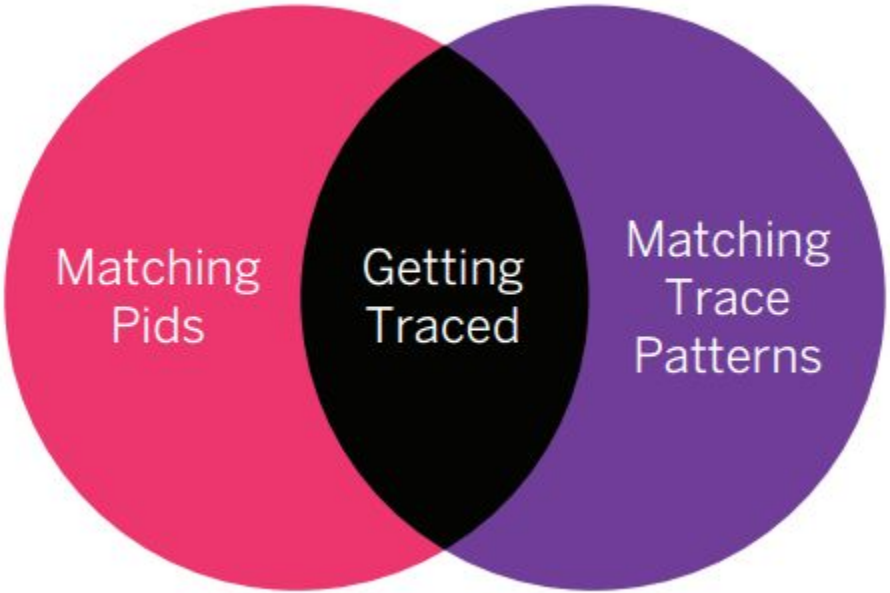
Elixir.
Sequential tracing

Denys Gonchar

Main source of information

- Erlang Tracing API:
  - erlang:trace/3
  - erlang:trace_pattern/3
- Elang Match Specification
- Erlang dbg module
- Erlang seq_trace module

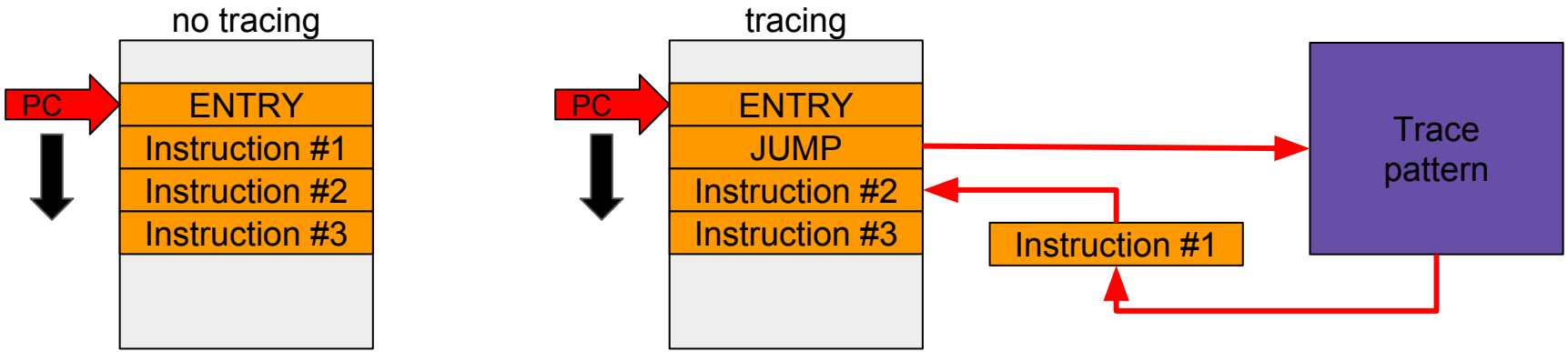
What is traced



Real practice

usually engineers use one of these two approaches:

- trace registred processes, so pid is known
- focus on functions tracing
  - :dbg.p(:all,:call) and then set required trace pattern



:dbg.c/3

- Provides nice interface to trace out execution flow for some API call.
- Traces out information about current and spawned from current processes.
- No reporting about other involved in call handling processes.
- Nothing extra to stop on this interface. In most cases it has no real value as APIs of gen_server/gen_fsm modules just send the message to the process and that’s it.
- Test it:
  - `:dbg.c KVServer.Command, :run, [{:get, "alaska", "bear"}]`
  - `:dbg.c KVServer.Command, :run, [{:get, "zzz", "bear"}]`

seq_trace

Sequential tracing is a way to trace a sequence of messages sent between different local or remote processes, where the sequence is initiated by a single message. In short, it works as follows:

Each process has a **trace token**, which can be empty or not empty. When not empty, the trace token can be seen as the tuple {Label, Flags, Serial, From}. The trace token is passed invisibly with each message.

To start a sequential trace, the user must explicitly set the trace token in the process that will send the first message in a sequence.

The trace token of a process is set each time the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not.

Generally it works like domino effect, let’s check some example diagram and find execution sequences on it

seq_trace example (direct :seq_trace.set_token/2 call)

```
:seq_trace.set_system_tracer(self)

spawn fn->
      :seq_trace.set_token(:send,:true)
      KVServer.Command.run({:get, "alaska", "bear"})
      :seq_trace.set_token([])
end
flush

p = :rpc.call(:"bar@szm-mac",ExTrace.Seq,:start_seq_tracer_process,[])
send {:seq_tracer,:"bar@szm-mac"}, {:set_tracer_pid,self()}
:rpc.call(:"bar@szm-mac",:seq_trace,:set_system_tracer,[p])

spawn fn->
      :seq_trace.set_token(:send,:true)
      KVServer.Command.run({:get, "zzz", "bear"})
      :seq_trace.set_token([])
end
flush
```

seq_trace example (direct :seq_trace.set_token call)

- Sequential tracing is not performed across ports, but is performed transparently between nodes
- to distinguish easily between different executions use labels:
  - :seq_trace.set_token(:label,:rand.uniform(100))
- set_token/1 - can be used to temporarily exclude message passing from the trace
- reset_trace/0 - resets token for all processes and all messages in message queues on the local node.
- print/1 & print/2 - can be used for additional output if the calling process currently is executing within a sequential trace

seq_trace example (setting token using trace pattern)

```
ExTrace.Seq.start_dbg_tracer
{:ok,pid} = :dbg.get_tracer()
:seq_trace.set_system_tracer(pid)
send {:seq_tracer,:"bar@szm-mac"}, {:set_tracer_pid,pid}

:dbg.tp(Elixir.KVServer.Command,:run,[{:"_",[{:"==",{:is_seq_trace},:false}],
     [{:set_seq_token,:send,:true}]}])
:dbg.p(:all,:call)  #it's good idea to set trace pattern first, and only then enable tracing


spawn fn->
     KVServer.Command.run({:get, "zzz", "bear"})
     :seq_trace.set_token([])
end
```

seq_trace example (setting token using trace pattern)

Pluses:

- no need to recompile the system, so it can be used in the real system

Minuses:

- no easy way to reset seq_trace token at the point you need, so you have to sort out all the irrelevant messages by yourself
- no chance to set unique label, but you can rely on serial information

combination of seq_trace and normal tracing

```
ExTrace.Seq.start_tracer

#list of modules taken from kv.app & kv_server.app files
for module <- [Elixir.KVServer, Elixir.KVServer.Command, Elixir.KV,
               Elixir.KV.Bucket, Elixir.KV.Bucket.Supervisor, Elixir.KV.Registry,
               Elixir.KV.Router, Elixir.KV.Supervisor] do
     :dbg.tp(module,[{:"_",[{:is_seq_trace}],[{:message,:true}]}])
end
:dbg.tp(Elixir.KVServer.Command,:run,[{:"_",[{:"==",{:is_seq_trace},:false}],
     [{:set_seq_token,:send,:true}]}])
:dbg.p(:all,:call)

spawn fn-> KVServer.Command.run({:get, "zzz", "bear"}) end
```

OTP19 features

Trace patterns for send & receive events. there was introduced the new :dbg.tpe/2 interface to reflect this change. Match spec for send event is useful feature for sequential tracing (few words about it the next slides). Unfortunately match spec for receive event is very limited:

> **A match specification for 'receive' trace can use all guard and body functions except caller, is_seq_trace, get_seq_token, set_seq_token, enable_trace, disable_trace, trace, silent, and process_dump.**

Receive notification indicates delivery of the message to the mailbox (not start of processing of the message). So it's quite logical to prohibit caller, enable_trace, disable_trace, trace, silent and process_dump functions as message can stay in the mailbox for undefined time period and you don't really know the process state at the message arrival. But is_seq_trace, get_seq_token and set_seq_token functions for receive event could operate with message's token, not process's. It seems so logical to start sequential tracing as reaction on the message, sad that it's not there (yet).

receive notification

also it’s worth mentioning that seq_trace receive notification indicates start of message processing, rather than arrival to the mailbox

```
ExTrace.Seq.start_tracer

spawn fn ->
      pid = spawn fn ->
            receive do :a -> :ok end
            receive do :b -> :ok end
      end

      flags=[:receive,:send]
      :dbg.p(pid,flags); :dbg.p(self(),flags)

      for f <- flags do :seq_trace.set_token(f, :true) end
      :seq_trace.set_token(:label,1);  send pid, :b; send pid, :b
      :seq_trace.set_token(:label,2);  send pid, :a
end
```

Limitations. Spawning and closure. Problem

There is no ‘set on spawn’ option for sequential trace token. so closure can do a cheap trick:

```
s=self()
spawn fn -> send  s, :anything end
receive do :anything -> :ok end
```

You won’t get sequential tracing for spawned process. Moreover - receiving the message from the spawned process you’re losing trace token.

There’s not much you can do about it without changes in code, but what’s important - **you can detect this situation!**

Limitations. Spawning and closure. Example

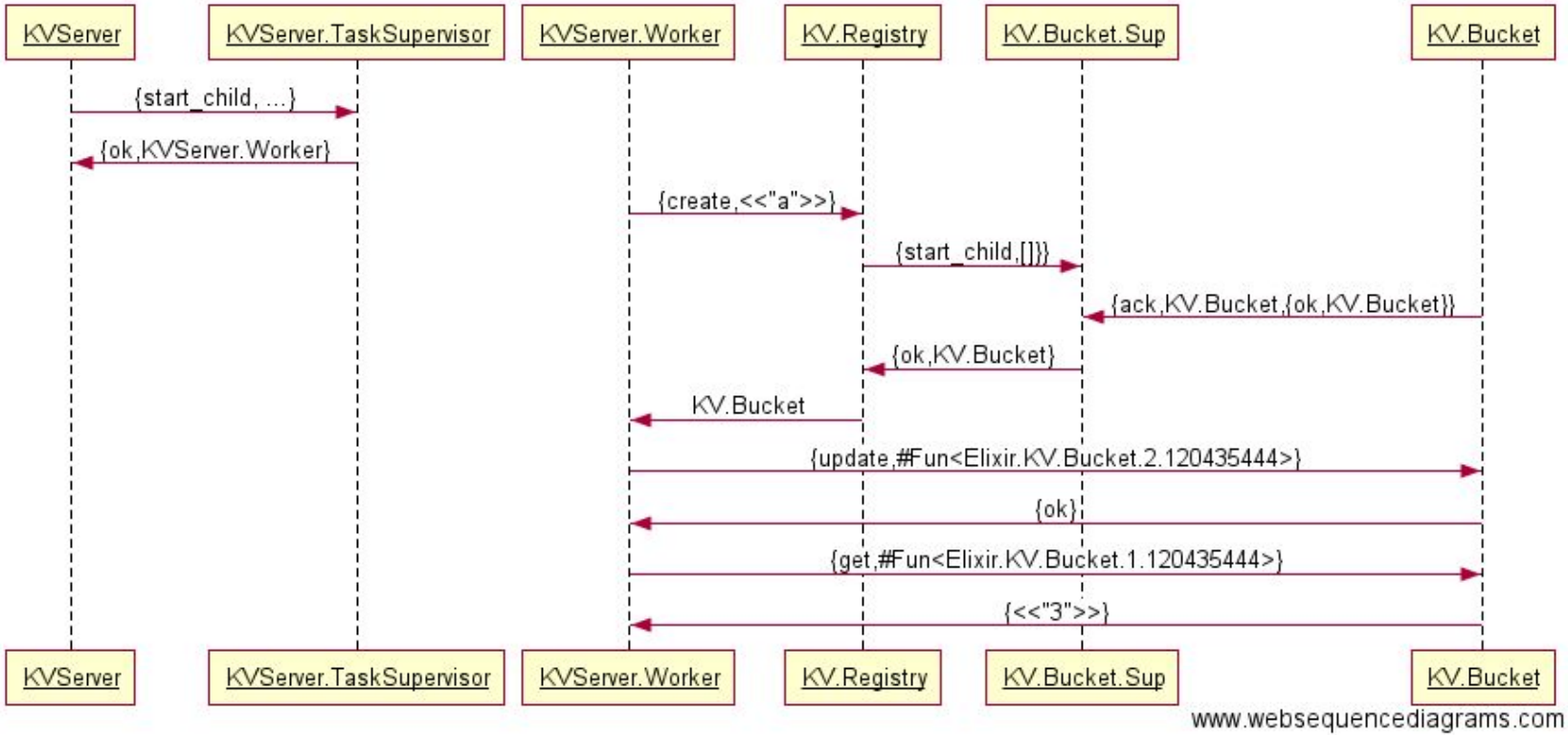
unfortunately such closure trick is quite common, we even have one in our demo application:

```
ExTrace.Seq.start_tracer()
#:application.which_applications
ExTrace.Seq.set_trace_flags_for_app(:kv_server,[:s])
ExTrace.Seq.set_trace_flags_for_app(:kv,[:s])
#:io.format("~p",[for n <- Process.registered do
#                    {Process.whereis(n), n}
#                end |> :lists.usort()])
:observer.start

# "a" is the name of the new bucket
# for the local node (:"foo@szm-mac")
ExTrace.test_kv_server "a"
```

Limitations. Spawning and closure. Example

```
#https://www.websequencediagrams.com/ model
KVServer                    ->   KVServer.TaskSupervisor : {start_child, ...}
KVServer.TaskSupervisor     ->   KVServer                : {ok,KVServer.Worker}

KVServer.Worker    ->   KV.Registry      : {create,<<"a">>}
KV.Registry        ->   KV.Bucket.Sup    : {start_child,[]}}
KV.Bucket          ->   KV.Bucket.Sup    : {ack,KV.Bucket,{ok,KV.Bucket}}
KV.Bucket.Sup      ->   KV.Registry      : {ok,KV.Bucket}
KV.Registry        ->   KVServer.Worker  : KV.Bucket
KVServer.Worker    ->   KV.Bucket        : {update,#Fun<Elixir.KV.Bucket.2.120435444>}
KV.Bucket          ->   KVServer.Worker  : {ok}
KVServer.Worker    ->   KV.Bucket        : {get,#Fun<Elixir.KV.Bucket.1.120435444>}
KV.Bucket          ->   KVServer.Worker  : {<<"3">>}
```

Limitations. Spawning and closure. Example



Limitations. Spawning and closure. Detection

so now it’s time to get back to OTP19 features, to detect the problem we can use match spec for send event.

```
ExTrace.Seq.start_tracer

:dbg.tp(KVServer.Command,:run,[{:"_",[{:"==",{:is_seq_trace},:false}],
     [{:set_seq_token,:label,1},{:enable_trace,:send},{:enable_trace,:procs}]}])
:dbg.tpe(:send,[{[:"$1",:"$2"],[{:is_seq_trace}],[{:enable_trace,:"$1",:send},
     {:enable_trace,:"$1",:procs}]}])
:dbg.p(:all,:call)

# "ab" is the name of the new bucket
# for the local node (:"foo@szm-mac")
spawn fn-> KVServer.Command.run({:create, "ab"}) end
```

Limitations. Spawning and closure. Detection

if you’re not lucky to have the latest OTP:

```
ExTrace.Seq.start_tracer

for fun <- [:spawn, :spawn_opt, :spawn_link] do
     :dbg.tp(:proc_lib,fun,[{:"_",[{:is_seq_trace}],[{:message,{:caller}}]}])
     :dbg.tp(:erlang,fun,[{:"_",[{:is_seq_trace}],[{:message,{:caller}}]}])
end
:dbg.tp(KVServer.Command,:run,[{:"_",[{:"==",{:is_seq_trace},:false}],
     [{:set_seq_token,:send,:true},{:enable_trace,:procs}]}])
:dbg.p(:all,:call)

# "abc" is the name of the new bucket
# for the local node (:"foo@szm-mac")
spawn fn-> KVServer.Command.run({:create, "abc"}) end
```

Finit

Questions???