Google

# Integrating R with the Go programming language using interprocess communication

Christoph Best, Karl Millar, Google Inc.

*chbest@google.com*

# Statistical software in practice & production

- Production environments !!!= R development environment
  - Scale: machines, people, tools, lines of code…
- “discipline of software engineering”
  - Maintainable code, common standards and processes
  - Central problem: The programming language to use
- *How do you integrate statistical software in production?*
  - Rewrite everything in your canonical language?
  - Patch things together with scripts, dedicated servers, ... ?

***Everybody should just write Java!***

# Programming language diversity

- Programming language diversity is hard …
  - Friction, maintenance, tooling, bugs, …
- … but sometimes you need to have it
  - Many statistics problems can “only” be solved in R*
- How do you integrate R code with production code?
  - without breaking production

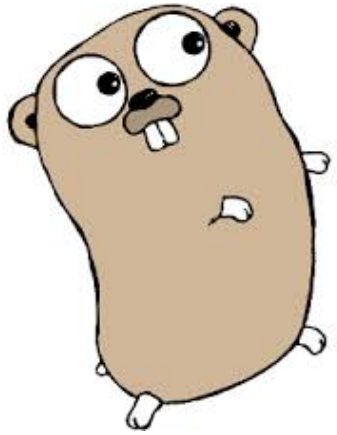
*though my colleagues keep pointing out that any Turing-complete language can solve any problem

# The Go programming language

- Open-source language, developed by small team at Google
- Aims to put the fun back in (systems) programming
- Fast compilation and development cycle, little “baggage”

- Made to feel like C (before C++)
- Made not to feel like Java or C++ (enterprise languages)

- Growing user base (inside and outside Google)



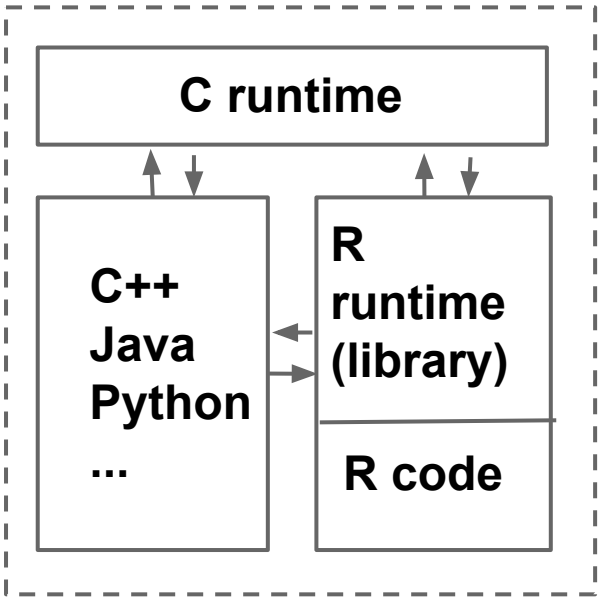
# Integration: Intra-process vs inter-process

- Intra-process: Link different languages through C ABI
  - smallest common denominator
  - issues: stability, ABI evolution, memory management, threads, …

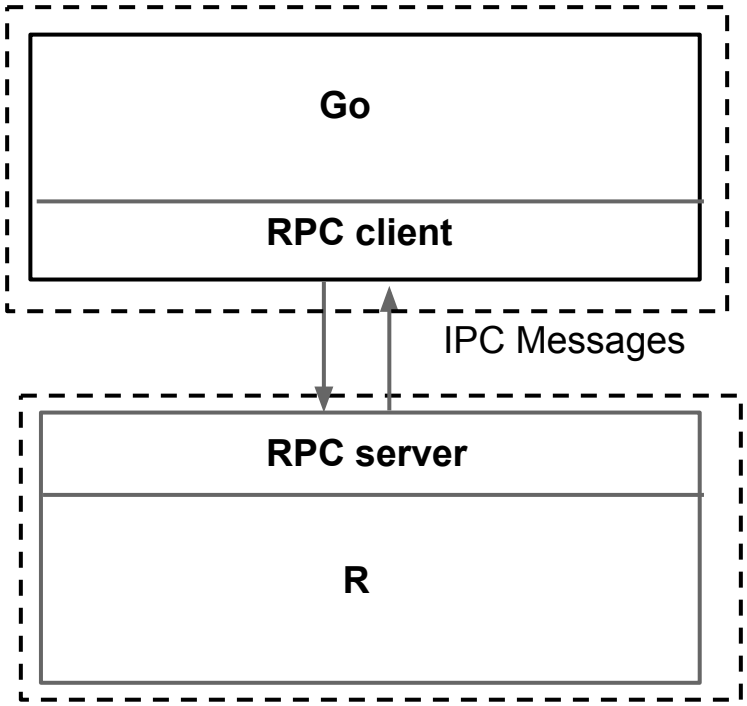
Can we do better? Or at least differently?

- Idea: Sick of crashes? Execute R in a separate process
  - Runs alongside main process, closely integrated: “lamprey”
- Provide communication layer between R and host process
  - A well-defined compact interface surface

# Integration: Intra-process vs inter-process




single process
shared memory
shared crashes




two processes
memory isolation

# How it works

- Host process starts R subprocess
  - Tightly coupled on same machine/container

"Lamprey"

- R subprocess loads required packages
- R executes `executionservice::RunExecutionService()`
  - listens for connections, executes incoming requests, returns results
  - leverages existing RPC package

- Communication layer: gRPC(-like) / Protocol buffers
  - All messages are proto buffers
  - R subprocess is server, host language process is client

# Data model

*READ-EVALUATE-PRINT LOOP*

- Host sees R subprocess as REPL
  Sends R commands and R values, reads results
  - Only R values, no references handled on this level

- R values encoded as proto buffers on wire
- Only basic R types go on the wire:
  - vectors of elementary data types
  - lists
  - everything else must be expressed by basic types

# Four simple requests from Go to R

- `CreateContext()` returns `Context`:
  - create an execution context (isolation)
- `Set(ctx, variableName, Rvalue)`
  - Assign a value to a named variable
- `Do(ctx, Rexpression)` returns `RValue`
  - Evaluate an expression (a string) in R
  - Expression refers to previously set variabkes
  - Return result value
- `CloseContext(ctx)`:
  - free resources in context (e.g. variables)

# Wire representation for R values

```
message REXP {
  required RClass    rclass = 1;

  repeated double    realValue = 2 [packed=true];
  repeated sint32    intValue = 3 [packed=true];
  repeated RBOOLEAN  booleanValue = 4;
  repeated STRING    stringValue = 5;
  repeated REXP      rexpValue = 8;

  repeated string    attrName = 11;
  repeated REXP      attrValue = 12;
}
```

STRING, INTEGER, REAL, LOGICAL, NULLTYPE, LIST

basic R vectors
list of R values
only one present

Object attributes

from RProtoBuf package,
Originally written by Saptarshi Guha for RHIPE (http://www.rhipe.org)

# Wire representation for R values

```
enum RBOOLEAN {
     F=0;
     T=1;
     NA=2;
}
```

Boolean is an enum with *three* values

String contains a flag to indicate NA value

```
message STRING {
  optional string strval = 1;
  optional bool   isNA = 2 [default=false];
}
```

# Set request: wire representation

```
message SetRequest {
  optional Context  context = 1;
  optional string   variable_name = 2;
  optional rexp.REXP value = 3;
}

message SetResponse {
}
```

Context in which to assign the variable

Variable name to assign to

Value in wire encoding

No response necessary
Error conditions are transmitted separately

# Evaluate request: wire representation

```
message EvaluateRequest {
  optional Context context = 1;

  repeated string  expression = 2;

  optional bool    return_result = 3 [default=true];
}

message EvaluateResponse {
  optional rexp.REXP result = 1;
}
```

Context in which to assign the variable

R expression as string
Can refer to variables

Indicates whether a result is expected

Result value in wire representation

# A quick example

```
service, err := rexp.NewService(context.Background())
x := []float64{1, 2, 3}
y := []float64{2, 4, 6}

r, err := service.Do(
    rexp.Set("x", x),
    rexp.Set("y", y),
    "d <- data.frame(x=x, y=y)",
    "m <- lm(x ~ y, d)",
    "list(coef=m$coefficients, res=m$residuals)")

coefficients := r.Get("coef").ToAny().([]float64)
residuals := r.Get("res").ToAny().([]float64)
```

Set up input data

Execute R code (magically sets up context etc.)

Transfer input data to R process

Make input data into a data frame

Do statistics here

Prepare results

Extract results

# Strategies

- Problem: You can only transfer “basic” R values
- Solution: Construct higher types explicitly (e.g. data frames)
  - In the future, we can hide this complexity using improvements to the Go libraries

- Problem: Only values can be transferred, no references
- Solution: You can keep references as variables on the R side
  - Go library code can allocate variable names, etc, automate a lot of things

This library only provides the “bottom layer”.

# Does it work?

- Yes.
  - Used in several experimental projects.
  - Statisticians/analysts able to deal with interface.

- Is it fast enough?
  - Yes, for reasonably sized datasets (10-100 MBytes)
  - About 3ms for CreateContext/Set/Evaluate/CloseContext sequence
  - About 50-100 MByte/s for transferring data
  - Speed more dominated by R runtime than wire protocol

# Future work

- Better data types on the Go side
  - Data frames natively in Go
  - Automatic construction of `data.frame` in R
- Callbacks and inverted server
  - Callbacks: Allow R to make calls to Go
  - Inverted server: Run Go as a subprocess of R
  - Could be used to extend R with Go code
- Open sourcing

# Summary

- Inter-process communication is a (surprisingly) effective way to couple two programming languages

- Simplicity
- Robustness
- Clarity