DSCD Assignment 1

Anshak Goel - 2020283
Deeptorshi Mondal - 2020294
Sahil Goyal - 2020326

Part 1: Using gRPC to Implement an Online Shopping Platform

Language: Python
Files:
There are four files for this part of the assignment. They are market.py, seller1.py, client1.py, and Client-Market.proto
Assumptions:
A few assumptions have been made for this part of the assignment. They are listed below.
- Sellers have to register with the market first before they can perform any other functionality.
- Buyers can only buy one item for each buy request.
- Buyers can only add a rating to one item for each rating request.
- Buyers can only add one item to the wishlist for each wishlist request.
- Sellers can only post one item at a time on the market.
- Everything is being stored in process memory.

Explanation of Code:
There are four files in this part of the assignment. They are market.py, seller1.py, client1.py, and Client-Market.proto. Their functions and functionality are explained below.
Client-Market.proto:
We are using proto3 as the syntax. In this proto file, we describe the various services and their rpc's. We also define an enum and various messages. We create three services: BuyerInteracts, SellerInteracts, and MarketInteracts. They all have rpc's defined within them, corresponding to all the functions that we have implemented in the Python files. The enum is used for the various categories of items, and the messages define the format for the various messages sent between the market, seller, and buyer. After we compile the proto file, three Python files are created, which handle the communication between the market, clients, and sellers.
Market.py:
When we start the market server, we create dictionaries to store the information about the sellers, the items, the buyer's wishlist, and ratings for items. We begin listening on port 50051 and start the server. After that, the server automatically listens for any incoming messages from the buyers or the sellers and responds accordingly.
There are two classes, SellerInteracts and BuyerInteracts. They are explained below:
SellerInteracts:
This class is responsible for interacting with the sellers. It contains the following functions.

- RegisterSeller(): This function is responsible for the registration of a new seller. The seller sends its ID(uuid), IP address, and port number to the market. We check whether or not the seller is already registered and send the appropriate response.
- SellItem(): This function is called when the seller wants to post a new item on the market. The seller sends its details and the item details such as name, price, category, quantity, and description. We check whether or not the item is already registered with the market, and if it isn't, we add the item to our list of items.
- UpdateItem(): This function is called when a seller wants to update information about their item. They send information such as item ID, the new quantity, and the new price. We check whether the seller and item have been registered, and the seller is the one who registered the item. After that, we update the information about the item. Finally, we iterate through the wishlist and send a notification to the buyers who have added the item to their wishlist and inform them about the new details regarding the particular item.
- DeleteItem(): This function is called when a seller wants to remove a particular item from the market. They send their details and the ID of the item to the market. After that, we check whether the seller and item have been registered and the seller is the one who registered the item. After that, we remove that item from the list of items and send the success response to the seller.
- DisplaySellerItems(): This function is called when the seller wishes to see all of the items they have put on the market. Seller sends their details to the market, and we use that to check all of the items that have been added by that seller. We then send a list of those items back to the seller.

BuyerInteracts:
This class is responsible for the market's interaction with the buyers. The functions and functionality are explained below:

- SearchItem(): This function is called when the buyer wants to search for a particular item. The buyer sends details of the category of item or item name it wants. If these details are not provided, the market returns the list of all items; otherwise, only items that match these specifications are sent to the buyer.
- BuyItem(): This function is called when the buyer wants to buy an item from the market. We check whether there is enough quantity of the item, and then we let the buyer buy the item. We update the item quantity on each successful transaction and send the status to the buyer.
- AddToWishList(): This function is called when the buyer wants to add an item to the wish list. We check whether an item is registered or already in the wishlist and then proceed to add the item to the buyer's wishlist.
- RateItem(): This function is called when the buyer wants to rate an item. We check whether the item has already been rated, and then we give the item a rating and store it.

Client.py:
First, we create a new thread for listening to notifications from the market. We begin listening on port 50051 and start the server. After that, we use a menu system so that the buyer can decide what they want to do. The functions and functionality are explained below.

- notification(): This function is responsible for starting the server on the buyer's side so that they can listen to notifications from the market.

- NotifyClient(): This function receives notifications from the market regarding an update in an item updated by the seller that was in the buyer's wishlist. It prints out the new details for the item.
- searchItems(): This function sends a request to the market to search for items. Buyers can input the name and category of the item and will receive a list of all items that match their specifications.
- buyItem(): This function sends a request to the market to buy an item. The buyer can input the ID and quantity of the item, and the market will handle the request and send an appropriate response.
- wishListItem(): This function sends a request to the market to add an item to the wishlist. The buyer inputs the item ID, and the market adds it to their wishlist.
- rateItem(): This function sends a request to the market to add a rating to an item. Buyers can input item ID and rating, and the market will add the rating to the item.

Seller.py:
We assign a unique ID to every seller (uuid). Then, we create a thread that starts a server on the seller's side so that they can listen to and receive notifications from the market. We also use a menu system so that the seller can perform various actions like registering and selling items. The functions and functionality are listed below.

- notification(): This function is responsible for starting a server for the seller so that they can listen to and receive notifications from the market.
- NotifyClient(): This function receives notification from the market when a buyer buys an item that was sold by the seller. The function prints various details like description, quantity, etc.
- registerSeller(): This function is responsible for registering the seller with the market. The seller sends the market its information, such as IP address, port, and UUID.
- sellItem(): This function is used to add an item to the market. The seller inputs item details like name, price, category, etc, and sends them to the market.
- updateItem(): This function is used when a seller wants to update the details for an item. They input details such as new quantity and price and send them to the market, which updates the item information.
- deleteItem(): This function removes an item from the market. The seller inputs the item ID and sends the request to the market, which removes the item from its list.
- displayItems(): This information retrieves information for all the items that have been posted by the seller and prints the information.

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

Language: Python
Files:
There are three files for this part of the assignment. They are message_server.py, group.py, and user.py
Assumptions:
A few assumptions have been made for this part of the assignment. They are listed below.

- Any newly created group must first register with the server before it can communicate with the users. Upon being created, they first have to input their name, after which they will automatically contact the message server and register themselves.
- All users must first contact the server and get the list of all active groups before they can communicate with any groups. If any group has been created after a user has already contacted the server, the user will have to contact the server again and get the new list of groups if it wants to communicate with the newly created group.
- When users try to retrieve messages from a group, and they want the messages from after a particular time, they have to input the timestamp in HH:MM:SS format.

Explanation of Code:

There are three files in this part of the assignment. They are message_server.py, group.py, and user.py. The functions and functionality are explained below.

Message_server.py:

When the server starts, an empty dictionary is created to maintain the list of all registered groups. Two sockets are made for the groups and the users, and we use a Zmq poller to listen for any incoming messages on these sockets. When the poller detects incoming messages for either socket, we create a new thread to handle the incoming request.

- messageServer(): The server receives a message from a group that wants to register itself. It adds the group details to a dictionary used to maintain details of all registered groups and sends back a success message.
- getGroupList(): The server receives a message from a user who wants to get the list of all active groups. It processes the request and sends back the group list to the user.

Group.py

When the group server is started, it creates sockets for communicating with the message server and the users. After registering with the server, it communicates with the users and listens to their requests. It maintains a list of users that are a part of the group and a dictionary of all the messages sent to the group.

- server_function(): As soon as the group server is created, it registers itself with the message server. It sends details such as the group server's name, IP address, port, and uuid. Then, it receives a success message from the message server and prints that.
- user_function(): When the group server receives a request from a user, it begins processing the request. It determines which type of request it is (join group, leave group, send message, or get message) and performs necessary actions. It either adds the user to the group, removes them from the group, adds a message they sent to the group chat, or sends the list of messages requested by the user to them. It also performs necessary error handling, such as checking whether the user is a part of the group.

User.py:

The user can communicate with the message server to retrieve a list of all active groups or perform various actions with group servers. It also maintains a list of all currently active groups that it retrieved from the message server.

- server_function(): The user sends a message to the message server with its uuid requesting the list of all groups that have been registered with the server. It updates its list of groups with this list and then prints the list of all registered groups.

- joinGroup(): The user is presented with a list of groups and has to choose to join one of them. Then, they create a socket and send a message to the group. The group adds them to its list of users, and the user prints a success message.
- leaveGroup(): The user chooses which group they want to leave and sends a message to the group with a leave request. The group then sends it a response message, and the user prints success or failure depending on the response.
- getMessage(): The user chooses the group from which it wants to retrieve messages. They then have a choice to enter a timestamp if they want messages from after a certain time; otherwise, they can leave it empty if they want all messages. Then, they send a message to the group. After they receive the message list, the user prints out all the messages it received.
- sendMessage(): The user chooses which group it wants to send the message to. After that, it sends a message to the group and prints out success or failure, depending on the response from the group server.

Part 3: Building a YouTube-like application with RabbitMQ

Language: Python

Files: There are three files for this part of the assignment. They are youtube_server.py, youtuber.py, and user.py

Assumptions:
A few assumptions have been made for this part of the assignment. They are listed below.

- The server is always assumed to be up and running, and all the data is stored in process memory. Thus it is also assumed that the amount of data is less than the RAM allocated to the process.
- It names of YouTubers and users are assumed to be unique, so they can be considered more like YouTube channel handles which are unique for each user.
- It is assumed that once the message is delivered to the exchange, it is a successful delivery to the other side. Therefore the delivery from the exchange to the client is assumed to be successful always.
- It is also assumed that every time a user wants to subscribe or unsubscribe to a YouTuber, they must log out and then log in again with additional command line parameters for a subscription.
- If a user has subscribed to a YouTuber and then the user logs out, and if during that duration the YouTuber uploads some video and then the user logs in again but also unsubscribes to the said YouTuber, all the videos that the YouTuber uploaded while the user was logged out will still receive notifications for.

Explanation of Code:
There are three files in this part of the assignment. They are youtube_server.py, youtuber.py, and user.py. The functions and functionality are explained below.

Youtube_server.py:
The YouTube server connects the YouTuber and the User. All the YouTubers and the Users communicate with each other via the YouTube server only.

- main(): This method defines an exchange and a Rabbitmq connection and defines two queues over which the server will receive the messages from the YouTubers and the users. Also, these queues are exclusive; that is to say, they only allow access by the current connection. These queues are bound to the channel, and the server consumes requests.
- callback2(): This method takes the message from the users, which contains the request for the subscription or un-subscription to a YouTuber. This request is then stored in a dictionary which maintains the user as key and a list of all the YouTubers he/she has subscribed to as value.
- callback(): This method receives the message from the YouTubers, which contains the YouTuber's name and the Video they uploaded. Then, this method further calls the method sendNotification() to notify the users.
- sendNotification(): This method takes the YouTuber's name and the Video they uploaded as input and sends the notification or message to all the users who have subscribed to this YouTuber. This is done by making another connection and publishing the content. This time we have used a direct connection and the queues are also durable. This is done to make the connection persistent so that even if the user is offline, he/she may receive the notifications later when they come back online, and no messages are lost. The queues defined here are unique for each user, as we need to notify different users differently.

Youtuber.py
This file is representative of a YouTuber and has only one functionality to provide an interface to upload a YouTube video from any YouTuber. It has the following method:
publishVideo(): This method allows the YouTuber to send the video to the YouTube server, and from there, the video can be distributed to the Users. This method also defines a connection and channel, establishes the exchange and the queue, and then publishes the video and the YouTuber's name to the server. Here it was optional to define the exchange and the queue as we know, the server runs first and is always up, but still, it is defined as part of best practice.

User.py:
This file is the representation of the user. Here, users can log in in two different ways: with a subscription/un-subscription request and the other is just to receive the notifications from the server. If the user logs in just for the notifications, the updateSubscription method will be bypassed. The detailed description of each method is as follows:
updateSubscription(): This method takes command line arguments from the user and then passes these arguments over to the server. Here a connection is defined, and the queues and exchange are declared. We also have the queue type as exclusive, as we know the server is always up. Also, for communication or publishing videos, all the users use the same queue, which acts as a pipeline to the server. There is no need to define a separate queue for each

user here as that will not serve any extra functionality. Once we have the arguments, we publish the message, and then the user calls the receiveNotification() method for receiving notifications.
callback(): This method is triggered when the server sends any notification to the user, and then the user here simply prints or shows the message from the server.
receiveNotification(): This method is used for receiving the the video notification from the server of the YouTubers the user is subscribed to. Here also, we first define a connection, an exchange and a queue, but the queue here is persistent or durable and unique for each user. This queue must be unique for each user, as each user will receive different notifications. Also, the exchange here is direct, directing the message sent to a particular user onto his queue using the routing_key.