Concurrency, Parallelism and Distribution (CPD)

Understanding concurrency
Intro to OTP

Joaquim Gabarro
(gabarro@cs.upc.edu)

Computer Science
BarcelonaTech, Universitat Politècnica de Catalunya

Basic reading

(1) Joe Armstrong
Programming Erlang, Software for a Concurrent World
Pragmatic Bookshelf, 2007.

Download the code from:

`https://pragprog.com/book/jaerlang/programming-erlang`

(2) Joe Armstrong, Erlang
CACM, September 2010, Vol 53, No 9, 68-75

(3) Jim Larson, Erlang for Concurrent Programming
CACM, March 2009, Vol 52, No 3, 48-56

Slides are based on Armstrong's book

Key-value dictionary, F.8 Module:dict

OTP

The Road to the Generic Server

Getting Started with gen_server

Key-value dictionary, F.8 Module: dict

First, recall some basic things.

```
...
erase(Key, Dict1) -> Dict2
   Erase a key from a dictionary.
fetch_keys(Dict) -> Keys
    Return all keys in a dictionary.
find(Key, Dict) -> {ok, Value} | error
    Search for a key in a dictionary.
new() -> dictionary()
   Create a dictionary.
store(Key, Value, Dict1) -> Dict2
   Store a value in a dictionary.
...
```

```
39> Dict1=dict:new().
{dict,0,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
      {{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]}}}
40>
40> Dict2=dict:store(joe, "at home", Dict1).
{dict,1,16,16,8,80,48, ...}
41>
41> dict:find(joe, Dict2).
{ok,"at home"}
42> Dict3=dict:store(kim, "at fib", Dict2).
{dict,2,16,16,8,80,48, ... }
43>
43> Keys1=dict:fetch_keys(Dict3).
[joe,kim]
44> Keys1.
[joe,kim]
45> Dict4=dict:erase(kim, Dict3).
{dict,1,16,16,8,80,48,...}
46>
46> Keys2=dict:fetch_keys(Dict4).
[joe]
47>
```

OTP introduction

- OTP stands for the Open Telecom Platform.
- The name is actually misleading, because OTP is far more general than you might think.
- It's an application operating system and a set of libraries and procedures used for building large-scale, fault-tolerant, distributed applications.
- It was developed at the Swedish telecom company Ericsson and is used within Ericsson for building fault-tolerant systems.

The Road to the Generic Server (Section 16.1)

This is the most important section in the entire book, so read it once, read it twice, read it 100 times—just make sure the message sinks in.

A behavior encapsulates common behavioral patterns—think of it as an application framework that is parameterized by a callback module.

We're going to write four little servers called server1, server2..., each slightly different from the last. The goal is to totally separate the nonfunctional parts of the problem from the functional parts of the problem. That last sentence probably didn't mean much to you now, but don't worry—it soon will. Take a deep breath....

Server 1: The Basic Server

It's a little server that we can parameterize with a callback module:

Download server1.erl

```
-module(server1).
-export([start/2, rpc/2]).

start(Name, Mod) ->
   register(Name,
        spawn(fun() -> loop(Name, Mod, Mod:init()) end)
       ).

rpc(Name, Request) ->
   Name ! {self(), Request},
   receive
      {Name, Response} -> Response
   end.
```

```
loop(Name, Mod, State) ->
   receive
      {From, Request} ->
         {Response, State1} = Mod:handle(Request, State),
      From ! {Name, Response},
      loop(Name, Mod, State1)
end.
```

This very small amount of code captures the quintessential nature of a server. Let’s write a callback for server1

Callback for server1

Download name_server.erl

```
-module(name_server).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server1, [rpc/2]).

%% client routines
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().
handle({add, Name, Place}, Dict) ->
    {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict) ->
    {dict:find(Name, Dict), Dict}.
```

Callback tasks

This code actually performs two tasks.

- It serves as a callback module that is called from the server framework code, and at the same time,
- itcontains the interfacing routines that will be called by the client.
- The usual OTP convention is to combine both functions in the same module.

Stop and think.

Just to prove that it works, do this:

```
1> server1:start(name_server, name_server).
true
2> name_server:add(joe, "at home").
ok
3> name_server:whereis(joe).
{ok,"at home"}
```

Now stop and think.

- The callback had no code for concurrency, no spawn, no send, no receive, no register.
- It is pure sequential code— nothing else.

This means we can write client-server models without understanding anything about the underlying concurrency. This is the basic pattern for all servers.

Server 2: A Server with Transactions

Download `server2.erl`

```
-module(server2).
-export([start/2, rpc/2]).

start(Name, Mod) ->...

rpc(Name, Request) ->
    Name ! {self(), Request},
    receive
        {Name, crash} -> exit(rpc);
        {Name, ok, Response} -> Response
    end
```

```
loop(Name, Mod, OldState) ->
   receive
      {From, Request} ->
         try Mod:handle(Request, OldState) of
            {Response, NewState} ->
               From ! {Name, ok, Response},
               loop(Name, Mod, NewState)
         catch
            _:Why ->
               log_the_error(Name, Request, Why),
               %% send a message to cause
               %% the client to crash
               From ! {Name, crash},
               %% loop with the *original* state
               loop(Name, Mod, OldState)
         end
   end.

log_the_error(Name, Request, Why) ->
   io:format("Server ~p request ~p ~n"
             "caused exception ~p~n" ,
             [Name, Request, Why]).
```

This one gives you transaction semantics in the server.

- It loops with the original value of State if an exception was raised in the handler function.
- But if the handler function succeeded, then it loops with the value of NewState provided by the handler function.

The callback module for this server is exactly the same as the callback module we used for server1.

- By changing the server and keeping the callback,we can change the nonfunctional behavior.
- Note: The last statement wasn't strictly true. We have to make a small change to the callback, that is to change the -import declaration from server1 to server2.

Server 3: A Server with Hot Code Swapping

Download `server3.erl`

```
-module(server3).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
   register(Name,
            spawn(fun() -> loop(Name,Mod,Mod:init()) end)
            ).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->
   Name ! {self(), Request},
   receive
      {Name, Response} -> Response
   end.
```

```
loop(Name, Mod, OldState) ->
   receive
      {From, {swap_code, NewCallBackMod}} ->
         From ! {Name, ack},
         loop(Name, NewCallBackMod, OldState);
      {From, Request} ->
         {Response, NewState} =
             Mod:handle(Request, OldState),
         From ! {Name, Response},
         loop(Name, Mod, NewState)
   end.
```

If we send a swap code message, then the server will change the callback module to the new module contained in the message.

We can demonstrate this by

1. starting server3 with a callback module and then
2. dynamically swapping the callback module.

- We can't use name_server as the callback module because we hard-compiled the name of the server into the module (look at -import(server1, [rpc/2])).
- So, we make a copy of this, calling it name_server1 where we change the name of the server (we take -import(server3, [rpc/2])).

Download name_server1.erl

```
-module(name_server1).
-export([init/0, add/2, whereis/1, handle/2]).
-import(server3, [rpc/2]).

%% client routines
add(Name, Place) ->
    rpc(name_server, {add, Name, Place}).
whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().
handle({add, Name, Place}, Dict) ->
    {ok, dict:store(Name, Place, Dict)};
handle({whereis, Name}, Dict) ->
    {dict:find(Name, Dict), Dict}.
```

First we'll start server3 with the name_server1 callback module:

```
1> server3:start(name_server, name_server1).
true
2> name_server:add(joe, "at home").
ok
3> name_server:add(helen, "at work").
ok
```

Now suppose we want to find all the names that are served by the name server.

1. There is no function in the API that can do this - the module name_server1 has only add and lookup access routines.
2. With lightning speed, we fire up our text editor and write a new callback module new_name_server

Download new_name_server.erl

```
-module(new_name_server).
-export([init/0, add/2, all_names/0,
         delete/1, whereis/1, handle/2]).
-import(server3, [rpc/2]).

%% interface
all_names() -> rpc(name_server, allNames).
add(Name, Place) -> rpc(name_server, {add, Name, Place}).
delete(Name) -> rpc(name_server, {delete, Name}).
whereis(Name) -> rpc(name_server, {whereis, Name}).

%% callback routines
init() -> dict:new().
handle({add, Name, Place}, Dict) ->
   {ok, dict:store(Name, Place, Dict)};
handle(allNames, Dict)->{dict:fetch_keys(Dict), Dict};
handle({delete, Name}, Dict)->{ok, dict:erase(Name, Dict)};
handle({whereis, Name}, Dict) ->
   {dict:find(Name, Dict), Dict}.
```

We compile this and tell the server to swap its callback module:

```
4> c(new_name_server).
{ok,new_name_server}
5> server3:swap_code(name_server, new_name_server).
ack
```

Now we can run the new functions in the server:

```
6> new_name_server:all_names().
[joe,helen]
```

Here we changed the callback module on the fly—this is dynamic code upgrade, in action before your eyes, with no black magic.

Now stop and think again

- The last two tasks we have done are considered to be very difficult.
- Servers with transaction semantics are difficult to write; servers with dynamic code upgrade are very difficult to write.

Traditionally:

- We think of servers as programs with state that change state when we send them messages.
- The code in the servers is fixed the first time it is called.
- If we want to change the code in the server, we have to stop the server and change the code, and then we can restart the server.

In the examples, the code in the server can be changed just as easily as we can change the state of the server.2

Server 4: Transactions and Hot Code Swapping

Download server4.erl

```
-module(server4).
-export([start/2, rpc/2, swap_code/2]).

start(Name, Mod) ->
   register(Name,
            spawn(fun() -> loop(Name,Mod,Mod:init()) end)
            ).

swap_code(Name, Mod) -> rpc(Name, {swap_code, Mod}).

rpc(Name, Request) ->
   Name ! {self(), Request},
   receive
      {Name, crash} -> exit(rpc);
      {Name, ok, Response} -> Response
end
```

```
loop(Name, Mod, OldState) ->
   receive
      {From, {swap_code, NewCallbackMod}} ->
         From ! {Name, ok, ack},
         loop(Name, NewCallbackMod, OldState);
      {From, Request} ->
         try Mod:handle(Request, OldState) of
            {Response, NewState} ->
               From ! {Name, ok, Response},
               loop(Name, Mod, NewState)
         catch
             _: Why ->
                log_the_error(Name, Request, Why),
                From ! {Name, crash},
                loop(Name, Mod, OldState)
          end
   end.

log_the_error(Name, Request, Why) ->
   io:format("Server ~p request ~p ~n"
   "caused exception ~p~n" ,
   [Name, Request, Why]).
```

Server 5: Even More Fun

Here's a server that does nothing at all until you tell it to become a particular type of server

Download `server5.erl`

```
-module(server5).
-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).

wait() ->
    receive
        {become, F} -> F()
    end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

If we start this and then send it a {become, F} message, it will become an F server by evaluating F().

We'll start it:

```
1> Pid = server5:start().
<0.57.0>
Our server
```

Our server does nothing and just waits for a become message.

Let's now define a server function. It's nothing complicated, just something to compute factorial:

Download my_fac_server.erl

```
-module(my_fac_server).
-export([loop/0]).

loop() ->
    receive
        {From, {fac, N}} ->
            From ! {self(), fac(N)},
            loop();
        {become, Something} ->
            Something()
    end.

fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

Just make sure it's compiled

```
2> c(my_fac_server).
{ok,my_fac_server}
3> Pid ! {become, fun my_fac_server:loop/0}.
{become,#Fun<my_fac_server.loop.0>}
```

Our process has become a factorial server, we can call it:

```
4> server5:rpc(Pid, {fac,30}).
265252859812191058636308480000000
```

Our process will remain a factorial server, until we send it a become, Something message and tell it to do something else.

Getting Started with gen_server (Section 16.2)

I'm going to throw you in at the deep end. Here's the simple three-point plan for writing a gen_server callback module:

1. Decide on a callback module name.
2. Write the interface functions.
3. Write the six required callback functions in the callback module.

This is really easy. Don't think—just follow the plan!

Step 1: Decide on the Callback Module Name

We're going to make a very simple payment system. We'll call the module my_bank.

Step 2: Write the Interface Routines

We'll define five interface routines, all in the module my_bank:

```
start()
   Open the bank.
stop()
   Close the bank.
new_account(Who)
   Create a new account.
deposit(Who, Amount)
   Put money in the bank.
withdraw(Who, Amount)
  Take money out, if in credit.
```

Each of these results in exactly one call to the routines in gen_server.

Download `my_bank.erl`

```
start() ->
   gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
stop() ->
   gen_server:call(?MODULE, stop).
new_account(Who) ->
   gen_server:call(?MODULE, {new, Who}).
deposit(Who, Amount) ->
   gen_server:call(?MODULE, {add, Who, Amount}).
withdraw(Who, Amount) ->
   gen_server:call(?MODULE, {remove, Who, Amount}).
```

- gen_server:start_link(local, Name, Mod, ...) starts a local server.4
- The macro ?MODULE expands to the module name my_bank.
- Mod is the name of the callback module.
- We'll ignore the other arguments to gen_server:start for now.
- gen_server:call(?MODULE, Term) is used for a remote procedure call to the server.

Step 3: Write the Callback Routines

We can use a number of templates to make a gen_server.

- The template contains a simple skeleton that we can fill in to make our server.
- The keyword -behaviour is used by the compiler so that it can generate warning or error messages if we forget to define the appropriate callback functions.

Download gen_server_template.mini

```
-module().
%% gen_server_mini_template
-behaviour(gen_server).
-export([start_link/0]).
%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2,
         handle_info/2, terminate/2, code_change/3]).

start_link() ->
   gen_server:start_link({local,?SERVER},?MODULE,[],[]).
init([]) -> {ok, State}.

handle_call(_Request, _From, State) ->
   {reply, Reply, State}.
handle_cast(_Msg, State) -> {noreply, State}.
handle_info(_Info, State) -> {noreply, State}.
terminate(_Reason, _State) -> ok.
code_change(_OldVsn, State, Extra) -> {ok, State}.
```

- We'll start with the template and edit it a bit.
- All we have to do is get the arguments in the interfacing routines to agree with the arguments in the template.
- The most important bit is the handle_call/3 function.
- We have to write code that matches the three query terms defined in the interface routines.
- That is, we have to fill in the dots in the following:

```erlang
handle_call({new, Who}, From, State} ->
   Reply = ...
   State1 = ...
   {reply, Reply, State1};
handle_call({add, Who, Amount}, From, State} ->
   Reply = ...
   State1 = ...
   {reply, Reply, State1};
handle_call({remove, Who, Amount}, From, State} ->
   Reply = ...
   State1 = ...
   {reply, Reply, State1};
```

- State is just a variable representing the global state of the server that gets passed around in the server.
- In our bank module, the state never changes; it's just an ETS table index that is a constant (although the content of the table changes).

For more detailson ETS look at chapter 15, *ETS and DETS Large Storage Mechanisms*

Download my_bank.erl

```
init([]) -> {ok, ets:new(?MODULE,[])}.

handle_call({new,Who}, _From, Tab) ->
   Reply = case ets:lookup(Tab, Who) of
                [] -> ets:insert(Tab, {Who,0}),
                      {welcome, Who};
                [_] -> {Who, you_already_are_a_customer}
           end,
   {reply, Reply, Tab};

handle_call({add,Who,X}, _From, Tab) ->...
```

So let's go visit the bank:

```
1> my_bank:start().
{ok,<0.33.0>}
2> my_bank:deposit("joe", 10).
not_a_customer
3> my_bank:new_account("joe").
{welcome,"joe"}
4> my_bank:deposit("joe", 10).
{thanks,"joe",your_balance_is,10}
5> my_bank:deposit("joe", 30).
{thanks,"joe",your_balance_is,40}
6> my_bank:withdraw("joe", 15).
{thanks,"joe",your_balance_is,25}
7> my_bank:withdraw("joe", 45).
{sorry,"joe",you_only_have,25,in_the_bank}
```