Distributed Systems Project 1 Phase 3

This Project aims to build a one-stop application to manage multiple social media accounts viz. Twitter, Reddit etc. For phase 3, the end-to-end work flow for Twitter and Reddit is implemented for a Publisher-Subscribe model, with multiple publishers, multiple subscribers and multiple Brokers along with the rendezvous algorithm.

Design:

1. **Publisher**: This is a flask application deployed as a docker container. There is a container for every Publisher, run using the same Docker Image. Every Publisher handles a few topics. These topics can be extended to any number of topics for a given Publisher. Every publisher advertises a new topic to it's assigned Broker, using a POST request. Likewise, the publishers can de-advertise topics as well. After this, for every publisher, data is fetched using Twitter and Reddit API, and the data is given to the corresponding Broker using a POST request. Each publisher publishes the data, for all its topics, to the corresponding Broker every 10 seconds.

2. **Subscriber**: A flask application using Python, which creates a basic UI for the user to enter their inputs. This application is deployed as a docker container. There is a container for every Subscriber, run using the same docker Image. Each subscriber registers with the corresponding Broker first using a POST request. Then, the Broker returns a list of global topics (its own topics and the topics from other brokers as well) for subscription. The subscriber then subscribes to multiple topics, by sending a POST request to the Broker. Then as and when the Publisher sends the data to the broker, the Subscriber will be able to see it in the UI. The Subscribers can unsubscribe a few topics as well, by sending a POST request to the Broker. Then, the Broker won't publish the data for the unsubscribed topics.

3. **Broker**: This is a flask application deployed as a docker container. There is a container for every Broker, run using the same docker Image. The Brokers communicate with each other using the Rendezvous algorithm to send Publisher and Subscriber information.

    a. Each subscriber sends a POST request to the corresponding Broker. The broker stores the subscription only if it owns the topic, otherwise, it forwards this subscription request to its neighboring brokers (fetched using the rendezvous node). If it owns the topics that the subscribers have subscribed to, it persists the subscription data in the Database.

    b. The Broker gets the event data, new advertised topics from Publisher, and notifies each of its subscribers, after applying a filter, using POST request.

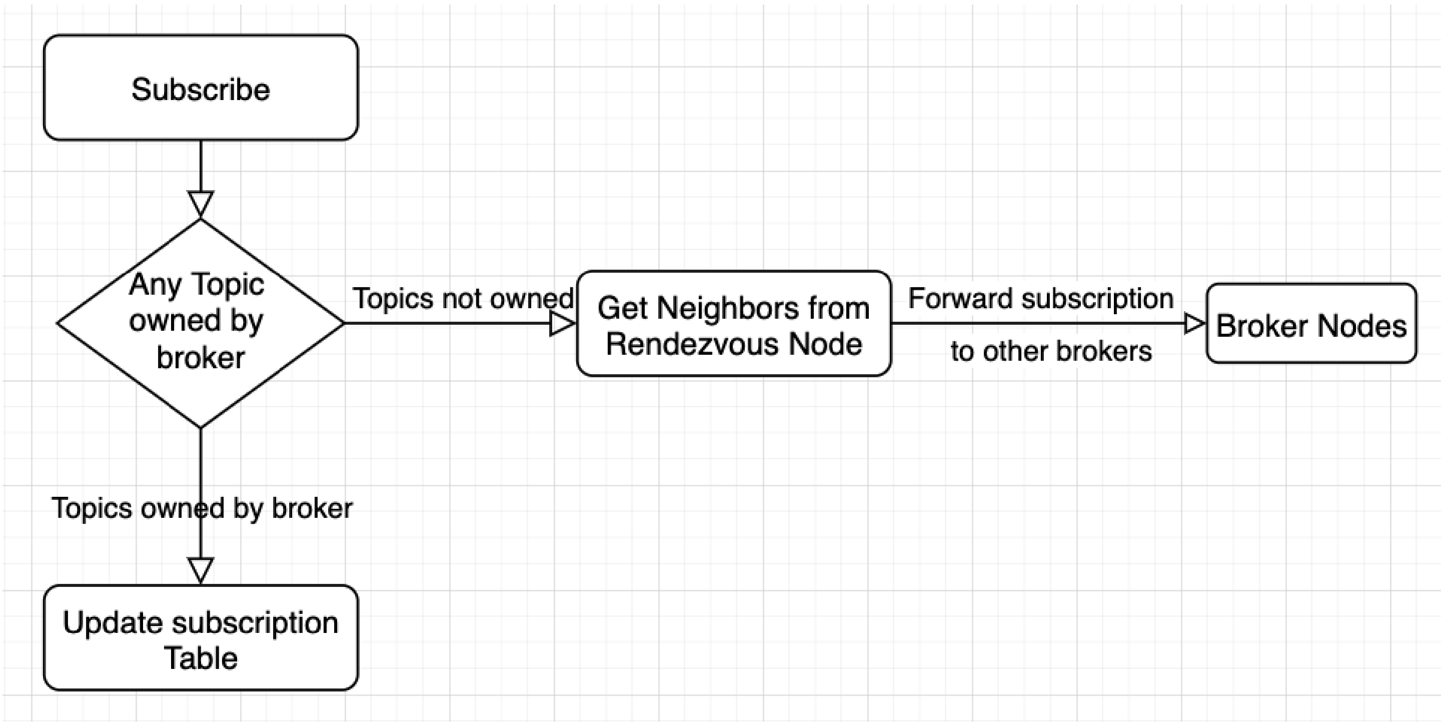
The Publisher data (events, advertised topics) is also sent to all the other brokers, so that those brokers can send the data to their respective subscribers.

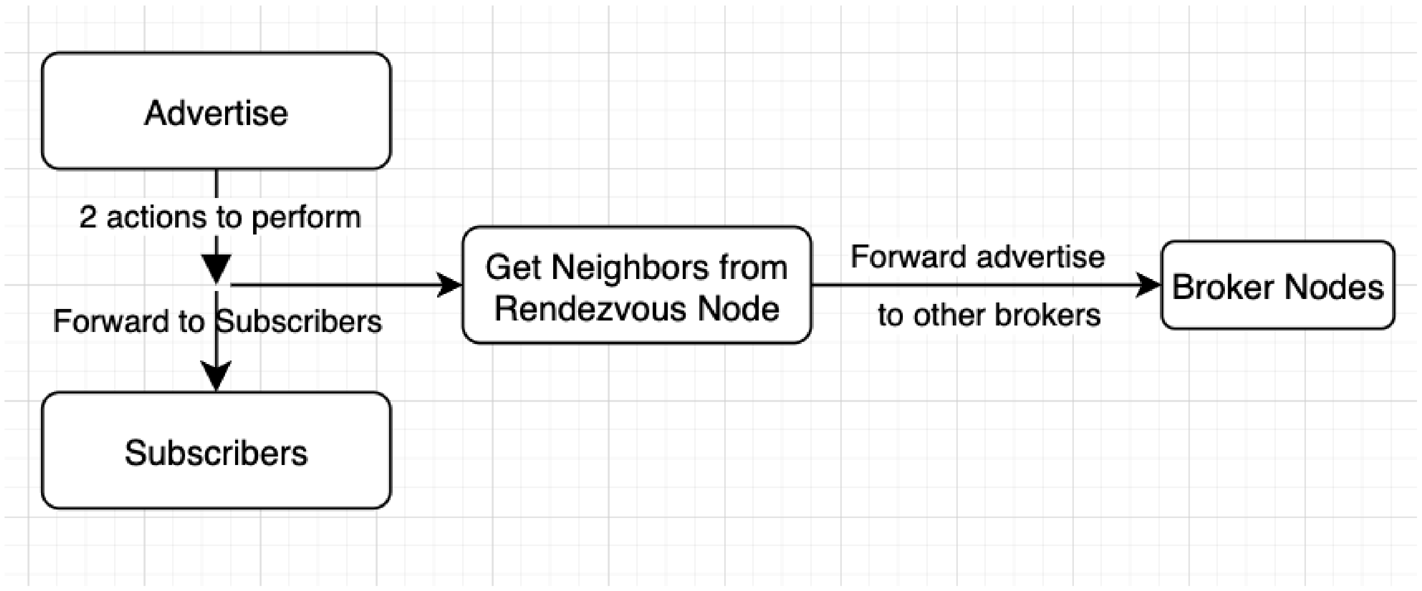
4. **Rendezvous Node**: This node facilitates the rendezvous algorithm implementation in the design. This is a flask application which is run as a docker container. The rendezvous node is responsible for storing all the available topics across brokers and for sharing the details of neighboring brokers for any given broker (which facilitates the dissemination of the subscription/publisher events between brokers).

5. **Database**: Solr NoSQL store deployed as a docker container, to index and store user query keywords and their corresponding tweets. This indexed data can be accessed from the DB by querying using the Keyword. The Brokers will push the events data along with the corresponding topics into Solr. The subscriber information is also pushed into the Database. Each broker has its own set of subscription and events data in the DB (distinguished using a field in Solr representing the broker)

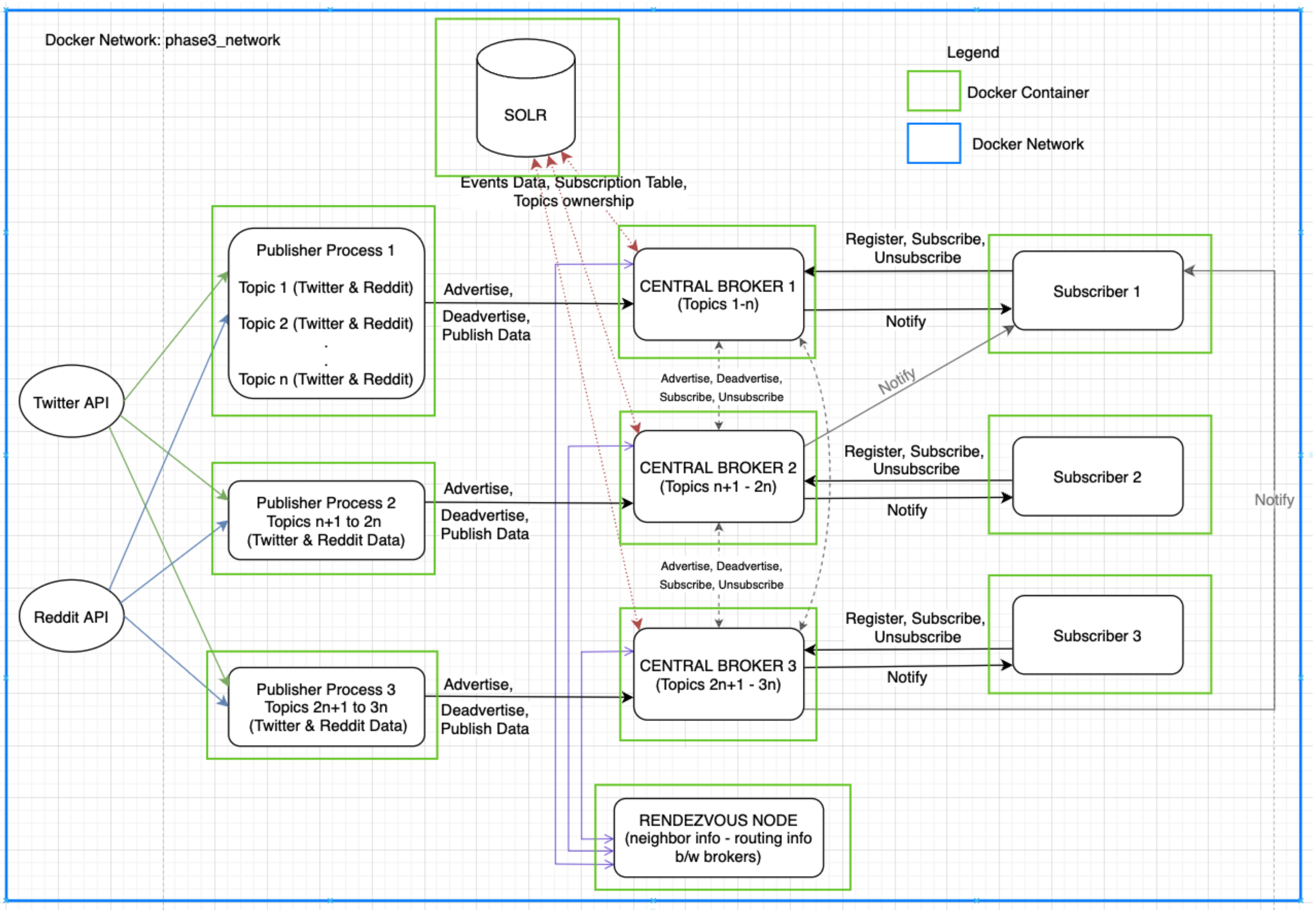
All containers are made visible to one another using docker networks. Steps to run the server and client apps are given below.

<u>Rendezvous Data Flow Diagram:</u>


Subscribe
Any Topic owned by broker
Topics not owned
Get Neighbors from Rendezvous Node
Forward subscription to other brokers
Broker Nodes
Topics owned by broker
Update subscription Table



Architecture Diagram:


Docker Network: phase3_network
SOLR
Events Data, Subscription Table, Topics ownership
Legend
Docker Container
Docker Network
Publisher Process 1
Topic 1 (Twitter & Reddit)
Topic 2 (Twitter & Reddit)
Topic n (Twitter & Reddit)
Publisher Process 2 Topics n+1 to 2n (Twitter & Reddit Data)
Publisher Process 3 Topics 2n+1 to 3n (Twitter & Reddit Data)
Twitter API
Reddit API
Advertise, Deadvertise, Publish Data
CENTRAL BROKER 1 (Topics 1-n)
CENTRAL BROKER 2 (Topics n+1 - 2n)
CENTRAL BROKER 3 (Topics 2n+1 - 3n)
Advertise, Deadvertise, Subscribe, Unsubscribe
Register, Subscribe, Unsubscribe
Notify
Subscriber 1
Subscriber 2
Subscriber 3
RENDEZVOUS NODE (neighbor info - routing info b/w brokers)

Dataflow:

1. The Brokers and the Rendezvous Node are started first, and they are now ready to accept Publishers and Subscribers. They expose REST APIs for all the required functionalities.
2. The Publishers are started, and they advertise their topics using a POST request to the corresponding Broker. (Each publisher process is configured to communicate with specific broker)
3. The Subscribers are started, and they register themselves with the corresponding Broker using a POST request.
4. All the Brokers will send all the topics (fetched using the rendezvous node) to the subscribers that they are connected to using a POST request.
5. Any Broker, on receiving “Advertise”/”Deadvertise” from a publisher, it will forward the these actions to all its subscribers and the neighboring brokers as well. Further, these neighboring brokers will act on this and forward it to their subscribers.
6. The Subscribers subscribe for the topics, and send the topics back to the Broker using a POST request. If this broker owns the topic, it’ll update its subscription table. If not, it’ll route this subscription to its neighboring brokers. Further, these neighboring brokers will use this event and update their subscription table accordingly.
7. The Publishers send the events data to the Brokers using a POST request.
8. The Broker applies a filtering logic, and finds matching topics for each subscriber in its subscription table and notifies only those events data respectively using a POST request.
9. The subscribers can unsubscribe to topics by sending a POST request to the Broker. This follows a similar workflow as the subscriber action (point 6 above).
10. The publishers can de-advertise topics by sending a POST request to the Broker, which communicates it to the subscribers, and other brokers as well.

Rendezvous Algorithm:

1. To implement the rendezvous algorithm, a rendezvous node has been utilized in the code. This node is responsible for
    a. **Maintaining a list of all available topics across brokers**: When a new subscriber registers to a specific broker, the broker will request this list from the rendezvous node and return it to the subscriber. This helps in sharing not just the topics that this broker owns, but also topics from other brokers.
    b. **Maintaining a map of neighbors for each broker**:

    i. Whenever a broker needs to forward its events/subscriptions to other brokers, it will request the rendezvous node for the neighboring node information. This helps is decoupling the neighboring node info. from the broker and effectively implement routing as required.
    ii. In current design, for each broker, all other broker nodes are shared as its neighbors.

2. Subscribe/Unsubscribe:
    a. Whenever a subscriber registers to a topic that is not handled by the corresponding broker, the broker will forward this subscribe action to its neighboring brokers.
    b. If a list of topics are shared in which few of the topics are owned by this broker, it will update its subscription table for the available topics and forward the rest of the subscription request to neighboring brokers.
    c. For Unsubscriber action, a similar flow is followed.
3. Advertise/De-advertise:
    a. On receiving topics for advertise/deadvertise, the broker will forward this information to all its subscribers and the neighboring broker nodes as well.
    b. So, each broker will be responsible for sending this action to its respective subscribers (Note: the publisher actually sends this request only to one broker)

Features:

1. **Understanding the options on the webpage:**

    ○ The list of topics to Subscribe appear as Checkboxes. Then the Social Media (Twitter, Reddit, or both) appear as Dropdowns next to the topics. A Subscriber can select multiple topics related to any social media.

    ○ Likewise, topics to Unsubscribe is displayed next, and user can select multiple topics to Unsubscribe as well. The topics to Unsubscribe will appear after the Subscriber subscribes to the topic.

2. **Independent Containers:**

    ○ The Publisher, Subscribers, Brokers, Rendezvous Node and the SOLR DB are running as separate containers. So even if the Brokers stops/crashes, there is no data loss as the data is already persisted in SOLR DB, which would still be running as a separate container. And the subscription table is persisted in the DB as well. If the Subscriber app stops/crashes, just a restart of it will suffice.

3. **Proper Abstraction of Data Flow (Time and Space Uncoupled):**

    - The Subscriber does not have direct access to the SOLR DB and would have to go through the Broker only to fetch data from SOLR DB. Likewise the Publisher is not aware of the Subscriber and can't send topics directly to it, nor does it have access to the SOLR DB. It has to go through the Broker. And this way, the Publisher and the Subscriber need not be running at the same time. So the app is Time and Space Uncoupled.

4. **Simple/Lightweight UI:**

    - The webpage has been developed using Flask to make it lightweight and user-friendly. Currently, the list of events data from Twitter and Reddit will be displayed in a scrollable fashion.

5. **Secure:**

    - Every Subscriber has to register with the Broker first, and the Broker captures the IP address of the Subscriber. This way the data is only sent to the relevant subscriber, and not to any other parties. And no one else can access the data as well, without registering first with the Broker.

6. **Scalable:**

    - Publishers run as separate docker containers. The way in which the publisher process is designed allows us to give any number of topics for a publishers process e.g. Publisher 1 can publish 5 topics (both twitter & reddit) to specified broker, Publisher 2 can publish 6 topics (twitter & reddit) to another broker.

    - Any number of central broker nodes can be spawned and the rendezvous node will be able to handle this.

    - Any number of Subscribers can be spawned and each subscriber can be registered to any broker during startup.

7. **APIs Implemented:**

    - **Publish()**

    - **Advertise()**

    - **Deadvertise()**

    - **Subscribe()**

- **Unsubscribe()**
- **Notify()**
- **Register()**

Commands to Deploy:

1. Create a docker network which will be used by all the containers. Run the below command

   docker network create phase3_network

   This creates a network with name **phase3_network**

2. Start the SOLR DB by pulling its image from docker hub. Run the below command to fire it up. (Note: SOLR needs a core to be created before pushing data, this command will precreate a core required to ingest data). Start it by attaching to the above network.

   docker run --publish 8983:8983 --net phase3_network --name solr solr solr-precreate base_core

3. Go to the RendezvousManager folder and build a docker image using the given Dockerfile. Run the below command.

   docker build . --tag rendezvous_node

   docker run --publish 8989:8989 --name rendezvous_node --net phase3_network rendezvous_node --port 8989

4. Go to the Central Broker folder and build a docker image using the given Dockerfile. This will bundle all the code and create a flask application image. Then run it using docker run and also attach it to the earlier created network. Run the below command.

   docker build . --tag central_broker

   docker run --publish 8990:8990 --name central_broker1 --net phase3_network central_broker --port 8990

docker run --publish 8991:8990 --name central_broker2 --net phase3_network central_broker --port 8990

docker run --publish 8992:8990 --name central_broker3 --net phase3_network central_broker --port 8990

4. Go to the Publisher folder and build a docker image using the given Dockerfile. This will bundle all the code and create a flask application image. Then run it using docker run and also attach it to the earlier created network. Run the below command. The Central Broker connection is mentioned as command line arguments.

   docker build . --tag publisher

   docker run --publish 8993:8991 --name publisher1 --net phase3_network publisher --broker central_broker1 --port 8991

   docker run --publish 8994:8991 --name publisher2 --net phase3_network publisher --broker central_broker2 --port 8991

   docker run --publish 8995:8991 --name publisher3 --net phase3_network publisher --broker central_broker3 --port 8991

5. Go to the Subscriber folder and build a docker image using the given Dockerfile. This will bundle all the code and create a flask application image. Then run it using docker run and also attach it to the earlier created network. Run the docker command for three subscribers, which run as three containers. Run the below command to spawn three independent subscribers

   docker build . --tag subscriber

   docker run --publish 8997:8992 --name subscriber1 --net phase3_network subscriber --broker central_broker1 --port 8992

   docker run --publish 8998:8992 --name subscriber2 --net phase3_network subscriber --broker central_broker2 --port 8992

   docker run --publish 8999:8992 --name subscriber3 --net phase3_network subscriber --broker central_broker3 --port 8992

6. Now, the RendezvousManager which is running as a container is mapped to the localhost port 8989, the Brokers are mapped to localhost port 8990, 8991 and 8992, the Publishers are mapped to localhost port 8993, 8994 and 8995, and the Subscribers are mapped to localhost port 8997, 8998 and 8999. The solr is mapped to port 8983.

Other Implementation Details:

1. **Should draw/explain which nodes handle which topics -**

   Every topic is fetched from Twitter & Reddit

$$\begin{tikzcd}
\textbf{Publisher 1} \arrow[r, "\text{Topics: Coronavirus, Cricket, Computerscience}"] & \textbf{Broker 1} \\
\textbf{Publisher 2} \arrow[r, "\text{Topics: Football, Fashion, Finance}"] & \textbf{Broker 2} \\
\textbf{Publisher 3} \arrow[r, "\text{Topics: Sports, Shopping, Stocks}"] & \textbf{Broker 3}
\end{tikzcd}$$

2. **Document should include contribution by each team member -**

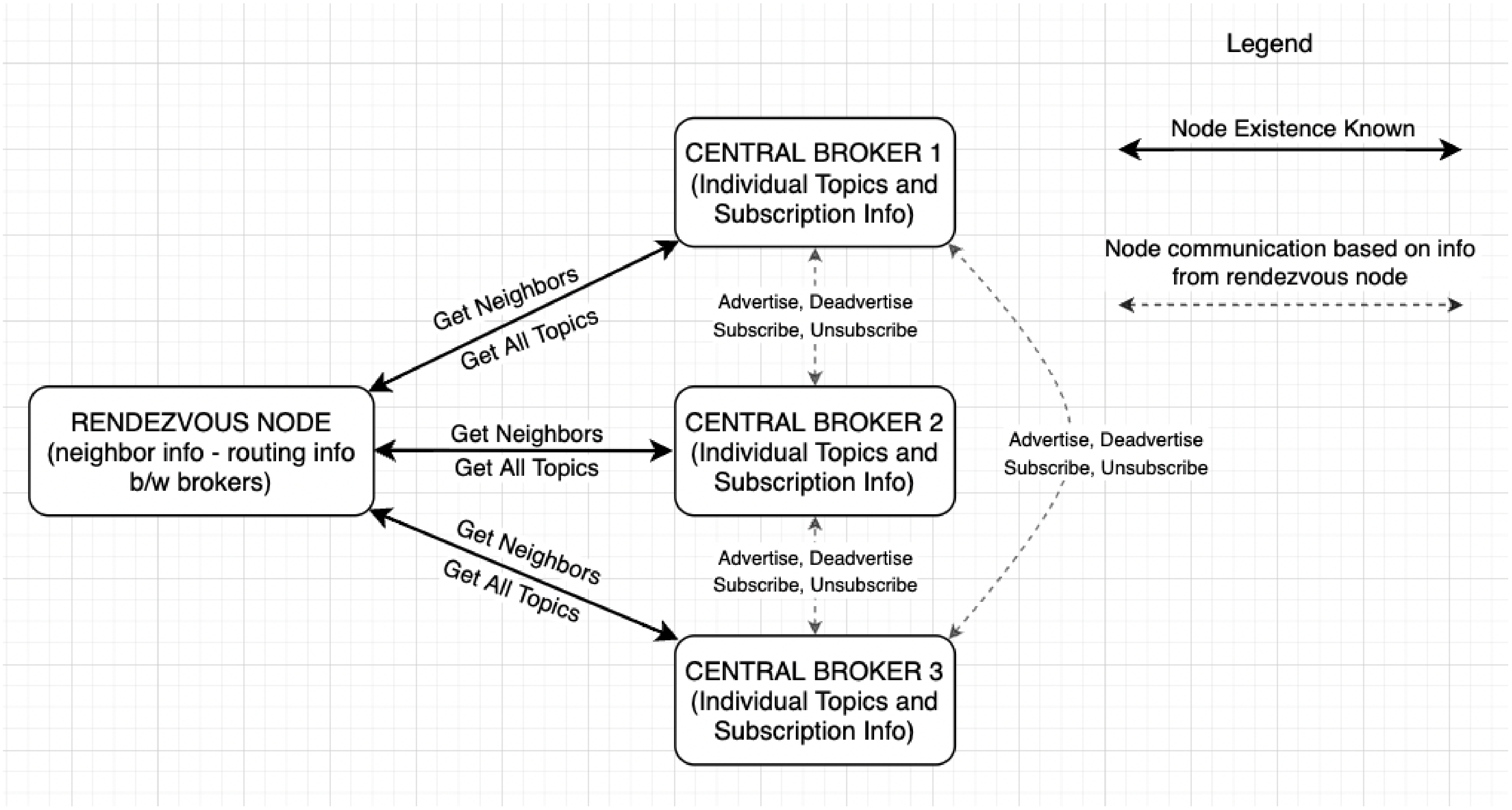
   Vishwas Nalka -

   - Implemented the connection between the Publishers, Brokers and Subscribers.
   - Implemented the code to fetch Twitter APIs.
   - Wrote the code to handle multiple Publishers, Brokers, and Subscribers.
   - Contributed to the code for the Rendezvous algorithm
   - Dockerised Publisher, Broker and Subscriber, and created Docker network to facilitate communication.
   - Implemented the text box field in the Subscriber UI to add events data for the topics.
   - Created the Sequence and Architecture diagrams, and wrote the content for the Design, and Rendezvous Algorithm, for the Report

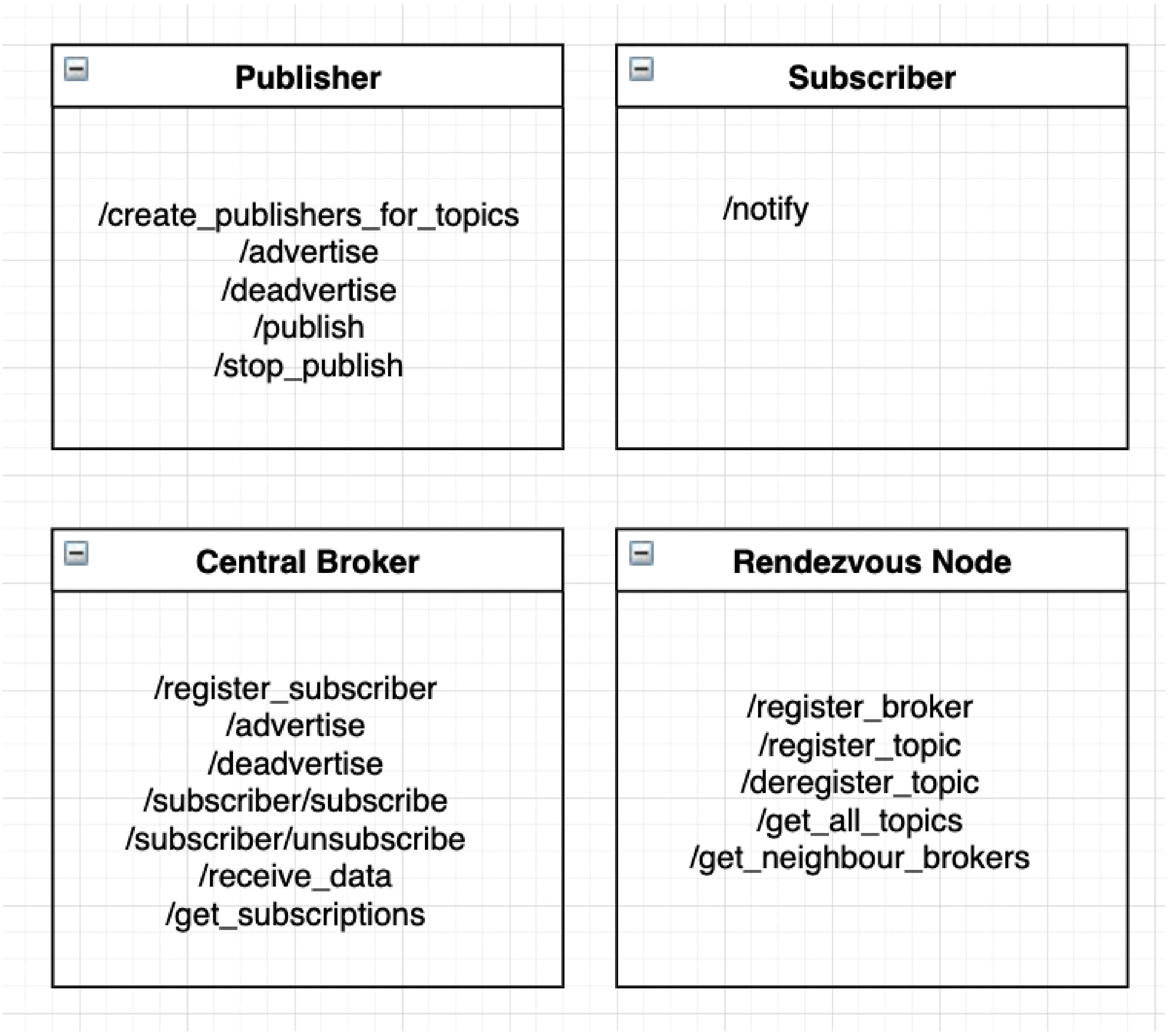
Vishal Raman -

- Implemented the code to fetch Reddit APIs.
- Implemented the connection between the Subscribers, Brokers and Publishers.
- Contributed to the code for the Rendezvous algorithm.
- Implemented the code for the Subscriber UI, using HTML, and hosted it as a flask application.
- Beautified the Subscriber UI using CSS, and added clickable checkboxes and dropdowns for Topic and Social Media.
- Handled the Subscriber Registration, and the logic for event data and topic reception by the Subscriber.
- Wrote Docker commands to run and test the entire workflow, using Postman.
- Created the diagram of the broker network
- Wrote the content for Data Flow, Features, Rendezvous Algorithm and Commands to Deploy, for the Report

3. Diagram of the broker network


Legend
Node Existence Known
Node communication based on info from rendezvous node
CENTRAL BROKER 1 (Individual Topics and Subscription Info)
CENTRAL BROKER 2 (Individual Topics and Subscription Info)
CENTRAL BROKER 3 (Individual Topics and Subscription Info)
RENDEZVOUS NODE (neighbor info - routing info b/w brokers)
Get Neighbors
Get All Topics
Get Neighbors
Get All Topics
Get Neighbors
Get All Topics
Advertise, Deadvertise Subscribe, Unsubscribe
Advertise, Deadvertise Subscribe, Unsubscribe
Advertise, Deadvertise Subscribe, Unsubscribe

4. **APIs implemented in each component**



Publisher
/create_publishers_for_topics
/advertise
/deadvertise
/publish
/stop_publish
Subscriber
/notify
Central Broker
/register_subscriber
/advertise
/deadvertise
/subscriber/subscribe
/subscriber/unsubscribe
/receive_data
/get_subscriptions
Rendezvous Node
/register_broker
/register_topic
/deregister_topic
/get_all_topics
/get_neighbour_brokers