DISTRIBUTED AND CLOUD COMPUTING

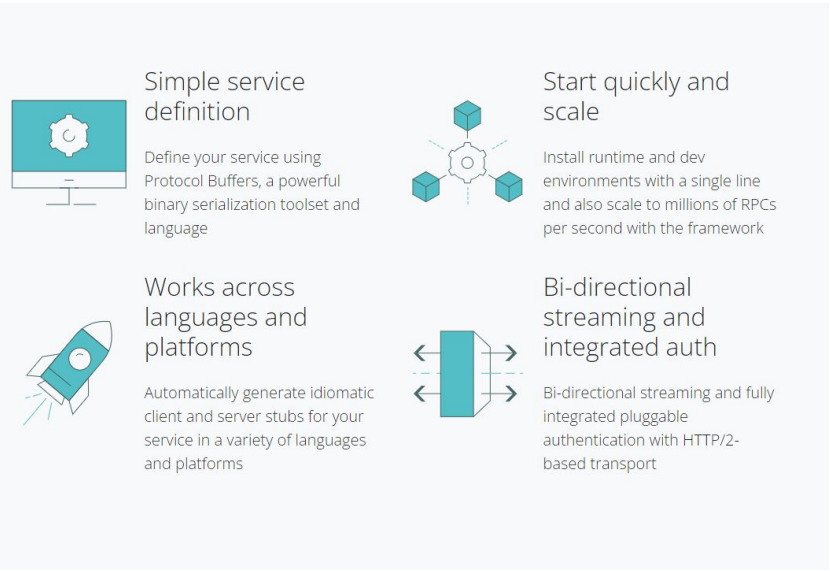
LAB 6: PROTOBUF + OTHER GRPC FEATURES

(Module: RPC & RESTFUL API)

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. …



https://grpc.io/

Serialization

- “Serialization is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later.”
- Text-based:
  a. **JSON**: key-value-based; commonly used in RESTful API
  b. XML: tag-based; commonly used in SOAP

```json
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": [
              "GML",
              "XML"
            ]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

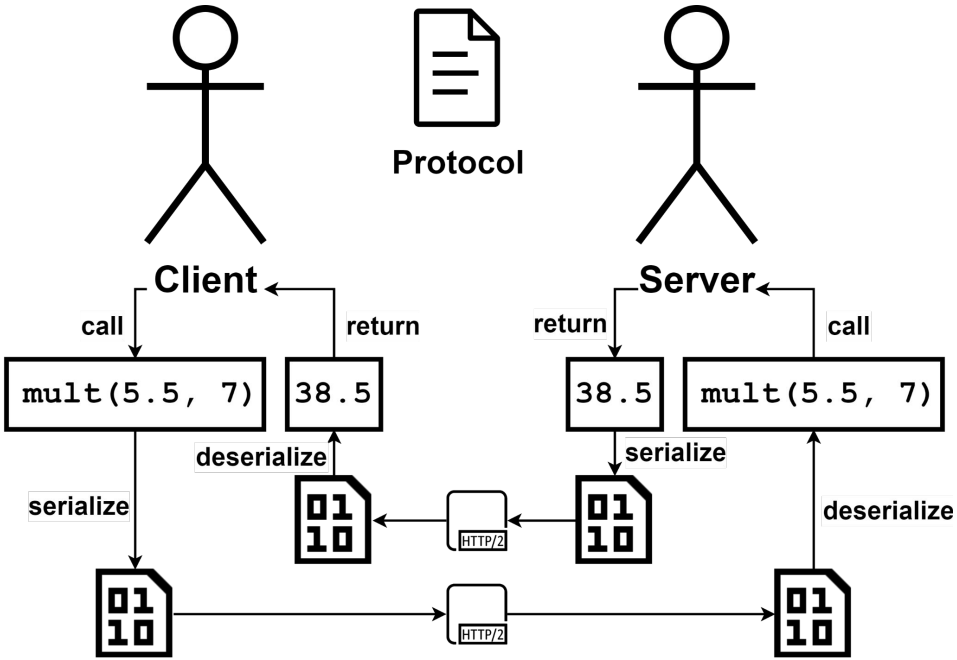
```xml
<!DOCTYPE glossary PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
<glossary>
  <title>example glossary</title>
  <GlossDiv>
    <title>S</title>
    <GlossList>
      <GlossEntry ID="SGML" SortAs="SGML">
        <GlossTerm>Standard Generalized Markup Language</GlossTerm>
        <Acronym>SGML</Acronym>
        <Abbrev>ISO 8879:1986</Abbrev>
        <GlossDef>
          <para>A meta-markup language, used to create markup
            languages such as DocBook.</para>
          <GlossSeeAlso OtherTerm="GML">
          <GlossSeeAlso OtherTerm="XML">
        </GlossDef>
        <GlossSee OtherTerm="markup">
      </GlossEntry>
    </GlossList>
  </GlossDiv>
</glossary>
```

https://json.org/example.html

Serialization

- “Serialization is the process of translating a data structure or object state into a format that can be stored or transmitted and reconstructed later.”
- Text-based:
  a. **JSON**
  b. XML
- Binary-based:
  a. MessagePack
  b. Thrift
  c. Protobuf
  d. Avro

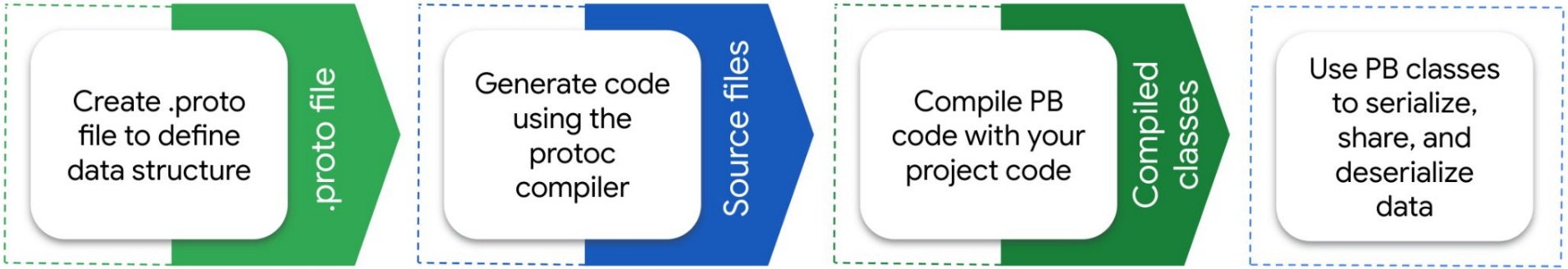
- How does Protobuf help gRPC boost data transmission?



Protocol Buffers (Protobuf)

- "Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data."
- Used by gRPC, Google Cloud Platform (GCP), Envoy Proxy, etc.
- Benefits:
  a. Compact data storage and fast parsing → how?
  b. Cross-Language support: proto files as IDL
  c. Automatic code generation: `protoc`

(already covered)



https://protobuf.dev/overview/

Protobuf - Encoding

- JSON → Binary (MessagePack)
  a. numeric values like 1337
     - 4B → 2B
  b. { [ ] } + : + , + " "
     - 22B → 0B
  c. object/array type
     - 0B → 2B
  d. key/value type
     - 0B → 7B

- **81** - 2 - 22 + 2 + 7 = **66**

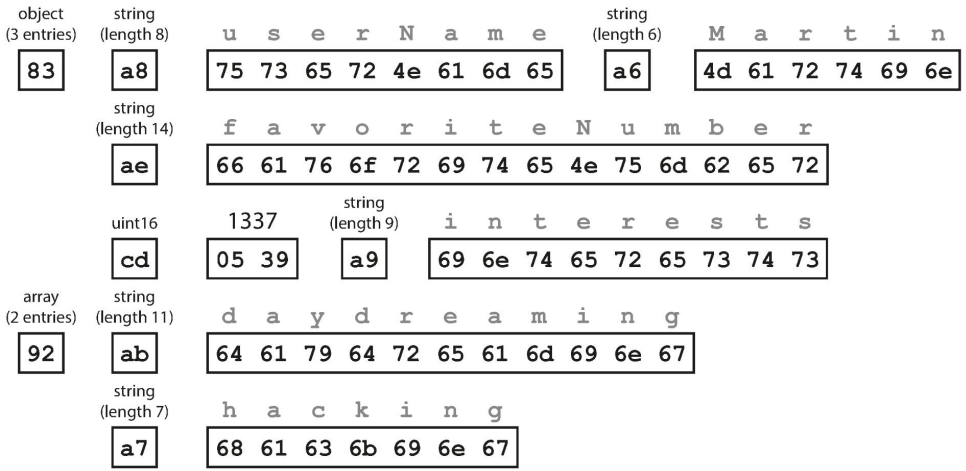
```
{                                                    (81B)
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

MessagePack

Byte sequence (66 bytes):

```
83 a8 75 73 65 72 4e 61 6d 65 a6 4d 61 72 74 69 6e ae 66 61
76 6f 72 69 74 65 4e 75 6d 62 65 72 cd 05 39 a9 69 6e 74 65
72 65 73 74 73 92 ab 64 61 79 64 72 65 61 6d 69 6e 67 a7 68
61 63 6b 69 6e 67
```

Breakdown:



https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html

Protobuf - Encoding

- MessagePack → Thrift Binary
  a. ignored keys
     - 31B → 0B (keys)
     - 9B → 20B (types)
  b. `uint16` → `i64`
     - 2B → 8B
  c. field tags
     - 0B → 6B
  d. end of struct
     - 0B → 1B
- **66** - 31 + 11 + 6 + 6 + 1 = **59**

```
struct Person {                    (Thrift IDL)
  1: required string       userName,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```

Thrift BinaryProtocol

Byte sequence (59 bytes):

```
0b 00 01 00 00 00 06 4d 61 72 74 69 6e 0a 00 02 00 00 00 00
00 00 05 39 0f 00 03 0b 00 00 00 02 00 00 00 0b 64 61 79 64
72 65 61 6d 69 6e 67 00 00 00 07 68 61 63 6b 69 6e 67 00
```

Breakdown:

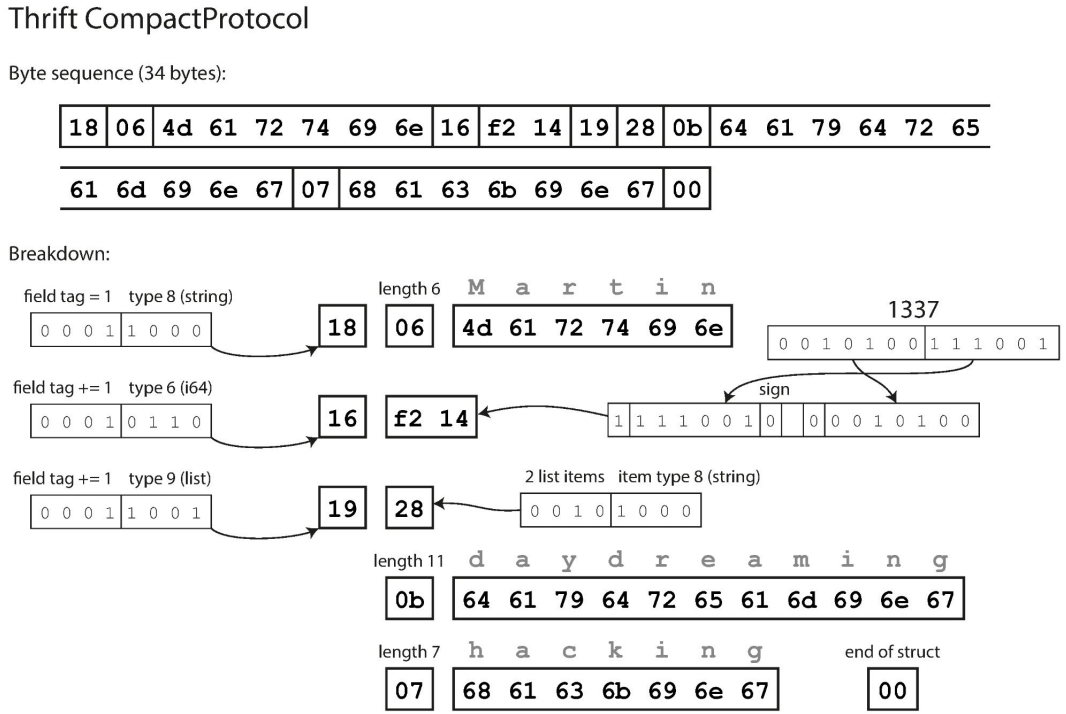
type 11 (string)	field tag = 1	length 6	M a r t i n
0b	00 01	00 00 00 06	4d 61 72 74 69 6e
type 10 (i64)	field tag = 2	1337	
0a	00 02	00 00 00 00 00 00 05 39	
type 15 (list)	field tag = 3	item type 11 (string)	2 list items
0f	00 03	0b	00 00 00 02
		length 11	d a y d r e a m i n g
		00 00 00 0b	64 61 79 64 72 65 61 6d 69 6e 67
		length 7	h a c k i n g end of struct
		00 00 00 07	68 61 63 6b 69 6e 67 00

https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html

Protobuf - Encoding

- Thrift Binary → Thrift Compact
  a. field tag + type
     - 9B → 3B (x3: 3B → 1B)
  b. `i64` → `varint`
     - Little-endian
     - MSB - continuation bit
     - LSB (1st byte) - sign
     - 8B → 2B
  c. string/array length → `varint`
     - 17B → 4B
- **59** - 6 - 6 - 13 = **34**

```
struct Person {                    (Thrift IDL)
  1: required string       userName,
  2: optional i64          favoriteNumber,
  3: optional list<string> interests
}
```



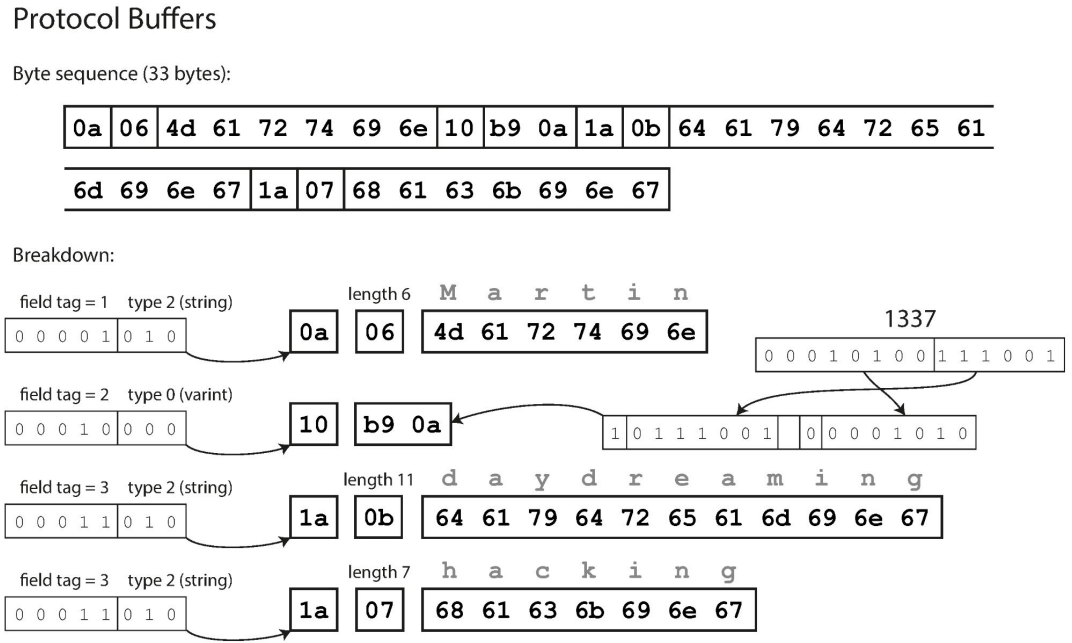
https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html

Protobuf - Encoding

- Thrift Compact → **Protobuf**
  a. field tag + type
     - 3B → 4B (array item)
  b. `varint`
     - Sign: ZigZag encoding
  c. end of struct
     - 1B → 0B
  d. array length
     - 1B → 0B

- **34** + 1 - 1 - 1 = **33** **Here we are!**

```
message Person {                                    (Protobuf IDL)
    required string user_name        = 1;
    optional int64  favorite_number = 2;
    repeated string interests        = 3;
}
```



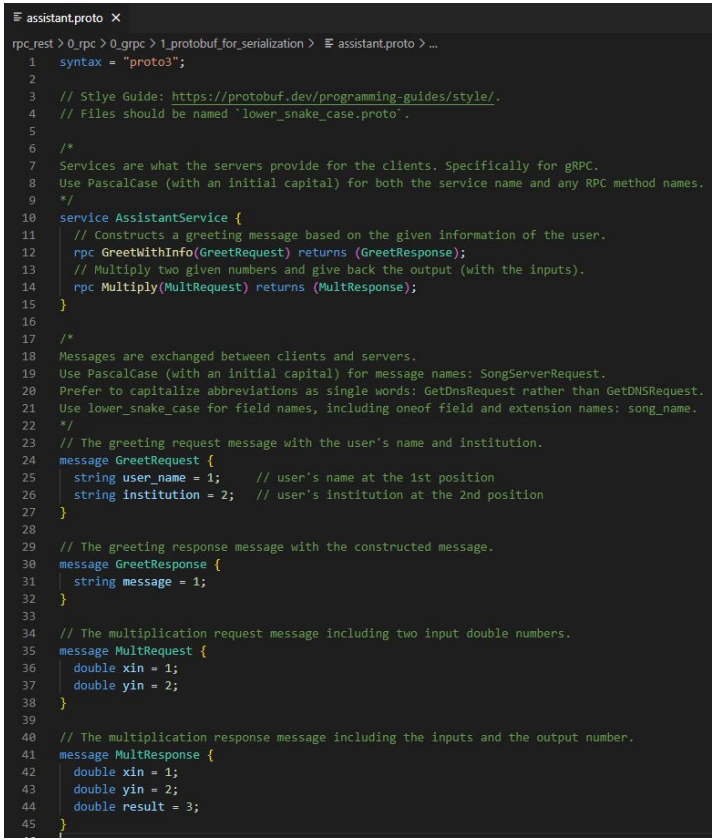
https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html

TASK: Protobuf - Exploration

Serialize protobuf message and compare with JSON.

> Reference codebase: `rpc_protobuf`

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=. assistant.proto`
   - Note that this time we only generate message classes.
4. Run `python x_main_reduced.py` to explore the behavior of Protobuf.


assistant.proto

rpc_rest > 0_rpc > 0_grpc > 1_protobuf_for_serialization > 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;
}
```

TASK: Protobuf - Exploration

Serialize protobuf message and compare with JSON.

> Reference codebase: `rpc_protobuf`

- `serialize_and_deserialize()`:
  a. Protobuf supports easy serialization and deserialization functions to implement our own gRPC, if necessary.
     i. `SerializeToString()`
     ii. `ParseFromString(binary_req)`

x_main_reduced.py

rpc_rest > 0_rpc > 0_grpc > 1_protobuf_for_serialization > x_main_reduced.py > ...

```python
def serialize_and_deserialize():
    init_msg = assistant_pb2.GreetRequest(user_name='Peter', institution='SUSTech')
    # init_msg = assistant_pb2.MultResponse(xin=3.5, yin=7, result=3.5*7)
    print(f'> Initial Message:\n{init_msg}')
    # serialize
    binary_req = init_msg.SerializeToString()
    print(f'> After Serialization: {binary_req}')
    print('>> Binary form:', ' '.join(f'{byte:08b}' for byte in binary_req))
    # check in hex string
    hex_req_str = binary_req.hex()
    beautify_hex_str = ' '.join(hex_req_str[i:i+2] for i in range(0, len(hex_req_str), 2))
    print(f'>> Hex Representation: {beautify_hex_str}')
    print('>>> Trying to decode the serialized message...')
    try_decode_proto_binary(binary_req)
    # deserialize
    recoverd_msg = assistant_pb2.GreetRequest() # empty initialization
    # recoverd_msg = assistant_pb2.MultResponse() # empty initialization
    recoverd_msg.ParseFromString(binary_req)
    print(f'> Deserialized Message:\n{recoverd_msg}')
```

Recall how gRPC uses Protobuf.

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)
```

```
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 00010010 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
>>> Trying to decode the serialized message...
----------------------------------------------------------------------
>>> Record: {'field_number': 1, 'wire_type': 2, 'wire_type_name': 'LEN', 'length': 5, 'payload': 'Peter'}
>>> Record: {'field_number': 2, 'wire_type': 2, 'wire_type_name': 'LEN', 'length': 7, 'payload': 'SUSTech'}
>>> Final Result:
{1: 'Peter', 2: 'SUSTech'}
----------------------------------------------------------------------
> Deserialized Message:
user_name: "Peter"
institution: "SUSTech"
```

TASK: Protobuf - Exploration

Serialize protobuf message and compare with JSON.

> Reference codebase: `rpc_protobuf`

- `serialize_and_deserialize()`:
  a. Protobuf supports easy serialization and deserialization functions to implement our own gRPC, if necessary.
  b. Protobuf uses a specific binary encoding scheme - explore via implementing: `try_decode_proto_binary(binary_str)`

x_main_reduced.py

rpc_rest > 0_rpc > 0_grpc > 1_protobuf_for_serialization > x_main_reduced.py > ...

```
28  def try_decode_proto_binary(binary_str):
66      # for this demo we only implement LEN decoding for string
67      if record['wire_type'] == 2:
68        # WIRE: https://protobuf.dev/programming-guides/encoding/#structure
69        record['wire_type_name'] = 'LEN'
70        """
71        https://protobuf.dev/programming-guides/encoding/#length-types
72        "The LEN wire type has a dynamic length,
73        specified by a varint immediately after the tag,
74        which is followed by the payload as usual."
75        """
76        # check length (NOTE: we are lazy so we just grab the next byte)
77        record['length'] = int(binary_str[ptr:ptr+8], 2)
78        ptr += 8
79        # check payload (the next `length` bytes are UTF-8 encoded chars)
80        payload_str = f'0b{binary_str[ptr:ptr+8*record['length']]}'
81        ptr += 8*record['length']
82        record['payload'] = binascii.unhexlify('%x' % int(payload_str, 2)).decode('utf-8')
83        result[record['field_number']] = record['payload']
84      print(f'>>> Record: {record}')
```

MINI-TASK:

Try to manually implement the decoding logic for `MultResponse`.

```
>>> Trying to decode the serialized message...
--------------------------------------------------------------------------
>>> Record: {'field_number': 1, 'wire_type': 1, 'wire_type_name': 'I64', 'payload': 3.5}
>>> Record: {'field_number': 2, 'wire_type': 1, 'wire_type_name': 'I64', 'payload': 7.0}
>>> Record: {'field_number': 3, 'wire_type': 1, 'wire_type_name': 'I64', 'payload': 24.5}
>>> Final Result:
{1: 3.5, 2: 7.0, 3: 24.5}
--------------------------------------------------------------------------
```

```
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 00010010 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
>>> Trying to decode the serialized message...
--------------------------------------------------------------------------
>>> Record: {'field_number': 1, 'wire_type': 2, 'wire_type_name': 'LEN', 'length': 5, 'payload': 'Peter'}
>>> Record: {'field_number': 2, 'wire_type': 2, 'wire_type_name': 'LEN', 'length': 7, 'payload': 'SUSTech'}
>>> Final Result:
{1: 'Peter', 2: 'SUSTech'}
--------------------------------------------------------------------------
> Deserialized Message:
user_name: "Peter"
institution: "SUSTech"
```

TASK: Protobuf - Exploration

Serialize protobuf message and compare with JSON.

> Reference codebase: `rpc_protobuf`

- `serialize_and_deserialize()`:
  a. Protobuf supports easy serialization and deserialization functions to implement our own gRPC, if necessary.
  b. Protobuf uses a specific binary encoding scheme - explore via implementing: `try_decode_proto_binary(binary_str)`
- `compare_with_json()`:
  a. Protobuf provides much lighter payload (16B) compared to JSON (48B).

x_main_reduced.py

rpc_rest > 0_rpc > 0_grpc > 1_protobuf_for_serialization > x_main_reduced.py > serialize_and_deserialize

```
89  def compare_with_json():
90    # JSON serialization
91    json_req = {
92      'user_name': 'Peter',
93      'institution': 'SUSTech',
94    }
95    ser_json = json.dumps(json_req).encode('utf-8')
96    print(f'> JSON serialized into {len(ser_json)} bytes: {ser_json}')
97    # serialization via Protobuf
98    proto_req = assistant_pb2.GreetRequest(**json_req)
99    binary_req = proto_req.SerializeToString()
100   print(f'> Protobuf serialized into {len(binary_req)} bytes: {binary_req}')
```

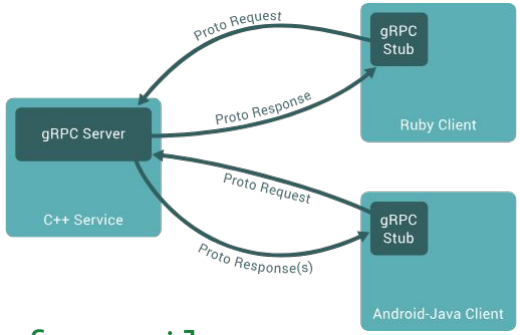
```
compare_with_json
> JSON serialized into 48 bytes: b'{"user_name": "Peter", "institution": "SUSTech"}'
> Protobuf serialized into 16 bytes: b'\n\x05Peter\x12\x07SUSTech'
```

If the “go install …” commands timeout you can try running: **export GOPROXY=https://goproxy.cn,direct**

TASK: gRPC - A Cross-Language Example

A Go gRPC client + a Python gRPC server.

> Reference codebase: `rpc_grpc_cross_language`



1. Set up Python (Miniconda is recommended).
2. Install Python dependencies into a Conda environment via:
   - `python -m pip install -r requirements.txt`
3. Set up Go. Install Go dependencies:
   - `sudo apt update; sudo apt install -y protobuf-compiler`
   - `go install google.golang.org/protobuf/cmd/protoc-gen-go@latest`
   - `go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest`
   - `export PATH="$PATH:$(go env GOPATH)/bin"`
4. Check the protocol file `assistant.proto`. **Note that the `go_package` option is set to properly generate code to the `pb` package.**

This command is only active for the current session. If you restart the terminal you need to run this again! To permanently add go to path you can add this command to the profile (~/.bashrc for bash)

assistant.proto

rpc_rest > 0_rpc > 0_grpc > 2_cross_language > assistant.proto > ...

```
6   // [START go_declaration] (https://protobuf.dev/getting-started/gotutorial/#protocol-format)
7   option go_package = "./pb";
8   // [END go_declaration]
```

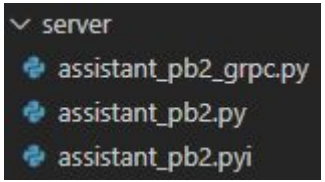
https://grpc.io/docs/what-is-grpc/introduction/

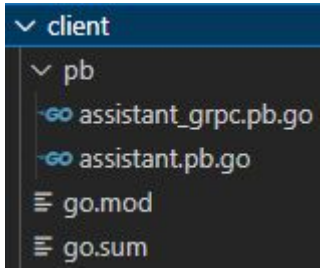
TASK: gRPC - A Cross-Language Example

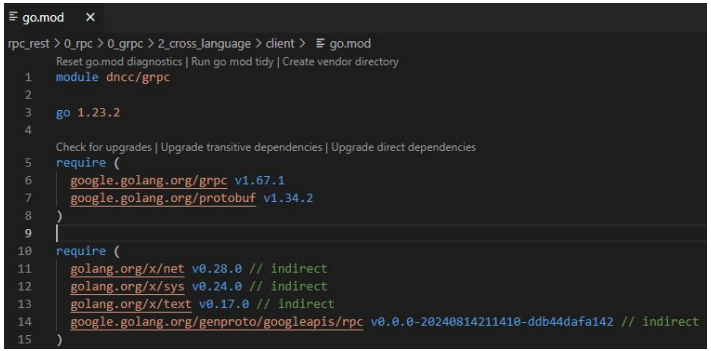
A Go gRPC client + a Python gRPC server.

> Reference codebase: `rpc_grpc_cross_language`

5. Use `protoc` to generate code for Python and Go:
   - `python -m grpc_tools.protoc -I./ --python_out=./server/ --pyi_out=./server/ --grpc_python_out=./server/ assistant.proto`
   - `protoc -I=./ --go_out=./client/ --go-grpc_out=./client/ assistant.proto`
   - After generating code for Go, run `go mod tidy` in the `client/` directory. The Go dependencies are automatically recorded in the `go.mod` file.







TASK: gRPC - A Cross-Language Example

A Go gRPC client + a Python gRPC server.

> Reference codebase: `rpc_grpc_cross_language`

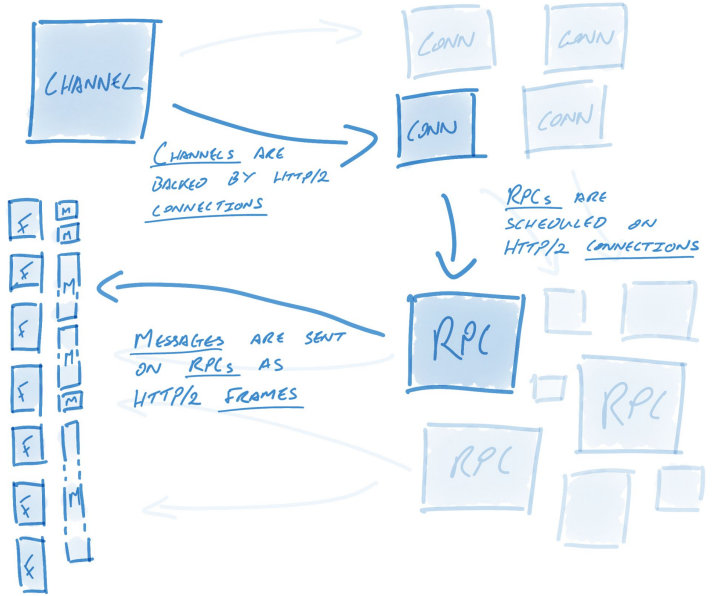
5. Implement the gRPC server in `hw_server.py`. Run the gRPC server via:
   - `python server/hw_server.py`
6. Implement the gRPC client in `hw_client.go`. Run `go mod tidy` again in the `client/` directory to update dependencies. Then, in another terminal, run the gRPC client via:
   - `cd client/; go run hw_client.go`

```
# python server/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 checked-in Protobuf gencode that can be obsolete.
  warnings.warn(
INFO:root:Server started, listening on 8082
```

```
# go run hw_client.go
> 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
```

gRPC over HTTP/2

- 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.



Request

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /google.pubsub.v2.PublisherService/CreateTopic
:authority = pubsub.googleapis.com
grpc-timeout = 1S
content-type = application/grpc+proto
grpc-encoding = gzip
authorization = Bearer y235.wef315yfh138vh31hv93hv8h3v

DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

Response

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
trace-proto-bin = jher831yy13JHy3hc
```

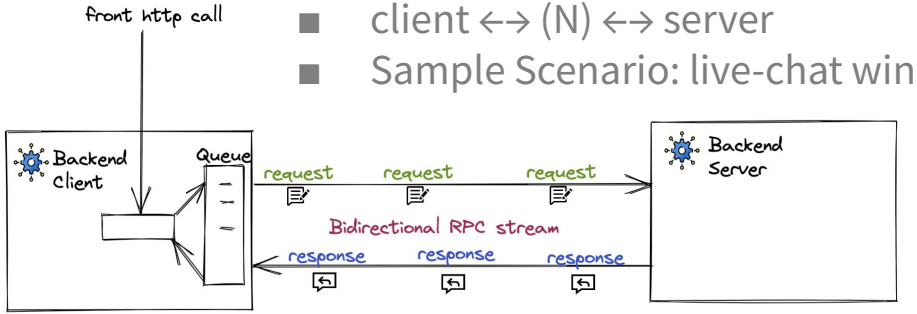
This is still a unary-call example.
Now we move to streaming!

```
service AssistantService {
  // Tells a story by streaming the story text.
  rpc TellStory(TellStoryRequest) returns (stream TellStoryResponse);
}
```

https://grpc.io/blog/grpc-on-http2/
https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md#example

gRPC - Streaming

- Unary RPC Call: client → (1 request message) → server → (1 response message) → client
- **Streaming**: the client/server may send a stream of multiple messages to the server/client.
  a. Feature: multiple messages within 1 single RPC call!
  b. **Server-side Streaming**:
     - client → (1) → server → (N) → client
     - Sample Scenario: subscribing to a Kafka topic; retrieving a large file by chunks.
  c. **Client-side Streaming**:
     - client → (N) → server → (1) → client
     - Sample Scenario: uploading a large file by chunks; data batch processing.
  d. **Bidirectional Streaming**:
     - client ↔ (N) ↔ server
     - Sample Scenario: live-chat windows; real-time gaming data exchange.



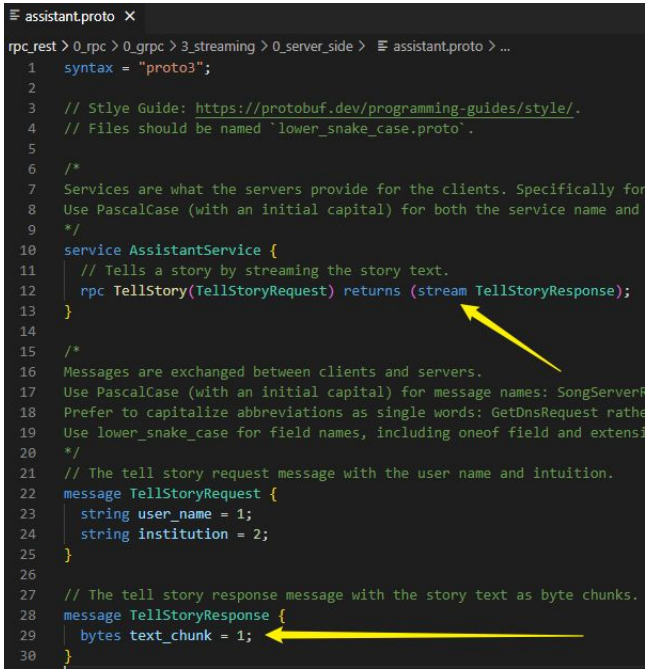
https://stackoverflow.com/questions/74265569/grpc-bidirectional-streaming-versus-back-to-back-http-call

TASK: Server-side Streaming

Stream a story to my console.

> Reference codebase: `rpc_grpc_streaming_server_side`

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



TASK: Server-side Streaming

Stream a story to my console.

> Reference codebase: `rpc_grpc_streaming_server_side`

server.py

rpc_rest > 0_rpc > 0_grpc > 3_streaming > 0_server_side > server.py > Assistant > TellStory

```python
class Assistant(AssistantServiceServicer):
    def __init__(self) -> None:
        super().__init__()
        self.encoding = 'utf-8'      # encoding format of the story text into bytes
        self.story_fn = 'alice.txt'  # where to read the story (to simulate story generation)
        self.buffer_size = 1024      # buffer size in bytes when reading the story file
        self.peek_size = 10          # buffer size in bytes for logging purpose

    def TellStory(self, request: TellStoryRequest, context: grpc.RpcContext):
        # Send back a greeting message
        logging.info(f'Received story-telling request from {request.user_name}, {request.institution}.')
        greeting = f'Hi {request.user_name} from {request.institution}! Here\'s your story:\n\n'
        yield TellStoryResponse(text_chunk=greeting.encode(self.encoding))
        # simulating story generation time
        time.sleep(3)
        # read the file with a buffer size and send the buffer data back
        logging.info(f'Start streaming {self.story_fn}')
        with open(self.story_fn, 'rb') as f:
            while True:
                # read a text chunk as a data buffer
                chunk = f.read(self.buffer_size)
                if not chunk: # EOF
                    logging.info('[EOF]')
                    break
                logging.info('Read %d bytes: %s ...' % (self.buffer_size, chunk[:self.peek_size].decode(self.encoding)))
                yield TellStoryResponse(text_chunk=chunk)
                time.sleep(0.05)
        # ending note
        ending = '\nThat\'s it! Hope you like the story~\n'
        yield TellStoryResponse(text_chunk=ending.encode(self.encoding))
        logging.info(f'Finishes streaming {self.story_fn} for {request.user_name} from {request.institution}.')
        # finsh the RPC call via cancellation
        context.cancel()
```

alice.txt

rpc_rest > 0_rpc > 0_grpc > 3_streaming > 0_server_side > alice.txt

```
[Alice's Adventures in Wonderland by Lewis Carroll 1865]

CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
it, 'and what is the use of a book,' thought Alice 'without pictures or
conversation?'

So she was considering in her own mind (as well as she could, for the
hot day made her feel very sleepy and stupid), whether the pleasure
of making a daisy-chain would be worth the trouble of getting up and
picking the daisies, when suddenly a White Rabbit with pink eyes ran
close by her.

There was nothing so VERY remarkable in that; nor did Alice think it so
VERY much out of the way to hear the Rabbit say to itself, 'Oh dear!
Oh dear! I shall be late!' (when she thought it over afterwards, it
occurred to her that she ought to have wondered at this, but at the time
it all seemed quite natural); but when the Rabbit actually TOOK A WATCH
OUT OF ITS WAISTCOAT-POCKET, and looked at it, and then hurried on,
Alice started to her feet, for it flashed across her mind that she had
never before seen a rabbit with either a waistcoat-pocket, or a watch
to take out of it, and burning with curiosity, she ran across the field
after it, and fortunately was just in time to see it pop down a large
rabbit-hole under the hedge.

In another moment down went Alice after it, never once considering how
in the world she was to get out again.

The rabbit-hole went straight on like a tunnel for some way, and then
dipped suddenly down, so suddenly that Alice had not a moment to think
about stopping herself before she found herself falling down a very deep
well.
```

alice.txt

rpc_rest > 0_rpc > 0_grpc > 3_streaming > 0_server_side > alice.txt

```
with the dream of Wonderland of long ago: and how she would feel with
all their simple sorrows, and find a pleasure in all their simple joys,
remembering her own child-life, and the happy summer days.
```

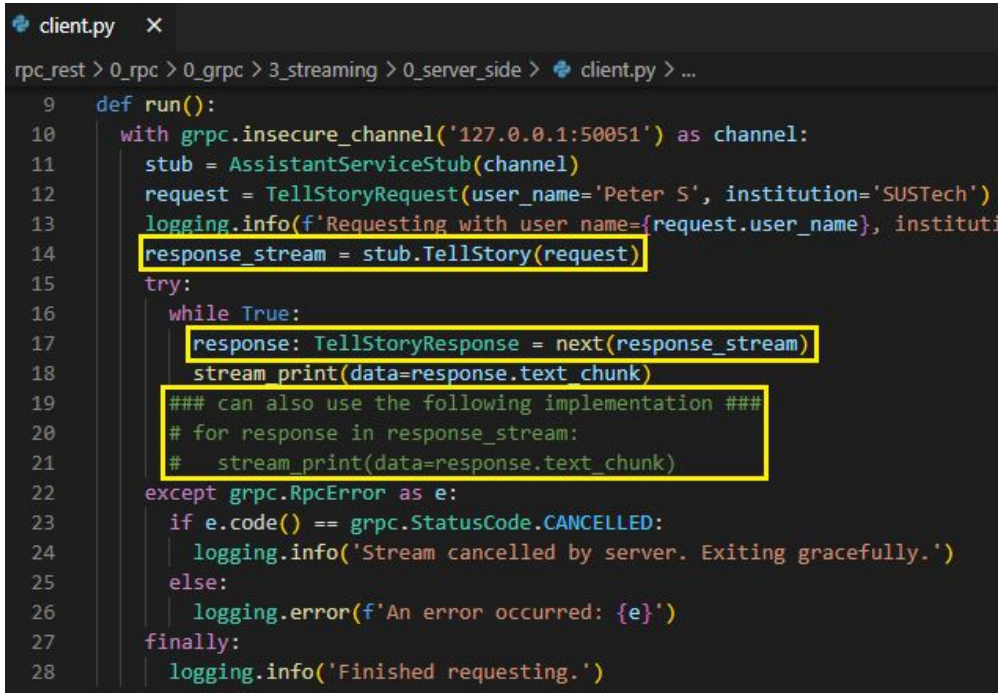
TASK: Server-side Streaming

Stream a story to my console.

> Reference codebase: `rpc_grpc_streaming_server_side`

- Server:
  - `yield`
  - “generate” responses
- Client:
  - `next` or `for`
  - Response stream “generator”

- Note how the server and client handle graceful shutdown in the streaming case:
  a. Cancellation
  b. gRPC status code



TASK: Server-side Streaming

Stream a story to my console.

> Reference codebase: `rpc_grpc_streaming_server_side`

```
# python client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/r
rWarning: Protobuf gencode version 5.27.2 is older than the runtime versi
to. Please avoid checked-in Protobuf gencode that can be obsolete.
  warnings.warn(
INFO:root:Requesting with user_name=Peter S, institution=SUSTech
Hi Peter S from SUSTech! Here's your story:

[Alice's Adventures in Wonderland by Lewis Carroll 1865]

CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
it, 'and what is the use of a book,' thought Alice 'without pictures or
conversation?'

So she was considering in her own mind (as well as she could, for the
hot day made her feel very sleepy and stupid), whether the pleasure
of making a daisy-chain would be worth the trouble of getting up and
```

```
childhood: and how she would gather about her other little children, and
make THEIR eyes bright and eager with many a strange tale, perhaps even
with the dream of Wonderland of long ago: and how she would feel with
all their simple sorrows, and find a pleasure in all their simple joys,
remembering her own child-life, and the happy summer days.

That's it! Hope you like the story~
INFO:root:Stream cancelled by server. Exiting gracefully.
INFO:root:Finished requesting.
```

```
# python server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/pr
rWarning: Protobuf gencode version 5.27.2 is older than the runti
to. Please avoid checked-in Protobuf gencode that can be obsolet
  warnings.warn(
INFO:root:Server started on port 50051
INFO:root:Received story-telling request from Peter S, SUSTech.
INFO:root:Start streaming alice.txt
INFO:root:Read 1024 bytes: [Alice's A ...
INFO:root:Read 1024 bytes: y TOOK A W ...
INFO:root:Read 1024 bytes: of the wel ...
INFO:root:Read 1024 bytes: s
was not  ...
INFO:root:Read 1024 bytes: somewhere. ...
INFO:root:Read 1024 bytes:  she jumpe ...
INFO:root:Read 1024 bytes:  too large ...
INFO:root:Read 1024 bytes:
impossibl ...
INFO:root:Read 1024 bytes: f you drin ...
INFO:root:Read 1024 bytes: fancy what ...
INFO:root:Read 1024 bytes: f croquet  ...
INFO:root:Read 1024 bytes: ngs to hap ...
INFO:root:Read 1024 bytes: rself how  ...
INFO:root:Read 1024 bytes: ll.

After ...
```

```
 ...
INFO:root:Read 1024 bytes: ttered dow ...
INFO:root:Read 1024 bytes: ttle toss  ...
INFO:root:Read 1024 bytes: y--the gra ...
INFO:root:Read 1024 bytes: mmer days. ...
INFO:root:[EOF]
INFO:root:Finishes streaming alice.txt for Peter S from SUSTech.
```

TASK: gRPC - Remote Counter

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

Requirements:

1. The client subscribes the counting service and logs the server response to the console.
2. The server streams the response messages.

```
# python client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-
ning: Protobuf gencode version 5.27.2 is older
se avoid checked-in Protobuf gencode that can b
  warnings.warn(
INFO:root:2024-10-23 04:55:14.621176: count=7
INFO:root:2024-10-23 04:55:16.621795: count=8
INFO:root:2024-10-23 04:55:18.622318: count=9
INFO:root:2024-10-23 04:55:20.622939: count=10
INFO:root:2024-10-23 04:55:22.623407: count=11
^CINFO:root:Stream cancelled by user (Ctrl+C).
```

```
# python server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/
ning: Protobuf gencode version 5.27.2 is older than the runtime
se avoid checked-in Protobuf gencode that can be obsolete.
  warnings.warn(
INFO:root:Server started on port 50051
INFO:root:Request: from=6, step=2
INFO:root:Response stream cancelled for request: from=6, step=2
```

local_procedure.py

rpc_rest > 0_rpc > 0_grpc > 3_streaming > 0x_remote_counter > task > local_procedure.py > count

```python
def count(from_val: int, step_sec: int) -> None:
    '''
    Progresses a counter from a starting value, adding 1 after every stepping seconds.
    '''
    logging.info(f'Request: from={from_val}, step={step_sec}')
    cnt = from_val
    try:
        while True:
            # pause for step_sec seconds
            time.sleep(step_sec)
            # increment counter by 1
            cnt += 1
            # log the counter update
            msg = f'{datetime.now()}: count={cnt}'
            logging.info(msg)
    except KeyboardInterrupt:
        # ctrl+c to stop
        logging.info('Process cancelled by user (Ctrl+C).')
    finally:
        logging.info(f'Process cancelled for request: from={from_val}, step={step_sec}')


if __name__ == '__main__':
    logging.basicConfig(level=logging.INFO)
    """
    INFO:root:Request: from=6, step=2
    INFO:root:2024-10-22 03:13:33.013209: count=7
    INFO:root:2024-10-22 03:13:35.012208: count=8
    INFO:root:2024-10-22 03:13:37.012531: count=9
    INFO:root:2024-10-22 03:13:39.012810: count=10
    INFO:root:2024-10-22 03:13:41.013144: count=11
    ...
    INFO:root:Process cancelled by user (Ctrl+C).
    INFO:root:Process cancelled for request: from=6, step=2
    """
    count(from_val=6, step_sec=2)
```

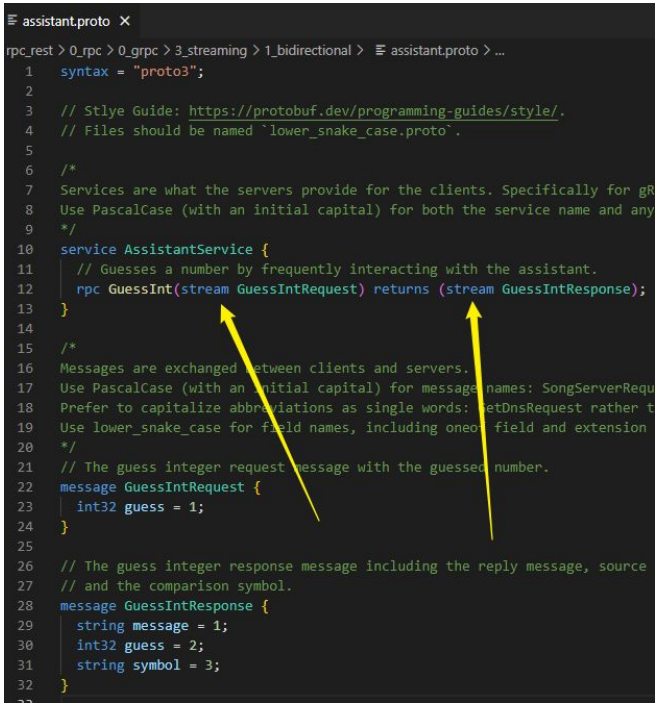
Try starting from scratch to be familiar with the process before Assignment 2.

TASK: Bidirectional Streaming

Real-time number guessing from the server.

> Reference codebase: `rpc_grpc_streaming_bi`

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



TASK: Bidirectional Streaming

Real-time number guessing from the server.

> Reference codebase: `rpc_grpc_streaming_bi`

- Manual guessing mode

```
python client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/g
ning: Protobuf gencode version 5.27.2 is older than the r
se avoid checked-in Protobuf gencode that can be obsolete
  warnings.warn(
Guessing start! Try an integer between 1 and 100.
Guess: 50
Your guess 50 < true value.
Guess: 75
Your guess 75 < true value.
Guess: 81
Your guess 81 > true value.
Guess: xx
ERROR:root:Invalid input xx. Please enter a valid number.
Guess: 80
Your guess 80 > true value.
Guess: 77
Your guess 77 < true value.
Guess: 78
Your guess 78 < true value.
Guess: 79
Your guess 79 = true value. Congratulations!!!
```

```
python server.py
/root/miniconda3/envs/dncc/lib/python3.12/sit
ning: Protobuf gencode version 5.27.2 is olde
se avoid checked-in Protobuf gencode that can
  warnings.warn(
/home/rainbow/asialab/dncc/dncc-lab/rpc_rest/
recationWarning: There is no current event lo
  loop = asyncio.get_event_loop()
INFO:root:Server started on port 50051
INFO:root:Received number guessing request.
INFO:root:True value is 79.
INFO:root:User guessed 50 < 79 (true value).
INFO:root:User guessed 75 < 79 (true value).
INFO:root:User guessed 81 > 79 (true value).
INFO:root:User guessed 80 > 79 (true value).
INFO:root:User guessed 77 < 79 (true value).
INFO:root:User guessed 78 < 79 (true value).
INFO:root:User guessed 79 = 79 (true value).
INFO:root:Guessing service finished.
```

TASK: Bidirectional Streaming

Real-time number guessing from the server.

> Reference codebase: `rpc_grpc_streaming_bi`

- Auto-guessing mode

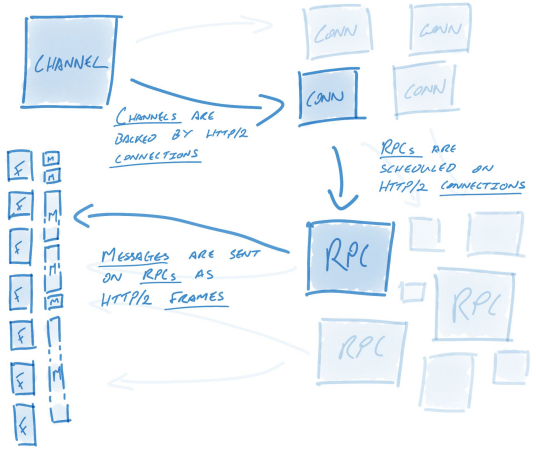
```
python client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-
ning: Protobuf gencode version 5.27.2 is older
se avoid checked-in Protobuf gencode that can b
  warnings.warn(
Guessing start! Try an integer between 1 and 10
INFO:root:Init guess: go for 50
Your guess 50 > true value.
INFO:root:Guessing: 25
Your guess 25 < true value.
INFO:root:Guessing: 37
Your guess 37 > true value.
INFO:root:Guessing: 31
Your guess 31 < true value.
INFO:root:Guessing: 34
Your guess 34 < true value.
INFO:root:Guessing: 35
Your guess 35 < true value.
INFO:root:Guessing: 36
Your guess 36 = true value. Congratulations!!!
```

```
python server.py
/root/miniconda3/envs/dncc/lib/python3.12/sit
ning: Protobuf gencode version 5.27.2 is olde
se avoid checked-in Protobuf gencode that can
  warnings.warn(
/home/rainbow/asialab/dncc/dncc-lab/rpc_rest/
recationWarning: There is no current event lo
  loop = asyncio.get_event_loop()
INFO:root:Server started on port 50051
INFO:root:Received number guessing request.
INFO:root:True value is 36.
INFO:root:User guessed 50 > 36 (true value).
INFO:root:User guessed 25 < 36 (true value).
INFO:root:User guessed 37 > 36 (true value).
INFO:root:User guessed 31 < 36 (true value).
INFO:root:User guessed 34 < 36 (true value).
INFO:root:User guessed 35 < 36 (true value).
INFO:root:User guessed 36 = 36 (true value).
INFO:root:Guessing service finished.
```

gRPC - Multiplexing

- Multiplexing: multiple requests from different RPC procedures handled concurrently over 1 single HTTP/2 connection.
  a. The request messages are immediately sent to the server.
  b. The server manages a thread pool that is able to summon a limited number of workers to handle incoming requests concurrently.
     - enough workers available → all request messages handled in parallel
     - otherwise → some request messages are queued at the server for later
  c. Everything is managed within one “channel” + multiple logical “streams”!

- Multiplexing and Streaming focus on different aspects:
  a. **Streaming**: same RPC, multiple messages - 1 RPC call
  b. **Multiplexing**: different RPCs - 1 connection



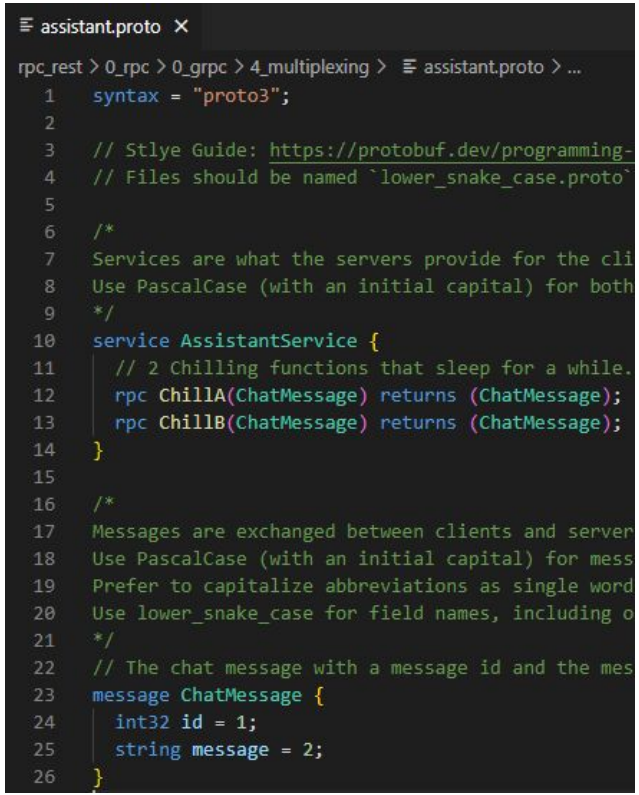
https://grpc.io/blog/grpc-on-http2/

TASK: Exploring Multiplexing

Pop up multiple long-time RPC calls simultaneously.

> Reference codebase: `rpc_grpc_multiplexing`

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`
4. Implement the gRPC server in `server.py`. Run the gRPC server via:
   - `python server.py`
5. Implement the gRPC client in `client.py`. In another terminal, run the gRPC client via:
   - `python client.py -a 1 -b 1`



TASK: Exploring Multiplexing

Pop up multiple long-time RPC calls simultaneously.

> Reference codebase: `rpc_grpc_multiplexing`

- `ChillA`
  - sleeps for 5 seconds.
- `ChillB`
  - sleeps for 3 seconds.

server.py

rpc_rest > 0_rpc > 0_grpc > 4_multiplexing > server.py > Assistant > ChillA

```python
class Assistant(AssistantServiceServicer):
    def ChillA(self, request: ChatMessage, context: grpc.RpcContext):
        logging.info(f'{datetime.now()}: {self.ChillA.__name__} {request.id} start')
        time.sleep(5) # sleep for 5 sec
        logging.info(f'{datetime.now()}: {self.ChillA.__name__} {request.id} end')
        return ChatMessage(id=request.id, message=request.message)

    def ChillB(self, request: ChatMessage, context: grpc.RpcContext):
        logging.info(f'{datetime.now()}: {self.ChillB.__name__} {request.id} start')
        time.sleep(3) # sleep for 5 sec
        logging.info(f'{datetime.now()}: {self.ChillB.__name__} {request.id} end')
        return ChatMessage(id=request.id, message=request.message)
```

client.py

rpc_rest > 0_rpc > 0_grpc > 4_multiplexing > client.py > ...

```python
def call_chill_a(stub: AssistantServiceStub, id: int):
    logging.info(f'{datetime.now()}: Calling ChillA {id}')
    start_t = time.time()
    _ = stub.ChillA(ChatMessage(id=id, message=MSG_DEFAULT))
    logging.info(f'{datetime.now()}: ChillA {id} finished after {tim

def call_chill_b(stub: AssistantServiceStub, id: int):
    logging.info(f'{datetime.now()}: Calling ChillB {id}')
    start_t = time.time()
    _ = stub.ChillB(ChatMessage(id=id, message=MSG_DEFAULT))
    logging.info(f'{datetime.now()}: ChillB {id} finished after {tim

def run(num_a_tasks: int, num_b_tasks: int):
    with grpc.insecure_channel('127.0.0.1:50051') as channel:
        stub = AssistantServiceStub(channel)
        start_time = time.time()
        with ThreadPoolExecutor() as executor:
            # execute concurrently
            futures_a = []
            for ai in range(num_a_tasks):
                futures_a.append(executor.submit(call_chill_a, stub, ai))
            futures_b = []
            for bi in range(num_b_tasks):
                futures_b.append(executor.submit(call_chill_b, stub, bi))
            # wait for all to finish
            for futures_x in [futures_a, futures_b]:
                for future_i in futures_x:
                    future_i.result()

    logging.info(f'Total execution time = {time.time() - start_time:
```

TASK: Exploring Multiplexing

Pop up multiple long-time RPC calls simultaneously.

> Reference codebase: `rpc_grpc_multiplexing`

- Enough workers available
  - `python client.py -a 1 -b 1`
  - `ChillA`
    - sleeps for 5 seconds
    - 1 call
  - `ChillB`
    - sleeps for 3 seconds
    - 1 call
  - Total execution time: ~5 seconds.

```
python server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packag
:112: UserWarning: Protobuf gencode version 5.27.2 is
2 at assistant.proto. Please avoid checked-in Protobu
  warnings.warn(
INFO:root:Server started on port 50051
INFO:root:2024-10-23 05:21:25.360505: ChillA 0 start
INFO:root:2024-10-23 05:21:25.360687: ChillB 0 start
INFO:root:2024-10-23 05:21:28.360830: ChillB 0 end
INFO:root:2024-10-23 05:21:30.360815: ChillA 0 end
```

```
python client.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/ru
:112: UserWarning: Protobuf gencode version 5.27.2 is older than the runti
2 at assistant.proto. Please avoid checked-in Protobuf gencode that can be
  warnings.warn(
INFO:root:2024-10-23 05:21:25.357470: Calling ChillA 0
INFO:root:2024-10-23 05:21:25.357746: Calling ChillB 0
INFO:root:2024-10-23 05:21:28.361504: ChillB 0 finished after 3.00 seconds
INFO:root:2024-10-23 05:21:30.361520: ChillA 0 finished after 5.00 seconds
INFO:root:Total execution time = 5.01 seconds.
```

TASK: Exploring Multiplexing

Pop up multiple long-time RPC calls simultaneously.

> Reference codebase: `rpc_grpc_multiplexing`

- Not enough workers
  - `python client.py -a 2 -b 2`
  - `ChillA`
    - sleeps for 5 seconds
    - 2 calls
  - `ChillB`
    - sleeps for 3 seconds
    - 2 calls
  - Total execution time might be: ~6 seconds (3+3 > 5).

```
# python server.py
/root/miniconda3/envs/dncc/lib/python3.12/site-packag
py:112: UserWarning: Protobuf gencode version 5.27.2
.28.2 at assistant.proto. Please avoid checked-in Pro
  warnings.warn(
INFO:root:Server started on port 50051
INFO:root:2024-10-23 05:33:29.415527: ChillA 1 start
INFO:root:2024-10-23 05:33:29.415827: ChillB 1 start
INFO:root:2024-10-23 05:33:29.415901: ChillA 0 start
INFO:root:2024-10-23 05:33:32.416143: ChillB 1 end
INFO:root:2024-10-23 05:33:32.416780: ChillB 0 start
INFO:root:2024-10-23 05:33:34.415931: ChillA 1 end
INFO:root:2024-10-23 05:33:34.416158: ChillA 0 end
INFO:root:2024-10-23 05:33:35.417026: ChillB 0 end
```

```
# python client.py -a 2 -b 2
/root/miniconda3/envs/dncc/lib/python3.12/site-packages/google/protobuf/ru
py:112: UserWarning: Protobuf gencode version 5.27.2 is older than the run
.28.2 at assistant.proto. Please avoid checked-in Protobuf gencode that ca
  warnings.warn(
INFO:root:2024-10-23 05:33:00.640993: Calling ChillA 0
INFO:root:2024-10-23 05:33:00.641458: Calling ChillA 1
INFO:root:2024-10-23 05:33:00.641752: Calling ChillB 0
INFO:root:2024-10-23 05:33:00.642332: Calling ChillB 1
INFO:root:2024-10-23 05:33:03.645509: ChillB 0 finished after 3.00 seconds
INFO:root:2024-10-23 05:33:05.645223: ChillA 1 finished after 5.00 seconds
INFO:root:2024-10-23 05:33:05.645622: ChillA 0 finished after 5.00 seconds
INFO:root:2024-10-23 05:33:06.646378: ChillB 1 finished after 6.00 seconds
INFO:root:Total execution time = 6.01 seconds.
```

Summary

- **Protobuf**
  a. Compact data storage and fast parsing: binary encoding with varint, etc.
  b. Cross-Language support: Proto files as IDL.
  c. Automatic code generation: `protoc`.
- **gRPC - Cross-Language Support**: thanks to Protobuf
- **gRPC - Streaming**
  a. Multiple messages within 1 single RPC call.
  b. Server-side Streaming: large file retrieval by chunks.
  c. Client-side Streaming: data batch processing.
  d. Bidirectional Streaming: live-chat windows.
- **gRPC -Multiplexing**
  a. Multiple requests from different RPC procedures handled concurrently over 1 single HTTP/2 connection.
  b. Request messages immediately are sent to server, queued when thread pool has no available workers.