

# Programming Distributed Systems

Erlang OTP

Annette Bieniusa, Albert Schimpf

AG Softech  
FB Informatik  
TU Kaiserslautern

Summer Term 2020

# 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

```
-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: ~pn", [Msg1])
```

```
    end,
```

```
    receive
```

```
        Msg2 -> io:format("Server 2 responded: ~pn", [Msg2])
```

```
    end.
```

Does this always work correctly?

## Example: Echo server 2

```
-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

```
-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.

```

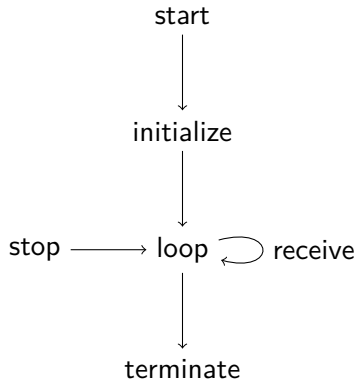


## Bounded Counter API (synchronous call)

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

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

# Generic Client/Servers



## Separating generic and specific parts

---

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

---

## Separating generic and specific parts

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

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)

```
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}.
```