

RProtoBuf: An R API for Protocol Buffers

Romain François

Dirk Eddelbuettel

Murray Stokely

Version 0.4.4 as of July 10, 2016

Abstract

Protocol Buffers is a software project by Google that is used extensively internally and also released under an Open Source license. It provides a way of encoding structured data in an efficient yet extensible format. Google formally supports APIs for C++, Java and Python.

This vignette describes version 0.4.4 of the RProtoBuf package which brings support for protocol buffer messages to R.

Contents

1	Protocol Buffers	2
2	Basic use: Protocol Buffers and R	3
2.1	Importing proto files dynamically	3
2.2	Creating a message	4
2.3	Access and modify fields of a message	4
2.4	Display messages	5
2.5	Serializing messages	5
2.6	Parsing messages	6
3	Classes, Methods and Pseudo Methods	7
3.1	messages	7
3.2	message descriptors	19
3.3	field descriptors	25
3.4	enum descriptors	30
3.5	enum value descriptors	34
3.6	file descriptors	36
3.7	service descriptors	42
3.8	method descriptors	42

4	Utilities	42
4.1	coercing objects to messages	42
4.2	completion	43
4.3	with and within	43
4.4	identical	44
4.5	merge	44
4.6	P	44
5	Advanced Features	45
5.1	Extensions	45
5.2	Descriptor lookup	45
5.3	64-bit integer issues	46
5.4	Deprecated Feature: Protocol Buffer Groups	47
6	Other approaches	47
7	Plans for future releases	48
8	Acknowledgments	48

1 Protocol Buffers

Protocol buffers are a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more.

Protocol Buffers offer key features such as an efficient data interchange format that is both language- and operating system-agnostic yet uses a lightweight and highly performant encoding, object serialization and de-serialization as well data and configuration management. Protocol buffers are also forward compatible: updates to the proto files do not break programs built against the previous specification.

While benchmarks are not available, Google states on the project page that in comparison to XML, protocol buffers are at the same time *simpler*, between three to ten times *smaller*, between twenty and one hundred times *faster*, as well as less ambiguous and easier to program.

The protocol buffers code is released under an open-source (BSD) license. The protocol buffer project (<http://code.google.com/p/protobuf/>) contains a C++ library and a set of runtime libraries and compilers for C++, Java and Python.

With these languages, the workflow follows standard practice of so-called Interface Description Languages (IDL) (c.f. [Wikipedia on IDL](#)). This consists of compiling a protocol buffer description file (ending in .proto) into language specific classes that can be used to create, read, write and manipulate protocol buffer messages. In other words, given the 'proto' description file, code is automatically generated for the chosen target language(s). The project page contains a tutorial for each of these officially supported languages: <http://code.google.com/apis/protocolbuffers/docs/tutorials.html>

Besides the officially supported C++, Java and Python implementations, several projects have been created to support protocol buffers for many languages. The list of known languages to support protocol buffers is compiled as part of the project page: <http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns>

The protocol buffer project page contains a comprehensive description of the language: <http://code.google.com/apis/protocolbuffers/docs/proto.html>

2 Basic use: Protocol Buffers and R

This section describes how to use the R API to create and manipulate protocol buffer messages in R, and how to read and write the binary *payload* of the messages to files and arbitrary binary R connections.

2.1 Importing proto files dynamically

In contrast to the other languages (Java, C++, Python) that are officially supported by Google, the implementation used by the RProtoBuf package does not rely on the `protoc` compiler (with the exception of the two functions discussed in the previous section). This means that no initial step of statically compiling the proto file into C++ code that is then accessed by R code is necessary. Instead, proto files are parsed and processed *at runtime* by the protobuf C++ library—which is much more appropriate for a dynamic language.

The `readProtoFiles` function allows importing proto files in several ways.

```
> args( readProtoFiles )
function (files, dir, package = "RProtoBuf", pattern = "\\..proto$",
  lib.loc = NULL)
NULL
```

Using the `file` argument, one can specify one or several file paths that ought to be proto files.

```
> proto.dir <- system.file( "proto", package = "RProtoBuf" )
> proto.file <- file.path( proto.dir, "addressbook.proto" )
```

```
> readProtoFiles( proto.file )
```

With the `dir` argument, which is ignored if the file is supplied, all files matching the `.proto` extension will be imported.

```
> dir( proto.dir, pattern = "\\..proto$", full.names = TRUE )
[1] "/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto"
[2] "/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/helloworld.proto"
[3] "/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/rexp.proto"
```

```
> readProtoFiles( dir = proto.dir )
```

Finally, with the `package` argument (ignored if `file` or `dir` is supplied), the function will import all `.proto` files that are located in the `proto` sub-directory of the given package. A typical use for this argument is in the `.onLoad` function of a package.

```
> readProtoFiles( package = "RProtoBuf" )
```

Once the `proto` files are imported, all message descriptors are available in the R search path in the `RProtoBuf:DescriptorPool` special environment. The underlying mechanism used here is described in more detail in section 5.2.

```
> ls( "RProtoBuf:DescriptorPool" )
[1] "rexp.CMPLX"           "rexp.REXP"
[3] "rexp.STRING"          "rprotobuf.HelloWorldRequest"
[5] "rprotobuf.HelloWorldResponse" "tutorial.AddressBook"
[7] "tutorial.Person"
```

2.2 Creating a message

The objects contained in the special environment are descriptors for their associated message types. Descriptors will be discussed in detail in another part of this document, but for the purpose of this section, descriptors are just used with the `new` function to create messages.

```
> p <- new( tutorial.Person, name = "Romain", id = 1 )
```

2.3 Access and modify fields of a message

Once the message is created, its fields can be queried and modified using the dollar operator of R, making protocol buffer messages seem like lists.

```
> p$name
[1] "Romain"
> p$id
[1] 1
> p$email <- "francoisromain@free.fr"
```

However, as opposed to R lists, no partial matching is performed and the name must be given entirely.

The `[[` operator can also be used to query and set fields of a messages, supplying either their name or their tag number :

```
> p[["name"]] <- "Romain Francois"
> p[[ 2 ]] <- 3
> p[[ "email" ]]
[1] "francoisromain@free.fr"
```

Protocol buffers include a 64-bit integer type, but R lacks native 64-bit integer support. A workaround is available and described in Section 5.3 for working with large integer values.

2.4 Display messages

Protocol buffer messages and descriptors implement `show` methods that provide basic information about the message :

```
> p
message of type 'tutorial.Person' with 3 fields set
```

For additional information, such as for debugging purposes, the `as.character` method provides a more complete ASCII representation of the contents of a message.

```
> writeLines( as.character( p ) )
name: "Romain Francois"
id: 3
email: "francoisromain@free.fr"
```

2.5 Serializing messages

However, the main focus of protocol buffer messages is efficiency. Therefore, messages are transported as a sequence of bytes. The `serialize` method is implemented for protocol buffer messages to serialize a message into the sequence of bytes (raw vector in R speech) that represents the message.

```
> serialize( p, NULL )
[1] 0a 0f 52 6f 6d 61 69 6e 20 46 72 61 6e 63 6f 69 73 10 03 1a
[21] 16 66 72 61 6e 63 6f 69 73 72 6f 6d 61 69 6e 40 66 72 65 65
[41] 2e 66 72
```

The same method can also be used to serialize messages to files :

```
> tf1 <- tempfile()
> tf1
[1] "/tmp/Rtmp1ALtkK/file1cec2a26bccf"
> serialize( p, tf1 )
> readBin( tf1, raw(0), 500 )
[1] 0a 0f 52 6f 6d 61 69 6e 20 46 72 61 6e 63 6f 69 73 10 03 1a
[21] 16 66 72 61 6e 63 6f 69 73 72 6f 6d 61 69 6e 40 66 72 65 65
[41] 2e 66 72
```

Or to arbitrary binary connections:

```

> tf2 <- tempfile()
> con <- file( tf2, open = "wb" )
> serialize( p, con )
> close( con )
> readBin( tf2, raw(0), 500 )
[1] 0a 0f 52 6f 6d 61 69 6e 20 46 72 61 6e 63 6f 69 73 10 03 1a
[21] 16 66 72 61 6e 63 6f 69 73 72 6f 6d 61 69 6e 40 66 72 65 65
[41] 2e 66 72

```

serialize can also be used in a more traditional object oriented fashion using the dollar operator :

```

> p$serialize( tf1 )
> con <- file( tf2, open = "wb" )
> p$serialize( con )
> close( con )

```

2.6 Parsing messages

The RProtoBuf package defines the read function to read messages from files, raw vector (the message payload) and arbitrary binary connections.

```

> args( read )
function (descriptor, input)
NULL

```

The binary representation of the message (often called the payload) does not contain information that can be used to dynamically infer the message type, so we have to provide this information to the read function in the form of a descriptor :

```

> message <- read( tutorial.Person, tf1 )
> writeLines( as.character( message ) )
name: "Romain Francois"
id: 3
email: "francoisromain@free.fr"

```

The input argument of read can also be a binary readable R connection, such as a binary file connection:

```

> con <- file( tf2, open = "rb" )
> message <- read( tutorial.Person, con )
> close( con )
> writeLines( as.character( message ) )

```

```
name: "Romain Francois"
id: 3
email: "francoisromain@free.fr"
```

Finally, the payload of the message can be used :

```
> payload <- readBin( tf1, raw(0), 5000 )
> message <- read( tutorial.Person, payload )
```

read can also be used as a pseudo method of the descriptor object :

```
> message <- tutorial.Person$read( tf1 )
> con <- file( tf2, open = "rb" )
> message <- tutorial.Person$read( con )
> close( con )
> message <- tutorial.Person$read( payload )
```

3 Classes, Methods and Pseudo Methods

The RProtoBuf package uses the S4 system to store information about descriptors and messages, but the information stored in the R object is very minimal and mainly consists of an external pointer to a C++ variable that is managed by the proto C++ library.

```
> str( p )
Formal class 'Message' [package "RProtoBuf"] with 2 slots
  ..@ pointer:<externalptr>
  ..@ type   : chr "tutorial.Person"
```

Using the S4 system allows the RProtoBuf package to dispatch methods that are not generic in the S3 sense, such as new and serialize.

The RProtoBuf package combines the *R typical* dispatch of the form `method(object, arguments)` and the more traditional object oriented notation `object$method(arguments)`.

3.1 messages

Messages are represented in R using the Message S4 class. The class contains the slots pointer and type as described on the Table 1.

Although the RProtoBuf package uses the S4 system, the @ operator is very rarely used. Fields of the message are retrieved or modified using the \$ or [[operators as seen on the previous section, and pseudo-methods can also be called using the \$ operator. Table 2 describes the methods defined for the Message class :

slot	description
pointer	external pointer to the Message object of the C++ proto library. Documentation for the Message class is available from the protocol buffer project page: http://code.google.com/apis/protocolbuffers/docs/reference/cpp/google.protobuf.message.html#Message
type	fully qualified path of the message. For example a Person message has its type slot set to <code>tutorial.Person</code>

Table 1: Description of slots for the Message S4 class

3.1.1 Retrieve fields

The `$` and `[[` operators allow extraction of a field data.

```
> message <- new( tutorial.Person,
+   name = "foo", email = "foo@bar.com", id = 2,
+   phone = list(
+     new( tutorial.Person.PhoneNumber, number = "+33(0)...",
+     type = "HOME" ),
+     new( tutorial.Person.PhoneNumber, number = "+33(0)###",
+     type = "MOBILE" )
+   ) )
> message$name
[1] "foo"
> message$email
[1] "foo@bar.com"
> message[[ "phone" ]]
[[1]]
message of type 'tutorial.Person.PhoneNumber' with 2 fields set

[[2]]
message of type 'tutorial.Person.PhoneNumber' with 2 fields set
> message[[ 2 ]]
[1] 2
```

Neither `$` nor `[[` support partial matching of names. The `$` is also used to call methods on the message, and the `[[` operator can use the tag number of the field.

Table 3 details correspondance between the field type and the type of data that is retrieved by `$` and `[[`.

method	section	description
has	3.1.3	Indicates if a message has a given field.
clone	3.1.4	Creates a clone of the message
isInitialized	3.1.5	Indicates if a message has all its required fields set
serialize	3.1.6	serialize a message to a file or a binary connection or retrieve the message payload as a raw vector
clear	3.1.7	Clear one or several fields of a message, or the entire message
size	3.1.8	The number of elements in a message field
bytesize	3.1.9	The number of bytes the message would take once serialized
swap	3.1.10	swap elements of a repeated field of a message
set	3.1.11	set elements of a repeated field
fetch	3.1.12	fetch elements of a repeated field
setExtension	3.1.13	set an extension of a message
getExtension	3.1.14	get the value of an extension of a message
add	3.1.15	add elements to a repeated field
str	3.1.16	the R structure of the message
as.character	3.1.17	character representation of a message
toString	3.1.18	character representation of a message (same as as.character)
as.list	3.1.19	converts message to a named R list
update	3.1.20	updates several fields of a message at once
descriptor	3.1.21	get the descriptor of the message type of this message
fileDescriptor	3.1.22	get the file descriptor of this message's descriptor

Table 2: Description of methods for the Message S4 class

3.1.2 Modify fields

The `$<-` and `[[<-` operators are implemented for Message objects to set the value of a field. The R data is coerced to match the type of the message field.

```
> message <- new( tutorial.Person,
+   name = "foo", id = 2 )
> message$email <- "foo@bar.com"
> message[["id"]] <- 2
> message[[ 1 ]] <- "foobar"
> writeLines( message$as.character() )
name: "foobar"
id: 2
email: "foo@bar.com"
```

field type	R type (non repeated)	R type (repeated)
double	double vector	double vector
float	double vector	double vector
uint32	double vector	double vector
fixed32	double vector	double vector
int32	integer vector	integer vector
sint32	integer vector	integer vector
sfixed32	integer vector	integer vector
int64	integer or character vector ¹	integer or character vector
uint64	integer or character vector	integer or character vector
sint64	integer or character vector	integer or character vector
fixed64	integer or character vector	integer or character vector
sfixed64	integer or character vector	integer or character vector
bool	logical vector	logical vector
string	character vector	character vector
bytes	character vector	character vector
enum	integer vector	integer vector
message	S4 object of class Message	list of S4 objects of class Message

Table 3: Correspondance between field type and R type retrieved by the extractors. 1. R lacks native 64-bit integers, so the `RProtoBuf.int64AsString` option is available to return large integers as characters to avoid losing precision. This option is described in Section 5.3. R also lacks an unsigned integer type.

Table 4 describes the R types that are allowed in the right hand side depending on the target type of the field.

3.1.3 Message\$has method

The `has` method indicates if a field of a message is set. For repeated fields, the field is considered set if there is at least on object in the array. For non-repeated fields, the field is considered set if it has been initialized.

The `has` method is a thin wrapper around the `HasField` and `FieldSize` methods of the `google::protobuf::Reflection` C++ class.

```
> message <- new( tutorial.Person, name = "foo" )
> message$has( "name" )
[1] TRUE
> message$has( "id" )
[1] FALSE
> message$has( "phone" )
[1] FALSE
```

internal type	allowed R types
double, float	integer, raw, double, logical
int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64	integer, raw, double, logical, character
bool	integer, raw, double, logical
bytes, string	character
enum	integer, double, raw, character
message, group	S4, of class Message of the appropriate message type, or a list of S4 objects of class Message of the appropriate message type.

Table 4: Allowed R types depending on internal field types.

3.1.4 Message\$clone method

The `clone` function creates a new message that is a clone of the message. This function is a wrapper around the methods `New` and `CopyFrom` of the `google::protobuf::Message` C++ class.

```
> m1 <- new( tutorial.Person, name = "foo" )
> m2 <- m1$clone( )
> m2$email <- "foo@bar.com"
> writeLines( as.character( m1 ) )
name: "foo"
> writeLines( as.character( m2 ) )
name: "foo"
email: "foo@bar.com"
```

3.1.5 Message\$isInitialized method

The `isInitialized` method quickly checks if all required fields have values set. This is a thin wrapper around the `IsInitialized` method of the `google::protobuf::Message` C++ class.

```
> message <- new( tutorial.Person, name = "foo" )
> message$isInitialized()
[1] FALSE
> message$id <- 2
> message$isInitialized()
[1] TRUE
```

3.1.6 Message\$serialize method

The `serialize` method can be used to serialize the message as a sequence of bytes into a file or a binary connection.

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> tf1 <- tempfile( )
> tf1
[1] "/tmp/Rtmp1ALtkK/file1cec64a7265d"
> message$serialize( tf1 )
> tf2 <- tempfile( )
> tf2
[1] "/tmp/Rtmp1ALtkK/file1cec47a01271"
> con <- file( tf2, open = "wb" )
> message$serialize( con )
> close( con )
```

The files `file1cec64a7265d` and `file1cec47a01271` both contain the message payload as a sequence of bytes. The `readBin` function can be used to read the files as a raw vector in R:

```
> readBin( tf1, raw(0), 500 )
[1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
> readBin( tf2, raw(0), 500 )
[1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
```

The `serialize` method can also be used to directly retrieve the payload of the message as a raw vector:

```
> message$serialize(NULL)
[1] 0a 03 66 6f 6f 10 02 1a 0b 66 6f 6f 40 62 61 72 2e 63 6f 6d
```

3.1.7 Message\$clear method

The `clear` method can be used to clear all fields of a message when used with no argument, or a given field.

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> writeLines( as.character( message ) )
name: "foo"
id: 2
```

```

email: "foo@bar.com"
> message$clear()
> writeLines( as.character( message ) )

> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> message$clear( "id" )
> writeLines( as.character( message ) )
name: "foo"
email: "foo@bar.com"

```

The clear method is a thin wrapper around the Clear method of the `google::protobuf::Message` C++ class.

3.1.8 Message\$size method

The size method is used to query the number of objects in a repeated field of a message :

```

> message <- new( tutorial.Person, name = "foo",
+               phone = list(
+                   new( tutorial.Person.PhoneNumber, number = "+33(0)...",
+                   type = "HOME" ),
+                   new( tutorial.Person.PhoneNumber, number = "+33(0)###",
+                   type = "MOBILE" )
+               ) )
> message$size( "phone" )
[1] 2
> size( message, "phone" )
[1] 2

```

The size method is a thin wrapper around the FieldSize method of the `google::protobuf::Reflection` C++ class.

3.1.9 Message\$bytesize method

The bytesize method retrieves the number of bytes the message would take once serialized. This is a thin wrapper around the ByteSize method of the `google::protobuf::Message` C++ class.

```

> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> message$bytesize()
[1] 20

```

```
> byteSize( message )
[1] 20
> length( message$serialize( NULL ) )
[1] 20
```

3.1.10 Message\$swap method

The swap method can be used to swap elements of a repeated field.

```
> message <- new( tutorial.Person, name = "foo",
+               phone = list(
+                 new( tutorial.Person.PhoneNumber, number = "+33(0)...",
+                 type = "HOME" ),
+                 new( tutorial.Person.PhoneNumber, number = "+33(0)###",
+                 type = "MOBILE" )
+               ) )
> message$swap( "phone", 1, 2 )
> writeLines( as.character( message$phone[[1]] ) )
number: "+33(0)###"
type: MOBILE
> writeLines( as.character( message$phone[[2]] ) )
number: "+33(0)..."
type: HOME
> swap( message, "phone", 1, 2 )
> writeLines( as.character( message$phone[[1]] ) )
number: "+33(0)..."
type: HOME
> writeLines( as.character( message$phone[[2]] ) )
number: "+33(0)###"
type: MOBILE
```

3.1.11 Message\$set method

The set method can be used to set values of a repeated field.

```
> message <- new( tutorial.Person, name = "foo",
+               phone = list(
+                 new( tutorial.Person.PhoneNumber, number = "+33(0)...",
+                 type = "HOME" ),
+                 new( tutorial.Person.PhoneNumber, number = "+33(0)###",
+                 type = "MOBILE" )
+               )
```

```

+           ) )
> number <- new( tutorial.Person.PhoneNumber,
+               number = "+33(0)--", type = "WORK" )
> message$set( "phone", 1, number )
> writeLines( as.character( message ) )
name: "foo"
phone {
  number: "+33(0)--"
  type: WORK
}
phone {
  number: "+33(0)###"
  type: MOBILE
}

```

3.1.12 Message\$fetch method

The fetch method can be used to get values of a repeated field.

```

> message <- new( tutorial.Person, name = "foo",
+               phone = list(
+                   new( tutorial.Person.PhoneNumber, number = "+33(0)...",
+                   type = "HOME" ),
+                   new( tutorial.Person.PhoneNumber, number = "+33(0)###",
+                   type = "MOBILE" )
+               ) )
> message$fetch( "phone", 1 )
[[1]]
message of type 'tutorial.Person.PhoneNumber' with 2 fields set

```

3.1.13 Message\$setExtension method

The setExtension method can be used to set an extension field of the Message.

```

> if (!exists("protobuf_unittest.TestAllTypes",
+             "RProtoBuf.DescriptorPool")) {
+   unittest.proto.file <- system.file("unitTests", "data",
+                                     "unittest.proto",
+                                     package="RProtoBuf")
+   readProtoFiles(file=unittest.proto.file)
+ }

```

```
> test <- new(protobuf_unittest.TestAllExtensions)
> test$setExtension(protobuf_unittest.optional_int32_extension,
+                   as.integer(1))
```

3.1.14 Message\$getExtension method

The `getExtension` method can be used to get values of an extension.

```
> test$getExtension(protobuf_unittest.optional_int32_extension)
[1] 1
```

3.1.15 Message\$add method

The `add` method can be used to add values to a repeated field.

```
> message <- new( tutorial.Person, name = "foo")
> phone <- new( tutorial.Person.PhoneNumber,
+             number = "+33(0)...", type = "HOME" )
> message$add( "phone", phone )
> writeLines( message$toString() )
name: "foo"
phone {
  number: "+33(0)..."
  type: HOME
}
```

3.1.16 Message\$str method

The `str` method gives the R structure of the message. This is rarely useful.

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> message$str()
Formal class 'Message' [package "RProtoBuf"] with 2 slots
 ..@ pointer:<externalptr>
 ..@ type   : chr "tutorial.Person"
> str( message )
Formal class 'Message' [package "RProtoBuf"] with 2 slots
 ..@ pointer:<externalptr>
 ..@ type   : chr "tutorial.Person"
```


3.1.17 Message\$as.character method

The `as.character` method gives the debug string of the message.

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> writeLines( message$as.character() )
name: "foo"
id: 2
email: "foo@bar.com"
> writeLines( as.character( message ) )
name: "foo"
id: 2
email: "foo@bar.com"
```

3.1.18 Message\$toString method

`toString` currently is an alias to the `as.character` function.

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> writeLines( message$toString() )
name: "foo"
id: 2
email: "foo@bar.com"
> writeLines( toString( message ) )
name: "foo"
id: 2
email: "foo@bar.com"
```

3.1.19 Message\$as.list method

The `as.list` method converts the message to an named R list

```
> message <- new( tutorial.Person, name = "foo", email = "foo@bar.com", id
= 2 )
> as.list( message )
$name
[1] "foo"

$id
```

```
[1] 2
```

```
$email
```

```
[1] "foo@bar.com"
```

```
$phone
```

```
list()
```

The names of the list are the names of the declared fields of the message type, and the content is the same as can be extracted with the `$` operator described in section 3.1.1.

3.1.20 Message\$update method

The update method can be used to update several fields of a message at once.

```
> message <- new( tutorial.Person )
> update( message,
+       name = "foo",
+       id = 2,
+       email = "foo@bar.com" )
message of type 'tutorial.Person' with 3 fields set
> writeLines( message$as.character() )
name: "foo"
id: 2
email: "foo@bar.com"
```

3.1.21 Message\$descriptor method

The descriptor method retrieves the descriptor of a message. See section 3.2 for more information about message type descriptors.

```
> message <- new( tutorial.Person )
> message$descriptor()
descriptor for type 'tutorial.Person'
> descriptor( message )
descriptor for type 'tutorial.Person'
```

3.1.22 Message\$fileDescriptor method

The fileDescriptor method retrieves the file descriptor of the descriptor associated with a message. See section 3.6 for more information about file descriptors.

```

> message <- new( tutorial.Person )
> message$fileDescriptor()
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
> fileDescriptor( message )
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)

```

3.2 message descriptors

Message descriptors are represented in R with the *Descriptor* S4 class. The class contains the slots pointer and type :

slot	description
pointer	external pointer to the Descriptor object of the C++ proto library. Documentation for the Descriptor class is available from the protocol buffer project page: http://code.google.com/apis/protocolbuffers/docs/reference/cpp/google.protobuf.descriptor.html#Descriptor
type	fully qualified path of the message type.

Table 5: Description of slots for the Descriptor S4 class

Similarly to messages, the \$ operator can be used to extract information from the descriptor, or invoke pseudo-methods. Table 6 describes the methods defined for the Descriptor class :

3.2.1 Extracting descriptors

The \$ operator, when used on a descriptor object retrieves descriptors that are contained in the descriptor.

This can be a field descriptor (see section 3.3), an enum descriptor (see section 3.4) or a descriptor for a nested type

```

> tutorial.Person$email
descriptor for field 'email' of type 'tutorial.Person'
> tutorial.Person$PhoneType
descriptor for enum 'PhoneType' with 3 values
> tutorial.Person$PhoneNumber
descriptor for type 'tutorial.Person.PhoneNumber'
> tutorial.Person.PhoneNumber
descriptor for type 'tutorial.Person.PhoneNumber'

```

Method	Section	Description
<code>new</code>	3.2.2	Creates a prototype of a message described by this descriptor.
<code>read</code>	3.2.3	Reads a message from a file or binary connection.
<code>readASCII</code>	3.2.4	Read a message in ASCII format from a file or text connection.
<code>name</code>	3.2.10	Retrieve the name of the message type associated with this descriptor.
<code>as.character</code>	3.2.6	character representation of a descriptor
<code>toString</code>	3.2.5	character representation of a descriptor (same as <code>as.character</code>)
<code>as.list</code>	3.2.7	return a named list of the field, enum, and nested descriptors included in this descriptor.
<code>asMessage</code>	3.2.8	return DescriptorProto message.
<code>fileDescriptor</code>	3.2.9	Retrieve the file descriptor of this descriptor.
<code>containing_type</code>	3.2.11	Retrieve the descriptor describing the message type containing this descriptor.
<code>field_count</code>	3.2.12	Return the number of fields in this descriptor.
<code>field</code>	3.2.13	Return the descriptor for the specified field in this descriptor.
<code>nested_type_count</code>	3.2.14	The number of nested types in this descriptor.
<code>nested_type</code>	3.2.15	Return the descriptor for the specified nested type in this descriptor.
<code>enum_type_count</code>	3.2.16	The number of enum types in this descriptor.
<code>enum_type</code>	3.2.17	Return the descriptor for the specified enum type in this descriptor.

Table 6: Description of methods for the Descriptor S4 class

3.2.2 The new method

The `new` method creates a prototype of a message described by the descriptor.

```
> tutorial.Person$new( )
message of type 'tutorial.Person' with 0 fields set
> new( tutorial.Person )
message of type 'tutorial.Person' with 0 fields set
```

Passing additional arguments to the method allows to directly set the fields of the message at construction time.

```
> tutorial.Person$new( email = "foo@bar.com" )
message of type 'tutorial.Person' with 1 field set
> update( tutorial.Person$new( ), email = "foo@bar.com" )
```

message of type 'tutorial.Person' with 1 field set

3.2.3 The read method

The read method is used to read a message from a file or a binary connection.

```
> message <- new( tutorial.Person.PhoneNumber,
+               type = "HOME", number = "+33(0)..." )
> tf <- tempfile()
> serialize( message, tf )
> m <- tutorial.Person.PhoneNumber$read( tf )
> writeLines( as.character( m ) )
number: "+33(0)..."
type: HOME
> m <- read( tutorial.Person.PhoneNumber, tf )
> writeLines( as.character( m ) )
number: "+33(0)..."
type: HOME
```

3.2.4 The readASCII method

The readASCII method is used to read a message from a text file or a character vector.

```
> text <- as.character(new(tutorial.Person, id=1, name="Murray"))
> text
[1] "name: \"Murray\"\\nid: 1\\n"
> msg <- tutorial.Person$readASCII(text)
```

3.2.5 The toString method

toString currently is an alias to the as.character function.

3.2.6 The as.character method

as.character prints the text representation of the descriptor as it would be specified in the .proto file.

```
> desc <- tutorial.Person
> writeLines( desc$toString() )
message Person {
  message PhoneNumber {
    required string number = 1;
    optional .tutorial.Person.PhoneType type = 2 [default = HOME];
```

```

    }
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    repeated .tutorial.Person.PhoneNumber phone = 4;
    extensions 100 to 199;
}
> writeLines( toString( desc ) )
message Person {
    message PhoneNumber {
        required string number = 1;
        optional .tutorial.Person.PhoneType type = 2 [default = HOME];
    }
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    repeated .tutorial.Person.PhoneNumber phone = 4;
    extensions 100 to 199;
}
> writeLines( as.character(tutorial.Person) )
message Person {
    message PhoneNumber {
        required string number = 1;
        optional .tutorial.Person.PhoneType type = 2 [default = HOME];
    }
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
}

```

```

    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    repeated .tutorial.Person.PhoneNumber phone = 4;
    extensions 100 to 199;
}

```

3.2.7 as.list

The `as.list` method returns a named list of the field, enum, and nested descriptors included in this descriptor.

```

> tutorial.Person$as.list()
$name
descriptor for field 'name' of type 'tutorial.Person'

$id
descriptor for field 'id' of type 'tutorial.Person'

$email
descriptor for field 'email' of type 'tutorial.Person'

$phone
descriptor for field 'phone' of type 'tutorial.Person'

$PhoneNumber
descriptor for type 'tutorial.Person.PhoneNumber'

$PhoneType
descriptor for enum 'PhoneType' with 3 values

```

3.2.8 asMessage

The `asMessage` method returns a message of type `google.protobuf.DescriptorProto` of the Descriptor.

```

> tutorial.Person$asMessage()
message of type 'google.protobuf.DescriptorProto' with 5 fields set

```

3.2.9 The fileDescriptor method

The fileDescriptor method retrieves the file descriptor of the descriptor. See section 3.6 for more information about file descriptors.

```
> desc <- tutorial.Person
> desc$fileDescriptor()
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
> fileDescriptor( desc )
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
```

3.2.10 The name method

The name method can be used to retrieve the name of the message type associated with the descriptor.

```
> tutorial.Person$name()
[1] "Person"
> tutorial.Person$name(full = TRUE)
[1] "tutorial.Person"
```

3.2.11 The containing_type method

The containing_type method retrieves the descriptor describing the message type containing this descriptor.

```
> tutorial.Person$containing_type()
NULL
> tutorial.Person$PhoneNumber$containing_type()
descriptor for type 'tutorial.Person'
```

3.2.12 The field_count method

The field_count method retrieves the number of fields in this descriptor.

```
> tutorial.Person$field_count()
[1] 4
```


3.2.13 The field method

The field method returns the descriptor for the specified field in this descriptor.

```
> tutorial.Person$field(1)
descriptor for field 'name' of type 'tutorial.Person'
```

3.2.14 The nested_type_count method

The nested_type_count method returns the number of nested types in this descriptor.

```
> tutorial.Person$nested_type_count()
[1] 1
```

3.2.15 The nested_type method

The nested_type method return the descriptor for the specified nested type in this descriptor.

```
> tutorial.Person$nested_type(1)
descriptor for type 'tutorial.Person.PhoneNumber'
```

3.2.16 The enum_type_count method

The enum_type_count method returns the number of enum types in this descriptor.

```
> tutorial.Person$enum_type_count()
[1] 1
```

3.2.17 The enum_type method

The enum_type method returns the descriptor for the specified enum type in this descriptor.

```
> tutorial.Person$enum_type(1)
descriptor for enum 'PhoneType' with 3 values
```

3.3 field descriptors

The class *FieldDescriptor* represents field descriptor in R. This is a wrapper S4 class around the `google::protobuf::FieldDescriptor` C++ class. Table 8 describes the methods defined for the *FieldDescriptor* class.

slot	description
pointer	External pointer to the FieldDescriptor C++ variable
name	simple name of the field
full_name	fully qualified name of the field
type	name of the message type where the field is declared

Table 7: Description of slots for the FieldDescriptor S4 class

method	section	description
as.character	3.3.1	character representation of a descriptor
toString	3.3.2	character representation of a descriptor (same as as.character)
asMessage	3.3.3	return FieldDescriptorProto message.
name	3.3.4	Return the name of the field descriptor.
fileDescriptor	3.3.5	Return the fileDescriptor where this field is defined.
containing_type	3.3.6	Return the containing descriptor of this field.
is_extension	3.3.7	Return TRUE if this field is an extension.
number	3.3.8	Gets the declared tag number of the field.
type	3.3.9	Gets the type of the field.
cpp_type	3.3.10	Gets the C++ type of the field.
label	3.3.11	Gets the label of a field (optional, required, or repeated).
is_repeated	3.3.12	Return TRUE if this field is repeated.
is_required	3.3.13	Return TRUE if this field is required.
is_optional	3.3.14	Return TRUE if this field is optional.
has_default_value	3.3.15	Return TRUE if this field has a default value.
default_value	3.3.16	Return the default value.
message_type	3.3.17	Return the message type if this is a message type field.
enum_type	3.3.18	Return the enum type if this is an enum type field.

Table 8: Description of methods for the FieldDescriptor S4 class

3.3.1 as.character

The `as.character` method brings the debug string of the field descriptor.

```
> writeLines( as.character( tutorial.Person$PhoneNumber ) )
message PhoneNumber {
  required string number = 1;
  optional .tutorial.Person.PhoneType type = 2 [default = HOME];
}
```

3.3.2 toString

toString is an alias of as.character.

```
> writeLines( tutorial.Person.PhoneNumber$toString() )
message PhoneNumber {
  required string number = 1;
  optional .tutorial.Person.PhoneType type = 2 [default = HOME];
}
```

3.3.3 asMessage

The asMessage method returns a message of type google.protobuf.FieldDescriptorProto of the FieldDescriptor.

```
> tutorial.Person$id$asMessage()
message of type 'google.protobuf.FieldDescriptorProto' with 4 fields set
> writeLines(as.character(tutorial.Person$id$asMessage()))
name: "id"
number: 2
label: LABEL_REQUIRED
type: TYPE_INT32
```

3.3.4 name

The name method can be used to retrieve the name of the field descriptor.

```
> name( tutorial.Person$id )
[1] "id"
> name( tutorial.Person$id, full=TRUE )
[1] "tutorial.Person.id"
```

3.3.5 fileDescriptor

The fileDescriptor method can be used to retrieve the file descriptor of the field descriptor.

```
> fileDescriptor(tutorial.Person$id)
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
> tutorial.Person$id$fileDescriptor()
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
```

3.3.6 containing_type

The `containing_type` method can be used to retrieve the descriptor for the message type that contains this descriptor.

```
> containing_type(tutorial.Person$id)
descriptor for type 'tutorial.Person'
> tutorial.Person$id$containing_type()
descriptor for type 'tutorial.Person'
```

3.3.7 is_extension

The `is_extension` method returns TRUE if this field is an extension.

```
> is_extension( tutorial.Person$id )
[1] FALSE
> tutorial.Person$id$is_extension()
[1] FALSE
```

3.3.8 number

The `number` method returns the declared tag number of this field.

```
> number( tutorial.Person$id )
[1] 2
> tutorial.Person$id$number()
[1] 2
```

3.3.9 type

The `type` method can be used to retrieve the type of the field descriptor.

```
> type( tutorial.Person$id )
[1] 5
> tutorial.Person$id$type()
[1] 5
```

3.3.10 cpp_type

The `cpp_type` method can be used to retrieve the C++ type of the field descriptor.

```
> cpp_type( tutorial.Person$id )
[1] 1
> tutorial.Person$id$cpp_type()
[1] 1
```

3.3.11 label

Gets the label of a field (optional, required, or repeated). The `label` method returns the label of a field (optional, required, or repeated). By default it returns a number value, but the optional `as.string` argument can be provided to return a human readable string representation.

```
> label( tutorial.Person$id )
[1] 2
> label( tutorial.Person$id , TRUE)
[1] "LABEL_REQUIRED"
> tutorial.Person$id$label(TRUE)
[1] "LABEL_REQUIRED"
```

3.3.12 is_repeated

The `is_repeated` method returns TRUE if this field is repeated.

```
> is_repeated( tutorial.Person$id )
[1] FALSE
> tutorial.Person$id$is_repeated()
[1] FALSE
```

3.3.13 is_required

The `is_required` method returns TRUE if this field is required.

```
> is_required( tutorial.Person$id )
[1] TRUE
> tutorial.Person$id$is_required()
[1] TRUE
```

3.3.14 is_optional

The `is_optional` method returns TRUE if this field is optional.

```
> is_optional( tutorial.Person$id )
[1] FALSE
> tutorial.Person$id$is_optional()
[1] FALSE
```

3.3.15 has_default_value

The `has_default_value` method returns TRUE if this field has a default value.

```
> has_default_value(tutorial.Person$PhoneNumber$type)
[1] TRUE
> has_default_value(tutorial.Person$PhoneNumber$number)
[1] FALSE
```

3.3.16 default_value

The `default_value` method returns the default value of a field.

```
> default_value( tutorial.Person$PhoneNumber$type )
[1] 1
> default_value( tutorial.Person$PhoneNumber$number )
[1] ""
```

3.3.17 message_type

The `message_type` method returns the message type if this is a message type field.

```
> message_type(tutorial.Person$phone)
descriptor for type 'tutorial.Person.PhoneNumber'
> tutorial.Person$phone$message_type()
descriptor for type 'tutorial.Person.PhoneNumber'
```

3.3.18 enum_type

The `enum_type` method returns the enum type if this is an enum type field.

```
> enum_type(tutorial.Person$PhoneNumber$type)
descriptor for enum 'PhoneType' with 3 values
```

3.4 enum descriptors

The class *EnumDescriptor* is an R wrapper class around the C++ class `google::protobuf::EnumDescriptor`. Table 10 describes the methods defined for the *EnumDescriptor* class.

slot	description
pointer	External pointer to the EnumDescriptor C++ variable
name	simple name of the enum
full_name	fully qualified name of the enum
type	name of the message type where the enum is declared

Table 9: Description of slots for the EnumDescriptor S4 class

method	section	description
as.list	3.4.2	return a named integer vector with the values of the enum and their names.
as.character	3.4.3	character representation of a descriptor
toString	3.4.4	character representation of a descriptor (same as as.character)
asMessage	3.4.5	return EnumDescriptorProto message.
name	3.4.6	Return the name of the enum descriptor.
fileDescriptor	3.4.7	Return the fileDescriptor where this field is defined.
containing_type	3.4.8	Return the containing descriptor of this field.
length	3.4.9	Return the number of constants in this enum.
has	3.4.10	Return TRUE if this enum contains the specified named constant string.
value_count	3.4.11	Return the number of constants in this enum (same as length).
value	3.4.12	Return the EnumValueDescriptor of an enum value of specified index, name, or number.

Table 10: Description of methods for the EnumDescriptor S4 class

3.4.1 Extracting descriptors

The `$` operator, when used on a EnumDescriptor object retrieves EnumValueDescriptors that are contained in the descriptor.

```
> tutorial.Person$PhoneType$WORK
[1] 2
> name(tutorial.Person$PhoneType$value(number=2))
[1] "WORK"
```

3.4.2 as.list

The `as.list` method creates a named R integer vector that captures the values of the enum and their names.

```

> as.list( tutorial.Person$PhoneType )
$MOBILE
[1] 0

$HOME
[1] 1

$WORK
[1] 2

```

3.4.3 as.character

The as.character method brings the debug string of the enum type.

```

> writeLines( as.character( tutorial.Person$PhoneType ) )
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

```

3.4.4 toString

The toString method brings the debug string of the enum type.

```

> writeLines( toString( tutorial.Person$PhoneType ) )
enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

```

3.4.5 asMessage

The asMessage method returns a message of type google.protobuf.EnumDescriptorProto of the EnumDescriptor.

```

> tutorial.Person$PhoneType$asMessage()
message of type 'google.protobuf.EnumDescriptorProto' with 2 fields set
> writeLines(as.character(tutorial.Person$PhoneType$asMessage()))
name: "PhoneType"

```



```

value {
  name: "MOBILE"
  number: 0
}
value {
  name: "HOME"
  number: 1
}
value {
  name: "WORK"
  number: 2
}

```

3.4.6 name

The name method can be used to retrieve the name of the enum descriptor.

```

> name( tutorial.Person$PhoneType )
[1] "PhoneType"
> name( tutorial.Person$PhoneType, full=TRUE )
[1] "tutorial.Person.PhoneType"

```

3.4.7 fileDescriptor

The fileDescriptor method can be used to retrieve the file descriptor of the enum descriptor.

```

> fileDescriptor(tutorial.Person$PhoneType)
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)
> tutorial.Person$PhoneType$fileDescriptor()
file descriptor for package tutorial
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)

```

3.4.8 containing_type

The containing_type method can be used to retrieve the descriptor for the message type that contains this enum descriptor.

```

> tutorial.Person$PhoneType$containing_type()
descriptor for type 'tutorial.Person'

```

3.4.9 length

The length method returns the number of constants in this enum.

```
> length(tutorial.Person$PhoneType)
[1] 3
> tutorial.Person$PhoneType$length()
[1] 3
```

3.4.10 has

The has method returns TRUE if this enum contains the specified named constant string.

```
> tutorial.Person$PhoneType$has("WORK")
[1] TRUE
> tutorial.Person$PhoneType$has("nonexistant")
[1] FALSE
```

3.4.11 value_count

The value_count method returnst he number of constants in this enum.

```
> value_count(tutorial.Person$PhoneType)
[1] 3
> tutorial.Person$PhoneType$value_count()
[1] 3
```

3.4.12 value

The value method extracts an EnumValueDescriptor. Exactly one argument of 'index', 'number', or 'name' must be specified to identify which constant is desired.

```
> tutorial.Person$PhoneType$value(1)
enum value descriptor tutorial.Person.MOBILE
> tutorial.Person$PhoneType$value(name="HOME")
enum value descriptor tutorial.Person.HOME
> tutorial.Person$PhoneType$value(number=1)
enum value descriptor tutorial.Person.HOME
```

3.5 enum value descriptors

The class *EnumValueDescriptor* is an R wrapper class around the C++ class `google::protobuf::EnumValueDescriptor`. Table 12 describes the methods defined for the EnumValueDescriptor class.

slot	description
pointer	External pointer to the EnumValueDescriptor C++ variable
name	simple name of the enum value
full_name	fully qualified name of the enum value

Table 11: Description of slots for the EnumValueDescriptor S4 class

method	section	description
number	3.5.1	return the number of this EnumValueDescriptor.
name	3.5.2	Return the name of the enum value descriptor.
enum_type	3.5.3	return the EnumDescriptor type of this EnumValueDescriptor.
as.character	3.5.4	character representation of a descriptor.
toString	3.5.5	character representation of a descriptor (same as as.character).
asMessage	3.5.6	return EnumValueDescriptorProto message.

Table 12: Description of methods for the EnumValueDescriptor S4 class

3.5.1 number

The number method can be used to retrieve the number of the enum value descriptor.

```
> number( tutorial.Person$PhoneType$value(number=2) )
[1] 2
```

3.5.2 name

The name method can be used to retrieve the name of the enum value descriptor.

```
> name( tutorial.Person$PhoneType$value(number=2) )
[1] "WORK"
> name( tutorial.Person$PhoneType$value(number=2), full=TRUE )
[1] "tutorial.Person.WORK"
```

3.5.3 enum_type

The enum_type method can be used to retrieve the EnumDescriptor of the enum value descriptor.

```
> enum_type( tutorial.Person$PhoneType$value(number=2) )
descriptor for enum 'PhoneType' with 3 values
```

3.5.4 as.character

The `as.character` method brings the debug string of the enum value type.

```
> writeLines( as.character( tutorial.Person$PhoneType$value(number=2) ) )  
WORK = 2;
```

3.5.5 toString

The `toString` method brings the debug string of the enum value type.

```
> writeLines( toString( tutorial.Person$PhoneType$value(number=2) ) )  
WORK = 2;
```

3.5.6 asMessage

The `asMessage` method returns a message of type `google.protobuf.EnumValueDescriptorProto` of the `EnumValueDescriptor`.

```
> tutorial.Person$PhoneType$value(number=2)$asMessage()  
message of type 'google.protobuf.EnumValueDescriptorProto' with 2 fields  
set  
> write-  
Lines(as.character(tutorial.Person$PhoneType$value(number=2)$asMessage()))  
name: "WORK"  
number: 2
```

3.6 file descriptors

File descriptors describe a whole .proto file and are represented in R with the *FileDescriptor* S4 class. The class contains the slots `pointer`, `filename`, and `package` :

Similarly to messages, the `$` operator can be used to extract fields from the file descriptor (in this case, types defined in the file), or invoke pseudo-methods. Table 14 describes the methods defined for the `FileDescriptor` class.

```
> f <- tutorial.Person$fileDescriptor()  
> f  
file descriptor for package tutorial  
(/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto)  
> f$Person  
descriptor for type 'tutorial.Person'
```

slot	description
pointer	external pointer to the FileDescriptor object of the C++ proto library. Documentation for the FileDescriptor class is available from the protocol buffer project page: http://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.descriptor.html#FileDescriptor
filename	fully qualified pathname of the .proto file.
package	package name defined in this .proto file.

Table 13: Description of slots for the FileDescriptor S4 class

method	section	description
name	3.6.5	Return the filename for this FileDescriptorProto.
package	3.6.6	Return the file-level package name specified in this FileDescriptorProto.
as.character	3.6.1	character representation of a descriptor.
toString	3.6.2	character representation of a descriptor (same as as.character).
asMessage	3.6.3	return FileDescriptorProto message.
as.list	3.6.4	return named list of descriptors defined in this file descriptor.

Table 14: Description of methods for the FileDescriptor S4 class

3.6.1 as.character

The as.character method brings the debug string of the file descriptor.

```
> writeLines( as.character(fileDescriptor(tutorial.Person)) )
syntax = "proto2";

package tutorial;

option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

message Person {
  message PhoneNumber {
    required string number = 1;
    optional .tutorial.Person.PhoneType type = 2 [default = HOME];
```

```

    }
    enum PhoneType {
        MOBILE = 0;
        HOME = 1;
        WORK = 2;
    }
    required string name = 1;
    required int32 id = 2;
    optional string email = 3;
    repeated .tutorial.Person.PhoneNumber phone = 4;
    extensions 100 to 199;
}

message AddressBook {
    repeated .tutorial.Person person = 1;
}

service EchoService {
    rpc Echo(.tutorial.Person) returns (.tutorial.Person);
}

```

3.6.2 toString

toString is an alias of as.character.

```
> writeLines( fileDescriptor(tutorial.Person)$toString() )
syntax = "proto2";
```

```
package tutorial;
```

```
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
```

```

message Person {
    message PhoneNumber {
        required string number = 1;
        optional .tutorial.Person.PhoneType type = 2 [default = HOME];
    }
    enum PhoneType {
        MOBILE = 0;

```

```

    HOME = 1;
    WORK = 2;
}
required string name = 1;
required int32 id = 2;
optional string email = 3;
repeated .tutorial.Person.PhoneNumber phone = 4;
extensions 100 to 199;
}

message AddressBook {
    repeated .tutorial.Person person = 1;
}

service EchoService {
    rpc Echo(.tutorial.Person) returns (.tutorial.Person);
}

```

3.6.3 asMessage

The asMessage method returns a protocol buffer message representation of the file descriptor.

```

> asMessage(tutorial.Person$fileDescriptor())
message of type 'google.protobuf.FileDescriptorProto' with 5 fields set
> writeLines( as.character(asMessage(tutorial.Person$fileDescriptor())) )
name:
"/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto"
package: "tutorial"
message_type {
  name: "Person"
  field {
    name: "name"
    number: 1
    label: LABEL_REQUIRED
    type: TYPE_STRING
  }
  field {
    name: "id"
    number: 2
    label: LABEL_REQUIRED

```

```

    type: TYPE_INT32
  }
  field {
    name: "email"
    number: 3
    label: LABEL_OPTIONAL
    type: TYPE_STRING
  }
  field {
    name: "phone"
    number: 4
    label: LABEL_REPEATED
    type: TYPE_MESSAGE
    type_name: ".tutorial.Person.PhoneNumber"
  }
  nested_type {
    name: "PhoneNumber"
    field {
      name: "number"
      number: 1
      label: LABEL_REQUIRED
      type: TYPE_STRING
    }
    field {
      name: "type"
      number: 2
      label: LABEL_OPTIONAL
      type: TYPE_ENUM
      type_name: ".tutorial.Person.PhoneType"
      default_value: "HOME"
    }
  }
}
enum_type {
  name: "PhoneType"
  value {
    name: "MOBILE"
    number: 0
  }
  value {
    name: "HOME"

```



```

        number: 1
    }
    value {
        name: "WORK"
        number: 2
    }
}
extension_range {
    start: 100
    end: 200
}
}
message_type {
    name: "AddressBook"
    field {
        name: "person"
        number: 1
        label: LABEL_REPEATED
        type: TYPE_MESSAGE
        type_name: ".tutorial.Person"
    }
}
service {
    name: "EchoService"
    method {
        name: "Echo"
        input_type: ".tutorial.Person"
        output_type: ".tutorial.Person"
    }
}
options {
    java_package: "com.example.tutorial"
    java_outer_classname: "AddressBookProtos"
}

```

3.6.4 as.list

The `as.list` method creates a named R list that contains the descriptors defined in this file descriptor.

```
> as.list( tutorial.Person$fileDescriptor() )
```

```
$Person  
descriptor for type 'tutorial.Person'
```

```
$AddressBook  
descriptor for type 'tutorial.AddressBook'
```

```
$EchoService
```

3.6.5 name

The name method can be used to retrieve the file name associated with the file descriptor. The optional boolean argument can be specified if full pathnames are desired.

```
> name( tutorial.Person$fileDescriptor() )  
[1] "addressbook.proto"  
> tutorial.Person$fileDescriptor()$name(TRUE)  
[1] "/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto"
```

3.6.6 package

The package method can be used to retrieve the package scope associated with this file descriptor.

```
> tutorial.Person$fileDescriptor()$package()  
[1] "tutorial"
```

3.7 service descriptors

Not fully implemented. Needs to be connected to a concrete RPC implementation. The Google Protocol Buffers C++ open-source library does not include an RPC implementation, but this can be connected easily to others.

3.8 method descriptors

Not fully implemented. Needs to be connected to a concrete RPC implementation. The Google Protocol Buffers C++ open-source library does not include an RPC implementation, but this can be connected easily to others.

4 Utilities

4.1 coercing objects to messages

The asMessage function uses the standard coercion mechanism of the as method, and so can be used as a shorthand :

```

> asMessage( tutorial.Person.PhoneType )
message of type 'google.protobuf.EnumDescriptorProto' with 2 fields set
> asMessage( tutorial.Person$email )
message of type 'google.protobuf.FieldDescriptorProto' with 4 fields set
> asMessage( fileDescriptor( tutorial.Person ) )
message of type 'google.protobuf.FileDescriptorProto' with 5 fields set

```

4.2 completion

The RProtoBuf package implements the `.DollarNames` S3 generic function (defined in the `utils` package) for all classes.

Completion possibilities include pseudo method names for all classes, plus :

- field names for messages
- field names, enum types, nested types for message type descriptors
- names for enum descriptors
- names for top-level extensions
- message names for file descriptors

In the unlikely event that there is a user-defined field of exactly the same name as one of the pseudo methods, the user-defined field shall take precedence for completion purposes by design, since the method name can always be invoked directly.

4.3 with and within

The S3 generic `with` function is implemented for class `Message`, allowing to evaluate an R expression in an environment that allows to retrieve and set fields of a message simply using their names.

```

> message <- new( tutorial.Person, email = "foo@bar.com" )
> with( message, {
+   # set the id field
+   id <- 2
+
+   # set the name field from the email field
+   name <- gsub( "[@]", " ", email )
+
+   sprintf( "%d [%s] : %s", id, email, name )
+ } )
[1] "2 [foo@bar.com] : foo bar.com"

```

The difference between `with` and `within` is the value that is returned. For `with` returns the result of the R expression, for `within` the message is returned. In both cases, the message is modified because RProtoBuf works by reference.

4.4 identical

The `identical` method is implemented to compare two messages.

```
> m1 <- new( tutorial.Person, email = "foo@bar.com", id = 2 )
> m2 <- update( new( tutorial.Person) , email = "foo@bar.com", id = 2 )
> identical( m1, m2 )
[1] TRUE
```

The `==` operator can be used as an alias to `identical`.

```
> m1 == m2
[1] TRUE
> m1 != m2
[1] FALSE
```

Alternatively, the `all.equal` function can be used, allowing a tolerance when comparing float or double values.

4.5 merge

`merge` can be used to merge two messages of the same type.

```
> m1 <- new( tutorial.Person, name = "foobar" )
> m2 <- new( tutorial.Person, email = "foo@bar.com" )
> m3 <- merge( m1, m2 )
> writeLines( as.character( m3 ) )
name: "foobar"
email: "foo@bar.com"
```

4.6 P

The `P` function is an alternative way to retrieve a message descriptor using its type name. It is not often used because of the lookup mechanism described in section 5.2.

```
> P("tutorial.Person")
descriptor for type 'tutorial.Person'
> new( P("tutorial.Person") )
message of type 'tutorial.Person' with 0 fields set
> tutorial.Person
descriptor for type 'tutorial.Person'
> new( tutorial.Person )
message of type 'tutorial.Person' with 0 fields set
```

5 Advanced Features

5.1 Extensions

Extensions allow you to declare a range of field numbers in a message that are available for extension types. This allows others to declare new fields for a given message type possibly in their own .proto files without having to edit the original file. See <https://developers.google.com/protocol-buffers/docs/proto#extensions>.

Notice that the last line of the Person message schema in addressbook.proto is the following line :

```
extensions 100 to 199;
```

This specifies that other users in other .proto files can use tag numbers between 100 and 199 for extension types of this message.

For example, the following file extend.proto:

```
import "/tmp/RtmpI2cyiR/Rinst192c11ccdd50/RProtoBuf/proto/addressbook.proto";
package tutorial;
extend Person {
  optional string nationality = 100;
}
```

After importing this new .proto file with an extension defined, we can get or set the value of this extension in Messages of type Person.

```
> library(RProtoBuf)
> readProtoFiles(extend.proto)
> person <- new(tutorial.Person, id=1, name="Murray")
> person
message of type 'tutorial.Person' with 2 fields set
> person$setExtension(P("tutorial.nationality"), "USA")
> cat(as.character(person))
name: "Murray"
id: 1
[tutorial.nationality]: "USA"
> person$getExtension(P("tutorial.nationality"))
[1] "USA"
```

The character output of this message places the extension type in brackets to differentiate it from other fields in the message, and so we know we need to use the `getExtension` method rather than a '\$' field extractor to get the value.

5.2 Descriptor lookup

The RProtoBuf package uses the user defined tables framework that is defined as part of the RObjectTables package available from the OmegaHat project.

The feature allows RProtoBuf to install the special environment *RProtoBuf:DescriptorPool* in the R search path. The environment is special in that, instead of being associated with a static hash table, it is dynamically queried by R as part of R's usual variable lookup. In other words, it means that when the R interpreter looks for a binding to a symbol (foo) in its search path, it asks to our package if it knows the binding "foo", this is then implemented by the RProtoBuf package by calling an internal method of the protobuf C++ library.

5.3 64-bit integer issues

R does not have native 64-bit integer support. Instead, R treats large integers as doubles which have limited precision. For example, it loses the ability to distinguish some distinct integers:

```
> 2^53 == (2^53 + 1)
[1] TRUE
```

Protocol Buffers are frequently used to pass data between different systems, however, and most other systems these days have support for 64-bit integers. To work around this, RProtoBuf allows users to get and set 64-bit integer types by treating them as characters when running on a platform with a 64-bit long long type available.

If we try to set an int64 field in R to double values, we lose precision:

```
> test <- new(protobuf_unittest.TestAllTypes)
> test$repeated_int64 <- c(2^53, 2^53+1)
> length(unique(test$repeated_int64))

[1] 1
```

However, we can specify the values as character strings so that the C++ library on which RProtoBuf is based can store a true 64-bit integer representation of the data.

```
> test$repeated_int64 <- c("9007199254740992", "9007199254740993")
```

When reading the value back into R, numeric types are returned by default, but when the full precision is required a character value will be returned if the `RProtoBuf.int64AsString` option is set to TRUE.

```
> options("RProtoBuf.int64AsString" = FALSE)

$RProtoBuf.int64AsString
[1] FALSE

> test$repeated_int64

[1] 9.007199e+15 9.007199e+15

> length(unique(test$repeated_int64))

[1] 1

> options("RProtoBuf.int64AsString" = TRUE)
```

```

$RProtoBuf.int64AsString
[1] FALSE

> test$repeated_int64
[1] "9007199254740992" "9007199254740993"

> length(unique(test$repeated_int64))
[1] 2

```

5.4 Deprecated Feature: Protocol Buffer Groups

Groups are a deprecated feature that offered another way to nest information in message definitions. For example, the `TestAllTypes` message type in `unittest.proto` includes an `OptionalGroup` type:

```

optional group OptionalGroup = 16 {
  optional int32 a = 17;
}

```

And although the feature is deprecated, it can be used with `RProtoBuf`:

```

> test <- new(protobuf_unittest.TestAllTypes)
> test$optionalgroup$a <- 3
> test$optionalgroup$a
[1] 3
> cat(as.character(test))
OptionalGroup {
  a: 3
}

```

Note that groups simply combine a nested message type and a field into a single declaration. The field type is `OptionalGroup` in this example, and the field name is converted to lower-case 'optionalgroup' so as not to conflict with the type name.

6 Other approaches

Saptarshi Guha wrote another package that deals with integration of protocol buffer messages with R, taking a different angle : serializing any R object as a message, based on a single catch-all proto file. Saptarshi's package is available at <http://ml.stat.purdue.edu/rhipe/doc/html/ProtoBuffers.html>.

Jeroen Ooms took a similar approach influenced by Saptarshi in his `RProtoBufUtils` package. Unlike Saptarshi's package, `RProtoBufUtils` depends on `RProtoBuf` for underlying message operations. This package is available at <https://github.com/jeroenooms/RProtoBufUtils>.

7 Plans for future releases

Protocol buffers have a mechanism for remote procedure calls (rpc) that is not yet used by RProtoBuf, but we may one day take advantage of this by writing a protocol buffer message R server, and client code as well, probably based on the functionality of the Rserve package.

Extensions have been implemented in RProtoBuf and have been extensively used and tested, but they are not currently described in this vignette. Additional examples and documentation are needed for extensions.

8 Acknowledgments

Some of the design of the package is based on the design of the rJava package by Simon Urbanek (dispatch on new, S4 class structures using external pointers, etc ...). We'd like to thank Simon for his indirect involvement on RProtoBuf.

The user defined table mechanism, implemented by Duncan Temple Lang for the purpose of the RObjectTables package allowed the dynamic symbol lookup (see [section 5.2](#)). Many thanks to Duncan for this amazing feature.