Systems programming

Week 2 – Lab 4

Protocol buffers

Protocol buffers is a high level library for data serialization. This library can be use to allow communication of heterogeneous computers (different architectures or programming languages)

In this laboratory students will use the Protocol buffers library to implement the messages sent by ZerMQ sockets.

1 Protocol buffers

https://protobuf.dev/overview/

Protocol Buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data.

It's like JSON, except it's smaller and faster, and it generates native language bindings. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

Protocol buffers are a combination of the definition language (created in **.proto** files), the code that the proto compiler generates to interface with data, language-specific runtime libraries, and the serialization format for data that is written to a file (or sent across a network connection).

1.1 Installation

The standard protocol buffer packages are composed of a set of libraries and tools for automatic code generation. For standard languages (C++, C#, Java, Python, …) only the official packages are required, while for the C programming language additional package are required.

Since there are are official packages for most of the Linux distributions and Mac OS X, it is possible to install the require tools and libraries using the common package managers:

Ubuntu

```
sudo apt update
sudo apt install protobuf-compiler
sudo apt install protobuf-c-compiler
sudo apt install libprotobuf-c-dev
```

MAC OS X

```
brew install protobuf
brew install protobuf-c
```

MAC OS X

In mac os X the correct way to install this library is running the previous brew commands this will install all the necessary files, but, unfortunately they are not copied to the correct directories. In order to compile programs with protocol buffers follow the next steps

1 - Verify if the files are in the correct directory:**/opt/homebrew/Cellar/protobuf-c/1.5.0/**

2 – If they are in the expected directory, specify those directories when compiling:

```
gcc application.c -o client-sub -lprotobuf-c
                  -L /opt/homebrew/Cellar/protobuf-c/1.5.0/lib
                  -I /opt/homebrew/Cellar/protobuf-c/1.5.0/include
```

If the package was installed in a different directory the command **brew info protobuf-c** will print that location

1.2 Programming model

Since the protocol buffers generates code to encode messages, besides the common steps related to the use of external libraries, it is necessary for programmer to defined the message structures and compile those definitions to C code. The gebnera programming workflow is as follows:

1. Definition of messages in the **.proto** file

2. Compilation of the **.proto** file to the required languages (**C** in our case)
3. Inclusion of the generated **.h** file into the code
4. Development of the code to pack/serialize and unpack/deserialize messages
5. Compile the application with the generated **.c** file and the **-lprotobuf-c** library

1.3 **.proto** file definition

The Protocol Buffers specification defines a syntax for the **.proto** files. Independently of the programming languages latter used the syntax and semantic of these files is the same as explained in the official documentation:

https://protobuf.dev/programming-guides/proto2/

The **.proto** files allows programmers to define message type (+- like structures) and the fields that are contained in each message.

Each field has a type (scallar, strings, byte arrays or enumerates) and some optional labels as described in the documentation and sumarized in the following tables:

https://protobuf.dev/programming-guides/proto2/#scalar

https://protobuf.dev/programming-guides/proto2/#enum

https://protobuf.dev/programming-guides/proto2/#field-labels

Filed value types	
double	
float	
int32	Inefficient for encoding negative numbers
int64	Inefficient for encoding negative numbers
uint32	
uint64	
sint32	Signed int value. These more efficiently encode negative numbers than regular int32s.
sint64	Signed int value. These more efficiently encode negative numbers than regular int64s.

fixed32	Always four bytes. More efficient than uint32 if values are often greater than 228.
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 256.
sfixed32	Always four bytes.
sfixed64	Always eight bytes.
bool	
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 232
bytes	May contain any arbitrary sequence of bytes no longer than 232.

Field Label	
optional	An optional field is in one of two possible states: • the field is set, and contains a value that was explicitly set or parsed from the wire. It will be serialized to the wire. • the field is unset, and will return the default value. It will not be serialized to the wire. You can check to see if the value was explicitly set.
repeated	this field type can be repeated zero or more times in a well-formed message. The order of the repeated values will be preserved.
required	When it is used, a well-formed message must have exactly one of this field.

The name of each field should be unique inside each message and the Field Numbers should:

- be a number between 1 and 536870911
- be unique among all fields for that message.
- not be changed once your message type is in use because it identifies the field in the message wire format.

1.4 **.proto** file compilation

After the definition of the message structure **.proto**, file it is necessary to compile this file to generate the required serialization code in the intended programming language.

For the officially supported languages (C++, C#, Java, Python, …), the programmer should run the **protoc** command as explained in the tutorials:

https://protobuf.dev/getting-started/

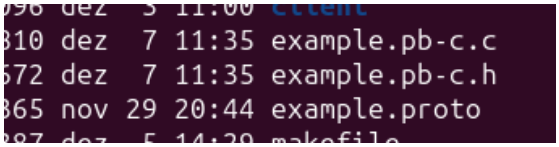
For the generation of the C files it is necessary to run the **protoc-c** command:

```
protoc-c --c_out=. example.proto
```

this basic usage of **protoc-c** requires two arguments:

- --c_out=OUT_DIR – location of the resulting C files
- **.proto** file name

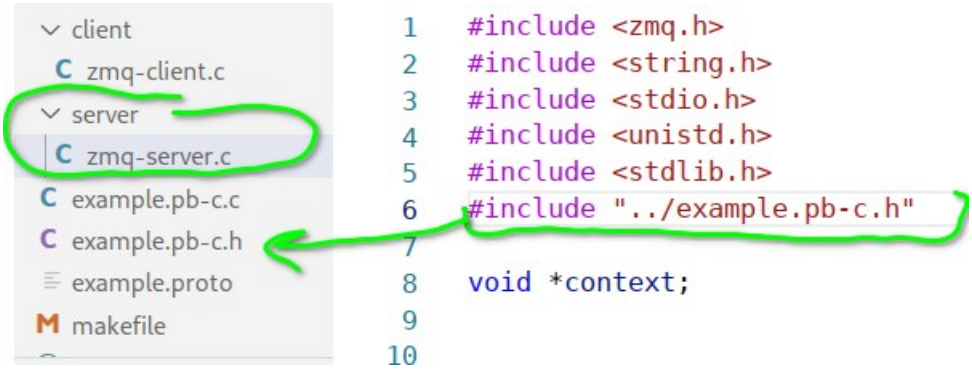
After the successful compilation, two new files are generated for each **.proto** file:



The name of the generated files depends on the original **.proto** file name.

Every time the **.proto** file is changes, it is necessary to regenerate the .c and .h files and recompile the whole project.

The generated .h file should be included with every source code file that will process the defined messages, and the .c file should be compiled and linked along the final program:




```
zmq-client: client/zmq-client.c example.pb-c.h
    gcc client/zmq-client.c example.pb-c.c -g -o zmq-client -lzmq -lprotobuf-c

zmq-server: server/zmq-server.c example.pb-c.h
    gcc server/zmq-server.c example.pb-c.c -g -o zmq-server -lzmq -lprotobuf-c
```

The following table shows the generated C structure for a sample message.

example.proto	**example**.pb-c.h

`syntax = "proto2";`
`enum EnumType {`
`  ENUM_ONE = 0;`
`  ENUM_TWO = 1;`
`}`
 	
`message Simple_message {`
 	
`  required string str_value = 1;`
 	
`  optional int32 int_number`
`  = 2 [default = 10];`
 	
`  repeated float float_array = 3;`
 	
`  optional EnumType enum_value = 4;`
 	
`}`
	`typedef enum _EnumType {`
	`  ENUM_TYPE__ENUM_ONE = 0,`
	`  ENUM_TYPE__ENUM_TWO = 1`
	`} EnumType;`
	
	`struct  SimpleMessage{`
	`  ProtobufCMessage base;`
	`  char *str_value;`
	
	`  protobuf_c_boolean has_int_number;`
	`  int32_t int_number;`
	
	`  size_t n_float_array;`
	`  float *float_array;`
	
	`  protobuf_c_boolean has_enum_value;`
	`  EnumType enum_value;`
	`};`

Along each message the **protoc-c** compiler also generates some auxiliary functions associated with each one:

`#define SIMPLE_MESSAGE__INIT`
 	
`void simple_message__init`
`(SimpleMessage *message)`
	Macro and function to initialize a **simple_mesage** structure

`size_t simple_message__get_packed_size` `(const SimpleMessage *message)`	Function to calculate how many bytes a message will occupy after serialized/packed. The returned value should be used to allocate the buffer that will contain the packed bytes.

`size_t simple_message__pack`
`(const SimpleMessage *message,`
`uint8_t *out);`
	Function that receives the message structure, verifies if all required filed are set and serializes/packs the data.
	
	The packed/serialized bytes are stored in the buffer pointed by out.
	
	This buffers should have been previously allocated with the size returned by simple_message__get_packed_size

`SimpleMessage * simple_message__unpack` `(ProtobufCAllocator *allocator,`	This function receives as argument a buffer

`size_t len,` `const uint8_t *data);`	c(containing a previously serialized/packed message) and the length of such buffer and returns a message structure with the data in the buffer. If it is imposiible to unpack the messgae null is returned. The allocator pointer should ne NULL.

`void simple_message__free_unpacked`
`(SimpleMessage *message,`
`ProtobufCAllocator *allocator);`
	This function frees a message sctruture returned by the simple_message__unpack function.
	The allocator pointer should ne NULL.

Messages with different names will have different message structures and functions generated, all with a associated name.

1.5 Development of the code to pack and unpack messages

With the help of the defined C message structures and functions, programmers can define the data to be serialized/packed and have it converted. The programmer will use variables of those a C structures to store the values to be serialized or retrieve the values just deserialized.

The general procedure for the use of protocol-buffers in a program requires:

- **for serializing/packing**
  - create a message C structure
  - fill the C message structure with the data values
  - pack the C message structure into a binary representation using the generated function
- **for serializing/unpacking**
  - store the binary representation of a message in a buffer
  - packing the buffer into the corresponding C message structure
  - access the data in the structure
  - destruct the message structure

The filling of the C message structures is done using regular C statements that manipulate the fields, depending on the types:

scalar	`required int32 a=1;`
	
	`msg.a = 12;`
		The C message structure will contain a field with the suitable type and the name defined in the message

string	`string d=1;`
	
	`msg.d = “xxxx”;`
		The C message structure will contain a pointer to a string. The name of this pointer is the one defined in the **.proto** message

bytes	`bytes name=1;`
	
	`msg.name.len= 10;`
	`msg.name.data = malloc(10);`
	`memcpy( msg.name.data, ptr, 10);`
		The C message structure will contain a variable with a pointer to the data and an integer with the length (in bytes) of the data. The data pointed by the data filed will be transferred unchanged.

optional	`optional int32 b=2;`
	
	`msg.has_b = 1; msg.b = 13;`
	
	`msg.has_b = 0;`
		The C message structure, besides the filed that will contain the data will contain another filed that states wether the corresponding fileld is present.

repeated	`repeated int32 c=1;`
	
	`msg.n_c = 200;`
	`msg.c = malloc (sizeof(int) * 200);`
	`// fill 200 int on msg.c`
		The C message structure will contain a pointer to an array of the specified values and the length of such array

The examples on the following page show how to define and access the various types of fields.

https://github.com/protobuf-c/protobuf-c/wiki/Examples

The example in folder **example-protobuff-c** shows the same procedures in messages with multiple fields

2 Protocol buffers and ZeroMQ and C

When using protocol buffers and sockets, it is necessary to read from the sockets packed messages, all of them with varying lengths. Since, for instance, arrays with different lengths will be packed into buffers also with different lengths, the programmer does not know what will be the length of the data to be read from the sockets. A solutions such as the following may not work if the packed data may not fit into 100 bytes:

```
char msg_data[100];
int msg_len = zmq_recv(req_socket, buffer, 100, 0);
simple_msg = simple_message__unpack(NULL, msg_len, msg_data);
```

If the packed data (transmitted in the socket) is larger that 100 bytes, part of the message will be lost and the **simple_message__unpack** function will return an error.

To circumvent this, when reading messages from ZeroMQ it is fundamental to use an intermediary data structure: the **zmq_msg_t**, i.e. the ZeroMQ message type.

Information about management of zmq_msg na be access in the following pages:

https://libzmq.readthedocs.io/en/zeromq3-x/zmq_msg_init.html
https://libzmq.readthedocs.io/en/zeromq3-x/zmq_msg_recv.html
https://libzmq.readthedocs.io/en/zeromq3-x/zmq_msg_data.html
https://libzmq.readthedocs.io/en/zeromq3-x/zmq_msg_close.html

The previous code would become the following:

```
zmq_msg_t zmq_msg;
zmq_msg_init (&zmq_msg);
int msg_len = zmq_recvmsg (responder, &zmq_msg, 0);
void * msg_data = zmq_msg_data (& zmq_msg);
simple_msg = simple_message__unpack(NULL, msg_len, msg_data);
zmq_msg_close (zmq_msg)
```

A variable of type **zmq_msg_t** is created and initialized, and used to store the received message instead of a byte buffer. The programmer should read the message using the

zmq_recvmsg, and use the function **zmq_msg_data** to get the buffer of the data associated with the received message. After being processed the variable of type **zmq_msg_t** can be destroyed/closed.

This way, independently of the length of the data transmitted with **zmq_send**, the receiver will always get the data in a suitable sized buffer.

The code in **lab9-exercise-1** folder already contains a protocol buffers message (defined in the file **balls.proto**) and the **remote-display-client** already deserializes one this type of messages using protocol buffer. The code to deserialize messages sent using ZeroMQ is the following:

```
BallDrawDisplayMsg  * zmq_read_BallDrawDisplayMsg(void * subscriber){

        zmq_msg_t msg_raw;
        zmq_msg_init (&msg_raw);
        int n_bytes = zmq_recvmsg(subscriber, &msg_raw, 0);
        char *pb_msg = zmq_msg_data (&msg_raw);

        BallDrawDisplayMsg  * ret_value =
                ball_draw_display_msg__unpack(NULL, n_bytes, pb_msg);
        zmq_msg_close (&msg_raw);
        return ret_value;
}
```

After calling this function, it is possible to access the fields in the returned C structure to fetch the data sent by the server.

Thje code in the server to create the corresponding message is as follows:

```
void zmq_send_BallDrawDisplayMsg(void * publisher,
                                 char ch, int x, int y, int random_secret){


    zmq_send(publisher, &random_secret, sizeof(random_secret), ZMQ_SNDMORE);

    BallDrawDisplayMsg  pb_m_stuct = BALL_DRAW_DISPLAY_MSG__INIT;
    pb_m_stuct.ch.data = malloc(sizeof(ch));
    memcpy(pb_m_stuct.ch.data, &ch, sizeof(ch));

    pb_m_stuct.ch.len = sizeof(ch);
    pb_m_stuct.x = x;
    pb_m_stuct.y = y;

    int size_bin_msg = ball_draw_display_msg__get_packed_size(&pb_m_stuct);
```

```
    char * pb_m_bin = malloc(size_bin_msg);
    ball_draw_display_msg__pack(&pb_m_stuct, pb_m_bin);

    zmq_send(publisher, pb_m_bin, size_bin_msg, 0);
}
```

This code does not include the destruction/free of the dynamically allocated memory.

When receiving messages, the programmer needs to know in advance what message was read from the socket to call the correct unpacking function. ZeroMQ helps here by allowing the programmer to send a message identifier (with the **ZMQ_SNDMORE**) before sending the packed buffer:

Senders	Receiver

`int msg_t = XXX;`
`zmq_send(sock1, &msg_t,`
`sizeof(msg_t), ZMQ_SNDMORE);`
 	
`zmq_send (sock1, &msg, msg_len, 0);`
	`int msg_t;`
	`zmq_recv (sock_2, &msg_type,`
	`sizeof(msg_type), 0);`
	`if(msg_t == 1){`
	`zmq_recvmsg(sock, &msg_raw, 0);`

If the protocol only accepts one specific message type it is not necessary to send the message type as a part of the ZeroMQ message.

3 Exercise 3

The provided code (**lab9-exercise-1**) is a solution to the exercise 3 of lab 5. Here we not only have a server, two clients (human and machine) that move letters, but also a pay-per-view display.

In this exercise students should replace all the messages sent as C structures by similar protocol buffer messages.

The messages sent from the **server** to the **remote-display-client** are already serialized with protocol buffers but no other interaction uses this.

Follow the next steps to fully convert this system to protocol buffers. In the end of this exercised all structures defined in the **remote-char.h** will be unused and can be removed from it.

3.1 remote-display-client connection

TODO 1 – Define in the file **balls.proto** the structure of the pay-per-view registration. In the request the remote-display-client send two strings (the client name and the credit card number) and in the replay the server send the secret to allow the access to the game updates.

TODO 2 – on the **remote-display-client.c** encode the two string (**client_name** and **cc_str**) int the C message structure and send the packed message to the server

TODO 3 – in the server read and process the **payperview_req** message

TODO 4 – in the server send the reply as a protocol buffer **payperview_resp** message

TODO 5 – in the **remote-display-client.c** read and process the **payperview_resp** message

3.2 Human and machine clients connection

TODO 6 – in **balls.proto** define the messages **client_connection_req** that will serialize initial client message

TODO 7 – in the **machine-client.c** and **human-client.c** create and fill a message of type **client_connection_req** and send the packed message to the server

TODO 8 – in the **server** read and process the **client_connection_req** message

3.3 Human and machine clients ball movements

TODO 9 – in **balls.proto** define the **movement_req** message that will serialize the movements of balls

TODO 10 – in the **machine-client.c** and **human-client.c** create and fill a message of type **movement_req** and send the packed message to the server

TODO 11 – in the **server** read and process the **movement_req** message

4 Exercise 2

Try to implement the machine client in a different language (for instance java or python)

In order to use protocol and ZeroMQ sockets students should also install the following:

- **for python**
  - pip install pyzmq
  - pip install protobuf
- **for java**
  - https://github.com/protocolbuffers/protobuf/blob/main/java/README.md
  - https://github.com/zeromq/jeromq

and follow the next tutorials

- **for python**
  - https://zeromq.org/languages/python/
  - https://protobuf.dev/getting-started/pythontutorial/
- **for java**
  - https://zeromq.org/languages/java/
  - https://protobuf.dev/getting-started/javatutorial/