Hadoop for Roboticists

Olivier Deiss, Cedric Pradalier

Hadoop for Roboticists

Olivier Deiss
Supervised by Prof. C. Pradalier
UMI 2958 - GeorgiaTech Lorraine
Metz, France

January 1, 2017

Contents

1 Introduction

The objective of this tutorial is to give an overview of *Hadoop*, its possibilities and its use in Robotics, as well as to provide explanations about the setup, design of mapreduce jobs and troubleshooting. *Hadoop* is a powerful software: while this tutorial is enough to get an overview and a better understanding of the framework and how to use it, we won't be covering here all the possibilities that *Hadoop* offers.

This tutorial is based on my experience while working on a Special Problem. I included all the information I have been looking for and everything I think can be useful when learning how to use *Hadoop*. Finally, while reading this tutorial, be aware that I am far from being a *Hadoop* expert and that I just started to learn about *Hadoop*. This is just a summary of the best of my knowledge at the time of writing.

After presenting *Hadoop* and how to set up a cluster, we will mainly focus on developing jobs in C++. Although jobs are usually written in Java, there are applications for which we need to be able to use C++. One of such situations is for being able to use the ROS API in our jobs. This will allow us to analyze datasets stored as *rosbag*s as one possible application.

2 About *Hadoop*

2.1 What is it?

In short, Apache™ *Hadoop*® is an open-source software framework for distributed storage and distributed processing of very large datasets on computer clusters built from commodity hardware. The true advantage of *Hadoop* is how it handles scalability and insane amounts of unstructured data. *Hadoop* works well on clusters of thousands of nodes. Here is a definition of the software from Apache's *Hadoop* page:

> It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop is supported by Hortonworks and Cloudera. Facebook and Yahoo! have been the first companies to massively use *Hadoop*; the software is now used by a majority of companies. Google's research papers [3] [2] inspired the *Hadoop* File System and the *MapReduce* implementation.

2.2 How does it work?

Before jumping into the technical part of the tutorial, this section will introduce the main components of Hadoop as well as an overview of what is a job. We will also introduce the architecture of Hadoop clusters and what processes need to run on the machines.

2.2.1 Components

Hadoop is the union of *base modules* and lots of other modules built on top of *Hadoop*. We will only focus on the *base modules* here. The base modules are the following ones:

- *Hadoop Common*: this is a core module that provides libraries to the other modules.
- *Hadoop Distributed File System (HDFS)*: HDFS is a distributed file system on which we will be storing the data we want to analyze using *Hadoop*. Alternatives to HDFS exist, but HDFS remains a good choice for the following reasons [13]: first the cost of storing data on HDFS is really low compared to other solutions that might not be open-source. HDFS is also tailored for MapReduce and allows a very high bandwidth even on very low-cost shared networks. Last, HDFS is designed to work on commodity hardware, and thus can handle failures and lost data autonomously.
- *Hadoop YARN*: YARN is used for the scheduling of the applications and the resource management.
- *Hadoop MapReduce*: *MapReduce* is an implementation of the *MapReduce* programming model. This is the framework we will use while writing jobs.

If you want to work with data, the first step is to upload it on HDFS. The next step is to write a *MapReduce* job that will be later run on the cluster. The scheduling will be taken care of by YARN.

Among the other available auxiliary projects built on top of *Hadoop*, Hive gives an SQL-like interface to run queries on the data stored on HDFS. Another one, HBase, is a distributed database based on Google's BigTable for storing large quantities of sparse data.

2.2.2 Writing jobs

Jobs are the applications that are run on the data stored on HDFS. Actually, *Hadoop* can also be run as a standalone application, without HDFS, but we won't be using *Hadoop* this way. A simple application consists of two parts: a Mapper and a Reducer. The framework reads the input data and provides pairs `<key, value>` to the Mapper. By default, *Hadoop* provides pairs with the following format: `<offset, line>`. Applications can provide their own Reader and Writers to customize these operations.

The Mapper receives these pairs and outputs another pair. This is all it does, and yet allows any kind of computation – we will see examples later. The Mapper can put whatever information is useful in these pairs. The Reducer receives the pairs from the Mapper and outputs the final pairs `<key, value>` that will be the output of the job.

Hadoop is written in Java and as a result, most of the documentation and online support is in Java. However, it is possible to write jobs in C++ using *Hadoop* Pipes. This is what we will be doing since we are roboticists who want to be able to use the *ROS* API in our jobs.

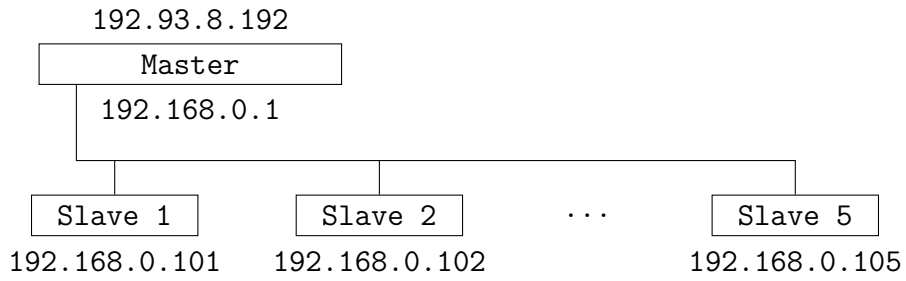
2.2.3 Cluster Architecture

Hadoop runs on clusters of – often thousands of – computers. For huge datasets that span over multiple racks, HDFS provides location awareness: the processing happens on nodes that are close to the requested data: this reduces network traffic and is one of the key advantages of using HDFS. The setup we will describe here is a much smaller one, using a cluster one master node and five slave nodes.

A typical *Hadoop* setup has the following structure:

- *Master* node: this node stores information about where is stored a specific information on the cluster. If this node becomes unavailable, the cluster as a whole becomes unavailable. For this reason, it is possible to run a second process on the same machine, which will handle the requests if the main process crashes.
- *Slave* nodes: these nodes are the workers. They store the data and execute the jobs when requested.

In this tutorial, we will use the cluster as a model the cluster I have been using which has the following structure:



Here is a list of the processes that run on the nodes. Some processes are used for HDFS while the other category is used during the *MapReduce* phase:

- *NameNode*, *JobTracker*: these processes run on the Master node. The *NameNode* is used as a registry of the data in the cluster, and the JobTracker is used during the *MapReduce* phase. A *SecondaryNameNode* can be run for the safety reasons discussed just above.
- *DataNode*, *TaskTracker*: these processes belong to the slaves. Note that it is also possible to run these processes on the Master node in small clusters. The *DataNode* handles the storage and the *TaskTracker* becomes useful during the *MapReduce* phase.

3 Setting Up a *Hadoop* Cluster

Now that we have scratched the surface of Hadoop in the previous introduction, we are ready for the first step in using Hadoop. Before writing jobs, we need to set up the structure on which we will launch these jobs. This step is about setting up a Hadoop cluster.

Hadoop can be run on one single node, and it might be useful to start from there to get familiar with the tools. The single node setup is well explained on the Apache tutorial [6]. Here, we will directly go to the setup of a cluster of multiple nodes. We will then run the example mapreduce jobs, before we can start developing jobs in C++.

We will start by installing *Hadoop* on all our machines and setting up all the tools that it needs. Then, we will configure our *Hadoop* installation.

3.1 Installing *Hadoop*

Here, we will almost be following the Apache tutorial [6]. The first step is to create a *Hadoop* user: let's call it *hadoop*, and everything *Hadoop*-related we will do will be done with this user. I have been using *Hadoop* 2.7.3 while working on this tutorial, you can get it here: http://hadoop.apache.org/releases.html. Install it wherever you find convenient, I chose */home/hadoop/bin*. We will also need to install Java if it is not already installed. *Hadoop* 2.7 and later versions require Java 7 [5].

Assuming Java and *Hadoop* are correctly installed, update your *.bashrc* as follows, in order to give *Hadoop* everything it needs to work properly. Here is what I have been using on my setup:

```
export HADOOP_HOME=/home/hadoop/bin/hadoop-2.7.3
export JAVA_HOME=/home/hadoop/bin/jdk1.8.0_102
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

Note that *HADOOP_HOME* is not required, but it will make it easier for us to use this variable to refer to where *Hadoop* is installed. In the same file, I also decided to add *bin* folders to my path. We will be using tools in *HADOOP_HOME/sbin*, which contains useful scripts, and in *HADOOP_HOME/bin*, which contains the hadoop and hdfs binaries:

```
export PATH=${HADOOP_HOME}/sbin/:${HADOOP_HOME}/bin/:${PATH}
```

We now jump to the configuration of *Hadoop*. Four files, on each machine, need to be correctly updated. There is one file for each base module. The read-only versions, with the default values, have names *base-module-default.xml*. The files that will need to be updated on the machines have the *-site.xml* suffix instead. They are listed below:

- *HADOOP_HOME/etc/hadoop/core-site.xml*: this file contains the core configuration. One important setting we will put in there is how to reach the file system.
- *HADOOP_HOME/etc/hadoop/hdfs-site.xml*: this is the configuration of the file system: the replication level, where to store the data, and the different types of nodes.
- *HADOOP_HOME/etc/hadoop/yarn-site.xml*: configuration for YARN, like how to reach the *ResourceManager*.
- *HADOOP_HOME/etc/hadoop/mapred-site.xml*: this file contains the configuration for the *MapReduce* jobs: they need to know how to reach the *JobTracker*.

The entire configuration goes into these four files. The parameters will influence the way *Hadoop* performs the tasks and you should have a look at the read-only versions of these files for a detailed explanation of the different parameters [1] [8] [14] [10].

Here I give the set of parameters that I have been using with my *Hadoop* setup. This configuration works well for simple tasks but you will likely need to tune the parameters if you plan to make extensive use of *Hadoop*.

3.1.1 Master Node

The configuration differs from master and slave nodes. We will be listing here the minimum set of parameters that need to be set on the master node's configuration files. All we need to do is put the following `<property>` tags between the `<configuration>` and `</configuration>` tags:

- *core-site.xml*: here we will be telling hadoop how to reach the *NameNode*. Since our *NameNode* is the Master node, this parameter should link to the Master node:

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://192.93.8.197:9000</value>
</property>
```

- *hdfs-site.xml*: in this file we can set the replication level and where to store the files on the machine. We choose a replication level of 3, note that this replication level should approximately be equal to the square root of the number of nodes in your cluster [10]:

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file:///home/hadoop/hadoop_data/namenode</value>
</property>
```

- *yarn-site.xml*: this is the configuration for YARN. Here, we tell YARN how to reach the *ResourceManager*, and that we want to perform *MapReduce* operations:

```
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.resourcemanager.hostname</name>
  <value>192.93.8.197</value>
</property>
```

- *mapred-site.xml*: last, we need to give the address of the *JobTracker* to *MapReduce,* and we can also set the replication level for the submitted jobs. By default, the value for `mapreduce.client.submit.file.replication` is set to 10, which will lead to under-replicated blocks in your setup if not changed.

```
<property>
  <name>mapreduce.jobtracker.address<name>
  <value>192.93.8.197:54311</value>
</property>
<property>
  <name>mapreduce.client.submit.file.replication</name>
  <value>2<value>
</property>
```

3.1.2 Slave Nodes

Now let's see the configuration of the slave nodes. Here again we need to configure the four files listed above, but the configuration is exactly the same for *core-site.xml* and *yarn-site.xml*.

- *hdfs-site.xml*: the configuration of HDFS will now require us to specify where to store the data, hence we will be using the property `dfs.datanode.data.dir` instead of `dfs.namenode.name.dir`:

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file:///home/hadoop/hadoop_data/datanode</value>
</property>
```

- *mapred-site.xml*: in this file we will be adding one more property compared to the configuration for the Master node, in order to tell the worker nodes to use YARN as their *MapReduce* framework:

```
<property>
  <name>mapreduce.jobtracker.address<name>
  <value>192.93.8.197:54311</value>
</property>
<property>
  <name>mapreduce.client.submit.file.replication</name>
  <value>2<value>
</property>
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn<value>
</property>
```

3.2 Hostnames, *SSH*

Now Hadoop should be correctly configured. But before we can start the nodes, we need to setup *ssh* on all the machines and set the hostnames of each as well. This is really important, and can be the source of lots of errors if not set right at first. *Hadoop* communicates over *ssh* between different nodes. It also uses the machines hostnames rather than their IP addresses.

Therefore we need to set up these two things. First, let's get *ssh* working. *Hadoop* requires that each node can reach any other node over *ssh*, without a passphrase [6]. If you cannot do so, running the following commands should solve the issue:

```
$ ssh-keygen -t dsa -P '' -f /home/hadoop/.ssh/id_dsa
$ cat /home/hadoop/.ssh/id_dsa.pub >> /home/hadoop/.ssh/authorized_keys
$ chmod 0600 /home/hadoop/.ssh/authorized_keys
```

If *ssh* is correctly configured, we can set the hostnames of our machines. On each machine, we need to update */etc/hosts* as well as */etc/hostname*.

3.2.1 Slave Nodes

On the Slave nodes, let's start by setting the hostname of each machine. On my setup, I have the hostnames `slave1` to `slave5`. In the */etc/hosts* file, each worker node should have access to all the other nodes. Each of my slave nodes has the following file:

```
127.0.0.1       localhost
192.168.0.101   slave1
192.168.0.102   slave2
192.168.0.103   slave3
192.168.0.104   slave4
192.168.0.105   slave5
192.168.0.1     cluster3
```

3.2.2 Master Node

On the Master node, we will need to have a list of all the available workers, in addition to correctly setting up the files */etc/hostname* and */etc/hosts*. The hostname of my master node is `cluster3`, and its *hosts* file looks like the following:

```
128.0.0.1       localhost localhost
127.0.1.1       cluster3.domain.fr cluster3
192.168.0.101   slave1 slave1
192.168.0.102   slave2 slave2
192.168.0.103   slave3 slave3
192.168.0.104   slave4 slave4
192.168.0.105   slave5 slave5
```

As stated above, we will also need to indicate what machines are available as masters or slaves in the *Hadoop* configuration. This is done in the files *masters* and *slaves* in *HADOOP_HOME/etc/hadoop*. If these files don't exist on your setup, just create them with this name.

For *HADOOP_HOME/etc/hadoop/masters*, we only have one master node so the file has just one entry:

```
cluster3
```

For *HADOOP_HOME/etc/hadoop/slaves*, I have listed the hostnames of the five available worker nodes:

```
slave1
slave2
slave3
slave4
slave5
```

3.3 Starting the Cluster

Everything should now be correctly set up and we should be able to run the cluster. We first need to format the file system:

```
$ hdfs namenode -format
```

Now that the file system is formatted, we can create our home folder where we will later upload our input files:

```
$ hdfs dfs -mkdir -p /user/hadoop
```

Note that a lot of basic commands are available on HDFS. We have just used `mkdir`, but likewise, `ls`, `cat`, `mv` are also available. We can now start HDFS, then YARN. The scripts are located in *HADOOP_HOME/sbin*:

```
HADOOP_HOME $ sbin/start-dfs.sh
HADOOP_HOME $ sbin/start-yarn.sh
```

If you have added this folder to your `PATH`, you should be able to run them without specifying the folder. Alternatively, you can run `start-all.sh`, which works well and starts both at once, even though it is deprecated. If you run `jps`, you should see this list on the master node:

```
NameNode
SecondaryNameNode
ResourceManager
```

If you run the same command on one of the slaves, you should see this list instead:

```
DataNode
NodeManager
```

Anytime, you can go to http://IP:50070 – with the IP address of the master node – to see the nodes usage in real-time. You can monitor all the applications at http://IP:8088. Useful tools are also available in command line:

```
$ hdfs dfsadmin -printTopology   # displays a topology of the setup
$ hdfs dfsadmin -report          # creates a complete report
```

3.4 *MapReduce* Example

To make sure our cluster is correctly set up, let's try to run a first *MapReduce* job. A job is provided in the *Hadoop* installation. It just counts the words in the input files and displays them with the associated count.

To run this example we first need to create our input. An input is a folder containing files we want to work on. You can just copy all the *xml* files in *HADOOP_HOME/etc/hadoop* in a folder *input*, and upload this folder on HDFS:

```
HADOOP_HOME $ mkdir input
HADOOP_HOME $ cp etc/hadoop/*.xml input
HADOOP_HOME $ hdfs dfs -put input .
```

You can make sure the folder has been correctly uploaded by running `hdfs dfs -ls`. We are now ready to launch the job:

```
$ hadoop jar \
         share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar \
         grep input output 'dfs[a-z.]+'
```

This command tells *Hadoop* to launch a jar, located in *HADOOP_HOME/share/....* In this jar, we are interested in launching the class `grep` – not the command-line grep tool – to perform the job. *Hadoop* will use *input* as its input folder and will output the results in *output*. Finally, the results will contain all the words that match the regular expression `dfs[a-z.]+`.

After a few seconds, the job should – successfully – complete. If the job failed, maybe you can find some help in the troubleshooting section at the end of this tutorial. You can also check the GUI anytime to check if everything is working properly. The output is available in *output*. We can read it from the file system, or download it and then read it:

```
$ hdfs dfs -cat output/*                          # read from HDFS
$ hdfs dfs -get output output && cat output/*     # download then read
```

If everything worked as expected, we are ready to go to the next step, where we will be using both *ROS* and *Hadoop Pipes* in a C++ mapreduce job. You can find more information about writing jobs in Java, and tutorials on the Apache website [11].

4 Writing Jobs in C++ using *Hadoop Pipes*

We now have a working installation of *Hadoop*, but we can only work with Java. Our goal is to be able to write jobs in C++, combining *ROS* and *Hadoop* in the same job. *Hadoop* provides *Hadoop Pipes*, which allows us to write jobs in C++. We will first see how to get *Hadoop Pipes* working, and then we will need to work a bit more to get *Hadoop Pipes* and *ROS* working together.

4.1 Running a *Hadoop Pipes* job

4.1.1 Compiling and Launching the Job

We will start with a simple example, following [7]. The code we are going to use in our first example is available in appendix A. We will need a Makefile in order to compile this program. This one will work, assuming you are on a 64-bit version:

```
CC             = g++
HADOOP_INSTALL = /home/hadoop/bin/hadoop-2.7.3
CPPFLAGS       = -m64 -std=c++00x -I$(HADOOP_INSTALL)/include
LIBS           = -L$(HADOOP_INSTALL)/lib -lhadooppipes -lhadoop
                 -lhadooputils -lcrypto -lpthread -lssl

word_count: word_count.cpp
  $(CC) $(CPPFLAGS) $< -o $@ -Wall $(LIBS) -g -O2
```

You might need to install additional packages like `libssl-dev` to be able to link without errors. Compile using `make word_count`. If you are getting errors, make sure you have correctly set `HADOOP_INSTALL` and that all the libraries are installed.

If you managed to compile the program, we will first need to upload it on the file system before we can run it:

```
$ hdfs dfs -put word_count .
```

Assuming you still have the input folder from the previous example, we can now run the job. If you did not previously delete the old output folder, you will get an error. To delete the old *output* folder:

```
$ hdfs dfs -rm -r -f output
```

We can now run the job:

```
$ hadoop pipes -D hadoop.pipes.java.recordreader=true \
               -D hadoop.pipes.java.recordwriter=true \
               -input input -output output \
               -program word_count
```

The first two options tell *Hadoop* to use the default Java *RecordReader* and *RecordWriter* classes. It is possible to implement these classes to get a custom behavior, but I could not find much documentation about this. There exists an example, *wordcount-nopipe.cc*, in appendix B, which shows how to do that. Be aware that this only works with local file reads though.

If everything ran successfully, we can print the output:

```
$ hdfs dfs -cat output/*
```

4.1.2 How Does This Job Work?

Let's quickly go through the code of this job and see what it does. This simple *MapReduce* job has two classes: a *Mapper* and a *Reducer*. Pretty straightforward.

Let's first focus on the *Mapper*. Its constructor takes one parameter, `context` that we are not using here. The interesting part is the `map` function, which takes a `MapContext` as its only parameter. This object contains all the pairs provided to this mapper from one partition of the input. The key and values are accessible via `getInputKey()` and `getInputValue()` respectively. For each word, the *Mapper* outputs a pair `<word, 1>`. Note that everything needs to be a string. This is all that it does.

We can now study the `Reducer`. The reduce function is called with a context that contains all the pairs with the same key. In order to count the words, we therefore need to initialize a counter to 0 and increment it for each pair in the context. This is done in the `while` loop. After what we can simply output a pair with the same key as the input key, and the value of the counter as the output value. If you wonder why we emitted ones in the *Mapper*, yes, this was absolutely useless, but we still needed to output some values.

This really is how this job works. If you want to familiarize yourself with the *MapReduce* framework, you can find lots of examples of jobs online. Even though they are in Java, it is a good start to get an intuition of how jobs should be written. You can follow the Apache tutorial on *MapReduce* [11], or this great tutorial [9].

So we managed to get *Hadoop Pipes* working and we – almost – wrote our first job. In the next section, we will see how to use *Hadoop Pipes* and *ROS* for writing our jobs.

4.2 Using *Hadoop Pipes* with *ROS*

We finally get to the section that initiated the writing of this tutorial: how to use *Hadoop* to write jobs that use ROS. *Hadoop* and Java are great for lots of domains, and jobs should be written in Java if possible. Using Pipes should only be used when there is no alternative. The fact that we want to use ROS in our jobs is one great reason to do so. The following will help you understand how to write jobs using the C++ *ROS* API. If you do not want to use *ROS*, you can skip this section.

4.2.1 Creating New Libraries

Unfortunately, the provided *Hadoop* libraries don't work well with the *ROS* libraries. This is related to the compiler Application Binary Interface (ABI). *ROS* and the *Hadoop* library don't use the same ABI, hence they work poorly together.

To get them working, you will need to recompile *Hadoop* from source. After downloading, following the steps on the Apache guide [12], run the following commands:

```
mvn package -Pdist,native -DskipTests -Dtar
```

In order to compile *Hadoop* successfully, you will need to install `protoc` – Protocol Buffers – version 2.5.0 and this version only. You can get it on Google's github [4]. The archive should be available at: https://github.com/google/protobuf/archive/v2.5.0.tar.gz.

You will get the newly-bilt library in *hadoop-dist/target/hadoop-2.7.3/lib/native*. From now on, use these libraries instead of the ones in *HADOOP_HOME/lib* and everything should work fine.

4.2.2 A Job with *ROS* and *Hadoop Pipes*

We can now go back to our initial goal of compiling a job with *Hadoop Pipes* that makes use of the *ROS* API. We will first see how to update our Makefile, them we will show how to use Catkin to do all the work for us.

We will need to update our Makefile to link the *ROS* libraries. Note that I've put the newly-built libraries in */home/hadoop/lib/hadoop/lib*.

```
CC              = g++
HADOOP_INSTALL = /home/hadoop/bin/hadoop-2.7.3
CPPFLAGS        = -m64 -std=c++00x -I$(HADOOP_INSTALL)/include
                  -I/opt/ros/kinetic/include
LIBS            = -L/home/hadoop/lib/hadoop/lib -lhadooppipes -lhadoop
                  -lhadooputils -lcrypto -lpthread -lssl
                  -L/opt/ros/kinetic/lib -lroscpp -lrosconsole -lrostime
                  -lrosbag -lrosbag_storage

ros_example: ros_example.cpp
  $(CC) $(CPPFLAGS) $< -o $@ -Wall $(LIBS) -g -O2
```

This new Makefile indicates where to find the right *libraries*. This is all we needed to add. Using this, you should be able to compile jobs in C++, using *Hadoop* and *ROS* libraries.

4.2.3 Using *Hadoop*, *ROS* and *Catkin*

For modern versions of ROS, compilation is made using *Catkin*. In order to keep using it to compile our jobs, we just need to create a package with the right dependencies and

CMakeLists.txt file. Let's follow the procedure from the beginning: first we need to create a new package:

```
$ catkin_create_pkg ros_example
```

We then need to edit the file *package.xml*. In the dependencies section, add `roscpp` to specify that we will be using C++, as well as all the other needed dependencies for your project:

```
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rostime</build_depend>
<build_depend>rosbag</build_depend>
<build_depend>rosbag_storage</build_depend>
<build_depend>rosconsole</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rostime</run_depend>
<run_depend>rosbag</run_depend>
<run_depend>rosbag_storage</run_depend>
<run_depend>rosconsole</run_depend>
<run_depend>std_msgs</run_depend>
```

More important, we need to edit the *CMakeLists.txt* file to include *Pipes* and link *Hadoop* as well as *ROS*:

```
cmake_minimum_required(VERSION 2.8.3)
project(example)
find_package(catkin REQUIRED COMPONENTS
  roscpp rosconsole rostime rosbag rosbag_storage
)
catkin_package(
  CATKIN_DEPENDS roscpp
)
include_directories(~/bin/hadoop-2.7.3/include ${catkin_INCLUDE_DIRS})
add_definitions(-g -ggdb -std=c++14 -Wall -O3)
link_directories(/home/hadoop/lib/hadoop/lib)
add_executable(example ros_example.cpp)
target_link_libraries(example
  hadooppipes hadoop hadooputils crypto pthread ssl
)
target_link_libraries(example ${catkin_LIBRARIES})
install(TARGETS example
  ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

There is nothing really special about this file. It is a standard *CMakeLists.txt* file with instructions to include and link everything required by *Hadoop*.

With this file, we should be able to compile our code using `catkin_make`. The binary will be in the directory `build` of your *Catkin* workspace.

4.3 An Example of Job to Analyze rosbags

To complete our introduction on using ROS with *Hadoop* we are going to write a job that analyzes rosbags and counts the different values for a given topic in the file.

The structure is very similar to that of the first job we have studied. We will have one class for the *Mapper*, one for the *Reducer*. This does not change. What changes is that rosbags cannot be read directly by *Hadoop*. While looking for a more elegant solution, we are going to store our rosbags on a shared folder that will be accessed by the nodes through NFS. The job's input will instead be a folder containing the paths of these files. Therefore each entry of the *Map* task will be the path of a file that we can open using the *ROS* API.

Once the file is open, we need to create a view of this file for the topic of our choice. The view can provide us with iterators through the file. As for the word count, we now just need to iterate through the values and output them as the key, with a value that will not be used. After iterating through the values, we close the bag and return.

Now for the *Reducer*, we read all the values, and increment a counter for every value. After iterating through all the keys, we output the key and its counter.

The code for this example is available in appendix C. You can compile it with the command `catkin_make`, upload the program on HDFS and run it with an input being a folder of files containing paths to your actual rosbags

5 Troubleshooting

Before even being able to write this simple tutorial, I have come a long way and had to face and solve lots of issues. I tried to list them all here with ways to solve them. They were mostly due to configuration problems so you might not have as many errors as I did. Otherwise this section is the right one.

5.1 Starting *Hadoop*

All the processes did not start

After running `start-all.sh` or `start-dfs.sh` and `start-yarn.sh`, you should always check with `jps` if all the processes started successfully. If they did not, you will find why in the logs. Most likely, you've been trying to restart them too quickly and the ports are reserved. You can check what process is using your port with:

```
sudo netstat -ap | grep :50090
```

Then kill this process and try again.

Hadoop does not work after a restart

Make sure you have correctly set the folders for *Hadoop* in *core-site.xml* on all the machines. By default, *Hadoop* writes in */tmp*.

5.2 MapReduce

Let's start with issues when launching a job. First you should go have a look in the log files. They are in *HADOOP_HOME/logs/user* and will always tell you the source of the error. Alternatively, you can check if everything is ok from a configuration aspect with:

```
$ hdfs dfsadmin -report
```

Running a Job Fails

Hadoop will not overwrite an output folder. You should always make sure to delete the previous output folders or to specify a different folder for later jobs. If *Hadoop* is trying to reach a node and is having trouble, check if all the nodes are up. Sometimes some nodes are *decommissioned* and you need to start the right processes again. Running `start-all.sh` from the master should launch all the missing nodes.

Once again, checking the logs will most likely tell you the source of the error. Another error related to the use of ROS is that some nodes could be missing libraries required by your job. You need to make sure all your nodes are correctly configured before going on.

The job hangs at *map 0%, reduce 0%*

Make sure you have set the hostnames right, and that you can *ssh* to any node without a passphrase. In any case, the logs should tell you what happened, but this issue happened to me due to network issues. This can also be due to an error in your code that causes an exception. Typically, if your *Mapper* is correct but your *Reducer* is wrong, you will most likely see the task hang at *map 100%, reduce 0%*.

5.3 Monitoring

Positive amount of under-replicated blocks

You need to check if your settings for replication are correctly set. Also, make sure `mapreduce.client.submit.file.replication` is set, otherwise its default value of 10 will lead to under-replicated blocks if your cluster has less than ten machines.

5.4 Writing Jobs

Finally, while writing jobs, make sure to never cut a stream, meaning that a *Mapper* or *Reducer* should only return once it has extracted all the pairs that it was given. Also, make sure your stream has pairs available to be read before extracting one.

6 Conclusion

In this tutorial, we have seen how to set up a cluster for *Hadoop*. We have also studied how to compile jobs for *Hadoop* in C++, using *Hadoop Pipes*, and how to use *ROS* for our jobs.

Regarding performances, during my work I have managed to interpolate three days of data in 40 minutes, using the cluster described above in the tutorial. However the way I have been using *Hadoop*, through shared files on NFS, would not scale properly to larger structures where we could not benefit of the rack-awareness. Instead, with a preprocessing step to extract the data in the *rosbags* in *csv* files, we should be able to use Hadoop the way it is supposed to be used.

I hope you managed to set up your own cluster and that this document has been helpful. Now that your *Hadoop* is working, it's time to write jobs!

A WordCount.cpp

```
#include <algorithm>
#include <limits>
#include <string>

#include "stdint.h"

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"

class WordCountMapper : public HadoopPipes::Mapper {
  public:
  WordCountMapper(HadoopPipes::TaskContext& context) {}
  // map function: receives a line, outputs (word,"1") to reducer.
  void map(HadoopPipes::MapContext& context) {
    // get line of text
    std::string line = context.getInputValue();
    // split it into words
    std::vector<string> words = HadoopUtils::splitString(line, " ");
    // emit each word tuple (word, "1")
    for (unsigned int i=0 ; i < words.size() ; i++) {
      context.emit(words[i], HadoopUtils::toString(1));
    }
  }
};

class WordCountReducer : public HadoopPipes::Reducer {
  public:
  WordCountReducer(HadoopPipes::TaskContext& context) {}
  void reduce(HadoopPipes::ReduceContext& context) {
    int count = 0;
    // get all tuples with the same key, and count their numbers
    while (context.nextValue()) {
      count += HadoopUtils::toInt(context.getInputValue());
    }
    // emit (word, count)
    context.emit(context.getInputKey(), HadoopUtils::toString(count));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(
    HadoopPipes::TemplateFactory<WordCountMapper,
                                 WordCountReducer>()
  );
}
```

B WordCount-NoPipe.cc

```
/**
  * Licensed to the Apache Software Foundation (ASF) under one
  * or more contributor license agreements.  See the NOTICE file
  * distributed with this work for additional information
  * regarding copyright ownership.  The ASF licenses this file
  * to you under the Apache License, Version 2.0 (the
  * "License"); you may not use this file except in compliance
  * with the License.  You may obtain a copy of the License at
  *
  *     http://www.apache.org/licenses/LICENSE-2.0
  *
  * Unless required by applicable law or agreed to in writing, software
  * distributed under the License is distributed on an "AS IS" BASIS,
  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
  * implied. See the License for the specific language governing
  * permissions and limitations under the License.
  */

#include "hadoop/Pipes.hh"
#include "hadoop/TemplateFactory.hh"
#include "hadoop/StringUtils.hh"
#include "hadoop/SerialUtils.hh"

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

const std::string WORDCOUNT = "WORDCOUNT";
const std::string INPUT_WORDS = "INPUT_WORDS";
const std::string OUTPUT_WORDS = "OUTPUT_WORDS";

class WordCountMap: public HadoopPipes::Mapper {
  public:
  HadoopPipes::TaskContext::Counter* inputWords;

  WordCountMap(HadoopPipes::TaskContext& context) {
    inputWords = context.getCounter(WORDCOUNT, INPUT_WORDS);
  }

  void map(HadoopPipes::MapContext& context) {
    std::vector<std::string> words =
      HadoopUtils::splitString(context.getInputValue(), " ");
    for (unsigned int i=0 ; i < words.size() ; i++) {
      context.emit(words[i], "1");
    }
    context.incrementCounter(inputWords, words.size());
  }
};

class WordCountReduce: public HadoopPipes::Reducer {
  public:
  HadoopPipes::TaskContext::Counter* outputWords;
```

```
  WordCountReduce(HadoopPipes::TaskContext& context) {
    outputWords = context.getCounter(WORDCOUNT, OUTPUT_WORDS);
  }

  void reduce(HadoopPipes::ReduceContext& context) {
    int sum = 0;
    while (context.nextValue()) {
      sum += HadoopUtils::toInt(context.getInputValue());
    }
    context.emit(context.getInputKey(), HadoopUtils::toString(sum));
    context.incrementCounter(outputWords, 1);
  }
};

class WordCountReader: public HadoopPipes::RecordReader {
  private:
  int64_t bytesTotal;
  int64_t bytesRead;
  FILE* file;
  public:
  WordCountReader(HadoopPipes::MapContext& context) {
    std::string filename;
    HadoopUtils::StringInStream stream(context.getInputSplit());
    HadoopUtils::deserializeString(filename, stream);
    struct stat statResult;
    stat(filename.c_str(), &statResult);
    bytesTotal = statResult.st_size;
    bytesRead = 0;
    file = fopen(filename.c_str(), "rt");
    HADOOP_ASSERT(file != NULL, "failed to open " + filename);
  }

  ~WordCountReader() {
    fclose(file);
  }

  virtual bool next(std::string& key, std::string& value) {
    key = HadoopUtils::toString(ftell(file));
    int ch = getc(file);
    bytesRead += 1;
    value.clear();
    while (ch != -1 && ch != '\n') {
      value += ch;
      ch = getc(file);
      bytesRead += 1;
    }
    return ch != -1;
  }

  /**
    * The progress of the record reader through the split
    * as a value between 0.0 and 1.0.
    */
  virtual float getProgress() {
    if (bytesTotal > 0) {
```

```
      return (float)bytesRead / bytesTotal;
    } else {
      return 1.0f;
    }
  }
};

class WordCountWriter: public HadoopPipes::RecordWriter {
  private:
  FILE* file;
  public:
  WordCountWriter(HadoopPipes::ReduceContext& context) {
    const HadoopPipes::JobConf* job = context.getJobConf();
    int part = job->getInt("mapred.task.partition");
    std::string outDir = job->get("mapred.work.output.dir");
    // remove the file: schema substring
    std::string::size_type posn = outDir.find(":");
    HADOOP_ASSERT(posn != std::string::npos,
                  "no schema found in output dir: " + outDir);
    outDir.erase(0, posn+1);
    mkdir(outDir.c_str(), 0777);
    std::string outFile = outDir + "/part-" +
                          HadoopUtils::toString(part);
    file = fopen(outFile.c_str(), "wt");
    HADOOP_ASSERT(file != NULL, "can't open file: " + outFile);
  }

  ~WordCountWriter() {
    fclose(file);
  }

  void emit(const std::string& key, const std::string& value) {
    fprintf(file, "%s -> %s\n", key.c_str(), value.c_str());
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(
    HadoopPipes::TemplateFactory<WordCountMap, WordCountReduce,
                                 void, void, WordCountReader,
                                 WordCountWriter>()
  );
}
```

C ros_example.cpp

```
#include <vector>

#include "Pipes.hh"
#include "StringUtils.hh"
#include "SerialUtils.hh"

#include "ros/ros.h"
#include "rosbag/view.h"
#include "rosbag/structures.h"
#include "geometry_msgs/Twist.h"

class ExampleMap : public HadoopPipes::Mapper {
 public:
  ExampleMap(HadoopPipes::TaskContext& context) {}
  void map(HadoopPipes::MapContext& context) {
    // extract paths from input
    std::vector<std::string> paths =
        HadoopUtils::splitString(context.getInputValue(), " ");
    for (unsigned int i=0; i < paths.size(); ++i) {
      // open rosbag file, create a view and an iterator on this view
      std::string path = "/mnt/nfs/" + paths[i];
      rosbag::Bag bag(path);
      rosbag::View topic_view(bag, rosbag::TopicQuery("/cmd_vel"));
      rosbag::View::iterator j = topic_view.begin();
      // iterate through the values
      for (; j != topic_view.end(); j++) {
        // instantiate a message from the view
        boost::shared_ptr<geometry_msgs::Twist> m =
            j->instantiate<geometry_msgs::Twist>();
        // emit linear.x as a key, with random value
        context.emit("angular x " + std::to_string(m->angular.x), "1");
      }
      bag.close();
    }
  }
};

class ExampleReduce : public HadoopPipes::Reducer {
 public:
  ExampleReduce(HadoopPipes::TaskContext& context) {}
  void reduce(HadoopPipes::ReduceContext& context) {
    int count = 0;
    while (context.nextValue()) count++;
    // output the final result
    context.emit(context.getInputKey(), std::to_string(count));
  }
};

int main(int argc, char *argv[]) {
  return HadoopPipes::runTask(
    HadoopPipes::TemplateFactory<ExampleMap, ExampleReduce>()
  );
}
```

References

[1] *core-default.xml*. URL: https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-common/core-default.xml.

[2] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. 2004. URL: http://static.googleusercontent.com/media/research.google.com/fr//archive/mapreduce-osdi04.pdf.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google File System*. 2003. URL: http://static.googleusercontent.com/media/research.google.com/fr//archive/gfs-sosp2003.pdf.

[4] *Google – protobuf*. URL: https://github.com/google/protobuf.

[5] *Hadoop Java Versions*. Apache. URL: https://wiki.apache.org/hadoop/HadoopJavaVersions.

[6] *Hadoop: Setting up a Single Node Cluster*. Apache. URL: http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-common/SingleCluster.html.

[7] *Hadoop Tutorial 2.2 – Running C++ Programs on Hadoop*. Smith College. URL: cs.smith.edu/dtfwiki/index.php/Hadoop_Tutorial_2.2_-_Running_C++_Programs_on_Hadoop.

[8] *hdfs-default.xml*. URL: http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[9] Andrea Iacono. *MapReduce by Examples*. SlideShare. URL: http://www.slideshare.net/andreaiacono/mapreduce-34478449.

[10] *mapred-default.xml*. URL: https://hadoop.apache.org/docs/r2.7.3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[11] *MapReduce Tutorial*. Apache. URL: http://hadoop.apache.org/docs/r2.7.3/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[12] *Native Libraries Guide*. Apache. URL: https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-common/NativeLibraries.html#Build.

[13] *Thinking about the HDFS vs. other Storage Technologies*. Apache. URL: http://hortonworks.com/blog/thinking-about-the-hdfs-vs-other-storage-technologies/.

[14] *yarn-default.xml*. URL: https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.