Publish Subscribe Systems
Distributed Systems

Yash Narendra Saraf
UB Person No. 50290453
MS Computer Science and Engineering
School of Engineering and Applied Science
ysaraf@buffalo.edu

Akshaya Krishnan Pattammal
UB Person No. 50491405
MS Computer Science and Engineering
School of Engineering and Applied Science
ak243@buffalo.edu

Abstract

Publish-Subscribe systems are one of the most common occurrences when we talk about notification systems. Many subscriptions based services like news applications, advertisements, financial trading applications and many more rely on such systems. So here we implement this indirect communication mode of communication in two ways, i.e. the centralized system where we have a common information processor and another distributed system across various containers. Both the models will be implemented using Docker containers. Subscribers give any input to the program. We ensure we have multiple publishers using docker.

1 Introduction

1.1 Client Server Model

The client-server model describes how a server provides resources and services to one or more clients. Examples of servers include web servers, mail servers, and file servers. Each of these servers provide resources to client devices, such as desktop computers, laptops, tablets, and smartphones. Most servers have a one-to-many relationship with clients, meaning a single server can provide resources to multiple clients at one time.

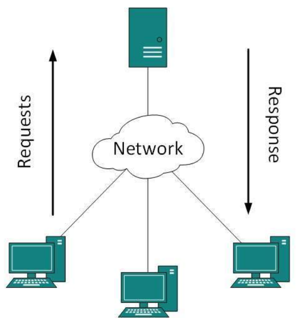

Figure 1: Client Server Model Representation

1.2 Publisher Subscriber Systems

In software architecture, publishsubscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are. This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data.

Characteristics of publish-subscribe systems - Publish-subscribe systems have two main characteristics:

Heterogeneity: When event notifications are used as a means of communication, components in a distributed system that were not designed to interoperate can be made to work together.

Asynchronicity: Notifications are sent asynchronously by event-generating publishers to all the subscribers that have expressed an interest in them to prevent publishers needing to synchronize with subscribers publishers and subscribers need to be decoupled.

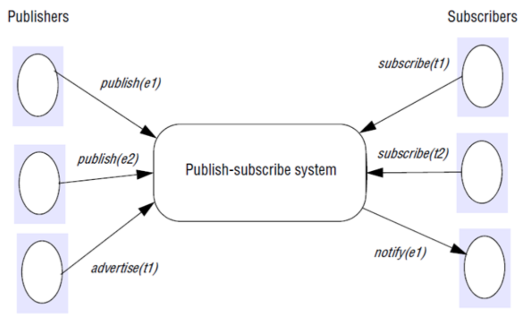

Figure 2: Publisher Subscriber Model

There are a number of schemes defined for the subscription filter model, here we are using the topic based PubSub system which is-in this approach, we make the assumption that each notification is expressed in terms of a number of fields, with one field denoting the topic. Subscriptions are then defined in terms of the topic of interest. This approach is equivalent to channel-based approaches, with the difference that topics are implicitly defined in the case of channels but explicitly declared as one of the fields in topic-based approaches. Subscribers expressing interest in the former will receive all events related to this chapter, whereas with the latter subscribers can instead express an interest in the more specific topic.

1.3 Docker

Docker is an open-source project based on Linux containers. It uses Linux Kernel features like namespaces and control groups to create containers on top of an operating system.

Ease of use: Docker has made it much easier for anyoneŁŁdevelopers, systems admins, architects and othersŁŁto take advantage of containers in order to quickly build and test portable applications.

Speed: Docker containers are very lightweight and fast. Since containers are just sandboxed environments running on the kernel, they take up fewer resources.

Docker Hub: Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an app store for Docker images.

Modularity and Scalability: Docker makes it easy to break out your applications functionality into individual containers.

Some important components related to creating an docker container

Dockerfile A Dockerfile is where you write the instructions to build a Docker image.

Docker Image Images are read-only templates that you build from a set of instructions written in your Dockerfile. Images define both what you want your packaged application and its dependencies to look like *and* what processes to run when its launched.

Docker Containers A Docker container, wraps an applications software into an invisible box with everything the application needs to run. That includes the operating system, application code, runtime, system tools, system libraries, and etc. Docker containers are built off Docker images. Since images are read-only, Docker adds a read-write file system over the read-only file system of the image to create a container.

Docker Compose Compose is a tool for defining and running complex applications with Docker. With Compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running.

2 Implementation

The implementation has been done using a flask server for all the three phases. The entire implementation is python based. For the frontend HTML Bootstrap, Javascript has been used.

2.1 Phase 1

The server is kept running in the docker. The client connects to the server using the web browser. The server processes the request and sends it back. The server compiles and executes a python code (inputted by the user). It returns the result of the execution to the client. This result is displayed in our web page.

Since we use docker, the image could be pulled easily and run in a different platform/ environment.

2.2 Phase 2

Our docker container consists of the centralized pubsub system. This system takes in the input entered by our client (either a publisher or a subscriber). It processes the input and matches the functionality depending upon the identity of the client.

Case a: Publisher - When the user is a publisher, he has to choose a topic to which he wants to publish information. The content to be published is received by the system. This content will be displayed in the subscribers end.

Case b: Subscriber - When the user is a subscriber, he can subscribe to a topic. He will be notified if there are new posts being published to this topic.

Thus, the system ideally maintains three hashtables.

1. Publisher id: Topic

2. Topic : Subscribers

3. Topic: content

2.3 Phase 3

Our docker now contains two containers. **Container a** Processes the input and displays the output.

Container b Stores the identity of the client, and the topic (subscribed/published to). In case he is a publisher, it also stores the content he published.

Here, we isolating the main pubsub system from the database. Thus, any number of pubsub systems can be created in other containers and linked to the database. This ensures decoupling of our database and the pubsub system.

To achieve this, we used the docker compose. It is a tool to maintain multiple containers, and link them accordingly.

2.3.1 UI Details:

A textbox is given, where the user must enter his userid (could be an email, or his name).

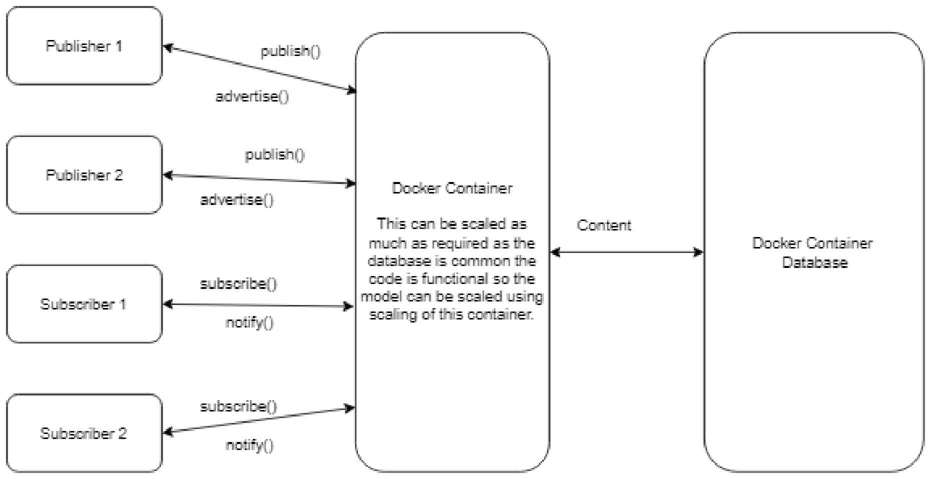
Publisher 1
publish()
advertise()
Publisher 2
publish()
advertise()
Subscriber 1
subscribe()
notify()
Subscriber 2
subscribe()
notify()
Docker Container
This can be scaled as much as required as the database is common the code is functional so the model can be scaled using scaling of this container.
Content
Docker Container Database

Figure 3: Phase 3 Model

He has to choose between Publisher and Subscriber. Once this is done, he has to choose one of the topics he wants to publish or subscribe to.

Case a: Publisher - Another drop down of topics is displayed. He has to choose the topics (from the topics he has published to) and publish content to that topic.

Case b: Subscriber - The content being published in that topic is displayed. This is the entire

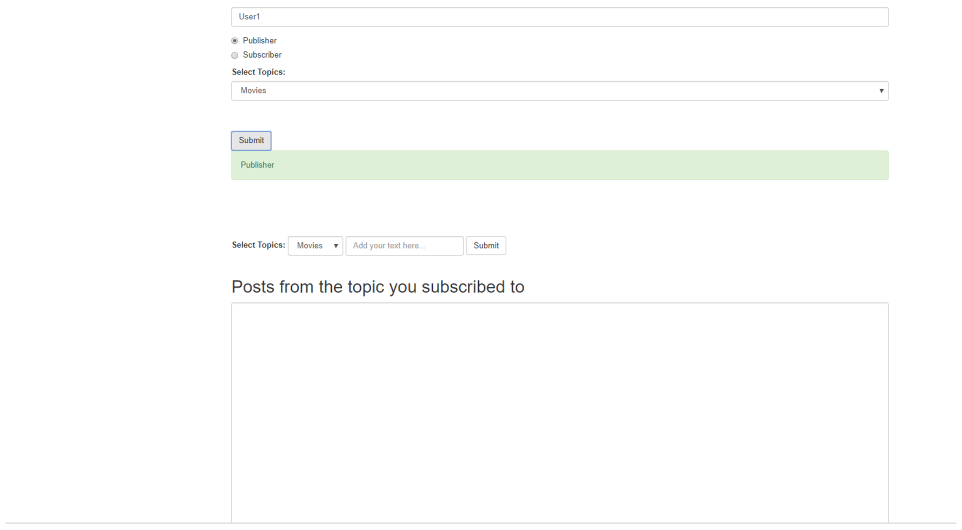

Figure 4: User Interface for Phase 3

functioning of the UI. The UI has been developed using bootstrap as it is visually appealing. Also all client server communication has been done using AJAX call so that the page retains all the information. The subscriber is sending data using the Server sent events. The SSE opens a stream of messages from server to client and notifies the client of all the messages that the subscriber has subscribed to.

The youtube link for the project has been attached.

`https://youtu.be/YKgqUSIVvYc`

3 Conclusion

Here we compared two implementations for the publish subscriber model. The first implementation is basic using the Simple client server architecture and has all the components in the single container itself. The second implementation is also based on the client server model but involves distributing the application among multiple containers so that the application can be scaled as and when necessary.

The publisher subscriber model is an efficient model for news applications, financial data passing etc. as the messages are notified to only to all the necessary subscribers. This is a model which ensures heterogeneity and asynchronicity.

References

[1] Docker Documentation, `https://docs.docker.com/`. 2017

[2] Docker Wikipedia, `https://en.wikipedia.org/wiki/Docker_(software)`. 2017

[3] Client Server model description, `https://techterms.com/definition/client-server_model`.

[4] Flask Documentation, `http://flask.pocoo.org/docs/1.0/`.

[5] Docker Blog Alicun Akkus, `https://medium.com/@caysever/docker-compose-a75171b9da48`.

[6] Distributed Systems: Concepts and Design, *Book by George Coulouris, Jean Dollimore, and Tim Kindberg*.