11 DECEMBER 2022

Introduction to system/backend programming

RPC/Network programming

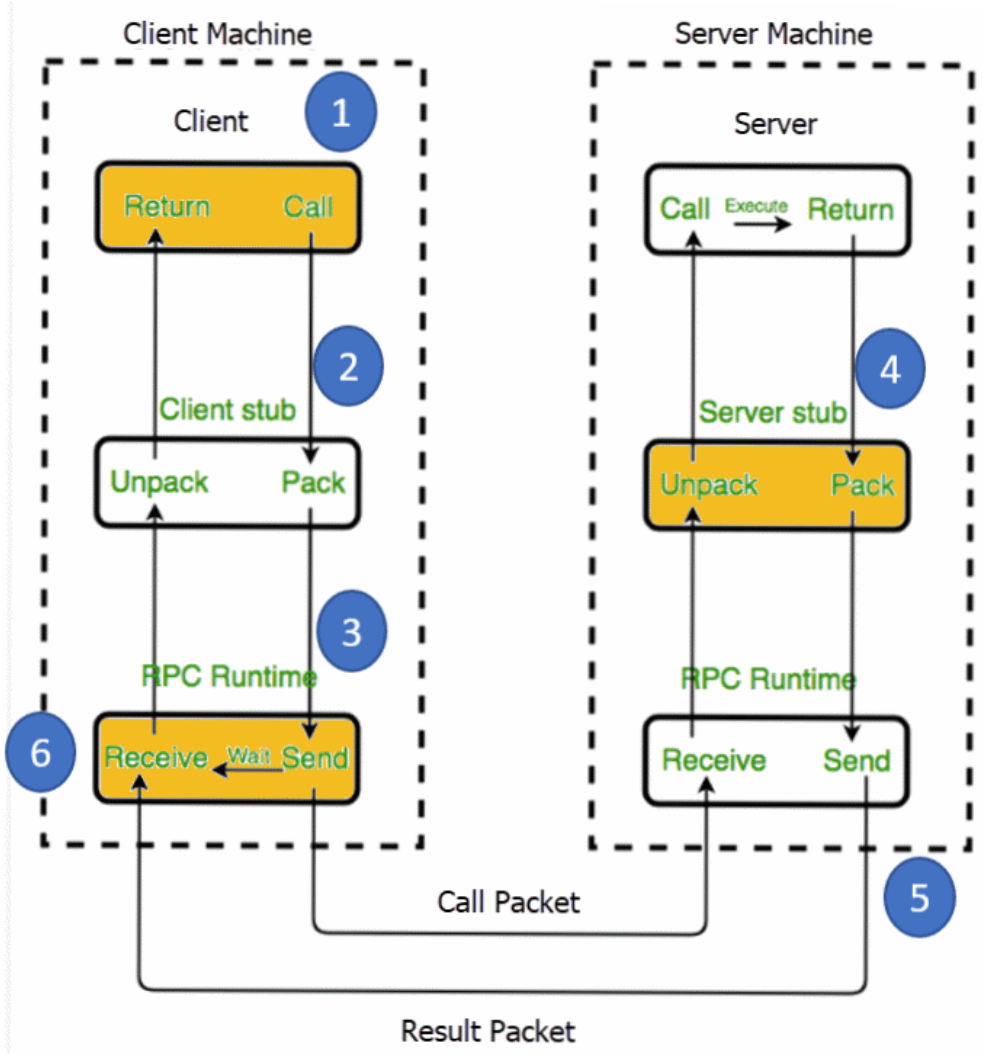
RPC

- The world doesn’t start/end with REST

RPC

- The world doesn’t start/end with REST
- Another solution is **R**emote **P**rocedure **C**all (RPC)
  - The idea is simple – invoke remote functions as if they were local
  - Communication is abstracted from the client, ideally you wouldn’t distinguish a local method vs remote method call
  - This is achieved via request-response message passing aka exchanging network packets
    - They carry arguments (request) and result (response)
  - RPC calls can be either synchronous or asynchronous
  - Potentially more efficient than REST (due to binary encoding)

RPC (2)



RPC (3)

- RPC as an idea is very old – first implementation around 1982, theoretical proposals 1970
- There are multiple RPC protocol
  - XML-RPC – uses XML to package request/response and utilizes HTTP as transport – akin to what we saw with REST. (http://xmlrpc.com/spec.md)
  - JSON-RPC – uses JSON to represent request/response data, no mandate on transport
  - gRPC
  - There are many more similar frameworks/RPC protocols

XML-RPC example

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.8.10
Date: Thu, 10 Mar 2022 13:19:44 GMT
Content-type: text/xml
Content-length: 129

<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><boolean>0</boolean></value>
</param>
</params>
</methodResponse>
```

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))

#python3 xml-rpc-client.py
3 is even: False
100 is even: True
```

```
POST / HTTP/1.1
Host: localhost:8000
Accept-Encoding: gzip
Content-Type: text/xml
User-Agent: Python-xmlrpc/3.8
Content-Length: 146

<?xml version='1.0'?>
<methodCall>
<methodName>is_even</methodName>
<params>
<param>
<value><int>3</int></value>
</param>
</params>
</methodCall>
```

XML-RPC example

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.6 Python/3.8.10
Date: Thu, 10 Mar 2022 13:38:55 GMT
Content-type: text/xml
Content-length: 315

<?xml version='1.0'?>
<methodResponse>
<fault>
<value><struct>
<member>
<name>faultCode</name>
<value><int>1</int></value>
</member>
<member>
<name>faultString</name>
<value><string>&lt;class 'Exception'&gt;:method "is_even2" is not
supported</string></value>
</member>
</struct></value>
```

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("100 is even: %s" % str(proxy.is_even2(100)))

#python3 xml-rpc-client.py
xmlrpc.client.Fault: <Fault 1: '<class \'Exception\'>:method
"is_even2" is not supported'>
```

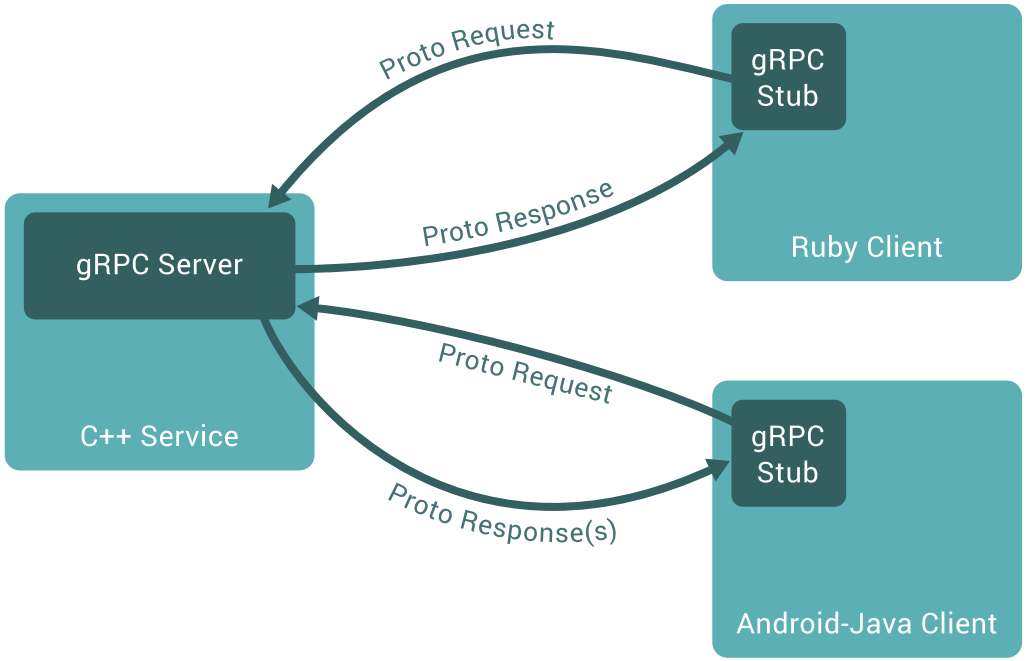
```
POST / HTTP/1.1
Host: localhost:8000
Accept-Encoding: gzip
Content-Type: text/xml
User-Agent: Python-xmlrpc/3.8
Content-Length: 149

<?xml version='1.0'?>
<methodCall>
<methodName>is_even2</methodName>
<params>
<param>
<value><int>100</int></value>
</param>
</params>
</methodCall>
</fault>
</methodResponse>
```

gRPC

- RPC framework from Google (hence the g)

gRPC

- RPC framework from Google (hence the g)
  - Relies on protobuf (also a google technology to do marshalling/unmarshalling of types)
    “Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.”
- Augments protobuf to also generate “service stubs”.
  - Currently C++/Java/Kotlin/Python/Go/Ruby/Objective-C/C#/Dart/PHP (protoc3) are supported



GRPC (2)

```
syntax = "proto3";

service Calculator {
	rpc isEven(Question) returns (Reply) {}
}

message Question {
	optional int32 value = 1;
}

message Reply {
	optional bool value = 1;
}
```

```
#python3 -m grpc_tools.protoc -I. --python_out=. --grpc_python_out=. Service.proto
```

gRPC (3) - Server

```
import Service_pb2
import Service_pb2_grpc
import grpc
from concurrent import futures

class CalculatorServer(Service_pb2_grpc.CalculatorServicer):
    def isEven(self, request, context):
        if request.value % 2 == 0:
            return Service_pb2.Reply(value=True)
        else:
            return Service_pb2.Reply(value=False)

server = grpc.server(futures.ThreadPoolExecutor(max_workers=2))
Service_pb2_grpc.add_CalculatorServicer_to_server(CalculatorServer(), server)
server.add_insecure_port('[::]:50051')
server.start()
server.wait_for_termination()
```

gRPC (4) - Client

```
import grpc
import Service_pb2_grpc
import Service_pb2

channel = grpc.insecure_channel('localhost:50051')
stub = Service_pb2_grpc.CalculatorStub(channel)
response = stub.isEven(Service_pb2.Question(value=5))
print("isEvent(5) :" + str(response.value))
response = stub.isEven(Service_pb2.Question(value=10))
print("isEvent(10) :" + str(response.value))

#python3 client.py
isEvent(5) :False
isEvent(10) :True
```

Network programming

- So far all topics revolved around network communication
- But it was all very abstracted. However...

Network programming

- So far all topics revolved around network communication
- But it was all very abstracted. However…
- .. the world is a messy place and you'll have to get your hands dirty at some point

Sockets

- Socket is the name of a software abstraction for network communication

Sockets

- Socket is the name of a software abstraction for network communication
- All network-aware programs work with sockets (however well those might be hidden behind further abstractions)

Sockets

- Socket is the name of a software abstraction for network communication
- All network-aware programs work with sockets (however well those might be hidden behind further abstractions)
- Lifecycle of a socket
  1) Create a socket – you get a descriptor, some way to refer to this socket
  2) Use this socket to either bind()/listen()/accept() for incoming connection or instantiate an outgoing (connect()) connection
  3) read/write data

Socket server

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Socket client

```
import socket

HOST = "127.0.0.1"
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

Conclusion

- We saw 2 different ways of service communication
  - RPC and REST

Conclusion

- We saw 2 different ways of service communication
  - RPC and REST
- Which method you choose will depend on your requirements
  - Time to market
  - Knowledge of given technology
  - Specific technical requirements (i.e latency)

Conclusion

- We saw 2 different ways of service communication
  - RPC and REST
- Which method you choose will depend on your requirements
  - Time to market
  - Knowledge of given technology
  - Specific technical requirements (i.e latency)
- Always follow the KISS principle (Keep it simple, stupid)

Thank You

For more information, contact SUSE at:

+1 800 796 3700 (U.S./Canada)

Maxfeldstrasse 5

90409 Nuremberg

www.suse.com