Distel User Manual

Distel 3.3, updated March 2003

Luke Gorrie

Table of Contents

1 Introduction

Distel is a set of Emacs-based programs for interacting with running Erlang nodes. The purpose is to make everyday Erlang programming tasks easier – looking up function definitions, debugging and profiling, experimenting with code snippets, and so on. It builds on the existing `erlang-mode` to provide more of the features common to Integrated Development Environments.

This manual describes Distel from the user's point of view. For details on how Distel works and how to write your own Distel-based programs, see the paper *Distel: Distributed Emacs Lisp (for Erlang)* from the proceedings of the 2002 Erlang User Conference. The paper is also available from the Distel home page, `http://distel.sourceforge.net`

1.1 Principles of Operation

Distel works by talking to Erlang nodes using the Erlang distribution protocol. It creates an "Emacs node," similar to the "C nodes" of `erl_interface`, and then talks directly to Erlang nodes with RPC and other forms of message-passing.

Of course, this means that to use Distel you need to have an Erlang node running. The node should be able to load some supporting modules for Distel to make RPCs to – setting this up is simple, and described in the '`INSTALL`' file in the distribution. Other aspects of the node's setup, such as which other modules it can find, will also affect Distel's operation. More on this in the relevant sections.

1.2 Conventions of Use

Most Distel commands need to know which Erlang node to talk to. The first command you use will prompt in the minibuffer for the name of the node to use. You can answer with either a `name@host` node name, or with just the `name` part as an abbreviation for a node on the local machine.

As a convenience, the node name you enter is cached and then reused in future commands. If you later want to talk to a different node, you can bypass the cache by prefixing a command's key sequence with the Emacs "universal prefix argument," *`C-u`*. All commands accept this prefix argument to mean "forget the cached node name and prompt for a new one." For example, *`M-.`* is the command to lookup a function definition, so *`C-u M-.`* will do the same thing but prompt for the node to use.

Some commands also accept a numeric prefix argument to alter their behaviour in some specific way. You can give a numeric prefix by holding either Control or Meta and pressing a number, followed by the command itself. For example, *`M-1 M-.`* tells the *`M-.`* command to prompt for the function to lookup, instead choosing one by looking at the source text in the buffer. The effect, if any, of a numeric prefix on a command is included in the command's documentation.

2 Programming Aids

Distel includes a few small sets of related commands to automate common programming chores. These are described in the following sections.

2.1 Cross-Referencing (Tags)

A "dynamic tags" facility effectively makes each function call in an Erlang source file into a hyperlink to the definition of that function. For example, if you have a line of code like this:

```
lists:keysort(2, L).
```

You can place the point on the function name, press *M-.*, and up pops '`lists.erl`' at the definition of `keysort/2`. After you have perused the definition to your satisfaction, you press *M-,* to jump back where you came from. You can also jump through several (sub)function definitions and then use *M-,* several times to unwind step-by-step back to where you came from.

This feature is a dynamic version of a traditional Emacs facility called "Tags." Whereas Tags needs you to maintain a special '`TAGS`' file to keep track of definitions, Distel simply asks an Erlang node, "Where is the source file for module `foo`?" The Erlang node makes a well-educated guess at which source file we want (based on the location and attributes of the beam file for the same module), and sends back the path. Emacs then opens the file and scans down to the definition of the function with the right arity.

If you have several versions of the same source file (perhaps belonging to separate branches in revision control), then Distel will find the one that matches the code in the Erlang node you're talking to. So, to work on a particular source tree you just connect to a node that has the matching code in its code path.

M-.
: Jump from a function call to the definition of the function (`erl-find-source-under-point`). If the variable `distel-tags-compliant` is non-nil, or a numeric prefix argument is given, this command prompts for the function name to lookup.

M-,
: Jump back from a function definition (`erl-find-source-unwind`). This is a multi-level unwind through a stack of positions from which you have jumped with *M-.* The command is also bound to *M-** for consistency with "etags."

To actually find the source file for a particular module, the Erlang node first ensures that it can load the module, and then tries each of these locations in order:

1. Same directory as the beam file.
2. '`../src/`' from the beam file.
3. '`../erl/`' from the beam file.
4. The directory from which the beam file was compiled. We can find this using `module_info/1`, because the compiler records it as an attribute in the beam file.

2.2 Completion of Modules and Functions

Completion allows you to write some part of a module or remote function name and then press *M-TAB* to have it completed automatically. When multiple completions exist they are displayed in a popup buffer, much like Emacs's normal filename completion. The completion buffer can simply be read to see which completions exist, or either *RET* or the middle mouse button can be used to select one.

M-TAB Complete the module or function at point. (`erl-complete`)

M-? Alternative binding for `erl-complete`, since *M-TAB* is often reserved by window managers.

2.3 Evaluting Erlang Code

Distel includes some simple ways to evaluate Erlang code, described here. More elaborate interactive evaluation is provided by Interactive Sessions (see Section 3.3 [Interactive Sessions], page 8).

C-c C-d : Read an Erlang expression from the minibuffer, evaluate it on an Erlang node, and show the result. (`erl-eval-expression`)

C-c C-d L Read a module name from the minibuffer and reload that module in an Erlang node. (`erl-reload-module`)

2.4 Refactoring

The Refactoring feature requires the syntax-tools package to be in the Erlang node's code path. You can download syntax-tools from the erlang.org "User Contributions" area.

Expressions within functions can be automatically "refactored" into their own subfunctions by using the `erl-refactor-subfunction` command (*C-c C-d f*). The command takes the text of the expression, determines which variables it needs from the original function, and then generates the new function and puts it on the kill ring for insertion by hand (with `yank`, *C-y*). The original function is rewritten with a call to the subfunction where the refactored expression used to be.

For example, suppose we want to refactor the following function:

```
eval_expression(S) ->
    case parse_expr(S) of
        {ok, Parse} ->
            case catch erl_eval:exprs(Parse, []) of
                {value, V, _} ->
                    {ok, flatten(io_lib:format("~p", [V]))};
                {'EXIT', Reason} ->
                    {error, Reason}
            end;
        {error, {_, erl_parse, Err}} ->
            {error, Err}
    end.
```

In this example we will take the inner `case` expression and move it into a new function called `try_evaluation`. We do this by setting the Emacs region (using the mouse or *C-SPC*) from the word `case` until the end of the word `end` – marking exactly one whole expression. We then enter *C-c C-d f* to refactor, and when prompted for the function name we respond with "`try_evaluation`". The original function is then rewritten to:

```
eval_expression(S) ->
    case parse_expr(S) of
       {ok, Parse} ->
           try_evaluation(Parse);
       {error, {_, erl_parse, Err}} ->
           {error, Err}
    end.
```

And at the front of the kill ring we have the new function definition, which can be pasted into the buffer wherever we want. The actual definition we get is:

```
try_evaluation(Parse) ->
    case catch erl_eval:exprs(Parse, []) of
       {value, V, _} ->
           {ok, flatten(io_lib:format("~p", [V]))};
       {'EXIT', Reason} ->
           {error, Reason}
    end.
```

Important note: This command is not a "pure" refactoring, because although it will import variables from the parent function into the subfunction, it will not export new bindings created in the subfunction back to the parent. However, if you follow good programming practice and never "export" variables from inner expressions, this is not a problem. An example of *bad* code that will not refactor correctly is this `if` expression:

```
if A < B -> X = true;
   B > A -> X = false
end,
foo(X)
```

This is in poor style – a variable created inside the `if` is used by code at an outer level of nesting. To work with refactoring, and to be in better style, it should be rewritten like this:

```
X = if A < B -> true;
       B > A -> false
    end,
foo(X)
```

2.5 Documentation

Simple online Erlang documentation is provided via an Erlang program called `fdoc`. The documentation is automatically scanned out of source files by building a searchable database of the comments appearing before each function. Naturally, the quality of documentation provided by this scheme will depend on the style in which the source files are commented.

C-c C-d d Describe an Erlang module or function by name. (`erl-fdoc-describe`)

C-c C-d a
: Show apropos information about Erlang functions, by regular expression. All functions whose names or comments match the regexp are displayed. (`erl-fdoc-apropos`)

With a numeric prefix argument, these commands rebuild the `fdoc` database before searching. This is useful after (re)loading a lot of modules, since `fdoc` only scans the currently loaded modules for documentation when it builds the database.

3 Applications

This chapter describes the larger applications included with Distel.

3.1 Process Manager

The process manager displays a list of all the processes running on an Erlang node, and offers commands to manipulate them.

C-c C-d l Popup a process manager buffer. (`erl-process-list`)

Within the process manager's buffer, the following commands are available:

q Quit the process manager, and restore the Emacs windows as they were before it popped up.

u Update the process list.

k Kill a process.

RET Pop up a buffer showing all the information about a process. The buffer also continuously traces the process by appending events to the buffer, until the buffer is killed with *q*.

i Show a piece of information about the process, specified by name. The name can be any key accepted by the `process_info/2` BIF.

b Show a backtrace for a process. The backtrace is a fairly low-level snapshot of the stack of a process, obtained from `process_info(P, backtrace)`. It may take a little pratice to learn how to read them.

m Show the contents of a process's mailbox.

3.2 Debugger

Distel includes a front-end to the Erlang debugger, using the same backend as the standard Tk-based OTP debugger. The Distel debugger has three parts: commands in Erlang source buffers for interpreting modules and configuring breakpoints, a "Monitor" buffer listing processes running interpreted code, and one "Attach" buffer for each process that is being single-stepped.

3.2.1 Basic Commands

C-c C-d i Toggle interpretedness of the current buffer's module. (`edb-toggle-interpret`)

C-x SPC Toggle a breakpoint on the current line. (`edb-toggle-breakpoint`)

C-c C-d m Popup the Monitor buffer. (`edb-monitor`)

3.2.2 Monitor Buffer

The monitor buffer displays all processes that the debugger knows about, line-by-line. This includes all processes that have run interpreted code, and all that are stopped in breakpoints. The current status of each process is shown – running, at breakpoint, or exited. You can attach to a debugged process by pressing *RET* on its summary line.

RET Popup an attach buffer for a process.

q Hide the Monitor buffer and restore Emacs' window configuration to the way it was before.

k Kill the monitor. This disconnects Emacs from the Erlang node's debugging state and deletes all the local debugging state (e.g. breakpoints in buffers.) The next debugger command will automatically re-attach the monitor.

3.2.3 Attach Buffer

An attach buffer corresponds to a particular Erlang process that is being debugged. It displays the source to the module currently being executed and, when the process is stopped at a breakpoint, an arrow showing the next line of execution. The attach buffer is accompanied by a buffer showing the variable bindings in the current stack frame.

SPC Step into the next expression. If the expression is a function call, the debugger will enter that function. (`edb-attach-step`)

n Step over the next expression, without going down into a subfunction. (`edb-attach-next`)

c Continue execution until the next breakpoint. (`edb-attach-continue`)

u Show the previous stack frame. (`edb-attach-up`)

d Show the next stack frame. (`edb-attach-down`)

b Toggle a breakpoint on the current line. (`edb-toggle-breakpoint`)

q Kill the attach buffer. This does not affect the actual Erlang process.

h Display online help, showing essentially this information. (`edb-attach-help`)

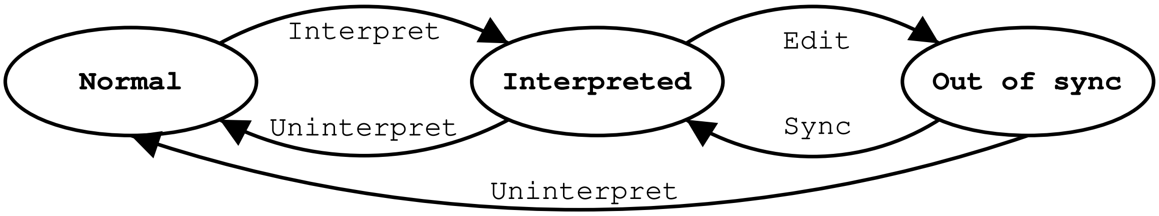
3.2.4 Synchronising Breakpoints

At any point in time the breakpoints in a particular buffer will be either "fresh" or "stale," depending on whether the buffer has been modified. Breakpoints are fresh when, as far as Emacs knows, the buffer's source text (line numbers) correspond with the code in the Erlang node. After the buffer is modified, the breakpoints become stale, because edits may change line numbers so that the breakpoints in Emacs no longer correspond with the actual program. Stale breakpoints are made fresh by using the `edb-synch-breakpoints` (*C-c C-d s*) command to reassert that they are correct. This command is typically used after recompiling and reloading the module.

Fresh breakpoints are marked in red, stale breakpoints are marked in purple.

C-c C-d s Synchronise breakpoints by discarding the ones in the Erlang node and then re-setting them from those in the Emacs buffer.

The overall debugger state machine for Erlang-mode buffers is shown in this figure:



In the "Normal" state, no breakpoints exist. In the "Interpreted" state, all breakpoints are fresh. In the "Out of sync" state, all breakpoints are stale. The transitions illustrate how you can navigate between the states.

Care must be taken to only synchronise breakpoints when the Erlang node is actually running the same code that is in the Emacs buffer. Otherwise, the Erlang processes may break in unexpected places.

When reloading modules during debugging, it is preferable to use the `erl-reload-module` command (*C-c C-d L*, see Section 2.3 [Evaluation], page 3) than to call `l(mymodule)` directly in the Erlang shell. This is because the Distel command is specially coded to make sure reloading interpreted modules keeps them interpreted, but this doesn't appear to work correctly in the Erlang shell.

3.2.5 Saving and Restoring Debugger State

The overall debugger state (set of breakpoints and interpreted modules) can be temporarily saved inside Emacs and then restored to the Erlang node. This is particularly useful when you want to restart the Erlang node and then continue debugging where you left off: you just save the debug state, restart the node, and then restore.

C-c C-d S Save the set of breakpoints and interpreted modules inside Emacs. (`edb-save-dbg-state`)

C-c C-d R Restore Emacs's saved debugger state to the Erlang node. (`edb-restore-dbg-state`)

3.3 Interactive Sessions

Interactive sessions are an Erlang version of the Emacs Lisp '`*scratch*`' buffer. You can enter arbitrary Erlang expressions and function definitions in an interactive session buffer and evaluate them immediately, without creating any files.

C-c C-d e Display the Interactive Session buffer for an Erlang node, creating it if necessary. (`erl-ie-show-session`)

Within the session buffer, these commands are available:

C-j Evaluate the Erlang expression on the current line, and insert the result in-line. (`erl-ie-eval-expression`)

C-M-x Evaluate the function definition before the point. Once defined, the function can then be called from expressions in the session buffer, and can be redefined later.

3.4 Profiler

Distel supports profiling function calls via the OTP `fprof` application. This is a very convenient profiler, in that it doesn't require any special compiler options or initialisation – you can use it whenever you want.

C-c C-d p Prompt for an Erlang expression, evaluate it with profiling, and then summarise the results. (`fprof`)

C-c C-d P Load and display prerecorded profiler data, from a file created by `fprof:analyse/1`. (`fprof-analyse`)

After an expression is profiled, the results are popped up in "profiler results" buffer. The buffer contains one line to describe each function that was called, each with the following columns:

`Calls` The total number of times the function was called.

`ACC` The total time (ms) spent in the function, including its callees.

`Own` The total time (ms) spent by the function itself, excluding time spent in its callees.

Furthermore, pressing *RET* on a summary line in the results buffer will pop up another buffer showing more information about the function: how much time it spent on behalf of each of its callers, and how much time it spent in each of its subfunctions.