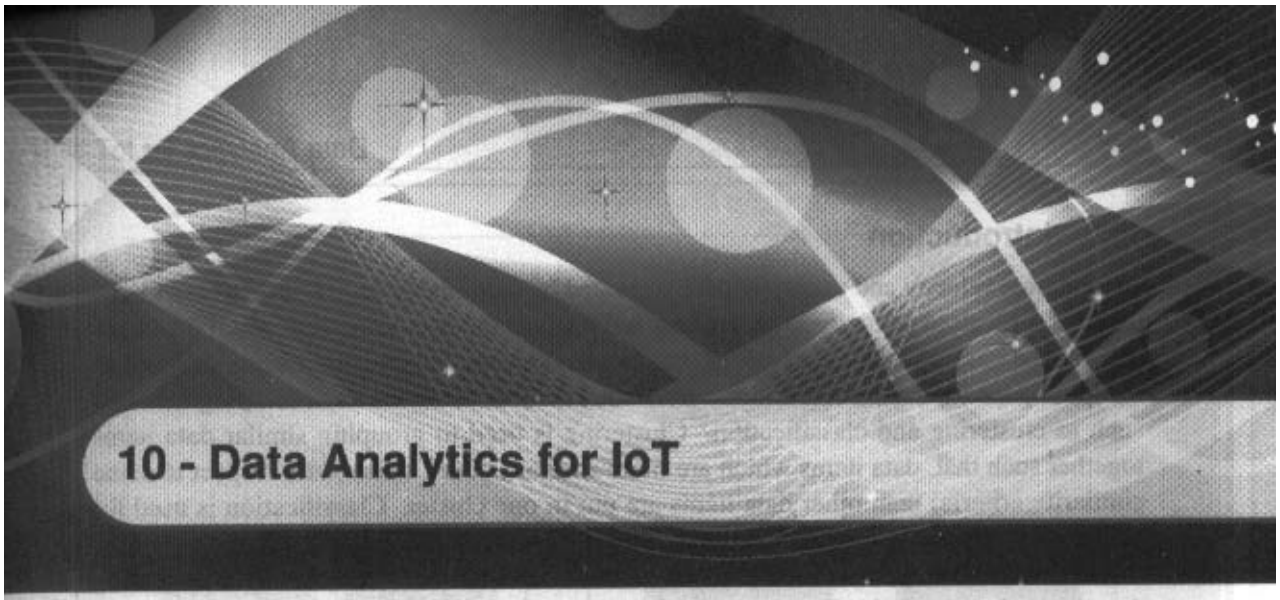

10 - Data Analytics for IoT

This Chapter Covers

- Overview of MapReduce parallel programming model
- Overview of Hadoop
- Case study on batch data analysis using Hadoop
- Case study on real-time data analysis using Hadoop
- Overview of Apache Oozie
- Overview of Apache Spark
- Overview of Apache Storm
- Case study on using Apache Storm for real-time data analysis

10.1 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 grouping similar data items together such that, data items which are more similar to each other (with respect to some similarity criteria) than other data items are put in one cluster. Classification is used for categorizing objects into predefined categories.

In this chapter, you will learn about various frameworks for data analysis including Apache Hadoop, Apache Oozie, Apache Spark and Apache Storm. Case studies on batch and real-time data analysis for a forest fire detection system are described. Before going into the specifics of the data analysis tools, let us look at the IoT system and the requirements for data analysis.

Figure 10.1 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 to predict whether fire has broken out in the forest.

Figure 10.2 shows an example of the data collected for forest fire detection. Each row in the table shows timestamped readings of temperature, humidity, light and *CO* sensors. By analyzing the sensor readings in real-time (each row of table), predictions can be made about the occurrence of a forest fire. The sensor readings can also be aggregated on a various timescales (minute, hourly, daily or monthly) to determine the mean, maximum and minimum readings. This data can help in developing prediction models.

Figure 10.3 shows a schematic diagram of forest fire detection end node. The end node is based on a Raspberry Pi device and uses DHT22 temperature and humidity sensor, light dependent resistor and MICS5525 *CO* sensor. Box 10.1 shows the Python code for the native controller service than runs on the end nodes. This example uses the Xively PaaS for storing data. In the *setupController* function new 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.

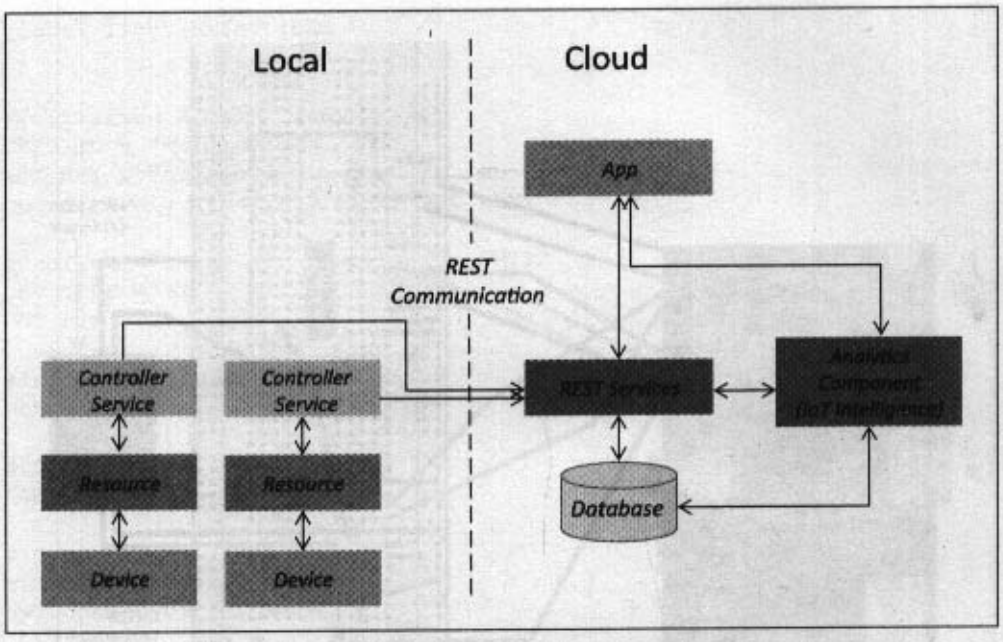

Figure 10.1: Deployment design of forest fire detection system

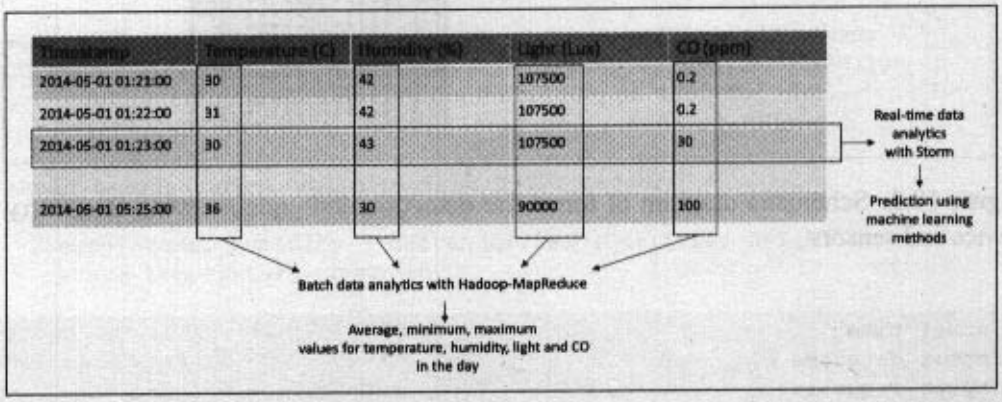

Figure 10.2: Data analysis for forest fire detection

■ Box 10.1: Controller service for forest fire detection system - controller.py

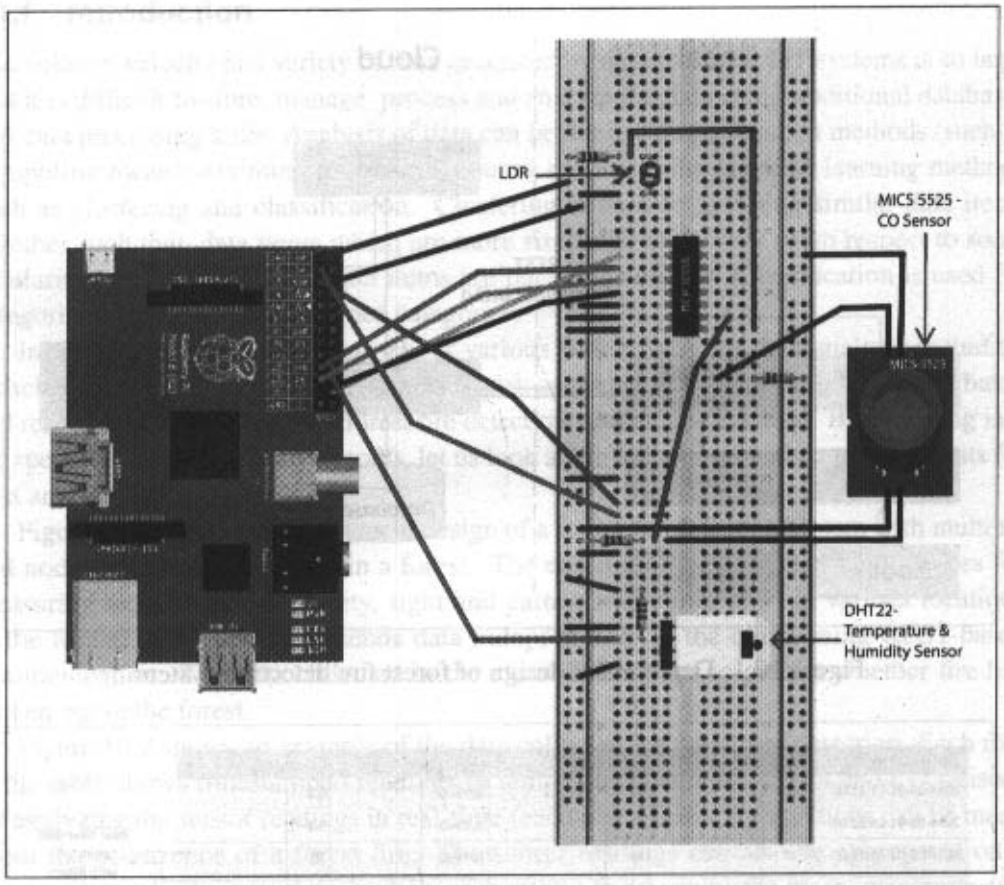

Figure 10.3: Schematic diagram of forest fire detection end node showing Raspberry Pi device and sensors

```
import time
import datetime
import requests
import xively
import dhtreader
import spidev

global temp_datastream
global CO_datastream
global humidity_datastream
```

```
global light_datastream


#Initialize Xively Feed
FEED_ID = "<enter feed ID>"
API_KEY = "<enter API key>"
api = xively.XivelyAPIClient(API_KEY)

#Configure these pin numbers for DHT22
DEV_TYPE = 22
DHT_PIN = 24

#Initialize DHT22
dhtreader.init()

#LDR channel on MCP3008
LIGHT_CHANNEL = 0

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)

#MICS5525 CO sensor channel on MCP3008
CO_CHANNEL = 1

#Conversions based on Rs/Ro vs ppm plot of MICS5525 CO sensor
CO_Conversions = [((0,100),(0,0.25)),((100,133),(0.25,0.325)),
      ((133,167),(0.325,0.475)),((167,200),(0.475, 0.575)),
      ((200,233),(0.575,0.665)),((233,267),(0.666,0.75)) ]

#Read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
   temperature, humidity = dhtreader.read(DEV_TYPE, DHT_PIN)
   return temperature, humidity

#Read LDR connected to MCP3008
def readLDR():
   light_level = ReadChannel(LIGHT_CHANNEL)
   lux = ConvertLux(light_level,2)
   return lux

#Convert LDR reading to Lux
def ConvertLux(data,places):
   R=10 #10k-ohm resistor connected to LDR
   volts = (data * 3.3) / 1023
```

```
    volts = round(volts,places)
    lux=500*(3.3-volts)/(R*volts)
    return lux

# Read SPI data from MCP3008 chip
def ReadChannel(channel):
    adc = spi.xfer2([1,(8+channel)<<4,0])
    data = ((adc[1]&3) << 8) + adc[2]
    return data

#Read MICS5525 CO sensor connected to MCP3008
def readCOSensor():
    result = ReadChannel(CO_CHANNEL)
    if result == 0:
        resistance = 0
    else:
        resistance = (vin/result - 1)*pullup
    ppmresult = converttoppm(resistance, CO_Conversions)
    return ppmresult

#Convert resistance reading to PPM
def converttoppm(rs,conversions):
    rsper = 100*(float(rs)/r0)
    for a in conversions:
        if a[0][0]>=rsper>a[0][1]:
            mid,hi = rsper-a[0][0],a[0][1]-a[0][0]
            sf = float(mid)/hi
            ppm = sf * (a[1][1]-a[1][0]) + a[1][0]
            return ppm
    return 0


#Controller main function
def runController():
    global temp_datastream
    global CO_datastream
    global humidity_datastream
    global light_datastream

    temperature, humidity=read_DHT22_Sensor()
    light=readLDR()
    CO_reading = readCOSensor()

    temp_datastream.current_value = temperature
    temp_datastream.at = datetime.datetime.utcnow()
```

```
    humidity_datastream.current_value = humidity
    humidity_datastream.at = datetime.datetime.utcnow()

    light_datastream.current_value = light
    light_datastream.at = datetime.datetime.utcnow()

    CO_datastream.current_value = CO_reading
    CO_datastream.at = datetime.datetime.utcnow()

    try:
        temp_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError({0}): {1}".format(e.errno, e.strerror)

    try:
        humidity_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError({0}): {1}".format(e.errno, e.strerror)

    try:
        light_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError({0}): {1}".format(e.errno, e.strerror)

    try:
        CO_datastream.update()
    except requests.HTTPError as e:
        print "HTTPError({0}): {1}".format(e.errno, e.strerror)

#Get existing or create new Xively data stream for temperature
def get_tempdatastream(feed):
    try:
        datastream = feed.datastreams.get("temperature")
        return datastream
    except:
        datastream = feed.datastreams.create("temperature",
tags="temperature")
        return datastream

#Get existing or create new Xively data stream for CO
def get_COdatastream(feed):
    try:
        datastream = feed.datastreams.get("CO")
        return datastream
```

```
    except:
        datastream = feed.datastreams.create("CO", tags="CO")
        return datastream

#Get existing or create new Xively data stream for humidity
def get_humiditydatastream(feed):
    try:
        datastream = feed.datastreams.get("humidity")
        return datastream
    except:
        datastream = feed.datastreams.create("humidity", tags="humidity")
        return datastream

#Get existing or create new Xively data stream for light
def get_lightdatastream(feed):
    try:
        datastream = feed.datastreams.get("light")
        return datastream
    except:
        datastream = feed.datastreams.create("light", tags="light")
        return datastream

#Controller setup function
def setupController():
    global temp_datastream
    global CO_datastream
    global humidity_datastream
    global light_datastream

    feed = api.feeds.get(FEED_ID)
    feed.update()

    temp_datastream = get_tempdatastream(feed)
    temp_datastream.max_value = None
    temp_datastream.min_value = None

    humidity_datastream = get_humiditydatastream(feed)
    humidity_datastream.max_value = None
    humidity_datastream.min_value = None

    light_datastream = get_lightdatastream(feed)
    light_datastream.max_value = None
    light_datastream.min_value = None

    CO_datastream = get_COdatastream(feed)
```

```
    CO_datastream.max_value = None
    CO_datastream.min_value = None

setupController()
while True:
    runController()
    time.sleep(1)
```

10.2 Apache Hadoop

Apache Hadoop [130] is an open source framework for distributed batch processing of big data. MapReduce is parallel programming model [85] suitable for analysis of big data. MapReduce algorithms allow large scale computations to be parallelized across a large cluster of servers.

10.2.1 MapReduce Programming Model

MapReduce is a widely used parallel data processing model for processing and analysis of massive scale data [85]. 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. Run-time 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 of 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 10.4 shows the flow of data for a MapReduce job. MapReduce programs take a set of input key-value pairs and produce a set of output key-value pairs. In the Map phase, data is read from a distributed file system, partitioned among a set of computing nodes in the cluster, and sent to the nodes as a set of key-value pairs. The Map tasks process 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 node running the Map task. When all the Map tasks are completed, the Reduce phase begins in which the intermediate data with the same 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 the 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, data

is moved to the compute nodes which results in delay in data transmission between the nodes in a cluster. MapReduce programming model moves the computation to where the data resides thus decreasing the transmission of data and improving efficiency. MapReduce programming model is well suited for parallel processing of massive scale data in which the data analysis tasks can be accomplished by independent map and reduce operations.

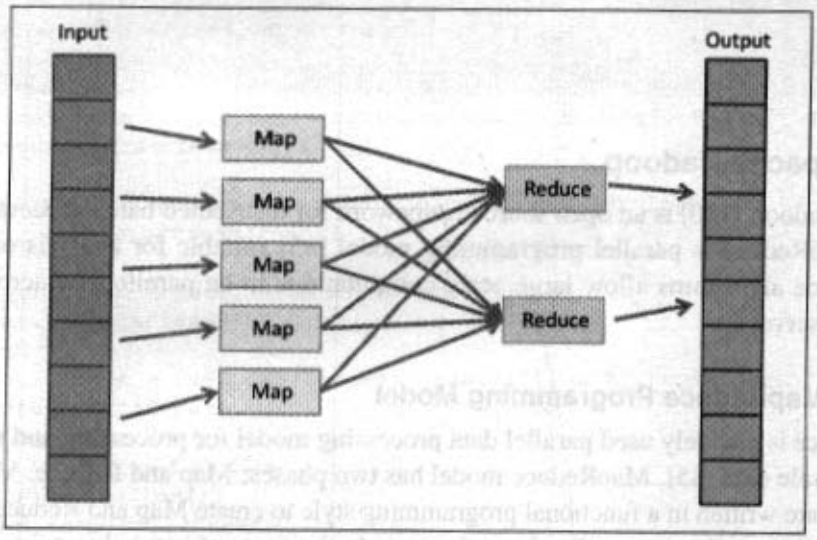

Figure 10.4: Data flow in MapReduce

10.2.2 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, or when they want to

add/copy/move/delete a file. The NameNode responds to the successful requests by returning a list of relevant DataNode servers where the data lives. NameNode serves as both directory namespace manager and 'inode table' for the Hadoop DFS. There is a single NameNode running in any DFS deployment.

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.

10.2.3 MapReduce Job Execution Workflow

Figure 10.6 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

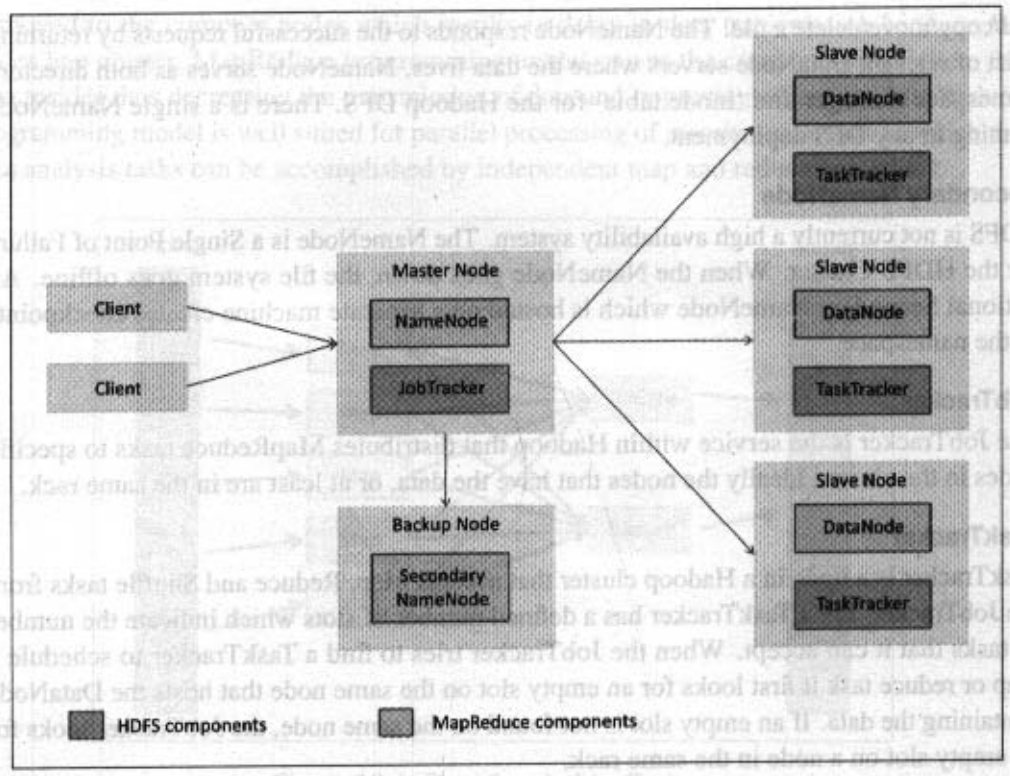

Figure 10.5: Components of a Hadoop cluster

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

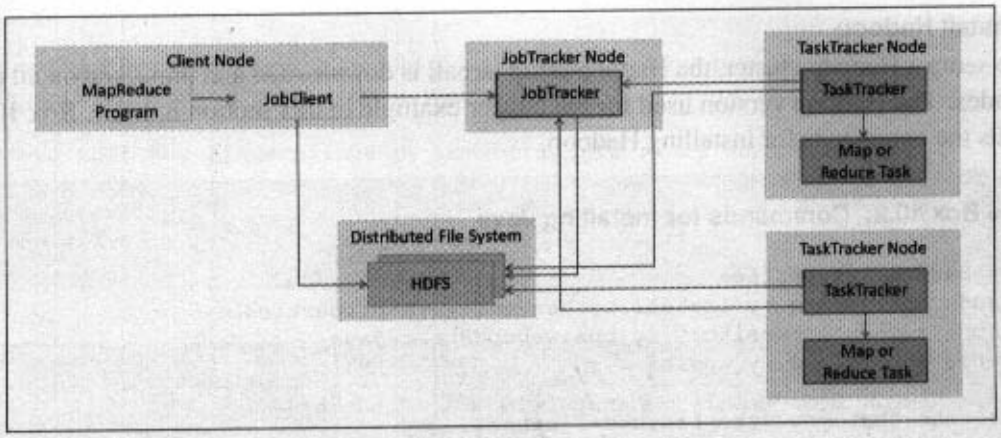

Figure 10.6: Hadoop MapReduce job execution

finishes, 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.

10.2.4 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 10.2 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 10.3 lists the commands for installing Hadoop.

Box 10.2: 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 10.3: 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
```

File Name	Description
core-site.xml	Configuration parameters for Hadoop core which are common to MapReduce and HDFS
mapred-site.xml	Configuration parameters for MapReduce daemons – JobTracker and TaskTracker
hdfs-site.sml	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

Table 10.1: Hadoop configuration files

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 10.1. Boxes 10.4, 10.5, 10.6 and 10.7 show the sample configuration settings for the Hadoop configuration files core-site.xml, mapred-site.xml, hdfs-site.xml, masters/slaves files respectively.

Box 10.4: Sample configuration – core-site.xml

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

Box 10.5: Sample configuration hdfs-site.xml

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

Box 10.6: Sample configuration mapred-site.xml

```
<?xml version="1.0"?>
<configuration>
<property>
```

```
<name>mapred.job.tracker</name>
<value>master:54311</value>
</property>
</configuration>
```

Box 10.7: 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 10.8: 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
```

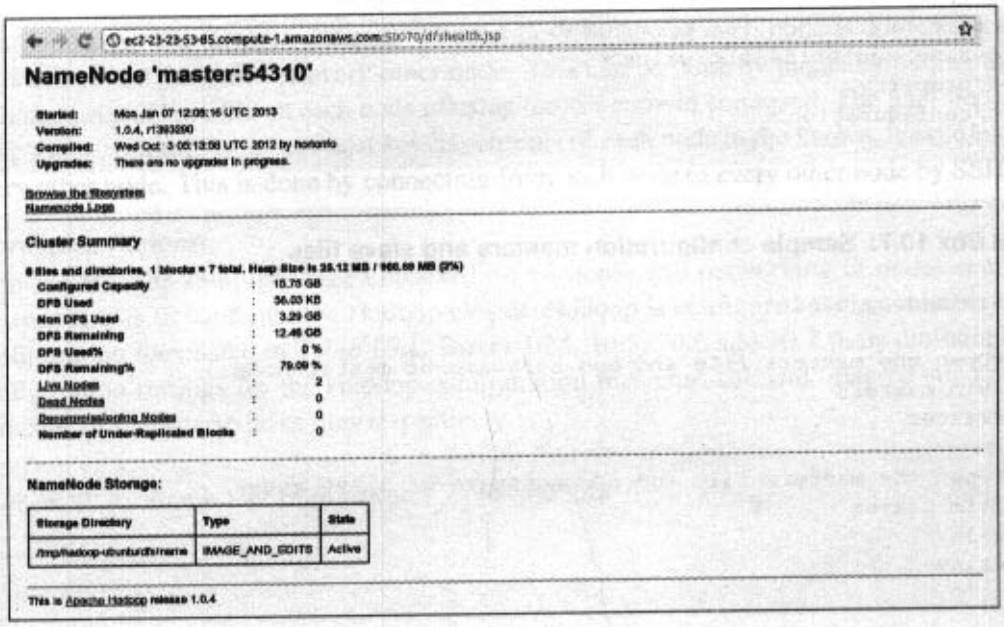

Figure 10.7: Hadoop NameNode status page

Starting and Stopping Hadoop Cluster

Having installed and configured Hadoop the next step is to start the Hadoop cluster. Box 10.8 lists the commands for starting and stopping the Hadoop cluster.

If the Hadoop cluster is correctly installed, configured and started, the status of the Hadoop daemons can be viewed using the administration web-pages for the daemons. Hadoop publishes the status of HDFS and MapReduce jobs to an internally running web server on the master node of the Hadoop cluster. The default addresses of the web UIs are as follows:

NameNode - http://<NameNodeHostName>:50070/

JobTracker - http://<JobTrackerHostName>:50030/

Figure 10.7 shows the Hadoop NameNode status page which provides information about NameNode uptime, the number of live, dead, and decommissioned nodes, host and port information, safe mode status, heap information, audit logs, garbage collection metrics, total load, file operations, and CPU usage.

Figure 10.8 shows the MapReduce administration page which provides host and port information, start time, tracker counts, heap information, scheduling information, current running jobs, retired jobs, job history log, service daemon logs, thread stacks, and a cluster

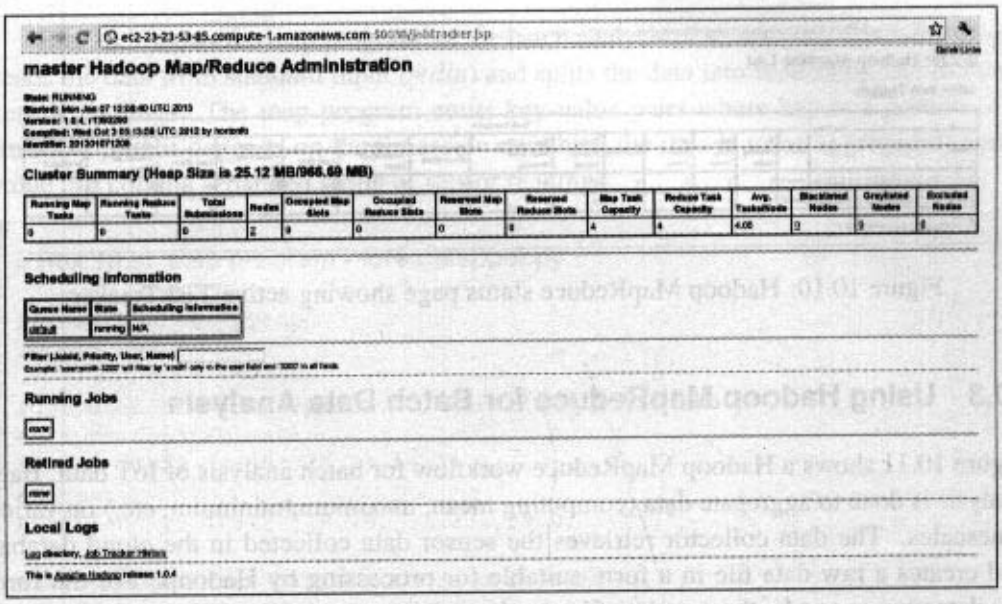

Figure 10.8: Hadoop MapReduce administration page

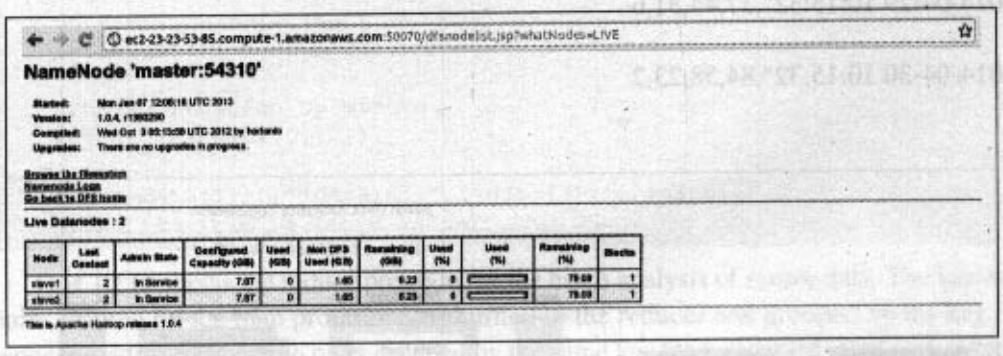

Figure 10.9: Hadoop HDFS status page showing live data nodes

utilization summary.

Figure 10.9 shows the status page of the live data nodes of the Hadoop cluster. The status page shows two live data nodes – slave1 and slave2.

Figure 10.10 shows the status page of the active TaskTrackers of the Hadoop cluster. The status page shows two active TaskTrackers that run on the slave1 and slave2 nodes of the cluster.

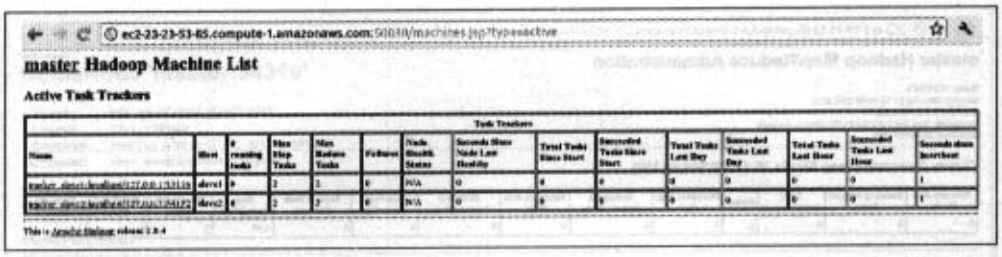

Figure 10.10: Hadoop MapReduce status page showing active TaskTrackers

10.3 Using Hadoop MapReduce for Batch Data Analysis

Figure 10.11 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. For the forest fire detection example, the raw data file consists of the raw sensor readings along with the timestamps as shown below:

"2014-04-29 10:15:32",37,44,31,6

:

"2014-04-30 10:15:32",84,58,23,2

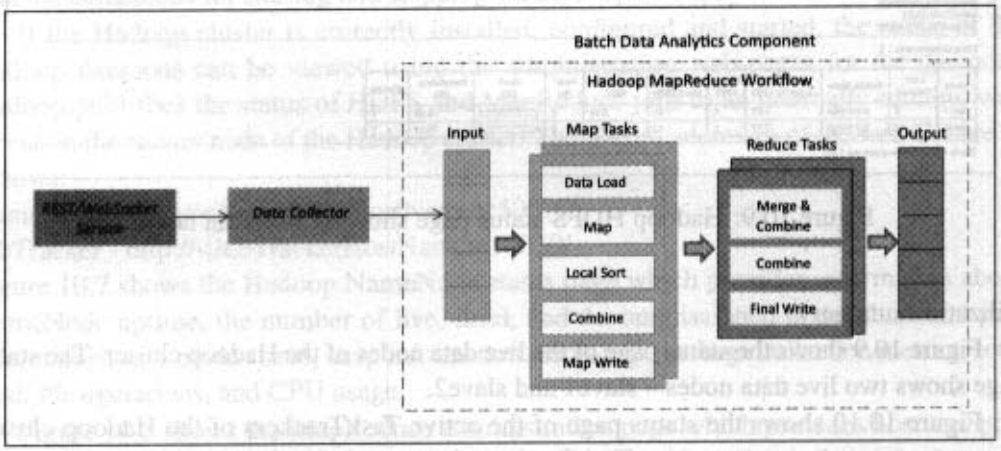

Figure 10.11: Using Hadoop MapReduce for batch analysis of IoT data

Box 10.9 shows the map program for the batch analysis of sensor data. The map program reads the data from standard input (*stdin*) and splits the data into timestamp and individual sensor readings. The map program emits key-value pairs where key is a portion of the timestamp (that depends on the timescale on which the data is to be aggregated) and the value is a comma separated string of sensor readings.

■ Box 10.9: 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
#Output:
#"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()
   data = line.split(',')
   l=len(data)

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

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

Box 10.10 shows the reduce program for the batch analysis of sensor data. The key-value pairs emitted by the map program are shuffled to the reducer and grouped by the key. The reducer reads the key-value pairs grouped by the same key from standard input and computes the means of temperature, humidity, light and CO readings.

■ Box 10.10: 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
         b = np.array(current_vals_list)
         meanval = [np.mean(b[0:l,0]),np.mean(b[0:l,1]),
         np.mean(b[0:l,2]), np.mean(b[0:l,3])]
         print '%s%s' % (current_key, str(meanval))

      current_vals_list = []
      current_vals_list.append(list_of_values)
      current_key = key

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

■ Box 10.11: Running MapReduce program on Hadoop cluster

```
#Testing locally
$cat data.txt | python forestMapper.py | python forestReducer.py
```

```
#Running on Hadoop cluster
#Copy data file to HDFS sudo -u user1 bin/hadoop dfs -copyFromLocal
data.txt input

#Run MapReduce job bin/hadoop jar
contrib/streaming/hadoop-*streaming*.jar
-mapper forestMapper.py -reducer forestReducer.py
-file /home/ubuntu/hadoop/forestMapper.py
-file /home/ubuntu/hadoop/forestReducer.py
-input input/* -output output

#View output bin/hadoop dfs -ls output
bin/hadoop dfs -cat output/part-00000
```

10.3.1 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 10.12. This makes YARN effectively an operating system for Hadoop that supports different processing engines on a Hadoop cluster such as MapReduce for batch processing, Apache Tez [131] for interactive queries, Apache Storm [134] for stream processing, etc.

Figure 10.13 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 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.
  - Applications Manager (AsM): AsM manages the running Application Masters in the cluster. AsM is responsible for starting application masters and for monitoring and restarting them on different nodes in case of failures.
- **Application Master (AM):** A per-application AM manages the application's life cycle. AM is responsible for negotiating resources from the RM and working with the NMs to execute and monitor the tasks.
- **Node Manager (NM):** A per-machine NM manages the user processes on that machine.

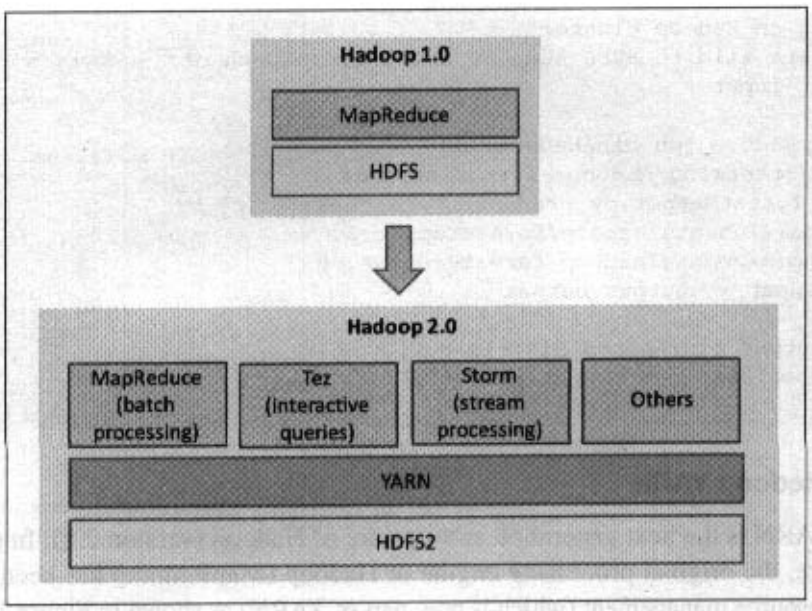

Figure 10.12: Comparison of Hadoop 1.x and 2.x architectures

- **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 task. Each node has an NM that spawns multiple containers based on the resource allocations made by the RM.

Figure 10.13 shows a YARN cluster with a Resource Manager node and three Node Manager nodes. There are as many Application Masters running as there are applications (jobs). 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, each container in YARN can be used for both map and reduce tasks. The resource allocation model in MR1 consists of a predefined number of map slots and reduce slots. This static allocation of slots results in low cluster utilization. The resource allocation model of YARN is more flexible with introduction of resource containers which improve cluster utilization.

To better understand the YARN job execution workflow let us analyze the interactions

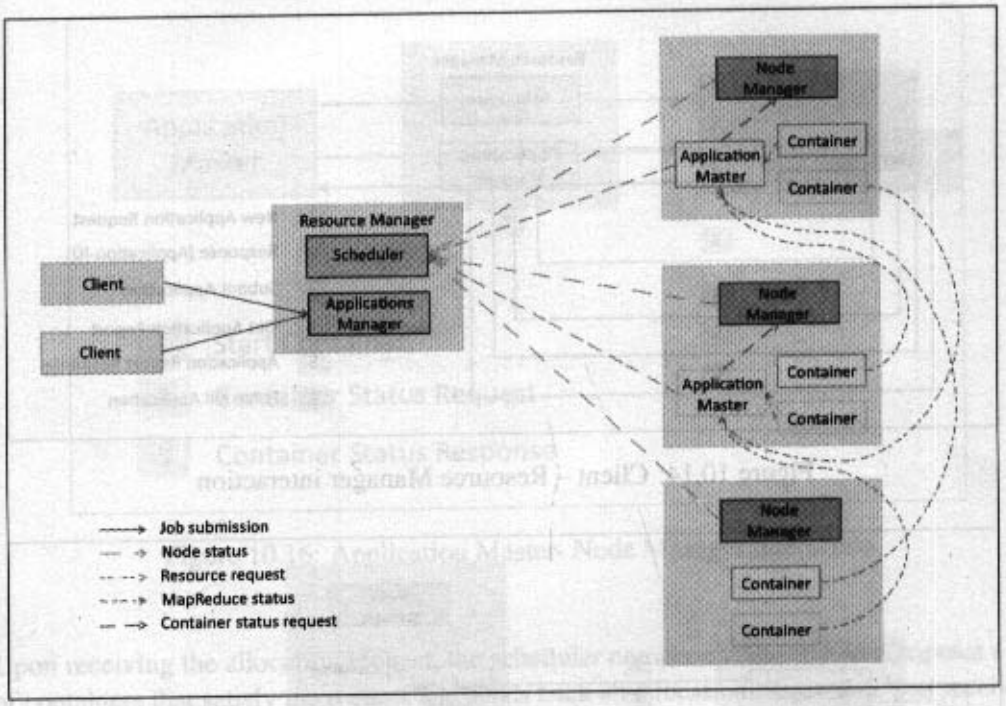

Figure 10.13: Hadoop MapReduce Next Generation (YARN) job execution

between the main components on YARN. Figure 10.14 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 scheduler queue, priority and user information. The Application Submission Context also contains a Container Launch Context which contains the application's jar, job files, 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 10.15 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 the 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

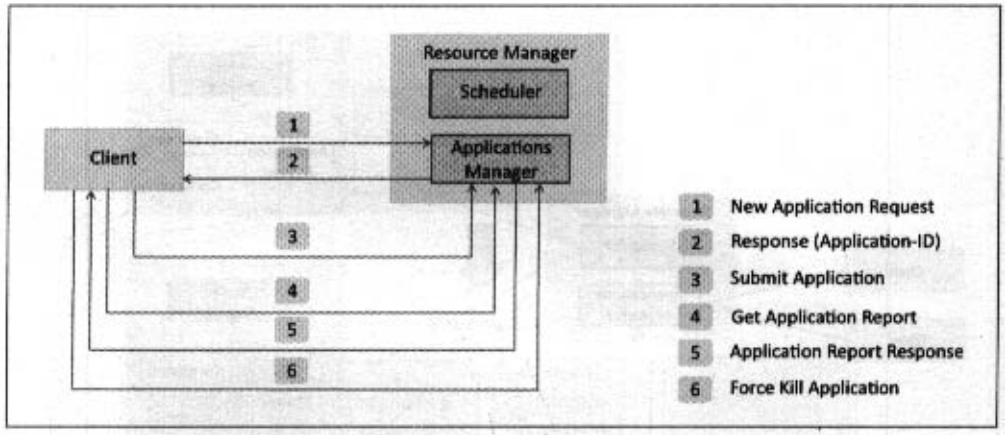

Figure 10.14: Client – Resource Manager interaction

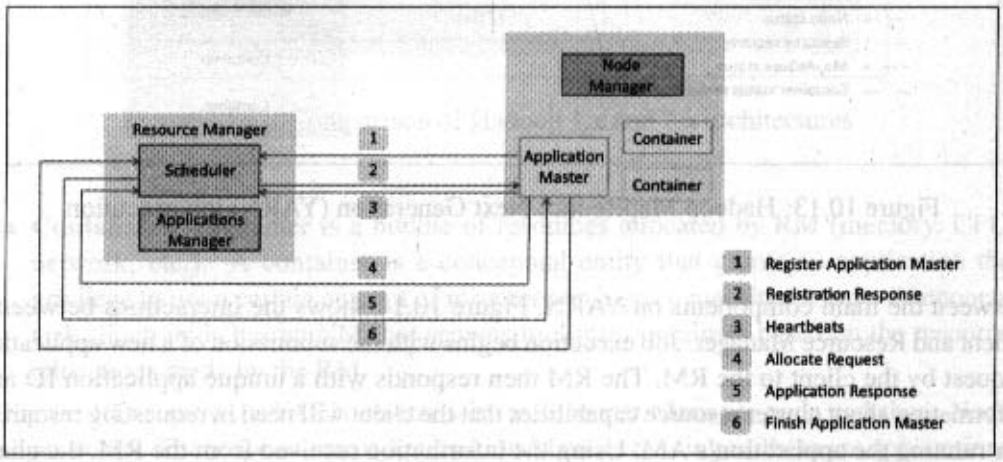

Figure 10.15: Resource Manager – Application Master interaction

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. The registration response from the RM contains information for the AM that is used in calculating and requesting any resource requests for the application's individual tasks (such as minimum and maximum resource capabilities for the cluster). The AM relays heartbeat and progress information to the RM. The AM sends resource allocation requests to the RM that contains a list of requested containers, and may also contain a list of released containers by the AM.

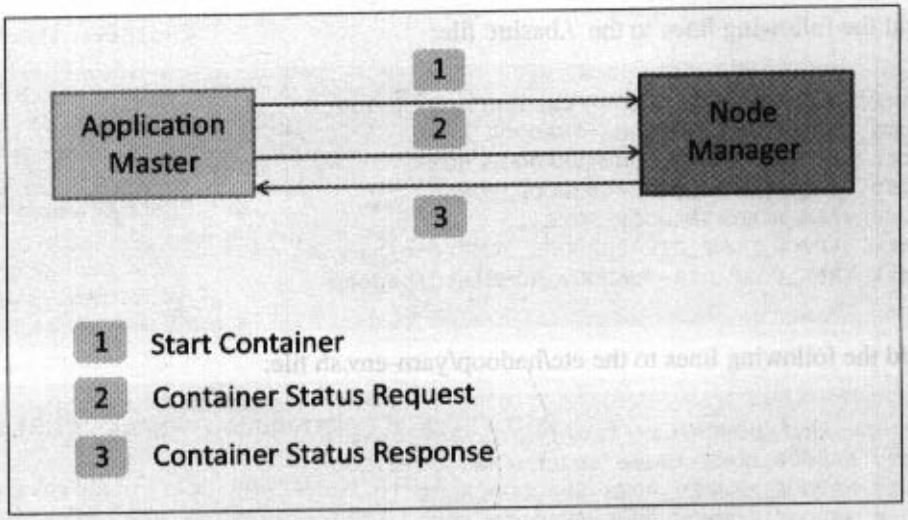

Figure 10.16: Application Master- Node Manager interaction

Upon receiving the allocation request, the scheduler component of the RM computes a list of containers that satisfy the request and sends back an allocation response. Upon receiving the resource list, the AM contacts the associated NMs for starting the containers. When the job finishes, the AM sends a Finish Application message to the RM.

Figure 10.16 shows the interactions between the an Application Master and Node Manager. Based on the resource list received from the RM, the AM requests the hosting NM for each container to start the container. The AM can request and receive a container status report from the Node Manager.

Setting up Hadoop YARN cluster

In the previous section you learned how to setup a Hadoop 1.x cluster. This section describes the steps involved in setting up Hadoop YARN cluster. The initial steps of setting up the hosts, installing Java and configuring the networking are the same as in Hadoop 1.x. The next step is to download the Hadoop YARN setup package and unpack it on all nodes as follows:

```
■ wget http://mirror.cc.columbia.edu/pub/software/apache/
hadoop/common/stable2/hadoop-2.2.0.tar.gz

tar -xzf hadoop-2.2.0.tar.gz
```

Add the following lines to the /.bashrc file:

```
export HADOOP_HOME=/home/ubuntu/hadoop-2.2.0
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

Add the following lines to the etc/hadoop/yarn-env.sh file:

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle/
export HADOOP_HOME=/home/ubuntu/hadoop-2.2.0
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

Next, create temporary folder in HADOOP_HOME:

```
$ mkdir -p $HADOOP_HOME/tmp
```

Next, add the slave hostnames to the etc/hadoop/slaves file on master machine:

```
slave1
slave2
slave3
```

The next step is to edit the Hadoop configuration files. Boxes 10.12, 10.13, 10.14 and 10.15 show the sample configuration settings for the Hadoop configuration files - core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml files respectively.

Box 10.12: Sample configuration – core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
```

```
  <configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://master:9000</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/ubuntu/hadoop-2.2.0/tmp</value>
  </property>
</configuration>
```

Box 10.13: Sample configuration hdfs-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
  <configuration>
  <property>
    <name>dfs.replication</name>
    <value>2</value>
  </property>
  <property>
    <name>dfs.permissions</name>
    <value>false</value>
  </property>
</configuration>
```

Box 10.14: Sample configuration mapred-site.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Box 10.15: Sample configuration yarn-site.xml

```
<?xml version="1.0"?>
<configuration>
   <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce.shuffle</value>
   </property>
   <property>
      <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
   </property>
   <property>
      <name>yarn.resourcemanager.resource-tracker.address</name>
      <value>master:8025</value>
   </property>
   <property>
      <name>yarn.resourcemanager.scheduler.address</name>
      <value>master:8030</value>
   </property>
   <property>
      <name>yarn.resourcemanager.address</name>
      <value>master:8040</value>
   </property>
</configuration>
```

Box 10.16 shows the commands for starting/stopping Hadoop YARN cluster.

Box 10.16: Starting and stopping Hadoop YARN cluster

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

sbin/hadoop-daemon.sh start namenode
sbin/hadoop-daemons.sh start datanode
sbin/yarn-daemon.sh start resourcemanager
sbin/yarn-daemons.sh start nodemanager
sbin/mr-jobhistory-daemon.sh start historyserver


sbin/mr-jobhistory-daemon.sh stop historyserver
```

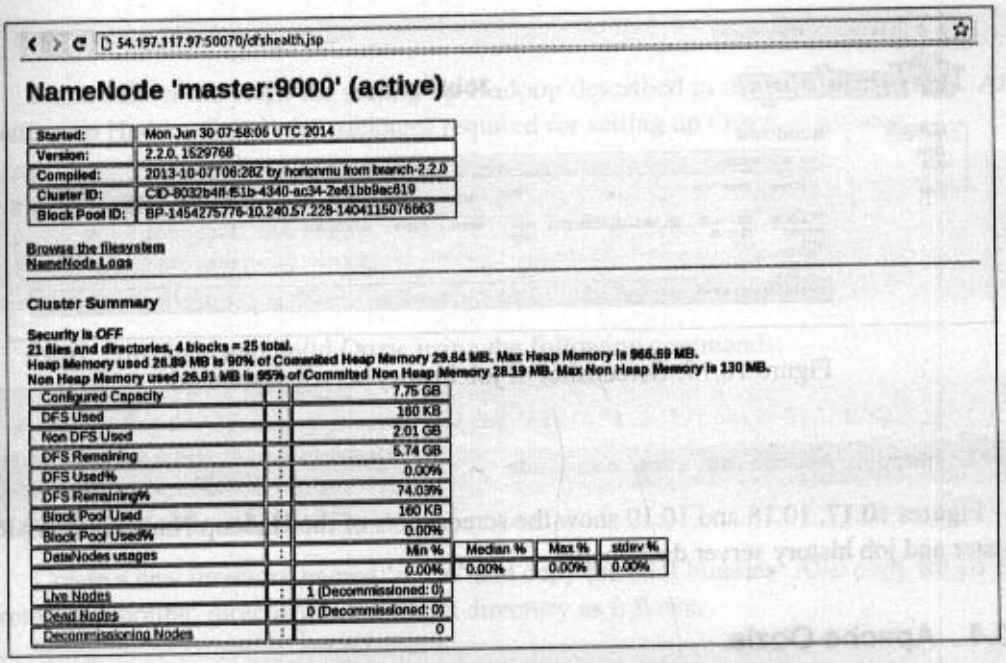

Figure 10.17: Screenshot of Hadoop Namenode dashboard

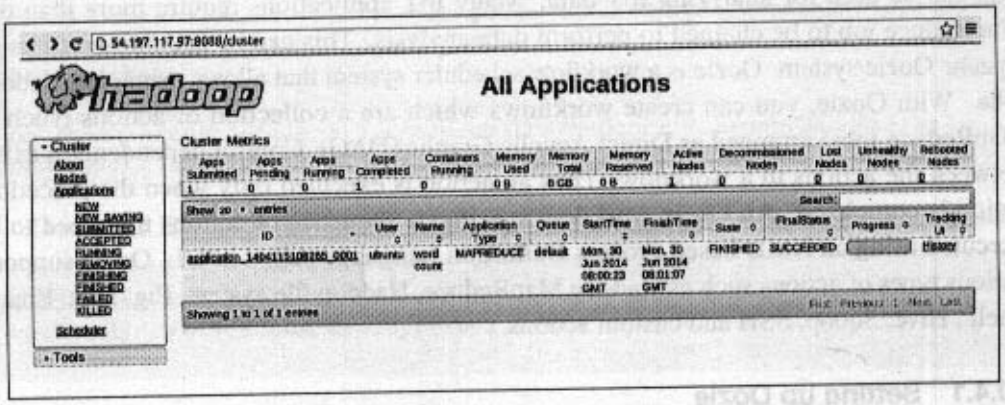

Figure 10.18: Screenshot of YARN cluster dashboard

```
sbin/yarn-daemons.sh stop nodemanager
sbin/yarn-daemon.sh stop resourcemanager
sbin/hadoop-daemons.sh stop datanode
```

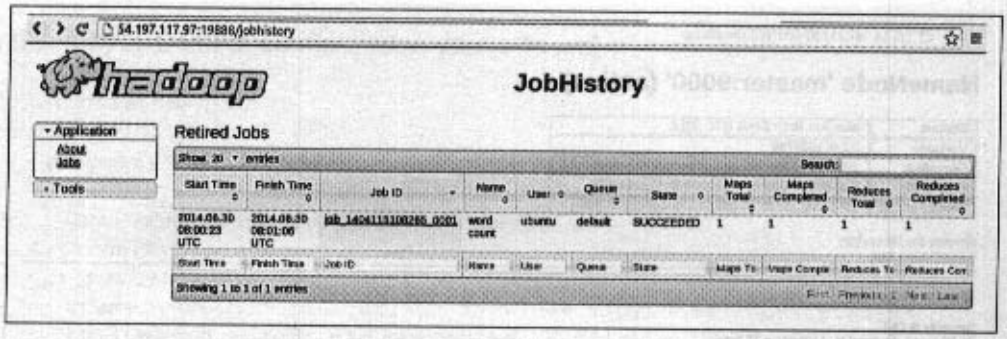

Figure 10.19: Screenshot of job history server dashboard

```
sbin/hadoop-daemon.sh stop namenode
```

Figures 10.17, 10.18 and 10.19 show the screenshots of the Hadoop Namenode, YARN cluster and job history server dashboards.

10.4 Apache Oozie

In the previous section you learned about the Hadoop framework and how the MapReduce jobs can be used for analyzing IoT 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. Oozie is a workflow scheduler system that allows managing Hadoop jobs. With Oozie, you can create workflows which are a collection of actions (such as MapReduce jobs) arranged as Direct Acyclic Graphs (DAG). Control dependencies exists between the actions in a workflow. Thus an action is executed only when the preceding action is completed. An Oozie workflow specifies a sequence of actions that need to be executed using an XML-based Process Definition Language called hPDL. Oozie supports various types of actions such as Hadoop MapReduce, Hadoop file system, Pig, Java, Email , Shell , Hive, Sqoop, SSH and custom actions.

10.4.1 Setting up Oozie

Oozie requires a Hadoop installation and can be setup up on either a single node or a cluster of two or more nodes. Before setting up Hadoop create a new user and group as follows:

```
■ sudo addgroup hadoop
sudo adduser -ingroup hadoop hduser
```

```
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/bin
./mkdistro.sh -DskipTests
```

Create a new directory named 'oozie' and copy the built binaries. Also copy the jar files from 'hadooplibs' 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-1/target/hadooplibs/
hadooplib-1.1.1.oozie-3.3.2/* /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
```

Next, create sharelib on HDFS as follows:

```
#Create sharelib on HDFS
./bin/oozie-setup.sh sharelib create -fs hdfs://master:54310
```

Next, create the OozieDB as follows:

```
./bin/ooziedb.sh create -sqlfile oozie.sql -run
```

Finally use the following command to start Oozie server:

```
#Start Oozie
./bin/oozied.sh start
```

The status of Oozie can be checked from command line or the web console as follows:

```
#To check the status of Oozie from command line:
./bin/oozie admin -oozie http://master:11000/oozie -status
#Oozie Web Console URL:
http://localhost:11000/oozie
```

To setup the Oozie client, copy the client tar file to the 'oozie-client' and add the path in .bashrc file as follows:

```
#Oozie Client Setup cd /home/hduser/
cp /home/hduser/oozie/oozie-client-3.3.2.tar.gz /home/hduser/
tar xvzf oozie-client-3.3.2.tar.gz

#Add to PATH in .bashrc
sudo vim .bashrc
export PATH=$PATH:/home/hduser/oozie-client-3.3.2/bin
```

10.4.2 Oozie Workflows for IoT Data Analysis

Let us look at an example of analyzing machine diagnosis data. Assuming that the data received from a machine has the following structure (including time stamp and the status/error code):

```
#timestamp, status/error "2014-07-01 20:03:18",115
"2014-07-01 20:04:15",106
:
"2014-07-01 20:10:15",110
```

The goal of the analysis job is to find the counts of each status/error code and produce an output with a structure as shown below:

```
#status/error, count 111, 6
112, 7
113, 12
```

Figure 10.20 shows a representation of the Oozie workflow comprising of Hadoop streaming MapReduce job action and Email actions that notify the success or failure of the job.

Boxes 10.17 and 10.18 show the map and reduce programs which are executed in the workflow. The map program parses the status/error code from each line in the input and emits key-value pairs where key is the status/error code and value is 1. The reduce program receives the key-value pairs emitted by the map program aggregated by the same key. For each key, the reduce program calculates the count and emits key-value pairs where key is the status/error code and the value is the count.

Box 10.17: Map program for computing counts of machine status/error codes

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

#Data format
#"2014-07-01 20:03:18",115

# 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(',')
   print '%s%s' % (data[1], 1)
```

Box 10.18: Reduce program for computing counts of machine status/error codes

```
#!/usr/bin/env python
from operator import itemgetter
import sys
```

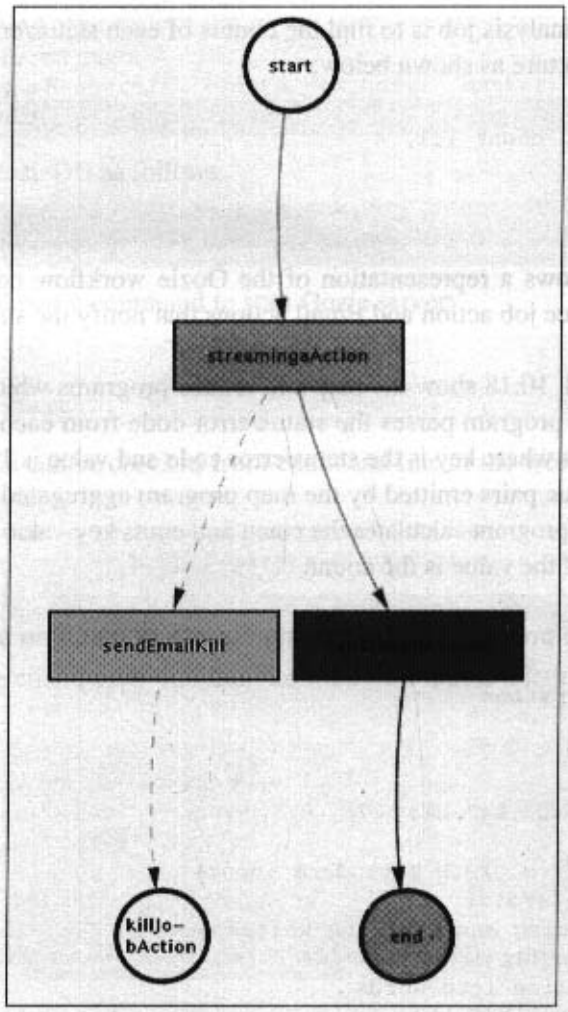

Figure 10.20: Oozie workflow

```
current_key = None
current_count = 0
key = None

# input comes from STDIN
```

```
for line in sys.stdin:
   line = line.strip()

   key, count = line.split('', 1)
   count = int(count)

   if current_key == key:
      current_count += count
   else:
      if current_key:
         unpackedKey = current_key.split(',')
         print '%s%s' % (current_key, current_count)
      current_count = count
         current_key = key

if current_key == key:
   unpackedKey = current_key.split(',')
   print '%s%s' % (current_key, current_count)
```

Box 10.20 shows the specification for the Oozie workflow shown in Figure 10.20. Oozie workflow has been parameterized with variables within the workflow definition. The values of these variables are provided in the job properties file shown in Box 10.19

■ Box 10.19: Job properties file for Oozie workflow

```
nameNode=hdfs://master:54310
jobTracker=master:54311
queueName=default

oozie.libpath=${nameNode}/user/hduser/share/lib
oozie.use.system.libpath=true
oozie.wf.rerun.failnodes=true

oozieProjectRoot=${nameNode}/user/hduser/oozieProject
appPath=${oozieProjectRoot}/pythonApplication
oozie.wf.application.path=${appPath}
oozieLibPath=${oozie.libpath}

inputDir=${oozieProjectRoot}/pythonApplication/data/
outputDir=${appPath}/output
```

Box 10.20: Oozie workflow for computing counts of machine status/error codes

```
<workflow-app name="PythonOozieApp" xmlns="uri:oozie:workflow:0.1">
 <start to="streamingaAction"/>
  <action name="streamingaAction">
<map-reduce>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
    <prepare>
     <delete path="${outputDir}"/>
    </prepare>
    <streaming>
     <mapper>python Mapper.py</mapper>
     <reducer>python Reducer.py</reducer>
    </streaming>
    <configuration>
<property>
<name>oozie.libpath</name>
    <value>${oozieLibPath}/mapreduce-streaming</value>
   </property>
     <property>
      <name>mapred.input.dir</name>
      <value>${inputDir}</value>
     </property>
     <property>
      <name>mapred.output.dir</name>
      <value>${outputDir}</value>
     </property>
<property>
      <name>mapred.reduce.tasks</name>
      <value>1</value>
     </property>
    </configuration>
<file>${appPath}/Mapper.py#Mapper.py</file>
<file>${appPath}/Reducer.py#Reducer.py</file>
  </map-reduce>
<ok to="sendEmailSuccess"/>
<error to="sendEmailKill"/>
  </action>

<action name="sendEmailSuccess">
  <email xmlns="uri:oozie:email-action:0.1">
    <to>${emailToAddress}</to>
    <subject>Status of workflow ${wf:id()}</subject>
```

```
    <body>The workflow ${wf:id()} completed successfully</body>
  </email>
  <ok to="end"/>
  <error to="end"/>
 </action>
 <action name="sendEmailKill">
  <email xmlns="uri:oozie:email-action:0.1">
   <to>${emailToAddress}</to>
   <subject>Status of workflow ${wf:id()}</subject>
   <body>The workflow ${wf:id()} had issues and was killed.
The error message is:  ${wf:errorMessage(wf:lastErrorNode())}</body>
  </email>
  <ok to="killJobAction"/>
  <error to="killJobAction"/>
 </action>

  <kill name="killJobAction">
   <message>"Killed job due to error:
${wf:errorMessage(wf:lastErrorNode())}"</message>
  </kill>
 <end name="end" />
</workflow-app>
```

Let us now look at a more complicated workflow which has two MapReduce jobs. Extending the example described earlier in this section, let us say we want to find the status/error code with the maximum count. The MapReduce job in the earlier workflow computed the counts for each status/error code. A second MapReduce job, which consumes the output of the first MapReduce job computes the maximum count. The map and reduce programs for the second MapReduce job are shown in Boxes 10.21 and 10.22.

Figure 10.21 shows a DAG representation of the Oozie workflow for computing machine status/error code with maximum count. The specification of the workflow is shown in Box 10.23.

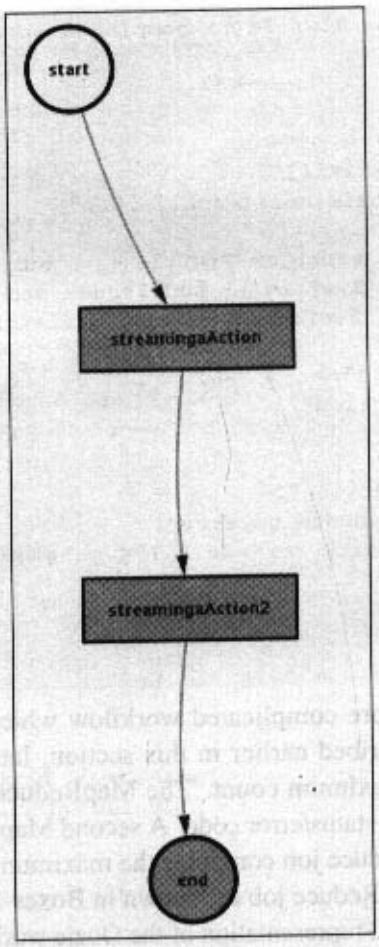

Figure 10.21: Oozie workflow for computing machine status/error code with maximum count

■ Box 10.21: Map program for computing machine status/error code with maximum count

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

#Data format
```

```
#"2014-07-01 20:03:18",115

# 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
   print '%s%s' % (data[0], data[1])
```

■ Box 10.22: Reduce program for computing machine status/error code with maximum count

```
#!/usr/bin/env python
from operator import itemgetter
import sys

current_key = None
current_count = 0
key = None
maxcount=0
maxcountkey=None

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

   # parse the input we got from mapper.py
   key, count = line.split('', 1)

   # convert count to int
   count = int(count)

   if count>maxcount:
      maxcount=count
      maxcountkey=key
      print '%s%s' % (maxcountkey, maxcount)
```

■ Box 10.23: Oozie workflow for computing machine status/error code with maximum count

```
<workflow-app name="PythonOozieApp" xmlns="uri:oozie:workflow:0.1">
 <start to="streamingaAction"/>
  <action name="streamingaAction">
<map-reduce>
    <job-tracker>${jobTracker}</job-tracker>
    <name-node>${nameNode}</name-node>
    <prepare>
     <delete path="${outputDir}"/>
    </prepare>
    <streaming>
     <mapper>python Mapper.py</mapper>
     <reducer>python Reducer.py</reducer>
    </streaming>
    <configuration>
<property>
<name>oozie.libpath</name>
    <value>${oozieLibPath}/mapreduce-streaming</value>
   </property>
     <property>
      <name>mapred.input.dir</name>
      <value>${inputDir}</value>
     </property>
     <property>
      <name>mapred.output.dir</name>
      <value>${outputDir}</value>
     </property>
<property>
      <name>mapred.reduce.tasks</name>
      <value>1</value>
     </property>
    </configuration>
<file>${appPath}/Mapper.py#Mapper.py</file>
<file>${appPath}/Reducer.py#Reducer.py</file>
  </map-reduce>
    <ok to="streamingaAction2"/>
    <error to="killJobAction"/>
  </action>

<action name="streamingaAction2">
<map-reduce>
    <job-tracker>${jobTracker}</job-tracker>
```

```
    <name-node>${nameNode}</name-node>
    <streaming>
     <mapper>python Mapper1.py</mapper>
     <reducer>python Reducer1.py</reducer>
    </streaming>
    <configuration>
<property>
<name>oozie.libpath</name>
    <value>${oozieLibPath}/mapreduce-streaming</value>
   </property>
     <property>
      <name>mapred.input.dir</name>
      <value>${outputDir}</value>
     </property>
     <property>
      <name>mapred.output.dir</name>
      <value>${outputDir}/output2</value>
     </property>
<property>
      <name>mapred.reduce.tasks</name>
      <value>1</value>
     </property>
    </configuration>
<file>${appPath}/Mapper1.py#Mapper1.py</file>
<file>${appPath}/Reducer1.py#Reducer1.py</file>
  </map-reduce>
    <ok to="end"/>
    <error to="killJobAction"/>
  </action>

  <kill name="killJobAction">
   <message>"Killed job due to error:
${wf:errorMessage(wf:lastErrorNode())}"</message>
  </kill>
 <end name="end" />
</workflow-app>
```

Figure 10.22 shows a screenshot of the Oozie web console which can be used to monitor the status of Oozie workflows.

10.5 Apache Spark

Apache Spark is yet another open source cluster computing framework for data analytics [121]. However, Spark supports in-memory cluster computing and promises to be faster than

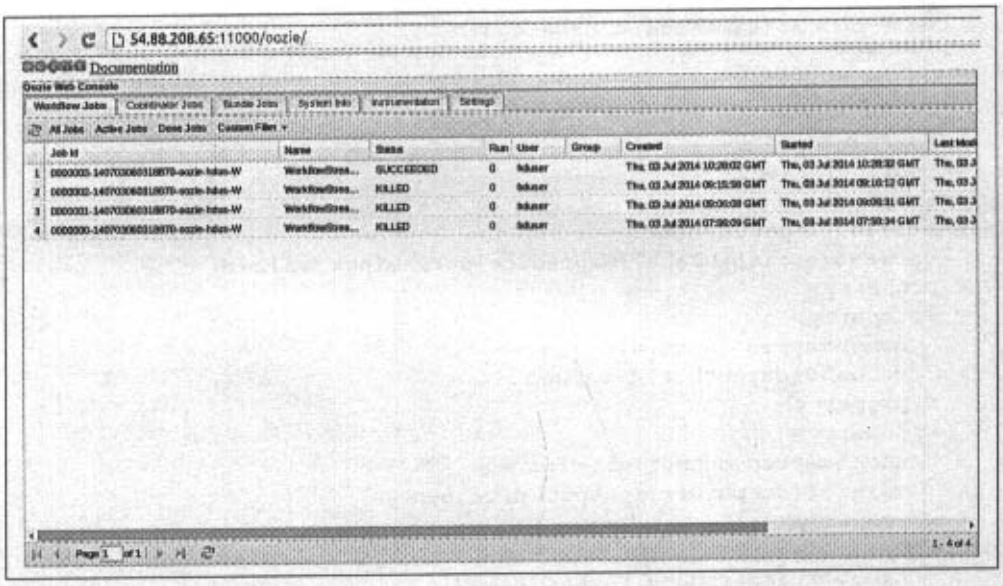

Figure 10.22: Screenshot of Oozie web console

Hadoop. Spark supports various high-level tools for data analysis such as Spark Streaming for streaming jobs, Spark SQL for analysis of structured data, MLlib machine learning library for Spark, GraphX for graph processing and Shark (Hive on Spark). Spark allows real-time, batch and interactive queries and provides APIs for Scala, Java and Python languages.

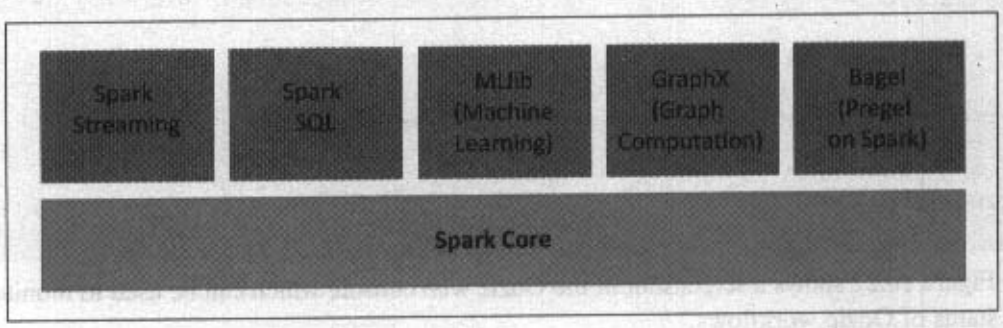

Figure 10.23: Spark tools

Figure 10.24 shows the components of a Spark cluster. Each Spark application consists of a driver program and is coordinated by a *SparkContext* object. Spark supports various cluster managers including Spark's standalone cluster manager, Apache Mesos and Hadoop

YARN. The cluster manager allocates resources for applications on the worker nodes. The executors which are allocated on the worker nodes run the application code as multiple tasks. Applications are isolated from each other and run within their own executor processes on the worker nodes. Spark provides data abstraction called resilient distributed dataset (RDD) which is a collection of elements partitioned across the nodes of the cluster. The RDD elements can be operated on in parallel in the cluster. RDDs support two types of operations - transformations and actions. Transformations are used to create a new dataset from an existing one. Actions return a value to the driver program after running a computation on the dataset. Spark API allows chaining together transformations and actions.

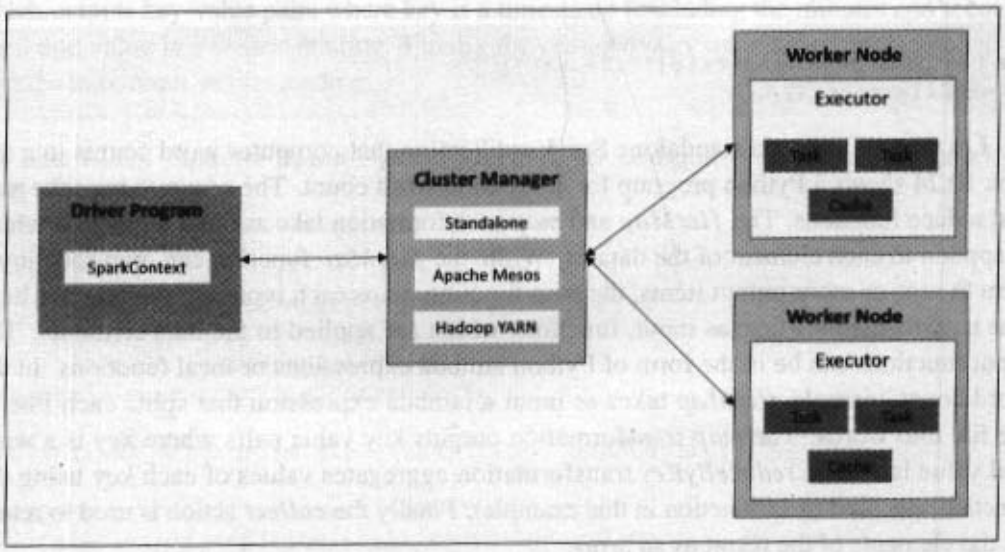

Figure 10.24: Components of a Spark cluster

Spark comes with a spark-ec2 script (in the spark/ec2 directory) which makes it easy to setup Spark cluster on Amazon EC2. With spark-ec2 script you can easily launch, manage and shutdown Spark cluster on Amazon EC2. To start a Spark cluster use the following command:

```
./spark-ec2 -k <keypair> -i <key-file> -s <num-slaves>
launch <cluster-name> -instance-type=<INSTANCE_TYPE>
```

Spark cluster setup on EC2 is configured to use HDFS as its default filesystem. To analyze contents of a file, the file should be first copied to HDFS using the following command:

```
bin/hadoop fs -put file.txt file.txt
```

Spark supports a shell mode with which you can interactively run commands for analyzing data. To launch the Spark Python shell, run the following command:

```
./bin/pyspark
```

When you launch a PySpark shell, a SparkContext is created in the variable called sc. The following commands show how to load a text file and count the number of lines from the PySpark shell.

```
textFile = sc.textFile("file.txt")
textFile.count()
```

Let us now look at a standalone Spark application that computes word counts in a file. Box 10.24 shows a Python program for computing word count. The program uses the map and reduce functions. The *flatMap* and *map* transformation take as input a function which is applied to each element of the dataset. While the *flatMap* function can map each input item to zero or more output items, the *map* function maps each input item to another item. The transformations take as input, functions which are applied to the data elements. The input functions can be in the form of Python lambda expressions or local functions. In the word count example *flatMap* takes as input a lambda expression that splits each line of the file into words. The *map* transformation outputs key value pairs where key is a word and value is 1. The *reduceByKey* transformation aggregates values of each key using the function specified (*add* function in this example). Finally the *collect* action is used to return all the elements of the result as an array.

Box 10.24: Apache Spark Python program for computing word count

```
from operator import add
from pyspark import SparkContext

sc = SparkContext(appName="WordCountApp")
lines = sc.textFile("file.txt")
counts = lines.flatMap(lambda x:
x.split(' ')).map(lambda x:  (x, 1)).reduceByKey(add)

output = counts.collect()
```

```
for (word, count) in output:
   print "%s:  %i" % (word, count)
```

Let us look at another Spark application for batch analysis of data. Taking the example of analysis of forest fire detection sensor data described in the previous section, let us look at a Spark application that aggregates the time-stamped sensor data and finds hourly maximum values for temperature, humidity, light and CO_2. The Python code for the Spark application is shown in Box 10.25. The sensor data is loaded as a text file. Each line of the text file contains time-stamped sensor data. The lines are first split by applying the *map* transformation to access the individual sensor readings. In the next step, a *map* transformation is applied which outputs key-value pairs where key is a timestamp (excluding the minutes and seconds part) and value is a sensor reading. Finally the *reduceByKey* transformation is applied to find the maximum sensor reading.

■ Box 10.25: Apache Spark Python program for computing maximum values for sensor readings

```
#Data format:
#"2014-06-25 10:47:44",26,36,2860,274
from pyspark import SparkContext

sc = SparkContext(appName="MyApp")
textFile = sc.textFile("data.txt")

splitlines = textFile.map(lambda line:  line.split(','))

maxtemp = splitlines.map(lambda line:
(line[0][0:17],int(line[1]))).reduceByKey(lambda a, b:
a if (a > b) else b).collect()

maxhumdity = splitlines.map(lambda line:
(line[0][0:17],int(line[2]))).reduceByKey(lambda a, b:
a if (a > b) else b).collect()

maxlight = splitlines.map(lambda line:
(line[0][0:17],int(line[3]))).reduceByKey(lambda a, b:
a if (a > b) else b).collect()

maxco = splitlines.map(lambda line:
(line[0][0:17],int(line[4]))).reduceByKey(lambda a, b:
a if (a > b) else b).collect()
```

```
print "Maximum temperature"
for item in maxtemp:
   print item

print "Maximum humidity"
for item in maxhumdity:
   print item

print "Maximum light"
for item in maxlight:
   print item

print "Maximum CO2"
for item in maxco:
   print item
```

Box 10.26 shows an example of using Spark for data filtering. This example uses the sensor data from forest fire detection IoT system.

■ Box 10.26: Apache Spark Python program for filtering sensor readings

```
#Data format:
#"2014-06-25 10:47:44",26,36,2860,274

from pyspark import SparkContext

sc = SparkContext(appName="App")
textFile = sc.textFile("data.txt")

splitlines = textFile.map(lambda line:  line.split(','))


splitlines.filter(lambda line:  int(line[1])>10).collect()

splitlines.filter(lambda line:
int(line[1])>20 and int(line[2])>20 and
int(line[3])>6000 and int(line[4])>200).collect()

#Alternative implementation
def filterfunc(line):
   if int(line[1])>20 and int(line[2])>20
and int(line[3])>6000 and int(line[4])>200:
      return line
   else:
```

```
    return ''

splitlines.filter(filterfunc).collect()
```

Spark includes a machine learning library, MLlib, which includes implementations of machine learning algorithms for classification, regression, clustering, collaborative filtering and dimensionality reduction. Let us look at examples of using MLlib for clustering and classifying data.

Box 10.27 shows an example of clustering data with k-means clustering algorithm. In this example, the data is loaded from a text file and then parsed using the *parseVector* function. Next, the KMeans object is used to cluster the data into two clusters.

■ Box 10.27: Apache Spark Python program for clustering data

```
#Data format:
#26.0,36.0,2860.0,274.0

import numpy as np
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans

#Specify number of clusters
k = 2

#Specify input data file
inputfile="data.txt"

def parseVector(line):
   return np.array([float(x) for x in line.split(',')])

sc = SparkContext(appName="KMeans")
lines = sc.textFile(inputfile)

data = lines.map(parseVector)

model = KMeans.train(data, k)
print "Final centers:  " + str(model.clusterCenters)
```

Box 10.28 shows an example of classifying data with Naive Bayes classification algorithm. The training data in this example consists of labeled points where value in the first column is the label. The *parsePoint* function parses the data and creates Spark

LabeledPoint objects. The labeled points are passed to the *NaiveBayes* object for training a model. Finally, the classification is done by passing the test data (as labeled point) to the trained model.

Box 10.28: Apache Spark Python program for classifying data

```
#Data format:
#1.0,26.0,36.0,2860.0,274.0

import numpy as np
from pyspark import SparkContext
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import NaiveBayes


# Parse a line of text into an MLlib LabeledPoint object
def parsePoint(line):
    values = [float(s) for s in line.split(',')]
    return LabeledPoint(values[0], values[1:])


sc = SparkContext(appName="App")
points = sc.textFile("nbdata.txt").map(parsePoint)

# Train a naive Bayes model.
model = NaiveBayes.train(points, 1.0)

# Make prediction.
prediction = model.predict([20.0, 40.0, 1000.0, 300.0])
print "Predition is:  " + str(prediction)
```

10.6 Apache Storm

Apache Storm is a framework for distributed and fault-tolerant real-time computation [134]. Storm can be used for real-time processing of streams of data. Figure 10.25 shows the components of a Storm cluster. A Storm cluster comprises of Nimbus, Supervisor and Zookeeper. Nimbus is similar to Hadoop's JobTracker and is responsible for distributing code around the cluster, launching works across the cluster and monitoring computation. A Storm cluster has one or more Supervisor nodes on which the worker processes run. Supervisor nodes communicate with Nimbus through Zookeeper. Nimbus sends signals to Supervisor to start or stop workers. Zookeeper is a high performance distributed coordination service for

maintaining configuration information, naming, providing distributed synchronization and group services [135]. Zookeeper is required for coordination of the Storm cluster.

A computation job on the Storm cluster is called a "topology" which is a graph of computation. A Storm topology comprises of a number of worker processes that are distributed on the cluster. Each worker process runs a subset of the topology. A topology is composed of Spouts and Bolts. Spout is a source of streams (sequence of tuples), for example, a sensor data stream. The streams emitted by the Spouts are processed by the Bolts. Bolts subscribe to Spouts, consume the streams, process them and emit new streams. A topology can consists of multiple Spouts and Bolts. Figure 10.26 shows a Storm topology with one Spout and three Bolts. Bolts 1 and 2 subscribe to the Spout and consume the streams emitted by the Spout. The outputs of Bolts 1 and 2 are consumed by Bolt-3.

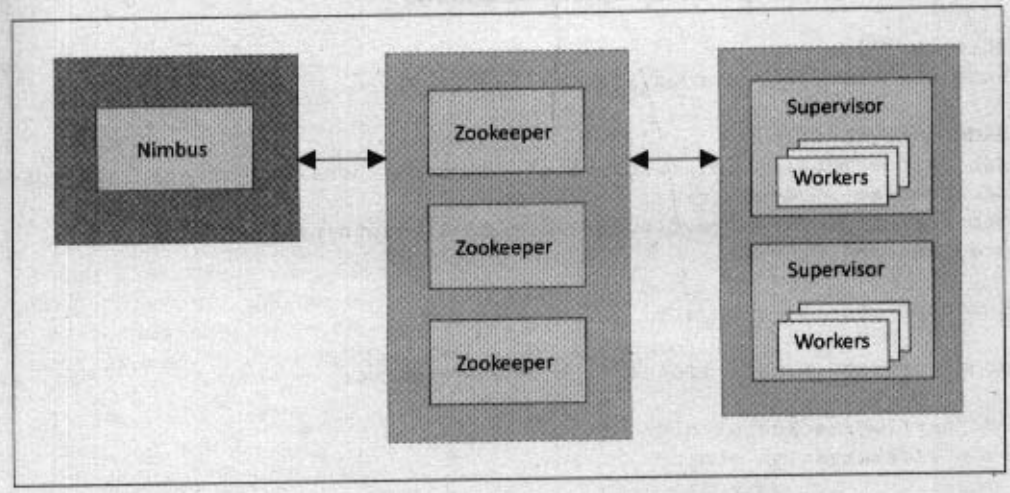

Figure 10.25: Components of a Storm cluster

10.6.1 Setting up a Storm Cluster

In this section you will learn how to setup a Storm cluster. The cluster setup explained in this section comprises of three Ubuntu Linux instances for Nimbus, Zookeeper and Supervisor. Before starting with the installation, make sure you have three instances running and they can connect securely to each other with SSH. Change the hostnames of the instances to "nimbus", "zookeeper" and "supervisor".

On the instance with hostname "zookeeper", setup Zookeeper by following the instructions in Box 10.29.

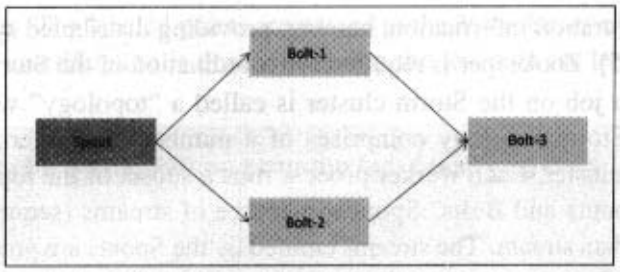

Figure 10.26: Example of a Storm topology

■ Box 10.29: Zookeeper installation instructions

```
#Create file:
sudo vim /etc/apt/sources.list.d/cloudera.list

#Add this to file:
deb [arch=amd64] http://archive.cloudera.com/cdh4/ubuntu/precise/amd64/
cdh precise-cdh4 contrib
deb-src http://archive.cloudera.com/cdh4/ubuntu/precise/amd64/cdh
precise-cdh4 contrib

sudo apt-get -q -y update

sudo apt-get install zookeeper zookeeper-server

cd /usr/lib/zookeeper/bin/
sudo ./zkServer.sh start

#Check if zookeeper is working:
echo ruok | nc zookeeper 2181
echo stat | nc zookeeper 2181
```

On the instances with hostnames "nimbus" and "supervisor", install Storm by following the instructions shown in Box 10.30.

■ Box 10.30: Apache Storm installation instructions

```
#INSTALL REQUIRED PACKAGES
sudo apt-get -q -y install build-essential
```

```
sudo apt-get -q -y install uuid-dev
sudo apt-get -q -y install git
sudo apt-get -q -y install pkg-config libtool autoconf automake
sudo apt-get -q -y install unzip

#-------------
#ZEROMQ INSTALLATION
wget http://download.zeromq.org/zeromq-2.1.7.tar.gz
tar -xzf zeromq-2.1.7.tar.gz
cd zeromq-2.1.7
./configure
make
sudo make install

#------------
#JZMQ INSTALLATION

export JAVA_HOME=/usr/lib/jvm/java-7-oracle

git clone https://github.com/nathanmarz/jzmq.git
cd jzmq
cd src

touch classdist_noinst.stamp
CLASSPATH=.:./.:$CLASSPATH javac -d .  org/zeromq/ZMQ.java
org/zeromq/ZMQException.java org/zeromq/ZMQQueue.java
org/zeromq/ZMQForwarder.java org/zeromq/ZMQStreamer.java

cd ..
./autogen.sh
./configure
make
sudo make install

#-------------
#STORM INSTALLATION

wget https://dl.dropbox.com/u/133901206/storm-0.8.2.zip
unzip storm-0.8.2.zip
sudo ln -s storm-0.8.2 storm

vim .bashrc
PATH=$PATH:"/home/ubuntu/storm"
source .bashrc
```

After installing Storm, edit the configuration file and enter the IP addresses of the Nimbus and Zookeeper nodes as shown in Box 10.31. You can then launch Nimbus and Storm UI. The Storm UI can be viewed in the broswer at the address http://<IP-address-of-Nimubs>:8080. Figures 10.27 and 10.28 show screenshots of the Storm UI. The commands for submitting topologies to Storm are shown in Box 10.31.

■ Box 10.31: Intructions for configuring and running Storm and submitting topologies to Storm

```
#Set storm/config/storm.yaml as follows:
#---------
storm.zookeeper.servers:
- "192.168.1.20" #IP Address of Zookeeper node

nimbus.host:  "nimbus"
- "192.168.1.21" #IP Address of Nimbus node

storm.local.dir:  "/home/ubuntu/stormlocal"
#---------

#Create storm local dir:
cd /home/ubuntu
mkdir stormlocal

#Launch nimbus:
cd storm
bin/storm nimbus

#Launch UI:
bin/storm ui

#Submit topology:
bin/storm jar storm-starter-0.0.1-SNAPSHOT.jar storm.starter.MyTopology
my-topology

#Kill topology
storm -kill my-topology
```

10.7 Using Apache Storm for Real-time Data Analysis

Apache Storm can be used for real-time analysis of data. Figure 10.29 shows workflow for real-time analysis of sensor data using Storm.

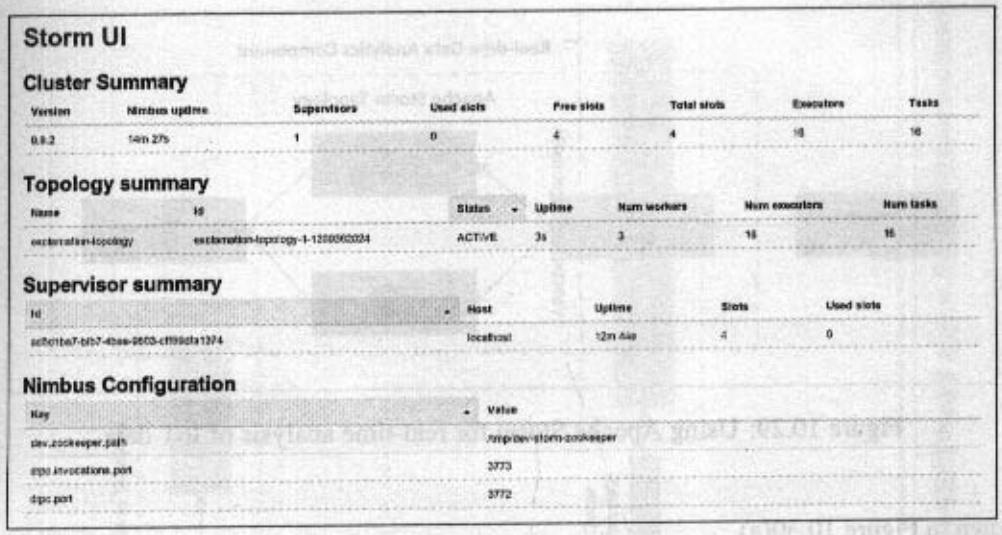

Figure 10.27: Screenshot of Storm UI showing cluster, topology and supervisor summary

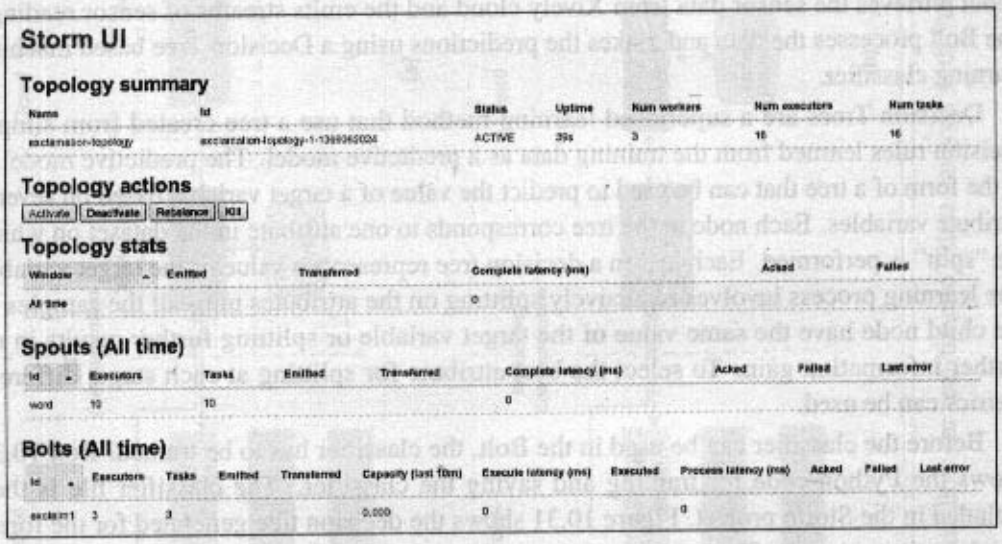

Figure 10.28: Screenshot of Storm UI showing details of a topology

10.7.1 REST-based approach

This section describes an example of real-time sensor data analysis for forest fire detection using a REST-based approach. The deployment design for the WebSocket implementation is

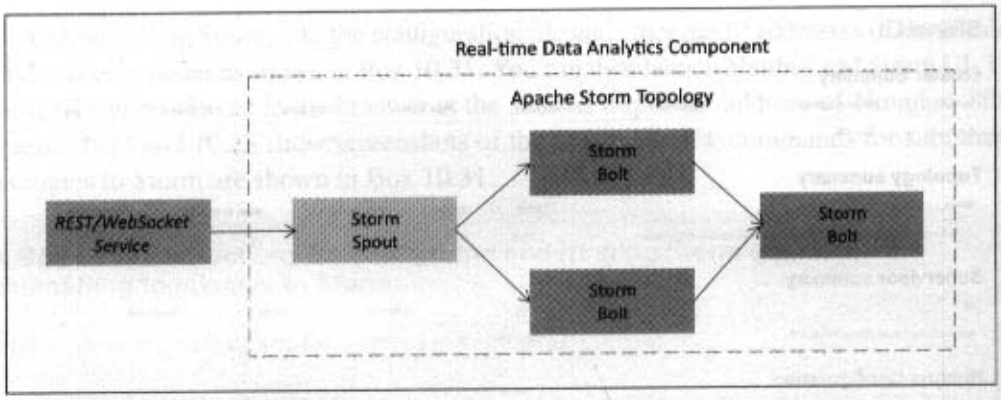

Figure 10.29: Using Apache Storm for real-time analysis of IoT data

shown in Figure 10.30(a).

The Storm topology used in this example comprises of one Spout and one Bolt. The Spout retrieves the sensor data from Xively cloud and the 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.

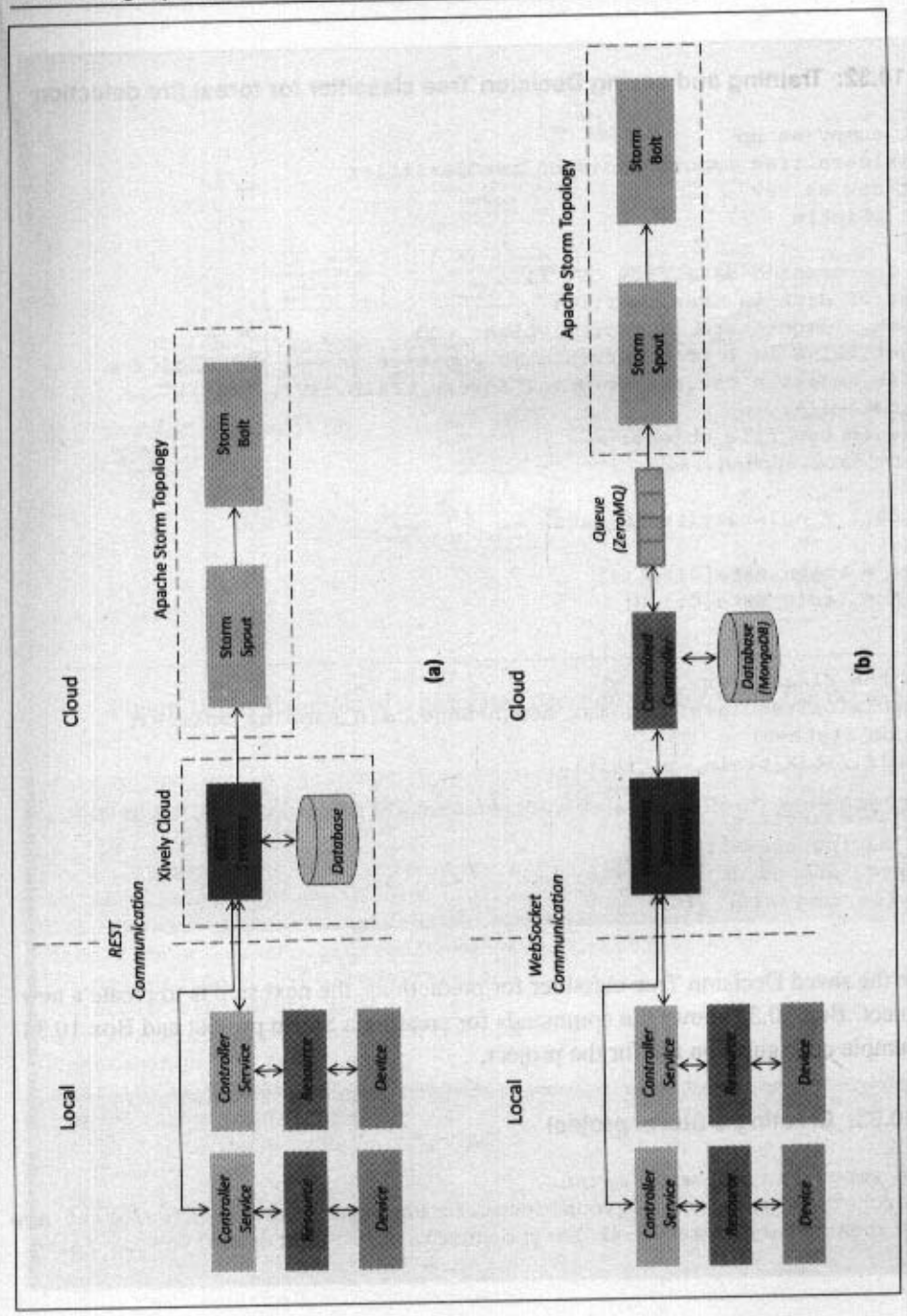

Figure 10.30: Workflow for real-time IoT data analysis with Storm: (a) REST-based approach, (b) WebSocket-based approach

Box 10.32: Training and saving Decision Tree classifier for forest fire detection

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
import csv as csv
import cPickle

#Load the traning data from CSV file
#Format of data in traning file:
# Target, Temperature, Humidity, Light, CO
# Target value is 1 if occurrence is positive and -1 for negative
csv_file_object = csv.reader(open('forest_train.csv', 'rb'))
train_data=[]
for row in csv_file_object:
   train_data.append(row)

train_data = np.array(train_data)

X_train = train_data[0::,1::]
y_train = train_data[0::,0]


#Train the classifier
clf= DecisionTreeClassifier(max_depth=None, min_samples_split=1,
   random_state=0)
clf = clf.fit(X_train, y_train)

#Save the classifier
print "Saving classifier"
with open('dumped_dt_classifier.pkl', 'wb') as fid:
   cPickle.dump(clf, fid)
```

To use the saved Decision Tree classifier for predictions, the next step is to create a new Storm project. Box 10.33 shows the commands for creating a Storm project and Box 10.34 shows a sample configuration file for the project.

Box 10.33: Creating a Storm project

```
#Create empty project with maven:
$ mvn archetype:generate -DgroupId=com.forest.app -DartifactId=forest-app
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

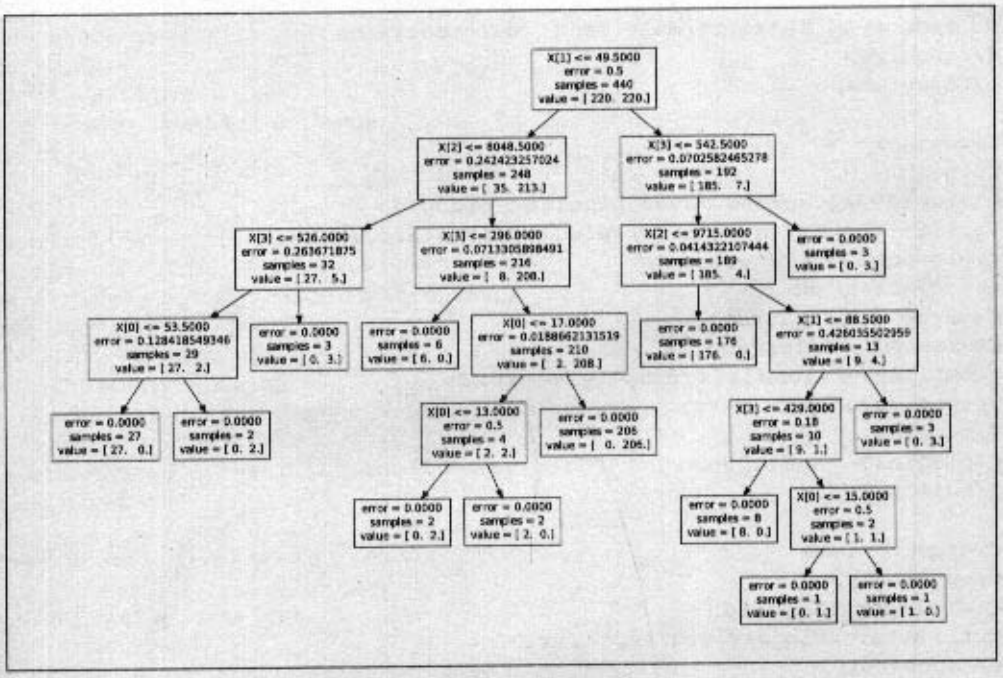

Figure 10.31: Example of a generated decision tree for forest fire detection

Box 10.34: Configuration file for forest fire detection Storm app - pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.forest.app</groupId>
<artifactId>forest-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>forest-app</name>
<url>http://maven.apache.org</url>

<build>
<resources>
<resource>
```

```
from storm import Spout, emit, log
import time
import datetime
from random import randrange
import xively


FEED_ID = "<enter feed ID>"
API_KEY = "<enter API key>"
api = xively.XivelyAPIClient(API_KEY)
feed = api.feeds.get(FEED_ID)

def getData():
   #Format of data - "temperature,humidity,light,CO"
   #e.g.  30,20,6952,10
   data = feed.datastreams.get("forest_data").current_value
   return data

class SensorSpout(Spout):

   def nextTuple(self):
      time.sleep(2)
      data = getData()
      emit([data])


SensorSpout().run()
```

Box 10.36 shows the Python code for the Storm Bolt. This Bolt receives the streams of sensor data emitted by the Spout. The Decision Tree classifier saved earlier is used in the Bolt to make predictions.

■ Box 10.36: Storm bolt for forest fire detection - sensorBolt.py

```
import storm
import numpy as np
import cPickle
from sklearn.tree import DecisionTreeClassifier

with open('dumped_dt_classifier.pkl', 'rb') as fid:
   clf = cPickle.load(fid)

class SensorBolt(storm.BasicBolt):
```

```
    def process(self, tup):
        data = tup.values[0].split(',')

    test_data=[]
    test_data.append(data)
    test_data = np.array(test_data)
    X_test = test_data[0::,0::]

    output = clf.predict(X_test)
    result= "Predicted:  "+ str(output)

    storm.emit([result])


SensorBolt().run()
```

After the Spout and Bolt programs are created the next step is to create a topology. Box 10.37 shows the Java program for creating a topology. To create a topology an object of the *TopologyBuilder* class is created. The Spout and Bolt are defined using the *setSpout* and *setBolt* methods. These methods take as input a user-specified id, objects to the Spout/Bolt classes, and the amount of parallelism required. Storm has two modes of operation - local and distributed. In the local mode, Storm simulates worker nodes within a local process. The distributed mode runs on the Storm cluster. The program in Box 10.37 shows the code for submitting topology to both local and distributed modes.

With all the project files created, the final step is to build and run the project. Box 10.38 shows the commands for building and running a Storm project.

▪ Box 10.37: Java Program for creating Storm topology for forest fire detection - app.java

```
package com.forest.app;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.task.ShellBolt;
import backtype.storm.spout.ShellSpout;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.IRichSpout;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
```

```
import backtype.storm.tuple.Fields;
import java.util.Map;


public class App {

public static class SensorSpout extends ShellSpout implements IRichSpout
{
   public SensorSpout() {
      super("python", "sensorSpout.py");
   }

   @Override
   public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("sensordata"));
   }

   @Override
   public Map<String, Object> getComponentConfiguration() {
      return null;
   }
}

public static class SensorBolt extends ShellBolt implements IRichBolt {
   public SensorBolt() {
      super("python", "sensorBolt.py");
   }

   @Override
   public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("sensordata"));
   }

   @Override
   public Map<String, Object> getComponentConfiguration() {
      return null;
   }
}


public static void main(String[] args) throws Exception {

   TopologyBuilder builder = new TopologyBuilder();

   builder.setSpout("spout", new SensorSpout(), 5);
```

```
    builder.setBolt("analysis",
    new SensorBolt(), 8).shuffleGrouping("spout");

    Config conf = new Config();
    conf.setDebug(true);

    if (args != null && args.length > 0) {
       conf.setNumWorkers(3);
       StormSubmitter.submitTopology(args[0], conf,
       builder.createTopology());
    }
    else {
       conf.setMaxTaskParallelism(3);

       LocalCluster cluster = new LocalCluster();
       cluster.submitTopology("forest-fire-detect", conf,
       builder.createTopology());

       Thread.sleep(10000);

       cluster.shutdown();
}
```

■ Box 10.38: Building and running a Storm project

```
#Create empty project with maven:
$ mvn archetype:generate -DgroupId=com.forest.app
-DartifactId=forest-app -DarchetypeArtifactId=
maven-archetype-quickstart -DinteractiveMode=false

#Build the Project
$cd myproject
$mvn package

#Run the Storm Project:
$mvn exec:java -Dexec.mainClass="com.forest.app.App"
```

10.7.2 WebSocket-based approach

The previous section described a REST-based implementation of the forest fire detection system. In this section you will learn about an alternative implementation of the IoT system based on the WebSocket approach. The WebSocket implementation is based on the Web

Application Messaging Protocol (WAMP) which is a sub-protocol of WebSocket. You learned about AutoBahn, an open source implementation of WAMP in Chapter 8. The deployment design for the WebSocket implementation is shown in Figure 10.30(b).

Box 10.39 shows the implementation of the native controller service that runs on the Raspberry Pi device. 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 Subscriber component is a part of the cloud-based centralized controller, the source code for which is shown in Box 10.40. 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. Boxes 10.41 and 10.42 show the implementations of the Storm Spout and Bolt for real-time analysis of data. The Storm Bolt uses a Decision Tree classifier for making the predictions.

Box 10.39: Forest fire detection controller service

```
import time
import datetime
import dhtreader
import spidev
from random import randint
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep
from autobahn.twisted.wamp import ApplicationSession

#Configure these pin numbers for DHT22
DEV_TYPE = 22
DHT_PIN = 24

#Initialize DHT22
dhtreader.init()

#LDR channel on MCP3008
LIGHT_CHANNEL = 0

# Open SPI bus
spi = spidev.SpiDev()
spi.open(0,0)
```

```
#MICS5525 CO sensor channel on MCP3008
CO_CHANNEL = 1

#Conversions based on Rs/Ro vs ppm plot of MICS5525 CO sensor
CO_Conversions = [((0,100),(0,0.25)),((100,133),(0.25,0.325)),
      ((133,167),(0.325,0.475)),((167,200),(0.475, 0.575)),
      ((200,233),(0.575,0.665)),((233,267),(0.666,0.75)) ]

#Function to read temperature & humidity from DHT22 sensor
def read_DHT22_Sensor():
   temperature, humidity = dhtreader.read(DEV_TYPE, DHT_PIN)
   return temperature, humidity

#Function to read LDR connected to MCP3008
def readLDR():
   light_level = ReadChannel(LIGHT_CHANNEL)
   lux = ConvertLux(light_level,2)
   return lux

#Function to convert LDR reading to Lux
def ConvertLux(data,places):
   R=10 #10k-ohm resistor connected to LDR
   volts = (data * 3.3) / 1023
   volts = round(volts,places)
   lux=500*(3.3-volts)/(R*volts)
   return lux

# Function to read SPI data from MCP3008 chip
def ReadChannel(channel):
   adc = spi.xfer2([1,(8+channel)<<4,0])
   data = ((adc[1]&3) << 8) + adc[2]
   return data

#Function to read MICS5525 CO sensor connected to MCP3008
def readCOSensor():
   result = ReadChannel(CO_CHANNEL)
   if result == 0:
      resistance = 0
   else:
      resistance = (vin/result - 1)*pullup
   ppmresult = converttoppm(resistance, CO_Conversions)
   return ppmresult

#Function to convert resistance reading to PPM
```

```
def converttoppm(rs,conversions):
   rsper = 100*(float(rs)/r0)
   for a in conversions:
      if a[0][0]>=rsper>a[0][1]:
         mid,hi = rsper-a[0][0],a[0][1]-a[0][0]
         sf = float(mid)/hi
         ppm = sf * (a[1][1]-a[1][0]) + a[1][0]
         return ppm
   return 0


#Controller main function
def runController():
   temperature, humidity=read_DHT22_Sensor()
   light=readLDR()
   CO_reading = readCOSensor()
   timestamp = datetime.datetime.fromtimestamp(time.time()).strftime(
'%Y-%m-%d%H:%M:%S')

   datalist = [timestamp, temperature, humidity, light, CO_reading]
   return datalist

#An application component that publishes an event every second.
class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      while True:
         datalist = runcontroller()
         self.publish('com.myapp.topic1', datalist)
         yield sleep(1)
```

■ Box 10.40: Centralized controller for forest fire detection

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession
import zmq
from pymongo import MongoClient
import time
import datetime
```

```
client = MongoClient()
client = MongoClient('localhost', 27017)

db = client['mydb']

collection = db['iotcollection']

# ZeroMQ Context
context = zmq.Context()

# Define the socket using the "Context"
sock = context.socket(zmq.PUSH)
sock.bind("tcp://127.0.0.1:5690")


class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      self.received = 0

   def on_event(data):
      #Inset data in MongoDB
      post={'timestamp': data[0], 'temperature':
      data[1], 'humidity': data[2], 'light': data[3], 'CO': data[4]}
      postid=db.collection.insert(post)

      #Send Data to ZMQ queue for further processing
      sock.send(data)

   #Subscribe to Topic
   yield self.subscribe(on_event, 'com.myapp.topic1')


   def onDisconnect(self):
      reactor.stop()
```

■ Box 10.41: Storm Spout for forest fire detection

```
from storm import Spout, emit, log
import time
import datetime
import zmq
```

```
# ZeroMQ Context
context = zmq.Context()
sock = context.socket(zmq.PULL)
sock.connect("tcp://127.0.0.1:5690")


class SensorSpout(Spout):

   def nextTuple(self):
      time.sleep(2)
      data = sock.recv()
      emit([data])


SensorSpout().run()
```

Box 10.42: Storm Bolt for forest fire detection

```
import storm
import numpy as np
import cPickle
from sklearn.tree import DecisionTreeClassifier

with open('dumped_dt_classifier.pkl', 'rb') as fid:
clf = cPickle.load(fid)

class SensorBolt(storm.BasicBolt):
def process(self, tup):
data = tup.values[0].split(',')

   data_without_timestamp = data[1:]

   test_data=[]
   test_data.append(data_without_timestamp)
   test_data = np.array(test_data)
   X_test = test_data[0::,0::]

   output = clf.predict(X_test)
   result= "Predicted at timestamp:  "+ data[0] + " result:  " +
   str(output)
```

```
    storm.emit([result])

SensorBolt().run()
```

10.8 Structural Health Monitoring Case Study

Structural Health Monitoring (SHM) systems use a network of sensors to monitor the vibration levels in the structures such as bridges and buildings. The data collected from these sensors is analyzed to assess the health of the structures.

This section provides a case study of an SHM system that uses 3-axis accelerometer sensors for measuring the vibrations in a structure. The accelerometer data is collected and analyzed in the cloud. The deployment design for the WebSocket implementation is shown in Figure 10.30(b). Figure 10.32 shows a schematic of the IoT device for monitoring vibrations in a structure, comprising of Raspberry Pi board and ADXL345 accelerometer module.

Discrete Fourier Transform (DFT) is useful for converting a sampled signal from time domain to frequency domain which makes the analysis of the signal easier. However, for streaming vibration data in which the spectral content changes over time, using DFT cannot reveal the transitions in the spectral content. Short Time Fourier Transform (STFT) is better suited for revealing the changes in the spectral content corresponding to the SHM data. To compute the STFT, windowed sections of the signal are first generated using a window function and then the Fourier Transform of each windowed section is computed.

The STFT of a signal $x[n]$ is given as

$$X[n,\omega) = \sum_{m=-\infty}^{+\infty} x[m] * w[n-m]e^{-j\omega n} \tag{10.1}$$

where $w[n]$ is a window function. Commonly used window functions are Hann and Hamming windows.

Alternatively, STFT can be interpreted as a filtering operation as follows,

$$X[n,k] = e^{-j\frac{2\pi}{N}kn}(x[n] * w[n]e^{-j\frac{2\pi}{N}kn}) \tag{10.2}$$

From STFT, the spectrogram of the signal can be computed which is useful for visualizing the spectrum of frequencies in the signal and their variations with time.

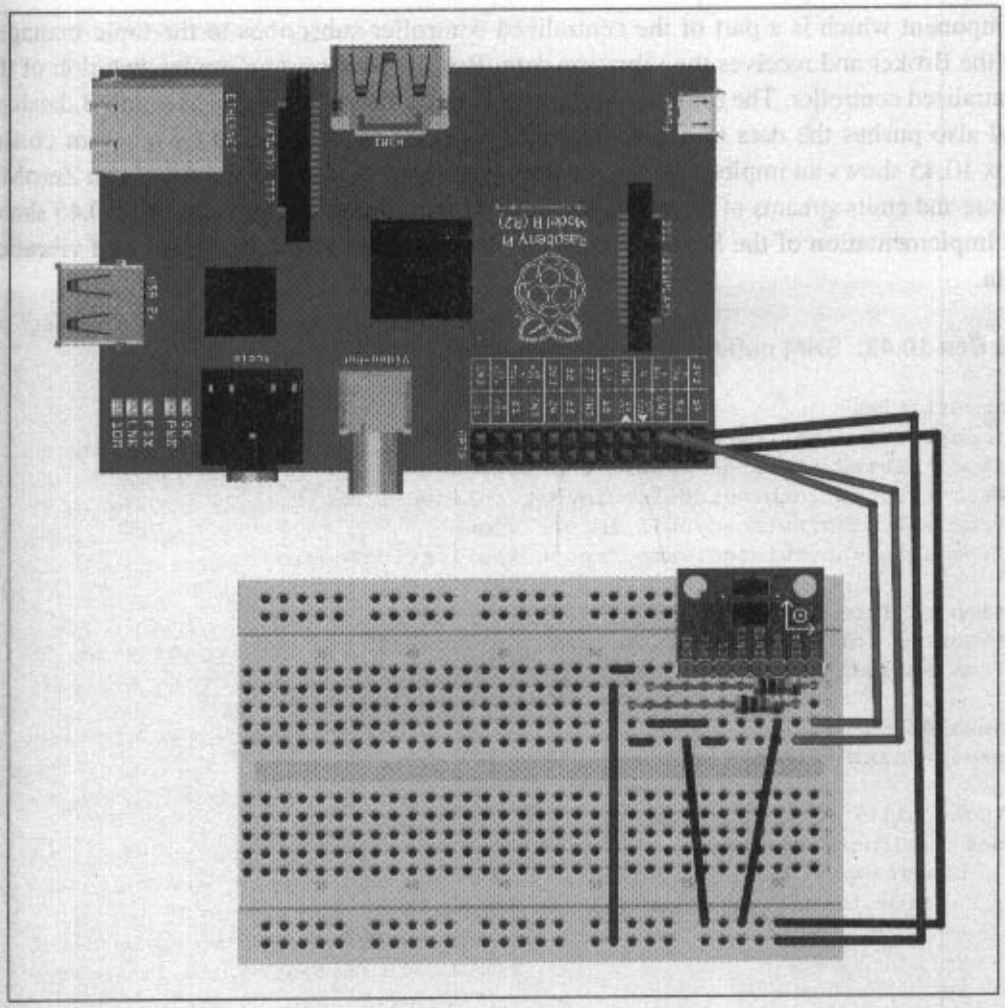

Figure 10.32: Schematic diagram of IoT device for structural health monitoring

$$Spectogram = |X[n,k]|^2 \tag{10.3}$$

Box 10.43 shows the implementation of the native controller service for the SHM system. The controller comprises of the WAMP Publisher application which publishes the 3-axis accelerometer data to a topic managed by the WAMP Broker. The WAMP Subscriber

component which is a part of the centralized controller subscribes to the topic managed by the Broker and receives the vibration data. Box 10.44 shows an implementation of the centralized controller. The centralized controller stores the vibration in a MongoDB database and also pushes the data to a ZeroMQ queue. The data is analyzed by a Storm cluster. Box 10.45 shows an implementation of the Storm Spout that pulls the data from ZeroMQ queue and emits streams of tuples which are consumed by the Storm Bolt. Box 10.46 shows an implementation of the Storm Bolt which computes the STFT for streams of vibration data.

■ Box 10.43: SHM native controller service

```
import time
import datetime
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.util import sleep
from autobahn.twisted.wamp import ApplicationSession

#Import from ADXL345
#Source:  https://github.com/pimoroni/adxl345-python
from adxl345 import ADXL345

adxl345 = ADXL345()
axes = adxl345.getAxes(True)

#Controller main function
def runController():
   timestamp = datetime.datetime.fromtimestamp(
      time.time()).strftime('%Y-%m-%d %H:%M:%S')
   x=[]
   y=[]
   z=[]

   for i in range(1000):
      x.append(axes['x'])
      y.append(axes['y'])
      z.append(axes['z'])

   datalist = [timestamp, X, Y, Z]
   return datalist

#An application component that publishes an event every second.
```

```
class Component(ApplicationSession):
  @inlineCallbacks
  def onJoin(self, details):
     while True:
        datalist = runcontroller()
        self.publish('com.myapp.topic1', datalist)
        yield sleep(1)
```

Box 10.44: SHM centralized controller

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession
import zmq
from pymongo import MongoClient
import time
import datetime


client = MongoClient()
client = MongoClient('localhost', 27017)

db = client['mydb']

collection = db['iotcollection']

# ZeroMQ Context
context = zmq.Context()

# Define the socket using the "Context"
sock = context.socket(zmq.PUSH)
sock.bind("tcp://127.0.0.1:5690")


class Component(ApplicationSession):
   @inlineCallbacks
   def onJoin(self, details):
      self.received = 0

   def on_event(data):
      #Inset data in MongoDB
      post={'timestamp':  data[0],
```

```
        'X': data[1], 'Y': data[2], 'Z': data[3]}
        postid=db.collection.insert(post)

        #Send Data to ZMQ queue for further processing
        sock.send(data)

    #Subscribe to Topic
    yield self.subscribe(on_event, 'com.myapp.topic1')


    def onDisconnect(self):
        reactor.stop()
```

■ Box 10.45: Storm Spout for SHM

```
from storm import Spout, emit, log
import time
import datetime
import zmq

# ZeroMQ Context
context = zmq.Context()
sock = context.socket(zmq.PULL)
sock.connect("tcp://127.0.0.1:5690")


class SensorSpout(Spout):

    def nextTuple(self):
        time.sleep(2)
        data = sock.recv()
        emit([data])

SensorSpout().run()
```

■ Box 10.46: Storm Bolt for SHM

```
import storm
import scipy
```

```
#Short Time Fourier Transform
def stft(d):
"""
x - signal
fs - sample rate
framesize - frame size
hop - hop size (frame size = overlap + hop size)
"""

   #Define analysis window
   framesamp = 200

   #Define the amount of overlap between windows
   hopsamp = 150

   #Define a windowing function
   w = scipy.hamming(framesamp)

   #Generate windowed segments and apply
   #the FFT to each windowed segment
   D = scipy.array([scipy.fft(w*x[i:i+framesamp])
      for i in range(0, len(x)-framesamp, hopsamp)])
   return D


class SensorBolt(storm.BasicBolt):
   def process(self, tup):
      data = tup.values[0].split(',')

      x = data[1]
      y = data[2]
      z = data[3]

      X=stft(x)
      Y=stft(y)
      Z=stft(z)

      output = scipy.absolute(X)
      result= "STFT - X : "+ str(output)

      output = scipy.absolute(Y)
      result= result + "STFT - Y : "+ str(output)

      output = scipy.absolute(Z)
      result= result + "STFT - Z : "+ str(output)
```

```
        storm.emit([result])

SensorBolt().run()
```

Summary

In this chapter you learned about various tools for analyzing IoT data. IoT systems can have varied data analysis requirements. For some IoT systems, the volume of data is so huge that analyzing the data on a single machine is not possible. For such systems, distributed batch data analytics frameworks such as Apache Hadoop can be used for data analysis. For IoT systems which have real-time data analysis requirements, tools such as Apache Storm are useful. For IoT systems which require interactive querying of data, tools such as Apache Spark can be used. Hadoop is an open source framework for distributed batch processing of massive scale data. Hadoop MapReduce provides a data processing model and an execution environment for MapReduce jobs for large scale data processing. Key processes of Hadoop include NameNode, Secondary NameNode, JobTracker, TaskTracker and DataNode. NameNode keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. Secondary NameNode creates checkpoints of the namespace. JobTracker distributes MapReduce tasks to specific nodes in the cluster. TaskTracker accepts Map, Reduce and Shuffle tasks from the JobTracker. DataNode stores data in an HDFS file system. You learned how to setup a Hadoop cluster and run MapReduce jobs on the cluster. You learned about the next generation architecture of Hadoop called YARN. YARN is framework for job scheduling and cluster resource management. Key components of YARN include Resource Manager, Application Master, Node Manager and Containers. You learned about the Oozie workflow scheduler system that allows managing Hadoop jobs. You learned about Apache Spark in-memory cluster computing framework. Spark supports various high-level tools for data analysis such as Spark Streaming for streaming jobs, Spark SQL for analysis of structured data, MLlib machine learning library for Spark, GraphX for graph processing and Shark (Hive on Spark). Finally, you learned about Apache Storm which is a framework for distributed and fault-tolerant real-time computation.

Lab Exercises

1. In this exercise you will create a multi-node Hadoop cluster on a cloud. Follow the steps below:
    - Create and Amazon Web Services account.

- From Amazon EC2 console launch two *m1.small* EC2 instances.
- When the instances start running, note the public DNS addresses of the instances.
- Connect to the instances using SSH.
- Run the commands given in Box 10.2 to install Java on each instance.
- Run the commands given in Box 10.3 to install Hadoop on each instance.
- Configure Hadoop. Use the templates for core-site.xml, hdfs-site.xml, mapred-site.xml and master and slave files shown in Boxes 10.4 - 10.7.
- Start the Hadoop cluster using the commands shown in Box 10.8.
- In a browser open the Hadoop cluster status pages:
  public-DNS-of-hadoop-master:50070
  public-DNS-of-hadoop-master:50030

2. In this exercise you will run a MapReduce job on a Hadoop cluster for aggregating data (computing mean, maximum and minimum) on various timescales. Follow the steps below:
   - Generate synthetic data using the following Python program:

```
#Synthetic data generate for forest fire detection system
#Data Format: "2014-06-25 10:47:44",26,36,2860,274

from random import randrange
import time
import datetime

fp= open('forestdata.txt','w')
readings=100

for j in range(0,readings):
    timestamp = datetime.datetime.fromtimestamp(time.time()).
strftime('%Y-%m-%d %H:%M:%S')
    data = timestamp + "," + str(randrange(0,100)) +
"," + str(randrange(0,100))+ "," + str(randrange(0,10000))+
"," + str(randrange(200,400))
    fp.write(data)

fp.close()
```

- Follow the steps in Exercise-1 to create a Hadoop cluster.
- Copy the synthetic data file to the Hadoop master instance. Use scp or copy or wget to download files. Copy the data file to a folder named 'data'.
- Copy the synthetic data file from the Hadoop master node local filesystem to HDFS:
  bin/hadoop dfs -copyFromLocal data/ input
- Create mapper and reducer Python programs as shown in Boxes 10.9 and 10.10.
- Run MapReduce job using the commands given in Box 10.11.

3. Box 10.32 shows the Python code for training and saving Decision Tree classifier for forest fire detection. Modify the code to train and save a Random Forest classifier. Use the classifier in the Storm bolt for forest fire detection.
4. This exercise is about analyzing weather monitoring data using Apache Storm. For the REST and WebSocket implementations of Weather Monitoring system described in Chapter-9, design a Storm topology (including Spout and Bolt) for predicting the current conditions from the weather data collected. The Storm topology should analyze the weather data (temperature, humidity, pressure and light data) in real-time and classify the current conditions to be one of the following - sunny, warm, hot, mild, cool, chilly, cold, freezing, humid, dry. Follow the steps below:
   - Save the weather monitoring data to a text or CSV file and manually classify and label the data (50-100 rows). For example:
     #Format of labeled file:
     #Label, Timestamp, Temperature, Humidity, Pressure, Light
     Hot, 2014-06-25 10:47:44,38,56,102997,2000
   - Using the labeled data, train and save the classifier. Try Decision Tree and Random Forest classifiers. Use a program similar to the one shown in Box 10.32.
   - Create a Storm project as shown in Box 10.33.
   - Implement Spout and Bolt similar to the implementations shown in Boxes 10.35 and 10.36.
   - Create a Storm topology with the Spout and Bolt created in the previous step using an implementation similar to the one shown in Box 10.37.
   - Build and run the Storm project using the commands shown in Box 10.38.