



ÉCOLE NATIONALE SUPÉRIEURE DES TÉLÉCOMMUNICATIONS

TRAVAUX PRATIQUES

Projet de groupe : Hadoop & Cloud

Réalisé par :

Yassine MERNISSI ARIFI

Julien LAFRANCE

Omar FEKIH-HASSEN

Alexandre DONNAT

Dirigé par :

M. Marc Jeanmougin

MS IA Expert Data & MLOps

Année universitaire 2025-2026

1 Introduction et objectifs

L'objectif de ce projet est de configurer un cluster Hadoop fonctionnel en équipe et de valider son bon fonctionnement à travers différents composants.

Le travail demandé consiste à :

- Déployer **HDFS** en mode *fully distributed*, afin de stocker les données de manière répartie entre les noeuds du cluster.
- Utiliser **YARN** pour gérer les ressources de nos noeuds et exécuter un job MapReduce en parallèle, puis implémenter nos propres `mapper.py` et `reducer.py` via Hadoop Streaming.
- Installer et configurer **Zookeeper**, nécessaire au bon fonctionnement de certains services distribués comme Hbase.
- Mettre en place **HBase** et vérifier son intégration avec Zookeeper.
- Lancer des tests avec **Spark** (PySpark) en mode cluster sous YARN.

Ce rapport présente de manière synthétique les choix techniques réalisés, les configurations mises en place et des usecases permettant d'illustrer le bon fonctionnement du cluster.

2 Architecture du cluster

Notre cluster est composé de quatre machines virtuelles auxquelles nous accédons via le réseau interne (bridge). Ces machines, nommées `tp-hadoop-9`, `tp-hadoop-22`, `tp-hadoop-30` et `tp-hadoop-31`, constituent l'infrastructure sur laquelle nous avons déployé Hadoop et ses composants après ajout de nos clés publiques de nos ordinateurs personnels sur le bridge.

TABLE 1 – Caractéristiques des machines du cluster

Machine	vCPU	RAM	Disque	Responsable
tp-hadoop-9	4	8 GB	25 GB	Yassine
tp-hadoop-22	4	8 GB	25 GB	Julien
tp-hadoop-30	4	8 GB	25 GB	Omar
tp-hadoop-31	4	8 GB	25 GB	Alexandre

Afin de permettre la communication entre les noeuds, nous avons distribué les clés publiques SSH de chaque machine dans le fichier `authorized_keys` des autres. Ainsi, chaque noeud peut se connecter en SSH sans mot de passe aux autres, ce qui rend possible la communication intra-cluster possible pour améliorer au bon fonctionnement d'Hadoop.

3 Installation, choix techniques et application

Vous pouvez retrouver tous les fichiers de configuration et scripts Python sur : <https://github.com/alexandremathiasdonnat/hadoopcluster-building>

3.1 HDFS

3.1.1 Brève introduction

HDFS (Hadoop Distributed File System) est le système de fichiers distribué de Hadoop. Il repose sur deux types de noeuds :

- **NameNode** : Gère la métadonnée (arborescence des fichiers, informations sur la localisation des blocs, réplication factor).
- **DataNodes** : Stockent physiquement les blocs de données et exécutent les opérations de lecture/écriture demandées par le NameNode.

Ce modèle permet de distribuer le stockage de fichiers volumineux sur plusieurs machines, tout en assurant la tolérance aux pannes grâce à la réplication des blocs.

3.1.2 Installation et configuration

Nous avons utilisé la version `hadoop-3.4.2-lean.tar.gz`, compatible avec `java-11-openjdk-amd64`. Hadoop requiert en effet Java 8 ou Java 11 selon la version. Afin d'uniformiser l'installation sur l'ensemble du cluster, nous avons appliqué les règles suivantes :

- Même version d'Hadoop installée sur chaque noeud
- Installation au même chemin (`/opt/hadoop`) sur chaque VM
- Même version de Java sur tous les noeuds (ici Java 11)

Les variables d'environnement (`HADOOP_HOME`, `JAVA_HOME`, etc. ainsi que le lien des chemins des binaires avec le `$PATH`) ont été exportées dans le fichier `.bashrc` dans chaque machine.

3.1.3 Paramètres principaux

Nous avons choisi la machine `tp-hadoop-22` comme **NameNode**. Nous avons également défini un **Secondary NameNode** sur la machine `tp-hadoop-9`. Celui-ci ne remplace pas le NameNode en cas de panne, mais sert à fusionner régulièrement les fichiers de métadonnées pour alléger la charge du NameNode. Pour permettre l'exécution correcte d'Hadoop, il a été nécessaire de modifier les fichiers de configuration XML situés dans `/opt/hadoop/etc/hadoop/`. Voici dans les grandes lignes les paramètres essentiels que nous avons configurés :

- `core-site.xml` (paramètres globaux)
 - `fs.defaultFS = hdfs://tp-hadoop-22:9000` Définit le système de fichiers par défaut et l'adresse du NameNode (`tp-hadoop-22` vu qu'il a été défini NameNode).
 - `hadoop.tmp.dir = /opt/hadoop/data` Répertoire temporaire utilisé par Hadoop.
- `hdfs-site.xml` (configuration HDFS)
 - `dfs.namenode.name.dir = /opt/hadoop/data/namenode` Répertoire où sont stockées les métadonnées du NameNode (ça nous évite d'avoir `/tmp`).
 - `dfs.blocksize = 134217728` La taille des blocks, trop grand : problème petits fichiers, trop petit : inefficace, on garde la valeur par défaut (128MB).
 - `dfs.datanode.data.dir = /opt/hadoop/data/datanode` Répertoire où les DataNodes stockent les blocs de données (ça nous évite d'avoir `/tmp`).
 - `dfs.replication = 3` Nombre de copies des blocs : nous avons choisi 3 comme compromis. Avec 3 noeuds, la perte d'un noeud reste tolérée, tandis qu'un facteur de 4 aurait représenté une précaution inutile et un surcoût de stockage.
 - `dfs.namenode.secondary.http-address = tp-hadoop-9:9868` Adresse du Secondary NameNode, configuré sur `tp-hadoop-9`.

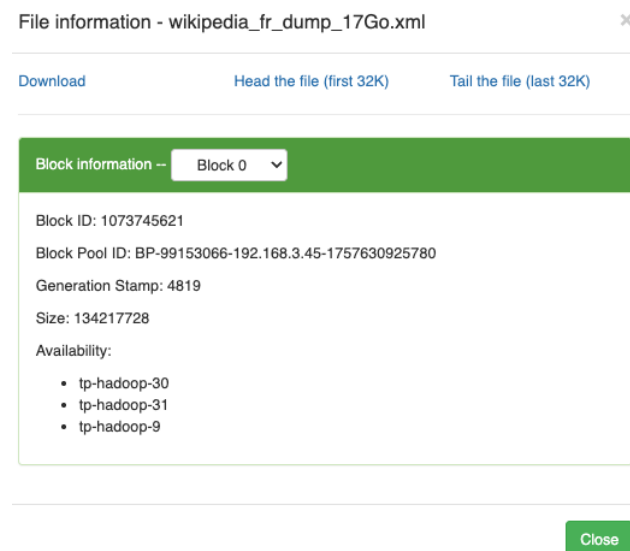
Le fichier `workers` a ensuite été configuré pour inclure l'ensemble des noeuds du cluster : `tp-hadoop-22`, `tp-hadoop-9`, `tp-hadoop-30` et `tp-hadoop-31`. Le NameNode (22) a également été déclaré DataNode, afin que toutes les machines participent au stockage distribué. Nous avons aussi créé un service `systemd` (`hadoop.service`) sur le NameNode pour lancer automatiquement HDFS et YARN au démarrage et éviter les arrêts non voulus.

3.1.4 Usages pratiques

Nous avons téléchargé le dump des articles de Wikipédia en français¹. Le fichier compressé au format BZ2 occupe environ 6,4 Go et atteint près de 26 Go une fois décompressé. Après l'avoir placé dans le répertoire `/data2/Public` accessible depuis le bridge, nous avons tenté de le transférer sur le HDFS en utilisant directement l'API WebHDFS exposée par le NameNode (`tp-hadoop-22`), au moyen de la commande `curl -i -X PUT`. Le transfert a échoué par manque d'espace. La commande `hdfs dfsadmin -report` indiquait environ 15 Go par noeud disponible, soit 60 Go au total. Avec un facteur de réplication de 3, la limite est $60/3 \approx 20$ Go, à laquelle s'ajoute un seuil de sécurité de 10 % de HDFS/YARN. La version décompressée de 26 Go dépassant cette limite, nous avons décidé de ne conserver que les 17 premiers Go du fichier

1. <https://dumps.wikimedia.org/frwiki/20250920/frwiki-20250920-pages-articles-multistream.xml.bz2>

décompressé, ce qui permet de rester sous le seuil imposé par la réplication. Cette portion a été réinjectée dans le HDFS via l'API WebHDFS et a servi de base à nos traitements avec Spark et MapReduce. Nous avons vérifié la bonne réplication des blocs avec la commande `hdfs fsck wikipedia_fr_dump_17Go.xml`, qui a indiqué un état **HEALTHY** avec les blocs répartis sur les 4 noeuds avec un facteur de réplication 3. L'interface du NameNode a confirmé cette répartition et l'UI accessible via le double pont SSH a fourni le détail de chaque bloc.



3.2 YARN

3.2.1 Brève introduction

YARN (Yet Another Resource Negotiator) est le gestionnaire de ressources de Hadoop. Il se compose d'un **ResourceManager**, chargé de l'allocation des ressources globales du cluster, et de plusieurs **NodeManagers**, qui exécutent les tâches localement sur chaque noeud.

3.2.2 Installation et configuration

Nous avons utilisé la même version d'Hadoop (`hadoop-3.4.2-lean`) et la même configuration Java que pour HDFS, avec les fichiers de configuration modifiés dans `/opt/hadoop/etc/hadoop/`.

3.2.3 Paramètres principaux

Nous avons choisi la machine `tp-hadoop-22` comme **ResourceManager**. Elle joue aussi le rôle de **NodeManager**, au même titre que `tp-hadoop-9`, `tp-hadoop-30` et `tp-hadoop-31`, afin que toutes les machines participent au calcul.

Voici dans les grandes lignes les paramètres essentiels que nous avons configurés :

- `mapred-site.xml` (configuration MapReduce sur YARN)
 - `yarn.app.mapreduce.am.env`, `mapreduce.map.env`, `mapreduce.reduce.env`
= `/opt/hadoop` Variables d'environnements pour le MapReduce
 - `mapreduce.framework.name = yarn` MapReduce doit s'exécuter sur YARN.
 - `mapreduce.jobhistory.address = tp-hadoop-22:10020`
 - `mapreduce.jobhistory.webapp.address = tp-hadoop-22:19888`

Ces paramètres activent le JobHistory Server : ils permettent de conserver l'historique des jobs MapReduce et de le consulter ensuite via une GUI sur le port 19888.

- `yarn-site.xml` (configuration YARN)
 - `yarn.resourcemanager.hostname = tp-hadoop-22` Définit le ResourceManager (on a décidé qu'il soit le même que sur le NameNode).
 - `yarn.nodemanager.aux-services = mapreduce_shuffle` Service de shuffle nécessaire à l'exécution des jobs MapReduce.
 - `yarn.nodemanager.resource.memory-mb = 3584` Mémoire maximale allouée par NodeManager (valeur choisie via `free -m`, en gardant une marge pour l'OS).
 - `yarn.nodemanager.resource.cpu-vcores = 4` Nombre de vCPU disponibles sur chaque machine (valeur choisie via `nproc`, utilisant le maximum de vCPUs).
 - `yarn.log-aggregation-enable = true` Activer le logging aggregate (pour avoir des preuves que les jobs se font sur plusieurs noeuds).

3.2.4 Usages pratiques

Nous avons validé le fonctionnement de notre cluster en lançant un job MapReduce de calcul de π : `hadoop jar /opt/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar pi 16 1000` Ce job (job_1759040748473_0001) a déclenché 16 tâches map réparties sur les quatre noeuds du cluster, suivies d'un unique reduce. L'exécution en parallèle a permis d'obtenir une estimation de $\pi \approx 3.1425$ en une vingtaine de secondes. Nous avons également retrouvé l'historique des MAP ATTEMPTS de ce job via le GUI (nodes différents) :

Show 20 entries				
Attempt	State	Status	Node	Logs
attempt_1759040748473_0001_m_000001_0	SUCCEEDED	Generated 1000 samples.	/default-rack/tp-hadoop-31:8042	logs
attempt_1759040748473_0001_m_000004_0	SUCCEEDED	Generated 1000 samples.	/default-rack/tp-hadoop-30:8042	logs
attempt_1759040748473_0001_m_000000_0	SUCCEEDED	Generated 1000 samples.	/default-rack/tp-hadoop-22:8042	logs
attempt_1759040748473_0001_m_000009_0	SUCCEEDED	Generated 1000 samples.	/default-rack/tp-hadoop-9:8042	logs

Nous avons ensuite codé nos propres scripts `mapper.py` et `reducer.py`, pour implémenter un **WordCount "à la main"**. Comme donnée d'entrée, nous avons utilisé l'archive partielle de Wikipédia en français précédemment téléchargée, afin de rechercher le mot **Télécom**.

Le job a été soumis via Hadoop Streaming, et nous avons pu suivre son exécution sur les **4 nœuds**. Le traitement s'est déroulé sur tous les noeuds, voici le résultat : `hdfs dfs -cat /user/ubuntu/output-mapper-reducer-wc/part-00000` nous donne *Télécom 1756*.

3.3 Zookeeper

3.3.1 Brève introduction

Zookeeper est un service centralisé de coordination utilisé par les systèmes distribués comme Hadoop ou HBase. Il élit un seul serveur comme **Leader**, chargé de proposer les mises à jour, tandis que les autres jouent le rôle de **Followers**. Une mise à jour n'est validée que si la majorité des serveurs (le quorum) l'accepte, ce qui permet de garantir la cohérence même en cas de panne de certains noeuds. Lorsqu'il est démarré, la commande `jps` affiche le processus `QuorumPeerMain`, indiquant qu'un serveur Zookeeper est bien en cours d'exécution.

3.3.2 Installation et configuration

Nous avons installé Zookeeper (`zookeeper-3.8.4`) dans `/opt/zookeeper`. Le fichier de configuration `/opt/zookeeper/conf/zoo.cfg` a été créé à partir du modèle fourni et adapté à notre cluster :

- `tickTime=3000`, `initLimit=30`, `syncLimit=5` : paramètres pour temporisation (on a doublé pour nous laisser le temps au noeud de se synchroniser).
- `dataDir = /opt/zookeeper/data` : répertoire de stockage des données de Zookeeper.
- `clientPort = 2181` : port d'accès par défaut.
- Définition des serveurs du quorum `tp-hadoop-22:2888:3888`, `tp-hadoop-9:2888:3888`, `tp-hadoop-30:2888:3888` et `tp-hadoop-31:2888:3888`. Dans cette configuration, le port 2888 sert à la communication entre serveurs ZooKeeper et le port 3888 est utilisé pour l'élection du leader.

Chaque serveur a reçu un identifiant unique (`myid`) placé dans `/opt/zookeeper/data/myid` (1 pour `tp-hadoop-22`, 2 pour `tp-hadoop-9`, etc.). Les services Zookeeper ont ensuite été démarrés sur chaque machine avec : `zkServer.sh start`. Pour automatiser, nous avons créé un service `systemd` (`zookeeper.service`) sur toutes les machines pour lancer automatiquement `ZooKeeper` et au démarrage et éviter les arrêts non voulus.

3.4 HBase

3.4.1 Brève introduction

HBase est une base de données NoSQL distribuée stockant les données en tables de type wide-columnar. Elle s'appuie sur Zookeeper pour la coordination et la tolérance aux pannes.

L'architecture distingue le **HMaster**, chargé de gérer les tables et l'attribution des régions, et les **RegionServers**, qui stockent et servent les données. Le HMaster est distinct de ZooKeeper : ce dernier gère le consensus/élection, tandis que le HMaster administre la base.

3.4.2 Installation et configuration

Nous avons installé HBase (hbase-2.6.3) dans /opt/hbase. La configuration repose principalement sur deux fichiers :

- /opt/hbase/conf/regionserver : liste des noeuds exécutant un **RegionServer**. Nous avons inclus tous les noeuds (9, 22, 30, 31) afin que chaque machine participe au stockage, y compris les HMaster.
- /opt/hbase/conf/hbase-site.xml : Définit les paramètres principaux du cluster :
 - hbase.cluster.distributed = true : pour activer le mode distribué
 - hbase.rootdir = hdfs://tp-hadoop-22:9000/hbase : attention, cette adresse ne désigne pas l'HMaster (qui est élu dynamiquement par ZooKeeper), mais simplement le **NameNode HDFS** utilisé pour stocker les données d'HBase.
 - hbase.zookeeper.quorum = tp-hadoop-(22,9,30,31) : liste des noeuds
 - hbase.zookeeper.property.clientPort = 2181 : port client (par défaut)
 - hbase.wal.provider = filesystem : journal par défaut (Write-Ahead-Log)

Nous avons également choisi de lancer deux HMaster (tp-hadoop-22 et tp-hadoop-9) afin de tester la tolérance aux pannes. Le démarrage des services s'effectue avec la commande `start-hbase.sh`. ZooKeeper élit automatiquement l'un des HMaster comme actif et maintient l'autre en standby, prêt à prendre le relais en cas de défaillance.

Pour automatiser, nous avons créé un service `systemd` (`hbase.service`) sur toutes les machines (avec une différence de `ExecStart` selon master/region) pour lancer automatiquement HBase et au démarrage et éviter les arrêts non voulus.

Après configuration, nous avons bien dans le shell : 1 active master, 1 backup masters, 4 servers, 0 dead, 0.5000 average load

3.5 Spark

3.5.1 Brève introduction

Spark est un moteur de calcul distribué en mémoire. Dans notre cas, il s'appuie sur **YARN** : le ResourceManager joue le rôle de **Master** et les NodeManagers ceux des **Workers**. Chaque application démarre un **Driver**, qui envoie des tâches à des **Executors** créés sur les Workers.

3.5.2 Installation et configuration

Nous avons installé Spark (`spark-3.5.6-bin-without-hadoop`) dans `/opt/spark`. Cette version *without-hadoop* a été choisie car elle s'intègre directement avec Hadoop déjà présent sur le cluster. Elle est compatible avec notre installation de Java (`openjdk-11`).

Les variables d'environnement (`SPARK_HOME`, `PATH`, `PYSPARK_PYTHON`) ont été ajoutées au `.bashrc`. La variable `SPARK_DIST_CLASSPATH` a été configurée pour pointer vers le classpath Hadoop (`$HADOOP_HOME/bin/hadoop classpath`), indispensable avec la version *without-hadoop*. Dans le fichier `/opt/spark/conf/spark-env.sh`, nous avons défini les répertoires temporaires et le lien vers Hadoop/YARN :

- `JAVA_HOME`, `HADOOP_HOME`, `HADOOP_CONF_DIR` `YARN_CONF_DIR`, `SPARK_DIST_CLASSPATH` :
Chemin pour intégration avec Java, Hadoop et YARN
- `SPARK_MASTER_HOSTS=tp-hadoop-22` : Identifiant du Master
- `SPARK_LOCAL_DIRS=/tmp/spark` : Stockage temporaire
- `SPARK_HISTORY_OPTS=-Dspark.history.fs.logDirectory=hdfs://tp-hadoop-22:9000`
`/spark-logs` : Configuration du History Server (logs stockés sur notre NameNode).

Nous n'avons pas besoin d'un `systemd` (`spark.service`) car Spark utilise directement YARN dans notre cas.

3.5.3 Usages pratiques

Nous avons validé l'installation en lançant un job PySpark en mode YARN. Pour ce test, nous avons codé un script `pyspark_count_word.py` qui compte les occurrences du mot **Jeanmougin** dans le fichier `wikipedia_fr_dump_17Go`. Le programme a été soumis via `/opt/spark/bin/spark-submit`, exécuté simultanément sur les 4 nœuds du cluster. Nous sommes **fiers de vous annoncer** que sur une bonne partie du Wikipédia français, le nom **Jeanmougin** est apparu plus de **57 fois** dans notre dump réduit (17 Go). Nous le remercions pour sa disponibilité et ses conseils.