Concurrent Web Crawling in Ruby with Async

By Joseph Izaguirre

2025-05-28

Of all the languages one could use to build a web crawler, Ruby is not one that immediately stands out. Web crawling is I/O heavy; it involves downloading dozens, hundreds, or even thousands of web pages. The proven solution to I/O heavy workloads is an event-driven, non-blocking architecture, something usually associated with languages like Go, JavaScript or Elixir. I'd like to put forth Ruby as a great language for handling I/O bound workloads, especially when paired with the Async library. I'm sure Go, JavaScript and Elixir are fine languages; however, in my opinion, they don't quite match Ruby in terms of readability and expressiveness. Let's first build

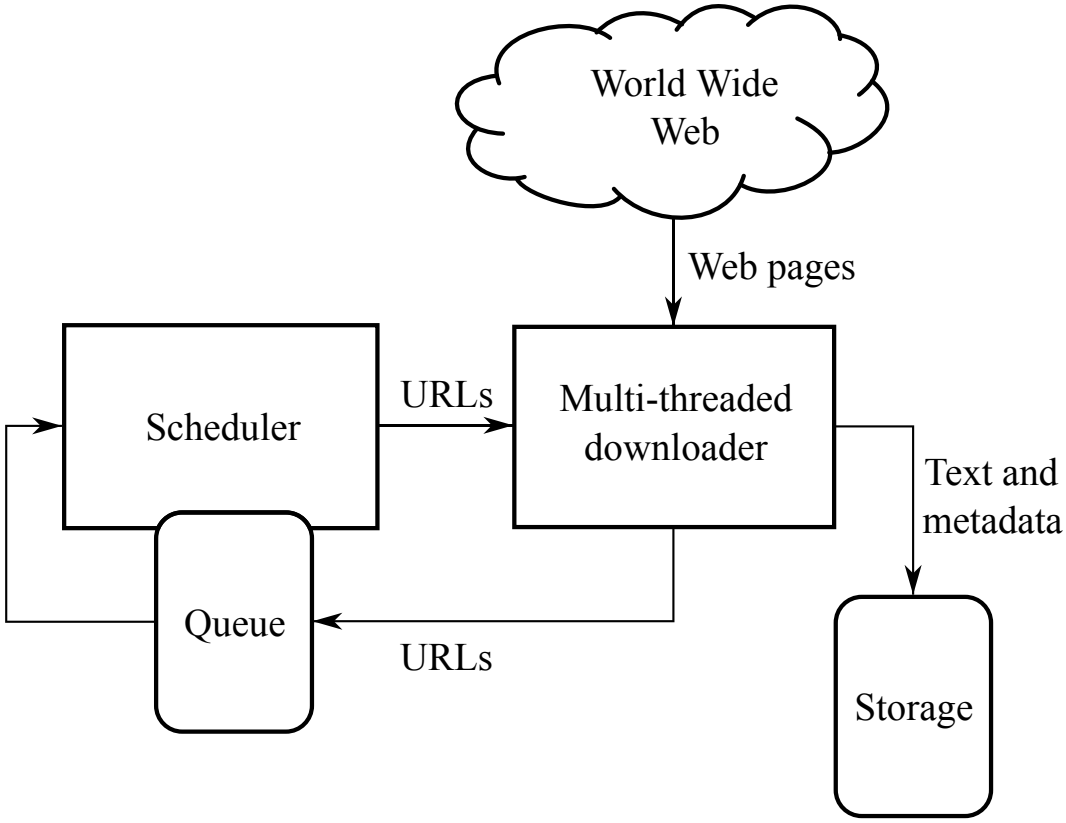
a web crawler without concurrency, downloading pages one at a time. Then, I will show how to add concurrency in a lightweight manner and with minimal changes to the code.

Before we get started, I want to clarify the intent of this article; the code here is intended mainly to demonstrate how easy it is to add event-driven, non-blocking I/O to existing Ruby code. This is not intended to be an exhaustive resource for building a web crawler. In particular, the path normalization and link scraping parts of the code have been greatly simplified, so don't take those as gospel. You're probably not going to want to store the results of web crawling in a giant hash, either.

Fundamentals of Web Crawling

You can boil the process of crawling a domain down to a few steps:

1. Pick a page to begin the crawling process, usually the root path. This is known as the **seed**.
2. Download that page and parse the HTML.
3. Parse all of the anchor tags on the page. Filter out the tags that aren't HTTP-based URLs, point to an external domain, or have already been scraped.
4. For each of those links, repeat steps 2 and 3.



Architecture of a web crawler, from Wikipedia. Note the Multi-threaded downloader, to be implemented later in the article.

Let's use recursion to implement our web crawler. Here is a naive implementation:

```
require "open-uri"
require "nokogiri"
require "set"

class WebCrawler
  attr_reader :results

  def initialize(seed_url)
    @seed_url = URI.parse(seed_url)
    @results = {}
    @discovered = Set.new
  end

  def crawl
    fetch(@seed_url.request_uri)
    @results
  end
```

```ruby
private

def fetch(raw_path)
  path = normalize(raw_path)
  return unless @discovered.add?(path)

  full_url = build_url(path)
  html = URI.open(full_url, "User-Agent" => "SpiderBot", &:read)

  @results[path] = html
  scrape_links(html).each { |link| fetch(link) }
rescue OpenURI::HTTPError, SocketError => e
  warn "⚠️  Failed to fetch #{full_url}: #{e.message}"
end


# This part of the code is not critical to this article, so I've blurred it out.
def build_url(path)
  "#{@seed_url.scheme}://#{@seed_url.host}#{path}"
end

def normalize(raw_path)
  raw_path = "/#{raw_path}" unless raw_path.start_with?("/")

  uri = URI(raw_path).normalize
  uri.fragment = nil

  path = uri.path.empty? ? "/" : uri.path
  path += "?#{uri.query}" if uri.query
  path
end

def scrape_links(html)
  return [] if html.nil? || html.strip.empty?

  Nokogiri::HTML(html).css("a[href]").filter_map do |a|
    href = a["href"].to_s.strip
    next if href.empty? || href.match?(/\A(?:mailto:|javascript:|#)/)

    begin
      uri = URI.parse(href)
      if uri.relative?
        href
```

```
        elsif uri.host == @seed_url.host
          uri.request_uri
        end
      rescue URI::InvalidURIError => e
        warn "⚠️  Invalid URL encountered: #{href.inspect} (#{e.message})"
        nil
      end
    end
  end
end

# Our crawler can be used like this:
spider = WebCrawler.new("https://losangelesaiapps.com/")
results = spider.crawl
puts results
```

This works, but it's not very fast; it scrapes 13 pages in about 3 seconds. There are two big bottlenecks here: we're downloading each page one at a time, and we're making a new TLS handshake for each request. This is known as an **I/O bound** workload. Let's add concurrent downloads using an event-driven architecture, and then we'll solve the TLS handshake problem.

Adding Concurrency with Fibers and Async

Ruby comes with 4 concurrency constructs: Processes, Ractors, Threads, and Fibers. Every process consists of at least 1 ractor, every ractor contains at least 1 thread, and every thread consists of at least 1 fiber. Processes are the heaviest in terms of memory use, and Fibers are the lightest. For CPU bound workloads, reach for Processes or Ractors; they can run tasks on multiple CPUs at the same time. For I/O bound workloads, such as our web crawler, Threads and Fibers are the best primitives to use.

Let's use Fibers. Fibers are more lightweight than Threads, and they aren't preempted by the VM. They are non-blocking by default, delegating blocking and waking to a scheduler. Ruby doesn't come with a scheduler built in, so let's use the one that

comes with the Async gem. Async provides two things that we will need to complete our web crawler: a Fiber scheduler and a simple DSL for creating and running asynchronous tasks.

Let's improve our humble web crawler by implementing concurrency with Async:

```
require "open-uri"
require "nokogiri"
require "set"
+ require "async"

class WebCrawler
  attr_reader :results

  def initialize(seed_url)
    @seed_url = URI.parse(seed_url)
    @results = {}
    @discovered = Set.new
  end

  def crawl
+     Sync do
      fetch(@seed_url.request_uri)
      @results
+     end
  end

  private

  def fetch(raw_path)
+     Async do
      path = normalize(raw_path)
      next unless @discovered.add?(path)

      full_url = build_url(path)
      html = URI.open(full_url, "User-Agent" => "SpiderBot", &:read)

      @results[path] = html
      # Each call to fetch is itself generating asynchronous child tasks!
      scrape_links(html).each { |link| fetch(link) }
    rescue OpenURI::HTTPError, SocketError => e
```

```
      warn "⚠️  Failed to fetch #{full_url}: #{e.message}"
+     end
  end

  # ...

end
```

We've wrapped our top-level #crawl function in a **Sync** block and the #fetch function in an **Async** block. The Sync method creates a Scheduler and an Event loop, if one doesn't already exist. The Scheduler intercepts blocking operations and redirects the corresponding Fiber to the Event loop. The event loop checks for a pending event, in our case a network response, and wakes the Fiber up when it's ready to continue. The Sync block also creates a root Task that acts as the parent of all of the nested child Tasks created within it. Within a Task, code is executed sequentially, as in a normal Ruby program.

The Async block wraps the entirety of our #fetch function, which makes each call to #fetch an asynchronous Task that runs in the background. The fetch function calls itself for each scraped link; this creates a hierarchy of Tasks, with each web page acting as a "parent" for each of the links scraped on that page.

The hierarchy of Tasks, annotated by the path of the URL for that page. This traces the path taken by our web crawler.

These small changes have dropped the time to crawl our website from 3 seconds to about half a second! Unfortunately, we've created a subtle problem. Look again at the figure above: one task can create two tasks, those two tasks can create four tasks, and so on. We're crawling websites at an exponential rate, potentially overwhelming the web servers used to host them! We need to use a **Semaphore** to limit the number of HTTP requests that we can make at one time.

```
require "open-uri"
require "nokogiri"
require "set"
require "async"
+ require "async/semaphore"

class WebCrawler
  attr_reader :results

+   def initialize(seed_url, download_concurrency: 3)
    @seed_url = URI.parse(seed_url)
    @results = {}
    @discovered = Set.new
+     @semaphore = Async::Semaphore.new(download_concurrency)
  end

  def crawl
    Sync do
      fetch(@seed_url.request_uri)
      @results
    end
  end

  private

  def fetch(raw_path)
    Async do
      path = normalize(raw_path)
      next unless @discovered.add?(path)
```

```
      full_url = build_url(path)
+       html = @semaphore.async do
        URI.open(full_url, "User-Agent" => "SpiderBot", &:read)
+       end.wait

      @results[path] = html
      scrape_links(html).each { |link| fetch(link) }
    rescue OpenURI::HTTPError, SocketError => e
      warn "⚠️  Failed to fetch #{full_url}: #{e.message}"
    end
  end
  # ...
end
```

Our Semaphore limits access to the URI.open calls to 3 at a time, by default. This *does* slow down our web crawler, but it's a good practice. We want our crawler to be polite, not barging in and taking down servers!

We can do better; each call to URI.open establishes a new HTTP connection, complete with its own TLS handshake. These take a surprising amount of time (they contributed a lot to the bottleneck in our original version). Let's pull in Async::HTTP, another tool from the Async ecosystem, to establish a persistent HTTP connection to the server that we're crawling:

```
- require "open-uri"
require "nokogiri"
require "set"
require "async"
require "async/semaphore"
+ require "async/http"

class WebCrawler
  attr_reader :results

  def initialize(seed_url, download_concurrency: 3)
+     @seed     = Async::HTTP::Endpoint.parse(seed_url)
+     @client   = Async::HTTP::Client.new(@seed)
    @results  = {}
    @discovered = Set.new
```

```
    @semaphore = Async::Semaphore.new(download_concurrency)
  end

  def crawl
    Sync do
      fetch(@seed.path)
      @results
    ensure
      @client.close
    end
  end

  private

  def fetch(raw_path)
    Async do
      path = normalize(raw_path)
      next unless @discovered.add?(path)

      html = @semaphore.async do
+         @client.get(path, { "User-Agent": "SpiderBot" }).read
      end.wait

      @results[path] = html

      scrape_links(html).each { |link| fetch(link) }
    end
  end

  # ...
end
```

Async::HTTP::Client automatically maintains a persistent HTTP connection and reuses it for all subsequent requests, requiring only one TLS handshake. Each resource then streams across that connection, with protocols like HTTP2 and HTTP3 being purpose-built to multiplex requests.

There's one last issue we should fix. If an uncaught error is thrown during the execution of #crawl, we want to ensure that all remaining Tasks are stopped before moving on. We don't want any tasks to leak from the use of our web crawler. We need

to use a **Barrier** to hold all of the tasks generated by the recursive calls to fetch. **Barriers** and **Semaphores** are designed to work together. A final implementation might look like this:

```
require "nokogiri"
require "set"
require "async"
require "async/semaphore"
require "async/http"
+ require 'async/barrier'

class WebCrawler
  attr_reader :results

  def initialize(seed_url, download_concurrency: 3)
    @seed = Async::HTTP::Endpoint.parse(seed_url)
    @client = Async::HTTP::Client.new(@seed)
    @results = {}
    @discovered = Set.new
+     @barrier = Async::Barrier.new
+     @semaphore = Async::Semaphore.new(download_concurrency, parent: @barrier)
  end

  def crawl
    Sync do
      fetch(@seed.path)
+       @barrier.wait
      @results
    end
  ensure
+     @barrier.stop
    @client.close
  end

  private

  def fetch(raw_path)
+     @barrier.async do
      path = normalize(raw_path)
      next unless @discovered.add?(path)

      html = @semaphore.async do
```

```
        @client.get(path, { "User-Agent": "SpiderBot" }).read
      end.wait

      @results[path] = html

      scrape_links(html).each { |link| fetch(link) }
    end
  end
end
```

We have made the Barrier the parent of the Semaphore, so that all tasks executed by the semaphore are added to the Barrier. We have also replaced the bare Async call to **@barrier.async** to add child Tasks to our Barrier.

Be Polite

Our humble Web Crawler is humble no more! As a great man once said, *With Great Power comes Great Responsibility*. Your web crawler should be polite when requesting resources from someone else's server:

- Respect robots.txt, including the Crawl-delay parameter.
- Identify yourself clearly with your User-Agent
- Be careful with the amount of concurrent requests to a single domain. Scrapy, a popular open-source crawling library, defaults to 8 simultaneous requests, for reference.

More information can be found here.

Concurrency in Ruby is a Delight

It’s tempting for a Ruby shop to evaluate other languages when event-driven non-blocking I/O is a requirement. But Ruby is great at solving these types of problems, especially when paired with Async. Ruby methods are colorless, so implementing concurrency doesn’t infect your codebase with the Async/Await virus, and there’s no Callback Hell. Just clean, expressive Ruby.