


# Event Tracer (ET)

Copyright © 2002-2022 Ericsson AB. All Rights Reserved.
Event Tracer (ET) 1.6.5
August 18, 2022

**Copyright © 2002-2022 Ericsson AB. All Rights Reserved.**
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

**August 18, 2022**

# 1 Event Tracer (ET) Users 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 R13BB or later. If you use the old GS based GUI it does suffice with R7B.

### 1.1.3 About This Manual

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

- Chapter 2: "Tutorial" provides a walk-through of the various parts of the application. The tutorial is based on Jayson Vantuyl's article http://souja.net/2009/04/making-sense-of-erlangs-event-tracer.html.
- Chapter 3: "Description" describes the architecture and typical usage of the application.
- Chapter 4: "Advanced 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, invisio and et.
- Programming Erlang: Software for a Concurrent World by Joe Armstrong; ISBN: 978-1-93435-600-5

# 1.2 Tutorial

## 1.2.1 Visualizing Message Sequence Charts

The easiest way of using ET, is to just use it as a graphical tool for displaying message sequence charts. In order to do that you need to first start a `Viewer` (which by default starts a `Collector`):

```
      {ok, ViewerPid} = et_viewer:start([{title,"Coffee Order"}]),
      CollectorPid = et_viewer:get_collector_pid(ViewerPid).
```

Then you send events to the `Collector` with the function `et_collector:report_event/6` like this:

```
      et_collector:report_event(CollectorPid,85,from,to,message,extra_stuff).
```

The `Viewer` will automatically pull events from the `Collector` and display them on the screen.

The number (in this case 85) is an integer from 1 to 100 that specifies the "detail level" of the message. The higher the number, the more important it is. This provides a crude form of priority filtering.

The `from`, `to`, and `message` parameters are exactly what they sound like. `from` and `to` are visualized in the `Viewer` as "lifelines", with the message passing from one to the other. If `from` and `to` are the same value, then it is displayed next to the lifeline as an "action". The `extra_stuff` value is simply data that you can attach that will be displayed when someone actually clicks on the action or message in the `Viewer` window.

The module `et/examples/et_display_demo.erl` illustrates how it can be used:

```
-module(et_display_demo).

-export([test/0]).

test() ->
    {ok, Viewer} = et_viewer:start([{title,"Coffee Order"}, {max_actors,10}]),
    Drink = {drink,iced_chai_latte},
    Size = {size,grande},
    Milk = {milk,whole},
    Flavor = {flavor,vanilla},
    C = et_viewer:get_collector_pid(Viewer),
    et_collector:report_event(C,99,customer,barrista1,place_order,[Drink,Size,Milk,Flavor]),
    et_collector:report_event(C,80,barrista1,register,enter_order,[Drink,Size,Flavor]),
    et_collector:report_event(C,80,register,barrista1,give_total,"$5"),
    et_collector:report_event(C,80,barrista1,barrista1,get_cup,[Drink,Size]),
    et_collector:report_event(C,80,barrista1,barrista2,give_cup,[]),
    et_collector:report_event(C,90,barrista1,customer,request_money,"$5"),
    et_collector:report_event(C,90,customer,barrista1,pay_money,"$5"),
    et_collector:report_event(C,80,barrista2,barrista2,get_chai_mix,[]),
    et_collector:report_event(C,80,barrista2,barrista2,add_flavor,[Flavor]),
    et_collector:report_event(C,80,barrista2,barrista2,add_milk,[Milk]),
    et_collector:report_event(C,80,barrista2,barrista2,add_ice,[]),
    et_collector:report_event(C,80,barrista2,barrista2,swirl,[]),
    et_collector:report_event(C,80,barrista2,customer,give_tasty_beverage,[Drink,Size]),
    ok.
```

When you run the `et_display_demo:test().` function in the example above, the `Viewer` window will look like this:

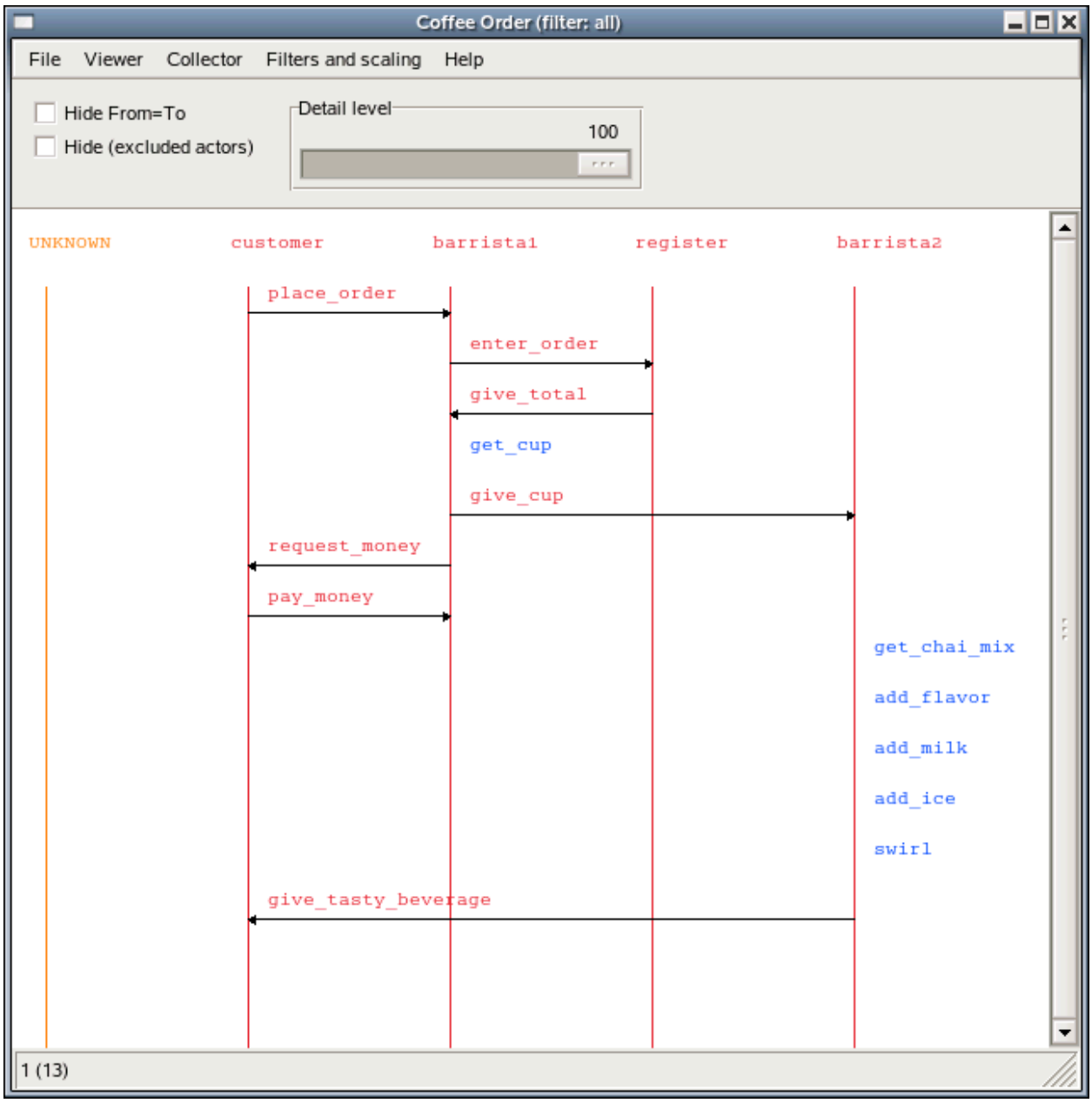



Figure 2.1: Screenshot of the Viewer window

## 1.2.2 Four Modules

The event tracer framework is made up of four modules:

- `et`
- `et_collector`
- `et_viewer`
- `et_selector`

In addition, you'll probably want to familiarize yourself with the `dbg` module and possibly `seq_trace` module as well.

### 1.2.3 The Event Tracer Interface

The `et` module is not like other modules. It contains a function called `et:trace_me/5`. Which is a function that does not do any useful stuff at all. Its sole purpose is to be a function that is easy to trace. A call to it may be something like:

```
et:trace_me(85,from,to,message,extra_stuff).
```

The parameters to `et:trace_me/5` are the same as to `et_collector:report_event/6` in the previous chapter. The big difference between the two is in the semantics of the two functions. The second actually reports an `Event` to the `Collector` while the first does nothing, it just returns the atom `hopefully_traced`. In order to make the parameters to `et:trace_me/5` turn up in the `Collector`, tracing of that function must be activated and the `Collector` must be registered as a `Tracer` of the `Raw Trace Data`.

Erlang tracing is a seething pile of pain that involves reasonably complex knowledge of clever ports, tracing return formats, and specialized tracing `MatchSpecs` (which are really their own special kind of hell). The tracing mechanism is very powerful indeed, but it can be hard to grasp.

Luckily there is a simplified way to start tracing of `et:trace_me/5` function calls. The idea is that you should instrument your code with calls to `et:trace_me/5` in strategic places where you have interesting information available in your program. Then you just start the `Collector` with global tracing enabled:

```
et_viewer:start([{trace_global, true}, {trace_pattern, {et,max}}]).
```

This will start a `Collector`, a `Viewer` and also start the tracing of `et:trace_me/5` function calls. The `Raw Trace Data` is collected by the `Collector` and a view of it is displayed on the screen by the `Viewer`. You can define your own "views" of the data by implementing your own `Filter` functions and register them in the `Viewer`.

### 1.2.4 The Collector and Viewer

These two pieces work in concert. Basically, the `Collector` receives `Raw Trace Data` and processes it into `Events` in a `et` specific format (defined in `et/include/et.hrl`). The `Viewer` interrogates the `Collector` and displays an interactive representation of the data.

You might wonder why these aren't just one module. The `Collector` is a generic full-fledged framework that allows processes to "subscribe" to the `Events` that it collects. One `Collector` can serve several `Viewers`. The typical case is that you have one `Viewer` that visualizes `Events` in one flavor and another `Viewer` that visualizes them in another flavor. If you for example are tracing a text based protocol like `HTML` (or `Megaco/H.248`) it would be useful to be able to display the `Events` as plain text as well as the internal representation of the message. The architecture does also allow you to implement your own `Viewer` program as long as it complies to the protocol between the `Collector/Viewer` protocol. Currently two kinds of `Viewers` exists. That is the old `GS` based one and the new based on `wxWidgets`. But if you feel for it you may implement your own `Viewer`, which for example could display the `Events` as ASCII art or whatever you feel useful.

The `Viewer` will by default create a `Collector` for you. With a few options and some configuration settings you can start collecting `Events`.

The `Collector` API does also allow you to save the collected `Events` to file and later load them in a later session.

### 1.2.5 The Selector

This is perhaps the most central module in the entirety of the `et` suite. The `Collector` needs "filters" to convert the `Raw Trace Data` into "events" that it can display. The `et_selector` module provides the default `Filter` and some API calls to manage the `Trace Pattern`. The `Selector` provides various functions that achieve the following:

- Convert `Raw Trace Data` into an appropriate `Event`
- Magically notice traces of the `et:trace_me/5` function and make appropriate `Events`

- Carefully prevent translating the `Raw Trace Data` twice
- Manage a `Trace Pattern`

The `Trace Pattern` is basically a tuple of a `module` and a `detail level` (either an integer or the atom max for full detail). In most cases the `Trace Pattern` `{et,max}` does suffice. But if you do not want any runtime dependency of `et` you can implement your own `trace_me/5` function in some module and refer to that module in the `Trace Pattern`.

The specified module flows from your instantiation of the `Viewer`, to the `Collector` that it automatically creates, gets stashed in as the `Trace Pattern`, and eventually goes down into the bowels of the `Selector`.

The module that you specify gets passed down (eventually) into `Selector`'s default `Filter`. The format of the `et:trace_me/5` function call is hardcoded in that `Filter`.

### 1.2.6 How To Put It Together

The `Collector` automatically registers itself to listen for trace `Events`, so all you have to do is enable them.

For those people who want to do general tracing, consult the `dbg` module on how to trace whatever you're interested in and let it work its magic. If you just want `et:trace_me/5` to work, do the following:

- Create a `Collector`
- Create a `Viewer` (this can do step #1 for you)
- Turn on and pare down debugging

The module `et/examples/et_trace_demo.erl` achieves this.

```
-module(et_trace_demo).

-export([test/0]).

test() ->
    et_viewer:start([
        {title,"Coffee Order"},
        {trace_global,true},
        {trace_pattern,{et,max}},
        {max_actors,10}
      ]),
      %% dbg:p(all,call),
      %% dbg:tpl(et, trace_me, 5, []),
      Drink = {drink,iced_chai_latte},
      Size = {size,grande},
      Milk = {milk,whole},
      Flavor = {flavor,vanilla},
      et:trace_me(99,customer,barrista1,place_order,[Drink,Size,Milk,Flavor]),
      et:trace_me(80,barrista1,register,enter_order,[Drink,Size,Flavor]),
      et:trace_me(80,register,barrista1,give_total,"$5"),
      et:trace_me(80,barrista1,barrista1,get_cup,[Drink,Size]),
      et:trace_me(80,barrista1,barrista2,give_cup,[]),
      et:trace_me(90,barrista1,customer,request_money,"$5"),
      et:trace_me(90,customer,barrista1,pay_money,"$5"),
      et:trace_me(80,barrista2,barrista2,get_chai_mix,[]),
      et:trace_me(80,barrista2,barrista2,add_flavor,[Flavor]),
      et:trace_me(80,barrista2,barrista2,add_milk,[Milk]),
      et:trace_me(80,barrista2,barrista2,add_ice,[]),
      et:trace_me(80,barrista2,barrista2,swirl,[]),
      et:trace_me(80,barrista2,customer,give_tasty_beverage,[Drink,Size]),
      ok.
```

Running through the above, the most important points are:

- Turn on global tracing

- Set a `Trace Pattern`
- Tell `dbg` to trace function Calls
- Tell it specifically to trace the `et:trace_me/5` function

When you run the `et_trace_demo:test()` function above, the `Viewer` window will look like this screenshot:

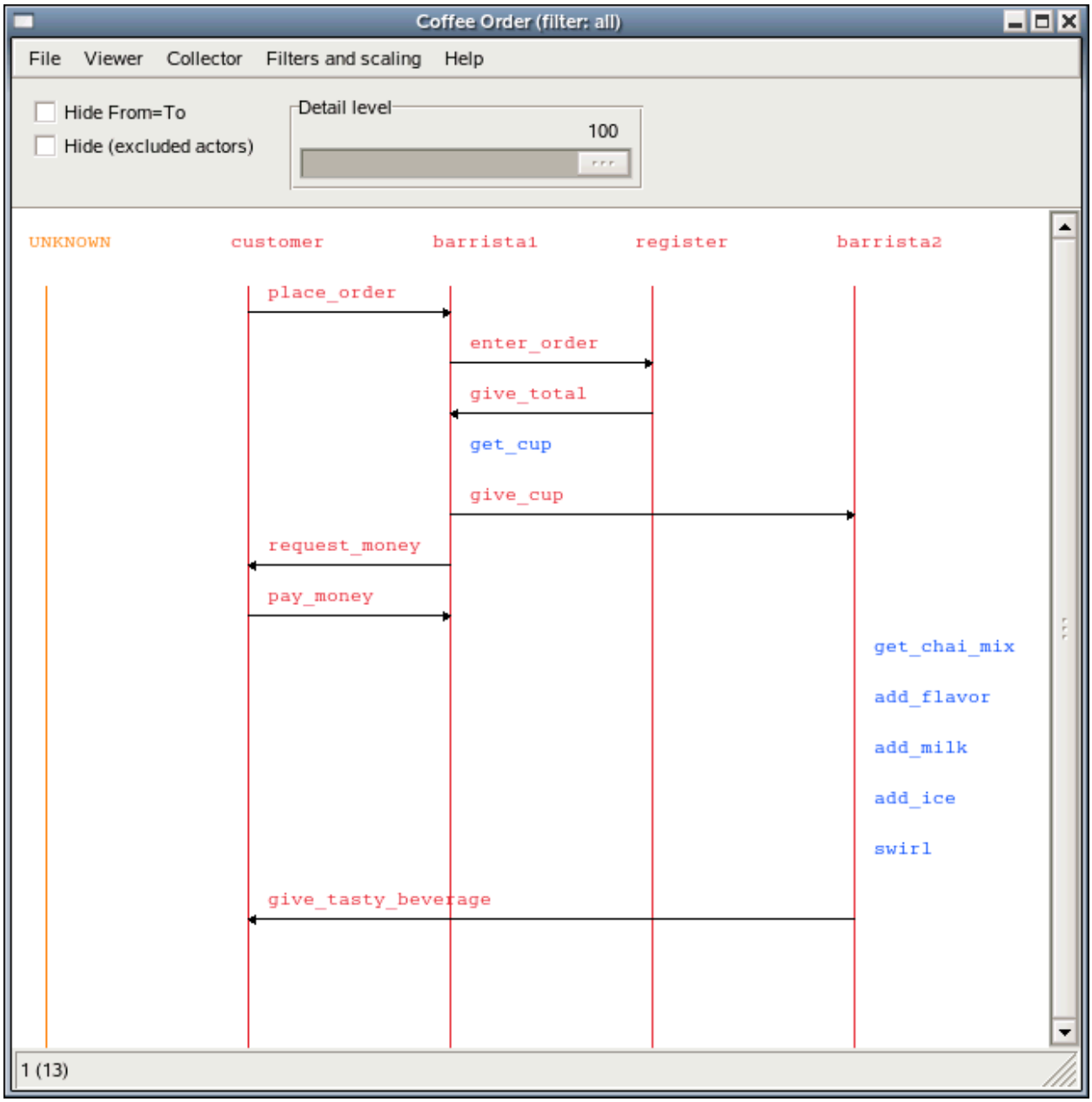



Figure 2.2: Screenshot of the Viewer window

# 1.3 Description

## 1.3.1 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. By default it will start both an et_collector and an et_viewer:

```
% erl -pa et/examples
Erlang R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
1> {ok, Viewer} = et_viewer:start([]).
{ok,<0.40.0>}
```

A Viewer gets trace Events from its Collector by polling it regularly for more Events to display. Events are for example reported to the Collector with et_collector:report_event/6:

```
2> Collector = et_viewer:get_collector_pid(Viewer).
<0.39.0>
3> et_collector:report_event(Collector, 60, my_shell, mnesia_tm, start_outer,
3>                           "Start outer transaction"),
3> et_collector:report_event(Collector, 40, mnesia_tm, my_shell, new_tid,
3>                           "New transaction id is 4711"),
3> et_collector:report_event(Collector, 20, my_shell, mnesia_locker, try_write_lock,
3>                           "Acquire write lock for {my_tab, key}"),
3> et_collector:report_event(Collector, 10, mnesia_locker, my_shell, granted,
3>                           "You got the write lock for {my_tab, key}"),
3> et_collector:report_event(Collector, 60, my_shell, do_commit,
3>                           "Perform  transaction commit"),
3> et_collector:report_event(Collector, 40, my_shell, mnesia_locker, release_tid,
3>                           "Release all locks for transaction 4711"),
3> et_collector:report_event(Collector, 60, my_shell, mnesia_tm, delete_transaction,
3>                           "End of outer transaction"),
3> et_collector:report_event(Collector, 20, my_shell, end_outer,
3>                           "Transaction returned {atomic, ok}").
{ok,{table_handle,<0.39.0>,16402,trace_ts,
     #Fun<et_collector.0.62831470>}}
```

This actually 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:

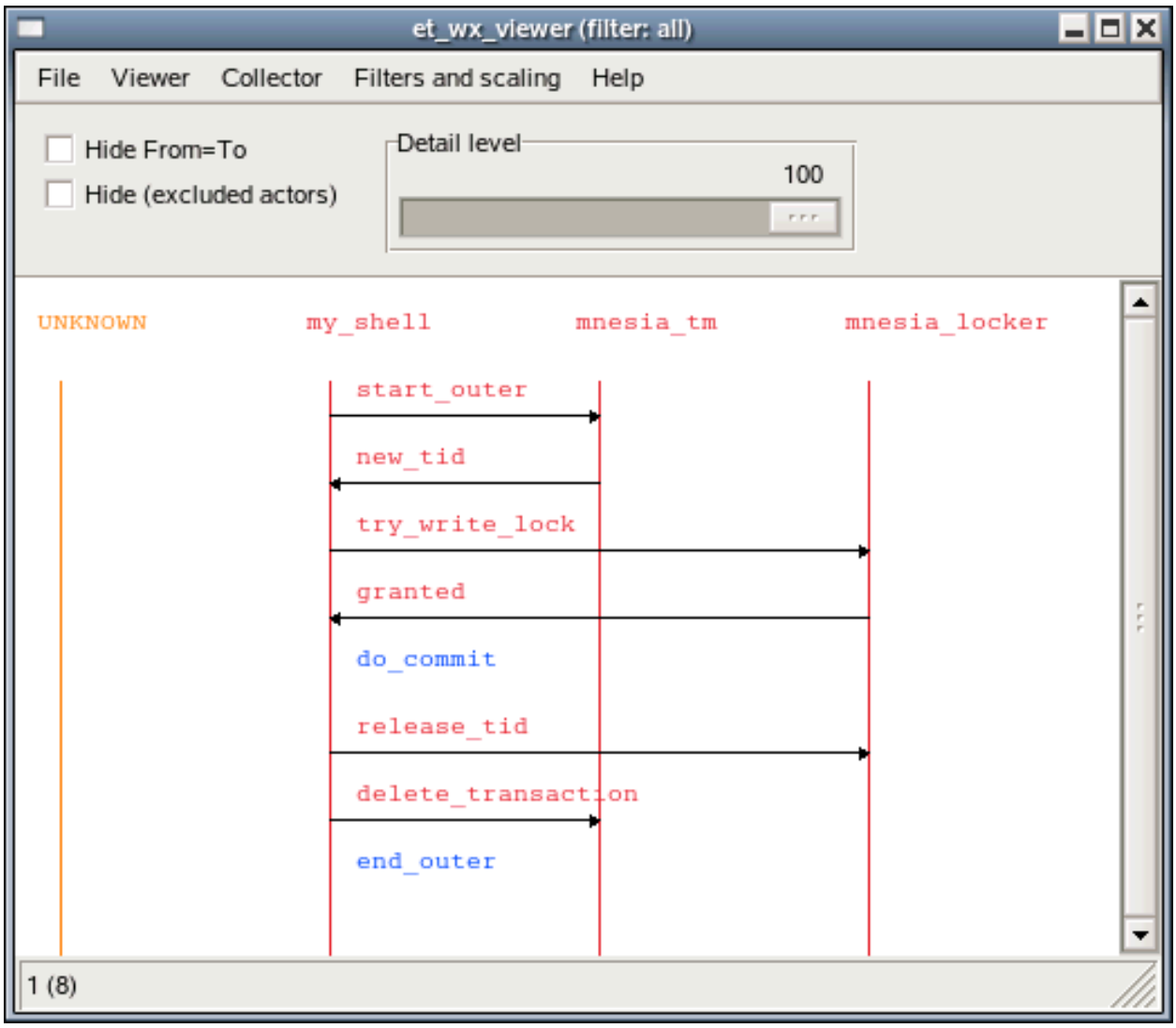



Figure 3.1: A simulated Mnesia transaction which writes one record

In the sequence chart, the actors (which symbolically has performed the `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:

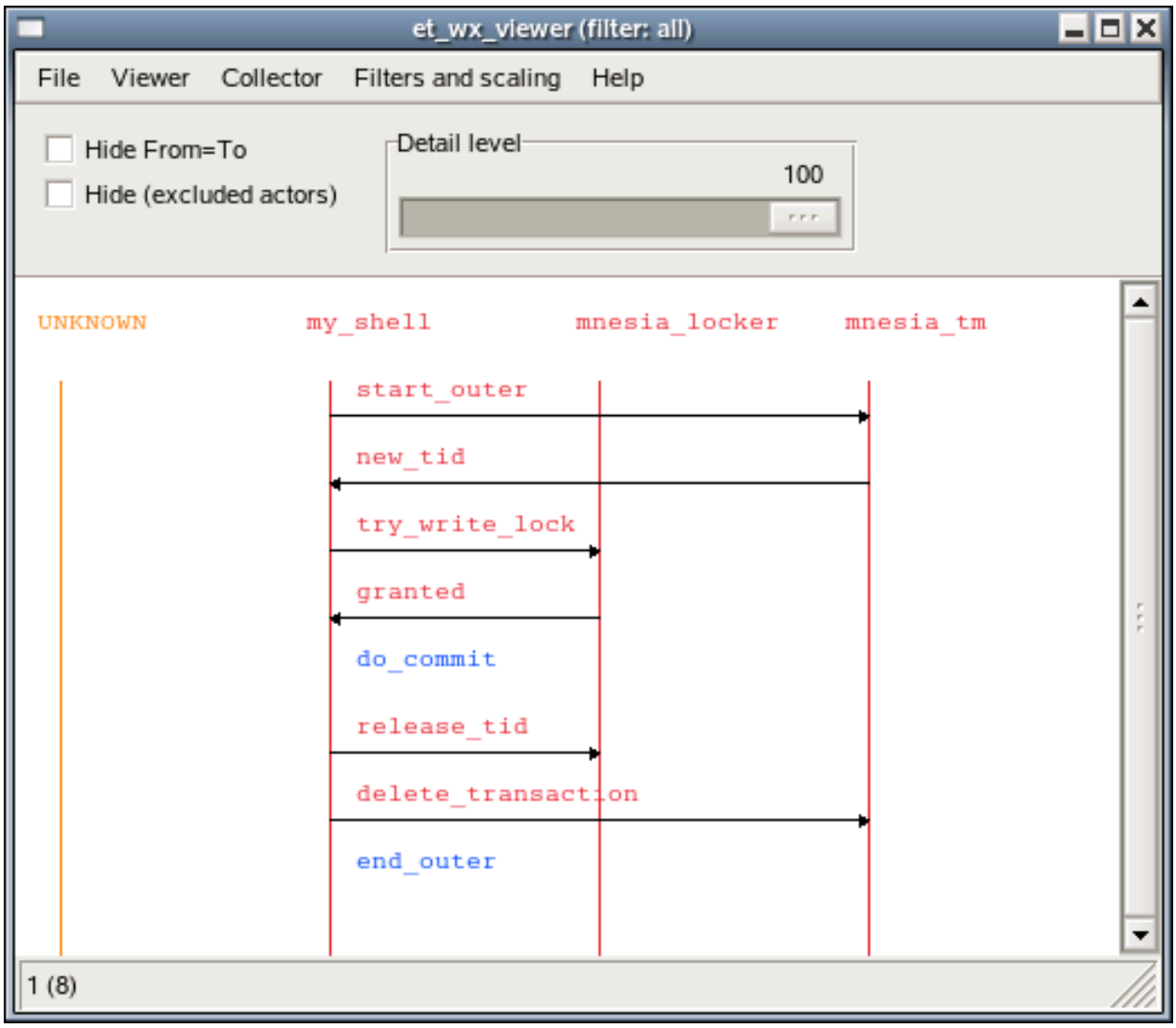



Figure 3.2: Two actors has switched places

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. When doing that a `Contents Viewer` window pops up. It may look like this:




Figure 3.3: Details of a write lock message

## 1.3.2 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) -> false | true | {true, NewEvent}

TraceData = Event | erlang_trace_data()
Event = #event{}
NewEvent = #event{}
```

The interface of the filter function is the same as the the filter functions for the good old `lists:filtermap/2`. If the filter returns `false` it means that the trace data should silently be dropped. `true` means that the trace data data already is an `Event Record` and that it should be kept as it is. `true` means that the `TraceData` already is an `Event Record` and that it should be kept as it is. `{true, NewEvent}` means that the original trace data should be replaced with `Event`. This provides means to get rid of unwanted `Events` as well as enabling alternate views of an `Event`.

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`.

All filters (registered in a `Collector` or in a `Viewer`) must be able to handle an `Event record` as input. The `Collector Filter` (that is the filter named `all`) is a little bit special, as its input also may be raw `Erlang Trace Data`

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 each `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 in `et/examples/et_demo.erl` replaces the actor names `mnesia_tm` and `mnesia_locker` and leaves everything else in the record as it was:

```
mgr_actors(E) when is_record(E, event) ->
    Actor = fun(A) ->
                case A of
                    mnesia_tm     -> trans_mgr;
                    mnesia_locker -> lock_mgr;
                    _             -> A
                end
            end,
    {true, E#event{from = Actor(E#event.from),
                   to = Actor(E#event.to),
                   contents = [{orig_from, E#event.from},
                               {orig_to,   E#event.to},
                               {orig_contents, E#event.contents}]}}.
```

If we now add the filter to the running `Collector`:

```
4> Fun = fun(E) -> et_demo:mgr_actors(E) end.
#Fun<erl_eval.6.13229925>
5> et_collector:dict_insert(Collector, {filter, mgr_actors}, Fun).
ok
```

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:

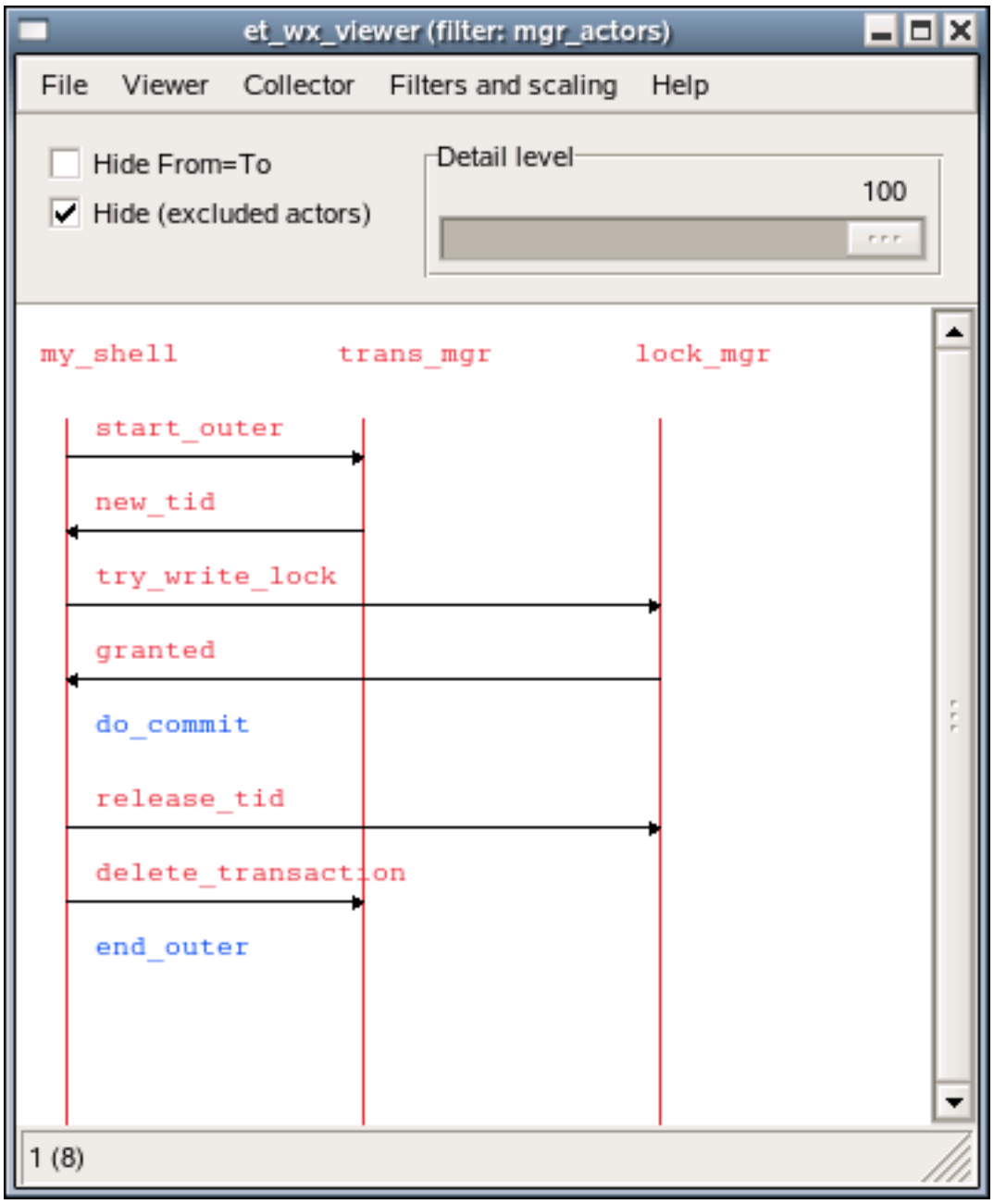



Figure 3.4: The same trace data in a different view

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 also is a filter menu that enables 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:

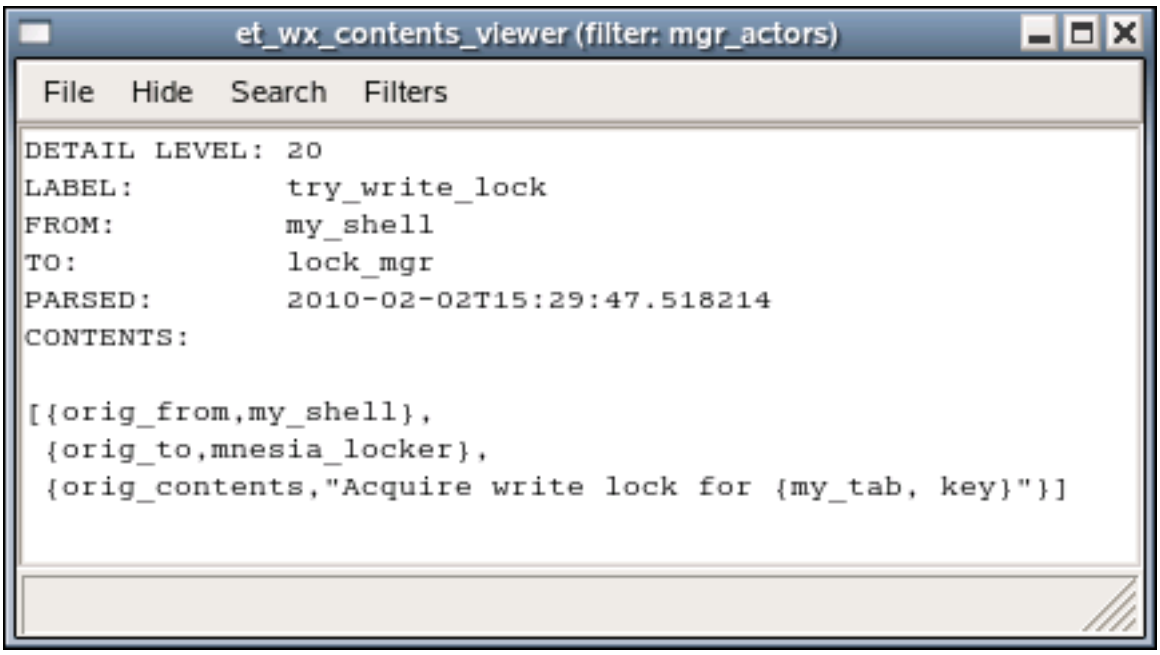



Figure 3.5: The trace Event in the mgr_actors view

Select the `all` entry in the `Filters` menu and a new `Contents Viewer window` will pop up showing the same trace `Event` in the collectors view:

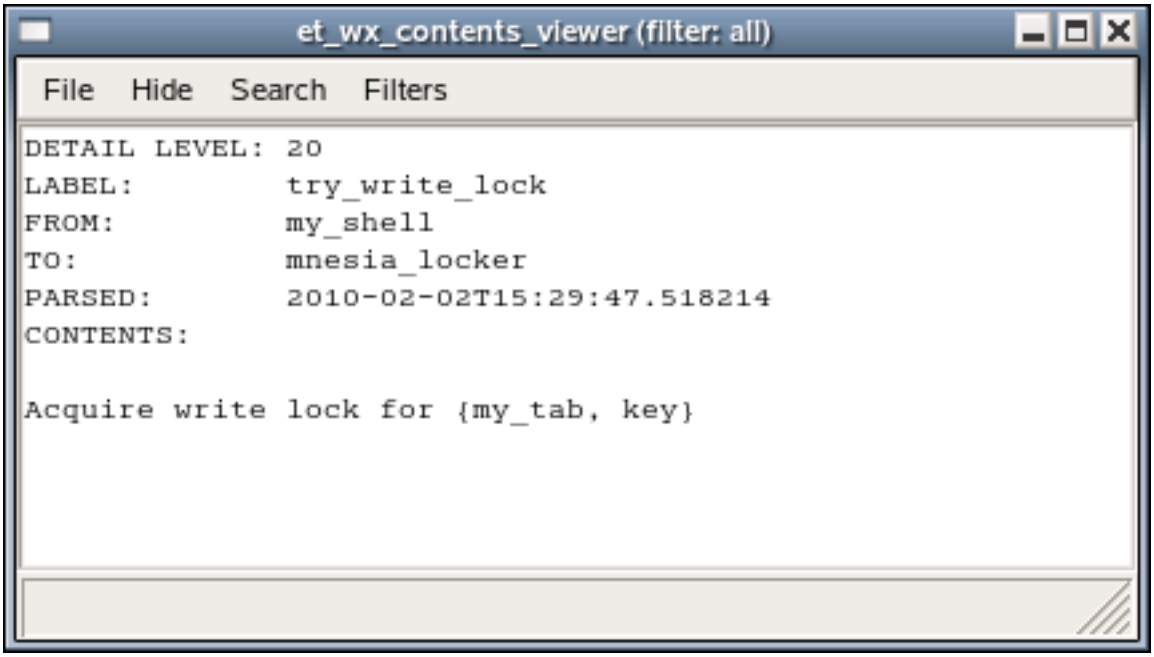



Figure 3.6: The same trace Event in the collectors view

### 1.3.3 Trace clients

As you have seen, it is possible to use the `et_collector:report_event/5,6` 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 raw 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.

## 1.3.4 Global tracing

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 has 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 newly connected 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:trace_me/4,5 functions. 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:trace_me/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.

## 1.3.5 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:

- `Clear all events in the Collector` - Deletes all `Events` stored in the `Collector` and notifies all connected `Viewers` about this.
- `Load events to the Collector from file` - Loads the `Collector` with `Events` from a file and notifies all connected `Viewers` about this.
- `Save all events in the Collector to file` - Saves all `Events` stored in the `Collector` to file.
- `Print setup` - Enables editing of printer setting, such as paper and layout.
- `Print current page` - Prints the events on the current page. The page size is dependent of the selected paper type.
- `Print all pages` - Prints all events. The page size is dependent of the selected paper type.
- `Close this Viewer` - Closes this `Viewer` window, but keeps all other `Viewers` windows and the `Collector` process.
- `Close other Viewers, but this` - Keeps this `Viewer` window and its `Collector` process, but closes all other `Viewers` windowsconnected to the same `Collector`.
- `Close all Viewers and the Collector` - Closes the `Collector` and all `Viewers` connected to it.

Viewer menu:

- `First` - Scrolls `this` viewer to the first `Event` in the `Collector`.
- `Last` - Scrolls `this` viewer to the last `Event` in the `Collector`.
- `Prev` - Scrolls `this` viewer one page backwards.
- `Next` - Scrolls `this` viewer one page forward.
- `Refresh` - Clears `this` viewer and re-read its `Events` from the `Collector`.
- `Up` - Scrolls a few `Events` backwards.
- `Down` - Scrolls a few `Events` forward.
- `Display all actors.` - Reset the settings for hidden and/or highlighted actors.

Collector menu:

- `First` - Scrolls`all` viewers to the first `Event` in the `Collector`.
- `Last` - Scrolls `all` viewers to the last `Event` in the `Collector`.
- `Prev` - Scrolls `all` viewers one page backwards.
- `Next` - Scrolls `all` viewers one page forward.
- `Refresh` - Clears `all` viewers and re-read their `Events` from the `Collector`.

Filters and scaling menu:

- `ActiveFilter (=)` - Starts a new `Viewer` window with the same active filter and scale as the current one.
- `ActiveFilter (+)` - Starts a new `Viewer` window with the same active filter but a larger scale than the current one.
- `ActiveFilter (-)` - Starts a new `Viewer` window with the same active filter but a smaller scale than the current one.
- `all (0)` - Starts a new `Viewer` with the `Collector Filter` as active filter. It will cause all events in the collector to be viewed.
- `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 get the shortcut number 2, the next one number 3 etc. The names are sorted.

Slider and radio buttons:

- `Hide From=To` - When true, this means that the `Viewer` will hide all `Events` where the from-actor equals to its to-actor. These events are sometimes called actions.

- `Hide (excluded actors)` - When true, this means that the `Viewer` will hide all `Events` whose actors are marked as excluded. Excluded actors are normally enclosed in round brackets when they are displayed in the `Viewer`.
- `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:

- `Vertical scroll` - Use mouse wheel and up/down arrows to scroll little. Use page up/down and home/end buttons to scroll more.
- `Display details of an event` - Left mouse click on the event label or the arrowand a new `Contents Viewer` window will pop up, displaying the contents of an `Event`.
- `Highlight actor (toggle)` - Left mouse click on the actor name tag. The actor name will be enclosed in square brackets `[]`. When one or more actors are highlighted, only events related to those actors are displayed. All others are hidden.
- `Exclude actor (toggle)` - Right mouse click on the actor name tag. The actor name will be enclosed in round brackets `()`. When an actor is excluded, all events related to this actor is hidden. If the checkbox `Hide (excluded actors)` is checked, even the name tags and corresponding vertical line of excluded actors will be hidden.
- `Move actor` - Left mouse button drag and drop on actor name tag. Move the 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.
- `Display all actors` - Press the 'a' button. Reset the settings for hidden and/or highlighted actors.

### 1.3.6 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`).

### 1.3.7 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.

## 1.4 Advanced examples

### 1.4.1 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:

```
sim_trans() ->
    sim_trans([]).

sim_trans(ExtraOptions) ->
    Options = [{dict_insert, {filter, mgr_actors}, fun mgr_actors/1}],
    {ok, Viewer} = et_viewer:start_link(Options ++ ExtraOptions),
    Collector = et_viewer:get_collector_pid(Viewer),
    et_collector:report_event(Collector, 60, my_shell, mnesia_tm, start_outer,
                              "Start outer transaction"),
    et_collector:report_event(Collector, 40, mnesia_tm, my_shell, new_tid,
                              "New transaction id is 4711"),
    et_collector:report_event(Collector, 20, my_shell, mnesia_locker, try_write_lock,
                              "Acquire write lock for {my_tab, key}"),
    et_collector:report_event(Collector, 10, mnesia_locker, my_shell, granted,
                              "You got the write lock for {my_tab, key}"),
    et_collector:report_event(Collector, 60, my_shell, do_commit,
                              "Perform  transaction commit"),
    et_collector:report_event(Collector, 40, my_shell, mnesia_locker, release_tid,
                              "Release all locks for transaction 4711"),
    et_collector:report_event(Collector, 60, my_shell, mnesia_tm, delete_transaction,
                              "End of outer transaction"),
    et_collector:report_event(Collector, 20, my_shell, end_outer,
                              "Transaction returned {atomic, ok}"),
    {collector, Collector}.
```

```
mgr_actors(E) when is_record(E, event) ->
    Actor = fun(A) ->
                case A of
                    mnesia_tm     -> trans_mgr;
                    mnesia_locker -> lock_mgr;
                    _             -> A
                end
            end,
    {true, E#event{from = Actor(E#event.from),
                   to = Actor(E#event.to),
                   contents = [{orig_from, E#event.from},
                               {orig_to,   E#event.to},
                               {orig_contents, E#event.contents}]}}.
```

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:

```
Erlang R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
1> {ok, Viewer} = et_viewer:start([]).
{ok,<0.40.0>;}
2> et_demo:sim_trans().
{ok,{table_handle,<0.45.0>,24596,trace_ts,
     #Fun<et_collector.0.62831470>}}
```

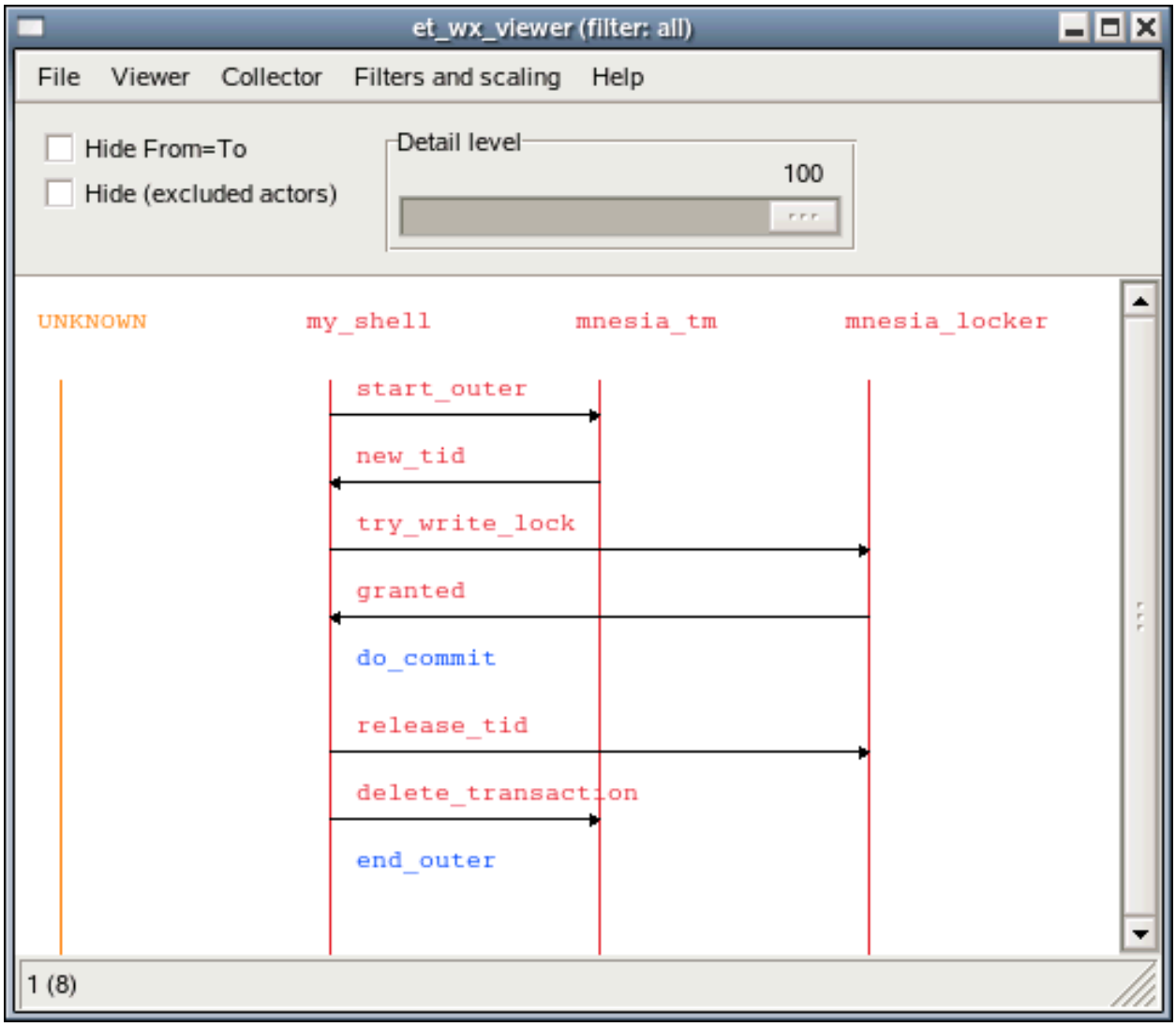



Figure 4.1: A simulated Mnesia transaction which writes one record

## 1.4.2 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:

```
module_as_actor(E) when is_record(E, event) ->
    case lists:keysearch(mfa, 1, E#event.contents) of
        {value, {mfa, {M, F, _A}}} ->
            case lists:keysearch(pam_result, 1, E#event.contents) of
                {value, {pam_result, {M2, _F2, _A2}}} ->
                    {true, E#event{label = F, from = M2, to = M}};
                _ ->
                    {true, E#event{label = F, from = M, to = M}}
            end;
        _ ->
            false
    end.
```

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

```
plain_process_info(E) when is_record(E, event) ->
    case E#event.label of
        send                          -> true;
        send_to_non_existing_process  -> true;
        'receive'                     -> true;
        spawn                         -> true;
        exit                          -> true;
        link                          -> true;
        unlink                        -> true;
        getting_linked                -> true;
        {seq_send, _Label}            -> true;
        {seq_receive, _Label}         -> true;
        {seq_print, _Label}           -> true;
        {drop, _N}                    -> true;
        _                             -> false
    end.
```

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:

```
plain_process_info_nolink(E) when is_record(E, event) ->
    (E#event.label /= link) and
    (E#event.label /= unlink) and
    (E#event.label /= getting_linked) and
    plain_process_info(E).
```

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:

```
start() ->
    start([]).

start(ExtraOptions) ->
    Options = [{trace_global, true},
               {parent_pid, undefined},
               {max_actors, infinity},
               {max_events, 1000},
               {active_filter, module_as_actor}],
    et_viewer:start_link(filters() ++ Options ++ ExtraOptions).
```

A simple one-liner starts the tool:

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

The filters are included by the following parameters:

```
filters() ->
    [{dict_insert, {filter, module_as_actor},
                   fun module_as_actor/1},
     {dict_insert, {filter, plain_process_info},
                   fun plain_process_info/1},
     {dict_insert, {filter, plain_process_info_nolink},
                   fun plain_process_info_nolink/1},
     {dict_insert, {filter, named_process_info},
                   fun named_process_info/1},
     {dict_insert, {filter, named_process_info_nolink},
                   fun named_process_info_nolink/1},
     {dict_insert, {filter, node_process_info},
                   fun node_process_info/1},
     {dict_insert, {filter, node_process_info_nolink},
                   fun node_process_info_nolink/1},
     {dict_insert, {filter, application_as_actor},
                   fun application_as_actor/1}
    ].
```

### 1.4.3 Erlang trace of a real 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:

```
trace_mnesia() ->
    Modules = mnesia:ms(),
    Spec = [{message, {caller}}, {return_trace}],
    Flags = [send, 'receive', procs, timestamp],
    dbg:p(all, [call, timestamp]),
    [dbg:tpl(M, [{'_', [], Spec}]) || M <- Modules],
    LocallyRunningServers = [M || M <- Modules, whereis(M) /= undefined],
    [dbg:p(whereis(RS), Flags) || RS <- LocallyRunningServers],
    dbg:p(self(), Flags),
    LocallyRunningServers.
```

The `et_demo:live_trans/0` function starts the global `Collector`, 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 a simple `Mnesia` transaction that writes a single record is run:

```
live_trans() ->
    live_trans([]).

live_trans(ExtraOptions) ->
    Options = [{title, "Mnesia tracer"},
        {hide_actions, true},
        {active_filter, named_process_info_nolink}],
    et_demo:start(Options ++ ExtraOptions),
    mnesia:start(),
    mnesia:create_table(my_tab, [{ram_copies, [node()]}]),
    et_demo:trace_mnesia(),
    register(my_shell, self()),

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

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

```
erl -pa ../examples
Erlang R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4]
                           [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
1> et_demo:live_trans().
{atomic,ok}
```

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

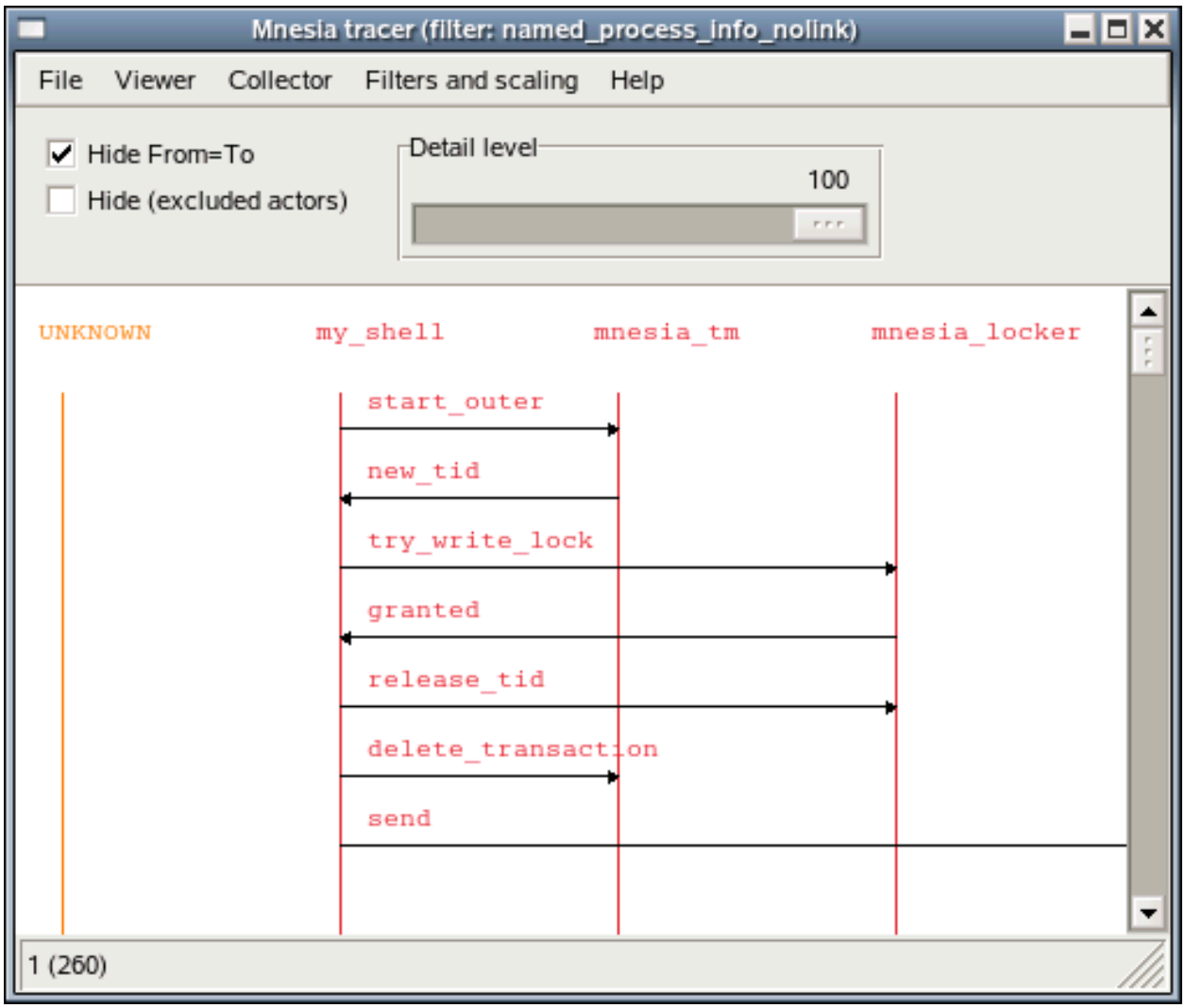



Figure 4.2: A real Mnesia transaction which writes one record

## 1.4.4 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:trace_me/5`. For each call a detail level is given in order to enable dynamic control of 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 a global `Collector` and its `Viewer`.

```
erl -sname observer
Erlang R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
(observer@falco)1> megaco_filter:start().
{ok,<0.48.0>}
```

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 R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
(mgc@falco)1> net:ping(observer@falco).
pong
(mgc@falco)2> megaco:start().
ok
(mgc@falco)3> megaco_simple_mgc:start().
{ok,[{ok,2944,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_pretty_text_encoder,[],megaco_tcp,dynamic}},
     {ok,2944,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_pretty_text_encoder,[],megaco_udp,dynamic}},
     {ok,2945,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_binary_encoder,[],megaco_tcp,dynamic}},
     {ok,2945,
         {megaco_receive_handle,{deviceName,"controller"},
                                megaco_binary_encoder,[],megaco_udp,dynamic}}]}
```

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:

```
Erlang R13B03 (erts-5.7.4) [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.7.4  (abort with ^G)
(mg@falco)1> net:ping(observer@falco).
pong
(mg@falco)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@falco)3> megaco:start().
ok
(mg@falco)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,"controller"},
                              asn1_NOVALUE,asn1_NOVALUE,asn1_NOVALUE,
                              asn1_NOVALUE}}}}]}]}}},
 {{deviceName,"gateway_tb"},
  {1,
   {ok,[{'ActionReply',0,asn1_NOVALUE,asn1_NOVALUE,
             [{serviceChangeReply,
                  {'ServiceChangeReply',
                      [{megaco_term_id,false,["root"]}],
                      {serviceChangeResParms,
                          {'ServiceChangeResParm',
                              {deviceName,"controller"},
                              asn1_NOVALUE,asn1_NOVALUE,asn1_NOVALUE,
                              asn1_NOVALUE}}}}]}]}}},
 {{deviceName,"gateway_ut"},
  {1,
   {ok,[{'ActionReply',0,asn1_NOVALUE,asn1_NOVALUE,
             [{serviceChangeReply,
                  {'ServiceChangeReply',
                      [{megaco_term_id,false,["root"]}],
                      {serviceChangeResParms,
                          {'ServiceChangeResParm',
                              {deviceName,"controller"},
                              asn1_NOVALUE,asn1_NOVALUE,asn1_NOVALUE,
                              asn1_NOVALUE}}}}]}]}}},
 {{deviceName,"gateway_ub"},
  {1,
   {ok,[{'ActionReply',0,asn1_NOVALUE,asn1_NOVALUE,
             [{serviceChangeReply,
                  {'ServiceChangeReply',
                      [{megaco_term_id,false,["root"]}],
                      {serviceChangeResParms,
                          {'ServiceChangeResParm',
                              {deviceName,"controller"},
                              asn1_NOVALUE,asn1_NOVALUE,
                              asn1_NOVALUE,...}}}}]}]}}}]
```

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:

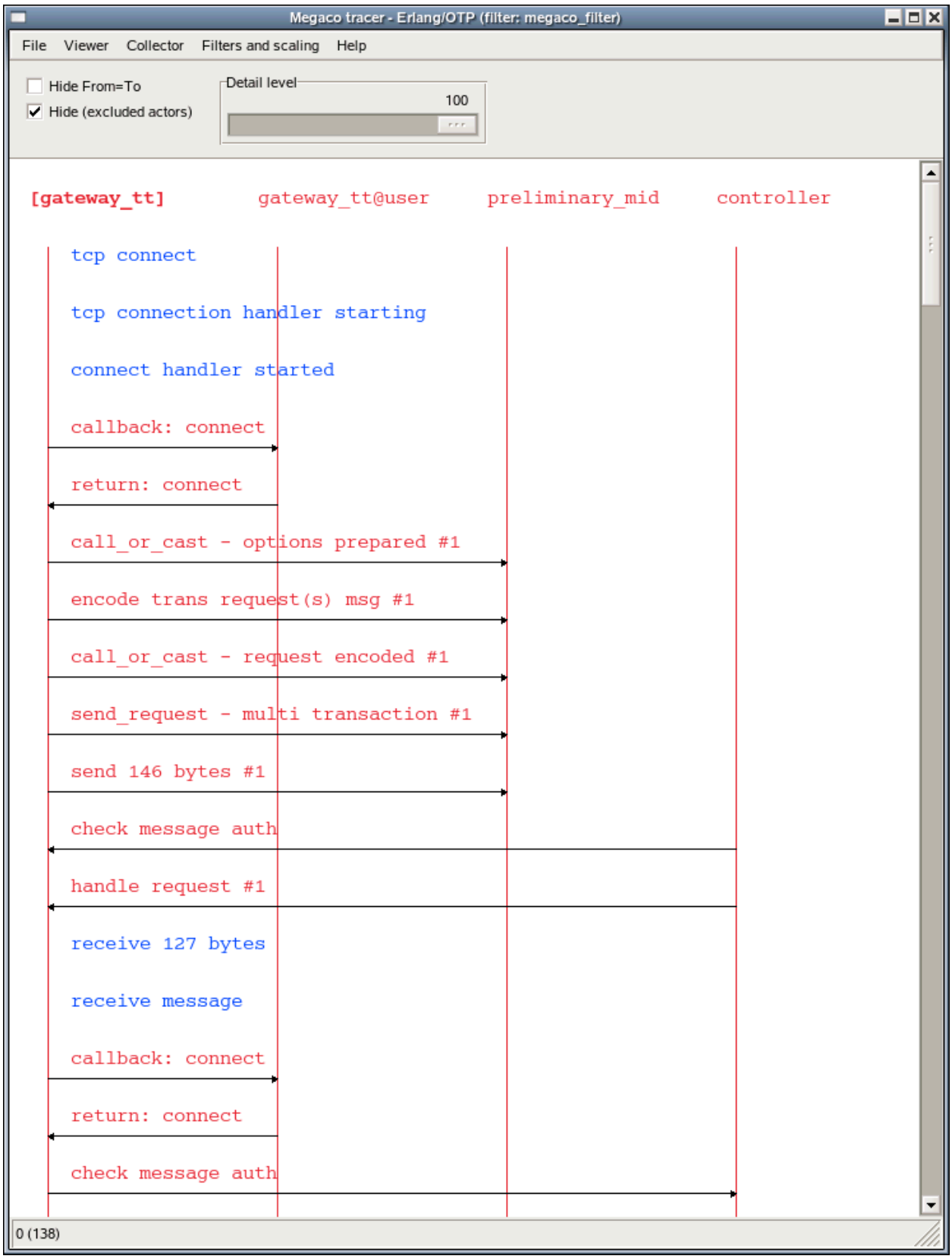



Figure 4.3: The viewer adopted for Megaco

A pretty printed `Megaco` message looks like this:

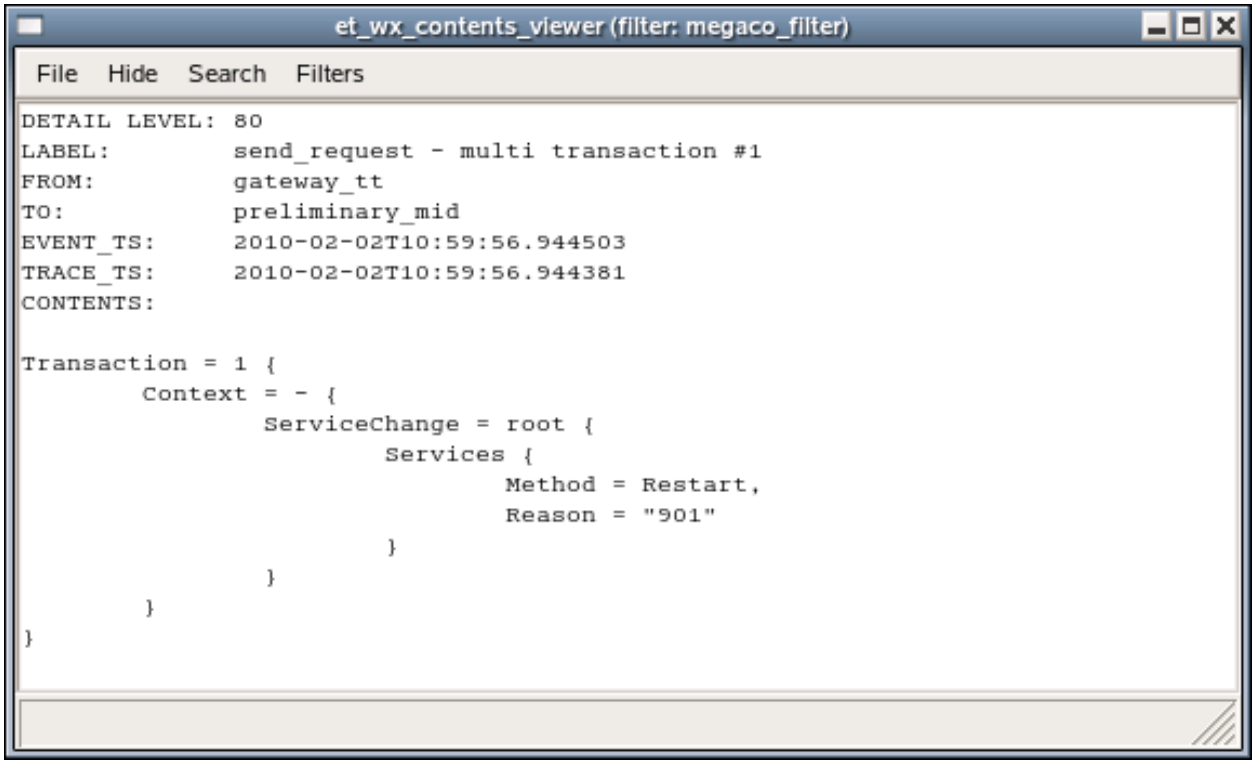



Figure 4.4: A textual Megaco message

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

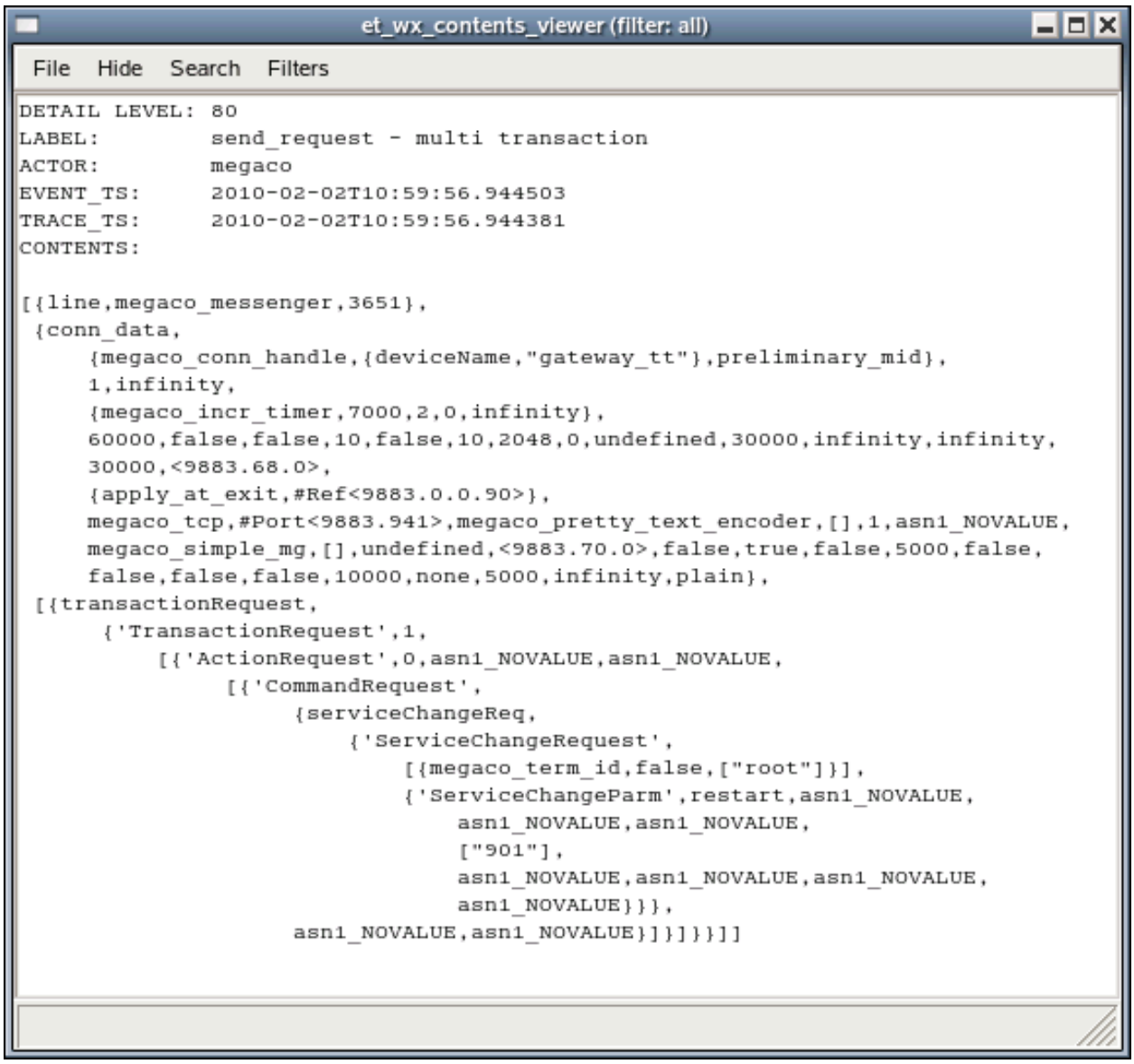



Figure 4.5: The internal form of a Megaco message

# 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

`trace_me(DetailLevel, From, To, Label, Contents) -> hopefully_traced`

Types:

```
DetailLevel = integer(X) when X =< 0, X >= 100
From = actor()
To = actor()
Label = atom() | string() | term()
Contents = [{Key, Value}] | term()
actor() = term()
```

A function that is intended to be traced.

This function is 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. The 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. It is preferred to use an atom but a string would also 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.

`trace_me(DetailLevel, FromTo, Label, Contents) -> hopefully_traced`

Invokes `et:trace_me/5` with both `From` and `To` set to `FromTo`.

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

These functions sends a signal to the outer space and the caller hopes that someone is listening. In other words, they invoke `et:trace_me/4` and `et:trace_me/5` respectively.

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

Deprecated functions which for the time being are kept for backwards compatibility. Invokes `et:trace_me/4` and `et:trace_me/5` respectively.

# 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
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.

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.

```
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_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
Acc = NewAcc = term()
```

Iterate over the currently stored events.

Iterates over the currently stored events and 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 `et:trace_me/4,5`

Max detail level activates tracing of all calls to `et:trace_me/4,5`

integer(X) detail level activates tracing of all calls to `et:trace_me/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 `et:trace_me/4,5`

`max` detail level activates tracing of all calls to `et:trace_me/4,5`

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

An empty match spec deactivates tracing of calls to `et:trace_me/4,5`

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

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

Types:

```
Mod = module_name() | undefined
module_name() = atom()
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:trace_me/4`and `et:trace_me/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_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_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.