پویا صادقی مجید فریدفر اولدوز نیساری زهرا حجتی

gRPC communication Patterns

gRPC از شناخته شده ترین پیاده سازی‌های $Remote\ Procedure\ Call(RPC)$ محسوب می‌شود. یکی از دلایل محبوبیت آن را می‌تواند استفاده از protobuf دانست که با پشتیبانی از اسکیما و کتابخانه‌ها و ابزارهای زیاد، درکنار مستقل از زبان برنامه نویسی و پلتفرم بودن، مسیر فراگیری gRPC را هموار ساخته است و باعث شده در ساختارهای پیچیده نیز بتوان از آن استفاده کرد. بکار گیری protobuf و http/2 موجب پرفورمنس بالای gRPC شده است. همچنین استفاده از استریم‌های http/2 قابلیت های فراوان و همچنین توانایی هندل کردن تعداد زیادی فراخوانی همزمان را پدید آورده است. البته نیاز به همین دسترسی سطح پایین موجب شده است که مرورگرهای وب از gRPC پشتیبانی نکنند.

gRPC از چندین الگوی ارتباطی برای تسهیل انواع مختلف تعاملات بین کلاینت ها و سرورها پشتیبانی می کند که شامل موارد زیر می‌شود:

1. Unary RPC: ساده‌ترین شکل RPC است که در آن کلاینت یک درخواست را به سرور ارسال می‌کند و یک پاسخ را دریافت می‌کند. این مشابه ارتباط سنتی درخواست-پاسخ[1] HTTP است. این روش برای درخواست های ساده و که در آن مشتری انتظار یک پاسخ سریع را دارد (و نه چندین پاسخ دریافت کند یا چندین درخواست ارسال کند) مناسب است.
2. Server Streaming RPC: در این الگو، کلاینت یک درخواست واحد را به سرور ارسال می کند و استریمی از پاسخ ها را دریافت می کند و سرور چندین پیام را در زمان‌های مختلف ارسال خواهد کرد. این الگو زمانی مفید است که سرور نیاز به ارسال حجم زیادی از داده یا جریان مداوم داده در پاسخ به یک درخواست کلاینت دارد.
3. Client Streaming RPC: کلاینت جریانی از درخواست ها را به سرور ارسال می کند و سپس منتظر یک پاسخ از سرور می ماند. سرور درخواست ها را دریافت و پردازش می کند، و پاسخ را برای کلاینت می فرستد. این الگو مناسب برای سناریوهایی است که در آن کلاینت نیاز به ارسال مقدار زیادی داده به سرور دارد، مانند آپلود یا ثبت داده ها.
4. Bidirectional Streaming RPC: این الگو به کلاینت و سرور اجازه می دهد تا جریانی از پیام ها را به صورت ناهمزمان ارسال کنند. کلاینت جریانی از درخواست‌ها را ارسال می‌کند و سرور با جریانی از پیام‌ها به کلاینت پاسخ می‌دهد. این الگو برای سناریوهایی که نیاز به ارتباط بی‌درنگ دارند، مانند برنامه‌های چت یا بازی چند نفره مفید است.

بطور کلی می‌توان الگوهای مکاتبه gRPC را از جنبه‌های زیر تحلیل کرد:

[1] Request-Response

جنبه	Unary RPC	Server Streaming RPC	Client Streaming RPC	Bidirectional Streaming RPC
مورد استفاده	درخواست های ساده و یکباره	ارسال داده‌های حجیم از طرف سرور به صورت استریم	ارسال داده‌های حجیم از طرف کلاینت به صورت استریم	ارتباط Real-time و دوطرفه
پرفورمنس	سربار کم و پرفورمنس بالا در حداقل داده	پرفورمنس بالا در ارتباط حجیم سرور به کلاینت	پرفورمنس بالا در ارتباط حجیم کلاینت به سرور	پرفورمنس بالا در مکاتبات دوطرفه
تاخیر	تاخیر کم، پاسخ سریع و فوری	زمان پردازش سرور باید افزوده شود	زمان انتقال کلاینت باید افزوده شود	تأخیر شبکه و پردازش باید در نظر گرفته شود
جهت مکاتبه	یک درخواست، یک پاسخ	سرور به کلاینت	کلاینت به سرور	دو طرفه و از طرف هر دو
حجم داده مکاتبه‌ای	داده‌های کم حجم	داده حجیم سمت سرور	داده حجیم سمت کلاینت	جریان دو طرفه یا متغیر داده
سربار شبکه	کم	به دلیل تک درخواست کاهش یافته	به دلیل انتقال تک کاهش یافته	دو طرفه و متعادل
مثال استفاده	احراز هویت	بروز رسانی داده کلاینت، دانلود فایل(از طرف کلاینت)	آپلود فایل	پیامرسان‌ها و بازی‌ها

توضیحات ساختار کد

1) فایل proto

در هر پروژه gRPC یک فایل proto داریم که سرویس هایی که استفاده می شوند و ساختار های داده ای که استفاده می شوند را مشخص می کنند .

```
syntax = "proto3";

package ordermanagement;

service OrderManagement {

    //Unary
    rpc getOrder (OrderRequest) returns (OrderResponse) {}
```

```
    //Server Streaming
    rpc searchOrders (OrderRequest) returns (stream OrderResponse) {}

    //Client Streaming
    rpc updateOrders (stream UpdateOrderRequest) returns
(UpdateOrderResponse) {}

    //Bidirectional Streaming
    rpc processOrders (stream OrderRequest) returns (stream
ShipmentResponse) {}
}

message OrderRequest {
    string order_name = 1;
}

message OrderResponse {
    string item_name = 1;
    string timestamp = 2;
}

message UpdateOrderRequest {
    string old_order_name = 1;
    string new_order_name = 2;
}

message UpdateOrderResponse {
    string confirmation = 1;
}

message ShipmentResponse {
    string id = 1;
    repeated string orders = 2;
}
```

در ابتدا مشخص می کنیم که بناست از ورژن 3 ، protocol buffers استفاده کنیم .
سپس package ای با نام ordermanagement ایجاد می کنیم .

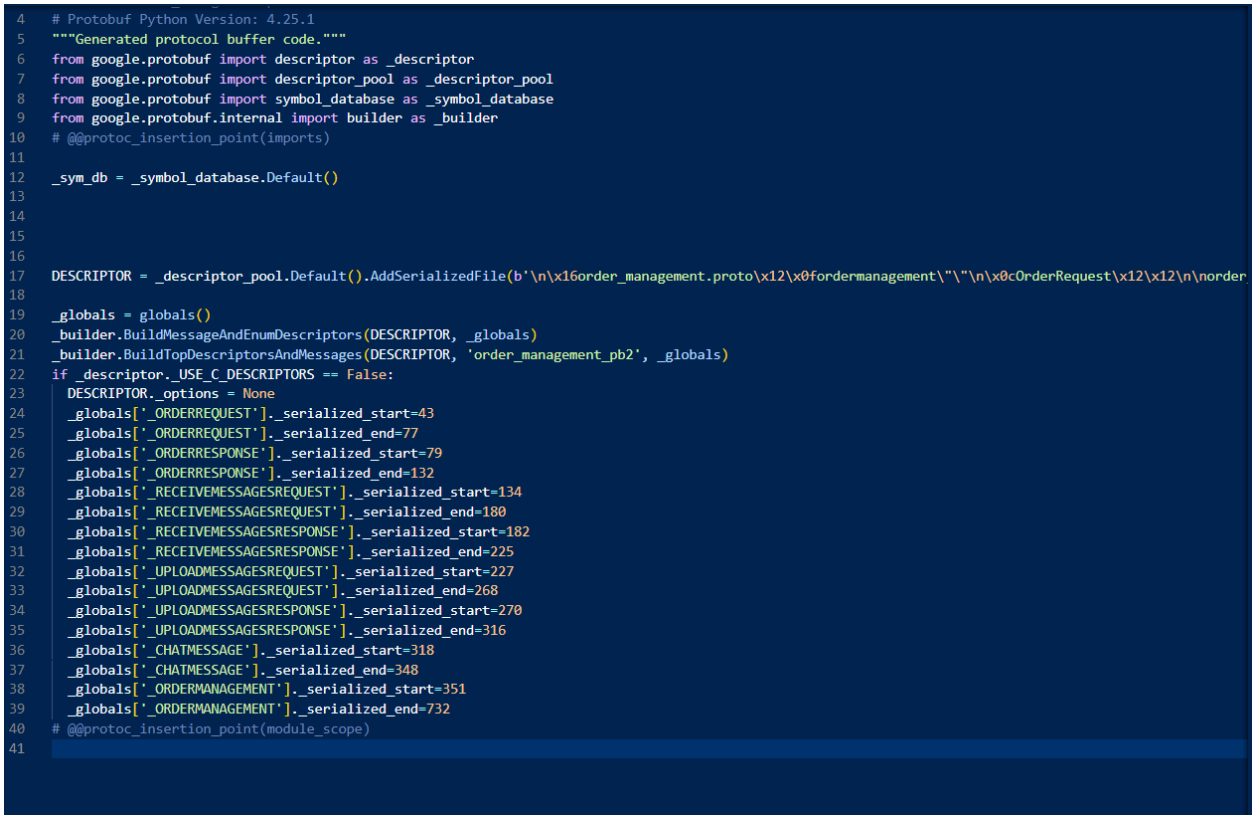
سپس یک سرویس RPC به نام OrderManagement تعریف می کنیم که شامل 4 متد RPC است :

- getOrder : متد از نوع unary است که پیام 'OrderRequest' را می گیرد و پیام 'OrderResponse' را بر می گرداند.
- searchOrders : متد از نوع server streaming است که ReceiveMessageRequest می گیرد و یک stream به نام RecieveMassageResponse بر می گرداند .
- updateOrders : متد از نوع client streaming است که stream ای به نام UploadMessagesRequest می گیرد و یک stream به نام UploadMessagesResponse بر می گرداند .
- processOrders : یک stream دو طرفه RPC است که stream ای به نام ChatMessage می گیرد و بر می گرداند .

حال در بخش بعد سراغ تعریف ساختار message ها بین کاربر و سرور می رویم :

- OrderRequest : فقط دارای فیلد اسم است .
- OrderResponse : دارای دو فیلد اسم آیتم و timestamp است.
- RecieveMessagesRequest : تنها دارای تعداد پیام هاست .
- RecieveMessagesResponse : دارای خود پیام است .
- UploadMessagesRequest : مجددا این پیام هم دارای فیلد تکرار شده پیام است.
- UploadMessagesResponse : دارای فیلد confirmation است .
- ChatMessage : تنها دارای فیلد message است.

2) فایل های code generation



```
# Protobuf Python Version: 4.25.1
"""Generated protocol buffer code."""
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import symbol_database as _symbol_database
from google.protobuf.internal import builder as _builder
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()




DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x16order_management.proto\x12\x0fordermanagement\"\"\n\x0cOrderRequest\x12\x12\n\norder

_globals = globals()
_builder.BuildMessageAndEnumDescriptors(DESCRIPTOR, _globals)
_builder.BuildTopDescriptorsAndMessages(DESCRIPTOR, 'order_management_pb2', _globals)
if _descriptor._USE_C_DESCRIPTORS == False:
  DESCRIPTOR._options = None
  _globals['_ORDERREQUEST']._serialized_start=43
  _globals['_ORDERREQUEST']._serialized_end=77
  _globals['_ORDERRESPONSE']._serialized_start=79
  _globals['_ORDERRESPONSE']._serialized_end=132
  _globals['_RECEIVEMESSAGESREQUEST']._serialized_start=134
  _globals['_RECEIVEMESSAGESREQUEST']._serialized_end=180
  _globals['_RECEIVEMESSAGESRESPONSE']._serialized_start=182
  _globals['_RECEIVEMESSAGESRESPONSE']._serialized_end=225
  _globals['_UPLOADMESSAGESREQUEST']._serialized_start=227
  _globals['_UPLOADMESSAGESREQUEST']._serialized_end=268
  _globals['_UPLOADMESSAGESRESPONSE']._serialized_start=270
  _globals['_UPLOADMESSAGESRESPONSE']._serialized_end=316
  _globals['_CHATMESSAGE']._serialized_start=318
  _globals['_CHATMESSAGE']._serialized_end=348
  _globals['_ORDERMANAGEMENT']._serialized_start=351
  _globals['_ORDERMANAGEMENT']._serialized_end=732
# @@protoc_insertion_point(module_scope)

```

این فایل توسط کامپایلر protocol buffer تولید می شود . در ابتدای آن ماژول های ضروری برای آن import می شوند از proto buf .

در بخش descriptor ، یک نمایش از تعریف پروتکل بافر به صورت serialized ارائه می شود . این بخش نوع پیام ها به همراه فیلد هایشان را مشخص می کند.
در ادامه از تابع global برای دسترسی متغیر های جهانی استفاده می شود و دستوری های builder برای ساختن پیام ها بر اساس serialized descriptor است .
و بعد متغیر های global یکی یکی ست می شوند . (شروع و پایان را مشخص می کنند .)

3) فایل server

```
import grpc
from concurrent import futures
import order_management_pb2
```

```
import order_management_pb2_grpc
from datetime import datetime

class
OrderManagementServicer(order_management_pb2_grpc.OrderManagementServicer):
    def __init__(self):
        self.server_orders = ['banana', 'apple', 'orange', 'grape', 'red
apple', 'kiwi', 'mango', 'pear', 'cherry', 'green apple']
        self.processedShipmentId = 1

    def getOrder(self, request, context):
        order_name = request.order_name.lower()

        matching_order = None
        for order in self.server_orders:
            if order_name == order.lower():
                matching_order = order
                break

        if matching_order:
            item_name = matching_order
            timestamp = str(datetime.now())
            return order_management_pb2.OrderResponse(item_name=item_name,
timestamp=timestamp)
        else:
            return order_management_pb2.OrderResponse(item_name="Item not
found", timestamp="")

    def searchOrders(self, request, context):
        order_name = request.order_name.lower()
        matching_orders = [order for order in self.server_orders if
order_name in order.lower()]

        if not matching_orders:
            yield order_management_pb2.OrderResponse(item_name="Item not
found", timestamp=str(datetime.now()))
        else:
            for order in matching_orders:
                yield order_management_pb2.OrderResponse(item_name=order,
timestamp=str(datetime.now()))

    def updateOrders(self, request_iter, context):
        confirmation_message = "Updates: "
```

```
        for request in request_iter:
            for i in range(len(self.server_orders)):
                if self.server_orders[i] == request.old_order_name:
                    self.server_orders[i] = request.new_order_name
                    confirmation_message += f"{request.old_order_name}
changed to {request.new_order_name} | "
                    break
            else:
                confirmation_message += f"{request.old_order_name} not
found | "

        return
order_management_pb2.UpdateOrderResponse(confirmation=confirmation_message)

    def processOrders(self, request_iterator, context):
        shipmentOrders = []
        for request in request_iterator:
            if request.order_name not in self.server_orders:
                continue

            shipmentOrders.append(request.order_name)
            if len(shipmentOrders) == 3:
                yield order_management_pb2.ShipmentResponse(id =
str(self.processedShipmentId), orders = shipmentOrders)
                shipmentOrders = []
                self.processedShipmentId += 1

        if shipmentOrders:
            yield order_management_pb2.ShipmentResponse(id =
str(self.processedShipmentId), orders = shipmentOrders)

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))

order_management_pb2_grpc.add_OrderManagementServicer_to_server(OrderManage
mentServicer(), server)
    server.add_insecure_port("localhost:50051")
    server.start()
    server.wait_for_termination()

serve()
```

در ابتدا کتابخانه های مربوط به gRPC و protocol buffer و dateTime را ایمپورت می کنیم .
در ادامه یه کلاس به نام OrderManagmentService تعریف می کنیم و منطق مرتبط به سرویس پیاده سازی شده در OrderManagment را پیاده سازی می کنیم . (
متد های این کلاس :

- getOrder : متدی برای هندل کردن درخواست از جنس unary است که بر اساس اسم بازیابی را انجام می دهد . این متد دنبال تطابق سفارش با لیستی که از قبل تعیین کرده ایم می گردد ، یک timestamp تولید می کند و پیام OrderResponse را بر می گرداند .
- searchOrder : این متد ریکوست های مبتنی بر server streaming برای انجام جست و جو را هندل می کند . کل سفارشات را stream می کند و به کلاینت ارسال می کند.
- updateOrderMethod : این متد ریکوست های client streaming را برای به روز کردن سفارش ها هندل می کند . این متد چندین پیام ریکوست دریافت می کند و یک پیام تایید UploadMessageResponse برمیگرداند .
- processOrders : این متد streaming به صورت bidirectional را هندل می کند . این متد پیام را از کلاینت می گیرد ."Server Recieved " را به ابتدای تمام پیام ها اضافه می کند و آن ها را به کلاینت stream می کند.

در ادامه تابع سرور را تعریف می کنیم که سرور gRPC را تنظیم و شروع می کند . این تابع orderManagmentServicer را به سرور اضافه می کند ، آدرس لوکال هاست را مشخص می کند ، سرور را شروع می کند و تا زمانی که terminate شود wait می کند .

در انتها هم در بلام main تابع server ران می شود.

4) فایل client

```
import grpc
import order_management_pb2
import order_management_pb2_grpc

def run():
    with grpc.insecure_channel("localhost:50051") as channel:
        client = order_management_pb2_grpc.OrderManagementStub(channel)
```

```
        while True:
            print("1. Get Order - Unary")
            print("2. Search Orders - Streaming Server")
            print("3. Update Orders - Client Streaming")
            print("4. Process Orders - Birdirectional Streaming")
            choice = input("Enter your choice (1-4): ")

            if choice == "1":
                order_name = input("Enter order name: ")
                response =
client.getOrder(order_management_pb2.OrderRequest(order_name=order_name))
                print(response)
            elif choice == "2":
                order_name = input("Enter order name: ")
                responses =
client.searchOrders(order_management_pb2.OrderRequest(order_name=order_name
))
                for response in responses:
                    print(response)
            elif choice == "3":
                names = []
                while True:
                    old_name = input("Enter order name (leave empty to
stop): ")
                    if not old_name:
                        break
                    new_name = input("Enter new order name: ")
                    names.append((old_name, new_name))
                response =
client.updateOrders(iter([order_management_pb2.UpdateOrderRequest(old_order
_name=old_name, new_order_name=new_name) for old_name, new_name in names]))
                print(response)
            elif choice == "4":
                order_names = []
                while True:
                    order_name = input("Enter order name (leave empty to
start processing): ")
                    if not order_name:
                        break
                    order_names.append(order_name)
                responses =
client.processOrders(iter([order_management_pb2.OrderRequest(order_name=ord
```

```
er_name) for order_name in order_names]))
                print(responses)
                for response in responses:
                    print(response)
            else:
                print("Invalid choice!")
                continue

run()
```

در این فایل بخش مربوط به کلاینت پیاده سازی شده است. در این پروژه ما چهار الگوی ارتباطی gRPC را پیاده کرده ایم و کلاینت به تناسب، باید از الگوی مناسب استفاده کند. در ابتدا کتابخانه های لازم مربوط به grpc را در محیط برنامه وارد میکنیم. توابع فانکشنال ما توسط .proto و با کمک کامپایلر مرتبط، ساخته شده اند. در ابتدا برنامه یک چنل gRPC جهت ارتباط با سرور، که برروی میزبان محلی و پورت 50051 قبلا مستقر شده، ایجاد می‌کند. برنامه کلاینت نقش عملگر سمت کاربر را دارد و چهار عملیات از سمت کاربر را پشتیبانی میکند:

1. Get Order: این عملگرد که بر الگوی Unary RPC می‌باشد، با دریافت شناسه سفارش (در اینجا، نام سفارش) از سرور اطلاعات مربوطه را می‌گیرد.
2. Search Orders: این عملگر بر الگوی Server Streaming RPC می‌باشد، بدین صورت که یک درخواست به سرور (در اینجا، به جهت جست و جوی سفارش‌ها) به سرور ارسال میکند، حال سرور از یک استریم برای پاسخ استفاده می‌کند، بدین صورت که برای هر سفارش پیدا شده، یک پاسخ از سرور به کلاینت فرستاده خواهد شد.
3. Update Orders: این عملگر مطابق الگوی Client Streaming RPC طراحی شده است، بدین صورت که کلاینت در استریمی از درخواست‌ها به سرور درخواست بروزرسانی سفارش‌ها را ارسال می‌کند و در انتها سرور وضعیت عملیات‌های بروزرسانی را در یک پاسخ برای کلاینت ارسال خواهد کرد.
4. Process Orders: در این عملگر، هدف استفاده از الگوی Bidirectional Streaming RPC می‌باشد. بدین صورت که هر دو کلاینت و سرور از مسیج استریمینگ استفاده می‌کنند؛ بدین صورت که کلاینت تعدادی سفارش که کاربر وارد کرده است را به سرور ارسال می‌کند و سرور قرار است شیپمنت مشخص کند(درنهایت به ازای هرسفارش یک پیام که مشخص کننده دریافت سفارش است ارسال می‌کند).

اجرای کد سرور و کلاینت

Unary RPC

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 1
Enter order name: apple
item_name: "apple"
timestamp: "2024-04-15 01:04:31.061110"
```

سناریوی موفق. response شامل نام سفارش و timestamp است.

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 1
Enter order name: watermelon
item_name: "Item not found"
```

حالتی که سفارش وارد شده در لیست server_orders موجود نیست. response شامل پیام خطا است.

Server Streaming RPC

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 2
Enter order name: apple
item_name: "apple"
timestamp: "2024-04-15 01:08:03.887377"

item_name: "red apple"
timestamp: "2024-04-15 01:08:03.887980"

item_name: "green apple"
timestamp: "2024-04-15 01:08:03.889507"
```

جستجوی apple، که response شامل لیست سفارشات match شده است (یک request و چند (stream) تا response)

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 2
Enter order name: g
item_name: "orange"
timestamp: "2024-04-15 01:10:09.820877"

item_name: "grape"
timestamp: "2024-04-15 01:10:09.820877"

item_name: "mango"
timestamp: "2024-04-15 01:10:09.821880"

item_name: "green apple"
timestamp: "2024-04-15 01:10:09.821880"
```

جستجوی g و لیست سفارشات match شده (سفارشاتی که شامل حرف g میباشند)

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 2
Enter order name: bananna
item_name: "Item not found"
timestamp: "2024-04-15 01:09:11.133468"
```

جستجوی bananna. همانطور که مشاهده می‌کنید، هیچ سفارشی پیدا نشده و response شامل پیغام خطای مناسب است.

Client Streaming RPC

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 3
Enter order name (leave empty to stop): apple
Enter new order name: a
Enter order name (leave empty to stop): banana
Enter new order name: b
Enter order name (leave empty to stop):
confirmation: "Updates: apple changed to a | banana changed to b | "
```

در این جا، تنها آپدیتی که روی اجناس می‌توانستیم انجام دهیم، این بود که نام آن ها را تغییر دهیم. در این مثال، نام apple را به a و نام banana را به b تغییر داده‌ایم (یک stream از requestها). از response دریافت شده از سرور، متوجه میشویم که تغییرات به درستی اعمال شده‌اند.

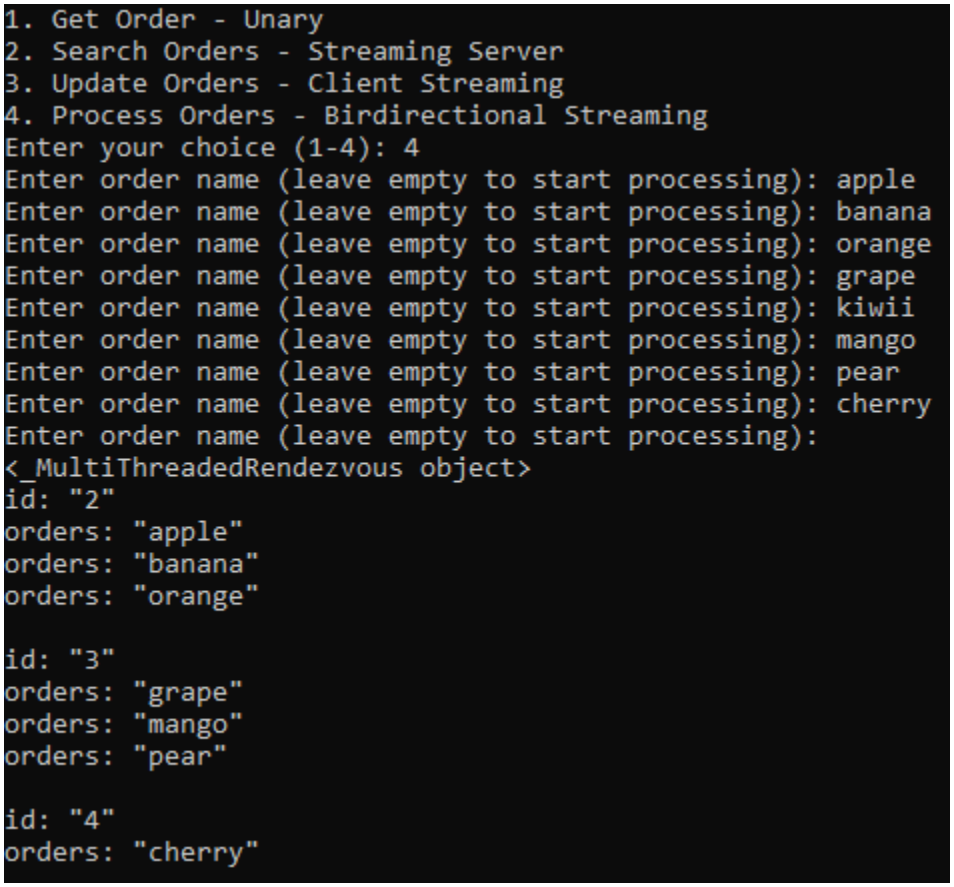
```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 1
Enter order name: apple
item_name: "Item not found"

1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 1
Enter order name: a
item_name: "a"
timestamp: "2024-04-15 01:21:27.064731"
```

برای اطمینان حاصل کردن، کلمات apple و a را به عنوان ورودی متود getOrder میدهیم. نتیجه را در اسکرین شات بالا مشاهده میکنید.

```
1. Get Order - Unary
2. Search Orders - Streaming Server
3. Update Orders - Client Streaming
4. Process Orders - Birdirectional Streaming
Enter your choice (1-4): 3
Enter order name (leave empty to stop): a
Enter new order name: apple
Enter order name (leave empty to stop): b
Enter new order name: banana
Enter order name (leave empty to stop): o
Enter new order name: orange
Enter order name (leave empty to stop):
confirmation: "Updates: a changed to apple | b changed to banana | o not found | "
```

در ادامه، نام a را دوباره به apple و b را به banana تبدیل میکنیم. از response دریافتی از سرور متوجه میشویم که تغییرات به درستی اعمال شده است. همچنین مورد آخر (o)، باعث تغییر خاصی نشده. چون o در server_orders موجود نبوده.

Bidirectional Streaming RPC



در اینجا فرض کرده ایم هر بسته ای از کالاها که توسط فروشگاه process میشود، حداکثر حاوی 3 نوع جنس متفاوت است. پس سفارشات مشتری (یک stream از requestها) در بسته های نهایتا سه تایی (یک stream از responseها) گذاشته شده و به او داده میشود. هر بسته یک id یکتا هم دارد. همچنین هر orderای که توسط مشتری ثبت میشود قبل از process شدن ابتدا بررسی میشود که در server_orders موجود است یا خیر. مثلا اینجا kiwii در server_orders موجود نیست، برای همین در هیچ بسته ای گذاشته نشده است و process نشده است.

نکاتی درباره پیشبرد پروژه و تقسیم کار :

در این پروژه پیاده سازی ساختار اصلی کد بر عهده زهرا بود . سپس در ادامه وظیفه دیباگ و تست کد بر عهده مجید بود . وظیفه تست نهایی ، تحلیل کد های نوشته شده به همراه نوشتن گزارشکار بر عهده اولدوز و پویا بود .