به نام خدا

مبانی رایانش توزیع شده

گزارش پروژه اول

مرضیه حریری ***810199404***

فاطمه زهرا برومندنیا ***810100094***

نیما تاجیک ***810100104***

امیرحسین راحتی ***810100144***

مقدمه

در این تمرین به بررسی فریمورک **gRPC** و طراحی یک سیستم **clinet** و **server** مبتنی بر **RPC** به وسیله این فریمورک میپردازیم .

این فریمورک یک ابزار بسیار کاربردی و بهینه برای برقراری **Remote procedure call** می باشد که با اکثر زبان های برنامه نویسی سازگاری دارد .

روش های برقراری ارتباط در gRPC

این فریمورک چهار روش مطرح برای برقراری ارتباط از طریق **remote procedure call** دارد که هر یک به اختصار بررسی می شود.

- **Send Message(Unary RPC)**

در این روش که ساده ترین روش ارتباط بین کلاینت و سرور است ، یک پیام از طرف کلاینت به سمت سرور ارسال می شود و سرور بلافاصله پاسخ درخواست را در یک پیام ارسال می کند.
ویژگی این روش این است که توسط هرکدام از کلاینت یا سرور ، فقط یک پیام ارسال میشود و فقط یک پیام دریافت میشود.
قطعه کد معادل این نوع درخواست در زبان **Go** به صورت زیر است :

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
```

- **Receive Messages(Server Streaming RPC)**

در این حالت ، به ازای یک پیام از طرف کلاینت ، چندین پاسخ به صورت پشت سر هم برای کلاینت ارسال میشود . این روش برای زمانی که به ازای یک درخواست کلاینت ، چندین پاسخ وجود دارد و تعداد آن را نمی دانیم مناسب تر است . همچنین باعث افزایش performance در ارتباط و پردازش در دریافت کننده میشود. چون کلاینت میتواند در این حین که منتظر پاسخ دوم است ، پردازش خود را روی پاسخ های دریافت شده شروع کند . در حالی که اگر از روش unary RPC استفاده میکردیم ، نیاز بود کلاینت مدت زیادی منتظر پاسخ بزرگی در ازای درخواستش بماند و عملا زمان زیادی تلف می شد. یک کاربرد برای این روش میتواند یک یک کاربرد برای این روش میتواند یک chat application باشد که در ازای درخواست کلاینت ، تعداد زیادی از پیام ها را برای آن برمیگرداند.

یکی از ایرادات این روش این است که کلاینت برای ارسال پیام بعدی باید منتظر بماند تا همه پاسخ های پیام قبلی اش را دریافت کرده باشد که این یعنی blocking و میتواند باعث افت performance در سیستم شود.

در پایان جریان پاسخ ها ، سرور با ارسال metadata خاصی پایان تماس را به اطلاع کلاینت می رساند . تصویر زیر معادل این نوع درخواست در زبان Go است :

```
service ChatService {
    rpc StreamMessages (MessageRequest) returns (stream MessageResponse);
  }
```

- **Upload Messages (client Streaming RPC)**

در این حالت ، برخلاف حالت قبلی ، کلاینت چندین پیام را راسال میکند ، اما در پایان دریافت پیام ها توسط سرور ، سرور به آن پاسخ میدهد . تفاوت این روش با روش قبل میتواند در قالب یک مثال ساده قابل بررسی باشد : فرض کنیم در کلاینت میخواهد یک یا چندین فایل را به سمت سرور ارسال کند یا چندین دیتا را تحویل سرور بدهد . با انتخاب روش client Streaming ، کلاینت میتواند پیام ها را به صورت کوتاه تر به سرور تحویل دهد تا مشکلات connection نتواند باعث fail شدن کل روند ارسال شود . از طرفی سرور فرصت بررسی و

پردازش همه درخواست ها را پیدا میکند و ناگهان با حجم زیاد پیام ها و **bottleneck** مواجه نمیشود .

یک مشکل دیگر که در این روش بر طرف میشود این است که وقتی کلاینت درخواست هارا به صورت یکجا (روش **unary RPC**) ارسال میکند ، ناگهان سرور باید بخش زیادی از توابع و دستورات را اجرا کند و این میتواند باعث شود بقیه کلاینت ها دچار **blocking** شود . اما در این حالت ریسک **block** شدن دیگر کلاینت ها کاهش می یابد.

در پایان جریان پاسخ ها ، سرور با ارسال **metadata** خاصی پایان تماس را به اطلاع سرور می رساند . تصویر زیر معادل این نوع درخواست در زبان **Go** است :

```
service FileUploadService {
    rpc UploadFile (stream FileRequest) returns (FileResponse);
  }
```

- **Chat Stream (Bidirectional Streaming RPC)**

این روش بیشترین انعطاف پذیری بین همه روش هارا دارد . به این صورت که هر دو سمت کلاینت و سرور میتوانند یک جریان از پیام ها را به سمت هم بفرستند و در عین حال از دیگری پاسخ را دریافت کنند . این روش برای سیستم های **real Time** که نیاز دارند به صورت پیوسته با هم ارتباط برقرار کنند و حداقل **blocking** را داشته باشند مناسب است .

یکی از چالش های این روش این است که به ازای هر پیامی که کلاینت از سرور دریافت میکند نیاز است یک فرآیند چک کردن انجام شود که مشخص شود این پاسخ متعلق به کدام درخواست بوده است . این مشکل زمانی نمود پیدا می کند که کلاینت برای پاسخ ها ترتیبی قائل نباشد و اهمیتی هم نداشته باشد .

یک مثال برای کاربرد این روش میتواند سرویس تماشای محتوای آنلاین باشد که در آن کلاینت نیاز دارد مرتبا درخواست هایی برای دریافت فریم های خاصی از فیلم را بدهد و سرور مرتبا فریم هارا برای کلاینت ارسال کند .

شیوه تعریف این نوع ارتباط در زبان **go** به صورت زیر است :

```
service ChatService {
    rpc Chat (stream ChatRequest) returns (stream ChatResponse);
  }
```

پیاده سازی Distributed ordering system using gRPC

برای بخش پیاده سازی ، نیاز به سه بخش client و server و بخش proto داریم که هرکدام به صورت زیر پیاده سازی می شود

proto/ordering.proto

proto > ordering.proto

```
syntax="proto3";
option go_package = "./proto";
package order_service;

service OrderService {
    // server streaming RPC
    rpc GetOrderServerStreaming(NamesList) returns (stream OrderResponse);
    // bidirectional streaming RPC
    rpc GetOrderBidirectionalStreaming(stream OrderRequest) returns (stream OrderResponse);
}


message OrderRequest {
    string name = 1;
}

message OrderResponse {
    string message = 1;
}

message NamesList {
    repeated string names = 1;
}
```

در این فایل ابتدا سرویس OrderService تعریف شده که در آن همه تابع‌هایی که از gRPC استفاده می‌کنند تعریف شده‌اند.

یعنی GetOrderServerStreaming و GetOrderBidirectionalStreaming . سپس سه نوع پیام به نام‌های OrderRequest, OrderResponse که هر کدام شامل یک string اند و NamesList که شامل لیستی از استرینگ‌هاست تعریف شده‌اند.

server/main.go

server > main.go

```go
package main

import (
    "log"
    "net"

    pb "github.com/m-hariri/basic-go-grpc/proto"
    "google.golang.org/grpc"
)
const (
    port = ":8080"
)
type orderServer struct {
    pb.OrderServiceServer
}

var ServerOrders = []string{"banana", "apple", "orange", "grape", "red apple",
"kiwi", "mango", "pear", "cherry", "green apple"}

func main() {

    lis, err := net.Listen("tcp", port)
    if err != nil {
        log.Fatalf("Failed to start server %v", err)
    }
    grpcServer := grpc.NewServer()

    pb.RegisterOrderServiceServer(grpcServer, &orderServer{})
    log.Printf("Server started at %v", lis.Addr())

    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("Failed to start: %v", err)
    }
}
```

در این بخش پس از import کردن مسیرهای مورد نیاز و مشخص کردن پورت 8080 در قسمت main ابتدا با استفاده از تابع listen سرور روی پورت مورد نظر گوش می‌دهد اگر نتیجه ناموفق بود ارور می‌دهد در غیر این صورت با استفاده از ()NewServer یک سرور جدید می‌سازد و با استفاده از RegisterOrderServiceServer سرویس را رجیستر می‌کند. lis در تابع serve همان پورت است و سرور gRPC باید از اینجا شروع کند.

server/server_stream.go

server > server_stream.go

```go
package main
import (
    "log"
    "strings"
    "time"
    pb "github.com/m-hariri/basic-go-grpc/proto"
)
func (s *orderServer) GetOrderServerStreaming(req *pb.NamesList, stream pb.OrderService_GetOrd
    log.Printf("Got request with names: %v", req.Names)
    for _, name := range req.Names {
        found := false

        for _, item := range ServerOrders {
            if strings.Contains(item, name) {
                found = true

                res := &pb.OrderResponse{
                    Message:  "Item found: " + item,
                }
                if err := stream.Send(res); err != nil {
                    return err
                }
                break
            }
        }
        if !found {
            res := &pb.OrderResponse{
                Message: "Item not found for: " + name,
            }
            if err := stream.Send(res); err != nil {
                return err
            }
        }

        time.Sleep(2 * time.Second)
    }

    return nil
}
```

در این بخش تابع **GetOrderServerStreaming** تعریف شده است که توسط کلاینت به صورت ریموت قابل صدا زدن است این تابع لیستی از نام‌ها را به همراه **stream** می‌گیرد و **nil** برمی‌گرداند. داخل تابع ابتدا لیست نام‌ها که توسط کلاینت فرستاده شده پرینت می‌شود. سپس داخل فور هر نامی در **NamesList** با هر ایتمی در **ServerOrders** مقایسه میشود و اگر ایتم شمل بخشی از نام بود اسم آن به همراه تایم استمپ‌ش برای کلاینت فرستاده میشود. (توجه شود که از **OrderResponse** برای فرستادن جواب استفاده شده) اگر پیدا نشد پیام یافت نشدن به کلاینت فرستاده میشود. در اخر دو ثانیه دیلی **simulate** میشود تا یک پراسس طولانی را شبیه سازی کند.

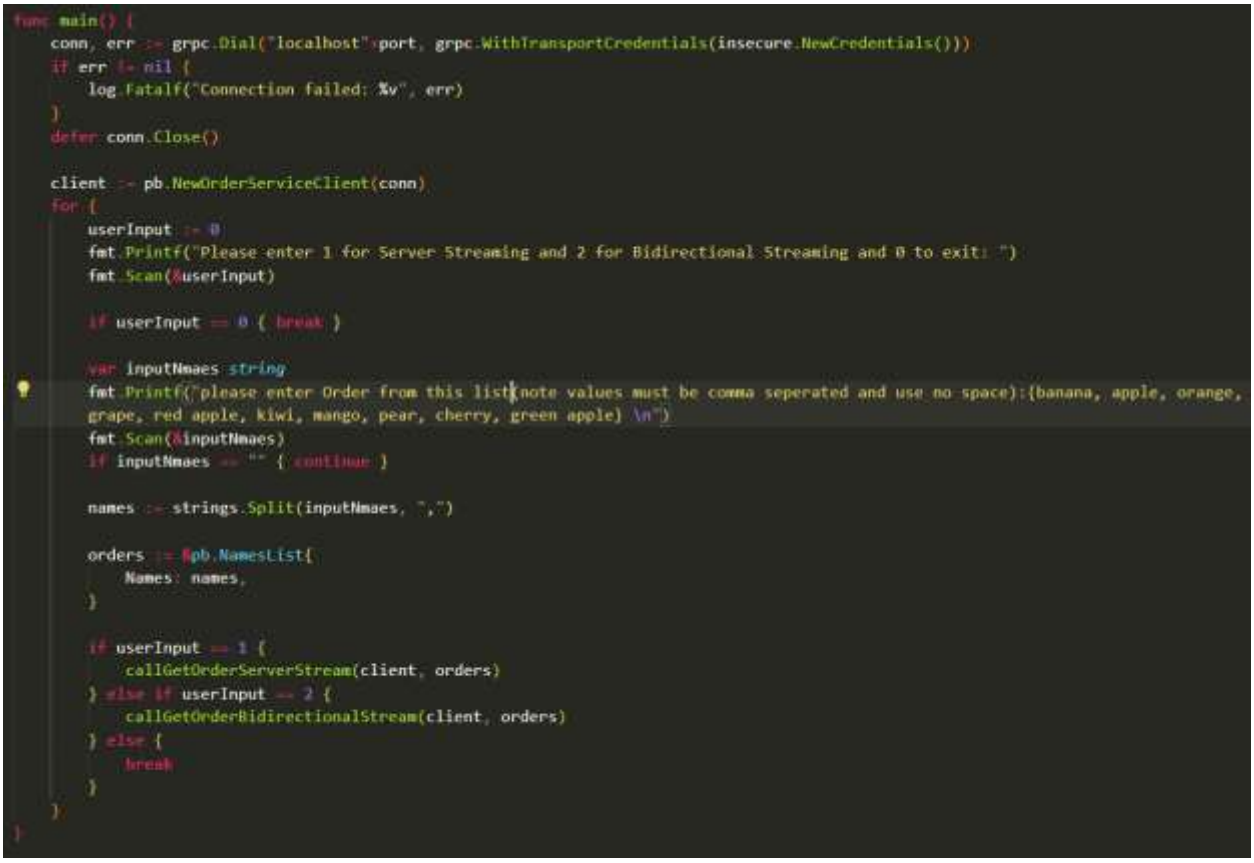
server/bi_stream.go

server > bi_stream.go

```go
package main
import (
    "io"
    "log"
    "strings"
    "strconv"

    pb "github.com/m-hariri/basic-go-grpc/proto"
)

func (s *orderServer) GetOrderBidirectionalStreaming(stream pb.OrderService_GetOrderBidirectionalStreamingServer) error {
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return nil
        }
        if err != nil {
            return err
        }

        log.Printf("Got request with name : %v", req.Name)

        found := false
        for i, item := range ServerOrders {
            if strings.Contains(item, req.Name) {
                found = true
                res := &pb.OrderResponse{
                    Message:  "Item found! Item number: " + strconv.Itoa(i+1) + ", Item name: " + item,
                }
                if err := stream.Send(res); err != nil {
                    return err
                }
                break
            }
        }
        if !found {
            res := &pb.OrderResponse{
                Message: "Item not found for: " + req.Name,
            }
            if err := stream.Send(res); err != nil {
                return err
            }
        }
    }
}
```

تابع **GetOrderBidirectionalStreaming** نیز مانند تابع قبلی است با این تفاوت که نیازی نیست **NamesList** را به آن پاس بدهیم و پاس دادن استریم برای دریافت کردن **request** از کلاینت کافیست. به این صورت که داخل فور یکی یکی نام‌های فرستاده شده توسط کلاینت دریافت می‌شوند و برای هر کدام مقایسه با آیتم‌های سرور اردر و.. انجام می‌شود.

client/main.go



```go
func main() {
    conn, err := grpc.Dial("localhost":port, grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatalf("Connection failed: %v", err)
    }
    defer conn.Close()

    client := pb.NewOrderServiceClient(conn)
    for {
        userInput := 0
        fmt.Printf("Please enter 1 for Server Streaming and 2 for Bidirectional Streaming and 0 to exit: ")
        fmt.Scan(&userInput)

        if userInput == 0 { break }

        var inputNmaes string
        fmt.Printf("please enter Order from this list(note values must be comma seperated and use no space):(banana, apple, orange,
        grape, red apple, kiwi, mango, pear, cherry, green apple) \n")
        fmt.Scan(&inputNmaes)
        if inputNmaes == "" { continue }

        names := strings.Split(inputNmaes, ",")

        orders := &pb.NamesList{
            Names: names,
        }

        if userInput == 1 {
            callGetOrderServerStream(client, orders)
        } else if userInput == 2 {
            callGetOrderBidirectionalStream(client, orders)
        } else {
            break
        }
    }
}
```

در قسمت **main** ابتدا با استفاده از **dial** کانکشن به پورت مربوطه برقرار می‌شود. اگر کانکشن برقرار نشد پیام ناموفق نمایش داده می‌شود و کانکشن بسته می‌شود. در غیر این صورت یک کلاینت جدید با استفاده از تابع **NewOrderServiceClient** ساخته می‌شود وبه همراه لیست نام‌های درخواستی توسط کلاینت به نام **orders** به یکی از دو تابع **callGetOrderServerStream** و یا **callGetOrderBidirectionalStream** پاس داده می‌شود.

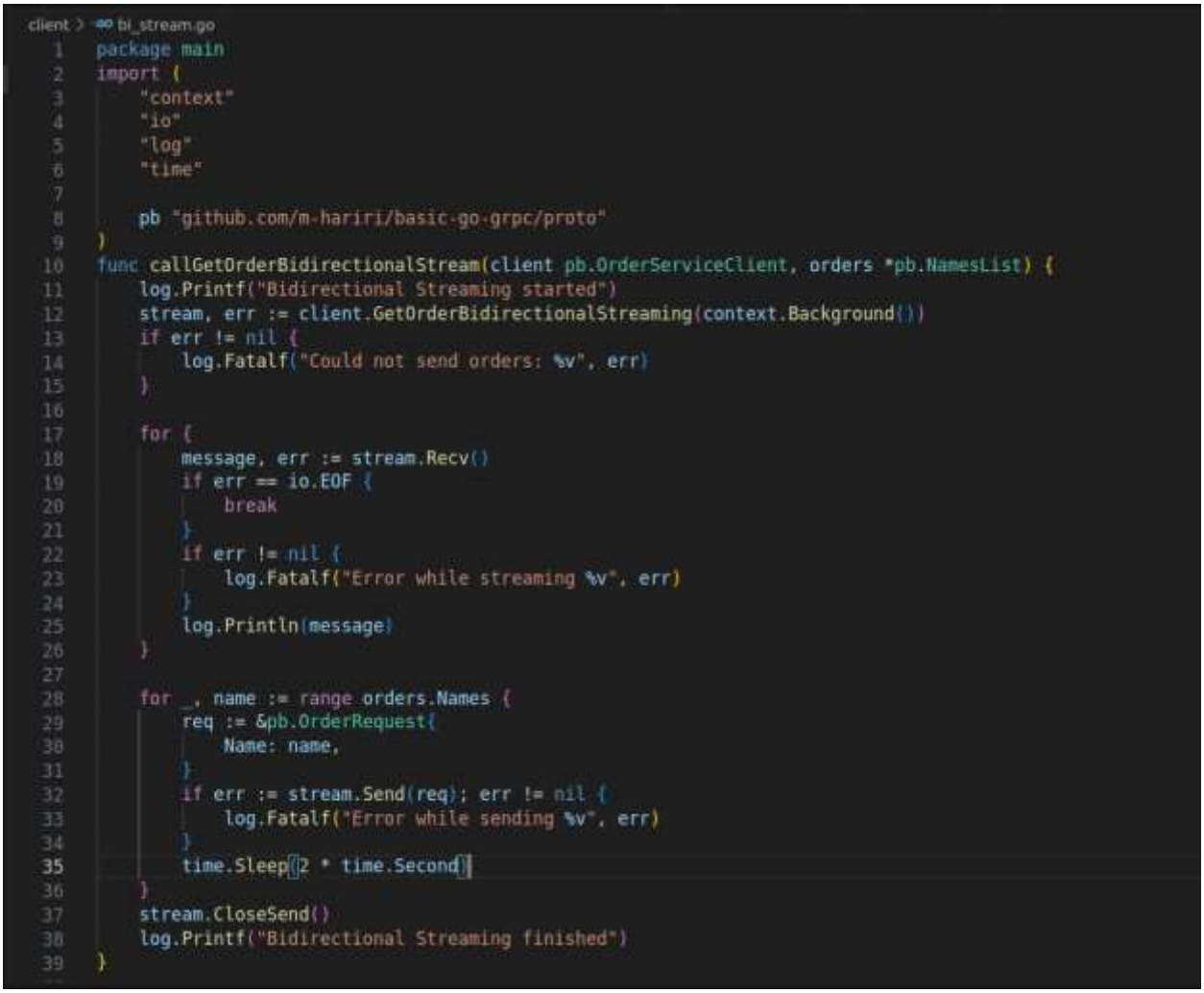
client/server_stream.go

client > server_stream.go

```go
package main

import (
    "context"
    "io"
    "log"

    pb "github.com/m-hariri/basic-go-grpc/proto"
)

func callGetOrderServerStream(client pb.OrderServiceClient, orders *pb.NamesList) {
    log.Printf("Server streaming started")
    stream, err := client.GetOrderServerStreaming(context.Background(), orders)
    if err != nil {
        log.Fatalf("Could not send orders: %v", err)
    }

    for {
        message, err := stream.Recv()
        if err == io.EOF {
            break
        }
        if err != nil {
            log.Fatalf("Error while streaming %v", err)
        }
        log.Println(message)
    }

    log.Printf("Server streaming finished")
}
```

در این تابع ابتدا تابع GetOrderServerStreaming به صورت ریموت توسط کلاینت کال شده تا orders به سرور فرستاده شود. در صورت خطا پیام مربوطه نمایش داده می‌شود سپس داخل فور یکی یکی پیام‌های فرستاده شده از طرف سرور دریافت شده و پرینت میشود.

client/bi_stream.go



در این قسمت ابتدا تابع GetOrderBidirectionalStreaming صدا زده می‌شود سپس یک تابع داخل main در خط 20 تعریف می‌کنیم تا عمل recieve را انجام دهد. در واقع این کار برای concurrent کردن send و recieve است تا بلاکینگ رخ ندهد. متغیر waitc یک چنل است که با استفاده از آن می‌توان تا زمانی که کار تابع خط 20 تمام شود صبر کرد و سپس استریم را بست. داخل این تابع با یک for پیام‌های فرستاده شده توسط سرور دریافت می‌شود و در فور داخل مین نام orderها به صورتOrderRequest به سرور ارسال می‌شود.

نتایج server streaming

کلاینت

سرور

همانطور که مشاهده می‌کنید سرور درخواست‌های کلاینت را در قالب یک لیست دریافت می‌کند چراکه در این روش کلاینت قابلیت streaming ندارد. در ترمینال کلاینت میبینیم که از بین 3 اردر فرستاده شده برای salt ایتمی پیدا نشده اما برای ban و mango ایتم‌های مورد نظر یافت شده‌اند.

نتایج bidirectional streaming

کلاینت

سرور

در این حالت مشاهده می‌کنید که سرور یکی یکی نام اردرها را دریافت می‌کند. این همان قابلیت client streaming است. در کلاینت نیز مانند حالت قبل برای اولین اردر ایتمی پیدا نشده و برای دو اردر بعدی ایتم های مربوطه به همراه timestamp آن و شماره آیتم پرینت شده است.