Poor man’s gen_server

About me

Inaka

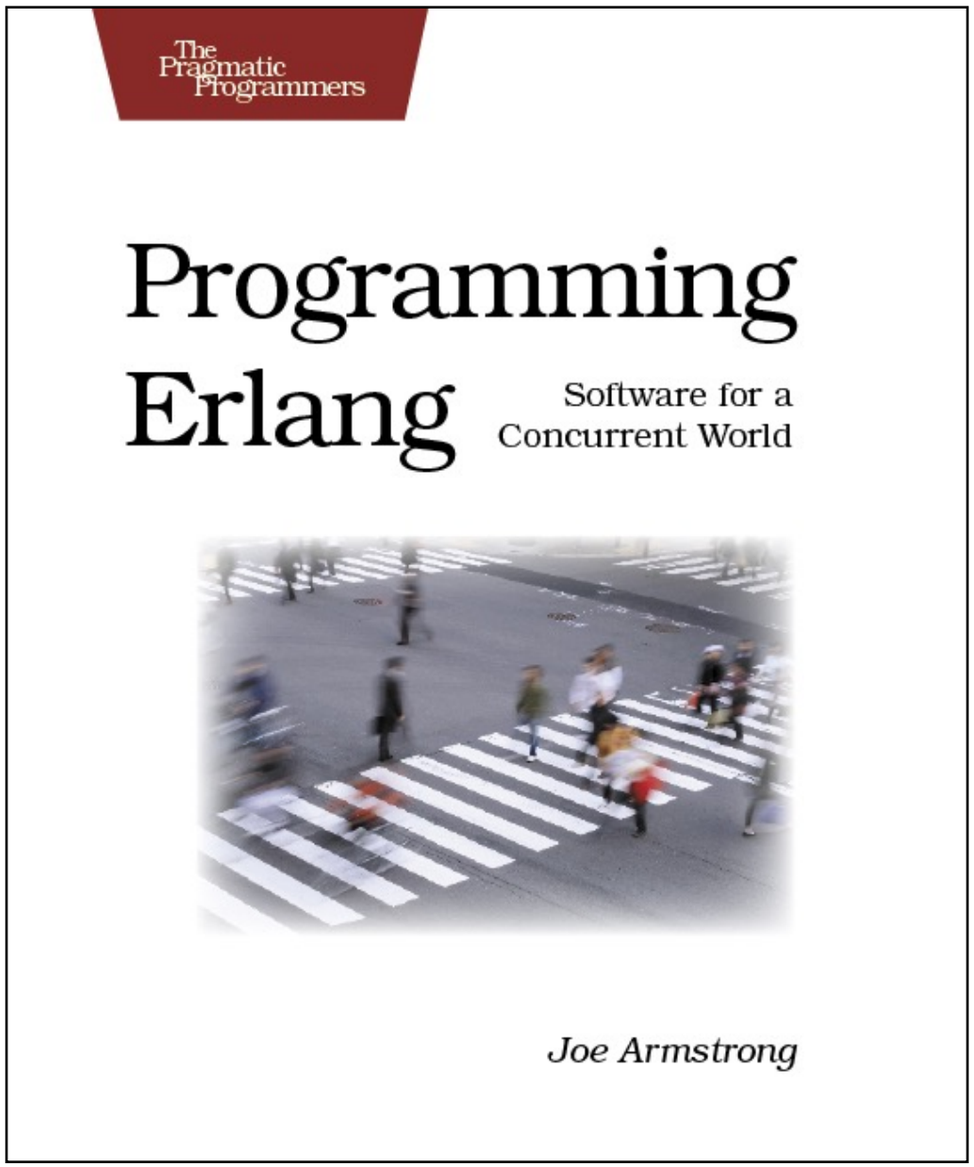
- Started in 2010

- Acquired by Erlang Solutions in 2014.

- We build end-to-end applications with Erlang, Elixir, Ruby, Web, iOS and Android components. And we develop highly concurrent application servers such as Whisper and TigerText.

- We're active contributors to the open-source community and focus a lot on training.

Where I'm coming from



My first task:

Write a small TCP server

This is rather simple, just use a tail recursive function

```erlang
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().
```

```erlang
loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().

loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().
```

```
loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket)
  loop().
```

```
loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().

loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().
```

```
loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().
```

```
loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data)
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```

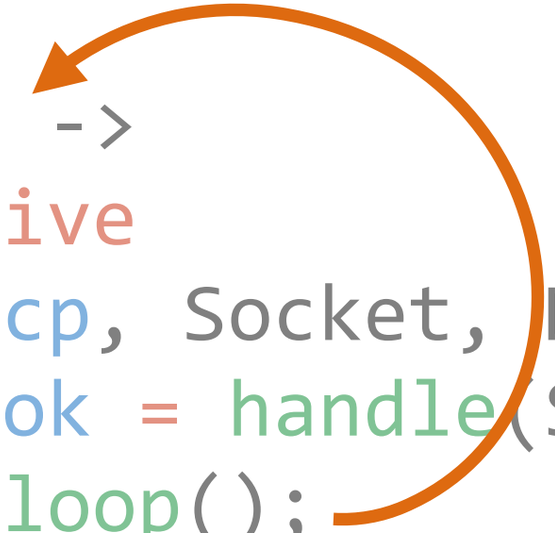
```
-module(my_server).
-export([start/1, init/1, loop/0]).

start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop().

loop() ->
  receive
    {tcp, Socket, Data} ->
      ok = handle(Socket, Data),
      loop();
  end.

handle(Socket, Data) ->
  io:format("Got ~p", [Data]).
```



Problem 1:

I need a state

Solution:

Use variables as arguments of the tail recursive function

```erlang
start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop(a, b, c).

loop(V1, V2, V3) ->
  receive
    {tcp, Socket, <<"message1">>} ->
      NewV1 = handle_message1(Socket, V1),
      loop(NewV1, V2, V3);
    {tcp, Socket, <<"message2">>} ->
      NewV2 = handle_message2(Socket, V2),
      loop(V1, NewV2, V3);
    {tcp, Socket, <<"message3">>} ->
      NewV3 = handle_message3(Socket, V3),
      loop(V1, V2, NewV3)
end.
```

```
start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop(a, b, c).

loop(V1, V2, V3) ->
  receive
    {tcp, Socket, <<"message1">>} ->
      NewV1 = handle_message1(Socket, V1),
      loop(NewV1, V2, V3);
    {tcp, Socket, <<"message2">>} ->
      NewV2 = handle_message2(Socket, V2),
      loop(V1, NewV2, V3);
    {tcp, Socket, <<"message3">>} ->
      NewV3 = handle_message3(Socket, V3),
      loop(V1, V2, NewV3)
end.
```

```
start(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(ListenSocket) ->
  {ok, _} = gen_tcp:accept(ListenSocket),
  loop(a, b, c).

loop(V1, V2, V3) ->
  receive
    {tcp, Socket, <<"message1">>} ->
      NewV1 = handle_message1(Socket, V1),
      loop(NewV1, V2, V3);
    {tcp, Socket, <<"message2">>} ->
      NewV2 = handle_message2(Socket, V2),
      loop(V1, NewV2, V3);
    {tcp, Socket, <<"message3">>} ->
      NewV3 = handle_message3(Socket, V3),
      loop(V1, V2, NewV3)
end.
```

Problem 2:

This is unmaintainable

Solution:

Use a record or a map

```erlang
start() ->
  {ok, spawn_link(?MODULE, loop, [#state{}])}.

loop(State) ->
  receive
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
end.
```

```
start() ->
    {ok, spawn_link(?MODULE, loop, [#state{}])}.

loop(State) ->
    receive
        {tcp, _Socket, Message} ->
            NewState = handle_message(Message, State),
            loop(NewState)
    end.
```

Problem 3:

I need to interact with it

Solution:

Send it messages, I can even add helper functions for this

```erlang
start() ->
  {ok, spawn_link(?MODULE, loop, [#state{}])}.

do_something(ServerRef) ->
  ServerRef ! do_something.

loop(State) ->
  receive
    do_something ->
      NewState = handle_do_something(State),
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
end.
```

```erlang
start() ->
  {ok, spawn_link(?MODULE, loop, [#state{}])}.

do_something(ServerRef) ->
  ServerRef ! do_something.

loop(State) ->
  receive
    do_something ->
      NewState = handle_do_something(State),
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
end.
```

```
start() ->
  {ok, spawn_link(?MODULE, loop, [#state{}])}.

do_something(ServerRef) ->
  ServerRef ! do_something.

loop(State) ->
  receive
    do_something ->
      NewState = handle_do_something(State),
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```erlang
start() ->
  {ok, spawn_link(?MODULE, loop, [#state{}])}.

do_something(ServerRef) ->
  ServerRef ! do_something.

loop(State) ->
  receive
    do_something ->
      NewState = handle_do_something(State),
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

Problem 4:

Sending messages to dead processes

Solution:

They need to reply, otherwise, the helper functions should fail

```erlang
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```erlang
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

```
do_something(ServerRef) ->
  ServerRef ! {do_something, self()},
  receive im_alive -> ok
  case is_process_alive(ServerRef) of
    true  -> throw(no_proc);
    false -> throw(timeout)
  end.
```

```
loop(State) ->
  receive
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState)
  end.
```

And yes, you are supposed to use monitors for this

Problem 5:

I need to retrieve values from this process

Solution:

Like in the previous example, I have a ‘receive’ waiting for a reply

```erlang
get_something(ServerRef) ->
  ServerRef ! {get_something, self()},
  receive {ok, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```erlang
loop(State) ->
  receive
    {get_something, Pid} ->
      Pid ! {ok, handle_get_something(State)},
      loop(State);
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState)
  end.
```

```
get_something(ServerRef) ->
  ServerRef ! {get_something, self()},
  receive {ok, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```
loop(State) ->
  receive
    {get_something, Pid} ->
      Pid ! {ok, handle_get_something(State)},
      loop(State);
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState)
  end.
```

```erlang
get_something(ServerRef) ->
  ServerRef ! {get_something, self()},
  receive {ok, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```erlang
loop(State) ->
  receive
    {get_something, Pid} ->
      Pid ! {ok, handle_get_something(State)},
      loop(State);
    {do_something, Pid} ->
      NewState = handle_do_something(State),
      Pid ! im_alive,
      loop(NewState)
  end.
```

Problem 6:

This doesn't differentiate replies!

What?

```erlang
%% client.erl
some_function() ->
  Message1Response = server:get_message1(),
  Message2Response = server:get_message2(),
  {Message1Response, Message2Response}.


%% server.erl
loop(State) ->
  receive
    {message2, Pid} ->
      Pid ! {ok, handle_message2(State)},
      loop(State);
    {message1, Pid} ->
      Pid ! {ok, handle_message1(State)},
      loop(NewState)
  end.
```

```erlang
%% client.erl
some_function() ->
  Message1Response = server:get_message1(),
  Message2Response = server:get_message2(),
  {Message1Response, Message2Response}.


%% server.erl
loop(State) ->
  receive
    {message2, Pid} ->
      Pid ! {ok, handle_message2(State)},
      loop(State);
    {message1, Pid} ->
      Pid ! {ok, handle_message1(State)},
      loop(NewState)
  end.
```

```erlang
%% client.erl
some_function() ->
    Message1Response = server:get_message1(),
    Message2Response = server:get_message2(),
    {Message1Response, Message2Response}.


%% server.erl
loop(State) ->
  receive
    {message2, Pid} ->
      Pid ! {ok, handle_message2(State)},
      loop(State);
    {message1, Pid} ->
      Pid ! {ok, handle_message1(State)},
      loop(NewState)
  end.
```

```erlang
%% client.erl
some_function() ->
  Message1Response = server:get_message1(),
  Message2Response = server:get_message2()
  {Message1Response, Message2Response}.


%% server.erl
loop(State) ->
  receive
    {message2, Pid} ->
      Pid ! {ok, handle_message2(State)},
      loop(State);
    {message1, Pid} ->
      Pid ! {ok, handle_message1(State)},
      loop(NewState)
  end.
```

Bottom line:

We can get the answer to the wrong question under certain circumstances

Solution:

I can add a random “id” to identify replies

```erlang
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State)
  end.
```

```
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State)
  end.
```

```
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something,{self(), Ref}
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State)
  end.
```

```
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State)
  end.
```

```
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      C{Ref, handle_get_something(State)},
      loop(State)
  end.
```

```
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref},
  receive {Ref, Reply} ->
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State) ->
  receive
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State)
  end.
```

Problem 7:

The message inbox gets crowded

Solution:

Finally one that's easy, just add a "catch-all" clause

```erlang
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      NewState = handle_do_something(State),
      Caller ! {Ref, im_alive},
      loop(NewState);
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      NewState = handle_do_something(State),
      Caller ! {Ref, im_alive},
      loop(NewState);
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      NewState = handle_do_something(State),
      Caller ! {Ref, im_alive},
      loop(NewState);
    {get_something, {Caller, Ref}} ->
      Caller ! {Ref, handle_get_something(State)},
      loop(State);
    {tcp, _Socket, Message} ->
      NewState = handle_message(Message, State),
      loop(NewState);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.
```

Problem 8:

Terminating properly!

Solution:

Just a try…catch!

```erlang
loop(State) ->
  try
    NewState = do_something(State),
    loop(NewState)
  catch
    _:_ -> do_cleanup(State)
end.
```

```
loop(State) ->
  try
    NewState = do_something(State),
    loop(NewState)
  catch
    _:_ -> do_cleanup(State)
end.
```

WAT!

Problem 8.1:

This is nesting try…catches

Solution:

Use multiple try catches

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          exit(proc_dead);
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          proc_dead;
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          proc_dead;
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          proc_dead
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          proc_dead;
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
      case catch handle_do_something(State) of
        {'EXIT', Reason} ->
          handle_error(Reason, State),
          proc_dead;
        {ok, NewState} ->
          Caller ! {Ref, im_alive},
          loop(NewState)
      end;
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
end.
```

And I need this for

EVERY METHOD

Problem 8.2:

Too much repeated code

Solution:

Code reuse

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.
```

```
do_execute(Fun, State, Caller, Ref) ->
```

```
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

d
```

```
case safe_execute(Fun, State) of
```

```
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

s
```

```
case catch Fun(State) of
```

```
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    Unexpected ->
```

```
{reply, Reply, NewState} ->
  Caller ! {Ref, Reply},
  loop(NewState);
{noreply, NewState} ->
  Caller ! {Ref, im_alive},
  loop(NewState);
{error, Reason} ->
  handle_error(Reason, State),
  proc_dead;
Unexpected ->
  handle_error({bad_return, Unexpected}, State),
  proc_dead
```

```
    Reply ->
      Reply
  end;
```

Problem 9:

I need this process on a supervision tree

Solution:

Go to the docs, study a bit, and make it OTP compliant

```erlang
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

```
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
```

```
system_continue(Parent, Deb, State) ->
```

```
system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

```erlang
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

```
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
```

```
system_get_state(Srv) ->
```

```
system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

```erlang
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

loop(State) ->
  receive
    {do_something, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something/1, State, Caller, Ref);
    {do_something_else, {Caller, Ref}} ->
     Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
     do_execute(fun handle_do_something_else/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}

system_replace_state(StateFun, Srv) ->

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).
```

Wait, I just lost track,

What does the code even look like now?

```erlang
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).
```

```
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).
```

```
init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).
```

```
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```erlang
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
```

```erlang
get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.
```

```erlang
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
```

```
loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);;
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      handle_unexpected(Unexpected, State),
      loop(State)
  end.
```

```
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;

system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
```

```
do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
    Reply ->
      Reply
  end;
```

```
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.

write_debug(Dev, Event, Name) ->
  io:format(Dev, "~p event = ~p~n", [Name, Event]).

handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```erlang
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
```

```erlang
system_continue(Parent, Deb, State) ->
  loop(State, Parent, Deb).

system_terminate(Reason, _Parent, _Deb, _State) ->
  exit(Reason).

system_get_state(Srv) ->
  {ok, Srv}.

system_replace_state(StateFun, Srv) ->
  NewSrv = StateFun(Srv),
  {ok, NewSrv, NewSrv}.
```

```erlang
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      proc_dead;
    Unexpected ->
      handle_error({bad_return, Unexpected}, State),
      proc_dead
  end.

safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
```

```erlang
handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}
```

```erlang
handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

```
-module(my_server).
-export([start_link/1, init/1, loop/0, do_something/1, get_something/1,
         system_continue/3, system_terminate/4,
         system_get_state/1, system_replace_state/3]).

-define(TIMEOUT, 5000).

-record(state, {num_calls = 0 :: integer()}).

start_link(ListenSocket) ->
  {ok, spawn_link(?MODULE, init, [ListenSocket])}.

init(Parent) ->
  register(?MODULE, self()),
  Deb = sys:debug_options([]),
  proc_lib:init_ack(Parent, {ok, self()}),
  loop(#state{}, Parent, Deb).

do_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {do_something, {self(), Ref}},
  receive {Ref, im_alive} -> ok
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

get_something(ServerRef) ->
  Ref = make_ref(),
  ServerRef ! {get_something, {self(), Ref}},
  receive {Ref, Reply} -> Reply
  after ?TIMEOUT ->
    case is_process_alive(ServerRef) of
      true  -> throw(no_proc);
      false -> throw(timeout)
    end
  end.

loop(State, Parent, Deb) ->
  receive
    {do_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_do_something/1, State, Caller, Ref);
    {get_something, {Caller, Ref}} ->
      Deb2 = sys:handle_debug(Deb, fun write_debug/3, ?MODULE, {in, alloc, From}),
      do_execute(fun handle_get_something/1, State, Caller, Ref);
    {system, From, Request} ->
      sys:handle_system_msg(Request, From, Parent, ?MODULE, Deb, State);
    Unexpected ->
      do_execute(fun(S) -> handle_unexpected(Unexpected, S) end, State, Caller, nil);
      loop(State)
  end.

do_execute(Fun, State, Caller, Ref) ->
  case safe_execute(Fun, State) of
    {reply, Reply, NewState} ->
      Caller ! {Ref, Reply},
      loop(NewState);
    {noreply, NewState} ->
      Caller ! {Ref, im_alive},
      loop(NewState);
    {error, Reason} ->
      handle_error(Reason, State),
      pr[illegible]dead;
    Unex[illegible]ed ->
      h[illegible]er[illegible]or([illegible]re[illegible]rn, U[illegible]xp[illegible]te[illegible], [illegible]ta[illegible])
      [illegible]
  end.
```

And this is the only “application code” we have

```
safe_execute(Fun, State) ->
  case catch Fun(State) of
    {'EXIT', Reason} ->
      {error, Reason};
```

```
handle_do_something(State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.

handle_get_something(State) ->
  {reply, State#state.num_calls, State}
```

```
handle_get_something(State) ->
  {reply, State#state.num_calls, State}.

handle_unexpected(_Message, State) ->
  {noreply, State#state{num_calls = State#state.num_calls + 1}}.
```

Whatever, it works

But the actual logic we want, are just a few lines lost in a huge file

EUREKA!

I should separate all that generic code to it's own module!

What's a good name for it?

Well, it's a server…

And also kind of generic right?

I KNOW!

generic_server.erl

generic_server.erl

Too long!

gen_server.erl

This is so clever!

I can’t believe I’m the first one to think of this!

I should open source it!

Hello Hernan, 1980s calling

And this is not even considering:

sys:trace
monitors
trap_exit
tests
extensibility
multiple calls
multiple nodes
hibernation
the other “gens”

Morale of the story?

Using OTP saves time in the end, and it's not boilerplate, it's useful code that we won't have to write ourselves.

It's like coding C++ without touching stl, I mean, standard libraries are optional…

But then again:

So is brushing your teeth.

http://inaka.github.io
http://inaka.net/blog
http://learnyousomeerlang.com/
http://www.erlang.org/doc/

Questions?