RProtoBuf: Protocol Buffers for R

Romain François¹  Dirk Eddelbuettel²

¹R Enthusiasts
²Debian Project

useR! 2010
National Institute of Standards and Technology (NIST)
Gaithersburg, Maryland, USA
Outline

1. Protocol Buffers
2. RProtoBuf
3. Summary / Outlook
Outline

1. Protocol Buffers
   - Overview
   - Example

2. RProtoBuf

3. Summary / Outlook
Google's Protocol Buffers are a flexible, efficient, automated mechanism for serializing structured data—think XML, but smaller, faster, and simpler.

Users define the data structures in a proto file, and then use special generated source code. Code is forwards- and backwards-compatible to proto changes.

This permits to easily write and read structured data to and from a variety of data streams, and using a variety of officially supported languages—Java, C++, or Python.

Or one can use third-party implementations for languages such as C#, Perl, Ruby, Haskell, and now R via the RProtoBuf package.
Protocol Buffers compare favourably against a number of competing data / messaging formats.

Source: http://www.slideshare.net/kevinweil/protocol-buffers-and-hadoop-at-twitter
package tutorial;
message Person {
  required string name = 1;
  required int32 id = 2; // Unique ID number for person.
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0; HOME = 1; WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}

message AddressBook {
  repeated Person person = 1;
}
#include "addressbook.pb.h"

using namespace std;

// Iterates through all people in the AddressBook
// and prints info about them.

void ListPeople(const tutorial::AddressBook& address_book) {
  for (int i=0; i < address_book.person_size(); i++) {
    const tutorial::Person& person = address_book.person(i);

    cout << "Person ID: " << person.id() << endl;
    cout << " Name: " << person.name() << endl;
    if (person.has_email()) {
      cout << " E-mail address: "
           << person.email() << endl;
    }
  }
}
for (int j = 0; j < person.phone_size(); j++) {
  const tutorial::Person::PhoneNumber &phone_number = person.phone(j);

  switch (phone_number.type()) {
    case tutorial::Person::MOBILE:
      cout << " Mobile phone #: ";
      break;
    case tutorial::Person::HOME:
      cout << " Home phone #: ";
      break;
    case tutorial::Person::WORK:
      cout << " Work phone #: ";
      break;
  }
  cout << phone_number.number() << endl;
}
}
Outline

1. Protocol Buffers
2. RProtoBuf
   - Overview
   - Examples
     - Addressbook
     - (Stylized) High-Frequency Financial Data
   - Writer
   - R Readers
3. Summary / Outlook
The **RProtoBuf** package implements R bindings to the C++ protobuf library from Google. **RProtoBuf** uses features of the protocol buffer library to support creation, manipulation, parsing and serialization of protocol buffers messages.

Taking advantage of facilities in the **Rcpp** package, **RProtoBuf** uses S4 classes and external pointers to expose objects that look and feel like standard R lists, yet are managed by the underlying C++ library.

These objects also conform to the language-agnostic definition of the message type allowing access to their content from other supported languages.
Addressbook example from R

See `demo(addressbook)`

```r
> # load the package
> require( RProtoBuf )
> # read the proto file
> readProtoFiles( files="addressbook.proto" )
> # create a prototype with a call to new
> # on the descriptor for the Person type,
> romain <- new( tutorial.Person )
> # then update the message
> romain <- update( romain ,
+    email = "romain@r-enthusiasts.com",
+    id = 1, name = "Romain Francois",
+    phone = new( tutorial.Person.PhoneNumber,
+      number = "+33(0)...", type = "MOBILE")
)```

Romain François and Dirk Eddelbuettel

RProtoBuf: Protocol Buffers for R @ useR! 2010
> # directly supply parameters to the ctor
> dirk <- new( tutorial.Person,
+   email = "edd@debian.org",
+   id = 2, name = "Dirk Eddelbuettel" )
> # update the phone repeated field with list
> dirk$phone <- list(
+   new( tutorial.Person.PhoneNumber,
+     number = "+01...", type = "MOBILE" ),
+   new( tutorial.Person.PhoneNumber,
+     number = "+01...", type = "HOME" ) )
```r
> # build the address book
> book <- new(tutorial.AddressBook,
+    person = list(romain, dirk))
> # debug content - this is not wire content
> writeLines(as.character(book))
> # the serialized message,
> # see also the io demo
> serialize(book, NULL)
```
Example `proto` file for Financial Data

```protobuf
text

// Namespace
package TradeData;

// A simple Fill, ie a completed trade
message Fill {
    required double timestamp = 1;
    required string symbol = 2;
    required double price = 3;
    required int32 size = 4;
}

// A sequence of Fills
message Trades {
    repeated Fill fill = 1;
}

text

See inst/examples/HighFrequencyFinance/ in the RProtoBuf package.
```
int main(int argc, char **argv) {
    const char* pbfile = "trades.pb";
    const int N = 1000;
    set_seed(123, 456);
    double tstamp = 1277973000;    // 2010-07-01 08:30:00
    double tprice = 100.0;        // gotta start somewhere
    char sym[] = "ABC";
    TradeData::Trades tr;
    for (int i=0; i<N; i++) {
        TradeData::Fill *fill = tr.add_fill();
        tstamp += runif(0.000, 0.100);
        tprice += round(rt(5) * 0.01 * 100)/100;
        int tsize = 100 + round(runif(0,9))*100;
        fill->set_timestamp(tstamp);
        fill->set_price(tprice);
        fill->set_symbol(sym);
        fill->set_size(tsize);
    }
    std::fstream output(pbfile, std::ios::out | std::ios::binary);
    if (!tr.SerializeToOstream(&output)) {
        std::cerr << "Failed to write data." << std::endl;
        return -1;
    }
    return 0;
}
We could add this to the proto file:

```protobuf
define exchType {
    NYSE = 0; NASDAQ = 1; ARCS = 2; BATS = 3;
}
optional exchType exchange = 5 [default = NYSE];
```

If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible [...]:

- you must not change the tag [...] of any existing fields.
- you must not add or delete any required fields.
- you may delete optional or repeated fields.
- you may add new optional or repeated fields but you must use fresh tag numbers [...]

See [http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html](http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html)
> basicUse <- function(verboset=TRUE) {
+   readProtoFiles("TradeData.proto")
+   x <- read(TradeData.Trades, "trades.pb")
+   xl <- as.list(x)
+   df <- do.call(rbind,
+                 lapply(as.list(xl$fill),
+                        function(.)
+                        as.data.frame(as.list(.))))
+   df[,1] <- as.POSIXct(df[,1], origin="1970-01-01")
+   if (verboset) print(summary(df))
+   invisible(df)
+ }

See inst/examples/HighFrequencyFinance/loadInR.r
Example R reader: Smarter

```r
> betterUse <- function(verbosetrue,
+ file="trades.pb") {
+ readProtoFiles("TradeData.proto")
+ x <- read( TradeData.Trades, "trades.pb"
+ xl <- lapply( x$fill, as.list )
+ df <- data.frame(timestamp= as.POSIXct(sapply(xl,
+ "[[", "timestamp" ),
+ origin="1970-01-01" ),
+ symbol = sapply(xl,"[[","symbol"),
+ price = sapply(xl,"[[","price"),
+ size = sapply(xl,"[[","size"))
+ if (verbose) print(summary(df))
+ invisible(df)
+ }
```

See inst/examples/HighFrequencyFinance/loadInR.r
> compiled <- function(verbose=FALSE, 
+                        file="trades.pb") {
+    stopifnot(file.exists(file))
+    df <- .Call("pbload", file);
+    if (verbose) print(summary(df))
+    invisible(df)
+ }

See inst/examples/HighFrequencyFinance/loadInR.r
extern "C" SEXP pbload(SEXP b) {
    std::string pbfile = Rcpp::as<std::string>(b);
    TradeData::Trades tr;
    std::fstream fs(pbfile.c_str(), std::ios::in | std::ios::binary);
    if (!tr.ParseFromIstream(&fs)) {
        std::cerr << "Trouble parsing..." << std::cout;
        return R_NilValue;
    }
    int n = tr.fill_size();
    Rcpp::DatetimeVector timestamp(n);
    Rcpp::CharacterVector tsym(n);
    Rcpp::NumericVector tprice(n);
    Rcpp::IntegerVector tsize(n);
    for (int i=0; i<n; i++) {
        const TradeData::Fill &fill = tr.fill(i);
        timestamp[i] = fill.timestamp();
        tsym[i] = fill.symbol();
        tprice[i] = fill.price();
        tsize[i] = fill.size();
    }
    return Rcpp::DataFrame::create(Rcpp::Named("times") = timestamp,
                                    Rcpp::Named("symbol") = tsym,
                                    Rcpp::Named("price") = tprice,
                                    Rcpp::Named("size") = tsize);
}

See inst/examples/HighFrequencyFinance/protoModule.cpp
Running the script `loadInR.r` from the aforementioned examples directory in the `RProtoBuf` package:

<table>
<thead>
<tr>
<th>Version</th>
<th>Elapsed Time</th>
<th>Relative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiled</td>
<td>0.006</td>
<td>1.0000</td>
</tr>
<tr>
<td>betterUse</td>
<td>0.138</td>
<td>23.0000</td>
</tr>
<tr>
<td>basicUse</td>
<td>4.606</td>
<td>767.6667</td>
</tr>
</tbody>
</table>

Times are total in seconds from on three replications each on relatively recent server, using the `rbenchmark` package.
Summary

- We have introduced the Google *Protocol Buffers* library as a means to generating efficient data interfacing code: fast, auto-generated and extensible.
- We illustrated its use via our nascent *RProtoBuf* package.
- *RProtoBuf* brings autogenerated accessors to *R*—which may however not be the fastest access.
- The *Rcpp* package makes it easy to *manually* add Protocol Buffers capabilities to our *R* analyses.
Goal: Use the new *Modules* feature in *Rcpp* to get at (almost) auto-generated yet very efficient (*C++*-based) access from *R*.

Second Goal: Add networking capabilities, maybe via *R*’s built-in http server.