~~Guild Implementation~~
Ractor report

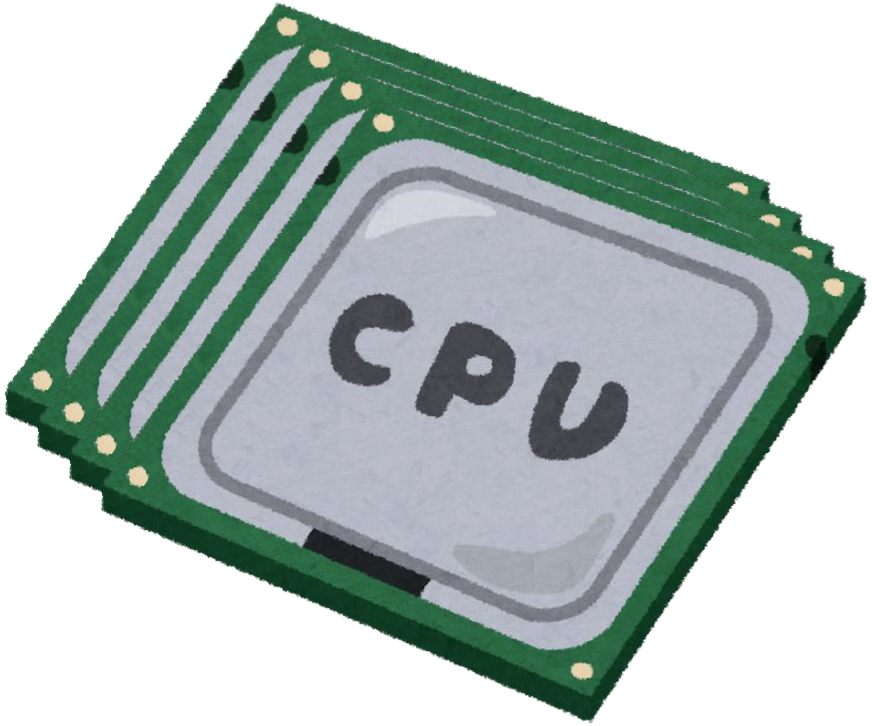
Koichi Sasada
Cookpad Inc.

Communication with me

- I will check tweets with “#ractor” hashtag on Twitter
- I’m at ruby-jp slack workspace, #concurrency

Background
Parallel programming

- Parallel execution on Multi-core CPUs is important
- Multi-process programming is not easy
  - Hard to communicate
  - Hard to control resource consumption
- Multi-thread doesn't support parallel execution on MRI



Background
Concurrent **Thread** programming is hard

- Required: Appropriate synchronization for threads
  - Threads can share everything
- Difficult debugging on non-deterministic nature
  - Data race
  - Race condition
  - Dead/live locking

Goal:
Easy and Parallel concurrent programming on Ruby

Our proposal:
Ractor – an Actor-like concurrent abstraction

Memory model: Limiting object sharing

Good communication API

“Guild” → “Ractor”

- Basic concept was proposed with “Guild” code name at RubyKaigi 2016 and 2018
  - http://rubykaigi.org/2016/presentations/ko1.html
  - https://rubykaigi.org/2018/presentations/ko1.html
- With Matz, we discussed the name of Guild and decided to change the class name from **Guild** to **Ractor** (Ruby’s Actor-like).

Ractor
Concepts

- Multiple Ractors in an interpreter process
- Limited object sharing
- Two-types communication between Ractors
- Copy & Move semantics to send messages

- Details: https://github.com/ko1/ruby/blob/ractor_parallel/doc/ractor.md

Ractor
Concept: Parallel execution

- Multiple Ractors in an interpreter process
  - **Ractors run in parallel**
  - `Ractor.new{ expr }` makes new Ractor
  - Ractor has at least 1 Ruby threads, and threads in a Ractor can not run in parallel (~2.7 compatible)

Ractor
Concept: Limited object sharing

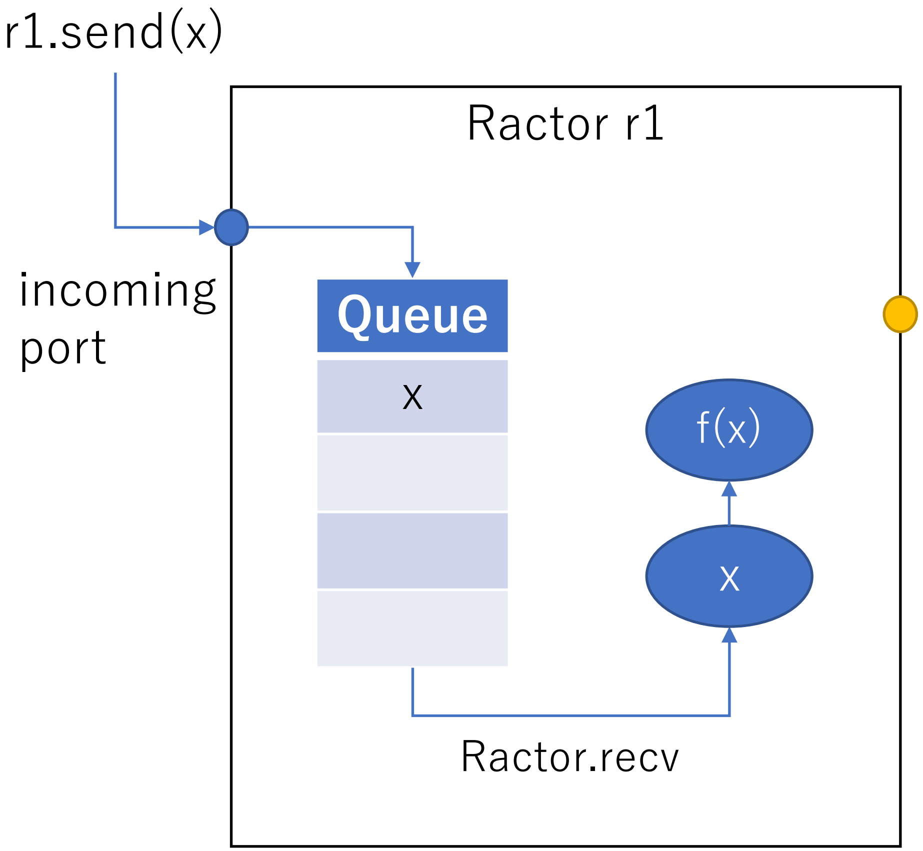
- Strictly separate objects into shareable and unshareable
  - Unshareable objects – most objects are **unshareable**
  - Sharable objects – special objects
    - Immutable objects (== frozen objects which refer shareable objects)
    - Class/module objects
    - Special shareable objects (Ractor objects etc.)
- Avoid data races and race conditions
  - **Most of objects** are unshareable objects
  - Shareable objects require appropriate synchronization by the interpreter or programmer

Ractor
Concept: Communication/synchronization

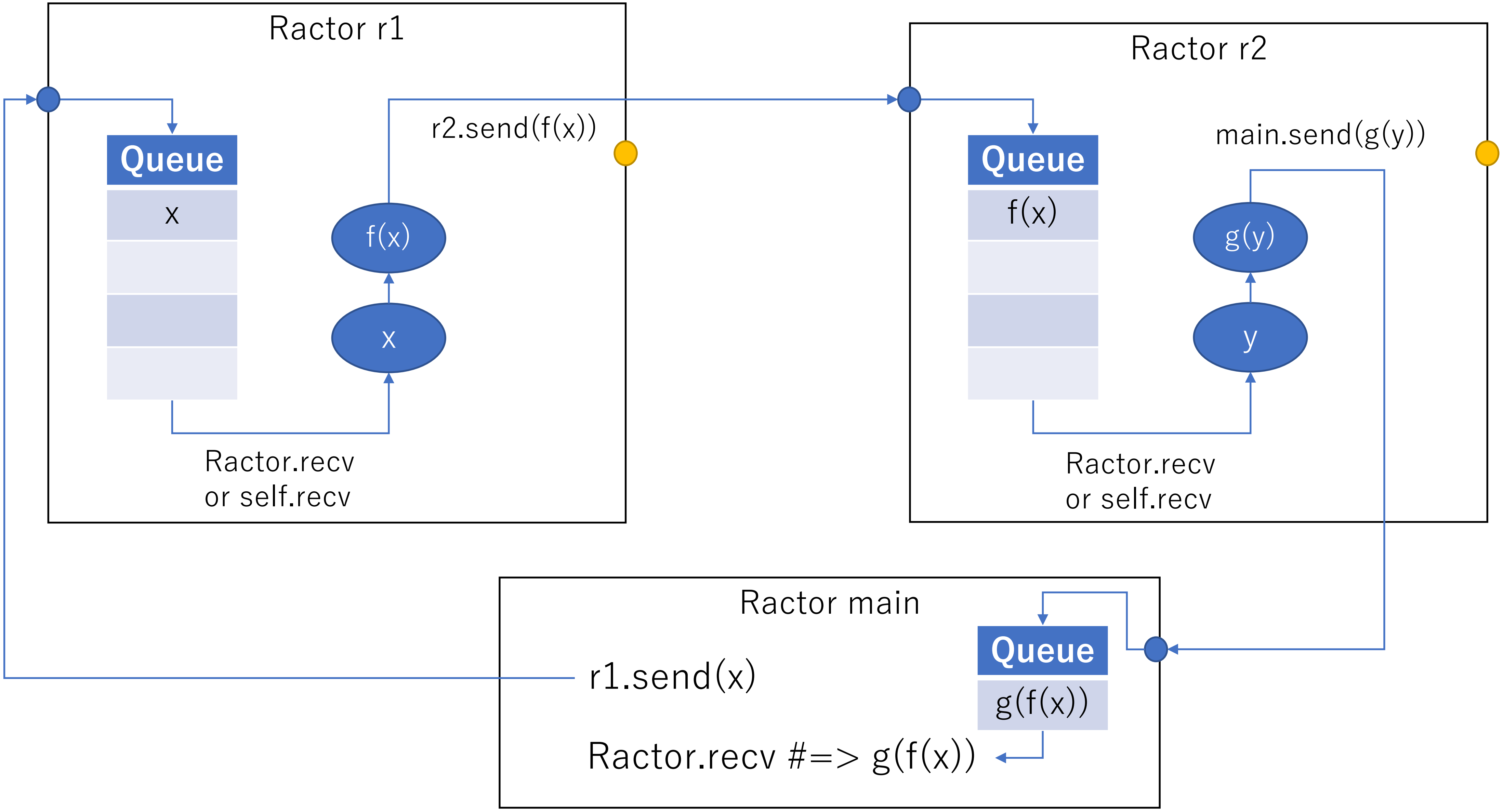
- Two-types communication between Ractors
  - Push type: Actor-**like** send/receive object transferring
    - **`Ractor#send(obj)`** and **`Ractor.recv`** pair
    - Sender knows receiver ractor (**dst.send(obj)**)
  - Pull type: Passive message passing style object transferring
    - **`Ractor.yield(obj)`** and **`Ractor#take`** pair
    - Receiver knows a sender Ractor (**`src.take`**)
- Copy & Move semantics to send messages
  - Passed objects will be copied (deep copy)
  - Move mode is also supported (shallow copy)
    - After moving, moved objects can’t be touched by sender Ractor

Ractor
Push/Active message passing

- Actor-like communication
  - Sender knows receiving Ractor
  - Receiver does not know sending Ractor
- Each Ractor has a queue which connected to the incoming port.
  - `r1.send(x)` enqueues x into the queue
    - Queue is unlimited queue, so non blocking
  - `Ractor.recv` dequeues queued x
    - Block if there is no queued objects

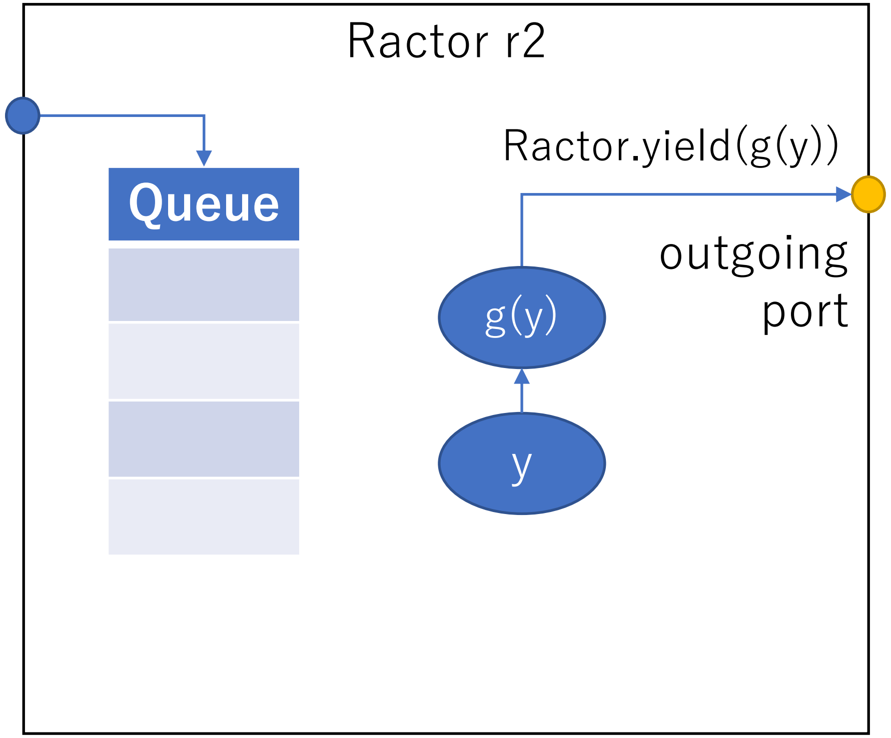


Pipeline with Traditional Actor model

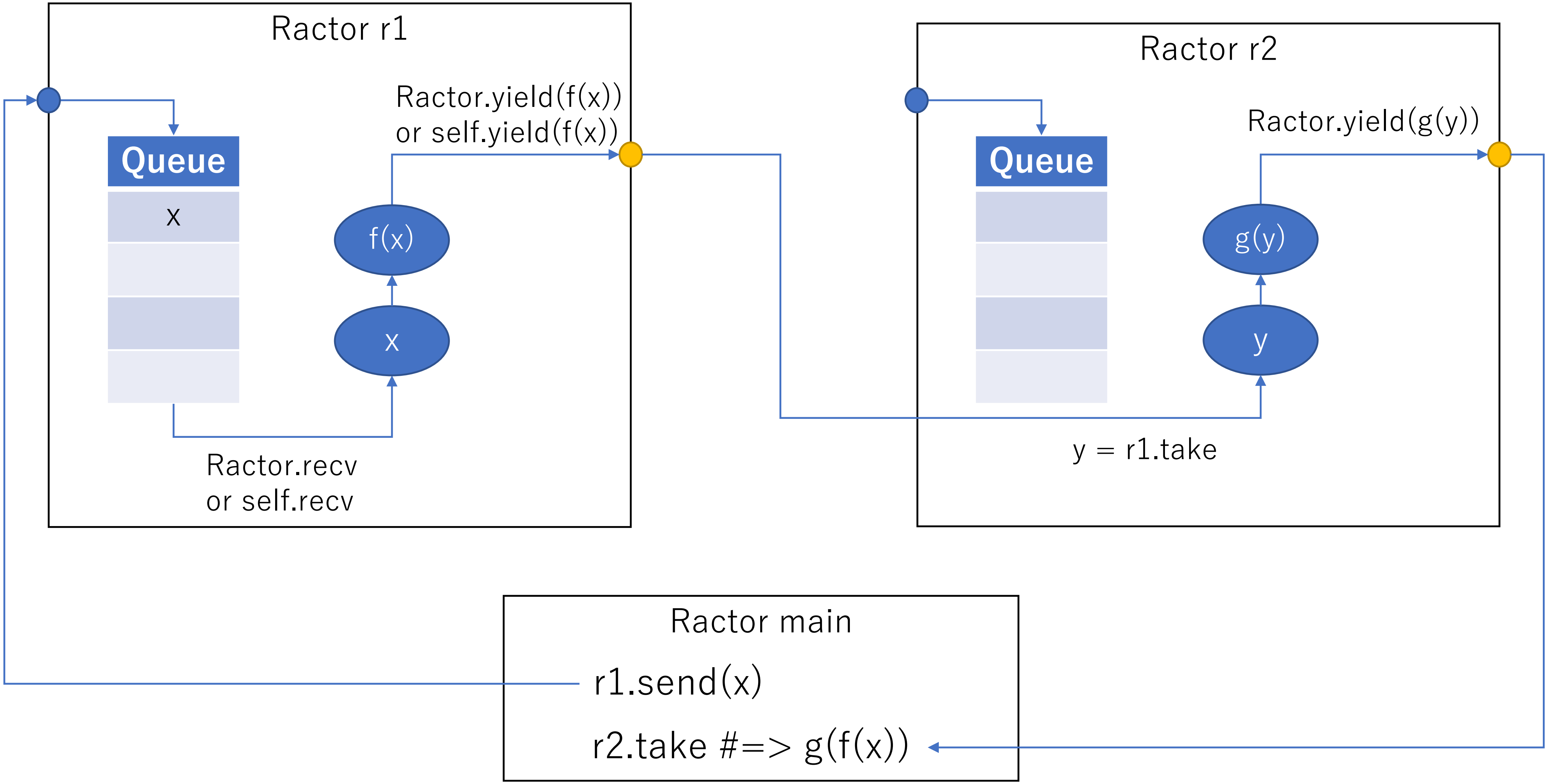


Ractor
Pull/Passive message passing

- Pull type communication
  - Sender does not know receiver
  - Receiver knows sender
- Each Ractor has outgoing port.
  - `Ractor.yield(y)` puts y on outgoing port
  - `r2.take` get y from r2's outgoing port
  - These methods will block until another Ractor take/yield → Rendezvous synchronization
- Block value of given block for `Ractor.new` will be returned by `Ractor.yield` implicitly → `Ractor#take` can supervise the Ractor's liveness.



Pipeline with yield/take



Ractor r1

Queue

x

f(x)

x

Ractor.yield(f(x))
or self.yield(f(x))

Ractor.recv
or self.recv

Ractor r2

Queue

g(y)

y

Ractor.yield(g(y))

y = r1.take

Ractor main

r1.send(x)

r2.take #=> g(f(x))

written in code…

```
r1 = Ractor.new do
  x = Ractor.recv
  Ractor.yield(f(x))
end
r2 = Ractor.new r1 do |r1|
  y = r1.take
  Ractor.yield(g(y))
end
r1.send(:x)
something()
r2.take #=> g(f(:x))
# parallel execution
#  something()
#  f() and g()
```

Ractor.yield and Ractor#take Similarity with Fiber

Fiber

```
f = Fiber.new do
  Fiber.yield 1
  Fiber.yield 2
  3
end
f.resume #=> 1
f.resume #=> 2
f.resume #=> 3
```

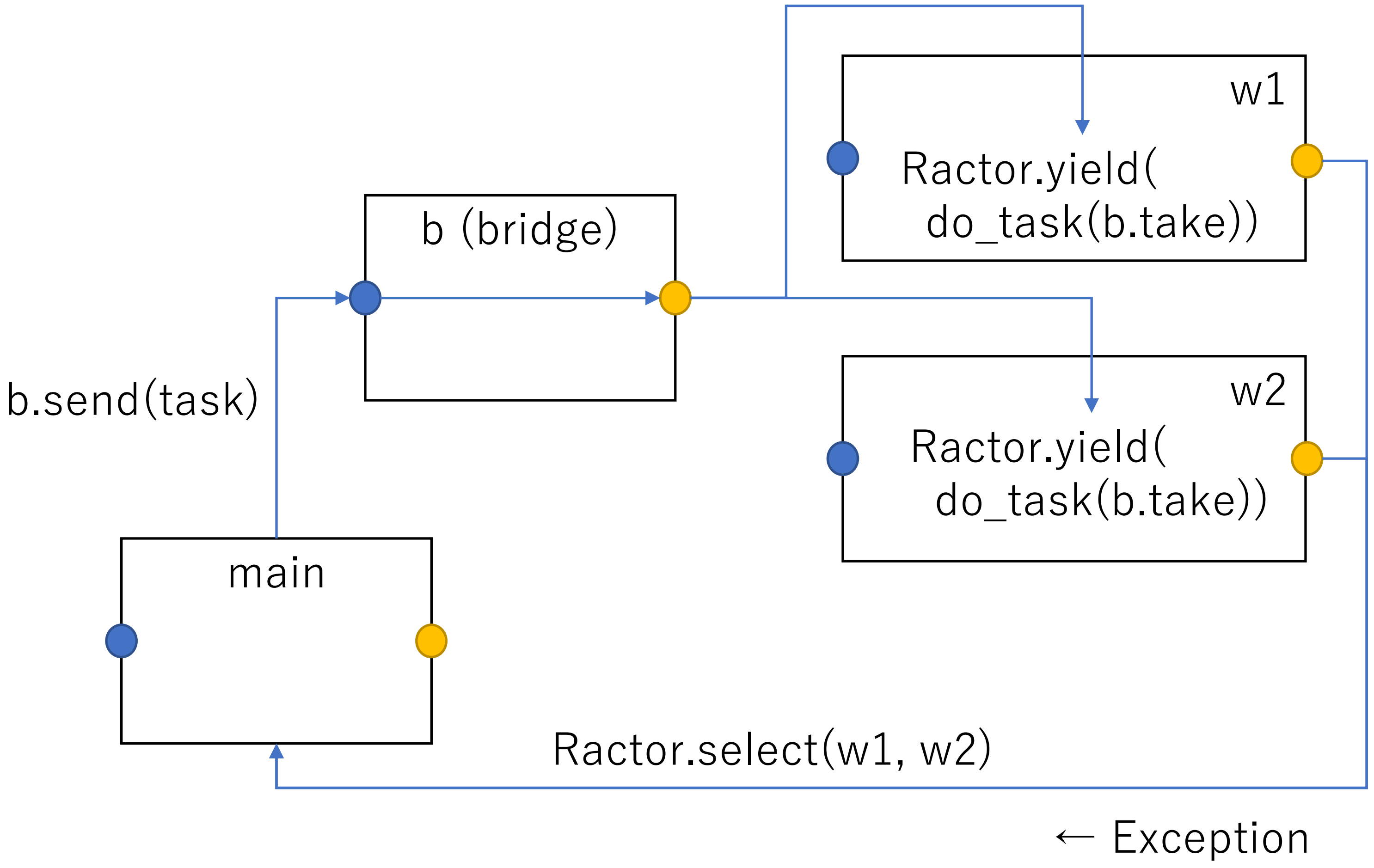
Ractor

```
r = Rator.new do
  Ractor.yield 1
  Ractor.yield 2
  3
end
r.take #=> 1
r.take #=> 2
r.take #=> 3
```

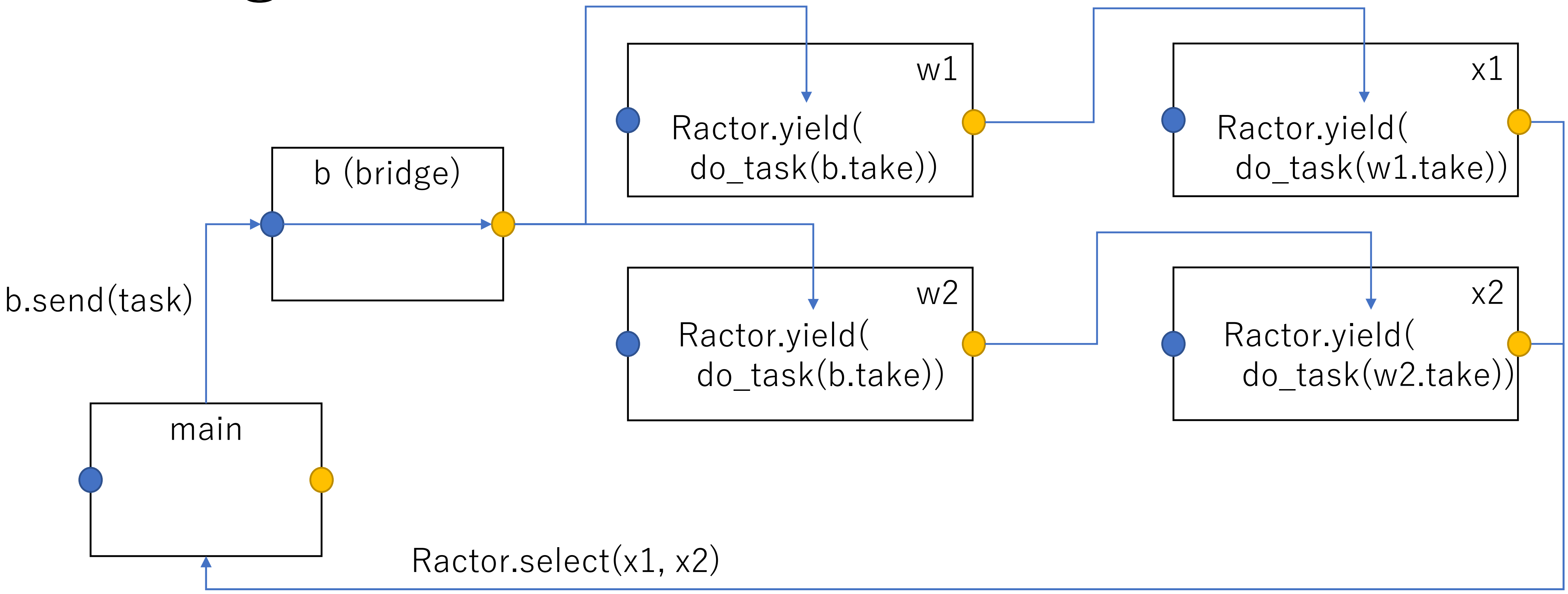
Ractor
Ractor#select

- `Ractor.select(r1, r2, …)` will wait from r1, r2, …
  - Similar to Go’s select statement
  - API can be improved more
    - For example: Event register approach such as Concurrent-ruby’s channel

Load-balancing multi-workers with a bridge Ractor



Load-balancing multi-workers with a bridge Ractor



incoming port/outgoing port

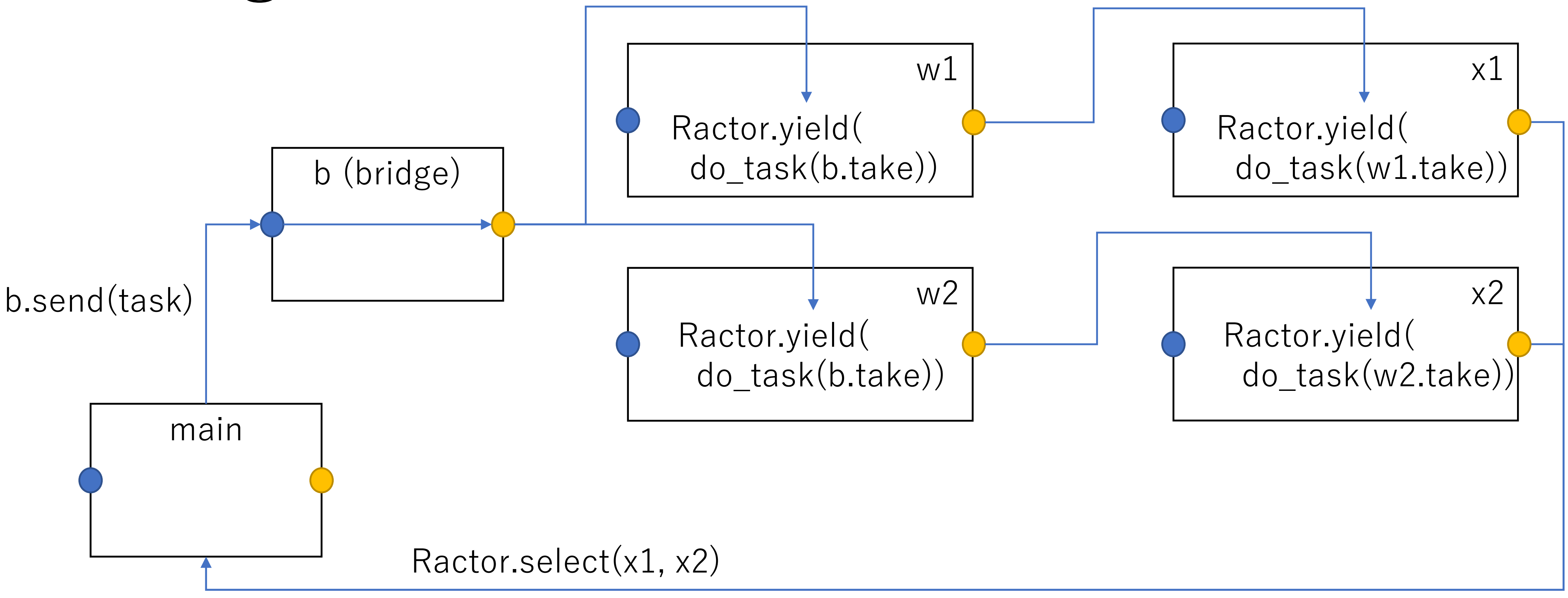
- Two ports
  - incoming port
    - Connected to the incoming queue
    - Sent message is put to the queue
  - outgoing port
    - Yielded message will be put
- They can be closed
  - close_incoming
    - Ractor#send raises an error if incoming port is closed
    - Ractor.recv raise an error if incoming queue is empty and port is closed
  - close_outgoing
    - Ractor#take raises an error if outgoing port is closed
    - Ractor.yield raise an error if outgoing port is closed
  - When Ractor terminates, both ports are closed automatically

Ractor
Supervise Ractors

- `Ractor#take` can supervise Ractors
  - This method can check return value of Ractor’s given block (`Ractor.new{ … }`) and **Block’s exception**.

  → `Ractor.select(r1, r2, …)` can supervise r1, r2, …
- Compare with other languages
  - Erlang: link to other process and death event will be notified to the linked process.
  - Go: causes panic on unexpected goroutine’s termination
  - Ruby (Ractor): Ractor.select(r1, r2, …) can supervise them

Load-balancing multi-workers with a bridge Ractor



b (bridge)

w1

Ractor.yield(
 do_task(b.take))

w2

Ractor.yield(
 do_task(b.take))

x1

Ractor.yield(
 do_task(w1.take))

x2

Ractor.yield(
 do_task(w2.take))

b.send(task)

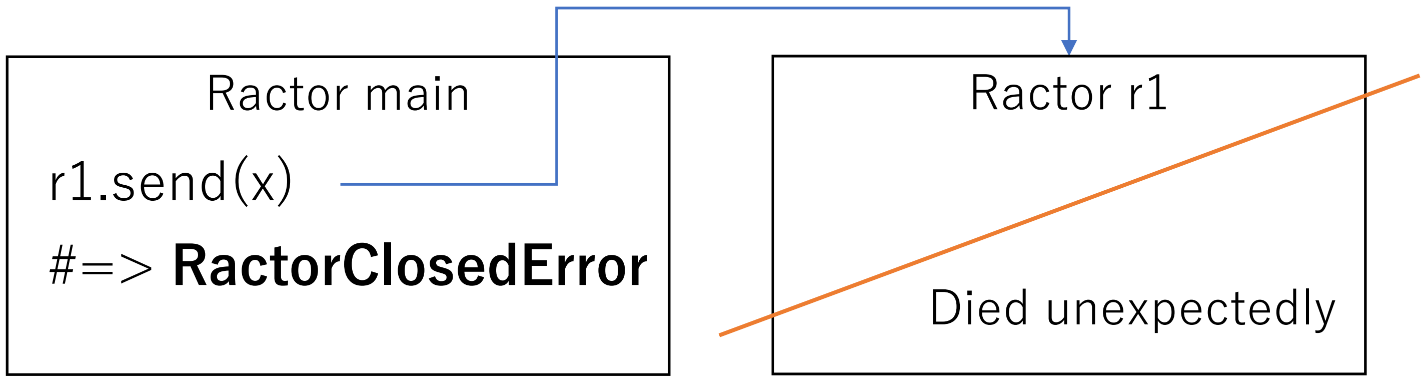
main

Ractor.select(x1, x2)

← Exception

Advantage of Actor-like based approach compare with channel-based approach

- Easy error detection
  - If receiver Ractor is died, the error will be occurred
  - Channel-based approach, we can’t detect destination side-Ractor’s termination without a trick (ex: close channel’s port in ensure clause)



```
r1.send(x)
#=> RactorClosedError
```

Ractor
Message passing options

- Reference
  - Shareable objects will be sent by reference (pointer)
- Copy: `Ractor#send(obj)`, `Ractor.yield(obj)`
  - Objects will be **deep** copied
- Move: `Ractor#send(obj, move:true)`, `Ractor.yield(obj, move:true)`
  - Shallow copy
    - Long string
    - IO (File, Socket, …)
  - Source Ractor can not touch moved objects (will cause exception)

Ractor Creation

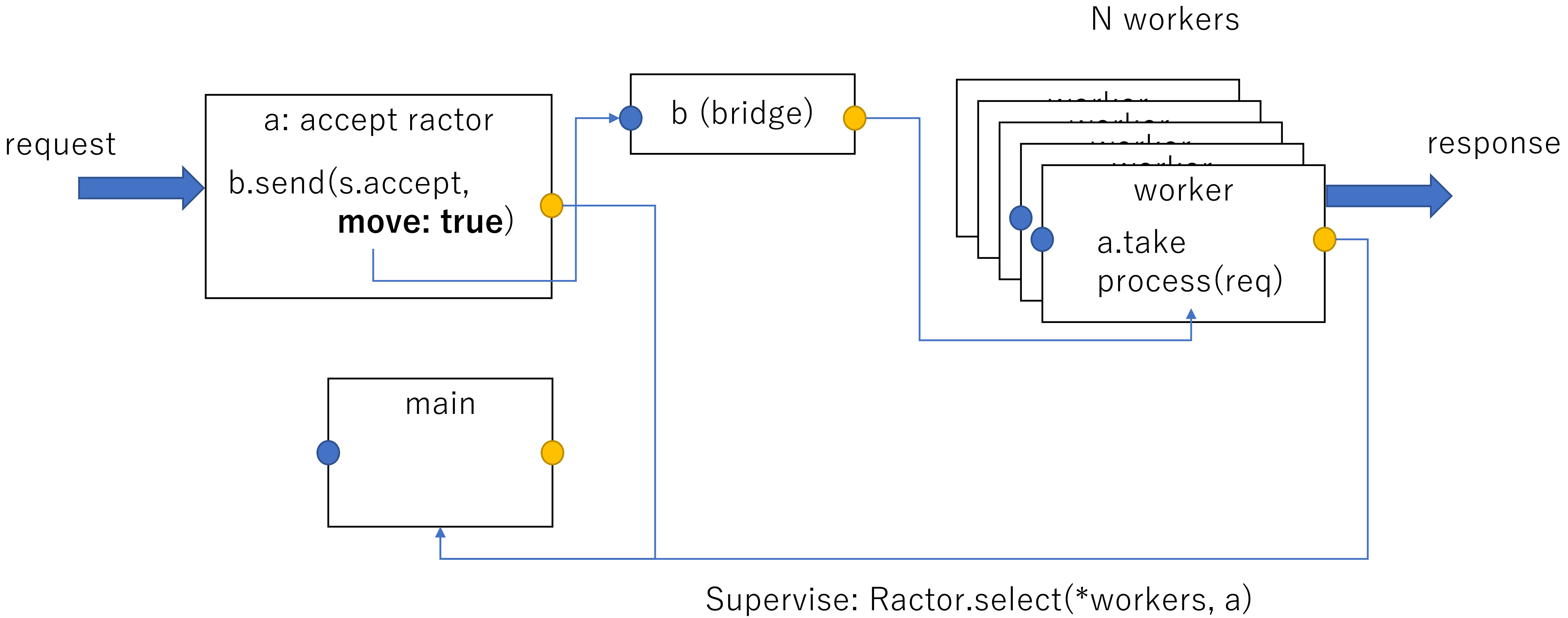
- `Ractor.new{ expr }` will create new Ractor and execute `expr` in parallel with other Ractors
- If `expr` contains reference to the outer-variables, it will be error
  - ex) `a = [1]; Ractor.new{ p a } #=> Error`
- Self of given block will be its Ractor object
- Block parameters will be sent block arguments
  - ex)

    ```
    Ractor.new([1]){|a| p a}
    #=> r = Ractor.new{a = Ractor.recv; p a}
    #   r.send([1])
    ```

Ractor
Semantic changes

- 100% compatible if only main Ractor is used
- Limited to main Ractor (first Ractor)
  - Global variables $gv
    - Some gvars ($stdout, …) will be Ractor local
  - Class variables @@cv
  - Instance variables of shareable objects
    - Ivars of class/module are prohibited
  - Constants refer to unshareable objects
    - `C = [1]` is prohibited
- For Ractor programming, many modifications are needed

Ractor
Example: Web application server



Ractor progress

- https://github.com/ko1/ruby/blob/ractor_parallel/
  - ☑ Basic Ractor API seems working
  - ☑ Ruby apps without Ractor can work (compatible w/ current)
  - ■ Complex application with Ractor (not enough synchs)
  - ■ Existing Ruby's API considerations
  - ■ C-extension supports
  - ■ Object passing copy/move support (support only a few types)
  - ■ Performance tuning
    - Poor algorithm for Ractor communications
    - TLS tuning
    - Object space tuning

```
$ ./miniruby -e Ractor.new{}
<internal:ractor>:37: warning: Ractor is experimental,
and the behavior may change in future versions of Ruby!
Also there are many implementation issues.
```

Ractor Evaluation

Evaluation
Create/Invoke/wait time comparison for 10k

	WSL2 (Ubuntu 20.04)	Ubuntu 18.04
process	9.608186	36.939180
ractor	**0.526030**	**0.259494**
thread	**0.451909**	**0.137313**
fiber	0.022461	0.020944
proc	0.005264	0.003301

(sec)

TODO: Make Ractors/threads creation faster as fibers (Ruby 3.1~)

https://gist.github.com/ko1/6257532de84cdb4212581c66415155ed

Evaluation
Prime number detection

- Ractor worker example
  - Create several worker ractors
  - Send tasks to them, and aggregate the answer
- Task is “Integer#prime?”
  - `1_000.times{|i| (2**TN + i).prime?}`
  - `TN = 10 to 50`
    - TN = 10 → 1024.prime?, 1025.prime?, …
    - TN = 50 → 1125899906842624.prime?, 1125899906842625.prime?, …

```
require 'prime'

RN = ARGV.shift.to_i
TN = ARGV.shift.to_i
N  = 1_000

if RN == 0
  # sequential program
  ans = N.times.map{|i|
    n = 2 ** TN + i
    [n, n.prime?]
  }
  # pp ans
else
```

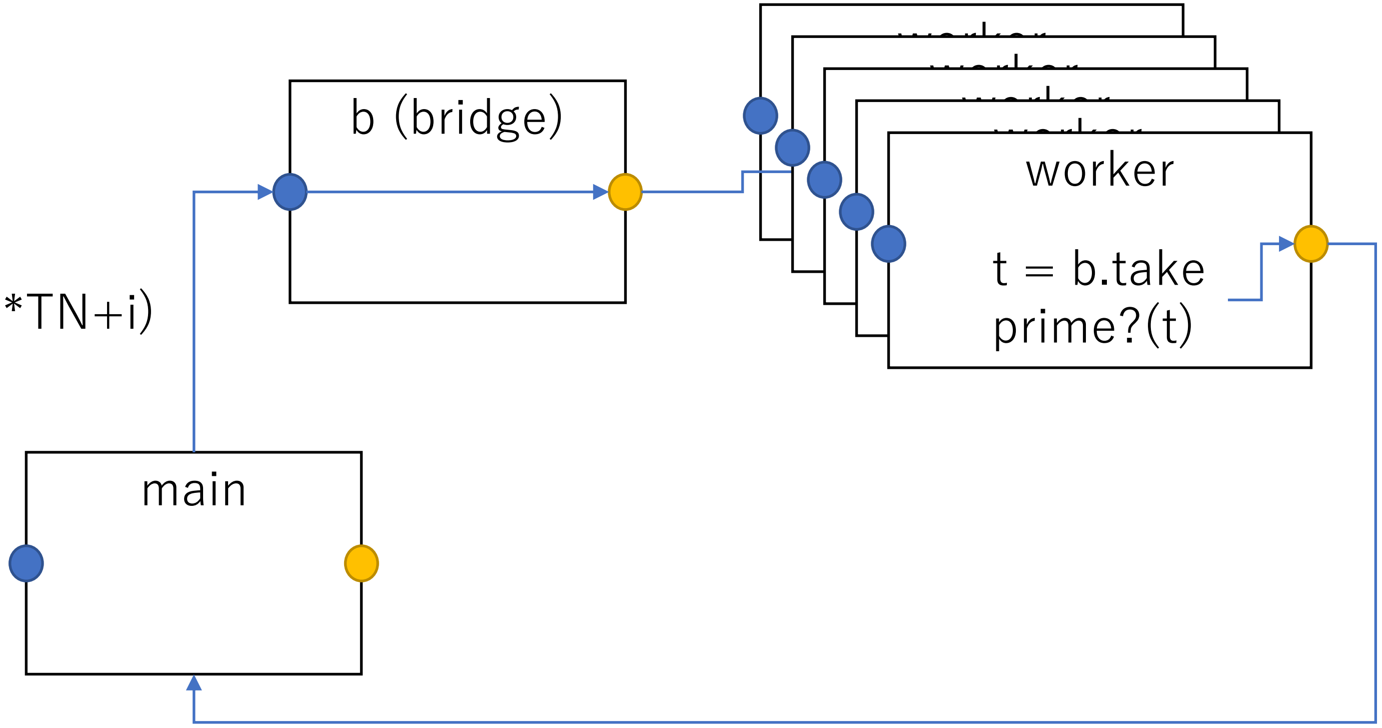
```
# parallel program
pipe = Ractor.new do
  loop do
    Ractor.yield Ractor.recv
  end
end

workers = (1..RN).map do
  Ractor.new pipe do |pipe|
    while n = pipe.take
      Ractor.yield [n, n.prime?]
    end
  end
end

(1..N).each{|i|
  pipe << 2 ** TN + I
}

ans = (1..N).map{
  _r, (n, b) = Ractor.select(*workers)
  [n, b]
}.sort_by{|(n, b)| n}
end
```

https://gist.github.com/ko1/0979898610f33aef921d864e2f936d0b



RN workers

b (bridge)

worker

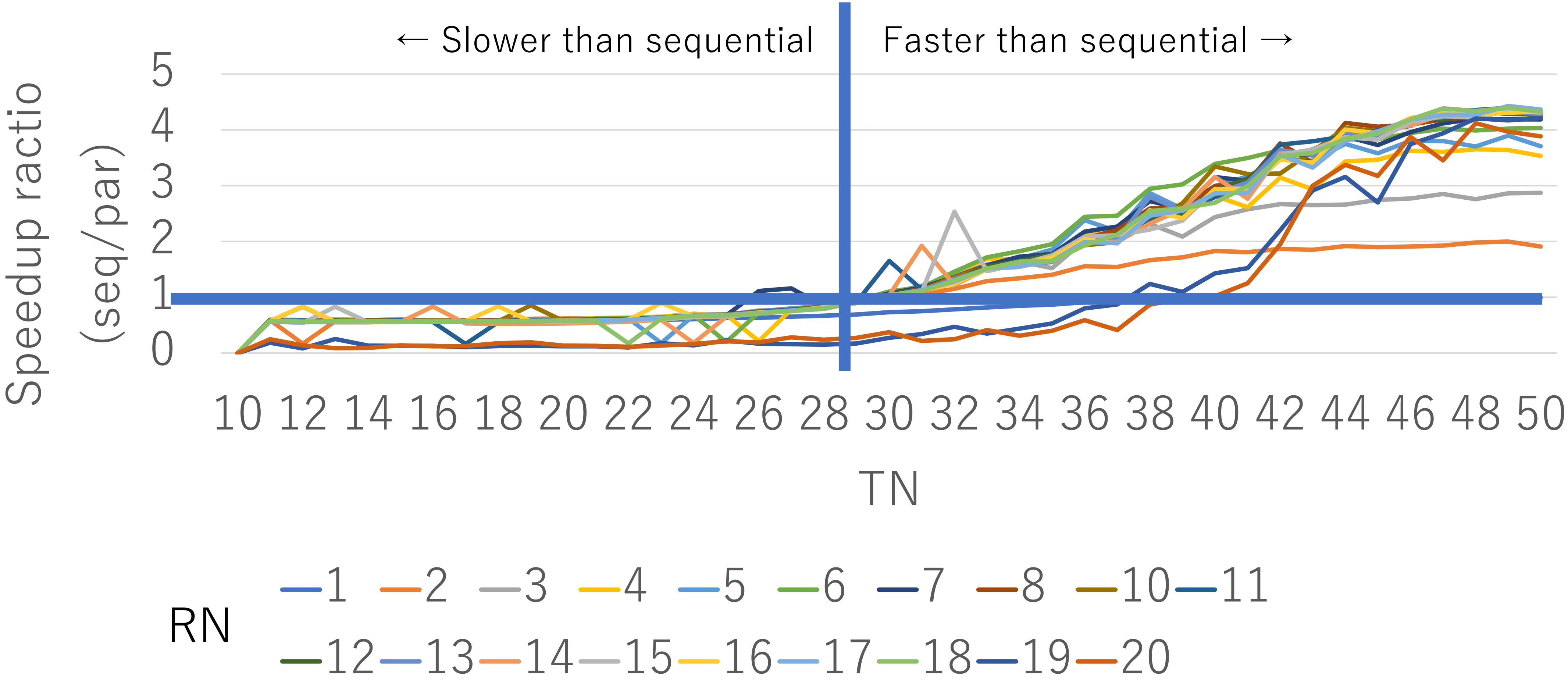
```
t = b.take
prime?(t)
```

```
N.times{|i|
  b.send(2**TN+i)
}
```

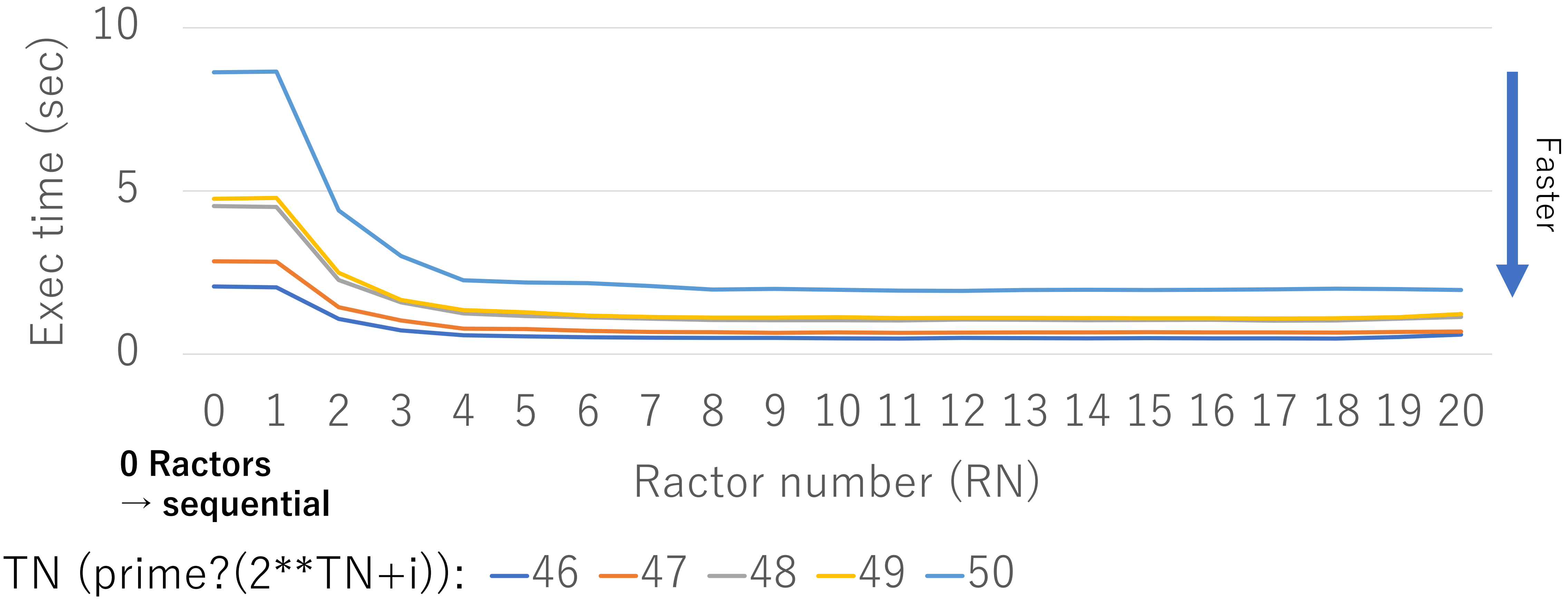
main

Ractor.select(workers)

Evaluation result on 4 core 8 threads machine

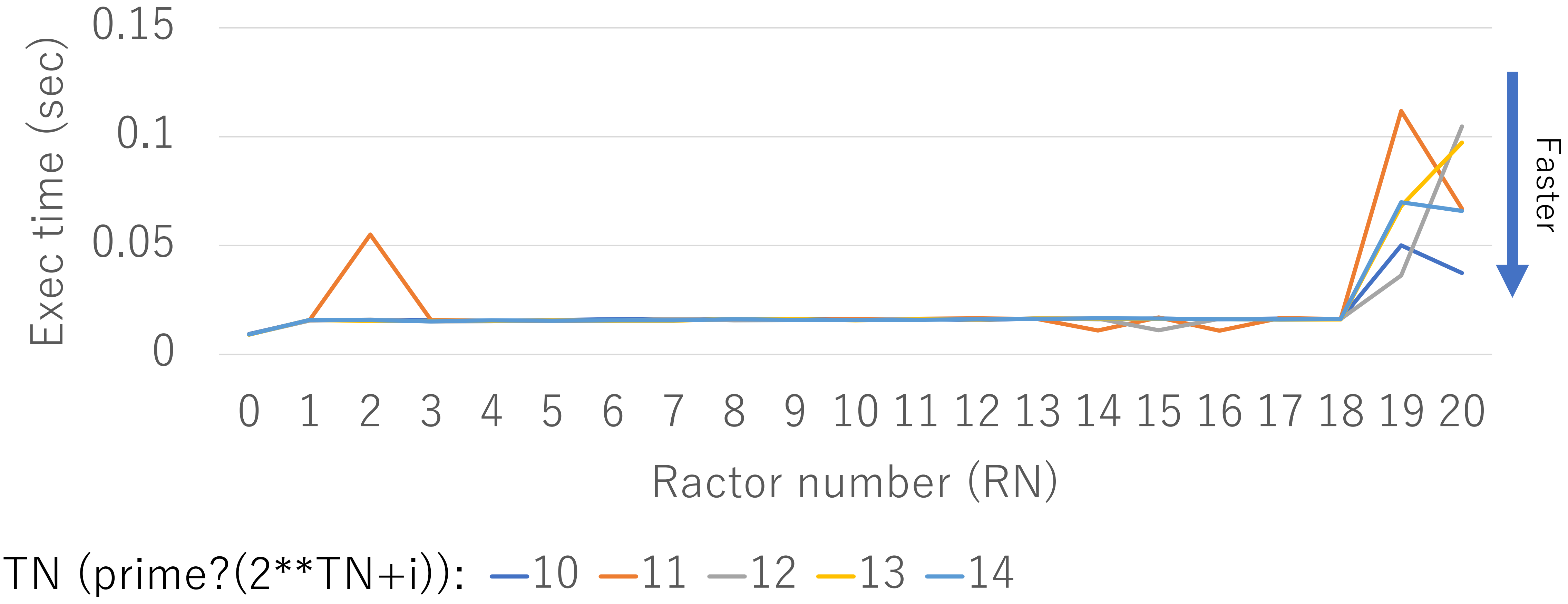


Evaluation result on 4 core 8 threads machine



0 Ractors
→ sequential

Evaluation result on 4 core 8 threads machine



Conclusion

- Ruby program can run in parallel with Ractor without thread-safety headache
- Ractor API and implementation is not matured, but we are working on it for Ruby 3