gRPC Implementation: Calculator Service

A Practical Demonstration (basic online calculator)

Engr. Daniel Moune

ICT University

August 12, 2025

Overview

What is gRPC?

- High-performance RPC framework by Google
- Uses Protocol Buffers (protobuf) as interface definition language
- Features:
  - Language-neutral
  - Bi-directional streaming
  - Pluggable authentication
  - Load balancing
- HTTP/2 based transport

Project Structure

- `calculator.proto` - Service definition
- `calculator_pb2.py` - Generated protobuf classes
- `calculator_pb2_grpc.py` - Generated gRPC classes
- `cloud.py` - Server implementation
- `client.py` - Client implementation

Protocol Buffers Definition

```
  rpc Sub (AddRequest) returns (AddResponse);
  rpc Mul (AddRequest) returns (AddResponse);
  rpc Div (AddRequest) returns (AddResponse);
  rpc Mod (AddRequest) returns (AddResponse);
}

message AddRequest {
  int32 num1 = 1;
  int32 num2 = 2;
}

message AddResponse {
  int32 result = 1;
}
```

- Defines service `Calculator` with 5 RPC methods
- All methods use `AddRequest` and `AddResponse` messages
- Simple message structure with two integers and a result

Generated Code

- `python -m grpc_tools.protoc` generates:
    - `calculator_pb2.py` - Message classes
    - `calculator_pb2_grpc.py` - Server and client classes
- Provides:
    - Serialization/deserialization
    - Client stub
    - Server interface

Server Implementation (1)

```python
import grpc
from concurrent import futures
import calculator_pb2
import calculator_pb2_grpc

class CalculatorSkeleton(calculator_pb2_grpc.
    CalculatorServicer):
    def Add(self, request, context):
        result = request.num1 + request.num2
        return calculator_pb2.AddResponse(result=result)
    def Sub(self, request, context):
```

- Extends `CalculatorServicer` generated class
- Implements all 5 RPC methods
- Each method performs the operation and returns a response

Server Implementation (2)

```
        return calculator_pb2.AddResponse(result=result)
    def Mul(self, request, context):
        result = request.num1 * request.num2
        return calculator_pb2.AddResponse(result=result)
    def Div(self, request, context):
        result = request.num1 // request.num2
        return calculator_pb2.AddResponse(result=result)
```

- Creates gRPC server with thread pool
- Adds servicer to the server
- Binds to port 50051
- Starts the server

Client Implementation (1)

```python
#import sys
import grpc
from concurrent import futures
import calculator_pb2
import calculator_pb2_grpc


class job :
    def __init__(self, operator, operand1, operand2) -> None
    :
        self.__operator = operator
        self.__operand1 = int(operand1)
        self.__operand2 = int(operand2)
    def operator(self) -> str:
        return self.__operator
    def operand1(self) -> int:
```

- job class encapsulates operation details
- Provides access to operator and operands
- Type hints for better code clarity

Client Implementation (2)

```python
    def operand2(self) -> int:
        return self.__operand2

def run(operator, num1, num2):
    with grpc.insecure_channel('localhost:50051') as channel:
        stub = calculator_pb2_grpc.CalculatorStub(channel)
        if (operator == "add"):
            response = stub.Add(calculator_pb2.AddRequest(num1=num1, num2=num2))
        elif (operator == "sub"):
            response = stub.Sub(calculator_pb2.AddRequest(num1=num1, num2=num2))
        elif (operator == "mul"):
            response = stub.Mul(calculator_pb2.AddRequest(num1=num1, num2=num2))
        elif (operator == "div"):
            response = stub.Div(calculator_pb2.AddRequest(num1=num1, num2=num2))
        elif (operator == "mod"):
            response = stub.Mod(calculator_pb2.AddRequest(
```

Client Implementation (3)

```
            print("Invalid operator")
            exit()
    print(f"Result: {response.result}")

# if __name__ == '__main__':
#     # Get user Input
#     operand = sys.argv[1]
#     num1 = sys.argv[2]
#     num2 = sys.argv[3]
#     # print(f"operand = {operand},  num1 = {num1}, num2 =
    {num2}")
```

- Uses thread pool for concurrent requests
- Predefined list of jobs to execute
- Submits each job to the thread pool
- Demonstrates concurrent RPC calls

Key Features

- **Type Safety**: Protobuf ensures type-safe communication
- **Concurrency**: Thread pool handles multiple requests
- **Simplicity**: Clean separation between client and server
- **Extensibility**: Easy to add new operations
- **Performance**: HTTP/2 and binary protocol

Demonstration

- Start the server: `python cloud.py`
- Run the client: `python client.py`
- Expected output:
  - Result: 660 (add)
  - Result: 606 (sub)
  - Result: 23 (div)
  - Result: 12 (mod)

Conclusion

- gRPC provides efficient RPC mechanism
- Protocol Buffers enable language-neutral contracts
- Python implementation is straightforward
- Suitable for microservices and distributed systems
- Next steps:
  - Add error handling
  - Implement streaming
  - Add authentication

References

- gRPC official documentation: `https://grpc.io/`
- Protocol Buffers guide: `https://developers.google.com/protocol-buffers`
- Python gRPC examples: `https://github.com/grpc/grpc/tree/master/examples/python`

Thank You!