ReactiveML and JoCaml: two concurrent extensions of OCaml

Louis Mandel

louis.mandel@lri.fr

Laboratoire de Recherche en Informatique

Université Paris-Sud 11

Synchron'08 – 03/12/2008

Programming of concurrent systems

General purpose programming language + dedicated constructs

Two experiments above Ocaml:

- Synchronous: ReactiveML
  - based on the synchronous reactive model of Boussinot
  - Programming systems with a lot of concurrence, communication and synchronisation
  - Interests: determinism, compositionnality, safety
- Asynchronous: JoCaml (Luc Maranget)
  - Based on the join-calculus
  - Programming of distributed systems
  - Interests: parallel execution

ReactiveML

killable

```
signal kill
val kill : (int, int list) event

let process killable p =
  let id = gen_id () in print_endline ("["^(string_of_int id)^"]");
  do run p
  until kill(ids) when List.mem id ids done
val killable : unit process -> unit process
```

Dynamic creation: reminder

```
let rec process extend to_add =
  await to_add(p) in
  run p || run (extend to_add)
val extend : ('a, 'b process) event -> unit process

signal to_add
  default process ()
  gather (fun p q -> process (run p || run q))
val add_to_me : (unit process, unit process) event
```

Dynamic creation with state

```
let rec process extend to_add state =
  await to_add(p) in
  run (p state) || run (extend to_add state)
val extend : ('a , ('b -> 'c process)) event -> 'b -> unit process

signal to_add
  default (fun s -> process ())
  gather (fun p q s -> process (run (p s) || run (q s)))
val to_add : (('_state -> unit process) , ('_state -> unit process)) event
```

extensible

```
signal add
val add : ((int * (state -> unit process)),
           (int * (state -> unit process)) list) event

let process extensible p_init state =
  let id = gen_id () in print_endline ("{"^(string_of_int id)^"}");
  signal add_to_me
    default (fun s -> process ())
    gather (fun p q s -> process (run (p s) || run (q s))) in
  run (p_init state) || run (extend add_to_me state)
  || loop
       await add(ids) in
       List.iter (fun (x,p) -> if x = id then emit add_to_me p) ids
     end
val extensible : (state -> 'a process) -> state -> unit process
```

JoCaml

JoCaml: one place buffer

```
let create () =
  def some(v) & get() = none() & reply v to get
  or none() & put(v) = some(v) & reply () to put
  in
  spawn none() ; (* buffer initially empty *)
  (put, get)
```

JoCaml: infinite buffer

```
let create () =
  def state(xs,y::ys) & get() =
    state(xs,ys) & reply y to get

  or state(xs,ys) & put(x) =
    state(x::xs,ys) & reply () to put

  or state(_::_ as xs,[]) & get() =
    state([],List.rev xs) & reply get() to get
  in
  spawn state([],[]) ; (* buffer initially empty *)
  (put, get)
```

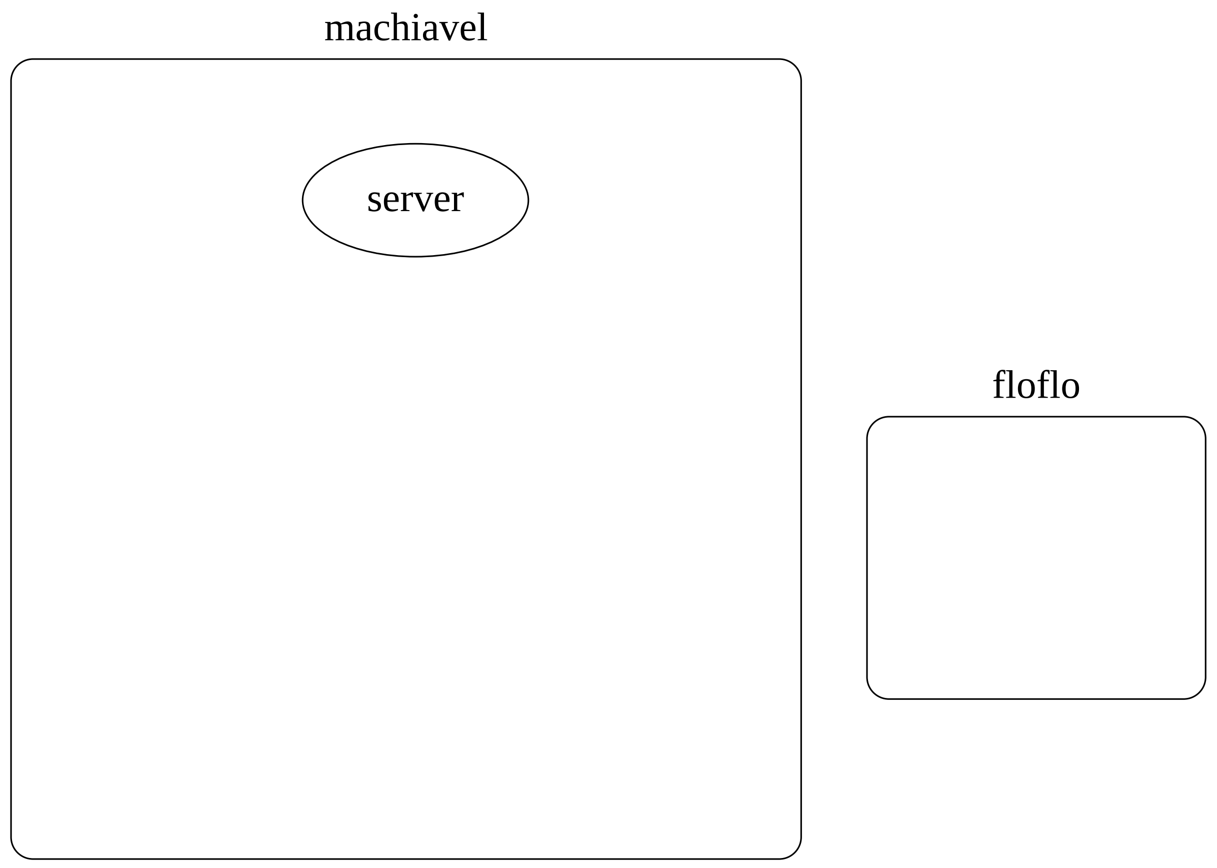
Boids

Simulation of a flock of birds, a school of fish . . .

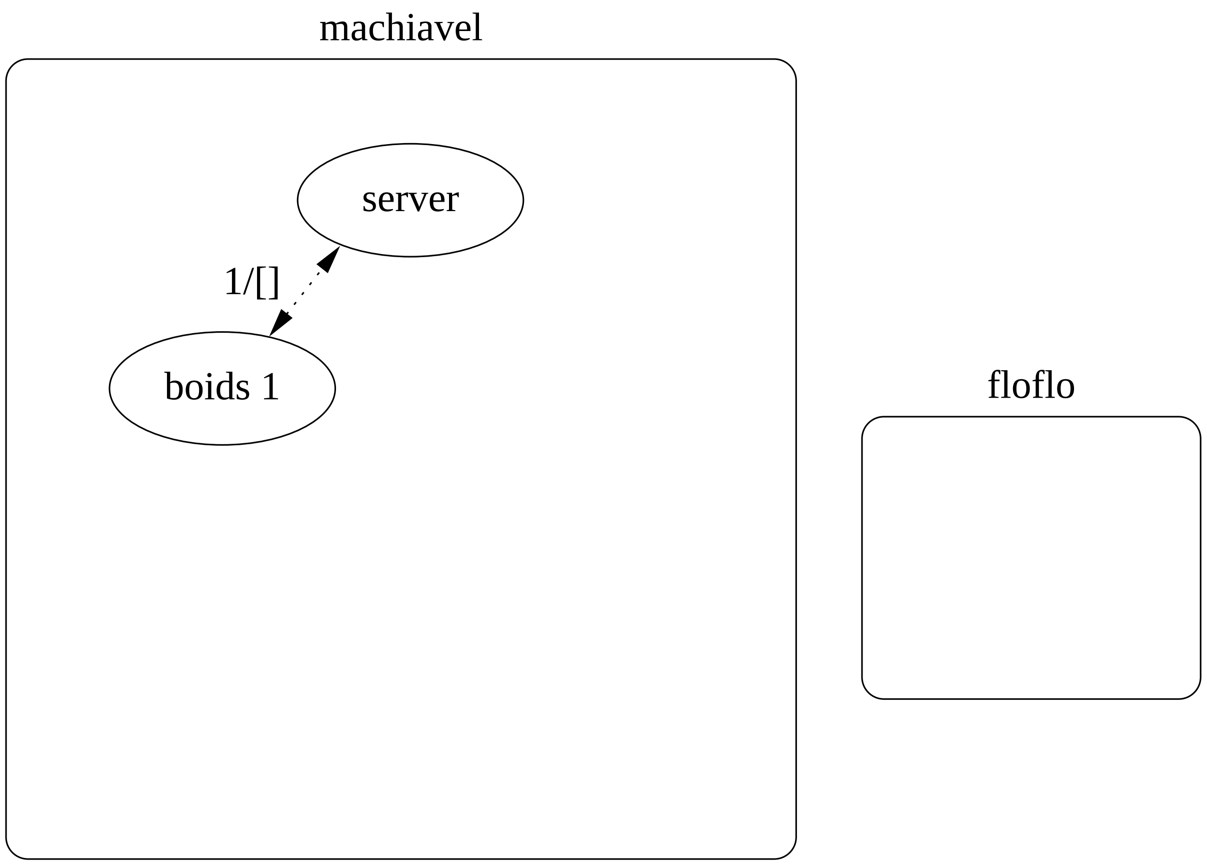
Main points:

- ReactiveML and JoCaml collaboration
- centralized and distributed execution
- channels mobility, dynamic aspects
- failure detection, timeout

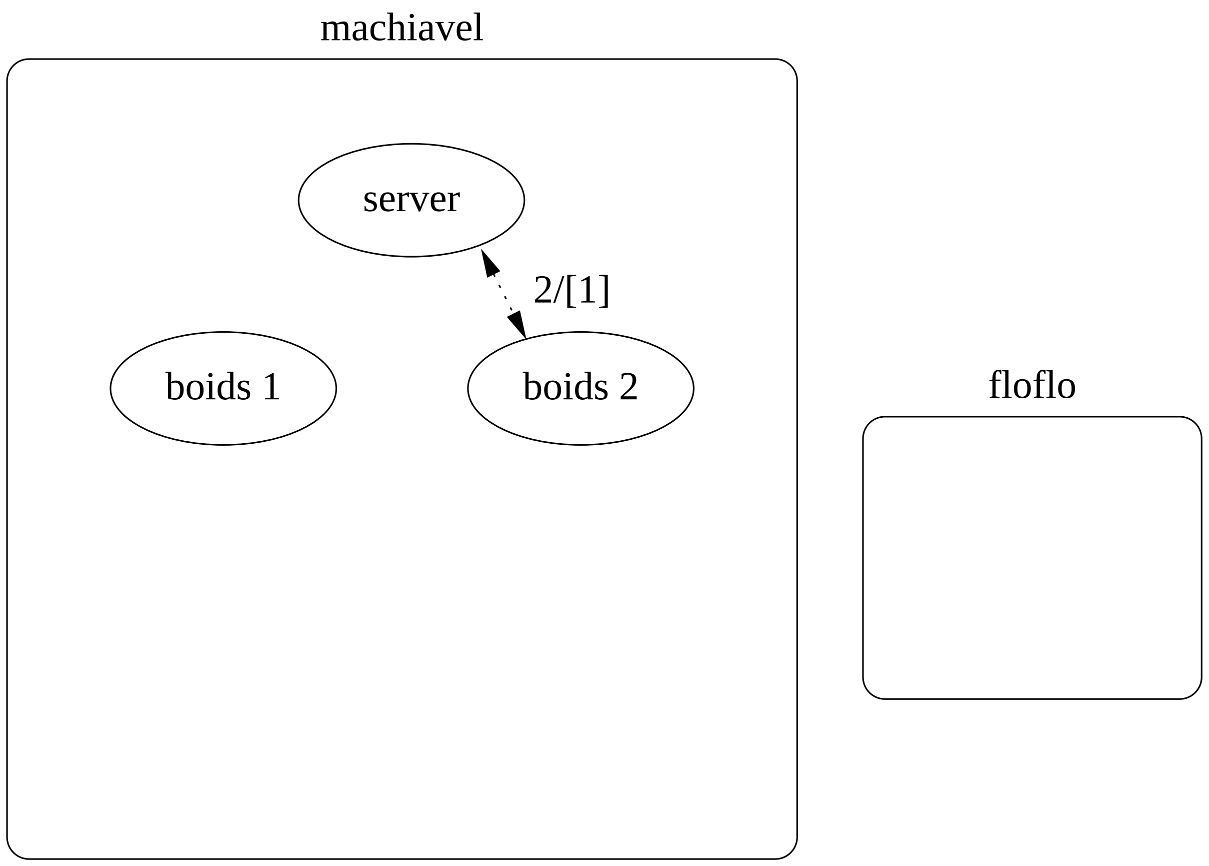
Boids



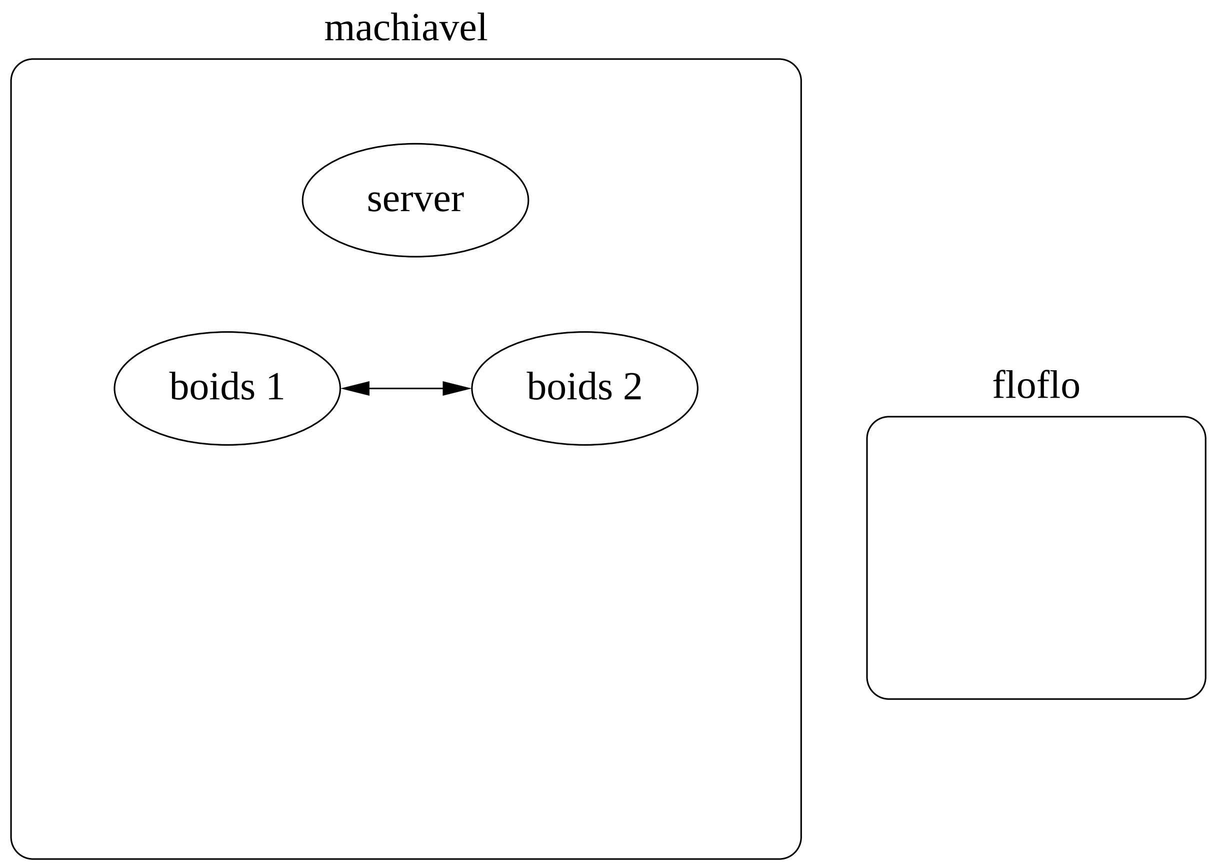
Boids



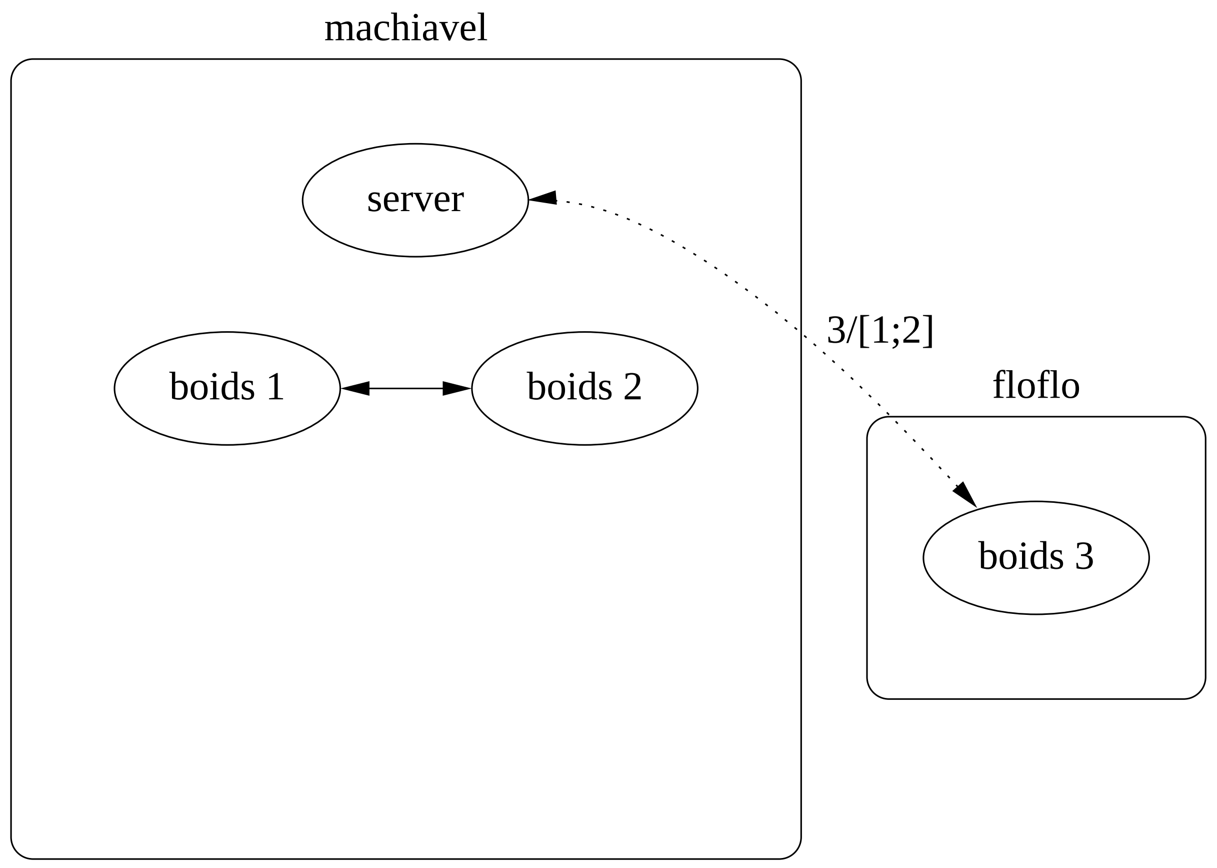
Boids



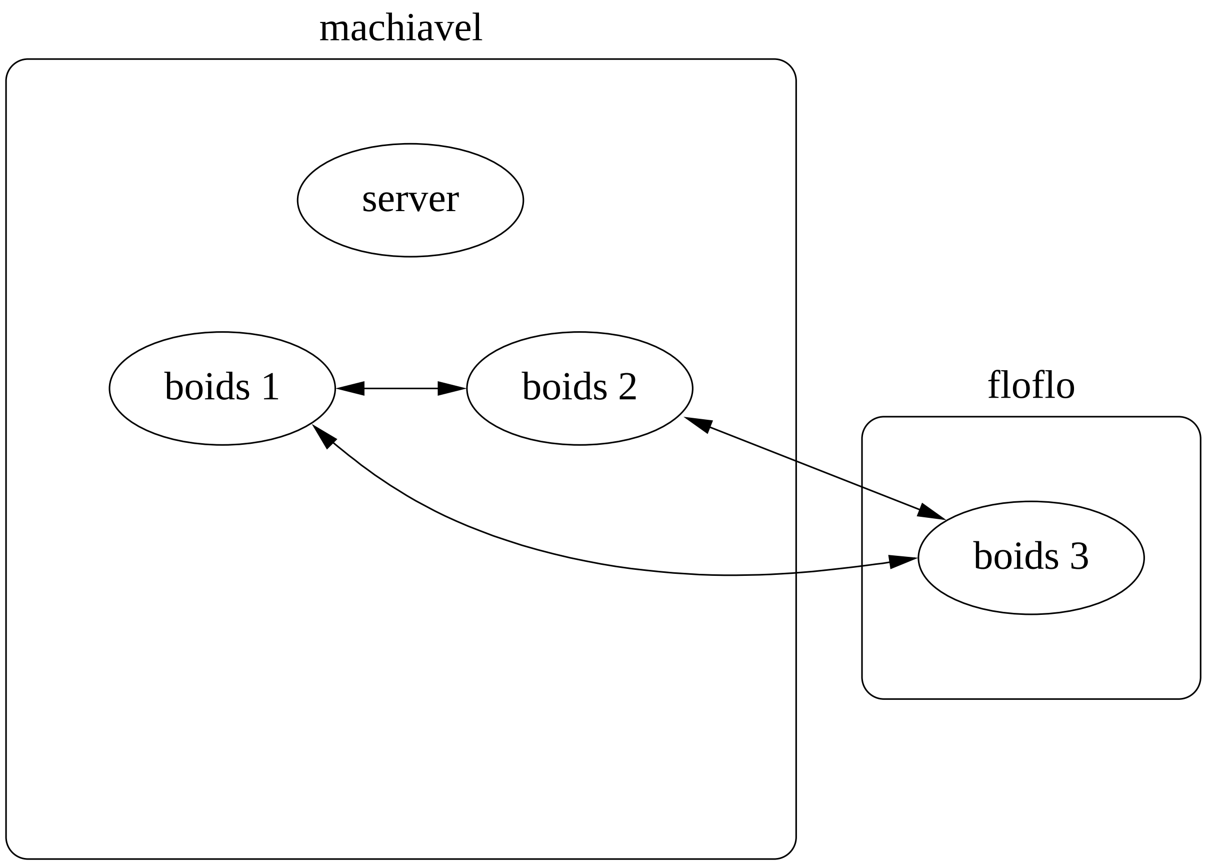
Boids



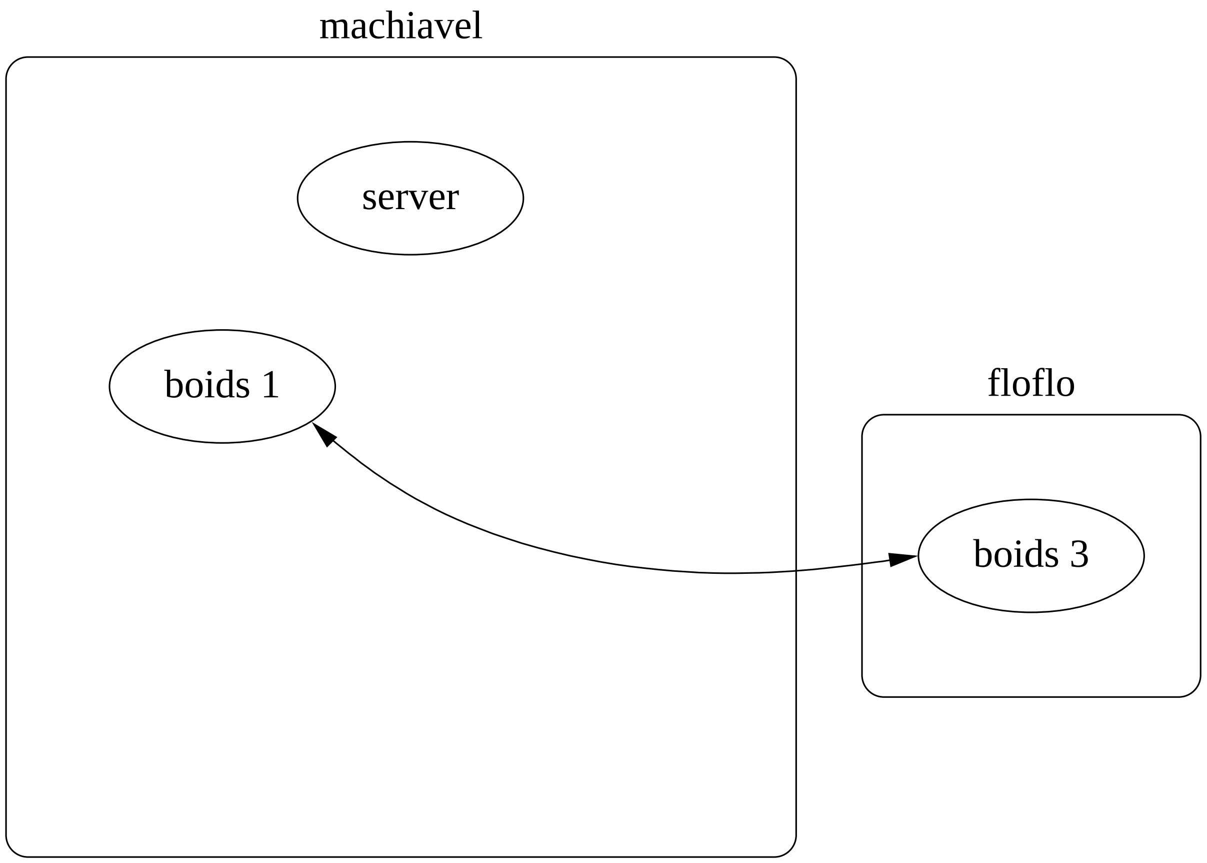
Boids



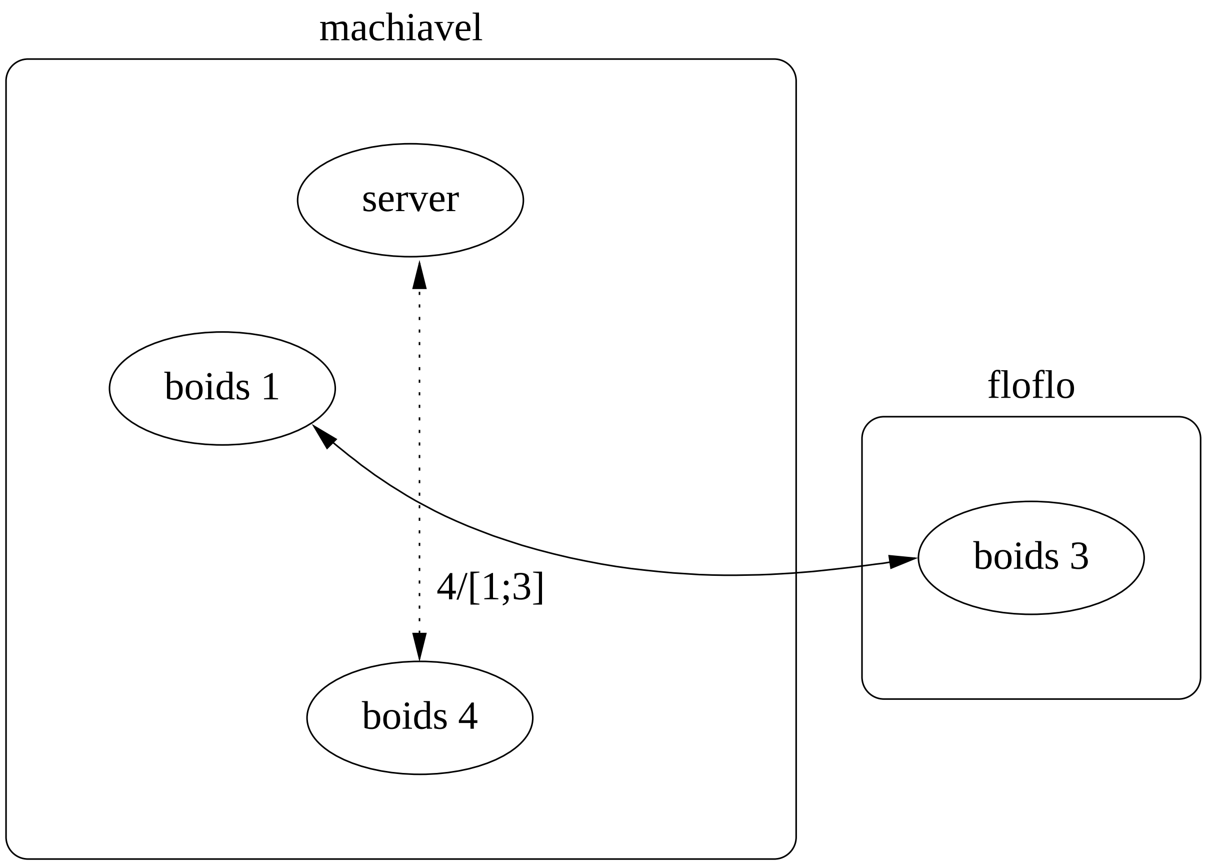
Boids



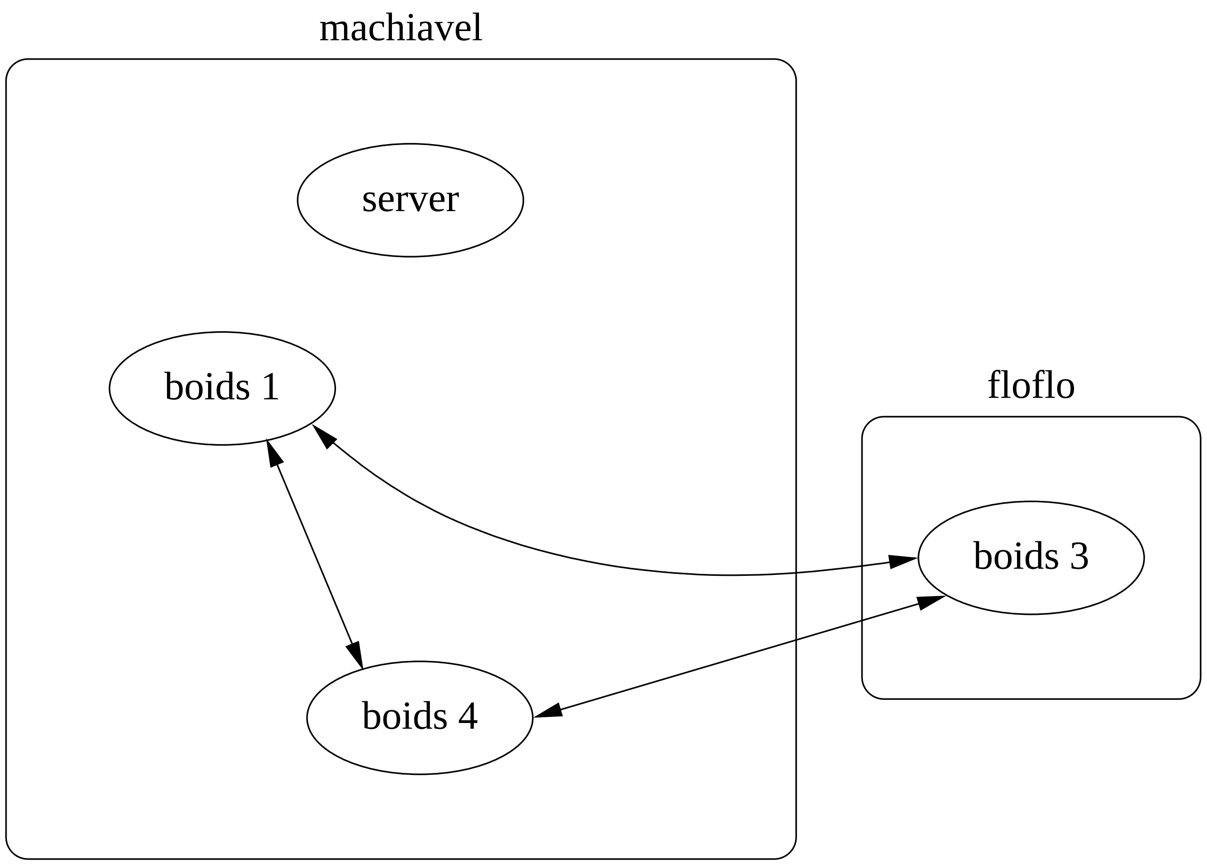
Boids



Boids



Boids



Implementations are Available

`http://rml.lri.fr`

`http://jocaml.inria.fr`

Asynchronous Communication

```
let new_cell () =
  def state (_) & set(x) = state(Some x) & reply () to set
   or state (Some x) & get() = state(None) & reply x to get in
  spawn (state None);
  (set, get)
val new_cell : ('a -> unit process, unit -> 'a process)

let set_step, get_step = new_cell()
let process generate_step =
  loop let n = run (get_step ()) in emit step n ; pause end
```