RProtoBuf: An R API for Protocol Buffers

Romain François Dirk Eddelbuettel

Version 0.0-6 as of January 11, 2010

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 0.0-6 of the `RProtoBuf` package which brings support for protocol buffer messages to R.

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 Static use: Revisiting the tutorial

In this section, we illustrate use of Protocol Buffers in a *static* fashion: based on the `proto` file, code is generated by the compiler and used by language-specific bindings.

2.1 The address book example

Through this document, we will use the `addressbook` example that is used by the official tutorials for Java, Python and C++. It is based on the following `proto` file:

```
package tutorial;
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";
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;
}
message AddressBook {
  repeated Person person = 1;
}
service EchoService {
  rpc Echo (Person) returns (Person);
}
```

The `proto` file defines :

- three message types
  - `tutorial.Person`,
  - `tutorial.Person.PhoneNumber` and
  - `tutorial.AddressBook`
- an enum type `tutorial.Person.PhoneType` with three values `MOBILE`, `HOME` and `WORK`

We see that a message type can contain several different items:

- sets of fields—for example the `Person` message type contains the required field `name` of primitive type `string` associated with the tag number 1;
- fields can be either required (as `name` or `id`) or optional (as `email`);
- other message type descriptions—`Person` contains the nested message type `PhoneNumber`; hence the fully qualified type of `PhoneNumber` is `tutorial.Person.PhoneNumber`
- enum type descriptions.

Using the `protoc` compiler, we can generate functions that access these Protocol Buffer messages and their components for both reading and writing using any of the three officially supported languages C++, Java and Python. For example, for C++ the call

```
protoc -cpp_out=.  addressbook.proto
```

generates almost eighteen hundreed lines of code: seven hundred in a header file `addressbook.pb.h` and almost elevenhundred in a file `addressbook.pb.cc`. These two files are used in the tutorial application programs `add_person.cc` and `list_people.cc`. The former adds a new record to an address book defined by the `proto` file shown above, and the latter prints the contents of all records in the address book.

2.2 Simple R accessors for the address book example

The Protocol Buffers tutorial contains two simple standalone programs to, respectively, add a record and list all records from an address book as defined by the `proto` file shown above.

In order to ease the transition from C++ to R when working with Protocol Buffers, we implemented two simple wrapper functions in C++ that accomplish essentially the same task, but are callable directly from R. This use the `Rcpp` package for interfacing C++ from R.

Adding a record: `addPerson()`

The R function `addPerson()` accepts five arguments:

```
> args(addPerson)

function (filename, id, name, emails, phones)
NULL
```

The first argument denotes the (binary) file into which the new address book record will be written. The next four argument describe the record to be added. Both `id` and `name` have to be of length one, whereas `emails` and `phones` can be of length zero as they correspond to optional fields.

The actual implementation in C++ is close to the tutorial example and can be used as gentle first step in programming with R and Protocol Buffers.

Listing all records: `listPeopleAsList()` and `listPeopleAsDataFrame()`

Displaying the content of an address book defined by the `proto` file above is straigtforward in the command-line example as records are simply printed to the screen.

For our use, these data need to be read from the file and transfered back to R. Given the definition of the `proto` file, we face an interesting problem: some fields are optional, and some fields can be repeated numerous types. That means our data structure can be textslragged: the number of entries per record cannot be expected to be constant.

Of course, R can handle such dynamic data structures rather easily. One approach is to use lists of lists which is implemented in `listPeopleasList()` which returns a `list` object to R with one entry per address book record. Each of these entries is itself a list comprised of two character vectors of length one (name and id) as well as further lists for emails and phone numbers.

Similarly, we can use the fact that the id field is key identifying a person and return two `data.frames` to R that that can then be merged on the id. This allows for both potentially missing entries (as for the optional email fields) as well as repeated fields (as for the phone number records). The R function `listPeopleAsDataFrame()` implements this approach, and its corresponding C++ function is very close to the tutorial file `list_people.cc`.

Both these functions show how R can use the C++ code generated by the Protocol Buffers compiler. Binding the generated functions to R is straightforward — but arguably tedious as new interface code needs to be written manually. But R as is dynamically-typed language, we would like to use Protocol Buffers in a less rigid fashion. The next few sections show how this can be done.

3 Dynamic 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.

3.1 Importing proto files

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")
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/Rinst1364685161/RProtoBuf/proto/addressbook.proto"
[2] "/tmp/Rinst1364685161/RProtoBuf/proto/helloworld.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 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 6.

```
> ls("RProtoBuf:DescriptorPool")

[1] "rprotobuf.HelloWorldRequest"  "rprotobuf.HelloWorldResponse"
[3] "tutorial.AddressBook"         "tutorial.Person"
```

3.2 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)
```

3.3 Access and modify fields of a message

Once the message created, its fields can be quiered 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 mesages, supplying either their name or their tag number :

```
> p[["name"]] <- "Romain Francois"
> p[[2]] <- 3
> p[["email"]]
```

```
[1] "francoisromain@free.fr"
```

3.4 Display messages

For debugging purposes, protocol buffer messages implement the `as.character` method:

```
> writeLines(as.character(p))
```

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

3.5 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/Rtmp3ztWaE/file109cf92e"
```

```
> 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 traditionnal object oriented fashion using the dollar operator :

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

3.6 Parsing messages

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

```
> read
```

```
nonstandardGenericFunction for "read" defined from package "RProtoBuf"

function (descriptor, input)
{
    standardGeneric("read")
}
<environment: 0xa288af0>
Methods may be defined for arguments: descriptor, input
Use  showMethods("read")  for currently available ones.
```

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)
```

4 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 traditionnal object oriented notation `object$method(arguments)`.

4.1 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. The 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 `tutorial.Person`

Table 1: Description of slots for the `Message` S4 class

4.1.1 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]]
[1] " message of type 'tutorial.Person.PhoneNumber' "

[[2]]
[1] " message of type 'tutorial.Person.PhoneNumber' "

> # using the tag number
> message[[ 2 ]] # id

[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.

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

method	section	description
`has`	4.1.3	Indicates if a message has a given field.
`clone`	4.1.4	Creates a clone of the message
`isInitialized`	4.1.5	Indicates if a message has all its required fields set
`serialize`	4.1.6	serialize a message to a file or a binary connection or retrieve the message payload as a raw vector
`clear`	4.1.7	Clear one or several fields of a message, or the entire message
`size`	4.1.8	The number of elements in a message field
`bytesize`	4.1.9	The number of bytes the message would take once serialized
`swap`	4.1.10	swap elements of a repeated field of a message
`set`	4.1.11	set elements of a repeated field
`fetch`	4.1.12	fetch elements of a repeated field
`add`		add elements to a repeated field
`str`	4.1.14	the R structure of the message
`as.character`	4.1.15	character representation of a message
`toString`	4.1.16	character representation of a message (same as `as.character`)
`update`	4.1.18	updates several fields of a message at once
`descriptor`	4.1.19	get the descriptor of the message type of this message
`fileDescriptor`	4.1.20	get the file descriptor of this message's descriptor

Table 2: Description of slots for the `Message` S4 class

4.1.2 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"
```

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

field type	R type (non repeated)	R type (repeated)
double	`double` vector	`double` vector
float	`double` vector	`double` vector
int32	`integer` vector	`integer` vector
int64	`integer` vector	`integer` vector
uint32	`integer` vector	`integer` vector
uint64	`integer` vector	`integer` vector
sint32	`integer` vector	`integer` vector
sint64	`integer` vector	`integer` vector
fixed32	`integer` vector	`integer` vector
fixed64	`integer` vector	`integer` vector
sfixed32	`integer` vector	`integer` vector
sfixed64	`integer` vector	`integer` 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.

4.1.3 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
```

4.1.4 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.

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`
`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.

```
> 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"
```

4.1.5 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
attr(,"uninitialized")
[1] "id"

> message$id <- 2
> message$isInitialized()

[1] TRUE
```

4.1.6 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/Rtmp3ztWaE/file580bd78f"
```

```
> message$serialize(tf1)
> tf2 <- tempfile()
> tf2
```

```
[1] "/tmp/Rtmp3ztWaE/file6a2342ec"
```

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

The files file580bd78f and file6a2342ec 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
```

4.1.7 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.

4.1.8 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.

4.1.9 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
```

4.1.10 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
```

4.1.11 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
}
```

4.1.12 Message$fetch method

The `fetch` 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"  )
+                 ) )
> message$fetch( "phone", 1 )
```

```
[[1]]
[1] " message of type 'tutorial.Person.PhoneNumber' "
```

4.1.13 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
}
```

4.1.14 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"
```

4.1.15 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"
```

4.1.16 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"
```

4.1.17 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 4.1.1.

4.1.18 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" )

[1] " message of type 'tutorial.Person' "

> writeLines( message$as.character() )

name: "foo"
id: 2
email: "foo@bar.com"
```

4.1.19 Message$descriptor method

The `descriptor` method retrieves the descriptor of a message. See section 4.2 for more information about message type descriptors.

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

[1] "descriptor for type 'tutorial.Person' "

> descriptor(message)

[1] "descriptor for type 'tutorial.Person' "
```

4.1.20 Message$fileDescriptor method

The `fileDescriptor` method retrieves the file descriptor of the descriptor associated with a message. See section 4.5 for more information about file descriptors.

```
> message <- new(tutorial.Person)
> message$fileDescriptor()

[1] "file descriptor"

> fileDescriptor(message)

[1] "file descriptor"
```

4.2 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 pseuso-methods.

4.2.1 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 4.3), an enum descriptor (see section 4.4) or a descriptor for a nested type

```
> tutorial.Person$email

[1] "descriptor for field 'email' of type 'tutorial.Person' "

> tutorial.Person$PhoneType

[1] "descriptor for enum 'PhoneType' of type 'tutorial.Person' "

> tutorial.Person$PhoneNumber
```

```
[1] "descriptor for type 'tutorial.Person.PhoneNumber' "
```

```
> tutorial.Person.PhoneNumber
```

```
[1] "descriptor for type 'tutorial.Person.PhoneNumber' "
```

4.2.2 The new method

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

```
> tutorial.Person$new()
```

```
[1] " message of type 'tutorial.Person' "
```

```
> new(tutorial.Person)
```

```
[1] " message of type 'tutorial.Person' "
```

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

```
> tutorial.Person$new(email = "foo@bar.com")
```

```
[1] " message of type 'tutorial.Person' "
```

```
> update(tutorial.Person$new(), email = "foo@bar.com")
```

```
[1] " message of type 'tutorial.Person' "
```

4.2.3 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
```

4.2.4 The toString method

4.2.5 The as.character method

4.2.6 The fileDescriptor method

4.2.7 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"
```

4.3 field descriptors

The class *FieldDescriptor* represents field descriptor in R. This is a wrapper S4 class around the `google::protobuf::FieldDescriptor` C++ 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 6: Description of slots for the `FieldDescriptor` S4 class

4.3.1 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];
}
```

4.3.2 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];
}
```

4.3.3 name

TODO

4.4 enum descriptors

The class *EnumDescriptor* is an R wrapper class around the C++ class `google::protobuf::EnumDescriptor`.

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 7: Description of slots for the `EnumDescriptor` S4 class

4.4.1 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   HOME   WORK
     0      1      2
```

4.4.2 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;
}
```

4.5 file descriptors

TODO: add content

4.6 service descriptors

TODO: add content

4.7 method descriptors

TODO: add content

5 Utilities

5.1 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)

[1] " message of type 'google.protobuf.EnumDescriptorProto' "

> asMessage(tutorial.Person$email)

[1] " message of type 'google.protobuf.FieldDescriptorProto' "

> asMessage(fileDescriptor(tutorial.Person))

[1] " message of type 'google.protobuf.FileDescriptorProto' "
```

5.2 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

5.3 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, {
+     id <- 2
+     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.

5.4 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.

5.5 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"
```

5.6 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 6.

```
> P("tutorial.Person")
```

```
[1] "descriptor for type 'tutorial.Person' "
```

```
> new(P("tutorial.Person"))

[1] " message of type 'tutorial.Person' "

> tutorial.Person

[1] "descriptor for type 'tutorial.Person' "

> new(tutorial.Person)

[1] " message of type 'tutorial.Person' "
```

6 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.

7 Plans for future releases

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. We plan to integrate this functionality into RProtoBuf. Saptarshi's package is available at http://ml.stat.purdue.edu/rhipe/doc/html/ProtoBuffers.html

Protocol buffers have a mechanism for remote procedure calls (rpc) that is not yet used by RProtoBuf, but we plan to 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.

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 involvment on RProtoBuf.

The user defined table mechasnism, implemented by Duncan Temple Lang for the purpose of the RObjectTables package allowed the dynamic symbol lookup (see section 6). Many thanks to Duncan for this amazing feature.