INTERNET-DRAFT
Intended Status: Proposed Standard
Expires: April 11, 2013

S. Stuart
Google
R. Fernando
Cisco

October 8, 2012

Encoding rules and MIME type for Protocol Buffers
draft-rfernando-protocol-buffers-00

Abstract

This document describes the encoding format for Protocol Buffers encoded data and registers a MIME type associated with Protocol Buffers encoded data.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Copyright and License Notice

Table of Contents

1 Introduction

Protocol buffers, referred to as protobuf in this document, is a commonly used interchange format to serialize structured data for storage and transmission between applications and systems. It supports simple and composite data types and provides rules to serialize those data types into a portable format that is both language and platform neutral. Since it encodes data into binary format, it is fast and efficient. It is also supported by a wide variety of programming languages.

While protocol buffers has gained wide spread use, it has so far been described only informally and has not been standardized. This document specifies the encoding rules for protobuf and registers the MIME type 'application/protobuf' for it in accordance with RFC 2048.

This document heavily borrows ideas from web page [GPBENC].

1.1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Message Structure

Protobuf defines all data elements in discrete units called "messages" [GPBOVW]. A message is a logical collection of related data items. It is similar to a "record" or a "structure" in a traditional programming language. Many standard simple data types are available as field types, including bool, int32, float, double and string. One can also add further structure to the outer message by using enums and other messages as field types.

The following is an example of a message definition in protobuf:

```
  message Person {

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

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

```
        message PhoneNumber {
            required string number = 1;
            optional PhoneType type = 2;
        }

        repeated PhoneNumber phone = 4;
    }
```

Note the presence of simple data types such as strings and int32s as well as complex data types such as enums and messages in the above message definition.

Each field is annotated with one of the following three modifiers:

1. required: a value for the field must be provided, otherwise the message will be considered malformed and the decoding entity will throw and exception.

2. optional: the field may or may not be set. If an optional field value isn't set, a default value is used.

3.repeated: the field may be repeated any number of times (including zero). The order of the repeated values will be preserved in the protocol buffer encoding.

The integer token to the right of the assignment operator is a field number. These field numbers uniquely identify a field in a message and together with the wire type is used to form the key for the key-value pairs in the serialized data stream. Field numbers 1-15 require one less byte to encode than higher numbers, so as an optimization one can decide to use those field numbers for the commonly used or repeated elements. Each element in a repeated field requires re-encoding the field number, so repeated fields are particularly good candidates for this optimization.

This document will not describe every syntactic element of the protbuf language but will restrict discussion to only those elements that are relevant to the encoding and decoding of data types.

3. Encoding Rules

This section describes the encoding rules for the different field types.

3.1 Numbers as VarInts

To understand protobuf encoding, we need to first understand VarInts.

All numbers in protobuf are represented as base 128 variable-length integers (or VarInt). VarInt is an encoding scheme that uses only as many bytes as is necessary to represent a number and it can be used to encode arbitrary large numbers. It achieves this by using a continuation bit in every byte. Each byte in a VarInt, except the last byte, has the most significant bit (msb) set indicating that there are more bytes to come. The last byte has the msb set to zero. The stream of 7-byte quantities (after msb has been removed) are then reversed and concatenated to produce one single binary representation of the number.

3.2 Encoding and Interpretation of Protobuf Messages

Protobuf messages are not self describing. In other words, the entity decoding the binary representation of the message needs to refer to the equivalent text definition of the message to interpret the fields. The "tag" that's associated with the field (with the "=" sign in the text definition) indicates to the decoder which field it is looking at currently.

To achieve backward compatibility a wire-type is also included for every field. Using the wire-type, the decoder can skip a field without interpreting it if it desires to do so. This can be useful to achieve backward compatibility when the decoder is not aware of a particular field's tag value.

Every field is encoded as a (key, value) pair. The key is a VarInt with the value ((field-tag << 3) | wire-type). In other words, the last three bits of the key VarInt is the wire type.

3.3 Wire Types

This document defines the following wire types, their interpretation and the data types that they are used for.

3.3.1 Wire Type 0

If the wire type is 0, the value field is simply a VarInt. This encoding is used to represent int32, int64, uint32, uint64, sint32, sint64, bool and enum. For positive integers the interpretation of the VarInt is straight forward as explained in section 3.1.

For example, consider the following message,

```
message Test1 {
    required int32 a = 1;
}
```

would be serialized as '08 96 01'.

If int32 and int64 are used for encoding negative integers, the resulting VarInt is always a ten byte quantity (effectively treating it as a large unsigned integer). If a singed type is used, a zigzag encoding scheme is used which assigns small VarInt values for small negative numbers. In this scheme, the numbers -2, -1, 0, 1, 2 would be represented as VarInts 3, 1, 0, 2, 4 and so on. Mathematically, each value 'n' is encoded using (n << 1) ^ (n >> 31) for sint32 or (n << 1) ^ (n >> 63) for sint64.

3.3.2 Wire Type 1

This is a fixed length 64-bit quantity. This wire type is used to represent fixed64, sfixed64 and double data types. The value is stored in little-endian format.

3.3.3 Wire Type 2

This is a length delimited stream of bytes. The value field is a VarInt encoded length followed by the specified number of bytes of data.

As an example, consider the following message,

```
message Test2 {
    required string b = 2;
}
```

would be serialized as, '12 0b 68 65 6c 6c 6f 20 77 6f 72 6c 64', if the string 'b' was set to "Hello World".

3.3.4 Wire Type 5

This is a fixed length 32-bit quantity. This wire type is used to represent fixed32, sfixed32 and float data types. The value is stored in little-endian format.

4. Embedded Messages

Embedded messages are encoded as follows. The inner (or the embedded) message is serialized first using the rules described above. The resultant byte stream is then treated as a Wire Type 2 field in the outer message and added to its encoding.

Consider the example,

```
message Test1 {
    required int32 foo = 1;
}

message Test2 {
    required Test1 c = 3;
}
```

If the field 'foo' were to take the value 150, the resultant encoded byte stream for the inner message would be 08 '96 01'. And for Test2 would be '1a 03 08 96 01'.

5. Optional and Repeated Elements

If the message definition has 'repeated' elements, then the encoded message has zero or more key-value pairs with the same field number. These repeated values do not have to appear consecutively; they may be interleaved with other fields.

If the message definition has 'optional' elements, then the encoded message may or may not have a key-value pair with that field number.

A repeated field could be a 'packed repeated field' in which case the encoding for the field is slightly different. A packed repeated field containing zero elements does not appear in the encoded message. Otherwise, all of the elements of the field are packed into a single key-value pair with the wire type 2 (length delimited). Each element is encoded the same way it would be normally, except without a field number preceding it.

6. Field Order

When a message is serialized its known fields should be written sequentially by field number. This allows parsing code to use optimizations that rely on field numbers being in sequence. However, protocol buffer parsers must be able to parse fields in any order, as not all messages are created by simply serializing an object - for

instance, it's sometimes useful to merge two messages by simply concatenating them.

7. IANA Considerations

The MIME media type for protobuf messages is application/protobuf.

Type name: application

Subtype name: protobuf

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: 8 bit binary, UTF-8

Security considerations:
Generally there are security issues with serialization formats if code is transmitted and executed on the decoder end. Since protobuf binary encoding does not carry code, we consider the encoding scheme itself to not introduce any security risks.

8. Security Considerations

See section 7.

9. Acknowledgements

We thank the engineers at Google for giving us the protocol buffers serialization format. All the concepts described in this document come from web pages [GPBENC, GPBOVW] defining protocol buffer mechanisms. This document is merely an attempt to standardize those mechanisms in IETF and assign a MIME type for protobuf encoded messages.

10. References

10.1 Informative References

[GPBENC] Google Protocol Buffer Encoding, https://developers.google.com/protocol-buffers/docs/encoding

[GPBOVW] Google Protocol Buffer Overview, https://developers.google.com/protocol-buffers/docs/overview

Authors' Addresses

Stephen Stuart
Google
1600 Amphitheatre Parkway
Mountain View, CA 94043
USA

EMail: sstuart@google.com

Rex Fernando
Cisco Systems
170 W. Tasman Dr.
San Jose, CA 95134

Email: rex@cisco.com