EMULATING PUB/SUB DISTRIBUTED SYSTEM USING DOCKER CONTAINERS

Project #2

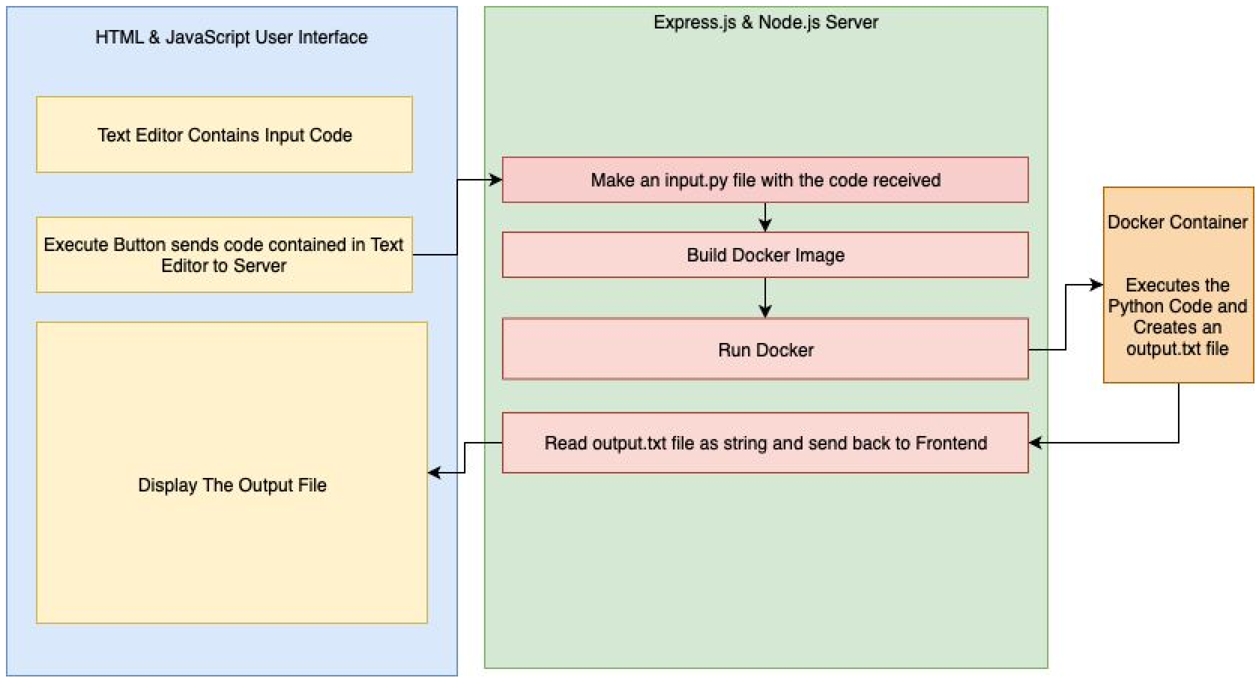
By

Jasmeet Kaur Chawla

Wei Cheng Guo

Phase I

Design and Implementation


HTML & JavaScript User Interface
Text Editor Contains Input Code
Execute Button sends code contained in Text Editor to Server
Display The Output File
Express.js & Node.js Server
Make an input.py file with the code received
Build Docker Image
Run Docker
Read output.txt file as string and send back to Frontend
Docker Container
Executes the Python Code and Creates an output.txt file

I am using plain HTML & JavaScript for the frontend, since this allows me to embed the Ace web text editor. This allows me to get the accurate input code with white space in all the right places. I used Express.js and Node.js, because Node.js has packages that allows for easy read/write of files and execution of shell commands in order to build and run the docker container.

The UI is very simple and consists of 2 halfs. The left half of the screen is the text editor where you type or copy and paste the input code. The right half displays the results of the code after execution. There is a large button spanning the entire width of the screen at the bottom of the screen that executes code when clicked.

The implementation for Part 1 is all very clearly explained in the diagram above.

Phase II

Problem Description

You are required to implement a pub/sub system in stages. The figure below gives a high level view of the pub/sub distributed systems model. It shows a set of subscribers who are interested in notification of events that a set of publishers publish.

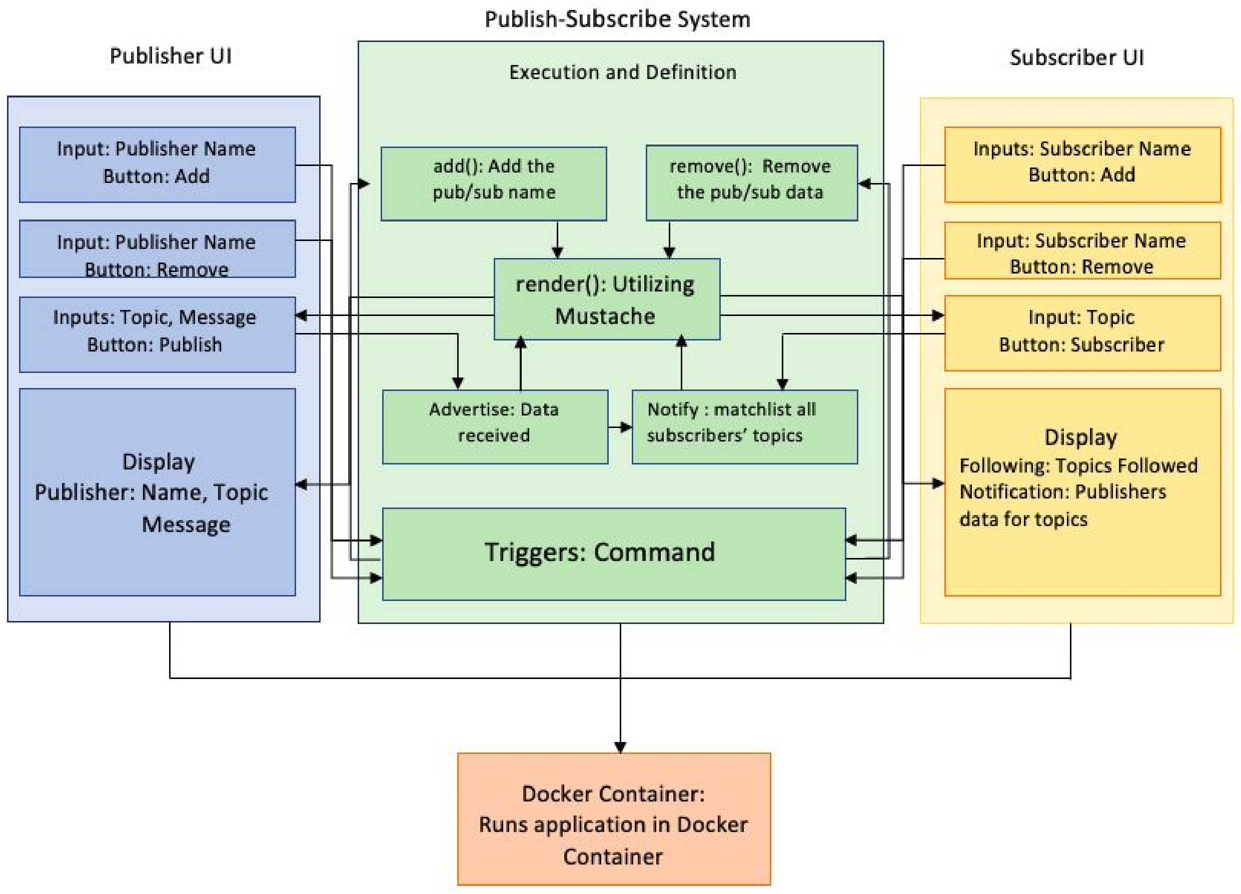
Publish-Subscribe System:

Definition: A publish-subscribe system is a system where *publishers* publish structured events to an event service and *subscribers* express interest in particular events through *subscriptions* which can be arbitrary patterns over the structured events.

Algorithm:

Topic-based : 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. The expressiveness of topic - based approaches can also be enhanced by introducing hierarchical organization of topics.
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.

Design and Implementation:


Publish-Subscribe System
Publisher UI
Execution and Definition
Subscriber UI
Input: Publisher Name Button: Add
Input: Publisher Name Button: Remove
Inputs: Topic, Message Button: Publish
Display Publisher: Name, Topic Message
add(): Add the pub/sub name
remove(): Remove the pub/sub data
render(): Utilizing Mustache
Advertise: Data received
Notify : matchlist all subscribers' topics
Triggers: Command
Inputs: Subscriber Name Button: Add
Input: Subscriber Name Button: Remove
Input: Topic Button: Subscriber
Display Following: Topics Followed Notification: Publishers data for topics
Docker Container: Runs application in Docker Container

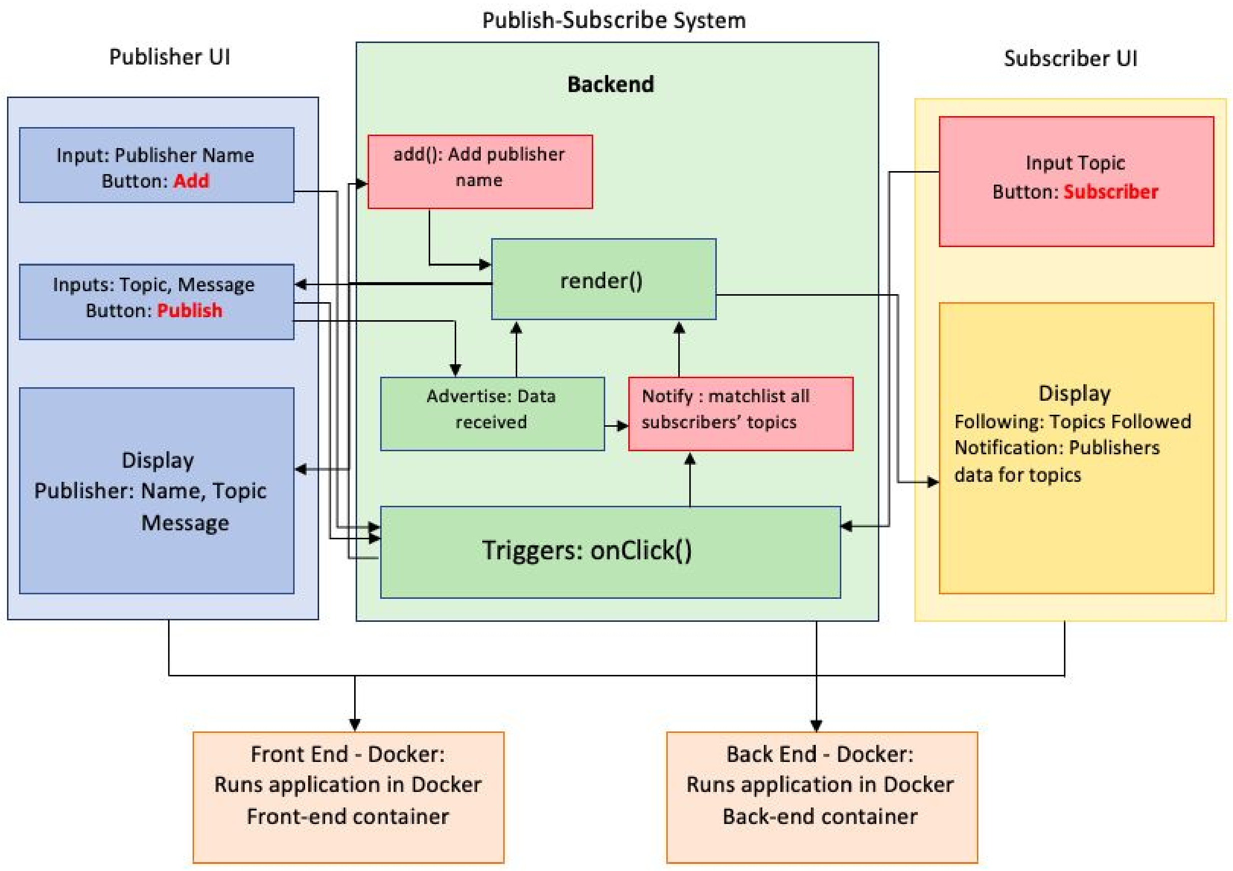
We're using jQuery and JavaScript for implementing the Publisher and Subscriber functions and Mustache to render them out to the front-end. Our code has three important parts.
The front-end : index.html
Pubsub.js : Which determines the function that is being triggered
Execution.js : Has the pubsub functionalities. add(), remove(), render(), advertise() and notify()

Implementation Issues:

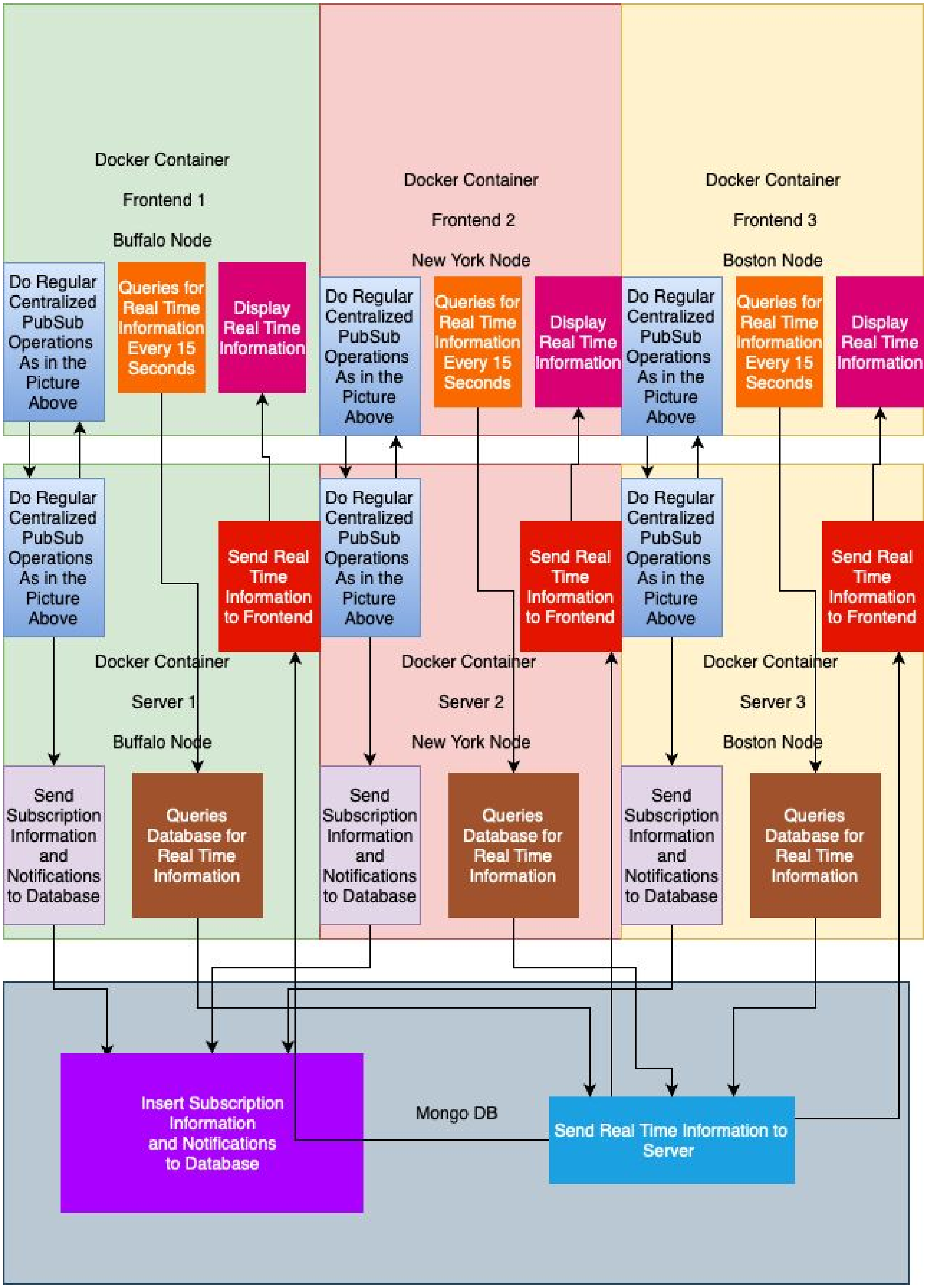
To ensure that events are delivered efficiently to all subscribers that have filters defined that match the event. Added to this, there may be additional requirements in terms of security, scalability, failure handling, concurrency and quality of service. This makes the implementation of publish-subscribe systems rather complex, and this has been an area of intense investigation in the research community.

Phase III

Design and Implementing



Phase 3 uses a frontend/server implementation of the phase 2. The new updated picture for phase 2 that is used in phase 3 shows a separate server using Express.js & Node.js that runs in a docker container. The updated phase 2 that is used in phase 3 shows a separate frontend using HTML & JavaScript that runs in a docker container. We are using a backend implementation of the Pub/Sub System for phase 3 in order to send the required information that all nodes would need to a centralized MongoDB Database. The MongoDB database represents a protocol layer that relays the information to and from nodes so that all the nodes have the same copy of the information.


Docker Container
Frontend 1
Buffalo Node
Do Regular Centralized PubSub Operations As in the Picture Above
Queries for Real Time Information Every 15 Seconds
Display Real Time Information
Docker Container
Frontend 2
New York Node
Do Regular Centralized PubSub Operations As in the Picture Above
Queries for Real Time Information Every 15 Seconds
Display Real Time Information
Docker Container
Frontend 3
Boston Node
Do Regular Centralized PubSub Operations As in the Picture Above
Queries for Real Time Information Every 15 Seconds
Display Real Time Information
Do Regular Centralized PubSub Operations As in the Picture Above
Send Real Time Information to Frontend
Docker Container
Server 1
Buffalo Node
Send Subscription Information and Notifications to Database
Queries Database for Real Time Information
Do Regular Centralized PubSub Operations As in the Picture Above
Send Real Time Information to Frontend
Docker Container
Server 2
New York Node
Send Subscription Information and Notifications to Database
Queries Database for Real Time Information
Do Regular Centralized PubSub Operations As in the Picture Above
Send Real Time Information to Frontend
Docker Container
Server 3
Boston Node
Send Subscription Information and Notifications to Database
Queries Database for Real Time Information
Insert Subscription Information and Notifications to Database
Mongo DB
Send Real Time Information to Server

Now we are inserting the relevant information including the subscribers of each topic and the messages that each subscriber should receive. The frontend is calling the function that updates the subscribers of each topic and the messages that each subscriber should receive every 15 seconds. This means that the frontend of each node will never be behind the centralized database by more than 15 seconds.

The frontend update function queries backend for the updated real time information and then backend queries the database for the updated real time information. The backend then sends the acquired information back to the frontend. The frontend checks if the new information is different from what it already have. If it’s not the same, the frontend updates its information based on the database information.

The implementation for Part 3 is all very clearly explained in the diagram above.

CONCLUSION

The implementation of centralized schemes was relatively straightforward, with the central service maintaining a repository of subscriptions and matching event notifications with this set of subscriptions. Similarly, the implementations of channel-based or topic-based schemes are relatively straightforward.

Implementation Issues:

To ensure that events are delivered efficiently to all subscribers that have filters defined that match the event. Added to this, there may be additional requirements in terms of security, scalability, failure handling, concurrency and quality of service. This makes the implementation of publish-subscribe systems rather complex, and this has been an area of intense investigation in the research community.

REFERENCES

- Distributed Systems: Concepts and Design, by G. Coulouris, J. Dollimore and T. Kingberg, Fourth Edition, Addison-Wesley, 2005.