About Me

PANACHE

- Wiebe-Marten Wijnja
- @WiebeMarten / Qqwy
- Creative Developer, Open Sourcerer, Dreamer
- elixirforum.com Moderator

- panache-it.com
- Consulting & Placement
- Ruby, PHP, Elixir, ...
- “Panache” *Bold, In a league of its own*

A Winner is You:

A Practical introduction to Concurrency in Elixir

What this talk is about

- What type of language is Elixir
- Why is concurrency important?
- How is concurrency different in Elixir?
- **Practical Example:**
- How do I make a concurrent application in Elixir?
- How do I make it fault-tolerant, scalable?

Assumed knowledge: Basic understanding of programming.

Tick Tock...

- Lot of ground to cover for 45 minutes.
- Both general overview and pratical example.
- **Not** going into all the syntax.
  - (The docs are amazing!)
- Explanation + Demonstration

What kind of language is  elixir ?

- **Concurrent** ← *Topic of this talk*
- Dynamically typed
- Immutable + Functional
- Built on top of Erlang: 29+ years of Robustness
- Flexibility, Macros
- Explicit over Implicit
  - “Instead of Magic (cool stuff happens but you don't know how or why)**,**
    Elixir gives you Power (cool stuff happens, and you know **exactly** how and why)”

Why is concurrency important?

Concurrent: “Happening at the same time”

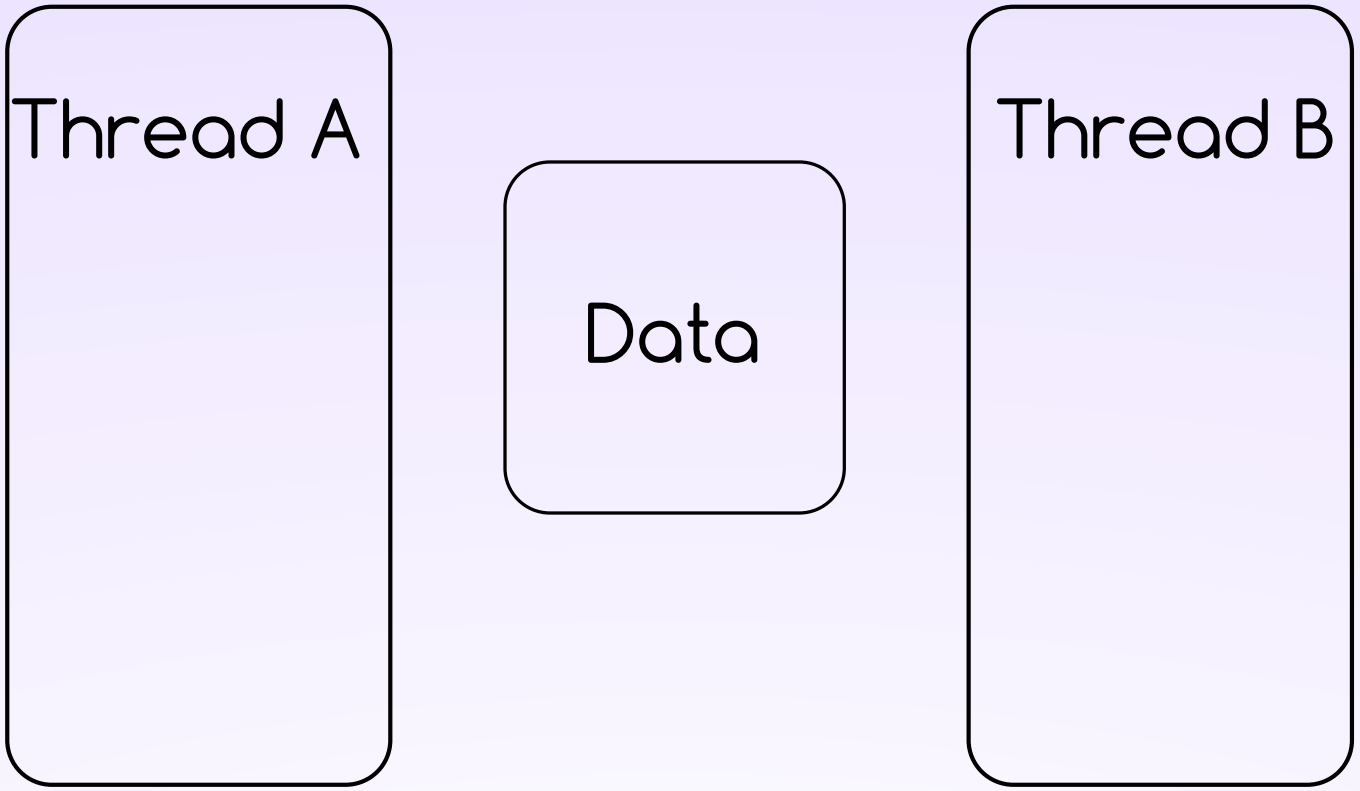
- **Speed**
  - Multicore Computers
  - While waiting for something, do something else as well

Why is concurrency hard?

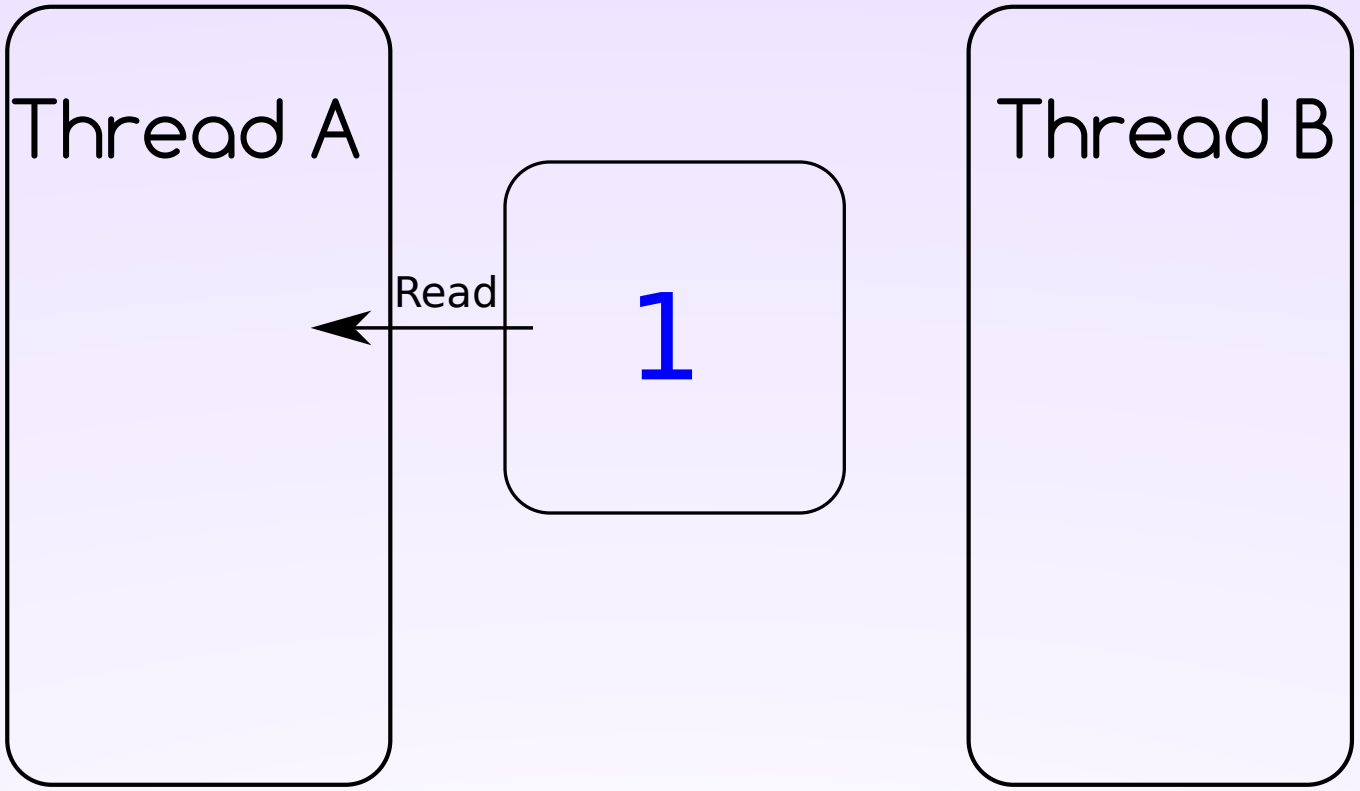
- Multiple things happen at the same time
- Execution order cannot be known

  → Hard to reason about

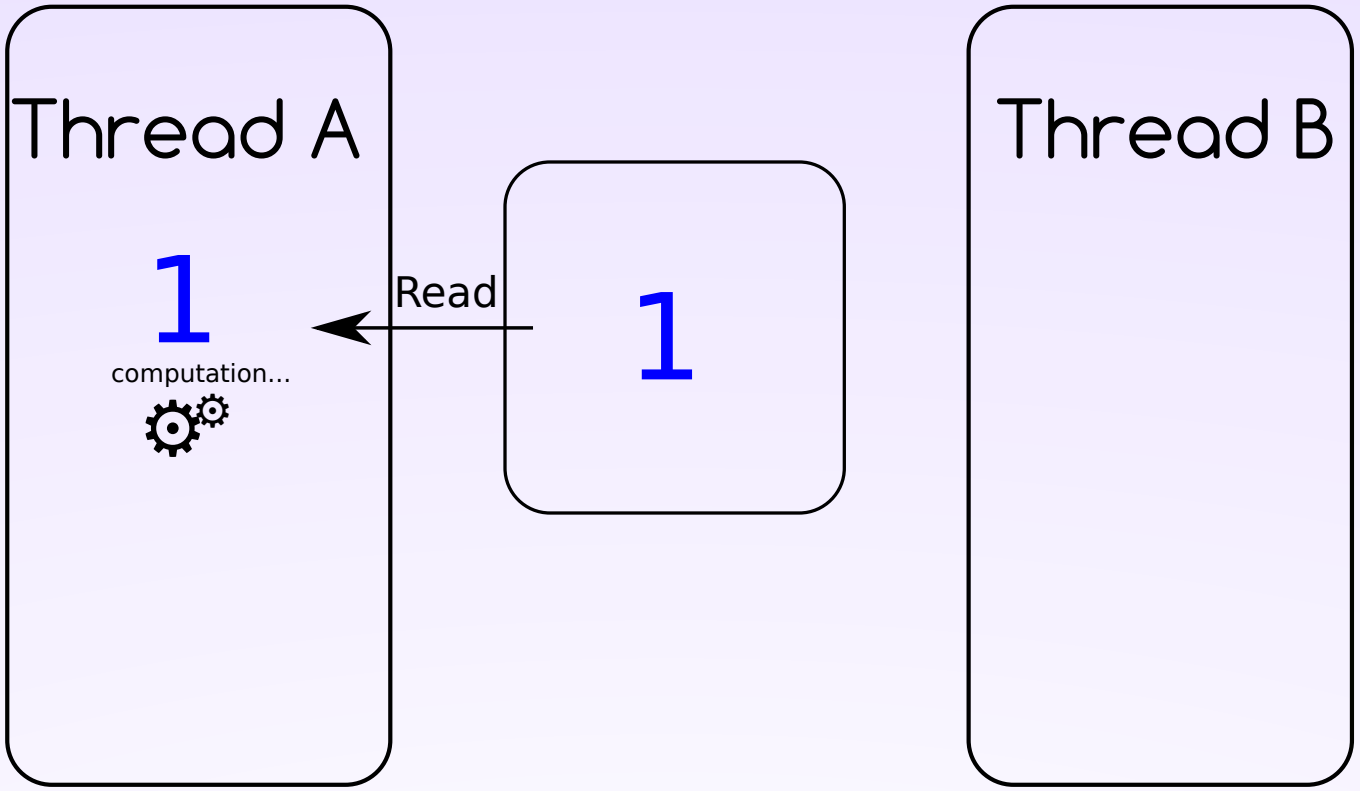
Why is concurrency hard?



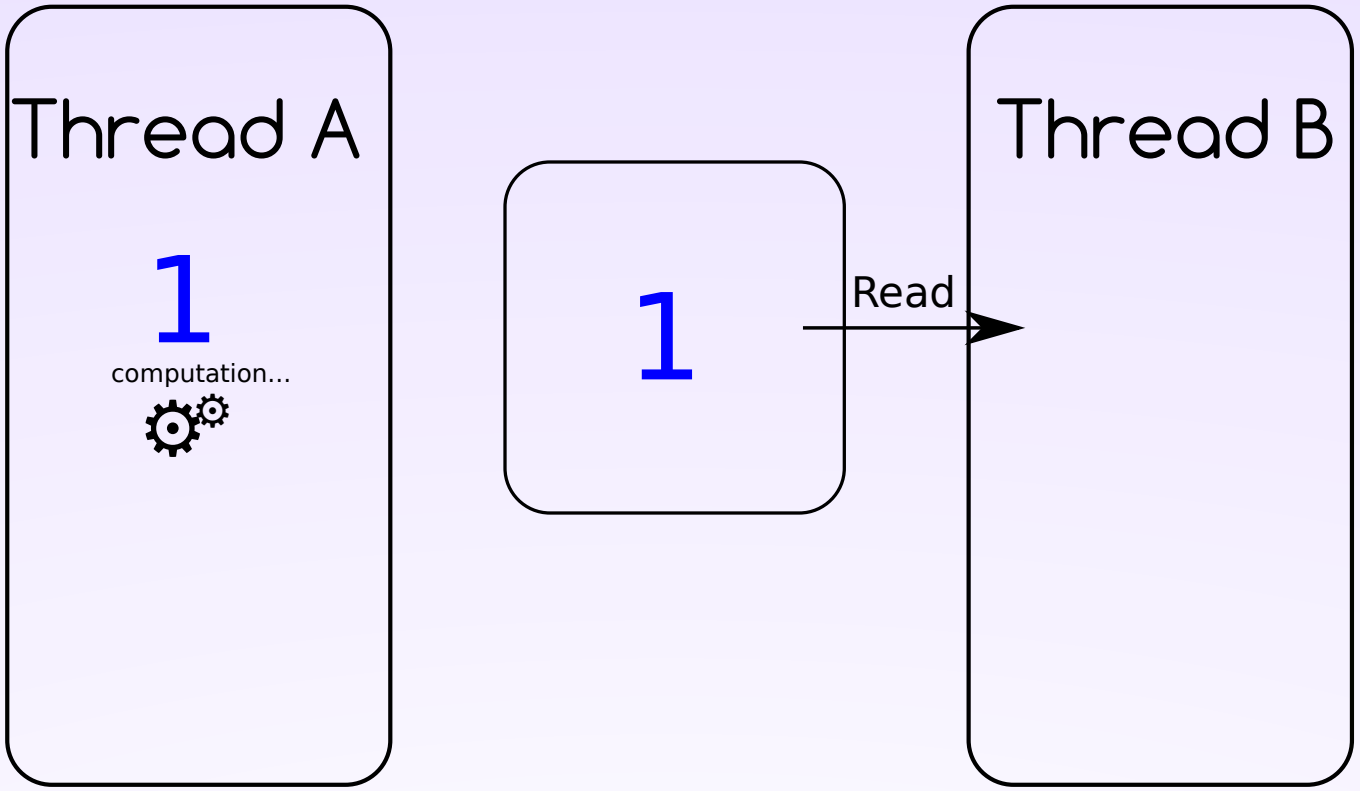
Why is concurrency hard?



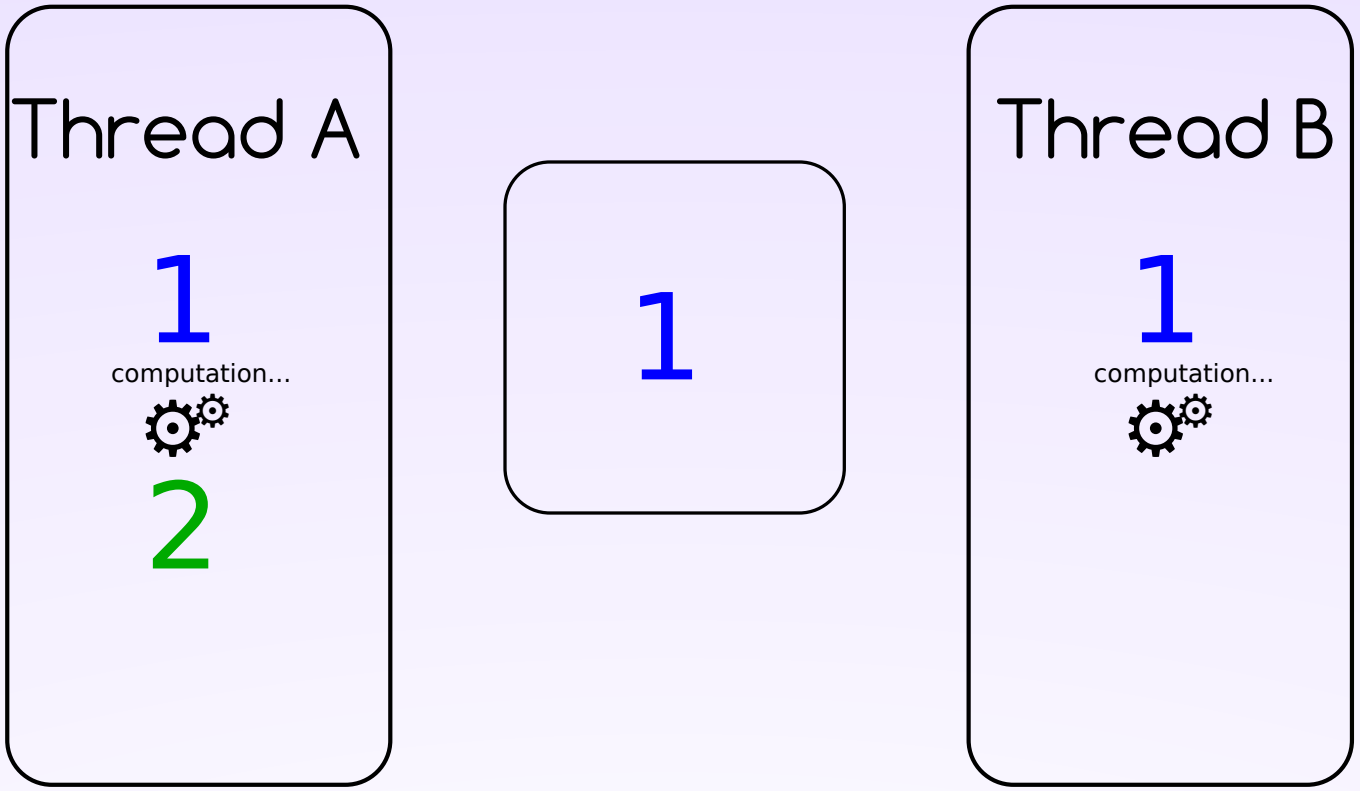
Why is concurrency hard?



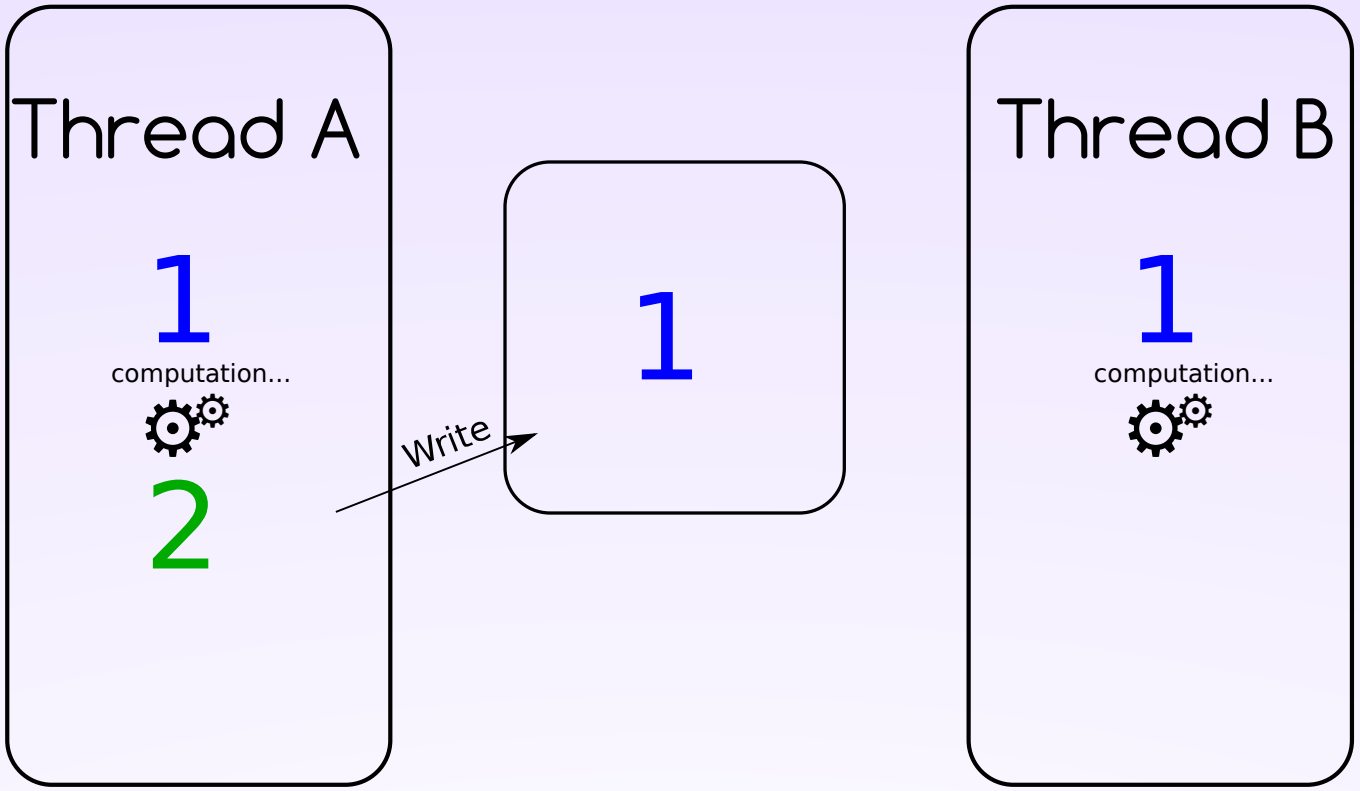
Why is concurrency hard?



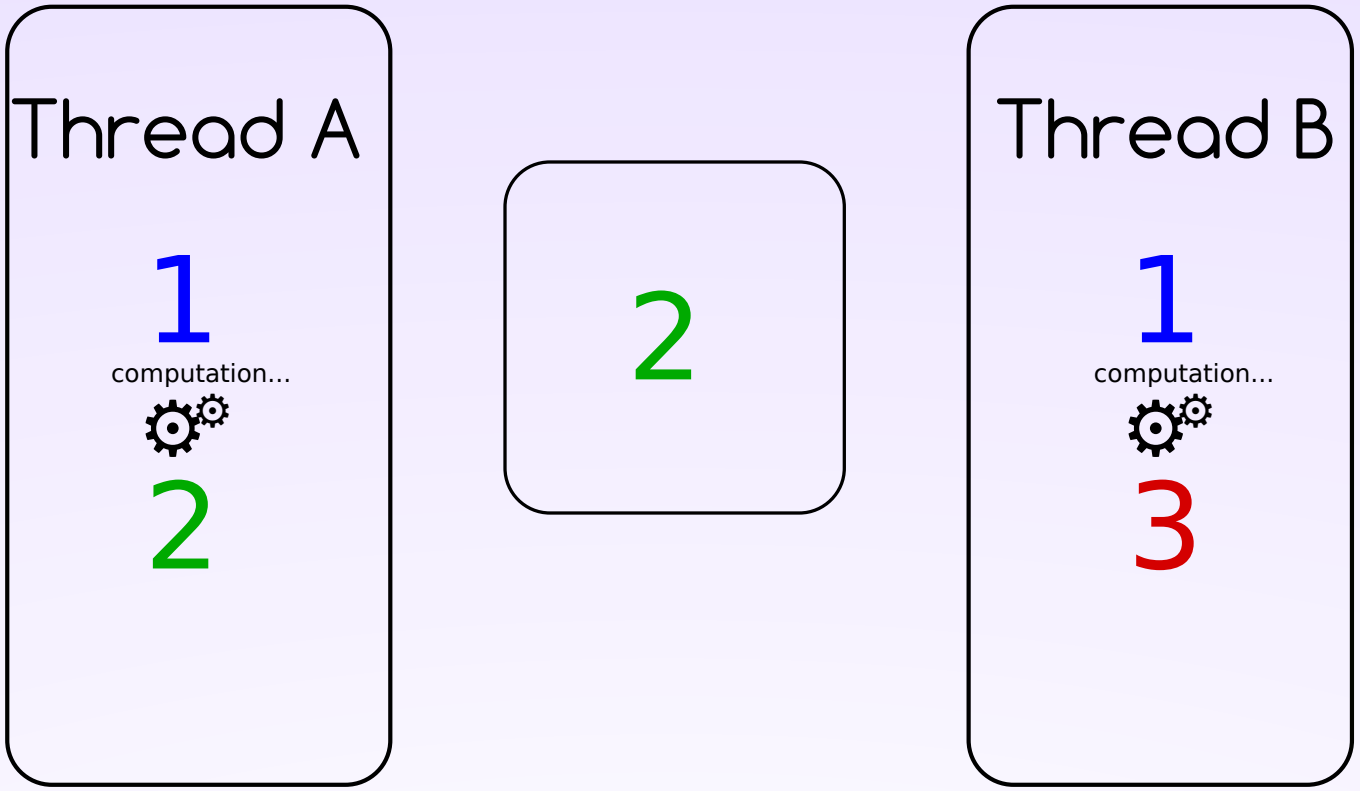
Why is concurrency hard?



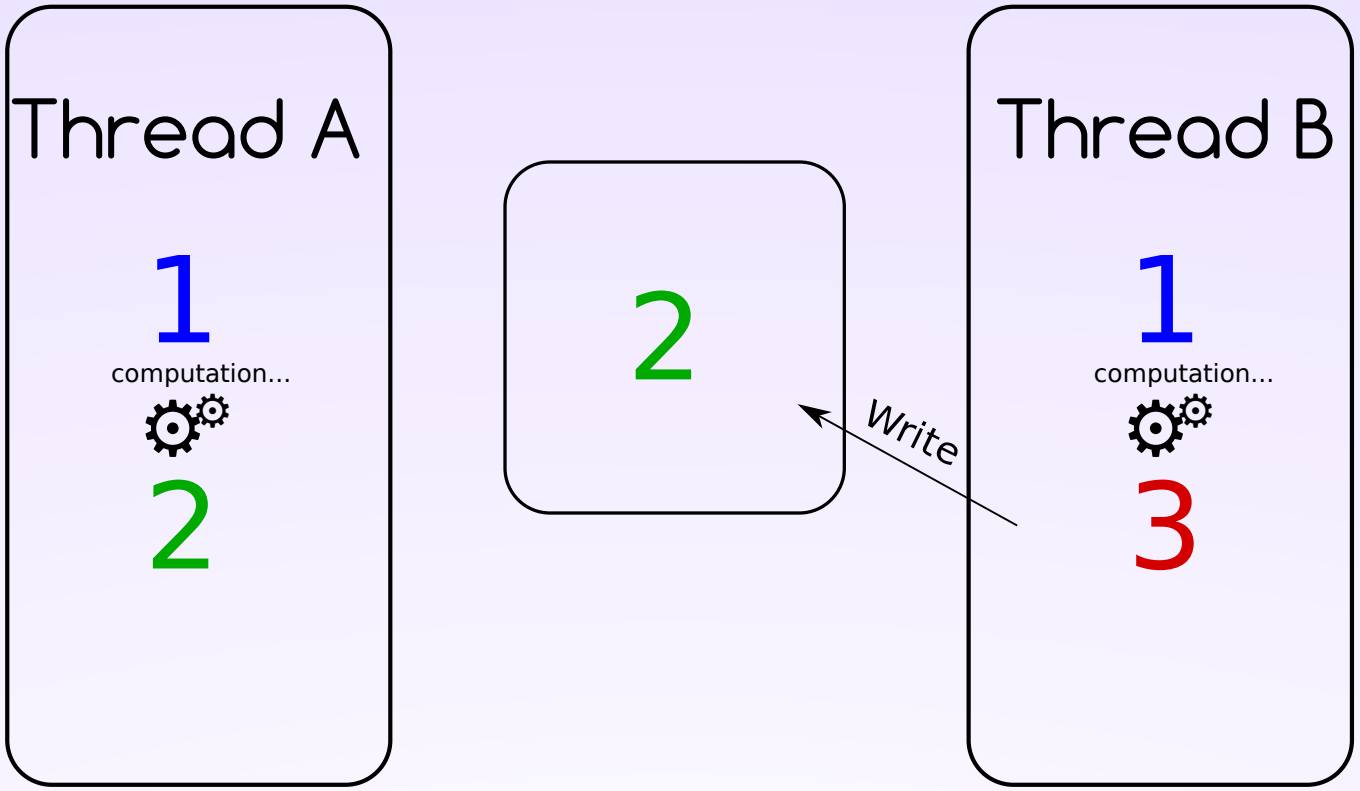
Why is concurrency hard?



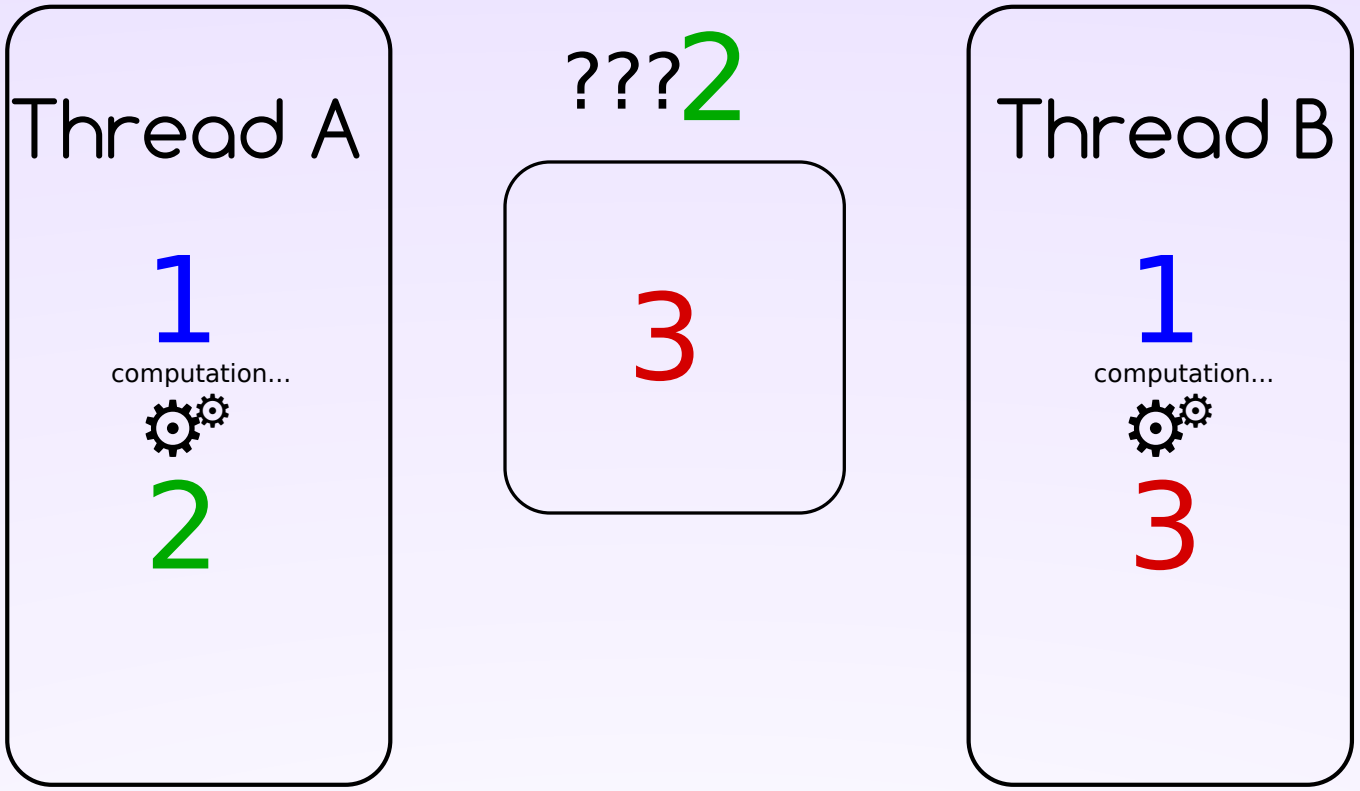
Why is concurrency hard?



Why is concurrency hard?

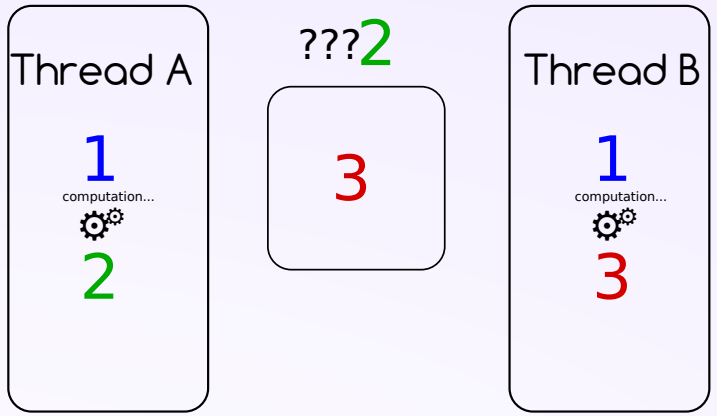


Why is concurrency hard?



Why is concurrency hard?

- Race Conditions
- Locks, Mutexes, etc. are only band-aids
- Very hard to reason about



The only real solution:

Don't share memory between threads!

The Actor-Model

1) A Process is a lightweight VM-thread with **isolated memory**.

2) The only way for processes to exchange information, is by **message passing**.

3) A process can spawn new processes, whenever necessary.

This is enforced *by the virtual machine!*

Memory Safety! Speed! No Stop-The-World Garbage Collection!

The Actor-Model

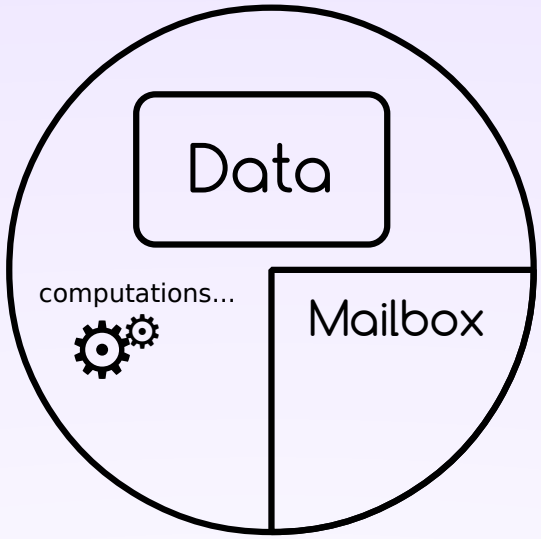
- **Concurrency:** Possible to *truly* run code at the same time.
- **Distribution:** Possible to distribute to other computers. (message-passing works exactly the same)
- **Fault-Tolerance:** When one part breaks, the rest of the system is unaffected.

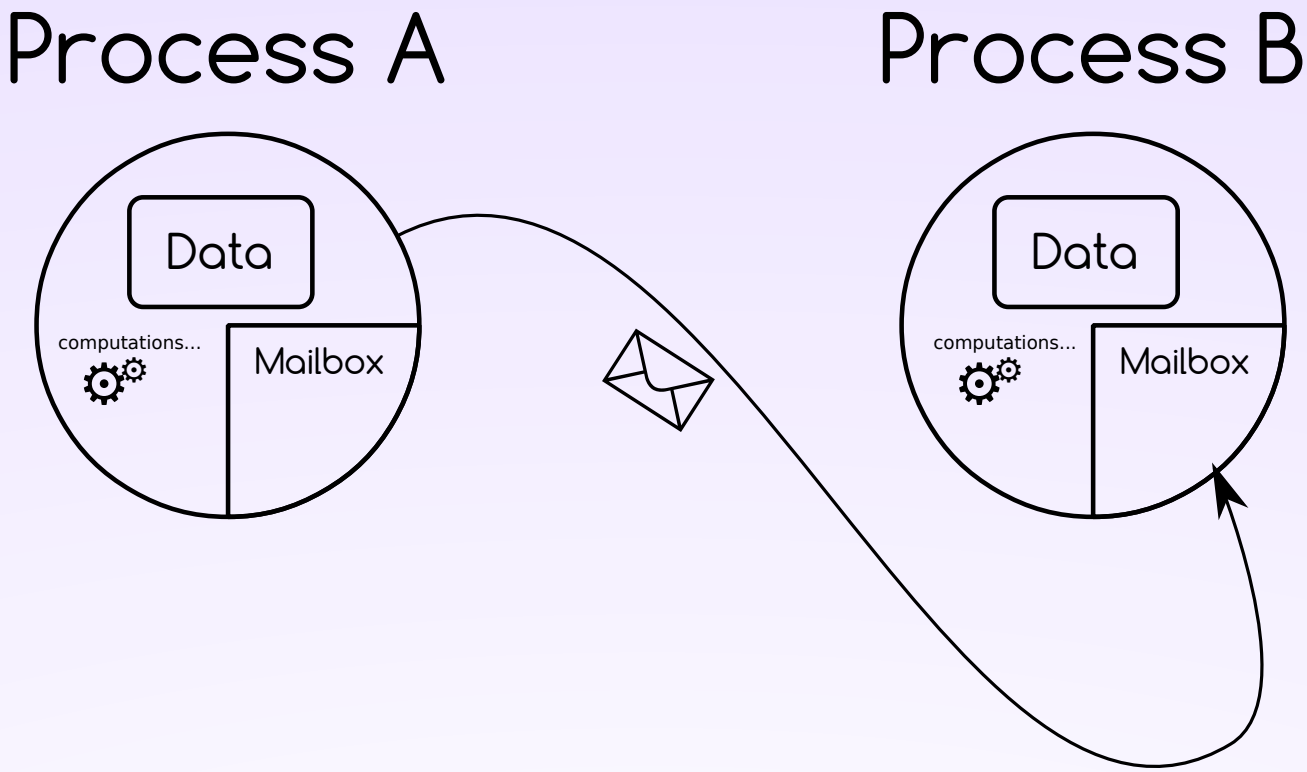
The Actor-Model

- Programming for Ant Colonies

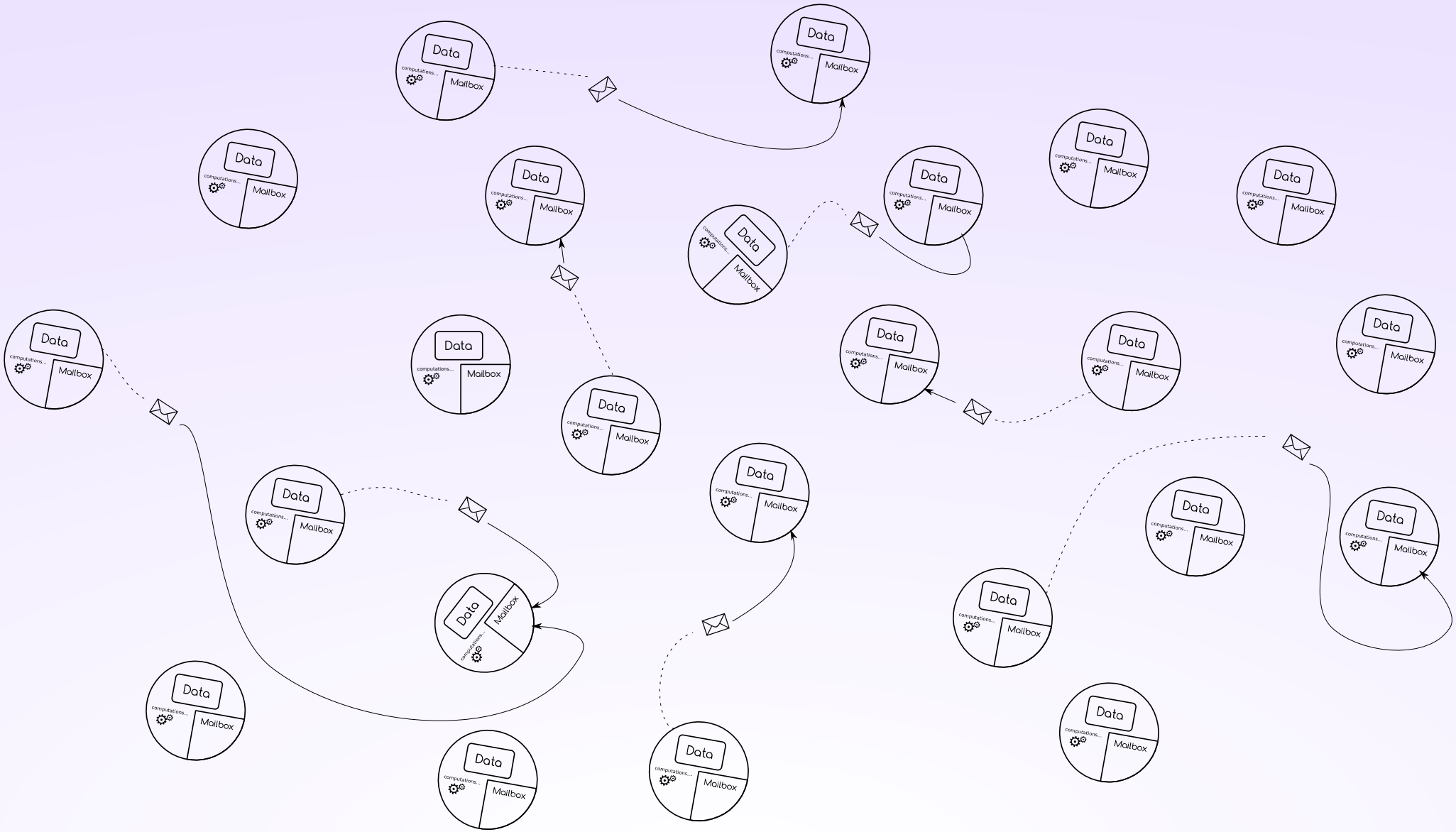
Meet the Process

A Process



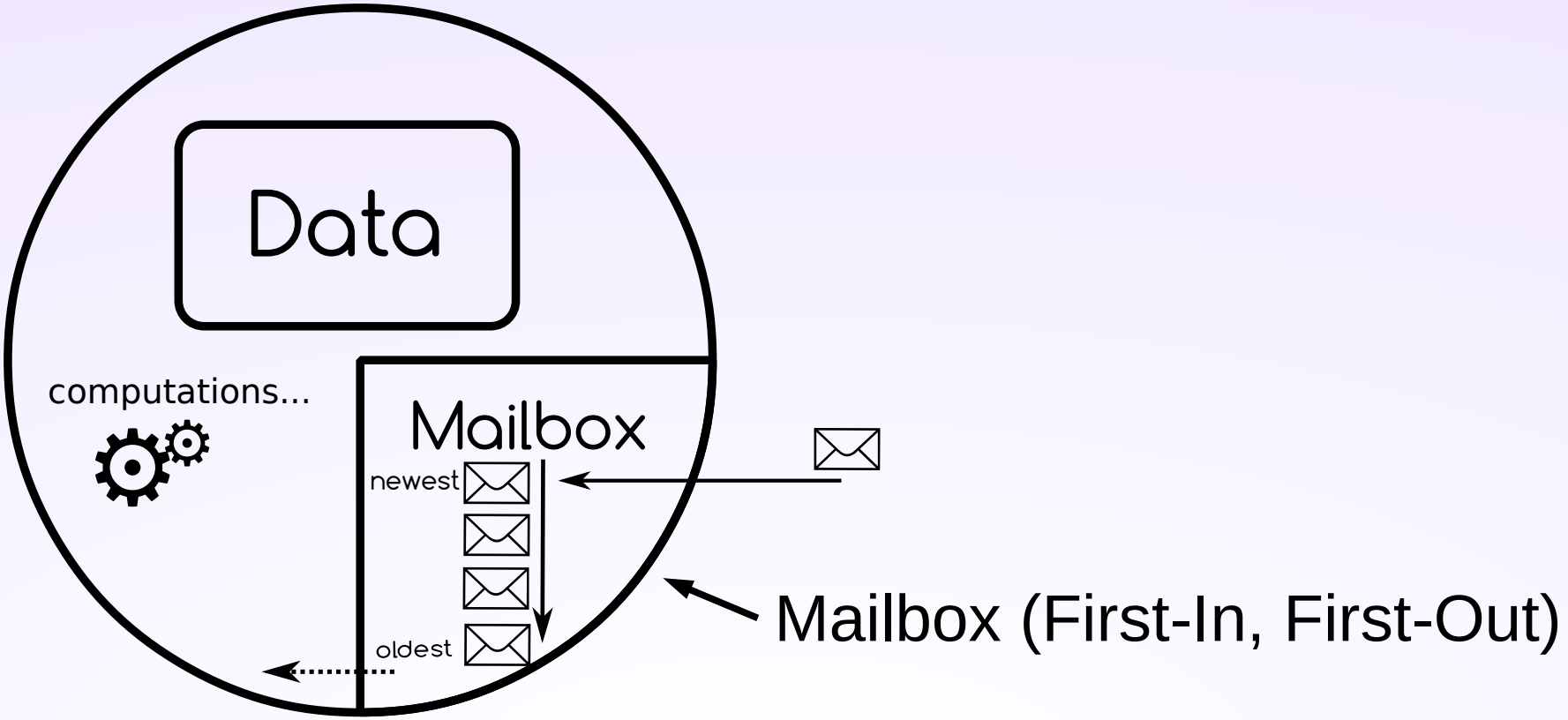

Process A
Process B
Data
computations...
Mailbox
Data
computations...
Mailbox

An Elixir Application:

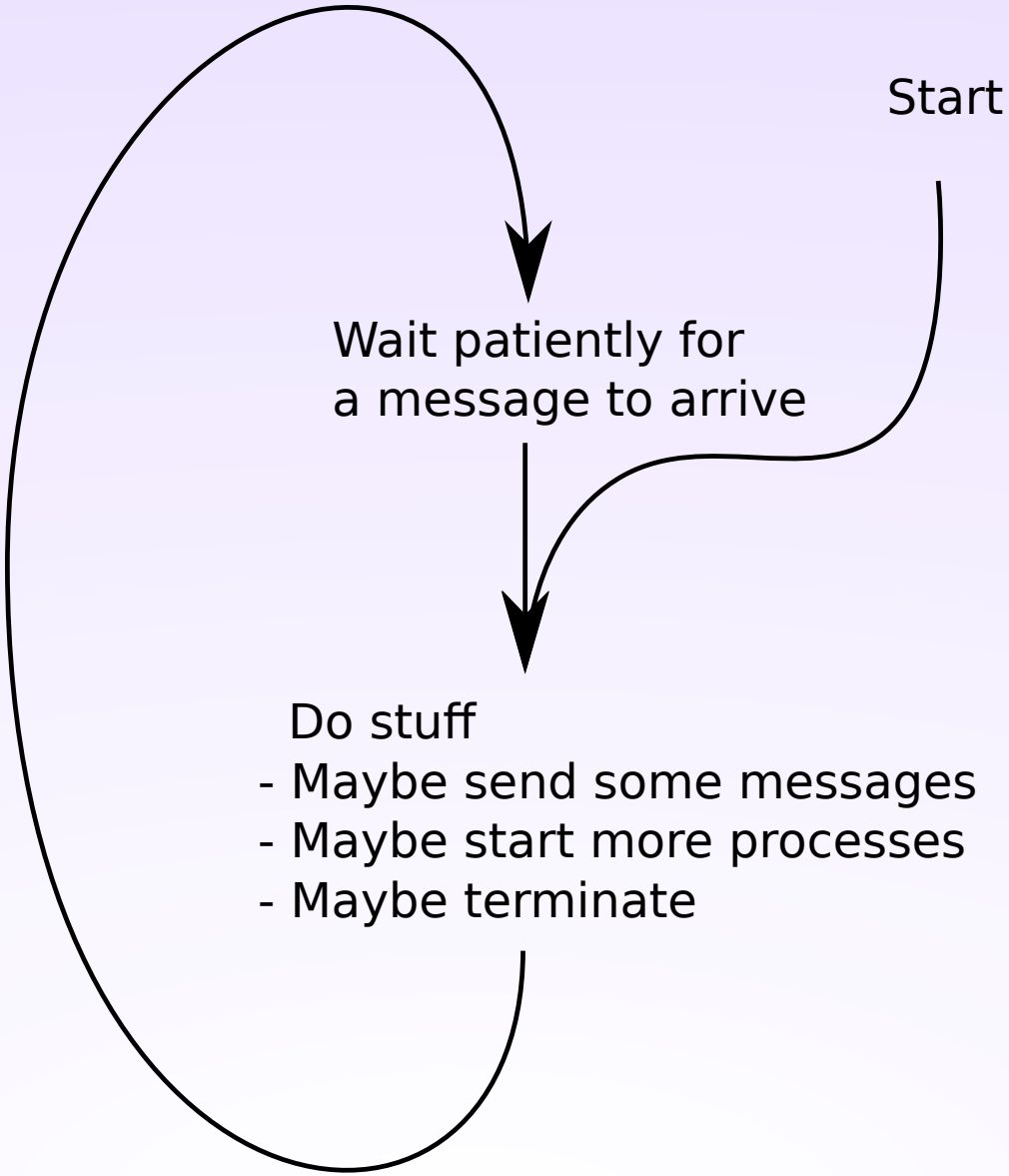


Anatomy of a Process

<0.124.0> ← Process ID (PID)



Life of a Process



Basic building blocks

- `self` → Get the current Process' PID.
- `spawn` → Spawn a new process that executes the given code.
- `send(pid, message)` → Sends a message to a process.
- `receive do` → Waits for the next message (and optionally continues after a timeout)

Very Basic Example

```elixir
other_pid = spawn(fn ->
  receive do
    {sender, contents} ->
      send(sender, "You sent me: #{inspect(contents)}")
  end
end)

send(other_pid, {self, "Hello, world!"})

receive do
  message -> IO.puts "Main Process received: #{inspect(message)}"
end
```

OTP

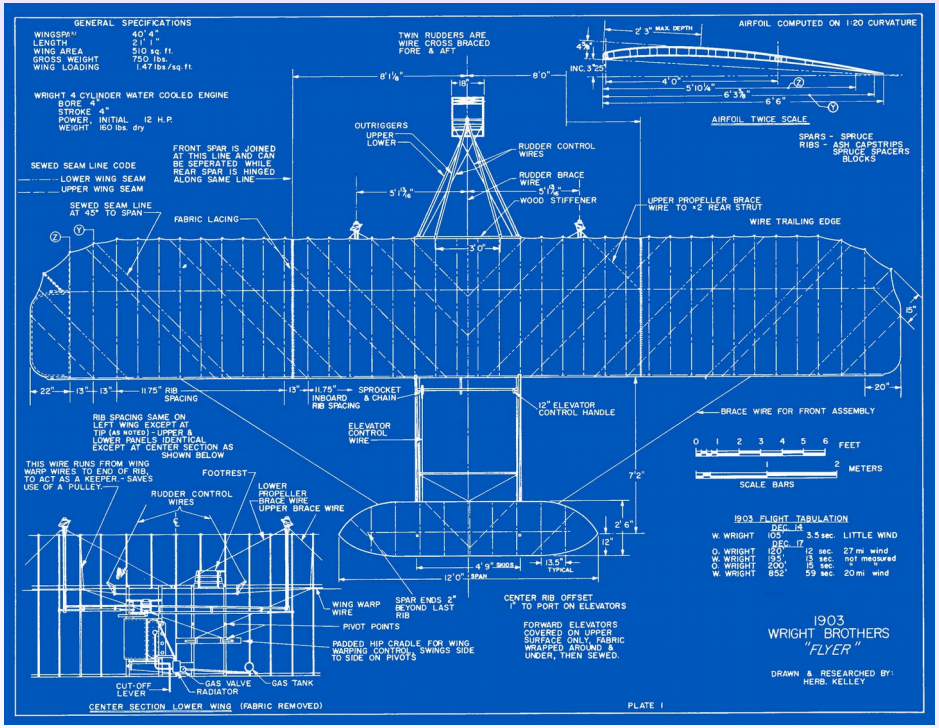
- "Open Telecom Platform"

  Telecommunication: *"The science and technology of the communication of messages over a distance using electric, electronic or electromagnetic impulses."*

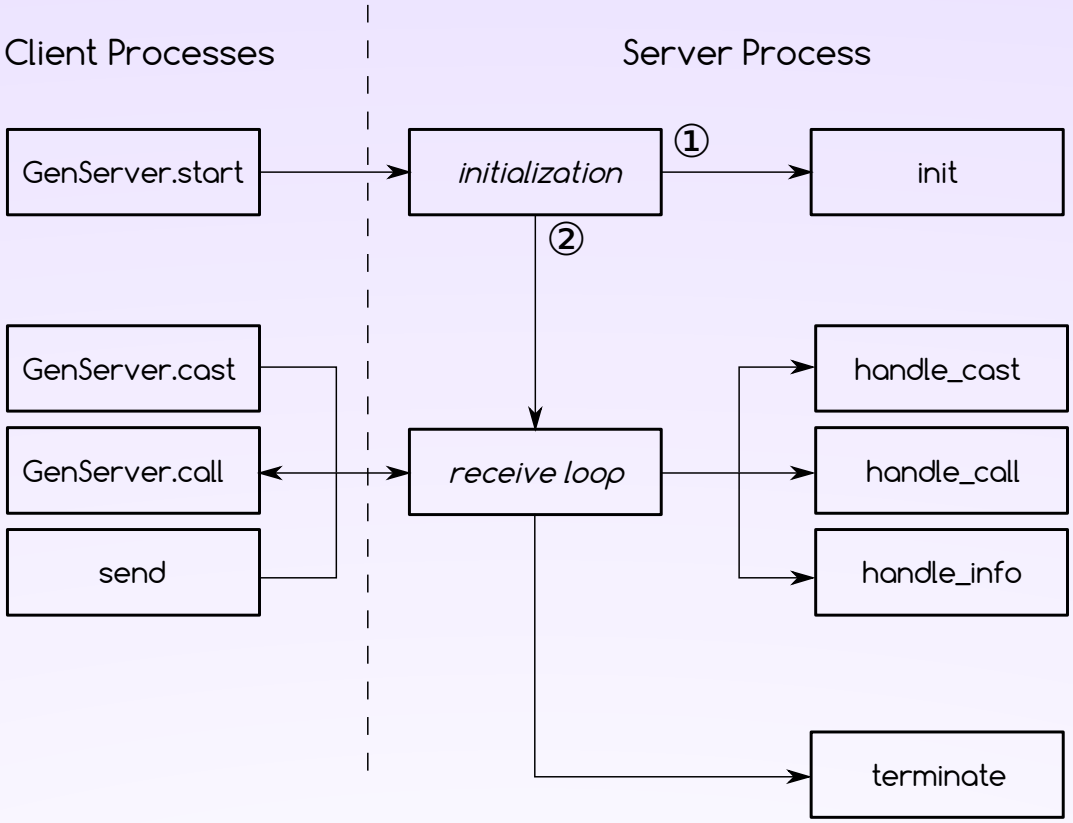
- Tools to make working with multiple processes easier:
  - Less boilerplate.
  - **Generic** blueprints for often-used kinds of processes.
  - Adds **synchronious** message passing.
  - **Supervision**
  - **Name registration** for processes.
  - **Debugging**
  - *More advanced features:*
    - *Hot code reloading*
    - *Distributed failover/takeover*

Generic Blueprints

- **Agent**: Simple wrapper around state.
- **Task**: Single-use concurrent computations.
- **GenServer**: Basic process behaviour blueprint that supports:
  - Initialization
  - Both sync and async message passing
  - Cleanup
  - Both *Agent* and *Task* are built on top of this.
- Some others that are used less frequently (GenEvent, gen_fsm)

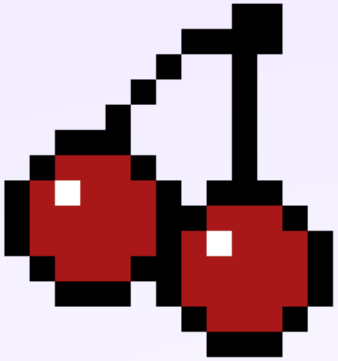


GenServer Life Cycle



Practical Example: High Scores Table

- Easy to understand
- Not too trivial
- Not a TODO-list



Practical Example: High Scores Table

- Save the top X player's scores.
- For each score, save:
  - player name
  - score
  - time
- Return ordered top X when requested.

`HighScoresTable` GenServer

- Internal state:
  - list of scores (ordered),
  - Max. list length
- `init` → Store how many players we should keep track of.
- `add_score` → adds score to list if higher than lowest.
  - Returns `{:congrats, position}` if the player made it in the high scores
  - Returns `:unfortunate` if the player did not.
- `current_scores` → Returns the current list of scores.

~Live Demonstration~

```elixir
defmodule HighScoresTable do
  use GenServer
  @moduledoc "A process that stores a table of high
scores."

  @doc "A table is list of scores and its max. length"
  defstruct scores: [], max_players: nil

  defmodule Score do
    @doc "A single score"
    defstruct name: "", score: 0, time: nil
  end


  # Outward API

  @doc "Starts the process, internally calls `init`."
  def start_link(max_players) do
    GenServer.start_link(__MODULE__, max_players)
  end

  @doc "Adds a new `score` for `name` to the table."
  def add_score(pid, name, score) do
    time = :erlang.time |> Time.from_erl!
    score = %Score{name: name, score: score, time:
time}
    GenServer.call(pid, {:add_score, score})
  end

  @doc "Lists current scores"
  def current_scores(pid) do
    GenServer.call(pid, :current_scores)
  end

 # internal GenServer callbacks

  @doc "Initializes the GenServer"
  def init(max_players) do
    initial_state = %HighScoresTable{max_players:
max_players}
    {:ok, initial_state}
  end

  @doc "Adds a score to the table, updating the internal
state."
  def handle_call({:add_score, score = %Score{}}, _from,
state) do
    {response, new_scores} =
maybe_add_score(state.scores, state.max_players, score)
    new_state = %HighScoresTable{state | scores:
new_scores}
    {:reply, response, new_state}
  end

  @doc "Lists the current scores"
  def handle_call(:current_scores, _from, state) do
    {:reply, state.scores, state}
  end

  defp maybe_add_score(scores, max_players, new_score) do
    split_scores = Enum.split_while(scores, fn score ->
score.score >= new_score.score end)
    case split_scores do
      {all, []} when length(all) >= max_players ->
        {:unfortunate, scores}
      {higher, lower} ->
        new_scores = (higher ++ [new_score] ++ lower) |>
Enum.take(max_players)
        {{:congrats, length(higher)+1}, new_scores}
    end
  end
end
```

It works!

But...

Hmm…

How do we make this:

- Fault-tolerant?
- Scale?

- Crash Handling / Supervision!

Hmm...

Scaling

- Just add More Processes

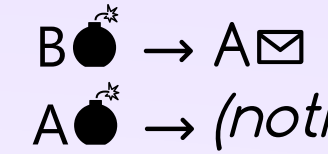
Fault Tolerancy

- **Keep track** of other processes
- **Restart things** when they fail

Keeping Track: Links, Monitors

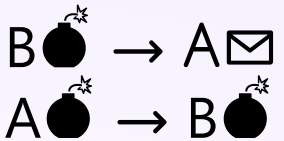
Crash Handling comes in two *(and a half)* kinds:

A Monitors B



A Linked to B

A Linked to B; A trapping exits



Restart when it fails: Supervision

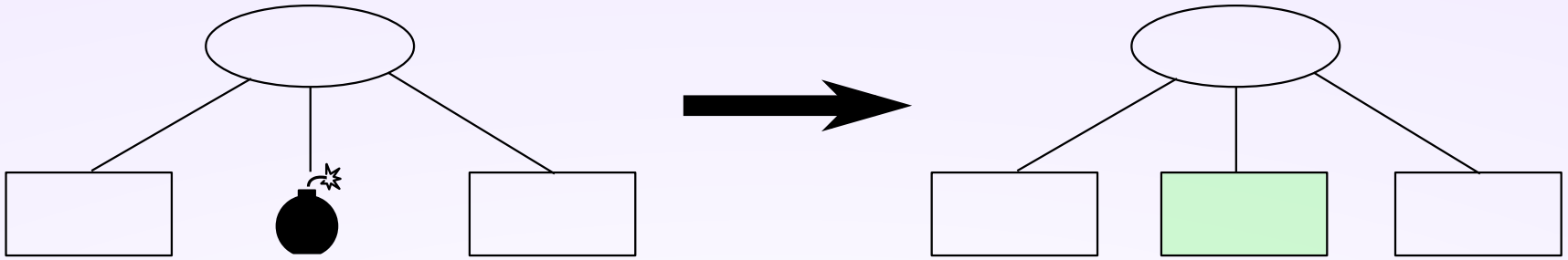
- **Workers** do work
- **Supervisors** supervise
  - Start workers
  - Restart workers *with known state* in case of a failure.
  - When too many failing workers in short time, terminate themselves.
- *Customizeable!*
  - **Supervision strategies**: “What workers should be restarted when one fails”
  - How many failing workers in X seconds are considered 'too many'?

Supervision Strategies

Supervision Strategies:

One For One

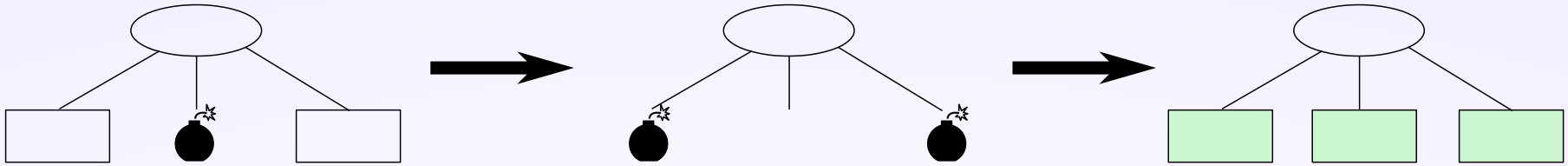
- When a worker dies, it is restarted.



Supervision Strategies:

One For All

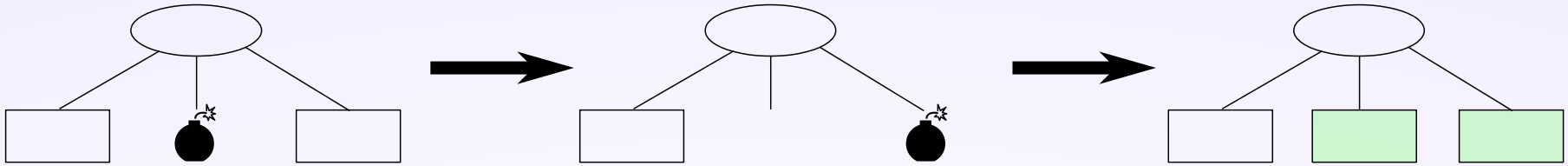
- When a worker dies, *all* of my workers are restarted.
- Useful if workers depend on each other.



Supervision Strategies:

Rest For One

- When a worker dies, that one *and all newer ones* are restarted.
- Useful if processes are in some kind of chain.



Supervision Strategies:

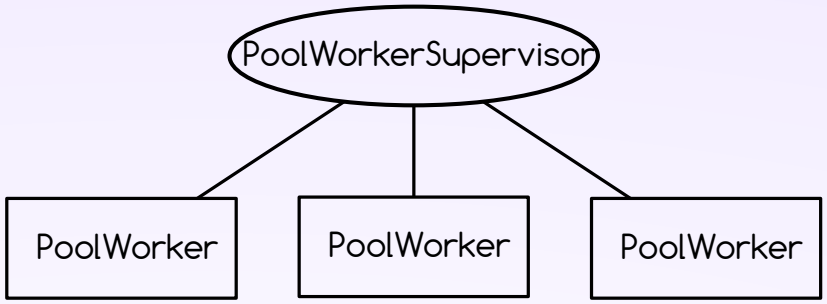
'Simple' One For One

- *is not like the others*
- Can only use workers of single type
- Does not start out with list of workers, but with *blueprint*.
- Useful for dynamically changing how many workers.

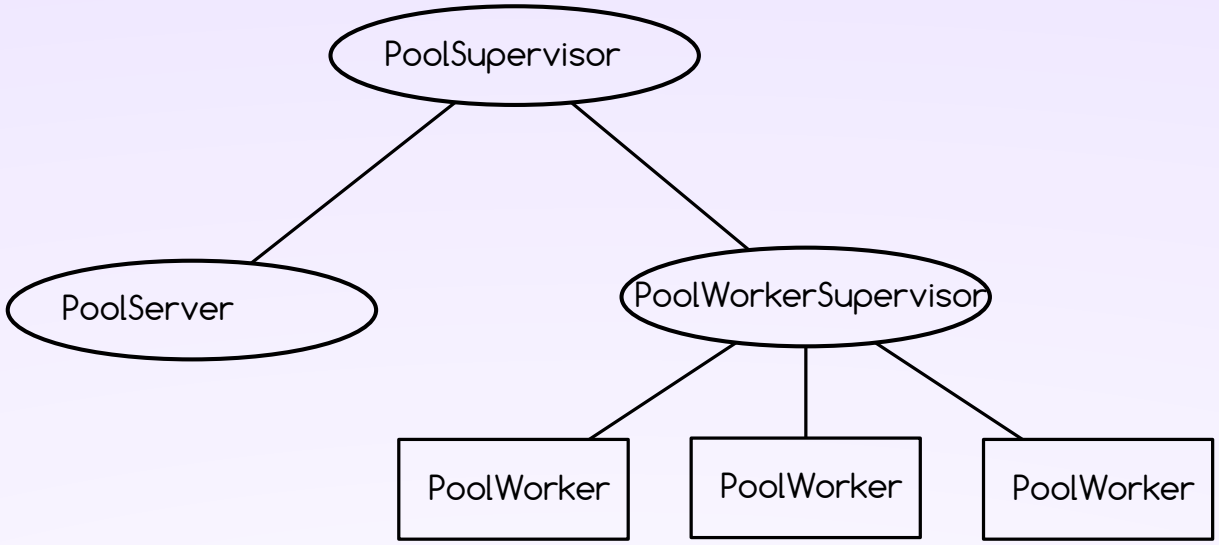
Supervision Trees

- Layers of Fault Tolerance
- Proper cleanup (prevents floating processes/memory leaks).

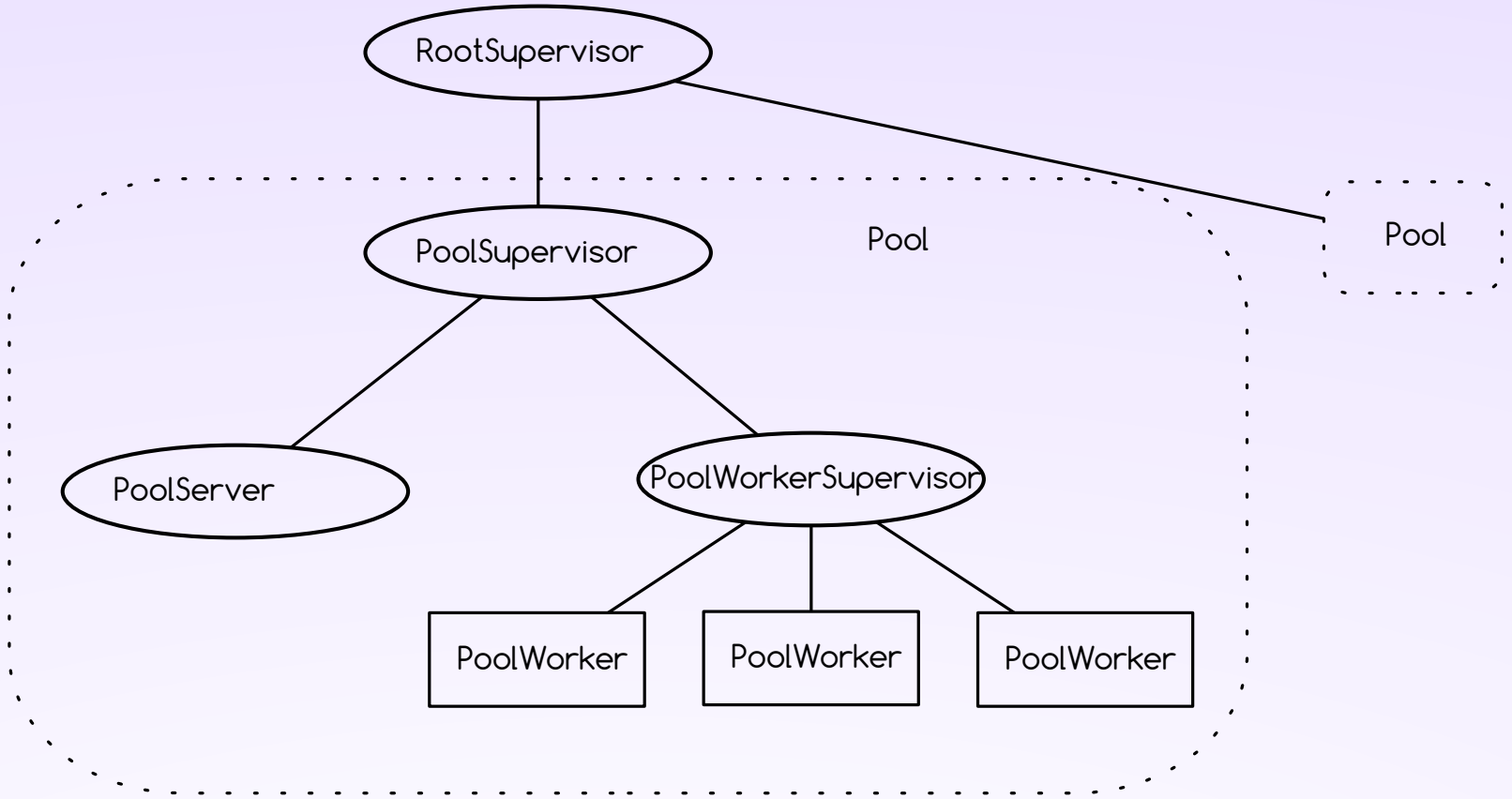
Supervision Trees: Example



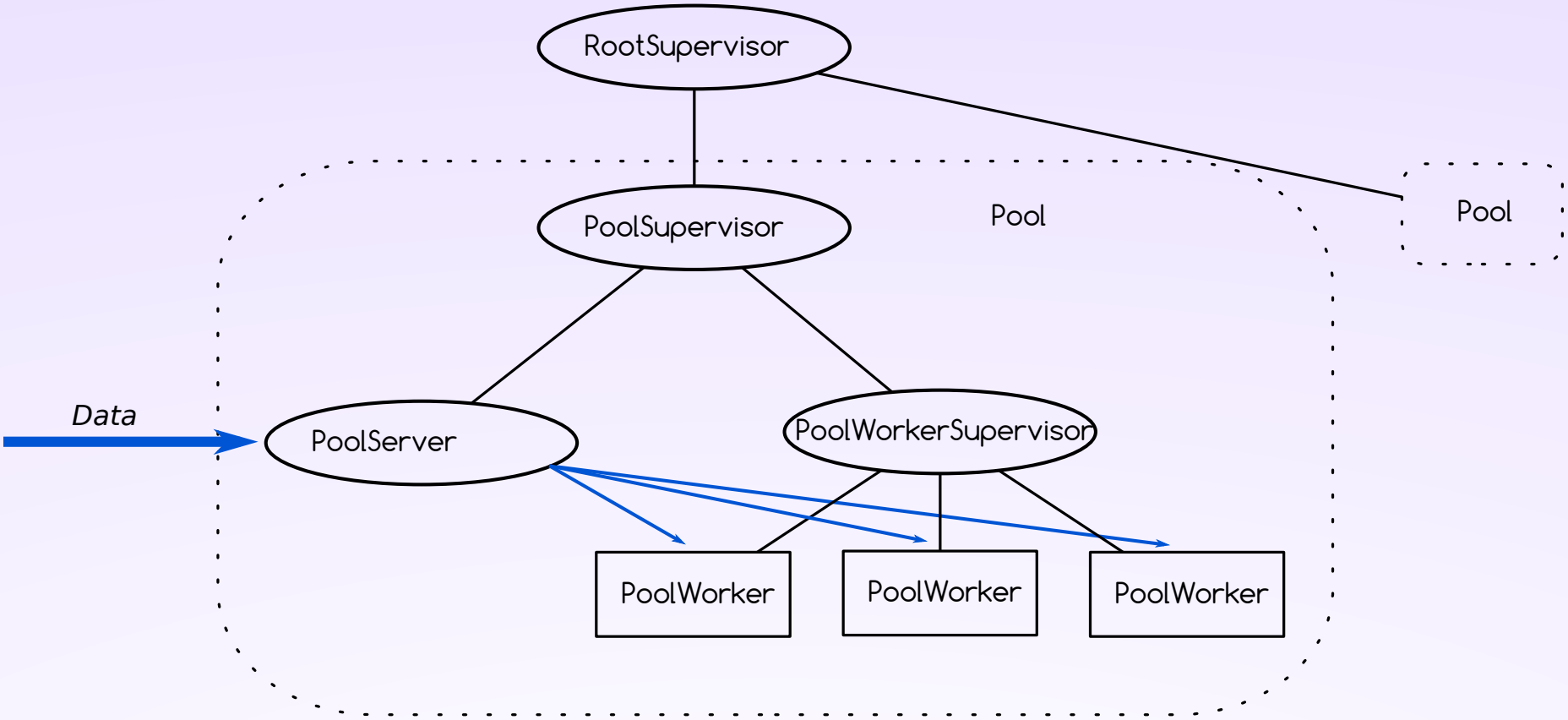
Supervision Trees: Example



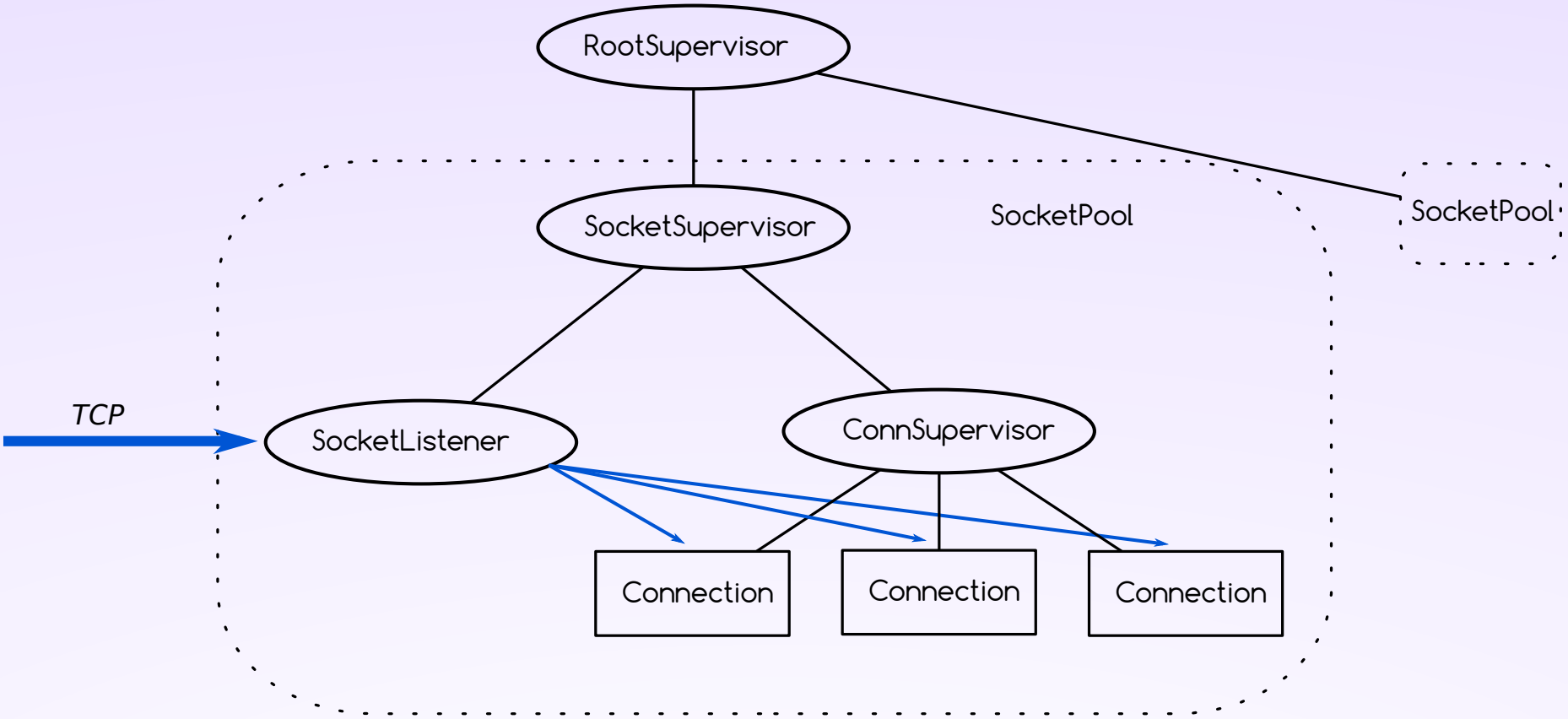
Supervision Trees: Example



Supervision Trees: Example



Supervision Trees: Example

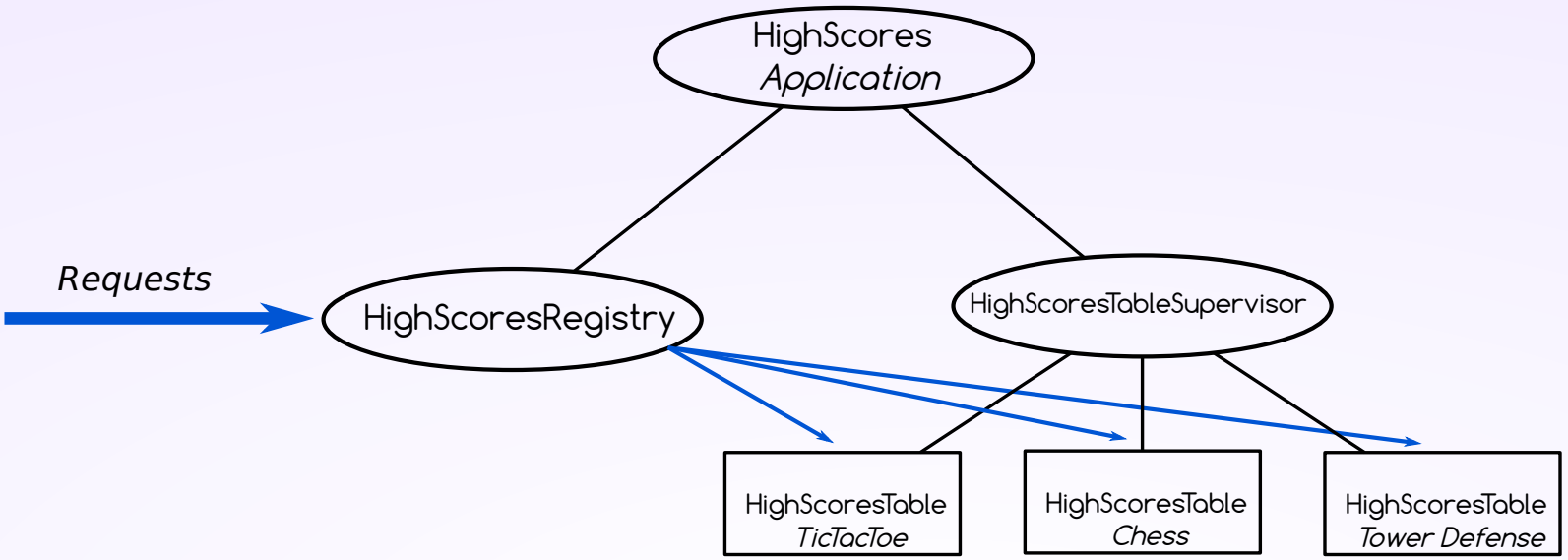


Application

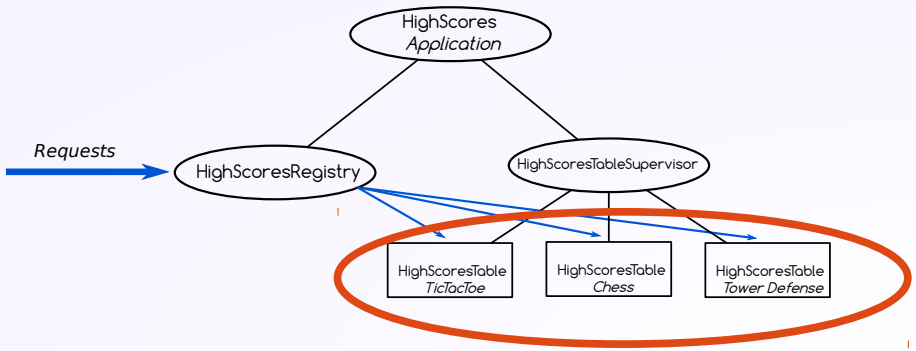
- Way to *package* a Supervision Tree
- *Stop/start/reload* as a unit.
- *Configurable* as a unit.
- *Re-usable* in other systems.
- Nice Debugging → `:observer.start`

High Scores: Multiple games

- Add our HighScoresTable as worker in a Supervision Tree
- Add Server that maps game names to HighScoreTables.

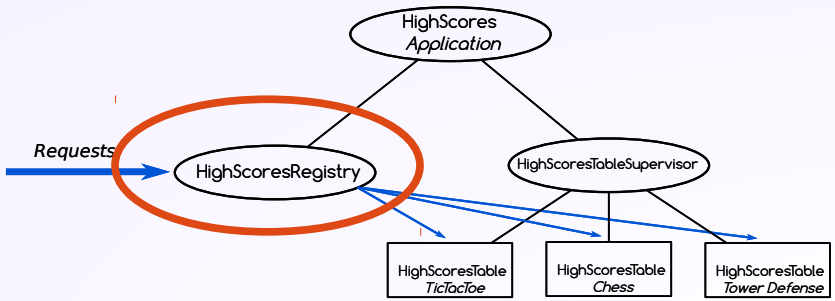


```elixir
defmodule HighScoresTable do
  use GenServer
  @moduledoc "A process that stores a table of high
scores."

  @doc "A table is list of scores and its max. length"
  defstruct scores: [], max_players: nil

  defmodule Score do
    @doc "A single score"
    defstruct name: "", score: 0, time: nil
  end


  # Outward API

  @doc "Starts the process, internally calls `init`."
  def start_link(max_players) do
    GenServer.start_link(__MODULE__, max_players)
  end

  @doc "Adds a new `score` for `name` to the table."
  def add_score(pid, name, score) do
    time = :erlang.time |> Time.from_erl!
    score = %Score{name: name, score: score, time:
time}
    GenServer.call(pid, {:add_score, score})
  end

  @doc "Lists current scores"
  def current_scores(pid) do
    GenServer.call(pid, :current_scores)
  end
```




```elixir
 # internal GenServer callbacks

  @doc "Initializes the GenServer"
  def init(max_players) do
    initial_state = %HighScoresTable{max_players:
max_players}
    {:ok, initial_state}
  end

  @doc "Adds a score to the table, updating the internal
state."
  def handle_call({:add_score, score = %Score{}}, _from,
state) do
    {response, new_scores} =
maybe_add_score(state.scores, state.max_players, score)
    new_state = %HighScoresTable{state | scores:
new_scores}
    {:reply, response, new_state}
  end

  @doc "Lists the current scores"
  def handle_call(:current_scores, _from, state) do
    {:reply, state.scores, state}
  end

  defp maybe_add_score(scores, max_players, new_score) do
    split_scores = Enum.split_while(scores, fn score ->
score.score >= new_score.score end)
    case split_scores do
      {all, []} when length(all) >= max_players ->
        {:unfortunate, scores}
      {higher, lower} ->
        new_scores = (higher ++ [new_score] ++ lower) |>
Enum.take(max_players)
        {{:congrats, length(higher)+1}, new_scores}
    end
  end
end
```

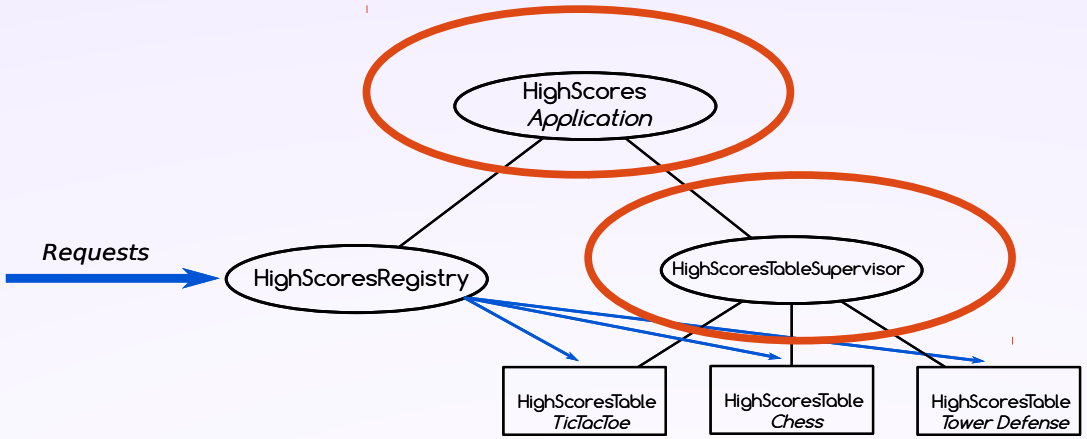
```elixir
defmodule HighScoresRegistry do
  use GenServer

  @doc "Starts the HighScoresRegistry, with the module name as
process alias so we don't have to pass the pid around"
  def start_link do
    GenServer.start_link(__MODULE__, :ok, name: __MODULE__)
  end

  def init(:ok) do
    {:ok, %{}}
  end

  # Outward API

  @doc "Adds a new game"
  def add_game(game, max_players \\ 10) do
    GenServer.call(__MODULE__, {:add_game, game, max_players})
  end

  @doc "Adds `name`'s `score` on `game` to that games table"
  def add_score(game, name, score) do
    {:ok, hst_pid} = get_game_pid(game)
    HighScoresTable.add_score(hst_pid, name, score)
  end

  @doc "Lists the current scores for `game`"
  def current_scores(game) do
    {:ok, hst_pid} = get_game_pid(game)
    HighScoresTable.current_scores(hst_pid)
  end

  # Returns the pid of a certain game.
  defp get_game_pid(game) do
    GenServer.call(__MODULE__, {:get_game, game})
  end
```




```elixir
  # internal GenServer callbacks

  def handle_call({:add_game, game_name, max_players},
_from, games) do
    if games[game_name] do
      {:reply, {:error, :game_already_exists}, games}
    else
      {:ok, pid} =
HighScoresTableSupervisor.start_child([max_players])
      Process.monitor(pid)
      games = Map.put_new(games, game_name, {pid,
max_players})
      {:reply, :ok, games}
    end
  end

  def handle_call({:get_game, game}, _from, games) do
    if games[game] do
      {pid, _} = games[game]
      {:reply, {:ok, pid}, games}
    else
      {:reply, {:error, :unknown_game}, games}
    end
  end

  @doc "Remove table processes when they crash, and start a
new table for that game."
  def handle_info({:DOWN, ref, :process, crashed_pid,
_reason}, games) do
    games =
      for {game_name, {pid, max_players}} <- games, into: %
{} do
        if pid == crashed_pid do
          {:ok, new_pid} =
HighScoresTableSupervisor.start_child([max_players])
          {game_name, {new_pid, max_players}}
        else
          {game_name, {pid, max_players}}
        end
      end
    {:noreply, games}
  end
end
```

```elixir
defmodule HighScoresTableSupervisor do
  use Supervisor

  def start_link do
    Supervisor.start_link(__MODULE__, :ok, name:
__MODULE__)
  end

  def init(:ok) do
    children = [
      worker(HighScoresTable, [])
    ]

    supervise(children, strategy: :simple_one_for_one)
  end

  def start_child(args) do
    Supervisor.start_child(__MODULE__, args)
  end

end
```

```elixir
defmodule HighScores do
  use Application

  def start(_type, _args) do
    import Supervisor.Spec, warn: false

    # Define workers and child supervisors to be
supervised
    children = [
      supervisor(HighScoresTableSupervisor, []),
      worker(HighScoresRegistry, [])
    ]

    opts = [strategy: :one_for_one, name:
HighScores.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```



~Live Demonstration~

Ways this could be improved

- Advanced Process naming using pg2.
- Storing things with ETS.
- Reduce more GenServer boilerplate using ExActor.
- Distribute score tables on multiple servers using `GenServer.multi_call`

Conclusions

- It's possible to quickly build scalable, concurrent applications in Elixir.
- Elixir has many built-in tools and mechanisms to make this easy.
- There's a lot more you can do.

Final Conclusion

- The most important thing, is to know what technologies are out there
- Use the right tool for the right job

Further Learning

- Book: Elixir in Action – *Saša Jurić*
  https://manning.com/books/elixir-in-action
- Book: Learn You Some Erlang – Fred Hébert
  http://learnyousomeerlang.com/
- Internet: Elixir Documentation
  http://elixir-lang.org/docs.html
- Talk: Discovering Processes – *Saša Jurić*
  https://youtu.be/y_b6RTes83c


CONGLATURATION !!!
YOU HAVE COMPLETED
A GREAT GAME.
AND PROOVED THE JUSTICE
OF OUR CULTURE.
NOW GO AND REST OUR
HEROES !