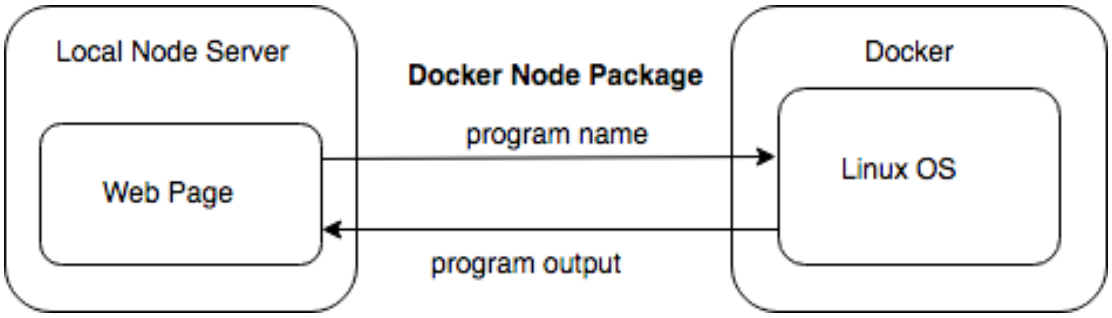
Project 2 Report

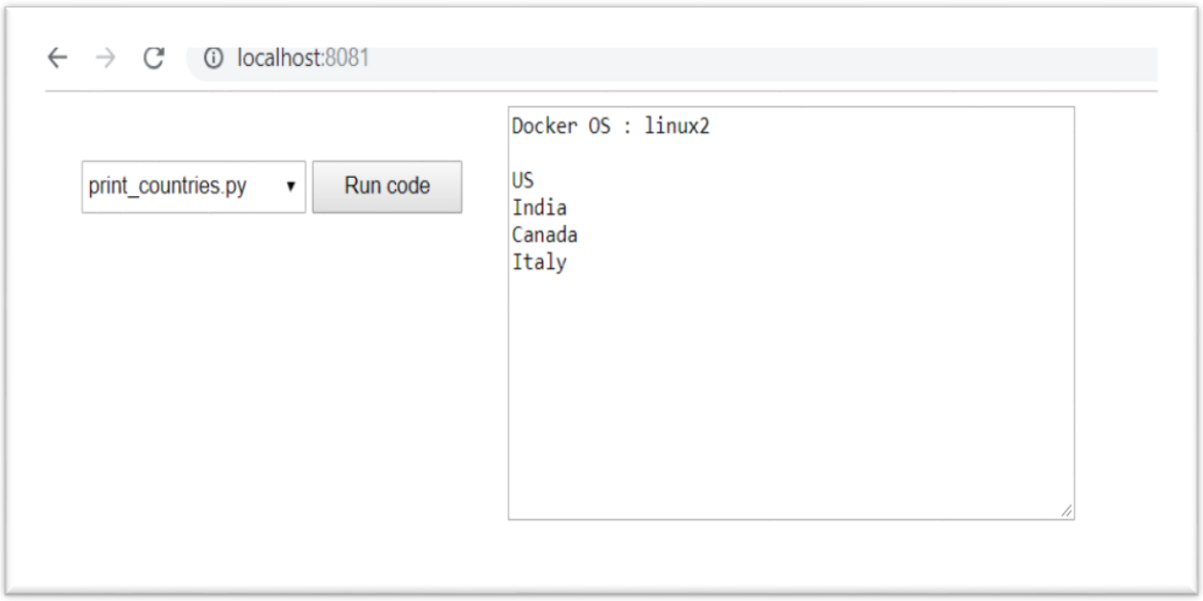
Phase1



Overview:

Input is taken from the webpage to run a particular program in docker and the docker output is displayed on the webpage

A simple webpage is hosted on local server using node, the webpage looks like this



The drop down has three options:

1) print_countries.py
2) print_colors.py
3) fibonacci_10.py

These represent three simple python programs residing in docker.

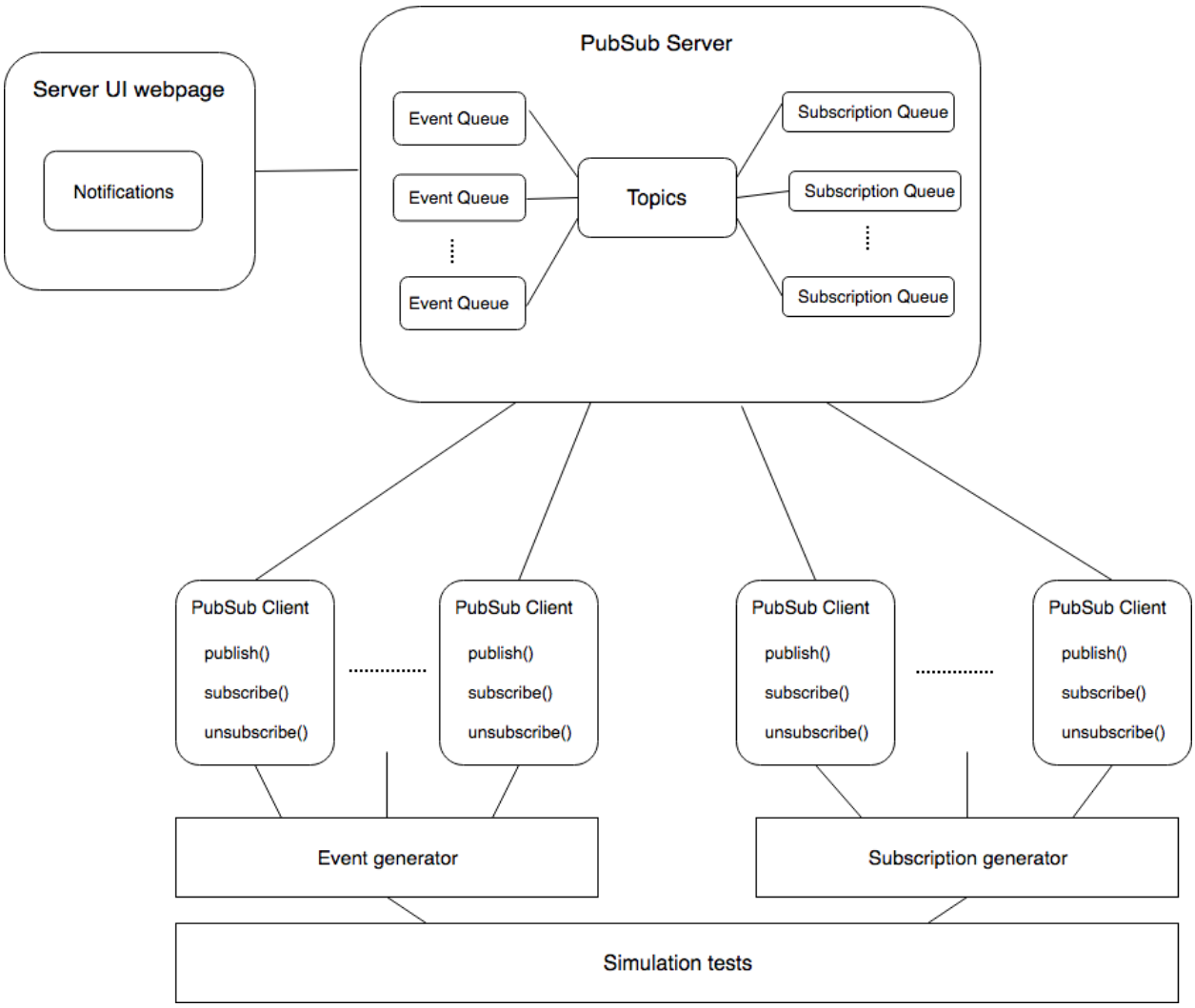
Working:

1) We use to ‘dockerode’ node package to issue docker commands using node. The docker config provided to the constructor so that a connection can be set up.
2) It assumes there exists an image ‘phase1_img’
3) When a user clicks on Run Code button. The selected option is obtained on the server.
4) The node server then issues a run command on the phase1_img using the docker.run() method of dockerode package to execute the program selected by user.

```
docker.run('phase1-img',[ "python", "./"+ file_to_run ], myStream, {}, handler)
```

5) The output of the docker is collected using a stream and then displayed on the text area as shown in the screenshot.

Phase2



Overview:

We have developed a Publisher/Subscriber System using websockets with a single central Server which perfroms the task of storing events , subscriptions, notifying subscribers about events etc.

Technologies used:

- Node.js for server
- WebSockets for 2-way communication
- HTML,CSS for UI
- Javascript for Event/Subscription Generators

Elements:

1) PubSub Server

It is a central server that can listen to messages from PubSub Clients (different from actual users) on a Websocket Server. It has two important data structures.

a) Event Queues.

It is a HashMap with key being topic name and value being the Queue of the events that were witnessed about that topic.It look like this.

Topic	Event Queue
Topic1	[event1_3 , event1_2 , event1_1]
Topic2	[event2_2 , event2_1]
Topic3	[event3_3 , event3_2 , event3_1 ,event3_0]

Whenever PubSub Server receives a publish message it stores the event in the event queue of respective topic.

b) Subscription Queues.

It is a HashMap with key being topic name and value being the Queue of the subscribers that subscribed to that topic.

Topic	Subscriber Queue
Topic1	[sub_id_1 , sub_id_4 , sub_id_21]
Topic2	[sub_id_4, sub_id_3]
Topic3	[sub_id_12 , sub_id_6 , sub_id_7, sub_id_2]

Whenever PubSub Server receives a subscribe message it stores the subscriber id in the subscription queue of respective topic.

Notifying:

Every 5 seconds notifying action takes place. It uses both the data structures. Every event of a particular topic is sent to every subscriber subscribed to that topic and then the event queue is emptied.

2) PubSub Client

It is an API that can be used to communicate to the PubSub Server and use the functions like publish() , subscribe() etc. Whenever you create an object of the this class to use in you application it generates a userID that represent that client and server will use this ID to address this client. This connects to the server via Websocket Client. All messages are sent using websocket's send method.

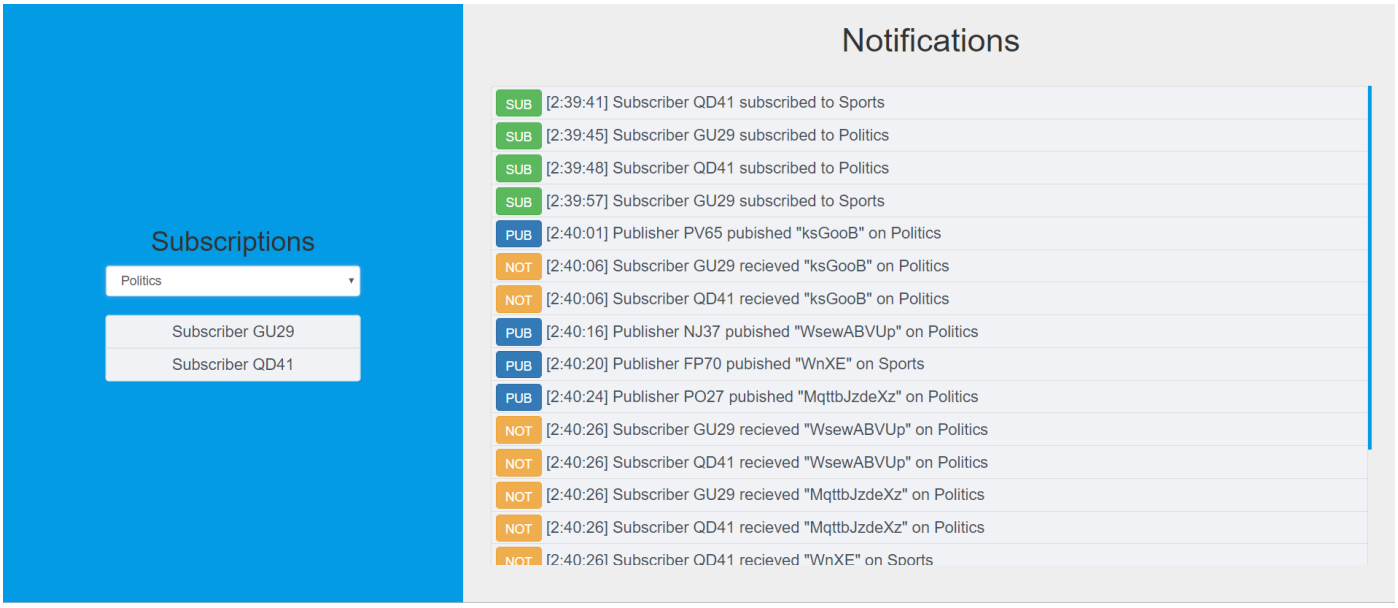
Usage:

```
myClient  = new pubsubClient()  // Create an object
myClient.publish(topic,message) // publish method
myClient.subscribe(topic)       // subscribe method
```

3) Server UI

It is a view of the server, you can see the actions server is taking, the messages it is receiving in the form of notifications. There are mainly three types of notifications

i) Subscribe
ii) Publish
iii) Notify



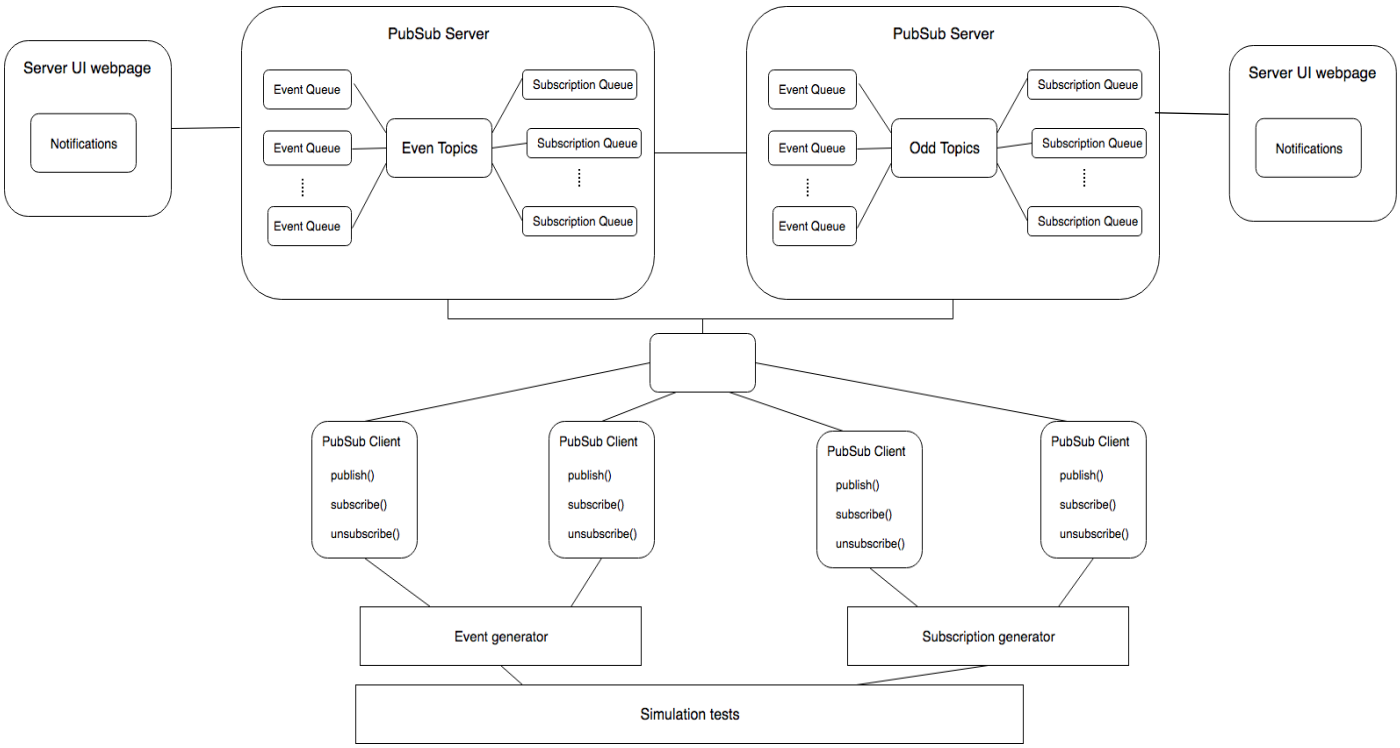
Event/Subscription Generators

For generating random events and subscriptions, We make an array of pubsubClient objects.

So we have now n pubsubClient, we choose randomly any one of them, choose a topic randomly, generate a random string as event and the publish that random event about randomly chosen topic using a random client and then we wait for random amount of time between 500 ms to 3000 ms before the next publish. We do this for 10 times. similarly, for subscription.

We generate 10 subscriptions randomly overall and 10 random events.

Phase3



Overview:

We have developed a distributed Publisher/Subscriber System with two server using websockets which performs the task of sharing load, forwarding messages, storing events , subscriptions, notifying subscribers about events etc.

Technologies used:

- Node.js for server
- WebSockets for 2-way communication
- HTML,CSS for UI
- Javascript for Event/Subscription Generators

Distributed Aspect:

The two servers share load with respect to Topics. Half of the topics are handled by server 0 and other half is handled my server 1 .

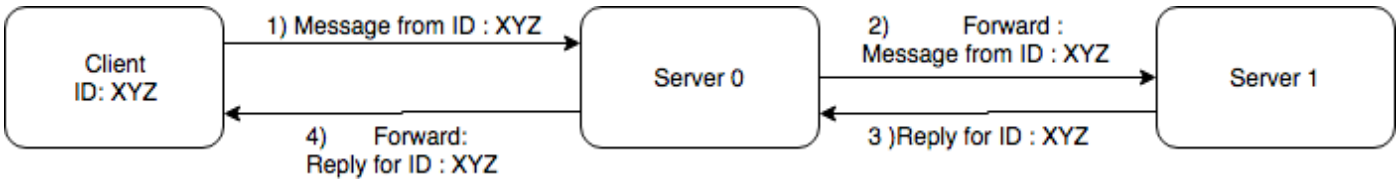
Server	Topics
Server 0	Sport , Politics
Server 1	Fashion, Science

Whenever a server 0 receives an message , it checks whether the concerned topic is its responsibility or not. If it is then it serves the request as normal, but if it is not, then the server forwards the exact same message to the other server i.e. server 1.

The server 1 thinks the message is from a client and serves the request as normal. When it comes to notifying, the server 1 notifies the server 0 because server 1 thinks server 0 is his client and the notifying message goes to server 0.

Server 0 receives this message from server 1 and realizes this is a notify message meant for for the original client, and he forwards the reply to the original client.

The clients do not know/realize that there is more than one server.



Individual working of servers is same as Phase 2, with added functionality of checking concerned topic, and forwarding requests and replies.

Event/Subscription Generation

For generating random events and subscriptions, We make an array of pubsubClient objects.

So we have now n pubsubClient, we choose randomly any one of them, choose a topic randomly, generate a random string as event and the publish that random event about randomly chosen topic using a random client and then we wait for

random amount of time between 500 ms to 3000 ms before the next publish. We do this for 10 times. similarly, for subscription.

We generate 10 subscriptions randomly overall and 10 random events.

The main difference from phase 2 is we only send messages to only one server.

This makes it easier to notice the distributed nature of phase implementation. Even though all requests are sent to one server, you can see both the server processing requests as one server forwards the messages which are not its responsibility.