Winter 2024 CSE530 Distributed Systems Assignment 1

This assignment has three parts:

- Using gRPC to implement an online shopping platform
- Using ZeroMQ to implement a group messaging application (think of groups in WhatsApp)
- Using RabbitMQ to implement a YouTube-like application

Important points to note for all the three above parts:

- While giving the demo of your implementation, you will be required to run your code on Google Cloud instances. Client/User should be running on your laptop/desktop while servers should be running on Google Cloud instances.
  - It is fine to have client/users also running on Google Cloud instances.
  - You will be given only limited credits for Google Cloud. So, do your implementation on your local machine. Only use Google Cloud instances for testing purposes.
  - Please ensure to shut down your Google cloud instances when not in use.
  - **Please refer to the email sent on January 30th (around 4:25 pm) for steps to avail the Google Cloud Credits.**
  - Google Cloud Tutorial
    - Creating a Virtual Machine & Updating Firewall Rules
      - VM Creation (Linux Preferred)
      - VPC Firewall Rules (For gRPC to communicate between VM Instances)
    - **We have also recorded a tutorial for Google Cloud available here.**
- Please stick to the specifications as much as possible. In case of any confusion, please clarify with the instructor / Teaching Fellow / TAs (preferably over Slack). The instructor and TAs will also hold several office hours before the assignment deadline.
- We have tried to mention as much detail as possible. Still, any reasonable assumptions are fine. It will be best if you confirm your assumptions with us before going ahead with them.
- Please take care of basic/reasonable error handling in your code.
- Start early on the assignment. Assignment is straightforward, but it will still require you to spend some time and effort.
- Use of LLM-based tools such as ChatGPT, etc., is ALLOWED. But if you are not able to answer questions in the demo/viva or if you are not able to explain your code, then you will be penalized. You cannot justify - “ChatGPT produced the code, I don’t know the exact purpose of this line of code!”
  - Also, we will be asking you to share your relevant conversations with ChatGPT and other LLM-based tools. This will not be used for judging you in any way. But it will be used for research purposes so that we can

build better LLM-based tools for students (you!!). **Do note that we will have a small non-zero weightage in this assignment's evaluation for sharing these details.**

Please fill this google form for sharing all your interactions with LLM-based tools.

Deliverables: Submit one zipped file having three folders (one each for gRPC, ZeroMQ and RabbitMQ implementations). Only one student needs to submit the zipped file on Google Classroom in each group. The name of the zip should be groupid_a1, for example, for group 1: 1_a1.zip.

Part 1: Using gRPC to implement an Online Shopping Platform

0. Learning Resources

For this part of this assignment, you would need to be familiar with

- Google Cloud
  - Creating a Virtual Machine & Updating Firewall Rules
  - VM Creation (Linux Preferred)
  - VPC Firewall Rules (For gRPC to communicate between VM Instances)
- Protocol Buffers
  - These are similar to JSON / XML, with the added benefit of being able to use them as objects directly in any language (even multiple languages simultaneously)
  - Tutorial (proto 3 preferred)
- gRPC
  - *Google's Open source Remote Procedure Call Framework*
  - Tutorial

The above links should give you all the necessary information to complete this assignment. Implementing the examples on the gRPC tutorial page is highly recommended.
We suggest you use Python or C++. The C++ tutorial page has commands only for Linux and macOS, while Python can be used for any OS.

PYTHON CODE FOR UUID :
```
import uuid
unique_id = str(uuid.uuid1())
```

I. Details:

The shopping platform will have the following components/nodes:

Market (Central Platform): The Market serves as the central platform that connects sellers and buyers. *Buyers and sellers don't communicate with each other directly*.

Seller: The Seller is a client interacting with the Central Platform to manage their items and transactions.

Buyer: The Buyer is another client interacting with the Central Platform to search for and purchase items.

All these nodes will reside on different virtual machine instances (at Google Cloud) and can communicate with each other through their external IP addresses.

Market (Central Platform):

The Market is similar to Amazon's central platform for buying and selling products. All clients (sellers and buyers) interact with the central platform. It maintains all the seller accounts, the items they sell, quantity, transaction logs, reviews, and so on. The central platform is also responsible for sending out notifications to buyers in cases where the seller updates an item, and sellers receive notifications when someone buys their products. The Market node is deployed at a certain address (ip:port), which is known to all the sellers and buyers.

Seller (Client):

Each seller resides at a different address (addresses are in the form of ip:port). **This address specifies where the Notification server for the current seller is hosted.** The seller interacts with the Market to register itself, add products for sale, and update/delete/view their products. The sellers also receive notifications displaying information about items recently purchased by the buyers. The sellers also generate and maintain a UUID, which is sent to market each time the seller interacts with the market for secure transactions.

Buyer (Client):

Each buyer resides at a different address (addresses are in the form of ip:port). **This address specifies where the Notification server for the current buyer is hosted.** The buyer interacts with the market to search for products and buy them. Buyers can also wishlist a product to receive notifications from the Market whenever the seller updates their wish-listed product (this scenario simulates the pub-sub pattern).

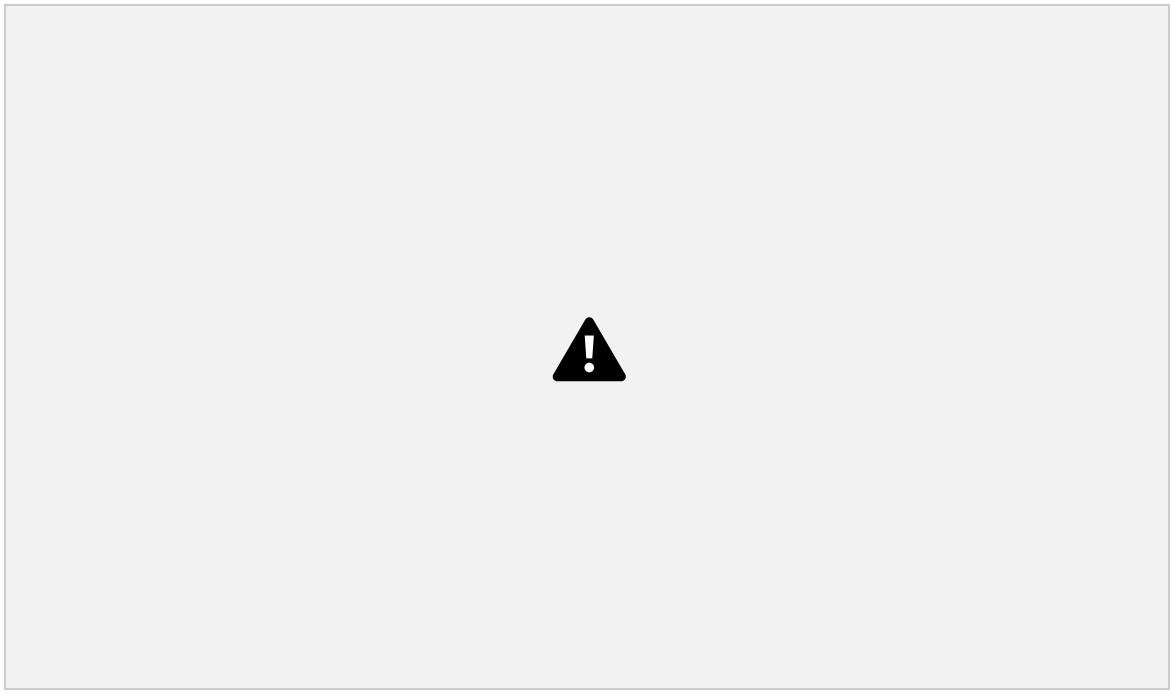


Fig: Overview of the Shopping Platform

II. RPC Implementation:

NOTE: All communication happens in the form of protos (even if for single strings), As shown on

the HelloWorld tutorial page. *Tip: It is better to have separate proto-definitions for each type of communication.*
For each communication, both sides must print what they received.
You may choose to print the proto directly. __str__(): returns a human-readable representation of the message, particularly useful for debugging. (Usually invoked as str(message) or print message.) src: https://developers.google.com/protocol-buffers/docs/pythontutorial

Seller ↦ Market

RegisterSeller: This functionality allows a seller to register. The seller registers with the market by sending its address (ip:port) **where the notification server is hosted** along with (unique) uuid. The market accepts the client as a seller and sends a SUCCESS response if the address isn't already registered else it sends a FAILED response.

Market prints: Seller join request from 192.13.188.178:50051[ip:port], uuid = 987a515c-a6e5-11ed-906b-76aef1e817c5
Seller prints: SUCCESS [or FAIL]

SellItem: This functionality allows a seller to post a new item/product on the market. The seller sends a request containing the details of the item, including information such as Product Name, Category (one of ELECTRONICS, FASHION, or OTHERS), Quantity, Description, Seller Address, and Price per unit along with the seller UUID to verify seller credentials. Upon receiving the details, the market assigns a **unique item ID** to the product and also takes care of storing the **buyer ratings**. The market processes the request, and if the item is successfully added, it responds with a SUCCESS message. If there are any issues, the market responds with a FAILED message.

Market prints: Sell Item request from 192.13.188.178:50051
Seller prints: SUCCESS [or FAIL or return unique item id]

UpdateItem: This functionality allows the seller to update their items posted in the market. The seller sends a request containing the Item ID of the item to be updated, along with the new Price, new Quantity, and the seller's address and UUID for credential verification. The market validates the credentials, checks if the specified item exists, and updates its details accordingly. If the update is successful, the market responds with a SUCCESS message; otherwise, it sends a FAILED message.

Note: This functionality also triggers notifications to all buyers who have wish-listed the product being updated.

Market prints: Update Item 3[id] request from 192.13.188.178:50051
Seller prints: SUCCESS [or FAIL]

DeleteItem: This functionality allows the seller to delete one of their items. The seller sends a request containing the Item ID to be deleted, along with the seller's address and UUID for credential verification. *This functionality need not trigger notifications.*

Market prints: Delete Item 3[id] request from 192.13.188.178:50051
Seller prints: SUCCESS [or FAIL]

DisplaySellerItems: This functionality allows the seller to see all their uploaded items. The seller sends a request containing their address and UUID (optional). The market needs to return the list of all the uploaded items along with all the details associated with the item (such as ID, price, quantity, rating, etc).

Market prints: Display Items request from 192.13.188.178:50051

Seller prints:

–

Item ID: 1, Price: $500, Name: iPhone, Category: Electronics,

Description: This is iPhone 15.

Quantity Remaining: 5

Seller: 192.13.188.178.50051

Rating: 4.3 / 5

–

Item ID: 2, …(and so on)

–

Buyer ↦ Market

SearchItem: Using this functionality, buyers can search for items for sale. The buyer sends a request containing the item name (can leave blank to display all items) and the item category (any one of ELECTRONICS, FASHION, OTHERS, ANY; where ANY will display items with all categories).

Market prints: Search request for Item name: <empty>, Category: ANY.

Buyer prints:

–

Item ID: 1, Price: $500, Name: iPhone, Category: Electronics,

Description: This is iPhone 15.

Quantity Remaining: 5

Rating: 4.3 / 5 | Seller: 192.13.188.178.50051

–

Item ID: 2, …(and so on)

–

BuyItem: Buyers can place orders to buy items using this functionality. The buyer must send a request containing the item ID, the quantity to purchase, and the buyer's address (ip:port) **where the notification server is hosted**. The market will return SUCCESS on successful transactions and FAILED in case of issues such as invalid item ID or not enough stock available. The market needs to automatically update item quantity on each successful transaction.

Note: This functionality also triggers notifications to the seller of the item.

Market prints: Buy request 1[quantity] of item 3[item id], from 120.13.188.178:50051[buyer address]

Buyer prints: SUCCESS [or FAIL]

AddToWishList: Buyers can subscribe to some items using this function to receive notifications. The request must contain the item ID and the buyer's address **where the notification server is hosted.**

Market prints: Wishlist request of item 3[item id], from 120.13.188.178:50051

Buyer prints: SUCCESS [or FAIL]

RateItem: Buyers can rate any item (irrespective of whether they've bought the item) using this functionality. The request must contain the Item ID and buyer's address, along with an integral rating ranging from 1 to 5. A buyer can rate a product only once.

Market prints: 120.13.188.178:50051 rated item 3[item id] with 4 stars.

Buyer prints: SUCCESS [or FAIL]

Market ↦ Buyer/Seller

NotifyClient: Some of the above functionalities trigger notifications to either the buyers or the sellers. The notification message must contain the updated item details in both cases. This functionality must be implemented by both the buyer and seller so that the central platform is able to send notifications whenever an update occurs.

Buyer/Seller prints:
########
The Following Item has been updated:
Item ID: 1, Price: $500, Name: iPhone, Category: Electronics,
Description: This is iPhone 15.
Quantity Remaining: 5
Rating: 4.3 / 5 | Seller: 192.13.188.178.50051
########

III. Evaluation:

Each project submission will be evaluated by running the files **(need not be Python files; can be language-specific)** from the deliverables submitted.

The TAs evaluating will run the files in the following order:

market.py / central_platform.py
seller.py for 1st seller
seller.py for 2nd seller
client.py for 1st client
client.py for 2nd client
README file - please add whatever you feel is necessary for TAs to understand your code.

Part 2: Using ZeroMQ to Low-Level Group Messaging Application

Links for ZeroMQ

For this part of the assignment, you would need to be familiar with ZeroMQ

Relevant Resources:

DSCD v4 textbook
https://zeromq.org/socket-api/#publish-subscribe-pattern
https://zeromq.org/socket-api/#request-reply-pattern

https://rfc.zeromq.org/spec/29/
https://rfc.zeromq.org/spec/28/
https://zguide.zeromq.org/
https://zeromq.org/get-started/ - Tutorial

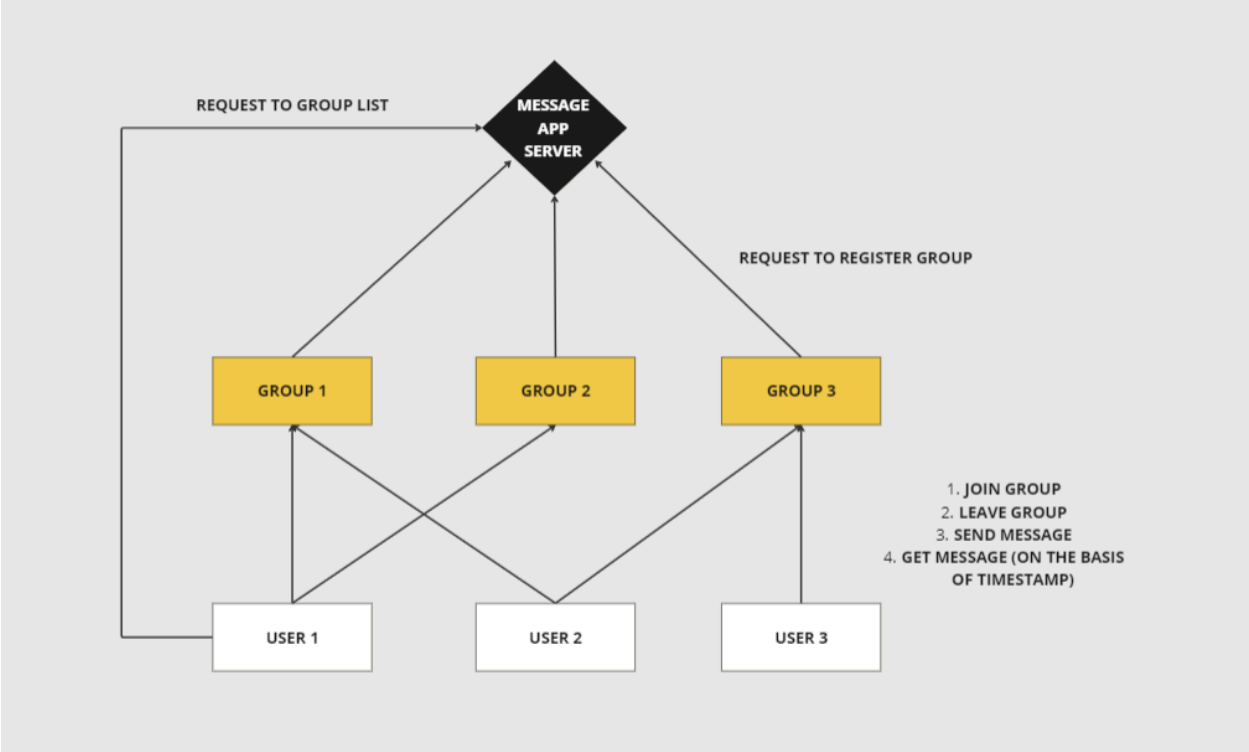
Suggested Programming languages: python / Go / Java / C++

1. Introduction

This assignment aims to build a low-level group messaging application using ZeroMQ. The application will consist of a central messaging app server, groups, and users interacting with each other in real-time. The main message app server maintains the group list. The groups manage the user and store and handle the message sent by the user. The user can join the group, send the message in the group, get all the messages sent in the group (including and after the given timestamp), and leave the grp.

2. Architecture Overview

The architecture comprises a central server, multiple groups, and users. The server maintains the list of groups along with their IP addresses, while users can join or leave groups and exchange messages within those groups.



3. Types of Nodes

Messaging App Server

Group List Maintenance

The server will maintain a list of groups identified by a unique identifier. It will store the IP addresses associated with each group to facilitate communication.

User Interaction

Users can request the list of available groups from the server. ~~The server will handle user requests to join or leave groups, updating the group membership accordingly.~~

Groups

User Management

Groups will maintain a list of users who are currently part of the group. Users can join or leave the group, and this information will be updated on the GROUP server.

Message Handling

Groups will store messages sent by users. When a user requests messages from a group, the GROUP server will fetch the relevant messages and send them to the user.

Message Storage

The GROUP server will store messages sent by users within each group. This includes the timestamp and the message content. Messages will be persisted (only in memory) to ensure they are available even if users join or leave the group.

Users

Group Interaction

Users can join multiple groups simultaneously. They can also leave a group when desired. The GROUP server will manage user group memberships.

Message Operations

Users can write messages within the groups they are part of. They can also fetch all messages from a specific group. To fetch the message, the user needs to mention the date, and it will return all the messages after that date. If no date is mentioned, the user will fetch all the messages. Messages will be sent to the GROUP server for storage and retrieval.

4. Detailed Explanation

Message_server ↔ Group

1) **MessageServer** - A group server will request the message_app Server to register itself. The message_app server registers the group server and sends a SUCCESS response. The group server sends its Name and IP Address to the message_server to Register itself.

message_server prints: JOIN REQUEST FROM LOCALHOST:2000 [IP: PORT]
Server prints: SUCCESS

User ↔ Message_server

1) **GetGroupList** - A user will request the message_server to furnish the list of live groups. The message_server returns a list of live servers in the following format [Server_number - IP_Address]

message_server prints: GROUP LIST REQUEST FROM LOCALHOST:2000 [or UUID]
The user prints: ServerName1 - localhost:1234
ServerName2 - localhost:1235

User ↔ group

1) **joinGroup**- A user requests a group to connect to it by sending its uuid. The group accepts the user, adds it to its USERTELE, and sends a SUCCESS response.

Group prints: JOIN REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5 [UUID OF USER]
User prints: SUCCESS

2) **LeaveGroup**- A user requests the GROUP server to remove itself from the group's
USERTELE.

Group prints: LEAVE REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5 [UUID OF USER]
Client prints: SUCCESS

3) **GetMessage**- A client requests a group for all the messages on this group, having the message after the specified date. If no date is specified, then print all the messages. The group then sends all the corresponding messages if the user is a part of its USERTELE. To connect with a user, the group deploys a new thread. In this manner, the server can serve multiple clients simultaneously.
For example -

a user can request a server for all messages after HH:MM: SS (including) timestamp.
a user can request a server for all messages if no date is specified.

Group prints: MESSAGE REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5 [UUID OF USER]
USER prints:
If the timestamp is specified, all the messages, including and after the given timestamp.
Else all the messages till now.

4) **SendMessage** - A user can send a message to the group. The group accepts a message if the user is part of its USERTELE and sends a SUCCESS response; otherwise, it sends a FAILED response. Once a message is received, the group updates it with the time it was received. To connect with a user, the group deploys a new thread. In this manner, the group can serve multiple users simultaneously.

Group prints: MESSAGE SEND FROM 987a515c-a6e5-11ed-906b-76aef1e817c5
User prints: SUCCESS [FAIL]

5. Evaluation

Each project submission will be evaluated by running the files from the deliverables submitted **(need not be Python files; can be language-specific)**. The TAs evaluating will run the files in the following order:

message_server.py
group.py
user_1.py for 1st user [A single user.py is also fine as long as you run each user on different instances/ports]
user_2.py for 2nd user

Part3: Building a YouTube-like application with RabbitMQ

1. Message Queues and RabbitMQ Introduction

Message Queues are fundamental to modern distributed systems, providing asynchronous communication between different components. You can learn more about message queues here: https://youtu.be/xErwDaOc-Gs?si=6rQoyJEIH0exKKuB

RabbitMQ is a widely used message broker that facilitates communication between different system parts. RabbitMQ follows the Advanced Message Queuing Protocol (AMQP), an intermediary for sending and receiving messages between distributed system components.

Going through the following official tutorial of RabbitMQ will be very helpful:
https://www.rabbitmq.com/tutorials/tutorial-one-python.html

You can learn more about RabbitMQ here:
https://youtu.be/7rkeORD4jSw?si=95NiBpRs2nsfZVTp

Knowledge of message queues and RabbitMQ is essential before starting the assignment, as you'll have to decide which type of queues and exchanges to use while communicating between different application services.

2. Introduction to YouTube Service

In this programming assignment, you will build a simplified version of a YouTube application using RabbitMQ. The system will consist of three components: (detailed deliverables are provided in the next section)

2a. YouTuber

Youtuber is a service that allows YouTubers to publish videos on the YouTube server.

Youtubers can publish videos by sending simple messages to the YouTube server, including the video name and the YouTuber's name.

2b. User

Users can subscribe or unsubscribe to YouTubers by sending messages to the YouTube server.

Users will receive real-time notifications whenever a YouTuber they subscribe to uploads a new video.

Note that YouTubers and users cannot directly communicate with each other; only through the YouTube server can they do so.

Users can log in by running the User.py script with their name as the first argument. Optionally, they can subscribe or unsubscribe to a YouTuber using the second and third arguments. (See deliverables).

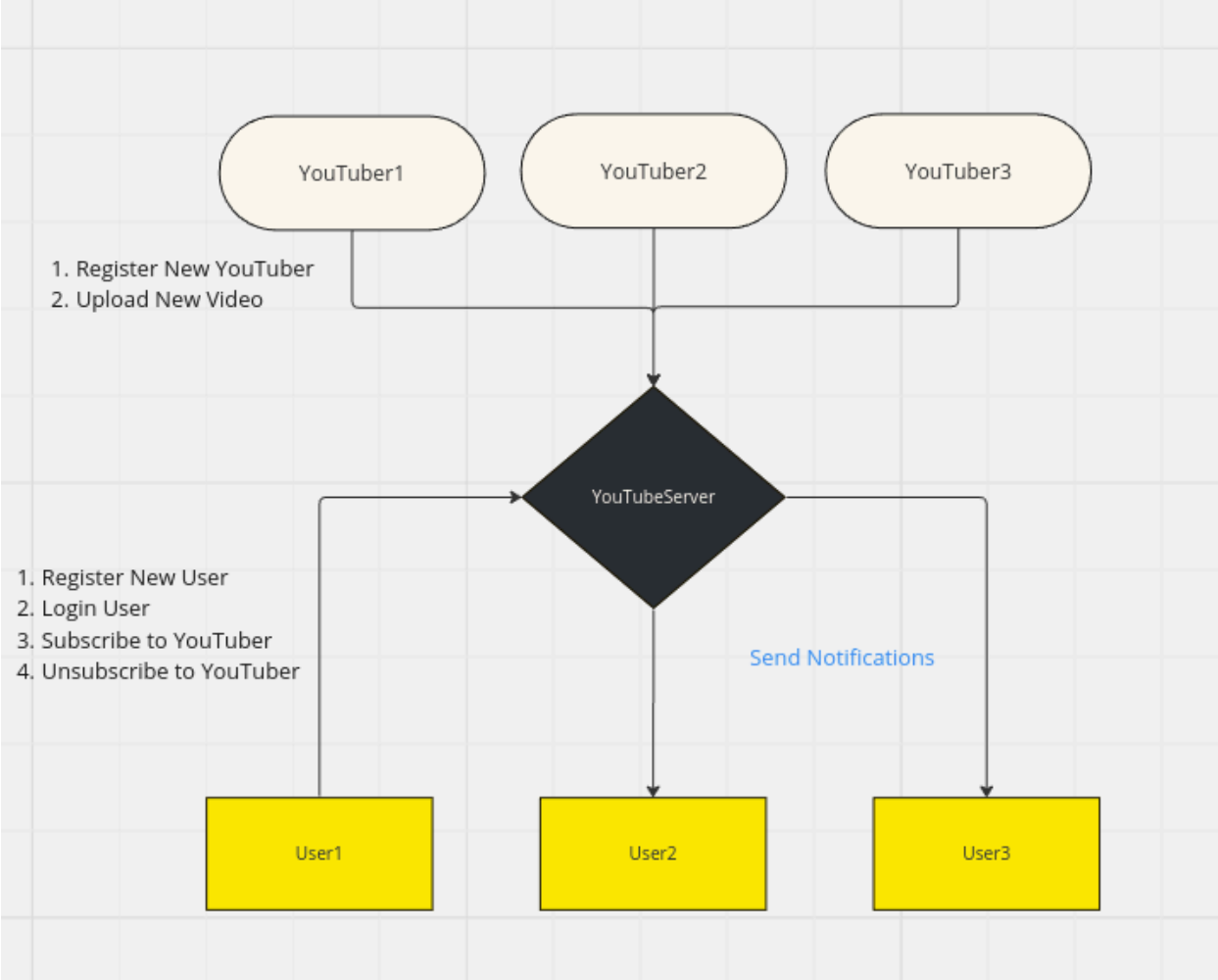
Whenever a user logs in, it immediately receives all the notifications of videos uploaded by its subscriptions while the user was logged out.

2c. YouTubeServer

The YouTube server consumes messages from Users and YouTubers and stores them as data, maintaining a list of YouTubers and their videos, and all users and their subscriptions.

It also processes subscription and unsubscription requests from users.

The server sends notifications to all subscribers of a YouTuber whenever they upload a new video.



4. Deliverables

You are required to submit three files **(need not be Python files, can be language-specific)**:

3a. YoutubeServer.py

This file sets up and runs the YouTube server. It is now ready to consume the following two messages:

Login and Subscription/Unsubscription Requests from Users.

New video upload from users.

Note that the server should be able to simultaneously consume messages from YouTubers and Users.

It does not take any command line arguments.

Methods:

-consume_user_requests()

Starts consuming Login and Subscription/Unsubscription Requests from Users.

Prints *"<username> logged in"* whenever a user logs in

Prints *"<username> <subscibed/ubsubscribed> to <youtuberName>"* whenever a user updates their subscriptions.

-consume_youtuber_requests()

Starts consuming video upload requests from YouTubers.

Prints *"<YouTuberName> uploaded <videoName>"* whenever a YouTuber uploads a new video.

-notify_users()

Sends notifications to all users whenever a YouTuber they subscribe to uploads a new video.

Example:

#Run the server

$ python youtubeServer.py

3b. Youtuber.py

This file represents the Youtuber service.

It takes two command line arguments: the YouTuber's name and the video it wants to publish.

Methods:

-publishVideo(youtuber, videoName)

Sends the video to the youtubeServer

The YouTube server adds a new YouTuber if the name appears for the first time else; it adds the video to the existing YouTuber.

Prints "*SUCCESS"* message when the video is received by the youtubeServer.

Example:

 # Run the Youtuber service to publish a video
$ python Youtuber.py TomScott After ten years, it's time to stop weekly videos.
(The Youtuber name should not contain spaces, but the name of the video can contain spaces)

3c. User.py

This file represents the User service.

It contains either 1 or 3 command line arguments. The first argument is the name of the user. If the user wants to subscribe or unsubscribe, the second argument is ***s*** or ***u,*** respectively, and the third argument is the name of the YouTuber. (Second and third arguments are optional)

The YouTube server adds a new user if the name appears for the first time; else logs in the user and subscribes/unsubscribes existing users accordingly

Methods

-updateSubscription

Sends the subscription/unsubscription request to the YouTubeServer with the following parameters in the body:

```
{
    "user": "username",
    "youtuber": "youTuberName",
    "subscribe": "True"
}
```

Prints "*SUCCESS"* message after completing the request.

-receiveNotifications

Receives any notifications already in the queue for the users subscriptions and starts receiving real-time notifications for videos uploaded while the user is logged in.

Prints *"New Notification: <YouTuberName> uploaded <videoName>"*

Examples:

 # Run the User service to log in, subscribe to a YouTuber, and receive notifications
$ python User.py username s TomScott
 # Run the User service to log in, unsubscribe to a YouTuber, and receive notifications
$ python User.py username u TomScott
 # Run the User service to log in and receive notifications
$ python User.py username

3d. README.md

A README file containing all necessary steps to run the program.

5. Flow of Service

Run YoutubeServer.py to start the server.

Run Youtuber.py and User.py in any sequence multiple times and simultaneously. **Multiple users can be logged in simultaneously and should receive notifications simultaneously**

Users receive real-time notifications when their subscribed YouTubers upload a new video.

Notes:

It is up to you to decide the appropriate queueing architecture, but data should only be transferred using RabbitMQ message queues between the three components.

The exact method parameters are up to you to decide, but the method functionality should remain the same. Feel free to use additional methods.

There can be multiple correct ways to satisfy the requirements of the three components.