MODULE 5

Data Analytics for IoT: Introduction, Apache Hadoop, Using Hadoop MapReduce for Batch Data Analytics, Apache Oozie, Apache Spark, Apache Storm, Using Apache Storm for Real-time Data Analysis.

5.1 Introduction

The volume, velocity and variety of data generated by data-intensive IoT systems is so large that it is difficult to store, manage, process and analyze the data using traditional databases and data processing tools. Analysis of data can be done with aggregation methods (such as computing mean, maximum, minimum, counts, etc.) or using machine learning methods such as clustering and classification. Clustering is used to group data together based on similarity criteria, and data which are more similar to each other (with respect to some similarity criterion) than other data items are put in one cluster. Classification is used for categorizing objects into predefined categories.

In this chapter, various frameworks for data analysis including Apache Hadoop, Apache Oozie, Apache Spark and Apache Storm will be discussed. Case studies on batch and real-time data analysis for a forest fire detection system are described. Before going into the applications and analytics tools, let us look at the IoT system and the application being discussed.

Figure 5.1 shows the deployment design of a forest fire detection system with multiple end nodes which are deployed in a forest. The end nodes are equipped with sensors for measuring temperature, humidity, light and carbon monoxide (CO) at various locations in the forest. Each end node sends data independently to the cloud using REST-based communication. The data collected in the cloud is analyzed before any notification has been triggered.

The figure shows the message flow of data collected for forest fire detection. Each node may have above three thousand readings of (temperature, humidity, light and CO sensors). By analyzing those time-series readings in real-time or near-real-time, we can get to know how the situation is changing in the forest. These readings enable the monitoring of environmental metrics (minute, hourly, daily or monthly) to determine the mean, maximum and minimum readings of data.

Figure 5.2 shows a Raspberry Pi-based end node used for this application. The end node designed on a Raspberry Pi device and uses DHT22 temperature and humidity sensor, photoresistor sensor and MQ135 CO sensor. Box 10.1 shows the Python code for the light and temperature measurement function on the node. This example uses the Xively Python API to control both the sensors on the end nodes. This example uses the Xively *push* for REST. The readController() function now Xively datastreams are created for temperature, humidity, light and CO data. The runController() function is called every second and the sensor readings are obtained. The Xively REST API is used for sending data to the Xively cloud.

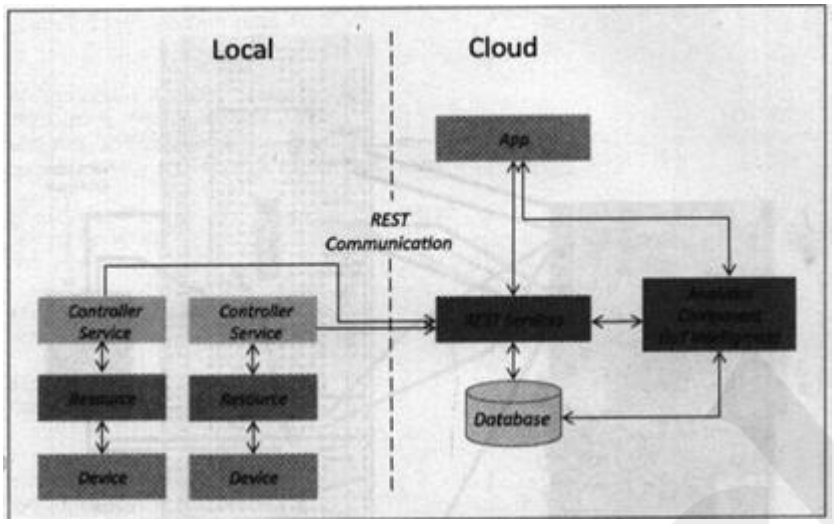

Fig 5.1 Deployement design of forest fire detection system

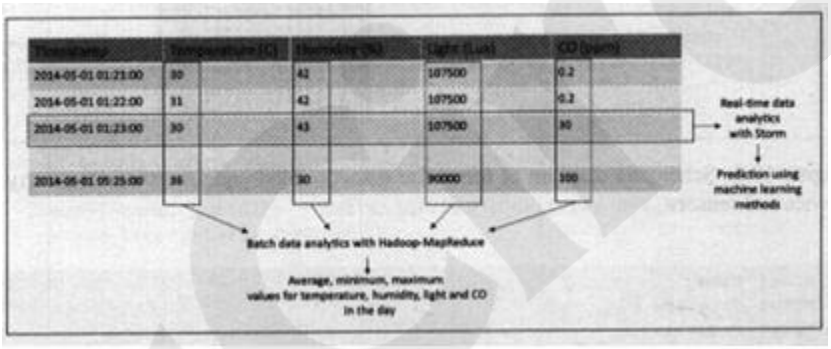

Fig 5.2 Data analysis for forest fire detection

Box 5.1 : Controller service for forest fire detection-script

```
import time
import random

# Thresholds (can be adjusted based on sensor calibration)
TEMPERATURE_THRESHOLD = 50.0  # in degrees Celsius
SMOKE_THRESHOLD = 300  # arbitrary units (depends on sensor)

def read_temperature():
    # Replace this with actual sensor reading code
    return round(random.uniform(30.0, 60.0), 2)

def read_smoke_level():
    # Replace this with actual sensor reading code
    return random.randint(100, 500)
```

```
def trigger_alert(temp, smoke):
    print("🔥 ALERT: Possible forest fire detected!")
    print(f"Temperature: {temp} °C, Smoke Level: {smoke}")
    # Add GPIO pin control or notification system here
    # For example: turn on buzzer, send SMS/email, etc.

def controller_service():
    print("🌲 Forest Fire Detection System Started")
    while True:
        temperature = read_temperature()
        smoke_level = read_smoke_level()

        print(f"Temperature: {temperature} °C, Smoke: {smoke_level}")

        if temperature >= TEMPERATURE_THRESHOLD and smoke_level >= SMOKE_THRESHOLD:
            trigger_alert(temperature, smoke_level)

        time.sleep(5)  # delay between readings

if __name__ == "__main__":
    try:
        controller_service()
    except KeyboardInterrupt:
        print("\n🛑 System stopped manually.")
```

5.2 Apache Hadoop

Apache Hadoop is an open source framework for distributed batch processing of big data. MapReduce is parallel programming model suitable for analysis of big data. MapReduce algorithms allow large scale computations to be parallelized across a large cluster of servers.

5.2.1 MapReduce Programming Model

MapReduce is a widely used parallel data processing model for processing and analysis of massive scale data. MapReduce model has two phases: **Map** and **Reduce**. MapReduce programs are written in a functional programming style to create Map and Reduce functions. The input data to the map and reduce phases is in the form of key-value pairs. Runtime systems for MapReduce are typically large clusters built of commodity hardware. The MapReduce run-time systems take care of tasks such partitioning the data, scheduling jobs and communication between nodes in the cluster. This makes it easier for programmers to analyze massive scale data without worrying about tasks such as data partitioning and scheduling.

Figure 5.4 shows the flow of data for a MapReduce job. MapReduce programs consist of input key-value pairs and produce a set of output key-value pairs. In the Map phase, as data is read from a

distributed file system, partitioned among a set of computing nodes in the cluster, and transformed as a set of key-value pairs. The Map task processes the input records independently of each other and produce intermediate results as key-value pairs. The intermediate results are stored on the local disk of the nodes where the task runs. When all the Map tasks are completed, the Reduce phase begins in which the intermediate data is sent as key is aggregated. An optional Combine task can be used to perform data aggregation on the intermediate data of the same key for the output of the mapper before transferring output to the Reduce task. MapReduce programs take advantage of locality of data and the data processing takes place on the nodes where the data resides. In traditional approaches for data analysis.

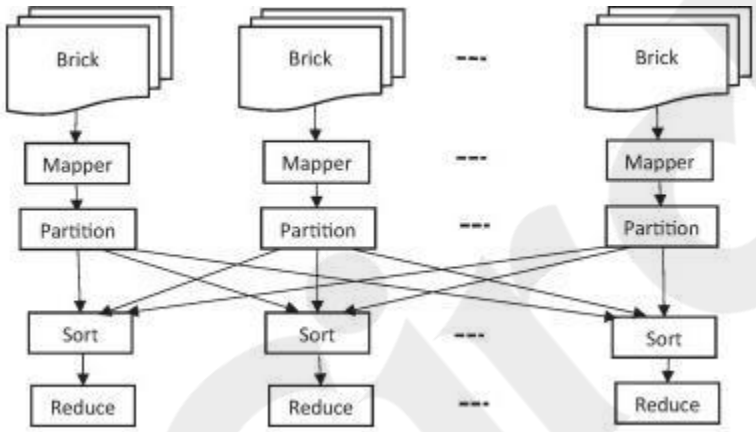

Fig 5.4: Data flow in Mapreduce

5.2.2 Hadoop MapReduce Job Execution

In this section you will learn about the MapReduce job execution workflow and the steps involved in job submission, job initialization, task selection and task execution. Figure 10.5 shows the components of a Hadoop cluster. A Hadoop cluster comprises of a Master node, backup node and a number of slave nodes. The master node runs the NameNode and JobTracker processes and the slave nodes run the DataNode and TaskTracker components of Hadoop. The backup node runs the Secondary NameNode process. The functions of the key processes of Hadoop are described as follows:

NameNode NameNode keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It does not store the data of these files itself. Client applications talk to the NameNode whenever they wish to locate a file.

Secondary NameNode HDFS is not currently a high availability system. The NameNode is a Single Point of Failure for the HDFS Cluster. When the NameNode goes down, the file system goes offline. An optional Secondary NameNode which is hosted on a separate machine creates checkpoints of the namespace.

JobTracker The JobTracker is the service within Hadoop that distributes MapReduce tasks to specific

nodes in the cluster, ideally the nodes that have the data, or at least are in the same rack.

TaskTracker TaskTracker is a node in a Hadoop cluster that accepts Map, Reduce and Shuffle tasks from the JobTracker. Each TaskTracker has a defined number of slots which indicate the number of tasks that it can accept. When the JobTracker tries to find a TaskTracker to schedule a map or reduce task it first looks for an empty slot on the same node that hosts the DataNode containing the data. If an empty slot is not found on the same node, the JobTracker looks for an empty slot on a node in the same rack.

DataNode A DataNode stores data in an HDFS file system. A functional HDFS filesystem has more than one DataNode, with data replicated across them. DataNodes connect to the NameNode on startup. DataNodes respond to requests from the NameNode for filesystem operations. Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data. Similarly, MapReduce operations assigned to TaskTracker instances near a DataNode, talk directly to the DataNode to access the files. TaskTracker instances can be deployed on the same servers that host DataNode instances, so that MapReduce operations are performed close to the data.

5.2.3 MapReduce Job Execution Workflow Figure 5.6 shows the MapReduce job execution workflow for Hadoop MapReduce framework. The job execution starts when the client applications submit jobs to the Job tracker. The JobTracker returns a JobID to the client application. The JobTracker talks to the NameNode to determine the location of the data. The JobTracker locates TaskTracker nodes with available slots at/or near the data. The TaskTrackers send out heartbeat messages to the JobTracker, usually every few minutes, to reassure the JobTracker that they are still alive. These messages also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster, new work can be delegated. The JobTracker submits the work to the TaskTracker nodes when they poll for tasks. To choose a task for a TaskTracker, the JobTracker uses various scheduling algorithms. The default scheduling algorithm in Hadoop is FIFO (first-in, first-out). In FIFO scheduling a work queue is maintained and JobTracker pulls the oldest job first for scheduling. There is no notion of the job priority or size of the job in FIFO scheduling. The TaskTracker nodes are monitored using the heartbeat signals that are sent by the TaskTrackers to JobTracker. The TaskTracker spawns a separate JVM process for each task so that any task failure does not bring down the TaskTracker. The TaskTracker monitors these spawned processes while capturing the output and exit codes. When the process inishes, successfully or not, the TaskTracker notifies the JobTracker. When a task fails the TaskTracker notifies the JobTracker and the JobTracker decides whether to resubmit the job to some other TaskTracker or mark that specific record as something to avoid. The JobTracker can blacklist a TaskTracker as unreliable if there are repeated task failures. When the job is completed, the JobTracker updates its status. Client applications can poll the JobTracker for status of the jobs.

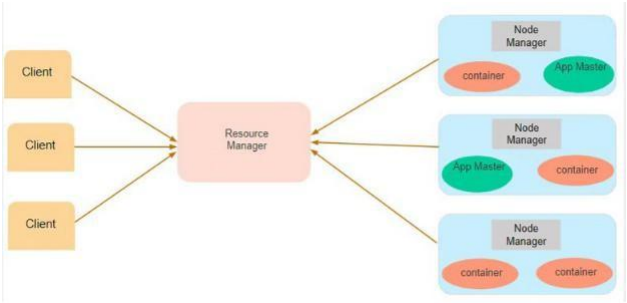

Fig 5.5: Components of Hadoop Cluster

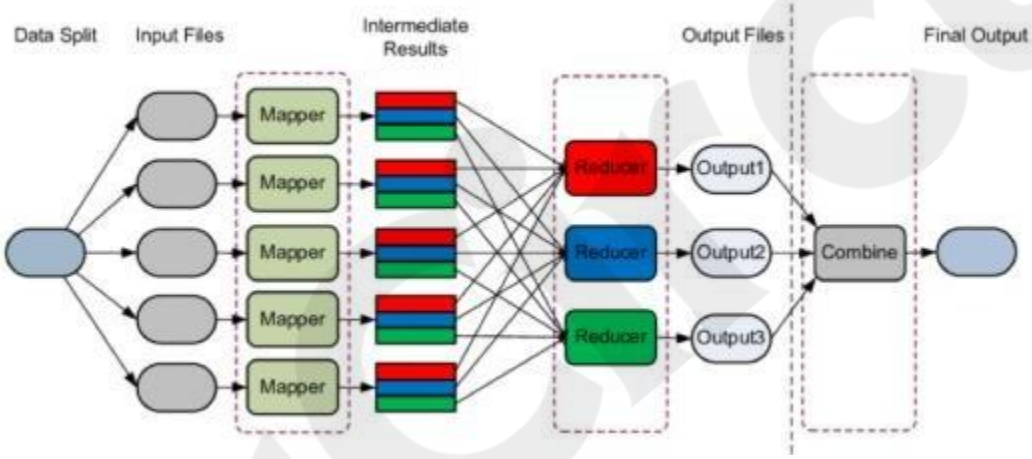

Fig 5.6: Hadoop MapReduce job execution

5.2.4 Hadoop Cluster Setup In this section you will learn how to setup a Hadoop cluster. The Hadoop open source framework is written in Java and has been designed to work with commodity hardware. The Hadoop filesystem HDFS is highly fault-tolerant. While the preferred operating system to host Hadoop is Linux, it can also be set up on Windows-like operating systems with a Cygwin environment.

A multi-node Hadoop cluster configuration will be described in this section comprising of one master node that runs the NameNode and JobTracker and two slave nodes that run the TaskTracker and DataNode. The hardware used for the Hadoop cluster described in this section consists of three Amazon EC2 (*m1.Large*) instances running Ubuntu Linux.

The steps involved in setting up a Hadoop cluster are described as follows:

Install Java Hadoop requires Java 6 or later version. Box 5.2 lists the commands for installing Java 7.

Install Hadoop To setup a Hadoop cluster, the Hadoop setup tarball is downloaded and unpacked on all the nodes. The Hadoop version used for the cluster example in this section is 1.0.4. Box 5.3 lists the commands for installing Hadoop.

Box 5.2 Commands for Installing Java

```
# Set the properties
sudo apt-get -q -y install python-software-properties
sudo add-apt-repository -y ppa:webupd8team/java
sudo apt-get -q -y update

# State that you accepted the license
echo debconf shared/accepted-oracle-license-v1-1 select true | sudo debconf-set-
selections
echo debconf shared/accepted-oracle-license-v1-1 seen true | sudo debconf-set-
selections

# Install Oracle Java 7
sudo apt-get -q -y install oracle-java7-installer

# Update environment variable
sudo bash -c 'echo JAVA_HOME=/usr/lib/jvm/java-7-oracle/ >> /etc/environment'
```

Box 5.3 Commands for Installing and configuring Hadoop

```
$wget    http://apache.techartifact.com/mirror/hadoop/common/hadoop-1.0.4/hadoop-
1.0.4.tar.gz
$tar xzf hadoop-1.0.4.tar.gz
#Change hostname of node
$sudo hostname master
$sudo hostname slave1
$sudo hostname slave2

#Modify /etc/hosts file and add private IPs of Master and Slave nodes:
$sudo vim /etc/hosts
<private_IP_master> master
<private_IP_slave1> slave1
<private_IP_slave2> slave2

$ssh-keygen -t rsa -f ~/.ssh/id_rsa
$sudo cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
#Open authorized keys file and copy authorized keys of each node $sudo vim ~/.ssh/authorized_keys
 #Save host key fingerprints by connecting to every node using SSH
$ssh master
 $ssh slave1
$ssh slave2
```

Table 5.1:Hadoop Configuration Files

File Name	Description
core-site.xml	Configuration parameters for Hadoop core which are common to MapReduce and HDFS

File Name	Description
mapred-site.xml	Configuration parameters for MapReduce daemons - JobTracker and TaskTracker
hdfs-site.xml	Configuration parameters for HDFS daemons - NameNode and Secondary NameNode and DataNode
hadoop-env.sh	Environment variables for Hadoop daemons
masters	List of nodes that run a Secondary NameNode
slaves	List of nodes that run TaskTracker and DataNode
log4j.properties	Logging properties for the Hadoop daemons
mapred-queue-acls.xml	Access control lists

Networking After unpacking the Hadoop setup package on all the nodes of the cluster, the next step is to configure the network such that all the nodes can connect to each other over the network. To make the addressing of nodes simple, assign simple host names to nodes (such master, slave1 and slave2). The /etc/hosts file is edited on all nodes and IP addresses and host names of all the nodes are added.

Hadoop control scripts use SSH for cluster-wide operations such as starting and stopping NameNode, DataNode, JobTracker, TaskTracker and other daemons on the nodes in the cluster. For the control scripts to work, all the nodes in the cluster must be able to connect to each other via a password-less SSH login. To enable this, public/private RSA key pair is generated on each node. The private key is stored in the file ~/.ssh/id_rsa and public key is stored in the file ~/.ssh/id_rsa.pub. The public SSH key of each node is copied to the ~/.ssh/authorized_keys file of every other node. This can be done by manually editing the ~/.ssh/authorized_keys file on each node or using the ssh-copy-id command. The final step to setup the networking is to save host key fingerprints of each node to the known_hosts file of every other node. This is done by connecting from each node to every other node by SSH.

Configure Hadoop With the Hadoop setup package unpacked on all nodes and networking of nodes setup, the next step is to configure the Hadoop cluster. Hadoop is configured using a number of configuration files listed in Table 5.1. Boxes 5.4, 5.5, 5.6 and 5.7 show the sample configuration settings for the Hadoop configuration files core-site.xml, mapred-site.xml, hdfs-site.xml, masters/slaves files respectively.

■ Box 5.4: Sample configuration - core-site.xml

XML
```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://master:54310</value>
</property>
</configuration>
```

■ Box 5.5: Sample configuration hdfs-site.xml

XML
```
<?xml version="1.0"?>
<configuration>
<property>
<name>dfs.replication</name>
<value>2</value>
</property>
</configuration>
```

■ Box 5.6: Sample configuration mapred-site.xml

XML
```
<?xml version="1.0"?>
<configuration>
<property>
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
</configuration>
```

■ Box 5.7: Sample configuration masters and slave files

```
$cd hadoop/conf/
#Open the masters file and add hostname of master node
$vim masters
#master

#Open the masters file and add hostname of slave nodes
$vim slaves
#slave1
#slave2
```

■ Box 5.8: Starting and stopping Hadoop cluster

```
$cd hadoop-1.0.4
#Format NameNode
$bin/hadoop namenode -format

#Start HDFS daemons
$bin/start-dfs.sh

#Start MapReduce daemons
```

```
$bin/start-mapred.sh

#Check status of daemons
$jps

#Stopping Hadoop cluster
$bin/stop-mapred.sh
$bin/stop-dfs.sh
```

Starting and stopping Hadoop cluster

1. Navigate to the Hadoop directory: `$cd hadoop-1.0.4`
2. Format the NameNode: `$bin/hadoop namenode -format`
3. Start HDFS daemons: `$bin/start-dfs.sh`
4. Start MapReduce daemons: `$bin/start-mapred.sh`
5. Check the status of daemons: `$jps`
6. Stopping the Hadoop cluster: `$bin/stop-mapred.sh $bin/stop-dfs.sh`

5.3 Using Hadoop MapReduce for Batch Data Analysis

Figure 5.7 shows a Hadoop MapReduce workflow for batch analysis of IoT data. Batch analysis is done to aggregate data (computing mean, maximum, minimum, etc.) on various timescales. The data collector retrieves the sensor data collected in the cloud database and creates a raw data file in a form suitable for processing by Hadoop.

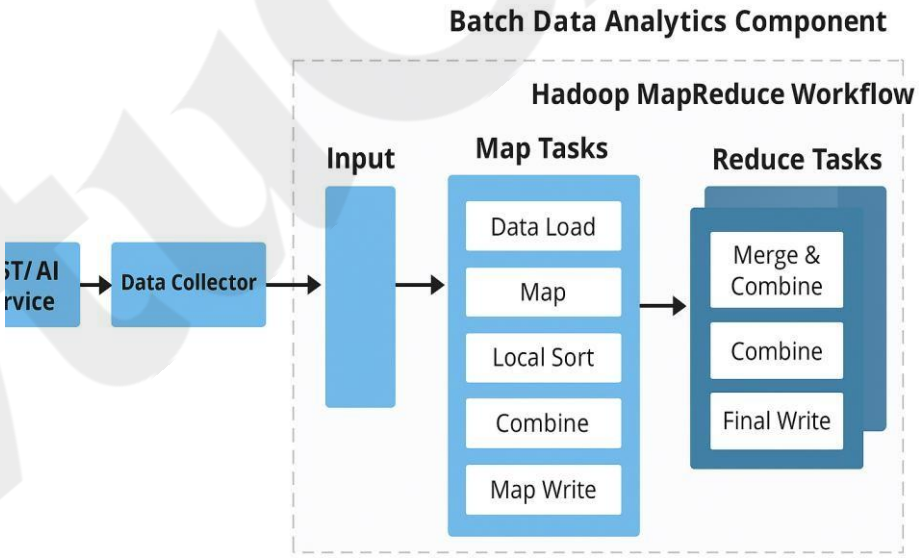

Fig 5.7 Using Hadoop MapReduce for batch analysis of IoT data

Box 5.9:Map program-forestMapper.py

```
#!/usr/bin/env python
import sys

# Calculates mean temperature, humidity, light and CO2
# Input data format:
# "2014-04-29 10:15:32",37,44,31,6
# "2014-04-29 10:15",48.75,31.25,29.0,16.5

# Input comes from STDIN (standard input)
for line in sys.stdin:
    # Remove leading and trailing whitespace
    line = line.strip()
    # Split the line into words
    data = line.split(',')

    # For aggregation by minute
    key = data[0][0:17]

    value = data[1] + ',' + data[2] + ',' + data[3] + ',' + data[4]
    print '%s\t%s' % (key, value)
```

Box 5.10: Reduce program – forestReducer.py

```
#!/usr/bin/env python

from operator import itemgetter

import sys

import numpy as np

current_key = None
current_vals_list = []
word = None

# Input comes from STDIN
for line in sys.stdin:
    # Remove leading and trailing whitespace
    line = line.strip()

    # Parse the input from mapper
    key, values = line.split('\t', 1)
    list_of_values = values.split(',')

    # Convert to list of strings to list of int
    list_of_values = [int(i) for i in list_of_values]
```

```
    if current_key == key:
        current_vals_list.append(list_of_values)
    else:
        if current_key:
            l = len(current_vals_list) + 1
            np_arr = np.array(current_vals_list)
            meanval = [np.mean(np_arr[0:l,0]), np.mean(np_arr[0:l,1]),
                    np.mean(np_arr[0:l,2]), np.mean(np_arr[0:l,3])]
            print "%s\t%s" % (current_key, str(meanval))

        current_vals_list = [list_of_values]
        current_key = key

# Output the last key if needed
if current_key == key:
    l = len(current_vals_list) + 1
    np_arr = np.array(current_vals_list)
    meanval = [np.mean(np_arr[0:l,0]), np.mean(np_arr[0:l,1]),
            np.mean(np_arr[0:l,2]), np.mean(np_arr[0:l,3])]
    print "%s\t%s" % (current_key, str(meanval))
```

Box 5.11: Running MapReduce program on Hadoop cluster

Testing locally

```
bash
CopyEdit
$ cat data.txt | python forestMapper.py | python forestReducer.py
# Running on Hadoop cluster
# Copy data file to HDFS
$ hadoop dfs -copyFromLocal data.txt input

# Run Hadoop job
$ hadoop jar /usr/lib/hadoop/hadoop-streaming.jar \
   -file forestMapper.py -mapper forestMapper.py \
   -file forestReducer.py -reducer forestReducer.py \
   -input input -output output

# View output in Hadoop DFS
$ bin/hadoop dfs -cat output/part-00000
```

5.3.1 Hadoop YARN

Hadoop YARN is the next generation architecture of Hadoop (version 2.x). In the YARN architecture, the original processing engine of Hadoop (MapReduce) has been separated from the resource

management (which is now part of YARN) as shown in Figure 5.12. This makes YARN extremely an open system for Hadoop that supports different processing engines on a Hadoop cluster such as MapReduce for batch processing, Apache Tez for stream processing, etc.

Figure 5.13 shows the MapReduce job execution workflow for next generation Hadoop MapReduce framework (MR2). The next generation MapReduce architecture divides the two major functions of the JobTracker – resource management and job life-cycle management – into two separate components: **ResourceManager** and **ApplicationMaster**. The key components of YARN are described as follows:

- **Resource Manager (RM):** RM manages the global assignment of compute resources to applications. RM consists of two main services:
  - **Scheduler:** Scheduler is a pluggable service that manages and enforces the resource scheduling policy in the cluster.
  - **Application Manager (AsM):** AsM maintains the running Application Masters in the cluster. AsM is responsible for starting application masters on node managers and restarting them on different nodes in case of failures.
- **Application Master (AM):** A per-application AM manages the application's lifecycle. AM is responsible for negotiating resources from the RM and working with the NM to execute and monitor the tasks.
- **Node Manager (NM):** A per-machine NM manages the user processes on that machine.

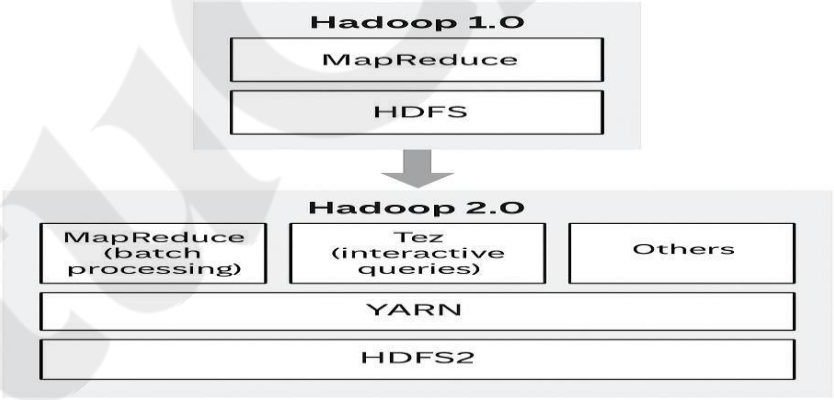

Fig 5.8 Comparison of Hadoop 1.x and 2.x Architecture

Containers: Container is a bundle of resources allocated by RM (memory, CPU, network, etc.). A container is a conceptual entity that grants an application the privilege to use a certain amount of resources on a given machine to run a component of the task. Each node has an NM that spawns multiple containers based on the resource allocations made by the RM.

Figure 5.8 shows a YARN cluster with a Resource Manager node and the Node Manager nodes. There are many Application Masters running as there are multiple applications. Each application's AM manages the application tasks such as starting, monitoring, and restarting tasks in case of failures. Each application has multiple tasks. Each task runs in a separate container. Containers in YARN architecture are similar to task slots in Hadoop MapReduce 1.x (MR1). However, unlike MR1 which differentiates between map and reduce slots, containers in YARN can be used for both map and reduce tasks. The slot allocation model in MR1 consists of a predefined number of map slots and reduce slots. This allocation of slots results in low cluster utilization. The resource allocation model of YARN is more flexible with introduction of resource containers which improve utilization.

To better understand the YARN job execution workflow let us analyze the interactions between the main components on YARN. Figure 5.9 shows the interactions between a Client and Resource Manager. Job execution begins with the submission of a new application request by the client to the RM. The RM then responds with a unique application ID and information about cluster resource capabilities that the client will need in requesting resources for running the application's AM. Using the information received from the RM, the client constructs and submits an Application Submission Context which contains information such as scheduled queue, priority and user information. The Application Submission Context also contains a Container Launch Context which contains the application's jar, job scripts, security tokens and any resource requirements. The client can query the RM for application reports. The client can also "force kill" an application by sending a request to the RM.

Figure 5.10 shows the interactions between Resource Manager and Application Master. Upon receiving an application submission context from a client, the RM finds an available container meeting the resource requirements for running the AM for that application. On finding a suitable container, the RM contacts the NM for the container to start the AM process on its node. When the AM is launched it registers itself with the RM. The registration process consists of handshaking that conveys information such as the RPC port that the AM will be listening on,the tracking URL for monitoring the application's status and progress etc.

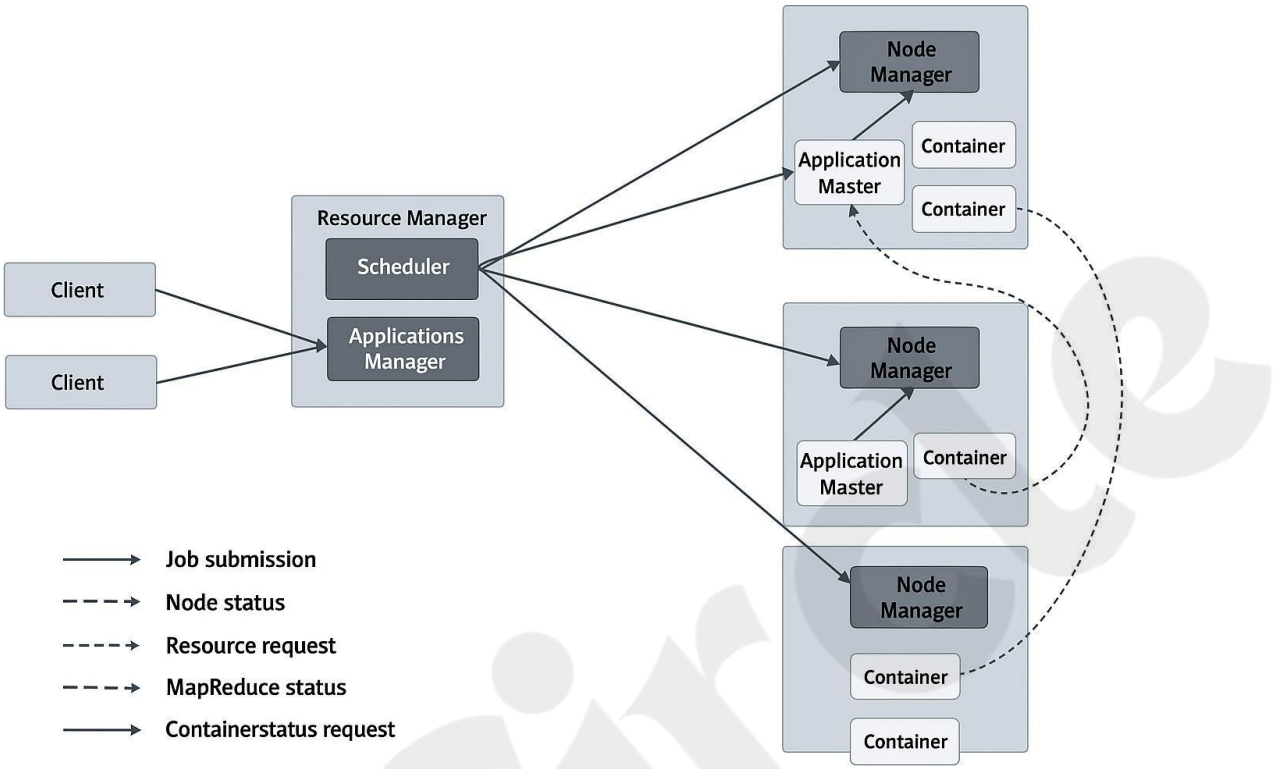

Fig 5.9 Hadoop MapReduce Next Generation (YARN) job execution

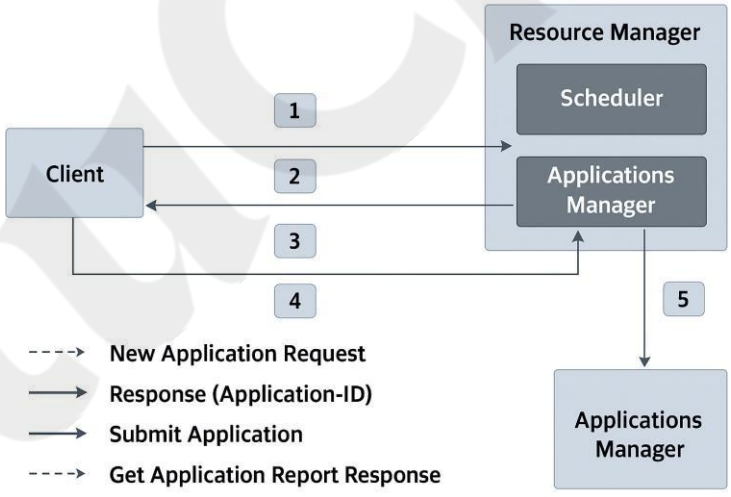

Fig 5.10 Client-Resource Manager Interaction

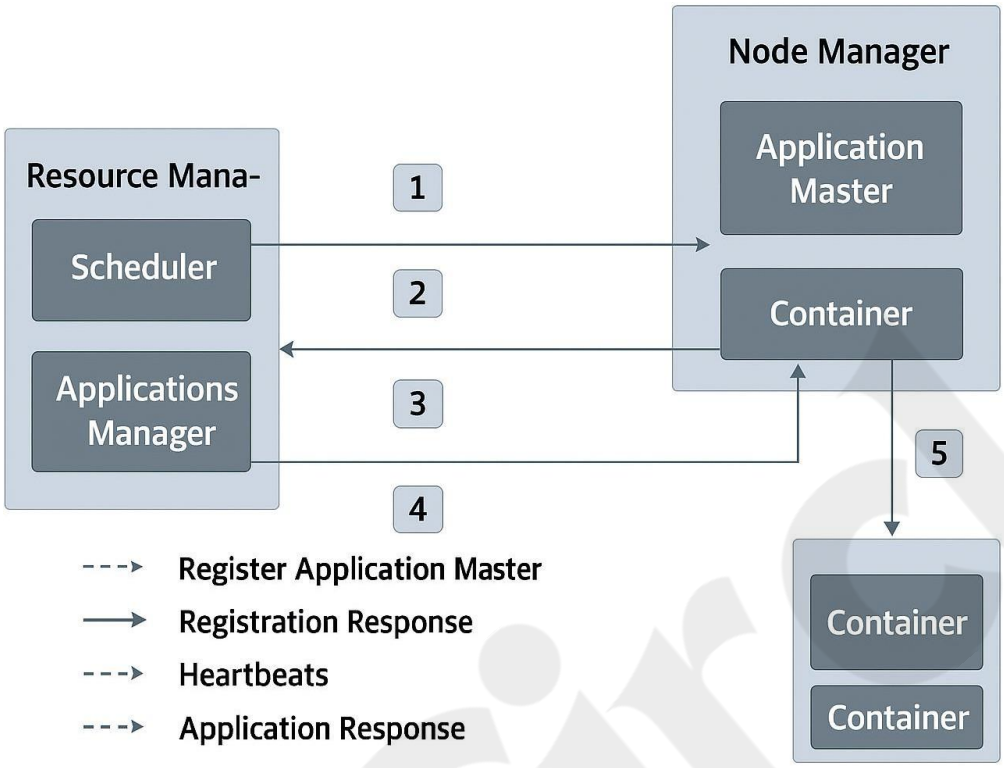

Fig 5.11 Resource Manager-Application Master Interaction

5.4 Apache Oozie

In the previous section you lead about the Hadoop framework and how the MapReduce jobs can be used for analyzing 1oT data. Many IoT applications require more than one MapReduce job to be chained to perform data analysis. This can be accomplished using Apache Oozie system. The workflow scheduler system tit allows managing Hadoop jobs. With Oozie, you can create workflows which area collection of actions (such as MapReduce jobs arranged as Direct Acyclic Graphs (DAG). Control dependencies exits between the ations in a workflow. Thus an ation is executed only when the preceding stion is completed. An Oozie workflow specifies a sequence of action thit need tobe executed using an XML-based Process Definition Language called hPDL. Ouzie supports various types of action such as Hadoop MapRedace, Hadoop le system, Pg, Java, Ema Shell, Hive, Sgoop, SSH and custom actions,

5.4.1 Setting up Oozle

Dorie requires « Hadoop installation and canbe setup upon ether single node ora cluster two or more nodes. Before sting up Hadoop create anew user and group a follows:
from PIL import Image

import pytesseract

```
# Load the image
image_path = "your_image.png"  # Replace with your actual image path
image = Image.open(image_path)

# Perform OCR
text = pytesseract.image_to_string(image)

print(text)
sudo adduser hduser sudo
```

Next, follow the steps for setting up Hadoop described in the previous section. After setting up Hadoop install the packages required for setting up Oozie as follows:

```
$ sudo apt-get install maven
sudo apt-get install zip
sudo apt-get install unzip
```

Next, download and build Oozie using the following commands:

```
$ wget http://supergsego.com/apache/oozie/3.3.2/oozie-3.3.2.tar.gz
tar xvzf oozie-3.3.2.tar.gz
cd oozie-3.3.2
./mkdistro.sh -DskipTests
```

Create a new directory named 'oozie' and copy the built binaries. Also copy the jar files from 'hadooplols' directory to the libext directory as follows:

```
$ cd /home/hduser
mkdir oozie
cp -R oozie-3.3.2/distro/target/oozie-3.3.2-distro/oozie-3.3.2/* oozie
cd /home/hduser/oozie
mkdir libext
cp /home/hduser/oozie-3.3.2/hadooplibs/hadoop-libraries/hadoop-1.1.1.oozie-3.3.2.2/
hadooplib-1.1.1.oozie-3.3.2.2/ /home/hduser/oozie/libext/
```

Download Ext2Js to the ‘libext’ directory. This is required for the Oozie web console:

```
$ cd /home/hduser/oozie/libext
$ wget http://extjs.com/deploy/ext-2.2.zip
```

Prepare the Oozie WAR file as follows:

```
#Prepare the WAR file
./bin/oozie-setup.sh                                                    prepare-war
```

5.5 Apache Spark

Apache Spark is yet another open source cluster computing framework for data analytics.However,Spark supports in-memory cluster computing and promises to be faster than Hadoop.Spark supports various high level tools for data analysis.

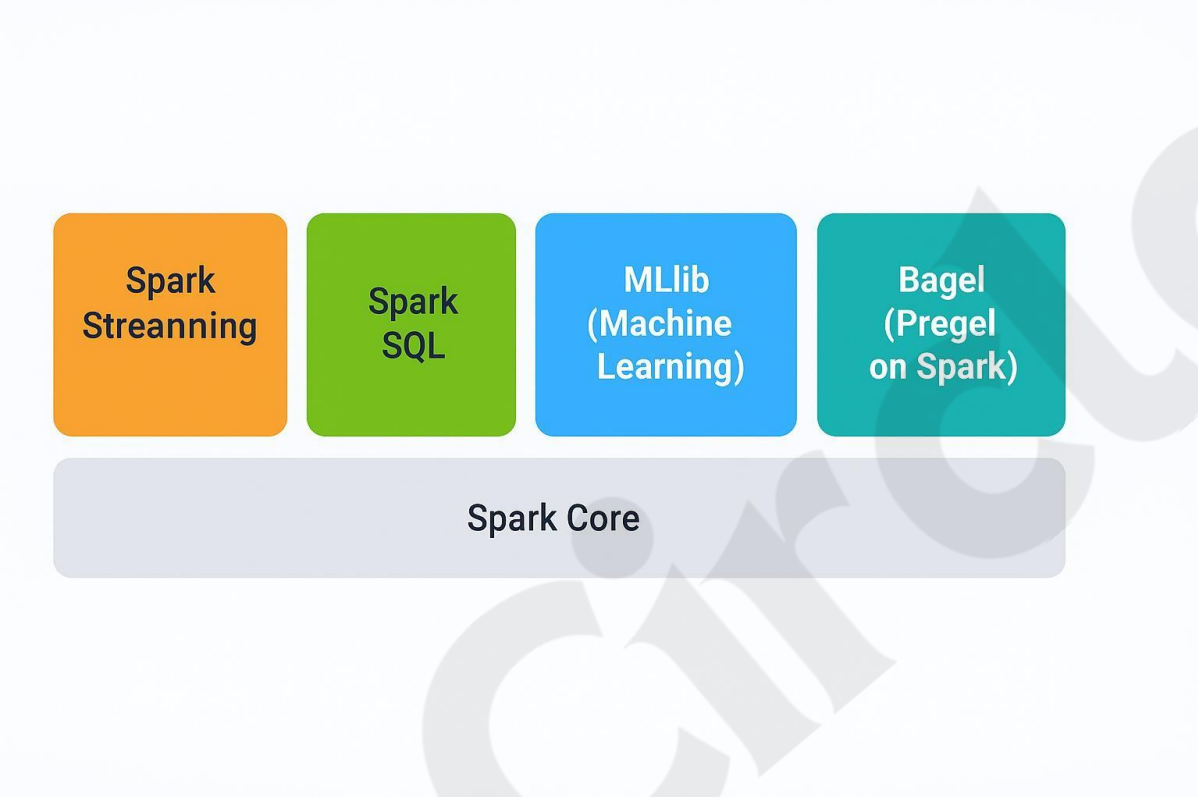

Fig 5.12 Spark Tools

1. Spark Core

- **Foundation layer** of Apache Spark.
- Handles **basic functionality** like:
  - Task scheduling
  - Memory management
  - Fault recovery
  - Interacting with storage systems (HDFS, S3, etc.)
- All other Spark libraries run **on top of Spark Core**.

2. Spark Streaming

- Enables **real-time data processing**.
- Breaks incoming data streams into small batches, processes them, and outputs results quickly.
- Used for applications like **real-time analytics, fraud detection, live dashboards**.

3. Spark SQL

- Allows working with **structured data** using **SQL-like queries**.
- Can read data from:
  - Hive
  - JSON
  - Parquet
  - JDBC
- Integrates with Spark's DataFrame and Dataset APIs.

4. MLlib (Machine Learning Library)

- Spark's **machine learning framework**.
- Provides tools for:
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering
- Designed for **scalable machine learning** on large datasets.

5. GraphX

- For **graph processing and computation**.
- Allows representation of graphs (nodes & edges) and running graph algorithms like:
  - PageRank
  - Connected Components
  - Shortest Path

---

6. Bagel (Pregel on Spark)

- Implementation of **Google's Pregel** graph processing model on Spark.
- Optimized for **large-scale graph computations**.

5.6 Apache Storm

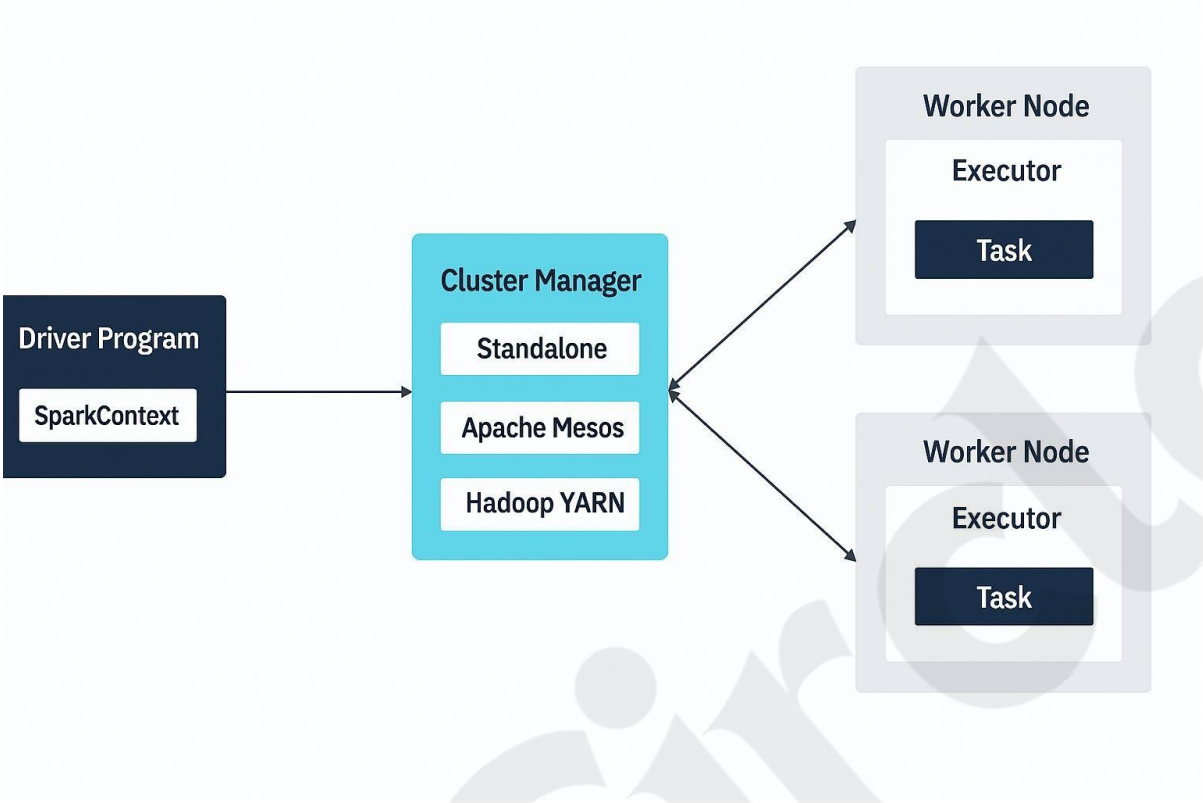

Fig 5.13 Components of a Storm Cluster

The diagram illustrates the Apache Spark Cluster Architecture.

1. Driver Program

The main application process that runs the user's Spark code. Contains the SparkContext, which coordinates the execution of tasks across the cluster. Responsible for converting Spark code into a logical plan, distributing tasks to executors, and collecting results.

2. Cluster Manager

Allocates resources across the cluster for Spark applications. Types include:

1. Standalone – Spark's built-in cluster manager.
2. Apache Mesos – General cluster manager for multiple distributed applications.
3. Hadoop YARN – Cluster manager from the Hadoop ecosystem.

3. Worker Nodes

Machines in the cluster that run the actual computations. Each worker node has:

- Executor: A process responsible for running individual tasks for a Spark job.
- Task: A single unit of work sent by the driver to be executed.

Workflow

1. Driver Program creates the SparkContext.
2. SparkContext connects to the Cluster Manager.
3. Cluster Manager allocates executors on Worker Nodes.
4. Executors run tasks in parallel and store intermediate data.
5. Results are sent back to the Driver Program.

Setting up a Storm Cluster: Study Python scripts

5.7 Using apache Storm for real time data analysis

REST-based Approach

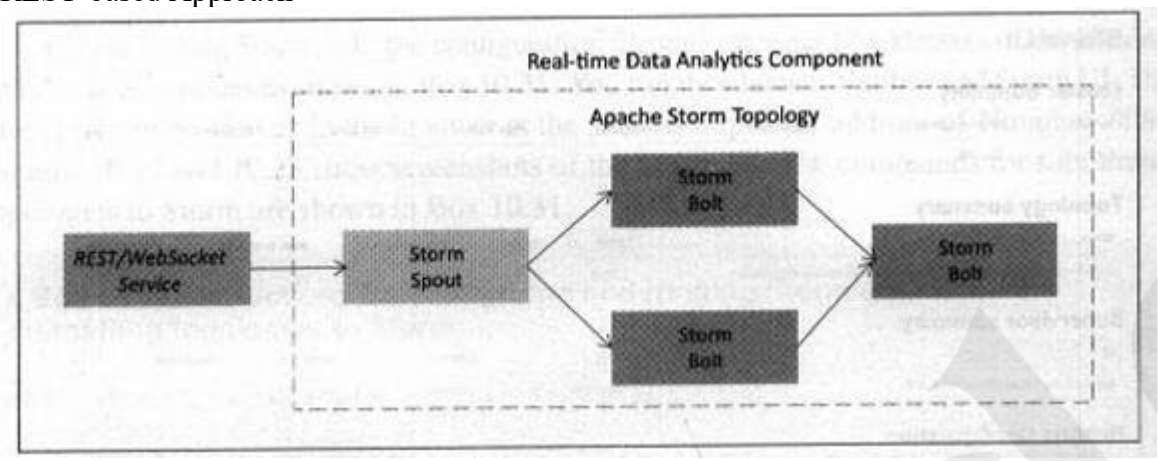

Fig 5.14 Using Apache Storm for real time data analysis of IoT data

Storm Topology and Decision Tree Classifier

The Storm topology used in this example comprises of one Spout and one Bolt. The Spout retrieves the sensor data from Xively cloud and then emits streams of sensor readings. The Bolt processes the data and makes the predictions using a Decision Tree based machine learning classifier.

Decision Trees are a supervised learning method that use a tree created from simple decision rules learned from the training data as a predictive model. The predictive model is in the form of a tree that can be used to predict the value of a target variable based on several attribute variables. Each node in the tree corresponds to one attribute in the dataset on which the "split" is performed. Each leaf in a decision tree represents a value of the target variable. The learning process involves recursively splitting on the attributes until all the samples in the child node have the same value of the target variable or splitting further results in no further information gain. To select the best attribute for splitting at each stage, different metrics can be used.

Before the classifier can be used in the Bolt, the classifier has to be trained. Box 10.32 shows the Python code for training and saving the classifier. The classifier file is then included in the Storm project. Figure 10.31 shows the decision tree generated for the forest fire detection example. The tree shows the attributes on which splitting is done at each step and the split values. Also shown in the figure are the error, total number of samples at each node and the number of samples in each class (in the value array). For example, the first split is done on the second column (attribute X[1] - Humidity) and the total number of samples in the training set is 440. On the first split, there are 248 samples in first class and 192 samples in the second class.

5.7.2 WebSocket based Approach

Application Messaging Protocol (WAMP) and Storm Integration

Application Messaging Protocol (WAMP) which is a sub-protocol of WebSocket. You learned about AutoBahn, an open source implementation of WAMP. The deployment design for the WebSocket implementation is shown

The WAMP Publisher application is a part of the controller component. The sensor data is published by the controller to a topic managed by the WAMP Broker. The WAMP Subscriber component subscribes to the topic managed by the Broker. The centralized controller stores the data in a MongoDB database and also pushes the data to a ZeroMQ queue.

The analysis of data is done by a Storm cluster. A Storm Spout pulls the data to be analyzed from the ZeroMQ queue and emits a stream of tuples. The stream is consumed and processed by the Storm Bolt. The implementations of the Storm Spout and Bolt for real-time analysis of data is shown. The Storm Bolt uses a Decision Tree classifier for making the predictions.