Event Tracer (ET)

Event Tracer (ET) 1.3.3

November 23 2009

November 23 2009

1 User's Guide

The Event Tracer (ET) uses the built-in trace mechanism in Erlang and provides tools for collection and graphical viewing of trace data.

1.1 Introduction

The Event Tracer (ET) uses the built-in trace mechanism in Erlang and provides tools for collection and graphical viewing of trace data.

The viewed trace data is normally collected from Erlang trace ports or files.

1.1.1 Scope and Purpose

This manual describes the Event Tracer (ET) application, as a component of the Erlang/Open Telecom Platform development environment. It is assumed that the reader is familiar with the Erlang Development Environment, which is described in a separate User's Guide.

1.1.2 Prerequisites

The following prerequisites is required for understanding the material in the Event Tracer (ET) User's Guide:

- familiarity with the Erlang system and Erlang programming in general and the especially the art of Erlang tracing.

The application requires Erlang/OTP release R7B or later.

1.1.3 About This Manual

In addition to this introductory chapter, the Megaco User's Guide contains the following chapters:

- Chapter 2: "Usage" describes the architecture and typical usage of the application.
- Chapter 3: "Examples" gives some usage examples

1.1.4 Where to Find More Information

Refer to the following documentation for more information about Event Tracer (ET) and about the Erlang/OTP development system:

- the Reference Manual of the `Event Tracer (ET)`.
- documentation of basic tracing in `erlang:trace/4` and `erlang:trace_pattern/3` and then the utilities derived from these: `dbg`, `observer` and `et`.
- Concurrent Programming in Erlang, 2nd Edition (1996), Prentice-Hall, ISBN 0-13-508301-X.

1.2 Usage

1.2.1 Overview

The two major components of the Event Tracer (ET) tool is a graphical sequence chart viewer (`et_viewer`) and its backing storage (`et_collector`). One collector may be used as backing storage for several simultaneous viewers where each one may display a different view of the same trace data.

The interface between the collector and its viewers is public in order to enable other types of viewers. However in the following text we will focus on usage of the `et_viewer`.

The main start function is `et_viewer:start/1`. It will by default start both an `et_collector` and an `et_viewer`:

```
% erl -pa et/examples
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
1> {ok, Viewer} = et_viewer:start([]).
{ok,<0.31.0>}
```

A viewer gets trace events from its collector by polling it regularly for more events to display. Trace events are for example reported to the collector with `et_collector:report_event/6`:

```
2> Collector = et_viewer:get_collector_pid(Viewer).
<0.30.0>
3> et_collector:report_event(Collector, 60, my_shell, mnesia_tm, start_outer,
                             "Start outer transaction"),
3> et_collector:report_event(Collector, 40, mnesia_tm, my_shell, new_tid,
                             "New transaction id is 4711"),
3> et_collector:report_event(Collector, 20, my_shell, mnesia_locker, try_write_lock,
                             "Acquire write lock for {my_tab, key}"),
3> et_collector:report_event(Collector, 10, mnesia_locker, my_shell, granted,
                             "You got the write lock for {my_tab, key}"),
3> et_collector:report_event(Collector, 60, my_shell, do_commit,
                             "Perform  transaction commit"),
3> et_collector:report_event(Collector, 40, my_shell, mnesia_locker, release_tid,
                             "Release all locks for transaction 4711"),
3> et_collector:report_event(Collector, 60, my_shell, mnesia_tm, delete_transaction,
                             "End of outer transaction"),
3> et_collector:report_event(Collector, 20, my_shell, end_outer,
                             "Transaction returned {atomic, ok}").
{ok,{table_handle,<0.30.0>,11,trace_ts,#Fun<et_collector.0.83904657>}}
4>
```

This is a simulation of the process events caused by a Mnesia transaction that writes a record in a local table:

```
mnesia:transaction(fun() -> mnesia:write({my_tab, key, val}) end).
```

At this stage when we have a couple of events, it is time to show how it looks like in the graphical interface of `et_viewer`:

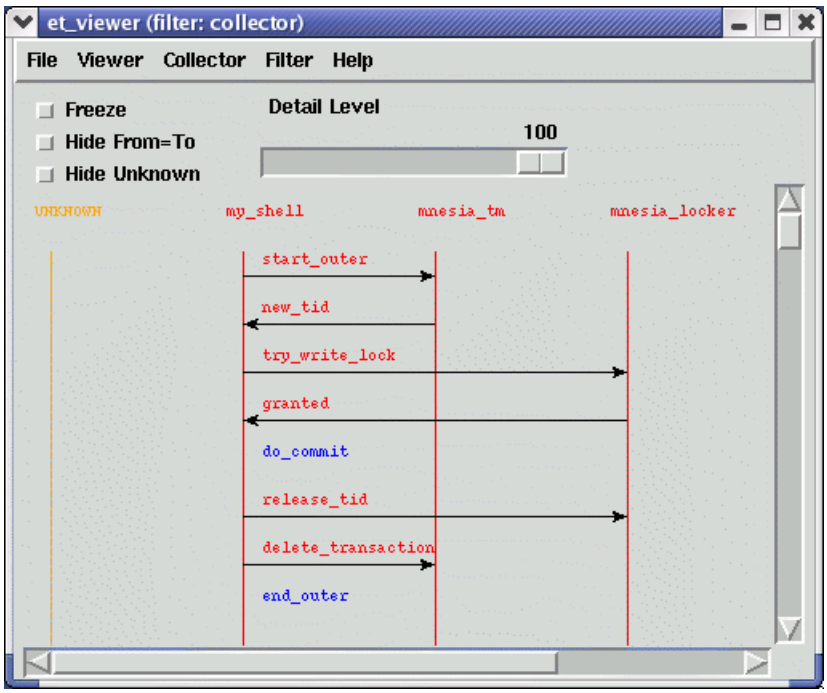

Figure 2.1: A simulated Mnesia transaction which writes one record

In the sequence chart, the actors (which symbolically has performed the traced event) are shown as named vertical bars. The order of the actors may be altered by dragging (hold mouse button 1 pressed during the operation) the name tag of an actor and drop it elsewhere:

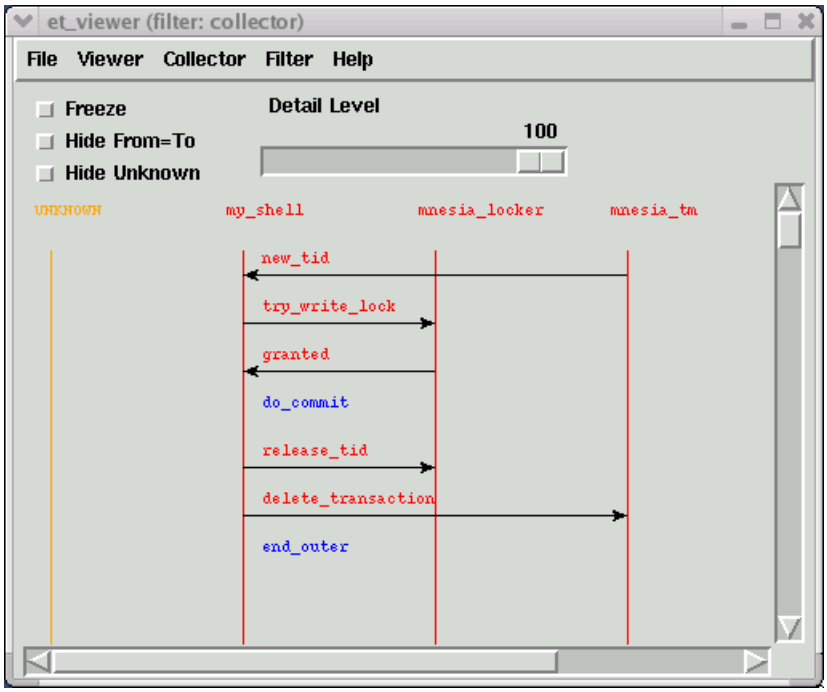

Figure 2.2: Two actors has switched places

An event may be an action performed by one single actor (blue text label) or it may involve two actors and is then depicted as an arrow directed from one actor to another (red text label). Details of an event can be shown by clicking (press and release the mouse button 1) on the event label text or on the arrow:

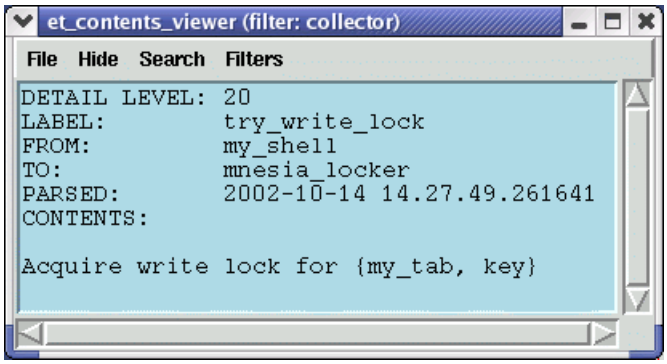

Figure 2.3: Details of a write lock message

1.2.2 Filters and dictionary

The Event Tracer (ET) uses named filters in various contexts. An Event Trace filter is an `Erlang fun` that takes some trace data as input and returns a possibly modified version of it:

```
filter(TraceData) -> true | {true, NewEvent} | false

TraceData = NewEvent | term()
NewEvent  = record(event)
```

The interface of the filter function is the same as the the filter functions for the good old `lists:zf/2`. If the filter returns `false` it means that the `TraceData` should be dropped. `{true, NewEvent}` means that the `TraceData` should be replaced with `NewEvent`. And `true` means that the `TraceData` data already is an event record and that it should be kept as it is.

The first filter that the trace data is exposed for is the collector filter. When a trace event is reported with `et_collector:report/2` (or `et_collector:report_event/5,6`) the first thing that happens, is that a message is sent to the collector process to fetch a handle that contains some useful stuff, such as the collector filter fun and an ets table identifier. Then the collector filter fun is applied and if it returns `true` (or `{true, NewEvent}`), the event will be stored in an ets table. As an optimization, subsequent calls to `et_collector:report`-functions can use the handle directly instead of the collector pid.

The collector filter (that is the filter named `collector`) is a little bit special, as its input may be any Erlang term and is not limited to take an event record as the other filter functions.

The collector manages a key/value based dictionary, where the filters are stored. Updates of the dictionary is propagated to all subscribing processes. When a viewer is started it is registered as a subscriber of dictionary updates.

In a viewer there is only one filter that is active and all trace events that the viewer gets from the collector will pass thru that filter. By writing clever filters it is possible to customize how the events looks like in the viewer. The following filter replaces the actor names `mnesia_tm` and `mnesia_locker` and leaves everything else in the record as it was:

If we now add the filter to the running collector:

```
4> Fun = fun(E) -> et_demo:mgr_actors(E) end.
#Fun<erl_eval.5.123085357>
5> et_collector:dict_insert(Collector, {filter, mgr_actors}, Fun).
ok
6>
```

you will see that the `Filter` menu in all viewers have got a new entry called `mgr_actors`. Select it, and a new viewer window will pop up:

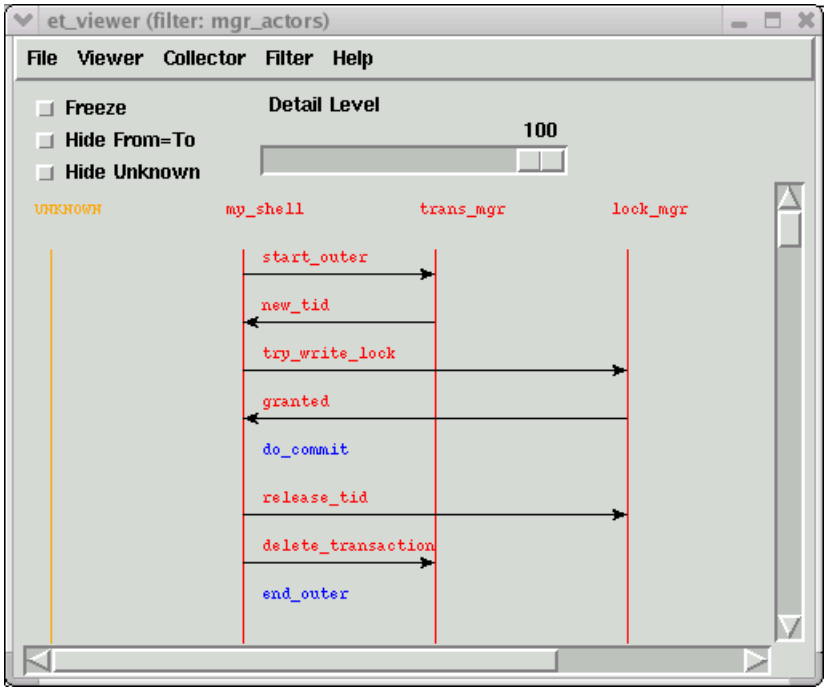

Figure 2.4: The same trace data in a different view

In order to see the nitty gritty details of an event you may click on the event in order to start a contents viewer for that event. In the contents viewer there is also a filter menu in order to enable inspection of the event from other views than the one selected in the viewer. A click on the `new_tid` event will cause a contents viewer window to pop up, showing the event in the `mgr_actors` view:

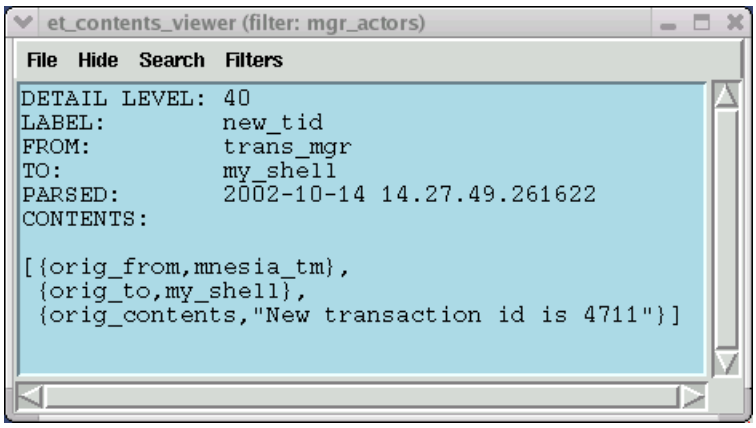

Figure 2.5: The trace event in the mgr_actors view

Select the `collector` entry in the `Filters` menu and a new contents viewer window will pop up showing the same trace event in the collectors view:

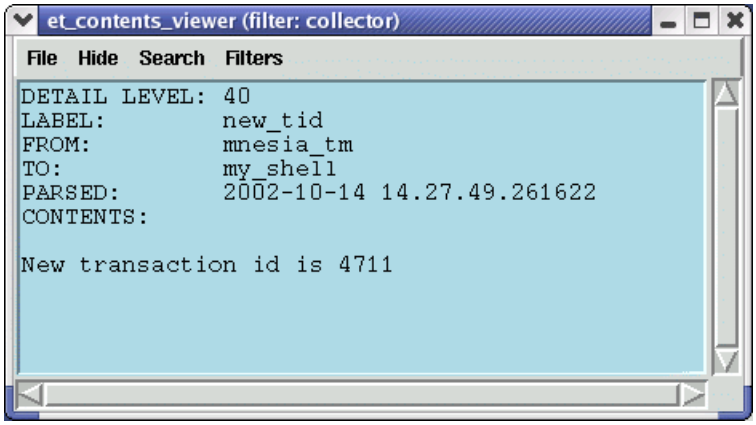

Figure 2.6: The same trace event in the collectors view

1.2.3 Trace clients

As you have seen it is possible to use the `et_collector:report`-functions explicitly. By using those functions you can write your own trace client that reads trace data from any source stored in any format and just feed the collector with it. You may replace the default collector filter with a filter that converts new exciting trace data formats to event-records or you may convert it to an event-record before you invoke `et_collector:report/2` and then rely on the default collector filter to handle the new format.

There are also existing functions in the API that reads from various sources and calls `et_collector:report/2`:

- The trace events that are hosted by the collector may be stored to file and later be loaded by selecting `save` and `load` entries in the viewers `File`-menu or via the `et_collector` API.
- It is also possible to perform live tracing of a running system by making use of the built-in trace support in the Erlang emulator. These Erlang traces can be directed to files or to ports. See the reference manual for `erlang:trace/4`, `erlang:trace_pattern/3`, `dbg` and `ttb` for more info.

  There are also corresponding trace client types that can read the Erlang trace data format from such files or ports. The `et_collector:start_trace_client/3` function makes use of these Erlang trace clients and redirects the trace data to the collector.

The default collector filter converts the Erlang trace data format into event-records.If you want to perform this differently you can of course write your own collector filter from scratch. But it may probably save you some efforts if you first apply the default filter in `et_selector:parse_event/2` before you apply your own conversions of its output.

1.2.4 Global tracing and phone home

Setting up an Erlang tracer on a set of nodes and connecting trace clients to the ports of these tracers is not intuitive. In order to make this it easier the Event Tracer as a notion of global tracing. When used, the `et_collector` process will monitor Erlang nodes and when one connects, an Erlang tracer will automatically be started on the other node. A corresponding trace client will also be started on the collector node in order to automatically forward the trace events to the collector. Set the boolean parameter `trace_global` to `true` for either the `et_collector` or `et_viewer` in order to activate the global tracing. There is no restriction on how many concurrent (anonymous) collectors you can have, but you can only have one global collector as its name is registered in `global`.

In order to further simplify the tracing you can make use of the `et:report_event/4,5` (or its equivalents `et:phone_home/4,5` :-). These functions are intended to be invoked from other applications when there are interesting events, in your application that needs to be highlighted. The functions are extremely light weight as they do nothing besides returning an atom. These functions are specifically designed to be traced for. As the caller explicitly provides the values for the event-record fields, the default collector filter is able to automatically provide a customized event-record without any user defined filter functions.

In normal operation the `et:report_event/4,5` calls are almost for free. When tracing is needed you can either activate tracing on these functions explicitly. Or you can combine the usage of `trace_global` with the usage of `trace_pattern`. When set, the `trace_pattern` will automatically be activated on all connected nodes.

One nice thing with the `trace_pattern` is that it provides a very simple way of minimizing the amount of generated trace data by allowing you to explicitly control the detail level of the tracing. As you may have seen the `et_viewer` have a slider called `"Detail Level"` that allows you to control the detail level of the trace events displayed in the viewer. On the other hand if you set a low detail level in the `trace_pattern`, lots of the trace data will never be generated and thus not sent over the socket to the trace client and stored in the collector.

1.2.5 Viewer window

Almost all functionality available in the `et_viewer` is also available via shortcuts. Which key that has the same effect as selecting a menu entry is shown enclosed in parentheses. For example pressing the key `r` is equivalent to selecting the menu entry `Viewer->Refresh`.

File menu:

- Close Collector and all Viewers - Close the collector and all viewers connected to that collector.
- Close other Viewers, but keep Collector - Keep this viewer and its collector, but close all other viewers connected to this collector.
- Close this Viewer, but keep Collector - Close this viewer, but all other viewers and the collector.
- Save Collector to file - Save all events stored in the collector to file.
- Load Collector from file - Load the collector with events from a file.

Viewer menu:

- First - Scroll `this` viewer to the first collector event.
- Prev - Scroll `this` viewer one "page" backwards. Normally this means that the first event displayed in the viewer will be the last one and the previous `max_events` events will be read from the collector.
- Next - Scroll `this` viewer one "page" forward. Normally this means that the last event displayed in the viewer will be the first one and `max_events` more events will be read from the collector.
- Last - Scroll `this` viewer to the last collector event.

- Refresh - Clear `this` viewer and re-read its events from the collector.
- Up 5 - Scroll 5 events backwards.
- Down 5 - Scroll 5 events forward.
- Abort search. Display all. - Switch the display mode to show all events regardless of any ongoing searches. Abort the searches.

Collector menu:

- First - Scroll `all` viewers to the first collector event.
- Prev - Scroll `all` viewers one "page" backwards. Normally this means that the first event displayed in the viewer will be the last one and the previous `max_events` events will be read from the collector.
- Next - Scroll `all` viewers one "page" forward. Normally this means that the last event displayed in the viewer will be the first one and `max_events` more events will be read from the collector.
- Last - Scroll `all` viewers to the last collector event.
- Refresh - Clear `all` viewers and re-read their events from the collector.

Filters menu:

- ActiveFilter (=) - Start a new viewer window with the same active filter and scale as the current one.
- ActiveFilter (+) - Start a new viewer window with the same active filter but a larger scale than the current one.
- ActiveFilter (-) - Start a new viewer window with the same active filter but a smaller scale than the current one.
- collector (0) - Start a new viewer with the collector filter as active filter.
- AnotherFilter (2) - If more filters are inserted into the dictionary, these will turn up here as entries in the `Filters` menu. The second filter will be number 2, the next one number 3 etc. The names are sorted.

Slider and radio buttons:

- Freeze - When true, this means that the viewer will not read more events from the collector until set to false.
- Hide From=To - When true, this means that the viewer will hide all events where the from-actor equals to its to-actor.
- Hide Unknown - When true, this means that the viewer will hide all events where either of the from-actor or to-actor is `UNKNOWN`.
- Detail level - This slider controls the resolution of the viewer. Only events with a detail level `smaller` than the selected one (default=100=max) are displayed.

Other features:

- Display details of an event - Click on the event name and a new window will pop up, displaying the contents of an event.
- Toggle actor search - Normally the viewer will be in a display mode where all events are shown. By clicking on an actor name the tool will switch display mode to only show events with selected actors.

  Click on an actor and only events with that actor will be displayed. Click on another actor to include that actor to the selected ones. Clicking on an already selected actor will remove it from the collections of selected actors. When the collection of selected actors becomes empty the normal mode where all actors are shown will be entered again.

  Abort actor search with the `a` key or with the `Viewer->Abort search` menu choice.
- Move actor - Drag and drop an actor by first clicking on the actor name, keeping the button pressed while moving the cursor to a new location and release the button where the actor should be moved to.

1.2.6 Configuration

The event-records in the ets-table are ordered by their timestamp. Which timestamp that should be used is controlled via the `event_order` parameter. Default is `trace_ts` which means the time when the trace data was generated. `event_ts` means the time when the trace data was parsed (transformed into an event-record).

1.2.7 Contents viewer window

File menu:

- Close - Close this window.
- Save - Save the contents of this window to file.

Filters menu:

- ActiveFilter - Start a new contents viewer window with the same active filter.
- AnotherFilter (2) - If more filters are inserted into the dictionary, these will turn up here as entries in the `Filters` menu. The second filter will be number 2, the next one number 3 etc. The names are sorted.

Hide menu:

- Hide actor in viewer - Known actors are shown as a named vertical bars in the viewer window. By hiding the actor, its vertical bar will be removed and the viewer will be refreshed.

  Hiding the actor is only useful if the `max_actors` threshold has been reached, as it then will imply that the "hidden" actor will be displayed as if it were `"UNKNOWN"`. If the `max_actors` threshold not have been reached, the actor will re-appear as a vertical bar in the viewer.
- Show actor in viewer - This implies that the actor will be added as a known actor in the viewer with its own vertical bar.

Search menu:

- Forward from this event - Set this event to be the first event in the viewer and change its display mode to be enter forward search mode. The actor of this event (from, to or both) will be added to the list of selected actors.
- Reverse from this event - Set this event to be the first event in the viewer and change its display mode to be enter reverse search mode. The actor of this event (from, to or both) will be added to the list of selected actors. Observe, that the events will be shown in reverse order.
- Abort search. Display all - Switch the display mode of the viewer to show all events regardless of any ongoing searches. Abort the searches.

1.3 Examples

1.3.1 A simulated Mnesia transaction

The Erlang code for running the simulated Mnesia transaction example in the previous chapter is included in the `et/examples/et_demo.erl` file:

If you invoke the `et_demo:sim_trans()` function a viewer window will pop up and the sequence trace will be almost the same as if the following Mnesia transaction would have been run:

```
mnesia:transaction(fun() -> mnesia:write({my_tab, key, val}) end).
```

And the viewer window will look like:

```
$ erl -pa ../examples
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
```

```
1> et_demo:sim_trans().
{ok,{table_handle,<0.30.0>,11,trace_ts,#Fun<et_collector.0.83904657>}}
2>
```

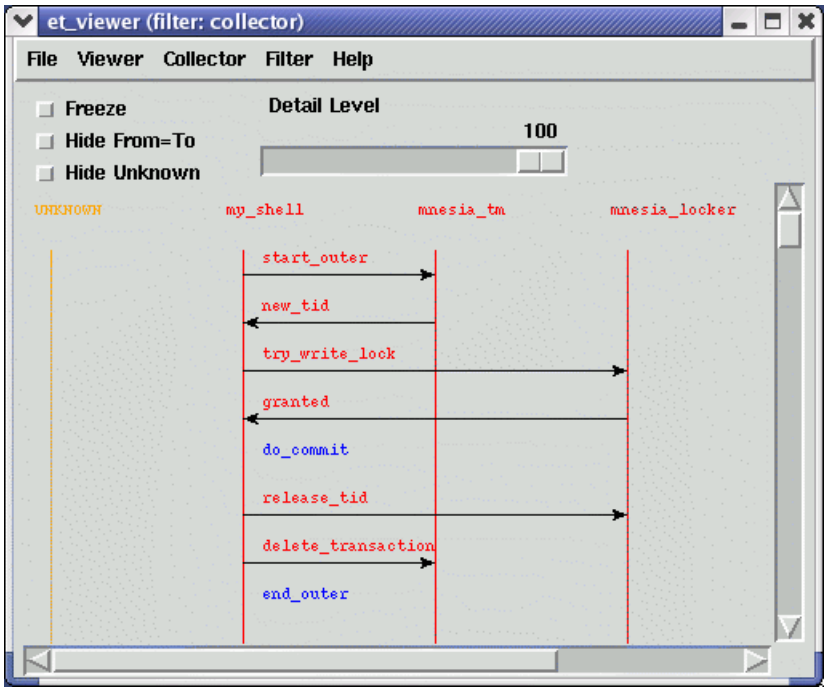

Figure 3.1: A simulated Mnesia transaction which writes one record

1.3.2 Some convenient functions used in the Mnesia transaction example

The `module_as_actor` filter converts the event-records so the module names becomes actors and the invoked functions becomes labels. If the information about who the caller was it will be displayed as an arrow directed from the caller to the callee. The `[{message, {caller}}, {return_trace}]` options to `dbg:tpl/2` function will imply the necessary information in the Erlang traces. Here follows the `module_as_actor` filter:

The `plain_process_info` filter does not alter the event-records. It merely ensures that the event not related to processes are skipped:

The `plain_process_info_nolink` filter does not alter the event-records. It do makes use of the `plain_process_info` , but do also ensure that the process info related to linking and unlinking is skipped:

In order to simplify the startup of an `et_viewer` process with the filters mentioned above, plus some others (that also are found in `et/examples/et_demo.erl` src/et_collector.erl the `et_demo:start/0,1` functions can be used:

A simple one-liner starts the tool:

```
erl -pa ../examples -s et_demo
```

The filters are included by the following parameters:

1.3.3 Erlang trace of a Mnesia transaction

The following piece of code `et_demo:trace_mnesia/0` activates call tracing of both local and external function calls for all modules in the Mnesia application. The call traces are configured cover all processes (both existing and those that are spawned in the future) and include timestamps for trace data. It do also activate tracing of process related events for Mnesia's static processes plus the calling process (that is your shell). Please, observe that the `whereis/1` call in the following code requires that both the traced Mnesia application and the `et_viewer`is running on the same node:

The `et_demo:live_trans/0` function starts the a global controller, starts a viewer, starts Mnesia, creates a local table, activates tracing (as described above) and registers the shell process is as 'my_shell' for clarity. Finally the a simple Mnesia transaction that writes a single record is run:

Now we run the `et_demo:live_trans/0` function:

```
erl -pa ../examples -s et_demo live_trans
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
1>
```

Please, explore the different filters in order to see how the traced transaction can be seen from different point of views:

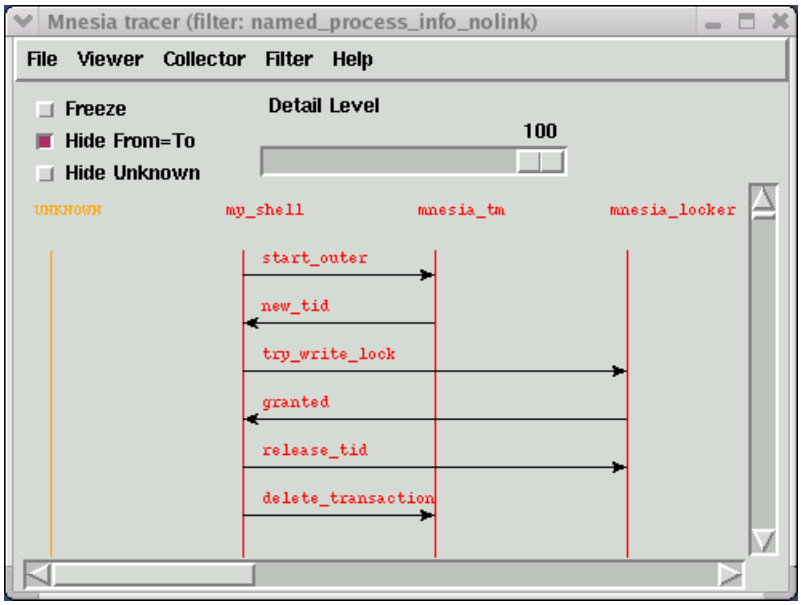

Figure 3.2: A real Mnesia transaction which writes one record

1.3.4 Erlang trace of Megaco startup

The Event Tracer (ET) tool was initially written in order to demonstrate how messages where sent over the Megaco protocol. This were back in the old days before the standard bodies of IETF and ITU had approved Megaco (also called H.248) as an international standard.

In the Megaco application of Erlang/OTP, the code is carefully instrumented with calls to `et:report_event/5`. For call a detail level is set in order to dynamically control the trace level in a simple manner.

The `megaco_filter` module implements a customized filter for Megaco messages. It does also make use of `trace_global` combined with usage of the `trace_pattern`:

```
-module(megaco_filter).
-export([start/0]).

start() ->
    Options =
        [{event_order, event_ts},
         {scale, 3},
         {max_actors, infinity},
         {trace_pattern, {megaco, max}},
         {trace_global, true},
         {dict_insert, {filter, megaco_filter}, fun filter/1},
         {active_filter, megaco_filter},
         {title, "Megaco tracer - Erlang/OTP"}],
    et_viewer:start(Options).
```

First we start an Erlang node with the a global collector and its viewer. The `et_viewer: search for: [] ++ ["gateway_tt"]` printout is caused by a click on the "gateway_tt" actor name in the viewer. It means that only events with that actor will be displayed in the viewer.

```
erl -sname observer -s megaco_filter
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
(observer@amrod)1> et_viewer: search for: [] ++ ["gateway_tt"]
```

Secondly we start another Erlang node which we connect the observer node, before we start the application that we want to trace. In this case we start a Media Gateway Controller that listens for both TCP and UDP on the text and binary ports for Megaco:

```
erl -sname mgc -pa ../../megaco/examples/simple
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
(mgc@amrod)1> net:ping(observer@amrod).
pong
(mgc@amrod)2> megaco:start().
ok
(mgc@amrod)3> megaco_simple_mgc:start().
{ok,[{ok,2944,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_pretty_text_encoder,
                                [],
                                megaco_tcp}},
     {ok,2944,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_pretty_text_encoder,
                                [],
```

```
                                megaco_udp}},
        {ok,2945,
            {megaco_receive_handle,{deviceName,"controller"},
                                megaco_binary_encoder,
                                [],
                                megaco_tcp}},
        {ok,2945,
            {megaco_receive_handle,{deviceName,"controller"},
                                megaco_binary_encoder,
                                [],
                                megaco_udp}}]}
(mgc@amrod)4>
```

And finally we start an Erlang node for the Media Gateways and connect to the observer node. Each Media Gateway connects to the controller and sends an initial Service Change message. The controller accepts the gateways and sends a reply to each one using the same transport mechanism and message encoding according to the preference of each gateway. That is all combinations of TCP/IP transport, UDP/IP transport, text encoding and ASN.1 BER encoding:

```
erl -sname mg -pa ../../megaco/examples/simple
Erlang (BEAM) emulator version 2002.10.08 [source]

Eshell V2002.10.08  (abort with ^G)
(mg@amrod)1> net:ping(observer@amrod).
pong
(mg@amrod)2> megaco_simple_mg:start().
[{{deviceName,"gateway_tt"},{error,{start_user,megaco_not_started}}},
 {{deviceName,"gateway_tb"},{error,{start_user,megaco_not_started}}},
 {{deviceName,"gateway_ut"},{error,{start_user,megaco_not_started}}},
 {{deviceName,"gateway_ub"},{error,{start_user,megaco_not_started}}}]
(mg@amrod)3> megaco:start().
ok
(mg@amrod)4>  megaco_simple_mg:start().
[{{deviceName,"gateway_tt"},
  {1,
   {ok,[{'ActionReply',0,
                       asn1_NOVALUE,
                       asn1_NOVALUE,
                       [{serviceChangeReply,
                            {'ServiceChangeReply',
                                [{megaco_term_id,false,["root"]}],
                                {serviceChangeResParms,
                                    {'ServiceChangeResParm',
                                        {deviceName|...},
                                        asn1_NOVALUE|...}}}}]}]}}},
 {{deviceName,"gateway_tb"},
  {1,
   {ok,[{'ActionReply',0,
                       asn1_NOVALUE,
                       asn1_NOVALUE,
                       [{serviceChangeReply,
                            {'ServiceChangeReply',
                                [{megaco_term_id,false,["root"]}],
                                {serviceChangeResParms,
                                    {'ServiceChangeResParm',
                                        {...}|...}}}}]}]}}},
 {{deviceName,"gateway_ut"},
  {1,
   {ok,[{'ActionReply',0,
                       asn1_NOVALUE,
                       asn1_NOVALUE,
```

```
                                [{serviceChangeReply,
                                     {'ServiceChangeReply',
                                         [{megaco_term_id,false,["root"]}],
                                         {serviceChangeResParms,
                                             {'ServiceChangeResParm',{...}|...}}}}]}]}}},
          {{deviceName,"gateway_ub"},
           {1,
            {ok,[{'ActionReply',0,
                                asn1_NOVALUE,
                                asn1_NOVALUE,
                                [{serviceChangeReply,
                                     {'ServiceChangeReply',
                                         [{megaco_term_id,false,["root"]}],
                                         {serviceChangeResParms,
                                             {'ServiceChangeResParm'|...}}}}]}]}}]
          (mg@amrod)5>
```

The Megaco adopted viewer looks like this, when we have clicked on the "gateway_tt" actor name in order to only display the events regarding that actor:

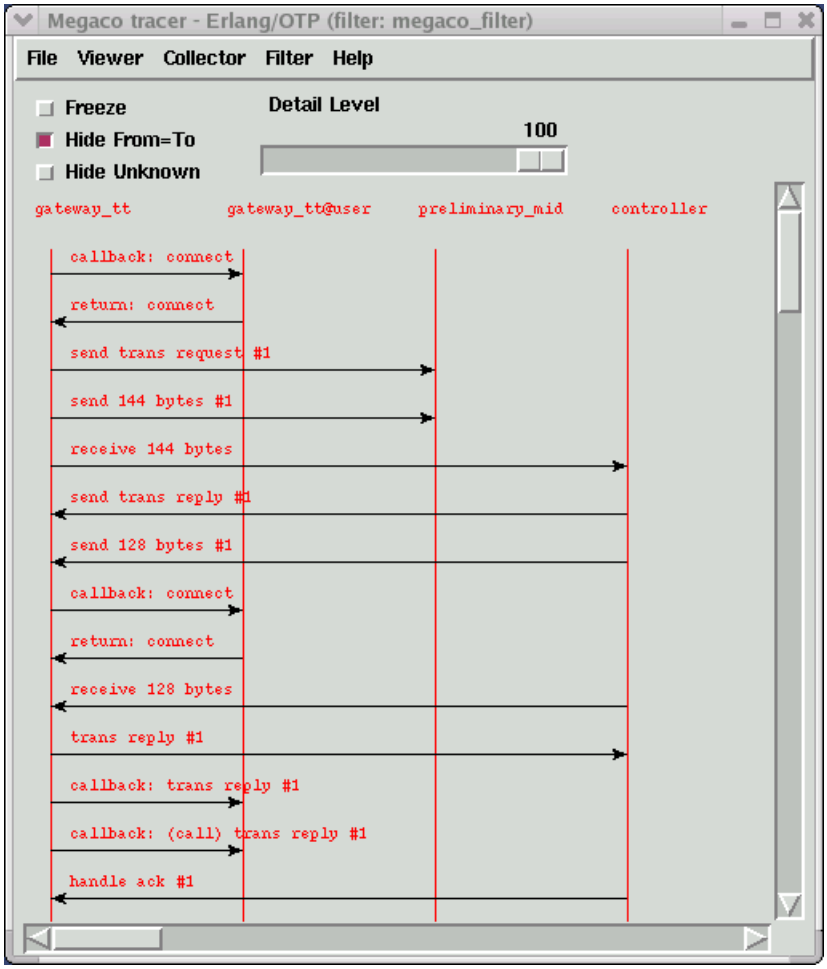

Figure 3.3: The viewer adopted for Megaco

A pretty printed Megaco message looks like this:

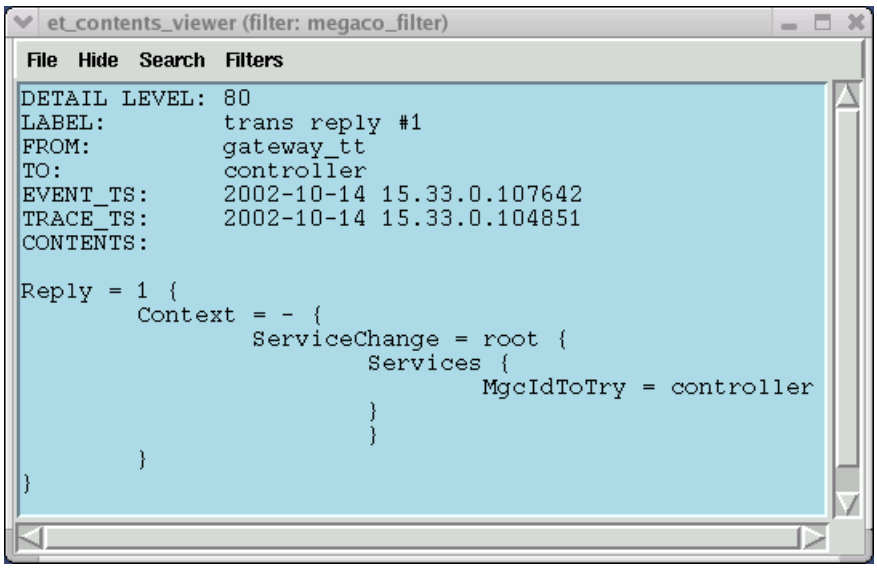

Figure 3.4: A textual Megaco message

And the corresponding internal form for the same Megaco message looks like this:

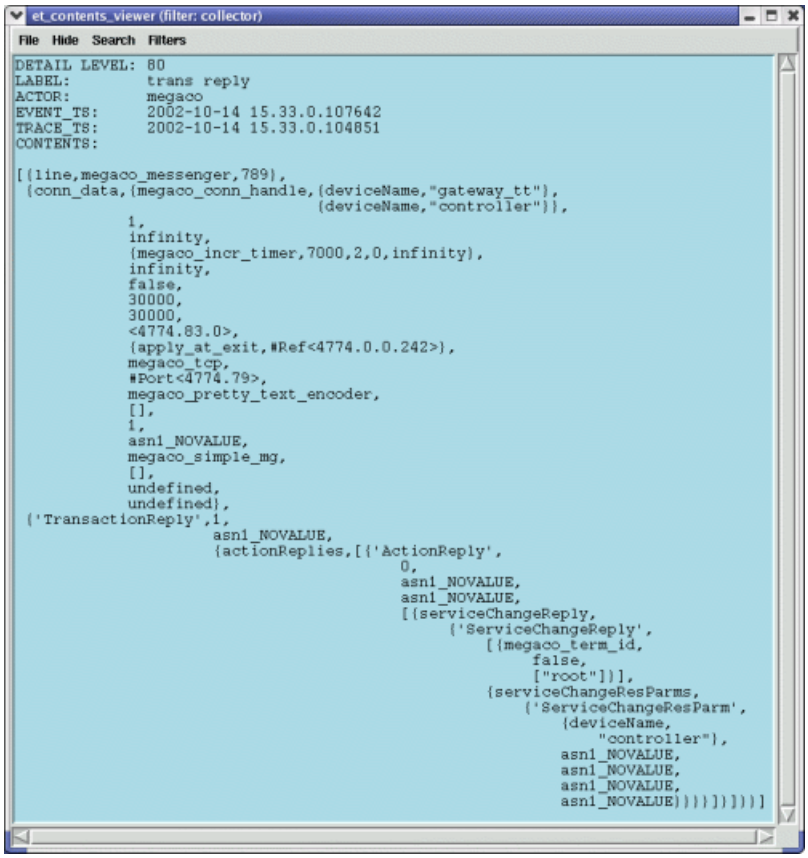

Figure 3.5: The internal form of a Megaco message

2 Reference Manual

The Event Tracer (ET) uses the built-in trace mechanism in Erlang and provides tools for collection and graphical viewing of trace data.

et

Erlang module

Interface module for the Event Trace (ET) application

Exports

```
report_event(DetailLevel, FromTo, Label, Contents) -> hopefully_traced
report_event(DetailLevel, From, To, Label, Contents) -> hopefully_traced
phone_home(DetailLevel, FromTo, Label, Contents) -> hopefully_traced
phone_home(DetailLevel, From, To, Label, Contents) -> hopefully_traced
```

Types:

DetailLevel = integer(X) when X =< 0, X >= 100

From = actor()

To = actor()

FromTo = actor()

Label = atom() | string() | term()

Contents = [{Key, Value}] | term()

actor() = term()

Reports an event, such as a message.

These functions are intended to be invoked at strategic places in user applications in order to enable simplified tracing. The functions are extremely light weight as they do nothing besides returning an atom. These functions are designed for being traced. The global tracing mechanism in et_collector defaults to set its trace pattern to these functions.

The label is intended to provide a brief summary of the event. A simple tag would do.

The contents can be any term but in order to simplify post processing of the traced events, a plain list of {Key, Value} tuples is preferred.

Some events, such as messages, are directed from some actor to another. Other events (termed actions) may be undirected and only have one actor.

et_collector

Erlang module

Interface module for the Event Trace (ET) application

Exports

`start_link(Options) -> {ok, CollectorPid} | {error, Reason}`

Types:

Options = [option()]

option() = {parent_pid, pid()} | {event_order, event_order()} | {dict_insert, {filter, collector}, collector_fun()} | {dict_insert, {filter, event_filter_name()}, event_filter_fun()} | {dict_insert, {subscriber, pid()}, dict_val()} | {dict_insert, dict_key(), dict_val()} | {dict_delete, dict_key()} | {trace_client, trace_client()} | {trace_global, boolean()} | {trace_pattern, trace_pattern()} | {trace_port, integer()} | {trace_max_queue, integer()}

event_order() = trace_ts | event_ts

trace_pattern() = {report_module(), extended_dbg_match_spec()} | undefined

report_module() = atom() | undefined <v>extended_dbg_match_spec()() = detail_level() | dbg_match_spec()

detail_level() = min | max | integer(X) when X =< 0, X >= 100

trace_client() = {event_file, file_name()} | {dbg_trace_type(), dbg_trace_parameters()}

file_name() = string()

collector_fun() = trace_filter_fun() | event_filter_fun()

trace_filter_fun() = fun(TraceData) -> false | true | {true, NewEvent}

event_filter_fun() = fun(Event) -> false | true | {true, NewEvent}

event_filter_name() = atom()

TraceData = erlang_trace_data()

Event = NewEvent = record(event)

dict_key() = term()

dict_val() = term()

CollectorPid = pid()

Reason = term()

Start a collector process.

The collector collects trace events and keeps them ordered by their timestamp. The timestamp may either reflect the time when the actual trace data was generated (trace_ts) or when the trace data was transformed into an event record (event_ts). If the time stamp is missing in the trace data (missing timestamp option to erlang:trace/4) the trace_ts will be set to the event_ts.

Events are reported to the collector directly with the report function or indirectly via one or more trace clients. All reported events are first filtered thru the collector filter before they are stored by the collector. By replacing the default collector filter with a customized dito it is possible to allow any trace data as input. The collector filter is a dictionary entry with the predefined key {filter, collector} and the value is a fun of arity 1. See et_selector:make_event/1 for interface details, such as which erlang:trace/1 tuples that are accepted.

The collector has a built-in dictionary service. Any term may be stored as value in the dictionary and bound to a unique key. When new values are inserted with an existing key, the new values will overwrite the existing ones. Processes

may subscribe on dictionary updates by using {subscriber, pid()} as dictionary key. All dictionary updates will be propagated to the subscriber processes matching the pattern {{subscriber, '_'}, '_'} where the first '_' is interpreted as a pid().

In global trace mode, the collector will automatically start tracing on all connected Erlang nodes. When a node connects, a port tracer will be started on that node and a corresponding trace client on the collector node. By default the global trace pattern is 'max'.

Default values:

- parent_pid - self().
- event_order - trace_ts.
- trace_global - false.
- trace_pattern - undefined.
- trace_port - 4711.
- trace_max_queue - 50.

`stop(CollectorPid) -> ok`

Types:

CollectorPid = pid()

Stop a collector process.

`save_event_file(CollectorPid, FileName, Options) -> ok | {error, Reason}`

Types:

CollectorPid = pid()
FileName = string()
Options = [option()]
Reason = term()
option() = event_option() | file_option() | table_option()
event_option() = existing
file_option() = write | append
table_option() = keep | clear

Save the events to a file.

By default the currently stored events (existing) are written to a brand new file (write) and the events are kept (keep) after they have been written to the file.

Instead of keeping the events after writing them to file, it is possible to remove all stored events after they have successfully written to file (clear).

The options defaults to existing, write and keep.

`load_event_file(CollectorPid, FileName) -> {ok, BadBytes} | exit(Reason)`

Types:

CollectorPid = pid()
FileName = string()
BadBytes = integer(X) where X >= 0
Reason = term()

Load the event table from a file.

```
report(Handle, TraceOrEvent) -> {ok, Continuation} | exit(Reason)
report_event(Handle, DetailLevel, FromTo, Label, Contents) -> {ok,
Continuation} | exit(Reason)
report_event(Handle, DetailLevel, From, To, Label, Contents) -> {ok,
Continuation} | exit(Reason)
```

Types:

Handle = Initial | Continuation

Initial = collector_pid()

collector_pid() = pid()

Continuation = record(table_handle)

TraceOrEvent = record(event) | dbg_trace_tuple() | end_of_trace

Reason = term()

DetailLevel = integer(X) when X =< 0, X >= 100

From = actor()

To = actor()

FromTo = actor()

Label = atom() | string() | term()

Contents = [{Key, Value}] | term()

actor() = term()

Report an event to the collector.

All events are filtered thru the collector filter, which optionally may transform or discard the event. The first call should use the pid of the collector process as report handle, while subsequent calls should use the table handle.

```
make_key(Type, Stuff) -> Key
```

Types:

Type = record(table_handle) | trace_ts | event_ts

Stuff = record(event) | Key

Key = record(event_ts) | record(trace_ts)

Make a key out of an event record or an old key.

```
get_table_handle(CollectorPid) -> Handle
```

Types:

CollectorPid = pid()

Handle = record(table_handle)

Return a table handle.

```
get_global_pid() -> CollectorPid | exit(Reason)
```

Types:

CollectorPid = pid()

Reason = term()

Return a the identity of the globally registered collector if there is any.

```
change_pattern(CollectorPid, RawPattern) -> {old_pattern, TracePattern}
```

Types:

CollectorPid = pid()

RawPattern = {report_module(), extended_dbg_match_spec()}

report_module() = atom() | undefined

extended_dbg_match_spec()() = detail_level() | dbg_match_spec()

RawPattern = detail_level()

detail_level() = min | max | integer(X) when X =< 0, X >= 100

TracePattern = {report_module(), dbg_match_spec_match_spec()}

Change active trace pattern globally on all trace nodes.

```
dict_insert(CollectorPid, {filter, collector}, FilterFun) -> ok
dict_insert(CollectorPid, {subscriber, SubscriberPid}, Void) -> ok
dict_insert(CollectorPid, Key, Val) -> ok
```

Types:

CollectorPid = pid()

FilterFun = filter_fun()

SubscriberPid = pid()

Void = term()

Key = term()

Val = term()

Insert a dictionary entry and send a {et, {dict_insert, Key, Val}} tuple to all registered subscribers.

If the entry is a new subscriber, it will imply that the new subscriber process first will get one message for each already stored dictionary entry, before it and all old subscribers will get this particular entry. The collector process links to and then supervises the subscriber process. If the subscriber process dies it will imply that it gets unregistered as with a normal dict_delete/2.

```
dict_lookup(CollectorPid, Key) -> [Val]
```

Types:

CollectorPid = pid()

FilterFun = filter_fun()

CollectorPid = pid()

Key = term()

Val = term()

Lookup a dictionary entry and return zero or one value.

```
dict_delete(CollectorPid, Key) -> ok
```

Types:

CollectorPid = pid()

SubscriberPid = pid()

Key = {subscriber, SubscriberPid} | term()

Delete a dictionary entry and send a {et, {dict_delete, Key}} tuple to all registered subscribers.

If the deleted entry is a registered subscriber, it will imply that the subscriber process gets is unregistered as subscriber as well as it gets it final message.

`dict_match(CollectorPid, Pattern) -> [Match]`

Types:

CollectorPid = pid()

Pattern = '_' | {key_pattern(), val_pattern()}

key_pattern() = ets_match_object_pattern()

val_pattern() = ets_match_object_pattern()

Match = {key(), val()}

key() = term()

val() = term()

Match some dictionary entries

`multicast(_CollectorPid, Msg) -> ok`

Types:

CollectorPid = pid()

CollectorPid = pid()

Msg = term()

Sends a message to all registered subscribers.

`start_trace_client(CollectorPid, Type, Parameters) -> file_loaded | {trace_client_pid, pid()} | exit(Reason)`

Types:

Type = dbg_trace_client_type()

Parameters = dbg_trace_client_parameters()

Pid = dbg_trace_client_pid()

Load raw Erlang trace from a file, port or process.

`iterate(Handle, Prev, Limit) -> NewAcc`

Short for iterate(Handle, Prev, Limit, undefined, Prev) -> NewAcc

`iterate(Handle, Prev, Limit, Fun, Acc) -> NewAcc`

Types:

Handle = collector_pid() | table_handle()

Prev = first | last | event_key()

Limit = done() | forward() | backward()

collector_pid() = pid()

table_handle() = record(table_handle)

event_key() = record(event) | record(event_ts) | record(trace_ts)

done() = 0

forward() = infinity | integer(X) where X > 0

backward() = '-infinity' | integer(X) where X < 0

Fun = fun(Event, Acc) -> NewAcc <v>Acc = NewAcc = term()

Iterate over the currently stored events.

Iterates over the currently stored eventsand applies a function for each event. The iteration may be performed forwards or backwards and may be limited to a maximum number of events (abs(Limit)).

`clear_table(Handle) -> ok`

Types:

Handle = collector_pid() | table_handle()

collector_pid() = pid()

table_handle() = record(table_handle)

Clear the event table.

et_selector

Erlang module

Exports

`make_pattern(RawPattern) -> TracePattern`

Types:

RawPattern = detail_level()

TracePattern = erlang_trace_pattern_match_spec()

detail_level() = min | max | integer(X) when X =< 0, X >= 100

Makes a trace pattern suitable to feed change_pattern/1

Min detail level deactivates tracing of calls to phone_home/4,5

Max detail level activates tracing of all calls to phone_home/4,5

integer(X) detail level activates tracing of all calls to phone_home/4,5 whose detail level argument is lesser than X.

See also erlang:trace_pattern/2 for more info about its match_spec()

`change_pattern(Pattern) -> ok`

Types:

Pattern = detail_level() | empty_match_spec() | erlang_trace_pattern_match_spec()

detail_level() = min | max | integer(X) when X =<0, X >= 100

empty_match_spec() = []

Activates/deactivates tracing by changing the current trace pattern.

Min detail level deactivates tracing of calls to phone_home/4,5

Max detail level activates tracing of all calls to phone_home/4,5

integer(X) detail level activates tracing of all calls to phone_home/4,5 whose detail level argument is lesser than X.

An empty match spec deactivates tracing of calls to phone_home/4,5

Other match specs activates tracing of calls to phone_home/4,5 accordingly with erlang:trace_pattern/2.

`parse_event(Mod, ValidTraceData) -> false | true | {true, Event}`

Types:

Mod = module_name() | undefined <v>module_name() = atom() <v>ValidTraceData = erlang_trace_data() | record(event)

erlang_trace_data() = {trace, Pid, Label, Info} | {trace, Pid, Label, Info, Extra} | {trace_ts, Pid, Label, Info, ReportedTS} | {trace_ts, Pid, Label, Info, Extra, ReportedTS} | {seq_trace, Label, Info} | {seq_trace, Label, Info, ReportedTS} | {drop, NumberOfDroppedItems}

Transforms trace data and makes an event record out of it.

See erlang:trace/3 for more info about the semantics of the trace data.

An event record consists of the following fields: detail_level - Noise has a high level as opposed to essentials. trace_ts - Time when the trace was generated. Same as event_ts if omitted in trace data. event_ts - Time when the event record

was created. from - From actor, such as sender of a message. to - To actor, such as receiver of message. label - Label intended to provide a brief event summary. contents - All nitty gritty details of the event.

See et:phone_home/4 and et:phone_home/5 for details.

Returns: {true, Event} - where Event is an #event{} record representing the trace data true - means that the trace data already is an event record and that it is valid as it is. No transformation is needed. false - means that the trace data is uninteresting and should be dropped

et_viewer

Erlang module

Exports

`file(FileName) -> {ok, ViewerPid} | {error, Reason}`

Types:

FileName() = string()

ViewerPid = pid()

Reason = term()

Start a new event viewer and a corresponding collector and load them with trace events from a trace file.

`start() -> ok`

Simplified start of a sequence chart viewer with global tracing activated.

Convenient to be used from the command line (erl -s et_viewer).

`start(Options) -> ok`

Start of a sequence chart viewer without linking to the parent process.

`start_link(Options) -> {ok, ViewerPid} | {error, Reason}`

Types:

Options = [option() | collector_option()]

option() = {parent_pid, extended_pid()} | {title, term()} | {detail_level, detail_level()} | {is_suspended, boolean()} | {scale, integer()} | {width, integer()} | {height, integer()} | {collector_pid, extended_pid()} | {event_order, event_order()} | {active_filter, atom()} | {max_events, extended_integer()} | {max_actors, extended_integer()} | {trace_pattern, et_collector_trace_pattern()} | {trace_port, et_collector_trace_port()} | {trace_global, et_collector_trace_global()} | {trace_client, et_collector_trace_client()} | {dict_insert, {filter, filter_name()}, event_filter_fun()} | {dict_insert, et_collector_dict_key(), et_collector_dict_val()} | {dict_delete, {filter, filter_name()}} | {dict_delete, et_collector_dict_key()} | {actors, actors()} | {first_event, first_key()} | {hide_unknown, boolean()} | {hide_actions, boolean()} | {display_mode, display_mode()}

extended_pid() = pid() | undefined

detail_level() = min | max | integer(X) when X >=0, X =< 100

event_order() = trace_ts | event_ts

extended_integer() = integer() | infinity

display_mode() = all | {search_actors, direction(), first_key(), actors()}

direction() = forward | reverse

first_key() = event_key()

actors() = [term()]

filter_name() = atom()

filter_fun() = fun(Event) -> false | true | {true, NewEvent}

Event = NewEvent = record(event)

ViewerPid = pid()

Reason = term()

Start a sequence chart viewer for trace events (messages/actions)

A filter_fun() takes an event record as sole argument and returns false | true | {true, NewEvent}.

If the `collector_pid` is `undefined` a new `et_collector` will be started with the following parameter settings: `parent_pid`, `event_order`, `trace_global`, `trace_pattern`, `trace_port`, `trace_max_queue`, `trace_client`, `dict_insert` and `dict_delete`. The new `et_viewer` will register itself as an `et_collector` subscriber.

Default values:

- parent_pid - self().
- title - "et_viewer".
- detail_level - max.
- is_suspended - false.
- scale - 2.
- width - 800.
- height - 600.
- collector_pid - undefined.
- event_order - trace_ts.
- active_filter - collector.
- max_events - 100.
- max_actors - 5.
- actors - ["UNKNOWN"].
- first_event - first.
- hide_unknown - false.
- hide_actions - false.
- display_mode - all.

`get_collector_pid(ViewerPid) -> CollectorPid`

Types:

ViewerPid = pid()

CollectorPid = pid()

Returns the identifier of the collector process.

`stop(ViewerPid) -> ok`

Types:

ViewerPid = pid()

Stops a viewer process.