The way to grpc-elixir

tony612

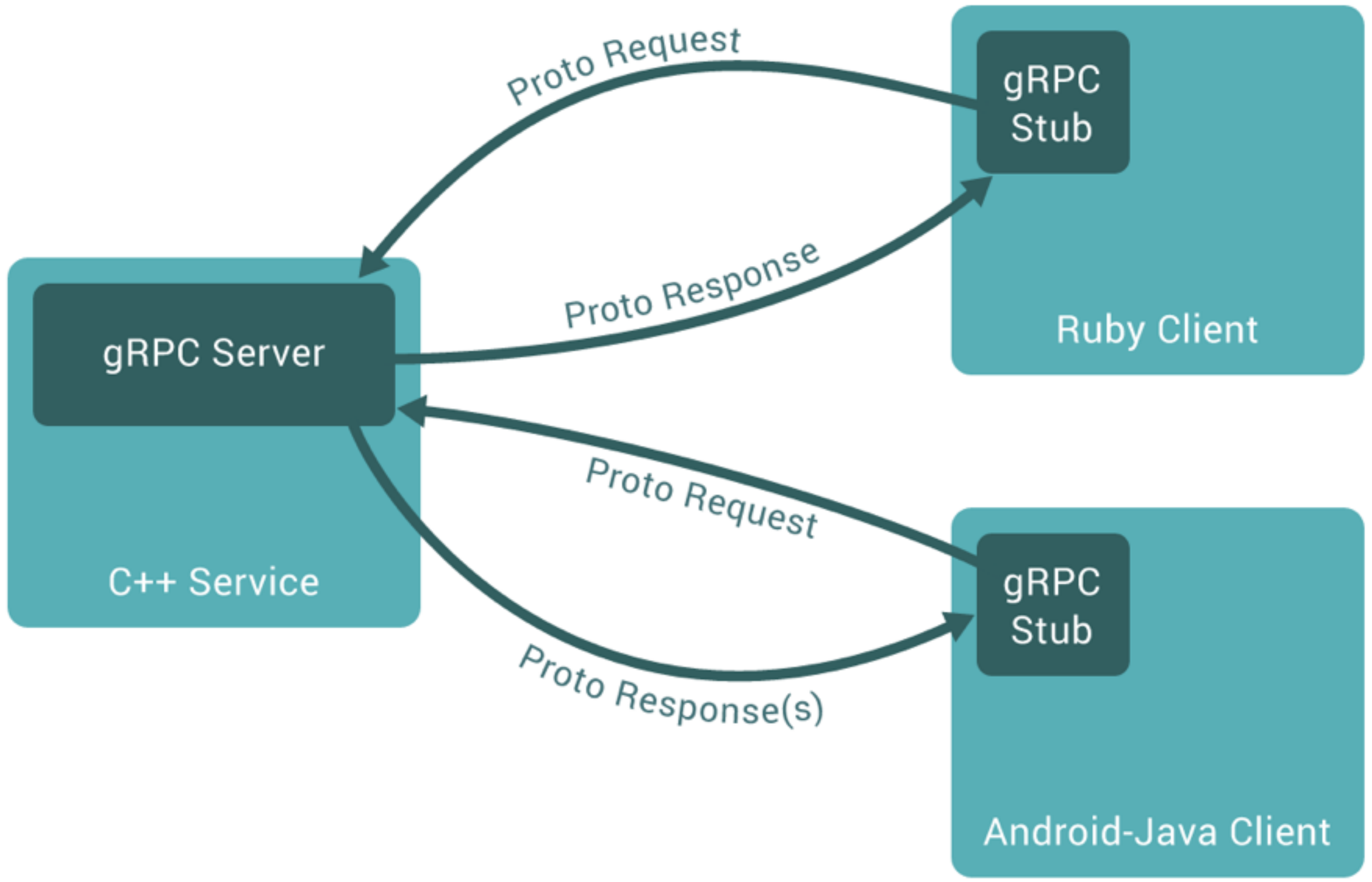
Overview

- Introduction to gRPC
- Implementation of gRPC in Elixir
    - Macro
    - Erlang NIF
    - Erlang ports
    - Pure Elixir
- The future of grpc-elixir

What’s gRPC?

http://www.grpc.io/

- A RPC framework by Google
- g is for gRPC(1.0), good(1.1) — grpc/grpc/pull/7912
- Based on HTTP/2 and (Google's) Protobuf(current supported format)
- Across languages and platforms
- Used by Google for a long time(underlying technologies and concepts), Square, Netflix, Docker and so on
- Used by Liulishuo from 2015.8.7 (0.6.0)



Officially Supported Platforms: http://www.grpc.io/about/#osp

Helloworld example

by gRPC ruby

// helloworld.proto written by us

```
syntax = "proto3";

package helloworld;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

```
$ grpc_tools_ruby_protoc helloworld.proto
```

// helloworld_pb.rb generated

```ruby
Google::Protobuf::DescriptorPool.generated_pool.build do
  add_message "helloworld.HelloRequest" do
    optional :name, :string, 1
  end
  add_message "helloworld.HelloReply" do
    optional :message, :string, 1
  end
end

module Helloworld
  HelloRequest = Google::Protobuf::DescriptorPool.
    generated_pool.lookup("helloworld.HelloRequest").msgclass
  HelloReply = Google::Protobuf::DescriptorPool.
    generated_pool.lookup("helloworld.HelloReply").msgclass
end
```

// helloworld_services_pb.rb generated

```
module Helloworld
  module Greeter
    class Service
      include GRPC::GenericService

      self.marshal_class_method = :encode
      self.unmarshal_class_method = :decode
      self.service_name = 'helloworld.Greeter'

      rpc :SayHello, HelloRequest, HelloReply
    end

    Stub = Service.rpc_stub_class
  end
end
```

// greeter_server.rb by us

```ruby
class GreeterServer < Helloworld::Greeter::Service
  def say_hello(hello_req, _unused_call)
    Helloworld::HelloReply.new(
      message: "Hello #{hello_req.name}")
  end
end

s = GRPC::RpcServer.new
s.add_http2_port('0.0.0.0:8080', :this_port_is_insecure)
s.handle(GreeterServer)
s.run_till_terminated
```

// greeter_client.rb by us

```ruby
stub = Helloworld::Greeter::Stub.new(
          'localhost:50051', :this_channel_is_insecure)
request = Helloworld::HelloRequest.new(name: 'world')
message = stub.say_hello(request).message
p message # => 'Hello world'
```

How to implement an Elixir gRPC?

Interface design

Proto in Elixir (bitwalker/exprotobuf)

```
defmodule Helloworld do
  @external_resource Path.expand(
    "../../priv/protos/helloworld.proto", __DIR__)
  use Protobuf, from: Path.expand(
    "../../priv/protos/helloworld.proto", __DIR__)
end
```

Creates Helloworld.HelloRequest & Helloworld.HelloReply

Service definition

```
defmodule Helloworld.Greeter.Service do
  use GRPC.Service, name: "helloworld.Greeter",
                    marshal_function: :encode,
                    unmarshal_function: :decode
  alias Helloworld.{HelloRequest, HelloReply}

  rpc :SayHello, HelloRequest, HelloReply  # macro
end
```

Client definition

```elixir
defmodule Helloworld.Greeter.Stub do
  use GRPC.Stub, service: Helloworld.Greeter.Service
end # creates say_hello method here using macro

channel = GRPC.Channel.connect(
            "localhost:50051", insecure: true)
request = Helloworld.HelloRequest.new(name: "grpc-elixir")
channel |> Helloworld.Greeter.Stub.say_hello(request)
```

Why macro?

- Service definition is simple, consistent with Proto
- API will be simple(for users), RPC-like
- Code generator will be easier to implement

Code example

tony612/grpc-elixir

- lib/grpc/service.ex
- lib/grpc/stub.ex

What about underlying implement?

Implement of `GRPC.Call.unary(channel, path, message, opts)`

Implements of other languages

GitHub repo	grpc/grpc (c based)	grpc/grpc-go	grpc/grpc-java
languages	c(core lib), C++, Ruby, NodeJS, Python, PHP, C#, Objective-C	go	java

Which one?

GitHub repo	xxx/grpc-elixir	xxx/grpc-elixir
languages	Interoperability http://erlang.org/doc/tutorial/introduction.html	pure Elixir

Compare(first impression)

GitHub repo	Interoperability	pure Elixir
Difficult for implement	Medium	Hard
Risk	Erlang interoperability	gRPC detail
Workload	Light(?)	Heavy(?)

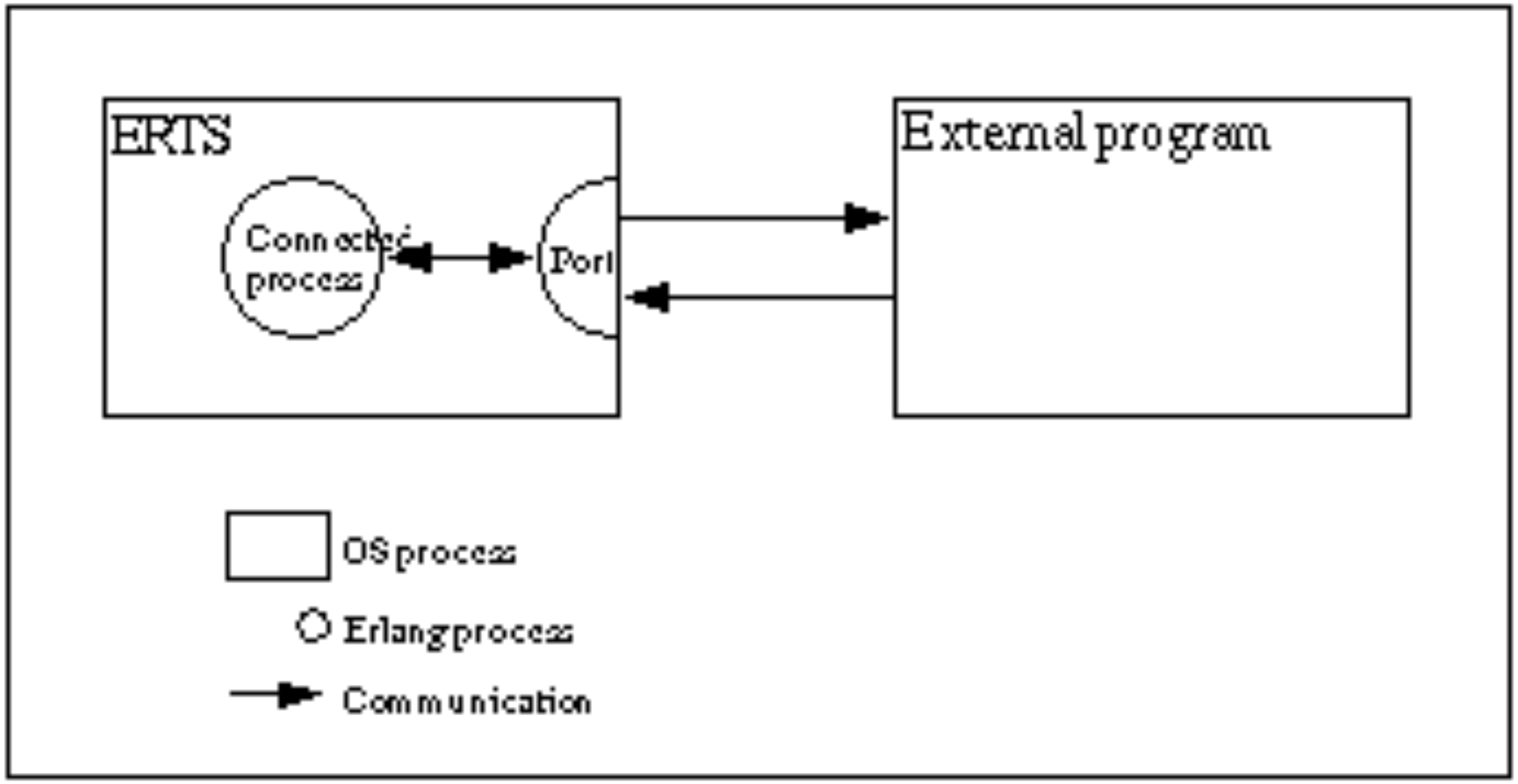
I tried interoperability first

(I’m lazy)

Solutions in Erlang

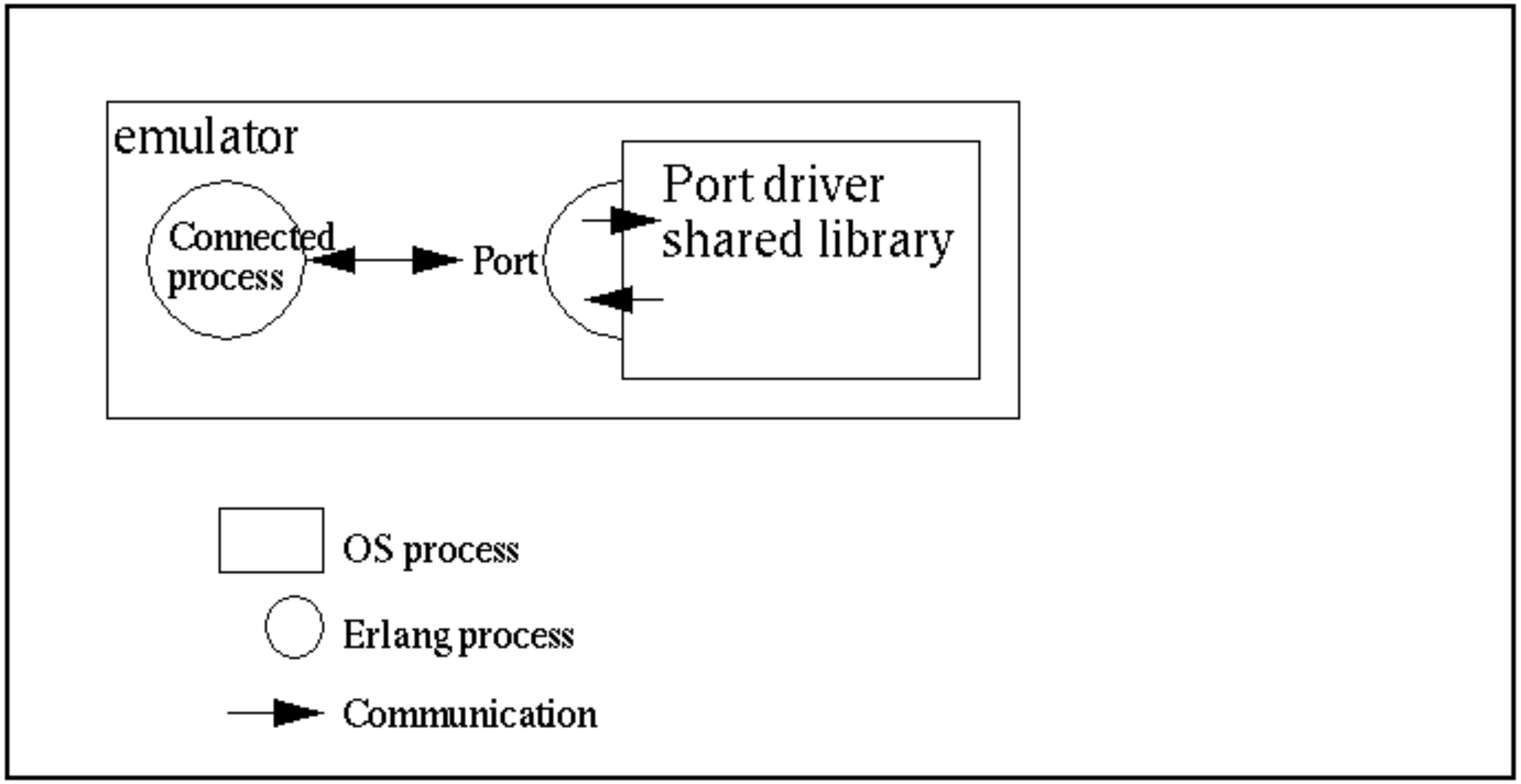
- Ports
- Port Drivers
- C Nodes
- NIFs

Ports



External program can written by any language in theory

Port Drivers



Similar to Ports, but external program is dynamically linked in ERST

C Node

- Similar to Ports
- Run in a Erlang Node
- Interaction is like talking to an Erlang node

NIFs

- Native Implemented Functions
- A NIF is a function in C/Java
- The NIFs of a module are compiled and linked into a dynamic library (SO in UNIX, DLL in Windows)
- A little like C extension in Ruby

	Ports	Port Drivers	C Nodes	NIFs
Difficulty	Medium	Hard	Medium	Medium
Speed	A little slow	Fast	A little slow	Fast
Safety	Safe	Dangerous*	Safe	Dangerous*

* Erlang runtime will crash if extension crashes

I tried NIFs and Port

- NIF: grpc-elixir/tree/8e05b5
- Port: Not finished

Pitfalls of NIFs

- C code is difficult to write :(
- Safety is really a big problem
- Much work needed for communication with gRPC C core lib
- Weird bugs(Segment fault, bus error..)
- A NIF function should be returned in 1ms (blocks scheduler)
    - Otherwise, miscellaneous strange problems are caused
    - Erlang provides solutions for this problem, but not good enough

Pitfalls of Ports

- Only C lib can be used for this project (golang and others are too high level)
- C code is difficult to write :(
- Communication between Erlang and C code is a difficult (via binary)
  - ei(Erlang Interface C lib) is hard to use
  - erl_eterm seems deprecated and doesn't support map
  - JSON or other format?
  - It's strange to call gRPC C code (Multiple function calls in a call)
- Not fast

So I tried to implement with pure Elixir

It’s much easier than I thought! 🚀

(for the moment)

Ideas

- It's just HTTP/2 with Protobuf
- gRPC has doc for HTTP/2 format: http://www.grpc.io/docs/guides/wire.html
- joedevivo/chatterbox for HTTP/2 client
- cowboy2 for HTTP/2 server (chatterbox has server, but I prefer cowboy)

Recap

- gRPC is great and worth using
- Interoperability may not be a good choice for your project
- Elixir is really very powerful (with Macro, pattern match, binary handling...)

Future of grpc-elixir

- ✔ Basic implement of client and server (unary)
- Support for some options (timeout, compress)
- Stream calls support for client and server
- Auth
- Code generator from proto files
- (?) Extract the underlying logic to a Erlang project for grpc-erlang

Bonus: some details of NIFs

- Official Demo: http://erlang.org/doc/tutorial/nif.html
- Official Doc: http://erlang.org/doc/man/erl_nif.html
- A better API doc: http://devdocs.io/erlang~19/erts-8.0/doc/html/erl_nif

NIFs in real world(Elixir) - 1

- Add C repo in deps

```
{:xxx, github: “xxx", app: false, compile: false}
```

- Add a Makefile in root path for compiling C lib and your C code
    - Remember to handle the case when your lib is used as a dep
- Use elixir-lang/elixir_make(will be merged to Elixir soon)
    - make will be run when running mix compile

details: grpc-elixir/tree/8e05b5

NIFs in real world(Elixir) - 2

- Data to C code should be handled with enif_get_* functions
- Resource should be passed to Elixir instead of C struct
  - Can't be used in Elixir, but is just passed back to C
  - You can define a wrapper if a struct(or part) is not declared in header
- You need to decide pass binary or char list to C as char list

details: grpc-elixir/tree/8e05b5

Q&A