Name: Vidhi Rathod

Image Processor

Software Design Document

**Introduction:**

Develop an application to allow users to perform any combination of operations on images like rotation, resize, flip, and so on in the order specified by the caller. This document gives in-depth information on the architecture, design, and high-level implementation details.

**Architecture:**

The application is assumed to be hosted on the cloud hence I decided to take up the Client-Server architectural style. In addition, the applications require a running service that can serve the image processing request. The Client-Server architecture style could be one of the best choices for this application.

In addition to Client-Server, I used Pipe-Filter architectural style on the server side. Since the request from the client can have multiple operations (or filters) to be performed on an image. These operations are performed, and the processed image is passed to another filter using the pipe component.

This architecture can allow more than one way to design APIs to establish communication between the Client and Server. However, RPC seems to be a better choice for the nature of the application. The image is not to be stored so no need for resource management. In addition, the operation on the image can be considered as commands to process the image and can be handled by remote procedures present on the server side. The client can provide a sequence of operations on the image as part of the request while the server performs the operations in provided sequence and returns the operated image with the list of thumbnail images if the thumbnail operations were part of the client request.

Diagram

Description automatically generated

Client-Server Architecture Pipe-Filter Architecture

**Design pattern:**

I chose Factory Pattern for the creational design pattern since the object for each image processing operation can be created based on the list of operations in the client request. Each of these objects is then used to create a command object and add it to the pipeline to perform the corresponding operations on the

image in the sequence.

One of the good choices for a behavioral design pattern could be a Command pattern for this application since each operation can be envisioned as a command. An interface for command could be created and each specific command can have its own concrete class to perform specific operations.

Out of the other few design patterns that can be chosen for this application is a Chain of Responsibility in which each image operation after completion can pass the control to another operation. The chain of responsibility may introduce some overhead in the object passing.

**Technologies:**

Implemented Client-Server architecture using Python and established communication over gRPC. Protobuf is used to design the APIs which is a contract between server and client. The main advantage of protobuf is that it is language and platform-independent (*Protocol Buffers*, n.d.-b). This allows a client with Python code or Ruby code to communicate with the same server procedures. Currently, the client just runs the Client python code to perform a list of operations on the image. gRPC is based on TCP and uses HTTP/2 allowing multiplexing and reliability. Hence, the order is also maintained which is also ideal for this application based on the requirement.

Graphical user interface, application

Description automatically generated

(Quick Start, 2023b)

No UI or database is used for this application since this process is based on processing an image and not storing it.

**API Design and Implementation Details:**

Protobuf is the contract between the Client and Server. The API specifications in protobuf give the information for the client to understand how the required list of operations can be performed. The following demonstrates the API definition available on the server which can be used by the client via RPC.

*// Definition of the ImageProcessor service*

*service ImageProcessor {*

*// Processes an input image with a list of requested image processing operations*

*rpc processImage(ProcessImageRequest) returns (ProcessImageResponse) {}*

*}*

*// Request message for the ProcessImage method*

*message ProcessImageRequest {*

*// The input image bytes*

*bytes image = 1;*

*// The list of requested image processing operations*

*repeated ImageProcessorOperation operations = 2;*

*}*

*// Image processing operation message*

*message ImageProcessorOperation {*

*string operationType = 1;*

*// One of the supported image processing operations*

*oneof operation {*

*FlipParams flip = 2;*

*GrayscaleParams grayscale = 3;*

*RotateParams rotate = 4;*

*ResizeParams resize = 5;*

*ThumbnailParams thumbnail = 6;*

*}*

*}*

*// Response message for the ProcessImage method*

*message ProcessImageResponse {*

*// The processed image bytes*

*repeated bytes image = 1;*

*}*

Each of the params structures is defined separately to hold parameters for each operation.

For example:

*// Flip Params*

*message FlipParams {*

*// The direction of the flip*

*FlipDirection direction = 1;*

*}*

The client must provide these operations as a list by using command line arguments. The operations are parsed and added to a list that can be passed when invoking the remote procedure stub just like any other method invocation.

*stub = ImageProcessor\_pb2\_grpc.ImageProcessorStub(channel)*

*request = ImageProcessor\_pb2.ProcessImageRequest(*

*image=image\_data,*

*operations=operations,*

*)*

On the Server side, each operation is seen as a command and a factory method is used to generate an object for each operation. The client sends a list of operations to be performed along with corresponding parameters. The server creates an object based on operation and stored it in the pipeline for it to be executed later sequentially. After each operation is performed the resultant image is used for further operations and the final response image is returned.

Factory Method:

class ImageProcessFactory:

*@staticmethod*

*#Factory method*

*def get\_operation(op,param=None):*

*if op == "gray":*

*return GrayScaleOperation()*

*elif op == "flip":*

*return FlipOperation(param)*

*#Flips the image based*

*class FlipOperation:*

*def \_\_init\_\_(self, isHorizontal):*

*self.isHorizontal\_ = isHorizontal*

*def process(self, img):*

*if self.isHorizontal\_:*

*flip = img.transpose(Image.FLIP\_LEFT\_RIGHT)*

*return flip*

*elif not self.isHorizontal\_:*

*flip = img.transpose(Image.FLIP\_TOP\_BOTTOM)*

*return flip*

Command Class:

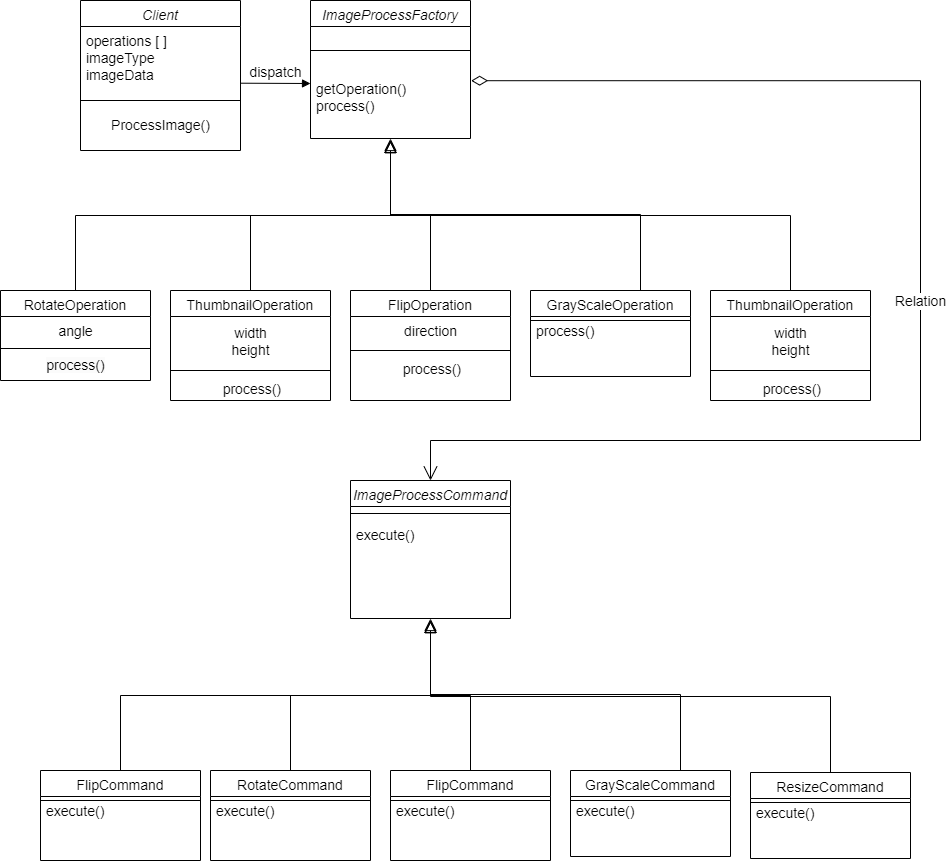
*#Command class to flip image*

*class FlipCommand(ImageProcessCommand):*

*def execute(self, img):*

*flip = self.operation.process(img)*

*return flip*



**Generic API Design Specification:**

Proto file:

1. Create a params structure for this new operation.

*// NewOperation Params*

*message NewOperationParams {*

*// The param type*

*Param1 param = 1;*

*}*

1. Add operation in the following structure.

*// Image processing operation message*

*message ImageProcessorOperation {*

*string operationType = 1;*

*// One of the supported image processing operations*

*oneof operation {*

*FlipParams flip = 2;*

*GrayscaleParams grayscale = 3;*

*RotateParams rotate = 4;*

*ResizeParams resize = 5;*

*ThumbnailParams thumbnail = 6;*

*NewOperationParams newOperation=7;*

*}*

*}*

Client Side:

The client can create a list of operations and send a request to the server. In order to create the right request for RPC, it is required to create an object of operation on the client side and then send it to the server.

*def newOperationImage(param):*

*operation = ImageProcessor\_pb2.ImageProcessorOperation(*

*operationType = "newOperation",*

*flip=ImageProcessor\_pb2.NewOperationParams(param=param)*

*)*

*return operation*

Then invoke the remote procedure as follows:

*response = stub.processImage(request)*

Returns a list of image bytes which includes the final processed image and a list of thumbnail images.

Server Side:

Add the new operation in the Factory method:

class ImageProcessFactory:

*@staticmethod*

*#Factory method*

*def get\_operation(op,param=None):*

*if op == "gray":*

*return GrayScaleOperation()*

*…*

*elif op == "newOperation":*

*return NewOperation(param)*

*class NewOperation:*

*def \_\_init\_\_(self, param):*

*self. param \_ = param*

*def process(self, img):*

….

Create equivalent command class:

*class NewCommand(ImageProcessCommand):*

*def execute(self, img):*

*newOp = self.operation.process(img)*

*return newOp*

On the server side, the process method will invoke each command on the list of operations based on the type of operation. It invokes the execute method of the command object under consideration.

*#Processes image based on Pipe-Filter architecture*

*processedImg = []*

*pipeline = ImageProcessorPipeline(request.operations)*

*pipeline.createPipe()*

*processedImg = pipeline.process(img, processedImg)*

**Little language:**

Server API:

*class ImageProcessorServicerImpl(ImageProcessor\_pb2\_grpc.ImageProcessorServicer):*

*def processImage(self, request, context):*

*"""*

*Process an image using a pipeline of image processing operations.*

*Request:*

*- image: bytes - the image to process, in JPEG format*

*- operations: List[str] - a list of operations to apply to the image*

*Response:*

*- image: bytes - the processed image, in JPEG format*

*Raises:*

*- grpc.RpcError: if an error occurs while processing the image*

*"""*

*try:*

*# ... implementation code ...*

*except Exception as e:*

*# ... error handling code ...*

**Errors and Exceptions:**

Client-Side Exception handling:

Each operation is parsed from the command line and each may or may not have supporting arguments. The syntax for each command is as follows:

*operation\_name/alias:arg1:arg2*

If the syntax is followed client side code throws an error right away, stating the operation and correct syntax. The code terminates demonstrating the help commands for each operation.

For example: In correct operation syntax for Flip operation will throw the following error string *“Incorrect syntax for flip operation*”

Image path is a mandatory argument if not provided client throws the exception “*No image path provided”*. Also, throws *"Path of the file is Invalid"* for the invalid path.

The client also handles invalid value errors or other unknown exceptions.

Server-Side Exception handling:

The following exceptions scenario are handled by the server and pass this information to the client using status codes supported by gRPC.

|  |  |
| --- | --- |
| Status Code | Scenario |
| grpc.StatusCode.INVALID\_ARGUMENT | When a client provides an invalid argument like raw image format |
| grpc.StatusCode.DATA\_LOSS | When a client provides an image that is corrupted or can’t be read |
| grpc.StatusCode.RESOURCE\_EXHAUSTED | When a client tries to use the server resources to completion |
| grpc.StatusCode.UNIMPLEMENTED | When a client tries to use an image operation that is not supported by the server. |
| grpc. StatusCode.UNAVAILABLE | When the server is not available |
| grpc.StatusCode.INTERNAL | When an unknown exception is caught in the server |
| grpc.StatusCode.OK | When the operations were successful. |

G. (n.d.).

**Scope and Assumption:**

The gRPC server adds a port to listen on without encryption and will accept connections from any IPv6 address at port number 50051. In this implementation, the server receives a request with an input image and a sequence of requested image processing operations. It then applies the sequence of image processing operations in the pipeline to the input image, with each filter function transforming the image as necessary. Finally, the output image is returned to the client as a response to the gRPC request. The response could be a list of images in case the list of operations also has thumbnail requests. If multiple thumbnail requests are placed in the operations list, then the corresponding list of thumbnail responses will be provided in the requested order. By default, the thumbnail size is 300x300 if not provided. Rotation other than a multiple of 90 degrees leads to shrinkage of the image while preserving the image and not cropping any parts of the image. All numerical arguments are considered whole numbers or integers and do not allow any float values for the sake of simplicity.

The scope of this implementation is limited to only one image and the application works on a local network currently which can be extended in the future. The supported image formats are png and jpg. Currently, the size of the image is not restricted, however if the resources of server are exhausted it terminates with relevant status code. The code is supported and well-tested on Linux and Windows. The code is not tested cross platform that is if server is hosted on linux or client is making request from windows or vice versa.

**Build and Deployment:**

The code is tested on windows 11 and ubuntu 22.04 separately. It is assumed that the system used for client or server has Python and pip installed. There are two sets of code files one for the server and the other for the client.

To successfully compile with grpc install grpcio and grpc\_tools:

*python -m pip install grpcio*

*python -m pip install grpcio-tools*

*python -m pip install Pillow*

To autogenerate the dependency for gRPC communication based on proto file:

*python -m grpc\_tools.protoc -I./protos --python\_out=./protos --grpc\_python\_out=./protos ./protos/ImageProcessor.proto*

The above steps should be performed on the client and server sides, I provided a setup python for both ends. If the ClientSetup.py and ServerSetup.py are executed it will download, import, and install all the necessary files. ClientFiles folder has all client related files and ServerFiles has all the server related files.

The server code should be executed first. It’s a blocking service that stays active until interrupted and waits for client requests. When the client request is received it serves the request and sends a response. The client is executed after the server is started. So, these two are required. To execute server and client use the following.

Server: python .\Server.py

Client:

python .\Client.py .\input.jpg gray rotateleft rotate:-45 thumbnail:300:300 flip:h resize:200:200

OR

python .\Client.py .\input.jpg g rl r:-45 t:300:300 f:h rz:200:200

All the commands are space-separated and the arguments for each operation are colon-separated. For help use the following command:

python .\Client.py help OR python .\Client.py h

**Non-Functional Requirements:**

Performance – Performance in this project is determined by each operation processing for the image. Strategy to keep the performance as good as possible for each operation. The client may provide multiple operations in one request and the client should wait for an unreasonably long time.

Simplicity – The objective of designing a system is to make it as simple as possible for both the developer and the user. For users, the experience can be improvised using frontend frameworks and for developers adding new features could be simplified by using good interfaces and reducing the number of unnecessary components.

Extensibility – The scope in the above section indicated how the existing project could be extended. If the developer wants to use a new RPC then the new API in the client code can be easily added.

Testability – Each API and RPC call should be well tested and a good testing framework if developed can be extended by new developers to test new APIs they develop.

References:

*Protocol Buffers*. (n.d.-b). Protocol Buffers Documentation. <https://protobuf.dev/>

*Quick start*. (2023b, February 16). gRPC. <https://grpc.io/docs/languages/python/quickstart/>

G. (n.d.). *grpc/statuscodes.md at master · grpc/grpc*. GitHub. <https://github.com/grpc/grpc/blob/master/doc/statuscodes.md>