Distributed Programming

Elixirconf EU Virtual 2020

Derek Kraan

Founder of Moose Code B.V.

Tutorial outline

This tutorial will have 4 parts, each with presentation and exercises

1. Introduction to Distributed Computing, Elixir/Erlang distribution primitives.
2. Distributed computing algorithms / patterns.
3. Extending Erlang’s supervision to a distributed context.
4. Strategies for dealing with network partitions.

Exercise: 2 Generals Problem

- 2 generals on opposite sides of a city must coordinate their attack.
- They may only communicate through messengers, who must pass through enemy territory, and may be captured.
- Each general wants to be certain the other will attack at the same time.

Exercise (5 minutes): devise a protocol that the generals can use to coordinate their attacks.

Consensus & FLP result

Consensus:

- All non-faulty processes must eventually decide on a value.
- All processes eventually choose the same value.
- The value that has been chosen must have been proposed by some process.

Impossibility of Distributed Consensus with One Faulty Process

Conclusion: there is no consensus protocol that can handle even a single process dying unexpectedly. (In a fully asynchronous setting)

MOOSE CODE

Local vs Distributed computing

Local:

- Shared memory
- Control shared access with locks
- Race conditions
- Exactly-once “delivery”

Distributed:

- Every “computational entity” has its own memory
- Nothing is shared
- Only message passing possible
- At most once or at least once delivery
- Must contend with networking issues (split brain, impossible to know if a message has arrived, etc)

What is distributed “really”?

Erlang: distributed only

Why use two programming models when you can use one?

8 fallacies of distributed computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn’t change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

MOOSE
CODE

Why distributed?

- Fault tolerance
- Performance

Why not distributed?

- Tradeoffs
- Complexity
- Design limitations

Is it worth it [for your problem domain]?

CAP Theorem

CAP theorem - ***C***onsistency, ***A***vailability, ***P***artition tolerance

Consistency - Every read receives the most recent write or an error

Availability - Every request receives a (non-error) response, without the guarantee that it contains the most recent write

Partition tolerance - The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

CAP Theorem

CAP theorem - ***C***onsistency, ***A***vailability, ***P***artition tolerance

Typically formulated as “choose 2”, but this is overly simplistic.

Commonly:

- ***CP*** - prioritize consistency: nobody attacks when messengers are intercepted in the field
- ***AP*** - prioritize availability: one side may attack alone when messengers are intercepted in the field
- ***CA(p)*** - some trade-off between C, A, and P, to ensure both C and A during certain P events (which are rare).

Key point: 100% C, 100% A, 100% P is impossible.

MOOSE CODE

Exactly-once delivery

Impossible. Have to choose between:

- At most once
- At least once

Idempotency

Idempotency: the same operation can be applied more than once without changing the result.

Eg: max/2

```
X |> max(5) |> max(5) |> max(5)
```

```
X |> max(5)
```

Idempotency to the rescue!

Idempotency turns “at least once” into “exactly once”.

This only works if you can design an operation to be idempotent.

MOOSE CODE

Idempotency to the rescue! pt II

Example:

Not idempotent: POST /account/XXX/transfer?to_account=YYY&amount=500.0

Idempotent: POST /account/XXX/transfer/ABC?to_account=YYY&amount=500.0

Receiver only processes transfer ABC once. Sender can send this request as many times until it gets confirmation of receipt.

Clustering topologies - mesh

Every node connected to every other node (Erlang clustering)

Clustering topologies - ring

Every node connected to its 2 neighbours

Clustering topologies - tree

Nodes are connected in a tree pattern

Clustering topologies - partial mesh

Nodes are connected to their neighbours in a ring + 2 other nodes across the ring

Clustering topologies - and many more

Erlang’s process model

- A process has its own memory and execution context.
- Lightweight “threads” (not OS threads) that are managed by the Erlang VM (BEAM)
- Process has its own stack and heap.
- Communication only possible with messages.

Erlang’s process model

Important BIFs (built-in functions):

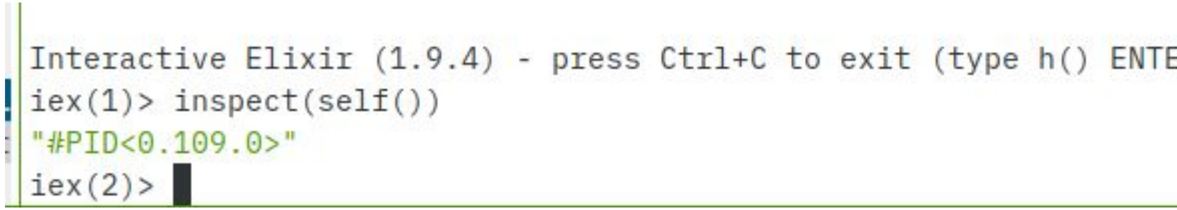
- spawn/3
- register/2
- whereis/1
- send/2
- receive/1

Anatomy of a pid

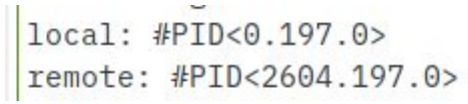
Process Identifier

Three-tuple: **#PID<node.process_id.version>**

Local **node** is always shown as “0”



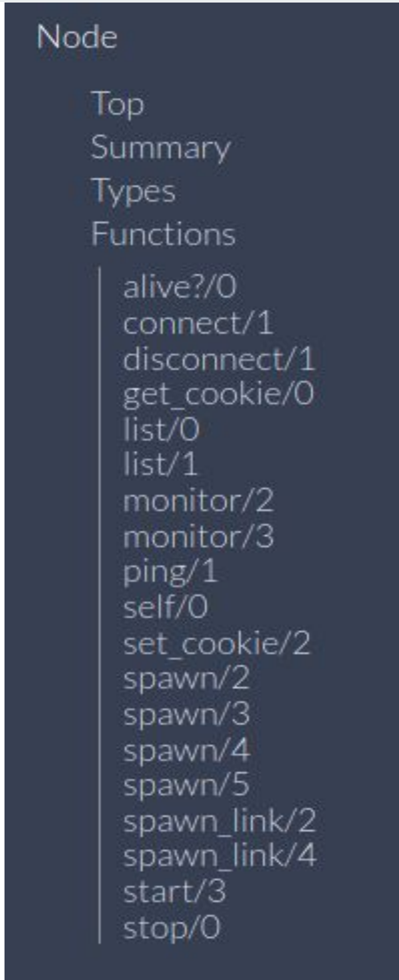
```
Interactive Elixir (1.9.4) - press Ctrl+C to exit (type h() ENTE
iex(1)> inspect(self())
"#PID<0.109.0>"
iex(2)>
```



```
local: #PID<0.197.0>
remote: #PID<2604.197.0>
```

Node

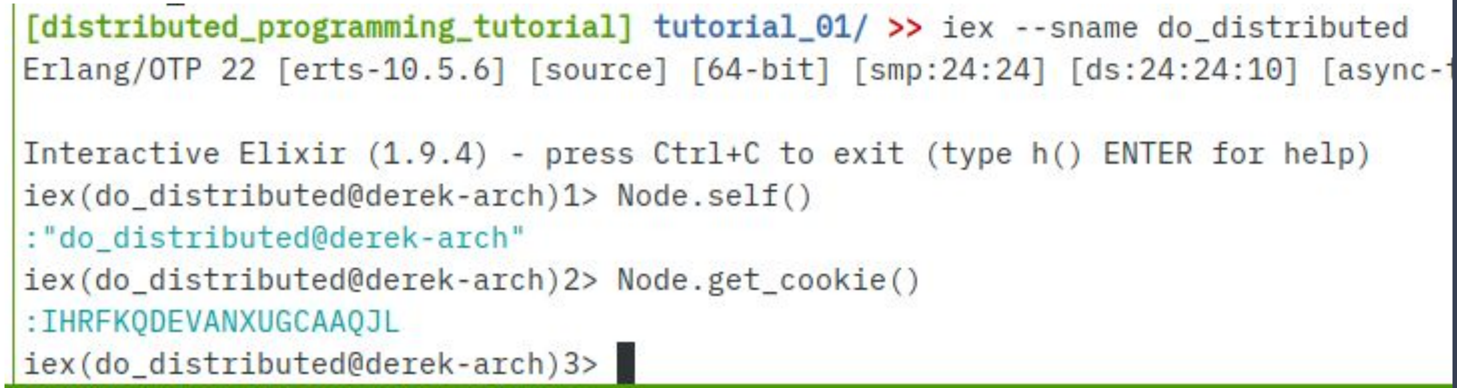
Node has functions related to BEAM nodes.



Anatomy of a node

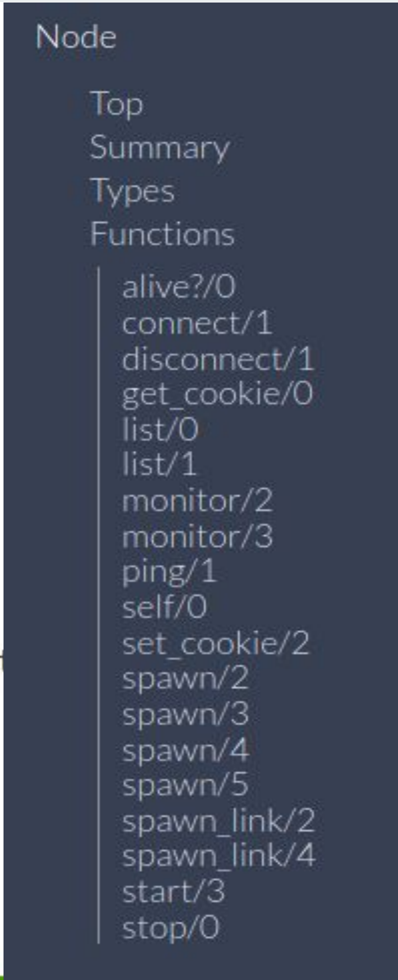
A node has a name and a cookie. Cookies must be the same for nodes to connect.

A cookie is not a password, you *must* ensure that you do not expose the port range used by epmd to set up erlang clustering to the internet.



```
[distributed_programming_tutorial] tutorial_01/ >> iex --sname do_distributed
Erlang/OTP 22 [erts-10.5.6] [source] [64-bit] [smp:24:24] [ds:24:24:10] [async-

Interactive Elixir (1.9.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(do_distributed@derek-arch)1> Node.self()
:"do_distributed@derek-arch"
iex(do_distributed@derek-arch)2> Node.get_cookie()
:IHRFKQDEVANXUGCAAQJL
iex(do_distributed@derek-arch)3>
```



Node

Top
Summary
Types
Functions

- alive?/0
- connect/1
- disconnect/1
- get_cookie/0
- list/0
- list/1
- monitor/2
- monitor/3
- ping/1
- self/0
- set_cookie/2
- spawn/2
- spawn/3
- spawn/4
- spawn/5
- spawn_link/2
- spawn_link/4
- start/3
- stop/0

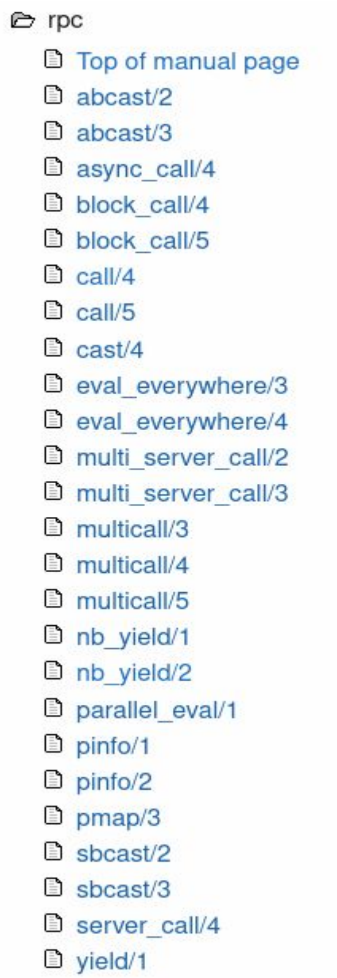
:rpc

Remote Procedure Calls - run code on a remote node

Example, :rpc.call/5

:rpc runs a process on each node in the cluster, which executes the function (and maybe returns the output)

Beware of bottlenecks!



rpc

- Top of manual page
- abcast/2
- abcast/3
- async_call/4
- block_call/4
- block_call/5
- call/4
- call/5
- cast/4
- eval_everywhere/3
- eval_everywhere/4
- multi_server_call/2
- multi_server_call/3
- multicall/3
- multicall/4
- multicall/5
- nb_yield/1
- nb_yield/2
- parallel_eval/1
- pinfo/1
- pinfo/2
- pmap/3
- sbcast/2
- sbcast/3
- server_call/4
- yield/1

Alternative to :rpc

Start your own process on each node and send messages to that process directly.

Addressing [remote] processes

There are a number of ways you can address a [remote] process:

- pid
- name *(local only, uses built-in process registry)*
- {name, node}
- {:global, name} *(uses :global)*
- {:via, module, name} *(via tuples are a GenServer construct)*

Addressing [remote] processes

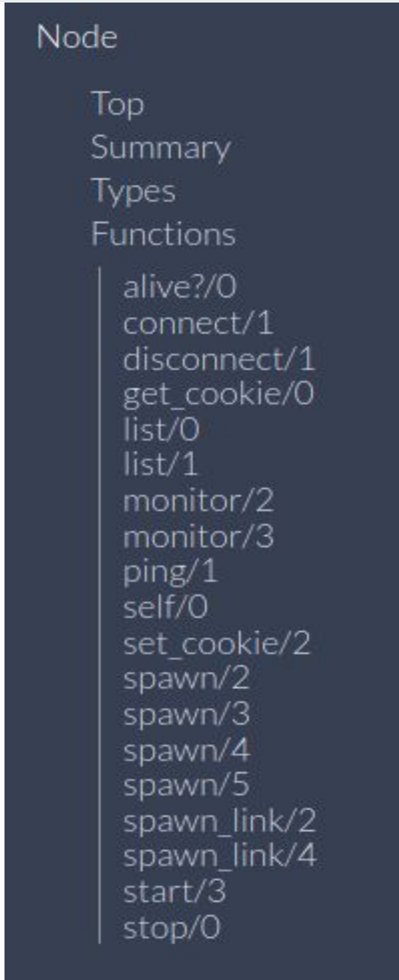
Via tuple: {:via, module, name}

GenServer expects *module* to implement *register_name/2*, *unregister_name/1*, *whereis_name/1* and *send/2*

We will talk about Registries later.

Connecting nodes

- Nodes must have the same cookie
- Nodes must have different names
- Node.connect/1 to connect
- Node.list/0 to list connected nodes


Node

Top
Summary
Types
Functions

alive?/0
connect/1
disconnect/1
get_cookie/0
list/0
list/1
monitor/2
monitor/3
ping/1
self/0
set_cookie/2
spawn/2
spawn/3
spawn/4
spawn/5
spawn_link/2
spawn_link/4
start/3
stop/0

Connecting nodes - libcluster

https://github.com/bitwalker/libcluster

Library to assist connecting your nodes together. Beats doing it all by hand.

Provides strategies:

- Epmd (static list of nodes)
- Gossip (dynamic membership based on a gossip protocol for discovery)
- Kubernetes (get node list from kubernetes, connect them all)
- Many others, and you can create your own easily.

MOOSE CODE

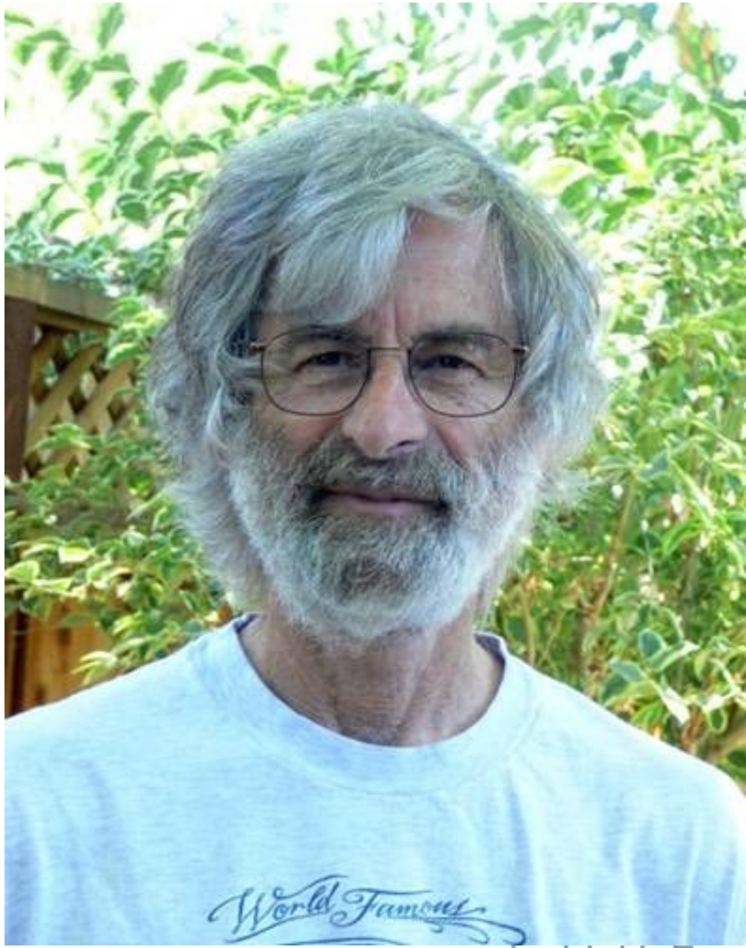
Exercises 1

MOOSE
CODE

Leslie Lamport

Inventor of:

- Lamport clock (vector clock)
- Paxos
- *Many* papers in distributed systems
- And he also developed LaTeX!



Lamport (vector) clock

Used to determine (partial) ordering of events in a distributed system.

Fundamental to many distributed algorithms.

Given N nodes, a vector clock is an n-tuple {a, b, … n} where a, b, etc, is a positive integer.

Each node has a single digit, only ever update your own digit.

Merge clocks: {max(a1, a2), max(b1, b2), … max(n1, n2)}

Can only be used to determine partial order (instead of absolute order)

Lamport (vector) clock

Rule: X < Y if all indices of X are <= equivalent indices in Y, and one index in X is < than equivalent index in Y

Examples:

{1, 2, 2} and {2, 1, 2} we can’t say anything about this

{1, 2, 2} and {2, 2, 2} we can say that {1, 2, 2} -> {2, 2, 2}

{1, 2, 1} and {4, 6, 4} we can say that {1, 2, 1} -> {4, 6, 4}

Paxos

Problem: We want to run a distributed algorithm. Problem: this gets very hard in the face of system failures.

We can invent our own (bad idea, probably miss many edge cases), or we can systematically design a system that does not miss these edge cases.

Enter Paxos: Introduced in 1989 by Leslie Lamport.

Paxos

- Safety (consistency) is guaranteed
- Can continue if a majority of nodes (50% + 1) are functioning
- Algorithm in very broad sense:
  - Algorithm is built around concept of a “log”
  - Any state machine can be described as a log of events
  - Leader election to decide on the “next value” in the log
  - Replicate “next value” to all members
  - Repeat, generating successive “next values” in the log
  - Sum of all events in the log determines the current state
- *Note: Paxos is CP*

MOOSE CODE

Raft

Successor to Paxos, intended to be simpler to understand

3 independent components:

- Leader election (decide next value)
- Log replication (only leader disseminates logs)
- Safety

MOOSE CODE

Conflict-free Replicated Data Types

CRDTs are:

- Eventually consistent
- Conflict-free (automatically resolving conflicts)
- Replicated
- Data Types

CRDTs - Eventually consistent

All updates are applied locally

Updates are then propagated to other nodes in the cluster

Propagation takes time = eventually consistent

CRDTs - Conflict-free

Conflict-free in the sense that conflicts are solved automatically

Vector clocks can be used to determine causality

When no causality can be determined, fall back to consistent merge function

CRDTs - Replicated

The contents of the CRDT are replicated across the cluster

CRDTs - Data Type

Fundamentally, a CRDT is a data type. Just another Data Structure with its own set of Algorithms.

CRDTs - Algorithm

1. Set own initial state
2. Send state to random neighbour
3. If received state from neighbour, merge with own state
4. Change own state [optional]
5. Repeat from step 2

CRDTs - Hello World CRDT

State: a single bit

Initial state is 0

Merge function is “OR”

OR		
	0	1
0	0	1
1	1	1

Demonstration

✊ = 0

☝ = 1

MOOSE CODE

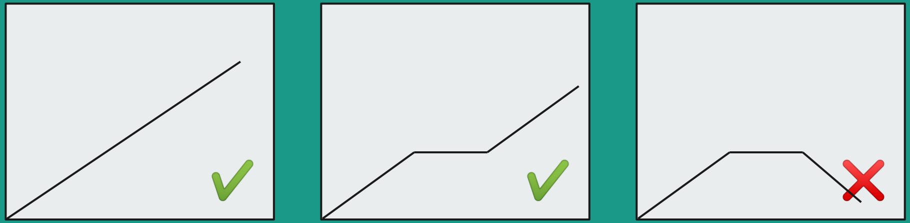
Properties of Merge Function

Must be:

-	Commutative	$a \bullet b = b \bullet a$
-	Associative	$(a \bullet b) \bullet c = a \bullet (b \bullet c)$
-	Idempotent	$a \bullet b = a \bullet b \bullet b$
-	Monotonic	0 -> 1 ✓ 1 -> 0 ❌

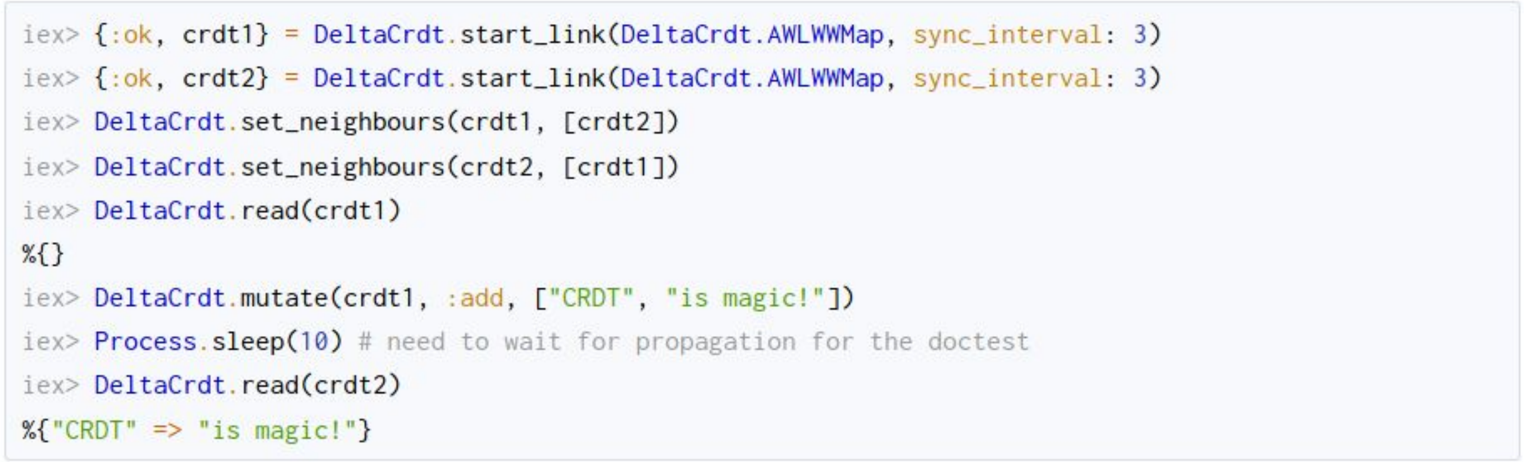
MOOSE CODE

Monotonicity



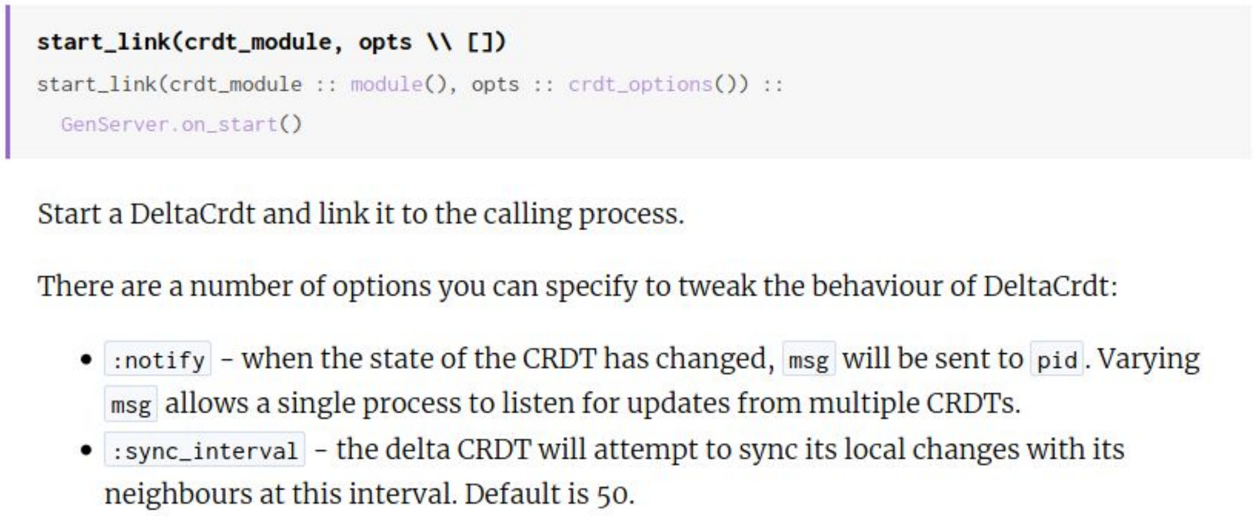
DeltaCrdt

Library written to support Horde, but useful in its own right.



```
iex> {:ok, crdt1} = DeltaCrdt.start_link(DeltaCrdt.AWLWWMap, sync_interval: 3)
iex> {:ok, crdt2} = DeltaCrdt.start_link(DeltaCrdt.AWLWWMap, sync_interval: 3)
iex> DeltaCrdt.set_neighbours(crdt1, [crdt2])
iex> DeltaCrdt.set_neighbours(crdt2, [crdt1])
iex> DeltaCrdt.read(crdt1)
%{}
iex> DeltaCrdt.mutate(crdt1, :add, ["CRDT", "is magic!"])
iex> Process.sleep(10) # need to wait for propagation for the doctest
iex> DeltaCrdt.read(crdt2)
%{"CRDT" => "is magic!"}
```

DeltaCrdt - start_link



```
start_link(crdt_module, opts \\ [])

start_link(crdt_module :: module(), opts :: crdt_options()) ::
  GenServer.on_start()
```

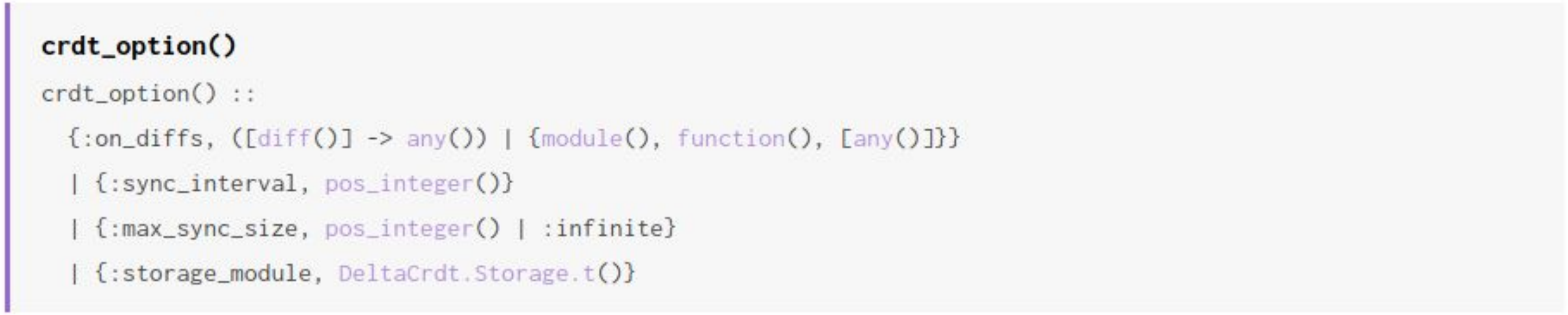
Start a DeltaCrdt and link it to the calling process.

There are a number of options you can specify to tweak the behaviour of DeltaCrdt:

- `:notify` - when the state of the CRDT has changed, `msg` will be sent to `pid`. Varying `msg` allows a single process to listen for updates from multiple CRDTs.
- `:sync_interval` - the delta CRDT will attempt to sync its local changes with its neighbours at this interval. Default is 50.

MOOSE CODE

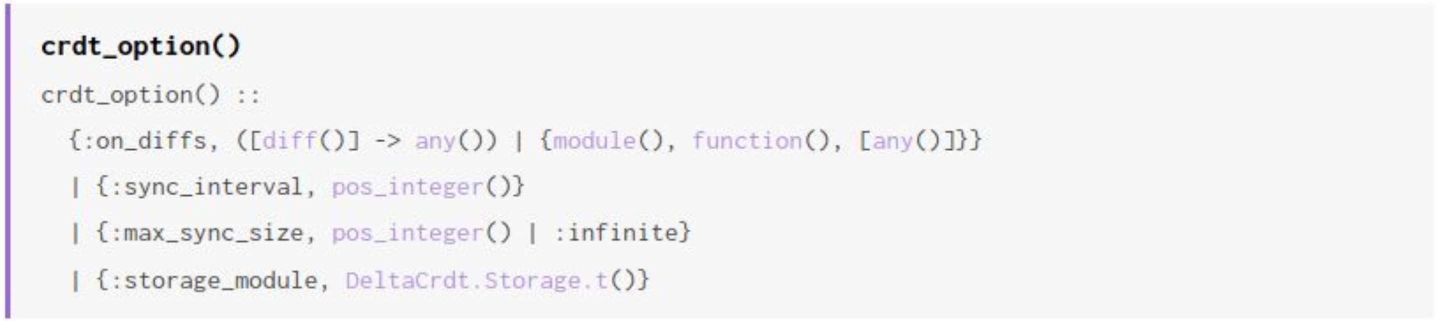
DeltaCrdt - crdt_options



crdt_option()

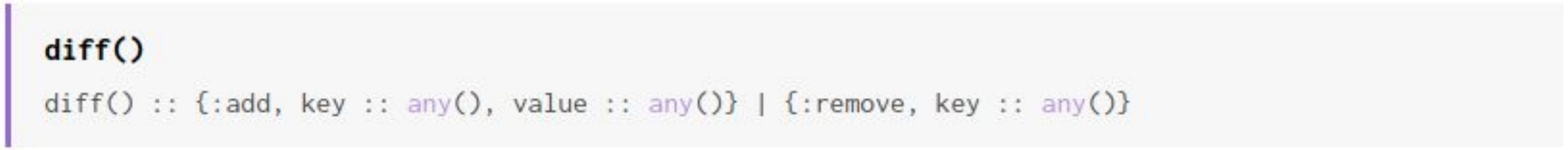
```
crdt_option() ::
  {:on_diffs, ([diff()] -> any()) | {module(), function(), [any()]}}
  | {:sync_interval, pos_integer()}
  | {:max_sync_size, pos_integer() | :infinite}
  | {:storage_module, DeltaCrdt.Storage.t()}
```

DeltaCrdt - on_diffs



crdt_option()

```
crdt_option() ::
  {:on_diffs, ([diff()] -> any()) | {module(), function(), [any()]}}
  | {:sync_interval, pos_integer()}
  | {:max_sync_size, pos_integer() | :infinite}
  | {:storage_module, DeltaCrdt.Storage.t()}
```



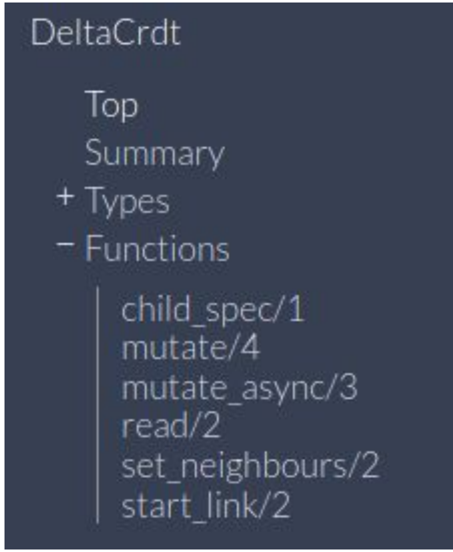
diff()

```
diff() :: {:add, key :: any(), value :: any()} | {:remove, key :: any()}
```

DeltaCrdt - set_neighbours/2

Synchronization is unidirectional

Using set_neighbours/2 you are free to implement any topology you want, tree, mesh, ring, etc.

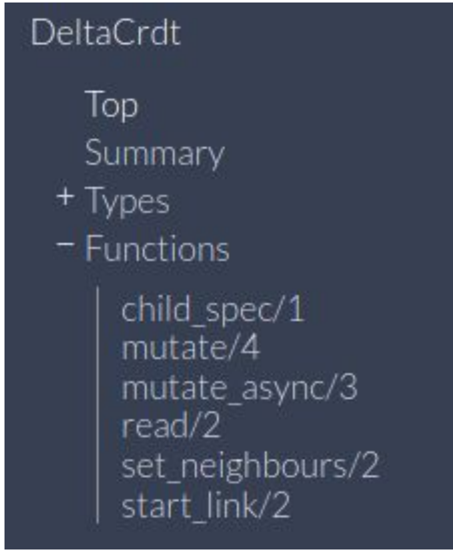


MOOSE CODE

DeltaCrdt - mutate/4

Arguments follow the familiar {module, function, argument} pattern. Module is specified in start_link/2.

Async version mutate_async/3.


DeltaCrdt

Top
Summary
+ Types
− Functions

child_spec/1
mutate/4
mutate_async/3
read/2
set_neighbours/2
start_link/2

DeltaCrdt - read/2

Returns the contents of the CRDT.

Exercises 2

Supervision

Classic erlang supervision:

- Supervision tree
- Uses *Process.link/1*
- When a process dies, restart it

Distributed Supervision

- Supervision tree?
- Uses *Process.link/1* works in a cluster!
- When a process dies, restart it!
- Pitfall: a process has more than one supervisor. 🤔
- Failure modes:
    - Process dies
    - Node dies
    - Node is unreachable (and may or may not return)
- We need to consider *all* of this when searching for a solution.

Distributed Supervision

- Libraries:
  - Horde.DynamicSupervisor (for many processes, uses DeltaCrdt)
  - Highlander (for just one process, uses :global)
  - Swarm

Distributed Supervision

Hot failover: Two nodes; one supervises the other, when the node fails, take over its processes.

Example: Postgres

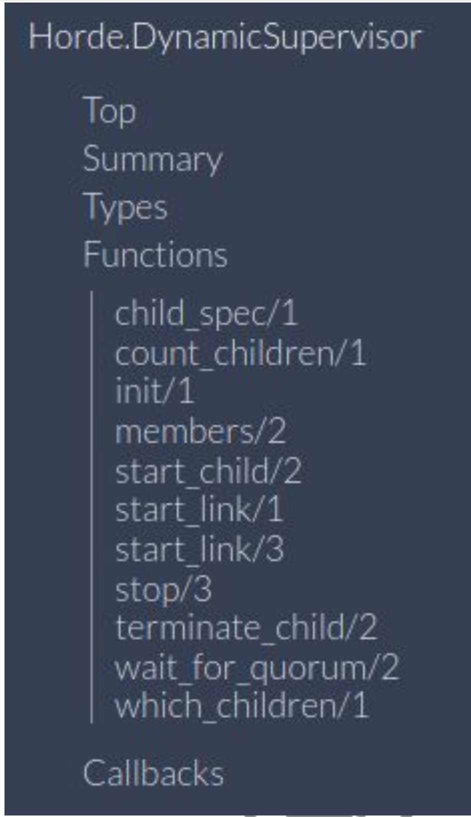
Horde

Horde is a library that implements a distributed DynamicSupervisor, and a distributed Registry

Horde.DynamicSupervisor

Implements the same API as Elixir.DynamicSupervisor, with some additions.

- Horde.DynamicSupervisor.wait_for_quorum/2


Horde.DynamicSupervisor

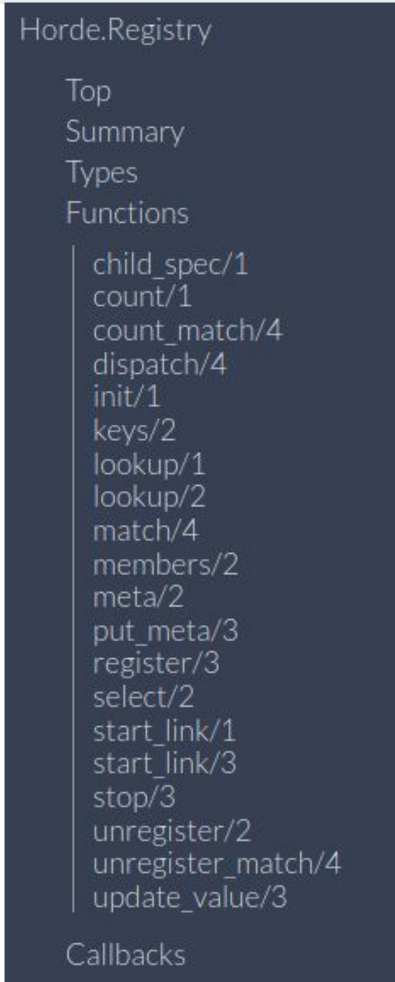
Top
Summary
Types
Functions
- child_spec/1
- count_children/1
- init/1
- members/2
- start_child/2
- start_link/1
- start_link/3
- stop/3
- terminate_child/2
- wait_for_quorum/2
- which_children/1

Callbacks

MOOSE CODE

Horde.Registry

Implements the same API as Elixir.Registry (exception: duplicate key registry).



Horde.Registry

Top
Summary
Types
Functions

child_spec/1
count/1
count_match/4
dispatch/4
init/1
keys/2
lookup/1
lookup/2
match/4
members/2
meta/2
put_meta/3
register/3
select/2
start_link/1
start_link/3
stop/3
unregister/2
unregister_match/4
update_value/3

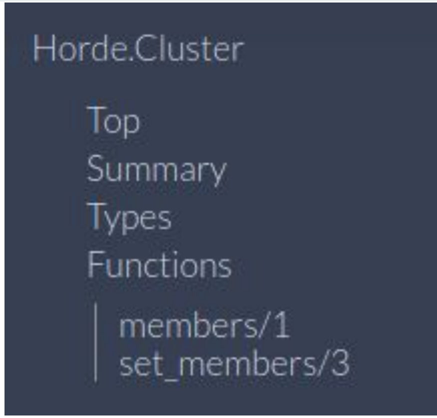
Callbacks

Horde.Cluster

Provides convenience functions to manage the members of the cluster.

Horde support dynamic cluster membership, but *you* must tell it which nodes are in the cluster.

Can also set members in init/1 callback of Horde.DynamicSupervisor and Horde.Registry.


Horde.Cluster

Top
Summary
Types
Functions

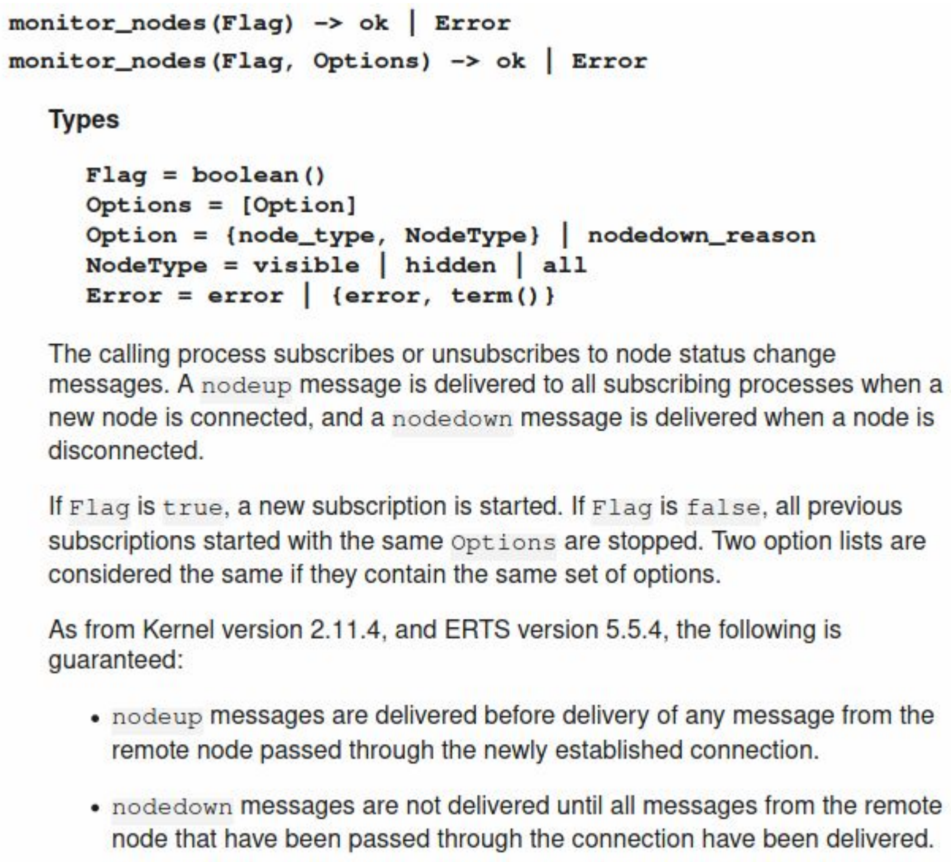
members/1
set_members/3

:net_kernel

Listening for changes in node membership, simple solution for maintaining Horde membership list.

Ideally you want something more authoritative.

Example: list of nodes in cluster from AWS



```
monitor_nodes(Flag) -> ok | Error
monitor_nodes(Flag, Options) -> ok | Error
```

Types

```
Flag = boolean()
Options = [Option]
Option = {node_type, NodeType} | nodedown_reason
NodeType = visible | hidden | all
Error = error | {error, term()}
```

The calling process subscribes or unsubscribes to node status change messages. A `nodeup` message is delivered to all subscribing processes when a new node is connected, and a `nodedown` message is delivered when a node is disconnected.

If `Flag` is `true`, a new subscription is started. If `Flag` is `false`, all previous subscriptions started with the same `Options` are stopped. Two option lists are considered the same if they contain the same set of options.

As from Kernel version 2.11.4, and ERTS version 5.5.4, the following is guaranteed:

- `nodeup` messages are delivered before delivery of any message from the remote node passed through the newly established connection.
- `nodedown` messages are not delivered until all messages from the remote node that have been passed through the connection have been delivered.

Exercises 3

MOOSE
CODE

Network partitions

Strong CAP: C, A, P, choose 2

Weak CAP: C, A, P, make trade-offs between CP and AP, end up somewhere in between

Example: Quorum.

What happens during a network partition?

Option 1: the system grinds to a halt

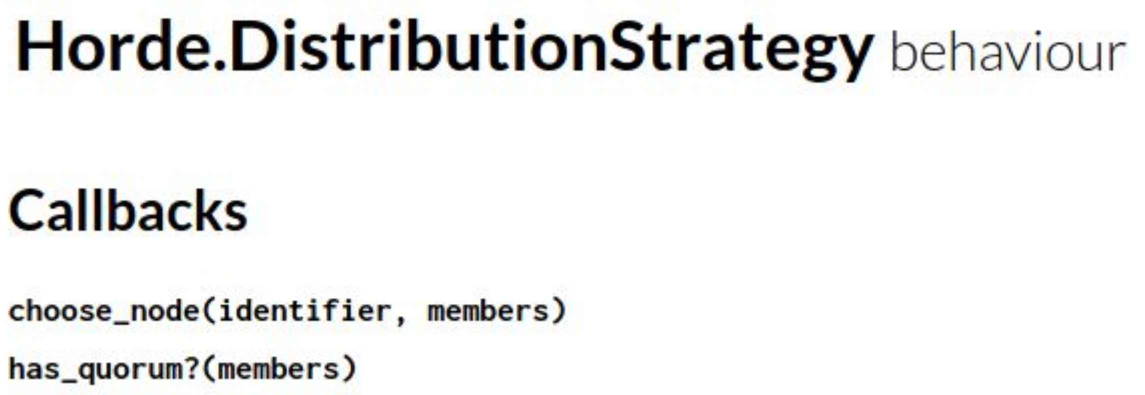
Option 2: there are two versions of the truth (split-brain)

Quorum in Horde

Horde includes two of these:

Horde.UniformDistribution

Horde.UniformQuorumDistribution



Horde.DistributionStrategy behaviour

Callbacks

```
choose_node(identifier, members)

has_quorum?(members)
```

Netsplit - Horde.DynamicSupervisor

Horde must assume that the other node is dead, therefore all processes on that node are dead.

Behaviour: restart all “dead” processes on other nodes.

BUT! “clone” them.

WHY? When netsplit resolves, we can’t have both Horde.DynamicSupervisor and Horde.Registry deciding on uniqueness in the cluster, will end up with weird results.

Netsplit - Horde.Registry

Horde.Registry allows a new registration if it can’t determine that the old registration is still alive (example: node of the process is unreachable).

When netsplit resolves, merge function will decide which processes survive.

Send `:EXIT` message to processes that were not chosen!

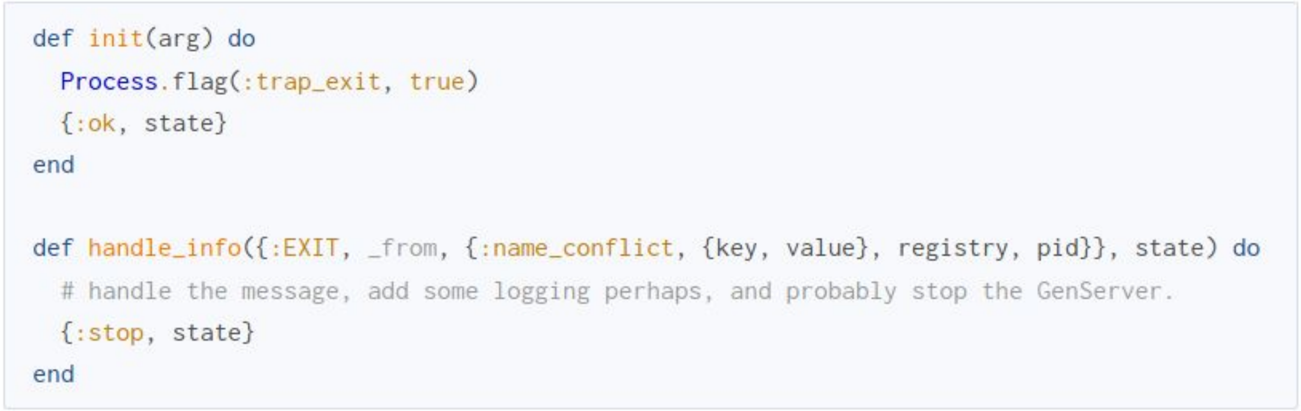
Netsplit - Horde

This can be tricky to get right, that’s why there is a guide written about this in the Horde documentation.

State handoff

Horde.Registry decides which processes survive, how to merge state?

- Trap exits
- Implement terminate/2 callback
- Implement handle_info/2 callback



```
def init(arg) do
  Process.flag(:trap_exit, true)
  {:ok, state}
end

def handle_info({:EXIT, _from, {:name_conflict, {key, value}, registry, pid}}, state) do
  # handle the message, add some logging perhaps, and probably stop the GenServer.
  {:stop, state}
end
```

Where to save data from state handoff?

On termination -> delta_crdt or another database

On merge conflict -> send new state directly to surviving process

No guarantee either of these will succeed!

Exercises 4

Papers We Love

Bonus content if time allows.

Time, Clocks, and the Ordering of Events in a Distributed System