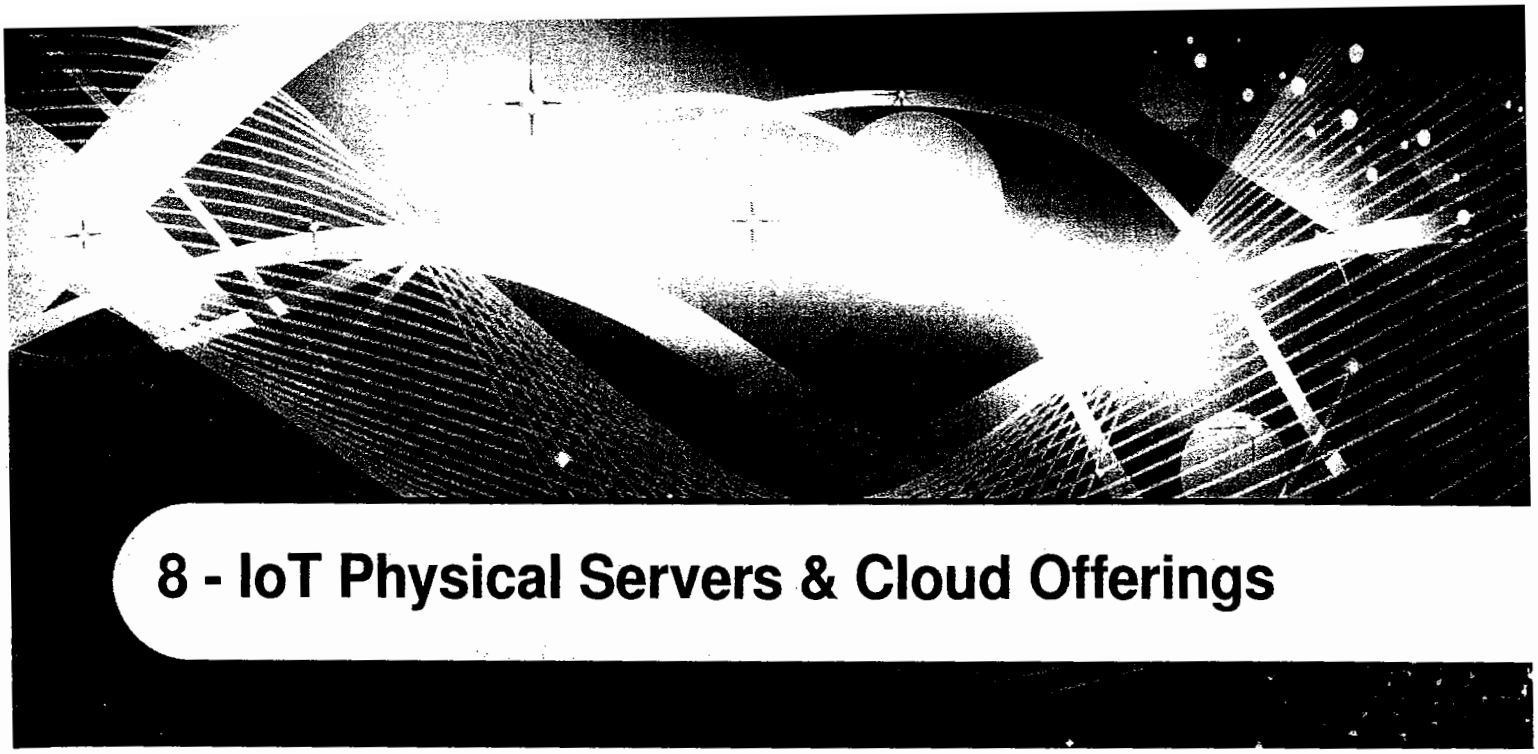

8 - IoT Physical Servers & Cloud Offerings

This Chapter Covers

- Cloud Storage Models & Communication APIs
- Web Application Messaging Protocol (WAMP)
- Xively cloud for IoT
- Python web application framework - Django
- Developing applications with Django
- Developing REST web services
- Amazon Web Services for IoT
- SkyNet IoT Messaging Platform

8.1 Introduction to Cloud Storage Models & Communication APIs

Cloud computing is a transformative computing paradigm that involves delivering applications and services over the Internet. NIST defines cloud computing as [77] - Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. The interested reader may want to refer to the companion book on Cloud Computing by your authors.

In this chapter you will learn how to use cloud computing for Internet of Things (IoT). You will learn about the Web Application Messaging Protocol (WAMP), Xively's Platform-as-a-Service (PaaS) which provides tools and services for developing IoT solutions. You will also learn about the Amazon Web Services (AWS) and their applications for IoT.

8.2 WAMP - AutoBahn for IoT

Web Application Messaging Protocol (WAMP) is a sub-protocol of Websocket which provides publish-subscribe and remote procedure call (RPC) messaging patterns. WAMP enables distributed application architectures where the application components are distributed on multiple nodes and communicate with messaging patterns provided by WAMP.

Let us look at the key concepts of WAMP:

- **Transport:** Transport is channel that connects two peers. The default transport for WAMP is WebSocket. WAMP can run over other transports as well which support message-based reliable bi-directional communication.
- **Session:** Session is a conversation between two peers that runs over a transport.
- **Client:** Clients are peers that can have one or more roles. In publish-subscribe model client can have following roles:
  - **Publisher:** Publisher publishes events (including payload) to the topic maintained by the Broker.
  - **Subscriber:** Subscriber subscribes to the topics and receives the events including the payload.

  In RPC model client can have following roles:
  - **Caller:** Caller issues calls to the remote procedures along with call arguments.
  - **Callee:** Callee executes the procedures to which the calls are issued by the caller and returns the results back to the caller.
- **Router:** Routers are peers that perform generic call and event routing. In publish-subscribe model Router has the role of a Broker:
  - **Broker:** Broker acts as a router and routes messages published to a topic to all

subscribers subscribed to the topic.
In RPC model Router has the role of a Broker:
- **Dealer:** Dealer acts a router and routes RPC calls from the Caller to the Callee and routes results from Callee to Caller.

- **Application Code:** Application code runs on the Clients (Publisher, Subscriber, Callee or Caller).

Figure 8.1 shows a WAMP Session between Client and Router, established over a Transport. Figure 8.2 shows the WAMP protocol interactions between peers. In this figure the WAMP transport used is WebSocket. Recall the WebSocket protocol diagram explained in Chapter-1. WAMP sessions are established over WebSocket transport within the lifetime of WebSocket transport.

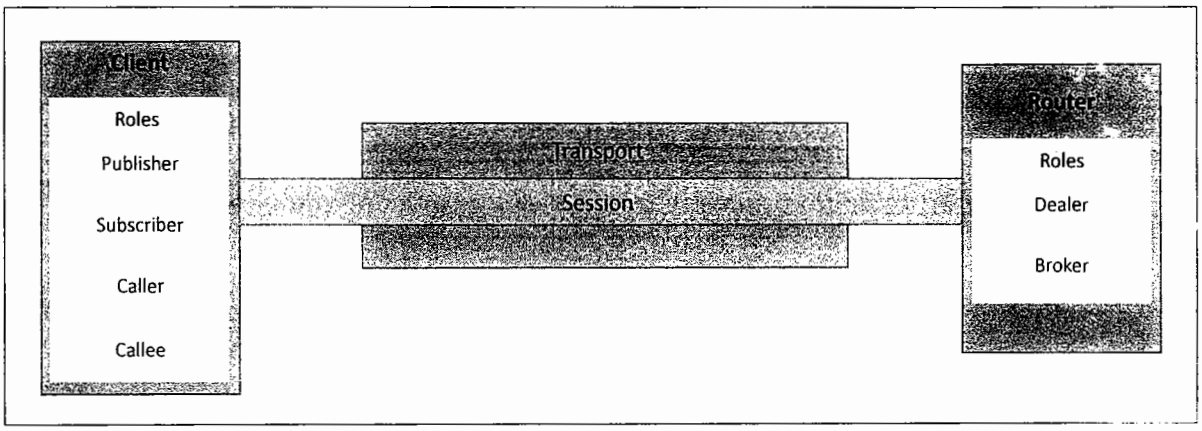

Figure 8.1: WAMP Session between Client and Router

For the examples in this hands-on book we use the AutoBahn framework which provides open-source implementations of the WebSocket and WAMP protocols [100].

Figure 8.3 shows the communication between various components of a typical WAMP-AutoBahn deployment. The Client (in Publisher role) runs a WAMP application component that publishes messages to the Router. The Router (in Broker role) runs on the Server and routes the messages to the Subscribers. The Router (in Broker role) decouples the Publisher from the Subscribers. The communication between Publisher - Broker and Broker - Subscribers happens over a WAMP-WebSocket session.

Let us look at an example of a WAMP publisher and subscriber implemented using AutoBahn. Box 8.1 shows the commands for installing AutoBahn-Python.

■ **Box 8.1: Commands for installing AutoBahn**

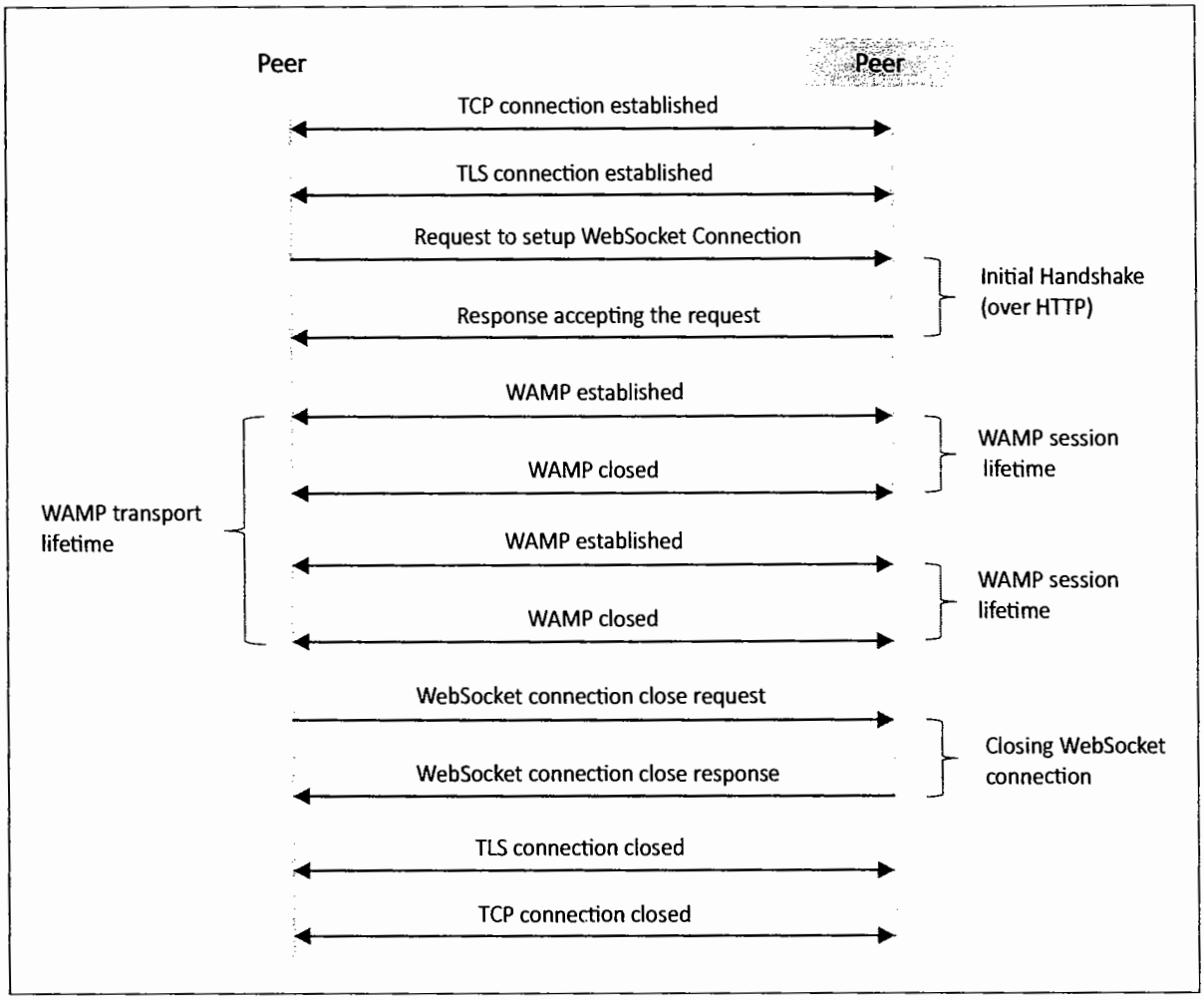

Figure 8.2: WAMP protocol

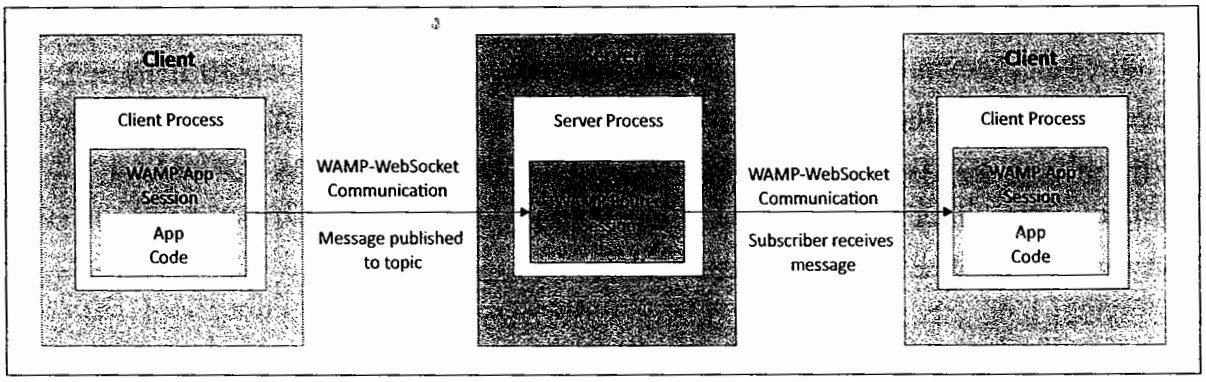

Figure 8.3: Publish-subscribe messaging using WAMP-AutoBahn

```
#Setup Autobahn
sudo apt-get install python-twisted python-dev
```

```
sudo apt-get install python-pip
sudo pip install -upgrade twisted
sudo pip install -upgrade autobahn
```

After installing AutoBahn, clone AutobahnPython from GitHub as follows:

- git clone https://github.com/tavendo/AutobahnPython.git

Create a WAMP publisher component as shown in Box 8.2. The publisher component publishes a message containing the current time-stamp to a topic named 'test-topic'. Next, create a WAMP subscriber component as shown in Box 8.3. The subscriber component that subscribes to the 'test-topic'. Run the application router on a WebSocket transport server as follows:

- python AutobahnPython/examples/twisted/wamp/basic/server.py

Run the publisher component over a WebSocket transport client as follows:

- python AutobahnPython/examples/twisted/wamp/basic/client.py –component "publisherApp.Component"

Run the subscriber component over a WebSocket transport client as follows:

- python AutobahnPython/examples/twisted/wamp/basic/client.py –component "subscriberApp.Component"

■ Box 8.2: Example of a WAMP Publisher implemented using AutoBahn framework - publisherApp.py

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

def getData():
   #Generate message
   timestamp = datetime.datetime.fromtimestamp(
```

```
    time.time()).strftime('%Y-%m-%d%H:%M:%S')
    data = "Message at time-stamp:  "+str(timestamp)
    return data

#An application component that publishes an event every second.
class Component(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        while True:
            data = getData()
            self.publish('test-topic', data)
            yield sleep(1)
```

■ Box 8.3: Example of a WAMP Subscriber implemented using AutoBahn framework - subscriberApp.py

```
from twisted.internet import reactor
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession

#An application component that subscribes and receives events
class Component(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        self.received = 0

        def on_event(data):
            print "Received message:  " + data
            yield self.subscribe(on_event, 'test-topic')

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

While you can setup the server and client processes on a local machine for trying out the publish-subscribe example, in production environment, these components run on separate machines. The server process (the brains or the "Thing Tank"!) is setup on a cloud-based instance while the client processes can run either on local hosts/devices or in the cloud.

8.3 Xively Cloud for IoT

Xively is a commercial Platform-as-a-Service that can be used for creating solutions for Internet of Things. With Xively cloud, IoT developers can focus on the front-end

8.6 Amazon Web Services for IoT

In this section you will learn how to use Amazon Web Services for IoT.

8.6.1 Amazon EC2

Amazon EC2 is an Infrastructure-as-a-Service (IaaS) provided by Amazon. EC2 delivers scalable, pay-as-you-go compute capacity in the cloud. EC2 is a web service that provides computing capacity in the form of virtual machines that are launched in Amazon's cloud computing environment. EC2 can be used for several purposes for IoT systems. For example, IoT developers can deploy IoT applications (developed in frameworks such as Django) on EC2, setup IoT platforms with REST web services, etc.

Let us look at some examples of using EC2. Box 8.23 shows the Python code for launching an EC2 instance. In this example, a connection to EC2 service is first established by calling *boto.ec2.connect_to_region*. The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to EC2 , a new instance is launched using the *conn.run_instances* function. The AMI-ID, instance type, EC2 key handle and security group are passed to this function. This function returns a reservation. The instances associated with the reservation are obtained using *reservation.instances*. Finally the status of an instance associated with a reservation is obtained using the *instance.update* function. In the example shown in Box 8.23, the program waits till the status of the newly launched instance becomes *running* and then prints the instance details such as public DNS, instance IP, and launch time.

■ Box 8.23: Python program for launching an EC2 instance

```
import boto.ec2
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
```

```
    aws_access_key_id=ACCESS_KEY,
    aws_secret_access_key=SECRET_KEY)

print "Launching instance with AMI-ID %s, with keypair
%s, instance type %s, security group
%s"%(AMI_ID,EC2_KEY_HANDLE,INSTANCE_TYPE,SECGROUP_HANDLE)

reservation = conn.run_instances(image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ] )

instance = reservation.instances[0]

print "Waiting for instance to be up and running"

status = instance.update()
while status == 'pending':
    sleep(10)
    status = instance.update()

if status == 'running':
    print " \n Instance is now running.  Instance details are:"
    print "Intance Size:  " + str(instance.instance_type)
    print "Intance State:  " + str(instance.state)
    print "Intance Launch Time:  " + str(instance.launch_time)
    print "Intance Public DNS: " + str(instance.public_dns_name)
    print "Intance Private DNS: " + str(instance.private_dns_name)
    print "Intance IP: " + str(instance.ip_address)
    print "Intance Private IP: " + str(instance.private_ip_address)
```

Box 8.24 shows the Python code for stopping an EC2 instance. In this example the *conn.get_all_instances* function is called to get information on all running instances. This function returns reservations. Next, the IDs of instances associated with each reservation are obtained. The instances are stopped by calling *conn.stop_instances* function to which the IDs of the instances to stop are passed.

■ Box 8.24: Python program for stopping an EC2 instance

```
import boto.ec2
from time import sleep
```

```
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to EC2"

conn = boto.ec2.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Getting all running instances"
reservations = conn.get_all_instances()
print reservations

instance_rs = reservations[0].instances
instance = instance_rs[0]
instanceid=instance_rs[0].id
print "Stopping instance with ID: " + str(instanceid)

conn.stop_instances(instance_ids=[instanceid])

status = instance.update()
while not status == 'stopped':
   sleep(10)
   status = instance.update()

print "Stopped instance with ID: " + str(instanceid)
```

8.6.2 Amazon AutoScaling

Amazon AutoScaling allows automatically scaling Amazon EC2 capacity up or down according to user defined conditions. Therefore, with AutoScaling users can increase the number of EC2 instances running their applications seamlessly during spikes in the application workloads to meet the application performance requirements and scale down capacity when the workload is low to save costs. AutoScaling can be used for auto scaling IoT applications and IoT platforms deployed on Amazon EC2.

Let us now look at some examples of using AutoScaling. Box 8.25 shows the Python code for creating an AutoScaling group. In this example, a connection to AutoScaling service is first established by calling *boto.ec2.autoscale.connect_to_region* function.

The EC2 region, AWS access key and AWS secret key are passed to this function.

After connecting to AutoScaling service, a new launch configuration is created by calling *conn.create_launch_configuration*. Launch configuration contains instructions on how to launch new instances including the AMI-ID, instance type, security groups, etc. After creating a launch configuration, it is then associated with a new AutoScaling group. AutoScaling group is created by calling
conn.create_auto_scaling_group. The settings for AutoScaling group include maximum and minimum number of instances in the group, launch configuration, availability zones, optional load balancer to use with the group, etc. After creating an AutoScaling group, the policies for scaling up and scaling down are defined. In this example, a scale up policy with adjustment type *ChangeInCapacity* and $scaling\_adjustment = 1$ is defined. Similarly a scale down policy with adjustment type *ChangeInCapacity* and $scaling\_adjustment = -1$ is defined. With the scaling policies defined, the next step is to create Amazon CloudWatch alarms that trigger these policies. In this example, alarms for scaling up and scaling down are created. The scale up alarm is defined using the *CPUUtilization* metric with the *Average* statistic and threshold greater 70% for a period of 60 sec. The scale up policy created previously is associated with this alarm. This alarm is triggered when the average CPU utilization of the instances in the group becomes greater than 70% for more than 60 seconds. The scale down alarm is defined in a similar manner with a threshold less than 50%.

Box 8.25: Python program for creating an AutoScaling group

```
import boto.ec2.autoscale
from boto.ec2.autoscale import LaunchConfiguration
from boto.ec2.autoscale import AutoScalingGroup
from boto.ec2.cloudwatch import MetricAlarm
from boto.ec2.autoscale import ScalingPolicy
import boto.ec2.cloudwatch

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

print "Connecting to Autoscaling Service"

conn = boto.ec2.autoscale.connect_to_region(REGION,
```

```
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)


print "Creating launch configuration"


lc = LaunchConfiguration(name='My-Launch-Config-2',
        image_id=AMI_ID,
        key_name=EC2_KEY_HANDLE,
        instance_type=INSTANCE_TYPE,
        security_groups = [ SECGROUP_HANDLE, ])


conn.create_launch_configuration(lc)

print "Creating auto-scaling group"

ag = AutoScalingGroup(group_name='My-Group',
        availability_zones=['us-east-1b'],
        launch_config=lc, min_size=1, max_size=2,
        connection=conn)

conn.create_auto_scaling_group(ag)

print "Creating auto-scaling policies"

scale_up_policy = ScalingPolicy(name='scale_up',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=1,
           cooldown=180)

scale_down_policy = ScalingPolicy(name='scale_down',
            adjustment_type='ChangeInCapacity',
           as_name='My-Group',
            scaling_adjustment=-1,
           cooldown=180)


conn.create_scaling_policy(scale_up_policy)
conn.create_scaling_policy(scale_down_policy)


scale_up_policy = conn.get_all_policies( as_group='My-Group',
```

```
policy_names=['scale_up'])[0]
scale_down_policy = conn.get_all_policies( as_group='My-Group',
policy_names=['scale_down'])[0]

print "Connecting to CloudWatch"

cloudwatch = boto.ec2.cloudwatch.connect_to_region(REGION,
        aws_access_key_id=ACCESS_KEY,
         aws_secret_access_key=SECRET_KEY)

alarm_dimensions = "AutoScalingGroupName":  'My-Group'

print "Creating scale-up alarm"

scale_up_alarm = MetricAlarm(
     name='scale_up_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='>', threshold='70',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_up_policy.policy_arn] ,
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_up_alarm)

print "Creating scale-down alarm"

scale_down_alarm = MetricAlarm(
     name='scale_down_on_cpu', namespace='AWS/EC2',
     metric='CPUUtilization', statistic='Average',
     comparison='<', threshold='50',
     period='60', evaluation_periods=2,
     alarm_actions=[scale_down_policy.policy_arn],
     dimensions=alarm_dimensions)

cloudwatch.create_alarm(scale_down_alarm)
print "Done!"
```

8.6.3 Amazon S3

Amazon S3 is an online cloud-based data storage infrastructure for storing and retrieving a very large amount of data. S3 provides highly reliable, scalable, fast, fully redundant and affordable storage infrastructure. S3 can serve as a raw datastore (or "Thing Tank") for IoT systems for storing raw data, such as sensor data, log data, image, audio and video data.

Let us look at some examples of using S3. Box 8.26 shows the Python code for uploading

a file to Amazon S3 cloud storage. In this example, a connection to S3 service is first established by calling *boto.connect_s3* function. The AWS access key and AWS secret key are passed to this function. This example defines two functions *upload_to_s3_bucket_path* and *upload_to_s3_bucket_root*. The *upload_to_s3_bucket_path* function uploads the file to the S3 bucket specified at the specified path. The *upload_to_s3_bucket_root* function uploads the file to the S3 bucket root.

■ Box 8.26: Python program for uploading a file to an S3 bucket

```
import boto.s3

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

conn = boto.connect_s3(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

def percent_cb(complete, total):
   print ('.')

def upload_to_s3_bucket_path(bucketname, path, filename):
   mybucket = conn.get_bucket(bucketname)
   fullkeyname=os.path.join(path,filename)
   key = mybucket.new_key(fullkeyname)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

def upload_to_s3_bucket_root(bucketname, filename):
   mybucket = conn.get_bucket(bucketname)
   key = mybucket.new_key(filename)
   key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)

upload_to_s3_bucket_path('mybucket2013', 'data', 'file.txt')
```

8.6.4 Amazon RDS

Amazon RDS is a web service that allows you to create instances of MySQL, Oracle or Microsoft SQL Server in the cloud. With RDS, developers can easily set up, operate, and scale a relational database in the cloud.

RDS can serve as a scalable datastore for IoT systems. With RDS, IoT system developers can store any amount of data in scalable relational databases. Let us look at some examples of using RDS. Box 8.27 shows the Python code for launching an Amazon RDS instance. In this example, a connection to RDS service is first established by calling *boto.rds.connect_to_region*

function. The RDS region, AWS access key and AWS secret key are passed to this function. After connecting to RDS service, the *conn.create_dbinstance* function is called to launch a new RDS instance. The input parameters to this function include the instance ID, database size, instance type, database username, database password, database port, database engine (e.g. MySQL5.1), database name, security groups, etc. The program shown in Box 8.27 waits till the status of the RDS instance becomes *available* and then prints the instance details such as instance ID, create time, or instance end point.

■ Box 8.27: Python program for launching an RDS instance

```
import boto.rds
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

print "Connecting to RDS"

conn = boto.rds.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Creating an RDS instance"

db = conn.create_dbinstance(ID, DB_SIZE, INSTANCE_TYPE,
USERNAME, PASSWORD, port=DB_PORT, engine=DB_ENGINE,
db_name=DB_NAME, security_groups = [
SECGROUP_HANDLE, ] )
print db

print "Waiting for instance to be up and running"
```

```
status = db.status
while not status == 'available':
   sleep(10)
   status = db.status

if status == 'available':
   print " \n RDS Instance is now running.  Instance details are:"
   print "Intance ID: " + str(db.id)
   print "Intance State:  " + str(db.status)
   print "Intance Create Time:  " + str(db.create_time)
   print "Engine:  " + str(db.engine)
   print "Allocated Storage:  " + str(db.allocated_storage)
   print "Endpoint:  " + str(db.endpoint)
```

Box 8.28 shows the Python code for creating a MySQL table, writing and reading from the table. This example uses the MySQLdb Python package. To connect to the MySQL RDS instance, the *MySQLdb.connect* function is called and the end point of the RDS instance, database username, password and port are passed to this function. After the connection to the RDS instance is established, a cursor to the database is obtained by calling *conn.cursor*. Next, a new database table named *TemperatureData* is created with *Id* as primary key and other columns. After creating the table some values are inserted. To execute the SQL commands for database manipulation, the commands are passed to the *cursor.execute* function.

■ Box 8.28: Python program for creating a MySQL table, writing and reading from the table

```
import MySQLdb

USERNAME = 'root'
PASSWORD = 'password'
DB_NAME = 'mytestdb'

print "Connecting to RDS instance"

conn = MySQLdb.connect (host =
"mysql-db-instance-3.c35qdifuf9ko.us-east-1.rds.amazonaws.com",
user = USERNAME,
passwd = PASSWORD,
db = DB_NAME,
port = 3306)

print "Connected to RDS instance"
```

```
cursor = conn.cursor ()
cursor.execute ("SELECT VERSION()")
row = cursor.fetchone ()
print "server version:", row[0]

cursor.execute ("CREATE TABLE TemperatureData(Id INT PRIMARY KEY,
Timestamp TEXT, Data TEXT) ")
cursor.execute ("INSERT INTO TemperatureData VALUES(1,
'1393926310', '20')")
cursor.execute ("INSERT INTO TemperatureData VALUES(2,
'1393926457', '25')")

cursor.execute("SELECT * FROM TemperatureData")
rows = cursor.fetchall()

for row in rows:
   print row

cursor.close ()
conn.close ()
```

8.6.5 Amazon DynamoDB

Amazon DynamoDB is a fully-managed, scalable, high performance No-SQL database service. DynamoDB can serve as a scalable datastore for IoT systems. With DynamoDB, IoT system developers can store any amount of data and serve any level of requests for the data.

Let us look at some examples of using DynamoDB. Box 8.29 shows the Python code for creating a DynamoDB table. In this example, a connection to DynamoDB service is first established by calling *boto.dynamodb.connect_to_region*. The DynamoDB region, AWS access key and AWS secret key are passed to this function. After connecting to DynamoDB service, a schema for the new table is created by calling *conn.create_schema*. The schema includes the hash key and range key names and types. A DynamoDB table is then created by calling *conn.create_table* function with the table schema, read units and write units as input parameters.

■ Box 8.29: Python program for creating a DynamoDB table

```
import boto.dynamodb
import time
```

```
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to DynamoDB"

conn = boto.dynamodb.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

table_schema = conn.create_schema(
   hash_key_name='msgid',
   hash_key_proto_value=str,
   range_key_name='date',
   range_key_proto_value=str
   )

print "Creating table with schema:"
print table_schema


table = conn.create_table(
   name='my-test-table',
   schema=table_schema,
   read_units=1,
   write_units=1
   )

print "Creating table:"
print table

print "Done!"
```

Box 8.30 shows the Python code for writing and reading from a DynamoDB table. After establishing a connection with DynamoDB service, the *conn.get_table* is called to retrieve an existing table. The data written in this example consists of a JSON message with keys - *Body*, *CreatedBy* and *Time*. After creating the JSON message, a new DynamoDB table item is created by calling *table.new_item* and the hash key and range key is specified. The data item is finally committed to DynamoDB by calling *item.put*. To read data from DynamoDB, the *table.get_item* function is used with the hash key and range key as input parameters.

■ Box 8.30: Python program for writing and reading from a DynamoDB table

```
import boto.dynamodb
import time
from datetime import date

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to DynamoDB"

conn = boto.dynamodb.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Listing available tables"
tables_list = conn.list_tables()
print tables_list

print "my-test-table description"
desc = conn.describe_table('my-test-table')
print desc

msg_datetime = time.asctime(time.localtime(time.time()))

print "Writing data"

table = conn.get_table("my-test-table")

hash_attribute = "Entry/" + str(date.today())

item_data =
   'Body': 'Test message',
   'CreatedBy': 'Vijay',
   'Time': msg_datetime,


item = table.new_item(
   hash_key=hash_attribute,
   range_key=str(date.today()),
   attrs=item_data
)
item.put()
```

```
print "Reading data"

table = conn.get_table('my-test-table')

read_data = table.get_item(
     hash_key=hash_attribute,
     range_key=str(date.today())
     )

print read_data
print "Done!"
```

8.6.6 Amazon Kinesis

Amazon Kinesis is a fully managed commercial service that allows real-time processing of streaming data. Kinesis scales automatically to handle high volume streaming data coming from large number of sources. The streaming data collected by Kinesis can be processed by applications running on Amazon EC2 instances or any other compute instance that can connect to Kinesis. Kinesis is well suited for IoT systems that generate massive scale data and have strict real-time requirements for processing the data. Kinesis allows rapid and continuous data intake and support data blobs of size upto 50Kb. The data producers (e.g. IoT devices) write data records to Kinesis streams. A data record comprises of a sequence number, a partition key and the data blob. Data records in a Kinesis stream are distributed in shards. Each shard provides a fixed unit of capacity and a stream can have multiple shards. A single shard of throughput allows capturing 1MB per second of data, at up to 1,000 PUT transactions per second and allows applications to read data at up to 2 MB per second.

Box 8.31 shows a Python program for writing to a Kinesis stream. This example follows a similar structure as the controller program in Box 8.4 that sends temperature data from an IoT device to the cloud. In this example a connection to the Kinesis service is first established and then a new Kinesis stream is either created (if not existing) or described. The data is written to the Kinesis stream using the *kinesis.put_record* function.

■ Box 8.31: Python program for writing to a Kinesis stream

```
import json
import time
import datetime
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
```

```
from random import randint

ACCESS_KEY = "<enter access key>"
SECRET_KEY = "<enter secret key>"

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
global stream


def readTempSensor():
#Return random value
   return randint(20,30)

#Controller main function
def runController():
   temperature = readTempSensor()
   timestamp = datetime.datetime.utcnow()
   record=str(timestamp)+":"+str(temperature)
   print "Putting record in stream:  " + record
   response = kinesis.put_record( stream_name=streamName,
data=record, partition_key=partitionKey)
   print ("-= put seqNum:", response['SequenceNumber'])

def get_or_create_stream(stream_name, shard_count):
   stream = None
   try:
      stream = kinesis.describe_stream(stream_name)
      print (json.dumps(stream, sort_keys=True, indent=2, separators=(',',
':  ')))
   except ResourceNotFoundException as rnfe:
      while (stream is None) or (stream['StreamStatus'] is not 'ACTIVE'):
         print ('Could not find ACTIVE stream:0 trying to create.'.format(
            stream_name))
         stream = kinesis.create_stream(stream_name, shard_count)
         time.sleep(0.5)

   return stream


def setupController():
   global stream
```

```
    stream = get_or_create_stream(streamName, shardCount)

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

Box 8.32 shows a Python program for reading from a Kinesis stream. In this example a shard iterator is obtained using the *kinesis.get_shard_iterator* function. The shard iterator specifies the position in the shard from which you want to start reading data records sequentially. The data is read using the *kinesis.get_records* function which returns one or more data records from a shard.

■ Box 8.32: Python program for reading from a Kinesis stream

```
import json
import time
import boto.kinesis
from boto.kinesis.exceptions import ResourceNotFoundException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException
from boto.kinesis.exceptions import
ProvisionedThroughputExceededException

ACCESS_KEY = "<enter access key>"
SECRET_KEY = "<enter secret key>"

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
aws_secret_access_key=SECRET_KEY)
streamName = "temperature"
partitionKey = "IoTExample"
shardCount = 1
iterator_type='LATEST'

stream = kinesis.describe_stream(streamName)
print (json.dumps(stream, sort_keys=True, indent=2,
separators=(',', ':  ')))
shards = stream['StreamDescription']['Shards']
print ('# Shard Count:', len(shards))

def processRecords(records):
    for record in records:
        text = record['Data'].lower()
```

```
        print 'Processing record with data:  '  + text

i=0
response = kinesis.get_shard_iterator(streamName, shards[0]['ShardId'],
'TRIM_HORIZON', starting_sequence_number=None)
next_iterator = response['ShardIterator']
print ('Getting next records using iterator:  ', next_iterator)
while i<10:
   try:
      response = kinesis.get_records(next_iterator, limit=1)
      #print response
      if len(response['Records']) > 0:
         #print 'Number of records fetched:'  +
str(len(response['Records']))
         processRecords(response['Records'] )

      next_iterator = response['NextShardIterator']
      time.sleep(1)
      i=i+1

   except ProvisionedThroughputExceededException as ptee:
      print (ptee.message)
      time.sleep(5)
```

8.6.7 Amazon SQS

Amazon SQS offers a highly scalable and reliable hosted queue for storing messages as they travel between distinct components of applications. SQS guarantees only that messages arrive, not that they arrive in the same order in which they were put in the queue. Though, at first look, Amazon SQS may seem to be similar to Amazon Kinesis, however, both are intended for very different types of applications. While Kinesis is meant for real-time applications that involve high data ingress and egress rates, SQS is simply a queue system that stores and releases messages in a scalable manner.

SQS can be used in distributed IoT applications in which various application components need to exchange messages. Let us look at some examples of using SQS. Box 8.33 shows the Python code for creating an SQS queue. In this example, a connection to SQS service is first established by calling *boto.sqs.connect_to_region*. The AWS region, access key and secret key are passed to this function. After connecting to SQS service, *conn.create_queue* is called to create a new queue with queue name as input parameter. The function *conn.get_all_queues* is used to retrieve all SQS queues.

■ Box 8.33: Python program for creating an SQS queue

```
import boto.sqs

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
      REGION,
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Creating queue with name:  " + queue_name
q = conn.create_queue(queue_name)

print "Created queue with name:  " + queue_name

print " \n Getting all queues"

rs = conn.get_all_queues()

for item in rs:
   print item
```

Box 8.34 shows the Python code for writing to an SQS queue. After connecting to an SQS queue, the *queue.write* is called with the message as input parameter.

■ Box 8.34: Python program for writing to an SQS queue

```
import boto.sqs
from boto.sqs.message import Message
import time

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
```

```
print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
     aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)

msg_datetime = time.asctime(time.localtime(time.time()))

msg = "Test message generated on:  " + msg_datetime
print "Writing to queue:  " + msg

m = Message()
m.set_body(msg)
status = q[0].write(m)

print "Message written to queue"

count = q[0].count()

print "Total messages in queue:  " + str(count)
```

Box 8.35 shows the Python code for reading from an SQS queue. After connecting to an SQS queue, the *queue.read* is called to read a message from a queue.

■ Box 8.35: Python program for reading from an SQS queue

```
import boto.sqs
from boto.sqs.message import Message

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect_to_region(
     REGION,
```

```
      aws_access_key_id=ACCESS_KEY,
      aws_secret_access_key=SECRET_KEY)

queue_name = 'mytestqueue'

print "Connecting to queue:  " + queue_name
q = conn.get_all_queues(prefix=queue_name)

count = q[0].count()

print "Total messages in queue:  " + str(count)

print "Reading message from queue"

for i in range(count):
   m = q[0].read()
   print "Message %d:  %s" % (i+1,str(m.get_body()))
   q[0].delete_message(m)

print "Read %d messages from queue" % (count)
```

8.6.8 Amazon EMR

Amazon EMR is a web service that utilizes Hadoop framework running on Amazon EC2 and Amazon S3. EMR allows processing of massive scale data, hence, suitable for IoT applications that generate large volumes of data that needs to be analyzed. Data processing jobs are formulated with the MapReduce parallel data processing model.

MapReduce is a 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.

Consider an IoT system that collects data from a machine (or sensor data) which is logged in a cloud storage (such as Amazon S3) and analyzed on hourly basis to generate alerts if a certain sequence occurred more than a predefined number of times. Since the scale of data involved in such applications can be massive, MapReduce is an ideal choice for processing such data.

Let us look at a MapReduce example that finds the number of occurrences of a sequence from a log. Box 8.36 shows the Python code for launching an Elastic MapReduce job. In this example, a connection to EMR service is first established by calling *boto.emr.connect_to_region*. The AWS region, access key and secret key are passed to this function. After connecting to EMR service, a jobflow step is created. There are two types of steps - streaming and

custom jar. To create a streaming job an object of the *StreamingStep* class is created by specifying the job name, locations of the mapper, reducer, input and output. The job flow is then started using the *conn.run_jobflow* function with streaming step object as input. When the MapReduce job completes, the output can be obtained from the output location on the S3 bucket specified while creating the streaming step.

■ Box 8.36: Python program for launching an EMR job

```
import boto.emr
from boto.emr.step import StreamingStep
from time import sleep

ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"
REGION="us-east-1"

print "Connecting to EMR"

conn = boto.emr.connect_to_region(REGION,
     aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

print "Creating streaming step"

step = StreamingStep(name='Sequence Count',
     mapper='s3n://mybucket/seqCountMapper.py',
     reducer='s3n://mybucket/seqCountReducer.py',
     input='s3n://mybucket/data/',
     output='s3n://mybucket/seqcountoutput/')


print "Creating job flow"

jobid = conn.run_jobflow(name='Sequence Count Jobflow',
     log_uri='s3n://mybucket/wordcount_logs',
     steps=[step])

print "Submitted job flow"

print "Waiting for job flow to complete"

status = conn.describe_jobflow(jobid)
print status.state
```

```
while status.state != 'COMPLETED' or status.state != 'FAILED':
   sleep(10)
   status = conn.describe_jobflow(jobid)

print "Job status:  " + str(status.state)

print "Done!"
```

Box 8.37 shows the sequence count mapper program in Python. The mapper reads the data from standard input (stdin) and for each line in input in which the sequence occurs, the mapper emits a key-value pair where key is the sequence and value is equal to 1.

■ Box 8.37: Sequence count Mapper in Python

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

#Enter the sequence to search
seq='123'
for line in sys.stdin:
   line = line.strip()
   if seq in line:
      print '%st%s' % (seq, 1)
```

Box 8.38 shows the sequence count reducer program in Python. The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key. The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and sums up the occurrences to compute the count for each sequence.

■ Box 8.38: Sequence count Reducer in Python

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

current_seq = None
current_count = 0
seq = None

for line in sys.stdin:
```

```
    line = line.strip()
    seq, count = line.split('t', 1)
    current_count += count

    try:
        count = int(count)
    except ValueError:
        continue

    if current_seq == seq:
        current_count += count
else:
    if current_seq:
        print '%st%s' % (current_seq,current_count)
    current_count = count
    current_seq = seq

if current_seq == seq:
    print '%st%s' % (current_seq, current_count)
```

8.7 SkyNet IoT Messaging Platform

SkyNet is an open source instant messaging platform for Internet of Things. The SkyNet API supports both HTTP REST and real-time WebSockets. SkyNet allows you to register devices (or nodes) on the network. A device can be anything including sensors, smart home devices, cloud resources, drones, etc. Each device is assigned a UUID and a secret token. Devices or client applications can subscribe to other devices and receive/send messages.

Box 8.39 shows the commands to setup SkyNet on a Linux machine. Box 8.40 shows a sample configuration for SkyNet. Box 8.41 shows examples of using SkyNet. The first step is to create a device on SkyNet. The POST request to create a device returns the UUID and token of the created device. The box also shows examples of updating a device, retrieving last 10 events related to a device, subscribing to a device and sending a message to a device. Box 8.42 shows the code for a Python client that performs various functions such as subscribing to a device, sending a message and retrieving the service status.

■ Box 8.39: Commands for Setting up SkyNet

```
#Install DB Redis:
sudo apt-get install redis-server

#Install MongoDB:
```