1. GRPC

In distributed systems and modern application development, Remote Procedure Call (RPC) is essential because it allows programs running on different machines to communicate and invoke each other's functions as if they were local. Instead of manually managing sockets, message serialization, and network protocols, RPC abstracts these complexities – developers simply call a function on a remote server, and the system handles the communication behind the scenes.

In the evolution of distributed systems, gRPC [1], developed by Google, represents a modern re-implementation of the Remote Procedure Call paradigm. gRPC delivers high performance, scalability, and reliability in cross-platform communication. The architecture of gRPC is depicted in Figure 1, where the Service, Server, and Client components can be implemented in any supported programming language.

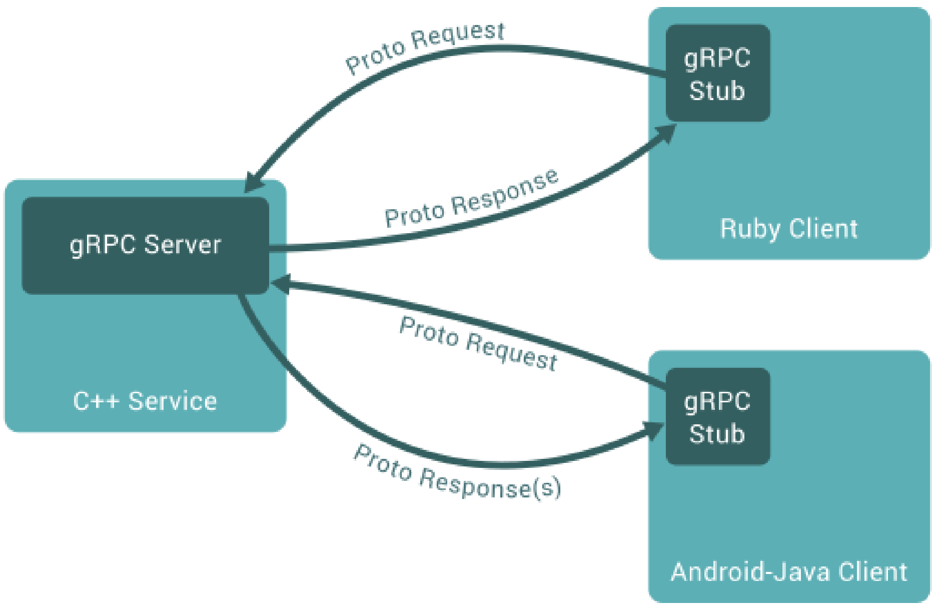

Figure 1: Architecture of gRPC.

To get started with gRPC, the first step is to install the necessary Python packages. gRPC and its supporting tools can be directly installed using the Python package manager **pip**. The essential packages include **grpcio, grpcio-tools, and protobuf**. However, we should install the following versions for compatibility with later exercises.

```
# virutal environment should be activated first
pip install "protobuf==3.20.3" "grpcio==1.48.2" "grpcio-tools==1.48.2"
```

gRPC provides a convenient mechanism to automatically generate the necessary client and server code from a service definition. Developers simply describe the service interfaces and message structures in a .proto file. The gRPC tools then use this file to

[1] gRPC documents: https://grpc.io/docs/what-is-grpc/introduction/.

automatically generate message classes, service stubs, and handlers for remote procedure calls. Once the code is generated, developers only need to implement the service logic on the server side and call the stub methods on the client side.

A proto file consists of two main components: services and messages. The service definitions describe the available remote procedures or functions that can be invoked by a client. The message definitions, on the other hand, represent the data structures exchanged between clients and servers. Message is defined quite similar to a normal Python class, containing a set of fields with explicit types, names, and unique numeric tags. These tag numbers are required and used internally by Protocol Buffers to identify fields in the serialized binary format.

```
service MyService {
  rpc SayHello (MyRequest) returns (MyResponse);
}

message MyRequest {
  string name = 1;
  int32 age = 2;
  bool is_student = 3;
}
```

The following proto file illustrates a simple example of a gRPC service that implements two types of remote procedure calls: a unary RPC, where the client sends one request and receives a single response, and a server-streaming RPC, where the client sends one request but receives multiple responses piece-by-piece from the server. For more details on defining a proto file, students are encouraged to refer to this link.

```
syntax = "proto3";  // Always define syntax version (proto3 is current standard)
package tutorial;   // Creates a namespace - useful if your app has many .proto files

service Greeter {
  // Unary RPC: one request, one response
  rpc SayHello (HelloRequest) returns (HelloReply) {}

  // Server-streaming: one request, many responses
  rpc LotsOfReplies (HelloRequest) returns (stream HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

To automatically generate the Python source files required for gRPC communication, run the below command. This command uses the grpcio-tools package to invoke the Protocol Buffers compiler and generate two output files:

- <FILE_NAME>_pb2.py contains the Python classes that represent the message types defined in the .proto file. These handle data serialization and deserialization.
- <FILE_NAME>_pb2_grpc.py contains the gRPC-specific classes such as service stubs (for clients) and servicers (for servers), which are related to the remote methods.

```
python3 -m grpc_tools.protoc -I . --python_out=. --grpc_python_out=. <FILE_NAME>.proto
```

The dummy code below shows how to define a simple gRPC server and the corresponding client that invokes a remote method defined in the proto file, respectively. The server implements the logic for the RPC method declared in the Protocol Buffers definition, while the client uses the generated stub to call this method remotely.

```python
# in server.py file
import time
import grpc

import hello_pb2
import hello_pb2_grpc

from concurrent import futures

class GreeterServicer(hello_pb2_grpc.GreeterServicer):
    def SayHello(self, request, context):
        # Unary: return one response
        msg = f"Hello, {request.name}!"
        return hello_pb2.HelloReply(message=msg)

    def LotsOfReplies(self, request, context):
        # Server streaming: yield multiple messages
        name = request.name or "friend"
        for i in range(5):
            yield hello_pb2.HelloReply(message=f"[{i}] Hi {name} from stream")
            time.sleep(0.3)  # for demo pacing

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10)) # set the number of threads server are using
    hello_pb2_grpc.add_GreeterServicer_to_server(GreeterServicer(), server)
    server.add_insecure_port("[::]:50051")   # listen on all interfaces in port 50051
    server.start()
    print("gRPC server running on :50051")

    try:
        server.wait_for_termination()
    except KeyboardInterrupt:
        print("Shutting down...")

if __name__ == "__main__":
    serve()
```

```python
# in client.py file
import grpc
import hello_pb2
import hello_pb2_grpc

def unary_example(stub):
    try:
        resp = stub.SayHello(hello_pb2.HelloRequest(name="Binh"), timeout=2.0)
        print("Unary response:", resp.message)
    except grpc.RpcError as e:
        print("Unary failed:", e.code(), e.details())

def server_streaming_example(stub):
    try:
        stream = stub.LotsOfReplies(hello_pb2.HelloRequest(name="Binh"), timeout=3.0)
        for msg in stream:
            print("Stream item:", msg.message)
    except grpc.RpcError as e:
        print("Stream failed:", e.code(), e.details())

def main():
    with grpc.insecure_channel("localhost:50051") as channel:
        stub = hello_pb2_grpc.GreeterStub(channel)
        unary_example(stub)
        server_streaming_example(stub)

if __name__ == "__main__":
    main()
```

Finally, open two terminal tabs, and run the server and client programs as you would any regular Python applications. If the client runs on a different machine, simply replace “localhost” with the IP address of the machine hosting the gRPC server.

Note: The server machine must ensure that the corresponding port is open and accessible to allow clients to establish connections.

2. LINUX MONITORING

Linux views “everything is a file”, which means that various system components, such as hardware devices, kernel interfaces, inter-process communication mechanisms, and even running processes, are exposed through the file system, particularly under virtual file systems like /proc and /sys. As a result, it is theoretically and practically possible to monitor nearly all aspects of the system, including hardware, kernel subsystems, user-space applications, and containers, by accessing and interpreting the data exported by the kernel to these file system interfaces.

The following command is used to collect the percentage of memory usage.

```
free | awk '/^Mem:/ { printf("%.2f\n", $3/$2 * 100) }'
```

The following command is used to collect the average CPU usage.

```
top -bn1 | grep "Cpu(s)" | awk '{print 100 - $8}'
```

The following two commands are used to collect read and write io (KB/s), respectively.

```
# read io
iostat -d -k 1 2 | awk 'NR>7 {read+=$3} END {print read}'
# write io
iostat -d -k 1 2 | awk 'NR>7 {write+=$4} END {print write}'
```

The following two commands are used to collect network in and out bandwidth (KB/s) through a specific network interface, respectively.

```
# sudo apt install ifstat if need
# network in through <INF_NAME> interface
ifstat -i <INF_NAME> 1 1 | awk 'NR>2 {print $1}'
# network out through <INF_NAME> interface
ifstat -i <INF_NAME> 1 1 | awk 'NR>2 {print $2}'
```

The “subprocess” module in Python is provided as a built-in and efficient interface for interacting with the system shell and retrieving command result from the standard output (stdout), enabling seamless integration of system-level commands into Python applications. Similar mechanisms for executing and managing shell commands exist in other programming languages as well, such as Java “ProcessBuilder”, Go’s “os/exec” package, or Node.js “child_process” module.

```
import subprocess

command = "ps aux | grep python"
result = subprocess.run(command, shell=True, capture_output=True, text=True)
print(result.stdout)
```

3. EXERCISES

Students are required to develop a simple monitoring tool that periodically sends data to a server and receives commands from the server for execution. Specifically, students must complete the following tasks:

- Implement multiple client applications (corresponding to the number of group members) that periodically send monitoring data to the server.
- Implement a server application that collects monitoring data from clients and sends control commands to clients for execution.

Note: The monitoring data should follow the structure `(time, hostname, metric, value)`.
Note: To enable event-driven command delivery from the server, students are asked to research and apply the concept of gRPC bidirectional streaming as defined below.

```
// bidirectional streaming protoype
service MyService {
    rpc CommandStream (stream CommandResponse) returns (stream CommandRequest);
}
```

**** Submission:** This course does not require individual submissions. Students are encouraged to work collaboratively in groups to complete the assigned tasks. However, each group will be required to present and demonstrate all aspects of their work during the final class session. The final presentation will be purely an engineering demonstration and no written report is required.