Guild → Ractor

ささだこういち

Ruby 3 さみっと 2020/04/17

背景

- Ruby の 1 プロセスでは（基本的には）並列処理できない
  - 同時に複数 CPU を使う処理
  - Ruby（MRI）のThread == 並行処理
  - マルチプロセスは難しそう…
- そもそも Thread 難しい
  - 適切な同期漏れ
    - データレース
    - レースコンディション
    - デッドロック、ライブロック
  - **再現性がなくデバッグが困難**

簡単に
並行・並列処理
したい！

間違いを起こさないためには？

- スレッドを良い感じにサポート
  - 再現性をあげる（OS scheduler に手を入れるなど）
  - デバッグサポートを行う（Thread Sanitizer, helgrind, etc）
- データの書き換えを禁止
  - Erlang など
- データ（オブジェクト）を共有しない
  - 同期が不用
  - でも、プログラミングは大変

目標

- 原則オブジェクトの共有を「できない」並行・並列機能の提供
- 良い感じにプログラミングできる API の提供

Guild

- Guild という名前で、RubyKaigi 2016 でコンセプトを発表
  - 複数のGuildを生成
  - それぞれの Guild（に所属する Thread）は並列に実行
    - （main）Guild内では完全に Ruby2 互換
  - Guild間で共有可能なオブジェクトに強い制限
    - 共有可能オブジェクト：通信用オブジェクト、Immutable オブジェクトなど
    - 共有不可能オブジェクト：ふつうのオブジェクト
    - まぁつまり、普通はオブジェクトが共有不可 → 共有による問題無し
  - 共有不可能オブジェクトを送る場合、複製か移動
    - 複製：ディープコピー
    - 移動：ヘッダだけコピー（軽量）、ただし送信元では使えない
    - 移動は所有権の移動みたいな感じ、移籍→Guild

なまえもんだい

matz「移動って使うの？」
ko1「滅多に使わないんじゃないすかね、普通はコピーで」
matz「じゃあGuildって名前よくないんじゃない？」
ko1（いいと思ったんだけどな…）

改名

- Guild → Ractor: Ruby's Actor
- API をどうするか悩んでいて、Actor ぽい API にふるかー、って勢いで名前を決めた
  - 対抗は CSP（チャンネルを用いた通信）
  - 実際は Actor というにはちょっと機能が足りないので、あくまで Ruby's Actor、もしくは Restricted Actor の略としたい
- 問題点：Reactor と似ている
  - 似たような領域だけど何も関係ないコンセプトだから紛らわしい
  - 造語なので、間違えて Reactor って書いちゃう人居そう
  - でも、たくさん書いてたら手が慣れちゃったよ

API

https://github.com/ko1/ruby/blob/ractor/ractor.ja.md

実装を試したければ、
https://github.com/ko1/ruby/tree/ractor
（いろいろ完全ではない）
（ただし、並列化はまだ対応していない（GVLで保護される））

生成

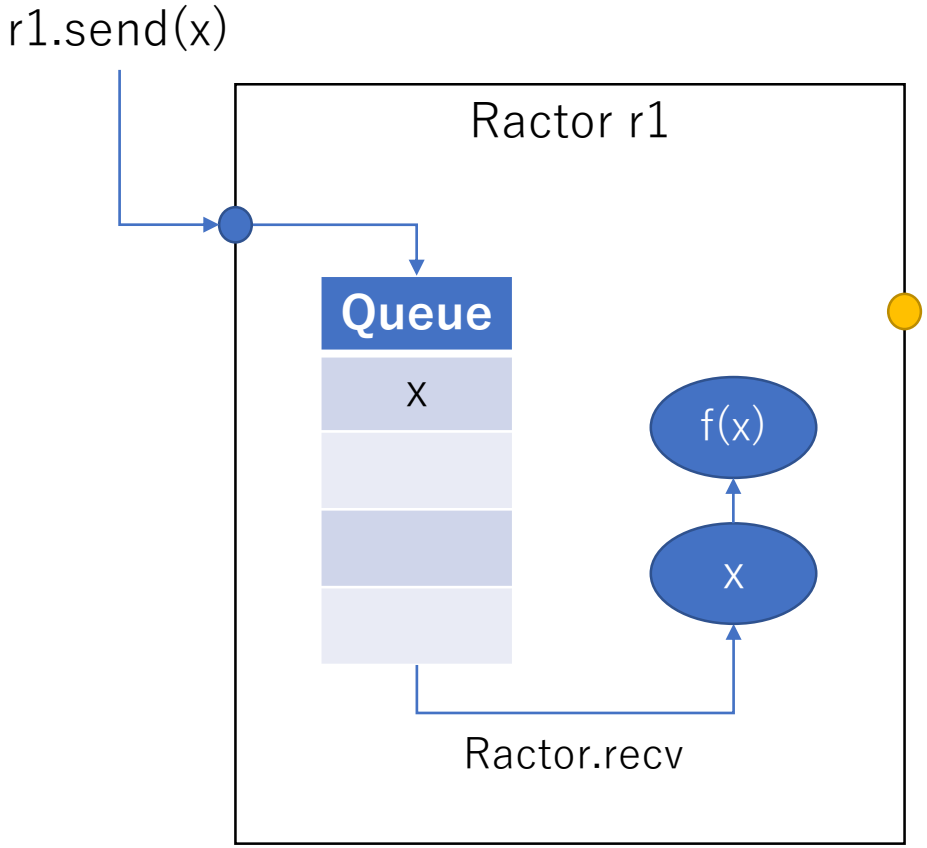
```
r1 = Ractor.new do
  expr1
end
r2 = Ractor.new do
  expr2
end

# r1, r2 を生成
# expr1, expr2 は並行・並列に実行される
```

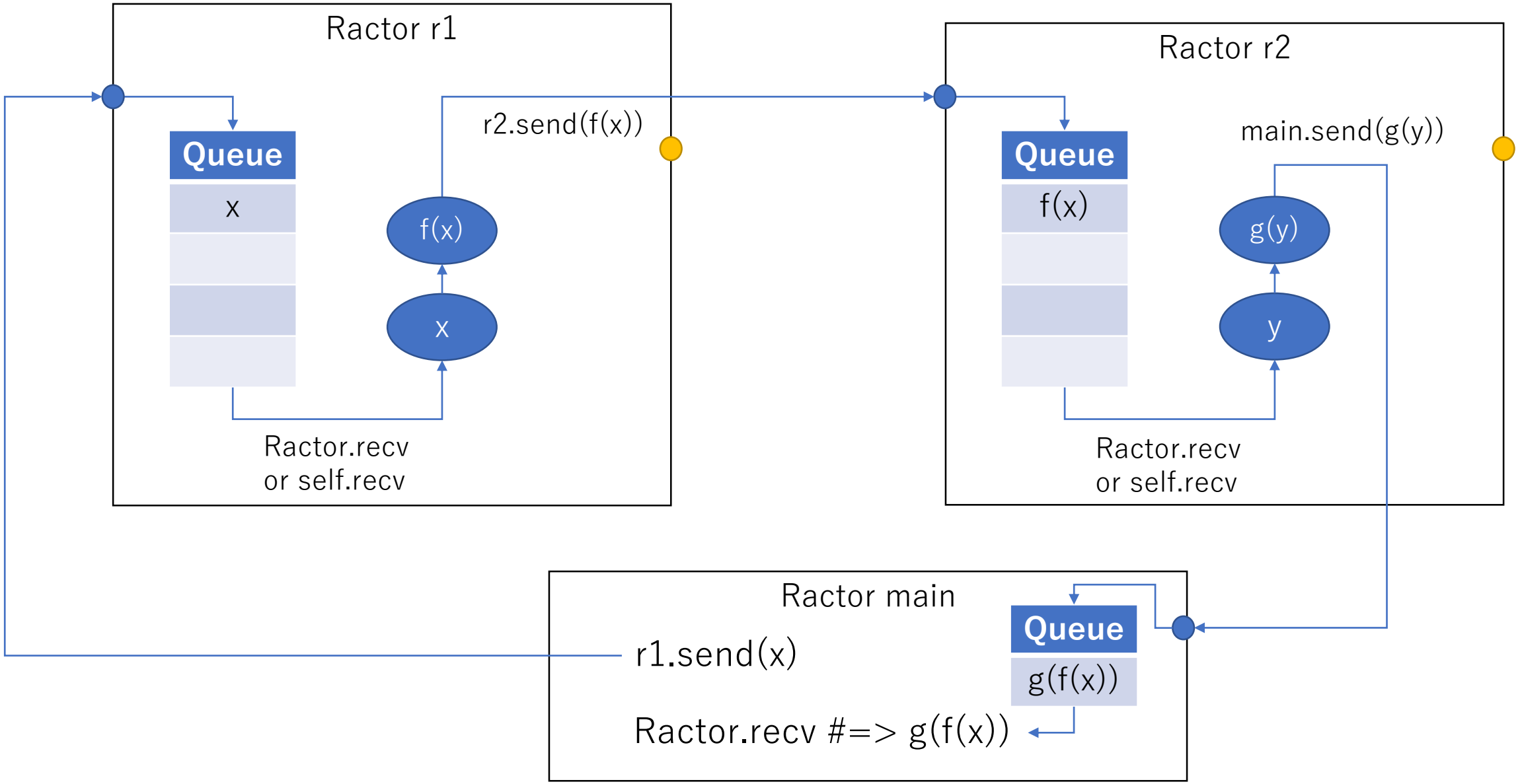
Actor モデルによる通信

- 各RactorはQueueを持つ
- r1.send(x) で r1 のQueueに突っ込む
- Ractor.recv で自Queueから取り出し
  - 何もなければブロック

- Push型通信
  - 送信者は、受信者を指定
  - 受信者は、送信者を知らない

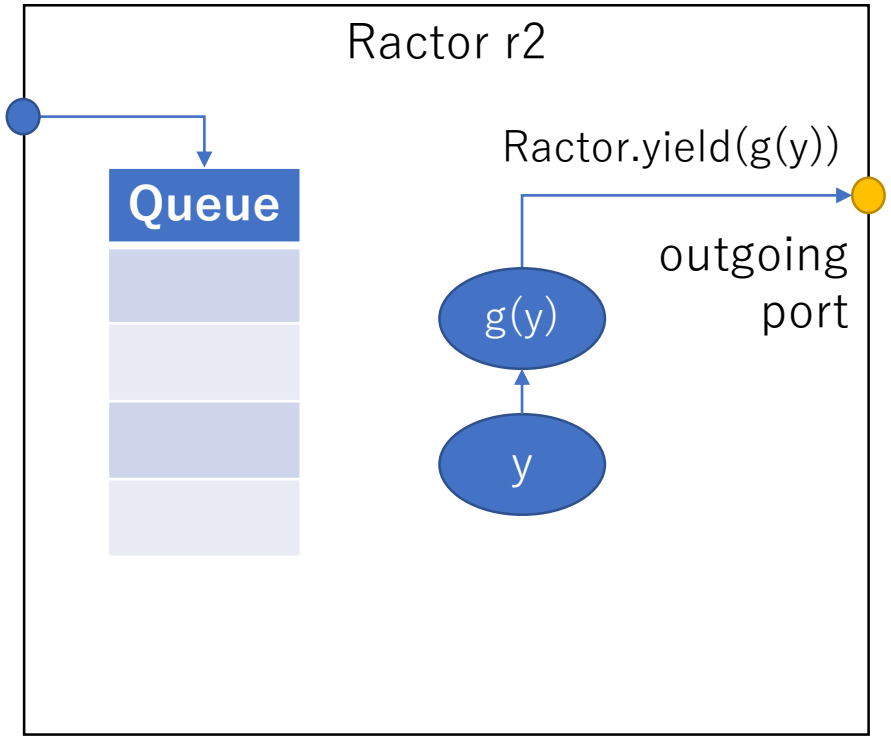


Pipeline with Traditional Actor model



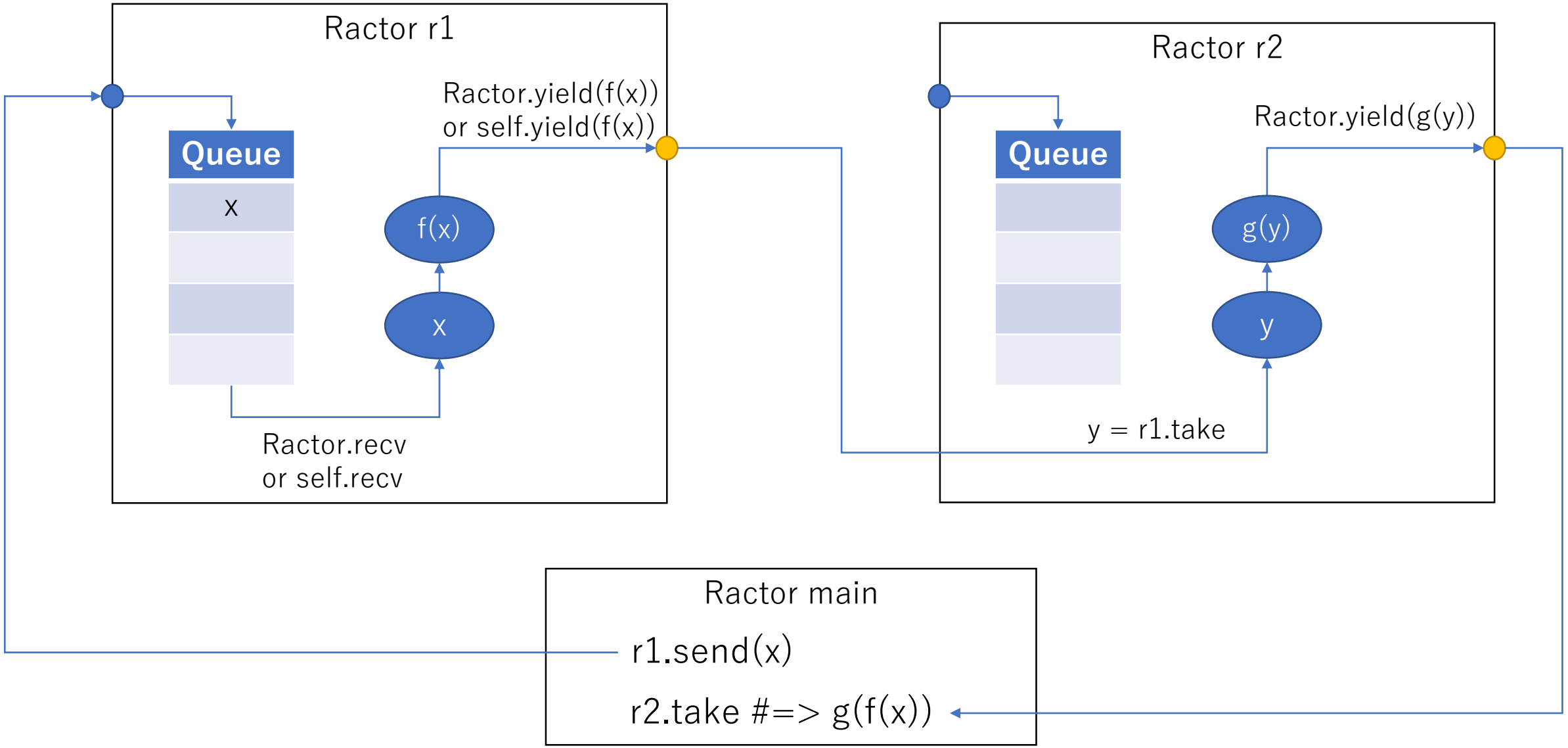
Outgoing port

- Outgoing port という概念を用意
  - Ractorから値を流す専属チャンネル
  - **Ractor.yield(y)** で oport に y を待機
  - **r2.take** で oport で待機している y を取得
  - ブロックの返値は自動的に yield
- ランデブー同期を実現



- Pull型通信
  - 送信者は、受信者を知らない
  - 受信者は、送信者を指定

Pipeline with yield/take



※ yield/take がランデブーポイントになるので、フロー制御が可能

Outgoing port
プログラムで書くと…

```
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)) # something() と g(f(:x)) を並列計算
```

Fiber との類似性

Fiber

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

Ractor

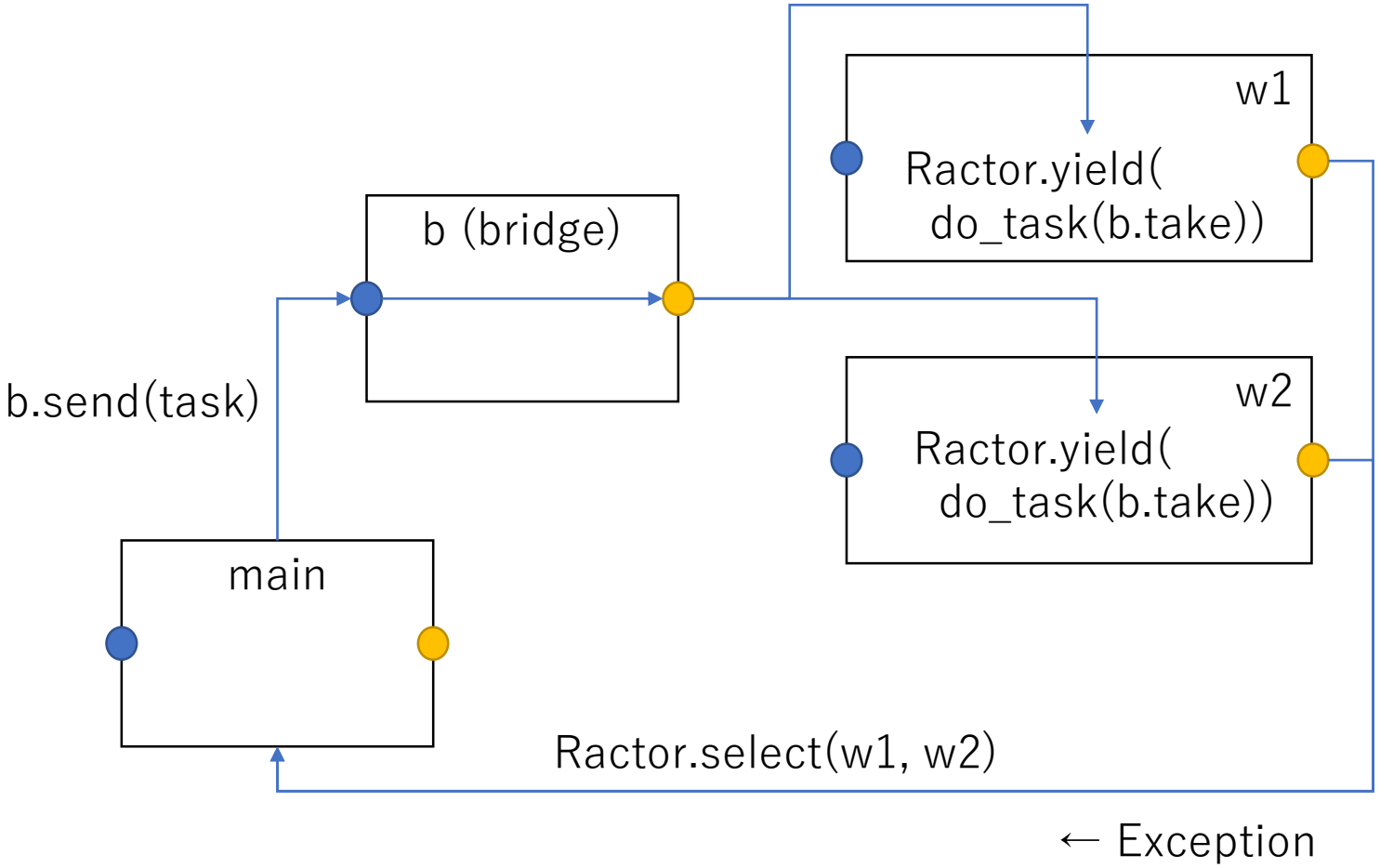
```
r = Rator.new do
  Ractor.yield 1
  Ractor.yield 2
  3
end
r.take #=> 1
r.take #=> 2
r.take #=> 3
```

Select

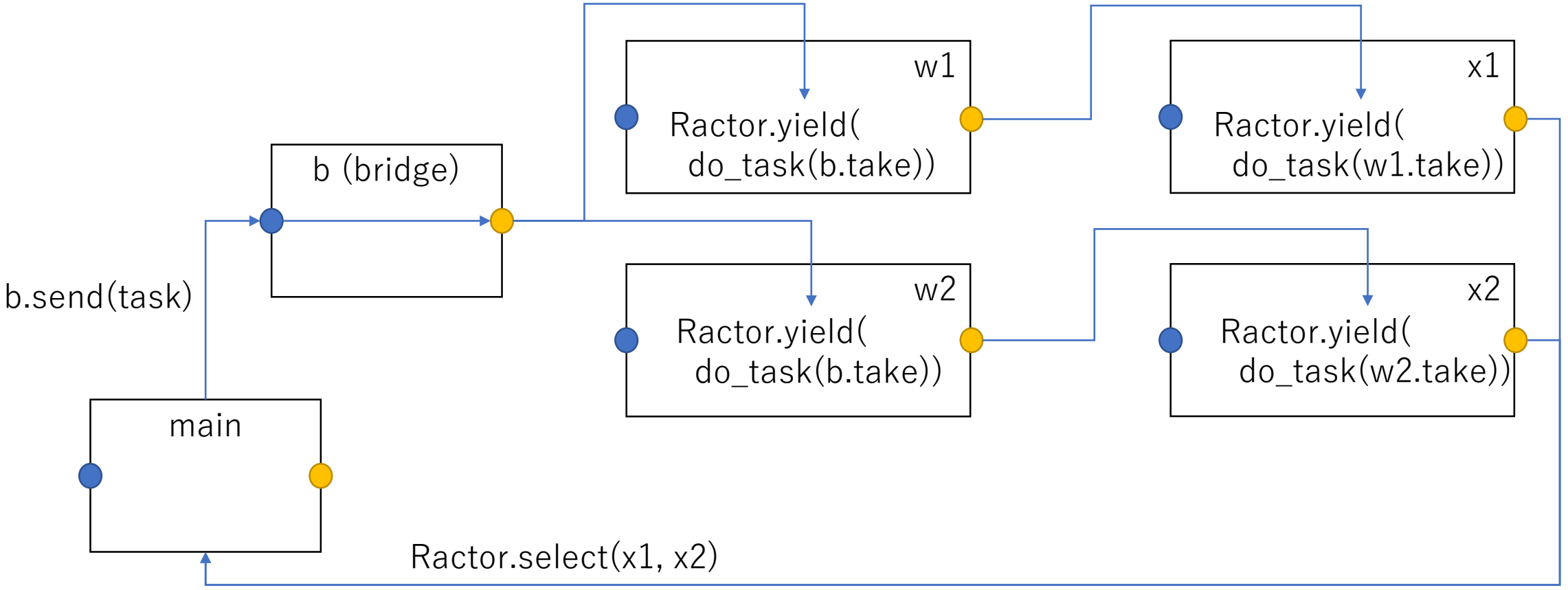
- Ractor.select(r1, r2, …) で、r1, r2, … のどれかから take
  - Go の select 文に相当
  - もっとイベントドリブンなインターフェースの方がいいだろうか？

```
# 他のインターフェース例（Concurrent-ruby Channelと相似）
Ractor.select{|s|
  s.take(r1){ do_something }
  s.take(r2){ do_something }
  s.yield(obj)
}
```

Load-balancing multi-workers with a bridge Ractor



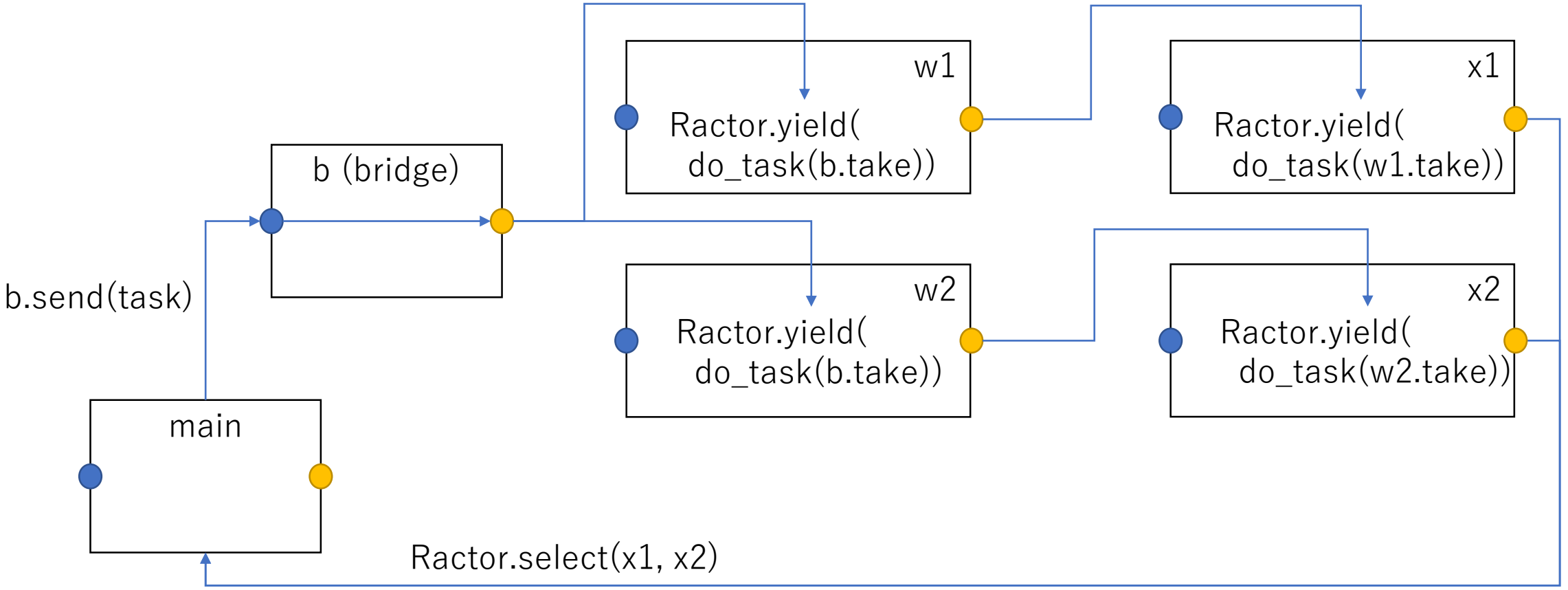
Load-balancing multi-workers with a bridge Ractor



Ractor の監視

- Ractor#take を用いた Ractor の監視
  - ブロックの返値は oport で取得可能
  - 例外も oport で取得可能
  - つまり、Ractor r1 の終了状態は r1.take で監視可能
  - rs = [r1, r2, …] を監視したければ、Ractor.select(*rs) で可能
- 他言語との比較
  - Erlang における link（死んだら監視Processへ通知）→ 面倒
  - Go は何かあれば panic（全部死亡、Thread.abort_on_exception 相当）
  - Ractor （Ruby）では、**簡単に監視を書きたかった**
    - タスクを送り、結果を取得（駄目なら例外）するという単純なケースを簡単に書ける
    - Actorの仕組みだけだと、監視を記述するのは面倒
    - 戻り先が決まっていない場合は、やっぱり面倒
    - チャンネル（CSP）だと、エラー伝搬が難しかったのでこのモデルに

Load-balancing multi-workers with a bridge Ractor



← Exception

incoming port/outgoing port

- incoming port
  - Queue につながっている
  - send されると、Queueに enqueue される（ブロックしない）
- outgoing port
  - 誰か take するを待つ（ブロックする）
- それぞれ close できる
  - close_incoming
    - Ractor#send がエラー
    - Queue が空の時、Ractor.recv がエラー
  - close_outgoing
    - Ractor#take がエラー
    - Ractor.yield がエラー（ブロックの返値はどうすんだろ…）

送信オプション

- 参照
  - 共有可能オブジェクトは自動的に参照だけ送られる
- 複製（Ractor#send(obj), Ractor.yield(obj)）
  - ディープコピーして送る
- 移動（Ractor#send(obj, move:true), Ractor.yield(obj, move:true)）
  - ヘッダだけコピーして送る
    - 大きな文字列
    - IO (File, Socket, …)
  - 移動したオブジェクトは、送信元がアクセスすると例外

共有可能オブジェクト

- Class, Module
- Ractor
- Immutable objects
  - Frozen であり、共有可能オブジェクトのみ参照（読み込み専用）
  - a = [1, 2, 3].freeze #=> 共有可能
  - b = [1, 2, [3]].freeze #=> b[2] は frozen ではないので駄目
- その他、検討中
  - STMなど同期が必須なデータ構造
    - 同期が必須→安全にアクセス可能。Clojure の書き換え可能オブジェクト
    - 共有可能オブジェクトのみ参照可能

Ractor間でオブジェクトを共有させないために Ractor に渡すブロック

- Proc#isolate
  - ブロックの外側の変数にアクセスできない Proc（Thread と違う）
  - Ractor.new に渡すブロックは Proc#isolate されたものが渡される
  - isolate 呼んだ瞬間にエラー

```
x = 1; Proc.new{ p x }.isolate #=> ArgumentError
```

- Ractor.new に渡したブロックの self は Ractor 自身
  - Ractor.new{ p self } #=> Ractor オブジェクト
  - 生成時の self が共有されてはまずいため
- 引数は send/recv したもの
  - Ractor.new(“foo”){|s| p s} #=> コピーされた “foo”

Ractor間でオブジェクトを共有させないためになんとか変数

- main Ractor でしか使えない（== Ractor なしなら使える）
  - グローバル変数　$gv
  - クラス変数　@@cv
  - 共有可能オブジェクトのインスタンス変数（クラスのivar使うよね？）
  - **共有不可能オブジェクト**が束縛された定数（Const = []）
- どのRactorでも使える
  - 共有可能オブジェクトが束縛された定数（ClassA::ClassB とか…）
- 相当プログラムの変更が必要
  - pp() も使えない（クラスのインスタンス変数を使っているため）
  - ライブラリの対応が必要で、書き換えコストが高い
  - まずは、簡単な時間かかる計算を Ractor で外に出すとかかな？

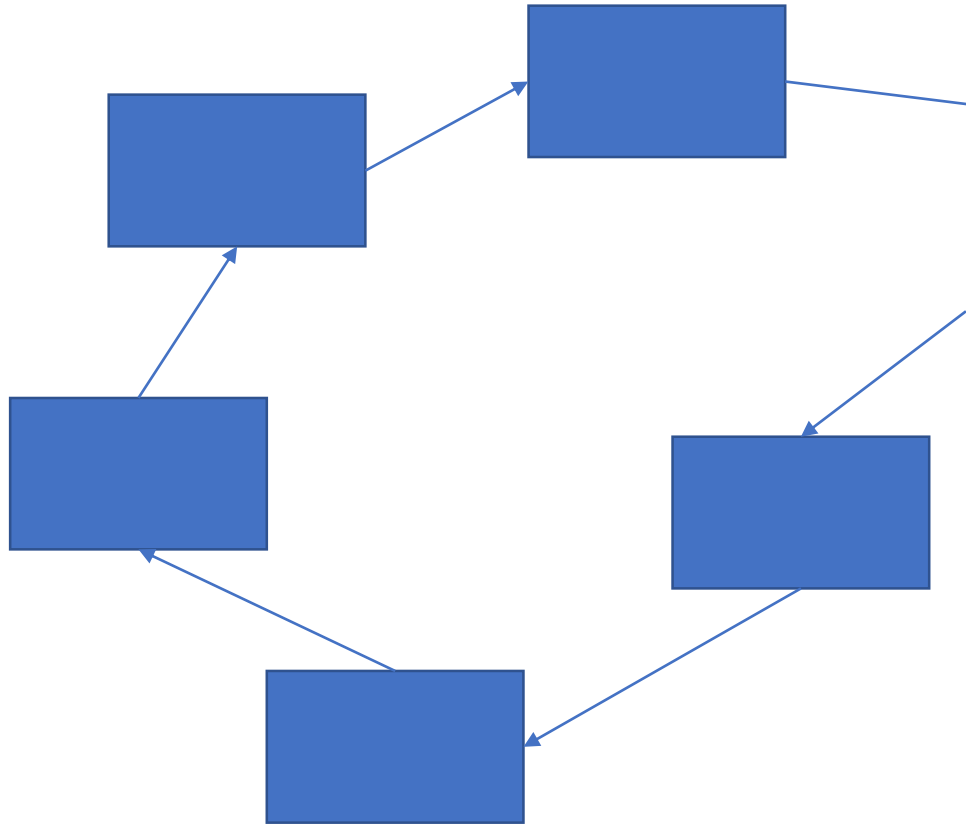
プログラム例
よくある Ring プログラム

```
RN = 10000
CR = Ractor.current

last_r = r = Ractor.new do
  p Ractor.recv
  CR << :fin
end

RN.times{
  r = Ractor.new r do |next_r|
    next_r << Ractor.recv
  end
}

p :setup_ok
r << 1
p Ractor.recv
```



プログラム例
RN個のフィボナッチ数をRN個のRactorで

```
def fib n
  if n < 2
    1
  else
    fib(n-2) + fib(n-1)
  end
end

RN = 10
```

```
rs = (1..RN).map do |i|
  Ractor.new i do |i|
    [i, fib(i)]
  end
end

until rs.empty?
  r, v = Ractor.select(*rs)
  rs.delete r
  p answer: v
end
```

プログラム例
N個の素数判定をRN個のRactorで（pool）

```
require 'prime'

N = 1000
RN = 10

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 << I }
pp (1..N).map{
  r, (n, b) = Ractor.select(*workers)
  [n, b]
}.sort_by{|(n, b)| n}
```

プログラム例
パイプライン2例

send/recv

```
r3 = Ractor.new Ractor.current do |cr|
  cr.send Ractor.recv + 'r3'
end
r2 = Ractor.new r3 do |r3|
  r3.send Ractor.recv + 'r2'
end
r1 = Ractor.new r2 do |r2|
  r2.send Ractor.recv + 'r1'
end
r1 << 'r0'
p Ractor.recv #=> "r0r1r2r3"
```

yield/take

```
r1 = Ractor.new do
  'r1'
end
r2 = Ractor.new r1 do |r1|
  r1.take + 'r2'
end
r3 = Ractor.new r2 do |r2|
  r2.take + 'r3'
end
p r3.take #=> 'r1r2r3'
```

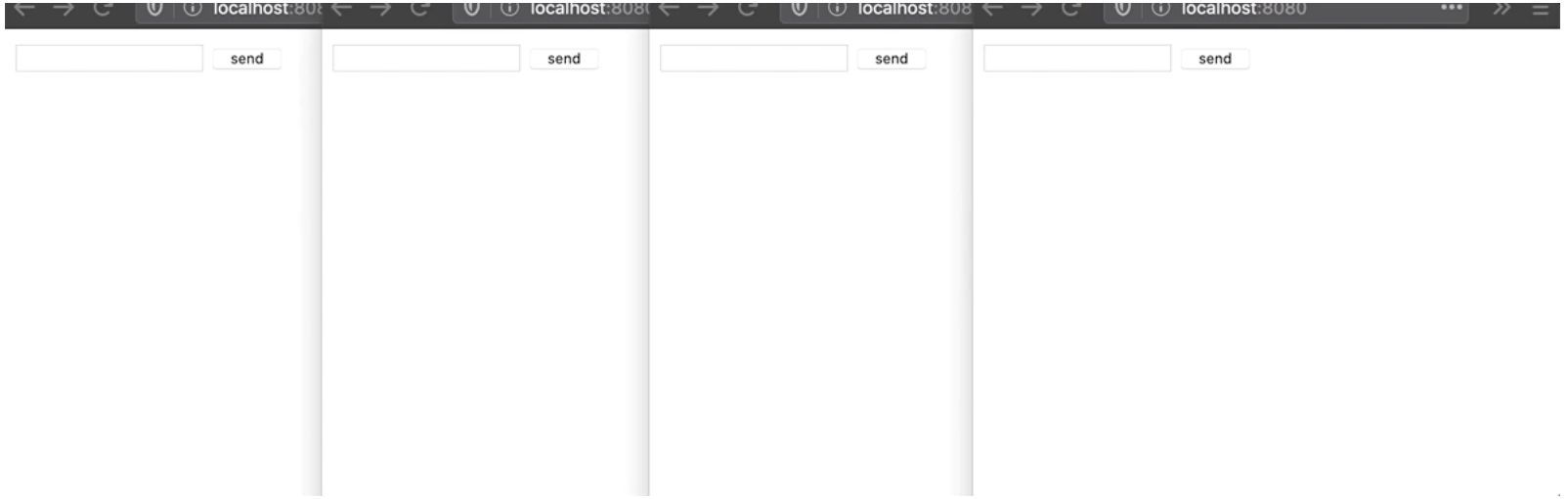
プログラム例
Ring with supervisor and re-start

```
def make_ractor r, i
  Ractor.new r, i do |r, i|
    loop do
      msg = Ractor.recv
      raise if /e/ =~ msg
      r.send msg + "r#{i}"
    end
  end
end
r = Ractor.current
rs = (1..10).map{|i|
  r = make_ractor(r, i)
}
```

```
msg = 'e0' # error causing message
begin
  r.send msg
  p Ractor.select(*rs, Ractor.current)
rescue Ractor::RemoteError
  r = rs[-1] = make_ractor(rs[-2], rs.size-1)
  msg = 'x0'
  retry
end
#=> [:recv, "x0r9r9r8r7r6r5r4r3r2r1"]
```

プログラム例
サーバ by niku さん

https://gist.github.com/niku/c19da11edf0b97470af27844b44d12fa



movie も by niku さん

niku さんのコメント

- Ractor と書くの慣れてきました。typoしなくなってきた。
- やりたいことのAPIもう揃っていてすてきです。特に Ractor.yield は親に何か送るという頻出パターンを親の ractor オブジェクトなしにできるところいいなとおもいました。

実装

- GVLつき並列に動かない版（APIが試せます）
  - https://github.com/ko1/ruby/tree/ractor
- GVLなし並列に動く版
  - 鋭意開発中
  - ちょっと動く
  - GC でバグる
  - 排他制御が凄く足りない
  - ほんと GVL 偉大…

積み残し作業（たくさん）

- APIの検討
  - 本当に今のでいいの？
  - require/autoload どうすんの？
  - なんかもっといろいろあったきがする
- 実装
  - 複製・移動の書き直し（現在は手抜き）
  - 大量の排他制御漏れの fix（MRI全コード見直し）
  - スレッドセーフでない C-extension どうすんの？
  - 高速化
    - オブジェクト生成のたびにインタプリタロックするのしんどいのでなんとか
    - rb_xxx 系の C API なんとかせんといけないような…

FAQ
Ractorの生成コストは？

- 今は Thread.new と同じくらい（native thread 作る）
- ので、バカバカ作ると重い、数の制限がある（数万）
- 今後、Thead 自体を M:N モデルにしていって軽量化したい
  → Fiber.new と同程度にしたい
  （Ruby 3.1 とかで？）

```
# 1万個生成、合流コスト（ただし Ruby は-O0）
 user     system      total        real
process      0.528566   3.191144  30.444054 ( 30.113236)
ractor       0.709152   0.842714   1.551866 (  1.061107)
thread       0.542611   0.918953   1.461564 (  0.966229)
fiber        0.143048   0.027120   0.170168 (  0.170188)
```

おわり

Ruby 3 をお楽しみに！

ruby-jp Slack の #concurrency でご意見募集しております。
夜中に（さぎょいぷ的） zoom やってることもあります。