DISTRIBUTED AND CLOUD COMPUTING

LAB 5: RPC - GRPC & JAVA RMI

(Module: RPC & RESTFUL API)

Remote Procedure Call (RPC)

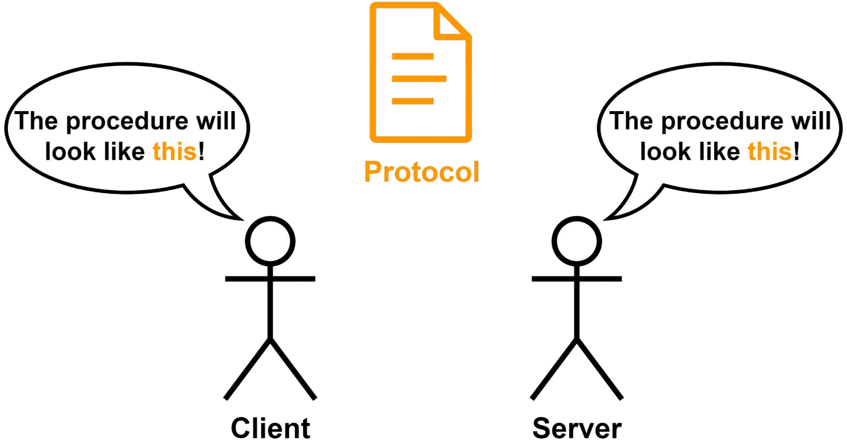
- “In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure to execute in a different address space (commonly on another computer on a shared computer network), which is written as if it were a normal (local) procedure call, without the programmer explicitly writing the details for the remote interaction.”
    a. **serve remotely**
    b. **invoke as if it were local**
    c. **abstract remote interaction details**
- An intuitive example: Holobox
    a. the person to talk to is not here
    b. talk to the person’s hologram
    c. do not care how the hologram is set up

- What are the **design concerns** for achieving the aforementioned key points?

https://www.holoconnects.com/wp-content/uploads/2023/10/holoconnects-branche-education-1-1920x1280.jpg

RPC - Design Concerns

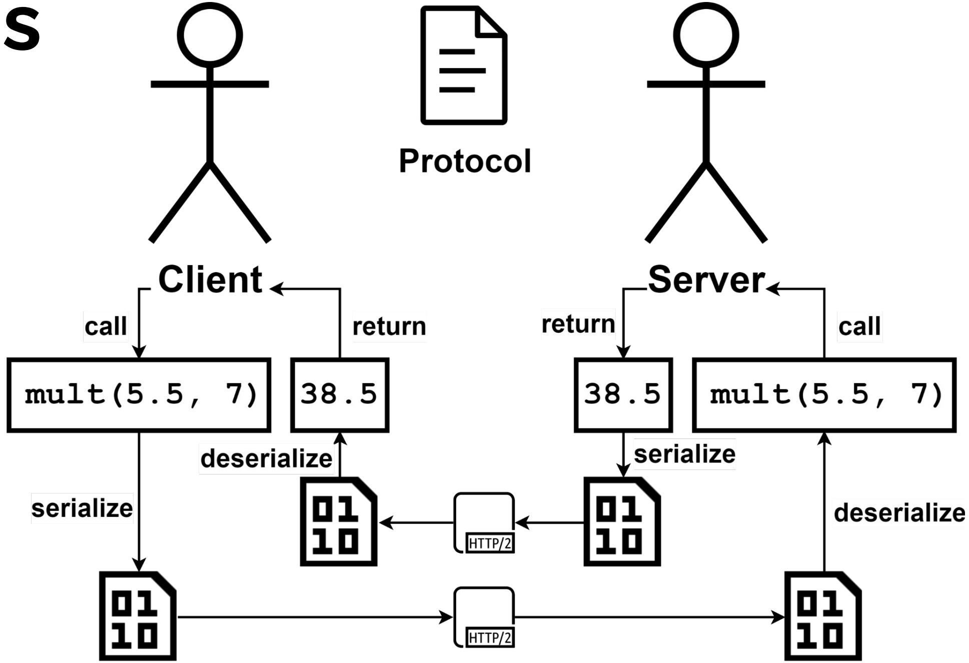
- **Protocol**:
  a. "If the procedure is on a remote server, how do I know its signature to properly execute it?"
  b. Also referred to as Contract, Interface, or Interface Definition Language (IDL).
  c. Procedure signature:
     - Name
     - Input parameters
     - Output format


The procedure will look like this!
Protocol
The procedure will look like this!
Client
Server

- Specifying the protocol in advance:
  a. allows clients and servers to perform implementations independently;
  b. supports advanced client/server-side code generation for RPC.

RPC - Design Concerns

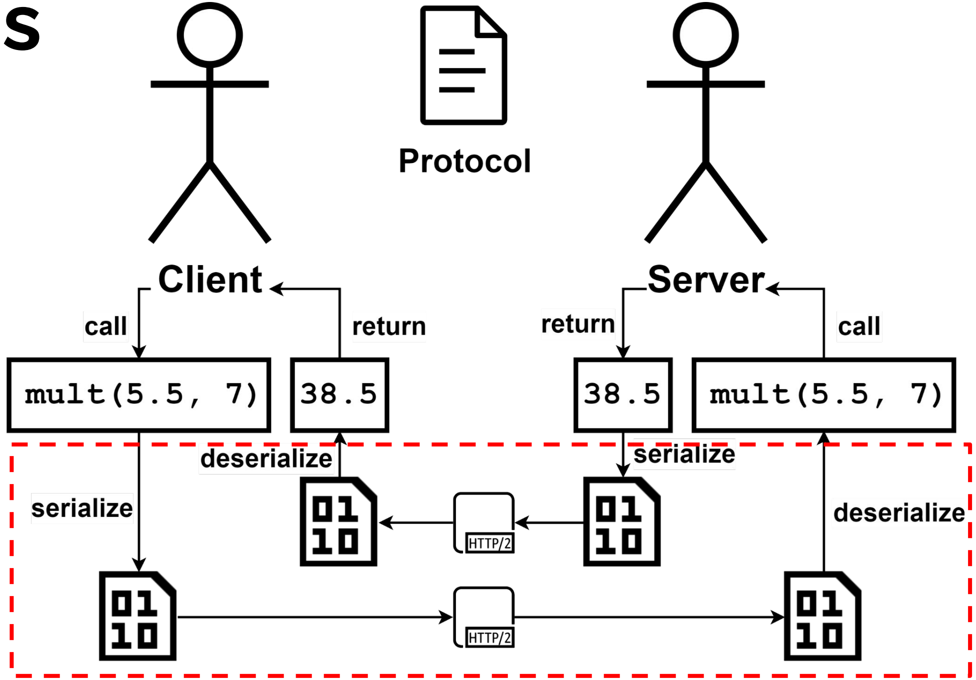
- **Protocol**
- **Communication**:
  a. "How do I connect to the remote server and invoke the remote procedure on it?"
  b. Also referred to as Transmission.
  c. **Serialization**: "how do I pack my request parameters into a transmittable format?"
  d. **Network Protocol**: "do I use TCP or HTTP to manage the internet connection?"



- To reduce the network communication overhead, we might:
  a. shrink the transmission data size;
  b. select an appropriate network protocol to manage multiple requests (potentially for multiple RPC procedures) efficiently.

RPC - Design Concerns

- Protocol
- **Communication**:
  a. "How do I connect to the remote server and invoke the remote procedure on it?"
  b. Also referred to as Transmission.
  c. **Serialization**: "how do I pack my request parameters into a transmittable format?"
  d. **Network Protocol**: "do I use TCP or HTTP to manage the internet connection?"

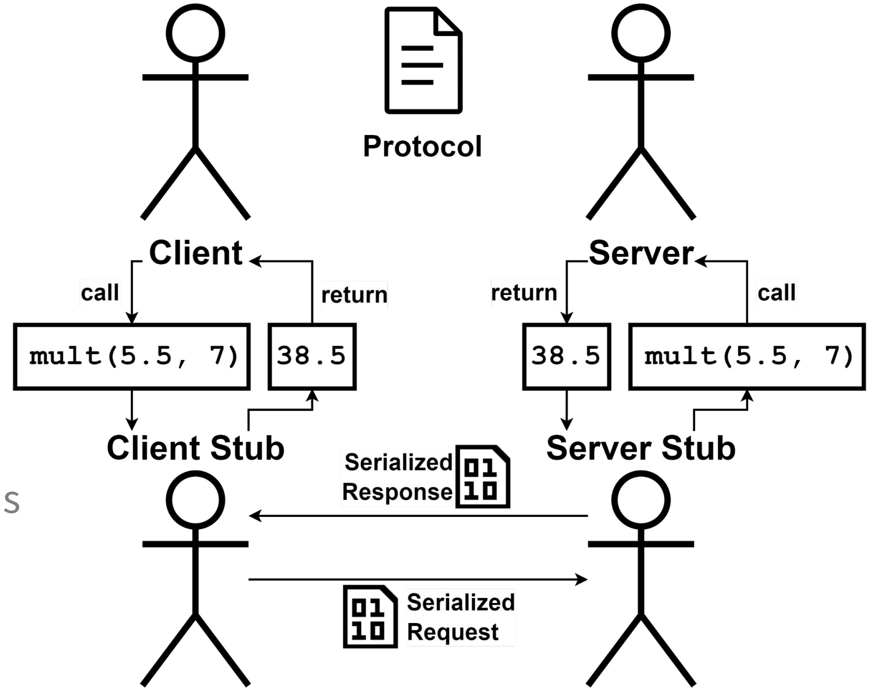


- Communication handling introduces too much complexity to both the client & the server!

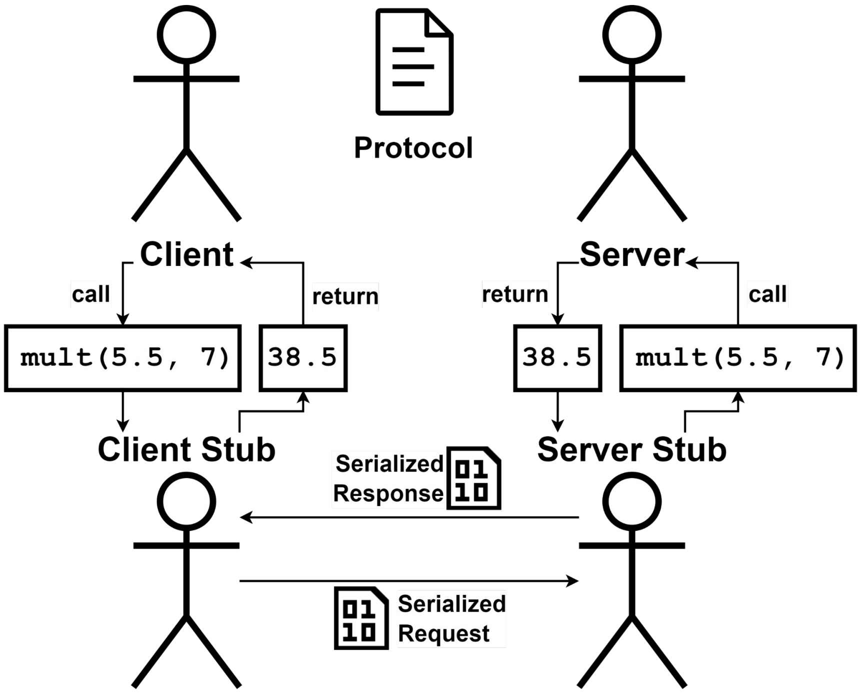
RPC - Design Concerns

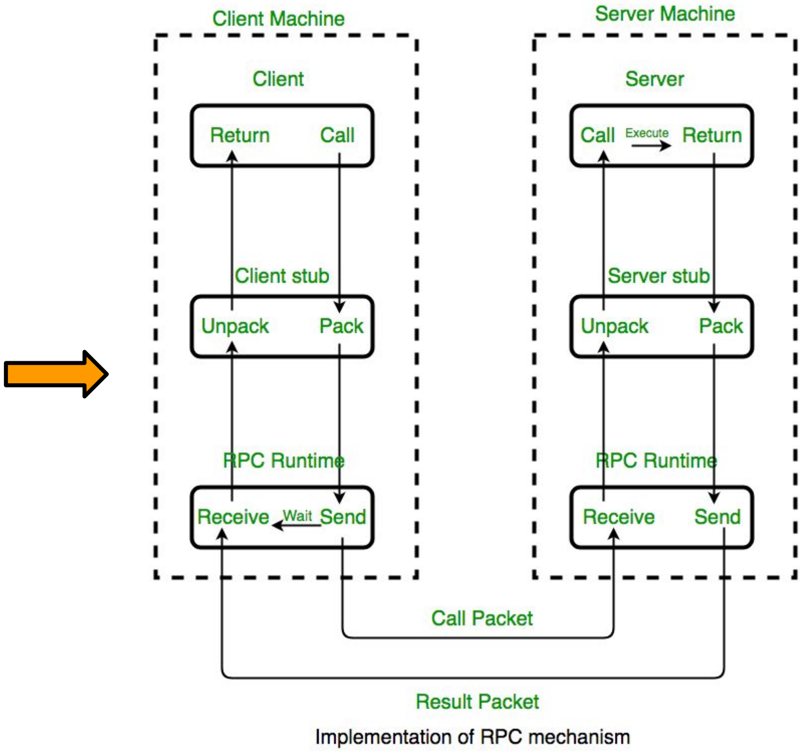
- Protocol
- Communication
- **Proxy**:
  a. **Client Stub**:
    - "Can I just focus on requesting the correct procedure with the correct parameters, while letting someone help me handle the rest?"
    - Also referred to as Client-side Proxy.

  b. **Server Stub**:
    - "Can the server developers just focus on the procedure implementation, while letting someone help handle the rest?"
    - Also referred to as Server-side Proxy, Skeleton.



RPC - Common Structure



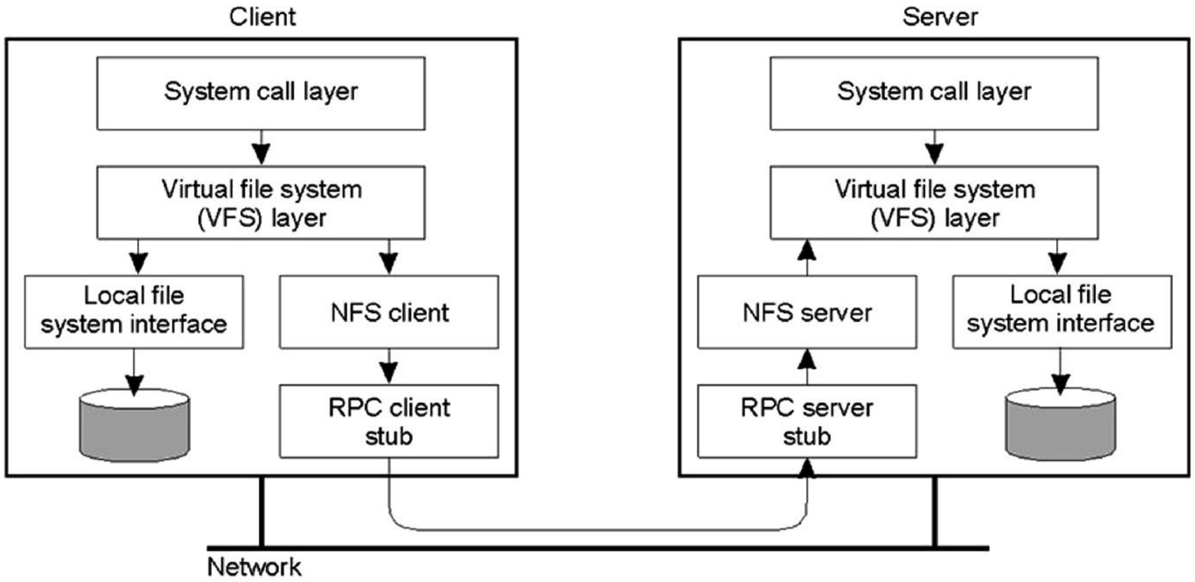


Implementation of RPC mechanism

https://media.geeksforgeeks.org/wp-content/uploads/operating-system-remote-call-procedure-working.png

RPC - Application Scenarios

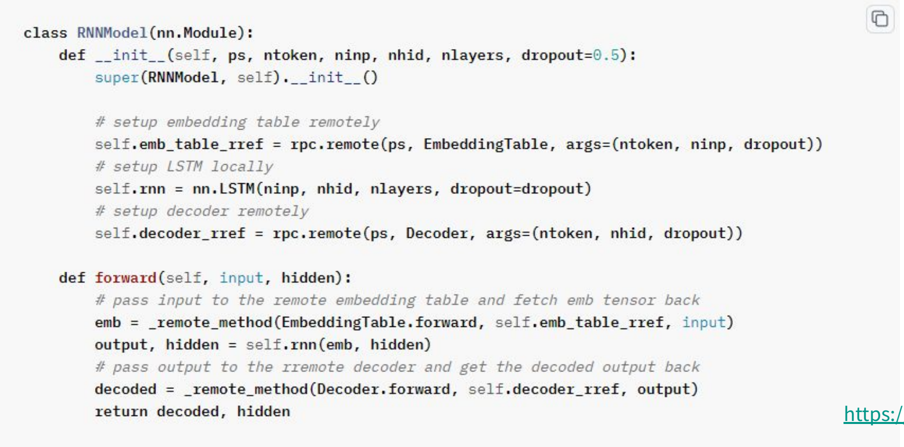
- **Network File System (NFS)**:
  a. Makes remote file operations (e.g., file copying) feel local.
  b. `cp /mnt/nfs/data.txt ~/Documents/new_data.txt` as usual, RPC will handle the communication details behind the scene.


Client
Server
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

https://link.springer.com/chapter/10.1007/978-3-030-68291-0_5/figures/1

RPC - Application Scenarios

- **Network File System (NFS)**:
  a. Makes remote file operations (e.g., file copying) feel local.
  b. `cp /mnt/nfs/data.txt ~/Documents/new_data.txt` as usual, RPC will handle the communication details behind the scene.
- **PyTorch Distributed RPC**:
  a. Manages multi-machine model training/inference.
  b. Supports referencing remote objects without copying the real data around.
- **Backend-to-backend Communication in Microservices**
- ...



```python
class RNNModel(nn.Module):
    def __init__(self, ps, ntoken, ninp, nhid, nlayers, dropout=0.5):
        super(RNNModel, self).__init__()

        # setup embedding table remotely
        self.emb_table_rref = rpc.remote(ps, EmbeddingTable, args=(ntoken, ninp, dropout))
        # setup LSTM locally
        self.rnn = nn.LSTM(ninp, nhid, nlayers, dropout=dropout)
        # setup decoder remotely
        self.decoder_rref = rpc.remote(ps, Decoder, args=(ntoken, nhid, dropout))

    def forward(self, input, hidden):
        # pass input to the remote embedding table and fetch emb tensor back
        emb = _remote_method(EmbeddingTable.forward, self.emb_table_rref, input)
        output, hidden = self.rnn(emb, hidden)
        # pass output to the rremote decoder and get the decoded output back
        decoded = _remote_method(Decoder.forward, self.decoder_rref, output)
        return decoded, hidden
```

https://pytorch.org/tutorials/intermediate/rpc_tutorial.html

gRPC As an RPC Implementation

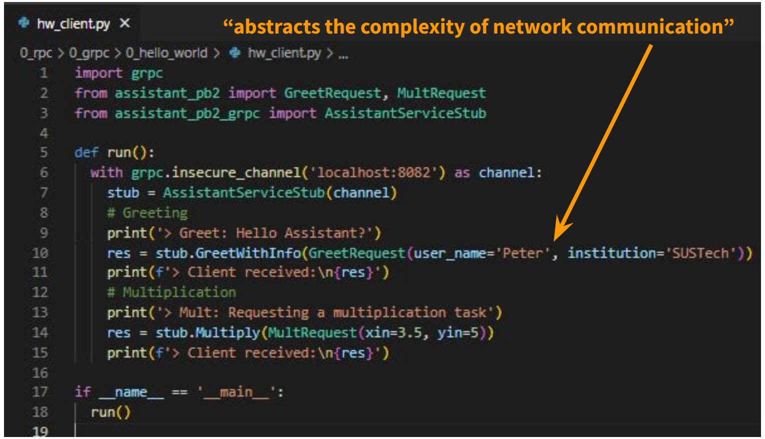
- "gRPC is a cross-platform high-performance remote procedure call (RPC) framework, created by Google."
- "gRPC is a **Cloud Native** Computing Foundation (CNCF) project."

gRPC

Used by

Square NETFLIX Cockroach LABS cisco JUNIPer NETWORKS

- How does gRPC fulfill the design concerns?
  a. Protocol
  b. Communication
  c. Proxy



"abstracts the complexity of network communication"

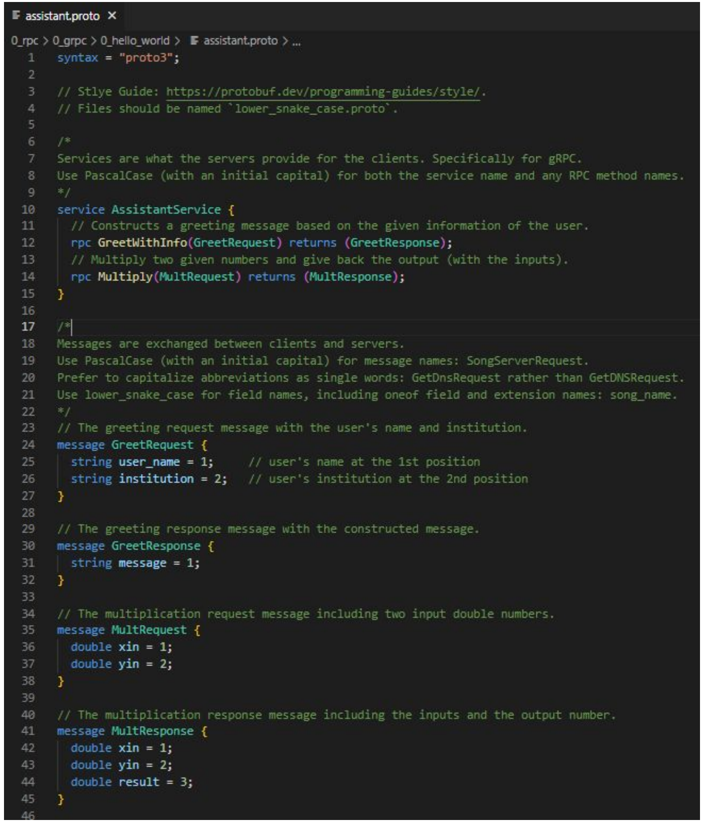
```python
import grpc
from assistant_pb2 import GreetRequest, MultRequest
from assistant_pb2_grpc import AssistantServiceStub

def run():
  with grpc.insecure_channel('localhost:8082') as channel:
    stub = AssistantServiceStub(channel)
    # Greeting
    print('> Greet: Hello Assistant?')
    res = stub.GreetWithInfo(GreetRequest(user_name='Peter', institution='SUSTech'))
    print(f'> Client received:\n{res}')
    # Multiplication
    print('> Mult: Requesting a multiplication task')
    res = stub.Multiply(MultRequest(xin=3.5, yin=5))
    print(f'> Client received:\n{res}')

if __name__ == '__main__':
  run()
```

TASK: gRPC - Hello World

Implement a Hello World gRPC Python example.

> Reference codebase: `rpc_grpc_hello_world`

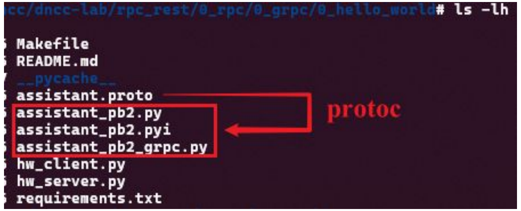
1. Set up Python (Miniconda is recommended).
2. Install Python dependencies into a Conda environment via:
   - ○ `python -m pip install -r requirements.txt`
3. Check the protocol file `assistant.proto`. Use `protoc` to generate some code:
   - ○ `python -m grpc_tools.protoc -I./ --python_out=. --pyi_out=. --grpc_python_out=. assistant.proto`
   - ○ These generated code will be utilized to implement the gRPC client and the gRPC server.



assistant.proto

0_rpc > 0_grpc > 0_hello_world > assistant.proto > ...

```
syntax = "proto3";

// Stlye Guide: https://protobuf.dev/programming-guides/style/.
// Files should be named `lower_snake_case.proto`.

/*
Services are what the servers provide for the clients. Specifically for gRPC.
Use PascalCase (with an initial capital) for both the service name and any RPC method names.
*/
service AssistantService {
  // Constructs a greeting message based on the given information of the user.
  rpc GreetWithInfo(GreetRequest) returns (GreetResponse);
  // Multiply two given numbers and give back the output (with the inputs).
  rpc Multiply(MultRequest) returns (MultResponse);
}

/*
Messages are exchanged between clients and servers.
Use PascalCase (with an initial capital) for message names: SongServerRequest.
Prefer to capitalize abbreviations as single words: GetDnsRequest rather than GetDNSRequest.
Use lower_snake_case for field names, including oneof field and extension names: song_name.
*/
// The greeting request message with the user's name and institution.
message GreetRequest {
  string user_name = 1;     // user's name at the 1st position
  string institution = 2;   // user's institution at the 2nd position
}

// The greeting response message with the constructed message.
message GreetResponse {
  string message = 1;
}

// The multiplication request message including two input double numbers.
message MultRequest {
  double xin = 1;
  double yin = 2;
}

// The multiplication response message including the inputs and the output number.
message MultResponse {
  double xin = 1;
  double yin = 2;
  double result = 3;
}

```



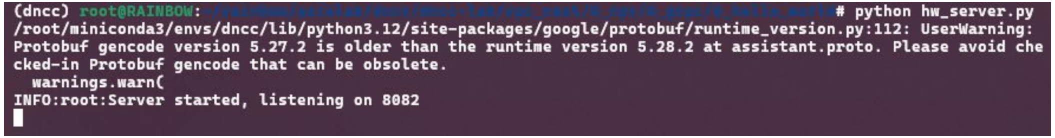
```
cc/dncc-lab/rpc_rest/0_rpc/0_grpc/0_hello_world# ls -lh

Makefile
README.md
__pycache__
assistant.proto
assistant_pb2.py
assistant_pb2.pyi
assistant_pb2_grpc.py
hw_client.py
hw_server.py
requirements.txt
```

TASK: gRPC - Hello World

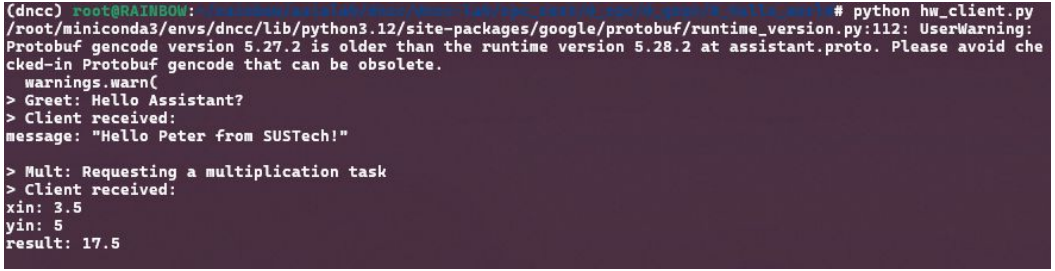
Implement a Hello World gRPC Python example.

> Reference codebase: `rpc_grpc_hello_world`

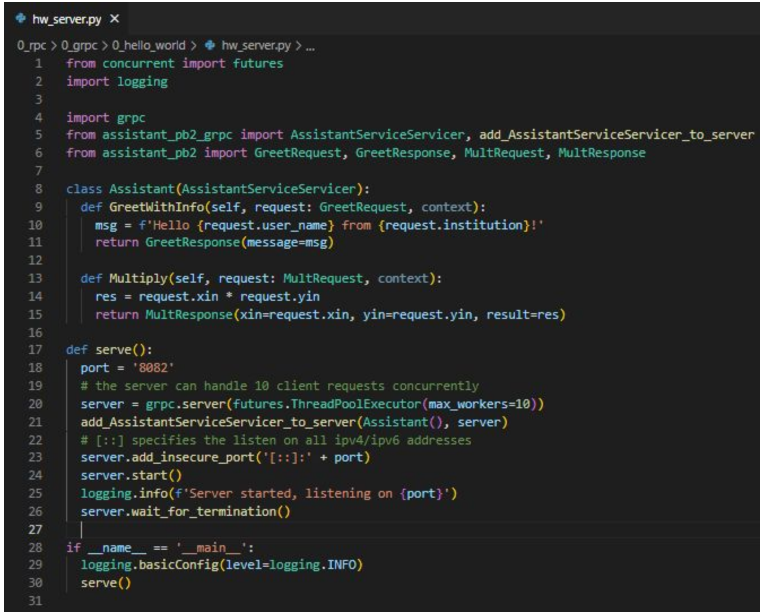
4. Implement the gRPC server in `hw_server.py`. Run the gRPC server via:
   - `python hw_server.py`
5. Implement the gRPC client in `hw_client.py`. In another terminal, run the gRPC client via:
   - `python hw_client.py`



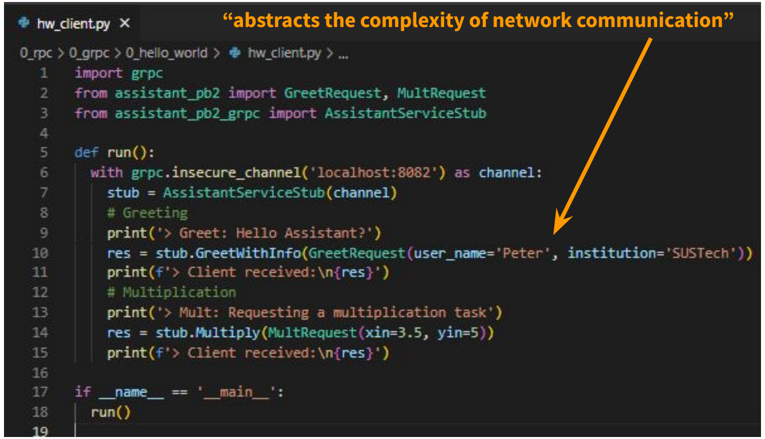
```
(dncc) root@RAINBOW:[redacted]# python hw_server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/runtime_version.py:112: UserWarning:
Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at assistant.proto. Please avoid che
cked-in Protobuf gencode that can be obsolete.
  warnings.warn(
INFO:root:Server started, listening on 8082
█
```



```
(dncc) root@RAINBOW:[redacted]# python hw_client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/runtime_version.py:112: UserWarning:
Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at assistant.proto. Please avoid che
cked-in Protobuf gencode that can be obsolete.
  warnings.warn(
> Greet: Hello Assistant?
> Client received:
message: "Hello Peter from SUSTech!"

> Mult: Requesting a multiplication task
> Client received:
xin: 3.5
yin: 5
result: 17.5
```



```python
from concurrent import futures
import logging

import grpc
from assistant_pb2_grpc import AssistantServiceServicer, add_AssistantServiceServicer_to_server
from assistant_pb2 import GreetRequest, GreetResponse, MultRequest, MultResponse

class Assistant(AssistantServiceServicer):
  def GreetWithInfo(self, request: GreetRequest, context):
    msg = f'Hello {request.user_name} from {request.institution}!'
    return GreetResponse(message=msg)

  def Multiply(self, request: MultRequest, context):
    res = request.xin * request.yin
    return MultResponse(xin=request.xin, yin=request.yin, result=res)

def serve():
  port = '8082'
  # the server can handle 10 client requests concurrently
  server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
  add_AssistantServiceServicer_to_server(Assistant(), server)
  # [::] specifies the listen on all ipv4/ipv6 addresses
  server.add_insecure_port('[::]:' + port)
  server.start()
  logging.info(f'Server started, listening on {port}')
  server.wait_for_termination()

if __name__ == '__main__':
  logging.basicConfig(level=logging.INFO)
  serve()

```

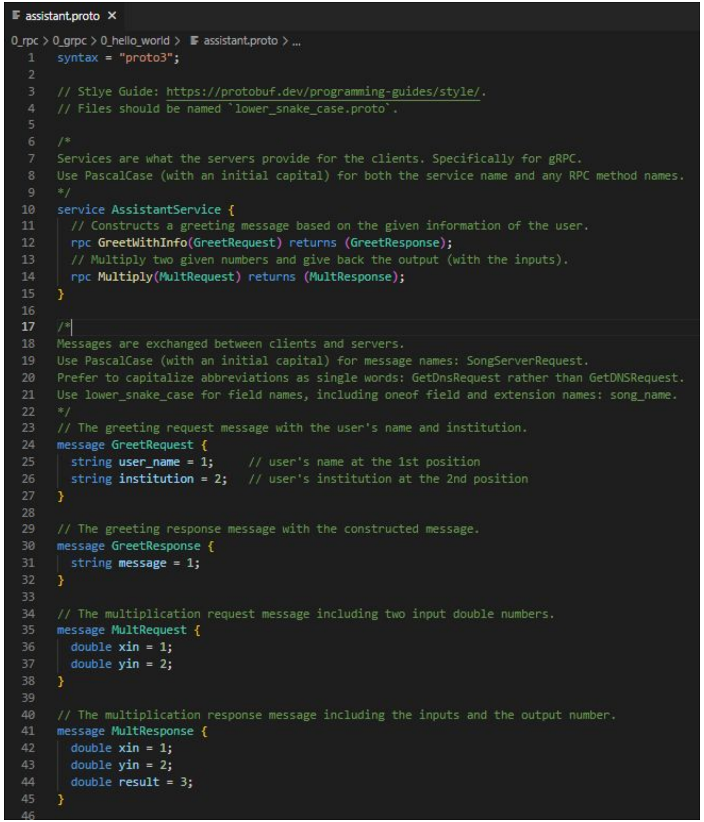


“abstracts the complexity of network communication”

```python
import grpc
from assistant_pb2 import GreetRequest, MultRequest
from assistant_pb2_grpc import AssistantServiceStub

def run():
  with grpc.insecure_channel('localhost:8082') as channel:
    stub = AssistantServiceStub(channel)
    # Greeting
    print('> Greet: Hello Assistant?')
    res = stub.GreetWithInfo(GreetRequest(user_name='Peter', institution='SUSTech'))
    print(f'> Client received:\n{res}')
    # Multiplication
    print('> Mult: Requesting a multiplication task')
    res = stub.Multiply(MultRequest(xin=3.5, yin=5))
    print(f'> Client received:\n{res}')

if __name__ == '__main__':
  run()

```

gRPC - Protocol Definition

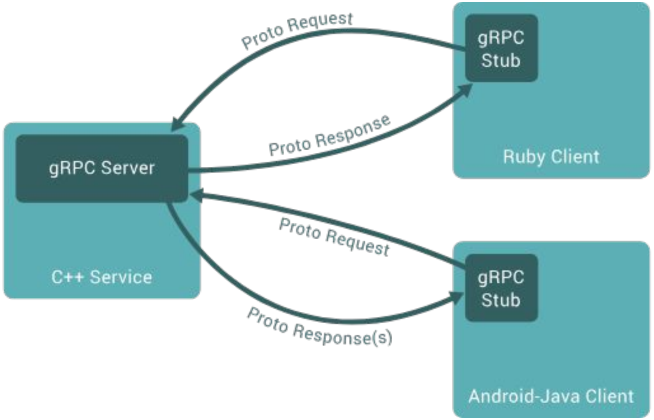
- gRPC uses **proto files** to define:
  - messages `repeated string snippets = 3;`
    - field property (e.g., `repeated`)
    - field type (e.g., `string`)
    - field name
    - field tag/number: ordering of fields
  - remote procedures
    - name
    - request message
    - response message
  - services
    - organized set of remote procedures
- Proto files are a part of Protocol Buffers (**Protobuf**). We will introduce a bit more about Protobuf in the next lab session.



```
syntax = "proto3";

// Stlye Guide: https://protobuf.dev/programming-guides/style/.
// Files should be named `lower_snake_case.proto`.

/*
Services are what the servers provide for the clients. Specifically for gRPC.
Use PascalCase (with an initial capital) for both the service name and any RPC method names.
*/
service AssistantService {
  // Constructs a greeting message based on the given information of the user.
  rpc GreetWithInfo(GreetRequest) returns (GreetResponse);
  // Multiply two given numbers and give back the output (with the inputs).
  rpc Multiply(MultRequest) returns (MultResponse);
}

/*
Messages are exchanged between clients and servers.
Use PascalCase (with an initial capital) for message names: SongServerRequest.
Prefer to capitalize abbreviations as single words: GetDnsRequest rather than GetDNSRequest.
Use lower_snake_case for field names, including oneof field and extension names: song_name.
*/
// The greeting request message with the user's name and institution.
message GreetRequest {
  string user_name = 1;     // user's name at the 1st position
  string institution = 2;   // user's institution at the 2nd position
}

// The greeting response message with the constructed message.
message GreetResponse {
  string message = 1;
}

// The multiplication request message including two input double numbers.
message MultRequest {
  double xin = 1;
  double yin = 2;
}

// The multiplication response message including the inputs and the output number.
message MultResponse {
  double xin = 1;
  double yin = 2;
  double result = 3;
}

```

gRPC - Protocol Definition

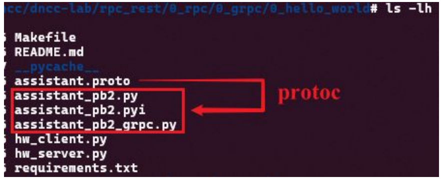
- gRPC uses **proto files** to define:
  - messages
  - remote procedures
  - services
- Proto files serve as the Interface Definition Language (IDL) for RPC.
  - “a language that lets a program written in one language communicate with another program written in an unknown language.”
  - Example: a gRPC server is implemented in C++. It can interact with a Ruby gRPC client, or a Java gRPC client from Android platform.



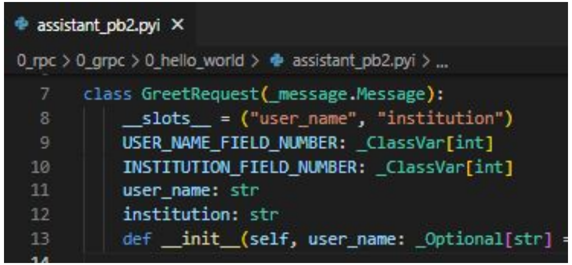
https://grpc.io/docs/what-is-grpc/introduction/

gRPC - Protocol Definition

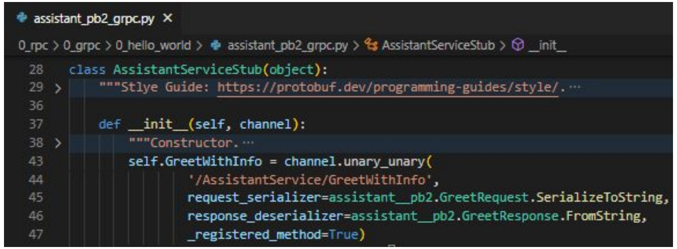
- gRPC uses **proto files** to define:
  - messages
  - remote procedures
  - services
- Proto files serve as the Interface Definition Language (IDL) for RPC.
  - “a language that lets a program written in one language communicate with another program written in an unknown language.”
- Proto files can be compiled by `protoc` to generate code from different programming languages:
  - message classes
  - gRPC client stub
  - gRPC server template



```
cc/dncc-lab/rpc_rest/0_rpc/0_grpc/0_hello_world# ls -lh

Makefile
README.md
__pycache__
assistant.proto
assistant_pb2.py
assistant_pb2.pyi
assistant_pb2_grpc.py
hw_client.py
hw_server.py
requirements.txt
```

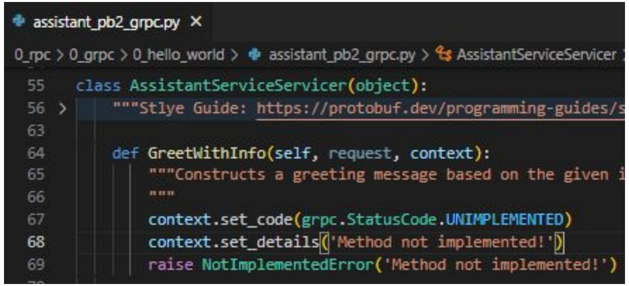


```python
class GreetRequest(_message.Message):
    __slots__ = ("user_name", "institution")
    USER_NAME_FIELD_NUMBER: _ClassVar[int]
    INSTITUTION_FIELD_NUMBER: _ClassVar[int]
    user_name: str
    institution: str
    def __init__(self, user_name: _Optional[str] =
```



assistant_pb2_grpc.py ×
0_rpc > 0_grpc > 0_hello_world > assistant_pb2_grpc.py > AssistantServiceStub > __init__

```python
class AssistantServiceStub(object):
    """Stlye Guide: https://protobuf.dev/programming-guides/style/.···

    def __init__(self, channel):
        """Constructor.···
        self.GreetWithInfo = channel.unary_unary(
                '/AssistantService/GreetWithInfo',
                request_serializer=assistant__pb2.GreetRequest.SerializeToString,
                response_deserializer=assistant__pb2.GreetResponse.FromString,
                _registered_method=True)
```



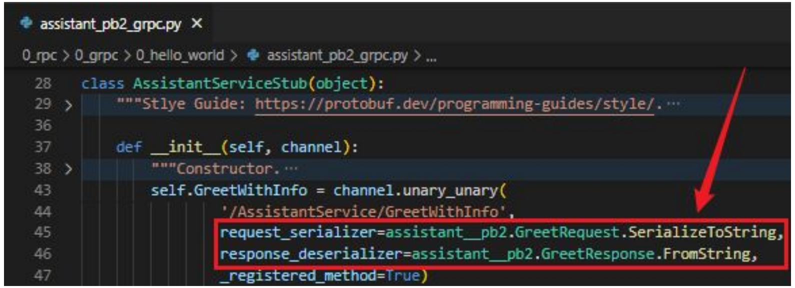
assistant_pb2_grpc.py ×
0_rpc > 0_grpc > 0_hello_world > assistant_pb2_grpc.py > AssistantServiceServicer

```python
class AssistantServiceServicer(object):
    """Stlye Guide: https://protobuf.dev/programming-guides/s

    def GreetWithInfo(self, request, context):
        """Constructs a greeting message based on the given i
        """
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')
```

gRPC - Communication Handling

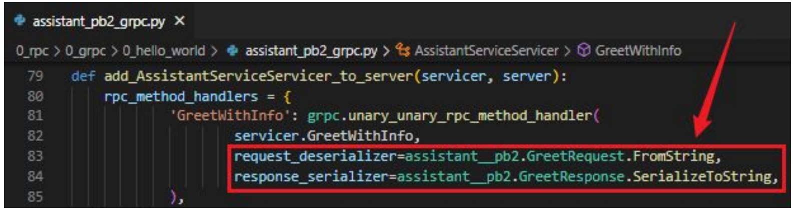
- **Serialization**:
  - Messages defined in the proto files are transmitted between the gRPC client and server.
  - gRPC uses **Protocol Buffers (Protobuf)** to serialize proto messages into binaries.
  - “Protocol Buffers are language-neutral, platform-neutral extensible mechanisms for serializing structured data.”
    - provides more compact data;
    - enables faster transmission.

- We will further examine how Protobuf serializes messages in the next lab session.



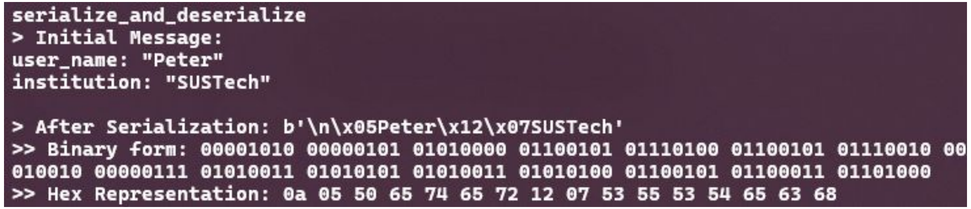
assistant_pb2_grpc.py ×
0_rpc > 0_grpc > 0_hello_world > assistant_pb2_grpc.py > ...

```python
class AssistantServiceStub(object):
    """Stlye Guide: https://protobuf.dev/programming-guides/style/.···

    def __init__(self, channel):
        """Constructor.···
        self.GreetWithInfo = channel.unary_unary(
                '/AssistantService/GreetWithInfo',
                request_serializer=assistant__pb2.GreetRequest.SerializeToString,
                response_deserializer=assistant__pb2.GreetResponse.FromString,
                _registered_method=True)
```



assistant_pb2_grpc.py ×
0_rpc > 0_grpc > 0_hello_world > assistant_pb2_grpc.py > AssistantServiceServicer > GreetWithInfo

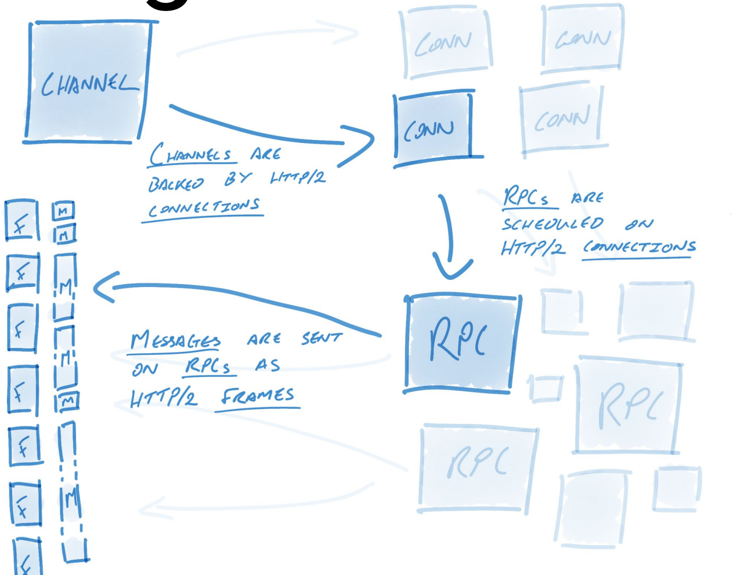
```python
def add_AssistantServiceServicer_to_server(servicer, server):
    rpc_method_handlers = {
            'GreetWithInfo': grpc.unary_unary_rpc_method_handler(
                    servicer.GreetWithInfo,
                    request_deserializer=assistant__pb2.GreetRequest.FromString,
                    response_serializer=assistant__pb2.GreetResponse.SerializeToString,
            ),
```

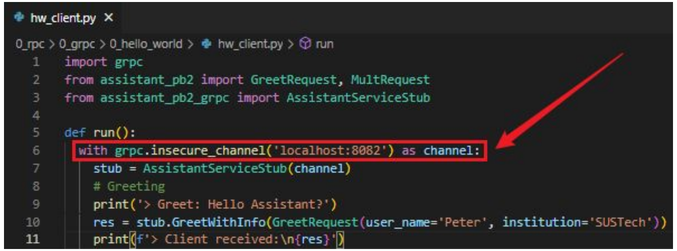


```
serialize_and_deserialize
> Initial Message:
user_name: "Peter"
institution: "SUSTech"

> After Serialization: b'\n\x05Peter\x12\x07SUSTech'
>> Binary form: 00001010 00000101 01010000 01100101 01110100 01100101 01110010 00
010010 00000111 01010011 01010101 01010011 01010100 01100101 01100011 01101000
>> Hex Representation: 0a 05 50 65 74 65 72 12 07 53 55 53 54 65 63 68
```

gRPC - Communication Handling

- **Serialization**:
  - Messages defined in the proto files are transmitted between the gRPC client and server.
  - gRPC uses **Protocol Buffers (Protobuf)** to serialize proto messages into binaries.
- **Network Protocol**:
  - gRPC transmits serialized request/response messages over **HTTP/2 framing**.
  - A channel represents an HTTP/2 connection (a TCP connection behind the scene).
  - A channel manages multiple logical HTTP/2 streams, each dedicated to one procedure.
  - Messages are sent on the corresponding HTTP/2 streams as HTTP/2 frames.


CHANNEL
CONN
CONN
CONN
CONN
CHANNELS ARE BACKED BY HTTP/2 CONNECTIONS
RPCs ARE SCHEDULED ON HTTP/2 CONNECTIONS
RPC
RPC
RPC
MESSAGES ARE SENT ON RPCs AS HTTP/2 FRAMES
F
M

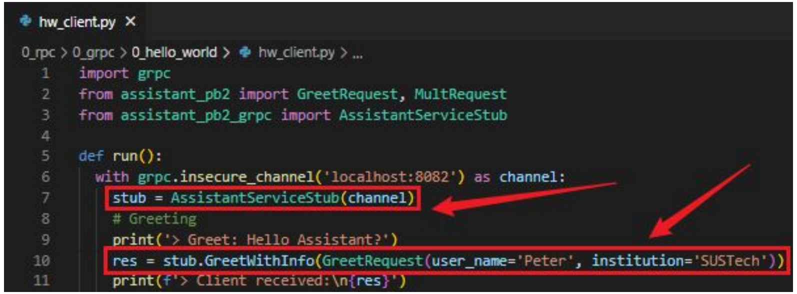


```python
import grpc
from assistant_pb2 import GreetRequest, MultRequest
from assistant_pb2_grpc import AssistantServiceStub

def run():
    with grpc.insecure_channel('localhost:8082') as channel:
        stub = AssistantServiceStub(channel)
        # Greeting
        print('> Greet: Hello Assistant?')
        res = stub.GreetWithInfo(GreetRequest(user_name='Peter', institution='SUSTech'))
        print(f'> Client received:\n{res}')
```

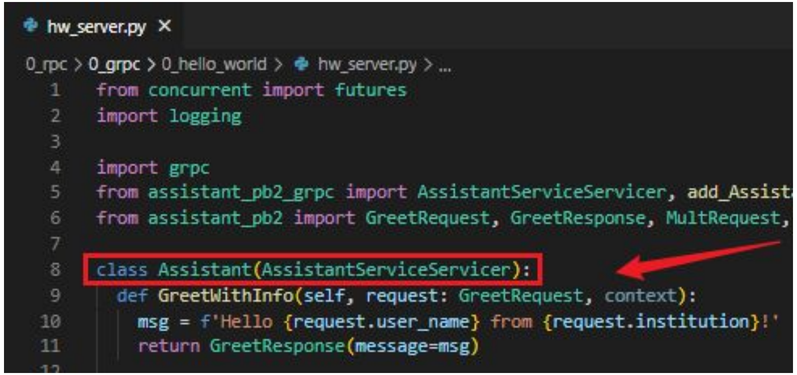
https://grpc.io/blog/grpc-on-http2/

gRPC - Proxy

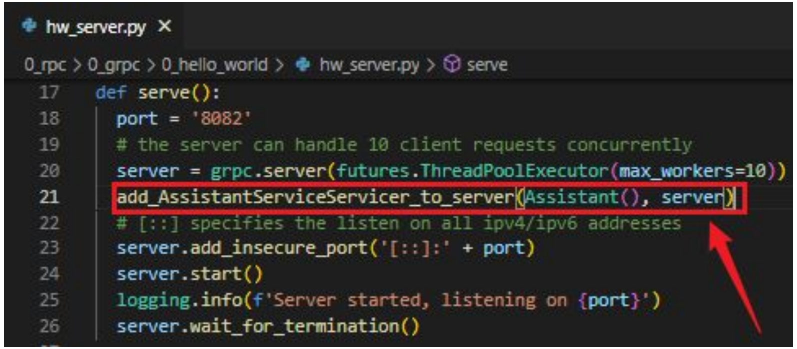
- Client and server stubs are pre-generated via `protoc` in a static fashion.
  - message classes
  - gRPC client stub
    - The client stub is retrieved from the gRPC channel.
    - The client calls the remote procedure via the client stub.
  - gRPC server template
    - The server extends the generated template and provides the procedure implementation.
    - The server registers the implemented procedure.



```python
import grpc
from assistant_pb2 import GreetRequest, MultRequest
from assistant_pb2_grpc import AssistantServiceStub

def run():
  with grpc.insecure_channel('localhost:8082') as channel:
    stub = AssistantServiceStub(channel)
    # Greeting
    print('> Greet: Hello Assistant?')
    res = stub.GreetWithInfo(GreetRequest(user_name='Peter', institution='SUSTech'))
    print(f'> Client received:\n{res}')
```



```python
from concurrent import futures
import logging

import grpc
from assistant_pb2_grpc import AssistantServiceServicer, add_Assist
from assistant_pb2 import GreetRequest, GreetResponse, MultRequest,

class Assistant(AssistantServiceServicer):
  def GreetWithInfo(self, request: GreetRequest, context):
    msg = f'Hello {request.user_name} from {request.institution}!'
    return GreetResponse(message=msg)
```



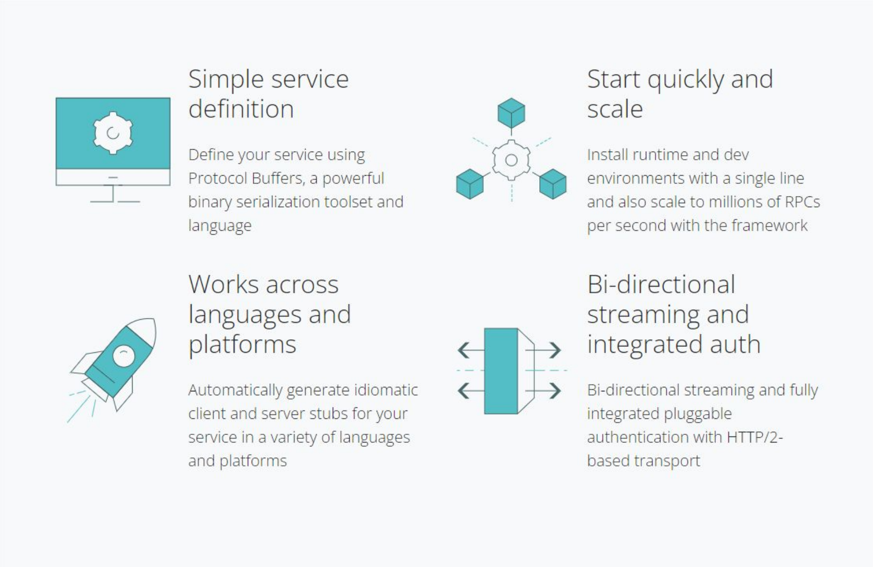
```python
def serve():
  port = '8082'
  # the server can handle 10 client requests concurrently
  server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
  add_AssistantServiceServicer_to_server(Assistant(), server)
  # [::] specifies the listen on all ipv4/ipv6 addresses
  server.add_insecure_port('[::]:' + port)
  server.start()
  logging.info(f'Server started, listening on {port}')
  server.wait_for_termination()
```

gRPC - Benefits

Why is gRPC so popular:

1. Static & automatic generation of client stubs & server templates via protoc.
2. Efficient communication via Protobuf.
3. Cross-Language + Cross-Platform support.
4. Bidirectional Streaming.
5. Multiplexing via HTTP/2.
6. …

- More details in the next lab session.



Simple service definition

Define your service using Protocol Buffers, a powerful binary serialization toolset and language

Start quickly and scale

Install runtime and dev environments with a single line and also scale to millions of RPCs per second with the framework

Works across languages and platforms

Automatically generate idiomatic client and server stubs for your service in a variety of languages and platforms

Bi-directional streaming and integrated auth

Bi-directional streaming and fully integrated pluggable authentication with HTTP/2-based transport

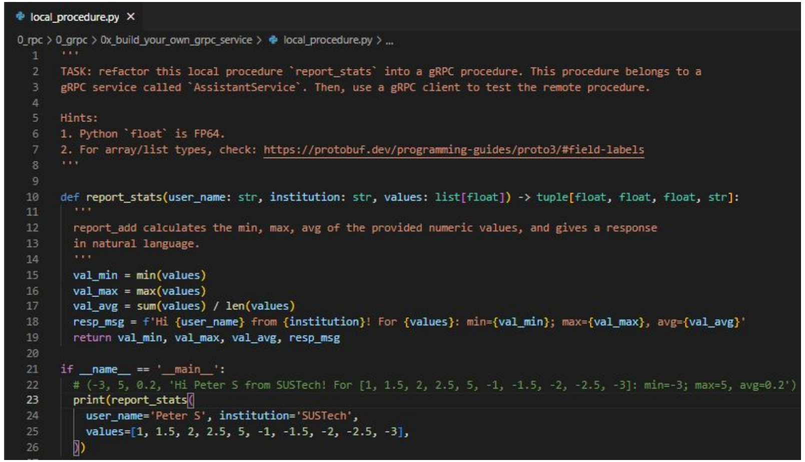
https://grpc.io/

TASK: gRPC - Build Your Own gRPC Service

Refactor a local procedure into a remote procedure.
> Reference codebase: `rpc_grpc_byogrpc`

Requirements:

1. The remote procedure should be hosted by the `AssistantService` in the gRPC server.
2. A gRPC client should be written to test the remote procedure.

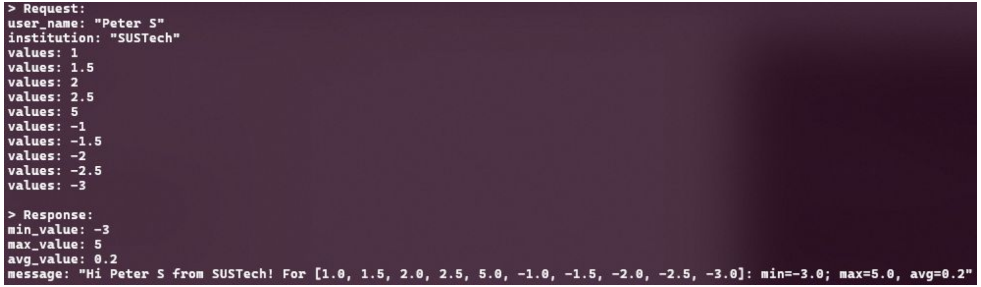


local_procedure.py ×
0_rpc > 0_grpc > 0x_build_your_own_grpc_service > local_procedure.py > ...

```python
'''
TASK: refactor this local procedure `report_stats` into a gRPC procedure. This procedure belongs to a
gRPC service called `AssistantService`. Then, use a gRPC client to test the remote procedure.

Hints:
1. Python `float` is FP64.
2. For array/list types, check: https://protobuf.dev/programming-guides/proto3/#field-labels
'''

def report_stats(user_name: str, institution: str, values: list[float]) -> tuple[float, float, float, str]:
    '''
    report_add calculates the min, max, avg of the provided numeric values, and gives a response
    in natural language.
    '''
    val_min = min(values)
    val_max = max(values)
    val_avg = sum(values) / len(values)
    resp_msg = f'Hi {user_name} from {institution}! For {values}: min={val_min}; max={val_max}, avg={val_avg}'
    return val_min, val_max, val_avg, resp_msg

if __name__ == '__main__':
    # (-3, 5, 0.2, 'Hi Peter S from SUSTech! For [1, 1.5, 2, 2.5, 5, -1, -1.5, -2, -2.5, -3]: min=-3; max=5, avg=0.2')
    print(report_stats(
        user_name='Peter S', institution='SUSTech',
        values=[1, 1.5, 2, 2.5, 5, -1, -1.5, -2, -2.5, -3],
    ))
```

Reference Steps:

1. Write the proto file.
2. Generate code stubs via protoc.
3. Implement and run the gRPC server.
4. Implement and run the gRPC client.

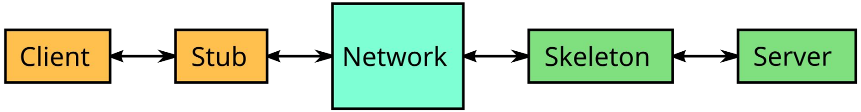


```
> Request:
user_name: "Peter S"
institution: "SUSTech"
values: 1
values: 1.5
values: 2
values: 2.5
values: 5
values: -1
values: -1.5
values: -2
values: -2.5
values: -3

> Response:
min_value: -3
max_value: 5
avg_value: 0.2
message: "Hi Peter S from SUSTech! For [1.0, 1.5, 2.0, 2.5, 5.0, -1.0, -1.5, -2.0, -2.5, -3.0]: min=-3.0; max=5.0, avg=0.2"
```

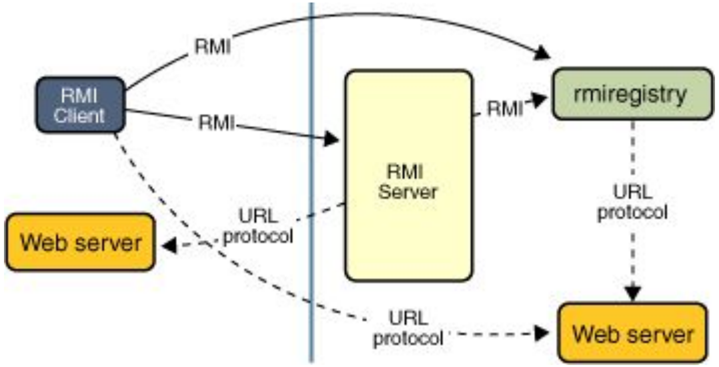
Try starting from scratch to be familiar with the process before Assignment 2.

Java RMI As Another RPC Implementation

- “Java Remote Method Invocation (Java RMI) enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts.”
- Java RMI is a **Java-specific Object-Oriented** implementation of the RPC protocol.



- We introduced only Java RMI in our previous semesters, but since gRPC has now become one of the most popular RPC solutions, we will put more focus on gRPC from now on.
- Nevertheless, it is still helpful to check Java RMI and think about why gRPC receives more flowers.



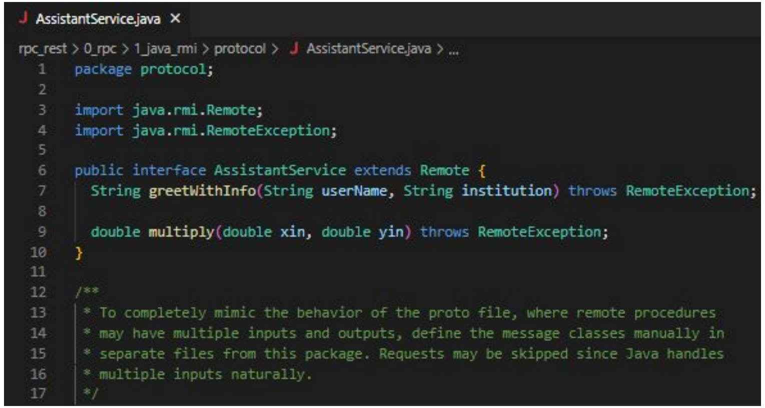
https://en.wikipedia.org/wiki/Java_remote_method_invocation
https://docs.oracle.com/javase/tutorial/rmi/overview.html

TASK: Java RMI - Hello World

Implement a Hello World Java RMI example.

> Reference codebase: `rpc_javarmi_hello_world`

1. Set up Java Development Kit (JDK) via:
   - `apt install default-jdk`
   - Java RMI is a built-in library, and RMI Registry is natively implemented in the `rmiregistry` program.
2. Check the remote interface definition file `AssistantService.java`.
   - A Java interface is defined with two remote functions.
   - The outputs are simplified to one single value, since Java cannot handle multiple output values naturally. To refine, define separate message classes.



AssistantService.java ×
rpc_rest > 0_rpc > 1_java_rmi > protocol > AssistantService.java > ...

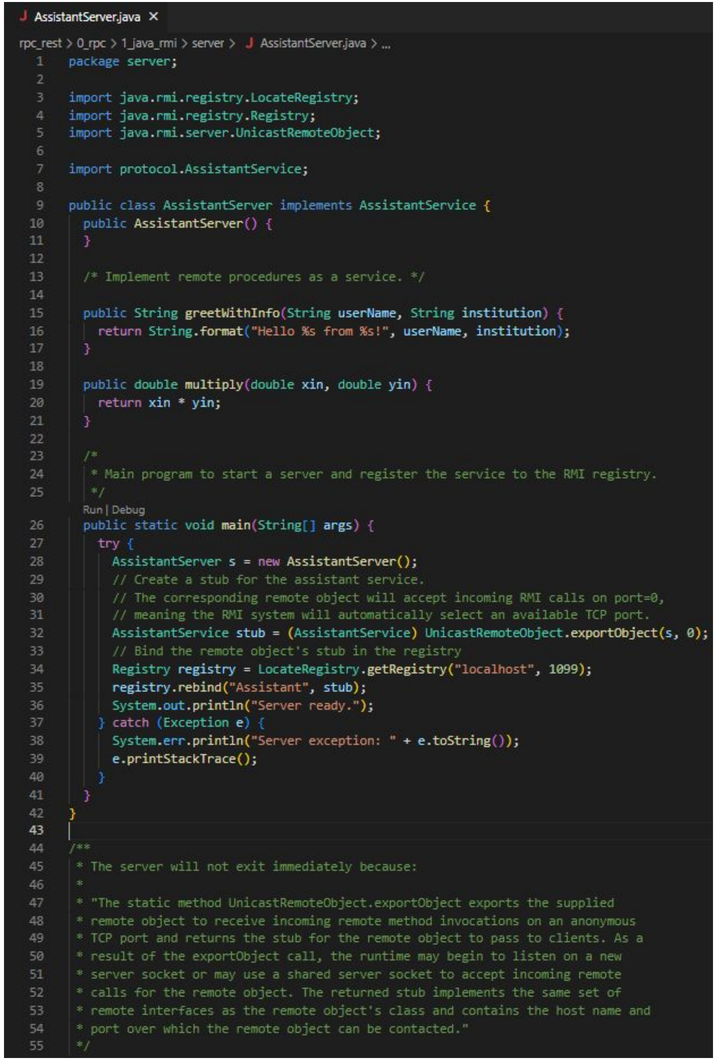
```java
package protocol;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AssistantService extends Remote {
  String greetWithInfo(String userName, String institution) throws RemoteException;

  double multiply(double xin, double yin) throws RemoteException;
}

/**
 * To completely mimic the behavior of the proto file, where remote procedures
 * may have multiple inputs and outputs, define the message classes manually in
 * separate files from this package. Requests may be skipped since Java handles
 * multiple inputs naturally.
 */
```

TASK: Java RMI - Hello World

Implement a Hello World Java RMI example.

> Reference codebase: `rpc_javarmi_hello_world`

1. Set up Java Development Kit (JDK) via:
   - `apt install default-jdk`
   - Java RMI is a built-in library, and RMI Registry is natively implemented in the `rmiregistry` program.
2. Check the remote interface definition file `AssistantService.java`.
3. Implement the RMI server: `AssistantServer.java`.
   - The interface is “implemented”.
   - A client stub is generated and registered to the RMI registry.
   - The server listens on a server socket and accept incoming remote calls after stub generation.


AssistantServer.java ×
rpc_rest > 0_rpc > 1_java_rmi > server > AssistantServer.java > ...
Run | Debug

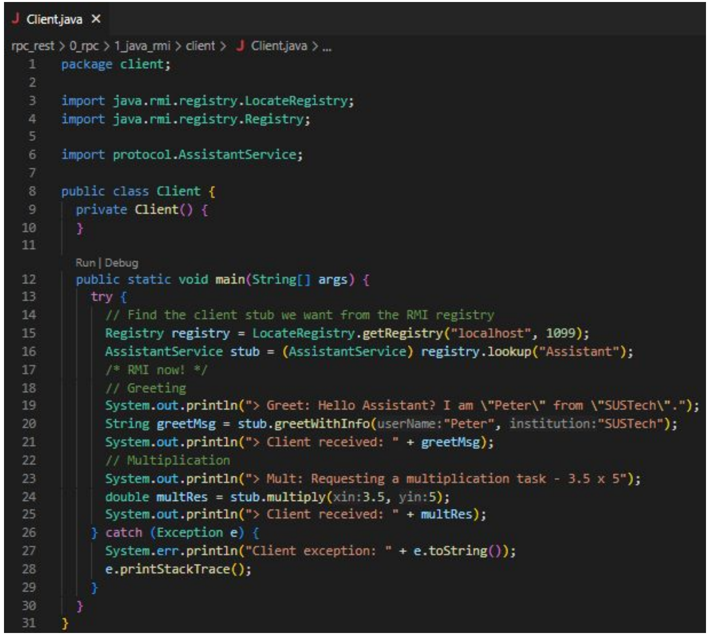
```java
package server;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import protocol.AssistantService;

public class AssistantServer implements AssistantService {
  public AssistantServer() {
  }

  /* Implement remote procedures as a service. */

  public String greetWithInfo(String userName, String institution) {
    return String.format("Hello %s from %s!", userName, institution);
  }

  public double multiply(double xin, double yin) {
    return xin * yin;
  }

  /*
   * Main program to start a server and register the service to the RMI registry.
   */
  public static void main(String[] args) {
    try {
      AssistantServer s = new AssistantServer();
      // Create a stub for the assistant service.
      // The corresponding remote object will accept incoming RMI calls on port=0,
      // meaning the RMI system will automatically select an available TCP port.
      AssistantService stub = (AssistantService) UnicastRemoteObject.exportObject(s, 0);
      // Bind the remote object's stub in the registry
      Registry registry = LocateRegistry.getRegistry("localhost", 1099);
      registry.rebind("Assistant", stub);
      System.out.println("Server ready.");
    } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
      e.printStackTrace();
    }
  }
}

/**
 * The server will not exit immediately because:
 *
 * "The static method UnicastRemoteObject.exportObject exports the supplied
 * remote object to receive incoming remote method invocations on an anonymous
 * TCP port and returns the stub for the remote object to pass to clients. As a
 * result of the exportObject call, the runtime may begin to listen on a new
 * server socket or may use a shared server socket to accept incoming remote
 * calls for the remote object. The returned stub implements the same set of
 * remote interfaces as the remote object's class and contains the host name and
 * port over which the remote object can be contacted."
 */
```

TASK: Java RMI - Hello World

Implement a Hello World Java RMI example.

> Reference codebase: `rpc_javarmi_hello_world`

1. Set up Java Development Kit (JDK) via:
   - `apt install default-jdk`
   - Java RMI is a built-in library, and RMI Registry is natively implemented in the `rmiregistry` program.
2. Check the remote interface definition file `AssistantService.java`.
3. Implement the RMI server: `AssistantServer.java`.
4. Implement the RMI client: `Client.java`.
   - The client stub is fetched from the RMI registry.
   - RMI is performed using the client stub as if the remote object were local.



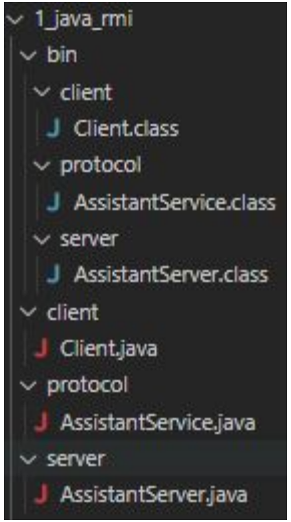
```java
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import protocol.AssistantService;

public class Client {
  private Client() {
  }

  public static void main(String[] args) {
    try {
      // Find the client stub we want from the RMI registry
      Registry registry = LocateRegistry.getRegistry("localhost", 1099);
      AssistantService stub = (AssistantService) registry.lookup("Assistant");
      /* RMI now! */
      // Greeting
      System.out.println("> Greet: Hello Assistant? I am \"Peter\" from \"SUSTech\".");
      String greetMsg = stub.greetWithInfo(userName:"Peter", institution:"SUSTech");
      System.out.println("> Client received: " + greetMsg);
      // Multiplication
      System.out.println("> Mult: Requesting a multiplication task - 3.5 x 5");
      double multRes = stub.multiply(xin:3.5, yin:5);
      System.out.println("> Client received: " + multRes);
    } catch (Exception e) {
      System.err.println("Client exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

TASK: Java RMI - Hello World

Implement a Hello World Java RMI example.

> Reference codebase: `rpc_javarmi_hello_world`

1. Set up Java Development Kit (JDK) via:
   - `apt install default-jdk`
   - Java RMI is a built-in library, and RMI Registry is natively implemented in the `rmiregistry` program.
2. Check the remote interface definition file `AssistantService.java`.
3. Implement the RMI server: `AssistantServer.java`.
4. Implement the RMI client: `Client.java`.
5. Compile the source files via:
   - `javac -d bin/ protocol/AssistantService.java server/AssistantServer.java client/Client.java`
   - The binaries will be output to the `bin/` folder.



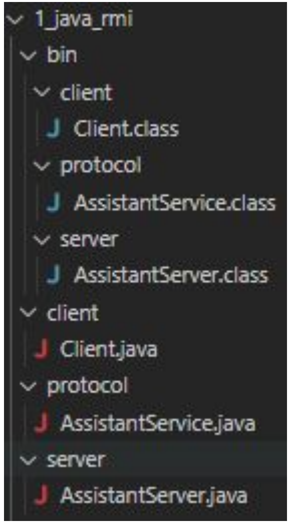
```
1_java_rmi
  bin
    client
      Client.class
    protocol
      AssistantService.class
    server
      AssistantServer.class
  client
    Client.java
  protocol
    AssistantService.java
  server
    AssistantServer.java
```

TASK: Java RMI - Hello World

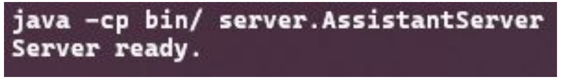
Implement a Hello World Java RMI example.

> Reference codebase: `rpc_javarmi_hello_world`

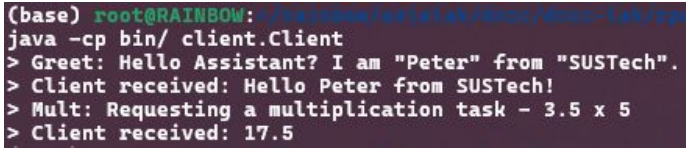
6. Run the RMI registry at port=`1099` from the class path directory `bin/`:
   - `cd bin/`
   - `rmiregistry 1099`
   - The RMI registry **MUST** be started from the generated class path folder, otherwise the server and the client will fail to locate the defined interface so that they can marshal and unmarshal remote objects as required in Java RMI.
7. Start the server in another terminal:
   - `java -cp bin/ server.AssistantServer`
8. Execute the client in another terminal:
   - `java -cp bin/ client.Client`



```
⌄ 1_java_rmi
  ⌄ bin
    ⌄ client
       J Client.class
    ⌄ protocol
       J AssistantService.class
    ⌄ server
       J AssistantServer.class
  ⌄ client
     J Client.java
  ⌄ protocol
     J AssistantService.java
  ⌄ server
     J AssistantServer.java
```



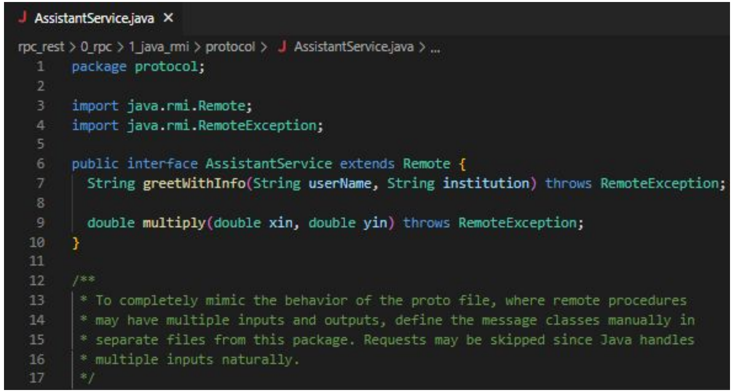
```
java -cp bin/ server.AssistantServer
Server ready.
```



```
(base) root@RAINBOW:[illegible]
java -cp bin/ client.Client
> Greet: Hello Assistant? I am "Peter" from "SUSTech".
> Client received: Hello Peter from SUSTech!
> Mult: Requesting a multiplication task - 3.5 x 5
> Client received: 17.5
```

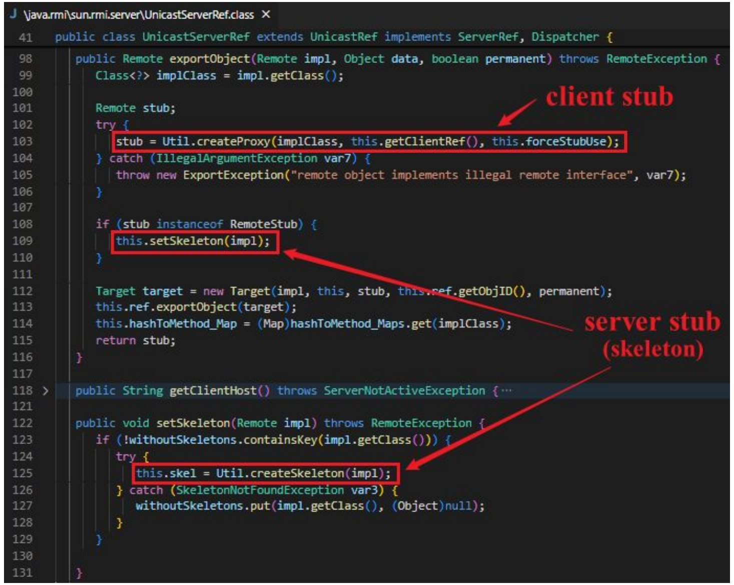
Java RMI - Design Concerns

- **Protocol**:
  - uses Java interfaces to specify the remote method signatures in an OOP fashion.
- **Communication**:
  - **Serialization**: object serialization via `java.io.Serializable`.
  - **Network Protocol**: naive Java RMI establishes a new TCP connection for each RMI request.
- **Proxy**:
  - Dynamic Proxy - client stubs & server stubs (skeletons) are dynamically generated at runtime utilizing Java Reflection.
  - Since client stubs are dynamically generated by the server, an additional RMI registry is needed to forward the stubs to the client.



AssistantService.java ×
rpc_rest > 0_rpc > 1_java_rmi > protocol > AssistantService.java > ...

```java
package protocol;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AssistantService extends Remote {
  String greetWithInfo(String userName, String institution) throws RemoteException;

  double multiply(double xin, double yin) throws RemoteException;
}

/**
 * To completely mimic the behavior of the proto file, where remote procedures
 * may have multiple inputs and outputs, define the message classes manually in
 * separate files from this package. Requests may be skipped since Java handles
 * multiple inputs naturally.
 */
```



```java
public class UnicastServerRef extends UnicastRef implements ServerRef, Dispatcher {
    public Remote exportObject(Remote impl, Object data, boolean permanent) throws RemoteException {
        Class<?> implClass = impl.getClass();

        Remote stub;
        try {
            stub = Util.createProxy(implClass, this.getClientRef(), this.forceStubUse);
        } catch (IllegalArgumentException var7) {
            throw new ExportException("remote object implements illegal remote interface", var7);
        }

        if (stub instanceof RemoteStub) {
            this.setSkeleton(impl);
        }

        Target target = new Target(impl, this, stub, this.ref.getObjID(), permanent);
        this.ref.exportObject(target);
        this.hashToMethod_Map = (Map)hashToMethod_Maps.get(implClass);
        return stub;
    }

    public String getClientHost() throws ServerNotActiveException {···

    public void setSkeleton(Remote impl) throws RemoteException {
        if (!withoutSkeletons.containsKey(impl.getClass())) {
            try {
                this.skel = Util.createSkeleton(impl);
            } catch (SkeletonNotFoundException var3) {
                withoutSkeletons.put(impl.getClass(), (Object)null);
            }
        }

    }
```

client stub

server stub (skeleton)

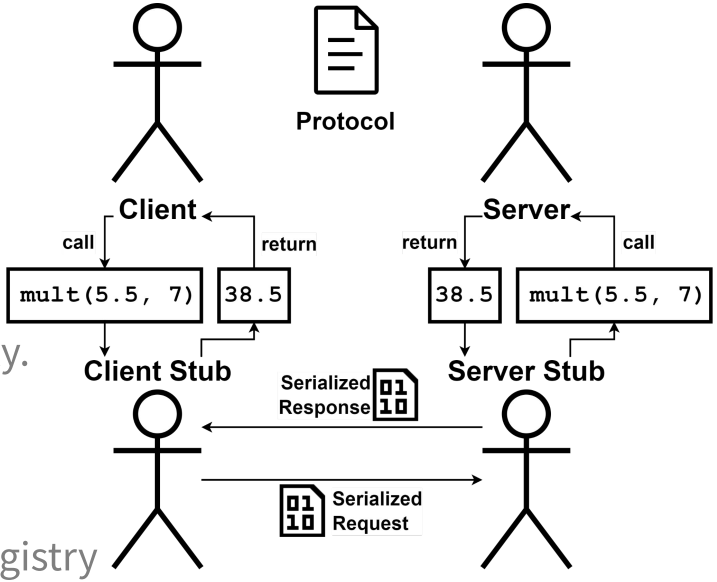
gRPC Compared to Java RMI

gRPC seems to be currently far more favorable than Java RMI, mainly because:

- **Cross-Language Support**:
  - Java RMI is Java-specific, but gRPC supports communication across different programming languages.
- **Serialization**:
  - Java Object Serialization is slower and more resource-intensive (due to its general-purpose design that records lots of object metadata) than Protobuf for gRPC.
- **Proxy Approach**:
  - Dynamic proxy from Java RMI reduces the codebase complexity, but at the same time introduces stub generation overhead at runtime, making the execution slightly slower than gRPC.

Summary

- Remote Procedure Call (RPC)
    a. **Principles**:
        i. serve remotely
        ii. invoke as if it were local
        iii. abstract remote interaction details
    b. **Design Concerns**: Protocol, Communication, Proxy.
- Java RMI (Sun Microsystems, 1997)
    a. Protocol: Java Interface
    b. Communication: Object Serialization + TCP
    c. Proxy: Dynamic Proxy via Java Reflection + RMI Registry
- **gRPC (Google, 2015)**
    a. Protocol: Proto files from Protobuf
    b. Communication: Protobuf + HTTP/2
    c. Proxy: `protoc`
- gRPC Benefits: auto codegen; efficient Protobuf; …



More details on gRPC in the next lab session…