Concurrency and Distribution

In the last three lectures we have seen:

- Elixir concurrency (send, receive, ...)
- Elixir fault handling (links, monitors, ...)
- Elixir distribution (node-node communication)

We have seen examples of Elixir programs using these primitives to implement rather complex behaviour in an elegant and compact fashion

- So, everything is perfect!
- Or? ...

POLITÉCNICA

Concurrency and Distribution

Let us compare with non-concurrent programming in Elixir:

- **Everything** can be programmed using just recursion
- **But** to prevent re-inventing code (and re-inventing bugs!) we prefer to use "higher-level" libraries like *Enum*, *List*, etc

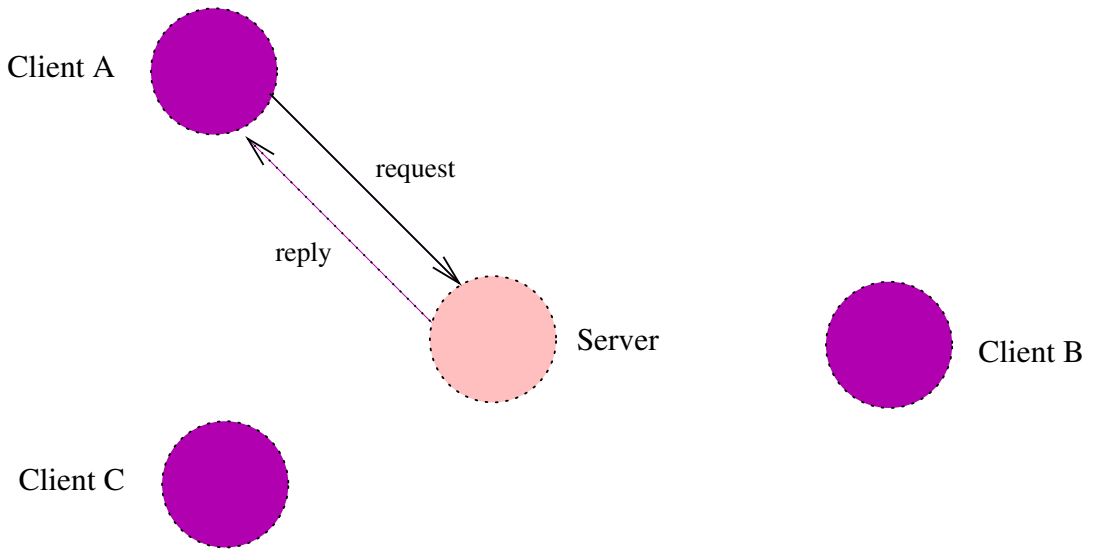
What are the "higher-level" libraries for concurrency in Elixir?

- Today: from Erlang – OTP: Open Telecom Platform:
  - *GenServer* – a library for client/server applications
  - *Supervisor* – a library to program fault-tolerant applications
- Later lectures:
  - *Agent*, *Task*, *PubSub*, *GenStage*, ...

POLITÉCNICA

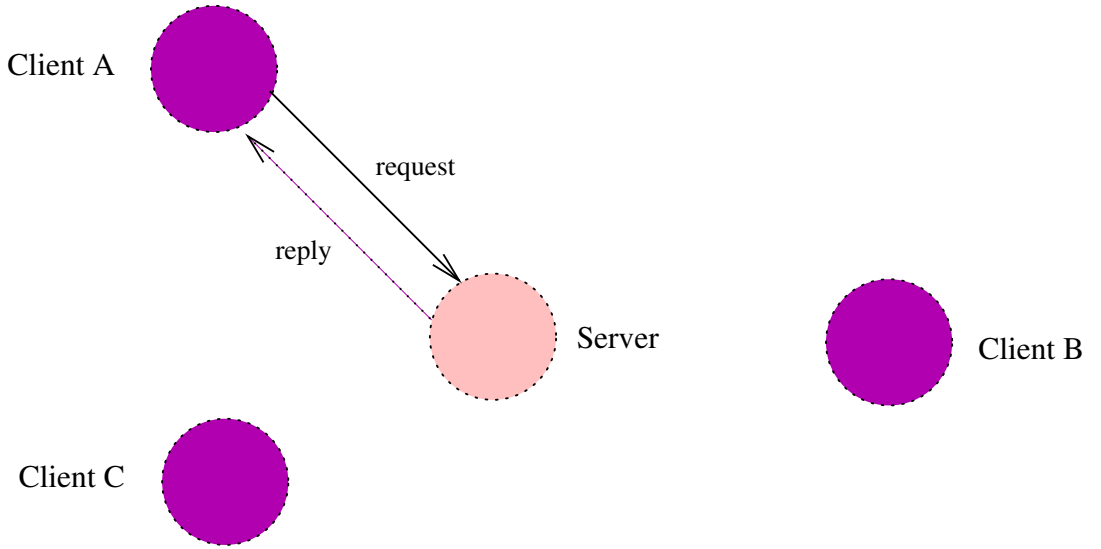
The *GenServer* – Generic Server – Library

- Generic Servers provide a standard way to implement client–server interactions
- One server interacts with an arbitrary number of clients:



POLITÉCNICA

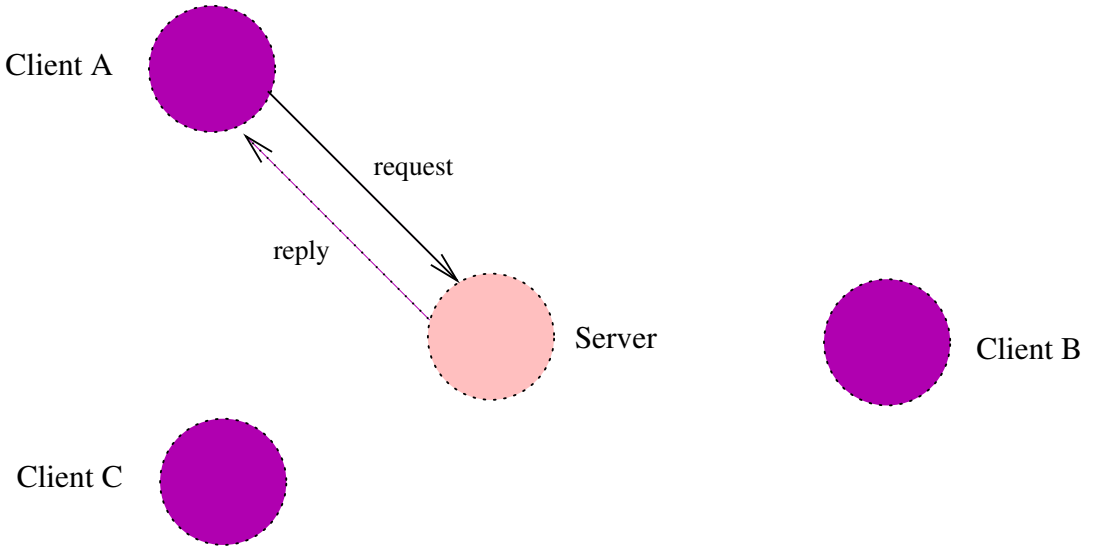
The *GenServer* – Generic Server – Library



- Clients makes *requests* to the server, which optionally *replies*
- Clients *block* until the server replies, or make *non-blocking* calls
- A server has a state, which is preserved between requests
- A server handles one call at a time

POLITÉCNICA

The *GenServer* – Generic Server – Library



- Clients makes *requests* to the server, which optionally *replies*
- Clients *block* until the server replies, or make *non-blocking* calls
- A server has a state, which is preserved between requests
- A server handles one call at a time
  - however a server may postpone the "return" from a call until later, thus permitting to handle later calls first
  - Example: a server implementing a "counter" with initial value 0, and with two operations:
    - `inc` – never blocks
    - `dec` – blocks until the value of the counter is positive

POLITÉCNICA

Accessing a GenServer from a Client

- Creating a Generic Server defined by the module *MyServer*:

```
# Start a GenServer registered as MyServer
GenServer.start_link(MyServer, InitArg, name: MyServer)
```

- A blocking call to the server:

```
# request can be any value
# timeout in milliseconds for when call fails
result = GenServer.call(MyServer, request, timeout)
```

- A non-blocking call to the server (no return value):

```
GenServer.cast(MyServer, request)
```

POLITÉCNICA

Definining a GenServer

The actions of the GenServer is defined in a number of functions that the user must define:

```
defmodule MyServer do
  use GenServer     # Declare that the module uses
                    # the GenServer library

  def init(InitArg)     # Returns the initial state
                        # of the server

  def handle_call(...) # Handles a blocking call
  def handle_cast(...) # Handles a non-blocking call
end
```

POLITÉCNICA

Handling Blocking Calls

```
def handle_call(request, from, state) do
  ...
end
```

- `request` is the argument from *`GenServer`*`.call`
- `from` represents the caller
- `state` is the state when the call was made

POLITÉCNICA

Handling Blocking Calls: Values Returned

```
def handle_call(request, from, state) do
  ...
  {:reply, return_value, new_state}
end
```

- The `handle_call` function *always* returns a new state, and *may* return a value to the caller, thus unblocking it
- Concretely a tuple is returned, either:
  - `{:reply, return_value, new_state}` – returns a new state and replies with the value `return_value` to the caller
  - `{:noreply, new_state}` – just returns a new state
- Note that when handling a new call, it is possible to reply to a previous call by calling

  `GenServer:reply(previous_from,return_value)`

POLITÉCNICA

Handling Non-Blocking Calls

- Non-blocking calls are handled in `handle_cast`:

```
def handle_cast(request, state) do
  ...
  {:noreply, new_state}
end
```

- The caller cannot receive a return value

POLITÉCNICA

Example: A Resource Handler

- A process *pidH* controls some piece of sensitive hardware (e.g., the brakes in a car)
- To interact with the hardware (e.g., brake the car) other processes sends messages to *pidH*
- To prevent bad interactions between different processes interacting with the hardware only one process may communicate with *pidH* at any given time.
- A client (process) must use a simple access protocol before sending messages to *pidH*:

```
Hw.request()          # request access to the hardware,
                      # return when permission given

send(pidH, msg1)      # interact with hardware
...
send(pidH, msgN)

Hw.release()          # notify stopping access to hardware
```

- Our task is to use a GenServer to implement the protocol, or in other words, to implement the *Hw* module.

POLITÉCNICA

The Server State

- Multiple clients may request access to the hardware at the same time, but we should only grant access to a single process at a time
- As `request` and `release` messages will arrive at the server in any order, it has to manage a queue of hardware access requests
- What should the server state be?
- One option: let the state be a list [*Head* | *Rest*] where
  - *Head* is the client (process) currently accessing the resource
  - and *Rest* are the clients (processes) waiting to access the resource

POLITÉCNICA

Handling Client Errors

- What happens if a client fails crashes while accessing the resource? (after request but before release)
- The GenServer will stay “locked” forever, other clients in the queue will never get access
- How to fix?
- We could monitor/link to the client and removing it from queue when it crashes
  - We have to implement the following callback function:

```
handle_info(msg, state) do
  ...
  {:noreply, newstate}
end
```

  which is called when non-call and non-cast messages are received by the *GenServer* process

POLITÉCNICA

Error Handling in the Generic Server component

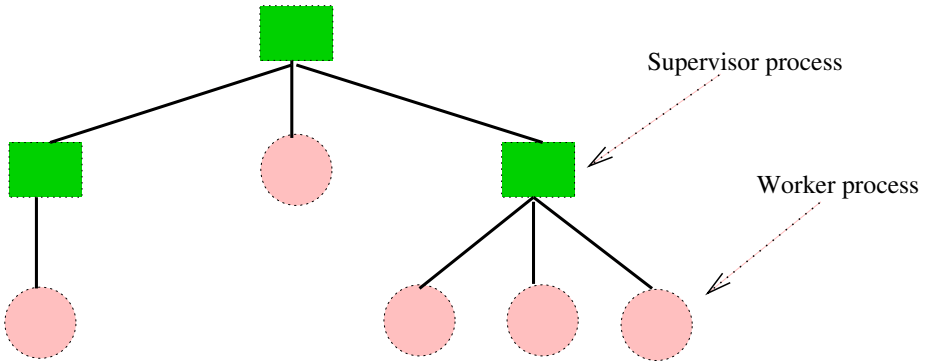
Note some nice properties of error handling in generic servers:

- We only handle errors in one place in the code
- We only handle errors at very controlled points in time (when not processing a request)
- We control error handling – we do not letting error handling control us!

POLITÉCNICA

The Supervisor Component

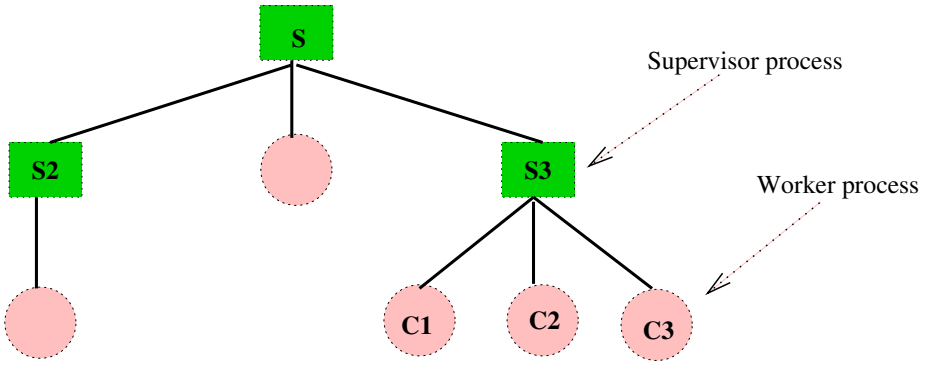
- Elixir Applications are often structured as *supervision trees*, consisting of *supervisor* and *worker* processes



- A supervisor
  - starts child processes in a specified order
  - monitors them – and possibly restarts them if they fail
  - stops them on request
- The actions of the supervisor are described in a declarative fashion (as a text description)
- Note that a child process may itself be a supervisor

POLITÉCNICA

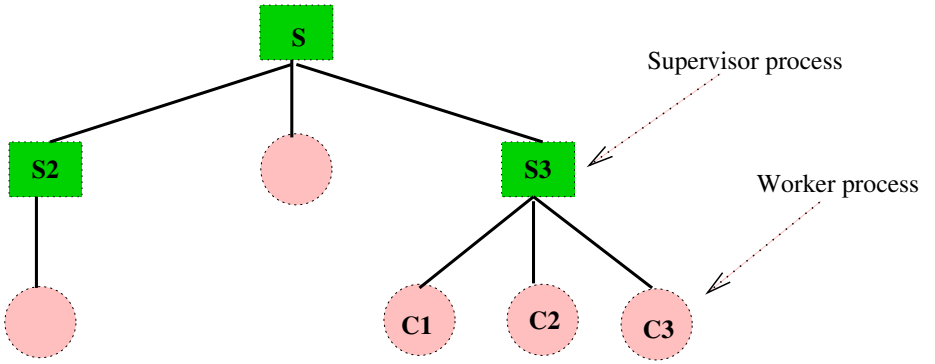
Supervision Dynamics



- We assume that the children has been started in the order C1, C2, C3.
- When a child process C2 dies, its supervisor S3 is notified.
  If the child C2 is declared
  - `:permanent` – the child is always restarted
  - `:temporary` – the child is not restarted (even if it terminated abnormally)
  - `:transient` – the child is restarted only if it terminated abnormally
- If the decision is that C2 should be restarted then different sibling restart strategies can be specified in supervisor S3.
  - `:one_for_one` – only restart the failing child, not its siblings
    - ★ i.e., only restart C2, do not restart C1 and C3
  - `:one_for_all` – restart **all** siblings
    - ★ i.e., terminate first all its siblings C1 and C3, and then restart C1, C2 and C3
  - `:rest_for_one` – terminate all siblings started **after** the failed child
    - ★ i.e., since C3 was started after C2 terminate it, and then restart C2 and C3

POLITÉCNICA

Supervision Dynamics



- One can also control the frequency of restarts, and the maximum number of restarts to attempt – it is no good having a process continuing to restart and crash:
  - `:max_restarts` – the maximum number of restarts permitted in a time frame. If exceeded the supervisor terminates *abnormally* (default 3).
  - `:max_seconds` – the time frame (default 5 seconds).

POLITÉCNICA

Supervisors: under-the-hood

- A child process being supervised by a supervisor should be *linked* to the supervisor
- Read more about supervisors on the web:
  - `https://hexdocs.pm/elixir/1.13/Supervisor.html`
  - and lots of tutorials...

POLITÉCNICA

Example: Worker Jobs

To experiment with supervisors we program a simple executor of jobs

```
Jobs.execute(
  %{
     job1_name: f_1,
     ...
     jobN_name: f_n
  })
```

- A job has a name and a execution specification, i.e., a function with arity zero.
- The *Jobs*.execute function starts each job in a separate process, in parallel, collects the results, and returns a map where the keys are the job names and the values the job results.
- Example:

```
Jobs.execute(
  %{ time: fn () -> Time.to_string(Time.utc_now()) end,
     date: fn () -> Date.to_string(Date.utc_today()) end})

==>

%{ time: "08:41:41.206419", date: "2022-04-22" }
```

POLITÉCNICA

Idea for implementing execute(jobs):

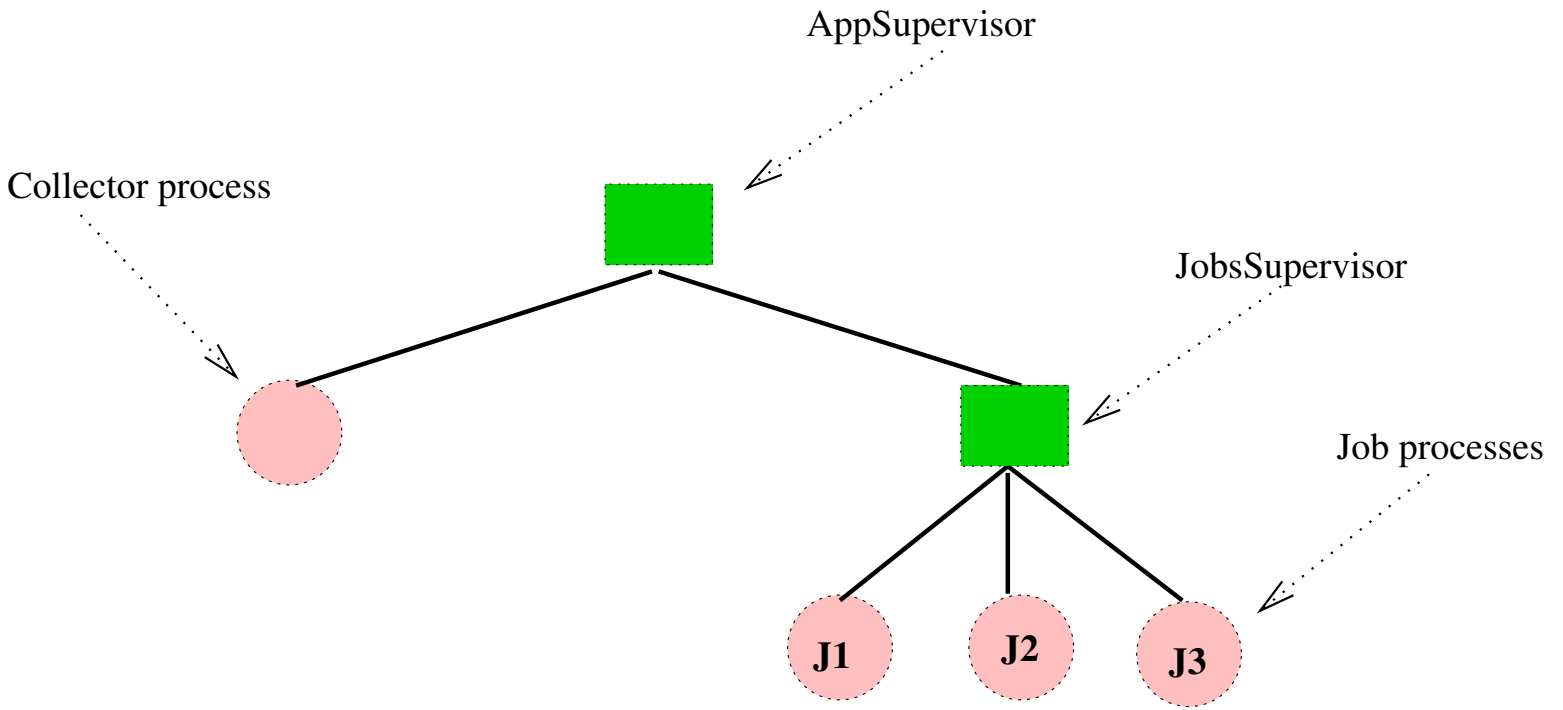
```
def execute(jobs) do
  # Spawn collector process
  myPid = self()
  spawn(fn () -> register(self(),:collector)
                 collector(myPid,map_size(jobs)) end)

  # Spawn job processes
  Enum.each(jobs, fn {job,f} ->
            spawn(fn () -> send(:collector,{job,f.()}) end)
        end)

  # Wait for results from collector process
  receive do x -> x end
end
```

1. When *Jobs*.execute(...) is invoked from the "shell" a "collector process" is started which is responsible for collecting the results from the jobs (time,date)
2. The job processes (time, date) start, compute their results, and sends them to the collector
3. When the collector process has received both results, it sends them to the "shell"

POLITÉCNICA

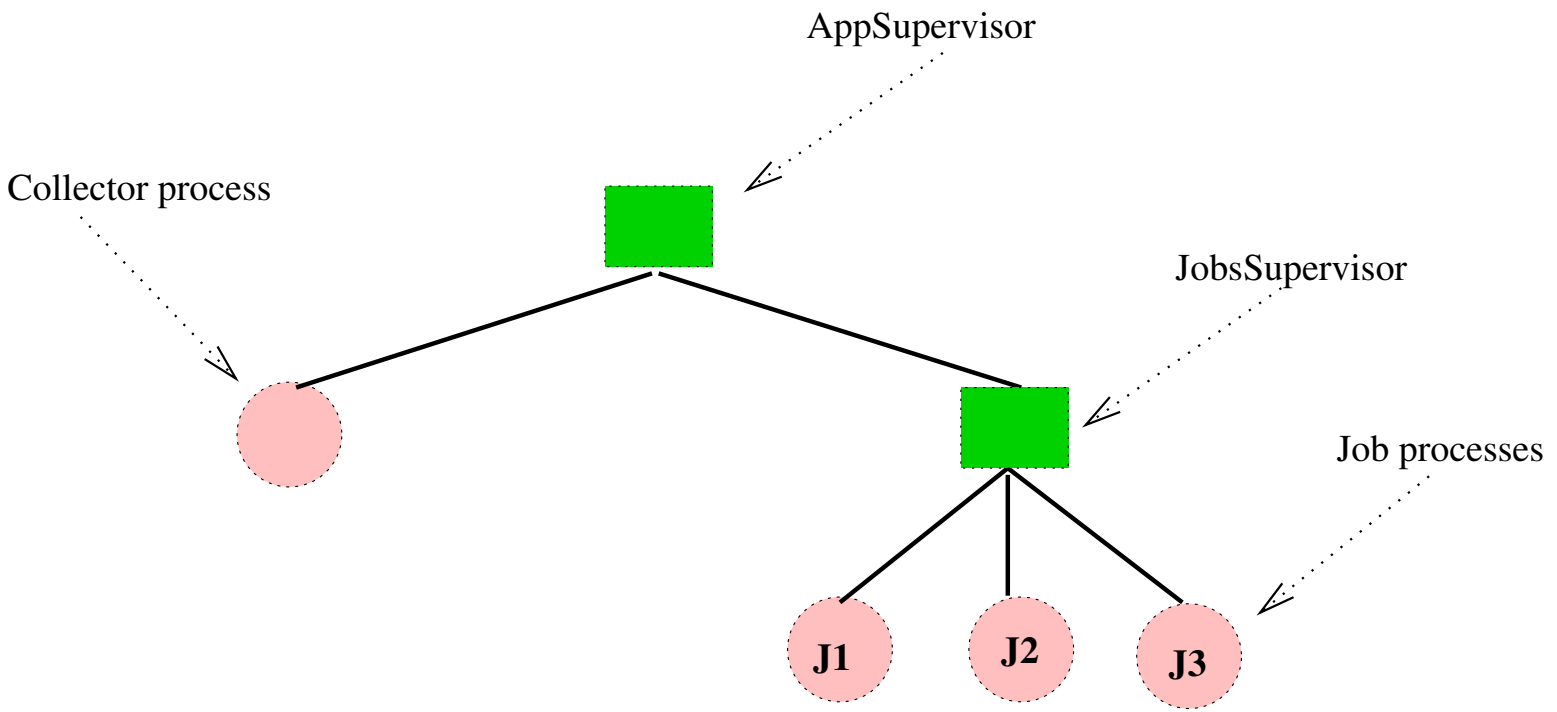
Supervision of the Job executor

- How do we program a *robust* Job executor?
  - One process gets started for each job to execute – `J1`, `J2`, ...
  - Additionally we create one process to accumulate the results – `Collector`
- Concretely we use a supervision tree of two three levels: one supervisor `JobsSupervisor` for the all the job workers, and one supervisor *AppSupervisor* for the `Collector` process **and** for the worker supervisor:



POLITÉCNICA

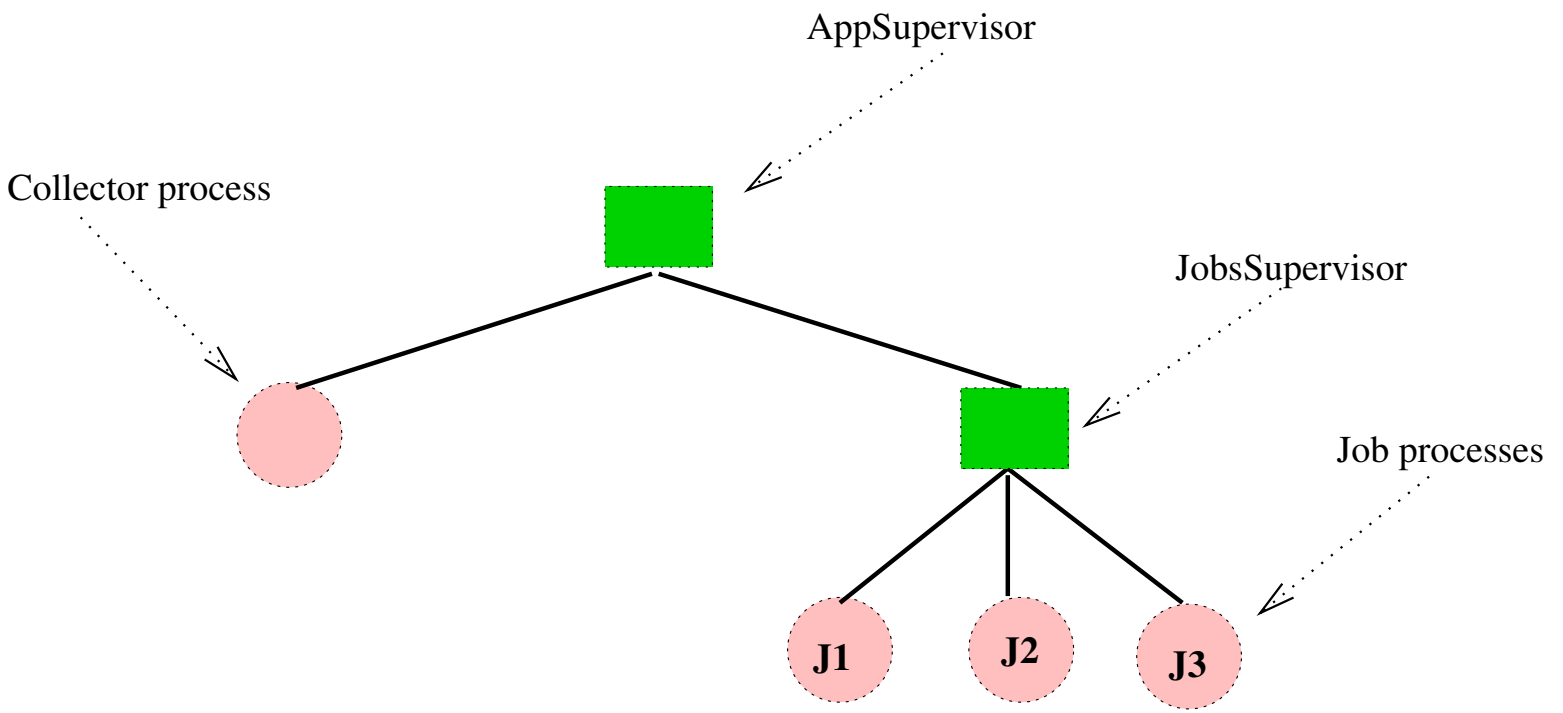
Decision: restart strategy for a Job worker process



- What should we do if a worker process terminates *abnormally*?
  - restart it
- What should we do if a worker process terminates *normally*?
  - nothing
- So a worker is a `:transient` process

POLITÉCNICA

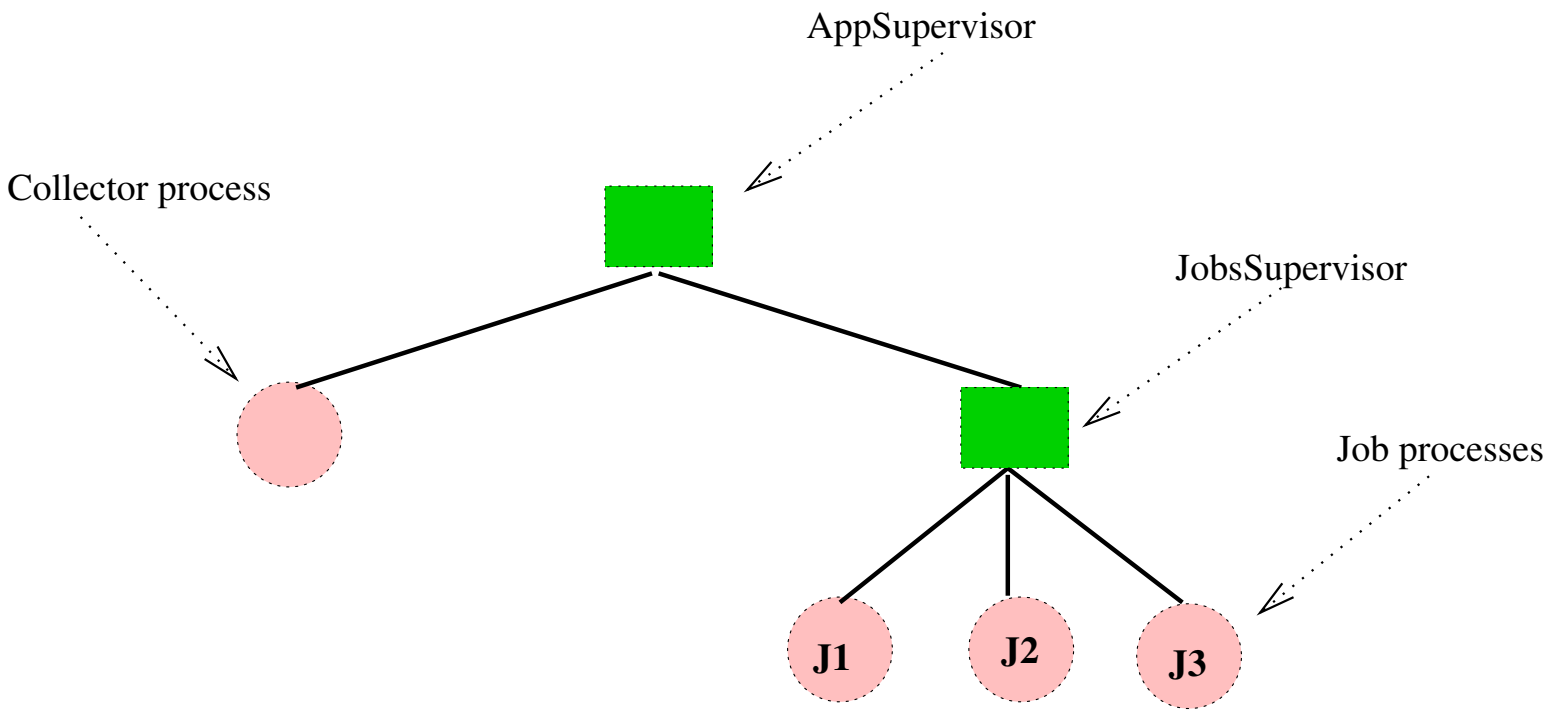
Decision: restart strategy for the `Collector` process



- Similarly the `Collector` process may terminate normally, it should be restarted only when it terminates abnormally
- It is a `:transient` process

POLITÉCNICA

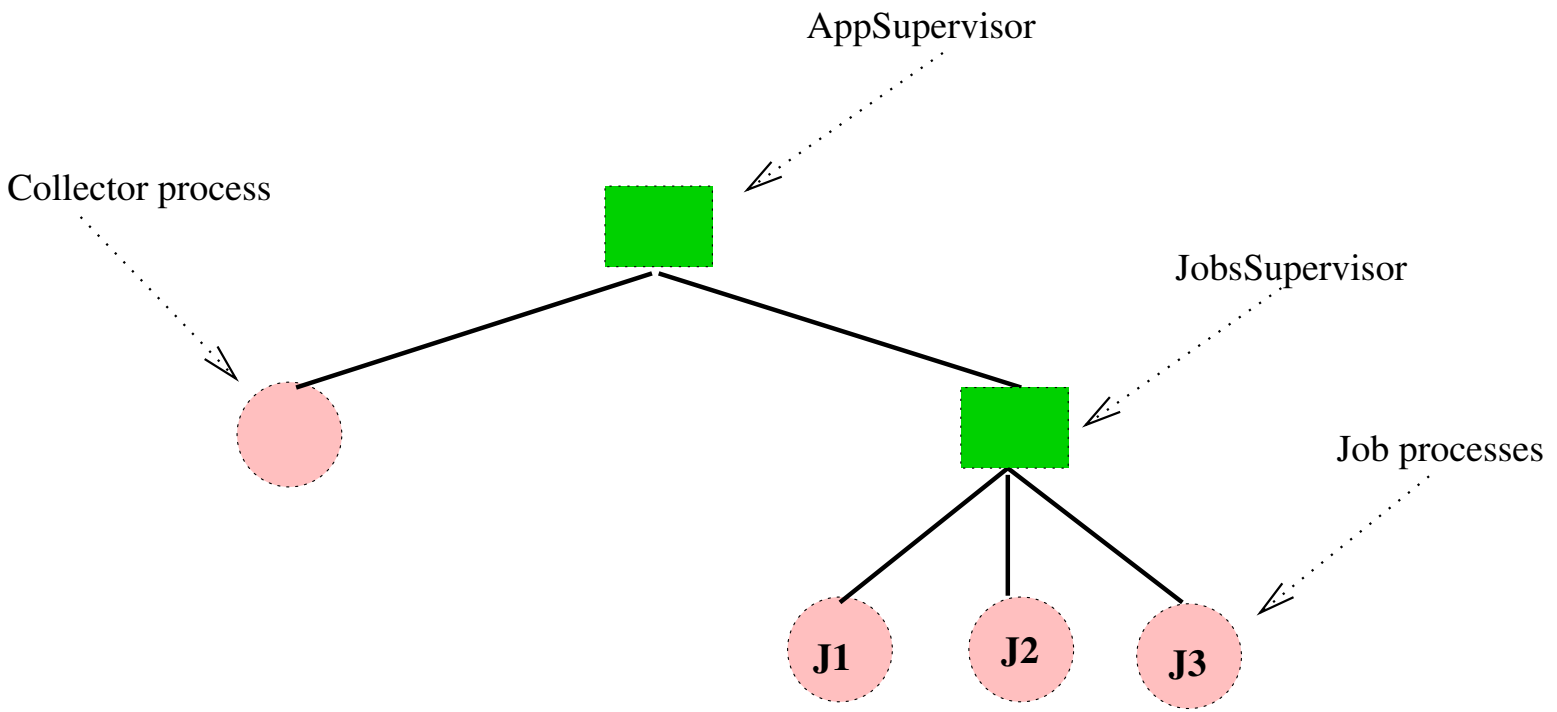
Decision: restart strategy for the *AppSupervisor*



- What should we do if the *JobsSupervisor* process terminates abnormally?
  - restart it **and** also restart the *Collector* process
- What should we do if the *Collector* process terminates abnormally?

POLITÉCNICA

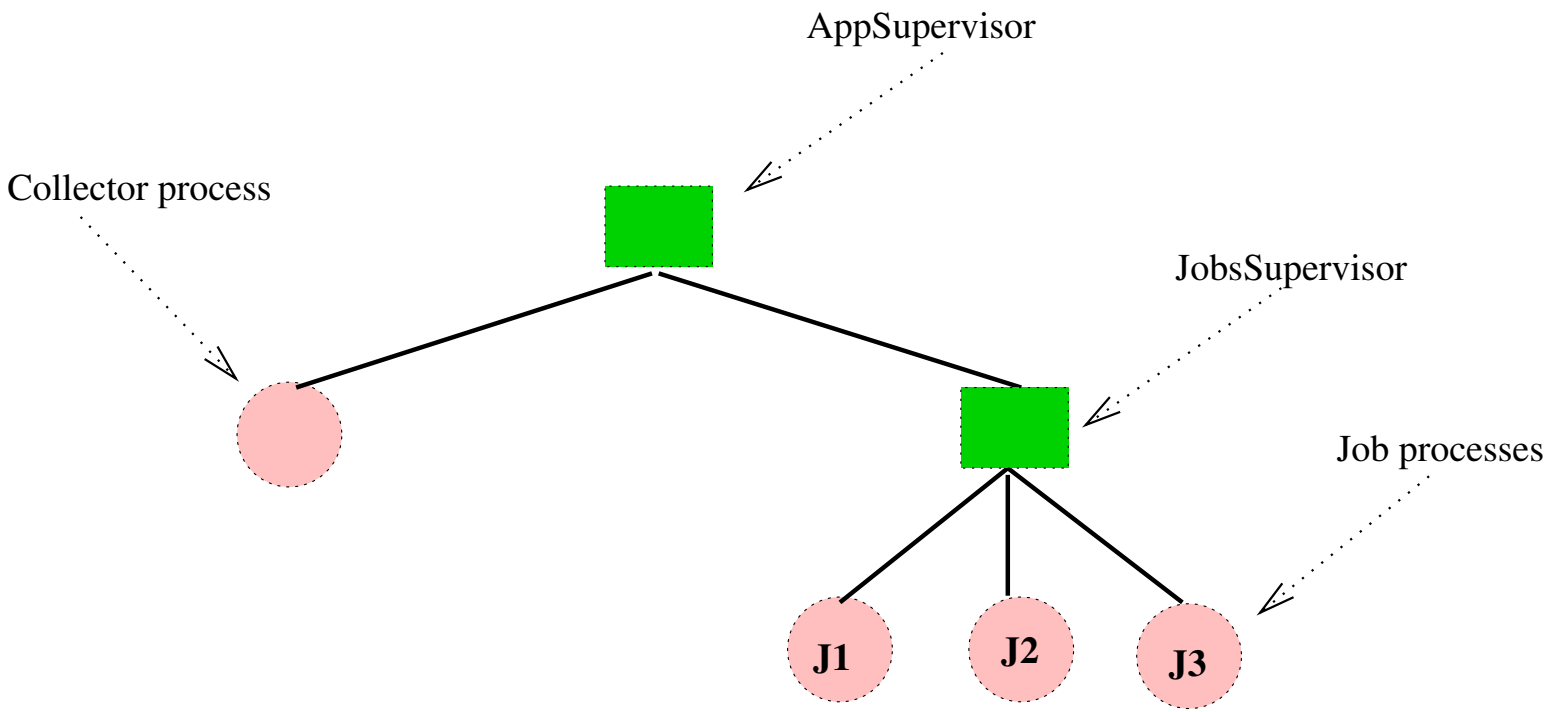
Decision: restart strategy for the *AppSupervisor*



- What should we do if the *JobsSupervisor* process terminates abnormally?
  - restart it **and** also restart the *Collector* process
- What should we do if the *Collector* process terminates abnormally?
  - restart it, **and** stop and restart the *JobsSupervisor* (and all the jobs – happens automatically)

POLITÉCNICA

Decision: restart strategy for the *AppSupervisor*



- What should we do if the *JobsSupervisor* process terminates abnormally?
  - restart it **and** also restart the *Collector* process
- What should we do if the *Collector* process terminates abnormally?
  - restart it, **and** stop and restart the *JobsSupervisor* (and all the jobs – happens automatically)
- So the *AppSupervisor* should use the strategy `one_for_all`

POLITÉCNICA

Starting Supervisors: example

To start the top supervisor process, executing `jobs`, we invoke:

```
Supervisor.start_link(children,
                      strategy: :rest_for_all,
                      name: AppSupervisor)
```

- `strategy` specifies the restart strategy for the children
- `name` specifies a registered name for the supervisor process
- `children` is a list of modules and arguments for the supervised children (collector, jobsSupervisor):

```
children =
  {Collector, [self(), map_size(jobs)]},
  {JobsSupervisor, [jobs]}
```

  - For each child, the supervisor
    1. retrieves its child specification,
    2. and then starts the child (see next slide)
  - `self()` is passed as a parameter to the collector so that it knows where to send the result
  - `maps_size(jobs)` is passed as a parameter so that it knows how many results to expect

POLITÉCNICA

Collector code

- To create the Collector from the recipe {*Collector*,arg}, the supervisor first retrieves a "child" specification by calling *Collector*.child_spec(arg):

```
def child_spec([parent_pid,num_jobs]) do
  %{
    id: Collector,       # identity
    restart: :transient # restart only if abnormal exit,
    start: {Collector, :start_link, [parent_pid,num_jobs]}
  }
end
```

- The `start` key specifies how to start the child (Collector), a tuple (module, function, arguments). The function should start and link to a new process, and return a tuple `{:ok, pid}` where `pid` is the pid of the new process:

```
def start_link(parent_pid, num_jobs) do
  {:ok,
   spawn_link(fn () ->
    Process.register(self(),:collector)
    # call main service loop
    collector(parent_pid, %{}, num_jobs)
  end)}
end
```

POLITÉCNICA

Supervisors: advantages

- A nice way for specifying the termination “dynamics” of an application
  - What should the application do when one of its components fails?
  - And what should it do when one of its components terminates normally?
- Permits starting and stopping the entire application with one just command
  - All the processes in the application are started when the topmost supervisor is started
  - All the processes in the application are stopped when the topmost supervisor is stopped

POLITÉCNICA