

CSE 486-DISTRIBUTED SYSTEM

Emulating PUB/SUB Distributed System Using Docker Container

REPORT

Ankit Sigroha

Rohit Balasayee

11.10.2018

Project 2

INTRODUCTION

This project aims at implementation of a PUB/SUB distributed system using a Docker container. The project demands a distributed PUB/SUB application using all nodes managing subscribers and events. In further progress of project implement a distributed rendezvous based efficient PUB/SUB system and at last deploy a docker node and a set of interacting docker images to emulate a PUB/SUB application. Publish/Subscribe is a messaging pattern that aims to decouple the sending (Publisher) and receiving (Subscriber) party. A real world example could be a sport mobile app that shows you up-to-date information of a particular football game you're interested in. In this case you are the subscriber, as you express interest in this specific game. On the other side sits the publisher, which is an online reporter that feeds a system with the actual match data. This system, which is often referred as the message broker brings the two parties together by sending the new data to all interested subscribers.

CORE IMPLEMENTATION

To present a high level view of the pub/sub distributed systems model which shows a set of subscribers who are interested in notification of events that a set of publishers publish. A centralized system with a web interface of `subscribe()`, `publish()`, and `notify()` function and also capable of implementing event generator and subscription generator. In further part of the system it requires the two Docker containers to communicate with each other.

DOCKER

Docker is a computer program that performs operating-system-level virtualization, also known as "containerization". It was first released in 2013 and is developed by Docker, Inc.

Docker is used to run software packages called "containers". Containers are isolated from each other and bundle their own tools, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and are thus more lightweight than virtual machines.

Containers are created from "images" that specify their precise contents. Images are often created by combining and modifying standard images downloaded from public repositories.

CONTAINERS IN DOCKER

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Available for both Linux and Windows-based applications, containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers that run on Docker Engine:

- Standard: Docker created the industry standard for containers, so they could be portable anywhere
- Lightweight: Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

TECHNOLOGIES USED

1. Python:- The core language for creating the basic functionality of the PUB/SUB distributed system.
2. Flask:- The python framework to handle the network essentials and host and run a server.
3. HTML, CSS:- The Front-end technologies to provide a UI to the PUB/SUB distributed system.
4. Javascript and jQuery:- To make ajax calls from the web UI to python to store the

data.

PHASE 1:

The Phase 1 requires to create a web interface where an input say in a TextBox is a small program which our team has coded in Python. The command button in this program will load the program in the Docker and execute it. The result of execution is shown as an alert box in the same window.

WORKING

In this program a web UI is created using the above said technologies i.e HTML, CSS and making an ajax call to the backend. In the textbox we enter the command in Python format to ask the program to print the data. For ex. print("This is phase 1");

The above statement is then taken at the backend by the ajax call and executed in the Docker and then sent back to the UI where the executed command results in an alert window to show output. (THE WORKING SCREENSHOTS ARE MENTIONED WITH THE COMMAND IN README FILE.)

SCREENSHOTS OF THE CODE

```
index.html
<!DOCTYPE html>
<html lang="en">
<head>
  <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
  <meta charset="UTF-8">
  <title>Proj 2 Phase 1</title>
  <center>
    <h1>DS Proj 2 Phase 1</h1>
    <p>Enter a python code in the below textarea</p>
    <p>The output of the code will be displayed as an alert </p>
  </center>
</head>
<body>
<script>

  //variable to store the output of the ajax call
  output = "";

  //function to make the AJAX call
  function make_ajax_call() {
    code = document.getElementById('code').value;

    // ajax call to python
    $.ajax({
      url: "/output",
      type: "POST",
      data: {
        code: $("#code").val()
      },
      async: false,
      datatype: 'JSON',
      success: function(json){
        //$('#demo').html(json);
        output = json;
        console.log("response" + output)
        show_response()
      },
      error: function(json,status,error) {
```

```

        console.log(error)
    }
    });
}

//function to send the output of the code as an alert message
function show_response() {
    console.log("testing" + output)
    alert(output)
}

</script>
<!-- form to get user code and submit it -->
<form onsubmit="make_ajax_call()">
    <textarea id="code" rows="30" cols="30">
    </textarea>
    <input type="submit" value="Submit">
</form>
</body>
</html>

```

RUN.py code for the phase 1:

```

run.py
#reference https://codehandbook.org/python-flask-jquery-ajax-post/
#reference http://containertutorials.com/docker-compose/flask-simple-app.html for Docker

#libraries
from flask import Flask
from flask import render_template
from flask import request
from time import sleep
import subprocess


app = Flask(__name__)

#default page to go to
@app.route('/')
def index():
    return render_template('index.html')

#run the python code inside docker
@app.route('/output', methods=['POST', 'GET'])
def output():
    #write the input code to another file called 'code.py'
    code = request.form['code']
    f = open("code.py", "w")
    f.write(code)
    f.close()
    sleep(1)
    proc = subprocess.Popen("python code.py", shell=True, stdout=subprocess.PIPE)
    script_response = proc.stdout.read()
    script_response = (script_response.decode('utf-8'))
    #return json.dumps({'status':str(script_response)})
    #return the response back to the UI
    return str(script_response)

#main function
if __name__ == '__main__':
    app.run(debug=False, host='0.0.0.0')

```



```
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
RUN mkdir /home/app
COPY . /home/app
WORKDIR /home/app
RUN ls
RUN chmod +x run.py
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["run.py"]
```

PHASE 2:

In this phase we implemented the functions namely `subscribe ()`, `publish ()`, and `notify ()` function and also capable of implementing event generator and subscription generator. The containerized docker application is capable of adding any number of:

1. Topic
2. Publisher
3. Subscriber
4. Random events
5. Random subscription

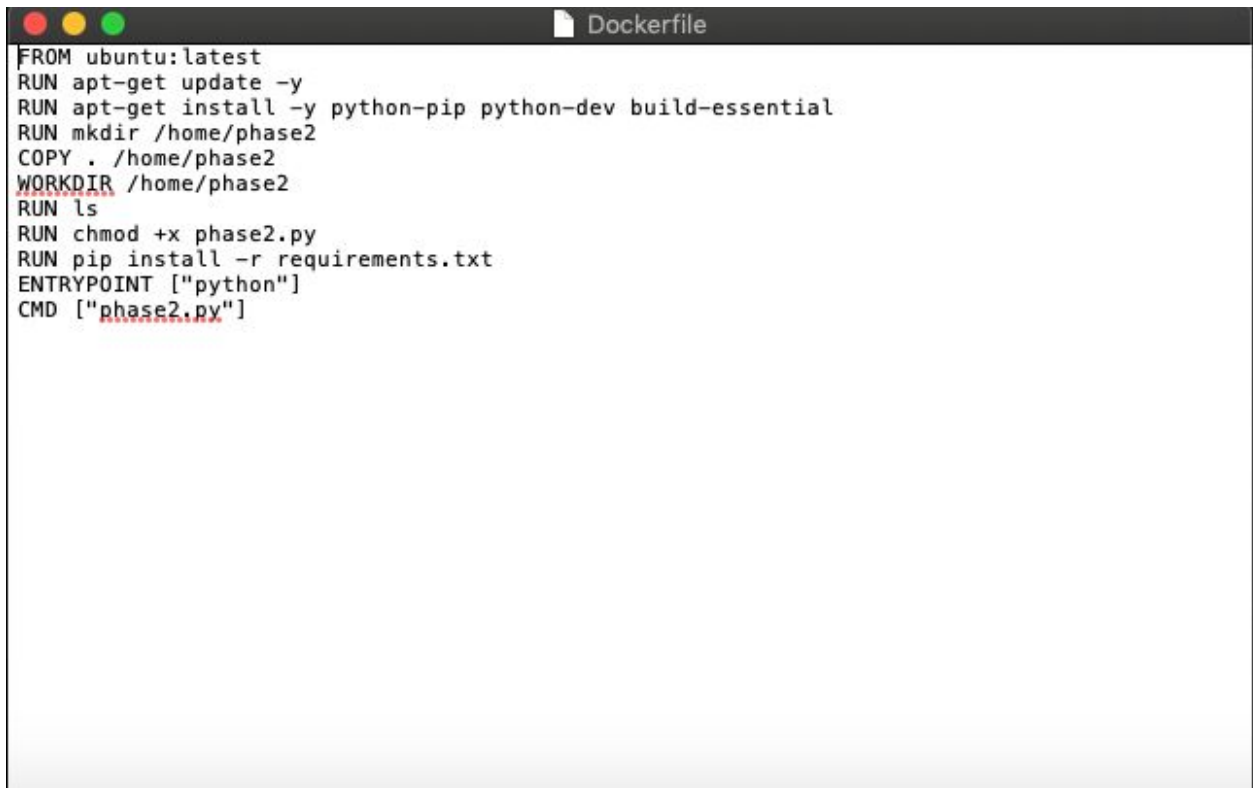
Thus the functions are coded for the backend in python language and again the same above discussed technologies are used to make the calls from the web UI and store the output, host a network etc.

A notify button is then used to allow all the subscribers to notify about the changes so made by publishers. Any number of subscribers can be subscribed to a topic and be

notified about the updates by just a click. (THE WORKING SCREENSHOTS ARE MENTIONED WITH THE COMMAND IN README FILE.)

SCREENSHOTS OF THE CODE

Dockerfile:



```
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
RUN mkdir /home/phase2
COPY . /home/phase2
WORKDIR /home/phase2
RUN ls
RUN chmod +x phase2.py
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["phase2.py"]
```

Phase2.py :

```
phase2.py x
#libraries
from flask import Flask
from flask import render_template
from flask import request
import random
import string
import json

app = Flask(__name__)

#global queue
queue = []
queue_testing = []

#hashmap to store which publisher publishes to which topic
'''pub_topic_map = {
    "pub1" : ["topic1"],
    "pub2" : ["topic2"]
}'''
pub_topic_map = {}
sub_topic_map = {}
topic_data_map = {}

#number of pub,sub and topics for random
pub_num = 0
sub_num = 0
topics_num = 0

#list of generated publishers,subscribers and topics based on user input
pubs_test = []
subs_test = []
topics_test = []

#map to store publisher,subscriber and topic connections for randomly generating and testing
pub_topic_map_test = dict()
sub_topic_map_test = dict()
topic_data_map_test = dict()
```



```

phase2.py      x
topic_data_map_test = dict()

#default page to go to
@app.route('/')
def index():
    return render_template('P2.html')

#match publisher and topics
@app.route('/pubTopic',methods=['POST','GET'])
def matchPubTopic():
    topic = request.form['topic']
    pub = request.form['pub']
    if pub in pub_topic_map.keys():
        pub_topic_map[pub].append(topic)
    else:
        pub_topic_map[pub] = [topic]
    return "success!!!"

#match subscribers and topics
@app.route('/subTopic',methods=['POST','GET'])
def subscribe():
    topic = request.form['topic']
    sub = request.form['sub']
    if topic in sub_topic_map.keys():
        sub_topic_map[topic].append(sub)
    else:
        sub_topic_map[topic] = [sub]
    return "success!!!"

#function to publish data
@app.route('/publish',methods=['POST','GET'])
def publish():
    msg = request.form['msg']
    pub = request.form['pub']
    data = msg + "," + pub
    queue.append(data)
    return "success"

```

```

phase2.py
global topic_data_map_test
topic_data_map_test= {}

global pub_num
pub_num = 0
global sub_num
sub_num = 0
global topics_num
topics_num = 0

global pubs_test
pubs_test = []
global subs_test
subs_test = []
global topics_test
topics_test = []

pub_num = int(request.form['pubsNum'])
sub_num = int(request.form['subsNum'])
topics_num = int(request.form['topicsNum'])

for i in range(pub_num):
    pubs_test.append("pub" + str(i+1))

for i in range(sub_num):
    subs_test.append("sub" + str(i+1))

for i in range(topics_num):
    topics_test.append("topic" + str(i+1))

for i in range((sub_num)):
    sub = random.choice(subs_test)
    topic = random.choice(topics_test)
    if bool(sub_topic_map_test) and topic in sub_topic_map_test.keys():
        sub_topic_map_test[topic].append(sub)
    else:
        sub_topic_map_test[topic] = [sub]
return eventGenerator()

```

Web UI:

```

P2.html x
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
  <title>PUBSUB</title>
  <style>
  body {
    background-color: lightblue;
  }

  h1 {
    color: yellow;
    text-align: center;
    font-family: courier;
    font-size: 48px;
  }

  p {
    font-family: verdana;
    font-size: 18px;
  }
  </style>
</head>
<body>

<script>

  //function to establish publisher and topic connection
  function setup_pub_topic () {

    // ajax call to python
    $.ajax({
      url:"/pubTopic",
      type:"POST",
      data:{
        topic:$("#chosenTopic").val(),
        pub:$("#chosenPublisher").val()
      },

```

```

P2.html x
    async:false,
    datatype:'JSON',
    success:function(json){
      output = json;
      console.log("response " + output)
    },
    error:function(json,status,error) {
      console.log(error)
    }
  });
}

//function to establish subscriber and topic connection
function setup_sub_topic () {

  // ajax call to python
  $.ajax({
    url:"/subTopic",
    type:"POST",
    data:{
      topic:$("#chosenTopicForSubscriber").val(),
      sub:$("#chosenSubscriber").val()
    },
    async:false,
    datatype:'JSON',
    success:function(json){
      output = json;
      console.log("response " + output)
    },
    error:function(json,status,error) {
      console.log(error)
    }
  });
}

//function to publish data to the queue
function publish () {

```

```
P2.html x
<form onsubmit="publish()">
  <p>Enter data to be published</p>
  <textarea id="code" rows="10" cols="10">
</textarea>
  <p>Enter Publisher Name to publish Data To</p>
  <input type="text" id="publisherToSendText"><br>
  <input type="submit" value="Publish">
</form>

<br><br><br><br>

<!-- Notify button to notify all subscribers of the messages they are supposed to get -->
<form onsubmit="notify()">
  <p>Click Button to notify</p>
  <input type="submit" value="Notify">
</form>

<br><br><br><br>

<!-- Enter number of publishers,subscribers,topics from user to randomly generate and test -->
<p>Press the button below to randomly generate publishers,topics and subscribers to Test</p>
<p>The number of publishers and Subscribers can be entered</p>
<form onsubmit="randomGenerator()">

  <p>Enter Number of Subscribers</p>
  <input type="text" id="subsNum"><br>

  <p>Enter Number of Topics </p>
  <input type="text" id="topicsNum"><br>

  <p>Enter Number of Publishers </p>
  <input type="text" id="pubsNum"><br>

  <input type="submit" value="Randomly Generate">
</form>
</body>
</html>
```

PHASE 3:

The third phase required to create the separate Docker images in the two containers and the publishers of one container are able to publish to other container and vice versa. Also the subscribers of one container are able to subscribe to other container and vice versa. This doesn't imply the subscriber and publisher to subscribe and publish respectively in their own containers.

So, we have created two containers running on 5000 and 5001 ports with same functionalities and in which PUB/SUB are able to communicate within and in other containers topics including the functionalities of phase 2. (THE WORKING SCREENSHOTS ARE MENTIONED WITH THE COMMAND IN README FILE.)

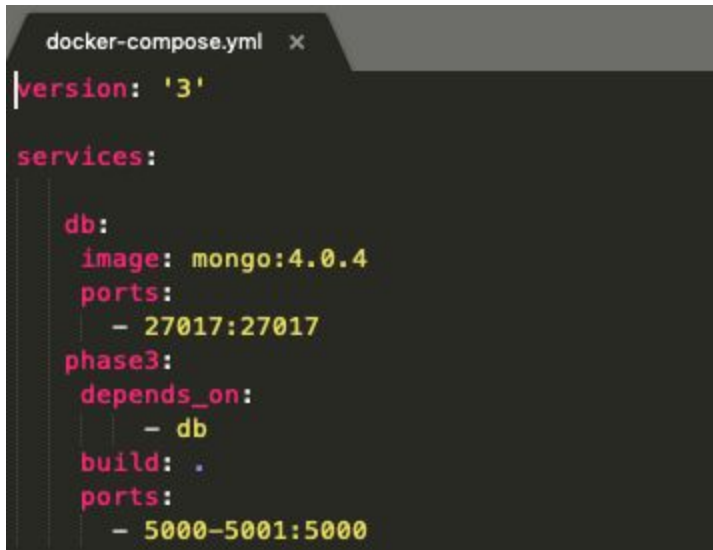
TECHNOLOGIES USED

In addition to the previously said technologies we added the following:

MongoDB Compass: To create a database for the two containers in which the data is stored and pulled on the “notify” button for the subscribers.

SCREENSHOTS OF THE CODE

Docker-Compose file:



```
docker-compose.yml x
version: '3'

services:
  db:
    image: mongo:4.0.4
    ports:
      - 27017:27017
  phase3:
    depends_on:
      - db
    build: .
    ports:
      - 5000-5001:5000
```

Phase3.py:

```

phase3.py  x
| #reference for mongo db : https://www.w3schools.com/python/python_mongodb_create_collection.asp
#libraries
from flask import Flask
from flask import render_template
from flask import request
import random
import string
import json
from pymongo import MongoClient
import sys

app = Flask(__name__)

#global variables
myclient = MongoClient('db', 27017)
mydb = myclient["phase3"]
mycol = None

#global queue
queue = []
queue_testing = []
local_subs = []

#hashmap to store which publisher publishes to which topic
'''pub_topic_map = {
    "pub1" : ["topic1"],
    "pub2" : ["topic2"]
}'''
pub_topic_map = {}
sub_topic_map = {}
topic_data_map = {}
sub_topic_map_diff = {}

#number of pub,sub and topics for random
pub_num = 0
sub_num = 0
topics_num = 0

```

```

phase3.py
topics_num = 0

#list of generated publishers,subscribers and topics based on user input
pubs_test = []
subs_test = []
topics_test = []

#map to store publisher,subscriber and topic connections for randomly generating and testing
pub_topic_map_test = dict()
sub_topic_map_test = dict()
topic_data_map_test = dict()

#default page to go to
@app.route('/')
def index():
    return render_template('P2.html')

#match publisher and topics
@app.route('/pubTopic',methods=['POST','GET'])
def matchPubTopic():
    topic = request.form['topic']
    pub = request.form['pub']
    if pub in pub_topic_map.keys():
        pub_topic_map[pub].append(topic)
    else:
        pub_topic_map[pub] = [topic]
    return "success!!!"

#match subscribers and topics
@app.route('/subTopic',methods=['POST','GET'])
def subscribe():
    topic = request.form['topic']
    sub = request.form['sub']
    same = int(request.form['same'])

    if(same):
        if topic in sub_topic_map.keys():
            sub_topic_map[topic].append(sub)

```



```

phase3.py
global sub_num
sub_num = 0
global topics_num
topics_num = 0

global pubs_test
pubs_test = []
global subs_test
subs_test = []
global topics_test
topics_test = []

pub_num = int(request.form['pubsNum'])
sub_num = int(request.form['subsNum'])
topics_num = int(request.form['topicsNum'])

for i in range(pub_num):
    pubs_test.append("pub" + str(i+1))

for i in range(sub_num):
    subs_test.append("sub" + str(i+1))

for i in range(topics_num):
    topics_test.append("topic" + str(i+1))

for i in range((sub_num)):
    sub = random.choice(subs_test)
    topic = random.choice(topics_test)
    if bool(sub_topic_map_test) and topic in sub_topic_map_test.keys():
        sub_topic_map_test[topic].append(sub)
    else:
        sub_topic_map_test[topic] = [sub]
return eventGenerator()

#main function
if __name__ == '__main__':
    app.run(debug=False,host='0.0.0.0')

```


WebUI:

```
P2.html x
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
<title>PUBSUB</title>
<style>
body {
  background-color: lightblue;
}

h1 {
  color: yellow;
  text-align: center;
  font-family: courier;
  font-size: 48px;
}

p {
  font-family: verdana;
  font-size: 18px;
}
</style>
</head>
<body>

<script>

  //function to establish publisher and topic connection
  function setup_pub_topic () {

    // ajax call to python
    $.ajax({
      url: "/pubTopic",
      type: "POST",
      data: {
        topic: $("#chosenTopic").val(),
        pub: $("#chosenPublisher").val()
      },
    },
```

```
P2.html x
    async:false,
    datatype:'JSON',
    success:function(json){
        output = json;
        console.log("response " + output)
    },
    error:function(json,status,error) {
        console.log(error)
    }
    });
}

//function to establish subscriber and topic connection
function setup_sub_topic () {

    // ajax call to python
    $.ajax({
        url:"/subTopic",
        type:"POST",
        data:{
            topic:$("#chosenTopicForSubscriber").val(),
            sub:$("#chosenSubscriber").val(),
            same:1
        },
        async:false,
        datatype:'JSON',
        success:function(json){
            output = json;
            console.log("response " + output)
        },
        error:function(json,status,error) {
            console.log(error)
        }
    });
}

//function to establish subscriber and topic connection (topic in another container)
function setup_sub_topic_diff () {
```

```
P2.html x
<form onsubmit="publish()">
  <p>Enter data to be published</p>
  <textarea id="code" rows="10" cols="10">
</textarea>
  <p>Enter Publisher Name to publish Data To</p>
  <input type="text" id="publisherToSendText"><br>
  <input type="submit" value="Publish">
</form>

<br><br><br><br>

<!-- Notify button to notify all subscribers of the messages they are supposed to get -->
<form onsubmit="notify()">
  <p>Click Button to notify</p>
  <input type="submit" value="Notify">
</form>

<br><br><br><br>

<!-- Enter number of publishers,subscribers,topics from user to randomly generate and test -->
<p>Press the button below to randomly generate publishers,topics and subscribers to Test</p>
<p>The number of publishers and Subscribers can be entered</p>
<form onsubmit="randomGenerator()">

  <p>Enter Number of Subscribers</p>
  <input type="text" id="subsNum"><br>

  <p>Enter Number of Topics </p>
  <input type="text" id="topicsNum"><br>

  <p>Enter Number of Publishers </p>
  <input type="text" id="pubsNum"><br>

  <input type="submit" value="Randomly Generate">
</form>
</body>
</html>
```

Dockerfile:

```
Dockerfile x
FROM ubuntu:latest
RUN apt-get update -y
RUN apt-get install -y python-pip python-dev build-essential
RUN mkdir /home/phase2
COPY . /home/phase2
WORKDIR /home/phase2
RUN ls
RUN chmod +x phase3.py
RUN pip install -r requirements.txt
ENTRYPOINT ["python"]
CMD ["phase3.py"]
```

CHALLENGES WE FACED:

1. Learn about the Docker and containerization.
2. To connect the web UI or the Front-end to Back-end and make the functions work to the expectations of the PUB/SUB distributed system.
3. To keep the track of data and published information and provide it to the specified subscriber at click of notify.
4. To learn about the Docker-compose and containerize the different images.
5. To setup and connect the MongoDB compass and configure it to save the data published by the publishers of different containers so as to provide to subscribers on click of notify.
6. To provide the static IP to the containers.