Observer

version 0.9

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 1

Observer User's Guide

The *OBSERVER* application contains tools for tracing and investigation of distributed systems.

1.1 Trace Tool Builder

1.1.1 Introduction

The Trace Tool Builder is a base for building trace tools for single node or distributed erlang systems. It requires the `runtime_tools` application to be available on the traced node.

The main features of the Trace Tool Builder are:

- Start tracing to file ports on several nodes with one function call.
- Write additional information to a trace information file, which is read during formatting.
- Restoring of previous configuration by maintaining a history buffer and handling configuration files.
- Some simple support for sequential tracing.
- Formatting of binary trace logs and merging of logs from multiple nodes.

Even though the intention of the Trace Tool Builder is to serve as a base for tailor made trace tools, it is of course possible to use it directly from the erlang shell. The application only allows the use of file port tracer, so if you would like would like to use other types of trace clients you will be better off using `dbg` directly instead.

1.1.2 Getting Started

The `ttb` module is the interface to all functions in the Trace Tool Builder. To get started the least you need to do is to start a tracer with `ttb:tracer/0/1/2`, and set the required trace flags on the processes you want to trace with `ttb:p/2`. Then, when the tracing is completed, you must stop the tracer with `ttb:stop/0/1` and format the trace log with `ttb:format/1/2`.

`ttb:tracer/0/1/2` opens a file trace port on each node that shall be traced. All trace messages will be written to this port and end up in a binary file (the binary trace log).

`ttb:p/2` specifies which processes that shall be traced. Trace flags given in this call specifies what to trace on each process. You can call this function several times if you like different trace flags to be set on different processes.

If you want to trace function calls (i.e. if you have the `call` trace flag set on any of your processes), you must also set trace patterns on the required function(s) with `ttb:tp` or `ttb:tpl`. A function is only traced if it has a trace pattern. The trace pattern specifies how to trace the function by using match specifications. Match specifications are described in the User's Guide for the erlang runtime system `erts`.

`ttb:stop/0/1` stops tracing on all nodes, deletes all trace patterns and flushes the trace port buffer.

`ttb:format/1/2` translates the binary trace logs into something readable. By default `ttb` presents each trace message as a line of text, but you can also write your own handler to make more complex interpretations of the trace information. A trace log can even be presented graphically via the Event Tracer application. Note that if you give the `format` option to `ttb:stop/1` the formatting is automatically done when stopping `ttb`.

Example: Tracing the local node from the erlang shell

This small module is used in the example:

```
-module(m).
-export([f/0]).
f() ->
   receive
      From when pid(From) ->
         Now = erlang:now(),
         From ! {self(),Now}
   end.
```

The following example shows the basic use of `ttb` from the erlang shell. Default options are used both for starting the tracer and for formatting. This gives a trace log named `Node-ttb`, where `Node` is the name of the node. The default handler prints the formatted trace messages in the shell.

```
(tiger@durin)47> %% First I spawn a process running my test function
(tiger@durin)47> Pid = spawn(m,f,[]).
<0.125.0>
(tiger@durin)48>
(tiger@durin)48> %% Then I start a tracer...
(tiger@durin)48> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)49>
(tiger@durin)49> %% and activate the new process for tracing
(tiger@durin)49> %% function calls and sent messages.
(tiger@durin)49> ttb:p(Pid,[call,send]).
{ok,[{<0.125.0>,[{matched,tiger@durin,1}]}]}
(tiger@durin)50>
(tiger@durin)50> %% Here I set a trace pattern on erlang:now/0
(tiger@durin)50> %% The trace pattern is a simple match spec
(tiger@durin)50> %% generated by dbg:fun2ms/1. It indicates that
(tiger@durin)50> %% the return value shall be traced.
(tiger@durin)50> MS = dbg:fun2ms(fun(_) -> return_trace() end).
[{'_',[],[{return_trace}]}]
(tiger@durin)51> ttb:tp(erlang,now,MS).
{ok,[{matched,tiger@durin,1},{saved,1}]}
(tiger@durin)52>
```

```
(tiger@durin)52> %% I run my test (i.e. send a message to
(tiger@durin)52> %% my new process)
(tiger@durin)52> Pid ! self().
<0.72.0>
(tiger@durin)53>
(tiger@durin)53> %% And then I have to stop ttb in order to flush
(tiger@durin)53> %% the trace port buffer
(tiger@durin)53> ttb:stop().
stopped
(tiger@durin)54>
(tiger@durin)54> %% Finally I format my trace log
(tiger@durin)54> ttb:format("tiger@durin-ttb").
({<0.125.0>,{m,f,0},tiger@durin}) call erlang:now()
({<0.125.0>,{m,f,0},tiger@durin}) returned from erlang:now/0 ->
{1031,133451,667611}
({<0.125.0>,{m,f,0},tiger@durin}) <0.72.0> !
{<0.125.0>,{1031,133451,667611}}
ok
```

Example: Build your own tool

This small example shows a simple tool for “debug tracing”, i.e. tracing of function calls with return values.

```
-module(mydebug).
-export([start/0,trc/1,stop/0,format/1]).
-export([print/4]).

%% Include ms_transform.hrl so that I can use dbg:fun2ms/2 to
%% generate match specifications.
-include_lib("stdlib/include/ms_transform.hrl").

%%% -------------Tool API-------------
%%% ----------------------------------
%%% Star the "mydebug" tool
start() ->
    %% The options specify that the binary log shall be named
    %% <Node>-debug_log and that the print/4 function in this
    %% module shall be used as format handler
    ttb:tracer(all,[{file,"debug_log"},{handler,{{?MODULE,print},0}}]),
    %% All processes (existing and new) shall trace function calls
    %% and include a timestamp in each trace message
    ttb:p(all,[call,timestamp]).

%%% Set trace pattern on function(s)
trc(M) when atom(M) ->
    trc({M,'_','_'});
trc({M,F}) when atom(M), atom(F) ->
    trc({M,F,'_'});
trc({M,F,_A}=MFA) when atom(M), atom(F) ->
    %% This match spec specifies that return values shall
    %% be traced. NOTE that ms_transform.hrl must be included
```

```
    %% if dbg:fun2ms/1 shall be used!
    MatchSpec = dbg:fun2ms(fun(_) -> return_trace() end),
    ttb:tpl(MFA,MatchSpec).

%%% Format a binary trace log
format(File) ->
    ttb:format(File).

%%% Stop the "mydebug" tool
stop() ->
    ttb:stop().

%%% --------Internal functions--------
%%% ----------------------------------
%%% Format handler
print(_Out,end_of_trace,_TI,N) ->
    N;
print(Out,Trace,_TI,N) ->
    do_print(Out,Trace,N),
    N+1.

do_print(Out,{trace_ts,P,call,{M,F,A},Ts},N) ->
    io:format(Out,
              "~w: ~w, ~w:~n"
              "Call      : ~w:~w/~w~n"
              "Arguments :~p~n~n",
              [N,Ts,P,M,F,length(A),A]);
do_print(Out,{trace_ts,P,return_from,{M,F,A},R,Ts},N) ->
    io:format(Out,
              "~w: ~w, ~w:~n"
              "Return from  : ~w:~w/~w~n"
              "Return value :~p~n~n",
              [N,Ts,P,M,F,A,R]).
```

To distinguish trace logs produced with this tool from other logs, the `file` option is used in `tracer/2`. The logs will therefore be named `Node-debug_log`, where `Node` is the name of the node where the log is produced.

By using the `handler` option when starting the tracer, the information about how to format the file is stored in the trace information file (`.ti`). This is not necessary, as it might be given at the time of formatting instead. It can however be useful if you e.g. want to automatically format your trace logs by using the `format` option in `ttb:stop/1`. It also means that you don't need any knowledge of the content of a binary log to be able to format it the way it was intended. If the `handler` option is given both when starting the tracer and when formatting, the one given when formatting is used.

The `call` trace flag is set on all processes. This means that any function activated with the `trc/1` command will be traced on all existing and new processes.

1.1.3 Running the Trace Tool Builder against a remote node

The Observer application might not always be available on the node that shall be traced (in the following called the “traced node”). It is still possible to run the Trace Tool Builder from another node (in the following called the “trace control node”) as long as

- The Observer application is available on the trace control node.
- The Runtime Tools application is available on both the trace control node and the traced node.

If the Trace Tool Builder shall be used against a remote node, it is highly recommended to start the trace control node as *hidden*. This way it can connect to the traced node without the traced node "seeing" it, i.e. if the `nodes()` BIF is called on the traced node, the trace control node will not show. To start a hidden node, add the `-hidden` option to the `erl` command, e.g.

```
% erl -sname trace_control -hidden
```

Diskless node

If the traced node is diskless, `ttb` must be started from a trace control node with disk access, and the `file` option must be given to the `tracer/2` function with the value `{local, File}`, e.g.

```
(trace_control@durin)1> ttb:tracer(mynode@diskless,[{file,{local,
{wrap,"mytrace"}}}]).
{ok,[mynode@diskless]}
```

1.1.4 Trace Information and the .ti File

In addition to the trace log file(s), a file with the extension `.ti` is created when the Trace Tool Builder is started. This is the trace information file. It is a binary file, and it contains the process information, trace flags used, the name of the node to which it belongs and all information written with the `write_trace_info/2` function.

To be able to use all this information during formatting, it is important that the trace information file exists in the same directory as the trace log, and that it has the same name as the trace log with the additional extension `.ti`.

Except for the process information, everything in the trace information file is passed on to the handler function when formatting. The `TI` parameter is a list of `{Key,ValueList}` tuples. The keys `flags`, `handler`, `file` and `node` are used for information written directly by `ttb`.

You can add information to the trace information file by calling `write_trace_info/2`. Note that `ValueList` always will be a list, and if you call `write_trace_info/2` several times with the same `Key`, the `ValueList` will be extended with a new value each time. Example:

`ttb:write_trace_info(mykey,1)` gives the entry `{mykey,[1]}` in `TI`. Another call, `ttb:write_trace_info(mykey,2)`, changes this entry to `{mykey,[1,2]}`.

1.1.5 Wrap Logs

If you want to limit the size of the trace logs, you can use wrap logs. This works almost like a ciclular buffer. You can specify the maximum number of binary logs and the maximum size of each log. `ttb` will create a new binary log each time a log reaches the maximum size. When the the maximum number of logs are reached, the oldest log is deleted before a new one is created.

Wrap logs can be formatted one by one or all at once. See Formatting [page 6].

1.1.6 Formatting

Formatting can be done automatically when stopping `ttb` (see Automatically collect and format logs from all nodes [page 10]), or explicitly by calling the `ttb:format/1/2` function.

Formatting means to read a binary log and present it in a readable format. You can use the default format handler in `ttb` to present each trace message as a line of text, or write your own handler to make more complex interpretations of the trace information. You can even use the Event Tracer `et` to present the trace log graphically (see Presenting trace logs with Event Tracer [page 7]).

The first argument to `ttb:format/1/2` specifies which binary log(s) to format. This can be the name of one binary log, a list of such logs or the name of a directory containing one or more binary logs. If this argument indicates more than one log, and the `timestamp` flag was set when tracing, the trace messages from the different logs will be merged according to the timestamps in each message.

The second argument to `ttb:format/2` is a list of options. The `out` option specifies the destination where the formatted text shall be written. Default destination is `standard_io`, but a filename can also be given. The `handler` option specifies the format handler to use. If this option is not given, the `handler` option given when starting the tracer is used. If the `handler` option was not given when starting the tracer either, a default handler is used, which prints each trace message as a line of text.

A format handler is a fun taking four arguments. This fun will be called for each trace message in the binary log(s). A simple example which only prints each trace message could be like this:

```
fun(Fd, Trace, _TraceInfo, State) ->
   io:format(Fd, "Trace: ~p~n", [Trace]),
   State
end.
```

`Fd` is the file descriptor for the destination file, or the atom `standard_io`. `_TraceInfo` contains information from the trace information file (see Trace Information and the .ti File [page 5]). `State` is a state variable for the format handler fun. The initial value of the `State` variable is given with the handler option, e.g.

```
ttb:format("tiger@durin-ttb", [{handler, {{Mod,Fun}, initial_state}}])
                                                  ^^^^^^^^^^^^^^
```

Another format handler could be used to calculate time spent by the garbage collector:

```
fun(_Fd,{trace_ts,P,gc_start,_Info,StartTs},_TraceInfo,State) ->
      [{P,StartTs}|State];
   (Fd,{trace_ts,P,gc_end,_Info,EndTs},_TraceInfo,State) ->
      {value,{P,StartTs}} = lists:keysearch(P,1,State),
      Time = diff(StartTs,EndTs),
      io:format("GC in process ~w: ~w milliseconds~n", [P,Time]),
      State -- [{P,StartTs}]
end
```

A more refined version of this format handler is the function `handle_gc/4` in the module `multitrace.erl` which can be found in the `src` directory of the Observer application.

By giving the format handler `et`, you can have the trace log presented graphically with `et_viewer` in the Event Tracer application (see Presenting trace logs with Event Tracer [page 7]).

Wrap logs can be formatted one by one or all in one go. To format one of the wrap logs in a set, give the exact name of the file. To format the whole set of wrap logs, give the name with '*' instead of the wrap count. An example:

Start tracing:

```
(tiger@durin)1> ttb:tracer(node(),[{file,{wrap,"trace"}}]).
{ok,[tiger@durin]}
(tiger@durin)2> ttb:p(...)
...
```

This will give a set of binary logs, like:

```
tiger@durin-trace.0.wrp
tiger@durin-trace.1.wrp
tiger@durin-trace.2.wrp
...
```

Format the whole set of logs:

```
1> ttb:format("tiger@durin-trace.*.wrp").
....
ok
2>
```

Format only the first log:

```
1> ttb:format("tiger@durin-trace.0.wrp").
....
ok
2>
```

To merge all wrap logs from two nodes:

```
1> ttb:format(["tiger@durin-trace.*.wrp","lion@durin-trace.*.wrp"]).
....
ok
2>
```

Presenting trace logs with Event Tracer

For detailed information about the Event Tracer, please turn to the User's Guide and Reference Manuals for the `et` application.

By giving the format handler `et`, you can have the trace log presented graphically with `et_viewer` in the Event Tracer application. `ttb` provides a few different filters which can be selected from the Filter menu in the `et_viewer` window. The filters are names according to the type of actors they present (i.e. what each vertical line in the sequence diagram represent). Interaction between actors is shown as red arrows between two vertical lines, and activities within an actor are shown as blue text to the right of the actors line.

The `processes` filter is the only filter which will show all trace messages from a trace log. Each vertical line in the sequence diagram represents a process. Erlang messages, spawn and link/unlink are typical interactions between processes. Function calls, scheduling and garbage collection are typical activities within a process. `processes` is the default filter.

The rest of the filters will only show function calls and function returns. All other trace message are discarded. To get the most out of these filters, `et_viewer` needs to known the caller of each function and the time of return. This can be obtained by using both the `call` and `return_to` flags when tracing. Note that the `return_to` flag only works with local call trace, i.e. when trace patterns are set with `ttb:tpl`.

The same result can be obtaind by using the `call` flag only and setting a match specification like this on local or global function calls:

```
1> dbg:fun2ms(fun(_) -> return_trace(),message(caller()) end).
[{'_',[],[{return_trace},{message,{caller}}]}]
```

This should however be done with care, since the {`return_trace`} function in the match specification will destroy tail recursiveness.

The `modules` filter shows each module as a vertical line in the sequence diagram. External function calls/returns are shown as interactions between modules and internal function calls/returns are shown as activities within a module.

The `functions` filter shows each function as a vertical line in the sequence diagram. A function calling itself is shown as an activity within a function, and all other function calls are shown as interactions between functions.

The `mods_and_procs` and `funcs_and_procs` filters are equivalent to the `modules` and `functions` filters respectively, except that each module or function can have several vertical lines, one for each process it resides on.

As an example this module is used, and the function `bar:f1()` is called from another module `foo`.

```
-module(bar).
-export([f1/0,f3/0]).
f1() ->
    f2(),
    ok.
f2() ->
    spawn(?MODULE,f3,[]).
f3() ->
    ok.
```

The `call` and `return_to` flags are used, and trace pattern is set on local calls in module `bar`.

`ttb:format("tiger@durin-ttb", [{handler, et}])` gives the following result:

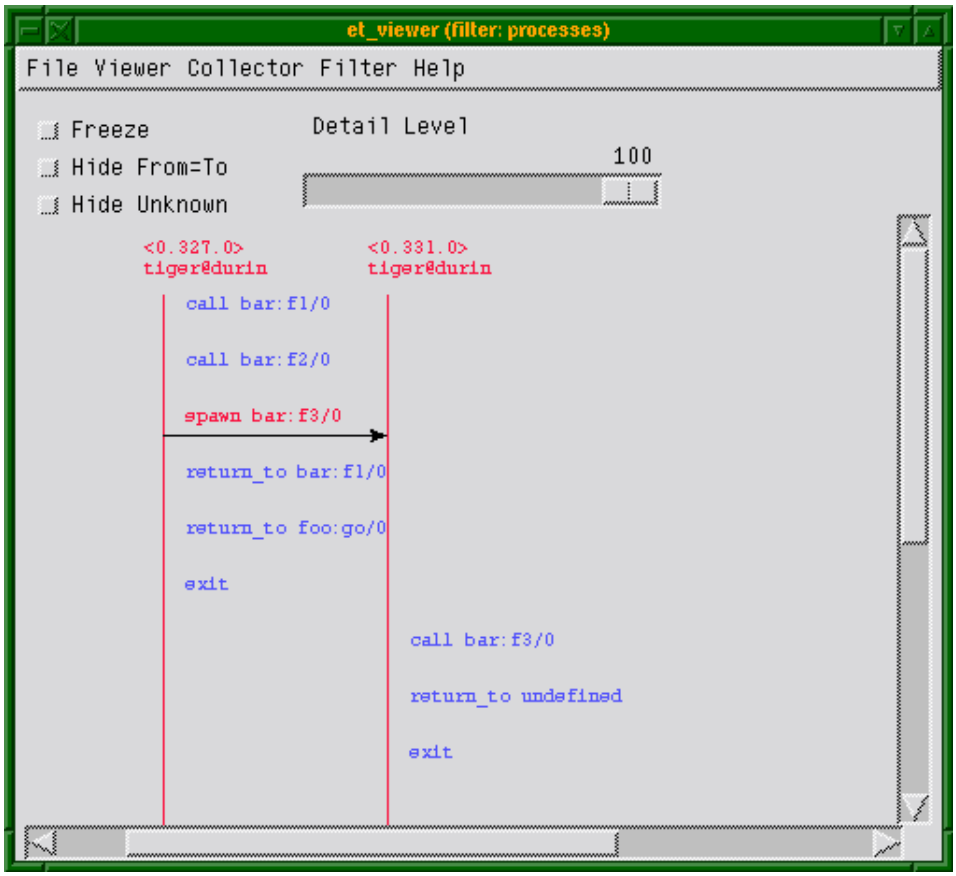

Figure 1.1: Filter: ”processes”

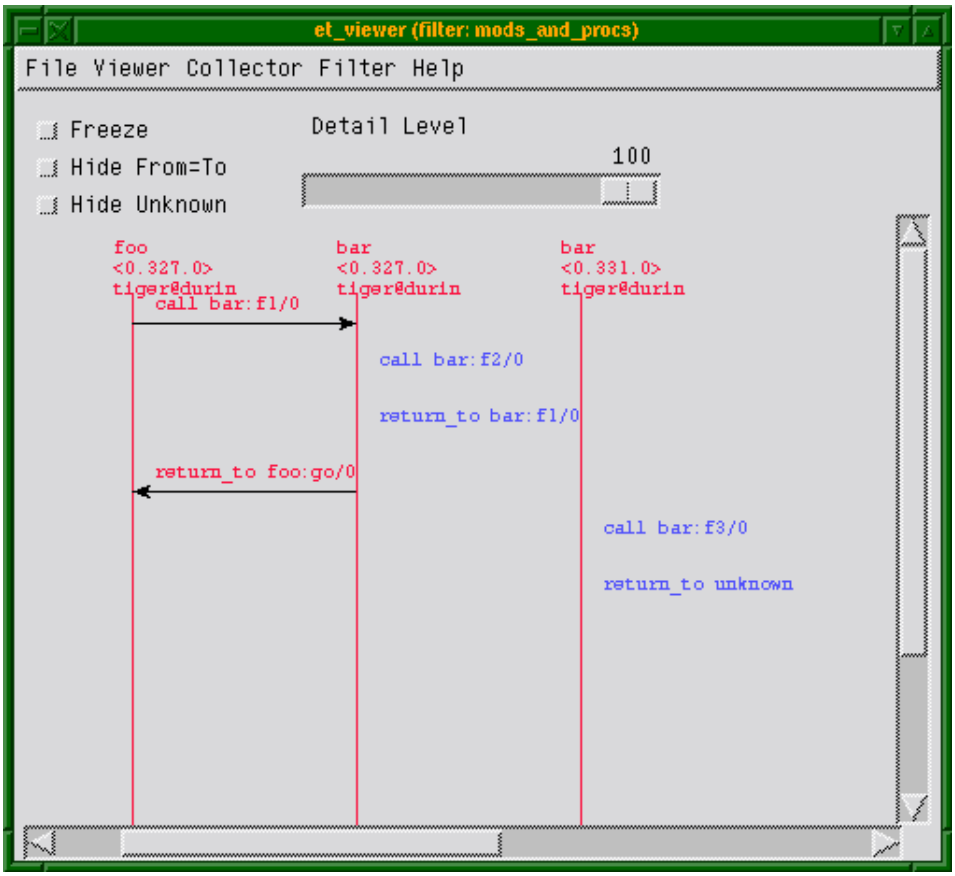

Figure 1.2: Filter: ”mods_and_procs”

1.1.7 Automatically collect and format logs from all nodes

If the option `fetch` is given to the `ttb:stop/1` function, trace logs and trace information files are fetched from all nodes after tracing is stopped. The logs are stored in a new directory named `ttb_upload-Timestamp` under the working directory of the trace control node.

If the option `format` is given to `ttb:stop/1`, the trace logs are automatically formatted after tracing is stopped. Note that `format` also implies `fetch`, i.e. the trace logs will be collected from all nodes as for the `fetch` option before they are formatted. All logs in the upload directory are merged during formatting.

1.1.8 History and Configuration Files

For the tracing functionality, `dbg` could be used instead of the `ttb` for setting trace flags on processes and trace patterns for call trace, i.e. the functions `p`, `tp`, `tpl`, `ctp`, `ctpl` and `ctpg`. The only thing added by `ttb` for these functions is that all calls are stored in the history buffer and can be recalled and stored in a configuration file. This makes it easy to setup the same trace environment e.g. if you want to compare two test runs. It also reduces the amount of typing when using `ttb` from the erlang shell.

Use `list_history/0` to see the content of the history buffer, and `run_history/1` to re-execute one of the entries.

The main purpose of the history buffer is the possibility to create configuration files. Any function stored in the history buffer can be written to a configuration file and used for creating a specific configuration at any time with one single function call.

A configuration file is created or extended with `write_config/2/3`. Configuration files are binary files and can therefore only be read and written with functions provided by `ttb`.

You can write the complete content of the history buffer to a config file by calling `ttb:write_config(ConfigFile,all)`. And you can write selected entries from the history by calling `ttb:write_config(ConfigFile,NumList)`, where `NumList` is a list of integers pointing out the history entries to write.

User defined entries can also be written to a config file by calling the function `ttb:write_config(ConfigFile,ConfigList)` where `ConfigList` is a list of `{Module,Function,Args}`.

Any existing file `ConfigFile` is deleted and a new file is created when `write_config/2` is called. The option `append` can be used if you wish to add something at the end of an existing config file, e.g. `ttb:write_config(ConfigFile,What,[append])`.

Example: History and configuration files

See the content of the history buffer

```
(tiger@durin)191> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)192> ttb:p(self(),[garbage_collection,call]).
{ok,{[<0.1244.0>],[garbage_collection,call]}}
(tiger@durin)193> ttb:tp(ets,new,2,[]).
{ok,[{matched,1}]}
(tiger@durin)194> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}}]
```

Execute an entry from the history buffer:

```
(tiger@durin)195> ttb:ctp(ets,new,2).
{ok,[{matched,1}]}
(tiger@durin)196> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}}]
(tiger@durin)197> ttb:run_history(3).
ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}
```

Write the content of the history buffer to a configuration file:

```
(tiger@durin)198> ttb:write_config("myconfig",all).
ok
(tiger@durin)199> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}}]
```

Extend an existing configuration:

```
(tiger@durin)200> ttb:write_config("myconfig",[{ttb,tp,[ets,delete,1,[]]}],
[append]).
ok
(tiger@durin)201> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}},
 {6,{ttb,tp,[ets,delete,1,[]]}}]
```

Go back to a previous configuration after stopping Trace Tool Builder:

```
(tiger@durin)202> ttb:stop().
ok
(tiger@durin)203> ttb:run_config("myconfig").
ttb:tracer(tiger@durin,[]) ->
{ok,[tiger@durin]}

ttb:p(<0.1244.0>,[garbage_collection,call]) ->
{ok,{[<0.1244.0>],[garbage_collection,call]}}

ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}

ttb:ctp(ets,new,2) ->
{ok,[{matched,1}]}

ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}

ttb:tp(ets,delete,1,[]) ->
{ok,[{matched,1}]}

ok
```

Write selected entries from the history buffer to a configuration file:

```
(tiger@durin)204> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
```

```
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}},
 {6,{ttb,tp,[ets,delete,1,[]]}}]
(tiger@durin)205> ttb:write_config("myconfig",[1,2,3,6]).
ok
(tiger@durin)206> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,tp,[ets,delete,1,[]]}}]
(tiger@durin)207>
```

1.1.9 Sequential Tracing

To learn what sequential tracing is and how it can be used, please turn to the reference manual for the `seq_trace` module in the `kernel` application.

The support for sequencial tracing provided by the Trace Tool Builder includes

- Initiation of the system tracer. This is automatically done when a trace port is started with `ttb:tracer/0/1/2`
- Creation of match specifications which activates sequential tracing

Starting sequential tracing requires that a tracer has been started with the `ttb:tracer/0/1/2` function. Sequential tracing can then either be started via a trigger function with a match specification created with `ttb:seq_trigger_ms/0/1`, or directly by using the `seq_trace` module in the `kernel` application.

Example: Sequential tracing

In the following example, the function `dbg:get_tracer/0` is used as trigger for sequential tracing:

```
(tiger@durin)110> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)111> ttb:p(self(),call).
{ok,{[<0.158.0>],[call]}}
(tiger@durin)112> ttb:tp(dbg,get_tracer,0,ttb:seq_trigger_ms(send)).
{ok,[{matched,1},{saved,1}]}
(tiger@durin)113> dbg:get_tracer(), seq_trace:reset_trace().
true
(tiger@durin)114> ttb:stop().
ok
(tiger@durin)115> ttb:format("tiger@durin-ttb").
({<0.158.0>,{shell,evaluator,3},tiger@durin}) call dbg:get_tracer()
SeqTrace [0]: ({<0.158.0>,{shell,evaluator,3},tiger@durin})
{<0.237.0>,dbg,tiger@durin} ! {<0.158.0>,{get_tracer,tiger@durin}}
[Serial: {0,1}]
SeqTrace [0]: ({<0.237.0>,dbg,tiger@durin})
{<0.158.0>,{shell,evaluator,3},tiger@durin} ! {dbg,{ok,#Port<0.222>}}
[Serial: {1,2}]
ok
(tiger@durin)116>
```

Starting sequential tracing with a trigger is actually more useful if the trigger function is not called directly from the shell, but rather implicitly within a larger system. When calling a function from the shell, it is simpler to start sequential tracing directly, e.g.

```
(tiger@durin)116> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)117> seq_trace:set_token(send,true), dbg:get_tracer(),
seq_trace:reset_trace().
true
(tiger@durin)118> ttb:stop().
ok
(tiger@durin)119> ttb:format("tiger@durin-ttb").
SeqTrace [0]: ({<0.158.0>,{shell,evaluator,3},tiger@durin})
{<0.246.0>,dbg,tiger@durin} ! {<0.158.0>,{get_tracer,tiger@durin}}
[Serial: {0,1}]
SeqTrace [0]: ({<0.246.0>,dbg,tiger@durin})
{<0.158.0>,{shell,evaluator,3},tiger@durin} ! {dbg,{ok,#Port<0.229>}}
[Serial: {1,2}]
ok
(tiger@durin)120>
```

In both examples above, the `seq_trace:reset_trace/0` resets the trace token immediately after the traced function in order to avoid lots of trace messages due to the printouts in the erlang shell.

All functions in the `seq_trace` module, except `set_system_tracer/1`, can be used after the trace port has been started with `ttb:tracer/0/1/2`.

1.1.10 Example: Multipurpose trace tool

The module `multitrace.erl` which can be found in the `src` directory of the Observer application implements a small tool with three possible trace settings. The trace messages are written to binary files which can be formatted with the function `multitrace:format/1/2`.

`multitrace:debug(What)` Start calltrace on all processes and trace the given function(s). The format handler used is `multitrace:handle_debug/4` which prints each call and return. `What` must be an item or a list of items to trace, given on the format {`Module,Function,Arity`}, {`Module,Function`} or just `Module`.

`multitrace:gc(Procs)` Trace garbage collection on the given process(es). The format handler used is `multitrace:handle_gc/4` which prints start and stop and the time spent for each GC.

`multitrace:schedule(Procs)` Trace in- and out-scheduling on the given process(es). The format handler used is `multitrace:handle_schedule/4` which prints each in and out scheduling with process, timestamp and current function. It also prints the total time each traced process was scheduled in.

1.2 Erlang Top

1.2.1 Introduction

Erlang Top, `etop` is a tool for presenting information about erlang processes similar to the information presented by `top` in UNIX.

1.2.2 Output

The output from etop can be graphical or text based.

Text based it looks like this:

```
==============================================================================================
 tiger@durin                                                                    13:40:32
 Load:  cpu         0               Memory:  total        1997    binary          33
        procs     197                        processes       0    code           173
        runq      135                        atom         1002    ets             95

Pid            Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
----------------------------------------------------------------------------------------------
<127.23.0>     code_server                0   59585   78064       0 gen_server:loop/6
<127.21.0>     file_server_2              0   36380   44276       0 gen_server:loop/6
<127.2.0>      erl_prim_loader            0   27962    3740       0 erl_prim_loader:loop
<127.9.0>      kernel_sup                 0    6998    4676       0 gen_server:loop/6
<127.17.0>     net_kernel                62    6018    3136       0 gen_server:loop/6
<127.0.0>      init                       0    4156    4352       0 init:loop/1
<127.16.0>     auth                       0    1765    1264       0 gen_server:loop/6
<127.18.0>     inet_tcp_dist:accept       0     660    1416       0 prim_inet:accept0/2
<127.5.0>      application_controll       0     569    6756       0 gen_server:loop/6
<127.137.0>    net_kernel:do_spawn_       0     553    5840       0 dbg:do_relay_1/1
==============================================================================================
```

And graphically it looks like this:

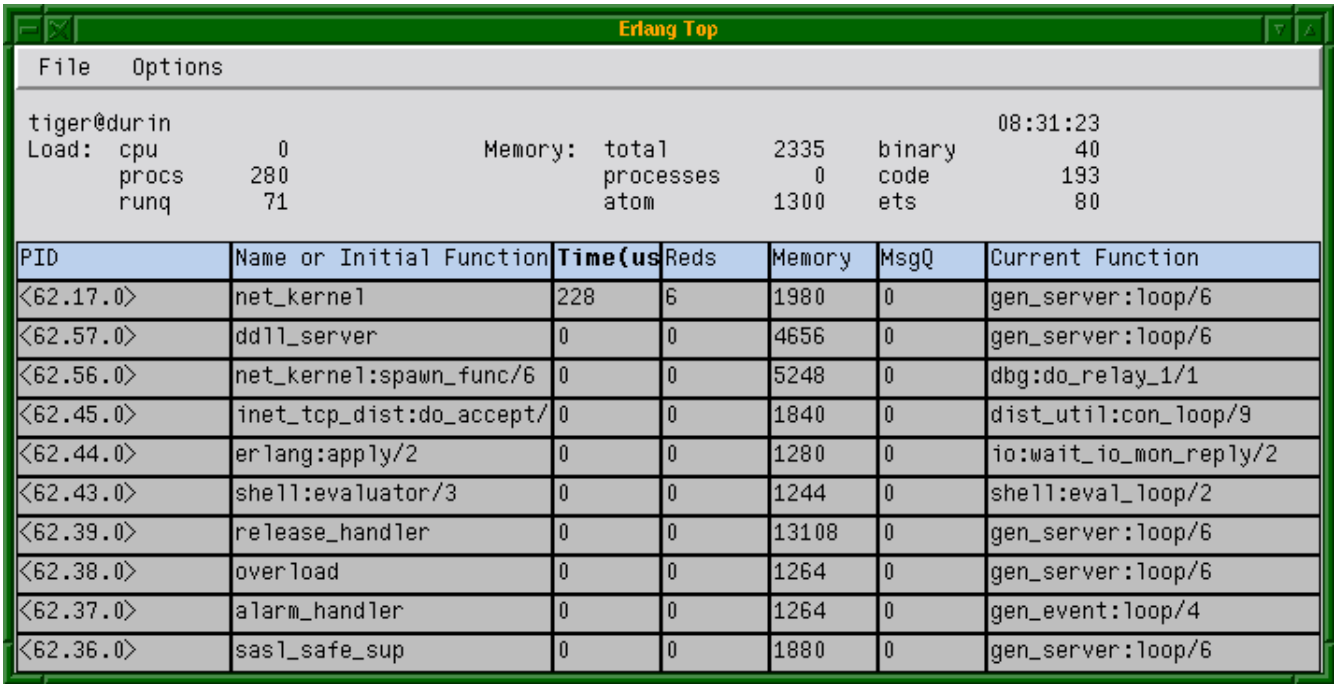

Figure 1.3: Graphical presentation of etop

The header includes some system information:

Load `cpu` is `Runtime/Wallclock`, i.e. the percentage of time where the node has been active, `procs` is the number of processes on the node, and `runq` is the number of processes that are ready to run.

Memory This is the memory allocated by the node in kilo bytes.

For each process the following information is presented:

Time This is the runtime for the process, i.e. the actual time the process has been scheduled in.

Reds This is the number of reductions that has been executed on the process

Memory This is the size of the process in bytes, obtained by a call to `process_info(Pid,memory)`.

MsgQ This is the length of the message queue for the process.

> Note:
> *Time* and *Reds* can be presented as accumulated values or as values since last update.

1.2.3 Start

To start etop with the graphical presentation, use the script `getop` or the batch file `getop.bat`, e.g. `getop -node tiger@durin`

To start etop with the text based presentation use the script `etop` or the batch file `etop.bat`, e.g. `etop -node tiger@durin`,

1.2.4 Configuration

All configuration parameters can be set at start by adding `-OptName Value` to the command line, e.g. `etop -node tiger@durin -setcookie mycookie -lines 15`.

The parameters `lines`, `interval`, `accumulate` and `sort` can be changed during runtime. Use the *Options* menu with the graphical presentation or the function `etop:config/2` with the text based presentation.

A list of all valid configuration parameters can be found in the reference manual for `etop`.

Note that it is even possible to change which information to sort by by clicking the header line of the table in the graphical presentation.

Example: Change configuration with graphical presentation

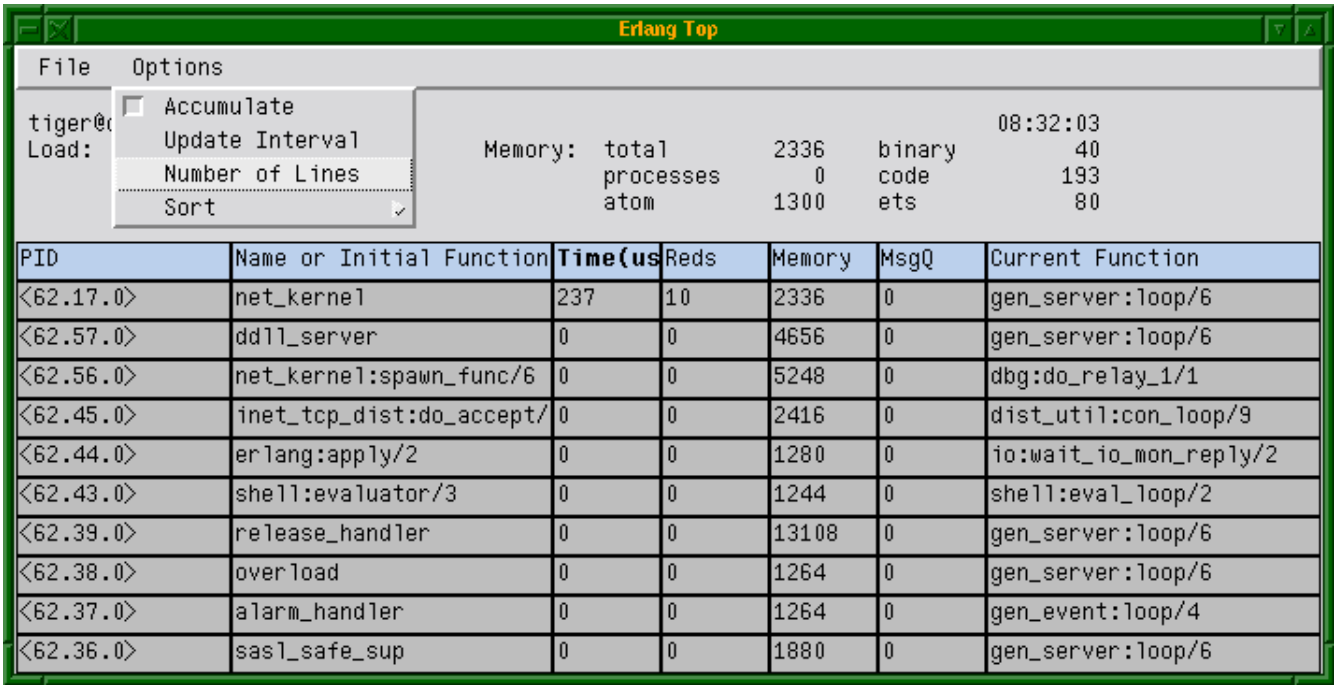

Figure 1.4: Select the option to change from the Options menu.

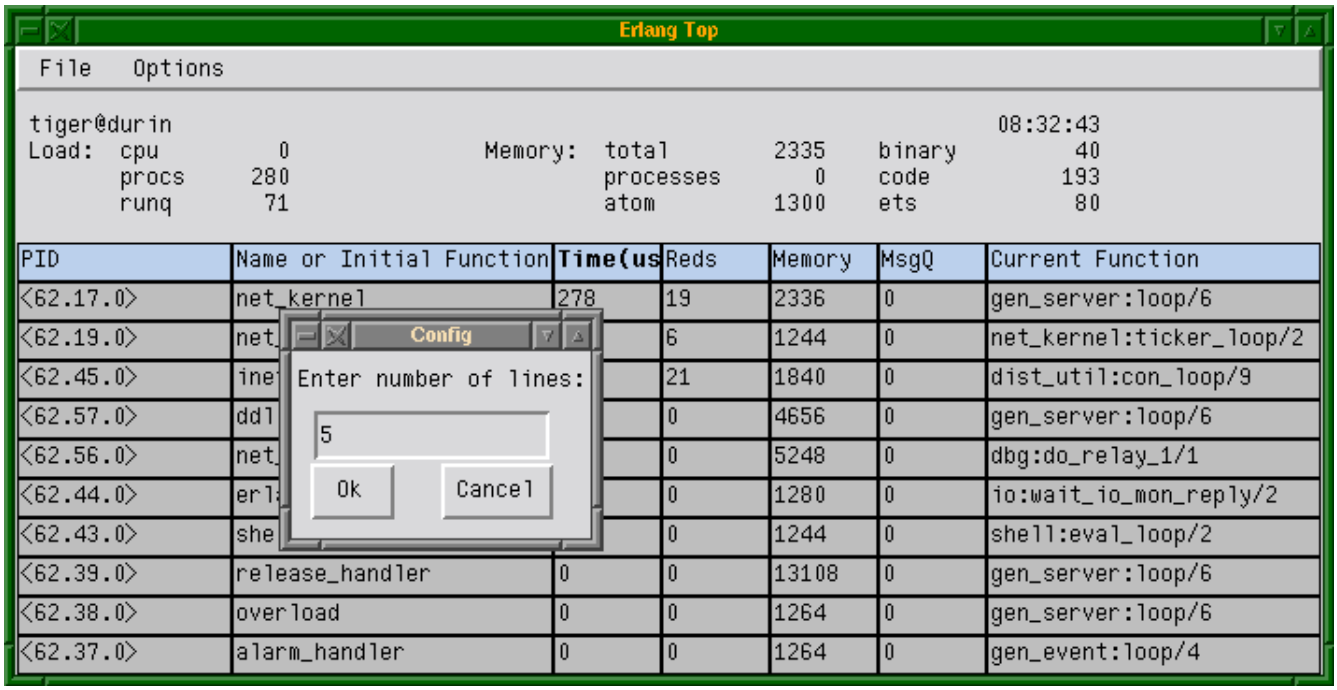

Figure 1.5: Enter the new value in the popup window and click ”Ok”

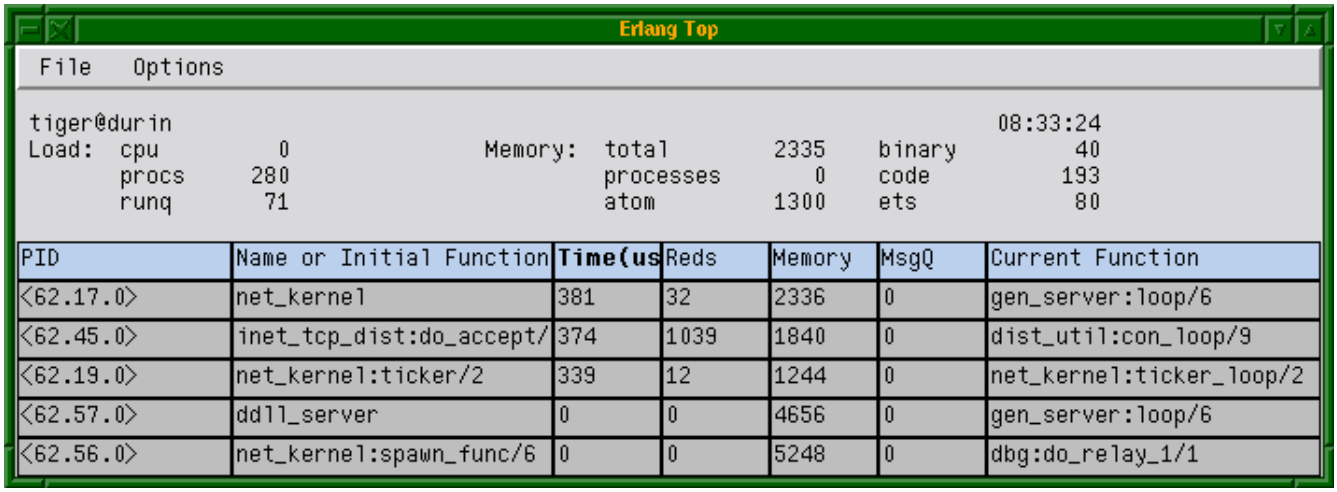

Figure 1.6: The interface is updated with the new configuration

Example: Change configuration with text based presentation

```
==============================================================================
 tiger@durin                                                      10:12:39
 Load:  cpu         0               Memory:  total        1858    binary          33
```

```
        procs     191                       processes       0    code          173
        runq        2                       atom         1002    ets            95

Pid             Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
----------------------------------------------------------------------------------------
<127.23.0>      code_server                0   60350   71176       0 gen_server:loop/6
<127.21.0>      file_server_2              0   36380   44276       0 gen_server:loop/6
<127.2.0>       erl_prim_loader            0   27962    3740       0 erl_prim_loader:loop
<127.17.0>      net_kernel                 0   13808    3916       0 gen_server:loop/6
<127.9.0>       kernel_sup                 0    6998    4676       0 gen_server:loop/6
<127.0.0>       init                       0    4156    4352       0 init:loop/1
<127.18.0>      inet_tcp_dist:accept       0    2196    1416       0 prim_inet:accept0/2
<127.16.0>      auth                       0    1893    1264       0 gen_server:loop/6
<127.43.0>      ddll_server                0     582    3744       0 gen_server:loop/6
<127.5.0>       application_controll       0     569    6756       0 gen_server:loop/6
========================================================================================

etop:config(lines,5).
ok

(etop@durin)2>
========================================================================================
 tiger@durin                                                              10:12:44
 Load:  cpu         0               Memory:  total        1859    binary         33
        procs     192                        processes       0    code          173
        runq        2                        atom         1002    ets            95

Pid             Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
----------------------------------------------------------------------------------------
<127.17.0>      net_kernel               183      70    4092       0 gen_server:loop/6
<127.335.0>     inet_tcp_dist:do_acc     141      22    1856       0 dist_util:con_loop/9
<127.19.0>      net_kernel:ticker/2      155       6    1244       0 net_kernel:ticker1/2
<127.341.0>     net_kernel:do_spawn_       0       0    5840       0 dbg:do_relay_1/1
<127.43.0>      ddll_server                0       0    3744       0 gen_server:loop/6
========================================================================================
```

1.2.5 Print to file

At any time, the current `etop` display can be dumped to a text file. Use *Dump to file* on the *File* menu with the grapical presentation or the function `etop:dump/1` with the text based presentation.

1.2.6 Stop

To stop `etop`, use *Exit* on the *File* menu for the graphical presentation, or the function `etop:stop/0` with the text based presentation.

1.3 Crashdump Viewer

1.3.1 Introduction

The Crashdump Viewer is an HTML based tool for browsing Erlang crashdumps. Crashdump Viewer runs under the WebTool application.

1.3.2 Getting Started

From an erlang node, start Crashdump Viewer by calling `crashdump_viewer:start()`. This will automatically start WebTool and display the web address where WebTool can be found. See the documentation for the WebTool application for further information about how to use WebTool.

Point your web browser to the address displayed, and you should now see the start page of WebTool. At the top of the page, you will see a link to "CrashDumpViewer". Click this link to get to the start page for Crashdump Viewer. (Note that if webtool is on localhost, you must configure your web browser to have direct connection to the internet, or you must set no proxy for localhost.)

You can also start WebTool, Crashdump Viewer and a browser in one go by running the `start_webtool` script found in the `priv` directory of the WebTool application, e.g.

```
>start_webtool crashdump_viewer
```

From the start page of Crashdump Viewer, push the "Load Crashdump" button to load a crashdump into the tool. Then enter the filename of the crashdump in the entry field and push the "Ok" button.

Crashdumps generated by OTP R9C and later are loaded directly into the Crashdump Viewer, while dumps from earlier releases first are translated by the Crashdump Translater. The Crashdump Translater creates a new file with the same name as the original crashdump, but with the extension `.translated`. If there is no write access to the directory of the original file, you will be asked to enter a new path and filename for the translated file.

1.3.3 Navigating

The lefthand frame contains a menu. Menu folders can be expanded and collapsed by clicking the folder picture. When a menu item is clicked, the item information is shown in the big information frame.

The filename frame above the information frame shows the full name of the currently viewed Erlang crashdump.

To load a new crashdump, click the "Load New Crashdump" button in the menu frame.

The various information shown in the information frame will contain links to process identifiers (PIDs) and port identifiers. Clicking one of these links will take you to the detailed information page for the process or port in question. Use the "Back" button in your browser to get back to the startingpoint. If the process or port resided on a remote node, there will be no information available. Clicking the link will then take you to the information about the remote node.

1.3.4 Help

Further help on how to use the Crashdump Viewer tool can be found in the tool's menu under 'Documentation':

'Crashdump Viewer help' is a short document describing each information page and any additional information that might occur, compared to the raw dump described in 'How to interpret Erlang crashdumps'.

'How to interpret Erlang crashdumps' is a document from the Erlang runtime system describing details in the raw crashdumps. Here you will also find information about each single field in the different information pages. This document can also be found directly in the OTP online documentation, via the Erlang runtime system user's guide.

OBSERVER Reference Manual

Short Summaries

- Application observer [page 25] – The Observer Application
- Erlang Module crashdump_viewer [page 26] – A HTML based tool for browsing Erlang crashdumps.
- Erlang Module etop [page 27] – Erlang Top is a tool for presenting information about erlang processes similar to the information presented by ”top” in UNIX.
- Erlang Module ttb [page 29] – A base for building trace tools for distributed systems.

observer

No functions are exported.

crashdump_viewer

The following functions are exported:

- `start() -> ok`
  [page 26] Start the crashdump_viewer
- `stop() -> ok`
  [page 26] Stop the crashdump_viewer

etop

The following functions are exported:

- `config(Key,Value) -> Result`
  [page 28] Change tool's configuration
- `dump(File) -> Result`
  [page 28] Dump the current display to a file.
- `stop() -> stop`
  [page 28] Terminate etop

ttb

The following functions are exported:

- `tracer() -> Result`
  [page 29] This is equivalent to tracer(node()).
- `tracer(Nodes) -> Result`
  [page 29] This is equivalent to tracer(Nodes,[]).
- `tracer(Nodes,Opts) -> Result`
  [page 29] Start a trace port on each given node.
- `p(Procs,Flags) -> Return`
  [page 30] Sets the given trace flags on the given processes.
- `tp, tpl, ctp, ctpl, ctpg`
  [page 30] Set and clear trace patterns.
- `list_history() -> History`
  [page 30] Returns all calls stored in history
- `run_history(N) -> ok | {error, Reason}`
  [page 30] Executes one entry of the history
- `write_config(ConfigFile,Config)`
  [page 31] Equivalent to write_config(ConfigFile,Config,[]).
- `write_config(ConfigFile,Config,Opt) -> ok | {error,Reason}`
  [page 31] Creates a config file.
- `run_config(ConfigFile) -> ok | {error,Reason}`
  [page 31] Executes all entries in a config file.
- `run_config(ConfigFile,NumList) -> ok | {error,Reason}`
  [page 31] Executes selected entries from a config file.
- `list_config(ConfigFile) -> Config | {error,Reason}`
  [page 31] Lists all entries in a config file.
- `write_trace_info(Key,Info) -> ok`
  [page 32] Writes any information to the `.ti` file.
- `seq_trigger_ms() -> MatchSpec`
  [page 32] Equivalent to seq_trigger_ms(all)
- `seq_trigger_ms(Flags) -> MatchSpec`
  [page 32] Returns a match_spec() which starts sequential tracing
- `stop()`
  [page 33] Equivalent to stop([])
- `stop(Opts) -> stopped`
  [page 33] Stop tracing and fetch/format logs from all nodes
- `format(File)`
  [page 33] Same as `format(File,[])`.
- `format(File,Options) -> ok | {error, Reason}`
  [page 33] Format a binary trace log

observer

Application

This chapter describes the *OBSERVER* application in OTP, which provides tools for tracing and investigation of distributed systems.

Configuration

There are currently no configuration parameters available for this application.

SEE ALSO

crashdump_viewer

Erlang Module

The Crashdump Viewer is an HTML based tool for browsing Erlang crashdumps. Crashdump Viewer runs under the WebTool application.

Exports

`start() -> ok`

This function starts the `crashdump_viewer`.

`stop() -> ok`

This function stops the `crashdump_viewer`.

etop

Erlang Module

`etop` should be started with the provided scripts `etop` and `getop` for text based and graphical presentation respectively. Under Windows the batch files `etop.bat` and `getop.bat` can be used.

All interaction with `etop` when running the graphical presentation should happen via the menus. For the text based presentation the functions described below can be used.

The following configuration parameters exist for `etop`.

node The measured node.
Value: atom()
Mandatory

setcookie Cookie to use for the etop node - must be the same as the cookie on the measured node.
Value: atom()

lines Number of lines (processes) to display.
Value: integer()
Default: 10

interval The time interval (in seconds) between each update of the display.
Value: integer()
Default: 5

accumulate If `true` the execution time and reductions are accumulated.
Value: boolean()
Default: `false`

sort Identifies what information to sort by.
Value: `runtime | reductions | memory | msg_q`
Default: `runtime` (`reductions` if `tracing=off`)

tracing `etop` uses the erlang trace facility, and thus no other tracing is possible on the measured node while `etop` is running, unless this option is set to `off`. Also helpful if the `etop` tracing causes too high load on the measured node. With tracing off, runtime is not measured.
Value: `on | off`
Default: `on`

Exports

`config(Key,Value) -> Result`

Types:

- Result = ok | {error,Reason}
- Key = lines | interval | accumulate | sort
- Value = term()

This function is used to change the tool's configuration parameters during runtime. The table above indicates the allowed values for each parameter.

`dump(File) -> Result`

Types:

- Result = ok | {error,Reason}
- File = string()

This function dumps the current display to a text file.

`stop() -> stop`

This function terminates `etop`.

ttb

Erlang Module

The Trace Tool Builder `ttb` is a base for building trace tools for distributed systems.

When using `ttb`, `dbg` shall not be used in parallel.

Exports

`tracer() -> Result`

This is equivalent to `tracer(node())`.

`tracer(Nodes) -> Result`

This is equivalent to `tracer(Nodes,[])`.

`tracer(Nodes,Opts) -> Result`

Types:

- Result = {ok, ActivatedNodes} | {error,Reason}
- Nodes = atom() | [atom()] | all | existing | new
- Opts = [Opt]
- Opt = {file,Client} | {handler, FormatHandler} | {process_info,PI}
- Client = File | {local, File}
- File = Filename | Wrap
- Filename = string()
- Wrap = {wrap,Filename} | {wrap,Filename,Size,Count}
- FormatHandler = See format/2
- PI = true | false

This function starts a file trace port on all given nodes and also points the system tracer for sequential tracing to the same port.

The given `Filename` will be prefixed with the node name. Default `Filename` is “ttb”.

`File={wrap,Filename,Size,Count}` can be used if the size of the trace logs must be limited. Default values are `Size=128*1024` and `Count=8`.

When tracing diskless nodes, `ttb` must be started from an external “trace control node” with disk access, and `Client` must be `{local, File}`. All trace information is then sent to the trace control node where it is written to file.

The `process_info` option indicates if process information should be collected. If `PI = true` (which is default), each process identifier `Pid` is replaced by a tuple `{Pid,ProcessInfo,Node}`, where `ProcessInfo` is the process' registered name its

globally registered name, or its initial function. It is possible to turn off this functionality by setting `PI = false`.

`p(Procs,Flags) -> Return`

Types:

- Return = {ok,[{Procs,MatchDesc}]}
- Procs = Process | [Process] | all | new | existing
- Process = pid() | atom() | {global,atom()}
- Flags = Flag | [Flag]

This function sets the given trace flags on the given processes.

Please turn to the Reference manual for module `dbg` for details about the possible trace flags. The parameter `MatchDesc` is the same as returned from `dbg:p/2`

Processes can be given as registered names, globally registered names or process identifiers. If a registered name is given, the flags are set on processes with this name on all active nodes.

`tp, tpl, ctp, ctpl, ctpg`

These functions should be used in combination with the `call` trace flag for setting and clearing trace patterns. When the `call` trace flag is set on a process, function calls will be traced on that process if a trace pattern has been set for the called function. Trace patterns specifies how to trace a function by using match specifications. Match specifications are described in the User's Guide for the erlang runtime system `erts`.

These functions are equivalent to the corresponding functions in `dbg`, but all calls are stored in the history. The history buffer makes it easy to create config files so that the same trace environment can be setup several times, e.g. if you want to compare two test runs. It also reduces the amount of typing when using `ttb` from the erlang shell.

`tp` Set trace pattern on global function calls

`tpl` Set trace pattern on local and global function calls

`ctp` Clear trace pattern on local and global function calls

`ctpl` Clear trace pattern on local function calls

`ctpg` Clear trace pattern on global function calls

`list_history() -> History`

Types:

- History = [{N,Func,Args}]

All calls to `ttb` is stored in the history. This function returns the current content of the history. Any entry can be re-executed with `run_history/1` or stored in a config file with `write_config/2/3`.

`run_history(N) -> ok | {error, Reason}`

Types:

- N = integer() | [integer()]

Executes the given entry or entries from the history list. History can be listed with `list_history/0`.

`write_config(ConfigFile,Config)`

Equivalent to `write_config(ConfigFile,Config,[])`.

`write_config(ConfigFile,Config,Opt) -> ok | {error,Reason}`

Types:

- ConfigFile = string()
- Config = all | [integer()] | [{Mod,Func,Args}]
- Mod = atom()
- Func = atom()
- Args = [term()]
- Opt = [] | [append]

This function creates or extends a config file which can be used for restoring a specific configuration later.

The content of the config file can either be fetched from the history or given directly as a list of `{Mod,Func,Args}`.

If the complete history is to be stored in the config file `Config` should be `all`. If only a selected number of entries from the history should be stored, `Config` should be a list of integers pointing out the entries to be stored.

If `Opt` is not given or if it is `[]`, `ConfigFile` is deleted and a new file is created. If `Opt = [append]`, `ConfigFile` will not be deleted. The new information will be appended at the end of the file.

`run_config(ConfigFile) -> ok | {error,Reason}`

Types:

- ConfigFile = string()

Executes all entries in the given config file.

`run_config(ConfigFile,NumList) -> ok | {error,Reason}`

Types:

- ConfigFile = string()
- NumList = [integer()]

Executes selected entries from the given config file. `NumList` is a list of integers pointing out the entries to be executed.

The content of a config file can be listed with `list_config/1`.

`list_config(ConfigFile) -> Config | {error,Reason}`

Types:

- ConfigFile = string()
- Config = [{N,Func,Args}]

Lists all entries in the given config file.

`write_trace_info(Key,Info) -> ok`

Types:

- Key = term()
- Info = Data | fun() -> Data
- Data = term()

The `.ti` file contains {`Key`,`ValueList`} tuples. This function adds `Data` to the ValueList associated with `Key`. All information written with this function will be included in the call to the format handler.

`seq_trigger_ms() -> MatchSpec`

Equivalent to `seq_trigger_ms(all)`

`seq_trigger_ms(Flags) -> MatchSpec`

Types:

- MatchSpec = match_spec()
- Flags = all | SeqTraceFlag | [SeqTraceFlag]
- SeqTraceFlag = atom()

A match specification can turn on or off sequential tracing. This function returns a match specification which turns on sequential tracing with the given `Flags`.

This match specification can be given as the last argument to `tp` or `tpl`. The activated `Item` will then become a *trigger* for sequential tracing. This means that if the item is called on a process with the `call` trace flag set, the process will be "contaminated" with the seq_trace token.

If `Flags = all`, all possible flags are set.

Please turn to the reference manual for the `seq_trace` module in the `kernel` application to see the possible values for `SeqTraceFlag`. For a description of the match_spec() syntax, please turn to the *User's guide* for the runtime system (*erts*). The chapter *Match Specification in Erlang* explains the general match specification "language".

> **Note:**
> The *system tracer* for sequential tracing is automatically initiated by `ttb` when a trace port is started with `ttb:tracer/0/1/2`.

Example of how to use the `seq_trigger_ms/0/1` function:

```
(tiger@durin)5> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)6> ttb:p(all,call).
{ok,{[all],[call]}}
(tiger@durin)7> ttb:tp(mod,func,ttb:seq_trigger_ms()).
{ok,[{matched,1},{saved,1}]}
(tiger@durin)8>
```

Whenever `mod:func(...)` is called after this, the seq_trace token will be set on the executing process.

`stop()`

Equivalent to `stop([])`.

`stop(Opts) -> stopped`

Types:

- Opts = [Opt]
- Opt = fetch | format

Stops tracing on all nodes.

The `fetch` option indicates that trace logs shall be collected from all nodes after tracing is stopped. This option is useful if nodes on remote machines are traced. Logs and trace information files are then sent to the trace control node and stored in a directory named `ttb_upload-Timestamp`, where `Timestamp` is on the form `yyyymmdd-hhmmss`. Even logs from nodes on the same machine as the trace control node are moved to this directory.

The `format` option indicates that the trace logs shall be formatted after tracing is stopped. Note that this option also implies the `fetch` option, i.e. logs are collected in a new directory on the trace control node before formatting. All logs in the directory will be merged.

`format(File)`

Same as `format(File,[])`.

`format(File,Options) -> ok | {error, Reason}`

Types:

- File = string() | [string()]
  This can be the name of a binary log, a list of such logs or the name of a directory containing one or more binary logs.
- Options = [Opt]
- Opt = {out,Out} | {handler,FormatHandler}
- Out = standard_io | string()
- FormatHandler = {Function, InitialState} | et
- Function = fun(Fd,Trace,TraceInfo,State) -> State
- Fd = standard_io | FileDescriptor
  This is the file descriptor of the destination file `Out`
- Trace = tuple()
  This is the trace message. Please turn to the Reference manual for the `erlang` module for details.
- TraceInfo = [{Key,ValueList}]
  This includes the keys `flags`, `client` and `node`, and if `handler` is given as option to the tracer function, this is also included. In addition all information written with the `write_trace_info/2` function is included.

Reads the given binary trace log(s). If a directory or a list of logs is given and the `timestamp` flag was set during tracing, the trace messages from the different logs are merged according to the timestamps.

If `FormatHandler = {Function,InitialState}`, `Function` will be called for each trace message. If `FormatHandler = et`, `et_viewer` in the *Event Tracer* application (`et`) is used for presenting the trace log graphically. `ttb` provides a few different filters which can be selected from the Filter menu in the `et_viewer`. If `FormatHandler` is not given, a default handler is used which presents each trace message as a line of text.

If `Out` is given, `FormatHandler` gets the filedescriptor to `Out` as the first parameter.

`Out` is ignored if `FormatHandler = et`.

Wrap logs can be formatted one by one or all in one go. To format one of the wrap logs in a set, give the exact name of the file. To format the whole set of wrap logs, give the name with '*' instead of the wrap count. See examples in the `ttb` User's Guide.

List of Figures

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.

write_config/2
ttb , 31

write_config/3
ttb , 31

write_trace_info/2
ttb , 32