Piqi-RPC

Exposing Erlang services via JSON, XML and Google Protocol Buffers over HTTP

Anton Lavrik

http://piqi.org

http://www.alertlogic.com

Overview

- Call Erlang functions using HTTP “POST”:
  - Server = Erlang
  - Client = anything else that talks over HTTP, e.g. CURL, JavaScript, PHP, ...
    - No state is maintained between the calls
- Input and output parameters can be represented in JSON, XML or Protocol Buffers
- There is Piqi-RPC command-line client -- interprets command-line arguments as input parameters

Inspiration: solving a real problem

- Make it easy to connect Erlang services with other programming languages and environments:
  - internal and public APIs
  - O&M and development tools (Erlang shell and rpc:call are not enough)
- If you have to do it manually:
  - manual parsing of XML, JSON -- not an easy task in Erlang
  - adding command-line interface, other formats make it practically impossible to manage

Inspiration: the Piqi project

- Most of parts were already implemented as a part of the Piqi project:
  - type-based converter between JSON, XML, Protocol Buffers
  - data serialization system for Erlang compatible with Protocol Buffers
- “All I had to do is to put these things together and communication layer on top of HTTP”
  - in reality, turned out to be a lot of work
  - and a lot of fun!

Layer I: Piqi serialization & data conversion

What is type-based data serialization

1. Describe your data using a high-level *data definition language (Piqi)*

2. Use *"Interface" compiler (piqic)* to generate serialization and deserialization functions for the target programming language

3. Use the generated functions to encode and decode program data to/from serialization format

   Examples:

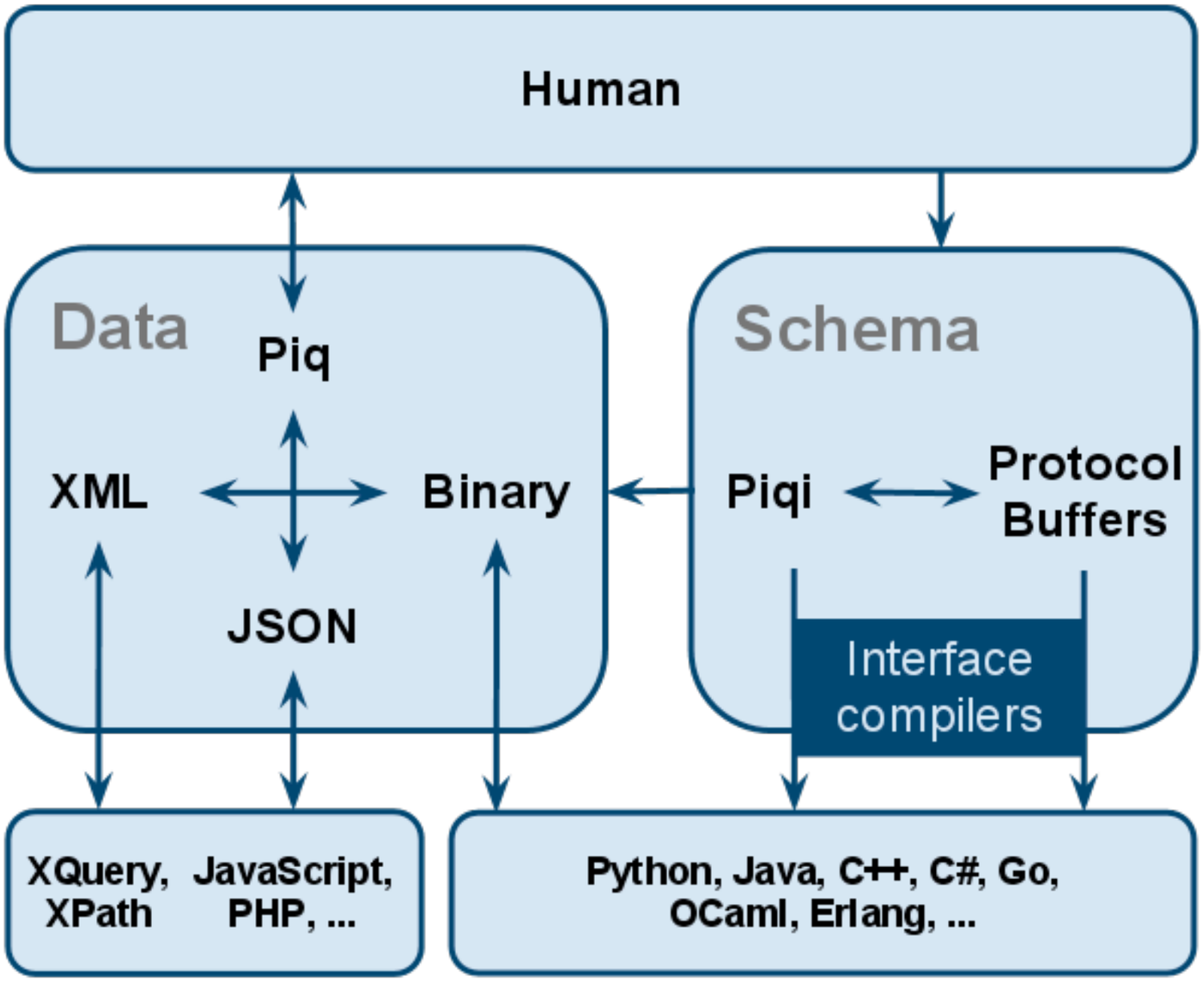
- Cross-language serialization systems: Protocol Buffers, ASN.1

- Many middleware and RPC systems: Apache Thrift, CORBA, UBF, ...

Demo #1: using Piqi for JSON, XML and Protobuf serialization and deserialization in Erlang program

examples/erlang/io_json_xml_pb.erl

The Piqi project



Piqi - data definition language

- primitive types: int, bool, float, string, binary, verbatim text
- user-defined types: record, variant (aka tagged unions), enum, list, alias
    - no tuples, atoms, function closures, polymorphism
- includes and imports
- type extensions
- functions
- the Piqi language itself is defined in the self-specification

Example: person.piqi - Piqi type definition module

```
% definition of a record type
.record [
    .name person        % record name
    .field [
        .name name      % field name
        .type string    % field type
    ]
    .field [
        .name id
        .type int
    ]
    .field [
        .name email
        .type string
        .optional       % field is optional
    ]
    .field [
        .name phone
        .type phone-number
        .repeated       % field can be repeated 0 or more times
    ]
]

% (continued on the next slide...)
```

```
% (example continued)

.record [
    .name phone-number
    .field [
        .name number
        .type string
    ]
    .field [
        .name type
        .type phone-type
        .optional
        .default.home  % default value for an optional field
    ]
]

% definition of an enum type
.enum [
    .name phone-type
    .option [ .name mobile ]
    .option [ .name home ]
    .option [ .name work ]
]
```

Example: person.piqi in “Piqi-light” syntax

```
type person =
  {
    - name :: string()
    - id :: int()
    ? email :: string()
    * phone :: phone-number()
  }
```

```
type phone-number =
  {
    - number :: string()
    ? type :: phone-type() = .home
  }

type phone-type =
    | mobile
    | home
    | work
```

generated by “piqi light person.piqi” command

Example: person.piqi definitions mapped to Erlang

```
-record(person, {
    name :: string() | binary(),
    id :: integer(),
    email :: string() | binary(),
    phone = [] :: [phone_number()]
}).
-record(phone_number, {
    number :: string() | binary(),
    type :: phone_type()
}).
-type(phone_type() ::
      mobile
    | home
    | work
).

-type(person() :: #person{}).
-type(phone_number() :: #phone_number{}).
```

<mod>_piqi.hrl generated by “piqic erang person.piqi” command

Example: person.piqi.proto (Protobuf definition)

```
message person {
    required string name = 1;
    required sint32 id = 2;
    optional string email = 3;
    repeated phone_number phone = 4;
}

message phone_number {
    required string number = 1;
    optional phone_type type = 2 [default = home];
}

enum phone_type {
    mobile = 1;
    home = 2;
    work = 3;
}
```

generated by “piqi to-proto person.piqi” command

Piq - data representation language

Syntax is designed to be convenient for viewing and editing data:

- lighter than JSON: no quotes around field names, no “,” separators
- less parenthesis than in S-expressions
- kind-of like Erlang proplists, but no need to wrap variants in tuples
- comments
- verbatim text literals
- repeated fields, like in Protocol Buffers and XML

Example: person.piq (corresponds to the previously defined “person” data type)

```
:person/person [
    .name "J. Random Hacker"
    .id 0
    .email "j.r.hacker@example.com"
    .phone [
        .number "(111) 123 45 67"
        % NOTE: phone type is "home" by default
    ]
    .phone [
        .number "(222) 123 45 67"
        .mobile
    ]
    .phone [
        .number "(333) 123 45 67"
        .work
    ]
]
```

Example: person.piq.json

```
{
  "name": "J. Random Hacker",
  "id": 0,
  "email": "j.r.hacker@example.com",
  "phone": [
    { "number": "(111) 123 45 67", "type": "home" },
    { "number": "(222) 123 45 67", "type": "mobile" },
    { "number": "(333) 123 45 67", "type": "work" }
  ]
}
```

generated by “piqi convert -t json person.piq” command

Example: person.piq.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <name>J. Random Hacker</name>
  <id>0</id>
  <email>j.r.hacker@example.com</email>
  <phone>
    <number>(111) 123 45 67</number>
    <type>home</type>
  </phone>
  <phone>
    <number>(222) 123 45 67</number>
    <type>mobile</type>
  </phone>
  <phone>
    <number>(333) 123 45 67</number>
    <type>work</type>
  </phone>
</person>
```

generated by “piqi convert -t xml person.piq” command

XML and JSON mapping details

- JSON & XML:
  - UTF-8 only
  - Floating point “NaN”, “Infinity” and “-Infinity” are represented as strings
  - Base64 for binaries
- XML (in addition to the above):
  - No support for DTD, Schemas, namespaces and attributes
  - Whitespace is significant in text nodes

Example: piqi getopt

```
piqi getopt -t (json|xml|piq) --piqtype person/person -- \
    --name "J. Random Hacker" \
    --id 0 \
    --email "j.r.hacker@example.com" \
    --phone [ --number "(111) 123 45 67" ] \
    --phone [ \
        --number "(222) 123 45 67" \
        --mobile \
    ] \
    --phone [ \
        --number "(333) 123 45 67" \
        --work \
    ]
```

Example: piqi getopt (short names)

```
piqi getopt -t (json|xml|piq) --piqtype person/person -- \
    "J. Random Hacker" \
    -i 0 \
    -e "j.r.hacker@example.com" \
    -p [ "(111) 123 45 67" ] \
    -p [ \
        "(222) 123 45 67" \
        -m \
    ] \
    -p [ \
        "(333) 123 45 67" \
        -w \
    ]
```

Layer II: Piqi-RPC & HTTP

Data serialization workflow (from a previous slide)

1. Describe your data using in *Piqi*

2. Use *“Interface” compiler (piqic erlang)* to generate serialization and deserialization functions for Erlang

3. Use the generated functions to encode and decode program data to/from serialization format

Piqi-RPC workflow

1. Describe your data **and functions** in *Piqi*

2. Use *"Interface" compiler (~~piqic~~ **piqic-erlang-rpc**)* to generate serialization and deserialization functions **and server stubs** for Erlang.

3. ~~Use the generated functions to encode and decode program data to/from serialization format~~

3. **Implement server callback functions**

4. **Run piqi_rpc:start(), piqi_rpc:add_service(..., "<URL path>")**

   Your services are up and running at http://.../<URL path>/<func-name>

HTTP layer

- Built using Webmachine -- great library for writing “well-behaving” HTTP servers
- “POST” for making calls with input parameters in the body
- Use any of these media types in “Content-Type”, “Accept” HTTP headers: application/json, application/xml", "application/x-protobuf”
- Successful calls return 200 “OK or 204 “No Content”
- Unsuccessful call return ~10 different meaningful HTTP status codes along with error description, e.g. 400 “Bad Request”, 404 “Not Found”, 415 “Unsupported media type”, 411 “Length required”, 500 “Internal Server Error”, 503 “Service Unavailable”, etc.

Piqi function definition example: addressbook.piqi

```
.include [ .module person ]

.function [
    .name add-person
    .input person
    .error string
]

.function [
    .name get-person
    .input [
        .field [
            .name id
            .type int
        ]
    ]
    .output person
    .error string
]
```

Generated Erlang types and signatures

```
% from addressbook_piqi.hrl:

-type(add_person_input() :: person()).

-record(get_person_input, {
    id :: integer()
}).
-type(get_person_output() :: person()).
-type(get_person_error() :: string() | binary()).
-type(get_person_input() :: #get_person_input{}).

% from addressbook_piqi_impl.hrl:

-spec add_person/1 :: (add_person_input()) -> ok.

-spec get_person/1 :: (get_person_input()) ->
      {ok, get_person_output()}
    | {error, get_person_error()}.
```

.hrl files generated by “piqic-erang-rpc addressbook.piqi”

Example: piqi call - Piqi-RPC command-line client

```
$ piqi call -t json http://localhost:8888/addressbook/add-person -- \
    --name "J. Random Hacker" \
    --id 0 \
    --email "j.r.hacker@example.com" \
    --phone [ --number "(111) 123 45 67" ] \
    --phone [ \
        --number "(222) 123 45 67" \
        --mobile \
    ] \

$ piqi call -t json http://localhost:8888/addressbook/get-person -- 0
{
  "name": "J. Random Hacker",
  "id": 0,
  "email": "j.r.hacker@example.com",
  "phone": [
    { "number": "(111) 123 45 67", "type": "home" },
    { "number": "(222) 123 45 67", "type": "mobile" },
    { "number": "(333) 123 45 67", "type": "work" }
  ]
}
```

Example: piqi call (getting command-line help)

```
$ piqi call http://localhost:8888/addressbook -h

Piqi-RPC functions (use -p flag for more details):

  add-person -- <person>, which is a combination of:

    --name <string>
    --id <int>
    --email <string> (optional)
    --phone <phone-number> (repeated)

  get-person -- <input>, which is:

    --id <int>
```

Demo #2: simple “address book” Piqi-RPC service

piqi-rpc/examples/addressbook/

Implementation details: OCaml part

- ~95% OCaml, ~5% Erlang
- OCaml part: two standalone natively compiled executables
  - piqi -- Piq and Piqi language implementations and tools including type-based data converter
  - piqic -- Piqi compiler & basic Erlang compiler backend that generates type definitions and Protocol Buffers codecs (piqic erlang)
- OCaml type-based converter communicates with Erlang using Erlang port interface over Unix pipe (they actually use Piqi-RPC/Protobuf!)

Implementation details: Erlang part

- “piqi” Erlang application
  - Runtime support library for data serialization (all formats)
  - piqi_tool.erl -- wrapper of OCaml “piqi server” Erlang port
- “piqi_rpc” Erlang application
  - Piqi compiler backend for stubs generation (piqic-erlang-rpc)
  - Runtime support library for generated stubs
  - Webmachine behavior implementing HTTP layer

Initial Erlang serialization benchmarks

- Input data: “addressbook.piq” (238 bytes in Protobuf format)
- Serialization / deserialization rate is in objects per second per CPU core on 2.4 GHz Core 2 Duo
- No performance optimizations has been done so far

	Read rate	Write rate
Protobuf	25000	50000
JSON	4500	4500 (pretty-printed)
XML	3000	5500

Project details

- 100% open-source licensed under Apache v2, code is on GitHub
- Easy to build and deploy non-Erlang self-contained dependencies
- Erlang part can be customized without hacking into OCaml
- OCaml part is fairly stable
- Tested on Linux, Mac OS X, Solaris, 32- and 64-bit
- Documentation, examples, tests

Future plans

- Piqi:
  - embedded documentation in Piqi specifications
  - performance & multi-core scalability
  - coordinated upgrades for generated Piqi Erlang code
- Piqi-RPC:
  - tweaks, usability improvements, etc
  - asynchronous messages

Demo #3: querying erlang:process_info() using Piqi-RPC

piqi-rpc/examples/process_info/

Why OCaml?

- Static type checker
- Mutability: mostly for record fields and global variables
- Raw speed, native code, self-contained executables
- Great libraries for language and tool development: utf8 lexing, parsing, pretty-printing, XML, JSON
- Sequential, single-threaded: no need for concurrency, fault-tolerance, etc.

Thank you!

Homepage: http://piqi.org/doc/piqi-rpc

Source code: http://github/alavrik/piqi

@alavrik