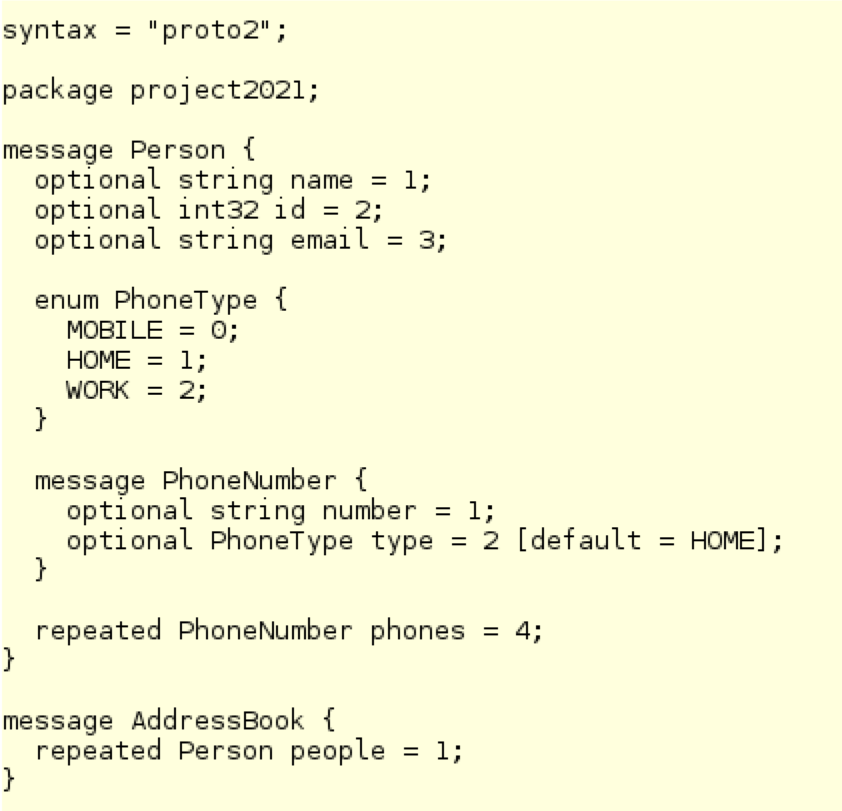
Project 2021: Using Google Protocol Buffers for structuring messages

What is Google Protocol Buffers

- Language-neutral, platform-neutral, extensible mechanism for serializing structured data
- Define a message type (fields, headers, etc.) and generates the code for formatting the messages
- Supports code generation for different languages
  - Python/C++/Java
  - Can support C language using the protobuf-c extensions

Specifying a message

- User writes a high-level specification of the messages
- This is written in a .proto file



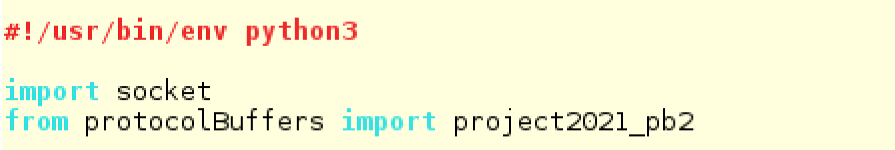
```
syntax = "proto2";

package project2021;

message Person {
  optional string name = 1;
  optional int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    optional string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

Generating the message files

We need to use the protobuf compiler to generate the messages

protoc --python_out=. project2021.proto

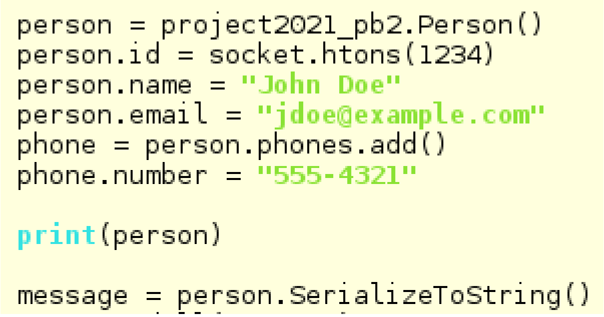
The command will generate the python files that can be included as a library in our python program (protocolBuffers is the folder that we generated the files into)



```
#!/usr/bin/env python3

import socket
from protocolBuffers import project2021_pb2
```

Using the generated files

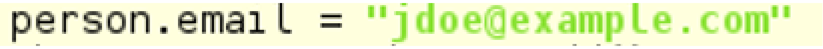
- Each message defined in the *.proto file is an object, with methods used for interacting with it



```
person = project2021_pb2.Person()
person.id = socket.htons(1234)
person.name = "John Doe"
person.email = "jdoe@example.com"
phone = person.phones.add()
phone.number = "555-4321"

print(person)

message = person.SerializeToString()
```

Optional Fields

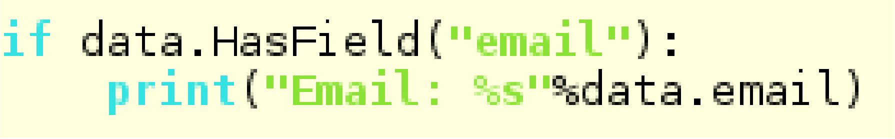
- In the .proto file, we included several optional parameters
- This means that the parameters might not exist on a message
- Methods exist for checking if the field is present, and therefore we can process it
- When the message is set in python, a separate variable is set that the message is using the field



```
person.email = "jdoe@example.com"
```

Optional Fields

- For example, at the receiver side we need to check if the field is set before accessing it
  - Otherwise, an exception will be raised



```
if data.HasField("email"):
    print("Email: %s"%data.email)
```

- This automatic updating of the HasField values is not present in all the supported languages
  - E.g. for using C, the has_*variable* field needs to be set
  - message->has_id = 1

Installing the library

On Ubuntu, the following packages need to be installed (for Python language)

```
user@machine# apt install protobuf-compiler
user@machine# apt install python3-pip
user@machine# pip3 install protobuf
```

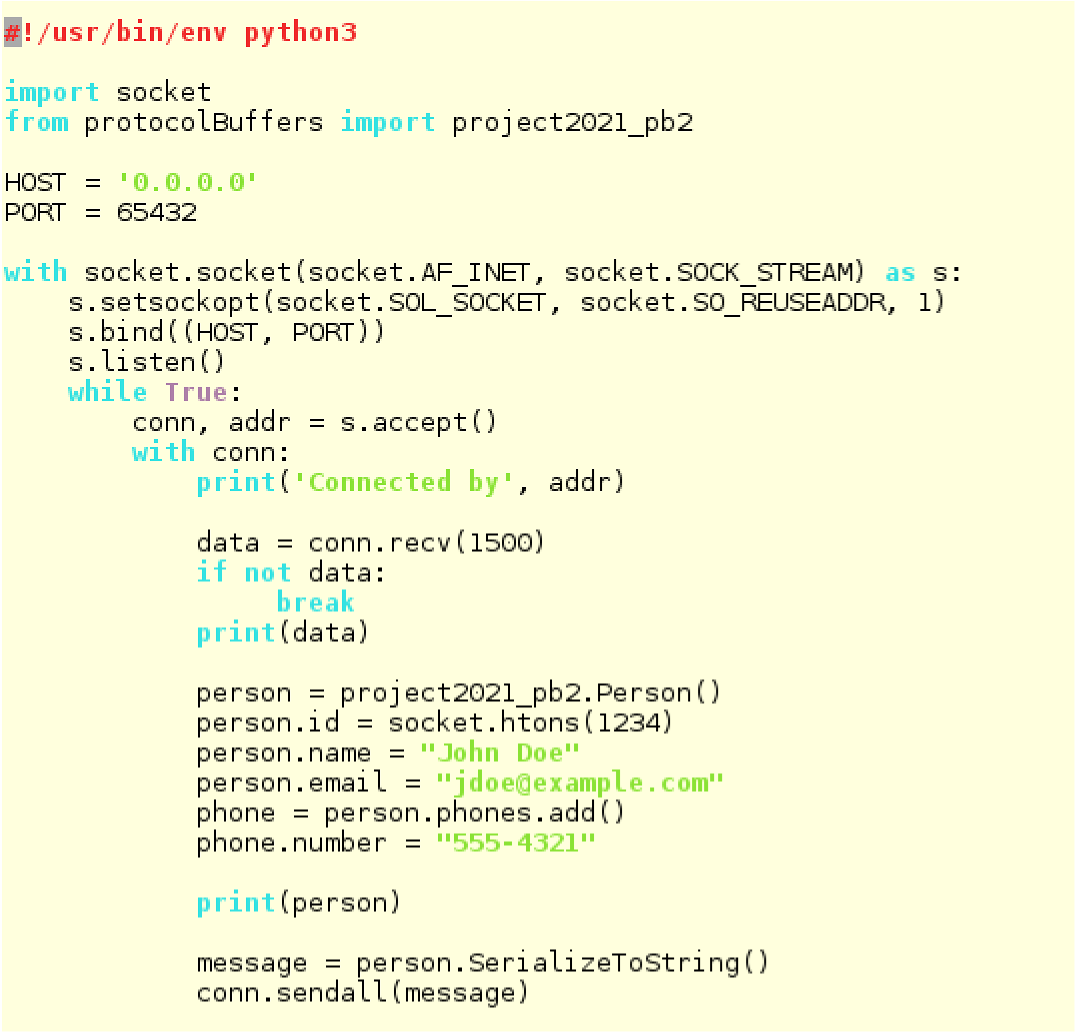
For C language, the following packages are needed

```
user@machine# apt install protobuf-c-compiler
user@machine# apt install libprotobuf-c-dev
```

When using GCC, we need to link the generated files to the protobuf library

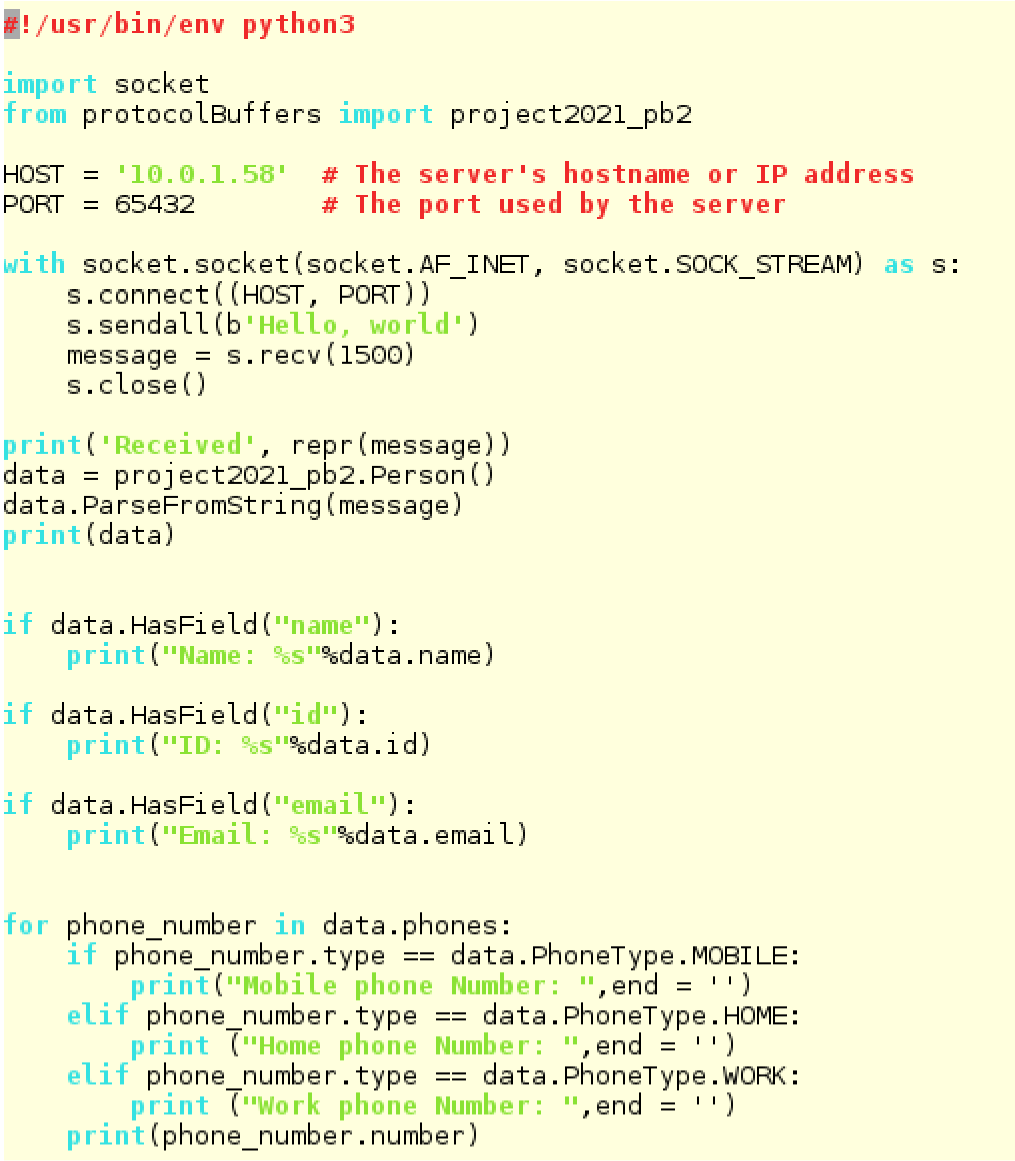
```
gcc client.c -o client -lprotobuf-c
```

Example – Server Side



```
#!/usr/bin/env python3

import socket
from protocolBuffers import project2021_pb2

HOST = '0.0.0.0'
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((HOST, PORT))
    s.listen()
    while True:
        conn, addr = s.accept()
        with conn:
            print('Connected by', addr)

            data = conn.recv(1500)
            if not data:
                 break
            print(data)

            person = project2021_pb2.Person()
            person.id = socket.htons(1234)
            person.name = "John Doe"
            person.email = "jdoe@example.com"
            phone = person.phones.add()
            phone.number = "555-4321"

            print(person)

            message = person.SerializeToString()
            conn.sendall(message)
```

Example – Client Side

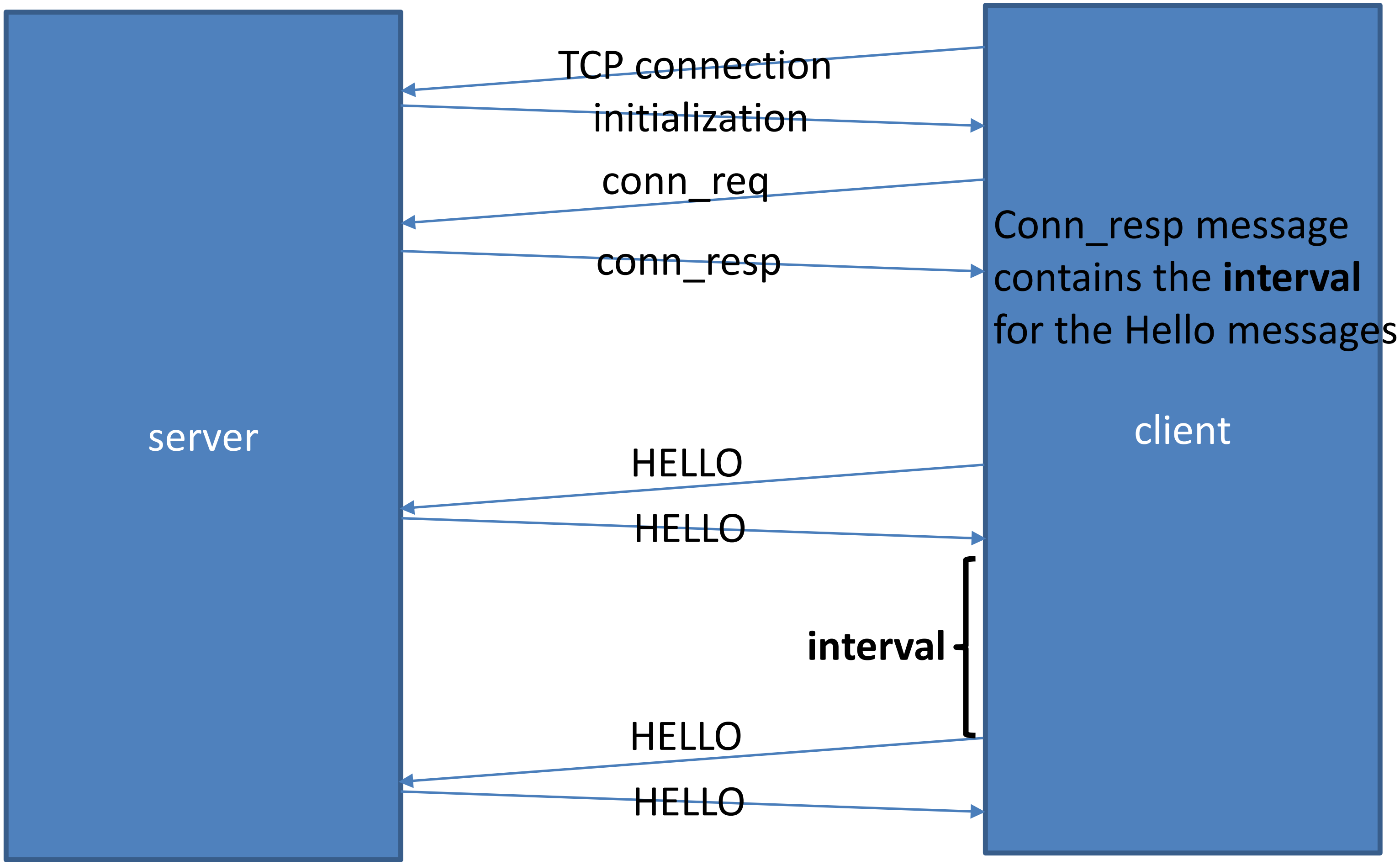


```
#!/usr/bin/env python3

import socket
from protocolBuffers import project2021_pb2

HOST = '10.0.1.58'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    message = s.recv(1500)
    s.close()

print('Received', repr(message))
data = project2021_pb2.Person()
data.ParseFromString(message)
print(data)


if data.HasField("name"):
    print("Name: %s"%data.name)

if data.HasField("id"):
    print("ID: %s"%data.id)

if data.HasField("email"):
    print("Email: %s"%data.email)


for phone_number in data.phones:
    if phone_number.type == data.PhoneType.MOBILE:
        print("Mobile phone Number: ",end = '')
    elif phone_number.type == data.PhoneType.HOME:
        print ("Home phone Number: ",end = '')
    elif phone_number.type == data.PhoneType.WORK:
        print ("Work phone Number: ",end = '')
    print(phone_number.number)
```

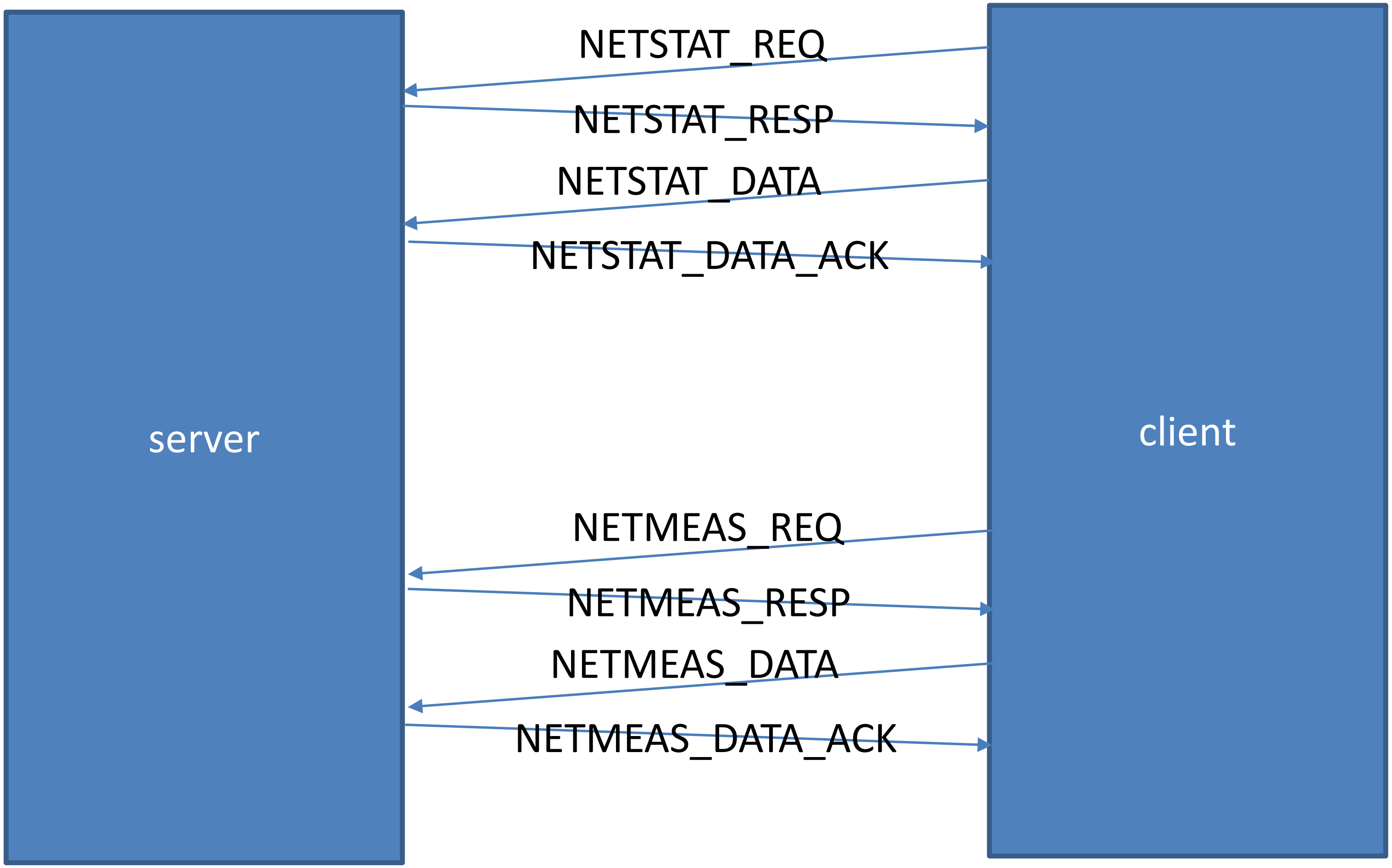
Project

- You will need to develop the functionality to communicate with a specific server based on a predefined protocol
- Types of Messages:
  - HELLO messages -> exchanged periodically, based on a random interval that the server is setting
  - CONN_REQ messages -> messages to initiate the connection
  - CONN_RESP messages -> messages that set parameters of the connection
  - NETSTAT_REQ messages -> send a message to indicate that you will send some parameters
  - NETSTAT_RESP messages -> Server responds to the NETSTAT_REQ
  - NETSTAT_DATA messages -> Connection data transmitted to the server
  - NETMEAS_REQ messages -> Send message to indicate that you will start a network measurement
  - NETMEAS_RESP messages -> Reply by the server that specifies the connection properties
  - NETMEAS_REPORT messages -> Measurement data transmitted to the server

Protocol

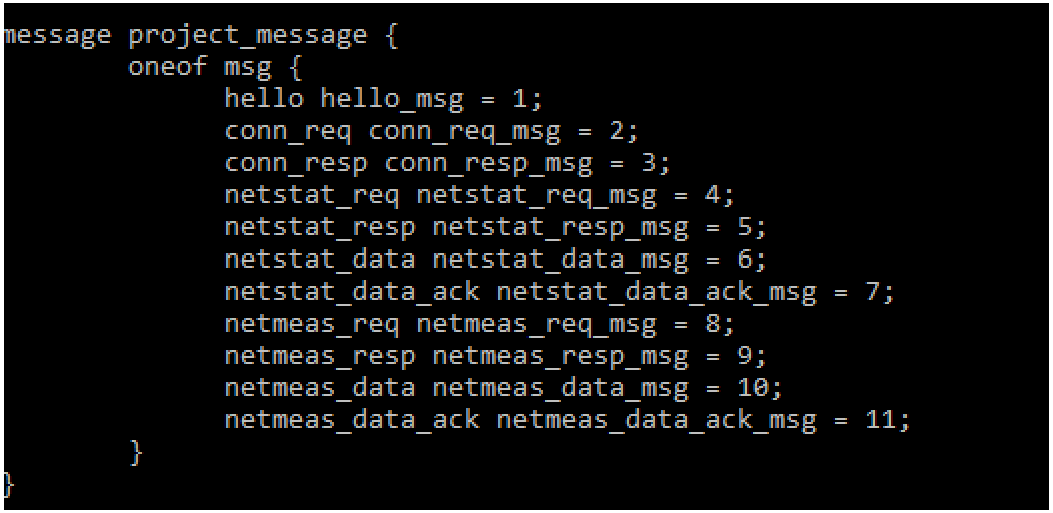

TCP connection
initialization
conn_req
conn_resp
Conn_resp message
contains the **interval**
for the Hello messages
server
client
HELLO
HELLO
interval
HELLO
HELLO

Protocol


NETSTAT_REQ
NETSTAT_RESP
NETSTAT_DATA
NETSTAT_DATA_ACK
server
client
NETMEAS_REQ
NETMEAS_RESP
NETMEAS_DATA
NETMEAS_DATA_ACK

Structure of the messages

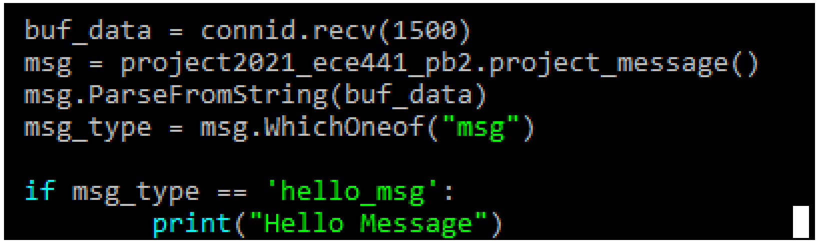
A base project_message is defined, that may include one of the types of messages



```
message project_message {
        oneof msg {
                hello hello_msg = 1;
                conn_req conn_req_msg = 2;
                conn_resp conn_resp_msg = 3;
                netstat_req netstat_req_msg = 4;
                netstat_resp netstat_resp_msg = 5;
                netstat_data netstat_data_msg = 6;
                netstat_data_ack netstat_data_ack_msg = 7;
                netmeas_req netmeas_req_msg = 8;
                netmeas_resp netmeas_resp_msg = 9;
                netmeas_data netmeas_data_msg = 10;
                netmeas_data_ack netmeas_data_ack_msg = 11;
        }
}
```

This allows the messages to be handled easier at the receiver

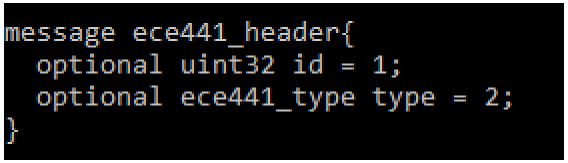
- Only need to define the generic type of message and subsequently check its type using the protocolBuffers methods



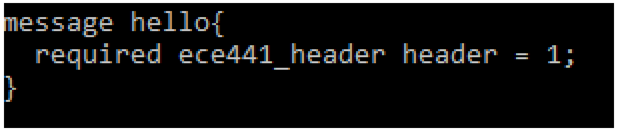
```
buf_data = connid.recv(1500)
msg = project2021_ece441_pb2.project_message()
msg.ParseFromString(buf_data)
msg_type = msg.WhichOneof("msg")

if msg_type == 'hello_msg':
        print("Hello Message")
```

Structure of the messages

Each of the messages has the same header

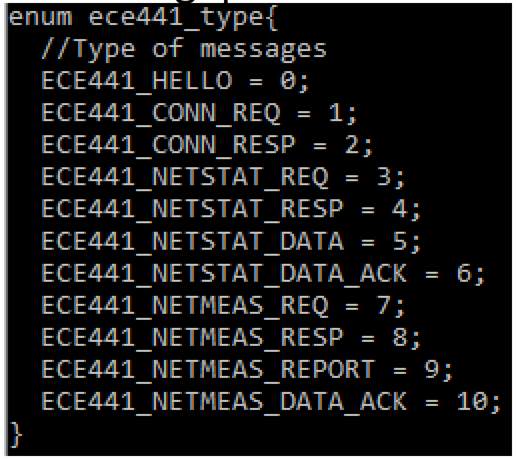


```
message ece441_header{
  optional uint32 id = 1;
  optional ece441_type type = 2;
}
```



```
message hello{
  required ece441_header header = 1;
}
```

Type should be initialized using predefined numbers

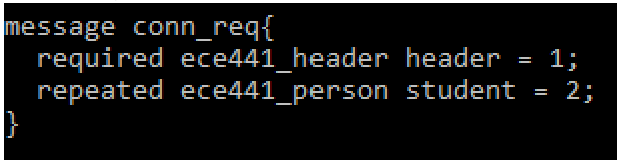


```
enum ece441_type{
  //Type of messages
  ECE441_HELLO = 0;
  ECE441_CONN_REQ = 1;
  ECE441_CONN_RESP = 2;
  ECE441_NETSTAT_REQ = 3;
  ECE441_NETSTAT_RESP = 4;
  ECE441_NETSTAT_DATA = 5;
  ECE441_NETSTAT_DATA_ACK = 6;
  ECE441_NETMEAS_REQ = 7;
  ECE441_NETMEAS_RESP = 8;
  ECE441_NETMEAS_REPORT = 9;
  ECE441_NETMEAS_DATA_ACK = 10;
}
```

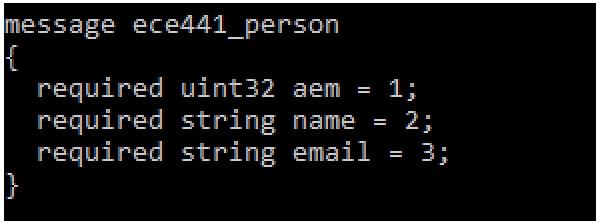
ID will be allocated to you by the instructor

Structure of the messages

For the conn_req message, you will need to send the details of the people involved in the project

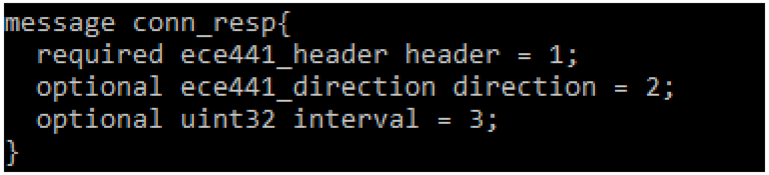


```
message conn_req{
  required ece441_header header = 1;
  repeated ece441_person student = 2;
}
```

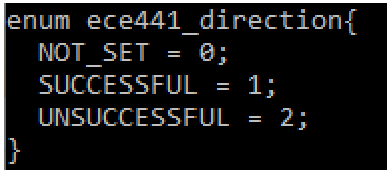


```
message ece441_person
{
  required uint32 aem = 1;
  required string name = 2;
  required string email = 3;
}
```

The server will reply with a conn_resp message, indicating whether the request is successful or not, and subsequently will start exchanging HELLO messages based on the configured interval



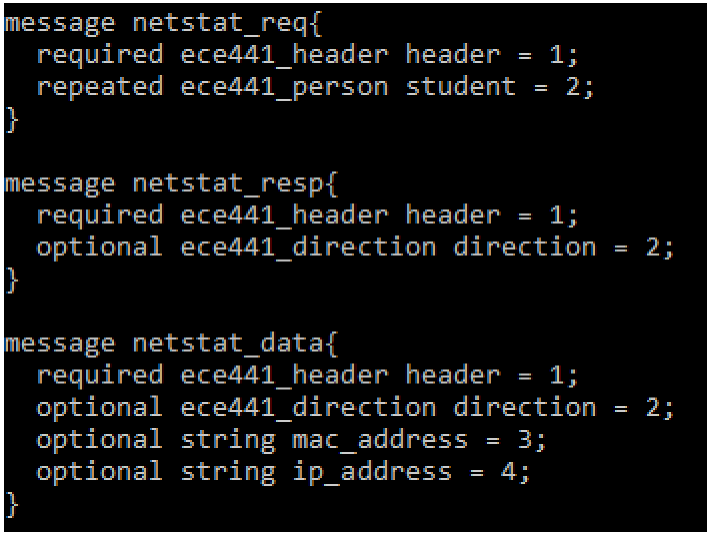
```
message conn_resp{
  required ece441_header header = 1;
  optional ece441_direction direction = 2;
  optional uint32 interval = 3;
}
```



```
enum ece441_direction{
  NOT_SET = 0;
  SUCCESSFUL = 1;
  UNSUCCESSFUL = 2;
}
```

Structure of the messages

In parallel to the HELLO messages, you will need to make two more message exchanges

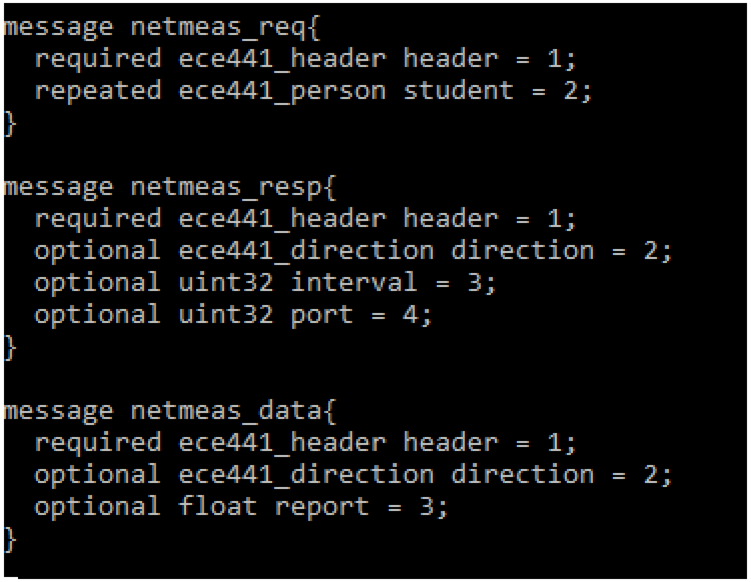


```
message netstat_req{
  required ece441_header header = 1;
  repeated ece441_person student = 2;
}

message netstat_resp{
  required ece441_header header = 1;
  optional ece441_direction direction = 2;
}

message netstat_data{
  required ece441_header header = 1;
  optional ece441_direction direction = 2;
  optional string mac_address = 3;
  optional string ip_address = 4;
}
```

Information sent in the netstat_data message will be logged by the server

Structure of the messages

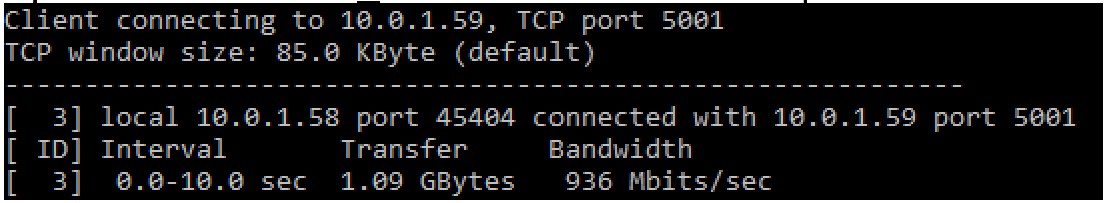
In parallel to the HELLO messages, you will need to make two more message exchanges



```
message netmeas_req{
  required ece441_header header = 1;
  repeated ece441_person student = 2;
}

message netmeas_resp{
  required ece441_header header = 1;
  optional ece441_direction direction = 2;
  optional uint32 interval = 3;
  optional uint32 port = 4;
}

message netmeas_data{
  required ece441_header header = 1;
  optional ece441_direction direction = 2;
  optional float report = 3;
}
```

Netmeas_resp message will contain the information needed to make some throughput tests with the server, using the “iperf” command

iperf –c SERVER_ADDR -t INTERVAL –p PORT



```
Client connecting to 10.0.1.59, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.1.58 port 45404 connected with 10.0.1.59 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.09 GBytes   936 Mbits/sec
```

The reported bandwidth value (936 for the illustrated case) will need to go in the netmeas_data message

Connection details

Server Address: 194.177.207.90
Server Port: 65432

Team ID: You will need to send an email to nimakris@uth.gr for being allocated an ID that your team will use for the exchanges with the server

What you should deliver: Source Code that is communicating successfully with the server side

Server is up & running, so you can check that your client is working with it