Milestone 2: Interprocess Communication, Remote Communication, and Indirect Communication

GITHUB LINK:
https://github.com/CamilleWong-elif/CECS-327-Music-Playlist-Project.git

1. **Architecture Overview**:

There are three classes: Client, Server, and Notification. Client has three functions: song_request(), receive_notifications(), and close(). song_request() sends a TCP pocket to Server via TCP sockets (IPC) from the socket library. receive_notification() establishes a connection and communication channel to the message broker, RabbitMQ, running on localhost. It creates a temporary queue that binds to each artist's routing key in the favorite artlist link. It begins consuming messages without blocking the rest of the client program via a background listener thread. Lastly, close() closes any open connections. Server.py has three functions: start(), handle_client(), and notify_clients(). start() creates a TCP socket; binds to a host and port; listens for incoming client requests, and spawns a new thread/handler for each connection. handle_client() receives the song name from Client and sends a song stream message back. notify_clients() publishes updates to RabbitMQ so all subscribed clients get notified. Notifications has the publish_artist_message() and close_connection() functions. publish_artist_message() builds a routing key ("artist.Taylor_swift") so clients subscribed to that artist will receive the notification. It also creates a message payload and sends it to RabbitMQ. close_connection() safely closes the open connection to RabbitMQ.

2. **Design Decisions**:

This program is a publish-subscribe model that employs a topic subscription filter to manage message delivery efficiently. Restful APIs, such as Flask, were not necessary for a music playlist application (we emailed Professor Ponce). In this design, Client connects to Server through a TCP handshake ensuring a working communication channel before message exchange begins.

The system leverages RabbitMQ as the data broker, which handles message queuing, routing, and delivery between publishers (Notification) and subscribers (Client). This architecture provides flexibility and fault tolerance, allowing multiple clients to exchange information asynchronously while maintaining message integrity and order.

3. **Implementation Summary**:

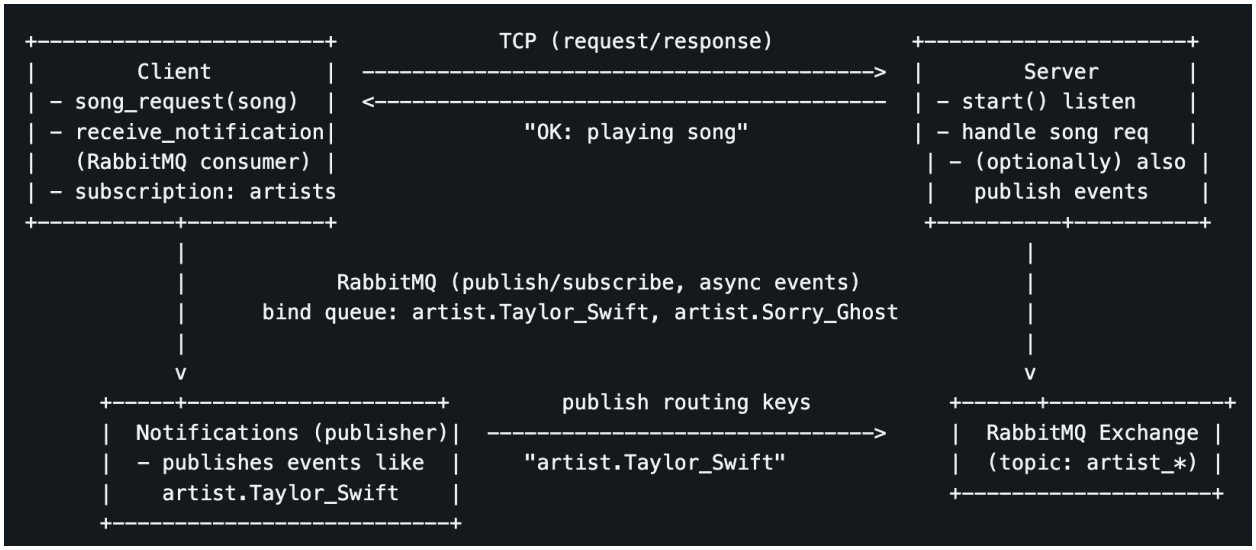
In our implementation of our code, we utilized the libraries sockets, threading, and RabbitMQ in order to accomplish this milestone. With the sockets' library, we established both a server-side and a client-side(server.py & client.py) as a way to tackle interprocess communication, particularly TCP pockets to servers. We utilized threading in order to handle multiple connections along the server we've created, and to showcase the scalability and strength of our architecture. Lastly, we utilized RabbitMQ library to implement a publisher-subscribe system, which we utilized to create a notification system that represents the same notifications seen when an artist releases a new song or album.

4. **Challenges & Resolutions**:

When we started the TCP server, it crashed with an OS error (Address already in use). This meant that something else on our machine was already listening on port 5000, and causing our server to be unable to bind() to it. We checked which conflicting process was using the port by using the command “lsof -nP -iTCP:5000 -sTCP:LISTEN”. We got the PID and attempted to kill the process, however, the port remained unavailable. We resolved to switch to a free port (5001) instead and updated our code to use 5001 rather than fight whatever was using 5000. This immediately resolved the issue.

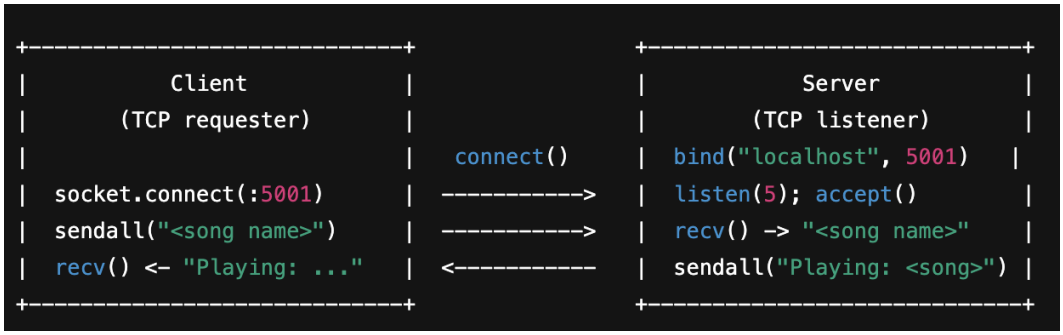
5. **Diagrams**:

- Communication flow between Client, Server, Notifications, & RabbitMQ:



- Publish-Subscribe System:

- TCP Client-Server:



Validation Summary

✅**Song Request Test Scenario**:

- From the list of songs displayed, the Client requests to play ‘Cruel Summer’.
- The Client opens a TCP connection which is verified by the output:

```
Which song do you want to play?: Cruel Summer
[Server] Listening on localhost:5001
```

- The Server reads the request and sends a response back. The Client reads the server's reply and prints it. Verified output:

```
server response: Playing song: Cruel Summer
```

✅**Notification Test Scenario**:

- The Client subscribes to the artists Taylor Swift and Sorry Ghost.
- The Publisher publishes updates for Taylor Swift, Sorry Ghost, and HUNTRX

```
Subscribed to artist updates: ['Taylor Swift', 'Sorry Ghost']

Publishing artist updates...
 [x] Sent 'artist.Taylor Swift':'New album released!'
 [x] Sent 'artist.Sorry Ghost':'New single out now!'
 [x] Sent 'artist.HUNTRX':'On tour this summer!'
```

- The Client receives notifications from Taylor Swift and Sorry Ghost, but not HUNTRX, since they are not subscribed to HUNTRX.

```
🎵 notification: {"type": "artist_update", "artist": "Taylor Swift", "message": "New album release
d!", "timestamp_utc": "2025-10-11T22:10:54.248426+00:00"}
Received song request: Cruel Summer

🎵 notification: {"type": "artist_update", "artist": "Sorry Ghost", "message": "New single out now
!", "timestamp_utc": "2025-10-11T22:10:54.248978+00:00"}
```

RabbitMQ: