Marlon - A Domain-Specific Language for Multi-Agent Reinforcement Learning on Networks

Tim Molderez, Bjarno Oeyen, Coen De Roover & Wolfgang De Meuter

Software Languages Lab

Vrije Universiteit Brussel

Context: distributed systems

Context: distributed systems

Smart grids

Context: distributed systems

Smart grids

Intelligent traffic systems

Context: distributed systems

Smart grids

Intelligent traffic systems

Cloud services

Problem statement

Problem statement

- Manually managing such large systems is difficult

Problem statement

- Manually managing such large systems is difficult
- Optimizing throughput, reliability, latency, resource usage, ...

Problem statement

- Manually managing such large systems is difficult
- Optimizing throughput, reliability, latency, resource usage, ...
- Automate with multi-agent RL (MARL)

Problem statement

- Manually managing such large systems is difficult
- Optimizing throughput, reliability, latency, resource usage, ...
- Automate with multi-agent RL (MARL)
- Problems inherent to distributed systems:
  - Networks are dynamic
  - Nodes, connections can fail or be unreliable
  - Communication cost

Marlon

Marlon

Multi-**A**gent **R**einforcement **L**earning **O**n **N**etworks

Marlon

Multi-**A**gent **R**einforcement **L**earning **O**n **N**etworks

- Domain-specific programming language to:
  - implement network environment
  - plug in existing MARL algorithms into this environment

Marlon

Multi-**A**gent **R**einforcement **L**earning **O**n **N**etworks

- Domain-specific programming language to:
  - implement network environment
  - plug in existing MARL algorithms into this environment
- Enable domain experts to use MARL with little background knowledge

Marlon

Multi-**A**gent **R**einforcement **L**earning **O**n **N**etworks

- Domain-specific programming language to:
  - implement network environment
  - plug in existing MARL algorithms into this environment
- Enable domain experts to use MARL with little background knowledge
- MARL researchers can focus on MARL, rather than the intricacies of distributed systems

Marlon

Marlon

- Implemented on top of the elixir language
  - Designed for scalable, fault-tolerant applications

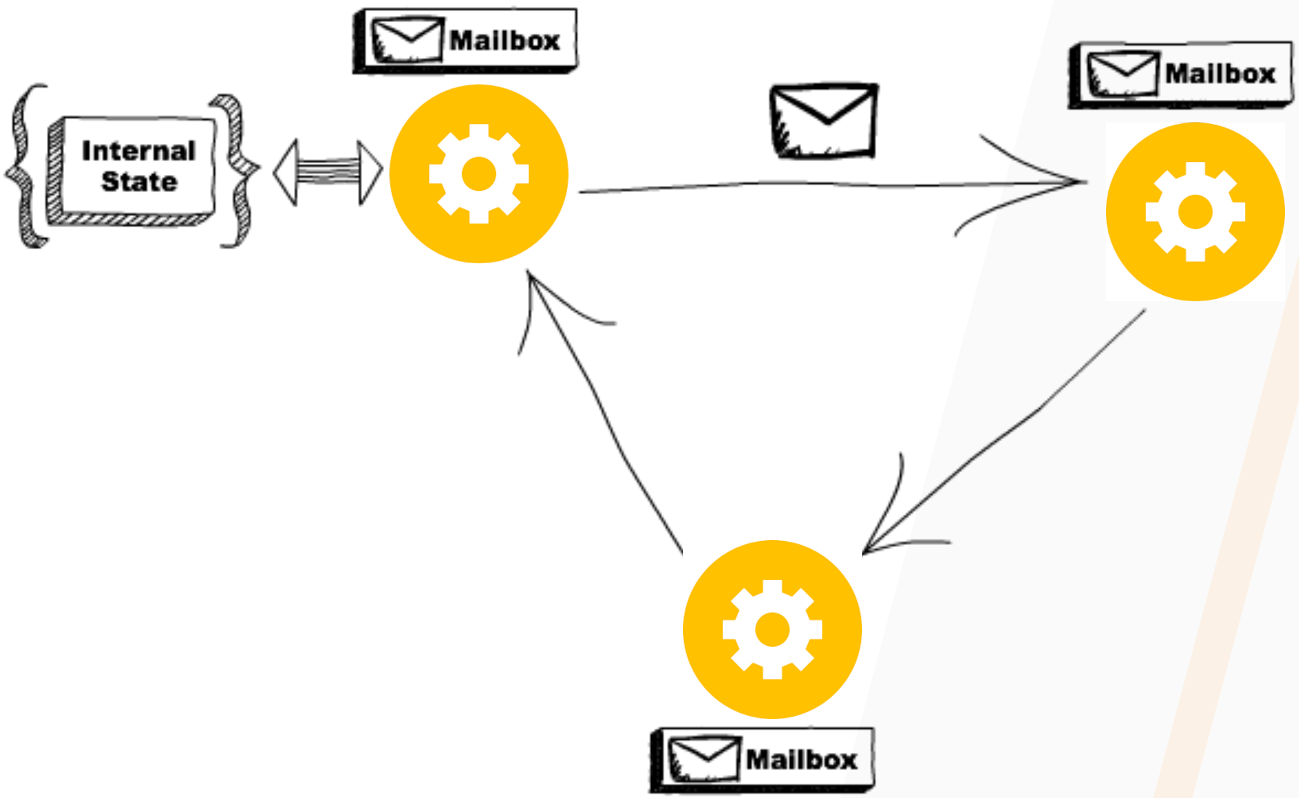
Marlon

- Implemented on top of the elixir language
  - Designed for scalable, fault-tolerant applications
- python™ integration to use existing MARL algos.

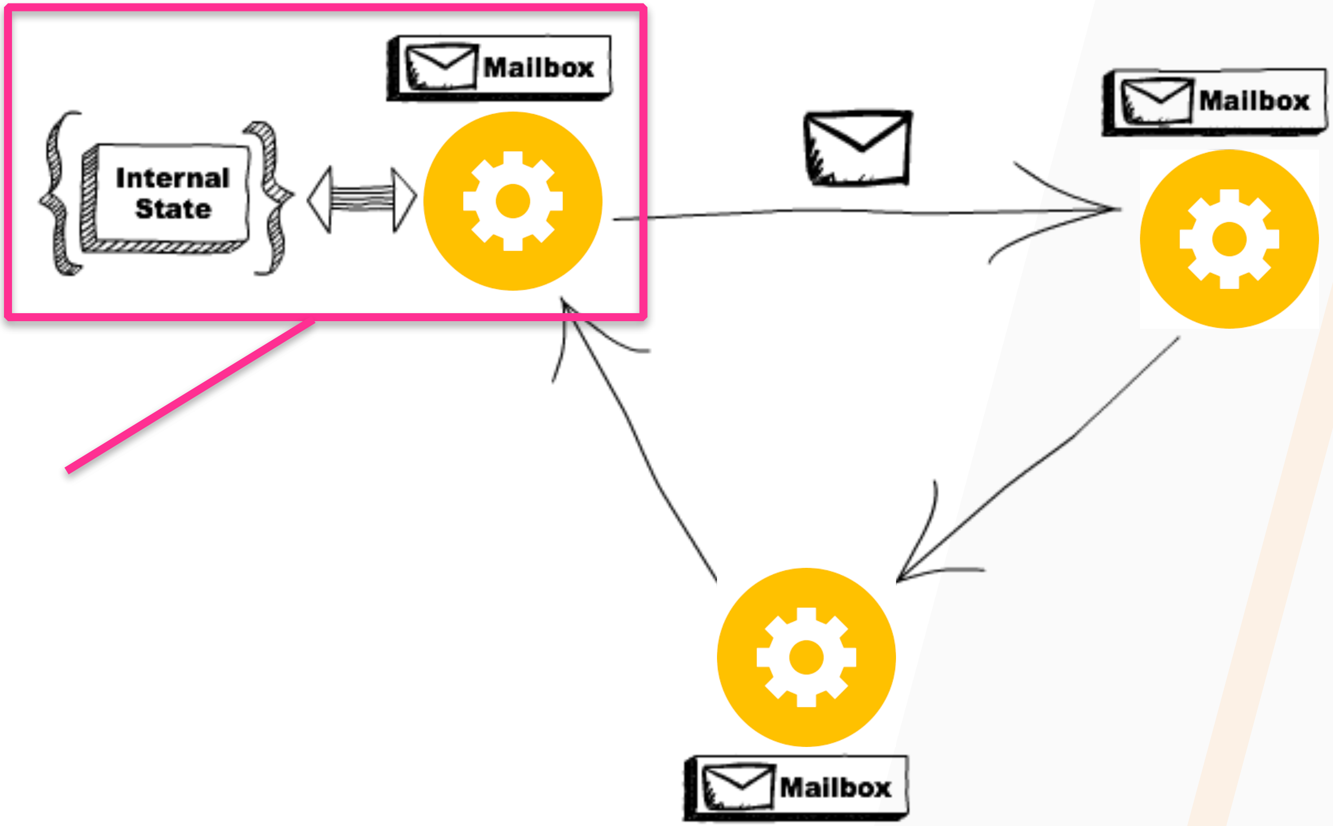
Marlon

- Implemented on top of the elixir language
    - Designed for scalable, fault-tolerant applications
- python™ integration to use existing MARL algos.
- Two main concepts: actors and agents
    - Environment represented as an actor-based system
    - Agents observe, interact and learn from the environment

Actor-based concurrency

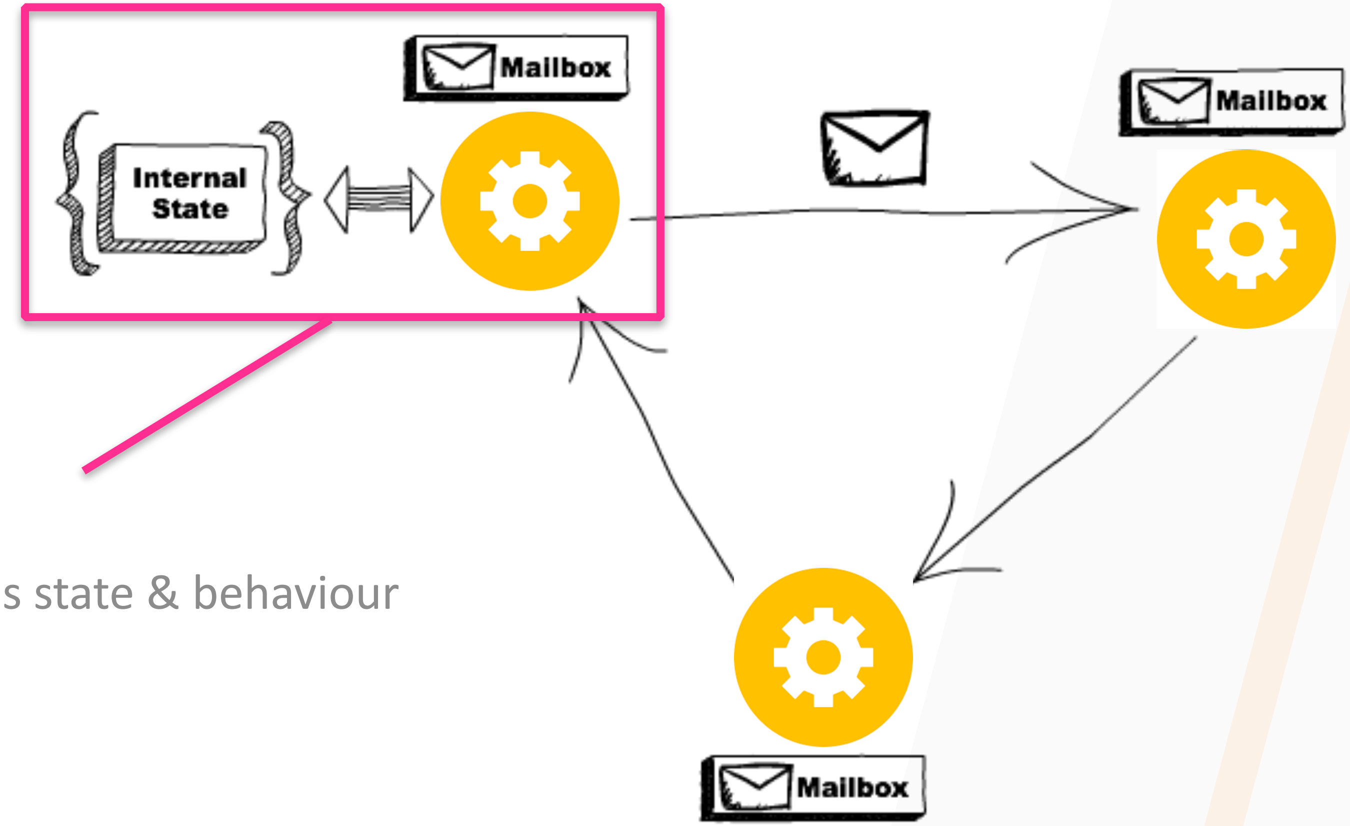


Actor-based concurrency



An actor

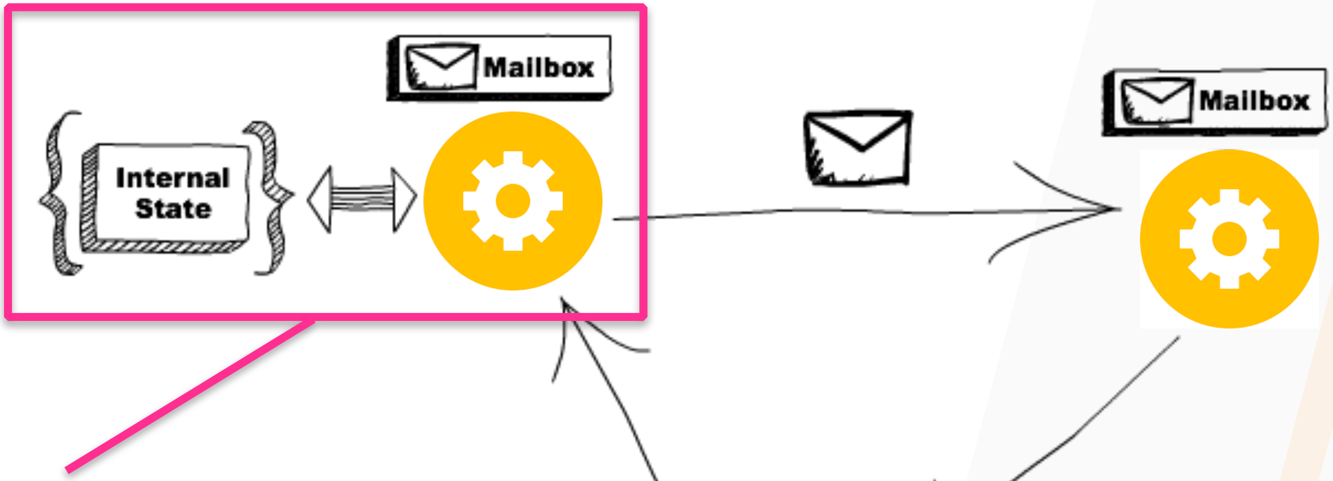
Actor-based concurrency

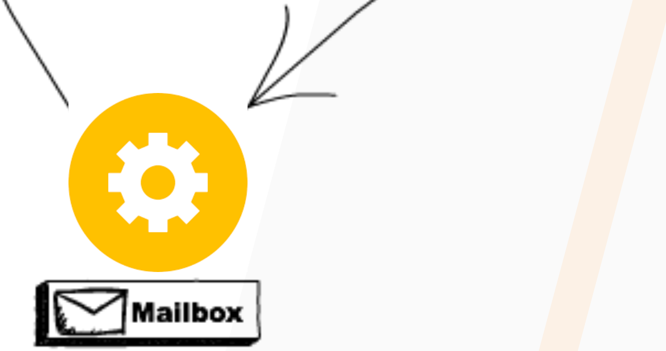


An actor

- Contains state & behaviour

Actor-based concurrency

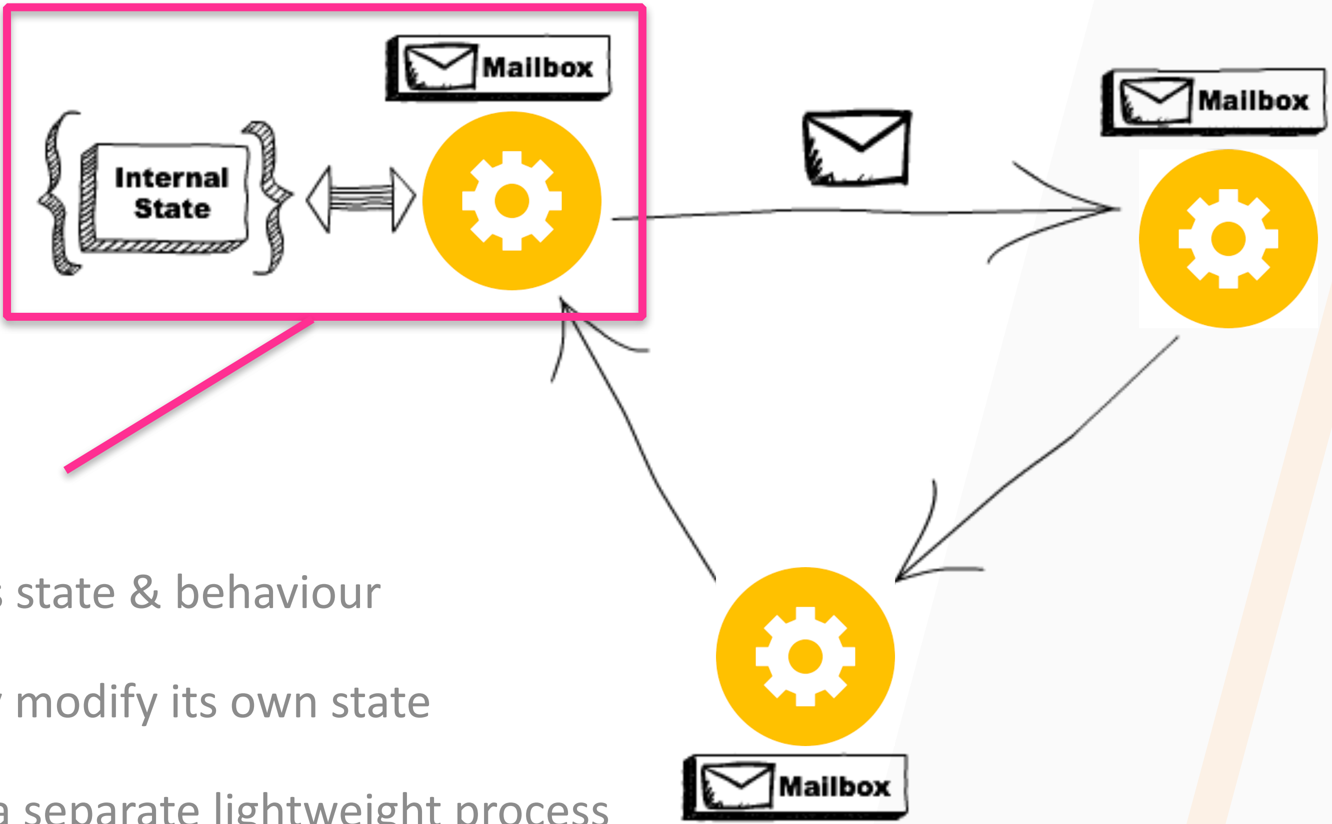




An actor

- Contains state & behaviour
- Can only modify its own state

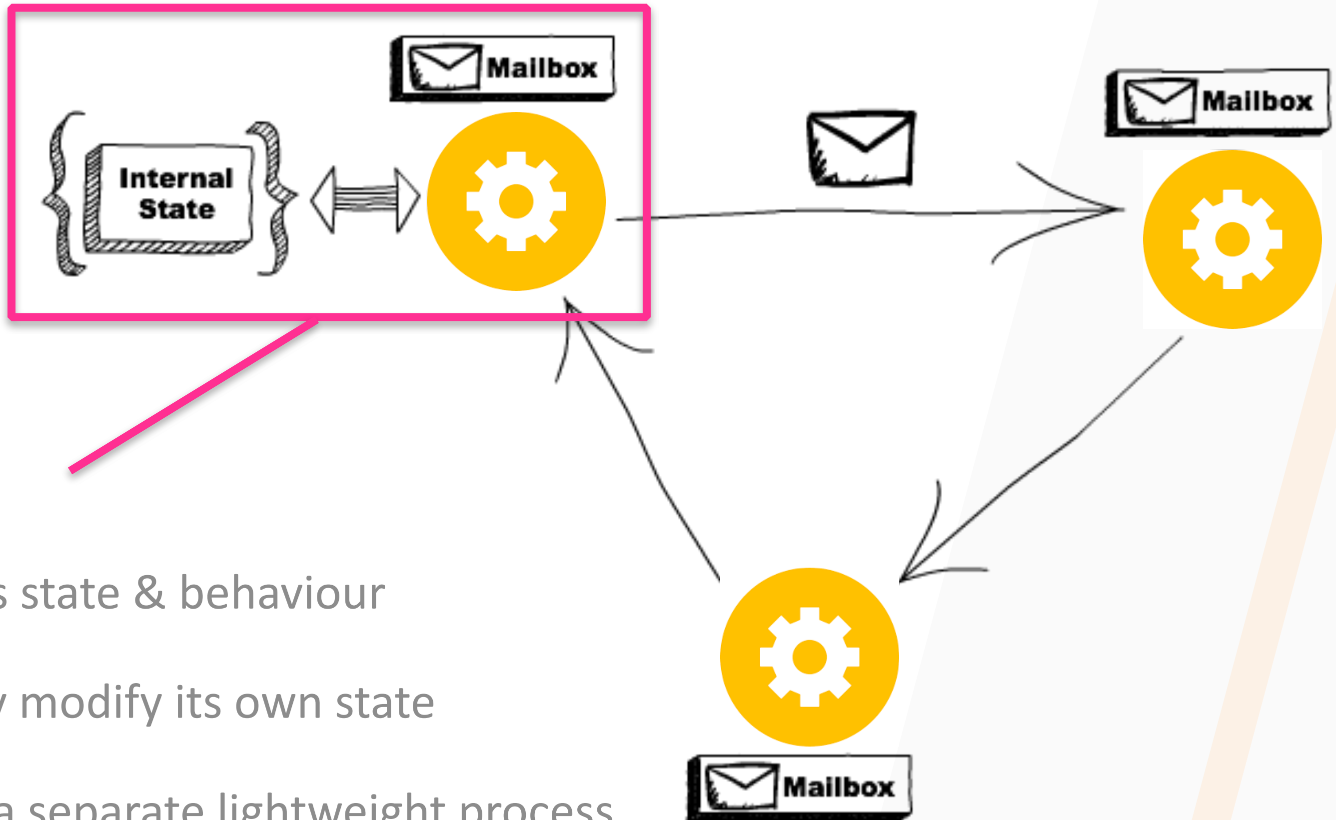
Actor-based concurrency



An actor

- Contains state & behaviour
- Can only modify its own state
- Runs in a separate lightweight process

Actor-based concurrency

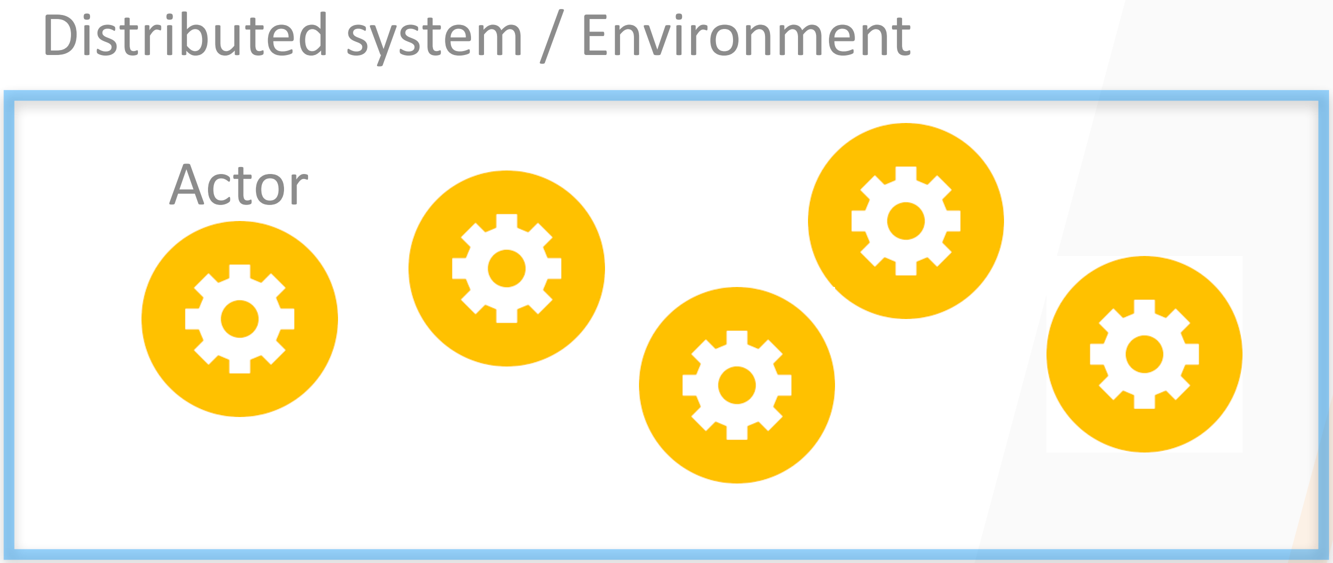


An actor

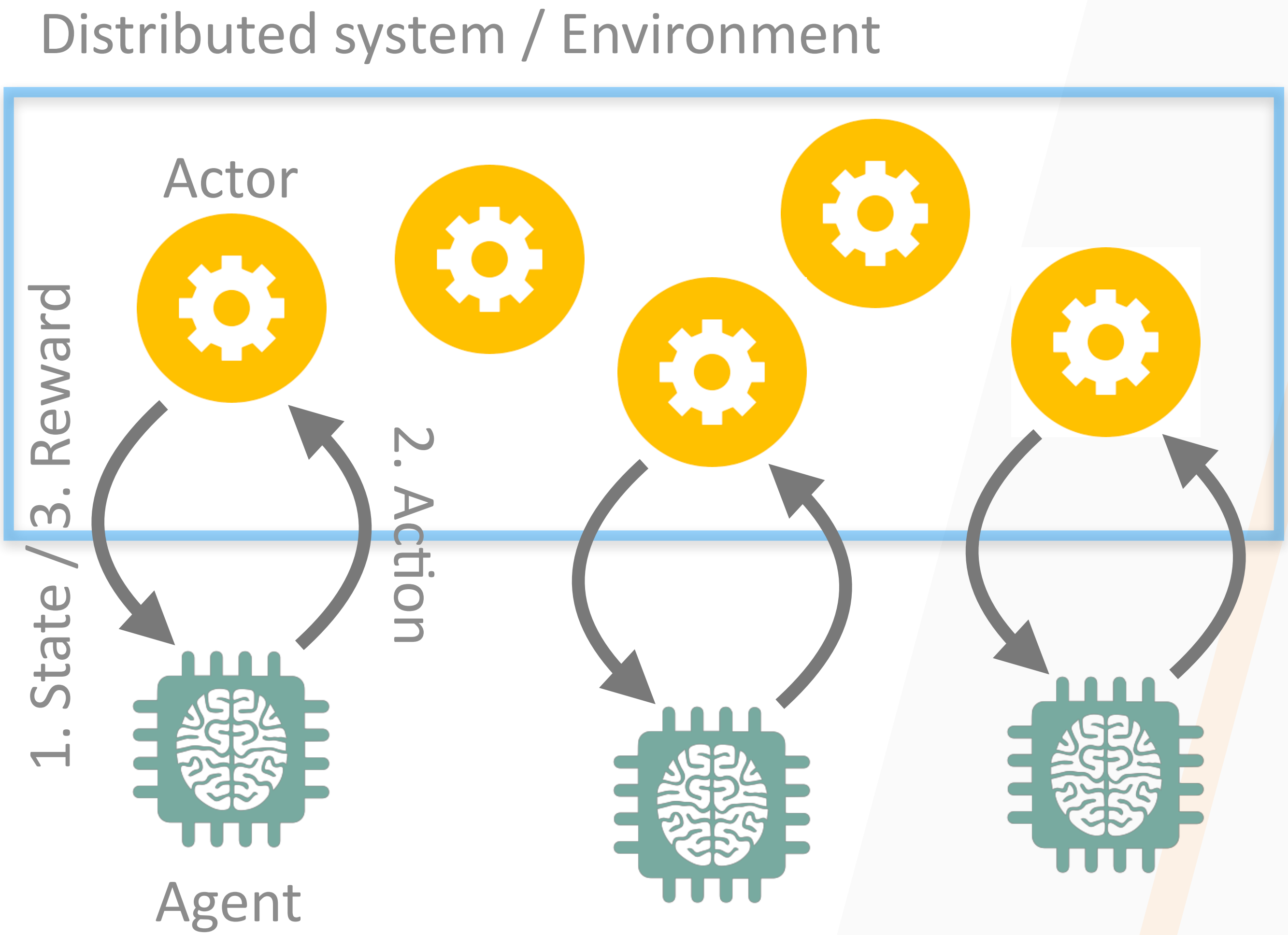
- Contains state & behaviour
- Can only modify its own state
- Runs in a separate lightweight process
- Communicates by sending messages

Marlon application overview

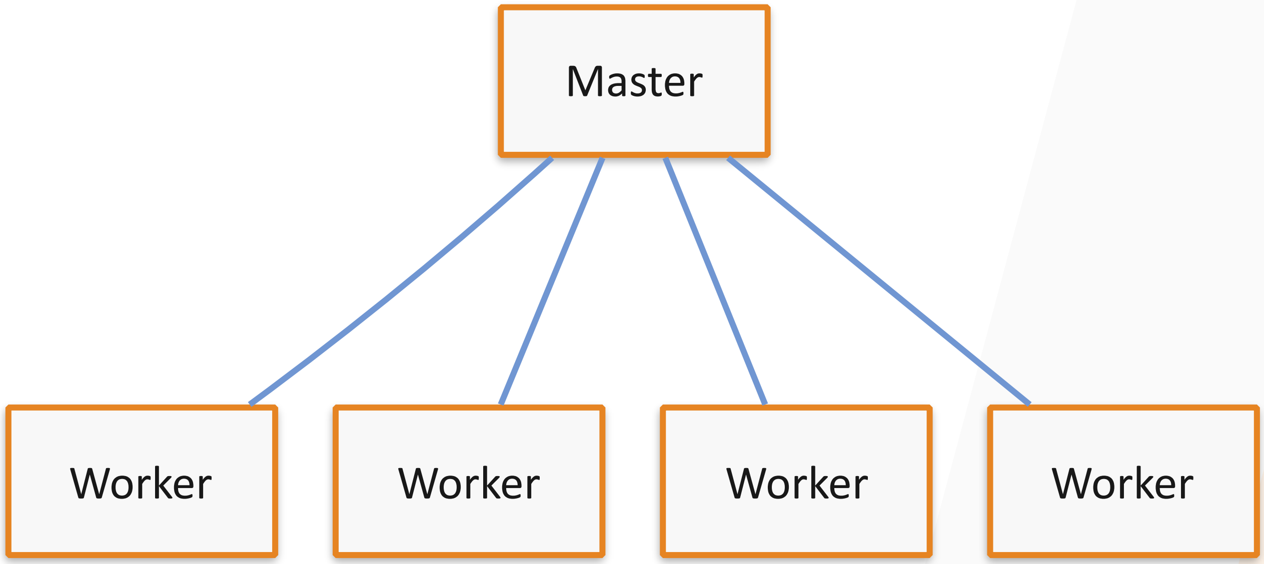
Marlon application overview



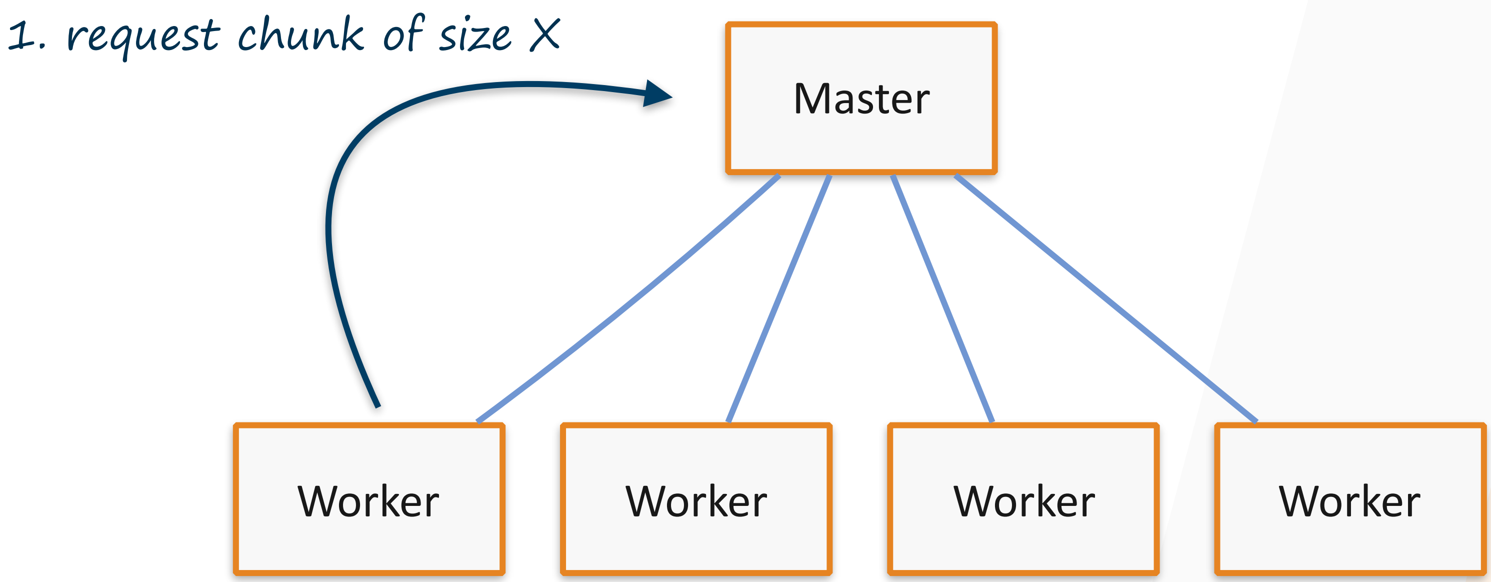
Marlon application overview



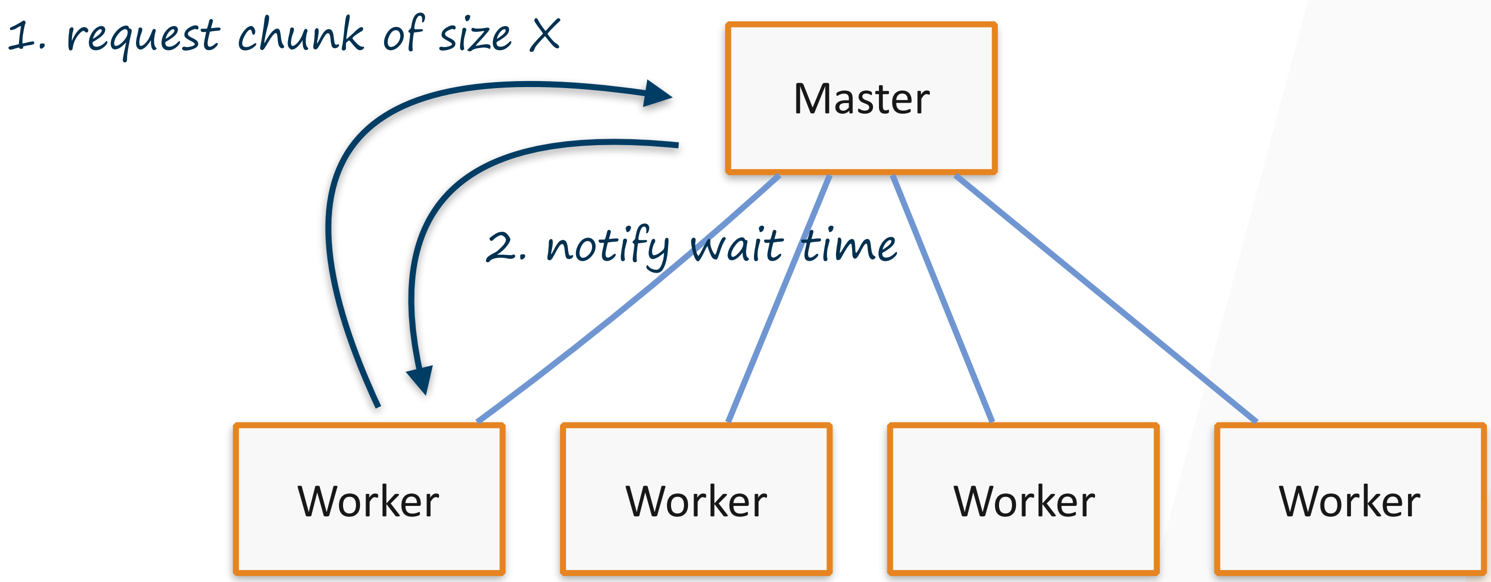
Load balancing example



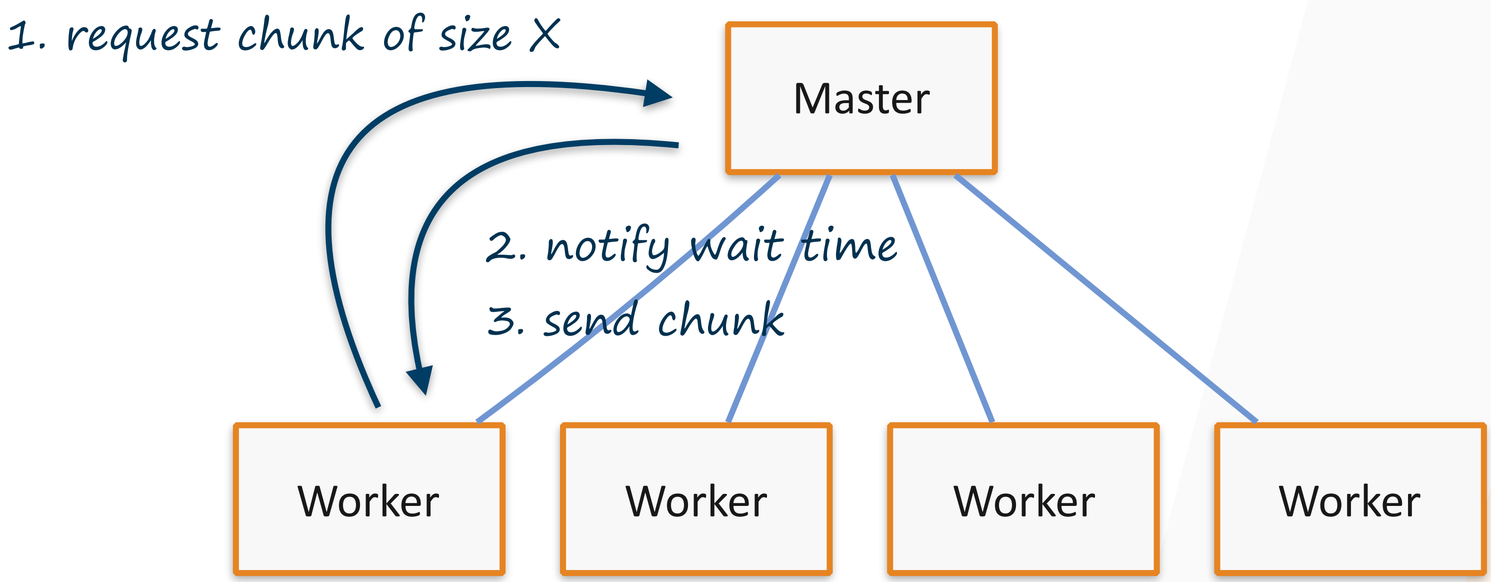
Load balancing example



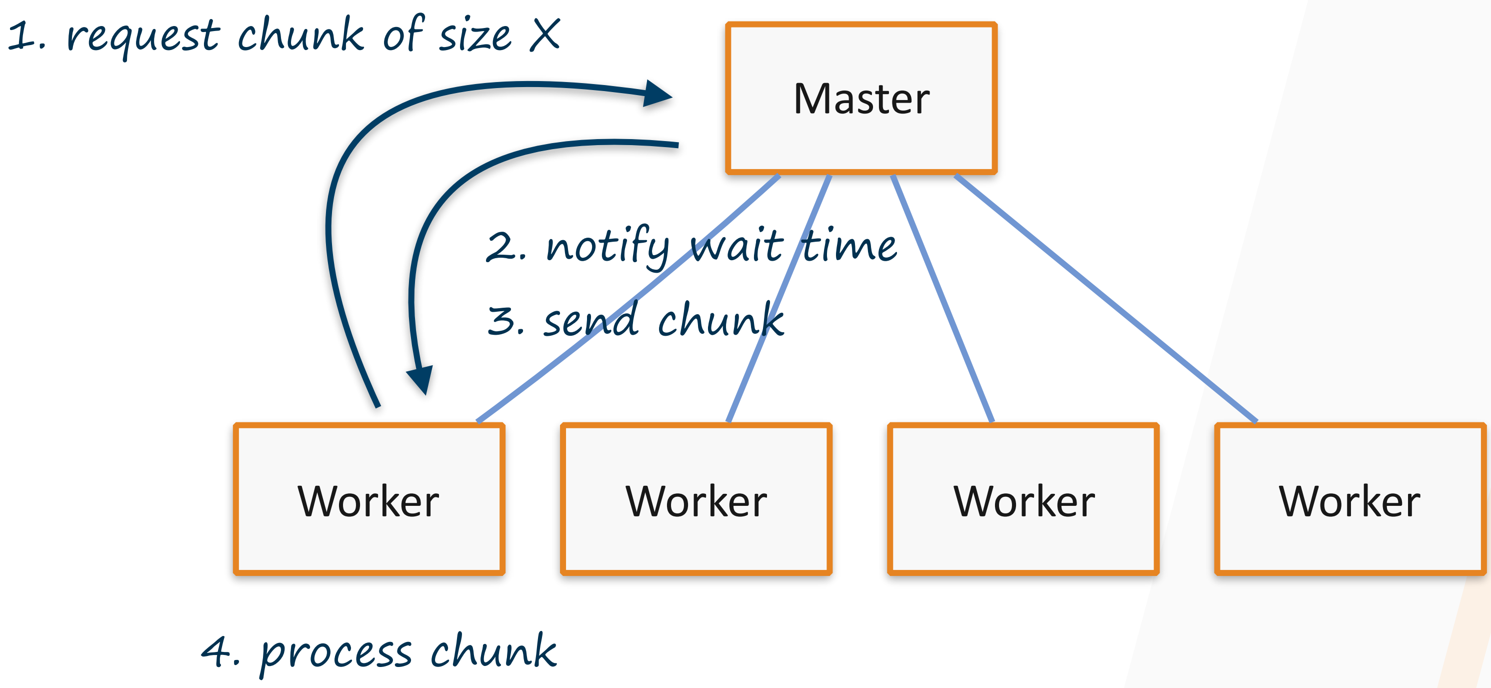
Load balancing example



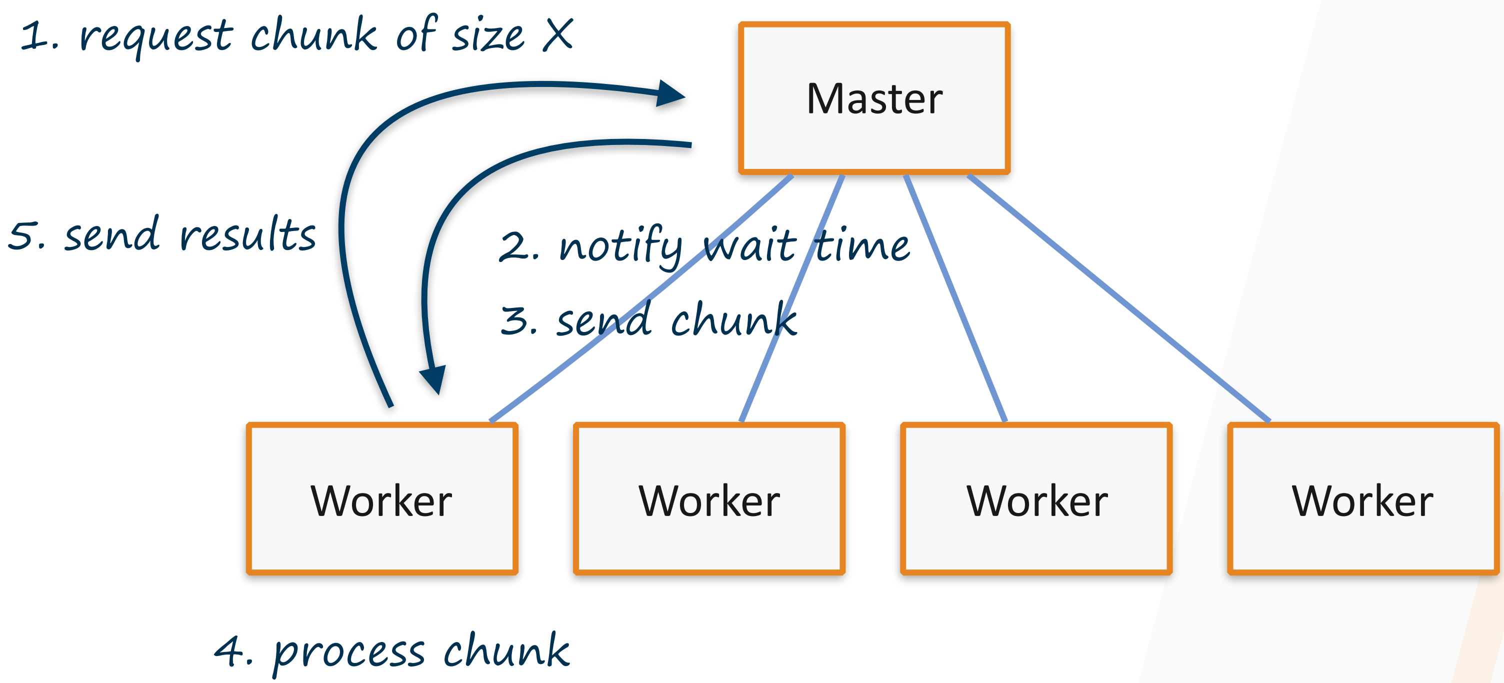
Load balancing example



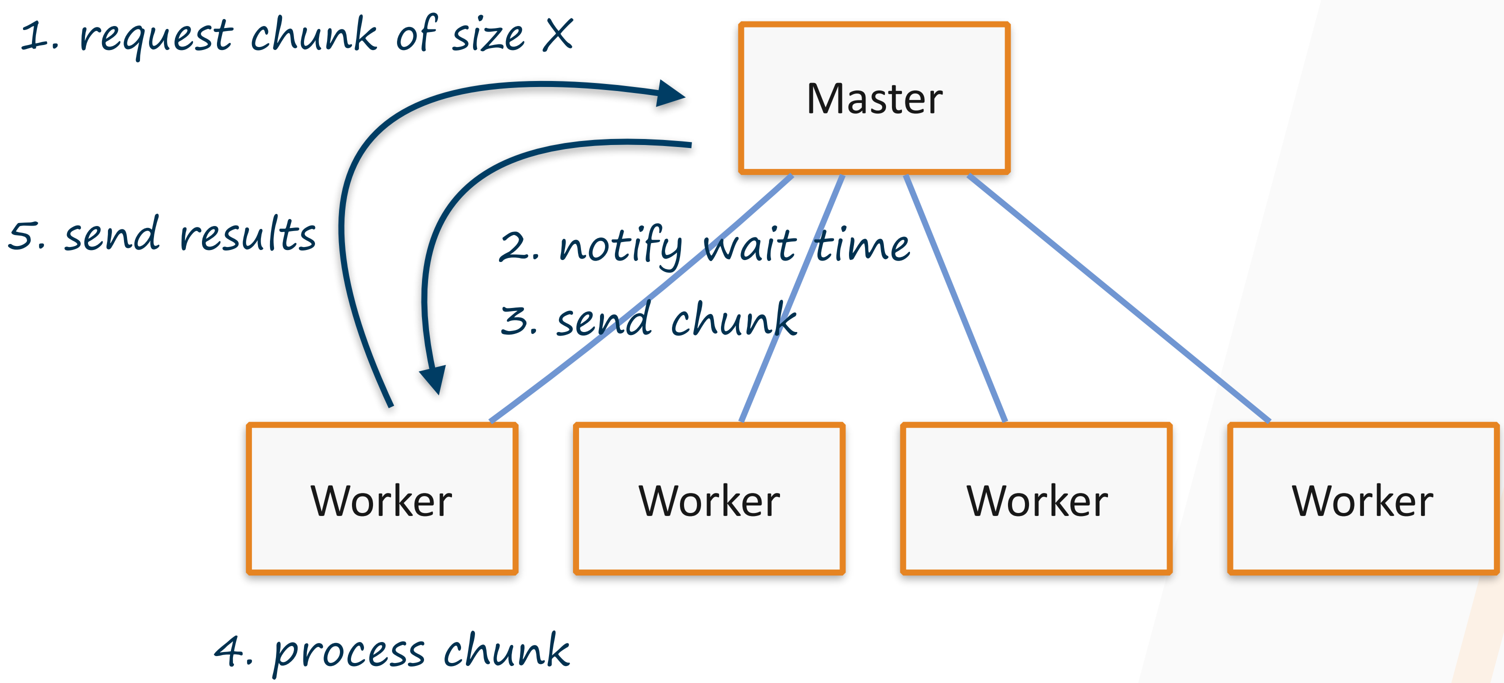
Load balancing example



Load balancing example

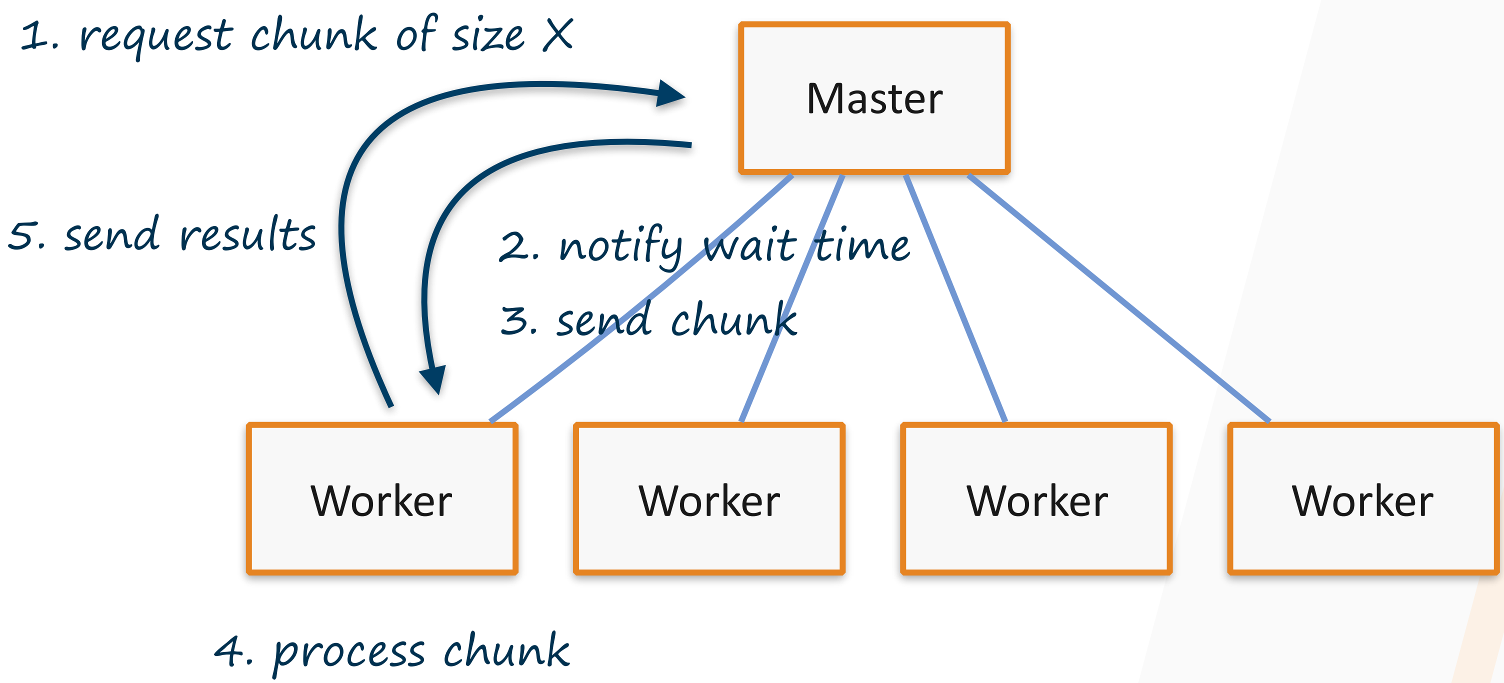


Load balancing example



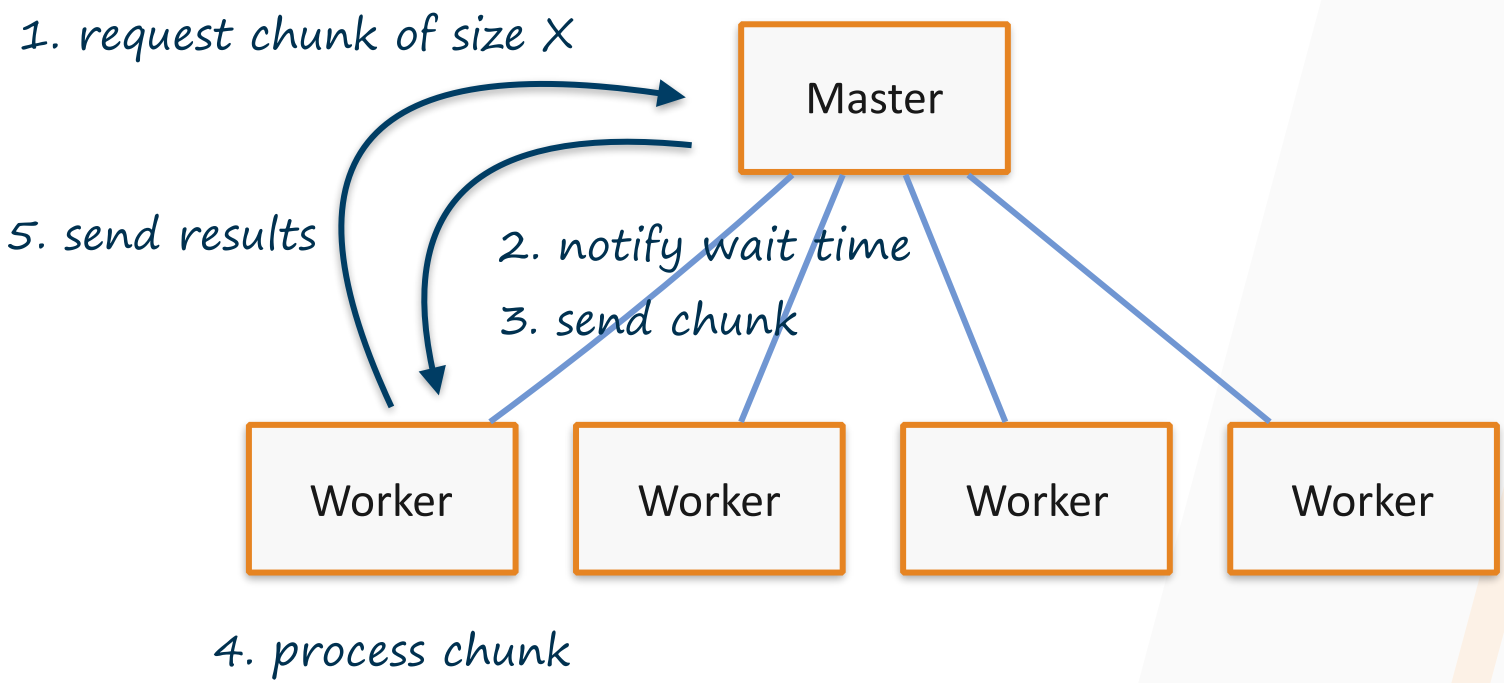
- Master can only handle one request at a time

Load balancing example



- Master can only handle one request at a time
- Workers can join/leave; they can have different processing speeds

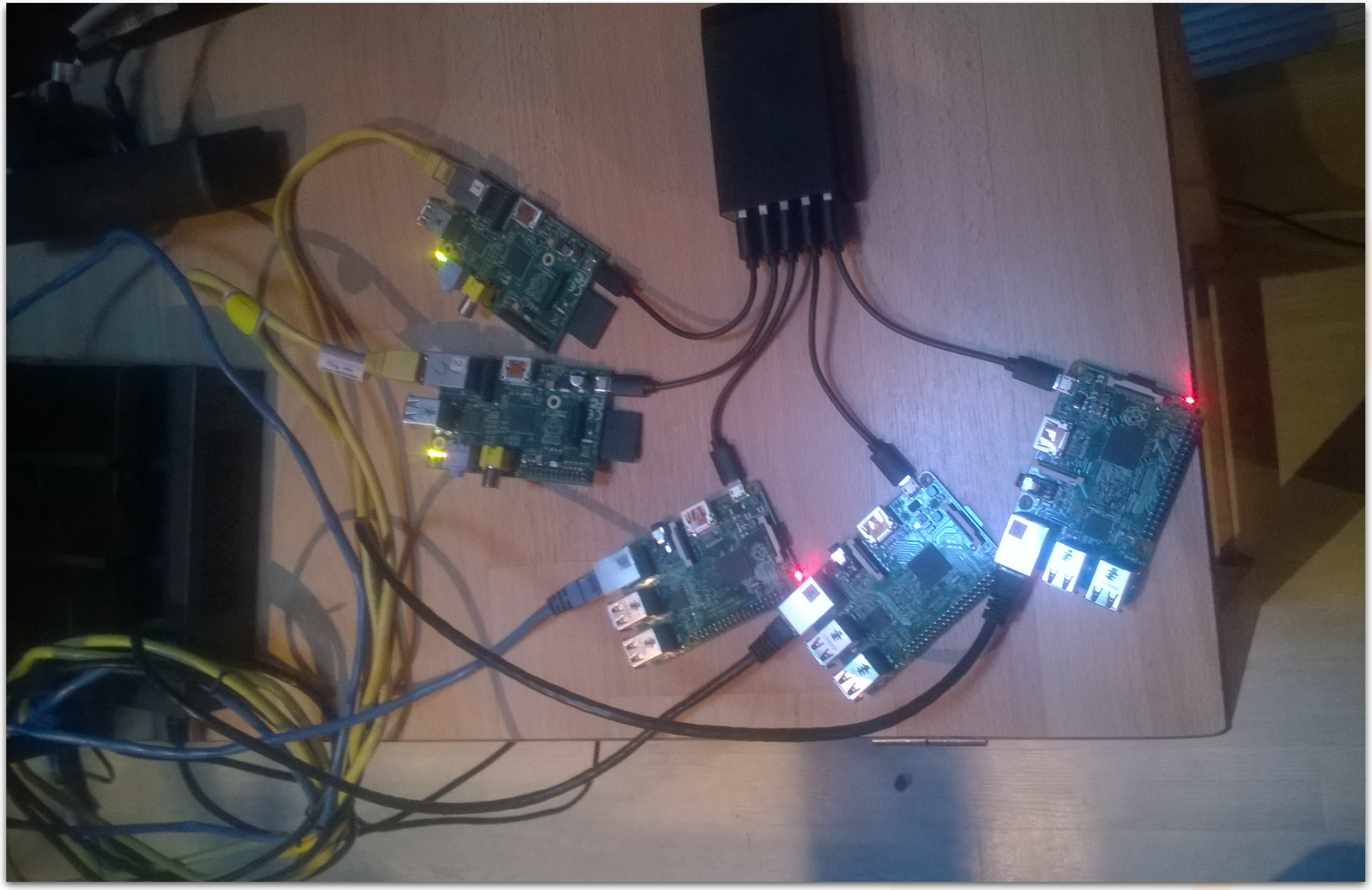
Load balancing example



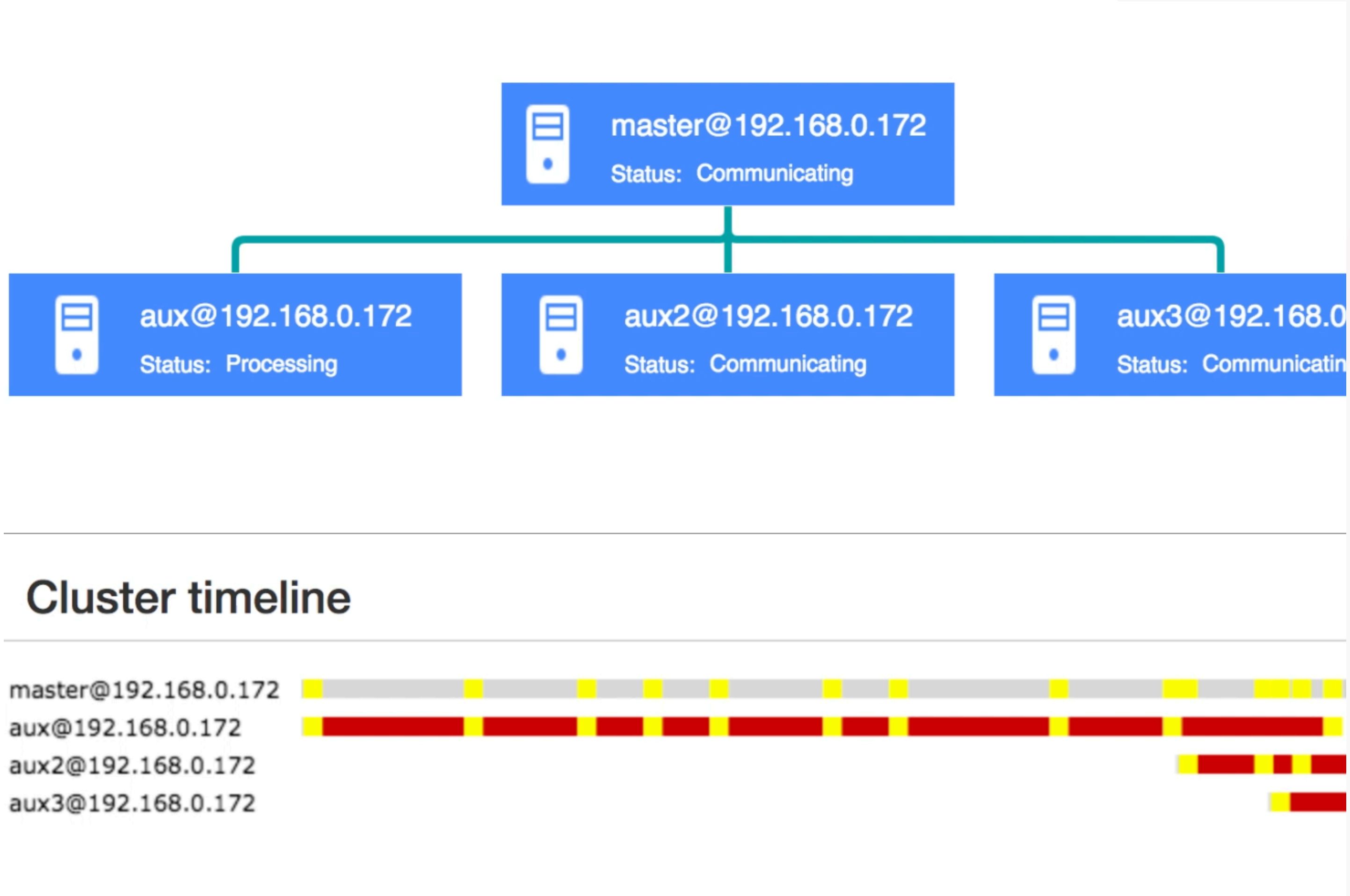
- Master can only handle one request at a time
- Workers can join/leave; they can have different processing speeds
- Goal: Minimize idle time by optimizing X for each worker

Load balancing example

Load balancing example



Load balancing example



Environment

```
{:ok, m} = Master.start_link([Application.fetch
    Master.create_job(m, 10000)
    Enum.map(Node.list(),
      fn(node) -> LoadBalancingExample.init_wo
    {:noreply, %{this | master: m}}
  end

def init_worker(this,master,node) do
  speed = Marlon.Utils.random_int(1,5)
  {:ok, w} = Worker.start_link_remote(node, [ma

  Worker.start(w)
  {:noreply, this}
end


defactor Worker do
  def init ([m, speed, chunk_time]) do
    {:ok, %{
      master: m,
      speed_factor: speed,
      wait_time: 0,
      chunk_time: chunk_time}}
  end

  async def start(this) do
    Worker.process_chunk(self(), 1)
    {:noreply, this}
  end

  async def process_chunk(this, chunk_size) do
    result = Master.request_work(this[:master],
    if result != :no_more_work do
      Process.send_after self(), {:processed_ch
        round(this[:chunk_time] * chunk_size /
    end
    {:noreply, this}
  end
```

```
reply def processed_chunk(this) do
  Master.work_finished(this[:master], self()
  Worker.process_chunk(self(), 1)
  {:noreply, this}
end

async def notify_wait_time(this, wait_time)
  new_state = %{this | wait_time: wait_time}

  {:noreply, new_state}
end
end

defactor Master do
  def init([comm_time]) do
    {:ok, %{
      comm_time: comm_time,
      chunks_remaining: 0,
      pending_request_size: 0,
      chunks_in_progress: %{}
    }}
  end

  sync def create_job(this, _from, job_size) do
    Logger.info "Job created (size: " <> to_string(job_size) <> ")"
    {:reply, :ok, %{this | chunks_remaining: job_size}}
  end

  sync def request_work(this, from, chunk_size) do
    new_pending = this[:pending_request_size] + chunk_size
    Worker.notify_wait_time(elem(from,0), new_pending * this[:comm_time])
    Process.send_after(self(), {:request_work_reply, chunk_size, from}, this[:comm_time])
    {:noreply, %{this | pending_request_size: new_pending}}
  end

  async def work_finished(this, worker) do
    {:noreply, %{this | chunks_in_progress: Map.delete(this[:chunks_in_progress], worker)}}
  end
```

```
  async def work_cancelled(this, worker) do
    revert_remaining = this[:chunks_remaining] - Map.
    {:noreply, %{this | chunks_in_progress: Map.delet
      chunks_remaining: revert_remaining}}
  end

  reply def request_work_reply(this, chunk_size, from
    {worker_pid, _} = from
    new_remaining = this[:chunks_remaining] - chunk_s
    new_pending = this[:pending_request_size] - chunk
    if (new_remaining >= 0) do
      GenServer.reply(from, :ok)
      {:noreply, %{this |
        chunks_remaining: new_remaining,
        pending_request_size: new_pending,
        chunks_in_progress: Map.put(this[:chunks_in_p
    else
      Logger.info "Master - No more work!"
      GenServer.reply(from, :no_more_work)
      {:noreply, %{this | chunks_remaining: 0, pendin
    end
  end
end
```

Environment + MARL integration

```
{:ok, m} = Master.start_link([Application.fetch
  Master.create_job(m, 10000)
  Enum.map(Node.list(),
    fn(node) -> LoadBalancingExample.init_wo
  {:noreply, %{this | master: m}}
end

def init_worker(this,master,node) do
  speed = Marlon.Utils.random_int(1,5)
  {:ok, w} = Worker.start_link_remote(node, [ma
  Worker.attach_agent(w, ChunkSizeGoal)
  Worker.start(w)
  {:noreply, this}
end


defactor Worker do
  def init ([m, speed, chunk_time]) do
    {:ok, %{
      master: m,
      speed_factor: speed,
      wait_time: 0,
      chunk_time: chunk_time}}
  end

  async def start(this) do
    Worker.do_action(self())
    {:noreply, this}
  end

  async def process_chunk(this, chunk_size) do
    result = Master.request_work(this[:master],
    if result != :no_more_work do
      Process.send_after self(), {:processed_ch
        round(this[:chunk_time] * chunk_size /
    end
    {:noreply, this}
  end
```

```
  reply def processed_chunk(this) do
    Master.work_finished(this[:master], self()
    Worker.do_action(self())
    {:noreply, this}
  end

  async def notify_wait_time(this, wait_time)
    new_state = %{this | wait_time: wait_time}
    Worker.update_reward(self(), new_state)
    {:noreply, new_state}
  end
end

defactor Master do
  def init([comm_time]) do
    {:ok, %{
      comm_time: comm_time,
      chunks_remaining: 0,
      pending_request_size: 0,
      chunks_in_progress: %{}
    }}
  end

  sync def create_job(this, _from, job_size) d
    Logger.info "Job created (size: " <> to_st
    {:reply, :ok, %{this | chunks_remaining: j
  end

  sync def request_work(this, from, chunk_size
    new_pending = this[:pending_request_size]
    Worker.notify_wait_time(elem(from,0), new_pending * this[:comm_time])
    Process.send_after(self(), {:request_work_reply, chunk_size, from}, this[:comm_time])
    {:noreply, %{this | pending_request_size: new_pending}}
  end

  async def work_finished(this, worker) do
    {:noreply, %{this | chunks_in_progress: Map.delete(this[:chunks_in_progress], worker)}}
  end
```

```
  async def work_cancelled(this, worker) do
    revert_remaining = this[:chunks_remaining] - Map.
    {:noreply, %{this | chunks_in_progress: Map.delet
      chunks_remaining: revert_remaining}}
  end

  reply def request_work_reply(this, chunk_size, from
    {worker_pid, _} = from
    new_remaining = this[:chunks_remaining] - chunk_s
    new_pending = this[:pending_request_size] - chunk
    if (new_remaining >= 0) do
      GenServer.reply(from, :ok)
      {:noreply, %{this |
        chunks_remaining: new_remaining,
        pending_request_size: new_pending,
        chunks_in_progress: Map.put(this[:chunks_in_p
    else
      Logger.info "Master - No more work!"
      GenServer.reply(from, :no_more_work)
      {:noreply, %{this | chunks_remaining: 0, pendin
    end
  end
end

defgoal OptimizeChunkSize do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]],
  reward fn(_agent, worker_state) ->  1 / worker_stat
end
```

Environment + MARL integration

```
{:ok, m} = Master.start_link([Application.fetch
   Master.create_job(m, 10000)
   Enum.map(Node.list(),
     fn(node) -> LoadBalancingExample.init_wo
   {:noreply, %{this | master: m}}
 end

def init_worker(this,master,node) do
  speed = Marlon.Utils.random_int(1,5)
  {:ok, w} = Worker.start_link_remote(node, [ma
  Worker.attach_agent(w, ChunkSizeGoal)
  Worker.start(w)
  {:noreply, this}
end


defactor Worker do
  def init ([m, speed, chunk_time]) do
    {:ok, %{
      master: m,
      speed_factor: speed,
      wait_time: 0,
      chunk_time: chunk_time}}
  end

  async def start(this) do
    Worker.do_action(self())
    {:noreply, this}
  end

  async def process_chunk(this, chunk_size) do
    result = Master.request_work(this[:master],
    if result != :no_more_work do
      Process.send_after self(), {:processed_ch
        round(this[:chunk_time] * chunk_size /
    end
    {:noreply, this}
  end
```

```
  reply def processed_chunk(this) do
    Master.work_finished(this[:master], self()
    Worker.do_action(self())
    {:noreply, this}
  end

  async def notify_wait_time(this, wait_time)
    new_state = %{this | wait_time: wait_time}
    Worker.update_reward(self(), new_state)
    {:noreply, new_state}
  end
end


defactor Master do
  def init([comm_time]) do
    {:ok, %{
      comm_time: comm_time,
      chunks_remaining: 0,
      pending_request_size: 0,
      chunks_in_progress: %{}
    }}
  end

  sync def create_job(this, _from, job_size) d
    Logger.info "Job created (size: " <> to_s
    {:reply, :ok, %{this | chunks_remaining:
  end

  sync def request_work(this, from, chunk_siz
    new_pending = this[:pending_request_size]
    Worker.notify_wait_time(elem(from,0), new_pending * this[:comm_time])
    Process.send_after(self(), {:request_work_reply, chunk_size, from}, this[:comm_time])
    {:noreply, %{this | pending_request_size: new_pending}}
  end

  async def work_finished(this, worker) do
    {:noreply, %{this | chunks_in_progress: Map.delete(this[:chunks_in_progress], worker)}}
  end
```

```
  async def work_cancelled(this, worker) do
    revert_remaining = this[:chunks_remaining] - Map.
    {:noreply, %{this | chunks_in_progress: Map.delet
      chunks_remaining: revert_remaining}}
  end

  reply def request_work_reply(this, chunk_size, from
    {worker_pid, _} = from
    new_remaining = this[:chunks_remaining] - chunk_s
    new_pending = this[:pending_request_size] - chunk
    if (new_remaining >= 0) do
      GenServer.reply(from, :ok)
      {:noreply, %{this |
        chunks_remaining: new_remaining,
        pending_request_size: new_pending,
        chunks_in_progress: Map.put(this[:chunks_in_p
    else
      Logger.info "Master - No more work!"
      GenServer.reply(from, :no_more_work)
      {:noreply, %{this | chunks_remaining: 0, pendin
    end
  end
end
```

```
defgoal OptimizeChunkSize do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]],
  reward fn(_agent, worker_state) -> 1 / worker_sta
end
```

MARL integration

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)
```

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)
```

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)
```

~~Worker.process_chunk(self(), 1)~~

```
Worker.do_action()
```

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)
```

~~Worker.process_chunk(self(), 1)~~

```
Worker.do_action()
```

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)
```

~~Worker.process_chunk(self(), 1)~~

```
Worker.do_action()
```

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

Learning algorithm (Exploring Selfish RL)

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)

~~Worker.process_chunk(self(), 1)~~
Worker.do_action()
```

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

Algorithm parameters

MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)
```

~~Worker.process_chunk(self(), 1)~~

```
Worker.do_action()
```

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

Action space

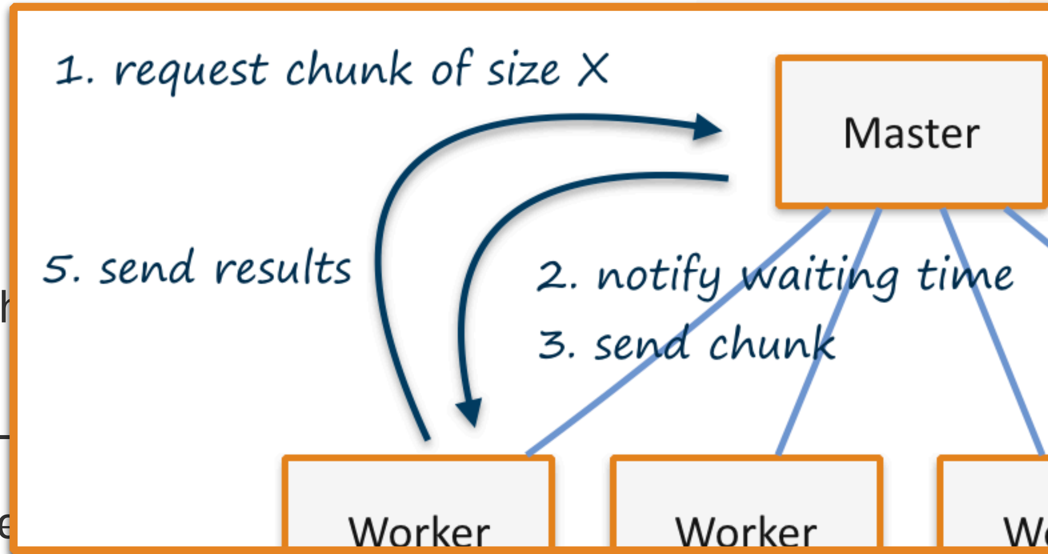
MARL integration

```
Worker.attach_agent(w, ChunkSizeGoal)

Worker.update_reward(self(), new_state)

~~Worker.process_chunk(self(), 1)~~
Worker.do_action()
```

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

Reward function

MARL integ



Worker.attach

Worker.update_

~~Worker.pre~~

Worker.do_action()

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
end
```

Reward function

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Which state to share with other agents

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Reduce the amount of communication

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Additional options

```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

Reduce the state space

Additional options

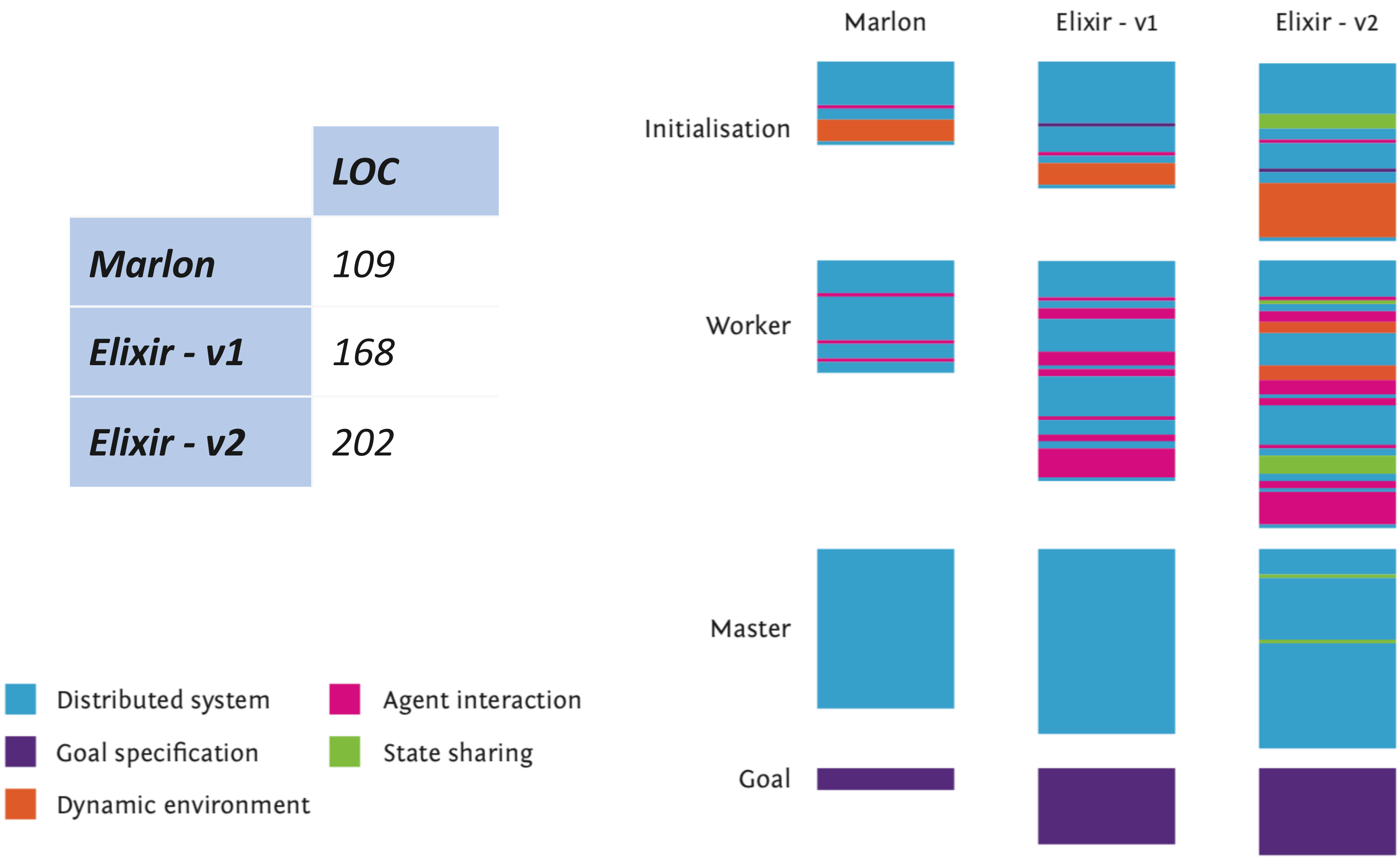
```
defgoal ChunkSizeGoal do
  type Marlon.ESRL
  params [explorations: 7, steps: 20]
  actions [process_chunk: [[1,2,3]]]
  reward fn(_agent, worker_state) ->
    1 / worker_state[:wait_time] end]
  shared [:wait_time]
  share_deviation [wait_time: 5]
  state_abstraction fn(worker_state) ->
    cond do
      worker_state[:wait_time] > 100 -> 2
      worker_state[:wait_time] > 10 -> 1
      true -> 0
    end
  end
end
```

MARL algorithm API

```
defclass Marlon.LearningAlgorithm do
  def init_agent(this, actor_state)
  def is_learning(this)
  def sample_action(this)
  def update(this, actor_state, reward)
  def action_probabilities(this)
  def agent_joined(this, agent_pid)
  def agent_left(this, agent_pid)
  ...
end
```

Marlon evaluation

	LOC
Marlon	*109*
Elixir - v1	*168*
Elixir - v2	*202*



Summary

Summary

- Marlon, a language to facilitate integrating MARL into distributed systems

Summary

- Marlon, a language to facilitate integrating MARL into distributed systems
- Future work: additional MARL algorithms, evaluate scalability

Summary

- Marlon, a language to facilitate integrating MARL into distributed systems
- Future work: additional MARL algorithms, evaluate scalability
- For more information, visit bit.ly/marlon-lang

Get in touch!

tmoldere@vub.be , boeyen@vub.be , cderoove@vub.be , wdmeuter@vub.be