erlang

@chitacan

chapters

chapters

- (재미없을 것 같은)ETS, common test 를 다루는 25, 28 장은 대충 보고,
- 재미있을 것 같은 26, 27 장을 집중적으로 팠습니다.

chapter 25. Bears, ETS, Beets

ETS

- `Erlang Term Storage`
- efficient **in-memory** database included with the Erlang VM
- store large amounts of data with constant access time
- supports **ONLY tuples**
- no transactions
- DETS (disk-based ETS)

ETS 사용하기 (1)

```
$ erl
1> ets:new(ingredients, [set, named_table]).
ingredients
2> ets:insert(ingredients, {bacon, great}).
true
3> ets:lookup(ingredients, bacon).
[{bacon,great}]
4> ets:insert(ingredients, {bacon, awesome}, {cabbage, alright}).
true
3> ets:lookup(ingredients, bacon).
[{bacon,awesome}]
4> ets:delete(ingredients, cabbage).
true
5> ets:lookup(ingredients, cabbage).
[]
```

ETS 사용하기 (2)

```
$ erl
1> ets:new(table, [ordered_set, named_table]).
table
2> ets:insert(table, [{a, 10}, {z, 70}, {c, 20}]).
true
3> ets:first(table).
a
4> ets:next(table, a).
c
4> ets:last(table).
z
```

ETS match & select

```
$ erl
1> ets:new(table, [bag, named_table]).
table
2> ets:insert(table, [{items,1,1},
                      {items,6,7},
                      {items,3,4},
                      {items,1,2}]).
true
3> ets:match(table, {items, '$1', '$1'}).
[[1]]
4> ets:match_object(table, {items, '$1', '$1'}).
[{items,1,1}]
5> ets:select(table, [{{items,'$1','_'},[{'>','$1',5}],['$_']}]).
[{items,6,7}]
6> ets:select(table, ets:fun2ms(fun(N = {items,X,_}) when X > 5 -> N end)).
[{items,6,7}]
```

chapter 26. Distribunomicon

Erlang 의 매력?

드디어 3번째 매력을 학습할 차례!!

- functional language
- semantics for concurrency
- **supports distribution**

"Distributed programming is like being left alone in the dark, with monsters everywhere."

Bad news

Distributed Erlang is **still** leaving you alone in the dark.

Good news

Erlang gives you some tools. (not solutions)

- many nodes (VMs) can connect & communicate with each other
- extending multiple process concept to many nodes

Tools

Erlang gives you

- `net_kernel`
- `global`
- `rpc`
- `remote shells`
- `dist_ac`
- …

Fallacies of Distributed Computing

- Erlang 이 주는 툴들을 이애하기 위해서는 분산환경에서 발생하는 문제들에 대해서 알고 있을 필요가 있습니다.
- 똑똑한 분들이 분산환경에서 잘못될 수 있는 가정들을 8가지로 정리해 놓았습니다.
- http://www.rgoarchitects.com/Files/fallacies.pdf

1. The network is reliable
2. There is no latency
3. Bandwidth is infinite
4. The network is secure
5. Topolgy doesn't change
6. There is only one admin
7. Transport cost is zero
8. The network is homogeneous

CAP Theorem

Consistency, **A**vailability, **P**arition tolerance 를 모두 만족하는건 불가능하다...

connecting nodes

```
$ erl -sname hi
(hi@cmbp)1>
```

```
$ erl -sname ho
(ho@cmbp)1> net_kernel:connect_node(hi@cmbp).
true
(ho@cmbp)2> net_kernel:i(hi@cmbp).
Node      = hi@cmbp
State     = up
Type      = normal
In        = 17
Out       = 16
Address   = 127.0.0.1:60313
ok
(ho@cmbp)2> nodes()
[hi@cmbp]
```

책에 나온 `net_kernel:connect` 는 없어진듯?

epmd

Erlang Port Mapper Daemon

- node (VM) 이 실행될때 같이 실행 되는 프로세스
  - `-start_epmd false` 옵션으로 epmd 실행을 제한할 수 있습니다.
- 하나의 시스템에 하나만 실행됩니다.
- 기본적으로 4369 포트를 사용합니다.
- 이름이 주어진 모든 node 는 epmd 에 자신의 이름을 등록합니다.
  - node 에 대한 port 정보는 epmd 를 통해 조회합니다.
  - 새로운 node 는 cluster 에 속한 node 중 하나라도 연결되면 나머지 node 의 이름을 알 수 있습니다.

sending messages

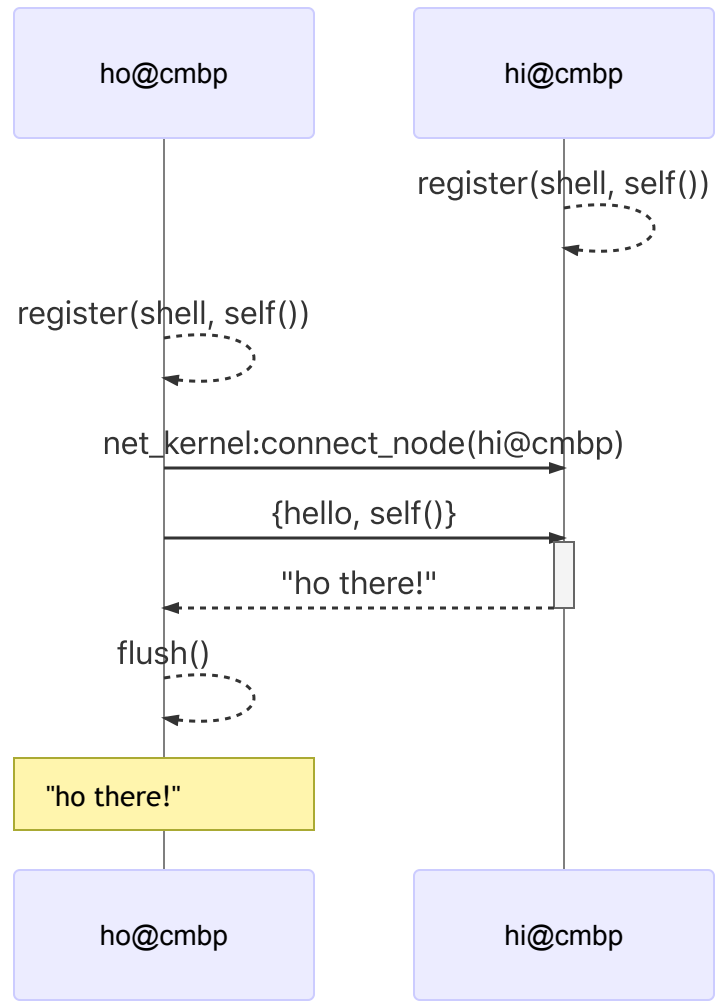
```
$ erl -sname hi
(hi@cmbp)1> register(shell, self()).
true
```

```
$ erl -sname ho
(ho@cmbp)1> register(shell, self()).
true
(ho@cmbp)2> net_kernel:connect_node(hi@cmbp).
true
(ho@cmbp)3> {shell, hi@cmbp} ! {hello, self()}.
```

```
(hi@cmbp)2> receive {hello, Other} -> Other ! "ho there!" end.
"ho there!"
```

```
(ho@cmbp)4> flush().
Shell got "ho there!"
ok
```

sending messages



spawn

```
$ erl -sname hi
(hi@cmbp)1>
```

```
$ erl -sname ho
(ho@cmbp)1> spawn(hi@cmbp, fun() ->
(ho@cmbp)1>    io:format("I'm on ~p~n", [node()])
(ho@cmbp)1> end).
I'm on hi@cmbp
<8490.145.0>
```

remote shells

Erlang shell 에서 `ctrl-G` 을 통해 다른 node 에 shell 을 실행할 수 있습니다.

- shell 은 별도의 프로세스로 실행됩니다.
- auto-completion 이 지원되지 않습니다.
- `q()` 또는 `init:stop()` 을 호출하면 node 가 종료되니 주의해야 합니다.

```
(hi@cmbp)1>
User switch command
 --> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
 --> j
   1   {shell,start,[init]}
   2 * {ho@cmbp,shell,start,[]}
 --> c
(ho@cmbp)1>
```

rpc module

remote procedure call

원격 node 의 함수를 MFA 스타일로 호출할 수 있습니다.

```
$ erl -sname hi
hi@cmbp1>
```

```
$ erl -sname hu
hu@cmbp1>
```

```
$ erl -sname ho
ho@cmbp1> rpc:call(hi@cmbp, lists, sort, [[a,e,f,t,h,s,a]]).
[a,a,e,f,h,s,t]
ho@cmbp2> Key = rpc:async_call(hi@cmbp, erlang, node, []).
<0.96.0>
ho@cmbp3> rpc:yield(Key).
hi@cmbp
ho@cmbp4> rpc:multicall([hi@cmbp, hu@cmbp], erlang, is_alive, []).
{[true,true],[]}
```

net_kernel:connect_node 를 미리 호출하지 않아도 됩니다.

rpc 🧙

code 모듈을 사용하면 원격 node 에 모듈을 전송할 수 있습니다.

```
-module(my_module).
-export([say/1]).
say(Str) -> io:format(Str++" world~n").
```

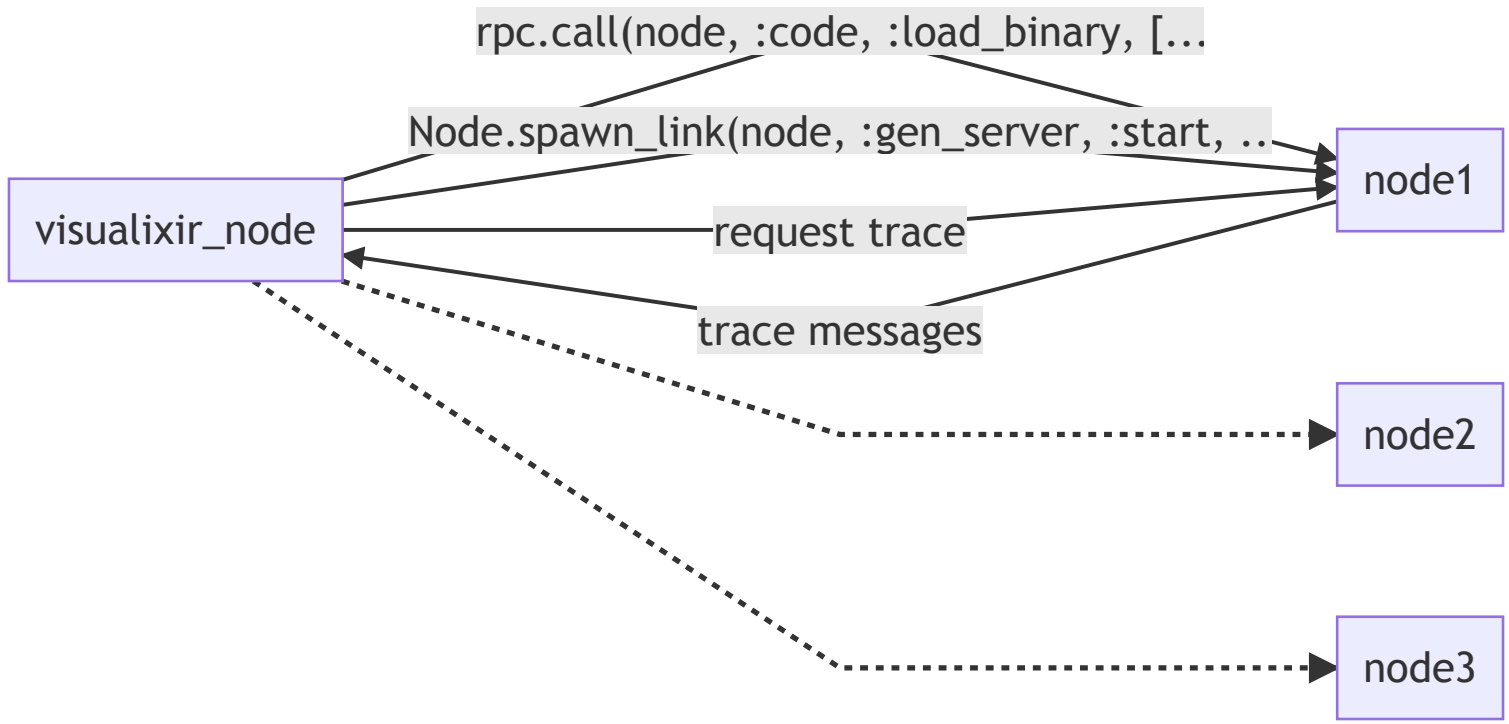
```
$ erl -sname hi
hi@cmbp1> c(my_module).
hi@cmbp2> {M, B, F} = code:get_object_code(my_module).
{my_module,<<70,79,82,49,...>>, "/root/my_module.beam"}
hi@cmbp3> rpc:call(ho@cmbp, code, load_binary, [M, F, B]).
{module,my_module}
hi@cmbp4> rpc:call(ho@cmbp, my_module, say, ["hello"]).
hello world
ok
```

```
ho@cmbp1> my_module:say("ho").
hello ho
ok
```

visualixir (1)

https://github.com/koudelka/visualixir

visualixir (2)



visualixir (3)

```
end

def pid_from_string(string) do
  string
  |> :erlang.binary_to_list()
  |> :erlang.list_to_atom()
  |> :erlang.whereis()
end

def send_module(node) do
  {module, binary, file} = :code.get_object_code(__MODULE__)
  :rpc.call(node, :code, :load_binary, [module, file, binary])
end

def cleanup(node) do
  :rpc.call(node, :code, :delete, [__MODULE__])
  :rpc.call(node, :code, :purge, [__MODULE__])
end
end
```

원격 node 로 모듈 자신을 전송합니다.

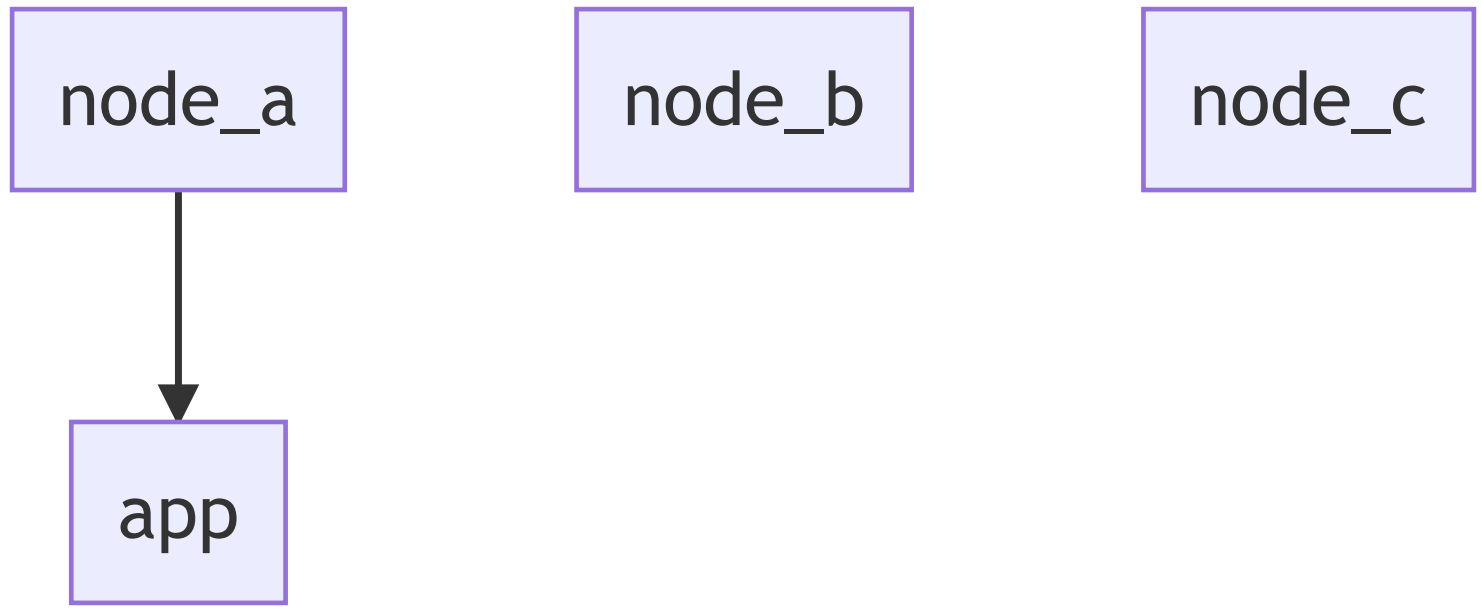
읽은 소감?

- 확실히 Erlang 은 여러가지 도구(`node`, `rpc` ...)를 제공하지만, 해결책을 정확하게 주지는 않습니다 ㅠ
- Erlang 이 제공하는 도구들을 활용해 분산환경에서 구현하고자 하는 앱의 컨셉 (`CAP Theorem`) 과 그 과정에서 발생하는 문제(`Fallacies of Distributed Computing`)에 대응할 수 있어야 한다고 생각합니다.

chapter 27. Distributed OTP Applications

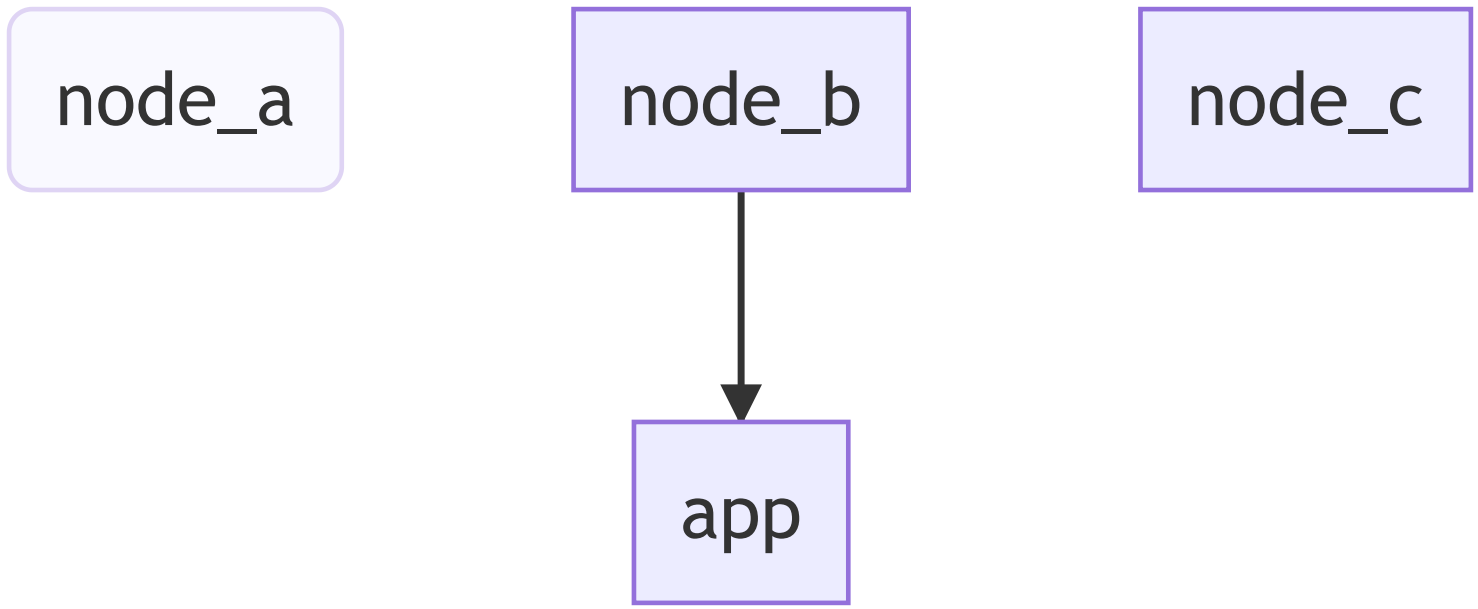
failover & takeover

failover



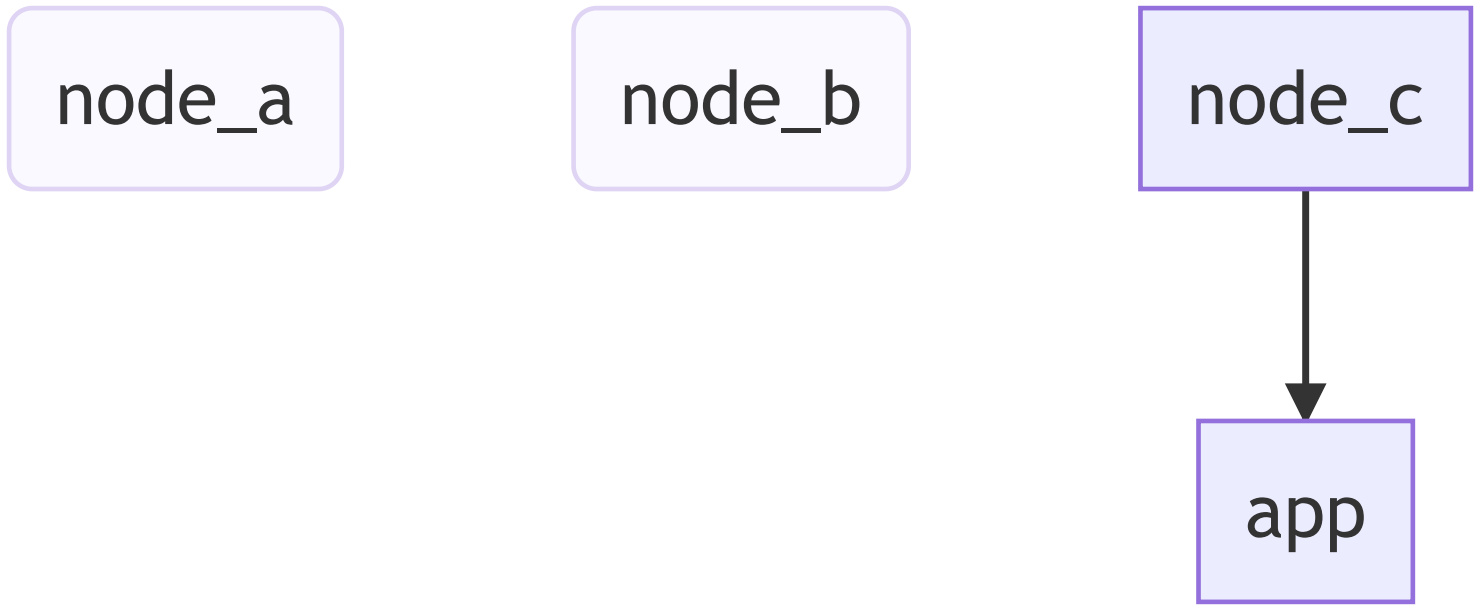
failover & takeover

failover

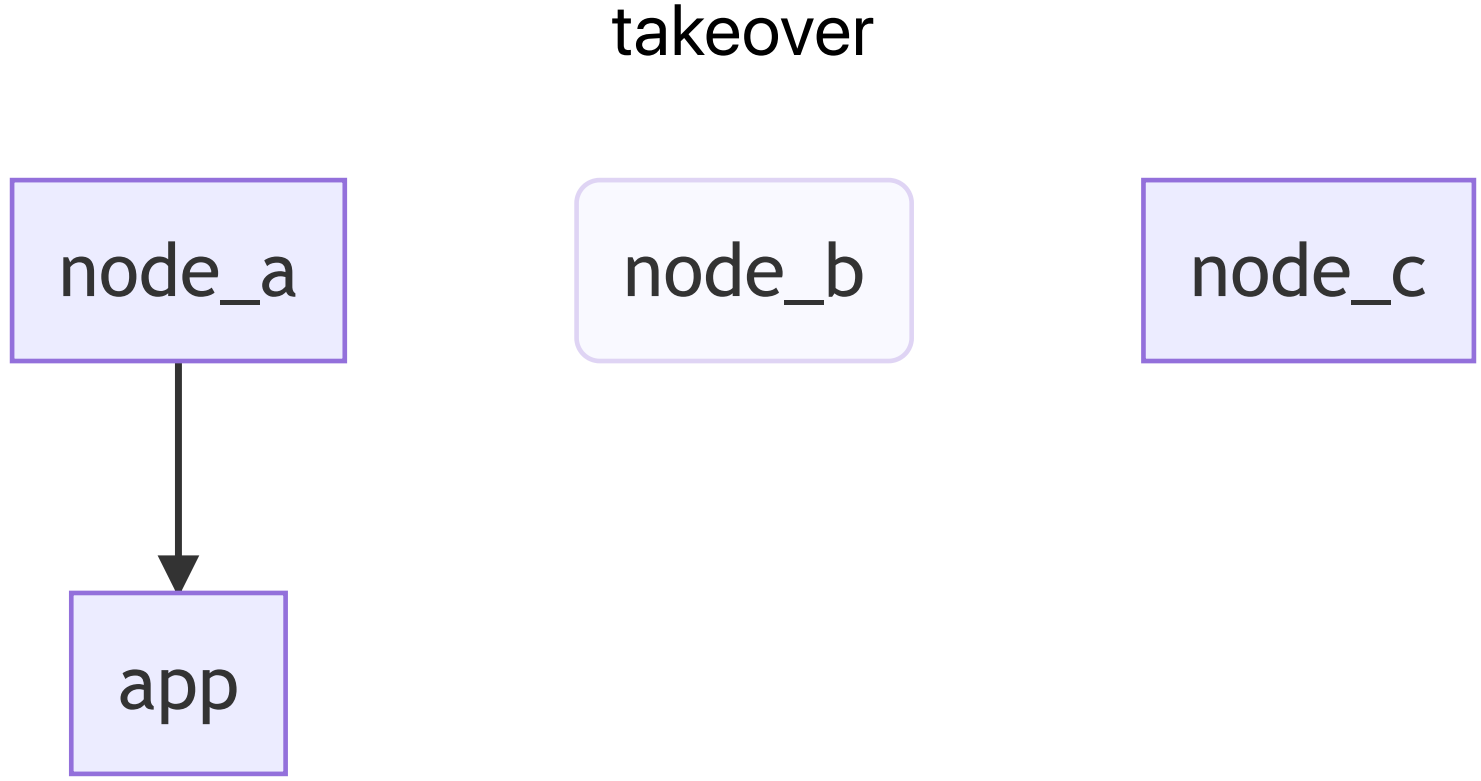


failover & takeover

failover

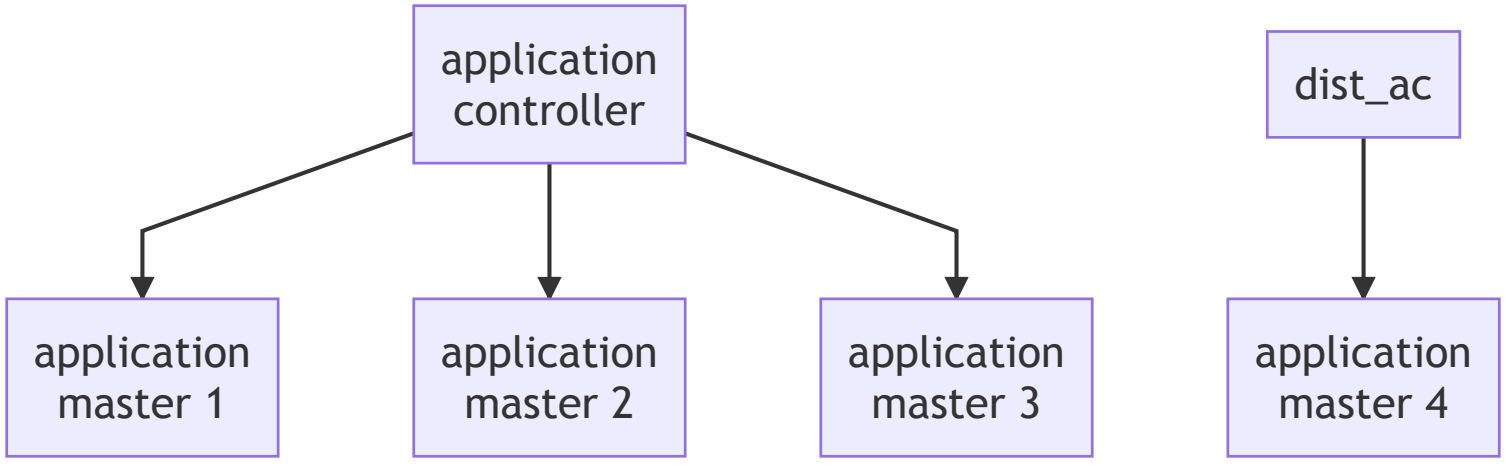


failover & takeover



failover & takeover

`dist_ac` (distributed application) 프로세스를 사용하면 failover & takeover 를 구현할 수 있습니다.



failover & takeover

`dist_ac` 를 사용하기 위해서는 config 파일에 `distributed` 설정이 필요합니다.

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@cmbp, {b@cmbp, c@cmbp}]}]},
   {sync_nodes_mandatory, [b@cmbp, c@cmbp]},
   {sync_nodes_timeout, 30000}
  ]
 }
].
```

http://erlang.org/doc/design_principles/distributed_applications.html

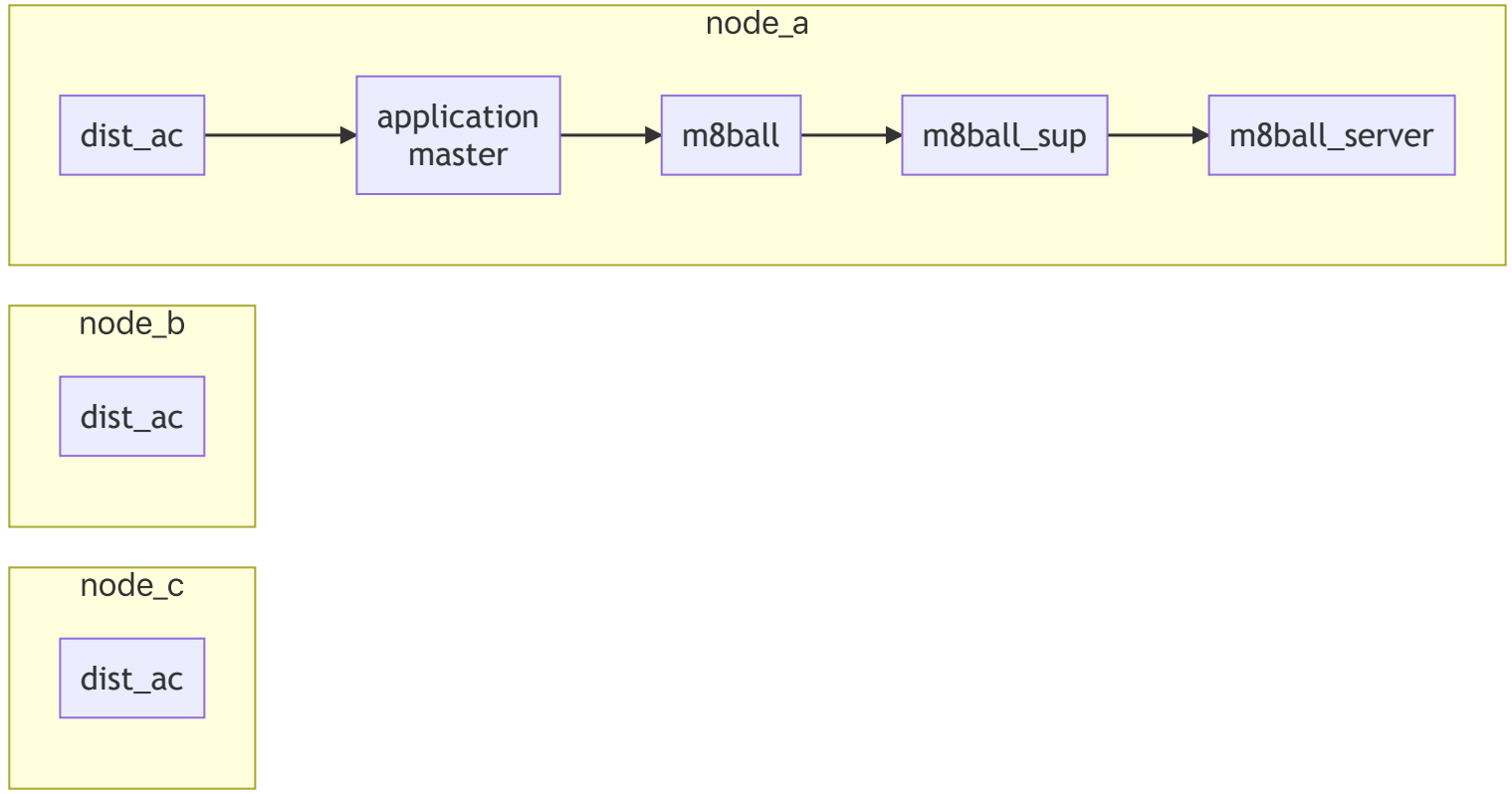
m8ball

env 에 설정된 값을 랜덤하게 출력하는 아주 간단한 앱.

```
{application, m9ball,
 [{vsn, "1.0.0"},
  {description, "Answer vital questions"},
  {modules, [m8ball, m8ball_sup, m8ball_server]},
  {applications, [stdlib, kernel, crypto]},
  {registered, [m8ball, m8ball_sup, m8ball_server]},
  {mod, {m8ball, []}},
  {env, [
    {answers, {<<"Yes">>, <<"No">>, <<"Doubtful">>,
               <<"I don't like your tone">>, <<"Of course">>,
               <<"Of course not">>, <<"*backs away slowly and runs away*">>}}
  ]}
 ]}.
```

```
$ erl -make
$ erl -pa ebin
1> m8ball:ask("hi").
<<"Doubtful">>
```

m8ball


node_a
dist_ac
application master
m8ball
m8ball_sup
m8ball_server
node_b
dist_ac
node_c
dist_ac

m8ball

distributed 설정덕분에 3개의 노드에서 모두 접근이 가능합니다.

```
$ erl -sname a -config config/a -pa ebin
(a@cmbp)1>
```

```
$ erl -sname b -config config/b -pa ebin
(b@cmbp)1>
```

```
$ erl -sname c -config config/c -pa ebin
(c@cmbp)1> m8ball:ask("hi").
<<"Of course not">>
```

m8ball_server.erl

```erl
-behaviour(gen_server).
-export([start_link/0, stop/0, ask/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

%%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%%
start_link() ->
    gen_server:start_link({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:call({global, ?MODULE}, stop).

ask(_Question) -> % the question doesn't matter!
    gen_server:call({global, ?MODULE}, question).
```

`pid` 대신에 `{global, NAME}` 을 사용해, 다른 node 에서도 `m8ball:ask`를 호출 할 수 있습니다.

m8ball.erl

takeover 때 실행되어야 하는 callback 을 추가합니다.

`m8ball`

- 설정에 명시된 3개의 node 가(`a`, `b`, `c`) 모두 실행되어야지만 앱이 시작됩니다.
  - `m8ball` 앱은 하나의 node 에서만 실행됩니다.
- 앱이 실행중인 node 가 종료되면, 다음 node 로 failover 됩니다.
- node 가 다시 실행되면, 기존 node 로 takeover 됩니다.
- 완벽한가? 🤔 몇번 실행하다보면 takeover 가 되지 않는 경우가 있었습니다...

읽은 소감?

- 설정만으로 distributed application 을 실행할 수 있는 것은 좋았지만,
  - 설정으로 인해 변하는 앱의 동작들을 좀 더 확인하고 싶어졌습니다.
  - `elixir` 에서는 어떻게 사용할 수 있는지 궁금해 졌습니다.
  - 이미 내장되어 있는 툴이 있는데, 왜 libcluster, swarm, horde 를 만드는지 궁금해 졌습니다.
- 과연 이것만으로 `Fallacies of Distributed Computing` 의 문제들을 해결할 수 있을지 🤔

끝