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

Organisation: Copyright (C) 2018-2020 Olivier Boudeville

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

Creation date: Sunday, December 23, 2018

Lastly updated: Wednesday, October 21, 2020

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

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 OS process (that, moreover, 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- raw **ports** and **linked-in drivers**: they are mostly obsolete for the task at hand (superseded by better counterparts)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, now, ei (previously 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 `ei` facilities). Doing so allows a seamless communication to happen, in spite of language heterogeneity.

C-Node and Erl_Interface/ei 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.).

As a result, the result of a Seaplus integration can be seen as an easily obtained ei-based C-Node on a bit of steroids.

`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 - i.e. whose sources may remain unknown):

```
#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 Foobar 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 (only) the type specifications of the targeted counterpart functions meant to be ultimately available (defined and exported) from Erlang, like in[2]:

```
-module(foobar).
```

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

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

-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() -> boolean().
-spec frob(tur_status()) -> text_utils:ustring().
```

The Seaplus header include allows notably to mark this `foobar` module as a service stub (so that the build system can determine this module is to be fleshed out).

It can be included in a more OTP-compliant fashion with:

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

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

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, should they involve no service-specific part; 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 receive an EXIT signal (and thus, unless trapping them, crashing 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 (that was caught by the user code)
- `stop/0`: stops the service

[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 triggered by the Seaplus-exposed `start`/`stop` service primitives.

So, for each facility function `start/0`, `start_link/0` and `stop/0`, the Seaplus parse transform determines whether it is already defined in the service at hand (i.e., for example, whether the user defined specifically a `foobar:start/0` function). If yes, then Seaplus enriches automatically that code with the one needed for its own initialisation (Seaplus'one taking place *before* the service's initialisation). If no, then Seaplus defines a brand new `start/0` that is limited to its own needs.

As a result, from the point of view of the service user, in all cases the service can be started or stopped with a single call (possibly taking care under the hood of both Seaplus and the service itself).

Of course such a module, as it was defined above (i.e. just as a set of function specifications, with no implementation thereof), 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) is available, the `Foobar` service becomes fully usable from Erlang, with no extra boilerplate code to be added by the Erlang integrator.

More precisely, for each of the function type specifications defined by the user in that module, a corresponding bridging implementation will be generated on the Erlang side 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):

```
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 as well, 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.

C-side elements will have been produced indeed by the Seaplus parse-transform (notably the function mapping include, used to map functions on either sides, and also, if not already existing, a compilable template of the C driver), but the conversion (thanks to ei now) 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? How the C implementation can maintain a state of its own between calls?

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

So: we saw that thanks to Seaplus nothing special had to be done on the Erlang side (the `foobar.erl` stub will suffice; refer to the Customising Function Bindings

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

on the Erlang Side section in order to address more specific/advanced needs), 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 a small, unedited *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` (namely, of course: `${FUNCTION_NAME}_${ARITY}_ID`) identifier (whose value happens to be `3` here[5]).

The C part of the bridge (i.e., the service driver), typically defined in `foobar_seaplus_driver.c`, 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.

Should no such driver implementation already exist, Seaplus will generate a template version of it (a template that can nevertheless be successfully compiled and linked), which will include everything needed but the (service-specific) C logic that shall be added by the service integrator in order to:

1. convert the received arguments (Erlang terms) into their C counterparts (see seaplus_getters.h for that, typically the `read_*_parameter` functions)
2. call the corresponding C integrated function
3. convert its result the other way round, so that a relevant Erlang term is returned (see seaplus_setters.h for that, typically the `write_*_result` functions)

See the full, unedited version of the generated foobar_seaplus_driver.c template corresponding to the Foobar service (one may note the placeholders in each `case` branch of the function identifier switch).

Seaplus offers moreover various helpers to facilitate the writing of this C driver (i.e. the filling of said generated template); 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()
{
```

[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 (stable by design) `BAZ_2_ID` identifier shall be trusted by user code.

```
byte * current_read_buf;

input_buffer read_buf = &current_read_buf;

// Provided by the Seaplus library:
start_seaplus_driver(read_buf);

// For the mandatory result:
output_buffer output_sm_buf;

/* Reads a full command from (receive) buffer, based on its initial lengt
 *
 * (a single term is expected hence read)
 *
 */
while (read_command(read_buf) > 0)
{

  // Current index in the input buffer (for decoding purpose):
  buffer_index index = 0;

  /* Will be set to the corresponding Seaplus-defined function identifier
   * whose value is FOO_1_ID):
   *
   */
  fun_id current_fun_id;

  /* Will be set to the number of parameters obtained from Erlang for the
   * function whose identifier has been transmitted:
   *
   */
  arity param_count;

  read_function_information(read_buf, &index, &current_fun_id, &param_cou

  prepare_for_command(&output_sm_buf);


  // 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:
       */
```

```
        long foo_a_param = read_int_parameter(read_buf, &index);

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

        // Sending of the result:
        write_int_result(&output_sm_buf, 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 = read_double_parameter(read_buf, &index);

        // Then the atom for foo_status():
        char * atom_name = read_atom_parameter(read_buf, &index);

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

        free( atom_name ) ;

        // Actual call (ownership of struct_res transferred to this caller)
        struct foo_data * struct_res = bar(bar_double_param,
                                           bar_status_param);

        // Defining a separated writing function is more convenient here:
        write_foo_data_record_from_struct(&output_sm_buf, struct_res);

        free(struct_res);

        break;

    [...]

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

    }

    finalize_command_after_writing(&output_sm_buf) ;

}
```

```
    // output_sm_buf internally already freed appropriately.

    stop_seaplus_driver(buffer);

}
```

One may finally compare the aforementioned generated template with - once it has been appropriately filled by the service integrator - the final version of this driver.

This version of course compiles, links and allows to run the `foobar_test` successfully (once Seaplus is built, one may run, from the `test/c-test` directory, `make test` for that).

If wanting to see, beyond this test, what could be an actual, more involved driver (larger, richer, partly interrupt-based), one may refer to the Ceylan-Mobile driver.

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 the final foobar_seaplus_driver.c - i.e. the ones that shall be written or filled by the service integrator), with:

- the generated ones, namely the header file for function identifier mapping (foobar_seaplus_api_mapping.h) and the original driver template (foobar_seaplus_driver.c)
- the ones implementing the Seaplus generic support, namely seaplus.hrl, seaplus.erl, seaplus.h, seaplus.c and seaplus_parse_transform.erl

As mentioned, beside the Seaplus-included Foobar example, one may refer to the Ceylan-Mobile project for a complete, standalone use of Seaplus.

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/she 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

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

- Erlang
- 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`.

Refer to the corresponding Myriad prerequisite section for more precise guidelines, knowing that Ceylan-Seaplus does not need modules with conditional support such as `crypto` or `wx`.

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.

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

One can then test the whole with:

```
$ cd test/c-test
$ make 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 on which it relies).

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[6] 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.
```

Beyond being able to collect traces about the behaviour of the driver, Seaplus more generally supports **general-purpose logging** thanks to its use of Myriad's trace_bridge (see trace_bridge_test.erl for an usage example thereof). This means that by default these messages will be output on the console (thanks to `trace_utils`), yet that any more advanced compliant trace system can be used instead (see trace_bridging_test.erl for an usage example thereof). So any library made available through Seaplus should be able to integrate nicely into one's logging system of choice.

Customising Function Bindings on the Erlang Side

We saw that, by default, no specific implementation is to be provided by the user in order to include a set of Erlang-level functions into a binding - this implementation is generated by Seaplus, and the required conversions are to be done (only) in the driver, i.e. on the C side.

However, in some cases, it may be convenient to perform transformations as well on the Erlang side, before and/or after that bridge, for example to adapt parameters or results, or to throw relevant exceptions instead of tagged tuples.

Taking this service as an example, we can see that the `get_backend_information/0` function is to return a version number that would be ideally a triplet (ex: `{1,40,0}`) so that we can compare versions easily. However the C-side happens to obtain that version from the original service as a string (ex: `"1.40.0"`). The parsing/conversion of that string into a relevant version triplet could be done in C (by building by steps a corresponding term), but it may be more convenient to do so in Erlang (ex: we may already have the right logic implemented for that).

Similarly, `get_hardware_information/0` may be not supported by the actual device, and one may prefer an exception to be thrown in that case rather than having

[6] 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.

to pattern-match the result of such a call against a tagged tuple like `{ok,Result}` vs `{error,Error}`.

This implies having the ability to **override**, on a per-function basis, the default Erlang-side implementation that would be generated by Seaplus by a user-defined one - preferably in a simple manner.

Fortunately, Seaplus offers a good support for that: should a user-provided *definition* of a function to bind be found in the service module (thus: in addition to its mere spec), it will be used (and a bit transformed automatically), instead of relying on the implementation that would be generated by default.

For that, Seaplus provides facilities to build one's custom implementation, notably the `seaplus:call_port_for/3` function that allows to automatically trigger a call on the C driver side.

So the following code will trigger a call through the port and the driver, and return its result:

```
get_backend_information() ->
    PortKey = seaplus:get_service_port_key(),
    FunctionDriverId = seaplus:get_function_driver_id(),
    {Backend,VersionString} =
        seaplus:call_port_for(PortKey,FunctionDriverId,_Args=[])
    % From here we can parse VersionString and return a triplet:
    [...]
```

Of course, should we have instead of:

```
-spec get_backend_information() -> {backend_type(), backend_version()}.
```

a function like:

```
-spec compute_sum(integer(), float()) -> float().
```

we could override the default Seaplus implementation with a one-liner that would perform exactly the same, such as:

```
compute_sum(MyInt,MyFloat) ->
    seaplus:call_port_for(seaplus:get_service_port_key(),
                          seaplus:get_function_driver_id(),
                          _Args=[MyInt,MyFloat]).
```

A user-defined implementation just has to know:

- what (service-specific) port key is to be used for that (needed by the binding, knowing that multiple different services may be bridged)
- what is the function driver identifier that was allocated to that function by Seaplus

These two information can respectively by obtained thanks to `seaplus:get_service_port_key/0` and `seaplus:get_function_driver_id()`[7].

We can see then how one can insert any (Erlang) code of interest *prior to* and/or *after* the call to the binding bridge.

[7] These are pseudo-functions that will be appropriately replaced at compilation-time with immediate values (thanks to the Seaplus parse transform). As a result, a rather optimal implementation will be obtained.

Not to mention that, on the C side, thanks to the service-specific driver, the same freedom exists as well: a call to the integrated library may be wrapped between any kind of pre/post transformations.

As a result, if needed, any mix of Erlang and C can be used to wrap any call to a library function made available through the binding.

Debugging a Seaplus-based Driver

Integrating C code is not so easy; more often than not, a SEGV will be encountered, and the fun begins in order to determine whom should we blame, typically your integration code (possible), Seaplus (possible as well) or the integrated library itself (often less likely).

The situation is never hopeless, though; we will take the integration of the libgammu library done by Ceylan-Mobile on Arch Linux as a mini-tutorial.

The type of errors that we want to track down are reported as such (real-life example of the execution of `mobile_test` while the Seaplus driver-level facilities was incorrectly dealing, memory-wise, with the parameters that were binary strings):

```
Sent first SMS whose report is: {success,255}.

<----------------
[error] Crash of the driver port (#Port<0.7>) reported.
---------------->

{"init terminating in do_boot",{{nocatch,{driver_crashed,unknown_reason}},[
```

So the driver crashed, we do not know why, and often, with such problems, nothing very relevant can be found in the Seaplus log (i.e. in `seaplus-driver.*.log`), except which API function was called when the crash happened (should you have left the corresponding `LOG_DEBUG` calls in your driver of course).

A first difficulty is that generally a (Linux) distribution will, at least by default, only include prebuilt binary packages whose libraries are stripped. For example:

```
$ file /usr/lib/libGammu.so.8.1.40.0
/usr/lib/libGammu.so.8.1.40.0: ELF 64-bit LSB shared object, x86-64, \
version 1 (SYSV), dynamically linked, BuildID[sha1]=[...], stripped
```

We *need* the debug symbols, otherwise we will lack much crucial information. Either your distribution provides a way of having unstripped, debug/development versions of some libraries, or you find it simpler and less system-jeopardizing to recompile your own unstripped versions, directly in your user account.

We go for the latter, for example with:

```
$ mkdir ~/Software/libgammu
$ cd ~/Software/libgammu
$ git clone https://github.com/gammu/gammu.git
$ ./configure --enable-shared --enable-debug --enable-protection \
   --prefix=~/Software/libgammu
$ make all install
$ file lib/libGammu.so.8.1.40.0
lib/libGammu.so.8.1.40.0: ELF 64-bit LSB shared object, x86-64, \
```

```
version 1 (SYSV), dynamically linked, BuildID[sha1]=[...], with \
debug_info, not stripped
```

Same version number - yet much better for debugging!

Now, provided that the Seaplus driver points to the right library, we should benefit from debug symbols.

A first option would be to run the driver through gdb (ex: `gdb -batch -ex run mobile_seaplus_driver`) when triggered by the application, yet we had not much luck with that approach.

Examining instead the core dump corresponding to the driver crash may offer relevant insights; provided that we find it and manage to study it.

In our case we used (as a one-liner), from the test directory, once a crash had been triggered, the following commands:

```
$ rm -f mobile_seaplus.core*
$ cp /var/lib/systemd/coredump/core.mobile_seaplus* mobile_seaplus.core.lz4
$ lz4 mobile_seaplus.core.lz4
$ gdb mobile_seaplus_driver
```

Following gdb command would then bring new information:

```
(gdb) core mobile_seaplus.core
warning: core file may not match specified executable file.
[New LWP 11607]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Core was generated by `./mobile_seaplus_driver'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007f894b2a5a26 in malloc () from /usr/lib/libc.so.6

(gdb) bt full
#0  0x00007f894b2a5a26 in malloc () from /usr/lib/libc.so.6
No symbol table info available.
#1  0x00007f894b432742 in GSM_PackSemiOctetNumber (Number=Number@entry=0x55
  semioctet=semioctet@entry=1) at [...]/libgammu/misc/coding/coding.c:1168
      format = <optimized out>
      length = 12
      i = <optimized out>
      skip = 0
[...]

(gdb) frame 2
#2  0x00007f7189618329 in GSM_EncodeSMSFrame () from /usr/lib/libGammu.so.8
(gdb) bt
#0  0x00007f7189305a26 in malloc () from /usr/lib/libc.so.6
[...]
```

While often useful, the debugger just tells us here that the SIGSEGV happened in a malloc that looks perfectly legit, and done by the inner workings of Gammu. We suspect that this library is not involved, but that we managed somehow to smash the heap in previous operations. Definitively not a good news!

So now it is time to use use Valgrind in order to investigate this possible error in memory management.

One should then have a look to the `init_driver/2` function of the seaplus.erl module, to uncomment the `DriverCommand` variation involving Valgrind.

Once using a Valgrind-based driver command and an updated environment (to select your debug library rather than the system's one), when looking at the specified log file (`/tmp/seaplus-valgrind.log`) you should end up with a report like:

```
==12257== Invalid read of size 1
==12257==    at 0x483AC74: strlen (vg_replace_strmem.c:460)
==12257==    by 0x10ACBE: main (mobile_seaplus_driver.c:438)
==12257==  Address 0x51b186c is 0 bytes after a block of size 12 alloc'd
==12257==    at 0x483777F: malloc (vg_replace_malloc.c:299)
==12257==    by 0x484DD28: erl_malloc (erl_malloc.c:234)
==12257==    by 0x484EF9A: erl_decode_it (erl_marshal.c:1041)
==12257==    by 0x484F19A: erl_decode_it (erl_marshal.c:959)
==12257==    by 0x484EE88: erl_decode_it (erl_marshal.c:1018)
==12257==    by 0x485042C: erl_decode (erl_marshal.c:1111)
==12257==    by 0x484B406: read_function_information (seaplus.c:498)
==12257==    by 0x10A7D1: main (mobile_seaplus_driver.c:245)
```

We were reading the content of a binary like if it was a zero-terminated char * (and moreover we used to wrongly take ownership of that buffer).

So neither the Ceylan-Mobile integration nor Gammu were the culprits, it was a Seaplus bug (of course fixed since then)!

Hopefully with this example one will be less afraid to hack around shared libraries (especially if they are open source): for each problem there are surely means of investigation - no rocket science involved.

Towards a more General C/C++ Interface

Functionally, ei (not to mention 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 the same 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 the Valgrind-based runtime mode for the driver is surely be useful and should be tested on a regular basis (note though that, when `Erl_Interface` was used prior to `ei`, `erl_eterm_statistics/2` and `erl_eterm_release/0` were used to monitor these issues at runtime, in debug mode - in order to ensure that on the C side no term was ever leaked)

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be sent through 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 switched to `ei` (another source of inspiration has been this article) and 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 (unless already available, their proper definitions are injected in the AST of the resulting service BEAM file, and they are exported)
- 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 integrator, based on the C template that is also generated in a proper version

As of June 2019, and related to the release of Erlang 22.0, we had to switch from the `Erl_Interface` API (now made obsolete) to the lower-level `ei` one (one may refer to the `update_to_ei` branch for that; for reference, the last version relying on `Erl_Interface`, which was working great, has been marked with the `before_switch_to_ei` tag).

A problem apparently induced by the direct use of `ei` is that, due to `term_to_binary/1` mistaking the `[0..255]` type for the `string()` one, such lists had to be special-cased, which is not so straightforward to support in a generic manner (like with Seaplus). The whole is correctly supported by Seaplus now.

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!