RProtoBuf: Efficient Cross-Language Data Serialization in R

Dirk Eddelbuettel
Debian Project

Murray Stokely
Google, Inc

Jeroen Ooms
University of California, Los Angeles

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 (**?**). 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 states into a format that can be stored or transmitted and then later reconstructed in the original form (**?**). 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 (**?**)

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 are used as separator or decimal point vary by country. *Extensible markup language* (XML) is a well-established and widely-supported format with the ability to define just about any arbitrarily complex schema (**?**). However, it pays for this complexity with comparatively large and verbose messages, and added complexity at the parsing side (these problems 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 is usually not a very practical format to store numeric data sets as they appear in statistical applications.

A more modern format is *JavaScript object notation* (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 (**???**). JSON natively supports arrays and four primitive types: numbers, strings, booleans, and null. However, as it also is a text-based format, numbers are stored in 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 (**?**), Apache Thrift (**?**), and Apache Avro (**?**) 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 perform well in the comparison of such formats by **?**.

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 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 compares it to 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.

This vignette corresponds to the paper published in the *Journal of Statistical Software*. For citations, please see the output of R function `citation("RProtoBuf")`.

This version corresponds to **RProtoBuf** version 0.4.14.1 and was typeset on February 07, 2020.

2. Protocol Buffers

Protocol Buffers are a modern, language-neutral, platform-neutral, extensible mechanism for sharing and storing structured data. Some of their features, particularly in the context of

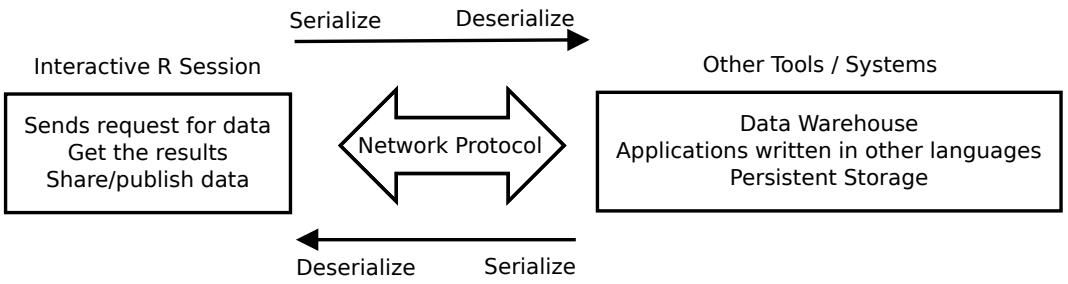


Figure 1: Example usage of Protocol Buffers.

data analysis, are:

- *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 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 deserializes the message, acts on the request, and responds with a new Protocol Buffer over the network. The key difference to, say, a request to an **Rserve** (**??**) 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 at `http://code.google.com/apis/protocolbuffers/docs/proto.html`. Table 1 shows an example `.proto` file that defines the `tutorial.Person` type[1]. The R code in the right column shows an example of creating a new message of this type and populating its fields.

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

[1]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).

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 Descriptor (*message type*) from a `.proto` file. A small number of message types are imported when the package is first loaded, including the `tutorial.Person` type we saw in the last section. All other types must be imported from `.proto` files 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 (**?**). Instead of being associated with a static hash table, this environment dynamically queries the in-memory database of loaded descriptors during normal variable lookup. This allows new descriptors to be parsed from `.proto` files and added to the global namespace.[2]

Creating, accessing, and modifying messages.

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

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 message, 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.

Printing, reading, and writing Messages

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

[2]Note that there is a significant performance overhead with this **RObjectTables** implementation. Because the table is on the search path and is not cacheable, lookups of symbols that are behind it in the search path cannot be added to the global object cache, and R must perform an expensive lookup through all of the attached environments and the Protocol Buffer definitions to find common symbols (most notably those in base) from the global environment. Fortunately, proper use of namespaces and package imports reduces the impact of this for code in packages.

```
R> p
```

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

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"
```

A primary benefit of Protocol Buffers is an 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 or arbitrary binary connections:

```
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
```

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, or a raw vector of serialized bytes.

4. Under the hood: S4 classes and methods

The **RProtoBuf** package uses the S4 system to store information about descriptors and messages. 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. S4 objects are immutable, and so the methods that modify field values of a message return a new copy of the object with R's usual functional copy on modify semantics[3]. Using the S4 system allows the package to dispatch methods that are not generic in the S3 sense, such as `new` and `serialize`.

The **Rcpp** package (**??**) 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 (**?**), which provide a more concise way of wrapping C++ functions and classes in a single entity.

Since **RProtoBuf** users are most often switching between two or more different languages as part of a larger data analysis pipeline, both generic function and message passing OO style calling conventions are supported:

- The functional dispatch mechanism of the the form `method(object, arguments)` (common to R).
- The message passing object-oriented notation of the form `object$method(arguments)`.

Additionally, **RProtoBuf** supports tab completion for all classes. Completion possibilities include 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 (**?**).

Table 2 lists the six primary message and descriptor classes in **RProtoBuf**. The package documentation provides a complete description of the slots and methods for each class.

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 '`FieldDescriptor`'s for the individual fields of the message.

```
R> new(tutorial.Person)

message of type 'tutorial.Person' with 0 fields set
```

[3] **RProtoBuf** was designed and implemented before Reference Classes were introduced to offer a new class system with mutable objects. If **RProtoBuf** were implemented today Reference Classes would almost certainly be a better design choice than S4 classes.

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.

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 methods.

When **RProtoBuf** is first loaded it calls `readProtoFiles` which reads in the example file `addressbook.proto` 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[4].

Field descriptors

```
R> tutorial.Person$email
descriptor for field 'email' of type 'tutorial.Person'
R> tutorial.Person$email$is_required()
[1] FALSE
R> tutorial.Person$email$type()
[1] 9
R> tutorial.Person$email$as.character()
[1] "optional string email = 3;\n"
R> class(tutorial.Person$email)
[1] "FieldDescriptor"
attr(,"package")
[1] "RProtoBuf"
```

Enum and EnumValue descriptors

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

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

operator can be used to retrieve the value of enum constants contained in the EnumDescriptor, or to invoke methods.

```
R> tutorial.Person$PhoneType

descriptor for enum 'PhoneType' with 3 values

R> tutorial.Person$PhoneType$WORK

[1] 2

R> class(tutorial.Person$PhoneType)

[1] "EnumDescriptor"
attr(,"package")
[1] "RProtoBuf"

R> tutorial.Person$PhoneType$value(1)

enum value descriptor tutorial.Person.MOBILE

R> tutorial.Person$PhoneType$value(name="HOME")

enum value descriptor tutorial.Person.HOME

R> tutorial.Person$PhoneType$value(number=1)

enum value descriptor tutorial.Person.HOME

R> class(tutorial.Person$PhoneType$value(1))

[1] "EnumValueDescriptor"
attr(,"package")
[1] "RProtoBuf"
```

File descriptors

The class 'FileDescriptor' represents file descriptors in R. The $ operator can be used to retrieve named fields defined in the 'FileDescriptor', or to invoke methods.

```
R> f <- tutorial.Person$fileDescriptor()
R> f

file descriptor for package tutorial \
    (/usr/local/lib/R/site-library/RProtoBuf/proto/addressbook.proto)

R> f$Person

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.

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 3: Correspondence between field type and R type retrieved by the extractors. R lacks native 64-bit integers, so the `RProtoBuf.int64AsString` option is available to return large integers as characters to avoid losing precision; see Section 5.3 below. All but the `Message` type can be represented in vectors of one or more elements; for the latter a list is used.

Table 3 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.

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** (**?**) or **bit64** (**?**) 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 (**?**). 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 natively 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 storage types `function`, `language`, and `environment` are specific to R and have no equivalent native type in Protocol Buffers. These three types are supported by first serializing with `base::serialize` in R and then stored in a raw bytes field.

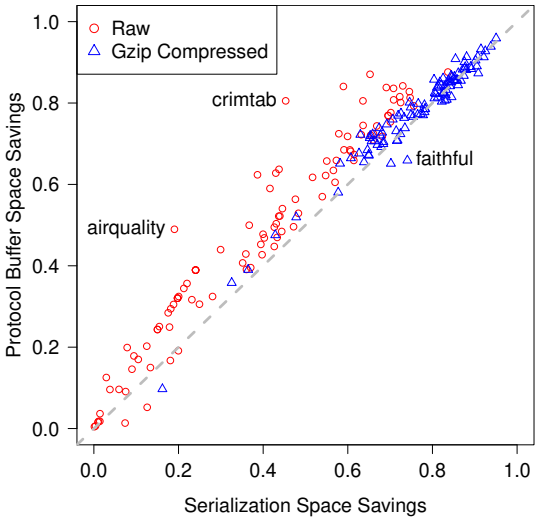
6.1. Evaluation: Serializing R data sets

This section evaluates the effectiveness of serializing arbitrary R data structures into Protocol Buffers. We use the 104 standard data sets included in the **datasets** package included with R as our evaluation data. These data sets include data frames, matrices, time series, tables, lists, and some more exotic data classes. For each data set, we compare how many bytes are used to store the data set using four different methods:

- normal R serialization (**?**),
- R serialization followed by gzip,
- normal Protocol Buffer serialization, and
- Protocol Buffer serialization followed by gzip.

Figure 2 shows the space savings $\left(1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}\right)$ for each of the data sets using each of these four methods. The associated table shows the exact data sizes for some outliers and the aggregate of all 104 data sets. Note that Protocol Buffer serialization results in slightly smaller byte streams compared to native R serialization in most cases (red dots), but this difference disappears if the results are compressed with gzip (blue triangles).

The `crimtab` dataset of anthropometry measurements of British prisoners (**?**) and the `airquality` dataset of air quality measurements in New York show the greatest difference in the space savings when using Protocol Buffers compared to R native serialization. The



Data set	`object.size`	R serialization		**RProtoBuf** serialization	
		default	gzipped	default	gzipped
`crimtab`	7,936	4,641 (41.5%)	714 (91.0%)	1,655 (79.1%)	576 (92.7%)
`airquality`	5,496	4,551 (17.2%)	1,242 (77.4%)	2,874 (47.7%)	1,294 (76.4%)
`faithful`	5,136	4,543 (11.5%)	1,339 (73.9%)	4,936 (3.9%)	1,776 (65.4%)
All	609,024	463,833 (24%)	139,814 (77%)	436,746 (28%)	142,783 (77%)

Figure 2: (Top) Relative space savings of Protocol Buffers and native R serialization over the raw object sizes of each of the 104 data sets in the **datasets** package. Points to the left of the dashed $y = x$ line represent datasets that are more efficiently encoded with Protocol Buffers. (Bottom) Absolute space savings of three outlier datasets and the aggregate performance of all datasets. R version 3.3.1 was used for both the figure and the table.

`crimtab` dataset is a 42×22 table of integers, most equal to 0, and the `airquality` dataset is a data frame of 154 observations of 1 numeric and 5 integer variables. In both data sets, the large number of small integer values can be very efficiently encoded by the *Varint* integer encoding scheme used by Protocol Buffers which use a variable number of bytes for each value.

The other extreme is represented by the `faithful` dataset of waiting time and eruptions of the Old Faithful geyser in Yellowstone National Park, Wyoming, USA (**?**). This dataset is a data frame with 272 observations of 2 numeric variables. The R native serialization of repeated numeric values is more space-efficient, resulting in a slightly smaller object size compared to the serialized Protocol Buffer equivalent.

This evaluation shows that the `rexp.proto` 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

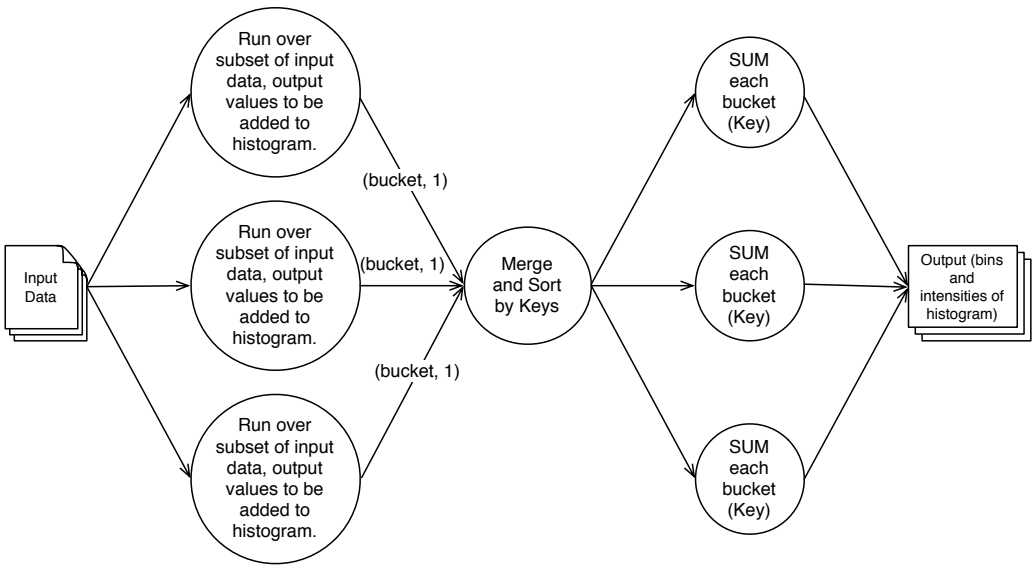


Figure 3: Diagram of MapReduce histogram generation pattern.

Protocol Buffers are used extensively at Google for almost all RPC protocols, and to store structured information on a variety of persistent storage systems (**?**). Since the initial release in 2010, hundreds of Google's statisticians and software engineers use the **RProtoBuf** package on a daily basis to interact with these systems from within R. The current section illustrates the power of Protocol Buffers to collect and manage large structured data in one language before analyzing it in R. Our example uses MapReduce (**?**), which has emerged in the last decade as a popular design pattern to facilitate parallel processing of big data using distributed computing clusters.

Big data sets in fields such as particle physics and information processing are often stored in binned (histogram) form in order to reduce storage requirements (**?**). Because analysis over such large data sets may involve very rare phenomenon or deal with highly skewed data sets or inflexible raw data storage systems, unbiased sampling is often not feasible. In these situations, MapReduce and binning may be combined as a pre-processing step for a wide range of statistical and scientific analyses (**?**).

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 3, each mapper rounds a data point to a bucket width and outputs that bucket as a key and '1' as a value. Reducers count how many times each key occurs and outputs a histogram 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 (**?**) 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 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.

7.1. A simple single-machine example for Python to R serialization

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()
```

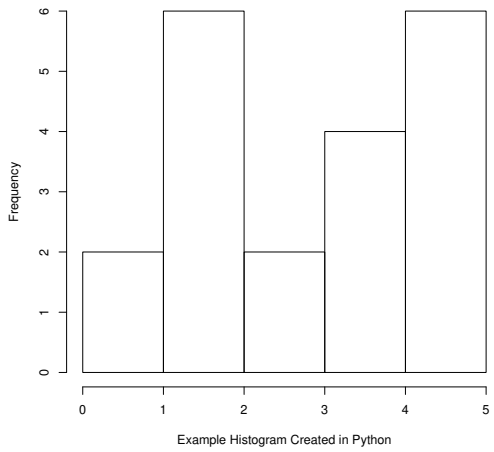
The Protocol Buffer created from this Python script can then be read into R and converted to a native R histogram object for plotting. The code below first attaches the **HistogramTools** package which imports **RProtoBuf**. Then reads all of the `.proto` descriptor definitions provided by **HistogramTools** and adds them to the environment as described in Section 3. Next the serialized Protocol Buffer is parsed using the `HistogramTools.HistogramState` schema. Finally the Protocol Buffer representation of the histogram is converted to a native R histogram object with `as.histogram` and passes the result to `plot` (see Figure **??**).

```
R> library("HistogramTools")
R> readProtoFiles(package="HistogramTools")
R> hist <- HistogramTools.HistogramState$read("/tmp/hist.pb")
```

```
R> hist

[1] "message of type 'HistogramTools.HistogramState' with 3 fields set"

R> plot(as.histogram(hist), main="")
```



This simple example uses a constant histogram generated in Python to illustrate the serialization concepts without requiring the reader to be familiar with the interface of any particular MapReduce implementation. In practice, using Protocol Buffers to pass histograms between another programming language and R would provide a much greater benefit in a distributed context. For example, a first-class data type to represent histograms would prevent individual histograms from being split up and would allow the use of combiners on Map workers to process large data sets more efficiently than simply passing around lists of counts and buckets.

One of the authors has used this design pattern with C++ MapReduces over very large data sets to write out histogram protocol buffers for several large-scale studies of distributed storage systems (**??**).

8. Application: Data interchange in web services

The previous section described an application where data from a program written in another language was saved to persistent storage and then read into R for further analysis. This section describes another common use case where Protocol Buffers are used as the interchange format for client-server communication. 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 this problem

by providing a cross-platform method for serializing arbitrary structures into well defined messages, which can then be exchanged using any protocol.

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 (**?**). 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 execute 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 `rexp.proto` descriptor and the `serialize_pb` and `unserialize_pb` functions described in Section 6 to convert between R objects and Protocol Buffer messages.

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://demo.ocpu.io/MASS/data/Animals/pb
```

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

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 an R data set encoded as a Protocol Buffer message from OpenCPU. In R, we use the HTTP client from the `httr` package (**?**). 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://demo.ocpu.io/MASS/data/Animals/pb')
R> output <- unserialize_pb(req$content)
R> identical(output, MASS::Animals)
```

Similarly, to retrieve the same data set in a Python client, 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. We use the HTTP client from the `urllib2` module in our example to retrieve the encoded Protocol Buffer from the remote server then parse and print it from Python.

```
import urllib2
from rexp_pb2 import REXP
req = urllib2.Request('https://demo.ocpu.io/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 previous example used a simple `HTTP GET` method to retrieve an R object from a remote service (**OpenCPU**) encoded as a Protocol Buffer. In many cases simple `HTTP GET` methods are insufficient, and a more complete RPC system may need to create compact Protocol Buffers for each request to send to the remote server in addition to parsing the response Protocol Buffers.

The **OpenCPU** framework allows us to do arbitrary R function calls from within any programming language by encoding the arguments in the request Protocol Buffer. 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 Protocol Buffer messages. 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://cloud.opencpu.org/ocpu/library/stats/R/rnorm/pb",
+    body = payload,
+    add_headers ("Content-Type" = "application/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 its 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 five 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 to 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 and Tim Hesterberg were 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. Feedback from two anonymous referees greatly improved both the presentation of this paper and the package contents.

A. The `rexp.proto` schema descriptor

Below a print of the `rexp.proto` schema (originally designed by **?**) 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;
    LANGUAGE = 8;
    ENVIRONMENT = 9;
    FUNCTION = 10;
  }
  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;
  optional bytes languageValue = 13;
  optional bytes environmentValue = 14;
  optional bytes functionValue = 15;
}
message STRING {
  optional string strval = 1;
  optional bool isNA = 2 [default=false];
}
message CMPLX {
  optional double real = 1 [default=0];
  required double imag = 2;
}
```

Affiliation:

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

Murray Stokely
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA, United States of America
E-mail: murray@stokely.org
URL: http://www.stokely.org/

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