Programming Distributed Systems

Erlang OTP

Annette Bieniusa, Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019
Erlang OTP
Example: Echo server 1

-module(echo).
-export([start_link/0]).

start_link() ->
    {ok, spawn_link(fun() -> loop() end)}.

loop() ->
    receive
        {From, Msg} ->
            From ! Msg,
            loop();
        stop ->
            true
    end.
Example: Echo server client 1

```erlang
-module(echo_client).
-export([test/0]).

test() ->
    {ok, Server1} = echo:start_link(),
    {ok, Server2} = echo:start_link(),
    Server1 ! {self(), hello},
    Server2 ! {self(), world},
    receive
        Msg1 -> io:format("Server 1 responded: ~p\n", [Msg1])
    end,
    receive
        Msg2 -> io:format("Server 2 responded: ~p\n", [Msg2])
    end.
```

Does this always work correctly?
Example: Echo server 2

```prolog
-module(echo2).
-export([start_link/0]).

start_link() ->
    {ok, spawn_link(fun() -> loop() end)}.

loop() ->
    receive
        {From, Msg} ->
            From ! {self(), Msg},
            loop();
        stop ->
            true
    end.

Sending own process-id (self()), so that receiver can match answer to request.
```
Example: Echo client 2

```erlang
-module(echo_client2).
-export([test/0]).

test() ->
    {ok, Server1} = echo2:start_link(),
    {ok, Server2} = echo2:start_link(),
    Server1 ! {self(), hello},
    Server2 ! {self(), world},
    receive
        {Server1, Msg1} -> io:format("1 responded: ~p~n", [Msg1])
    end,
    receive
        {Server2, Msg2} -> io:format("2 responded: ~p~n", [Msg2])
    end.
```
Example: Counting server

-module(counter).
-export([start_link/0, loop/1]).

start_link() ->
  {ok, spawn_link(?MODULE, loop, [0])}.

loop(Counter) ->
  receive
    {From, increment} ->
      From ! {self(), ok},
      loop(Counter + 1);
    {From, read} ->
      From ! {self(), Counter},
      loop(Counter);
    stop ->
      true
  end.
Example: Bounded Counter

-module(bounded_counter).
-export([start_link/1, loop/1, increment/1, read/1]).
-record(state, {limit, count}).

start_link(Limit) ->
    State = #state{limit = Limit, count = 0},
    {ok, spawn_link(?MODULE, loop, [State])}.

loop(State = #state{count = Counter, limit = Limit}) ->
    receive
        {From, increment} when Counter < Limit ->
            From ! {self(), ok},
            loop(State#state{count = Counter + 1});
        {From, increment} ->
            From ! {self(), {error, limit_reached}},
            loop(State);
        {From, read} ->
            From ! {self(), Counter},
            loop(State);
    stop ->
        true
    end.
Bounded Counter API (synchronous call)

increment(Server) ->
    Server ! {self(), increment},
    receive
        {Server, Msg} -> Msg
    end.

read(Server) ->
    Server ! {self(), read},
    receive
        {Server, Msg} -> Msg
    end.
Generic Client/Servers

start

initialize

stop → loop ↪ receive

terminate
### Separating generic and specific parts

<table>
<thead>
<tr>
<th>Generic</th>
<th>Specific (Counter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawning the server</td>
<td>Initial State:</td>
</tr>
<tr>
<td>Storing the loop data</td>
<td><code>#state{limit = Limit, count = 0}</code></td>
</tr>
<tr>
<td>Sending requests to server</td>
<td>Handling of requests (increment, read)</td>
</tr>
<tr>
<td>Sending replies to client</td>
<td></td>
</tr>
<tr>
<td>Receiving server replies</td>
<td></td>
</tr>
<tr>
<td>Stopping</td>
<td>(cleaning up)</td>
</tr>
</tbody>
</table>
### Separating generic and specific parts

<table>
<thead>
<tr>
<th>Generic</th>
<th>Specific (Counter)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawning the server</td>
<td>Initial State:</td>
</tr>
<tr>
<td>Storing the loop data</td>
<td>#state{limit = Limit, count = 0}</td>
</tr>
<tr>
<td>Sending requests to server</td>
<td>Handling of requests (increment, read)</td>
</tr>
<tr>
<td>Sending replies to client</td>
<td></td>
</tr>
<tr>
<td>Receiving server replies</td>
<td></td>
</tr>
<tr>
<td>Stopping</td>
<td>(cleaning up)</td>
</tr>
</tbody>
</table>

Implement generic part once, use callbacks for specific parts
Specific part

-module(bounded_counter2).
-export([start_link/1, increment/1, read/1]).
-export([init/1, handle_call/3]).

-record(state, {limit, count}).

start_link(Limit) ->
    my_gen_server:start_link(?MODULE, [Limit], []).

increment(Server) ->
    my_gen_server:call(Server, increment).

read(Server) ->
    my_gen_server:call(Server, read).

init([Limit]) ->
    {ok, #state{limit = Limit, count = 0}}.

handle_call(increment, _From, State = #state{count = Counter, limit = Limit}) ->
    case Counter < Limit of
        true ->
            {reply, ok, State#state{count = Counter + 1}};
        false ->
            {reply, {error, limit_reached}, State}
    end;

handle_call(read, _From, State) ->
    {reply, State#state.count, State}.
Simple generic server

-module(my_gen_server).
-export([start_link/3, call/2]).

start_link(Module, Args, _Options) ->
  {ok, InitialState} = Module:init(Args),
  {ok, spawn_link(fun() -> loop(Module, InitialState) end)}.

call(P, Msg) ->
  P ! {call, self(), Msg},
  receive
    {reply, P, Response} ->
      Response
  end.

loop(Module, State) ->
  receive
    {call, From, Msg} ->
      {reply, Reply, NewState} = Module:handle_call(Msg, From, State),
      From ! {reply, self(), Reply},
      loop(Module, NewState)
  end.
Implementation in standard library: gen_server

- More robust than my_gen_server
  - Timeouts and monitors to handle failures
- Init called in new process
- More events:
  - `handle_call` and `gen_server:call` for synchronous requests
  - `handle_cast` and `gen_server:cast` for asynchronous requests
  - `handle_info` for other messages
- `handle_call` can reply later (e.g. handle reply in other process)
- `callback terminate` for cleaning up
- `callback code_change` for handling dynamic code reloading
Example: gen_server (1/2)

```
-module(bounded_counter3).
-behavior(gen_server).
-export([start_link/1, increment/1, read/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, 
        terminate/2, code_change/3]).

-record(state, {limit, count}).

start_link(Limit) ->
    gen_server:start_link(?MODULE, [Limit], []).

increment(Server) ->
    gen_server:call(Server, increment).

read(Server) ->
    gen_server:call(Server, read).

init([Limit]) ->
    {ok, #state{limit = Limit, count = 0}}.
```
Example: gen_server (2/2)

```erlang
handle_call(increment, _From, State = #state{count = Counter, limit = Limit}) ->
  case Counter < Limit of
    true -> {reply, ok, State#state{count = Counter + 1}};
    false -> {reply, {error, limit_reached}, State}
  end;
handle_call(read, _From, State) ->
  {reply, State#state.count, State}.

handle_cast(_Msg, State) ->
  {noreply, State}.

handle_info(_Msg, State) ->
  {noreply, State}.

terminate(_Reason, _State) ->
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.
```
Error handling in Erlang

Two kinds of errors:

- **Predictable errors**
  - Wrong user input, connection problem, error reading file
  - Often handled with special return values, e.g.
    
    ```
    read_file(Filename) -> {ok, Binary} | {error, Reason}
    ```
  - Sometimes handled with exceptions

- **Unpredictable errors**
  - Software bugs, corrupt state, system resources exhausted
  - Handled by monitoring whole processes (⇒ supervisors)
Linked processes and monitoring

- Processes can be linked
  - A link has no direction
  - `spawn_link` spawns a new process and links it to the current
    - Also: `link` and `unlink` functions
  - If a process terminates, all linked processes are notified:
    - by default linked process terminates as well (with same reason)
    - if `process_flag(trap_exit, true)` is set, a special message `{'EXIT', Pid, Reason}` is sent instead

- Processes can be monitored
  - Only one direction
  - If monitored process terminates, monitoring process receives message `{'DOWN', MonitorRef, Type, Object, Info}`
Supervisors

- Start child processes (with link)
- Trap exits
- Handle termination of child processes (e.g. restart)
- Cleanly terminate applications
- Usually organized hierarchical
Generic Supervisor

Just implement callback `init/1` to specify the supervisor.

```
{ok, {SupFlags, [ChildSpec]}}.
```

SupFlags is a map with the following keys:

- **strategy**: Strategy for restarting children
  - `one_for_one`: Restart only terminated process (default value)
  - `one_for_all`: Restart all child processes
  - `rest_for_one`: Restart all processes, that were started after the terminating process
  - `simple_one_for_one`: Like `one_for_one`, but all children run the same code

- **intensity** (`MaxR`) and **period** (`MaxT`)
  - If more than `MaxR` number of restarts occur in the last `MaxT` seconds, the supervisor terminates all the child processes and then itself.
Supervisor Children

ChildSpec is a map with the following keys:

- **id**: Name of the child
- **start**: Tuple \{Module, Func, Args\} to call for initialization
- **restart**:
  - **permanent**: always restart
  - **temporary**: never restart
  - **transient**: restart only after crash
- **shutdown**: How long to terminate children
- **type**: worker or supervisor
- **modules**: [ModuleName] or dynamic (used for managing releases)

Children can be dynamically added and removed:

- **start_child**(SupRef, ChildSpec)
- **delete_child**(SupRef, Id)
Supervisor example

-module(example_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).
-export([stop/0]).

start_link() ->
    supervisor:start_link(?MODULE, []).

init(_) ->
    ChildSpecList = [child(service1), child(service2)],
    {ok,{{intensity => 2, period => 3600}, ChildSpecList}}.

cchild(Module) ->
    {id => Module, start => {Module, start_link, []},
     restart => permanent, shutdown => 2000}. 
Erlang OTP

- Generic servers \((\text{gen\_server})\)
- Generic Supervisors \((\text{supervisor})\)

More features:

- Generic state machine behavior \((\text{gen\_statem})\) (different states accept different messages)
- Generic event handling behavior \((\text{gen\_event})\) (multiple event handlers receive notification for one event)
- Applications, releases and release handling