Project: 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 = "proto3";

package addressbook;

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

    message PhoneNumber {
        string number = 1;
        enum PhoneType {
            MOBILE = 0;
            HOME = 1;
            WORK = 2;
        }
        PhoneType type = 2;
    }

    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

ubuntu$ *protoc -I=. --python_out=./python_compiler addressbook.proto*

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

```
import socket
from python_compiler import addressbook_pb2
```

Using the generated files

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

```
address_book = addressbook_pb2.AddressBook()

person = address_book.people.add()
person.id = 1
person.name = "John Doe"
person.email = "johndoe@example.com"

phone = person.phones.add()
phone.number = "555-1234"
phone.type = addressbook_pb2.Person.PhoneNumber.PhoneType.MOBILE
```

Optional Fields

- In the .proto file, we can include several optional parameters
- This means that the parameters might not exist on a message
- Compilation with a flag for allowing optional fields:

```
// addressbook.proto
syntax = "proto3";

package addressbook;

message Person_opt {
    string name = 1;
    int32 id = 2;
    string email = 3;
    optional string unique_email = 4;

    message PhoneNumber {
        string number = 1;
        enum PhoneType {
            MOBILE = 0;
            HOME = 1;
            WORK = 2;
        }
        PhoneType type = 2;
    }

    repeated PhoneNumber phones = 5;
}

message AddressBook_optional {
    repeated Person_opt people = 1;
}
```

- *protoc -I=. --python_out=./python_compiler addressbook_optional.proto --experimental_allow_proto3_optional*

Optional Fields

- In the .proto file, we can include 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

```
    for phone in person.phones:
        type_name = addressbook_optional_pb2.Person_opt.PhoneNumber.PhoneType.Name(phone.type)
        print(f"Phone: {phone.number} ({type_name})")
        if person.HasField("unique_email"):
            print("User unique email: %s"%person.unique_email)
        else:
            print("No unique email for the user")
```

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 person.HasField("unique_email"):
    print("User unique email: %s"%person.unique_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

```python
# server.py
import socket
from python_compiler import addressbook_optional_pb2

HOST = '0.0.0.0'
PORT = 5000

def start_server():
    # Create a TCP socket
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((HOST, PORT))
    server_socket.listen(1)
    print(f"Server listening on {HOST}:{PORT}...")

    while True:
        # Accept a client connection
        client_socket, client_address = server_socket.accept()
        print(f"Connected by {client_address}")

        # Receive the serialized message length first
        data_length = int.from_bytes(client_socket.recv(4), 'big')
        print(f"Expecting {data_length} bytes of data")

        # Receive the serialized data
        data = client_socket.recv(data_length)

        # Deserialize the data
        address_book = addressbook_optional_pb2.AddressBook_optional()
        address_book.ParseFromString(data)
        print("Received address book:")

        # Print out the address book contents
        for person in address_book.people:
            print(f"ID: {person.id}")
            print(f"Name: {person.name}")
            print(f"Email: {person.email}")
            for phone in person.phones:
                type_name = addressbook_optional_pb2.Person_opt.PhoneNumber.PhoneType.Name(phone.type)
                print(f"Phone: {phone.number} ({type_name})")
                if person.HasField("unique_email"):
                    print("User unique email: %s"%person.unique_email)
                else:
                    print("No unique email for the user")

        # Close the client connection
        client_socket.close()

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

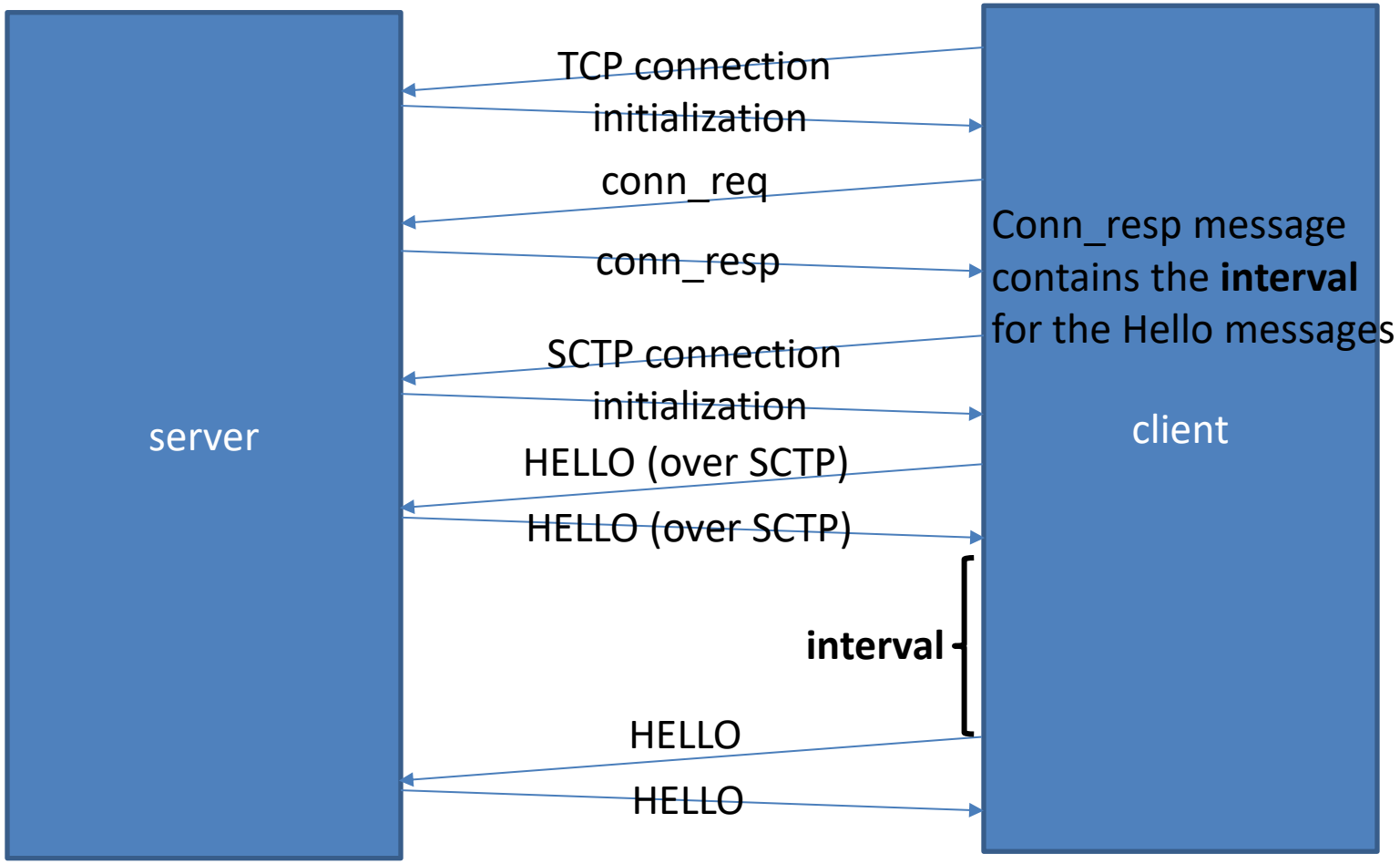
Example – Client Side

```
# client.py
import socket
from python_compiler import addressbook_optional_pb2

SERVER_HOST = '127.0.0.1'
SERVER_PORT = 5000

def create_and_send_address_book():
    # Create an AddressBook message
    address_book = addressbook_optional_pb2.AddressBook_optional()

    person = address_book.people.add()
    person.id = 1
    person.name = "John Doe"
    person.email = "johndoe@example.com"
    person.unique_email = "myjohndoe@example.com"
    phone = person.phones.add()
    phone.number = "555-1234"
    phone.type = addressbook_optional_pb2.Person_opt.PhoneNumber.PhoneType.MOBILE

    # Serialize the message to bytes
    serialized_data = address_book.SerializeToString()

    # Connect to the server and send data
    with socket.create_connection((SERVER_HOST, SERVER_PORT)) as client_socket:
        # Send the length of the message first
        client_socket.send(len(serialized_data).to_bytes(4, 'big'))

        # Send the serialized message
        client_socket.sendall(serialized_data)
        print("Address book sent to server")

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

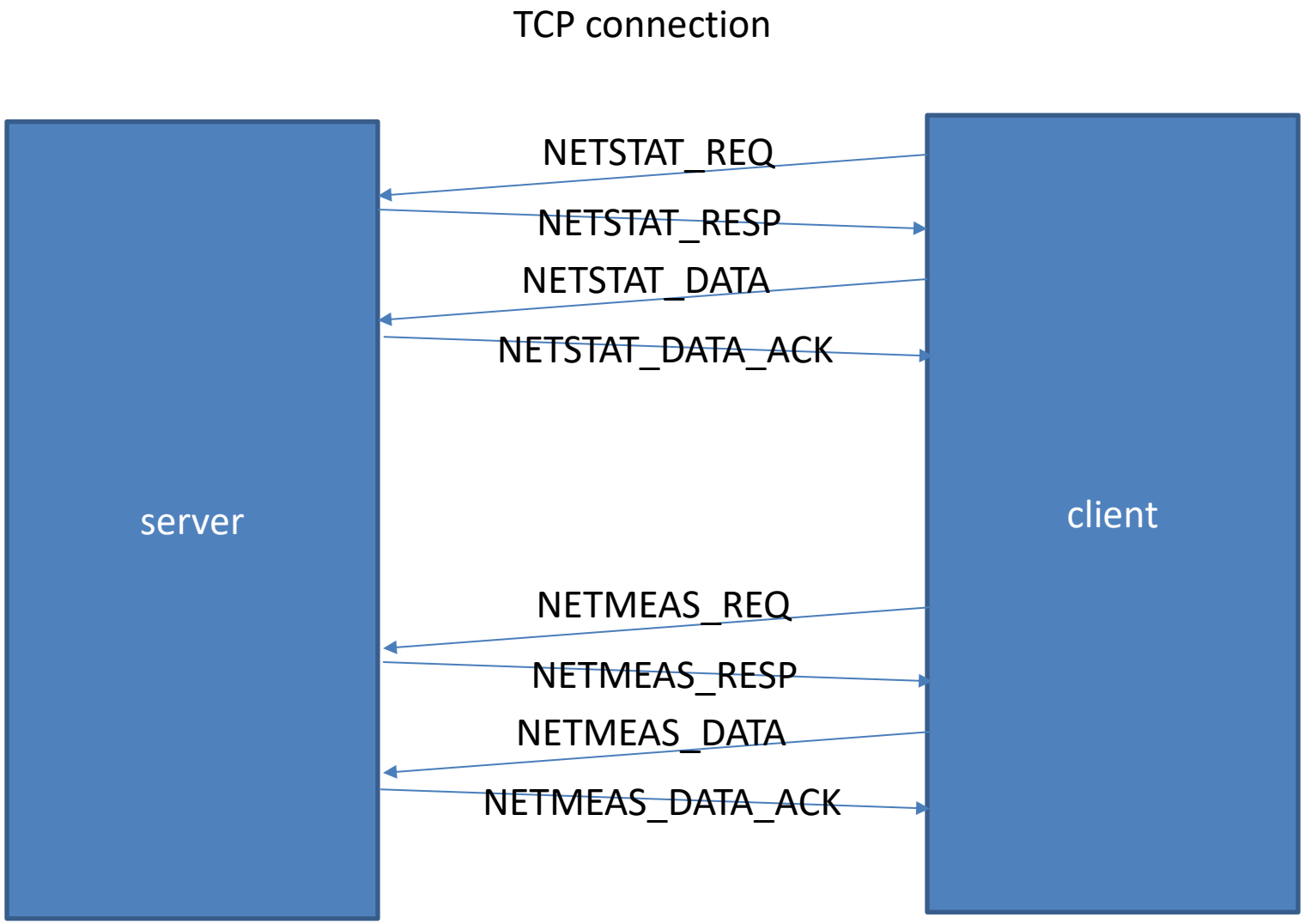
Project

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

Protocol


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

Protocol


TCP connection
server
client
NETSTAT_REQ
NETSTAT_RESP
NETSTAT_DATA
NETSTAT_DATA_ACK
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

Structure of the messages

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

```
# Deserialize the data
msg = project_messages_pb2.project_msg()
msg.parseFromString(data)
msg_type = msg.WhichOneof("msg")

if msg_type == 'hello_msg':
   print("Hello Message")
elif msg_type == 'conn_req_msg':
   print("Connection Request Message")
elif msg_type == 'conn_req_resp':
   print("Connection Request Message")
elif msg_type == 'netstat_req':
   print("Netstat Request Message")
elif msg_type == 'netstat_resp':
   print("Netstat Response Message")
elif msg_type == 'netstat_data':
   print("Netstat Data Message")
elif msg_type == 'netstat_data_ack':
   print("Netstat Data Ack Message")
elif msg_type == 'netmeas_req':
   print("Netmeasurement Req Message")
elif msg_type == 'netmeas_resp':
   print("Netmeasurement Resp Message")
elif msg_type == 'netmeas_data':
   print("Netmeasurement data Message")
elif msg_type == 'netmeas_data_ack':
   print("Netmeasurement data ack 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 based on your team ID

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;
}
```

You can also use a lib like the following: https://iperf3-python.readthedocs.io/en/latest/examples.html#client

Netmeas_resp message will contain the information needed to make some throughput tests with the server, using the “iperf3” command

iperf3 –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

Threading example

```python
import threading

def start_tcp_server():
    # Function for running the tcp server
    # ....
    # end of function

def start_sctp_server():
    # Function for running the sctp server
    # ....
    # end of function

if __name__ == "__main__":
    start_tcp_server()
    start_sctp_server()

    # Start server and client threads
    tcp_server = threading.Thread(target=start_tcp_server)
    sctp_server = threading.Thread(target=start_sctp_server)

    tcp_server.start()
    sctp_server.start()

    tcp_server.join()
    sctp_server.join()
```

Message creation example

```
    header = project_messages_pb2.ece441_header()
    header.id = 1;
    header.type = project_messages_pb2.ece441_type.ECE441_NETMEAS_DATA_ACK;

    hello_msg = project_messages_pb2.hello()
#    hello_msg = project_messages_pb2.netmeas_data_ack()
    hello_msg.header.CopyFrom(header)

    msg = project_messages_pb2.project_msg()
    msg.hello_msg.CopyFrom(hello_msg)
#    msg.netmeas_data_ack.CopyFrom(hello_msg)

    # Serialize the message to bytes
    serialized_data = msg.SerializeToString()
```

Connection details

Server Address: **Will be notified through email when the server is up**
TCP Server Port: 65432
SCTP Server Port: 54321

Team ID: You will need to register on the teams that have been created in eClass – ID will be the one on the team that you register

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

Server will be up & running by 20/11, so you can check that your client is working with it

Team registration & ID