# Distel: Distributed Emacs Lisp (for Erlang)

Luke Gorrie*

November 10, 2002

## Abstract

Distel is an Emacs-based user-interface toolkit for Erlang. We introduce "Emacs nodes" using the Erlang inter-node distribution protocol, and make communication natural by extending Emacs Lisp with Erlang's concurrent programming model. The extensions are intended for creating Emacs front-ends to Erlang programs, in combination with Emacs's traditional user interface facilities.

We present an introduction and tutorial on Distel programming, and show how to write a complete Erlang process manager in Emacs Lisp. We then present a suite of Emacs extensions for Erlang development called the `erlang-extended-mode`, and describe the implementation of the Distel runtime system.

## 1 Introduction

Distel (rhymes with "crystal") is intended for controlling Erlang [1] [2] programs with Emacs. The idea is to take the most essential features of Erlang and integrate them into Emacs Lisp [3], so that the two can communicate in a natural way. The features we selected are processes, pattern matching, and distribution, and they are reproduced faithfully at a high level, though many details differ. In general, higher priority is given to neat integration with Emacs Lisp than to exact reproduction of Erlang semantics.

The core of Distel is essentially an Erlang distribution library, much like the `erl_interface` for C in OTP [2], extended with ideas from the `Etos` [4] Erlang-to-Scheme compiler. Whereas `Etos` implements a complete Erlang compiler and runtime system in Scheme, Distel is a hybrid system, and implements "just enough" of Erlang to support concurrent programming in Emacs Lisp. In particular, Distel is implemented only with normal Lisp functions and macros, and has no special interpreter loop or compiler.

This paper is organised as follows. Sections 2-4 describe the Emacs Lisp programming extensions, and Section 5 uses them to present a small but complete process-manager application, as a tutorial for Distel application development. Section 6 describes the implementation of Distel itself. Section 7 describes the `erlang-extended-mode` and the development tools it includes. Sections 8-10 discuss the past, present, and future of Distel, Section 11 describes related work, and Section 12 concludes.

## 2 Processes

Emacs Lisp processes are the fundamental feature of Distel, and are provided with a set of Lisp functions and macros that correspond to Erlang's Built-In Functions (BIFs) and language constructs. In fact, the programming interface for Emacs processes is similar enough to Erlang that the best introduction is to see how a simple Erlang process can be rewritten in Emacs Lisp. A message-counting Erlang process is shown in Figure 1, and an Emacs Lisp version in Figure 2. The similarities of the programs should help to shed light on how the Emacs Lisp process works – we'll fill in the details as we go along.

We can test the Emacs message counter by spawning one and sending it some messages:

```
(erl-spawn
  (spawn-counter)
  (erl-send 'counter 'one)
  (erl-send 'counter 'two)
  (erl-send 'counter 'three))
```

*luke@bluetail.com

```
spawn_counter() ->
    spawn(fun() ->
              register(counter, self()),
              counter_loop(1)
          end).

counter_loop(Count) ->
    receive
        Msg ->
            io:format("Got msg #~p: ~p~n",
                      [Count, Msg])
    end,
    counter_loop(Count + 1).
```

Figure 1: Message counter process in Erlang

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


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

Figure 2: Message counter process in Emacs Lisp

Which will produce the following reports in the "`*Messages*`" buffer:

```
Got msg #1: one
Got msg #2: two
Got msg #3: three
```

`(erl-spawn ...)` creates a new process. It is a macro, and the enclosed code is executed in the new process. The process has its own buffer, which can be used in any way – contain text, use modes, or visit files. The buffer isn't displayed automatically, but can be made visible with Emacs functions like `display-buffer`. Because each process has its own buffer, buffer-local variables are effectively process-local – they can be used to store process state, much like the Erlang process dictionary.

`(erl-send` *who message*`)` sends a message to a process. The *who* argument accepts the same types as Erlang's `!` operator: a PID (local or remote), a registered name denoted by a symbol, or a remote registered name denoted by a `[name node]` vector. (Here, as elsewhere in Distel, vectors are used where Erlang uses tuples.)

`(erl-register` *name*`)` assigns the current process a registered name, like the `register/2` BIF in Erlang.

`(erl-receive` *saved-vars clauses after...*`)` receives a message by pattern matching (it is more complicated than Erlang's `receive`, due to implementation trade-offs discussed in Section 6.) *Saved-vars* names the local variables that will be used once a message is received (other local variables become unbound.) *Clauses* specifies which messages can be received and how they are handled. The syntax for each clause is (*pattern body...*), where *pattern* is an Erlang-style pattern (described in Section 4), and *body* is one or more Lisp expressions to run when the pattern is matched. There are also zero or more *after* expressions, which run after a message is handled, regardless of which clause matches.

Most importantly, `erl-receive` *never returns*. Instead it bundles up the execution state and `throw`'s it directly back up to a scheduler loop, bypassing any code on the stack. This is the biggest difference from Erlang programming style: in Erlang a `receive` means "handle a message and then return," but `erl-receive` means "this process state is complete – here is the next one." This is an important point for programming with Distel, and leads to writing Emacs processes in *continuation-passing style* [5] [6], where "what to do afterwards" is given explicitly to `erl-receive` instead of relying on the stack.

Because `erl-receive` doesn't return, and nor do functions that call it, they should only be *tail-called* – called as the last thing a function does. This rule is made explicit in Distel programs by the convention of naming each function that leads to `erl-receive` with an "`&`" prefix, so that we know to only call it in tail position. The `&` naming is applied to all functions that call either `erl-receive` or another `&`-function, except when the calls are wrapped in an `erl-spawn`, because `erl-spawn` catches the `throw` and returns normally.

Returning to Figure 2, we can see that `&counter-loop` is specially named because it directly calls `erl-receive`, while `spawn-counter` is not because although it calls an `&`-function, it does so inside an `erl-spawn`.

# 3 Distribution

Emacs processes can communicate directly with actual Erlang processes in other nodes, via the Erlang distribution protocol [7]. Like in Erlang, most BIFs accept either local or remote PIDs, for example `erl-send`, `erl-link`, `erl-exit`, and so on. The Erlang method of sending messages to remote registered processes also works, so to achieve:

```
{foo, bar@cockatoo} ! Message.
```

We write the equivalent:

```
(erl-send [foo bar@cockatoo] message)
```

This simple mechanism suffices to bootstrap full communication, because normal Erlang nodes automatically run a set of useful registered servers. The RPC server, registered with the name `rex`, is the most handy – it receives requests to apply a function with some arguments, and sends back the results. This server is used throughout Distel programs to make RPCs to Erlang nodes.

Of course, when a message is sent from Emacs to Erlang (or vice-versa), it is necessary to translate the data in the message between languages. In other words, we need a mapping between Erlang types and Emacs Lisp types. For Distel we have chosen a mapping that is convenient to use, though not complete or symmetric.

Some types map perfectly: lists, atoms with symbols, tuples with vectors. Integers are mapped directly, but the mapping is partial because Emacs Lisp integers are only 27 bits (Emacs has no bignums.) PIDs, Ports, and References are mapped onto vector-based structures, and tagged with a special uninterned symbol[1] to distinguish them from the vectors used for tuples.

Mapping strings from Erlang to Emacs Lisp is troublesome. The Erlang binary term encoding includes a string type, but it is used loosely – you never know whether an Erlang string will be encoded as a string or as a list of integers. To sidestep the problem, Erlang binaries are mapped onto Emacs strings, and we always use binaries to reliably send text to Emacs. Emacs Lisp strings are mapped onto Erlang strings.

Other types, such as floats and functions, are not yet mapped, and attempting to send them triggers an error.

# 4 Pattern Matching

Distel has three pattern matching macros, one being `erl-receive`, which has already been introduced. Each macro uses the same pattern syntax, described below.

`(mlet` *pattern object body...*`)` matches *object* with *pattern*, and on success executes the *body* forms with all pattern variables bound. If the match fails, an error is signalled. `mlet` is similar to Erlang's `=` operator.

`(mcase` *object clauses*`)` matches an object with a series of clauses, where the syntax of each clause is `(`*pattern body...*`)`. The first clause whose *pattern* successfully matches is selected, and its *body* forms are then executed with all pattern variables bound. If no clause matches, an error is signalled. `mcase` is of course based on Erlang's `case` expressions.

## 4.1 The Pattern Syntax

The pattern syntax is very similar to Erlang, though it lacks guards in the current implementation. The syntax is specified below, and followed by some examples.

**Trivial:** `t, nil, [], 42, ...`

Constants, matched literally.

[1] An uninterned symbol in Emacs Lisp is like a `ref` in Erlang, but it looks like a symbol.

**Sequence:** (pat1 ...), [pat1 ...]

Sequence patterns match the "shape" of the sequence, as well as each individual subpattern. The pattern can be either a list or a vector, and will only match a sequence of the same type.

**Pattern variable:** `var, my-variable, ...`

Symbols denote variables that the pattern should bind. The first time a particular variable is used it binds to the corresponding value, and then further occurrences must match this bound value.

Following a successful pattern match, a Lisp variable is bound for each pattern variable.

**Constant:** `'symbol, '(x y z)`

Quoted constants are matched literally by value.

**Bound variable:** ,var

The pattern `,var` matches the value of the pre-bound Lisp variable `var`. This is like using an already bound variable in a pattern in Erlang.

**Wild card:** _ (underscore)

Matches anything, with no binding.

For example, the Erlang code:

```
case Result of
    {ok, Value}     -> Value;
    {error, Reason} -> exit(Reason)
end
```

could be written in Lisp as:

```
(mcase result
  (['ok value]     value)
  (['error reason] (erl-exit reason)))
```

and similarly,

```
{ok, Value} = Result,
Value
```

could be written as:

```
(mlet ['ok value] result
  value)
```

# 5 A Process Manager

This section describes the design and implementation of a small but complete process-manager application. The program does two things: it presents a list of the processes running on an Erlang node, and it provides some commands to operate on them. The process list is shown in an Emacs buffer, with a one-line summary for each process. The summary line shows the PID, registered name (if any), number of reductions, and number of unreceived messages, as shown in Figure 3.

The first step in designing the application is to divide up the work between Emacs and Erlang, and decide how they will interact. The goals are to do the work on the side that makes it the easiest, and to keep the program simple by minimising the interactions.

The task for the Erlang side of the process manager is to create formatted summaries of all the processes in the system, ready for Emacs to display. The Emacs side then must fetch a process list, display it in a buffer, and provide some commands for operating on the processes. The interactions are driven from Emacs, using RPCs to the `rex` server (mentioned in Section 3.)

## 5.1 The Erlang Side

The Erlang side is implemented by the `procman` module of Figure 4, which exports the function `process_list/0`. This function returns the PID and a one-line summary of each process in the node, plus an extra line containing column headings to match the summary lines. Note that all the text is returned as binaries, to avoid the problem with strings discussed in Section 3.

```
Pid      Name                Reds     Msgs
<0.0.0>  init                3836     0
<0.2.0>  erl_prim_loader     45203    0
<0.4.0>  error_logger        245      0
<0.5.0>  application_contr   2414     0
<0.7.0>  <none>              59       0
```

Figure 3: Process Manager "screenshot"

```
-module(procman).

-export([process_list/0]).

%% Returns: {ok, Header, [ProcessInfo]}
%% ProcessInfo = {Pid, Summary}
%% Header = Summary = binary()
%%
%% Returns a one-line summary of each
%% running process along with its pid,
%% plus a heading that matches the
%% summary format.
process_list() ->
    {ok,
     fmt_row("Pid", "Name", "Reds", "Msgs"),
     [{P, info(P)} || P <- processes()]}.

info(Pid) ->
    PidName = pid_to_list(Pid),
    Reg = item(Pid, registered_name),
    Reds = item(Pid, reductions),
    Msgs = item(Pid, message_queue_len),
    fmt_row(PidName, Reg, Reds, Msgs).

item(Pid, Item) ->
    case process_info(Pid, Item) of
        {Item, Value} -> to_string(Value);
        []            -> "<none>"
    end.

fmt_row(A,B,C,D) ->
    list_to_binary(
      io_lib:format("~-8s ~-17s ~-10s ~s~n",
                    [A,B,C,D])).

to_string(X) ->
    io_lib:format("~p", [X]).
```

Figure 4: Erlang side of process manager

```
(defun pman (node)
  "Show a list of all processes on NODE."
  (interactive (list (erl-read-nodename)))
  (erl-spawn
      (display-buffer (current-buffer))
      (erl-send-rpc
       node 'procman 'process_list '())
      (erl-receive ()
          ((['rex ['ok header plist]]
            (pman-insert header plist)
            (erl-idle))
           (['rex ['badrpc reason]]
            (message "RPC failed: %S"
                     reason))))))

(defun pman-insert (header plist)
  "Insert all process information.
  PLIST is a list of [PID Summary]."
  (insert header)
  (dolist (pinfo plist)
    (mlet [pid text] pinfo
      (insert
       (propertize text
                   'pid pid)))))
```

Figure 5: Emacs pman process

### 5.2 The Emacs Lisp Side

The job for the Emacs program is to call `process_list/0` on some Erlang node and present the result. It must also record an association between summary text in a buffer and the PID of the process it represents, so that later we can write commands to operate on the process represented by a particular line of text. The code for the Emacs process is given in Figure 5.

The command `pman` creates an Emacs process and uses it to display the process list. The command takes one parameter, the Erlang node to summarise. The `interactive` declaration says that when the command is called interactively (by a key binding or `M-x`), `erl-read-nodename` is called to choose the node. This function is predefined, and will either prompt the user for a node or reuse the most recently chosen one from a cache.

The body of the function is wrapped in an erl-spawn, so it runs in a new process. Because an Emacs process has its own buffer, we use display-buffer to show it on the screen directly.

Next, the process sends an RPC to the Erlang node to call procman:process_list(). The predefined erl-send-rpc function is similar to rpc:call/4 in Erlang, its parameters are *node*, *module*, *function*, and *arguments*. The RPC server sends back the result in a {rex, Result} message, so we have an erl-receive with two patterns: one to receive the summary information on success, and one to handle any error on the Erlang side (for example, the procman module not being available.) If the summary arrives successfully, it is inserted into the buffer, and then the process calls erl-idle to enter an idle loop. The idle loop is like a receive with no patterns, meaning "schedule out indefinitely." If we had just returned without entering a receive, the process would terminate with reason normal and the user-interface buffer would be killed.

The pman-insert function takes the data we got from Erlang and puts it into the buffer for display. The header line is inserted at the top, then each summary is destructured with the mlet pattern matching macro and inserted. To preserve the association between the summary text and the process it represents, we use an Emacs feature called "text properties," which allows text in strings and buffers to be tagged with arbitrary key/value properties. The call to propertize tags the summary line with a pid property, so that later we can use get-text-property to look up the PID belonging to a piece of text in the buffer.

The process summary part is now complete, and running "M-x pman" will display a summary buffer as we showed in Figure 3.

What remains is to define a way to do things with the processes. Figure 6 shows a command to kill the process on the current line. It finds out which process we want to kill by calling get-pid-at-point, which looks up our pid property at the current location in the buffer (i.e. where the cursor is). Then it sends the process an exit signal with reason kill via the built-in erl-exit function, which is equivalent to erlang:exit/2.

```
(defun pman-kill ()
  "Kill the process under the cursor."
  (interactive)
  ;; send an EXIT signal to the process
  (erl-exit 'kill (get-pid-at-point)))

(defun get-pid-at-point ()
  "PID of the process at the point."
  (or (get-text-property (point) 'pid)
      (error "No process at point")))
```

Figure 6: Emacs kill process command

A command for displaying a process backtrace is shown in Figure 7. This is more involved than killing a process, because we must send a request for the backtrace and then receive and display the reply asynchronously. We achieve this by spawning a new process to request the backtrace, and then display the result in its own buffer when the reply arrives.

Before spawning the new process, we look up the PID that we want a backtrace for. We do this first because the code inside the erl-spawn will run in the new process' buffer, and the lookup has to be done in the buffer that has the process list. Next the new process is spawned, and uses pop-to-buffer to make its own buffer visible somewhere on the screen.

The process then makes an RPC to erlang:process_info(Pid, backtrace). The return type is {backtrace, BacktraceBinary}, which is very convenient for our purposes, since the binary will be received as a string. When the result arrives, we simply insert the backtrace text into the buffer, and enter an idle loop.

### 5.3 Summary

This procman application, though simple, is complete and useful. The approach to design used here is a good one: minimise the interactions, and do things where they are easiest. It is often best for Erlang to spoon feed Emacs, just as the procman:process_list/0 function returns a structure that is trivial for Emacs to display.

```
(defun pman-backtrace ()
  "Show backtrace of process at cursor.
  The backtrace pops up in a buffer."
  (interactive)
  (let ((pid (get-pid-at-point)))
    (erl-spawn
      (pop-to-buffer (current-buffer))
      (send-backtrace-rpc pid)
      (erl-receive ()
          ((['rex ['backtrace text]]
            (insert text)
            (erl-idle))
           (['rex ['badrpc reason]]
            (message "RPC failed: %S"
                     reason)))))))

(defun send-backtrace-rpc (pid)
  "Send an RPC for the backtrace of PID."
  (erl-send-rpc (erl-pid-node pid)
                'erlang
                'process_info
                (list pid 'backtrace)))
```

Figure 7: Emacs "backtrace" command

# 6 Runtime System

The Distel runtime system creates and schedules processes, delivers their messages, cleans up after their errors, and communicates with other nodes on the network. This section sketches the gory details of the implementation, and is not required reading for the rest of the paper.

## 6.1 Processes and Scheduling

An Emacs Lisp process is represented as an Emacs buffer, with all of its identity and state stored in buffer-local variables. The actual variables we use are `erl-self` (the PID), `erl-mailbox`, `erl-links`, and so on. There are also some cute mappings of process mechanics onto Emacs buffers, for example the `kill-buffer-hook` is used to propagate exit signals, and registered names are implemented with buffer names of "`*reg` *name*`*`". Note that because all process state is stored in buffer-local variables, context-switching just means changing buffers.

While a process is scheduled out, its state also includes a *continuation function* that can be called to resume execution from where it left off. We only ever schedule a process out when it blocks to wait for a message, so the continuations are created by `erl-receive`. The extra arguments that `erl-receive` requires reflect the difficulty of capturing the control state in Emacs Lisp, which lacks lexical closures and first-class continuations.

Each time a new process is spawned, or a message arrives from the network, the scheduler loops by invoking processes one at a time until they have all terminated or blocked in a receive. The scheduler invokes a process by switching to its buffer and then calling the continuation function, which does what it does and then either throws back a new continuation via `erl-receive`, raises an error, or simply returns. If it returns a new continuation then the process is scheduled out until a new message arrives, otherwise it is terminated by setting an `erl-exit-reason` variable and then killing its buffer (which propagates an exit signal via `kill-buffer-hook`.) This simple scheduler is based on a technique called Trampolined Style [8].

While a process is scheduled in and running, it can call BIFs to send messages and to do other process-related things. The semantics of BIFs are based on the Erlang 4.7 specification [9], and their implementation is very simple, averaging about 5 lines of code each. For example, when `(erl-send` $P$ $M$`)` is called, it either passes the request to the distribution module (if $P$ is remote), or just switches into $P$'s buffer, adds $M$ to the end of `erl-mailbox`, and marks the process as schedulable. Similarly, if process $P$ calls `(erl-link` $Q$`)`, then $Q$ is added to the `erl-links` list of $P$, and either the same is done with $Q$ or the request is handed off to distribution, depending on whether $Q$ is local.

## 6.2 Network Distribution

Distribution over the network is built from three modules: a library for binary encoding, a framework for writing network-attached state machines, and the state machine for the Erlang distribution protocol [7]. The binary coding library is a straightforward implementation of the Erlang external term format [10] using the mapping from Section 3. The networking framework supports writing simple state machines and attaching them to TCP sockets, with the crucial property of being purely event-driven and using non-blocking I/O. It is necessary that all I/O be done asynchronously, to avoid freezing Emacs while a background task waits on I/O – an often lamented property of many other Emacs networking programs.

The Distributed Erlang state machine first authenticates itself and negotiates features, and then serves requests bidirectionally. The implementation is straightforward because the distribution protocol is very high level – each message maps neatly onto a BIF. The messages implemented in Distel are:

- SEND(*PID*, *MSG*)
- LINK(*FROM*, *TO*)
- UNLINK(*FROM*, *TO*)
- EXIT(*FROM*, *TO*, *REASON*)
- REG_SEND(*FROM*, *NAME*, *MSG*)
  Send a message addressed by registered name. The PID of the sender is included so that an EXIT signal can be sent back if no such name is registered.

When a request arrives from another node, the arguments are decoded and the corresponding BIF is called. Similarly, when an Emacs BIF is called with a remote process, the request is encoded and forwarded to the node where the process is running – perhaps first being queued while a TCP connection is established.

Optional extensions, such as process monitoring, have not yet been implemented.

# 7 Applications

The Distel software distribution includes a variety of applications and tools for Erlang development. These tools are unified with a minor mode called the `erlang-extended-mode`, which complements the standard `erlang-mode`. The major features are described below, along with their commands and key bindings.

## 7.1 Dynamic "TAGS"

Distel includes a small source code cross-referencer for Erlang. The basic feature is to jump from a function call in a program to the definition of that function – for instance from the text `lists:sort(L)` to the definition of `sort/1` in `lists.erl`. The feature is similar to `etags` [3], but uses an Erlang node to dynamically find the right source files, instead of a statically generated database. The advantage is that running an Erlang node is a lot easier than maintaining a TAGS file, so the feature can be used all the time.

**erl-find-source-under-point** (`M-.`)

> Jump to a function definition. The definition will be chosen from the text at the point – either a function call, or declaration in an export list.

**erl-find-source-unwind** (`M-*`)

> Jump back from a function definition. This is a multi-level way to backtrack after following a chain of function definitions.

## 7.2 Debugger

An Erlang debugger interface, called `edb`, is also included with Distel. This uses the same interpreter-based back-end as the OTP `debugger` application, but replaces the `Tk`-based front-end with an Emacs interface. Erlang mode buffers can use `edb` commands to toggle debug-interpretation of a file, toggle a breakpoint on a line, and to pop up a "monitor buffer" to view and control debugged processes.

The monitor buffer shows all processes running debugged code, and lets you "attach" to any process that is stopped in a breakpoint. Attaching to a process pops up a buffer containing the source code of the process's current module, with a visual marker pointing to the current line. From this buffer the process can be single-stepped, its local variables can be inspected, and so on.

**edb-toggle-interpret** (`C-c C-d i`)

Toggle debug-interpretation of the current file.

**edb-toggle-breakpoint** (`C-c C-d b`)

Toggle a breakpoint on the current line.

**edb-monitor** (`C-c C-d m`)

Popup the debugger monitor buffer.

### 7.3 Process Manager

Distel includes a process manager based on the OTP `pman` application. This program is like the `procman` example of Section 5, but more polished: it uses a major mode for key bindings, and supports tracing process events via the `trace` BIF.

**erl-process-list** (`C-c C-d l`)

Pop up a process manager buffer.

### 7.4 Profiler

A front-end to the OTP `fprof` profiler is included. The `fprof` command prompts for an Erlang expression to profile, executes it with profiling on an Erlang node, and presents the results in an Emacs buffer. The result summary shows the time spent in each Erlang function, and can "zoom in" on each function to show its callers and callees.

**fprof** (`C-c C-d p`)

Profile an Erlang expression from the minibuffer.

### 7.5 Dilber: The disk_log Viewer

Dilber is a viewer for Erlang `disk_log` files, in the spirit of Unix `tail`. It is also the first "third party" Distel application – written by Vladimir Sekissov, and in on-going use as a system administration tool.

Dilber will be included in a future release of Distel.

### 7.6 Interactive Sessions

An Interactive Session buffer is to Erlang as the `*scratch*` buffer is to Emacs Lisp – a scratchpad where code snippets can be hacked and executed. The advantages over the Erlang shell are that session buffers are random-access, and that local Erlang functions can be defined individually in the buffer. This is especially useful for playing with code snippets for the `erlang-questions` mailing list – you can try Erlang functions without creating and compiling a real source file.

Interactive session buffers were conceived and implemented by David Wallin, and are included in the Distel distribution.

**erl-ie-show-session** (`C-c C-d s`)

Pop up a session buffer, creating it if necessary.

**erl-ie-copy-buffer-to-session** (`C-c C-d c`)

Create a session buffer, and copy the contents of the current buffer into it.

**erl-ie-copy-region-to-session** (`C-c C-d r`)

Create a session buffer, and copy the contents of the region into it.

### 7.7 Miscellany

**erl-eval-expression** (`C-c C-d :`)

Evaluate an Erlang expression from the minibuffer.

**erl-reload-module** (`C-d C-d L`)

Reload an Erlang module, given by name in the minibuffer.

## 8 History

Distel represents the evolution of several attempts at using Emacs as a user interface for Erlang. The first was "`erlext.el`", which began as an implementation of the Erlang external term format and was later extended with TCP socket communication. The drawback of this approach is that it needs a special TCP server to run in the Erlang node, which turned out to be too much of an obstacle for spontaneous use.

This was followed by Ermacs,[2] a concurrent Emacs clone written completely in Erlang. Ermacs is fairly complete – it has major modes for Erlang and Scheme programming, a built-in Erlang shell, and support for efficiently editing large files. However, once the core editor was complete, it was obvious that GNU Emacs has an *incredibly large* set of wonderful features, and that extending Ermacs to include "enough" of them was completely out of the question.

The lessons learned from Ermacs lead to Distel, which continues where `erlext` left off. Version 1.0 replaced `erlext`'s custom socket protocol with the Erlang distribution protocol, added very basic Emacs Lisp processes, and included a small process manager application. Version 2.0 greatly improved the programming interface with `erl-receive` and pattern matching, which made it possible for later versions to include the substantial collection of Erlang development tools available today.

## 9 Implementation Status

Distel is a stable piece of software, compatible with all recent versions of GNU Emacs and XEmacs, and suitable as an Erlang development tool without additional programming. The implementation is free software, with development hosted on SourceForge[3], and source code and documentation available on the Distel homepage:

`http://distel.sourceforge.net/`

At the time of writing, the implementation is 3,714 lines of Emacs Lisp and 994 lines of Erlang. It breaks down as follows:

- 608 lines of Emacs Lisp for the scheduler, BIFs, and process representation.

- 1,231 lines of Emacs Lisp for the distribution protocol (264 for networking, 395 for encoding and decoding, 99 for the port mapper (`epmd`) client, and 473 for the distribution protocol.)

- 1,489 lines of Emacs Lisp for the `erlang-extended-mode` (544 for the debugger, 200 for interactive sessions, and no clear division for the remainder.) All of the Erlang code is used for supporting the `erlang-extended-mode`, Distel's core doesn't require any.

The rest is made up of random examples and test suites.

## 10 Future Directions

Distel development is focused on the `erlang-extended-mode` and related tools, with language and runtime system extensions being made as they are needed. The plan is to continue adding new applications and extending Distel's capabilities as an integrated Erlang development environment. It would also be desirable to merge the useful features of Distel that don't require the runtime system into the standard (and *wonderful*) `erlang-mode`.

Using Distel for general Emacs-to-Emacs concurrent and distributed programming is another exciting possibility. Today this would require only an implementation of the port mapper (`epmd`) and for Emacs to listen for incoming connections,[4] though it may be preferable to use a completely different communications layer.

## 11 Related Work

The three main types of related work are Erlang distribution libraries for other languages, the `Etos` compiler, and other Emacs-based integrated development environments (IDEs).

Just like Distel has "Emacs nodes," the OTP applications `erl_interface` and Jive have C and Java nodes respectively. David Schere's "Erlang-Python"[5] implements Python nodes, using a binding to `erl_interface`. Others implementations may well also exist.

`Etos` [4] is an Erlang to Scheme compiler, which is related to Distel in that they both imple-

[2] http://www.bluetail.com/~luke/ermacs/

[3] http://www.sourceforge.net/

[4] At the time of writing, this seems to only be possible with the CVS version of GNU Emacs, or with an external helper program to bind the listen socket.

[5] http://starship.python.net/crew/gandalf/PyErlang/

ment high-level Erlang runtime systems in Lisp dialects. `Etos` was a good source of inspiration, and anyone who studies Distel owes it to them self to see how much more neatly things can be done with first-class continuations.

Two popular and mature Emacs-based IDEs are the Java Development Environment for Emacs (JDEE)[6], and ILISP [11] for Lisp. We hope that Distel will fill a similar niche for Erlang programmers.

Anders Lindgren's "Erl'em" program is said to have been similar in scope and purpose to Distel, but appears to have been swept away in the winds of time.[7] Anders is the main author of the Emacs `erlang-mode`.

## 12 Conclusion

We have extended Emacs Lisp for concurrent and distributed programming, and applied the extension to developing Erlang development tools. This has been a practical endeavour, and the resulting tools are immediately available to all Erlang programmers who use Emacs, as is a familiar programming interface for writing more tools.

We have also further demonstrated the power and flexibility of Emacs. Several Distel applications are highly concurrent, particularly the `edb` debugger which monitors and controls multiple processes as they run, without interfering with the user's editing. The ease with which these applications are written suggests that Emacs Lisp is very easily extended into a powerful concurrent and distributed programming system – in this case using Erlang's model, but it is easy to envision others.

Is there anything Emacs can't do?

## 13 Acknowledgements

I would like to thank Vladimir Sekissov, David Wallin, and Mats Cronqvist for their Distel hacking; Darius Bacon and Martin Björklund for their help with Distel's design and invaluable reviews of drafts of this paper (usual disclaimer applies); and all the colleagues and `erlang-questions` readers who have installed Distel and helped to iron out the (many) teething problems.

[6] http://jdee.sunsite.dk

[7] If you have a copy of this that you are allowed to distribute, please get in touch with me.

## References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] The open source erlang website. `http://www.erlang.org/`.

[3] Bill Lewis, Dan LaLiberte, and Richard Stallman. *The GNU Emacs Lisp Reference Manual*. Free Software Foundation.

[4] Marc Feeley. Etos: an erlang to scheme compiler. August 1997.

[5] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 1992.

[6] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, MIT AI Lab, December 1975.

[7] Erlang distribution protocol. Described in a text file included with the Erlang/OTP source distribution, under `lib/kernel/internal_doc/`.

[8] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *International Conference on Functional Programming*, pages 18–27, 1999.

[9] Jonas Barklund and Robert Virding. Erlang 4.7.3 reference manual. Draft (0.7), February 1999.

[10] The erlang extended term format. Described in a text file included with the Erlang/OTP source distribution, under `erts/emulator/internal_doc/`.

[11] Todd Kaufmann, Chris McConnell, Ivan Vazquez, Marco Antoniotti, Rick Campbell, and Paolo Amoroso. Ilisp user manual.