Faculty of Engineering
and Architecture

Department of Information Technology
Internet Based Communication Networks

How to use Google Protobuffers for the EVARILOS benchmarking suite

Tom Van Haute

1 Introduction

The EVARILOS project addresses one of the major problems of indoor localization research: The pitfall to reproduce research results in real life scenarios suffering from uncontrolled RF interference and the weakness of numerous published solutions being evaluated under individual, not comparable and not repeatable conditions.

EVARILOS Accurate and robust indoor localization is a key enabler for context-aware Future Internet applications, whereby robust means that the localization solutions should perform well in diverse physical indoor environments under realistic RF interference conditions.

One of the major activities is the Open Challenge, a RF-based indoor localization competition in which participating teams compete in precise localization under the influence of different RF-interference scenarios. The competing localization solutions will be evaluated following the EVARILOS benchmarking methodology, aligned with the upcoming ISO/IEC 18305 standard: Test and evaluation of localization and tracking systems.

2 The EVARILOS Benchmarking Platform

In order to successfully complete this open challenge, an EVARILOS benchmarking platform will be provided. This is a complex system that integrates a number of different hardware and software systems. To guarantee interoperability and extensibility under these conditions, there is a need of an efficient, language and platform neutral way for serialization of the structured processed and raw data for use in communication protocols between the different components, as well as, for persistence in data storage. To this end, the EVARLILOS platform leverages the open source Protocol Buffers [1] framework from Google, that offers automatic, flexible and efficient encoding of structured data.

3 The Google Protocol Buffer

3.1 What are protocol buffers?

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data (think XML, but smaller, faster, and simpler). 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: Java, C++, or Python.

[1] The Google Protocol Buffer

3.2 How do they work?

You specify how you want the information you're serializing to be structured by defining protocol buffer message types in `.proto` files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. Here's a very basic example of a `.proto` file that defines a message containing information about a person:

Listing 1: .proto example structure

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

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

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

  repeated PhoneNumber phone = 4;
}
```

As you can see, the message format is simple each message type has one or more uniquely numbered fields, and each field has a name and a value type, where value types can be numbers (integer or floating-point), booleans, strings, raw bytes, or even (as in the example above) other protocol buffer message types, allowing you to structure your data hierarchically. You can specify optional fields, required fields, and repeated fields.

Once you've defined your messages, you run the protocol buffer compiler for your application's language on your `.proto` file to generate data access classes. These provide simple accessors for each field (like `query()` and `set_query()`) as well as methods to serialize/parse the whole structure to/from raw bytes so, for instance, if your chosen language is C++, running the compiler on the above example will generate a class called Person. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages. You might then write some code like this:

Listing 2: C++ example: serialize

```
Person person;
person.set_name(``John Doe'');
```

```
person.set_id(1234);
person.set_email(``jdoe@example.com'');
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Then, later on, you could read your message back in:

Listing 3: C++ example: deserialize

```
fstream input(``myfile'', ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << ``Name: '' << person.name() << endl;
cout << ``E-mail: '' << person.email() << endl;
```

You can add new fields to your message formats without breaking backwards-compatibility; old binaries simply ignore the new field when parsing. So if you have a communications protocol that uses protocol buffers as its data format, you can extend your protocol without having to worry about breaking existing code.

4 Protocol Buffers in the EVARILOS Benchmarking Suite

4.1 Create your .proto-file

The first step for using the Protocol Buffers is to define the data structure in a `.proto`-file. As an example, the `raw_rssi.proto` is given in the listing below:

Listing 4: raw_rssi.proto

```
package evarilos;

message RawRSSIReading {
     optional string sender_id = 1;
     optional string sender_bssid = 2;
     optional string receiver_id = 3;
     optional string receiver_bssid = 4;
     optional string frequency = 5;
     required int32 rssi = 6;
     optional float lqi = 7;
     optional int64 timestamp_utc = 8;
     required int32 run_nr = 9;
     optional bool is_ack = 10 [default = false];
     optional Location sender_location = 11;
     optional Location receiver_location = 12;
```

```
    message Location {
        optional double coordinate_x = 1;
        optional double coordinate_y = 2;
        optional double coordinate_z = 3;
        optional string room_label = 4;
        optional string node_label = 5;
    }
}

message RawRSSIReadingCollection {
    required string metadata_id = 1;
    optional string receiver_id = 2;
    optional string sender_id = 3;
    repeated RawRSSIReading rawRSSI = 4;
    required string data_id = 5;
    optional int32 seq_nr = 6 [default = 1];
    optional bytes _id = 7;
    optional int64 timestamp_utc_start = 8;
    optional int64 timestamp_utc_stop = 9;
}
```

This `.proto`-file describes a *RawRSSIReadingCollection*, a collection of RAW RSSI measurements. In the lower message, there are some `required`, `optional` and `repeated` fields. These types could be `strings`, `bytes`, integers (`int32`), etc. Not only primary types are available, we can also create our own type, in this example: `RawRSSIReading` was created, it is defined in the message above. This contains again a list of types that defines this message.

More information about the `.proto` messages can be found on the developers guide[2] of Google Protocol Buffer.

4.2 Compile your .proto-file

The next step towards using the Protocol Buffers is compiling the `.proto`-file we just created. With the protocol buffer compiler `protoc`, we can compile the `.proto`-file to a class-file in Java, C/C++ or Python. In order to install the compiler: install the installation binary and follow the `readme` instructions[3].

Once the compiler is installed successfully, we can compile our `.proto-file` using the follwing command:

[2] Details of the .proto structure
[3] Protocol Buffer Compiler download page

Listing 5: Compile to C++, JAVA or Python

```
protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=
   DST_DIR --python_out=DST_DIR path/to/file.proto
```

- `IMPORT_PATH` specifies a directory in which to look for `.proto` files when resolving import directives. If omitted, the current directory is used. Multiple import directories can be specified by passing the `--proto_path` option multiple times; they will be searched in order. `-I=IMPORT_PATH` can be used as a short form of `--proto_path`.
- We can provide one or more *output directives*:
  - `--cpp_out` generates C++ code in `DST_DIR`. See the C++ reference for more.
  - `--java_out` generates Java code in `DST_DIR`. See Java reference for more.
  - `--python_out` generates Python code in `DST_DIR`. See the Python reference.
- We must provide one or more `.proto` files as input. Multiple `.proto` files can be specified at once. Although the files are named relative to the current directory, each file must reside in one of the `IMPORT_PATH`s so that the compiler can determine its canonical name.

The generate class file (in this example `RawRSSI.java`) should start with:

Listing 6: Header of the generated class file (Java example)

```
// Generated by the protocol buffer compiler.  DO NOT EDIT!
// source: raw_rssi.proto

package evarilos;

public final class RawRssi {
...
```

It is already mentioned in the listing above, if we need to change the structure, adapt the `.proto`-file and recompile it in stead of changing the auto-generated file.

4.3 Use the generated class file

The final step is using the auto-generated file that was created in the subsection above. In this tutorial, we will continue using the Java-version (`RawRSSI.java`). Before start coding, the Java-file should be included in the project, and the `protobuf-java-2.5.0.jar` library needs to be included as well. Then, we can start creating our own protocol buffer objects.

Normally, we can create an object by using the constructor of the class: `Object o = new Object();`. With the protocol buffers, to create an object of a message, we use:

Listing 7: Creating an object of a message

```
RawRSSIReading.Builder rawRssi = RawRSSIReading.newBuilder();
```

Once the object is created, we can use the provided getters and setters of the attributes:

Listing 8: Setting the primary attributes

```
rawrssi.setRunNr(5);
rawrssi.setReceiverId(12);
rawrssi.setRssi(-78);
rawrssi.setSenderId(4);
```

As you can see in Listing 4, the `RawRSSIReading` message can optionally contain a `Location` message (for sender and receiver). Assume that we would like to add the location details of the sender, then we need to create a `Location`-object first:

Listing 9: Setting the primary attributes

```
RawRSSIReading.Location.Builder loc =
                                RawRSSIReading.Location.newBuilder();

loc.setCoordinateX(10);
loc.setCoordinateY(18);

rawrssi.setSenderLocation(loc);
```

The `set`-method can be used when the attributes are `required` or `optional`. But when the attributes is `repeated`, we have to use the `add`-method. For example: one `RawRssiCollection` has multiple `RawRssiReadings`:

Listing 10: Adding one or more attributes

```
RawRSSIReadingCollection.Builder rawData =
                                RawRSSIReadingCollection.newBuilder();

rawData.addRawRSSI(rawrssi);
rawData.addRawRSSI(rawrssi2);
rawData.addRawRSSI(...);
```

Finally, when the object is filled with the necessary data, we can finalize it by executing the `build()` command. Then, only the getters are accessible.

Listing 11: Building the object

```
RawRSSIReadingCollection finalCollection = rawData.build();
```

4.4 Sending and receiving a Protobuffer

The final step is sending or receiving protobuffers. E.g. using a file or HTTP requests, etc. The technique is very straight forward. The example code (in Java) is given below.

If we want to write the object to a file, we just have to use the `writeTo(<outputStream>)` function:

Listing 12: Write protobuffer to a file

```
FileOutputStream output = new FileOutputStream("myproto.rawData");
rawData.build().writeTo(output);
output.close();
```

There is a small difference if we want to use for example a HTTP request. Then we need to convert the object to a byte array and then write those bytes:

Listing 13: Write protobuffer to a stream

```
byte[] protoBytes = rawData.build().toByteArray();
OutputStream responseBody = exchange.getResponseBody();
responseBody.write(protoBytes);
responseBody.close();
```

In the other way, we can also read from an inputstream. It doesn't make any difference if the inputstream is from a file or a `HttpUrlConnection`).

Listing 14: Reading protobuffers from an inputstream

```
FileInputStream in = new FileInputStream("myproto.rawData");
RawRSSIReadingCollection collection =
                         RawRSSIReadingCollection.parseFrom(in);
in.close();
```

To make this tutorial complete, after parsing the object, we can use the getters and further manipulate the data. If the attribute is repeated, a `list` is used:

Listing 15: Using the data

```
string dataId = collection.getDataId();
int seqNr = collection.getSeqNr();
List<RawRSSIReading> readings = collection.getRawRSSIReadings();
```