Distel: Distributed Emacs Lisp

3.0, updated 29 August 2002

Luke Gorrie

Table of Contents

1 Overview

Distel extends Emacs Lisp with Erlang's processes and message passing. Processes are spawned and scheduled, they send and receive messages, link with one another, crash, and so on, all within Emacs. They also use Erlang's distribution protocol to communicate with other processes in real Erlang nodes, just as other nodes would. This integration makes Emacs Lisp suitable for writing clients and front-ends to Erlang programs.

For example, this is an Emacs process that brutally terminates a remote Erlang node, via the RPC server that all nodes run by default:

```
(erl-spawn
  (erl-send [rex mynode@myhost]
            `[,erl-self [call erlang halt () ,erl-group-leader]]))
```

The equivalent Erlang program is:

```
spawn(fun() -> {rex, mynode@myhost} !
                   {self(), {call, erlang, halt, [], group_leader()}}
      end).
```

You can see that the Emacs Lisp process has much the same structure as the Erlang one. Of course, not just any Erlang program can be translated into Emacs Lisp, because only an "essential" subset of Erlang is implemented. This subset includes pattern matching, messages, process links, registered names, and distribution. It excludes the "bit syntax", ports, and so on.

The overall intention is to make Emacs a suitable user-interface toolkit for Erlang programs, especially development and debugging tools. Distel currently includes several such tools, and their implementation is straight-forward.

Here's a more meaty example to whet your appetite:

```
(defun spawn-math-server ()
  "Start a server process for doing simple calculations."
  (erl-spawn
    (erl-register 'math)
    (&math-server-loop)))

(defun &math-server-loop ()
  (erl-receive ()
      ((['calculate who what]
        (erl-send who (mcase what
                        (['add x y] (+ x y))
                        (['sub x y] (- x y))
                        (other `[bad_operation ,other]))))
       (other (message "Unexpected message: %S" other)))
    (&math-server-loop)))
```

2 Data Types

Since processes in Emacs and in Erlang can exchange messages, it's necessary to map data structures between the languages. Fortunately, there are very natural translations of most types.

2.1 Type Mapping

The following table summarises the mapping of Erlang types onto Emacs Lisp:

Atom	Symbol.
Integer	Integer. Because Emacs Lisp only supports signed 28-bit integers, this is a partial mapping.
List	List.
Tuple	Vector. For example, `{1, 2, 3}` ⇒ `[1 2 3]`
Binary	String.
PID	Vector of `[`*TYPE* `erl-pid` *node id serial creation*`]`
Port	Vector of `[`*TYPE* `erl-port` *node id creation*`]`

The *TYPE* field is an uninterned symbol, and is used to distinguish tuple-vectors from PIDs other other datatypes.

The Erlang External Term Format also includes a "string" type, which is has no clear meaning for Erlang. In general, sending strings from Erlang might not do what you want, for example the empty string will come out in emacs as NIL. When you really want to give Emacs a string, you should send it as a binary.

Some types aren't mapped yet, such as Float, Function, and Ref. It is an error to send these types to an Emacs Lisp node!

3 Processes

Processes are buffers that have PIDs and can send and receive messages. In other respects they are like normal buffers – they can contain text, have key bindings, use modes, etc.

Some processes may do nothing interesting with their buffers, like the example RPC client. User-interface processes can be more interesting by displaying information in their buffer, binding keys to commands that do work and send messages, and updating the buffer contents when messages are received.

3.1 Process State

The state of a process is recorded in buffer-local variables. Such variables make up all of the process's identity and state, so "context switching" is simply a matter of changing to another process's buffer. Although some variables are "internal", these ones can be accessed directly:

erl-self Variable

This variable is bound to the PID of the current process.

erl-group-leader Variable

This variable contains the PID of the group leader process, which handles all I/O of the process.[1]

erl-trap-exits Variable

This variable can be set to true to achieve the effect of:

```
process_flag(trap_exit, true)
```

3.2 Current Process

There is always some "current process" bound to *erl-self*, which is either a process that has been scheduled, or otherwise "the null process". The null process is a pseudo-process that never dies and is never scheduled, but is used when BIFs are called from non-process buffers. This means that any buffer can invoke BIFs, though only process buffers with their own PIDs are able to receive messages, be scheduled, create links, etc.

erl-null-pid Variable

PID of the *null process*. All messages sent to this PID are written to the '`*erl-lost-msgs*`' buffer and then discarded. When Emacs is not in the buffer of any particular process, *erl-self* is bound to `erl-null-pid`.

[1] The I/O protocol differs from the native Erlang one. See the bottom of '`erl.el`'.

4 Programming Interface

The programming interface is very much like Erlang's, and most functions do just what you would expect. There are however some important differences.

The most important difference is that when a process "schedules out" it has to return from its current function to the "scheduler loop", with a note saying what it wants to do the next time it gets scheduled (unless it's finished). This returning up the Elisp stack means that scheduling out causes all of the process's dynamic variable bindings to be undone, its `unwind-protect`'s executed, and the functions on its stack returned to (actually – bypassed by `throw`). The overall effect is that variable bindings and control state have to be explicitly passed between schedules.

In practice this is not a great imposition, since the only time a process needs to reschedule is when it blocks in "receive". The `erl-receive` construct is conveniently extended to save and restore specific variable bindings for when the process is rescheduled, and to accept series of forms to execute after the matching clause runs.

See Chapter 5 [Scheduler Internals], page 9 for more details on the scheduler's internal workings.

4.1 Spawning and Running

Emacs Lisp processes, like Erlang ones, are created by being "spawned" with some code to execute.

erl-spawn &rest *forms* Macro

Create a new process to execute *forms*, and return its PID. The process is run immediately in its own buffer, using the current dynamic environment. This means that the caller's `let` bindings are visible to the new process.

A simple example,

```
(erl-spawn (message "New PID: %S" erl-self))
⊣ New PID: [erl-pid erlmacs@kookaburra 188 0 0]
⇒ [erl-pid erlmacs@kookaburra 188 0 0]
```

(See Chapter 2 [Data Types], page 2, for details of the PID data structure.)

erl-spawn-link &rest *forms* Macro

Create a new process like `erl-spawn`, but link it with the current process before executing.

erl-spawn-async &rest *forms* Macro

Create a new process to execute *forms*, and return its PID. The process may be scheduled to run later, and thus is not guaranteed to run in the present dynamic environment.

erl-spawn-link-async &rest *forms* Macro

Create a new process like `erl-spawn-async`, but link it with the current process before executing.

erl-receive *saved-vars clauses* &rest *after* Macro

Receive a message from the process mailbox, like Erlang's `receive`. The complete syntax is similar to `mcase` (See Section 4.2 [Pattern Matching], page 5):

```
(erl-receive (saved-var ...)
    ((pattern  body...)
     ...)
  after...)
```

The first message matching a *pattern* is removed from the process mailbox, the corresponding *body* forms are executed, and then the *after* forms are executed regardless of which pattern matched. If necessary the process waits until a matching message arrives. Receiving a message involves a return to the scheduler (via `throw`), so dynamic variable bindings will *not* be preserved, except for the *saved-vars* which are saved and then restored before the *body* is executed. Variables that are matched by value in a *pattern* must also be included is *saved-vars*.

Here's a complete example, excerpted from '`erl-example.el`':

```
(defun spawn-counter ()
  (erl-spawn
    (erl-register 'counter)
    (&counter-loop 1)))

(defun &counter-loop (count)
  (erl-receive (count)
      ((msg (message "Got message #%S: %S" count msg)))
    (&counter-loop (1+ count))))
```

Important Note: Because the caller's stack is `throw`'n away by scheduling out, all calls to `erl-receive`, or to functions that will lead to its being called, should be in tail-position. In aid of this, all functions that lead to erl-receive are prefixed with an `&`, which makes them stand out in the code. With this convention, we must ensure that all calls to `&`-functions are in tail-position, and then we are safe from accidentally losing important state on the stack.

See Section 6.1 [Common Pitfalls], page 10 for important tips, either now or when you start wondering "why the fuck isn't X getting called?!" `;-)`

4.2 Pattern Matching

There are macros for pattern matching, roughly equivalent to Erlang's (minus guards). The following summarises the pattern syntax:

Trivial: `t`, `nil`, `42`, `[]`
: Numbers, and atomic types other than symbols of variables are matched literally with `equal`.

Constant: `'symbol`, `'(a b c)`
: Quoted constants are matched literally with `equal`.

Pattern Variable: `my-variable`
: Symbols denote variables. Pattern variables take on the value of the corresponding object, and are bound to lisp variables if the match succeeds. If the

same pattern variable occurs more than once, then all of its bindings must agree, as in Erlang.

Bound Variable: `,bound-var`
: Symbol preceeded by an comma. Matches the value of the (already bound) lisp variable *bound-var* with `equal`.

Wildcard: `_` (underscore)
: As in Erlang, a wildcard matches anything without creating a binding.

Sequence: `(pat1 ...)`, `[pat1 ...]`
: List or vector of patterns. Matches the "shape" (type and length) of the sequence, as well as each subpattern.

This pattern syntax is used for the macros that follow, as well as `erl-receive`.

mlet *pattern object* &rest *body* — Macro

Match *pattern* with *object*, and execute *body* with all pattern variable bindings in effect. If *pattern* doesn't match, an error is signaled. For example,

```
(mlet [value Key] my-tuple
  (message "The key is %S" key))
```

mcase *object* &rest *clauses* — Macro

Pattern matching "case" expression. The full syntax is:

```
(mcase expr
  (pattern  body...)
   ...)
```

For example, we can translate Erlang `case` expressions into Emacs `mcase`:

```
case X of
    {add, X, Y} -> X + Y;
    {sub, X, Y} -> X - Y;
    _           -> error
end.
  ⇒
(mcase x
  (['add x y] (+ x y))
  (['sub x y] (- x y))
  (_         'error))
```

4.3 BIFs

erl-send *who message* — Function

Send the term *message* to the process *who*. As in Erlang, *who* can be any of the following:

- The PID of a process, either local or remote.
- A symbol, interpreted as the registered name of a local process.
- A tuple of the form `[`*name node*`]`, indicating a remote registered process.

erl-link *pid* Function

Link the current process with *pid*.

erl-unlink *pid* Function

Unlink the current process from *pid*.

erl-exit *why* Function

Terminate the current process, with *why* as the exit reason.

erl-exit/2 *pid why* Function

Terminate the process *pid*, with *why* as exit reason.

erl-register *name* &optional *process* Function

Register *process* as *name*. If *process* is unspecified, the current process is registered.

erl-whereis *name* Function

Return the PID of the process registered with *name*, or nil if the name is unregistered.

erl-term-to-binary *object* Function

Encode *object* into the external term format, and return the result as a string.

erl-binary-to-term *string* Function

Decode *string* from the external term format into a data structure.

4.4 Tuples

tuple &rest *elements* Function

Construct a tuple from *elements*. For example,

```
(tuple 1 2 3) ⇒ [1 2 3]
```

tuple-elt *tuple n* Function

Access the *n*th element of *tuple*. As in Erlang, indexing starts from 1.

tuplep *object* Function

Type predicate for tuples.

tuple-to-list *tuple* Function

Convert *tuple* to a list. For example,

```
(tuple-to-list (tuple 1 2 3)) ⇒ (1 2 3)
```

tuple-arity *tuple* Function

Return the number of elements in *tuple*.

4.5 PIDs

erl-pid-p *object* Function

Type predicate for PIDs.

erl-local-pid-p *object* Function

Predicate for PIDs of processes inside this Emacs.

erl-remote-pid-p *object* Function

Predicate for PIDs on remote nodes.

erl-pid-node *pid* Function

Return the node name (a symbol) of the *pid*.

5 Scheduler Internals

Because Emacs Lisp has no built-in concurrency, a custom scheduler is used to run processes. This scheduler is based on a technique called *Trampolined Style* (See Section 7.3 [Related Work], page 11), which is related to continuation-passing style. The gist is that each process has a "continuation" variable containing its "next function" – the function to call when the process is next scheduled. Initially, the process's continuation function will execute the code given in the `erl-spawn` call.

To run a process, the scheduler simply switches into the process's buffer and calls its continuation function. The function does a finite amount of processing and then either simply returns, or makes a "blocking" call to `erl-receive` which sets up a next continuation to wait for the right message. If the process just returned without setting a next continuation, it is exited with reason `normal`. Otherwise the process is marked as "waiting" and left alive. Waiting processes become runnable when they receive a message, and are then invoked again with their new continuation, and so on. `erl-receive` works by continually installing itself as the next continuation until a matching message arrives, and then executing the appropriate body clause.

The scheduler itself is event-driven: the only events that can cause a process to become runnable are process creation and message delivery. On each of these events, the scheduler loop runs until no processes are left runnable, and then returns control to Emacs.

The pattern-matching `erl-receive` is not actually built into the scheduler, but is implemented in terms of the primitive function `erl-continue`.

erl-continue *function* &rest *args* Function

Set the "next continuation" of the process, such that the next time it is scheduled it will apply *function* to *args*. This will *not* occur in the same dynamic environment – the *let* bindings in effect when `erl-continue` is called won't be available to the continuation. Important state and bindings should be passed in *args* (much like an Erlang function call), or stored in a buffer local variable (much like the Erlang process dictionary).

6 Tips and Tricks

Processes' buffers are usually named either '`*pid <0.`*ID*`.0>*`', though these names can be changed to anything. Special significance is given to the pattern '`*reg` *name*`*`', which is a registered process. If you want to kill off some zombie processes, you can find their buffers (e.g. with `M-x list-buffers`) kill them with ⟨C-x k⟩ just like any other buffer. This is equivalent to the *interrupt* feature of the Erlang shell, and the process's *kill-buffer-hook* will still propagate the exit signal.

All messages between distributed nodes are recorded in trace buffers named '`*trace` *nodename*`*`', which you can look in to see the actual messages being exchanged between nodes. The buffers associated with sockets to other nodes are named '`*derl` *nodename*', and killing these buffers will safely sever connections.

6.1 Common Pitfalls

An easy mistake to make when "thinking in Erlang" is to make a call to `erl-receive` that is not in tail position. For example:

```
(defun &server-loop-bad ()
  "DO NOT WRITE CODE LIKE THIS!"
  (erl-receive ()
      ((msg (message "Got %S" msg))))
  (&server-loop-bad))
```

The `erl-receive` is not in tail position because more code (the recursion) is supposed to run after it returns. What actually happens is that `erl-receive throw`'s back up the stack to the scheduler loop, so the recursive call to `server-loop-bad` is never reached. The process just terminates after processing the first message.

The correct way to write the function is this:

```
(defun &server-loop-good ()
  "This function is fine."
  (erl-receive ()
      ((msg (message "Got %S" msg)))
    (&server-loop-good)))
```

Or alternatively,

```
(defun &server-loop-good2 ()
  "This function is also fine."
  (erl-receive ()
      ((msg (message "Got %S" msg)
            (&server-loop-good2)))))
```

Here the recursion is explicit in the body of a receive clause or in the "after" code, rather than implicit in the caller's stack.

You must *always* be careful to place calls to `erl-receive` in tail position. The same applies to calling any function that can in turn call `erl-receive`. If you forget, then code that looks like it should run will get skipped.

7 Hacking Distel

7.1 Hitch Hikers Guide

‘`erl.el`’	The process runtime system, which is the core of the system.
‘`erl-service.el`’	“High-level” Emacs commands for viewing process lists, process tracing, and so on.
‘`derl.el`’	The distribution protocol module.
‘`erlext.el`’	External term format encoding and decoding.
‘`epmd.el`’	Client for epmd, for looking up node ports.
‘`net-fsm.el`’	Generalised framework for network state machines, used to implement ‘`derl`’ and ‘`epmd`’.
‘`erl-test.el`’	A few test cases.
‘`erl-example.el`’	Example code for doing nothing-in-particular.

7.2 Documents

Various goodies are described in text files in the OTP distribution: ‘`erts/emulator/internal_doc/erl_ext_dist.txt`’ describes the distribution protocol, epmd protocol, and term format. The authentication “handshake” protocol for connecting to a node is documented in ‘`lib/kernel/internal_doc/distribution_handshake.txt`’.

7.3 Related Work

Related libraries in OTP versions past or present are *erl_interface*, *JInterface*, and *JIVE*, as well as the distribution code in the emulator itself.

Distel is also similar to *ETOS*, the Erlang to Scheme compiler, in a modest sort of a way – both systems implement Erlang runtime systems in Lisp. It’s pretty interesting to see just how neatly this can be done in Scheme, using first-class continuations. *ETOS* is available from `http://www.iro.umontreal.ca/~etos/`. For some reason the latest version is only available in binary form, but the source to an older version is there. I don’t know why it’s so.

The method of scheduling processes in Distel comes from the paper *Trampolined Style* by Steven Ganz, Daniel Friedman, and Mitchell Wand. You can find a copy at `http://citeseer.nj.nec.com/217102.html`. It’s a very pleasant read if you’re familiar with Scheme.

7.4 Contact

The Distel homepage is at `http://www.bluetail.com/~luke/distel/`. Send feedback and patches to `luke@bluetail.com`.

7.5 Document Revision History

2.0 Major additions are pattern matching and `erl-receive`. This has simplified the programming interface a lot, causing fundamentals like `erl-mailbox` to become undocumented and `erl-continue` to be reclassified as an "internal" function.

Added a reference to Trampolined Style.

1.0 First version.