Chapter 8

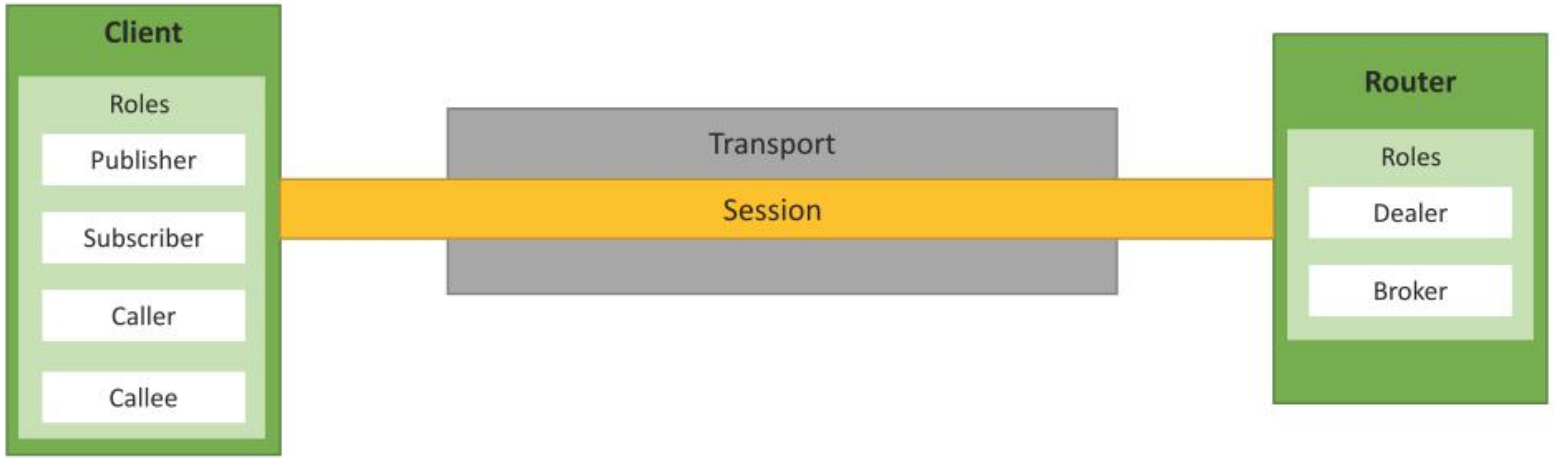
IoT Physical Servers & Cloud Offerings

Outline

- WAMP - AutoBahn for IoT
- Python for Amazon Web Services
- Python for MapReduce
- Python Packages of Interest
- Python Web Application Framework - Django
- Development with Django

WAMP 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 - Concepts

- Transport: Transport is channel that connects two peers.
- 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).

Amazon EC2 – Python Example

- Boto is a Python package that provides interfaces to Amazon Web Services (AWS)

  - In this example, a connection to EC2 service is fi rst 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.

```
#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"

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

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

Amazon AutoScaling – Python Example

- AutoScaling Service
  - A connection to AutoScaling service is first established by calling boto.ec2.autoscale.connect_to_region function.
- Launch Configuration
  - After connecting to AutoScaling service, a new launch configuration is created by calling conn.create_launch_con f iguration. Launch configuration contains instructions on how to launch new instances including the AMI-ID, instance type, security groups, etc.
- AutoScaling Group
  - 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 such as the maximum and minimum number of instances in the group, the launch configuration, availability zones, optional load balancer to use with the group, etc.

```
#Python program for creating an AutoScaling group (code excerpt)
import boto.ec2.autoscale
:
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)
```

Amazon AutoScaling – Python Example

- AutoScaling Policies
  - 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_ad justment = 1 is defined.
  - Similarly a scale down policy with adjustment type ChangeInCapacity and scaling_ad justment = -1 is defined.

```
#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)
```

Amazon AutoScaling – Python Example

- CloudWatch Alarms
  - With the scaling policies defined, the next step is to create Amazon CloudWatch alarms that trigger these policies.
  - 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%.

```
#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'}

#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)

#Creating scale-down alarm
scale_down_alarm = MetricAlarm(
        name='scale_down_on_cpu', namespace='AWS/EC2',
        metric='CPUUtilization', statistic='Average',
        comparison='<', threshold='40',
        period='60', evaluation_periods=2,
        alarm_actions=[scale_down_policy.policy_arn],
        dimensions=alarm_dimensions)
cloudwatch.create_alarm(scale_down_alarm)
```

Amazon S3 – Python Example

- In this example, a connection to S3 service is first established by calling boto.connect_s3 function.
- The upload_to_s3_bucket_path function uploads the file to the S3 bucket specified at the specified path.

```
# Python program for uploading a file to an S3 bucket
import boto.s3

conn = boto.connect_s3(aws_access_key_id='<enter>',
    aws_secret_access_key='<enter>')

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)
```

Amazon RDS – Python Example

- 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.

```
#Python program for launching an RDS instance (excerpt)
import boto.rds

ACCESS_KEY="<enter>"
SECRET_KEY="<enter>"
REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance-3"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

#Connecting to RDS
conn =  boto.rds.connect_to_region(REGION,
  aws_access_key_id=ACCESS_KEY,
  aws_secret_access_key=SECRET_KEY)


#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, ] )
```

Amazon DynamoDB – Python Example

- In this example, a connection to DynamoDB service is first established by calling boto.dynamodb.connect_to_region.
- 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.

```
# Python program for creating a DynamoDB table (excerpt)
import boto.dynamodb

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

#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
  )

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

Python for MapReduce

- The example shows inverted index mapper program.
- The map function reads the data from the standard input (stdin) and splits the tab-limited data into document-ID and contents of the document.
- The map function emits key-value pairs where key is each word in the document and value is the document-ID.

```
#Inverted Index Mapper in Python

#!/usr/bin/env python
import sys
for line in sys.stdin:
        doc_id, content = line.split('')
        words = content.split()
        for word in words:
                print '%s%s' % (word, doc_id)
```

Python for MapReduce

- The example shows inverted index reducer program.
- 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 creates a list of document-IDs in which the word occurs.
- The output of reducer contains key value pairs where key is a unique word and value is the list of document-IDs in which the word occurs.

```
#Inverted Index Reducer in Python

#!/usr/bin/env python
import sys
current_word = None
current_docids = []
word = None

for line in sys.stdin:
        # remove leading and trailing whitespace
        line = line.strip()
        # parse the input we got from mapper.py
        word, doc_id = line.split('')
        if current_word == word:
                current_docids.append(doc_id)
        else:
                if current_word:
                        print '%s%s' % (current_word, current_docids)
                        current_docids = []
                        current_docids.append(doc_id)
                        current_word = word
```

Python Packages of Interest

- JSON
  - JavaScript Object Notation (JSON) is an easy to read and write data-interchange format. JSON is used as an alternative to XML and is is easy for machines to parse and generate. JSON is built on two structures - a collection of name-value pairs (e.g. a Python dictionary) and ordered lists of values (e.g.. a Python list).
- XML
  - XML (Extensible Markup Language) is a data format for structured document interchange. The Python minidom library provides a minimal implementation of the Document Object Model interface and has an API similar to that in other languages.
- HTTPLib & URLLib
  - HTTPLib2 and URLLib2 are Python libraries used in network/internet programming
- SMTPLib
  - Simple Mail Transfer Protocol (SMTP) is a protocol which handles sending email and routing e-mail between mail servers. The Python smtplib module provides an SMTP client session object that can be used to send email.
- NumPy
  - NumPy is a package for scientific computing in Python. NumPy provides support for large multi-dimensional arrays and matrices
- Scikit-learn
  - Scikit-learn is an open source machine learning library for Python that provides implementations of various machine learning algorithms for classification, clustering, regression and dimension reduction problems.

Python Web Application Framework - Django

- Django is an open source web application framework for developing web applications in Python.
- A web application framework in general is a collection of solutions, packages and best practices that allows development of web applications and dynamic websites.
- Django is based on the Model-Template-View architecture and provides a separation of the data model from the business rules and the user interface.
- Django provides a unified API to a database backend.
- Thus web applications built with Django can work with different databases without requiring any code changes.
- With this fiexibility in web application design combined with the powerful capabilities of the Python language and the Python ecosystem, Django is best suited for cloud applications.
- Django consists of an object-relational mapper, a web templating system and a regular-expression-based URL dispatcher.

Django Architecture

- Django is Model-Template-View (MTV) framework.

- Model
  - The model acts as a definition of some stored data and handles the interactions with the database. In a web application, the data can be stored in a relational database, non-relational database, an XML file, etc. A Django model is a Python class that outlines the variables and methods for a particular type of data.
- Template
  - In a typical Django web application, the template is simply an HTML page with a few extra placeholders. Django's template language can be used to create various forms of text files (XML, email, CSS, Javascript, CSV, etc.)
- View
  - The view ties the model to the template. The view is where you write the code that actually generates the web pages. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template.

Further Reading

- boto, http://boto.readthedocs.org/en/latest/
- Python JSON package, http://docs.python.org/library/json.html
- Python socket package, http://docs.python.org/2/library/socket.html
- Python email package, http://docs.python.org/2/library/email
- Python HTTPLib, http://code.google.com/p/httplib2/
- Python URLLib, http://docs.python.org/2/howto/urllib2.html
- Python SMTPLib, http://docs.python.org/2/library/smtplib.html
- NumPy, http://www.numpy.org/
- Scikit-learn, http://scikit-learn.org/stable/
- Django, https://docs.djangoproject.com/en/1.5/
- Google App Engine, https://developers.google.com/appengine/
- Google Cloud Storage, https://developers.google.com/storage/
- Google BigQuery, https://developers.google.com/bigquery/
- Google Cloud Datastore, http://developers.google.com/datastore/
- Google Cloud SQL, https://developers.google.com/cloud-sql/
- AFINN, http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010
- Tweepy Package, https://github.com/tweepy/tweepy