Assignment 1 - Publish-Subscribe system using gRPC, RabbitMQ, and ZeroMQ

Overview

- In this assignment, you will implement a publish-subscribe system (PubSub) (or a simplified version of Discord).
- Your PubSub will allow you to publish simple articles that can be subscribed to.
- With this assignment, you will get hands-on experience with RPCs.
- You must do the system with gRPC, RabbitMQ, and ZeroMQ.
    - ***Should make all 3 separate implementations***
- The resources to assist you are in the ***Getting Familiar*** Section.
- The Dos and Don'ts are given, starting **from the *Details*** section.
- ***We suggest that you start early. No extensions will be provided whatsoever.***

Deliverables

Submit all the code files, for each of the three implementations (gRPC, zeroMQ, RabbitMQ), including the ones generated by the Protocol Buffer compiler, in a single zip file. Ensure your code files are well segregated into respective folders and properly named corresponding to the registry_server, server, client_1, and client_2. The zip file containing the 3 folders for 3 implementations should be named *groupid_a1*, for example, for group 1: ***1_a1.zip***

Getting Familiar

gRPC - PART 1

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

- 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
- gRPC
    - *Google's Open source Remote Procedure Call Framework*
    - Tutorial

The above two links should give you all the necessary information to complete this assignment. It is highly recommended to implement the examples on the gRPC tutorial page.
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())

RabbitMQ - Part 2

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

- RabbitMQ:
    - *Description:* https://www.rabbitmq.com/tutorials/amqp-concepts.html
    - *AMQP:* https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol
    - *Installation:* https://www.rabbitmq.com/download.html
    - ~~*Toy example:* https://www.rabbitmq.com/tutorials/tutorial-three-python.html~~
    - *Toy example:* https://www.rabbitmq.com/tutorials/tutorial-six-python.html

The above links should give you all the necessary information to complete this assignment. It is **highly recommended** to play around with the toy example. *We suggest you use **python** or **Java.***

ZeroMQ - Part 3

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++.***

Details

PubSub will have three types of nodes:

- Registry Server - can be thought of as the central server of Discord, which maintains the information of all the servers (Name - Address). *Note: Address is of the form ip:port, for example - localhost:8000. Assume all the servers and clients know the address of the registry server.*
- Servers - individual servers for each community
- Clients - individual users

All these nodes reside on your own system. *Tip: Open separate terminals for each node and deploy them on a different port.*

Registry Server

The registry server can be considered the central server of Discord, which maintains the information of all the servers (Name - Address). The name and address of each server will be unique.

Servers

Each server will be deployed at a particular address. On deployment, it needs to register itself with the registry server. *Each server represents a different community.* Each server hosts articles or articles published by the clients connected to this server and sends articles to the clients who subscribe to this server. Each server also maintains the information about the clients that have connected to it in its CLIENTELE - *list of clients*.

Clients

Each client can connect to the registry server first and requests the list of all active/alive servers. Then it can connect directly to the server it is interested in. Clients can connect to multiple servers, i.e., a client can be a part of numerous servers' CLIENTELES. Clients can request a server for a subset of articles. Clients can also publish articles to a server.

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

Server ↔ Registry Server

1) `Register` - A server will request the Registry Server to register itself. The Registry Server registers the Server if the current number of servers is less than MAXSERVERS and sends a `SUCCESS` response, else it sends a `FAILED` response.

   The server sends its Name and Address to the registry server to Register itself.

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

Client ↔ Registry Server

1) `GetServerList` - A client will request the Registry Server to furnish the list of live servers. The registry server returns a list of live servers in the following format -
   `Server1 - Address1`
   `Server2 - Address2`
   Registry Server prints: SERVER LIST REQUEST FROM LOCALHOST:2000
   Client prints: ServerName1 - localhost:1234
   ServerName2 - localhost:1235

Client ↔ Server

1) `JoinServer` - A client requests a server to connect to it by sending its uuid. The server accepts the client and adds it to its `CLIENTELE` if the existing number of clients is less than `MAXCLIENTS` and sends a `SUCCESS` response else it sends a `FAILED` response.

   Server prints: JOIN REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5 [UUID OF CLIENT]
   Client prints: SUCCESS [FAIL]
2) `LeaveServer` - A client requests the server to remove itself from the server's `CLIENTELE`.
   Server prints: LEAVE REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5 [UUID OF CLIENT]
   Client prints: SUCCESS [FAIL]
3) `GetArticles` - A client requests a server for all the articles on this server having the tags specified. The server then sends all the corresponding articles if the client is a part of its `CLIENTELE`. ~~To connect with a client, the server deploys a new thread. In this manner, the server can serve multiple clients simultaneously.~~
   For example -
   a) a client can request a server for all articles of type sports by the author Jack published after 1st January 2023.
   b) a client can request a server for all articles the author Jack published after 1st January 2023.
   c) a client can request a server for all articles of type sports published after 1st January 2023.

Server prints:	ARTICLES REQUEST FROM 987a515c-a6e5-11ed-906b-76aef1e817c5
[UUID OF CLIENT]	FOR SPORTS, <BLANK>, 2/02/2020
Client prints :	1)SPORTS Jack Dorsey *04/02/2023* Messi has won the FIFA World Cup 2)SPORTS Harsha Bhogle *07/02/2023* Virat is doing great OR [FAIL]

4) `PublishArticle` - A Client can publish an article to the server. The server accepts an article if the client is a part of its `CLIENTELE` and sends a `SUCCESS` response else it sends a `FAILED` response. Once an article is received, the server updates it with the time it was received. ~~To connect with a client, the server deploys a new thread. In this manner, the server can serve multiple clients simultaneously.~~

Server prints: ARTICLES PUBLISH FROM 987a515c-a6e5-11ed-906b-76aef1e817c5
Client prints: SUCCESS [FAIL]

Article Format

You must define your own proto for an article. The article includes the following.

1) Type - one of [SPORTS, FASHION, POLITICS] *use oneof datatype (for gRPC) to check fields automatically.*
2) Author - a string of characters for the author's name
3) Time - Time when the message was received at the server
4) Content - a string of a maximum of 200 characters.

Article Request Format

You must define your own proto (Compulsory to use protos for gRPC, do not use JSON / XML, etc.) for a request. A response includes the following.

1) Type - one of [SPORTS, FASHION, POLITICS] *use oneof datatype (for gRPC) to check fields automatically.*
2) Author - a string of characters for the author's name
3) Time - Time when the message was received at the server

Examples of Legal Formats

1) Only for `PublishArticle`

SPORTS
Jack Dorsey
<blank - to be filled by the server>
Messi has won the FIFA World Cup

Cannot have anything blank! The time stamp will get get populated by the server.

2) Only for `GetArticles`

SPORTS	*<blank>*	SPORTS
Jack Dorsey	Jack Dorsey	*<blank>*
01/01/2023	01/01/2023	01/01/2023

Examples of Illegal Formats

1) Only for `PublishArticle`

<blank>
Jack Dorsey
10/01/2023
Messi has not won the FIFA World Cup

<blank>
<blank>
<blank>

All blanks are not allowed

Bonus

To get bonus marks, ensure that a server can become a client of another server and automatically gets all the articles published on the server without requesting articles using the `GetArticles` call.
Example 1 - Server A joins Server B. All articles published on server B would be sent to server A automatically but not vice-versa. In effect, a client of server A can now access articles of server B too. Ensure that there are no race conditions.
Example 2 - Server A joins Server B, and Server B joins Server C, and a user makes a `GetArticles` call to server A, then they should receive the corresponding filtered messages from Server B and Server C.

Evaluation

Each project submission will be evaluated by running the files from the deliverables submitted. The TAs evaluating will run the files in the following order:

- registry_server.py
- server.py
- client.py for 1st client
- client.py for 2nd client