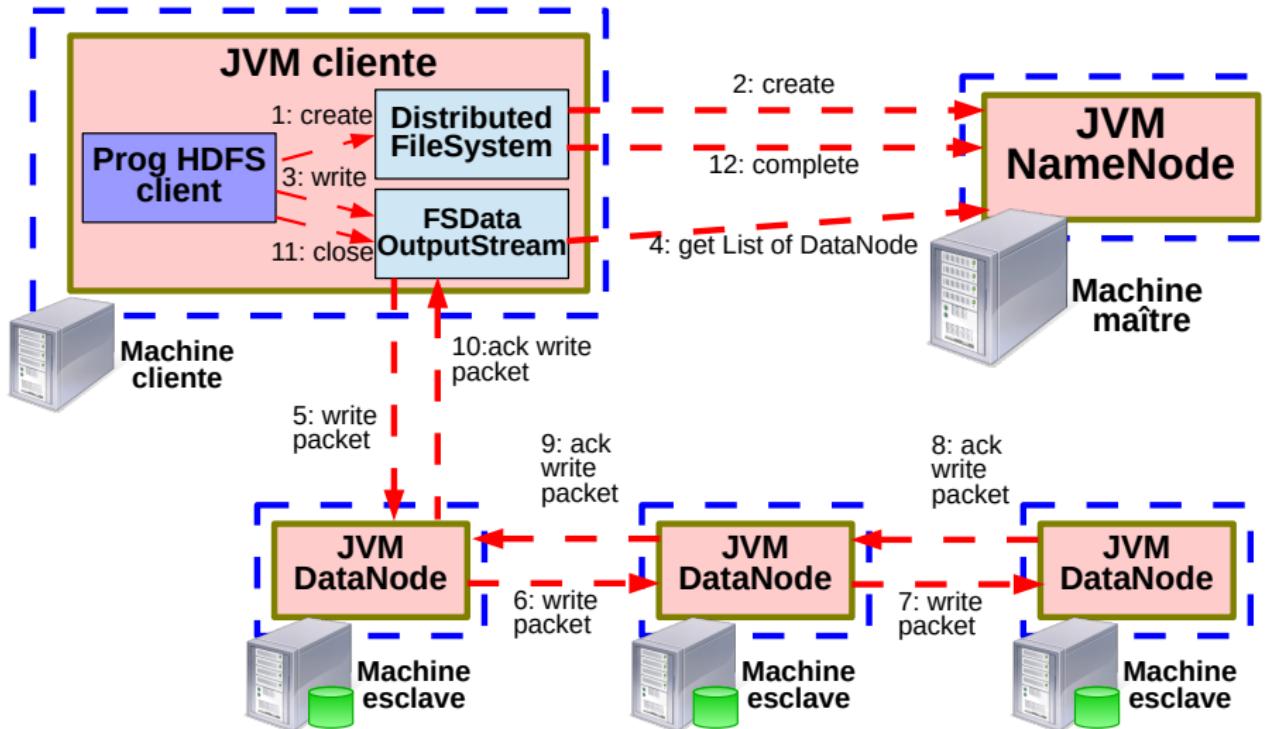
# HDFS : Lecture d'un fichier



## Avantages

- Prise en compte de la localité des blocs
- Les clients s'adressent directement aux DataNodes

# HDFS : Écriture d'un fichier



## Stratégie courante

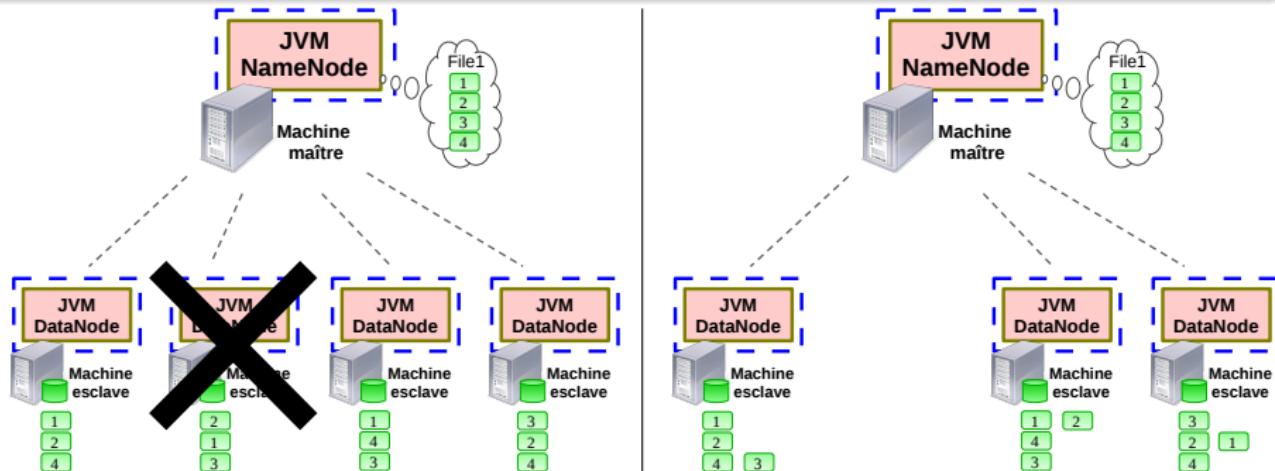
- un réplica sur le rack local
- un second réplica sur un autre rack
- un troisième réplica sur un rack d'un autre datacenter
- les réplicas supplémentaires sont placés de façon aléatoire

**Le client lit le plus proche réplica**

# HDFS : tolérance aux fautes

## Crash du Datanode :

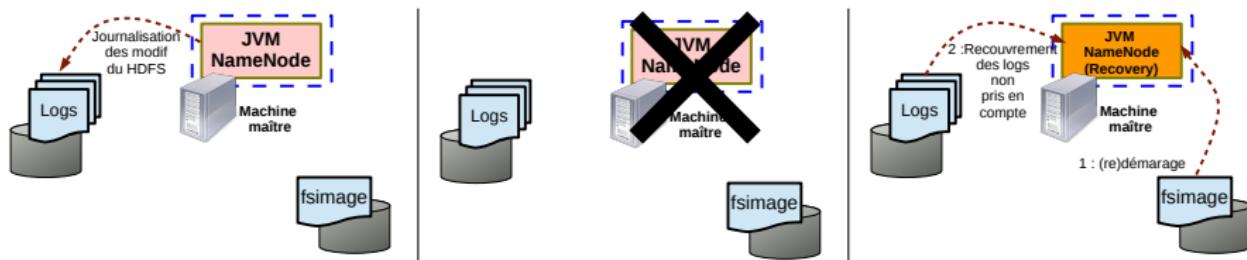
- les données sont toujours disponibles (réplications)
- Le NameNode ne reçoit plus de heartbeat :  
⇒ copie des réplicas manquants sur un autre DN



# HDFS : tolérance aux fautes

## Crash du NameNode :

- Le service devient indisponible
- Sauvegarde préventive des logs de transaction sur un support stable  
→ si redémarrage, chargement de la dernière image du HDFS et application des logs de transaction

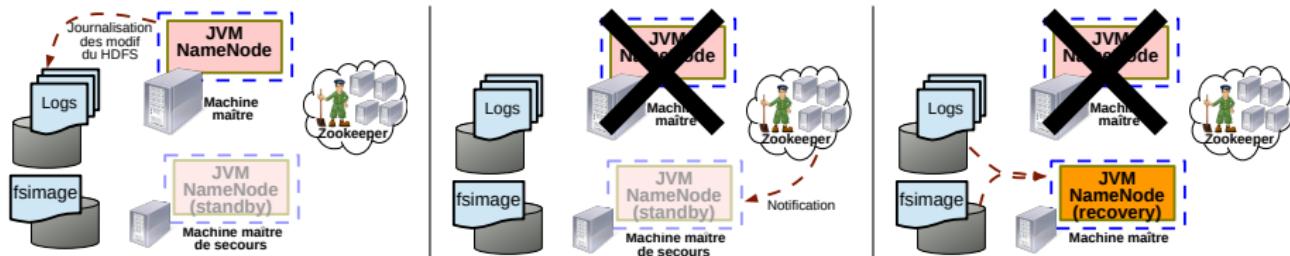


# HDFS : tolérance aux fautes

## Solution complémentaire aux crashes du NameNode

Avoir un NameNode de secours qui remplace automatiquement l'original :

- Utilisation de ZooKeeper pour détecter la panne du NameNode et notifier le NameNode de secours



# HDFS : tolérance aux fautes

## Problème !

Rares redémarrages du Namenode

- ⇒ énorme quantité de logs de transaction : **stockage conséquent**
- ⇒ Prise en compte de beaucoup de changements : **redémarrages longs**

## Solution

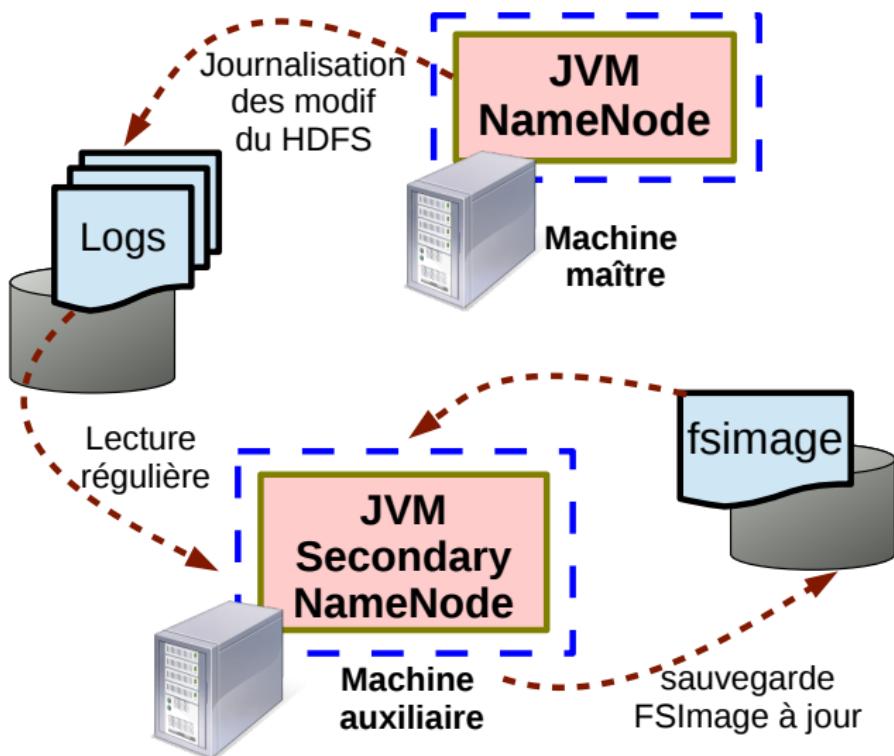
Produire des images HDFS plus récentes :

- utilisation d'un (ou plusieurs) processus distant(s) appelé **SecondaryNameNode**

## Attention

Le secondaryNameNode n'est pas un NameNode de secours.

# HDFS : le secondary Namenode

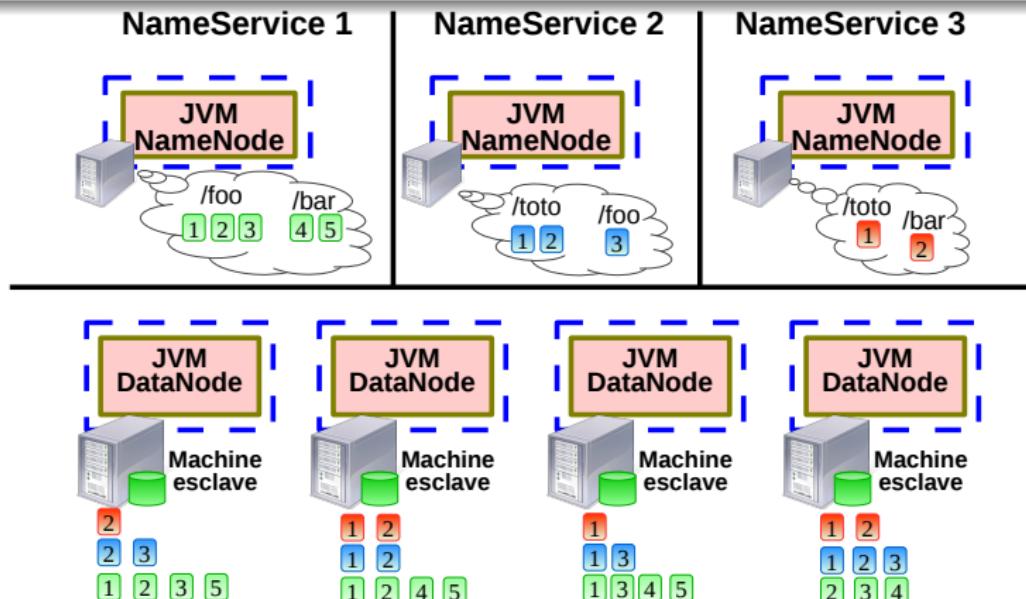


# Fédération de HDFS

## Objectif

Mutualiser les mêmes DataNode pour plusieurs instances de HDFS :

- ⇒ isolation entre plusieurs espaces de nom
- ⇒ équilibrage de charge parmi les NameNodes



## Manipuler le HDFS

- avec une API Java
- avec un terminal grâce à la commande :  
`hdfs dfs <commande HDFS>`

## Exemple de commandes

Commandes HDFS similaires celles d'Unix en ajoutant un tiret devant :

```
hdfs dfs -ls <path>
hdfs dfs -mv <src> <dst>
hdfs dfs -cp <src> <dst>
hdfs dfs -cat <src>
hdfs dfs -put <localsrc> ... <dst>
hdfs dfs -get <src> <localdst>
hdfs dfs -mkdir <path>
```

RTFM pour les autres :).

## Obtenir une référence Java du HDFS

- Le HDFS est représentée par un objet FileSystem :

```
final FileSystem fs = FileSystem.get(conf);
```

- L'argument configuration contient des properties JAVA indiquant l'URL du NameNode

## Quelques méthodes

```
//copie d'un fichier/dossier local client vers le HDFS  
fs.copyFromLocalFile(src,dst);  
//copie d'un fichier/dossier HDFS vers le fs local client  
fs.copyToLocalFile(sr,dst);  
//creation/effacement d'un fichier  
FSDataOutputStream out = fs.create(f);  
//test d'existence d'un fichier  
boolean b = fs.exists(f);  
//ouverture en écriture (append only)  
FSDataOutputStream out = fs.append(f);  
//ouverture en lecture  
FSDataInputStream in = fs.open(f);
```

# Yet Another Resource Negotiator



# YARN : Généralités

Qu'est ce que c'est ?

Service de gestion de ressources de calcul distribuées

## Caractéristiques

- Existe depuis la version 2 d'Hadoop
- Une architecture maître-esclaves
- Gestion d'applications distribuées :
  - ⇒ Allocation/placement de containers sur les nœuds esclaves
- Une container = une abstraction de ressources de calcul :
  - ⇒ couple <nb processeurs, quantité mémoire>

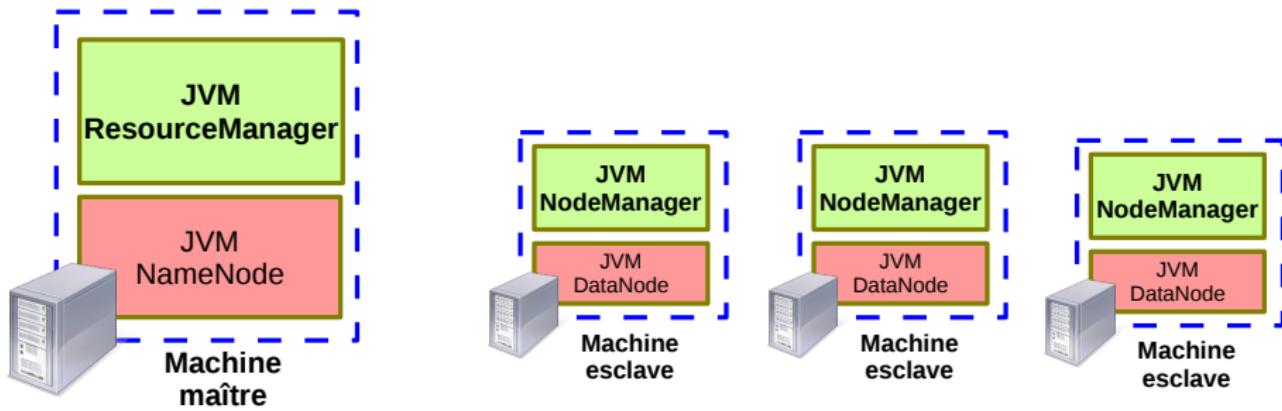
# YARN : les démons maîtres-esclaves

## ResourceManager (RM)

- JVM s'exécutant sur le nœud maître
- contrôle toutes les ressources du cluster
- ordonne les requêtes clientes

## NodeManager (NM)

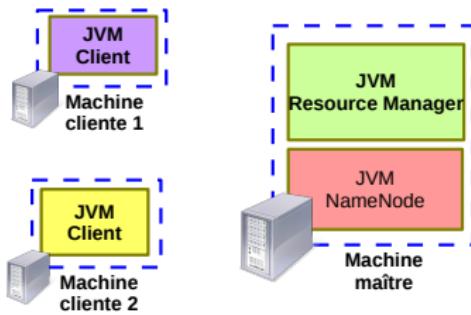
- une JVM par machine esclave
- gère les ressources du nœud
- lance les container et les monitore



# YARN : Composants d'une application

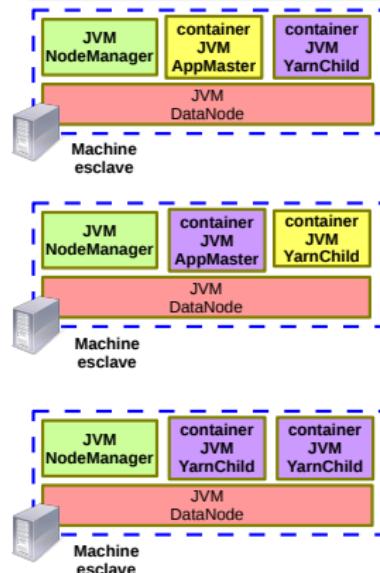
## Container ApplicationMaster (AM)

- container maître de l'application
- négocie avec le RM l'allocation de containers esclaves

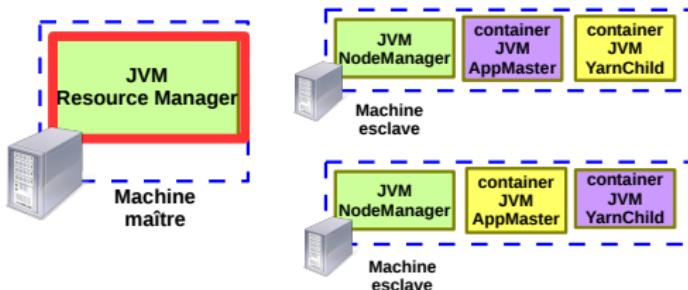


## Container YarnChild (YC)

- container esclave
- exécute une tâche de l'application



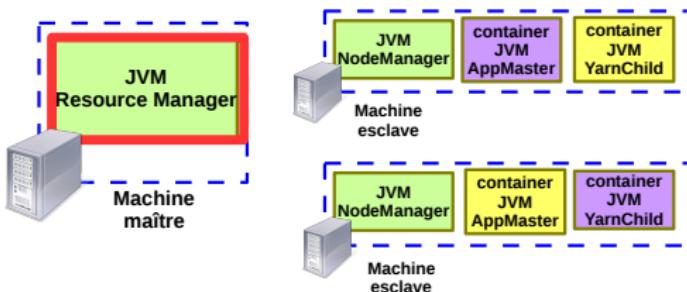
# YARN : ResourceManager



## Objectifs du ResourceManager

- Orchestrer l'ensemble des démons du cluster
- Décide où placer les conteneurs de chaque application
- Lien entre l'extérieur et les composants internes
- Vision globale de l'état du système

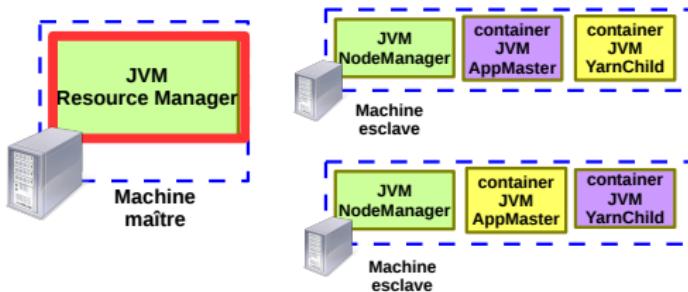
# YARN : ResourceManager



## Interface avec l'extérieur

- Traite les requêtes clientes :
  - soumission d'applications :
    - ⇒ allocation, déploiement et démarrage d'un container AM
    - ⇒ AM négocie ensuite l'allocation des YarnChild
  - suppression d'applications :
    - ⇒ Libération des containers de l'application
  - informations sur l'état courant des soumissions
- Traite les requêtes d'administration

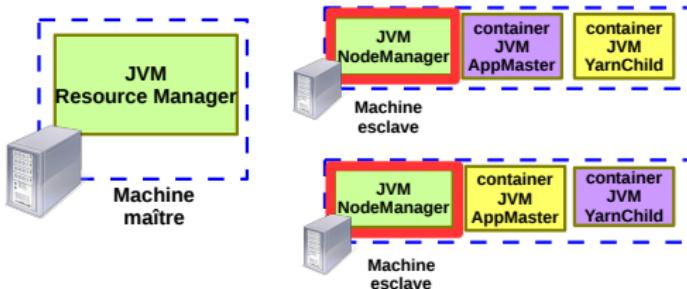
# YARN : ResourceManager



## Interface avec les composants internes

- Reçoit et ordonne les requêtes d'allocation de containers des AM (**YARN Scheduler**)
- Connaît de l'état de santé des NodeManagers
  - ressources allouées et disponibles sur chaque NM
  - affectation des containers sur les nœuds esclaves
- Gère la sécurité des communications entre les container d'une application :  
⇒ stockage et génération des clés de cryptage

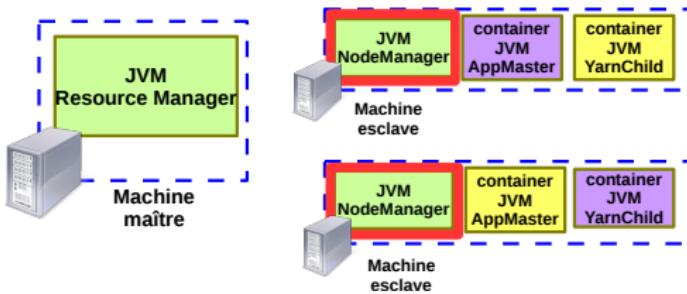
# YARN : NodeManager



## Rôle du NodeManager

- Héberge/ déploie localement des containers
- Libère un container à la demande du RM ou du AM correspondant
  - ⇒ Les données produites par la tâche du conteneur peuvent cependant être maintenues jusqu'à la fin de l'application
- Monitore l'état de santé du nœud physique et des containers hébergé
- Maintient des clés de sécurité provenant du RM pour d'autentifier l'utilisation des containers

# YARN : NodeManager

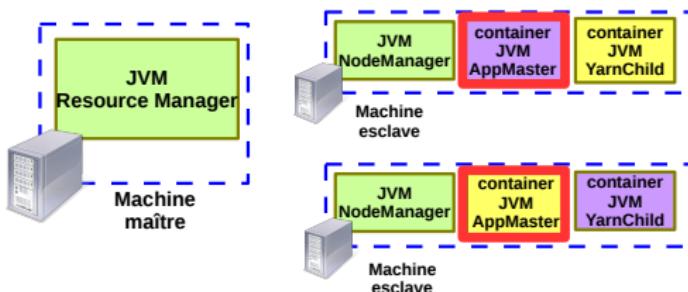


## Mode de communication avec le ResourceManager

### Mécanisme de Heartbeat :

- message aller :
  - états des containers hébergés
  - état du nœud
- message retour :
  - des clés de sécurités pour la communication avec les AM
  - des container à libérer
  - possibilité de directive d'arrêt ou de resynchronisation

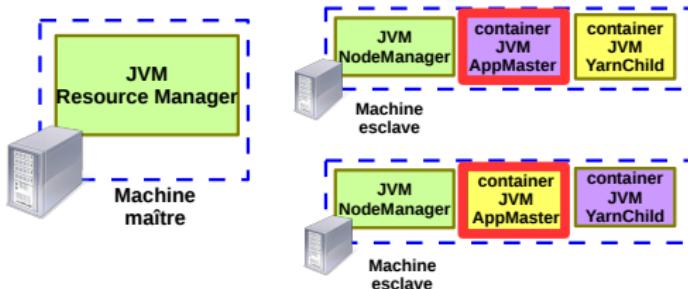
# YARN : Application Master



## Rôle d'un container Application Master

- Négocie l'allocation des ressources auprès du RM
- Collabore avec les NodeManager pour démarrer/arrêter les container alloués pour l'application
- Gère le cycle de vie de l'application
  - Ajustement de la consommation des ressources
  - La sémantique des communication avec les containers dépend de l'application

# YARN : Application Master (2/2)

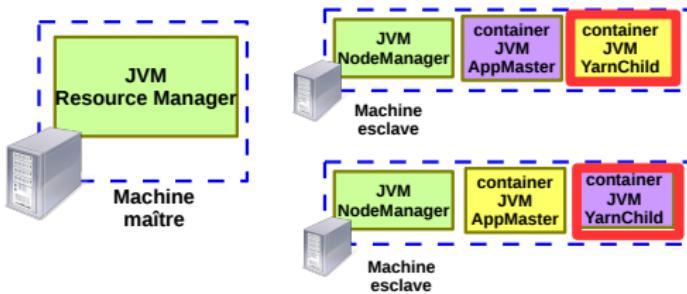


## Mode de communication avec le ResourceManager

### Hearbeat avec le ResourceManager

- message aller :
  - progrès courant de l'application
  - containers demandés : nombre requis, préférence de localité
  - container libérés
- message retour :
  - liste des containers nouvellement alloués et leur clé de sécurité
  - les ressources disponibles dans le cluster
  - possibilité de directive d'arrêt ou de resynchronisation

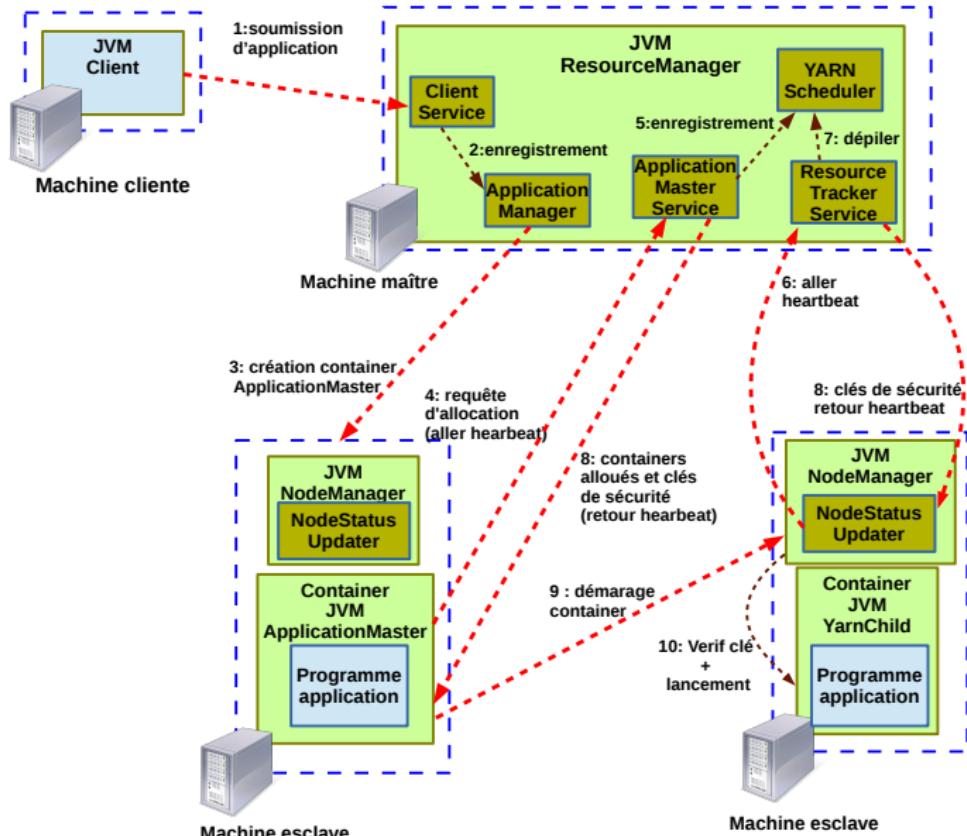
# YARN : Container



## Rôle d'un container YarnChild

- Exécute les tâches de l'application

# Schéma d'un déploiement d'application



# YARN : les différentes politiques du YARN Scheduler

## FIFO scheduler

Les requêtes sont traités dans l'ordre de leur arrivée

- Très simple
- Partage du cluster peu efficace

## Capacity scheduler

Plusieurs files d'attentes hiérarchiques avec une capacité max prédéfinie

- Les petites requêtes ne sont pas pénalisés
- Toutes les ressources ne sont pas utilisées

## Fair scheduler

La capacité allouée à chaque client est la même :

- Allocation dynamique, pas d'attente
- La capacité allouée à un job diminue avec la charge

## Actions préventives

Sauvegardes régulières de l'état du RM sur support stable (ex : HDFS) :

- liste de toutes les applications soumises et de leur méta-données
- liste des états des nœuds esclaves du cluster

## Au redémarrage du Resource Manager

- **1ère phase** : restauration de sa connaissance des applications soumises et de l'état des nœuds depuis le support stable
- **2ème phase** : restauration de sa vision du contexte d'exécution de toute les applications et de leurs conteneurs :
  - depuis les méta-données sauvegardées
  - puis progressivement via les heartbeats (statut des conteneur) des NM et des AM (requêtes de conteneur) :

**Permet de reconstruire l'ordonnanceur du RM sans redémarrer depuis le début l'ensemble des applications**

# YARN : tolérance aux crashes d'un nœud esclave

## Pour le NodeManager

Le NodeManager n'émet plus de heartbeat au RM

- Le RM supprime le nœud de ses esclaves opérationnels tant qu'il n'a pas redémarré

## Pour les ApplicationMaster

Les containers AppMaster n'émettent plus de heartbeat au RM

- redémarrage d'une nouvelle instance de l'application en créant un nouvel AppMaster l'application

## Pour les YarnChild

Les containers YarnChild ne peuvent plus communiquer avec leur AM

- Le traitement à faire dépend de l'application

## Mécanisme de recovery au redémarrage du NodeManager

# Les applications YARN

## Exemples d'applications pouvant s'exécuter sur YARN

- **Hadoop MapReduce** : traitement de jobs MapReduce
- **Spark** : traitement/analyse de large volume de données



- **Giraph** : calcul de graphe
- **Hbase** : Base de données NoSQL utilisant le HDFS



- **Tez** : Exécution de workflow complexes



Toutes ces applications peuvent s'exécuter sur le même cluster  
Hadoop

# Coder une application YARN

## From Scratch

- Pas de couche logicielle intermédiaire entre l'appli et Yarn
- Assez complexe et fastidieux à programmer
- Adapté si l'application a des exigences complexes d'ordonnancement

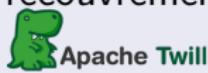
## Apache Slider

Couche logicielle permettant d'exécuter et de déployer des plateformes distribuées existantes.



## Apache Twill

- Programmation simplifiée : définition des tâches via des Java Runnable
- Des outils intégrés : Logging temps-réel, messages de commande (du client vers les Runnables), recouvrement d'état, ....



# Hadoop Map Reduce



## Qu'est ce que c'est ?

Appli Yarn pour l'exécution et le déploiement de programmes Map-Reduce

### Responsabilité des containers YarnChild

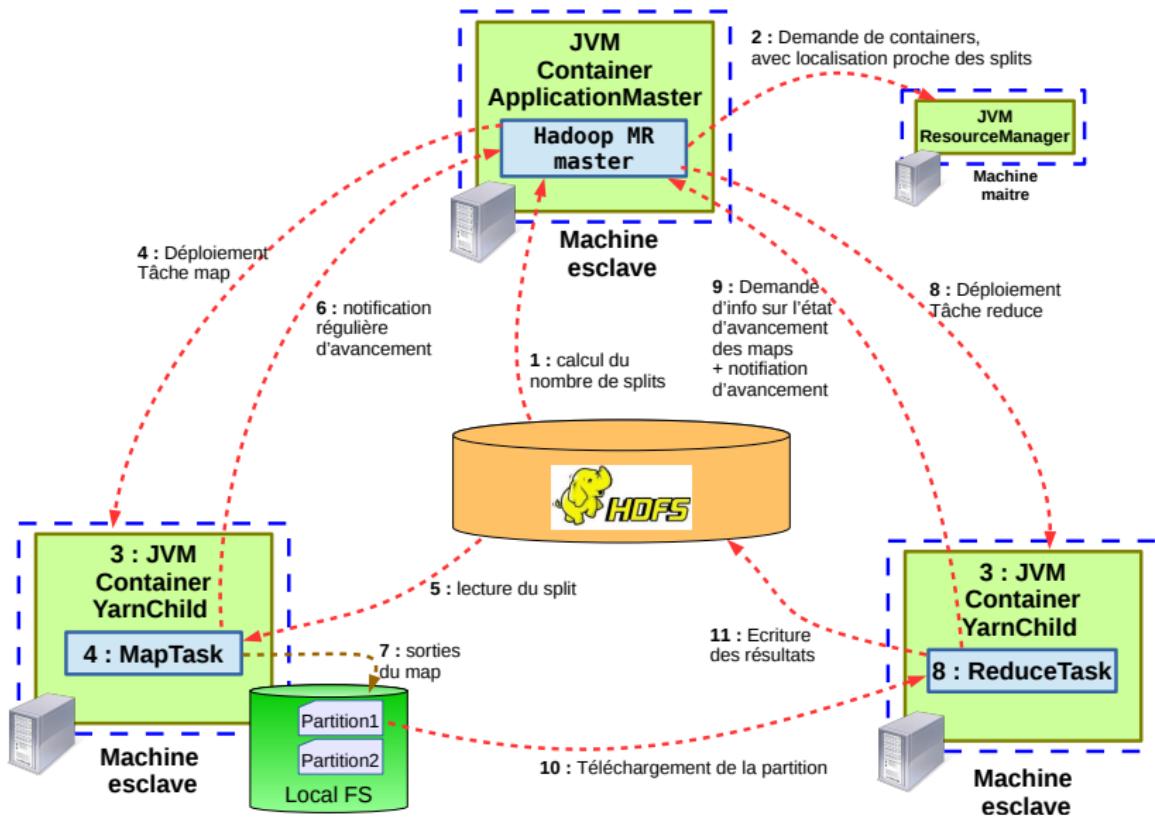
- exécutent une instance de tâche de map ou de reduce
- fournissent leur état d'avancement à l'AM correspondant

### Responsabilité du container AM

Coordination du Job Map-Reduce :

- Définit à sa création le nombre de containers à demander au RM
- Les requêtes de container prennent en compte la localité des données d'entrée du job
- Déploie les tâches dans les containers alloués par le RM :
  - **les maps** : le plus proche possible de leur split respectif
  - **les reduces** : à partir d'une certaine proportion de maps terminés

# Hadoop MapReduce : Exécution d'un Job



# Hadoop MapReduce : Spéculation

## Principe

Possibilité de démarrer de nouvelles instances de tâche en cours d'exécution

## Mécanismes

- la décision d'un lancement d'une spéculation est faite par l'AppMaster
- si la spéculation est jugée nécessaire, affectation d'une nouvelle instance de tâche dans un container.

## Intérêt

Permettre d'anticiper les tâches jugées trop lentes et donc potentiellement défaillante

# Hadoop MapReduce : Écriture des données finales

## Principe

**Chaque reduce écrit ses résultats dans son fichier HDFS**

## Problème

Plusieurs instances d'une même tâche peuvent s'exécuter. Comment assurer la cohérence sur le fichier HDFS ?

## Solution

- 1) chaque instance écrit dans un fichier temporaire
- 2) L'appMaster désigne l'instance qui peut copier définitivement ses données sur le HDFS (en général la première qui se termine)
- 3) Les autres instances ainsi que leur données sont détruites une fois que la copie s'est terminée

## Définir ses types de clé/valeur

- les valeurs doivent implémenter l'interface `Writable` de l'API Hadoop
- les clés doivent implémenter l'interface `WritableComparable<T>` (= un extends de `Writable` et `Comparable<T>` de Java)
- `Writable` contient deux méthodes :
  - `void write(DataOutput out)` : sérialisation
  - `void readFields(DataInput in)` : dé-sérialisation

## Quelques Writables prédéfinis dans l'API

`BooleanWritable`, `DoubleWritable`, `FloatWritable`, `IntWritable`, `LongWritable`,  
`Text`, `NullWritable`

## Définir le format des données d'entrée

- Toute classe héritant de `InputFormat<K, V>`.
- Classe fournies par l'API Hadoop :
  - `TextInputFormat` (format par défaut)
    - clé : `LongWritable`, offset du premier caractère de la ligne
    - valeur : `Text`, contenu de la ligne
  - `SequenceFileInputFormat<K, V>` : format binaire
  - `KeyValueTextInputFormat` : `<Text, Text>` avec séparateur

## Définir le format des données de sortie

- Toute classe héritant de `OutputFormat<K, V>`.
- Classe fournies par l'API Hadoop :
  - `TextOutputFormat` (format par défaut)
    - clé : `Text`, première partie de la ligne + un séparateur
    - valeur : `Text`, deuxième partie de la ligne
  - `SequenceFileOutputFormat<K, V>`

# API JAVA Hadoop MapReduce : Codage du Map

## Principe

- Classe héritant de la classe `Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`
- redéfinition de la méthode **map**, appelée à chaque lecture d'une nouvelle paire clé/valeur dans le split :

```
protected void map(KEYIN key, VALUEIN value, Context context);
```

## Exemple : Mapper du WordCount

```
public class WordCountMapper extends  
    Mapper<LongWritable, Text, Text, IntWritable> {  
  
    public void map(LongWritable key, Text value, Context context)  
        throws IOException, InterruptedException {  
  
        StringTokenizer itr = new StringTokenizer(value.toString());  
        while (itr.hasMoreTokens()) {  
            String word = itr.nextToken();  
            context.write(new Text(word), new IntWritable(1));  
        }  
    }  
}
```

## Principe

- Classe héritant de la classe `Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>`
- redéfinition de la méthode `reduce`, appelée à chaque lecture d'une nouvelle paire clé/list(valeur) :

```
protected void reduce(KEYIN key, Iterable<VALUEIN> values,
                      Context context)
```

## Exemple : Reducer du WordCount

```
public class WordCountReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values,
                      Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values)
            sum += val.get();
        result.set(sum);
        context.write(key, new IntWritable(sum));
    }
}
```

## Principe

- défini la politique de répartition des sorties de map
- classe héritant de la classe abstraite `Partitioner<KEY, VALUE>`
- implémentation de la méthode abstraite `getPartition`, appelée par le `context.write()` du map :

```
public abstract int getPartition(KEY key, VALUE value,  
                                int numPartitions);
```

## Partitionneur par défaut

Hachage de la clé (`HashPartitioner`) :

```
public class HashPartitioner<K, V> extends Partitioner<K, V>{  
    public int getPartition(K key, V value,  
                           int numReduceTasks) {  
        return(key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;  
    }  
}
```

# API JAVA Hadoop MapReduce : squelette client

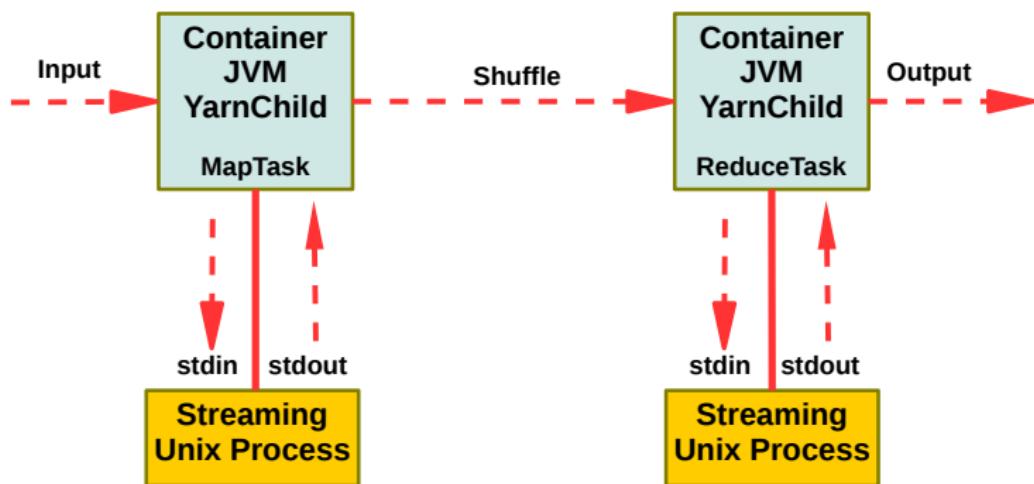
```
public class MyProgram {  
    public static void main(String[] args){  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf, "MonJob");  
        job.setJarByClass(MyProgram.class); // jar du programme  
        job.setMapperClass(...); // classe Mapper  
        job.setReducerClass(...); // classe Reducer  
        job.setMapOutputKeyClass(...); // classe clé sortie map  
        job.setMapOutputValueClass(...); // classe valeur sortie map  
        job.setOutputKeyClass(...); // classe clé sortie job  
        job.setOutputValueClass(...); // classe valeur sortie job  
        job.setInputFormatClass(...); // classe InputFormat  
        job.setOutputFormatClass(...); // classe OutputFormat  
        job.setPartitionerClass(HashPartitioner.class); // partitioner  
        job.setNumReduceTasks(...); // nombre de reduce  
        FileInputFormat.addInputPath(job, ...); // chemins entrée  
        FileOutputFormat.setOutputPath(job, ...); // chemin sortie  
        job.waitForCompletion(true); // lancement du job  
    }  
}
```

# Hadoop Streaming : Map-Reduce pour tout langage

## Principe

Externalisation du code (Python, Ruby, Bash, C, ...):

- Création d'un nouveau processus avec redirection des flux standards
- Les <clé,valeur> d'entrée sont lues sur l'entrée standard
- Les <clé,valeur> de sortie sont émises sur la sortie standard



Version stable (3.3.1) disponible depuis juin 2021.

## Principales évolutions

- Java 8 ou supérieur
- Erasure Coding pour les données froides
  - ⇒ Réduit le coût de la réPLICATION de blocs de 200% à 50%
- Amélioration du Shell
- Optimisation des I/O
  - ⇒ Performance Shuffle augmenté de 30%
- plusieurs NameNode de secours possibles
- une meilleure gestion des ports d'écoute
- Compatibilité avec le système de fichier de Microsoft Azure

## Références

- [1] Jeffrey Dean and Sanjay Ghemawat, MapReduce : Simplified Data Processing on Large Clusters, OSDI'04 : Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- [2] Hadoop : the definitive guide 4th edition, White Tom, O'Reilly, 2015, ISBN : 978-1-491-90163-2
- [3] Hadoop web site : <http://hadoop.apache.org/>
- [4] <http://hortonworks.com/hadoop/yarn/>