Ruby FF : A Ruby implementation of FastFlow

Guy Tremblay
Professeur
Département d'informatique

UQAM
http://www.labunix.uqam.ca/~tremblay

25 février 2015

Presentation outline

Ruby’s characteristics

Some of Ruby's characteristics

Dynamic typing ⇒ No compile-time type checking

```
def create( *args )
  if args[0].class == Fixnum
    ...
  elsif args[0].class == Proc
    ...
  else
    ...
  end
end
```

⇒ Flexible constructors

Some of Ruby's characteristics

Duck typing

- The type of an object does not matter.
- What matters is the messages it can respond.

Methods can be defined "on the spot"

```ruby
class A
  def foo; puts "foo"; end
end

a = A.new
a.define_singleton_method :bar { puts "bar" }

a.foo => foo
a.bar => bar
```

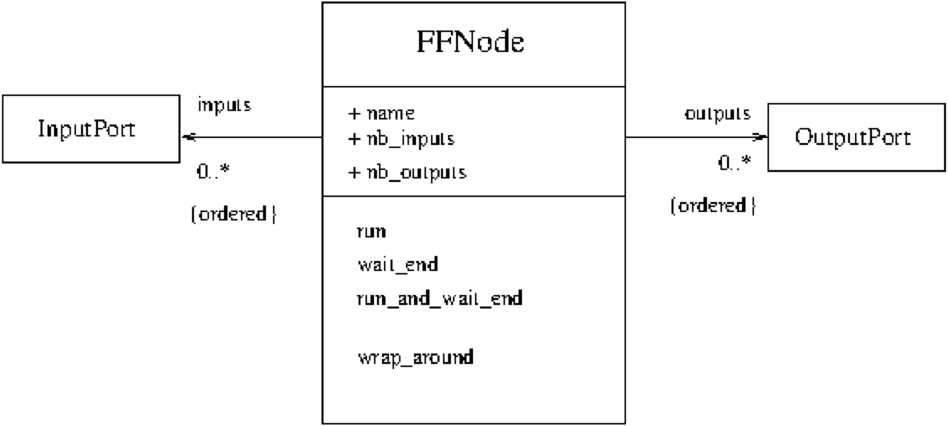
Some of Ruby's characteristics

Flexible syntax

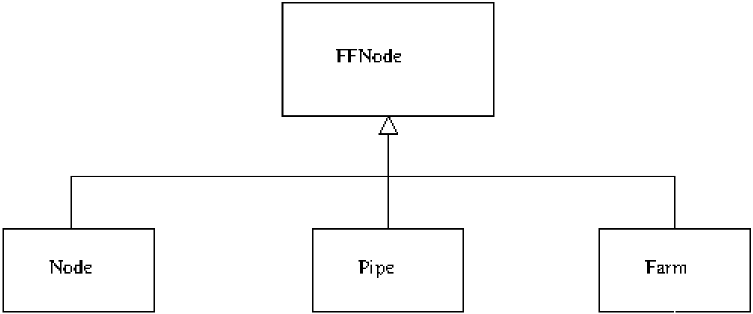
```
def m( a, args )
  ... a ...
  ... args[:size] ...
  ... args[:name] ...
end

m x, name: "foo", size: 10
```

Ruby FF’s meta-model

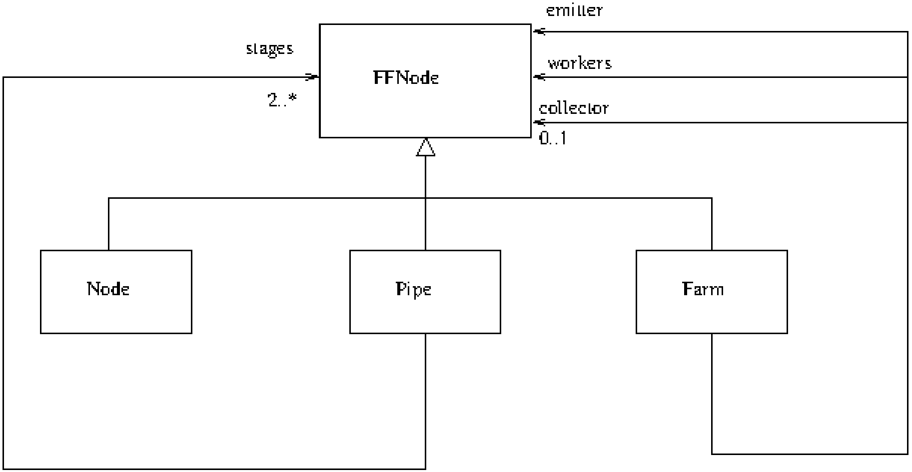
`FFnodes` have an arbitrary number of input/output ports (including 0 !)



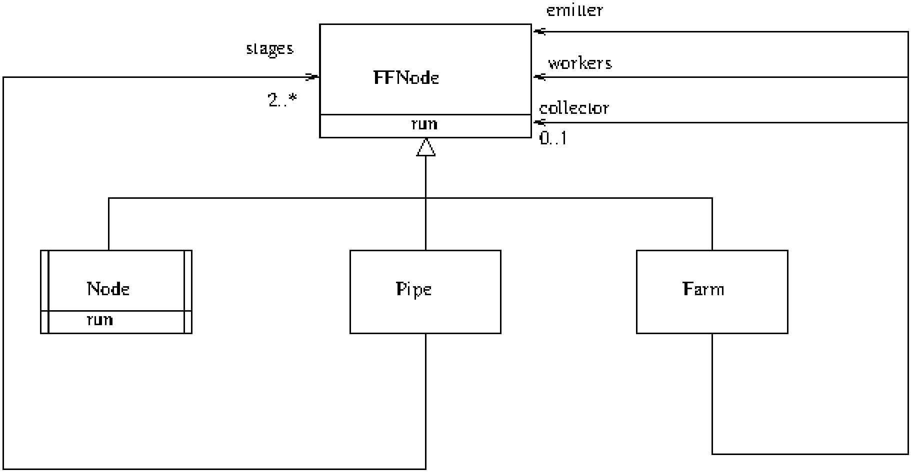
FFnodes are (abstract) composite objects (composite design pattern)



FFnodes are (abstract) composite objects (composite design pattern)

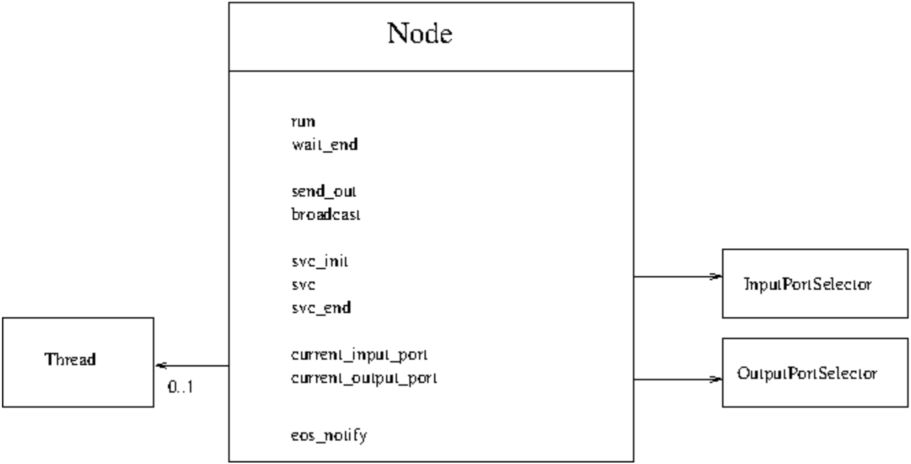


Any `FFNode` can be run as a thread, but `Nodes` are the real active objects

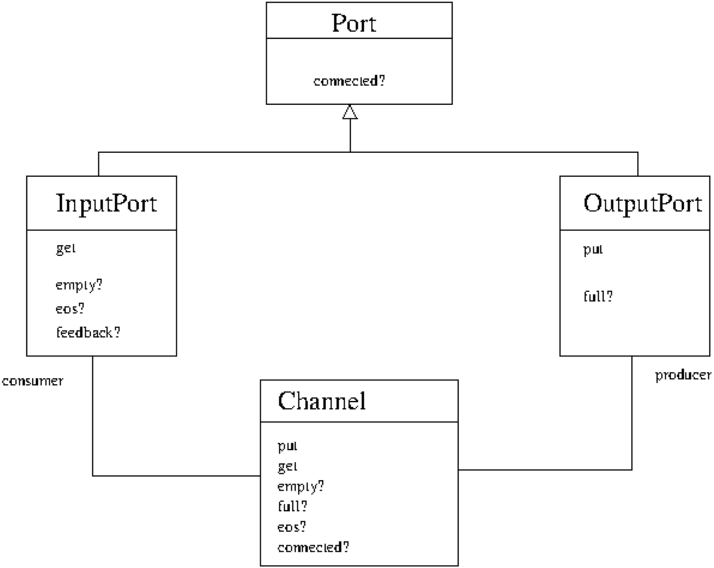



```
FFNode.run = inner_nodes.map(&:run)
Pipe.inner_nodes = stages
Farm.inner_nodes = [emitter, *workers, collector]
```

`Nodes` handle the low-level operations, provide the `svc*` methods, etc.



Nodes send/receive to/from Channels through Ports (act as proxies)



Some examples

Some remarks about the examples

- Most of the examples are adapted from “Parallel Programming Using FastFlow (Version September 2014)”, by M. Torquati.

- All my examples are executable test cases — although I don’t always show all the detailed code

- I write code using (pragmatic) TDD :
  - = Test-Driven Development
  - ⇒ *«Code the unit test first !»*
  - ≈ Any new code should be accompanied by new tests

Some facts about my Ruby code

- Number of classes (excluding tests)

`Debug`	1
`DBC`	1
`FF`	10
Total	12

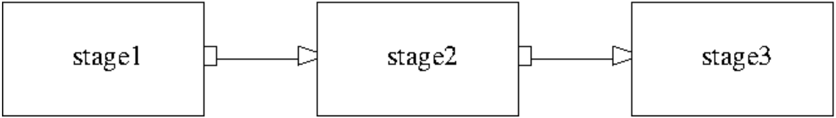
Note : `DBC` = Design By Contract

- Number of lines of code (including tests)

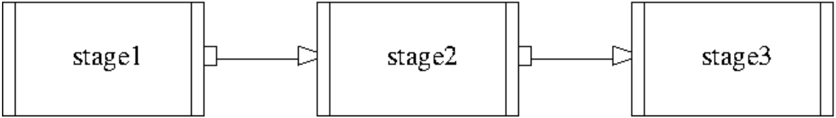
	KLOC
`Debug`, `DBC`	0.1
`FF`	1.3
Test cases	3.4

Pipes

A. An “Hello world !” example : Graphical representation



A. An “Hello world !” example : Graphical representation



A. An “Hello world !” example : Detailed test case

```
it "runs a simple three stage pipeline" do
  stage1 = FF.node(source: true) do |_task, ff|
    1.upto(10) { |i| ff.send_out i }
    :EOS
  end

  stage2 = FF.node { |task|  task }

  output = []
  stage3 = FF.node(sink: true) do |task|
    output << task
    :GO_ON
  end

  FF.pipe(stage1, stage2, stage3).run_and_wait_end

  output.must_equal [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
end
```

A. An “Hello world !” example : Detailed test case

```
it "runs a simple three stage pipeline" do
  stage1 = FF.node(source: true) do |_task, ff|
    1.upto(10) { |i| ff.send_out i }
    :EOS
  end

  stage2 = FF.node { |task|  task }

  output = []
  stage3 = FF.node(sink: true) do |task|
    output << task
    :GO_ON
  end

  (stage1 | stage2 | stage3).run_and_wait_end

  output.must_equal [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
end
```

A. An “Hello world !” example :
Detailed test case with unbounded channels

```
it "runs a simple three stage pipeline" do
  stage1 = FF.node(source: true) do |_task, ff|
    1.upto(10) { |i| ff.send_out i }
    :EOS
  end

  stage2 = FF.node { |task|  task }

  output = []
  stage3 = FF.node(sink: true) do |task|
    output << task
    :GO_ON
  end

  FF.pipe(stage1, 0, stage2, 0, stage3).run_and_wait_end

  output.must_equal [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
end
```

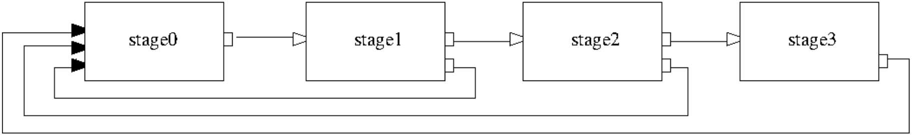
A. An “Hello world !” example :
Detailed test case with bounded channels

```
it "runs a simple three stage pipeline" do
  stage1 = FF.node(source: true) do |_task, ff|
    1.upto(10) { |i| ff.send_out i }
    :EOS
  end

  stage2 = FF.node { |task|  task }

  output = []
  stage3 = FF.node(sink: true) do |task|
    output << task
    :GO_ON
  end

  FF.pipe(stage1, 1, stage2, 1, stage3).run_and_wait_end

  output.must_equal [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
end
```

B. Fancy pipeline : Graphical representation from FastFlow Tutorial

```
/* ******************************** */
/* ******* fancy_pipeline.cpp ***** */

/*
 *   Stage0 -----> Stage1 -----> Stage2 -----> Stage3
 *   ^   ^ ^            |              |              |
 *    \   \ \------------              |              |
 *     \   \-------------------------------           |
 *      \------------------------------------------------
 */
```

B. Fancy pipeline : Alternative graphical representation



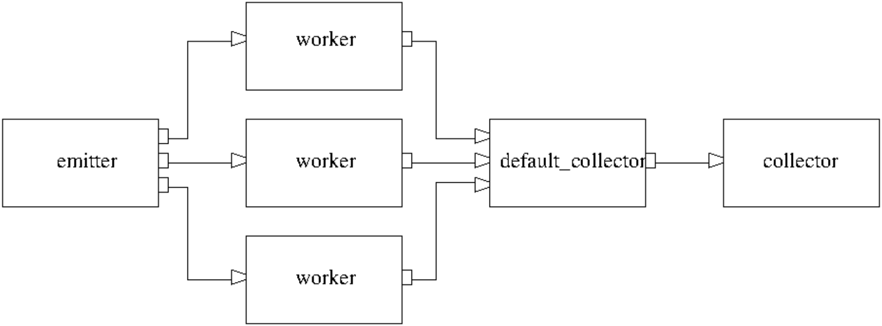
Note : Black input ports are feedback ports.

B. Fancy pipeline : Ruby code

```
stage0 = FF.node(source: true) ...
stage1 = FF.node ...
stage2 = FF.node ...
stage3 = FF.node(sink: true) ...

pipe = ((( stage0 |
           stage1 ).wrap_around |
           stage2 ).wrap_around |
           stage3 ).wrap_around
```

Farms

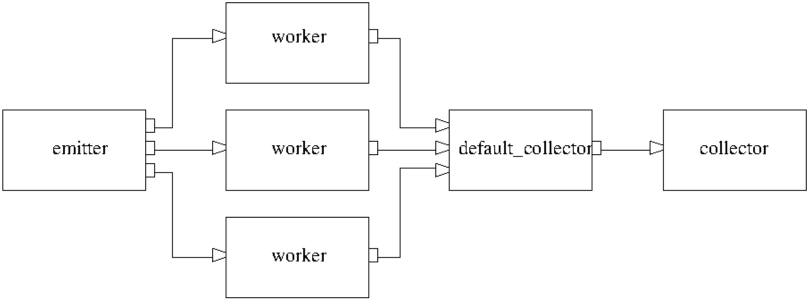
C. A farm with the default collector : Graphical representation



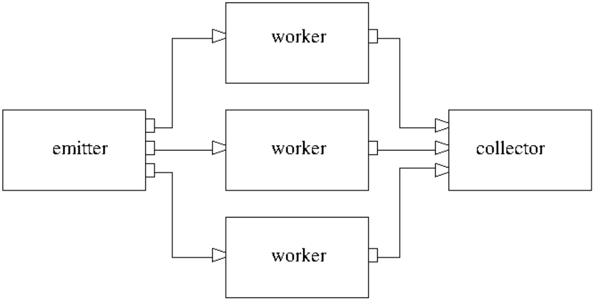
C. A farm with with the default collector : Ruby code

```ruby
emitter = ...
worker = ...

farm = FF.farm( emitter: emitter,
                worker: worker,
                nb_workers: 3 )


output = []
collector = FF.node(sink: true) do |task|
  output << task
  :GO_ON
end

(farm | collector).run_and_wait_end
```

C. A farm with the default collector : Graphical representation



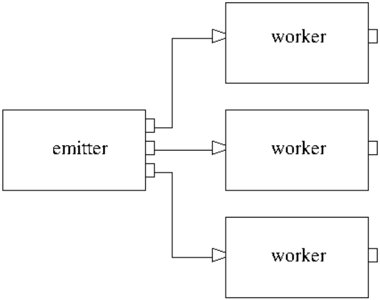
D. A farm with an explicit collector : Graphical representation



D. A farm with an explicit collector : Ruby code

```
FF.farm( emitter: emitter,
         worker: worker,
         nb_workers: 3,
         collector: collector ).run_and_wait_end
```

E. A farm with no collector : Graphical representation



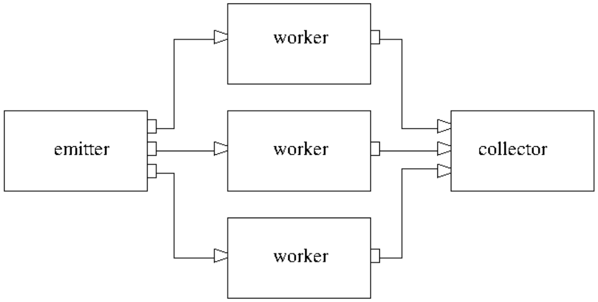
E. A farm with no collector : Ruby code

```
farm = FF.farm( emitter: emitter,
                worker: worker,
                nb_workers: 3,
                collector: :none )
```

E. A farm with no collector used in a pipe : Ruby code

```
farm = FF.farm( emitter: emitter,
                worker: worker,
                nb_workers: 3,
                collector: :none )


collector = FF.node(sink: true) ...

(farm | collector).run_and_wait_end
```

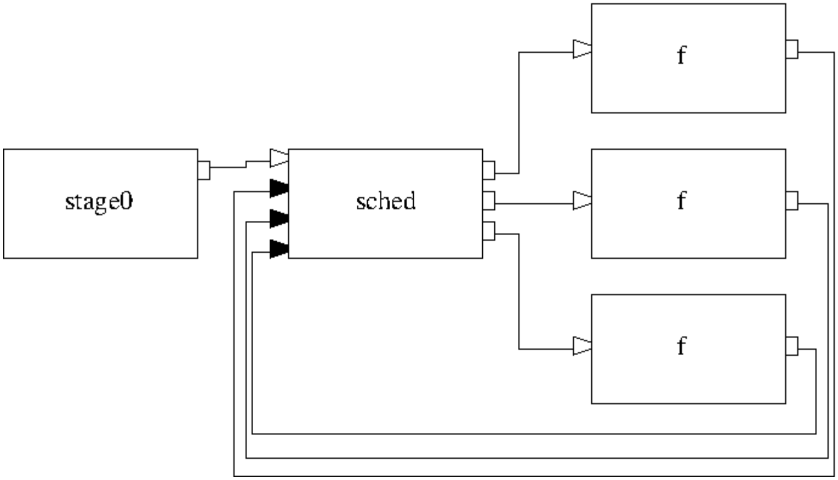
E. A farm with no collector used in a pipe : Graphical representation



F. A farm with feedback :
Graphical representation from FastFlow Tutorial

```
/* ****************************** */
/* ******* feedback.cpp ***** */

/*
 *                  _____________
 *                 |             |
 *                 |     --> F --
 *                 v    |    .
 *   Stage0 --> Sched   |    .
 *                 ^    |    .
 *                 |     --> F --
 *                 |             |
 *                  _____________
 */
```

F. A farm with feedback : Alternative graphical representation



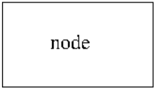
F. A farm with feedback : Ruby code

```
stage0 = FF.node(source: true) { ... }

eos_notifier = proc { |ff| ff.broadcast :EOS }
sched = FF.node( eos_notify: eos_notifier ) do |task, ff|
  ff.current_input_port.feedback? ? :GO_ON : task
end

f = proc { |task| task }

farm = FF.farm( emitter: sched,
                worker: f,
                nb_workers: 3,
                collector: :none  )

( stage0 | farm.wrap_around ).run_and_wait_end
```

Additional “features”

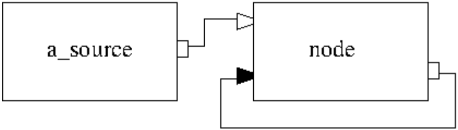
G. A node by itself can be run

```
it "can run a single node, even if it has no port" do
  res = []
  node = FF.node(source: true, sink: true) do
    1.upto(5) do |i|
      res << i
    end
    :GO_ON
  end

  node.run_and_wait_end
  res.must_equal [1, 2, 3, 4, 5]
end
```



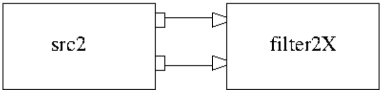
H. A node can be wrapped-around itself and it can have an arbitrary `input_port_selector`

```
nb = -1
ips = proc { |ff| nb += 1; nb % 2 }

node = FF.node( sink: true,
                input_port_selector: ips ) do |task|
  ...
end

( a_source | node.wrap_around ).run_and_wait_end
```



I. Nodes with multiple inputs/outputs can be piped and the ports are adapted (if possible)

```
src2 = FF.node source: true, nb_outputs: 2 do |_t, ff|
  1.upto(5)  { |i| ff.send_out i, 0 }
  6.upto(10) { |i| ff.send_out i, 1 }
  :EOS
end

filter2X = FF.node do |task, ff|
  puts "Task from " + ff.current_input_port
  task
end

( src2 | filter2X | ... )
```



J. The input (resp. output) port selector can be specified in various ways

```
FF.node( ...,
         input_port_selector: :strict_round_robin )

# Default
FF.node( ...,
         input_port_selector: :pseudo_round_robin )



selector_method = proc { |ff| ... ff.inputs[i].empty? ... }
FF.node( ...,
         input_port_selector: selector_method )


FF.node( ...,
         input_port_selector: :dataflow )
```

K. A form of dataflow scheduling is available

```
emitter = FF.node source: true, nb_outputs: 2 do |_, ff|
  1.upto(5) do |i|
    ff.send_out i,   0        # 1, 2, 3, 4, 5  go to c0
    ff.send_out i+5, 1        # 6, 7, 8, 9, 10 go to c1
  end
  :EOS
end

adder = FF.node input_port_selector: :dataflow do |task|
  task[0] + task[1]
end

output = []
collector = FF.node(sink: true) { |x| output << x }

( emitter | adder | collector ).run_and_wait_end

output.must_equal [7, 9, 11, 13, 15]
```

Next steps

Next steps

- Implement freezing and thawing (?)

Freezing and thawing : some news from Montreal

FROID EXTRÊME

DES CENTAINES DE CONDUITES GELÉES À MONTRÉAL

L'hiver 2015 est particulièrement froid. Le mercure est tombé sous la barre des - 20 °C à 21 reprises depuis le début de l'année. C'est deux fois plus qu'à pareille date, l'hiver dernier. Montréal a eu une température moyenne de - 15,8 °C en février, contre - 9 °C l'an dernier.

Next steps

- Implement freezing and thawing (?)

- Run some “real programs” (with medium/coarse grain tasks) and see if some speed up can be obtained (?)

- Look at macro dataflow examples
- Find a way to express dynamic macro dataflow networks

Remarks ? Questions ?