Emulating Pub/Sub Distributed System Using Docker

Group Member Names:
Redwan Ibne Seraj Khan (rikhan)
Md Moniruzzaman Monir (mdmoniru)

Abstract

In this project we have implemented a publisher-subscriber(Pub/Sub) distributed system using Docker. Pub/Sub systems disseminates events to multiple recipients through an intermediary. The project was completed through incremental development in our private github repository (https://github.com/RedwanIbneSerajKhan/Pub-Sub-Distributed-System-Using-Docker) which would be made public after the submission due date. In the first phase we have worked with Docker and used it to deploy a simple app. In the next phase we implemented the publisher subscriber system using a central server. In the third phase we implemented a truly distributed pub/sub system. In making the project, we have used python mostly for communication among processes which is known as IPC. For the first phase we used a weather application that we did in Project 1 and used Docker to run it. For UI design in the second phase, we have used Javascript, JQuery, HTML and CSS. In the third phase we use multithreading and socket programming in python for implementing the distributed pub-sub architecture.

Dockerizing a Simple Web Application (Phase 1)

In phase 1, we used the web interface which we used for Project 1. There were two text boxes where the user put down the origin and the destination and the application showed the route from origin to destination along with the weather information along that route. In this phase, we used Docker to achieve the functionalities of the application.

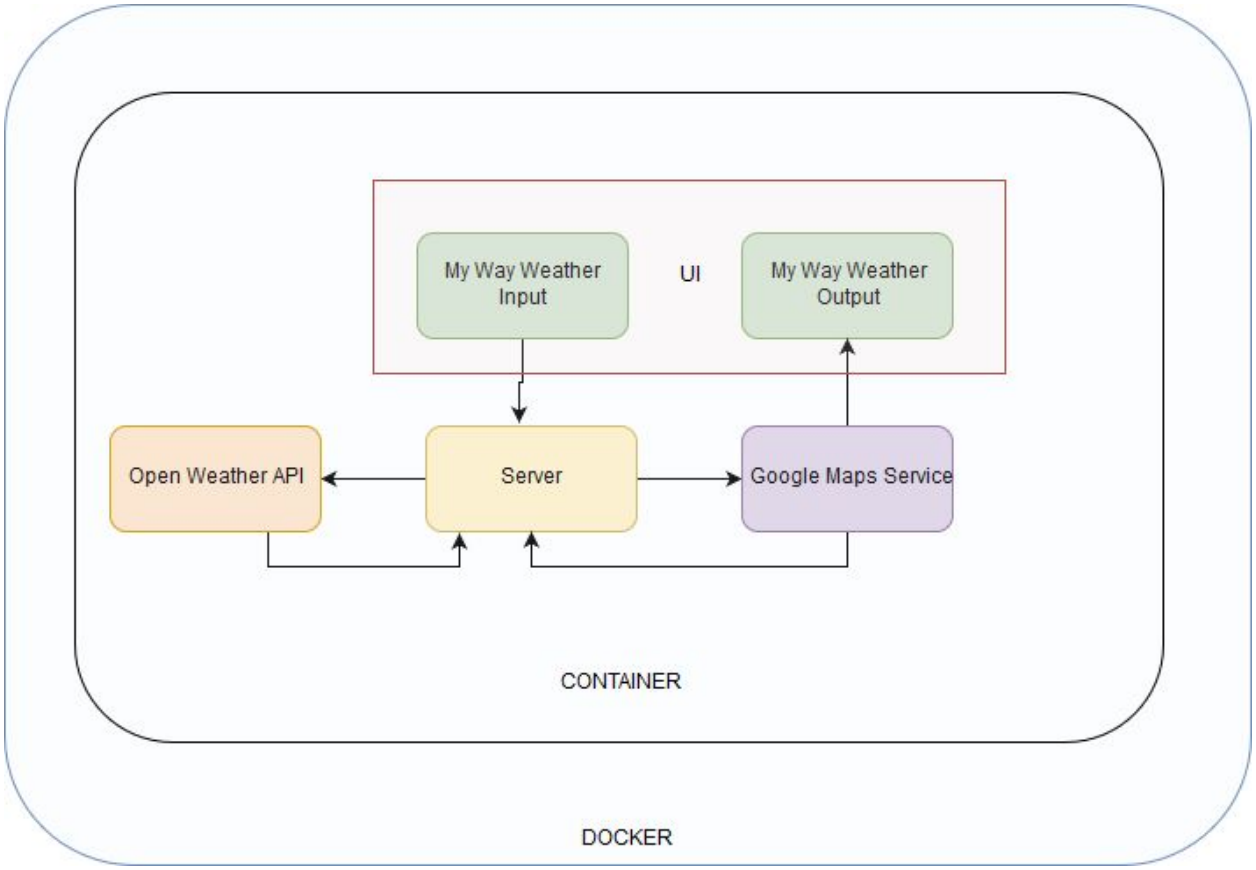


Fig. System Model of Weather App implemented through Docker

The textboxes were the origin and the destination. When we clicked on the 'search' button, the whole application started running through **docker** and gave us an output of the weather information which is displayed in the web output area. The whole application here was run in a single container.

Centralized Pub/Sub Application (Phase 2)

In this phase we implemented a **centralized** version of the pub/sub system. In the first version of the centralized pub/sub application we made the application run according to inputs from the user and in another version we generated the subscriptions and events in a **random** manner.

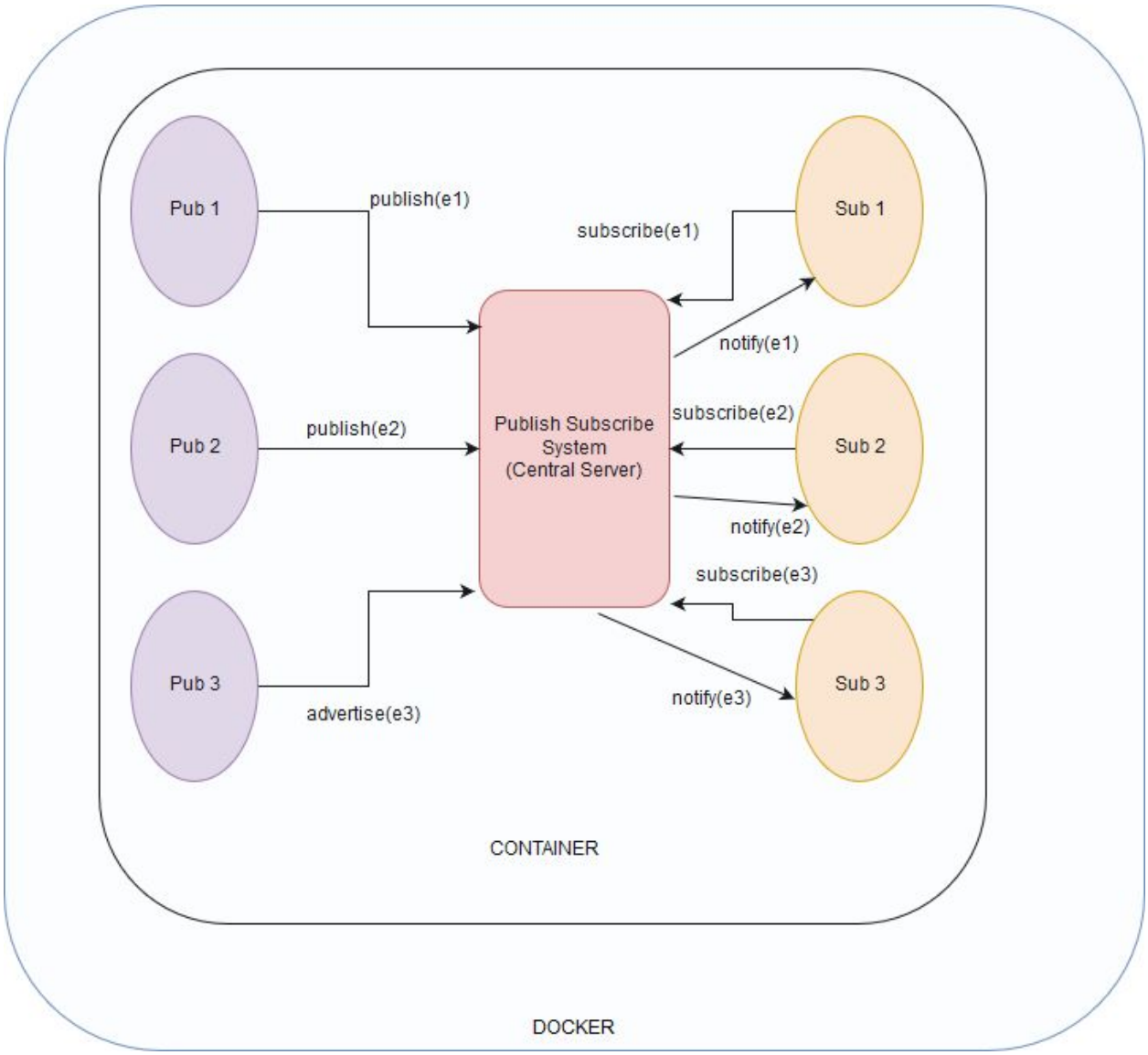


Fig: System Model of Centralized Pub/Sub Application

However, in the both the cases the main theme was constant which was using a central server to ensure the communication between the publishers and the subscribers. Publishers used functions like publish and advertise to make their events public to the clients who used methods subscribe to subscribe to the different events. The server was used to notify the subscribers about different events that the subscribers subscribed to.

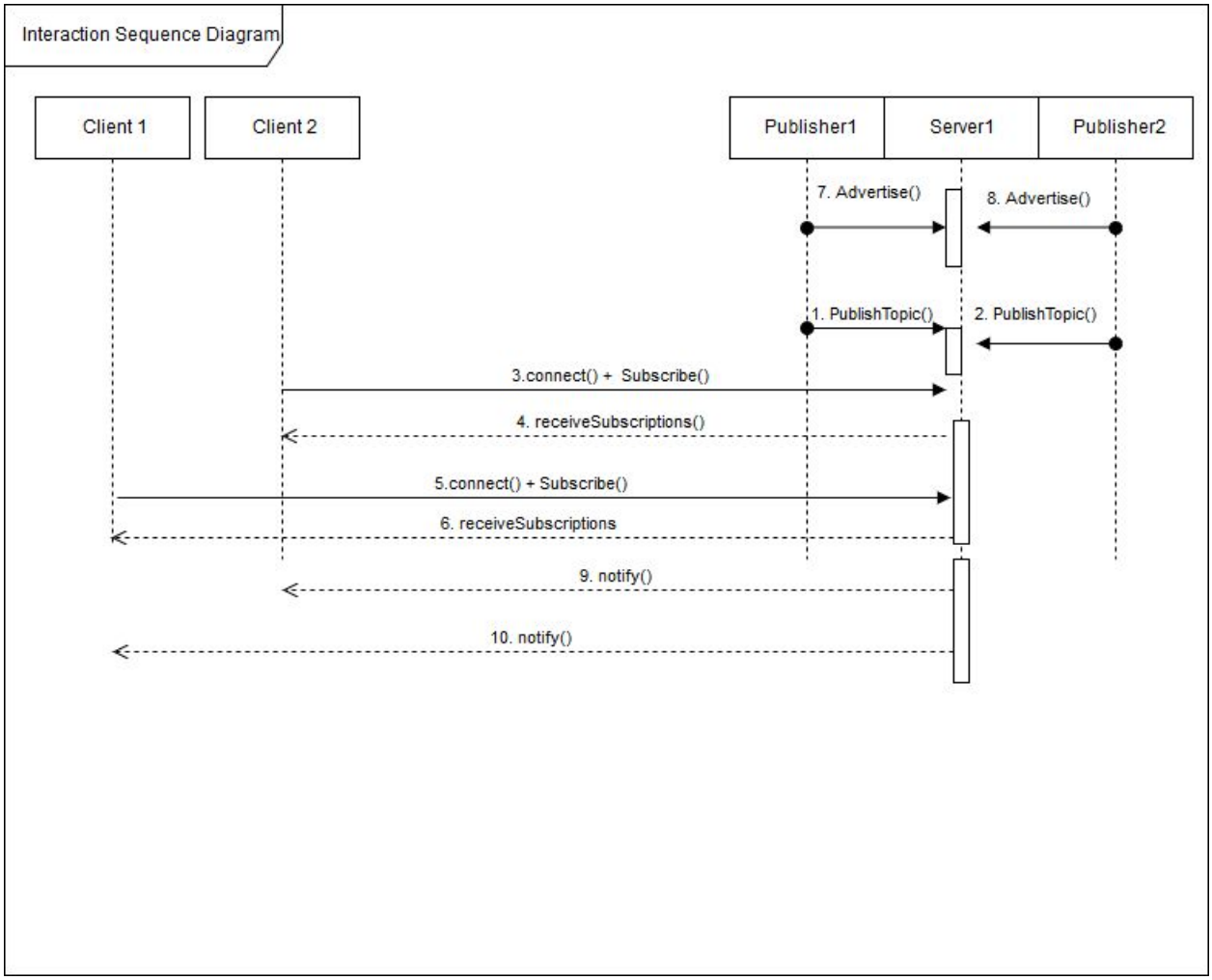


Fig: UML Interaction Sequence Diagram (Centralized Pub/Sub System)

Distributed Pub/Sub system (Phase 03)

In the third phase of the project we implemented the distributed version of the pub/sub application. Each of the clients and servers along with publishers were in **different docker containers**. The containers were bound together with a custom bridge network. The rendezvous routing algorithm was followed to implement this phase.

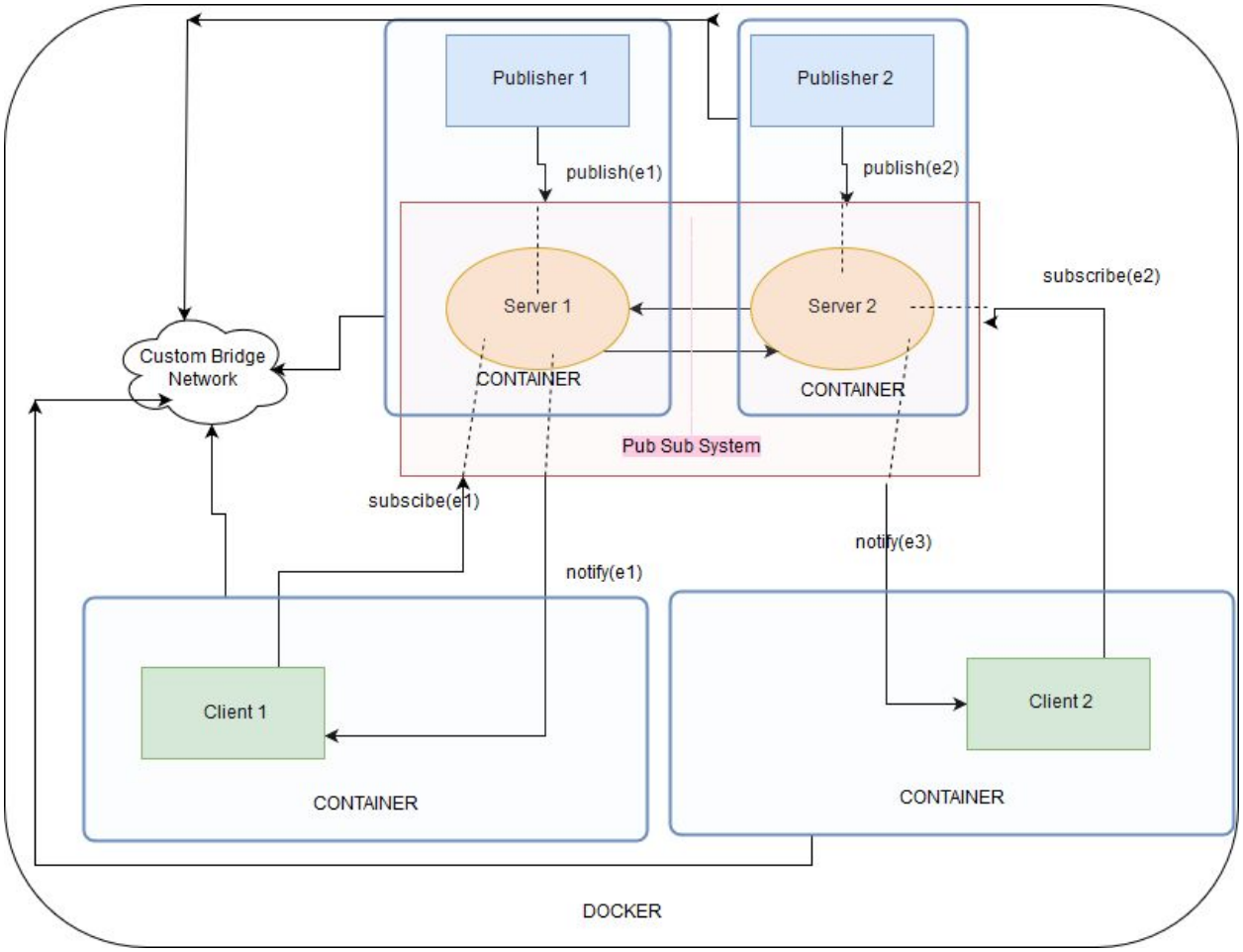
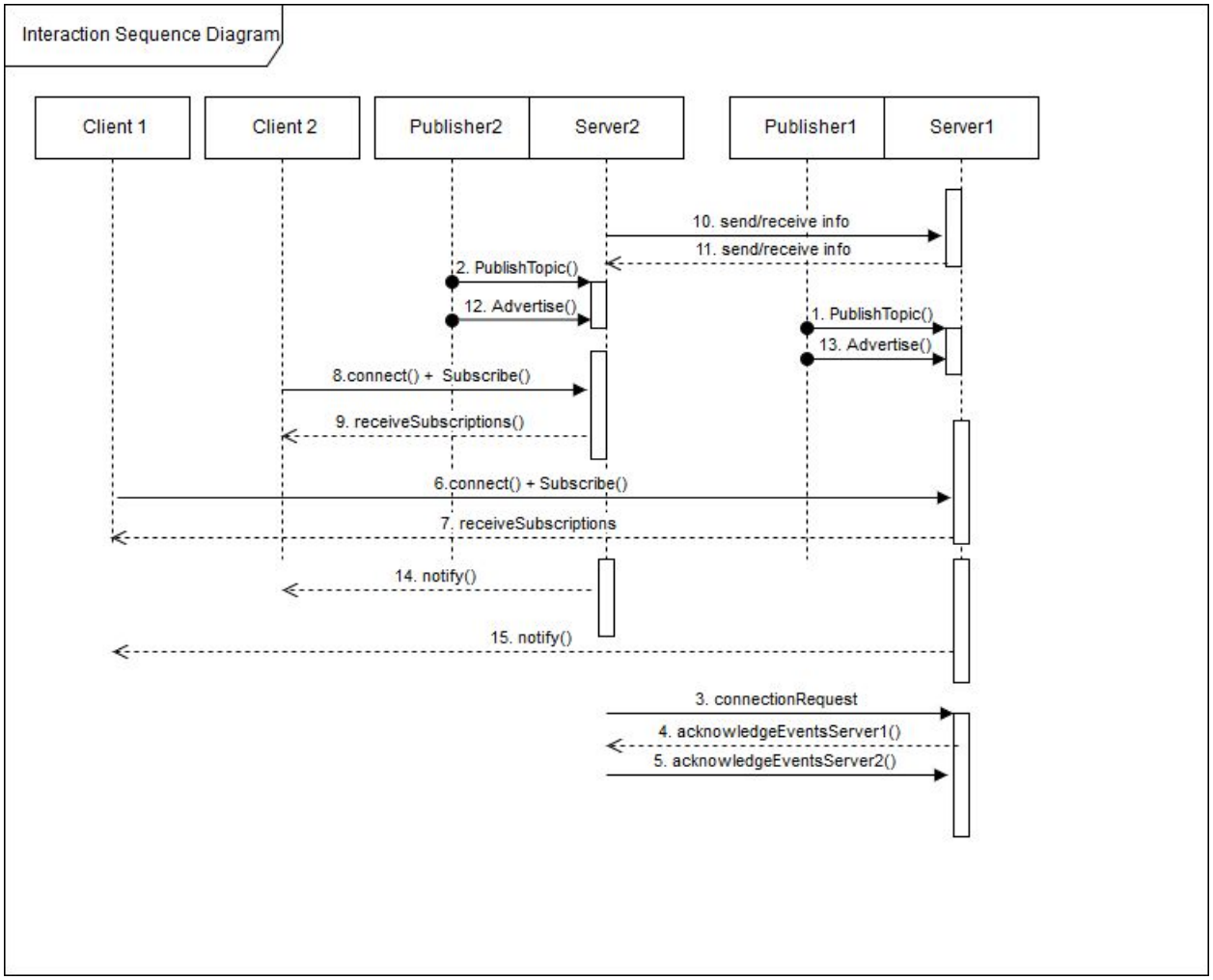


Fig: System Model of Distributed Pub/Sub System

To understand what is actually going on in this system model, let us take an example. Let for example, there are five topics - **Weather, Politics, Sports, UB, cse586**. Suppose, Pub1 decides to publish topics related to Weather, Politics and Sports and Pub2 decided to publish topics related to UB and cse586. These publishers would let the servers know which topics they want to publish. So for example pub1 uses server1 and pub2 uses server 2. So now, server 1 will handle topics related to Weather, Politics and Sports. While on the other hand, server 2 will handle topics related to UB and cse586. Each client makes a connection with any server.



UML Interaction Sequence Diagram (Distributed Pub/Sub System)

Suppose for example client1 establishes a connection with server 1. Then server 1 will assign client 1 with a random list of topics which the client will be subscribed to. Suppose client 1 has been assigned topics Sports and UB as subscriptions. Sports is handled by Server1 but UB is handled by server 2. So, how will they communicate? Here comes the distributed nature of the application. The server1 will now inform server2 that one of the clients that is connected to server 1 needs information about a topic which is handled by server2. Server2 after knowing that sends that information to server1 which then delivers the information back to client1. In this way resource is shared among the different servers and no particular server needs to bear the load of the entire data. At the same time all of the clients are getting news about different topics.

Comparison between Phase 2 and Phase 3

In phase 1 only a central server was used to handle all of the events published by different publisher and events that were subscribed to by different subscribers. This model is bound to fail when the numbers of publishers and subscribers increase. This model is not **scalable**. Changes cannot be made easily in this system.

On the other hand, in the distributed model of the pub/sub system the total load is being carried by different servers. This model is scalable and can meet the needs of large number of publishers and subscribers. Moreover as the system is distributed it is **loosely coupled** and changes can be made easily to different parts of the application.

Technologies and UI Details

For designing the user interface, HTML, CSS, Javascript and JQuery was used. Background of the site was designed by taking free images from www.unsplash.com and different icons were taken from www.flaticon.com. Both the server and the client side of the project was designed with Python. The application was implemented using Docker.