Seaplus: Streamlining a safe execution of C/C++ code from Erlang

Organisation: Copyright (C) 2018-2019 Olivier Boudeville

Contact: about (dash) seaplus (at) esperide (dot) com

Creation date: Sunday, December 23, 2018

Lastly updated: Wednesday, March 6, 2019

Dedication: Users and maintainers of the `Seaplus` bridge, version 1.0.

Abstract: The role of the `Seaplus` bridge is to control C or C++ code from Erlang, not as NIF but thanks to a port, and to streamline the corresponding integration process.

The latest version of this documentation is to be found at the official Seaplus web-site (`http://seaplus.esperide.org`).

The documentation is also mirrored here.

Table of Contents

Important Note

Seaplus is still **work in progress** - it is functional, yet no code generation yet!

Overview

A typical use-case is **having a C or C++ library of interest that we would like be able to use from Erlang**, whereas, for any reason (availability of sources, complexity, size, performance or interest), recoding it (in Erlang) is not desirable.

However tempting it may be to integrate tightly C/C++ code to the Erlang VM (typically through a NIF), one may prefer trading maximum performances for safety, and run that C/C++ code (which is often at last partly foreign, hence possibly unreliable) into a separate, isolated (operating system) process.

Then the integrated code will not be able to crash the Erlang application, and for example any memory leak it would induce would only affect its own process (that, morevoer, depending on the use case, may be safely restarted) - not the application one.

Indeed, taking into account the Erlang Interoperability Tutorial, the following approaches are the most commonly considered ones when having to make C/C++ code available from Erlang:

- raw **ports** and **linked-in drivers**: they are mostly obsolete for the task at hand (superseded by better counterparts)
- `os:cmd/1`: a rudimentary solution that offers little control and requires much syntactic parsing effort
- custom **socket-based protocol**: quite low-level and complicated
- `NIF`: as mentioned, they may jeopardise the VM (depending on the use case, this may be acceptable or not)
- C-Node and Erl_Interface: this is the combination that we preferred for Seaplus, and that we tried to streamline/automate here, at least partially

In a nutshell, this approach consists on spawning a "fake" Erlang node written in C (the `C-Node`) and using the standard *Erlang external term format* in order to communicate with it (relying for that on the `Erl_Interface` facility). Doing so allows a seamless communication to happen, despite language heterogeneity.

C-Node and Erl_Interface help a lot, yet, as shown in this reference example, quite a lot of boiler-plate/bridging code (home-made encoding and conventions) remains needed.

The **goal of Seaplus is to reduce that interfacing effort**, thanks to a set of generic, transverse functions on either side (modules in Erlang, a library in C/C++) and the use of metaprogramming (i.e. the Seaplus parse transform) in order to generate at least a part of the code needed in both sides, while leaving to the developer enough leeway so that he can define precisely the mapping interface that he prefers (ex: with regards to naming, types introduced and used, management of resource ownership, etc.).

`Ceylan-Seaplus` relies on various facilities offered by the Ceylan-Myriad toolbox.

Usage

So we would have here a (possibly third-party) service (typically a library, directly usable from C, offering a set of functions) that we want to integrate, i.e. to make available from Erlang.

Let's suppose that said service is named `Foobar`, and that the functions it provides (hence on the C side) are declared as (typically in some `foobar.h` header file[1], referring to a possibly opaque `foobar.so` library):

```
#include <stdbool.h>

struct foo_data { int count; float value } ;

enum foo_status {low_speed,moderate_speed,full_speed};
enum tur_status {tur_value,non_tur_value};

int foo(int a);
struct foo_data * bar(double a, enum foo_status status);
enum tur_status baz(unsigned int u, const char * m);
bool tur();
char * frob(enum tur_status);
```

With the definition of this example, we ensured to reproduce real-life situations, like atoms vs enums, dynamic memory allocation (for the returned struct) and runtime failures (since calling `foo(0)` is to trigger a division by zero).

What would be the corresponding ideal Erlang interface to make such a fantastic service available?

First of all, multiple corresponding Erlang APIs can be considered, and some design choices have to be made (we can foresee that some are more elegant/convenient than others, and that a perfect, universal, one-size-fit-all automated mapping does not seem so achievable).

An easy step is to decide, at least in most cases, to map each of these C functions to an Erlang counterpart function that, unsurprisingly, bears the same name and most of the time has the same arity, and to have them gathered into a single module that would be best named `foobar` (and thus shall be defined in `foobar.erl`).

We believe that, in order to rely on a convenient Erlang-side API for this service, adaptations have to be made (ex: with regard to typing), and thus that it should preferably be defined in an ad-hoc manner (i.e. it should be tailor-made, rather than be automatically generated through a mapping possibly suffering from impedance mismatch).

So such a service-specific API shall be devised by the service integrator (i.e. the developer in charge of the integration of the C/C++ code to Erlang). But how?

At the very least, what will be offered on the Erlang side by our `foobar` module shall be somehow specified. A very appropriate way of doing so is to list the type specifications of the targeted counterpart functions meant to be ultimately available (defined and exported) from Erlang, like in[2]:

```
-module(foobar).

-include("seaplus.hrl").
```

[1] See the full, unedited version of the foobar.h test header that is actually used.

```
-record(foo_data, {count :: integer(), value :: float()}).
-type foo_data() :: #foo_data{}.

-type foo_status() :: 'low_speed'|'moderate_speed'|'full_speed'.
-type tur_status() :: 'tur_value'|'non_tur_value'.

-spec foo(integer()) -> integer().
-spec bar(float(), foo_status()) -> foo_data().
-spec baz(integer(), text_utils:ustring()) -> tur_status().
-spec tur() -> bool().
-spec frob(tur_status()) -> text_utils:ustring().
```

The Seaplus include allows notably to mark this `foobar` module as a service stub.

Comments (description, usage, examples) are also expected to be joined to these specs, they are omitted in this documentation for brevity.

Other facility functions that all integrated services will need, and whose signature (if not implementation) would be the same from a service to another (ex: to start/stop this service from Erlang), will also certainly be needed. However listing these facility functions in our `foobar` module would offer little interest (as they are the same for all integrated services), so these extra functions are to remain implicit here[3].

These service-level built-in functions automatically defined by Seaplus of user interest are, notably:

- `start/0`: starts said service, a `{driver_crashed,ErrorReason}` exception being thrown should the driver or the integrated library crash (ex: SEGV)
- `start_link/0`: starts and links said service to the user process, expected to crash in turn should the driver or the integrated library crash
- `restart/0`: restarts the service, typically after it was started with `start/0`, failed and threw an exception
- `stop/0`: stops the service

Of course such a module, as it was defined above (i.e. just a set of function specifications), is useless and would not even compile as such. But the Seaplus parse transform will automatically enrich and transform it so that, once the C part (the driver) will be available, the `Foobar` service will become fully usable from Erlang, with no extra boilerplate code to be added by the Erlang integrator.

More precisely, for each of the function type specification, a corresponding bridging implementation will be generated and added (unless the `foobar` module already includes one, so that the user can selectively override the Seaplus code generation), whilst all the needed facility functions will be included as well.

Here is a corresponding (mostly meaningless) usage example[4] of this `foobar` module, when executed from any given process (ex: a test one):

[2]See the full, unedited version of the foobar.erl API module that is actually used, together with its foobar.hrl header file.

[3]Note though that, at least for some services, specific initialisation/tear-down functions may exist in the vanilla, C version of that service. In that case, they should be added among said function specifications (preferably named for example `init/teardown` or alike, in order to distinguish from the Seaplus-reserved `start/stop` primitives), so that they are available from Erlang as well.

```
foobar:start(),
MyFooData = foobar:bar(3.14,full_speed),
NewCount = foobar:foo(MyFooData#foo_data.count),
Res = case foobar:tur() of
  true ->
    foobar:baz(NewCount,"Hello");
  false ->
    non_tur_value
end,
io:format("Having: ~s~n",[foobar:frob(Res)]),
foobar:stop().
```

At this point, one may think that, thanks to these function specs, the full counterpart C bridging code might have been automagically generated, in the same movement as the Erlang bridging code? Unfortunately, not exactly! At least, not yet; maybe some day (if ever possible and tractable). Currently: only *parts* of it are generated.

Indeed C-side elements will have been produced by the Seaplus parse-transform (notably the function selector include, used to map functions on either sides), but the conversion (thanks to `Erl_Interface`) from the Erlang terms received by the port into arguments that will feed the C functions and on the other way round (i.e. from the C results to the Erlang terms that shall be sent back) is still left to the service integrator.

This work remains, yet it is also a chance to better adapt the bridging code to the interfacing contract one would like to be fulfilled, for example with regard to resource ownership. Indeed, should the C part take pointers as arguments, shall it delete them once having used them? Conversely, should a C function return a pointer to a dynamically allocated memory, who is responsible for the eventual deallocation of it?

To address these questions, service-specific choices and conventions have to be applied, and this information cannot be found or deduced from the C/C++ pre-existing code generically by an algorithm (including the Seaplus one). As a result, we believe that in all cases some effort remains to be done by the service integrator.

So: we saw that nothing special had to be done on the Erlang side (the `foobar.erl` stub will suffice), and that the C side deserved some love to be complete; what kind of extra work is needed then?

Seaplus generated an header file, `foobar_seaplus_api_mapping.h` (see here for an *example* of it), in charge of telling that C side about the actual encoding of the service functions across the bridge. In our example this generated header would contain:

```
#define FOO_1_ID  1
#define BAR_2_ID  2
#define BAZ_2_ID  3
#define TUR_0_ID  4
#define FROB_1_ID 5
```

This indicates that for example the `baz/2` Erlang function, as hinted by its type specification in `foobar.erl`, has been associated by Seaplus to the `BAZ_2_ID`

[4] See the full, unedited version of the foobar_test.erl module used to test the Erlang-integrated service (emulating an actual use of that service).

(namely, of course: `${FUNCTION_NAME}_${ARITY}_ID`) identifier (whose value happens to be `3` here[5]).

The C part of the bridge, typically defined by the service integrated in `foobar_seaplus_driver.c`[6], is thus to include that `foobar_seaplus_api_mapping.h` generated header in order to map the Erlang function identifier in a call request to its processing.

Seaplus offers moreover various helpers to facilitate the writing of this C driver; they are gathered in the Seaplus library (typically `libseaplus.so`) and available by including the Seaplus C header file, `seaplus.h` (see here).

Based on these elements, the actual bridging code can be written, like in the following shortened version. The `FOO_1_ID` case is among the simplest possible call, while the `BAR_2_ID` one is more complex; for both calls no memory leak is involved (see the full source of this test driver, notably for the conversion helpers used for `bar/2`):

```
[...]
int main()
{

  // Provided by the Seaplus library:
  byte * buffer = start_seaplus_driver();

  while (read_command(buffer) > 0)
  {

    fun_id current_fun_id;
    arity param_count;
    ETERM ** parameters = NULL;

    ETERM * call_term = get_function_information(buffer,
      &current_fun_id, &param_count, &parameters);

    // Now, taking care of the corresponding function call:
    switch(current_fun_id)
    {

      case FOO_1_ID:
        // -spec foo(integer()) -> integer() vs int foo(int a)
        check_arity_is(1, param_count, FOO_1_ID);

        /*
         * So we expect the (single, hence first) parameter to
         * be an integer:
         */
        int foo_a_param = get_parameter_as_int(1, parameters);

        // Actual call:
        int foo_result = foo(foo_a_param);
```

[5] Of course no code should rely on that actual value, which could change from a generation to another, or as the API is updated; only the `BAZ_2_ID` identifier shall be trusted by user code.

[6] See the full, unedited version of the foobar_seaplus_driver.c driver, i.e. the core of the service-specific, C-side integration.

```
            // Sending of the result:
            write_as_int(buffer, foo_result);

            break;

          case BAR_2_ID:

            /* -spec bar(float(), foo_status()) -> foo_data() vs
             * struct foo * bar(double a, enum foo_status status)
             */
            check_arity_is(2, param_count, BAR_2_ID);

            // Getting first the Erlang float:
            double bar_double_param = get_parameter_as_double(1, parameters);

            // Then the atom for foo_status():
            char * atom_name = get_parameter_as_atom(2, parameters);

            // Converting said atom for the C API:
            enum foo_status bar_status_param =
                get_foo_status_from_atom(atom_name);

            // Actual call:
            struct foo_data * struct_res = bar(bar_double_param,
                                               bar_status_param);

            // Converting this result into a relevant term:
            ETERM * foo_data_res =
                      get_foo_data_record_from_struct(struct_res);

            // Sending of the result record:
            write_term(buffer, foo_data_res);

            break;

        [...]

        default:
            raise_error("Unknown function identifier: %u", current_fun_id);

        }

        clean_up_command(call_term,parameters);

      }

      stop_seaplus_driver(buffer);

    }
```

Wrapping Up

We believe that, in order to make a pre-existing C/C++ library available to Erlang while not going the NIF route (typically when not wanting to jeopardise the Erlang VM for that), Seaplus offers a good option in terms of safety, low overhead and simplicity.

The overall integration process is quite streamlined, and we tried to reduce as much as possible the size and complexity of the service-specific integration code that remains needed.

For example one may contrast the few Foobar-specific files (foobar.hrl, foobar.erl and foobar_seaplus_driver.c, i.e. the ones that shall be written by the service integrator), and the ones implementing the Seaplus generic support (namely seaplus.hrl, seaplus.erl, seaplus.h, seaplus.c and seaplus_parse_transform.erl).

Licence

Seaplus is licensed by its author (Olivier Boudeville) under a disjunctive tri-license giving you the choice of one of the three following sets of free software/open source licensing terms:

- Mozilla Public License (MPL), version 1.1 or later (very close to the former Erlang Public License, except aspects regarding Ericsson and/or the Swedish law)
- GNU General Public License (GPL), version 3.0 or later
- GNU Lesser General Public License (LGPL), version 3.0 or later

This allows the use of the Seaplus code in as wide a variety of software projects as possible, while still maintaining copyleft on this code.

Being triple-licensed means that someone (the licensee) who modifies and/or distributes it can choose which of the available sets of licence terms he is operating under.

We hope that enhancements will be back-contributed (ex: thanks to merge requests), so that everyone will be able to benefit from them.

Current Stable Version & Download

Using Stable Release Archive

Currently no source archive is specifically distributed, please refer to the following section.

Using Cutting-Edge GIT

We try to ensure that the main line (in the `master` branch) always stays functional. Evolutions are to take place in feature branches.

This integration layer, `Ceylan-Seaplus`, relies (only) on:

- Erlang, version 21.0 or higher
- a suitable C/C++ compiler, typically gcc
- the Ceylan-Myriad base layer

We prefer using GNU/Linux, sticking to the latest stable release of Erlang, and building it from sources, thanks to GNU `make`.

For that we devised the install-erlang.sh script; a simple use of it is:

```
$ ./install-erlang.sh --doc-install --generate-plt
```

One may execute `./install-erlang.sh --help` for more details about how to configure it, notably in order to enable all modules of interest (`crypto`, `wx`, etc.) even if they are optional in the context of Seaplus.

As a result, once proper Erlang and C environments are available, the Ceylan-Myriad repository should be cloned and built, before doing the same with the Ceylan-Seaplus repository, like in:

```
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Myriad
$ cd Ceylan-Myriad && make all && cd ..
$ git clone https://github.com/Olivier-Boudeville/Ceylan-Seaplus
$ cd Ceylan-Seaplus && make all
```

One can then test the whole with:

```
$ cd tests/c-test
$ make integration-test
```

Miscellaneous Technical Points

Seaplus Log System

When integrating a C service, the most difficult part is ensuring the sanity of the C driver, i.e. knowing what happens within it whenever converting terms back and forth, handling pointers, allocating memory, crashing unexpectedly, etc. (a.k.a. the joys of C programming).

To facilitate troubleshooting, Seaplus provides a log system, allowing to trace the various operations done by the driver (including the user code and the Seaplus facilities that it relies on).

This log system is enabled by default. To disable it (then no runtime penalty will be incurred), set `SEAPLUS_ENABLE_LOG` to `0` (ex: add the `-DSEAPLUS_ENABLE_LOG=0` option when compiling the library, see GNUmakevars.inc for the various build settings).

So running a Seaplus-integrated service, with log system enabled, should produce a `seaplus-driver.N.log` timestamped text log file, where `N` is the (operating system level) PID[7] of the process corresponding to the driver.

Example content:

```
[2019/3/6 14:32:42][debug] Starting Seaplus session...
[2019/3/6 14:32:42][debug] Starting the Seaplus C driver, with a buffer of
[2019/3/6 14:32:42][trace] Driver started.
[2019/3/6 14:32:42][debug] Read 2 bytes.
[2019/3/6 14:32:42][debug] Will read 37 bytes.
[2019/3/6 14:32:42][debug] Read 37 bytes.
[2019/3/6 14:32:42][trace] New command received.
[2019/3/6 14:32:42][debug] Read integer 2.
[2019/3/6 14:32:42][debug] Reading command: function identifier is 2.
[2019/3/6 14:32:42][debug] 2 parameter(s) received for this function.
[2019/3/6 14:32:42][debug] Executing bar/2.
[2019/3/6 14:32:42][debug] Read double 2.000000e+00.
[2019/3/6 14:32:42][debug] Read head as atom 'moderate_speed'.
[2019/3/6 14:32:42][debug] Will write 47 bytes.
```

Towards a more General C/C++ Interface

Functionally, Erl_Interface and the Erlang NIF support provide the same services, and could probably be unified under a common API (that one day Seaplus could provide).

This could enable the possibility of integrating C/C++ code seamlessly as a C-Node and/or as a NIF, for a greater flexibility of use.

Issues & Planned Enhancements

- thorough testing of the C-side should be done, notably with regard to the hunt for memory leaks; so a Valgrind-based runtime mode for the driver would surely be useful (note though that `erl_eterm_statistics/2` and `erl_eterm_release/0` are already used at runtime, in debug mode, to ensure that on the C side no term is ever leaked)

[7] Including the PID in the filename allows notably, in case of driver restart, to ensure that the logs of the new instance do not overwrite the ones of the restarted one.

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent to the project interface, or directly at the email address mentioned at the beginning of this document.

Seaplus Inner Workings

It is mostly the one described in the Erl_Interface tutorial, once augmented with conventions and automated by the Seaplus parse transform as much as realistically possible (hence a code generation that is exhaustive on the Erlang side, and partial of the C side) and adapted for increased performances (notably: no extra relay process between the user code and the port involving more messages and processing, no string-based mapping of function signatures across the bridge - direct integer identifiers used instead).

The parse transform just:

- derives from the type specifications of the Erlang service API (as specified by the service integrator) the implementation of the corresponding (Erlang-side) functions (they are injected in the AST of the resulting service BEAM file)
- adds the facility functions to start, stop, etc. that service (they are actually directly obtained through the Seaplus include)
- generates the Seaplus service-specific C header file, ready to be included by the C-side service driver that is to be filled by the service integration

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Ending Word

Have fun with Seaplus!