IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

RÉPUBLIQUE
FRANÇAISE
Liberté
Égalité
Fraternité

Elixir

concurrency & distribution

Fabien Dagnat

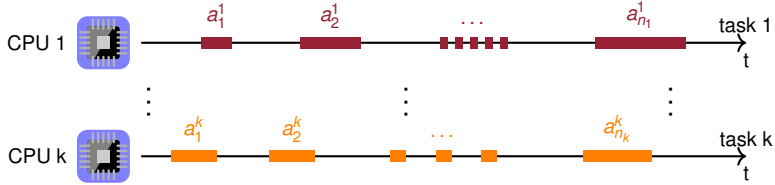
ILSD – FIAB – 2– 2025-2026

Tasks

- A task is a *sequence* of actions
- Multiple forms
  - architectural abstraction
  - structure of the code (require a specific syntax or an API)
  - at runtime (require runtime support)
- Various granularity
  - On different computing devices
  - Known of the OS
  - Managed by a VM / a runtime
- Several names
  - task, activity, ...
  - process, lightweight process, thread, ...

Tasks and execution

- Abstractly tasks are independant
  - if enough processors / cores, simultaneous execution



  - else interleaving

  - Often a mix
- If several tasks are running simultaneously: parallelism
  - it meeds several CPU

Concurrency

- If two tasks share a resource, they are in concurrency to use it
- Examples
  1. two machines access a server
  2. two processes on one machine share the same OS
  3. two different application *tasks* share data on disk
  4. two *tasks* of one application share data in memory
- The tasks must be coordinated: synchronization

Example of problems

- In case 4 of the previous list
  - The data is a pair of coordinates x and y
  - Task 1 wants to update them to the values x' and y'
  - Task 1 starts by updating x to x'
  - Task 2 consults the coordinates and obtains x', y!
  - Task 1 updates y to y'
- A concurrent architecture / application contains elements to prevent such problems

Solutions

- In languages with a lot of sharing (C, Java, Python), protection mechanisms are needed
  - the concept of lock and mutual exclusion (*mutex*)
  - the first task takes a lock and modify x then y
  - if the second task tries to access x or y, she must take the lock and is blocked
  - task 1 unlock the lock, it releases any task blocked on this lock
- When the language rely on immutable data, the problem is more on the coordination

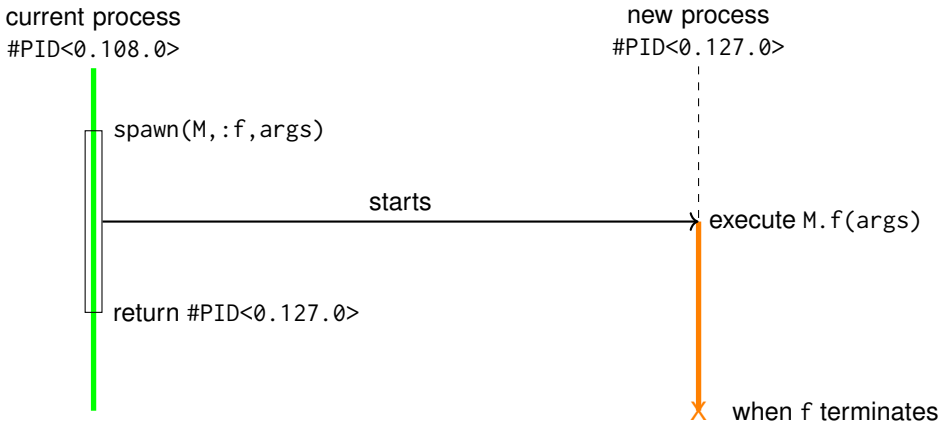
Actors

- The Erlang virtual machine (and therefore Elixir) rely on the concept of actors
- In this context
  - an application is a set of actors that cooperate
  - communication takes place through the exchange of messages between actors
  - an actor is an independent task with a mailbox
  - messages are asynchronous; when sent, they are placed in the mailbox(es) of the recipient(s)
  - An actor processes the messages in its mailbox when it wants to
  - Actors do not share data (everything is in the messages)
- Algorithms must be adapted (sharing and asynchronism)
- In Elixir, actors are called processes

Processes

- A process has an identity given by the VM
- Such an identity cannot be forged ⇒ you need to know someone knowing it to contact a process
- All computation takes place in a process (`self/0`)
  - when starting execution a first (root) process is created
- `spawn(Module,:fun_name,[])`
  - creates a new process
  - starts the process that executes `Module.fun_name()`
  - return the process identifier (*a.k.a* PID)
- Very lightweight model
  - many processes can be created (on my machine with default config: 1048517)
  - creating a process is efficient (intersting even for low computing functions)

Creating a process



Sending and receiving messages

- Sending is done using the function `send/2`
  - first argument is the PID of the recipient
  - second is the message
- Often a message is a tuple with an atom as first element
  - ex: `{:connect, user, pid}`
- The message is put in recipient mailbox and the sender continue its execution
- Receiving is done using **`receive do`** *pattern matching* **`end`**
  - the mailbox is browsed starting by the older message
  - each message is matched with each pattern case
  - the first message satisfying a pattern is removed from the mailbox
  - the expression corresponding to this case is evaluated
  - if no message match, the caller is blocked

An example

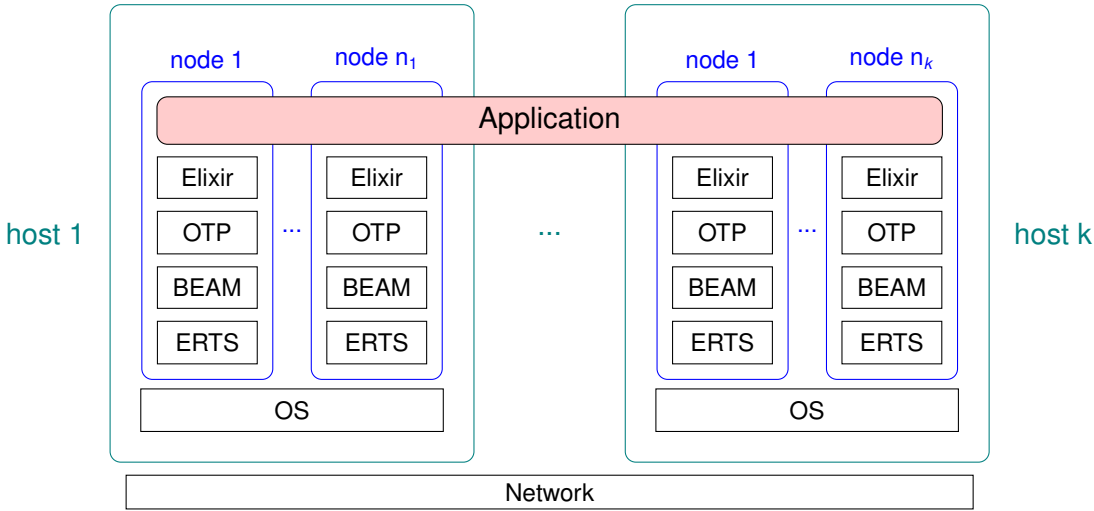
```
defmodule Pong do
 def listen do receive do :ping -> IO.puts "received ping" end end
 def listen_and_reply do
  receive do {sender, :ping} -> send(sender, :pong) end
 end
end
pid1 = spawn(Pong, :listen, [])
send(pid, :ping)
pid2 = spawn(Pong, :listen_and_reply, [])
send(pid2,{self, :ping})
receive do v -> inspect v end
```

Another example

```
defmodule Pong do
 def listen_and_reply_always do
  receive do
   {sender, :ping} -> send(sender, :pong); listen_and_reply_always()
  end
 end
end
pid = spawn(Pong, :listen_and_reply_always, [])
send(pid,{self, :ping})
receive do v -> inspect v end
send(pid,{self, :ping})
receive do v -> inspect v end
```

How to stop such a server?

L'architecture d'exécution



Demo

- `iex --sname` *name* or `iex --name` *name*
  - the first node on a host starts `epmd`
- `Node.self/0`
- `Node.list/0`
  - the connection must be explicit
- `Node.connect/1`, atom name of the node
  - bidirectional connection
  - must share a common secret
- `Node.spawn/2`, atom name of the node + function with no argument
  - concept of group leader for IO

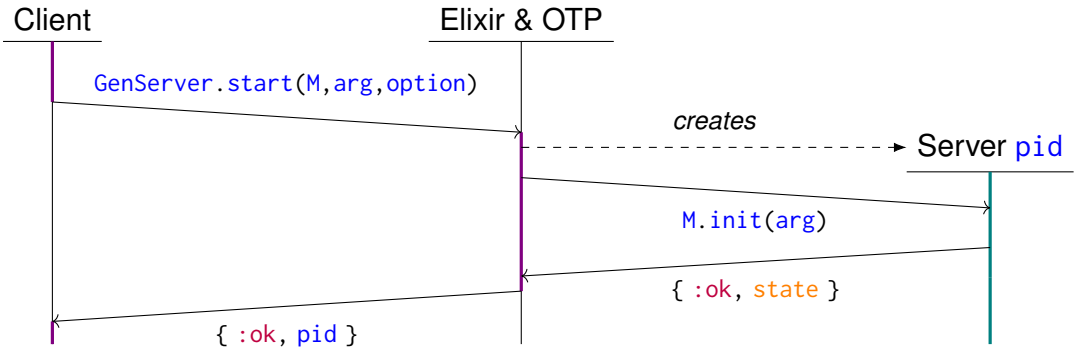
OTP

- Framework providing abstractions to build robust distributed applications
- An application is a set of processes following a behaviour
- A behaviour is a set of callback functions that must be defined
- These functions
  - are assembled by OTP
  - have standardized names
  - perform initialization, message management and state transitions, termination
- In this UE, only `GenServer` and `Supervisor`

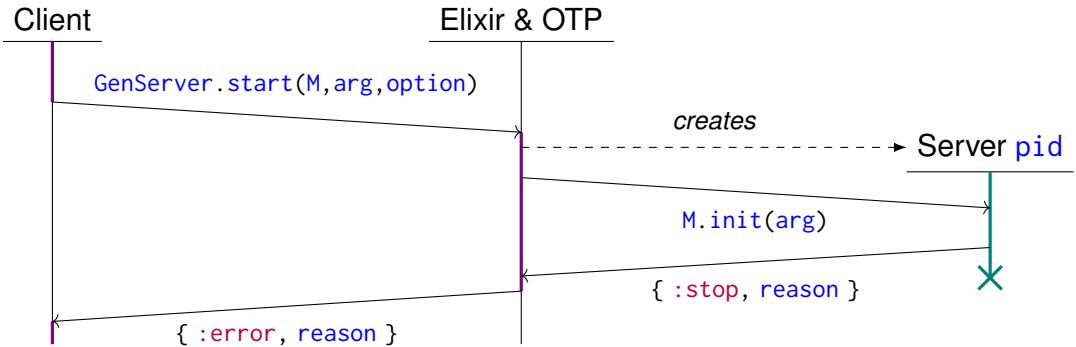
The concept of *callback*

- Objective: defining a library function using a user defined function
- The library contains
  ```
  def run(module, args) do module.do_run(args) end
  ```
- The user must provide the module containing the function `do_run`
  ```
  defmodule Example1 do
    def do_run(args) do IO.puts "Example1 running with #{inspect args}" end
  end
  ```
- Called by `run(Example1,2)`
- Allow to provide library function that may be adapted by the user
- Rely on naming conventions

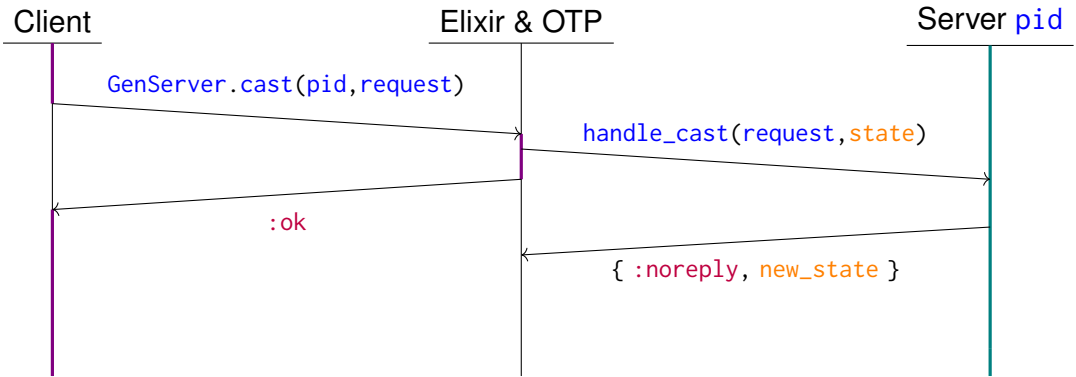
GenServer: starting the server



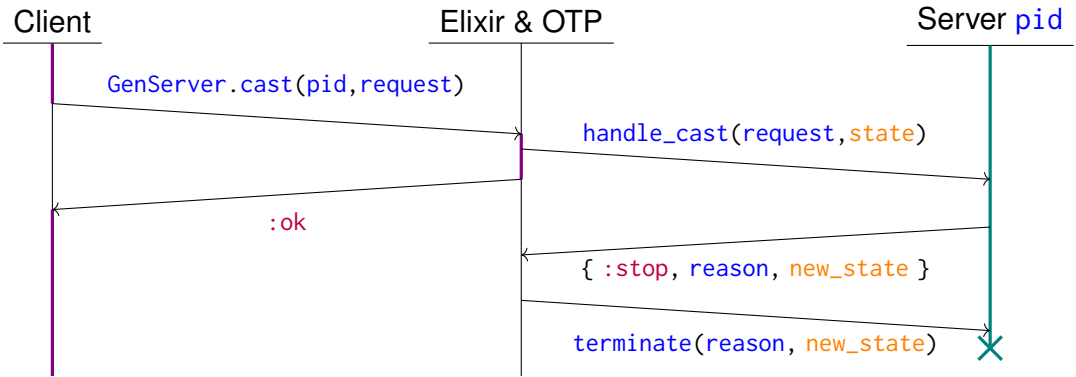
GenServer: starting the server



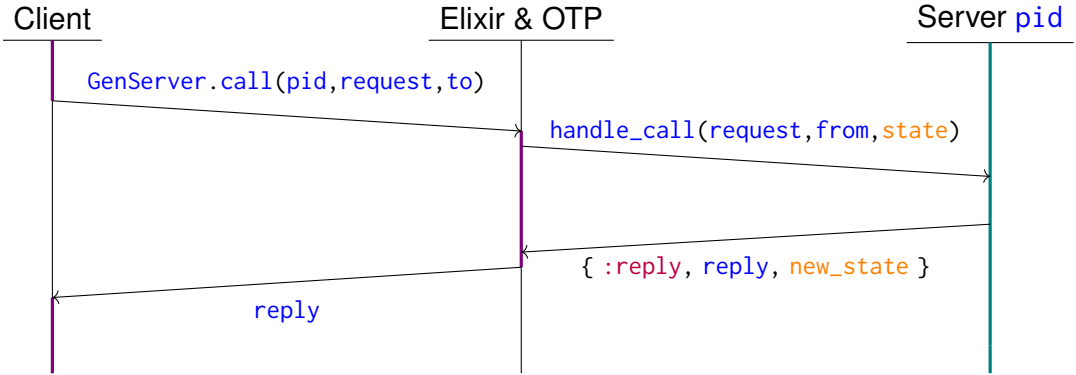
GenServer: asynchronous messages



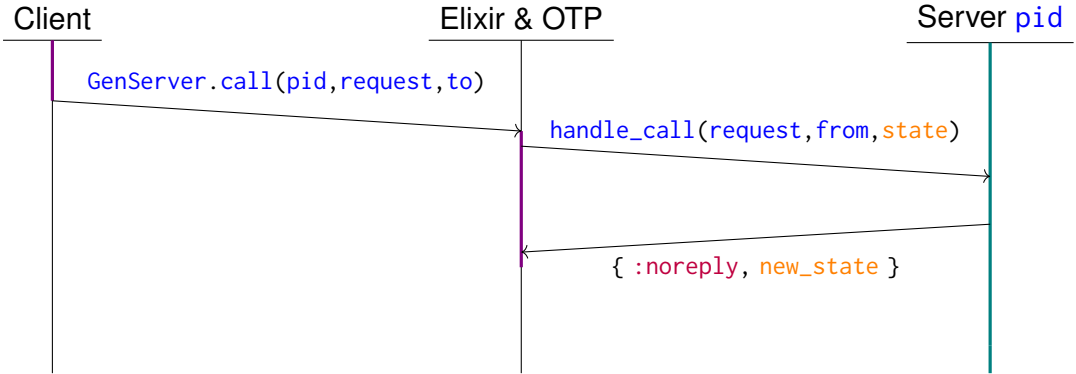
GenServer: asynchronous messages, error



GenServer: synchronous messages

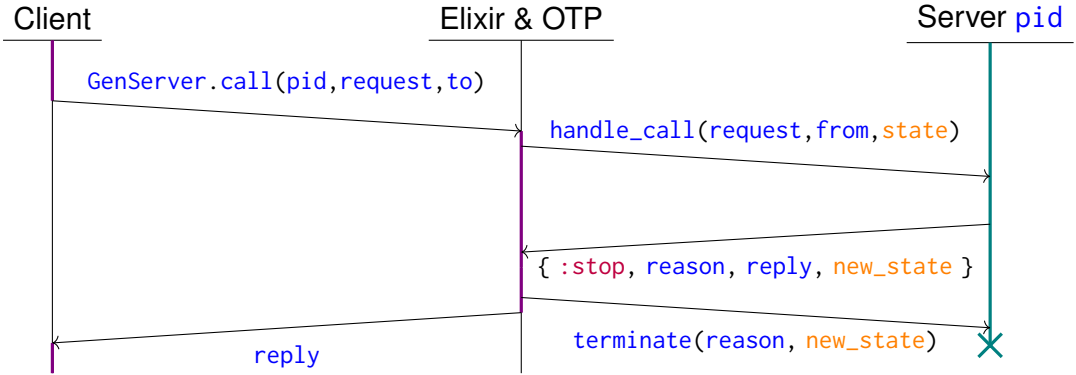


GenServer: synchronous messages, no reply



- a process must call `reply/2` to release the client

GenServer: synchronous messages, error



GenServer

- All other messages are handled by the `handle_info/2` callback
- Rely on the Elixir concept of behaviour

```
defmodule GenServer do
  @callback init(init_arg) :: {:ok, state} | {:stop, reason}
  @callback handle_call(request, from, state) :: {:reply, reply, new_state}
    | {:noreply, new_state} | {:stop, reason, reply, new_state}
  @callback handle_cast(request, state) :: {:noreply, new_state}
    | {:stop, reason, new_state}
  @callback handle_info(msg, state) :: {:noreply, new_state} | {:stop, reason, new_state}
  @callback terminate(reason, state)
end
```

Defining a server

```
defmodule CountServer do
  use GenServer
  def init(val) do {:ok, {val, val}} end
  def handle_call(:next, _, {curr, val}) do {:reply, curr, {curr + 1, val}} end
  def handle_cast(:reset, {_, val}) do {:noreply, {val, val}} end
  ...
```

- All `GenServer` callbacks are optional (they have a default behaviour)
- Often, the `GenServer` calls are hidden behind a business interface

```
  ...
  def start(number) do GenServer.start(__MODULE__, number, name: :count) end
  def next do GenServer.call(:count, :next) end
  def reset do GenServer.cast(:count, :reset) end
end
```

Conclusion

- This week concurrent and distributed part
  - Read elixir site!
    - `https://p4s.enstb.org/elixir`
- For next week
  - Questions
  - finish TP + homework
- Next week, start of the project
- Learning by/through practice