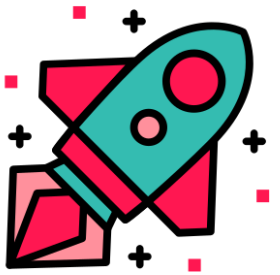
concurrent-ruby
v1.1

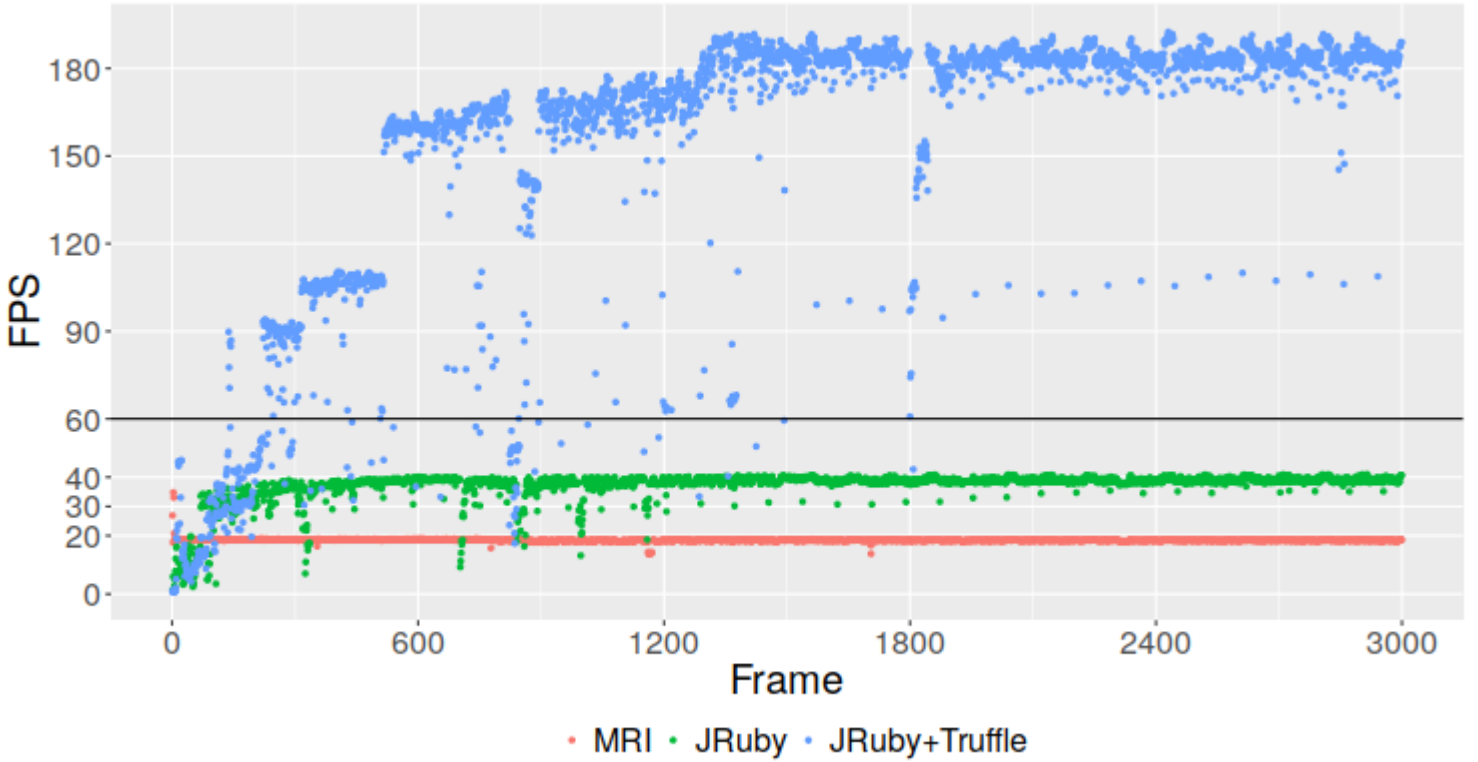
Petr Chalupa

- Principal Member of Technical Staff at
- Oracle Labs – VM research group
- **TruffleRuby**
- **Lead maintainer of concurrent-ruby**



TRUFFLE
RUBY

TruffleRuby performance



Source: https://eregon.me/blog/2016/11/28/optcarrot.html

Agenda

Short concurrent-ruby introduction
Contributors wanted!
New Promises framework

concurrent-ruby

concurrent-ruby

- RubyGem (since 2013)
  - unopinionated toolbox
  - low level abstractions
  - high level abstractions
- No dependencies
- Backward compatibility
- Extensive documentation
- Ruby implementation independent
  - CRuby, JRuby, TruffleRuby, and (Rubinius)
- Open source, MIT

Concurrency in Ruby

CRuby has **GIL** $\Longrightarrow$ parallelism ✗

JRuby, TruffleRuby and Rubinius have no ~~**GIL**~~ $\Longrightarrow$ parallelism ✓

- Stdlib: `Thread`, `Queue`, `Mutex`, `Monitor`, `ConditionVariable`
- Implementation specific:
  - `JRuby::Synchronized`, Java interoperation
  - `Rubinius::Channel`, `Rubinius.lock`, etc.
- No volatile variables (java meaning)
- `fork`ing $\Longrightarrow$ memory consuming, incompatible with JRuby
- **TL;DR** Just stdlib tools are hard to use

Abstractions

High-level:

- Async
- TimerTask
- Promises
- Executors
- Channel
- Agent
- Actor
- TVar (Software Transactional Memory)

Atomics:

- AtomicFixnum
- AtomicBoolean
- AtomicReference

Synchronization primitives:

- CountDownLatch
- Event
- Condition
- Semaphore

Other:

- ThreadLocalVar
- MVar
- Exchanger
- Delay
- LazyRegister
- ThreadPools

Who uses it?

- Over 4.2K Github stars
- 313 gems directly depedend on concurrent-ruby
  - sucker_punch
  - sidekiq
  - rails
  - hanami-*
  - dry-*
  - google-cloud-*

Who uses it?

- directly – 313 gems, 0.2% of all gems

Who uses it?

- directly – 313 gems, 0.2% of all gems
- 1 indirection – 10 933 gems, **7.0%** of all gems

Who uses it?

- directly – 313 gems, 0.2% of all gems
- 1 indirection – 10 933 gems, **7.0%** of all gems
- 2 indirections – 31 354 gems, **20.2%** of all gems

Who uses it?

- directly – 313 gems, 0.2% of all gems
- 1 indirection – 10 933 gems, **7.0%** of all gems
- 2 indirections – 31 354 gems, **20.2%** of all gems
- 3 indirections – 34 386 gems, 22.1% of all gems
- ...
- 8 indirections – 34 625 gems, 22.3% of all gems

Who uses it?

- directly – 313 gems, 0.2% of all gems
- 1 indirection – 10 933 gems, **7.0%** of all gems
- 2 indirections – 31 354 gems, **20.2%** of all gems
- 3 indirections – 34 386 gems, 22.1% of all gems
- ...
- 8 indirections – 34 625 gems, 22.3% of all gems

Downloads

- It has **79 566 504** downloads

Who uses it?

- directly – 313 gems, 0.2% of all gems
- 1 indirection – 10 933 gems, **7.0%** of all gems
- 2 indirections – 31 354 gems, **20.2%** of all gems
- 3 indirections – 34 386 gems, 22.1% of all gems
- ...
- 8 indirections – 34 625 gems, 22.3% of all gems

Downloads

- It has **79 566 504** downloads
- It is **60th** most downloaded gem out of 155 008 existing gems
  - The percentile rank is 99.96%

Help

- **Contributors and Mainers wanted!**
- It's mostly just me
- What you get?
  - An impressive line to your CV
  - You can learn something
    - New
    - Important
    - Which not many people know about

How to contribute?

- Fix / write documentation
- Answer user questions
- Look up existing issue with tag `looking-for-contributor`
- Contribute something completely new
- Become a maintainer

Promises

Promises

- New framework
- Integrates in one framework features of older abstactions:
  - `Future` and `dataflow`
  - `Promise`
  - `IVar`
  - `Event`
  - `Delay`
  - `TimerTask`
- Familiar names based on JS promises with extra features

Advantages

- Uses synchronisation layer from `concurrent-ruby` providing volatile variables with CAS operations
  - No `Mutex` $\Longrightarrow$ faster
  - Promises are non-blocking and lock-free
    (With the exception of obviously blocking operations like `#wait`, `#value`)
- Integrates with other concurrency abstractions: `Actor`, `Channel`, `ProcessingActor`
  - Keeps up with your growing needs

Basics

Main classes

`Future` – a value which may not yet be available

`Future#state`

- `:pending`, `#pending?`
- `:fulfilled`, `#fulfilled?`, `#resolved?`
- `:rejected`, `#rejected?`, `#resolved?`

Reading result

- not applicable, blocks
- `#value`, `#wait`
- `#reason`, `#wait`

`Event` – an event which may not yet happen, has no value or failure

`Event#state`

- `:pending`, `#pending?`
- `:resolved`, `#resolved?`

Resolution

- not applicable, blocks
- `#wait`

Example

Using REST API asynchronously

Example

- Background requests to REST APIs
- Queering repository for open PRs
- Checking status of PRs on Travis and Appveyor CIs

Asynchronous execution

Putting work on background to communicate with user fast

```
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×000002 pending>

ui 'Getting pull-requests'                  # ⇒ "Getting pull-requests"
prs.value                                   # ⇒ ["#845", "#953"]
ui 'Getting done'                           # ⇒ "Getting done"
```

`#ui` further omitted

Asynchronous execution

Putting work on background to communicate with user fast

```
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×000002 pending>

ui 'Getting pull-requests'                  # ⇒ "Getting pull-requests"
prs.value                                   # ⇒ ["#845", "#953"]
ui 'Getting done'                           # ⇒ "Getting done"
```

`#ui` further omitted

`Future([#1, #2])`

Chaining

Get status of all PRs

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000003 pending>

statuses.wait
# ⇒ #<Concurrent::Promises::Future:0×000003 fulfilled>
```

Chaining

Check all statuses are green

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000004 pending>

statuses.value                              # ⇒ [true, true]
```

Chaining

Rather check there was no error

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000005 pending>

statuses.value!                          # ⇒ [true, true]
```

Raised in the main thread.

Chaining

Rather check there was no error

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000006 pending>

statuses.value! rescue $!
# ⇒ #<RuntimeError: no-internet when checking #845>
```

Does not even try to get status of second pr

Parallel

Parallel get_travis_status sending

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map do |pr|
    Concurrent::Promises.future(pr) { |pr| get_travis_status pr }
  end
end
# ⇒ #<Concurrent::Promises::Future:0×000007 pending>

statuses.value! rescue $!
# ⇒ [#<Concurrent::Promises::Future:0×000008 pending>,
#     #<Concurrent::Promises::Future:0×000009 pending>]
```

Missing results of get_travis_status though.

Parallel

Parallel get_travis_status sending

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map do |pr|
    Concurrent::Promises.future(pr) { |pr| get_travis_status pr }
  end
end
# ⇒ #<Concurrent::Promises::Future:0×000007 pending>

statuses.value! rescue $!
# ⇒ [#<Concurrent::Promises::Future:0×000008 pending>,
#     #<Concurrent::Promises::Future:0×000009 pending>]
```

Missing results of get_travis_status though.

```
Future([ Future(#1.status), Future(#2.status) ])
```

Parallel

Parallel get_travis_status sending

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map do |pr|
    Concurrent::Promises.future(pr) { |pr| get_travis_status pr }
  end
end
# ⇒ #<Concurrent::Promises::Future:0×000007 pending>

statuses.value! rescue $!
# ⇒ [#<Concurrent::Promises::Future:0×000008 pending>,
#     #<Concurrent::Promises::Future:0×000009 pending>]
```

Missing results of get_travis_status though.

```
statuses.value!.map(&:value!) rescue $!
# ⇒ [true, true]
```

```
Future([ Future(#1.status), Future(#2.status) ])
```

Side notes

FactoryMethods

`Concurrent::Promises` can be annoying, let's include the factory methods

```
include Concurrent::Promises::FactoryMethods
# ⇒ Object
statuses = future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  prs.map do |pr|
    future(pr) { |pr| get_travis_status pr }
  end
end
# ⇒ #<Concurrent::Promises::Future:0×00000a pending>

statuses.value!.map(&:value!) rescue $!
# ⇒ [true, true]
```

Passing arguments

Using captured local variables is not thread-safe

```
prs.value.map do |pr|
  # using captured variable is unsafe
  future { get_travis_status pr }
end
```

It should be

```
prs.value.map do |pr|
  future(pr) { |pr| get_travis_status pr }
end
# ⇒ [#<Concurrent::Promises::Future:0×00000b pending>,
#     #<Concurrent::Promises::Future:0×00000c pending>]
```

same as

```
Thread.new(1) { |i| i.succ }.value       # ⇒ 2
```

Back to Example

Zip

Aggregate promises together

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  travis_statuses = prs.map do |pr|
    Concurrent::Promises.future(pr) { |pr| get_travis_status pr }
  end
  Concurrent::Promises.zip(*travis_statuses)
end
# ⇒ #<Concurrent::Promises::Future:0×00000d pending>

statuses.value!.value! rescue $!          # ⇒ [true, true]
```

Without `#zip` it was `statuses.value!.map(&:value!) rescue $!`.

Zip

Aggregate promises together

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  travis_statuses = prs.map do |pr|
    Concurrent::Promises.future(pr) { |pr| get_travis_status pr }
  end
  Concurrent::Promises.zip(*travis_statuses)
end
# ⇒ #<Concurrent::Promises::Future:0×00000d pending>

statuses.value!.value! rescue $!          # ⇒ [true, true]
```

Without `#zip` it was `statuses.value!.map(&:value!) rescue $!`.

`Future( Future( [#1.status, #2.status] ) )`

Zip

Simplify with different factory method

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0x00000e pending>

statuses.value!.value! rescue $!          # ⇒ [true, true]
```

Flat

Get result of inner future

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0x00000f pending>

statuses.value! rescue $!                    # ⇒ [true, true]
```

Flat

Get result of inner future

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0x00000f pending>

statuses.value! rescue $!                    # ⇒ [true, true]
```

```
Future( [#1.status, #2.status ] )
```

Flat

Avoid calling blocking methods!

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }.
    value! # BAD!
end

statuses.value! rescue $!                    # ⇒ [true, true]
```

Zip

Revisiting failure of zipped futures, we are now getting partial results

```
statuses = Concurrent::Promises.future do
  get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0×000010 pending>

statuses.value! rescue $!
# ⇒ #<RuntimeError: no-internet when checking #953>
statuses.rejected?                          # ⇒ true
statuses.value                              # ⇒ [true, nil]
statuses.reason
# ⇒ [nil, #<RuntimeError: no-internet when checking #953>]
```

Branching

It's not limited to single chain

```
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×000011 pending>
travis = prs.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0×000012 pending>
appveyor = prs.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_appveyor_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0×000013 pending>

travis.value! rescue $!                  # ⇒ [true, true]
appveyor.value! rescue $!                # ⇒ [true, true]
```

Branching

It's not limited to single chain

```
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×000011 pending>
travis = prs.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0×000012 pending>
appveyor = prs.then do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_appveyor_status pr }
end.flat
# ⇒ #<Concurrent::Promises::Future:0×000013 pending>

travis.value! rescue $!                  # ⇒ [true, true]
appveyor.value! rescue $!                # ⇒ [true, true]
```

travis ← prs → appveyor

Branching

It's not limited to single chain

```
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×000014 pending>
travis = prs.then_flat do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_travis_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000015 pending>
appveyor = prs.then_flat do |prs|
  Concurrent::Promises.zip_futures_over(prs) { |pr| get_appveyor_status pr }
end
# ⇒ #<Concurrent::Promises::Future:0×000016 pending>

travis.value! rescue $!                  # ⇒ [true, true]
appveyor.value! rescue $!                # ⇒ [true, true]

(travis & appveyor).value                # ⇒ [[true, true], [true, true]]
```

travis ← prs → appveyor

Advanced features

Lazy

Thread-safe lazy initialization

```
config = Concurrent::Promises.delay { load_configuration }
# ⇒ #<Concurrent::Promises::Future:0×000017 pending>
```

Does not execute until `#value`, `#reason`, or `#wait` is called on it.

```
sleep_a_bit
config.pending?                                  # ⇒ true
config.value
# ⇒ {:user⇒"Adam", :password⇒"Vaidyaigmyb1"}
```

Lazy

Thread-safe lazy initialization

```
config = Concurrent::Promises.delay { load_configuration }
# ⇒ #<Concurrent::Promises::Future:0×000017 pending>
```

Does not execute until `#value`, `#reason`, or `#wait` is called on it.

```
sleep_a_bit
config.pending?                                  # ⇒ true
config.value
# ⇒ {:user⇒"Adam", :password⇒"Vaidyaigmyb1"}
```

Can be anywhere in the chain

```
head = Concurrent::Promises.future { 'evaluated' }
chain = head.delay.then { |s| s + ' delayed append' }
sleep_a_bit; [head.state, chain.state, chain.value]
# ⇒ [:fulfilled, :pending, "evaluated delayed append"]
```

Schedule

Run asynchronous task in X seconds or on given time

```
Concurrent::Promises.schedule(60) do
  ui 'A message for user'
end
# ⇒ #<Concurrent::Promises::Future:0×000018 pending>
```

Schedule

Run asynchronous task in X seconds or on given time

```
Concurrent::Promises.schedule(60) do
  ui 'A message for user'
end
# ⇒ #<Concurrent::Promises::Future:0×000018 pending>
```

Can be anywhere in the chain

```
Concurrent::Promises.
  future { 'evaluated' }.
  schedule(1).
  then { |s| s + ' append delayed by 1s' }
```

Throttling

Enforce concurrency limit on certain tasks

```
max_one = Concurrent::Throttle.new 1
# ⇒ #<Concurrent::Throttle:0×000019 limit:1 can_run:1>
prs = Concurrent::Promises.future do
 get_open_prs 'ruby-concurrency', 'concurrent-ruby'
end
# ⇒ #<Concurrent::Promises::Future:0×00001a pending>
appveyor = prs.then_flat do |prs|
  Concurrent::Promises.zip(*prs.map do |pr|
      max_one.throttled_future(pr) { |pr| get_appveyor_status pr }
    end)
end
# ⇒ #<Concurrent::Promises::Future:0×00001b pending>

appveyor.value! rescue $!                # ⇒ [true, true]
```

Actors

Use when a state has to be maintained, e.g. DB connections

```
data = Array.new(4) { |i| '*' * i }      # ⇒ ["", "*", "**", "***"]
DB = Concurrent::Actor::Utils::Pool.spawn!('db', size = 2) do |index|
  # DB connection constructor
  Concurrent::Actor::Utils::AdHoc.spawn!("connection-#{index}") do
    → message { data[message] } # query a DB
  end
end

concurrent_jobs = 4.times.map do |index|
  # limited concurrency to 2 for asking the DB
  DB.ask(index).then(&:size)
end

Concurrent::Promises.zip(*concurrent_jobs).value!
# ⇒ [0, 1, 2, 3]
```

Even more

- Channels
- Backpressure
- Process
- Cancellation
- ProcessingActor
- ...

Conclusion

Conclusion

- Simple usage
- Integration between different abstractions
- Implementation independent
  - You can start with MRI and scale on JRuby
- Everything runs on a thread pool
  - Not limited by number of threads
- Fast

Conclusion

- Simple usage
- Integration between different abstractions
- Implementation independent
  - You can start with MRI and scale on JRuby
- Everything runs on a thread pool
  - Not limited by number of threads
- Fast

Contributors and Maintainers wanted!
Get your line in the CV :)

Thanks!

Questions
Answers
Links

- concurrent-ruby.com
- twitter: pitr_ch, github: pitr-ch
- Ruby snippets evaluated with md-ruby-eval gem

Find me later ...