

Tweet App Using Distributed Publisher/Subscriber System

COEN 317 - Distributed Systems - Project Report

Savio DCosta (W1610922)
Akhil Kattukaran (W1650162)
Rohit Chanda (W1649984)

Table of Contents

1. Goals & Motivation
2. Challenges
3. Literature Review
4. Project Design
 - a. Design Choices
 - b. Algorithms Implemented
 - c. Architecture & Workflow
5. Evaluation
 - a. Design Evaluation
 - b. Issues Addressed
 - i. Fault Tolerance
 - ii. Scalability
 - iii. Consistency
 - iv. Concurrency
 - v. Security
6. Implementation Details
 - a. Architecture Style
 - b. UML Diagrams
 - c. Software components and services implemented
7. Demonstration
8. Analysis of Expected Performance & Results
9. Testing Results
10. Future Scope
11. Conclusion
12. Contribution Distribution
13. References

Goals & Motivation

We created this application to visualize and understand the concepts of running a social media application in a distributed environment. On a smaller scale, we have created a Twitter-like application in which users post tweets and interact with other users on the platform in real time while concurrently working with large amounts of data.

We have implemented a multithreaded socket programming interface using Python so that a secure connection between the server and client can be made. Message handling is simplified where each process of a client will use a single thread to handle its operation.

Our motivation was to build a scalable and reliable Publisher/Subscriber system that handles message failures, fault-tolerance, and reliability aspects to provide a simple and powerful abstraction for information sources to publish messages and information sinks to subscribe to messages of interest.

Challenges

Below are some of the challenges we faced while working on this project:

- During the initial phases of the project while testing out the tweet delivery between clients, we noticed a time delay between sending and receiving a tweet. This was solved when we upgraded our database to SQLite3.
- We also initially implemented the server and client functions using a single thread which led to a performance hit when we tested it for multiple clients. Since we needed to implement this project on a large scale for multiple clients, we used a multithreaded approach to solve this issue.

Literature Review

A role-based distributed publish/subscribe system in IoT

The IoT (Internet of Things) is a new technology that has greatly expanded the Internet's scope and generated enormous volumes of information. The vast majority of earlier research has only addressed the issues of connectivity and interoperability across various entities and corporate application systems in a dynamic interactive environment. The challenges of effectively distributing sensing data across information providers and users on demand, however, have not yet been fully investigated.

This study presented a role-based distributed publish/subscribe system for the on-demand transmission of diverse sorts of data in the Internet of Things (RBDS). Grid's hierarchical routing approach is the foundation of RBDS. First, the organization of IoT data maps the subscribers to the appropriate role broker node. Second, organize broker nodes according to their respective roles. Broker nodes are divided into super nodes and common nodes in each group based on their capabilities. In order to distribute events across various groups, the super nodes are arranged in the higher Grid structure. Additionally, in order to disperse events within the same group, common nodes are arranged in a lower Grid structure. It reduces latency and increases scalability. Last but not least, dynamically determine each broker node's load depending on its subscribers and capacity. This effectively balances the overall number of subscribers on each broker node. The RBDS can increase scalability, reduce latency, and balance load efficiently, according to experimental results.

Distributed Publish/Subscribe Query Processing on the Spatio-Textual Data Stream

On the Internet, there is an enormous quantity of data that includes both visual and textual information, such as geotagged tweets. For the millions of users with diverse interests in various regions and terms, this

spatio-textual data stream provides important information. By allowing users to register continuous queries with both spatial and linguistic limitations, publish/subscribe systems facilitate efficient and effective information delivery. However, the current centralized publish/subscribe systems for spatio-textual data streams have encountered difficulties due to the exponential expansion of data size and user population. In this study, we present PS2Stream, a distributed publish/subscribe system that processes enormous spatio-textual data streams and targets users based on their registered interests.

In terms of both reducing overall workload and distributing the burden across workers, PS2Stream outperforms prior solutions in terms of task distribution. To do this, we suggest a brand-new workload allocation technique that takes into account the data's textual and spatial features. PS2Stream is adaptable since it also offers dynamic load modifications to account for changes in workload. A thorough empirical examination using a real-data commercial cloud computing platform proves the quality of our system architecture and the benefits of our strategies for enhancing system performance.

Multi-Client Transactions in Distributed Publish/Subscribe Systems

In business settings, publish/subscribe (pub/sub) systems are increasingly necessary to execute client transactions. For instance, workflow management demands that publishers and (un-)subscriptions by various clients be carried out in accordance with ACID semantics when dispatching or consolidating process instances. Such features are frequently disregarded in pub/sub systems since they are typically tuned for efficiency and scalability, which leads to unanticipated system behavior. In this work, we present a methodology for multiclient pub/sub transaction support. We describe a consistency model and isolation level necessary in the aforementioned circumstances, as well as codify ACID characteristics for pub/sub. For two different transaction types, S-TX and D-TX/D-TXNI, where a coordinator has complete static knowledge of every activity in a transaction, and D-TX/D-TXNI, where other clients' activities are dynamic and the coordinator is unaware of them, respectively, we propose three

different ways. We outline the algorithms that implement these strategies and empirically assess them in comparison to a reference mechanism that partially simulates these assurances by requiring user intervention in between operations. Our findings demonstrate that D-TX/D-TXNI is expensive and only acceptable for small setups or exceptional circumstances due to the uncertainty induced by the dynamic behavior. Contrarily, S-TX provides enhanced semantics for several applications in a scalable way without interfering with normal event routing.

Approaches to provide security in publish/subscribe systems — A review

Publish/Subscribe systems, in which publishers are in charge of introducing information into the system and subscribers are in charge of consuming it, have shown to be an efficient information-sharing paradigm for distributed systems. Researchers frequently concentrate on ideas like event forwarding and event routing for these kinds of systems. On providing security to pub/sub systems, very little effort is done. In the context of a publish/subscribe system, this study focuses on several strategies that enable confidentiality and access control.

SELECT: A Distributed Publish/Subscribe Notification System for Online Social Networks

When designing extensive notification systems for Online Social Networks, publish/subscribe (pub/sub) techniques represent an alluring communication paradigm (OSNs). Pub/sub techniques require thousands of servers spread across several data centers all over the world to handle the massive volumes of notifications generated by OSNs, incurring high overheads. We suggest SELECT, a distributed pub/sub social notification system over peer-to-peer (P2P) networks, to reduce the need for pub/sub resources. According to the social structure and user availability, SELECT arranges the peers in a ring topology and offers an adaptive P2P connection formation method, in which each peer determines the

necessary number of connections. This enables message propagation among the users' social friends with fewer hops.

The proposed technique is a useful heuristic to an NP-hard issue that transfers workload graphs to structured P2P overlays and generates a large number of messages that are near to the theoretically lowest amount. Studies reveal that when compared to the most advanced pub/sub notification systems, SELECT significantly decreases the number of relay nodes, by up to 89%. Furthermore, we show that SELECT has an advantage over socially conscious P2P overlay networks by demonstrating that communication between socially linked peers is reduced by an average of at least 64% hops while maintaining 100% communication availability even in the presence of severe churn.

Community Clustering for Distributed Publish/Subscribe Systems

An essential method to increase overall system effectiveness in a distributed publish/subscribe system is the optimal placement of clients. The client is treated as either a pure publisher or subscriber, but not as both, by current approaches like interest clustering or publisher placement. Additionally, the expense of customer migration is frequently disregarded. The assumption given by existing techniques is broken by many applications built on publish/subscribe systems, which represent clients as both publishers and subscribers simultaneously. We suggest a new community-oriented clustering technique based on the formation of client clusters that demonstrate intensive communication linkages and low client mobility costs in light of the intricate interdependence among clients. The assessment based on a publicly available data set demonstrates that our method is effective, adapts to various experimental situations, and outperforms the well-known interest clustering methodology in terms of message volume, propagation hop count, and end-to-end latency.

Project Design

Design Choices

Being dependable and having real-time outputs were essential since we decided to create a social platform application. Python was chosen as our programming language because it satisfies those standards for consistency and performance. To comprehend some of Twitter's functionalities and how they may be applied on a smaller scale, we carefully examined several of its features and design patterns.

Since we save all the data from the functions of the logged-in user in the database, SQLite has been heavily utilized and relied upon. The database is updated whenever the user makes a state change or performs an action.

This app's security is crucial. Only after creating an account and logging in may a person tweet or message another user or group. Only certain users' profiles and tweets are accessible to visitors.

Algorithms Implemented

1. Concurrency Control

It is used to manage concurrent requests to a database without impeding another user by using concurrency control. Concurrency control allows users to access a database in a multi-programmed manner while maintaining the appearance that each user is working on a separate machine.

Preventing database changes from one user from interfering with updates and retrievals from another user's database is the main technological obstacle in attaining this aim. This was put into place in our project to prevent the cross-posting of tweets from one user to another.

2. Broadcast / Multicast

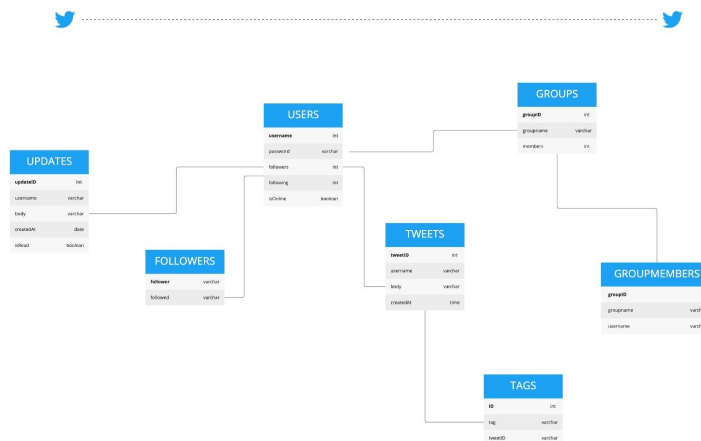
We may broadcast messages to other users in the distributed system using this approach. A message sent by one person will eventually be received by all of the appropriate participants. We used the Broadcast/Multicast method in this project, allowing each member of the system to receive tweets made by a single user.

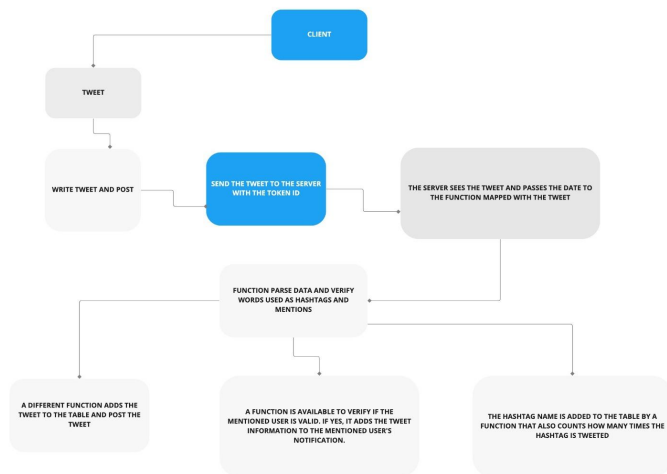
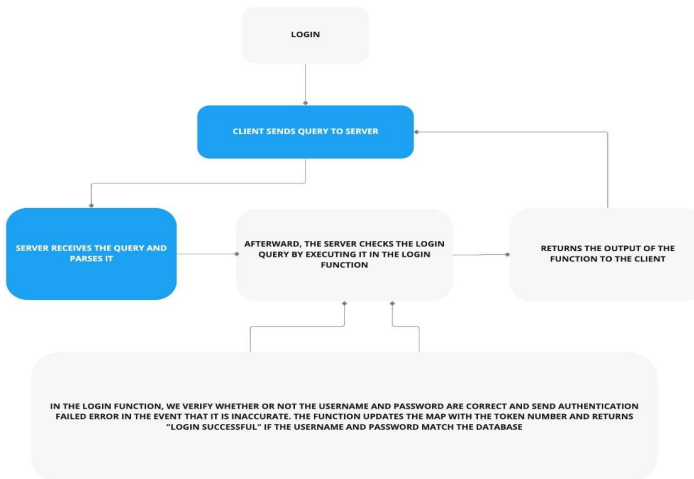
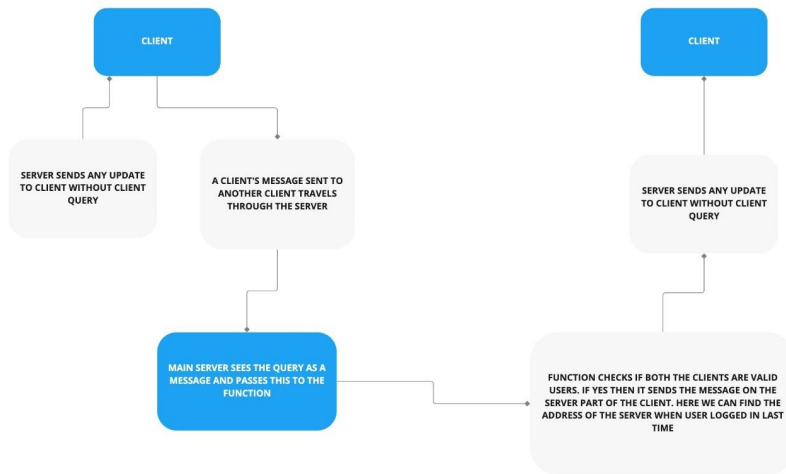
3. Replication and consistency protocols

In most cases, data is replicated to increase reliability or performance. Keeping replicas constant is one of the most difficult challenges. So, this means that if one copy is updated, the other copies must also be updated; otherwise, the replicas will no longer be identical. In our system, we replicate all data stored in 2 databases simultaneously so that it provides better protection against corrupt data and prevents loss of information in case the site goes down.

Architecture & Workflow

Following are the database architecture and workflow of the functionalities we have implemented in this project





Evaluation

Design Evaluation

This application is a multithreaded concurrent client-server implementation which essentially means that the application is capable of handling multiple concurrent requests coming in from different clients at the same time. Each client uses a process to communicate with the server which is then executed as individual threads to keep concurrent execution at its best. Each client can run the application and connect to the server using the command prompt. A connection is established between the server and client using sockets and the requests are processed accordingly.

Another important factor that needed to be considered was openness. Hiding the background complexities and presenting just the essential ones to the client in order to keep the application as user-friendly as possible was one of the main factors that we had to cater to. We designed the application in such a way that the user can operate and carry out different tasks and functionalities in the application without having to understand the complexities running in the background.

Issues Addressed

While designing the application there were some key aspects that we had to keep in mind. These aspects are like the pillars of any distributed system and should be catered to in order to build a successful application running on the principles of distributed systems. Following are some of the key aspects to which we paid special attention:

Fault Tolerance

The application should be able to handle failures, if any, and be able to still perform as it is. This is the reason we chose multithreading so that the server can jump from one thread to another as and when needed. The server waits for a client to accept any request. If the client is not responding then it moves on to the next client or request

and executes it without stopping.

Scalability

Our system is flexible enough to operate with many customers without compromising any functionality. Because of its scalable nature, we can manage a significant spike in usage. We may instantly switch to the other thread and resume the desired functionality if a process doesn't reply.

Consistency

The information is consistent throughout our application. Because we utilize different databases to store all the data in real time, the data is reliable and consistent across the whole program.

Concurrency

This program can handle concurrency; with the aid of multi-threading, we can manage several concurrent customers at once. Multiple threads are active at once to handle diverse functionalities.

Security

Any application working on the principles of distributed systems should be secure and stable. Any unregistered user should not be able to manipulate the data unless he or she has created an account in the database and registered into the system. A guest user can, however, view the feed and public posts made by other users.

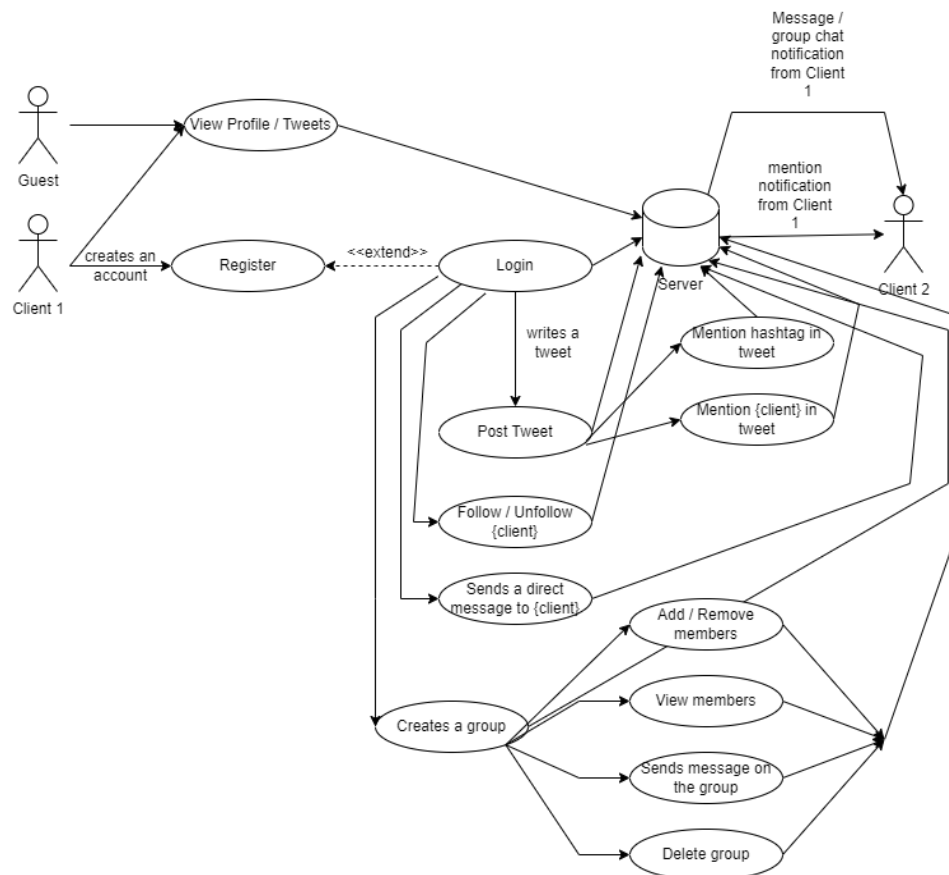
Implementation Details

Architecture Style

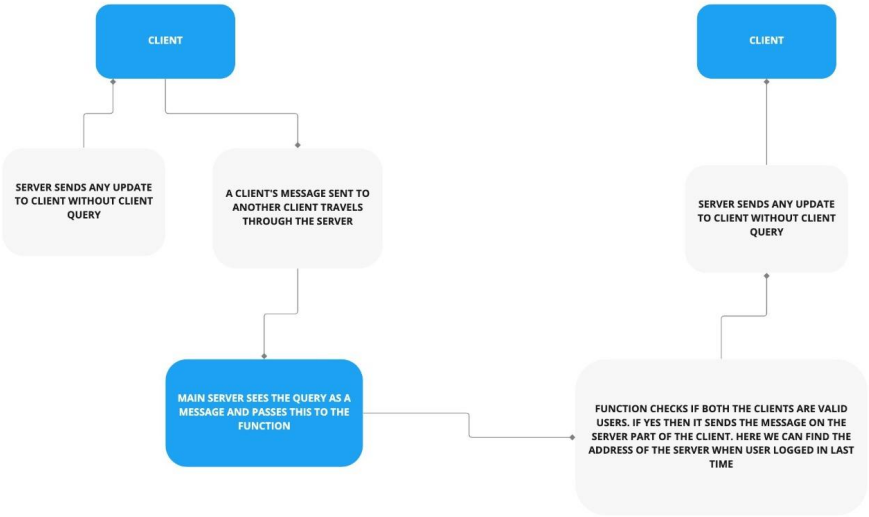
The entire application is basically built as two tiers: the server side and the client side. The client-side can have multiple clients which essentially means that every client registering into the application is basically a system that is connecting to the server. The client subscribes to the server and posts requests to the same. The server then performs execution on those requests and publishes the information to all the other clients present in the system.

UML Diagrams

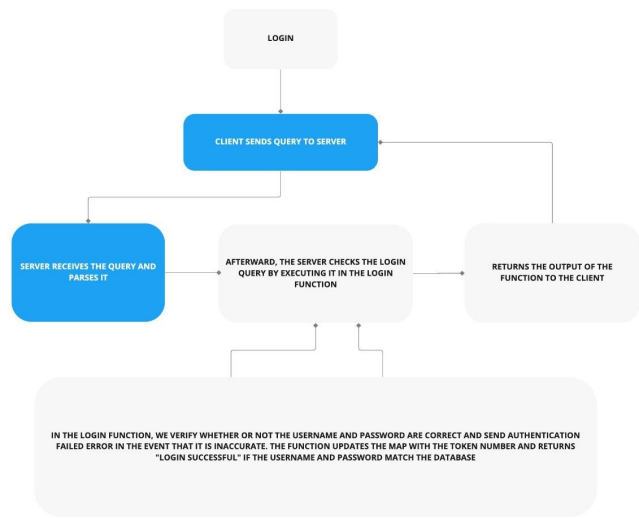
Use Case Diagram



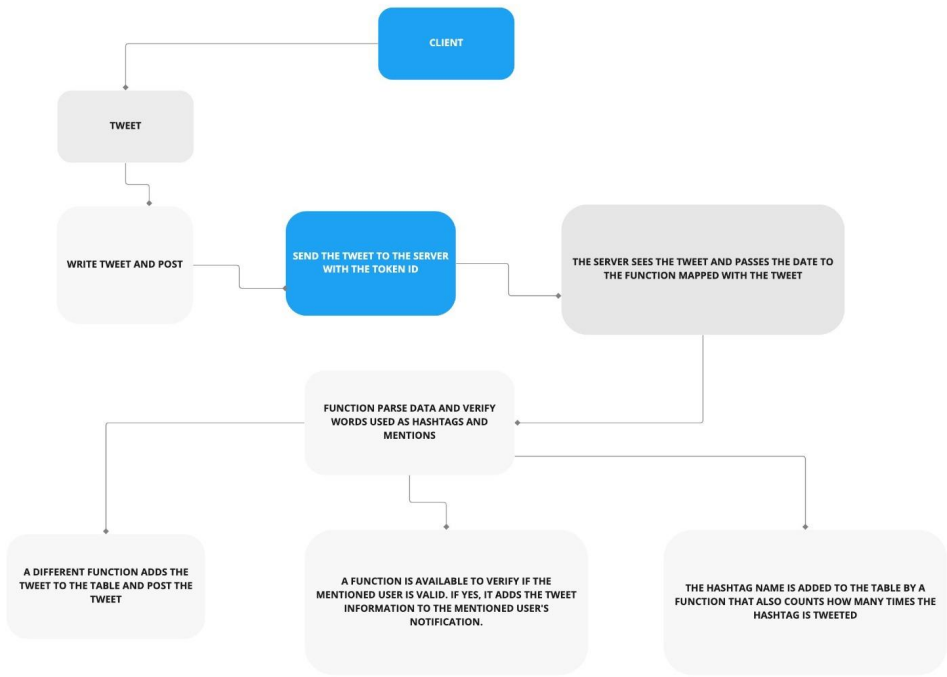
Component Diagram



Communication Diagram

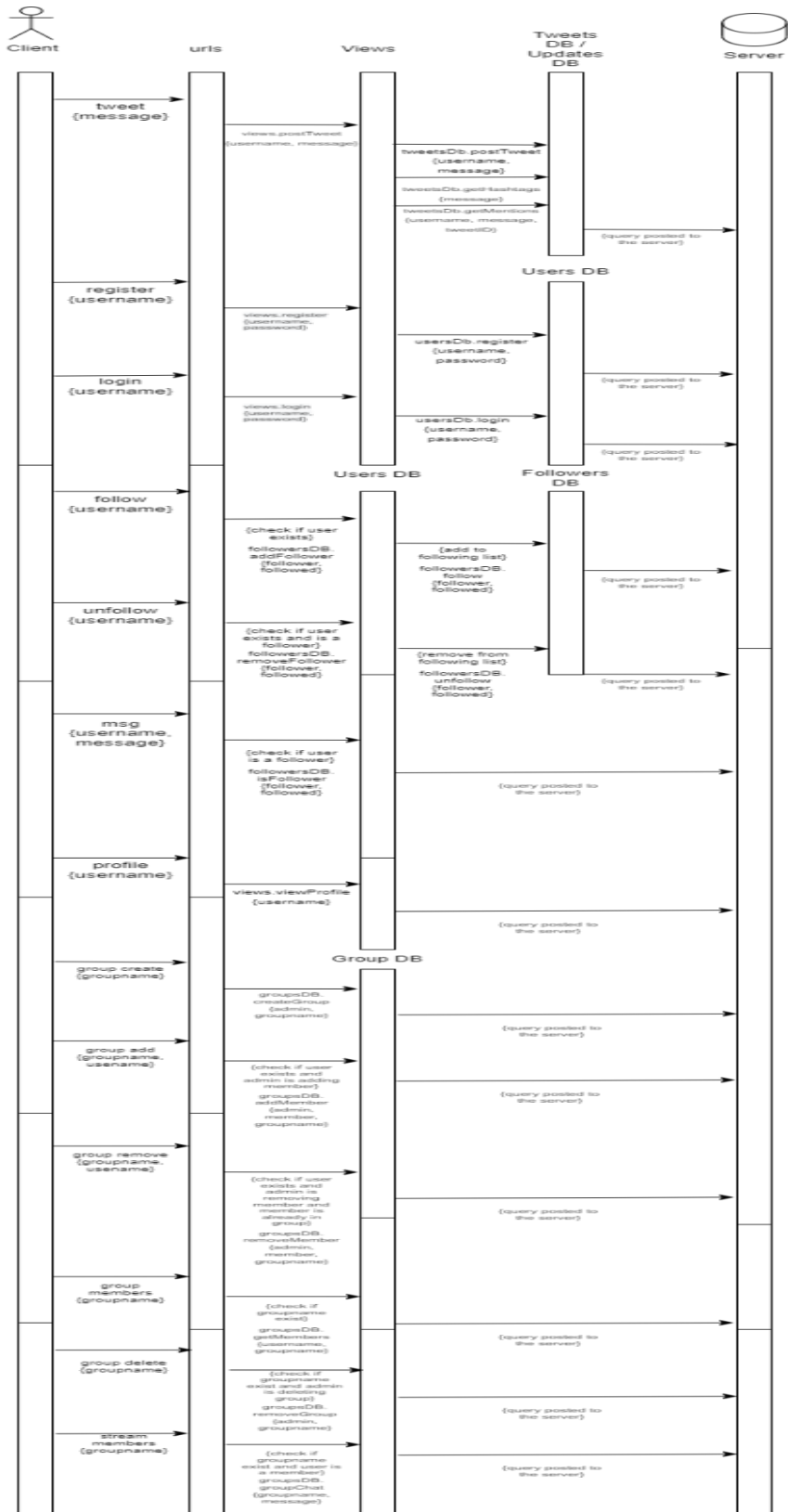


miro

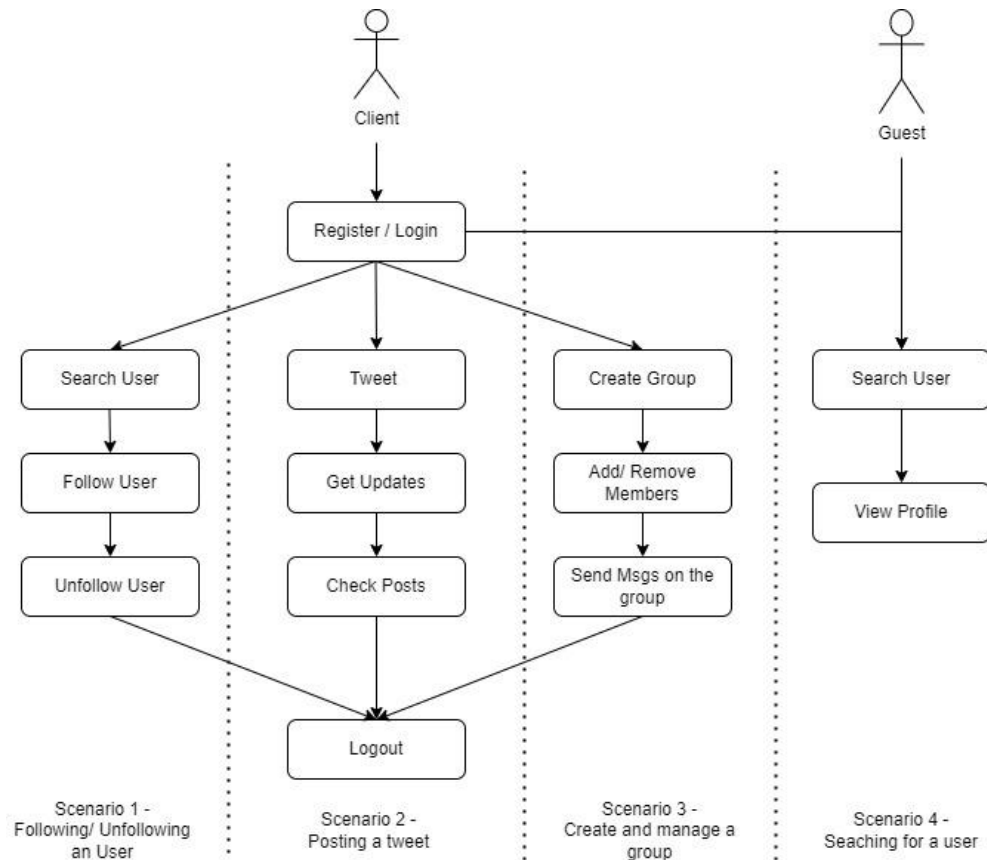


miro

Sequence Diagram



Activity and State Diagram



Software components and services implemented

Below is the technology stack we used in this project:

Python

We chose Python as the language to code in the entire project because of its simplicity and ease of implementation.

SQLite3 Relational Database

We chose SQLite3 because it is an embedded serverless relational database management system and the ease with which it can be integrated with Python using the Python SQLite3 module.

Shell Script

To automate the use of commands without having to type them manually every time.

Socket APIs

We're using Python's default socket library which is compatible with Linux and Unix-based systems. This socket library allows us to have a communication interface between the server and its clients.

Demonstration

The following modules are required to be set up before running the project.

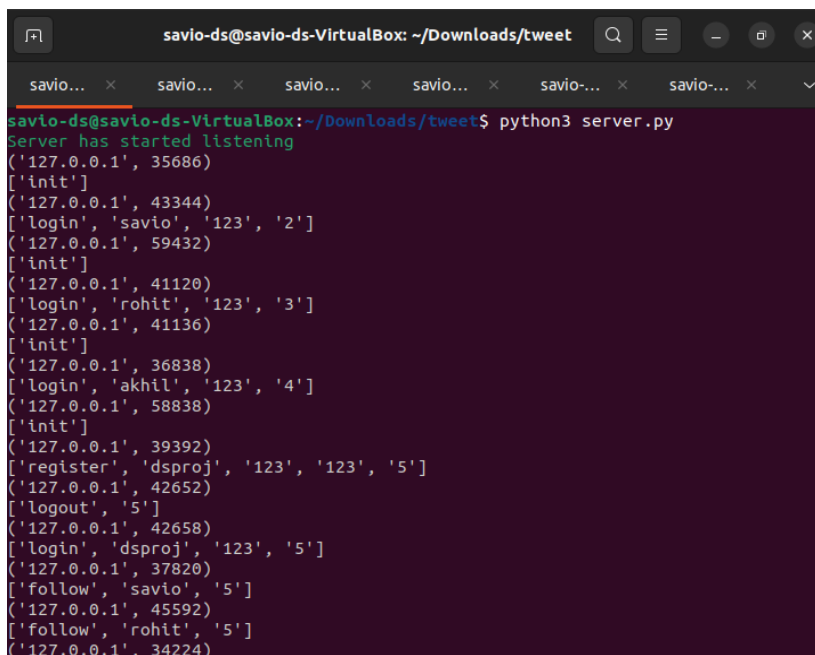
- pip3
- colorama

In the terminal, enter the following command:

```
sudo apt install -y python3-pip  
pip3 install colorama
```

* Instructions to run

- Open the terminal in the home directory of the project and then run server.py
Command: `python3 server.py`



```
savio-ds@savio-ds-VirtualBox: ~/Downloads/tweet  
savio-ds@savio-ds-VirtualBox:~/Downloads/tweet$ python3 server.py  
Server has started listening  
(127.0.0.1, 35686)  
[init]  
(127.0.0.1, 43344)  
[login, 'savio', '123', '2']  
(127.0.0.1, 59432)  
[init]  
(127.0.0.1, 41120)  
[login, 'rohit', '123', '3']  
(127.0.0.1, 41136)  
[init]  
(127.0.0.1, 36838)  
[login, 'akhil', '123', '4']  
(127.0.0.1, 58838)  
[init]  
(127.0.0.1, 39392)  
[register, 'dsproj', '123', '123', '5']  
(127.0.0.1, 42652)  
[logout, '5']  
(127.0.0.1, 42658)  
[login, 'dsproj', '123', '5']  
(127.0.0.1, 37820)  
[follow, 'savio', '5']  
(127.0.0.1, 45592)  
[follow, 'rohit', '5']  
(127.0.0.1, 34224)
```

- Open a new terminal in the home directory of the project and run the client.py
Command: python3 client.py

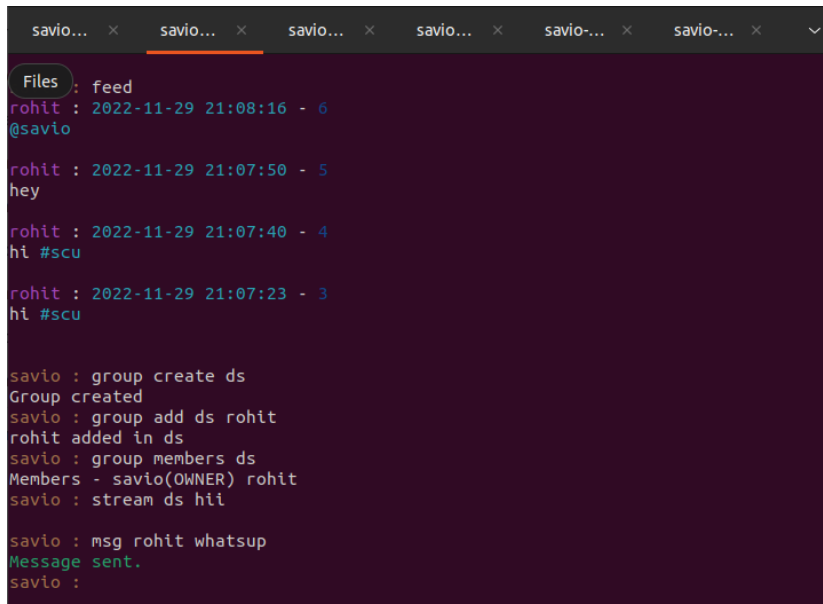
```
savio-ds@savio-ds-VirtualBox:~/Downloads/tweet$ python3 client.py
5
guest : register dsproj
Please enter your password:
Please re-enter your password:
Welcome to the Tweet App, dsproj
dsproj : logout
Log out successful
guest : login dsproj
Please enter your Password:
Logged in successfully
dsproj : follow savio
Successfully started following savio
dsproj : follow rohit
Successfully started following rohit
dsproj : unfollow rohit
dsproj : unfollow rohit
Successfully unfollowed rohit
dsproj : profile rohit
rohit - Followers : 1 Following : 1

dsproj : profile dsproj
dsproj - Followers : 0 Following : 1

dsproj :
```

```
savio-ds@savio-ds-VirtualBox:~/Downloads/tweet$ python3 client.py
2
guest : login savio
Please enter your Password:
Logged in successfully
savio : tweet hi
Successfully posted
savio : posts
savio : 2022-11-29 21:06:52 - 2
Trash
savio : 2022-11-29 20:33:14 - 1
yo

savio : online
*rohit
*dsproj
savio : feed
rohit : 2022-11-29 21:07:50 - 5
hey
rohit : 2022-11-29 21:07:40 - 4
hi #scu
rohit : 2022-11-29 21:07:23 - 3
hi #scu
```



```
Files : feed
rohit : 2022-11-29 21:08:16 - 6
@savio

rohit : 2022-11-29 21:07:50 - 5
hey

rohit : 2022-11-29 21:07:40 - 4
hi #scu

rohit : 2022-11-29 21:07:23 - 3
hi #scu

savio : group create ds
Group created
savio : group add ds rohit
rohit added in ds
savio : group members ds
Members - savio(OWNER) rohit
savio : stream ds hii

savio : msg rohit whatsapp
Message sent.
savio :
```

Below are some of the commands we implemented in the project:

login <username>	logs in the existing user
register <username>	registers a new user
logout	logs out the logged-in user
search <username>	user can search for other registered users
follow <username>	user can follow another user
unfollow <username>	user can unfollow an existing follower
profile <username>	user can see the follower and following
count	
tweet <your tweet text>	user can tweet messages. user can also
just enter tweet to type a long tweet	
msg <username> <your message text>	user can send a message to another user
provided the user is in the following list	
posts	user can check their own tweets and
retweets	
retweet	user can retweet a tweet from the tweetID.
group create <groupname>	user can create a group and will be the
owner of the group.	
group add <groupname> <username>	the user(owner) can add members to the
group.	

group remove <groupname> <username> the user(owner) can remove members to the group.

group members <groupname> the user can see who are the members of the group

delete <groupname> the user(owner) can delete the group

stream <groupname> <your message text> members of the group can send messages to all the other members in the group.

Analysis of Expected Performance & Results

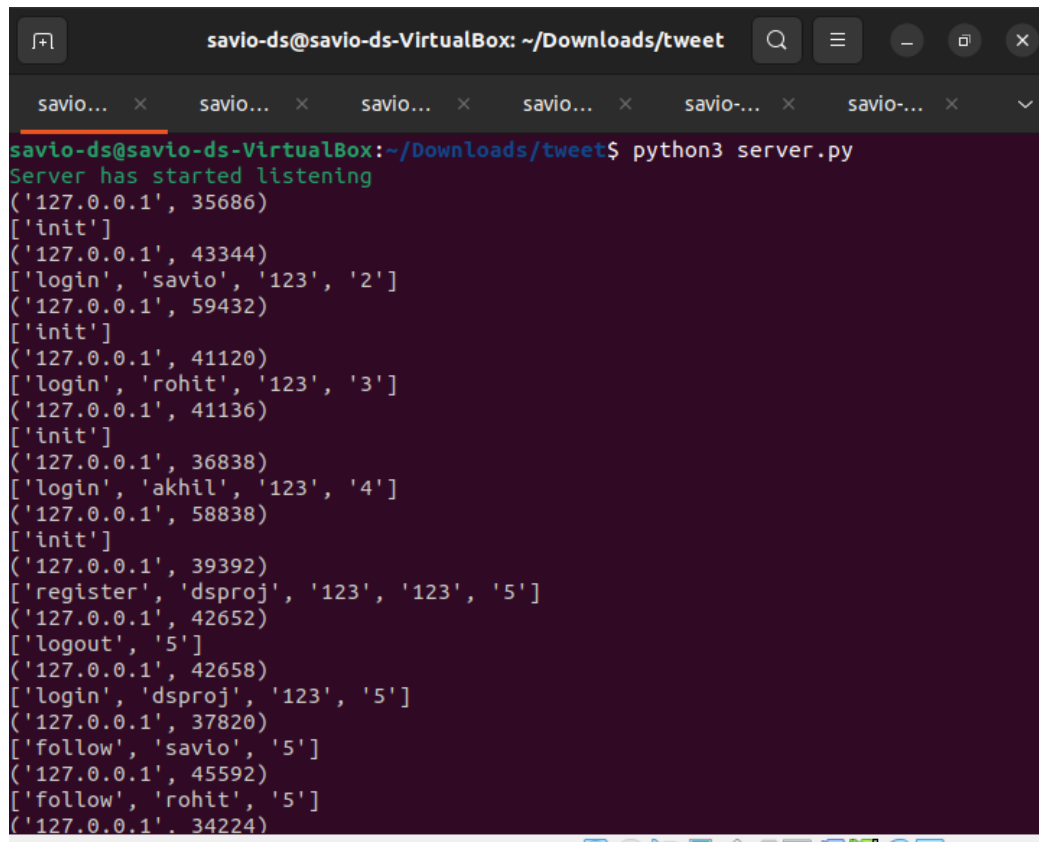
We tested it with one client before beginning implementation with one thread. Later, we developed multithreading to ensure concurrency by having each client reach the server in a separate way. Multithreading significantly increased performance.

We introduced 5 to 10 clients at once, and the performance remained constant. The application still needs to be load tested with a sizable number of users. When we transfer it to the cloud, this would occur. We anticipate that the performance of our application won't change for a bigger number of nodes. We also compared the performance to the time. There is no delay and every post and message is posted instantly.

Furthermore, the application was able to perform all the complicated functionalities with precision and no latency, like Group Chat, Personal Messages on the Group Chat, Tagging, etc. We tested the application on several grounds and it withstood the parameters.

Testing Results

Server logs



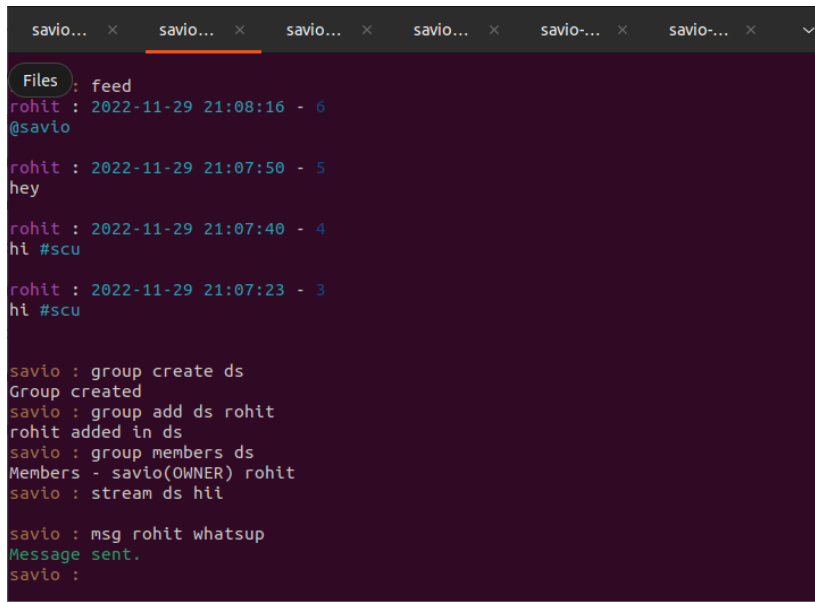
The image shows a terminal window titled 'savio-ds@savio-ds-VirtualBox: ~/Downloads/tweet'. The terminal displays the output of running 'python3 server.py'. The logs show the server starting and listening on 127.0.0.1. It then processes several requests: three 'init' requests, two 'login' requests (for 'savio' and 'rohit'), one 'register' request (for 'dsproj'), one 'logout' request, and two 'follow' requests (for 'savio' and 'rohit'). Each request is followed by a list of data returned by the server.

```
savio-ds@savio-ds-VirtualBox: ~/Downloads/tweet$ python3 server.py
Server has started listening
('127.0.0.1', 35686)
['init']
('127.0.0.1', 43344)
['login', 'savio', '123', '2']
('127.0.0.1', 59432)
['init']
('127.0.0.1', 41120)
['login', 'rohit', '123', '3']
('127.0.0.1', 41136)
['init']
('127.0.0.1', 36838)
['login', 'akhil', '123', '4']
('127.0.0.1', 58838)
['init']
('127.0.0.1', 39392)
['register', 'dsproj', '123', '123', '5']
('127.0.0.1', 42652)
['logout', '5']
('127.0.0.1', 42658)
['login', 'dsproj', '123', '5']
('127.0.0.1', 37820)
['follow', 'savio', '5']
('127.0.0.1', 45592)
['follow', 'rohit', '5']
('127.0.0.1', 34224)
```

Client Commands

```
savio-ds@savio-ds-VirtualBox:~/Downloads/tweet$ python3 client.py
2
guest : login savio
Please enter your Password:
logged in successfully
savio : tweet hi
Successfully posted
savio : posts
savio : 2022-11-29 21:06:52 - 2
Trash
savio : 2022-11-29 20:33:14 - 1
yo
savio : online
*rohit
*dsproj
savio : feed
rohit : 2022-11-29 21:07:50 - 5
hey
rohit : 2022-11-29 21:07:40 - 4
hi #scu
rohit : 2022-11-29 21:07:23 - 3
hi #scu
```

```
savio-ds@savio-ds-VirtualBox:~/Downloads/tweet$ python3 client.py
5
guest : register dsproj
Please enter your password:
Please re-enter your password:
Welcome to the Tweet App, dsproj
dsproj : logout
Log out successful
guest : login dsproj
Please enter your Password:
Logged in successfully
dsproj : follow savio
Successfully started following savio
dsproj : follow rohit
Successfully started following rohit
dsproj : unfollo rohit
dsproj : unfollow rohit
Successfully unfollowed rohit
dsproj : profile rohit
rohit - Followers : 1 Following : 1
dsproj : profile dsproj
dsproj - Followers : 0 Following : 1
dsproj :
```



```
savio... x savio... x savio... x savio... x savio... x savio... x
Files : feed
rohit : 2022-11-29 21:08:16 - 6
@savio
rohit : 2022-11-29 21:07:50 - 5
hey
rohit : 2022-11-29 21:07:40 - 4
hi #scu
rohit : 2022-11-29 21:07:23 - 3
hi #scu

savio : group create ds
Group created
savio : group add ds rohit
rohit added in ds
savio : group members ds
Members - savio(OWNER) rohit
savio : stream ds hii

savio : msg rohit whatsapp
Message sent.
savio :
```

Future Scope

Notification System

To check for updates to the profile, we must log in as the user and proceed to the terminal. Whether there are any unread messages. After integrating a UI, we will build up a notification system in our program.

Media sharing

In our Twitter application, only text-based tweets and messages are permitted to be posted. The capability of transferring both photographs and videos through it will be integrated. One of the essential aspects today's consumers need is media sharing.

User interface

The capability is currently operational with the terminals after being installed. Any future front-end technology will be combined with an appropriate user interface, as anticipated.

Mention other users in tweets

We also intend to add a feature that will let us mention other users in our own tweets, much like every other social media platform does at the moment

Cloud deployment

In the future, we want to deploy our application using the scalability and user-friendliness of cloud computing platforms like AWS/Azure to lessen the need on local programs.

Conclusion

In order to create a distributed social network application and better understand distributed systems, we applied a variety of distributed systems methods and methodologies. Concurrency control, broadcast/multicast, replication, and consistency protocols are the algorithms we employed. We took into account a variety of choices, underlying constraints, and challenges, as well as cutting-edge methods aimed at getting around some of the constraints while creating our system. Additionally, we investigated any other distributed systems methodologies that would be required as the program expands.

Using the aforementioned methods, we were able to successfully design and build a working prototype of a compact and effective distribution system.

Contribution Distribution

Task Done	Team Member Name
Proposal	Savio, Akhil, Rohit
Socket Programming	Savio, Rohit
Concurrency Control	Savio, Rohit
Replication and consistency protocol	Akhil, Rohit
Broadcast/Multicast	Savio, Akhil
Presentation	Savio, Akhil, Rohit
Report	Savio, Akhil, Rohit

References

1. Weipeng Li, Hai Huang and Lei Zhang, "A role-based distributed publish/subscribe system in IoT," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), 2015, pp. 128-133, doi: 10.1109/ICCSNT.2015.7490721.
2. Z. Chen, G. Cong, Z. Zhang, T. Z. J. Fuz and L. Chen, "Distributed Publish/Subscribe Query Processing on the Spatio-Textual Data Stream," 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 1095-1106, doi: 10.1109/ICDE.2017.154.
3. M. Jergler, K. Zhang and H. -A. Jacobsen, "Multi-Client Transactions in Distributed Publish/Subscribe Systems," 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 120-131, doi: 10.1109/ICDCS.2018.00022.
4. A. V. Terkhedkar and M. A. Shah, "Approaches to provide security in publish/subscribe systems — A review," 2016 International Conference on Inventive Computation Technologies (ICICT), 2016, pp. 1-4, doi: 10.1109/INVENTIVE.2016.7823221.
5. N. Apolónia, S. Antaris, S. Girdzijauskas, G. Pallis and M. Dikaiakos, "SELECT: A Distributed Publish/Subscribe Notification System for Online Social Networks," 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 970-979, doi: 10.1109/IPDPS.2018.00106.
6. W. Li, S. Hu, J. Li and H. -A. Jacobsen, "Community Clustering for Distributed Publish/Subscribe Systems," 2012 IEEE International Conference on Cluster Computing, 2012, pp. 81-89, doi: 10.1109/CLUSTER.2012.67.