A Funny Thing Happened On The Way To The Phoenix

Mat Trudel
@mattrudel
github.com/mtrudel/talks

(Photo of the Phoenix Concert Theatre, Toronto)

Agenda

- What *is* this Phoenix thing, anyway?
- Phoenix and Plug
- Serving the Plug API
- Bandit Update
- What *is* an HTTP server, anyway?
- Scaling with processes
- Putting it all together

What’s underneath Phoenix?

What *is* this Phoenix thing, anyway?

(A gross simplification)

What *is* this Phoenix thing, anyway?

- It’s a web server. It serves web content
- Clients make requests for pages, API resources, sockets
- There is a lot more going on in Phoenix (DB, PubSub, etc)
- Today, we're looking at the HTTP part of Phoenix

HTTP in Phoenix

- Clients make requests for pages, API resources, sockets
- Requests are fundamentally isolated from one another
- The 'first point of entry' for an HTTP request into Phoenix is via the `Phoenix.Endpoint` behaviour

Phoenix.Endpoint

- "the boundary where all requests to your web application start"
- Also has a ton of 'upward facing' behaviour
- Everything 'up' from there (controllers, views, &c) is left as an exercise to the reader

Phoenix.Endpoint

- "the boundary where all requests to your web application start"
- Requests come to Phoenix via the `Plug` API
- `Phoenix.Endpoint` implements the `Plug` behaviour*

* `lib/phoenix/endpoint.ex:463` if you're curious

What is the Plug API?

Plug is an abstraction over HTTP requests & responses

Plug

```
defmodule HelloWorldPlug do
  # ...

  def call(%Plug.Conn{} = conn, _opts) do
    conn
    |> Plug.Conn.put_resp_content_type(“text/plain”)
    |> Plug.Conn.send_resp(200, "Hello World!")
  end
end
```

Phoenix is just a

(very complex)

Plug

What’s underneath Phoenix?

What’s underneath Plug?

A Plug-aware HTTP server

Cowboy is a Plug-aware* HTTP server

* when paired with `Plug.Cowboy`

Tying this back to Phoenix

- Phoenix runs an instance of Cowboy
- Hosted within Phoenix’s process tree
- Let’s take a look!

Tying this back to Phoenix

DEMO
(empex_demo + stack traces)

Bandit now supports Phoenix

Bandit is
a Plug-*native*
HTTP
server

Bandit, a pure Elixir Cowboy alternative

- Plug-native
- Written 100% in Elixir
- Robust HTTP/1.1 and HTTP/2 conformance
- Written from the ground up for correctness, performance & clarity
- Incredible performance (up to 5x Cowboy)

Bandit update

- Bandit 0.5.0 supports HTTP(S) Phoenix apps
- One-line change in Phoenix to enable
- Work entirely contained within Bandit project
- Websocket work up next

DEMO
(empex_demo + bandit adapter)

Bandit 0.5.0 drops *today*!

github.com/mtrudel/bandit

What’s underneath Phoenix?

What’s underneath Plug?

A Plug-aware HTTP server

What does an HTTP server *do*, exactly?

What does a webserver *do*, exactly?

1. Listen for connections
2. Handle each connection separately
    1. Parse the HTTP request into a `Plug.Conn` struct
    2. Pass this struct to a Plug implementation (eg: Phoenix)
    3. Provide backing support to read / write response

Problem naturally splits into two parts

1. Listen for connections (generic)
2. Handle each connection (HTTP specific)

```
# Generic
loop do
  socket = wait_for_connection()

  # HTTP Specific
  socket
  |> build_conn()
  |> plug_module.call()
end
```

Problem naturally splits into two parts

Protocol Specific

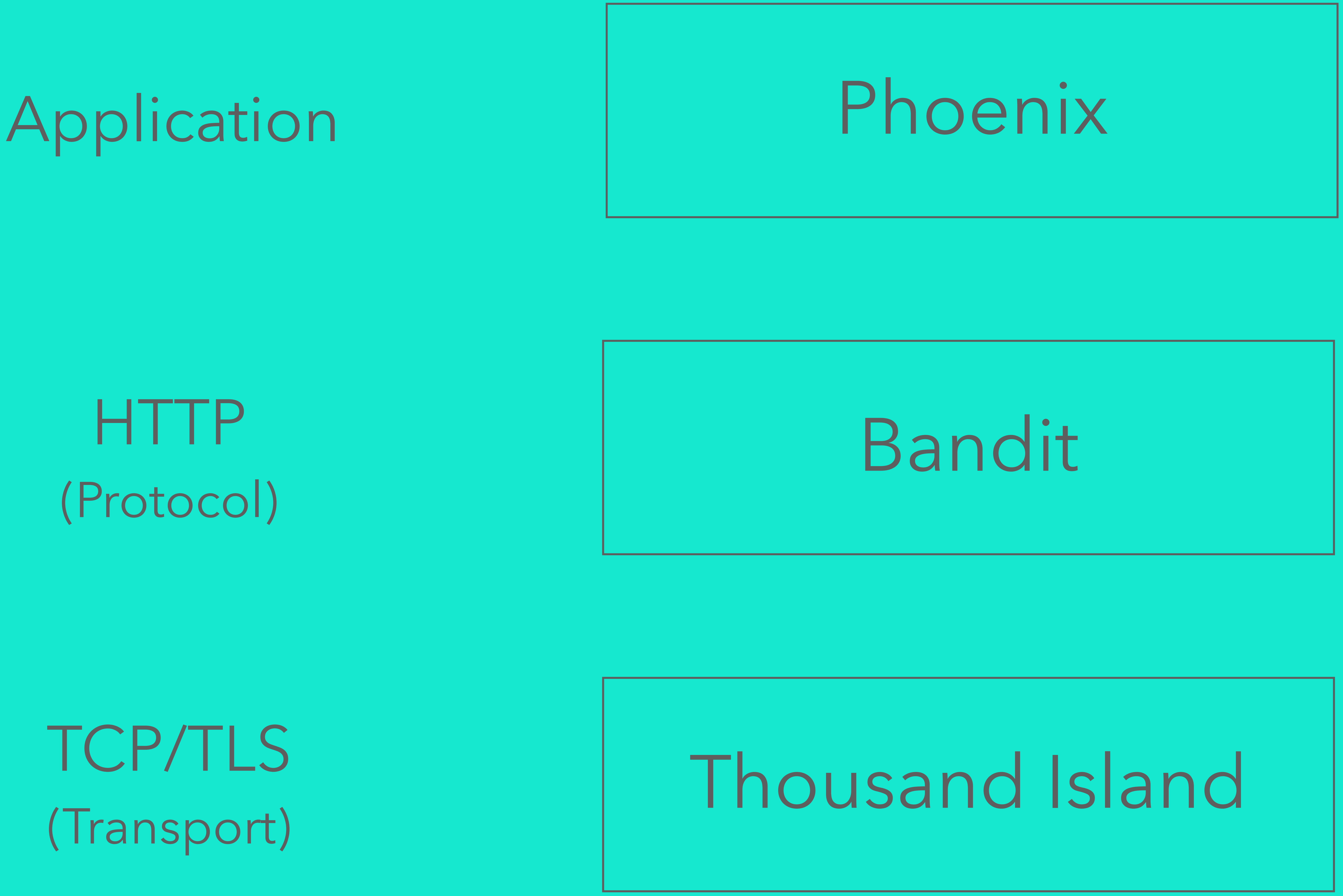
```
defmodule HTTPServer do
  def handle_connection(socket) do
    socket
    |> build_conn()
    |> plug_module.call()
  end
end
```

Generic

```
loop do
  socket = wait_for_connection()
  HTTPServer.handle_connection(socket)
end
```

Problem naturally splits into two parts

Protocol Specific	Bandit
Generic	Thousand Island


Application
Phoenix
HTTP
(Protocol)
Bandit
TCP/TLS
(Transport)
Thousand Island

Let’s talk about Thousand Island

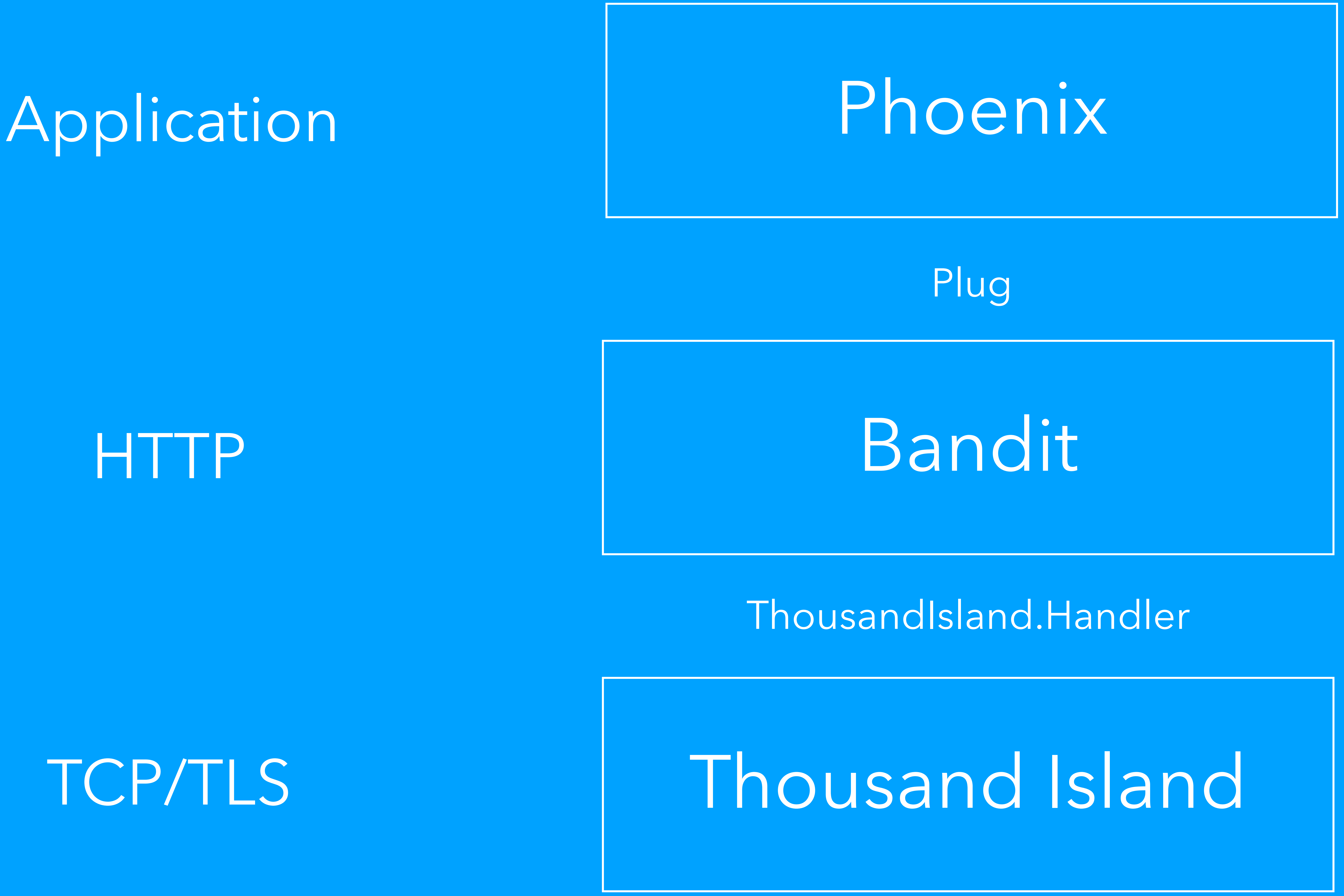
Thousand Island is a socket server

What does a socket server *do*, exactly?

- Listens for client connections over TCP/TLS
- Hands them off to an upper protocol layer (eg: an HTTP server)
- Provides send / receive / &c functionality
- Handles transport concerns (TLS, connection draining, etc)
- Does all of this efficiently and scalably

Thousand Island

- 100% Elixir socket server
- Supports TCP, TLS & Unix Domain sockets
- Fully wired for telemetry, including socket-level tracing
- Incredibly scalable, equally easy to understand (<1700 LoC)
- Extremely simple & powerful Handler behaviour for building protocols on top


Application
Phoenix
Plug
HTTP
Bandit
ThousandIsland.Handler
TCP/TLS
Thousand Island

`ThousandIsland.Handler`

'A GenServer-like API for sockets'

ThousandIsland.Handler

```elixir
defmodule ThousandIsland.Handler do
  @callback handle_connection(socket, state)
    :: {:close, state} | {:continue, state}

  @callback handle_data(data, socket, state)
    :: {:close, state} | {:continue, state}

  # …plus a few more for shutdown, errors, &c
end
```

Daytime

(RFC 867)

```elixir
defmodule Daytime do
  use ThousandIsland.Handler

  @impl ThousandIsland.Handler
  def handle_connection(socket, state) do
    time = DateTime.utc_now() |> to_string()
    ThousandIsland.Socket.send(socket, time)
    {:close, state}
  end
end

{:ok, pid} = ThousandIsland.start_link(handler_module: Daytime)
```

Echo

(RFC 862)

```
defmodule Echo do
  use ThousandIsland.Handler

  @impl ThousandIsland.Handler
  def handle_data(data, socket, state) do
    ThousandIsland.Socket.send(socket, data)
    {:continue, state}
  end
end

{:ok, pid} = ThousandIsland.start_link(handler_module: Echo)
```

Bandit

(Gross simplification)

```
defmodule Bandit.Handler do
  use ThousandIsland.Handler

  @impl ThousandIsland.Handler
  def handle_connection(socket, state) do
    Bandit.do_http(socket)
    {:close, state}
  end
end

{:ok, pid} = ThousandIsland.start_link(handler_module: Bandit.Handler)
```

Thousand Island Handler Processes

- Handlers are GenServers under the hood
- One process per client connection
- ...Bandit is implemented as a Handler
- ...Phoenix is hosted by Bandit
- ...so...

Each Phoenix request is run inside its own GenServer*

* Not entirely true (see HTTP/2)

The simplest socket server

```elixir
# Listen (this binds the port)
{:ok, listen_socket} = :gen_tcp.listen(4000, [active: false])

# Accept (this waits for a connection)
{:ok, connection_socket} = :gen_tcp.accept(listen_socket)

# Pass to Handler module to handle connection
Handler.do_connection(connection_socket)

# Close the connection
:gen_tcp.close(connection_socket)
```

Only runs once

Try #2: Run in a loop

```
def run do
  {:ok, listen_socket} = :gen_tcp.listen(4000, [active: false])
  accept(listen_socket)
end

defp accept(listen_socket) do
  {:ok, connection_socket} = :gen_tcp.accept(listen_socket)
  Handler.do_connection(connection_socket)
  :gen_tcp.close(connection_socket)
  accept(listen_socket)
end
```

Only one connection at a time

Try #3: Spin off a Task

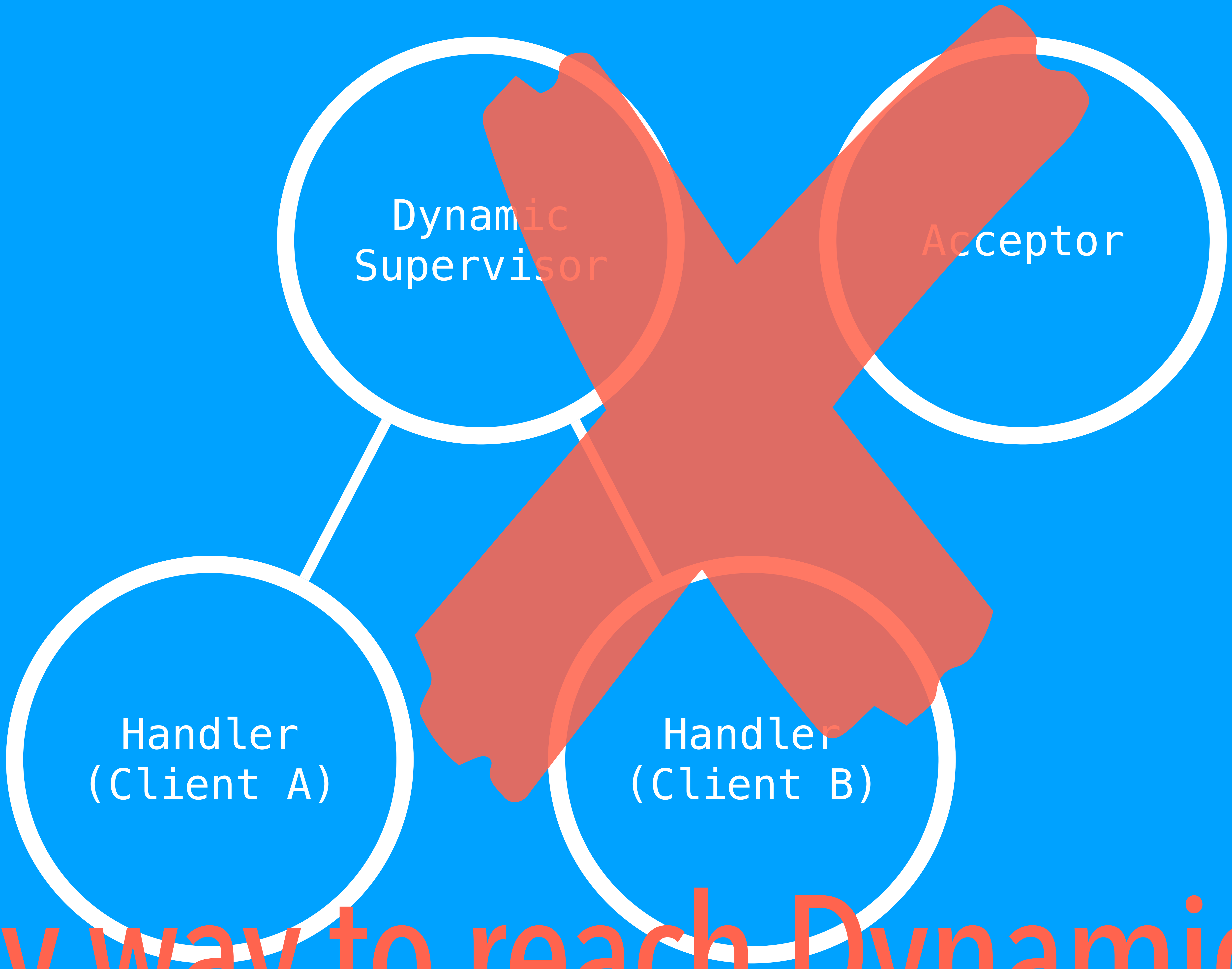
```elixir
def run do
  {:ok, listen_socket} = :gen_tcp.listen(4000, [active: false])
  accept(listen_socket)
end

defp accept(listen_socket) do
  {:ok, connection_socket} = :gen_tcp.accept(listen_socket)
  Task.start_link(fn ->
    Handler.do_connection(connection_socket)
    :gen_tcp.close(connection_socket)
  end)
  accept(listen_socket)
end
```

No supervision

Try #4: Use a Supervisor

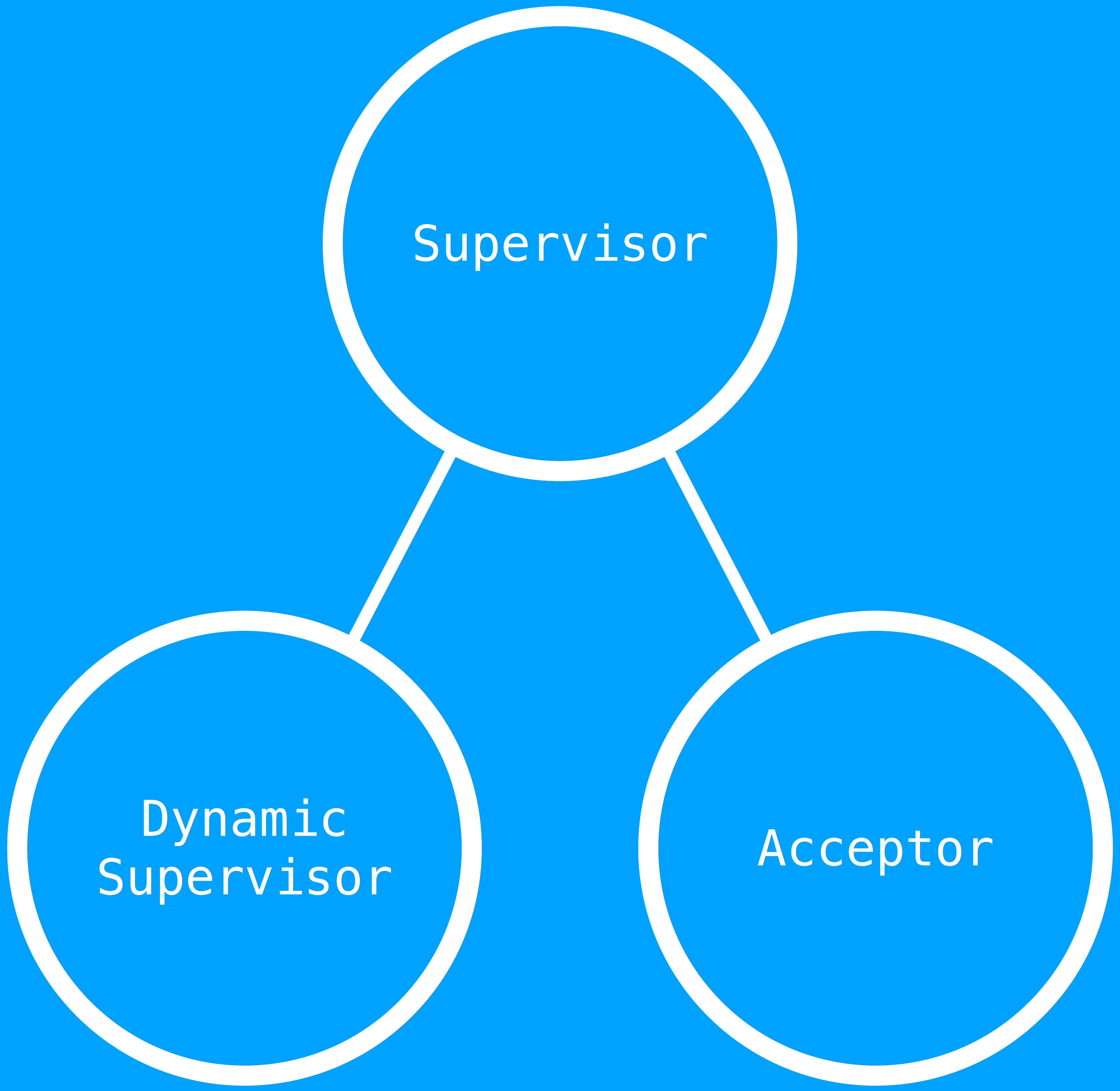
```elixir
def run do
  {:ok, listen_socket} = :gen_tcp.listen(4000, [active: false])
  accept(listen_socket)
end

defp accept(listen_socket) do
  {:ok, connection_socket} = :gen_tcp.accept(listen_socket)
  child_spec = {Handler, connection_socket}
  DynamicSupervisor.start_child(dyn_sup_pid, child_spec)
  accept(listen_socket)
end
```

Try #4: Use a Supervisor



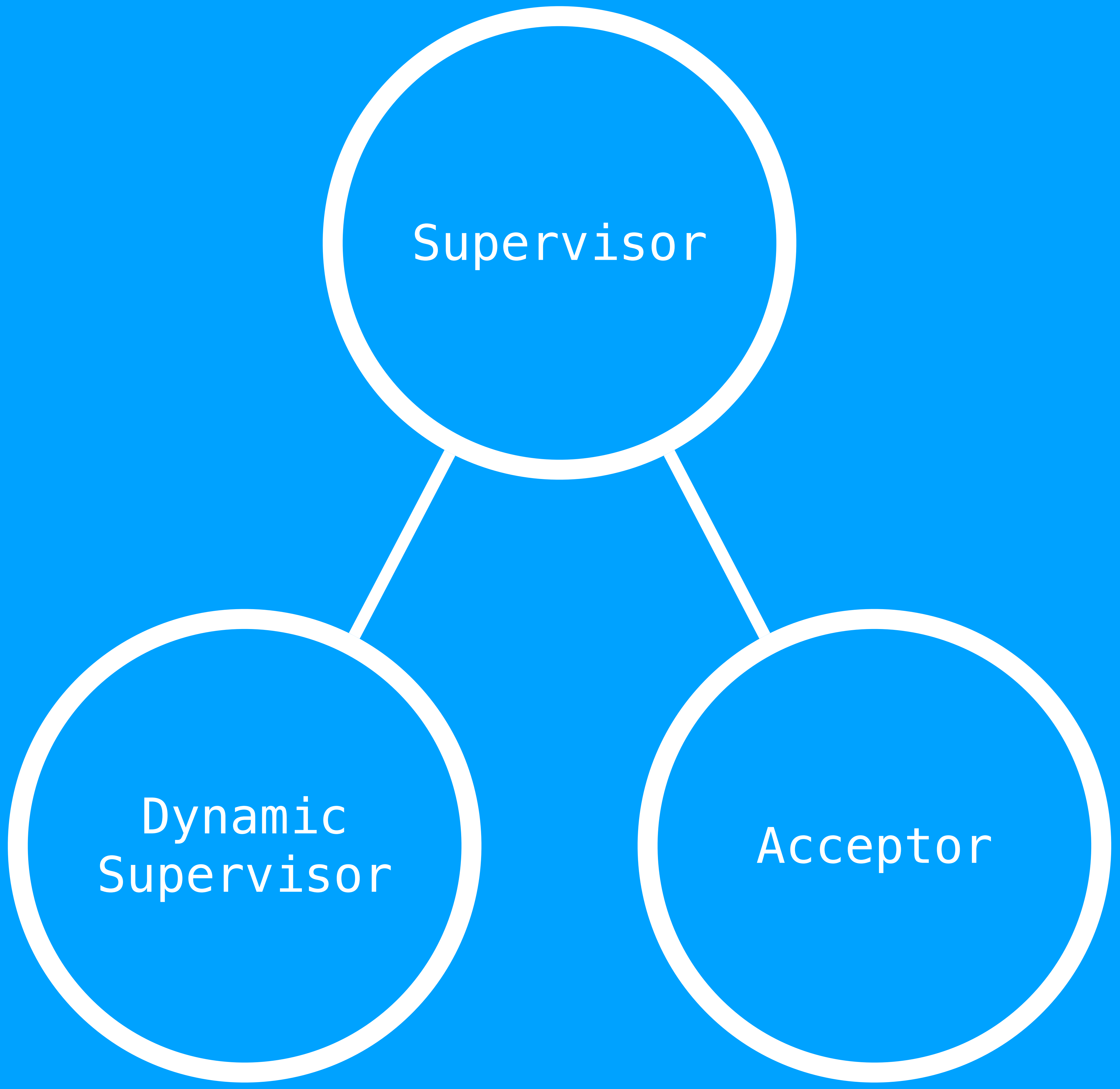
No easy way to reach Dynamic Supervisor

Try #5: Supervised Acceptor



Try #5: Supervised Acceptor

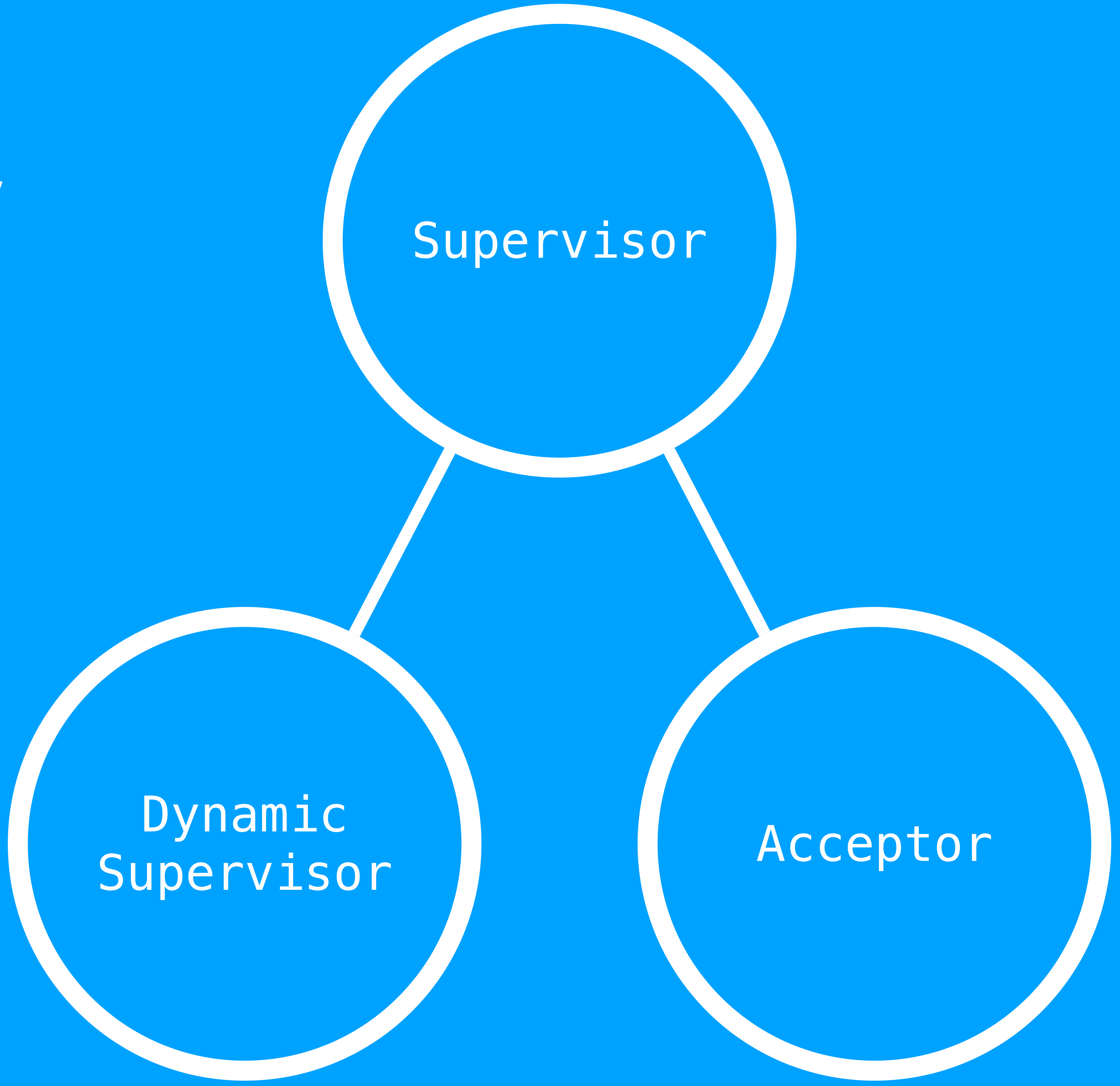
- Start Dynamic Supervisor first
- Acceptor only starts once Dynamic Supervisor is running
- The Acceptor can learn of the DynamicSupervisor's PID at startup
- This models dependencies properly



Observations

DynamicSupervisor

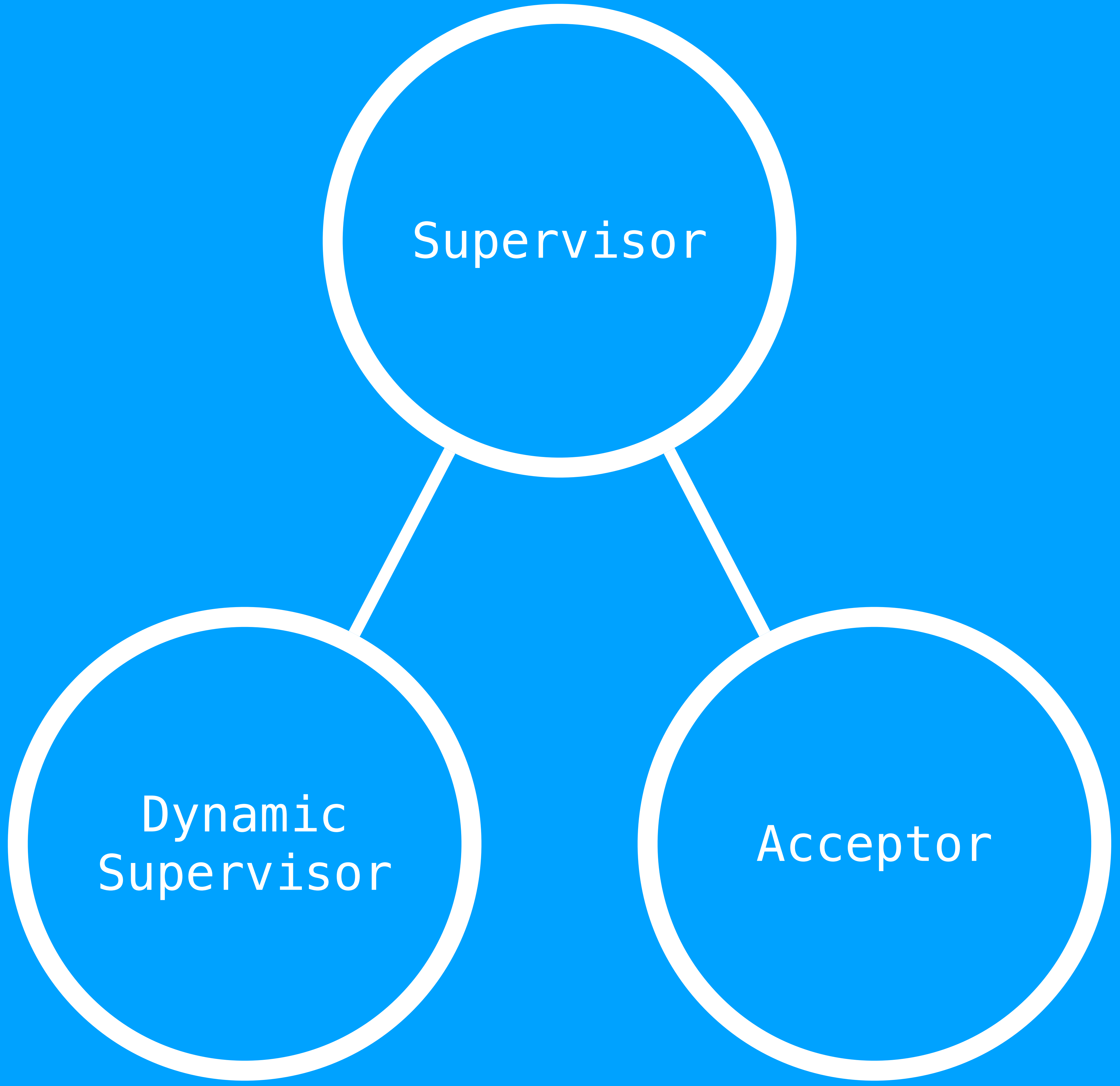
- Use when children are ‘on demand’ and independent
- Often paired with a ‘creator’ task such as an Acceptor that listens on network or queue
- Does not (and cannot) model dependencies between children



Observations

Supervisor

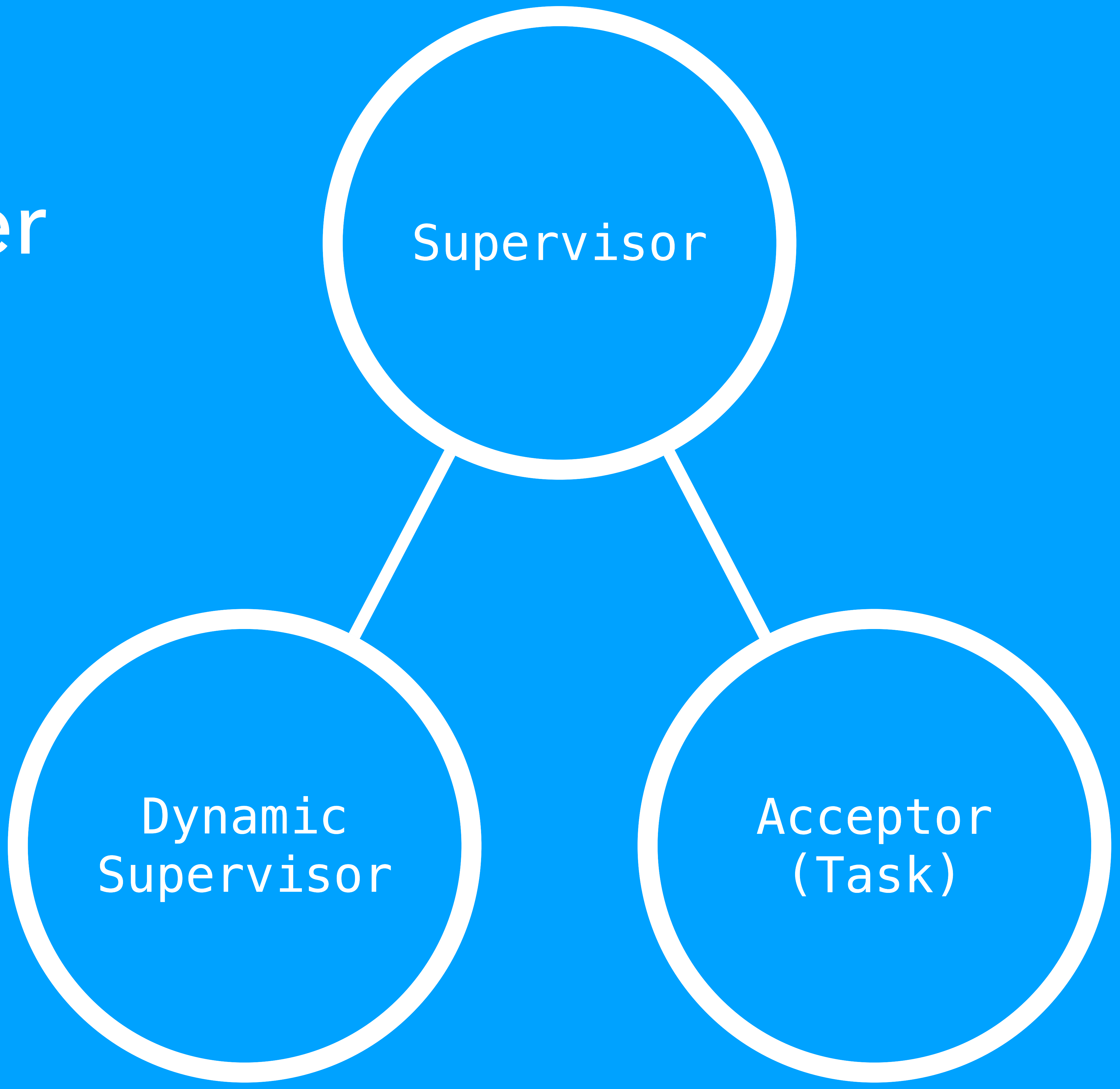
- Works best with predefined groups of processes
- Model dependencies via:
  - Start order
  - Supervision strategies (`:one_for_all`, `:rest_for_one`, &c)
- Keep domain out of Supervisors



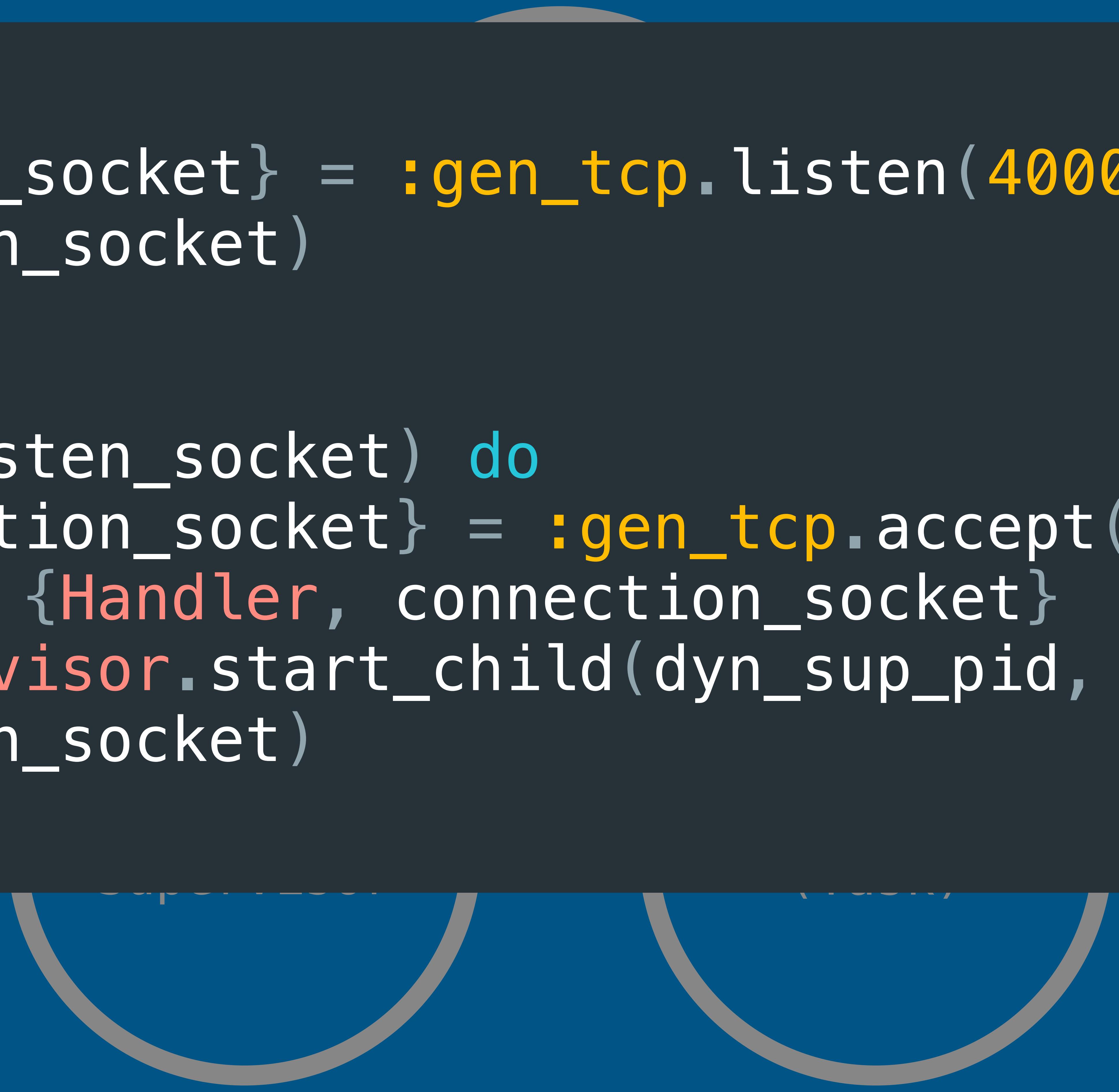
Observations

Acceptor is a Task!

- Very useful alternative to GenServer
- Doesn't need to be ephemeral
  `(use Task, restart: :permanent)`
- 'Does it need to be reachable?' is the key question to ask when deciding on Task vs. GenServer

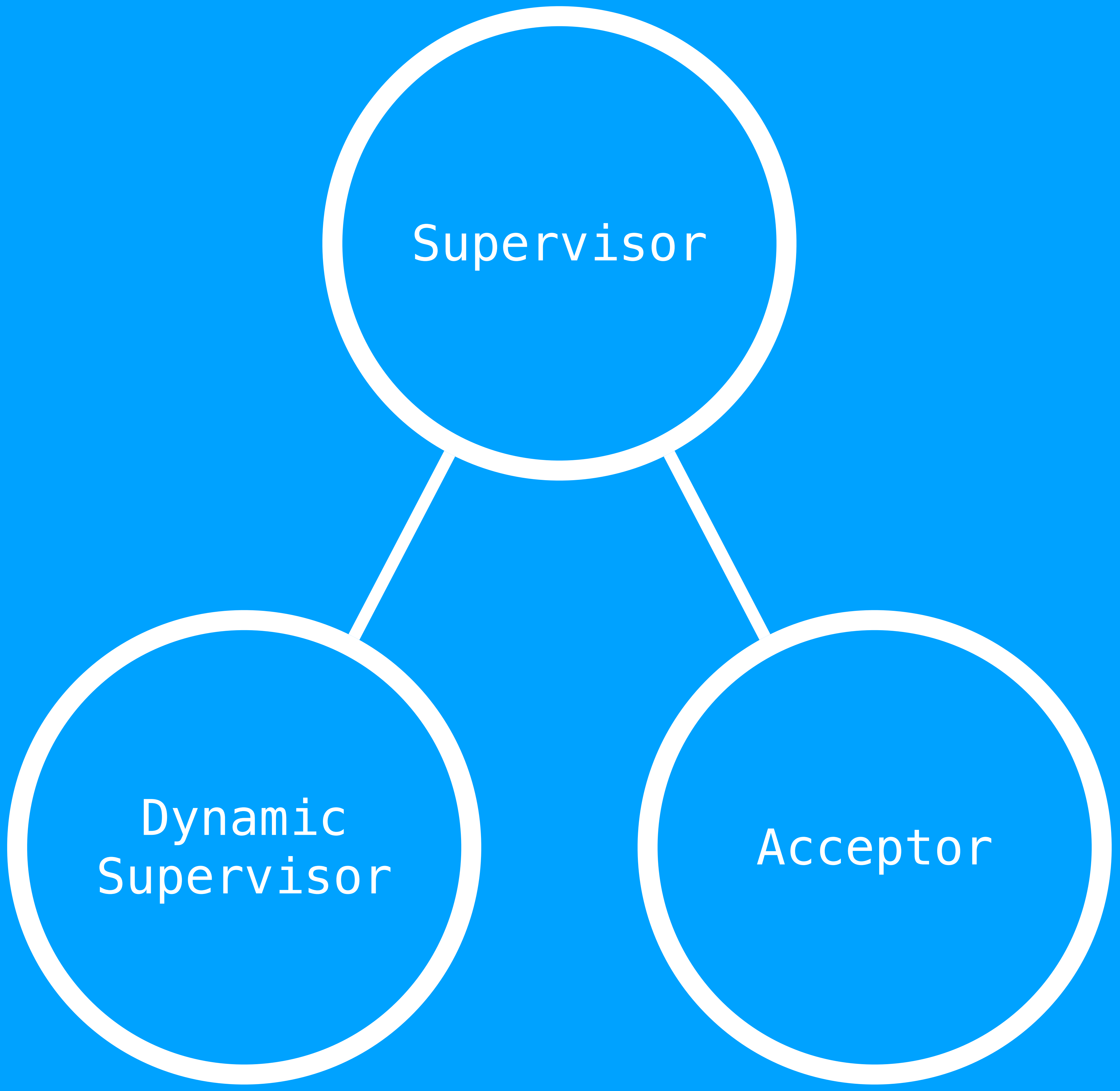


Try #5: Supervised Acceptor

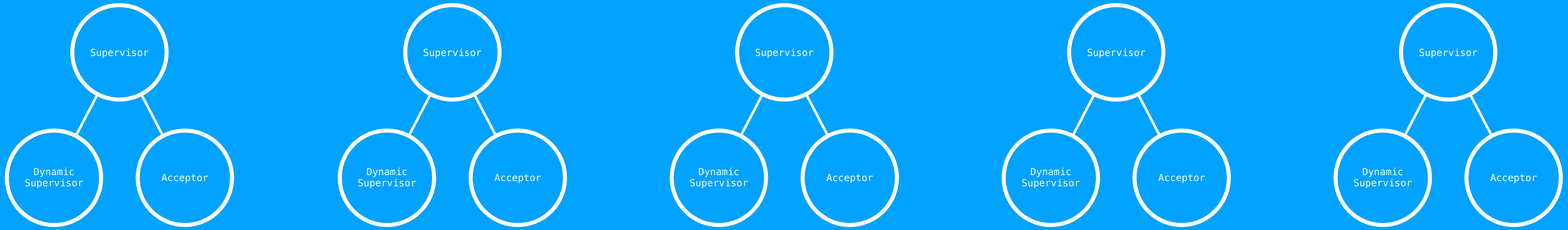


```elixir
def run do
  {:ok, listen_socket} = :gen_tcp.listen(4000, [active: false])
  accept(listen_socket)
end

defp accept(listen_socket) do
  {:ok, connection_socket} = :gen_tcp.accept(listen_socket)
  child_spec = {Handler, connection_socket}
  DynamicSupervisor.start_child(dyn_sup_pid, child_spec)
  accept(listen_socket)
end
```

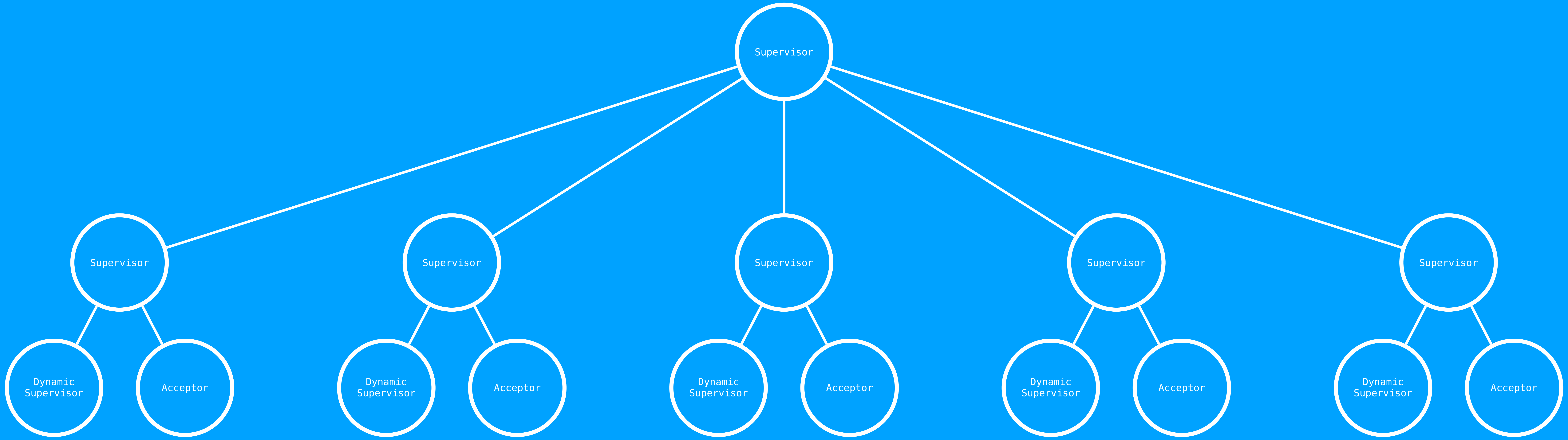
Try #6: Multiple Acceptor Trees



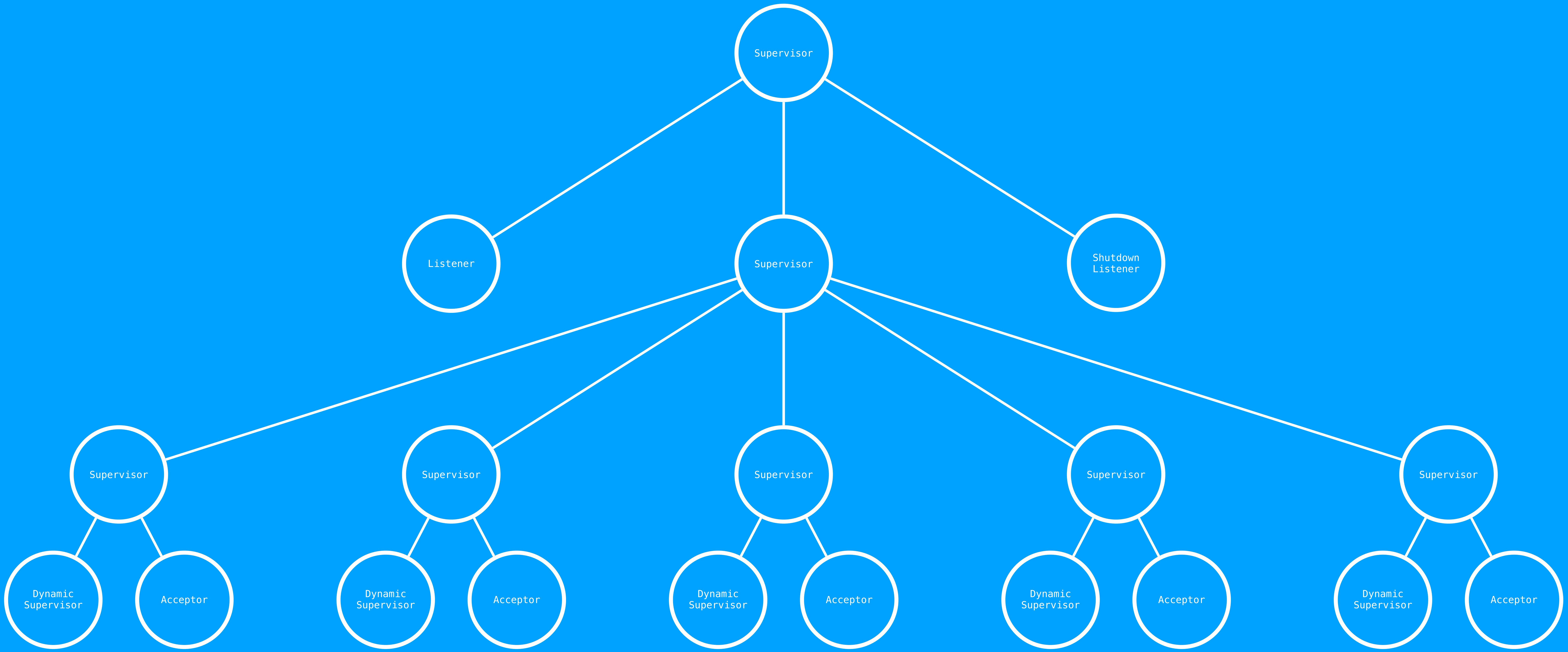
Try #6: Multiple Acceptor Trees



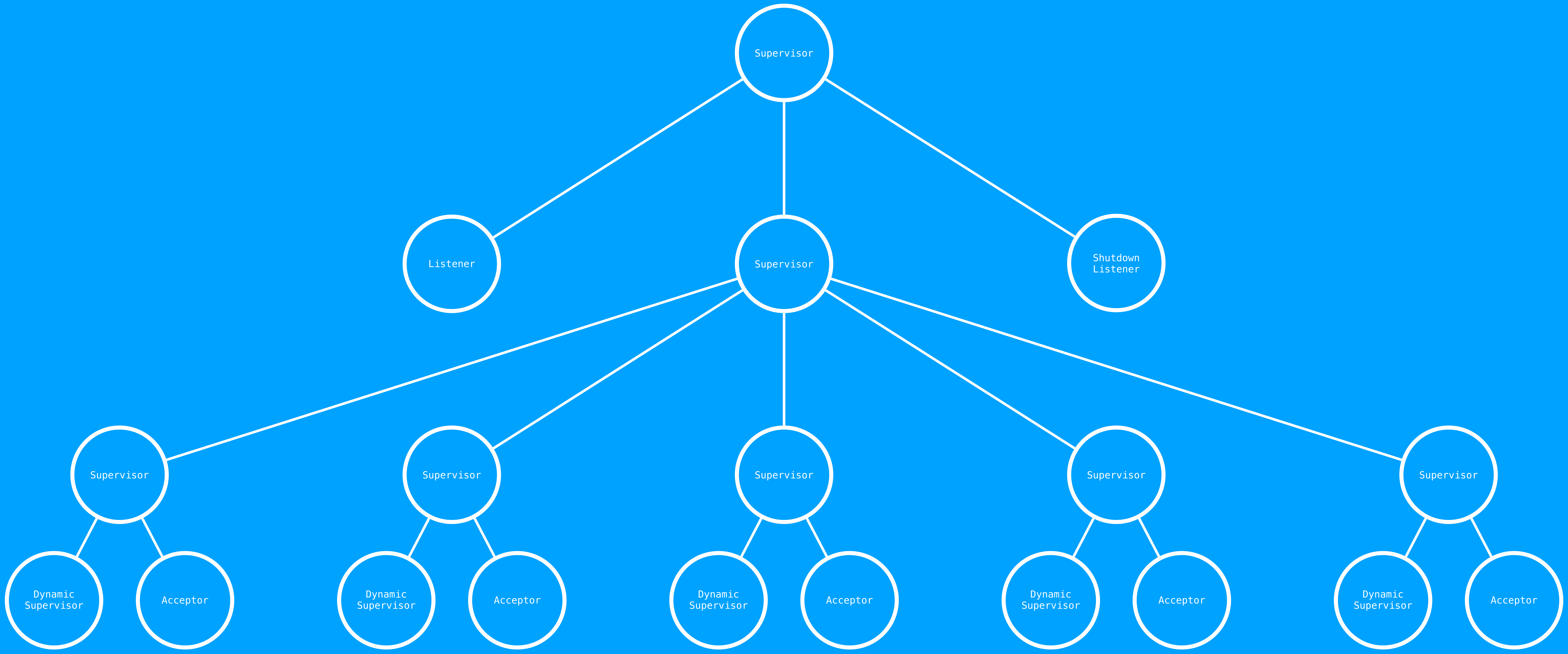
Try #6: Multiple Acceptor Trees



Try #6: Multiple Acceptor Trees



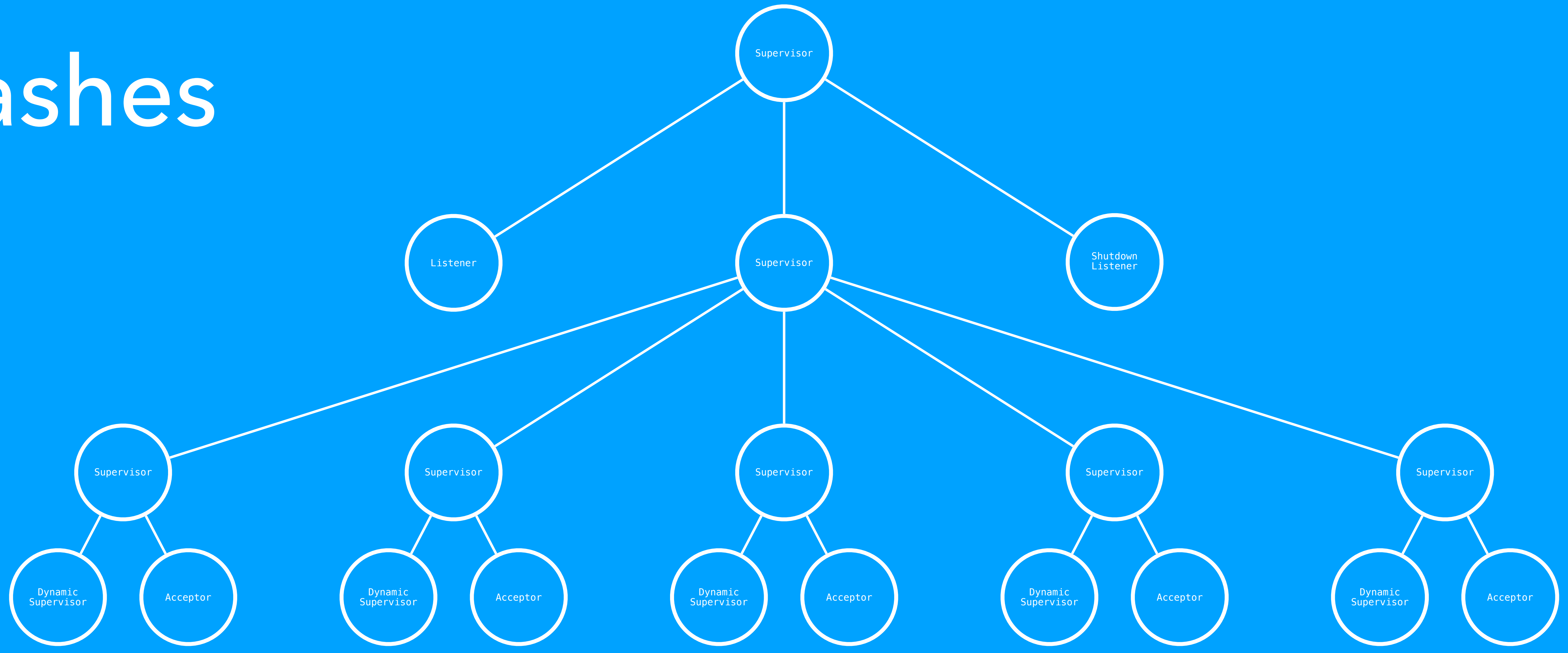
This is what Thousand Island looks like



Observations

Multiple Acceptor Trees

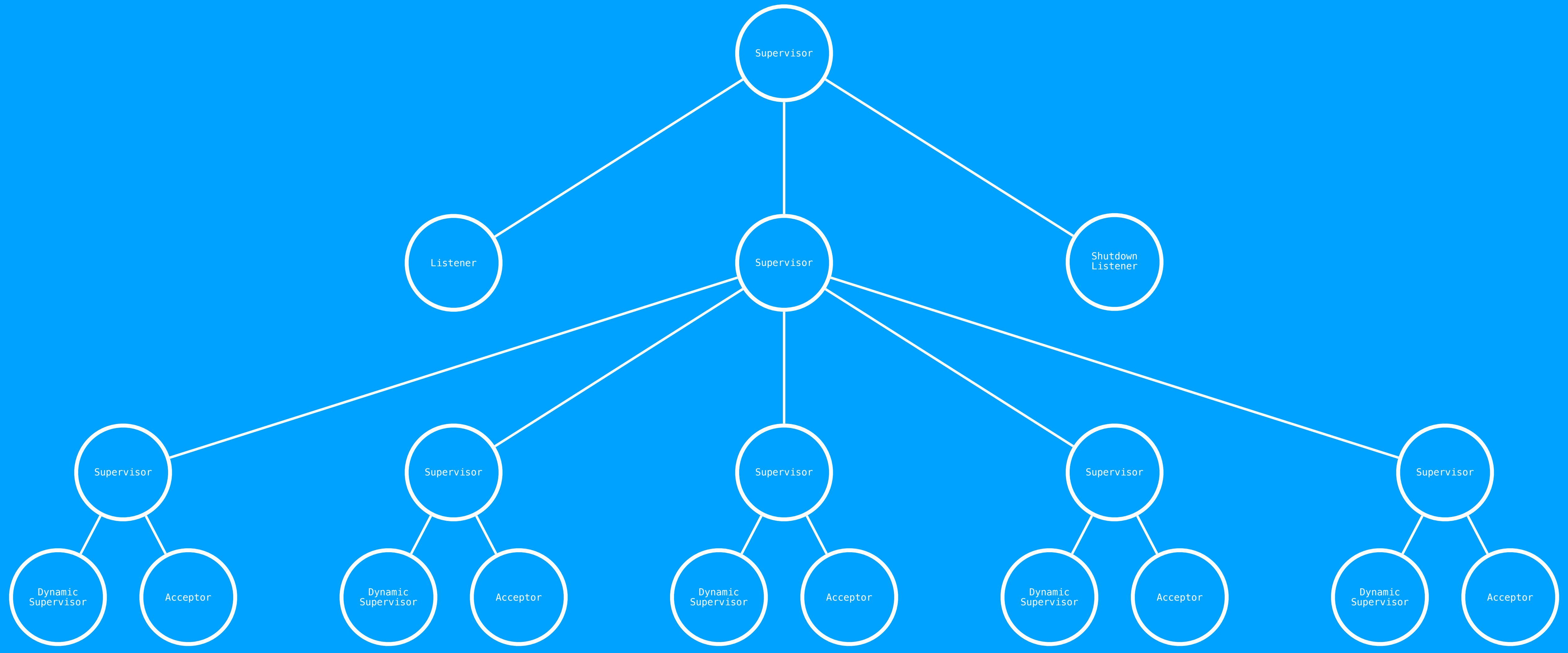
- Each acceptor tree is isolated
- Minimizes ‘blast radius’ of crashes
- Reduces contention for shared resources



Observations

Prefer Composition

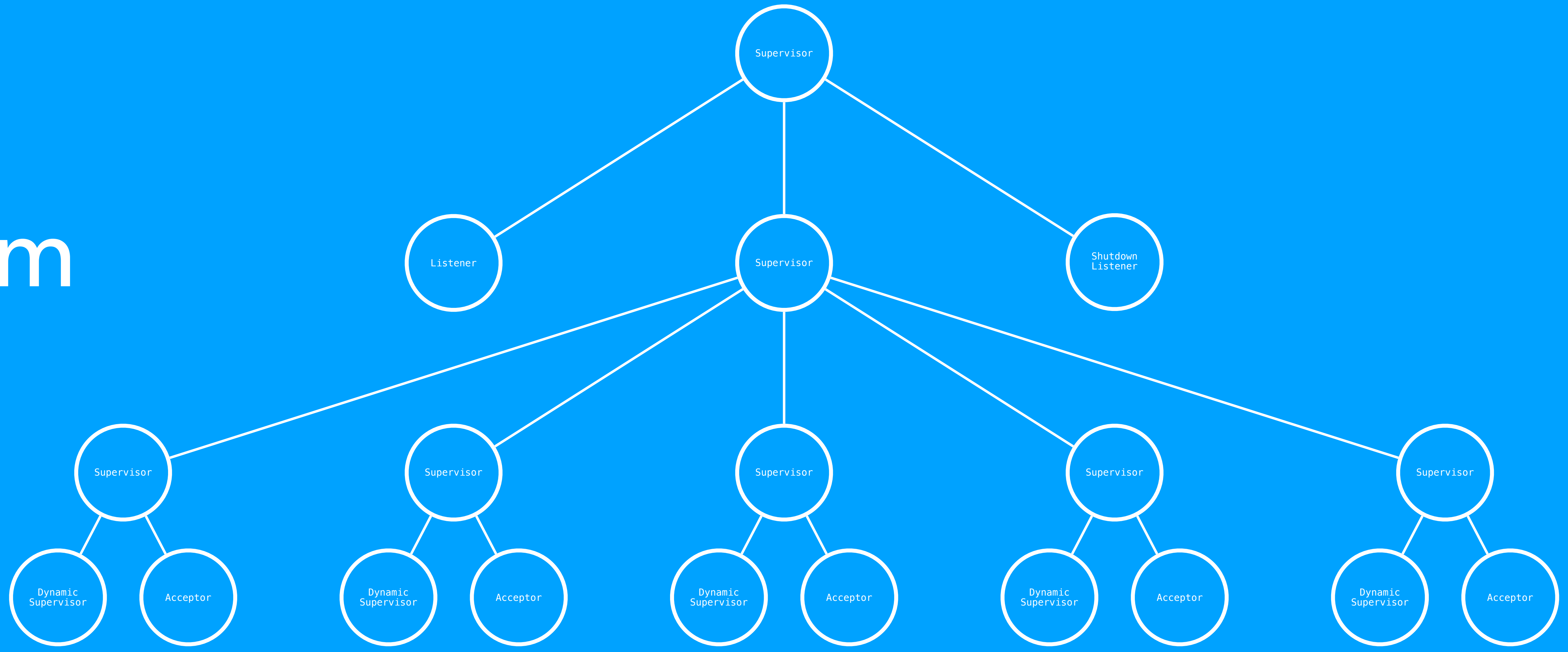
- Lots of simple supervisors is better than fewer complex ones
- Supervisors are just processes themselves
- Supervisors supervising supervisors is fine!



Observations

Prefer To Expose Process Trees

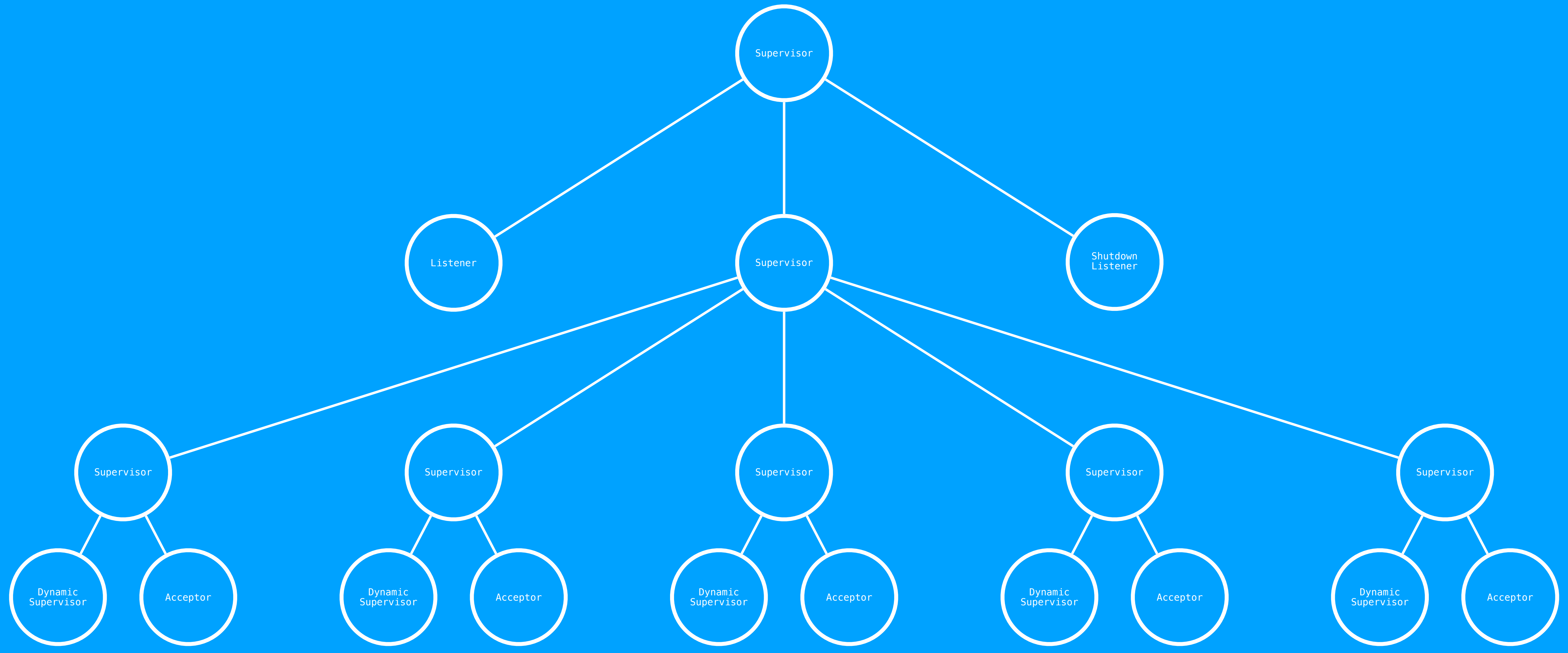
- Let users compose with their existing application
- Express dependencies to/from your existing application
- More flexible
  - eg: multiple Thousand Island servers on different ports



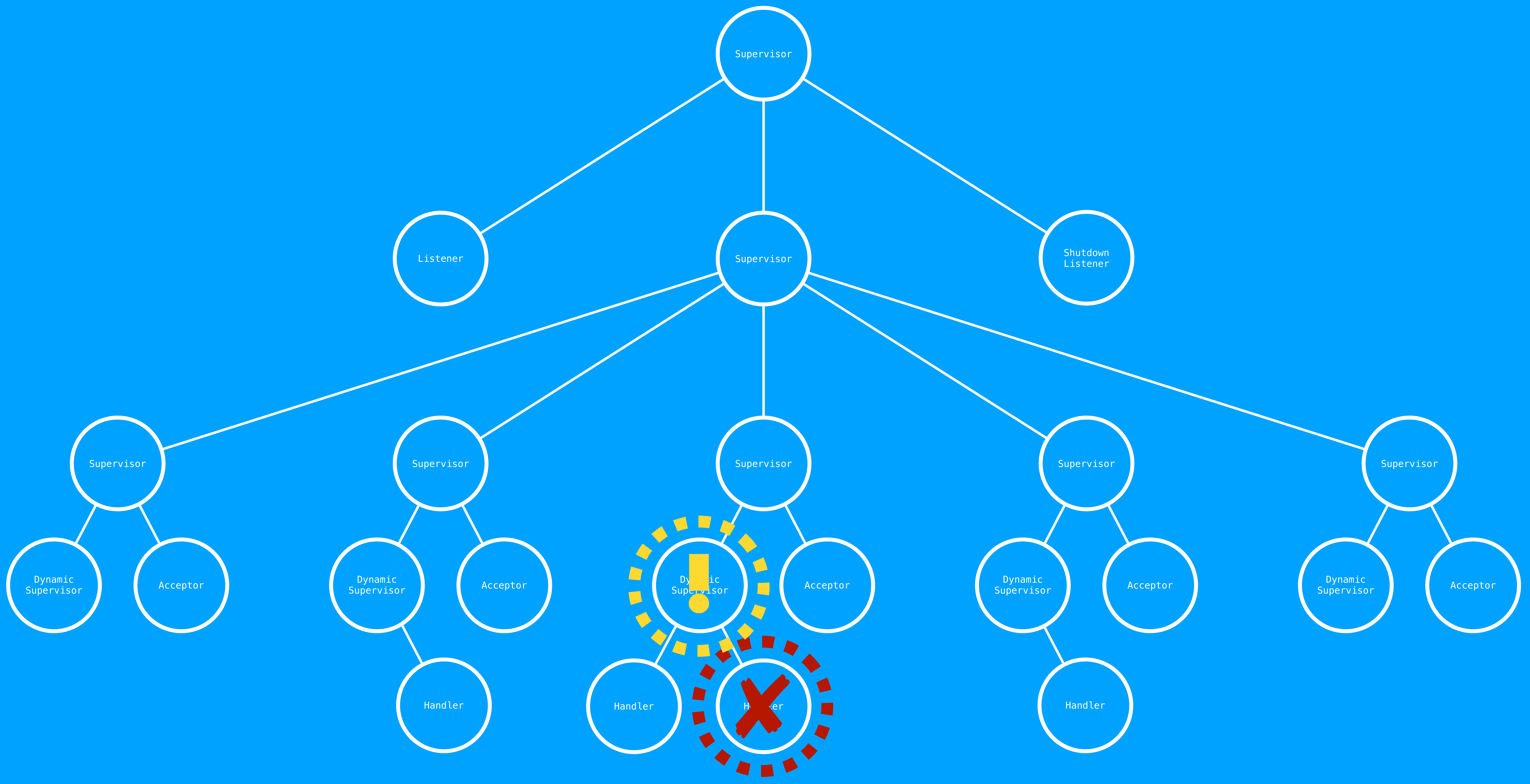
Observations

Supervisors are 'Restart Machines'

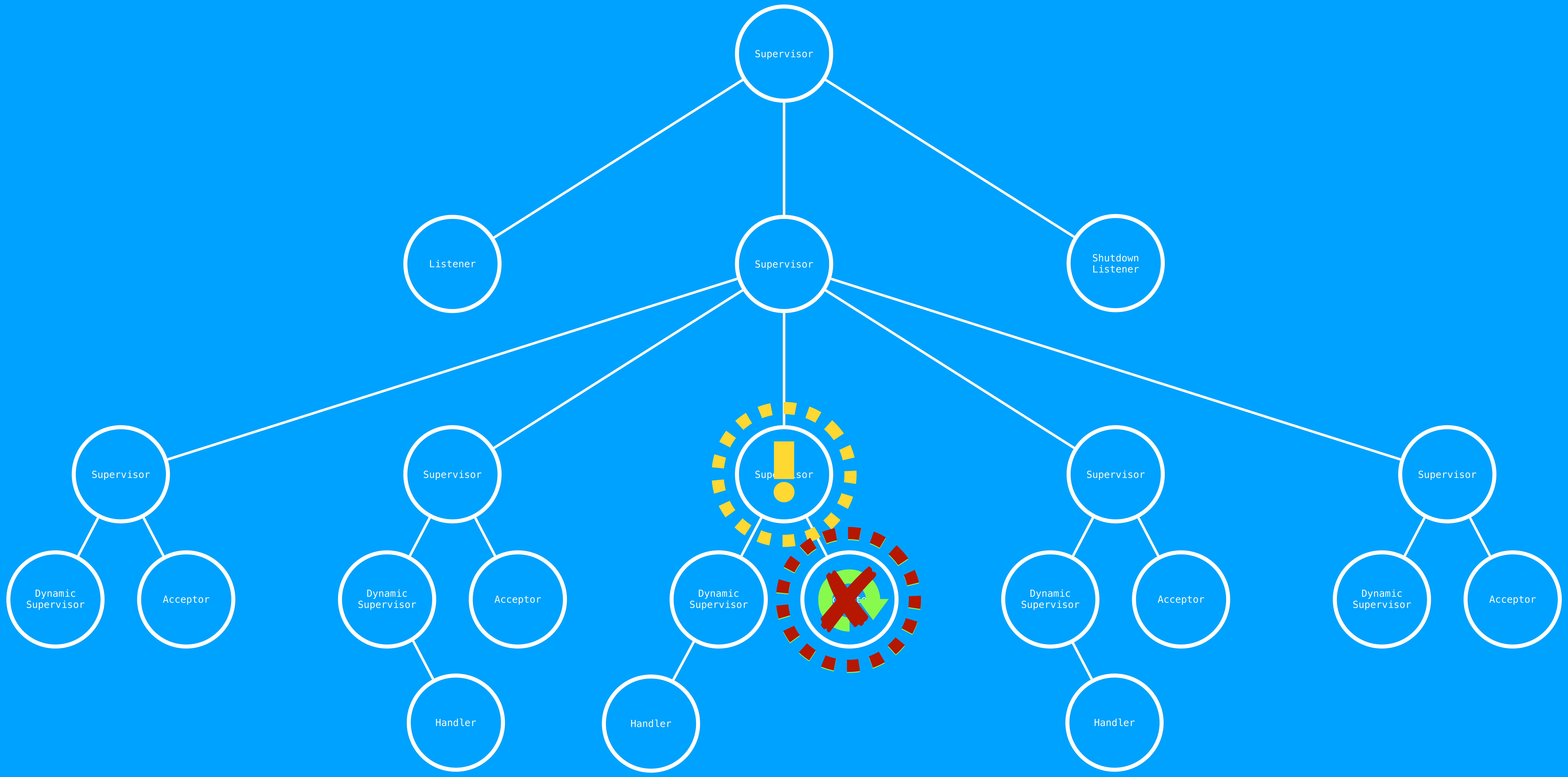
- Supervision helps contain the 'blast radius' of crashes
- Examples!



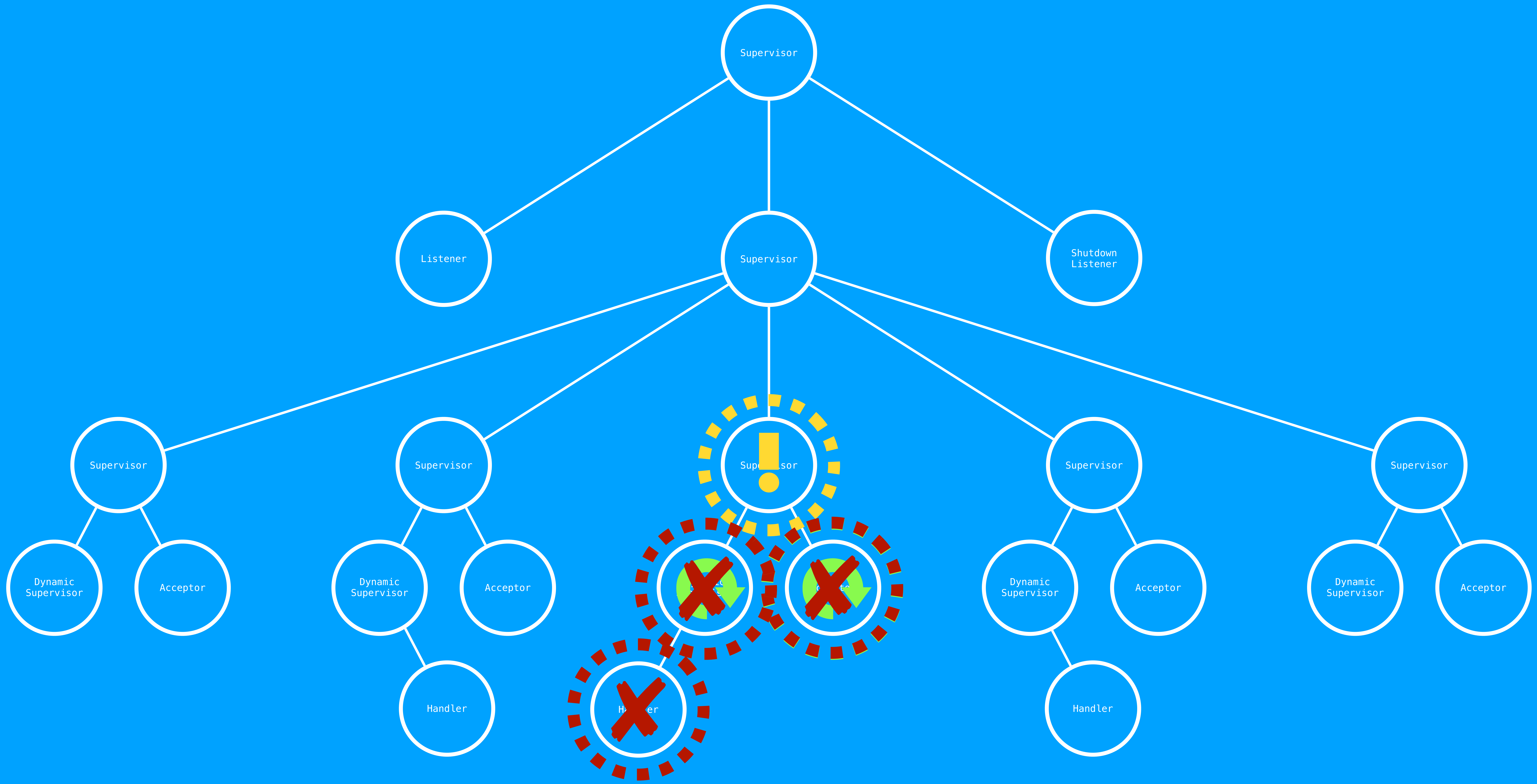
Supervisors are 'Restart Machines'



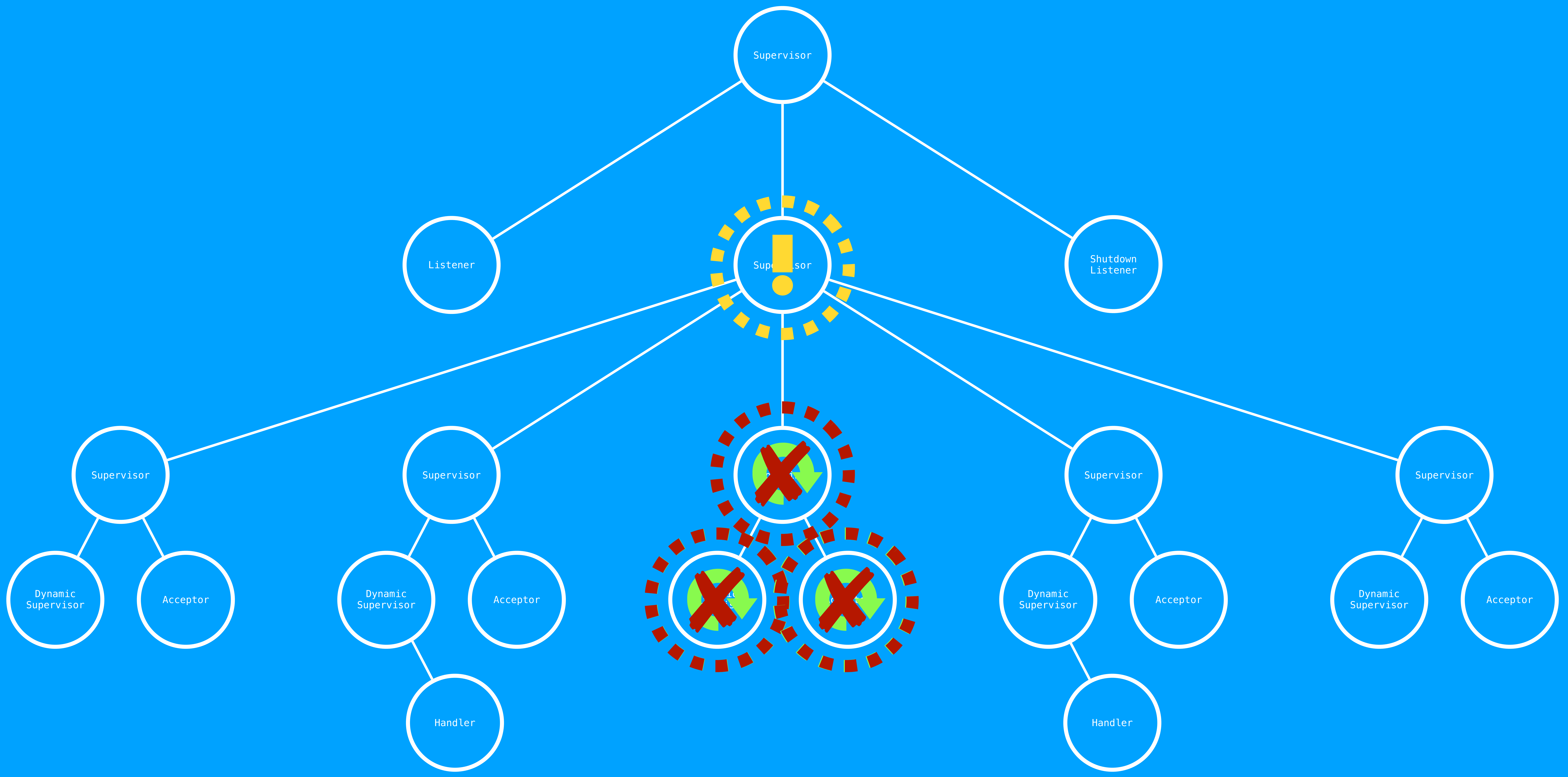
Supervisors are 'Restart Machines'



Supervisors are 'Restart Machines'



Supervisors are 'Restart Machines'



You can't

'Let It Crash'

without also knowing

How To Restart It

Putting It All Together

What's underneath Phoenix?

What's Underneath Phoenix?

Phoenix exposes itself as a Plug via `Phoenix.Endpoint`

… which is called via an HTTP server such as Bandit

… which is implemented as a Thousand Island Handler

… which is run inside a fresh GenServer process for each connection

… which is created by an acceptor task & supervised by a supervisor

… of which there are multiple instances

… all managed by a tree of Supervisors

… rooted at a PID returned by `ThousandIsland.start_link`

… that is wired into your Phoenix instance's process tree

What’s underneath Phoenix?

Very little magic

Not *too* many surprises

All things we’ve seen before

Careful fault containment

A textbook example of OTP

NOTHING TO BE SCARED OF

Agenda

- github.com/mtrudel/bandit
- github.com/mtrudel/thousand_island
- github.com/mtrudel/talks
- mat.geeky.net
- mat@geeky.net
- @mattrudel
- Thanks!

fin