Developer Guide

Welcome to the developer documentation for protocol buffers – a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.

This documentation is aimed at Java, C++, or Python developers who want to use protocol buffers in their applications. This overview introduces protocol buffers and tells you what you need to do to get started – you can then go on to follow the tutorials or delve deeper into protocol buffer encoding. API reference documentation is also provided for all three languages, as well as language and style guides for writing `.proto` files.

What are protocol buffers?

Protocol buffers are a flexible, efficient, automated 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. You can even update your data structure without breaking deployed programs that are compiled against the "old" format.

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:

```
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. You can find more information about writing `.proto` files in the Protocol Buffer Language Guide.

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:

```
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:

```
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.

You'll find a complete reference for using generated protocol buffer code in the API Reference section, and you can find out more about how protocol buffer messages are encoded in Protocol Buffer Encoding.

Why not just use XML?

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler
- are 3 to 10 times smaller
- are 20 to 100 times faster
- are less ambiguous
- generate data access classes that are easier to use programmatically

For example, let's say you want to model a `person` with a `name` and an `email`. In XML, you need to do:

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

while the corresponding protocol buffer message (in protocol buffer text format) is:

```
# Textual representation of a protocol buffer.
# This is *not* the binary format used on the wire.
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

When this message is encoded to the protocol buffer binary format (the text format above is just a convenient human-readable representation for debugging and editing), it would probably be 28 bytes long and take around 100-200 nanoseconds to parse. The XML version is at least 69 bytes if you remove whitespace, and would take around 5,000-10,000 nanoseconds to parse.

Also, manipulating a protocol buffer is much easier:

```
  cout << "Name: " << person.name() << endl;
  cout << "E-mail: " << person.email() << endl;
```

Whereas with XML you would have to do something like:

```
  cout << "Name: "
       << person.getElementsByTagName("name")->item(0)->innerText()
       << endl;
  cout << "E-mail: "
       << person.getElementsByTagName("email")->item(0)->innerText()
       << endl;
```

However, protocol buffers are not always a better solution than XML – for instance, protocol buffers would not be a good way to model a text-based document with markup (e.g. HTML), since you cannot easily interleave structure with text. In addition, XML is human-readable and human-editable; protocol buffers, at least in their native format, are not. XML is also – to some extent – self-describing. A protocol buffer is only meaningful if you have the message definition (the `.proto` file).

Sounds like the solution for me! How do I get started?

Download the package – this contains the complete source code for the Java, Python, and C++ protocol buffer compilers, as well as the classes you need for I/O and testing. To build and install your compiler, follow the instructions in the README.

Once you're all set, try following the tutorial for your chosen language – this will step you through creating a simple application that uses protocol buffers.

A bit of history

Protocol buffers were initially developed at Google to deal with an index server request/response protocol. Prior to protocol buffers, there was a format for requests and responses that used hand marshalling/unmarshalling of requests and responses, and that supported a number of versions of the protocol. This resulted in some very ugly code, like:

```
if (version == 3) {
  ...
} else if (version > 4) {
  if (version == 5) {
    ...
  }
  ...
}
```

Explicitly formatted protocols also complicated the rollout of new protocol versions, because developers had to make sure that all servers between the originator of the request and the actual server handling the request understood the new protocol before they could flip a switch to start using the new protocol.

Protocol buffers were designed to solve many of these problems:

- New fields could be easily introduced, and intermediate servers that didn't need to inspect the data could simply parse it and pass through the data without needing to know about all the fields.
- Formats were more self-describing, and could be dealt with from a variety of languages (C++, Java, etc.)

However, users still needed to hand-write their own parsing code.

As the system evolved, it acquired a number of other features and uses:

- Automatically-generated serialization and deserialization code avoided the need for hand parsing.
- In addition to being used for short-lived RPC (Remote Procedure Call) requests, people started to use protocol buffers as a handy self-describing format for storing data persistently (for example, in Bigtable).
- Server RPC interfaces started to be declared as part of protocol files, with the protocol compiler generating stub classes that users could override with actual implementations of the server's interface.

Protocol buffers are now Google's *lingua franca* for data – at time of writing, there are 48,162 different message types defined in the Google code tree across 12,183 `.proto` files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems.

Last updated 四月 *2, 2012.*