

Module-5

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

Textbook: Ch.10

5.1 Introduction

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

5.2 Apache Hadoop

Apache Hadoop is an open source framework for distributed batch processing of big data.

Map Reduce is parallel programming model suitable for analysis of big data.

Map Reduce algorithms allow large scale computations to be parallelized across a large cluster of servers.

5.2.1 Map Reduce Programming Model

Map Reduce is a widely used parallel data processing model for processing and analysis of large scale data. 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. Map Reduce programming model moves the computation to where the data resides thus decreasing the transmission of data and improving efficiency.

Map Reduce programming model is well suited for parallel processing of large scale data in which the data analysis tasks can be accomplished by independent map and reduce operations. Map Reduce programs take a set of input key-value pairs and produce a set of output key-value pairs. Figure 5.1 shows the flow of data for a MapReduce job.

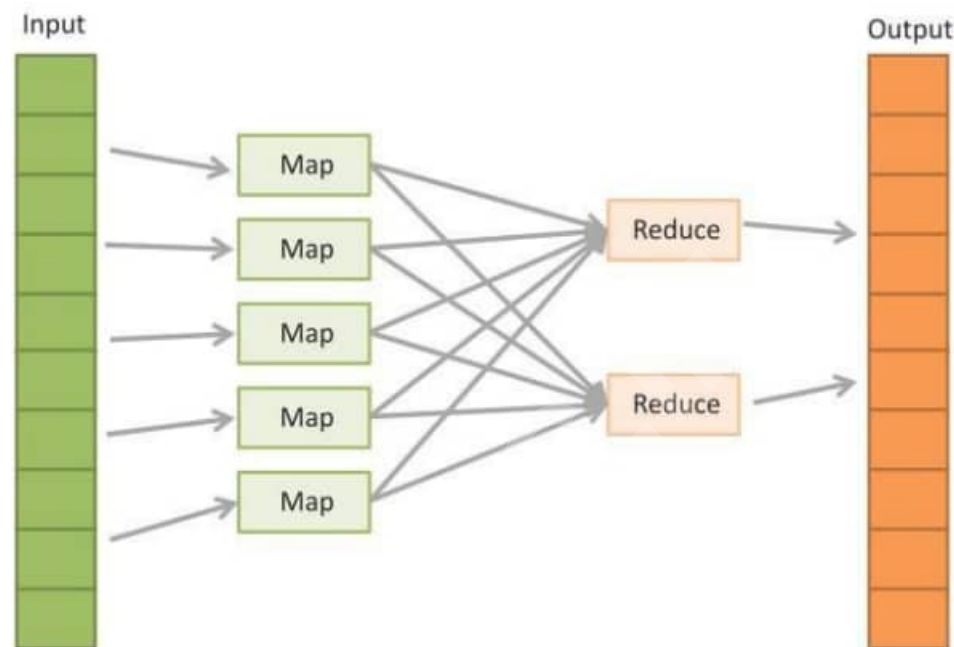


Figure 5.1: Data flow in Map Reduce

Map Reduce model has two phases: Map and Reduce.

- **Map:** In the Map phase, data is read from a distributed file system and partitioned among a set of computing nodes in the cluster. The data is 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.
- **Reduce:** When all the Map tasks are completed, the Reduce phase begins in which the intermediate data with the same key is aggregated.
- The **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.

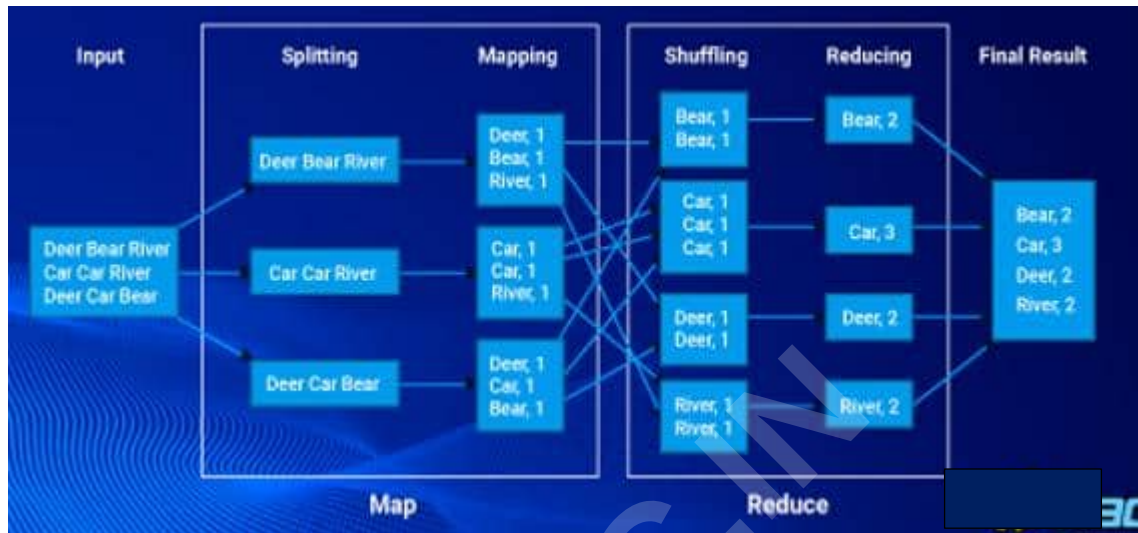


Figure 5.2: Understanding Map Reduce in Hadoop

5.2.2 Hadoop Map Reduce Job Execution:

the Map Reduce job execution workflow explains job submission, job initialization, task selection and task execution. Figure 5.3 shows the components of a Hadoop cluster.

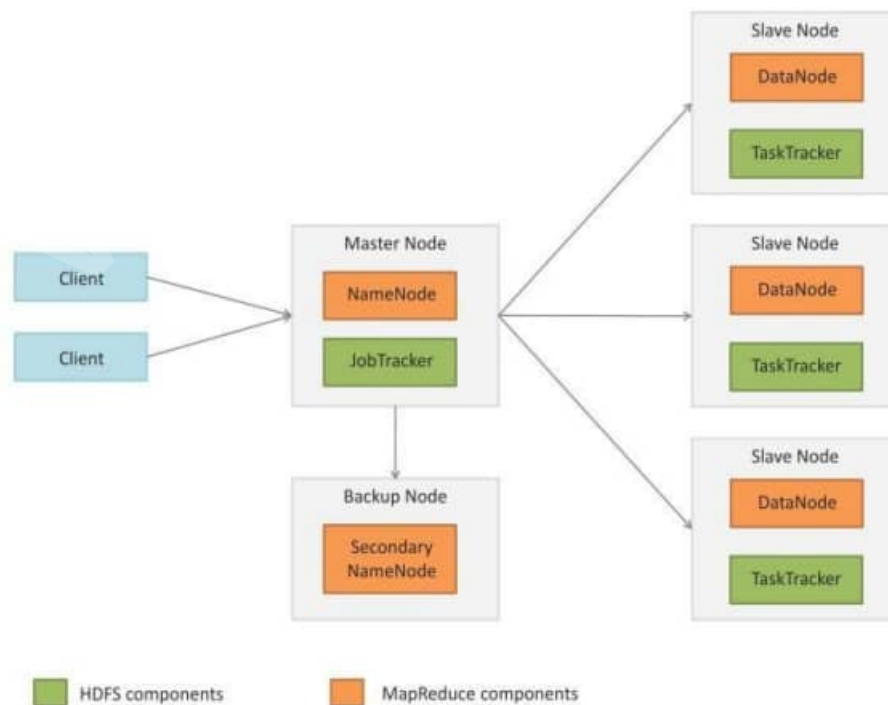


Figure 5.3: Components of a Hadoop cluster

- 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.
- 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
 - 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 respond to requests from the NameNode for filesystem operations.
 - Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data.
 - Similarly, MapReduce operations assigned to TaskTracker instances near a DataNode, talk directly to the DataNode to access the files.
 - TaskTracker instances can be deployed on the same servers that host DataNode instances, so that MapReduce operations are performed close to the data.

5.2.3 Map Reduce Job Execution Workflow

Figure 5.4 shows the MapReduce job execution workflow for Hadoop MapReduce framework.

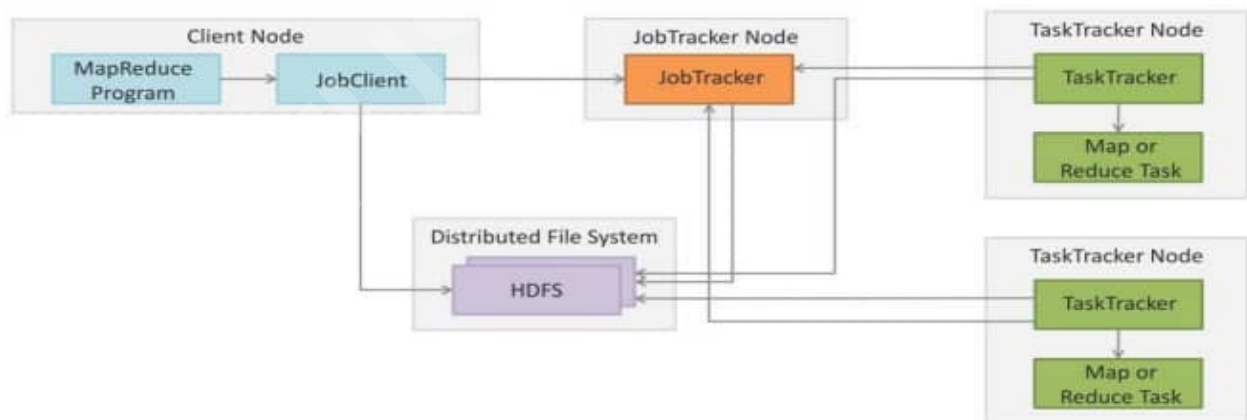


Figure 5.4: Hadoop MapReduce job execution

- MapReduce job execution starts when the client applications submit jobs to the Job tracker.
- The JobTracker returns a JobID to the client application. The JobTracker talks to the NameNode to determine the location of the data.
- The JobTracker locates TaskTracker nodes with available slots at/or near the data.
- The TaskTrackers send out heartbeat messages to the JobTracker, usually every few minutes, to reassure the JobTracker that they are still alive.

- These messages also inform the JobTracker of the number of available slots, so the JobTracker can stay up to date with where in the cluster, new work can be delegated.
- The JobTracker submits the work to the TaskTracker nodes when they poll for tasks. To choose a task for a TaskTracker, the JobTracker uses various scheduling algorithms (default is FIFO).
- 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 finishes, successfully or not, the TaskTracker notifies the JobTracker.
- When a task fails the Task Tracker 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 Task Tracker 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.

5.2.4 Hadoop Cluster Setup

- The Hadoop open source framework is written in Java and has been designed to work with commodity hardware.
 - The Hadoop file system HDFS is highly fault-tolerant.
 - The preferred operating system to host Hadoop is Linux, it can also be set up on Windows-like operating systems with a Cygwin environment.
 - The steps involved in setting up a Hadoop cluster are described as follows:
1. Install Java: Hadoop requires Java 6 or later version.

2. Install Hadoop : To setup a Hadoop cluster, the Hadoop setup tarball is downloaded and unpacked on all the nodes.

3. Networking:

- Configure the network such that all the nodes can connect to each other over the network.
- 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.

4. Configure Hadoop: Hadoop is configured using a number of configuration files listed in Table 5.1.

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

Table5.1: Hadoop configuration files

5.3 Using Hadoop Map Reduce for Batch Data Analysis:

Figure 5.5 shows a Hadoop MapReduce workflow for batch analysis of IoT data.

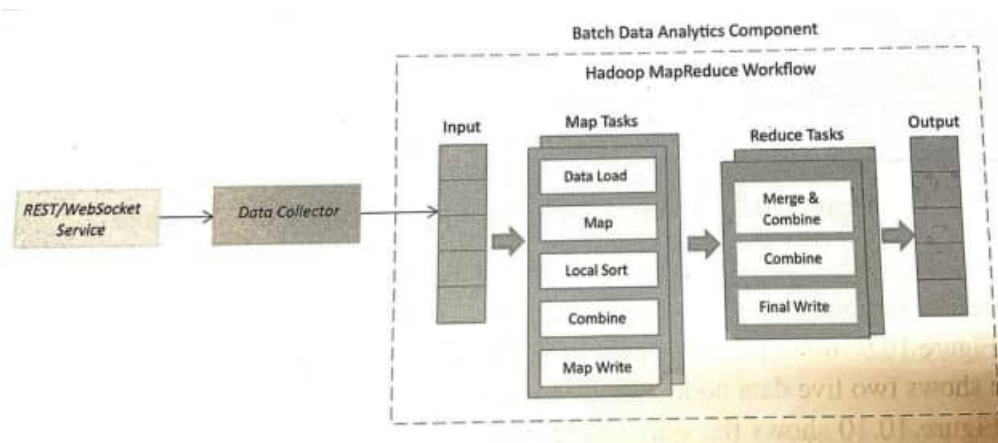


Figure 5.5: Using Hadoop MapReduce for batch analysis of IoT data

- 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
- 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 and the value is a comma separated string of sensor readings.
- 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 perform the computation.

5.3.1 Hadoop YARN

Hadoop YARN is the next generation architecture of Hadoop (version 2.x). In the YARN architecture, the original processing engine of Hadoop (MapReduce) has been separated from the resource management (which is now part of YARN) as shown in Figure 5.6. 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 for interactive queries, Apache Storm for stream processing, etc.

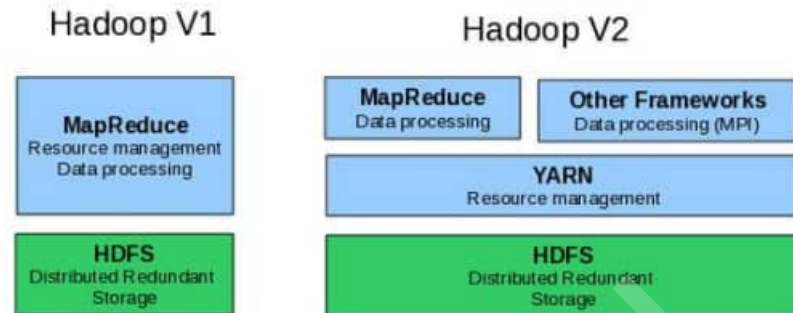


Figure 5.6: Comparison of Hadoop 1.x and 2.x architectures

- In Hadoop 2.0 the original processing engine of Hadoop (MapReduce) has been separated from the resource management (which is now part of YARN).
- 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 for interactive queries, Apache Storm for stream processing, etc.
- Figure 5.7 shows the MapReduce job execution workflow for next generation Hadoop MapReduce framework (MR2).

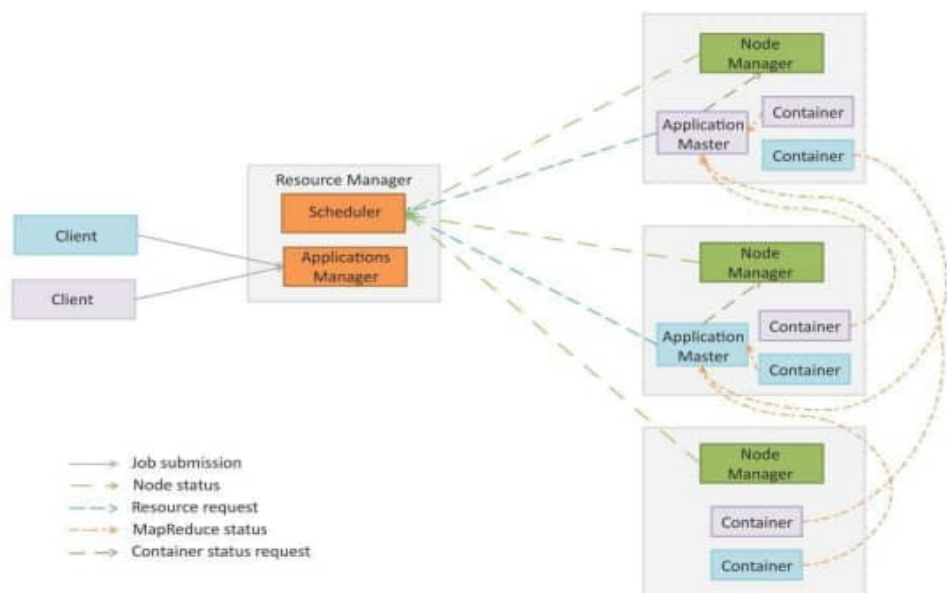


Figure 5.7: Hadoop MapReduce Next Generation (YARN) job execution

- YARN architecture divides the two major functions of the JobTracker - resource management and job life-cycle management - into separate components:
 - ResourceManager(RM).
 - ApplicationMaster.(AM).

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.

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.

- Figure 5.7 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.
- Each container in YARN can be used for both map and reduce tasks.
- The resource allocation model of YARN is more flexible with introduction of resource containers which improve cluster utilization.

Interaction between Client and Resource Manager in YARN:

Figure 5.8 shows the interactions between a Client and Resource Manager.

1. Job execution begins with the submission of a new application request by the client to the RM.
2. The RM 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.

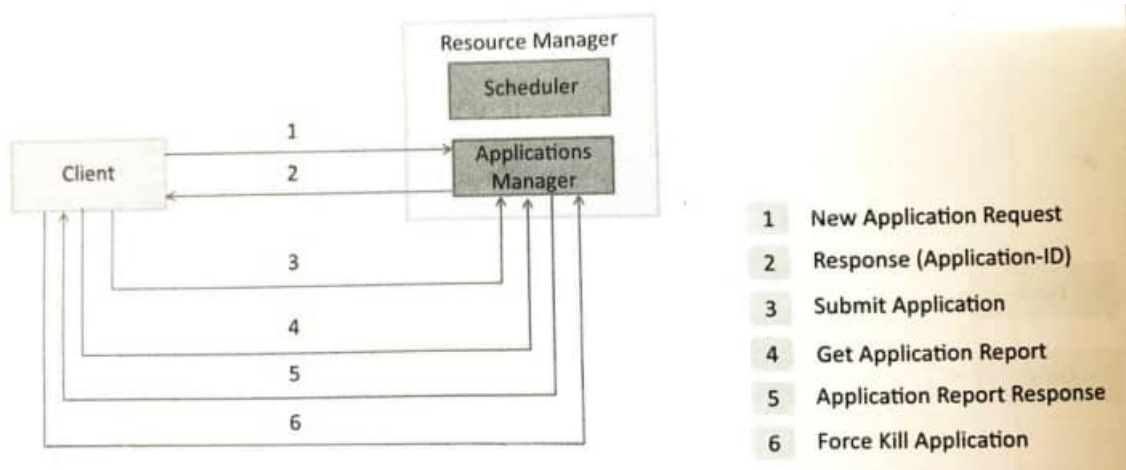


Figure 5.8: Client - Resource Manager interaction

3. 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.
4. The Application Submission also contains a Container Launch Context which contains the application's jar, job files, security tokens and any resource requirements.
5. The client can query the RM for application reports.
6. The client can also "force kill" an application by sending a request to the RM.

Interaction between Resource Manager and Application Master in YARN:

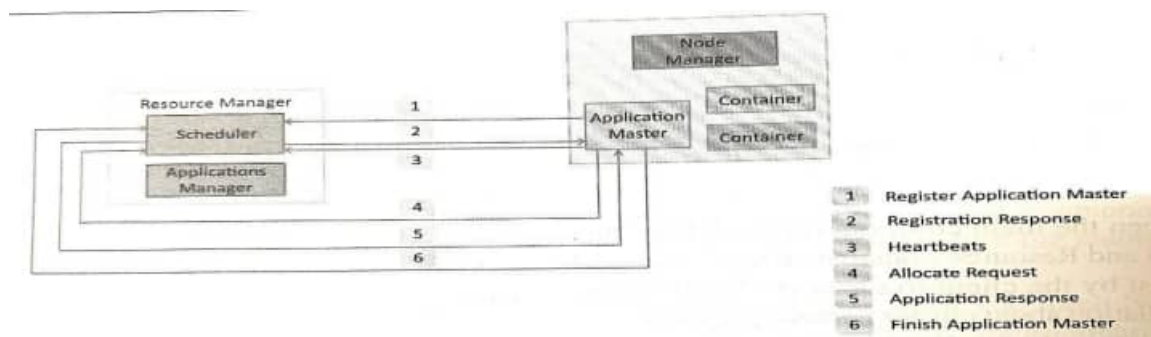


Figure 5.9: Resource Manager - Application Master interaction

Figure 5.9 shows the interactions between Resource Manager and Application Master.

- After 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.
- After finding a suitable container, the RM contacts the NM for the container to start the AM process on its node.
- When the AM is launched, it registers itself with the RM. The registration process consists of handshaking that conveys information such as the RPC port that the AM will be listening on, the tracking URL for monitoring the application's status and progress, etc.
- 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.
- After 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.

Interaction between Application Master and Node Manager in YARN:

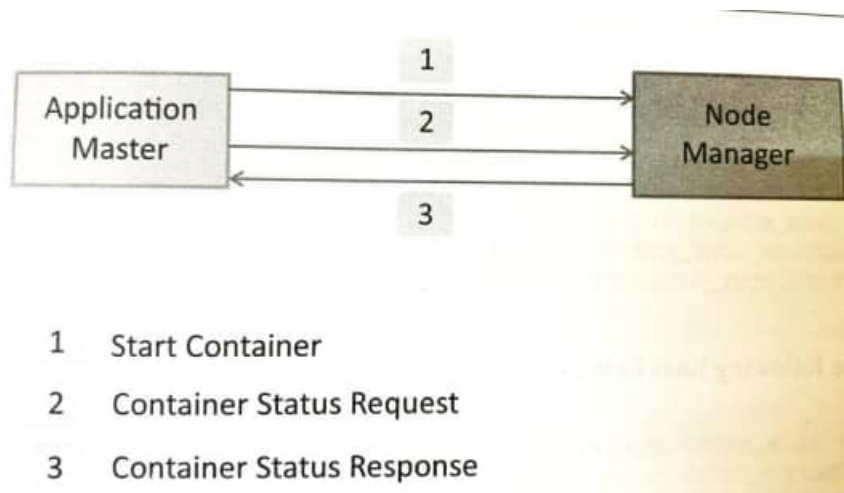


Figure 5.10: Application Master- Node Manager interaction

Figure 5.10 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:

1. Setting up the hosts, installing Java and configuring the networking.
2. Download the Hadoop YARN setup package and unpack it on all nodes.
3. Edit the Hadoop configuration files.
4. Starting/Stopping Hadoop YARN cluster.

5.4 Apache Oozie

- 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, we can create workflows which are a collection of actions (such as MapReduce jobs) arranged as Direct Acyclic Graphs (DAG).
- There may be 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.

5.4.1 Setting up Oozie

1. After setting up Hadoop install the packages required for setting up Oozie.
2. Download and build Oozie.
3. Create a new directory named 'oozie' and copy the built binaries. Also copy the jar files from 'hadooplibs' directory to the libext directory.
4. Download Ext2Js to the 'libext' directory. This is required for the Oozie web console.
5. Prepare the Oozie WAR file.
6. Create sharelib on HDFS.
7. Next create the OozieDB.
8. Finally start Oozie server.
9. The status of Oozie can be checked from command line or the web console.

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

5.4.2 Oozie Workflows for IoT Data Analysis.

- Oozie workflow for IoT data analysis is done by using the example of analyzing machine diagnosis data.
- Assuming that data received from the machine has time stamp and the status /error code. The goal of the analysis job is to find the counts of each status/error code and produce the output.
- Figure 5.11 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

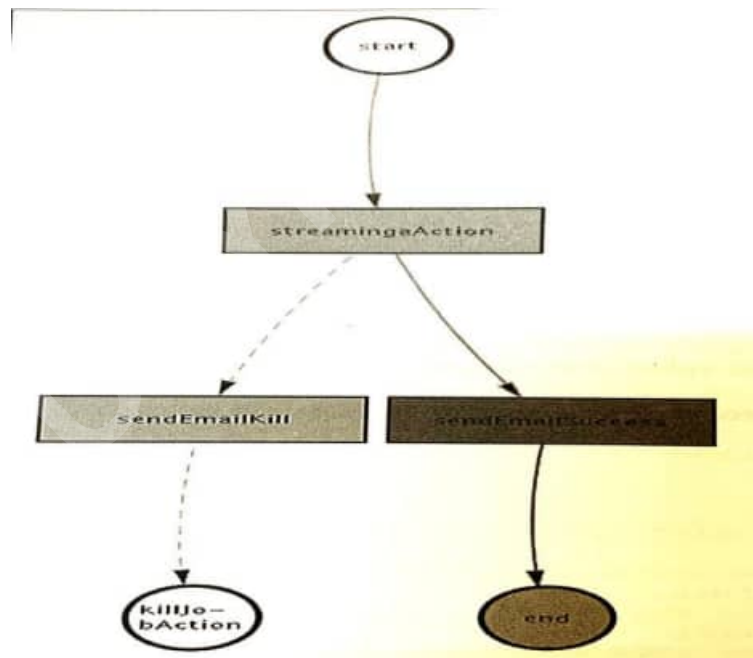


Figure 5:11 Oozie workflow

- 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.

- Oozie workflow has been parameterized with variables within the workflow definition.
- The values of these variables are provided in the job properties file.

10.5 Apache Spark

- Apache Spark is another open source cluster computing framework for data analytics.
- Spark supports in-memory cluster computing and promises to be faster than Hadoop.
- Figure 5.12 shows various high-level tools supported by spark 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.

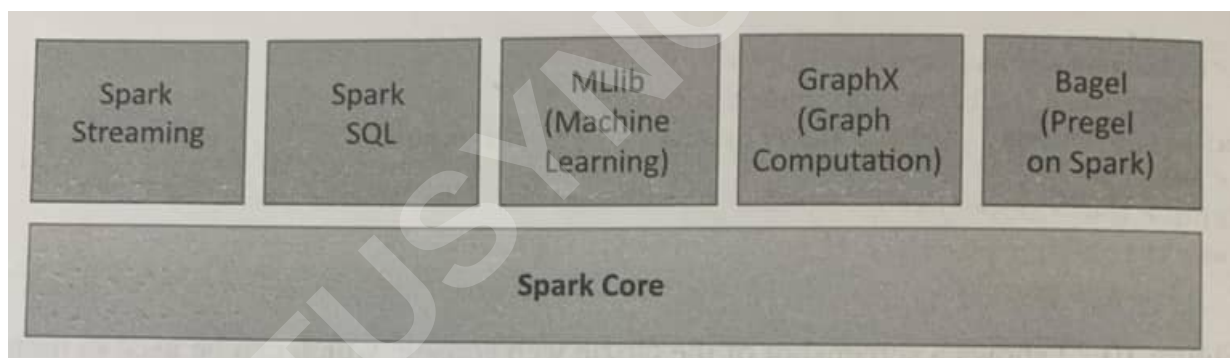


Figure 5:12: Spark tools

- Figure 5.13 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.

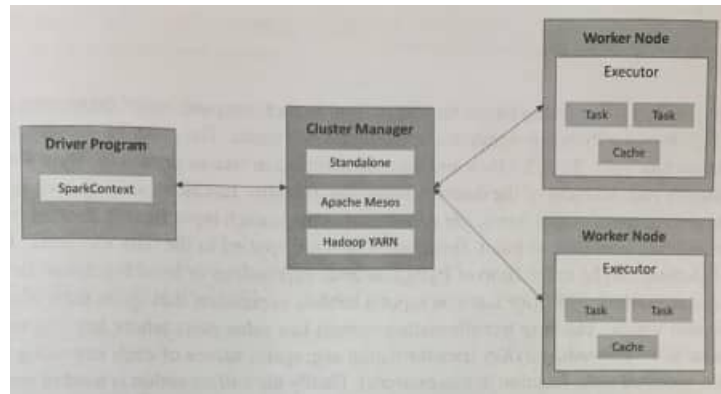


Figure 5.13: Components of a Spark cluster

- 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.

Example-1 stand alone Spark application that computes word counts in a file.

```
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)
```

Box 5.1: Apache Spark Python program for computing word count

- Box 5.1 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.

Example-2 Spark application for batch analysis of data.

```
#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()

maxhumidity = 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 maxhumidity:
    print item

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

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

```

Box 5.2: Apache Spark Python program for computing maximum values for sensor readings

Box 5.2 Apache Spark Python program for computing maximum values for sensor readings.

1. In this example it consider the fire detection sensor data .
2. This sSpark application aggregates the time-stamped sensor data and finds hourly maximum values for temperature, humidity light and CO2.
3. The sensor data is loaded as a text file.
4. Each line of the text file contains time-stamped sensor data.
5. The lines are first split by applying the map transformation to access the individual sensor readings.
6. 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.
7. Finally the reduceByKey transformation is applied to find the maximum sensor reading.

Box 5.3 shows an example of using Spark for data filtering. This example uses the sensor data from forest fire detection IoT system.

```

#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()

```

Box 5.3 : Apache Spark Python program for filtering sensor readings

Spark includes a machine learning library, MLlib, which includes implementations of machine learning algorithms for classification, regression, clustering, collaborative filtering and dimensionality reduction.

Example 5.4: Using MLlib for clustering and classifying data.

1. Box 5.4 shows an example of clustering data with k-means clustering algorithm.
2. In this example, the data is loaded from a text file and then parsed using the `parseVector` function.
3. Next, the `KMeans` object is used to cluster the data into two clusters.
4. Box 5.5 shows an example of classifying data with Naive Bayes classification algorithm.
5. The training data in this example consists of labeled points where value in the first column is the label.
6. The `parsePoint` function parses the data and creates Spark Labeled Point objects.
7. The labeled points are passed to the `NaiveBayes` object for training a model.
8. Finally, the classification is done by passing the test data (as labeled point) to the trained model.

```

#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 5.4: Apache Spark Python program for clustering 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)

```

Box 5.5: Apache Spark Python program for classifying data

10.6 Apache Storm

- Apache Storm is a framework for distributed and fault-tolerant real-time computation
- Storm can be used for real-time processing of streams of data.
- Figure 5.14 shows the components of a Storm cluster.

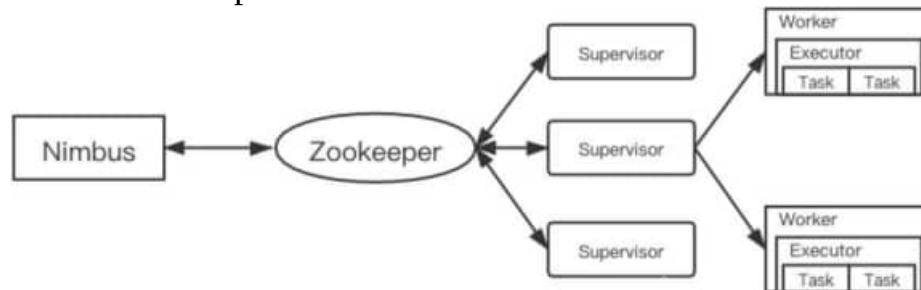


Figure 5.14: Components of a Storm cluster

- 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 .
- Zookeeper is required for coordination of the Storm cluster.
- A computation job on the Storm cluster is called a "topology". It 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 5.15 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.

10.6 Using Apache Storm for Real-time Data Analysis

Apache Storm can be used for real-time analysis of data. Figure 5.15 shows workflow for real-time analysis of sensor data using Storm.

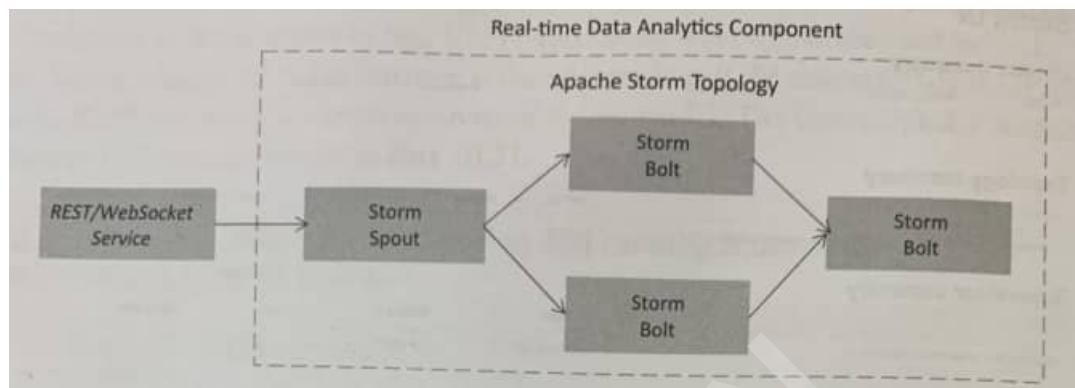


Figure 5.15 :Using Apache Storm for real-time analysis of IoT data

5.7.1 REST-based approach.

- This section describes an example of real-time sensor data analysis for forest fire detection using a REST-based approach.

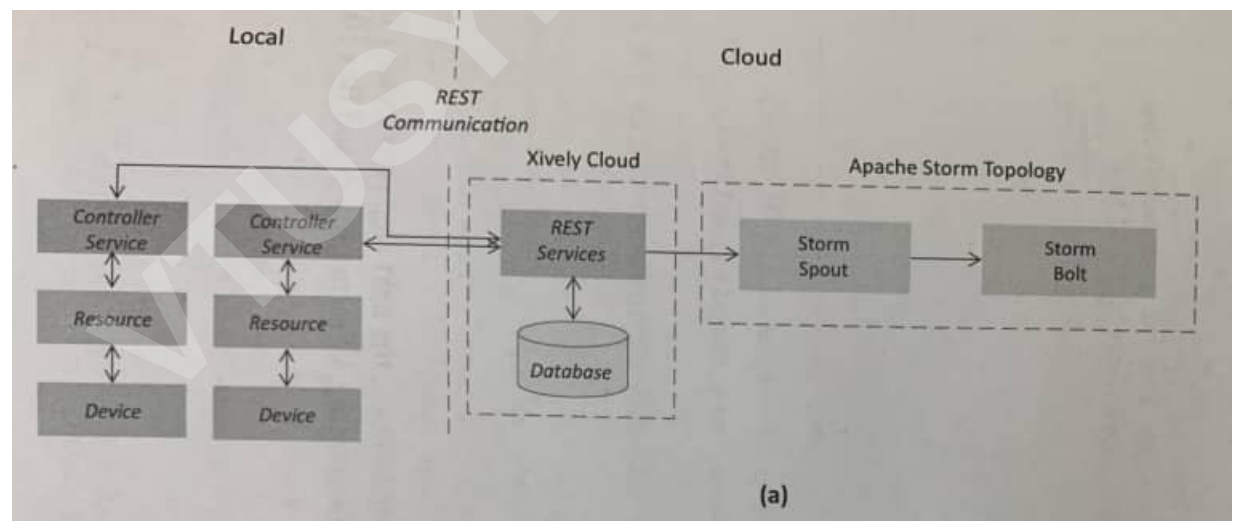


Figure 5.16: Workflow for real-time IoT data analysis with Storm: approach (a) REST-based approach.

- The deployment design for the REST-based implementation is shown in Figure 5.16.
- 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.
- Before the classifier can be used in the Bolt, the classifier has to be trained.
- Box 5.6 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.