dbg/ttb

The Erlang text based tracer and trace tool builder

Mazen Harake
mazen.harake@erlang-solutions.com

Erlang Solutions

0

Index

What we will cover ...

1. dbg Introduction and Basics
2. Match Specifications
3. dbg Extended
4. ttb Introduction and Basics
5. Where to go from here ...

1

dbg Introduction and Basics

Getting started

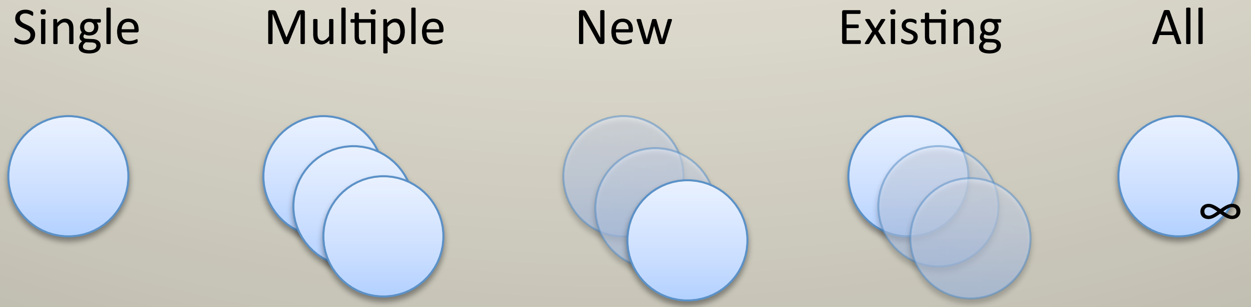
dbg Introduction and Basics

`dbg` is ...

- A textbased tracer
- Suitable for low level tracing in the shell
- Built on the trace BIFs (no “trace-compile”)
- Small system impact (if done right)
- Flexible and extendable
- Part of the `runtime_tools` application

dbg Introduction and Basics

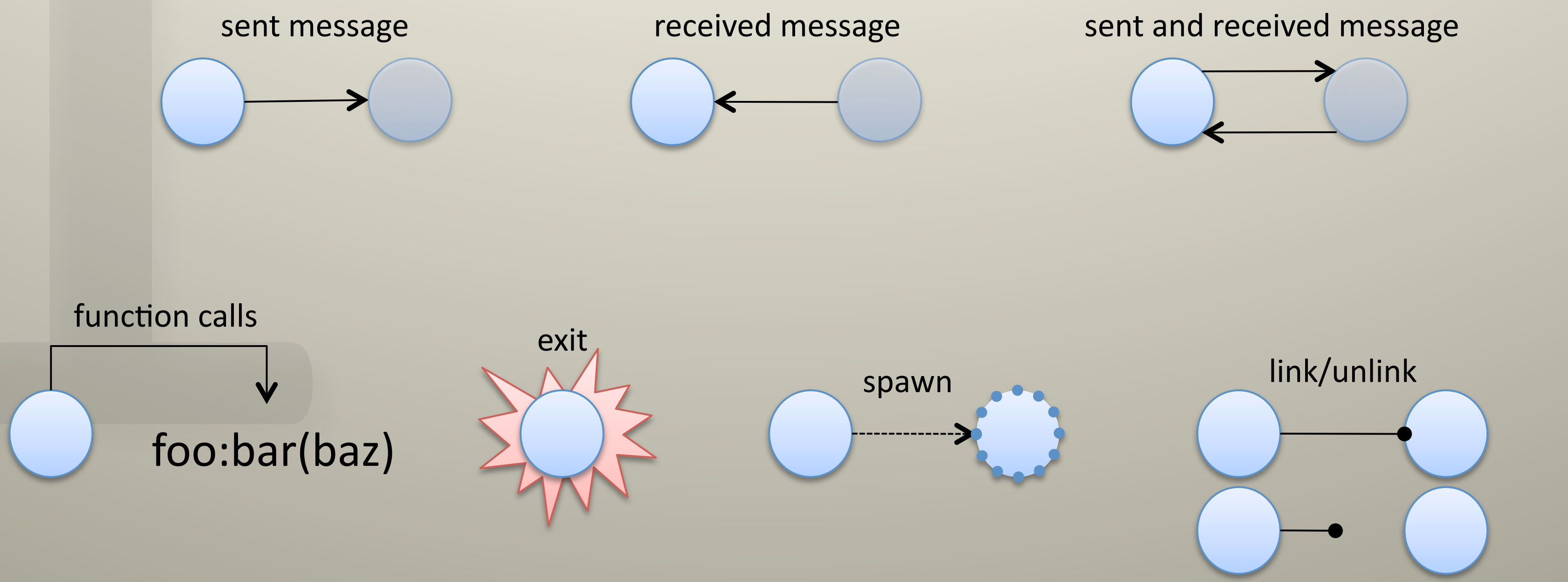
First of all:
We only trace processes!



We **don’t** trace functions!

dbg Introduction and Basics

Each trace has flags for what behaviour to trace on, E.g ...



sent message

received message

sent and received message

function calls

foo:bar(baz)

exit

spawn

link/unlink

and more ...

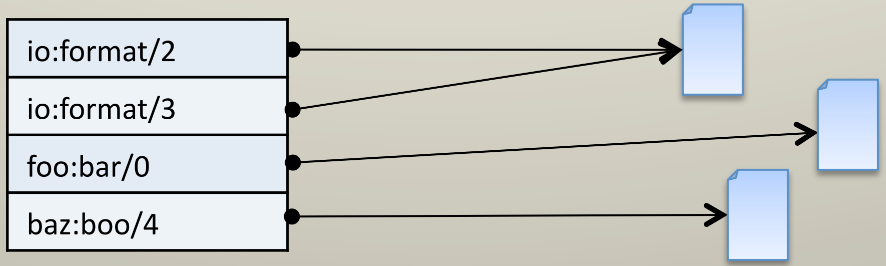
dbg Introduction and Basics

A trace emits trace messages,
one process receives these messages

This process is usually the `dbg` process
(but can be another one)

dbg Introduction and Basics

When tracing calls, a pattern table is created to know which module, function and arity to react on



io:format/2

io:format/3

foo:bar/0

baz:boo/4

A match specification is then used to perform some additional checks, filtrations and side effects *when a trace has been triggered*

dbg Introduction and Basics

A typical way to use `dbg`

1. Start the tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Observe trace
5. Clear the traces
6. Stop the tracer process

dbg Introduction and Basics

1. Start the tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Observe trace
5. Clear the traces
6. Stop the tracer process

`dbg:tracer()`

- Starts a process that will recieve all the trace messages
- Only one can be active on local node
- Only one will receive trace messages in a cluster
- No options simply prints out the trace using `io:format`

```
> dbg:tracer().
{ok,<0.33.0>}
```

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:p(Item, FlagList)
```

- Defines a trace for a process (`p` for process?)
- `Item` is a term that identifies one or more processes
- `FlagList` is a list of options to enable in the trace
- Comes into effect immediately

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

`dbg:p(`**`Item,`**` FlagList)`

`Item` can be one of the following:

- `pid()`
- `all`, `new` or `existing`
- A Registered name (except for mentioned keywords)
- An integer (meaning `<0.int().0>`)
- `{X,Y,Z}` or "`<X.Y.Z>`" (meaning `<X.Y.Z>`)

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:p(Item, FlagList)
```

`FlagList` is a list of zero or more of:

- `s` – Sending messages
- `r` – Receiving messages
- `m` – Sending and receiving messages
- `c` – Calls to functions
- `p` – Process related events (spawn, links etc)
- `so[f]s` – Set on [first] spawn. Inherit flags for new processes
- `so[f]l` – Set on [first] link. Inherit flags when linking new process
- `all` – Set all flags
- `clear` – Clear all flags

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:p(Item, FlagList)
```

`FlagList` can also include what ever `erlang:trace/2` accepts:

- `running` – Process scheduling
- `garbage_collection` – When GC occurs
- `timestamp` – Attach a timestamp to the trace
- `arity` – Shows arity instead of argument

See documentation for more options

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:p(Item, FlagList)
```

- returns `{ok, [{matched, Node, N}]}`
- Shows how many processes `N` that matched on each node `Node`
- More on how to add nodes later.

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

`dbg:p(Item, FlagList)`

```
> dbg:p(self(), [m, timestamp]).
...
{ok,[{matched,nonode@nohost,1}]}
```

```
> dbg:p(self(), [garbage_collection]).
{ok,[{matched,nonode@nohost,1}]}
> ls().
...
```

```
> dbg:p(all, [c, timestamp]).
{ok,[{matched,nonode@nohost,26}]}
```

Use the flag `clear` in between the examples

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

A word of advice ...

Know what you are tracing, think before you trace!

```
> dbg:p(all, [m]).
...
```

Before you run this ... tell me what you think it does.

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:tp({Module, Function, Arity}, MatchSpec)
```

- Defines a trace pattern (`tp`) for global calls
- For local calls use `dbg:tpl`
- Only useful when used together with the `call` flag
- Multiple trace patterns can be defined

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:tp({Module, Function, Arity}, MatchSpec)
```

- Module has to be specified
- Wildcards are `{M,F,'_'}` and `{M,'_','_'}`
- No other combination allowed e.g. `{'_',F, '_'}`

Alternative API

```
dbg:tp(Module, Function, Arity, MatchSpec)
dbg:tp(Module, Function, MatchSpec)
dbg:tp(Module, MatchSpec)
```

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

`dbg:tp({Module, Function, Arity}, MatchSpec)`

- Will be described later. Use `[]` for now.

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:tp({Module, Function, Arity}, MatchSpec)
```

- returns `{ok, [{matched, Node, N}]}`
- Shows how many functions `N` that matched on each node `Node`
- More on how to add nodes later.

dbg Introduction and Basics

1. Start *the* tracer process
2. Specify which processes to trace and how (using flags)
3. Specify *trace patterns* and *match specifications*
4. Run/Observe trace
5. Clear the traces
6. Stop the tracer process

```
dbg:ctp({Module, Function, Arity})
dbg:p(Item, clear)
```

- First command clears the patterns but not the traces
- Second command stops the traces (using the `clear` flag)
- Return the same format as when setting traces/patterns I.e. `{ok, [{matched, Node, N}]}`
- Use `dbg:stop/0`, `dbg:stop_clear/0` to stop the tracer process

Exercises

1. Compile the file `traceme.erl` and load the beam
2. Run: `{P1, P2} = traceme:init/0`
3. Enable tracing you think will answer the questions
4. Run `traceme:runit({P1, P2})` and then `traceme:stopit({P1, P2})`
5. Clear trace and repeat for each in the following list

Using tracing ...

- What functions in `traceme` are being called by the shell process?
- What messages that are sent to and from `P1` and `P2`?
- How often is garbage collection run on each processes
- How long does it take between the two calls the shell does to `init/2`

2

Match Specifications

What and how

Match Specifications

A Match Specification (MS) is ...

- A set of Erlang terms describing a small "program"
- The purpose is to, using this "program", to match input data
    - `dbg` call traces
    - `ets` objects

  and decide on output/actions to take
- More efficient than calling Erlang functions
- Limited in functionality and performs only a few actions
- If the MS match, one or more trace events will be sent

Match Specifications

Matching is done in three steps

```
MatchSpec = [{MatchHead, MatchConditions, MatchBody}]
```

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

`{`**`MatchHead`**`, MatchConditions, MatchBody}`

- A list of values/terms and/or variables; the length of this list must be equal to the arity of the function being matched on
- Matches (binds) variables in the form of `'$N'`
  - `N = 1 ... 100000000` (Doubt you will need it though)
- E.g. `['$1', '$2']` to match function `foo:bar("baz", 3)`
- `'_'` can be used to ignore individual arguments
  - E.g. `['_', '$2']`, `['_', '_']`, `['$1', '_']`
- `'$N'` doesn't have to be in order (E.g. 1, 2, 3 ...)
- `'_'` can be used to ignore the whole MatchHead

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

```
{MatchHead, MatchConditions, MatchBody}
```

Examples

Function call	MatchHead	Match?
`io:format("hi~n",[])`	`'_'`	`Yes`
`erlang:max(3, 5)`	`[3, '$2']`	`Yes: '$2'==5`
`erlang:now()`	`['$1']`	`No`
`gen_tcp:listen(8000, [binary])`	`['$1', '$2','$3']`	`No`
`erlang:size({one, two})`	`['_']`	`Yes`
`lists:foldl(fun foo/2, [], [1,2,3])`	`['_','$1','_']`	`Yes: '$1'==[]`
`erlang:max(6, 6)`	`['$1', '$1']`	`Yes: '$1'==6`

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

```
{MatchHead, MatchConditions, MatchBody}
```

- A list of terms to where each one is a matching condition
- Evaluates to either `true` or `false`
- Uses previously bound variables only, no new can be bound
- Only guard functions allowed: `is_integer`, `hd`, `length`, `'>='`
- Functions are specified as a tuple E.g. `{hd, '$1'}`
- Literal tuples are written as: `{{a, b}}`
- Can be nested to perform more complex expressions
- All conditions must evaluate to `true`
- `[]` == No conditions == `true`
- All function names that are language constructs must use single quotes such as `'>'`, `'=='` and `'+'`
- See documentation for full list

Match Specifications

1. Bind variables (MatchHead)
2. Check against conditions (MatchConditions)
3. Perform actions (MatchBody)

```
{MatchHead, MatchConditions, MatchBody}
```

Examples

Function arguments	MatchHead	Condition	Match?
f("",[1])	['_','$1']	[{is_integer, '$1'}]	false
f("",[6])	['_', ['$1']]	[{'<', 5, '$1'}]	true
f(52)	['$1']	[{'==',{rem, 5, '$1'}, 2}]	true
f({["4"]})	'_'	[true]	true
f({["4"]})	'_'	[false]	false
f(52)	['$1']	[{is_integer,'$1'},{'>','$1',0}]	true
f(0)	'_'	[]	true

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

```
{MatchHead, MatchConditions, MatchBody}
```

- A list of terms where each is an action to perform
- Only when `MatchHead` and `MatchConditions` succeeded
- Actions include
  - Sending messages
  - Printing to stdout
  - Enabling/Disabling additional flags/traces
  - Returning trace information
- Can include conditional expressions
- Literal tuples are written as: `{{a, b}}`
- Actions must always be tuples even if they are of arity 1

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

`{MatchHead, MatchConditions,` **`MatchBody`**`}`

Action	Description
`{message , term()}`	Appends term() to the trace message
`{return_trace}`	Generates a trace message when the call returns from the function (breaks tail-recursion)
`{enable_trace, TraceFlag} / {enable_trace, Pid, TraceFlag}`	Enables a trace for the given Pid if specified, otherwise self()
`{disable_trace, TraceFlag} / {disable_trace, Pid, TraceFlag}`	Disables a trace for the given Pid if specified, otherwise self()
`{caller}`	Returns the `{M,F,A}` of the function the current function was called from (used with `message`)
`{process_dump}`	Returns a binary textual representation about the current process (used with `message`)
`{display, term()}`	Prints term() out to the stdout

Match Specifications

1. Bind variables (MatchHead)
2. Check against conditions (MatchConditions)
3. Perform actions (MatchBody)

Match Specifications Examples

```
> dbg:tracer().
{ok,<0.430.0>}
250> dbg:p(self(), [c, arity, timestamp]).
{ok,[{matched,nonode@nohost,1}]}
251> dbg:tp(traceme, foo, [{['$1'], [{'>', '$1', 5}], []}]).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
252> traceme:foo(1). %% Not greater than 5
ok
253> traceme:foo(6). %% Greater than 5
ok
(<0.297.0>) call traceme:foo/1 (Timestamp: {1289,134618,807870})
```

Match Specifications

1. Bind variables (MatchHead)
2. Check against conditions (MatchConditions)
3. Perform actions (MatchBody)

Match Specifications Examples

```
> dbg:p(all,clear), dbg:ctp().
{ok,[{matched,nonode@nohost,8830}]}
> dbg:p(self(), [c]).
{ok,[{matched,nonode@nohost,1}]}
> dbg:tp(traceme, bar, [{['$1', '$2'], [{'andalso', {'>=', '$1', 5}, {'not',
{'is_list', '$2'}}}], []}]).
{ok,[{matched,nonode@nohost,1},{saved,2}]}
> traceme:bar(6, {hej}). %% First argument > 4 and second argument not a list
(<0.297.0>) call traceme:bar(6,{hej})
ok
> traceme:bar(6, []). %% Second argument is a list
ok
```

Match Specifications

1. Bind variables (MatchHead)
2. Check against conditions (MatchConditions)
3. Perform actions (MatchBody)

Match Specifications Examples

```
> dbg:p(all,clear), dbg:ctp().
{ok,[{matched,nonode@nohost,8830}]}
> dbg:p(self(), [c, timestamp]).
{ok,[{matched,nonode@nohost,1}]}
> dbg:tp(traceme, baz, [{['_', '_', '$7000'], [{'==', {'element', 1, '$7000'}, ok}],
[{message,{'element',2,'$7000'}}, {return_trace}]}]).
> {ok,[{matched,nonode@nohost,1},{saved,3}]}
> traceme:baz(1,2,3). %% Not tuple but doesn't crash
ok
> traceme:baz(1,2,{ok, msg}). %% Tuple and first element is ok
(<0.297.0>) call traceme:baz(1,2,{ok,msg}) (msg) (Timestamp: {1289,135992,580490})
(<0.297.0>) returned from traceme:baz/3 -> ok (Timestamp: {1289,136400,564081})
>
```

Match Specifications

1. Bind variables (`MatchHead`)
2. Check against conditions (`MatchConditions`)
3. Perform actions (`MatchBody`)

```
dbg:fun2ms(LiteralFun) -> MatchSpec
```

- `LiteralFun` is a fun which is replaced by a Match Specification at compile time
- `LiteralFun` must be declared in the call to `fun2ms`
- The fun translate to different parts of the MS
  - Funtion header translates to `MatchHead` (same rules)
  - Variables are named in a normal way (`A`, `Var` etc ...)
  - Guards allowed and translated into `MatchConditions`
  - Function body translates into `MatchBody`
  - MS functions are written as *`function()`*; E.g. `message()` and `return_trace()` etc

Match Specifications

1. Bind variables (MatchHead)
2. Check against conditions (MatchConditions)
3. Perform actions (MatchBody)

Examples

```
> dbg:fun2ms(fun([A, B]) when is_list(A) andalso is_integer(B) -> message(caller()) end).
[{['$1','$2'],
  [{'andalso',{is_list,'$1'},{is_integer,'$2'}}],
  [{message,{caller}}]}]

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

> dbg:fun2ms(fun([A, A]) -> ok end).
[{['$1','$1'],[],[ok]}]

> dbg:fun2ms(fun([A, B]) when A > B -> enable_trace(garbage_collection) end).
[{['$1','$2'],
  [{'>','$1','$2'}],
  [{enable_trace,garbage_collection}]}]
```

Do not affect tracing

Exercises

1. Compile the file `traceme.erl` and load the beam
2. Run: `{P1, P2} = traceme:init/0`
3. Enable tracing you think will answer the questions
4. Run `traceme:randit({P1, P2})` and then `traceme:stopit({P1, P2})`
5. Clear trace and repeat for each in the following list

Using tracing ...

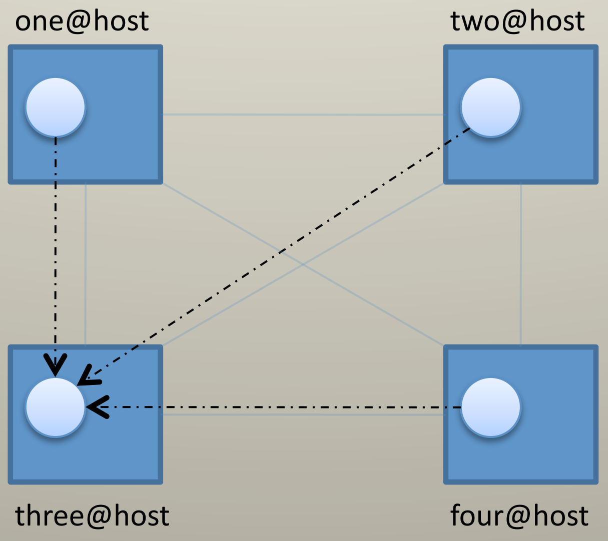
- How long do the `init/2` and `ping/1` functions take to complete (approx)
- What are the return values of the functions being called
- What data is in the process dump binary (display it or return it)
- Trace shell's calls to the traceme:rand1/1 but only show those where the first argument is greater than 400
- Trace shell's calls to the traceme:rand2/1 but only show those where the second element in the argument is less than 100

3 dbg Extended

Handling trace messages manually; tracing to a file or port and tracing in a multinode environment

dbg Extended

Distributed tracing
A cluster can be traced with output going to one node

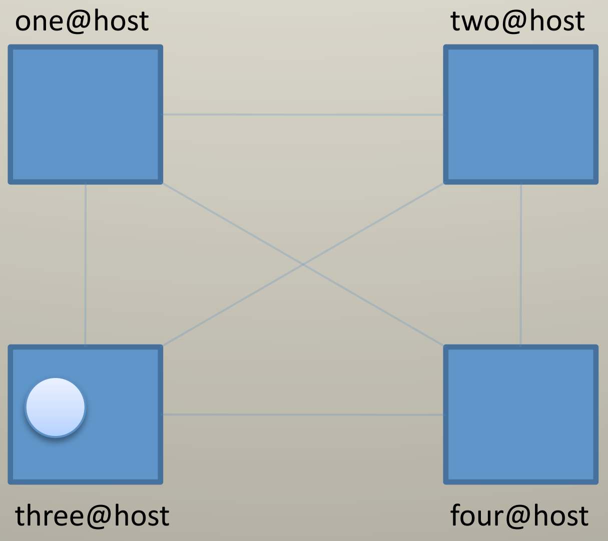


dbg Extended

Distributed tracing

Start a tracer

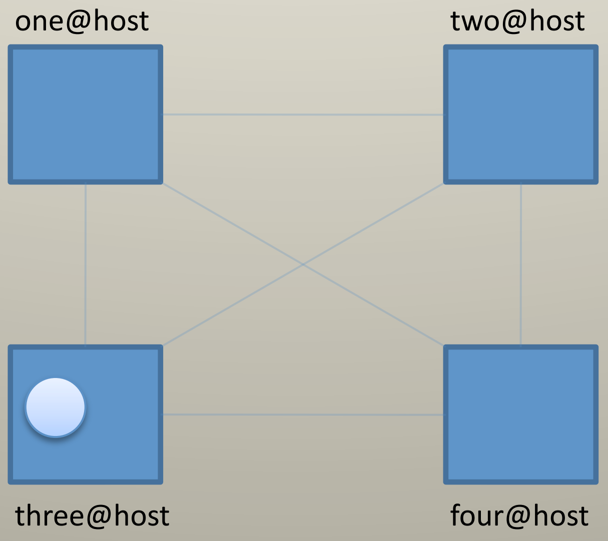
```
dbg:tracer()
```



dbg Extended

Distributed tracing
Add nodes to the list of nodes to start traces on

```
dbg:n(Node)
```



```
[one@host, two@host, four@host]
```

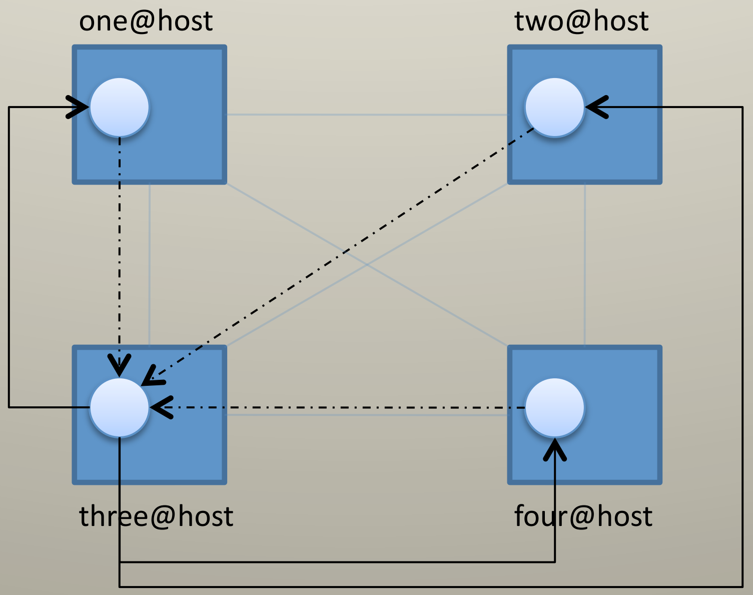
```
dbg:cn(Node) - Removes a node from list
dbg:ln() - List all nodes
```

dbg Extended

Distributed tracing

When a trace starts it will start on all known nodes

```
dbg:p(Item, Flags)
```



Executed on each node

dbg Extended

Distributed tracing; Example

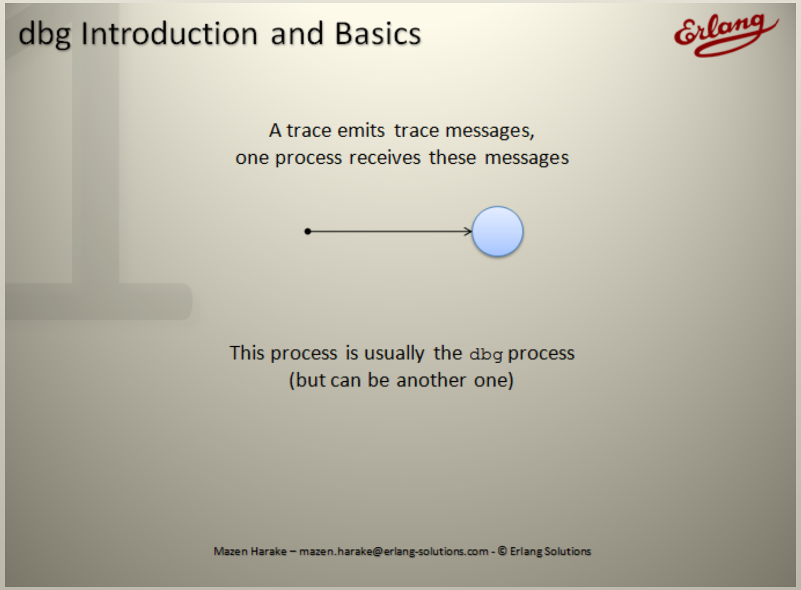
```
1> net_kernel:start([foo, shortnames]).
{ok,<0.33.0>}
(foo@Pasha)2> dbg:tracer().
{ok,<0.40.0>}
(foo@Pasha)3> dbg:n(bar@Pasha).
{ok,bar@Pasha}
(foo@Pasha)4> dbg:p(all, c).
{ok,[{matched,bar@Pasha,34},
     {matched,foo@Pasha,35}]}
(foo@Pasha)5> dbg:tp(traceme, foo, []).
{ok,[{matched,bar@Pasha,1},
     {matched,foo@Pasha,1}]}
(foo@Pasha)6>
(<6566.31.0>) call traceme:foo(bar@Pasha)
(foo@Pasha)6> traceme:foo(node()).
ok
(<0.31.0>) call traceme:foo(foo@Pasha)
(foo@Pasha)7>
```

```
1> net_kernel:start([bar, shortnames]).
{ok,<0.33.0>}
(bar@Pasha)2> traceme:foo(node()).
ok
(bar@Pasha)3>
```

Make sure all nodes have relevant code loaded when using `dbg:tp/2,3,4`

dbg Extended

Remember this?



dbg Introduction and Basics

A trace emits trace messages,
one process receives these messages

This process is usually the `dbg` process
(but can be another one)

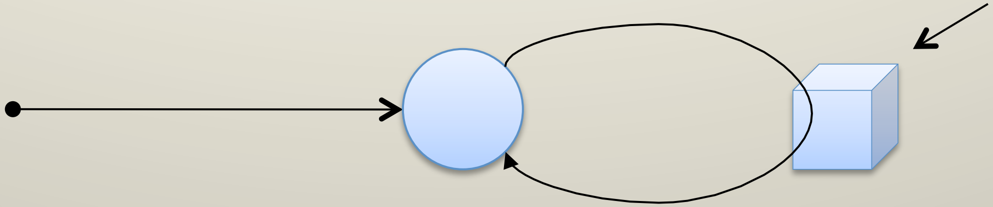
Mazen Harake – mazen.harake@erlang-solutions.com - © Erlang Solutions

The way to change that is ...

`dbg:tracer/2`

dbg Extended

Handling trace messages manually



Suppose to represent a fun

```
dbg:tracer(process, {HandlerFun, InitialState})
 HandlerFun = fun(TraceMsg, State) -> NewState
```

- `HandlerFun` is a fun of arity 2.
- First argument in `HandlerFun` is the trace message; Second argument is the custom state which initially is `InitialState`
- Side effects are allowed in the `HandlerFun`
- Value returned from `HandlerFun` is the new state

dbg Extended

Handling trace messages manually; Example

```
> dbg:tracer(process, {fun(Trace, N) -> io:format("TRACE (#~p): ~p~n", [N, Trace]), N+1
end, 0}).
{ok,<0.485.0>}
> dbg:p(self(), [c]).
{ok,[{matched,nonode@nohost,1}]}
> dbg:tp(traceme, foo, []).
{ok,[{matched,nonode@nohost,1}]}
> traceme:foo(1).
TRACE (#0): {trace,<0.297.0>,call,{traceme,foo,[1]}}
ok
> traceme:foo(2).
TRACE (#1): {trace,<0.297.0>,call,{traceme,foo,[2]}}
Ok
> dbg:p(all,clear).
{ok,[{matched,nonode@nohost,28}]}
> dbg:p(self(), [c, timestamp]).
{ok,[{matched,nonode@nohost,1}]}
> traceme:foo(10).
ok
TRACE (#2): {trace_ts,<0.297.0>,call,{traceme,foo,"\n"},{1289,139850,193370}}
>
```

dbg Extended

Trace messages; examples

```
{trace, Pid, 'receive', Msg}

{trace, Pid, send, Msg, To}

{trace, Pid, call, {M, F, Args}}

{trace, Pid, return_from, {M, F, Arity}, ReturnValue}

{trace, Pid, exception_from, {M, F, Arity}, {Class, Value}}

{trace, Pid, spawn, Pid2, {M, F, Args}}
```

dbg Extended

Trace messages; with flag `timestamp`:

```
{trace_ts, Pid, call, {M, F, A}, {MegaSec, Sec, MicroSec}}
```

with `message` in the match specification:

```
{trace, Pid, call, {M, F, Args}, Message}
```

With both:

```
{trace_ts, Pid, call, {M, F, Args}, Message, {MegaSec, Sec, MicroSec}}
```

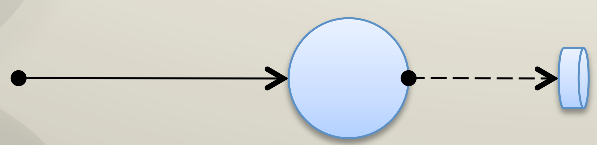
Using flag `arity`:

```
{trace, Pid, call, {M, F, N}}
```

See `erlang:trace/2` for detailed information

dbg Continued

Trace output to a port

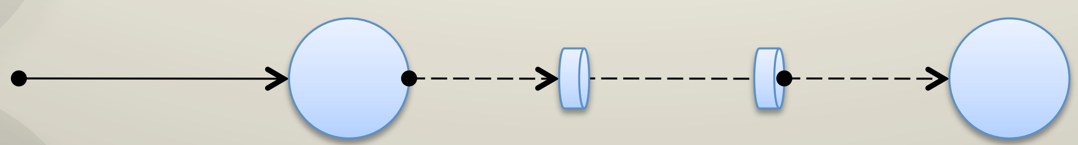


```
dbg:tracer(port, PortGenerator)
```

- Sends trace messages to a port (a file or a socket)
- Much more efficient way of tracing (low overhead)
- `PortGenerator` is a `fun/0` which opens a file or a socket port
- `trace_port/2` should be used to create that fun

dbg Continued

Trace output to a port: Socket



```
trace_port(ip, PortNumber|{PortNumber, QueueSize})
```

- Listens on port for a client to connect
- `PortNumber` is the port to listen on
- `QueueSize` is the max number of messages which will be buffered before trace messages are being discarded
- Any client can connect but should preferably be connecting using `trace_client/2` from an Erlang shell

dbg Continued

Trace output to a port: Socket

```
trace_client(ip, PortNumber|{Hostname, PortNumber})
```

- Connects to a port on a hostname
- Printouts have the same formats as tracing in the shell
- Any client can be used but this is preferred

Warning: Will disconnect silently if connection is broken
or unable to connect

dbg Continued

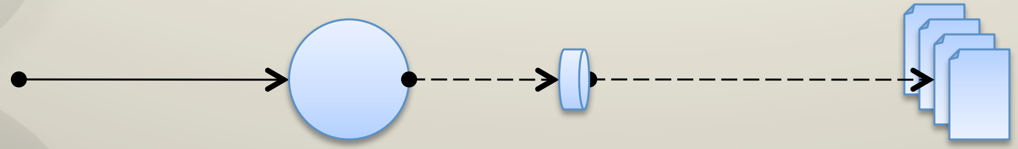
Trace output to a port: Socket; example

```
> dbg:tracer(port,
             dbg:trace_port(ip, 9922)).
{ok,<0.80.0>}
> dbg:p(all, c).
{ok,[{matched,nonode@nohost,26}]}
> dbg:tp(traceme, foo, []).
{ok,[{matched,nonode@nohost,1}]}
> traceme:foo(node()).
ok
> traceme:foo("Hello!").
ok
```

```
> dbg:trace_client(ip, {"localhost", 9922}).
<0.53.0>
> (<0.33.0>) call traceme:foo(nonode@nohost)
(<0.33.0>) call traceme:foo("Hello!")
```

dbg Continued

Trace output to a port: File



`trace_port(file, Filename|WrapFileSpec)`

- Filename is the file to dump traces to
- Using WrapFileSpec one can rotate files based on count and size/time and give it a suffix
- It is recommended to use `trace_client/2` from an Erlang shell read these files (they are in binary format)

dbg Continued

Trace output to a port: File

```
trace_client(file|follow_file, Filename)
```

- Opens the file Filename and prints the tracemessages
- If `follow_file` is used then the file will be continously read (like `tail -f filename` in *nix systems)
- Does not "cross over" to read files when they wrap

dbg Continued

Trace output to a port

```
dbg:trace_client(_, _, {HandlerFun, InitialState})
  HandlerFun = fun(TraceMsg, State) -> NewState
```

- Same concept as when using `tracer/2`
- `{drop, N}` is sent if the ip port is too congested; N is the number of messages dropped.
- `end_of_trace` is sent when trace is finished in file ports

Exercises

Using tracing ...

- Start N number of erlang nodes and trace something you previously traced but from all nodes simultaneously
- Write a custom handler to print the messages you get differently from how they are shown
  - Try adding `timestamp` and/or `message` in your match specification
  - Try writing a handler that measures the avarage runtime for a function over a sample of 100 calls
- Start a tracer using `ip` and connect with `trace_client/2` and `telnet`
- Start a tracer using file
  - Trace to a single file and use `follow_file`
  - Trace to multiple files tweaking rotation, size and time parameters

4

ttb Introduction and Basics

Getting started again

ttb Introduction and Basics

What we know so far ...

- Start a trace with different flags (calls, messages etc)
- Use Match Specifications to refine call traces
- Connect several nodes into a trace
- Observe a trace from a client
- Save trace output to one or more files
- Customize the trace handler to handle the trace messages

So now what?

ttb Introduction and Basics

Building distributed system tracing
Some examples:

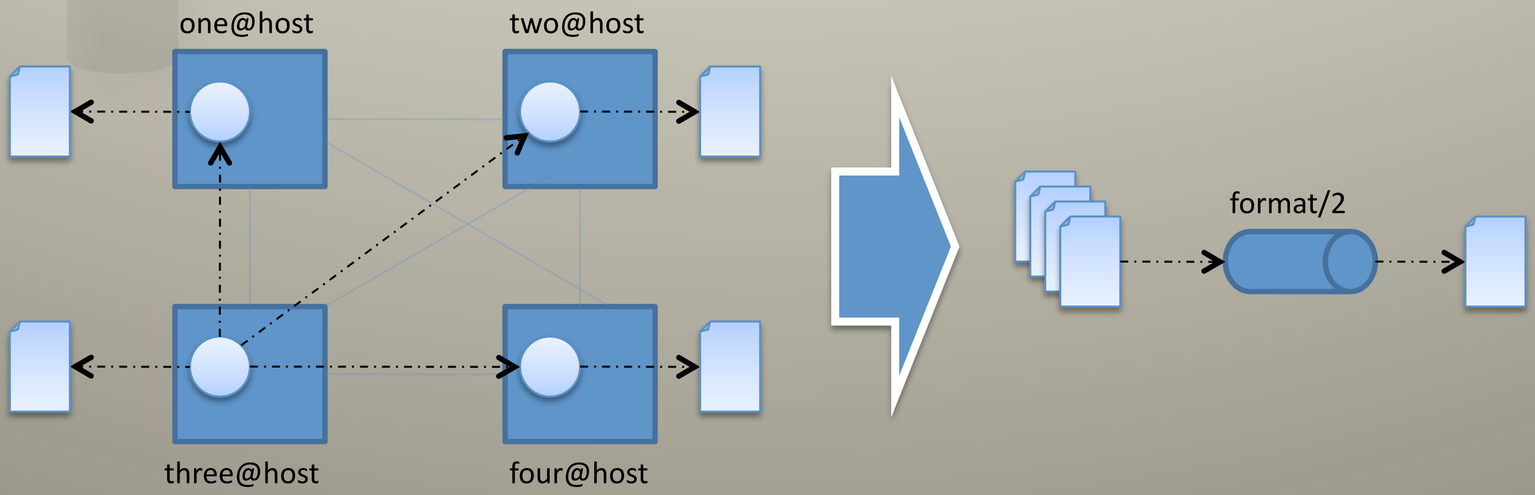
- Trace a specific action of the system (E.g. a session/message)
- Crude system sampling (E.g. How many refill their account with more than £5 per day/hour/second)
- Bug trapping (triggers E.g. on bad ets inserts)
- Any analysis/trace etc that is temporarily enabled
- System testing/verification

Important: This does ***not*** *replace logging*

ttb Introduction and Basics

ttb basics

- Starts one file port tracer on several nodes at the same time
- Starts one or more traces on these nodes
- When tracing is stopped the files are aggregated on the node that started the tracing
- A format function is used to format each entry which can then be written to a file (or whatever we want E.g. write a report)



ttb Introduction and Basics

`ttb` usage

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:tracer(Nodes, Options)
```

- The first parameter `Nodes` is a list of nodes where the tracer will be started
- Returns `{ok, NNodes}` where `NNodes` is a list of nodes where the tracers were started
- Options is a list of key-value tuples. The common options are:
  - `{file, Filename}` – Saves all files as `<nodename>-Filename`
  - `{handler,{HandlerFun, InitialState}}` – Same concept as when using `tracer/2` but `HandlerFun` differs and is described in later slides.
  - `{process_info, true|false}` – Specifies if extra process info should be included in the trace message; default is `true`
- More variations of these options exist (see the manual)

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:tracer(Nodes, Options)
```

```
(foo@Pasha)1> ttb:tracer([foo@Pasha, bar@Pasha], [{file,"MyTraceFile"}]).
{ok,[bar@Pasha,foo@Pasha]}
```

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:p(Procs, Flags)
```

- Same functionality as `dbg:p/2`
- `Procs` is a list of process identifiers (or a single item)
  - `registered | atom() | pid() | all | new | existing`
- Registered processes are identified by each node
- Globally registered processes use `{global, RegisteredName}`
- Returns `{ok,[{Procs, MatchDesc}]}` where `MatchDesc` is the same as `dbg:p/2` and `Procs` is which process it started a trace on
- Uses the same flags as `dbg:p/2`

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:tp|tpl|ctp|ctpl/2,3,4
```

- Same functionality as the `dbg` functions
- Same return value

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

ttb:p(Procs, Flags)

```
(foo@Pasha)2> ttb:p(all, [c]).
{ok,[{all,[{matched,bar@Pasha,36},{matched,foo@Pasha,37}]}]}
```

ttb:tp|tpl|ctp|ctpl/2,3,4

```
(foo@Pasha)3> ttb:tp(traceme, foo, []).
{ok,[{matched,bar@Pasha,1},{matched,foo@Pasha,1}]}
```

```
(foo@Pasha)4> traceme:foo(node()).
ok
```

```
(bar@Pasha)1> traceme:foo(node()).
ok
```

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

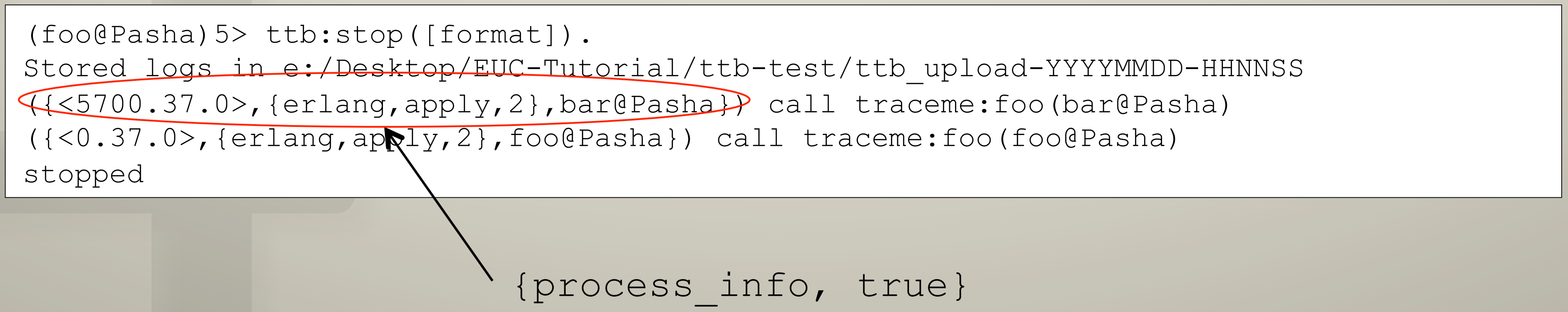
```
ttb:stop(Options)
```

- Stops the tracing
- `Options` is a list of one of two items; `fetch` and `format`
- `format` implies `fetch`
- `fetch` retrieves the files from the remote nodes and stores them in a directory of the node running the command
- `format` first fetches the files just like `fetch` but then automatically calls `ttb:format/1` on the fetched files
- Not specifying an option will leave the files on the remote nodes

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

`ttb:stop(Options)`



```
(foo@Pasha)5> ttb:stop([format]).
Stored logs in e:/Desktop/EUC-Tutorial/ttb-test/ttb_upload-YYYYMMDD-HHNNSS
({<5700.37.0>,{erlang,apply,2},bar@Pasha}) call traceme:foo(bar@Pasha)
({<0.37.0>,{erlang,apply,2},foo@Pasha}) call traceme:foo(foo@Pasha)
stopped
```

`{process_info, true}`

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:format(File, Options)
```

- Formats a file, list of files or a directory
- If no options are given this print the trace to stdout
- Options can be
  - `{out, standard_io|Filename}` – Specifies where the formatting should be written to
  - `{handler,{HandlerFun, InitialState}}` - The `fun` to handle the trace messages
  - Graphical interface will not be covered: `{handler, et}`

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:format(..., [{handler, {HandlerFun, ...}}])
```

- A fun of arity 4: `fun(Fd, Trace, TraceInfo, State)`
- `Fd` – The file descriptor to the file the `out` option specified or `standard_io`
- `Trace` – The trace message according to given flags and events as previously described
- `TraceInfo` – A list of key-value entries which has information about the process in context
- `State` – The custom state
- Last line is used as the new state
- `Trace` = `end_of_trace` when trace is finished from a given file

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

`ttb:format(File, Options)`

```
HandlerFun =
    fun(Fd, Trace, TraceInfo, _) ->
        io:format(Fd,"= TraceInfo:",[]),
        io:format(Fd,"~1000p~n",[TraceInfo]),
        io:format(Fd,"= Trace    :",[]),
        io:format(Fd,"~1000p~n~n",[Trace])
    end.
```

```
(foo@Pasha)6> ttb:format("ttb_upload-20101113-202517", [{out,"mytrace.log"}, {handler,
{HandlerFun, ok}}]).
ok
```

Examine the output file

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

`ttb:format(File, Options)`

```
= TraceInfo:[{flags,[{[all],[call]}]},{file,["./bar@Pasha-MyTraceFile"]},{node,[bar@Pasha]}]
= Trace    :{trace,{<5700.37.0>,{erlang,apply,2},bar@Pasha},call,{traceme,foo,[bar@Pasha]}}

= TraceInfo:[{flags,[{[all],[call]}]},{file,["./bar@Pasha-MyTraceFile"]},{node,[bar@Pasha]}]
= Trace    :end_of_trace

= TraceInfo:[{flags,[{[all],[call]}]},{file,["./foo@Pasha-MyTraceFile"]},{node,[foo@Pasha]}]
= Trace    :{trace,{<0.37.0>,{erlang,apply,2},foo@Pasha},call,{traceme,foo,[foo@Pasha]}}

= TraceInfo:[{flags,[{[all],[call]}]},{file,["./foo@Pasha-MyTraceFile"]},{node,[foo@Pasha]}]
= Trace    :end_of_trace
```

Try it again using the `timestamp` flag in your trace

ttb Introduction and Basics

1. Start a tracer on a set of nodes
2. Start a trace (and trace patterns if tracing calls)
3. Stop the trace (automatic aggregation)
4. Format the output

```
ttb:write_trace_info(Key, Info)
```

- Adds a `{Key, Info}` tuple to the `TraceInfo` element
- Useful to "save" enviornment information with the trace
- Must be called on the node that started the `ttb` tracer
- Resets after `ttb:stop` has been called

Exercises

Using the trace tool builder ...

- Use the trace tool builder to run traces you previously used `dbg` for
- Create a module with a simple interface that takes a module, function and arity and traces that for a given number of seconds and then turns the trace of and formats the results to a file

Hardcore:

- Create a trace for all processes calling `traceme:foo/1` on three nodes
- After the trace is active call the the function many times on all nodes with `random:uniform (1000)` as input then stop the trace
- Format the output so that you get a file with 10 lines showing how many calls were within a given range (1-100, 101-200, 201-300 etc)

5 Where to go from here …

What I didn't include

Where to go from here ...

Look into ...

- `dbg:c/3,4`
- Sequential tracing
- Load/Save/Change configuration
-

?

?

?