Journées Francophones des Langages Applicatifs 2010

Cours de ReactiveML

Louis Mandel

Laboratoire de Recherche en Informatique
Université Paris-Sud 11
INRIA Saclay – Ile-de-France

ANR-08-EMER-010

Sémantique

Le noyau ReactiveML

$$
\begin{array}{lcl}
e & ::= & x \mid c \mid (e,e) \mid \lambda x.e \mid e\ e \mid \texttt{rec}\ x \texttt{=} e \mid \texttt{process}\ e \\
 & & \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{and}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{pause} \mid \texttt{run}\ e \\
 & & \mid \texttt{signal}\ x\ \texttt{default}\ e\ \texttt{gather}\ e\ \texttt{in}\ e \\
 & & \mid \texttt{present}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{emit}\ e\ e \mid \texttt{pre}\ e \mid \texttt{pre ?}e \\
 & & \mid \texttt{do}\ e\ \texttt{until}\ e(x)\ \texttt{->}\ e\ \texttt{done} \mid \texttt{do}\ e\ \texttt{when}\ e \\
c & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{0} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots
\end{array}
$$

Opérateurs dérivés

$$
\begin{array}{rcl}
e_1 \texttt{||} e_2 & \stackrel{def}{=} & \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{in ()} \\
 & \ldots &
\end{array}
$$

Sémantiques statiques

Analyse d'instantanéité

- exemple :

```
let f x =
  let y = x + 1 in
  pause;
  print_int y
```

incorrect

```
let process f x =
  let y = x + 1 in
  pause;
  print_int y
```

correct

Typage

- Extension conservative du typage de ML

$$\frac{H \vdash e_1 : (\tau_1, \tau_2)\,\texttt{event} \quad H \vdash e_2 : \tau_1}{H \vdash \texttt{emit}\ e_1\ e_2 : \texttt{unit}} \qquad \dots$$

Sémantiques dynamiques

Sémantique comportementale ("grands pas")

- qu'est ce qu'une réaction valide ?
- abstraction de l'ordonnancement à l'intérieur d'un instant

$$N \vdash e \xrightarrow[S]{E,\, b} e'$$

Sémantique opérationnelle ("petits pas")

- comment obtenir une réaction valide ?
- description de tous les ordonnancements possibles

$$e/S_0 \rightarrow e_1/S_1 \rightarrow \cdots \rightarrow e_n/S \rightarrow_{eoi} e'$$

La sémantique comportementale

Forme des réductions

$$N \vdash e \xrightarrow[S]{E, b} e'$$

- N ensemble des noms de signaux n créés par la réaction de e
- E signaux émit par la réaction de e
- S environnement de signaux dans lequel e doit réagir
- b statut de terminaison

Comme pour Esterel, nous avons l'invariant $E \sqsubseteq S$.

Sémantique Comportementale

Exemple de règles

$$\emptyset \vdash v \xrightarrow[S]{\emptyset,\, true} v \qquad\qquad \emptyset \vdash \texttt{pause} \xrightarrow[S]{\emptyset,\, false} \texttt{()}$$

$$\frac{N_1 \vdash e \xrightarrow[S]{E,\, true} n \qquad n \in S \qquad N_2 \vdash e_1 \xrightarrow[S]{E_1,\, b} e_1'}{N_1 \cdot N_2 \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E \sqcup E_1,\, b} e_1'}$$

$$\frac{N \vdash e \xrightarrow[S]{E,\, true} n \qquad n \notin S}{N \vdash \texttt{present}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \xrightarrow[S]{E,\, false} e_2}$$

Sémantique Comportementale

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, b_1} e_1' \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2, b_2} e_2' \qquad b_1 \wedge b_2 = \mathit{false}}{N_1 \cdot N_2 \vdash e_1 \,||\, e_2 \xrightarrow[S]{E_1 \sqcup E_2, \mathit{false}} e_1' \,||\, e_2'}$$

$$\frac{N_1 \vdash e_1 \xrightarrow[S]{E_1, \mathit{true}} v_1 \qquad N_2 \vdash e_2 \xrightarrow[S]{E_2, \mathit{true}} v_2}{N_1 \cdot N_2 \vdash e_1 \,||\, e_2 \xrightarrow[S]{E_1 \sqcup E_2, \mathit{true}} ()}$$

⇒ L'environnement S est global.

Sémantique opérationnelle

La sémantique opérationnelle se décompose en 3 étapes :

- réaction pendant l'instant

$$e/S \rightarrow^* e'/S'$$

- calcul des sorties

$$O = next(S)$$

- réaction de fin d'instant

$$O \vdash e' \rightarrow_{eoi} e''$$

Sémantique opérationnelle

Réduction en tête de terme

$$(\lambda x.e)\ v \,/\, S \ \rightarrow_\varepsilon\ e[x \leftarrow v] \,/\, S \qquad\qquad \texttt{emit}\ n\ v \,/\, S \ \rightarrow_\varepsilon\ \texttt{()} \,/\, S + [v/n]$$

$$\texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \,/\, S \ \rightarrow_\varepsilon\ e_1 \,/\, S \ \text{ si }\ n \in S \qquad \ldots$$

Contextes

$$\begin{array}{rcl} \Gamma & ::= & [\,] \mid \Gamma\,\texttt{;}\,e \mid \texttt{present}\ \Gamma\ \texttt{then}\ e\ \texttt{else}\ e \\ & & \mid \texttt{let}\ x\ \texttt{=}\ \Gamma\ \texttt{and}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{and}\ x\ \texttt{=}\ \Gamma\ \texttt{in}\ e \mid \ldots \end{array}$$

$$\frac{e \,/\, S \ \rightarrow_\varepsilon\ e' \,/\, S'}{\Gamma(e)/S \rightarrow \Gamma(e')/S'} \qquad \frac{n \in S \qquad e/S \rightarrow e'/S'}{\Gamma(\texttt{do}\ e\ \texttt{when}\ n)/S \rightarrow \Gamma(\texttt{do}\ e'\ \texttt{when}\ n)/S'}$$

Sémantique opérationnelle

Fin d'instant

$$\frac{n \notin O}{O \vdash \texttt{present}\ n\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightarrow_{eoi} e_2}$$

$$O \vdash \texttt{pause} \rightarrow_{eoi} \texttt{()}$$

$$\ldots$$

avec $O = next(S)$

Propriétés

Sémantique comportementale

- déterministe

Sémantique opérationnelle

- preuve de sûreté du typage

Sémantiques comportementale et opérationnelle

- équivalence entre les deux sémantiques
- absence de problème de causalité

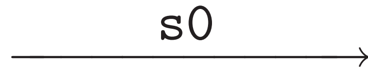
Implantation de ReactiveML

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```




```
await immediate s1 || await immediate s0; emit s1
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

$$\xrightarrow{\texttt{s0}}$$

```
await immediate s1 || await immediate s0; emit s1
```

$$\xrightarrow{\texttt{s0, s1}}$$

```
await immediate s1
```

Les clés d'un interprète efficace : l'attente passive

```
await immediate s1 || await immediate s0; emit s1 || emit s0
```

$$\xrightarrow{\texttt{s0}}$$

```
await immediate s1 || await immediate s0; emit s1
```

$$\xrightarrow{\texttt{s0, s1}}$$

```
await immediate s1
```

$$\xrightarrow{\texttt{s0, s1}} \quad \texttt{()}$$

⇒ Il faut réactiver une instruction uniquement lorsque le signal dont elle dépend est émis : utilisation de files d'attente

Les clés d'un interprète efficace

D'autres points clés :

- Exécution du code Ocaml sans surcoût
- Gestion efficace des signaux
  - accès en temps constant
  - allocation/désallocation automatique
- ...

Sémantique et implantation sans suspension ni préemption

L_k : un langage à base de continuations

- traduction de ReactiveML vers L_k : $C_k[e_1\,;e_2] = C_{(C_k[e_2])}[e_1]$ ...
- exemple :

```
let nat k =
  fun _ ->
    (let cpt = ref 0 in
     Lk_record.rml_loop
      (fun k' ->
        Lk_record.rml_compute (fun () -> print_int !cpt; ...)
          (Lk_record.rml_pause k'))
      ())
```

Sémantique de $\mathbf{L}_k$

Sémantique gloutonne

- toujours aller de l'avant
- représentation du programme
  - $\mathcal{C}$ ensemble des expressions à exécuter instantanément
  - $\mathcal{W}$ ensemble des expressions en attente d'un signal
  - J ensemble des points de synchronisation

Exécution d'une étape de réaction

$$S,\ J,\ \mathcal{W} \vdash \mathtt{<} e \mathtt{,}\ v \mathtt{>} \longrightarrow S',\ J',\ \mathcal{W}' \vdash \mathcal{C}$$

- e expression à exécuter
- v valeur précédente

Implantation en OCaml

Les règles de la sémantique L_k peuvent se traduire quasiment directement en des fonctions de transition de type :

$$\begin{array}{lcl} step & = & env \times value \rightarrow env \\ env & = & signal_env \times join \times waiting \times current \end{array}$$

En implantant l'environnement directement dans le tas, les fonctions de transitions ont le type Ocaml suivant :

```
type 'a step = 'a -> unit
```

Implantation en OCaml : `compute`

$$\frac{e/S \Downarrow v'/S'}{S,\ J,\ \mathcal{W} \vdash \texttt{<} e.k \texttt{,}\ v \texttt{>} \longrightarrow S',\ J,\ \mathcal{W} \vdash \texttt{<} k \texttt{,}\ v' \texttt{>}}$$

Implantation en OCaml : `compute`

$$\frac{e/S \Downarrow v'/S'}{S,\ J,\ \mathcal{W} \vdash <e.k\,,\,v> \longrightarrow S',\ J,\ \mathcal{W} \vdash <k\,,\,v'>}$$

La fonction de transition `compute` est définie par :

```
let compute e k =
  fun v ->
    let v' = e() in
    k v'
val compute : (unit -> 'a) -> 'a step -> 'b step
```

Implantation en OCaml : `await/immediate`

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\ J,\ \mathcal{W} \vdash \texttt{<}\,\texttt{await immediate}\ e.k\,\texttt{,}\ v\,\texttt{>} \longrightarrow S',\ J,\ \mathcal{W} \vdash \texttt{<}\,k\,\texttt{,}\ \texttt{()}\,\texttt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \textit{self} = \texttt{await immediate}\ n.k}{S,\ J,\ \mathcal{W} \vdash \texttt{<}\,\texttt{await immediate}\ e.k\,\texttt{,}\ v\,\texttt{>} \longrightarrow S',\ J,\ \mathcal{W} + [\texttt{<}\,\textit{self}\,\texttt{,}\ v\,\texttt{>}/n] \vdash \emptyset}$$

Implantation en OCaml : `await/immediate`

$$\frac{e/S \Downarrow n/S' \qquad n \in S'}{S,\ J,\ \mathcal{W} \vdash \texttt{<}\,\texttt{await immediate}\ e.k\,\texttt{,}\ v\,\texttt{>} \longrightarrow S',\ J,\ \mathcal{W} \vdash \texttt{<}\,k\,\texttt{,}\ ()\,\texttt{>}}$$

$$\frac{e/S \Downarrow n/S' \qquad n \notin S' \qquad \textit{self} = \texttt{await immediate}\ n.k}{S,\ J,\ \mathcal{W} \vdash \texttt{<}\,\texttt{await immediate}\ e.k\,\texttt{,}\ v\,\texttt{>} \longrightarrow S',\ J,\ \mathcal{W} + [\texttt{<}\,\textit{self}\,\texttt{,}\ v\,\texttt{>}/n] \vdash \emptyset}$$

```
let await_immediate e k =
  fun v ->
    let (n, w) = e() in
    let rec self () =
      if Event.status n then k ()
      else w := self :: !w
    in self ()
val await_immediate : (unit -> ('a, 'b) event) -> unit step -> 'c step
```

Implantation en OCaml : `emit`

```
let emit e1 e2 k =
  fun v ->
    let (n, w) = e1() in
    let v' = e2() in
    Event.emit n v';
    current := !w @ !current;
    !w := [];
    k ()
val emit :
  (unit -> ('a, 'b) event) -> (unit -> 'a) -> unit step
    -> 'c step
```

Sémantique de L$_k$

Les suspensions et préemptions ?

- on a perdu la structure du programme !
- utilisation d'un arbre de contrôle

Bibliothèque pour la programmation réactive

```
val rml_compute: (unit -> 'a) -> 'a expr
val rml_seq: 'a expr -> 'b expr -> 'b expr
val rml_par: 'a expr -> 'b expr -> unit expr
...
```

L'expression ReactiveML :

```
(await s1 || await s2); emit s3
```

se traduit en Ocaml par :

```
rml_seq
  (rml_par
    (rml_await (fun () -> s1))
    (rml_await (fun () -> s2)))
  (rml_emit (fun () -> s3)))
```

ReactiveML

Toplevel

`rmltop` : le toplevel ReactiveML

Basé sur l'idée des Reactive Scripts [Boussinot & Hazard 96]

Utile pour :

- comprendre le modèle réactif
- faire des expériences de reconfiguration dynamique
- concevoir des systèmes réactifs

Démo

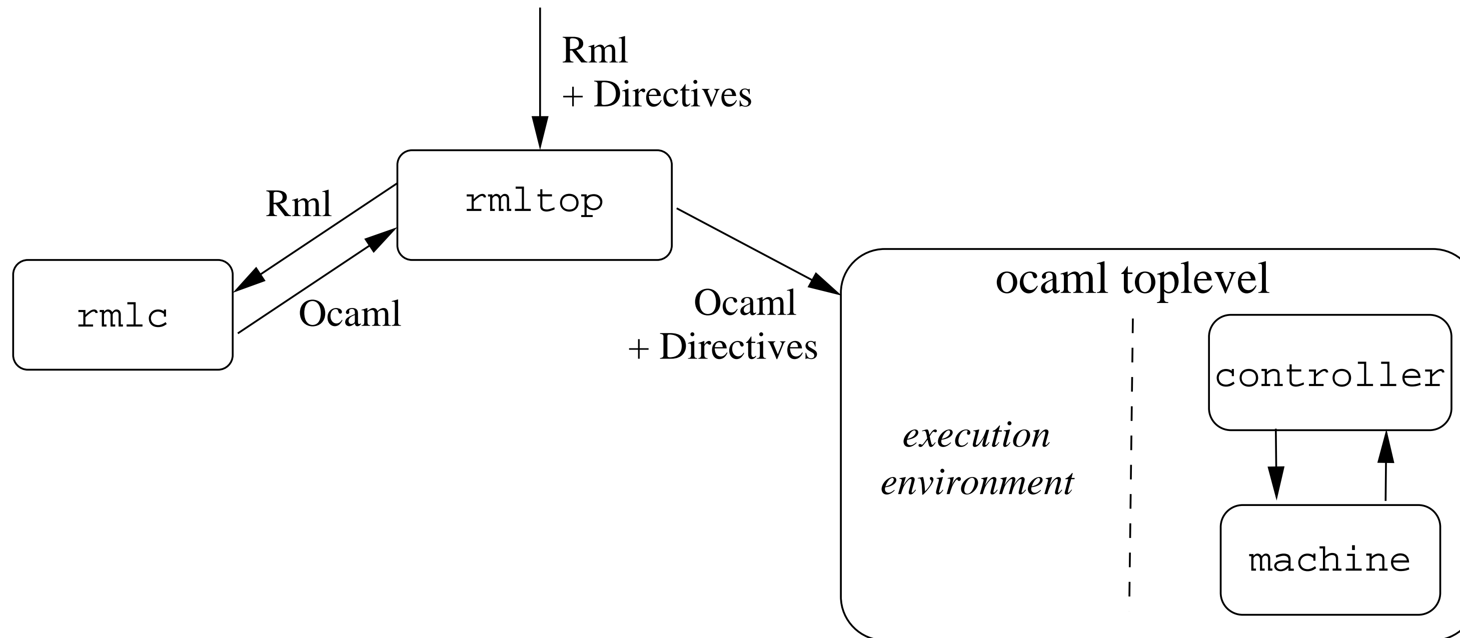
Glonemo

- `rmltop graphics.cma glonemo.cma`

n-Corps

- `http://rml.lri.fr/rmltop`

Implantation

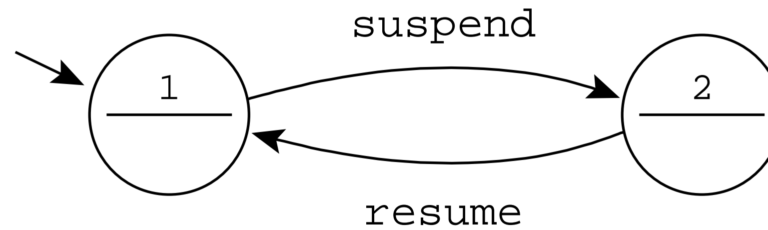


contrôleur implémenté en ReactiveML

Contrôleur

```
let process sampled =
  loop Rmltop_reactive_machine.rml_react(get_to_run()); pause end

let process step_by_step =
  loop
    await step(n) in
    do
      for i = 1 to n do
        Rmltop_reactive_machine.rml_react(get_to_run()); pause
      done
    until suspend done
  end
```

Contrôleur




```
let process machine_controller =
  loop
    do run sampled until suspend done;
    do run step_by_step until resume done
  end
```

Reconfiguration dynamique

Langage pour étudier la reconfiguration dynamique

Des combinateurs pour manipuler (individuellement) des processus en cours d'exécution

- tuer
- suspendre/reprendre
- ajouter des branches parallèles supplémentaires
- ...

Facilement programmable en ReactiveML

- utilisation de l'ordre supérieur et du polymorphisme

```
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
```

Création dynamique : rappel

```
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
```

Création dynamique avec état

```
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

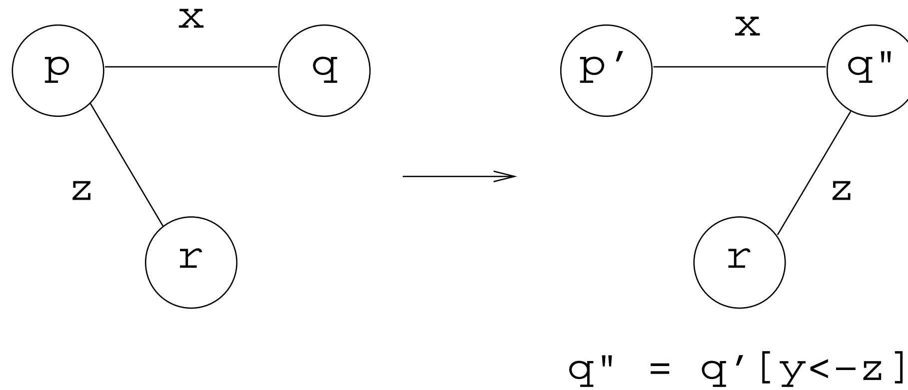
```
signal add
val add : ((int * (state -> unit process)),
           (int * (state -> unit process)) list) event

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
```

Bibliothèque pour le toplevel

```
type ident

val kill: (int , ident list) event
val killable: 'a process -> 'a option process

val sr: (int , ident list) event
val suspendable: 'a process -> 'a process

val extensible:
  ('a, (int * ('state -> unit process)) list) event ->
  ('state -> unit process) -> 'state -> unit process

val ps: unit -> unit
```

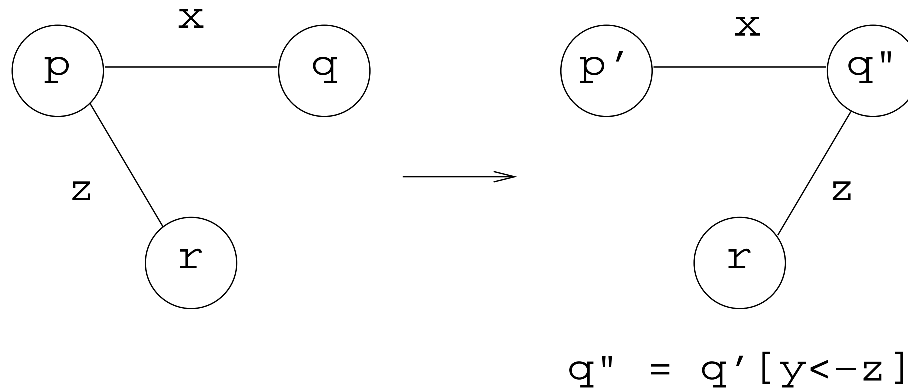
Autres exemples

Mobilité du π-calcul




```
let process p x z =
  emit x z;
  run (p' x)
val p : ('a, 'b) event -> 'a -> unit process


let process q x =
  await one x(y) in
  run (q' y)
val q : ('a, 'a list) event -> unit process
```

Mobilité du π-calcul




```
let process r z = ...
val r : ('a, 'b) event -> unit process


let process mobility x z =
  run (p x z) || run (q x) || run (r z)
val mobility :
  (('a, 'b) event, ('a, 'b) event list) event ->
  ('a, 'b) event -> unit process
```

Encodage des canaux du π-calcul `pi.rml`

Collaboration entre ReactiveML et JoCaml

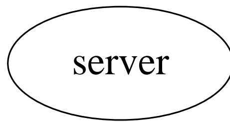
JoCaml

Extension de Ocaml basée sur le join-calcul

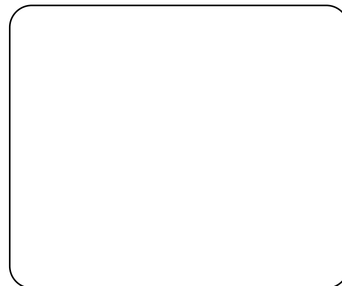
- asynchrone
- exécution distribuée

Boids

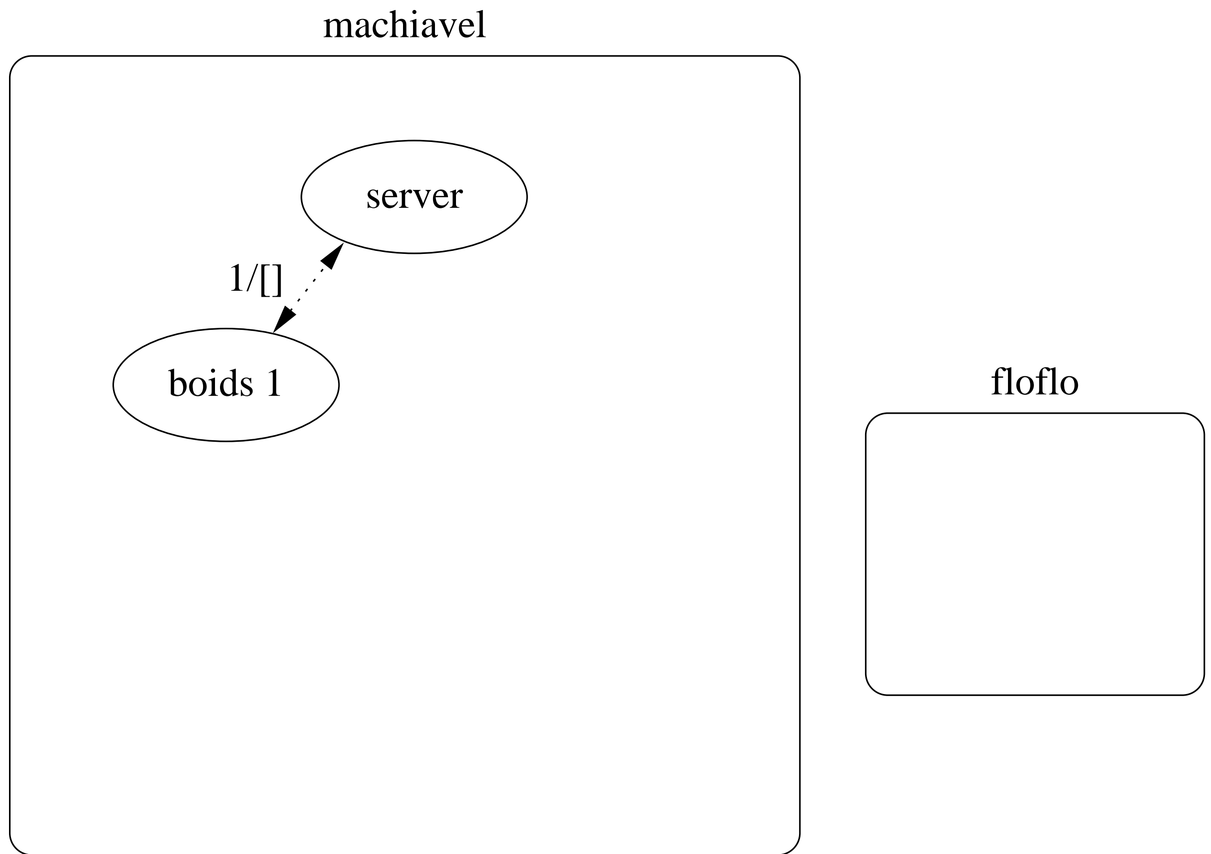
machiavel



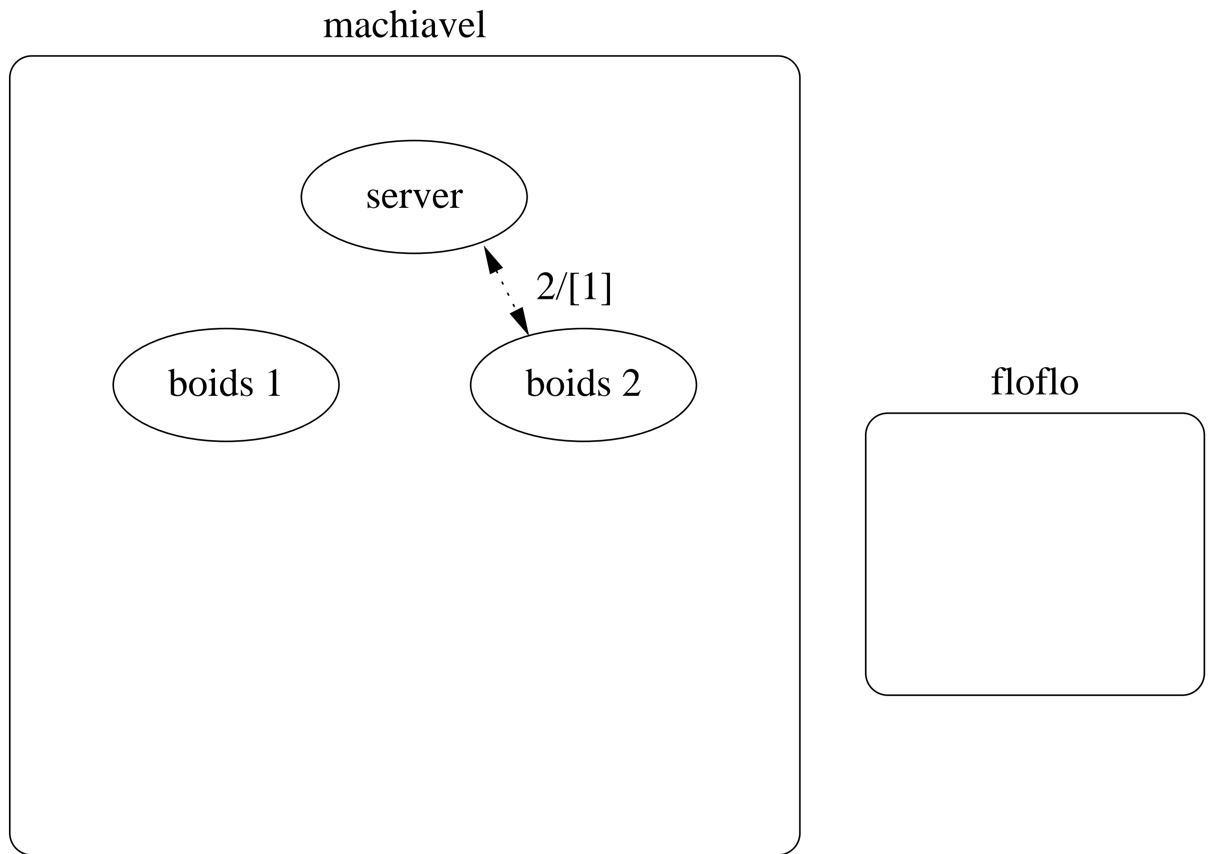
floflo



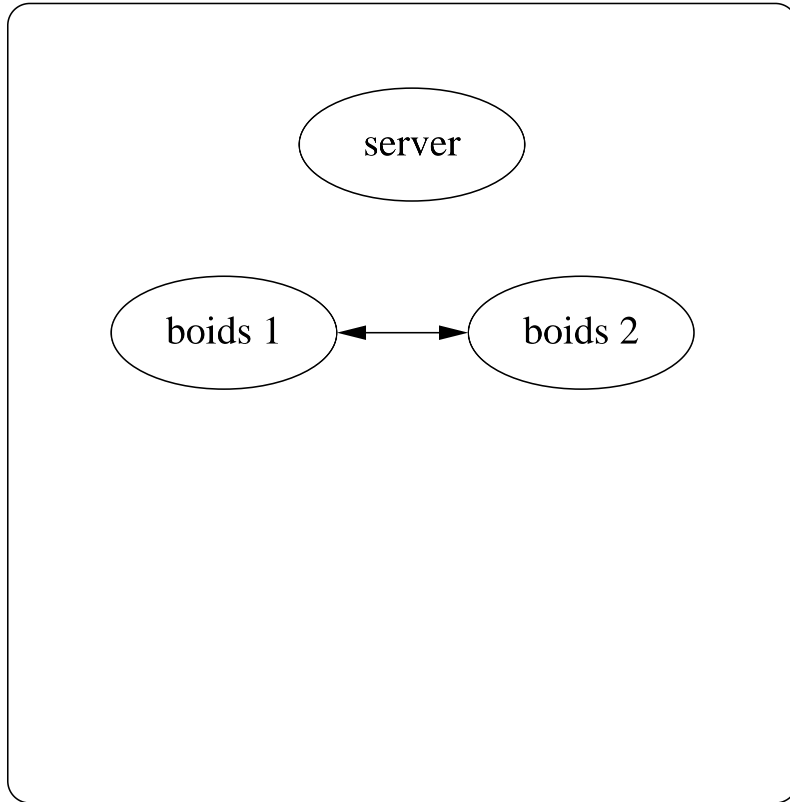
Boids

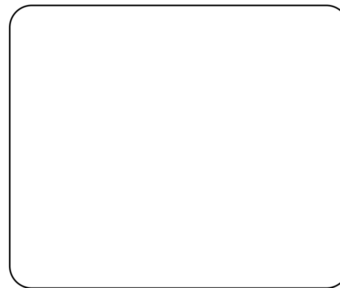


Boids

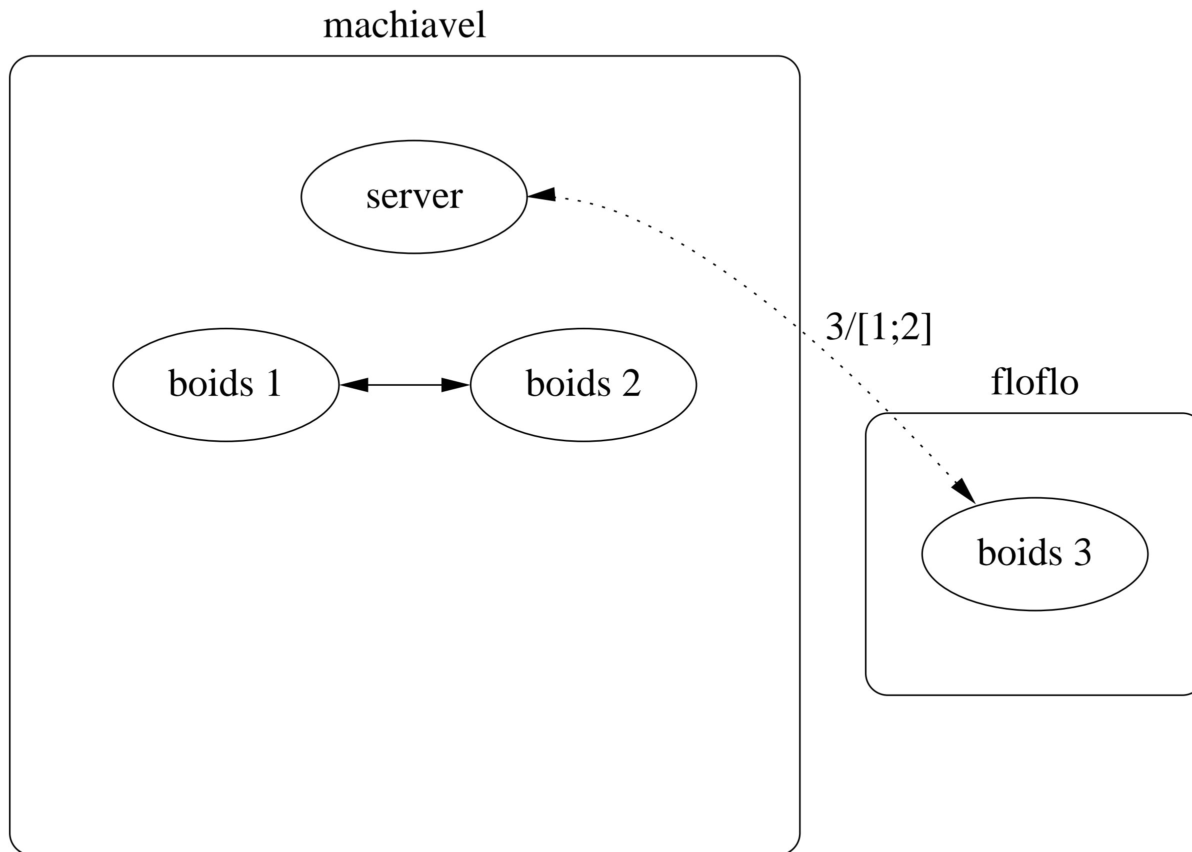


Boids

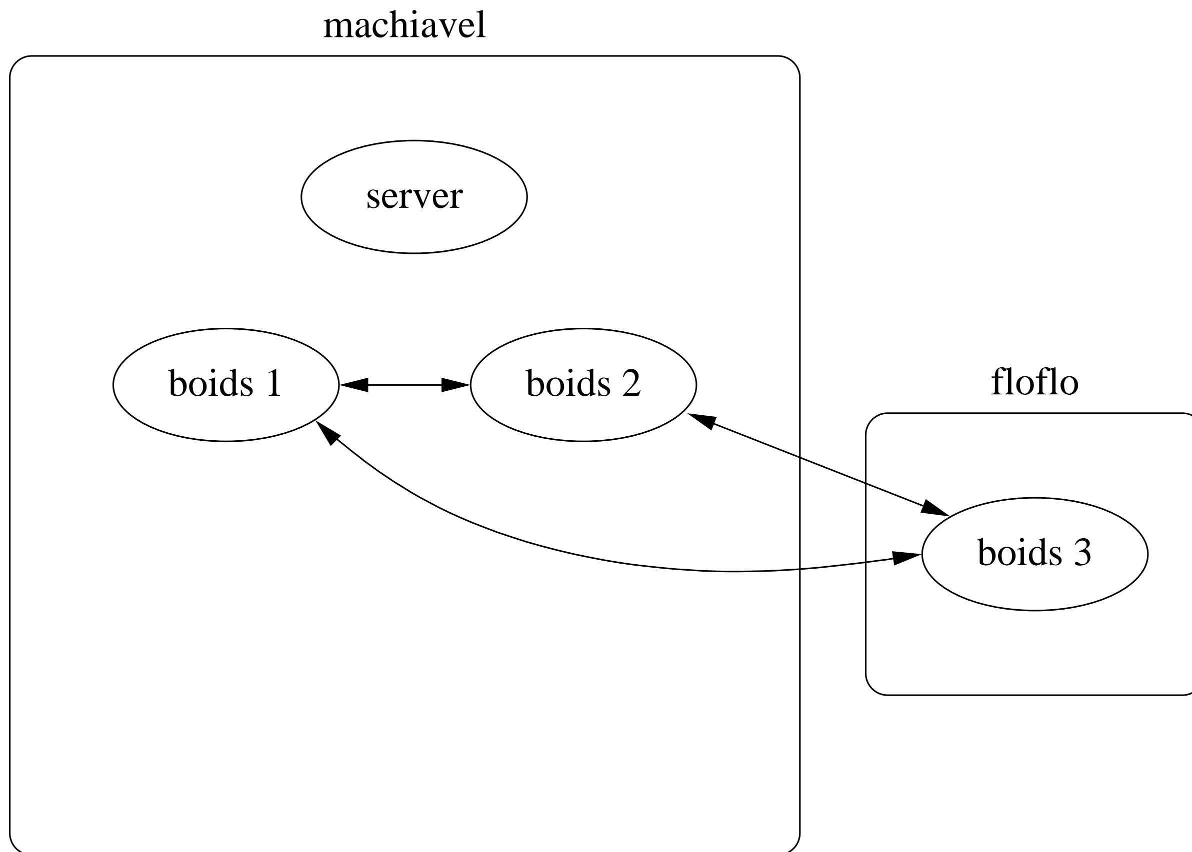




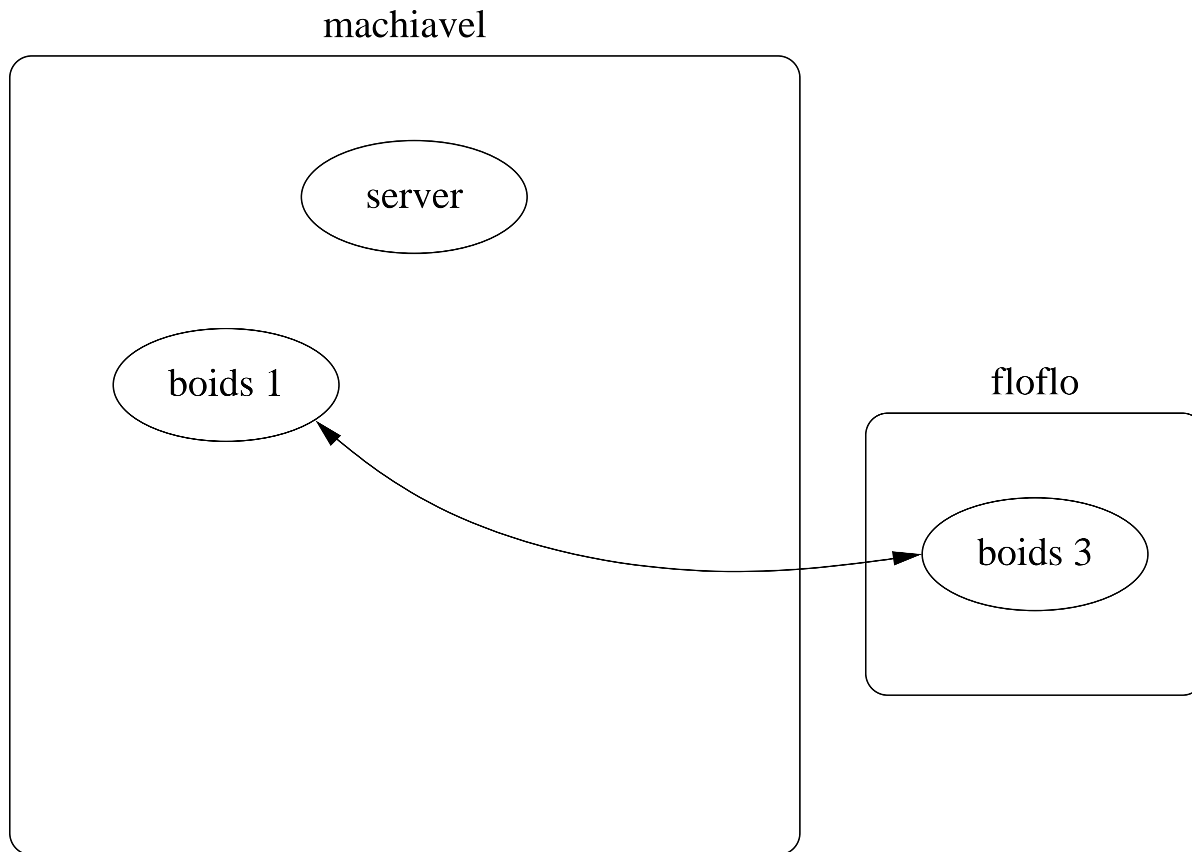
Boids



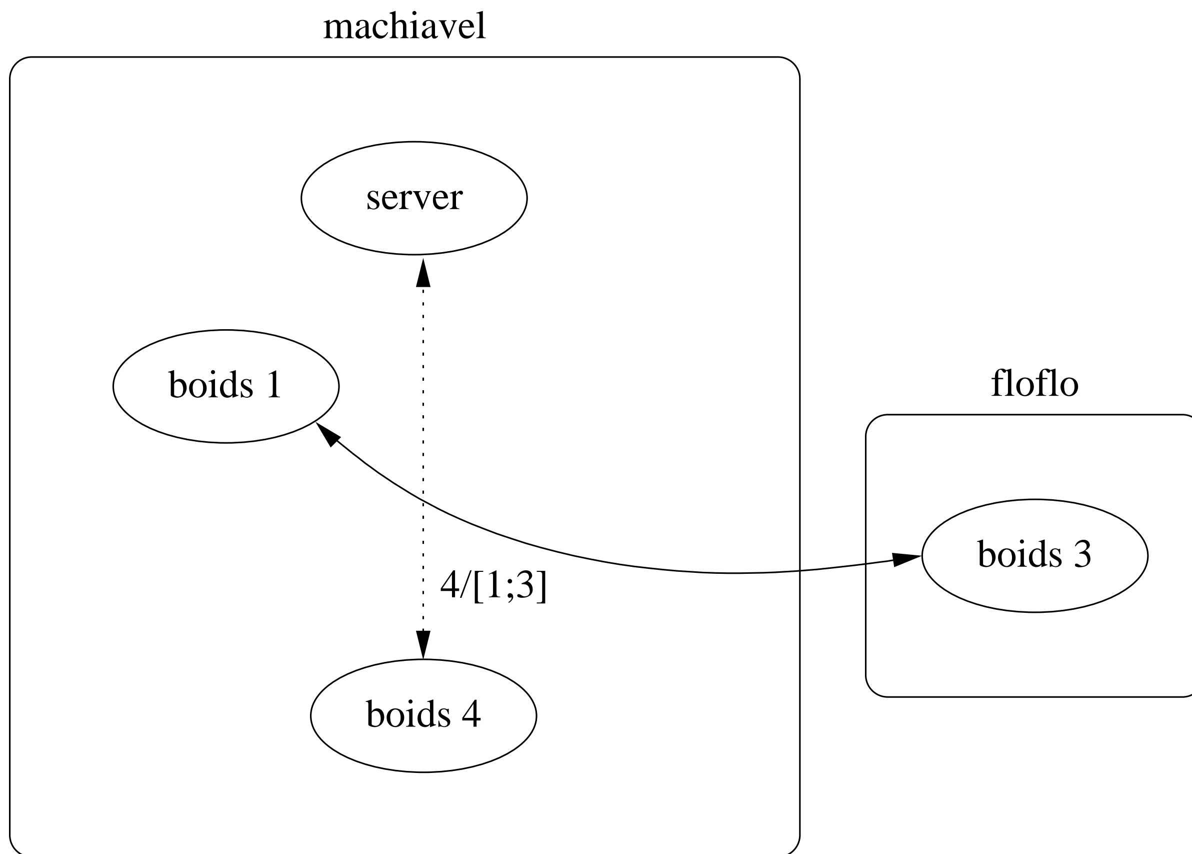
Boids



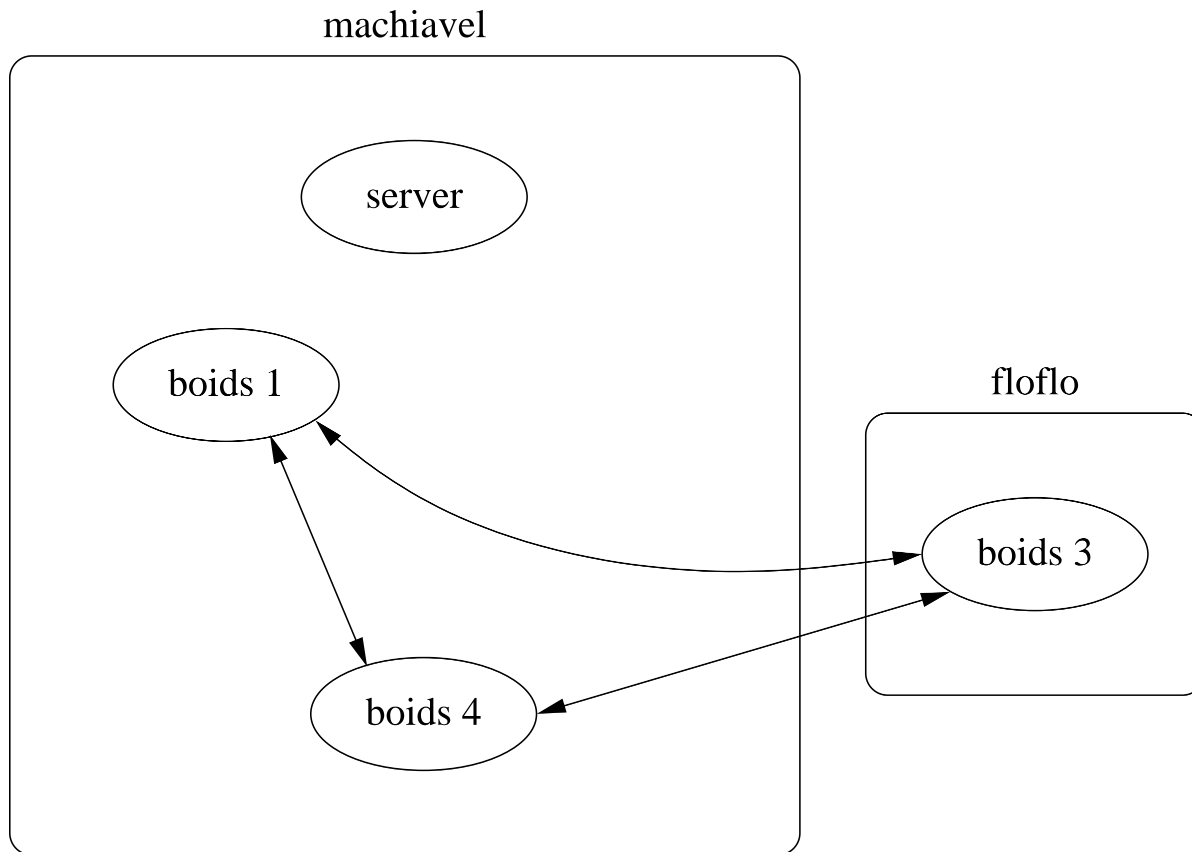
Boids



Boids



Boids



Conclusion

`http://rml.lri.fr`