Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

RProtoBuf: Efficient Cross-Language Data Serialization in R

Dirk Eddelbuettel
Debian Project

Murray Stokely
Google, Inc

Jeroen Ooms
UCLA

Abstract

Modern data collection and analysis pipelines often involve a sophisticated mix of applications written in general purpose and specialized programming languages. Many formats commonly used to import and export data between different programs or systems, such as `CSV` or `JSON`, are verbose, inefficient, not type-safe, or tied to a specific programming language. Protocol Buffers are a popular method of serializing structured data between applications—while remaining independent of programming languages or operating systems. They offer a unique combination of features, performance, and maturity that seems particularly well suited for data-driven applications and numerical computing. The **RProtoBuf** package provides a complete interface to Protocol Buffers from the R environment for statistical computing. This paper outlines the general class of data serialization requirements for statistical computing, describes the implementation of the **RProtoBuf** package, and illustrates its use with example applications in large-scale data collection pipelines and web services.

Keywords: R, **Rcpp**, Protocol Buffers, serialization, cross-platform.

1. Introduction

Modern data collection and analysis pipelines increasingly involve collections of decoupled components in order to better manage software complexity through reusability, modularity, and fault isolation (Wegiel and Krintz 2010). These pipelines are frequently built using different programming languages for the different phases of data analysis — collection, cleaning, modeling, analysis, post-processing, and presentation — in order to take advantage of the unique combination of performance, speed of development, and library support offered by different environments and languages. Each stage of such a data analysis pipeline may produce intermediate results that need to be stored in a file, or sent over the network for further processing.

Given these requirements, how do we safely and efficiently share intermediate results between different applications, possibly written in different languages, and possibly running on different computer systems? In computer programming, *serialization* is the process of translating data structures, variables, and session state into a format that can be stored or transmitted and then reconstructed in the original form later (Cline 2013). Programming languages such as R, Julia, Java, and Python include built-in support for serialization, but the default formats are usually language-specific and thereby lock the user into a single environment.

Data analysts and researchers often use character-separated text formats such as `CSV` (Shafranovich 2005) to export and import data. However, anyone who has ever used `CSV` files will have noticed that this method has many limitations: it is restricted to tabular data, lacks type-safety, and has limited precision for numeric values. Moreover, ambiguities in the format itself frequently cause problems. For example, conventions on which characters is used as separator or decimal point vary by country. *Extensible Markup Language* (`XML`) is another well-established and widely-supported format with the ability to define just about any arbitrarily complex schema (Nolan and Temple Lang 2013). However, it pays for this complexity with comparatively large and verbose messages, and added complexity at the parsing side (which are somewhat mitigated by the availability of mature libraries and parsers). Because `XML` is text-based and has no native notion of numeric types or arrays, it usually not a very practical format to store numeric data sets as they appear in statistical applications.

A more modern format is *JavaScript ObjectNotation* (`JSON`), which is derived from the object literals of JavaScript, and already widely-used on the world wide web. Several R packages implement functions to parse and generate `JSON` data from R objects (Couture-Beil 2012; Temple Lang 2011; Ooms 2014). `JSON` natively supports arrays and four primitive types: numbers, strings, booleans, and null. However, as it too is a text-based format, numbers are stored as human-readable decimal notation which is inefficient and leads to loss of type (double versus integer) and precision. A number of binary formats based on `JSON` have been proposed that reduce the parsing cost and improve efficiency, but these formats are not widely supported. Furthermore, such formats lack a separate schema for the serialized data and thus still duplicate field names with each message sent over the network or stored in a file.

Once the data serialization needs of an application become complex enough, developers typically benefit from the use of an *interface description language*, or *IDL*. IDLs like Protocol Buffers (Google 2012), Apache Thrift (Apache Software Foundation 2013), and Apache Avro (Apache Software Foundation 2014) provide a compact well-documented schema for cross-language data structures and efficient binary interchange formats. Since the schema is provided separately from the data, the data can be efficiently encoded to minimize storage costs when compared with simple "schema-less" binary interchange formats. Protocol Buffers performs well in the comparison of such formats by Sumaray and Makki (2012).

This paper describes an R interface to Protocol Buffers, and is organized as follows. Section 2 provides a general high-level overview of Protocol Buffers as well as a basic motivation for their use. Section 3 describes the interactive R interface provided by the **RProtoBuf** package, and introduces the two main abstractions: *Messages* and *Descriptors*. Section 4 details the implementation details of the main S4 classes and methods. Section 5 describes the challenges of type coercion between R and other languages. Section 6 introduces a general R language schema for serializing arbitrary R objects and evaluates it against the serialization capabilities built directly into R. Sections 7 and 8 provide real-world use cases of **RProtoBuf** in MapReduce and web service environments, respectively, before Section 9 concludes.

2. Protocol Buffers

Protocol Buffers are a modern, language-neutral, platform-neutral, extensible mechanism for sharing and storing structured data. Some of the key features provided by Protocol Buffers for data analysis include:

- *Portable*: Enable users to send and receive data between applications as well as different computers or operating systems.
- *Efficient*: Data is serialized into a compact binary representation for transmission or storage.
- *Extensible*: New fields can be added to Protocol Buffer schemas in a forward-compatible way that does not break older applications.
- *Stable*: Protocol Buffers have been in wide use for over a decade.

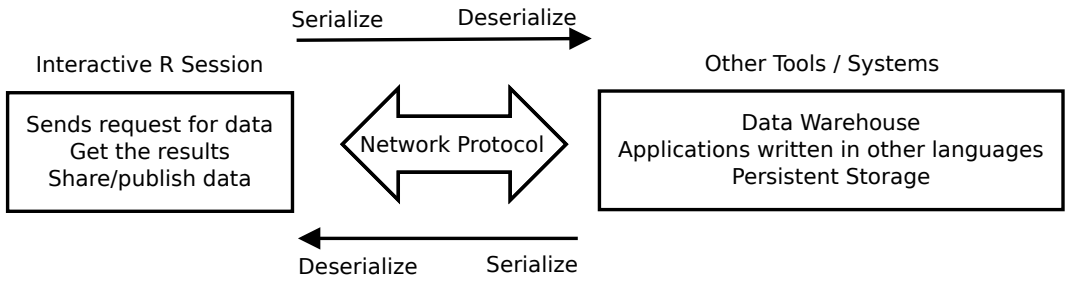


Figure 1: Example usage of Protocol Buffers.

Figure 1 illustrates an example communication work flow with Protocol Buffers and an interactive R session. Common use cases include populating a request remote-procedure call (RPC) Protocol Buffer in R that is then serialized and sent over the network to a remote server. The server would then deserialize the message, act on the request, and respond with a new Protocol Buffer over the network. The key difference to, say, a request to an **Rserve** (Urbanek 2003, 2013) instance is that the remote server may be implemented in any language.

While traditional IDLs have at times been criticized for code bloat and complexity, Protocol Buffers are based on a simple list and records model that is flexible and easy to use. The schema for structured Protocol Buffer data is defined in `.proto` files, which may contain one or more message types. Each message type has one or more fields. A field is specified with a unique number (called a *tag number*), a name, a value type, and a field rule specifying whether the field is optional, required, or repeated. The supported value types are numbers, enumerations, booleans, strings, raw bytes, or other nested message types. The `.proto` file syntax for defining the structure of Protocol Buffer data is described comprehensively on Google Code[1]. Table 1 shows an example `.proto` file that defines the `tutorial.Person` type[2]. The R code in the right column shows an example of creating a new message of this type and populating its fields.

[1]See `http://code.google.com/apis/protocolbuffers/docs/proto.html`.

[2]The compound name `tutorial.Person` in R is derived from the name of the message (*Person*) and the name of the package defined at the top of the `.proto` file in which it is defined (*tutorial*).

Schema : `addressbook.proto`

```
package tutorial;
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;
  }
  repeated PhoneNumber phone = 4;
}
```

Example R session

```
R> library("RProtoBuf")
R> p <- new(tutorial.Person, id=1,
+           name="Dirk")
R> p$name
[1] "Dirk"
R> p$name <- "Murray"
R> cat(as.character(p))
name: "Murray"
id: 1
R> serialize(p, NULL)
 [1] 0a 06 4d 75 72 72 61 79 10 01
R> class(p)
[1] "Message"
attr(,"package")
[1] "RProtoBuf"
```

Table 1: The schema representation from a `.proto` file for the `tutorial.Person` class (left) and simple R code for creating an object of this class and accessing its fields (right).

For added speed and efficiency, the C++, Java, and Python bindings to Protocol Buffers are used with a compiler that translates a Protocol Buffer schema description file (ending in `.proto`) into language-specific classes that can be used to create, read, write, and manipulate Protocol Buffer messages. The R interface, in contrast, uses a reflection-based API that makes some operations slightly slower but which is much more convenient for interactive data analysis. All messages in R have a single class structure, but different accessor methods are created at runtime based on the named fields of the specified message type, as described in the next section.

3. Basic usage: Messages and descriptors

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 representation of the message (often called the *payload*) to files and arbitrary binary R connections. The two fundamental building blocks of Protocol Buffers are *Messages* and *Descriptors*. Messages provide a common abstract encapsulation of structured data fields of the type specified in a Message Descriptor. Message Descriptors are defined in `.proto` files and define a schema for a particular named class of messages.

3.1. Importing message descriptors from `.proto` files

To create or parse a Protocol Buffer Message, one must first read in the message type specification from a `.proto` file. The `.proto` files are imported using the `readProtoFiles` function, which can either import a single file, all files in a directory, or every `.proto` file provided by a particular R package.

After importing proto files, the corresponding message descriptors are available by name from the `RProtoBuf:DescriptorPool` environment in the R search path. This environment is implemented with the user-defined tables framework from the **RObjectTables** package available from the OmegaHat project (Temple Lang 2012). Instead of being associated with a static hash table, this environment dynamically queries the in-memory database of loaded descriptors during normal variable lookup.

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

[1] "rexp.CMPLX"                    "rexp.REXP"
[3] "rexp.STRING"                   "rprotobuf.HelloWorldRequest"
[5] "rprotobuf.HelloWorldResponse" "tutorial.AddressBook"
[7] "tutorial.Person"
```

3.2. Creating a message

New messages are created with the `new` function which accepts a Message Descriptor and optionally a list of "name = value" pairs to set in the message.

```
R> p1 <- new(tutorial.Person)
R> p <- new(tutorial.Person, name = "Murray", id = 1)
```

3.3. 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.

```
R> p$name

[1] "Murray"

R> p$id

[1] 1

R> p$email <- "murray@stokely.org"
```

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:

```
R> p[["name"]] <- "Murray Stokely"
R> p[[ 2 ]] <- 3
R> p[["email"]]
[1] "murray@stokely.org"
```

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.

3.4. Display messages

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

```
R> p
[1] "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.

```
R> writeLines(as.character(p))
name: "Murray Stokely"
id: 3
email: "murray@stokely.org"
```

3.5. Serializing messages

One of the primary benefits of Protocol Buffers is the efficient binary wire-format representation. The `serialize` method is implemented for Protocol Buffer messages to serialize a message into a sequence of bytes (raw vector) that represents the message. The raw bytes can then be parsed back into the original message safely as long as the message type is known and its descriptor is available.

```
R> serialize(p, NULL)
 [1] 0a 0e 4d 75 72 72 61 79 20 53 74 6f 6b 65 6c 79 10 03 1a 12 6d 75
[23] 72 72 61 79 40 73 74 6f 6b 65 6c 79 2e 6f 72 67
```

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

```
R> tf1 <- tempfile()
R> serialize(p, tf1)
R> readBin(tf1, raw(0), 500)
 [1] 0a 0e 4d 75 72 72 61 79 20 53 74 6f 6b 65 6c 79 10 03 1a 12 6d 75
[23] 72 72 61 79 40 73 74 6f 6b 65 6c 79 2e 6f 72 67
```

Or to arbitrary binary connections:

```
R> tf2 <- tempfile()
R> con <- file(tf2, open = "wb")
R> serialize(p, con)
R> close(con)
R> readBin(tf2, raw(0), 500)
 [1] 0a 0e 4d 75 72 72 61 79 20 53 74 6f 6b 65 6c 79 10 03 1a 12 6d 75
[23] 72 72 61 79 40 73 74 6f 6b 65 6c 79 2e 6f 72 67
```

`serialize` can also be called in a more traditional object oriented fashion using the dollar operator.

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

Here, we first serialize to a file `tf1` before we serialize to a binary connection to file `tf2`.

3.6. Parsing messages

The **RProtoBuf** package defines the `read` and `readASCII` functions to read messages from files, raw vectors, or arbitrary connections. `read` expects to read the message payload from binary files or connections and `readASCII` parses the human-readable ASCII output that is created with `as.character`.

The binary representation of the message 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:

```
R> msg <- read(tutorial.Person, tf1)
R> writeLines(as.character(msg))
name: "Murray Stokely"
id: 3
email: "murray@stokely.org"
```

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

```
R> con <- file(tf2, open = "rb")
R> message <- read(tutorial.Person, con)
R> close(con)
R> writeLines(as.character(message))
name: "Murray Stokely"
id: 3
email: "murray@stokely.org"
```

Finally, the raw vector payload of the message can be used:

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

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

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

Here we read first from a file, then from a binary connection and lastly from a message payload.

4. Under the hood: S4 classes, methods, and pseudo methods

The **RProtoBuf** package uses the S4 system to store information about descriptors and messages. Using the S4 system allows the package to dispatch methods that are not generic in the S3 sense, such as `new` and `serialize`. Table 2 lists the six primary Message and Descriptor classes in **RProtoBuf**. Each R object contains an external pointer to an object managed by the `protobuf` C++ library, and the R objects make calls into more than 100 C++ functions that provide the glue code between the R language classes and the underlying C++ classes.

The **Rcpp** package (Eddelbuettel and François 2011; Eddelbuettel 2013) is used to facilitate this integration of the R and C++ code for these objects. Each method is wrapped individually which allows us to add user-friendly custom error handling, type coercion, and performance improvements at the cost of a more verbose implementation. The **RProtoBuf** package in many ways motivated the development of **Rcpp** Modules (Eddelbuettel and François 2014), which provide a more concise way of wrapping C++ functions and classes in a single entity.

The **RProtoBuf** package supports two forms for calling functions with these S4 classes:

- The functional dispatch mechanism of the the form `method(object, arguments)` (common to R), and

Class	Slots	Methods	Dynamic dispatch
Message	2	20	yes (field names)
Descriptor	2	16	yes (field names, enum types, nested types)
FieldDescriptor	4	18	no
EnumDescriptor	4	11	yes (enum constant names)
EnumValueDescriptor	3	6	no
FileDescriptor	3	6	yes (message/field definitions)

Table 2: Overview of class, slot, method and dispatch relationships.

- The traditional object oriented notation `object$method(arguments)`.

Additionally, **RProtoBuf** supports tab completion for all classes. Completion possibilities include pseudo-method names for all classes, plus *dynamic dispatch* on names or types specific to a given object. This functionality is implemented with the `.DollarNames` S3 generic function defined in the **utils** package that is included with R (R Core Team 2014).

4.1. Messages

The `Message` S4 class represents Protocol Buffer Messages and is the core abstraction of **RProtoBuf**. Each `Message` contains a pointer to a `Descriptor` which defines the schema of the data defined in the Message, as well as a number of `FieldDescriptors` for the individual fields of the message. A complete list of the slots and methods for `Messages` is available in Table 3.

```
R> new(tutorial.Person)
[1] "message of type 'tutorial.Person' with 0 fields set"
```

4.2. Descriptors

Descriptors describe the type of a Message. This includes what fields a message contains and what the types of those fields are. Message descriptors are represented in R by the *Descriptor* S4 class. The class contains the slots `pointer` and `type`. Similarly to messages, the `$` operator can be used to retrieve descriptors that are contained in the descriptor, or invoke pseudo-methods.

When **RProtoBuf** is first loaded it calls `readProtoFiles` to read in the example `addressbook.proto` file included with the package. The `tutorial.Person` descriptor and all other descriptors defined in the loaded `.proto` files are then available on the search path[3].

```
R> tutorial.Person$email
[1] "descriptor for field 'email' of type 'tutorial.Person' "
R> tutorial.Person$PhoneType
[1] "descriptor for enum 'PhoneType' of type 'tutorial.Person' with 3 values"
R> tutorial.Person$PhoneNumber
[1] "descriptor for type 'tutorial.Person.PhoneNumber' "
R> tutorial.Person.PhoneNumber
[1] "descriptor for type 'tutorial.Person.PhoneNumber' "
```

Table 4 provides a complete list of the slots and available methods for Descriptors.

4.3. Field descriptors

[3]This explains why the example in Table 1 lacked an explicit call to `readProtoFiles`.

Slot	Description
`pointer`	External pointer to the `Message` object of the C++ protobuf library. Documentation for the `Message` class is available from the Protocol Buffer project page.
`type`	Fully qualified name of the message. For example a `Person` message has its `type` slot set to `tutorial.Person`
Method	**Description**
`has`	Indicates if a message has a given field.
`clone`	Creates a clone of the message
`isInitialized`	Indicates if a message has all its required fields set
`serialize`	serialize a message to a file, binary connection, or raw vector
`clear`	Clear one or several fields of a message, or the entire message
`size`	The number of elements in a message field
`bytesize`	The number of bytes the message would take once serialized
`swap`	swap elements of a repeated field of a message
`set`	set elements of a repeated field
`fetch`	fetch elements of a repeated field
`setExtension`	set an extension of a message
`getExtension`	get the value of an extension of a message
`add`	add elements to a repeated field
`str`	the R structure of the message
`as.character`	character representation of a message
`toString`	character representation of a message (same as `as.character`)
`as.list`	converts message to a named R list
`update`	updates several fields of a message at once
`descriptor`	get the descriptor of the message type of this message
`fileDescriptor`	get the file descriptor of this message's descriptor

Table 3: Description of slots and methods for the `Message` S4 class.

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.
`type`	Fully qualified path of the message type.
Method	**Description**
`new`	Creates a prototype of a message described by this descriptor.
`read`	Reads a message from a file or binary connection.
`readASCII`	Read a message in ASCII format from a file or text connection.
`name`	Retrieve the name of the message type associated with this descriptor.
`as.character`	character representation of a descriptor
`toString`	character representation of a descriptor (same as `as.character`)
`as.list`	return a named list of the field, enum, and nested descriptors included in this descriptor.
`asMessage`	return DescriptorProto message.
`fileDescriptor`	Retrieve the file descriptor of this descriptor.
`containing_type`	Retrieve the descriptor describing the message type containing this descriptor.
`field_count`	Return the number of fields in this descriptor.
`field`	Return the descriptor for the specified field in this descriptor.
`nested_type_count`	The number of nested types in this descriptor.
`nested_type`	Return the descriptor for the specified nested type in this descriptor.
`enum_type_count`	The number of enum types in this descriptor.
`enum_type`	Return the descriptor for the specified enum type in this descriptor.

Table 4: Description of slots and methods for the `Descriptor` S4 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
Method	**Description**
`as.character`	Character representation of a descriptor
`toString`	Character representation of a descriptor (same as `as.character`)
`asMessage`	Return FieldDescriptorProto message.
`name`	Return the name of the field descriptor.
`fileDescriptor`	Return the fileDescriptor where this field is defined.
`containing_type`	Return the containing descriptor of this field.
`is_extension`	Return TRUE if this field is an extension.
`number`	Gets the declared tag number of the field.
`type`	Gets the type of the field.
`cpp_type`	Gets the C++ type of the field.
`label`	Gets the label of a field (optional, required, or repeated).
`is_repeated`	Return TRUE if this field is repeated.
`is_required`	Return TRUE if this field is required.
`is_optional`	Return TRUE if this field is optional.
`has_default_value`	Return TRUE if this field has a default value.
`default_value`	Return the default value.
`message_type`	Return the message type if this is a message type field.
`enum_type`	Return the enum type if this is an enum type field.

Table 5: Description of slots and methods for the `FieldDescriptor` S4 class.

The class *FieldDescriptor* represents field descriptors in R. This is a wrapper S4 class around the `google::protobuf::FieldDescriptor` C++ class. Table 5 describes the methods defined for the `FieldDescriptor` class.

4.4. Enum descriptors

The class *EnumDescriptor* represents enum descriptors in R. This is a wrapper S4 class around the `google::protobuf::EnumDescriptor` C++ class. Table 6 describes the methods defined for the `EnumDescriptor` class.

The `$` operator can be used to retrieve the value of enum constants contained in the EnumDescriptor, or to invoke pseudo-methods.

The `EnumDescriptor` contains information about what values this type defines, while the `EnumValueDescriptor` describes an individual enum constant of a particular type.

```
R> tutorial.Person$PhoneType

[1] "descriptor for enum 'PhoneType' of type 'tutorial.Person' with 3 values"

R> tutorial.Person$PhoneType$WORK

[1] 2
```

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
Method	**Description**
`as.list`	return a named integer vector with the values of the enum and their names.
`as.character`	character representation of a descriptor
`toString`	character representation of a descriptor (same as `as.character`)
`asMessage`	return EnumDescriptorProto message.
`name`	Return the name of the enum descriptor.
`fileDescriptor`	Return the fileDescriptor where this field is defined.
`containing_type`	Return the containing descriptor of this field.
`length`	Return the number of constants in this enum.
`has`	Return TRUE if this enum contains the specified named constant string.
`value_count`	Return the number of constants in this enum (same as `length`).
`value`	Return the EnumValueDescriptor of an enum value of specified index, name, or number.

Table 6: Description of slots and methods for the `EnumDescriptor` S4 class.

4.5. Enum value descriptors

The class *EnumValueDescriptor* represents enumeration value descriptors in R. This is a wrapper S4 class around the `google::protobuf::EnumValueDescriptor` C++ class. Table 7 describes the methods defined for the `EnumValueDescriptor` class.

The `$` operator can be used to invoke pseudo-methods.

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

4.6. File descriptors

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

The `$` operator can be used to retrieve named fields defined in the FileDescriptor, or to invoke pseudo-methods.

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
Method	**Description**
`number`	return the number of this EnumValueDescriptor.
`name`	Return the name of the enum value descriptor.
`enum_type`	return the EnumDescriptor type of this EnumValueDescriptor.
`as.character`	character representation of a descriptor.
`toString`	character representation of a descriptor (same as `as.character`).
`asMessage`	return EnumValueDescriptorProto message.

Table 7: Description of slots and methods for the `EnumValueDescriptor` S4 class.

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.
Method	**Description**
`name`	Return the filename for this FileDescriptorProto.
`package`	Return the file-level package name specified in this FileDescriptorProto.
`as.character`	character representation of a descriptor.
`toString`	character representation of a descriptor (same as `as.character`).
`asMessage`	return FileDescriptorProto message.
`as.list`	return named list of descriptors defined in this file descriptor.

Table 8: Description of slots and methods for the `FileDescriptor` S4 class.

```
R> f <- tutorial.Person$fileDescriptor()
R> f
[1] "file descriptor for package tutorial (/usr/local/lib/R/site-library/RProtoBuf/proto/a
R> f$Person
[1] "descriptor for type 'tutorial.Person' "
```

5. Type coercion

One of the benefits of using an Interface Definition Language (IDL) like Protocol Buffers is that it provides a highly portable basic type system. This permits different language and hardware implementations to map to the most appropriate type in different environments.

Table 9 details the correspondence between the field type and the type of data that is retrieved by `$` and `[[` extractors. Three types in particular need further attention due to specific differences in the R language: booleans, unsigned integers, and 64-bit integers.

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	`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 9: Correspondence between field type and R type retrieved by the extractors. Note that 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.

5.1. Booleans

R booleans can accept three values: `TRUE`, `FALSE`, and `NA`. However, most other languages,

including the Protocol Buffer schema, only accept `TRUE` or `FALSE`. This means that we simply can not store R logical vectors that include all three possible values as booleans. The library will refuse to store `NA`s in Protocol Buffer boolean fields, and users must instead choose another type (such as enum or integer) capable of storing three distinct values.

```
R> a <- new(JSSPaper.Example1)
R> a$optional_bool <- TRUE
R> a$optional_bool <- FALSE
R> a$optional_bool <- NA
Error: NA boolean values can not be stored in bool Protocol Buffer fields
```

5.2. Unsigned integers

R lacks a native unsigned integer type. Values between 2^{31} and $2^{32} - 1$ read from unsigned integer Protocol Buffer fields must be stored as doubles in R.

```
R> as.integer(2^31-1)
[1] 2147483647
R> as.integer(2^31 - 1) + as.integer(1)
[1] NA
R> 2^31
[1] 2.147e+09
R> class(2^31)
[1] "numeric"
```

5.3. 64-bit integers

R also does not support the native 64-bit integer type. Numeric vectors with integer values greater or equal to 2^{31} can only be stored as floating-point double precision variables. This conversion incurs a loss of precision, and R loses the ability to distinguish between some distinct integer variables:

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

Most modern languages do have support for 64-bit integer values, which becomes problematic when **RProtoBuf** is used to exchange data with a system that requires this integer type. To work around this, **RProtoBuf** allows users to get and set 64-bit integer values by specifying them as character strings.

On 64-bit platforms, character strings representing large decimal numbers will be coerced to `int64` during assignment to 64-bit Protocol Buffer types to work around the lack of native 64-bit types in R itself. The values are stored as distinct `int64` values in memory. But when

accessed from R language code, they will be coerced into numeric (floating-point) values. If the full 64-bit precision is required, the `RProtoBuf.int64AsString` option can be set to `TRUE` to return `int64` values from messages as character strings. Such character values are useful because they can accurately be used as unique identifiers, and can easily be passed to R packages such as **int64** (François 2011) or **bit64** (Oehlschlägel 2012) which represent 64-bit integers in R.

6. Converting R data structures into Protocol Buffers

The previous sections discussed functionality in the **RProtoBuf** package for creating, manipulating, parsing, and serializing Protocol Buffer messages of a defined schema. This is useful when there are pre-existing systems with defined schemas or significant software components written in other languages that need to be accessed from within R. The package also provides methods for converting arbitrary R data structures into Protocol Buffers and vice versa with a universal R object schema. The `serialize_pb` and `unserialize_pb` functions serialize arbitrary R objects into a universal Protocol Buffer message:

```
R> msg <- serialize_pb(iris, NULL)
R> identical(iris, unserialize_pb(msg))

[1] TRUE
```

In order to accomplish this, **RProtoBuf** uses the same catch-all `proto` schema used by **RHIPE** for exchanging R data with Hadoop (Guha 2010). This schema, which we will refer to as `rexp.proto`, is printed in the appendix. The Protocol Buffer messages generated by **RProtoBuf** and **RHIPE** are naturally compatible between the two systems because they use the same schema. This shows the power of using a schema-based cross-platform format such as Protocol Buffers: interoperability is achieved without effort or close coordination.

The `rexp.proto` schema supports all main R storage types holding *data*. These include `NULL`, `list` and vectors of type `logical`, `character`, `double`, `integer`, and `complex`. In addition, every type can contain a named set of attributes, as is the case in R. The `rexp.proto` schema does not support some of the special R specific storage types, such as `function`, `language` or `environment`. Such objects have no native equivalent type in Protocol Buffers, and have little meaning outside the context of R. When serializing R objects using `serialize_pb`, values or attributes of unsupported types are skipped with a warning. If the user really wishes to serialize these objects, they need to be converted into a supported type. For example, the can use `deparse` to convert functions or language objects into strings, or `as.list` for environments.

6.1. Evaluation: Converting R data sets

To illustrate how this method works, we attempt to convert all of the built-in data sets from R into this serialized Protocol Buffer representation.

```
R> datasets <- as.data.frame(data(package="datasets")$results)
R> datasets$name <- sub("\\s+.*$", "", datasets$Item)
R> n <- nrow(datasets)
```

There are 103 standard data sets included in the **datasets** package included with R. These data sets include data frames, matrices, time series, tables lists, and some more exotic data classes. The `can_serialize_pb` method is used to determine which of those can fully be converted to the `rexp.proto` Protocol Buffer representation. This method simply checks if any of the values or attributes in an object is of an unsupported type:

```
R> m <- sum(sapply(datasets$name, function(x) can_serialize_pb(get(x))))
```

96 data sets can be converted to Protocol Buffers without loss of information (93%). Upon closer inspection, all other data sets are objects of class `nfnGroupedData`. This class represents a special type of data frame that has some additional attributes (such as a *formula* object) used by the **nlme** package (Pinheiro *et al.* 2013). Because formulas are R *language* objects, they have little meaning to other systems, and are not supported by the `rexp.proto` descriptor. When `serialize_pb` is used on objects of this class, it will serialize the data frame and all attributes, except for the formula.

```
R> attr(CO2, "formula")

uptake ~ conc | Plant
<environment: R_EmptyEnv>

R> msg <- serialize_pb(CO2, NULL)
R> object <- unserialize_pb(msg)
R> identical(CO2, object)

[1] FALSE

R> identical(class(CO2), class(object))

[1] TRUE

R> identical(dim(CO2), dim(object))

[1] TRUE

R> attr(object, "formula")

list()
NULL
```

6.2. Compression performance

This section compares how many bytes are used to store data sets using four different methods:

- normal R serialization (Tierney 2003),
- R serialization followed by gzip,
- normal Protocol Buffer serialization, and
- Protocol Buffer serialization followed by gzip.

Table 10 shows the sizes of 50 sample R data sets as returned by object.size() compared to the serialized sizes. Note that Protocol Buffer serialization results in slightly smaller byte streams compared to native R serialization in most cases, but this difference disappears if the results are compressed with gzip. One takeaway from this table is that the universal R object schema included in **RProtoBuf** does not in general provide any significant saving in file size compared to the normal serialization mechanism in R. The benefits of **RProtoBuf** accrue more naturally in applications where multiple programming languages are involved, or when a more concise application-specific schema has been defined. The example in the next section satisfies both of these conditions.

7. Application: Distributed data collection with MapReduce

Many large data sets in fields such as particle physics and information processing are stored in binned or histogram form in order to reduce the data storage requirements (Scott 2009). In the last decade, the MapReduce programming model (Dean and Ghemawat 2008) has emerged as a popular design pattern that enables the processing of very large data sets on large compute clusters.

Many types of data analysis over large data sets may involve very rare phenomenon or deal with highly skewed data sets or inflexible raw data storage systems from which unbiased sampling is not feasible. In such situations, MapReduce and binning may be combined as a pre-processing step for a wide range of statistical and scientific analyses (Blocker and Meng 2013).

There are two common patterns for generating histograms of large data sets in a single pass with MapReduce. In the first method, each mapper task generates a histogram over a subset of the data that it has been assigned, serializes this histogram and sends it to one or more reducer tasks which merge the intermediate histograms from the mappers.

In the second method, illustrated in Figure 2, each mapper rounds a data point to a bucket width and outputs that bucket as a key and '1' as a value. Reducers then sum up all of the values with the same key and output to a data store.

In both methods, the mapper tasks must choose identical bucket boundaries in advance if we are to construct the histogram in a single pass, even though they are analyzing disjoint parts of the input set that may cover different ranges. All distributed tasks involved in the pre-processing as well as any downstream data analysis tasks must share a schema of the histogram representation to coordinate effectively.

The **HistogramTools** package (Stokely 2013) enhances **RProtoBuf** by providing a concise schema for R histogram objects:

```
package HistogramTools;

message HistogramState {
  repeated double breaks = 1;
  repeated int32 counts = 2;
  optional string name = 3;
}
```

This HistogramState message type is designed to be helpful if some of the Map or Reduce

Data Set	object.size	R Serialization		RProtoBuf Serial.	
		default	gzipped	default	gzipped
uspop	584	268	172	211	148
Titanic	1960	633	257	481	249
volcano	42656	42517	5226	42476	4232
euro.cross	2728	1319	910	1207	891
attenu	14568	8234	2165	7771	2336
ToothGrowth	2568	1486	349	1239	391
lynx	1344	1028	429	971	404
nottem	2352	2036	627	1979	641
sleep	2752	746	282	483	260
co2	4176	3860	1473	3803	1453
austres	1144	828	439	771	410
ability.cov	1944	716	357	589	341
EuStockMarkets	60664	59785	21232	59674	19882
treering	64272	63956	17647	63900	17758
freeny.x	1944	1445	1311	1372	1289
Puromycin	2088	813	306	620	320
warpbreaks	2768	1231	310	811	343
BOD	1088	334	182	226	168
sunspots	22992	22676	6482	22620	6742
beaver2	4184	3423	751	3468	840
anscombe	2424	991	375	884	352
esoph	5624	3111	548	2240	665
PlantGrowth	1680	646	303	459	314
infert	15848	14328	1172	13197	1404
BJsales	1632	1316	496	1259	465
stackloss	1688	917	293	844	283
crimtab	7936	4641	713	1655	576
LifeCycleSavings	6048	3014	1420	2825	1407
Harman74.cor	9144	6056	2045	5861	2070
nhtemp	912	596	240	539	223
faithful	5136	4543	1339	4936	1776
freeny	5296	2465	1518	2271	1507
discoveries	1232	916	199	859	180
state.x77	7168	4251	1754	4068	1756
pressure	1096	498	277	427	273
fdeaths	1008	692	291	635	272
euro	976	264	186	202	161
LakeHuron	1216	900	420	843	404
mtcars	6736	3798	1204	3633	1206
precip	4992	1793	813	1615	815
state.area	440	422	246	405	235
attitude	3024	1990	544	1920	561
randu	10496	9794	8859	10441	9558
state.name	3088	844	408	724	415
airquality	5496	4551	1241	2874	1294
airmiles	624	308	170	251	148
quakes	33112	32246	9898	29063	11595
islands	3496	1232	563	1098	561
OrchardSprays	3600	2164	445	1897	483
WWWusage	1232	916	274	859	251
Relative Size	100%	83.7%	25.3%	80.1%	25.6%

Table 10: Serialization sizes for default serialization in R and **RProtoBuf** for 50 R data sets.

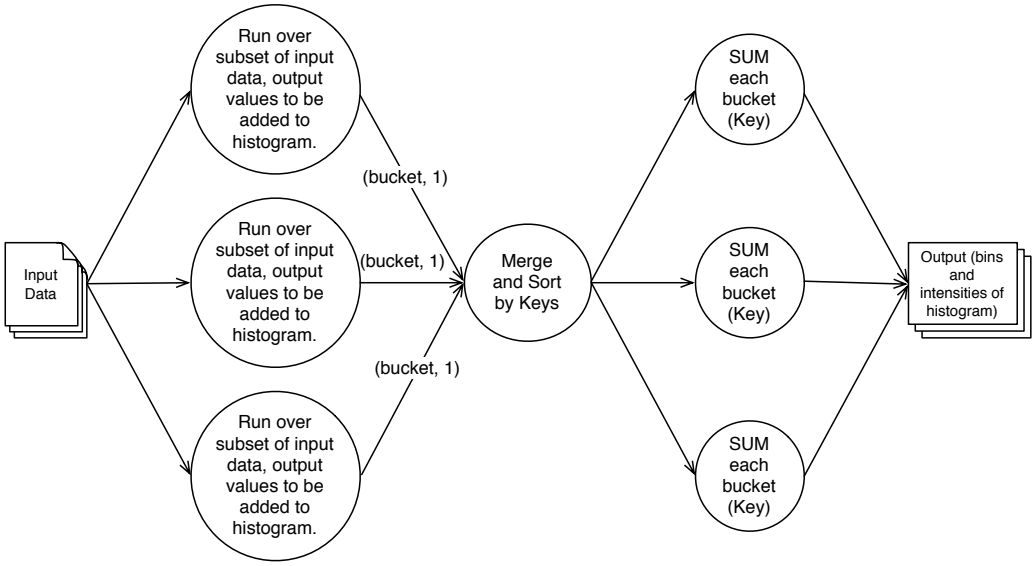


Figure 2: Diagram of MapReduce histogram generation pattern.

tasks are written in R, or if those components are written in other languages and only the resulting output histograms need to be manipulated in R. For example, to create HistogramState messages in Python for later consumption by R, we first compile the `histogram.proto` descriptor into a python module using the `protoc` compiler:

```
protoc histogram.proto --python_out=.
```

This generates a Python module called `histogram_pb2.py`, containing both the descriptor information as well as methods to read and manipulate the histogram message data. The following simple Python script uses this generated module to create a histogram (to which breakpoints and binned data are added), and writes out the Protocol Buffer representation to a file:

```
from histogram_pb2 import HistogramState;

hist = HistogramState()

hist.counts.extend([2, 6, 2, 4, 6])
hist.breaks.extend(range(6))
hist.name="Example Histogram Created in Python"

outfile = open("/tmp/hist.pb", "wb")
outfile.write(hist.SerializeToString())
outfile.close()
```

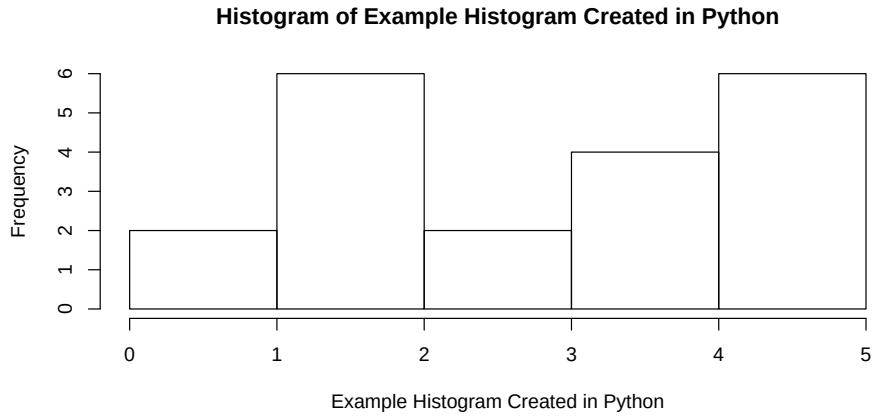
The Protocol Buffer can then be read into R and converted to a native R histogram object for plotting. Here, the schema is read first, then the (serialized) histogram is read into the variable `hist` which is then converted a histogram object which is display as a plot.

```
library("RProtoBuf")
library("HistogramTools")

readProtoFiles(package="HistogramTools")

hist <- HistogramTools.HistogramState$read("/tmp/hist.pb")
hist
[1] "message of type 'HistogramTools.HistogramState' with 3 fields set"

plot(as.histogram(hist))
```



One of the authors has used this design pattern for several large-scale studies of distributed storage systems (Stokely *et al.* 2012; Albrecht *et al.* 2013).

8. Application: Data interchange in web services

As described earlier, the primary application of Protocol Buffers is data interchange in the context of inter-system communications. Network protocols such as HTTP provide mechanisms for client-server communication, i.e., how to initiate requests, authenticate, send messages, etc. However, network protocols generally do not regulate the *content* of messages: they allow transfer of any media type, such as web pages, static files or multimedia content. When designing systems where various components require exchange of specific data structures, we need something on top of the network protocol that prescribes how these structures are to be represented in messages (buffers) on the network. Protocol Buffers solve exactly this problem by providing a cross-platform method for serializing arbitrary structures into well defined messages, which can then be exchanged using any protocol. The descriptors (`.proto` files) are used to formally define the interface of a remote API or network application. Libraries to parse and generate protobuf messages are available for many programming languages, making it relatively straightforward to implement clients and servers.

8.1. Interacting with R through HTTPS and Protocol Buffers

One example of a system that supports Protocol Buffers to interact with R is OpenCPU (Ooms

2013). OpenCPU is a framework for embedded statistical computation and reproducible research based on R and LaTeX. It exposes a HTTP(S) API to access and manipulate R objects and allows for performing remote R function calls. Clients do not need to understand or generate any R code: HTTP requests are automatically mapped to function calls, and arguments/return values can be posted/retrieved using several data interchange formats, such as Protocol Buffers. OpenCPU uses the `serialize_pb` and `unserialize_pb` functions from the **RProtoBuf** package to convert between R objects and protobuf messages. Therefore, clients need the `rexp.proto` descriptor mentioned earlier to parse and generate protobuf messages when interacting with OpenCPU.

8.2. HTTP GET: Retrieving an R object

The `HTTP GET` method is used to read a resource from OpenCPU. For example, to access the data set `Animals` from the package `MASS`, a client performs the following HTTP request:

```
GET https://public.opencpu.org/ocpu/library/MASS/data/Animals/pb
```

The postfix `/pb` in the URL tells the server to send this object in the form of a protobuf message. Alternative formats include `/json`, `/csv`, `/rds` and others. If the request is successful, OpenCPU returns the serialized object with HTTP status code 200 and HTTP response header `Content-Type: application/x-protobuf`. The latter is the conventional MIME type that formally notifies the client to interpret the response as a protobuf message.

Because both HTTP and Protocol Buffers have libraries available for many languages, clients can be implemented in just a few lines of code. Below is example code for both R and Python that retrieves a data set from R with OpenCPU using a protobuf message. In R, we use the HTTP client from the `httr` package (Wickham 2014). In this example we download a data set which is part of the base R distribution, so we can verify that the object was transferred without loss of information.

```
R> library("RProtoBuf")
R> library("httr")
R> req <- GET('https://public.opencpu.org/ocpu/library/MASS/data/Animals/pb')
R> output <- unserialize_pb(req$content)
R> identical(output, MASS::Animals)
```

This code suggests a method for exchanging objects between R servers, however this might as well be done without Protocol Buffers. The main advantage of using an inter-operable format is that we can actually access R objects from within another programming language. For example, in a very similar fashion we can retrieve the same data set in a Python client. To parse messages in Python, we first compile the `rexp.proto` descriptor into a python module using the `protoc` compiler:

```
protoc rexp.proto --python_out=.
```

This generates Python module called `rexp_pb2.py`, containing both the descriptor information as well as methods to read and manipulate the R object message. In the example below we use the HTTP client from the `urllib2` module.

```
import urllib2
from rexp_pb2 import REXP

req = urllib2.Request('https://public.opencpu.org/ocpu/library/MASS/data/Animals/pb')
res = urllib2.urlopen(req)

msg = REXP()
msg.ParseFromString(res.read())
print(msg)
```

The `msg` object contains all data from the Animals data set. From here we can easily extract the desired fields for further use in Python.

8.3. HTTP POST: Calling an R function

The example above shows how the `HTTP GET` method retrieves a resource from OpenCPU, for example an R object. The `HTTP POST` method on the other hand is used for calling functions and running scripts, which is the primary purpose of the framework. As before, the `/pb` postfix requests to retrieve the output as a protobuf message, in this case the function return value. However, OpenCPU allows us to supply the arguments of the function call in the form of protobuf messages as well. This is a bit more work, because clients needs to both generate messages containing R objects to post to the server, as well as retrieve and parse protobuf messages returned by the server. Using Protocol Buffers to post function arguments is not required, and for simple (scalar) arguments the standard `application/x-www-form-urlencoded` format might be sufficient. However, with Protocol Buffers the client can perform function calls with more complex arguments such as R vectors or lists. The result is a complete RPC system to do arbitrary R function calls from within any programming language.

The following example R client code performs the remote function call `stats::rnorm(n=42, mean=100)`. The function arguments (in this case `n` and `mean`) as well as the return value (a vector with 42 random numbers) are transferred using a protobuf message. RPC in OpenCPU works like the `do.call` function in R, hence all arguments are contained within a list.

```
R> library("httr")
R> library("RProtoBuf")
R> args <- list(n=42, mean=100)
R> payload <- serialize_pb(args, NULL)
R> req <- POST (
+    url = "https://public.opencpu.org/ocpu/library/stats/R/rnorm/pb",
+    body = payload,
+    add_headers (
+      "Content-Type" = "application/x-protobuf"
+    )
+ )
R> output <- unserialize_pb(req$content)
R> print(output)
```

The OpenCPU server basically performs the following steps to process the above RPC request:

```
R> fnargs <- unserialize_pb(inputmsg)
R> val <- do.call(stats::rnorm, fnargs)
R> outputmsg <- serialize_pb(val)
```

9. Summary

Over the past decade, many formats for interoperable data exchange have become available, each with their unique features, strengths and weaknesses. Text based formats such as CSV and JSON are easy to use, and will likely remain popular among statisticians for many years to come. However, in the context of increasingly complex analysis stacks and applications involving distributed computing as well as mixed language analysis pipelines, choosing a more sophisticated data interchange format may reap considerable benefits. The Protocol Buffers standard and library offer a unique combination of features, performance, and maturity, that seems particularly well suited for data-driven applications and numerical computing.

The **RProtoBuf** package builds on the Protocol Buffers C++ library, and extends the R system with the ability to create, read, write, parse, and manipulate Protocol Buffer messages. **RProtoBuf** has been used extensively inside Google for the past three years by statisticians, analysts, and software engineers. At the time of this writing there are over 300 active users of **RProtoBuf** using it to read data from and otherwise interact with distributed systems written in C++, Java, Python, and other languages. We hope that making Protocol Buffers available to the R community will contribute towards better software integration and allow for building even more advanced applications and analysis pipelines with R.

Acknowledgments

The first versions of **RProtoBuf** were written during 2009 - 2010. Very significant contributions, both in code and design, were made by Romain François whose continued influence on design and code is greatly appreciated. Several features of the package reflect the design of the **rJava** package by Simon Urbanek. The user-defined table mechanism, implemented by Duncan Temple Lang for the purpose of the **RObjectTables** package, allows for the dynamic symbol lookup. Kenton Varda was generous with his time in reviewing code and explaining obscure Protocol Buffer semantics. Karl Millar was very helpful in reviewing code and offering suggestions. Saptarshi Guha's work on RHIPE and implementation of a universal message type for R language objects allowed us to add the `serialize_pb` and `unserialize_pb` methods for turning arbitrary R objects into Protocol Buffers without a specialized pre-defined schema.

Appendix: The rexp.proto schema descriptor

Below a print of the `rexp.proto` schema (originally designed by Guha (2010)) that is included with the **RProtoBuf** package and used by `serialize_pb` and `unserialize_pb`.

```
package rexp;

message REXP {
  enum RClass {
    STRING = 0;
    RAW = 1;
    REAL = 2;
    COMPLEX = 3;
    INTEGER = 4;
    LIST = 5;
    LOGICAL = 6;
    NULLTYPE = 7;
  }
  enum RBOOLEAN {
    F=0;
    T=1;
    NA=2;
  }

  required RClass rclass = 1 ;
  repeated double realValue = 2 [packed=true];
  repeated sint32 intValue = 3 [packed=true];
  repeated RBOOLEAN booleanValue = 4;
  repeated STRING stringValue = 5;
  optional bytes rawValue = 6;
  repeated CMPLX complexValue = 7;
  repeated REXP rexpValue = 8;
  repeated string attrName = 11;
  repeated REXP attrValue = 12;
}
message STRING {
  optional string strval = 1;
  optional bool isNA = 2 [default=false];
}
message CMPLX {
  optional double real = 1 [default=0];
  required double imag = 2;
}
```

References

Albrecht C, Merchant A, Stokely M, Waliji M, Labelle F, Coehlo N, Shi X, Schrock E (2013). "Janus: Optimal Flash Provisioning for Cloud Storage Workloads." In *Proceedings of the USENIX Annual Technical Conference*, pp. 91–102. 2560 Ninth Street, Suite 215, Berkeley, CA 94710, USA. URL https://www.usenix.org/system/files/conference/atc13/atc13-albrecht.pdf.

Apache Software Foundation (2013). "Apache Thrift." Software Framework for Scalable Cross-Language Services, Version 0.9.1, URL http://thrift.apache.org.

Apache Software Foundation (2014). "Apache Avro." Data Serialization System, Version 1.7.6, URL http://avro.apache.org.

Blocker AW, Meng XL (2013). "The Potential and Perils of Preprocessing: Building new Foundations." *Bernoulli*, **19**(4), 1176–1211. doi:10.3150/13-BEJSP16. URL http://dx.doi.org/10.3150/13-BEJSP16.

Cline M (2013). "C++ FAQ." *Also available as http://www. parashift. com/c++-faq-lite/index. html*.

Couture-Beil A (2012). *rjson: JSON for R*. R package version 0.2.10, URL http://CRAN.R-project.org/package=rjson.

Dean J, Ghemawat S (2008). "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, **51**(1), 107–113.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Springer-Verlag.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18.

Eddelbuettel D, François R (2014). *Exposing C++ Functions and Classes with Rcpp Modules*. Vignette included in R package Rcpp, URL http://CRAN.R-project.org/package=Rcpp.

François R (2011). *int64: 64 Bit Integer Types*. R package version 1.1.2, URL http://CRAN.R-project.org/package=int64.

Google (2012). *Protocol Buffers: Developer Guide*. URL http://code.google.com/apis/protocolbuffers/docs/overview.html.

Guha S (2010). *RHIPE: A Distributed Environment for the Analysis of Large and Complex Datasets*. URL http://www.stat.purdue.edu/~sguha/rhipe/.

Nolan D, Temple Lang D (2013). *XML and Web Technologies for Data Sciences with R*. Springer-Verlag.

Oehlschlägel J (2012). *bit64: A S3 class for Vectors of 64bit Integers*. R package version 0.9-3, URL http://CRAN.R-project.org/package=bit64.

Ooms J (2013). *OpenCPU System for Embedded Statistical Computation and Reproducible Research*. R package version 1.2.2, URL http://www.opencpu.org.

Ooms J (2014). *jsonlite: A Smarter JSON Encoder/Decoder for R*. R package version 0.9.4, URL http://github.com/jeroenooms/jsonlite#readme.

Pinheiro J, Bates D, DebRoy S, Sarkar D, EISPACK authors, R Core (2013). *nlme: Linear and Nonlinear Mixed Effects Models*. R package version 3.1-113, URL http://CRAN.R-project.org/package=nlme.

R Core Team (2014). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Scott DW (2009). *Multivariate Density Estimation: Theory, Practice, and Visualization*, volume 383. Wiley. com.

Shafranovich Y (2005). "Common Format and Mime Type for Comma-Separated Values (csv) Files." URL http://tools.ietf.org/html/rfc4180.

Stokely M (2013). *HistogramTools: Utility Functions for R Histograms*. R package version 0.3, URL https://r-forge.r-project.org/projects/histogramtools/.

Stokely M, Mehrabian A, Albrecht C, Labelle F, Merchant A (2012). "Projecting Disk Usage Based on Historical Trends in a Cloud Environment." In *ScienceCloud 2012 Proceedings of the 3rd International Workshop on Scientific Cloud Computing*, pp. 63–70.

Sumaray A, Makki SK (2012). "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform." In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pp. 48:1–48:6. ACM, New York, NY, USA. ISBN 978-1-4503-1172-4. doi:10.1145/2184751.2184810. URL http://doi.acm.org/10.1145/2184751.2184810.

Temple Lang D (2011). *RJSONIO: Serialize R objects to JSON, JavaScript Object Notation*. R package version 0.96-0, URL http://CRAN.R-project.org/package=RJSONIO.

Temple Lang D (2012). *User-Defined Tables in the R Search Path*. URL http://www.omegahat.org/RObjectTables/RObjectTables.pdf.

Tierney L (2003). "A New Serialization Mechanism for R." URL http://www.cs.uiowa.edu/~luke/R/serialize/serialize.ps.

Urbanek S (2003). "Rserve: A Fast Way to Provide R Functionality to Applications." In K Hornik, F Leisch, A Zeileis (eds.), *Proceedings of the 3rd International Workshop on Distributed Statistical Computing, Vienna, Austria*. ISSN 1609-395X, URL http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Proceedings/.

Urbanek S (2013). *Rserve: Binary R server*. R package version 1.7-3, URL http://CRAN.R-Project.org/package=Rserve.

Wegiel M, Krintz C (2010). "Cross-language, Type-safe, and Transparent Object Sharing for Co-located Managed Runtimes." *SIGPLAN Not.*, **45**(10), 223–240. ISSN 0362-1340. doi:10.1145/1932682.1869479. URL http://doi.acm.org/10.1145/1932682.1869479.

Wickham H (2014). *httr: Tools for Working with URLs and HTTP*. R package version 0.3, URL http://CRAN.R-project.org/package=httr.

Affiliation:

Dirk Eddelbuettel
Debian Project
River Forest, IL, USA
E-mail: edd@debian.org
URL: http://dirk.eddelbuettel.com

Murray Stokely
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA, USA
E-mail: mstokely@google.com
URL: http://www.stokely.org/

Jeroen Ooms
UCLA Department of Statistics
University of California
Los Angeles, CA, USA
E-mail: jeroen.ooms@stat.ucla.edu
URL: http://jeroenooms.github.io

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd