Week 7-9: RPC via Protocol Buffers
November 2020

Understanding Protocol Buffers

Before writing the code for the client, we must understand how Protocol Buffer messages are encoded.

For eg:, let’s take an example syscallprot.proto:

```
syntax = "proto3";


package sys_rpc;

syntax = "proto3";


package sys_rpc;


message xRPC_message_type {

 enum Type {

   request = 0;

   response = 1;

 }

 Type type = 1;// stores whether the message is a request or response

 enum Procedure {

   settimeofday = 0;
   gettimeofday = 1;
 }
 Procedure procedure = 2;//indicates the function: settimeofday or gettimeofday.
}
message settimeofdayRequest {
 message timeval {
   uint32 tv_sec = 1;// store time elapse since Jan 1, 1970 00:00 in seconds
   uint32 tv_usec = 2; // store time elapse since Jan 1, 1970 00:00 in microseconds
 }
 //Scrapped the timezone message, since it is now obsolete. (USAGE OF UTC)
 timeval timeval_s= 1;
}
message settimeofdayResponse {
   sint32 return_value = 1; //stores if function call was succesful or not
   sint32 errno_alt = 2; //stores error no., if fault occurs
}


//message gettimeofdayRequest {
//    int32 stub = 1;
//}


message gettimeofdayResponse{
   message timeval {

   uint32 tv_sec = 1;

   uint32 tv_usec = 2;

   }

   //planning to scrap timezone completely, as it is obsolete

 message gettimeofdayRequestStatus {

   sint32 return_value = 1;//stores if function call was succesful or not

   sint32 errno_alt = 2; //stores error no., if fault occurs

   }

   timeval timeval_r = 1;

   gettimeofdayRequestStatus status = 2;

}

message xRPC_message{

 xRPC_message_type mes_type = 1;

 settimeofdayRequest setTimeRequest = 2;

 settimeofdayResponse setTimeResponse = 3;

//  gettimeofdayRequest getTimeRequest = 4;

 gettimeofdayResponse getTimeResponse = 4;

}
```

The above protocol buffer message schema is used to send/ receive requests and responses for gettimeofday() and settimeforday() functions.

The comments explain the purpose of each parameter in the messages.

We can observe in the above schema that some of the messages also include submessages in them. This helps to reuse the parameters, since if two messages use the same parameters, then those parameters can be defined in a separate message which can be used as a submessage. For eg: the message xRPC_message contains a submessage xRPC_message_type which defines whether the message sent is a settimeofday request, gettimeofday response, etc.

When all parameters in the messages have been initialized with the required data, the message can be easily visualized as a JSON message.

For example: if we want to send a settimeofday request message, xRPC_message is the main message field, rest message fields are used as submessages.

- mes_type(xRPC_message_type): type is set as request and procedure is set as settimeofday
- setTImeRequest(settimeofdayRequest): The parameters tv_sec and tv_usec in submessage timeval_r are set with the required time values.

Since we do not require the other fields for this message, the main parameters and submessage parameters in those fields are initialized with NULL values.

So, we can visualize the message structure as:

mes_type {

type: 0 (Remember that enums are set as integer types. In this case, enum value ‘request’ is set as 0 and enum value ‘response’ is set as 1. So, if this message was of response type, value of type would have been 1. The values are determined by the order in which the enums are defined.)

procedure: 0 (0 indicates enum value ‘settimeofday’)

}

IMPORTANT: Whenever the value assigned to parameters in protocol buffer messages is 0, those parameters will not be stored in the message, so when blank messages are received , the receiver interprets the values of parameters in those messages to be 0. Hence , mes_type shown above will be blank and will look like:

```
mes_type{
}
setTimeRequest {
  timeval_s {
            tv_sec: 1605930811
            tv_usec: 786673
  }
}
setTimeResponse {
}
getTimeResponse {
  timeval_r {
  }
  status {
  }
}
```

We can see that the two message fields getTimeResponse and setTimeRespose are blank, except for the fact that getTimeResponse has two submessages timeval_r and status whose parameters are blank, so those submessages are shown. The message setTimeRequest has a submessage timeval_s which has parameters tv_sec and tv_usec set to the required values.

The visualization shown above is an unserialized protocol buffer format, which is very similar to the JSON format. However, after serialization, the above fields are converted into binary stream with each field being assigned a varint, and the blank fields will not be stored.

The binary stream of the above message in bytes format:

```
\n\x00\x12\x0c\n\n\x08\xbb\x9e\xe2\xfd\x05\x10\xf1\x810\x1a\x00"\x04\n\x00\x12\x00
```

Each \n….\n represents a message field. We can see that there are only 2 \n…\n pairs for the message. (Check Google Protocol Buffers documentation for more detail on this.)

Compiling Protocol Buffers for C:

The proto file has to be converted to C format by the protoc compiler. SInce the protoc compiler does not support C natively, we'll use protobuf-c to compile the .proto file to C format.

To install: sudo apt-get install protobuf-c-compiler

To compile, navigate to the directory where the .proto file is stored , then execute the command below to compile the .proto file:

```
protoc-c --c_out=. <filename>.proto
```

Eg: protoc-c --c_out=. syscallprot.proto

This generates two files, syscallprot.pb-c.c and syscallprot.pb-c.h .These two files contain functionality for handling protocol buffers in C.

Compiling Protocol Buffers for Python:

The proto file has to be converted to python format by the protoc compiler. This can be done by navigating to the folder where the .proto file is stored and executing the below command:

```
protoc --python_out=. <filename>.proto
```

Eg:

protoc --python_out=. syscallprot.proto

This will generate a file named syscallprot_pb2.py, which will contain the functionality for handling protocol buffers in Python.

Handling Protocol Buffers in C explained using the ESP32 application:

This section will explain how to handle protocol buffers in C using the ESP32 application as an example.

Let’s open up the application code (app_main.c in the main directory of our ESP32 application project directory) alongside this document to understand how to implement protocol buffer handling in C.

First, let’s understand some of the ESP32 specific code.

```
static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)
```

This is the function which is used to handle MQTT events. It takes an argument **event** of type **esp_matt_event_handle_t;** which represents a MQTT receive, MQTT publish, MQTT connection, etc. which is defined in the header mqtt_client.h . Detailed documentation regarding this can be found here: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/mqtt.html

Let’s move on to the next statement:

```
esp_mqtt_client_handle_t client = event->client;
```

A variable client of type esp_mqtt_client_handle_t is defined, which contains information about the mqtt client (our ESP32 board)which is used to handle the client operations such as publishing and subscribing to MQTT topics.

Next, two variables len and len2 to store length of unpacked(deserialized) messages and packed(serialized) messages respectively are declared, which are of type size_t.

Next, we can see the switch…case structure which s used to handle MQTT events. Each event is represented by and event_id which is a variable stored in the event argument passed to the **mqtt_event_handler_cb** function.

The events handled are MQTT_EVENT_CONNECTED, MQTT_EVENT_DISCONNECTED, MQTT_EVENT_SUBSCRIBED, MQTT_EVENT_UNSUBSCRIBED, MQTT_EVENT_PUBLISHED, MQTT_EVENT_DATA and MQTT_EVENT_ERROR. The cases where these events are triggered are defined below.

MQTT_EVENT_CONNECTED: When a connection to MQTT broker is established.

MQTT_EVENT_DISCONNECTED: When connection to MQTT broker is terminated.

MQTT_EVENT_SUBSCRIBED: When the client subscribes to a MQTT topic

MQTT_EVENT_UNSUBSCRIBED: When the client unsubscribes from a MQTT topic

MQTT_EVENT_PUBLISHED: When the client publishes a message to a topic

MQTT_EVENT_DATA: When a client receives a MQTT message from the broker

MQTT_EVENT_ERROR: When an error occurs, i.e connectivity to broker ends abruptly, etc.

We also observe a lot of **ESP_LOGI(TAG, “message”, msg_id)** statements. These statements are used to log events in the monitor(idf.py monitor), we can use these to see the events taking place in this program. The variables TAG and msg_id are defined at the beginning of **mqtt_event_handler_cb function.**

The events we need to focus on are: MQTT_EVENT_CONNECTED and MQTT_EVENT_DATA.

When our ESP32 client establishes a connection with the MQTT broker, we need to publish to the topic 101/xRPC_Request so that we can receive the incoming settimeofday and gettimeofday requests.

The most important event that needs to be handled is the MQTT_EVENT_DATA, which is triggered when we receive messages from other clients.

MQTT_EVENT_DATA handling:

When we receive an incoming message from other clients, the message is stored in the variable data in event(event->data). Remember that this variable event contains information about the incoming message data, message length, event_id, etc. Details can be found here:
https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/protocols/mqtt.html

To store this raw message, we have to declare a void pointer named buffer(**void *buffer**), and allocate memory to this buffer using malloc(). The length of message is extracted from event->data_len and stored into variable len(size_t len declared earlier), and allocate memory of required size(buffer = malloc(len)). After allocating memory, the message is stored to the buffer (buffer = event->data), and now we need to deserialize the data in order to process it.

Now, let's take an example where the incoming message is of type settimeofday request and we need to send back a settimeofday response message.

Referring to the syscallprot.proto file definition posted in Understanding Protocol Buffers(pages 1-3), we see that the main message field here is xRPC_message. The other message fields, namely xRPC_message_type, SettimeofdayRequest, SettimeofdayResponse, GettimeofdayResponse are just submessages of xRPC_message. From this, we know the incoming message will have parameters mapped to the submessasges of xRPC_message. The whole message is then set as a package named sys_rpc.

According to the naming conventions of protoc-c compiler, the data types are named as <package-name>__<messagename>. In this case package name is set to Sys_Rpc and <message-name> can be XRPC_Message, GettimeofdayResponse, SettimeofdayRequest and SettimeofdayResponse. All these possible message types are defined at the beginning of the program with appropriate comments for explanation.

The naming convention is explained here- Refer to naming and casing conventions section.

https://github.com/protobuf-c/protobuf-c/wiki/Generated-Code

So, for declaring a variable with message field type xRPC_message, the corresponding C data type generated is: Sys_Rpc__XRPC_Message.

__ is used to separate between packages, message fields.

The characters after every _ is capitalized, so for package sys_rpc the first character is capitalized according to the CamelCase convention , and the 'r' after the _ is capitalized , so we get Sys_Rpc. xRPC_message is a message field inside the package sys_rpc, so a __ is put to separate the package name and message field name. Then X is capitalized as per CamelCase convention and the 'm' which comes after _ is capitalized, so we get Sys_Rpc__XRPC_Message.

Enums and inits are defined in all caps.

For eg:

enum value 'request' of type in xRPC_message_type message field in package sys_rpc is defined as: **SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE_request**

enum value 'settimeofday' of procedure in xRPC_message_type message field in package sys_rpc is defined as: **SYS_RPC__X_RPC_MESSAGE_TYPE_PROCEDURE__settimeofday**

Initializer for message field xRPC_message in package sys_rpc is defined as: **SYS_RPC__X_RPC_MESSAGE__INIT**

Initializer for submessage field Timeval in message field settimeofdayRequest in package sys_rpc is defined as: **SYS_RPC__SETTIMEOFDAY_REQUEST__TIMEVAL_INIT** (here, when there is a capitalized letter in any field(package, message or submessage), a _ is inserted before it.)

Lastly, all generated functions will be in lower case. For eg:, function unpack which unpacks message of type xRPC_message of package sys_rpc is named as:
sys_rpc__x_rpc_message__unpack()

Now, we declare a pointer recvd of type Sys_Rpc__XRPC_Message to store the unserialized message.

```
SysRpc__XRPCMessage *recvd = sys_rpc__x_rpc_message__unpack(NULL, len,
buffer)
```

Next, we check if the message type is request and if the procedure is gettimeofday.

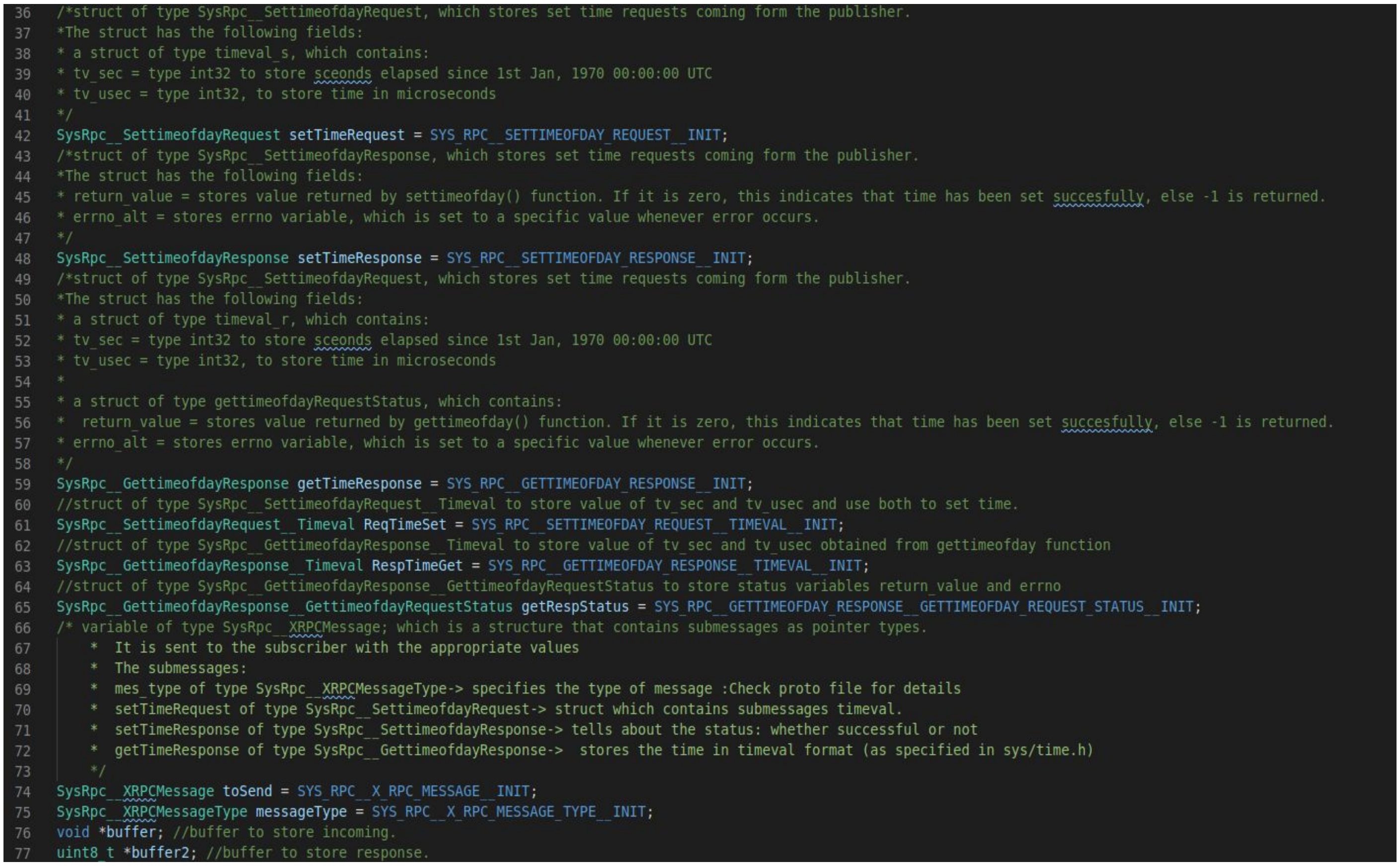
```
if(recvd->mes_type->type == SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE__request &&
recvd->mes_type->procedure ==
SYS_RPC__X_RPC_MESSAGE_TYPE__PROCEDURE__gettimeofday)
```

This turns out to be false, so we move on to the next if:

```
if(recvd->mes_type->type == SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE__request &&
recvd->mes_type->procedure ==
SYS_RPC__X_RPC_MESSAGE_TYPE__PROCEDURE__settimeofday)
```

Which is true for our example(settimeofday request). Hence, we move on with assigning values to the parameters in xRPC_message. A message variable toSend of type Sys_Rpc__XRPC_Message has already been declared and initialized with SYS_RPC__X_RPC_MESSAGE__INIT at the beginning of the code, just after the #include statements. Now we have to set the submessage variables mes_type, setTimeRequest, setTimeResponse and getTimeResponse, in which we will pass parameters to setTimeResponse and mes_type.

First we have to set the message type as response and procedure as settimeofday. , which is stored in submessage mes_type of type xRPC_message_type. To do so, we need to first initialize a variable of type xRPC_message_type, store the parameters in it and pass its reference to toSend.mes_type . This is so because internally submessage structures are initialized as pointers in the main message structure definition in syscallprot.pb-c.h header, and when we initialize the main message structure of type xRPC_message, the submessage structure pointers are set to NULL. Trying to access parameters of submessage structures directly using the main message structure would result in a “attempt to access NULL pointer” error. So, we need to initialize the submessage types separately and pass their references to the main message structure variables.



```
/*struct of type SysRpc__SettimeofdayRequest, which stores set time requests coming form the publisher.
*The struct has the following fields:
* a struct of type timeval_s, which contains:
* tv_sec = type int32 to store sceonds elapsed since 1st Jan, 1970 00:00:00 UTC
* tv_usec = type int32, to store time in microseconds
*/
SysRpc__SettimeofdayRequest setTimeRequest = SYS_RPC__SETTIMEOFDAY_REQUEST__INIT;
/*struct of type SysRpc__SettimeofdayResponse, which stores set time requests coming form the publisher.
*The struct has the following fields:
* return_value = stores value returned by settimeofday() function. If it is zero, this indicates that time has been set succesfully, else -1 is returned.
* errno_alt = stores errno variable, which is set to a specific value whenever error occurs.
*/
SysRpc__SettimeofdayResponse setTimeResponse = SYS_RPC__SETTIMEOFDAY_RESPONSE__INIT;
/*struct of type SysRpc__SettimeofdayRequest, which stores set time requests coming form the publisher.
*The struct has the following fields:
* a struct of type timeval_r, which contains:
* tv_sec = type int32 to store sceonds elapsed since 1st Jan, 1970 00:00:00 UTC
* tv_usec = type int32, to store time in microseconds
*
* a struct of type gettimeofdayRequestStatus, which contains:
*  return_value = stores value returned by gettimeofday() function. If it is zero, this indicates that time has been set succesfully, else -1 is returned.
* errno_alt = stores errno variable, which is set to a specific value whenever error occurs.
*/
SysRpc__GettimeofdayResponse getTimeResponse = SYS_RPC__GETTIMEOFDAY_RESPONSE__INIT;
//struct of type SysRpc__SettimeofdayRequest__Timeval to store value of tv_sec and tv_usec and use both to set time.
SysRpc__SettimeofdayRequest__Timeval ReqTimeSet = SYS_RPC__SETTIMEOFDAY_REQUEST__TIMEVAL__INIT;
//struct of type SysRpc__GettimeofdayResponse__Timeval to store value of tv_sec and tv_usec obtained from gettimeofday function
SysRpc__GettimeofdayResponse__Timeval RespTimeGet = SYS_RPC__GETTIMEOFDAY_RESPONSE__TIMEVAL__INIT;
//struct of type SysRpc__GettimeofdayResponse__GettimeofdayRequestStatus to store status variables return_value and errno
SysRpc__GettimeofdayResponse__GettimeofdayRequestStatus getRespStatus = SYS_RPC__GETTIMEOFDAY_RESPONSE__GETTIMEOFDAY_REQUEST_STATUS__INIT;
/* variable of type SysRpc__XRPCMessage; which is a structure that contains submessages as pointer types.
    *  It is sent to the subscriber with the appropriate values
    *  The submessages:
    *  mes_type of type SysRpc__XRPCMessageType-> specifies the type of message :Check proto file for details
    *  setTimeRequest of type SysRpc__SettimeofdayRequest-> struct which contains submessages timeval.
    *  setTimeResponse of type SysRpc__SettimeofdayResponse-> tells about the status: whether successful or not
    *  getTimeResponse of type SysRpc__GettimeofdayResponse->  stores the time in timeval format (as specified in sys/time.h)
    */
SysRpc__XRPCMessage toSend = SYS_RPC__X_RPC_MESSAGE__INIT;
SysRpc__XRPCMessageType messageType = SYS_RPC__X_RPC_MESSAGE_TYPE__INIT;
void *buffer; //buffer to store incoming.
uint8_t *buffer2; //buffer to store response.
```

So, as seen in the above picture, each submessage type has been initialized, we can take a look at the app_main code to see in more detail. The comments explain briefly the functionality of each submessage type.

To store the message type and procedure, we declare a variable messageType of type Sys_Rpc__XRPCMessageType which contains the parameters type and procedure. Both are enums, in which type stores values ‘response’ and ‘request’ and procedure stores ‘gettimeofday’ and ‘settimeofday’. Since we have to publish a settimeofday response, we set messageType.type as **SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE_response** and messageType.procedure as **SYS_RPC__X_RPC_MESSAGE_TYPE__PROCEDURE_settimeofday**.

Then, we have to pass reference of messageType to toSend.mes_type to make the pointer mes_type point towards messageType. We have just set the procedure and type.

Next, we have to set the time by calling the appropriate function, which is done by calling a specific element function pointer array xRPC_func, which stores the name of function to be used. We can find out the index of the name of required function by referring to the array declaration.

```
static pf xRPC_func[] = {x_gettimeofday, x_settimeofday};
```

The function pointer array is of type pf, which is a function pointer of type int which takes in two pointer arguments of type void: void *request and void*response. It is defined using typedef.

```
typedef int (*pf)(void *request, void *response);
```

From thexRPC_func pointer function array, we can see that we require the function name x_settimeofday, which is the 2nd element. So, its index would be 1 and the required function call would be:

```
ret = (*xRPC_func[1])(recvd->settimerequest, &setTimeResponse)
```

Here, the * operator is used to dereference the pointer so that we get the value to which this pointer points, i.e x_settimeofday. The function x_settimeofday is called which is provided the arguments recvd->settimerequest and &setTimeResponse. The argument recvd->settimerequest contains the parameters required to set time using settimeofday() function, and the address of setTimeResponse is sent to store the response of settimeofday() in.

x_settimeofday():

At first, we declare a structure of type timeval tv, which is used internally in sys/time.h to store the time elapsed in seconds and microseconds since JAN 1, 1970 00:00:00 in variables tv.tv_sec and tv.tv_usec respectively.

We have to set tv.tv_sec and tv.uscec to values stored in tv_sec and tv_usec parameters in timeval field in settimeofdayRequest submessage(recvd->settimerequest) respectively.

First, the request parameter must be set to pointer type Sys_Rpc_SettimeofdayRequest* , so that we can pass the deferenced values to ReqTimeSet which is of type **Sys_Rpc_SettimeofdayRequest__Timeval**.

```
ReqTimeSet = *(((SysRpc__SettimeofdayRequest*)request)->timeval_s);
```

Using ReqTimeSet, we now store values of settimeofdayRequest parameters tv_sec and tv_usec to tv.tv_sec and tv.tv_usec.

```
   tv.tv_sec = ReqTimeSet.tv_sec;

   tv.tv_usec = ReqTimeSet.tv_usec;
```

Now, we pass the structure timeval tv to settimeofday function and store the value returned by this function in status.

```
   int status = settimeofday(&tv, NULL);
```

Then, we need to store the return value and errno to the response. The response parameter is converted to (SysRpc_SettimeofdayResponse*) since it is a void type pointer. Then, the return_value and errno_alt parameters in setTimeResponse(here it is pointed by SysRpc_SettimeofdayResponse* response), and the function returns the status value, which should be zero if the settimeofday() is executed successfully.

```
   ((SysRpc__SettimeofdayResponse*)response)->return_value = status;

   ((SysRpc__SettimeofdayResponse*)response)->errno_alt = errno;
```

We return to the MQTT_EVENT_DATA block after exiting x_settimeofday().

The variable ret stores return variable of x_settimeofday(), which should be zero if the settimeofday() executes successfully. The next if statement checks whether the value of ret is zero, which indicates successful operation. If it is zero, it moves on to assign the response setTimeResponse which was set inside x_settimeofday() and stores its reference to toSend.settimeresponse.

```
   toSend.settimeresponse = &setTimeResponse;
```

Next, the parameters of other variables are set to fill the message(I think that it’s unnecessary).

```
//store values to settimerequest

ReqTimeSet.tv_sec = recvd->settimerequest->timeval_s->tv_sec;

ReqTimeSet.tv_usec = recvd->settimerequest->timeval_s->tv_usec;

setTimeRequest.timeval_s = &ReqTimeSet;

toSend.settimerequest = &setTimeRequest;
```

This sets the values of original settime request tv_sec and tv_usec to the response. As said earlier, first a variable ReqTImeSet of type **Sys_Rpc_SettimeofdayRequest__Timeval** is initialized. Then values are stored, and its reference is passed to setTimeRequest.timeval_s.

Then the reference of setTimeRequest is passed to toSend.settimerequest to complete assigning parameters of submessage field SettimeofdayRequest.

SImilary, other fields are set with zero value:

```
//store values to gettimeresponse

 RespTimeGet.tv_sec = 0; //get time from gettimeofday(), update both
tv_sec and tv_usec

 RespTimeGet.tv_usec = 0;

 getTimeResponse.timeval_r = &RespTimeGet;

 getRespStatus.return_value = 0;

 getRespStatus.errno_alt = 0;

 getTimeResponse.status = &getRespStatus;

 toSend.gettimeresponse = &getTimeResponse;
```

Lastly, the message must be packed before sending.The length of packed message is calculated using the sys_rpc__x_rpc_message__get_packed_size() function.

```
len2 = sys_rpc__x_rpc_message__get_packed_size(&toSend);
```

Then memory is allocated to buffer2 to store the serialized message.

```
buffer2 = malloc(len2);
```

FInally , using the sys_rpc_x_rpc_message__pack() function, the message is serialized.

```
sys_rpc__x_rpc_message__pack(&toSend, buffer2);
```

Finally, the message is published and logged.

```
msg_id = esp_mqtt_client_publish(client, "101/xRPC_Response",
((char*)buffer2), len2, 0, 0);

ESP_LOGI(TAG, "sent publish response successful, msg_id=%d", msg_id);
```

In the same way, we can handle the gettimeofday requests.

Structure of toSend:

The general structure of toSend message with the datatypes of each is given below:

toSend (Sys_Rpc__XRPCMessage)

- mes_type (Sys_Rpc__XRPCMessageType)
    - type- enum('request' , 'response')
    - procedure- enum('gettimeofday' , 'settimeofday')
- settimerequest (Sys_Rpc__SettimeofdayRequest)
    - timeval_s (Sys_Rpc__SettimeofdayRequest__Timeval)
        - tv_sec
        - tv_usec
- settimeresponse (Sys_Rpc__SettimeofdayResponse)
    - return_value
    - errno_alt
- gettimeresponse (Sys_Rpc_GettimeofdayResponse)
    - timeval_r (Sys_Rpc_GettimeofdayResponse__Timeval)
        - tv_sec
        - tv_usec
    - status (Sys_Rpc_GettimeofdayResponse__GettimeofdayRequestStatus)
        - return_value
        - errno_alt

To define mes_type.type = 'request': SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE_request

To define mes_type.type = 'response': SYS_RPC__X_RPC_MESSAGE_TYPE__TYPE_response

To define mes_type.procedure = 'gettimeofday':
SYS_RPC__X_RPC_MESSAGE_TYPE__PROCEDURE_settimeofday

To define mes_type.procedure = 'gettimeofday':
SYS_RPC__X_RPC_MESSAGE_TYPE__PROCEDURE_settimeofday

PS: The variable names here are from an earlier version of the project, so the names of protocol buffer files may vary. However, the methodology shown here is exactly the same. This document was developed with the intent to clear up things regarding usage of Protocol Buffers in C; since it is not clearly documented as it is for C++, Python, Java, etc.