HDFS Cluster Installation Automation for TupleWare

Xinyi Lu
Department of Computer Science
Brown University
Providence, RI 02912
xinyi_lu@brown.edu
March 26, 2014

Abstract

TupleWare[1] is a C++ Framework that provides certain functions to the user with which a parallelized algorithm is executed easily. It fits data in main memory and resides on HDFS. The Hadoop Distributed File System(HDFS)[2] is a distributed file system and the primary distributed storage used by Hadoop applications. The cluster includes one name node to manage meta data and several data nodes to store actual data. This project implements an automatic way to build HDFS on Amazon EC2 in a remote terminal using two python libraries: boto and fabric.

1. Introduction

As the amount of available data becomes tremendous and data handling changes, many companies are considering or are already using small clusters of high performance machines with low communication costs instead of big cloud computing networks for economic reasons. The amount of statistical/analytical queries is increasing that a

sequential engine cannot handle. TupleWare addresses these issues by providing an ML framework built considering the characteristics of small infiniband clusters, which makes data fits in main memory and resides on HDFS, while query is to be compiled to machine code and parallelized. The reason we choose HDFS is because it is designed to support very large files, and it's a mature open source distributed system which provides both Java and C++ libraries for file operations. Most importantly, it saves each file into blocks with typical size of 64MB, and each block will reside on a different data node if possible, which would certainly help us achieve larger parallelism according to the principle of TupleWare since it makes each data node to execute queries on its own blocks of the file locally.

HDFS is designed to run on commodity hardware. Amazon Elastic Compute Cloud (EC2)[3] is a web service that provides resizable compute capacity in the cloud. Usually when people want to create an instance on EC2, they would open a webpage, login to their account and follow the steps popping out of the window one by one. If we want to configure a cluster with several nodes, we have to open a bunch of webpages, create instances one by one, download HDFS on each instance, and then configure each node separately. It gets much more complicated when the number of nodes grows up.

To solve this problem, we have two python libraries to help. Firstly, boto.ec2[4] is a module that provides an interface to the EC2 service from AWS. It could be used to create any number of instances for the HDFS cluster and control their overall start/stop functionality in a single command. Secondly, fabric[5] is a command-line tool that provides a basic suite of operations for executing local or remote shell commands and uploading/downloading files. HDFS cluster installation and configuration could be done on the instances we just built using this library.

2. Overview

The whole process includes EC2 instances creation, HDFS download, HDFS configuration, among which HDFS download is just downloading the HDFS installation package to each instance. To make things easier, we could make our own AMI(which is an Amazon Machine Image that provides the information required to launch an instance) with downloaded HDFS and java. Then create instances with this AMI. Therefore we don't need to download HDFS and set JAVA_HOME on every instance each time we try to create a cluster. Followings are the detailed steps of the other two sections.

2.1 EC2 instances creation

Figure 2.1 Steps to create instances

2.2 HDFS configuration

We need to firstly write configuration files for each node (both the name node and data nodes):

For each node:

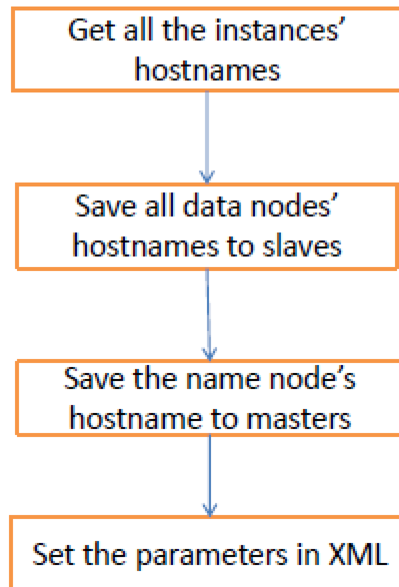

Figure 2.2.1 Steps to configure HDFS

Then set up SSH configuration so that the master node could connect to each of the data nodes:

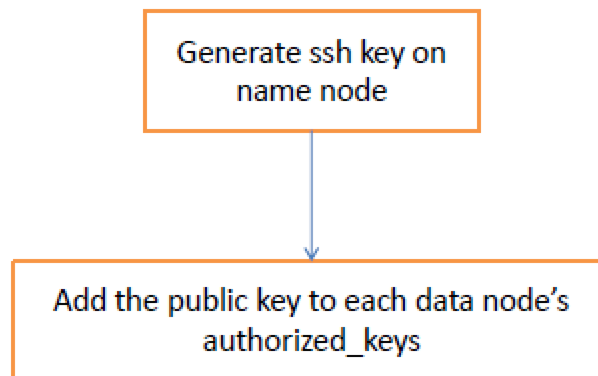

Figure 2.2.2 SSH configuration

3. Design and Implementation

This project includes two parts according to the different python libraries they are based on.

3.1 Create and configure EC2 instances using boto.ec2

Firstly, we need to connect to the AWS server and create key, security group and instances:

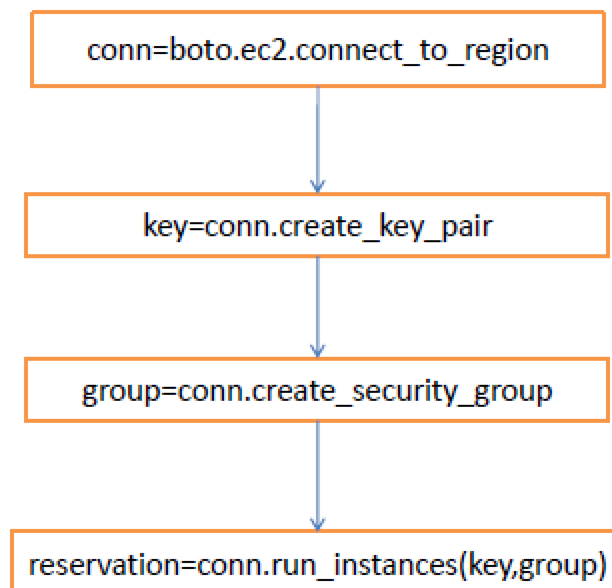

Figure 3.1.1 create instances using boto

Then we could start each instance in this reservation we just got:

```
For instance in reservation.instances:
        instance.start
        wait until   instance.status='running'
```

Figure 3.1.2 start all instances in the cluster

After the above steps, the instances have been created and they are all running. To stop or terminate all these instances in this cluster, we just need to do:

```
For instance in reservation.instances:
        instance.stop/terminate
        wait until   instance.status='stopped'/'terminated'

conn.delete_key_pair(key.name)
conn.delete_security_group(group.name)
```

Figure 3.1.3stop all the instances in the cluster

3.2 HDFS installation using fabric

Fabric provides operations for executing remote shell commands by calling run(command), while that for executing local shell commands is local(command). So if we want to install any application on a remote machine, we just need to use run() to execute the download, unzip and install command.

After the HDFS package is extracted, we should write all the config files under directory conf. Since it's not convenient to write XML nodes into a remote file, we could copy and write them locally, then replace those original file on the remote machine.

The configuration includes:

- Export JAVA_HOME to hadoop-env.sh
- Write name node's hostname to masters
- List data nodes' hostnames on slaves
- Set the name of the default file system - fs.default.name in core-site.xml so that it always points back at master
- Set the number of nodes to which data should be replicated for failover and redundancy purposes - dfs.replication in hdfs-site.xml
- Set the job tracker location on the master server in case we use MapReduce

Since python provides specific library xml.etree.ElementTree[6] for XML operations, no doubt this is the best language for this kind of configuration.

To make the name node connects to the data nodes unidirectionally, we should do SSH configuration before formatting the name node.

- Connect to name node, generate SSH key
- Copy name node's public key to local
- Add name node's public key to each data node's authorized_keys

After the above steps, the name node could ssh to all the data nodes according to their hostnames. Now we could format the name node and start the whole HDFS cluster.
Then we could use this cluster to run the applications that work in distributed file systems.

4. User's Manual

To run this program, you need to:

- Have an account at Amazon Web Service ,retrieve your Access Key ID and Secret Access Key from the web-based console
- Set the environment variable AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY temporarily or permanently.
- Install the latest released version of boto and fabric
- Compile an AMI with java installed and HDFS package downloaded
- Run createInstances.py with key name and the number of nodes you want to create in this cluster
- Run start_cluster with key name to configure and format the HDFS cluster
- Run stop.py with key name to terminate all the instances in this cluster and delete the key as well as the security group created for this reservation

```
agnes@agnes-ThinkPad-Edge-E430:~/aws$ ./createInstances.py report_test1 3
[Instance:i-44c58015, Instance:i-47c58016, Instance:i-46c58017] Reservation:r-97d9f4b9
i-44c58015started
i-47c58016started
i-46c58017started
creating instances finished
```

Figure 4.1 Create instances that form a three node cluster with key name

```
14/03/26 14:50:10 INFO blockmanagement.DatanodeManager: dfs.block.invalidate.limit=1000
14/03/26 14:50:10 INFO util.GSet: Computing capacity for map BlocksMap
14/03/26 14:50:10 INFO util.GSet: VM type       = 64-bit
14/03/26 14:50:10 INFO util.GSet: 2.0% max memory = 966.7 MB
14/03/26 14:50:10 INFO util.GSet: capacity      = 2^21 = 2097152 entries
14/03/26 14:50:10 INFO blockmanagement.BlockManager: dfs.block.access.token.enable=false
14/03/26 14:50:10 INFO blockmanagement.BlockManager: defaultReplication         = 1
14/03/26 14:50:10 INFO blockmanagement.BlockManager: maxReplication             = 512
14/03/26 14:50:10 INFO blockmanagement.BlockManager: minReplication             = 1
14/03/26 14:50:10 INFO blockmanagement.BlockManager: maxReplicationStreams      = 2
14/03/26 14:50:10 INFO blockmanagement.BlockManager: shouldCheckForEnoughRacks  = false
14/03/26 14:50:10 INFO blockmanagement.BlockManager: replicationRecheckInterval = 3000
14/03/26 14:50:10 INFO blockmanagement.BlockManager: encryptDataTransfer        = false
14/03/26 14:50:10 INFO namenode.FSNamesystem: fsOwner             = ubuntu (auth:SIMPLE)
14/03/26 14:50:10 INFO namenode.FSNamesystem: supergroup          = supergroup
14/03/26 14:50:10 INFO namenode.FSNamesystem: isPermissionEnabled = true
14/03/26 14:50:10 INFO namenode.FSNamesystem: HA Enabled: false
14/03/26 14:50:10 INFO namenode.FSNamesystem: Append Enabled: true
14/03/26 14:50:10 INFO util.GSet: Computing capacity for map INodeMap
14/03/26 14:50:10 INFO util.GSet: VM type       = 64-bit
14/03/26 14:50:10 INFO util.GSet: 1.0% max memory = 966.7 MB
14/03/26 14:50:10 INFO util.GSet: capacity      = 2^20 = 1048576 entries
```

Figure 4.2 Screen display of cluster attributes when formatting the name node

```
agnes@agnes-ThinkPad-Edge-E430:~/aws$ ./stop.py report_test1
i-44c58015 is terminated
i-47c58016 is terminated
i-46c58017 is terminated
```

Figure 4.3 Stop all the instances in this cluster given key name

5. Integration with TupleWare

After we have setup the HDFS, we could set up TupleWare on EC2 and make it run on HDFS.

- Clone the repository from Github to all the nodes in the cluster
- Make TupleWare remotely on each node
- Start TupleWare on master node and run tests to examine whether the master node could successfully dispatch jobs to the slave nodes.

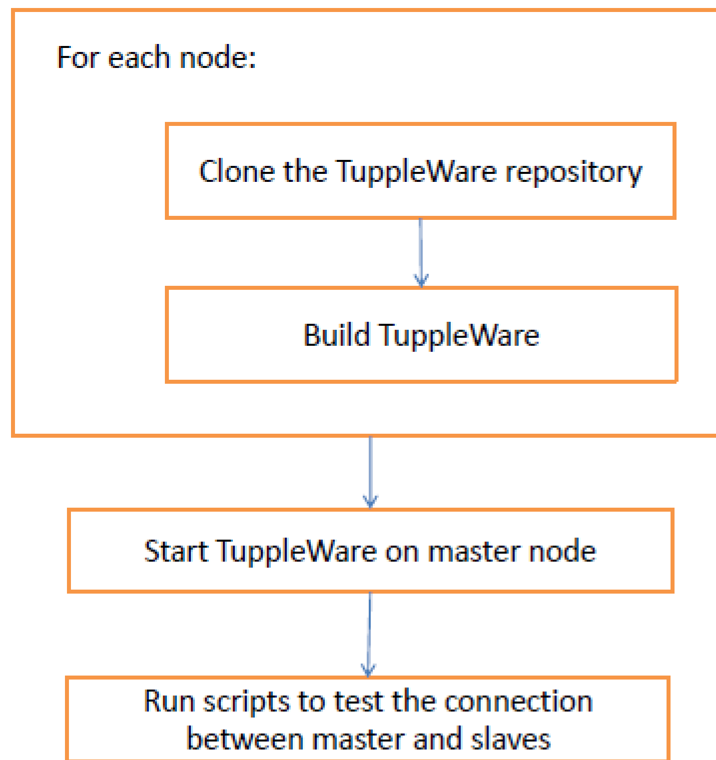

Figure 5.1 Steps to build TupleWare in cluster

6. Summary

This project implements the automation of building a HDFS cluster on Amazon EC2 for TupleWare, which includes create all the instances for the cluster, build HDFS on each node, and install TupleWare to make it work.

Python boto has a module ec2 that provides API for communicating with as well as Operating on Amazon EC2, which helps us build the underlying hardware platform for HDFS Cluster. Fabric provides remote control APIs that allows us to communicate with all the instances in a single terminal on the local machine, which enables us to configure the HDFS Cluster and run the TupleWare application above it.

Although this HDFS cluster automation program is built for TupleWare initially, it could be expanded to support any applications that built on HDFS.

7. Acknowledgements

I would like to thank Alex Galakatos, who helped me a lot by giving me advices on the python packages I could choose and the requirements of the project, and my advisor Ugur Cetintemel who gave me this chance to take part in this significant project TupleWare.

8. References

[1] TupleWare Documentation.
[2] Dhruba Borthakur. HDFS Architecture Guide
[3] Amazon Elastic Compute Cloud User Guide (API Version).
http://aws.amazon.com/documentation/ec2/
[4] Python-boto v2.27.0 Documentation. http://boto.readthedocs.org
[5] Python-fabric 1.8.2 Documentation. http://docs.fabfile.org
[6] Python XML Processing Modules Documentation.
http://docs.python.org/2/library/xml